SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

User Interface

In this lab, we will look at adding a user interface to OpenGL programs using the AntTweakBar library.

Index Buffer Object (IBO)

In examples, we defined separate vertices for each triangle. However, it is possible to render primitives that use share vertices using index buffers. This is illustrated in Figure 1 below:

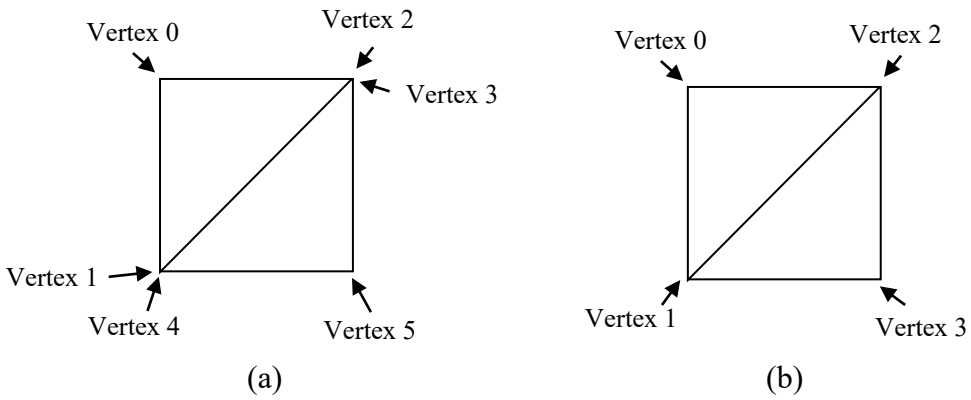

Figure 1

In Figure 1(a), the square is made up of two triangles that do not share any vertices. Therefore, a total of 6 vertices must be specified. Whereas for the square in Figure 1(b), vertex 2 and 3 are shared between both triangles, thus reducing the total number of vertices to 4.

Open the Lab4 project and look at the code in lab4.cpp. In the init() function, where the vertices are specified, notice that only 4 vertices are defined:

```
std::vector<GLfloat> vertices = {
       -0.2f, 0.2f, 0.0f,   // vertex 0: position
       1.0f, 0.0f, 0.0f,    // vertex 0: colour
       ...
       ...
       0.2f, -0.2f, 0.0f,   // vertex 3: position
       1.0f, 1.0f, 0.0f,    // vertex 3: colour
};
```

Based on these 4 vertices, the indices for 2 triangles are specified where triangle 1 consists of vertex 0, 1 and 2, while triangle 2 consists of vertex 2, 1 and 3. Refer to the illustration in Figure 1(b).

```
std::vector<GLuint> indices = {
```

```
        0, 1, 2,      // triangle 1
        2, 1, 3,      // triangle 2
};
```

An index buffer object (IBO) has to be used to store the index data. As such, an identifier for the IBO is declared.

```
GLuint gIBO = 0;            // index buffer object identifier
```

Similar to a VBO, an IBO is created and the data is buffered. Note that we bind the IBO to type GL_ELEMENT_ARRAY_BUFFER:

```
glGenBuffers(1, &gIBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, gIBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) * indices.size(), &indices[0],
GL_STATIC_DRAW);
```

Also, note that the IBO is set as the state of a VAO.

```
glGenVertexArrays(1, &gVAO);                    // generate unused VAO identifier
glBindVertexArray(gVAO);                        // create VAO
glBindBuffer(GL_ARRAY_BUFFER, gVBO);            // bind the VBO
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, gIBO);    // bind IBO
...
```

When rendering based on an IBO, use **glDrawElements()** instead of glDrawArrays(). This is in the render_scene() function:

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

The second parameter is the number of elements to be rendered; two triangles specified using 6 indices. The third parameter is the type, and the last parameter is the element offset.

Compile and run the program.

Notice that the shared vertices between the two triangles cannot have distinct colours as they share the same colour attribute.[1]

Graphics User Interface (GUI)

To add a basic graphical user interface (GUI) to OpenGL programs, we will use the AntTweakBar library. This is convenient for adding an interface so users 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 2. 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 left 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

[1] You may be wondering whether GL_TRIANGLE_STRIP can achieve the same thing. The short answer is "yes" in the case of this simple example, but "no" for more complicated cases. You will see an example of a colour cube in the next lab where a single triangle strip will not work.

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.

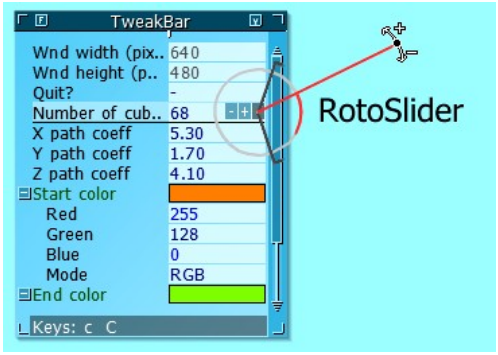

Figure 2: RotoSlider

To use AntTweakBar, its header must be included:

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

Windows only:

The corresponding dynamic linked libraries (AntTweakBar.dll) must be located in current directory.

To initialise the library, the following is in the main() function:

```
TwInit(TW_OPENGL_CORE, nullptr);
TwBar* tweakBar = create_UI("Main");
```

The second line calls the create_UI() function that we will use to create and populate a tweak bar.

To delete and uninitialise before exiting the program, the following is at the end of the main() function:

```
TwDeleteBar(tweakBar);
TwTerminate();
```

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

```
static void cursor_position_callback(GLFWwindow* window, double xpos, double ypos)
{
	// pass cursor position to tweak bar
	TwEventMousePosGLFW(static_cast<int>(xpos), static_cast<int>(ypos));
}

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

Which were set as the GLFW mouse callback functions:

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

Tweak Bar Elements (Add the code in this section yourself)

First, add some global variables that we will use to control the display:

```
bool gWireframe = false;                  // wireframe mode on or off
glm::vec3 gBackgroundColor(0.2f);  // variables to set the background color
glm::vec3 gMoveVec(0.0f);                 // object translation
```

Now, go to the create_UI() function. You can see that the function creates a new tweak bar and returns the tweak bar handle.

Add the following so the tweak bar knows the size of the graphics window:

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

Note that the GLFW window was also created with these window dimension variables. Add the next two functions 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
```

To display the tweak bar, go to the main() function and add the following to the rendering loop AFTER the call to the render_scene() function, but BEFORE swapping buffers:

```
TwDraw();                          // draw tweak bar
```

AntTweakBar use a definition string to modify the behaviour of its elements. Go back to the create_UI() function and add the following:

```
TwDefine(" Main label='MyGUI' refresh=0.02 text=light size='220 320' ");
```

Notice that the parameter list in the function is simply one long string, where you can add certain 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='MyGUI'` 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='220 320'`, 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.

Compile and run the program to see the tweak bar.

Now that we have created a tweak bar, let's fill it with entries.

The examples we will add show 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(twBar, "Frame Rate", TW_TYPE_FLOAT, &gFrameRate, " group='Frame Stats' 
precision=2 ");

TwAddVarRO(twBar, "Frame Time", TW_TYPE_FLOAT, &gFrameTime, " group='Frame Stats' ");
```

The first argument is a pointer to the specific tweak bar; the second argument is the entry's name (must be a unique name); the third is the entry's datatype (a floating point value in these cases); the fourth is a pointer to the variable where the information is stored; 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 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 group name "Frame Stats" does not exist yet, this group will be created.

Both entries display float values and belong to the group "Frame Stats". For the first entry, the ***precision*** is set to 2, which means that 2 significant digits will be displayed after the decimal place.

The next entry shows how to use Boolean values (like a 'checkbox'), to toggle between wireframe and solid-fill polygon mode. Add the following:

```
TwAddVarRW(twBar, "Wireframe", TW_TYPE_BOOLCPP, &gWireframe, " group='Display' ");
```

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

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

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

render_scene();
```

Then, 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.

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(twBar, "BgColor", TW_TYPE_COLOR3F, &gBackgroundColor, " 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.

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.

To allow this to work, add the following code in the update_scene() function to set the clear colour:

```
glClearColor(gBackgroundColor.r, gBackgroundColor.g, gBackgroundColor.b, 1.0f);
```

Compile and run the program.

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

The next entry simply adds a separation line:

```
TwAddSeparator(TweakBar, nullptr, nullptr);
```

Add the following to control the object's translation:

```
TwAddVarRW(twBar, "Move: x", TW_TYPE_FLOAT, &gMoveVec.x, " group='Tranformation' min=-1.0 max=1.0 step=0.01 ");

TwAddVarRW(twBar, "Move: y", TW_TYPE_FLOAT, &gMoveVec.y, " group='Tranformation' min=-1.0 max=1.0 step=0.01 ");
```

Note that the definition string sets the minimum and maximum values of the variables, and also sets the size per step.

To get these controls to work, add the following to the update_scene() function:

```
gModelMatrix = glm::translate(gMoveVec);
```

Compile and run the program.

Try using the controls to change the object's translation.

Viewport Transformation

Instead of rendering to the entire frame buffer (the default viewport transformation), we can change the viewport transform to set the portion of the frame that we want to render to. This is done using the **glViewport()** function.

Let's say we only want to render to the centre of the frame, we can create a border of 100 pixels from the top, down, left and right boundaries of the frame where we do not want anything to be drawn. To do this, we can add the following:

```
glViewport(100, 100, gWindowWidth - 200, gWindowHeight - 200);
```

Compile and run the program.

The scene will look smaller (as it now fits within a smaller display area). Try moving the object to the sides of the window. Notice that the object will be clipped out when it is moved outside the viewport.

Exercises

By now, you should have an understanding of how to use the tweak bar. Try modifying the program to do the following:

- Add a control to change the object's size
- Add a control to change the object's rotation
- By default, when you resize the graphics window, the scene will only be rendered to the lower left corner. How would you get it to render to the entire window?

References

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

- https://www.khronos.org/opengl/wiki/
- http://anttweakbar.sourceforge.net/doc/