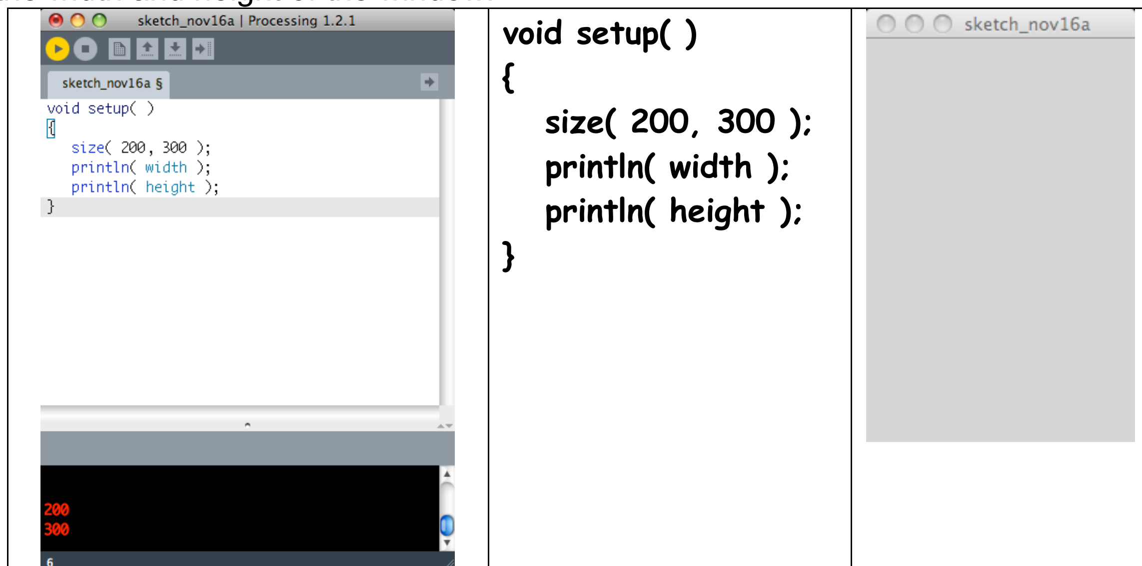
Images and the Graphics Window

Prior to beginning the work with different parts of the libraries, we spent time with classes. One reason for that was to provide some background when we started this part of the work. For the information here and for part of next week, almost everything works with classes. The first part of this will work with the window in which we have been drawing stuff since day 1. Then we will move to an image that loads a .jpg file. Then we move to video. All of these, the window, the image, and the video fall within the same family tree of classes. What we can do with the window, we can do with the image and the video. So, we are doing the same thing three times.

The "Window":

We have worked inside the "window" since day 1 when you were sent out to draw your initials. We know how to size the window and how to get the value of the width and height of the window:



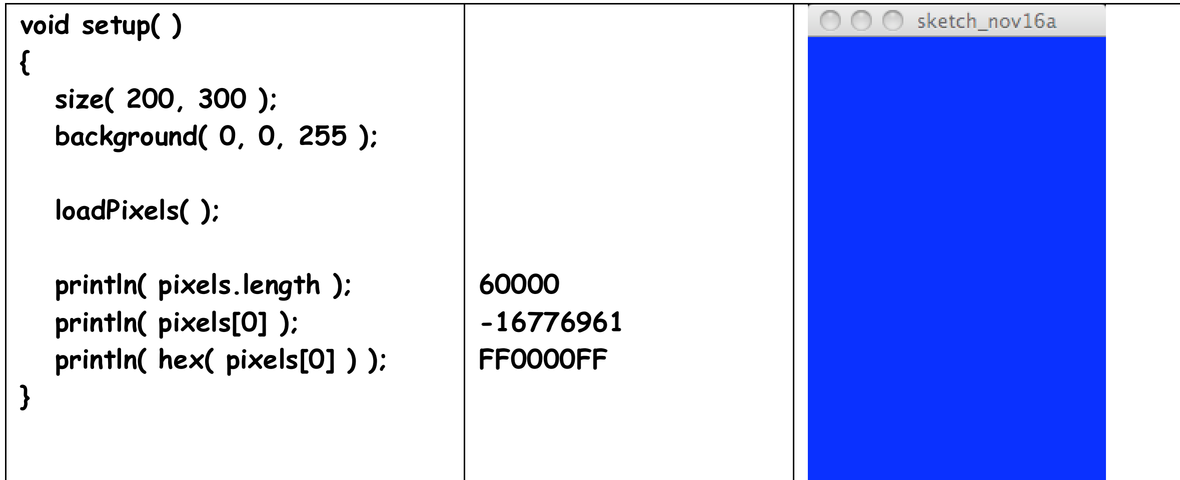
The window has two fields or variables named width and height. Their default values are 100 but we can set them to other values by calling the size( ) function. Nothing new here…

Each element of the window is called a pixel (contraction of picture element [1]). The window declared above is 200 pixels wide and 300 pixels high so it has 60,000 pixels(200 X 300).
Somewhere in the memory of the computer is a sequence of transistors delegated to store information about each pixels. Such areas are called buffers – in this case, the video buffer. This buffer must be large enough to store the data for 60,000 pixels. The data for each pixel is one int value. This int value represents the alpha [2], red, blue, and green values for the pixel. To determine the red, green, and blue value, the int value can be converted to its

[1] So.. where does the 'x' come from???
[2] This is the amount of translucence of the pixel.

Copyright Jim Roberts November 2012 Pittsburgh, PA 15221 All Rights Reserved

corresponding hex value: The example below shows these values for a blue window:




```
void setup( )
{
   size( 200, 300 );
   background( 0, 0, 255 );

   loadPixels( );

   println( pixels.length );
   println( pixels[0] );
   println( hex( pixels[0] ) );
}
```

```
60000
-16776961
FF0000FF
```

Since the entire window is blue, each pixel has the values for blue. The computer's memory must store the value for blue for each pixel. We can get a copy of this memory by calling the loadPixels( ) function. This copies the color value in The computer's memory for every pixel in the window into the array named pixels. Then we can print any value in the array using the [ ] array notation. In the code above, we print the value of the zeroth element of pixels. What we see is a mysterious int value(-16776961). However, when we convert this to the equivalent hex value things look more familiar. We see 8 hex digits. If we group them in sets of two, we see:

FF 00 00 FF

Which are the values of the alpha, red, green, and blue for the pixel in the upper left corner of the window. The base 10 equivalent of FF is 255. So the settings for the pixel in the upper left corner are:

- **alpha == 255 or opaque**
- **red == 0 or no red**
- **green == 0 or no green**
- **blue == 255 for all on (100% blue)**

Note the length of the pixel array – it has 60,000 elements.
Note that altering the array pixels does not directly alter the memory in the video buffer. There is a way we can do that – but it comes later.

So we now know that the window has three variables:

width height pixels

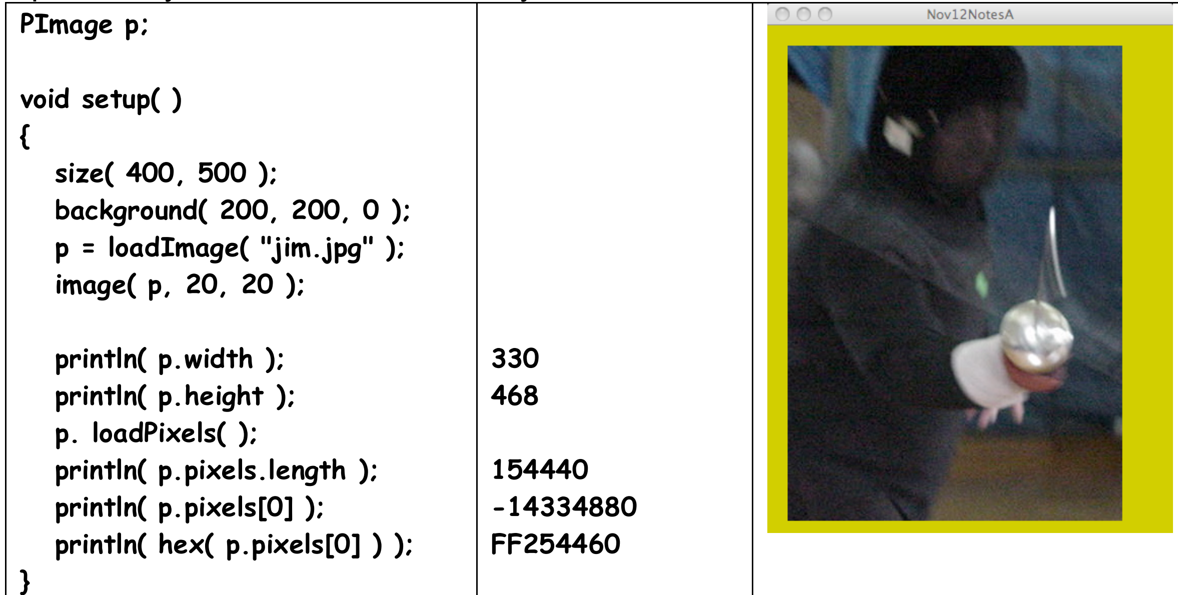
and it has at least two functions that can work with those variables:

size( ) loadPixels( )

Copyright Jim Roberts November 2012 Pittsburgh, PA 15221 All Rights Reserved

The “Image”:

We know how to create a PImage reference and reference it to a PImage object. We also know how to display the image it references: Since PImage and the window are in the same family tree of classes, we can do the same thing with a PImage that we did to the window. We can access the PImage’s width and length and we can get values stored in the image buffer copied into the image’s pixel array. This code is essentially the same code that we looked at above:



Code	Output
`PImage p;`	
`void setup( )`	
`{`	
`   size( 400, 500 );`	
`   background( 200, 200, 0 );`	
`   p = loadImage( "jim.jpg" );`	
`   image( p, 20, 20 );`	
`   println( p.width );`	330
`   println( p.height );`	468
`   p. loadPixels( );`	
`   println( p.pixels.length );`	154440
`   println( p.pixels[0] );`	-14334880
`   println( hex( p.pixels[0] ) );`	FF254460
`}`	

Notice the following:

- The length of the array is the same as the product of the width and length:
  330 X 468 == 154440
- The functions that are called are the same BUT they have the period or dot syntax that shows possession. We want the width and height of p so we use p.----- as the syntax.

Suppose we want to alter the actual data in the image (the window’s video buffer)? Altering the pixel array does not directly alter the buffer. Remember, the array is a copy of what is in the computer’s memory. If we want to do this, we use the function:

```
p.updatePixels( );
```

which copies the contents of the pixel array into the buffer.

Another thing to notice is the size of the array p.pixels. It is 154440 elements long. Each element contains an int. In Processing, a single int of information stored in memory requires 32 transistors or 32 bits (binary digits) or 4 bytes (1 byte is composed of 8 bits). This means that the image in this example requires

4 X 154440 bytes of memory or **617760** bytes of memory.

Copyright Jim Roberts November 2012 Pittsburgh, PA 15221 All Rights Reserved

Yet, if we look at the information displayed in the folder that show the size of the image file we see:

Name	Dat…fied	Da…ted	Size ▼	Kind
▼ data	Today	Today	68 KB	Folder
jim.jpg	8/20/06	8/…06	68 KB	JPEG image
Nov12NotesA.pde	Today	Today	4 KB	Proce…ce File

The file requires 68,000 bytes of space on the disk. Why is the array about 10 times larger?

617,760 vs 68,000

The reason is compression. The file is a jpg file – pronounced jay-peg. This type of files is compressed. The compression allows us to store very large files in smaller spaces and to transfer them (mail, fetch, . . .) much quicker. It is the case that the compression used for .jpg files does cause some loss of information. The rule is fairly simple. The more you compress the image (the smaller you make the .jpg file) the more of the original information you lose.

Converting an (x, y) coordinate of the image into an array index for the pixels array:

Shiffman does a very nice job explaining this in the book. See pages 262 through 264. The important thing to remember is that any (x, y) coordinate in the window or image can be converted to an index of the pixels array for the widow or image with this arithmetic:

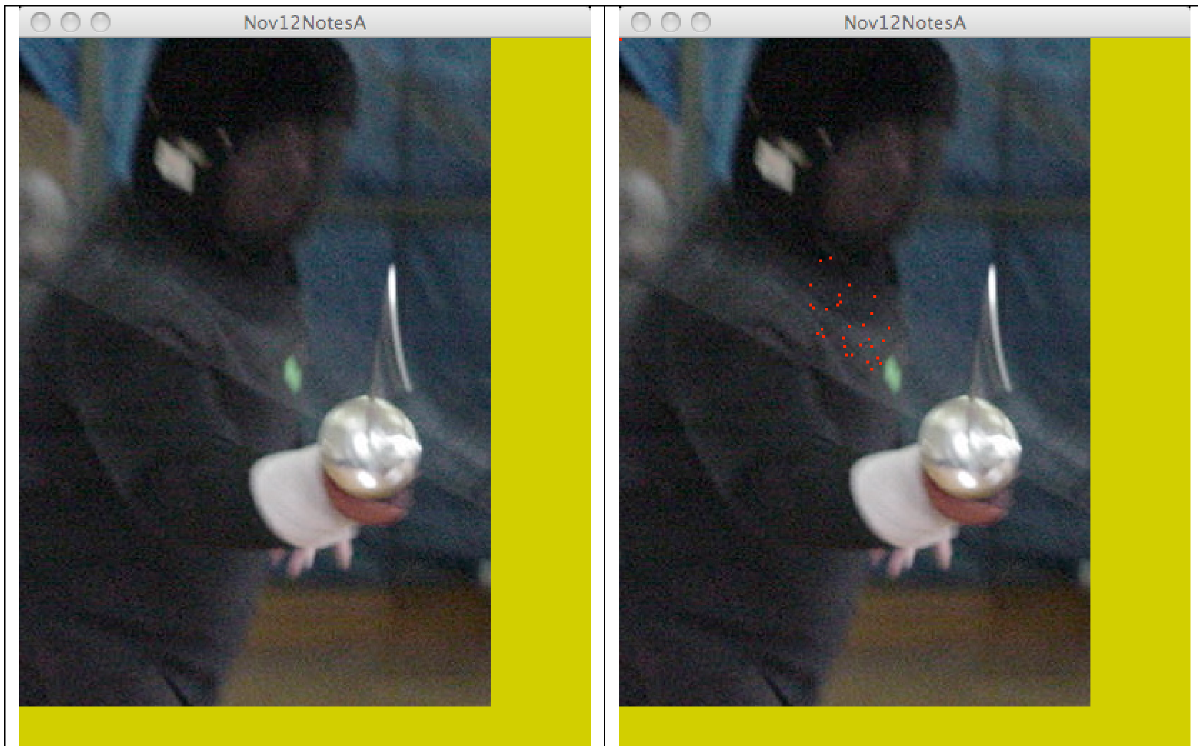
```
int index =  ( y *  width )   + x  ; // for the window
int index =  ( y *  p.width ) + x  ; // for the image
```

BUT – if you are using the variables mouseX and mouseY to determine the (x, y) coordinates, you MUST anchor the upper corner of the image at the (0, 0) position of the window. Assuming the image is anchored at the (0, 0) position, this code will use the mouse location to determine the index:

```
int index =  (mouseY *  p.width ) + mouseX  ;  // for the image
```

Here is how we can use this. The following shows Jim before and after a fencing match:

Copyright Jim Roberts November 2012 Pittsburgh, PA 15221 All Rights Reserved



Here is the code that produced the changed image on the right:

```
void draw( )
{
   image( p, 0, 0 );
}

void mousePressed( )
{
   p.loadPixels( );
   int index = (mouseY * p.width ) + mouseX;
   p.pixels[ index ] = color( 255, 0, 0 );
   p.pixels[ index + 1 ] = color( 255, 0, 0 );
   p.pixels[ index + p.width]  = color( 255, 0, 0 );
   p.pixels[ index + p.width + 1 ] = color( 255, 0, 0 );
   p.updatePixels( );
}
```

This code uses the mouse location to find the index of the pixel that was clicked and change four pixels to red. The pixels that are changed are :

- the pixel clicked

  `p.pixels[ index ] = color( 255, 0, 0 );`

- *the pixel that is the right neighbor*

  `p.pixels[ index + 1 ] = color( 255, 0, 0 );`

Copyright Jim Roberts November 2012 Pittsburgh, PA 15221 All Rights Reserved

- the pixel immediately below the pixel that was clicked,
  ```
  p.pixels[ index + p.width]  = color( 255, 0, 0 );
  ```
- its right neighbor
  ```
  p.pixels[ index + p.width + 1 ] = color( 255, 0, 0 );
  ```

Notice the purple code. This is how we "get to the next row." By adding the value of the width of the image, we move to the array element that has the color for the pixel that is directly beneath the pixel represented by the [index] element. This is important – you should work this out on paper if you do not understand this.

You should note that in this example, the image has been moved up and to the left so it is anchored at the (0, 0) coordinate of the window.

Shiffman's examples in the book in chapter 15 work with these ideas. If you are interested in exploring images, you should work with his examples. The course web page has a link to the text's web site where you can download code for all of the examples in the book.

Copyright Jim Roberts November 2012 Pittsburgh, PA 15221 All Rights Reserved