Media Processing in Processing

Collection Editor:
Davide Rocchesso

Media Processing in Processing

Collection Editor:
Davide Rocchesso

Authors:
Anders Gjendemsjø
Pietro Polotti
Davide Rocchesso

Online:
< http://cnx.org/content/col10268/1.14/ >

C O N N E X I O N S

Rice University, Houston, Texas

Table of Contents

Chapter 1

Programming in Processing[1]

1.1 Introduction

NOTE: This introduction is based on Daniel Shiffman's tutorial [2] .

Processing is a language and development environment oriented toward **interaction design** . In the course **Media Processing in Processing (MPP)**, processing is one of the main instruments used to introduce some fundamentals in sound and image processing. Processing is an extension of Java that supports many Java structures with a simplified syntax.

Processing can be used in three

Programming Modes

Basic - Sequence of commands for simple drawing by graphic primitives. –

applet without nose[3]	`size(256,256);`
	`background(0);`
	`stroke(255);`
	`ellipseMode(CORNER);`
	`ellipse(72,100,110,130);`
	`triangle(88,100,168,100,128,50);`
	`stroke(140);`
	`strokeWeight(4);`
	`line(96,150,112,150);`
	`line(150,150,166,150);`
	`line(120,200,136,200);`

Table 1.1

Intermediate - Procedural programming –

[1] This content is available online at <http://cnx.org/content/m12968/1.8/>.
[2] http://www.shiffman.net/itp/classes/ppaint/
[3] http://cnx.org/content/m12968/latest/pinocchiononose.html

applet with nose[4]	
	```
	void setup() {
	  size(256,256);
	  background(0);
	}

	void draw() {
	  stroke(255);
	  strokeWeight(1);
	  ellipseMode(CORNER);
	  ellipse(72,100,110,130);
	  triangle(88,100,168,100,128,50);
	  stroke(140);
	  beginShape(TRIANGLES);
	  vertex(114, 180);
	  vertex(mouseX, mouseY);
	  vertex(140, 180);
	  endShape();
	  strokeWeight(4);
	  line(96,150,112,150);
	  line(150,150,166,150);
	  line(120,200,136,200);
	}
	```

Table 1.2

Complex - Object-Oriented Programming (Java) –

[4]http://cnx.org/content/m12968/latest/pinocchionose.html

applet with colored nose[5]

```
Puppet pinocchio;

void setup() {
  size(256,256);
  background(0);
  color tempcolor = color(255,0,0);
  pinocchio = new Puppet(tempcolor);
}
void draw() {
  background(0);
  pinocchio.draw();
}
class Puppet  {
  color colore;
  Puppet(color c_) {
   colore = c_;
  }
  void draw () {
   stroke(255);
   strokeWeight(1);
   ellipseMode(CORNER);
   ellipse(72,100,110,130);
   stroke(colore);
   beginShape(TRIANGLES);
   vertex(114, 180);
   vertex(mouseX, mouseY);
   vertex(140, 180);
   endShape();
   strokeWeight(4);
   line(96,150,112,150);
   line(150,150,166,150);
  }
}
```

Table 1.3

The Processing programs can be converted into Java applets. In order to do that, one just goes to the **File** menu and chooses **Export**. As a result, five files will be created and put in an `applet` folder:

- **index.html** - html code to visualize the applet
- **filename.jar** - the compiled applet, including all data (images, sounds, etc.)
- **filename.pde** - the Processing source code
- **filename.java** - the Java code embedding the Processing source code
- **loading.gif** - an image to be displayed while the applet is being loaded.

Moreover, by means of **Export Application** it is possible to generate an executable application for Linux, MacOS, or Windows platforms.

[5]http://cnx.org/content/m12968/latest/pinocchioclassy.html

1.2 Data Types

1.2.1 Variables

A variable is a pointer to a memory location, and it can refer either to primitive values (`int`, `float`, ecc.) or to objects and arrays (tables of primitive-type elements).

The operation of assignment `b = a` produces

- The copy of the content of `a` into `b`, if the variables refer to primitive types.
- The creation of a new reference (pointer) to the same object or array, if the variables refer to objects or arrays.

NOTE: To have a clear understanding of computer science terms such as those that follow, we recommend looking at Wikipedia[6]

Definition 1.1: scope
within a program, it is a region where a variable can be accessed and its value modified

Definition 1.2: global scope
defined outside the methods setup() and draw(), the variable is visible and usable anywhere in the program

Definition 1.3: local scope
defined within a code block or a function, the variable takes values that are local to the block or function, and any values taken by a global variable having the same name are ignored.

Example 1.1: Array declaration and allocation

```
int[] arrayDiInteri = new int[10];
```

1.3 Programming Structures

1.3.1 Conditional Instructions

- if:

```
if (i == NMAX) {
   println("finished");
   }
else {
   i++;
   }
```

1.3.2 Iterations

- while:

```
int i = 0; //integer counter
while (i < 10) { //write numbers between 0 and 9
```

[6]http://wikipedia.org/

```
    println("i = "+ i);
    i++;
  }
```

- for:

```
  for (int i = 0; i < 10; i++) { //write numbers between 0 and 9
    println("i = "+ i);
  }
```

Example 1.2: Initializing a table of random numbers

```
int MAX = 10;
float[] tabella = new float[MAX];
for (int i = 0; i < MAX; i++)
   tabella[i] = random(1); //random numbers between 0 and 1
println(tabella.length + " elements:");
        println(tabella);
```

1.4 Functions

Functions allow a modular approach to programming. In Processing, in the **intermediate** programming mode, we can define functions other than `setup()` and `draw()`, usable from within `setup()` and `draw()`.

Example 1.3: Example of function

```
int raddoppia(int i) {
   return 2*i;
}
```

A function is characterized by the entities (with reference to the example (Example 1.3: Example of function)) :

- return type (`int`)
- name (`raddoppia`)
- parameters (`i`)
- body (`return 2*i`)

1.5 Objects and Classes

A class is defined by a set of data and functions. An object is an instance of a class. Vice versa, a class is the abstract description of a set of objects.

NOTE: For an introduction to the concepts of object and class see Objects and Classes[7].

Example 1.4: Example of class

```
Dot myDot;
void setup() {
  size(300,20);
  colorMode(RGB,255,255,255,100);
  color tempcolor = color(255,0,0);
  myDot = new Dot(tempcolor,0);
}

void draw() {
  background(0);
  myDot.draw(10);
}

class Dot
{

  color colore;
  int posizione;

  //****CONSTRUCTOR*****//
  Dot(color c_, int xp) {
    colore = c_;
    posizione = xp;
  }

  void draw (int ypos)    {
    rectMode(CENTER);
    fill(colore);
    rect(posizione,ypos,20,10);
  }
}
```

A class is characterized by the following entities (with reference to the example (Example 1.4: Example of class)) :

- name (`Dot`)
- data (`colore, posizione`)
- constructor (`Dot()`)
- functions (or methods, `draw()`)

[7]"Objects and Classes" <http://cnx.org/content/m11708/latest/>

An object (instance of a class) is declared in the same way as we declare a variable, but we have to allocate a space for it (as we did for the arrays) by means of its constructor (with reference to the example (Example 1.4: Example of class)).

- Declaration: (`Dot myDot;`)
- Allocation: (`myDot = new Dot(tempcolor,0)`)
- Use: (`myDot.draw(10);`)

NOTE: For a quick introduction to the Java syntax see Java Syntax Primer[8]

Exercise 1.1 ***(Solution on p. 11.)***

With the following `draw()` method we want to paint the window background with a gray whose intensity depends on the horizontal position of the mouse pointer.

```
void draw() {
    background((mouseX/100)*255);
  }
```

However, the code does not do what it is expected to do. Why?

Exercise 1.2 ***(Solution on p. 11.)***

What does the following code fragment print out?

```
int[] a = new int[10];
a[7] = 7;
int[] b = a;
println(b[7]);
b[7] = 8;
println(a[7]);
int c = 7;
int d = c;
println(d);
d = 8;
println(c);
```

Exercise 1.3 ***(Solution on p. 11.)***

The following sketch generates a set of 100 moving circles and draws all chords linking the intersection points of all couples of intersecting circles.

```
/*

   Structure 3

   A surface filled with one hundred medium to small sized circles.
   Each circle has a different size and direction, but moves at the same slow rate.
   Display:
   A. The instantaneous intersections of the circles
   B. The aggregate intersections of the circles
```

[8]"Java Syntax Primer" <http://cnx.org/content/m11791/latest/>

```
   Implemented by Casey Reas <http://groupc.net>
   8 March 2004
   Processing v.68 <http://processing.org>

   modified by Pietro Polotti
   28 March, 2006
   Processing v.107 <http://processing.org>


*/

int numCircle = 100;
Circle[] circles = new Circle[numCircle];

void setup()
{
  size(800, 600);
  frameRate(50);
  for(int i=0; i<numCircle; i++) {
    circles[i] = new Circle(random(width),
      (float)height/(float)numCircle * i,
      int(random(2, 6))*10, random(-0.25, 0.25),
      random(-0.25, 0.25), i);
  }
  ellipseMode(CENTER_RADIUS);
  background(255);
}


void draw()
{
  background(255);
  stroke(0);


  for(int i=0; i<numCircle; i++) {
    circles[i].update();
  }
  for(int i=0; i<numCircle; i++) {
    circles[i].move();
  }
  for(int i=0; i<numCircle; i++) {
    circles[i].makepoint();
  }
  noFill();
}


class Circle
{
  float x, y, r, r2, sp, ysp;
  int id;

  Circle( float px, float py, float pr, float psp, float pysp, int pid ) {
    x = px;
    y = py;
    r = pr;
    r2 = r*r;
    id = pid;
    sp = psp;
    ysp = pysp;
  }

  void update() {
    for(int i=0; i<numCircle; i++) {
      if(i != id) {
        intersect( this, circles[i] );
      }
    }
  }

  void makepoint() {
     stroke(0);
     point(x, y);
  }

  void move() {
    x += sp;
    y += ysp;
    if(sp > 0) {
      if(x > width+r) {
        x = -r;
      }
    } else {
      if(x < -r) {
        x = width+r;
      }
    }
    if(ysp > 0) {
      if(y > height+r) {
        y = -r;
      }
    } else {
      if(y < -r) {
        y = height+r;
      }
    }
  }
}

void intersect( Circle cA, Circle cB )
```

```
{
  float dx = cA.x - cB.x;
  float dy = cA.y - cB.y;
  float d2 = dx*dx + dy*dy;
  float d = sqrt( d2 );

  if ( d>cA.r+cB.r || d<abs(cA.r-cB.r) ) {
    return; // no solution
  }

  //  calculate the two intersections between the two circles cA and cB, //
  //  whose coordinates are (paX, paY) and (pbX, pbY), respectively.     //

  stroke(255-dist(paX, paY, pbX, pbY)*4);
  line(paX, paY, pbX, pbY);
}
```

1. Complete the missing part that is expected to compute the intersections of the circles, in such a way to draw the chords linking the intersection points. It is possible to use the computation of intersection coordinates in a ad-hoc reference system (**"Circle-Circle Intersection"**), then converting the result into the Processing window coordinate system.
2. Make the chords time-variable by giving different speeds to different circles.

Exercise 1.4 ***(Solution on p. 13.)***
Make the sketch of Exercise 1.3 interactive. For example, make the circle displacement dependent on the horizontal position of the mouse.

Solutions to Exercises in Chapter 1

Solution to Exercise 1.1 (p. 7)
The variable `mouseX` is of `int` type, and the division it is subject to is of the integer type. It is necessary to perform a **type casting** from `int` to `float` by means of the instruction `(float)mouseX`.
Solution to Exercise 1.2 (p. 7)

```
7
8
7
7
```

Solution to Exercise 1.3 (p. 7)

```
/*

   Structure 3

   A surface filled with one hundred medium to small sized circles.
   Each circle has a different size and direction, but moves at the same slow rate.
   Display:
   A. The instantaneous intersections of the circles
   B. The aggregate intersections of the circles

   Implemented by Casey Reas <http://groupc.net>
   8 March 2004
   Processing v.68 <http://processing.org>

   modified by Pietro Polotti
   28 March, 2006
   Processing v.107 <http://processing.org>


*/

int numCircle = 100;
Circle[] circles = new Circle[numCircle];

void setup()
{
  size(800, 600);
  frameRate(50);
  for(int i=0; i<numCircle; i++) {
    circles[i] = new Circle(random(width),
      (float)height/(float)numCircle * i,
      int(random(2, 6))*10, random(-0.25, 0.25),
      random(-0.25, 0.25), i);
  }
  ellipseMode(CENTER_RADIUS);
  background(255);
}


void draw()
{
  background(255);
  stroke(0);


  for(int i=0; i<numCircle; i++) {
    circles[i].update();
  }
  for(int i=0; i<numCircle; i++) {
    circles[i].move();
  }
  for(int i=0; i<numCircle; i++) {
    circles[i].makepoint();
  }
  noFill();
}


class Circle
{
  float x, y, r, r2, sp, ysp;
  int id;

  Circle( float px, float py, float pr, float psp, float pysp, int pid ) {
    x = px;
    y = py;
    r = pr;
    r2 = r*r;
    id = pid;
    sp = psp;
    ysp = pysp;
  }

  void update() {
    for(int i=0; i<numCircle; i++) {
      if(i != id) {
        intersect( this, circles[i] );
      }
    }
  }

  void makepoint() {
     stroke(0);
     point(x, y);
  }
  void move() {
    x += sp;
    y += ysp;
    if(sp > 0) {
      if(x > width+r) {
        x = -r;
      }
    } else {
      if(x < -r) {
        x = width+r;
      }
    }
    if(ysp > 0) {
      if(y > height+r) {
        y = -r;
      }
    } else {
      if(y < -r) {
        y = height+r;
      }
    }
  }
}

void intersect( Circle cA, Circle cB )
{
  float dx = cA.x - cB.x;
  float dy = cA.y - cB.y;
  float d2 = dx*dx + dy*dy;
  float d = sqrt( d2 );

  if ( d>cA.r+cB.r || d<abs(cA.r-cB.r) ) {
    return; // no solution
  }

  float a = (cA.r2 - cB.r2 + d2) / (2*d);
  float h = sqrt( cA.r2 - a*a );
  float x2 = cA.x + a*(cB.x - cA.x)/d;
  float y2 = cA.y + a*(cB.y - cA.y)/d;

  float paX = x2 + h*(cB.y - cA.y)/d;
  float paY = y2 - h*(cB.x - cA.x)/d;
  float pbX = x2 - h*(cB.y - cA.y)/d;
  float pbY = y2 + h*(cB.x - cA.x)/d;

  stroke(255-dist(paX, paY, pbX, pbY)*4);
  line(paX, paY, pbX, pbY);
}
```

Solution to Exercise 1.4 (p. 10)

```
     /*

   Structure 3

   A surface filled with one hundred medium to small sized circles.
   Each circle has a different size and direction, but moves at the same slow rate.
   Display:
   A. The instantaneous intersections of the circles
   B. The aggregate intersections of the circles

   Implemented by Casey Reas <http://groupc.net>
   8 March 2004
   Processing v.68 <http://processing.org>

   modified by Pietro Polotti
   28 March, 2006
   Processing v.107 <http://processing.org>


*/

int numCircle = 100;
Circle[] circles = new Circle[numCircle];

void setup()
{
  size(800, 600);
  frameRate(50);
  for(int i=0; i<numCircle; i++) {
    circles[i] = new Circle(random(width),
      (float)height/(float)numCircle * i,
      int(random(2, 6))*10, random(-0.25, 0.25),
      random(-0.25, 0.25), i);
  }
  ellipseMode(CENTER_RADIUS);
  background(255);
}


void draw()
{
  background(255);
  stroke(0);

  if(mousePressed){
  for(int i=0; i<numCircle; i++) {
    circles[i].sp = mouseX*random(-5, 5)/width;
  }
  }

  for(int i=0; i<numCircle; i++) {
    circles[i].update();
  }
  for(int i=0; i<numCircle; i++) {
    circles[i].move();
  }
  for(int i=0; i<numCircle; i++) {
    circles[i].makepoint();
  }
  noFill();
}


class Circle
{
  float x, y, r, r2, sp, ysp;
  int id;

  Circle( float px, float py, float pr, float psp, float pysp, int pid ) {
    x = px;
    y = py;
    r = pr;
    r2 = r*r;
    id = pid;
    sp = psp;
    ysp = pysp;
  }

  void update() {
    for(int i=0; i<numCircle; i++) {
      if(i != id) {
        intersect( this, circles[i] );
      }
    }
  }

  void makepoint() {
     stroke(0);
     point(x, y);
  }

  void move() {
    x += sp;
    y += ysp;
    if(sp > 0) {
      if(x > width+r) {
        x = -r;
      }
    } else {
      if(x < -r) {
        x = width+r;
      }
    }
    if(ysp > 0) {
      if(y > height+r) {
        y = -r;
      }
    } else {
      if(y < -r) {
        y = height+r;
      }
    }
  }
}

void intersect( Circle cA, Circle cB )
{
  float dx = cA.x - cB.x;
  float dy = cA.y - cB.y;
  float d2 = dx*dx + dy*dy;
  float d = sqrt( d2 );

  if ( d>cA.r+cB.r || d<abs(cA.r-cB.r) ) {
    return; // no solution
  }

  float a = (cA.r2 - cB.r2 + d2) / (2*d);
  float h = sqrt( cA.r2 - a*a );
  float x2 = cA.x + a*(cB.x - cA.x)/d;
  float y2 = cA.y + a*(cB.y - cA.y)/d;

  float paX = x2 + h*(cB.y - cA.y)/d;
  float paY = y2 - h*(cB.x - cA.x)/d;
  float pbX = x2 - h*(cB.y - cA.y)/d;
  float pbY = y2 + h*(cB.x - cA.x)/d;

  stroke(255-dist(paX, paY, pbX, pbY)*4);
  line(paX, paY, pbX, pbY);
}
```

Chapter 2

Media Representation in Processing[1]

2.1 Visual Elements

2.1.1 Coordinates

In Processing, the representation of graphic objects is based on a cartesian 3D coordinate system, as displayed in Figure 2.1 (Coordinate system).

Coordinate system

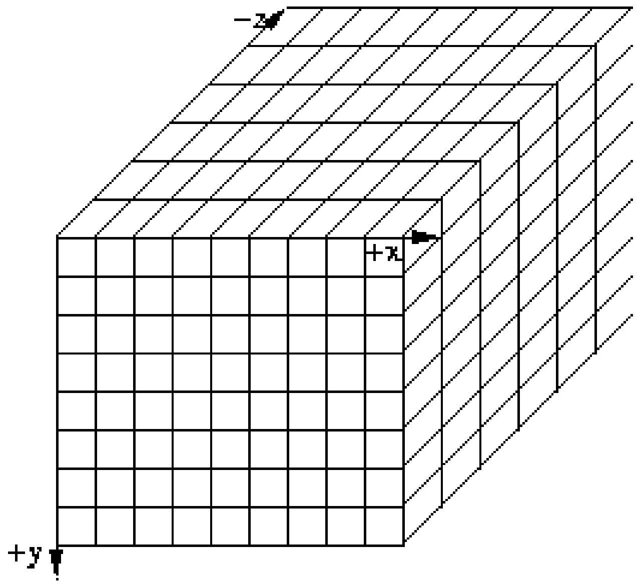


Figure 2.1: 3D coordinate system used in Processing

2D images are processed by acting on the X-Y plane, thus assuming that the Z coordinate is zero. The function `size()` defines the display window size and the **rendering engine** that will be used to paint onto the window. The default engine is JAVA2D, the 2D graphic Java libray. A bidimensional rendering engine,

[1]This content is available online at <http://cnx.org/content/m12983/1.13/>.

especially suitable for faster pixel-based image processing, is P2D (Processing 2D). If one wants to program in 3D, he must choose either the P3D (Processing 3D) rendering engine, especially suited for web-oriented graphics, or OPENGL, which delegates many typical 3D operations to the graphic board thus freeing the CPU from many computations. Moreover, if the objective is high-quality printing with vector graphics, a PDF rendering option is available.

2.1.2 Images

In Processing, an image can be assigned to an object of the class `PImage`. The function `loadImage("myImage")` takes a file (gif or jpg) `myImage`, containing the pixel coding of an image, and gives back the content of such image, which can be assigned to a variable of type `PImage`. The file `myImage` must be loaded in the `data` folder of the directory having the same name as the Processing sketch we are working at.

NOTE: When the `New` command is executed, processing opens up a folder named `sketch_???????` within the `Processing` directory, corresponding to the name assigned bye the system to the newly created file. Such folder is accessible from the Processing menu item `Sketch/Add File`.

The class `PImage` gives access, by the fields `width` and `height`, to the width and height of the loaded image. The image content is accessed via the `pixels[]` field.

Example 2.1: Loading and visualizing an image

```
size(400,300);
PImage b;
b = loadImage("gondoliers.jpg");
println("width=" + b.width + " height=" + b.height);
image(b, 0, 0, 400, 300); // position (0,0); width=400; height=300;
image(b, 20, 10, 100, 80); // position (20,10); width=100; height=80;
```

2.1.3 Colors

Since our color receptors (**cones**), each tuned to a wavelength region, are of three kinds, color models are always referred to a three-dimensional space. In additive color models, each of three axes correspond to a base color, and by mixing three colored light beams one can obtain all colors within a **gamut** volume in the space defined by the three axes. The three base colors can be chosen arbitrarily or, more often, based on the application domain (e.g., color of three phosphors or laser beams). In printing processes, subtractive color models are used, where the starting point is the white surface and primary ink colors are used to subtract color from white.

NOTE: Guide to color models: http://en.wikipedia.org/wiki/color_space[2]

In processing `color` is a primitive type used to specify colors. It is realized by a 32-bit number, where the first byte specifies the alpha value, and the other bytes specify a triple either in the RGB or in the HSB model. The choice of one model or the other is made by the `colorMode()` function. With three bytes, a number of $256 \times 256 \times 256 = 16777216$ are representable.

[2]http://en.wikipedia.org/wiki/color_space

2.1.3.1 The RGB model

Colors are represented by a triple of numbers, each giving the intensity of the primary colors Red, Green, and Blue. Each number can be an unsigned integer, thus taking values between 0 and 255, or be expressed as a floating point number between 0.0 and 1.0. With even larger flexibility, the model, type, and range of colors can be set with the function `colorMode()`. The RGB model is additive.

2.1.3.2 HSB Model

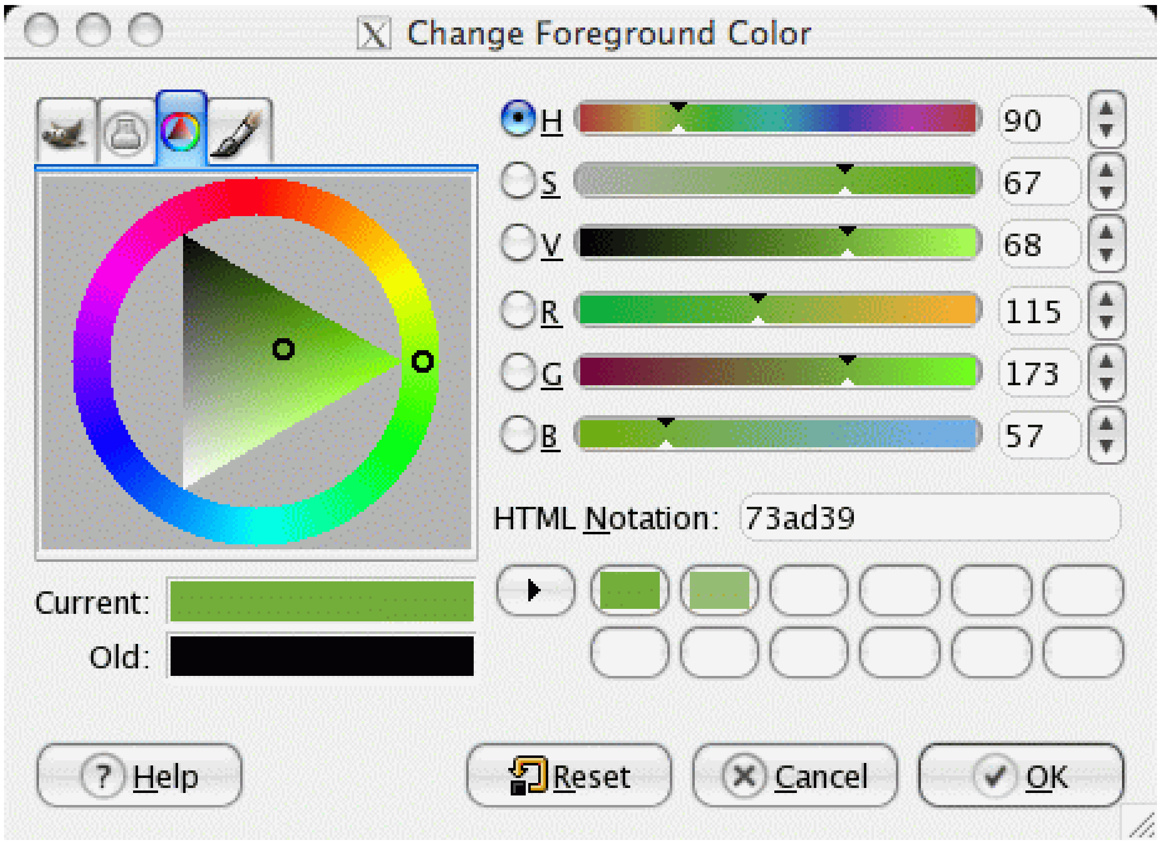
Colors are represented by a triple of numbers, the first number giving the Hue, the second giving Saturation, and the third giving the Brightness.

NOTE: Often the model is called HSV, where V stands for Value.

The hue takes values in degrees between 0 (red) and 360, being the various hues arranged along a circumference and being red positioned at 0° . Saturation and brightness vary between 0 and 100. The saturation is the degree of purity of color. If a pure color is added with a white light its degree of purity decreases until the color eventually sits on a gray scale when saturation is zero. In physical terms, the brightness is proportional to the signal power spectrum. Intuitively, the brightness is increased when the light intensity increases. The three-dimensional HSB space is well represented by a cylinder, with the hue (nominal scale) arranged along the circumference, the saturation (ratio scale) arranged along the radius, and the brightness (interval scale) arranged along the longitudinal axis. Alternatively, such three-dimensional space can be collapsed into two dimensions, as in the **color chooser** of the image-processing program Gimp[3] , displayed in Figure 2.2 (Gimp color chooser). Along the circumference, the three primary colors (red, green, and blue) are visible, 120° apart from each other, separated from the secondary colors (magenta, cyan, yellow). Each secondary color is complementary to the primary color in front of it in the circumference. For instance, if we take the green component out of a white light, we obtain a magenta light. The triangle inscribed in the circumference has a vertex pointing to a selected hue. The opposite side contains the gray scale, thus representing colors with null saturation and variable brightness. Going from the reference vertex to the opposite side we have a gradual decrease in saturation.

[3]http://www.gimp.org

Gimp color chooser



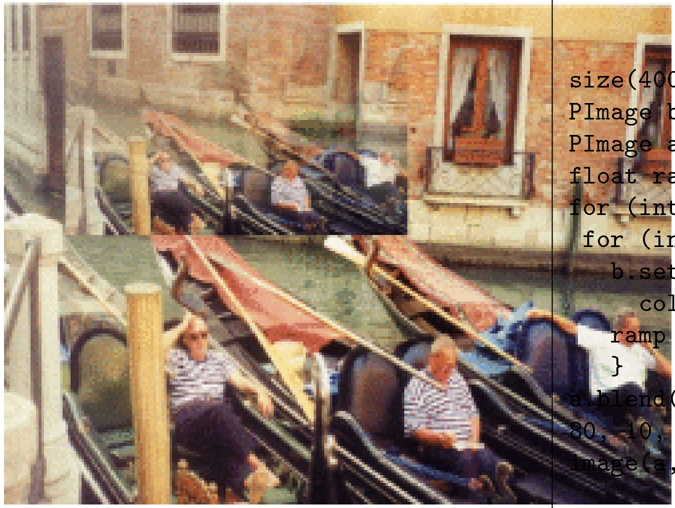
H 90
S 67
V 68
R 115
G 173
B 57
HTML Notation: 73ad39

Figure 2.2: Color chooser of the software Gimp

2.1.3.3 Alpha channel

It is a byte used to blend and interpolate between images, for example to render transparency. It can be obtained, from a variable of type `color`, with the method `alpha()`. The alpha channel can be manipulated with the method `blend()` of the class `PImage`.

Example 2.2: Loading and visualizing an image with transparency



```
size(400,300);
PImage b = loadImage("gondoliers.jpg");
PImage a = loadImage("gondoliers.jpg");
float ramp = 0;
for (int j = 0; j < b.height; j++)
 for (int i = 0; i < b.width; i++) {
   b.set(i, j, b.get(i,j) +
     color(0,0,0, 255 - (int)((1-ramp)*255)) );
   ramp = ramp + 1/(float)(b.width * b.height);
   }
a.blend(b, 0, 0, b.width, b.height,
30, 10, 450, 250, BLEND);
image(a, 0, 0, 400, 300);
```

Table 2.1

In Processing, it is possible to assign a color to a variable of type `color` by means of the function `color()`, and the model can be previously set with `colorMode()`. The functions `red()`, `green()`, `blue()`, `hue()`, `saturation()`, and `brightness()` allow to move from one model to the other.

```
colorMode(RGB);
color c1 = color(102, 30,29);
colorMode(HSB);
color c2 = color(hue(c1), saturation(c1), brightness(c1));
colorMode(RGB);
color c3 = color(red(c2), green(c2), blue(c2));
// the variables c1, c2, and c3 contain the coding of the same color
```

2.1.3.4 Tinging an image

An image can be tinged with a color and its transparency can be set by assigning a given value to the alpha channel. For this purpose, the function `tint()` can be used. For example, a blue tone can be assigned to the inlaid image of Example 2.1 (Loading and visualizing an image) by just preceding the second `image()` command with `tint(0, 153, 204, 126)` .

2.1.4 Translations, Rotations, and Scale Transformations

Representing Points and Vectors

In computer graphics, points and vectors are represented with the

Definition 2.1: homogeneous coordinates
quadruples of numbers, where the first triple is to be read in the X-Y-Z space, while the fourth number indicates a **vector** if it takes value 0, or a **point** if it takes value 1.

A translation is obtained by adding, in homogeneous coordinates, a vector to a point, and the result is a point. Alternatively we ca see a translation as a matrix-vector product (see Matrix Arithmetic[4]), where the

[4]"Matrix Arithmetic" <http://cnx.org/content/m10090/latest/>

matrix is $\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$, and the vector is the one representing the point $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$. An anti-clockwise rotation by the angle θ around the axis z (**roll**), is obtained by the rotation matrix $\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.
Rotations around the axes x (**pitch**) and y (**yaw**) are realized by means of rotation matrices of the same kind, and a rotation around an arbitrary axis can be obtained by composition (left multiply) of elementary rotations around each of the main axes.
Translations
The function `translate()` moves an object in the image window. It takes two or three parameters, being the displacements along the directions x, y (and z), respectively.
Rotations
In two dimensions, the function `rotate()` is used to rotate objects in the image window. This is obtained by (left) multiplying the coordinates of each pixel of the object by a rotation matrix. Rotations are always specified around the top left corner of the window ($[0, 0]$ coordinate). Translations can be used to move the rotation axis to other points. Rotation angles are specified in radians. Recall that $2\pi\text{rad} = 360^\circ$. For example, insert the rotation `rotate(PI/3)` before the second `image()` command in Example 2.1 (Loading and visualizing an image). In three dimensions, we can use elementary rotations around the coordinate axes `rotateX()`, `rotateY()`, e `rotateZ()`.
Scale Transformations
The function `scale()` allows to expand or contract an object by multiplication of its point coordinates by a constant. When it is invoked with two or three parameters, different scalings can be applied to the three axes.

2.1.5 Typographic Elements

Every tool or language for media manipulation gives the opportunity to work with written words and with their fundamental visual elements: typographic characters.

The aspect of a **type** has two main components: **font** and size.

Processing has the class `PFont` and the methods `loadFont()` (to load a font and assign it to an object of the `PFont` class) and `textFont()` (to activate a font with a specific size). In order to load a font, this has to be pre-loaded into the directory `data` of the current sketch. The tool `Create Font`, accessible from the `Tools` menu in Processing, creates the bitmaps of the characters that the programmer intends to use. The file with the bitmaps is put in the `data` directory. After these preliminary operations, the font can be used to write some text, using the function `text()`. With this function, a string of characters can be put in the 2D or 3D space, possibly inserting it within a rectangular box. The alignment of characters in the box is governed by the function `textAlign()`. In the default configuration, the written text can be spatially transformed like any other object. The color of characters can be set with the usual `fill()`, like for any other graphic object.

Example 2.3: Overlapped text

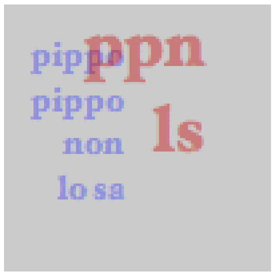
	`PFont fonte;`
	`/*The font have been previously created`
	`in the data folder*/`
	`fonte = loadFont("HoeflerText-Black-48.vlw");`
	`textFont(fonte, 12);`
	`fill(10, 20, 250, 80);`
	`textAlign(RIGHT);`
	`text("pippo pippo non lo sa", 10, 14, 35, 70);`
	`textFont(fonte, 24);`
	`fill(200, 0, 0, 100);`
	`text("ppnls", 25, 5, 50, 90);`

Table 2.2

Processing allows a tight control of the spatial occupation of characters and of the distance between contiguous characters (see Figure 2.3 (Typeface metrics)). The function `textWidth()` computes the horizontal extension of a character or a string. It can be used, together with the exact coordinates passed to `text()`, to control the **kerning** and the **tracking** between characters. The `textSize()` allows to redefine the size of characters. The `textLeading()` re-defines the distance in pixels between adjacent text lines. This distance is measured between the **baselines** of the strings of characters. Letters such as "p" or "q" extend below the baseline for a number of pixels that can be obtained with the `textDescent()`. Instead, the `textAscent()` gives back the maximum extension above the baseline (typically, the height of the letter "d").

Typeface metrics

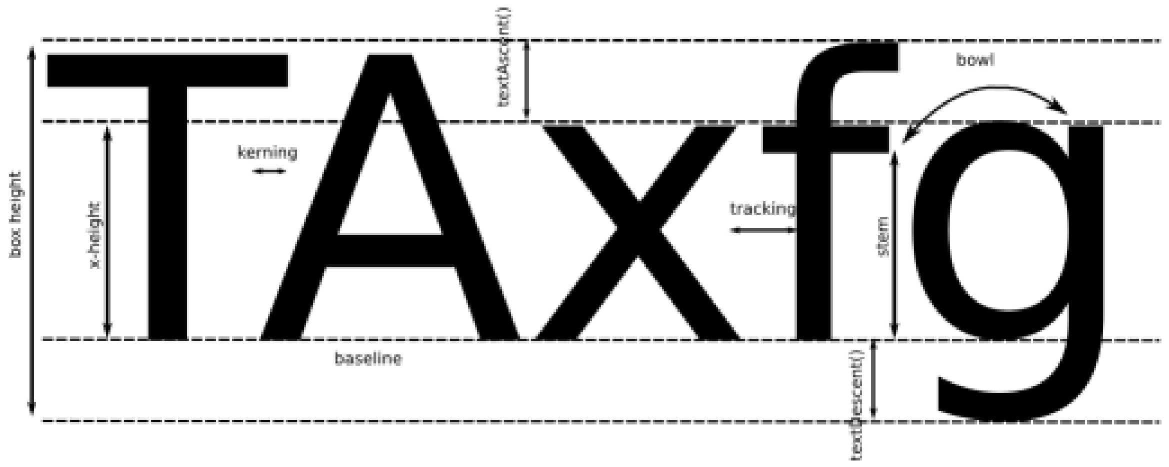


Figure 2.3: Typeface metrics

2.2 Auditory Elements

2.2.1 Sounds

Untill version beta 112, Processing gave the possibility to program several audio functionalities by means of some core primitives. In those older versions only two basic primitives are available to playback and load `.wav` files. In more recent versions, Processing delegate sound management and processing functionalities to external libraries[5] . The most used libraries are Ess[6] , Sonia[7] , and Minim[8] . Only the latter is included in the base installation of Processing. Ess and Sonia need an explicit installation process. Recently, a well-structured and documented Java library called Beads[9] has also been introduced. It is well suited to the construction of audio-processing algorithms based on chains of base objects. As in the case of images, in order to process and playback sounds the source files have to be stored in the `data` folder of the current sketch. The library Sonia [10] is the most complex one. With its functions, one can do **sample playback**, realtime Fourier-based spectral analysis, `.wav` file saving. In order to use the Sonia library, the programmer has to download the `.zip` file from Sonia[11] . Once decompressed, the directory `Sonia_?_?` has to be copied into the directory `Processing/libraries`. Finally, the command `import` has to be inserted into the code by selecting it from the menu item `Sketch / Import Library / Sonia_?_?`.

> NOTE: In order to run the applets produced with Sonia from a web browser, the Phil Burk's JSyn plugin has to be downloaded and installed from the site http://www.softsynth.com/jsyn/plugins/[12]
> .

The library Minim[13] , based on Java Sound[14] , is more user-friendly, well-documented and recommended, if one wants to work with sounds employing high-level primitives, without dealing with low-level numerical details and buffer management.

2.2.2 Timbre

In this section, we first use then analyze an application for the exploration of timbres, similar in conception to the Color Chooser of Figure 2.2 (Gimp color chooser), and here called Sound Chooser. For the moment, let us think about a sound timbre in analogy with color in images. For example, the various instruments of the orchestra have different and characterizing timbres (colors). Later on, we will define the physical and perceptual aspects of timbre more accurately. In the Sound Chooser applet, four sounds with different timbres can be played by clicking onto one of the marked radii. Each radius corresponds to a musical instrument (timbre/color). By changing position along the radius it is possible to hear how the brightness is changed. More precisely, as long as we proceed toward the centre, the sounds gets poorer.

Let us analyze the Processing code that implements the Sound Chooser in its salient aspects. The `Sonia.start(this)` command is necessary to activate the Sonia audio engine. The line `Sample mySample1` declares a variable aimed at containing audio samples. Several methods can be applied to such variable. Among these, the `play` methods plays the sound sample back. In the `draw()` code section the graphic aspect of the applet is defined. Finally, by the function `mouseReleased()`, we detect when the mouse is released after being pressed, and where it has been released. At this point a sequenceo of `if` conditions finds what instrument/timbre has been selected according to the clicking point. Moreover, within the function `mouseReleased()` the function `filtra(float[] DATAF, float[] DATA, float RO, float WC)`

[5]http://processing.org/reference/libraries/index.html
[6]http://www.tree-axis.com/Ess/
[7]http://sonia.pitaru.com/
[8]http://code.compartmental.net/tools/minim/
[9]http://www.beadsproject.net/
[10]http://sonia.pitaru.com/
[11]http://sonia.pitaru.com/
[12]http://www.softsynth.com/jsyn/plugins/
[13]http://code.compartmental.net/tools/minim/
[14]http://java.sun.com/j2se/1.5.0/docs/guide/sound/programmer_guide/contents.html

is invoked. This function, which is implemented in the last segment of the code listing, performs a sound filtering. More precisely, it is a low-pass filter, thus a filter that leaves the low frequencies unaltered and reduces the intensity of the high frequencies. According to the radial position of the mouse click, the filtering effect changes, being more dramatic (that is the sound becomes darker) as the mouse is released closer and closer to the centre. A lighter realization of the Sound Chooser by means of the library Minim is proposed in problem Exercise 2.4. The problem Exercise 2.5 explores the recent library Beads.

Trumpet[15]	
Oboe[16]	
Violin[17]	
Flute[18]	
Applet: choosing a timbre and controlling its brightness [19]	

```
import pitaru.sonia_v2_9.*;

Sample mySample1, mySample2, mySample3, mySample4;
Sample mySample1F, mySample2F, mySample3F, mySample4F;

float[] data1, data2, data3, data4;
float[] data1F, data2F, data3F, data4F;

int sr = 11025;  // sampling rate


void setup()
{
  size(200, 200);
  colorMode(HSB, 360, height, height);
  Sonia.start(this);

  mySample1 = new Sample("flauto.aif");
    mySample2 = new Sample("oboe.wav");
      mySample3 = new Sample("tromba.wav");
        mySample4 = new Sample("violino.wav");

  mySample1F = new Sample("flauto.aif");
 // ... OMISSIS ...

  data1  = new float[mySample1.getNumFrames()];
  // creates new arrays the length of the sample
  // for the original sound
 // ... OMISSIS ...

  data1F  = new float[mySample1.getNumFrames()];
  // creates new arrays the length of the sample
  // for the filtered sound
// ... OMISSIS ...


  mySample1.read(data1);
// ... OMISSIS ...

}

void draw()
{
// ... OMISSIS ...
}

void mouseReleased()
{

float ro;
float roLin;
float wc;


  // FLAUTO
```

Table 2.3

Exercise 2.1 ***(Solution on p. 28.)***

The content of a `PImage` object is accessible through its `pixels[]` field. The pixels, corresponding to a row-by-row reading of the image, are contained in this array of size `width*height`. Modify the code in Example 2.2 (Loading and visualizing an image with transparency) to use the field `pixels[]` instead of the method `get()`. The final outcome should remain the same.

Exercise 2.2 ***(Solution on p. 28.)***

Complete the code reported in Table 2.3 to obtain the complete Sound Chooser applet.

Exercise 2.3 ***(Solution on p. 28.)***

Add some color to the radii of the Sound Chooser, by replacing the `line` instructions with `rect` instructions and coloring the bars with a brightness that increases goint from the centre to the periphery.

Exercise 2.4 ***(Solution on p. 28.)***

Produce a new version of the Sound Chooser of problem Exercise 2.2 employing the library Minim. Note the gained compact form and simplicity of the code.

Exercise 2.5 ***(Solution on p. 30.)***

Produce a new version of the Sound Chooser of problem Exercise 2.2 using the Beads library. The signal-processing flow is particularly readable from the resulting code.

Exercise 2.6: Vectorial fonts ***(Solution on p. 33.)***

Processing programmers are encouraged to use bitmap fonts, encoded in a file with extension `.vlw`. This makes Processing independent from the fonts that are actually installed on a specific machine. However, it is possible to use vectorial fonts (e.g., `TrueType`) by inserting their files (e.g., with extension `.ttf`) in the `Data` folder. Try experimenting with vectorial fonts by using the `createFont()` function. If we give up the invariance of behavior on different machines, we can pass this function the name of a font that is installed on a specific computer and not found in the `Data` folder. Finally, under `JAVA2D` rendering mode, it is possible to use logical fonts, by passing `Serif`, `SansSerif`, `Monospaced`, `Dialog`, or `DialogInput` as a string that specifies the font as an argument of `createFont()`. Without the need of loading any font files in the `Data` folder, the correspondence between logical and physical fonts will be system dependent. Try experimenting with logical fonts on your computer.

[15] See the file at <http://cnx.org/content/m12983/latest/./tromba.wav>

[16] See the file at <http://cnx.org/content/m12983/latest/./oboe.wav>

[17] See the file at <http://cnx.org/content/m12983/latest/./violino.wav>

[18] See the file at <http://cnx.org/content/m12983/latest/./flauto.aif>

[19] See the file at <http://cnx.org/content/m12983/latest/./sound_chooser.html>

Solutions to Exercises in Chapter 2

Solution to Exercise 2.1 (p. 27)
The invocation `b.set()` should be replaced by

```
b.set(i,j,b.pixels[j*b.width+i]+ color(0,0,0, 255 - (int)((1-ramp)*255)) );
```

Solution to Exercise 2.2 (p. 27)
Processing source code. [20]
Solution to Exercise 2.3 (p. 27)
Applet with Processing source code. [21]
Solution to Exercise 2.4 (p. 27)

```
    import ddf.minim.*;
import ddf.minim.effects.*;

Minim minim;
AudioPlayer mySample1, mySample2, mySample3, mySample4;
LowPassSP lpf1, lpf2, lpf3, lpf4;
float cutoff1, cutoff2, cutoff3, cutoff4;

void setup()
{
  size(200, 200);
  colorMode(HSB, 360, height, height);
  minim = new Minim(this);

  mySample1 = minim.loadFile("flauto.aif");
    mySample2 = minim.loadFile("oboe.wav");
      mySample3 = minim.loadFile("tromba.wav");
        mySample4 = minim.loadFile("violino.wav");

  lpf1 = new LowPassSP(4000, mySample1.sampleRate());
  lpf2 = new LowPassSP(4000, mySample2.sampleRate());
  lpf3 = new LowPassSP(4000, mySample3.sampleRate());
  lpf4 = new LowPassSP(4000, mySample4.sampleRate());
  mySample1.addEffect(lpf1);
  mySample2.addEffect(lpf2);
  mySample3.addEffect(lpf3);
  mySample4.addEffect(lpf4);

}

void draw()
{
  stroke(255);
  strokeWeight(1);
  fill(0, 88, 88);
```

[20] See the file at <http://cnx.org/content/m12983/latest/./sound_chooser.pde>
[21] See the file at <http://cnx.org/content/m12983/latest/./sound_chooser_color.pde>

```
  ellipseMode(CORNER);
  ellipse(50,50,100,100);

  beginShape(LINES);
  vertex(50, 100);
  vertex(90, 100);

  vertex(110, 100);
  vertex(150, 100);

  vertex(100, 50);
  vertex(100, 90);

  vertex(100, 110);
  vertex(100, 150);
  endShape();
}

void mouseReleased()
{

  // FLUTE
  if ((mouseX > 95) && (mouseX < 105)&& (mouseY > 50)&& (mouseY < 90)) {
    cutoff1 = map(mouseY, 50, 90, 1000, 30);
    lpf1.setFreq(cutoff1);
    println(mouseY + " +  " +cutoff1);
    mySample1.rewind();
    mySample1.play();
  }

  // OBOE
  if ((mouseX > 110) && (mouseX < 149)&& (mouseY > 95)&& (mouseY < 105)) {
    cutoff2 = map(mouseX, 110, 149, 30, 1000);
    lpf2.setFreq(cutoff2);
    println(mouseX + " +  " +cutoff2);
    mySample2.rewind();
    mySample2.play();
  }

  // TRUMPET
  if ((mouseX > 95) && (mouseX < 105)&& (mouseY > 110)&& (mouseY < 149)) {
    cutoff3 = map(mouseY, 110, 149, 30, 1000);
    lpf3.setFreq(cutoff3);
    println(mouseY + " +  " +cutoff3);
    mySample3.rewind();
    mySample3.play();
  }

  // VIOLIN
  if ((mouseX > 50) && (mouseX < 90)&& (mouseY > 95)&& (mouseY < 105)) {
    cutoff4 = map(mouseX, 50, 90, 1000, 30);
    lpf4.setFreq(cutoff4);
    println(mouseX + " +  " +cutoff4);
    mySample4.rewind();
    mySample4.play();
  }
}

// safely stop the Minim engine upon shutdown.
public void stop(){
  mySample1.close();
  mySample2.close();
  mySample3.close();
  mySample4.close();
  minim.stop();
  super.stop();

}
```

Solution to Exercise 2.5 (p. 27)

```
import beads.*;

AudioContext ac;

String sourceFile; //path to audio file
SamplePlayer mySample1, mySample2, mySample3, mySample4;
Gain g;
Glide cutoff1, cutoff2, cutoff3, cutoff4;
OnePoleFilter lpf1, lpf2, lpf3, lpf4;

void setup() {
  size(200, 200);
  colorMode(HSB, 360, height, height);

  ac = new AudioContext();

  sourceFile = sketchPath("") + "data/flauto.aif";
  try {
    mySample1 = new SamplePlayer(ac, new Sample(sourceFile));
  }
  catch (Exception e) {
    println("Exception while attempting to load sample.");
    e.printStackTrace(); // description of error
    exit();
  }
  mySample1.setKillOnEnd(false);

  sourceFile = sketchPath("") + "data/oboe.wav";
```

```
try {
  mySample2 = new SamplePlayer(ac, new Sample(sourceFile));
}
catch (Exception e) {
  println("Exception while attempting to load sample.");
  e.printStackTrace(); // description of error
  exit();
}
mySample2.setKillOnEnd(false);  sourceFile = sketchPath("") + "data/flauto.aif";

sourceFile = sketchPath("") + "data/tromba.wav";
try {
  mySample3 = new SamplePlayer(ac, new Sample(sourceFile));
}
catch (Exception e) {
  println("Exception while attempting to load sample.");
  e.printStackTrace(); // description of error
  exit();
}
mySample3.setKillOnEnd(false);  sourceFile = sketchPath("") + "data/flauto.aif";

sourceFile = sketchPath("") + "data/violino.wav";
try {
  mySample4 = new SamplePlayer(ac, new Sample(sourceFile));
}
catch (Exception e) {
  println("Exception while attempting to load sample.");
  e.printStackTrace(); // description of error
  exit();
}
mySample4.setKillOnEnd(false);

cutoff1 = new Glide(ac, 1000, 20);
lpf1 = new OnePoleFilter(ac, cutoff1);
lpf1.addInput(mySample1);
cutoff2 = new Glide(ac, 1000, 20);
lpf2 = new OnePoleFilter(ac, cutoff2);
lpf2.addInput(mySample2);
cutoff3 = new Glide(ac, 1000, 20);
lpf3 = new OnePoleFilter(ac, cutoff3);
lpf3.addInput(mySample3);
cutoff4 = new Glide(ac, 1000, 20);
lpf4 = new OnePoleFilter(ac, cutoff4);
lpf4.addInput(mySample4);

g = new Gain(ac, 1, 1);
g.addInput(lpf1);
g.addInput(lpf2);
g.addInput(lpf3);
g.addInput(lpf4);
ac.out.addInput(g);
```

```
  ac.start();
  background(0);
}


void draw()
{
  stroke(255);
  strokeWeight(1);
  fill(0, 88, 88);
  ellipseMode(CORNER);
  ellipse(50,50,100,100);

  beginShape(LINES);
  vertex(50, 100);
  vertex(90, 100);

  vertex(110, 100);
  vertex(150, 100);

  vertex(100, 50);
  vertex(100, 90);

  vertex(100, 110);
  vertex(100, 150);
  endShape();
}

void mouseReleased(){
  // FLAUTO
  if ((mouseX > 95) && (mouseX < 105)&& (mouseY > 50)&& (mouseY < 90)) {
    cutoff1.setValue(map(mouseY, 50, 90, 1000, 30));
    mySample1.setToLoopStart();
    mySample1.start();
  }

  // OBOE
  if ((mouseX > 110) && (mouseX < 149)&& (mouseY > 95)&& (mouseY < 105)) {
    cutoff2.setValue(map(mouseX, 110, 149, 30, 1000));
    mySample2.setToLoopStart();
    mySample2.start();
  }

  // TROMBA
  if ((mouseX > 95) && (mouseX < 105)&& (mouseY > 110)&& (mouseY < 149)) {
    cutoff3.setValue(map(mouseY, 110, 149, 30, 1000));
    mySample3.setToLoopStart();
    mySample3.start();
  }

  // VIOLINO
  if ((mouseX > 50) && (mouseX < 90)&& (mouseY > 95)&& (mouseY < 105)) {
    cutoff4.setValue(map(mouseX, 50, 90, 1000, 30));
    mySample4.setToLoopStart();
    mySample4.start();
  }
}
```

Solution to Exercise 2.6 (p. 27)
This is an example of solution. Please make sure that the fonts used are present in your computer or in the `Data` folder.

```
size(200,200, JAVA2D);
PFont fonte;
fonte = loadFont("HoeflerText-Black-48.vlw"); // previously created and inserted in Data
textFont(fonte, 12);
fill(10, 20, 250, 80);
textAlign(RIGHT);
text("pippo pippo non lo sa", 10, 14,  35, 70);
textFont(fonte, 94);
textAlign(LEFT);
fill(200, 0, 0, 100);
text("ppnls", 25, 5, 150, 190);
fonte = createFont("Serif", 10, false); // Java logical font
textFont(fonte, 80);
fill(0, 200, 0, 170);
rotate(PI/6);
text("LO SO", 20, 20, 280, 280);
fonte = createFont("cmsy10", 10, true); // font installed in the system
textFont(fonte, 80);
fill(0, 20, 150, 170);
rotate(PI/12);
text("ECCO", 20, 20, 280, 280);
fonte = createFont("grunge.ttf", 10, true); // vectorial font in the Data folder
textFont(fonte, 80);
fill(100, 100, 0, 170);
rotate(-PI/6);
text("qui", 20, 20, 280, 280);
```

Chapter 3

Graphic Composition in Processing[1]

3.1 Graphic primitives

In Processing, we can arrange points, lines, surfaces, and volumes (objects in 0, 1, 2, and 3 dimensions, respectively) in a 3D space or, where this makes sense in the 2D space of the image window. The number of parameters of the object primitives will determine if these objects have to be positioned in the X-Y or in the X-Y-Z space.

3.1.1 0D

Points
The `point()` is the simplest of the graphic primitives. When invoked with two coordinate parameters, it positions a point in the X-Y space. When invoked with three coordinate parameters, it positions a point in the X-Y-Z space. A point, in geometric sense, does not have dimension, but it can be assigned an occupation in pixels and a color, by the functions `strokeWeight()` and `stroke()`, respectively. For example, the simple Processing sketch

```
stroke(180,0,0);
strokeWeight(10);
point(60,60);
```

draws a dot in the image window.
Collections of points
A set of points can be grouped into a single object (cloud of points) by the delimiters `beginShape(POINTS)` and `endShape()`. Between them each point has to be specified with the `vertex()` function. Transformations of rotation, translation, and scaling do not apply to the inside of composite objects described with `beginShape()` and `endShape()`, but they can precede the definition of a composite object and apply to the whole.

3.1.2 1D

Straight Lines
The `line()` draws a line segment between two points in the plane or the 3D space, with width and color that can be set with `strokeWeight()` and `stroke()`, respectively.

[1]This content is available online at <http://cnx.org/content/m12986/1.10/>.

Collections of segments
A set of segments can be defined, as we saw for points, by the delimiters `beginShape()` and `endShape()`. Between them, vertices are listed by calls to the function `vertex()`. Using the invocation `beginShape(LINES)` the vertices are taken in couples, each identifying a segment. With the argument-free invocation `beginShape()` the vertices, taken one after the other, define a polygonal line. With the closure `endShape(CLOSE)` the line is closed on itself by linking the first and last vertices. The color of such polygon can be set by using the `fill()` function or, conversely, left equal to the background color with the `noFill()`.
Curves
The function `curve()`, when called with eight parameters, draws a curve on the image plane, with initial and final points determined, respectively, by the second and third couple of coordinates passed as arguments. The first and last couple of coordinates define two control points for the curve, which is an interpolating spline, thus passing for all four points. In Processing, however, only the curve segment between the intermediate points is visualized.

Definition 3.1: Spline
Piecewise-polynomial curve, with polynomials connected with continuity at the **knots**

NOTE: See Introduction to Splines[2] and, for an introduction to the specific kind of splines (Catmull-Rom) used in Processing, the term **spline** in Wikipedia.

In order to have an arbitrary number of control points we must use the function `curveVertex()` to specify each point in the block delimited by `beginShape()` and `endShape()`.

As opposed to the `curve()`, in the `bezier()` function call the two control points specified by the four middle parameters are not points touched by the curve. They only serve to define the shape of the **approximating Bézier curve**, which has the following interesting properties:

- it is entirely contained in the **convex hull** defined by the extremal points and the control points;
- transformations of translation, rotation, or scaling, appied to the extremal and control points determine a similar transformation of the curve.

As we can see by running the code

```
    stroke(255, 0, 0);
line(93, 40, 10, 10);
line(90, 90, 15, 80);
stroke(0, 0, 0);
noFill();
bezier(93, 40, 10, 10, 90, 90, 15, 80);
```

the control points lay on the tangent passing by the extremal points. In order to have an arbitrary number of control points one must use the `bezierVertex()` to specify each point within a block delimited by `beginShape()` and `endShape()`. In this way, an arbitrarily involute curve can be traced in the 3D space. In 2D, the function `bezierVertex()` has six parameters that correspond to the coordinates of two control points and one anchor point. The first invocation of `bezierVertex()` has to be preceded by a call to `vertex()` which fixes the first anchor point of the curve.

There are other methods that allow to read the coordinates or the slope of the tangent to an arbitrary point of a Bézier or spline curve. Such point can be specified by a parameter t that can go from 0 (first extreme) to 1 (second extreme). It is also possible to set the precision of approximating or interpolating curves in 3D. For details see the Processing reference manual [3] .

The Processing sketch in table (Table 3.1) shows the difference between the spline interpolating curve and the Bézier curve.

NOTE: See the term **Bézier curve** in Wikipedia.

[2]"Introduction to Splines" <http://cnx.org/content/m11153/latest/>
[3]http://www.processing.org/reference/index_ext.html

applet that compares the Bézier curve (red) and the interpolating spline (black) [4]	
	```
	void setup() {
	  c1x = 120;
	  c1y = 110;
	  c2x = 50;
	  c2y = 70;
	  background(200);
	  stroke(0,0,0);
	  size(200, 200);
	}

	int D1, D2;
	int X, Y;
	int c1x, c1y, c2x, c2y;

	void draw() {
	  if (mousePressed == true) {
	    X = mouseX; Y = mouseY;
	    // selection of the point that is modified
	    D1 = (X - c1x)*(X - c1x) + (Y - c1y)*(Y - c1y);
	    D2 = (X - c2x)*(X - c2x) + (Y - c2y)*(Y - c2y);
	    if (D1 < D2) {
	    c1x = X; c1y = Y;
	    }
	    else {
	     c2x = X; c2y = Y;
	     }
	    }
	    background(200);
	    stroke(0,0,0);
	    strokeWeight(1);

	    noFill();
	    beginShape();
	    curveVertex(10,  10);
	    curveVertex(10,  10);
	    curveVertex(c2x, c2y);
	    curveVertex(c1x, c1y);
	    curveVertex(190, 190);
	    curveVertex(190, 190);
	    endShape();

	    stroke(255,30,0);
	    bezier(10,10,c2x,c2y,c1x,c1y,190,190);
	    strokeWeight(4);
	    point(c1x,c1y);
	    point(c2x,c2y);
	}
	```

Table 3.1

3.1.3 2D

NOTE: Objects in two or three dimensions take a color that can be determined by the illumination, as explained in Section 3.3 (Lighting), or established by the method `fill()`, which also gives the possibility to set the degree of transparency.

Triangles
The triangle is the fundamental construction element for 3D graphics. In fact, by juxtaposition of triangles one can approximate any continuous surface. In Processing, however, the triangles are specified in 2D by the primitive `triangle()`, whose six parameters correspond to the coordinates of the vertices in the image window. Even though each triangle is defined in 2D, it can be rotated and translated in the 3D space, as it happens in the Processing sketch

```
    void setup(){ size(200, 200, P3D); fill(210, 20,
20); }

float angle = 0;

void draw(){
    background(200); // clear image
    stroke(0,0,0);
    angle += 0.005;
    rotateX(angle);
    triangle(10,10,30,70,80,80);
}
```

Collections of triangles
A set of triangles can be defined, similarly to what we did for points and segments, by the delimiters `beginShape()` and `endShape()`. Between them, the vertices of the triangles are listed by calls to the function `vertex()`. By the invocation `beginShape(TRIANGLES)` the vertices are taken in triples, each defining a triangle, while the invocation `beginShape(TRIANGLE_STRIP)` takes the vertices one after the other to define a strip mad of triangular facets. If the `vertex()` has three arguments, the vertices are located in the 3D space and the corresponding triangles identify planar surfaces in space.

Quadrilaterals
Rectangles are defined, in Processing, by the function `rect()` of four parameters, where the first couple specifies, by default, the position in the the 2D plane of the top-left corner, and the third and fourth parameters specify the width and height, respectively. The meaning of the first couple of parameters can be changed with the function `rectMode()`: `rectMode(CORNER)` gives the default positioning; `rectMode(CENTER)` gives the positioning of the center of the rectangle at the specified point; with the `rectMode(CORNERS)` the four parameters are interpreted as the coordinates of the top-left and bottom-right vertices, respectively. A generic quadrilateral is defined by the coordinates of its four vertices, passed as parameters to the function `quad()`. It is important to notice that in 3D, while a triangle stays planar in any case, a quadruple of points does not necessarily lay on a plane. Viceversa, the quadrilaterals that are defined by 3D roto-translations of quadruples of 2D vertices, remain planar. Processing allows only eight parameters to be passed to `quad()`, thus forcing the definition of a quadrilateral as a quadruple of vertices in 2D.

Collections of quadrilaterals
A set of quadrilaterals can be defined, similarly to what we saw for triangles, by the delimiters `beginShape()` and `endShape()`. Between them, vertices are listed by calls to the function `vertex()`. By using the invocation `beginShape(QUADS)` the vertices are taken in quadruples, each identifying a quadrilateral, while the

[4] See the file at <http://cnx.org/content/m12986/latest/bezier_curve.html>

invocation `beginShape(QUAD_STRIP)` takes the vertices one after the other to define a strip mad of quadrilateral facets. If the `vertex()` have three parameters, the planarity of the resulting faces is not ensured, and the resulting rendering can be misleading. For instance, by running the code

```
size(200,200,P3D);
lights();
beginShape(QUADS);
vertex(20,31, 33);
vertex(80, 40, 38);
vertex(75, 88, 50);
vertex(49, 85, 74);
endShape();
```

we realize that the quadrilateral is rendered as the juxtaposition of two triangles belonging to different planes.
Polygons
A generic polygon is defined as a set of vertices, and it has a surface that can be colored. In Processing the vertices are listed within a couple `beginShape(POLYGON); - endShape();` Actually, the polygon has to be intended in a generalized sense, as it is possible to use the `bezierVertex()` and `curveVertex()` to specify curved profiles. For instance, the reader may try to draw the moon:

```
fill(246, 168, 20);
beginShape(POLYGON);
vertex(30, 20);
bezierVertex(80, 10, 80, 75, 30, 75);
bezierVertex(50, 70, 60, 25, 30, 20);
endShape();
```

Ellipses
The function `ellipse()` draws an ellipse in the 2D plane. Its four parameters are interpreted, as in the case of `rect()`, as position followed by width and height. The position can be set in different ways according to the `ellipseMode()`, whose parameter can take values `CORNER, CORNERS, CENTER, CENTER_RADIUS`. The first couple of these possible values have to be referred to the rectangle that is circumscribed to the ellipse.

3.1.4 3D

Processing offers a very limited repertoire of 3D-object primitives, essentially only balls and boxes.
Boxes
The function `box()` produces a cube when invoked with a single parameter (edge), a parallelepiped when invoked with three parameters (width, height, depth).
Balls
The function `sphere()` produces, by an approximating polyhedron, a sphere whose radius is specified as a parameter.The function `sphereDetail()` can be used to specify the number of vertices of the polyhedron that approximates the ideal sphere.

3.2 The stack of transformations

A rotation or a translation can be imagined as operations that rotate or translate the Cartesian reference system. In other terms, after a `rotate()` or a `translate()` the following positioning operations of the objects will have a new coordinate system. When various objects are positioned in different ways in space, it is useful to keep trace of the coordinate systems that are set, one after the other. The data structure that is suited for containing such systems is the stack of transformations (**matrix stack**). With the function `pushMatrix()` the current coordinate system is put on top of the stack. On the other hand, to revert to the coordinate system before the last transformation, we have to call a `popMatrix()`. Actually, the stack contains the affine transformation matrices, according to what is dictated by **OpenGL** and described in Section 3.5 (Pills of OpenGL).

Example 3.1
In this example two objects are positioned in the 3D space: a planar square and a cube. The first `pushMatrix()` saves the coordinate system onto the stack, then some transformations are applied, and finally the square is drawn. To go back to the previous coordinate system and apply new transformations to position the cube, we apply a `popMatrix()`. Essentially, the `pushMatrix()` and `popMatrix()` determine the scope for the geometric positioning of an object.

```
float angle;

void setup(){
          size(100, 100, P3D);
  int angle = 0;
}

void draw(){
  background(200);
  angle += 0.003;
  pushMatrix();
  translate(25,50);
  rotateZ(angle);
  rotateY(angle);
  rectMode(CENTER);
  rect(0,0,20,20);
  popMatrix();
  translate(75,50,-25);
  rotateX(angle);
  box(20);
}
```

3.3 Lighting

The Processing lighting model echoes the model used in **OpenGL**, that is the **Phong reflection model**. Such model is not physically justified, but it is particularly efficient. OpenGL considers as illuminated each polygon whose normal forms an acute angle with the direction of incoming light. This happens regardless of any masking objects. Therefore, shadows are not cast. OpenGL is said to use a local illumination model, since multiple reflections among surfaces and cast shadows are not automatically rendered.

An environmental light is available, which is not coming from any particular direction, and whose color is specified by the parameters of the activation call `ambientLight()`. A directional light source is set with the `directionalLight()`, whose parameters specify color and incoming direction. The method `lights()` activates a default combination of gray ambient light and directional light, the latter also gray, coming from the frontal direction. It is possible to set a point light source in a given point of space by the call `pointLight()`. Finally, the method `spotLight()` activates a light beam which can be controlled in its color, position, direction, aperture, and concentration around the axis. The exponent e tunes the falloff around the axis:

$$\cos^e(\phi) \tag{3.1}$$

When hitting a planar surface, a directional light produces reflected light along several directions, depending on the surface properties. In the case of perfectly-diffusive (or **Lambertian**) surface, the light radiates evenly from the surface along all directions, with an intensity that is larger for incident directions closer to the surface normal. Vice versa, if the surface is perfectly reflecting, light is only reflected along the direction that is specularly symmetric (about the surface normal) to the incident direction. In OpenGL, to have some flexibility in defining the illumination, each source has the three illumination components: ambient, diffuse, and specular. These three components are separately defined and interact with the respective components that define the surface properties of objects. The colors defined in the methods `directionalLight()`, `pointLight()`, and `spotLight()` define the Lambertian component of illumination. The `lightSpecular()` specifies the color of the component of incoming light that is subject to specular reflection.

In Processing, the properties of surfaces are controlled by the methods `ambient()` (acting on the ambient component of incoming lights) and `specular()` (acting on the specular component). The function `shininess()` controls the concentration of the specularly-reflected beam, by a coefficient that acts similarly to the exponent of (3.1). The represented objects can also be considered as sources of light, and they can be assigned an emission light by the `emmissive()` call. However, the sources defined in this way do not illuminate the other objects on the scene.

In OpenGL the point, spot, and ambient lights are attenuated with increasing distance, according to the model

$$\text{attenuation} = \frac{1}{a + bd + cd^2} \tag{3.2}$$

The method `ligthFalloff()` allows to specify the parameters a, b, and c.

Example 3.2
Here, a cube and a `QUAD_STRIP` are positioned in space and illuminated by a rotating source. Moreover, a soft fixed light is set. Notice the absence of shadows and the apparent planarity of surfaces in the `QUAD_STRIP`.

```
float r;
float lightX, lightY, lightZ;

void setup() {
  size(400, 400, P3D);
  r = 0;
  ambient(180, 90, 0);
  specular(0, 0, 240);
  lightSpecular(200, 200, 200);
  shininess(5);
}

void draw() {
  lightX = 100*sin(r/3) + width/2;
  lightY = 100*cos(r/3) + height/2;
  lightZ = 100*cos(r);
  background(0,0,0);
  noStroke();
  ambientLight(153, 102, 0);

  lightSpecular(0, 100, 200);
  pointLight(100, 180, 180,
             lightX, lightY, lightZ);
  pushMatrix();
    translate(lightX, lightY, lightZ);
    emissive(100, 180, 180);
    sphere(4); //Put a little sphere where the light is
    emissive(0,0,0);
  popMatrix();
  pushMatrix();
    translate(width/2, height/2, 0);
    rotateX(PI/4);
    rotateY(PI/4);
    box(100);
  popMatrix();
  pushMatrix();
    translate(width/4, height/2, 0);
    beginShape(QUAD_STRIP);
    vertex(10,13,8);
    vertex(13,90,13);
    vertex(65,76,44);
    vertex(95,106,44);
    vertex(97,20,70);
    vertex(109,70,80);
   endShape();
  popMatrix();
  r+=0.05;
}
```

3.4 Projections

3.4.1 Perspective projections

A perspective projection is defined by a **center of projection** and a **plane of projection**. The **projector rays** connect the points in the scene with the center of projection, thus highlighting the corresponding points in the plane of projection. The Figure 3.1 shows a section where the plane of projection produces a straight line whose abscissa is $-d$, and the center of projection is in the origin.

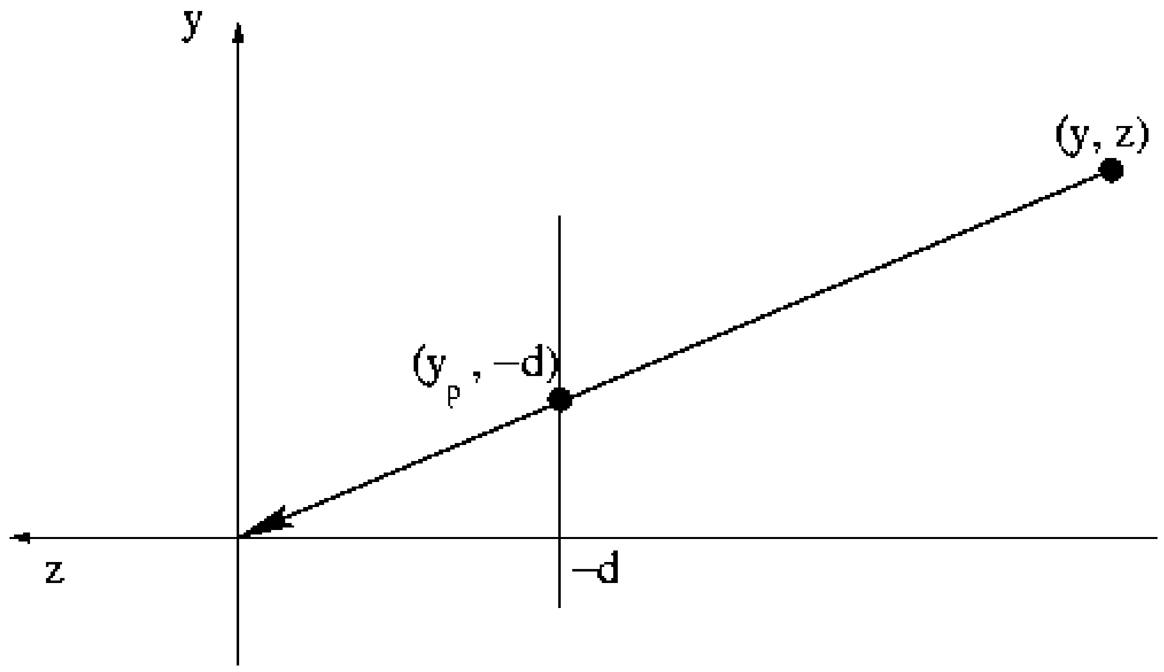


Figure 3.1

By similarity of two triangles it is easy to realize that the point having ordinate y gets projected onto the plane in the point having ordinate $y_p = -\frac{yd}{z}$.

In general, the projection of a point having homogeneous coordinates $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$ onto a plane orthogonal to the z axis and intersecting such axis in position $-d$ is obtained, in homogeneous coordinates, by multiplication with the matrix $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{pmatrix}$. The projected point becomes $\begin{pmatrix} x \\ y \\ z \\ -\frac{z}{d} \end{pmatrix}$, which can be normalized by multiplication of all its element by $-\frac{d}{z}$. As a result, we obtain $\begin{pmatrix} -\frac{xd}{z} \\ -\frac{yd}{z} \\ -d \\ 1 \end{pmatrix}$

3.4.2 Parallel views

Parallel views are obtained by taking the center of projection back to infinity (∞). In this way, the projector rays are all parallel.

3.4.2.1 Orthographic projection

The orthographic projection produces a class of parallel views by casting projection rays orthogonal to the plane of projection. If such plane is positioned orthogonally to the z axis and passing by the origin, the projection matrix turns out to be particolarly simple: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$. Among orthographic projections, the **axonometric projections** are based on the possibility to measure the object along three orthogonal axes, and on the orientation of the plane of projection with respect to these axes. In particular, in the isometric projection[5] the projections of the axes form angles of $120\,^\circ$. The isometric projection has the property that equal segments on the three axes remain equal when they are projected onto the plane. In order to obtain the isometric projection of an object whose main axes are parallel to the coordinate axes, we can first rotate the object by $45\,^\circ$ about the y axis, and then rotate by $\arctan\left(\frac{1}{\sqrt{2}}\right) = 35.264\,^\circ$ about the x axis.

3.4.2.2 Oblique projection

We can talk about oblique projection every time the projector rays are oblique (non-orthogonal) to the projection plane. In order to deviate the projector rays from the normal direction by the angles θ and ϕ we must use a projection matrix $\begin{pmatrix} 1 & 0 & -\tan(\theta) & 0 \\ 0 & 1 & -\tan(\phi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

3.4.3 Casting shadows

As we have seen, Processing has a local illumination model, thus being impossible to cast shadows directly. However, by manipulating the affine transformation matrices we can cast shadows onto planes. The method is called **flashing in the eye**, thus meaning that the optical center of the scene is moved to the point where the light source is positioned, and then a perspective transformation is made, with a plane of projection that coincides with the plane where we want to cast the shadow on.

Example 3.3
The following program projects on the floor the shadow produced by a light source positioned on the y axis. The result is shown in Figure 3.2 (Casting a shadow)

[5] http://en.wikipedia.org/wiki/Isometric_projection

Casting a shadow

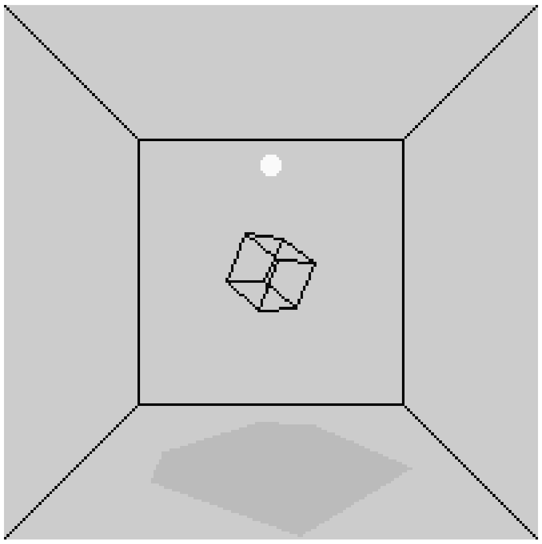


Figure 3.2

```
    size(200, 200, P3D);
float centro = 100;
float yp = 70; //floor (plane of projection) distance from center
float yl = 40; //height of light (center of projection) from center

translate(centro, centro, 0); //center the world on the cube

noFill();
box(yp*2); //draw of the room

pushMatrix();
  fill(250); noStroke();
  translate(0, -yl, 0); // move the virtual light bulb higher
  sphere(4); //draw of the light bulb
  stroke(10);
popMatrix();

pushMatrix(); //draw of the wireframe cube
  noFill();
  rotateY(PI/4); rotateX(PI/3);
  box(20);
popMatrix();

// SHADOW PROJECTION BY COMPOSITION
// OF THREE TRANSFORMATIONS (the first one in
// the code is the last one to be applied)
```

```
translate(0, -yl, 0); // shift of the light source and the floor back
              // to their place (see the translation below)

applyMatrix(1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0,
            0, 1/(yp+yl), 0, 0); // projection on the floor
                                 // moved down by yl

translate(0, yl, 0); // shift of the light source to center
                     // and of the floor down by yl

pushMatrix();    // draw of the cube that generate the shadow
  fill(120, 50); // by means of the above transformations
  noStroke();
  rotateY(PI/4); rotateX(PI/3);
  box(20);
popMatrix();
```

3.5 Pills of OpenGL

OpenGL is a set of functions that allow the programmer to access the graphic system. Technically speaking, it is an **Application Programming Interface (API)**. Its main scope is the graphic rendering of a scene populated by 3D objects and lights, from a given viewpoint. As far as the programmer is concerned, OpenGL allows to describe geometric objects and some of their properties, and to decide how such objects have to be illuminated and seen. As far as the implementation is concerned, OpenGL is based on the **graphic pipeline** , made of modules as reported in Figure 3.3 (The OpenGL pipeline). An excellent book on interactive graphics in OpenGL was written by *Angel* [1].

The OpenGL pipeline

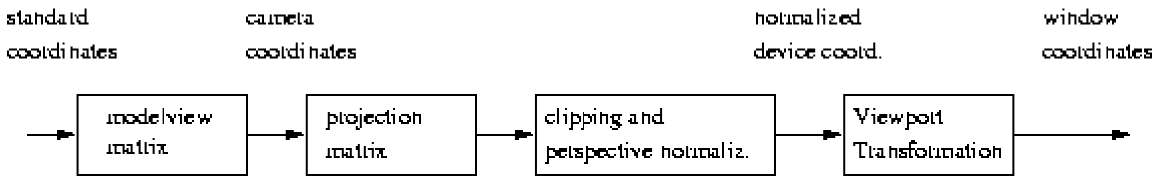


Figure 3.3

In Processing (and in OpenGL), the programmer specifies the objects by means of world coordinates (**standard coordinates**). The **model-view matrix** is the transformation matrix used to go from standard coordinates to a space associated with the camera. This allows to change the camera viewpoint and orientation dynamically. In OpenGL this is done with the function `gluLookAt()`, which is reproduced in Processing by the `camera()`. The first triple of parameters identifies the position, in world coordinates, of the optical center of the camera (**eye point**). The second triple of parameters identifies a point where the camera is looking at (**center of the scene**). The third triple of coordinates identifies a vector aimed at specifying the viewing vertical. For example, the program

```
void setup() {
           size(100, 100, P3D);
           noFill();
           frameRate(20);
        }

        void draw() {
           background(204);
           camera(70.0, 35.0, 120.0, 50.0, 50.0, 0.0,
             (float)mouseX /width, (float)mouseY /height, 0.0);
           translate(50, 50, 0);
           rotateX(-PI/6);
           rotateY(PI/3);
           box(45);
        }
```

draws the **wireframe** of a cube and enables the dynamic rotation of the camera.

The **projection matrix** is responsible for the projection on the viewing window, and this projection can be either parallel (orthographic) or perspective. The orthographic projection can be activated with the call `ortho()`. The perspective projection is the default one, but it can be explicitly activated with the call `perspective()`. Particular projections, such as the oblique ones, can be obtained by distortion of objects by application of the `applyMatrix()`. There is also the **texture matrix**, but textures are treated in another module.

For each type of matrix, OpenGL keeps a stack, the current matrix being on top. The stack data structure allows to save the state (by the `pushMatrix()`) before performing new transformations, or to remove the current state and activate previous states (by the `popMatrix()`). This is reflected in the Processing operations described in Section 3.2 (The stack of transformations). In OpenGL, the transformations are applied according to the sequence

1. Push on the stack;
2. Apply all desired transformations by multiplying by the stack-top matrix;
3. Draw the object (affected by transformations);
4. Pop from the stack.

A **viewport** is a rectangular area of the display window. To go from the perspective projection plane to the viewport two steps are taken: (i) transformation into a 2 x 2 window centered in the origin (**normalized device coordinates**) (ii) mapping the normalized window onto the viewport. Using the normalized device coordinates, the **clipping** operation, that is the elimination of objects or parts of objects that are not visible through the window, becomes trivial. `screenX()`, `screenY()`, and `screenZ()` gives the X-Y coordinates produced by the viewport transformation and by the previous operators in the chain of Figure 3.3 (The OpenGL pipeline).

The viewing **frustum** is the solid angle that encompasses the perspective projection, as shown in Figure 3.4 (The viewing frustum). The objects (or their parts) belonging to the viewing volume are visualized, the remaining parts are subject to clipping. In Processing (and in OpenGL) the frustum can be defined by positioning the six planes that define it (`frustum()`), or by specification of the vertical angle, the, **aspect ratio**, and the positions of the front and back planes (`perspective()`). One may ask how the system removes the hidden faces, i.e., those faces that are masked by other faces in the viewing volume. OpenGL uses the **z-buffer** algorithm, which is supported by the graphic accelerators. The board memory stores a 2D memory area (the z-buffer) corresponding to the pixels of the viewing window, and containing depth values. Before a polygon gets projected on the viewing window the board checks if the pixels affected by such polygon have a depth value smaller than the polygon being drawn. If this is the case, it means that there is an object that masks the polygon.

The viewing frustum

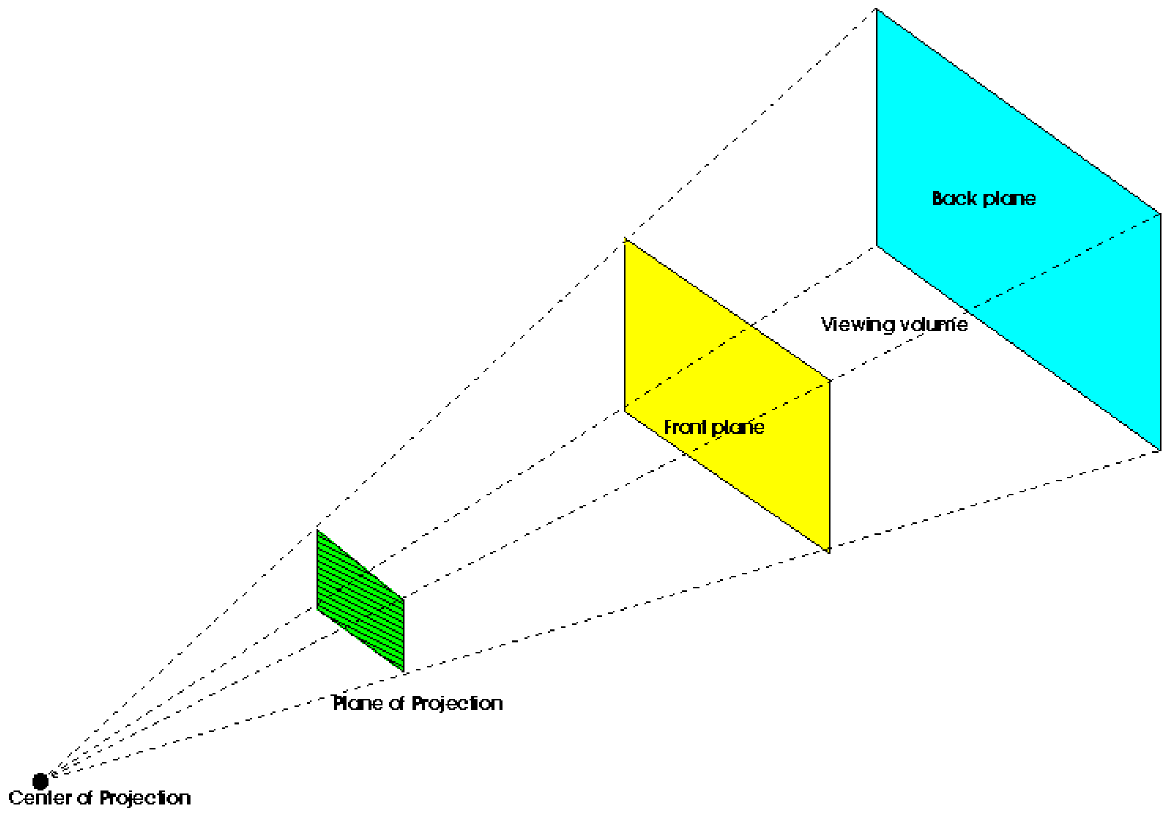


Figure 3.4

Sophisticated geometric transformations are possible by direct manipulation of the projection and model-view matrices. This is possible, in Processing, starting from the unit matrix, loaded with `resetMatrix()`, and proceeding by matrix multiplies done with the `applyMatrix()`.

Exercise 3.1 ***(Solution on p. 50.)***

Run and analyze the Processing code

```
size(200, 200, P3D);
println("Default matrix:"); printMatrix();
noFill();
ortho(-width/2, width/2, -height/2, height/2, -100, 100);
translate(100, 100, 0);
println("After translation:"); printMatrix();
rotateX(atan(1/sqrt(2)));
```

```
  println("After about-X rotation:"); printMatrix();
  rotateY(PI/4);
  println("After about-Y rotation:"); printMatrix();
  box(100);
```

What is visualized and what it the kind of projection used? How do you interpret the matrices printed out on the console? Can one invert the order of rotations?

Exercise 3.2 ***(Solution on p. 50.)***

Write a Processing program that performs the oblique projection of a cube.

Exercise 3.3 ***(Solution on p. 50.)***

Visualize a cube that projects its shadow on the floor, assuming that the light source is at infinite distance (as it is the case, in practice, for the sun).

Solutions to Exercises in Chapter 3

Solution to Exercise 3.1 (p. 48)
The wireframe of a cube is visualized in isometric projection. The latter three matrices represent, one after the other, the three operations of translation (to center the cube to the window), rotation about the x axis, and rotation about the y axis. A sequence of two rotations correspond to the product of two rotation matrices, and the outcome is not order independent (product is not commutative). The product of two rotation matrices $R_x R_y$ correspond to performing the rotation about y first, and then the rotation about x.
Solution to Exercise 3.2 (p. 49)
For example:

```
size(200, 200, P3D);
float theta = PI/6;
float phi = PI/12;
noFill();
ortho(-width/2, width/2, -height/2, height/2, -100, 100);
translate(100, 100, 0);
applyMatrix(1, 0, - tan(theta), 0,
            0, 1, - tan(phi), 0,
                    0, 0, 0, 0,
                    0, 0, 0, 1);
box(100);
```

Solution to Exercise 3.3 (p. 49)
We do it similarly to Example 3.3, but the transformation is orthographic:

```
size(200, 200, P3D);
  noFill();
  translate(100, 100, 0);
  pushMatrix();
    rotateY(PI/4); rotateX(PI/3);
    box(30);
  popMatrix();
  translate(0, 60, 0); //cast a shadow from infinity (sun)
  applyMatrix(1, 0, 0, 0,
              0, 0, 0, 0,
                      0, 0, 1, 0,
                      0, 0, 0, 1);
  fill(150);
  pushMatrix();
  noStroke();
  rotateY(PI/4); rotateX(PI/3);
  box(30);
  popMatrix();
```

Chapter 4

Signal Processing in Processing: Sampling and Quantization[1]

4.1 Sampling

Both sounds and images can be considered as signals, in one or two dimensions, respectively. Sound can be described as a fluctuation of the acoustic pressure in time, while images are spatial distributions of values of luminance or color, the latter being described in its RGB or HSB components. Any signal, in order to be processed by numerical computing devices, have to be reduced to a sequence of discrete **samples**, and each sample must be represented using a finite number of bits. The first operation is called **sampling**, and the second operation is called **quantization** of the domain of real numbers.

4.1.1 1-D: Sounds

Sampling is, for one-dimensional signals, the operation that transforms a continuous-time signal (such as, for instance, the air pressure fluctuation at the entrance of the ear canal) into a discrete-time signal, that is a sequence of numbers. The discrete-time signal gives the values of the continuous-time signal read at intervals of T seconds. The reciprocal of the sampling interval is called **sampling rate** $F_s = \frac{1}{T}$. In this module we do not explain the theory of sampling, but we rather describe its manifestations. For a a more extensive yet accessible treatment, we point to the *Introduction to Sound Processing* [2]. For our purposes, the process of sampling a 1-D signal can be reduced to three facts and a theorem.

- The Fourier Transform[2] of a discrete-time signal is a function (called **spectrum**) of the continuous variable ω, and it is periodic with period 2π. Given a value of ω, the Fourier transform gives back a complex number that can be interpreted as magnitude and phase (translation in time) of the sinusoidal component at that frequency.
- Sampling the continuous-time signal $x(t)$ with interval T we get the discrete-time signal $x(n) = x(nT)$, which is a function of the discrete variable n.
- Sampling a continuous-time signal with sampling rate F_s produces a discrete-time signal whose frequency spectrum is the periodic replication of the original signal, and the replication period is F_s. The Fourier variable ω for functions of discrete variable is converted into the frequency variable f (in Hertz) by means of $f = \frac{\omega}{2\pi T}$.

The Figure 4.1 (Frequency spectrum of a sampled signal) shows an example of frequency spectrum of a signal sampled with sampling rate F_s. In the example, the continuous-time signal had all and only the frequency components between $-F_b$ and F_b. The replicas of the original spectrum are sometimes called **images**.

[1]This content is available online at <http://cnx.org/content/m13045/1.6/>.
[2]"Derivation of the Fourier Transform" <http://cnx.org/content/m0046/latest/>

Frequency spectrum of a sampled signal

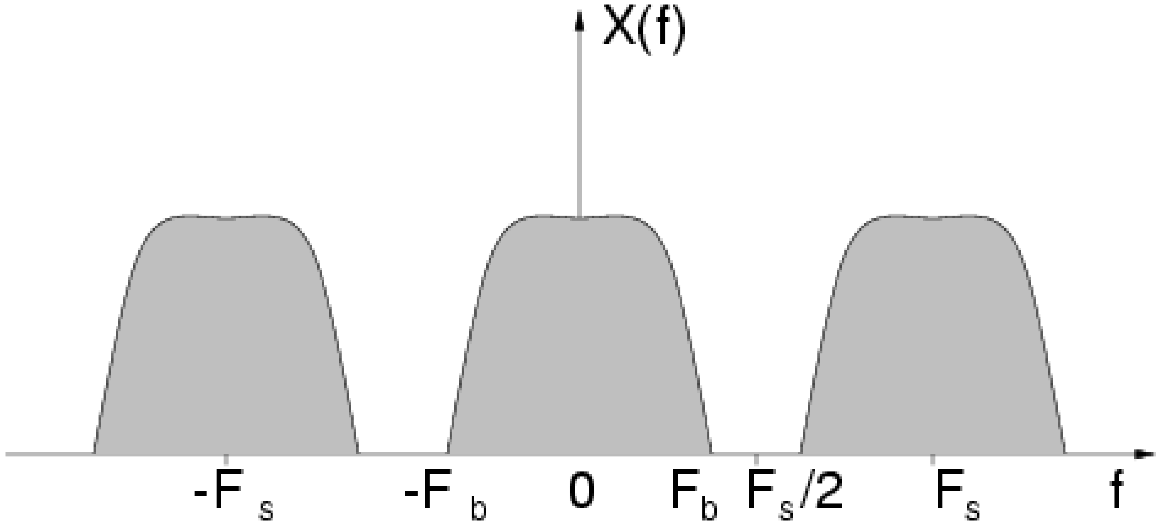


Figure 4.1

Given the facts (p. 51), we can have an intuitive understanding of the Sampling Theorem, historically attributed to the scientists Nyquist and Shannon.

Theorem 4.1: Sampling Theorem

A continuous-time signal $x(t)$, whose spectral content is limited to frequencies smaller than F_b (i.e., it is band-limited to F_b) can be recovered from its sampled version $x(n)$ if the sampling rate is larger than twice the bandwidth (i.e., if $F_s > 2F_b$)

The reconstruction can only occur by means of a filter that cancels out all spectral images except for the one directly coming from the original continuous-time signal. In other words, the canceled images are those having frequency components higher than the **Nyquist frequency** defined as $\frac{F_s}{2}$. The condition required by the sampling theorem (Theorem 4.1, Sampling Theorem, p. 52) is equivalent to saying that no overlaps between spectral images are allowed. If such superimpositions were present, it wouldn't be possible to design a filter that eliminates the copies of the original spectrum. In case of overlapping, a filter that eliminates all frequency components higher than the Nyquist frequency would produce a signal that is affected by **aliasing**. The concept of aliasing is well illustrated in the Aliasing Applet[3], where a continuous-time sinusoid is subject to sampling. If the frequency of the sinusoid is too high as compared to the sampling rate, we see that the the waveform that is reconstructed from samples is not the original sinusoid, as it has a much lower frequency. We all have familiarity with aliasing as it shows up in moving images, for instance when the wagon wheels in western movies start spinning backward. In that case, the sampling rate is given by the **frame rate**, or number of pictures per second, and has to be related with the spinning velocity of the wheels. This is one of several stroboscopic [4] phenomena.

In the case of sound, in order to become aware of the consequences of the 2π periodicity of discrete-time signal spectra (see Figure 4.1 (Frequency spectrum of a sampled signal)) and of violations of the condition of the sampling theorem, we examine a simple case. Let us consider a sound that is generated by a sum

[3]"Aliasing Applet" <http://cnx.org/content/m11448/latest/>
[4]http://www.michaelbach.de/ot/mot_strob/

of sinusoids that are harmonics (i.e., integer multiples) of a fundamental. The spectrum of such sound would display peaks corresponding to the fundamental frequency and to its integer multiples. Just to give a concrete example, imagine working at the sampling rate of 44100 Hz and summing 10 sinusoids. From the sampling theorem we know that, in our case, we can represent without aliasing all frequency components up to 22050 Hz. So, in order to avoid aliasing, the fundamental frequency should be lower than 2205 Hz. The Processing (with Beads library) code reported in table Table 4.1 implements a generator of sounds formed by 10 harmonic sinusoids. To produce such sounds it is necessary to click on a point of the display window. The x coordinate would vary with the fundamental frequency, and the window will show the spectral peaks corresponding to the generated harmonics. When we click on a point whose x coordinate is larger than $\frac{1}{10}$ of the window width, we still see ten spectral peaks. Otherwise, we violate the sampling theorem and aliasing will enter our representation.

Aliasing test: Applet to experience the effect of aliasing on sounds obtained by summation of 10 sinusoids in harmonic ratio [5]

```
    import beads.*; // import the beads library
import beads.Buffer;
import beads.BufferFactory;

AudioContext ac;
PowerSpectrum ps;

WavePlayer wavetableSynthesizer;
Glide frequencyGlide;
Envelope gainEnvelope;
Gain synthGain;

int L = 16384; // buffer size
int H = 10; //number of harmonics
float freq = 10.00; // fundamental frequency [Hz]
Buffer dSB;

void setup() {
  size(1024,200);

  frameRate(20);

  ac = new AudioContext();  // initialize AudioContext and create buffer

  frequencyGlide = new Glide(ac, 200, 10); // initial freq, and transition time
  dSB = new DiscreteSummationBuffer().generateBuffer(L, H, 0.5);
  wavetableSynthesizer = new WavePlayer(ac, frequencyGlide, dSB);

  gainEnvelope = new Envelope(ac, 0.0); // standard gain control of AudioContext
  synthGain = new Gain(ac, 1, gainEnvelope);
  synthGain.addInput(wavetableSynthesizer);
  ac.out.addInput(synthGain);

  // Short-Time Fourier Analysis
  ShortFrameSegmenter sfs = new ShortFrameSegmenter(ac);
  sfs.addInput(ac.out);
  FFT fft = new FFT();
  sfs.addListener(fft);
  ps = new PowerSpectrum();
  fft.addListener(ps);
  ac.out.addDependent(sfs);

  ac.start(); // start audio processing
  gainEnvelope.addSegment(0.8, 50); // attack envelope
}

void mouseReleased(){
  println("mouseX = " + mouseX);
}

void draw()
{
  background(0);

  text("click and move the pointer", 800, 20);
  frequencyGlide.setValue(float(mouseX)/width*22050/10); // set the fundamental fre
```

Table 4.1

4.1.2 2-D: Images

Let us assume we have a continuous distribution, on a plane, of values of luminance or, more simply stated, an image. In order to process it using a computer we have to reduce it to a sequence of numbers by means of sampling. There are several ways to sample an image, or read its values of luminance at discrete points. The simplest way is to use a regular grid, with spatial steps X e Y. Similarly to what we did for sounds, we define the spatial sampling rates $F_X = \frac{1}{X}$ and $F_Y = \frac{1}{Y}$. As in the one-dimensional case, also for two-dimensional signals, or images, sampling can be described by three facts and a theorem.

- The Fourier Transform of a discrete-space signal is a function (called **spectrum**) of two continuous variables ω_X and ω_Y, and it is periodic in two dimensions with periods 2π. Given a couple of values ω_X and ω_Y, the Fourier transform gives back a complex number that can be interpreted as magnitude and phase (translation in space) of the sinusoidal component at such spatial frequencies.
- Sampling the continuous-space signal $s(x, y)$ with the regular grid of steps X, Y, gives a discrete-space signal $s(m, n) = s(mX, nY)$, which is a function of the discrete variables m and n.
- Sampling a continuous-space signal with spatial frequencies F_X and F_Y gives a discrete-space signal whose spectrum is the periodic replication along the grid of steps F_X and F_Y of the original signal spectrum. The Fourier variables ω_X and ω_Y correspond to the frequencies (in cycles per meter) represented by the variables $f_X = \frac{\omega_X}{2\pi X}$ and $f_Y = \frac{\omega_Y}{2\pi Y}$.

The Figure 4.2 (Spectrum of a sampled image) shows an example of spectrum of a two-dimensional sampled signal. There, the continuous-space signal had all and only the frequency components included in the central hexagon. The hexagonal shape of the spectral support (region of non-null spectral energy) is merely illustrative. The replicas of the original spectrum are often called spectral **images**.

[5]See the file at <http://cnx.org/content/m13045/latest/./index.html>

Spectrum of a sampled image

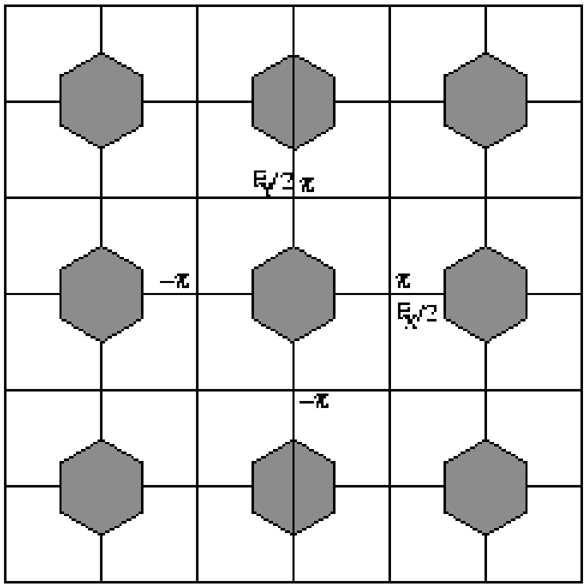


Figure 4.2

Given the above facts (p. 55), we can have an intuitive understanding of the Sampling Theorem.

Theorem 4.2: Sampling Theorem (in 2D)

A continuous-space signal $s(x, y)$, whose spectral content is limited to spatial frequencies belonging to the rectangle having semi-edges F_{bX} and F_{bY} (i.e., bandlimited) can be recovered from its sampled version $s(m, n)$ if the spatial sampling rates are larger than twice the respective bandwidths (i.e., if $F_X > 2F_{bX}$ and $F_Y > 2F_{bY}$)

In practice, the spatial sampling step can not be larger than the semi-period of the finest spatial frequency (or the finest detail) that is represented in the image. The reconstruction can only be done through a filter that eliminates all the spectral images but the one coming directly from the original continuous-space signal. In other words, the filter will cut all images whose frequency components are higher than the **Nyquist frequency** defined as $\frac{F_X}{2}$ and $\frac{F_Y}{2}$ along the two axes. The condition required by the sampling theorem (Theorem 4.2, Sampling Theorem (in 2D), p. 56) is equivalent to requiring that there are no overlaps between spectral images. If there were such overlaps, it wouldn't be possible to eliminate the copies of the original signal spectrum by means of filtering. In case of overlapping, a filter cutting all frequency components higher than the Nyquist frequency would give back a signal that is affected by aliasing.

We note how aliasing can be produced by down-sampling (or decimating) a sampled image. Starting from a discrete-space image, we can select only a subset of samples arranged in a regular grid. This will determine the periodic repetition of the spectral images, that will end up overlapping.

In order to explore the concepts of sampling, down-sampling, and aliasing, run the applet drawing ellipses [6]. With the keyboard arrow you can double or halve the horizontal and vertical sampling steps.

A simple introduction to the first elements of image processing is found in Digital Image Processing Basics[7].

[6]See the file at <http://cnx.org/content/m13045/latest/resampling_ellipse.html>
[7]"Digital Image Processing Basics" <http://cnx.org/content/m10973/latest/>

4.2 Quantization

With the adjective "digital" we indicate those systems that work on signals that are represented by numbers, with the (finite) precision that computing systems allow. Up to now we have considered discrete-time and discrete-space signals as if they were collections of infinite-precision numbers, or real numbers. Unfortunately, computers only allow to represent finite subsets of rational numbers. This means that our signals are subject to quantization.

For our purposes, the most interesting quantization is the linear one, which is usually occurring in the process of conversion of an analog signal into the digital domain. If the memory word dedicated to storing a number is made of b bits, then the range of such number is discretized into 2^b quantization levels. Any value that is found between two quantization levels can be approximated by truncation or rounding to the closest value. The Figure 4.3 (Sampling and quantization of an analog signal) shows an example of quantization with representation on 3 bits in two's complement[8].

Sampling and quantization of an analog signal

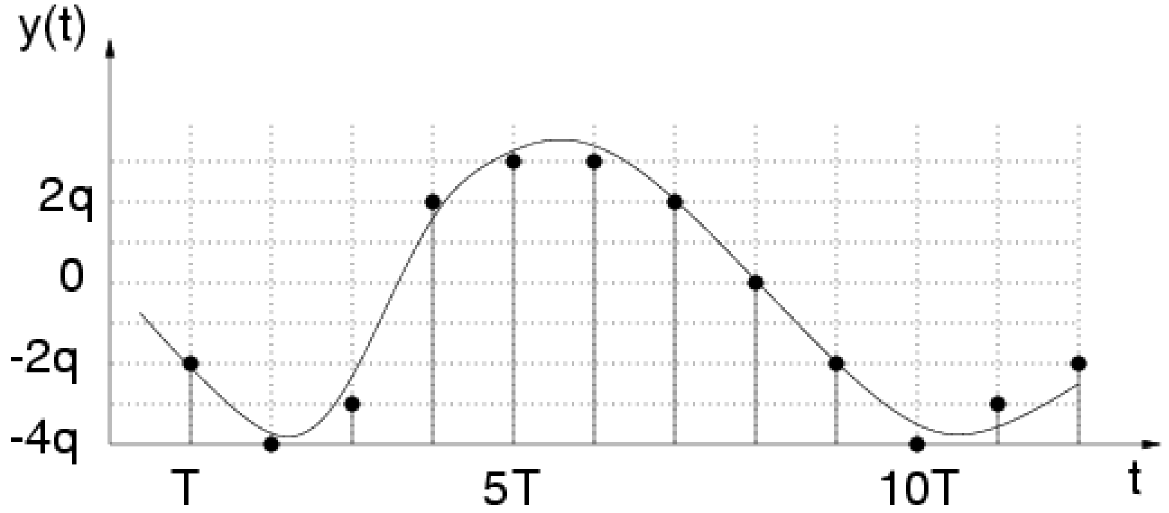


Figure 4.3

The approximation introduced by quantization manifests itself as a noise, called **quantization noise**. Often, for the analysis of sound-processing circuits, such noise is assumed to be white and de-correlated with the signal, but in reality it is perceptually tied to the signal itself, in such an extent that quantization can be perceived as an effect.

To have a visual and intuitive exploration of the phenomenon of quantization, consider the applet [9] that allows to vary between 1 and 8 the number of bits dedicated to the representation of each of the RGB channels representing color. The same number of bits is dedicated to the representation of an audio signal coupled to the image. The visual effect that is obtained by reducing the number of bits is similar to a **solarization**.

Exercise 4.1 ***(Solution on p. 59.)***

Extend the code of the applet Table 4.1 to add some interaction features:

[8]"Two's Complement and Fractional Arithmetic for 16-bit Processors" <http://cnx.org/content/m10808/latest/>
[9]See the file at <http://cnx.org/content/m13045/latest/quantagondoleBeads.html>

- Make the fundamental frequency of the automatically-generated sound changing randomly at each frame (see `random()`[10]).
- Make the framerate (and the metronome for generating notes) dependent on the horizontal position of the mouse (`mouseX`).
- Make the number of harmonics of the sound (i.e., its brightness) dependent on the vertical position of the mouse (`mouseY`).
- Paint the window background in such a way that moving from left to right the tint changes from blue to red. In this way, a blue color will correspond to a slow tempo, and a red color to a fast tempo.
- Make the color saturation of the background dependent on the vertical position of the mouse. In this way a sound with a few harmonics (low brightness) will correspond to a grayish color, while a sound rich of harmonics (high brightness) will correspond to a saturated color.
- Add a control to stop the computation and display of the spectrum and create an effect of image freezing, while sound continues to be generated (for instance by keeping the mouse button pressed).
- Add a control to cancel the dependence of tempo, brightness, and color saturation on mouse position (for instance by pressing a key).
- Add a control that, in case of image freezing (mouse button pressed), will stop the generation of new notes while "freezing" the current note.
- Finally, add a control that, in case of image freezing, will stop the sound generation and make the application silent.

[10]http://www.processing.org/reference/random_.html

Solutions to Exercises in Chapter 4

Solution to Exercise 4.1 (p. 57)

The proposed extensions are implemented in the Processing code[11].

[11]See the file at <http://cnx.org/content/m13045/latest/./aliasingfermoDBeads.pde>

Chapter 5

Signal Processing in Processing: Convolution and Filtering[1]

5.1 Systems

For our purposes, a system is any processing element that, given as input a sequence of samples $x(n)$, produces as output a sequence of samples $y(n)$. If the samples are coming from a temporal series we talk about **discrete-time systems**. In this module we will not be concerned with continuous-time processing, even though the principles here described can be generalized to functions of continuous variable. Instead, the sequence of number can come from the sampling of an image, and in this case it will be appropriate to talk of **discrete-space systems** and use two indeces m and n if sampling is done by a rectangular grid of rows and columns.

In this module we are only dealing with **linear systems**, thus meaning that the following principle holds:

Definition 5.1: Superposition principle
If y_1 and y_2 are the responses to the input sequences x_1 and x_2 then the input $a_1x_1+a_2x_2$ produces the response $a_1y_1+a_2y_2$

Another important concept is time (and space) invariance.

Definition 5.2: Time invariance
A system is time-invariant if a time shift of D samples in the input results in the same time shift in the output, i.e., $x(n-D)$ produces $y(n-D)$.

Cases of non-invariance are found whenever the system changes its characteristics in time (or space), for example as an effect of human control. Those systems where the sampling rate at the input is different than the one at the output are also non-invariant. For instance, decimators are time-variant systems.

A series connection of **linear time-invariant** (LTI) blocks is itself a linear and time-invariant system, and the order of blocks can be changed without affecting the input-output behavior.

LTI systems can be thoroughly described by the response they give to a unit-magnitude impulse.

Definition 5.3: The impulse in discrete time (space)
is the signal δ with value1 at the instant zero (in the point with coordinates $[0,0]$), and 0 in any other instant (point).

[1]This content is available online at <http://cnx.org/content/m13046/1.3/>.

5.2 Impulse response and convolution

We call h the output signal of a LTI system whose input is just an impulse. Such output signal is called **impulse response**. Since any discrete-time (-space) signal can be thought of as a weighted sum of translated impulses, each sample that shows up to the input **activates** an impulse response whose amplitude is determined by the value of the sample itself. Moreover, since the impulse responses are activated at a distance of one sampling step from each other and are extended over several samples, the effect of each input sample is distributed over time, on a number of contiguous samples of the output signal. Being the system linear and time-invariant, the successive impulse responses sum their effects. In other words, the system has memory of the past samples, previously given as input to the system, and it uses such memory to influence the present.

To have a physical analogy, we can think of regular strokes of a snare drum. The response to each stroke is distributed in time and overlaps with the responses to the following strokes.

Example 5.1

Consider the signal x that is zero everywhere but at the instants -1, 0, and 1 where it has values 1, 0.5, and 0.25, respectively. At every instant n, $x(n)$ can be expressed as $1\delta(n+1) + 0.5\delta(n) + 0.25\delta(n-1)$. By linearity, the output can be obtained by composition of carefully translated and weighted impulse responses: $y(n) = 1h(n+1) + 0.5h(n) + 0.25h(n-1)$.

To generalize the example Example 5.1 we can define the operation of **convolution**.

Definition 5.4: Convolution of two signals h and x

$y(n) = \mathrm{h} * \mathrm{x}(n) = \sum_{m=-\infty}^{\infty} x(m)\, h(n-m)$

The operation of convolution can be fully understood by the explicit construction of some examples of convolution product. The module Discrete-Time Convolution[2] gives the graphic construction of an examples and it offers pointers to other examples.

5.2.1 Properties

The properties of the convolution operation are well illustrated in the module Properties of Convolution[3]. The most interesting of such properties is the extension:

Property 5.1:

If $x(n)$ is extended over M_1 samples, and $h(n)$ is extended over M_2 samples, then the convolution product $y(n)$ is extended over $M_1 + M_2 - 1$ samples.

Therefore, the signal **convolution product** is longer than both the input signal and the impulse response.

Another interesting property is the commutativity of the convolution product, such that the input signal and the impulse response can change their roles without affecting the output signal.

5.3 Frequency response and filtering

The Fourier Transform[4] of the impulse response is called **Frequency Response** and it is represented with $H(\omega)$. The Fourier transform of the system output is obtained by multiplication of the Fourier transform of the input with the frequency response, i.e., $Y(\omega) = H(\omega)\, X(\omega)$.

The frequency response shapes, in a multiplicative fashion, the input-signal spectrum or, in other words, it performs some **filtering** by emphasizing some frequency components and attenuating some others. A filtering can also operate on the phases of the spectral components, by delaying them of different amounts.

Filtering can be performed in the **time domain** (or space domain), by the operation of convolution, or in the **frequency domain** by multiplication of the frequency response.

[2]"Discrete Time Convolution" <http://cnx.org/content/m10087/latest/>
[3]"Properties of Continuous Time Convolution" <http://cnx.org/content/m10088/latest/>
[4]"Derivation of the Fourier Transform" <http://cnx.org/content/m0046/latest/>

Exercise 5.1 ***(Solution on p. 64.)***

Take the impulse response that is zero everywhere but at the instants -1, 0, and 1 where it has values 1, 0.5, and 0.25, respectively. Redefine the filtering operation `filtra()` of the Sound Chooser[5] presented in the module Media Representation in Processing (Chapter 2). In this case filtering is operated in the time domain by convolution.

5.3.1 Causality

The notion of causality is quite intuitive and it corresponds to the experience of stimulating a system and getting back a response only at future time instants. For discrete-time LTI systems, this happens when the impulse response is zero for negative (discrete) time instants. Causal LTI systems can produce with no appreciable delay an output signal **sample-by-sample**, because the convolution operator acts only on present and past values of the input signal. In Exercise 5.1 the impulse response is not causal, but this is not a problem because the whole input signal is already available, and the filter can process the whole block of samples.

5.4 2D Filtering

The notions of impulse response, convolution, frequency response, and filtering naturally extend from 1D to 2D, thus giving the fundamental concepts of image processing.

Definition 5.5: Convolution of two 2D signals (images)

$y(m,n) = \mathrm{h} * \mathrm{x}(m,n) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} x(k,l)\, h(m-k, n-l)$

If x is the image that we are considering, it is easy to realize that convolution is performed by multiplication and translation in space of a **convolution mask** or **kernel** h (it is the impulse response of the processing system). As in the 1D case filtering could be interpreted as a combination of contiguous samples (where the extension of such cluster depends on the extension of the filter impulse response) that is repeated in time, sample by sample. So, in 2D space filtering can be interpreted as a combination of contiguous samples (pixels) in a cluster, whose extension is given by the convolution mask. The so-called memory of 1-D systems becomes in 2-D a sort of **distance effect**.

As in the 1D case, the Fourier transform of the impulse response is called **Frequency response** and it is indicated by $H(\omega_X, \omega_Y)$. The Fourier transform of the system output is obtained by Fourier-transforming the input and multiplying the result by the frequency response. $Y(\omega_X, \omega_Y) = H(\omega_X, \omega_Y)\, X(\omega_X, \omega_Y)$.

Exercise 5.2 ***(Solution on p. 64.)***

Consider the Processing code of the blurring example[6] and find the lines that implement the convolution operation.

[5]"Rappresentazione di Media in Processing" <http://cnx.org/content/m12664/latest/#sound_chooser>
[6]http://processing.org/learning/topics/blur.html

Solutions to Exercises in Chapter 5

Solution to Exercise 5.1 (p. 62)

```
  void filtra(float[] DATAF, float[] DATA, float WC, float RO) {
    //WC and RO are useless, here kept only to avoid rewriting other
    //parts of code
    for(int i = 2; i < DATA.length-1; i++){
      DATAF[i] = DATA[i+1] + 0.5*DATA[i] + 0.25*DATA[i-1];
      }
    }
```

Solution to Exercise 5.2 (p. 63)

```
for(int y=0; y<height; y++) {
  for(int x=0; x<width/2; x++) {
    float sum = 0;
    for(int k=-n2; k<=n2; k++) {
      for(int j=-m2; j<=m2; j++) {
        // Reflect x-j to not exceed array boundary
        int xp = x-j;
        int yp = y-k;
        //... omissis ...
//auxiliary code to deal with image boundaries
        sum = sum + kernel[j+m2][k+n2] * red(get(xp, yp));
      }
    }
    output[x][y] = int(sum);
  }
}
```

Chapter 6

Convolution - Discrete time[1]

6.1 Introduction

The idea of **discrete-time convolution** is exactly the same as that of continuous-time convolution[2]. For this reason, it may be useful to look at both versions to help your understanding of this extremely important concept. Convolution is a very powerful tool in determining a system's output from knowledge of an arbitrary input and the system's impulse response.

It also helpful to see convolution graphically, i.e. by using transparencies or Java Applets. Johns Hopkins University[3] has an excellent Discrete time convolution[4] applet. Using this resource will help understanding this crucial concept.

6.2 Derivation of the convolution sum

We know that any discrete-time signal can be represented by a summation of scaled and shifted discrete-time impulses, see here[5]. Since we are assuming the system to be linear and time-invariant, it would seem to reason that an input signal comprised of the sum of scaled and shifted impulses would give rise to an output comprised of a sum of scaled and shifted impulse responses. This is exactly what occurs in **convolution**. Below we present a more rigorous and mathematical look at the derivation:

Letting $\mathcal{H}$ be a discrete time LTI system, we start with the folowing equation and work our way down the the convoluation sum.

$$\begin{aligned} y(n) &= \mathcal{H}(x(n)) \\ &= \mathcal{H}\left(\sum_{k=-\infty}^{\infty} x(k)\,\delta(n-k)\right) \\ &= \sum_{k=-\infty}^{\infty} \mathcal{H}(x(k)\,\delta(n-k)) \\ &= \sum_{k=-\infty}^{\infty} x(k)\,\mathcal{H}(\delta(n-k)) \\ &= \sum_{k=-\infty}^{\infty} x(k)\,h(n-k) \end{aligned} \tag{6.1}$$

Let us take a quick look at the steps taken in the above derivation. After our initial equation we rewrite the function $x(n)$ as a sum of the function times the unit impulse. Next, we can move around the $\mathcal{H}$ operator and the summation because $\mathcal{H}(\cdot)$ is a linear, DT system. Because of this linearity and the fact that $x(k)$ is a constant, we pull the constant out and simply multiply it by $\mathcal{H}(\cdot)$. Finally, we use the fact that $\mathcal{H}(\cdot)$ is time invariant in order to reach our final state - the convolution sum!

[1]This content is available online at <http://cnx.org/content/m11539/1.4/>.
[2]"Continuous Time Convolution" <http://cnx.org/content/m10085/latest/>
[3]http://www.jhu.edu
[4]http://www.jhu.edu/signals
[5]"Discrete time signals": Section The unit sample <http://cnx.org/content/m11476/latest/#s3s1>

Above the summation is taken over all integers. Howerer, in many practical cases either $x(n)$ or $h(n)$ or both are finite, for which case the summations will be limited. The convolution equations are simple tools which, in principle, can be used for all input signals. Following is an example to demonstrate convolution; how it is calculated and how it is interpreted.

6.2.1 Graphical illustration of convolution properties

A quick graphical example may help in demonstrating why convolution works.

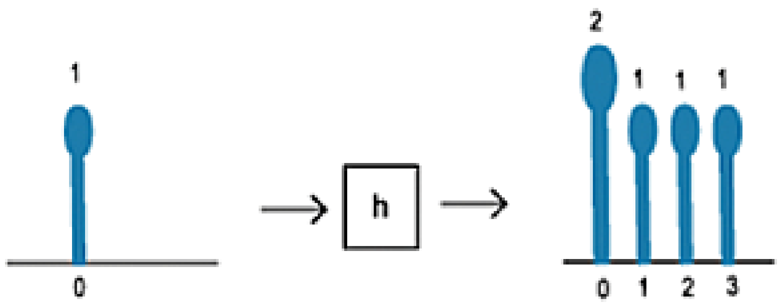


Figure 6.1: A single impulse input yields the system's impulse response.

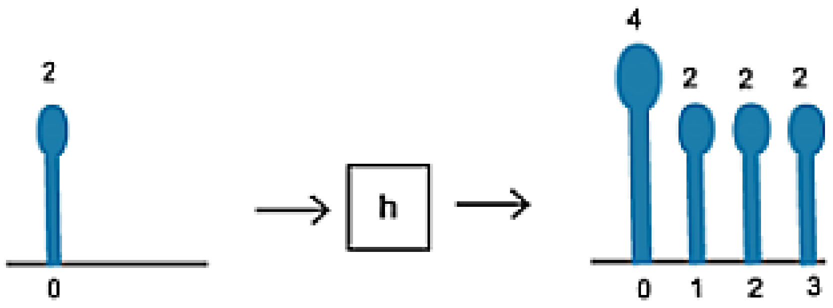


Figure 6.2: A scaled impulse input yields a scaled response, due to the scaling property of the system's linearity.

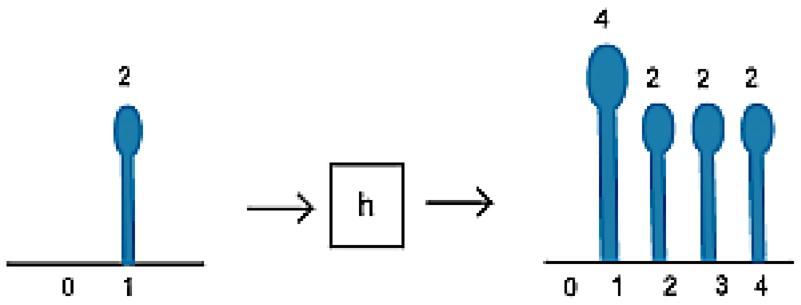


Figure 6.3: We now use the time-invariance property of the system to show that a delayed input results in an output of the same shape, only delayed by the same amount as the input.

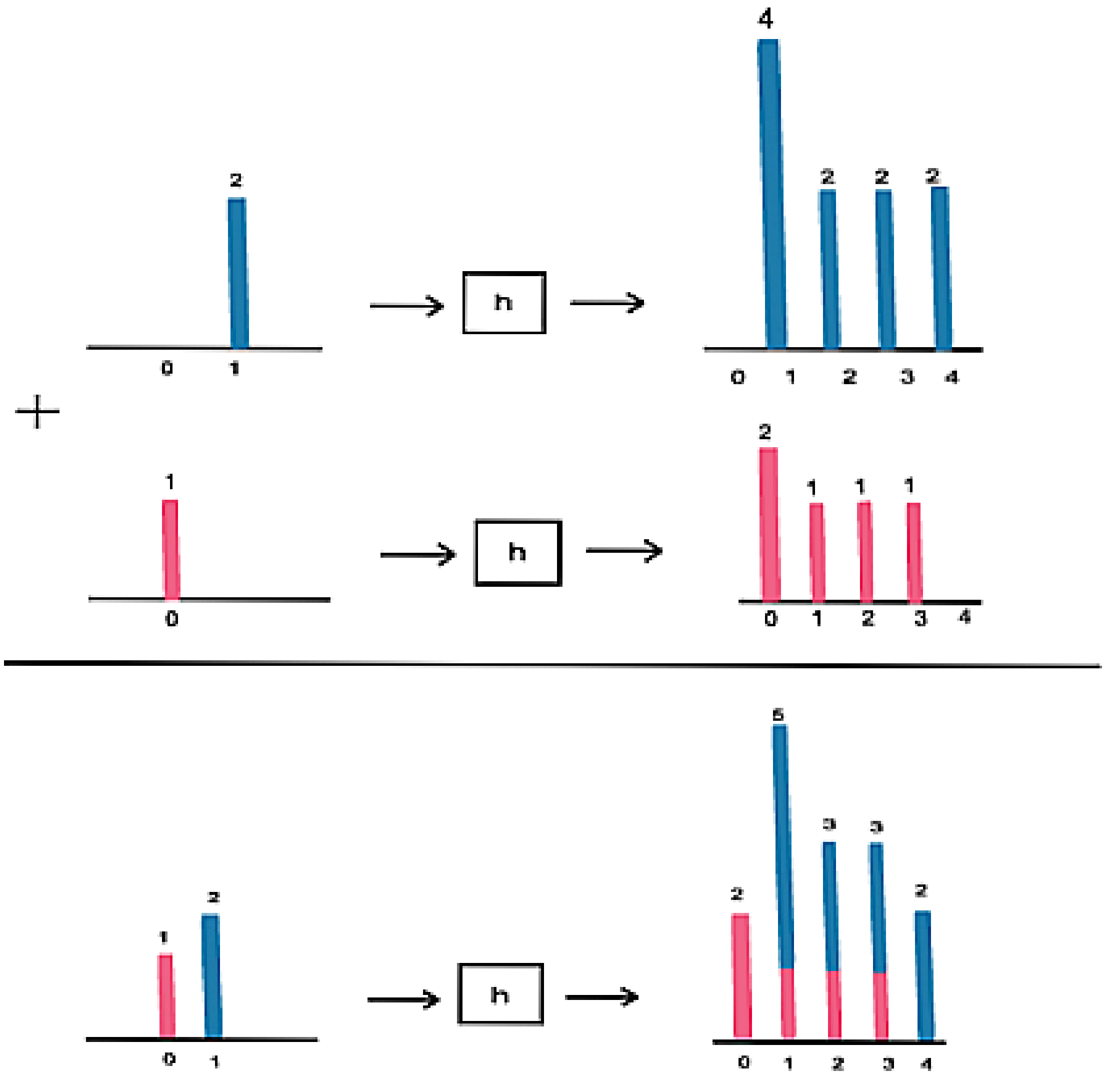


Figure 6.4: We now use the additivity portion of the linearity property of the system to complete the picture. Since any discrete-time signal is just a sum of scaled and shifted discrete-time impulses, we can find the output from knowing the input and the impulse response.

6.3 Convolution Sum

As mentioned above, the convolution sum provides a concise, mathematical way to express the output of an LTI system based on an arbitrary discrete-time input signal and the system's response. The **convolution sum** is expressed as

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)\, h(n-k) \tag{6.2}$$

As with continuous-time, convolution is represented by the symbol *, and can be written as

$$y(n) = x(n) * h(n) \tag{6.3}$$

By making a simple change of variables into the convolution sum, $k = n - k$, we can easily show that convolution is **commutative**:

$$\begin{aligned} y(n) &= x(n) * h(n) \\ &= h(n) * x(n) \end{aligned} \tag{6.4}$$

From (6.4) we get a convolution sum that is equivivalent to the sum in (6.2):

$$y(n) = \sum_{k=-\infty}^{\infty} h(k)\, x(n-k) \tag{6.5}$$

For more information on the characteristics of convolution, read about the Properties of Convolution[6].

6.4 Convolution Through Time (A Graphical Approach)

In this section we will develop a second graphical interpretation of discrete-time convolution. We will begin this by writing the convolution sum allowing x to be a causal, length-m signal and h to be a causal, length-k, LTI system. This gives us the finite summation,

$$y(n) = \sum_{l=0}^{m-1} x(l)\, h(n-l) \tag{6.6}$$

Notice that for any given n we have a sum of the m products of $x(l)$ and a time-delayed $h(n-l)$. This is to say that we multiply the terms of x by the terms of a time-reversed h and add them up.

Going back to the previous example:

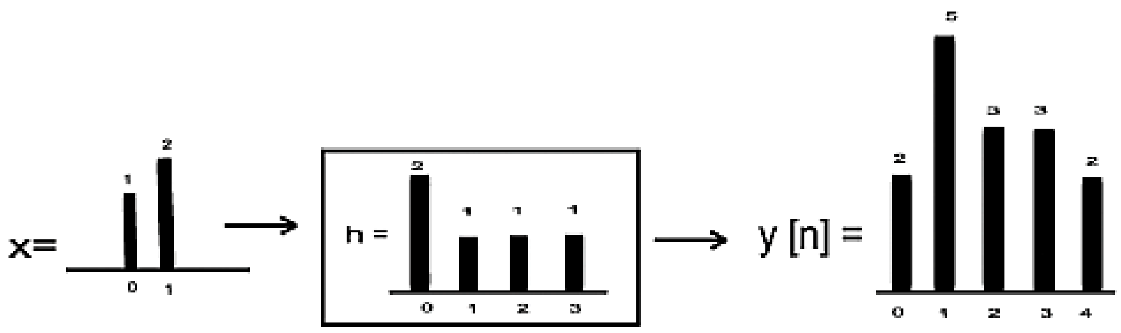


Figure 6.5: This is the end result that we are looking to find.

[6]"Properties of Continuous Time Convolution" <http://cnx.org/content/m10088/latest/>

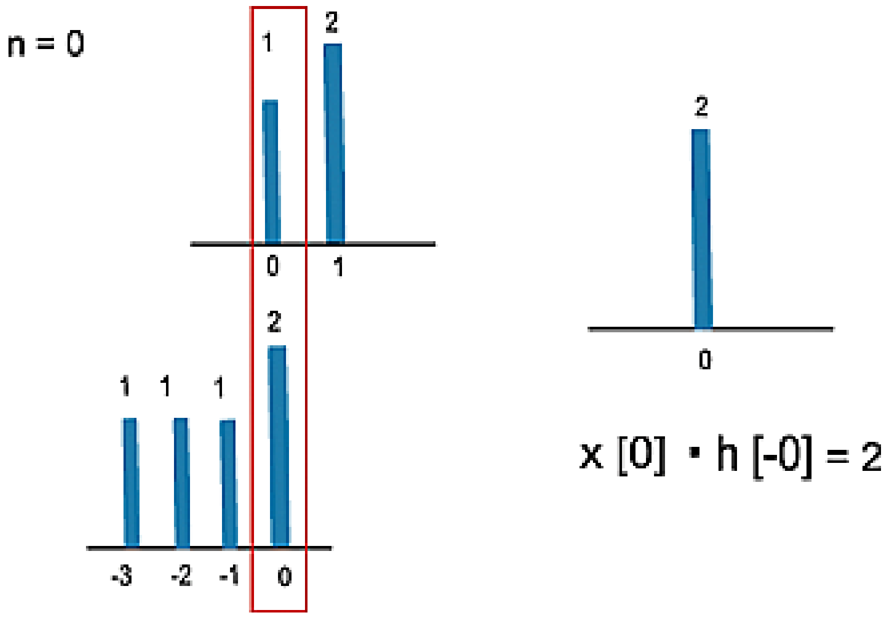


Figure 6.6: Here we reverse the impulse response, h , and begin its traverse at time 0.

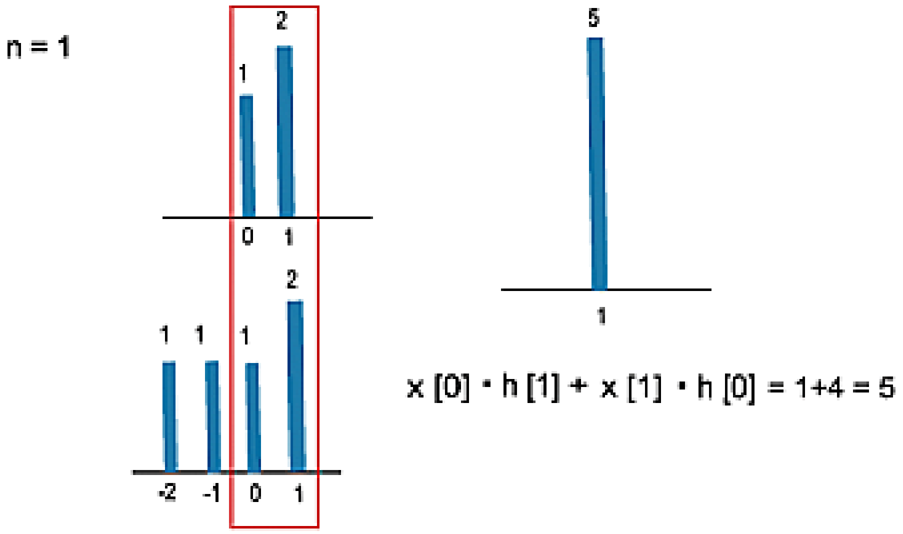


Figure 6.7: We continue the traverse. See that at time 1, we are multiplying two elements of the input signal by two elements of the impulse respone.

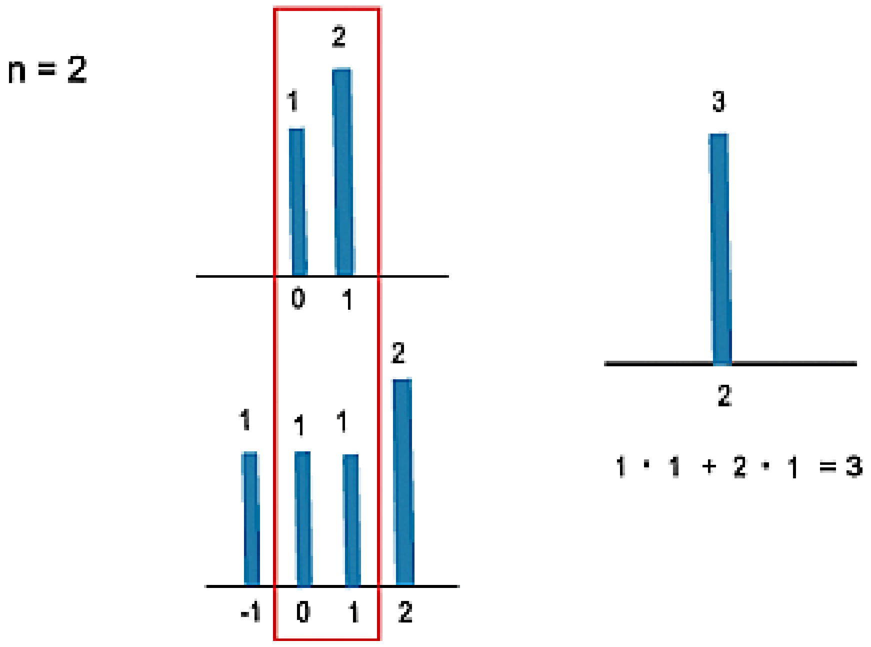


Figure 6.8

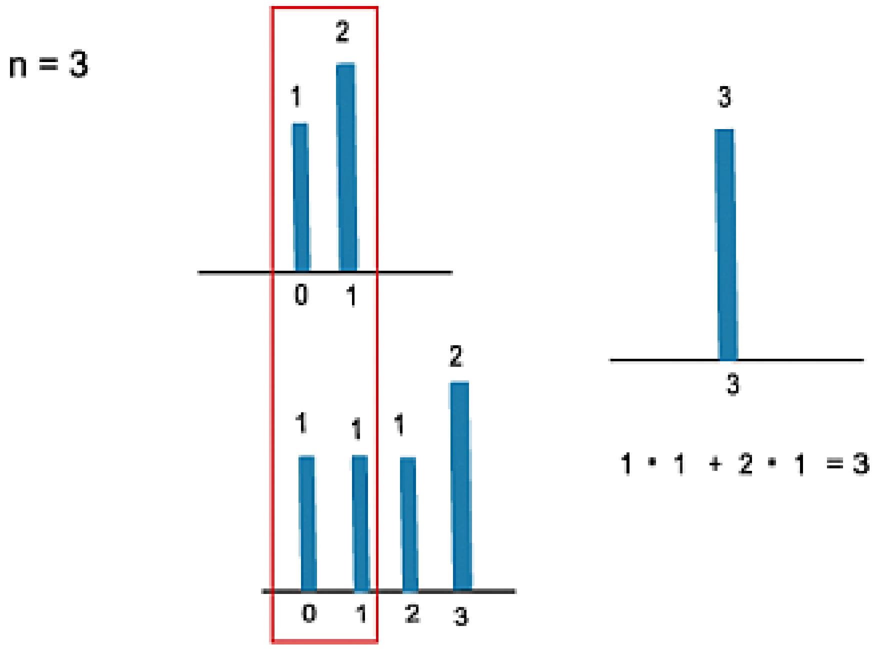


Figure 6.9: If we follow this through to one more step, $n = 4$, then we can see that we produce the same output as we saw in the intial example.

What we are doing in the above demonstration is reversing the impulse response in time and "walking it across" the input signal. Clearly, this yields the same result as scaling, shifting and summing impulse responses.

This approach of time-reversing, and sliding across is a common approach to presenting convolution, since it demonstrates how convolution builds up an output through time.

Chapter 7

Signal Processing in Processing: Elementary Filters[1]

7.1 FIR filters

The **Finite Impulse Response** (FIR) filters are all those filters characterised by an impulse response with a finite number of samples. They are realized by the operation of convolution[2]. For each sample of the convolution product a weighted sum of a finite number of input samples is computed.

7.1.1 Averaging filter

The simplest non trivial FIR filter is the filter that computes the running average of two contiguous samples, and the corresponding convolution can be expressed as

$$y(n) = 0.5x(n) + 0.5x(n-1) \tag{7.1}$$

. The impulse response has values 0.5 at instants 0 and 1, and zero anywhere else.

If we put a sinusoidal signal into the filter, the output will still be a sinusoidal signal scaled in amplitude and delayed in phase according to the frequency response [3], which is

$$H(\omega) = \cos\left(\frac{\omega}{2}\right) e^{-\left(i\frac{\omega}{2}\right)} \tag{7.2}$$

and its magnitude and phase are represented in Figure 7.1 (Magnitude and phase response for the averaging filter).

[1]This content is available online at <http://cnx.org/content/m13047/1.4/>.

[2]"Signal Processing in Processing: Convoluzione e Filtraggio": Section Risposta all'impulso e convoluzione <http://cnx.org/content/m12809/latest/#convolution>

[3]"Signal Processing in Processing: Convoluzione e Filtraggio" <http://cnx.org/content/m12809/latest/#freqrespp>

Magnitude and phase response for the averaging filter

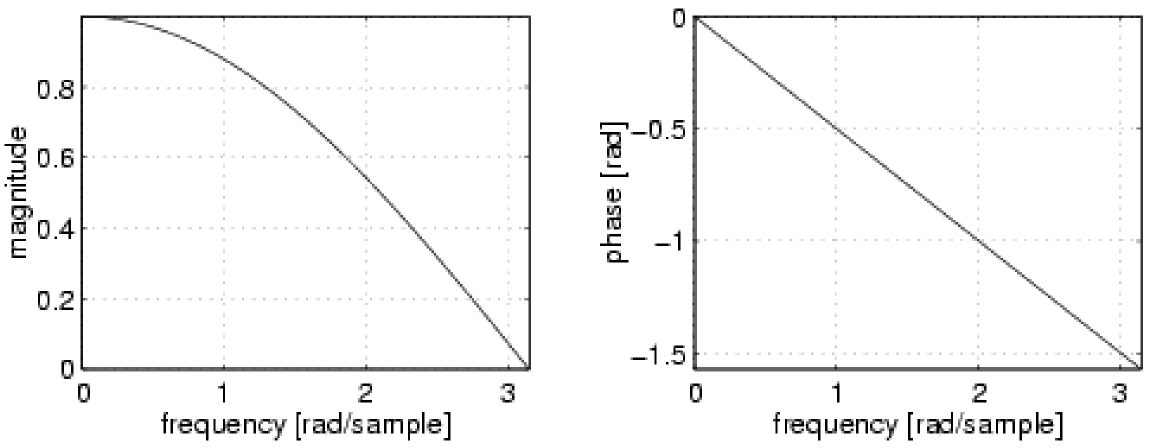


Figure 7.1

The one just presented is a first-order (or lenght 2) filter, because it uses only one sample of the past sequence of input. From Figure 7.1 (Magnitude and phase response for the averaging filter) we see that the frequency response is of the **low-pass** kind, because the high frequencies are attenuated as compared to the low frequencies. Attenuating high frequencies means smoothing the rapid signal variations. If one wants a steeper frequency response from an FIR filter, the order must be increased or, in other words, more samples of the input signal have to be processed by the convolution operator to give one sample of output.

7.1.2 Symmetric second-order FIR filter

A symmetric second-order FIR filter has an impulse response whose form is $[a_0, a_1, a_0]$, and the frequency response turns out to be $H(\omega) = (a_1 + 2a_0 \cos(\omega))\, e^{-(i\omega)}$. Convolution can be expressed as

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_0 x(n-2) \tag{7.3}$$

In the special case where $a_0 = 0.17654$ and $a_1 = 0.64693$ the frequency response (magnitude and phase) is represented in Figure 7.2 (Magnitude and phase response of a second-order FIR filter).

Magnitude and phase response of a second-order FIR filter

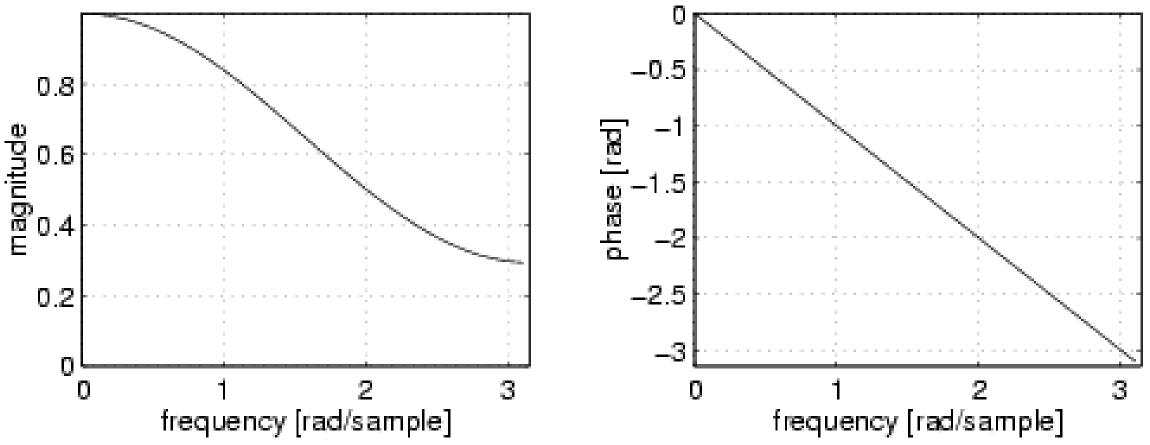
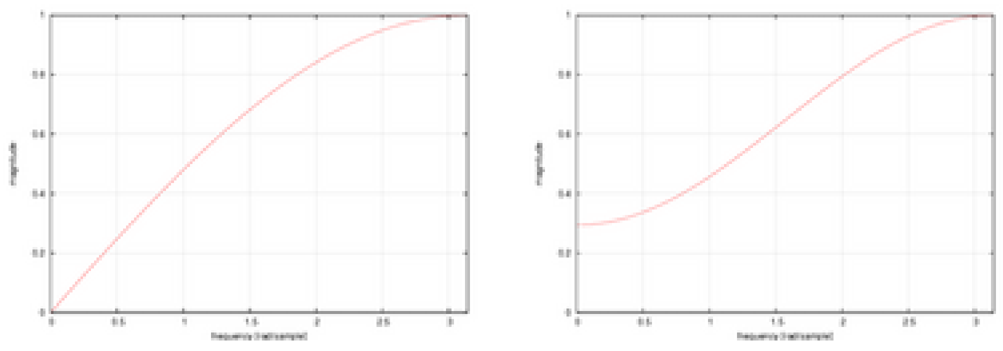


Figure 7.2

7.1.3 High-pass filters

Given the simple low-pass filters that we have just seen, it is sufficient to change sign to a coefficient to obtain a **high-pass** kind of response, i.e. to emphasize high frequencies as compared to low frequencies. For example, the Figure 7.3 (Frequency response (magnitude) of first- (left) and second- (right) order high-pass FIR filter.) displays the magnitude of the frequency responses of high-pass FIR filters of the first and second order, whose impulse responses are, respectively $[0.5, -0.5]$ and $[0.17654, -0.64693, 0.17654]$.

Frequency response (magnitude) of first- (left) and second- (right) order high-pass FIR filter.



(a) (b)

Figure 7.3

To emphasize high frequencies means to make rapid variations of signal more evident, being those variations time transients in the case of sounds, or contours in the case of images.

7.1.4 FIR filters in 2D

In 2D, the impulse response of an FIR filter is a convolution mask with a finite number of elements, i.e. a matrix. In particular, the **averaging filter** can be represented, for example, by the convolution matrix $\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$.

Example 7.1: Noise cleaning
The low-pass filters (and, in particular, the smoothing filters) perform some sort of **smoothing** of the input signal, in the sense that the resulting signal has a smoother design, where abrupt discontinuities are less evident. This can serve the purpose of reducing the perceptual effect of noises added to audio signals or images. For example, the code reported below (p. 78) loads an image, it corrupts with white noise, and then it filters half of it with an averaging filter, thus obtaining Figure 7.4 (Smoothing).

Smoothing

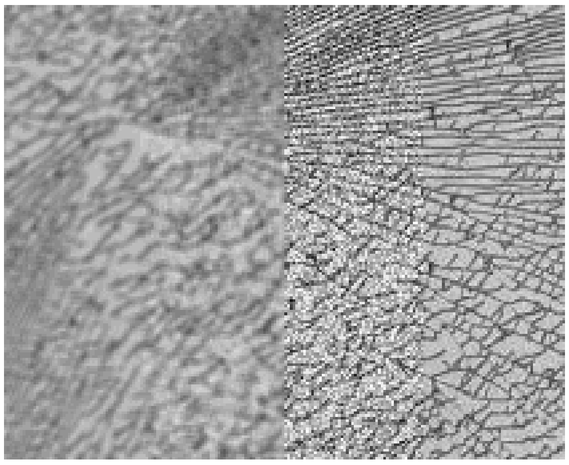


Figure 7.4

```
    // smoothed_glass
// smoothing filter, adapted from REAS:
// http://processing.org/learning/topics/blur.html

size(210, 170);
PImage a;  // Declare variable "a" of type PImage
a = loadImage("vetro.jpg"); // Load the images into the program
image(a, 0, 0); // Displays the image from point (0,0)

// corrupt the central strip of the image with random noise
```

```
float noiseAmp = 0.2;
loadPixels();
for(int i=0; i<height; i++) {
  for(int j=width/4; j<width*3/4; j++) {
    int rdm = constrain((int)(noiseAmp*random(-255, 255) +
      red(pixels[i*width + j])), 0, 255);
    pixels[i*width + j] = color(rdm, rdm, rdm);
  }
}
updatePixels();

int n2 = 3/2;
int m2 = 3/2;
float  val = 1.0/9.0;
int[][] output = new int[width][height];
float[][] kernel = { {val, val, val},
                     {val, val, val},
                     {val, val, val} };

// Convolve the image
for(int y=0; y<height; y++) {
  for(int x=0; x<width/2; x++) {
    float sum = 0;
    for(int k=-n2; k<=n2; k++) {
      for(int j=-m2; j<=m2; j++) {
        // Reflect x-j to not exceed array boundary
        int xp = x-j;
        int yp = y-k;
        if (xp < 0) {
          xp = xp + width;
        } else if (x-j >= width) {
          xp = xp - width;
        }
        // Reflect y-k to not exceed array boundary
        if (yp < 0) {
          yp = yp + height;
        } else if (yp >= height) {
          yp = yp - height;
        }
        sum = sum + kernel[j+m2][k+n2] * red(get(xp, yp));
      }
    }
    output[x][y] = int(sum);
  }
}

// Display the result of the convolution
// by copying new data into the pixel buffer
loadPixels();
for(int i=0; i<height; i++) {
  for(int j=0; j<width/2; j++) {
    pixels[i*width + j] =
      color(output[j][i], output[j][i], output[j][i]);
  }
}
updatePixels();
```

For the purpose of smoothing, it is common to create a convolution mask by reading the values of a Gaussian bell in two variables. A property of gaussian functions is that their Fourier transform is itself gaussian. Therefore, impulse response and frequency response have the same shape. However, the transform of a thin bell is a large bell, and vice versa. The larger the bell, the more evident the smoothing effect will be, with consequential loss of details. In visual terms, a gaussian filter produces an effect similar to that of an opalescent glass superimposed over the image. An example of Gaussian bell is $\frac{1}{273}\begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix}$.

Conversely, if the purpose is to make the contours and salient tracts of an image more evident (**edge crispening** or sharpening), we have to perform a high-pass filtering. Similarly to what we saw in Section 7.1.3 (High-pass filters) this can be done with a convolution matrix whose central value has opposite sign as compared to surrounding values. For instance, the convolution matrix $\begin{pmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{pmatrix}$ produces the effect of Figure 7.5 (Edge crispening).

Edge crispening

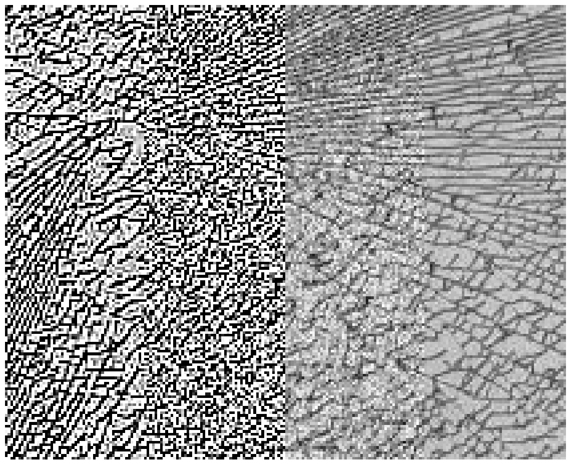


Figure 7.5

7.1.4.1 Non-linear filtering: median filter

A filter whose convolution mask is signal-dependent looses its characteristics of linearity. Median filters use the mask to select a set of pixels of the input images, and replace the central pixel of the mask with the median value of the selected set. Given a set of N (odd) numbers, the median element of the set is the one that separates $\frac{N-1}{2}$ smaller elements from $\frac{N-1}{2}$ larger elements. A typical median filter mask is cross-shaped. For example, a 3×3 mask can cover, when applied to a certain image point, the pixels with values $\begin{pmatrix} x & 4 & x \\ 7 & 99 & 12 \\ x & 9 & x \end{pmatrix}$, thus replacing the value 99 with the mean value 9.

Exercise 7.1 ***(Solution on p. 84.)***

Rewrite the filtering operation `filtra()` of the Sound Chooser (Table 2.3) presented in the module Media Representation in Processing (Chapter 2) in such a way that it implements the FIR filter whose frequency response is represented in Figure 7.2 (Magnitude and phase response of a second-order FIR filter). What happens if the filter is applied more than once?

Exercise 7.2 ***(Solution on p. 84.)***

Considered the Processing code of the blurring example[4] contained in the Processing examples[5] , modify it so that it performs a Gaussian filtering.

Exercise 7.3 ***(Solution on p. 85.)***

Modify the code of Example 7.1 (Noise cleaning) so that the effects of the averaging filter (p. 78) mask and the $\frac{1}{10}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$ are compared. What happens if the central value of the convolution mask is increased further? Then, try to implement the median filter with a 3×3 cross-shaped mask.

7.2 IIR Filters

The filtering operation represented by (7.1) is a particular case of **difference equation**, where a sample of output is only function of the input samples. More generally, it is possible to construct **recursive** difference equations, where any sample of output is a function of one or more other output samples.

$$y(n) = 0.5y(n-1) + 0.5x(n) \tag{7.4}$$

allows to compute (causally) each sample of the output by only knowing the output at the previous instant and the input at the same instant. It is easy to realize that by feeding the system represented by (7.4) with an impulse, we obtain the infinite-length sequence $y = [0.5, 0.25, 0.125, 0.0625, ...]$. For this purpose, filters of this kind are called **Infinite Impulse Response** (IIR) filters. The **order** of an IIR filter is equal to the number of past output samples that it has to store for processing, as dictated by the difference equation. Therefore, the filter of (7.4) is a first-order filter. For a given filter order, IIR filters allow frequency responses that are steeper than those of FIR filters, but phase distortions are always introduced by IIR filters. In other words, the different spectral components are delayed by different time amounts. For example, Figure 7.6 (Magnitude and phase response of the IIR first-order filter) shows the magnitude and phase responses for the first-order IIR filter represented by the difference equation (7.4). Called a the coefficient that weights the dependence on the output previous value (0.5 in the specific (7.4)), the impulse response takes the form $h(n) = a^n$. The more a is closer to 1, the more sustained is the impulse response in time, and the frequency response increases its steepness, thus becoming emphasizing its low-pass character. Obviously, values of a larger than 1 gives a divergent impulse response and, therefore, an **unstable** behavior of the filter.

[4]http://processing.org/learning/topics/blur.html
[5]http://processing.org/learning/topics/

Magnitude and phase response of the IIR first-order filter

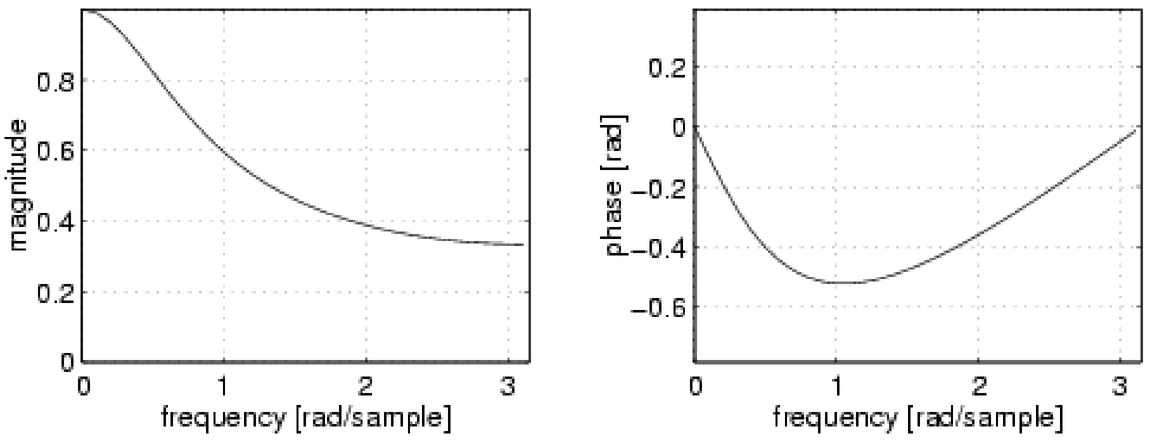


Figure 7.6

IIR filters are widely used for one-dimensional signals, like audio signals, especially for real-time sample-by-sample processing. Vice versa, it doesn't make much sense to extend recursive processing onto two dimensions. Therefore, in image processing FIR filters are mostly used.

7.2.1 Resonant filter

In the audio field, second-order IIR filters are particularly important, because they realize an elementary resonator. Given the difference equation

$$y(n) = a_1 y(n-1) + a_2 y(n-2) + b_0 x(n) \quad (7.5)$$

one can verify that it produces the frequency response of Figure 7.7 (Magnitude and phase response of the second-order IIR filter). The coefficients that gives dependence on the past can be expressed as $a_1 = 2r\cos(\omega_0)$ and $a_2 = -r^2$, where ω_0 is the frequency of the resonance peak and r gives peaks that gets narrower when approaching 1.

Magnitude and phase response of the second-order IIR filter

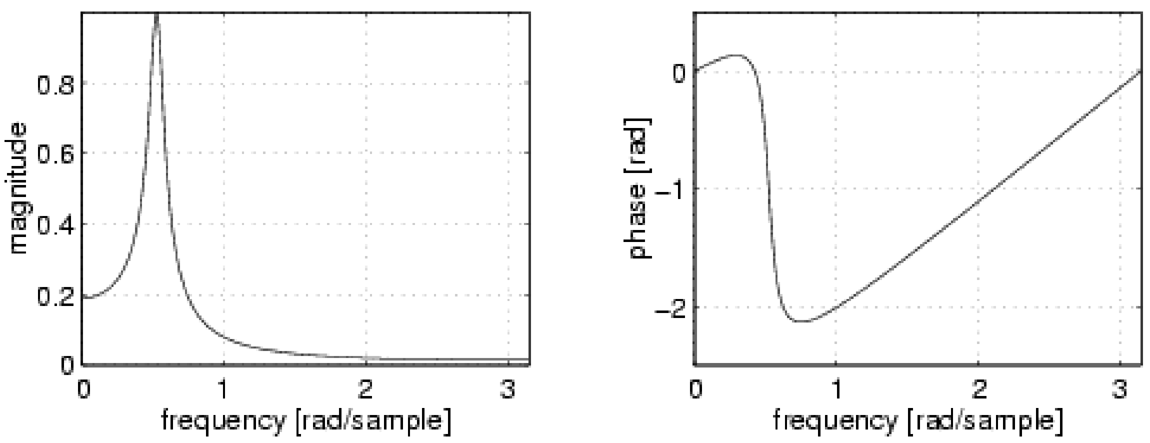


Figure 7.7

Exercise 7.4

Verify that the filtering operation `filtra()` of the Sound Chooser (Table 2.3) presented in module Media Representation in Processing (Chapter 2) implements an IIR resonant filter. What is the relation between r and the mouse position along the small bars?

Solutions to Exercises in Chapter 7

Solution to Exercise 7.1 (p. 81)

```
 //filtra = new function
void filtra(float[] DATAF, float[] DATA, float a0, float a1) {

  for(int i = 3; i < DATA.length; i++){
    DATAF[i] = a0*DATA[i]+a1*DATA[i-1]+a0*DATA[i-2];//Symmetric FIR filter of the second order
  }
}
```

By writing a `for` loop that repeats the filtering operation a certain number of times, one can verify that the effect of filtering is emphasized. This intuitive result is due to the fact that, as far as the signal is concerned, going through m filters of order N (in our case $N = 2$) is equivalent to going through a single filter of order mN

Solution to Exercise 7.2 (p. 81)

```
    // smoothing Gaussian filter, adapted from REAS:
// http://processing.org/learning/topics/blur.html

size(200, 200);
PImage a;  // Declare variable "a" of type PImage
a = loadImage("vetro.jpg"); // Load the images into the program
image(a, 0, 0); // Displays the image from point (0,0)

int n2 = 5/2;
int m2 = 5/2;
int[][] output = new int[width][height];
float[][] kernel = { {1, 4, 7, 4, 1},
                     {4, 16, 26, 16, 4},
                     {7, 26, 41, 26, 7},
                     {4, 16, 26, 16, 4},
                     {1, 4, 7, 4, 1} };

for (int i=0; i<5; i++)
  for (int j=0; j< 5; j++)
    kernel[i][j] = kernel[i][j]/273;
// Convolve the image
for(int y=0; y<height; y++) {
  for(int x=0; x<width/2; x++) {
    float sum = 0;
    for(int k=-n2; k<=n2; k++) {
      for(int j=-m2; j<=m2; j++) {
        // Reflect x-j to not exceed array boundary
        int xp = x-j;
        int yp = y-k;
        if (xp < 0) {
          xp = xp + width;
        } else if (x-j >= width) {
          xp = xp - width;
        }
        // Reflect y-k to not exceed array boundary
        if (yp < 0) {
          yp = yp + height;
        } else if (yp >= height) {
          yp = yp - height;
        }
        sum = sum + kernel[j+m2][k+n2] * red(get(xp, yp));
      }
    }
    output[x][y] = int(sum);
  }
}

// Display the result of the convolution
// by copying new data into the pixel buffer
loadPixels();
for(int i=0; i<height; i++) {
  for(int j=0; j<width/2; j++) {
    pixels[i*width + j] = color(output[j][i], output[j][i], output[j][i]);
  }
}
updatePixels();
```

Solution to Exercise 7.3 (p. 81)
Median filter:

```
// smoothed_glass
// smoothing filter, adapted from REAS:
// http://www.processing.org/learning/examples/blur.html

size(210, 170);
PImage a;  // Declare variable "a" of type PImage
a = loadImage("vetro.jpg"); // Load the images into the program
image(a, 0, 0); // Displays the image from point (0,0)

// corrupt the central strip of the image with random noise
float noiseAmp = 0.1;
loadPixels();
for(int i=0; i<height; i++) {
  for(int j=width/4; j<width*3/4; j++) {
    int rdm = constrain((int)(noiseAmp*random(-255, 255) +
      red(pixels[i*width + j])), 0, 255);
    pixels[i*width + j] = color(rdm, rdm, rdm);
  }
}
updatePixels();

int[][] output = new int[width][height];
int[] sortedValues = {0, 0, 0, 0, 0};
int grayVal;

// Convolve the image
for(int y=0; y<height; y++) {
  for(int x=0; x<width/2; x++) {
    int indSort = 0;
    for(int k=-1; k<=1; k++) {
      for(int j=-1; j<=1; j++) {
        // Reflect x-j to not exceed array boundary
        int xp = x-j;
        int yp = y-k;
        if (xp < 0) {
          xp = xp + width;
        } else if (x-j >= width) {
          xp = xp - width;
        }
        // Reflect y-k to not exceed array boundary
        if (yp < 0) {
          yp = yp + height;
        } else if (yp >= height) {
          yp = yp - height;
        }
        if ((((k != j) && (k != (-j))) ) || (k == 0)) { //cross selection
          grayVal = (int)red(get(xp, yp));
          indSort = 0;
          while (grayVal < sortedValues[indSort]) {indSort++; }
          for (int i=4; i>indSort; i--) sortedValues[i] = sortedValues[i-1];
          sortedValues[indSort] = grayVal;
        }
      }
    }
    output[x][y] = int(sortedValues[2]);
    for (int i=0; i< 5; i++) sortedValues[i] = 0;
  }
}

// Display the result of the convolution
// by copying new data into the pixel buffer
loadPixels();
for(int i=0; i<height; i++) {
  for(int j=0; j<width/2; j++) {
    pixels[i*width + j] =
      color(output[j][i], output[j][i], output[j][i]);
  }
}
```

```
updatePixels();
```

Chapter 8

Textures in Processing[1]

8.1 Color Interpolation

As seen in Graphic Composition in Processing[2], one can obtain surfaces as collections of polygons by means of the definition of a vertex within the couple `beginShape() - endShape()`. It is possible to assign a color to one or more vertices, in order to make the color variations continuous (**gradient**). For example, you can try to run the code

```
size(200,200,P3D);
        beginShape(TRIANGLE_STRIP);
        fill(240,   0, 0); vertex(20,31, 33);
        fill(240, 150, 0); vertex(80, 40, 38);
        fill(250, 250, 0); vertex(75, 88, 50);
                           vertex(49, 85, 74);
        endShape();
```

in order to obtain a continuous nuance from red to yellow in the strip of two triangles.

8.1.1 Bilinear Interpolation

The graphical system performs an interpolation of color values assigned to the vertices. This type of **bilinear interpolation** is defined in the following way:

- For each polygon of the collection
- · For each side of the polygon one assigns to each point on the segment the color obtained by means of linear interpolation of the colors of the vertices i e j that define the polygon: $C_{ij}\left(\alpha\right) = \left(1 - \alpha C_i + \alpha C_j\right.$
- · A **scan line** scans the polygon (or, better, its projection on the image window) intersecting at each step two sides in two points l ed r whose colors have already been identified as C_l e C_r. In each point of the scan line the color is determined by linear interpolation $C_{lr}\left(\beta\right) = \left(1 - \beta C_l + \beta C_r\right.$

A significative example of interpolation of colors associated to the vertices of a cube can be found in examples of Processing[3] , in the code RGB Cube[4] .

[1]This content is available online at <http://cnx.org/content/m13048/1.4/>.
[2]"Composizione Grafica in Processing" <http://cnx.org/content/m12665/latest/>
[3]http://processing.org/learning/3d/
[4]http://processing.org/learning/3d/rgbcube.html

8.2 Texture

When modeling a complex scene by means of a composition of simple graphical elements one cannot go beyond a certain threshold of complexity. Let us think about the example of a modelization of a natural scene, where one has to represent each single vegetal element, including the grass of a meadow. It is unconceivable to do this manually. It would be possible to set and control the grass elements by means of some algorithms. This is an approach taken, for example, in rendering the hair and skin of characters of the most sophisticated animation movies (see for example, the Incredibles[5]). Otherwise, especially in case of interactive graphics, one has to resort to using **textures**. In other words, one employs images that represent the visual texture of the surfaces and map them on the polygons that model the objects of the scene. In order to have a qualitative rendering of the surfaces it is necessary to limit the detail level to fragments not smaller than one pixel and, thus, the **texture mapping** is inserted in the rendering chain at the **rastering** level of the graphic primitives, i.e. where one passes from a 3D geometric description to the illumination of the pixels on the display. It is at this level that the removal of the hidden surfaces takes place, since we are interested only in the visible fragments.

In Processing, a texture is defined within a block `beginShape()` - `endShape()` by means of the function `texture()` that has as unique parameter a variable of type `PImage`. The following calls to `vertex()` can contain, as last couple of parameters, the point of the texture corresponding to the vertex. In fact, each texture image is parameterized by means of two variables u and v, that can be referred directly to the line and column of a **texel** (pixel of a texture) or, alternatively, normalized between 0 and 1, in such a way that one can ignore the dimension as well as the width and height of the texture itself. The meaning of the parameters u and v is established by the command `textureMode()` with parameter `IMAGE` or `NORMALIZED`.

Example 8.1
In the code that follows the image representing a broken glass [6] is employed as texture and followed by a color interpolation and the default illumination. The **shading** of the surfaces, produced by means of the illumination and the colors, is modulated in a multiplicative way by the colors of the texture.

```
size(400,400,P3D);
PImage a = loadImage("vetro.jpg");
lights();
textureMode(NORMALIZED);
beginShape(TRIANGLE_STRIP);
texture(a);
fill(240,   0, 0); vertex(40,61, 63, 0, 0);
fill(240, 150, 0); vertex(340, 80, 76, 0, 1);
fill(250, 250, 0); vertex(150, 176, 100, 1, 1);
                   vertex(110, 170, 180, 1, 0);
endShape();
```

8.3 Texture mapping

It is evident that the mapping operations from a texture image to an object surface, of arbitrary shape, implies some form of interpolation. Similarly to what happens for colors, only the vertices that delimit the surface are mapped onto exact points of the texture image. What happens for the internal points has to be established in some way. Actually, Processing and OpenGL behave according to what illustrated in

[5]http://www.computerarts.co.uk/in_depth/features/inside_the_incredibles
[6]See the file at <http://cnx.org/content/m13048/latest/.>

Section 8.1.1 (Bilinear Interpolation), i.e. by bilinear interpolation: a first linear interpolation over each boundary segment is cascaded by a linear interpolation on a scan line. If u and v exceed the limits of the texture image, the system (Processing) can assume that this is repeated periodically and fix it to the values at the border.

A problem that occurs is that a pixel on a display does not necessarly correspond exactly to a texel. One can map more than one texel on a pixel or, viceversa, a texel can be mapped on several pixels. The first case corresponds to a downsampling that, as seen in Sampling and Quantization[7], can produce aliasing. The effect of aliasing can be attenuated by means of low pass filtering of the texture image. The second case corresponds to upsampling, that in the frequency domain can be interpreted as increasing the distance between spectral images.

8.4 Texture Generation

Textures are not necessarely imported from images, but they can also be generated in an algorithmic fashion. This is particularly recommended when one wants to generate regular or pseudo-random patterns. For example, the pattern of a chess-board can be generated by means of the code

```
      PImage textureImg =
loadImage("vetro.jpg"); // dummy image colorMode(RGB,1);

int biro = 0;
int bbiro = 0;
int scacco = 5;
for (int i=0; i<textureImg.width; i+=scacco) {
   bbiro = (bbiro + 1)%2; biro = bbiro;
   for (int j=0; j<textureImg.height; j+=scacco) {
     for (int r=0; r<scacco; r++)
       for (int s=0; s<scacco; s++)
        textureImg.set(i+r,j+s, color(biro));
    biro = (biro + 1)%2;
  }
}
image(textureImg, 0, 0);
```

The use of the function random, combined with filters of various type, allows a wide flexibility in the production of textures. For example, the pattern represented in Figure 8.1 (Algorithmically-generated pattern) was obtained from a modification of the code generating the chess-board. In particular, we added the line `scacco=floor(2+random(5));` within the outer `for`, and applied an averaging filter.

[7]"Signal Processing in Processing: Campionamento e Quantizzazione" <http://cnx.org/content/m12751/latest/>

Algorithmically-generated pattern

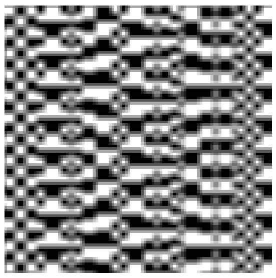


Figure 8.1

Exercise 8.1 ***(Solution on p. 95.)***
How could one modify the code Example 8.1 in order to make the breaks in the glass more evident?

Exercise 8.2
The excercise consists in modifying the code of the generator of the chess-board in Section 8.4 (Texture Generation) in order to generate the texture Figure 8.1 (Algorithmically-generated pattern).

Exercise 8.3
This exercise consists in running and analyzing the following code. Try then to vary the dimensions of the small squares and the filtering type.

```
size(200, 100, P3D);

PImage textureImg = loadImage("vetro.jpg"); // dummy image
colorMode(RGB,1);

int biro = 0;
int bbiro = 0;
int scacco = 5;
for (int i=0; i<textureImg.width; i+=scacco) {
   // scacco=floor(2+random(5));
   bbiro = (bbiro + 1)%2; biro = bbiro;
   for (int j=0; j<textureImg.height; j+=scacco) {
     for (int r=0; r<scacco; r++)
       for (int s=0; s<scacco; s++)
        textureImg.set(i+r,j+s, color(biro));
    biro = (biro + 1)%2;
  }
}
image(textureImg, 0, 0);
textureMode(NORMALIZED);
beginShape(QUADS);
  texture(textureImg);
  vertex(20, 20, 0, 0);
  vertex(80, 25, 0, 0.5);
  vertex(90, 90, 0.5, 0.5);
  vertex(20, 80, 0.5, 0);
endShape();

// ------ filtering -------

PImage tImg = loadImage("vetro.jpg"); // dummy image
float val = 1.0/9.0;
float[][] kernel = { {val, val, val},
                     {val, val, val},
                     {val, val, val} };
int n2 = 1;
int m2 = 1;
colorMode(RGB,255);
// Convolve the image
for(int y=0; y<textureImg.height; y++) {
  for(int x=0; x<textureImg.width/2; x++) {
    float sum = 0;
    for(int k=-n2; k<=n2; k++) {
      for(int j=-m2; j<=m2; j++) {
        // Reflect x-j to not exceed array boundary
        int xp = x-j;
        int yp = y-k;
        if (xp < 0) {
          xp = xp + textureImg.width;
        } else if (x-j >= textureImg.width) {
          xp = xp - textureImg.width;
        }
        // Reflect y-k to not exceed array boundary
        if (yp < 0) {
          yp = yp + textureImg.height;
        } else if (yp >= textureImg.height) {
          yp = yp - textureImg.height;
        }
        sum = sum + kernel[j+m2][k+n2] * red(textureImg.get(xp, yp));
      }
    }
    tImg.set(x,y, color(int(sum)));
  }
}

translate(100, 0);
beginShape(QUADS);
  texture(tImg);
  vertex(20, 20, 0, 0);
  vertex(80, 25, 0, 0.5);
  vertex(90, 90, 0.5, 0.5);
  vertex(20, 80, 0.5, 0);
endShape();
```

Solutions to Exercises in Chapter 8

Solution to Exercise 8.1 (p. 92)

It is sufficient to consider only a piece of the texture, with calls of the type `vertex(150, 176, 0.3, 0.3);`

Chapter 9

Signal Processing in Processing: Miscellanea[1]

9.1 Economic Color Representations

In Media Representation in Processing[2] we saw how one devotes 8 bits to each channel corresponding to a primary color. If we add to these the alpha channel, the total number of bits per pixel becomes 32. We do not always have the possibility to use such a big amount of memory for colors. Therefore, one has to adopt various strategies in order to reduce the number of bits per pixel.

9.1.1 Palette

A first solution comes from the observation that usually in an image, not all of the 2^{24} representable colors are present at the same time. Supposing that the number of colors necessary for an ordinary image is not greater than 256, one can think about memorizing the codes of the colors in a table (**palette**), whose elements are accessible by means of an index of only 8 bits. Thus, the image will require a memory space of 8 bits per pixel plus the space necessary for the palette. For examples and further explanations see color depth[3] in Wikipedia.

9.1.2 Dithering

Alternatively, in order to have a low number of bits per pixel, one can apply a digital processing technique called **dithering**. The idea is that of obtaining a mixture of colors in a perceptual way, exploiting the proximity of small points of different color. An exhaustive presentation of the phenomenon can be found at the voice dithering[4] of Wikipedia.

9.1.3 Floyd-Steinberg's Dithering

The Floyd-Steinberg's algorithm is one of the most popular techniques for the distribution of colored pixels in order to obtain a dithering effect. The idea is to minimize the visual artifacts by processing the error-diffusion. The algorithm can be resumed as follows:

- While proceeding top down and from left to right for each considered pixel,
- · calculate the difference between the goal color and the closest representable color (error)

[1]This content is available online at <http://cnx.org/content/m13085/1.5/>.
[2]"Rappresentazione di Media in Processing" <http://cnx.org/content/m12664/latest/>
[3]http://en.wikipedia.org/wiki/Color_depth
[4]http://en.wikipedia.org/wiki/Dithering

· spread the error on the contiguous pixels according to the mask $\frac{1}{16}\begin{pmatrix} 0 & 0 & 0 \\ 0 & X & 7 \\ 3 & 5 & 1 \end{pmatrix}$. That is, add $\frac{7}{16}$ of the error to the pixel on the right of the considered one, add $\frac{3}{16}$ of the error to the pixel bottom left with respect to the considered one, and so on.

By means of this algorithm it is possible to reproduce an image with different gray levels by means of a device able to define only white and black points. The mask of the Floyd-Steinberg's algorithm was chosen in a way that a uniform distribution of gray intensities $\frac{1}{2}$ produces a chessboard layout pattern.

Exercise 9.1 ***(Solution on p. 113.)***

By means of Processing, implement a program to process the file lena[5], a "dithered" black and white version of the famous Lena[6] image. The image, treated only in the left half, should result similar to that of Figure 9.1

[5]See the file at <http://cnx.org/content/m13085/latest/lena.jpg>
[6]http://en.wikipedia.org/wiki/Lenna

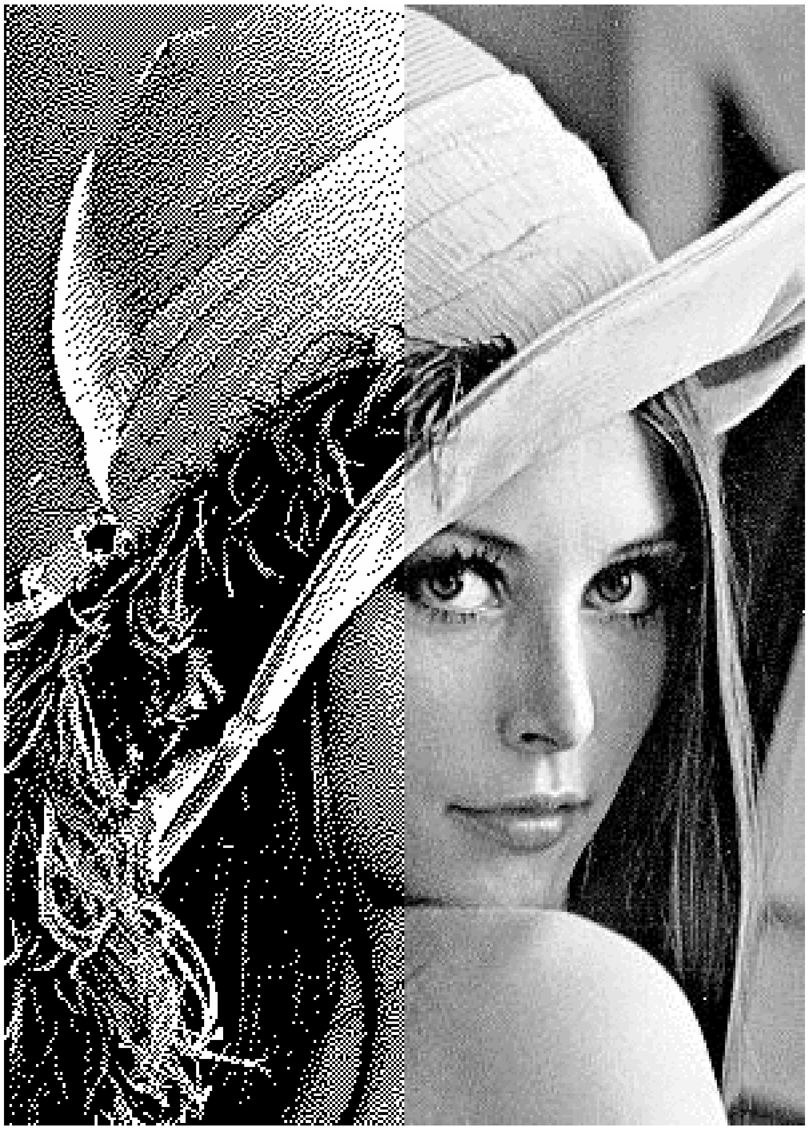


Figure 9.1

9.2 Economic Sound Representations

In case of audio signals, the use of dithering aims at reducing the perceptual effect of the error produced by the changes in the quantization resolution, that one typically performs when recording and processing audio signals. For example, when recoding music, more than 16-bits of quantization are usually employed. Furthermore, mathematical operations applied to the signal (as, for instance, simple dynamical variations) require an increasing of the bit depth that is of the number of bits. As soon as one reaches the final product, the audio CD, the number of quantization bits has to be reduced to 16. In each of these consecutives processes of re-quantization one introduces an error, that adds up. In case of a reduction of the number of bits, it is possible to truncate the values (that is the digits after the decimal point are neglected and set to zero) or round them (that is the decimal number is approximated to the closest integer). In both cases one introduces an error. In particular, when one considers signals with a well defined **pitch** (as in the case of musical signals), the error becomes periodic-like. From the example of the voice dithering[7] of Wikipedia, the reason of this additional pseudo-periodic noise (i.e. harmonic-like) becomes clear. This distortion corresponds to a buzz-like sound that "follows" the pitch of the quantized sound. The whole result is quite annoying from a perceptual point of view.

In case of audio signals, thus, dithering has the function of transforming this buzz-like sound in a background noise similar to a rustling, less annoying from a listening point of view. In Figure 9.2 an example of some periods of the waveform of a clarinet quantized with 16 bits is reported. The result of a reduction of the number of bits to 8 is represented in Figure 9.3. It is clearly visible how the reduction of the quantization levels produces series of lines with constant amplitude. The application of dithering generates a further transformation that, how one can see in Figure 9.4 "breaks" the constant lines by means of the introduction of white noise. The Figure 9.5, Figure 9.6 and Figure 9.7 represent the Fourier transforms of the sounds of Figure 9.2, Figure 9.3 and Figure 9.4, respectively. In the frequecy representation, it is also visible how the change to a quantization at 8 bits introduces improper harmonics (Figure 9.6) not present in the sound at 16 bit (Figure 9.5). These harmonics are canceled by the effect of dithering (Figure 9.7).

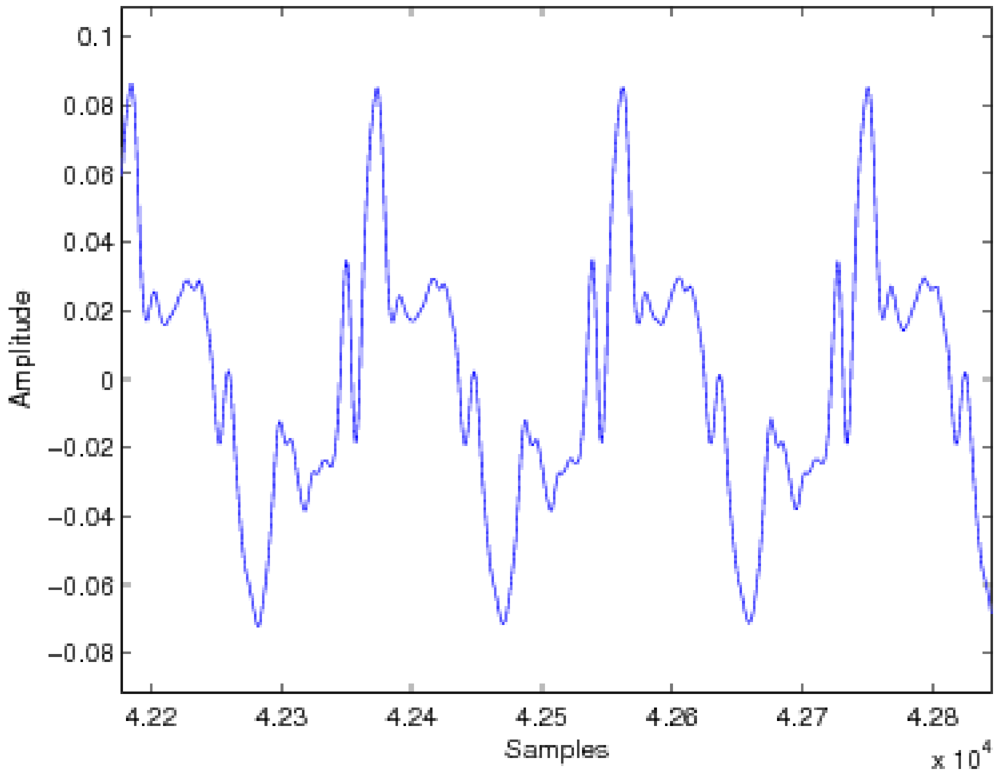


Figure 9.2

[7]http://en.wikipedia.org/wiki/Dithering

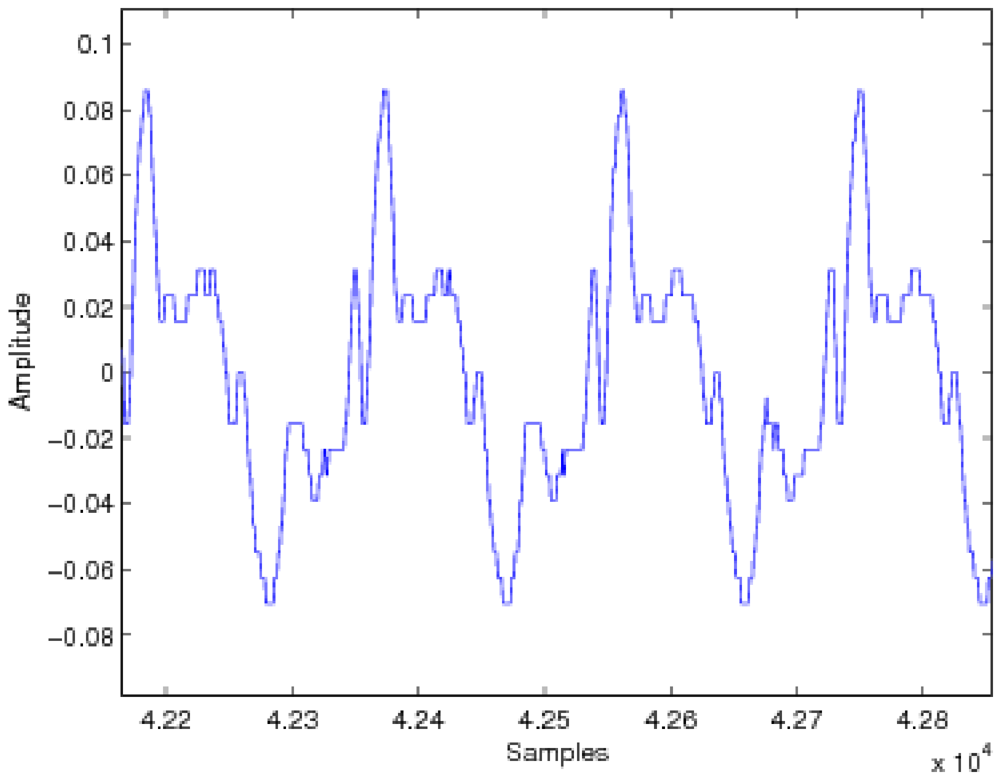

Amplitude
0.1
0.08
0.06
0.04
0.02
0
−0.02
−0.04
−0.06
−0.08
4.22
4.23
4.24
4.25
4.26
4.27
4.28
Samples
x 10^4

Figure 9.3

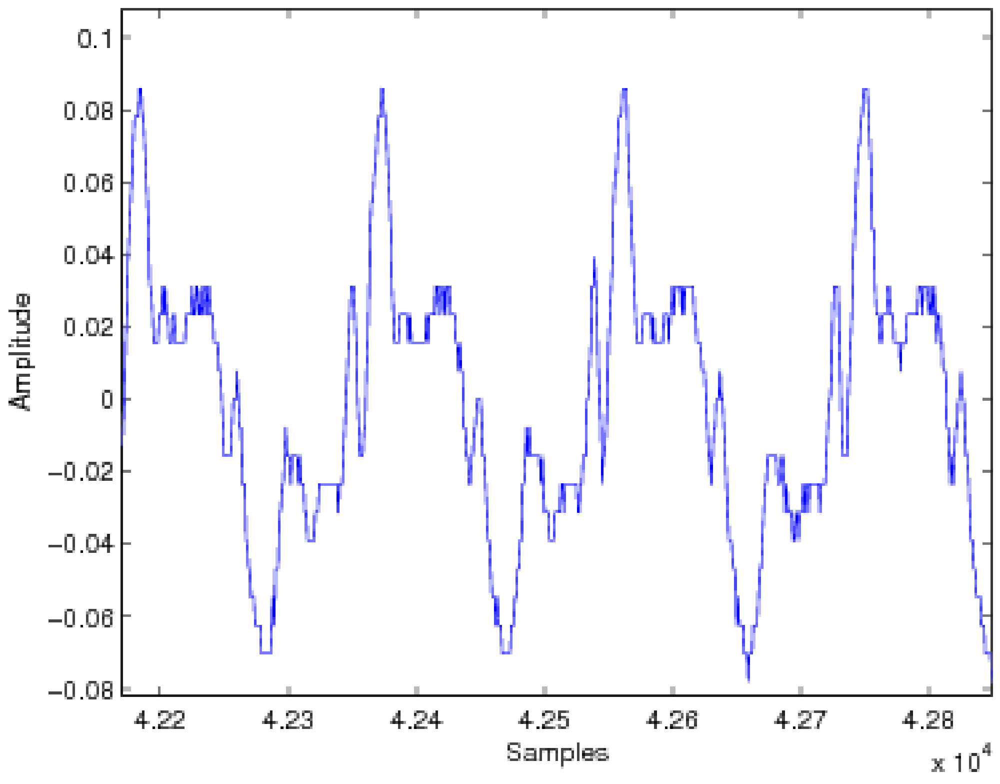

Amplitude
0.1
0.08
0.06
0.04
0.02
0
−0.02
−0.04
−0.06
−0.08
4.22
4.23
4.24
4.25
4.26
4.27
4.28
Samples
x 10^4

Figure 9.4

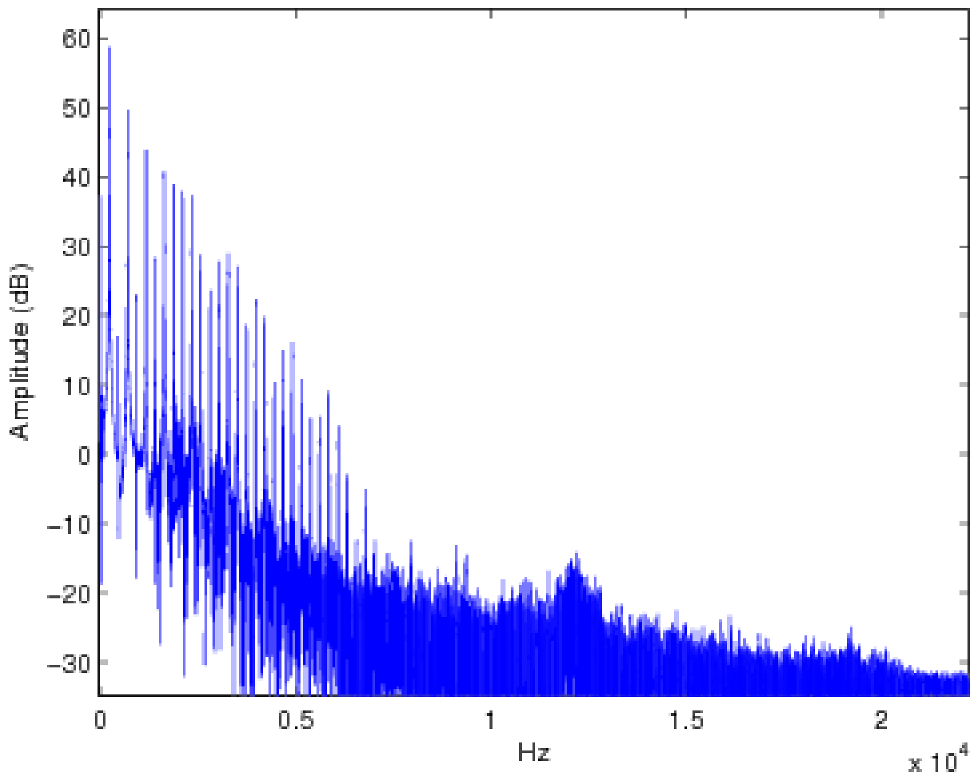


Figure 9.5

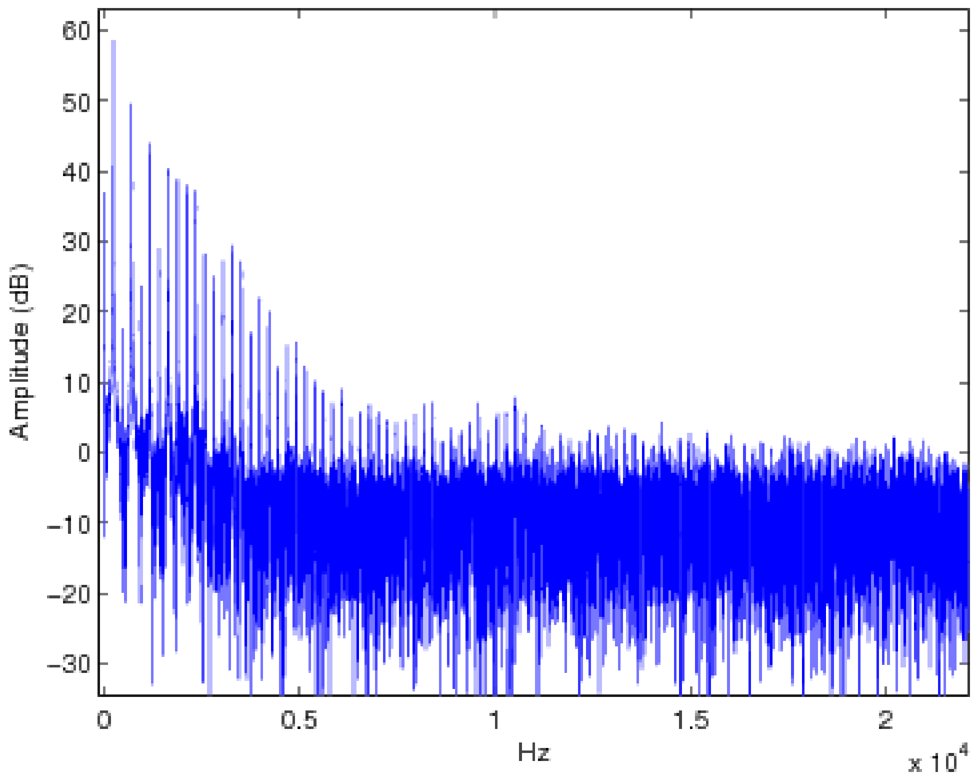


Figure 9.6

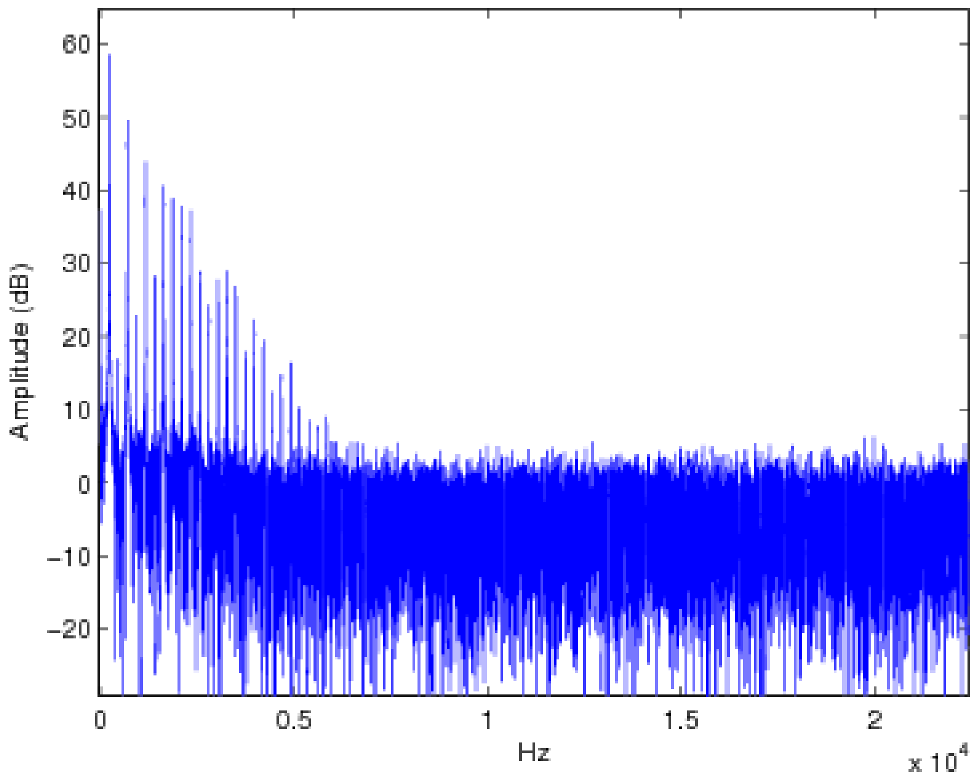


Figure 9.7

There exist also methods that exploit perceptual factors as the fact that our ear is more sensitive in the central region of the audio band and less sensitive in the higher region. This allows one to make the effect of a re-quantization less audible. This is the case of the **noise shaping** techniques. This method consists in "modeling" the quantization noise. Figure 9.8 represents the sound of a clarinet re-quantized with 8 bits. A dithering and a noise-shaping were applied to the sound. The result, apparently destructive from the point of view of the waveform, corresponds to a sound, whose spectrum is closer to that of the original sound at 16 bits, excepted for a considerable increasing of energy in the very high frequency region (Figure 9.9). This high frequency noise produces this "ruffled" waveform, i.e. high energy fast amplitude variations. This noise is anyway not audible. Thus, at a listening test, the final result is better than the previous one.

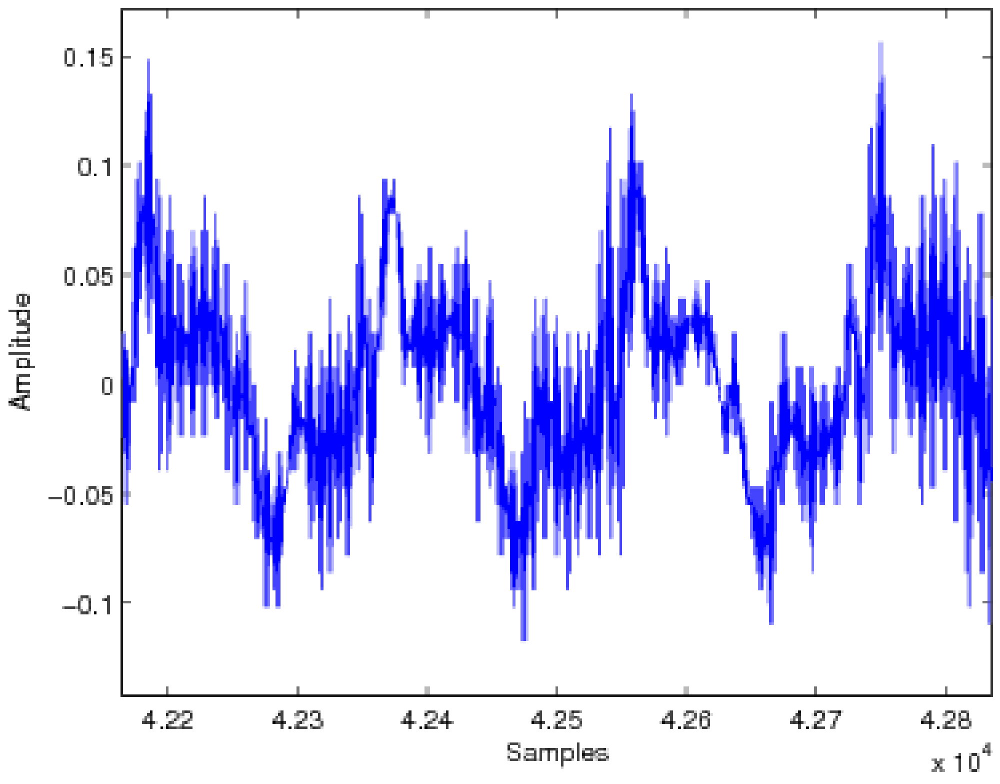

0.15
0.1
0.05
0
−0.05
−0.1
Amplitude
4.22
4.23
4.24
4.25
4.26
4.27
4.28
Samples
x 10^4

Figure 9.8

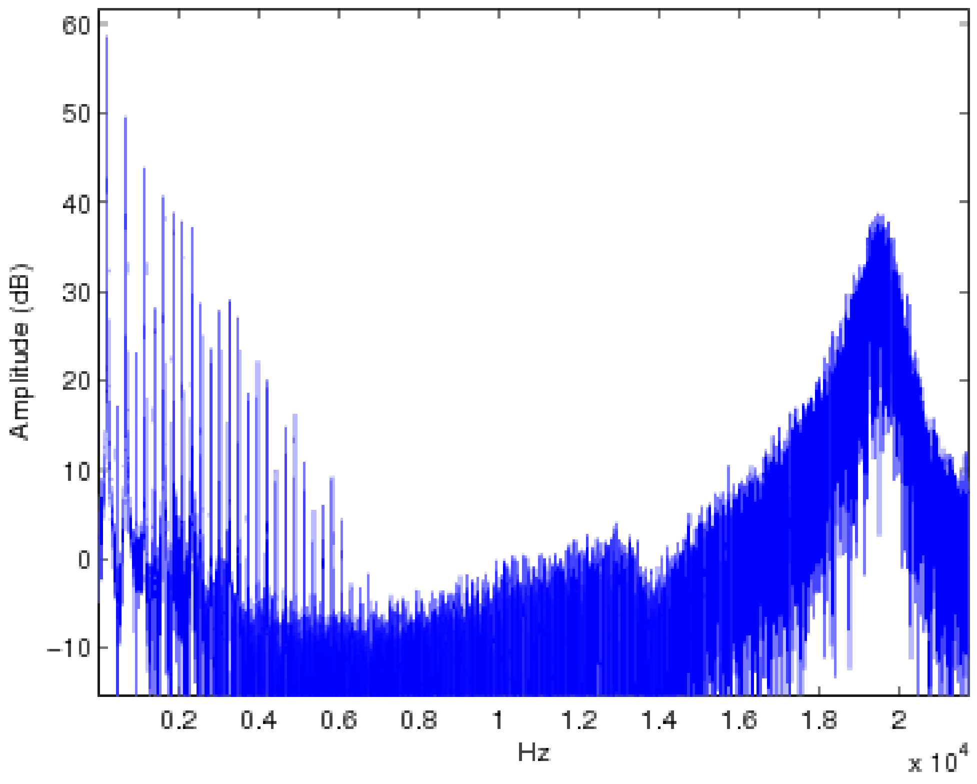


Figure 9.9

It is possible to think about the noise shaping as the audio counterpart of the Floyd-Steinberg's algorithm for graphics. In the audio case the error propagation occurs in the time-domain instead of the space-domain. The most simple version of noise shaping can be obtained by means of the definition of the quantization error

$$e(n) = y(n) - Q(y(n)) \tag{9.1}$$

where

$$y(n) = x(n) + e(n-1) \tag{9.2}$$

and x is the non-quantized signal. Further details about noise shaping can be found at Wikipedia, noise shaping[8] . What presented above can be tested in the sound examples clarinet[9] , clarinet at 8 bits[10], clarinet at 8 bits with dithering[11] and clarinet at 8 bit with noise shaping[12] that contain the clarinet sounds represented in Figure 9.2, Figure 9.3, Figure 9.4, e Figure 9.8, respectively.

[8]http://en.wikipedia.org/wiki/Noise_shaping
[9]See the file at <http://cnx.org/content/m13085/latest/./clarinetto.wav>
[10]See the file at <http://cnx.org/content/m13085/latest/./clarinetto8.wav>
[11]See the file at <http://cnx.org/content/m13085/latest/./clarinetto8dith.wav>
[12]See the file at <http://cnx.org/content/m13085/latest/./clarinetto8dithNs.wav>

9.3 Histogram-Based Processing.

Definition 9.1: Image Histograms
Graphic representation by means of vertical bars, where each bar represents the number of pixels present in the image for a given intensity of the gray scale (or color channel). Wikipedia definition.[13]

Among the examples in Processing[14] , one finds the code Histogram[15] that overlap an image to its own histogram.

The histogram offers a synthetic representation of images, in which one looses the information concerning the pixel positions and considers only the chromatic aspects. This provides information about the **Tonal Gamma** of an image (gray intensity that are present) and about the **Dynamics** (extension of the Tonal Gamma). The image of a chess board, for example, has a Tonal Gamma that includes only two gray levels (black and white) but it has a maximal dynamics (since white and black are the two extremity of the representable gray levels).

The histogram is the starting point for various processing effects aiming at balancing or altering the chromatic contents of an image. In general, the question is building a map $g_o = f(g_i)$ for the gray levels (or color-channel levels) that can be applied to each pixel. The histogram can drive the definition of this map.

9.3.1 Translation and Expansion of an Histogram

If the map is of the kind $g_o = g_i + k$ the histogram is translated in the sense of a higher or lower brightness, according to the sign of k. On the other side, if the map is of the kind $g_o = kg_i$ the histogram will be expanded or compressed, for values of k smaller or greater than 1, respectively.

The **contrast stretching** is one of the operations of this kind of linear scaling that tries to extend the dynamic range of an image. The interval by means of which one performs the scaling is set on the basis of the histogram, neglecting, for example, the tails of the distribution corresponding to 10% of the darkest and brightest pixels.

9.3.2 Non Linear Scaling

More in general, the map $g_o = f(g_i)$ can be non linear, and this allows a greater flexibility in the manipulation of the histogram. A useful instrument is the one that allows to manipulate interactively the scaling map and to see the results on the image and/or on the histogram in real time. The instrument `Color Tools/Curves` of the image processing software Gimp[16] does this, using an interpolating spline. In Processing it is possible to build a similar instrument, as reported in Example 9.1.

Example 9.1
Applet that allows to apply a non linear scaling to the gray levels and to analyze the effect by means of an histogram. [17]

9.3.3 Equalization of an Histogram

The non linear scaling is the tool to **equalize** the histogram, that is to shape it in a desirable way. An image has a balanced tonal gamma, if all of the gray levels are represented and if the distribution is approximately uniform. In other words, one aims at a flat histogram. Without entering too much into the mathematical details, one can say that the non linear map to be used for the equalization is obtained from the **cumulated**

[13]http://en.wikipedia.org/wiki/Color_histogram
[14]http://www.processing.org/learning/topics/
[15]http://www.processing.org/learning/topics/histogram.html
[16]http://www.gimp.org
[17]See the file at <http://cnx.org/content/m13085/latest/histogram_t.html>

distribution of the histogram of the image $f(g_i) = \sum_{k=0}^{g_i} h(k)$, where $h(k)$ is the frequency, properly scaled by means of a normalization constant, with which the k-th gray level appears .

Exercise 9.2 ***(Solution on p. 114.)***

Modify the Processing code of Example 9.1 in order to add the operation of equalization of the histogram.

9.4 Segmentation and Contour Extraction

9.4.1 Contours

The objects that populate a scene represented in an image are usually detectable by means of their contours: Thread-like profiles that correspond to a fast variation of color or of gray intensity. The contour extraction is one of the typical operation of the image processing. From the point of view of the Mathematical Analysis, a fast variation of a function corresponds to a high value of the **derivative** function. In the frame of Digital Signal Processing (DSP), the derivative can be approximated by means of a difference operation, i.e. by means of a filter. The filters that let fast variations through and eliminate slow variations are of a high-pass type. It is not surprising, thus, that for contour extraction, one uses convolution masks similar to that seen in Elementary Filters[18] employed for edge crispening. More in detail, one can say that in 2D one looks for points with maximum amplitude of the **gradient**, i.e. points, where the **Laplacian** of the image that is its second spatial derivative is necessarily zero. The Laplacian can be approximated (in the discrete space) by means of a convolution mask $\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. The direct application of the Laplacian is often not satisfying, since the result is too sensitive to noise and to small details. It is, thus, useful to combine the Laplacian mask with that of a low-pass filter. The combination of a Laplacian and a Gaussian (**LoG**) filter produces, in the case 5 by5, the mask $\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}$ In the software Gimp, a contour enhancer is available, based on the difference between two Gaussian filters, corresponding to two Gaussian curves with different amplitude. This produces an inhibition effect outside the principal contours, in a way similar to what happens in our perceptual system.

9.4.2 Regions

In many applications it is necessary to isolate the various objects that populate a scene, starting from their representation, as pixel collections of an image. For example, it could be interesting to isolate an object in **foreground** from the **background**. In this case, one talks about segmentation or extraction of regions. The most simple way for isolating different regions is that of doing it on a color basis, or on a gray-intensity basis. Also in this case, the operation can be driven by the histogram that can be helpful in order to establish a gray **threshold**. All the darker pixels will be mapped to black, while all the lighter ones will be mapped to white. For example, if the histogram presents two evident maxima, it is possible to assign the region of one maximum to the foreground (since it is, for example, lighter) and the region corresponding to the other maximum to the background (since it is, for example, darker). The threshold will be chosen in between the two maxima. Sometimes it is necessary to establish a multiplicity of thresholds, in order to isolate different

[18]"Signal Processing in Processing: Filtri Elementari" <http://cnx.org/content/m12827/latest/#crispp>

regions by means of different gray levels. For color images, the thresholds can be different for different RGB channels.

Exercise 9.3

Apply the Laplacian filter and the LoG filter to the image of Lena.

Exercise 9.4 ***(Solution on p. 115.)***

Show that the extraction of a background figure by means of a threshold can be implemented by means of non linear map $g_o = f(g_i)$. What should be the form of this map?

Exercise 9.5

Employ a program for image processing (ex. Gimp[19]) in order to isolate (by means of thresholding) the breaks in the image of the broken glass[20].

9.5 Audio Dynamic Compression

For audio signals, similarly to what seen for images, one considers the problem of reducing the data necessary to represent a sound, while preserving an acceptable quality of the signal from a perceptual point of view. It is not easy to define, what is meant by "acceptable quality", when one perform a data reduction or, better, a data compression. In general the qualitative evaluation parameters of the audio compression standards are statistical, based on the results of listening tests made on groups of listeners, representing a wide gamma of users. The audio compression standards are usually founded on the optimization of the dynamics of the signal, that is on the optimization of the number of bits employed for the quantization. A well known example of compression standard is that of mp3, in which one exploits psychoacoustic phenomena, as the fact that louder sounds mask (make inaudible) softer sounds. In the reproduction of digitalized sounds, the thing that one wants to mask is the quantization noise. In other words, if the sound has a wide dynamics (it is loud) one can adopt a greater quantization step, since the louder quantization noise produced by the rougher subdivision of the quantization levels is anyway masked by the reproduced sound. Still simplifying things in a drastic way, one could say that mp3 varies the step of quantization according to the dynamics of the sound and in a different way in different bands of frequency. In other words, the signal is divided into many frequency bands (in a similar way as the Equalizer of a Hi-fi system does) and each band is quantized separately. This allows a reduction of even 20 times of the number of bits with respect to a fixed 16 bit dynamics. Another compression technique is provided by the mu-law (μ-law). This standard is used mainly in audio systems for digital communication in North America and Japan. In this case, the main idea is to modify the dynamic range of the analogical audio signal before the quantization. Behind these compression techniques, there is once more a psychoacoustic phenomenon that is the fact that our perception of the intensity is not linear but logarithmic-like. In other words, our perception behaves approximately according to what shown in Figure 9.10.

[19]http://www.gimp.org

[20]See the file at <http://cnx.org/content/m12837/latest/vetro.jpg>

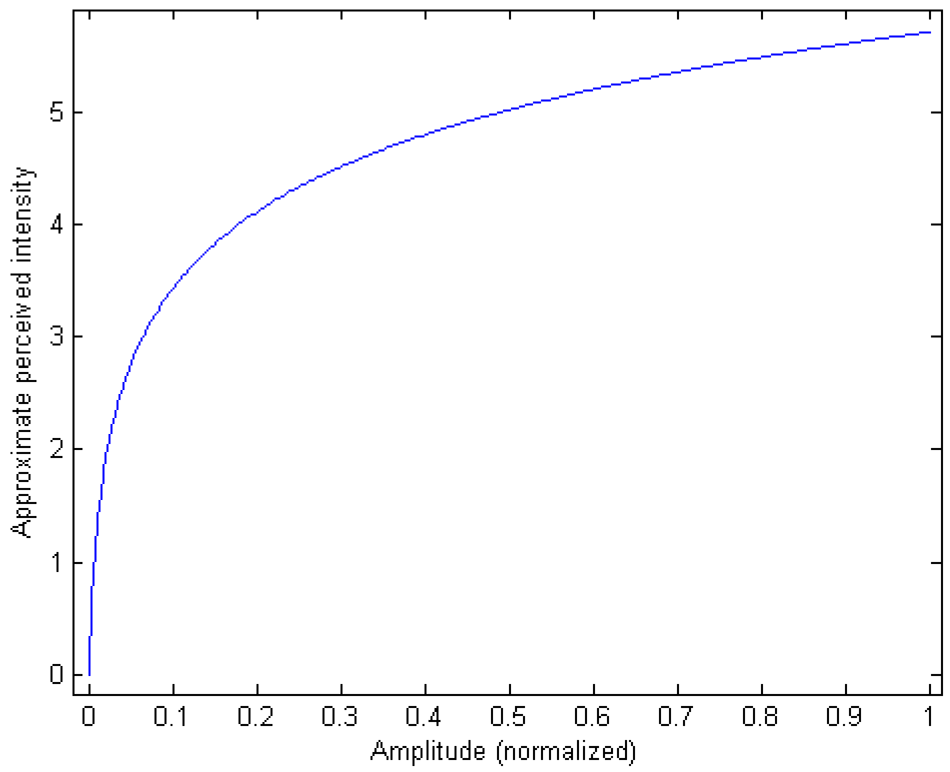


Figure 9.10

What the mu-law actually performs is a reduction of the dynamic range of the signal by means of an amplitude re-scaling according to the map described in Figure 9.11. It is visible how the effect is that of amplifying the small amplitudes, reducing the range of amplitude values of the signal (in the sense of big amplitudes) and, as a consequence, increasing the relationship (the amplitude difference) between the sound and the quantization noise. Afterwards, a linear quantization of the non linearly distorted signal is performed. As one wants to play back the digital signal, this is first converted into an analogical signal and then transformed by means of a curve performing an inverse amplitude distortion with respect to that of Figure 9.11. The global result is equivalent to a non linear quantization of sound, where the quantization step is bigger (rougher) for bigger amplitudes and smaller (more detailed) for smaller amplitudes. This corresponds, at least from a qualitative point of view, to the way of functioning of our perceptual system. We are more sensitive to the intensity differences in case of soft sounds and less sensitive in case of loud and very loud sounds. The A-law, adopted in the digital systems in Europe, is very similar to the mu-law.

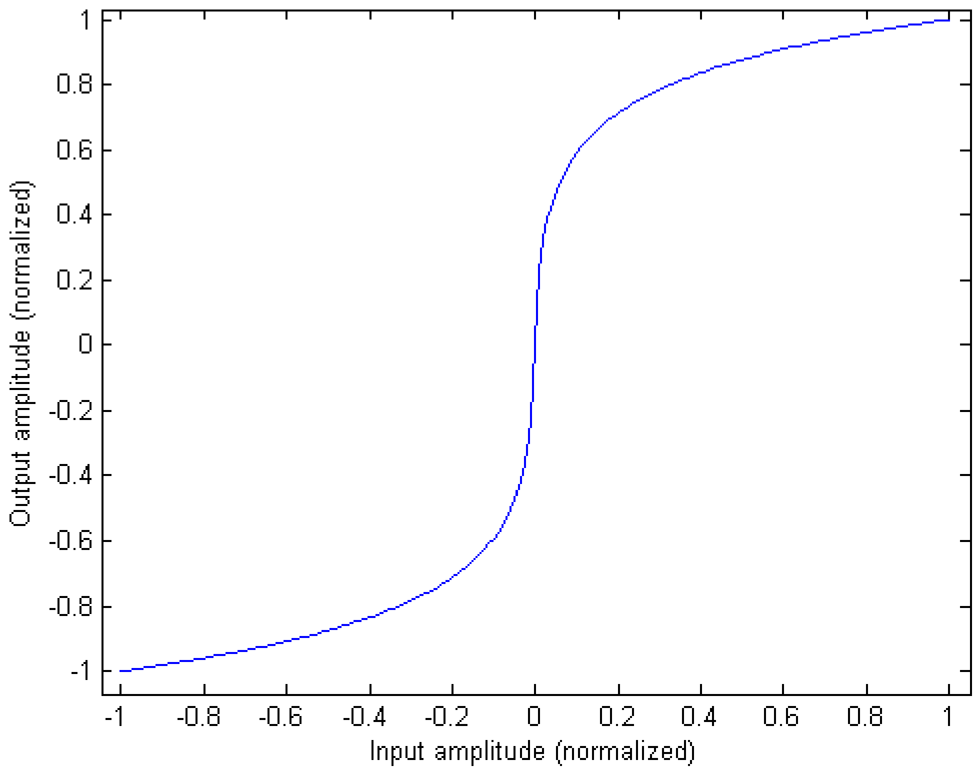

Output amplitude (normalized)
Input amplitude (normalized)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

Figure 9.11

Solutions to Exercises in Chapter 9

Solution to Exercise 9.1 (p. 98)

```
size(300, 420);
PImage a;  // Declare variable "a" of type PImage
a = loadImage("lena.jpg"); // Load the images into the program
image(a, 0, 0); // Displays the image from point (0,0)

int[][] output = new int[width][height];
for (int i=0; i<width; i++)
  for (int j=0; j<height; j++) {
    output[i][j] = (int)red(get(i, j));
    }
int grayVal;
float errore;
float val = 1.0/16.0;
float[][] kernel = { {0, 0, 0},
                     {0, -1, 7*val},
                     {3*val, 5*val, val }};

for(int y=0; y<height; y++) {
  for(int x=0; x<width; x++) {
    grayVal = output[x][y];// (int)red(get(x, y));
    if (grayVal<128) errore=grayVal;
      else errore=grayVal-256;
    for(int k=-1; k<=1; k++) {
      for(int j=-1; j<=0 /*1*/; j++) {
        // Reflect x-j to not exceed array boundary
        int xp = x-j;
        int yp = y-k;
        if (xp < 0) {
          xp = xp + width;
        } else if (x-j >= width) {
          xp = xp - width;
        }
        // Reflect y-k to not exceed array boundary
        if (yp < 0) {
          yp = yp + height;
        } else if (yp >= height) {
          yp = yp - height;
      }
      output[xp][yp] = (int)(output[xp][yp] + errore * kernel[-j+1][-k+1]);
      }
    }
  }
}

for(int i=0; i<height; i++)
```

```
  for(int j=0; j<width; j++)
    if (output[j][i] < 128) output[j][i] = 0;
    else output[j][i] = 255;

// Display the result of dithering on half image
loadPixels();
for(int i=0; i<height; i++) {
  for(int j=0; j<width/2; j++) {
    pixels[i*width + j] =
      color(output[j][i], output[j][i], output[j][i]);
  }
}
updatePixels();
```

Solution to Exercise 9.2 (p. 109)

```
int grayValues = 256;
int[] hist = new int[grayValues];
int[] histc = new int[grayValues];
PImage a;

void setup() {
  background(255);
  stroke(0,0,0);
  size(300, 420);
  colorMode(RGB, width);
  framerate(5);
  a = loadImage("lena.jpg");
  image(a, 0, 0);
}

void draw() {
    // calculate the histogram
  for (int i=0; i<width; i++) {
    for (int j=0; j<height; j++) {
      int c = constrain(int(red(get(i,j))), 0, grayValues-1);
      hist[c]++;
    }
  }

  // Find the largest value in the histogram
  float maxval = 0;
  for (int i=0; i<grayValues; i++) {
    if(hist[i] > maxval) {
      maxval = hist[i];
    }
  }

  // Accumulate the histogram
  histc[0] = hist[0];
  for (int i=1; i<grayValues; i++) {
    histc[i] = histc[i-1] + hist[i];
  }

  // Normalize the histogram to values between 0 and "height"
  for (int i=0; i<grayValues; i++) {
    hist[i] = int(hist[i]/maxval * height);
  }

  if (mousePressed == true) { //equalization
    for (int i=1; i<grayValues; i++) {
    println(float(histc[i])/histc[grayValues-1]*256);
    }
    loadPixels();
    println("click");
    for (int i=0; i<width; i++)
      for (int j=0; j<height; j++) {
        //normalized cumulated histogram mapping
        pixels[i+j*width] = color(
          int(
          float(histc[constrain(int(red(a.get(i,j))), 0, grayValues-1)])/
            histc[grayValues-1]*256));
      }
    updatePixels();
     }

  // Draw half of the histogram
  stroke(50, 250, 0);
  strokeWeight(2);
  for (int i=0; i<grayValues; i++) {
    line(i, height, i, height-hist[i]);
  }
}
```

Solution to Exercise 9.4 (p. 110)

It is a step map, with transition from 0 to 255 set in correspondence of the chosen threshold.

Glossary

C **Convolution of two 2D signals (images)**

$y(m,n) = \mathrm{h} \ast \mathrm{x}(m,n) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} x(k,l)\, h(m-k, n-l)$

Convolution of two signals h and x

$y(n) = \mathrm{h} \ast \mathrm{x}(n) = \sum_{m=-\infty}^{\infty} x(m)\, h(n-m)$

G **global scope**

defined outside the methods setup() and draw(), the variable is visible and usable anywhere in the program

H **homogeneous coordinates**

quadruples of numbers, where the first triple is to be read in the X-Y-Z space, while the fourth number indicates a **vector** if it takes value 0, or a **point** if it takes value 1.

I **Image Histograms**

Graphic representation by means of vertical bars, where each bar represents the number of pixels present in the image for a given intensity of the gray scale (or color channel). Wikipedia definition.[21]

L **local scope**

defined within a code block or a function, the variable takes values that are local to the block or function, and any values taken by a global variable having the same name are ignored.

S **scope**

within a program, it is a region where a variable can be accessed and its value modified

Spline

Piecewise-polynomial curve, with polynomials connected with continuity at the **knots**

NOTE: See Introduction to Splines[22] and, for an introduction to the specific kind of splines (Catmull-Rom) used in Processing, the term **spline** in Wikipedia.

Superposition principle

If y_1 and y_2 are the responses to the input sequences x_1 and x_2 then the input $a_1x_1 + a_2x_2$ produces the response $a_1y_1 + a_2y_2$

T **The impulse in discrete time (space)**

is the signal δ with value1 at the instant zero (in the point with coordinates $[0,0]$), and 0 in any other instant (point).

Time invariance

A system is time-invariant if a time shift of D samples in the input results in the same time shift in the output, i.e., $x(n-D)$ produces $y(n-D)$.

[21] http://en.wikipedia.org/wiki/Color_histogram
[22] http://cnx.org/content/m12986/latest/

Bibliography

[1] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach With OPENGL primer package-2nd Edition*. Prentice-Hall, Inc., 2001.

[2] Davide Rocchesso. *Introduction to Sound Processing*. Mondo Estremo, 2003. http://www.mondo-estremo.com.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Media Processing in Processing
A course on fundamentals of image/sound processing and graphic programming explained by means of the free language and environment Processing.