CG Programming I – Assignment #1 (Points in the complex plane)

In this lab, you will implement several methods of rotation in the 2D plane. Rotations will be implemented using three different methods, and each method should produce identical results. Each of the three methods will be displayed simultaneously.

In addition, a simple fragment shader will render point-sprites in the shape of ellipses. The `gl_PointCoord` will be used to determine the location within each point-sprite. This coordinate will be evaluated against the equation of an ellipse. Coordinates inside the ellipse will be rendered in some color, and coordinates outside the ellipse will not be rendered.

A fair amount of base code will be provided (please refer to the course website for links). A video of the expected final output will be shown in class. The assignment will be implemented in three parts. Each part will be due in successive weeks.

Part 1: Due on 3-October-2012 by the end of class

- Download the base code and get it to compile.
- Modify `explicit_rotation.vert` to rotate the incoming complex number, stored in the attribute `z`, using a direct application of the rotation formula.

Part 2: Due on 10-October-2012 at the start of class

- Modify `angle_addition.vert` to convert the incoming modulus and angle, stored in the `.x` and `.y` components, respectively, of the attribute `z`, to real and imaginary components. Store these in the `.x` and `.y` components of `gl_Position`.
- Implement rotation by adding the `rotation_angle` to the angle of the complex number (from `z.y`).
- Modify `Redisplay` in `main.cpp` to calculate the rotation matrix corresponding to the rotation angle stored in `angle_offset`. Store this in the variable `m`. This variable is already passed into the various shaders.

At this point all three regions of the screen should display the same rotating configuration of squares.

Part 3: Due on 10-October-2012 at the end of class

The final part will combine the ideas of rotation in the complex plane developed in the first parts of the assignment with coordinate frames and a simple procedural texture generator.

- In the fragment shader, emit the incoming `gl_PointCoord` as the red and green components of the `gl_FragColor`. `gl_PointCoord` takes values on the range $[0, 1] \times [0, 1]$, with $(0, 0)$ at the upper-left corner of the sprite and $(1, 1)$ at the lower-right. This should produce a predictable color pattern on each sprite.
- Since $[0, 1] \times [0, 1]$ is not a useful coordinate space for performing the ellipse calculation, convert this range to traditional Cartesian coordinates $[-1, 1] \times [1, 1]$. In this new coordinate space, $(-1, -1)$ is in the lower-left and $(1, 1)$ is in the upper-right. This should also produce a predictable color pattern on each sprite. Color components less than 0 will be clamped to 0 (i.e., black) in the output.
- Modify the fragment shader to draw an ellipse on the sprite. Apply the Cartesian coordinate calculated in the previous step to the equation of an ellipse (below). Each fragment with a coordinate inside the ellipse should get one color value written to `gl_FragColor` and each fragment outside the ellipse should get a different color.

  In the equation of an ellipse, a and b are the lengths of the X and Y axes of the ellipse.

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

- Notice how parts of sprites outside the ellipse obscure parts of other sprites. Modify the fragment shader to `discard` fragments outside the ellipse instead of giving them a different color.
- Rotate the ellipses so that they maintain a consident orientation with the pattern of moving sprites. It should appear as though the whole image is rotating instead of the individual sprites rotating. The only coordinates available to transform are the coordates of the coordinate frame (basis). Since the basis is being transformed instead of vectors in the basis, the transformation must be implemented in a slightly different manner.

Criteria	Excellent	Good	Satisfactory	Unacceptable
Completion	Program correctly implements all required elements in a manner that is readily apparent when the program is executed. User interface is complete and responsive to input. Program documents user interface functionality.	Program implements all required elements, but some elements may not function correctly. User interface is complete and responsive to input.	Program implements most required elements. Some of the implemented elements may not function correctly. User interface is complete and responsive to input.	Many required elements are missing. User interface is incomplete or is not responsive to input.
Correctness	Program executes without errors. Program handles all special cases. Program contains error checking code.	Program executes without errors. Program handles most special cases.	Program executes without errors. Program handles some special cases.	Program does not execute due to errors. Little or no error checking code included.
Efficiency	Program uses solution that is easy to understand and maintain. Programmer has analysed many alternate solutions and has chosen the most efficient. Programmer has included the reasons for the solution chosen.	Program uses an efficient and easy to follow solution (i.e., no confusing tricks). Programmer has considered alternate solution and has chosen the most efficient.	Program uses a logical solution that is easy to follow, but it is not the most efficient. Programmer has considered alternate solutions.	Program uses a difficult and inefficient solution. Programmer has not considered alternate solutions.
Presentation & Organization	Program code is formatted in a consistent manner. Variables, functions, and data structures are named in a logical, consistent manner. Use of white space improves code readability.	Program code is formatted in mostly consistent with occasional inconsistencies. Variables, functions, and data structures are named in a logical, mostly consistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted with multiple styles. Variables, functions, and data structures are named in a logical but inconsistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted in an inconsistent manner. Variables, functions, and data structures are poorly named. Use of white space hurts code reability.
Documentation	Code clearly and effectively documented including descriptions of all global variables and all non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of most global variables and most non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of the most important global variables and the most important local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted.	No useful documentation exists.

This rubric is based loosely on the "Rubric for the Assessment of Computer Programming" used by Queens University (http://educ.queensu.ca/ compsci/assessment/Bauman.html).