

DotNetProcessing Documentation

Table of Contents

<u>DotNetProcessing Documentation</u>	1
<u>Jonatan Rubio</u>	1
<u>Santi Serrano</u>	1
<u>I. User Documentation</u>	3
<u>Chapter 1. Introduction</u>	4
<u>Chapter 2. The Processing Language</u>	5
<u>2.1. Processing origins</u>	5
<u>2.2. Process VS Timeline</u>	6
<u>2.3. Processing as an intermediate language</u>	6
<u>2.4. Related software</u>	6
<u>2.4.1. DBN</u>	6
<u>2.4.2. Logo</u>	8
<u>2.4.3. Flash</u>	10
<u>2.4.3.1. Bitmap VS Vectors</u>	10
<u>2.4.3.2. Animations</u>	12
<u>2.4.4. Director</u>	12
<u>2.5. Processing is Open Source</u>	13
<u>2.6. Digital art</u>	15
<u>2.7. Processing in the world</u>	17
<u>2.8. First steps</u>	20
<u>Chapter 3. Why DotNetProcessing?</u>	21
<u>Chapter 4. Installation</u>	22
<u>4.1. Prerequisites</u>	22
<u>4.1.1. Windows</u>	22
<u>4.1.2. Linux</u>	22
<u>4.2. Downloading</u>	22
<u>4.3. Installing</u>	22
<u>4.4. Running</u>	22
<u>4.4.1. Windows</u>	22
<u>4.4.2. Linux</u>	22
<u>Chapter 5. Using the program</u>	23
<u>5.1. Graphical Interface</u>	23
<u>5.1.1. Playing with the examples</u>	23
<u>5.1.2. The Syntax ComboBox</u>	23
<u>5.1.3. Writing your first sketch</u>	24
<u>5.1.4. Exporting your sketch</u>	24
<u>5.1.4.1. Executable</u>	24
<u>5.1.4.2. Web</u>	25

Table of Contents

<u>Chapter 5. Using the program</u>	
<u>5.1.4.3. .NET User Control</u>	25
<u>5.1.4.3.1. Using an exported sketch in Microsoft Visual Studio.NET</u>	25
<u>5.1.4.3.2. Using an exported sketch in a .NET application (without Visual Studio)</u>	26
<u>5.1.4.3.3. Going one step further (the full power of .NET on your hands)</u>	28
<u>5.2. Command Line</u>	31
<u>Chapter 6. A Case Study</u>	33
<u>II. Developer Documentation</u>	34
<u>Chapter 7. Getting the sources</u>	35
<u>7.1. The easy way (download them)</u>	35
<u>7.2. The not so easy way (get them by cvs)</u>	35
<u>7.2.1. Windows</u>	35
<u>7.2.2. Linux</u>	36
<u>Chapter 8. Compiling</u>	37
<u>8.1. Windows</u>	37
<u>8.1.1. Microsoft Visual Studio</u>	37
<u>8.1.2. Command Line</u>	37
<u>8.2. Linux</u>	37
<u>Chapter 9. Architecture</u>	38
<u>9.1. DotNetProcessing building blocks</u>	38
<u>9.2. The Syntax</u>	39
<u>9.3. The Primitives</u>	40
<u>9.4. The Surface</u>	41
<u>9.5. Sketch Exportation</u>	41
<u>9.6. Execution Model</u>	42
<u>Chapter 10. Cross-platform issues</u>	43
<u>Chapter 11. Dotnetprocessing vs Processing</u>	44
<u>11.1. Functional differences</u>	44
<u>11.2. Performance test</u>	44
<u>Chapter 12. TODO's</u>	54
<u>Chapter 13. Implementation status</u>	55

Table of Contents

<u>Chapter 14. Writing documentation for DotNetProcessing with DocBook.....</u>	<u>56</u>
<u>Chapter 15. Updating the DotNetProcessing web site.....</u>	<u>57</u>
<u>Chapter 16. Roadmap.....</u>	<u>58</u>
<u>16.1. Mozilla Plugin for .NET User Controls.....</u>	<u>58</u>
<u>16.2. Support for Java and Visual Basic.NET syntax under Linux.....</u>	<u>58</u>
<u>16.3. Live Processing.....</u>	<u>58</u>
<u>16.4. GTK# Environment.....</u>	<u>58</u>
<u>16.5. DotNetProcessing Arena.....</u>	<u>58</u>
<u>16.6. Windows Vista.....</u>	<u>58</u>
<u>16.7. DotNetProcessing for mobile devices.....</u>	<u>60</u>

DotNetProcessing Documentation

Jonatan Rubio

Santi Serrano

Table of Contents

I. User Documentation

1. [Introduction](#)
2. [The Processing Language](#)
 - 2.1. [Processing origins](#)
 - 2.2. [Process VS Timeline](#)
 - 2.3. [Processing as an intermediate language](#)
 - 2.4. [Related software](#)
 - 2.4.1. [DBN](#)
 - 2.4.2. [Logo](#)
 - 2.4.3. [Flash](#)
 - 2.4.4. [Director](#)
 - 2.5. [Processing is Open Source](#)
 - 2.6. [Digital art](#)
 - 2.7. [Processing in the world](#)
 - 2.8. [First steps](#)
3. [Why DotNetProcessing?](#)
4. [Installation](#)
 - 4.1. [Prerequisites](#)
 - 4.1.1. [Windows](#)
 - 4.1.2. [Linux](#)
 - 4.2. [Downloading](#)
 - 4.3. [Installing](#)
 - 4.4. [Running](#)
 - 4.4.1. [Windows](#)
 - 4.4.2. [Linux](#)
5. [Using the program](#)
 - 5.1. [Graphical Interface](#)
 - 5.1.1. [Playing with the examples](#)
 - 5.1.2. [The Syntax ComboBox](#)
 - 5.1.3. [Writing your first sketch](#)
 - 5.1.4. [Exporting your sketch](#)
 - 5.2. [Command Line](#)
6. [A Case Study](#)

II. Developer Documentation

7. [Getting the sources](#)
 - 7.1. [The easy way \(download them\)](#)
 - 7.2. [The not so easy way \(get them by cvs\)](#)
 - 7.2.1. [Windows](#)
 - 7.2.2. [Linux](#)
8. [Compiling](#)

- 8.1. [Windows](#)
 - 8.1.1. [Microsoft Visual Studio](#)
 - 8.1.2. [Command Line](#)
- 8.2. [Linux](#)
- 9. [Architecture](#)
 - 9.1. [DotNetProcessing building blocks](#)
 - 9.2. [The Syntax](#)
 - 9.3. [The Primitives](#)
 - 9.4. [The Surface](#)
 - 9.5. [Sketch Exportation](#)
 - 9.6. [Execution Model](#)
- 10. [Cross-platform issues](#)
- 11. [Dotnetprocessing vs Processing](#)
 - 11.1. [Functional differences](#)
 - 11.2. [Performance test](#)
- 12. [TODO's](#)
- 13. [Implementation status](#)
- 14. [Writing documentation for DotNetProcessing with DocBook](#)
- 15. [Updating the DotNetProcessing web site](#)
- 16. [Roadmap](#)
 - 16.1. [Mozilla Plugin for .NET User Controls](#)
 - 16.2. [Support for Java and Visual Basic.NET syntax under Linux](#)
 - 16.3. [Live Processing](#)
 - 16.4. [GTK# Environment](#)
 - 16.5. [DotNetProcessing Arena](#)
 - 16.6. [Windows Vista](#)
 - 16.7. [DotNetProcessing for mobile devices](#)

List of Tables

- 2-1. [DBN example](#)
- 2-2. [DBN example](#)
- 2-3. [DBN example](#)
- 2-4. [Logo example](#)
- 2-5. [Logo example](#)
- 2-6. [Logo example](#)
- 5-1. [Sketch syntaxes and their associated file extensions](#)
- 7-1. [CVS Access](#)
- 9-1. [Execution model](#)

List of Examples

- 5-1. [VB.NET sketch](#)
- 11-1. [Modified code to show performance](#)

I. User Documentation

Table of Contents

1. [Introduction](#)
 2. [The Processing Language](#)
 - 2.1. [Processing origins](#)
 - 2.2. [Process VS Timeline](#)
 - 2.3. [Processing as an intermediate language](#)
 - 2.4. [Related software](#)
 - 2.4.1. [DBN](#)
 - 2.4.2. [Logo](#)
 - 2.4.3. [Flash](#)
 - 2.4.3.1. [Bitmap VS Vectors](#)
 - 2.4.3.2. [Animations](#)
 - 2.4.4. [Director](#)
 - 2.5. [Processing is Open Source](#)
 - 2.6. [Digital art](#)
 - 2.7. [Processing in the world](#)
 - 2.8. [First steps](#)
 3. [Why DotNetProcessing?](#)
 4. [Installation](#)
 - 4.1. [Prerequisites](#)
 - 4.1.1. [Windows](#)
 - 4.1.2. [Linux](#)
 - 4.2. [Downloading](#)
 - 4.3. [Installing](#)
 - 4.4. [Running](#)
 - 4.4.1. [Windows](#)
 - 4.4.2. [Linux](#)
 5. [Using the program](#)
 - 5.1. [Graphical Interface](#)
 - 5.1.1. [Playing with the examples](#)
 - 5.1.2. [The Syntax ComboBox](#)
 - 5.1.3. [Writing your first sketch](#)
 - 5.1.4. [Exporting your sketch](#)
 - 5.1.4.1. [Executable](#)
 - 5.1.4.2. [Web](#)
 - 5.1.4.3. [.NET User Control](#)
 - 5.2. [Command Line](#)
 6. [A Case Study](#)
-

Chapter 1. Introduction

[Processing](#) is a simplified programming language used by artists, designers or students to create images, animations and sounds. It was born in the [Aesthetics and Computation group](#) of the [MIT](#) and its main purpose was to establish a bridge between artists and engineers. They thought the best way of using a computer as an expression element was talking to it in its own language. But programming languages were too difficult. That's why they created Processing, a programming language so easy that anyone could use it but at the same time so powerful it is possible to do almost everything with it.

DotNetProcessing started as a student final project in the [Barcelona School of Informatics](#). Its main goal is to construct an operative port of the original Processing language (which is based on Java) to the Microsoft .NET framework, extending it with specific tools focused on this platform. This parallel software includes interesting features related to .NET like multiple syntaxes, user control sketch exportation or [mono](#) compatibility.

Chapter 2. The Processing Language

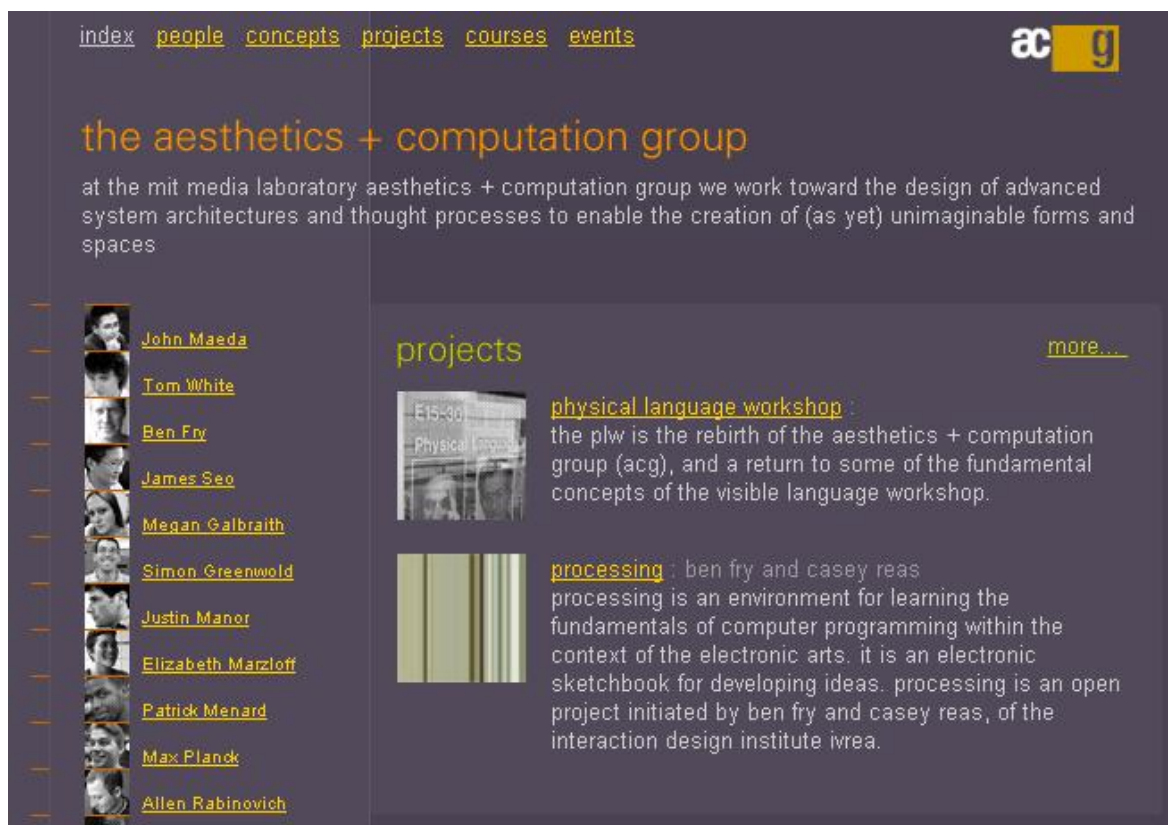
2.1. Processing origins

Processing was born in the Aesthetics and Computation Group at MIT Media Laboratory led by John Maeda. This group is formed by an hybrid mix of designers and engineers that explore computing and aesthetics worlds, apparently very different, building bridges between them.

John Maeda always thought that the best way of using a computer as an expression element was talking to it in its own language. But that implied for artists and designers to learn programming, and that was something hard for what everybody was not prepared. It was necessary to build bridges between this space that separated designers and technicians.

In 1999 John Maeda created Design by Numbers, a programming language with an easy syntax which he used to teach. Design by numbers had something very interesting. Maeda's students could see in every moment the graphical results of what they were programming. That reduced significantly the learning curve because the human brain has innate capacity for spacial recognition.

But Design by numbers was very limited so three years later, Casey Reas and Ben Fry, two students of Maeda, initiated Processing, a language to create graphics so easy as Design by Numbers but at the same time so powerful as any general purpose programming language.

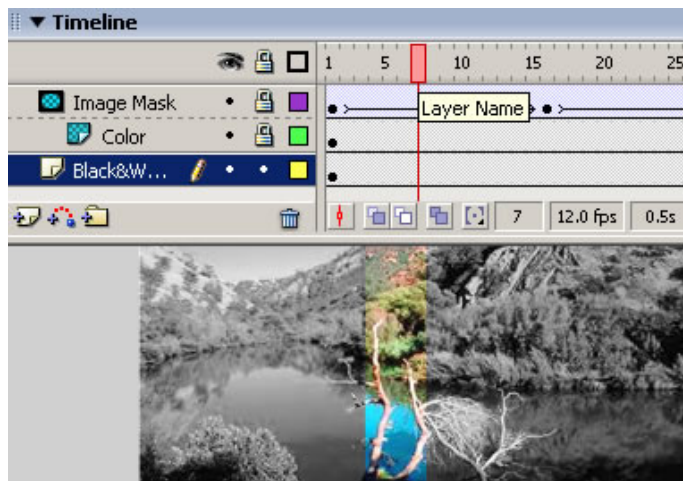


Aesthetics and Computation Group Web at MIT

2.2. Process VS Timeline

If we look at the software that designers use, we can see that most of them use a format in which there is a timeline and the final result is conceived as a movie. Processing, as its name suggests, is based on process definition, a very different perspective focused on the process of creation, not necessary on the final result. In fact, computers are machines that process and combine low level symbols to create high level representations.

This idea is related to the fact that Processing gives the artists complete freedom to do exactly what they imagine. Something that doesn't occur with tools based on countless presets where usually you don't have pixel level precision.



Typical Macromedia Flash Timeline

2.3. Processing as an intermediate language

In the beginning, the idea was not that people stuck on Processing. It only hoped to near the world of programming to artists. In some cases it has become this way. Processing is used by artists to create prototypes because of its easiness. Later, they decide how to build their definitive work. In other cases Processing has been used as part of a more complex project as happened with the videoclip made by the REM music group where the part of following particles in movement was made with Processing.

2.4. Related software

2.4.1. DBN

DBN (Design By Numbers) is a programming language created by John Maeda used as an introduction to computational design. The user can see in every moment the graphical results of what they are programming, reducing significantly the learning curve because the human brain has innate capacity for spacial recognition. Programming was something too boring for artists but with DBN was like

playing.

DBN is also a development environment where it is possible to code and execute programs in the context of drawing. Visual elements like points and lines are combined along with programming concepts to create images.

DBN is a not general purpose programming language like C or Java. It was created for people to make their first steps in digital art. It's free, multiplatform, easy, and you can even export your creations to the web.

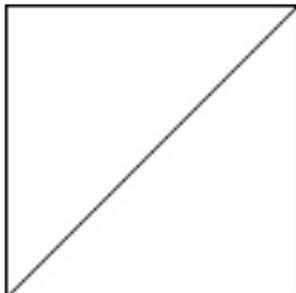
We can say that Processing is the evolution of Design by Numbers.

The number of commands available in DBN is very limited because it was created mainly for teaching. Thus, it's not very powerful but very easy. Every command has a numeric attribute normally between 0 and 100.

For example, if we execute *Paper 50* we obtain a grey canvas (50% black). With *Pen 0* we obtain a white pen. Lines are created with commands like *Line 0 0 100 100* meaning we are drawing a line from point (0,0) to point (100,100).

Let's see a simple example using those three commands:

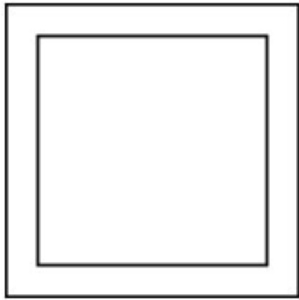
Table 2–1. DBN example

Example	Result
<pre>Paper 0 Pen 100 Line 0 0 100 100</pre>	

We can combine lines to create forms:

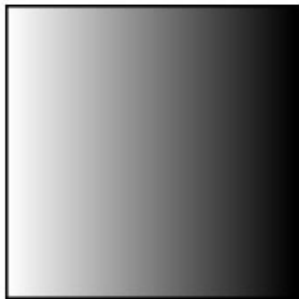
Table 2–2. DBN example

Example	Result
---------	--------

<pre>Paper 0 Pen 100 Line 10 10 10 90 Line 10 90 90 90 Line 90 90 90 10 Line 90 10 10 10</pre>	
--	--

In this example we create a gradient with a simple loop:

Table 2–3. DBN example

Example	Result
<pre>Paper 0 Pen 100 Repeat A 0 100 { Pen A Line A 0 A 100 }</pre>	

2.4.2. Logo

Logo is a programming language designed as a tool for teaching. It was created in 1967 and was based on LISP. Its intellectual roots are in artificial intelligence. The idea of logo is that a turtle with a pen can be instructed to do simple things like move forward 100 spaces or turn around. From these building blocks you can create more complex shapes like squares, triangles or circles.

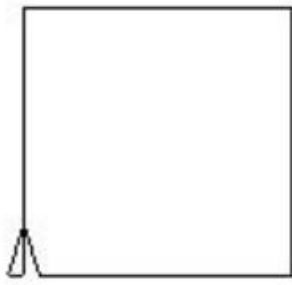
The language is very intuitive being very easy to learn. A student could understand (and predict and reason about) the turtle's motion by imagining what they would do if they were the turtle. That made the language ideal for teaching computing concepts. At the same time, experimented users can make complex projects.

The turtle itself can be replaced with other shapes like birds, cars, airplanes or anything else. That makes possible even to create simple games.

Let's see how we can draw a square with Logo:

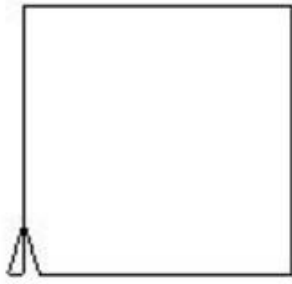
Table 2–4. Logo example

Example	Result
---------	--------

<pre>forward 50 right 90 forward 50 right 90 forward 50 right 90 forward 50 right 90</pre>	
--	--


In the example above there's a pattern that's being repeated. We could replace it with a simple loop:

Table 2–5. Logo example

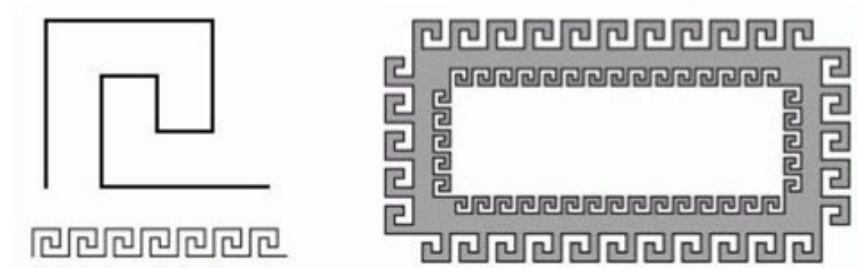
Example	Result
<pre>repeat 4 [forward 50 right 90]</pre>	

The square itself can be used to create other compositions:

Table 2–6. Logo example

Example	Result
<pre>repeat 12 [square right 30]</pre>	

With a little more work it's possible to create things like this:



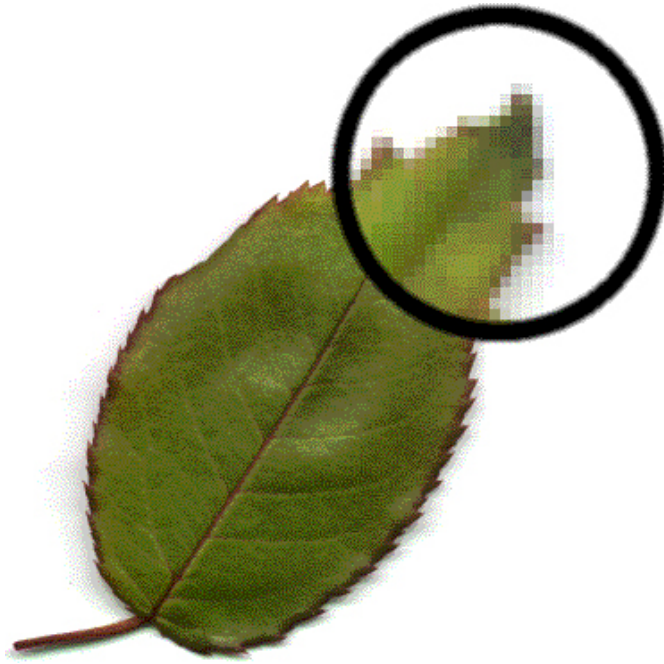
2.4.3. Flash

Macromedia Flash is a tool for creating 2D animations. It supports interesting features like vector graphics, a scripting language called ActionScript, bidirectional streaming of audio and video, compression and interaction possibilities. Its user interface is very easy and will be familiar with everyone that have worked with vector graphics since it uses concepts that are on other popular commercial tools like Freehand, Illustrator or Corel Draw. The inclusion of sound is another feature of Flash. It supports many formats including the possibility to reproduce audio files even before they have been downloaded using buffers. Sounds can be synchronized with special effects and events.

Flash has become very popular for creating rich elements on web pages. Almost every internet browser today has installed the corresponding plugin that reproduces ".SWF" files (the file format for Flash animations). This popularity has been also criticized because most web pages have abused of Flash animations substituting traditional elements of navigation and presenting information by non-standard Flash animations. Once again, the tool is not the problem but the use people made of it. In this way, Flash is not convenient for developing an entire web page but is ideal for creating visual design elements or games. Flash was not specifically created for the web and it can run in many other platforms like telephones or PDAs.

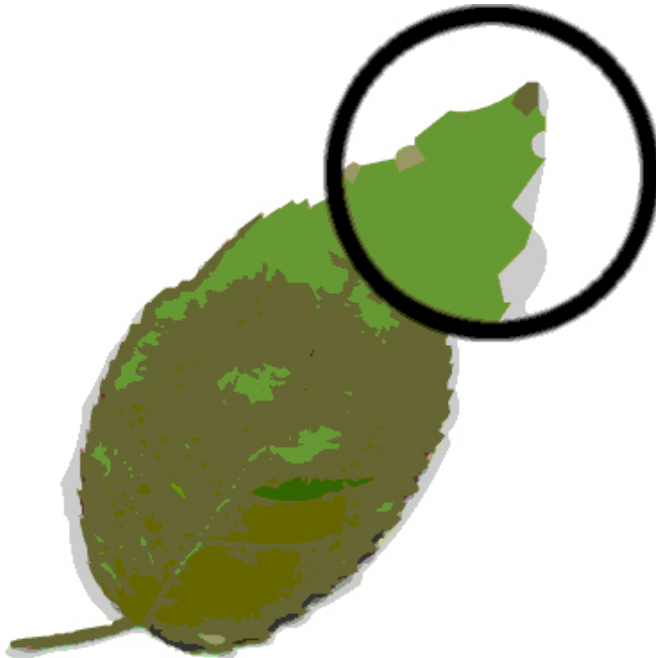
2.4.3.1. Bitmap VS Vectors

Bitmap Graphics create images with *pixels*, a color inside a cell. For example, this leaf has an associated color for every pixel on the image creating an image very similar to a mosaic. This has the problem that since we have no information about the shapes that form the image, it is impossible to make changes like resizing the image without loosing quality.



Bitmap Graphics

However, when we work with vector graphics, it is possible to resize, move or change colors of an image without losing the quality of the original image. Vector graphics are resolution independent and can be displayed in a variety of mediums without losing quality.

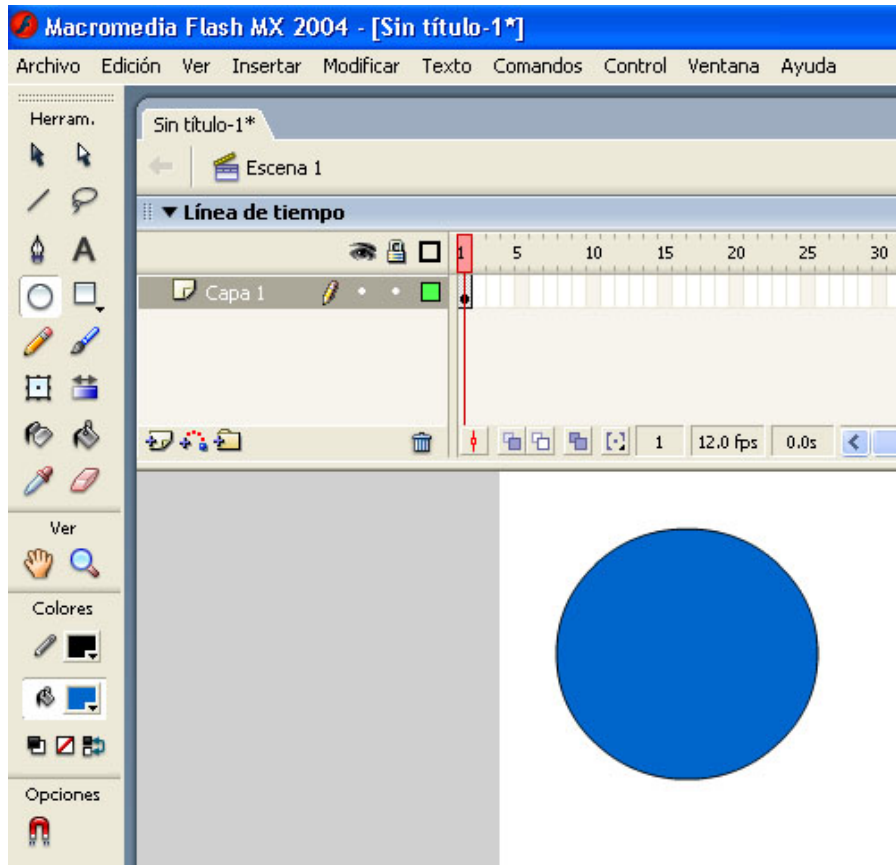


Vector Graphics

In this example, the image of the leaf is created with points and lines going from point to point defining the leaf outline. Both the outline and the inside of the leaf have their associated color.

2.4.3.2. Animations

Animations are created inserting objects in a timeline working with different layers. This way, it is possible to create complex animations easy and fast. To get better results we can apply many effects like deformation or contrast. Object can even point to an URL when some action occur.



Macromedia Flash Environment

2.4.4. Director

Macromedia Director is a tool for creating multimedia applications. It is possible to combine images, sounds, text or video in one unique file which can be exported to many formats including AVI and Shockwave. It is not a program for creating contents but for integrating them. The user acts as the *Director* of a movie.

The environment is similar to Flash, elements are combined in a timeline. The program includes support for visual effects, vector graphics, Lingo scripting and many more. Those features are kept in groups called *media assets*.

2.5. Processing is Open Source

Processing has contributed to the free software community and specifically to digital art building a free tool whose source code is available through the [Processing Developer page](#).

In this world where commercial tools from companies like Adobe or Macromedia are on the spot, Processing has made his own space offering an close alternative to digital art. That was from the beginning since Processing was born in an educational context targeting a wide group of people.

Anybody can download a free copy of Processing and start working immediately in an easy environment not full of countless options that saturate users. Furthermore, the Processing community grows day by day and it's possible to access the source code of the majority of sketches that are on the net, stimulating exchange and learning. That's the spirit of Processing.

Another important thing is that Processing has versions available for Windows, Mac and Linux operating systems. That's because it's based on Java, a free multiplatform developing software.

Processing is already on the Creative Commons trend. A project initiated three years ago offering artists and creators simple licenses that lets them distribute and share their works with full legal security. No need to choose between Public Domain (no rights reserved) / Copyright (all rights reserved) any more. Now it's possible to explicit if a work can be copied, distributed, changed or even used commercially. Creative commons introduce the concept of "some rights reserved".

 Develop Processing

[Cover](#) \ [Build](#) \ [Source](#) \ [Bugs](#) \ [Reference](#) \ [Libraries](#)




↳ [ViewCVS Help](#) \ [Development]

Index of /

Current revision: [2244](#) (of [2244](#))

Jump to directory revision:

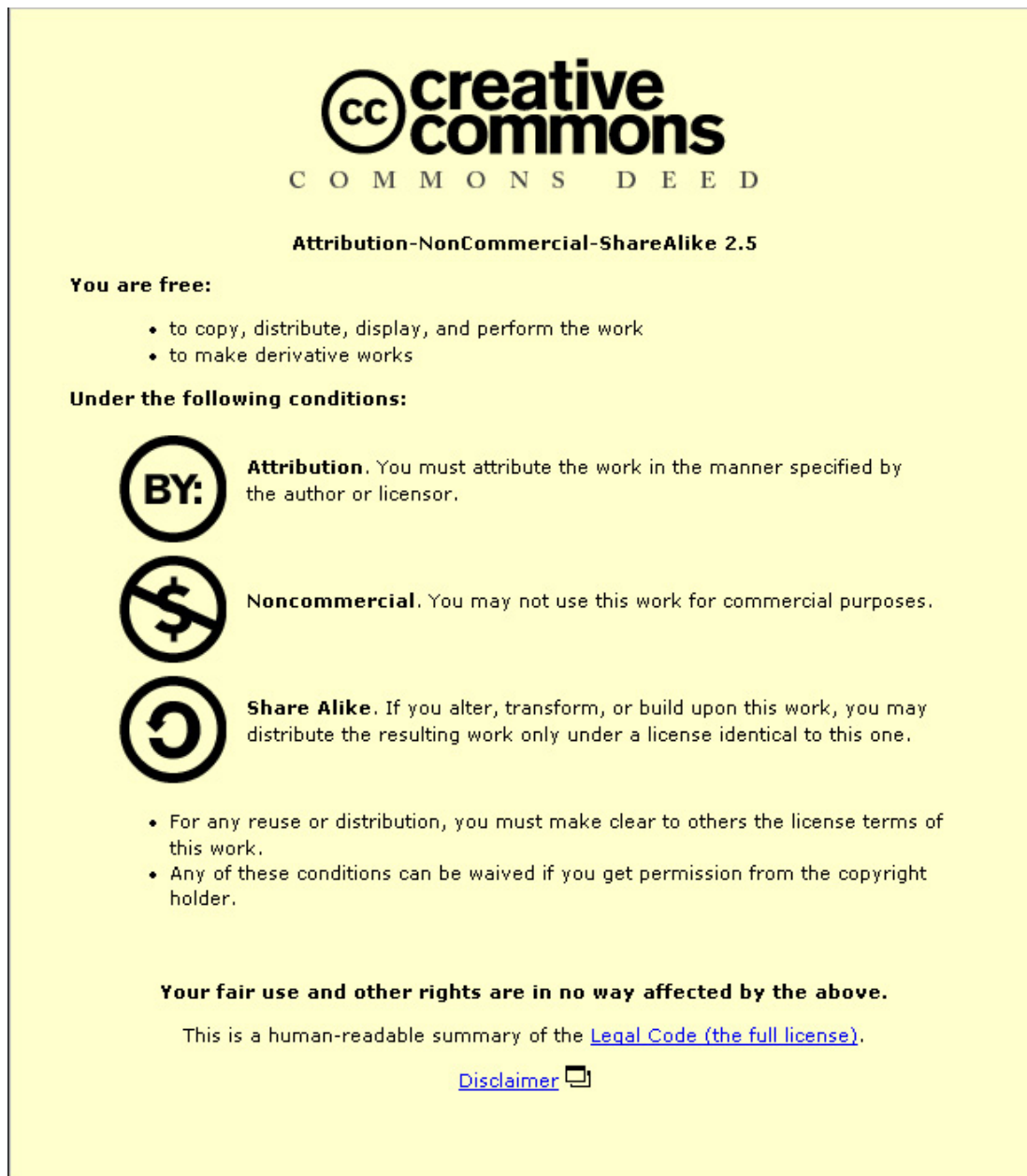
Files shown: **0**

File ^	Rev.	Age	Author	Last log entry
 _branches/	2180	4 weeks	jdf	Force 1.1-compatible
 _tags/	2231	11 days	fry	rev 0114, bug fixes and hint(ENABLE_DEPTH_SORT)
 _trunk/	2244	79 minutes	fry	beginning work on beginGL/endGL, misc todo items

Processing is an open project initiated by [Ben Fry](#) and [Casey Reas](#)

© [Info](#) \ Source browsing: [ViewCVS 1.0-dev](#) \ Site hosted by [Media Temple!](#)

Processing code available to download by CVS



Creative Commons License



No Rights Reserved



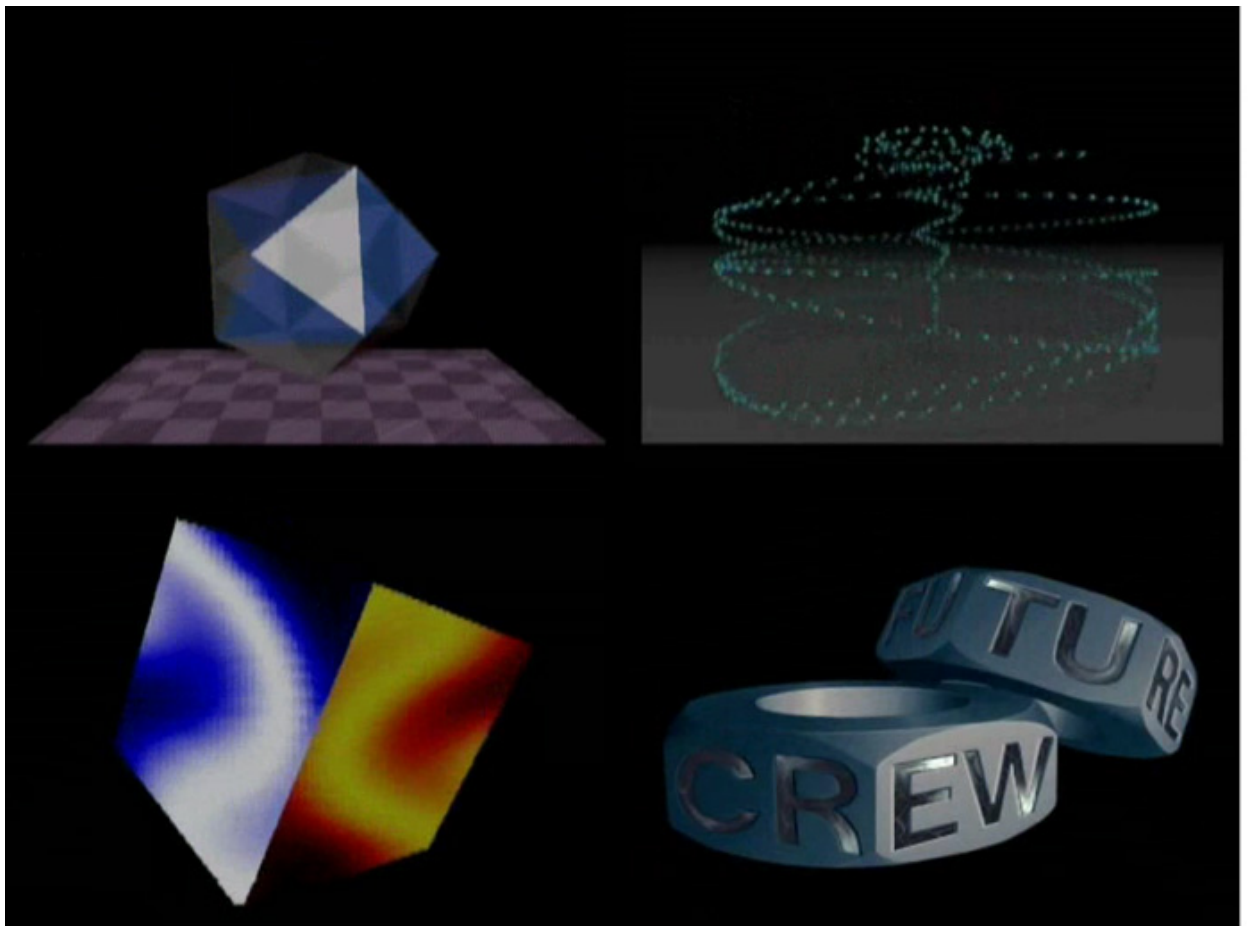
Some Rights Reserved

2.6. Digital art

Digital art is art created on a computer in digital form. Digital art can be purely computer-generated or taken from another source, such as a scanned photograph. Processing is in fact a software for creating art with a computer. In this area there are other interesting disciplines:

Demoscene

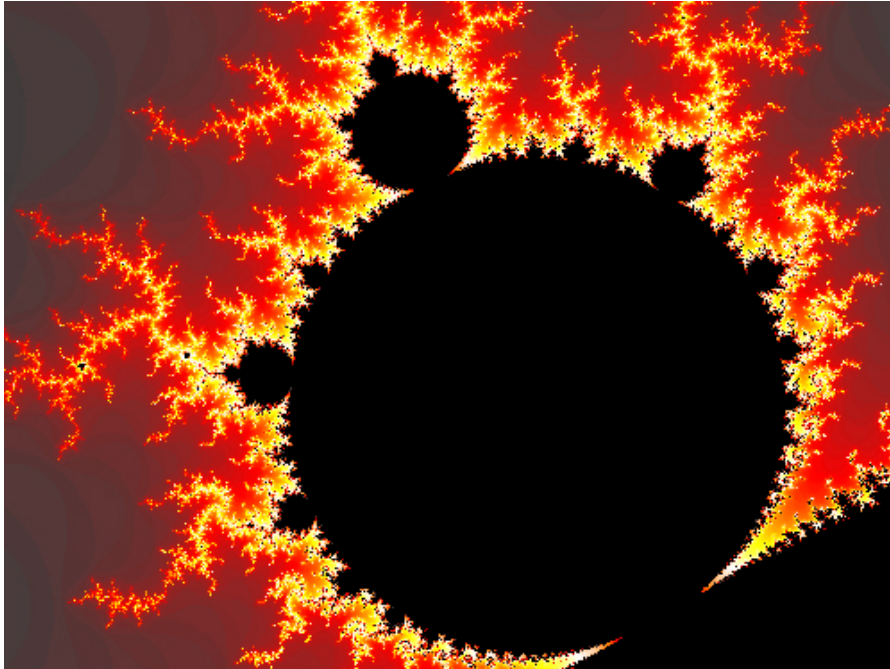
The demoscene is a computer subculture that appear in the late 1970s and early 1980s. Initially they were impressive-looking graphical animations used by hackers as a symbol. Lately they become a way of generating complex 3D graphics usually programmed in low level assembly language. That was because by the time demos appeared, computers where not powerful enough to process such complexity from a high level programming language.



Some captures from Second Reality by Future Crew. Winner of Assembly'93 PC demo competition.

Fractals

Fractals are shapes that are recursively constructed or self-similar. Fractals are used in many disciplines like medicine, music, cosmology and computer graphics where mathematical functions generate still images, animations and even music.



Fractal example

ASCII art

ASCII art consists of pictures created with any text editor using characters defined by ASCII. Originally they were used when early printers lacked graphics ability. Probably the simplest form of ASCII art are the well-known smilies, combinations of two or three characters for expressing emotion in text.



ASCII Art image

Rendering

Rendering is the process of generating an image from a model, by means of a software program. The model is a description of three dimensional objects in a strictly defined language or data structure. It would contain geometry, viewpoint, texture and lighting information. It has uses in computer and video games, simulators, movies or TV special effects.

Raytracing

Ray tracing is a general technique from geometrical optics of modelling the path taken by light by following rays of light as they interact with optical surfaces. The term is applied to mean a specific rendering algorithmic approach in 3D computer graphics, where mathematically-modelled visualizations of programmed scenes are produced using a technique which follows rays from the eyepoint outward, rather than originating at the light sources.



Photo-realistic image made with the PovRay software.

Video games

Video games are computer games where a video display such as a monitor or television is the primary feedback device. The social and artistic importance of video games has recently been officially acknowledged by The British Academy of Film and Television Arts elevating the sector to become an equal to those for Film and Television.

2.7. Processing in the world

At the moment, Processing is used mainly for teaching because of its easiness. Important universities such as [Washington](#), Virginia, [Copenhagen](#), Londres, Berlin, [New York](#) or Roma use them, as well as in many art schools.

For example, in the Chicago School of the Art Institute, the department of art and technology uses Processing in one of its [courses](#), along with Director in order to juxtapose traditional practices of analog drawing with the process of sketching in code.

programming
for
automatic
drawing
systems

instructor
tiff holmes

overview
syllabus
final
examples
links
people
events

wiki

about the
instructors:

tiff holmes is an
assistant
professor of art
and technology
studies at the
school of the art
institute of
chicago—her
recent work is
focused on
eco-visualization.

email
tholme (at) artic
(dot) edu

seth hunter is a
TA for PADS '06

email
seth (at)
perspectum (dot)
com

overview

In PADS, we juxtapose traditional practices of analog drawing with the process of sketching in code. Technically, the course will introduce the fundamentals of computer programming within a visual context through the development of an online software sketchbook. The first five tutorials are based in Lingo (Director) the second five are based in Processing (Open shareware, Java API). Studio demonstrations will include: software that responds to environmental changes (temperature, oxygen, light), animations that change based on mouse input, software that draws from online data, and software that draws from a human heartbeat.

course requirements

- weekly electronic drawing sketch posted online with source code
- occasional readings + class presentations
- midterm
- final project

readings + screenings

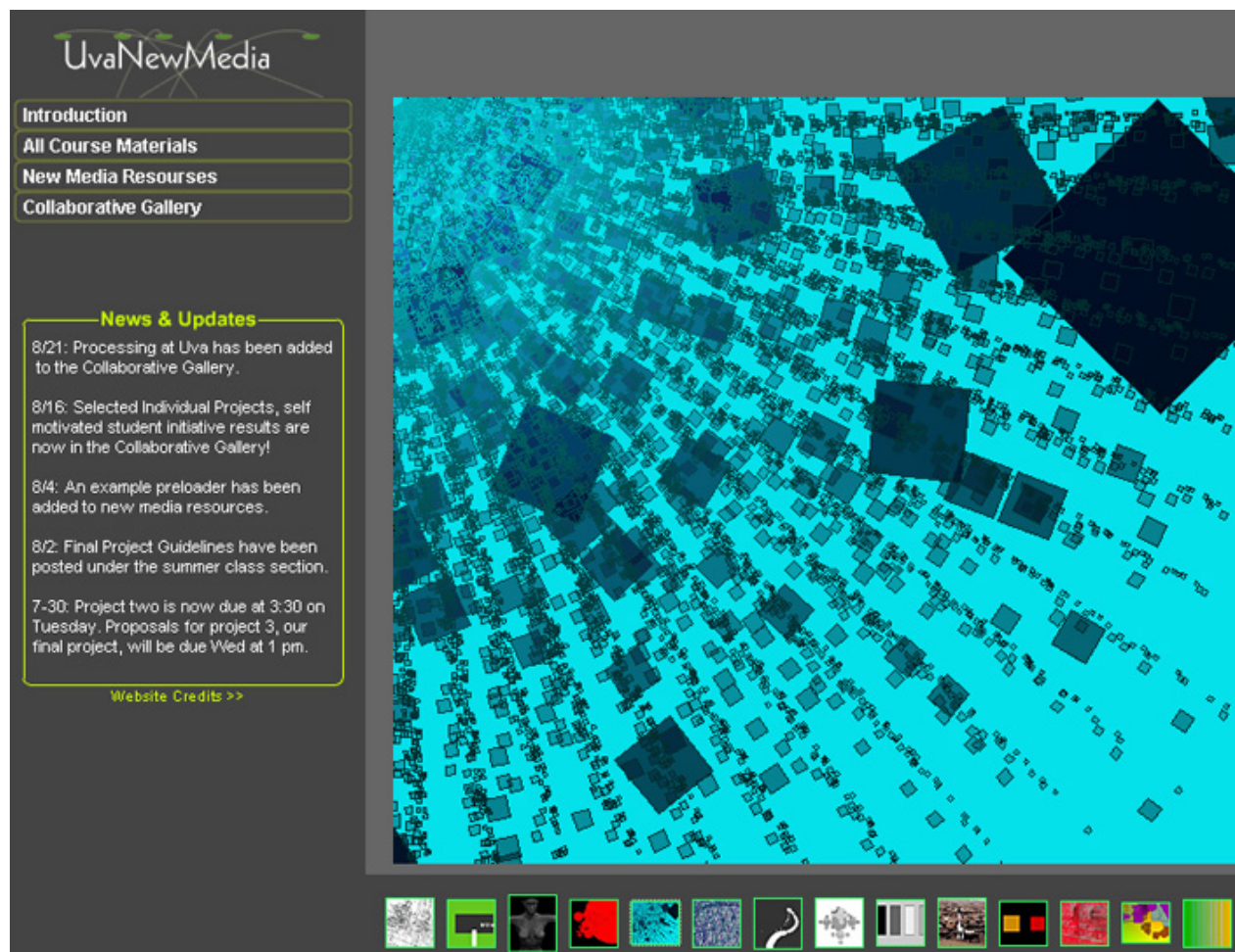
Jean Tinguely, Matthew Ritchie, John Cage, Carolee Schneemann, Douglas Englebart, Harold Cohen, John Maeda, John Simon, Ben Fry, Mary Flanagan, Casey Reas, Golan Levin, Camille Utterback, Martin Wattenberg, Harold Cohen, and others.

attendance policy

3 or more unexcused absences = No Credit

Programming for automatic drawing systems











At [uvanewmedia](#), a virtual gallery, resource toolkit, and collaborative meeting space for new media artists of all disciplines at the University of Virginia, they use Processing for teaching algorithmic graphics, and interactive java applets.



UvaNewMedia

In the Processing page there is also a very active forum where it is discussed many things related with the language: announcing events, tools, syntax, libraries, bugs and many others.

Processing 1.0 (BETA)

Forum name	Topics	Posts	Last post
Discussion			
 Community, Collaboration, Status Interface with the Processing developers and read about the current status <i>Moderators: fry, REAS</i>	126	740	May 6 th , 2006, 1:45pm by guytam
 Events, Publications, Opportunities Processing related conferences, courses, workshops, concerts, articles, jobs, etc. <i>Moderators: fry, REAS</i>	89	212	Apr 20 th , 2006, 11:19pm by mflux
Programming Questions & Help			
 Syntax Questions about the Processing Language <i>Moderators: fry, REAS</i>	460	1606	May 6 th , 2006, 8:00am by fjen
 Programs Questions about a specific Processing program <i>Moderators: fry, REAS</i>	287	1078	May 5 th , 2006, 10:22am by mark_h
 Integration Processing and other systems (Java, PHP, MAX, MySQL, etc) <i>Moderators: fry, REAS</i>	140	547	May 5 th , 2006, 2:42pm by fry
Suggestions & Bugs			
 Software, Website Suggestions Do you have suggestions for the Processing software or website? <i>Moderators: fry, REAS</i>	119	336	Today at 11:06am by seltar
 Software Bugs Have you found an error in the Processing software? <i>Moderators: fry, REAS</i>	217	886	May 5 th , 2006, 8:40pm by zai
 Website, Reference, Example, Bugs Have you found an error in the Processing website, examples, or reference? <i>Moderators: fry, REAS</i>	123	297	May 6 th , 2006, 12:07pm by fjen
Topics & Contributions			
 3D, OpenGL Working with 3D graphics using P3D and OpenGL <i>Moderators: fry, REAS</i>	167	700	May 5 th , 2006, 3:40am by flight404
 Performance, Games, Interfaces, Widgets Interactions between people and programs <i>Moderators: fry, REAS</i>	60	218	Apr 25 th , 2006, 7:47pm by fry

The Processing Forum

2.8. First steps

[...]

Chapter 3. Why DotNetProcessing?

Although DotNetProcessing is far from getting the maturity of the original Processing software there are a few exclusive features to take into account:

- You can write sketches in different syntaxes. Not only Java.
 - Integrate your sketches in your own .NET applications thanks to the *.NET User Control Export* option.
 - The DotNetProcessing binaries not counting the Microsoft .NET Framework have a size less than *200 Kb*.
 - .NET is *fast* multi-platform technology.
 - More to come.
-

Chapter 4. Installation

4.1. Prerequisites

In order to run the program your system has to be capable of running .NET applications. Make sure one of the following is installed on your system:

4.1.1. Windows

You can choose:

- [Microsoft .NET Framework](#) (recommended)
- [Mono for Windows](#)

4.1.2. Linux

Get the Mono platform:

- [Mono for Linux](#)

4.2. Downloading

The program can be downloaded in the [DotNetProcessing download page](#). Get the *OS Independent Binaries*.

4.3. Installing

Just unzip the downloaded file to some place in your hard disk.

4.4. Running

4.4.1. Windows

Double click on the `DotNetProcessing.exe` executable file.

4.4.2. Linux

Use the following command to run the program:

mono DotNetProcessing.exe



At the time of writing this documentation the DotNetProcessing graphical environment is very unstable under Linux. For better results use the [command line](#).

Chapter 5. Using the program

5.1. Graphical Interface

5.1.1. Playing with the examples

One of the first things you can do when entering the program is load some of the examples that comes with it. They are on the File menu under the Examples submenu. Note that the source code of the examples shows in the center of the window. Now push the Build and Run button and the resulting sketch will appear.

5.1.2. The Syntax ComboBox

One of the key features of .NET technology is the possibility of writing code in various different programming languages. This is because only syntax change between them. The underlying .NET class library is the same for all. It was a natural thing to have Processing sketches written in different .NET languages.

In the upper part of the environment window there is a combobox which lets you choose the syntax of your sketch. At the time of writing this documentation there are four possibilities:

J#

This is the most compatible syntax with the original Processing language since it is based on Java. Under Windows, if you don't have the [J# Redistributable Package](#) installed in your system you won't see this option. Under Linux you won't see this option.

C#

For sketches written in C#. Note that although Java and C# have similar syntax they are not exactly equal and many sketches written for the original Processing language won't compile under this option.

VB.NET

For sketches written in VB.NET.

Java Emulation

This option tries to emulate Java syntax parsing your sketch and changing those parts that are not C# compatible since the compiler internally used is C#. This is specially appropriate to try Java syntax sketches without installing the J# Redistributable Package. Note that at the time of writing this documentation this option is in a very early stage.

Every syntax has an associated file extension which the program uses to detect in which syntax the sketch you are loading is written. Java and J# sketches have the same extension as the original Processing language: ".pde". C# sketches have the ".cs.pde" extension. VB.NET sketches have the ".vb.pde" extension.

Table 5–1. Sketch syntaxes and their associated file extensions

Syntax	File Extension
Java/J#	.pde

C#	.cs.pde
Visual Basic.NET	.vb.pde
Java Emulation	.pde

Example 5–1. VB.NET sketch

```

Dim x1 As Integer
Dim y1 As Integer
Dim x2 As Integer
Dim y2 As Integer

Sub setup()
    size(400,400)
    x1 = 1
    y1 = 1
    x2 = 400
    y2 = 400
End Sub

Sub draw()
    If (x2>0) then
        rect(x1,y1,x2,y2)
        x1=x1+1
        y1=y1+1
        x2=x2-2
        y2=y2-2
    End If
End Sub

```

5.1.3. Writing your first sketch

If you are new to the Processing language take some time navigating through the original Processing language home page:

- processing.org
- [The complete functions reference](#)
- [First steps](#)

Once you are familiar with the Processing language you can load one of the examples and make little changes to see how they affect the resulting sketch.

5.1.4. Exporting your sketch

5.1.4.1. Executable

Just what it says. Export to executable and you'll have a ".exe" file containing your sketch. In Windows double click to show it. Linux users type **mono sketchname.exe** (substitute sketchname with the name of your sketch).



At the time of writing this documentation the program has the limitation that if your sketch is written in J# or VB.NET some DLLs will be generated with your executable. Just keep all them together for proper functioning.

5.1.4.2. Web

When you choose this option the program will generate an HTML and one or more DLL files. If you drop all them to a web server, visitors will be able to see the sketch if they have Internet Explorer and the .NET Framework installed. Linux users won't be able to see the sketch.



At the time of writing this documentation the program has the limitation that if your sketch is written in J# or VB.NET you'll have to host the exported sketch in an IIS Server because the generated files include more than one DLLs and only IIS Servers will send all of them to the client.

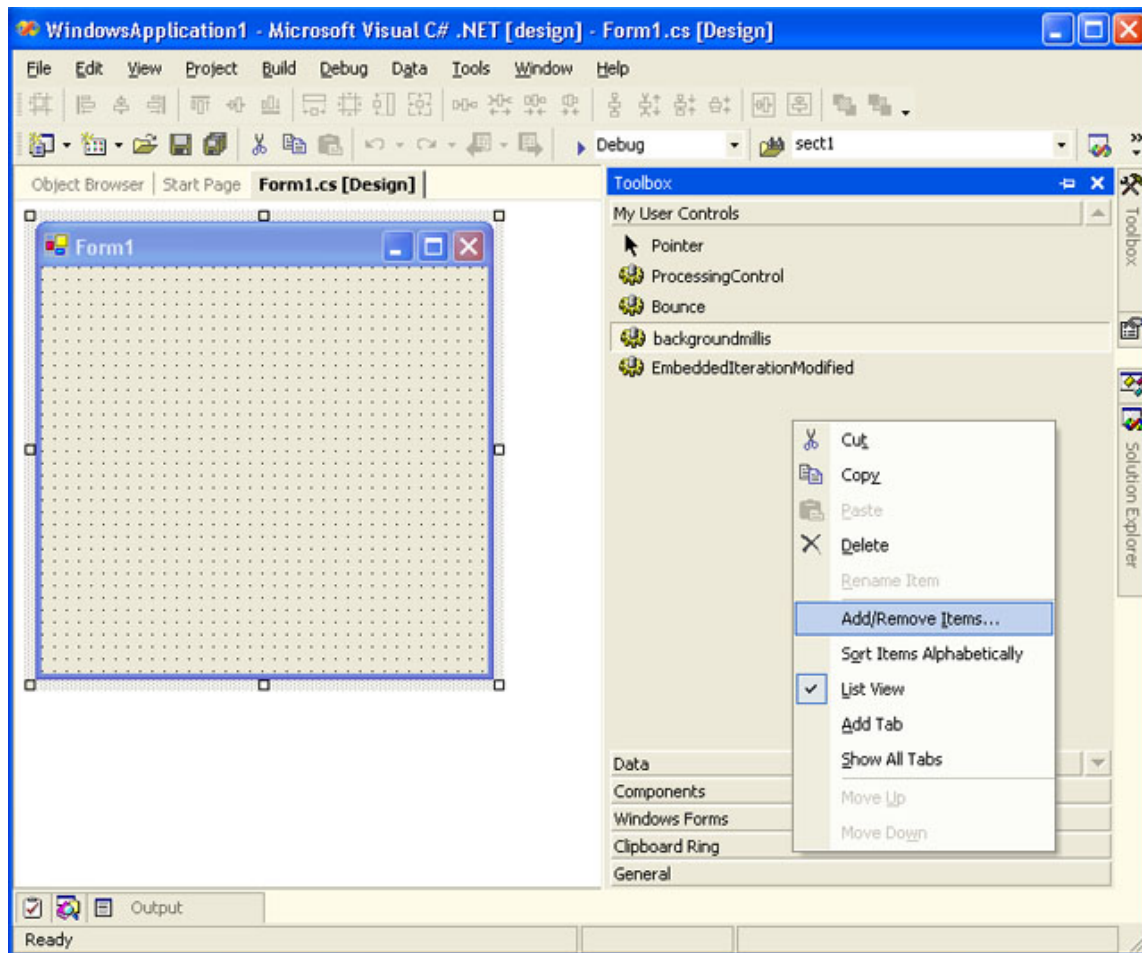
5.1.4.3. .NET User Control

This is quite an interesting feature. When you export a sketch to executable or to web the result is something closed. You can't do anything with it apart from sending it to someone or displaying it in a web page. But it would be nice to use your sketch along with something else, for example, a .NET application. That's exactly what this option is about. The program will generate one or more DLL files containing your sketch as a .NET User Control. .NET User Controls are graphical OOP classes. In other words, is what you see in a typical windows form: buttons, menus, check boxes, grids, tabs, etc. Your sketch will be another one.

5.1.4.3.1. Using an exported sketch in Microsoft Visual Studio.NET

We are going to see the whole process of how you can integrate your sketch in a .NET application developed with Microsoft Visual Studio.NET.

1. Export your sketch to some folder on your system
2. Open Visual Studio
3. Create a new windows application project
4. Go to the toolbox – My User Controls
5. Right mouse click – Add/Remove Items
6. Under the .NET Framework Components tab browse to the generated DLL that contains your sketch. If you see more than one take the one that has the name you gave to the sketch.
7. Press the OK button and your sketch will be now on the toolbox
8. Drag the sketch to a form
9. Drag two buttons to the form
10. Call the sketch start method in the first button mouseclick event:
`sketchname1.Start(); // C# - substitute sketchname with the name of your sketch`
11. Call the sketch stop method in the second button mouseclick event:
`sketchname1.Stop(); // C# - substitute sketchname with the name of your sketch`
12. Run the program



Add user control to the toolbox

5.1.4.3.2. Using an exported sketch in a .NET application (without Visual Studio)

Now let's suppose we don't have Visual Studio and we have to hard code our application with a simple editor and build it with the command line. The example `EmbeddedIterationModified.cs.pde` will be used. Substitute everywhere with the name of your sketch.

1. Export your sketch to some folder on your system
2. In this folder create a file called `EmbeddedIterationModified.cs` with the following code:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace EmbeddedIterationModifiedNamespace
{
    public class EmbeddedIterationModifiedForm : System.Windows.Forms.Form
    {
```

```

private DotNetProcessing.Running.EmbeddedIterationModified
    EmbeddedIterationModifiedSketch;
private System.Windows.Forms.Button startButton;
private System.Windows.Forms.Button stopButton;

private System.ComponentModel.Container components = null;

public EmbeddedIterationModifiedForm()
{
    InitializeComponent();
}

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}

private void InitializeComponent()
{
    this.EmbeddedIterationModifiedSketch =
        new DotNetProcessing.Running.EmbeddedIterationModified();
    this.startButton = new System.Windows.Forms.Button();
    this.stopButton = new System.Windows.Forms.Button();
    this.SuspendLayout();

    this.startButton.Location = new System.Drawing.Point(5, 210);
    this.startButton.Name = "startButton";
    this.startButton.TabIndex = 0;
    this.startButton.Text = "Start";
    this.startButton.Click += new System.EventHandler(this.startButton_Click);

    this.stopButton.Location = new System.Drawing.Point(120, 210);
    this.stopButton.Name = "stopButton";
    this.stopButton.TabIndex = 0;
    this.stopButton.Text = "Stop";
    this.stopButton.Click += new System.EventHandler(this.stopButton_Click);

    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(200, 240);
    this.Controls.Add(this.startButton);
    this.Controls.Add(this.stopButton);
    this.Controls.Add(this.EmbeddedIterationModifiedSketch);
    this.Name = "EmbeddedIterationModifiedForm";
    this.Text = "EmbeddedIterationModified";
    this.ResumeLayout(false);

    this.EmbeddedIterationModifiedSketch.Location = new System.Drawing.Point(0, 0);
    this.EmbeddedIterationModifiedSketch.Name = "EmbeddedIterationModifiedSketch";
    this.EmbeddedIterationModifiedSketch.Size = new System.Drawing.Size(200, 200);
    this.EmbeddedIterationModifiedSketch.TabIndex = 0;
}

[STAThread]
static void Main()

```

```

    {
        Application.Run(new EmbeddedIterationModifiedForm());
    }

    private void startButton_Click(object sender, System.EventArgs e)
    {
        this.EmbeddedIterationModifiedSketch.Start();
    }

    private void stopButton_Click(object sender, System.EventArgs e)
    {
        this.EmbeddedIterationModifiedSketch.Stop();
    }
}

```

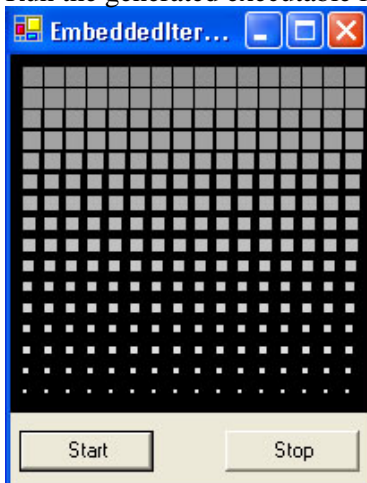
3. Build the code with the following command line:

```

csc -target:winexe -out:EmbeddedIterationModified.exe
-r:EmbeddedIterationModifiedSketch.dll EmbeddedIterationModified.cs

```

4. Run the generated executable file



.NET User Control Sketch in your application

5.1.4.3.3. Going one step further (the full power of .NET on your hands)

Ok. You've exported your sketch to Visual Studio and have two cute methods: *Start* and *Stop*. But, what if you want more? There are a few tricks that will let you define your sketches specifically for being used from another .NET application. Basically thanks to *Properties* and *Events*. Take a sit.

Properties

Properties are variables that you don't access directly. They have two special methods called *get* and *set* specially designed to perform additional processing before reading or setting its value. If you want to see your variables in Visual Studio Property Inspector you have to encapsulate them using properties.

Events

Once you've designed events for your sketch you will be able to subscribe to them in your .NET application and therefore be advised when something happens on your sketch.

Let's go for an example. We are going to take the Collision example. It's like a ping-pong game where you have to push a ball with a paddle. So how can we interact with this sketch? Imagine we want to have a variable paddle size. That means you are going to have the possibility of changing the paddle size even after the sketch has been exported. We can even do something everytime the ball touches the paddle. Follow this steps:

1. Open the "Collision.cs.pde" example.
2. Insert the following lines before the beginning of the sketch:

```
public int PaddleHeight
{
    get
    {
        return paddle_height;
    }
    set
    {
        paddle_height = value;
    }
}
```

Those lines convert the *paddle_height* variable in a property with its get and set method. The resulting property name is *PaddleHeight* and is what you will see in Visual Studio Property Inspector.

3. Insert the following lines after the property definition:

```
public delegate void PaddleTouchDelegate();
public event PaddleTouchDelegate PaddleTouch;

private void FireAwayPaddleTouch()
{
    if (PaddleTouch != null) PaddleTouch();
}
```

That's a little more tricky. The first line is declaring a delegate called *PaddleTouchDelegate*. The second line is declaring an event of type *PaddleTouchDelegate*. And then there is the method which you will call inside your sketch to fire the event in case somebody is subscribed to it.

4. We are missing something yet. As we said before, the event has to be fired when the ball touches the paddle. Locate the following code and insert the line in bold:

```
// Test to see if the ball is touching the paddle
float py = width-dist_wall-paddle_width-ball_size;
if(ball_x == py
    && ball_y > paddle_y - paddle_height - ball_size
    && ball_y < paddle_y + paddle_height + ball_size) {
    ball_dir *= -1;
    if(mouseY != pmouseY) {
        dy = (mouseY-pmouseY)/2.0;
        if(dy > 5) { dy = 5; }
        if(dy < -5) { dy = -5; }
    }
}
```

```
FireAwayPaddleTouch( );  
}
```

That's it. Our sketch is ready to be exported.



5. Export your sketch to somewhere on your system. Call it *MyCollision*
6. Create a new Visual studio project of type *Windows Application* and *C# Syntax*
7. Add the exported sketch to the *toolbox* and drop one instance to the form.
8. Now look at the *property inspector* and search for the *PaddleHeight* property. Change its value to *30*.
9. Again in the property inspector open the events list and search for the *PaddleTouch* event. Double click and type the following:

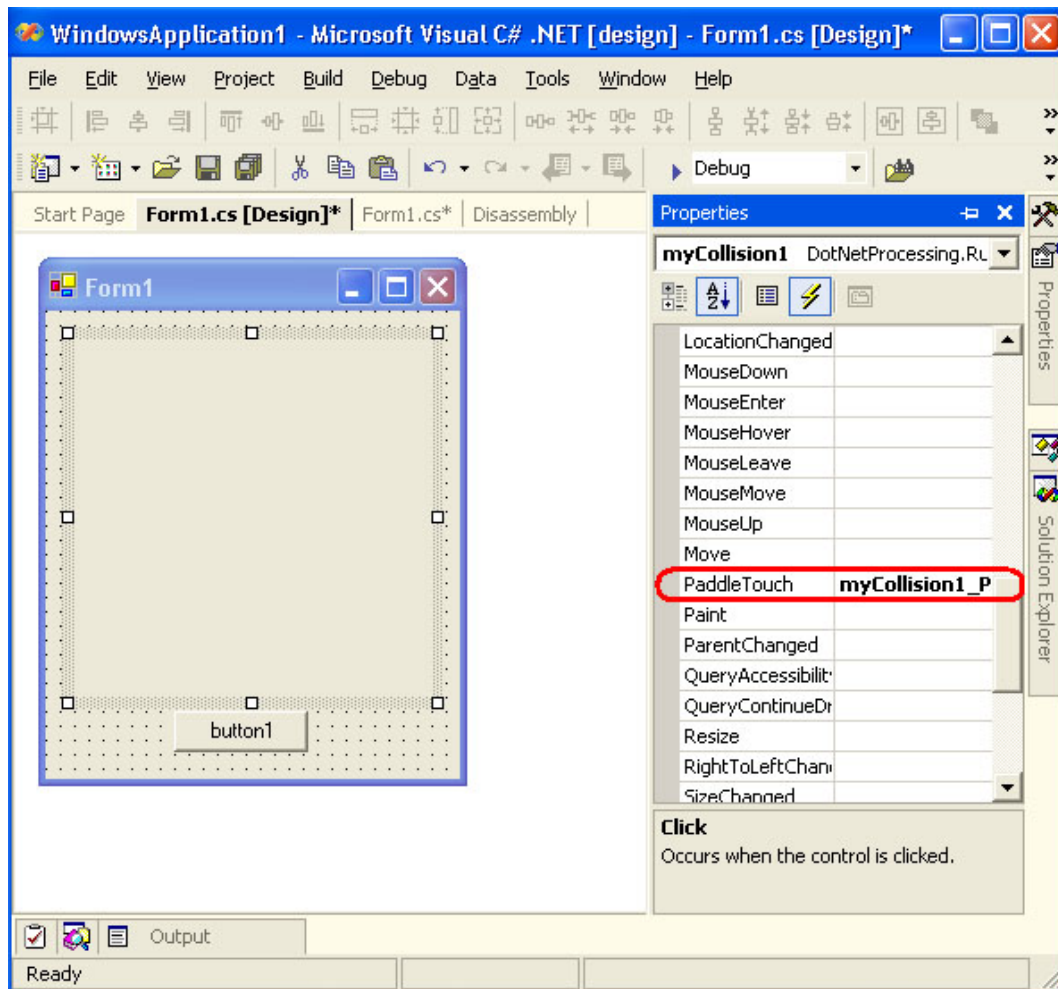
```
MessageBox.Show( "Good!!" );
```

10. Drop a button to the form and type the following in the click event:

```
myCollision1.Start();
```

11. Execute the program. You'll see a bigger paddle and a congratulations message everytime you hit the paddle.

-  You don't need to create properties to access your variables from a .NET application. Prefix the *public* keyword before them in your sketch code and they will be accessible outside the sketch. Only they won't appear in the *property inspector*.
-  Have you noticed you can even call DotNetProcessing primitives from your application?



Events in your sketch.

5.2. Command Line

There is a special executable file called `dnf.exe` which lets you build, show and export sketches from the command line. Under Linux remember to use the **mono dnf.exe** command. This is the help information you will get when executing the program without parameters:

```

DotNetProcessing

Usage:

dnf.exe example.pde [syntax] [-e:export_type]

possible values for syntax:

jcsharp: Java Emulation (default for .pde)
csharp : C#              (default for .cs.pde)
jsharp  : J#

```

vb : VB.NET (default for .vb.pde)

possible values for export_type:

0: executable
1: web
2: .net user control

* When exporting, the destination files are put in a directory
with the same name as the file name that contains the code

Chapter 6. A Case Study

[...]

II. Developer Documentation

Table of Contents

- 7. [Getting the sources](#)
 - 7.1. [The easy way \(download them\)](#)
 - 7.2. [The not so easy way \(get them by cvs\)](#)
 - 7.2.1. [Windows](#)
 - 7.2.2. [Linux](#)
 - 8. [Compiling](#)
 - 8.1. [Windows](#)
 - 8.1.1. [Microsoft Visual Studio](#)
 - 8.1.2. [Command Line](#)
 - 8.2. [Linux](#)
 - 9. [Architecture](#)
 - 9.1. [DotNetProcessing building blocks](#)
 - 9.2. [The Syntax](#)
 - 9.3. [The Primitives](#)
 - 9.4. [The Surface](#)
 - 9.5. [Sketch Exportation](#)
 - 9.6. [Execution Model](#)
 - 10. [Cross-platform issues](#)
 - 11. [Dotnetprocessing vs Processing](#)
 - 11.1. [Functional differences](#)
 - 11.2. [Performance test](#)
 - 12. [TODO's](#)
 - 13. [Implementation status](#)
 - 14. [Writing documentation for DotNetProcessing with DocBook](#)
 - 15. [Updating the DotNetProcessing web site](#)
 - 16. [Roadmap](#)
 - 16.1. [Mozilla Plugin for .NET User Controls](#)
 - 16.2. [Support for Java and Visual Basic.NET syntax under Linux](#)
 - 16.3. [Live Processing](#)
 - 16.4. [GTK# Environment](#)
 - 16.5. [DotNetProcessing Arena](#)
 - 16.6. [Windows Vista](#)
 - 16.7. [DotNetProcessing for mobile devices](#)
-

Chapter 7. Getting the sources

7.1. The easy way (download them)

Get them in the [DotNetProcessing download page](#).

7.2. The not so easy way (get them by cvs)

DotNetProcessing sources are hosted in [Sourceforge](#) CVS servers. CVS is a tool used by many software developers to manage changes within their source code. That means many developers can work concurrently on the source code having the most up to date changes at any time. When one developer tries to update some changes on a file that previously has been modified by another developer, the cvs server alerts of this situations and the developer has to revise both changes and send a working version to the cvs server.

The best way to understand what all this is about is read some CVS information on the Sourceforge page:

- [DotNetProcessing Sourceforge CVS information](#)
- [Sourceforge CVS general information](#)
- [Sourceforge CVS software recommendations](#)
- [Sourceforge CVS windows access with Tortoise CVS](#)

There are basically two access modes to the sources (*anonymous* and *authenticated*). The difference is that authenticated access permits uploading changes to the cvs server. To be an authenticated user you have to create a Sourceforge account and ask for appropriate privileges to the DotNetProcessing administrators.

7.2.1. Windows

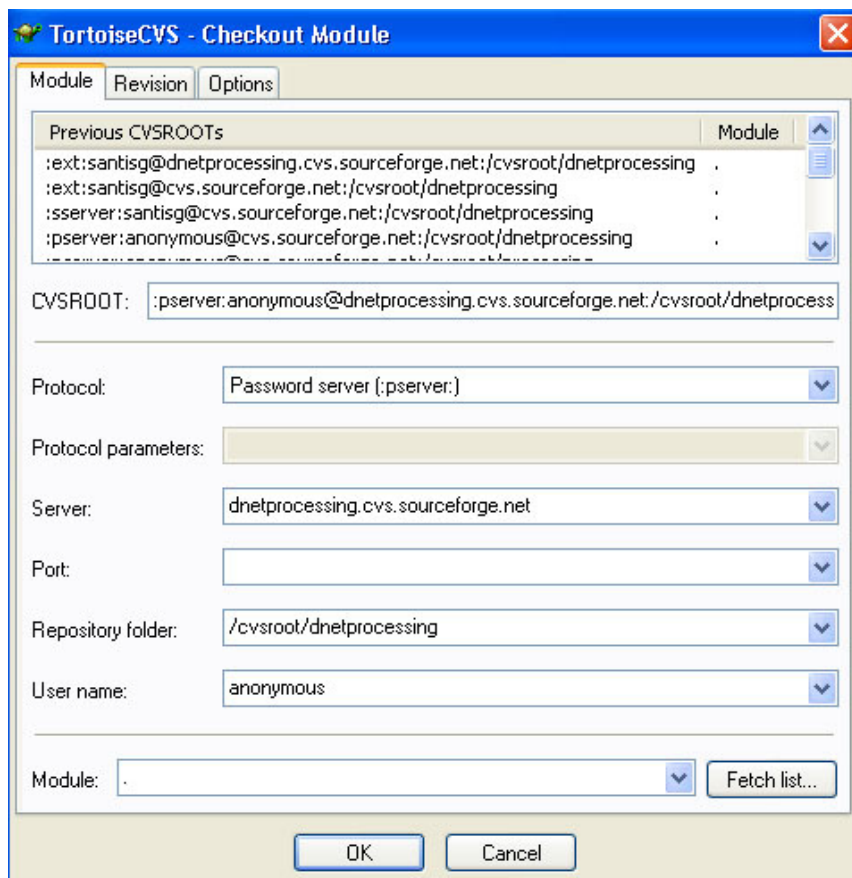
For quick access to the DotNetProcessing sources with [Tortoise CVS](#) follow these steps:

1. Download and install Tortoise CVS
2. Create an empty folder anywhere on your system
3. Right mouse click on this folder
4. Choose the CVS Checkout option on the menu
5. Configure the options either for *anonymous* or *authenticated* access:

Table 7–1. CVS Access

	Anonymous	Authenticated
Protocol	Password server (:pserver:)	Secure Shell (:ssh:)
Server	dnetprocessing.cvs.sourceforge.net	dnetprocessing.cvs.sourceforge.net
Repository folder	/cvsroot/dnetprocessing	/cvsroot/dnetprocessing
Username	anonymous	your sourceforge username
Module	.	.

6. Press the OK button



Getting the sources with Tortoise CheckOut CVS anonymous access

7.2.2. Linux

[...]

Chapter 8. Compiling

8.1. Windows

8.1.1. Microsoft Visual Studio

Open the DotNetProcessing Solution filename: `DotNetProcessing.sln`. Visual Studio will open all associated projects with the solution. Compile as usual.

8.1.2. Command Line

Use one of the following commands to compile the sources:

build_csc: For compiling with the Microsoft .NET Framework C# compiler

build_mcs: For compiling with the Mono C# compiler

 If you have problems check that the `PATH` System Variable is properly set up.

8.2. Linux

Use the following command to compile the sources:

sh build.sh

Chapter 9. Architecture

9.1. DotNetProcessing building blocks

DotNetProcessing is formed of various modules, each one with its own purpose:

DotNetProcessing.Environment

This is the graphical user interface for DotNetProcessing. It contains a textbox for writing the sketch, a combobox for setting the syntax and other operations included in the menus. When the user compiles or exports the sketch, control is passed to the Parsing module.

DotNetProcessing.EnvironmentConsole

This is the equivalent console application to the graphical version of the Environment. Its functionalities are exposed through command line parameters.

DotNetProcessing.Parsing

In the Parsing module the user sketch code is embedded in the Running module and the resulting code is compiled with the corresponding .NET compiler instance depending on the associated syntax.

DotNetProcessing.Canvas

The Canvas is only a simple Windows Form that holds the UserControl sketch generated in the Running module. It's the main entry point of the executable file resulting of exporting the sketch to executable.

DotNetProcessing.Running

This module inherits from AbstractRunning and has one different version for each possible syntax accepted by the program. This is its main purpose and thus the code inside it is very short. The parsing module embeds the user sketch code here and compiles it with the appropriate compiler instance. This module corresponds to the resulting dll when exporting to .net user control.

DotNetProcessing.AbstractRunning

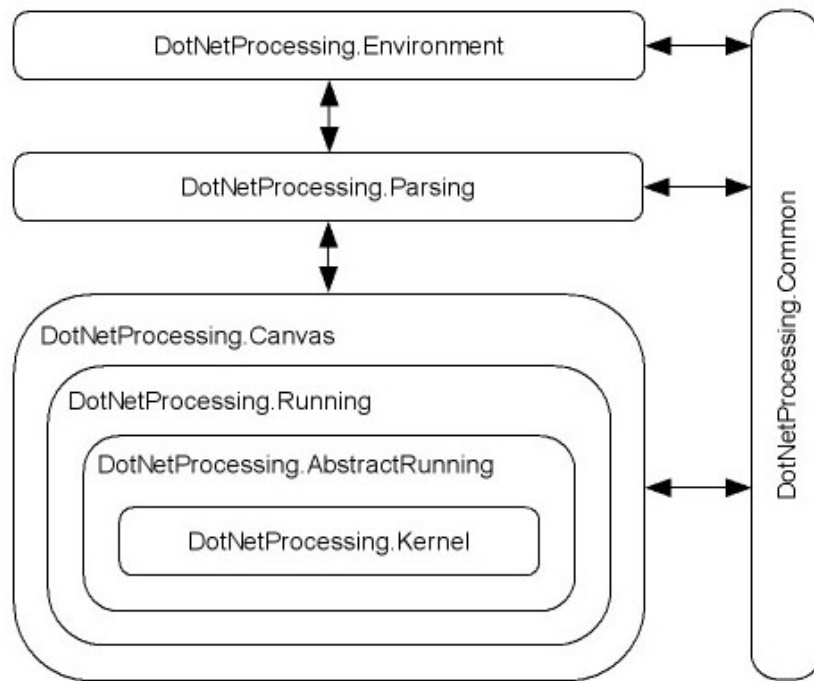
This module inherits from the Kernel and has the main loop of the program. The one that calls repeatedly to the user *draw* method and paints through the primitives called in the Kernel module.

DotNetProcessing.Kernel

The kernel module is a class that inherits from *System.Windows.Forms.UserControl* and implements all the Processing primitives using *GDI+*. Those primitives paint to the UserControl surface.

DotNetProcessing.Common

This module is used by all other modules. It contains constant definitions, events, exceptions and other stuff not related with any specific module.



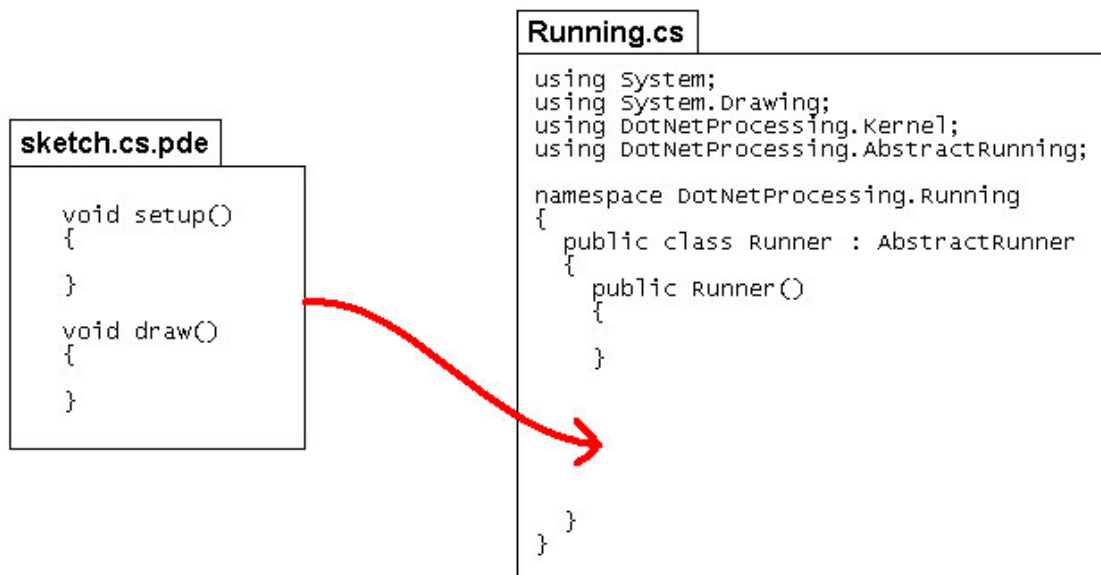
DotNetProcessing Modules

9.2. The Syntax

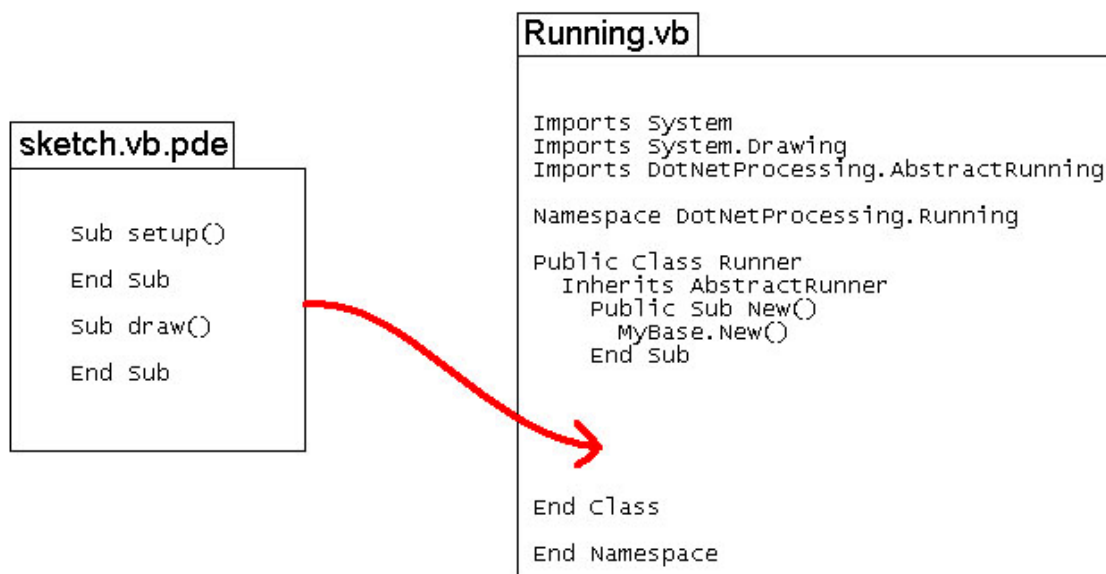
When we had the idea of porting the Processing language to the .NET platform one of the first things that came to our minds was "That's about compilers". In fact we needed to do something with code written in a specified language by the final user. So we began to write a grammar for the Processing language in a free YACC-like C# based tool called [Grammatica](#). But soon we saw that was a lot of work.

Fortunately, after doing a little research in the source code of the original Processing software we saw the approach followed was very different. The idea was to embed the sketch source code in a java class and compile the result with the java compiler. This has some disadvantages, mostly related with not having fine-grained control of the process in which the sketch is executed. But it has one big advantage, the dirty work is done by the underlying compiler (Java Virtual Machine in case of Processing and .NET in case of DotNetProcessing). That permitted developing a working solution in a very short period of time.

This approach had indirectly another advantage. By relaying on the underlying compiler for the sketch compiling process we can write sketches in different .NET supported syntaxes. A different compiler instance is used for every different syntax supported.



Sketch source code is embedded in the Runner class (C#)



Sketch source code is embedded in the Runner class (VB.NET)

9.3. The Primitives

Another important thing to address was how to implement the Processing primitives. We initially thought of three possibilities (GDI+, DirectX and OpenGL). After some research through them we saw *GDI+* had two big advantages. The first one is that GDI+ is included in the .NET framework. The second advantage is that many Processing primitives has an equivalent in GDI+. GDI+ does not have the power of more complex graphics libraries but is enough to implement the basic primitives. In the

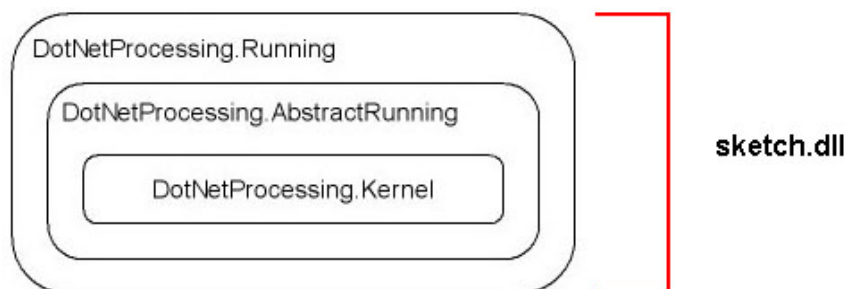
future, DirectX can be used to implement 3D as the original Processing language does with OpenGL.

9.4. The Surface

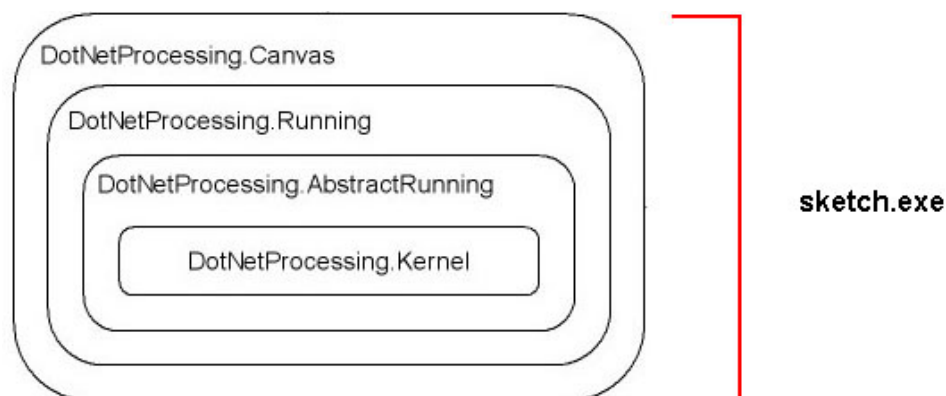
We needed a place to paint to. In GDI+ all painting is done to a *Graphics* object. The user control, which is in fact a form because it inherits from it, has an associated Graphics object that can be repainted in the *OnPaint* event. One of the problems we had to face was that changes in the graphics object are not permanent. This means that we needed a way of tracking all the changes that were applied to the surface in every loop. The solution we followed consisted in maintaining a *Bitmap* object with the surface content. Then in the *OnPaint* event this content is dropped to the user control graphics object through the *DrawImage* method.

9.5. Sketch Exportation

The way in that DotNetProcessing modules are designed makes very easy the process of exporting the sketch to one of its three options (executable, web and user control). When the user builds a sketch, internally, in the Parsing module the program generates a .NET user control containing the sketch. After this, exporting the sketch to .NET user control is as easy as copying the resulting dll to a specified folder. In case of web, the same dll is deployed along with an html file that uses it. Then both files have to be hosted in a web server to complete the process. In case of executable the Canvas module (which resulting type is of executable) is deployed. But in fact, it is just a container of the .NET user control. It is also the file that is executed when the user runs the sketch in the Environment.



Export to .NET User Control



Export to Executable

9.6. Execution Model

In this section we present in a chronological way the main tasks that occur when a sketch is compiled and executed:

Table 9–1. Execution model

<i>Compilation</i>	<i>Environment</i>	The user writes the sketch and clicks on the <i>Build and Run</i> button
	<i>Parsing</i>	The <i>size</i> primitive is located in the sketch code and once the sketch size is known it is transferred to the Kernel and the Canvas module setting the user control and the form dimensions according to the sketch size. Also the html used for exporting the sketch to web is modified for setting the sketch size.
		Some modifications are made to the sketch code before compiling it: <ul style="list-style-type: none"> • In those pair primitives that have the same name but with different behavior (one as variable and one as function), one of them is changed because as this is permitted in syntaxes like Java it is not permitted in others like C#. Examples of this are <i>framerate</i>, <i>keyPressed</i> and <i>mousePressed</i>. • User functions related with keyboard and mouse events like <i>keyReleased</i> or <i>mouseDragged</i> are prefixed with syntax specific keywords. For example, in case of C#, <i>void</i> is replaced by <i>public override void</i>.
		Source sketch code is embedded in the Running module
		The main classes are substituted with the sketch name so that when the user exports the sketch to .NET user control it is personalized. This is done in the Running and Canvas modules which correspond to the .NET user control and the executable.
		Kernel, AbstractRunning, Running, Canvas and Common modules are compiled generating the resulting sketch
<i>Execution</i>	<i>Canvas</i>	The sketch is executed
	<i>AbstractRunning</i>	The <i>setup</i> sketch method is processed
		We enter the main <i>loop</i> doing the following tasks: <ul style="list-style-type: none"> • Process the <i>draw</i> sketch method where all the painting is done though the <i>Kernel</i> module • Handle <i>frame rate</i> and sleep the process the corresponding time • Save <i>mouse position</i> for handling <i>pmouseX</i> and <i>pmouseY</i> primitives

Chapter 10. Cross-platform issues

Since DotNetProcessing can run both in Windows and Linux there are a few things you have to be specially carefull when writting code:

- Only write managed .NET code. That means, only use those classes included in the .NET class library. If you call to special Windows API functions you are writting unmanaged code which won't work under Linux.
- Use the `System.IO.Path.DirectorySeparatorChar` special variable when accessing the file system using paths.
- Don't make things like launching Internet Explorer.
- Try the program under both platforms periodically. It will help detecting possible problems.



At the time of writing this documentation the DotNetProcessing Environment, although very unstable, works under some Linux distributions. That's why we've make a very simple graphical interface without icons and other rich elements. More elements can be added as [Mono Winforms](#) evolve.

Chapter 11. Dotnetprocessing vs Processing

11.1. Functional differences

Dotnetprocessing is a port of the original language. But as they are implemented with different languages and in a different way, they have different features:

- The original language needs the Java Virtual Machine (JVM) installed in the client computer to running sketches, so that it was coded in Java. It makes it so much portable. Dotnetprocessing only can be executed in computers with .NET Framework or Mono, because it's implemented with .NET technology. That's why it's less portable than the original.
- Processing sketches can only be coded in Java language, while dotnetprocessing allows programming in some .NET languages: C#, J# and Visual Basic. This is one of the advantages of using our language: it's more flexible and can be used by more programmers.
- Processing is a multiplatform language and it can be displayed in almost all operating systems with graphical environment. It has this feature because it uses the JVM. As we had explained in the first paragraph, our language depends on .NET Framework and it makes it less multiplatform than the original.
- Actually, dotnetprocessing only works with Internet Explorer, while Processing can be executed at least with Explorer and Mozilla browsers. It's because of Mozilla doesn't have yet a plugin for execute our code.
- Our sketches can be exported to web controls, to DLL's (for use in all .NET applications) and to autoexecutable programs. The original language can export only web controls. This is one extra feature in favour of our port.
- Dotnetprocessing makes possible interactions between two or more sketches. We can implement a project using some sketches and catch events of one of them to make an action in other one. That functionality isn't present in the original environment.
- Processing is an older language and it's more developed than dotnetprocessing. It has more extra features like 3D graphics or sounds. It has a lot of libraries that can expand his features and makes it very powerful and easy to use. It has a great community that supports it. Our language is only a part of it, and it isn't yet so extended, but it's a good beginning of its alternative.
- Interactuate between sketches...

11.2. Performance test

One of the more important test of our project is to evaluate the performance of the two platforms. As dotnetprocessing uses the .NET Framework, that is embedded in the operating system and we don't need to run the Java Virtual Machine, it should be more efficient and faster. Next, we will show that we were right. To make these tests, we show in the sketch the average framerate and the average miliseconds that take long a loop. For these calculations, we use millis() primitive and framerate environment variable.

In the code of one of the examples, you can see how we do this calculations:

Example 11–1. Modified code to show performance

```
// Sine_Cosine
// by REAS

// Linear movement with sin() and cos().
// Numbers between 0 and PI*2 (TWO_PI which is roughly 6.28)
// are put into these functions and numbers between -1 and 1 are
// returned. These values are then scaled to produce larger movements.

// Updated 21 August 2002

int i = 45;
int j = 225;
float pos1 = 0;
float pos2 = 0;
float pos3 = 0;
float pos4 = 0;
int sc = 40;
int radio_bola = 100;

// PERFORMANCE -----
double mediams = 0;
double mediafr = 0;
int ii = 0;

void setup()
{
  size(500, 500);
  noStroke();
  smooth();

  // PERFORMANCE -----
  // framerate(60);
  PFont font;
  font = createFont("Tahoma",8);
  textFont(font, 12);
}

void draw()
{
  background(0);

  fill(51);
  rect(150, 150, 200, 200);

  fill(153);
  ellipse(pos1, 75, radio_bola , radio_bola );

  fill(255);
  ellipse(75, pos2, radio_bola , radio_bola );

  fill(153);
  ellipse(pos3, 425, radio_bola , radio_bola );

  fill(255);
  ellipse(425, pos4, radio_bola , radio_bola );
```

```

i += 10;
j -= 10;

if(i > 405) {
    i = 45;
    j = 225;
}

float ang1 = radians(i); // convert degrees to radians
float ang2 = radians(j); // convert degrees to radians
pos1 = width/2 + (2 * sc * cos(ang1));
pos2 = width/2 + (2 * sc * sin(ang1));
pos3 = width/2 + (2 * sc * cos(ang2));
pos4 = width/2 + (2 * sc * sin(ang2));

// PERFORMANCE -----

mediams = ((mediams * ii) + duration.TotalMilliseconds) / (ii + 1);
mediafr = ((mediafr * ii) + framerate) / (ii + 1);
ii++;

text("ms por iteracion: "+mediams.ToString(), 10, 10);
text("framerate: "+mediafr.ToString(), 10, 30);
}

```

We deactivate the frameset so that sketches can be runned as fast as it could be possible. So we can get the values when every platform is forced to be runned in it highest performance. The execution of every test was about one minute. With this time, the values of the variables had enough time to be stable.

Next, we show our first test: we executed the Bounce example with dotnetprocessing and with processing and we get these values:



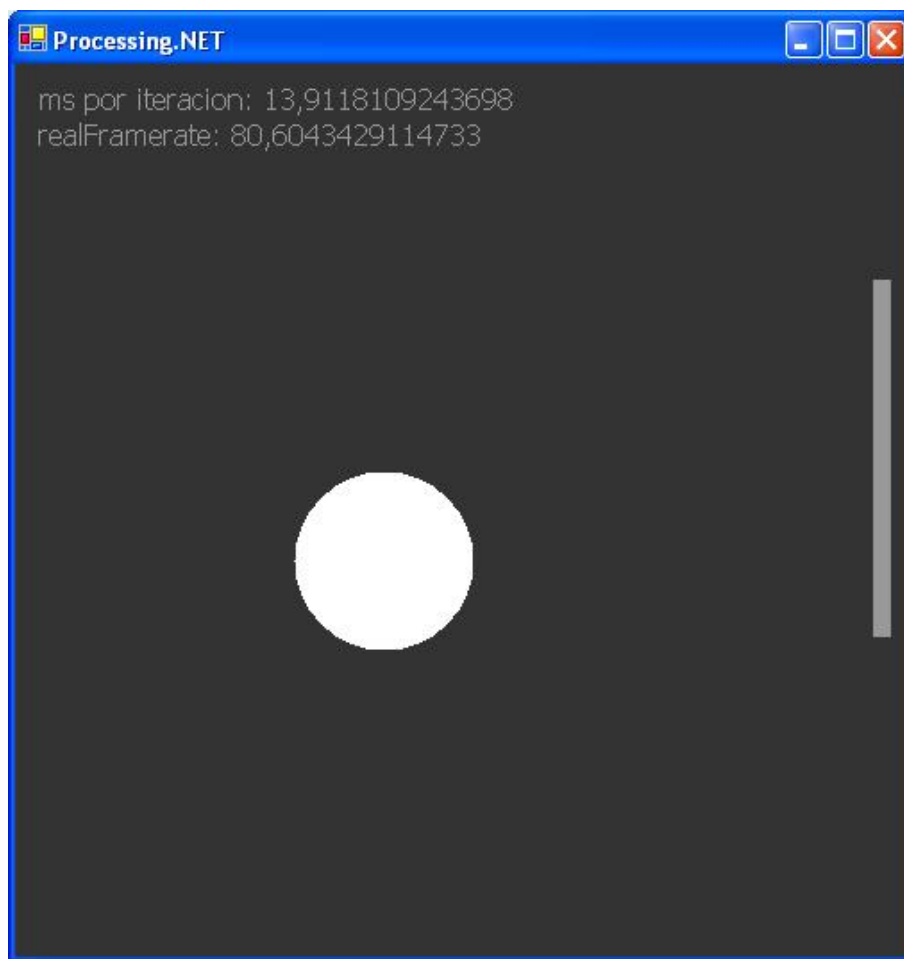
Bounce performance in dotnetprocessing



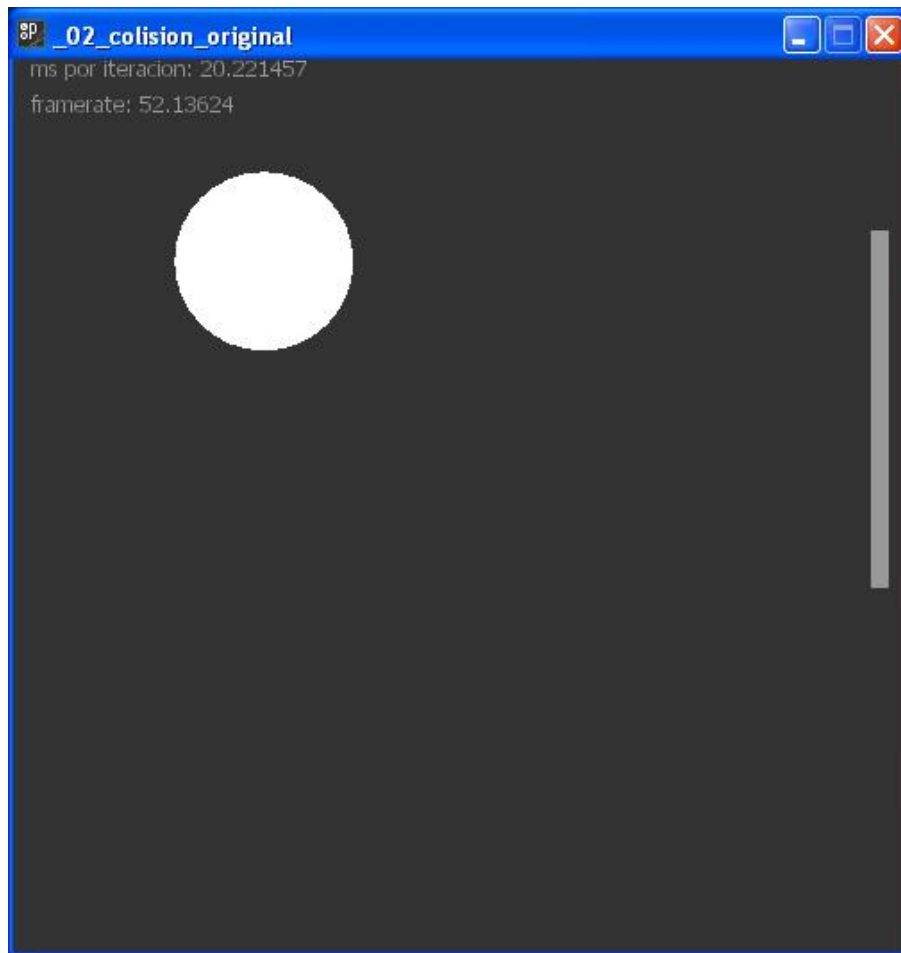
Bounce performance in processing

As we can see, dotnetprocessing is 12 milliseconds faster per loop, and so, it can display about 7 frames per second more.

Next example is Collision, executed in both platforms. This is an interactive sketch where we can play with the ball, as it was a ping pong game. We played with every sketch one minut and only lost one ball in every one.



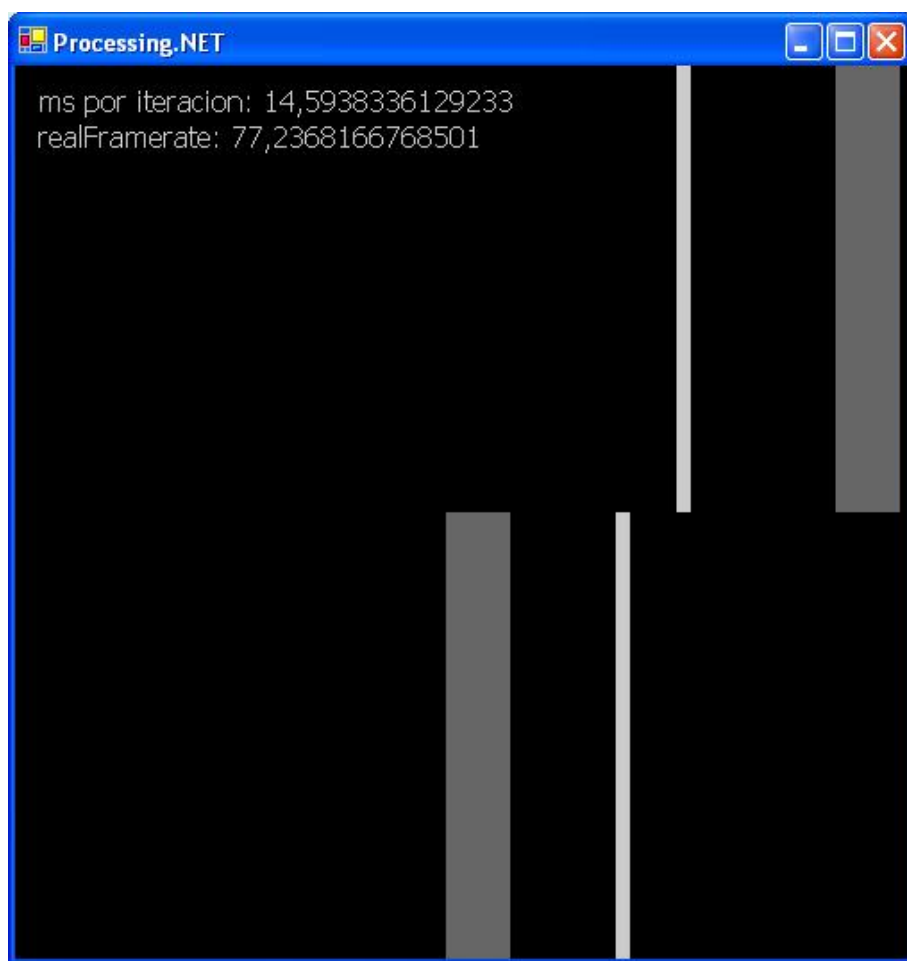
Collision performance in dotnetprocessing



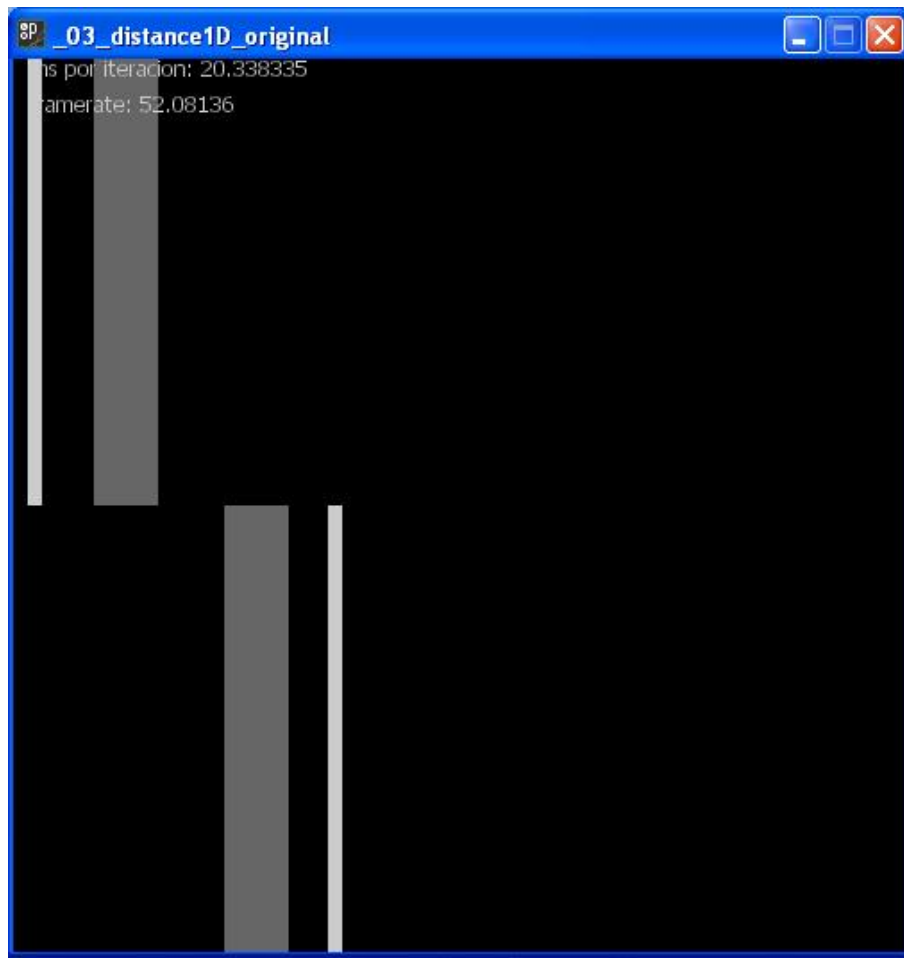
Collision performance in processing

In this test, we can see that our processing framerate was much more high that in the original language.

We made more tests to be sure that it was always more efficient, and it wasn't chance. The next example is Distance1D. The sketch is divided in 2 parts: the top and the bottom. The bars from the top are moving to the opposite side of the bottom ones. It also is interactive, and the movement of the mouse makes the bars be faster, and set de direction of the bars movements:

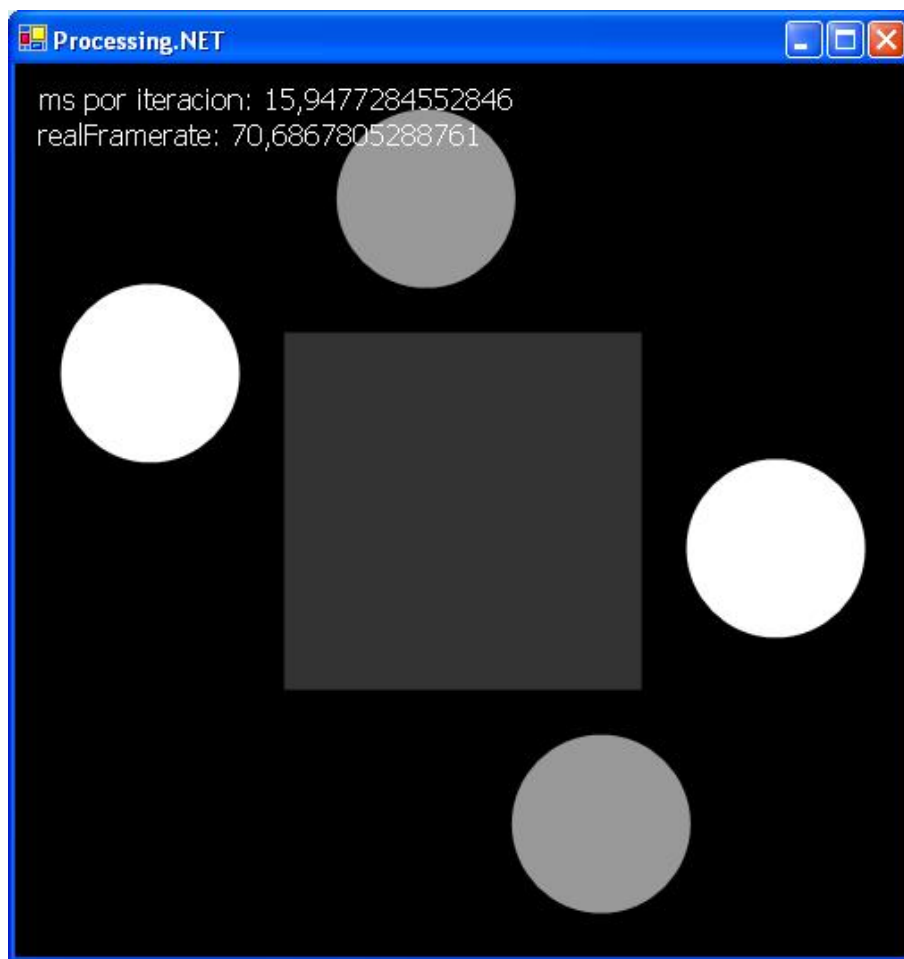


Distance1D performance in dotnetprocessing

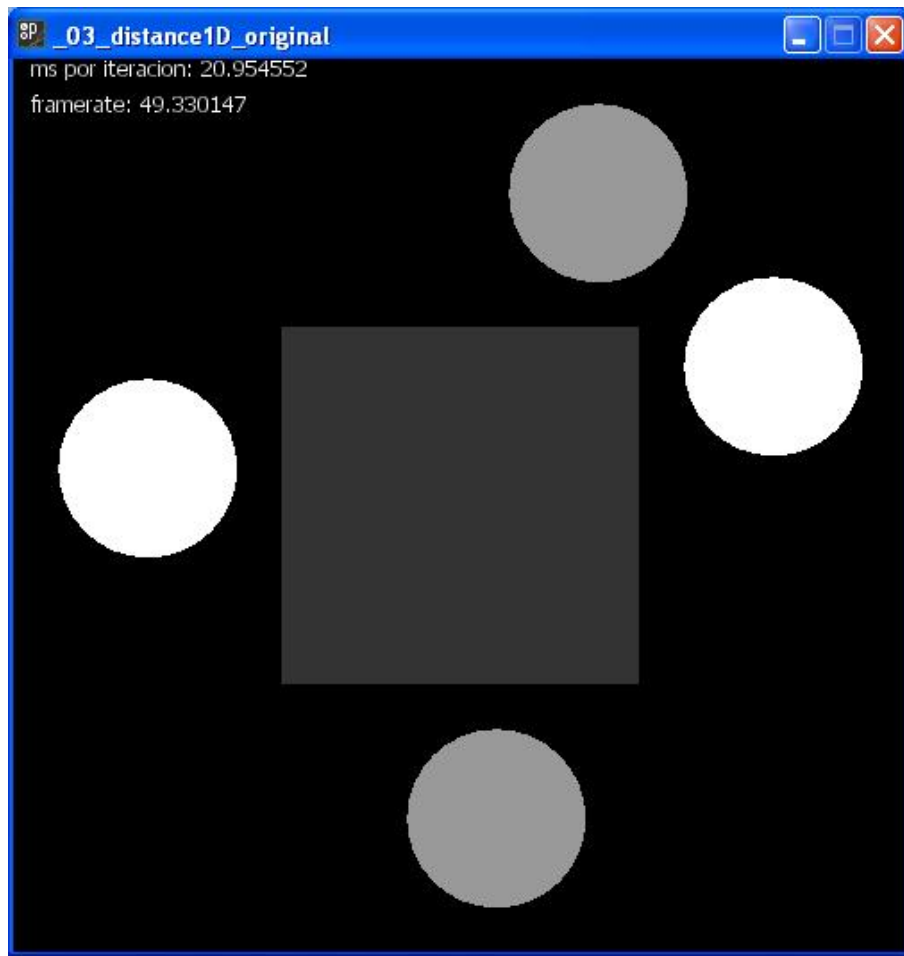


Distance1D performance in processing

The last test was made with SineCosine example. Here we have a fixed square and 4 balls around it. The balls are moved from side to side of the square. It was not a interactive sketch, but has 4 objects in movement at the time:



SineCosine performance in dotnetprocessing



SineCosine performance in processing

As we had proved, in all tests dotnetprocessing had a lower time of looping and a bigger framerate. Then, we can affirm that the executions with .NET Framework are more efficient than with Java Virtual Machine in all cases: interactives or not sketches.

Chapter 12. TODO's

We are going to present here a list of things which we think will be desirable to address in a short-term future:

- *Closing Internet Explorer:* When displaying a sketch through Internet Explorer the user has to press the Stop button in order to close the window. This is because while the main loop (which is located in the AbstractRunning module) is running on the client's machine, Internet Explorer events are not captured. The main loop has the Application.DoEvents(); but it doesn't solve the problem. The solution seems to be related with threads. If we put the main loop in a different thread the user can close Internet Explorer without problems, but unfortunately this has some other drawbacks. Since the control is located in one thread and the code that paints to that control is located in another thread random errors occur. This is a [documented issue](#). In Windows you can not paint from one thread to another.
 - *Exporting to one unique DLL or EXE file:* When exporting a sketch written in a syntax which is not C# the generated files include various DLL files. This is because when the sketch is written in C# the program creates programmatically one instance of the CSharp compiler and builds all the application in one unique EXE or DLL file. But when the sketch is written in another syntax the Running module has to be compiled with another compiler instance different from CSharp and then an independent DLL is created. The solution for this may be merging all the resulting DLL's in one unique DLL or EXE file. There is a suitable application for this called [ILMERGE](#) but it only runs in the v2.0 .NET Runtime and doesn't work with Mono.
 - *Sketches that need more privileges when running through Internet Explorer:*
 - *Decouple Syntax parsing:* Now, if we want to add a new syntax in which users can write their sketches (for example Delphi or C++), we have to modify the source code of the Parsing module to handle the new syntax with a new compiler. If there was a way to decouple all this and make the process of supporting new syntaxes more dynamic it would be great.
 - *Complete Kernel Primitives:*
 - *Complete Java Emulation:*
 - *No more source code distributed with the binaries:*
 - *Support for additional libraries:*
 - *Ask user for multiple dll exportation:*
 - *Ask user for namespace when exporting to .NET User Control*
-

Chapter 13. Implementation status

We have implemented some of the primitives (more than the 50%) of the original language, but as it's on beta versions, it changes continuously and the implementation becomes a bit complicated. Now, we are going to explain the main original language parts, and the level of implementation of every one of them:

- *Structure*: It defines the structure of the language. It depends on the programming language. It is completely implemented.
- *Environment*: There are variables and functions that allow customize some aspects of sketches, or get values, like framerate or width. The level of implementation is about 60%.
- *Data*: This part is used to define primitives and to make data conversions. It's related to the programming language and that's why it's partially implemented. Its level of implementation is about 35%.
- *Control*: It makes possible to do iterations, relations and conditional sentences. The base language of processing (or the new processing) completely implements it. So all the functionalities are implemented by it.
- *Shape*: It is one of the most important group of primitives. It allows drawing polygons, lines, points and curves in a simple way. We have ported to the .NET platform 80% of primitives approximately.
- *Input / Output*: These primitives catch key and mouse events, let to print information in the debug window, and it also load and save files of bytes and strings. With them, sketches can interact with users. We have implemented 60% of primitives in the new environment.
- *Transform*: It also is one of the basic part of the core of processing. With these functions, we can change the properties of shapes, like scale or angle of rotation. Almost all the animated sketches used it. Due to its importance, the level of implementation is about 90%.
- *Color*: We can use these primitives to fill polygons or to change strokes or lines colours. We have full control to make all possible colours. We have coded about 50% of primitives.
- *Image*: This part essentially let load, display and modify images of many types. We only have redesign about 30% of primitives.
- *Typography*: With these functions, we can make words and phrases and we can display it in sketches in a very simple way. We can change text attributes. The level of implementation of this part is about 25%.
- *Math*: Some of the compositions needed to make complex calculations. This group of primitives allows doing it. We have fully translated all the functions.
- *Sound*: It let storing and playing sounds of the wave format. We discard it because it's one of the newest features of the original language, and we focussed in essential primitives.

Chapter 14. Writing documentation for DotNetProcessing with DocBook

According to [Dave Pawson's DocBook FAQ](#) DocBook is:

DocBook provides a system for writing structured documents using SGML or XML. It is particularly well-suited to books and papers about computer hardware and software, though it is by no means limited to them.

—Dave Pawson's DocBook FAQ

The good thing about DocBook is that when you write documentation you do exactly this, writing documentation. You don't care about the format, only about the contents. At any time you can apply a styleshet to your document and generate the resulting HTML, PDF, PS, CHM, RTF, TXT, etc.

The source docbook code for DotNetProcessing documentation (what you are reading now) can be seen [here](#). It is also available by CVS for updating purposes. See the [CVS DotNetProcessing documentation section](#). [GemDoc](#) is used to generate the final documents.

Read more about DocBook on the following links:

- [The official home page for "DocBook: The Definitive Guide" O'Reilly book.](#)
 - [The DocBook Project](#)
 - [The DocBook Technical Committee](#)
 - [A DocBook Tutorial](#)
-

Chapter 15. Updating the DotNetProcessing web site

At the time of writing this documentation the DotNetProcessing web page is hosted on Sourceforge servers and does not use any of its [php](#) or [mysql](#) capabilities. The idea is to migrate the web page to another server capable of running [ASP.NET](#) code. This would help to elaborate the [DotNetProcessing Arena](#). One of the options may be the [GotDotNet Workspaces](#) or some [free Mono hosting](#).

The source web code for the DotNetProcessing web page is available by CVS for updating purposes. See the [CVS DotNetProcessing documentation section](#).

Chapter 16. Roadmap

This document is intended to outline some mid and long term objectives to be achieved with DotNetProcessing

16.1. Mozilla Plugin for .NET User Controls

By now there is no such plugin that would be perfect for running DotNetProcessing sketches in [Mozilla](#) based internet browsers.

16.2. Support for Java and Visual Basic.NET syntax under Linux

Since [Mono](#) only has support for C# we can't process Java and Visual Basic.NET syntax sketches under Linux.

Some research through [IKVM](#) may be the solution for Java. [Visual Basic.NET mono compiler](#) is under development.

16.3. Live Processing

Live Processing is a live linux distribution that can run from a CD and has everything set up for beginning to work with DotNetProcessing.

Live Processing could be based on [Monoppix](#) which is based on [Knoppix](#) which is based on [Debian](#).

16.4. GTK# Environment

This would allow a unified rich multiplatform graphical interface for DotNetProcessing. It may not be necessary since [Mono WinForms](#) are being developed pretty fast.

16.5. DotNetProcessing Arena

DotNetProcessing Arena intends to be a web based interface for the DotNetProcessing environment. This would allow a user to test their own "on the fly" sketches through the DotNetProcessing web site.

16.6. Windows Vista

[Windows Vista](#) is the scheduled next version of Microsoft Windows operating system, superseding Windows XP. In this version, all the graphics subsystems have been rewritten and many things will change for developers writing programs that deal with graphics, and DotNetProcessing is one of them.

Basically, there are two DotNetProcessing modules that might be affected by Windows Vista additions:

DotNetProcessing.Environment

This module makes use of WinForms controls included in the System.Windows.Forms namespace. On Windows Vista, controls will be much richer and with great 3D capabilities. Some new namespaces hanging from System.Windows will include the classes to instantiate them. Another important thing is that on Windows Vista, user interfaces will be completely independent of the application source code. They will be defined in a declarative XML based language called [XAML](#).

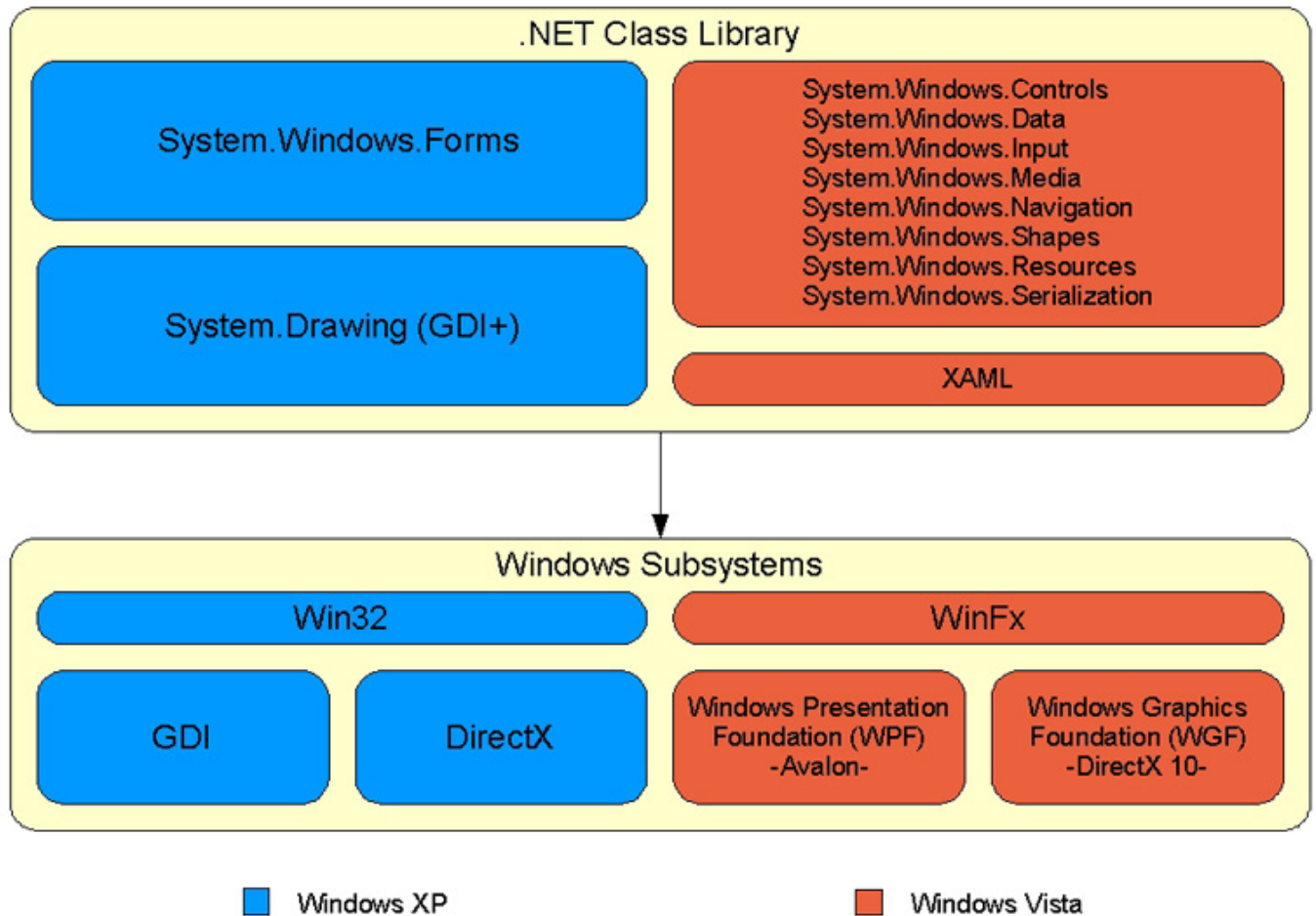
DotNetProcessing.Kernel

The Kernel is based completely in GDI+, the 2D .NET engine. Unfortunately GDI+ is only a wrapper of the old GDI engine which is based on the Win32 windows API. On Windows Vista, both 2D and 3D will be handled through the [Avalon](#) subsystem. Some new namespaces also hanging from System.Windows will include the corresponding classes. In the future might be interesting to rewrite this module according to the new Avalon subsystem.

The good news is that Windows Vista will be under development still for sometime and it will include full compatibility with both GDI+ and WinForms. Microsoft is doing a lot of effort to transmit the message that this will be true. And even there will be a mechanism called [Crossbow](#) to integrate WinForms controls inside Avalon Controls and vice versa.

Anyway, it can be good to keep an eye on what's going on with all this new cool things that Windows Vista is preparing for us and see how DotNetProcessing can incorporate them. Maybe we can see some day XAML user interfaces embedded in a DotNetProcessing sketch.

The mono guys have also included a few lines on their [roadmap](#) relating to how Windows Vista can affect their project.



Graphics in Windows Vista

16.7. DotNetProcessing for mobile devices

There is already a project focused on porting DotNetProcessing to the .NET Compact Framework. It's in an early stage but when finished it will be possible to run processing code in devices such as PDAs or mobile phones.

Generated by the free version of [GemDoc](http://www.gemdoc.net). Purchase now at www.gemdoc.net/purchase
 DocBook Made Easy – A single source, Windows based, multiple format solution for your document needs.