**Processing****: A flexible software sketchbook**

Processing is a flexible software sketchbook and a language for learning how to code within the context of the visual arts.

Some great advantages of Processing are:

-It is completely open source and free to use

-It has OpenGL integration that allows expandable scope for your projects

-Its interactive outputs make working on projects quite simple, and it also had a wide range of outputs

-it has over 100 different libraries that can be used in your projects

-It is platform independent

With these advantages, Processing can easily be the first step for someone that has no prior experience working with Graphical Designing and with its easily expandable scope even veterans in the industry of Graphical Designing can quickly put their thought in code form using Processing.

**Getting Started**

To get stared we must first download the software from the official website which is https://processing.org/download, and download the setup file for your preferred operating system.

Processing is available for Windows, Mac and Linux, and we shall go over on how to install them on each operating system.

Device Specific Installation instructions:

* On Windows you will be downloading a .zip file, extract it your preferred location on your hard disk. Once extracted, simply enter the folder and run processing.exe.

-On Mac you will be downloading a .zip file, extract it your preferred location

on your hard disk. Once extracted, simply enter the folder and run processing.app.

* On Linux you will be downloading a .tar.gz file. To extract it simply open a terminal and type

tar xvfz {file\_name}.tgz

this creates a folder named processing-xxx[xxxx is the version] , open terminal, change directory to the extracted folder [processing-xxxx] and run processing

$> cd processing-xxxx $> ./processing

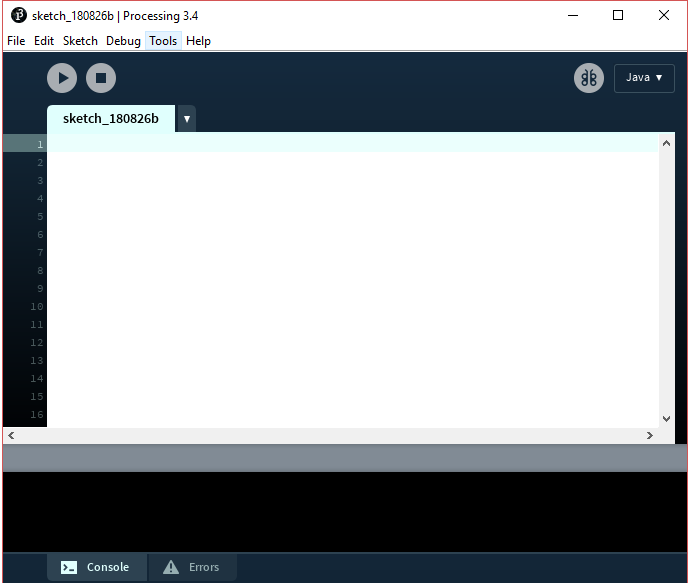


Figure 1: Sketch Board of Processing 3.4[1.png]

Once you done with your installation and if you have no trouble in the previous step, then you should be seeing the screen as shown in Figure 1. In-case if you have any trouble, please visit the wiki section in the oﬃcial website.

<https://github.com/processing/processing/wiki>

**Setting up the workspace**

The default language to use with Processing is Java, however in this guide we will be using Python, as it is more user friendly.

To change the programming language, use the drop-down menu at the top right corner to select the mode and select “Add Mode”

This will pop up a window, now as shown in Figure 2 navigate to the “Modes” tab and select “Python Mode for Processing 3”. If you wish to use other programming language like js, you can also select your preferred mode. Once selected, click on “Install” on the bottom right corner as shown in Figure 3 to install the python mode for Processing 3

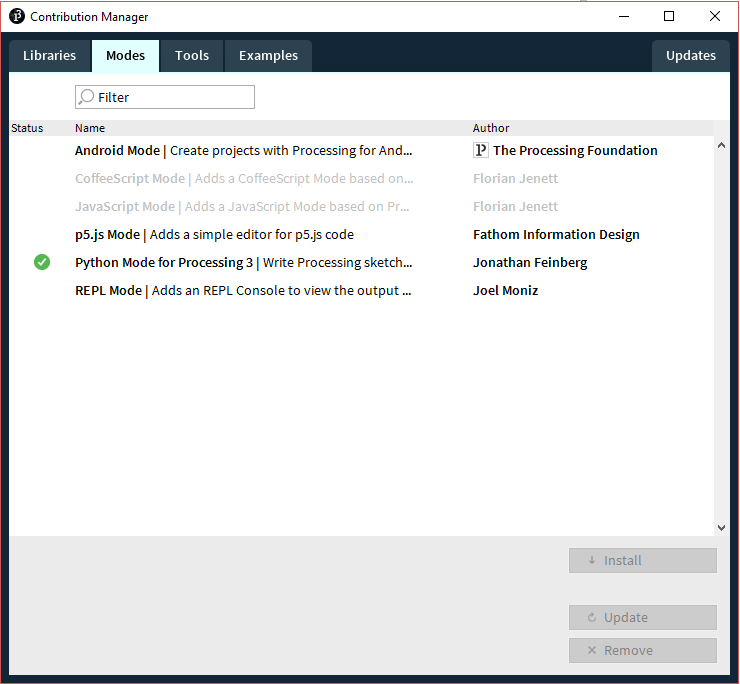


Figure 2: Python Mode [2.png]

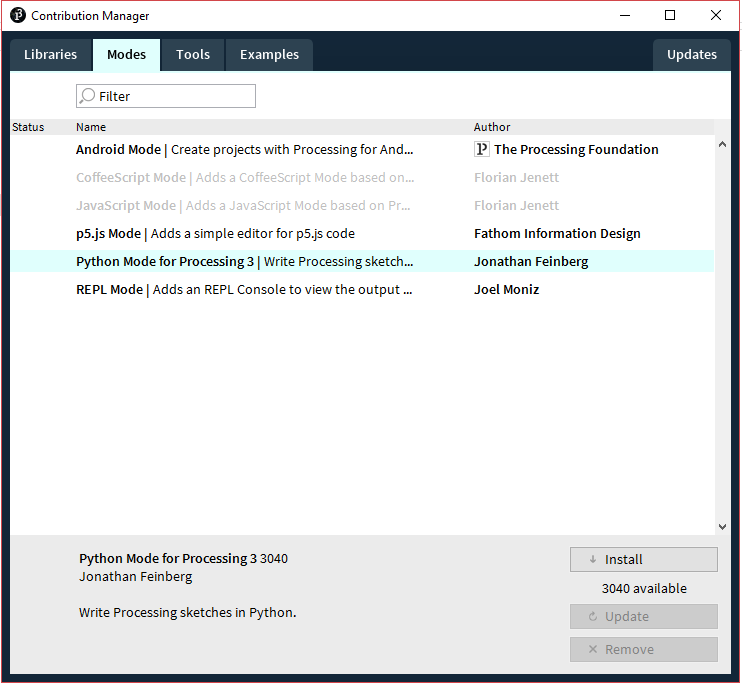


Figure 3: Installation Button [3.png]

**Basic Window**

Now we are done with setup process, let’s start coding our first graphics code. As a beginner, to kick start, we create a simple canvas window of size 800x600

In the editor, type the following code:

----------Code Start------------------

def setup():

#this defines your window size, here width is 800, height is 600 size(800, 600)

#the color of your background, you can use a value from 0-255 (black to white)

background(0)

#displays your display’s width and height on the print(displayWidth, displayHeight)

#displays your window’s width and height on the print(width, height)

----------Code End------------------

NOTE: Be cautious about indentation, as python implements code blocks only through indentation.

Once you are done your editor will be looking be something like Figure 4, hit CTRL + R or press the play button to run your code.

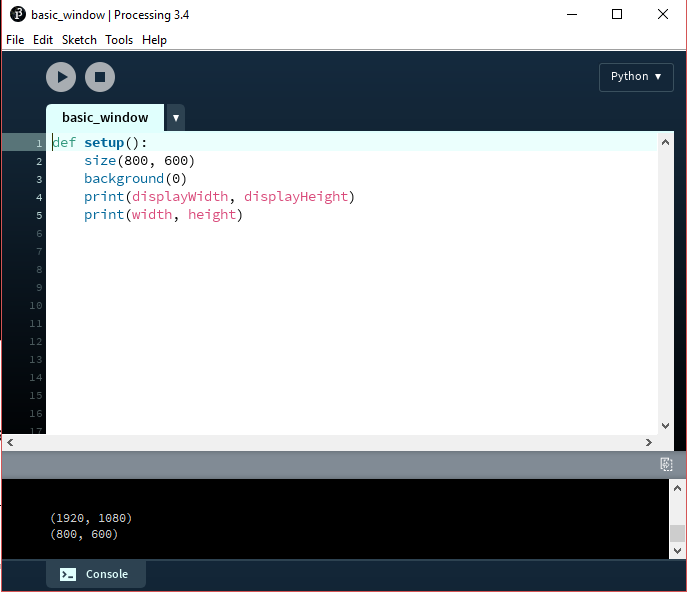


Figure 4: Editor view [4.png]

If you got the blank screen as show in Figure 5, Congratulations! You completed your first graphics code in canvas creation.

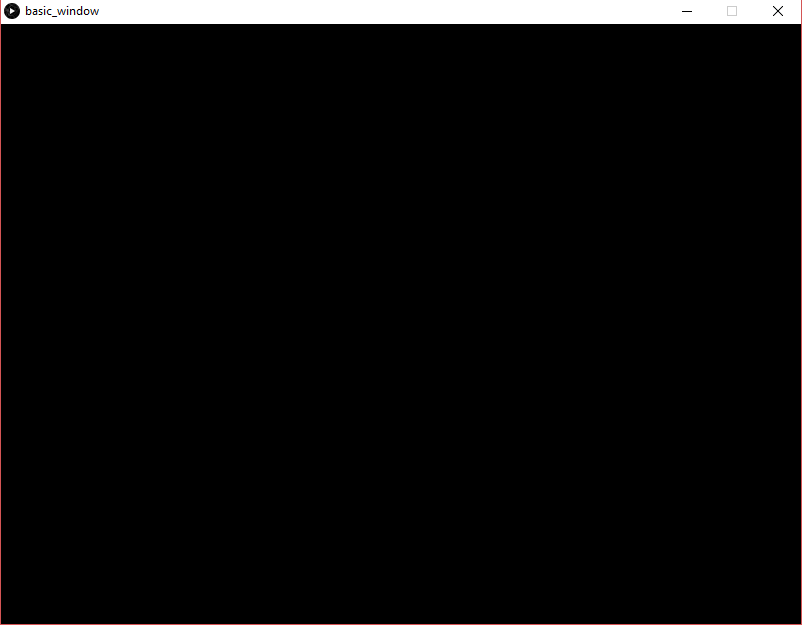


Figure 5: Output Window[5.png]

**Theory Intermission**

Before we move on to the next part of our guide, we need to understand a little theory of how everything comes together and works.

So, we currently know how to create a simple canvas window with nothing drawn in it by using the setup function. And this is where we shall begin our theory class, there are a few points to keep note about setup():

-The function runs only once when the program starts

-It is used to define the initial environment, i.e., the window and other parameters relating to it

-There MUST be only one setup function, and it must NOT be called in any other function

-The first line of the setup function must always define the size of the window (using size() ) if the user desires a window other than the default window.

-variables that are declared in the setup function are not accessible to other function in the same program, for python users, this can be overcome by using the “global” keywords whilst declaring the variable.

Next, we need to discuss about the created window. We know that the window size is user dependent, but to achieve total control of the canvas we need to know the position of the origin as it is very essential to understand our reference origin point while drawing any sort of graphics

Origin(0,0)

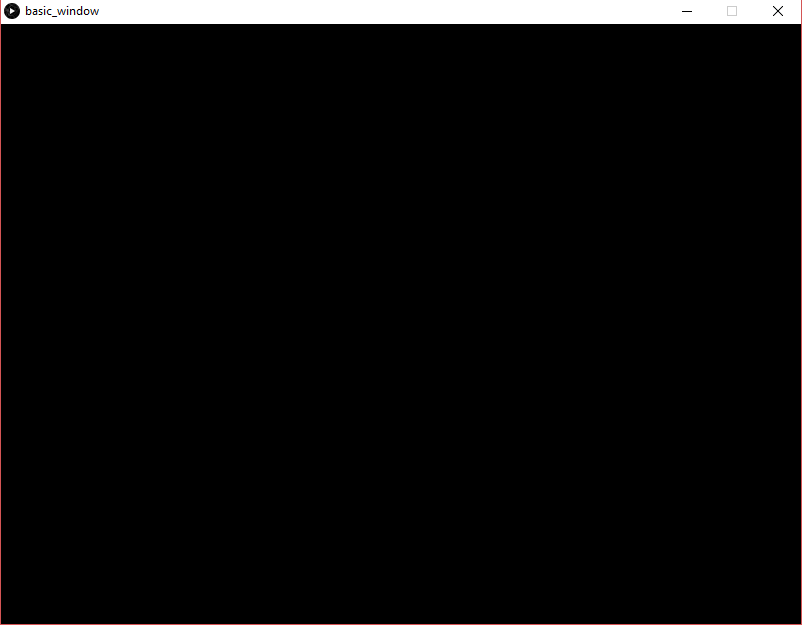
(Width, Height)

Figure 6: Coordinate System of Window [5.png]

As shown in figure 6, the Origin is in the top left corner of the window and this the reference point. On the other end, the bottom right corner of the window is the full size(maxWidth, maxHeight) of the window that you defined in the size() in the setup()

**Adding a Viewport**

The next thing that we are going to learn is a viewport and the importance of it. A viewport is a viewing region (area of visibility) in which the object can get rendered. A viewport is important due to the fact that a single screen can have multiple viewports each with diﬀerent dimensions and diﬀerent object which can get rendered in each viewport, allowing the user to have more control on your graphics.

To make a viewport that fills the entire screen, type in the following code into the editor and hit run.

----------Code Start------------------

def setup():

global viewport

size(800, 600)

#creating a viewport of the same dimensions as the screen

#achieved using the function createGraphics()

viewport = createGraphics(width, height)

def draw():

#begins drawing the viewport

viewport.beginDraw()

#sets the background color of the viewport

viewport.background(100)

#your models go here

#end the viewport drawing

viewport.endDraw()

#places the viewport on the screen at 0,0

image(viewport, 0, 0)

----------Code End-------------------



Figure 7: Basic Viewport [6.png]

If your code executed properly then your output window will resemble Figure 7, and you have now achieved the second milestone.

As you may have noticed, we have used a second function named draw(), this function is used after the setup() and it is called in a continuous loop. Thus the code in the draw function gets executed repeatedly, drawing on the screen and creating an impression of smooth animation, whereas in reality the objects are just being redrawn multiple time each second.

**Drawing Primitive on the screen**

Now that we have understood the creation of a screen, and the went on to learn the creation of a viewport, we can now concentrate on drawing our models. In our case we shall go will something, a basic Primitive, which are basic shapes such as point, line, rectangles, etc.

We shall divide this task in to two parts, the first half we shall discuss on how to draw directly on the screen without using a viewport

----------Code Start------------------

def setup():

size(800, 600)

def draw():

#setting the background color (black)

background(0)

# setting the fill color (white) of the ellipse

fill(255)

#drawing an ellipse of radius 60 & 60 at mouse position

ellipse(mouseX, mouseY, 60, 60)

----------Code End-------------------

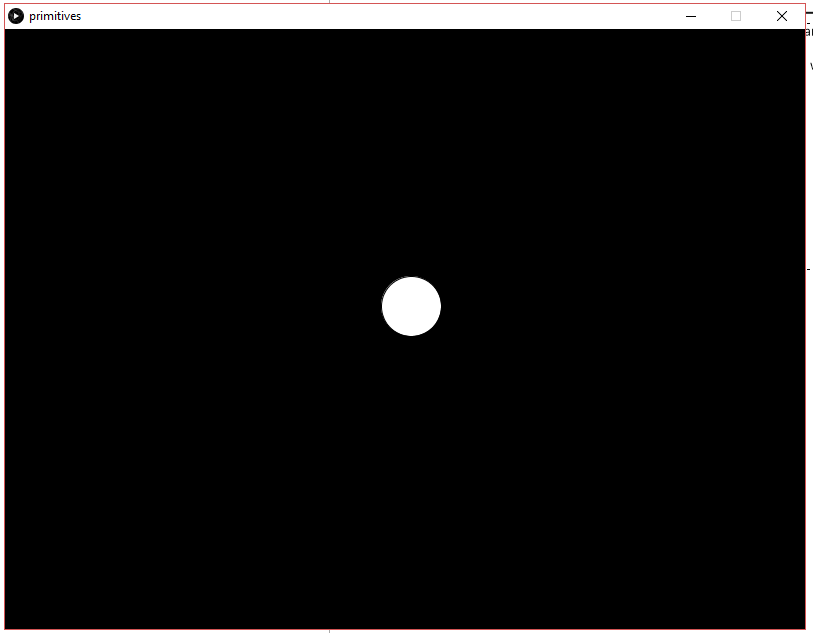


Figure 8:Drawing Primitive on Screen[7.png]

When you execute the above code, you will see an ellipse just like in Figure 8 that will move with you mouse cursor, this is due to the face that we have specified that the centre of the ellipse is always at the position of the cursor.

For more inbuilt functions that can be used to draw primitives, visit the official processing website and head over to the Reference’s 2D Primitives section, <https://processing.org/reference/>. While going through the inbuilt functions, take care to note exactly what coordinates/measurements are required to draw each shape, as it differs between shapes.

**Drawing Primitive on the Viewport**

Now that the first part of our task is complete, we shall move on to the second part that is drawing the same primitive on a viewport

----------Code Start------------------

def setup():

global viewport

size(800, 600)

#making the viewport 100 units smaller than screen in both dimensions viewport = createGraphics(width-100, height-100)

def draw():

#setting the background color of the screen

background(0)

#starting the viewport drawing

viewport.beginDraw()

#setting the viewport background color

viewport.background(100)

#setting the fill color of ellipse to be white

fill(255)

#drawing the ellipse in the viewport viewport.ellipse(mouseX-50, mouseY-50, 50, 50) #ending the viewport drawing

viewport.endDraw()

#placing the viewport at 50, 50

image(viewport, 50, 50)

----------Code End-------------------

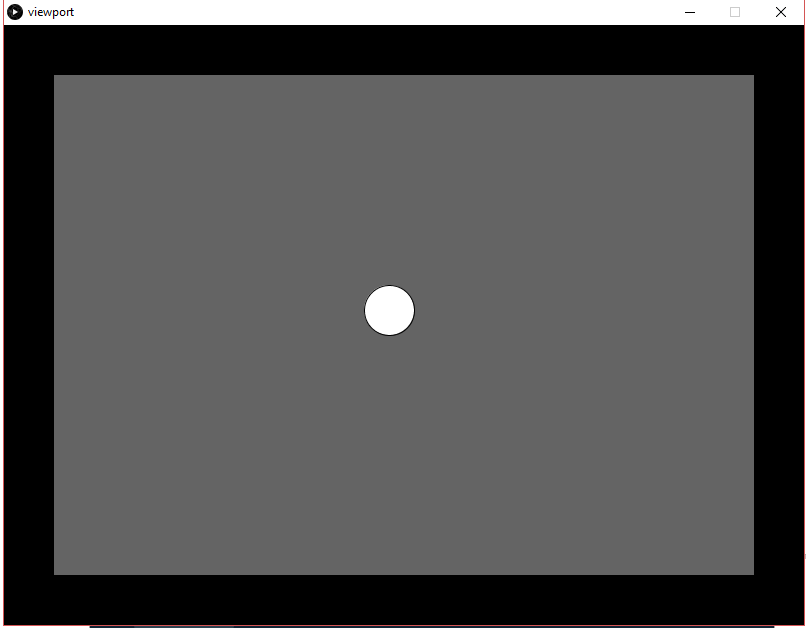


Figure 9: Primitive along with a Viewport[8.png]

When the code is executed, you will see that the output is like Figure 9. The black border is the screen behind the viewport, and when you execute the code you will notice that your ellipse will not be able to go in to the black area because you are only drawing it in the viewport and not on the screen (canvas).

NOTE: we subtract 50 from mouse coordinates while drawing on the viewport to avoid any misalignment caused due to the shifted viewport.

**Drawing a Stick man on the screen**

With our current knowledge, we can now try to make something a bit more complicated. The following code is for drawing a stick man on the screen using the basic primitives.

----------Code Start------------------

def setup():

size(800, 600)

def draw():

background(0)

#head

fill(0,15)

ellipse(400, 200, 60, 60)

#body

line(400, 200, 400, 500)

#left hand

line(400, 300, 300, 400)

#right hand

line(400, 300, 500, 400)

#left leg

line(400, 500, 300, 600)

#right leg

line(400, 500, 500, 600)

----------Code End-------------------

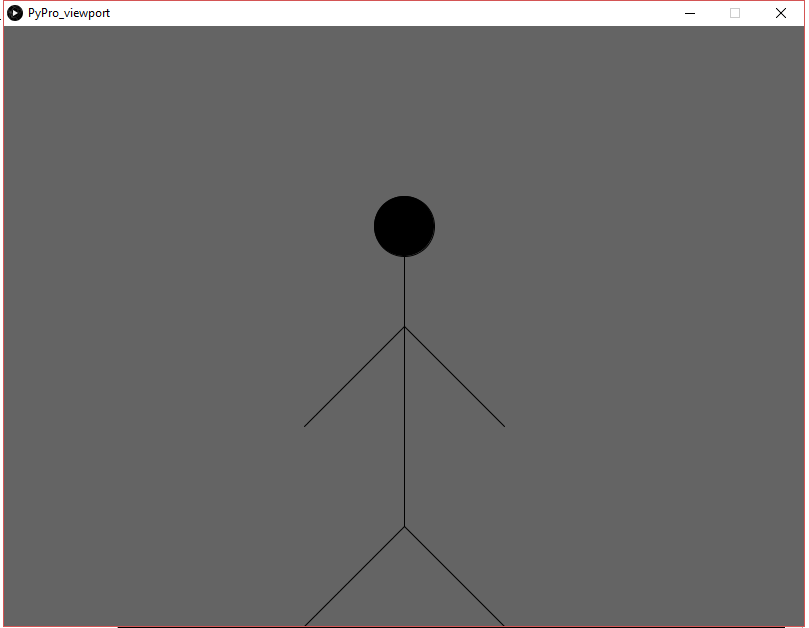


Figure 10: Stick Man[9.png]

As you will be seeing, when you executed your code your output window will be like Figure 10 and the entire figure is solely consisted of basic primitives. Thus in this manner we can build up more complicated shapes using nothing but the basic primitives that we have already learnt

**Simple Block Jumper Game**

Now that we have grasped the concepts of drawing primitives, we can try something a little more exciting. We are going to try to make a simple block jumper game using the knowledge that we already have along with the knowledge of using classes in python

NOTE: If you are unsure about using OOPs concepts in python, you should take this time to get familiar with at least the basic concepts as we will **NOT** be covering the how-to’s of python in this tutorial

**Basic Setup**

To start off our game we must first decide on the general layout of the screen, and for this case we shall be working by directly drawing on the screen. And another thing to take into consideration is the dimension of our screen, here since we are making a block jumper a rectangular screen will suit our needs better.

With that decided we must now think of how the player character will look like, for simplicity we shall be going with a simple circle created using the ellipse (). And a filled rectangle will do the job to give our game some semblance of the ground portion

----------Code Start------------------

def setup():

global x, y, diameter

size(800, 300)

diameter = 30

x = 50

y = (200-(diameter/2))

def draw():

background(255)

fill(50)

rect(0, 200, 800, 300)

fill(255)

ellipse(x, y, diameter, diameter)

----------Code End-------------------

The above code block implements the above concept and when you run it you will see a screen as shown in Figure 11



Figure 11: Basic Setup[10.png]

**Making Obstacles**

Now that we have achieved creating the basic setup, we can now shift our thought to creating the obstacles and moving. For this we shall we will be using a class called obstacle, as this will allow us to create multiple objects and use their methods easily

----------Code Start------------------

import random

class obstacle:

def \_\_init\_\_(self, x, y, w, h):

self.x = x

self.y = y

self.w = w

self.h = h

def draw\_ob(self):

fill(60)

rect(self.x, self.y, self.w, self.h)

def move(self):

global speed

self.x -= speed

if self.x <= 0 :

self.x=780

self.h = random.randrange(20, 50)

self.y = 200 - self.h

----------Code End-------------------

The above class makes it possible to create multiple obstacles and as you can see it also contains the methods to draw it and move the obstacles.

As you can see the draw\_ob() draw the obstacles, which are rectangles with the cords that are defined previously. The move() works by reducing the x coordinates of the obstacle, by adding the speed variable that is initialized in the setup(), till reaches the left most end of the screen, where it then resets it’s x coordinates back to the right most end of the screen.

Now with the class defined we can now move on to creating the objects and then calling the draw\_ob() and the move() functions.

Now to achieve the former we just use a simple loop to create 4 obstacles in the setup() as shown:

----------Code Start------------------

obs = []

x\_val = 780

y\_val = 180

ht = 20

wdt = 20

for i in range(4):

ob = obstacle(x\_val, y\_val, wdt, ht)

x\_val += 200

ht = random.randrange(20, 50)

y\_val = 200 - ht

obs.append(ob)

----------Code End-------------------

As shown in the code segment, we create object ob while varying the coordinates and the height of the obstacles and then append them to the list obs, creating a list of 4 objects

To draw the objects and then update the coordinates to achieve movement, we iterate over the newly created list obs and call the methods draw\_ob() and move() for each of the objects in the draw(), as shown in the code segment

----------Code Start------------------

for ob in obs:

ob.draw\_ob()

ob.move()

----------Code End-------------------

NOTE: You will need to declare the variable obs as global whenever you use it

If you have followed along correctly, when executed you will be getting an output like Figure 12

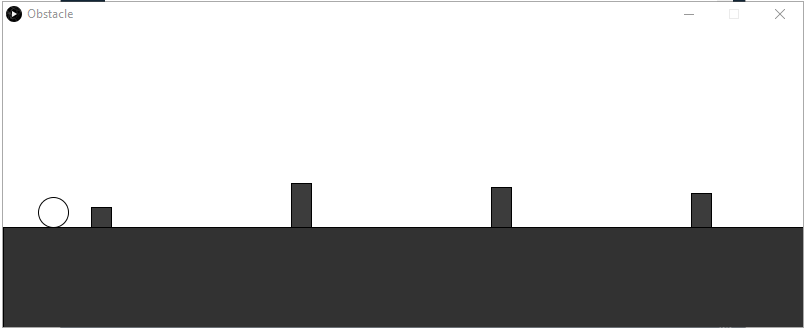


Figure 12: Creating the obstacles[11.png]

**NOTE:** If you are not able to get the desired output, worry not at the end of the tutorial, the entire code of the game will be given. Refer to it if you face any problems.

**Collison And How to Avoid Them**

As you may have noticed while executing the previous code, we cannot control our player circle and thus the player goes behind our obstacles as shown in Figure 13

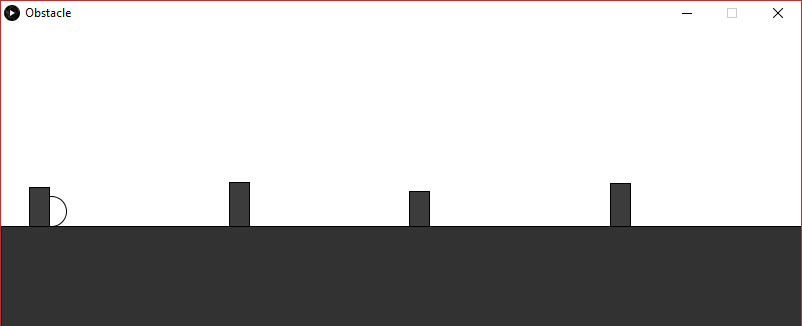
****

Figure 13: Overlapping of obstacles and player[12.png]

To fix this issue we must include some conditions to check for collision between the player and the obstacles, we must also include a functionality for the player to move the player sprite using an input from the keyboard so as to avoid the obstacle.

The conditions to check for collision must be put in to the obstacle class to make it easier to use it with our list of objects obs.

----------Code Start------------------

def collision(self, x, y):

global diameter

if( self.y < (y+(diameter/2)) ) and

self.x < ( x+(diameter/2)) and (x-(diameter/2)) < (self.x+self.w)):

return True

else:

return False

----------Code End-------------------

Now that we can collide with the obstacles, we must put in a condition to ensure that the game ends when you hit an obstacle, we will make use of a Boolean variable game\_over and an if-else block. Initially the game\_over variable is declared as false so that the game runs, and it gets updated by the means of the collision() that we call for every loop of the draw()

----------Code Start------------------

if game\_over == False:

for ob in obs:

ob.draw\_ob()

ob.move()

game\_over = ob.collision(x, y)

if game\_over == True:

break

else:

background(0)

textSize(32);

textAlign(CENTER)

fill(200);

text("GAME OVER", 402, 152, -30)

fill(255)

text("GAME OVER", 400, 150)

----------Code End-------------------

As you can see, if the game\_over variable remains false, then the game continues but if collision occurs then the collision() returns true which in turn makes the program control go into the else part of the if-else block.

In here we display the text “GAME OVER” in a slightly staggered manner by the use of the text(). textSize() and textAlign() sets the size and the alignment respectively of the text that comes after it.

Thus, when you collide into an obstacle, now the game will end and the screen shown in Figure 14 will be displayed



Figure 14: Game Over screen[13.png]

Moving on to the next part, Avoidance of the obstacles, we will use an inbuilt function keyPressed(). This function gets evoked every time you press a key on your keyboard, and it is in here within a condition that we will check if the upper key is pressed or not.

----------Code Start------------------

def keyPressed():

global lim, y

if key == CODED:

if keyCode == UP and lim < 1:

y -= 180

lim += 1

if y<=15:

y=15

----------Code End-------------------

In this inbuilt function, we can check for specific coded keys such as UP arrow easily, and adding an extra lim variable, initialized in the setup with the value 0, will make sure that the player cannot press up more than once while in the air. It is here we also check if the ball is going more than the maximum window limit.

----------Code Start------------------

if(y<185):

y += speed + 0.5

if(y>=185):

lim = 0

----------Code End-------------------

Adding these if statements ensure that the ball will come down after pressing UP, and also if the ball reaches the ground then to set back the lim to zero so as to allow the player to jump again

If you got this part down, then your game will look something like Figure 15

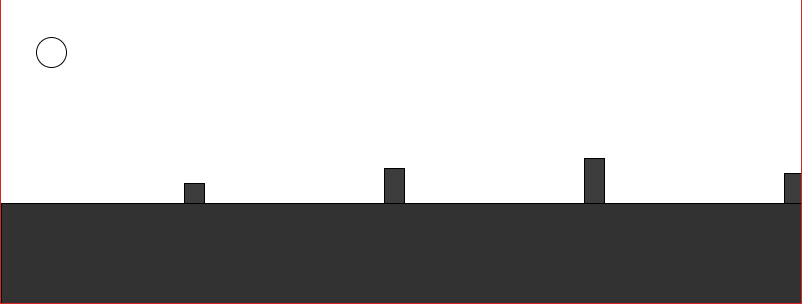


Figure 15: Jumping [14.png]

**Adding Score to the Game**

Now that our game is finally coming together, we can finally think of some way to add a score to the player, here in this tutorial we achieve that by creating another class which will contain the necessary arguments

----------Code Start------------------

class stats:

def \_\_init\_\_(self, score=0):

self.score = score

def update\_score(self):

global highscore

self.score += .005

if highscore < self.score:

highscore = self.score

----------Code End-------------------

Now that we have our class, lets look at the method present in the class update\_score() which updates the score based on a predefined value, here 0.005, and it is here that we also check for the highscore (Look at final code for the exact call of the function).

In addition to this class, we will also have to initialize a few variables in the setup(), as shown in the following code segment

----------Code Start------------------

highscore = 0

stat = stats(highscore)

----------Code End-------------------

And in our main game if condition, adding the following code segment will allow us to display the score and the highscore at the top right corner. The next code segment goes into the else part of the main game if condition, and this will allow us to display the player’s score in the game over screen

----------Code Start------------------

textSize(14)

textAlign(CENTER)

fill(0)

text("Score", 720, 25, -30)

text(int(stat.score), 770, 25, -30)

text("HIGH SCORE", 700, 50, -30)

text(int(highscore), 770, 50, -30)

----------Code End-------------------

----------Code Start------------------

textSize(14)

text("YOUR SCORE", 370, 200)

text(int(stat.score), 460, 200)

----------Code End-------------------

Now your game screen will be looking like Figure 16 and when the game over you will be getting Figure 17

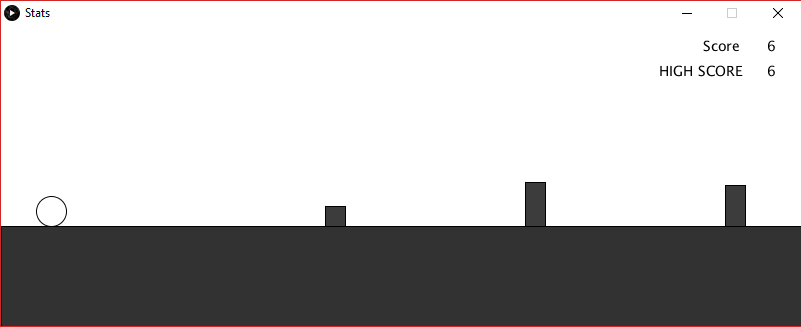
****

Figure 16: Player Stats [15.png]

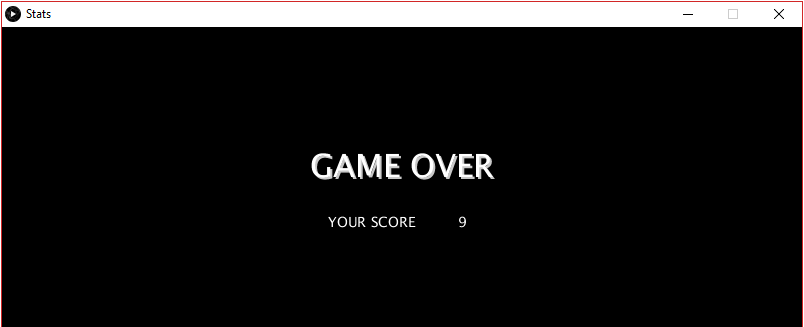
****

Figure 17: Game Over with Score[16.png]

With our score we can now increase the difficulty of the game by increasing the speed variable based on the score. An example is shown in the below code segment

----------Code Start------------------

def speed\_inc(score):

global speed

score \*= .00001

speed += score

----------Code End-------------------

**Finishing Touches**

Some of the finishing touches that we can add to our game to make it a smoother experience for the player are a replay functionality and play/pause functionality. Both of these can easily be implemented by using the keyPressed() and using text() for giving prompts to the user

The implementation of the functions in the keyPressed() is show in the below code segment

----------Code Start------------------

if key == ' ':

game\_over = False

stat.score = 0

speed = 1.5

if key == 'p' or key == 'P':

noLoop()

if key == 'r' or key == 'R':

loop()

----------Code End-------------------

The replay function works by resetting the game\_over variable to false again resulting in the main game loop in running again, thus reset the score and speed back to the initial condition. For pausing the game we use the inbuilt function noLoop() which stops the looping nature of the draw() resulting in the game being paused, and for resume, we use the inbuilt function loop() which makes the draw() loop again continuing the animation.

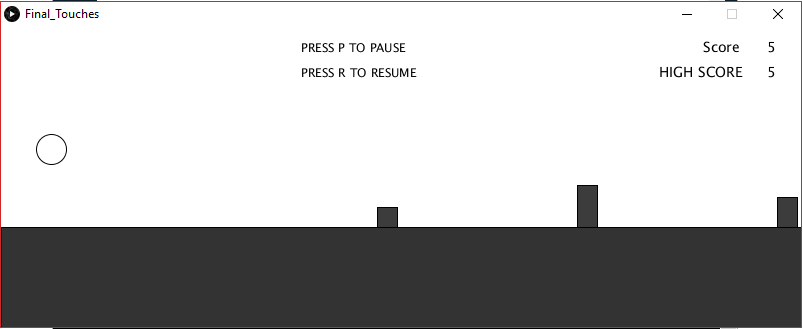


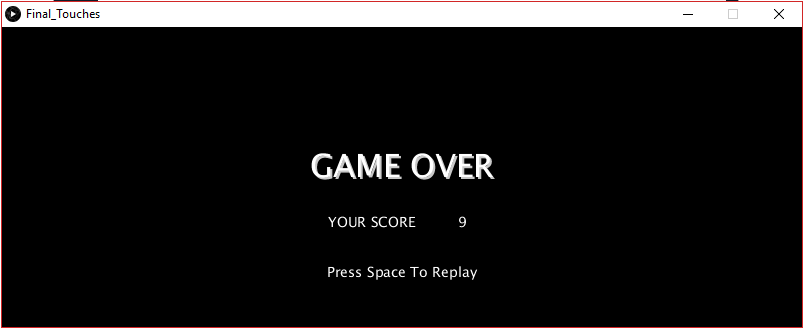
Figure 18: Play/Pause in Game[17.png]

Figure 19: Replay On End Game[18.png]

**Full Code for the Game**

import random

class obstacle:

def \_\_init\_\_(self , x, y, w, h):

self.x = x

self.y = y

self.w = w

self.h = h

def draw\_ob(self):

fill(60)

rect(self.x, self.y, self.w, self.h)

def move(self):

global speed

self.x -= speed

if self.x <= 0 :

self.x=780

self.h = random.randrange(20, 50)

self.y = 200 - self.h

def collision(self, x, y):

global diameter

if( self.y < (y+(diameter/2)) ) and ( self.x < ( x+(diameter/2)) and (x-(diameter/2)) < (self.x+self.w)):

return True

else:

return False

class stats:

def \_\_init\_\_(self, score=0):

self.score = score

def update\_score(self):

global highscore

self.score += .005

if highscore < self.score:

highscore = self.score

def speed\_inc(score):

global speed

score \*= .00001

speed += score

def setup():

global x, y, diameter, lim, game\_over, highscore, obs, stat, speed

size(800, 300)

x = 50

y = 185

diameter = 30

lim = 0

game\_over = False

highscore = 0

obs = []

x\_val = 780

y\_val = 180

ht = 20

wdt = 20

for i in range(4):

ob = obstacle(x\_val, y\_val, wdt, ht)

x\_val += 200

ht = random.randrange(20, 50)

y\_val = 200 - ht

obs.append(ob)

stat = stats(highscore)

speed = 1.5

def draw():

global x, y, diameter, lim, game\_over, speed

background(255)

if game\_over == False:

fill(51)

rect(0, 200, 800, 300)

fill(255)

ellipse(x, y, diameter, diameter)

if(y<185):

y += speed + 0.5

if(y>=185):

lim = 0

for ob in obs:

ob.draw\_ob()

ob.move()

game\_over = ob.collision(x, y)

if game\_over == True:

break

else:

stat.update\_score()

speed\_inc(stat.score)

textSize(12);

textAlign(LEFT)

fill(0)

text("PRESS P TO PAUSE", 300, 25, -30)

text("PRESS R TO RESUME", 300, 50, -30)

textSize(14);

textAlign(CENTER)

fill(0)

text("Score", 720, 25, -30)

text(int(stat.score), 770, 25, -30)

text("HIGH SCORE", 700, 50, -30)

text(int(highscore), 770, 50, -30)

else:

background(0)

textSize(32);

textAlign(CENTER)

fill(200);

text("GAME OVER", 402, 152, -30)

fill(255)

text("GAME OVER", 400, 150)

textSize(14)

text("YOUR SCORE", 370, 200)

text(int(stat.score), 460, 200)

text("Press Space To Replay", 400, 250)

x\_val = 780

for ob in obs:

ob.x = x\_val

x\_val += 200

def keyPressed():

global lim

global y

global game\_over, stat, speed

if key == CODED:

if keyCode == UP and lim < 1:

y -= 180

lim += 1

if key == ' ':

game\_over = False

stat.score = 0

speed = 1.5

if key == 'p' or key == 'P':

noLoop()

if key == 'r' or key == 'R':

loop()

if y<=15:

y=15

**Acknowledgement**

The author is grateful to Sibi Chakkaravarthy Sethur, Department of Computer Science and Engineering in Vellore Institute of Technology- Andhra Pradesh (VIT-AP) University, Amaravati for his valuable support.

The author is also grateful to Hari Seetha, Head, Department of Computer Science and Engineering in Vellore Institute of Technology- Andhra Pradesh (VIT-AP) University, Amaravati for her valuable support.

**Author Biography**

Jatin Karthik Tripathy is a second-year student of Department of Computer Science and Engineering in Vellore Institute of Technology- Andhra Pradesh (VIT-AP) University, Amaravati. His interests include Graphics programming, Graphics Modelling, Artificial Intelligence etc. He can be reached through jatinkarthik (dot) tripathy (at) vitap(dot) ac(dot) in.