Extension 3: Vision

Golan Levin

A well-known anecdote relates how, sometime in 1966, the legendary Artificial Intelligence pioneer Marvin Minsky directed an undergraduate student to solve "the problem of computer vision" as a summer project (1). This anecdote is often resuscitated to illustrate how egregiously the difficulty of computational vision has been underestimated. Indeed, nearly forty years later, the discipline continues to confront numerous unsolved (and perhaps unsolvable) challenges, particularly with respect to high-level "image understanding" issues such as pattern recognition and feature recognition. Nevertheless, the intervening decades of research have yielded a great wealth of well-understood, low-level techniques that are able, under controlled circumstances, to extract meaningful information from a camera scene. These techniques are indeed elementary enough to be implemented by novice programmers at the undergraduate or even high-school level.

Computer vision in interactive art

The first interactive artwork to incorporate computer vision was, interestingly enough, also one of the first interactive artworks. Myron Krueger's legendary Videoplace, developed between 1969 and 1975, was motivated by his deeply felt belief that the entire human body ought to have a role in our interactions with computers. In the Videoplace installation, a participant stands in front of a backlit wall and faces a video projection screen. The participant's silhouette is then digitized, and its posture, shape and gestural movements analyzed. In response, Videoplace synthesizes graphics such as small "critters" which climb up the participant's projected silhouette, or colored loops drawn between the participant's fingers. Krueger also allowed participants to paint lines with their fingers, and, indeed, entire shapes with their bodies; eventually, Videoplace offered over 50 different compositions and interactions. Videoplace is notable for many "firsts" in the history of human-computer interaction. Some of its interaction modules, for example the ones shown in the figures, allowed two participants in mutually remote locations to participate in the same shared video space, connected across the network — an implementation of the first multi-person virtual reality, or, as Krueger termed it, an "artificial reality." Videoplace, it should be noted, was developed before the mouse became the ubiquitous desktop device it is today, and was (in part) created to demonstrate interface alternatives to the keyboard terminals which dominated computing so completely in the early 1970's.

Messa di Voce, created by this article's author in collaboration with Zachary Lieberman, uses whole-body vision-based interactions similar to Krueger's, but combines them with speech analysis and situates them within a kind of projection-based augmented reality. In this audiovisual performance, the speech, shouts and songs produced by two abstract vocalists are visualized and augmented in real-time by synthetic graphics. To accomplish this, a computer uses a set of vision algorithms to track the locations of the performers' heads; this computer also analyzes the audio signals coming from the performers' microphones. In response, the system displays various kinds of visualizations on a projection screen located just behind the performers; these visualizations are synthesized in ways which are tightly coupled to the sounds being spoken and sung. With the help of the head-tracking system, moreover, these visualizations are projected such that they appear to emerge directly from the performers' mouths.

Rafael Lozano-Hemmer's installation Standards and Double Standards (2004) incorporates full-body input in a less direct, more metaphorical context. This work consists of fifty leather belts, suspended at waist height from robotic servo-motors mounted on the ceiling of the exhibition room. Controlled by a computer vision-based tracking system, the belts rotate automatically to follow the public, turning their buckles slowly to face passers-by. Lozano-Hemmer's piece "turns a condition of pure surveillance into an 'absent crowd' using a fetish of paternal authority: the belt" (2).

The theme of surveillance plays a foreground role in David Rokeby's Sorting Daemon (2003). Motivated by the artist's concerns about the increasing use of automated systems for profiling people as part of the "war on terrorism", this site-specific installation works toward the automatic construction of a diagnostic portrait of its social (and racial) environment. Rokeby writes: "The system looks out onto the street, panning, tilting and zooming, looking for moving things that might be people. When it finds what it thinks might be a person, it removes the person's image from the background. The extracted person is then divided up according to areas of similar colour. The resulting swatches of colour are then organized [by hue, saturation and size] within the arbitrary context of the composite image"(3) projected onsite at the installation's host location.

Another project themed around issues of surveillance is Suicide Box by the Bureau of Inverse Technology (Natalie Jeremijenko and Kate Rich). Presented as a device for measuring the hypothetical "Despondency Index" of a given locale, the Suicide Box nevertheless records very real data regarding suicide jumpers from the Golden Gate Bridge. According to the artists, "The Suicide Box is a motion-detection video system, positioned in range of the Golden Gate Bridge, San Francisco in 1996. It watched the bridge constantly and when it recognized vertical motion, captured it to a video record. The resulting footage displays as a continuous stream the trickle of people who jump off the bridge. The Golden Gate Bridge is the premiere suicide destination in the United States; a 100 day initial deployment period of the Suicide Box recorded 17 suicides. During the same time period the Port Authority counted only 13." (4) Elsewhere, Jeremijenko has explained that "the idea was to track a tragic social phenomenon which was not being counted — that is, doesn't count"(5). The Suicide Box has met with considerable controversy, ranging from ethical questions about recording the suicides, to others disbelieving that the recordings could be real. Jeremijenko, whose aim is to address the hidden politics of technology, has pointed out that such attitudes express a recurrent theme — "the inherent suspicion of artists working with material evidence" — evidence obtained, in this case, with the help of machine-vision based surveillance.

Considerably less macabre is Christian Möller's clever Cheese installation (2003), which the artist developed in collaboration with the California Institute of Technology and the Machine Perception Laboratories of the University of California, San Diego. Motivated, perhaps, by the culture-shock of his relocation to Hollywood, the German-born Möller directed "six actresses to hold a smile for as long as they could, up to one and half hours. Each ongoing smile is scrutinized by an emotion recognition system, and whenever the display of happiness fell below a certain threshold, an alarm alerted them to show more sincerity" (6). The installation replays recordings of the analyzed video on six flat panel monitors, with the addition of a fluctuating graphic level-meter to indicate the strength of each actress' smile. The technical implementation of this artwork's vision-based emotion recognition system is quite sophisticated.

As can be seen from the examples above, artworks employing computer vision range from the highly formal and abstract, to the humorous and sociopolitical. They concern themselves with the activities of willing participants, paid volunteers, or unaware strangers. They track people of interest at a wide variety of spatial scales, from extremely intimate studies of their facial expressions, to the gestures of their limbs, and to movements of entire bodies. The examples above represent just a small selection of notable works in the field, and of ways in which people (and objects) have been tracked and dissected by video analysis. Other noteworthy artworks which use machine vision include Marie Sester's Access; Joachim Sauter and Dirk Lüsebrink's Zerseher and Bodymover; Scott Snibbe's Boundary Functions and Screen Series; Camille Utterback and Romy Achituv's TextRain; Jim Campbell's Solstice; Christa Sommerer and Laurent Mignonneau's A-Volve; Danny Rozin's Wooden Mirror; Chico MacMurtrie's Skeletal Reflection, and various works by Simon Penny, Toshio Iwai, and numerous others. No doubt many more vision-based artworks remain to be created, especially as these techniques gradually become incorporated into developing fields like physical computing and robotics.

Elementary computer vision techniques

To understand how novel forms of interactive media can take advantage of computer vision techniques, it is helpful to begin with an understanding of the kinds of problems that vision algorithms have been developed to address, and their basic mechanisms of operation. The fundamental challenge presented by digital video is that it is computationally “opaque.” Unlike text, digital video data in its basic form — stored solely as a stream of rectangular pixel buffers — contains no intrinsic semantic or symbolic information. There is no widely agreed-upon standard for representing the content of video, in a manner analogous to HTML, XML or even ASCII for text (though some new initiatives, notably the MPEG-7 description language, may evolve into this in the future). As a result, a computer, without additional programming, is unable to answer even the most elementary questions about whether a video stream contains a person or object, or whether an outdoor video scene shows daytime or nighttime, etcetera. The discipline of computer vision has developed to address this need.

Many low-level computer vision algorithms are geared to the task of distinguishing which pixels, if any, belong to people or other objects of interest in the scene. Three elementary techniques for accomplishing this are frame differencing, which attempts to locate features by detecting their movements; background subtraction, which locates visitor pixels according to their difference from a known background scene; and brightness thresholding, which uses hoped-for differences in luminosity between foreground people and their background environment. These algorithms, described below, are extremely simple to implement and help constitute a base of detection schemes from which sophisticated interactive systems may be built.

Example 1: Detecting motion
The movements of people (or other objects) within the video frame can be detected and quantified using a straightforward method called frame differencing. In this technique, each pixel in a video frame F1 is compared with its corresponding pixel in the subsequent frame F2. The difference in color and/or brightness between these two pixels is a measure of the amount of movement in that particular location. These differences can be summed across all of the pixels’ locations, in order to provide a single measurement of the aggregate movement within the video frame. In some motion detection implementations, the video frame is spatially subdivided into a grid of cells, and the values derived from frame differencing are reported for each of the individual cells. For accuracy, the frame differencing algorithm depends on relatively stable environmental lighting, and on having a stationary camera (unless it is the motion of the camera which is being measured).

Example 2: Detecting presence
A technique called background subtraction makes it possible to detect the presence of people or other objects in a scene, and to distinguish the pixels which belong to them from those which do not. The technique operates by comparing each frame of video with a stored image of the scene’s background, captured at a point in time when the scene was known to be empty. For every pixel in the frame, the absolute difference is computed between its color and that of its corresponding pixel in the stored background image; areas which are very different from the background are likely to represent objects of interest. Background subtraction works well in heterogeneous environments, but it is very sensitive to changes in lighting conditions, and depends on objects of interest having sufficient contrast against the background scene.

Example 3: Detection through brightness thresholding
With the aid of controlled illumination (such as backlighting) and/or surface treatments (such as high-contrast paints), it is possible to ensure that objects of interest are considerably darker than, or lighter than, their surroundings. In such cases objects of interest can be distinguished based on their brightness alone. To do this, each video pixel’s brightness is compared to a threshold value, and tagged as foreground or background accordingly.

Example 4: Brightness tracking
A rudimentary scheme for object tracking, ideal for tracking the location of a single illuminated point (such as a flashlight), finds the location of the single brightest pixel in every fresh frame of video. In this algorithm, the brightness of each pixel in the incoming video frame is compared with the brightest value yet encountered in that frame; if a pixel is brighter than the brightest value yet encountered, then the location and brightness of that pixel are stored. After all of the pixels have been examined, then the brightest location in the video frame is known. This technique relies on an operational assumption that there is only one such object of interest. With trivial modifications, it can equivalently locate and track the darkest pixel in the scene, or track multiple, differently-colored objects.

Naturally, many more software techniques exist, at every level of sophistication, for detecting, recognizing, and interacting with people and other objects of interest. Each of the tracking algorithms described above, for example, can be found in elaborated versions which amend its various limitations. Other easy-to-implement algorithms can compute specific features of a tracked object, such as its area, center of mass, angular orientation, compactness, edge pixels, and contour features such as corners and cavities. On the other hand, some of the most difficult-to-implement algorithms, representing the cutting edge of computer vision research today, are able (within limits) to recognize unique people, track the orientation of a person's gaze, or correctly identify facial expressions. Pseudocodes, source codes, and/or ready-to-use, executable implementations of all of these techniques can be found on the Internet in excellent resources like Daniel Huber's Computer Vision Homepage, Robert Fisher's HIPR (Hypermedia Image Processing Reference), or in the software toolkits discussed below.

Computer vision in the physical world

Unlike the human eye and brain, no computer vision algorithm is completely "general", which is to say, able to perform its intended function given any possible video input. Instead, each software tracking or detection algorithm is critically dependent on certain unique assumptions about the real-world video scene it is expected to analyze. If any of these expectations is not met, then the algorithm can produce poor or ambiguous results, or even fail altogether. For this reason, it is essential to design physical conditions in tandem with the development of computer vision code, and/or to select software techniques which are best compatible with the available physical conditions.

Background subtraction and brightness thresholding, for example, can fail if the people in the scene are too close in color or brightness to their surroundings. For these algorithms to work well, it is greatly beneficial to prepare physical circumstances which naturally emphasize the contrast between people and their environments. This can be achieved with lighting situations that silhouette the people, for example, or through the use of specially-colored costumes. The frame-differencing technique, likewise, fails to detect people if they are stationary, and will therefore have very different degrees of success detecting people in videos of office waiting rooms compared with, for instance, videos of the Tour de France bicycle race.

A wealth of other methods exists for optimizing physical conditions in order to enhance the robustness, accuracy and effectiveness of computer vision software. Most are geared towards ensuring a high-contrast, low-noise input image. Under low-light conditions, for example, one of the most helpful such techniques is the use of infrared (IR) illumination. Infrared, which is invisible to the human eye, can supplement the light detected by conventional black-and-white security cameras. Using IR significantly improves the signal-to-noise ratio of video captured in low-light circumstances, and can even permit vision systems to operate in (apparently) complete darkness. Another physical optimization technique is the use of retroreflective marking materials, such as those manufactured by 3M Corporation for safety uniforms. These materials are remarkably efficient at reflecting light back towards their source of illumination, and are ideal aids for ensuring high-contrast video of tracked objects. If a small light is placed coincident with the camera's axis, objects with retroreflective markers will be detected with tremendous reliability.

Finally, some of the most powerful physical optimizations for machine vision can be made without intervening in the observed environment at all, through well-informed selections of the imaging system's camera, lens, and frame-grabber components. To take one example, the use of a "telecentric" lens can significantly improve the performance of certain kinds of shape-based or size-based object recognition algorithms. For this type of lens, which has an effectively infinite focal length, magnification is nearly independent of object distance. As one manufacturer describes it, "an object moved from far away to near the lens goes into and out of sharp focus, but its image size is constant. This property is very important for gaging three-dimensional objects, or objects whose distance from the lens is not known precisely" (7). Likewise, polarizing filters offer a simple, non-intrusive solution to another common problem in video systems, namely glare from reflective surfaces. And a wide range of video cameras is available, optimized for conditions like high-resolution capture, high-frame-rate capture, short exposure times, dim light, ultraviolet light, or thermal imaging. Clearly, it pays to research imaging components carefully.

As we have seen, computer vision algorithms can be selected to best negotiate the physical conditions presented by the world, and likewise, physical conditions can be modified to be more easily legible to vision algorithms. But even the most sophisticated algorithms and highest-quality hardware cannot help us find meaning where there is none, or track an object which cannot be described in code. It is therefore worth emphasizing that some visual features contain more information about the world, and are also more easily detected by the computer, than others. In designing systems to "see for us," we must not only become freshly awakened to the many things about the world which make it visually intelligible to us, but also develop a keen intuition about their ease of computability. The sun is the brightest point in the sky, and by its height also indicates the time of day. The mouth cavity is easily segmentable as a dark region, and the circularity of its shape is also closely linked to vowel sound. The pupils of the eye emit an easy-to-track infrared retroreflection, and they also indicate a person's direction of gaze. Or in the dramatic case of Natalie Jeremijenko's Suicide Box, discussed earlier: vertical motion in the video frame is easy to find through simple frame-differencing, and (in a specific context) it can be a stark indicator of a tragic event. In judging which features in the world are most profitably selected for analysis by computer vision, we will do well to select those graphical facts about the world which not only are easy to detect, but also simplify its semantic understanding.

Tools for computer vision

It can be a rewarding experience to implement machine vision techniques directly from first principles using code such as the examples provided in this section. To make this possible, the only requirement of one's software development environment is that it should provide direct read-access to the array of video pixels obtained by the computer's frame-grabber. Hopefully, the example algorithms discussed earlier illustrate that creating low-level vision algorithms from first principles isn't so hard. Of course, a vast range of functionality can also be immediately obtained from readymade, "off-the-shelf" solutions. Some of the most popular machine vision toolkits take the form of "plug-ins" or extension libraries for commercial authoring environments geared towards the creation of interactive media. Such plug-ins simplify the developer's problem of connecting the results of the vision-based analysis to the audio, visual and textual affordances generally provided by such authoring systems.

Many vision plug-ins have been developed for Max/MSP/Jitter, a visual programming environment which is widely used by electronic musicians and VJs. Originally developed at the Parisian IRCAM research center in the mid-1980s, and now marketed commercially by the California-based Cycling'74 company, this extensible environment offers powerful control of (and connectivity between) MIDI devices, real-time sound synthesis and analysis, OpenGL-based 3D graphics, video filtering, network communications, and serial control of hardware devices. The various computer vision plug-ins for Max/MSP/Jitter, such as David Rokeby's SoftVNS, Eric Singer's Cyclops, and Jean-Marc Pelletier's CV.Jit, can be used to trigger any Max processes or control any system parameters. Pelletier's toolkit, which is the most feature-rich of the three, is also the only which is freeware. CV.Jit provides abstractions to assist users in tasks such as image segmentation, shape and gesture recognition, motion tracking, etc. as well as educational tools that outline the basics of computer vision techniques.

Some computer vision toolkits take the form of stand-alone applications, and are designed to communicate the results of their analyses to other environments (such as Processing, Director or Max) through protocols like MIDI, serial RS-232, UDP or TCP/IP networks. BigEye, developed by the STEIM (Studio for Electro-Instrumental Music) group in Holland, is a simple and inexpensive example. BigEye can track up to 16 objects of interest simultaneously, according to their brightness, color and size. The software allows for a simple mode of operation, in which the user can quickly link MIDI messages to many object parameters, such as position, speed and size. Another example is the powerful EyesWeb open platform, a free system developed at the University of Genoa. Designed with a special focus on the analysis and processing of expressive gesture, EyesWeb includes a collection of modules for real-time motion tracking and extraction of movement cues from human full-body movement; a collection of modules for analysis of occupation of 2D space; and a collection of modules for extraction of features from trajectories in 2D space. EyesWeb's extensive vision affordances make it highly recommended for students.

The most sophisticated toolkits for computer vision generally demand greater familiarity with digital signal processing, and require developers to program in compiled languages like C++, rather than languages like

Java, Lingo or Max. The Intel Integrated Performance Primitives (IPP) library for example, is among the most general commercial solutions available for computers with Intel-based CPUs. The OpenCV library, by contrast, is a free, open-source toolkit with nearly similar capabilities, and a tighter focus on commonplace computer vision tasks. The capabilities of these tools, as well as all of those mentioned above, are continually evolving.

Processing includes a basic video library that handles getting pixel information from a camera or movie file, as demonstrated in the examples included with this text. The computer vision capabilities of Processing are extended by libraries like Myron, which handles video input and has basic image processing capabilities. Other libraries connect Processing to EyesWeb and OpenCV, and can be found on the libraries page of processing.org: http://processing.org/reference/libraries.

Conclusion

Computer vision algorithms are increasingly used in interactive and other computer-based artworks to track people's activities. Techniques exist which can create real-time reports about people's identities, locations, gestural movements, facial expressions, gait characteristics, gaze directions, and other characteristics. Although the implementation of some vision algorithms require advanced understandings of image processing and statistics, a number of widely-used and highly effective techniques can be implemented by novice programmers in as little as an afternoon. For artists and designers who are familiar with popular multimedia authoring systems like Macromedia Director and Max/MSP/Jitter, a wide range of free and commercial toolkits are additionally available which provide ready access to more advanced vision functionalities.

Since the reliability of computer vision algorithms is limited according to the quality of the incoming video scene, and the definition of a scene's "quality" is determined by the specific algorithms which are used to analyze it, students approaching computer vision for the first time are encouraged to apply as much effort to optimizing their physical scenario as they do to their software code. In many cases, a cleverly designed physical environment can permit the tracking of phenomena that might otherwise require much more sophisticated software. As computers and video hardware become more available, and software-authoring tools continue to improve, we can expect to see the use of computer vision techniques increasingly incorporated into media-art education, and into the creation of games, artworks and many other applications.

Notes:

(1) http://mechanism.ucsd.edu/~bill/research/mercier/2ndlecture.pdf
(2) http://www.fundacion.telefonica.com/at/rlh/eproyecto.html
(3) http://homepage.mac.com/davidrokeby/sorting.html
(4) http://www.bureauit.org/sbox/
(5) http://www.wired.com/news/culture/0,1284,64720,00.html
(6) http://www.christian-moeller.com/
(7) http://www.mellesgriot.com/pdf/pg11-19.pdf

Code

Video can be captured into Processing from USB Cameras, IEEE 1394 Cameras, or Video Cards with composite or S-video input devices. The examples that follow assume you already have a camera working with Processing. Before trying these examples, first get the examples included with the Processing software to work. Sometimes you can plug a camera into your computer and it will work immediately. Other times it's a difficult process involving trail-and-error changes. It depends on the operating system, the camera, and how the computer is configured. For the most up-to-date information, refer to the Video reference on the Processing website: http://www.processing.org/reference/libraries/video/

Example 1: Frame Differencing

```
// Code to quantify the amount of movement in the video frame
// using a simple frame-differencing technique.

import processing.video.*;

int numPixels;
int previousFrame[];
Capture video;

void setup(){
  size(320, 240);
  framerate(30); //frameRate(30);
  video = new Capture(this, width, height, 30);
  numPixels = video.width * video.height;
  // Create an array to store the previously captured frame
  previousFrame = new int[numPixels];
}

void captureEvent(Capture c) {
  video.read();
}

void draw() {
  color currColor, prevColor;  // Current and previous pixels
  int currR, currG, currB;     // Current color values
  int prevR, prevG, prevB;     // Previous color values
  int diffR, diffG, diffB;     // Computed differences
  int movementSum = 0;         // Amount of movement in the frame
  loadPixels(); //beginPixels();
  // For each pixel in the video frame...
  for (int i=0; i<numPixels; i++){
    currColor = video.pixels[i];
    prevColor = previousFrame[i];
    // Extract the red, green, and blue components from current pixel
    currR = (currColor >> 16) & 0xFF;
    currG = (currColor >> 8) & 0xFF;
    currB =  currColor & 0xFF;
    // Extract red, green, and blue components from previous pixel
    prevR = (prevColor >> 16) & 0xFF;
    prevG = (prevColor >> 8) & 0xFF;
    prevB = prevColor & 0xFF;
    // Compute the difference of the red, green, and blue values
    diffR = abs(currR - prevR);
    diffG = abs(currG - prevG);
    diffB = abs(currB - prevB);
    // Add these differences to the running tally.
    movementSum += diffR + diffG + diffB;
    // Render the difference image to the screen.
    pixels[i] = color(diffR, diffG, diffB);
    // Swap the current information into the previous.
    previousFrame[i] = currColor;
  }
  updatePixels(); //endPixels();
  println(movementSum);  // Print out the total amount of movement
}
```

Example 2: Background Subtraction

```
// Detect the presence of people and objects in the frame using
// a simple background-subtraction technique. To initialize the
// background, pressing a key.

import processing.video.*;

int numPixels;
int backgroundImage[];
Capture video;

void setup(){
  size(320, 240);
  framerate(30); //frameRate(30);
  video = new Capture(this, width, height, 30);
  numPixels = video.width * video.height;
  // Create array to store the background image
  backgroundImage = new int [numPixels];
}

void captureEvent(Capture c) {
  video.read();
}

void draw(){
  color currColor, bkgdColor;  // Current and background pixel
  int currR, currG, currB;     // Hold current color values.
  int bkgdR, bkgdG, bkgdB;     // Hold previous color values.
  int diffR, diffG, diffB;     // Hold computed differences
  // Stores, for the current frame, the difference between the
  // current frame and the stored background
  float presenceSum = 0;
  loadPixels(); //beginPixels();
  // For each pixel in the video frame...
  for (int i=0; i < numPixels; i++){
    // Fetch the current color in that location, and also the color
    // of the background in that spot.
    currColor = video.pixels[i];
    bkgdColor = backgroundImage[i];
    // Extract the red, green, and blue components of the current
    // pixel’s color.
    currR = (currColor >> 16) & 0xFF;
    currG = (currColor >> 8) & 0xFF;
    currB = currColor & 0xFF;
    // Extract the red, green, and blue components of the
    // background pixel’s color.
    bkgdR = (bkgdColor >> 16) & 0xFF;
    bkgdG = (bkgdColor >> 8) & 0xFF;
    bkgdB = bkgdColor & 0xFF;
    // Compute the difference of the red, green, and blue values
    diffR = abs (currR - bkgdR);
    diffG = abs (currG - bkgdG);
    diffB = abs (currB - bkgdB);
    // Add these differences to the running tally.
    presenceSum += diffR+diffG+diffB;
    // Render the difference image to the screen.
    pixels[i] = color(diffR,diffG,diffB);
  }
  updatePixels(); //endPixels();
  println(presenceSum);  // Print out the total amount of movement
}

// When a key is pressed, capture the background image into the
// backgroundImage buffer, by copying each of the current frame’s
// pixels into it.
void keyPressed(){
  loadPixels(); //beginPixels();
  arraycopy(video.pixels, backgroundImage);
  updatePixels(); //endPixels();
}
```

Example 3: Brightness Thresholding

```
// Determines if a test location (such as the cursor)
// is contained within the silhouette of a dark object.

import processing.video.*;

int numPixels;
Capture video;

void setup(){
  size(320, 240);
  noStroke();
  framerate(30); //frameRate(30);
  video = new Capture(this, width, height, 30);
  numPixels = video.width * video.height;
}

void captureEvent(Capture c) {
  video.read();
}

void draw(){
  color black = color(0);
  color white = color(255);
  int threshold = 127;
  // Declare variables to store a pixel’s color.
  color pixelValue;
  float pixelBrightness;

  // Split the image into dark and light areas:
  // For each pixel in the video frame, fetch the current color in
  // that location,and compute the brightness of that pixel.
  loadPixels(); //beginPixels();
  for (int i=0; i<numPixels; i++){
    pixelValue = video.pixels[i];
    pixelBrightness = brightness (pixelValue);
    // Make the video into black-and-white
    // depending on whether each pixel is brighter
    // or darker than a threshold value.
    if (pixelBrightness > threshold){
      pixels[i] = white;
    } else {
      pixels[i] = black;
    }
  }
  updatePixels(); //endPixels();
  // Test a location to see where it is contained.
  // Fetch the pixel at the test location (the cursor),
  // and compute its brightness.
  int testValue = get (mouseX, mouseY);
  float testBrightness = brightness (testValue);
  // If the test location is brighter than threshold, draw a yellow
  // ellipse at the test location. Otherwise, draw a blue ellipse
  if (testBrightness > threshold){
    fill (255, 204, 0);
    ellipse (mouseX-10, mouseY-10, 20,20);
  } else {
    fill (0, 153, 204);
    ellipse (mouseX-10, mouseY-10, 20,20);
  }
}
```

Example 4: Brightness Tracking

```
// Tracks the brightest pixel in a live video signal

import processing.video.*;

Capture video;

void setup(){
  size(320, 240);
  framerate(30); //frameRate(30);
  noStroke();
  video = new Capture(this, width, height, 30);
}

void captureEvent(Capture c) {
  video.read();
}

void draw(){
  // Draw the webcam video onto the screen.
  image(video, 0, 0, width, height);
  // Declare some numbers to be computed later
  int brightestX = 0;        // X-coordinate of the brightest video pixel
  int brightestY = 0;        // Y-coordinate of the brightest video pixel
  float brightestValue = 0;  // Brightness of the brightest video pixel
  // Search for the brightest pixel:
  // For each row of pixels in the video image and for each pixel in
  // the y'th row,compute each pixel's index in the video,
  video.loadPixels(); //video.beginPixels();
  int index = 0;
  for (int y=0; y<video.height; y++){
    for (int x=0; x<video.width; x++){
      int pixelValue = video.pixels[index];
      // Fetch the color stored in that pixel,
      // and determine the brightness of that pixel.
      float pixelBrightness = brightness(pixelValue);
      // If that value is brighter than any previous, then store the
      // brightness of that pixel, as well as its (x,y) location.
      if (pixelBrightness > brightestValue){
        brightestValue = pixelBrightness;
        brightestY = y;
        brightestX = x;
      }
      index++;
    }
  }
  video.updatePixels(); //video.endPixels();
  // Draw a circle at the brightest pixel.
  fill(255, 204, 0);
  ellipse( brightestX-10, brightestY-10, 20,20);
}
```

References

Computer vision software toolkits

Camurri, Antonio et. al. Eyesweb. Vision-oriented software development environment. http://www.eyesweb.org/

Cycling'74 Inc. Max/MSP/Jitter. Graphic software development environment. http://www.cycling74.com/

Davies, Bob. et. al. OpenCV. Open-source computer vision library.
http://sourceforge.net/projects/opencvlibrary/

Nimoy, Joshua. Myron. Xtra (plug-in) for Macromedia Director and Processing.
http://webcamxtra.sourceforge.net/

Pelletier, Jean-Marc. CV.Jit. Extension library for Max/MSP/Jitter.
http://www.iamas.ac.jp/~jovan02/cv/

Rokeby, David. SoftVNS. Extension library for Max/MSP/Jitter.
http://homepage.mac.com/davidrokeby/softVNS.html

Singer, Eric. Cyclops. Extension library for Max/MSP/Jitter.
http://www.cycling74.com/products/cyclops.html

STEIM (Studio for Electro-Instrumental Music). BigEye. Video analysis software.
http://www.steim.org/steim/bigeye.html

Myron (WebCamXtra)
http://webcamxtra.sourceforge.net/

Texts and artworks

Bureau of Inverse Technology. Suicide Box
http://www.bureauit.org/sbox/

Bechtel, William. The Cardinal Mercier Lectures at the Catholic University of Louvain: An Exemplar Neural Mechanism: The Brain's Visual Processing System. Ch. 2, p.1., 2003.
http://mechanism.ucsd.edu/~bill/research/mercier/2ndlecture.pdf

Fisher, Robert, et. al. HIPR (The Hypermedia Image Processing Reference).
http://homepages.inf.ed.ac.uk/rbf/HIPR2/index.htm

Fisher, Robert, et. al. CVonline: The Evolving, Distributed, Non-Proprietary, On-Line Compendium of Computer Vision. http://homepages.inf.ed.ac.uk/rbf/CVonline/

Huber, Daniel et. al. The Computer Vision Homepage.
http://www-2.cs.cmu.edu/~cil/vision.html

Krueger, Myron. Artificial Reality II. Addison-Wesley Professional, 1991.

Levin, Golan and Lieberman, Zachary. Messa di Voce. Interactive installation, 2003.
http://www.tmema.org/messa

Levin, Golan and Lieberman, Zachary. "In-Situ Speech Visualization in Real-Time Interactive Installation and Performance." Proceedings of The 3rd International Symposium on Non-Photorealistic Animation and Rendering, June 7-9 2004, Annecy, France.
http://www.flong.com/writings/pdf/messa_NPAR_2004_150dpi.pdf

Lozano-Hemmer, Rafael. Standards and Double Standards. Interactive installation.
http://www.fundacion.telefonica.com/at/rlh/eproyecto.html

Melles Griot Corporation. Machine Vision Lens Fundamentals.
http://www.mellesgriot.com/pdf/pg11-19.pdf

Moeller, Christian. Cheese. Installation artwork, 2003.
http://www.christian-moeller.com/

Rokeby, David. Sorting Daemon. Computer-based installation, 2003.
http://homepage.mac.com/davidrokeby/sorting.html

Shachtman, Noah. “Tech and Art Mix at RNC Protest”. Wired News, 8/27/2004.
http://www.wired.com/news/culture/0,1284,64720,00.html

Sparacino, Flavia. “(Some) computer vision based interfaces for interactive art and entertainment installations”. INTER_FACE Body Boundaries, ed. Emanuele Quinz, Anomalie, n.2, Paris, France, Anomos, 2001.
http://www.sensingplaces.com/papers/Flavia_isea2000.pdf