**User Manual**

**FroggerProject**

**Ari Butowsky**

**Table of Contents**

Page #

1. **GENERAL INFORMATION…………………………………………………………….3-3**
   1. **System Overview…………………………………………………………………..3-3**
   2. **Organization of the Manual………………………………………………………3-3**
2. **SYSTEM SUMMARY…………………………………………………………………….4-4**
   1. **System Configuration……………………………………………………………..4-4**
   2. **User Access Levels………………………………………………………………...4-4**
   3. **Contingencies……………………………………………………………………...4-4**
3. **GETTING STARTED……………………………………………………………………5-17**
   1. **Installation of Necessary Software………………………………………………5-6**
   2. **Building the Project………………………………………………………………6-17**
4. **RUNNING THE PROJECT……………………………………………………………18-18**

**4.1 How to Run the Project………………………………………………………..........18-18**

**4.2 What Happens When the Project is Run………………………………………….18-18**

1. **GENERAL INFORMATION**

This section explains in general terms the system and the purpose for which it is intended.

**1.1 System Overview**

FroggerProject is a Java applet that provides a simulation of Frogger. It is created using Eclipse, which can operate on Windows, Mac OS X, or Linux with the right Java Development Kit, Android Development Kit (ADK), and Interactive Development Environment (IDE).

**1.2 Organization of the Manual**

The user’s manual consists of four sections: General Information, System Summary, Getting Started, and Running the Project.

General Information explains in general terms the system and the purpose for which it is intended.

System Summary provides a general overview of the system, outlining the uses of the system’s hardware and software requirements, system’s configuration, user access levels, and system’s behavior in case of any contingencies.

Getting Started explains how to install the software needed to create FroggerProject and how to build it.

Running the Game explains how to run FroggerProject and what happens when it is run.

1. **SYSTEM SUMMARY**

This section provides a general overview of the system, outlining the uses of the system’s hardware and software requirements, system’s configuration, user access levels, and system’s behavior in case of any contingencies.

**2.1 System Configuration**

Eclipse can operate on Windows, Mac OS X, and Linux. It requires no connection to the Internet in order to save data to the workspace nor is an Internet connection required to run Java applications or applets built using Eclipse.

After its installation, Eclipse can only be used with the right JDK, ADK, and IDE.

**2.2 User Access Levels**

Everyone can use the application.

**2.3 Contingencies**

FroggerProject can be saved in the Eclipse workspace by clicking on the Save or Save All button on the software’s menu located at the very top. If Eclipse shuts down prematurely, there is no way to restore the work that has been done between opening Eclipse and the point when it shut down. Basically, there are no contingencies in case Eclipse shuts down before data can be saved.

1. **GETTING STARTED**

This section explains how to install the software needed to create FroggerProject and how to build it.

**3.1 Installation of Necessary Software**

Before working on FroggerProject, you have to download the Android Software Development Kit (SDK), which works on Windows, Linux, and Mac OS X. You also must install Java and an IDE. Follow these steps to get everything you need:

**1.** Obtain a copy of Java since the program being written uses the Java language. Although JDK 5 or 6 is required in this case, but it is recommended that Java Development Kit (JDK) 8 be downloaded from the Oracle download website (<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>) since that is the best choice.

**2.** Download the Java runtime environment (JRE) 8 from <http://java.com/en/download/manual.jsp>.

**3.** Go to the Eclipse downloads page (<http://www.eclipse.org/downloads>) and select “Eclipse IDE for Java Developers”. Download the package into a temporary directory, unpack it, and move the whole unpacked directory to a permanent location (ex. C:\Eclipse on Windows or /Applications/Eclipse on Mac OS X.).

**4.** Go to <http://d.android.com/sdk> and select and download the SDK Starter Package, along with the SDK Components. Afterward, unpack it to a permanent directory.

**5.** Initiate the SDK Setup program. On Windows, run SDK Manager.exe, but on Linux and Mac OS X, run the tools/android program, select Available Packages, put a check mark next to every package, and click Install Selected.

**6.** The SDK Setup program now displays a list of Android SDK Components. Click on all the components that you want to install onto Eclipse and then click on “Install packages…”. This could take a long time depending on the amount of components you select.

**7.** Install the Android Development Toolkit (ADT) by following these steps:

**a)** Start Eclipse by running “eclipse.exe” on Windows or “eclipse” on Mac OS X or Linux.

**b)** Select the Help menu followed by Install New Software.

**c)** Click the “Add” button.

**d)** Enter the URL of the Android Development Tools update site: <https://dl-ssl.google.com/android/eclipse/>. If you have trouble with using “https”, try “http” instead.

**e)** If the address is correct, click OK to return to the Install dialog.

**f)** Select the checkbox next to Developer Tools and click Next.

**g)** Review the list of items to be installed, click Next again, accept the license agreements, and then click Finish to start the download and install process.

**h)** Restart Eclipse when the install is done.

**i)** Once Eclipse comes back up, select Window > Preferences > Android (Eclipse > Preferences on Mac OS X), enter the SDK install directory, and click OK.

**j)** Select Help > Check for Updates to make sure you have the latest version of Eclipse and the ADT.

**3.2 BUILDING THE PROJECT**

To build the project, follow these steps:

**1.** Select File > New > Java Project to open the New Java Project dialog box.

**2.** Next to Project name:, type *FroggerProject* and click Finish.

**3.** Double-click on *FroggerProject* to view all of its sections and right-click on src.

**4.** Select New > Package and in the dialog box that comes up, type *froggerPackage* next to where it says “Name:”. Click on Finish.

**5.** Right-click on *froggerPackage* and select New > Class. Next to where it says Name:, type *Frog* and click Finish. Follow this step to create the *Frogger, Lane, and Vehicle* classes.

**6.** Create an HTML file known as *Frogger.html* with the following code:

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=macintosh">

<title>

HTML Test Page

</title>

</head>

<body>

MagnetGame will appear below in a Java enabled browser.<br>

<applet

codebase = "."

code = "Frogger.class"

name = "TestApplet"

width = "400"

height = "400"

hspace = "0"

vspace = "0"

align = "middle"

>

</applet>

</body>

</html>

Once you are done writing this code, save it under a directory of your choosing.

**7.** Right-click on *froggerPackage* and select Import.

**8.** In the Import dialog box, select General > File System. Then, click on Browse to find and click on the directory under which *Frogger.html* is saved. On the right dialog box, click on *Frogger.html* and click Finish.

**9.** Search the Internet for various pictures and save them as .gif files. Give each of them one of the following names: froggy, froghops, jeep\_left, jeepRight, oldcar\_left, oldcar\_right, taxi\_left, taxi\_right, van\_left, and van\_right. One of the images you find must be given the name Thumbs and be saved as a .db file.

**10.** Copy and paste the images you found into the bin section of *FroggerProject*.

**11.** Go to [www.eventfuljava.cs.williams.edu/library.html](http://www.eventfuljava.cs.williams.edu/library.html) and download objectdrawV1.1.2.jar.

**12.** Right-click on FroggerProject and select Properties > Java Build Path > Libraries.

**13.** Click on Add External JARs and search for and select objectdrawV1.1.2.jar. Click OK.

**14.** Fill out the code for the classes in the following ways:

**FROG**

**package** froggerPackage;

**import** objectdraw.\*;

**import** java.awt.\*;

// The frog that is trying to get across the highway. It moves in the direction of

// mouse clicks. It is killed when a vehicle hits it and is reincarnated when the user

// restarts.

**public** **class** Frog

{

// The location where the frog starts when trying to cross the highway. We remember

// this so we can reincarnate the frog.

**double** startX;

**double** startY;

// What the frog looks like

**private** VisibleImage frogImage;

// How far the frog hops

**private** **int** hopDistance;

// Message the frog displays when it is hit by a car

**private** Text ouch;

// Remembers if the frog is alive or not

**private** **boolean** alive;

// Known from examining the gif

**private** **static** **final** **int** ***FROG\_HEIGHT*** = 48;

// Draws a frog

// Parameters:

// frogPic - what the frog looks like

// highwayCenterX - where the frog should start

// highwayBottom - the lowest part of the highway

// laneWidth - how far across a lane, the frog can hop one lane at a time

// canvas - canvas to draw on

**public** Frog(Image frogPic, **int** highwayCenterX, **int** highwayBottom, **int** laneWidth, DrawingCanvas canvas)

{

// Remember the starting location for reincarnation

**double** frogSpacing = (laneWidth - ***FROG\_HEIGHT***) / 2;

startX = highwayCenterX + frogSpacing;

startY = highwayBottom + frogSpacing;

// Display the frog in the right place.

frogImage = **new** VisibleImage (frogPic, highwayCenterX, highwayBottom, canvas);

// Remember how far the frog can hop.

hopDistance = laneWidth;

// Create the ouch text to display when the frog is killed.

ouch = **new** Text ("OUCH!", startX, startY + 30, canvas);

ouch.setFontSize (20);

ouch.setColor (Color.***red***);

ouch.hide();

// The frog starts out alive.

alive = **true**;

}

// Returns the image of the frog

**public** **boolean** overlaps(VisibleImage anImage) {

**if** (frogImage.overlaps(anImage))

**return** **true**;

**else**

**return** **false**;

}

// Displays the ouch text and causes the frog to not move until after it is reincarnated.

//

// Precondition: The frog is alive

**public** **void** kill ()

{

ouch.show();

alive=**false**;

}

// Brings the frog back to life. Removes the ouch text and moves the frog back to the

// start

//

// Precondition: the frog is not alive

**public** **void** reincarnate()

{

ouch.hide();

alive=**true**;

frogImage.moveTo (startX, startY);

}

// Causes the frog to move one hop towards the point where the user clicked. Left/right

// moves are given preference over forward/back moves.

//

// Precondition: The frog is alive.

**public** **void** hopToward(Location point)

{

**if** (point.getX() < frogImage.getX())

{

frogImage.move (-hopDistance, 0);

}

**else** **if** (point.getX() > frogImage.getX() + frogImage.getWidth())

{

frogImage.move (hopDistance, 0);

}

**else** **if** (point.getX() < frogImage.getY())

{

frogImage.move (0, -hopDistance);

}

**else** **if** (point.getX() > frogImage.getY() + frogImage.getHeight())

{

frogImage.move (0, hopDistance);

}

// Else clicked on the image. Don't move.

}

// Returns true if the frog is alive.

**public** **boolean** isAlive()

{

**if**(alive)

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

}

**FROGGER**

**package** froggerPackage;

**import** objectdraw.\*;

**import** java.awt.\*;

**public** **class** Frogger **extends** WindowController{

// Constants defining the sizes of the background components.

**private** **static** **final** **int** ***HIGHWAY\_LENGTH*** = 700;

**private** **static** **final** **int** ***LANE\_WIDTH*** = 100;

**private** **static** **final** **int** ***NUM\_LANES*** = 4;

**private** **static** **final** **int** ***HIGHWAY\_WIDTH*** = ***LANE\_WIDTH*** \* ***NUM\_LANES***;

**private** **static** **final** **int** ***LINE\_WIDTH*** = ***LANE\_WIDTH*** / 10;

// Constants defining the locations of the background components

**private** **static** **final** **int** ***HIGHWAY\_LEFT*** = 50;

**private** **static** **final** **int** ***HIGHWAY\_RIGHT*** = ***HIGHWAY\_LEFT*** + ***HIGHWAY\_LENGTH***;

**private** **static** **final** **int** ***HIGHWAY\_TOP*** = 100;

**private** **static** **final** **int** ***HIGHWAY\_BOTTOM*** = ***HIGHWAY\_TOP*** + ***HIGHWAY\_WIDTH***;

**private** **static** **final** **int** ***LINE\_SPACING*** = ***LINE\_WIDTH*** / 2;

// The frog that tries to jump across the road.

**private** Frog theFrog,secondFrog;

// Remembers if the mouse is in the window. Ignore mouse releases outside the window.

**private** **boolean** inWindow = **true**;

**public** **void** begin()

{

// Draw the background

FilledRect highway = **new** FilledRect (***HIGHWAY\_LEFT***, ***HIGHWAY\_TOP***, ***HIGHWAY\_LENGTH***, ***HIGHWAY\_WIDTH***, canvas);

**new** FilledRect(0,0,***HIGHWAY\_LEFT***,canvas.getHeight(), canvas).setColor(Color.***white***);

**new** FilledRect(***HIGHWAY\_LEFT***+***HIGHWAY\_LENGTH***, 0, canvas.getWidth() - ***HIGHWAY\_LEFT***+***HIGHWAY\_LENGTH***,

canvas.getHeight(),canvas).setColor(Color.***white***);

// Draw the lane dividers

**int** whichLine = 1;

**while** (whichLine < ***NUM\_LANES***) {

**if** (whichLine == ***NUM\_LANES*** / 2) {

// The middle line is a no passing line

drawNoPassingLine (***HIGHWAY\_TOP*** + (whichLine \* ***LANE\_WIDTH***) - (***LINE\_SPACING*** / 2 + ***LINE\_WIDTH***));

}

**else** {

drawPassingLine (***HIGHWAY\_TOP*** + (whichLine \* ***LANE\_WIDTH***) - (***LINE\_WIDTH*** / 2));

}

whichLine = whichLine + 1;

}

// Add the frog

Image frogImage = getImage("FroggerProject/froggy.gif");

Frog theFrog = **new** Frog (frogImage, 0, ***HIGHWAY\_BOTTOM***, ***LANE\_WIDTH***, canvas);

Image carImage = getImage("FroggerProject/jeepRight.gif");

Vehicle car = **new** Vehicle(carImage, .10, ***HIGHWAY\_TOP*** + 70, ***HIGHWAY\_LEFT***, ***HIGHWAY\_RIGHT***, ***HIGHWAY\_WIDTH***, theFrog, canvas);

// Get the cars moving

**int** lane = 1;

// There are 4 types of vehicles. Pick the type for a lane randomly.

RandomIntGenerator vehicleGenerator = **new** RandomIntGenerator (1, 4);

**int** nextVehicleNum; // Next type of vehicle

Image nextVehicle; // Image corresponding to the next type

String direction; // Direction the vehicle will travel

**boolean** goingRight; // True if the vehicle is driving from left to right

// Create the lanes

**while** (lane <= ***NUM\_LANES***) {

// The bottom half lanes go to the right

**if** (lane > ***NUM\_LANES*** / 2) {

direction = "right";

goingRight = **true**;

}

**else** {

direction = "left";

goingRight = **false**;

}

// Figure out the next vehicle type and load the appropriate image for that type and the direction

// it should travel.

nextVehicleNum = vehicleGenerator.nextValue ();

System.***out***.println ( "here" + nextVehicleNum);

**if** (nextVehicleNum == 1) {

nextVehicle = getImage ("jeep\_" + direction + ".gif");

}

**else** **if** (nextVehicleNum == 2) {

nextVehicle = getImage ("oldcar\_" + direction + ".gif");

}

**else** **if** (nextVehicleNum == 3) {

nextVehicle = getImage ("taxi\_" + direction + ".gif");

}

**else** {

nextVehicle = getImage ("van\_" + direction + ".gif");

}

// Create the lane telling it where it is, which direction its cars should go, what its cars should

// look like, and what the frog is.

**new** Lane (***HIGHWAY\_TOP*** + (lane - 1) \* ***LANE\_WIDTH***, ***HIGHWAY\_LEFT***, ***HIGHWAY\_RIGHT***, ***LANE\_WIDTH***,

goingRight, nextVehicle, theFrog, canvas, 30.0);

lane = lane + 1;

}

}

// Draws a pair of solid yellow lines to represent a no passing divider between lanes

// Parameter: y - the top of the top line

**private** **void** drawNoPassingLine (**int** y) {

// Draw the solid dividing lines

FilledRect topLine = **new** FilledRect (***HIGHWAY\_LEFT***, y, ***HIGHWAY\_LENGTH***, ***LINE\_WIDTH***, canvas);

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

// Parameters: y - the top of the line.

**private** **void** drawPassingLine (**int** y) {

**int** x = ***HIGHWAY\_LEFT***;

**final** **int** DASH\_LENGTH = ***LANE\_WIDTH*** / 3;

**final** **int** DASH\_SPACING = DASH\_LENGTH / 2;

FilledRect dash;

**while** (x < ***HIGHWAY\_RIGHT***) {

// Draw the next dash.

dash = **new** FilledRect (x, y, DASH\_LENGTH, ***LINE\_WIDTH***, canvas);

dash.setColor (Color.***white***);

x = x + DASH\_LENGTH + DASH\_SPACING;

}

}

// If the mouse is alive, tell the mouse which direction to move. If the mouse is not alive, ignore

// mouse releases except for ones in the starting area. Ignore all mouse releases outside the window.

**public** **void** onMousePress(Location point)

{

**if** (inWindow) {

**if** (theFrog.isAlive ())

{

point.getY();

}

**else** **if** (point.getY() > ***HIGHWAY\_BOTTOM***)

{

theFrog.reincarnate();

}

}

}

// Remembers if the user enters the window so we can start paying attention to mouse presses again.

**public** **void** onMouseEnter(Location point){

inWindow = **true**;

}

// Remembers if the user leaves the window so we can ignore mouse presses.

**public** **void** onMouseExit(Location point){

inWindow = **false**;

}

}

**LANE**

**package** froggerPackage;

**import** objectdraw.\*;

**import** java.awt.\*;

// The lane on which cars drive. This class is responsible for generating new cars,

// keeping a gap between the cars.

**public** **class** Lane **extends** ActiveObject

{

// Location and size of the lane.

**private** **int** top;

**private** **int** left;

**private** **int** right;

**private** **int** width; // This is the width of the lane as viewed by a moving car.

// The image for the vehicles that should drive on a lane.

**private** Image vehicleImage;

// The canvas that the vehicles should be drawn on.

**private** DrawingCanvas canvas;

// The frog that is trying to cross the highway.

**private** Frog theFrog;

// Car speed in pixels/millisecond

**private** **double** speed;

// Construct a new lane at the given location and size.

// Parameters:

// top - the top edge of the lane on the canvas

// left - the left edge of the lane

// right - the right edge of the lane

// width - the distance between the top and bottom of the lane

// goingRight - true means cars should drive from left to right

// vehicleImage - the picture of the vehicles that should drive in the lane

// frog - the frog trying to cross the highway

// canvas - the canvas to draw on

**public** Lane(**int** top, **int** left, **int** right, **int** width, **boolean** goingRight,

Image vehicleImage, Frog frog, DrawingCanvas canvas, **double** speed)

{

// Figure out how fast the cars should go. Use a negative speed if the car should

// go from right to left.

RandomIntGenerator speedGenerator = **new** RandomIntGenerator (1, 4);

speed = speedGenerator.nextValue() / 50.0;

**if** (!goingRight)

{

speed = -speed;

}

// Remember the other information so that vehicles can be generated correctly.

**this**.top = top;

**this**.left = left;

**this**.right = right;

**this**.width = width;

**this**.speed = speed;

**this**.canvas = canvas;

**this**.vehicleImage = vehicleImage;

**this**.theFrog = frog;

// Start generating vehicles

start();

}

// Continuously generate vehicles until the applet stops.

**public** **void** run()

{

// The length of the longest vehicle. This was determined by examining the gif.

**int** maxVehicleLength = 139;

// Randomly generate the gap that this lane uses

RandomIntGenerator gapGenerator = **new** RandomIntGenerator (maxVehicleLength, right - left - 2\*maxVehicleLength);

**int** nextGap = gapGenerator.nextValue();

// Delay between checking if we should generate a new car

**int** delay = 50;

// The next car

Vehicle nextVehicle;

// Generate the cars until the applet stops

**while** ( **true** )

{

nextVehicle = **new** Vehicle (vehicleImage, speed, top, left, right, width,

theFrog, canvas);

// Only generate a car when we have traveled at least the desired gap distance

**while** (tooClose (nextVehicle, nextGap)) {

*pause* (delay);

}

}

}

// Return true if the last vehicle is the desired distance away from the start

// Parameters:

// lastVehicle - the last vehicle created

// desiredGap - the distance we want to keep from the last vehicle

**public** **boolean** tooClose (Vehicle lastVehicle, **int** desiredGap) {

// We are not too close if the last vehicle has already reached the end.

**if** (lastVehicle.atLaneEnd ()) {

**return** **false**;

}

// See how far the vehicle has gone. If it has driven more than the gap we want,

// we can add another car.

**else** {

**return** lastVehicle.distanceTraveled () < desiredGap + lastVehicle.getLength();

}

}// end method tooClose

}

**VEHICLE**

**package** froggerPackage;

**import** objectdraw.\*;

**import** java.awt.\*;

// Vehicle defines the actions of a car driving on the highway. Cars stay in a single lane and

// drive at a constant speed from one end to the other. Along the way, they may kill the frog

// by running into it.

**public** **class** Vehicle **extends** ActiveObject

{

// The image of the frog on the screen.

**private** VisibleImage vehicleImage;

// The speed that the vehicle goes in pixels/millisecond. A positive speed results in the

// car going from left to right. A negative speed results in the car going from right to left.

**private** **double** speed;

// The distance that the car has moved since it was created. This is negative for cars moving

// from right to left.

**private** **double** totalDistance = 0;

// The location of the far end of the lane that the car is driving on.

**private** **int** laneEnd;

// The frog that the car might hit

**private** Frog theFrog;

// Construct a new car and start it moving across the screen.

// Parameters:

// vehiclePic - the depicting of the vehicle to display

// speed - the speed of the vehicle in pixels/millisecond

// laneTop - the top of the lane that the vehicle should drive in

// laneLeft - the left edge of the lane

// laneRight - the right edge of the lane

// laneWidth - the width of the lane that the car drives in. This is interpreted in the

// sense of the real world width. Since lanes go across the screen, the width

// is the up-down distance

// frog - the frog that the car might hit

// canvas - the canvas to draw on

**public** Vehicle(Image vehiclePic, **double** speed, **int** laneTop, **int** laneLeft, **int** laneRight,

**int** laneWidth, Frog frog, DrawingCanvas canvas )

{

// Remember the frog and speed for later.

**this**.theFrog = frog;

**this**.speed = speed;

// Figure out which end is the starting and finishing end. Put the car on the starting end.

**if** (speed > 0) {

**this**.laneEnd = laneRight;

vehicleImage = **new** VisibleImage (vehiclePic, laneLeft, laneTop,

canvas);

vehicleImage.sendToBack();

vehicleImage.sendForward();

vehicleImage.move (-vehicleImage.getWidth(), 0);

}

**else** {

**this**.laneEnd = laneLeft;

vehicleImage = **new** VisibleImage (vehiclePic, laneRight, laneTop, canvas);

vehicleImage.sendToBack();

vehicleImage.sendForward();

// Move the vehicle left so it starts entirely on the lane

}

// Center the vehicle in the lane

vehicleImage.move (0, (laneWidth - vehicleImage.getHeight()) / 2);

// Start the car moving.

start();

}

// Move the car across the screen in a straight line until it reaches the end. Kill the frog

// if you hit it.

@SuppressWarnings("deprecation")

**public** **void** run()

{

// Delay between car movements in milliseconds.

**int** delay = 30;

// The last time the car was moved.

**double** lastTime = getTime();

// The distance we should move the car this time to get smooth motion at the desired speeed.

**double** distance;

// Loop until the active object is told to stop or the car reaches the end of the lane.

**while** (!atLaneEnd ()) {

// Compute how far the car should move for the amount of time it was paused and the

// speed it should appear to move at.

distance = (getTime() - lastTime) \* speed;

// Move the car

vehicleImage.move (vehicleImage.getWidth(), 0);

totalDistance = totalDistance + distance; ;

// See if it killed the frog

//if (vehicleImage.overlaps(theFrog) && frog.overlaps (vehicleImage))

//{

// theFrog.kill();

//}

// Remember the time for the next iteration and wait.

lastTime = getTime ();

*pause* (delay);

}

// The vehicle should disappear when it reaches the end or the applet is stopped.

vehicleImage.hide();

}

// Returns true if the vehicle has reached the end of the lane that it is driving towards.

**public** **boolean** atLaneEnd () {

**if** (speed > 0) {

**return** getLeft() > laneEnd;

}

**else** {

**return** getRight() < laneEnd;

}

}

// Returns the location of the left edge of the vehicle

**public** **double** getLeft ()

{

**return** vehicleImage.getX() + getLength();

}

// Returns the location of the right edge of the vehicle

**public** **double** getRight () {

**return** vehicleImage.getX () + getLength();

}

// Returns the length of the vehicle from the front bumper of the car to the rear bumper.

**public** **double** getLength () {

**return** vehicleImage.getX();

}

// Return the total distance the vehicle has moved since it was created.

**public** **double** distanceTraveled () {

**if** (speed > 0) {

**return** totalDistance;

}

**else** {

**return** -totalDistance;

}

}

}

1. **RUNNING THE PROJECT**

This section explains how to run FroggerProject and what happens when it is run.

**4.1 How to Run the Project**

To run *FroggerProject*, right-click on it and select Run As > Java Applet.

**4.2 What Happens When the Project is Run**

When *FroggerProject* runs, a separate window will open in which a simulated Frogger game will ensue.