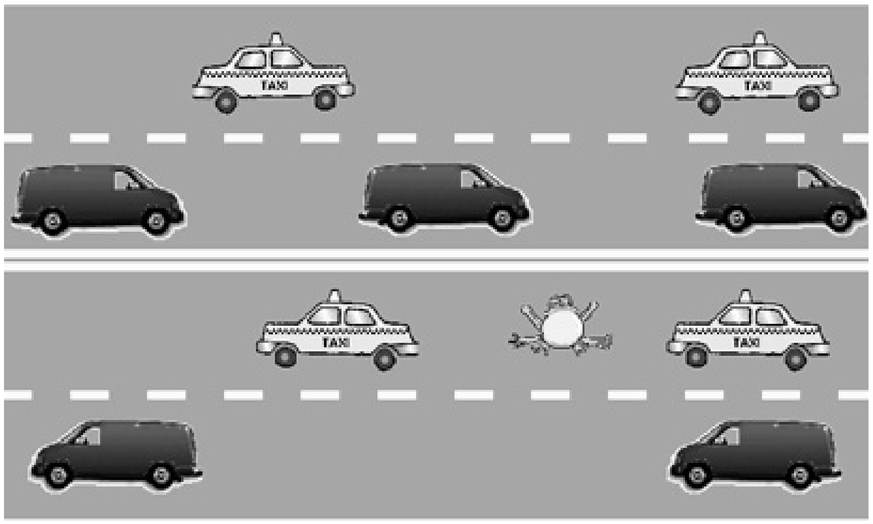
CS 051 Homework Laboratory # 5
It's not Easy Being Green

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.



Design A design for the classes `Frog` and `Vehicle` as well as the main program/object, `frogger`, is due at the beginning of lab. While we expect a design for *all* aspects of the program, we urge you to write and debug the program as suggested in the **Getting Started** section later in this handout.

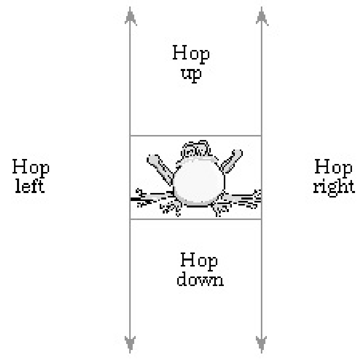
Preparing for this lab Read this entire handout before doing anything else. In this handout we will begin by describing the classes you need to implement. After we describe the classes, we will outline one plan for proceeding with your implementation.

The Objects As you learned in the past two weeks with the magnet and boxball labs, the key to good object design is the design of classes to represent the objects in your program. There are two different kinds of objects involved in the Frogger game. There is the frog 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.

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, 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` constructor.

The class `vehicle` will be unusual because it won't really have any methods. It will instead do a lot of initialization and then run a while loop with pauses to animate the vehicle. The while loop should:

1. Move all the components of the car.
2. Find out if the vehicle squished the frog and kill the frog with the `kill` method if it did.

You can calculate the distance that the car components should be moved by multiplying the speed by the time between redrawing the images. You may assume that that is the pause time specified in your `while()pausing()do()` loop, but for extra credit, measure the actual time between redrawing (remember it is always greater than or equal to the specified time, but often greater) using `system.elapsed` and calculating the difference in times between successive redraws.

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 four lane highway where the bottom two lanes go from left to right and the top two go from right to left.

The Main Program As usual the main program will be an object that **inherits** `graphicApplication` The constructor 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.

We suggest that you create a confidential method `fillLane` that can be invoked four times, once for each of the four lanes, to create streams of cars for each of the four lanes. The method `fillLane` would take parameters indicating where cars should start in that lane and what speed those cars should be traveling. The 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.

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 `fillLane` method. 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 `randomNumberFrom()to()` rather than `randomIntFrom()to()` because you want fractional numbers to be generated.

The `fillMain` method's main responsibility is to periodically place a new car on the screen. In 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!

Getting Started The "starter" folder contains the file `Frogger.grace`. This file contains skeletons of code which you will need to complete. The `frogger` object contains code that draws the highway background and markings for you. A listing of the contents of the `Frogger.grace` file included in the starter folder is attached to this handout.

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 having the initialization code in the `frogger` object 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 method of `fillLane`. Move the code that creates a car to the `fillLane` method so that a lane generates a stream of cars on the lane. 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 two lanes going from left to right and the top two 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 includes your name in the title. As usual, your lab is due at 11 p.m. on Monday evening.

Table 1: Grading Guidelines

Value	Feature
	Design preparation (3 pts total)
1 pt.	instance variables & defs
1 pt.	class headers and parameters
1 pt.	methods and parameters
	Documentation (5 pts total)
2 pts.	Descriptive comments
1 pts.	Good names
1 pts.	Good use of constants
1 pt.	Appropriate formatting
	Program Quality (5 pts total)
1 pt.	Good use of boolean expressions
1 pt.	Not doing more work than necessary
1 pts.	Check if killed and reincarnate at appropriate times
2 pts.	Parameters used appropriately
	Correctness (7 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 reincarnated properly

Frogger.grace

```
dialect "objectdrawDialect"
import "Animation" as animator

type Frog = {
    overlaps (other : Graphic2D)−>Boolean

     kill −>Done

     reincarnate −>Done

    hopToward(point: Point) −> Done

     isAlive −> Boolean
}

class frog . atX(highwayCenterX:Number)below(highwayBottom:Number)
                laneWidth(laneWidth:Number)on(canvas:DrawingCanvas) −> Frog {
    def frogHeight = 48

    def hopDistance:Number = laneWidth

    // fill in missing defs and vars

    method isAlive −> Boolean {
          false    // FIX!!
    }

    method overlaps(other:Graphic2D) −> Boolean {

    }

    method kill −> Done {

    }

    method reincarnate −> Done {

    }

    method hopToward(point: Point) −> Done {

    }
}


def carHeight = 40

class Vehicle . someStuff(params) // fill in method name and parameters
{
    def pauseTime: Number = 30

    // fill in the blanks ...

}

// Application that plays frogger game
def frogger: GraphicApplication = object {
    inherits graphicApplication.size(800,600)

    // Constants defining the sizes of the background components.
    def highwayLength:Number = 700
    def laneWidth:Number = 100
    def numLanes:Number = 4
    def highwayWidth:Number = laneWidth * numLanes
    def lineWidth:Number = laneWidth / 10

    // Constants defining the locations of the background components
    def highwayLeft:Number = 50
    def highwayRight:Number = highwayLeft + highwayLength
    def highwayTop:Number = 100
    def highwayBottom:Number = highwayTop + highwayWidth

    // Constants describing the lines on the highway
    def lineSpacing:Number = lineWidth / 2
    def dashLength:Number = laneWidth / 3
    def dashSpacing:Number = dashLength / 2

    def carWidth = 100

    // start constructor code
     filledRect.at(highwayLeft @ highwayTop)size( highwayLength,
                                  highwayWidth)on(canvas)

    var whichLine:Number := 1

    while {whichLine < numLanes} do {
        if (whichLine == (numLanes / 2)) then {
            print "draw no passing"
            // The middle line is a no passing line
            drawNoPassingLine((highwayTop + (whichLine * laneWidth))
                                    - ((lineSpacing / 2) + lineWidth))
        } else {
            print "passing lane"
            drawPassingLine((highwayTop + (whichLine * laneWidth))
                                  - (lineWidth / 2))
        }
        whichLine := whichLine + 1
    }

    method drawNoPassingLine(y:Number)-> Done {
        def topLaneStart:Point = highwayLeft @ y
        // Draw the solid dividing lines
        def topLine:Graphic = filledRect .at(topLaneStart) size( highwayLength,
                                    lineWidth) on (self.canvas)
        topLine.color := yellow

        def bottomLaneStart:Point = topLaneStart+ (0 @ (lineWidth + lineSpacing))
        def bottomLine:Graphic = filledRect .at(bottomLaneStart)
                                          size(highwayLength, lineWidth) on (self.canvas)
        bottomLine.color:=yellow
    }

    method drawPassingLine(y:Number)-> Done {
        var x:Number := highwayLeft
        var dash:Graphic

        while {x < highwayRight} do {
            // Draw the next dash
            dash := filledRect .at(x @ y) size (dashLength, lineWidth) on (self.canvas)
            dash.color := white
            x := x + dashLength + dashSpacing
        }

    }

    // Lots of stuff omitted!


    // Move the frog when mouse is clicked.
    method onMouseClick(mousePoint:Point)->Done{

    }

    startGraphics

}
```