Processing

A Tutorial Workshop originally aimed at ITP Students, supplementary to course material for "Introduction to Computational Media", "Programming for Non-Programmers", and "Code and Me"

Language: English , Japanese , Korean

Workshop Teacher
Josh Nimoy
contact: jn429 [at] nyu [dot] edu
website

Software Creators
Processing is an open project initiated by Ben Fry and Casey Reas

Japanese Translation by
Hironobu Fujiyoshi and Ayako Takabatake
contact: hf [at] cs [dot] chubu [dot] ac [dot] jp

Korean Translation by
Koo-Chul Lee
contact: kclee [at] phya [dot] snu [dot] ac [dot] kr

Description

Processing is a context for exploring the emerging conceptual space enabled by electronic media. It is an environment for learning the fundamentals of computer programming within the context of the electronic arts and it is an electronic sketchbook for developing ideas.
www.Proce55ing.net

The Processing environment is the easiest Java compiler / interactive graphics and multimedia programming environment known to man. The system can be used to produce locally run pieces, as well as web-embeddable Java applets. Quite deliberately, the system is also designed to bridge the gap between educational graphics programming environments, and "real Java." Processing can be used like training wheels, but does not have to be.

The goal of this tutorial is to introduce users of Macromedia Flash and Director to the Processing environment by comparing and contrasting the systems. The theory is that the knowledge gained from these Macromedia tools can easily transfer, reducing the amount of required teaching. It assumes you have a basic understanding of either Macromedia product. By the end of this tutorial, you should be able to produce and publish your own Processing (Java) pieces, and communicate through a serial port with a BX-24 chip.

Table of Contents

Introduction
Obtaining the Processing Software
A tour of the interface
Lower Level Media Manipulation
Syntax Structure
Static 2D Drawing
Time and Motion
Mouse & Keyboard
Presentation / Exporting
Drawing Image Files
3D Form
Pixels
Typography
Serial
The Future

Introduction

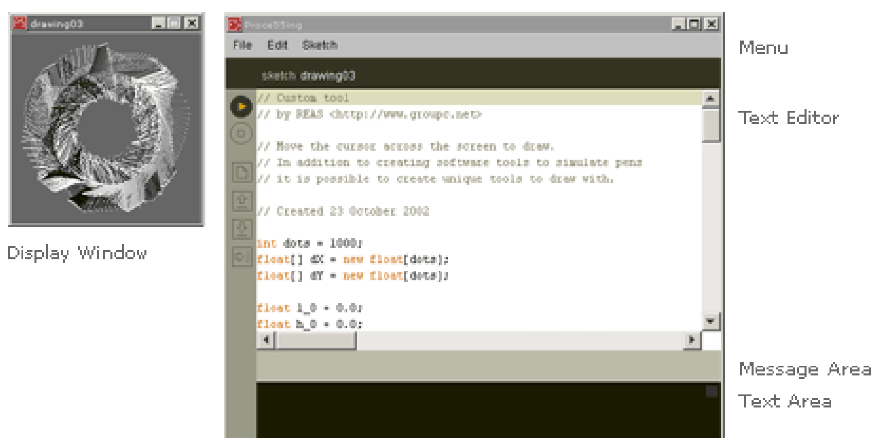
Currently in on-screen interaction design classes, the dominant teaching vehicle has been Flash or Director. Students are beginning to produce geometrically dynamic and more algorithmically complex pieces influenced by works done in environments other than their own. At ITP, I witnessed an experiment in one such class (although since then, they've been using Processing). In the middle of teaching Director Lingo, a one-week excursion into Java programing introduced students to a language other than Lingo - in hopes that they would get a more diverse view of different programming systems. A template was given to them - and they simply changed the code with a quick rosetta stone reference. After this week of confusion, several students were left with an empty yearning to learn more Java. There was no easy way to tell them that the average college Java course usually has you working in a text console, and has little or no relationship to "Applet Graphics" unless it is a course specifically targeted to teach that. In the following lesson, I hope to bridge this academic gap with the help of the Processing environment. It is not meant as a replacement to those Java courses; it is a supplement which takes care of the logistics - without getting deep into syntax nuances. Also, Processing and Java are not being presented as the next level beyond Macromedia, nor are they being presented as a lower level system. They are simply an alternative, capable of doing different things in different ways. If you are currently attending a Java course, it may be possible to use Processing for your assignments, depending on how flexible your instructor is. This tutorial is a mixture of my own writing and images - with those found on one state of the Processing website, maintained by Casey Reas and Ben Fry. You will get to know these two names if you are a Processing user.

Obtaining the Processing Software

Processing is free (free as in free beer, free as in free speech, free as in free country), and still in development stages. It will continue to be free even after it is finished. The software is currently in ALPHA phase, which is the thing that comes before BETA. Bugs are being fixed, and features are being added. In order to download the cross-platform install program you can email its developers to join the testing community. On the Processing website, click **Download** for further instructions. Additionally, there is an avid messaging system amongst the testers. It is highly recommended that you create a login for yourself. This community is the best way to receive help on any topic - from other testers, from the authors themselves, and from lurking geeks such as the author of this tutorial. It is also important to note that this is the online community that helps develop Processing by discussing features in those forums. On the Processing website, click **Discourse**. It is also important to note that the Processing software and website are constantly being updated. Check back for new additions to the reference, and new versions of the software. Right now is an exciting time!

A tour of the interface

The following image is taken from www.Proce55ing.net. To see it in context, click **Reference**, then **Environment**.

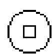


Immediately, you are probably thinking "Wow, this is such a simple interface. How could it possibly be as capable as Director or Flash?" Both Director and Flash have all

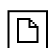
kinds of import and media editing interfaces based on common tasks in commercial multimedia. In Processing, all of this is either done using another program, or done by programming in Java. For example, Flash provides its own mini-Illustrator, while Director provides its own mini-Photoshop. In consequence, a large body of work done in both pieces of software have resembled the restrictions of their integrated editors. In Processing (and in Java), you provide your own list of vector paths or GIF files, and you render them using programming. You are free to generate your own forms and structures using the language to control the pixels on the screen more directly. For those experimental people who aim to produce new forms independently or ahead of the status quo and its automation tools, Processing can be more convenient.

Here is an introduction to the six buttons on the left side of the window.

The **play** button is the same as in Director and Flash. Press this to see your code execute as a program.



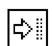
The **stop** button is the same as in Director and Flash. Press this to see your program stop executing.



Creates a **new** file. Processing calls them *sketches*. You can even call them applets, programs, or interactive pieces. Director and Flash call them *movies*.

Opens a pre-existing sketch. A menu will pop up and you can choose from your own collection, saved in the special Processing sketch folder which I will show you later. You can also choose from a wide variety of example sketches by famous new media designer/artists, in order to learn from them and use them as a code reference.

Saves the current sketch into the Processing sketches folder. If you want to give the sketch a name other than the current date, you can choose *save As* from the *File* menu.



Exports the current sketch into the Processing sketches folder - this time as a Java applet - complete with its very own HTML file. This feature will be covered in more depth.

For detailed and advanced information about the Processing environment, see the Processing Environment reference.

Lower Level Media Manipulation

In Director, one imports or creates media into a *cast*, then drag it onto a *stage* where it will exist as a *sprite*. In Flash, one also imports or creates media into a *library*, then instances them as *movieclips* on a similar stage. In Processing (and in Java) this media importing is all done in code, similar to the way HTML works. Additionally, any custom media that you invent (vector systems, DNA data, color samples from the film, Fargo) can all be embedded as part of the Java code. In fact, you are not restricted to having any external images or sounds if you want to keep everything in one tidy file - because the pixels of an image can also be converted to be part of your code, and sound data can also be stored as a large array of data. The benefit of having a library or cast is for better control over formatting, in order to save disk space / memory, and in order to add specific point-&-click features onto a common file system metaphor. The benefit of a sprite or movieclip is so that a visible, tangible object can sit on the screen and be an easy way for people to make buttons, videogame characters, individual graphic elements, and other visual, controllable, positive space elements. However, in creating cooperative groups of elements, and objects that are neither positive nor negative space - this metaphor has become a burden to some. In Processing, this complex layer does not exist; there is only mouse/keyboard/serial events in conjunction with basic drawing routines. One takes care of redrawing the scene repeatedly, in response to input changes and time. In a

matter of speaking, it is your responsibility to write your own sprite or movieclip system, but you are not required to. Invent another metaphor that would be more useful to you as an artist. It also affords the opportunity to deeply diversify the aesthetic from those works we often spot as Macromedia-influenced.

In the coming sections within this guide, I introduce methods for rendering imagery to the screen. Then I will introduce time and animation. Finally, I will show you how to add interaction with the mouse, keyboard, and serial port. These are the basic building blocks for anything you wanted to do in high-level tools. You will be capable of doing them in Java if you put your mind to constructing them yourself.

Syntax Structure

For those of you who use FlashMX, this is a review. The following is called **structure00** in the example sketches.

```
// Statements & Comments
// by REAS

// Statements are the elements that make up programs.
// The ";" is used to end statements. It is called the "statement terminator."
// Comments are used for making notes to help people better understand
programs.
// A comment begins with two forward slashes ("//").

// Created 1 September 2002

// The size function is a statement that tells the computer
// how large to make the window.
// Each function statement has zero or more parameters.
// Parameters are data passed into the method
// and used as values for specifying what the computer will do.
size(200, 200);

// The background function is a statement that tells the computer
// which color to make the background of the window
background(102);
```

And Java variables are as follows:

```
int x = 0;
println(x);
x=x+1;
println(x);
x=x+1;
println(x);
```

Press play and see this:

```
0
1
2
3
```

Flash people: there is no such thing as **var**. For in-depth information on variables, consult the official Java language tutorial.
Here is the part about variables.

What about if-then?

```
int a = 1;
int b = 2;

if(a==b){
  println("same");
}else{
  println("different");
}
```

Press play and see this: different

Things are a bit different from Lingo when it comes to comparison. The programmer uses a single "=" to assign a variable some value. One uses a "==" (double equal) when trying to determine whether or not a number equals another number.

Additionally, "not equal to" is no longer "<>" - it is now "!=" - and the rest are the same ("<" , ">" , ">=" , and "<="). For more information on conditionals, consult the authority.

What about repeating loops?

```
for(int i=0 ; i<5 ; i++){
    println(i);
}
```

Press play and see this:

```
0
1
2
3
4
```

Lingo people, this is the same as "repeat with i = 0 to 4." Inside those parentheses, there are three special statements separated by two semicolons. The first statement creates a temporary variable. The second statement specifies a condition which allows the loop to continue looping. As soon as ***i*** is no longer smaller than 5, then the loop will stop. The third statement gives you a chance to incrementally change ***i*** however you want. "i++" is shorthand for "i = i + 1". Sun can tell you much more about for loops.

While loops are similar to the structure of an if-then.

```
while(6!=2){
    println("muhuhaha!");
}
```

Don't run this program! **:)**

If you are curious about getting heavy on the flow-control syntax, here is a broader link to the Java language tutorial, the part about flow-control.

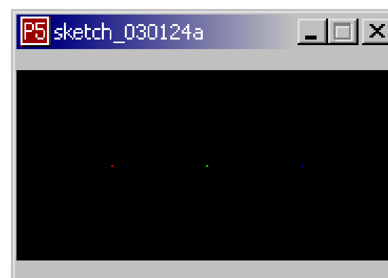
I will get into custom routines (functions) in a little while. I give you these basic introductions to the syntax - not in order to be thorough, but in order that you will understand the drawing functions that are about to follow.

For more introduction to the structures, see the Processing Language Comparison and the Processing Structure Examples.

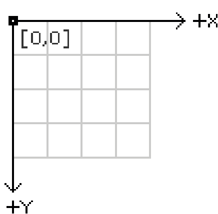
Static 2D Drawing

```
size(200,100);
background(0,0,0);
stroke(255,0,0);
point(50,50);
stroke(0,255,0);
point(100,50);
stroke(0,0,255);
point(150,50);
```

Run this code, and you should hopefully get the following image. This is a wide black window with three pixels colored red, green, and blue.



Let us break this code down, line for line.



First of all, it is necessary to know that the screen is a graph of pixels - each individually addressable by a unique (X,Y) coordinate. The origin (0,0) is at the top left corner of the rectangle. As you add to Y, you move down. As you add to X, you move to the right. It's similar to playing the boardgame, **Battleship**. This is no different from Director or Flash.

size(200,100);

Calling the **size** function lets the size of the canvas to 200 pixels high, 100 pixels wide. If you do not call this at the beginning, the default will be 100x100.

```
background(0,0,0);
```

Calling the **background** function lets you change the color of the entire canvas. In Director, this is the stage color. In Flash, this is the document's background color. 0,0,0 means black. If you never call background in Processing, then the default will be gray.

```
stroke(255,0,0);
```

Calling the **stroke** function lets you change the current drawing color so that every drawing command called after it will draw using that color. 255,0,0 means red. If you never call stroke in Processing, then the default will be black.

```
point(50,50);
```

Calling the **point** function will set the pixel at 50,50 to the current stroke color. In this case, red.

```
stroke(0,255,0);
point(100,50);
```

This code makes a green dot in the center.

```
stroke(0,0,255);
point(150,50);
```

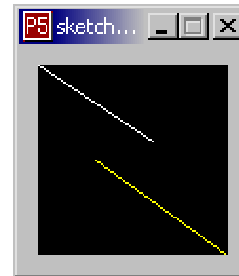
This code makes the blue dot on the right.

As you can see, this is similar to **draw( )** Image Lingo, or those drawing methods in ActionScript. You control the screen like it is a canvas that understands certain drawing operations. Let us now expand to more complex shapes.

```
background(0,0,0);
stroke(255,255,255);
line(0,0,60,40);
stroke(255,255,0);
line(30,50,100,100);
```

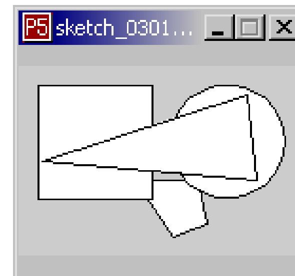
Here, I am drawing two lines. The first one is white, and the second one is yellow.

In the line function, the first two parameters are the first x,y coordinate, and the last two parameters are the second x,y coordinate. The line is drawn from the first coordinate to the second.



Now, here are some pre-fab shapes.

```
size(150,100);
quad(61,60, 94,60, 99,83, 81,90);
rect(10,10,60,60);
ellipse(80,10,60,60);
triangle(12,50, 120,15, 125,60);
```



triangle will draw a three-pointed polygon. It has six parameters. Parameters 1 and 2 are the first X,Y coordinate. Parameters 3 and 4 are the second X,Y coordinate. Parameters 5 and 6 are the third X,Y coordinate.

```
triangle(x1, y1, x2, y2, x3, y3);
```

quad will draw a four-pointed polygon. The structure of the parameters are similar to that of triangle, but this time, a fourth pair of parameters are added to specify a fourth X,Y coordinate.

```
quad(x1, y1, x2, y2, x3, y3, x4, y4);
```

rect will draw a rectangle. The first and second parameter will specify the position, while the third and fourth parameters specify the width and height.

```
rect(x, y, width, height);
```

ellipse will draw an oval. Its parameters work the same way as those in rect.

```
ellipse(x, y, width, height);
```

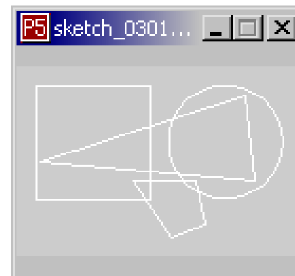
Now I will modify this program to show you something new. The new code is **marked**.

```
size(150,100);
fill(#CC6600);
stroke(#FFFFFF);
quad(61,60, 94,60, 99,83, 81,90);
rect(10,10,60,60);
ellipse(80,10,60,60);
triangle(12,50, 120,15, 125,60);
```

(Notice here, I am specifying colors in a different way than before - this time HTML style)

Fill is introduced as the cousin of stroke. Fill is what makes the polygons green, while stroke is what makes the outlines red. Fill's parameters specify a color, like stroke. The default fill is white. But what if I want no fill?

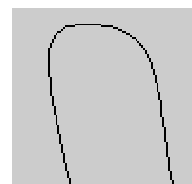
```
size(150,100);
noFill( );
stroke(#FFFFFF);
quad(61,60, 94,60, 99,83, 81,90);
rect(10,10,60,60);
ellipse(80,10,60,60);
triangle(12,50, 120,15, 125,60);
```



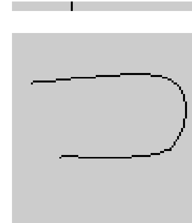
Now, you can see the quad underneath the oval because only the strokes are being drawn. Similarly, there is **noStroke**, which disables the outline from being drawn. To enable stroke or fill once more, you must call stroke or fill, specifying a color.

Drawing with curves is slightly more complex than drawing with straight lines. Specifying a curve requires providing non-visual information that helps to define the severity and direction of curvature. Processing provides both the curve( ) and bezier( ) methods.

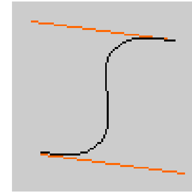
```
curve(84, 91, 68, 19, 21, 17, 32,
100);
```



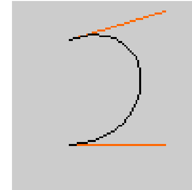
```
curve(10, 26, 83, 24, 83, 61, 25, 65);
```



```
stroke(255, 102, 0);
line(85, 20, 10, 10);
line(90, 90, 15, 80);
stroke(0, 0, 0);
bezier(85, 20, 10, 10, 90, 90, 15,
80);
```



```
stroke(255, 102, 0);
line(30, 20, 80, 5);
line(80, 75, 30, 75);
stroke(0, 0, 0);
bezier(30, 20, 80, 5, 80, 75, 30, 75);
```



```
curve(x1, y1, x2, y2, x3, y3, x4, y4);
bezier(x1, y1, x2, y2, x3, y3, x4, y4);
```

For the **curve( )** function, the first and second parameters specify the first point of the curve and the last two parameters specify the second point of the curve. The middle parameters set the points for defining the shape of the curve.

For the **bezier( )** function, the first two parameters specify the first point in the curve and the last two parameters specify the last point. The middle parameters provide the context for defining the shape of the curve.

In the **bezier( )** examples above, the orange lines reveal the hidden control points for the curves.

Although Processing has provided these quick primitives, you are still free (and encouraged) to construct your own shapes.

Using the **beginShape( )** and **endShape( )** methods are the key to creating more complex forms. **beginShape( )** begins recording vertices for a shape and **endShape( )** stops recording. The **beginShape( )** command requires a parameter to tell it which type of shape to create from the provided vertices. The parameters available for **beginShape( )** are LINES, LINE_STRIP, LINE_LOOP, TRIANGLES, TRIANGLE_STRIP, QUADS, QUAD_STRIP, and POLYGON. After giving the **beginShape( )** command, a series of **vertex( )** commands must follow. To stop drawing the shape, give the **endShape( )** command. **Vertex( )** commands with two parameters specify a position in 2D and **vertex( )** commands with three parameters specify a position in 3D. Each shape will be outlined with the current stroke color and filled with the fill color (see the Color section for more information).

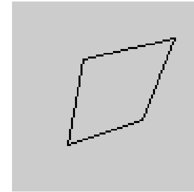
The following note is found in the Processing reference:

> Processing is only able to draw convex polygons, but we are working on the code for supporting concave polygons. For future releases there will be separate parameters for CONVEX_POLYGON and CONCAVE_POLYGON.

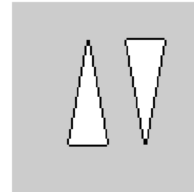
Although this is the case, you can still look up some examples on the web that will tell you how to construct anything you want from convex polygons.

Here are some examples from Proce55ing.net

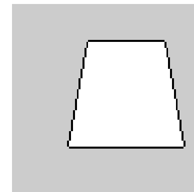
```
beginShape(LINE_LOOP);
vertex(30, 20, -50);
vertex(85, 20, 0);
vertex(85, 75, -80);
vertex(30, 75, 0);
endShape( );
```



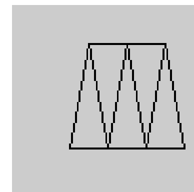
```
beginShape(TRIANGLES);
vertex(30, 75);
vertex(40, 20);
vertex(50, 75);
vertex(60, 20);
vertex(70, 75);
vertex(80, 20);
vertex(90, 75);
endShape( );
```



```
beginShape(TRIANGLE_STRIP);
vertex(30, 75);
vertex(40, 20);
vertex(50, 75);
vertex(60, 20);
vertex(70, 75);
vertex(80, 20);
vertex(90, 75);
endShape( );
```

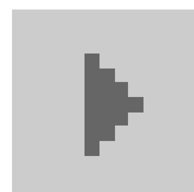


```
noFill( );
beginShape(TRIANGLE_STRIP);
vertex(30, 75);
vertex(40, 20);
vertex(50, 75);
vertex(60, 20);
vertex(70, 75);
vertex(80, 20);
vertex(90, 75);
endShape( );
```



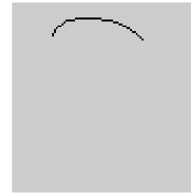
```
noStroke( );
fill(153, 153, 153);
beginShape(TRIANGLE_STRIP);
vertex(30, 75);
vertex(40, 20);
vertex(50, 75);
vertex(60, 20);
vertex(70, 75);
vertex(80, 20);
vertex(90, 75);
endShape( );
```

```
noStroke( );
fill(102);
beginShape(POLYGON);
vertex(38, 23);
vertex(46, 23);
vertex(46, 31);
vertex(54, 31);
vertex(54, 38);
vertex(61, 38);
vertex(61, 46);
vertex(69, 46);
vertex(69, 54);
vertex(61, 54);
vertex(61, 61);
vertex(54, 61);
vertex(54, 69);
vertex(46, 69);
vertex(46, 77);
```

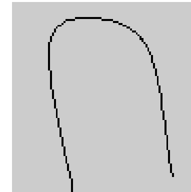


```
vertex(38, 77);
endShape( );
```

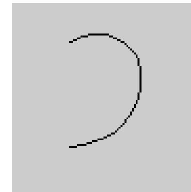
```
beginShape(LINE_STRIP);
curveVertex(84, 91);
curveVertex(68, 19);
curveVertex(21, 17);
curveVertex(32, 100);
endShape( );
```



```
beginShape(LINE_STRIP);
curveVertex(84, 91);
curveVertex(84, 91);
curveVertex(68, 19);
curveVertex(21, 17);
curveVertex(32, 100);
curveVertex(32, 100);
endShape( );
```



```
beginShape(LINE_STRIP);
bezierVertex(30, 20);
bezierVertex(80, 0);
bezierVertex(80, 75);
bezierVertex(30, 75);
endShape( );
```



For more detailed information about vector drawing, see the Processing Form Examples, the Processing Shape reference.

There is much more to draw and render to the screen, but I have only given you the 2D drawing routines so that we can cover animation and interactivity. Then we will return to the other drawing methods.

Time and Motion

In Director, there is the score. You are given a playback head, and tweening methods for sprites. Things like video, embedded Flash, QTVRs, and sound seem to animate in their own time-space. If you are working on more of a dynamic animation, you might only use one frame and code in responses to an **ExitFrame** or **PrepareFrame** event. In Flash, you are given a timeline, and a bit more complex tweening support than director. Those who choose to work completely in ActionScript commonly use two frames - one for setup and to call a looping frame, and the other to loop forever. Actionscript also allows you to respond in code in an **onClipEvent (enterFrame)**. Processing has no timeline, score, or tweening methods, unless you choose to structure your code as such. Like Lingo and Actionscript, Processing allows you to respond to a frame-progression event handler with your own drawing routine. Up until now, I have been showing you Processing code in **Basic Mode.** This mode is for drawing static images. It is merely a shopping list of visual elements. Processing has three modes of operation: basic, standard, and advanced. **Advanced Mode** is conventional Java, without training wheels. In order to begin with time and motion, we will now move forward to **Standard Mode.** If you had been clicking my tangential links in my text, you may have seen one or two standard mode Processing programs. Here is a simple example:

```
int x = 0;

void setup( ){
   noStroke( );
}

void loop( ){
   background(190);
   rect(x, 0, 5, 100);
   x=x+1;
}
```

In this example, a white rectangle moves from left to right, only once.

An animated GIF shows this motion on the right.

The optional **setup( )** section runs once when the program begins. The **loop( )** section runs forever until the program is stopped. In Lingo, **setup( )** is similar to **beginSprite** or **startMovie** - and **loop( )** is similar to **ExitFrame** or **PrepareFrame**. In Flash, **setup( )** is similar to the first frame of the animation that only executes once, then calls the loop. **Setup( )** and **loop( )** are both **functions.** You can also write your own functions for organization and encapsulation of complexity. For more information on writing custom functions in Java, see Sun's Java language tutorial - Implementing Methods section.

Once you have written the first function in Processing, that program will automatically switch to standard mode. Any statements outside of your function that are not variable initializes will no longer work when you press play. You can move this code to either setup( ) or loop( ). If you want a variable to be global (meaning that it retains its value outside the scope of the functions) then declare it at the top of the program, outside of both loop( ) and setup( ). In the example above, the variable **x** was declared global.

In Processing, the **framerate(**n**)** function can be used to slow down or speed up the entire sketch, but it is certainly possible to move things are differing speeds simply by varying the amount that you increment, or by using floats and only adding a fraction to them. For more long term and more time-precise control, Processing gives you full access to Western time measurement.

Processing has several methods for getting the date and time from your computer's clock.

year( ) // current year, i.e. 2002, 2003, etc.
month( ) // returns the current month, from 1..12
day( ) // returns the day of month, from 1..31
hour( ) // the current hour, from 0..23
minute( ) // the current minute, from 0..59
second( ) // the current second, from 0..59

A special function called **millis( )** returns the number of milliseconds (thousandths of a second) since starting the applet. This is often used for timing animation sequences.

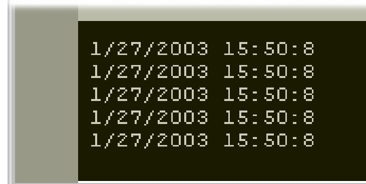
millis( ) // number of milliseconds since starting applet.

It is also possible to make your applet wait by using the delay function. Using this function can effectively adjust frame rates.

delay(40**);** // takes a nap for 40 milliseconds

```
void loop( ){
   print(month( )+"/");
   print(day( )+"/");
   print(year( )+" ");
   print(hour( )+":");
   print(minute( )+":");
   println(second( ));
}
```

Here is a simple example in which the current time and date are constantly output to the text area at the bottom of the Processing window.



This is not a very pretty example, but it's simple to understand. For some nicer (and more complex) examples of time, see Clock, by Mescobosa, and Milliseconds, by REAS. These two were done in Processing. Also, for more examples of animated motion, see the Processing Motion Examples.

Mouse & Keyboard

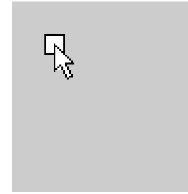
Access to the mouse and keyboard are both similar to the way Flash and Director do it. In Lingo, the mouse is addressed with **the mouseLoc**, **the mouseH**, and **the mouseV**. Additionally, there are also mouse event handlers such as **on mouseDown**. In Flash, there is **onClipEvent (mouseDown)**, etc. In Processing, the **mousePressed( )** function is called every time the mouse is pressed and the **mouseReleased( )** method is called every time the mouse if released. All you have to do is add the function to your code, just like **loop( )**.

```
void loop( ) {
   background(190);
   rect(mouseX-5, mouseY-5, 10, 10);
}

void mousePressed( ) {
   fill(0);
}
void mouseReleased( ) {
   fill(255);
}
```

In this simple example, a square is drawn where ever the mouse goes.

If you hold your mouse down, the square will turn black.

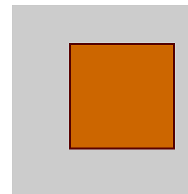


For further information on the mouse, see the Processing Mouse reference, and don't forget to check out these exquisite Processing Mouse examples.

Keyboard input is equally similar to Flash and Director.

```
void loop( ) {
   if(keyPressed) {
      fill(102, 0, 0);
   } else {
      fill(204, 102, 0);
   }
   rect(30, 20, 55, 55);
}
```

In this simple example, the square turns dark red if any keyboard key is being held down. No background redraw is needed!



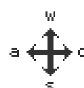
The keyboard input can also be delivered to you in the form of an event handling function.

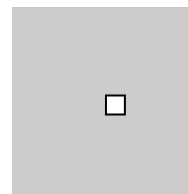
```
int x = 50;
int y = 50;

void loop( ){
   background(190);
   rect(x,y,10,10);
}

void keyPressed( ){
   if(key=='w'||key=='W'){
      y--;
   }else if(key=='s'||key=='S'){
      y++;
   }else if(key=='a'||key=='A'){
      x--;
   }else if(key=='d'||key=='D'){
      x++;
   }
}
```

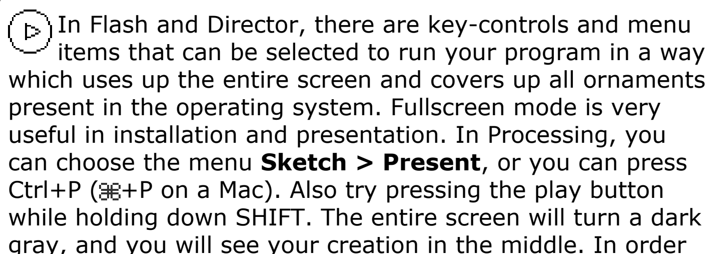
In this simple example, keys on the keyboard will move the square around.





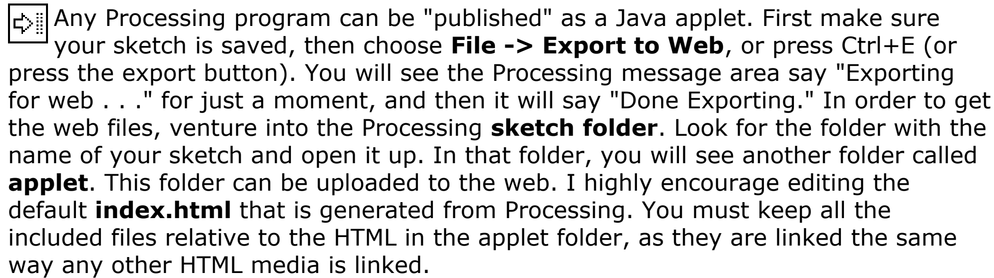
For further information on keyboard input, see the Processing Keyboard reference, and don't forget to check out these exquisite Processing Keyboard examples.

Presentation / Exporting



In Flash and Director, there are key-controls and menu items that can be selected to run your program in a way which uses up the entire screen and covers up all ornaments present in the operating system. Fullscreen mode is very useful in installation and presentation. In Processing, you can choose the menu **Sketch > Present**, or you can press Ctrl+P (⌘+P on a Mac). Also try pressing the play button while holding down SHIFT. The entire screen will turn a dark gray, and you will see your creation in the middle. In order

to return to normal, you will be able to press ESC. If that does not work, then there is a "stop" button at the bottom left corner.



Any Processing program can be "published" as a Java applet. First make sure your sketch is saved, then choose **File -> Export to Web**, or press Ctrl+E (or press the export button). You will see the Processing message area say "Exporting for web . . ." for just a moment, and then it will say "Done Exporting." In order to get the web files, venture into the Processing **sketch folder**. Look for the folder with the name of your sketch and open it up. In that folder, you will see another folder called **applet**. This folder can be uploaded to the web. I highly encourage editing the default **index.html** that is generated from Processing. You must keep all the included files relative to the HTML in the applet folder, as they are linked the same way any other HTML media is linked.

Processing Folder/
  sketchbook/
    default/
      ***your_sketch_name*/**
        applet/
          ***your_sketch_name*.java**
          ***your_sketch_name*.class**
          ***your_sketch_name*.jar**
          index.html

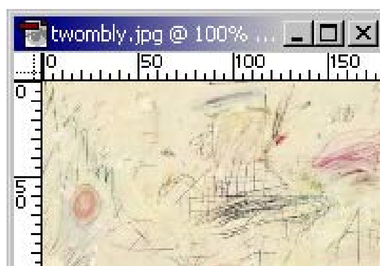
save( ) and saveFrame( )
If you need to export to non-interactive formats, it is possible to make .tif files of the Processing window by using the **saveFrame( )** function. Placing this method at the end of the **loop( )** will save the image on the screen. If **saveFrame( )** is called multiple times, it will create an image sequence as follows: screen-0001, screen-0002, screen-0003, etc. Using **save( )** will let you choose a file name. It is simple to import these images into Quicktime or other video programs to make an animated documentation of a Processing program. Although Processing has built in this easy image saving function, it is also possible to export to other formats with a bit more work. For example, here is a Processing program that exports to Adobe Illustrator.

Drawing Image Files

Getting an image onto a Processing sketch is simple. Java only accepts JPGs or GIFs (unless you want to do extra work). You can place the image into your sketch by using the file system and a couple lines of code. First, save your sketch. Then you can find the sketch file by looking in the Processing program folder. In that folder, you will see a folder called **sketchbook**. In that folder, you will probably see two folders - one called **examples** and another called **default**. In **default**, look for the folder with the same name as your sketch. Inside that folder, you will see another folder called **data**. This is the folder in which you must place your image file for easy Processing. Here is another way to say it:

Processing Folder/
  sketchbook/
    default/
      ***your_sketch_name*/**
        data/
          your_imagefile.gif

Let us assume that I have a sketch called **image_example_1** and I want to draw the following image called **twombly.jpg**, a drawing by Cy Twombly:

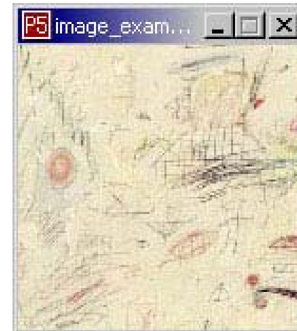


I would save it in the appropriate folder:

Processing Folder/
sketchbook/
default/
***image_example_1*/**
data/
twombly.jpg

And now I am ready to add the code.

```
size(150,150);
BImage b =
loadImage("twombly.jpg");
image(b,0,0,150,150);
```

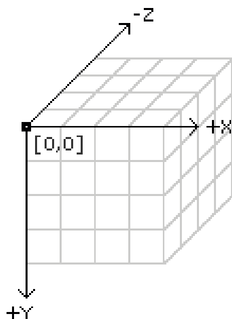


BImage is an object that will hold your loaded file until you draw it. **b** is what I chose to call this one. **image( )** is what actually draws the image to the screen.
image(BImage, x, y, width, height);
You may also choose to omit the width and height, and the image will then draw at normal scale.

Straight imported files are not the only way, as URLs also work.

For more detailed information concerning images, you may consult the Processing reference. Here is the part called Loading_and_Displaying. Building upon this knowledge, it is also possible to show sequencial images (video footage). For more excitement, see the Processing Image Examples.

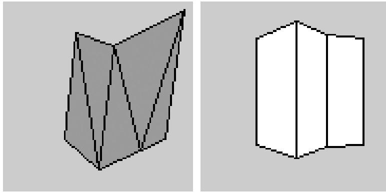
3D Form



There has been a lot of fuss concerning 3D being introduced into inherently 2D environments. In the case of Flash, numerous third party tools have been developed. In Director's case, a 3D vector graphics sprite was retrofitted very recently. The systems are so complex that a lot of people are intimidated about even starting to learn.

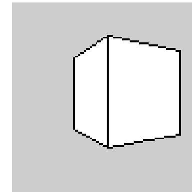
In Processing, 3D only means adding a z-axis.

vertex(x, y, z);
line(x1, y1, z1, x2, y2, z2);
bezierVertex(x, y, z);
curveVertex(x, y, z);

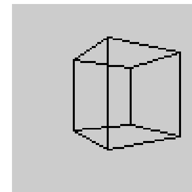


```
box(size);
box(width, height, depth);
sphere(size);
```

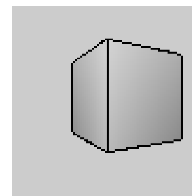
```
translate(58, 48, 0);
rotateY(0.5);
box(40);
```



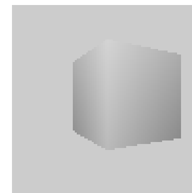
```
noFill( );
translate(58, 48, 0);
rotateY(0.5);
box(40);
```



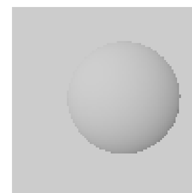
```
lights( );
translate(58, 48, 0);
rotateY(0.5);
box(40);
```



```
noStroke( );
lights( );
translate(58, 48, 0);
rotateY(0.5);
box(40);
```



```
noStroke( );
lights( );
translate(58, 48, 0);
sphere(28);
```



Note that **box** and **sphere** do not ask you to specify position coordinates! In these examples, it is necessary to use **translate** and **rotate**. There is also **scale**, and a pair of functions called **push** and **pop** which allow you to bookmark your translations in a very organized fashion. To learn the details on this useful way to organize your drawing, see the Processing Transform Reference and the Processing Transform Examples. Of course, if you do not care for these transformations, then there is always a solution.

Also note the use of **lights( )** and **noLights( )**. Using lights will render the 3D shape in a manner which suggests shading. For more information concerning lighting, see the Processing Lights Reference.

"What? that's it for 3D?"

If you think that this is not enough 3D to allow you to make interesting things, then check out all the wonderful art that has already been created at Processing Software. And this is only the beginning.

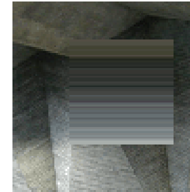
Pixels

Control over the pixels is currently far from Flash. **SetPixel** and **GetPixel** have just been added to Director (and already, a well known Director-loving interactive artist has adopted the nickname SetPixel). However, Director is quite possibly the slowest pixel addressing system you will ever work with. ITP Students attend a class taught by Danny Rozin called **The World - Pixel by Pixel**, and continue to program in C because it is the only thing fast enough for them to achieve their conceptual goals (Lingo and MAX being the only alternatives). It is popular for such ITP students to prepare for Danny's class by attending a C course. Working with Processing pixels is considerably faster than Image Lingo, and arguably less complex. Although Java does not compare to the speed of C, soon Danny's students might explore the possibility of using Processing to speed the learning curve.

```
get(x, y); // Returns an integer
set(x, y, color);
pixels[index]; // Array containing the display window
```

```
int width = 100;
int height = 100;
BImage b; // declare variable "b" of type BImage
b = loadImage("basel.gif");

image(b, 0, 0);
for (int i=30; i<(width-15); i++) {
    for(int j=20; j<(height-25); j++) {
        color here = get(30, j);
        set(i, j, here);
    }
}
```



With control over the pixels, you can also implement your very own drawing routines. For example, here is transparency. Writing the rest of the Director inks would not be so hard. Here is a dotted line function.

For more information about pixel play, see the Processing Image reference and the Processing Image Examples.

Typography

The typographic rendering system currently uses a Processing-specific font file format. The makers of the software have included a font import menu item to help you out, and even then - they have provided the Processing programmer with a wealth of fonts to choose from. Click here to see all of the currently included fonts. These fonts are stored as bitmap images. Here is a very simple example of rendering text to the sketch. Go ahead and run this program - and expect to get an error.

```
size(200,100);
background(#FFFFFF);
fill(#000000);
BFont f =
loadFont("Bodoni-Italic.vlw.gz");
textFont(f, 50);
text("handglove", 14, 60);
```

The error will say that it cannot find **Bodoni-Italic.vlw.gz**. This is because you have not yet imported the font file into your data folder. (see the images section of this tutorial for more information on your data folder). After choosing the font you like, look in the **fonts** folder within the Processing program folder. **Copy** your font of choice into your sketch's data folder, and the program should run fine after that.

BFont f = loadFont("Bodoni-Italic.vlw.gz" **);** loads that font file into the variable f.

textFont(f, size**);** sets the current font and size before drawing the text.

text("handglove", x, y**);** renders the text in place.

This example incorporates rotation, and a simple **for** loop.

```
size(200,100);
noStroke( );
BFont f =
loadFont("Univers66.vlw.gz");
textFont(f, 50);
fill(#FFFFFF);
ellipse(-50,-55,150,150);
fill(#CC6600);
for(int i=0;i<20;i++){
   rotateZ(0.2);
   text("dizzy", 90,0);
}
```

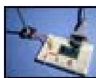
Okay. If you are not a "typography person" and do not care for any of this, there is always a way to simplify things, at the expense of control. Furthermore, if you need something like a text entry field, it's useful to write your own. Most widgets of vernacular user interface are not hard to add to a project if you think of them as small, modular, interactive exercises - rather than seeing them as some kind of thing that the operating system exclusively provides. The thing you gain is ultimate control over the design. For more vernacular interface re-inventions, see the Processing GUI Examples.

Also see the whimsical Processing typography examples. Processing comes from a group of people concerned with aesthetics and computation. New forms of typography is one of the things that their particular MIT Media Lab research group is world famous for.

Serial

Through a serial port, a computer can communicate with anything from Palm Pilots to medical equipment. It is common in electronic arts to use the serial port to talk to custom built devices. Director is able to do this with the help of third party Xtras. Flash does not support serial ports, nor does it have a third-party plugin system. Processing has serial communication built in. In this example, we will make a turning knob interact with a Processing sketch. A word of warning, however; this section is technically more advanced than the previous sections, as it requires a working knowledge of basic electronic circuitry.

This circuit uses a BX-24, which is a common prototyping integrated circuit widely used at ITP, but also at other similar places. For more information on setting up a BX-24, see Tom Igoe's Physical Computing reference as well as the lab assignments from his Physical Computing class at ITP.



Here is a picture of the circuit, excluding the +5v power supply - in order to simplify the photo. It uses a 10K potentiometer with a 1K resistor at pin 13. For details on setting this up, see the ITP Intro to BX-24.

```
sub main()
   delay 0.5
   do
      debug.print cStr(getADC(13))
      delay 0.1
   loop
end sub
```

Here is the program to download into the chip. All it does is feed the angle of the knob back to the PC. You will know it is working when you see a stream of numbers in the BasicX debug monitor.

Processing allows you to choose which serial port it will work with through the interface **Sketch -> Serial Port** submenu.

```
String buff = "";
int val = 0;

void setup() {
   beginSerial(19200);
}

void loop() {
   background(val,val,val);
}

void serialEvent() {
   if(serial!=10){
      buff += (char)serial;
   }else{
      buff =
buff.substring(0,buff.length()-1);
      val = Integer.parseInt(buff)/4;
      buff = "";
   }
}
```

This Processing program will listen to the turning knob, and change the background color between black and white.

For more information about serial, see the Processing serial reference.

The Future

Processing is a brand new work in progress, and as I said before, it is important to continue checking the website for updates. A new release can easily be installed in place of the new one, and your **sketchbook/default** folder can simply be copied from version to version. As I write this document, Processing is at version ALPHA 0050, when I came back to update the tutorial in Fall 2004, they were at ALPHA 0068. Processing might change the names of the three modes, **basic**, **standard** and **advanced** to something more fitting. At the time of ALPHA 0050, Casey was telling me it might also change its name from **Proce55ing to Processing**, and we all watched the change happen. The graphics engine has improved by adding **anti-aliased** rendering, alpha values in color (transparency factor), and the polygon fill will be improved. This is not to say that one cannot already write these things with a bit of research. Processing was written in the same language that it asks users to program in. This is different from Flash and Director, which are authored mainly in C. Amit Pitaru has created **Sonia**, a library for creating sound. As network communication is very useful, it is a prime item on the todo list. This may include downloading files from the web, but also generalized TCP/IP allowing you to talk to telnet, FTP, Gnutella, **Carnivore**, other Processing apps, and even Flash and Director programs. There is already network code floating around the Processing discussion pages, and it will probably be formally supported in the future. For camera vision, there is a library called **JMyron**. Myron is a Processing implementation of WebCamXtra, an open source camera vision Xtra for Director. In the spirit of integrated development environments, Processing hopes to add a more built-in tools such as a color picker, and a bezier editor. In addition to the software, Casey and Ben hope to expand the community. A code repository will be established as a central directory for useful pieces of Processing code. Processing is also open-source, which means that anyone can edit the Processing software itself. A proper open-source framework will be established so that anyone can download the code, recompile Processing, and then contribute the new version. There is already an accumilating page of **Processing extension libraries**.

With all this in mind, I hope that you will find Processing useful in addition to Macromedia tools that you are either learning, or have already mastered. Please publish or send any Processing examples around so that we might learn from your explorations, and do not forget to be a part of the online community by engaging in the discourse section of the Processing site. Happy creating!

Josh Nimoy was a graduate student in the Interactive Telecommunications

Benjamin Fry is a doctoral candidate at the MIT Media Laboratory. His research focuses on

Casey Reas is an Associate Professor at the newly established Interaction Design

Program at New York University's Tisch School of the Arts in 2004. He creates and exhibits interactive media work concerned with vernacular digital interactivity, nature, and experimental typography systems. Nimoy values the effects of good teaching, good communication, and honest work. He also holds a BA in Design and Media Arts from UCLA School of Arts and Architecture, specializing in digital cultures and technologies. Josh was a visiting undergraduate researcher at the MIT Media Laboratory in 1999 in the Aesthetics and Computation Group, led by John Maeda, where he worked with Ben Fry and Casey Reas.
website

methods of visualizing large amounts of data from dynamic information sources. The work uses ideas from distributed and adaptive systems to form organic representations that react and respond to the input data. This work is currently directed towards Genomic Cartography which is a study into new methods to represent the data found in the human genome. At MIT, he is a member of the Aesthetics and Computation Group, led by John Maeda. Ben received an undergraduate degree from the School of Design at Carnegie Mellon University, with a major in Graphic Design and a minor in Computer Science.
website

Institute Ivrea in northern Italy. His work explores abstractions of biological and natural systems through diverse digital media including software art, digital prints, and responsive installations. In 2001, Casey received his M.S. degree in Media Arts and Sciences from the MIT Media Laboratory, where he was a member of John Maeda's Aesthetics and Computation Group (ACG). Casey has lectured and exhibited in Europe, Asia, and the United States. His work has recently been shown at the American Museum of the Moving Image, Ars Electronica, Interaction01 in Ogaki, New York Digital Salon, Museum of Modern Art, P.S.1, and Siggraph2000.
website

---

last updated October 31, 2005