16 Video

"I have no memory. It's like looking in a mirror and seeing nothing but mirror."
—Alfred Hitchcock

In this chapter:
– Displaying live video.
– Displaying recorded video.
– Creating a software mirror.
– How to use a video camera as a sensor.

16.1 Before Processing

Now that we have explored static images in *Processing*, we are ready to move on to moving images, specifically from a live camera (and later, from a recorded movie). Using a digital video camera connected to a PC or Mac requires a few simple steps before any code can be written.

On a Mac:

- Attach a camera to your computer.
- Make sure any necessary software and drivers for the camera are installed.
- Test the camera in another program, such as iChat, to make sure it is working.

On a PC:

- Attach a camera to your computer.
- Make sure any necessary software and drivers for the camera are installed.
- Test the camera. (Use whatever software that came with your camera that allows you to view the video stream.)
- Install QuickTime (version 7 or higher). If you are using a previous version of QuickTime, you must make sure you choose "custom" installation and select "QuickTime for Java."
- You will also need to install a vdig (video digitizer) that allows QuickTime applications (what we are creating) to capture video in Windows. The vdig situation is in flux so I recommend you check this book's web site for updates. Although no longer being developed, a free vdig is available at this site: *http://eden.net.nz/7/20071008/*. You can also consider using Abstract Plane's vdig (*http://www.abstractplane.com.au/products/vdig.jsp*), which costs a small fee, but has more advanced features.

Admittedly, the process is a bit less complicated on a Mac. This is because *Processing* uses the QuickTime libraries in Java to handle video. On Apple computers, QuickTime comes preinstalled (although sometimes software updates can cause issues), whereas on Windows, we have to make sure we have taken the steps to install and properly configure QuickTime. At the end of this chapter, we will see

that there are few third party libraries available in *Processing* that do not require QuickTime for video capture on Windows.

Exercise 16-1: Hook a camera up to your computer. Does it work in another program (not Processing*)? If you are on a PC, install a vdig and test.*

16.2 Live Video 101

Once you have made sure you have a working camera connected to your machine, you can start writing *Processing* code to capture and display the image. We begin by walking through the basic steps of importing the video library and using the Capture class to display live video.

Step 1. Import the *Processing* video library.

If you skipped Chapter 12 on *Processing* libraries, you might want to go back and review the details. There is not much to it here, however, since the video library comes with the *Processing* application. All you need to do is import the library. This is done by selecting the menu option Sketch→Import Library→video, or by typing the following line of code (which should go at the very top of your sketch):

```
import processing.video.*;
```

Using the "Import Library" menu option does nothing other than automatically insert that line into your code, so manual typing is entirely equivalent.

Step 2. Declare a Capture object

We learned in Chapter 12 how to create objects from classes built into the *Processing* language. For example, we made PImage objects from the *Processing* PImage class. PImage, it should be noted, is part of the *processing.core* library and, therefore, no import statement was required. The *processing.video* library has two useful classes inside of it—Capture, for live video, and Movie, for recorded video. We will start with declaring a Capture object.

```
Capture video;
```

Step 3. Initialize the Capture object.

The Capture object "video" is just like any other object. As we learned in Chapter 8, to construct an object, we use the ***new*** operator followed by the constructor. With a Capture object, this code typically appears in ***setup()***, assuming you want to start capturing video when the sketch begins.

```
video = new Capture();
```

The above line of code is missing the appropriate arguments for the constructor. Remember, this is not a class we wrote ourselves so there is no way for us to know what is required between the parentheses without consulting the reference. The online reference for the Capture constructor can be found on the *Processing* web site at:

http://www.processing.org/reference/libraries/video/Capture.html

The reference will show there are several ways to call the constructor (see *overloading* in Chapter 23 about multiple constructors). A typical way to call the Capture constructor is with four arguments:

```
void setup() {
  video = new Capture(this,320,240,30);
}
```

Let's walk through the arguments used in the Capture constructor.

- **this**—If you are confused by what *this* means, you are not alone. We have never seen a reference to *this* in any of the examples in this book so far. Technically speaking, *this* refers to the instance of a class in which the word *this* appears. Unfortunately, such a definition is likely to induce head spinning. A nicer way to think of it is as a self-referential statement. After all, what if you needed to refer to your *Processing* program within your own code? You might try to say "me" or "I." Well, these words are not available in Java, so instead we say "this." The reason we pass "this" into the Capture object is we are telling it: "Hey listen, I want to do video capture and when the camera has a new image I want you to alert *this* applet."
- **320**—Fortunately for us, the first argument, *this*, is the only confusing one. 320 refers to the width of the video captured by the camera.
- **240**—The height of the video.
- **30**—The desired framerate captured, in frames per second (fps). What framerate you choose really depends on what you are doing. If you only intend to capture an image from the camera every so often when the user clicks the mouse, for example, you would not need a high framerate and could bring that number down. However, if you want to display full motion video onscreen, then 30 is a good bet. This, of course, is simply your desired framerate. If your computer is too slow or you request a very high resolution image from the camera, the result might be a slower framerate.

Step 4. Read the image from the camera.

There are two strategies for reading frames from the camera. We will briefly look at both and somewhat arbitrarily choose one for the remainder of the examples in this chapter. Both strategies, however, operate under the same fundamental principle: *we only want to read an image from the camera when a new frame is available to be read*.

In order to check if an image is available, we use the function ***available()***, which returns true or false depending on whether something is there. If it is there, the function ***read()*** is called and the frame from the camera is read into memory. We do this over and over again in the ***draw()*** loop, always checking to see if a new image is free for us to read.

```
void draw() {
  if (video.available()) {
    video.read();
  }
}
```

The second strategy, the "event" approach, requires a function that executes any time a certain event, in this case a camera event, occurs. If you recall from Chapter 3, the function ***mousePressed()*** is executed whenever the mouse is pressed. With video, we have the option to implement the function ***captureEvent()***, which is invoked any time a capture event occurs, that is, a new frame is available from the camera. These event functions (***mousePressed()***, ***keyPressed()***, ***captureEvent()***, etc.) are sometimes referred to as a "callback." And as a brief aside, if you are following closely, this is where *this* fits in. The Capture object, "video," knows to notify *this* applet by invoking ***captureEvent()*** because we passed it a reference to ourselves when creating "video."

captureEvent() is a function and therefore needs to live in its own block, outside of ***setup()*** and ***draw()***.

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

To summarize, we want to call the function ***read()*** whenever there is something for us to read and we can do so by either checking manually using ***available()*** within ***draw()*** or allowing a callback to handle it for us—***captureEvent()***. Many other libraries that we will explore in later chapters (such as network and serial) will work exactly the same way.

Step 5. Display the video image.

This is, without a doubt, the easiest part. We can think of a Capture object as a PImage that changes over time and, in fact, a Capture object can be utilized in an identical manner as a PImage object.

```
image(video,0,0);
```

All of this is put together in Example 16-1.

Example 16-1: Display video

```
// Step 1. Import the video library
import processing.video.*;

// Step 2. Declare a Capture object
Capture video;

void setup() {
  size(320,240);
  // Step 3. Initialize Capture object via Constructor
  // video is 320x240, @15 fps
  video = new Capture(this,320,240,15);
}
```

Step 1. Import the video library!

Step 2. Declare a Capture object!

Step 3. Initialize Capture object! This starts the capturing process.

fig. 16.1

```
void draw() {
  // Check to see if a new frame is available
  if (video.available()) {
    // If so, read it.
    video.read();
  }
  // Display the video image
  image(video,0,0);
}
```

Step 4. Read the image from the camera.

Step 5. Display the image.

Again, anything we can do with a PImage (resize, tint, move, etc.) we can do with a Capture object. As long as we ***read()*** from that object, the video image will update as we manipulate it. See Example 16-2.

Example 16-2: Manipulate video image

```
// Step 1. Import the video library
import processing.video.*;

Capture video;

void setup() {
  size(320,240);
  video = new Capture(this,320,240,15);
}

void draw() {
  if (video.available()) {
    video.read();
  }

  // Tinting using mouse location
  tint(mouseX,mouseY,255);
  // Width and height according to mouse
  image(video,0,0,mouseX,mouseY);
}
```

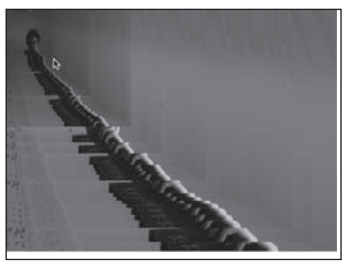


fig. 16.2

A video image can also be tinted and resized just as with a PImage.

Every single image example from Chapter 15 can be recreated with video. Following is the "adjusting brightness" example (15-19) with a video image.

Example 16-3: Adjust video brightness

```
// Step 1. Import the video library
import processing.video.*;

// Step 2. Declare a Capture object
Capture video;

void setup() {
  size(320,240);
  // Step 3. Initialize Capture object via Constructor
  video = new Capture(this,320,240,15); // video is 320x240, @15 fps
  background(0);
}
```

```
void draw() {
  // Check to see if a new frame is available
  if (video.available()) {
    // If so, read it.
    video.read();
  }

  loadPixels();
  video.loadPixels();
  for (int x = 0; x < video.width; x++) {
    for (int y = 0; y < video.height; y++) {
      // calculate the 1D location from a 2D grid
      int loc = x + y*video.width;
      // get the R,G,B values from image
      float r,g,b;
      r = red   (video.pixels[loc]);
      g = green (video.pixels[loc]);
      b = blue  (video.pixels[loc]);

      // calculate an amount to change brightness based on proximity to the mouse
      float maxdist = 100;// dist(0,0,width,height);
      float d = dist(x,y,mouseX,mouseY);
      float adjustbrightness = (maxdist-d)/maxdist;
      r *= adjustbrightness;
      g *= adjustbrightness;
      b *= adjustbrightness;
      // constrain RGB to make sure they are within 0-255 color range
      r = constrain(r,0,255);
      g = constrain(g,0,255);
      b = constrain(b,0,255);
      // make a new color and set pixel in the window
      color c = color(r,g,b);
      pixels[loc] = c;
    }
  }
  updatePixels();
}
```

fig. 16.3

Exercise 16-2: Recreate Example 15-14 (pointillism) to work with live video.

16.3 Recorded Video

Displaying recorded video follows much of the same structure as live video. *Processing*'s video library only accepts movies in QuickTime format. If your video file is a different format, you will either have to convert it or investigate using a third party library. Note that playing a recorded movie on Windows does *not* require a vdig.

Step 1. Instead of a Capture object, declare a Movie object.

```
Movie movie;
```

Step 2. Initialize Movie object.

```
movie = new Movie(this, "testmovie.mov");
```

The only necessary arguments are *this* and the movie's filename enclosed in quotes. The movie file should be stored in the sketch's data directory.

Step 3. Start movie playing.

Here, there are two options, ***play()***, which plays the movie once, or ***loop()***, which loops it continuously.

```
movie.loop();
```

Step 4. Read frame from movie.

Again, this is identical to capture. We can either check to see if a new frame is available, or use a callback function.

```
void draw() {
  if (movie.available()) {
    movie.read();
  }
}
```

Or:

```
void movieEvent(Movie movie) {
  movie.read();
}
```

Step 5. Display the movie.

```
image(movie,0,0);
```

Example 16-4 shows the program all put together.

Example 16-4: Display QuickTime movie

```
import processing.video.*;

Movie movie; // Step 1. Declare Movie object

void setup() {
  size(200,200);
  // Step 2. Initialize Movie object
  movie = new Movie(this, "testmovie.mov"); // Movie file should be in data folder
  // Step 3. Start movie playing
  movie.loop();
}

// Step 4. Read new frames from movie
void movieEvent(Movie movie) {
  movie.read();
}

void draw() {
  // Step 5. Display movie.
  image(movie,0,0);
}
```

Although *Processing* is by no means the most sophisticated environment for dealing with displaying and manipulating recorded video (and it should be noted that performance with large video files will tend to be fairly sluggish), there are some more advanced features available in the video library. There are functions for obtaining the duration (length measured in seconds) of a video, for speeding it up and slowing it down, and for jumping to a specific point in the video (among others).

Following is an example that makes use of *the* ***jump()*** (jump to a specific point in the video) and ***duration()*** (returns the length of movie) functions.

Example 16-5: Scrubbing forward and backward in movie

```
// If mouseX is 0, go to beginning
// If mouseX is width, go to end
// And everything else scrub in between

import processing.video.*;

Movie movie;

void setup() {
  size(200,200);
  background(0);
  movie = new Movie(this, "testmovie.mov");
}

void draw() {
  // Ratio of mouse X over width
  float ratio = mouseX / (float) width;
  // Jump to place in movie based on duration
  movie.jump(ratio*movie.duration());
  movie.read();       // read frame
  image(movie,0,0);  // display frame
}
```

The ***jump()*** function allows you to jump immediately to a point of time within the video. ***duration()*** returns the total length of the movie in seconds.

*Exercise 16-3: Using the **speed()** method in the Movie class, write a program where the user can control the playback speed of a movie with the mouse. Note **speed()** takes one argument and multiplies the movie playback rate by that value. Multiplying by 0.5 will cause the movie to play half as fast, by 2, twice as fast, by −2, twice as fast in reverse, and so on.*

16.4 Software Mirrors

With small video cameras attached to more and more personal computers, developing software that manipulates an image in real-time is becoming increasingly popular. These types of applications are sometimes referred to as "mirrors," as they provide a digital reflection of a viewer's image. *Processing*'s extensive library of functions for graphics and its ability to capture from a camera in real-time make it an excellent environment for prototyping and experimenting with software mirrors.

As we saw earlier in this chapter, we can apply basic image processing techniques to video images, reading and replacing the pixels one by one. Taking this idea one step further, we can read the pixels and apply the colors to shapes drawn onscreen.

We will begin with an example that captures a video at 80 × 60 pixels and renders it on a 640 × 480 window. For each pixel in the video, we will draw a rectangle 8 pixels wide and 8 pixels tall.

Let's first just write the program that displays the grid of rectangles. See Figure 16.4.

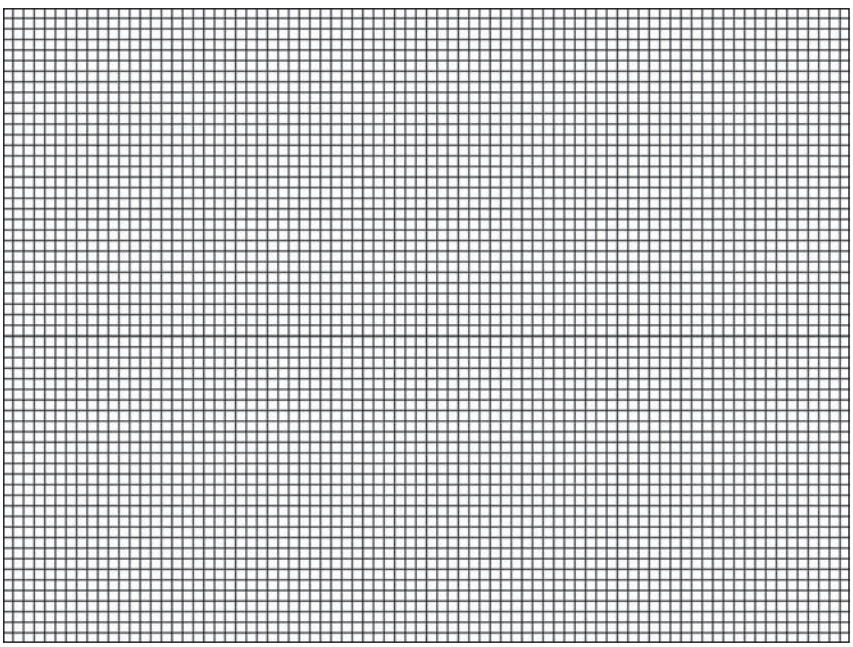


fig. 16.4

Example 16-6: Drawing a grid of 8 × 8 squares

```
// Size of each cell in the grid, ratio of window size to video size
int videoScale = 8;
// Number of columns and rows in our system
int cols, rows;

void setup(){
  size(640,480);
  // Initialize columns and rows
  cols = width/videoScale;
  rows = height/videoScale;
}

void draw(){
  // Begin loop for columns
  for (int i = 0; i < cols; i++) {
    // Begin loop for rows
    for (int j = 0; j < rows; j++) {
      // Scaling up to draw a rectangle at (x,y)
      int x = i*videoScale;
      int y = j*videoScale;
      fill(255);
      stroke(0);
      rect(x,y,videoScale,videoScale);
    }
  }
}
```

The videoScale variable tells us the ratio of the window's pixel size to the grid's size.

80 * 8 = 640
60 * 8 = 480

For every column and row, a rectangle is drawn at an (*x,y*) location scaled and sized by videoScale.

Knowing that we want to have squares 8 pixels wide by 8 pixels high, we can calculate the number of columns as the width divided by eight and the number of rows as the height divided by eight.

- 640/8 = 80 columns
- 480/8 = 60 rows

We can now capture a video image that is 80 × 60. This is useful because capturing a 640 × 480 video from a camera can be slow compared to 80 × 60. We will only want to capture the color information at the resolution required for our sketch.

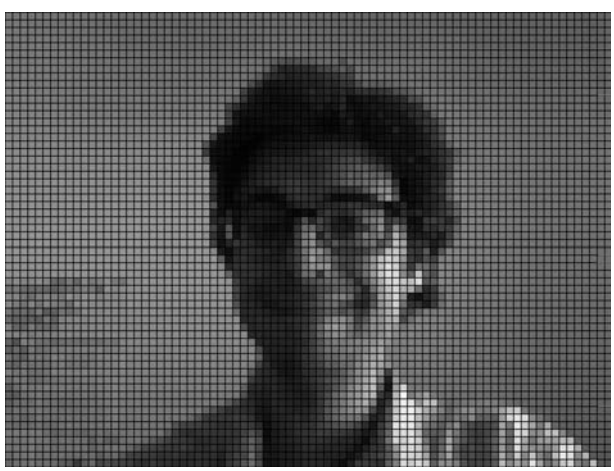


fig. 16.5

For every square at column *i* and row *j*, we look up the color at pixel (*i*,*j*) in the video image and color it accordingly. See Example 16-7 (new parts in bold).

Example 16-7: Video pixelation

```
// Size of each cell in the grid, ratio of window size to video size
int videoScale = 8;
// Number of columns and rows in our system
int cols, rows;
// Variable to hold onto Capture object
Capture video;

void setup() {
  size(640,480);
  // Initialize columns and rows
  cols = width/videoScale;
  rows = height/videoScale;
  background(0);
  video = new Capture(this,cols,rows,30);
}

void draw() {
  // Read image from the camera
  if (video.available()) {
    video.read();
  }

  video.loadPixels();
  // Begin loop for columns
  for (int i = 0; i < cols; i++) {
    // Begin loop for rows
    for (int j = 0; j < rows; j++) {
      // Where are we, pixel-wise?
      int x = i*videoScale;
      int y = j*videoScale;

      // Looking up the appropriate color in the pixel array
      color c = video.pixels[i+j*video.width];

      fill(c);
      stroke(0);
      rect(x,y,videoScale,videoScale);
    }
  }
}
```

The color for each square is pulled from the Capture object's pixel array.

As you can see, expanding the simple grid system to include colors from video only requires a few additions. We have to declare and initialize the Capture object, read from it, and pull colors from the pixel array.

Less literal mappings of pixel colors to shapes in the grid can also be applied. In the following example, only the colors black and white are used. Squares are larger where brighter pixels in the video appear, and smaller for darker pixels. See Figure 16.6.

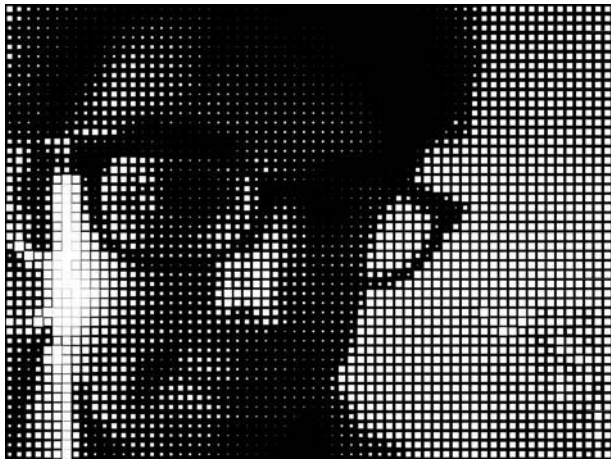


fig. 16.6

Example 16-8: Brightness mirror

```
// Each pixel from the video source is drawn as a
// rectangle with size based on brightness.

import processing.video.*;

// Size of each cell in the grid
int videoScale = 10;
// Number of columns and rows in our system
int cols, rows;
// Variable for capture device
Capture video;

void setup() {
  size(640,480);
  // Initialize columns and rows
  cols = width/videoScale;
  rows = height/videoScale;
  smooth();
  // Construct the Capture object
  video = new Capture(this,cols,rows,15);
}

void draw() {
  if (video.available()) {
    video.read();
  }
  background(0);

video.loadPixels();
  // Begin loop for columns
  for (int i = 0; i < cols; i++) {
  // Begin loop for rows
     for (int j = 0; j < rows; j++) {
       // Where are we, pixel-wise?
       int x = i*videoScale;
       int y = j*videoScale;
       // Reversing x to mirror the image
       int loc = (video.width - i - 1) + j*video.width;
```

In order to mirror the image, the column is reversed with the following formula:

mirrored column = width – column – 1

```
      // Each rect is colored white with a size determined by brightness
      color c = video.pixels[loc];
      float sz = (brightness(c)/255.0)*videoScale;
      rectMode(CENTER);
      fill(255);
      noStroke();
      rect(x + videoScale/2,y+videoScale/2,sz,sz);
      }
    }
}
```

A rectangle size is calculated as a function of the pixel's brightness. A bright pixel is a large rectangle, and a dark pixel is a small one.

It is often useful to think of developing software mirrors in two steps. This will also help you think beyond the more obvious mapping of pixels to shapes on a grid.

Step 1. Develop an interesting pattern that covers an entire window.
Step 2. Use a video's pixels as a look-up table for coloring that pattern.

For example, say for Step 1, we write a program that scribbles a random line around the window. Here is our algorithm, written in pseudocode.

- Start with *x* and *y* as coordinates at the center of the screen.
- Repeat forever the following:
  —Pick a new *x* and *y* (staying within the edges).
  —Draw a line from the old (*x*,*y*) to the new (*x*,*y*).
  —Save the new (*x*,*y*).

Example 16-9: The scribbler

```
// Two global variables
float x;
float y;

void setup() {
  size(320,240);
  smooth();
  background(255);
  // Start x and y in the center
  x = width/2;
  y = height/2;
}

void draw() {

  // Pick a new x and y
  float newx = constrain(x + random(-20,20),0,width);
  float newy = constrain(y + random(-20,20),0,height);

  // Draw a line from x,y to the newx,newy
  stroke(0);
  strokeWeight(4);
  line(x,y,newx,newy);

  // Save newx, newy in x,y
  x = newx;
  y = newy;
}
```

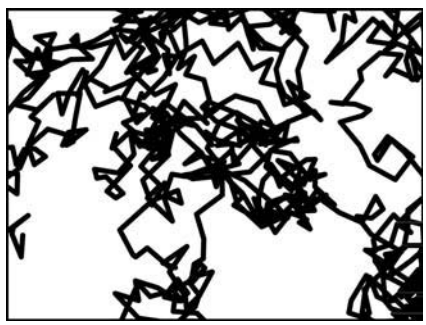


fig. 16.7

A new *x,y* location is picked as the current (*x,y*) plus or minus a random value. The new location is constrained within the window's pixels.

We save the new location in (*x,y*) in order to start the process over again.

Now that we have finished our pattern generating sketch, we can change ***stroke()*** to set a color according to the video image. Note again the new lines of code added in bold in Example 16-10.

Example 16-10: The scribbler mirror

```
import processing.video.*;

// Two global variables
float x;
float y;

// Variable to hold onto Capture object
Capture video;

void setup() {
  size(320,240);
  smooth();
  // framerate(30);
  background(0);
  // Start x and y in the center
  x = width/2;
  y = height/2;
  // Start the capture process
  video = new Capture(this,width,height,15);
}

void draw() {

  // Read image from the camera
  if (video.available()) {
    video.read();
  }
  video.loadPixels();

  // Pick a new x and y
  float newx = constrain(x + random(-20,20),0,width-1);
  float newy = constrain(y + random(-20,20),0,height-1);

  // Find the midpoint of the line
  int midx = int((newx + x) / 2);
  int midy = int((newy + y) / 2);
  // Pick the color from the video, reversing x
  color c =  video.pixels[(width-1-midx) + midy*video.width];

  // Draw a line from x,y to the newx,newy
  stroke(c);
  strokeWeight(4);
  line(x,y,newx,newy);

  // Save newx, newy in x,y
  x = newx;
  y = newy;
}
```

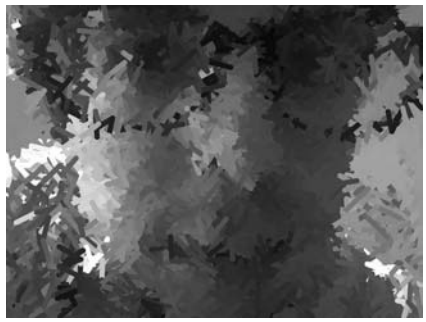


fig. 16.8

If the window were larger (say 800 × 600), we might want to introduce a videoScale variable so that we do not have to capture such a large image.

The color for the scribbler is pulled from a pixel in the video image.

Exercise 16-4: Create your own software mirror using the methodology from Examples 16-9 and 16-1 0. Create your system without the video first and then incorporate using the video to determine colors, behaviors, and so on.

16.5 Video as Sensor, Computer Vision

Every example in this chapter has treated the video camera as a data source data for digital imagery displayed onscreen. This section will provide a simple introduction to things you can do with a video camera when you do not display the image, that is, "computer vision." Computer vision is a scientific field of research dedicated to machines that *see*, using the camera as a sensor.

In order to better understand the inner workings of computer vision algorithms, we will write all of the code ourselves on a pixel by pixel level. However, to explore these topics further, you might consider downloading some of the third party computer vision libraries that are available for *Processing*. Many of these libraries have advanced features beyond what will be covered in this chapter. A brief overview of the libraries will be offered at the end of this section.

Let's begin with a simple example.

The video camera is our friend because it provides a ton of information. A 320 × 240 image is 76,800 pixels! What if we were to boil down all of those pixels into one number: the overall brightness of a room? This could be accomplished with a one dollar light sensor (or "photocell"), but as an exercise we will make our webcam do it.

We have seen in other examples that the brightness value of an individual pixel can be retrieved with the ***brightness()*** function, which returns a floating point number between 0 and 255. The following line of code retrieves the brightness for the first pixel in the video image.

```
float brightness = brightness(video.pixels[0]);
```

We can then compute the overall (i.e., average) brightness by adding up all the brightness values and dividing by the total number of pixels.

```
video.loadPixels();
// Start with a total of 0
float totalBrightness = 0;
// Sum the brightness of each pixel
for (int i = 0; i < video.pixels.length; i++) {
  color c = video.pixels[i];
  totalBrightness += brightness(c);
}

// Compute the average
float averageBrightness = totalBrightness / video.pixels.length;
// Display the background as average brightness
background(averageBrightness);
```

Sum all brightness values.

Average brightness = total brightness / total pixels

Before you start to cheer too vigorously from this accomplishment, while this example is an excellent demonstration of an algorithm that analyzes data provided by a video source, it does not begin to harness the power of what one can "see" with a video camera. After all, a video image is not just a collection of colors, but it is a collection of spatially oriented colors. By developing algorithms that search through the pixels and recognize patterns, we can start to develop more advanced computer vision applications.

Tracking the brightest color is a good first step. Imagine a dark room with a single moving light source. With the techniques we will learn, that light source could replace the mouse as a form of interaction. Yes, you are on your way to playing Pong with a flashlight.

First, we will examine how to search through an image and find the *x*,*y* location of the brightest pixel. The strategy we will employ is to loop through all the pixels, looking for the "world record" brightest pixel (using the ***brightness()*** function). Initially, the world record will be held by the first pixel. As other pixels beat that record, they will become the world record holder. At the end of the loop, whichever pixel is the current record holder gets the "Brightest Pixel of the Image" award.

Here is the code:

```
// The record is 0 when we first start
float worldRecord = 0.0;
// Which pixel will win the prize?
int xRecordHolder = 0;
int yRecordHolder = 0;

for (int x = 0; x < video.width; x++) {
  for (int y = 0; y < video.height; y++) {
    // What is current brightness
    int loc = x+y*video.width;
    float currentBrightness = brightness(video.pixels[loc]);
    if (currentBrightness > worldRecord) {
      // Set a new record
      worldRecord = currentBrightness;
      // This pixel holds the record!
      xRecordHolder = x;
      yRecordHolder = y;
    }
  }
}
```

When we find the new brightest pixel, we must save the (*x,y*) location of that pixel in the array so that we can access it later.

A natural extension of this example would be to track a specific color, rather than simply the brightest. For example, we could look for the most "red" or the most "blue" in a video image. In order to perform this type of analysis, we will need to develop a methodology for comparing colors. Let's create two colors, *c*1 and *c*2.

```
color c1 = color(255,100,50);
color c2 = color(150,255,0);
```

Colors can only be compared in terms of their red, green, and blue components, so we must first separate out these values.

```
float r1 = red(c1);
float g1 = green(c1);
float b1 = blue(c1);
float r2 = red(c2);
float g2 = green(c2);
float b2 = blue(c2);
```

Now, we are ready to compare the colors. One strategy is to take the sum of the absolute value of the differences. That is a mouthful, but it is really fairly simple. Take *r*1 minus *r*2. Since we only care about the magnitude of the difference, not whether it is positive or negative, take the absolute value (the positive version of the number). Do this for green and blue and add them all together.

```
float diff = abs(r1-r2) + abs(g1-g2) + abs(b1-b2);
```

While this is perfectly adequate (and a fast calculation at that), a more accurate way to compute the difference between colors is to take the "distance" between colors. OK, so you may be thinking: "Um, seriously? How can a color be far away or close to another color?" Well, we know the distance between *two points* is calculated via the Pythagorean Theorem. We can think of color as a point in three-dimensional space, only instead of *x*, *y*, and *z*, we have *r*, *g*, and *b*. If two colors are near each other in this color space, they are similar; if they are far, they are different.

```
float diff = dist(r1,g1,b1,r2,g2,b2);
```

Although more accurate, because the ***dist()*** function involves a square root in its calculation, it is slower than the absolute value of the difference method. One way around this is to write your own color distance function without the square root. colorDistance = $(r1 - r2)*(r1 - r2) + (g1 - g2)*(g1 - g2) + (b1 - b2)*(b1 - b2)$

Looking for the "most red" pixel in an image, for example, is therefore looking for the color "closest" to red—(255,0,0).

By adjusting the brightness tracking code to look for the closest pixel to any given color (rather than the brightest), we can put together a color tracking sketch. In the following example, the user can click the mouse on a color in the image to be tracked. A black circle will appear at the location that most closely matches that color. See Figure 16.9.

Example 16-11: Simple color tracking

```
import processing.video.*;

// Variable for capture device
Capture video;
color trackColor;

void setup() {
  size(320,240);
  video = new Capture(this,width,height,15);
  // Start off tracking for red
  trackColor = color(255,0,0);
  smooth();
}
```

A variable for the color we are searching for.

fig. 16.9

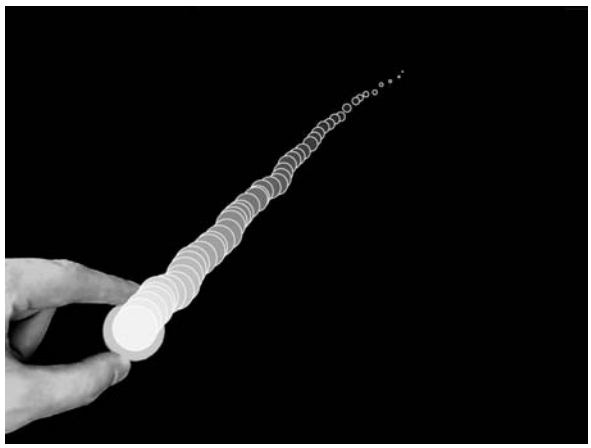
```
void draw() {
  // Capture and display the video
  if (video.available()) {
    video.read();
  }
  video.loadPixels();
  image(video,0,0);


  // Closest record, we start with a high number
  float worldRecord = 500;
  // XY coordinate of closest color
  int closestX = 0;
  int closestY = 0;
  // Begin loop to walk through every pixel
  for (int x = 0; x < video.width; x++) {
    for (int y = 0; y < video.height; y++) {
      int loc = x + y*video.width;
      // What is current color
      color currentColor = video.pixels[loc];
      float r1 = red(currentColor);
      float g1 = green(currentColor);
      float b1 = blue(currentColor);
      float r2 = red(trackColor);
      float g2 = green(trackColor);
      float b2 = blue(trackColor);
      // Using euclidean distance to compare colors
      float d = dist(r1,g1,b1,r2,g2,b2);

      // If current color is more similar to tracked color than
      // closest color, save current location and current difference
      if (d < worldRecord) {
        worldRecord = d;
        closestX = x;
        closestY = y;
      }
    }
  }


  if (worldRecord < 10) {
    // Draw a circle at the tracked pixel
    fill(trackColor);
    strokeWeight(4.0);
    stroke(0);
    ellipse(closestX,closestY,16,16);
  }
}


void mousePressed() {
  // Save color where the mouse is clicked in trackColor variable
  int loc = mouseX + mouseY*video.width;
  trackColor = video.pixels[loc];
}
```

Before we begin searching, the "world record" for closest color is set to a high number that is easy for the first pixel to beat.

We are using the ***dist()*** function to compare the current color with the color we are tracking.

We only consider the color found if its color distance is less than 10. This threshold of 10 is arbitrary and you can adjust this number depending on how accurate you require the tracking to be.

Exercise 16-5: Take any Processing *sketch you previously created that involves mouse interaction and replace the mouse with color tracking. Create an environment for the camera that is simple and high contrast. For example, point the camera at a black tabletop with a small white object. Control your sketch with the object's, location. The picture shown illustrates the example that controls the "snake" (Example 9-9) with a tracked bottlecap.*



16.6 Background Removal

The distance comparison for color proves useful in other computer vision algorithms as well, such as background removal. Let's say you wanted to show a video of you dancing the hula, only you did not want to be dancing in your office where you happen to be, but at the beach with waves crashing behind you. Background removal is a technique that allows you to remove the background of an image (your office) and replace it with any pixels you like (the beach), while leaving the foreground (you dancing) intact.

Here is our algorithm.

- Memorize a background image.
- Check every pixel in the current video frame. If it is very different from the corresponding pixel in the background image, it is a foreground pixel. If not, it is a background pixel. Display only foreground pixels.

To demonstrate the above algorithm, we will perform a reverse green screen. The sketch will remove the background from an image and replace it with green pixels.

Step one is "memorizing" the background. The background is essentially a snapshot from the video. Since the video image changes over time, we must save a copy of a frame of video in a separate PImage object.

```
PImage backgroundImage;

void setup() {
  backgroundImage = createImage(video.width,video.height,RGB);
}
```

When backgroundImage is created, it is a blank image, with the same dimensions as the video. It is not particularly useful in this form, so we need to copy an image from the camera into the background image when we want to memorize the background. Let's do this when the mouse is pressed.

```
void mousePressed() {
  // Copying the current frame of video into the backgroundImage object
  // Note copy takes 5 arguments:
  // The source image
  // x,y,width, and height of region to be copied from the source
  // x,y,width, and height of copy destination
  backgroundImage.copy(video,0,0,video.width,video.height,0,0,video.width,video.height);
  backgroundImage.updatePixels();
}
```

copy( ) allows you to copy pixels from one image to another. Note that ***updatePixels( )*** should be called after new pixels are copied!

Once we have the background image saved, we can loop through all the pixels in the current frame and compare them to the background using the distance calculation. For any given pixel (x,y), we use the following code:

```
int loc = x + y*video.width;                // Step 1, what is the 1D pixel location
color fgColor = video.pixels[loc];          // Step 2, what is the foreground color
color bgColor = backgroundImage.pixels[loc];// Step 3, what is the background color

// Step 4, compare the foreground and background color
float r1 = red(fgColor); float g1 = green(fgColor); float b1 = blue(fgColor);
float r2 = red(bgColor); float g2 = green(bgColor); float b2 = blue(bgColor);
float diff = dist(r1,g1,b1,r2,g2,b2);

// Step 5, Is the foreground color different from the background color
if (diff > threshold) {
  // If so, display the foreground color
  pixels[loc] = fgColor;
} else {
  // If not, display green
  pixels[loc] = color(0,255,0);
}
```

The above code assumes a variable named "threshold." The lower the threshold, the *easier* it is for a pixel to be in the foreground. It does not have to be very different from the background pixel. Here is the full example with threshold as a global variable.

Example 16-12: Simple background removal

```
// Click the mouse to memorize a current background image

import processing.video.*;

// Variable for capture device
Capture video;
// Saved background
PImage backgroundImage;
// How different must a pixel be to be a foreground pixel
float threshold = 20;
```

fig. 16.10

```
void setup() {
  size(320,240);
  video = new Capture(this, width, height, 30);
  // Create an empty image the same size as the video
  backgroundImage = createImage(video.width,video.height,RGB);
}

void draw() {
  // Capture video
  if (video.available()) {
    video.read();
  }

  loadPixels();
  video.loadPixels();
  backgroundImage.loadPixels();

// Draw the video image on the background
image(video,0,0);
// Begin loop to walk through every pixel
for (int x = 0; x < video.width; x++) {
  for (int y = 0; y < video.height; y++) {
    int loc = x + y*video.width; // Step 1, what is the 1D pixel location
    color fgColor = video.pixels[loc]; // Step 2, what is the foreground color
    // Step 3, what is the background color
    color bgColor = backgroundImage.pixels[loc];
    // Step 4, compare the foreground and background color
    float r1 = red(fgColor);
    float g1 = green(fgColor);
    float b1 = blue(fgColor);
    float r2 = red(bgColor);
    float g2 = green(bgColor);
    float b2 = blue(bgColor);
    float diff = dist(r1,g1,b1,r2,g2,b2);
    // Step 5, Is the foreground color different from the background color
    if (diff > threshold) {
      // If so, display the foreground color
      pixels[loc] = fgColor;
      } else {
      // If not, display green
      pixels[loc] = color(0,255,0);
    }
  }
}
 updatePixels();
}

void mousePressed() {
  // Copying the current frame of video into the backgroundImage object
  // Note copy takes 5 arguments:
  // The source image
  // x,y,width, and height of region to be copied from the source
  // x,y,width, and height of copy destination
  backgroundImage.copy(video,0,0,video.width,video.height,0,0,video.width,video.
  height);
  backgroundImage.updatePixels();
}
```

We are looking at the video's pixels, the memorized backgroundImage's pixels, as well as accessing the display pixels. So we must ***loadPixels()*** for all!

We could choose to replace the background pixels with something other than the color green!

When you ultimately get to running this example, step out of the frame, click the mouse to memorize the background without you in it, and then step back into the frame; you will see the result as seen in Figure 16.10.

If this sketch does not seem to work for you at all, check and see what "automatic" features are enabled on your camera. For example, if your camera is set to automatically adjust brightness or white balance, you have a problem. Even though the background image is memorized, once the entire image becomes brighter or changes hue, this sketch will think all the pixels have changed and are therefore part of the foreground! For best results, disable all automatic features on your camera.

Exercise 16-6: Instead of replacing the background with green pixels, replace it with another image. What values work well for threshold and what values do not work at all? Try controlling the threshold variable with the mouse.

16.7 Motion Detection

Today is a happy day. Why? Because all of the work we did to learn how to remove the background from a video gets us motion detection for free. In the background removal example, we examined each pixel's relationship to a stored background image. Motion in a video image occurs when a pixel color differs greatly from what it used to be one frame earlier. In other words, motion detection is exactly the same algorithm, only instead of saving a background image once, we save the previous frame of video constantly!

The following example is identical to the background removal example with only one important change—*the previous frame of video is always saved whenever a new frame is available.*

```
// Capture video
if (video.available()) {
  // Save previous frame for motion detection!!
  prevFrame.copy(video,0,0,video.width,video.height,0,0,video.width,video.height);
  video.read();
}
```

(The colors displayed are also changed to black and white and some of the variable names are different, but these are trivial changes.)

Example 16-13: Simple motion detection

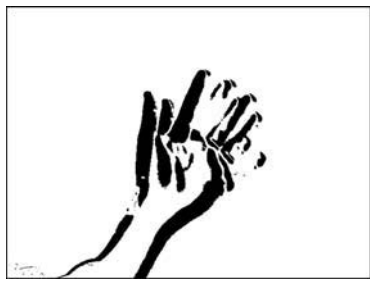


fig. 16.11

```
import processing.video.*;

// Variable for capture device
Capture video;
// Previous Frame
PImage prevFrame;
// How different must a pixel be to be a "motion" pixel
float threshold = 50;

void setup() {
  size(320,240);
  video = new Capture(this, width, height, 30);
  // Create an empty image the same size as the video
  prevFrame = createImage(video.width,video.height,RGB);
}

void draw() {
  // Capture video
  if (video.available()) {
    // Save previous frame for motion detection!!
    prevFrame.copy(video,0,0,video.width,video.height,0,0,video.width,video.height);
    prevFrame.updatePixels();
    video.read();
  }

  loadPixels();
  video.loadPixels();
  prevFrame.loadPixels();

  // Begin loop to walk through every pixel
  for (int x = 0; x < video.width; x++) {
    for (int y = 0; y < video.height; y++) {
      int loc = x + y*video.width; // Step 1, what is the 1D pixel location
      color current = video.pixels[loc]; // Step 2, what is the current color
      color previous = prevFrame.pixels[loc]; // Step 3, what is the previous color
      // Step 4, compare colors (previous vs. current)
      float r1 = red(current); float g1 = green(current); float b1 = blue(current);
      float r2 = red(previous); float g2 = green(previous); float b2 = blue(previous);
      float diff = dist(r1,g1,b1,r2,g2,b2);
      // Step 5, How different are the colors?
      if (diff > threshold) {
        // If motion, display black
        pixels[loc] = color(0);
      } else {
        // If not, display white
        pixels[loc] = color(255);
      }
    }
  }
  updatePixels();
}
```

Before we read the new frame, we always save the previous frame for comparison!

If the color at that pixel has changed, then there is "motion" at that pixel.

What if we want to just know the "overall" motion in a room? At the start of section 16.5, we calculated the average brightness of an image by taking the sum of each pixel's brightness and dividing it by the total number of pixels.

Average Brightness = Total Brightness / Total Number of Pixels

We can calculate the average motion the same way:

Average Motion = Total Motion / Total Number of Pixels

The following example displays a circle that changes color and size based on the average amount of motion. Note again that you do not need to *display* the video in order to analyze it!

Example 16-14: Overall motion

```
import processing.video.*;

// Variable for capture device
Capture video;
// Previous Frame
PImage prevFrame;
// How different must a pixel be to be a "motion" pixel
float threshold = 50;

void setup() {
  size(320,240);
  // Using the default capture device
  video = new Capture(this, width, height, 15);
  // Create an empty image the same size as the video
  prevFrame = createImage(video.width,video.height,RGB);
}

void draw() {
  background(0);

  // If you want to display the videoY
  // You don't need to display it to analyze it!
  image(video,0,0);

  // Capture video
  if (video.available()) {
    // Save previous frame for motion detection!!
    prevFrame.copy(video,0,0,video.width,video.height,0,0,video.width,video.height);
    prevFrame.updatePixels();
    video.read();
  }

  loadPixels();
  video.loadPixels();
  prevFrame.loadPixels();

  // Begin loop to walk through every pixel
  // Start with a total of 0
  float totalMotion = 0;
  // Sum the brightness of each pixel
  for (int i = 0; i < video.pixels.length; i++) {
    color current = video.pixels[i];
    // Step 2, what is the current color
    color previous = prevFrame.pixels[i];
    // Step 3, what is the previous color
```

```
    // Step 4, compare colors (previous vs. current)
    float r1 = red(current); float g1 = green(current);
    float b1 = blue(current);
    float r2 = red(previous); float g2 = green(previous);
    float b2 = blue(previous);

    float diff = dist(r1,g1,b1,r2,g2,b2);

    totalMotion += diff;
  }

  float avgMotion = totalMotion / video.pixels.length;

  // Draw a circle based on average motion
  smooth();
  noStroke();
  fill(100+avgMotion*3,100,100);
  float r = avgMotion*2;
  ellipse(width/2,height/2,r,r);
}
```

Motion for an individual pixel is the difference between the previous color and current color.

totalMotion is the sum of all color differences.

averageMotion is total motion divided by the number of pixels analyzed.

Exercise 16-7: Create a sketch that looks for the average location of motion. Can you have an ellipse follow your waving hand?

16.8 Computer Vision Libraries

There are several computer vision libraries already available for *Processing* (and there will inevitably be more). The nice thing about writing your own computer vision code is that you can control the vision algorithm at the lowest level, performing an analysis that conforms precisely to your needs. The benefit to using a third party library is that since there has been a great deal of research in solving common computer vision problems (detecting edges, blobs, motion, tracking color, etc.), you do not need to do all of the hard work yourself! Here is a brief overview of three libraries currently available.

***JMyron (WebCamXtra) by Josh Nimoy* et al.**
http://webcamxtra.sourceforge.net/

One advantage of using JMyron is its freedom from needing a vdig on Windows. It also includes many built-in functions to perform some of the tasks explained in this chapter: motion detection and color tracking. It will also search for groups of similar pixels (known as blobs or globs).

LibCV by Karsten Schmidt
http://toxi.co.uk/p5/libcv/

Much like JMyron, LibCV does not require QuickTime or WinVDIG for Windows machines. Instead of using native code, however, it uses the Java Media Framework (JMF) to connect to and capture images from a digital video camera. LibCV also includes functions not available in some of the other computer vision libraries, such as "background learning, background subtraction, difference images, and keystoning (perspective correction)."

BlobDetection by Julien "v3ga" Gachadoat
http://www.v3ga.net/processing/BlobDetection/

This library, as made obvious by its name, is specifically designed for detecting blobs in an image. Blobs are defined as areas of pixels whose brightness is above or below a certain threshold. The library takes any image as input and returns an array of Blob objects, each of which can tell you about its edge points and bounding box.

16.9 The Sandbox

Up until now, every single sketch we have created in this book could be published online. Perhaps you have already got a web site full of your *Processing* work. Once we start working with a video camera, however, we run into a problem. There are certain security restrictions with web applets and this is something we will see here and there throughout the rest of the book. A web applet, for example, cannot connect to a video camera on a user's computer. For the most part, applets are not allowed to connect to any local devices.

It makes sense that there are security requirements. If there weren't, a programmer could make an applet that connects to your hard drive and deletes all your files, send an e-mail out to all of his or her friends and say: "Check out this really cool link!" Applications do not have security requirements. After all, you can go and download applications that erase and reformat hard drives. But downloading and installing an application is different from just popping on a URL and loading an applet. There is an assumed level of trust with applications.

Incidentally, whether or not a feature of *Processing* will work in a web applet is listed on every single reference page under "Usage." If it says "Web" it can be used in a web applet.

If you must publish a camera connecting a *Processing* sketch online, there are ways around this and I will offer some tips and suggestions on this book's website: *http://www.learningprocssing.com/sandbox/*.