Getting Started. Welcome to Processing!

Processing is a simple programming environment that was created to make it easier to develop visually oriented applications with an emphasis on animation and providing users with instant feedback through interaction. The developers wanted a means to “sketch” ideas in code. As its capabilities have expanded over the past six years, Processing has come to be used for more advanced production-level work in addition to its sketching role. Originally built as a domain-specific extension to Java targeted towards artists and designers, Processing has evolved into a full-blown design and prototyping tool used for large-scale installation work, motion graphics, and complex data visualization.

Processing is based on Java, but because program elements in Processing are fairly simple, you can learn to use it even if you don't know any Java. If you're familiar with Java, it's best to forget that Processing has anything to do with Java for a while, until you get the hang of how the API works.

The latest version of Processing can be downloaded at *http://processing.org/download*

An important goal for the project was to make this type of programming accessible to a wider audience. For this reason, Processing is free to download, free to use, and open source. But projects developed using the Processing environment and core libraries can be used for any purpose. This model is identical to GCC, the GNU Compiler Collection. GCC and its associated libraries (e.g. libc) are open source under the GNU Public License (GPL), which stipulates that changes to the code must be made available. However, programs created with GCC (examples too numerous to mention) are not themselves required to be open source.

Processing consists of:

- The Processing Development Environment (PDE). This is the software that runs when you double-click the Processing icon. The PDE is an Integrated Development Environment (IDE) with a minimalist set of features designed as a simple introduction to programming or for testing one-off ideas.
- A collection of functions (also referred to as commands or methods) that make up the “core” programming interface, or API, as well as several libraries that support more advanced features such as drawing with OpenGL, reading XML files, and saving complex imagery in PDF format.
- A language syntax, identical to Java but with a few modifications.
- An active online community, hosted at *http://processing.org*.

For this reason, references to “Processing” can be somewhat ambiguous. Are we talking about the API, the development environment, or the web site? We'll be careful in this text when referring to each.

Sketching with Processing

A Processing program is called a *sketch*. The idea is to make Java-style programming feel more like scripting, and adopt the process of scripting to quickly write code. Sketches are stored in the *sketchbook*, a folder that's used as the default location for saving all of your projects. When you first run Processing, the sketch last used will automatically open. If this is the first time Processing is used (or if the sketch is no longer available), a new sketch will open.

Sketches that are stored in the sketchbook can be accessed from File → Sketchbook. Alternatively, File → Open... can be used to open a sketch from elsewhere on the system.

Advanced programmers need not use the PDE, and may instead choose to use its libraries with the Java environment of choice. However, if you're just getting started, it's recommended that you use the PDE for your first few projects to gain familiarity with the way things are done. While Processing is based on Java, it was never meant to be a Java IDE with training wheels. To better address our target audience, the conceptual model (how programs work, how interfaces are built, and how files are handled) is somewhat different from Java.

Hello world

The Processing equivalent of a "Hello World" program is simply to draw a line:

```
line(15, 25, 70, 90);
```

Enter this example and press the Run button, which is an icon that looks like the Play button from any audio or video device. Your code will appear in a new window, with a gray background and a black line from coordinate (15, 25) to (70, 90). The (0, 0) coordinate is the upper left-hand corner of the display window. Building on this program to change the size of the display window and set the background color, type in the code below:

```
size(400, 400);
background(192, 64, 0);
stroke(255);
line(150, 25, 270, 350);
```

This version sets the window size to 400 x 400 pixels, sets the background to an orange-red, and draws the line in white, by setting the stroke color to 255. By default, colors are specified in the range 0 to 255. Other variations of the parameters to the `stroke()` function provide alternate results:

```
stroke(255);               // sets the stroke color to white
stroke(255, 255, 255);     // identical to the line above
stroke(255, 128, 0);       // bright orange (red 255, green 128, blue 0)
stroke(#FF8000);           // bright orange as a web color
stroke(255, 128, 0, 128);  // bright orange with 50% transparency
```

The same alternatives work for the `fill()` function, which sets the fill color, and the `background()` function, which clears the display window. Like all Processing functions that affect drawing properties, the fill and stroke colors affect all geometry drawn to the screen until the next fill and stroke functions.

Advanced Topic: External editor

It's also possible to use the editor of your choice instead of the built-in editor. Simply select “Use External Editor” in the Preferences window (Processing → Preferences on Mac OS X, or File → Preferences on Windows and Linux). When using an external editor, editing will be disabled in the PDE, but the text will reload whenever you press Run.

Hello mouse

A program written as a list of statements (like the previous examples) is called a *static* mode sketch. In static mode, a series of functions are used to perform tasks or create a single image without any animation or interaction. Interactive programs are drawn as a series of frames, which you can create by adding functions titled `setup()` and `draw()` as shown in the code below. These are built-in functions that are called automatically.

```
void setup() {
  size(400, 400);
  stroke(255);
  background(192, 64, 0);
}

void draw() {
  line(150, 25, mouseX, mouseY);
}
```

The `setup()` block runs once, and the `draw()` block runs repeatedly. As such, `setup()` can be used for any initialization; in this case, setting the screen size, making the background orange, and setting the stroke color to white. The `draw()` block is used to handle animation. The `size()` function must always be the first line inside `setup()`.

Because the `background()` function is used only once, the screen will fill with lines as the mouse is moved. To draw just a single line that follows the mouse, move the `background()` function to the `draw()` function, which will clear the display window (filling it with orange) each time `draw()` runs.

```
void setup() {
  size(400, 400);
  stroke(255);
}

void draw() {
  background(192, 64, 0);
  line(150, 25, mouseX, mouseY);
}
```

Static mode programs are most commonly used for extremely simple examples, or for scripts that run in a linear fashion and then exit. For instance, a static mode program might start, draw a page to a PDF file, and exit.

Most programs will employ active mode, which use the `setup()` and `draw()` blocks. More advanced mouse handling can also be introduced; for instance, the `mousePressed()` function will be called whenever the mouse is pressed. So in the following example, when the mouse is pressed, the screen is cleared via the `background()` function:

```
void setup() {
  size(400, 400);
  stroke(255);
}

void draw() {
  line(150, 25, mouseX, mouseY);
}

void mousePressed() {
  background(192, 64, 0);
}
```

More about static versus active mode programs can be found in the Programming Modes section of the reference, which can be viewed from Help → Environment, or online at *http://processing.org/reference/environment*

Exporting and distributing your work

One of the most significant features of the Processing environment is its ability to bundle your sketch into an applet or application with just one click. Select File → Export to package your current sketch as an applet. This will create a folder named applet inside your sketch folder. Opening the *index.html* file inside that folder will open your sketch in a browser. The applet folder can be copied to a web site intact, and will be viewable by anyone who has Java installed on their system. Similarly, you can use File → Export Application to bundle your sketch as an application for Windows, Mac OS X, and Linux.

The applet and application folders are overwritten whenever you export—make a copy or remove them from the sketch folder before making changes to the *index.html* file or the contents of the folder.

More about the export features can be found in the reference at *http://processing.org/reference/environment/export.html*

Creating images from your work

If you don't want to distribute the actual project, you might want to create images of its output instead. Images are saved with the `saveFrame()` function. Adding `saveFrame()` at the end of `draw()` will produce a numbered sequence of TIFF-format images of the program's output, named *screen-0001.tif*, *screen-0002.tif*, and so on. A new file will be saved each time `draw()` runs — watch out, this can quickly fill your sketch folder with hundreds of files. You can also specify your own name and file type for the file to be saved with a function like:

```
saveFrame("output.png")
```

To do the same for a numbered sequence, use # (hash marks) where the numbers should be placed:

```
saveFrame("output-####.png");
```

For high quality output, you can write geometry to PDF files instead of the screen, as described in the later section about the `size()` function.

Examples and reference

While many programmers learn to code in school, others teach themselves and learn on their own. Learning on your own involves looking at lots of other code: running, altering, breaking, and enhancing it until you can reshape it into something new. With this learning model in mind, the Processing software download includes dozens of examples that demonstrate different features of the environment and API.

The examples can be accessed from the File → Examples menu. They're grouped into categories based on their function (such as Motion, Typography, and Image) or the libraries they use (PDF, Network, and Video).

Find an interesting topic in the list and try an example. You'll see functions that are familiar, e.g. `stroke()`, `line()`, and `background()`, as well as others that have not yet been covered. To see how a function works, select its name, and then right-click and choose Find in Reference from the pop-up menu (Find in Reference can also be found beneath the Help menu). This will open the reference for that function in your default web browser.

In addition to a description of the function's syntax, each reference page includes an example that uses the function. The reference examples are much shorter (usually four or five lines apiece) and easier to follow than the longer code examples.

More about size()

The `size()` function also sets the global variables width and height. For objects whose size is dependent on the screen, always use the width and height variables instead of a number. This prevents problems when the size() line is altered.

```
size(400, 400);

// The wrong way to specify the middle of the screen
ellipse(200, 200, 50, 50);

// Always the middle, no matter how the size() line changes
ellipse(width/2, height/2, 50, 50);
```

In the earlier examples, the `size()` function specified only a width and height for the window to be created. An optional parameter to the `size()` function specifies how graphics are rendered. A renderer handles how the Processing API is implemented for a particular output function (whether the screen, or a screen driven by a high-end graphics card, or a PDF file). Several renderers are included with Processing, each having a unique function. At the risk of getting too far into the specifics, here's a description of the possible drawing modes to use with Processing.

```
size(400, 400, JAVA2D);
```

The Java2D renderer is used by default, so this statement is identical to size(400, 400). The Java2D renderer does an excellent job with high-quality 2D vector graphics, but at the expense of speed. In particular, working with pixels is slower compared to the P2D and P3D renderers.

```
size(400, 400, P2D);
```

The Processing 2D renderer is intended for simpler graphics and fast pixel operations. It lacks niceties like stroke caps and joins on thick lines, but makes up for it when you need to draw thousands of simple shapes or directly manipulate pixels of an image or video.

```
size(400, 400, P3D);
```

Similar to P2D, the Processing 3D renderer is intended for speed and pixel operations. It also produces 3D graphics inside a web browser, even without the use of a library like Java3D. Image quality is poorer (the `smooth()` function is disabled, and image accuracy is low) but you can draw thousands of triangles very quickly.

```
size(400, 400, OPENGL);
```

The OpenGL renderer uses Sun's JOGL (Java for OpenGL) library for faster rendering, while using Processing's simpler graphics APIs and the Processing development environment's easy applet and application export. To use OpenGL graphics, in addition to altering your `size()` function, you must select Import Library, then OpenGL from the Sketch menu. OpenGL applets also run within a web browser without additional modification, but a dialog box will appear asking the user whether they trust “Sun Microsystems, Inc.” to run Java for OpenGL on their computer. If this poses a problem, the P3D renderer is a simpler, if less full-featured, solution.

```
size(400, 400, PDF, "output.pdf");
```

The PDF renderer draws all geometry to a file instead of the screen. Like the OpenGL library, you must import the PDF library before using it. This is a cousin of the Java2D renderer, but instead writes directly to PDF files.

Each of the renderers has its own specific role. P2D and P3D are great for pixel-based work, while the JAVA2D and PDF settings will give you the highest quality 2D graphics. When the Processing project first began, the P2D and P3D renderers were a single choice (and in fact, the only available renderer). This was an attempt to offer a unified mode of thinking about drawing, whether in two or three dimensions. However, this became too burdensome because of the number of tradeoffs that must be made between 2D and 3D. A very different expectation of quality exists for 2D and 3D, for instance, and trying to cover both sides meant doing both poorly.

Loading and displaying data

One of the unique aspects of the Processing API is the way files are handled. The `loadImage()` and `loadStrings()` functions each expect to find a file inside a folder named *data*, which is a subdirectory of the sketch folder.

Advanced Topic: Notes on the data folder

The *data* folder addresses a common frustration when dealing with code that is tested locally but deployed over the web. Like Java, software written with Processing is subject to security restrictions that determine how a program can access resources such as the local hard disk or other servers via the Internet. This prevents malicious developers from writing code that could harm your computer or compromise your data.

The security restrictions can be tricky to work with during development. When running a program locally, data can be read directly from the disk, though it must be placed relative to the user's “working directory,” generally the location of the application. When running online, data must come from a location on the same server. It might be bundled with the code itself (in a JAR archive, discussed later; or from another URL on the same server). For a local file, Java's `FileInputStream` class can be used. If the file is bundled in a JAR archive, the `getResource()` function is used. For a file on the server, `URL.openStream()` might be employed. During the journey from development to deployment, it may be necessary to use all three of these functions.

With Processing, each of these scenarios (and some others) is handled transparently by the file API functions. By placing resources in the data folder, Processing packages the files as necessary for online and offline use.

File handling functions include `loadStrings()`, which reads a text file into an array of String objects, and `loadImage()` which reads an image into a PImage object, the container for image data in Processing.

```
// Examples of loading a text file and a JPEG image
// from the data folder of a sketch.
String[] lines = loadStrings("something.txt");
PImage image = loadImage("picture.jpg");
```

These examples may be a bit easier to read if you know the programming concepts of data types and classes. Each variable has to have a data type, such as `String` or `PImage`.

The `String[]` syntax means “an array of data of the class `String`.” This array is created by the `loadStrings` function and is given the name lines; it will presumably be used later in the program under this name. The reason `loadStrings` creates an array is that it splits the *something.txt* file into its individual lines. The following function creates a single variable of class `PImage`, with the name image.

To add a file to the data folder of a Processing sketch, use the Sketch → Add File menu option, or drag the file into the editor window of the PDE. The data folder will be created if it does not exist already.

To view the contents of the sketch folder, use the Sketch → Show Sketch Folder menu option. This opens the sketch window in your operating system's file browser.

Libraries add new features

A *library* is a collection of code in a specified format that makes it easy to use within Processing. Libraries have been important to the growth of the project, because they let developers make new features accessible to users without needing to make them part of the core Processing API.

Several core libraries come with Processing. These can be seen in the Libraries section of the online reference (also available from the Help menu from within the PDE.) These libraries can be seen at *http://processing.org/reference/libraries/*

One example is the PDF Export library. This library makes it possible to write PDF files directly from Processing. These vector graphics files can be scaled to any size and output at very high resolutions.

To use the PDF library in a project, choose Sketch → Import Library → PDF Export. This will add the following line to the top of the sketch:

```
import processing.pdf.*;
```

Java programmers will recognize the `import` command. In Processing, this line is also used to determine what code is packaged with a sketch when it is exported as an applet or application.

Now that the PDF library is imported, you may use it to create a file. For instance, the following line of code creates a new PDF file named *lines.pdf* that you can draw to.

```
beginRecord(PDF, "line.pdf");
```

Each drawing function such as `line()` and `ellipse()` will now draw to the screen as well as to the PDF.

Other libraries provide features such as writing movie files, reading images from a camera, sending and receiving MIDI and OSC commands, sophisticated 3D camera control, and access to MySQL databases.

Sketching and scripting

Processing sketches are made up of one or more tabs, with each tab representing a piece of code. The environment is designed around projects that are a few pages of code, and often three to five tabs in total. This covers a significant number of projects developed to test and prototype ideas, often before embedding them into a larger project or building a more robust application for broader deployment.

The idea of sketching is identical to that of scripting, except that you're not working in an interpreted scripting language, but rather gaining the performance benefit of compiling to Java class files. Of course, strictly speaking, Java is itself an interpreted language, but its bytecode compilation brings it much closer to the "metal" than languages such as JavaScript, ActionScript, Python, or Ruby.

Processing was never intended as the ultimate language for programming visuals; instead, we set out to make something that was:

- A sketchbook for our own work, simplifying the majority of tasks that we undertake.
- A programming environment suitable for teaching programming to a non-traditional audience.
- A stepping stone from scripting languages to more complicated or difficult languages such as full-blown Java or C++.

At the intersection of these points is a tradeoff between speed and simplicity of use. If we didn't care about speed, it might make sense to use Python, Ruby, or many other scripting languages. This is especially true for the education side. If we didn't care about making a transition to more advanced languages, we'd probably avoid a

C++ or Java-style syntax. But Java makes a nice starting point for a sketching language because it's far more forgiving than C/C++ and also allows users to export sketches for distribution via the web.

Processing assembles our experience in building software of this kind (sketches of interactive works or data-driven visualization) and simplifies the parts that we felt should be easier, such as getting started quickly, and insulating new users from issues like those associated with setting up Java.

Don't start by trying to build a cathedral

If you're already familiar with programming, it's important to understand how Processing differs from other development environments and languages. The Processing project encourages a style of work that builds code quickly, understanding that either the code will be used as a quick sketch, or ideas are being tested before developing a final project. This could be misconstrued as software engineering heresy. Perhaps we're not far from "hacking," but this is more appropriate for the roles in which Processing is used. Why force students or casual programmers to learn about graphics contexts, threading, and event handling functions before they can show something on the screen that interacts with the mouse? The same goes for advanced developers: why should they always need to start with the same two pages of code whenever they begin a project?

In another scenario, the ability to try things out quickly is a far higher priority than sophisticated code structure. Usually you don't know what the outcome will be, so you might build something one week to try an initial hypothesis, and build something new the next based on what was learned in the first week. To this end, remember the following considerations as you begin writing code with Processing:

- Be careful about creating unnecessary structures in your code. As you learn about encapsulating your code into classes, it's tempting to make ever-smaller classes, because data can always be distilled further. Do you need classes at the level of molecules, atoms, or quarks? Just because atoms go smaller doesn't mean that we need to work at a lower level of abstraction. If a class is half a page, does it make sense to have six additional subclasses that are each half a page long? Could the same thing be accomplished with a single class that is a page and a half in total?
- Consider the scale of the project. It's not always necessary to build enterprise-level software on the first day. Explore first: figure out the minimum code necessary to help answer your questions and satisfy your curiosity.

The argument is not to avoid continually rewriting, but rather to delay engineering work until it's appropriate. The threshold for where to begin engineering a piece of software is much later than for traditional programming projects because there is a kind of *art* to the early process of quick iteration.

Of course, once things are working, avoid the urge to rewrite for its own sake. A rewrite should be used when addressing a completely different problem. If you've managed to hit the nail on the head, you should refactor to clean up function names and class interactions. But a full rewrite of already finished code is almost always a bad idea, no matter how "ugly" it may seem.

Adapted from chapter two of Visualizing Data by Ben Fry, O'Reilly and Associates, 2007.