CISC 120: Lab 3

1 Introduction

The format of this lab will be a bit different from what we've done up to now. Rather than examining a specific programming problem, we will do a series of small ones, designed to introduce you to the core ideas of using the `stddraw` module. We will begin with very simple drawings, and then progress to slightly more complex ones, culminating in a simple animation.

2 Creating Drawings

The `stddraw` module provides a very simple interface for creating programs that have a graphical output. It isn't anything you'll want to use to create graphical interfaces, but it will allow us to draw simple objects, and later, work with pictures. This is sufficient to get a basic idea for how computer graphics work.

To begin with, we need to understand one key concept. All of our drawings will involve specifying coordinates (draw a line between point x and point y, etc.), which will look and feel a lot like Cartesian coordinates. By default, in the standard drawing module, the origin point of the system is located in the lower left-hand corner, and the upper-limits for the two coordinates is 1. As we will see in a bit, we can re-adjust this default behavior to better suit the problem that we are working in, but we'll start with the defaults.

The second core concept we need to grasp is that the standard drawing module makes use of a background canvas, upon which all drawing operations are applied. This background canvas is an abstraction, and is not actually visible. You can apply your drawing calls to this canvas at will. Then, when you are ready to show the user your drawing, you must call the `stddraw.show()` function. This will reveal the background canvas.

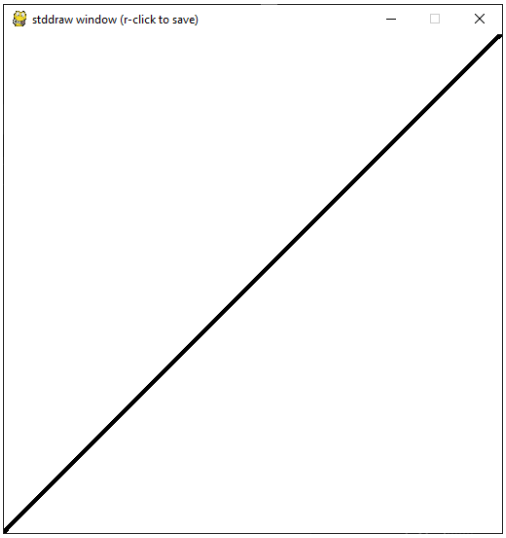
We'll begin with Sedgewick's first program, the "hello, world" of graphics. This is a triangle with a dot in the center of it. For this program, you will need two drawing functions: `stddraw.line` and `stddraw.point`. The behavior of these two functions should be pretty obvious–the former will draw a line on the background canvas, and the latter will draw a point.

More specifically, `stddraw.line` will accept four floating point numbers as input. These four floats represent the x and y coordinates of two points, between which a line will be drawn. For example, if we wanted to display a line cutting across the center of our default canvas, we would need the lower-left point, $(0, 0)$ and the upper right, $(1, 1)$. The program to draw this line would be,

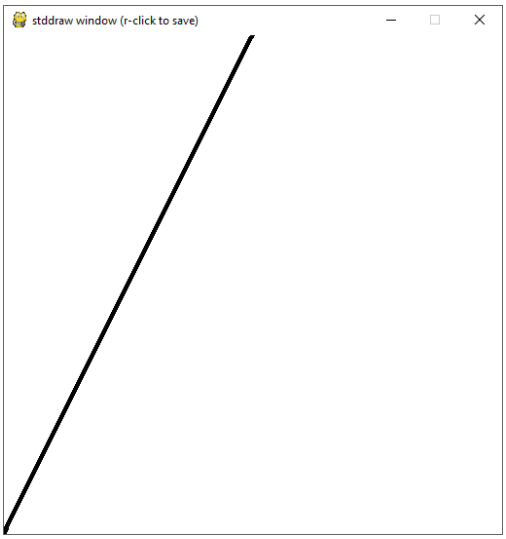
```
import stddraw

stddraw.line(0, 0, 1, 1)
stddraw.show()
```

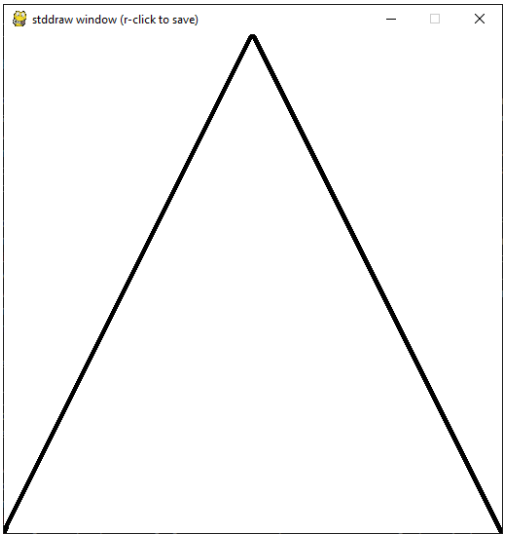
and the output would be,



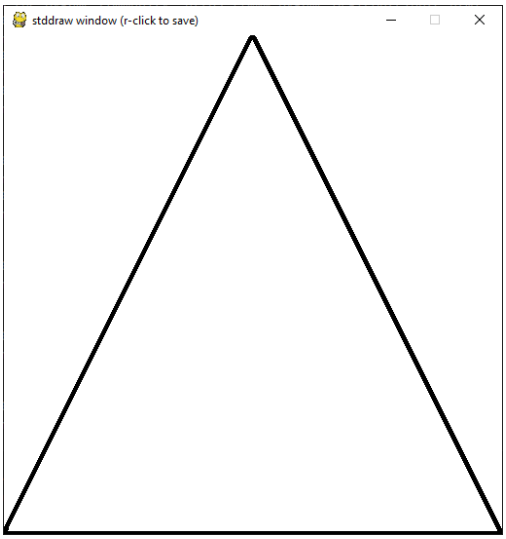
To draw a triangle, we will need to make three calls to this function, to give us our three sides. If we wanted to draw a triangle that takes up the entire drawing canvas, for example, we would need to draw a line from $(0, 0)$ to $(.5, 1)$, to get us a line from the bottom left corner to the top center of the window, like so,



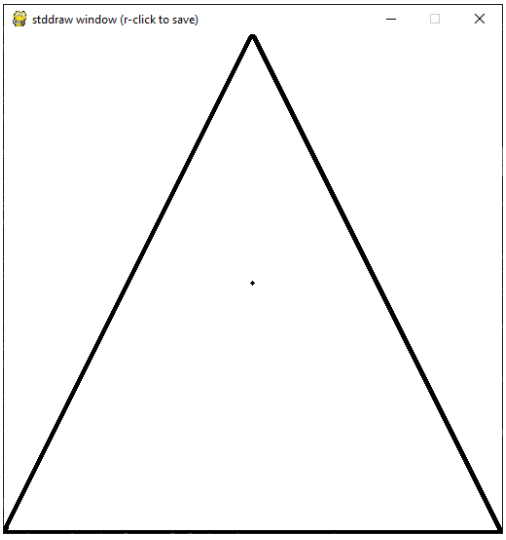
and then from $(.5, 1)$ down to $(1, 0)$ to get the next side, running from top-center to bottom-right,



and then finally from $(1, 0)$ to $(0, 0)$ to get the base of the triangle,



and, finally, we need to draw the point in the center. This can be accomplished using the `stddraw.point` function, specifying the coordinates of the center of the canvas, $(.5, .5)$.



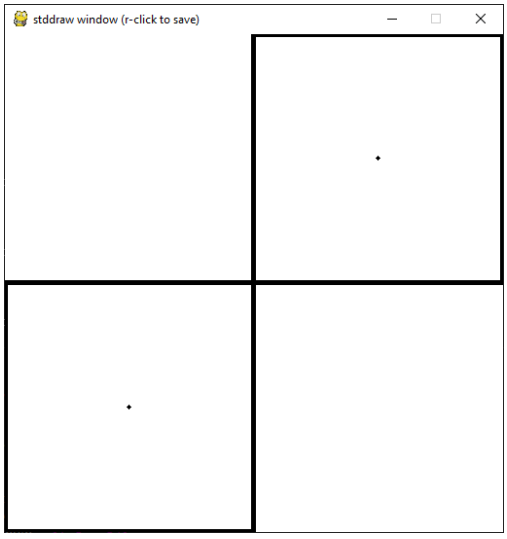
Taken together, the code to do this is,

```
import stddraw


stddraw.line(0, 0, .5, 1)
stddraw.line(.5, 1, 1, 0)
stddraw.line(1, 0, 0, 0)
stddraw.point(0.5, 0.5)
stddraw.show()
```

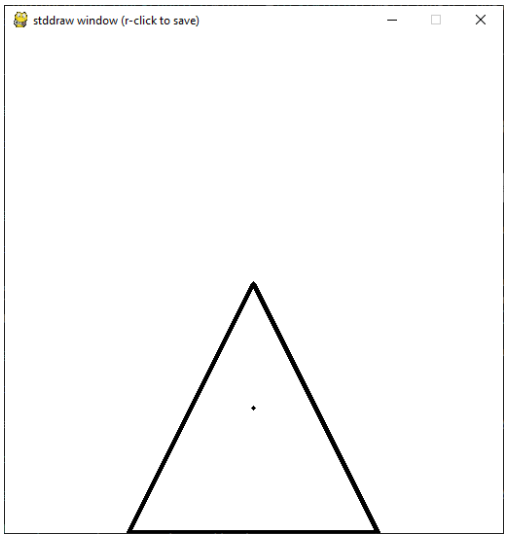
Double Squares

Now, it is your turn. Write a program that will draw two squares, one in the upper-right hand corner and one in the lower-left hand corner, each with a dot in the center. Your final output should look like this, **10pts.**



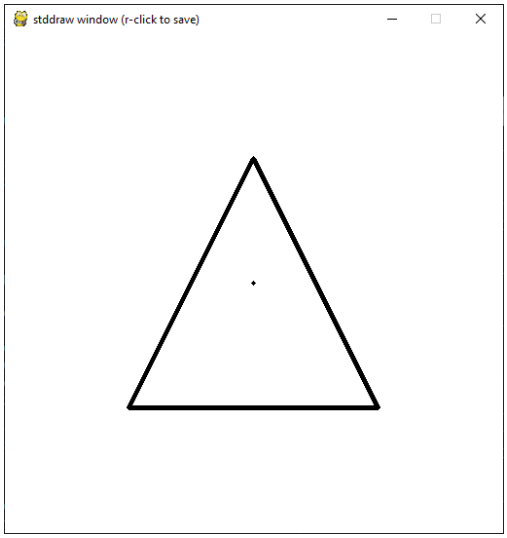
Smaller Triangle

Next, return to the triangle code above. Modify this program so that the triangle displayed is half the size of the current one. Leave the bottom of the triangle at the bottom edge of the canvas, but center it horizontally. And make sure to adjust the point in the center so that it still remains centered in the triangle. Your output should look like this, **20pts.**



Vertically Centered Triangle

And now, modify the above program to center the triangle vertically as well as horizontally, updating the dot location too. Your output should like as follows, **10pts.**



3 Simple Animations

This same module can be used to create simple animations as well. Fundamentally, all that a video/animation is is a sequence of static images displayed in rapid succession.

Thus, we can set up an image, display it for a fixed amount of time, and then set up and display a new one. If we do this quickly enough, we will have an animation.

In order to do this, we must first be able to display multiple things automatically. As it stands, calls to `stddraw.show` will effectively lock the program up indefinitely. Luckily, we can get around this by passing an argument to the function call. If we give it an integer as input, then `stddraw.show` will display the content of the background canvas for that many milliseconds, before releasing the program to keep running.

This means that, if we place our call within a loop, we can display a sequence of images in rapid succession–resulting in an animation.

For example, copy and execute the following program. This should show an animation of a dot that bounces back and forth along the center of the canvas.

WARNING 1 *Note that this program contains a `while True:` loop, and will thus run endlessly. When you tire of watching the bouncing dot, either click on the terminal window in which the program is running and use the keyboard interrupt (ctrl-c) to kill the program, or close out of the animation canvas.*

```
import stddraw

x_coord = 0
y_coord = .5
velocity = .01

while True:
    # Clear the canvas (so that only one dot appears)
    stddraw.clear()

    # Draw the point
    stddraw.point(x_coord, y_coord)

    # Display the background canvas
    stddraw.show(10)

    # Update the coordinates of the point
    # in the next frame.

    # If we have hit either boundary, reverse the direction
    # of motion (to keep the dot on the canvas).
    if (x_coord + velocity) >= 1 or (x_coord + velocity) <= 0:
        velocity = -velocity

    # Get new coordinate
    x_coord = x_coord + velocity
```

Experimentation

Before writing your own animations, it pays to play around with this code and see what happens if you tweak some of the dials and nobs. Using this code as your basis, make the following changes, and describe what happens when you make the change. Each change should be made in isolation. This means that once you make and document the first change, return the program to its original state before making the next one.

(a) **Comment out/Remove the `stddraw.clear` function call.** **5pts.**

If one will comment out or remove the stddraw.clear() call, the background canvas will not be cleared in each iteration of the loop. Consequently, instead of seeing a single dot moving across the screen, one will see a continuous line drawn by the moving dot because all previous positions of the dot remain on the screen.

(b) **Modify the velocity. Try at least 3 different values, both increasing and decreasing.** **5pts.**

Changing the velocity value will change the speed at which the dot moves across the screen. If you increase the velocity, the dot will move faster, making the animation quicker. If you decrease the velocity, the dot will move slower, and the animation will appear more sluggish. Trying different values will give you a good sense of how the speed of the dot affects the perceived animation.

(c) **Replace the point with a line, which has its first point fixed at $(0, 0)$ and which uses the bouncing point as its end point.** **10pts.**

By replacing the point with a line that starts at (0,0) and ends at the moving dot's coordinates, one will create a visual effect of a line that swings back and forth like a pendulum, anchored at the origin.

Your First Animation

Now that you've had a chance to experiment with animations, its your turn to write one. Create an animation that is based upon your code the question "Vertically Centered Triangle". This animation should feature the same triangle. However, the dot should bounce up and down (so update the y coordinate) inside of the triangle. When it reaches the edges of the triangle, its velocity should be reversed so that it always remains inside of the shape. **20pts.**

Your Second Animation

Finally, we will animate "Double Squares". Create a program which is based on this one, except have the two dots bounce within their respective squares. One dot should bounce horizontally, like the dot in the example animation above. The second dot should bounce along its respective square's *diagonal*. Note that this will require modification of both the x and the y coordinate simultaneously. **20pts.**

4 Submission

To earn full credit for this lab, you must submit five documents by the assigned deadline. First, make a photocopy of **this lab packet**, with your answers to the questions written into it, and submit this as a .pdf file. Then, submit **four .py files**. One for each of the following questions:

- Double Squares
- Smaller Triangle
- Vertically Centered Triangle
- Your First Animation
- Your Second Animation

If you fail to turn in one of these files, you will forfeit any points associated with that file.

Both of these Python files should run. If you call the first of these programs, `squares.py`, for example, then I will attempt to execute the following on command prompt, with my working directory set to the folder containing all of your submitted documents,

```
% python squares.py
```

If the program throws an error immediately, be it a syntax error, type error, name error, or any other error, then you won't receive any points for that particular program. If you cannot get the program to work, then implement as much of the required functionality as you can without running into any of these errors. This will maximize your opportunity to earn partial credit. It is better to turn in a half-functional program that runs, than an attempt to implement all of the functionality that does not.

5 Rubric

Question	Points	Score
Double Squares	10	
Smaller Triangle	20	
Vertically Centered Triangle	10	
Experimentation	20	
Your First Animation	20	
Your Second Animation	20	
Total:	100	