Processing

A Beginners Guide To Coding (BETA 2)

Author *Nikolaus Gradwohl*
Web www.local-guru.net

Table of Contents

1. Introduction 3
2. Quick Start - The First Program 4
    2.1. The first Sketch 4
    2.2. Exporting as Applet 7
    2.3. Exporting as Application 7
3. When things go wrong 9
    3.1. The "I have stolen one character" - monster 9
    3.2. The "I removed your data" - monster 10
    3.3. The "I do what you say" - monster 11
4. Just For Fun 13
    4.1. Shapes, colors and coordinates 13
    4.2. Please Follow the Mouse 15
    4.3. Make the ball bounce around 16
    4.4. Some more Crayons 20
5. Rotate, Scale, Translate 24
    5.1. Green Microplanet 24
    5.2. Rotating Image 28
6. A Simple Game 31
7. Another Dimension 40
8. Data Visualization 42
9. Interaction 46
    9.1. Mouse 46
    9.2. Keyboard 49
    9.3. Live Video 52
    9.4. Hello Computer? 56
    9.5. Midi-Events 61
    9.6. Joystics, Gamepads, etc 63
10. Installation 67
    10.1. Installation on Macos X 67
    10.2. Installation on Windows 67
    10.3. Installation on Linux 67

1. Introduction

TBD

2. Quick Start - The First Program

This chapter shows you how to install Processing on your computer and guide you through your first Processing-Sketch. Sketch is what programs are called in Processing. I find this a lot less scary than "program". If you ask someone to make a beautiful image, they will most certainly tell you that they can't draw - but if you ask them to do a quick sketch most will understand and feel confident in their ability. So don't think of writing programs (which sounds complicated and scary), think of writing small sketches which is easy and fun to do.

To install Processing, go to http://www.processing.org and download a version for your operating system. Processing supports Windows, Linux and MacOS-X. For more detailed Install instructions see appendix ...

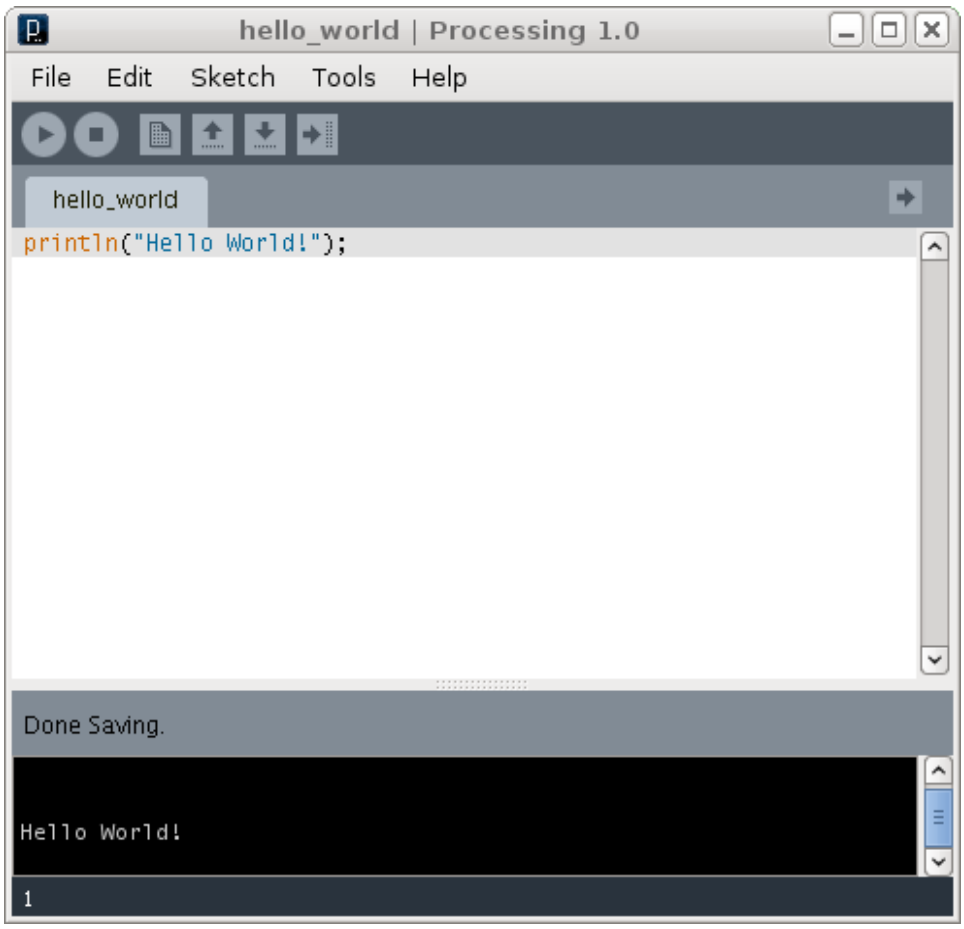
2.1. The first Sketch

Ok now that Processing is installed on your computer it's time to write your first sketch. Since the late 60's programming introductions start by writing a program that prints "Hello World" on the screen. Thats pretty boring in Processing since it consists only of one line. But because its a tradition let's write one anyway.

Open Processing and write the following code in the editor area.

```
println("Hello World!");
```

Now click on the little play button to run your Hello world program. In the bottom section of Processing console the text "hello world" should appear. Congratulations - you wrote your first sketch and ran it successfully. The sketch we wrote has not yet been saved. So let's do this now. Click on the icon with the little arrow that's pointing down and give your sketch a name (hello_world would be a nice idea :-)). If you click the save button the next time you don't have to enter the name again, since Processing already knows how you want to call your sketch.



Now that you have fullfilled your traditional duties as a programming novice, we can start a new sketch and write our first real sketch, featuring a friendly smiling face. Click on the "new Sketch" icon or select "New" from the "File"-menu and start write the following code into the editor area.

```
void setup() {
}
```

```
void draw() {
}
```

This is the basic skeleton that every sketch we will write starts with. void setup() is a method. It tells Processing what to executed at the beginning of the program. void draw() will be called several times a second and handle the - you sure guessed it already - drawing of the window content.

If you click the play button now a small gray window will open and do exactly what we specified in the draw method - nothing. Let's explain to our computer what we want it to do when we click on the play button. First we change our setup method to look like this

```
void setup() {
    size( 300, 300 );
}
```

When we click the play button this time the gray window is much bigger. Now we change the draw method and tell the computer to set the background to white with the command background(255). Then change the fill color to yellow by using the command fill(255,255,0). And finally draw a circle by using ellipse(150,150,200,200), which draws an ellipse at the coordinates 150,150 with the width 150 and the height 200.

```
void draw() {
    background( 255 );
    fill( 255, 255, 0 );
    ellipse( 150, 150, 200, 200 );
}
```

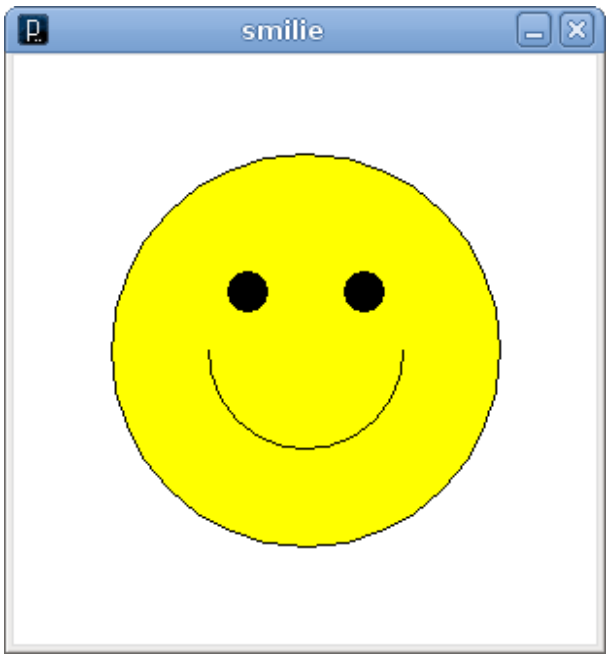
Now we add two small black circles for the eyes. Add the following three lines on the bottom of the draw methode (just before the closing curly brace)

```
  fill(0);
  ellipse( 120, 120, 20, 20 );
  ellipse( 180, 120, 20, 20 );
```

For the mouth we want a half circle but we don't want it to be filled. So, we use the noFill() command to tell Processing to skip the filling and the arc command for drawing the lower half of a circle. Add the following two lines at the bottom of the draw method

```
  noFill();
  arc( 150, 150, 100, 100, 0, PI );
```

Now click the play button and welcome your first smiling sketch. Don't forget to save it by clicking on the save icon and giving your sketch a name. Congratulations! You made your first real sketch in Processing.



2.2. Exporting as Applet

What good is your first sketch if you have to carry around your computer to show it to anyone? Processing allows you to export your work and deploy it on a webserver. If you click on the export icon or select export from the file menu Processing makes a folder called "applet" and generates all the code for you that is needed to put your sketch on a website.

If you like to add some text to the website (copyright notices, links to your blog, explanations how to use the sketch, ...) just add them as a comment at the start of your sketch like in the following code block.

```
/**
This is a comment that will be shown in a html page of an applet-export
*/
```

Putting /** on one line will start a commentblock, */ will end it again. you can also use html code fragments in this comment block to add links or make some section bold or italic.

If you don't have a webserver you still can share your sketches with the Processing community by posting it to openprocessing.org. This site is for Processing sketches what youtube is for videos and flickr to photos. The site is also a great source for inspiration and learning, because all published sketches also have the sourcecode available.

2.3. Exporting as Application

Applets are fine to show your sketches to a broad audience, but sometimes they simply don't work. If you don't have a website or your sketch uses code that is not allowed in an applet for security reasons, don't worry - there is an alternative. You can export your sketch as an Application for Linux, MacOS X or Windows by selecting "Export Application" from the File menu and selecting which plattforms you want to support.

Processing then creates a folder for every platform and generates an exe file for Windows, an application package for MacOS X, and a startup shellscript for Linux. This folder can be copied around, and used on another computer, as long as java is installed on the target machine.

3. When things go wrong

At first I wanted to name the chapter "If things go wrong" but that didn't feel right, because it is not a question of IF but only a question of WHEN things go wrong. They will for sure, when you least expect it, and you will have no clue what to do.

I hope I didn't scare you too much, but this is so. In this section I will show you some of the most common failures that can (and mostly will) occur when writing computer a program. I hope to give you some starting points to look for when something doesn't work the way you expect. But, be prepared for the unexpected since errors are very creative little monsters that try to catch you wherever they can.

In this chapter I will use the following little sketch to demonstrate what it looks like when the bugs bite you and what you can do to get rid of them.

```
void setup() {
  size(300,300);
}

void draw() {
  background(0);
  ellipse(150,150,200,200);
}
```

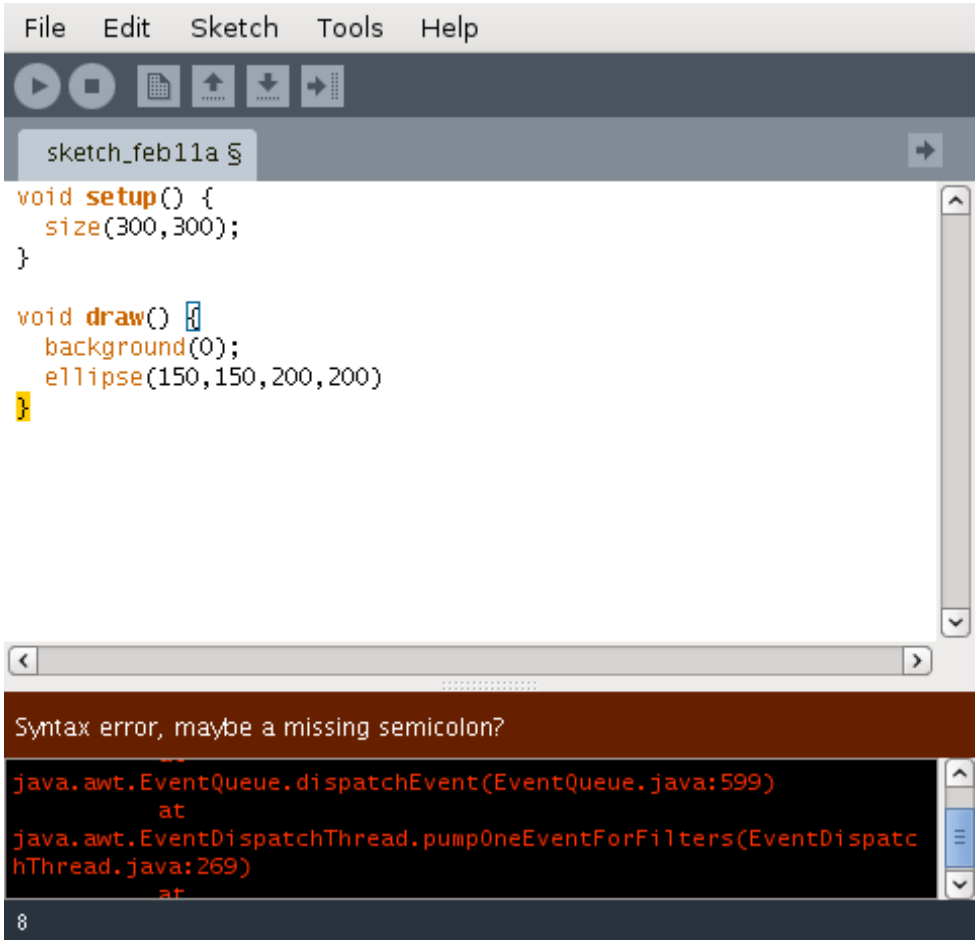
3.1. The "I have stolen one character" - monster

This little guy is most annoying and it tries to bite you very often. It tries to remove characters as you type your code. It really loves to remove semicolons or closing brackets.

Fortunately, this monster is caught by Processing most times. If, for example, the "I have stolen one character" - monster steals a semicolon in the draw method like this

```
void setup() {
  size(300,300);
}

void draw() {
  background(0);
  ellipse(150,150,200,200)
}
```

When you tell Processing to run your sketch something like this happens



This is the way Processing tries to catch the monster and give you a clue where it might have caused the monster's damage. When the missing character is a semicolon it doesn't show the line where it is missing, but the next line.

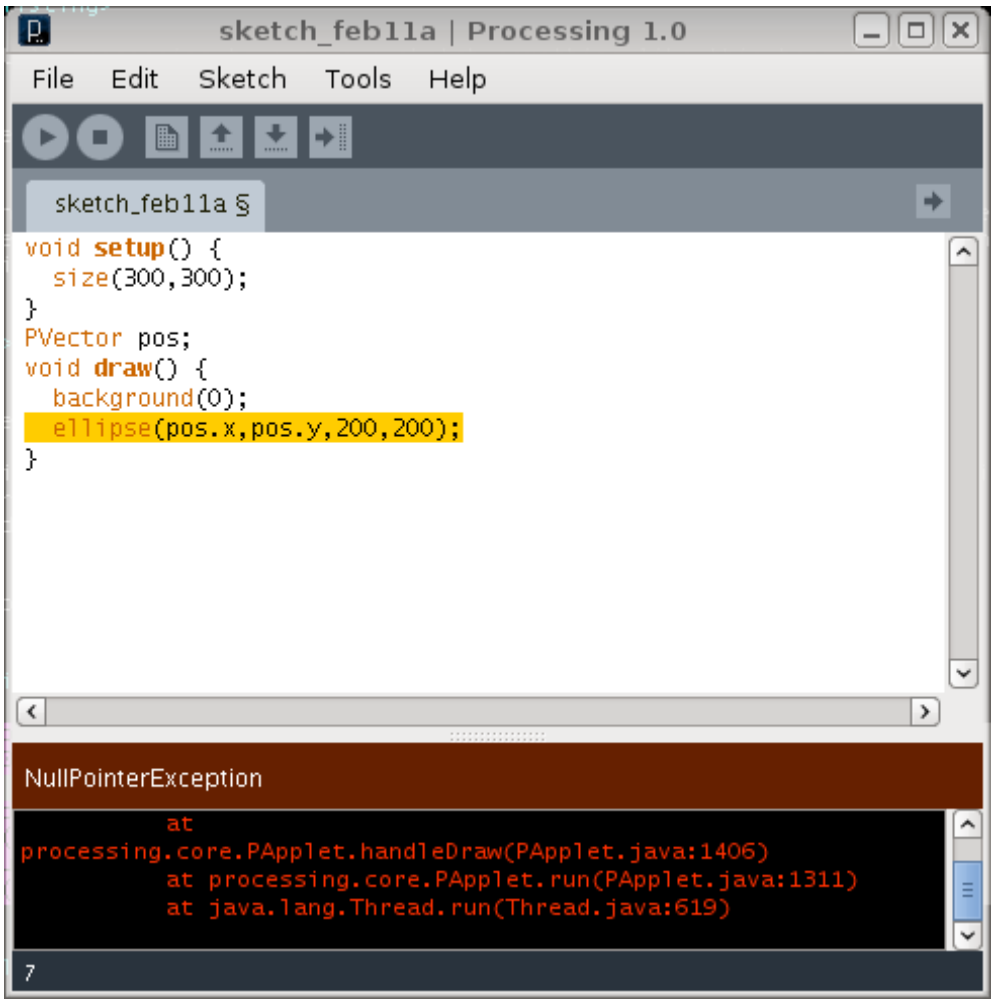
3.2. The "I removed your data" - monster

This one is a very tricky and dangerous one, because you don't see the effect it causes at compile time. It removes some data you try to access (it had to be removed by someone, since if you where not sure the data will be there, why would you have tried to access it?).

To show you how it looks like when this monster strikes we will have to modify the sketch a little bit.

```
void setup() {
  size(300,300);
}
PVector pos;
void draw() {
  background(0);
  ellipse(pos.x,pos.y,200,200);
}
```

When you hit the play button this time, the program starts without any problems, but it stops very soon showing an error like this one



What Processing is trying to say is that pos has no value. Nobody told it where to draw the circle. It's relatively easy to spot in this simple example, but can be really tricky if you load some data form external sources (urls, files, ...) that do not exist when you run your sketch. Or get removed after a while. So if you are loading data from anywhere, make sure it still is where your sketch expects it to be.

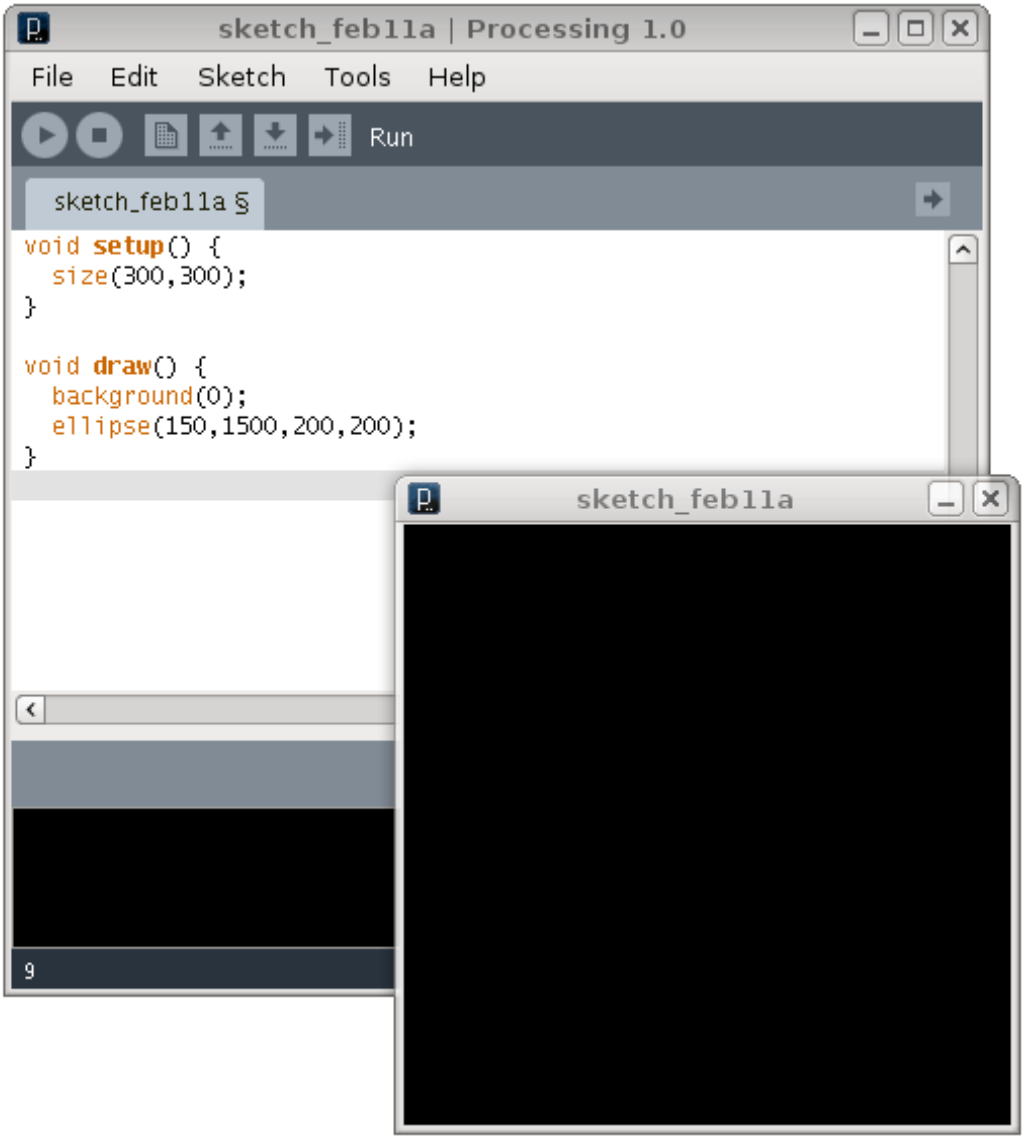
3.3. The "I do what you say" - monster

This is the most annoying monster of all. No error appears at compile time, no error appears at runtime, but the program simply doesn't do what you want. It does what you told it to do.

This little harmless looking sketch for example starts and runs without any problems when you hit the play button.

```
void setup() {
  size(300,300);
}

void draw() {
  background(0);
  ellipse(150,1500,200,200);
}
```

But, there is no white circle displayed!. Why doesn't the circle get drawn? After some hours of searching (and depending on your temperament - shouting and throwing things, etc) you find out that Processing has drawn a circle. It drew it exactly where you told it at position 150 and _1500_ and not at 150,150 where you wanted it to be.



To detect such errors it sometimes helps to get away from the screen for a few minutes and then try to search it again. Another tip is to show the code to a colleague or friend, it doesn't even have to be a programmer, just explain your program to someone - step by step. This helps surprisingly often to find the "I do what you say" - monster, because you are forced to look at the spots where you normally don't search.

4. Just For Fun

In this chapter I will explain some easy sketches step my step. You will learn the basic shapes you need for drawing on the window and how to make some simple interactive sketches that react to input from the real world. Some of the sketches we write in this chapter will be reused later in the book, so we will also try to make our code readable and maintainable.

4.1. Shapes, colors and coordinates

Processing uses a coordinate system where the left upper corner is (0,0) and the right lower corner is (width,height). The size of the window is defined by the size() command we use in the setup() method. In the draw() method we can use a variety of shapes to draw our content. So let's try some of them to draw a little image. Start with a setup() method and define a window with a size of 300x300 pixels

```
void setup() {
    size(300,300);
}
```

Now we set the background color of our sketch in the draw method. This time we want the background to be light blue. Colors can be specified in various ways in Processing. If we want to specify a gray level we supply one value, if we want another color we need to specify an RGB value, red, green and blue. This will be mixed to our color. To get a shade of blue we set the background(180,180,255).

There are even more ways for defining colors, but we don't cover them in depth here. If you want to know what HSB means or need to define your own color ranges have a look at the Processing reference.

```
void draw() {
    background( 180,180,255 );
}
```

Now, we tell our computer to draw a rectangle in the middle of the window. This rectangle should be filled with white. So we change the fillcolor using the fill() command and draw a big white box.

```
void draw() {
    ...
    fill(255);
    rect( 50,100,200, 100);

}
```

Good now lets add two more boxes. A brown one on the left and a blue one on the right.

```
  ...
  fill(200, 100, 0);
  rect( 85, 125, 50, 75 );
  fill( 180, 180, 255 );
  rect( 175, 125, 50, 50 );
  ...
```

OK. I guess you already know where this is going. We have our little door and window in place. So how do we draw a roof? The first part is easy we want the roof to be red so we set the fill() color to (255,0,0). Then we use a feature of Processing that alows us to draw even the most complex polygons. We tell Processing to draw a polygon by defining the edges. When we want to draw a polygon we tell processing to start with beginShape(). Then we specify the edges by calling vertex and when we finished we call endShape(CLOSE). The CLOSE parameter indicates that we want our polygon to be closed, so Processing will draw a line from the last vertex we specify to the firstone.

```
  ...
  fill(255,0,0);
  beginShape();
  vertex(50,100);
  vertex(100,50);
  vertex(200,50);
  vertex(250,100);
  endShape(CLOSE);
  ...
```

Now that our house has a decent roof we need to add some ground level, since our house seems to float in space. So, we add another rectangle colored in green, but this time we have to add it bevore the white rectangle or our grass will get drawn in front of the house.

```
void draw() {
  background(180,180,255);

  fill( 100, 255, 0 );
  rect(0,150,width,150);
  ...
```

This time we want to get rid of the fine black line that surrounds the green rectangle. So we tell Processing to turn it off by calling noStroke() before the rectangle. To turn the stroke on again we can use the stroke() command and specify a color our lines should have.

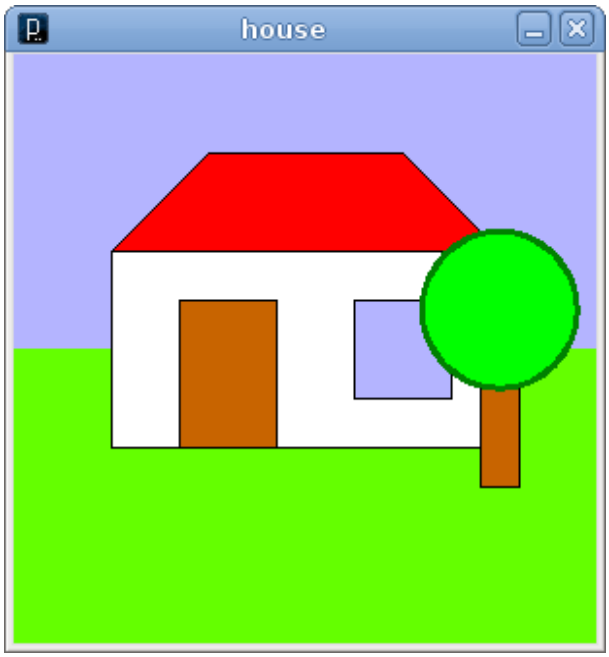
```
void draw() {
  background(180,180,255);

  fill( 100, 255, 0 );
  noStroke();
```

```
rect(0,150,width,150);
stroke(0);
strokeWeight(1);
...
```

So we have a house with a roof, a door and a window, we have our floor, so let's add a tree. First we draw another small brown rectangle and then we add a circle for the treetop. For this circle we will use ellipse() and tell Proccesing to use dark green and a three pixel wide border, by using the stroke() and the strokeWeight() methods.

```
...
fill( 200, 100, 0 );
rect( 240, 130, 20, 90 );

fill( 0, 255, 0 );
stroke( 0, 128, 0 );
strokeWeight( 3 );
ellipse( 250, 130, 80, 80 );
...
```

Perfect! Now we have a small house with a red roof and a tree in front and without much effort. We have learned how to draw shapes, how to specify fill and draw color, and how to specify coordinates in Processing. Now, let's try to make another picture by using the shapes we have learned.



4.2. Please Follow the Mouse

Let's start a new Sketch. The sketch will feature a little white ball that really loves your mouse pointer and therefore will follow it wherever it goes. We start again by defining a setup() method and tell Processing how big our window should be.

```
void setup() {
    size( 300, 300 );
}
```

Now we add a draw() method to set the background to black and draw a little white ball in the middle of the window. It won't move yet it just sits there and waits.

```
void draw() {
    background( 0 );
    strokeWeight( 3 );
    stroke( 255 );
    fill( 128 );
    ellipse( widht/2, height/2, 20, 20 );
}
```

We have our ball sitting in the middle of the screen and waiting patiently. Let's make it move. Replace the first coordinate of the ellipse() by mouseX and the second by mouseY

```
void draw() {
    background( 0 );
    strokeWeight( 3 );
    stroke( 255 );
    fill( 128 );
    ellipse( mouseX, mouseY , 20, 20 );
}
```

Hit the play button again and - TADA - you wrote your first interactive computer program. A small ball that follows the mouse. In chapter XXX we will cover interaction with mouse, keyboard and various other devices.

4.3. Make the ball bounce around

The next sketch we are going to write features our little ball again. This time we will not let it follow the mouse but bounce around in the window on its own. Every time it hits the wall it will bounce off and move in the opposite direction. Start again by writing our setup() method and tell Processing how big our window should be.

```
void setup() {
    size( 300, 300 );
}
```

Add commands in the setup() method to change the stroke and fillcolor. We also add the smooth() command which tells Processing to smooth the edges of circles and lines in our drawing.

```
    ...
    smooth();

    fill(128);
    strokeWeight(3);
    stroke(255);
    ...
```

Now lets add a draw() method again and add our little white ball in the center.

```
void draw() {
    background(0); // make the background black
    ellipse( width/2, height/2, 20, 20 ); // draw an ellipse at the center of the window
}
```

OK, now we will make our ball move. Lets step back a little bit and think of what we need. First we define a little universe without friction or gravity for our ball. so if it hits a wall it doesn't loose any speed and it won't get dragged towards the floor. Next we need to know the exact position and the exact speed our little ball has at any given time (so there won't be a Heisenberg uncertanty in our little universe). If we know the position and the velocity of the ball at time t we can calculate the position of the ball at time t+1 by simply adding the velocity to the current position. So we need a way of storing the current position and the velocity.

Add the following two lines before the setup method.

```
PVector pos;
PVector vel;
```

These are variables where we can store the position and speed of our ball. You can imagine them as little post-its your computer uses to store the values you tell it to remember. In our setup() method we tell the computer what the inital values of these two variables should be and in the draw method we calculate the next position of our ball.

TODO: explain PVector

We want to start our ball in the center of the screen and move it upwards and to the right. So we set our pos variable to (width/2, height/2) and our speed to (10,-5). The ellipse() in our draw() method will now be drawn at pos.x, pos.y instead of width/2, height/2.

```
    ...
    pos = new PVector( widht/2, height/2 );
    vel = new PVector( 10, -5 );
    ...
```

When we hit the play button not much has changed. The ball still stays in the middle of the screen and waits patiently for us to tell him what to do. So we modify our draw() method and calculate the next position by adding the velocity to the current position.

```
void draw() {
    ellipse( pos.x, pos.y, 20, 20 );

    pos.add( vel ); // add the velocity to the current position
}
```

When we hit the play button this time the ball starts moving - but wait it flies out of the window! We have to tell our program what to do when the ball hits the wall. So, we modify our draw() method again and add some if(){} statements. An if(){} statement is a way of telling the computer what todo if a certain conditions occurs. We want to tell our computer that IF the ball hits the right wall it shall move to the left, IF it hits the left wall it shall move to the right, IF it hits the top wall it shall move down and IF it hits the button wall it shall move upwards.

How can the computer check if the ball has hit a wall? By comparing the position of the ball and the wall. Let's start with the right wall. The right border is at position width. If we want to see if our ball has hit the wall we check if pos.x is larger (>) than width. Let's try this and add the following lines to our draw method. the code tells our computer to check if our ball has hit the wall and if so simply stop it.

```
void draw() {
    background(0);

    ellipse( pos.x, pos.y, 20, 20 );

    pos.add( vel ); // add the velocity to the current position
    if ( pos.x > width ) {
        vel = new PVector( 0, 0 ); // stop the ball by setting the velocity to (0, 0)
    }
}
```

Click the play button and the ball should move towards the right wall and - plop - hit it and stick there. Very good. Now how do we make our ball to bounce off in the other direction? By reversing the sign of our velocity. When the ball hits the right or the left wall we reverse the sign of the X direction, when the ball hits the top or the bottom wall we reverse sign of the velocity in the the Y-direction. Let's add some more code to our draw() method to make the direction change.

```
void draw() {
  background(0);

  ellipse( pos.x, pos.y, 20, 20);

  pos.add( vel );
  if ( pos.x > width ) { // right wall
    vel = new PVector( -vel.x, vel.y );
  }

  if ( pos.x < 0 ) { // left wall
    vel = new PVector( -vel.x, vel.y );
  }
```

```
  if ( pos.y > height ) { // top wall
    vel = new PVector( vel.x, -vel.y );
  }

  if ( pos.y < 0 ) { // bottom wall
    vel = new PVector( vel.x, -vel.y );
  }
}
```

Now we click the play button again and now our ball should bounce off the walls just as we excpected. Congratulations! You wrote your first physical simulation program (did i mention that most programmers live near the border to megalomania?). But to be honest our program could optimise in our code. The first thing is that our ball is a bit fast and if we run the program on a slower computer our ball will move at a different speed. This is because Processing calls the draw() method as often as it can. So if Processing can call the method 12 times on one computer and 60 times on a faster computer the ball will move 5 times faster. But we would like it to run the same speed on all computers. So, let's tell Processing how often we want the draw method to be called by adding frameRate(25) to our setup() method.

```
void setup() {
  size(300,300);
  smooth();
  strokeWeight(3);
  fill(128);
  stroke( 255 );


  pos = new PVector( width/2, height/2 );
  vel = new PVector( 10, -5 );

  frameRate( 25 ); //  display the draw method 25 times a second
}
```

Now-, The program runs the same speed on all computers (except on a very slow one that can't run the draw method 25 times a second). The next thing that is not perfect is our wall detection. The ball seems to sink into the wall a bit every time. This is because our check code does use the middle of our ball to compare it with the wall. We have to take the radius of our ball into consideration. So, let's change and add (or subtract) the radius to the ball to the postion when checking for a hit.

```
void draw() {
  background(0);
  ellipse( pos.x, pos.y, 20, 20);

  pos.add( vel );
  if ( pos.x + 10 > width ) { // right wall
    vel = new PVector( -vel.x, vel.y );
  }

  if ( pos.x - 10  < 0 ) { // left wall
    vel = new PVector( -vel.x, vel.y );
  }
```

```
  if ( pos.y + 10 > height ) { // top wall
    vel = new PVector( vel.x, -vel.y );
  }

  if ( pos.y - 10 < 0 ) { // bottom wall
    vel = new PVector( vel.x, -vel.y );
  }
}
```

The next thing we will change won't affect the behaviour of our code but the look of the code. When we look at the if(){} sections, we see that the code we entered for handing the left wall and handing the right wall is exactly the same. So let's combine the if checks and tell the computer to execute the code if the ball hits the left wall OR the ball hits the right wall. And then we do the same for the top and the bottom walls.

```
void draw() {
  background(0);
  ellipse( pos.x, pos.y, 20, 20);

  pos.add( vel );
  if ( pos.x + 10 > width ||  pos.x - 10  < 0 ) { // ball hits either right OR left wall
    vel = new PVector( -vel.x, vel.y );
  }

  if ( pos.y + 10 > height || pos.y - 10 < 0 ) { // ball hits either top OR bottom wall
    vel = new PVector( vel.x, -vel.y );
  }
}
```

Now our draw method looks much more compact and is easier to maintain. "Maintain?!", I hear you say. Yes little padawan, even some very small sketches like this one will get reused sooner or later, so always try to write your code to be readable. It can be really embarrasing when you have to look at your own code some time later and having no clue what it does.

4.4. Some more Crayons

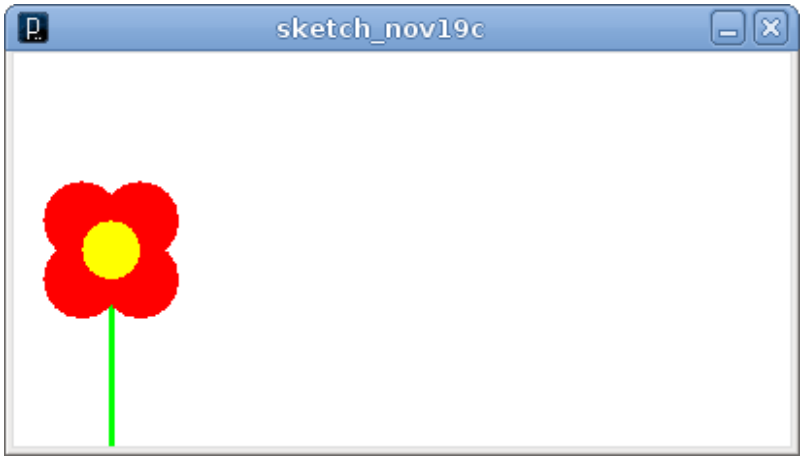
Besides having written a view funny sketches, you have learnd nearly all the stuff a programmer needs to survive. Yes, really! You know how if works, you have written some statements to methods, you have compiled your program, exported it, defined and used some variables, and so on. The last view "Crayons" you haven't used so far will be introduced in this section, but have no fear. We will just write a little sketch as we did in the sections before.

So let's start another sketch by starting Processing or clicking on the new button. Add the setup() and the draw() method and tell Processing to make our window 400 by 200 pixels.

```
void setup() {
  size( 400, 200 );
}
```

Now, let's draw some flowers. Start with a green line and add some circles in red and yellow.

```
void draw() {
  background(255);

  strokeWeight(3);
  stroke(0,255,0);

  line( 50, 200, 50, 100 );

  noStroke();
  fill( 255, 0, 0 );
  ellipse( 35, 85, 40, 40 );
  ellipse( 65, 85, 40, 40 );
  ellipse( 65, 115, 40, 40);
  ellipse( 35, 115, 40, 40 );

  fill( 255, 255, 0 );
  ellipse( 50, 100, 30, 30 );

}
```



If we want to move our flower to the right, this would require us to correct the position of all the ellipses and our line. We would need to edit 5 ellipses and a line and change 7 coordinates just to move our flower a bit to the right. Hmmm. That doesn't sound fun. So let's change the flower to make the moving part easier. We need a way to tell Processing where the flower should be drawn and define the draw operation relative to that position. So lets introduce a variable.

Variables are like little containers, computer programs can use to store values of a certain type. In our case we want to store a number. We don't need fractions and we need no very small numbers so we can happily use a variable of the type integer to store our values. Let's change our program a bit to make the draw() method look like this.

```
void draw() {
  background(255);

  strokeWeight(3);
  stroke(0,255,0); // green
  int pos = 50;

  line( pos, 200, pos, 100 );
```

```
  noStroke();
  fill( 255, 0, 0 );
  ellipse( pos - 15, 85, 40, 40 );
  ellipse( pos + 15, 85, 40, 40 );
  ellipse( pos + 15, 115, 40, 40);
  ellipse( pos - 15, 115, 40, 40 );

  fill( 255, 255, 0 );
  ellipse( pos, 100, 30, 30 );
}
```

If we want to move our flower to the right, now we only have to change the content of one variable: pos. Since most computer programmers don't like stupid repetiteive work, this is much better. So, now we want to change the height of our little flower. So lets introduce another variable and change our draw method().

```
void draw() {
  background(255);

  strokeWeight(3);
  stroke(0,255,0);
  int pos = 50;
  int hi = 100; // height of the flower

  line( pos, 200, pos, 200 - hi );

  noStroke();
  fill( 255, 0, 0 );
  ellipse( pos - 15, 200 - hi - 15, 40, 40 );
  ellipse( pos + 15, 200 - hi - 15, 40, 40 );
  ellipse( pos + 15, 200 - hi + 15, 40, 40 );
  ellipse( pos - 15, 200 - hi + 15, 40, 40 );

  fill( 255, 255, 0 );
  ellipse( pos, 200 - hi, 30, 30 );
}
```

The 200 - hi stuff is necessary since we want our flower to grow from bottom to top and the bottom line is 200 and the top line is 0 so if we want our flower to grow from the bottom line and be 100 pixel tall, it has to start at 200 and bloom at 200 - 100.

If we want Processing to draw a second flower, we could duplicate all the code in the draw() method, but again - that wouldn't be much fun. So let's make something different and introduce a method. We already defined some methods - setup() and draw(). So they are not that new. Create the following functions have a return value, which can be void if we don't want to calculate or return any values, and some optional parameters. We want our flower function to have two parameters position from the left and height of the flower. So let's define a function and move the code for drawing our flower from the draw method to our new function. Then instead of drawing the ellipses() and lines() in the draw() method we call the function we just defined.

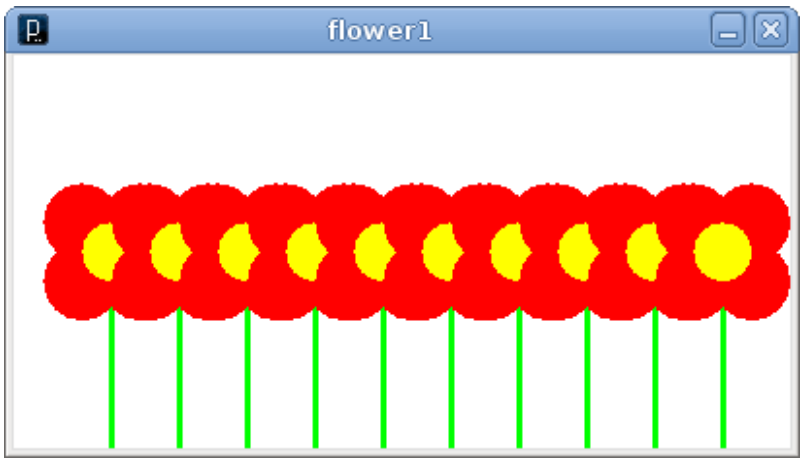
```
void draw() {
  background(255);
```

```
  flower( 50, 100 );
  flower( 100, 110 );
}

void flower( int pos, int hi ) {
  strokeWeight(3);
  stroke(0,255,0);

  line( pos, 200, pos, 200 - hi );

  noStroke();
  fill( 255, 0, 0 );
  ellipse( pos - 15, 200 - hi - 15, 40, 40 );
  ellipse( pos + 15, 200 - hi - 15, 40, 40 );
  ellipse( pos + 15, 200 - hi + 15, 40, 40 );
  ellipse( pos - 15, 200 - hi + 15, 40, 40 );

  fill( 255, 255, 0 );
  ellipse( pos, 200 - hi, 30, 30 );

}
```

Great! Now, we can define as many flowers as we like by simply calling our flower method. But if we would like to draw 10 of them? Do we need to write flower(bla, bla) 10 times? This is exactly the boring repetitive work programmers don't like, isn't it? Luckily computers are very good in automatising boring repetitive work - in fact they where invented to do this kind of stuff. So let's tell our computer to draw() the 10 flowers for us by changing the draw method like this

```
void draw() {
  background(255);
  for ( int i = 0; i < 10; i++ ) {
     flower( 50 + 18 * i, 100 );
  }
}
```

The construct for() is called a for loop it defines a variable i and sets its start value to 0. Then while i is less than 10 it executes the code block (in our case the call to flower) and increments the variable (i++ means add 1 to the variable i). Then it repeats until i is not less than 10.



We have grown flowers, now let's gow trees in the next chapter!

5. Rotate, Scale, Translate

In the last section you learned how to grow a bunch of flowers in your Processing sketch. This time we go one step further and plant some trees. And because planting virtual trees in a row would be too easy, we will plant them on our own little microplanet.

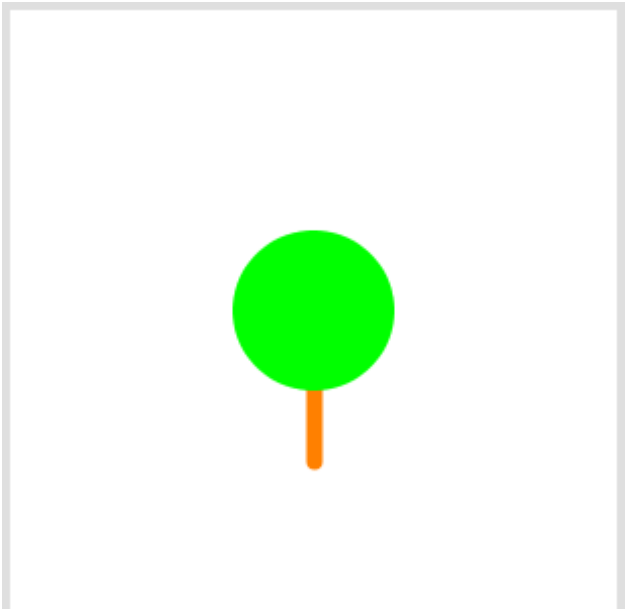
5.1. Green Microplanet

Let's start simple with a new sketch that draws a single tree. Start again with a setup() method and set the size of the window to 300 by 300. We also don't want the draw method to be looped this time - we only draw a single frame. So, add a noLoop() statement to the setup() method.

```
setup() {
    size(300,300);
    smooth();
    noLoop();
}
```

Now add a draw method and draw a simple tree using a line and an ellipse.

```
draw() {
    background(255);
    strokeWeight(8);
    stroke(255,128,0);
    line( 150, 225, 150, 150 );
    noStroke();
    fill(0,255,0);
    ellipse( 150,150,80,80);
}
```



We want to plant many trees on our little planet, so we will move the drawing code to a method again. But this time we don't add parameters to move the tree around but draw it with 0, 0, as origin. This allows us to move the tree around by using the translate() method. It will also make it a lot easier to rotate the tree around our planet.

So let's add a new method named tree that draws a tree starting at 0, 0.

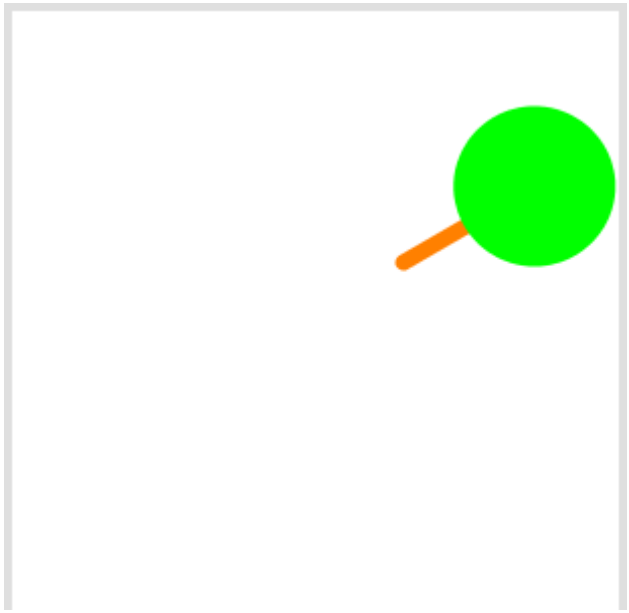
```
void tree() {
  strokeWeight(8);
  stroke(255, 128, 0);
  line( 0, 0, 0, -75 );
  noStroke();
  fill(0, 255, 0 );
  ellipse( 0, -75, 80, 80 );
}
```

To make the tree visible we have to translate the origin to 150, 300 by using translate and then call our tree() method. By using translate() you can move the origin of your drawing area around in the window like a sheet of paper that is below a frame.

```
void draw() {
    background( 255 );
    translate( 150, 300 );
    tree();
}
```

Now we move our tree to the center of the window and add the rotate() method too. Since our little planet will be round and we don't want all the trees to grow on the top of it.

```
void draw() {
    background( 255 );
    translate( 150, 150 );
    rotate( radians( 60 ));
    tree();
}
```



Now let's add our little planet by drawing a circle in the center of the screen. To make the circle for our planet independent from our trees, we need to surround the drawings of our tree by pushMatrix and popMatrix. PushMatrix starts a new drawing context that can be moved or rotated independently. Imagine it like adding a overhead transparency to your drawing that can be moved around. If you are to young to know overhead projectors, go ask your parents or grandparents.

```
void draw() {
    background( 255 );
    strokeWeight(2);
    stroke(0);
    fill( 0, 255, 0 );
    ellipse( 150, 150, 100, 100 );

    pushMatrix();

    translate( 150, 150 );
    rotate( radians( 60 ));
    tree();

    popMatrix();

}
```

Hmmm. Now the tree seems to grow from the center of the planet, instead of the surface. So let's add another translate() method that moves the tree to the surface of the planet. This one has to be added before the call to rotate since we still want the tree to be rotated at the center of the planet. So change the draw method to this:

```
void draw() {
    background( 255 );
    strokeWeight(2);
    stroke(0);
    fill( 0, 255, 0 );
    ellipse( 150, 150, 100, 100 );
```

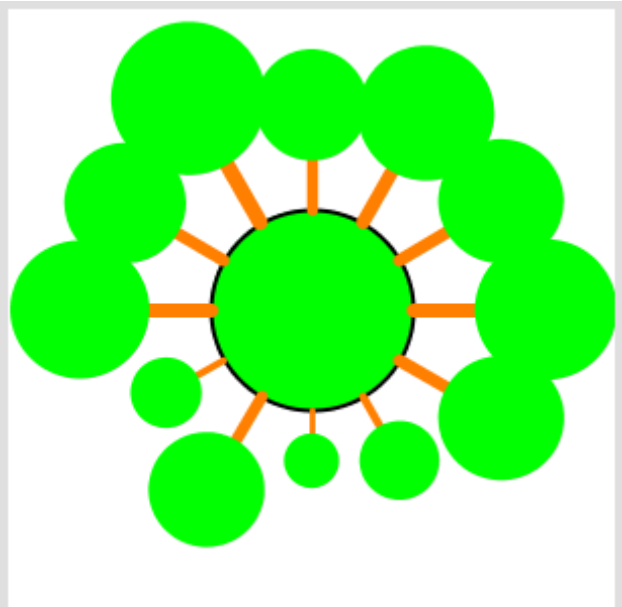
```
    pushMatrix();

    translate( 150, 150 );
    rotate( radians( 60 ));
    translate( 0, -50 );
    tree();

    popMatrix();

}
```

Much better. Now we can go on and add some more trees. We use a for loop again and plant 12 trees on our little planet.

```
void draw() {
    background( 255 );
    strokeWeight(2);
    stroke(0);
    fill( 0, 255, 0 );
    ellipse( 150, 150, 100, 100 );

    for( int i=0; i < 12; i++ ) {
      pushMatrix();
      translate( 150,150 );
      rotate(radians(30 * i));
      translate(0, -50);
      tree();
      popMatrix();
    }
}
```

Now the trees grow around our little planet, but they are all the same size. Let's add another command to make the trees grow a little bit less uniformly. The last command I will introduce in this section is scale(). We will scale the tree by a random factor before we translate and rotate it so some of the trees will grow smaller than others. Change the for loop in the draw method to look like this

```
  for( int i=0; i < 12; i++ ) {
    pushMatrix();
    translate( 150,150 );
    rotate(radians(30 * i));
    translate(0, -50);

    float s = random( 0.3, 1 );
    scale(s,s);

    tree();
    popMatrix();
  }
```



5.2. Rotating Image

The transformation of the drawing context we used in the last section is not limited to images you draw, but can also be used to transform images. In this section we scale and rotate a image we load from the disk.

Start with a new Processing sketch and set the size to 300,300 and turn on the smoothing.

```
void setup() {
    size(300,300);
    smooth();
}
```

Now we add an image to the Processing sketch. Select "Add file ..." from the "Sketch"-Menu and select an image you like. The image gets copied to the "data" folder of the sketch. This is where Processing sketches look first if you tell them to load something. This also works transparently with sketches you export as applets.

In the Processing sketch we need a variable that acts as a reference to the image. So define a variable type PImage in front of the setup() method and load the image using the loadImage() method:

```
PImage img;

void setup() {
    size(300,300);
    smooth();
    img = loadImage( "image.png" );
}
```

In the draw() method we can now tell Processing to draw the image in our window, by using a method called image:

```
void draw() {
    background(255);
    image( img, 0, 0 );
}
```

To make the image centered we can use the translate() method. We tell Processing to draw the image so that the left upper corner of the image is at (-img.width/2, -img.height/2) and then move the coordinate system to the center of the window. Sounds complicated, but is much simpler when you see it in code. Change the draw() method to:

```
void draw() {
    background( 255 );
    translate( 150, 150 );
    image( img, -img.width/2, -img.height/2 );
}
```

Now we can use the mouse to rotate the image. Therefore we map the value of the mouseX variable to an angle we want the image to rotate. Processing has a very convenient command called map() that can be used to map a value from one range to a value from the second range. We tell map() to take a value from the range 0-300 and map it to a value from the range -PI - +PI. So if our value is 0 it gets mapped to -PI and if our value is 150 it gets mapped to 0.

This is exacly what we need for the rotate method since Processing wants angles defined in radians (a circle doesn't have 360 degrees but 2PI degrees rad)

```
void draw() {
  background( 255 );
  translate( 150,150 );
  rotate( map( mouseX, 0, 300, -PI, PI ));
  image( img, -img.width/2, -img.height/2);
}
```

Cool! We can rotate the image now by moving the mouse to the left or to the right, but wait our mouse has another coordinate we haven't used so far! Let's scale the image by using the mouseY variable. This time we map the input range of 0-150 to -2 - +2 and use the scale() command to resize the image:

```
void draw() {
  background( 255 );
  translate( 150,150 );
  rotate( map( mouseX, 0, 300, -PI, PI ));
  float s = map( mouseY, 0, 300, 0.25, 4 );
  scale( s, s );
  image( img, -img.width/2, -img.height/2);
}
```

6. A Simple Game

In this chapter you will learn how to write a simple game in Processing - an arcanoid clone. The game has a paddle on the bottom of the window that can be controlled by the mouse and a moving ball, similar to the bouncing ball demo, I showed you earlier. When the ball hits the bottom border of the window, one live is lost. On top of the game window there are rows of bricks which disappear when hit by the ball. The game ends when all the bricks are removed or the player has no lives left.

Sounds pretty simple. Let's start with something we already know - a moving ball. So open the bouncing ball sketch from the last chapter and save it under a new name using "Save As" in the file menu. It's important that you save it under a new name or you would overwrite the original sketch. So your sketch should look like this:

```
PVector pos;
PVector vel;

void setup() {
  size(300,300);
  smooth();
  strokeWeight(3);
  fill(128);
  stroke( 255 );
  frameRate( 25 );
  pos = new PVector( width/2, height/2 );
  vel = new PVector( 10, -5 );
}

void draw() {
  background(0);
  ellipse( pos.x, pos.y, 20, 20);

  pos.add( vel );
  if ( pos.x + 10 > width ||  pos.x - 10  < 0 ) {
    vel = new PVector( -vel.x, vel.y );
  }

  if ( pos.y + 10 > height || pos.y - 10 < 0 ) {
    vel = new PVector( vel.x, -vel.y );
  }
}
```

The first thing we do is seperate the calculation of the position form the drawing code. We define another method named update() that calculates the changing positions of the ball.

```
void draw() {
  background(0);

  update();

  ellipse( pos.x, pos.y, 20, 20);
```

```
}

void update() {
  pos.add( vel );
  if ( pos.x + 10 > width ||  pos.x - 10  < 0 ) {
    vel = new PVector( -vel.x, vel.y );
  }

  if ( pos.y + 10 > height || pos.y - 10 < 0 ) {
    vel = new PVector( vel.x, -vel.y );
  }
}
```

Now we add a new boolean variable that indicates if the game is currently paused. We update the new position only if the game is not paused and tell Processing to toggle the pause variable if someone clicks the mousebutton. First we add a paused variable before the setup method and set it to true;

```
boolean paused = true;
```

Then we change our draw method, to make it call update only if the paused variable is NOT true (the exclamation mark means "not" in Java and Processing):

```
void draw() {
  background(0);

  if ( !paused ) {
      update();
  }

  ellipse( pos.x, pos.y, 20, 20);
}
```

And finally we add a new method that gets called everytime a mousebutton is clicked. This method get's a special name - mousePressed(). If Processing finds a method named mousePressed(), it will get called when the mousebutton gets clicked. There are other special named methods like keyPressed to - we will use them in a later chapter. If you want to impress some of your friends these methods are called "callback handler".

```
void mousePressed() {
  paused = !paused;
}
```

When you click the play button, the ball will stay in the middle of the screen and wait until the mouse button click in the application window. When you click the mouse again the ball will stop. Now we edit the behaviour of our program. When the ball hits the bottom wall we don't want

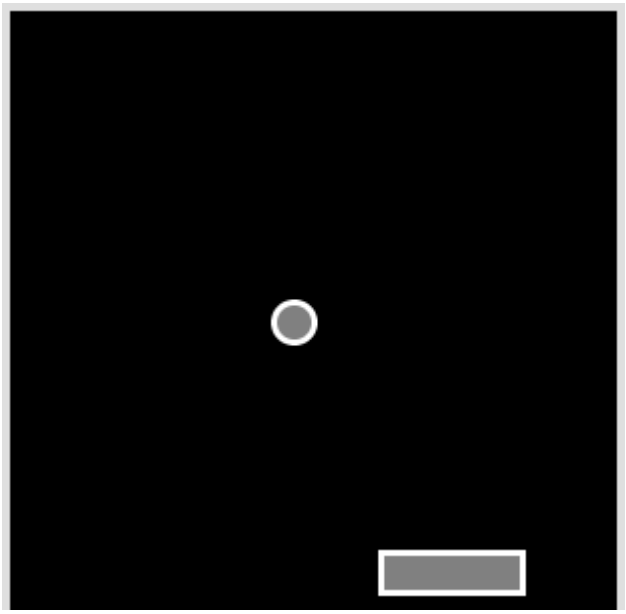
it to bounce off anymore but to jump right back in the middle of the screen and pause there. Later we will decrease the number of lives left - but let's take at a time. Let's split up the if(){} that handles the top wall and the bottom wall, since we want them to do different things now.

```
void update() {
  pos.add( vel );
  if ( pos.x + 10 > width ||  pos.x - 10  < 0 ) {
    vel = new PVector( -vel.x, vel.y );
  }

  if ( pos.y - 10 < 0 ) {
    vel = new PVector( vel.x, -vel.y );
  }

  if ( pos.y + 10 > height ) {
    vel = new PVector( vel.x, -vel.y );
    pos = new PVector( width/2, height/2 );
    paused = true;
  }

}
```

To make it visible for the player that the ball has dropped out on the bottom side we pause the game. The velocity gets reversed on the y axis so when the player clicks the mousebutton the ball starts flying to the top and not to the bottom. Otherwise the ball would be really difficult to catch.

Now that the ball can drop off the screen on the bottom we should give the player an opportunity to catch it before that happens. So let's add a paddle that is controlled by the mouse. Our paddle will be a rectangle on the bottom of the screen. Change the draw method and add the rectangle.

```
void draw() {
  background(0);

  if ( !paused ) {
      update();
  }

  ellipse( pos.x, pos.y, 20, 20);

  rect( mouseX - 35, 270, 70, 20 );
}
```



So far we drew the paddle, but the ball doesn't even notice it's there. So let's change the update method to make the ball bounce off. We will change the angle and the speed of the ball depending on position it hits the paddle. When the ball hits the paddle near the edge it will be faster and the angle will be more flat than when it hits the paddle near the center.

Therefore we use a new command named map, which maps a value from a source range to a target range. When you have 20 people in a room and 2 have red hair and you need to calculate the percentage of redhaired people you could use the map command with the source range 0 - 20 and the target range 0 - 100. I'm sure you didn't sleep in math and you could make such calculations if i wake you up in the middle of the night, but if Processing offers such nice little helper methods, why not use them? So we want to map the distance of the ball to the mousepointer to the x coordinate of the speed. Since the paddle is 70 pixel wide and the mousepointer is in the middle of it we have a source range of -35 to 35. The x coordinates of our ball should range from -10 to 10. Add the following lines to the update method

```
// check if the paddle was hit
if ( pos.y >= 260  && pos.x >= mouseX - 35 && pos.x <= mouseX +35 ) {
  vel = new PVector( int(map( pos.x - mouseX, -35, 35, -10, 10 ), -vel.y);
}
```

The if command above shows a new way of combining statements. This time we have not combined them with "or" but with "and" so the check we are performing translates to IF pos.y is greater or equal 270 AND p.x is bigger then left side of our paddle AND p.x is smaller than the right side of the paddle THEN we calculate a new velocity.

So what's missing now are the bricks our ball will remove. We need rows and columns of stones on the top of the screen and some code to test if any of the stones was hit. We need also some logic to check if all stones are removed, because then the player has won the game. So let's start with a place to store the stones. We need to define something called array. Arrays are like a row of variables that have an index. So lets start with one row of stones. Insert the following line before the setup method.

```
int[] stones;
```

This line of code defines an array of integer values. But Processing doesn't yet know how big our array should be and what values go inside. So let's tell it in the setup method.

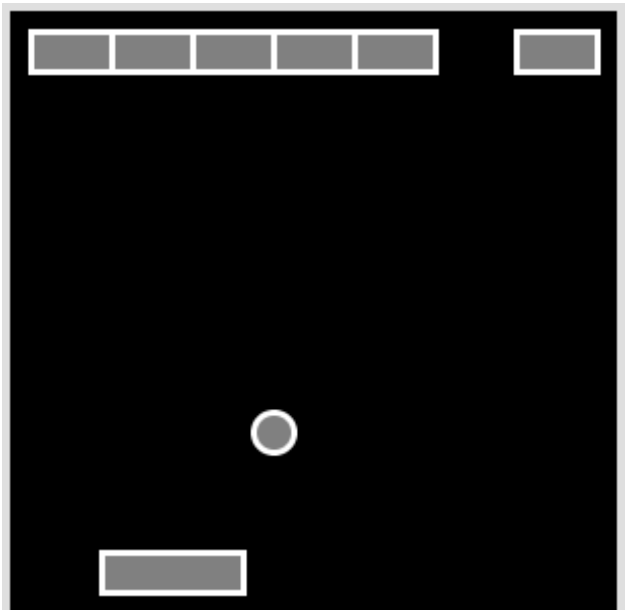
```
stones = new int[7];
for( int x = 0; x < 7; x++ ) {
    stones[x] = 1;
}
```

Now add the following lines to the draw method to make the stones visible

```
for( int x = 0; x < 7; x++ ) {
    if ( stones[x] > 0 ) {
        fill( 128 );
        rect( 10 + x * 40, 10, 40, 20 );
    }
}
```

Now we have a single row of stones in place, but the ball still doesn't care about them. We need additional checks in our update method to check if the ball hits a block. If so we reverse the direction and remove the ball by setting the entry in the array to 0. The check method also has to care if the stone was hit on the side because in that case we have to reverse the x-axis of the velocity. if not we reverse the y-axis. Add the following lines to your update method.

```
  // check if the ball hits a block
  for( int x = 0; x < 7; x++) {
    if ( stones[x] > 0 ) {

      if ( pos.x + 10 > 10 + x * 40 && pos.x - 10 < 10 + x * 40 + 40 &&
        pos.y  + 10 > 10 && pos.y - 10 < 30 ) {
        stones[x] = 0;

        // change the velocity in y direction if the block has been hit
        // on the bottom or on the top
        if ( pos.x > 10 + x * 40 && pos.x < 10 + x * 40 + 40 ) {
            vel = new PVector( vel.x, -vel.y );
        }

        // change the velocity in the x direction if the block has been hit on the side
        if ( pos.y > 10 && pos.y < 30 ) {
            vel = new PVector( -vel.x, vel.y );
        }
      }
    }
  }
```



Now we have some stones in place and we can remove them, but it would be much cooler if we had not only one row of stones. So we have to change our code again a bit. Until now we have an array of int variables representing our row of stones. To represent a grid of stones we need an array of an array of ints - sounds cool doesn't it? You can imagine it as two nested arrays we have an array of rows and one row is an array of integers. Change the definition of the stones array and add another pair of braces

```
int[][] stones;
```

Then we need to change our setup code. We need another loop now that loops over the rows when we want to initialize the array.

```
stones = new int[7][4];
for( int x = 0; x < 7; x++ ) {
  for( int y = 0; y < 4; y++ ) {
    stones[x][y] = 1;
  }
}
```

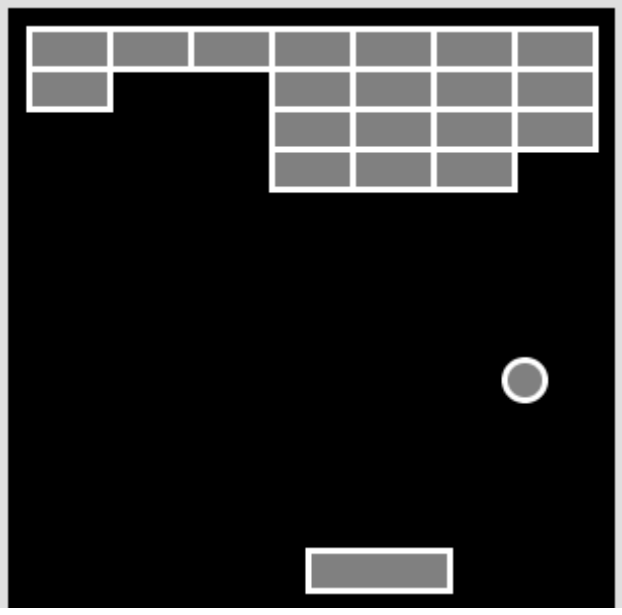
Now we must change the draw method. The change is pretty much the same as in the setup method. We just add another for loop, that loops over the rows and add the y variable to the rect so the rows don't get drawn on the same y position.

```
for( int x = 0; x < 7; x++ ) {
  for( int y = 0; y < 4; y++ ) {
    if ( stones[x][y] > 0 ) {
        fill( 128 );
        rect( 10 + x * 40, 10 + y * 20, 40, 20 );
    }
  }
```

```
}
```

The final change we have to make affects our update method. We have to add another loop for the rows of our array and then change the if's that check the y position.

```
// check if the ball hits a block
for( int x = 0; x < 7; x++) {
  for( int y = 0; y < 4; y++ ) {
    if ( stones[x][y] > 0 ) {

      if ( pos.x + 10 > 10 + x * 40 && pos.x - 10 < 10 + x * 40 + 40 &&
         pos.y  + 10 + y * 20 > 10 && pos.y - 10 < 10 + y * 20 + 20 ) {
        stones[x][y] = 0;

        // change the velocity in y direction if the block has been hit
        // on the bottom or on the top
        if ( pos.x > 10 + x * 40 && pos.x < 10 + x * 40 + 40 ) {
            vel = new PVector( vel.x, -vel.y );
        }

        // change the velocity in the x direction if the block has been hit on the side
        if ( pos.y > 10 + y * 20  && pos.y < 10 + y * 20 + 20 ) {
            vel = new PVector( -vel.x, vel.y );
        }
      }
    }
  }
}
```



Now we have rows of stones that can be removed by our little ball. But the ball doesn't stop if we have alle stones removed. The code is really simple compared to the nested array stuff you just managed. So lets add a new boolean variable named done before the setup method.

```
boolean done = false;
```

And in the draw method we need to check if stones are left that can be hit. So before the two nested for loops we set our done variable to true and if we find another stone to be drawn we set it to false again. So the section with the two for loops should look like this now.

```
done = true;
for( int x = 0; x < 7; x++ ) {
  for( int y = 0; y < 4; y++ ) {
    if ( stones[x][y] > 0 ) {
        done = false;
        fill( 128 );
        rect( 10 + x * 40, 10 + y * 20, 40, 20 );
    }
  }
}
```

right after the for loops we add an if block that checks if we are done and if so pauses the game.

```
if ( done ) {
    paused = true;
}
```

The little game starts getting playable. Now we need a way to restart it when all the stones have been removed. So lets change our mousePressed method and add a check if we are done. If so we reset the position of the ball and reinitialize the stones in the array.

```
void mousePressed() {
  paused = !paused;
  if ( done ) {
    for( int x = 0; x < 7; x++ ) {
      for( int y = 0; y < 4; y++ ) {
        stones[x][y] = 1;
      }
    }
    pos = new PVector( width/2, height/2 );
    vel = new PVector( 10, -5 );
    done = false;
  }
}
```

Now we have a little game that can actually be played and it also can be restarted by the player when all the stones have been removed. But our game isn't very talkative yet. It would be nice to tell the player what he has to do if the game is paused, or write a message on the screen if the player wins the game.

Processing knows a command named text, which can be used to draw a piece of text on the screen. But before we can use it we have to tell Processing what font to use. Processing uses

it's own type of fonts, but there is a tool that can be used to convert TTF fonts. So click on "Create Font" in the "Tools" menu and select a font you like. The generated font gets stored in the data folder of your sketch. So if you can't remember the exact name of the font you created, do not worry, you can go to the sketch menu and ask Procesing to show you the sketch folder.

After you have generated a font we need to tell Processing to use it. So add the following line at the bottom of the setup method. In this example I used "Verdata-Bold-48.vlw", if you want to use another make sure to change the fontname you submit to the loadFont method.

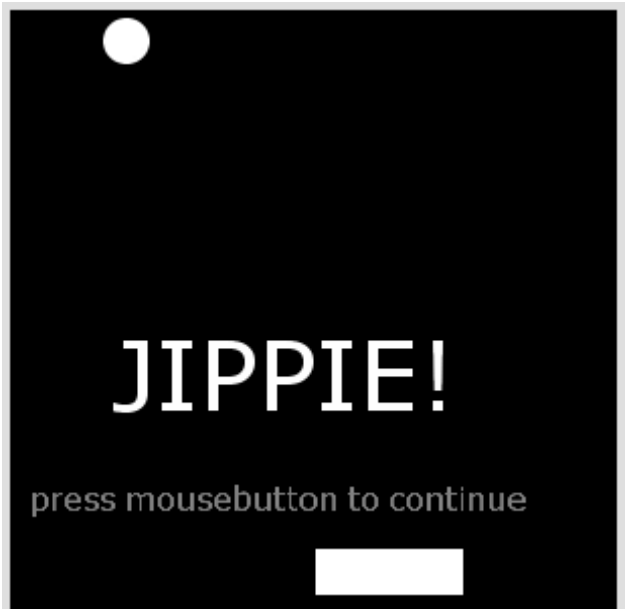
```
    textFont( loadFont( "Verdana-Bold-48.vlw" ));
```

Now we can add some lines to the draw method to display a hint when the game is paused. Use textSize() to tell Processing how big your font should be and use fill to set the textcolor

```
if ( paused ) {
    textSize( 16 );
    fill(128);
    text( "press mousebutton to continue", 10, 250 );
}
```

And finally we want to display a friendly message if the player wins the game. Add the following lines to your draw method.

```
if ( done ) {
    paused = true;
    fill(255);
    textSize( 48 );
    text( "JIPPIE!", 50, 200 );
}
```



7. Another Dimension

Processing has not only a rich set of function for drawing 2dimensional graphics, most of them can also be used for generating 3D images. In this section we are going to rework the little house from section XXXXX to make it 3D.

Even though we live in a 3D world, most of us draw and work in a 2D world, so don't be worried if you have difficulties imagining 3D coordinates or get them wrong in the first try, you will get used to them after a while.

Lets start with a first simple 3D sketch. The "Hello World" analogon for 3D programming is a rotating cube. so let's start writing one. Start with a setup and a draw method and set the size to 300x300 but this time tell Processing that you want the drawing space to be 3D by adding another parameter containing the value P3D.

```
void setup() {
  size(300,300,P3D);
}
```

In the draw method we set the background to black and tell processing to turn the lights one. Now we can add a cube by using the box method. Since the box gets drawn at 0x0x0 we need to translate it to the center of the window.

```
void draw() {
  background(0);
  translate(width/2, height/2);
  lights();

  box(100);
}
```

When you click the play button you should see your first 3D sketch, but it's not very exiting yet since it looks like a simple square. So let's rotate the cube, that everyone else also recognises what a great 3D programmer you are.

Add a float variable that holds our rotation angle and set the framerate int he setup method. In the draw method we add a rotateY method and increase a for every Frame.

insert this code before your draw method:

```
float a = 0;
```

and add this code before the box() method of your draw() method:

```
  a += 0.01;
```

```
  rotateY(a);
```

TODO: finish chapter

8. Data Visualization

Now we know how to interact with our user and how to draw something on the screen so lets move to the next area where Processing really shines - data visualisations. This is a huge topic, that doesn't fill only one but several books. So I will give you a brief introduction to get your first sketches running.

The first thing you need when visualizing data is a data source. Data can be stored in various formes ranging from simple files to complex databases, from text files to calender formats. There are plenty libraries to access these datastructures.

The next thing you need to do is extract the data from the source and prepare them for the visualisation you have in mind. Most datasources store huge loads of data, and you will need only one or two aspects of the data offered. So most of the times you will need to select, filter, aggregate, ... the data prior visualising it.

And the last (and most fun part) of it is displaying the data.

Enough theory let's get something done. In this section you will learn how to compare the length of some twitter posts on two topics. Just in case you slept in a cave for the last few years, Twitter is a micro-blogging plattform where messages cannot be longer than 140 chars. So we are going to query the twitter search using a topic as a parameter and count messages of every length. Then we are going to display the length distribution of two topics side by side.

Let's start again with a new sketch and define our setup and draw methods.

```
void setup() {
}

void draw() {
}
```

Now we are going to fetch some data from Twitter. Define a method named fetch that takes a String as parameter and returns an array of Strings as result. Twitter offeres a very simple api we can call from Processing by simply providing a url containing our search term and we will get some xml back. Processing offers various ways for parsing XML-Structures, but we will keep it simple here. Since we only need the messages and none of the other provided metadata like author, timestamp or even location we simply search for strings that start with <title> and end with </title>. Everything between these two tags is the message we are interested in. In a first step we simply print the messages using println to see if we got the fetching and string matching parts right.

```
void setup() {
    String[] tmp = fetch( "processing" );
}

String[] fetch( String about ) {
  String[] tmp = loadStrings( "http://search.twitter.com/search.atom?q=%23"+about );
  for( int i = 0; i < tmp.length; i++) {
    String[] m = match( tmp[i], "<title>(.*)</title>" );
```

```
    if( m != null ) {
       println( m[1] );
    }
  }
  return new String[] {};
}
```

Good, now lets add these strings to a data-container and return them as an array. TODO: explain match, explain ArrayList

```
String[] fetch( String about ) {
  ArrayList res = new ArrayList();
  String[] tmp = loadStrings( "http://search.twitter.com/search.atom?q=%23"+about );
  for( int i = 0; i < tmp.length; i++) {
    String[] m = match( tmp[i], "<title>(.*)</title>" );
    if( m != null && m[1].length() < 140 ) {
      res.add( m[1] );
    }
  }
  return (String[])res.toArray(new String[0]);
}
```

The next thing on our list is data preparation. We aren't really interested in the messages themselfes but only in their length, and we need to know how many messages of length x exist. Since Twitter messages can't be longer than 140 characters we can define an array containing 140 integers that will hold our sums. So if we want to know how many messages are 50 characters long we simply look up the value on index 50 in our array and there we are. So add an array variable above the setup method.

```
int[] data;
```

Now we can add another method called prepare, that takes our array of strings, counts the length of the messages and fills our data array. For every entry in the String array we look at the length and simply add one to the sum at the index of the length.

```
int[] prepare( String in[] ) {
  int[] res = new int[140];
  for (int i =0; i < 140; i++) {
    res[i]=0;
  }
  for (int i =0; i < in.length; i++ ) {
    int idx = in[i].length() -1 ;
    res[ idx ] += 1;
  }
  return res;
}
```

In the setup method we call our new prepare method to fill the data array.

```
void setup() {
    size( 280, 200 );
```

```
    data = prepare( fetch( "processing" ));
}
```

For the displaying part we don't need to invent another method since there already is one that fits perfectly. Let's add some code to the draw method. First we loop throught the data array and search for the maximum value. Then we iterate the data array a second time and draw rectangles for every entry and map the value to the height of the sketch.

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

  int maxdata = 0;
  for( int i =0; i < 140; i++) {
    maxdata = max( maxdata, data[i]);
  }

  for( int i =0; i < 140; i++) {
    float h = map( data[i], 0, maxdata, 0, 200 );
    rect(i*2,200 - h,2,h);
  }
}
```

Now we can display the length distribution of the last 15 twitterstreams on a topic (sounds important - doesn't it?). But our graphic would look far better if we compared more messages. So lets call the search several times and add the page parameter since twitter search returns 15 entries per page. Change the fetch method and add another for loop to call the twitter api 5 times.

```
String[] fetch( String about ) {
  ArrayList res = new ArrayList();
  for( int p=1; p < 6; p++) {
    String[] tmp = loadStrings( "http://search.twitter.com/search.atom?q=%23"+about+"&page="+p);
    for( int i = 0; i < tmp.length; i++) {
      String[] m = match( tmp[i], "<title>(.*)</title>" );
      if( m != null && m[1].length() < 140 ) {
        res.add( m[1] );
      }
    }
  }
  return (String[])res.toArray(new String[0]);
}
```

Since our inital goal was to compare two search terms we need another data variable and fill it with data by using prepare and fecht on another search term.

```
int[] data;
int[] data2;

void setup() {
    data = prepare( fetch( "processing" ));
    data2 = prepare( fetch( "java" ));
}
```

Now we change the draw method to display our two datasets side by side, to make them compareable. We need to search the maximum value of both datasets so we add another for loop that searches the second one, and the rectangles get draw from the middle of the sketch upwards for the first dataset and downwards for the second one.

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

  int maxdata = 0;
  for( int i =0; i < 140; i++) {
    maxdata = max( maxdata, data[i]);
  }
  for( int i =0; i < 140; i++) {
    maxdata = max( maxdata, data2[i]);
  }

  for( int i =0; i < 140; i++) {
    fill( 0, 0, 255 );
    float h = map( data[i], 0, maxdata, 0, 100 );
    rect(i*2,100 - h,2,h);
    fill( 0, 255, 0 );
    float h2 = map( data2[i], 0, maxdata, 0, 100 );
    rect(i*2,100,2,h2);
  }
}
```

If you click the run button the setup method will take some time to fetch all the data from twitter search.

IMAGE

9. Interaction

In this chapter you will learn how to make your Processing sketches interact with your user. We will start with mouse and keyboard. Then I will show you how your sketches can learn how to see, listen or feel.

9.1. Mouse

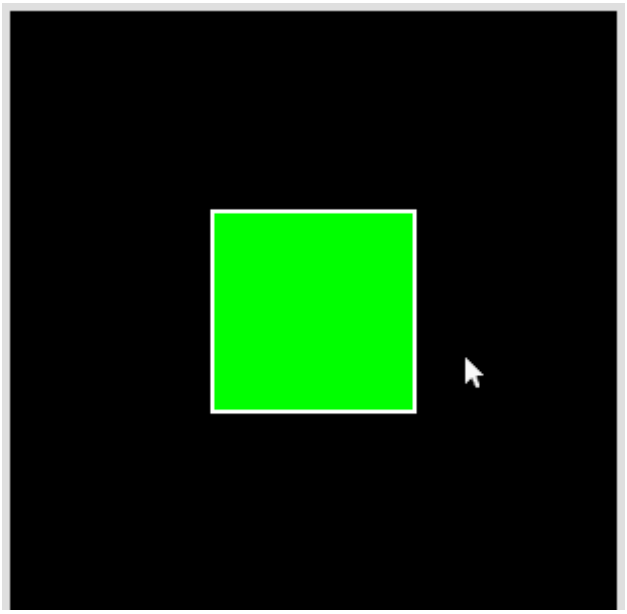
The easiest way for you to add some sort of interaction is using a mouse. Processing allows you to access the current mouse coordinates by using the mouseX and mouseY variable and you can access the currently pressed button by accessing the mouseButton variable.

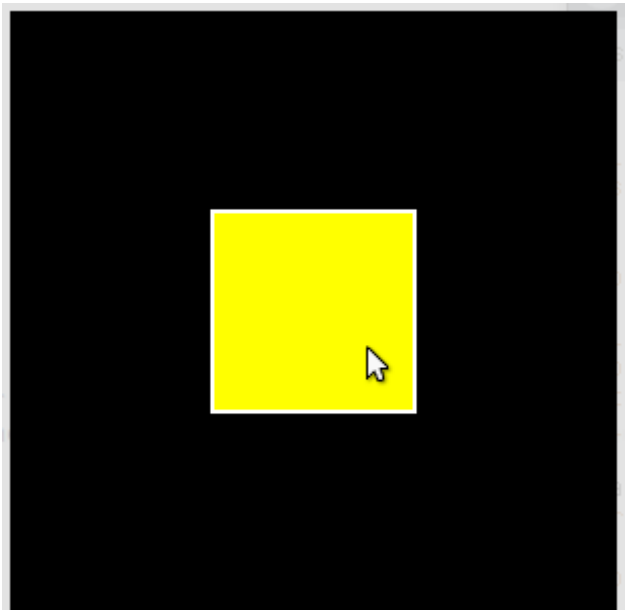
Let's write a sketch that uses the mouseX and mouseY variable to tests if the mouse is in a certain area of the window. Start with a setup and a draw method and set the size of the window to 300, 300 in the setup method.

```
void setup() {
    size(300,300);
}

void draw() {
}
```

Now we test if the X coordinate is bigger than 100 and lower than 200 and than we do the same for the same for the Y coordinate. If the mouse pointer is in the area we are looking for we change the color.

```
void draw() {
  background(0);

  stroke( 255 );
  strokeWeight( 2 );
  color c;
  if ( mouseX  > 100 && mouseX < 200 && mouseY > 100 && mouseY < 200 ) {
    c = color( 255, 255, 0 );
  } else {
    c = color( 0, 255, 0 );
  }
  fill( c );
  rect( 100,100,100,100);
}
```





So far nothing we haven't already used before. Now let's add another test to change the color only if the mousebutton is pressed and the mouse is inside our rectangle. Change the if-else section in the draw method to

```
...
  if ( mouseX  > 100 && mouseX < 200 && mouseY > 100 && mouseY < 200 ) {
    if ( mousePressed == true &&  mouseButton == LEFT ) {
      c = color( 255, 0, 0 );
    } else {
      c = color( 255, 255, 0 );
    }
  } else {
    c = color( 0, 255, 0 );
  }
...
```

Now we change the color to yellow if the mouse enters the rectange and change it to red if it is inside, a mouse button is pressed and this happens to be the left button.

Most mice today come with a scroll wheel. To access it from Processing a little more work is needed. We want to add some code to change the blue level of our colors by using the scroll wheel. So we define a variable named b of type int and initalize it to 0 and change all color definitions in our draw method to use the b variable.

So the if block in our draw method looks like this

```
...
  if ( mouseX  > 100 && mouseX < 200 && mouseY > 100 && mouseY < 200 ) {
    if ( mousePressed == true &&  mouseButton == LEFT ) {
      c = color( 255, 0, b );
    } else {
      c = color( 255, 255, b );
    }
  } else {
    c = color( 0, 255, b );
  }
...
```

Now we need to define an eventhandler for mouseWheelEvents in our setup method. Sounds far more complicated than it is. Add the following lines to the setup method.

```
int b = 0;

void setup() {
  size(300,300);

  addMouseWheelListener(new java.awt.event.MouseWheelListener() {
    public void mouseWheelMoved(java.awt.event.MouseWheelEvent evt) {
      mouseWheel(evt.getWheelRotation());
  }});
}
```

This code tell Processing to call a method named mouseWheel every time a mouseWheelEvent occures. Our operating system sends these events whenever the mousewheel is turned up or down.

Now we have to define the mouseWheel method which has an integer parameter that tells us if the wheel has moved up or down. In our mouseWheel method we change the value of b. So when the mousewheel is rotated forward the bluelevel increases, when it is rotated backwards the level of blue gets decreased. We also have to add some sanity checks in the bottom to ensure that the value of b doesn't go below 0 or higher than 255.

```
void mouseWheel( int delta ) {
  println( delta );
  if ( delta > 0 ) {
    b -= 10;
  } else {
```

```
    b += 10;
  }

  if ( b < 0 ) {
    b = 0;
  }

  if ( b > 255 ) {
    b = 255;
  }
}
```

Now click the play-button and change the color by using the mousewheel, moving around and clicking the button

9.2. Keyboard

Working with the keyboard is very similar to working with mousebutton events in Processing. You write a callback method named keyPressed and the actual key that is pressed by the user gets stored in a variable named key or keyCode

In key the letter or number of the key the user pressed gets stored. KeyCode contains some sort of identification number for the key on the keyboard. When you want to get the letters the user has pressed you use the key variable, when you need to access keys like the cursor block or the Function-Keys you need the raw key number.

In this section we will write a little game that tests now many random letters a user can type on the keyboard in 20 seconds. Let's start again with a setup method.

```
void setup() {
}

void draw() {
}
```

Now we add an array of char containing the lowercase letters from A-Z we will use random elements from this array as our target key. We also need a variable of type char to store the letter and two variables of type integer to store it's position on the screen.

```
char[] letters = new char[] { 'a','b','c','d','e','f',
'g','h','i', 'j', 'k', 'l','m','n','o','p','q',
'r', 's', 't','u','v','w','x','y','z' };

char letter;
int x, y;
```

Every time the user hits the right key on the keyboard we will choose another random element from the letters array and set x and y to a random number between 50 and 250. So lets define a function that does exactly this.

```
void newLetter() {
  letter = letters[ (int)random(26)];
  x = int(random( 50, 250 ));
  y = int(random( 50, 250 ));
}
```

Now we need to add a font to our sketch since our draw method will use text to display the letter. Select "Create Font..." from the Tools menu and add a font you like. Then load the font in the setup method and set the fonts size. If you haven't choosen Verdana adjust the fontname you use for loadFont.

```
void setup() {
  size(300,300);

  newLetter();

  textFont( loadFont( "Verdana-48.vlw"));
  textSize( 48 );
  fill(0);
}
```

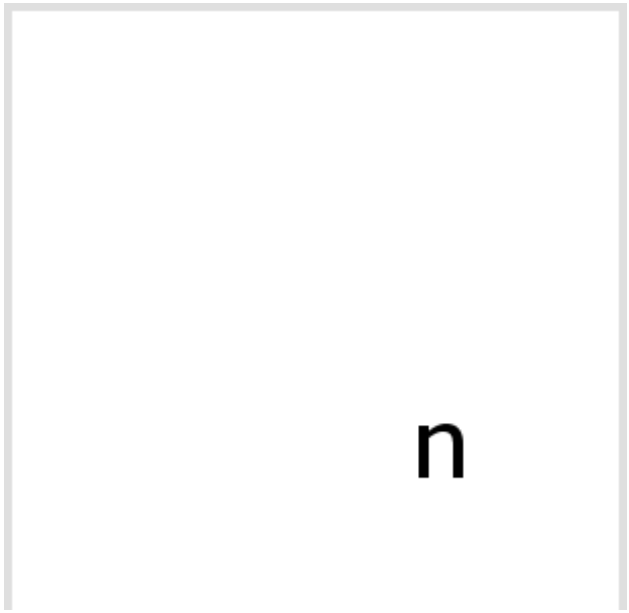
In the draw method we simply call text and display the letter on the x and y positions.

```
void draw() {
    background(255);
    text( "" + letter, x, y );
}
```

Now we need to define our keyPressed method, where we compare the key variable to our current letter and choose a new one if they match.

```
void keyPressed() {
  if ( key == letter) {
    newLetter();
  }
}
```

Click the play button and try to catch some letters. Now we need to count the letters our user has catched so far and add a timing component to limit the game to 20 seconds.



We need to add 3 more variables. A varable named count to track the number of letters so far, a variable that holds the starttime and a boolean variable that indicates if the game is running or not. Add them before the setup method

```
int count = 0;
int starttime;
boolean running = true;
```

In the setup method we set starttime to the current time and in the draw method we test if the current time minus the start time is lower than 20,000 milliseconds. If so the game is still running and we display the current letter. If not we display the gamescore and set the running variable to false

So add the following line to the setup method

```
starttime = millis();
```

And change the draw method to

```
void draw() {
  if ( millis() - starttime < 20000 ) {
    background(255);
    text( "" + letter, x, y );
  } else {
    running=false;
    background(255);
    text( "COUNT: " + count, 10, 150 );
  }
}
```

To keep track of the number of letters we increase the counter variable every time the newLetter method gets called. So your method should look like this

```
void newLetter() {
  count++;
  letter = letters[ (int)random(26)];
  x = int(random( 50, 250 ));
  y = int(random( 50, 250 ));
}
```

And in our keyPressed method we test if we are still running before we compare the keys and call newLetter

```
void keyPressed() {
  if (running && key == letter) {
    newLetter();
  }
}
```

9.3. Live Video

In this chapter we use a webcam to make your sketches react to video. Working with live video is a very complicated matter in most programming language, but in Processing accessing live-video streams and working with the live video images can be acomplished in very few lines of code. Before you can start to use video you need to install the video library. If you are using MacOS X you already done. If you use Linux or Windows you need to download the GStreamer-Library.

You can find the Processing GStream library http://sourceforge.net/projects/gsvideo/

TODO: Download and install

Let's write your first sketch that accesses live video. Connect your webcam and start with a setup and a draw method.

```
void setup() {
}

void draw() {
}
```

The Setup method looks a bit different if you are using Macos X. On Macos X we need to import the video library and initialize a Capture class

```
import processing.video.*;

Capture cam;

void setup() {
    size( 320, 240 );
    cam = new Capture( this, 320, 240, 12 );
}
```

On Linux or Windows the setup method is similar but we use GSCapture instead of the Caputure class.

```
import codeanticode.gsvideo.*;

GCapture cam;

void setup() {
    size( 320, 240 );
    cam = new GSCapture(this, 320, 240, 12 );
}
```

Now we add the draw method, which is the same on all operating systems.

```
void draw() {
  if ( cam.available()) {
     cam.read();
     image( cam, 0, 0, width, height );
   }
}
```

Now that you have written your first live videostream let's add some more features. At the moment it doesn't work like a mirror, because the image isn't reversed. This is really irritating when you watch yourself. So lets mirror the image on the Y-Axis. We scale the output by -1 which reverses it and change the start coordinate to -width.

```
void draw() {
  if ( cam.available()) {
     cam.read();

     pushMatrix();
     scale(-1,1);
     image( cam, -width, 0, width, height );
     popMatrix();

  }
}
```

IMAGE

Now we have a sketch that reacts like a mirror. Let's add some more features. The Capture or GSCapture objects react very similar to an image. We can even copy the content of the current frame to an image object. Add a variable of the type PImage in front of your setup method and make your sketch a bit bigger.

```
...
PImage img = null;
void setup() {
    size( 320, 320 );

...
```

Now we have a little extra space on the buttom of the window. We need to tell Processing that it shall not copy the camera frames to the whole window anymore but limit it to 240,320. Then we add another if to our draw method that draws the image we stored in the variable img if it is not null.

```
void draw() {
  if ( cam.available()) {
     cam.read();

     //image( cam, 0, 0, width, height );

     pushMatrix();
     scale(-1,1);
     image( cam, -width, 0, width, 240 );
     if ( img != null ) {
       image( img, -212, 240, 106, 80 );
     }
     popMatrix();
  }
}
```

Now add a litte mouseClicked method. When the user clicks the mouse we create a new image that has the same size as our camera frame and copy it's content.

```
void mouseClicked() {
  img = createImage(320,240,16);
  img.copy( cam, 0, 0, 320, 240, 0, 0, 320, 240 );
}
```

Video images can be used in various ways to let your user interact with your sketch. You can use it to detect movement for example, by comparing 2 frames of your video stream and test if the number of pixels that have changed in the two frames succeed a certain treshold.

Start With a new sketch and use the setupmethod from above, that's apropriate to your plattform. This time we use the image method to store the previous frame. So on a mac use the following setup.

IMAGE

```
import processing.video.*;

Capture cam;
PImage img = null;

void setup() {
    size( 320, 240 );
    cam = new Capture( this, 320, 240, 12 );
}
```

Or on linux or windows use this one

```
import codeanticode.gsvideo.*;

GCapture cam;
PImage img = null;

void setup() {
    size( 320, 240 );
    cam = new GSCapture(this, 320, 240, 12 );
}
```

In our draw method we use the pixels array of the image and the current frame and count all the pixels that are different. we tolerate a certain amount of change because camera images are noisy, so even if nothing moves, there are some pixels that are not the same. If the sum of the differences between the pixels of the current frame and the pixels of the last frame are > x then we print a message. We only take the green value of the image, which is very a quick and dirty method to change the images to grayscale.

```
void draw() {
  if ( cam.available()) {
    cam.read();

    cam.loadPixels();
    img.loadPixels();
```

```
    int diff = 0;
    for( int i = 0; i < 320 * 240; i++ ) {
      diff += abs( green(cam.pixels[i]) - green(img.pixels[i]));
    }
    if ( diff > 500000 ) {
      println( "Alarm " + diff );
    }

    image( cam, -0, 0, width, height );
    img.copy( cam, 0, 0, 320, 240, 0, 0, 320, 240 );
  }
}
```

IMAGE

By using the pixel arrays you can also access the color values of the pixels in your current frame. Let's make another sketch and copy the setup method again. In our draw method we draw an elipse on the mouse position using the same color the pixel unter the cursor has.

```
void draw() {
  if ( cam.available()) {
    cam.read();
    image( cam, 0, 0, width, height );
    cam.loadPixels();
    color c = cam.pixels[ mouseX + mouseY * cam.width];
    fill( c );
    ellipse( mouseX, mouseY, 20, 20 );
  }
}
```

As far as video goes we haven't even scratched the surface of the possibilities Processing offers yet. Play around with the video libraries and have a look at the Processingr forum to see some really cool examples of what can be acomplished in a view lines of code.

IMAGE

9.4. Hello Computer?

Another cool way to let your sketch interact with your user, is the use of audio. In the next session you will use the minim framework, to visualize prerecorded audio, and to react to sounds using microphones. The minim frameworks comes bundled with Processing and works on all plattforms. Used to play sounds in various formats as well as for sound generation and applying filters to audio material.

Start a new empty sketch and import the minim framework by clicking on the "Sketch" and selecting "Import library" and then clicking on minim. Now we need to add some samples to our sketch. Download the samples from the example site on TBD: url and use "Add File" in the sketch menue to add the boom.wav and the tschak.wav.

Now we define two AudioSamples before the setup method an one for Minim.

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

Minim minim;
AudioSample boom;
AudioSample tschak;
```

In the setup method initialize minim and use it to load the audiosamples you just have added to your sketch. If you do want to use your own samples, adjust the filenames in the loadSample method

```
void setup() {
  size(300,300);

  minim = new Minim( this );
  boom = loadSample( "boom.wav", 2048 );
  tschack = loadSample( "tschack.wav", 2048 );
}
```

The draw method stayes empty so far, but we add a mouseClicked method that triggers the first sample if the left mousebutton is clicked and the second one if the right mousebutton is clicked (CTRL-click if you are using a mac).

```
void draw() {
}

void mouseClicked() {
  if ( mouseButton == LEFT ) {
    boom.trigger();
  } else {
    tschack.trigger();
  }
}
```

Now we have to stop method. This method gets called when the Processing sketch is closed. There we close the audio samples and stop minim.

```
void stop() {
  boom.close();
  tschack.close();
  minim.stop();
  super.stop();
}
```

Click on the play button and use the mouse to trigger some samples. Be warned, if you have kids, it could take a while before you get your mouse back - trust me.

Now let's add some graphic to the sketch. Every AudioSample object has a so called mix buffer, containing the samples that are currently played. This buffer works similar to an array of float values. We can get the size of the buffer and we can access the float values by using the get Method.

Lets draw a rectangle for each sample we have and rotate it by the maximum float value our mix buffer contains. This makes the rectangles shake a bit when you trigger the corresponding sample. First we need to access the mix buffer and get the highest or lowest value to get the rotation angle. So add a method named getMax that takes a AudioSample as a parameter and returns a float value.

```
float getMax( AudioSample sample ) {
  float a = 0;
  for( int i = 0; i < sample.bufferSize() - 1; i++) {
    float temp = sample.mix.get(i);
    if (abs( temp ) > abs( a ) ) {
       a = temp;
    }
  }
  return a;
}
```

Now we can add the rectangles in the draw method. Let's start with the rectangle that shakes to the boom sample. First we use our friends pushMatrix and popMatrix again to create a new drawing context for the boom sample since our two boxes should shake independently. We draw the rectangle centered to the origin and move it to 80, 150 by using translate

```
void draw() {
  strokeWeight(3);
  fill( 255, 255, 0 );
  stroke( 255, 0, 0 );

  pushMatrix();
  translate( 80, 150 );
  rect( -50, -50, 100, 100 );
  popMatrix();
}
```

Now we can add the rotate part. We fetch the current maximum sample value using our getMax method and use rotate to make our little box shake da booooooooody - aehm sorry.

```
void draw() {
  float ab = getMax( boom ) * 30;

  strokeWeight(3);
  fill( 255, 255, 0 );
  stroke( 255, 0, 0 );
  pushMatrix();
  translate( 80, 150 );

  rotate( radians( ab ));
```

```
  rect( -50, -50, 100, 100 );
  popMatrix();
}
```

Next thing we do is adding another box for our tschack sample, because no sample should be without it's dancing box in this sketch. Copy the block from pushMatrix to popMatrix and fetch the angle to rotate it by using getMax on tschack.

```
  ...
  float at = getMax( tschack ) * 30;

  fill( 255, 255, 0 );
  stroke( 0, 255, 0 );
  pushMatrix();
  translate( 220, 150 );
  rotate( radians( at ));
  rect( -50, -50, 100, 100 );
  popMatrix();
  ...
```

TODO: fft

So far we have played audio and reacted to it, in the next sketch we are going to react to live audio that is recorded from the microphone. We will draw a small cloud that is moved across the window when someone blows into the microphone. Ok we are cheating a bit, because we react to any sound that comes, but it's a nice effect anyway.

So let's start with a draw and a setup method again and set the size to 500,300.

```
void setup() {
    size(500,300);
    smooth();
    frameRate(25);
}

void draw() {
}
```

Now add a method that draws our cloud.

```
void cloud( int x, int y, int w ) {
    int u = w/6;
    // this is a bit of a hack to prevent floating point rounding errors
    int u15 = int(u * 1.5);
    int u25 = int(u * 2.5);
    fill(255, 200);
```

```
    noStroke();
    rect( x - 4 * u , y-u15, 8 * u + 1, 3*u );

    stroke(0);
    strokeWeight(2);

    arc( x - 4 * u, y, 3 * u, 3 * u, PI/2 , PI * 1.5);
    arc( x + 4 * u, y, 3 * u, 3 * u, PI*1.5, PI * 2.5);

    arc( x - u15, y - u15, 5 * u, 5 * u, PI, 2*PI );
    arc( x + u25, y - u15, 3 * u, 3 * u, PI, 2*PI );

    line( x - 4 * u, y + u15, x + 4 * u, y + u15 );
}
```

To make Processing listen for audio we need to define an AudioInput object, so import the minim libraries again and define a variable for Minim and AudioInput. We also define an int variable that stores the current position of the cloud we are going to move across the window.

```
import ddf.minim.*;

Minim minim;
AudioInput in;
int pos = 0;

void setup() {
    size(500,300);
    smooth();
    frameRate(25);

    minim = new Minim( this );
    in = minim.getLineIn( Minim.MONO, 512 );
}
```

Ok, lets add some code to the draw method. First we are going to set the background to blue. Then we take the audio input buffer and search for the biggest value, similar to what we did in the boom/tschack sketch, and move the cloud by that value.

```
void draw() {
  background( 128, 128, 255 );
  float m = 0;
  for(int i = 0; i < in.bufferSize() - 1; i++) {
    if ( abs(in.mix.get(i)) > m ) {
      m = abs(in.mix.get(i));
    }
  }

  pos += m * 50;
  cloud( pos, 150, 80 );
  if ( pos > width ) {
    pos = 0;
  }
}
```

9.5. Midi-Events

Midi - short for Musical Instrument Digital Interface - is a protocoll that was defined in the 80s to controll musical instruments. In this section I will show you how to use Processing to send and receive midi data, but we won't discuss all aspects of the midi-protocol, since this would require a book on it's own.

Midi uses 16 channels and on each of them another sound could be controlled. On each of them we can send events like note on or off and controllerdata like change the volume of channel X. Now we will write a small xylophone that sends midi-data to a synthisizer. Don't worry if you don't have one at hand, java comes with a midi-player that can turn our note events to sounds.

To use midi in Processing we need to install the proMidi library which can be downloaded at http://.... Unzip it in the library folder of your sketchfolder and restart Processing. Now import the proMidi library by using the sketch->"Import Library ..." menu and define a setup and a draw method.

```
import promidi.*;

void setup() {
}

void draw() {
}
```

Now we define a variable of the type MidiIO above the setup method and use the getInstance method to initialize it. The midiIO object can be used to query all the midiinterfaces Processing has found by using printDevices();

```
import promidi.*;

MidiIO midiIO;

void setup() {
  midiIO = MidiIO.getInstance( this );
  midiIO.printDevices();
}

void draw() {
}
```

Now look at the output of the Processing sketch and search for the midi interface you want to use. Add a MidiOut variable above the setup method and use the getMidiOut method to open the interface and channel you want to use.

```
import promidi.*;

MidiIO midiIO;
MidiOut midiOut;
```

```
void setup() {
  midiIO = MidiIO.getInstance( this );
  midiIO.printDevices();

  midiOut = midiIO.getMidiOut(0,0);
}
```

Now we can use sendNote to send some midi-events. We will send a short note every time the mouse is clicked. Tell Processing to draw some lines in the draw method and add a mouseClicked method. There we will use the X-Axis if the current mouse position to determine the pitch of the note we will send. Add the following code to the setup method

```
 ...
  background(0);
  stroke( 255 );
  for( int i =0; i < 36; i++) {
    line( i * width/36, 0, i * width/36, height );
  }
 ...
```

Now add a mouse Clicked method that maps the mouseX to 3 octaves of midinotes starting with C1

```
void mouseClicked() {
  midiOut.sendNote( new Note( 48 + int(map(mouseX, 0, width, 0, 36)), 127, 100));
}
```

If you have a midi-synthisizer or a midi-controller you can also receive the midi-note signals that are sent from the synth. Add the openInput statement to the setup method to open one of the input interfaces and tell Processing on what channel to listen for midi events. These events trigger a callback method if a note-event is sent.

```
   ...
   midiIO.openInput(0,0);
   ...
```

Now we add a note-on method that gets called every time Processing receives a midi event. We can now use the pitch to change the color of the rectangles we draw. Define a variable named n to store the current played key and add a if block to the draw method that checks if one of the bars should change the color. In the noteOn method we change the variable everytime we receive a note on event.

```
void noteOn( Note note, int device, int midiChannel ) {
  if ( note.getVelocity() > 0 )  {
   fill(128);
```

```
  } else {
    fill(0);
  }
  if (note.getPitch() >= 48 && note.getPitch() < 84 ) {
    int x = (note.getPitch() - 48) * width/36;
    rect( x, 0, width/36, height );
  }
}
```

The velocity of a midinote is > 0 if the key is pressed and 0 if it is released. So we map the pitch of the midinote to the x coordinate of the rectangle and fill it with 0 if the key is released and with 128 if the key is pressed.

As already said in the introduction of this section midi has far more to offer than can be covered here. Have a look at the reference documentation of proMidi and look at the examples it provides to see how to handle controller data or playing midi sequences.

9.6. Joystics, Gamepads, etc

In this section you will learn how to use a joystick or gamepad with Processing. This adds a whole new range of devices to your arsenal of interaction tecnologies. To make use of game controllers we use the proControl library. Download the proControl at http://... and unzip it to your library folder.

Now connect your gamepad, joystick or whatever gamedevice you have and let's have a look if Processing is able to detect them. ProControll has a printDevices method, which is very convinient to check what devices Processing has detected, and how your computer names them. This may vary between different operating systems (or even different versions of the device driver).

So start a new sketch and import the proControl libraries, by using the "Sketch" -> "Import Library..." menu and add the following code to your setup method. The Draw method stays empty for now.

```
import procontroll.*;

ControllIO controll;

void setup() {
    size( 300, 300 );
    controll = ControllIO.getInstance( this );
    controll.printDevices();
}

void draw() {
}
```

When you click on the play button Processing should print a list of input devices. ProControll also lists inputdevices like keyboard and mouse, it's not limited to gamecontroler.

Since the number of buttons, sticks, and other input devices varies dramaticaly between the different gamecontrollers proControll also allows you to query the capabilities of a device.

Chooce the device you want to use in your sketch and asl controllIO for an instance. Then we can use printButtons, printSliders and printSticks to query the chapabilities of your gamepad.

The gamepad i'm using in this example is a PS3 compatible noname usb-gamepad. On my linux box it's called "Demo KeyBoard PS3/USB Corded Gamepad". Just replace that with the name of your gamepad.

```
import procontroll.*;

ControllIO controll;
ControllDevice pad;

void setup() {
    size( 300, 300 );
    controll = ControllIO.getInstance( this );
    controll.printDevices();

    pad = controll.getDevice( "Demo KeyBoard PS3/USB Corded Gamepad" );
    println("Buttons: " );
    pad.printButtons();

    println( "Sliders: " );
    pad.printSliders();

    println( "Sticks: " );
    pad.printSticks();
}

void draw() {
}
```

Processing print's varios buttons, sliders and 2 sticks, it found on my gamepad. Let's use those 2 sticks to move some small elipses in the window.

To use the sticks we need two more variables of the type ControllStick. Add the before the setup section.

```
...
ControllStick stick1;
ControllStick stick2;
...
```

In the setup method we instanciate the sticks by using the getStick method of the ControllDevice. The names we need are the one that printSticks has printed. Add the following lines of code to the setup method.

```
...
  stick1 = pad.getStick( "y x" );
  stick2 = pad.getStick( "rz z" );
...
```

Now we need to define float variables that hold the x and y coordinates for our ellipses.

```
...
float x1 = 150;
float y1 = 150;
float x2 = 150;
float y2 = 150;
...
```

Now we can use stick1 and stick2 to move some ellipses in the draw method. They provide a getX and getY method, that contains the angle the sticks are currently at. We simply add them to our current position. On some controllers (like the one i'm using here) the X-axis and the Y-axis are reversed. So we need to update the x-coordinate of the ellipse by using getY of the stick and vice versa.

```
void draw() {
  background( 255 );

  x1 += stick1.getY();
  y1 += stick1.getX();

  x2 += stick2.getY();
  y2 += stick2.getX();

  fill( 255, 0, 0 );
  ellipse( x1, y1, 10, 10 );

  fill( 0, 255, 0 );
  ellipse( x2, y2, 10, 10 );
}
```

To make use of the buttons, we define a ControllButton object for every button we want to react to. Add a ControllButton variable above the setup method.

```
    ...
    ControllButton buttonA;
    ...
```

In the setup method we call the getInstance method of the ControllDevice object and give it one of the names the printButtons method has listed.

```
    buttonA = pad.getButton( "A" );
```

Now we can add an if to the draw method to check ifthe button has been pressed. To make Processing switch the background color to black as long as button A is pressed, replace the background statement in the draw method with the following code.

```
...
if ( buttonA.pressed()) {
  background( 0 );
} else {
  background( 255 );
}
...
```

10. Installation

Processing can be downloaded from www.processing.org. There are versions for Linux, Windows and Macos X. just download the file that fits your operating system.

10.1. Installation on Macos X

Double click the dmg file if it doesn't open automatically and drag the Processing icon the the Application folder. To Start Processing go to the Application folder and double click the Processing icon.

10.2. Installation on Windows

There are two install Processing install packages for windows, one where java is already bundled and an expert package where java has to be already installed on your system.

Both come as a zip file. Simply unzip them at any directory you like. To start double click the processing.exe file.

10.3. Installation on Linux

The Linux install file comes as a tgz file. open a shell and change to the directory where you want Processing to be installed (I suggest your homefolder or a subdirectory in your home unless you know what you do and how to change filepermissions) and unpack the tgz file using

```
tar -zvxf processing-1.0.9.tgz
```

To start Processing change to the directory you just created and start the Processing shell script whith

```
./processing
```

or by opening the directory in your filemanager and doubleclicking the Processing file.