

CS 051G Homework Laboratory # 3

Repulsive Behavior

Objective: To gain experience implementing classes and methods.

Note that you must bring a program design to lab this week!

The Scenario. For this lab, we would like you to write a program that simulates the action of two bar magnets. Each magnet will be represented by a simple rectangle with one end labeled “N” for north and the other labeled “S” for south. Your program should allow the person using it to move either magnet around the screen by dragging it with the mouse. You know that opposite poles attract, while similar poles repel each other. So, if one magnet is dragged to a position where one or both of its poles is close to the similar poles of the other magnet, the other magnet should move away as if repelled by magnetic forces. If, on the other hand, opposite poles come close to one another, the free magnet should move closer and become stuck to the magnet being dragged.

To make things a bit more interesting one should be allowed to flip a magnet from end to end (swapping the poles) by clicking on the magnet without moving the mouse. This will provide a way to separate the two magnets if they get stuck together (since as soon as one of them is reversed it will repel the other).

As usual, copy the folder `Lab3-Magnets` from the folder `cs051G/labs`. Copy this into your `CSC51GWorkspace` folder. This particular program uses two program files: `poleModule.grace` and `magnetgame.grace`. The first one has been written by the instructor. You will NOT need to make any changes to it. You will put all of your code in the second, `magnetgame.grace`.

Open Firefox to <http://www.cs.pomona.edu/~kim/minigrace>. This week you will need to add the two files listed above. Add them in the order listed above and click on “Build” for each of them (in the same order). Bring the file `magnetGame.grace` into the window and click on “Run” after it has been built. It will just pop up a blank window. As usual, you will edit the program in the browser window. Be sure to work in very small pieces so that you test what you have written very often.

Some Video Game Physics If you are worried that you might not remember (or never knew) enough about magnetic fields to do this assignment, relax. First, we will be providing you with most of the code that does the “physics”. Even if you had to write all the code yourself, you still would not need a deep knowledge of magnetism and mechanics. Instead, you could exploit something every special effects expert or video game author must know. Most humans observe the world carelessly enough that even a very coarse approximation of reality will appear “realistic”. Our code takes advantage of this by simplifying the behavior of our magnets. We never compute the force between two magnets, just the distance between them (or, more accurately, their poles). If they get too close together, our code moves them apart or makes them stick together.

Despite the fact that we will provide most of the code to handle this approximate version of physics, there are two aspects of the magnet’s behavior that will impact the code you write. The first is a simplifying assumption. The magnets will not be allowed to rotate. They only slide up, down and across the screen.

More significantly, there is one aspect of the behavior of the real magnets that we must model fairly well. Above, we said that we just compute the distance between two magnets. This would not really be accurate enough, since it is not the distance between the magnets that matters, but the distances between their similar and opposite poles.

Consider the two pairs of magnets shown below:



The magnets shown in the left pair are the same distance apart as the magnets in the right pair. In the pair on the left, however, the opposite poles are close together while the similar poles are relatively far apart. The situation is reversed in the other pair. In one case, one would expect the magnets to attract, in the other to repel. (Remember similar poles repel each other, opposites attract.)

So it is the poles rather than the magnets that really matter when deciding whether something should be attracted or repelled. As a result, instead of just manipulating magnet objects in your program, you will also need objects that explicitly represent poles.

Design of the program. We will help you design this program by identifying the classes and methods needed. In particular, you will need two classes named `magnet`, which generates objects of type `Magnet`, and `pole`, which generates objects of type `Pole`, as well as an object `magnetGame` that inherits `graphicApplication`. We will provide the code for the `pole` class. You will write the other class and object definition in the file `magnetgame.grace`.

The pole class You will be able to use the `pole` class we have defined much like one of the built-in graphics classes in `objectdraw`. In this handout, we explain how to construct a new object of type `Pole` and describe the methods that can be used to manipulate Poles. You can then write code to work with Poles just as you wrote code to work with filled rectangles. We will see, however, that the interaction of Poles with the rest of your program is a little more complex than that of rectangles.

A `Pole`'s constructor will expect you to specify the point at which it should initially appear and the label that should be displayed (i.e. "N" or "S"). It will also require you to provide as parameters the `canvas` and the `Magnet` to which the new pole will belong. The header of the `pole` class is:

```
class pole, inside (container') at (center': Point)
    isNorth (isNorth': Boolean) on (canvas: DrawingCanvas) -> Pole
```

Since you will usually create the Poles within the code of the `magnet` constructor, the name `self` will refer to the `Magnet` that contains the `Pole`. Thus, the code to construct a `Pole` might look like:

```
pole, inside (self) at (poleLocn) isNorth (true) on (canvas)
```

When evaluated this would create a north pole (because of the `true`) at `poleLocn` on `canvas`, with the pole remembering that the magnet executing this code contains it.

The `Pole` type is defined in `pole.grace` as:

```

type Pole = {
  // return magnet containing the pole
  container -> Magnet

  // return the coordinates of the center of the pole
  center -> Point

  // move the pole by dx to right and dy down
  moveBy(dx: Number, dy: Number) -> Done

  // if close enough to oppositePole then move other magnet to be adjacent
  tryAttract (opposite: Pole) -> Done

  // if close enough to similar then push other magnet away
  tryRepel (similar: Pole) -> Done
}

```

Method `container` returns the magnet containing the pole, while `center` provides the location of the pole (hopefully inside the magnet!). Method `moveBy` just moves the pole by the given amount (just like the graphic items in `objectdraw`).

In addition, the `Pole` class has two more specialized methods: `tryAttract` and `tryRepel`. Each of these methods expects to be passed the `Pole` of another magnet as a parameter. If you say,

```
somePole.tryAttract ( anotherPole )
```

then `somePole` and `anotherPole` should have opposite polarities. If `somePole` is a north pole, then `anotherPole` must be a south pole and vice versa. If the two poles are close enough then executing the method request will result in the magnet associated the the parameter being pulled to be adjacent to the magnet associated with the receiver of the request. (This is all taken care of in the class `pole` that we have written for you. You need not worry about how this is done.)

The `tryRepel` method, on the other hand, assumes that the pole provided as its parameter has the same polarity as the object to which the method is applied. Therefore, if you write:

```
somePole.repel ( anotherPole )
```

and `somePole` is a north pole, then `anotherPole` should also be a north pole. This time if the poles are close enough, executing the method will result in the magnet associated with the parameter being pushed away from the magnet associated with the receiver.

If these methods discover the other pole is close enough to attract or repel, they will use the `moveBy` and `moveTo` methods of the magnets to bring the magnet used as a parameter either closer to the receiver or far enough apart that they would no longer interact.

The good news is that we have already written the code for all the methods described above and will provide these methods to you. *Important: Do not modify the provided `pole` or its type in any way!! It contains sufficient methods to write this program and we will be testing your code with our version of `pole`. If you change it, then it will not run in our tests and you will get very little credit.*

Design of part 1 For the first part of this program, you should just worry about creating the magnets and moving them around. We'll worry about their interactions (attracting and repelling) later. Just like last week, we want you to come to lab with a written design. At the end of this section, we will be more specific about what you should bring with you to lab. To give you a better sense of what we mean by a written design, you can see a sample design for the first part of the laundry lab at <http://www.cs.pomona.edu/classes/cs051G/labs/magnets/LaundrySorterAppletDesignPart1Soln.grace>.

The key to this lab is the design of the `magnet` class. A magnet is represented by a framed rectangle, a filled rectangle (for the red background) and two poles. To ensure that our `Poles` work well with

your `Magnets`, each magnet should be 150 by 50 pixels. The poles should be located near each end of the magnet. We recommend locating them so the distance from the pole to the closest end, top, and bottom, are all $1/2$ the height of the magnet (*i.e.* 25 pixels away from each).

Your magnet class generate objects of type `Magnet`, where `Magnet` will have all the methods that will enable someone running your program to drag magnets about within a window. The `Magnet` type is defined as follows:

```
type Magnet = {
  // location of magnet
  location -> Point

  // move this object to newLocn
  moveTo(newLocn:Point)->Done

  // move this object dx to the right and dy down
  moveBy(dx:Number,dy:Number)->Done

  // Does this object contain locn
  contains(locn:Point)->Boolean

  // Dimensions of magnet
  width -> Number
  height -> Number

  // flip the magnet left-right
  flip -> Done

  // return the poles of the magnet
  northPole -> Pole
  southPole -> Pole

  // move other if it is close enough to this magnet
  interact (other:Magnet) -> Done
}
```

The headers for these methods are already included in the starter file for the `magnet` class. These methods should behave just like the corresponding methods for rectangles and ovals. Method request `someMagnet.location` should return a `Point` value, and `someMagnet.contains(point)` should return a `boolean`. For this portion of the lab we will only be concerned with the methods `moveBy`, `moveTo`, `location`, and `contains`. We will address the others later.

In order to write these methods, your magnet will need to contain several instance variables. A magnet should consist of a filled and a framed rectangle and two poles, and you will need instance variables for each of those. The constructor for a `magnet` needs the following parameters:

- The point corresponding to the upper-left corner of the magnet,
- The canvas that will hold the magnet

The header of the `magnet` class should be:

- **class** `magnet.at(locn': Point) on (canvas:DrawingCanvas)-> Magnet`

It should construct the filled rectangle forming the red background, the framed rectangle forming the outline of the magnet, and should create two poles in the correct positions inside the magnet (see the earlier discussion on the constructor for class `pole`).

Once these instance variables have been declared and initialized, writing the methods should be easy. The `moveBy` and `moveTo` methods should simply move the rectangle and poles to their new positions. The `move` method is pretty straightforward as all three items get moved the same distance, but `moveTo` takes a little thought as `Pole` objects do not have a `moveTo` method. Instead you'll need to calculate how far to move them. (Hint: check to see how far the rectangle is moving from its current position.) The method `location` will simply return the location of the rectangle, while a magnet `contains` a point exactly when the rectangle does.

When you have this much of the `magnet` class written, you can test it by writing `magnetGame`, an extension of the `WindowController` class that creates a magnet, and then write methods `onMousePress` and `onMouseDown` that will allow you to drag it around. We suggest you set the window to be 600 by 600 pixels.

Once `onMousePress` and `onMouseDown` work, it should be pretty easy to add a second magnet and be able to drag either of them around. We suggest declaring a variable (`movingMagnet`) that can be associated with the appropriate magnet and used to remember which magnet to move whenever the mouse is dragged. This variable will be useful in other parts of your assignment as well. (See the Tshirt example from class.)

As you were writing the methods for the `magnet` class, you probably noticed that definitions of `width` and `height` are already included there. As you will learn soon in class, declaring those identifiers to be `public` means that Grace will automatically supply methods that will return their values. These methods are also named `width` and `height`, so you do not need to do anything else to define them. These are used by the `Pole` class in ensuring that the methods `attract` and `repel` draw the magnets appropriately in the window.

As mentioned earlier, you should bring a design with you to lab. The design should show us how you plan to organize your `magnet` class and `magnetGame` object to accomplish the actions required of the first part of this lab only. We have told you what methods each class should have and the behavior that they should provide. You should write (in English, not Grace) your plan for how each method will provide the necessary behavior. You should start with the starter file <http://www.cs.pomona.edu/classes/cs051G/labs/magnets/magnetgame.grace> and modify it to contain the complete design. (Notice that we left out the `pole` class, because we are providing the complete code for it.)

Be sure to describe in your design (in English) what refs and variables you feel are necessary for each class as well as filling in a detailed design for all methods of classes `magnet` and `magnetGame`. This level of preparation will allow you to progress much more quickly in lab so that you can take better advantage of the presence of the instructors and TAs. This week your design will be worth 10% of your lab grade.

Part 2: Flipping the magnet When you click on a magnet, it should reverse its north and south poles. Add method `flip` to class `magnet` that interchanges the north and south poles. Remember that you can move a `Pole`, and one possible way to implement `flip` is to just move the north pole to the south pole's position and vice versa. While the concept is simple, it is easy to mess this up, so be careful and try out your method by hand by showing what happens when you step through your commands slowly on paper.

Add an `onMouseClicked` method to your program that invokes `flip` when the mouse is pressed.

Part 3: Interacting magnets Finally, after you move or flip a magnet, you will need to tell the magnet to check if it is close enough to the other magnet to move it. To make this possible, include a method named `interact` in your `magnet` class. The method `interact` should be invoked on the moving (or changing) magnet, and should take as a parameter the `magnet` that has not just been moved or flipped. It should effect the interaction by calling the `tryAttract` and `tryRepel` methods of its poles with the poles

of the other magnet passed as parameters appropriately. For simplicity, you might want to just check for attraction first, and only worry about repelling after the attraction works correctly.

When writing `interact` you will discover you need to add two more methods in the `magnet` class to enable you to access the other magnet's poles: `northPole` and `southPole`. You may either write these explicitly or have them generated automatically by Grace if you annotated the definitions of these identifiers with `is public`. Both of these methods will return objects with type `Pole`. Also, note that the `attract` method that we have provided in the `pole` class calls the `moveTo` method that you must define in the `magnet` class. If you did not fill in the body of this method correctly, attraction will not work properly. Figure out a way of testing this (e.g., in a `onMouseExit` method that you will erase after doing this testing).

You will need to call the `interact` method every time one of the magnets is either moved or flipped. Because you want to send the `interact` message to the magnet that moved and provide the other magnet as the parameter, you will need to keep track of which is which. As we suggested above, whenever you start dragging a magnet (i.e., in the `onMousePressed` method), you should associate a name with the moving magnet. You will also find it convenient to associate a name with the resting magnet in order to call your `interact` method appropriately.

When your program is finished, your `magnet` class should have a constructor and method bodies implemented for `location`, `move`, `moveTo`, and `contains`, for which headers were provided. In addition, you will need to provide the methods `interact`, `northPole`, `southPole`, and `flip`. You should think carefully about the structure of the method headers for each of these. To help you in formulating your ideas, the following gives typical uses of the methods:

- `someMagnet.interact(otherMagnet)` *// someMagnet & otherMagnet are magnets*
- `def theNorthPole: Pole = someMagnet.northPole`
- `def theSouthPole: Pole = someMagnet.southPole`
- `someMagnet.flip` *// someMagnet is a magnet*

Getting Started The starter folder contains files `pole.grace` and `magnetgame.grace`. The file `magnetgame.grace` should be used to write the `magnet` class as well as the `magnetGame` object that will serve as your “main program”. This file will initially contain skeletons of the code that you will need to complete. The `pole.grace` file will hold our implementation of the `pole` class. Remember, you should not change `pole`.

Due Date This assignment is due at 11 pm on Monday evening.

Submitting Your Work Before submitting your work, make sure that your program file includes your name in the comment heading up the code. Also, before turning in your work, be sure to double check both its logical organization and your style of presentation. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations. Make sure your indentation is all consistent.

Turn in your project the same as in past weeks. Be sure to put your program in a folder whose name includes your name and identifies the lab. Then drag your folder to the dropoff folder.

Table 1: Grading Guidelines

Value	Feature
Design (2 pts total)	
1 pts.	Constants & Variables
1 pts.	Methods
Readability (6 pts total)	
2 pts.	Descriptive comments
1 pts.	Good names
2 pts.	Good use of constants
1 pts.	Appropriate formatting (including whitespace)
Code Quality (4 pts total)	
2 pts.	Good use of boolean expressions
1 pt.	Not doing more work than necessary
1 pt.	Using most appropriate methods
Correctness (8 pts total)	
1 pt.	Drawing magnets correctly at startup
1 pt.	Dragging a magnet
1 pt.	Ability to move either magnet
1 pt.	Moving a magnet to the right place when attracted
1 pt.	On attraction, moving the magnet not pointed to
1 pt.	Flipping a magnet
1 pt.	Attracting and repelling at the right times
1 pt.	No other problems