Exercise 3

Computer Graphics - COMP.CE.430

Rasterization pipeline

This week we will be writing relatively simple vertex and fragment shaders using SHADERed. The goal of this week's exercises is to understand how the data gets into the vertex shader and how it gets transformed in the vertex and fragment shaders. All the tasks that are not marked as bonus tasks are requirements for the project assignment. You don't need to complete all of these to pass the course, but some of the later exercises during the course may assume that you have completed them. The tasks marked as bonuses give you extra points that increase your grade. After completing the mandatory parts for this week, you should have a 3D model on the screen.

Problem set

Start by creating a new empty project in SHADERed. Right-click the *Pipeline* view on the left side and select *Create Shader Pass*, and choose a name for the shader. Doing so will create an empty vertex shader and an empty fragment shader for you. At this stage, the shaders won't compile and you'll get an error. The minimum you need to add to get the shaders to compile is the version declaration (this must be the first line in the shader) and an empty main function that takes no parameters and returns void. For the version declaration, SHADERed defaults to 3.30, but I've used the newest version, which is 4.60 for writing these instructions. To use version 4.60, add #version 460 as the first line of the shader. If your machine doesn't support version 4.60, you can use version 3.30 instead by adding #version 330. In this case, you'll have to write some things slightly differently. If you run into problems, please ask the course personnel for assistance.

1. Render a triangle on the screen. You can start with the 2D template. In the desktop version, the 2D template is called *QuadGLSL*. In the web version this is called *2D GLSL*. Delete the quad by right-clicking the Quad object and selecting *Delete*. You can add a triangle by right-clicking the Shader Pass (called Main in the template), selecting *Add -> Geometry* and then selecting *Triangle* from the *Geometry type* drop-down menu.

2. Make the triangle move over time by shifting the position in the vertex shader. This is similar to last week's exercise, but it is done in the vertex shader this time. **(0.5p)**

3. Give each vertex a different color. You can do this by hard-coding an array of *vec3* variables that define a color for each vertex and indexing that array with *gl_VertexID*. You should define this array in the vertex shader. You can read input variables in the fragment shader by declaring an input variable such as:

```
layout(location=0) in vec3 in_color;
```

To output a variable from the vertex shader, you need to declare a corresponding output variable such as:

```
layout(location=0) out vec3 out_color;
```

Note that the *location* identifiers need to match. The hardware does interpolation for the variables that are passed to the fragment shader so you should see a smooth gradient across the triangle even though you're only using 3 different colors. **(0.5p)**

4. Load a glTF model in SHADERed. You should probably create a new *Shader Pass* for this step. This way you can use a different shader for the triangle and the 3D model, and you can hide individual shader passes in the *Pipeline* view by clicking the eye icon.

   You can load 3D models from files by right-clicking the shader pass and selecting *Add -> 3D Model*. You can either choose your own model or use one of the sample models that are available on Moodle. If you're using your own model, it is recommended to use a model that has normal vectors and texture coordinates available. **If you are using the web version of SHADERed**, you can just add a cube or a sphere instead.

   The glTF format is a standard format for 3D scenes and models. The Khronos Group is responsible for maintaining the glTF specification. The glTF is supported by many 3D art programs such as Blender and many game engines and renderers are capable of importing glTF files. It also has support for physically based materials and animations. **(0.5p)**
5. Render the loaded 3D model on the screen. The vertex input will be automatically bound to the shader after completing the previous step. To render 3D the model on the screen, the input vertices need to be transformed into homogeneous clip space. Again, for now, you don't need to worry about what exactly this means, we will cover it in future lectures. In practice, you will need to multiply the input vertex coordinate by a view-projection matrix in the vertex shader. You can create a view-projection matrix by right-clicking the shader and clicking *Variables*, then selecting mat4 as the *Type* and *ViewProjection* from the *System* drop-down menu. How these matrices are derived will also be covered in future lectures, so you can treat this as a black box for now. You also need to define the uniform in the vertex shader. You can do this by typing for example

```
uniform mat4 viewproj;
```

in the global scope. The name should match the name you gave it in the *Variables* window. You can use this matrix to multiply the input position and output the result to *gl_Position*. GLSL has overloaded the standard multiplication operator for matrix-vector multiplication so you can just type

```
gl_Position = viewproj * vec4(in_pos, 1.0);
```

The view-projection matrix is a 4x4 matrix so we convert the position to a homogeneous coordinate by adding a 4th component that is set to 1.0. The result of this multiplication is a 4D vector in homogeneous clip space.

You can change the shading to a flat color in the fragment shader for this step. **(0.5p)**

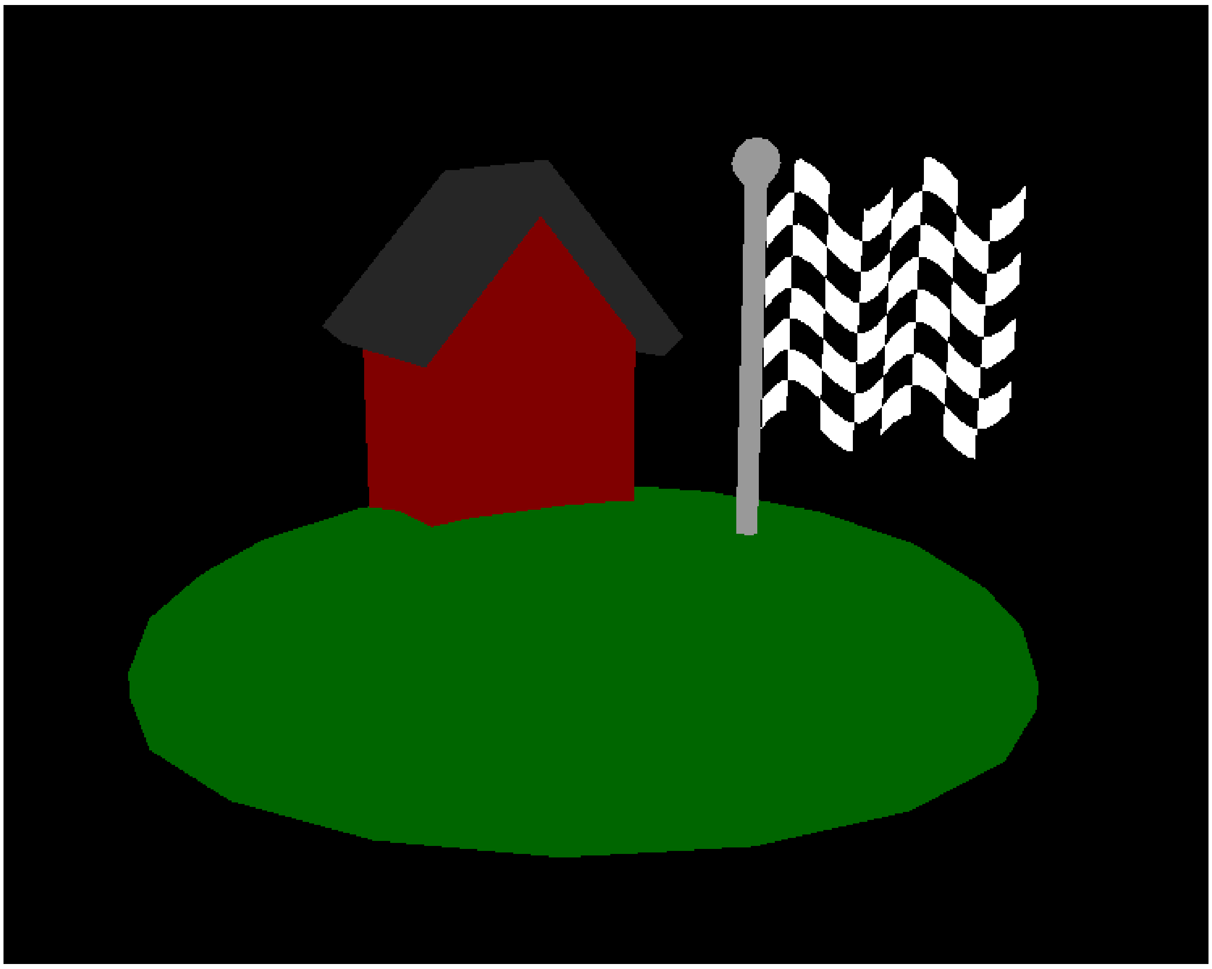
6. We have a 3D object on the screen now, but it doesn't look very 3-dimensional when it's flat shaded. Lighting will be covered next week, but for now, you can visualize the 3-dimensionality by outputting the normal vectors as the color. Add an input for the normal vector in the vertex shader the same way you did for the position, and an output and input in the vertex and fragment shaders the same way you did for color in step 3. Normal vectors are typically unit-length vectors. Normal vectors are in general no longer unit-length after interpolation, so you should renormalize them in the fragment shader. You should also rescale them to be in a range from 0 to 1 so you can visualize them better. **(1p)**

Bonus

1. Create a procedural animation in the vertex shader. The shape of the object should change during the animation, simply making a moving object won't qualify for extra points. You can for example create a sphere and morph it into some sort of a blob by offsetting the vertex positions using some function based on the vertex position and time. The vertices should stay connected during the animation so no gaps are introduced due to the offsets. **(1p)**

2. Create a procedural texture in the fragment shader. You can use similar techniques as in the first exercise. However, instead of deriving *uv* from *gl_FragCoord* like in the example solutions, you can use the texture coordinates from the 3D model as the *uv* coordinate. Add the appropriate texture coordinate inputs and outputs the same way you did for the normal. Texture coordinates are usually two-dimensional. They're commonly used for texture mapping, as you may have guessed based on the name, but you can of course use (or abuse) them for anything you'd like. You don't have to use texture coordinates for this exercise of course, how you implement this is completely up to you. **(1p)**

Example scene



This is an example of what your scene could look like after implementing the optional features. The scene is a 3D model created in Blender and loaded in from a glTF file. The flag has a procedural animation implemented in the vertex shader and a procedural texture applied in the fragment shader. The colors for each object are defined as uniforms in the shader.