6.1 Basic OpenGL Pipeline (20 + 15 + 20 + 10 = 65 Points)

In this task, you will setup your very own OpenGL shader pipeline.

a) Write the shaders:

- Write a vertex shader:
  - Its inputs should be a position, a color, and a uniform `float` scaling parameter.
  - Its outputs should be a 4D `gl_Position` (implicitly defined, make sure to set its component $w = 1$) and a 3D (or 4D) color.

  Scale the position by the scaling parameter and pass through the color value.
- Write a fragment shader:
  - Its inputs are the outputs of the vertex shader.
  - Its output is a 3D (or 4D) color (you need to name it).

  Simply set the output to the given input color.
- use GLSL `#version 330` or later
- (You can start with a simpler version first, and add more features later.)

b) Setup and cleanup: Create following things in the constructor `MyLittleShader()`. Delete them all in the destructor `~MyLittleShader()`.

- Create a shader program and store its "handle" in the `GLuint program`. You can use `GLShaderProgram::compileShaderProgram` to simplify shader compilation.

  glCreateProgram, glCreateShader, glShaderSource, glCompileShader, glAttachShader, glDeleteShader, glLinkProgram, glDeleteProgram
- Create a vertex array and store its "name" in the `GLuint vertexArray`.

  glCreateVertexArrays, glDeleteVertexArrays

- Create as many buffers as you need and store their "names" in `std::array<GLuint, 4> buffers`.
  glCreateBuffers, glDeleteBuffers
- Make sure that nothing is bound when you are done. (Bind to `0`)
- Regularly check for errors using the given `CHECK_GL();` macro.

c) Upload the buffers: Create and upload the data necessary to have a colored quad like seen in the lecture slides using two triangles in the constructor `MyLittleShader()`. You will need one buffer for positions and one buffer for colors. You can use the existing types `Point2D` (2 floats), `Point3D` (3 floats), `Color` (4 floats) or use your own data.

- Upload the positions to one buffer and the colors to another buffer.
  glBindBuffer, glBufferData
- Enable and set the vertex attribute for position and color matching your shader code. Make sure to specify the correct data type here. Set `normalized` to `GL_FALSE`.
  glBindVertexArray, glGetAttribLocation, glEnableVertexAttribArray, glVertexAttribPointer
- Make sure that nothing is bound when you are done. (Bind to `0`)
- Regularly check for errors using the given `CHECK_GL();` macro.

d) Rendering:

- Enable your shader program.
  glUseProgram
- Set the uniform parameter(s):
  - Set the scaling parameter to $0.5 + 0.25 \cdot \sin(\text{time})$.

  glGetUniformLocation, glUniform
- Draw the triangles.
  glBindVertexArray, glDrawArrays
- Make sure that nothing is bound when you are done. (Bind to `0`)
- Regularly check for errors using the given `CHECK_GL();` macro.

Use only the raw OpenGL methods except for compiling the shader program. Since OpenGL uses a C-style API, you will have to read up on how to use pointers if you're not familiar with using them. Your result should look like the image in Figure 1.

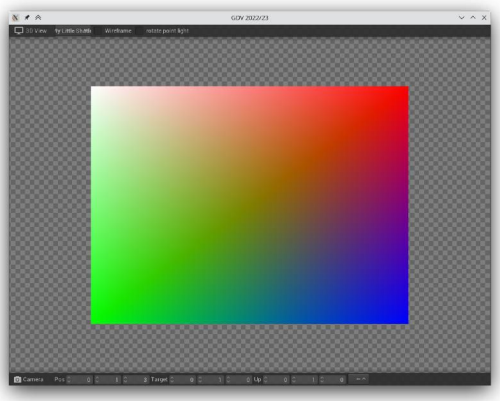

Figure 1: Colored Quad rendered by `MyLittleShader`. The size of the quad animates between $\frac{1}{4}$ and $\frac{3}{4}$ of the window size.

6.2 Simple Fragment Shading (5 + 5 + 5 = 15 Points)

Next, we implement some debugging shaders. We want to recreate the output generated by

```
Color RayTracer::depthIntegrator(const Scene& scene, const Ray& cameraRay) const;
Color RayTracer::positionIntegrator(const Scene& scene, const Ray& cameraRay) const;
Color RayTracer::normalIntegrator(const Scene& scene, const Ray& cameraRay) const;
```

You can find their implementations in `raytracer.cpp`. Use their implementation as a reference for your shader implementation. Your result should match their output.
As basis for your fragment shader, the `fragment_shader_debug` is already given. It is used in

```
void MeshShader::drawDebug(const AABB& bounds, const Point3D& cameraPos, RenderMode
    mode, const Matrix4D& vp);
```

a) Depth shader: Return the distance to the camera origin, normalized by the scene's AABB's maximum extent (including the camera position in the AABB – already given).

b) Position shader: Return the world space position, normalized by the scene's AABB.

c) Normal shader: Return the world space normal, normalized to $[0, 1]^3$.

Your output should look like the one shown in Figure 2. You can also enable the ray tracer in the `main.cpp` and compare side-by-side.

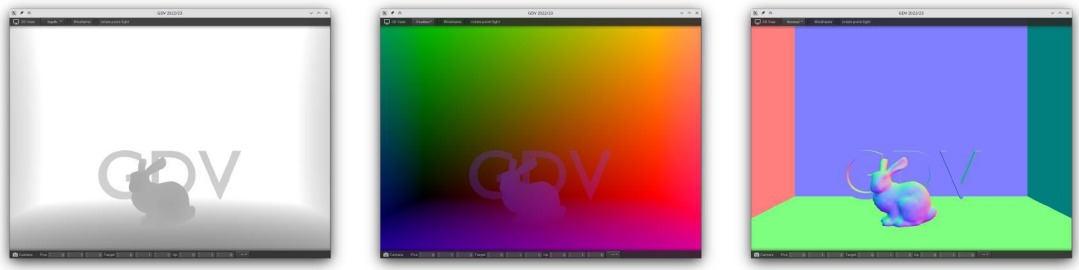

Figure 2: Depth, position, and normal shader.

6.3 Vertex Shader (10 Points)

Using vertex shaders, we can efficiently transform large amounts of scene geometry. Your task here is to implement a bouncy-castle like transformation we call "wobble":

$$\text{wobble}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 + 0.25 \cdot \sin(5 \cdot (\texttt{time} + y)) \cdot x \\ y \\ 1 + 0.25 \cdot \sin(5 \cdot (\texttt{time} + y)) \cdot z \end{pmatrix}$$

Apply the wobble to the input `position` before further processing it through the given matrices.
Your result should look like the image in Figure 3.

6.4 Diffuse Shading (10 Points)

Now, we want to compute some diffuse shading using point light illumination.
A fragment shader template is already given:

```
extern const std::string fragment_shader
```

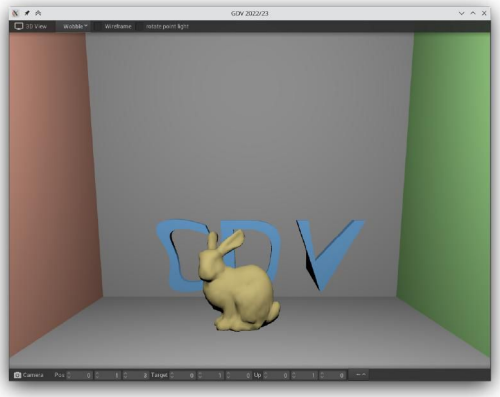

Figure 3: Wobble shader. Here, only the shapes of the objects matter. The (fragment) shading will be part of the next task.

Diffuse shading using a point light:

$$L_o(x, \omega_o) = \underbrace{\frac{\rho}{\pi}}_{f_r(\omega_o, x, \omega_i)} \cdot \underbrace{\frac{\Phi}{4\pi} \frac{1}{d^2}}_{L_i(x, \omega_i)} \cdot \max(0, \cos\theta_i)$$

Here, Φ is the point light's power, ρ is the diffuse albedo, θ_i is the angle between the surface normal n at point x and the direction towards the point light at position y, and $d = \|x - y\|_2$ is the distance between x and y.
We use the maximum of 0 and $\cos\theta_i$ to discard any illumination coming from the back-facing side.
You can compare your result with the Whitted-style ray tracer by enabling it in the `main.cpp`. Except for shadows, it should look the same.