UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Modelling Transformations

In this lab, we will look at modelling transformations using the GLM library. We will also look at determining the frame time and frame rate of your OpenGL programs.

Modelling Transformations

Transformations are performed in the vertex shader as per-vertex operations. However, unlike vertex attributes that are typically different for each vertex, modelling transformations are, in general, the same for all vertices of an object/model. As such, they are not passed as input using the *in* keyword, but are instead passed to a shader as *uniform* variables. Uniform variables are implicitly constant within a shader. Attempting to change them will result in a compiler error.

Have a look at the code in the modelTransform.vert shader, the model transformation matrix is declared as:

```
uniform mat4 uModelMatrix;
```

In the shader's main function, each vertex position is transformed by the model matrix:

```
    gl_Position = uModelMatrix * vec4(aPosition, 1.0f);
```

Now look at the code in lab3a.cpp. We will use the GLM library for vectors, matrices and their operations. To do this, we need to include the following headers:

```
#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
//using namespace glm;
```

The transform.hpp header includes GLM transformation functions that we will use later. You can uncomment "using namespace glm" if you want to avoid using glm::. However, glm:: will be used in the code to show you that these functions/structs belong to the GLM library.

We will render three objects. The three objects will be rendered using the same vertices, but different transformations. We will use a C++ map data structure to store the model transformation matrices.

```
std::map<std::string, glm::mat4> gModelMatrix;  // object model matrices
```

In the init() function, the model transformation matrices are initialised to the identity matrix using:

```
gModelMatrix["Object1"] = glm::mat4(1.0f);
```

```
gModelMatrix["Object2"] = glm::mat4(1.0f);
gModelMatrix["Object3"] = glm::mat4(1.0f);
```

We leave the matrix for the first object as the identity matrix, but multiply the other two matrices with transformation matrices generated using the GLM library's transformation functions:

```
gModelMatrix["Object2"] *= glm::translate(glm::vec3(0.5f, 0.0f, 0.0f));
gModelMatrix["Object2"] *= glm::rotate(glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
gModelMatrix["Object2"] *= glm::scale(glm::vec3(0.5f, 0.5f, 1.0f));

gModelMatrix["Object3"] *= glm::scale(glm::vec3(0.5f, 0.5f, 1.0f));
gModelMatrix["Object3"] *= glm::rotate(glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
gModelMatrix["Object3"] *= glm::translate(glm::vec3(0.5f, 0.0f, 0.0f));
```

Note that even though the values in the transformation functions are the same, the multiplication order is different. In the rotate function, the **glm::radians()** function converts degrees to radians, since the **glm::rotate()** function accepts angles in radians.

In the render_scene() function, the three objects are rendered using:

```
glBindVertexArray(gVAO);    // make VAO active

// object 1
gShader.setUniform("uModelMatrix", gModelMatrix["Object1"]);  // set model matrix
glDrawArrays(GL_TRIANGLES, 0, 6);

// object 2
gShader.setUniform("uModelMatrix", gModelMatrix["Object2"]);  // set model matrix
glDrawArrays(GL_TRIANGLES, 0, 6);

// object 3
gShader.setUniform("uModelMatrix", gModelMatrix["Object3"]);  // set model matrix
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Since the same VAO is used for all three objects, we only need to bind it once to set the current state. Before the vertices are rendered using **glDrawArrays()**, we first pass the respective model transformation matrix to the shader program as a uniform variable using the ShaderProgram's **setUniform()** function.

Compile and run the program.

You should see three objects (squares), each consisting of two triangles. The objects are at different locations, and have different orientations and sizes. The first object at the origin has not been transformed (since all vertices were multiplied by the identity matrix). For the other two objects, the same transformations were applied, but in a different order. Note that the **order of transformations makes a difference** because **matrix multiplication is not commutative**. Hence, objects 2 and 3 are rendered at different locations, with different orientations and sizes.

Transformation using Keyboard Input (Add the code in this section yourself)

In this section, we want to add code to apply transformations to an object using on keyboard input. We will use the polling approach for this. To avoid missing keyboard input, add the following GLFW function in the main function. You can do this after the **glfwSetKeyCallback()** function:

```
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
```

A function to update the scene has been added to the rendering loop:

```
update_scene(window);
```

Define a set of transformation sensitivities as global variables:

```
const float gTranslateSensitivity = 0.05f;
const float gRotateSensitivity = 0.05f;
const float gScaleSensitivity = 0.05f;
```

Note that you can change the sensitivity values to change the transformation speed.

Now, look at the update_scene() function. It does not contain much other than some variable declarations:

```
glm::vec3 moveVec(0.0f);
glm::vec3 scaleVec(1.0f);
float rotateAngle = 0.0f;
```

These variables will be used to translate, scale and rotate the object. To update these variables based on keyboard input, add the following:

```
if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
	moveVec.x -= gTranslateSensitivity;
if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
	moveVec.x += gTranslateSensitivity;
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
	moveVec.y += gTranslateSensitivity;
if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
	moveVec.y -= gTranslateSensitivity;

if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS)
	rotateAngle += gRotateSensitivity;
if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS)
	rotateAngle -= gRotateSensitivity;

if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS)
{
	scaleVec.x += gScaleSensitivity;
	scaleVec.y += gScaleSensitivity;
}
if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS)
{
	scaleVec.x -= gScaleSensitivity;
	scaleVec.y -= gScaleSensitivity;
}
```

Next, update object 1's model matrix based on the updated variables:

```
	gModelMatrix["Object1"] *= glm::translate(moveVec)
		* glm::rotate(rotateAngle, glm::vec3(0.0f, 0.0f, 1.0f))
		* glm::scale(scaleVec);
```

Compile and run the program.

Try pressing the respective keys to observe changes to object 1's position, orientation and size.

Note that we are using continuous input for this (i.e. if we press and hold a key, it will keep performing an action), as opposed to once-off input (i.e. if we press and hold a key, it will only perform the action once). Therefore, it is easier to use the polling approach rather than using the keyboard callback function. The keyboard callback function approach is useful for once-off input.

While we can use the keyboard callback function with GLFW_REPEAT, many keyboards have a delay before triggering a key repeat. To observe this, add the following the key_callback() function to scale the object when the I or K keys are pressed:

```
if (key == GLFW_KEY_I && action == GLFW_REPEAT)
{
	gModelMatrix["Object1"] *= glm::scale(glm::vec3(gScaleSensitivity + 1.0f,
		gScaleSensitivity + 1.0f, 1.0f));
	return;
}
if (key == GLFW_KEY_K && action == GLFW_REPEAT)
{
	gModelMatrix["Object1"] *= glm::scale(glm::vec3(-gScaleSensitivity + 1.0f,
		-gScaleSensitivity + 1.0f, 1.0f));
	return;
}
```

Run the program, press and hold the I or K key. On many keyboards, there will be a delay before key repeat is triggered.

With the program running, rotate the object, then translate it. Notice that the translation is in the rotated direction. The reason for this is because the matrix is updated based on contents of the previous matrix (i.e. *=):

```
gModelMatrix["Object1"] *= glm::translate(moveVec)
	* glm::rotate(rotateAngle, glm::vec3(0.0f, 0.0f, 1.0f))
	* glm::scale(scaleVec);
```

What if you do not want the object to translate in the direction of its rotation, but rather to translate about the coordinate system's *x* and *y* axes?

Frame Time and Frame Rate (Add the code in this section yourself)

In interactive computer graphics, it is useful to keep track of the number of frames that are rendered per second (i.e. frames per second), as well as the time per frame (i.e. frame time).

Define the following global variables to store frame rate and time respectively:

```
float gFrameRate = 60.0f;
float gFrameTime = 1 / gFrameRate;
```

In the main function, just before the rendering loop, add and initialise the following variables to keep track of the time of the last update, the time since the last update and the number of frames since the last update:

```
double lastUpdateTime = glfwGetTime();    // last update time
double elapsedTime = lastUpdateTime;      // time since last update
```

```
int frameCount = 0;                         // number of frames since last update
```

The **glfwGetTime()** function returns time in seconds since GLFW was initialised.

At the end of the rendering loop, add the following code:

```
frameCount++;
elapsedTime = glfwGetTime() - lastUpdateTime;   // time since last update

// if elapsed time since last update > 1 second
if (elapsedTime > 1.0)
{
	gFrameTime = elapsedTime / frameCount;   // average time per frame
	gFrameRate = 1 / gFrameTime;             // frames per second

	std::string str = "Lab 3 - FPS: " + std::to_string(gFrameRate)
		+ ", FT: " + std::to_string(gFrameTime);
	glfwSetWindowTitle(window, str.c_str());

	lastUpdateTime = glfwGetTime();          // set last update time to current time
	frameCount = 0;                          // reset frame counter
}
```

What this does is increment the frame count for each frame, and compute the time since the last frame time update. If it has been 1 second or more since the last update, calculate the new average frame time and frame rate values. Display this on the window title, before updating the time since the last update and resetting the frame count to zero.

Compile and run the program. After 1 second, the frame rate and frame time values will be displayed on the window title.

NOTE that on some systems, **vertical sync may be enabled by default**, whereas on other systems it may be disabled by default. This setting will affect the frame rate and frame time values. The following explanation assumes that your system has vertical sync enabled. Nevertheless, even if this is not the case, it is important to understand the difference especially since you want your program to run consistently on different systems.

On systems where the vertical sync is enabled by default, the frame rate will usually be ~60 (on some systems like laptops this may be lower, e.g., ~30, on battery mode to conserve energy) and the frame time will be ~0.016667. This is because the program is currently set up to swap buffers on the monitor's vertical sync signal (if enabled), which was set using:

```
glfwSwapInterval(1);              // swap buffer interval
```

Since most computer monitors have a 60Hz refresh rate, this explains the frame rate and frame time values. If your display device has a different refresh rate, then the values may be different. [Note however that if your system has the vertical sync disabled by default, it won't matter what you set the swap buffer interval to, as your system will ignore this. If that's the case, the rendering loop will simply run as fast as your system will allow.]

Rather than waiting for the vertical sync signal to swap buffers, this can be disabled by setting the value in **glfwSwapInterval()** to 0:

```
glfwSwapInterval(0);              // swap buffer interval
```

Compile and run the program.

The rendering loop will now run as fast as your system will allow, and the frame rate and frame time values will fluctuate. [If your system has vertical sync disabled by default, you won't notice any difference because your system ignores the swap buffer interval anyway.]

Try using keyboard input to transform object 1, you may notice a large increase in transformation speed due to the increased frame rate.

To make the transformation speed (and any updates/animations may you have) more consistent, movements/animations should be scaled by the frame time.

Exercises

By now, you should have an understanding of transformations using matrices. Try modifying the program to do the following:

- Make the transformation speed more consistent irrespective of frame rate
- Make object 2 orbit object 1
- Make object 3 orbit object 2
- Object 1 currently translates based on the direction of its rotation, how would you get it to translate about the coordinate system's x and y axes?

References

Among others, much of the material in this lab was sourced from:

- https://www.khronos.org/opengl/wiki/
- https://learnopengl.com/