CG Programming I – Assignment #1 (2D ellipses)

In this part of the assignment, you will implement a simple 2D graphics effect using GLSL shaders. Using the fragment shader, draw a rotating grid of rotating ellipses. The ellipses and the rotation will be implemented in a fragment shader. The screen will be drawn using a single draw call (two triangles to cover the entire window). 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. The first, part 0, is due at the end of class today.

Part 0: Due on 5-October-2011 by the end of class

Using the provided base code, implement a fragment shader that will draw a tiled screen of ellipses. Each ellipse "tile" should be 100 pixels by 100 pixels. The radius of the major axis of each ellipse should be 50 pixels, and the radius of the minor axis of each ellipse should be 25 pixels. Use either the X-axis or the Y-axis as the ellipse's major axis (and the other for the minor axis).

Implement this part in several steps:

- Download the base code and get it to compile.
- Implement a fragment shader that will output a constant color for all pixels. Put this shader in a text file named `ellipse.frag`. This is the file name that base code expects to load from disk.
- Modify the fragment shader to draw a single ellipse. The only input available to your fragment shader is `gl_FragCoord`. This represents the location of the current fragment in the window. The lower-left corner of the window is $(0, 0)$. The fragment shader should use this coordinate and the equation of an ellipse to determine whether each fragment is inside the ellipse or outside the ellipse. Output a different color for each case.

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

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

  Since performing a division for every fragment is expensive, think about ways to implement this equation efficiently.
- Modify the fragment shader to subdivide the window into 100x100 tiles.

Extra credit: The algorithm described above only generates two possible color values. Each pixel is either inside the ellipse or outside the ellipse. The result is an image with unpleasing jaggy edges. This is called *aliasing* and will be discussed at length later in the term. For pixels near the edge of the ellipse, the true edge will actually pass through the pixel.

Each of these edge pixels is partially, or fractionally, covered by the ellipse. Knowledge of this fractional coverage can be used to anti-alias the drawing by blending the inside and outside colors. For example, a pixel that is 10% covered, should have 10% inside color and 90% outside color.

For extra credit, calculate the coverage factor and use the `mix` function to generate the weighted pixel color. Calculating the coverage factor for ellipses is actually quite tricky. Be sure to include a clear description of the calculation you use, and defend the reason you think it works. Incomplete solutions *may* earn some extra credit.

Turn in the extra credit portion with part 1.

Part 1: Due on 12-October-2011 at the start of class

Timed animation is a critical component of any real-time rendering application. Animations should play at a consistent rate regardless of the rendered framerate. It would look awfully silly if a character's walk cycle played faster on a high-end computer than on a low-end computer.

In this part of the assignment you will modify the radii of the major and minor axis of the ellipses based on the amount of elapsed time.

- Add a uniform to the fragment shader that represents the radii of the major and minor axis of the ellipses. Initialize the value of the uniform in the shader with values to generate the same ellipses as before. Note: You must declare `#version 120` at the top of your shader.
- Add code to the base C++ code to get the "location" of the new uniform variable (using `glGetUniformLocation`).
- In the `Redisplay` function of the base code, set the uniform (likely using `glUniform2f`) to the same values set in the initializer in the shader, and remove the initializer. The program should produce the same output.
- In the `Idle` function in the base code, calculate new values for the radii based on the amount of elapsed time. The radii should oscillate between 5 and 50. Use these values in `Redisplay` to set the uniform. The sine and cosine functions can be used to produce a visually pleasing oscillation.

Part 2: Due on 19-October-2011

The final part is to make each ellipse rotate around the center of its grid cell and make the grid of cells rotate around the center of the screen.

- Add a `mat2` uniform to the fragment shader and initialize it to the identity matrix. Use this matrix to rotate each ellipse around the center of its cell. Since the matrix only contains the identity, the output should be the same.
- In the `Idle` function, generate a rotation angle that increases with a fixed rate based on elapsed time.
- In the `Redisplay` function, generate a 2×2 rotation matrix. Pass this matrix to the rotation matrix uniform added in a previous step. Note: Be sure to set the `transpose` parameter to `glUniformMatrix2fv` correctly depending on whether your matrix is stored in column-major (`transpose` is false) or row-major (`transpose` is true) order.
- Repeat the previous steps (with different uniform names and matrix values), to make the cells rotate around the center of the screen. Some changes will be necessary. Recall that the center of each cell is $(0, 0)$. This makes rotating around the center of the cell trivial. However, the center of the screen is *not* $(0, 0)$. On the screen, $(0, 0)$ is at the lower left corner.

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).