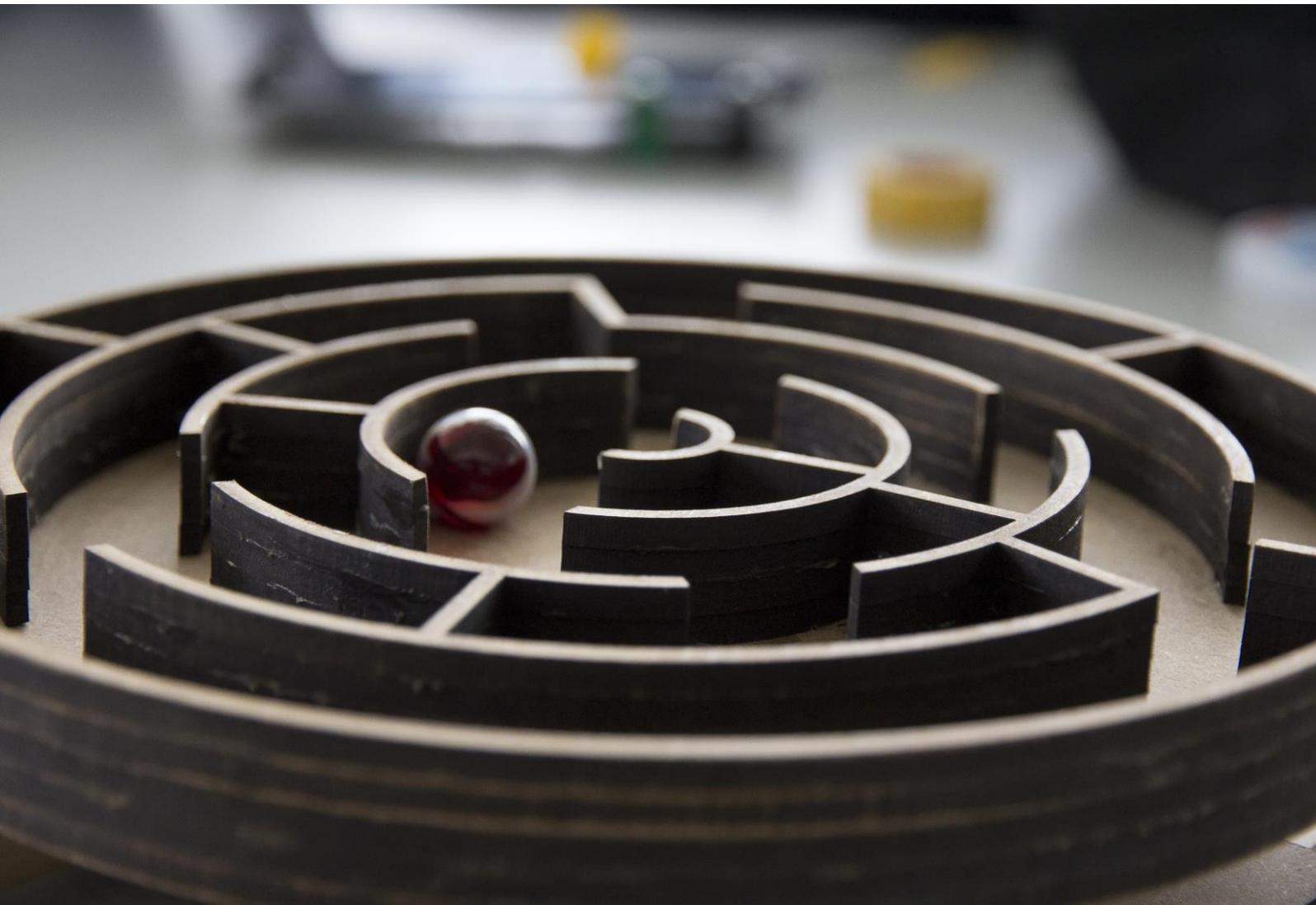


# Sense your Heart

---

**MODULE REPORT**

**BY ALEXANDER VAN DAM & RICK DE VISSER**



## Table of Content

Introduction.....	3
Idea generation.....	3
Concept .....	4
Implementation.....	4
PPG heart sensor.....	5
Servo motor .....	5
Processing.....	5
Sinus noise reduction.....	5
Prototyping .....	6
Appendix A: Coding.....	7
Appendix B: laser cutting drawing of the prototype.....	11

## Introduction

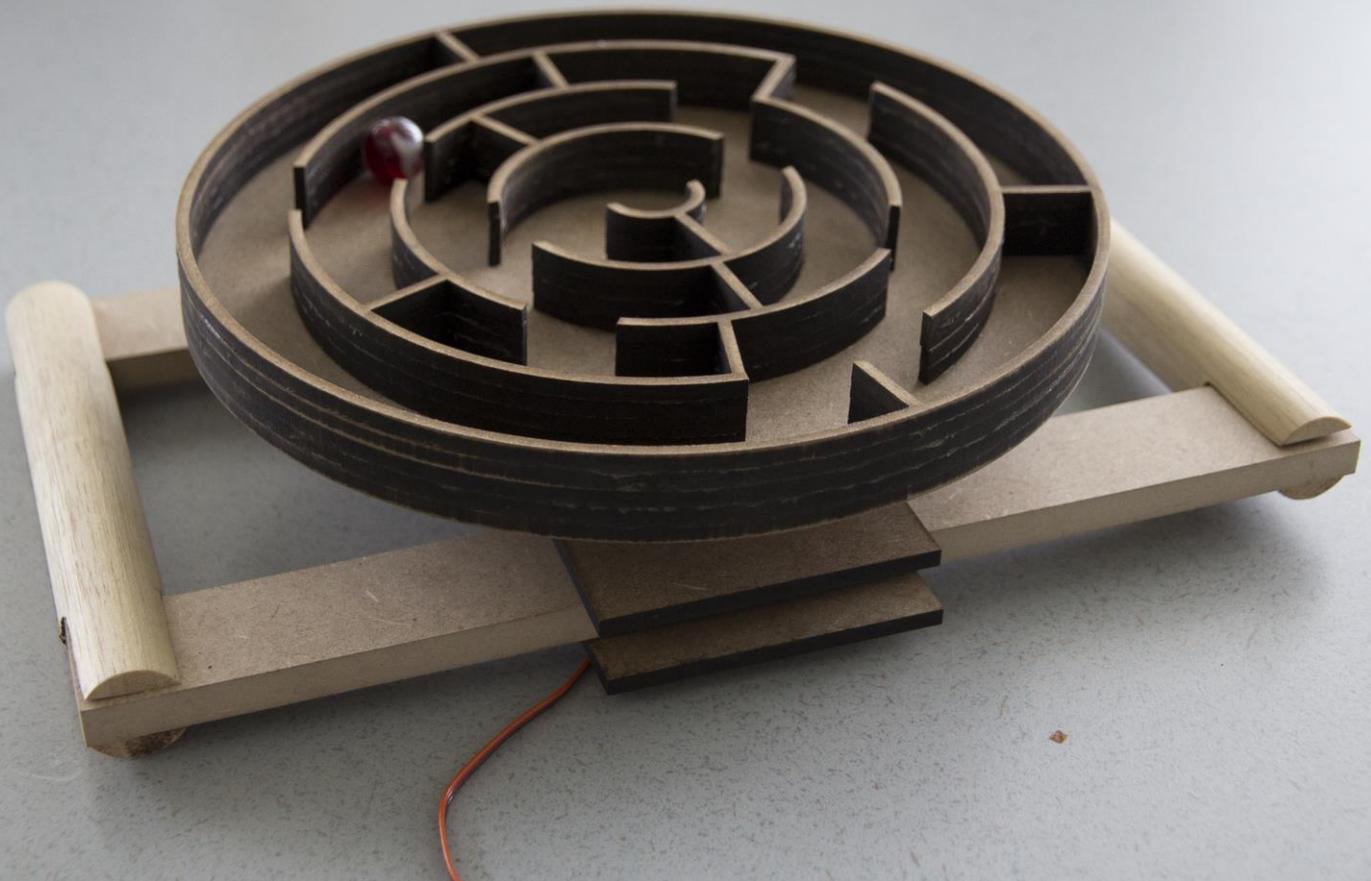
In this report we discuss our process of creating an application for a PPG heart sensor. The PPG heart sensor is based on an arduino board and self build as part of the module. From here one we did a short design iteration of two days in which we created a game as the application, made two working prototypes with a second arduino, tested them in game play and made some small adjustments to improve the game.

## Idea generation

The design assignment for this module was to design an application for our self-build heart sensor. We wanted the design concept to cover two aspects that we found interesting. Firstly, we focused on creating a physical concept for the more rich interaction opportunities that this will offer. Secondly, shooting games inspired us for an idea of integrating a PPG heart sensor application in a physical aiming device that you can use in the game. In a real-life situation the shooter has to control his state of mind and body to be calm and steady before he shoots. To much stress or excitement could make it much more difficult to aim straight. These states can be measured with our PPG heart sensor.

We took these two aspects to come up with an idea which we presented to Jun Hu. Our idea was to create a simple (wooden) prototype of a rifle with the PPG sensor integrated. In this prototype we would place a laser pointer attached to two small motors, one for the x-axis movement and one of the y-axis movement. The laser pointer would function as an aiming tool that you can point at a target. However, the system will give the laser a deviation depending on the Heart Rate Variability measurement. The lower the HRV, the bigger the movement of the motors and the bigger the deviation will be, which will make it harder if not impossible to hit your target.

The downfall of this system was that is was very hard to make it measurable whether the shooter would be able to hit his target in specific states of the Heart Rate Variability. This complexity of the system made us decide to change our concept towards another game in which aiming is still important and could also be influenced by your own HRV.

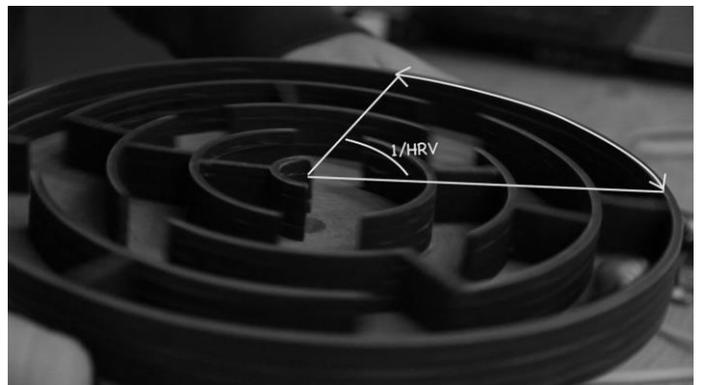


### Concept

We came up with the idea of a marble maze that can rotate from its middle axis automatically. A marble maze is an existing game in which a small marble is placed inside a handheld maze. The marble has to reach a specific point in the maze that is normally hard to reach. The player can only move the marble by tilting the whole maze towards the direction in which he wants the marble to go. This is a very delicate movement and is often experienced as very frustrating. We make this experience more intense, because frustration and agitation will be detected by the system and would make the game even more difficult to win.

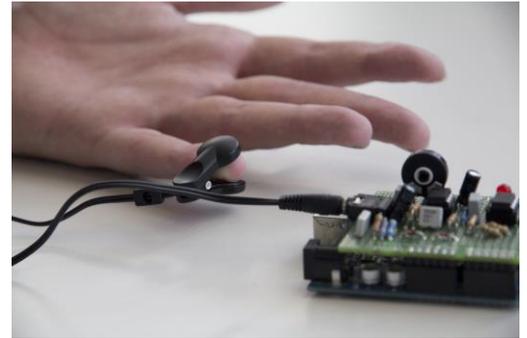
### Implementation

For the technical implementation of the concept two Arduino's were used, one as a heart rate sensor and one to drive the servo rotating the maze. It was chosen to do this using two Arduinos rather than one in order to avoid trouble with two interrupt driven devices on one Arduino and the small possibility of affecting the signal from the sensor. In order to process the data between the devices both were connected to a computer running processing.



### PPG heart sensor

The heart rate sensor was build using the PPG sensor shield template and code provided during the module and was left unaltered from this. In practice, when the sensor and the processing sketch are connected, the sensor sends the respiration rate through the com port to the processing sketch.



### Servo motor

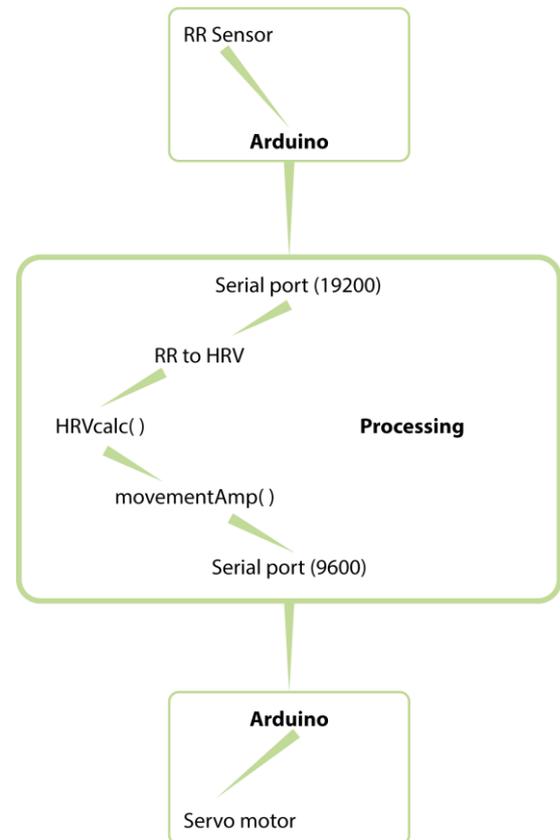
The second Arduino was directly connected to the servo. The software reads out serial data and sets the servo rotation to the position that is received from processing.

### Processing

Nearly all of the data processing is done within Processing by a modified version of the RR example that was provided. The main alterations consisted of adding support for a second Arduino as output and adding two new functions.

The first function (HRVcalc) takes the heart rate variability which is already calculated within the example and uses a map function in order to map the HVR values between 20 and 200 (boundaries estimated by observing the values of several test persons) to an amplitude of between 45 and 0 for use in the second function.

The second function (movementAmp) moves the servo back and forth around the 90° mark based on a sinus function with a fixed speed. The amplitude is dictated by HRVcalc and is thus in between 45° and 135° at maximum.



### Sinus noise reduction

After testing out the prototype it appeared that the servo occasionally behaved very erratically. While investigating it was found that it was likely due to the implementation of the sine function, which was created by putting in the amount of milliseconds since the sketch was running into `sin()`. At first it was tried to do this less often, but the result stayed the same. In the end it was decided to filter the spikes out using a low pass filter class found on the internet written for processing<sup>1</sup>. This helped reducing the results substantially. In hindsight another way of generating the sine wave should have been investigated, like the sinewave example on the processing examples page<sup>2</sup>.

<sup>1</sup> [http://jeremah.co.uk/blog/permalink/a\\_low\\_pass\\_filter\\_in\\_processing](http://jeremah.co.uk/blog/permalink/a_low_pass_filter_in_processing)

<sup>2</sup> <http://processing.org/examples/sinewave.html>

## Prototyping

The prototyping of the physical maze was done in two iterations, a cardboard prototype and the final prototype. The cardboard prototype was made by sticking a marble maze made out of foam core directly on top of a servo motor. The user had to hold the servo motor itself in order to move the maze. This prototype was used early in the process both as proof of concept and in order to test the code that was available so far, to see if the swaying of the maze would be too wild or not enough. This prototype made the early discovery of the random noise send to the servo possible, and allowed for enough time to find a solution. The final prototype was made using the laser cutter in order to get the accuracy and polish that was needed in order to make the maze feel smooth, something lacking from the cardboard prototype, were the marble would occasionally get stuck. The prototype was made by making a floor plate and five identical versions of the walls of the maze, together with two planks in order to hold the servo in place. The maze parts were glued on top of each other to form the top, and the servo was attached underneath. After this the final parts for the handle were made and assembled by hand. In the end the maze worked largely as intended, noticeably making the game more difficult for those who were not calm enough, while posing barely any resistance against those who were. The one thing that was not ideal was that the top was not connected to the servo well enough, the side to side movement was fine, but the maze had leeway to tilt more than intended when trying to move the marble.



## Appendix A: Coding

These is de code that we used for our final prototype. The module provided us we a standard code to work with, which included the measurements for the PPG sensor. Here we only show the parts of code that we have changed or added parts in.

### Processing code

```
import processing.serial.*;

float vPrev, lPrev, v; // Filter stuff
LowPass lp;

boolean RRDISPLAY=false;
boolean HRVDISPLAY=true;
boolean CIRCLEDISPLAY=false;

int prevtime = 0;
float sinval = 0;

RRport myport;
RRparser myparser;
Serial ARPort; // Create object from Serial class

RRdisplay rrdisplay;
HRVdisplay hrvdisplay;
CIRCLEdisplay circledisplay;

PrintWriter output;
PFont myFont;
boolean settedup=false;

void setup()
{
  int bars=75;
  frameRate(10);
  //myFont = createFont("TimesNewRomanPSMT",16);
  //textFont(myFont);
  size(5*bars,5*bars);
  myport = new RRport(this);
  myparser = new RRparser();
  rrdisplay = new RRdisplay();
  hrvdisplay = new HRVdisplay();
  circledisplay = new CIRCLEdisplay();

  lp = new LowPass(4); // Create the lowpass filter

  SimpleDateFormat sdf = new SimpleDateFormat("-yyyy-MM-dd/HH-mm");
  Date now = new Date();
  //println("log to logfiles"+sdf.format(now));
  output = createWriter("logfiles"+sdf.format(now)+".txt");

  String portName = Serial.list()[2];
  ARPort = new Serial(this, portName, 9600);

  fill(10,10,10);
  rect(0,0,width,height);
  settedup=true;
}

int RRavg = 1200;
```

```

int RRstd = 50;

int time = 0;
int count =0;

void draw() {
    count++;
    count=count%10;
    if (count==9) output.flush();
    time++;
    movementAmp(HRVcalc());
}

void serialEvent(Serial p) {
    if (settedup) {
        //print("!");
        //zat allemaal eerst in draw
        myport.step();
        myparser.step();
        //if (time/10 > 120) stop();
        if (myparser.event()) {
            int RR = myparser.val;
            //print(" RR=");
            //print(RR);
            output.println(RR);
            RRavg = int((23 * RRavg + 1*RR)/24);
            //print(" AVG=");
            //print(RRavg);
            RRstd = int((15.0 * RRstd + abs(RR - RRavg)) / 16.0);
            //print(" STD=");
            //print(RRstd);//niet met kwadraat ivm outliers
            //println();
            rrdisplay.next(RR);
            hrvdisplay.next(RRstd);
            circledisplay.next(RRavg,RR);
            if (RRDISPLAY==true) rrdisplay.step();
            if (HRVDISPLAY==true) hrvdisplay.step();
            if (CIRCLEDISPLAY==true) circledisplay.step();

            //float BTopt = 15.00; //assumed resonant breathing rate in seconds
            //int advice = int(BTopt*1000/RRavg); //idem in heart beats
            //fill(30); stroke(0); rect(0,height-16,16,2*16);
            //fill(150); text(Integer.toString(advice),0, height);

        }
    }
}

public void stop() {
    output.flush(); // Writes the remaining data to the file
    output.close(); // Finishes the file
    //println("THANK YOU, GOODBYE");
    exit(); // Stops the program
}

void keyPressed() {

public int HRVcalc() {
    int result;
    result = (int) map(RRstd, 20, 200, 45, 0);
}
}

```

```

// Result between 0 & 45!
return result;
}

public void movementAmp(int amp) {
    int midle = 90;
    int time = millis();
    int delaytime = 10;
    int rotval = 0;

    if (time-prevtime > delaytime)
    {
        sinval = time;
        prevtime = time;
        // println(sinval);
        rotval = (int) map(sin(sinval), -1, 1, midle-amp, midle+amp);
        lp.input(rotval);
        ARPort.write((int)lp.output); // Send angle to the servo
        println ("/*rotval + ", " + */lp.output);
    }
}

class LowPass {
    ArrayList buffer;
    int len;
    float output;

    LowPass(int len) {
        this.len = len;
        buffer = new ArrayList(len);
        for(int i = 0; i < len; i++) {
            buffer.add(new Float(0.0));
        }
    }

    void input(float v) {
        buffer.add(new Float(v));
        buffer.remove(0);

        float sum = 0;
        for(int i=0; i<buffer.size(); i++) {
            Float fv = (Float)buffer.get(i);
            sum += fv.floatValue();
        }
        output = sum / buffer.size();
    }
}

```

### *Arduino servo code*

```
#include <Servo.h>

Servo myservo1;
int pos1 = 0;
int incomingVal = 0;

void setup()
{
  myservo1.attach(2);
  myservo1.write(0);
  Serial.begin(9600);
  Serial.print("Ready");
}

void loop()
{
  if ( Serial.available() ) {
    incomingVal = Serial.read();
    //Serial.print("I received: ");
    //Serial.println(incomingByte, DEC);
    myservo1.write(incomingVal);
  }
}
```

## Appendix B: laser cutting drawing of the prototype

scale 1:3

