Course notes, CS664, 8/26/04

- Administrivia:
  - Sign up sheet (name, netid, Pr(credit), year, field/major)
  - 2 Problem Sets, done in pairs – implement an algorithm described in class
  - Short oral report on a vision paper of your choice
  - Possibly a few short in-class quizes on lecture material
  - Final Project – Write a proposal. What is and isn't original research.

- Course emphasis will be on **algorithms** for computer vision (oddly enough, this is not a typical approach)

- Most of our time will be spent on some mathematical techniques that are widely used in vision

- We will also talk about some applications

- First part of the course: short tour of vision. Why it is compelling and hard.

- We will focus on problem definitions and applications, leaving technical details for later.

- You should quickly get a sense of the field, and perhaps some ideas for a final project area?

- What is a computer vision algorithm? Input is one or more images, output is (what, exactly?)

- Well, something that depends upon the content of the images, i.e. what is in them.

- Not, for example, the image's dimensions or a compressed version of the image.

- This is actually a bit subtle, since the most recent image compression algorithms like MPEG-4 make use of information about content.

- For example, to efficiently compress "talking heads" videoconferencing video, you code the background and the head differently, which requires figuring out which is which.

- OK, what is the input? An image is a 2D array of pixels, each of which has an intensity.

- For a grayscale image, typically 0 (black) to 255 (white). Color is fairly similar (RGB).

- How could such a simple datastructure merit an entire field? Very easily, as it turns out.

- To see why, we will jump right into a simple vision algorithm for edge detection.

- Algorithms, of course, are computational ways of solving **problems**

- CS dogma insists on a clean separation of problem and algorithm, a notion which, BTW, some vision people like to think they invented...

- Classical example: sorting **problem** is defined by its input (list of numbers) and output (same numbers, in increasing order).

- A sorting **algorithm** like mergesort or quicksort is a method to solve this problem.

- An algorithm in turn might depend upon various mathematical techniques, such as dynamic programming.

- So first of all we need to define the edge detection problem, before we can talk about algorithms.

- What is the problem definition? Right away we are in trouble.

- The intuition behind edge detection is to produce a binary image where the 1's represent "edges".

- The basic idea is to compute something like a black and white artists sketch, which you then use for further processing (edge detection isn't particularly useful in and of itself).

- For instance, you might do edge detection and then look for ovoid collections of edge pixels as possible faces.

- Or you might have a 3D wireframe model of some object, like a car or a stapler.

- You could then find that object's pose and position using edges.

- Another motivation, which some people in vision find compelling, is that there is good evidence that the human visual system does something like edge detection fairly early.

- When you compare this with, say, sorting, you see that the problem definition is very imprecise.

- In fact, there is no precise problem definition for edge detection (or for almost any other problem in computer vision). Situation is a bit like in simulation for engineering (think of simulating a bridge or an airplane wing).

- The way that edge detectors usually work is by looking for "big changes" in the intensity values and labeling those as edges. Works real well for synthetic images!

- This simple issue has important consequences
  - Most papers describe an algorithm that computes binary images somehow, and give some intuitions about why it should be "edge-like", and show 1-2 example images
  - Can't tell if your algorithm is working; it's a judgement call. (Having an application in mind is very helpful, but not a panacea)
  - Cannot compare two algorithms in any meaningful way. (State of the art in empirical evaluation is woeful, though much improved of late)

- But it's worse than this example suggests, because edge detection is much harder than it looks.

- This is a frustrating thing about vision; people think calculus is hard and recognizing people is easy.

- In fact, the opposite is true for computers (and even for humans, judging by neuron count).

- The "important" edges, which any artist would draw, are often very subtle when you look at the actual intensities.

- Examples: edges of person's face or nose; corners in a room. These usually do not in fact result in big changes in the intensities!

- Obviously, people/artists are using a great deal of knowledge that isn't in the image to figure out what's going on.

- This observation is true but not helpful... ("The world would be much better off if we all were healthy and happy")

- No one really knows how to represent such information, or how

to use it effectively.

- The only (partial) exception is medical imaging, of which we will speak more later

- In fact, for many applications (especially robotics) you can't really make the kind of assumptions that would help you (such as knowing all possible objects). [Caveat: OCR, which isn't really vision]

- There is also a lot of interest in exploiting the statistics of natural imagery, an idea that came out of Cornell's Psychology department. But it doesn't (at this point) result in better algorithms.

- A final reason that vision problems are hard lies in the noisy nature of the data.

- If you take two pictures of a stationary scene, even a fraction of a second apart, you won't get identical images.

- Take a close look at an HDTV picture sometime, you'll see "mosquito noise" plus lots of other junk, some of which can be fixed by using higher quality sensors (uncompressed or lossy compression)

- Edge detectors will not as a general rule produce the same output from what appears to a human to be the same input.

- Part of this is due to the fact that at some point the algorithm has to essentially threshold how big an intensity change there is.

- When the change is close to the threshold, the noise will cause instability.

- It's not just a matter of building better cameras; you really need algorithms that will ignore various changes that people usually "don't see". Example: color constancy.

- Note that as a rule this threshold is an algorithm parameter (usually called "scale"), and the same detector can produce coarser or finer outputs (fewer or more edges).

- Many vision problems are just like edge detection, in that they aren't really **problems** in the formal sense of the term (Garey & Johnson).

- For example,consider the image segmentation problem.

- The goal is to divide the input image into "meaningful" pieces.

- The output of a segmentation algorithm will be a partition of the image, just like the output of an edge detection algorithm is a binary image.

- Ideally the pieces would correspond to "objects".

- Segmentation would be incredibly useful if it actually worked (the same is true of teleportation and cold fusion...)

- For instance you could "colorize" movies easily, or have a smart version of Photoshop that allowed you to easily fake news pictures of Kerry & Fonda or Bush & Bin Laden.

- In fact, many vision algorithms would be extremely useful for special effects, an area where a huge amount is done by hand even today.

- But segmentation is extremely ill-defined; how many objects are there in this room? Is my arm an object? My shirt sleeve?

- Segmentation algorithms generally have some resemblance to edge detection algorithm, in that they look for big intensity changes.

- Obviously, they produce regions rather than edges (but note that some edge detectors produce edges form connected edges).

- Moreover, as a rule they use information from farther way in the image.

- They also usually have some kind of scale parameter.

- A currently popular view of segmentation is to think of it as graph partitioning, where the nodes are pixels, the edges go to neighbors (often 4-connected), and the edge weight represents "affinity".

- Obvious way to compute affinity is $w_{p,q} = K - |I(p) - I(q)|$. Need this to be non-negative (why?) and also high for similar pixels (why?)

- This actually describes a number of recent papers: take some graph partitioning algorithm (a new one, or a variant of an old one), apply it to the graph from an image, produce a few pictures and publish. Popular approaches involve cuts and/or spectral methods.

- It's also a good example of the way the field works, and what we will study in this course: an ill-defined problem (albeit one that people have strong intuitions about), and algorithms based on fairly sophisticated mathematical techniques. This is also an example of a good area for final projects...

- The classical segmentation problem works from intensity values only.

- There are some interesting variants, for example involving texture.

- A typical problem with segmentation algorithms is that they perform poorly when there are highly-textured objects present (such as a plaid shirt).

- Texture is an extremely ill-defined notion, even by the standards of vision ("texture is what makes things look the way they look independent of lighting conditions, scale and position"). A

philosopher once called this maneuver "giving a name to your confusion".

- A particularly cool application of texture is "image inpainting" where texture is essentially extrapolated, to allow you to delete things from a scene (imagine a person standing in front of a cornfield). This requires both texture synthesis and texture segmentation.