Using the code for intro/gl_example2 as a starting point, create a program that will allow you to experiment with 2D transformations as follows. Initially, draw a triangle with the vertices

\[
\text{vec3( 0.0, 0.0, 1.0 ), vec3( 0.5, 0.0, 1.0 ), vec3( 0.5, 0.25, 1.0 )},
\]

(Note we are treating each triple as a set of 2D homogeneous coordinates, which is why the third component is always 1.) Also draw a set of axes for reference. Make the triangle and axes a different color.

Then add keyboard controls to perform transformations as follows:

r: apply rotation 30 degrees clockwise (about origin)
R: apply rotation 30 degrees ccw (about origin)
s: apply scale factor (0.5, 0.5)
S: apply scale factor (2.0, 2.0)
t: translate by (0.3, 0.2)
T: translate by (-0.3, -0.2)
x: x shear by 0.5y
X: x shear by -0.5y
c: rotate 30 degrees cw about centroid of triangle
C: rotate 30 degrees ccw about centroid of triangle (the centroid of a triangle is just the average of the vertices.)
Notes

1) The simplest (not necessarily the “best”) way to do this is to keep the three vertices of the triangle in a global variable, which you can modify in the keyboard handler, and then update the buffer in the GPU with the changed vertices whenever `display()` is called. For example, if `buffer[0]` is a handle to a GPU buffer containing your vertex data, and if `vertices` is a three-element array containing your transformed vertices, you can use the following in your `display()` function:

   ```cpp
   glBindBuffer( GL_ARRAY_BUFFER, buffer[0] );
   glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices );
   ```

   (where the “0” in the second argument means we have assumed that the vertex data is the first thing in the buffer).

2) In our minds, the vertices of the triangle are 2D homogeneous coordinates. To OpenGL, however, a triple such as `vec3( 0.5, 0.25, 1.0 )` is a point on the plane $z = 1$. To ensure that the triangle is visible when we send the vertices into the pipeline, we should project it down to the plane $z = 0$, which can easily be done in the vertex shader. Modify the given shader `vshader31.glsl` as follows:

   ```cpp
   #version 150

   in  vec4 vPosition;
   in  vec4 vColor;
   out vec4 color;

   void main()
   {
     gl_Position = vPosition;
     gl_Position.z = 0.0;         // project onto plane z = 0
     color = vColor;
   }
   ```

3) You might find it useful to write functions for generating transformation matrices, e.g.

   ```cpp
   mat3 Translate2D(GLfloat x, GLfloat y)
   {
     mat3 m; // creates identity matrix
     m[0][2] = x;
     m[1][2] = y;
     return m;
   }
   ```

   (C++ review question: how can we get away with allocating “m” as a local variable on the stack?)