CS377S

DESIGNING APPLICATIONS THAT SEE

Computer Vision for HCI

ASSIGNMENT #2: IMAGE PROCESSING AND OBJECT TRACKING

Due February 5, 2008 (in lecture)

 Reflection  Ideation  Exercise  Bonus Challenge

Image Processing *(12 Points)*

a) What is an image brightness histogram? What is histogram stretching, and what does it accomplish?

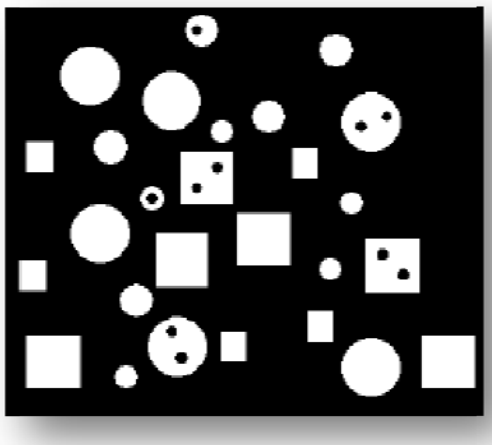
b) Briefly describe the effect of each of the following filters:

0.1	0.1	0.1
0.1	0.1	0.1
0.1	0.1	0.1

0	0	0
0	0	0
0	0	1

-1	-2	-1
0	0	0
1	2	1

c) You are hired by a widget factory to design an inspection algorithm that separates widgets into two categories: widgets with holes and widgets without holes. As widgets pass by on a conveyor belt, a camera captures black and white images, like the one below. The widgets do not overlap or touch, but may be close to each other. They can be any shape or size.



Write a pseudo-code description of an algorithm to count the number of widgets with holes and the number of widgets without holes. Include a simple block diagram of the algorithm to make it easy to understand. State any assumptions you make in your algorithm about the size or shape of the widgets.

d) Name two problems that can occur when capturing a digital image. For each problem, explain how it can be rectified using digital image processing techniques.

Computer Vision Design Challenges (*9 Points*)

EyeToy: Kinetic Combat is a new physical fitness video game for the Sony Playstation 2. It teaches players martial arts moves, giving them automatic feedback on their technique based on input from the EyeToy camera. The creators of *Kinetic Combat* made an instructional video explaining how to get started with the game. Watch the video here:

http://cs377s.stanford.edu/assignments/eyetoy.html

The setup for the game is more complicated than one might expect! List the variety of problems that can arise, and how they might impact the game's computer vision algorithms. How do the game designers suggest preventing these problems? How might one design around some of these issues, either by changing the nature of the activity or by changing the underlying computer vision algorithm?

Motion Tracking in MATLAB (*12 Points*)

In MATLAB, video sequences are typically represented as an array with four dimensions: height, width, color, and time. In this exercise you will load a video sequence into MATLAB and perform some basic object tracking.

a) Start by downloading the following video:
http://cs377s.stanford.edu/assignments/ball.avi
Verify that you can view the video file in a standard player, such as Windows Media Player, before proceeding. If you are unable to play the video, you may need to install the DivX codec, available at http://www.divx.com.

b) Start MATLAB and switch to the directory containing the video. Begin by getting some basic information about the video:
```
aviinfo('ball.avi')
```
This command will display the video size, length, and framerate, among other statistics.

c) Load the video into MATLAB with the command:
```
ball = aviread('ball.avi');
```
This will take some time, as MATLAB will decompress each frame of the video and load the entire sequence into memory, in a very large multidimensional matrix.

d) Try watching the video you just loaded with the `movie` command:
```
movie(ball);
```
MATLAB should open a figure window and play the video. You will see a ball hanging on a string from the ceiling, swinging past some books in the Gates library.

e) The variable `ball` contains a data structure that holds the video frame in memory. This structure consists of two fields, `cdata` and `colormap`. The `colormap` field is a table of colors that is used only for *indexed color* videos. *Indexed color* is a more compact, lower

fidelity representation of an image that stores a color index at each pixel instead of a color value. To determine the color at that pixel, you would look up the color index in the color map. However, most videos that you will encounter, including this one, use *true color*, meaning that a full 3-component color value is stored at each pixel. This information is stored in the `cdata` structure. You can think of `ball.cdata` as a four-dimensional array of size 240-by-320-by-3-by-500. It represents an image sequence with 240 rows, 320 columns, 3 color components, and 500 frames. You can extract the first frame of the video as follows:

```
frame = ball(1);
```

Since we are only interested in `cdata`, the image component of the frame, we need to extract this component into a separate variable to view the frame:

```
frameimage = frame.cdata;
imshow(frameimage);
```

f) We can process the video by thinking of each frame as a separate image. For example, suppose we wanted to apply a Gaussian blur to every fifth video frame. We could step through the video in increments of five frames, running the blur operation each time:

```
filter = fspecial('gaussian', [15, 15], 7);
for i=5:5:size(ball,2)
  I=ball(i).cdata;
  ball(i).cdata = imfilter(I, filter, 'symmetric', 'conv');
end
```

Run this code and play the video again with the `movie` command to see the difference.

g) After modifying a video, we sometimes want to save a copy. Export our new video as follows:

```
movie2avi(ball,'blurryball.avi');
```

If successful, this command will store the processed video in a new file in your working directory. If you get errors about a missing video compressor, don't worry about it for now – you can download the compressor if you like, but it's not strictly necessary for this exercise.

h) Now that we have learned how to load and process video, we will explore two simple techniques for tracking the ball in this video sequence: frame differencing and background subtraction. Begin by reloading the video, since we want to use the original copy, not the one we modified:

```
ball = aviread('ball.avi');
```

i) The absolute difference between successive frames can be used to divide an image frame into changed and unchanged regions. Since only the ball moves, we expect the changed region to be associated only with the ball, or possibly with its shadow. To begin, we convert each frame to grayscale using `rgb2gray`, and store it in a new variable called `grball`. Running the loop "backwards," from `numframes` down to 1, is a common MATLAB programming trick to ensure that `grball` is initialized to its final size the first time through the loop.

```
numframes = size(ball,2);
for k = numframes:-1:1
  grball(:, :, k) = rgb2gray(ball(k).cdata);
end
```

j) Next we will compute differences between successive frames using the `imabsdiff` function, and store them in a variable called `framediffs`.

```
for k = numframes-1:-1:1
  framediffs(:,:,k) = imabsdiff(grball(:,:,k), grball(:,:, k+1));
end
```

Let's examine one of the difference images. Load the image representing the change between frames 100 and 101 of the video sequence:

```
imshow(framediffs(:,:,100), [])
```

k) Notice that there are two bright arcs in the image, corresponding to the ball positions in frame 100 and frame 101. We can now threshold this sequence of difference images using some functions you have seen before:

```
for k = numframes-1:-1:1
  bwdiffs(:,:, k) = ...
    im2bw(framediffs(:,:,k),graythresh(framediffs(:,:,k)));
end
```

l) Finally, we can label each individual region (using `bwlabel`) and compute its corresponding center of mass (using `regionprops`). Because most of our difference frames have two or more regions, we will average the centroids of all of the detected regions to estimate the location of the ball, and store the list of locations in a variable called `position`.

```
for k = numframes-1:-1:1
  s = regionprops(bwlabel(bwdiffs(:,:, k)), 'centroid');
  centroids = [s.Centroid];
  xavg = mean(centroids(:,1:2:end));
  yavg = mean(centroids(:,2:2:end));
  position(:,k) = [xavg,yavg];
end
```

m) Use this sequence of commands to construct a plot of the x- and y-coordinates of the ball over time. Save this plot and include it in your assignment hand-in.

```
subplot(2, 1, 1);
plot([1:499], position(1,:)), ylabel('x');
subplot(2, 1, 2);
plot([1:499], position(2,:)), ylabel('y');
xlabel('time (s)');
```

n) There seem to be some glitches in the data. Inspect the sequence of frame differences and try to diagnose the problem. See if you can smooth out the data by doing additional processing on the difference frames before extracting the region positions, or by some other method. If you manage to improve the results, save the new plot and include it in your hand-in along with your modified code.

o) To see how good our tracking is, we can overlay the detected ball position back onto the original video. Use these commands to draw a yellow rectangle over the detected ball location in the original video clip:

```
for k = 1:length(position)
  I = ball(k).cdata;
  xpos = int32(position(1,k));
  ypos = int32(position(2,k));
  I(ypos-5:ypos+5,xpos-5:xpos+5,1:2) = 255;
```

```
  ball(k).cdata = I;
end
```

Play the modified clip with the `movie` command and see how closely our tracking matches the actual ball position.

p) Another approach to tracking the ball is to estimate the background image and subtract it from each frame. There are various ways of doing this, but a simple way is to find the pixel-wise maximum among a range of neighboring frames. We can do this using dilation, a morphological operation that will activate a pixel (change it from 0 to 1) if one of its neighboring pixels is activated. Normally we dilate a single image in the x- and y-directions, but we can also dilate between multiple images by using a structuring element oriented along the frame dimension. This will produce a new image in which each pixel has the maximum value at its location across a range of frames. We will use a structuring element that is 1x1x10 – a single pixel spanning ten frames.

```
background = imdilate(grball, ones(1, 1, 10));
imshow(background(:,:,100))
```

q) Next, we compute the absolute difference between each frame and its corresponding background estimate. For each frame, we subtract the background and then threshold the image.

```
for k = numframes:-1:1
  diff = imabsdiff(grball(:,:,k), background(:,:,k));
  thresh = graythresh(diff);
  bwdiffs(:,:,k) = im2bw(diff,thresh);
end
```

r) Now we want to compute the location of the ball in each frame. Some frames contain small extra spots resulting from noise. We can solve this problem by assuming that the ball is the largest object in each frame. We will initialize a variable called `centroids` that will store a list of region centers.

```
centroids = zeros(numframes, 2);
for k = 1:numframes
  L = bwlabel(bwdiffs(:, :, k));
  s = regionprops(L, 'area', 'centroid');
  area_vector = [s.Area];
  [tmp, idx] = max(area_vector);
  centroids(k, :) = s(idx(1)).Centroid;
end
```

s) As before, let's examine the quality of our tracking by creating a plot of the ball's estimated locations as a function of time:

```
subplot(2, 1, 1)
plot([1:500], centroids(:,1)), ylabel('x')
subplot(2, 1, 2)
plot([1:500], centroids(:, 2)), ylabel('y')
xlabel('time (s)')
```

Save this plot to include in your assignment hand-in. How does this tracking method compare

to the previous one? Can you think of a way to improve on this one, or to combine the two approaches for better results?

t) Tracking objects in the real world is often more complicated than in this contrived example. Try running your object tracking algorithms on this video example:
http://cs377s.stanford.edu/assignments/ball2.avi
Produce new plots using both motion tracking techniques. What exactly is the problem here? How might you improve your tracking algorithm to cope with this sort of messy data? You don't have to actually implement any changes – just suggest one or more possible approaches.

Setting Up Processing (0 *Points*)

The goal of this exercise is simply to set up the Processing and JMyron toolkits on your computer for use in the next problem. Since these tools are designed to work with live video, you will need a webcam in order to use them. If you don't have a webcam, you can still install the tools, but you won't be able to verify that they are working. Although these tools are quite easy to use once installed, getting them working can be a headache, particularly on Windows platforms. Please allow yourself some time to troubleshoot installation problems.

Processing is an open-source Java-based language and environment for programming interactive multimedia applications. Its creators describe it as "an environment for learning the fundamentals of computer programming within the context of the electronic arts… an electronic sketchbook for developing ideas." Because of its ease of learning and focus on interactive images, animation, and sound, it is frequently used by artists and designers. Processing can produce programs that run locally as well as web-embeddable Java applets.

a) Begin by downloading and installing the latest release of the Processing environment from http://www.Processing.net/. You don't need to run an install program; simply unzip the `processing-0135` folder and you're ready to run Processing.

b) JMyron is a video capture and analysis plug-in that can be used in conjunction with Processing. Designed for artists and inexperienced computer vision programmers, it provides a variety of basic image analysis functions for tracking motion, color, and shape. Download JMyron from SourceForge:
http://webcamxtra.sourceforge.net/
Note that there are several versions available; you should download the version designed for use with Processing.

c) After downloading and unzipping JMyron, open the `JMyron0025` folder. You will find three subfolders: `JMyron`, `JMyron Examples`, and `Extra DLLs`. Copy the `JMyron` folder into your `Processing\libraries` directory. Copy the `JMyron Examples` folder into your `Processing\Examples` directory. Copy the two DLL files in the `Extra DLLs` folder into your `Processing` root directory.

d) To use Processing's built-in video capture library in Windows, you will also need to install both QuickTime and WinVDig, a QuickTime video digitizer for Windows. Mac users can skip this step. QuickTime is available here:

http://www.apple.com/quicktime/
You can find WinVDig in the course software directory:
http://cs377s.stanford.edu/software/WinVDIG_101.exe
Use QTCap, the capture application included with the WinVDig install, to make sure that WinVDig is working properly.

e) Once you get QTCap working, you're ready to go! Start Processing and load the *AsciiVideo* example, located under *File ▸ Examples ▸ Libraries ▸ Video(Capture) ▸ AsciiVideo*. If the built-in video library is working, running the example should display the live video from your camera, rendered as ASCII art.

f) You should also test JMyron by running the *Myron_BoundingBoxes* example, located under *File ▸ Examples ▸ JMyron Examples*. Click the play button and a point your camera at something bright. You should see a live video window with boxes drawn around the brightest objects in the frame.

g) Problems with JMyron? If you are using an Intel-based Mac, you may need a different version of one of the JMyron library files. You can download it from the course software directory:
http://cs377s.stanford.edu/software/libJMyron.jnilib.zip
Unzip the archive and replace the old `libJMyron.jnilib` file by copying the new lib to the `Processing/libraries/JMyron/library/` directory. Restart Processing and try running one of JMyron examples again. The JMyron methods `getForcedWidth()` and `getForcedHeight()` don't work on Macs, so you may need to comment out calls to these functions in order to run the examples.

Camera Control in Processing (*6 Points*)

Use the Processing environment to build a very simple game that is driven by computer vision input. The particular method of camera input is up to you. You might control the game by motion tracking through background subtraction or frame differencing, or you might try tracking the position of a particular color. You might require the player to make special hand gestures, or you might even have the player control the game with a custom prop, such as an LED flashlight or a specially colored paddle, that is detected by the camera.

The choice of game is also left to your imagination. You are welcome to find the source code for an existing game written in Processing (or Java) and modify it to use vision-based tracking, provided the game was not originally designed to use a camera as input. Here are links to several games written in Processing that might lend themselves particularly well to camera control. All of these games have source code available that you can modify.

- 3D Pong (by Charlie Mezak)
  http://www.charliemezak.com/java/paddleGame/
- Roach Attack (by Cadin Batrack)
  http://www.thepencilfarm.com/games/roach_attack/
- Cytris (by Andre Michelle)
  http://processing.andre-michelle.com/cytris/

- SpaceGame (by Benjie Nelson)
  http://www.stanford.edu/~bmnelson/

You do not need to constrain yourself to a traditional game. If you prefer, you may build an interesting visualization that is driven by the camera, provided that (a) it is fun and engaging and (b) it is clear to the user that he is controlling the visualization using the camera, and he can take conscious steps to produce interesting audio or visual output. Here are some examples of interesting visualizations that might be modified to use camera input:

- VineGardener (by Jeffrey Crouse)
  http://idt.gatech.edu/~jcrouse/6310/VineGardener/applet/index.html
- WindPainter (by Marcello Bastéa-Forte)
  http://www.cs.unm.edu/~cello/processing/windpainter/

You are also welcome to program your own game logic, if you have a clever idea and are feeling ambitious. Here are examples of some game ideas that might work well:

- Whack-A-Mole: Animated targets pop up on the screen, and you must wiggle your hand or head over the targets to hit them.
- Bumper Cars: Point a webcam in different directions to steer a car and hit or dodge obstacles.
- Simon Says: Mimic a series of increasingly complex sequences by moving in certain ways or holding up different colored objects in front of the camera.

When you have finished programming your game, record a digital video of someone playing it. Post this video online, along with a copy of your source code. Include the URL of the gameplay video and a link to the game's source code in your assignment hand-in.