CSE341 Spring '05 Due Friday, June 3
Assignment 6

The planning segment of the last ML homework has gotten your TA very excited about robots and artificial intelligence. For this assignment, we'll use Squeak to create things that are similar to, but not quite, robots, that fall somewhat short of intelligence of any sort.

The object of this assignment will be to create a miniature ecosystem of mobile robots. Most robots are lowly *seekers*, but there are rare special *pubahs*. Seekers live out their meager existence zooming through the world, bouncing off walls and each other. If a seeker ever bumps into a pubah, it gets excited and tries to stay nearby. Given the limited intelligence of seekers, its attempts to stay nearby usually consists of spinning around in a circle. Some seekers may track their pubah more reliably, or even distinguish between pubahs—but that will be up to you.

Goals

There are two goals for this assignment. The first is the obvious one: for you to learn how to code in a late-binding pure object-oriented language. The second goal, equally important, is to learn how to navigate a modern development environment.

Programming in ML and Scheme was really all about the code. When programming anything else—including any programming you'd do out in the real world—one spends more time navigating the development environment and figuring out how to use existing libraries or tools than writing code.

You'll get plenty of practice browsing around Squeak and figuring out how to do things. Unlike previous assignments, nothing in Squeak is off-limits. If you find a class or message that helps you out, use it!

The Assignment

Create a new morphic project by going to `projects...` from the world menu, and then selecting `create new morphic project`. A little window will be created; click it to enter your new project. From there define your project. You may want to start by drawing some graphics with the painting tool in the widget flap.

Next, use the File List found in the tools flap to filein the `TrafficController.st` class found on the course web. This is a class we wrote that you'll use to detect collisions between your robots. This class also handles bouncing your robots off of each other. This class is described in more detail in a section below. A File List lets you navigate your computer's storage; navigate to where you downloaded the `.st` file in the first and second pane, click on the file in the second pane, the click the `filein` button that appears.

You must complete the following tasks for your assignment.

- You must implement a base class, `Robot`, with two subclasses, `Seeker` and `Pubah`. The two subclasses must have unary `init` methods that display the robot to the screen, add it to the traffic controller, and start it moving. Specifically, `s := Seeker new init` should display a seeker on the screen and assign it to `s`. `Pubah` should behave similarly.

You must use good object-oriented design here, putting as much common code as possible in the base `Robot` class. `Robot` must have `Morph` as an ancestor; my solution has `Robot` a subclass of `ImageMorph`.

- Your robots must display a graphic that rotates to indicate the current direction the robot is traveling in. You must update this at any change in the robot's velocity, including when bounced by the traffic controller. The graphics used for pubahs and seekers must be clearly different and easily identifiable by your TAs (who aren't much smarter than seekers themselves).

  The graphics should be saved as a GIF or other convenient file type. Your robots must read their graphics files in from the current default directory. This is important as otherwise it will be difficult for us to test your code. The sample code contains examples of the correct way to read in graphics files.

- Your robots must bounce off the edge of the screen, which are found in bounds of the owner morph. These can be accessed in any `Morph` subclass by `self owner bounds`.

- In order to be used by the traffic controller, your robots must implement the following methods.

  - `velocity`, which reports the current velocity (a `Point`),
  - `velocity:`, which sets the current velocity to a `Point` (this is probably the right place to update the robot graphic).

- The normal operation of a seeker should be such that it covers a lot of ground. When a seeker collides with a pubah, it must change its behavior to be much more localized, or to follow the pubah. The movement of a seeker must change over time in at least one of its behaviors, for example by shifting its direction.

- Red-clicking on a robot should pause it, so that it doesn't move any more. Red-clicking on a paused robot should release it back into its normal behavior. You shouldn't do anything on a yellow or blue click (leave the default behavior in place).

- The maximum speed of any robot must be limited to 300-400 pixels per second.

The `TrafficController` Class

The traffic controller class is given in a file `TrafficController.st` which can be found on the web page. When this file is read in to Squeak (use the File List tool), it will give you a `TrafficController` class. It may be useful to read the code for `TrafficController` to fully understand how it works.

A `TrafficController` coordinates collisions between robots. Two main selectors are used to interact with the traffic controller, `addRobot:atCollision:` and `collisionsWith:do:`. There's a few more selectors which should be self-explanatory should you need them.

`addRobot:r atCollision:bk` This selector adds `r` to the traffic controller. `bk` is a *callback*, a block that will be called with one argument if `r` is involved in a collision. The argument will be set to another robot that has collided with `r`. If `r` is already in the controller, its block is replaced with `bk`.

`collisionsWith:r do:bk` This selector checks if `r` has collided with any robot registered with the controller. If so, the callback `bk` is called for all such a robots. If `r` is involved in a collision and it's also registered with the controller any block passed in for `r` with `addRobot:` is *not* invoked. If a collision is detected, the traffic controller will change the velocities of the involved objects (through the `velocity` and `velocity:` methods) so that they bounce off of each other. This will be done before any callbacks are evaluated.

Only one traffic controller instance exists at any time, and is accessible from the `control` class method. Using the traffic controller might look like the following.

```
TrafficController control collisionsWith: self do:[:r|...]
```

Hints and Suggestions

The sample code file from the course web contains the code that fleshes out the hints below.

- Subclass `Robot` from `ImageMorph`. Use the `image:` selector with a form from `Form fromFileNamed:` to load in a GIF. GIFs are nice because they can be transparent which makes for a pretty spiffy robot.
- Read in your files from the default directory. The default directory starts out at where your squeak image and changes are. It can be changed by doing `FileDirectory setDefaultDirectory:`*`dir-name`* from a Workspace. Note that the File List will start out in the default directory, which is helpful if you're not sure what it's set to. You should obviously not set the default directory in your code as you have no idea what system we'll be testing your programs in.
- Use `WarpBlt` to rotate the image in your `ImageMorph`. Keep the original image and always rotate from it, otherwise visual artifacts will accumulate with each rotate. Translating your robot can be accomplished by simply translating the `Morph` bounds: `self bounds:(self bounds translateBy:delta)`.
- Use the `step` message to animate your Robot. Look at the comment in its definition in the `Morph` class; you'll need to override both this method and the `stepTime` method. I've found that overriding `stepTime` to return about 100 milliseconds is about right (any less than that doesn't seem translate into smoother animation, at least on my poor little laptop).
- Your `Robot` class should contain all the drawing, rotating and velocity update routines. The `Pubah` subclass may not be anything more than the `init` message; the `Seeker` subclass will contain a more complicated `step` implementation, and very probably a selector or two to keep the block passed to the traffic controller from being too complicated.
- The `click:` selector handles mouse clicks, but it must be activated first. See the comment in the `click:` implementation in the `Morph` class. The message you want to send to the event in the `mouseDown:` message will want to look something like `evt hand waitForClicksOrDrag:self event:evt`. To activate this all you need to override `handlesMouseDown:` to return true if you want to receive the click associated with that event; look for methods containing the word `redButton` in the method finder to figure out what button an event is about.

Extra Credit and Extensions

We've designed this project so there's a lot of latitude for extensions and creativity. As much as possible for a 341 assignment, you should have fun with this.

The obvious extension is to improve a seeker's search and reaction to bumping into a pubah. An obvious way would be to use the reference to the pubah acquired with the collision. More advanced strategies would add methods to seekers to expose what they know about the location of the pubah or pubahs, so that when seekers bump into each other seekers that haven't found the pubah can find it more quickly.

Further extensions would enrich the game a little bit, such as counting the number of times each seeker bumps into the pubah, or dividing seekers up into multiple teams and seeing which team can find all pubahs first.

Another direction to go is to improve the existing physics of the game. As reading through `TrafficController` will show, collision handling is quite primitive and could be much improved to, for example, take into account the actual shape of the robots rather than just the circular approximation used. Other fun things might be to vary the mass of robots. This could make much more entertaining collisions.

There's a lot of other directions to go. Talk to the professor or any of the TAs for any ideas or if you have questions. Remember that you should complete the base assignment before doing an extra credit. Make totally new classes for any extra credit you do, so that we can grade your base assignment separately.

Turning In

Your three classes (`Robot`, `Pubah`, `Seeker`) should be made in the category `341`. No other classes should be in that category. For the base assignment, you should not need to define any other classes.

To turn-in, fileout the `341` category (yellow-click on it in a browser and select `fileOut`). Zip or tar that with your graphics files and submit that to the link we'll put on the class link. For example, your archive might include the files `341.st`, `seeker.gif`, `pubah.gif`. Your archive **should not contain any directories**. Your turnin files **should not be inside a directory** in the archive. E-mail us if you're confused about this.

If you have any extra-credit, put it in a category `extra-credit`. File that out and put it in the same archive as your base turnin. Include a file `readme.txt` in your archive that explains what your extra-credit does, and how to make it work. You'll only get credit for what you explain in this file.