Lab 2 - 3D Transformations and Vertex Shaders

Support code: /course/cs123/src/labs/lab02
Demo: /course/cs123/bin/cs123_lab02_demo

Intro

Last week you learned how to draw 2D shapes in OpenGL using VBOs and VAOs. In this lab, we will use OpenGL to draw a 3D scene with moving shapes! Before beginning the lab, there are a few concepts we need to go over.

Transformations

In graphics we often want to move and position objects in 3D space. We can achieve this using three basic types of transformations: translation, scaling, and rotation.

In mathematics and in code, we represent 3D transformations with a 4-by-4 matrix. Later in the course you will learn about the structure of these matrices but for these labs we will use a library called GLM to provide them for us.

If we have a square centered at the origin like in the diagram below, we can translate that square to a new location by multiplying each vertex position by a translation matrix $T$.

![Translation Diagram](image)

Note that matrices are multiplied on the left of each vertex position ($T \times v_1$). Multiplying the other way would be an invalid operation.

Representing positions and directions as vectors

We represent positions and directions with column vectors. In order to multiply these vectors by 4-by-4 transformation matrices, we represent points not just by an $(x, y, z)$ triplet, but by a vector with four components: $(x, y, z, w)$. 
What goes in the w component?
  ● For vectors representing a **position**, w always equals 1.
  ● For vectors representing a **direction**, w always equals 0.

In this lab you’ll just be dealing with positions. Later on we’ll deal with directions and it will become clear why this difference exists.

**GLM**

In many of our OpenGL programs we will use a mathematics library called GLM (OpenGL Mathematics). Some GLM types that you will use in this lab are:
  ● `glm::vec3` - A vector of size three.
  ● `glm::vec4` - A vector of size four.
  ● `glm::mat4` - A 4x4 matrix.

GLM can also provide transformation matrices, using these functions:
  ● `glm::translate(glm::vec3 v)` - Returns a matrix that translates in X, Y, and Z according to the given vector.
  ● `glm::scale(glm::vec3 v)` - Returns a matrix that scales in X, Y, and Z according to the given vector.
  ● `glm::rotate(float angle, glm::vec3 axis)` - Returns a matrix that rotates by the given angle about the given axis.

Here is a brief example of how to use these:

```cpp
glm::vec4 p = glm::vec4(1, 0, 0, 1); // Creates a point at (1,0,0).

// Creates a transformation matrix that translates by 1 in the // x-direction and by 3 in the z-direction.
```
glm::mat4 T = glm::translate( glm::vec3(1, 0, 3) ) ;

p = T * p; // Translates p.

assert( p == glm::vec4(2, 0, 3, 1) ) ;

Getting Started
The support code is located in /course/cs123/src/labs/lab02. Copy this directory into your course directory and open your copy of the provided .pro file with Qt Creator.

Task 0:
In the Projects tab on the left of QtCreator, de-select the “Shadow Build” option, then Clean and Build your code.

Support Code
Take a look at GLWidget::initializeGL(). Here we have initialized m_sphere and m_square. These are both instances of OpenGLShape, the class you implemented last week. We have included our implementation of OpenGLShape in this lab’s support code. Take a look for a refresher!

The sphere has a radius of 0.5 and is centered at the origin. (The position data is hardcoded in sphere.h so that we don’t give you the answer to Shapes!) The only thing left to do is draw it.

Task 1:
In GLWidget::paintGL(), draw m_sphere after the call to glUniform3f (which we’ll discuss later). Run your program. The output should look a lot like the circle you drew in Lab 1.

The Vertex Shader
Let’s say instead of having the sphere in the center of the screen, we want to move it up along the y-axis.

One way we could do this would be to redefine all our vertex positions with new y-values and send them to the GPU using OpenGLShape::setVertexData(...). This is a bad way to do it. Firstly, it is cumbersome and costly to recompute all the vertex positions for a sphere. Secondly, shuttling data from the CPU to the GPU is expensive. In many programs objects are moving constantly, so we really want to avoid calling setVertexData(...) on every frame.

Vertex shaders to the rescue! **Vertex shaders** are programs that process each vertex individually before rendering. Here’s the OpenGL pipeline diagram again to give an idea of where it fits in the big picture.
Shaders run directly on the GPU, so they are a blazing fast way to do computations for each vertex. Their main job is to output the vertex’s final position. They can also do other useful things, which we’ll see in Lab 3.

Take a look at the file shader.vert, located in the “Other files” folder in Qt Creator. It is written in a language called GLSL (OpenGL Shading Language) which is designed to resemble C. Let’s take a closer look:

- `#version 400 core`
  All shaders begin with a line like this that identifies the GLSL version being used.

- `layout(location = 0) in vec3 position;`
  This declares a vertex attribute (called an in variable) with type vec3 that is called position. On each execution of the vertex shader, position gets one of the positions that we stored in the VBO. layout(location = 0) is an optional part of the declaration that allows us to set the location (or index) of this attribute. Remember in Lab 1 when we used 0 as the "index" when calling OpenGLShape::setAttribute()? That was because the position attribute’s location was set to 0 in the vertex shader.

- `void main() { ... }`
  All vertex shaders must contain a main function, which is automatically called for each vertex.

- `gl_Position = vec4(position, 1.0);`
  The main function is required to set the value of gl_Position, a built-in variable that determines the final position of the vertex. Here we are simply setting it to the value of position. We need to provide a vec4, though, and we set the w-component to 1.0.

GLSL has a lot of built-in types and functions, which mostly are identical to GLM! So a vec4 in GLSL is the same as a glm::vec4 in C++.
Okay, so we have a sphere centered at the origin, but we want all of its vertex positions to be translated in the y direction. Let's use the vertex shader to do this!

**Task 2:**
In the vertex shader, add 0.5 to gl_Position.y. Run the program. The sphere should be moved up on the screen!

**Uniform Variables**

Instead of hardcoding the transformation like this in the shader, we would like the shader to work for arbitrary transformations. To do this, we'd like to send a transformation matrix from the C++ code to the vertex shader.

We've already seen one way to pass information to the shader: store it in a VBO and have it show up in an “in” variable. This is good for data that varies across vertices (like positions). But we want the transformation matrix to be the same for all vertices, so instead we'll use something called a **uniform**!

To set the value of a uniform, you must call glUniform before drawing. On each execution of the vertex shader, it will use the last value that was set using glUniform.

Here’s a chart to summarize the differences between a vertex attributes and uniforms:

<table>
<thead>
<tr>
<th>Vertex attribute</th>
<th>Uniform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different values for different vertices</td>
<td>Same value for all vertices</td>
</tr>
<tr>
<td>Defined as an “in” in the vertex shader</td>
<td>Defined as a “uniform” in the shader</td>
</tr>
<tr>
<td>Values are set using a VBO and a VAO</td>
<td>Value is set by calling glUniform</td>
</tr>
</tbody>
</table>

The first argument of glUniform is the “location” of the uniform variable, which you can obtain from glGetUniformLocation.

Here’s an example of setting a uniform variable with type vec3:

```c++
// This sets the active shaders. It must be called before setting // any uniforms.
glUseProgram(m_program);

GLint uniformLoc = glGetUniformLocation(m_program, "myUniform");
```
// Sets the uniform to (1, 0, 0).
glUniform3f(uniformLoc, 1, 0, 0);

// Draws the shape with myUniform set to (1, 0, 0).
m_shape.draw();

// Alternatively you can use glUniform*v and pass a pointer to a GLM
// type.
glm::vec3 v = glm::vec3(0, 1, 0);
glUniform3fv(uniformLoc, 1, glm::value_ptr(v));

// Draws the shape with the uniform set to (0, 1, 0).
m_shape.draw();

We already defined a uniform in the vertex shader called “model.” The **model matrix** is a matrix that transforms an object (discussed more in the next section).

Note: glUniform is a family of functions that have different names depending on what you’re passing. glUniform3fv passes a vec3, whereas glUniformMatrix3fv passes a mat3.

**Task 3:**
*In paintGL(), pass a translation matrix to the “model” uniform. Check the glUniform documentation to decide which version of the function to use. Then in the shader, instead of adding 0.5 to gl_Position.y, transform gl_Position using the matrix. Run your program. The sphere should be moved according to the translation matrix!*

You may have noticed that the support code sets another uniform called “color” in paintGL(). This is used by the **fragment shader** (which we’ll cover in Lab 3) to assign the object’s color. Feel free to change that value to change the sphere’s color!

**Model, View, and Projection Matrices**

We used a model matrix to transform an object from **object coordinates** (where the origin is at the center of the object) to **world coordinates** (where the origin is at another set location).
In a typical 3D scene, there are two additional matrices we use to produce the final vertex positions.

Firstly, the camera might be located at some arbitrary location in world coordinates. The **view matrix** takes all objects from world coordinates to **camera coordinates**, where the camera is at the origin and pointed along the negative z-axis.

You could make your own view matrix by combining a translation and a rotation matrix, but GLM has a handy-dandy function to do this for you. glm::lookAt takes in the camera’s position, the center point where the camera is looking, and the camera’s “up” direction (usually (0,1,0)); and it returns the corresponding view matrix.

The final matrix we care about is the **projection matrix**. This is a more complicated matrix that takes care of making a perspective projection, which you’ll learn more about in a lecture later in the semester. For now, know that it transforms from eye coordinates to **clip coordinates**, where closer objects appear bigger.
To obtain a projection matrix you can call glm::perspective, which takes in some camera parameters as well as the aspect ratio of the screen.

In order to render a scene with a perspective projection, we need gl_Position to be in clip coordinates. Since the sphere’s vertex positions are defined in object coordinates, we need to multiply each vertex position by all three of these matrices.

**Task 4:**
*In the vertex shader, create new uniforms for the view and projection matrices. Multiply gl_Position by all three matrices.***Pay attention to the multiplication order!*** Matrices are multiplied right-to-left, so the transformation matrix on the left will be applied last. Now, in paintGL(), use glm::lookAt, glm::perspective, and the variables we defined for you in paintGL() to create the view and projection matrices. Send the matrices to the respective uniform variables.

Your program should now show a smaller sphere, because the camera is farther away at (0, 1, 6).

**Drawing Multiple Shapes**
Now let’s draw m_square! We’ll assign it a different color so we can tell the difference between the sphere and the square.

**Task 5:**
*After drawing m_sphere, set the “color” uniform to a different RGB color. Then draw m_square.*

You should now see a colored square. But where’s the sphere? Based on the vertices’ initial positions, we would expect to see the front half of the sphere in front of the square...
By default, OpenGL will draw each shape on top of what it already drew. But we only want to draw on top if we’re drawing a triangle that’s closer to the camera! Fortunately, it’s easy to make OpenGL do a “depth test” so farther triangles aren’t drawn on top of closer ones.

**Task 6:**
In `initializeGL()`, call `glEnable(GL_DEPTH_TEST)`. You should now see half of a sphere poking out front of your square.

**If you still don’t see the sphere**, make sure you set the “color” uniform to a new color between `m_sphere.draw()` and `m_square.draw()`.

Now let’s use our knowledge of uniforms and transformations to make the scene more interesting!

**Task 7:**
Make your scene do something like the demo! Here’s what the demo does:
- The square is scaled by 2 and rotated to lie on the XZ plane.
- The ball is translated to look like it’s bouncing. For this, you can use the equation $y = 0.5 + \text{fabs}(\sin(3 \times \text{time}))$;
- Camera position ($e_y e$) goes in a circle of radius 6 where $y$ always equals 1.

Once you’ve got a scene with a moving camera and transformed objects, have a TA check you off!