CPE 471 - Program 2A
Practice with OpenGL, GLSL data, implicit equations, vectors and data

*This program is due **Sunday 1/29/17** at 11:55pm – 7% of your final grade*

The goal of this program is to practice using OpenGL and GLSL in order to simulate random points in space converging on a circle, along with a rotating line animation. You will need to manage the data and computations on both the CPU and GPU to create the specific animation and rendering. You will need to apply your knowledge about the representation of points in space (relative to a circle and a line – aka implciti circles and implicit lines) and vectors used to travel in a given direction. In general, the vertex shader will be used to reposition the points and the fragment shader will be used to control render attributes of your program.

Your **task** is to make a simple simulation of random points that are attracted to the boundary of an implicit circle. The implicit equation for a circle is:

$$f(x,y) = (x - x_c)^2 + (y - y_c)^2 - r^2$$

where {xc, yc} are the center of the circle and r is the radius of the circle.

You also need the background to be colored based on a rotating implicit line (see second half of write up).

To start with work on the points:

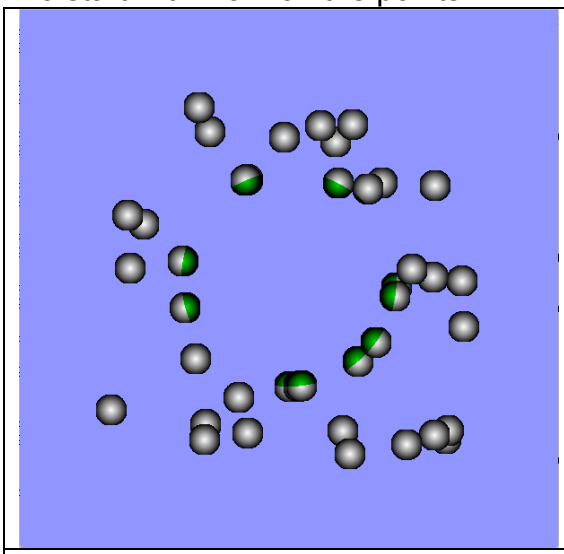


Figure 1: At the program start all the points will be randomly distributed

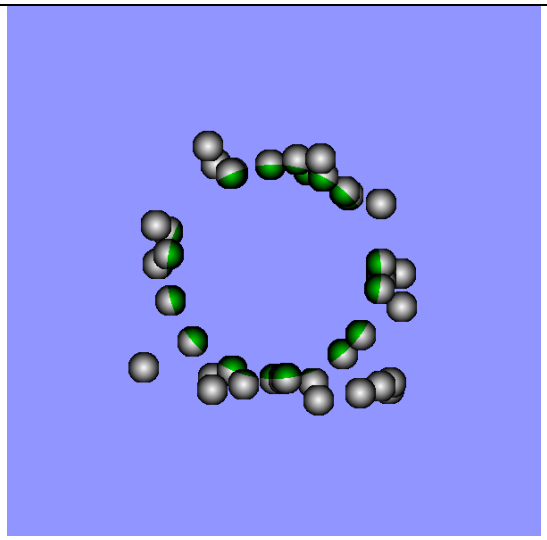


Figure 2: As the program runs, the dots will move toward the center but then stick to the outside of a circle (yellow)

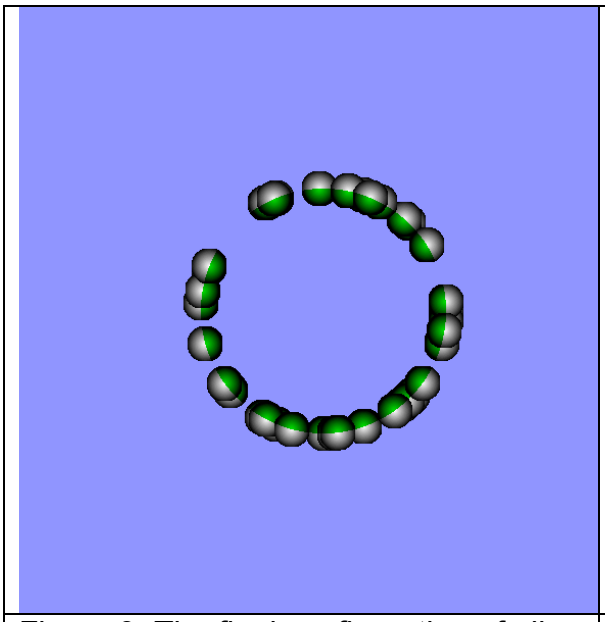


Figure 3: The final configuration of all the points

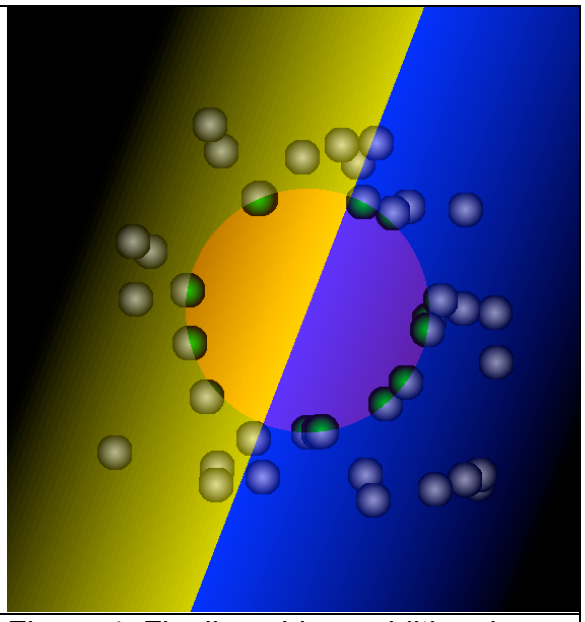
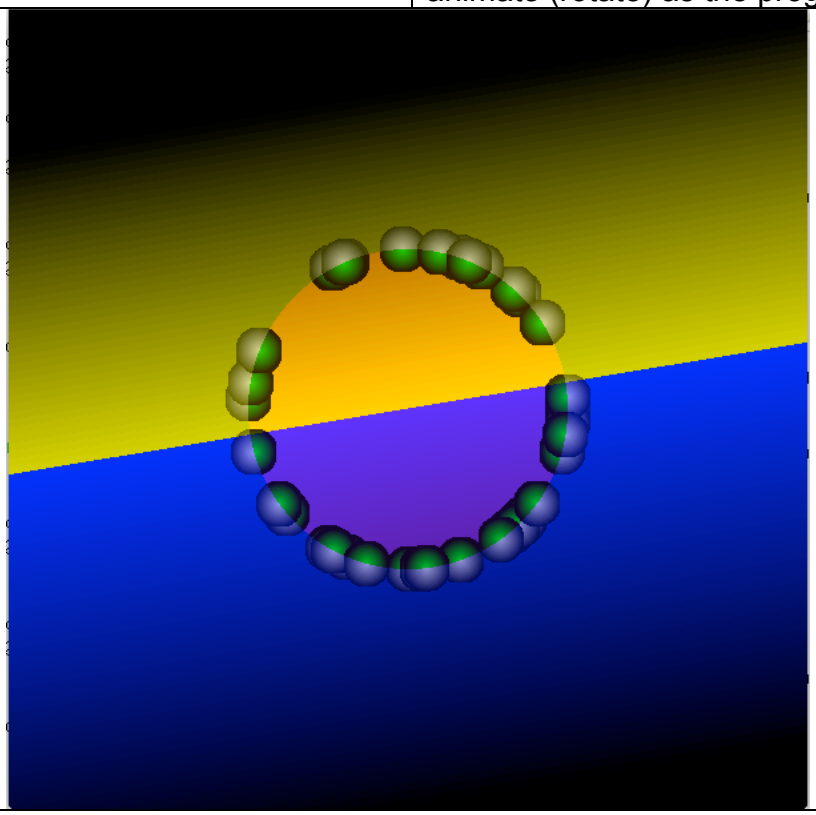


Figure 4: Finally, add an additional shader that shades every fragment in the background based on its distance to an implicit line. The implicit line will animate (rotate) as the program runs.



Fragments in the background are colored based on the line and circle

To complete this task:

- Make an array of point locations that are randomly distributed in the 2D world currently in view (ie in the range {-1, -1} to {1, 1}). Make 40 points total. See Figure 1 for examples of an initial configurations of the points. (If you are feeling adventurous you can support more points).

- First get your code to work with normal "point" drawing, ie use "`glDrawArrays(GL_POINTS, 0, g_numP);`" assuming you've created a buffer object that holds the points that you have bound) (You will need to set glPointSize to something large – I used 34 – your points will not look beautiful but will look like big squares).. Once you get the points behaving as you'd like you need to, have each point colored using a gradient (color ramp) using the distance of a given fragment to the center of the point – (you can use gl_PointCoord to turn your squares into circles).

- In the vertex shader write code to update the point's positions based on moving toward the center of the circle:
    - You will need to think about this carefully and add enough data and computation to the vertex shader to update where the points draw. *Note that the data in the buffer will not actually change*, you will just modify the position sent to gl_Position using a vector to represent the direction of travel and a coefficient to represent how far to travel along that vector in order to reach the circle for each frame rendered.

    - When the points reach the circle, they should stop moving.

Finally modify the points fragment shader in order to make the rendering of your animation more interesting (read about gl_FragCoord or gl_PointCoord to control coloring within each point). You must do several things:

- Each point should be colored with a gradient as stated above. "have each point colored using a gradient (color ramp) using the distance of a given fragment to the center of the point. "
- Color any fragments that are a part of each point that are on the inside of the circle one color, while fragments for a point that are outside the circle are another color.

Next create another set of shaders in order to color the background:

- Create a large "background" polygon that when rasterized has each of its fragments colored based on where that fragment is relative to the circle (see Figure 4) and an animating implicit line!!. i.e. a gradient based on distance to the animating implicit line and just add a bit more of another color for those fragments on the interior of the circle
- The implicit line equation for a line defined by two points (x0, y0) and (x1, y1) is: f(x, y) = (y0 – y1)*x + (x1 – x0)*y + x0*y1 +x1*y0
- Animate the implicit line by updating the values of (x0, y0) and (x1, y1) on the CPU (consider using the parametric equation to pass in two different

points on different sides of the circle) – send the updated endpoints for each frame.
- Play with transparency to have the colors of the background and foreground blend together in a pleasing way

Some general comments:

- Consider choosing colors that look good together. Use Abode Kuler to choose colors: https://color.adobe.com/create/color-wheel/
- You will likely need to use a uniform variable to represent time, which changes for every frame.
- You will need to set up two different shaders (each with a pair, vertex and fragment shader) – one for the points and one for the large quad in the background – this is good practice for later on when you will want different shaders for different elements in your scene.
- You will need to figure out how to complete this assignment using the math we learn. Enjoy the puzzle.
- For those with retina displays – you will need to make sure your program works on the lab machines. This means you will likely also need to specify uniforms for width and height. As gl_FragCoord will be relative to your retina display (and thus too large/out of bounds on the lab machines).

  Percentage point break down:
  - 10% points as circular gradients
  - 30% working simulation of points that move towards the boundary of an implicit circle and stop on the boundary
  - 35% rendering effects for background, including animating implicit line coloring and shift based on circle boundary
  - 10% rendering effects of points on the boundary (inside colored one color, outside colored another)
  - 15% general sanity of your code and program execution