INM376 / IN3005 Computer Graphics using OpenGL Lab 2

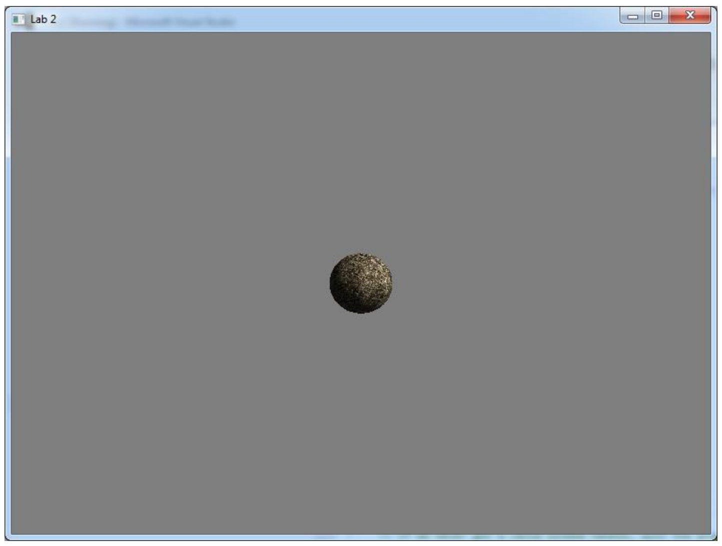
In this lab, we will experiment with matrix transformations. We'll implement modeling matrix transformations using GLM as well as a matrix stack. Then, we'll make multiple assemblies of multi-part objects. The skills you'll develop in this lab are fundamental to geometric modeling in computer graphics.

Download the Demo

1. Download Lab 2 Source Code ("Lab2Demo.zip") from the "Session 2" area on Moodle.
2. Copy the file onto the D: drive. This accesses a local work drive on your machine, so it will be quick to compile but your work will not be stored for future use. You will need to copy it back into your own workspace to use it in the future.
3. Unzip the zip file onto the D: drive.

Open the "Lab2.sln" file, and press 'F5' to compile the program and run it. Hopefully you will see a brown textured sphere rendered to the centre of the screen, like in the figure below.

The sphere is centred at the origin and has a radius of one. In this example, the y axis is up, x axis is to the right, and the z axis is coming out of the screen (forming a right handed coordinate system – you can use the right hand rule to double check this by putting your right hand in the direction of the x axis, and curling your fingers towards the y axis. You thumb will point in the direction of the z axis.)



Task 1: Spin out

Let's program the sphere to automatically rotate about the y axis, using the standard matrix code provided in GLM.

In the Game.cpp class, there is a member variable `m_rotY` already defined. We'll use this variable to keep track of the rotation angle. Copy the following line of code and paste it into the `Game::Update()` method:

```
m_rotY += 100.0f * m_dt;
```

Here, `m_dt` is the amount of time taken to render the previous frame, measured in seconds. By multiplying the rotation increment by `m_dt`, we will get a consistent rotation rate independent of the speed of the computer. It's customary to update moving objects in an `Update()` method, and render them in a `Render()` method.

Now in the `Game::Render()` method, you will notice the modeling matrix is initially set to the identity matrix `glm::mat4 mModelMatrix = glm::mat4(1);`

As we learned in class, the identity matrix is the default transformation, and does not change the vertex positions. Let's apply a rotation to the modeling matrix so that the sphere will rotate when rendered. After the modeling matrix is initialised, apply a rotation of `m_rotY` degrees around the y axis to the modeling matrix. Recall the general syntax for a rotation matrix in GLM is

```
   mModelMatrix = glm::rotate(mModelMatrix, angle, glm::vec3(x, y, z));
```

where angle is the angle in degrees, and $[x, y, z]^T$ defines the vector be rotated about. In your call to `glm::rotate()`, be sure to replace angle and x, y, z the rotation angle and the y axis, respectively, with appropriate arguments. Also, be sure that your rotation is applied before the line

```
   m_pShaderProgram->SetUniform("modelMatrix", mModelMatrix);
```

This line sends the modeling matrix to the vertex shader. We'll cover vertex shaders in detail in Lecture 4. But for now, you should be aware that the vertex shader uses the modeling matrix to transform each vertex of the sphere.

Run your program. Hopefully you see the sphere rotating!

Task 2: Using a matrix stack

While it is possible to use GLM in this way to create matrices and manipulate them, it is often more convenient to use what is known as a "matrix stack". As discussed in lecture, a matrix stack provides a way to save and restore matrices. This can be quite handy when rendering multiple objects.

A matrix stack has two fundamental operations:

- *push*: pushing a matrix on to the stack saves it for later use
- *pop*: popping the stack restores the previously saved matrix

Let's replace the code we have just written so that it uses a matrix stack instead.

In the `Game::Render()` method, replace the line

```
glm::mat4 mModelMatrix = glm::mat4(1);
```

with

```
glutil::MatrixStack modelMatrixStack;
modelMatrixStack.SetIdentity();
```

This uses the namespace `glutil` to create a matrix stack called `modelMatrixStack`, and initialises the matrix stack to the identity matrix.

Now replace your call to `glm::rotate` with a call to

```
modelMatrixStack.Rotate(glm::vec3 vector, float angle);
```

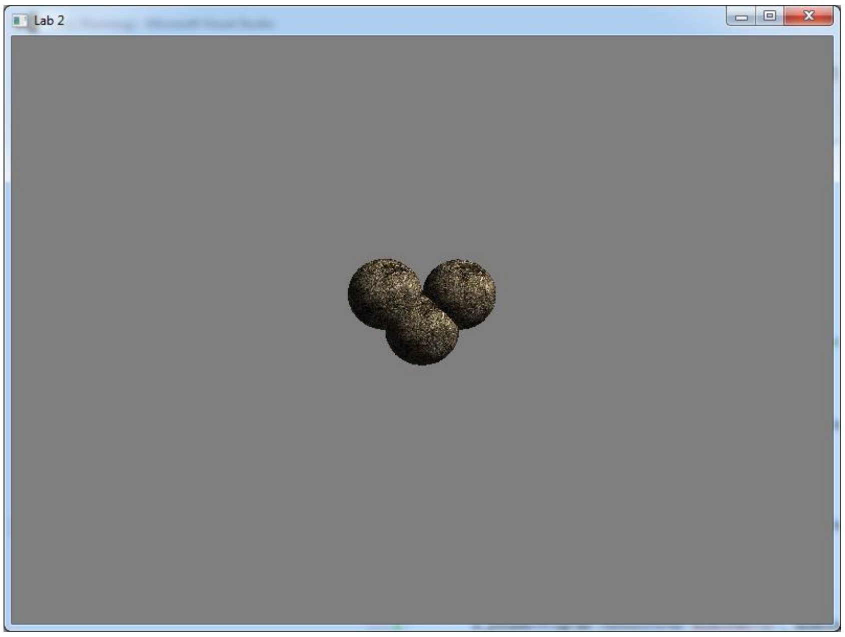
Be sure to provide valid arguments into the `Rotate` method. Finally, we must pass the matrix from the matrix stack to the vertex shader. This is done using the call

```
m_pShaderProgram->SetUniform("modelMatrix", modelMatrixStack.Top());
```

This retrieves the top-most matrix from the stack, which is the one that includes our rotation. Now rerun your program. Hopefully you see the exact same rotating sphere – only now we're using a matrix stack.

Task 3: More spheres

Now, we'll use the matrix stack to render two more spheres to get something that looks like this:



After the call to `modelMatrixStack.Rotate()`, place the code that renders the sphere inside a push / pop block, like this:

```
modelMatrixStack.Push(); {
       m_pShaderProgram->SetUniform("modelMatrix", modelMatrixStack.Top());
       m_pSphere->Render();
} modelMatrixStack.Pop();
```

The push will save the current modeling matrix (the one that has the rotation) on the stack, so that we can restore it later. The curly braces help with indenting in Visual Studio 2015.

Now render another sphere, centred at $[-1, 1, 0]^T$, by copying the above four lines, and inserting a call to `modelMatrixStack.Translate()` after the push but before the call to `m_pShaderProgram->SetUniform()`.

Next, render a third sphere in a similar fashion but centred at $[1, 1, 0]^T$. Now rerun your program. Hopefully you see an assembly of three spheres rotating a group!

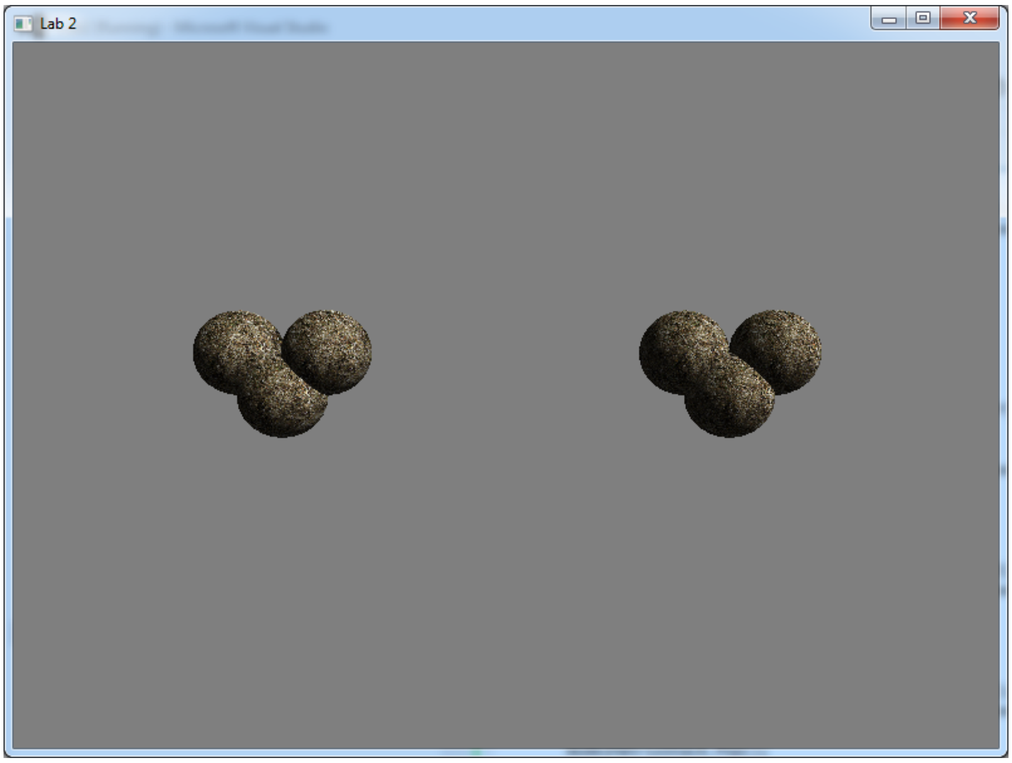
For each sphere, once we've pushed (saved) the modeling matrix, we can apply other transformations to the modeling matrix before rendering an object. Then we pop the matrix stack to restore the previously saved modeling matrix.

Task 4: Two assemblies

Based on your understanding of the matrix stack, see if you can translate the assembly of spheres five units along the x-axis. The assembly should rotate about its bottom-most sphere.

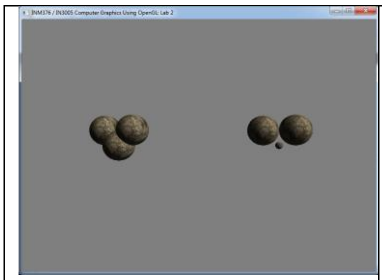
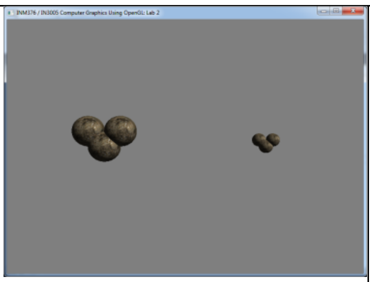
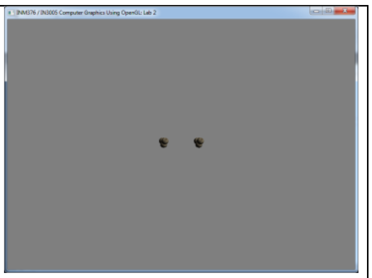
Then, create another assembly of sphere translated *negative* five units along the x-axis. The final result should look something like that below, with both assemblies rotating about their lowest sphere.

Hint: It may be helpful to push the modeling matrix stack just after it is initialised to the identity matrix. Be sure to pop after rendering the assembly to restore the modeling matrix to the identity matrix. Remember SoRT!



Task 5: Scale

Some of you may take some time to complete Task 4. If so, that's absolutely fine; we're here to learn. However, if you finish this task quickly, now try incorporating a uniform scaling term, for example, `float s = sin(m_rotY / 100.0f);` in the your transformations. See what happens if you scale an individual sphere, an assembly of three spheres, or everything, based on where you put the scaling transformation relative to the calls to `modelMatrixStack.Push();` and `modelMatrixStack.Pop();`.

		
Scaling an individual sphere	Scaling an assembly of three spheres	Scaling everything

Why does the assembly (or assemblies) go upside-down?

Comment out your scaling term before working on Task 6.

Task 6: Disintegration

As we discussed in class, the fragment shader is used to colour fragments in the framebuffer. The fragment shader has an interesting keyword called `discard`. Whenever a fragment shader encounters this keyword, the fragment shader returns without setting a colour. We can use this to create nifty visual effects.

As in the last lab, the client program includes a timer used to store the elapsed time since the program was initialised. This elapsed time is being passed to the fragment shader as a variable *t*. Let's use this variable in conjunction with the `discard` keyword to create a disintegration effect!

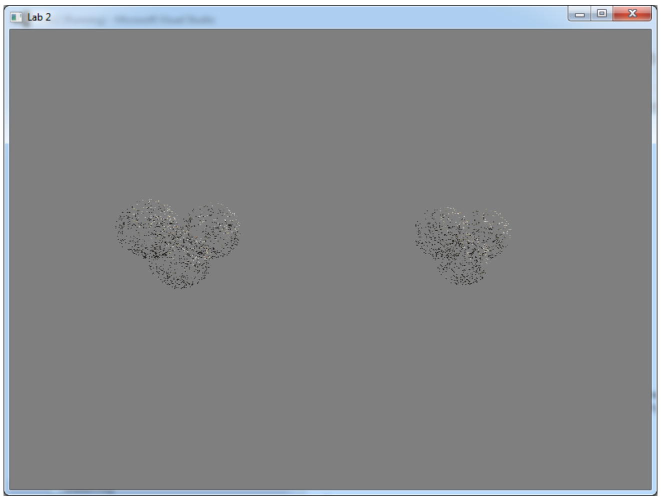
In the fragment shader, the line

```
vec4 vTexColour = texture(sampler0, vTexCoord);
```

is reading a colour from an image used to colour the sphere (this is known as *texture mapping*, to be discussed in Lecture 5). vTexColour is a four-dimensional vector, representing colour in red, green, blue, alpha (RGBA) format. Add the following two lines to the fragment shader after the texture colour is read, and then run the program.

```
if (vTexColour.r < fract(t/5))
       discard;
```

Based on what you see, why does the code above achieve the disintegration effect? Note: if you're not sure what `fract` does, you can look it up here: http://www.opengl.org/sdk/docs/manglsl/xhtml/fract.xml



Now modify the code the product a *materialisation* effect, that is, have the spheres emerge from nothingness!