Praktikum: Echtzeit Computergrafik

Assignment 6 – *Mesh Rendering and Lighting*

10 Points

In this assignment, we will load the "cockpit" mesh and corresponding textures from `external\`, and render it in front of our camera (first-person view). You may start by exploring `external\art\01-Cockpit\cockpit\` for a first impression of the resource files for the cockpit. The .obj files can be opened with nearly any model viewing software, such as e.g. MeshLab.

Resources for the "cockpit" Mesh (0P)

- Similar to our terrain textures, we will first convert the resources for the mesh into an appropriate format that can be loaded at runtime by game.exe. To do this, add the following four lines to the NMake build command line of the project `ResourceGenerator`:

```
"$(SolutionDir)..\..\external\Tools\bin\obj2t3d.exe" -i
"$(SolutionDir)..\..\external\art\01-Cockpit\cockpit\final\cockpit_o_low.obj" -o
"$(OutDir)resources\cockpit_o_low.t3d" –y


"$(OutDir)texconv" -o "$(OutDir)resources" -srgbi -f BC1_UNORM_SRGB
"$(SolutionDir)..\..\external\art\01-Cockpit\cockpit\final\cockpit_m_diffuse.png"


"$(OutDir)texconv" -o "$(OutDir)resources" -srgbi -f BC1_UNORM_SRGB
"$(SolutionDir)..\..\external\art\01-Cockpit\cockpit\final\cockpit_m_specular.png"


"$(OutDir)texconv" -o "$(OutDir)resources" -srgbi -f BC1_UNORM_SRGB
"$(SolutionDir)..\..\external\art\01-Cockpit\cockpit\final\cockpit_m_glow.png"
```

- Build `ResourceGenerator`, and verify the existence of `cockpit_o_low.t3d`, `cockpit_m_diffuse.dds`, `cockpit_m_specular.dds` and `cockpit_m_glow.dds` in your `$(OutDir)/resources` folders.

Background Info: The cockpit mesh is given as an obj-file in `external\`. Though this is a very flexible and open format it is very hard to parse. The tool obj2t3d.exe converts the obj-mesh into our own file format. The t3d-format is specifically tailored for our purpose, and provides you with ready-to-use vertex and index buffers for the mesh (a class for loading t3d-files is available).

- Now add the resource files to your config. In your `game.cfg`, add the following single line (the keyword `Mesh` followed by the name/identifier `Cockpit`, which we will need in future assignments, and the filenames of the four new resources):

```
Mesh Cockpit cockpit_o_low.t3d cockpit_m_diffuse.dds cockpit_m_specular.dds
cockpit_m_glow.dds
```

- Extend your `ConfigParser` such that the five arguments behind the keyword `Mesh` are read and stored in the parser. The five arguments are mesh identifier, mesh file, diffuse, specular and glow texture filename. Remember that these files must be loaded from the `resources` directory.

Mesh Class (1P)

For every mesh that we want to use we need a whole bunch of D3D11 resources (`ID3D11Buffer`, `ID3D11Texture2D`, …). We will encapsulate these resources into a class `Mesh` to ease the management of multiple meshes in future assignments. This class is already given.

- Copy the files `Mesh.h`, `Mesh.cpp`, `T3d.h` and `T3d.cpp` from `external\templates\mesh\` to your project folder `GEDGame\projects\Game\src` and in Visual Studio add them to your game project.
- Investigate the declaration of the `Mesh` class in the header file: Each mesh object (for now) consists of several resource filenames, geometry information (vertex and index buffer) and three different textures together with corresponding shader resource views (diffuse, specular and glow).
- In `game.cpp`: Include `Mesh.h`, and add the following global variables:

```
Mesh*               g_cockpitMesh     = nullptr;
```

- In `game.cpp`: Add an empty function `void DeinitApp()`. In your `wWinMain()` function, add a call to `DXUTShutdown()` and `DeinitApp()` (in this order) right after the call to `DXUTMainLoop()`.
  `DeinitApp()` will be our counterpart to `InitApp()`, i.e. for every `new` in `InitApp()` you have to add a `delete` in `DeinitApp()`. The call to `DXUTShutdown()` ensures that `OnD3D11DestroyDevice()` is called by DXUT before `DeinitApp()`.
- In `game.cpp`: In `InitApp()` create a new `Mesh` object and store it in `g_CockpitMesh` (use the `new` operator and pass the paths for the cockpit mesh from your config parser to the `Mesh` constructor). In `DeinitApp()` add the corresponding delete call for `g_CockpitMesh` (you can use the `SAFE_DELETE` macro instead of delete).
- In `game.cpp`: In `OnD3D11CreateDevice()` call the `create()` method of `g_CockpitMesh`, and in `OnD3D11DestroyDevice()` add the corresponding call to `destroy()`.

Pixel and Vertex Shaders for the Mesh (4P)

In order to render the mesh and perform lighting (in world space), we need some additional variables in the effect file `game.fx` and a new pixel and vertex shader.

- Add new effect variables to the `GameEffect` class (`GameEffect.h`):

```
ID3DX11EffectShaderResourceVariable* specularEV;
ID3DX11EffectShaderResourceVariable* glowEV;
```

```
ID3DX11EffectVectorVariable*        cameraPosWorldEV;
ID3DX11EffectPass*                  meshPass1;
ID3DX11EffectMatrixVariable*        worldNormalsEV; //you added this variable
                        //already in assignment 5, maybe under a different name
```

- For each of the newly added effect variables, add a corresponding variable in your effect file.
  - Hint: The two shader resource variables are textures, just as the already existing diffuse texture, while the others are of type `matrix` or `float4`, respectively.
  - Hint: Put all variables but the textures into the existing `cbChangesEveryFrame` cbuffer. Remember that textures are not stored in cbuffers!
- In `game.fx`: Add two new structs:

```
struct T3dVertexVSIn {
    float3 Pos : POSITION; //Position in object space
    float2 Tex : TEXCOORD; //Texture coordinate
    float3 Nor : NORMAL;   //Normal in object space
    float3 Tan : TANGENT;  //Tangent in object space (not used in Ass. 5)
};

struct T3dVertexPSIn {
    float4 Pos      : SV_POSITION; //Position in clip space
    float2 Tex      : TEXCOORD;    //Texture coordinate
    float3 PosWorld : POSITION;    //Position in world space
    float3 NorWorld : NORMAL;      //Normal in world space
    float3 TanWorld : TANGENT;     //Tangent in world space (not used in Ass. 5)
};
```

Note: the first struct corresponds to `T3dVertex` from `T3d.h` and the input layout created with `Mesh::CreateInputLayout()`, and the second struct contains the view space information about each vertex for world space lighting in the pixel shader.

- In `game.fx`: Create a new vertex and a new pixel shader `MeshVS()` and `MeshPS()`. The vertex shader's input is of type `T3dVertexVSIn`, the output of type `T3dVertexPSIn`. The pixel shader must receive the corresponding input to match the vertex shader and output a `float4` with semantic `SV_Target0`. Check the slides from last week for examples on how to deal with in- and output given as structs!
- In `game.fx`: Create a new pass by copying the existing pass `P0` in `technique11 Render`. Name the new pass `P1_Mesh`, change the vertex/pixel shaders `TerrainVS()/TerrainPS()` to `MeshVS()/MeshPS()` and the rasterizer state to `rsCullBack` (this orders the rasterizer to enable the culling of backfacing triangles, i.e. the rasterizer will throw away triangles that are facing away from the camera).
- "Business as usual" in `GameEffect.h`: Bind the new variables in game.cpp to the new variables and the new pass in game.fx using the `SAFE_GET_*` macros.
- In `game.cpp`: Create the input layout in `OnD3D11CreateDevice()` using `Mesh::createInputLayout()` (as pass use `g_gameEffect.meshPass1`), and don't forget to release it in `OnD3D11DestroyDevice()` by calling `Mesh::destroyInputLayout()`.
- In `game.fx`: Write the vertex shader `MeshVS` according to the slides: Transform the vertex position into clip space (`output.Pos`), copy the texture coordinate

(`output.Tex`) and transform the vertex position & normal & tangent into world space (`output.PosWorld & output.NorWorld & output.TanWorld`). The vertex tangent is reserved for future assignments and should be treated as a directional vector (i.e. like the normal).

- In `game.fx`: Write the pixel shader `MeshPS` according to the slides (phong lighting in world space). Use the already known anisotropic sampler when sampling the textures.
  Hint: Get your program running first by completing the rest of the assignment – in the meantime use the following code in `MeshPS` (simple unlit texturing for debugging):

```
return g_Diffuse.Sample(samAnisotropic, Input.Tex);
```

Transformations and Render Call (2P)

Finally, we will add a draw call in `OnD3D11FrameRender` in `game.cpp` order to render the already loaded mesh with our new shaders. Binding of the required resources is already implemented in the mesh class. For correct positioning of the cockpit mesh setup the transformation from the mesh's object space into view space. You can do this either in `OnD3D11FrameRender()` or in `Mesh::render()` before the draw call.

- Create the transformation matrices for the cockpit mesh and set the corresponding effect variables of `g_gameEffect`:
  `worldEV:` object space to world space
  `worldNormalsEV:` object space to world space (for normals)
  `worldViewProjectionEV:` object space to projection space
  - Note: $WorldNormals = (World^{-1})^T$. Refer to slides for reference and examples.
  - Note: The cockpit should move and rotate with the camera. Therefore use the camera's world matrix.
- Set the effect variable `g_gameEffect.cameraPosWorldEV` to the camera position in world space. For this you can use the value from `g_camera.GetEyePt()`.
- Now call the `g_cockpitMesh->render()` method from `OnD3D11FrameRender()`.

Adjust the Camera (1P)

When your cockpit mesh is rendered properly, make some final adjustments to the game camera. With what will become a stationary turret cockpit, we want to be positioned in the center of the terrain slightly above ground level.

- In `game.cpp`: In `OnD3D11CreateDevice()`, search for the part where the camera `g_camera` is initialized. Change the parameter `Eye` to be in the center of the terrain, i.e. set `Eye.x` and `Eye.z` to `0`, and set `Eye.y` to an appropriate value depending on your heightfield (Hint: move the camera initialization code to after the position where the heightfield is read; also the `terrainHeight` read from `game.cfg` might help).
- In `game.cpp`: Search for other occurrences of `g_camera` and disable the position movement of the camera.
- Set `g_terrainSpinning` to `0` (= disable spinning by default). You could also add this to your config.

- In `game.cpp`: For debugging purposes, you still might want to be able to "fly around". One possibility to achieve this is binding a hotkey that re-enables the camera movement when pressed. To do so, add a similar line to `OnKeyboard()`:

```
//Enable position movement when the C-key is pressed
if (nChar=='C' && bKeyDown) { /* enable position movement here */ }
```

Info: The callback function `OnKeyboard()` is called whenever keys are pressed by the user. To bind special keys like `F1`, you can use the `VK_*` macros (e.g. `VK_F1` instead of `'C'`).

Questions (2P)

Create a section for assignment 6 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.

- Write down a 3×3 matrix which realizes the following transformation: A 2D perspective projection in the x/y-plane onto the line $y = 0$. The center of projection is at $(0, -6)$. After the projection the homogeneous component should be such that the homogeneous division generates the correct values. **(1P)**
- Write down a single 3×3 matrix which realizes the following transformation: A rotation in $\mathbb{R}^3$ about 90 degrees about the axis aligned with the vector $(1,1,0)$. Note that it is not sufficient to write down a sequence of matrices. **(1P)**