Game Engine Design

Assignment 5 – *Programmable Pipeline*

10 Points

In this assignment, we will create a vertex shader to calculate the vertex position directly from the heightmap. We will also integrate the normal map from the "TextureGenerator" to calculate local lighting in the pixel shader.

Remember: You can stop your ResourceGenerator from rebuilding your heightfield each time by unloading the project – right-click the ResourceGenerator project in the solution explorer and select "Unload Project". If you need your resources to be rebuilt, right-click the project again and select "Reload Project".

GPU Side: Creating new variables (game.fx) (0P)

In the effect file there are some preparations necessary before you can start to write your shaders:

- In the "Shader resources" section at the beginning of the file, add a float buffer variable for the height map of the terrain ("`Buffer<float> g_HeightMap;`") and a Texture2D variable named "`g_NormalMap`".
- In "`cbConstant`": Add the variable "`int g_TerrainRes;`".
- In "`cbChangesEveryFrame`": Add the variable "`matrix g_WorldNormals`".
- Under "`struct PosTexLi{…}`" add a struct "`PosTex`". It should only contain the position and texture coordinates from "`PosTexLi`", using the same semantic names.

CPU Side: Additional Shader Variables in C++ (GameEffect.h) (0P)

- Extend your GameEffect structure by shader variables for:
  - Heightmap (of type `ID3DX11EffectShaderResourceVariable*`)
  - Normalmap (`ID3DX11EffectShaderResourceVariable*`)
  - Resolution (`ID3DX11EffectScalarVariable*`)
  - World Normals Matrix (`ID3DX11EffectMatrixVariable*`)

- Study the existing GameEffect::create() code to learn how the "`SAFE_GET_*()`" macros are used to bind effect variables to the rendering effect. Then bind the new effect variables to the ones you created in your game.fx
  - Hint: Check the names of the types of your newly created variables to find out which of the `SAFE_GET_*()` macros to use!

CPU Side: Additional Resources (Terrain.h and game.cpp) (0P)

- Remove the definitions of “`g_TerrainVertexLayout`” and “`Terrain::vertexBuffer`”. Also remove the creation and destruction of the input layout and vertex buffer; you can also remove the code that calculates the vertex positions for the terrain.
    - Hint: You may want to comment out the according code instead of deleting it, it may serve as a guideline for later assignments.
      For the input layout, look for “`CreateInputLayout`” in “game.cpp”.
- Add a shader resource and a shader resource view for the normal map to the terrain class; also add a shader resource (of type `ID3D11Buffer*` instead of `ID3D11Texture2D*`) and view (`ID3D11ShaderResourceView*` as usual) for the terrain heightmap (we will refer to this as `heightBuffer` and `heightBufferSRV` in this assignment).
    - Hint: Check the declaration of the existing terrain diffuse texture and create the normal texture variables in the same way!

CPU Side: Filling the new buffers (Terrain.cpp) (1P)

- In `Terrain::create()`, load the normal texture with `DirectX::CreateDDSTextureFromFile()`, just like the color texture.
- In `Terrain::create()`, create a buffer for the terrain height map (use your `heightBuffer` variable!).
  This works similar to the old vertex buffer creation, but you fill the buffer’s initial data directly with the content of your heightfield after loading it from file. Your `D3D11_SUBRESOURCE_DATA::pSysMem` will thus be your heightfield array, `D3D11_SUBRESOURCE_DATA::SysMemPitch` is changed to `sizeof(float)`; be sure your heightfield array is float and not double at this point!
  In your `D3D11_BUFFER_DESC`, set the `BindFlags` to `D3D11_BIND_SHADER_RESOURCE` and change the `ByteWidth` so it matches to total byte size of your heightfield.
- In `Terrain::create()`, create a shader resource view (use `heightBufferSRV`) for the new buffer. This time, you need to create a `D3D11_SHADER_RESOURCE_VIEW_DESC`. Check the slides on how the resource view description needs to be filled out correctly!
- In `Terrain::destroy()`, destroy all resources created in this assignment.

CPU Side: Matrices (Game.cpp) (0P)

- In `Game::OnFrameMove` extend the code such that `g_terrainWorld` now includes a scaling transformation that scales the terrain in accordance with the TerrainWidth/Depth/Height provided by your config parser.
- In `Game::OnD3D11FrameRender` set the world normals matrix effect variable in `g_gameEffect` such that it contains the inverse transposed `g_terrainWorld` matrix.
  Hint: This is done similarly to setting the `worldEV` and `worldViewProjectionEV` matrix.

CPU Side: Changing the Render Call (Terrain.cpp) (0P)

- The new render call will not have any vertex buffer bound. For this, in `Terrain::render()`, set the input layout to nullptr instead of `g_terrainVertexLayout`. Also set the content of the array vbs to nullptr and the content of strides to 0.
- In `Terrain::render()`, bind the shader resource views for your normal map and height map to the shader variables (similar to your color texture!)
- Set the resolution shader variable to your terrain resolution (use the `SetInt()` method of the shader variable!)

GPU Side: Terrain Rendering Vertex Shader (game.fx) (4 Points)

Since we do not bind a vertex buffer to the pipeline, we will now need to calculate the position of each vertex directly in the vertex shader using the system-generated value SV_VertexID and the height stored in our heightmap.

- In pass "`P0`" of technique "`Render`": In "`SetVertexShader()`" replace "`SimpleVS()`" with "`TerrainVS()`" and in "`SetPixelShader()`" replace "`SimplePS()`" with "`TerrainPS()`" – we will add the new shaders with new terrain rendering functionality lateron.
- Create a new shader function named "`TerrainVS`" after the existing "`SimplePS()`" shader.
- The only input of shader function is the system generated value "`SV_VertexID`" (see "Windows DirectX Graphics Documentation" for more info). Therefore the parameter list of your vertex shader "function" should look like this: "`uint VertexID : SV_VertexID`".
- The outputs of the shader are the transformed position and the texture coordinates of the vertex. Therefore the return value should be of type "`PosTex`". **(1P for correct shader declaration)**
- In the body of the shader function, define a variable named "`output`" of type "`PosTex`" which is initialized to "0" (cf. "`SimpleVS()`").
- Calculate the xz-position of the vertex using the vertex id and the resolution of the terrain (which you stored as a variable before!) and read the y-position from your heightmap buffer using the vertex id (you can access an element of a buffer in HLSL with the `[]` operator). Translate the positions so the terrain floor is centered at [0,0,0] and apply the terrain scaling. **(1P for correct height, 1P for correct xz position)**
  - Hint: This operation is basically the same as done in the last assignment during vertex buffer creation!
- Just as in `SimpleVS()`, apply the `worldViewProjection` transformation using the HLSL intrinsic `mul`.
- Calculate the texture coordinate from the vertex id. This is pretty much the same as the calculation of the xz-position, only the result needs to be in [0;1]. **(1P)**

GPU Side: Terrain Rendering Pixel Shader (game.fx) (4 Points)

The pixel shader retrieves the terrain's normal and diffuse color from the respective texture maps and applies simple $\boldsymbol{N} \cdot \boldsymbol{L}$ lighting.

- Create a new shader function named "`TerrainPS`" after the "`TerrainVS()`" shader - The input of your pixel shader must have the same type as the output of your vertex shader.
- The output of your pixel shader should be a four-component float vector representing a color ($\mathrm{r}, \mathrm{g}, \mathrm{b} \in [0,1],\ \mathrm{a} = 1$). The semantic name of the output should be "`SV_Target0`", so that that the output color is written to the first render-target. **(1P for correct shader declaration)**
- In the shader body, declare a variable named "`n`" of type `float3` to hold the terrain's normal.
- The normal map uses a compressed image format, in which only the r and g (which map to x and z in terrain space!) components from your original normal map are stored. To fill "`n.xz`", sample the normal texture "`g_Normal`" using an anisotropic sampler and the input's texture coordinates (cf. "`SimplePS()`"). Since "`Sample()`" yields values in the [0, 1] range, you should map the result back to the
[-1, 1] range. **(1P)**
    - Hint: Using HLSL's vector subscript notation and the ability to combine scalar and vector operands, it is possible to do all of this in one line. Example: "`float2 v = (float4(1.0, 2.0, 3.0, 4.0).xyz + 7.0).zx * 9;`"
- Calculate the y-component of the normal. Since you know that the length of "`n.xyz`" is 1 and that all terrain normals point upwards, the value of y is uniquely defined for given x and z. **(1P)**
- Transform the normal to world space and renormalize as done in `SimpleVS()` before. **(1P)**
- Retrieve the terrains material color "`float3 matDiffuse`" by sampling the "`g_Diffuse`" texture (similar to "`SimplePS()`").
- Calculate the intensity of the incoming light "float i" as scalar product (HLSL: "`dot()`") between the terrain's normal and the light direction in object space "`g_LightDir`". Then use HLSL's "`saturate()`" function to restrict the intensity to the $[0, 1]$ range.
- Calculate the r,g,b components of the final output color as "`i * matDiffuse`". **(1P)**

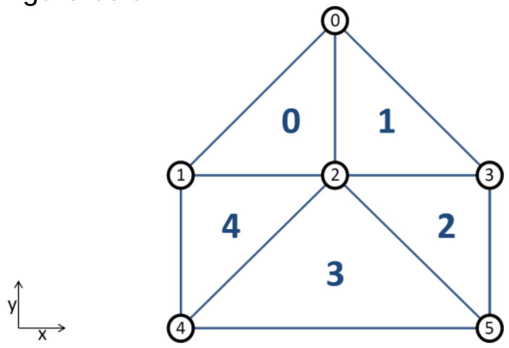
Questions (1P)

Create a section for assignment 5 in "`<username>/readme.txt`" and answer the following questions. You may also commit an image or pdf file containing the solution, but in this case include a reference to your solution file in the readme file.

- A three dimensional geometric model should be rendered using Direct3D's "DrawIndexed" command and the primitive topology "triangle list". The vertex buffer depicted below, contains the x, y and z coordinates of the model vertices in object space.

Vertex 0			Vertex 1			Vertex 2			Vertex 3			Vertex 4			Vertex 5		
x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2.0	2.0	0.0	1.0	1.0	0.0	2.0	1.0	0.0	3.0	1.0	0.0	1.0	0.0	0.0	3.0	0.0	0.0

An index buffer is created to specify the connectivity of the triangle mesh as shown in the figure below.



The content of the index buffer must fulfill the following constraints:

Let v_1; v_2; v_3 be the first, second and third vertex of a triangle in the index buffer.

- The order of the triangles in the index buffer must coincide with the numbering of the triangles in the figure.
- v_1 is the vertex with the lowest index of v_1; v_2; v_3.
- In object space, the vector $c = (v_2 - v_1) \times (v_3 - v_1)$ has a negative z component.

Write down the entries of the index buffer, such that all of these requirements are met. **(0.5P)**

- Given is a perfectly specularly reflecting plane containing the origin *(0,0,0)* in 3D space. The plane is described by the set of all points *(x,y,z)* such that $(x, y, z) \cdot (1,2,0) = 0$. A point light source is positioned at *(-10,0,0)*. Compute the normalized direction of the reflected light at the point in the plane at position *(0,0,0)*. Note: Answers without a derivation will not be graded. Arithmetic expressions like square roots do not have to be evaluated. **(0.5P)**

If you face any difficulties, please use the Q&A Forum at https://qage.in.tum.de/ or ask your tutor.