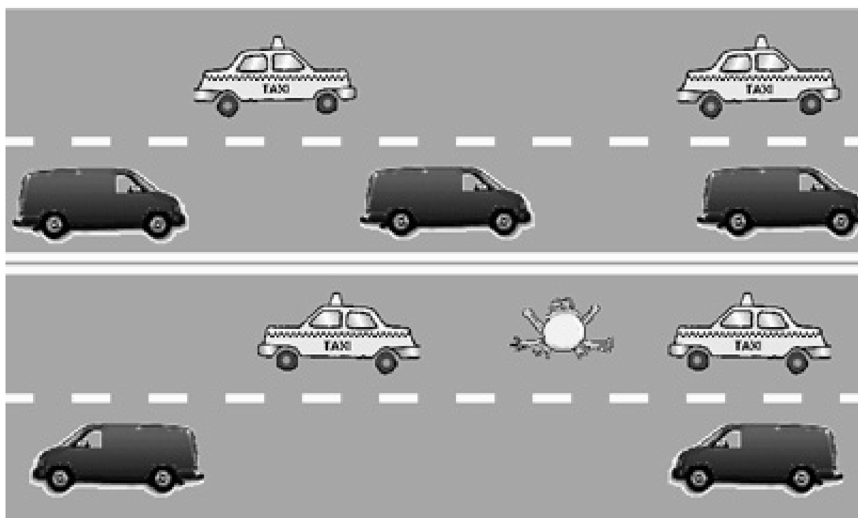
CS 051 Homework Laboratory # 5
It's not Easy Being Green

Objective: To gain experience with loops and active objects.

—

The Scenario For this lab, we would like you to write a program that plays the game frogger. In this game, you control a frog that is trying to cross a busy 4-lane highway. Each lane has cars or trucks zooming by. The vehicles in a given lane all travel at the same speed, but vehicles in different lanes may travel at different speeds (and even in different directions if you would like). The user is in control of a frog. Clicking in front of the frog moves it forward one hop (one hop is the width of a lane of traffic), clicking behind moves it back, and similarly for clicking to the left and right of it. The goal is for the user to get the frog across the highway without it getting squished.

If the frog does get squished it should display an "OUCH!" message at the bottom of the screen or change color to indicate the change in state. The user should be able to restart the frog from its original starting position by clicking the mouse once in the area below the lanes of the highway. In that case the message should disappear or the frog should change back to its original color.



Images for the program Images for the frog and vehicles can be found on line at http://www.cs.pomona.edu/classes/cs051G/Images/froggy.png, http://www.cs.pomona.edu/classes/cs051G/Images/taxi_right.png, http://www.cs.pomona.edu/classes/cs051G/Images/jeep_right.png, http://www.cs.pomona.edu/classes/cs051G/Images/jeep_left.png, and http://www.cs.pomona.edu/classes/cs051G/Images/taxi_left.png.

Design We are going to try something a bit different this time. Rather than turning in a regular design for each class, we want you to instead fill out the Frogger Planning Sheet. It directs you to think about particular problems that you will encounter with Frogger and how you will solve them. Please turn in a printed copy of this when you enter the lab so that we can grade them when you are taking the quiz.

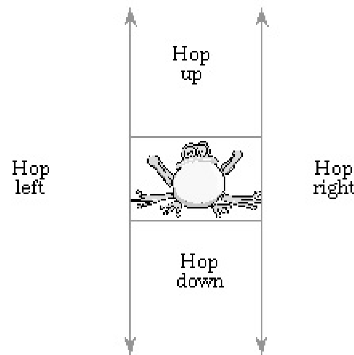
Preparing for this lab Read this entire handout before doing anything else. In this handout we will begin by describing the classes you need to implement. After we describe the classes, we will outline one plan for proceeding with your implementation.

The Objects As you learned in the past two weeks with the lightballoon and boxball labs, the key to good object design is the design of classes to represent the objects in your program. There are three different kinds of objects involved in the frogger game (aside from the game object itself). There are the frog, objects that fill lanes of traffic with new vehicles, and there are the vehicles that go on the road. (There are also some graphical objects on the screen to represent the lane markings on the highway, but we won't discuss these in detail as they are easily constructed. In fact, we have provided in the starter folder the code that draws the highway markings.)

The Frog The frog is a very important object. The frog will be displayed on the screen by creating a graphical image of the frog (its complexity is up to you). The `frog` class will need one or more `def`s to keep track of its parts.

The `frog` class should create the frog image and place it just below the lowest lane of the highway, approximately halfway from each end of the highway. Your class will require several parameters, including the location where the frog should start, how wide a lane is (so the frog knows how far to hop), and the canvas.

The frog clearly needs to be able to hop in each of the four directions in response to a user click. We suggest writing a single method `hopToward` that takes a `Point` parameter representing where the user clicks. Depending on which side of the frog the point is on, the frog should move in the appropriate direction. To keep the testing required to determine how the frog should jump simple, we suggest you divide the space around the frog as shown in the diagram below.



Unfortunately, the other thing that happens to a frog is that it gets splattered on the road. A vehicle will be responsible for determining whether it has killed a frog. To do so, it will need to ask the frog if any part of the frog's body overlaps the image that represents the vehicle. To make this possible, your `frog` class should include the definition of an `overlaps` method that takes a rectangle (the outline of the car) as a parameter and returns a boolean.

If a vehicle hits the frog, it kills the frog by sending a `kill` message to the frog. The frog's kill method causes an "OUCH!" message to appear at the bottom of the screen or causes it to change color.

Finally, through the miracles of modern medicine, we'd like our frog to be able to come back to life. Add a method `reincarnate` which moves the frog back to its starting point and `show`s the image. Of course, you should not reincarnate a frog unless it is dead. So, include a boolean instance variable that

keeps track of the condition of the frog (alive or dead), and check this variable in `reincarnate`. As described above, the user will reincarnate the frog by clicking the mouse below the highway. The `frog` class should include an `isAlive` accessor method that returns a boolean to enable the `onMousePress` method to determine whether the frog should hop or possibly be reincarnated. (This can be an explicit method or just a variable that is declared to be `readable`, which you will recall implicitly creates such a method.)

The following is the type that should be generated by class `frog`:

```
type Frog = {
    overlaps (other: Graphic2D) -> Boolean
    kill -> Done
    reincarnate -> Done
    hopToward (point: Point) -> Done
    isAlive -> Boolean
}
```

You should fill in comments for what each method does.

The Vehicles The class vehicle will need lots of parameters that specify, among other things, where the vehicle should be created, the URL of the image, information about where the road starts and ends, and the velocity with which the vehicle should move. In order to check if the vehicle runs over the frog, you also need to pass the frog as a parameter to the `vehicle` class.

The class `vehicle` will be like our other animated objects and won't really have any methods aside from `start`. In the previous (boxball) project we moved a ball a fixed distance each time we went through the while loop. On a busy system, we may wind up pausing for longer than we expected, and this can result in erratic movement. The standard solution to this problem is to check how long we actually paused, and make the distance we move proportional to that time.

Inside the run while loop:

1. Save the current time. (Get the current time by calling sys.elapsedTime if the file includes an `import "System"as sys`).
2. Pause for (at least) 30 milliseconds.
3. Determine how long it actually paused for (e.g., subtract the time saved earlier from the current time) and move the appropriate distance. (Remember your equation from physics: rate * time = distance.) You need to do this to ensure smooth motion of your vehicles. With so many active objects, the computer cannot ensure that the length of pauses will be very precise.
4. Find out if the vehicle squished the frog and kill the frog with the kill method if it did.

You may assume that all vehicle velocities will be positive. That is, your highway may be a one-way street. You are encouraged (for extra credit), however, to add the ability to handle vehicles with negative velocities so that you can have some lanes where traffic goes from left to right and others where traffic moves from right to left. That way you can model a two or four lane highway where the bottom lanes go from left to right and the top ones go from right to left.

The laneFiller class The class `laneFiller` will be used to create an object for each of the `numLanes` lanes in your simulation. As such, the lane does not actually correspond to any drawing on the screen. Its job is to create streams of cars for each of the lanes.

As you recall from above, the vehicle class requires a lot of parameters: its starting location, its image, etc. So far all the lane knows is how fast the cars drive. Where will Lane get this other

information? The car should be located at one end of the lane initially and should drive until it reaches the other end. If the lane knew where it was located, it could pass this information on to the car. Its location is relative to the entire highway. Our window controller can provide this information to the lane when it constructs it so that the lane can pass the information on to the vehicle.

What about the image? Rather than copying the URL into the vehicle class, you should create a constant in the laneFiller class or main program. It can then be passed (if necessary) to the laneFiller class and the laneFiller class can pass it on to each vehicle. This organization will make it easier later on to have different vehicle images for each lane (or even multiple images in each lane). The lane constructor remembers the image of the cars for its lane and passes this image to the vehicle class.

Finally, the vehicle needs to know about the frog so it can tell if it hit the frog. Again, the main program created the frog. It can pass the frog to the Lane in its constructor. The lane can remember the frog so that it can tell the vehicles about the frog in the Vehicle constructor call.

The class `laneFiller` will (among many others) take parameters indicating where cars should start in that lane and what speed those cars should be traveling. Its `start` method should use the method `while()pauseVarying()do()` from module "animation" to generate cars at intervals.

We want you to use `while()pauseVarying()do()` because we do not want successive pairs of cars to have exactly the same gap between each other. Instead we want the gaps to be randomly chosen to be between 1 and 4 car lengths. We'll come back to this in a moment. This new method differs from the usual `while()pause()do()` because the argument after `pause` must be in curly braces. This is because it must be re-evaluated every time through the loop (after all, we want the gaps between cars to vary during the game).

All of the traffic in a lane should drive at the same speed so that cars do not run into each other. The main program should pick a random speed for each of the lanes and pass it in to the `laneFiller` class. We have found speeds in the range 0.033 to 0.1 pixels/millisecond to be good (though you may want to start with a slower set of speeds while you debug your program). You will want to use the method `between (low) and (high)` from module `random` rather than `integerIn (low) to (high)` because you want fractional numbers to be generated. This method gives uniformly distributed real numbers in the range from low up to (but not including) high. Thus `rand.between (1) and (2)` gives numbers like 1.34, 1.01, 1.9999, but never .999 or 2.0.

The `laneFiller`'s `start` method's main responsibility is to periodically place a new car on the screen. Using the `while()pauseVarying()do()` method of `animator`, the lane should generate a car, wait a while to allow a gap between cars, generate another car, and so on.

How long should the gap be? The pause should be at least long enough so that there will be a one car-length gap between pairs of vehicles. The pause should never be so long that it leaves more than about four car lengths between vehicles.

If a vehicle is moving with a certain velocity, how long must it move before it's a full vehicle-length away from its original location? As an example, suppose the speed is 2 pixels/millisecond and the car is 140 pixels long. How long until the vehicle has gone one car length? How long until there is a gap of one car length between two vehicles? The answers to these two questions are different!

For simplicity, you may use a fixed value for the pause time when you first write your program, but eventually the pause time must be selected randomly. That is, the gaps between successive cars should not be the same, but should be distributed randomly with the constraint that there is at least one car length and no more than 4 car lengths between successive cars in the same lane.

Warning: A very common mistake is to miscalculate the gaps with an off-by-one error so that the gaps are 0 to 3 car lengths rather than 1 to 4. Try out your calculations by hand to make sure they are correct!

The Main Program As usual the main program will be an object that will `inherit graphicApplication` The initialization code for the main program will draw the highway (we have provided code for that) and create the frog. It must also create a steady stream of cars in each of the four lanes. Of course it is also responsible for moving the frog when the user clicks the mouse.

Getting Started The "starter" folder contains four program files: `Frogger.grace`, `Frog.grace`, `LaneFiller.grace`, and `Vehicle.grace`. We chose to separate the program into the four files to save you time in recompiling the pieces. When you make a change to one of the pieces, you need only recompile (build) that piece and any files that import it.

For example, if you make a change just to the `Frogger.grace` file, then you need only build and run it without worrying about the other files, as none of the other files import it. On the other hand, if you make a change to `Frog.grace` then you must also build all the other files as they all import `Frog`.

The files mentioned above contain skeletons of code which you will need to complete to get this program to work. They are accessible via these links:

- http://www.cs.pomona.edu/classes/cs051G/labs/frogger/Frogger.grace
- http://www.cs.pomona.edu/classes/cs051G/labs/frogger/Frog.grace
- http://www.cs.pomona.edu/classes/cs051G/labs/frogger/LaneFiller.grace
- http://www.cs.pomona.edu/classes/cs051G/labs/frogger/Vehicle.grace

Don't forget that the `frogger` object contains code that draws the highway background and markings for you.

There are many ways of proceeding with implementing this program. Here is one suggested ordering that we have found works well for students:

1. Read the given code in `frogger` to understand how we've drawn the highway background for you.
2. Write the `frog` class except for the `kill` and `reincarnate` methods.
3. Modify the `frogger` object to create the frog on the screen, and write `onMousePress` to control the movements of the frog. Make sure that the frog hops around appropriately.
4. Write the `vehicle` class.
5. Test the `vehicle` class by adding simple initialization code in the `frogger` object to put one car on the road. Move the frog into the road to see if it gets killed by the car. Add code to reincarnate the frog and test it.
6. Write the class `fillLane`. Write the code for the `start` method so that a lane generates a stream of cars on the lane (removing the similar code you wrote in the previous item in frogger). Make sure they don't bump into each other (they shouldn't as they are all going the same speed). Make sure that the frog gets killed if hit by any of the cars (though our cars will be hit-and-run – they don't stop!).

Advanced Features For some extra credit create vehicles that move in both directions so that you can have the bottom lanes going from left to right and the top going from right to left. Only attempt this, however, after completing the construction of a program that meets the basic requirements.

Submitting Your Work Before submitting your work, make sure that the `.grace` file includes a comment containing your name. Also, before turning in your work, be sure to double check both its logical organization and correctness. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations. Refer to the lab style guide for more information about program presentation.

Turn in your project the same way as in past weeks. Be sure the project's folder is labeled in the following form: `LastNameFirstName_LabXX`. Because this lab is more complex, you get an extra day. Thus your lab is due at 11 p.m. on Tuesday evening. However, I strongly urge you to get an early jump on the program as it will likely take you a substantial amount of time outside lab to complete it.

Table 1: Grading Guidelines

Value	Feature
	Design preparation (3 pts total)
1 pt.	Frog leaps
1 pt.	Vehicle generation
1 pt.	Frog killing
	Code readability (4 pts total)
1 pts.	Descriptive comments
1 pts.	Good names
1 pts.	Good use of constants
1 pt.	Appropriate formatting
	Code Quality (5 pts total)
1 pt.	Good use of boolean expressions
1 pt.	Not doing more work than necessary
1 pts.	Using most appropriate methods
1 pts.	Good use of if and while statements
1 pts.	Good choice of parameters
	Correctness (87 pts total)
1 pt.	Car moves smoothly
1 pt.	Car disappears at end of lane
1 pt.	Car kills frog appropriately
1 pt.	Cars spaced appropriately in lane
1 pt.	Frog moves correctly on clicks
1 pt.	Frog says "ouch" when killed
1 pt.	Frog cannot move when dead
1 pt.	Frog reincarnated properly