CS4731 Project: Gesture Recognition

Hart Lambur
Department of Computer Science
Columbia University
hal2001@columbia.edu

Blake Shaw
Department of Computer Science
Columbia University
bs2018@columbia.edu

December 21, 2004

1 Introduction

1.1 The "Idea"

The user interface (UI) of the personal computer has evolved from a text-based command line to a graphical interface with keyboard and mouse inputs. Undoubtedly, human-computer interaction will continue to evolve towards more natural forms of input, like human movement recognition. Popular culture has long viewed computer vision and visual gesture recognition as one of the "future" ways we will interact with our computers; indeed, many high-end 3D environments already use some form of human movement recognition in their user interfaces. In our project, we explore the feasibility of implementing a simple system to understand a basic set of hand gestures.

Our main interest is exploring the computer vision techniques involved in real-time tracking of visual input and the analysis of that input to recognize "gestures". In the most general sense, recognizing gestures seems to be an extremely complex task. A sophisticated gesture recognition system would have to incorporate many complex algorithms spanning many fields of computer science as well as incorporate an abundance of knowledge about the nature of human movement. Building a robust system which could recognize a large set of motions is a very compelling problem, but implementing this kind of system is currently not feasible for a semester long project. However, by defining a limited vocabulary of gestures, we can create a compact system which can effectively recognizes this set of simple gestures.

In our system, the user controls a well-defined visual input (in our case, a bright thimble of LEDs worn on the index finger) to perform certain basic computer tasks. Simple left/right/up/down gestures are recognized under variable light conditions with little or no operator training; using this simple four gesture vocabulary, basic computer functions can be controlled. Below, we give a brief outline of some of our initial project goals before discussing the details of our user interface and implementation.

1.2 Initial Project Goals

Realizing the broad reach of our gesture recognition project across many disciplines and fields of research, we carefully documented our project goals during the planning phase. Below we outline what we attempted to accomplish, and what we has been successfully implemented.

i. *Tracking user input:* Initially, we needed to track the real-time position of the user input. Using a self-constructed LED finger-light, we were able to track the position of the user's finger under varying light conditions. By constraining the environment to search for a single, neon-green-colored LED object in an otherwise average-intensity scene, we have been able to achieve accurate results using simple techniques.

ii. *Gestures in a direction:* To recognize our basic four gesture vocabulary, we implemented an algorithm to recognize user movements in the left/right/up/down directions. This task required us to take into account the speed at which the gesture is made in order to differentiate between deliberate actions and general random movement. By keeping track of the last N points of the user's path,

we analyze this point list for strong movements in a direction by averaging the local distance between points. We compute these averages in both the x and y directions separately and, using a threshold corresponding to gesture-speed, we determine the general direction of a swift movement.

iii. *"Holding" gestures:* As an extension to our basic gesture vocabulary, we extended our system to recognize "holding gestures"—the visual equivalent to holding down a mouse button. This problem is more complicated than simply recognizing the user swiping in a direction. It involves recognizing a fast movement in a direction and gauging if the user remains a certain distance away from some sort of relative center which, when the user returns to, constitutes "releasing the button." Our system defines the relative center on a gesture-by-gesture basis, considering the center to be the initial starting location of the movement. Establishing a robust method for tracking this relative center and holding motion was one of the more challenging aspects of the project.

iv. *Recognizing other motions:* There are innumerable extensions to our gesture recognition system that would enhance our gesture vocabulary. Ideas include matching gestures to actions using shapes (like drawing a circle in the air), or recognizing sequences/patterns of simple up/down/left/right motions. This was one aspect of the project which we were unable to spend much time developing.

v. *Integrating system with computer:* The final step of our project connected the gesture recognition system to basic functions of the computer. We developed a backend to execute simple shell and apple scripts to control a variety of computer applications using decisions from our gesture recognition system. Our scripting framework allows for extreme flexibility in developing new applications for the system. Currently, we can control the functions of an MP3 player (play/pause/volume/track selection), and surf web pages (using back/forward/link selections commands) with our basic gesture vocabulary. Future applications include basic video game control for Mario Brothers (or an equivalent game), and scrolling through text applications.

Figure 1: The Apple iSight webcam used in our vision system. For gesture recognition, the iSight is particularly well-suited since it sits on top of the user's monitor, at eye level. By placing the camera directly in front of the user, motions can be directed at the computer screen rather than at some off-axis camera, allowing for a more intuitive interface.

2 User Interface

Developing an interface through which one can control a computer with simple hand gestures is not an easy task. People have grown accustomed to using well defined input devices to interact with computers—devices such as the keyboard and mouse easily capture user input in a well understood fashion. Designing a computer system to recognize hand movements presents difficult problems, both in developing the algorithms to handle such a task, and in developing an intuitive interface through which users can interact naturally with the system.

We designed our interface around two basic hardware components: the Apple iSight webcam, and a light glove/wand. After a quick overview of this hardware, we comment on the general setup of our user interface, and some of the problems we have encountered while designing an intelligent, intuitive system to recognize hand gestures.

2.1 Computer Hardware

As the eyes of our vision system, we use the Apple iSight digital webcam to capture a real-time video feed (see figure **??**). The iSight is a beautifully designed webcam ideally suited to our gesture recognition system. The camera's fast auto-focus and auto-exposure allow sharp im-

Figure 2: The four LED thimble attached to a user's index finger. The four LEDs are splayed to distribute the light intensity over a range of angles. This allows a bright light to be seen by the camera no matter what orientation the user prefers to keep their finger.

ages to be captured at a high frame rates, a feature that has provided indispensable when attempting to track fast gesture motions of the user. Furthermore, the camera is specifically designed to sit on top of the computer monitor, positioning the lens at eye level directly in front of the user, and not off-center on a desk like many other webcams. By positioning the camera directly in front of the user, making gestures is more natural, and the user is given the impression that he or she is interacting directly with the computer monitor and not with some distant camera.

In an effort to improve the robustness of the system, we designed a light glove/wand which the user can use to interact with the gesture recognition system. We soldered four bright green LEDs together and attached them to a 9V battery pack. Figure **??** shows the LED finger-thimble attached to the user's index finger. Holding the battery pack in the palm of his or her hand, the user can then "point" with their index finger and have the vision system track the bright green LEDs for gesture recognition.

2.2 Defining a Relative Center

In an effort to keep our gesture recognition system as natural to use as possible, we wanted to ensure that the user's motions could be recognized at any position within the camera's field of view, and not just in the center of the video frame. To do this, we developed a concept of a *relative center*, to allow the user to make gestures starting from any place inside the frame of the video feed.

In our system, the relative center is defined as the position from which the user's hand must move a certain amount within a given time frame for the hand motion to be recognized as a gesture. By changing the time and speed that the user's hand must move away from this "center", the sensitivity of the system to user movements (that are not intended to be gestures) can be adjusted.

To provide the user with some visual feedback for their motions, the relative center position of the user is shown in real-time overlaid on top of the video feed. The current center position is drawn on top of the image with a standard cross-hair target; furthermore, the outer circle of the cross-hair defines how far the user must move from the center position (within a certain time frame) to trigger an action. It is also important to note that the video image is mirrored when displayed to the user, creating a more natural environment for the user to observe his or her interaction with the vision system.

2.3 Gesture Recognition

For our basic system, we implemented a simple vocabulary of four gestures: left, right, up and down. Given our relative center, this left/right/up/down motions are recognized when the user moves outsider the outer circle of the center cross-hair within a given time. Details of the algorithmic solution to this problem are discussed below.

The system is calibrated so that slow, wandering hand movements are not recognized as gestures. These types of movements are slow enough to not move outside the specified motion distance within the specified time. By allowing the user to move his or her hand freely up to a reasonable threshold, the system is robust to false positives, motions that were not intended to be recognized as gestures.

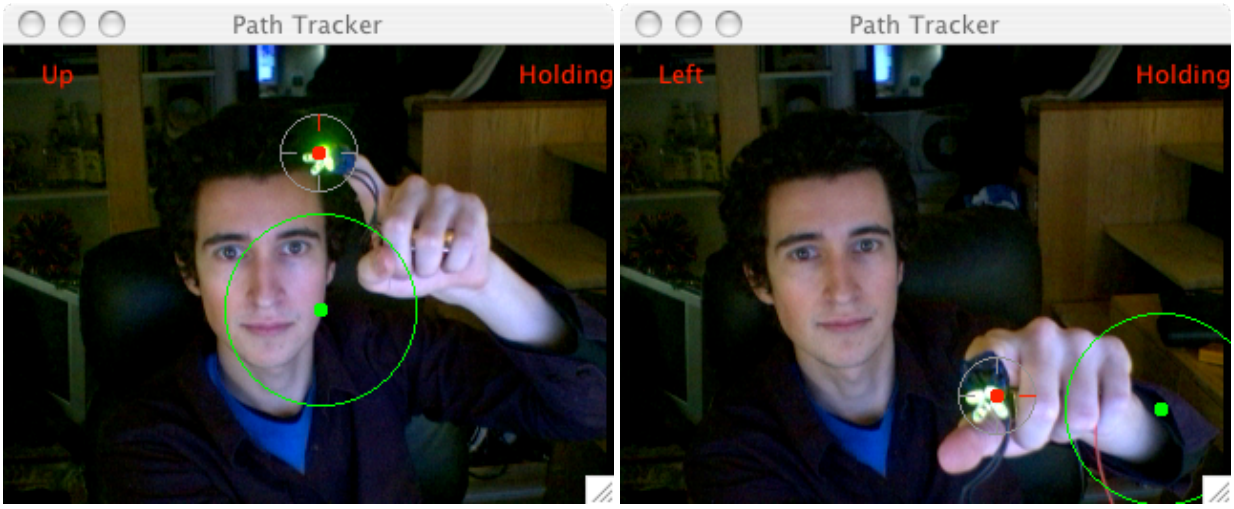

Figure 3: Two screenshots of the system tracking "holding" gestures. Blake's index finger is tracked using the cross-hair as seen above. When a quick motion is made, the system remembers the relative center from which the motion originated, and freezes that position with the green holding circle. As long as Blake keeps his finger outside the green holding radius, the action is "held" and the cross-hair displays the direction of the action in red. When Blake brings his finger back inside the green circle, the holding gesture is turned off.

2.4 "Holding" Gestures

As a final element to our user interface, we extended our basic four gesture vocabulary to recognize "holding" gestures. In essence, we designed our UI to differentiate between quick movements outside the relative center and back, and slow movements that are held away from the relative center. We gauge the initial fast movement away from the center in the same way as before—gestures outside a specified distance within a give time frame are recognized as actions—but the system checks to see if the movement is held away from the center for a given time interval following the quick action.

The UI provides the the user with some visual feedback if they are holding a gesture. If the user quickly moves outside the outer circle of the cross-hair triggering a fast motion, but maintains his position there for longer than a half second or so, the cross-hair turns red and "HOLDING" is displayed in the top right corner of the display (see figure **??**). The relative center cross-hair from before the user's initial quick action is "frozen" on the display, and the holding action will not be released until the user moves back within the holding radius of the frozen center. With this visual information, holding gestures become a natural extension to the standard left/right/up/down motions, and a new user is quickly able to learn how to use "holding" gestures.

3 Implementation

The implementation for our gesture-recognition system is divided into six main areas: building the light source to track, capturing video, tracking the path of the light source, recognizing gestures from path analysis, displaying the graphical interface, and executing scripts upon completion of gestures. Figure **??** shows an outline of our code structure.

3.1 Building the Light Source

There are many considerations for picking an object to track, because we want to find an easy means by which we can distinguish the object from the background under a variety of lighting conditions. Initially we simply tracked a green highlighter, using its distinct color to iden-

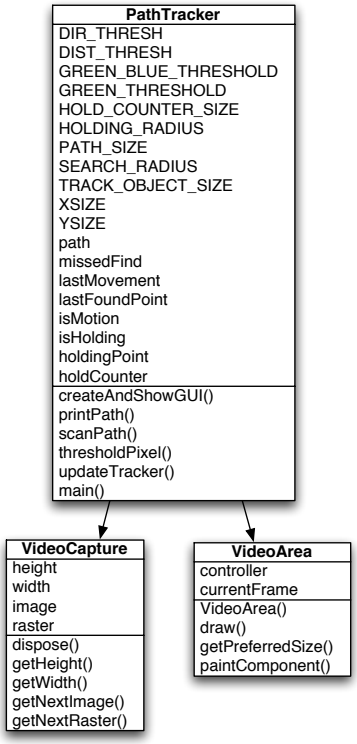

Figure 4: A class diagram for project's Java implementation. The main PathTracker class calls VideoCapture to grab images from the iSight camera and uses VideoArea to draw the processed images back on the display using Java's Swing libraries.

tify it. There are not many every day objects that are bright green, so this was an adequate choice. However, in order to make our system more robust to different backgrounds and lighting conditions, we decided to use four green LEDs attached to the user's finger (see figure **??**). The green LEDs are easy to identify because their red and green intensities are registered by the camera as almost maxed out, allowing us to pick a high threshold. Furthermore, the blue channel is approximately 20% less bright then the red and green channels. We use this fact to distinguish the green LEDs from other bright lights. We found the LED system seems to be quite robust, working in a variety of visual environments including those containing green objects.

Figure 5: We constructed an LED finger thimble out of four green LED lights. The LEDs are solder together in a serial circuit and connected to a 9V battery pack.

3.2 Capturing Video

We decided to use the Apple iSight webcam as the hardware for our video capture system. For software, we use a Java Class called VideoCapture which simplifies the Quicktime library routines for grabbing each frame of video as it is read from the iSight. We obtained the basic framework for the VideoCapture class from the internet and then modified it for our own purposes (see references). From the VideoCapture class, we obtain each frame as a Raster object which we then wrap into a BufferedImage. This setup allows for easy access to pixels of each frame.

3.3 Tracking the Path of the Light Source

For each frame we scan through every pixel and determine if the pixel is part of the light source or not. We implement this by checking if the red and green components of the pixel's color are above a specific threshold. Also we make sure that the difference between the green channel and the blue channel is above a certain threshold; this blue-green test allows us to pick out the light source from other non-green bright light sources in the scene. We sum up the x and y positions for each pixel found to be part of

the light source and keep a count of how many of these pixels have been found. If this count is above a certain threshold, we consider the light-source to be in the scene, and we calculate its position using the average of the x and y components for each pixel found to be part of the light source. For each frame we then register the position of the light source in a global path list, which keeps track of the last N positions of the light source found in the last N frames.

3.3.1 Tracking Optimization

We recognized that scanning every pixel of every frame was not necessarily needed to effectively find the light source in the scene. By assuming a certain amount of locality of the user's movement, we can simply search a small window around the user's last know position. However, if the light source is not found within this small window, a flag is marked, and on the next pass, the entire frame is scanned in order to ensure that the light source is no where in the frame. Figure **??** shows an example of tracking the light source with the localize search window.

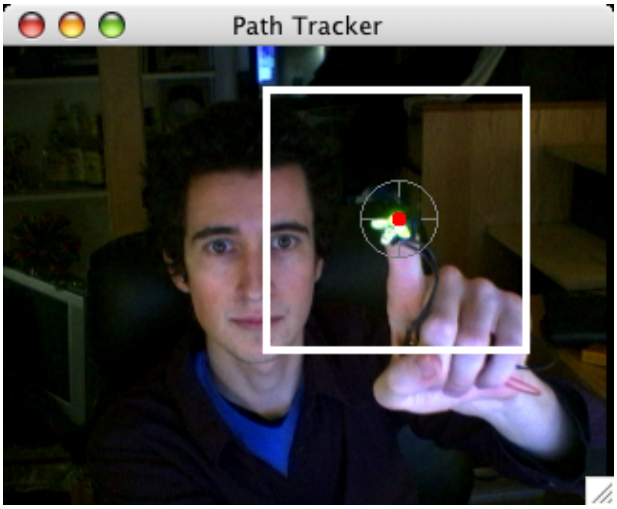

Figure 6: Here we see the search window used for our tracking optimization overlaid on the video display. If the light source is not found by searching this smaller area, the entire video frame is searched in the next iteration.

3.4 Path Analysis for Gesture Recognition

We analyze this global path list of positions of the light source over the last N frames to determine if a gesture has been made. The algorithm first computes the average distance between all points on the list, if this average distance is greater then a certain threshold, there must have been a swift movement, and therefore we recognize that a gesture has taken place (see figure **??**). The algorithm then considers the sum of the differences between the points in both the x and y components. If the sum of the differences of the x components of all points on the path is greater then sum of the differences of the y components, we can determine the motion to be in the x direction, and vice versa. By looking at the sign of the sum of the differences we can distinguish up from down, and left from right movements.

We do further analysis with regard to the gesture equivalent of holding a button versus simply pressing a button. When the initial movement is made, the last point on the global path list is marked as a special return point. After the initial movement, the user has a certain number of frames to return the light-source to within a certain holding radius of this return point, in order to register pressing the button. Otherwise, if the user remains outside a certain radius of the return point, the system recognizes this as a "holding gesture" until the user finally returns to within the holding radius of the return point.

3.5 Displaying the Graphical Interface

We chose to use the Java Swing library for displaying the graphics and interface for our system. For each frame we draw the BufferedImage coming from the video feed to the screen, and then overlay a variety of visual aids. We draw a cross hair around the current position of the light source (see figure **??**). When a motion is made in a specific direction, that portion of the cross-hair is colored red; when the user executes a holding gesture, the "HOLDING" is printed on the screen. Furthermore, when a decision is made about the direction of the gesture, that decision is printed on the screen (left, right, up, down).

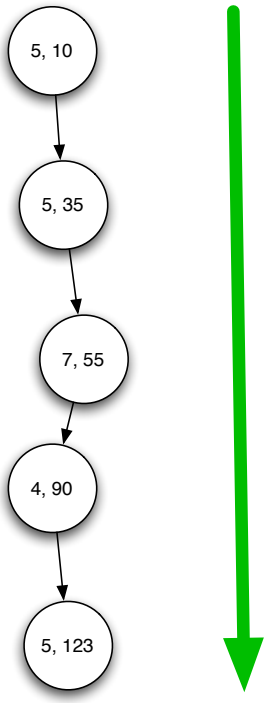

Figure 7: The above diagram shows the path analysis for *path-size* = 5. In this example, distance_counter registers as 113.243 based on the sum of the distances between these 5 points. Since distance_counter > DIST_THRESH, an action is triggered in the specified direction (here the gesture is down).

3.6 Executing Scripts upon Completion of Gestures

Upon recognizing that a gesture has taken place, the system uses a Java Process object combined with the Mac OS X command line utility "osascript" to execute Applescripts on the host machine. These scripts are defined in external files, and can be easily modified for a variety of applications. Furthermore these scripts can simulate basic keyboard commands applicable to all applications running.

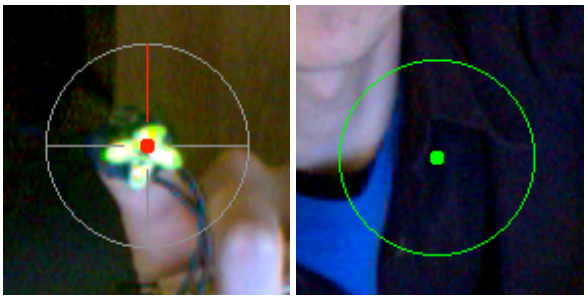

Figure 8: Here we see images of the common elements used in our UI design. The cross-hair on the left shows the user's current position, while the green return circle on the right is used to mark the relative center when holding a gesture.

4 Conclusion and Discussion

Producing a gesture recognition system robust to different users under different lighting conditions is not an easy task. Real-time processing in computer vision is hard! Developing a responsive, usable system took considerable effort. However, we are very satisfied with our results. By imposing intelligent and meaningful limits on the visual environment, and by developing an intuitive user interface, we were able to create a simple yet robust gesture recognition system. Below, we outline our project successes and failures and discuss the current state-of-the-art in gesture recognition.

4.1 Project Successes and Failures

Simple algorithms and techniques proved to work best when trying to build a usable vision system that works in real-time. Since real-time video processing demands considerable CPU power, we needed to impose intelligent limits on the visual environment that would simplify our algorithmic complexity. We established two such limits: we required the user to wear an LED thimble, and we limited our gesture set to movements in the four cardinal directions.

The initial problem of tracking a visual input, like a light wand, is easily done in a setting without time constraints, but this problem quickly becomes computationally intractable at 30 frames per second. To reduce the CPU power needed to find an object in the video stream,

we require the user to wear the green LED thimble, giving us a unique, bright light source to search for. The unusual neon green color allows us to find and track the LEDs in a single pass through the image's pixels using nothing more than a simple threshold search technique.

Attempting to deduce inputs from a user wildly moving his or her finger all over the screen is impossible, even with the most complicated and robust of gesture recognition systems. But detecting gestures becomes tractable when the vocabulary is limited in some meaningful way. We recognize gestures in the four cardinal directions. Our simple path tracking algorithm deduces movements by analyzing a list of past positions and determining if the user has moved a certain threshold in a certain direction within a given time limit. The simplicity of the system allows for robust recognition of simple gestures for different users and different style of motion—with little or no effort a new operator can become comfortable with the controls without the algorithm having to adapt to the new user's input style. This robustness is not common in more complicated environments; given the "noise" common to human motion (such as the swaying of a user's hand), recognizing more than a few simple gestures demands new techniques to "learn" a specific user's style of input. We were quite happy to find that our simple path tracking algorithm was fast, responsive and robust to different users motions.

With the pervasive use of personal computers in modern society, nearly everyone has an understanding of how keyboards and mice provide input to a GUI. However, gesture recognition is not common. Few people have interacted with their computers by pointing with their fingers or simply waving their hands. This means that an intelligent user interface design is crucial to building a gesture recognition system that can be quickly understood by new users. We feel that we successfully implemented such an interface. The video feed is mirrored to the user, and a bright cross-hair is displayed over the LED thimble. When a quick motion is made, a holding circle showing the relative center from which this motion originated is displayed, giving the user visual input to decide if they want to "hold" the gesture. When combined with robust gesture recognition, this intelligent user interface allows new operators to become "experts" in the system quickly and effortlessly.

For all our project successes, we encountered our share of disappointments. Despite our best attempts to limit the algorithmic complexity of our recognition system, the software is computationally intensive, and only offers smooth video when a small frame size is used. Furthermore, there is a short lag time between the user's motions and the video display; while this lag is small, is can be aggravating to a user who repeats several motions thinking the first one was not recognized. Some of these performance issues likely stem from our use of the Java programming language. It is reasonable to assume that native C implementations may result in improved performance, allowing for smoother, larger video display with little lag.

4.2 Future Directions

Building a simple gesture recognition system was a pedagogically rewarding and instructive experience (and a lot of fun). But we still feel we only touched the tip of the iceberg. Below we list some of possible extensions of our project:

i. *Multiple Gestures:* We limited our gesture vocabulary to a simple set of four gestures, but there are many interesting applications that involve more gestures and motions.

ii. *Two or more light systems:* Putting LEDs on multiple fingers would allow for many more motions and types of motion to be detected. LEDs on the thumb and index finger would allow users to make rotational movements that could, among other things, scroll through documents/windows, or zoom in and out of images.

iii. *Tracking without LEDs:* The most useful hand gesture recognition system wouldn't require the use of an LED glove or light wand. Accurately tracking the position of a user's fingers is an exceptionally difficult problem, however, and requires much more complicated machine learning algorithms to recognize fingers under varying light conditions.

iv. *Recognizing hand posture:* Besides tracking the position of a user's fingers, much information can be deduced from a user's hand posture. Developing a system to recognize if the user is holding out their palm (to stop a program, for example), or is holding a clenched fist would be useful in building a true vision-based computer interface.

Current state-of-the-art research focuses around psycholinguistic studies that describe a philology of gestures. Researchers have categorized hand movements into conversational gestures, controlling gestures, manipulative gestures, and communicative gestures in an effort to better distinguish and interpret our hand motions. Sign language, an important and highly structured form of communicative gestures, has been an obvious focus. More complicated algorithms using statistical machine learning techniques, like Bayesian networks and Hidden Markov Models have been used to semantically model meaningful human movements, with good results. These current efforts have produced far more complicated systems then we have implemented, being able to recognize movements without the visual limitations we have imposed, and point to a future where visual gesture recognition will be as pervasive as today's keyboard and mouse.

A Code Listing

A.1 figures/PathTracker.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.image.*;
import java.awt.geom.AffineTransform;
import java.util.Vector;

public class PathTracker{

    private static final int XSIZE = 160;
    private static final int YSIZE = XSIZE*3/4;
    private static final int PATH_SIZE = 5;

    private static final int GREEN_THRESHOLD = 240;
    private static final int GREEN_BLUE_THRESHOLD = 80;
    private static final int TRACK_OBJECT_SIZE = 6;
    private static final int HOLD_COUNTER_SIZE = 9;
    private static final int DIR_THRES = 50;
    private static final int DIST_THRES = 10*PATH_SIZE;
    private static final int HOLDING_RADIUS = 25;
    private static final int SEARCH_RADIUS = 25;

    private static String APP_NAME = "scripts/safari";

    private String lastMovement = null;
    private boolean isHolding = false;
    private boolean isMotion = false;
    private Point holdingPoint;
    private int holdCounter;
    private Vector path = new Vector(); //Current Path
    private boolean missedFind = true;
    private Point lastFoundPoint = new Point(XSIZE/2, YSIZE/2);


    public int thresholdPixel(BufferedImage image, int x, int y){
        int blueGreenNum, greenNum;
        Color c = new Color(image.getRGB(x,y));
        greenNum = c.getGreen();
        blueGreenNum = greenNum - c.getBlue();

        if (greenNum > GREEN_THRESHOLD && blueGreenNum > GREEN_BLUE_THRESHOLD) return 1;
        else return 0;
    }
```

```
public Point findObject(BufferedImage image) {
    int l;
    int area = 0, posX = 0, posY = 0;
    double aveX = 0, aveY = 0;
    Color c;

    if(!missedFind){
        for (int y=lastFoundPoint.y - SEARCH_RADIUS;
                y<lastFoundPoint.y + SEARCH_RADIUS; y++){
            for (int x=lastFoundPoint.x - SEARCH_RADIUS;
                    x<lastFoundPoint.x + SEARCH_RADIUS; x++){
                if(x > 0 && x < image.getWidth() && y > 0 && y < image.getHeight()){
                    l = thresholdPixel(image, x, y);
                    if(l != 0){
                        //gather area and positions of each objects
                        area++;
                        posX += x;
                        posY += y;
                    }
                }
            }
        }
    } else {
        for (int y=0; y<image.getHeight(); y++){
            for (int x=0; x<image.getWidth(); x++){
                l = thresholdPixel(image, x, y);
                if(l != 0){
                    //gather area and positions of each objects
                    area++;
                    posX += x;
                    posY += y;
                }
            }
        }
    }

    aveX = (double)posX/area;
    aveY = (double)posY/area;
    Point newP;
    if(area > TRACK_OBJECT_SIZE){
        newP = new Point((int)Math.floor(aveX), (int)Math.floor(aveY));
        lastFoundPoint = newP;
        missedFind = false;
        return newP;
    } else {
```

```
            lastFoundPoint = new Point((int)Math.round(XSIZE/2), (int)Math.round(YSIZE/2));
            missedFind = true;
            //System.out.println("MISSED TRACKING...");
            return null;
        }
    }

    public void printPath(){
        Point p;
        System.out.println("Path:");
        for(int i=0; i<path.size(); i++){
            System.out.println("(" + ((Point)path.get(i)).x + ", " +
                ((Point)path.get(i)).y + ")");
        }
    }

    public String scanPath()
    {
        Point p1, p2;
        double distanceCounter = 0;
        double xDirectionCounter = 0, yDirectionCounter = 0;
        double slope;
        String movement = null;
        String decision = null;

        for(int i=0; i<path.size()-1; i++){
            p1 = (Point)path.get(i);
            p2 = (Point)path.get(i+1);
            distanceCounter += Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
            xDirectionCounter += (p1.x - p2.x);
            yDirectionCounter += (p1.y - p2.y);
        }

        if(distanceCounter > DIST_THRES){
            if(Math.abs(xDirectionCounter) > Math.abs(yDirectionCounter)) {
                //Movement is more horizontal
                if(xDirectionCounter < 0){
                    movement = "Right";
                } else {
                    movement = "Left";
                }
            } else {
                // movement is more vertical
                if(yDirectionCounter > 0){
                    movement = "Up";
                } else {
```

```
                movement = "Down";
            }
        }
    }

    if(isMotion){
        boolean insideHoldRadius = false;
        Point curPoint = (Point)path.lastElement();

        if(Math.sqrt(Math.pow(holdingPoint.x - curPoint.x, 2) +
                Math.pow(holdingPoint.y - curPoint.y, 2)) < HOLDING_RADIUS)
            insideHoldRadius = true;

        if(insideHoldRadius){
            isMotion = false;
            isHolding = false;
            decision = lastMovement;
            lastMovement = null;
            path = new Vector();
        }
        else if(--holdCounter <= 0) {
            isHolding = true;
            decision = lastMovement;
        }

    }
    else if(isHolding == false && movement != null && !movement.equals(lastMovement)){
        System.out.println(movement + " movement started:\t" +
                "Position is [" + xDirectionCounter + "," + yDirectionCounter + "]\t" +
                "Average Dist is " + distanceCounter/path.size());
        isMotion = true;
        holdingPoint = (Point)path.get(0);
        holdCounter = HOLD_COUNTER_SIZE;
        lastMovement = movement;
    }

    return decision;
}

public String updateTracker(Point p) {
    if(p != null) {
        path.add(p);
        if(path.size() > PATH_SIZE){
            path.removeElementAt(0);
        }
        return scanPath();
```

```
        }
        return null;
    }

    /**
     * Create the GUI and show it.  For thread safety,
     * this method should be invoked from the
     * event-dispatching thread.
     */
    private void createAndShowGUI() {
        //Make sure we have nice window decorations.
        JFrame.setDefaultLookAndFeelDecorated(true);

        //Create and set up the window.
        JFrame frame = new JFrame("Path Tracker");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Set up the content pane.
        VideoArea videoArea = new VideoArea( this );
        frame.getContentPane().add(videoArea);

        //Display the window.
        frame.pack();
        frame.setVisible(true);

        while(true) {
            videoArea.draw();
        }
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                PathTracker controller = new PathTracker();
                controller.createAndShowGUI();
            }
        });
        System.out.println( "Exited SwingUtilities.invokeLater()" );
    }

    public static class VideoArea extends JComponent {

        private static final int DOT_RADIUS = 3;
        private static final int XHAIR_LENGTH = 20;
```

```
private static final int XHAIR_RADIUS = 20;

private VideoCapture vc;
private PathTracker controller;
private BufferedImage currentFrame = new BufferedImage(XSIZE, YSIZE,
    BufferedImage.TYPE_INT_RGB);

private static Process process;
private static boolean runScript = true;

public VideoArea(PathTracker controller) {
    this.controller = controller;

    try {
        vc = new VideoCapture(XSIZE,YSIZE);
    } catch (Exception e) {
        System.err.println( "Error initalizing camera: " + e);
        e.printStackTrace();
        System.exit(1);
    }

}

public Dimension getPreferredSize() {
    return new Dimension(XSIZE,YSIZE);
}

public void draw() {

    try {
        // Flip the image horizontally
        AffineTransform tx = AffineTransform.getScaleInstance(-1, 1);
        tx.translate(-currentFrame.getWidth(null), 0);
        currentFrame.setData(    (new AffineTransformOp(tx,
                    AffineTransformOp.TYPE_NEAREST_NEIGHBOR)).filter(
                        vc.getNextRaster(), null));
    } catch (Exception e) {
        System.out.println("Error getting image: " + e);
        e.printStackTrace();
    }

    Point p = controller.findObject(currentFrame);
    String decision = controller.updateTracker(p);
    Point h = controller.holdingPoint;
    String exString = "";
```

```
Graphics g = this.getGraphics();

g.drawImage(currentFrame, 0, 0, null);
if( p != null ) {
    g.setColor(Color.RED);
    g.fillOval(p.x - DOT_RADIUS, p.y - DOT_RADIUS, DOT_RADIUS*2, DOT_RADIUS*2);
    g.setColor(Color.GRAY);
    g.drawOval(p.x - XHAIR_RADIUS, p.y - XHAIR_RADIUS,
        XHAIR_RADIUS*2, XHAIR_RADIUS*2);
    g.drawLine(p.x - 4*DOT_RADIUS, p.y, p.x - XHAIR_LENGTH, p.y);
    g.drawLine(p.x + 4*DOT_RADIUS, p.y, p.x + XHAIR_LENGTH, p.y);
    g.drawLine(p.x, p.y - 4*DOT_RADIUS, p.x, p.y - XHAIR_LENGTH);
    g.drawLine(p.x, p.y + 4*DOT_RADIUS, p.x, p.y + XHAIR_LENGTH);

    g.setColor(Color.RED);

    if(decision != null){
        if(decision.equals("Left")){
            g.drawLine(p.x + 4*DOT_RADIUS, p.y, p.x + XHAIR_LENGTH, p.y);
            exString = "osascript " + APP_NAME + "-key-left.scpt";
        } else if(decision.equals("Right")){
            g.drawLine(p.x - 4*DOT_RADIUS, p.y, p.x - XHAIR_LENGTH, p.y);
            exString = "osascript " + APP_NAME + "-key-right.scpt";
        } else if(decision.equals("Up")){
            g.drawLine(p.x, p.y - 4*DOT_RADIUS, p.x, p.y - XHAIR_LENGTH);
            exString = "osascript " + APP_NAME + "-key-up.scpt";
        } else if(decision.equals("Down")){
            g.drawLine(p.x, p.y + 4*DOT_RADIUS, p.x, p.y + XHAIR_LENGTH);
            exString = "osascript " + APP_NAME + "-key-down.scpt";
        }

        if(runScript && !controller.isHolding){
            try{
                process = Runtime.getRuntime().exec(exString);
            } catch (Exception i){
                System.out.println(i);
            }
        }
    }
}

if(controller.isHolding) g.drawString("Holding", XSIZE - 50, 20);
if(decision != null) g.drawString(decision, 20, 20);

g.setColor(Color.GREEN);
if(controller.isMotion && h != null){
```

```
                g.fillOval(h.x - DOT_RADIUS, h.y - DOT_RADIUS, DOT_RADIUS*2, DOT_RADIUS*2);
                g.drawOval(h.x - HOLDING_RADIUS, h.y - HOLDING_RADIUS,
                    HOLDING_RADIUS*2, HOLDING_RADIUS*2);
            }
        }

        protected void paintComponent(Graphics g) {
            draw();
        }

    } // end public static class VideoArea()

} // end public class PathTracker()
```

A.2 figures/VideoCapture.java

```
import java.awt.Image;
import java.awt.image.*;
import java.io.File;

import javax.imageio.ImageIO;

import quicktime.QTRuntimeException;
import quicktime.QTRuntimeHandler;
import quicktime.QTSession;
import quicktime.qd.PixMap;
import quicktime.qd.QDGraphics;
import quicktime.qd.QDRect;
import quicktime.std.StdQTConstants;
import quicktime.std.sg.SGVideoChannel;
import quicktime.std.sg.SequenceGrabber;
import quicktime.util.RawEncodedImage;

public class VideoCapture {
    private SequenceGrabber grabber;
    private SGVideoChannel channel;
    private RawEncodedImage rowEncodedImage;

    private int width;
    private int height;
    private int videoWidth;

    private int[] pixels;
    private BufferedImage image;
    private WritableRaster raster;
```

```
public VideoCapture(int width, int height) throws Exception {
    this.width = width;
    this.height = height;
    try {
        QTSession.open();
        QDRect bounds = new QDRect(width, height);
        QDGraphics graphics = new QDGraphics(bounds);
        grabber = new SequenceGrabber();
        grabber.setGWorld(graphics, null);
        channel = new SGVideoChannel(grabber);
        channel.setBounds(bounds);
        channel.setUsage(StdQTConstants.seqGrabPreview);
        //channel.settingsDialog();
        grabber.prepare(true, false);
        grabber.startPreview();
        PixMap pixMap = graphics.getPixMap();
        rowEncodedImage = pixMap.getPixelData();

        videoWidth = width + (rowEncodedImage.getRowBytes() - width * 4) / 4;
        pixels = new int[videoWidth * height];
        image = new BufferedImage(
            videoWidth, height, BufferedImage.TYPE_INT_RGB);
        raster = WritableRaster.createPackedRaster(DataBuffer.TYPE_INT,
            videoWidth, height,
            new int[] { 0x00ff0000, 0x0000ff00, 0x000000ff }, null);
        raster.setDataElements(0, 0, videoWidth, height, pixels);
        image.setData(raster);
        QTRuntimeException.registerHandler(new QTRuntimeHandler() {
            public void exceptionOccurred(
                    QTRuntimeException e, Object eGenerator,
                    String methodNameIfKnown, boolean unrecoverableFlag) {
                System.out.println("what should i do?");
            }
        });
    } catch (Exception e) {
        QTSession.close();
        throw e;
    }
}

public void dispose() {
    try {
        grabber.stop();
        grabber.release();
        grabber.disposeChannel(channel);
        image.flush();
```

```
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            QTSession.close();
        }
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    public int getVideoWidth() {
        return videoWidth;
    }

    public int getVideoHeight() {
        return height;
    }

    public void getNextPixels(int[] pixels) throws Exception {
        grabber.idle();
        rowEncodedImage.copyToArray(0, pixels, 0, pixels.length);
    }

    public Image getNextImage() throws Exception {
        grabber.idle();
        rowEncodedImage.copyToArray(0, pixels, 0, pixels.length);
        raster.setDataElements(0, 0, videoWidth, height, pixels);
        image.setData(raster);
        return image;
    }

    public Raster getNextRaster() throws Exception {
        grabber.idle();
        rowEncodedImage.copyToArray(0, pixels, 0, pixels.length);
        raster.setDataElements(0, 0, videoWidth, height, pixels);
        return (Raster)raster;
    }

    public static void main(String args[]){
        try{
        System.out.println("Starting up");
```

```
        VideoCapture myVC = new VideoCapture(640, 480);
        System.out.println("Constructor done");
        BufferedImage myImage;
        int numPics = 100;
        for(int i=0; i<numPics; i++){
            myImage = (BufferedImage) myVC.getNextImage();
            System.out.println("Image Taken: " + i);
            ImageIO.write(myImage, "jpg", new File("test/Test" + (i + 10*numPics) + ".jpg"));
        }

        System.out.println("Image Write Done");
        } catch (Exception e){
            System.out.println(e);
        }
    }
}
```