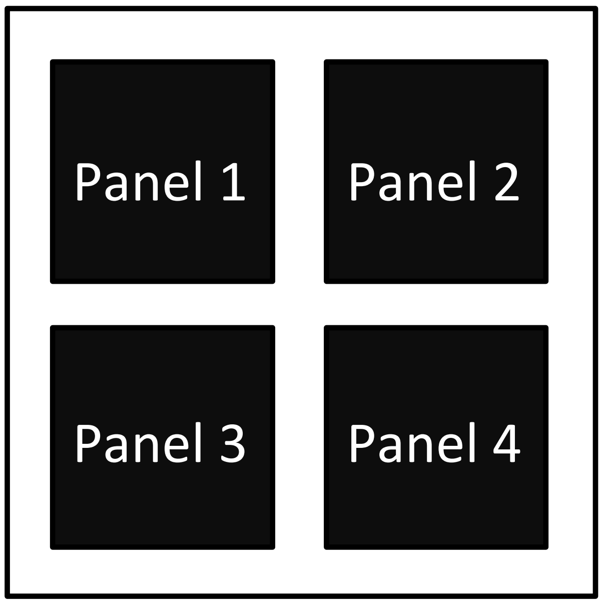
SFU CMPT 361
Computer Graphics
Spring 2017
Assignment 1

Assignment due Monday, January 30, by 11:59 pm.

For this assignment, you are to implement three line drawers and a polygon drawer.

These rendering components should be shown in a program that displays five (or six) pages. Each of the five required pages consists of four panels, which I will call panel 1, panel 2, etc., and are as shown in the following diagram:



Each panel should be 300 x 300 pixels, the the white margins and gutters between panels are all 50 pixels wide or tall. Thus, the total display area is 750 x 750 pixels. Use white and black for the backgrounds, as shown above.

The first three pages will show the output of your line drawers. On each of these pages, one test is performed using three different renderers (the DDA, Bresenham's, and an antialiased line drawer) as follows:

Panel 1: DDA
Panel 2: Bresenham's
Panel 3: A drawing that alternates between lines with the DDA and Bresenham's. The first line drawn is with the DDA, the second with Bresenham's, the third with the DDA, etc.
Panel 4: An antialiased line renderer

The antialiased line drawer can use any algorithm you choose. The better the antialiasing, the better your mark. (In other words, a very poor job or a very good job of antialiasing will get noticed and rewarded appropriately.) The antialiasing that I discussed in class, with the correct area computation (see below), is sufficient for getting full marks.

You may code up all eight octants of your line drawers, or you may code up one and transform the coordinates of the endpoints for the other seven quadrants. If you do this, then you must apply the opposite transform to the pixel coordinates that the line drawer produces.

Treat the repetition in the following tests as a design problem (i.e. eliminate duplication by good design) and not a copy-and-paste problem! Copy-and-paste results in brittle, buggy code.

Page 1. The test to display on the first page is the "starburst" test. Let *c* be one of the four center points of the panel (be consistent across panels). Draw 90 lines from *c*, each of length approximately 125 pixels, equally spaced in angle around *c*. This means one line at 0 degrees, one line at 4 degrees, one at 8 degrees, etc. Parameterize the subroutine you use with *c*, the length of the lines, and the number of them to draw. (And any other parameters you need.) Do not hardwire constants into your code…use const (C++) or final (java) quantities declared at class level.

Page 2. The test to display on the second page is the "parallelogram" test. With x coordinates from left to right in a panel, starting at 0, and y coordinates from **top to bottom** in the panel, starting at 0, draw the lines:

(20, 80+p) to (150, 150+p)

For p = 0 to 50.

Also draw the lines:

(160+p, 270) to (240+p, 40)

Again, for p = 0 to 50.

Page 3. The test to display on the third page is the "random" test. Draw 30 random lines (generate the two endpoints uniformly and independently on the interval from 0 to 299 in x and in y) with a *random color* for each line. (Your line renderers must therefore be capable of handling the color of a line: simply multiply the color's R, G, and B by the pixel coverage (which is 1 for DDA and Bresenham's) and add this to (1 – pixel's coverage)*(the color that's already in the pixel). **Use the same random endpoints and colors for all four panels**. I don't care how you randomly choose the colors.

Page 4. On the fourth page, display the following four scenes of **filled** polygons. In each panel, draw polygons each with a randomly-chosen color. Do no anti-aliasing.

- The first panel should be like the starburst test, only you are drawing 90 triangles from the center (each with two sides of length approximately 125 pixels, and subtending 4 degrees).
- The second panel should have 162 (= 2 * 81) triangles. Start by spacing out a 10 by 10 array of points as evenly as possible in the panel (with some margin, please!) Then connect these points in the obvious way to form a regular grid of 9 x 9 = 81 squares. In each square, put in the (min x, min y) to (max x, max y) diagonal to form 162 triangles. Render these triangles.
- The third panel is like the second, except that before you create the triangles, you randomly shift each point in x and in y. Shift each point once; do not shift it to a different place for each triangle that uses it. Choose the shift for each coordinate from the range [-12, 12].
- In the fourth panel, generate 20 random triangles (endpoints chosen randomly and uniformly from [0, 299] in both x and y) and render them. They will overlap; this is okay.

Page 5. The fifth page should show the same four scenes of filled polygons as shown on page four. However, each polygon should have a color of full white (or (1, 1, 1) in 0-1 coordinates), and an opacity of some smallish constant, say .14. To write a pixel *x, y* with color *color* and opacity *opacity*, follow the pseudocode:

```
Color oldColor = getPixel(x, y);
Color newColor = opacity * color + (1-opacity) * oldColor
SetPixel(x, y, newColor);
```

Here a Color is an (R, G, B) vector. Be sure to multiply by opacity once per color channel; do not try to multiply opacity by a color packed into a 32-bit integer (as this can sometimes create "interesting" colored artifacts). Again, **do not** store opacity in the "A" or alpha channel.

Page 6 (optional). Display any number (up to four) of panels that you think highlight the (positive) attributes of your polygon or line rendering. Do **not** exceed six pages.

The program should switch pages at the press of an on-screen GUI button. The skeleton files provided are already set up to do this.

Include a README file with your project that explains what you did, what you didn't do, and what features you are trying to highlight if you include a page 6.

You may not use any graphics software (libraries, etc.) that is not provided in the skeleton. You may not call OpenGL or Qt or javaFX yourself, for instance. You may not obtain code from the web or other outside sources, and you may not share code with any other students.

Note 1: color is represented by red (R), green (G), and blue (B) values. Each of R, G, and B is called a color channel. Abstractly, and oftentimes in the middle of a renderer, the color values are in the range [0, 1] with 0 being "none" and 1 being "the most intense value of this color we can make." Concretely, down in the hardware, most graphics hardware uses 8 bits for each of these three color values. This leads to some programs representing each of the colors with an integer value from 0 to 255. You are expected to be able to quickly switch between thinking of each channel between 0 and 1, and thinking of each channel between 0 and 255. To convert from one to the other is simply to multiply or divide by 255.

The libraries that we are using represent the colors with 8 bits per channel. These three channels are all *packed* into one 32bit integer with the blue value (0-255) in the bottom 8 bits, the green in the next lowest 8 bits, and the red in the next lowest. This leaves the 8 most significant bits unused. Well, rather than leave them unused, those 8 bits are often used for an *alpha* (or opacity) *value*, which represents how much of the pixel in question is covered by the object in that pixel. Since one hexadecimal digit represents 4 bits, we can think of the color being represented (little-endian) as AARRGGBB which means two digits of alpha, two digits of red, two digits of green, and two digits of blue (in that order, from most significant bit to least significant bit). For example, a color ff104077 means that the alpha value is ff_{16} (= 255_{10}), the red value is 10_{16} (= 16_{10}), the green value is 40_{16} (= 64_{10}), and the blue value is 77_{16} = (119_{10}).

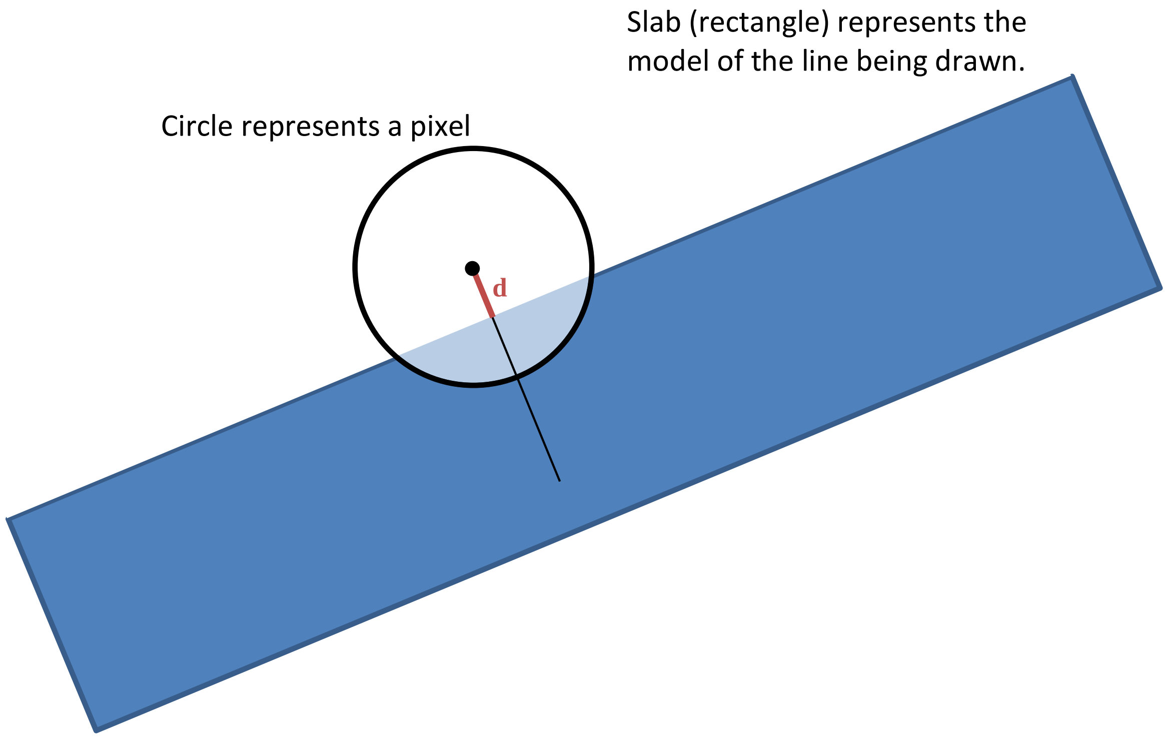
To pack and unpack R, G, and B values from a color represented this way, some bit manipulation is necessary. For instance (here I'm using the prefix 0x for hexadecimal numbers):

```
color = (0xff << 24) + ((r & 0xff) << 16) + ((g & 0xff) << 8) + (b & 0xff);
```

The bitwise-and operations in the above can be omitted if you know **for sure** that r, g, and b are integers between 0 and 255, inclusive. The above calculation puts 255 (i.e., 0xff) into the alpha bits. You should do this for all of your colors. One of the libraries I have provided requires it.

The above statement constructs an ARGB color from r, g, and b. Going the other way also requires bit-shifts and masking, but I'll leave you to determine how to do that.

Note 2: the antialiasing I discussed in class requires that you find the area inside the intersection of a slab with a circle, based on the distance from the edge of the slab to the center of the circle. In other words, if you know *d* in the following diagram, you must find the area of the lightly-shaded part.

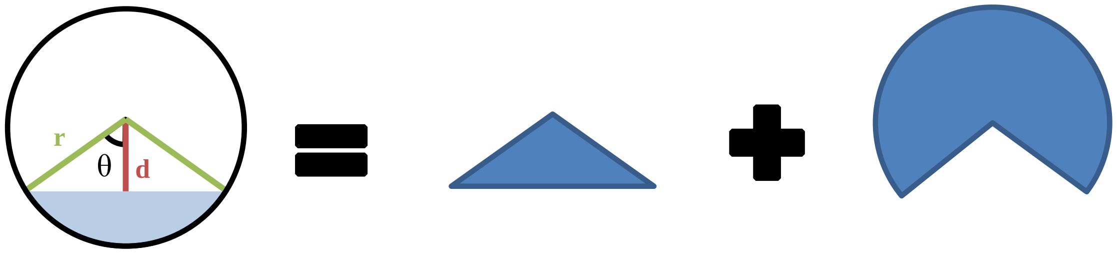

Slab (rectangle) represents the model of the line being drawn.
Circle represents a pixel
d

Popular choices for *r*, the radius of the circle representing the pixel, are 0.5 and sqrt(2). (These give cicles barely contained in, and barely containing, the square for the pixel.)

The area of the full circle is πr^2. Thus the fraction of the pixel that is covered is (the area of the lightly-shaded part above) / πr^2.

The area of the lightly-shaded part above is (πr^2 – (the white area in the circle)), so the fraction of the pixel that is covered is
(πr^2 – (the white area in the circle)) / πr^2,
Or
1 – (the white area in the circle) / πr^2. **(*)**

The white area in the circle can be divided into two parts: a triangle and a "pie wedge". See the diagram below, where everything has been rotated so that the segment of distance d is vertical:



From here, things should be easy for you. The triangle shown has twice the area of the right triangle with hypotenuse r and one side d. The other side of that triangle must be sqrt($r^2 - d^2$), so the right triangle has area (d•sqrt($r^2 - d^2$)) / 2, and the large one has area d•sqrt($r^2 - d^2$).

One can determine θ because cos θ = d/r. The angle of the pie wedge is 2π - 2θ. The area of the pie wedge is thus

$$\frac{2\pi - 2\theta}{2\pi}(area\ of\ entire\ circle) = \frac{2\pi - 2\theta}{2\pi}\pi r^2 = (1 - \frac{\theta}{\pi})\,\pi r^2$$

And the area of the entire white part is thus

$$\left(1 - \frac{\theta}{\pi}\right)\pi r^2 + d\sqrt{r^2 - d^2}$$

We substitute this into the equation **(*)** for the fraction of the pixel that is covered, and we get:

$$1 - \frac{\left(\left(1 - \frac{\theta}{\pi}\right)\pi r^2 + d\sqrt{r^2 - d^2}\right)}{\pi r_2}$$

You should be able to do the algebra from there. Don't forget that cos θ = d/r, or θ = $\cos^{-1}$ d/r.