Game Engine Design

Assignment 9 – *Sprites*

10 Points

In this assignment, we will implement a sprite renderer. A sprite is a 2D image that is used as a (heavily simplified) representation for a volumetric, usually transparent object, such as a ball of plasma. This will be used in the next assignment to render projectiles from the guns.

Resources

As always, we first need to add some new resources to the content pipeline. However, this time it's only the two textures in `external/art/05-Sprites/simple`.

- Add both images in `external/art/05-Sprites/simple` to your resource generator. Use BC3_UNORM_SRGB as a format (-f) parameter instead of BC1_UNORM_SRGB.

Background info: These textures contain an alpha channel, which defines their opacity at each pixel. This is put into the `.dds` file automatically, and is accessible when you sample the texture in a shader. We will use this later to implement transparency. However, the BC1_UNORM_SRGB format used for our diffuse textures only stores 1 bit for alpha (see http://msdn.microsoft.com/de-de/library/windows/desktop/hh308955(v=vs.85).aspx for a reference).

SpriteRenderer (1P)

To keep your code tidy, it would be a good idea to encapsulate the sprite rendering functionality into its own class. To get you started, a header file (`SpriteRenderer.h`) is provided in `external/templates/`. It contains a `struct SpriteVertex`, which defines one sprite, and a `class SpriteRenderer`, which will contain the rendering functionality and also manage all required GPU resources. This is only meant to assist you – using it is not mandatory, and you can change anything you want (e.g. add members or function parameters).

- Create the corresponding `.cpp` file and write dummy implementations for all member functions, so they can be called. The constructor should initialize all member variables to reasonable values (use `nullptr` / `0` where no real value is known yet); all other member functions can be empty for now (except for returning a value where required – e.g. `S_OK` as default value for an `HRESULT` return type).
- Then integrate the `SpriteRenderer` into the `game.cpp`: Declare a global instance (e.g. `SpriteRenderer* g_SpriteRenderer`), create it (`new`) in `InitApp`, delete it again in `DeInitApp`, call `Create/Destroy` in `OnD3D11Create/DeleteDevice`, and call `Reload/ReleaseShader` in `Reload/ReleaseShader`. You can write the filenames for the sprite textures directly into the `game.cpp` for now – we will fix this in the next assignment.

Checkpoint: At this point, check that your code compiles without errors, and produces no memory leaks during runtime.

GPU Resources: Effect, Vertex Buffer, Input Layout (1P)

For rendering, we are going to add a new effect (i.e. `.fx` file and corresponding `ID3DX11Effect`; contains shaders and related D3D state) and a vertex buffer with a corresponding input layout.

- Effect: Add a new `SpriteRenderer.fx` file to the project. In this file, define a `technique` and a `pass` (look at the `game.fx` to see how this is done). The `pass` specifies a vertex and pixel shader; for now just add some dummy functions like the following:

```
void DummyVS(out float4 pos : SV_Position) {
    pos = float4(0, 0, 0.5, 1);
}
float4 DummyPS(float4 pos : SV_Position) : SV_Target0 {
    return float4(1, 1, 0, 1);
}
```

This vertex shader always outputs a constant position (in the middle of the screen), the pixel shader outputs the constant color yellow.

The pass also specifies a `RasterizerState`, `DepthStencilState` and `BlendState`. For now, you can copy `rsCullNone`, `EnableDepth` and `NoBlending` from `game.fx`.

  - **HINT***:* The correct build tool, the `.fx` compiler, is applied automatically by Visual Studio. Set the output to `$(OutDir)shader/%(Filename).fxo` in the properties. If you see an error message “main: entrypoint not found” upon compilation, change the properties of “SpriteRenderer.fx” to use shader type “Effect (/fx)” and shader model “Shadermodel 5 (/5_0)”.
  - **Checkpoint:** Build your project (F7) and make sure the `SpriteRenderer.fx` compiles without errors, and the file `shader/SpriteRenderer.fxo` is created.
- In `SpriteRenderer::ReloadShader`, create the `ID3DX11Effect`. To do that, you have to load the content of the `SpriteRenderer.fxo` file into memory and then call `D3DX11CreateEffectFromMemory`. (Hint: Search in `GameEffect.h` for an example for how this can be done.) Release it again in `ReleaseShader`.
- In `SpriteRenderer::Create`, create a Vertex Buffer for the sprites using `ID3D11Device::CreateBuffer`. If you need information on this function and its parameters, look at the D3D documentation. Choose the size of the buffer so that it can contain some fixed number of sprites (e.g. `1024 * sizeof(SpriteVertex)`). Do not fill the buffer with data yet, i.e. specify `NULL` for the parameter `pInitialData` (we will fill the buffer immediately before rendering).
- In `SpriteRenderer::Create`, create an Input Layout based on the `SpriteVertex` struct using `ID3D11Device::CreateInputLayout`. Again, look at the online docs and the slides for more info. (Hint: There is an example in `T3d.cpp`.)
- In `SpriteRenderer::Destroy`, release everything you created in `Create`.
- In `SpriteRenderer.fx`, define a `struct SpriteVertex` similar to the one in `SpriteRenderer.h`, using the semantics you defined in the Input Layout (see slides). Use it as an input parameter to `DummyVS()`.

Checkpoint: Run your program to make sure everything is created and released correctly.

Hint: All the `ID3D11Device::Create*` functions return an error code of type `HRESULT`, which is `S_OK` if everything went fine, and `E_*` (e.g. `E_FAIL`) otherwise. You should always check the value of this error code (i.e. using the `V(…)` and `V_RETURN(…)` macros). If something went wrong, look at Visual Studio's Output window (during a Debug run) – D3D usually prints a detailed error message there telling you exactly what is wrong.

Point Rendering (3P)

Having created the required resource, we can now implement the rendering. First, we need some sprites that we can render.

- In `OnD3D11FrameRender`, for now just fill an `std::vector<SpriteVertex>` with some fixed sprites (choose some arbitrary world-space positions near the camera; radius should be `> 0` and texture index should be `0` or `1`). We will get dynamic sprites in the next assignment. Call `g_SpriteRenderer->RenderSprites(…)` after all other draw calls (transparent objects must be rendered after opaque objects!).

Now, we will implement `SpriteRenderer::RenderSprites`. We will render the sprites as point primitives (single pixels), and later create a geometry shader that extrudes each point to a quad.

- The first step is to fill the vertex buffer with data from the passed-in sprites. To do that, call `UpdateSubresource` on the passed device context to fill the buffer with data (again: see D3D docs or slides).
- Now set the Input Assembler state: Bind the Vertex Buffer (`IASetVertexBuffers`), set the Input Layout (`IASetInputLayout`), and set the Primitive Topology (`IASetPrimitiveTopology`) to point list.
- Apply the pass from your effect (see slides), and call `Draw` on the device context.
- Don't forget to release the context, or you will get a "non-zero reference count" warning when exiting your program.

Checkpoint: Now when you start your program, you should see a single yellow pixel in the middle of your screen (if you are not sure why this happens, look at the vertex/pixel shader again – the data from the vertex buffer isn't used at all so far!).

- To get the points/sprites to the correct position in the terrain, we have to apply the view and projection transformations (given by the camera) to their positions (which are already in world space. Create a new global variable `g_ViewProjection`, and use it to transform the sprite position in `DummyVS`. Finally, fill `g_ViewProjection` with the correct matrix in `SpriteRenderer::RenderSprites` **before** you apply the pass.

Checkpoint: Now when you start your program, the yellow pixels should appear at the correct world-space positions you defined (Since these are only single pixels, they might be hard to see – to ease testing, you can comment out the terrain rendering in `game.cpp` for now, and set a black background color. Also make sure they're not hidden inside another object…)

Points → Sprites (3P)

Finally, we are going to create a geometry shader to extrude our points to quads. For each point primitive, the geometry shader should output two triangles to form this quad.

- To make the quads screen-aligned, we are going to need to camera's "right" and "up" vectors (`camera.GetWorld*`) in the shader, so create variables for them and set their values in `RenderSprites`.
- In `SpriteRenderer.fx`, create a function for the geometry shader, and use it in `SetGeometryShader` in your `pass`. The function header could look like this:

```
[maxvertexcount(4)]
void SpriteGS(point SpriteVertex vertex[1], inout TriangleStream<PSVertex> stream)
```

Background info: `[maxvertexcount(4)]` specifies the maximum number of vertices the geometry shader can output, and is required. `point SpriteVertex vertex[1]` specifies that our geometry shader operates on point primitives, and each point is specified by a `SpriteVertex`. The array of size 1 seems unnecessary here, but is required anyway for reasons of consistency – if the primitive type were `triangle` instead, the array size would have to be 3 etc. The special `TriangleStream` parameter is where we put out output vertices using its `Append` method. `PSVertex` is a `struct` (which you have to define yourself) that will be passed to the pixel shader.

- Change the Vertex Shader output and Pixel Shader input so that they match the new Geometry Shader. Your effect should compile without errors now.
- Implement the geometry shader function so that it outputs the four corner vertices (i.e. "position +/- radius*right +/- radius*up"). Note that you have to do the transformation with `g_ViewProjection` **after** you have created the corner vertices, so you have to do it in the geometry shader instead of the vertex shader now. The vertex shader should just pass its input through unmodified.

Checkpoint: Run your program again – you should now see yellow squares instead of points.

- Now it's time to finally put the textures on the sprites. Extend `create` and `destroy` in `SpriteRenderer` so that the `Texture2D`s and `ShaderResourceView`s are created and released. Add shader variables for the textures, and set them in `RenderSprites`.
- Extend your geometry shader so that it generates texture coordinates, and you them in the pixel shader to sample one of the textures. (Hint: You have to `switch` over the `TextureIndex` to decide which texture to sample!) For debugging, it can be helpful to output the texture coordinates as a color so you can see what arrives in the pixel shader (e.g. `return float4(input.TexCoord, 1, 1);`).

Checkpoint: Run your program – you should now see the sprite textures, but they are not transparent yet.

- To make them transparent, you have to configure the `BlendState` in the effect file to perform back-to-front alpha blending (see the slides).

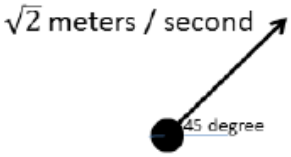
Final Checkpoint: Your program should display several transparent sprites floating at fixed world-space positions (visible from the initial camera position) with different radii, using both of the provided textures.

One last thing remains: For transparent objects, the order in which they are rendered matters! To get correct results when two sprites overlap on the screen, we have to sort the sprites into back-to-front order before rendering. But we will leave that for next week…

After you are done, check your program again for memory leaks or unreleased resources.

Questions (2P)

A particle of mass 10 kg is positioned at the point (10, 10) in meters. The particle has an initial velocity of magnitude $\sqrt{2}\frac{m}{s}$ into a direction of 45° with respect to the x-axis (see the figure to the right).



Assume a gravity constant of g = 10 $\frac{m}{s}$ and neglect air resistance.

1) Compute the **velocity** of the particle after 1 second using the explicit Euler method with a time step of 0.5 seconds.
2) Compute the **position** of the particle after 1 second using the semi-explicit Euler method and a time step of 0.5 seconds.

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