Appendix E

Optimization

Optimization is making changes to a program so that it will run faster. This can provide tremendous benefit by increasing the number of frames displayed per second or by allowing more to be drawn to the screen each frame. Increasing the speed can also make a program more responsive to mouse and keyboard input.

Code should usually not be optimized until a late stage in a program's development. Energy diverted to optimization detracts from refining the concept and aesthetic considerations of the software. Optimization can be very rewarding because of increased performance, but such technical details should not be allowed to distract you from the ideas. There are a few important heuristics to guide the process:

Work slowly and carefully. It's easy to introduce new bugs when optimizing, so work with small pieces of code at a time. Always keep the original version of the code. You may want to comment out the old version of a function and keep it present in your program while you work on its optimization.

Optimize the code that's used most. The majority of code in a program gets used very little, so make sure that you're focusing on a section that needs work. Advanced programmers can use a *profiler* for this task—a tool that identifies the amount of time being spent in different sections of code. Profilers are too specific to be covered here, but books and online references cover profiling Java code, and this methodology can be applied to Processing programs.

If the optimization doesn't help, revert to the original code. Lots of things seem like they'll improve performance but actually don't (or they don't improve things as much as hoped). The "optimized" version of the code will usually be more difficult to read—so if the benefits aren't sufficient, the clearer version is better.

There are many techniques to optimize programs; some that are particularly relevant to Processing sketches are listed below.

Bit-shifting color data

The `red()`, `green()`, `blue()`, and `alpha()` functions are easy to use and understand, but because they take the `colorMode()` setting into account, using them is much slower than making direct operations on the numbers. With the default color mode, the same numerical results can be achieved with greater speed by using the `>>` (right shift) operator to isolate the components and then use the bit mask `0xFF` to remove any unwanted data. These operators are explained in Appendix D (p. 669). The following example shows how to shift color data to isolate each component.

```
color c = color(204, 153, 102, 255);
float r = (c >> 16) & 0xFF;  // Faster version of red(c)
float g = (c >> 8) & 0xFF;   // Faster version of green(c)
float b = c & 0xFF;          // Faster version of blue(c)
float a = (c >> 24) & 0xFF;  // Faster version of alpha(c)
println(r + ", " + g + ", " + b + ", " + a);
```

E-01

Each component of a color is 8 bits. The bits in a pixel are ordered like this . . .

```
AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB
```

. . . where the A's are alpha bits, R's are red, G's are green, and B's are blue. After the red component is shifted 16 to the right (the `>> 16` above), the pixel values looks like this:

```
00000000000000000AAAAAAAARRRRRRRR
```

The hexadecimal value `0xFF` is 8 bits all set to 1 (equivalent to 255). The bitwise AND operator is applied to remove the other bits and just save the 8 bits needed for the color component itself:

```
  00000000000000000AAAAAAAARRRRRRRR
& 00000000000000000000000011111111
----------------------------------
  00000000000000000000000RRRRRRRR
```

This calculation completely isolates the red component of the color. To put a color "back together," use the following code:

```
int a = 255;
int r = 102;
int g = 51;
int b = 255;
color c = (a << 24) | (r << 16) | (g << 8) | b;
```

E-02

The following code is useful when the alpha setting is opaque, because no shift is needed:

```
color c = 0xFF000000 | (r << 16) | (g << 8) | b;
```

The hex digits 0xFF000000 (p. 671) are equivalent to 255 shifted to the left by 24. Hex digits can be used to replace the red and green components as well.

Bit shifting is much faster than using the `color()` method, because it ignores the `colorMode()` setting. For the same reason, specifying colors using hex notation (e.g., `#FFCC00`) has zero overhead.

Avoid creating objects in `draw()`

Creating an object slows a program down. When possible, create the objects within `setup()` so they are created only once within the program. For example, load all images and create objects within `setup()`. The following two examples show common misunderstandings about creating objects that slow programs down.

```
// AVOID loading an image within draw(), it is slow                    E-03
void draw()  {
  PImage img = loadImage("tower.jpg");
  image(img, 0, 0);
}
```

```
// AVOID creating an array inside draw(), it is slow                   E-04
void draw()  {
  int[] values = new int[200];
  // Do something with the array here
}
```

In this case, the array will be re-created and destroyed on each trip through the `draw()` method, which is extremely wasteful. The programs 43-07 (p. 404) and 44-05 (p. 417) show faster ways of creating objects.

Using the `pixels[]` array

The `get()` and `set()` functions are easy to use, but they are not as fast as accessing and setting the pixels of an image directly through the `pixels[]` array (p. 356). The following examples show four different ways of accessing the data within the `pixels[]` array, each faster than the previous one.

```
// Converts (x,y) coordinates into a position in the pixels[] array    E-05
loadPixels();
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    pixels[y*height + x] = color(102);
  }
}
updatePixels();
```

```
// Replaces the multiplication y*height with an addition               E-06
int offset = 0;
loadPixels();
for (int y = 0; y < height; y++) {
```

```
  for (int x = 0; x < width; x++) {                    E-06
    pixels[offset + x] = color(102);                   cont.
  }
  offset += width; // Avoids the multiply
}
updatePixels();
```

```
// Avoid the calculation y*height+width                E-07
int index = 0;
loadPixels();
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    pixels[index++] = color(102);
  }
}
updatePixels();
```

```
// Avoids (x,y) coordinates                            E-08
int wh = width*height;
loadPixels();
for (int index = 0; index < wh; index++) {
    pixels[index] = color(102);
}
updatePixels();
```

```
// Only calculate the color once                       E-09
int wh = width*height;
color c = color(102);
loadPixels();
for (int index = 0; index < wh; index++) {
    pixels[index] = c;
}
updatePixels();
```

When manipulating `pixels[]`, use the `loadPixels()` and `updatePixels()` functions only once within `draw()`. When possible, use `color()` outside of loops. It is not very fast because it must take into account the current color mode.

Tips for working with arrays

Adding one value at a time to an array is slower than doubling the size of the array when it's full. If a program will be continually adding data to the end of an array, use the `expand()` function once each time the array fills up in place of running `append()`

many times. Code 33-19 (p. 309) shows how to manage a growing array with `expand()`.

The `arraycopy()` function is the fastest way of copying data from one array to another. Copying the data from one array to another inside a `for` structure is much slower when copying data from large arrays. The `arraycopy()` function is demonstrated in code 33-20 (p. 310). Arrays are also much faster (sometimes 2×) than the Java classes ArrayList and Vector.

Avoid repeating calculations

If the same calculation is made more than once, it's faster to make the calculation once and store it in a variable. Instead of writing . . .

```
float x = (width/2) * 4;
float y = (width/2) * 8;
```

. . . save the result of the division in a variable and substitute it for the calculation:

```
float half = width/2;
float x = half * 4;
float y = half * 8;
```

Multiplications and divisions from inside a `for` structure can slow a program down significantly, especially if there are more than 10,000 iterations. When possible, make these calculations outside of the structure. This is demonstrated above in code E-08.

Because of the way computers make calculations, addition is faster than multiplication and multiplication is faster than division. Multiplication can often be converted to addition by restructuring the program. For example, compare the difference in run time between code E-05 and code E-06.

Lookup tables

The idea behind a lookup table is that it is faster to make reference to a value stored within a data structure than to make the calculation. One example is making calculations for `sin()` and `cos()` at each frame. These numbers can be generated once within `setup()` and stored within an array so they may be quickly retrieved. The following example shows how it's done.

E-10

```
int res = 16;                          // Number of data elements
float[] x = new float[res];            // Create x-coordinate array
float[] y = new float[res];            // Create y-coordinate array

void setup() {
  size(100, 100);
```

E-10 cont.

```
  for (int i = 0; i < res; i++) {
    x[i] = cos(PI/res * i);             // Sets x-coordinates
    y[i] = sin(PI/res * i);             // Sets y-coordinates
  }
}

void draw() {
  for (int i = 0; i < res; i++) {       // Access each point
    point(50 + x[i]*40, 50 + y[i]*40);  // Draws point on a curve
  }
}
```

You can change the resolution of the values by altering the length of the array.

Optimizers beware!

Optimized code can sometimes be more difficult to read. When deciding whether to optimize, balance the need for speed against the value of legibility. For example, in the bit-shifting example presented above, the expression

```
red(color)
```

is more clear than its optimized equivalent:

```
(color >> 16) & 0xFF
```

The name of the `red()` function clearly states its purpose, whereas the bit shift and mask are cryptic to everyone except those familiar with the technique. The confusion can be alleviated with comments, but always consider whether the optimization is more important than the simplicity of your code..