

APCS :: Graphics Redux Lab 1

Introduction:

This lab is meant to guide you through a few exercises, building on what we've done in class recently.

It DOES NOT hold your hand through everything. You will have to look things up, experiment, and play in order to figure things out.

Some parts have more lengthy explanations in an effort to provide clarity and relate to something you've already done.

Don't be afraid to help each other.

1. Download the starter code, compile it, run it – THEN READ THIS.

I have changed some things in this code from the last time you saw it. Below is an explanation of the changes. You should follow along with the applicable code:

1.1 Rectangle vs. Rectangle2D.Double (see the “Cyclist” class)

Turns out, there is a class Rectangle that extends Rectangle2D. This class will be more convenient to use for the time being because it provides some helpful methods for manipulating rectangles, in particular: translate(x,y) which moves the rectangle by the given amounts in the x- and y-directions.

Also, look at Rectangle's constructor (in the API), it is overloaded several ways: you can construct it with the actual x- and y-coordinates, width and height, or by using the Point and Dimension classes.

Point is just like Location in Grid World.

Dimension is just a pair of variables describing a width and height.

I used Point and Dimension for variety, but they will become important and helpful classes later on.

1.2. Drawing things – a better way (see “MainDisplayPanel” and “Cyclist”)

In class, when the main display JPanel needed to draw the cyclist, it asked the cyclist to give it the shape to draw and then it drew it. A better way is to let the Cyclist class know how to draw itself. I do this by providing a method public void draw(Graphics2D g) in the Cyclist class. When the main display panel needs to draw the cyclist, it just called the cyclist's draw method and passes it the current Graphics context object.

There is a bit of a blurring here between Model and View, but we can separate it later if necessary. Really, the Cyclist class shouldn't know anything about display factors – we should have a separate class called something like CyclistDisplay in charge of that. But CyclistDisplay would probably have a draw method just like the one I've added to the Cyclist – we'll do this later once the cyclist becomes an actual image.

1.3. Nudging and movement (see KeyListener and Cyclist)

The code provided moves the rectangle a little bit differently than we did in class. Notice that the nudgeRight method now accepts an integer – the amount to move right. The nudgeRight method sets a variable, **dx**, to the amount it's supposed to move and that's it. Notice also that a move “right” with a negative value would result in a move to the left. NOTE: dx and dy are terms borrowed from the mathematical realm where dx = “delta x” = “the amount of change in the x-direction”

The move method just moves the rectangle by **dx** in the x-direction and **dy** in the y-direction. It then notifies any observers that it has moved. But what calls the move method? The key listener. Why? Two reasons:

1. Practical: it allows us to have notifyObservers() in just one place within the Cyclist code and thus gives us more explicit control over when and how it's called. This is desirable since notifyObservers is kind of a black hole in terms of computing cost. We don't know how many observers there might be and we don't know how expensive all of their update methods are.
2. Design: By separating the assignment of a value to dx, from actually altering the x,y location of the rectangle, we add flexibility to the design. It allows us to potentially modify dx in multiple ways before applying it to the rectangle.

2. You are now ready to start modifying this code to do more stuff.

Task 1:

As an appetizer, modify the key listener to handle each of the four arrow keys and modify Cyclist so that the arrow keys can move the rectangle up, down, left, and right. For the time being, don't worry about keeping the rectangle on the screen. Let it go off for now.

Task 2 – natural movement:

Uncomment the code in “Step 5” at the end of the MyMainFrame constructor that constructs and starts the AnimThread. Look at the AnimThread code to see what it does. Then run the program and see what happens...

Since the AnimThread is ALWAYS telling the cyclist to move (about 32 times per second), once you set dx, that value will be used for all calls to move. Thus, you have to be careful about how you modify dx, and dy.

Your task to modify the Cyclist code so that the rectangle moves "naturally." There are two ways you could think of as "natural" (we're going to try the 2nd one, but you should know about both):

1. Smooth, EXACT movement.

Say that a `nudgeRight(N)` means you want the rectangle to move exactly N pixels to the right. But you don't want it all to happen in one time frame. You'd like it to move at a constant rate, taking several frames of animation. To do this, you need to decide how fast things can move, and thus your dx will always be fixed. Then, the Cyclist needs to maintain a variable that keeps track of how far it needs to travel in a particular direction. The move method always moves by dx and subtracts from the distance remaining to be traveled. For example if you fix dx at 5, and you want to travel 25 pixels, set a distance variable to 25, and subtract dx from it each time move is called. Once distance is 0, set dx to 0.

2. Non-exact movement with acceleration and deceleration.

With this mindset the rectangle's dx and dy are *never* fixed. Instead of dx being fixed, the amount that dx can change is fixed...

To start with, try this: modify `nudgeRight()` so that it explicitly sets $dx=5$ (ignoring the **amt** passed to the method for now). Then, in the `move()` method, after translating the rectangle, subtract 0.2 from dx if $dx > 0$. Thus, dx decreases slightly with each call to move until dx is 0. See what happens when you nudge right now. Cool, huh? If you did it right, this makes the rectangle drift a little bit and slow to a stop as though there is some friction.

QUESTION: If dx is initially set to 5, how many pixels will the rectangle actually travel, and how many frames of animation will it take, before it comes to rest? Answer: $5 + 4.8 + 4.6 + 4.4 + \dots + 0.4 + 0.2 = 65$ pixels and 25 frames of animation.

Modify the code so that it works for all directions: up, down, left and right. NOTE: you have to be careful about what happens when dx and dy are getting close to 0.

2.1 improving movement.

This is still a *little* unnatural though, because the initial nudge still has the rectangle jump 5 pixels in some direction. (In truthfulness, 5 pixels is small enough that it still seems natural, but try this anyway...) To make it more realistic, the rectangle would actually have to accelerate from a dead stop. That is, it should build up speed as well as slow down.

So what you really want is for a bunch of nudges, fired in rapid succession, to have a cumulative effect. Try this:

When `nudgeRight` is called, rather than setting $dx=5$, add 1.5 to the current value of dx

Now, one nudge will move the rectangle a little bit. But if you rapidly hit the key (or hold it down) it will pick up speed very quickly and accelerate.

So now you should play around with the rates of acceleration and deceleration, how and when they're added and subtracted and try to get a feel for the physics of the system. You'll see how tightly coupled they are. Try to fiddle to make it more and more realistic.

For example, in reality, you probably want to set an upper limit on the absolute value of dx so that things cannot accelerate without bounds. Also, there are laws of diminishing returns on acceleration – certainly for a cyclist. For example, it takes me about 1 second to go from 0 up to 5 mph on my bike; from there, each additional 5 mph I want to go takes about twice as long as it did the previous 5 mph. 0-5mph = 1 second, 5-10mph = 2 more seconds, 10-15mph = 4 seconds, 15-20 = 8 seconds. Thus, the faster you're going the less you can accelerate.

Task 3: Add another observer

This task sounds daunting, but it's really just a bunch of small steps.

We're going to modify the application to display the cyclist's current speed in the label in NORTH region of window. To do this we need to do the following:

1. Provide an accessor method in the Cyclist to return the current dx (or dy, take your pick).
2. (For this step, use MainDisplayPanel as your guide) Make a class that extends JPanel and implements Observer. Call it "SpeedDisplayPanel" or something like that. It should have a private JLabel and its constructor should make a new JLabel and add it to the panel (by simply calling its inherited add method).

Then fulfill the Observer interface by implementing public void update(Observable obs, Object arg){...} (This class will be an Observer of the Cyclist so this method will be called when the cyclist notifies Observers). Since you know the obs variable will be the Cyclist, you can cast it as a Cyclist and get its speed by calling the method you wrote in the previous step.

So, get the speed and change the JLabel to display that speed (by calling JLabel's setText method).

3. Then call repaint() as the last thing in the update method.
4. Compile, run. See if it works. You should see a number that changes constantly as you move the rectangle around.
5. Improve it. The number you're seeing is of course arbitrary. You may want to multiply it by some value and cast it as an integer so it looks prettier. If you're

ambitious you could override your new panel's `paintComponent` method so that it draws a rectangle whose size is related to the speed.