UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Transformations

In this tutorial, we will look at modelling transformations using the GLM library. We will also look at adding a simple Graphical User Interface (GUI) library to your OpenGL programs.

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 ModelSpaceVS.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);
```

Go back to the code in the Tutorial3a.cpp file. The following is used to store the model transformation matrix's location in the shader and the actual transformation matrices:

```
GLuint g_modelMatrixIndex = 0;     // location in shader
glm::mat4 g_modelMatrix[3];        // object's model matrix
```

In this program, we will render three objects. All three objects will be rendered using the same vertices, but will use different transformations. Hence, we declare an array of three model transformation matrices. Note that the matrix is from the GLM library which was included at the top of the code:

```
#include <glm/glm.hpp>
#include <glm/gtx/transform.hpp>
using namespace glm; // to avoid having to use glm::
```

The transform.hpp header includes GLM transformation functions that we will use later. Since we are "using namespace glm", we don't need the glm::. However, it will be used in the code to show you that these functions/structs are from the GLM library.

In the init function, the model transformation matrix's location is obtained using the OpenGL **glGetUniformLocation** function (note that for attributes it was **glGetAttributeLocation**):

```
g_modelMatrixIndex = glGetUniformLocation(g_shaderProgramID, "uModelMatrix");
```

The transformation matrices are initialised to the identity matrix using:

```
g_modelMatrix[0] = glm::mat4(1.0f);
g_modelMatrix[1] = glm::mat4(1.0f);
g_modelMatrix[2] = 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:

```
g_modelMatrix[1] *= glm::translate(glm::vec3(0.5f, 0.0f, 0.0f));
g_modelMatrix[1] *= glm::rotate(glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
g_modelMatrix[1] *= glm::scale(glm::vec3(0.5f, 0.5f, 1.0f));

g_modelMatrix[2] *= glm::scale(glm::vec3(0.5f, 0.5f, 1.0f));
g_modelMatrix[2] *= glm::rotate(glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
g_modelMatrix[2] *= 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 for the two model transformation matrices. In the rotate functions, the **glm::radians** function converts degrees to radians, since the **glm::rotate** function accepts angles in radians.

Finally, we render the three objects:

```
	glBindVertexArray(g_VAO);		// make VAO active
// object 1
	// set uniform model transformation matrix
	glUniformMatrix4fv(g_modelMatrixIndex, 1, GL_FALSE, &g_modelMatrix[0][0][0]);

	glDrawArrays(GL_TRIANGLES, 0, 6);

// object 2
	// set uniform model transformation matrix
	glUniformMatrix4fv(g_modelMatrixIndex, 1, GL_FALSE, &g_modelMatrix[1][0][0]);

	glDrawArrays(GL_TRIANGLES, 0, 6);

// object 3
	// set uniform model transformation matrix
	glUniformMatrix4fv(g_modelMatrixIndex, 1, GL_FALSE, &g_modelMatrix[2][0][0]);

	glDrawArrays(GL_TRIANGLES, 0, 6);
```

Since we are using the same VAO for all three objects, we only need to bind it once to set the current state. Before we render the respective arrays using **glDrawArrays**, we first pass the respective model transformation matrix to the shader program as a uniform variable using **glUniformMatrix4fv**. The "4" in the function indicates a 4x4 matrix (i.e. 16 values), the "f" indicates that the values are floats and the "v" indicates that we are passing the values via a pointer.

Compile and run the program.

You should see three objects (i.e. squares), each consisting of two triangles. All three objects are at different locations, and have different orientations and sizes. The first object at the origin has undergone no transformation (since all vertices were multiplied by the identity matrix). For the other two objects, the same transformations were applied, but in different orders. Note that the **order of transformations makes a difference** because **matrix multiplication is not commutative**. Hence, the two objects are rendered at different locations, orientations and sizes.

Transformations using Keyboard Input

Open the Tutorial3b project and have a look at the code in Tutorial3b.cpp. The other files are the same as Tutorial3a. In this program, we want to transform the object using keyboard input. We will use the polling approach for this. Hence, the following GLFW function is added in the main function:

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

In the rendering loop, an update scene function is also added to update the scene:

```
update_scene(window);
```

In the init function, the model matrix is initialised to the identity matrix:

```
g_modelMatrix = glm::mat4(1.0f);
```

A number of sensitivities are defined for the respective transformations:

```
#define MOVEMENT_SENSITIVITY 0.05f
#define ROTATION_SENSITIVITY 0.05f
#define SCALE_SENSITIVITY 0.05f
```

Note that on some systems, you will have to lower these values. Otherwise the object will move very fast. The next tutorial will explain why this is the case and will provide a solution for it.

Next, look at the update_scene function. First, some variables to store the respective transformations are declared:

```
// declare variables to transform the object
glm::vec3 moveVec(0.0f, 0.0f, 0.0f);
glm::vec3 scaleVec(1.0f, 1.0f, 1.0f);
float rotateAngle = 0.0f;
```

These variables are updated based on keyboard input:

```
// update translation vector
if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
	moveVec.x -= MOVEMENT_SENSITIVITY;
if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
	moveVec.x += MOVEMENT_SENSITIVITY;
if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
	moveVec.y += MOVEMENT_SENSITIVITY;
```

```
if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
	moveVec.y -= MOVEMENT_SENSITIVITY;

// update rotation angle
if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS)
	rotateAngle += ROTATION_SENSITIVITY;
if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS)
	rotateAngle -= ROTATION_SENSITIVITY;

// update scale vector
if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS)
{
	scaleVec.x += SCALE_SENSITIVITY;
	scaleVec.y += SCALE_SENSITIVITY;
}
if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS)
{
	scaleVec.x -= SCALE_SENSITIVITY;
	scaleVec.y -= SCALE_SENSITIVITY;
}
```

Once the variables are updated, the model transformation matrix is updated based on the variables:

```
g_modelMatrix *= glm::translate(moveVec)
	* glm::rotate(rotateAngle, glm::vec3(0.0f, 0.0f, 1.0f))
	* glm::scale(scaleVec);
```

Compile and run the program.

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 illustrate this, the following is added to the `key_callback` function to scale the object when the I or K keys are pressed:

```
if (key == GLFW_KEY_I && action == GLFW_REPEAT)
{
	g_modelMatrix *= glm::scale(glm::vec3(SCALE_SENSITIVITY + 1.0f,
		SCALE_SENSITIVITY + 1.0f, 1.0f));
	return;
}
if (key == GLFW_KEY_K && action == GLFW_REPEAT)
{
	g_modelMatrix *= glm::scale(glm::vec3(-SCALE_SENSITIVITY + 1.0f, -
		SCALE_SENSITIVITY + 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 the contents of the previous matrix (i.e. *=):

```
g_modelMatrix *= glm::translate(moveVec)
        * glm::rotate(rotateAngle, glm::vec3(0.0f, 0.0f, 1.0f))
        * glm::scale(scaleVec);
```

This is fine, if that is what you want. However, 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? We can do so by modifying the code slightly.

Change the variables to static variables (to maintain their state between function calls):

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

Remove the * from *=:

```
g_modelMatrix = glm::translate(moveVec)
        * glm::rotate(rotateAngle, glm::vec3(0.0f, 0.0f, 1.0f))
        * glm::scale(scaleVec);
```

Compile and run the program. With the program running, rotate the object then translate it. Notice the difference?

In this version, we are updating the translation, rotation and scaling values and overwriting the transformation matrix by creating matrices from the updated translation, rotation and scaling values, rather than updating the matrix by multiplying it with its previous contents.

Updates based on Frame Time

Open Tutorial3c. Time is an important factor that must be considered when attempting to update the graphics in a scene. This is because different systems may operate differently, and if so, your program will update at different update rates when it is run on different systems. If time is not used to update your scene, its behaviour will be inconsistent.

If your system has vertical sync enabled, you can see the difference by opening the *previous program*, i.e. ***Tutorial3b***. Compile and run the program. Move and rotate the colour cube and remember the speed at which it moves.

Now set **glfwSwapInterval** to 0:

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

Compile and run the program. Move and rotate the colour cube. Notice that it now moves a lot faster. [Note that if your system has vertical sync disabled by default, you won't notice any difference because your system ignores the swap buffer interval anyway].

The reason for this is because the code in the update_scene function was written in a way that does not cater for frame time variations. It is only consistent if the rendering loop updates at the same update rate on different systems. However, this cannot be guaranteed as different systems may operate differently. Hence, a better approach is to update the scene based on the frame time (i.e. the time that has elapsed since the last update).

This is done in the update_scene function in Tutorial3c.cpp. Note that the frame time is passed to the function and all updates are multiplied by the frame time. E.g.:

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

Try compiling and running the program with glfwSwapInterval set to 0 (you may have to wait a few seconds for the frame time to stabilise), and again when glfwSwapInterval is set to 1. The movement will more consistent, despite the different update rates.

Adding a Simple GUI

To add basic GUI functionality to your OpenGL programs, we will look at the AntTweakBar library. This is a convenient library that is typically used for adding graphical interfaces so programmers can easily see the effects of changing certain variables in an interactive graphics program.

AntTweakBar provides a unique RotoSlider for rapid editing of numerical values. An image of this is shown in Figure 1. Instead of clicking on the '-' and '+' buttons to modify a numerical value, the RotoSlider allows one to click on the roto button (the RotoSlider will appear while the mouse button is held), and while the mouse button is held, to rotate the RotoSlider counter-clockwise or clockwise to increase or decrease the numeric value. The lines in the circle represent the minimum and maximum allowable values. The different colour on the circumference of the circle shows the original position on the circle and the amount that the slider has moved by.

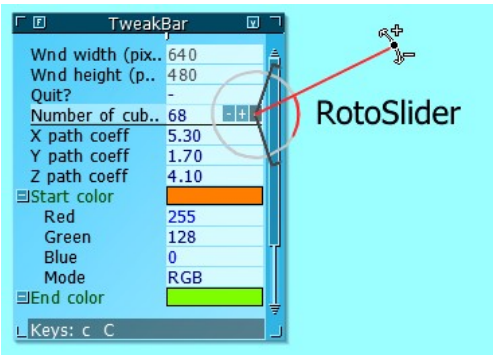


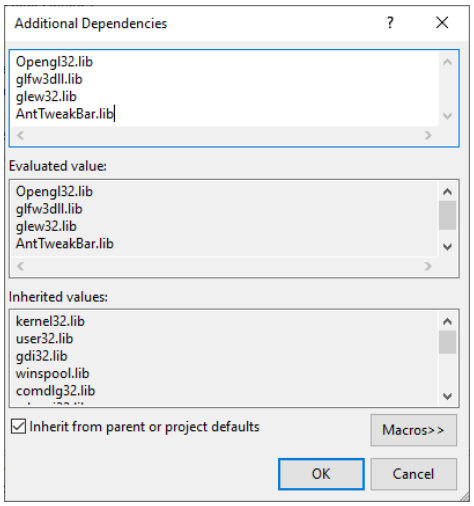
Figure 1: RotoSlider

We will now look at how we can add a tweakbar to an OpenGL program that uses the GLFW library. Open the Tutorial3d project. Note that while the AntTweakBar library contains numerous functions, we will only look at some of the important ones.

To include the library:

```
#include <AntTweakBar.h>
```

Note: To use AntTweakBar, your program will need the AntTweakBar.dll file. This is located in the same directory as the .cpp files in Tutorial3d. However, if you want to distribute your program, the .dll file needs to be in the same directory as the application program's .exe file. You will also need to add the library in project settings:



Now look in the main function. One of the first things to do is to initialise the library:

```
TwInit(TW_OPENGL_CORE, NULL);
```

This initialises the library using the OpenGL Core profile (OpenGL 3.2 and above), as oppose to the compatibility/legacy profile.

Next, the tweak bar needs to know the size of the graphics area:

```
TwWindowSize(g_windowWidth, g_windowHeight);
```

Note that the GLFW window was also created with these window dimension variables. The next two functions are to hide the help menu and to set the tweak bar's font size to large:

```
TwDefine(" TW_HELP visible=false ");      // disable help menu
TwDefine(" GLOBAL fontsize=3 ");          // set large font size
```

Try commenting these out and you will see what these functions do. There are 3 font sizes, with 3 being the largest size. The font size can also be altered when the program is running by clicking on the 'F' located at the top-left hand corner of the tweak bar (this can be seen in Figure 1).

The next function creates a tweakbar (you can obviously create more than one). The tweak bar's name is set as "Main", you can call it anything you want. Multiple tweak bars must have different names.

```
TweakBar = TwNewBar("Main");
```

A pointer to the tweak bar was previously declared at the start of the main function:

```
TwBar *TweakBar;
```

The library uses a definition string to modify the behaviour of its elements. An example of changing the behaviour can be seen in the next function, using a definition string that contains several bar parameters:

```
TwDefine(" Main label='My GUI' refresh=0.02 text=light size='220 250' ");
```

Notice that the parameter list in the function is simply one long string, where you can add whatever parameters you want within the string.

- The first parameter `Main` is the tweak bar's name. This was the name that we used when we created the tweak bar (since you can have multiple tweak bars, it needs to know which bar you are referring to).
- The next parameter `label='My GUI'` sets the tweak bar's title (if no label is set, the bar's title will default to using the name, i.e. "Main" in this case).
- This is followed by `refresh=0.02`, which sets the tweak bar's refresh rate (in seconds) to slightly more than 1/60 (i.e. 0.016667 seconds) of a second.
- The next parameter `text=light` sets the font colour. This value can either be "light" or "dark", and should depend on the background colour (i.e. use light for a dark background and dark for a light background).
- Finally, the size of the tweak bar is set using `size='220 250'`, this is measured in terms of pixels.

Besides the parameters listed above, there are a variety of other parameters that can be set for a tweak bar. Some of the more important ones include ***color*** for the bar's colour, ***alpha*** to control the bar's translucency/opacity, ***position*** to specify the bar's location in the application window (measured from the upper-left corner). There are other parameters for controlling the bar's behaviour when moving the bar, iconifying the bar, and controlling the behaviour of multiple bars when they overlap.

Now that we have created a tweak bar, let's fill it with entries. The first entry shows how to use Boolean values (like a 'checkbox'), to toggle between wireframe and solid-fill polygon mode:

```
TwAddVarRW(TweakBar, "Wireframe", TW_TYPE_BOOLCPP, &g_wireFrame, " group='Display' ");
```

The **TwAddVarRW** function is for adding a variable to a tweak bar for Reading and Writing (hence the RW).

The first parameter is a pointer to the specific tweak bar; the second parameter is the entry's name (must be a unique name); the third is the entry's datatype (a Boolean value in this case); the fourth is a pointer to the variable; and the last parameter is the variable parameters definition string (much like the bar parameters definition string that was used for specifying the behaviour of the tweak bar).

Note that the global variables that are used for the tweak bar are declared near the top of the code as:

```
GLuint g_windowWidth = 800;
GLuint g_windowHeight = 800;
bool g_wireFrame = false;
float g_backgroundColor[] = { 0.0f, 0.0f, 0.0f };
```

Note that no ***label*** parameter was set in the parameter definition string, hence, the tweak bar will display the entry's name by default. The ***group*** parameter is used for hierarchical organisation. Since the groupname called "Display" does not exist yet, this group will be created.

To get this to work, mouse button input must be passed to the tweak bar. This is done in the mouse call back functions:

```
static void cursor_position_callback(GLFWwindow* window, double xpos, double ypos)
{
	// pass mouse data to tweak bar
	TwEventMousePosGLFW(xpos, ypos);
}

static void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
	// pass mouse data to tweak bar
	TwEventMouseButtonGLFW(button, action);
}
```

Which were set as the GLFW mouse callback functions (in the main function) using:

```
glfwSetCursorPosCallback(window, cursor_position_callback);
glfwSetMouseButtonCallback(window, mouse_button_callback);
```

For the wireframe/solid-fill polygon mode to work, the following was added in the rendering loop before the scene is rendered:

```
	if (g_wireFrame)
		glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);


	render_scene();
```

Make sure you set the polygon mode to solid-fill mode after rendering the scene.

```
	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

This is because you do not want to render the tweak bar in wireframe mode. The next command renders the tweak bars. Note that this is done before the buffer swap function:

```
	TwDraw();
```

Before exiting the program, the tweak bar library should be closed using:

```
TwTerminate();
```

Compile and run the program. Try clicking on the wireframe entry to toggle the polygon render mode.

Next, we will look at a tweak bar entry for adjusting colour:

```
TwAddVarRW(TweakBar, "BgColor", TW_TYPE_COLOR3F, &g_backgroundColor,
       " label='Background Color' group='Display' opened=true ");
```

This entry will be used for changing the background colour. Note the datatype that is used is `TW_TYPE_COLOR3F`, which allows for changing the values of an array of 3 floating point numbers to be used for colour, i.e.:

```
float g_backgroundColor[] = { 0.0f, 0.0f, 0.0f };
```

At the start of the render_scene function, the clear colour is set using:

```
glClearColor(g_backgroundColor[0], g_backgroundColor[1], g_backgroundColor[2], 1.0f);
```

This time, we set the ***label*** parameter in the definition string. When you run the program, you will see that the name that is displayed is the name specified in the label and not the default entry's name. This entry also belongs to the "Display" group, which already exists. The final parameter, ***opened*** defines whether the colour list is expanded or minimised. Try changing the value to false and you will see the difference.

Compile and run the program. Use the RotoSliders, or the '-' and '+', to change the background colour.

Frame Rate and Frame Time

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).

In the main function, just before the rendering loop, the following variables are declared to keep track of various frame and time information:

```
double lastUpdateTime = glfwGetTime();
double elapsedTime = lastUpdateTime;
float frameTime = 0.0f;
int frameCount = 0;
```

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

At the end of the rendering loop, the following code has been added:

```
frameCount++;
```

```
elapsedTime = glfwGetTime() - lastUpdateTime;   // current time - last update time

if (elapsedTime >= 1.0f)    // if time since last update >= to 1 second
{
	frameTime = static_cast<float>(1.0f / frameCount);     // calculate frame time

	string str = "FPS = " + to_string(frameCount) + "; FT = " + to_string(frameTime);

	glfwSetWindowTitle(window, str.c_str()); // update window title

	frameCount = 0;                    // reset frame count
	lastUpdateTime += elapsedTime;     // update last update time
}
```

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, then calculate the new average frame time and set the window title to display the frames per second and frame time values, before resetting the frame count and updating the time since the last update.

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

NOTE that on some systems, the **vertical sync is enabled by default**, whereas on other systems it may be disabled by default. This setting will affect the frames per second 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 frames per second will be ~60 (on some 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 frames per second 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 update as fast as your system will allow.]

Rather than waiting for the vertical sync signal to swap buffers and render the scene, this can be disabled by setting the value to 0:

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

Compile and run the program. The rendering loop will now update as fast as your system will allow, and the frames per second and frame time values will fluctuate, and will be very different from the previous values. [If your system has vertical sync disabled by default, you won't notice any difference because your system ignores the swap buffer interval anyway.]

As an example of how use the tweak bar to display information. The following code displays the frame rate and frame time. Notice the function name's RO stands for Read-Only:

```
TwAddVarRO(TweakBar, "FPS", TW_TYPE_INT32, &FPS, " group='Frame' ");
TwAddVarRO(TweakBar, "Frame Time", TW_TYPE_DOUBLE, &frameTime,
       " group='Frame' precision=4 ");
```

The first displays a 32-bit integer value, while the second displays a double value. They both belong to group "Frame". For the second entry, the ***precision*** is set to 4, which means that 4 significant digits will be displayed after the decimal place.

We can define the "Frame" group to be a sub-group of the "Display" group using:

```
TwDefine(" Main/Frame group='Display' ");
```

The above function's parameter string states that the "Frame" group in the "Main" tweak bar is a sub-group of the "Display" group. Try commenting this out and you will see the difference.

The next entry simply adds a separation line:

```
TwAddSeparator(TweakBar, NULL, NULL);
```

Compile and run the program.

Things to do

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

- Create an object that orbits around another object

References

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

- http://anttweakbar.sourceforge.net/doc/
- https://www.khronos.org/opengl/wiki/
- http://www.opengl-tutorial.org/
- http://ogldev.atspace.co.uk/index.html