**Cat and Mouse**

The aim of this exercise is to provide insight into a simple computer vision application. By providing this kind of entry-level understanding, it should be possible to gain an appreciation of how more complex computer vision systems operate.

For this exercise, we will be using a Java-based programming language called Processing. It’s a powerful language, but has a gentle learning curve and is easy to get started for less experienced programmers.

**Task 1: Understand and Run Template Code**

A template program has been provided to help get you started. Take a look at it and try to understand what it does. Comments in the code have been included to explain some of the less obvious lines of code. When you feel that you have a reasonable grasp of the code, run it by clicking the “play” button.



Note: Don’t worry if you get a “nat:sink” warning – this is to do with the webcam driver and frame drawing. This is not out of the ordinary and won’t effect your ability to finish the assignment.

**Task 2: Choose Fimo Colour**

In some of the remaining steps you will need a colour to “train” your code on. Choose a block of Fimo with an appealing colour (Note: It will make your life a lot easier if you choose a bright and bold colour !) You may wish to make a small model with your chosen colour (don’t mix the colours, it will cause problems later on).

**Task 3: Detect Hue and Saturation**

In the template code, you can see how the “brightness” function has been used to test each pixel in frames grabbed from the camera. At the moment it just distinguishes between bright and dark pixels. Make this IF statement more sophisticated using the “hue” and “saturation” functions in order to allow the code to be able to identify your chosen Fimo colour. The objective is to “blank out” as many of the pixels of your chosen colour as possible. Try to get the balance right, so that you blank out the Fimo, but no other pixels of similar colour from the rest of the scene. This will take a fair amount of tinkering and patience. To help you get to grips with Hue, Saturation and Brightness, take a look at the “Colour Selector” tool (Top-bar menu Tools> Colour Selector…)

**Task 4: Draw image**

Now let’s make things more interesting by drawing an image over the top of the Fimo object. An image has already been added to the project (in the *data* folder) it is called “kitten.png”. You can add additional images of your own by dropping them into the *data* folder. Once you have an image that you would like to use, you must add the following code to your program:

1. Near the top of program declare a variable to hold the image: *PImage myPicture;*
2. In the setup method, load the image in from file: *myPicture = loadImage(“kitten.png”);*
3. At the end of your draw function, show the image with: *image(myPicture, 100, 200);*

This last step draws the your image 100 pixels along and 200 pixels down from the top left-hand corner of the window. Obviously you’ll need to change these numbers to move the image to a different location. Perhaps the best way to position the image is to get the average x and y coordinates of ALL of the “Fimo coloured” pixels. Clearly you’ll need to need a bit of maths for this !

**Task 5: Scale image**

So we now have the image tracking the Fimo object in 2 dimensions. But nothing happens if we move the object towards, or away from the screen. This is because we can’t use the camera to track objects in 3D space (other devices can do this, but not a simple camera). We can however simulate 3D tracking with a simple trick – the closer the object is to the camera, the more “Fimo coloured” pixels there will be; the further away it is, the less pixels. As you loop through the pixels in a camera frame, count up how many pixels there are of your chosen colour. You can then use this number to scale your image by passing two additional parameters when you call the *image* function:

*image(myImage, xPosition, yPosition, imageWidth, imageHeight)*

It might take a little bit of experimentation (trial and error) to devise the right scaling calculation !

**Task 6: Move mouse (optional)**

Let’s try something a little bit different using the pixel tracking data. Rather than just moving an image around the camera window, we are going to create a mouse driver to move the mouse pointer around the whole screen. To do this, we are going to need to use a special “Robot” object. This is an object that allows us to automate various desktop activities (including mouse movement). First add the following two lines of code to the end of your setup function:

*try { mouseController = new Robot();*

*} catch(AWTException awte) { }*

These create a new Robot to help us move the mouse around. Next add the following line to the end of your draw function:

*mouseController.mouseMove(pointerX, pointerY);*

Where pointerX and pointerY are the coordinate of where you want to mouse to move to. It might take a bit of work to map camera frame coordinates to screen coordinates. There are built-in variables called *displayWidth* and *displayHeight* that might help with this.

**Task 7: Play “tough growth” (optional)**

In order to fully test the mouse moving capabilities of your software, we need a task to assess a user’s ability to control a mouse pointer. With your program running, open the following web page:

<http://www.freewebarcade.com/game/tough-growth/>

Use this game to thoroughly test your software (you might want to turn the sound off !)

**Task 8: Self learning (optional)**

In the previous tasks we “taught” the system to recognise your chosen colour by manually encoding the hue, saturation and brightness thresholds in the code. Can you think of a way to get the software to automatically learn these parameters ? Imagine using the mouse to click on an area of colour shown in the camera view window, so that the computer would use this as the tracking colour. You might find it useful to add to the mousePressed() function.