This is a relatively short exercise to get you to try out some basic OpenGL calls and review the concept of linear interpolation.

Using the “intro” sample code as a starting point, make a simple “color-picker” application that works as follows. There is a square on the left side of the window whose corners are red, green, blue, and white, respectively (CCW from lower right), with the colors interpolated as in gl_example2 (draw as GL_TRIANGLE_FAN). On the right side there is a circle. When you click somewhere on the square, the circle turns the color of the pixel where you clicked.

**What to turn in**

Please turn in a zipped Visual Studio project (not the entire solution). The correctness criterion is that the project will build and run on the gaming lab computers when copied into, and then added to, an unmodified intro.zip.

Specifically, the following should work (and you should verify this):

1. Download and unzip intro.zip on one of the machines in the gaming lab.
2. Unzip your submission.
3. Copy your project directory into the intro directory of the intro solution.
4. Open the solution in VS.
5. Right-click on the solution in the Solution Explorer and select “Add -> Existing project…”.
6. Navigate to the project file (.vcxproj) in your project directory to add it to the solution.
7. Build and run your code in VS.

See the README in the intro.zip for tips on creating a new project. Your project should include the appropriate .cpp file, your shaders, and the VS project files. You should clean the project before submission and you can delete the Debug directory and any .user files. You are welcome to copy/paste the existing code but please update any descriptive comments if you do so.

**Details and hints**

As a warm-up, you might first try modifying gl_example2 to make a square instead of a triangle. Order the vertices CCW from lower right, using colors red, green, blue, and white. Draw using GL_TRIANGLE_FAN.

Then, try drawing a second figure, something easy, like another square. Use two different VAOs for the two figures. Getting handles for multiple VAOs is similar to
getting handles for two buffers, as seen on line 32 of example2.cpp. In the display() function, you’ll need to bind one VAO and call glDrawArrays, then bind the other and call glDrawArrays again.

You’ll want to use a different shader program for the two figures. The shader pair from gl_example1 should be fine as is for your square. To draw a solid color, just modify the fragment shader from gl_example0 (as we did in class) so that you can set the color as a uniform variable. (The draw call will use whichever shader program is currently bound.)

Uniform variables can be declared in a shader with the “uniform” keyword. To set the value of a uniform from the C++ code, you’ll need to get the index and then invoke an appropriate form of glUniform. E.g., this would set the values (1.0, 0.0, 0.0, 1.0) to a vec4 uniform called myColor:

```c++
GLuint loc = glGetUniformLocation(program, "myColor");
glUniform4f(loc, 1.0, 0.0, 0.0, 1.0);
```

(The suffix “4f” refers to “four floats”). Alternatively, if you want to use the vec4 data type on the C++ side that is defined in vec.h, you could use the “vector” form that takes a pointer to an array of floats (the “1” means “one four-float vector”):

```c++
vec4 v(1.0, 0.0, 0.0, 1.0);
.glUniform4fv(loc, 1, (GLfloat *) &v)
```

Your circle will actually be a polygon with (e.g.) 24 sides. Remember, it is easy to generate the vertices of a circle at a given angle theta,

\[(x, y) = (r \cos \theta , r \sin \theta)\]

So looping through theta in increments of 15 degrees would give you 24 vertices. (Of course, you’ll need to add an offset to shift to the right, since the above would be centered at the origin).

To determine the mouse position, see the glut_events project. Then you can figure out whether it’s inside your square, and if so, which of the two triangles it is in. (When you draw with GL_TRIANGLE_FAN there will be two triangles as shown below, assuming that your vertices are ordered CCW starting with lower right.) Find the vertical offset within the triangle and interpolate the R, G, B values along the vertical side of the square, and along the diagonal. Then use the horizontal offset and interpolate between those values.

![Diagram](image.png)