

Lab 5

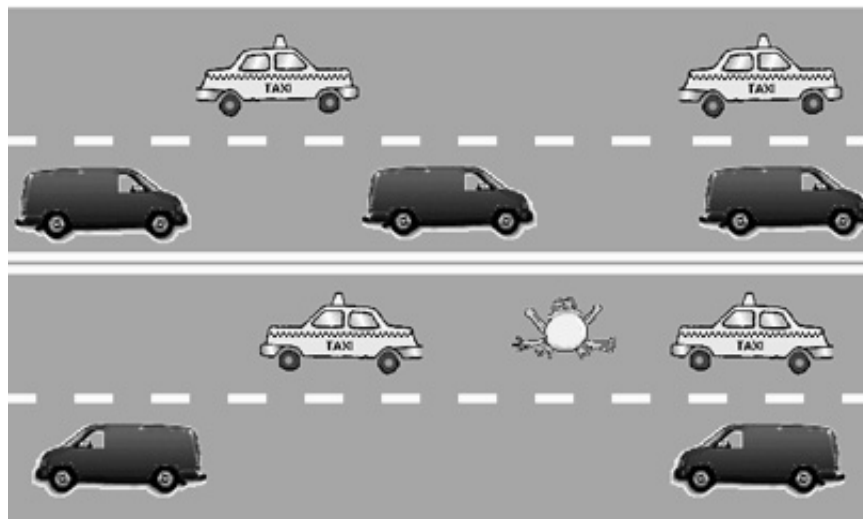
Frogger

Objective To gain experience with loops and active objects.

The Scenario For this lab, you will 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).

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 by a car or truck.

If the frog does get squished, it should display an “OUCH!” message at the bottom of the screen. The user should be able to restart the frog from its original starting position by clicking the mouse in the area below the lanes of the highway. A demo version of this program can be found on the handouts page of the class website. Playing with the demo will give you an idea of what we have in mind for the program.



Design A design for all of the classes used in this program (Frogger, Frog, Vehicle, and Lane) 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.

Frogger Overview

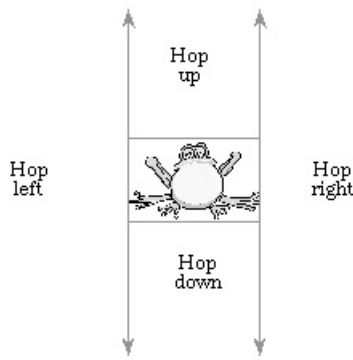
There are three important classes of objects involved in the Frogger game. There is the frog, there are the lanes of traffic (four of them), and there are the vehicles that go on the road. (There are also some graphical objects on the screen to represent the lane markings on the highway, but we will

not discuss these in detail as they are easily constructed. (In fact, the code that draws the highway markings is provided in the starter folder.)

The Frog The frog will be displayed on the screen by creating a `VisibleImage` using an image file we will provide. The `Frog` class will need an instance variable to keep track of this `VisibleImage`.

The constructor for a `Frog` should create the `VisibleImage` and place it just below the lowest lane of the highway, approximately halfway from each end. Your constructor will require several parameters: the `Image` of the frog, the location where the frog should start, how wide a lane is (so the frog knows how far to hop), and the canvas. *Remember, only `WindowController` objects can call `getImage`. Thus, get the image in the `Frogger` class and pass it to the `Frog` constructor as we suggest.*

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 `Location` parameter (i.e., a point). Depending on which side of the frog the point is, 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 `VisibleImage` displayed to represent the vehicle. To make this possible, your `Frog` class should include the definition of an `overlaps` method that takes a `VisibleImage` as a parameter and returns a boolean.

If a vehicle hits the frog, it kills the frog by calling a `kill` method on the frog. This causes an "OUCH!" message to appear at the bottom of the screen.

Finally, through the miracles of modern medicine, the frog can come back to life. Add a method `reincarnate` which moves the frog back to its starting point. 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 Vehicles The constructor for a vehicle will need parameters specifying where the vehicle should be located (two `doubles` or a single `Location` parameter), the `Image` used to display the vehicle on the screen, the velocity with which the vehicle should move, the distance the vehicle should travel before disappearing, and the `DrawingCanvas`. 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.

`Vehicle` extends `ActiveObject`. Thus, you must define the method

```
public void run() { ... }
```

The while loop inside `run` should:

1. Save the current time. (Get the current time by calling `System.currentTimeMillis()`).

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: $\text{rate} * \text{time} = \text{distance}$.) 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 free (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.

The Lane The purpose of the `Lane` object is to continuously generate the vehicles that travel in a particular lane on your highway. As such, the lane does not actually correspond to any drawing on the screen. Instead, a `Lane` will be an `ActiveObject` that creates `Vehicles`. This is similar to the way in which the `Tree` objects was an active object that created `FallingLeaf` objects in the `Fall` example from lecture. We recommend refreshing your memory by looking at the code for that example while preparing your `Lane` class.

The constructor for a `Lane` is quite simple. The lane was already drawn as part of the background in our window controller so we do not need to update the display. One thing we know is that all the traffic in a lane should drive at the same speed so that cars do not run into each other. The `Frogger` class should pick a random speed for each of the lanes and pass it in to the `Lane` constructor. We have found speeds in the range .033 to .133 pixels/millisecond to be good (though you may want to start with a slower set of speeds while you debug your program).

A lane's main responsibility is to periodically place a new car on the screen. It will do this inside the `run` method. In the while loop of the `run` method, the lane should generate a car, wait a while to allow a gap between cars, generate another car, and so on.

Recall from above that the `Vehicle` constructor 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? Since the image can only be loaded from the window controller, so this information must also be passed down to the `Vehicle` by passing it to the `Lane` constructor. The `Lane` constructor remembers the image of the cars for its lane and passes this image to the `Vehicle` constructor. (Because of this limitation, it is simplest for all the cars in a lane to have the same image.)

Finally, the vehicle needs to know about the frog so it can tell if it hit the frog. Since our window controller 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.

After generating one vehicle, the lane should pause for some time. 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.

For simplicity, you may use a fixed value for the pause time when you first write your program, but eventually the pause time should 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. To do this, your code will have to use the speed with which vehicles travel and their lengths in pixels to compute the minimum and maximum amount of time you should pause between generating new cars.

The Image Files There is an image file, names `froggy.gif`, for the frog. We provide 8 images of vehicles. They show 4 different types of vehicles. For each type of vehicle there is an image of that

vehicle facing right and another facing left. The image files are included in the starter folder and are named:

```
jeep_left.gif    oldcar_left.gif    taxi_left.gif    van_left.gif
jeep_right.gif   oldcar_right.gif   taxi_right.gif   van_right.gif
```

It is fine if you only use a single image file so that all your vehicles look the same and move in the same direction. Using multiple pictures is a feature that we hope you add (it does make the display look better), but it is not required.

The frog is 83 pixels wide and 48 pixels tall. The widest vehicle is 139 pixels wide. The tallest vehicle is 66 pixels tall. This information should help you figure out how to place the vehicles and frog within the lanes. Remember to use constants effectively so that you would not need to change many values if we introduced a much taller vehicle, for example.

Getting Started

We have provided a starter project for you to use. To download the starter project, visit

<http://www.cs.williams.edu/~cs134/>

and then follow the link to the Handouts page. Click on the link for “Starter Code” under Lab 5.

This will download a file archive called `Lab5Frogger.tar.gz` to your `cs134` folder. The files of this archive will then be extracted to a folder in your `cs134` folder called `Lab5Frogger`. (If this does not happen automatically, simply double-click on the downloaded `Lab5Frogger.tar.gz` file to extract the archive.) You may delete the `Lab5Frogger.tar.gz` file at this point. Rename the “Lab5Frogger” folder to include your last name, as usual, and open the project folder in BlueJ.

The “starter” folder contains several files intended to hold Java code. The file `Frogger.java` should be used to write the extension of the `WindowController` that will serve as your “main program”. The `Frog.java` file should be used to hold your code for the `Frog` class, `Lane.java` will hold the code for the `Lane` class, and `Vehicle` hold the code for the `Vehicle` class. The `Frogger.java` file contains code that draws the highway background and markings for you. All the files contain skeletons of code which you will need to complete. These skeletons are printed at the end of the handout for your reference.

There are many ways of proceeding with this lab. Here is one suggested ordering:

1. Read the `begin` method in `Frogger.java` to understand how we have drawn the highway background for you.
2. Write the `Frog` class except for the `kill` and `reincarnate` methods.
3. Modify the `Frogger` class 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 `begin` method in the `Frogger` class put one car on the road. (*You do not want the code here in the end, but for now it lets you test the `Vehicle` class before you have written the `Lane` class.*) 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 constructor of `Lane`. Move the code that creates a car from the `Frogger` class to the `run` method of the `Lane` class so that a lane generates a stream of cars on the lane. Make sure they do not bump into each other (they should not 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 do not stop!).

Advanced Features You may notice that the sample version of Frogger we have provided (and depicted in the image at the beginning of this handout) has more features than we have required. Vehicles move in both directions and several different images are used to represent vehicles. Feel free to incorporate these or any other extensions in your program. Only do so, however, after completing the construction of a program that meets the basic requirements.

Submitting Your Work

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment:

- First, return to the Finder. You can do this by clicking on the smiling Macintosh icon in your dock.
- Click “Go” and select “Connect to Server.”
- For the server address, type in “Cortland” and click “Connect.”
- Select the button next to “Guest” and click “Connect.”
- A selection box should appear. Select “Courses” and click “Ok.”
- You should now see a Finder window with a “cs134” folder. Open this folder.
- You should now see the drop-off folders for the three labs sections. As with last week, drag your program folder into the appropriate lab section folder. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click “OK.”
- Log off of the computer before you leave.

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Labs, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before the 11 p.m. deadline. The Mac will not let you submit again unless you change the name of your folder slightly. It does this to prevent another student from accidentally overwriting one of your submissions. Just add something to the folder name (like “v2”) and the resubmission will work fine.

Grading Guidelines

As on labs, we will evaluate your program for both style and correctness. Here are some specific items to keep in mind and focus on while writing your program:

Design preparation

- instance variables and constants
- constructors and parameters
- methods and parameters

Style

- Descriptive comments
- Good names
- Good use of constants
- Appropriate formatting

Program Quality

- Good use of boolean expressions
- Not doing more work than necessary
- Check if killed and reincarnate at appropriate times
- Parameters used appropriately

Correctness

- Car moves smoothly
- Car disappears at end of lane
- Car kills frog appropriately
- Cars spaced appropriately in lane
- Frog moves correctly on clicks
- Frog says “ouch” when killed
- Frog reincarnated properly

Frogger.java

```
public class Frogger extends WindowController {

    // The size of the window for Frogger
    private static final int SCREEN_WIDTH = 800;
    private static final int SCREEN_HEIGHT = 644;

    // The sizes of the background components.
    private static final double HIGHWAY_LENGTH = 700;
    private static final double LANE_WIDTH = 100;
    private static final int NUM_LANES = 4;
    private static final double HIGHWAY_WIDTH = LANE_WIDTH * NUM_LANES;
    private static final double LINE_WIDTH = LANE_WIDTH / 10;

    // The locations of the background components
    private static final double HIGHWAY_LEFT = 50;
    private static final double HIGHWAY_RIGHT = HIGHWAY_LEFT + HIGHWAY_LENGTH;
    private static final double HIGHWAY_TOP = 100;
    private static final double HIGHWAY_BOTTOM = HIGHWAY_TOP + HIGHWAY_WIDTH;

    // The lines on the highway
    private static final double LINE_SPACING = LINE_WIDTH / 2;
    private static final double DASH_LENGTH = LANE_WIDTH / 3;
    private static final double DASH_SPACING = DASH_LENGTH / 2;

    // Size of the frog image
    private static final double FROG_WIDTH = 83;
    private static final double FROG_HEIGHT = 48;

    /*
     * This method current just draws the highway. You will have to
     * add instructions to create the frog and the Lane ActiveObjects.
     */
    public void begin() {
        // resize window
        this.resize(SCREEN_WIDTH, SCREEN_HEIGHT);

        // draw the highway using the methods below.
        this.createHighway();

        // ADD YOUR CODE TO CREATE THE FROG AND THE LANES
    }

    /*
     * Note: Use onMousePress rather than onMouseClick to decide when
     * to move the frog.
     */
    public void onMousePress(Location point) {
        // ADD CODE HERE
    }

    /*
     * Draw the highway background.
     * YOU SHOULD NOT NEED TO MODIFY THIS METHOD
     */
}
```

```

*/
public void createHighway() {
    // Draw the pavement
    new FilledRect(HIGHWAY_LEFT, HIGHWAY_TOP, HIGHWAY_LENGTH, HIGHWAY_WIDTH, canvas);

    // Draw the lane dividers
    int whichLine = 1;
    while (whichLine < NUM_LANES) {
        if (whichLine == (NUM_LANES / 2)) {
            // The middle line is a no passing line
            this.drawNoPassingLine((HIGHWAY_TOP + (whichLine * LANE_WIDTH)) -
                ((LINE_SPACING / 2) + LINE_WIDTH));
        } else {
            this.drawPassingLine((HIGHWAY_TOP + (whichLine * LANE_WIDTH)) -
                (LINE_WIDTH / 2));
        }
        whichLine = whichLine + 1;
    }
}

/*
 * Draws a pair of solid yellow lines to represent a no passing
 * divider. The parameter y is the top of the top line.
 * YOU SHOULD NOT NEED TO MODIFY THIS METHOD
 */
public void drawNoPassingLine(double y) {
    FilledRect topLine = new FilledRect(HIGHWAY_LEFT, y, HIGHWAY_LENGTH,
        topLine.setColor(Color.yellow);

    FilledRect bottomLine = new FilledRect(HIGHWAY_LEFT,
        y + LINE_WIDTH + LINE_SPACING,
        HIGHWAY_LENGTH, LINE_WIDTH,
        canvas);

    bottomLine.setColor(Color.yellow);
}

/*
 * Draws a dashed white line to represent a passing line dividing
 * two lanes of traffic The parameter is the top of the line.
 * YOU SHOULD NOT NEED TO MODIFY THIS METHOD
 */
public void drawPassingLine(double y) {
    double x = HIGHWAY_LEFT;
    while (x < HIGHWAY_RIGHT) {
        // Draw the next dash.
        FilledRect dash = new FilledRect(x, y,
            DASH_LENGTH, LINE_WIDTH,
            canvas);

        dash.setColor(Color.white);
        x = x + DASH_LENGTH + DASH_SPACING;
    }
}
}

```

Frog.java

```
public class Frog {

    // This should refer to the image of the frog. Note that it is not
    // initialize by the code we have provided.
    private VisibleImage frogImage;

    // add parameters to construct as needed.
    public Frog( ) {

    }

    public boolean overlaps(VisibleImage vehicleImage) {
        return false;    // YOU NEED TO CHANGE THIS!
    }

    public void kill ( ) {
    }

    public void reincarnate() {
    }

    public void hopToward(Location point) {
    }

    public boolean isAlive ( ) {
        return false;    // YOU NEED TO CHANGE THIS!
    }
}
```

Lane.java

```
public class Lane extends ActiveObject {

    // Distance from front bumper to back bumper of the
    // longest vehicle, in pixels.
    private static final int MAX_VEHICLE_SIZE = 139;

    // Add parameters to constructor as needed
    public Lane( ) {
        // Add code to construct the lane here.

        this.start();
    }

    public void run() {

        // Add code here if necessary

        // Loop until the program stops.
        while (true) {
        }
    }
}
```

Vehicle.java

```
public class Vehicle extends ActiveObject {

    // add parameters to the constructor as needed
    public Vehicle( ) {

        // insert code to construct the vehicle here

        this.start();
    }

    public void run() {

    }

}
```