Graphics Part II: Code Tutorial for Week 8

Dr P. Perakis

Preparation

We have provided a fully working code for the loading of a terrain and a TerrainNode class to render a Terrain with texture in the `Exercise_06_1.zip` file. We have added a Camera class to our framework and a handler for navigation keys to fly around the terrain. 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. You will also need to use the files provided in the `Exercise_08_Files.zip` file.

Introduction

This week, we will update the TerrainNode class to your framework to incorporate blending of terrain textures. and incorporating an updated shader for this purpose. As before, the way I suggest to do it here, is just a suggestion. Feel free to implement it in any way you wish.

The files provided in `Exercise_08_Files.zip` file are as follows:

TerrainNode.h and .cpp	The updated classes that represent the new terrain node.
DDSTextureLoader.h and .cpp	Texture loader for .dds files
terrainshader.hlsl	Updated terrain shader
lightdirt.dds, grass_2.dds, slope.dds, stone.dds, snow.dds, rock.dds, darkdirt.dds	Various textures for the terrain

Blending Terrain Textures

The first step in this tutorial showcases how we can blend textures to give a more realistic terrain.

Using this approach, we will apply multiple textures to each square in the grid and use a blend map to determine how much of each texture is displayed.

To do this, we first need to change the vertex format of the terrain since we will need two sets of texture coordinates – one to index into the blend map and one to index into the individual textures. The vertex format we are going to use (and is expected by the shader we have supplied) is:

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

This structure, the modified vertex layout description required, and the two additional methods described below are provided in `TerrainNode.h` and `.cpp`. The shaders that handle this are supplied in `terrainshader.hlsl`.

We calculated the `TexCoords` values when creating the terrain grid. For tiling textures:

```
//Tiling Texture
terrainVerts[inx].Tex.x = (float)(i % 2);
```

```
terrainVerts[inx].Tex.y = (float)(j % 2);
```

We now need to add the BlendMapTexCoords. When creating the vertices, the BlendMapTexCoords values for U and V should be the positions into the blend map which is stretched across the entire grid. For a stretched blend map:

```
//Stretched Blendmap
terrainVerts[inx].BTex.x = i / (float)_xnumVert;
terrainVerts[inx].BTex.y = j / (float)_znumVert;
```

The tiled textures are loaded using the method LoadTerrainTextures which we have supplied for making things easier for you, since this kind of code is not very well documented in the official documentation or in books.

This method loads five different textures into a Texture2DArray element.

```
wstring terrainTextureNames[5] = {
                                   L"Terrain\\lightdirt.dds",
                                   L"Terrain\\grass_2.dds",
                                   L"Terrain\\slope.dds",
                                   L"Terrain\\stone.dds",
                                   L"Terrain\\snow.dds"
                                   };
```

lightdirt.dds

grass_2.dds

slope.dds

stone.dds

snow.dds

We have provided a number of example textures that you can play with. The code we have provided expects these to be in a folder called Terrain that you have to add to your Visual Studio code folder (do not add them to the Visual Studio solution though).

You will see that the textures are .dds files. This is a format used by DirextX for some textures. The code we have provided uses the function CreateDDSTextureFromFileEx to load the textures. This function is provided in the code files DDSTextureLoader.h and DDSTextureLoader.cpp. These come from the DirectX Toolkit. We have provided them for you, and you should add them to your Visual Studio code folder and to the Visual Studio solution.

Creating the Blend Map

Now we need to create the blend map. This is supplied to the shader as another texture. We could create it using an art package using different values for r, g, b and a at each point to indicate how much of each of the overlaid textures are displayed. However, this will not work for procedurally generated terrain.

So, we have provided the starting point for code that will generate a blend map at run time. This is provided in the method GenerateBlendMap.

You will need to add code to calculate the appropriate r, g, b and a values in the range of [0, 255] for each texture in the array:

- A value of `0` for `r`, g, `b` and `a` at each position will mean that only the zeroth texture is used over the entire terrain (here the `lightdirt.dds` texture).
- The value in the `r` component determines how much of the first texture is blended in (here the `grass_2.dds` texture).
- The value in the g component determines how much of the second texture is blended in (here the `slope.dds` texture).
- The value in the `b` component determines how much of the third texture is blended in (here the `stone.dds` texture).
- Finally, the value in the `a` component determines how much of the fourth texture is blended in (here the `snow.dds` texture).

You should use the height values at each point in the grid and possibly the amount of slope to determine which textures to use – how you do this is entirely up to you based on the type of terrain you want to create.

In the provided code, as an example, we use the following normalized height levels, since height is also normalized:

```
float level0 = 0.35f;
float level1 = 0.50f;
float level2 = 0.70f;
float level3 = 0.80f;
```

and the following sample code for blending values:

```
if ((height <= level0))
{

}

if ((height > level0) && (height <= level1))
{
       r = 255;
}

if ((height > level1) && (height <= level2))
{
       r = 64;
       g = 255;
}
if ((height > level2) && (height <= level3))
{

       g = 128;
       b = 128;
}
if ((height > level3) && (height <= 1.00f))
{
       a = 255;
}
```

and of course you may try your own mixing.

Once the `r`, `g`, `b` and `a` values have been calculated, they are combined into one 32-bit value where the `a` component occupies the top 8 bits, the `b` component the next 8 bits, the `g` component the next 8 bits and the `r` component the lowest 8 bits.

```
DWORD mapValue = (a << 24) + (b << 16) + (g << 8) + r;
```

The five colors sampled from the corresponding textures are then sequentially interpolated to give the resulting color for a specific pixel using the r, g, b, and a weights from the blend map. This is done in the pixel shader in `terrainshader.hlsl`.

Once you have the blend map and the terrain array, these need to be passed to the shader before the terrain is rendered. The lines of code needed to do this are in `TerrainNode::Render()` method:

```
// Set the texture to be used by the pixel shader
//_deviceContext->PSSetShaderResources(0, 1, _texture.GetAddressOf());
_deviceContext->PSSetShaderResources(0, 1, _blendMapResourceView.GetAddressOf());
_deviceContext->PSSetShaderResources(1, 1, _texturesResourceView.GetAddressOf());
```

Don't forget to make the changes to the input assembler layout in BuildVertexLayout()

```
D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
	{ "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 }
};
```

and to the code that compiles the shaders `BuildShaders()` to take account of the new shader file name:

```
//Compile vertex shader
HRESULT hr = D3DCompileFromFile(L"terrainshader.hlsl",
	nullptr, D3D_COMPILE_STANDARD_FILE_INCLUDE,
	"VS", "vs_5_0",
	shaderCompileFlags, 0,
	_vertexShaderByteCode.GetAddressOf(),
	compilationMessages.GetAddressOf());
```

and

```
// Compile pixel shader
hr = D3DCompileFromFile(L"terrainshader.hlsl",
	nullptr, D3D_COMPILE_STANDARD_FILE_INCLUDE,
	"PS", "ps_5_0",
	shaderCompileFlags, 0,
	_pixelShaderByteCode.GetAddressOf(),
	compilationMessages.GetAddressOf());
```

Since we are going to use the pixel shader it will now need to access the constant buffer. So we need to add a line to our rendering code in your node classes to make the constant buffer visible to your pixel shader (additionally to the vertex shader we did already). The line to add is:

```
_deviceContext->VSSetConstantBuffers(0, 1, _constantBuffer.GetAddressOf());
_deviceContext->PSSetConstantBuffers(0, 1, _constantBuffer.GetAddressOf());
```

Using the Pixel Shader to Calculate the Blended Colour

The blending process is done in the pixel shader in `terrainshader.hlsl`.

- The five colors c0, c1, c2, c3, c4 from the five textures are at first sampled at a specific texture point (texel)
- The five colors are sequentially interpolated to give the resulting color for the specific pixel using the r, g, b, and a weights from the blend map.
- The resulting texture color has then to be combined with the shading color calculated from the lighting model.

The interpolated result is given by the formula:

c = (1-a)*((1-b)*((1-g)*((1-r)*c0 + r*c1) + g*c2) + b*c3) + a*c4, where r, g, b and a are normalized.

The proposed code for this is:

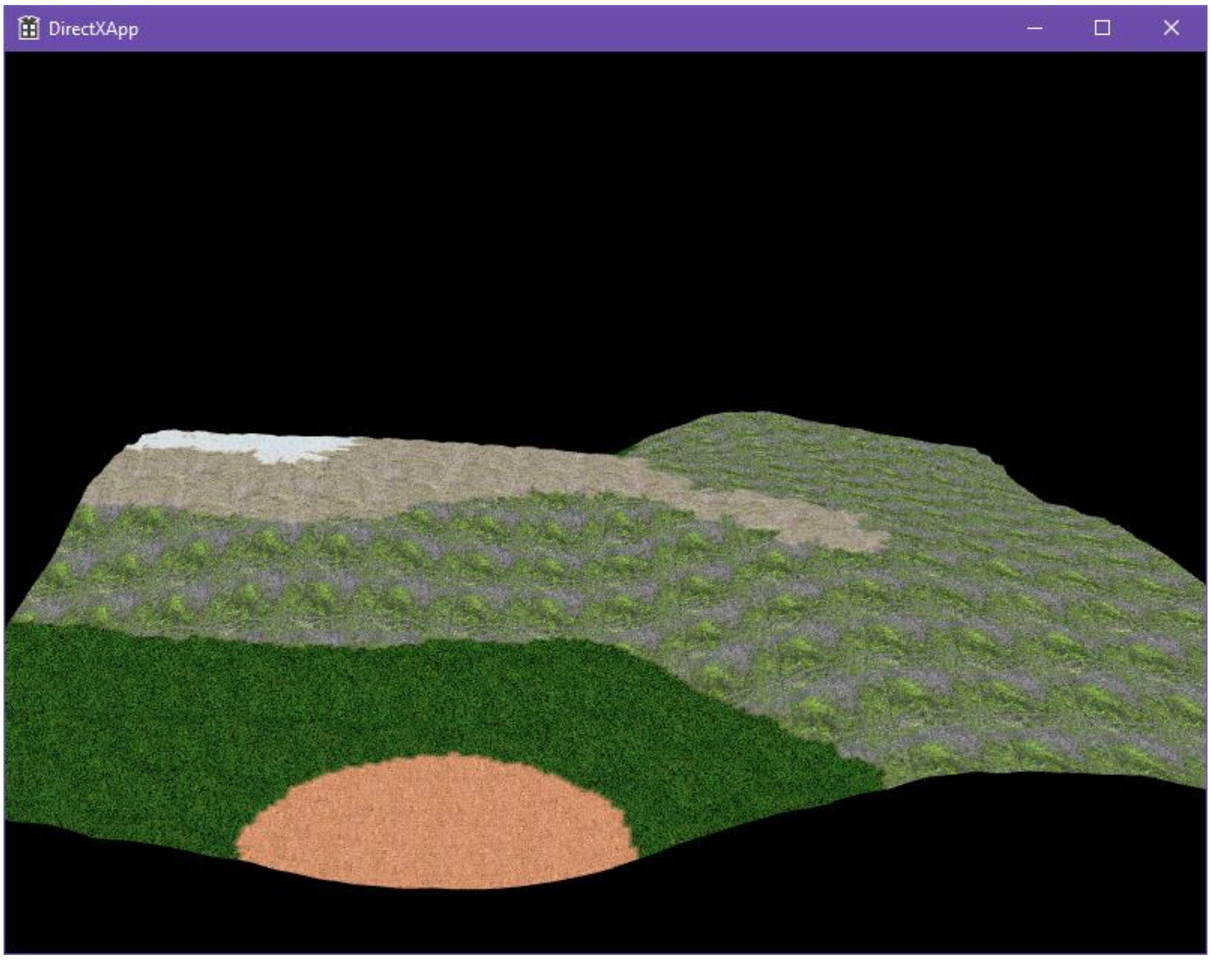
```
// The following code selects the appropriate pixel value from the
// texture array using the values from the blend map
float4 color;

// Sample layers in texture array.

float4 c0 = TexturesArray.Sample(ss, float3(input.TexCoord, 0.0f));
float4 c1 = TexturesArray.Sample(ss, float3(input.TexCoord, 1.0f));
float4 c2 = TexturesArray.Sample(ss, float3(input.TexCoord, 2.0f));
float4 c3 = TexturesArray.Sample(ss, float3(input.TexCoord, 3.0f));
float4 c4 = TexturesArray.Sample(ss, float3(input.TexCoord, 4.0f));

// Sample the blend map.
float4 t = BlendMap.Sample(ss, input.BlendMapTexCoord);

// Blend the layers on top of each other.
color = c0;
color = lerp(color, c1, t.r);
color = lerp(color, c2, t.g);
color = lerp(color, c3, t.b);
color = lerp(color, c4, t.a);

// Now adjust the colour using your calculated lighting values before returning
color = shade * color;
```

Of course, we can use the already calculated per-vertex shading color in the pixel shader:

```
//per-vertex shading from Vertex shader (Gouraud)
float4 shade = input.VColor;
```

The output of the executable

When you run your code, the result will depend on the values you put in your blend map. Using the values of the given example and without any light shading for the terrain, you see something like the following as a view.



Introducing a new Shader for the Terrain

For calculating a complete lighting model with ambient, diffuse and specular components we need the corresponding parameters to be passed in the constant buffer. Thus, the new constant buffer in the shader is:

```
cbuffer ConstantBuffer
{
    matrix completeTransformation;
    matrix worldTransformation;
    float4 cameraPosition;
    float4 lightVector;           // the light's vector
    float4 lightColor;            // the light's color
    float4 ambientColor;          // the ambient light's color
    float4 diffuseCoefficient;    // The diffuse reflection cooefficient
    float4 specularCoefficient;   // The specular reflection cooefficient
    float  shininess;             // The shininess factor
    float  opacity;               // The opacity (transparency) of the material. 0 = 
fully transparent, 1 = fully opaque
    float2 padding;
}
```

Instead of using a single material color we have added two reflection coefficients for the specular and diffuse light components which can result in nice colouring effects. This is particularly true if you want to use a colour for the diffuse reflection but keep white for the specular highlights. You may also add an ambient reflection coefficient to complete the colour components of a material.

These have become properties of the TerrainNode class:

```
struct CBUFFER
{
       Matrix        WorldViewProjection;
       Matrix        World;
       Vector4       CameraPosition;
       Vector4       LightVector;         // the light's vector
       Vector4       LightColor;          // the light's color
       Vector4       AmbientColor;        // the ambient light's color
       Vector4       DiffuseCoefficient;  // The diffuse reflection cooefficient
       Vector4       SpecularCoefficient; // The specular reflection cooefficient
       float         Shininess;           // The shininess factor
       float         Opacity;             // The opacity (transparency) of the 
material. 0 = fully transparent, 1 = fully opaque
       Vector2       Padding;
};
```

and are initialized in the corresponding constructor:

```
TerrainNode::TerrainNode(wstring name) : SceneNode(name)
{
       _name = name;
       //_materialColour = Vector4(1.0f, 1.0f, 1.0f, 1.0f);
       _diffuseColour = Vector4(1.0f, 1.0f, 1.0f, 1.0f);
       _specularColour = Vector4(1.0f, 1.0f, 1.0f, 1.0f);
       _shininess = 16.0f;
       _opacity = 1.0f;
       _useHeightMap = false;
}
```

One thing to be aware of is that your constant buffer should be a multiple of 16 bytes in size. This is not always needed but can cause problems on some video cards if it is not. I would suggest calculating up the size of your constant buffer and adding some pad if it is not a multiple of 16. Note that a Vector4 contains 4 floats, each of which is 4 bytes long, giving a total of 16 bytes.

Thus, in our case we need to add 2 more floats for padding since we have two single floats in :

```
       Vector2        Padding;
```

and similarly in the shader CBUFFER structure:

```
       float2         padding;
```

This field is not used, but it does ensure that the buffer is a multiple of 16 bytes in size.

The eye position should be the same as the one used by your camera. You can retrieve the current camera transformation from the `DirectXFramework` :

```
Matrix _viewTransformation = DirectXFramework::GetDXFramework()-
>GetViewTransformation();
Matrix _projectionTransformation = DirectXFramework::GetDXFramework()-
>GetProjectionTransformation();

Matrix completeTransformation = _worldTransformation * _viewTransformation *
_projectionTransformation;
```

The new light and material parameters and the transformations have to be passed to the cBuffer in TerrainNode::Render():

```
CBUFFER cBuffer;
//Set light
cBuffer.AmbientColor = Vector4(0.3f, 0.25f, 0.25f, 1.0f);
cBuffer.LightVector = Vector4(0.0f, 2.0f, -10.0f, 1.0f);
cBuffer.LightColor = Vector4(1.0f, 1.0f, 1.0f, 1.0f);
//Set material
cBuffer.DiffuseCoefficient = _diffuseColour;
cBuffer.SpecularCoefficient = _specularColour;
cBuffer.Shininess = _shininess;
cBuffer.Opacity = _opacity;
//Set transforms
cBuffer.WorldViewProjection = completeTransformation;
cBuffer.World = _worldTransformation;
```

Using the Vertex Shader to Calculate the Lighting Shade

So far, we have been doing the lighting calculations on a per-vertex basis in the vertex shader. This has been implementing the Gouraud shading model where colours are calculated at the vertices of polygons and then we let DirectX interpolate the colours between the vertices to give us the colour for each pixel.

Using the VertexShaderInput structure:

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

```
	float2 BlendMapTexCoord : TEXCOORD1;
};
```

We can calculate the per-vertex lighting shade:

```
PixelShaderInput VS(VertexShaderInput vin)
{
    PixelShaderInput output;
    float3 position = vin.Position;
    output.Position = mul(completeTransformation, float4(position, 1.0f));
    output.PositionWS = mul(worldTransformation, float4(position, 1.0f));
    output.NormalWS = float4(mul((float3x3)worldTransformation, vin.Normal), 0.0f);
    output.TexCoord = vin.TexCoord;
    output.BlendMapTexCoord = vin.BlendMapTexCoord;

    // calculate the diffuse light and add it to the ambient light
    //float4 vectorBackToLight = normalize(lightVector); // directional light
    float4 vectorBackToLight = normalize(lightVector - output.PositionWS); // point 
light

    float4 adjustedNormal = normalize(output.NormalWS);
    float diffuseBrightness = saturate(dot(adjustedNormal, vectorBackToLight));

    output.VColor = saturate(ambientColor + diffuseCoefficient * diffuseBrightness * 
lightColor); // The vertex color (Gouraud)

    return output;
}
```

We can pass the per-vertex colour `output.VColor` to the pixel shader as a default shading colour.

Using the Pixel Shader to Calculate the Lighting Shade

The next thing we will do is to change our code to perform lighting calculations on a per-pixel basis (Phong shading model) in the pixel shader.

There are some things we need to do before we can move the calculations to the pixel shader:

We need to copy the normal and position in world space to the output from the vertex shader (which will be interpolated by DirectX and input to the pixel shader). The normal is needed for all of the lighting calculations and the position in world space will be needed when you calculate the vector back to the eye/camera position.

The vertex shader output structure which is also the pixel shader input structure might look something like the following:

```
struct PixelShaderInput
{
	float4 Position : SV_POSITION;
	float4 VColor : COLOR;
	float4 PositionWS : TEXCOORD2;
	float4 NormalWS : TEXCOORD3;
	float2 TexCoord : TEXCOORD0;
	float2 BlendMapTexCoord : TEXCOORD1;
};
```

Our vertex shader needs to perform the necessary calculations to populate these values.

Note the odd semantic names (TEXCOORD2 and TEXCOORD3) on the Normal and WorldPosition. HLSL requires semantic names on all fields that will be passed between shaders. However, there is no specific meaning attached to these two fields. So we can use any valid semantic name we want that is not reserved for a specific use. In this case, we use TEXCOORD2 and TEXCOORD3 even though neither of these is a texture coordinate. Just consider it one of the weird aspects of the HLSL language.

Now we can do the lighting colour calculations in the pixel shader. We start by just doing the diffuse calculation to make sure this still works and then add in the specular highlights.

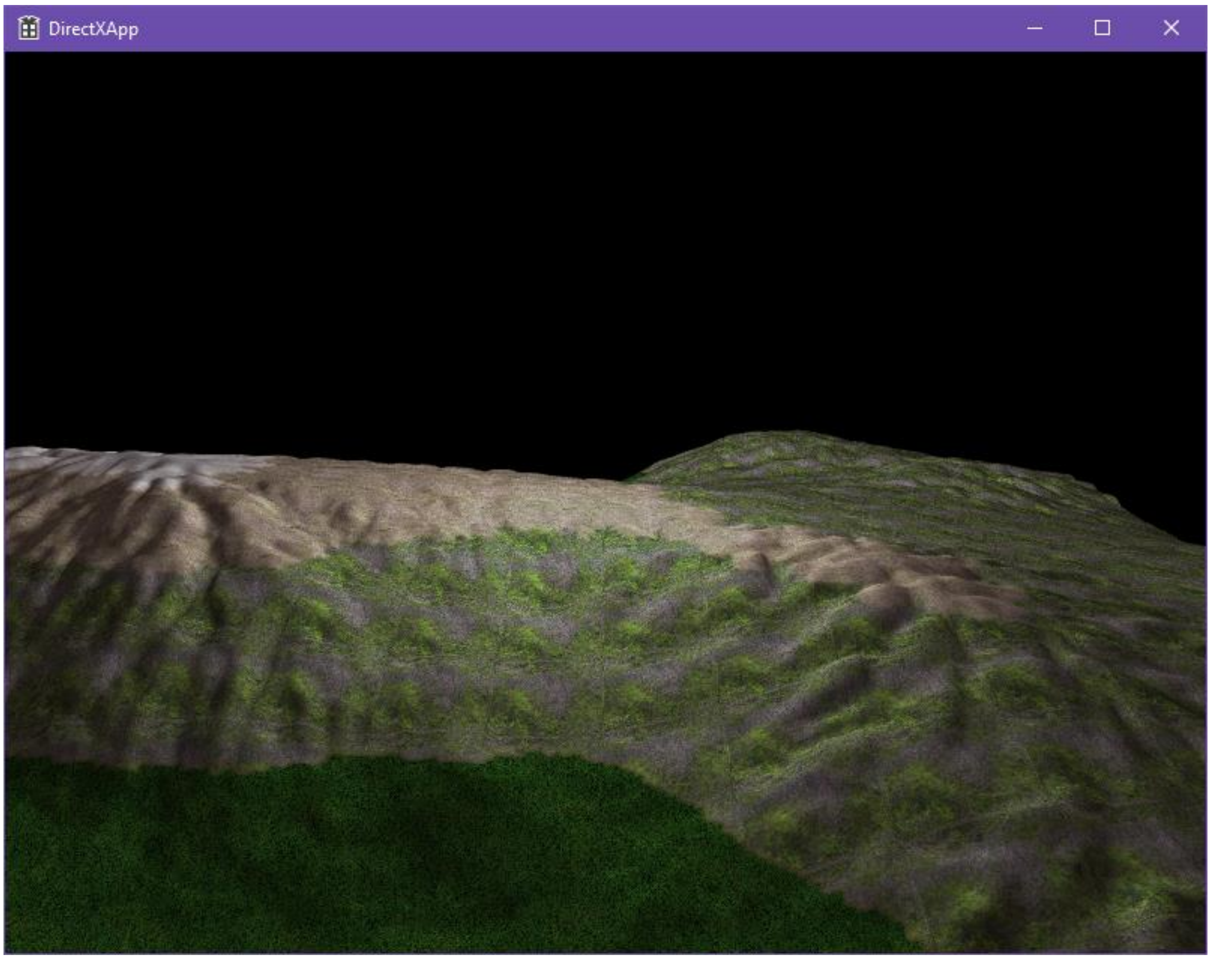
The provided code has an implementation up to the diffuse calculations:

```
//per-vertex shading from Vertex shader (Gouraud)
float4 shade = input.VColor;

// Do your lighting calculations here - per-pixel shading (Phong)
// calculate the diffuse light and add it to the ambient light

//float4 vectorBackToLight = normalize(lightVector); // directional light
float4 vectorBackToLight = normalize(lightVector - input.PositionWS); // point light

float4 adjustedNormal = normalize(input.NormalWS);
float diffuseBrightness = saturate(dot(adjustedNormal, vectorBackToLight));

//shade = saturate(ambientColor); // Ambient
//shade = saturate(diffuseBrightness * lightColor); // Diffuse
shade = saturate(ambientColor + diffuseCoefficient * diffuseBrightness * lightColor);
// ALL
```

Adding the specular component is left for you as an exercise.

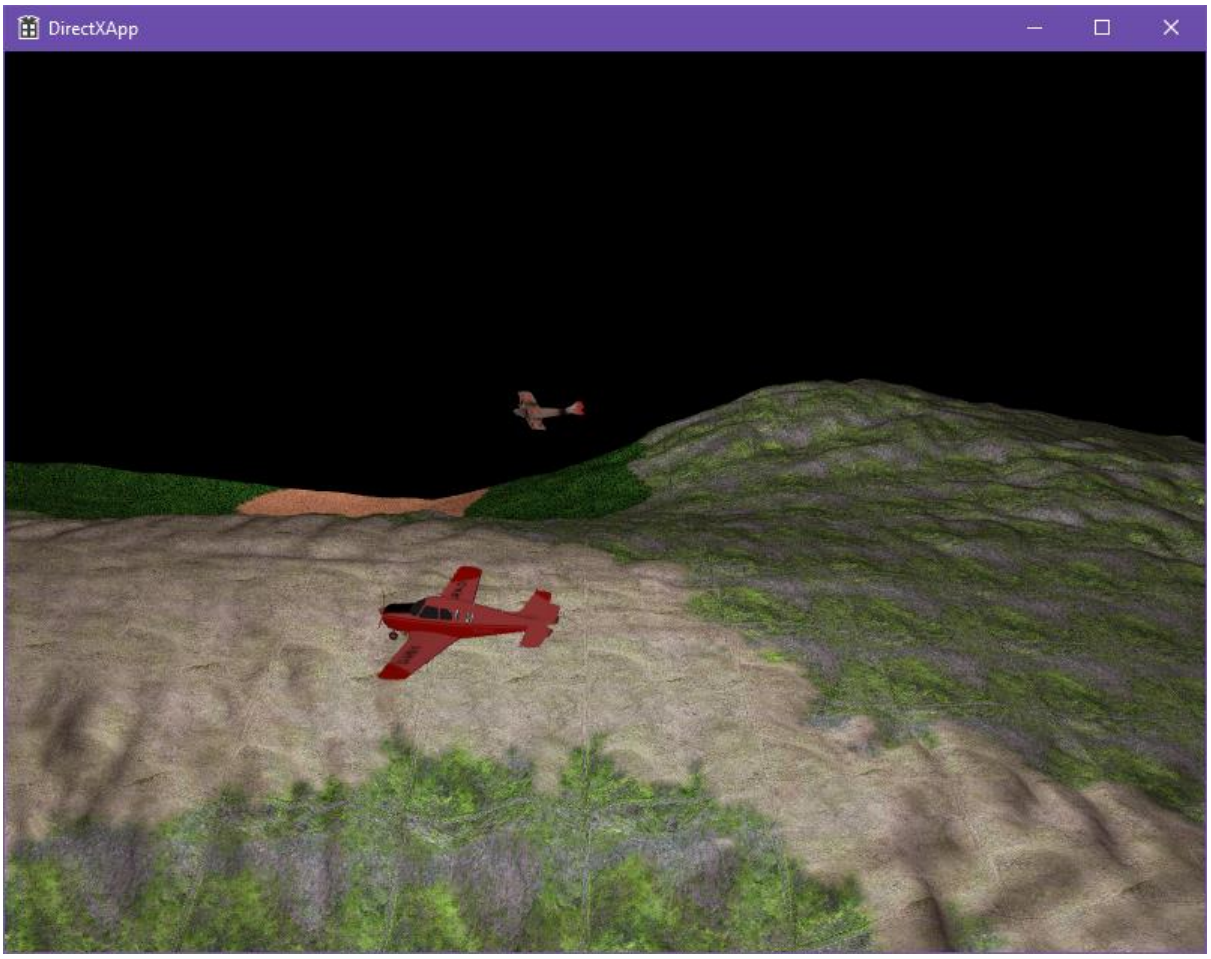
The output of the executable

When you run your code, the result will depend on the values you put in your blend map. Using the values of the given example and with point-light shading (ambient+diffuse) on a per-pixel basis (Phong) for the terrain, you see something like the following as a view.



Exercise

Try, as an exercise, to add the two airplane models to the scene!


DirectXApp