README.md - Grip

Lab 1: Rasterizer

In this assignment you will implement a simple rasterizer, including features like antialiasing. At the end, you'll have a functional vector graphics renderer that can take in modified SVG (Scalable Vector Graphics) files, which are widely used on the internet.

Logistics

Submission

- The entire lab needs to be submitted on LMS by 03:00 pm.
- A late submission with 20% penalty is allowed till 11:59 pm.

Getting started

You can either download the zipped assignment straight to your computer or clone it from GitHub using the command

```
git clone https://github.com/cs452LumsAssignments/lab1.git
```

Building the assignment

We will be using CMake to build the assignments. If you don't have CMake (version >= 2.8) on your personal computer, you can install it using apt-get / apt on Linux or Macports/Homebrew (replace with brew) on OS X:

```
sudo apt-get install cmake libx11-dev xorg-dev libglu1-mesa-dev freeglut3-dev libglew1.5 libglew1.5-dev libglu1-mesa libglu1-mesa-dev libgl1-
```

To build the code, start in the folder that GitHub made or that was created when you unzipped the download. Run

```
mkdir build
```

to create a build directory, followed by

```
cd build
```

to enter the build directory. Then

```
cmake ..
```

to have CMake generate the appropriate Makefiles for your system, then

```
make
```

to make the executable, which will be deposited in the build directory.

Don't forget to run `make` every time you make changes to your files.

What you will turn in

You will need to *zip* your **src** folder (located in your main directory) and submit it on LMS.

Using the GUI

You can run the executable with the command

```
./draw ../svg/basic/test1.svg
```

After finishing Part 3, you will be able to change the viewpoint by dragging your mouse to pan around or scrolling to zoom in and out. Here are all the keyboard shortcuts available (some depend on you implementing various parts of the assignment):

Key	Action
`' '`	return to original viewpoint
`'-'`	decrease sample rate
`'='`	increase sample rate
`'Z'`	toggle the pixel inspector
`'P'`	switch between texture filtering methods on pixels
`'L'`	toggle scanLine
`'S'`	save a *png* screenshot in the current directory
`'1'-'9'`	switch between svg files in the loaded directory

The argument passed to `draw` can either be a single file or a directory containing multiple *svg* files, as in

```
./draw ../svg/basic/
```

If you load a directory with up to 9 files, you can switch between them using the number keys 1-9 on your keyboard.

Project structure

- Part 1: Rasterizing points
- Part 2: Rasterizing lines
- Part 3: Coloring the bounding boxes
- Part 4: Calculating dot products
- Part 5: Rasterizing single color triangles
- Part 6: Retrieving sub-pixel colors
- Part 7: Filling in sub-pixels

There is a fair amount of code in the CGL library, which we will be using for future assignments. The relevant header files for this assignment are *vector2D.h*, *matrix3x3.h*, *color.h*, and *renderer.h*. Understand that you might have to use some basic *cmath* functions like floor() and abs() so do keep them in mind, make as many local variables as required in your functions and print out anything you might be uncertain about.

Here is a very brief sketch of what happens when you launch `draw`: An `SVGParser` (in *svgparser.**) reads in the input *svg* file(s), launches a OpenGL `Viewer` containing a `DrawRend` renderer, which enters an infinite loop and waits for input from the mouse and keyboard. DrawRend (*drawrend.**) contains various callback functions hooked up to these events, but its main job happens inside the `DrawRend::redraw()` function. The high-level drawing work is done by the various `SVGElement` child classes (*svg.**), which then pass their low-level point, line, and triangle rasterization data back to the three `DrawRend` rasterization functions.

TASK I: Drawing

In this part you will learn how to render points and lines (the basis of all shapes) on a raster display. You may want to take a look at the `sample_buffer` struct in drawrend.h before you begin.

Part 1: Rasterizing points

Go to `DrawRend::rasterize_point` function in *drawrend.cpp*.

Given the exact float coordinates of one point to be rendered from the SVG, use the `fill_pixel()` function to fill the given color into the corresponding SampleBuffer (pixel), in the 2D vector of SampleBuffer's called "samplebuffer". (Make sure you read the drawrend.h file thoroughly! Never forget to look at global variables and how they are being used. How can we get the size of the samplebuffer?)

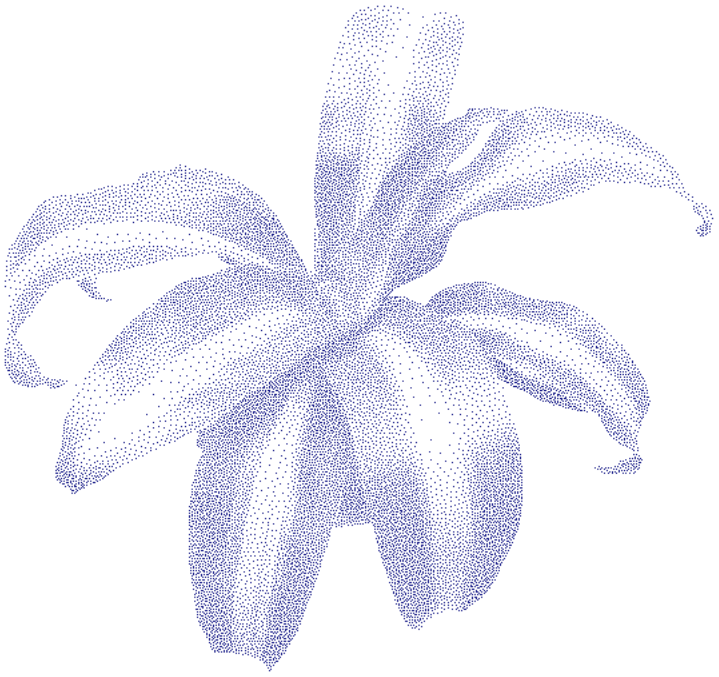
The `fill_pixel()` function call should look something like: `samplebuffer[row][column].fill_pixel(color)`

Do not forget to deal with edge cases (i.e. don't color if the corresponding pixel does not exist in the samplebuffer, stay inside the bounds of its size).

To test you implementation, run the following command:

```
make;./draw ../svg/basic/test1.svg
```

If implemented correctly, the following image should render:



Part 2: Rasterizing lines

Go to `DrawRend::rasterize_line` function in *drawrend.cpp*.

Given the coordinates of the starting (x0, y0) and ending points (x1, y1) of a line to be rendered, use the `rasterize_point()` function (that you implemented in Part1) to rasterize all the points on the line.

For details on how to render a line, you may want to refer back to slides 13 through 23 of lecture 2 for the DDA algorithm.

To test you implementation, run the following commands:

```
make;./draw ../svg/basic/test2.svg
```

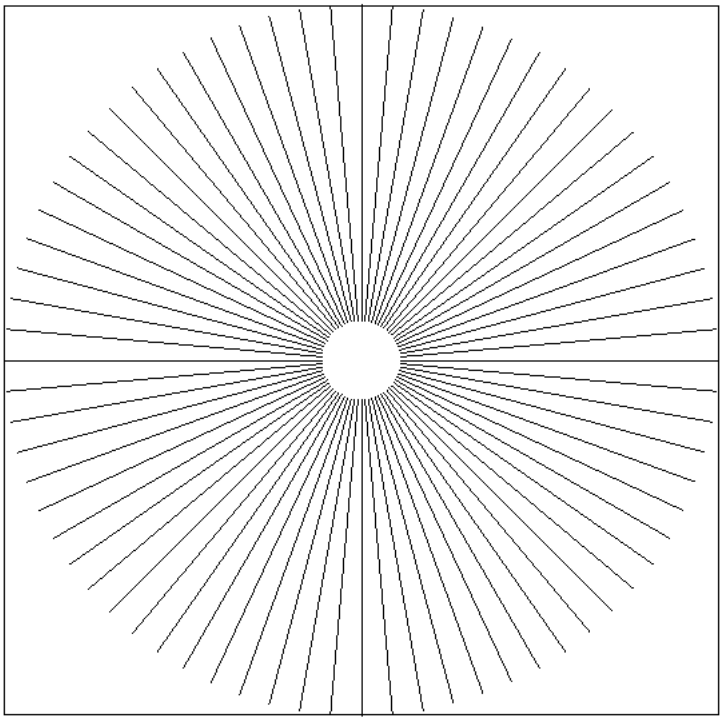
```
make;./draw ../svg/basic/test6.svg
```

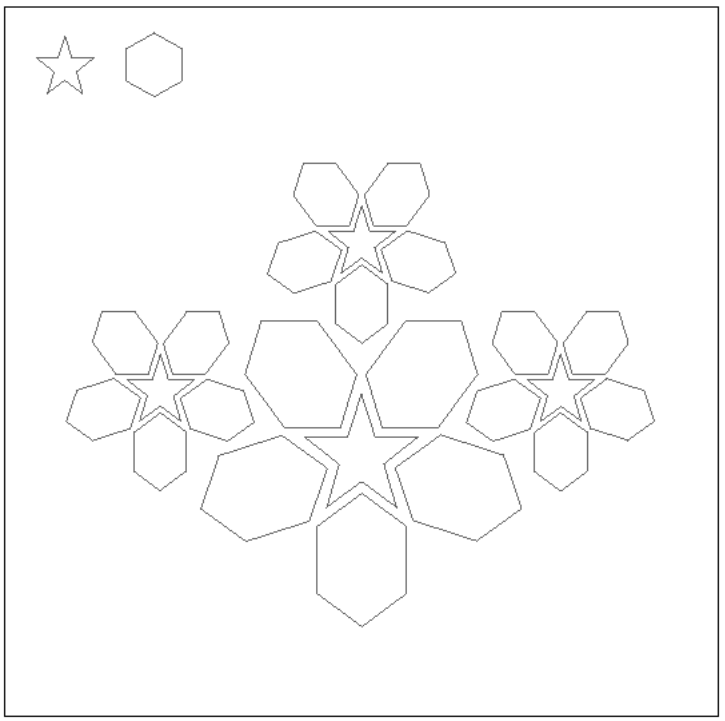
```
make;./draw ../svg/basic/test3.svg
```

Watch out for edge cases again! Think about the 'direction' you are iterating in and the order of the parameters given to the function. They may not be in the ideal positions/order given in the slides, so try fixing it and apply the appropriate constraints on the terminating condition of your main loop.

Also, can you detect and handle lines with slopes that are too steep? (think about the values of your coordinates for a steep slope and how they conditionally affect the 'steps' you have to take)

If implemented correctly, the following images should render:





If you're done early, try testing your implementation for the following files:

```
make;./draw ../svg/illustration/
```

TASK II: Coloring

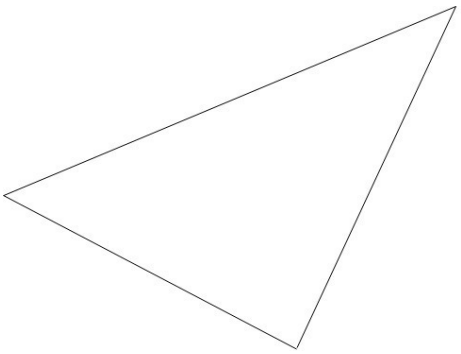
Triangle rasterization is a core function in the graphics pipeline to convert input triangles into framebuffer pixel values. In this task, you will implement triangle rasterization using the methods discussed in lecture 3 to fill in the `DrawRend::rasterize_triangle(...)` function in *drawrend.cpp*.

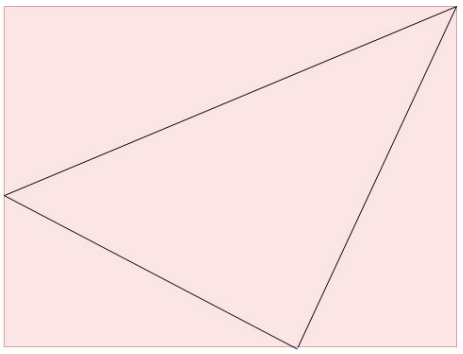
Part 3: Coloring the bounding boxes

Go to `DrawRend::rasterize_triangle` function in *drawrend.cpp*.

Given the coordinates of the 3 vertices of a triangle, find and loop over the 2D minimum bounding box around this triangle. Use the `fill_pixel()` function to color all the pixels within this bounding box.

The following image illustrates a triangle and its bounding box.

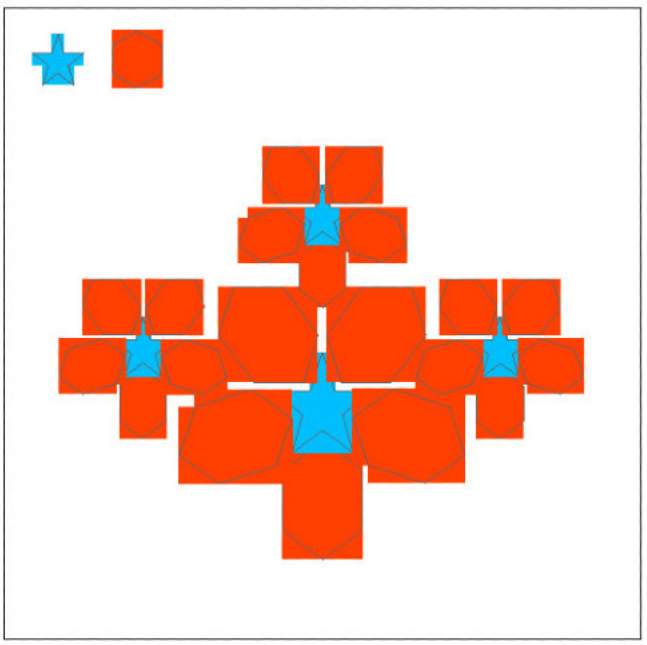


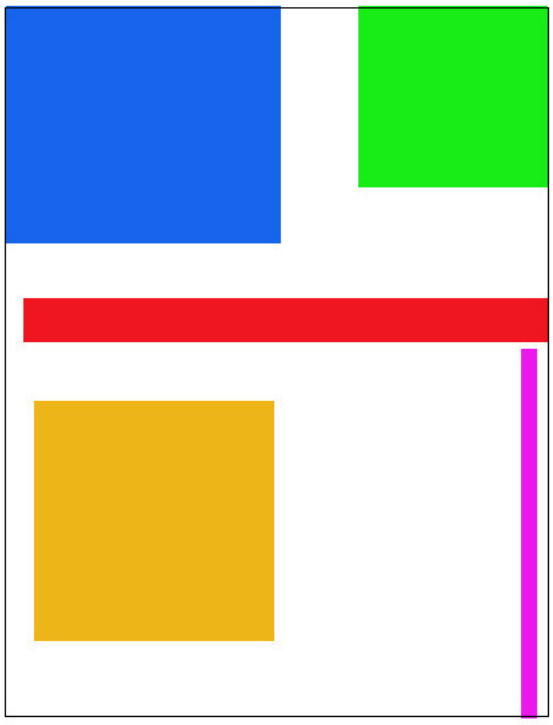


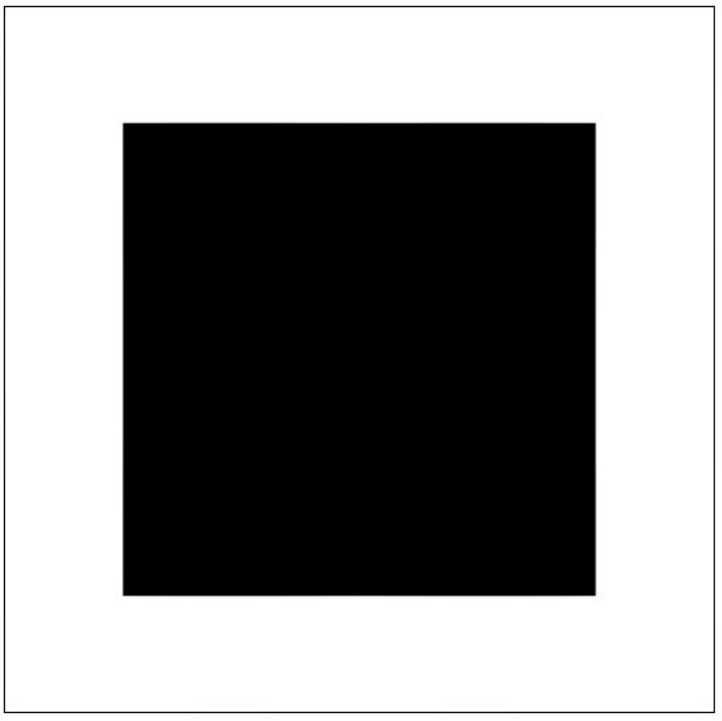
To test you implementation, run the following command:

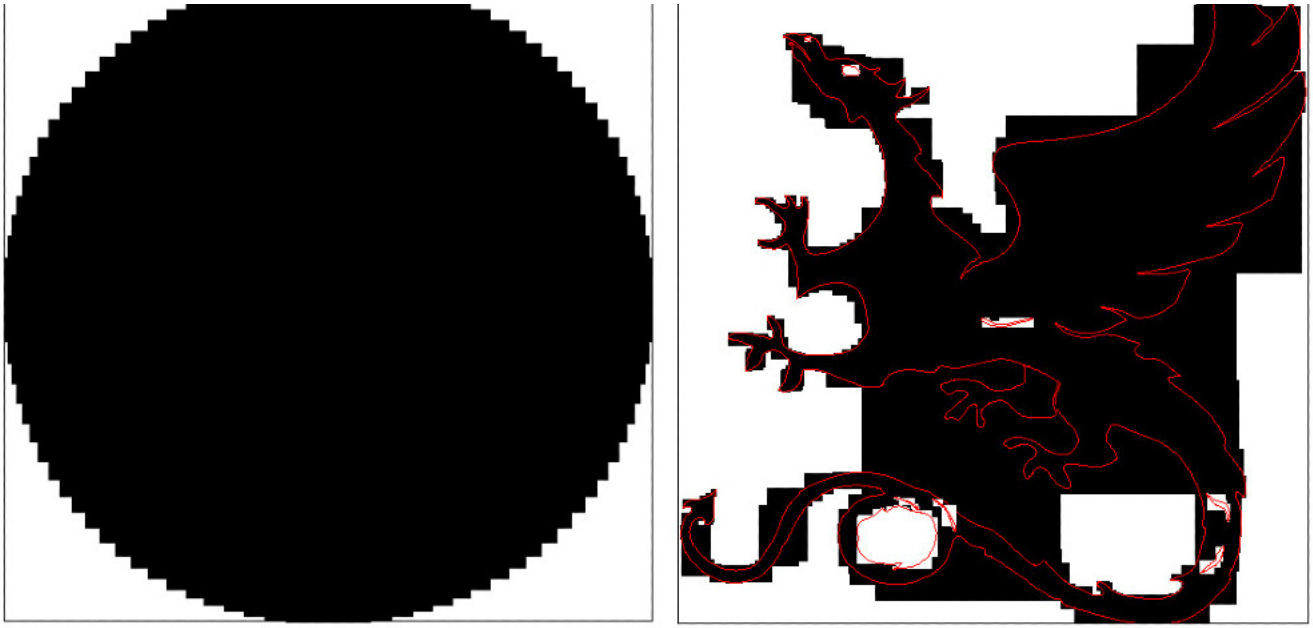
```
make;./draw ../svg/basic/
```

If implemented correctly, the following images should render upon cycling from 1 to 8.









Part 4: Calculating dot products

Go to the `dot_product()` function in *drawrend.cpp*

Given the coordinates of the end points of an egde, calculate the dot product between the **norm** of this edge and the given **point**.

You may want to refer to slides 14-18 from lecture 3 for details on dot product.

Part 5: Rasterizing single color triangles

Go to the `rasterize_triangle()` function in *drawrend.cpp*

Modify your loop from Part3 and use the `dot_product()` function from Part2 to perform **point-in-triangle test** on each pixel in the bounding box. Use the `fill_pixel()` function to color the points that lie inside the triangles.

You may want to refer to slides 19-26 from lecture 3 for details on point-in-triangle test.

Remember to do point-in-triangle tests with the point exactly at the center of the pixel, not the corner. Your coordinates should be equal to some integer point plus (.5,.5). Make sure the performance of your algorithm is no worse than one that checks each sample once within the bounding box of the triangle.

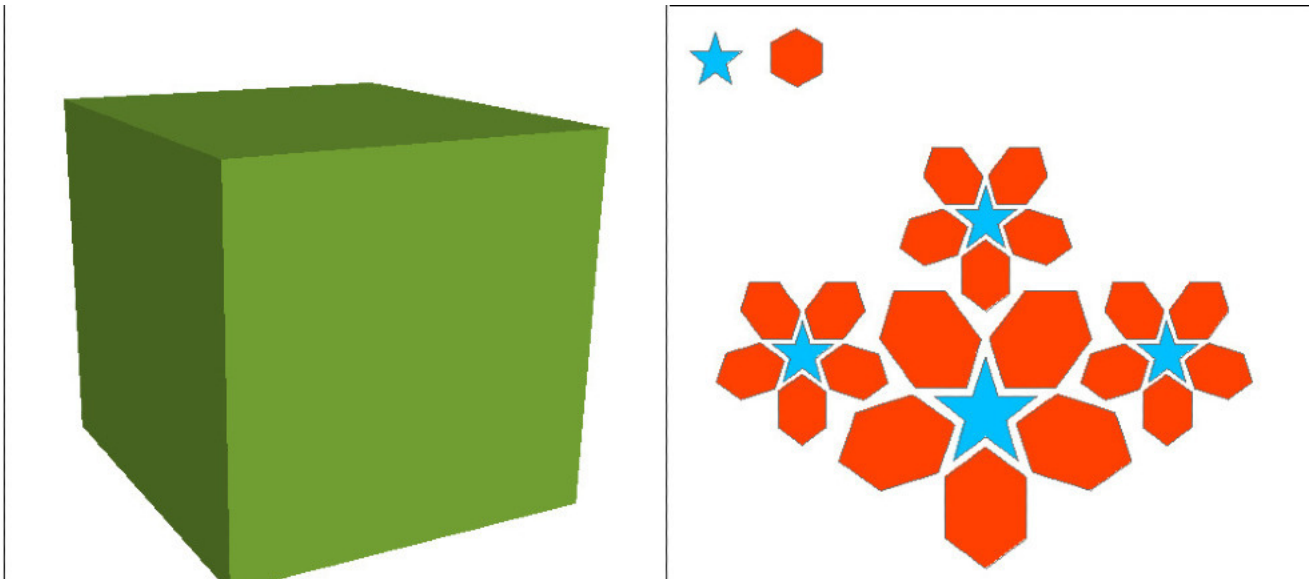
You are encouraged but not required to implement the edge rules for samples lying exactly on an edge.

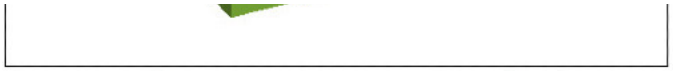
To test you implementation, run the following command:

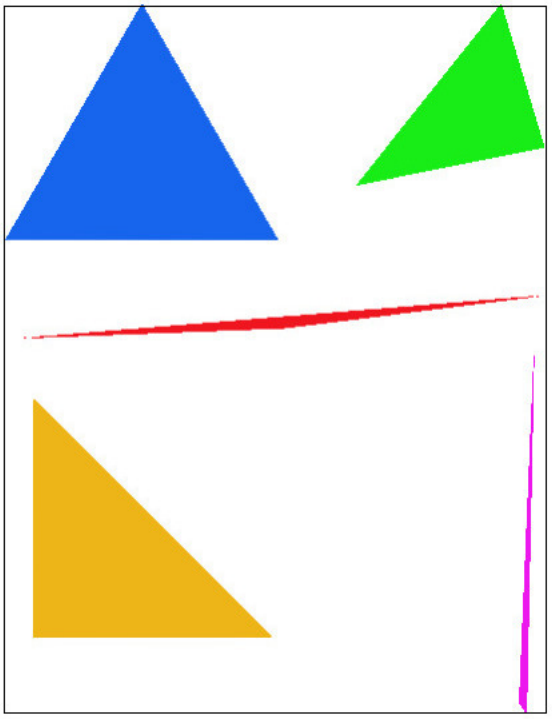
```
make;./draw ../svg/basic/
```

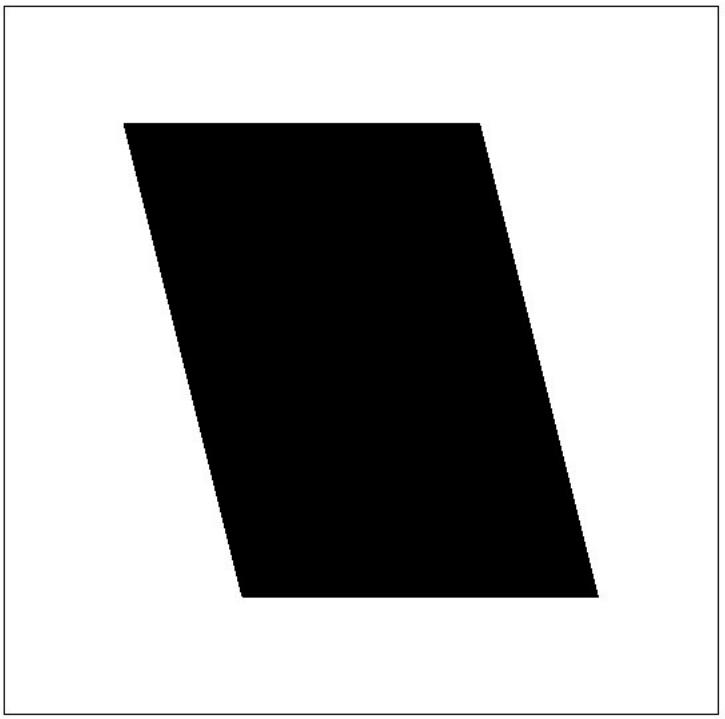
Do not be afraid to make any functions of your own as required, just ensure they do not contradict predefined ones in the codebase - change the name if errors occur.

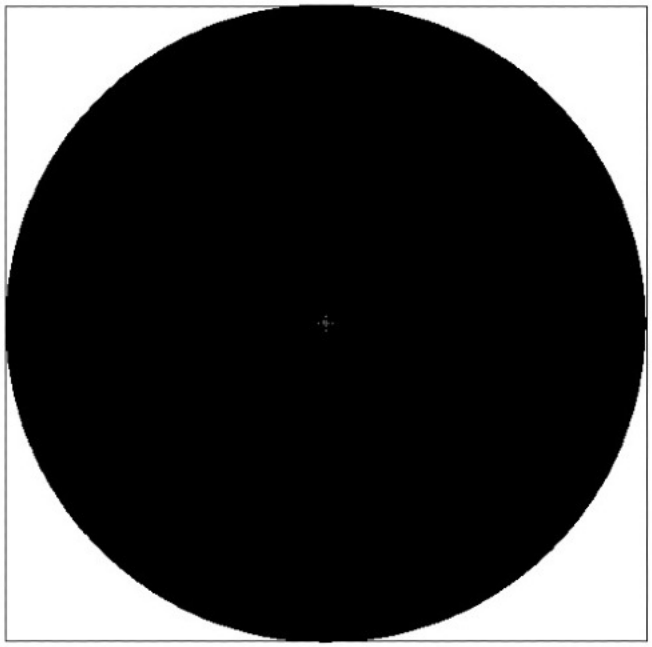
If implemented correctly, the following images should render upon cycling from 1 to 8.











If you encounter some partially filled stars or a disconnected circle, then there was something wrong with your implementation, consider fixing it before submission.

If you're done early, try testing your implementation for the following files:

```
make;./draw ../svg/illustration/
```

This assignment is adapted from UC Berkeley CS184 (https://cs184.eecs.berkeley.edu/article/3).