Graphics Part II: Code Tutorial for Week 9

Dr P. Perakis

Preparation

Last week, we had updated the TerrainNode class to incorporate blending of terrain textures and an updated terrain shader for this purpose. 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_09_Files.zip` file.

Introduction

Now you have terrain, it would be better if the area above the terrain was more realistic. This tutorial takes you through what you need to create a node that can display a sky box.

In the files this week, you have been provided a shader that can be used to render the sky box (`skyboxshader.hsls`), as well as a texture cube map that represents the sky (`SkyBox_Tex.jpg`).

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. As before, the way I suggest doing it here, is just a simplistic suggestion using a cube. Feel free to implement it in any way you wish. For more realistic results you can use a sphere as a skybox.

The files provided in `Exercise_09_Files.zip` file are as follows:

SkyboxNode.h and .cpp	The updated class that represents the skybox node.
skyboxshader.hlsl	The skybox shader
SkyBox_Tex.jpg	The texture for the skybox

Create a New Node called SkyNode

The constructor for this node should take as parameter the name of the node. 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 following:

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

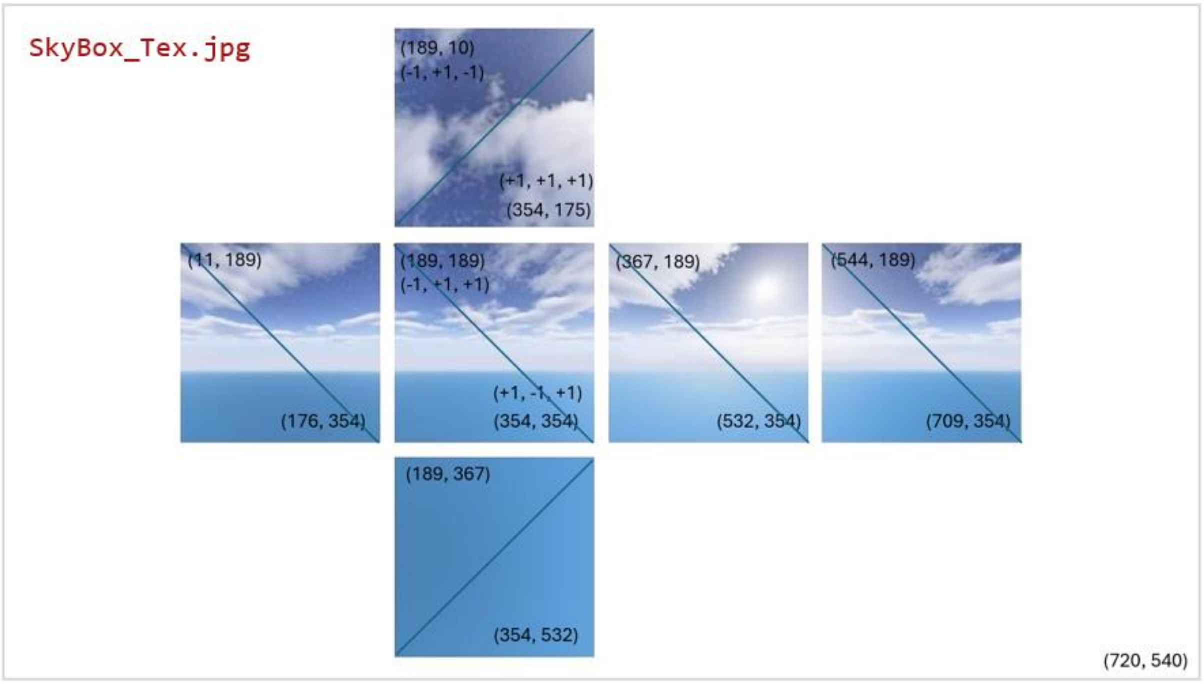
The constant buffer needs to pass in the combined world-view-projection transformation matrix, the model to the world transformation matrix, and an ambient colour for colouring effects.

```
struct CBUFFER
{
	Matrix		WorldViewProjection;
	Matrix		World;
	Vector4		AmbientColor; // the sky's ambient color
};
```

The geometry you need for the node, i.e. the vertices and indices, is that of a cube surrounding the scene. A method to create such a cube is included in the files `SkyboxNode.h` and `.cpp`.

The Sky Cube geometry and texture is defined in `SkyboxNode::BuildGeometry()`.

The texture map used is the following:



You can use the WICTextureLoader or the DDSTexturLoader to load the texture cube into memory.

There are two other things you need to setup. You may 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 `SkyboxNode.cpp`.

Rendering the Sky Cube

The steps for rendering the sky cube are as follows:

- Calculate the world transformation for the sky cube. This should simply be a translation and a scaling according to the position and the size of the terrain that is going to be covered.
- Now you can calculate the complete world-view-projection transformation using this world transformation. We calculate the complete transformation as we have done before.
- Setup the constant buffer.
- You pass the sky map texture through to the shader using PSSetShaderResources just like you have done previously.
- Before the call to DrawIndexed, you may 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());
```

Introducing a new Shader for the Skybox

For a sky box we only need to calculate a simple lighting model with ambient component only. We can use the ambient light colours to give effects to the sky such as full lighting at noon, or dark-blue lighting at night, or even orange-red lighting during sunset.

Thus, the new constant buffer in the shader is:

```
cbuffer ConstantBuffer
{
    matrix worldViewProjection;
    matrix worldTransformation;
    float4 ambientColour;   // The ambient light's colour
};
```

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 light parameters and the transformations have to be passed to the cBuffer in SkyboxNode::Render():

```
CBUFFER cBuffer;
//Set light
//cBuffer.AmbientColor = Vector4(0.90f, 0.50f, 0.25f, 1.0f); //reddish sky
cBuffer.AmbientColor = Vector4(0.1f, 0.1f, 0.2f, 1.0f); //night sky
//cBuffer.AmbientColor = Vector4(1.0f, 1.0f, 1.0f, 1.0f); //noon sky

//Set transforms
cBuffer.WorldViewProjection = completeTransformation;
cBuffer.World = _worldTransformation;
```

Using the Vertex Shader to Calculate the Shading Colour

For the sky box, which is a planar object, we can do the lighting calculations on a per-vertex basis in the vertex shader. This involves only the ambient lighting component since the sky is not illuminated by another lighting source, and then we let DirectX interpolate the colours between the vertices to give us the colour for each pixel.

Using the `VertexIn` structure:

```
struct VertexIn
{
	float3 Position : POSITION;
	float3 Normal   : NORMAL;
	float2 TexCoord : TEXCOORD;
};
```

We can calculate the per-vertex lighting shade:

```
VertexOut VS(VertexIn vin)
{
    VertexOut vout;

    // Transform to homogeneous clip space.
    vout.Position = mul(worldViewProjection, float4(vin.Position, 1.0f));

    // use the ambient light
    vout.Colour = saturate(ambientColour);

    vout.TexCoord = vin.TexCoord;

    return vout;
}
```

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

```
float4 PS(VertexOut pin) : SV_Target
{
     return pin.Colour * Texture.Sample(ss, pin.TexCoord);
}
```

Using different shaders

So far, we have been using a simple shader for all renderings. Here we need to use two different shaders, one for the sky box and one for the terrain.

For this purpose, we need to reset the shaders in every call of the two rendering functions. This is done for the sky box in `SkyboxNode::Render()` using:

```
// Set the shaders to be used by the dc
_deviceContext->VSSetShader(_vertexShader.Get(), 0, 0);
_deviceContext->PSSetShader(_pixelShader.Get(), 0, 0);
```

The same has to be done in `TerrainNode::Render()` for the terrain.

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`.

```
Matrix ModelTrans;

SceneNodePointer skyboxNode_ptr = make_shared<SkyboxNode>(L"Skybox");
skyboxNode_ptr->Initialise();
sceneGraph->Add(skyboxNode_ptr);
```

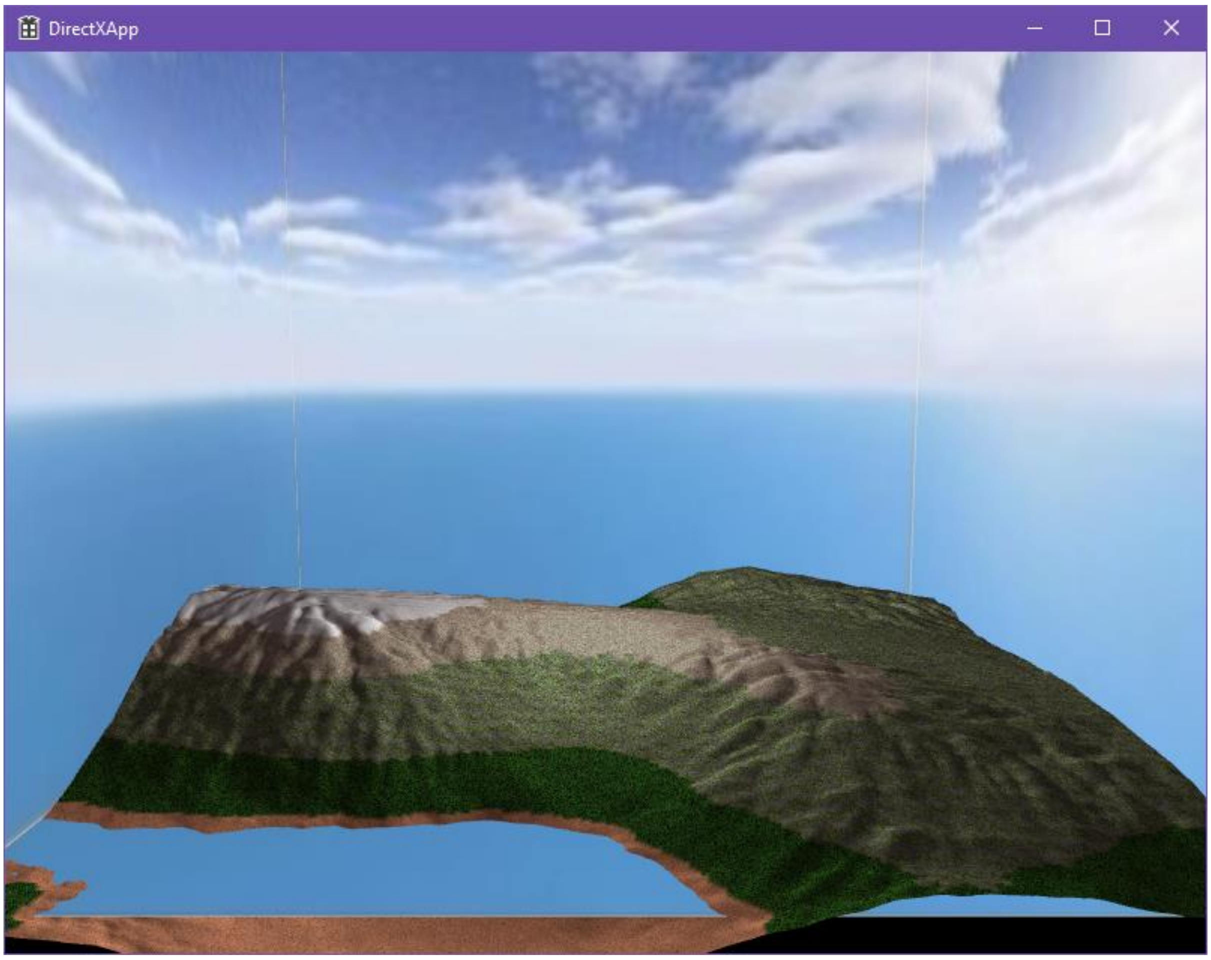
```
ModelTrans = Matrix::CreateScale(120.0f, 120.0f, 120.0f);
ModelTrans = ModelTrans * Matrix::CreateTranslation(0.0f, 75.0f, 0.0f);
skyboxNode_ptr->SetWorldTransform(ModelTrans);
```

When you run your code now, you will hopefully see a close to realistic sky representation above your terrain.

The output of the executable

When you run your code, the result will depend on the values you put in the parameters of the terrain and the skybox. Using the values of the given example and with ambient shading for the skybox you see something like the following as a view.

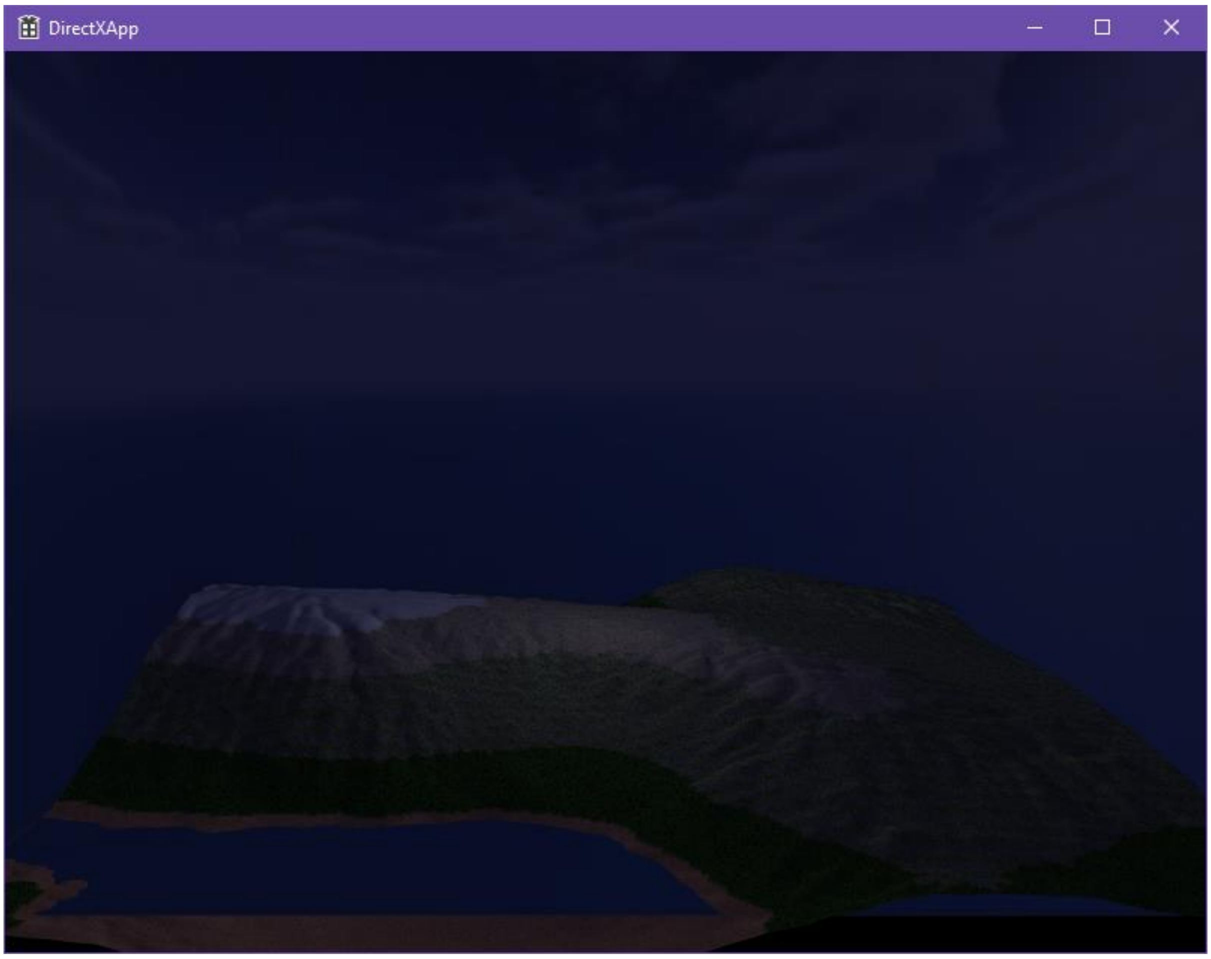
"Daylight sky"



```
⇨  TerrainNode::Render()
   cBuffer.AmbientColor = Vector4(0.3f, 0.25f, 0.25f, 1.0f);
   cBuffer.LightColor = Vector4(1.0f, 1.0f, 1.0f, 1.0f);

⇨  SkyboxNode::Render()
   cBuffer.AmbientColor = Vector4(1.0f, 1.0f, 1.0f, 1.0f); //noon sky
```

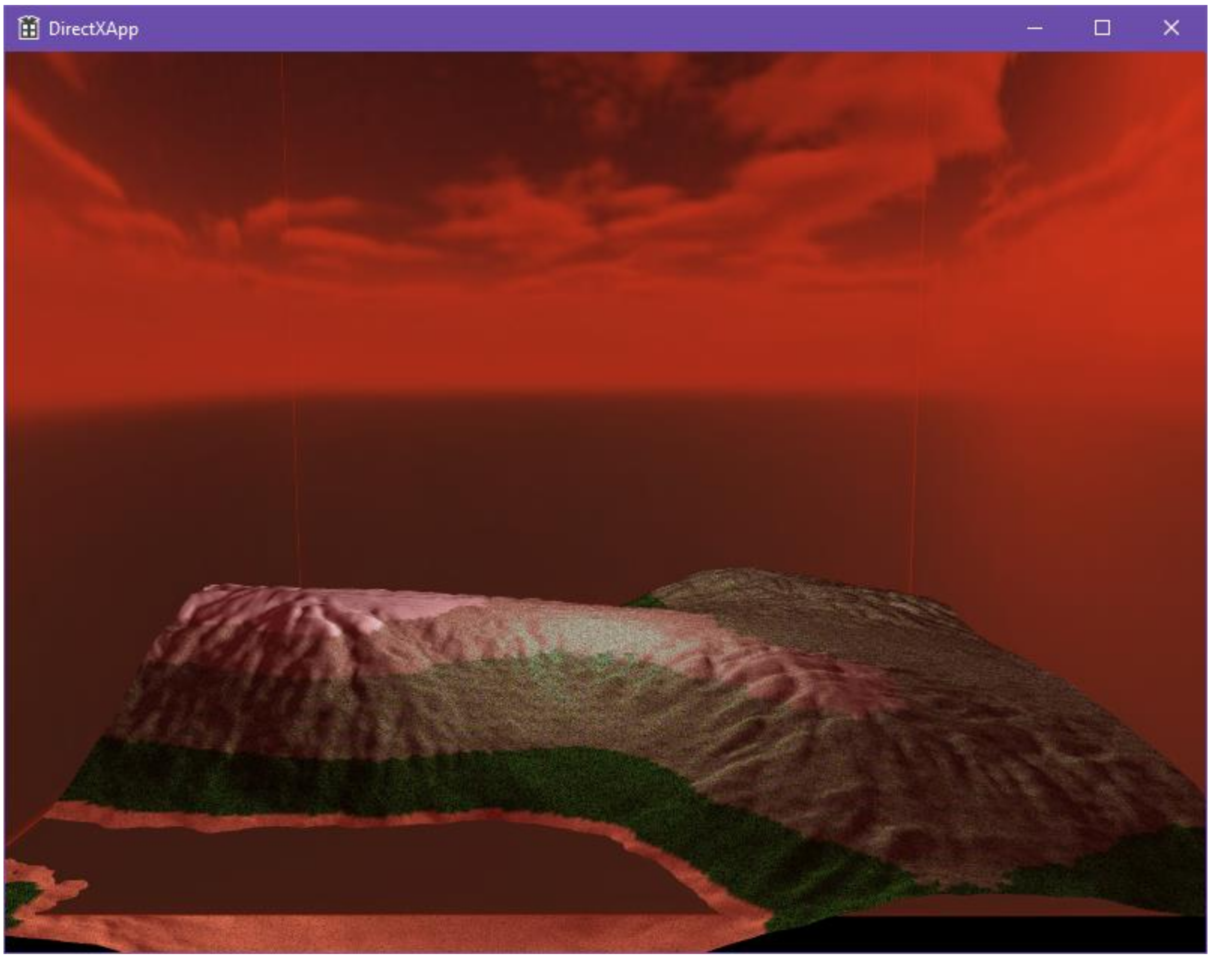
"Night sky"



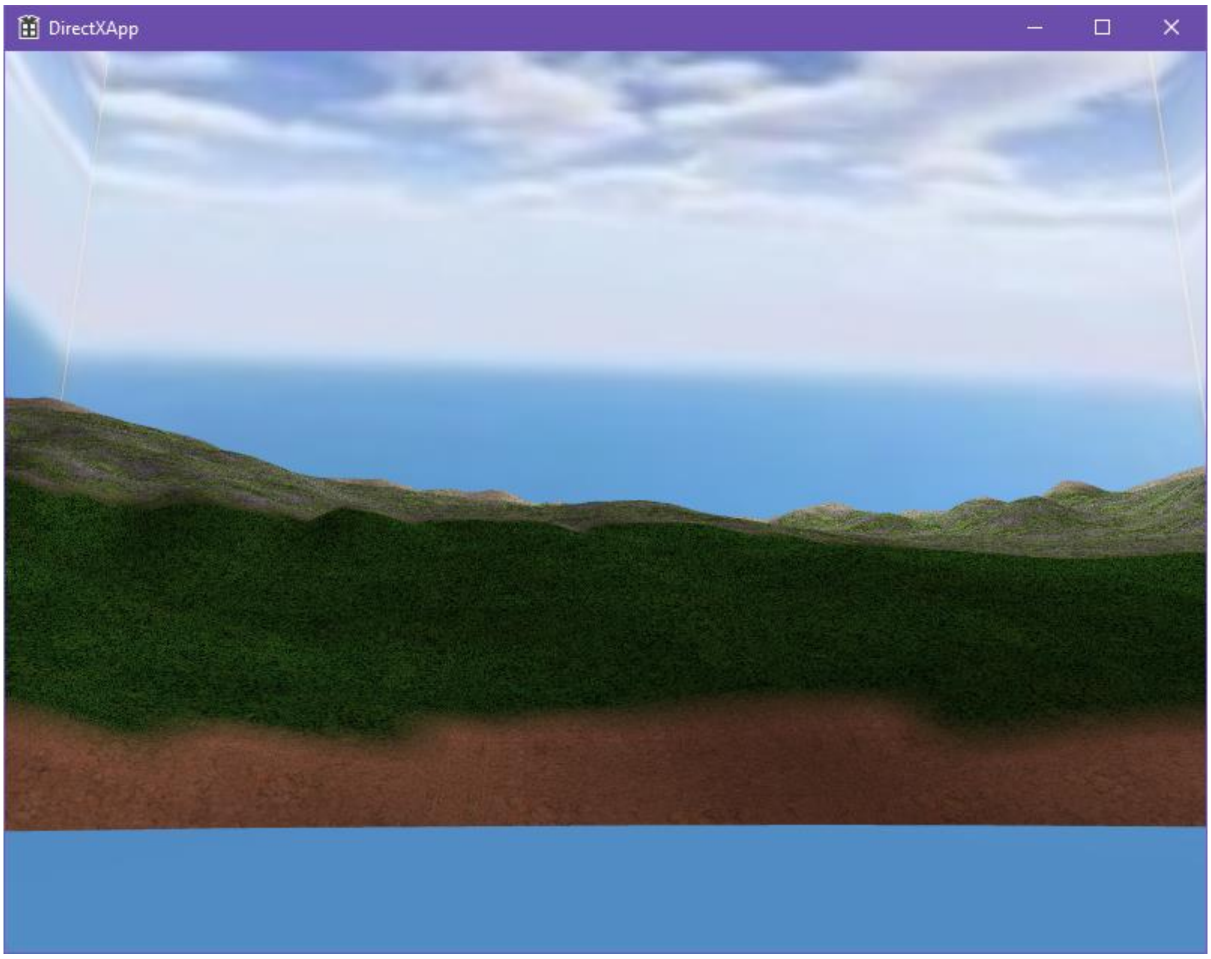
- ⇨ ```
  TerrainNode::Render()
  cBuffer.AmbientColor = Vector4(0.1f, 0.1f, 0.2f, 1.0f); //night sky
  cBuffer.LightColor = Vector4(0.2f, 0.2f, 0.2f, 1.0f); //night sky
  ```

- ⇨ ```
  SkyboxNode::Render()
  cBuffer.AmbientColor = Vector4(0.1f, 0.1f, 0.2f, 1.0f); //night sky
  ```

"Sunset sky"



Lower-level view of terrain and sky



Exercise

Try, as an exercise, to use a sphere as the surrounding sky box instead of a simple cube for more realistic results!