CodeDojo
TNMOC Bletchley Park
26 May 2019

Processing p5.js

For Computational Art and Graphics

P5.js is a javascript library which brings the power of Processing for artists and designers to the modern webpage. With this you can think of the webpage as your canvas and unleash creativity through the Processing library in order to integrate art, text, drawing, images, video, animation and sound.

Web Editor for p5.js

We are going to use the p5 web editor to write our programs to make art.

Go to : editor.p5js.org and Click "sign up" at the top right hand corner. If you already have an account, please sign in. If not, enter an username, email and password to create a new account. If you do not want to save your work, you do not need to get the account. The editor still works.

Sketch setup

Once logged in, let us start to code to make our first artwork. But, first it is important to know that each sketch consists of at least two functions called setup( ) and draw( ). The statements in the setup( ) function execute once when the program begins. This means any initialisation work or start up effort must be performed inside this function. The statements in draw( ) are executed when play is clicked until stopped. The program executes in sequence. When the last line is executed it is followed by the first line and so on repeatedly. This is a very

powerful feature as it allows us to do things repeatedly with minor changes that in effect gives life to the objects as well as logic inside the program.

At startup, you will see:

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
}
```

createcanvas( ) is a function that creates the workspace in which we shall draw. background( ) function sets the colour of the background. In order to create our first art, we need to know the coordinates on the canvas. The top left is [0,0]. The canvas is "width" pixels wide left to right and "height" pixels tall top to bottom. The pixels start from 0 and goes all the way to width-1. The coordinate of bottom right of the page is [width-1,height-1].

Pen on Canvas

Now let us draw some lines. stroke( ) is a function that tells the program what colour is the pen going to be. The parameters are the amount of Red,Green and Blue in the final colour. For example, Stroke(0,0,255) represents blue. Draw a line with that pen using the line( ) function where the arguments are [x1,y1] and [x2,y2], the two points that represent the start and end points of the line. By changing the coordinates and stroke values we can draw lines at different places and with different colours. Try a program that looks like this and see how it shows up on the canvas.

```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);

  stroke(255,0,0);
  line(0,height*0.5,width,height*0.25);

  stroke(0,255,0);
  line(0,height*0.5-10,width,height*0.25-10);

  stroke(0,0,255);
  line(0,height*0.5+20,width,height*0.25+20);
}
```

code editor

Basic Animation

Now that we learnt to draw a line, why not try to animate a shape and code some movement. Let us draw a square and make it float up in the air. We want to make it stop floating up when mouse is pressed and let it continue to move up again if mouse is released. With that goal, let us create a variable called y which will be used to store the position of the square. In the Setup( ) part of the program let us create a canvas, set pen to white and set the starting position of the square to be the middle of the canvas. The code should look like this:

```
let y;

function setup() {
  createCanvas(720, 400);
  stroke(255);
  y = height * 0.5;
}
```

Next we need to write the logic for animating the square inside the draw( ) function. Every time draw( ) renders the square we want it to have moved up by a pixel. After setting the background to black, we add the logic for reducing the height coordinate of the square object. Then, we check if reduction of y results in its value going below zero. In that case we need to set it to "height" so that the square re-appears from the bottom of the canvas. We draw the square with the square( ) function whose arguments are the [x,y] coordinates of the midpoint of the square and its size. Once this is written, the code should look like this:

```
function draw() {
  background(0); // Set the background to black

  y = y - 1;

  if (y < 0) {
    y = height;
  }

  square(width/2, y, 15);
}
```

loop( ) and noLoop( ) are functions that tell the program whether to run the draw( ) again and again. noLoop( ) function will make draw( ) run once and prevents looping. loop( ) function reverses this and allows continuous multiple execution of each line inside draw( ). In order to link this loop / noLoop logic to the mouse press and release, we write two very small

functions that tells the computer when to draw( ) and when to stop. With all these code snippets added together, the final code should be as follows:

```
let y;

function setup() {
  createCanvas(720, 400);
  stroke(255);
  y = height * 0.5;
}

function draw() {
  background(0); // Set the background to black

  y = y - 1;

  if (y < 0) {
    y = height;
  }

  square(width/2, y, 15);
}

function mousePressed() {
  noLoop();
}

function mouseReleased() {
  loop();
}
```

Self-Similarity and Recursion

Next, let us try to draw a more complicated pattern. This involves the concept of recursion which can be seen in nature by way of fractal patterns. Let us initialise the setup with createCanvas( ), noLoop( ) and noStroke( ).

```
function setup() {
  createCanvas(720, 400);
  noStroke();
  noLoop();
}
```

The draw( ) contains only one line which has the function drawCircle( ) that we have to define. The function has three arguments, the first of which is the x-coordinate of the centre of the first circle. We will draw the circle in circle pattern half-way in terms of height. This means that the y-coordinate for all circles will be "height / 2". The second argument is the radius of the circle. If we have to draw circles inside circles, we need to have a logic in our program to specify how many levels deep the pattern should be, each level displaying circles that have smaller radii. This will be specified with the third argument "level".

```
function draw() {
  drawCircle(width / 2, 200, 4);
}
```

We start the definition of the recursive function with a const "tt" that has different values dependent on which level circle is being drawn. This parameter determines the fill colour of the circle. In Processing it is possible to define a circle using "ellipse" function with the coordinates for the centre and height & width parameters being same. Once the main circle is drawn, the value of level is reduced by 1 and two new smaller circles are drawn by invoking the drawCircle( ) function twice with different centres, smaller level and smaller radii. The self reflecting aspect of the recursion is implemented this way.

```
function drawCircle(x, radius, level) {
  const tt = (126 * level) / 4.0;
  fill(tt);
  ellipse(x, height / 2, radius * 2, radius * 2);
  if (level > 1) {
    level = level - 1;
    drawCircle(x - radius / 2, radius / 2, level);
    drawCircle(x + radius / 2, radius / 2, level);
  }
}
```

The final code should look like this.

```
function setup() {
  createCanvas(720, 400);
  noStroke();
  noLoop();
}

function draw() {
  drawCircle(width / 2, 200, 4);
}

function drawCircle(x, radius, level) {
  const tt = (126 * level) / 4.0;
  fill(tt);
  ellipse(x, height / 2, radius * 2, radius * 2);
  if (level > 1) {
    level = level - 1;
    drawCircle(x - radius / 2, radius / 2, level);
    drawCircle(x + radius / 2, radius / 2, level);
  }
}
```