# Final project report

# 12122211

# Rebecca Bronczyk

## Definition of classes, subclasses in pygame and circuitpython

1. The classes **Bubble** (in bubble.py), **Bullet** (in bullet.py) and **Paddle** (in paddle.py) were created in separate files containing only the Bubble, Bullet or Paddle class in the pygame application. Pygame is imported (1), so that the library can be accessed in the class. Pygame is needed to create shapes and draw them to the pygame window. In the case of the bubble, the shape used is a circle. A constructor is created in each class (2), for the creation of the attributes of each instance (instance variables)(3). The rest of the methods include string representation(4), to return a readable description of the class and the methods containing the functionality of the objects, such as movement (5) and getters (6). In the main.py code, all classes are imported (a(Pygame main code)) and an instance of each object is created (bubble\_instance, bullet\_instance and paddle\_instance) (1(Pygame main code)). In the game loop, the methods regarding functionality of the objects are called when appropriate (b(Pygame main code)). In the main code, interactions between the objects are created using getters, to get the object’s attributes (location on the x and y axes, radius,…; See Pygame main code). This is needed since private instance variables cannot be accessed in another file even if it is imported. For example, the Bubble was created by importing the bubble class (from bubble import Bubble). Then an instance was created with the attributes of the bubble (bubble\_instance = Bubble(DISPLAY\_WIDTH//2, DISPLAY\_HEIGHT-(DISPLAY\_HEIGHT-bubble\_radius), [150, 0, 100], bubble\_radius, bubble\_speed, bubble\_speed) #self, x, y, colour, radius, speed\_x, speed\_y). This sets the instance variables in the constructor of the class. In the game loop, the methods of the class are then called using the instance to create behaviour: bubble\_instance.move(horizontal\_bound, vertical\_bound), calls the method move in the Bubble class, that lets the bubble move across the pygame display, within the bounds in the parameters (horizontal\_bound and vertical\_bound). This is called encapsulation: the functionality of the Bubble can be used in the main.py code without needing to know how the methods work, only the parameters the constructor and methods need to work have to be known to create an instnace and create a behaviour.

Bubble:

# import pygame

import pygame



# import random

import random

 # create bulbble class

class Bubble:

    # create init method to construct classes attributes

    def \_\_init\_\_(self, x, y, colour, radius, speed\_x, speed\_y):



        '''

        constructor to set up bubble class

        defend parameter choice in comments

        '''

        # create private instance variables --> instance variables not class variables so that several instances of the bubble with unique attributes can be created and commented out as needed

        # bubble on x axis (int); needed to locate the position of the bubble on the x axis



        self.\_\_x = x # instance variables for coordinates: can position bubble differently (i.e. start in left corner)

        # bubble on y axis (int); needed to locate the position of the bubble on the y axis

        self.\_\_y = y

        # bubble radis (int); needed to set the size of the bubble to know when it hit bounds

        self.\_\_radius = radius # instance variable so size of bubble can change

        # bubble coordinates with x and y (list); needed to get the x and y coordinates of the bubble at once & relocate it to x any y position at same time

        self.\_\_coordinates = [self.\_\_x, self.\_\_y] # instance variable to accept instance variables x & y

        # speed of horizontal bubble movement (x-axis; int); needed to define speed bubble moves horizontally for diagonal movement

        self.\_\_speed\_x = speed\_x # speed instance variables so it can be modified (i.e. create easier and harder versionof game: higher level = faster)

        # vertical speed of bubble movement (y axis; int); needed to define speed bubble moves vertically for diagonal movement

        self.\_\_speed\_y = speed\_y

        # both speeds needed because width and height of constraints might be different; i.e. if both the same horizontal might be too fast and never reaches line

        # colour of bubble (list:[r,g,b]); needed for drawing of bubble to window (and colouring it)

        self.\_\_colour = colour # instance variable so can change colour of bubble

    def \_\_str\_\_(self):



        '''

        string representation containing reader-friendly information concerning the object's instance, including the color and

        position of the Bubble.

        '''

        # create string representation

        bubble\_info = 'Instance of class bubble with the colour ' + self.\_\_colour + ' and the coordinates of '+ self.\_\_coordinates + '.'

        # return string for string representaiton

        return bubble\_info

    def draw(self, display):

        '''

        method to draw the bubble on the display window

        - display window is passed in

        - draw bubble with pygame features (circle)

        - Bubble not drawn in other methods.

        '''

        # draw bubble as a circle on pygame display using colour, coordinates and radius instance variables

        pygame.draw.circle(display, self.\_\_colour, self.\_\_coordinates, self.\_\_radius)



    def move(self, horizontal\_bound, vertical\_bound):



        '''

        method to move the bubble diagonally within constraints of the pygame window (called from main)

        - speed of bubble defined in instance variables (value of programmer's choice)

        constraints for the bubble:

        - bubble movement constrained within horizontal\_bound and vertical\_bound

        - bubble changes direction when collides with horizontal\_bound

        - bubble relocates to a random position when collides with the vertical\_bound (line)

        '''

        # if the bubble has not hit the line

        if self.\_\_y < vertical\_bound-self.\_\_radius:

            # --- move bubble diagonally

            # move it downwards on x-axis

            self.\_\_y += self.\_\_speed\_y

            # move it right/left on x-axis

            self.\_\_x += self.\_\_speed\_x

            # if bubble hits left or right horizontal bounds

            if self.\_\_x >= horizontal\_bound or self.\_\_x <= 0+self.\_\_radius:

                # change direction of bubble's horizontal movement

                self.\_\_speed\_x = -self.\_\_speed\_x

            # update y coordinate of bubble in coordinates instance variable

            self.\_\_coordinates[1] = self.\_\_y

            # update x coordinate of bubble in coordinates instance variable

            self.\_\_coordinates[0] = self.\_\_x

        # if the bubble hit the line

        else:

            # set y-axis position of bubble to random location over line & below vertical bound

            self.\_\_y = random.randint(self.\_\_radius, (vertical\_bound-self.\_\_radius))

            # update y coordinate of bubble in coordinates instance variable

            self.\_\_coordinates[1] = self.\_\_y

            # set y-axis position of bubble to random location in horizontal bounds

            self.\_\_x = random.randint((self.\_\_radius\*2), (horizontal\_bound-self.\_\_radius))

            # update x coordinate of bubble in coordinates instance variable

            self.\_\_coordinates[0] = self.\_\_x

    def relocate(self, horizontal\_bound, vertical\_bound):

        '''

        Method to generate a new random position for the bubble within the horizontal and vertical bounds passed in as parameters

        '''

        # set y-axis position of bubble to random location over line & below vertical bound

        self.\_\_y = random.randint(self.\_\_radius, vertical\_bound-self.\_\_radius)

        # set y-axis position of bubble to random location in horizontal bounds

        self.\_\_x = random.randint((self.\_\_radius\*2), (horizontal\_bound-self.\_\_radius))

        # update coordinates of bubble in coordinates instance variable

        self.\_\_coordinates = [self.\_\_x, self.\_\_y]

    def change\_size(self, lower\_limit, upper\_limit, resize\_factor):

        '''

        Method to change bubble size between lower and upper limits (inclusive of both) given as parameters

        '''

        # check if the radius if changed is still within the upper and lower limit

        # if would be out of limits

        if not self.\_\_radius + resize\_factor <= upper\_limit or not self.\_\_radius + resize\_factor >= lower\_limit:

            # turn the resize\_factor around

            resize\_factor=-resize\_factor

        # add resize factor to radius

        self.\_\_radius = self.\_\_radius + resize\_factor

        # return the new resize factor (to keep it updated: change direction only when a limit would be surpassed)

        return(resize\_factor)

# --- creating getters for the x, y, and size the bubble



    def get\_bubble\_x(self):

        '''

        getter to return x coordinate of the bubble

        '''

        return self.\_\_x

    def get\_bubble\_y(self):

        '''

        getter to return y coordinate of the bubble

        '''

        return self.\_\_y

    def get\_bubble\_size(self):

        '''

        getter to return bubble size (radius)

        '''

        return self.\_\_radius

1. The **SensorLightDisplay** class was created in a separate file (see code below Feature 2). The class is imported to the circuitpython code (code.py). All functionality of the object, including variables and methods, are encapsulated in the SensorLightDisplay class in the sensorlightdiplay.py file. Circuitpython is imported to the sensorlightdisplay.py file (A), to access the library’s functionality and access the board. The SensorLightDisplay class includes class variables (B), a constructor (C) holding board variables (D) (like the pixel brightness (cp.pixels.brightness)) and methods (E) to create behaviours. Therefore, an instance/object of the class has to be created in the code.py (sensorlightdisplay\_instance), with which the methods in the SensorLightDisplay class are called (sensorlightdisplay\_instance.advanced\_control\_feedback(x)). The methods contain the attributes and functionality of the object in their code.

## Pygame main code

All classes of objects: Bubble, Paddle and Bullet are imported into the main code to create interactions. An Instance of each object is created in the code (1). By calling the methods of the classes paddle, bubble and bullet, the objects are drawn to the pygame window (2). This is done in the game loop, so that when a method changes the position of an object, this is updated(3) while the pygame window is open.

The interaction between the **bubble** and **bullet** (collision), is created in the main code (A). Getters are used to get the coordinates and radii of the bubble and bullet (A1), both of which are pygeme circles. Using the x and y coordinates of each, the distance between both circles is calculated in the main code (A2). If the distance between their coordinates is smaller than the sum of the radii, the bullet and bubble collided. Then, four methods in the bullet.py and bubble.py files are called (A3), two for each object. First, the relocate() method in the Bubble class (bubble.py) is called using the instance of the bubble, to relocate the bubble to a random position within the horizontal and vertical bounds. Furthermore, the change\_size() method in the Bubble class is called, to resize the bubble by the resize\_factor, to a size within the upper and lower bounds. In order to relocate the bullet to the paddle after a collision, the wall\_collision() method in the Bullet.py (Bullet class) is called, using the instance of the bullet that was created in the code. The second method called in the Bullet class, is the change speed method. It changes the bullet’s speed to a random speed within the lower and upper bounds, passed in as parameters.

The interactions between the **bullet** and the **paddle** are also created in the main code (B). The bullet is located on top of the paddle, and moves with it, when the paddle is moved using the left and right arrow keys. In the action listener, moveleftbool (B1) is set to true while the left arrow key is pressed and moverightbool is set to true if the same happens for the right arrow key (B2). Therefore, if moveleftbool is set to true, the method move\_left() is called using the instance of the paddle, which causes the paddle to move left on the screen (B1a). Afterwards, the paddles coordinates are updated in the main code using the get\_paddle\_coordinates\_and\_size() method in the paddle class(B1b). If the bullet was not shot (shootbool = False: set to true if the space bar was pressed), the bullet is moved using the move() method in the paddle class, giving the updated location of the paddle as a parameter and moving the bullet to the location of the paddle(B1c). The same is the case if the right arrow key is pressed and moverightbool is set to true (B2a). Therefore, the method in the paddle.py relocates the bullet location to the location of the paddle.

# --- main.py to create interactions between classes and set up pygame window

# import pygame to use its features

import pygame

# import Bubble class

from bubble import Bubble

# import paddle class

from paddle import Paddle



# import Bullet class

from bullet import Bullet

from math import sqrt

# initialise pygame

pygame.init()

# --- initialise display characteristics

# set the background colour to black

background = [0,0,0]

# set the display width and height (here. to 600px)

DISPLAY\_WIDTH = 600

DISPLAY\_HEIGHT = 600

# create list for display size out ot display width and height

DISPLAY\_SIZE = [DISPLAY\_WIDTH, DISPLAY\_HEIGHT]

# set up pygame display

display = pygame.display.set\_mode(DISPLAY\_SIZE)

# initialise pygame time/clock

clock = pygame.time.Clock()

# CREATE CLASS INSTANCES

# --- create instance of BUBBLE

# set up instance of bubble

# create bubble radius variable

bubble\_radius = (DISPLAY\_WIDTH+DISPLAY\_HEIGHT)/50

#create speed variable to be used for speed\_x and speed\_y

bubble\_speed = 1

# create instance of BUBBLE

bubble\_instance = Bubble(DISPLAY\_WIDTH//2, DISPLAY\_HEIGHT-(DISPLAY\_HEIGHT-bubble\_radius), [150, 0, 100], bubble\_radius, bubble\_speed, bubble\_speed) #self, x, y, colour, radius, speed\_x, speed\_y



resize\_factor = (bubble\_radius\*2)//10

# --- create instance of PADDLE

# setup fot instance of paddle

# create paddle\_width variable according to DISPLAY\_WIDTH

paddle\_width = DISPLAY\_WIDTH//10

# create paddle\_height variable according to paddle\_with

paddle\_height = paddle\_width//3

# paddle\_x coordinate is the middle of the display (subtract paddle\_width//2 because rectangle draws from upper left corner)

paddle\_x = DISPLAY\_WIDTH//2-paddle\_width//2

# set height of paddle to display height (bottom of display); subtract paddle height because rectangle draws from upper left corner (else exceeds display height and is not visible)

paddle\_y = DISPLAY\_HEIGHT-paddle\_height

# create paddle colour variable

paddle\_colour = [0,255,255]

# create paddle speed

paddle\_speed = bubble\_speed\*5

# create instance of PADDLE

paddle\_instance = Paddle(paddle\_x, paddle\_y, paddle\_width, paddle\_height, paddle\_colour, paddle\_speed)



# --- create instance of BULLET

# setup fot paddle\_instance

# set bullet radius to half of bubble

bullet\_radius = bubble\_radius//2

# position bullet on middle of screen / x-axis

bullet\_x = DISPLAY\_WIDTH//2

# positon bullet on paddle

bullet\_y = DISPLAY\_HEIGHT-paddle\_height-bullet\_radius

# variable to store speed of the bullet

bullet\_speed = bubble\_speed \*5

# create instance of BULLET

bullet\_instance = Bullet(bullet\_x, bullet\_y, bullet\_radius, [255, 255, 255], bullet\_speed) #x, y, colour, radius, speed\_x, speed\_y



#create line\_with variable

line\_width = 5

# set run\_game to true so the game loop can run

run\_game = True

# set moveleftbool to move paddle (with bullet if bullet on paddle) left to false

moveleftbool = False

# set moverightbool to move paddle (with bullet if bullet on paddle) right to false

moverightbool = False

# set shootbool to shoot bullet to false

shootbool = False

# get coordinated of the paddle to later move the bullet with the paddle coordinates

paddle\_coord = paddle\_instance.get\_paddle\_coordinates\_and\_size()

# --- CREATE GAME LOOP

while run\_game:

   # fill background

   display.fill(background)

   # --- prepare drawable elements

   # draw bubble to window

   bubble\_instance.draw(display)



   # create vertical\_bound variable according to DISPLAY\_HEIGHT

   vertical\_bound = DISPLAY\_HEIGHT//2

   # horizontal bound is DISPLAY\_WIDTH-bubble\_radius

   horizontal\_bound = DISPLAY\_WIDTH-bubble\_instance.get\_bubble\_size()

   # draw the line to window

   pygame.draw.line(display, [0,255,0], [0, vertical\_bound], [DISPLAY\_WIDTH, vertical\_bound],line\_width)

   # draw the paddle

   paddle\_instance.draw(display)



   # draw the bullet

   bullet\_instance.draw(display)

   # move the bubble

   bubble\_instance.move(horizontal\_bound, vertical\_bound)

   # --- get size and positions of bubble and bullet

   # get location of bullet on x axis

   bullet\_x = bullet\_instance.get\_bullet\_x()



   # get location of bullet on y axis

   bullet\_y = bullet\_instance.get\_bullet\_y()

   # get radius of bullet

   bullet\_size = bullet\_instance.get\_bullet\_radius()

   # get radius of bubble

   bubble\_size = bubble\_instance.get\_bubble\_size()

   # get location of bubble on x axis

   bubble\_x = bubble\_instance.get\_bubble\_x()

   # get location of bubble on x axis

   bubble\_y = bubble\_instance.get\_bubble\_y()



   distance = sqrt((bubble\_x-bullet\_x)\*\*2+(bubble\_y-bullet\_y)\*\*2)

   # if the distace between bubble and bullet is smaller or equal to the sum of their radii (aka if they collide)

   if distance <= bubble\_size+bullet\_size:



      # relocate the bubble to a random position

      bubble\_instance.relocate(horizontal\_bound, vertical\_bound)



      # relocate bullet to board after hitting so that bubble is not relocated over and over again

      bullet\_instance.wall\_collision()

      # change the bubbles size using calling the change\_size method with the bubble\_radius as a minimum, double it as maximum radius and the resize factor

      resize\_factor = bubble\_instance.change\_size(bubble\_radius, bubble\_radius\*2, resize\_factor)

      # change the speed of the bullet calling the change\_speed method in the bullet.py using half the bullet\_speed as the lower and double it as the upper limit

      bullet\_instance.change\_speed(bullet\_speed//2, bullet\_speed\*2)

# called from ACTION LISTENER

   # if LEFT arrow pressed

   if moveleftbool:



      # call move\_left method in paddle to move paddle left

      paddle\_instance.move\_left()

      # get new paddle coordinates to move bullet with paddle

      paddle\_coord = paddle\_instance.get\_paddle\_coordinates\_and\_size()



      # if the bullet is not shot, otherwise would move with paddle while shot

      if shootbool == False:



         # move the bullet with the (coordinates of) the paddle

         bullet\_instance.move(paddle\_coord)

   if moverightbool:



      # call move\_right method in paddle to move paddle right, give display\_width to create bound (so that paddle does not go off screen)

      paddle\_instance.move\_right(DISPLAY\_WIDTH)

      # get new paddle coordinates to move bullet with paddle

      paddle\_coord = paddle\_instance.get\_paddle\_coordinates\_and\_size()

      # if the bullet is not shot, otherwise would move with paddle while shot

      if shootbool == False:

         # move the bullet with the (coordinates of) the paddle

         bullet\_instance.move(paddle\_coord)

   # if shootbool was set to true because SPACE was pressed

   if shootbool:

      # call move method in bullet to move bullet up

      bullet\_instance.move(paddle\_coord)

      # call wall\_collision so shootbool is set to false if the bullet has met the top of the y-axis/pygame window

      bullet\_instance.wall\_collision()

      # if shootbool set to false: wall\_collision method calls set\_trigger in bullet to set bullet\_triggered to false

      # so that the move method relocates the bullet on top of the paddle

   # --- EVENT LISTENER

   # for every event on the pygame display - loop

   for event in pygame.event.get():

      # if user has closed the pygame window

      if event.type == pygame.QUIT:

         # set run\_game to false to stop GAME LOOP

         run\_game = False

      # IF KEY WAS PRESSED

      if event.type == pygame.KEYDOWN:

         # IF DOWN ARROW PRESSED

         if event.key == pygame.K\_LEFT:



            # set moveleftbool to True



            moveleftbool = True

         if event.key == pygame.K\_RIGHT:

            # set moverightbool to True

            moverightbool = True



         # IF SPACE KEY pressed

         if event.key == pygame.K\_SPACE:

            # set shootbool to True because bullet was triggered

            shootbool = True

            # call set\_trigger method in bullet to set\_trigger to True

            bullet\_instance.set\_trigger(True)

      # IF KEY WAS RELEASED

      if event.type == pygame.KEYUP:

         # if LEFT arrow was released

         if event.key == pygame.K\_LEFT:

            # set moveleftbool to False

            moveleftbool = False

         # if RIGHT arrow was released

         if event.key == pygame.K\_RIGHT:

            # set moverightbool to False

            moverightbool = False

   # update framerate

   clock.tick(70)

   # update display

   pygame.display.update()



# quit pygame

pygame.quit()

# quit application

quit()

## Circuitpython main code

First, imports are made, to ensure the interaction of the main code with the circuitplayground, sensorlightdisplay code, time feature and computer keyboard (1). Instances of the sensorlightdisplay and keyboard are created, so that they can be used (2). If the boards switch is set to true(3), the code in the loop runs. The x acceleration of the board is read (4) and flipped\_x set to a positive value of the x acceleration(5). If the value of the acceleration\_x is within the law of acceleration set in the acceleration\_peak variable, the rest of the code is executed(6). The advanced\_control\_feedback() method in the SensorLightDisplay class is called, to light up neopixels on the board according to its movement (7). If the board is tilted to the right, with x >3 (8), the right arrow key of the keyboard is pressed while the left arrow key is pressed if x< -3 (the board is tilted to the left)(9). This creates the interaction between the board and the pygame code. Since the pygame bubbleshooter responds to the keyboard arrows being pressed, translating the circuitplayground tilts and button presses to a keyboard activity controls the pygame. The unused keys are released or the keys are released if no key is pressed, so that no errors come up. This is because, press keeps a key pressed until it is released. This is used, since the paddle should move smoothly while the board is tilted. If one of the buttons on the circuitplayground is pressed (10), the space bar of the keyboard is pressed one and released. Send does this and is used, since the bubble can only be fired once. It has to disappear from the screen and be relocated to the paddle before it can be used again. If the switch is set off mid-game the arrows are released to stop them from messing with for example, text documents. Furthermore, the pixels are swiched off if the switch is turned off

# import circuitplayground

from adafruit\_circuitplayground import cp

# import SensorLightDisplay class

from sensorlightdisplay import SensorLightDisplay



# import time

import time

# import human interaction devices

import usb\_hid

# import they keyboard

from adafruit\_hid.keyboard import Keyboard

# import the circuitpython keycodes to use the keys of the keyboard

from adafruit\_hid.keycode import Keycode

# create an instance of the sensorlightdisplay class setting the pixel brightness to 10%

sensorlightdisplay\_instance = SensorLightDisplay(0.1)

# initialise the keyboard to use it

kbd = Keyboard(usb\_hid.devices)



# assign the acceleration peak to a variable

acceleration\_peak = 9.81

black = [0,0,0]

# --- create loop

while True:

    # get the acceleration of the board

    x,y,z = cp.acceleration



    # if the switch is on

    if cp.switch:



        # if the acceleration of the x axis is negative (tilted left)

        if x < 0:

            # assign a positive value of x to flipped\_x

            flipped\_x = x\*-1



        # if it is positive

        else:

            # set flipped\_x to x

            flipped\_x = x

        # check if x is within the bounds of the law of acceleration to reduce noise

        if flipped\_x <= acceleration\_peak:



            # FEATURE 2 - call the advanced\_control\_feedback method in the sensorlightdisplay class

            sensorlightdisplay\_instance.advanced\_control\_feedback(x)



            # if the acceleration is greater than 3 (aka board tilted to the right)

            if x > 3:



                # press the right arrow of the keybard (use of instance of keyboard and cp keycodes)

# press and release used because paddle should move while tilted (smooth movement)



                kbd.press(Keycode.RIGHT\_ARROW)

                # realease the left arrow of the keyboard (use of instance of keyboard and cp keycodes)

                kbd.release(Keycode.LEFT\_ARROW)

            # if the acceleration is smaller than -3 (aka board tilted to the left)

            elif x < -3:



                # press the left arrow of the keybard (use of instance of keyboard and cp keycodes)

# press and release used because paddle should move while tilted (smooth movement)

                kbd.press(Keycode.LEFT\_ARROW)

                # realease the right arrow of the keyboard (use of instance of keyboard and cp keycodes)

                kbd.release(Keycode.RIGHT\_ARROW)

            # if the acceleration id between -3 and 3

            else:

                # realease the left arrow of the keyboard (use of instance of keyboard and cp keycodes)

                kbd.release(Keycode.LEFT\_ARROW)

                # realease the right arrow of the keyboard (use of instance of keyboard and cp keycodes)

                kbd.release(Keycode.RIGHT\_ARROW)

        # if one of the buttons on the board are pressed

        if cp.button\_a or cp.button\_b:



            # send (press and release) the space bar of the keyboard (use of instance of keyboard and cp keycodes): because the bubble can not be fired continuously

            kbd.send(Keycode.SPACE)

# if the switch is not set

else:

# release the pressed keys to stop any movement caused by them (else they keep pressed even if switch off)

kbd.release(Keycode.LEFT\_ARROW)

kbd.release(Keycode.RIGHT\_ARROW)

# turn the pixels off aka colour them black

cp.pixels.fill(black)

cp.pixels.show()

## Feature 1: Bubble and Bullet interactions in pygame application

When the **bullet and bubble collide**, two methods are called in main.py. One method, changes the size of the bubble, while the other changes the speed of the bullet (See A3 in code in “Pygame main code”). The size of the bubble is changed by calling the change\_size() method in the Bubble class. It takes the smallest and biggest possible radii of the bubble as parameters (lower\_limit and upper\_limit). Furthermore, the factor, by which the size is changed is handed to the method as a parameter (resize\_factor). The method changes the resize\_factor to its negative, if the size of the bubble would exceed one of the limits if the current resize\_factor was applied(1). The resize\_factor is then applied to change the radius of the bubble(2). Lastly, the resize\_factor is returned, so that the next time the function is called, the last updated version of the resize\_factor is applied(3). This means, that the resize factor is applied and bubble gets bigger until the upper limit is hit. Then, the resize factor is turned to a negative causing the bubble to decrease in size until the lower limit is hit, when the resize\_factor is turned positive again, causing the bubble to increase its radius again.

    def change\_size(self, lower\_limit, upper\_limit, resize\_factor):

        '''

        Method to change bubble size between lower and upper limits (inclusive of both) given as parameters

        '''

        # check if the radius if changed is still within the upper and lower limit

        # if would be out of limits

        if not self.\_\_radius + resize\_factor <= upper\_limit or not self.\_\_radius + resize\_factor >= lower\_limit:

            # turn the resize\_factor around



            resize\_factor=-resize\_factor

        # add resize factor to radius

        self.\_\_radius = self.\_\_radius + resize\_factor



        # return the new resize factor (to keep it updated: change direction only when a limit would be surpassed)



        return(resize\_factor)

The **speed of the bullet** is changed with the change\_speed() method in the Bullet class in bullet.py. The method is called in the main code, if a collusion occurred. The method takes a lower and upper limit as parameters, that cannot be exceeded. The method changes the speed of the bullet (a private instance variable) to a random speed (1) within the bounds defined in the parameters (2).

    def change\_speed(self, lower\_limit, upper\_limit):

        '''

        parameters containing the minimum (lower\_limit) and maximum (upper\_limit) speed of the bullet

        changes bullet speed to random speed within limits

        '''

        # import random to use library

        import  random

        # set speed of bullet to a random integer within the limits



        self.\_\_speed = random.randint(lower\_limit, upper\_limit)

## Feature 2: creating and calling a new method for controller feedback in SensorLightDisplay class

The advanced\_control\_feedback() method in the SensorLightDisplay class, takes the value for the acceleration on the x axis of the board as a parameter (1). Checking, whether the acceleration is within the bounds of the acceleration\_peak, class variable to reduce noise (2). If this is the case, it checks whether the acceleration is below -3 or above 3 to make the pixels only light up if the board is actually tilted and not only due to oscillations when it is held in hand (3).



If the board is tilted to the right (acceleration\_x>3), the neopixels light up from the middle (pixel 2) and the more it is tilted right, the more pixels are lit on the right side of the board (the last pixels to be lit are pixels 0 and 4)(4). This is achieved by calculating a mapping value between 0 and 2 (4a) that is added/subtracted to/from the middle pixel to get the last neopixels on each side of the middle pixel to be lit (4b). Then, the code loops over the pixels (4c) and colours them if they are within the pixels calculated with the mapping value (4d). The colour, the neopixel(s) are lit when the board is tilted also changes(5). If the board is only tilted slightly, the neopixel is lit up purple and the farther it is tilted to the right the more red is introduced to the colour until the neopixels are lit red when the board is held vertically. This is achieved using the factor (255/ acceleration\_peak) to be able to later map the acceleration\_x to a colour value. For the red, the acceleration\_x is multiplied with the factor, so that the higher acceleration\_x the higher the red value (up to 255). For the blue r,g,b value, the opposite is done. The 255/acceleration\_peak\*acceleration\_x is subtracted from 255, flipping the values around and taking away more of the blue colour the more the board is tilted(5).



If the board is tilted to the left (acceleration\_x<-3), the same is done (6). Starting with the 7th neopixels, the acceleration\_x value is mapped to a value between 0 and 2 meaning, that either neopixel 7 (7+-0), neopixels 6,7,8 (7+-1) or all neopixels on the left of the board (5 to 9 (7+-2)) are lit (6a). The same factor is used to map the acceleration\_x to a colour, only that the red value decreases as the board is tilted while the blue value increases (6b). This means, the further the board is tilted to the left the less red and the more blue the colour of the lit up neopixels becomes, until the neopixels are coloured blue when the board is held vertically. The pixels are lit by looping over them and colouring them with the colour created using the factor (6c).



If the acceleration\_x value is between -3 and 3, all neopixels are switched off (7).



# import circuitpython

from adafruit\_circuitplayground import cp



# import time

import time

# create sensorlightdisplay class

class SensorLightDisplay:

    # create class variable: off state of pixels (because 0,0,0 is always the rgb for black)

    \_\_pixels\_off\_state = [0, 0, 0]



    # create class variable: acceleration peak (because law of acceleration does not change)

    \_\_acceleration\_peak = 9.81

    # create class variable:reand in the length of neopixels (does not change no matter what board is connected)

    \_\_neopixels\_amount = len(cp.pixels)

    # create constructor, with brightness of pixels as parameter

    def \_\_init\_\_(self, brightness):



        # set board variable brightness to the value in the parameter

        cp.pixels.brightness = brightness



        # set auto write to false that pixels are only lit up when instructed (no delay in loops)

        cp.pixels.auto\_write = False

    # --- CREATE FEATURE 2: control\_feedback\_x method with acceleration\_x as parameter



    def advanced\_control\_feedback(self, acceleration\_x):



        # check wether the acceleration is between -9.81 and 9.81 - only execute the following code if this is the case (done in main as well so redundant but let it in in case class imported to another file)

        if acceleration\_x <= SensorLightDisplay.\_\_acceleration\_peak and SensorLightDisplay.\_\_acceleration\_peak >= -SensorLightDisplay.\_\_acceleration\_peak:



            # if the acceleration is above 3 or smaller than -3

            if acceleration\_x > 3 or acceleration\_x<-3:



                # assign the factor, by which the colour should be determined to factor

                factor= 255/SensorLightDisplay.\_\_acceleration\_peak

                # assign the acceleration\_x to flipped\_acceleration

                flipped\_acceleration = acceleration\_x

                # if the acceletation is negative (tilted to the left)

                if acceleration\_x <0:

                    # turn the negative into a positive value

                    flipped\_acceleration \*=-1

            # if the acceleration on the x axis is above 3 (light right side of board)

            if acceleration\_x > 3:



                # create the purple colour using the flipped acceleration value and the factor (becomes more red the more turn right)

                purple = [flipped\_acceleration\*factor, 0, 255-(flipped\_acceleration\*factor)]



                # assign the n//4th pixel to first\_nepixel\_location to start from the middle

                first\_neopixel\_location = int(SensorLightDisplay.\_\_neopixels\_amount//4)

                # calculate the mapping value to calculate the last neopixels to be coloured (use neopixel\_amount//4 because 10//4 = 2 and 2 npixels left and right of first pixel to be lit)

                mapping\_value = int(acceleration\_x \* SensorLightDisplay.\_\_neopixels\_amount//4 / SensorLightDisplay.\_\_acceleration\_peak)



                # add the mapping value to the first neopixel location to get the last pixel to be lit above it

                last\_neopixel\_location\_a = first\_neopixel\_location + mapping\_value



                # subtract the mapping value to the first neopixel location to get the last pixel to be lit below it

                last\_neopixel\_location\_b = first\_neopixel\_location - mapping\_value

                # loop over the neopixels in the range of the first to the last neopixel that can possibly be lit

                for pixel in range(first\_neopixel\_location, SensorLightDisplay.\_\_neopixels\_amount//2):



                    # if the pixel is within the last neopixel locations

                    if pixel <= last\_neopixel\_location\_a and pixel >=last\_neopixel\_location\_b:

                        # colour the corresponding pixel purple (first np or np below the first neopixel location (pixels 3 and 4))

                        cp.pixels[((SensorLightDisplay.\_\_neopixels\_amount//2)-1)-pixel] = purple



                        # colour the pixel purple (first np of np above first neopixel location (pixels 0 and 1))

                        cp.pixels[pixel] = purple

                        # show the coloured pixels now

                        cp.pixels.show()

                    # if pixel is not within the bounds of the last neopixel locations

                    else:

                        # turn the corresponding pixel off (first np or np below the first neopixel location (pixels 3 and 4)); using the off\_state class variable

                        cp.pixels[((SensorLightDisplay.\_\_neopixels\_amount//2)-1)-pixel] = SensorLightDisplay.\_\_pixels\_off\_state

                        # turn the pixel off (first np of np above first neopixel location (pixels 0 and 1)); using the off\_state class variable

                        cp.pixels[pixel] = SensorLightDisplay.\_\_pixels\_off\_state

                        # show the turned off pixels now

                        cp.pixels.show()

            # if the acceleration on the x axis is above 3 (light right side of board)

            elif acceleration\_x< -3:



                # create the purple colour using the flipped acceleration value and the factor (becomes more blue the further tilt to left)

                purple = [255-(flipped\_acceleration\*factor), 0, flipped\_acceleration\*factor]



                # calculate the first neopixel location to be lit (need to find the pixel dividing the upper quarter from the lower three quarters of the pixels)

                first\_neopixel\_location = int(SensorLightDisplay.\_\_neopixels\_amount\*(3/4))

                # calculate the mapping value to calculate the last neopixels to be coloured (use neopixel\_amount//4 because 10//4 = 2 and 2 npixels left and right of first pixel to be lit); take \*-1 because acceleration negative if board tilted to left

                mapping\_value = int(acceleration\_x \* SensorLightDisplay.\_\_neopixels\_amount//4 / SensorLightDisplay.\_\_acceleration\_peak)\*-1



                # calculate last neopixel above first neopixel to be lit (pixels 8 or 9, or 7 if only one pixel to be lit)

                last\_neopixel\_location\_a = first\_neopixel\_location + mapping\_value

                # calculate last neopixel above first neopixel to be lit (pixels 5 or 6, or 7 if only one pixel to be lit)

                last\_neopixel\_location\_b = first\_neopixel\_location - mapping\_value

                # loop for the range of pixels that can be lit (3)

                for pixel in range(first\_neopixel\_location, SensorLightDisplay.\_\_neopixels\_amount):



                    # if the pixel is within the pixels calculated

                    if pixel <= last\_neopixel\_location\_a and pixel >=last\_neopixel\_location\_b:

                        # colour the pixel purple (7, 8 or 9)

                        cp.pixels[pixel] = purple

                        # colour the corresponding pixel purple (7, 5 or 6)

                        cp.pixels[first\_neopixel\_location\*2-pixel] = purple

                        # show the coloured pixels

                        cp.pixels.show()

                    # if pixel not within the bounds

                    else:

                        # use the class variable to turn off the pixel

                        cp.pixels[pixel] = SensorLightDisplay.\_\_pixels\_off\_state

                        # and also turn off the corresponding pixel

                        cp.pixels[first\_neopixel\_location\*2-pixel] = SensorLightDisplay.\_\_pixels\_off\_state

                        # show the turned off pixels

                        cp.pixels.show()

            # if the acceleration not above 3 or below -3

            else:



                # turn all pixels off using the class variable

                cp.pixels.fill(SensorLightDisplay.\_\_pixels\_off\_state)

                # show the turned off pixels now

                cp.pixels.show()