Computer Science project.

Maksims Kovalovs

# Analysis

## Description of Current System

Most pool game are hosted on flash game websites like miniclip.com, and others, unfortunately these websites do not provide you with game files, and you have to play the game in your browser. This is bad not only because that does not allow for any customisation for the game files if the user wants to do so, but also players dont know how the physics of the game works, for example the players dont know the friction coefficients between the balls and the table which means that the game experience could deviate from the one the players used to in real life.

## Identification of End Users

The primary use for this project is for pool game enthusiasts who want the ability to customise their experience to their needs. The casual pool game players will enjoy the game as well, but these people already have a lot of options available to them online.

## User Needs

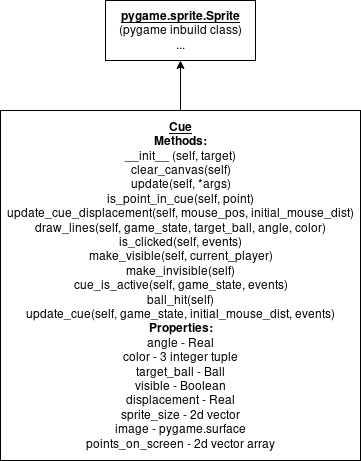
Make the constumisable options of the game like the friction coefficients easily accesible in the config file, so that even people without coding experience could adjust the values. The game should be open source so that people who are interested in the inner workings of the game could analyse it.

## Acceptable Limitations

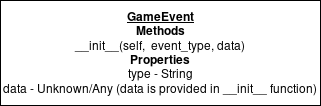
The purpose of the game is to provide a highly customisable experience, and thus the actual graphics of the game might be less appealing than other games online.

## Object diagram

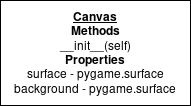
The project has a lot of different classes, and I have made object diagrams from every file in the project.



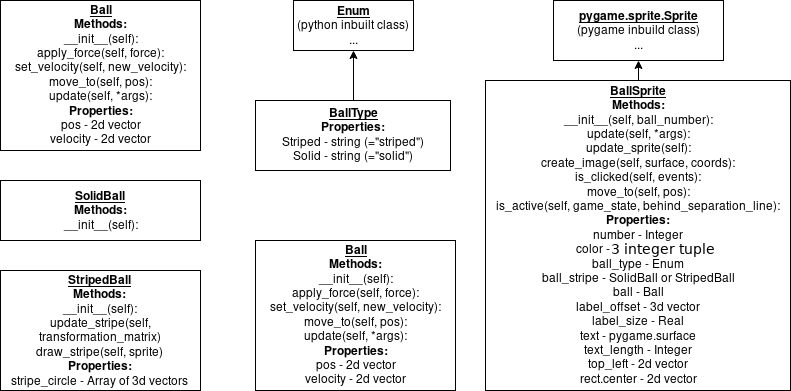
cue.py



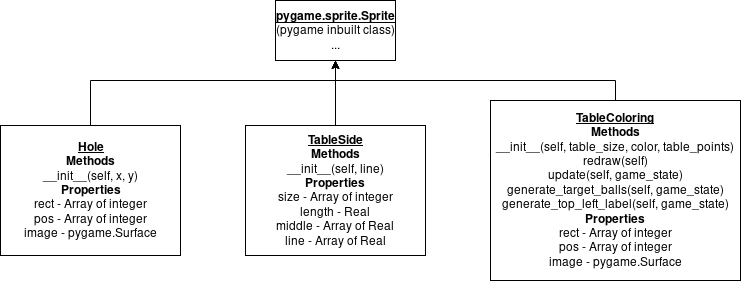
event.py



graphics.py



ball.py



table\_sprites.py

## Proposed Solution

Create an open source game with a configuration file to suit user needs. The program will implement classes for better code readability and maybe even some of the harder OOP ideas like overloading. The program will store the information about the state of the game, which will mostly consist of the coordinates and the velocity of the pool balls on the table. There will be 3 types of collision - ball to ball, ball to table side and ball to table hole. Every frame the game will check for these collisions, and if it detects any, it will resolve the collision based on the velocity and the position of the balls using linear algebra and vector operations. The game would be written in python and some additional libraries:

* Pygame
* Numpy
* Zope.Event

### Pygame

Python has a library called turtle that is part of the standard python installation, however it’s mainly for education purposes and does not nearly have the functionality for making a pool game. Here is an except from the pygame website Pygame (the library) is a Free and Open Source python programming language library for making multimedia applications like games built on top of the excellent SDL library. Like SDL, pygame is highly portable and runs on nearly every platform and operating system.

### Numpy

Numpy is a powerful tool for numerical calculations often involving matrices. In this project matrices are used for drawing the ball sprites, specifically 3d rotation matrices. ## Evidence of Analysis Physics behind the simulation The game will have to resolve elastic ball collisions and other physical processes. The balls will be modeled To understand the maths behind the collisions I will mainly use my knowledge from maths Mechanics 1 and 2 modules but I will also research the topic online. (<https://www.gamasutra.com/view/feature/131424/pool_hall_lessons_fast_accurate_.php> and <https://en.wikipedia.org/wiki/Elastic_collision> )

## Similar games currently available

I’ve studied some pool games online to get an idea what features I could add to my program, which are missing from the majority of games available online. The main problems with pool games online are: There is a plethora of games which are closed source, but it’s hard to find the source code online. By making the game open-source the users will gain the ability to study the inner- workings of the game. The code of the game can be used to study other things besides computers and code, for instance my game could be used to simulate famous Newtons cradle experiment, by removing friction. The games cannot be edited/configured. By limiting the ability of changing the inner- workings of the game the user might not get the experience he wants, for example the user might want to practise a specific kind of shot, do trick shots or change the size and the colour of the balls. By making an open-source game the user can change everything!

## Objectives

* Create a main menu at start of the game
* Allow the game to be played by two players
* The user can change the following game constants through the config file:
  + The size of the balls by changing the ball size integer in the Config file
  + The colour of the balls

## Critical Path

To make the development easier, im going to split the program in 4 stages: \* In the first stage I will implement the general shapes and objects - the table the balls. The balls will have position and velocity vectors but collisions at this stage of the game will be ignored \* In the second stage I will ball collisions with the table and with other balls. \* In the third stage I will add the cue stick and the table pockets. The physics of the game should be finished after this stage. \* In the fourth and final stage I will add the “game” parts of the game, namely, pool rules (stuff like player turns and whether you are allowed to move the white ball on your turn) and a main menu.

# Documented design

## Overall System Design

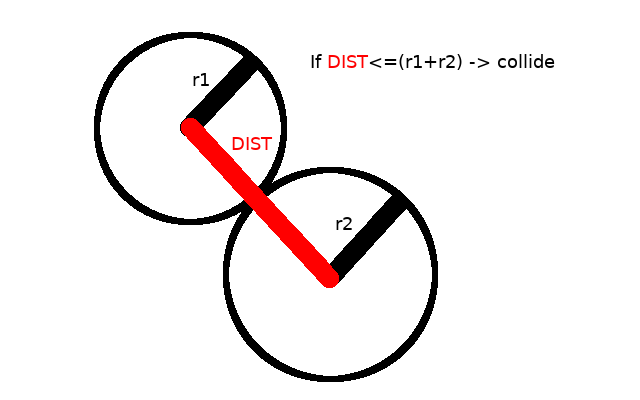
## Modular Design

* Main menu
  + The game
    - Stop the game
    - Game over
  + Quit the game

## Algorithms

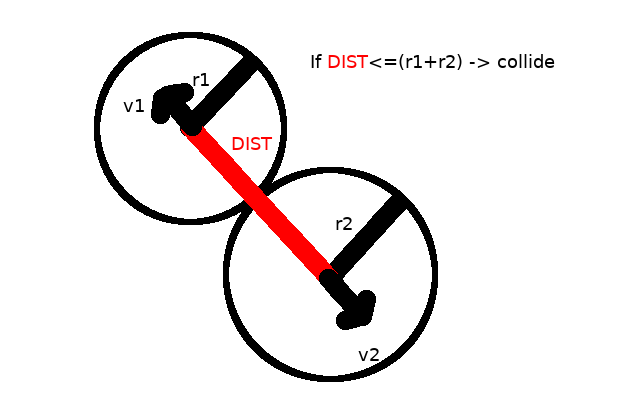
### Collision detection

One way detect a ball to ball collision is to check every frame the distances between every ball and, if the distance between two balls is less that the sum of their radii, that means that they have collided.



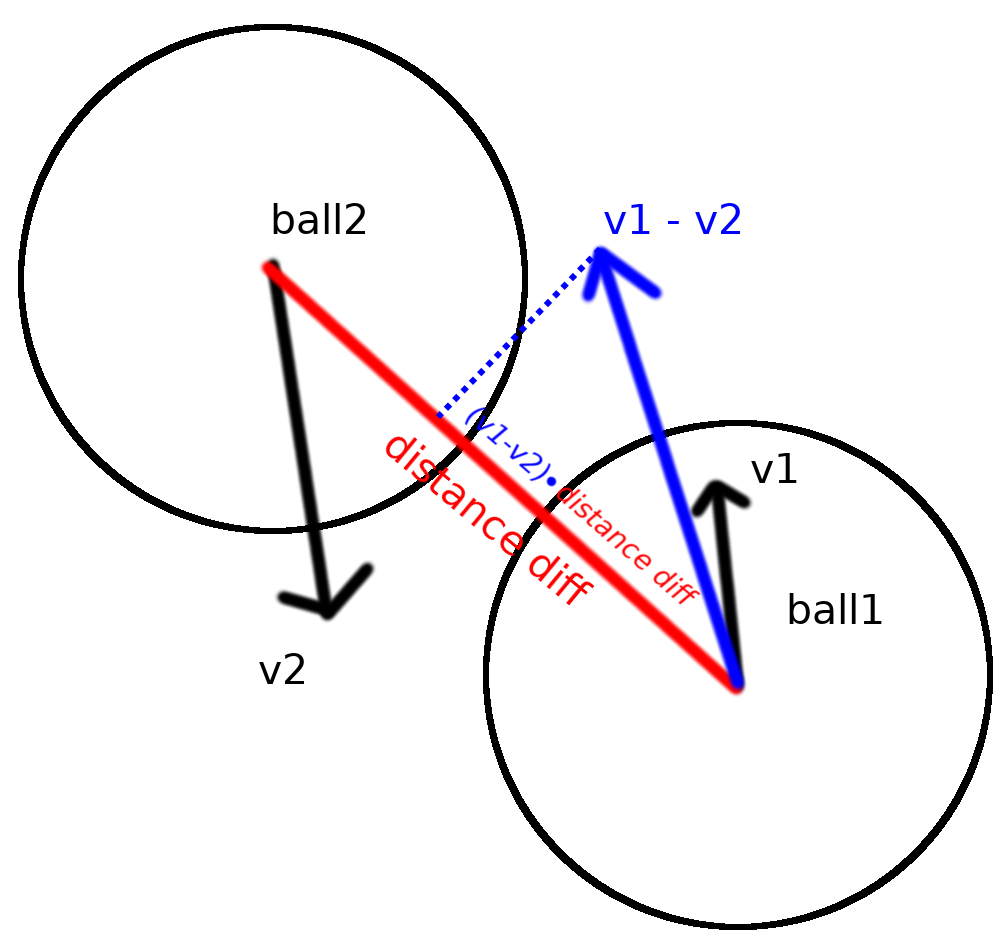
naive collision

Unfortunately, such a strategy sometimes makes the balls “stick” to each other. That is because this strategy doesn’t take into consideration the direction the balls are going. For example, in the previous picture, if the balls are going in the opposite directions, the balls shouldn’t collide because if they do, they are going to be moving towards each other after the collision, which isn’t right.



collision with velocities

We have to consider the directions of the balls velocities to detect collisions accurately. To do that we will take the dot product of the displacement difference between 2 balls[[1]](#footnote-47) and the velocity difference vector (labeled as v1-v2 on the figure). If the result is positive that means that the balls are moving towards each other and we can collide them.



collision with dot product

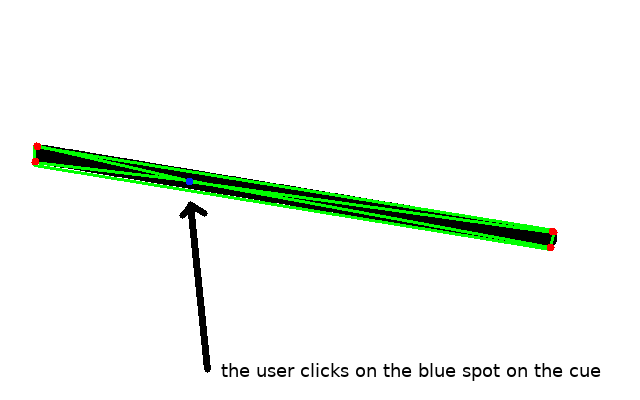
### Collision resolve

There are 2 types of collisions in the game - *ball to line* and *ball to ball* which are actually closely related. Both types are resolved using newtonian physics.

* Ball to line: When a ball collides with a line the perpendicular component of the balls velocity to the line is reversed (multiplied by -1). The component parralel to the line is unchanged. This is because the force from the wall on the ball is applied strictly perpendicularly to the line.
* Ball to ball: Using the law of the conservation of momemtum we do m1\*u1 + m2\*u2 = m1\*v1 + m2\*v2 Normaly to calculate the ball collision vectors you would have to consider kinetic energies and momentum to resolve properly, but if the masses of the balls are the same you could do a shortcut.
* m\*u1 + m\*u2 = m\*v1 + m\*v2  
  u1 + u2 = v1 + v2  
  v1 - u1 = -(v2 - u2)  
  Δv1 = -Δv2
* From this we see that any change in speed in the first ball has to be provided by the second ball, in other words the change in the velocity of the first ball will equal to change of the velocity of the second ball. When a ball collides with a ball the force on the balls goes through the collision point between the balls. Furthermore, since we are modeling elastic collisions we know that initial and final velocity will be equal, therefore the component of the balls velocity that goes through the collision point is flipped.

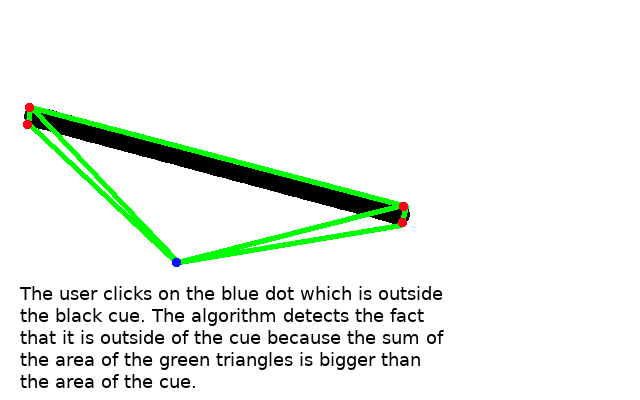
### Cue stick click detection

Because the cue stick is modeled as a rectangle, the problem here can be simplified down to figuring out if 2d point is in a rectangle (which is defined with 4 2d points). To do that we find the area of 4 triangles which can be made with 4 rectangle points and the points. We can find the sarea of triangles using herons formula.



click inside cue

The area of the green triangles equals the area of the cue, therefore the click is inside the cue.



click outside cue

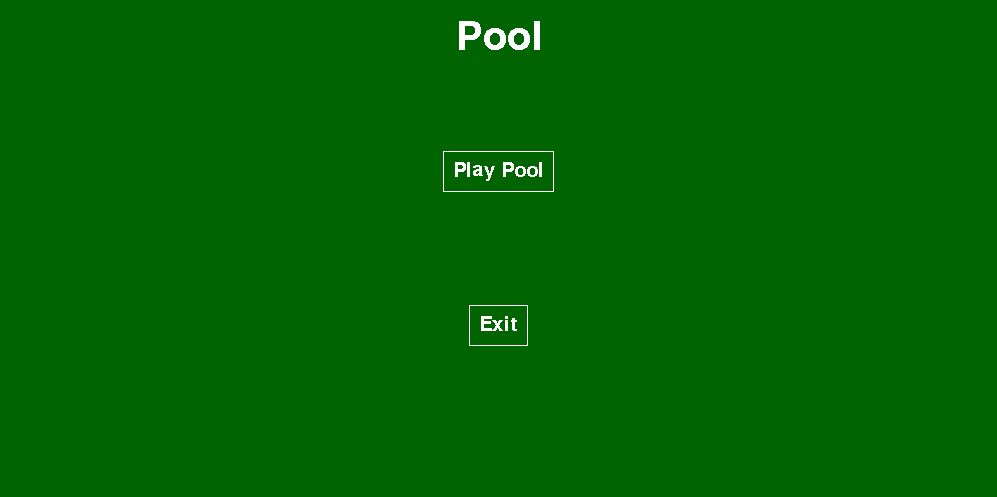
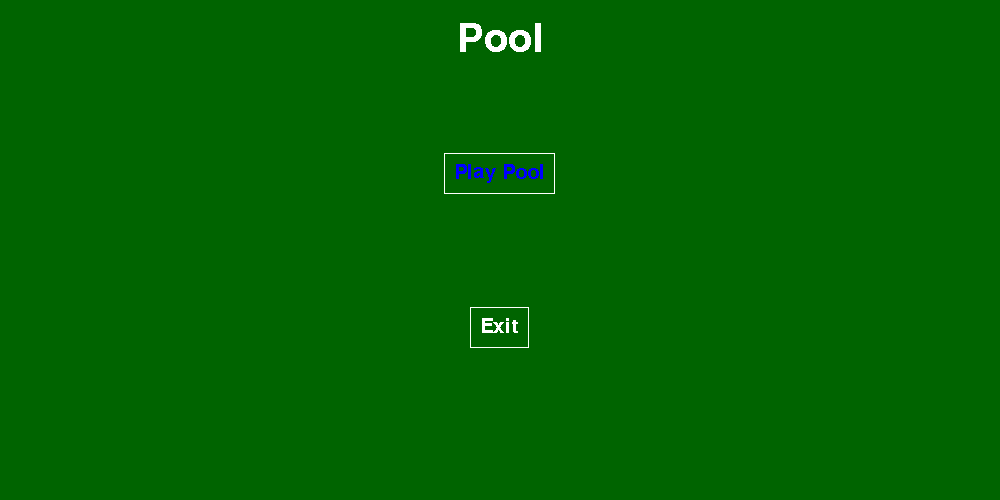
### The main game loop

Pseudocode:

While not\_closed:  
 button\_pressed\_in\_main\_menu = main\_menu\_get\_button()  
  
 if button\_pressed\_in\_main\_menu == "Play\_button":  
 While not\_closed or game\_is\_over():  
 resolve\_collisions()  
 do\_one\_frame()  
  
 If all\_balls\_not\_moving():  
 check\_pool\_rules()  
 use\_cue()  
  
 if button\_pressed\_in\_main\_menu == "Exit\_button":  
 not\_closed=False

This is the whole game split into functions. Every turn, one player uses the cue, then while the balls are moving, the game will check for any collisions, resolve them and do *one frame*, meaning it will move every moving object and redraw everything. The main game loop will continue doing that until the game is over or the game is closed.

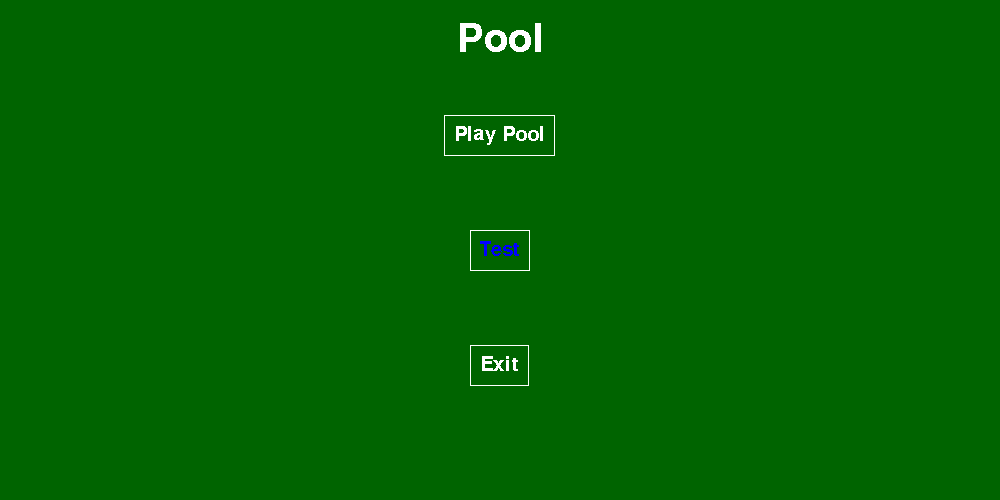
## User Interface

The game has a very minimal user interface. The game has a small main menu.  The menu only has two buttons - *Play Pool*, upon clicking which the game starts, and *Exit* which exits the game. The buttons change color when the player mouses over the button.  Even though the menu is very minimal, it is very customizable. Many things can be customized in the menu section in the config file (config.py).

...  
# menu  
menu\_text\_color = (255, 255, 255)  
menu\_text\_selected\_color = (0, 0, 255)  
menu\_title\_text = "Pool"  
menu\_buttons = ["Play Pool", "Exit"]  
menu\_margin = 20  
menu\_spacing = 10  
menu\_title\_font\_size = 40  
menu\_option\_font\_size = 20  
exit\_button = 2  
play\_game\_button = 1  
...

For example, to add a new button, you need to add a new string to menu\_buttons array. And change the exit\_button and the play\_game\_button variables. The variable exit\_button has the index of the “exit button” which, when is clicked, closes the game.

# menu  
...  
menu\_buttons = ["Play Pool", "Test", "Exit"]  
...  
exit\_button = 3  
play\_game\_button = 1  
...



main menu test

## Design considerations

Judging from other computer games, the pool game should include these things:

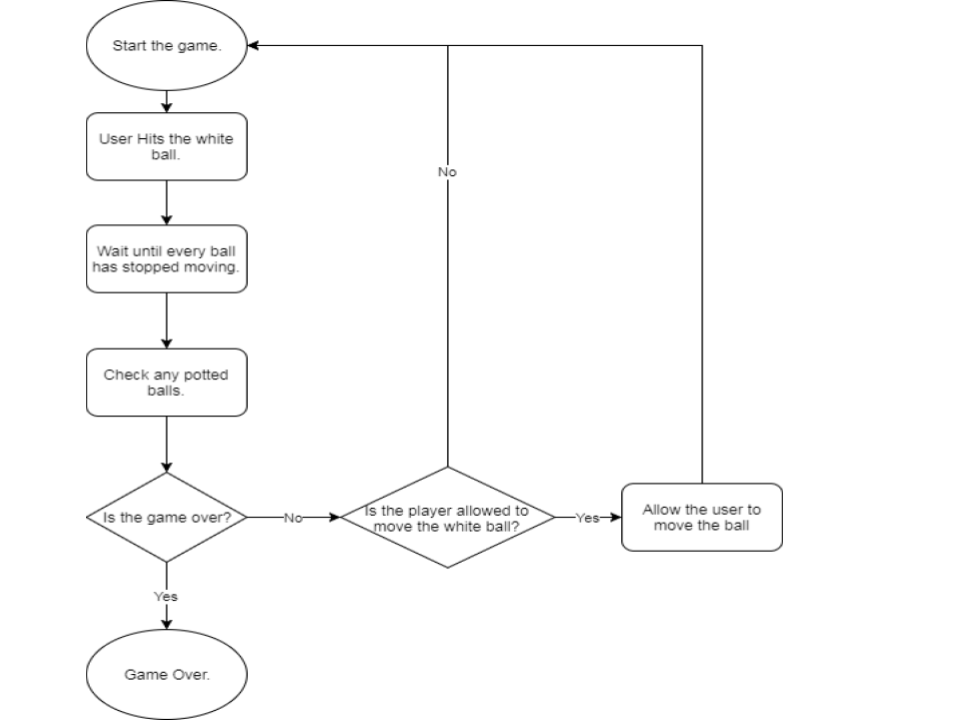
* The game screen should be big enough to see everything clearly.
* The balls should look realistic, and have easily distinguishable stripes on them so the players wouldn’t be confused which balls they need to pot.
* The game should have aiming lines to help the player aim. These lines are needed because in real life the player can line up the ball and see where it will go, so the game should have simillar functionality.
* The game should have a border so that the cue stick could be used even if the ball is in a courner.
* The game should have a minimal game menu.

## Input considerations

* The game should be almost fully controlled through the cue stick to hit the balls. The cue stick will be interactable via the mouse. An exception to the previous rule is if the white ball was potted and it must be moveable. In that case the ball can be interacted with via the mouse, specifically if the game rules allow the ball to be moved the player can click the ball and move it.
* In the menu the main input method is the mouse, however the user should be able to quit the game by pressing ESC key.
* While in game, the user should be able to press the ESC button to close the game.

## Processing considerations

* The balls will have a velocity and position vector. The velocity vector will be added to the position vector every second.
* There will be collision functions for ball to ball collisions and ball to wall collisions.
* The balls will have a function which will generate the ball sprite depending on the position and the movement of the ball.
* Functions which are responsible for checking that the game rules are not broken.



game flow

## File structure

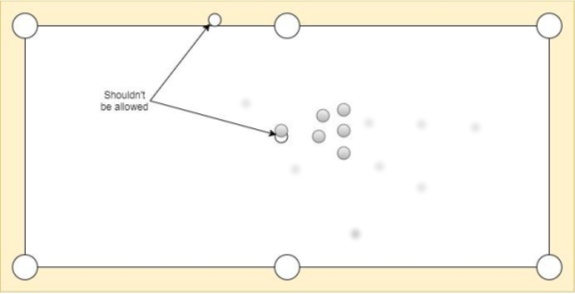
The project contains many files, all of them have a special purpose.

* main.py is the main game file which uses every other file to make the game run
* collision.py is the file responsible for detecting various collisions, like ball to ball and ball to table side collisions.
* ball.py contains the Ball class.
* gamestate.py contains the game state class, which has various functions related to actual game flow like the rules of the game.
* config.py the configuration file which contains most of the constants of the game Most classes are in separate classes for ease of reading.

Ball the ball class contains all of the methods associated with the pool ball, like sprite manipulation algorithms, position properties and others. Game State has various functions related to actual game flow like the rules of the game. Sprites table pieces like the table side and holes.

## Input Validation

In the game there is a special case where there is a need for extra validation. When the player is allowed to move the ball, we must make sure that the player doesn’t move the ball outside of the game border or into another ball.



bad ball placement

## Methods

# Technical solution

All of the code is available on github.com/max-kov/pool with testing results.

## collisions.py

import itertools  
import random  
  
import zope.event  
  
import config  
import event  
import physics  
  
  
def resolve\_all\_collisions(balls, holes, table\_sides):  
 # destroys any circles that are in a hole  
 for ball\_hole\_combination in itertools.product(balls, holes):  
 if physics.distance\_less\_equal(ball\_hole\_combination[0].ball.pos, ball\_hole\_combination[1].pos, config.hole\_radius):  
 zope.event.notify(event.GameEvent("POTTED", ball\_hole\_combination[0]))  
  
 # collides balls with the table where it is needed  
 for line\_ball\_combination in itertools.product(table\_sides, balls):  
 if physics.line\_ball\_collision\_check(line\_ball\_combination[0], line\_ball\_combination[1].ball):  
 physics.collide\_line\_ball(line\_ball\_combination[0], line\_ball\_combination[1].ball)  
  
 ball\_list = balls.sprites()  
 # ball list is shuffled to randomize ball collisions on the 1st break  
 random.shuffle(ball\_list)  
  
 for ball\_combination in itertools.combinations(ball\_list, 2):  
 if physics.ball\_collision\_check(ball\_combination[0].ball, ball\_combination[1].ball):  
 physics.collide\_balls(ball\_combination[0].ball, ball\_combination[1].ball)  
 zope.event.notify(event.GameEvent("COLLISION", ball\_combination))  
  
  
def check\_if\_ball\_touches\_balls(target\_ball\_pos, target\_ball\_number, balls):  
 touches\_other\_balls = False  
 for ball in balls:  
 if target\_ball\_number != ball.number and \  
 physics.distance\_less\_equal(ball.ball.pos, target\_ball\_pos, config.ball\_radius \* 2):  
 touches\_other\_balls = True  
 break  
 return touches\_other\_balls

## ball.py

import itertools  
import math  
from enum import Enum  
  
import numpy as np  
import pygame  
  
import collisions  
import config  
import event  
import physics  
  
  
class Ball():  
 def \_\_init\_\_(self):  
 self.pos = np.zeros(2, dtype=float)  
 self.velocity = np.zeros(2, dtype=float)  
  
 def apply\_force(self, force, time=1):  
 # f = ma, v = u + at -> v = u + (f/m)\*t  
 self.velocity += (force / config.ball\_mass) \* time  
  
 def set\_velocity(self, new\_velocity):  
 self.velocity = np.array(new\_velocity, dtype=float)  
  
 def move\_to(self, pos):  
 self.pos = np.array(pos, dtype=float)  
  
 def update(self, \*args):  
 self.velocity \*= config.friction\_coeff  
 self.pos += self.velocity  
  
 if np.hypot(\*self.velocity) < config.friction\_threshold:  
 self.velocity = np.zeros(2)  
  
  
class BallType(Enum):  
 Striped = "striped"  
 Solid = "solid"  
  
  
class StripedBall():  
 def \_\_init\_\_(self):  
 # every point is a 3d coordinate on the ball  
 # a circle will be drawn on the point if its Z component is >0 (is  
 # visible)  
 point\_num = config.ball\_stripe\_point\_num  
 self.stripe\_circle = config.ball\_radius \* np.column\_stack((np.cos(np.linspace(0, 2 \* np.pi, point\_num)),  
 np.sin(np.linspace(  
 0, 2 \* np.pi, point\_num)),  
 np.zeros(point\_num)))  
  
 def update\_stripe(self, transformation\_matrix):  
 for i, stripe in enumerate(self.stripe\_circle):  
 self.stripe\_circle[i] = np.matmul(  
 stripe, transformation\_matrix)  
  
 def draw\_stripe(self, sprite):  
 for num, point in enumerate(self.stripe\_circle[:-1]):  
 if point[2] >= -1:  
 pygame.draw.line(sprite, (255, 255, 255), config.ball\_radius + point[:2],  
 config.ball\_radius + self.stripe\_circle[num + 1][:2], config.ball\_stripe\_thickness)  
  
  
class SolidBall():  
 def \_\_init\_\_(self):  
 pass  
  
  
class BallSprite(pygame.sprite.Sprite):  
 def \_\_init\_\_(self, ball\_number):  
 self.number = ball\_number  
 self.color = config.ball\_colors[ball\_number]  
 if ball\_number <= 8:  
 self.ball\_type = BallType.Solid  
 self.ball\_stripe = SolidBall()  
 else:  
 self.ball\_type = BallType.Striped  
 self.ball\_stripe = StripedBall()  
 self.ball = Ball()  
 pygame.sprite.Sprite.\_\_init\_\_(self)  
 # initial location of the white circle and number on the ball, a.k.a  
 # ball label  
 self.label\_offset = np.array([0, 0, config.ball\_radius])  
 self.label\_size = config.ball\_radius // 2  
 font\_obj = config.get\_default\_font(config.ball\_label\_text\_size)  
 self.text = font\_obj.render(str(ball\_number), False, (0, 0, 0))  
 self.text\_length = np.array(font\_obj.size(str(ball\_number)))  
 self.update\_sprite()  
 self.update()  
 self.top\_left = self.ball.pos - config.ball\_radius  
 self.rect.center = self.ball.pos.tolist()  
  
 def update(self, \*args):  
 if np.hypot(\*self.ball.velocity) != 0:  
 # updates label circle and number offset  
 perpendicular\_velocity = -np.cross(self.ball.velocity, [0, 0, 1])  
 # angle formula is angle=((ballspeed\*2)/(pi\*r\*2))\*2  
 rotation\_angle = -np.hypot(  
 \*(self.ball.velocity)) \* 2 / (config.ball\_radius \* np.pi)  
 transformation\_matrix = physics.rotation\_matrix(  
 perpendicular\_velocity, rotation\_angle)  
 self.label\_offset = np.matmul(  
 self.label\_offset, transformation\_matrix)  
 if self.ball\_type == BallType.Striped:  
 self.ball\_stripe.update\_stripe(transformation\_matrix)  
 self.update\_sprite()  
 self.ball.update()  
  
 def update\_sprite(self):  
 sprite\_dimension = np.repeat([config.ball\_radius \* 2], 2)  
 new\_sprite = pygame.Surface(sprite\_dimension)  
 colorkey = (200, 200, 200)  
 new\_sprite.fill(self.color)  
 new\_sprite.set\_colorkey(colorkey)  
  
 label\_dimension = np.repeat([self.label\_size \* 2], 2)  
 label = pygame.Surface(label\_dimension)  
 label.fill(self.color)  
 # 1.1 instead of 1 is a hack to avoid 0 width sprite when scaling  
 dist\_from\_centre = 1.1 - (self.label\_offset[0] \*\* 2 +  
 self.label\_offset[1] \*\* 2) / (config.ball\_radius \*\* 2)  
  
 if self.label\_offset[2] > 0:  
 pygame.draw.circle(label, (255, 255, 255),  
 label\_dimension // 2, self.label\_size)  
  
 if self.number != 0:  
 label.blit(self.text, (config.ball\_radius - self.text\_length) / 2)  
  
 # hack to avoid div by zero  
 if self.label\_offset[0] != 0:  
 angle = -math.degrees(  
 math.atan(self.label\_offset[1] / self.label\_offset[0]))  
 label = pygame.transform.scale(  
 label, (int(config.ball\_radius \* dist\_from\_centre), config.ball\_radius))  
 label = pygame.transform.rotate(label, angle)  
  
 new\_sprite.blit(  
 label, self.label\_offset[:2] + (sprite\_dimension - label.get\_size()) / 2)  
 if self.ball\_type == BallType.Striped:  
 self.ball\_stripe.draw\_stripe(new\_sprite)  
  
 # applies a circular mask on the sprite using colorkey  
 grid\_2d = np.mgrid[-config.ball\_radius:config.ball\_radius +  
 1, -config.ball\_radius:config.ball\_radius + 1]  
 is\_outside = config.ball\_radius < np.hypot(\*grid\_2d)  
  
 for xy in itertools.product(range(config.ball\_radius \* 2 + 1), repeat=2):  
 if is\_outside[xy]:  
 new\_sprite.set\_at(xy, colorkey)  
  
 self.image = new\_sprite  
 self.rect = self.image.get\_rect()  
 self.top\_left = self.ball.pos - config.ball\_radius  
 self.rect.center = self.ball.pos.tolist()  
  
 def create\_image(self, surface, coords):  
 surface.blit(self.image, coords)  
  
 def is\_clicked(self, events):  
 return physics.distance\_less\_equal(events["mouse\_pos"], self.ball.pos, config.ball\_radius)  
  
 def move\_to(self, pos):  
 self.ball.move\_to(pos)  
 self.rect.center = self.ball.pos.tolist()  
  
 def is\_active(self, game\_state, behind\_separation\_line=False):  
 game\_state.cue.make\_invisible()  
 events = event.events()  
  
 while events["clicked"]:  
 events = event.events()  
 # checks if the user isn't trying to place the ball out of the table or inside another ball  
 if np.all(np.less(config.table\_margin + config.ball\_radius + config.hole\_radius, events["mouse\_pos"])) and \  
 np.all(np.greater(config.resolution - config.table\_margin - config.ball\_radius - config.hole\_radius,  
 events["mouse\_pos"])) and \  
 not collisions.check\_if\_ball\_touches\_balls(events["mouse\_pos"], self.number, game\_state.balls):  
 if behind\_separation\_line:  
 if events["mouse\_pos"][0] <= config.white\_ball\_initial\_pos[0]:  
 self.move\_to(events["mouse\_pos"])  
 else:  
 self.move\_to(events["mouse\_pos"])  
 game\_state.redraw\_all()  
 game\_state.cue.make\_visible(game\_state.current\_player)

## config.py

import math  
  
import numpy as np  
import pygame  
  
  
# fonts need to be initialised before using  
def get\_default\_font(size):  
 font\_defualt = pygame.font.get\_default\_font()  
 return pygame.font.Font(font\_defualt, size)  
  
  
def set\_max\_resolution():  
 infoObject = pygame.display.Info()  
 global resolution  
 global white\_ball\_initial\_pos  
 resolution = np.array([infoObject.current\_w, infoObject.current\_h])  
 white\_ball\_initial\_pos = (resolution + [table\_margin + hole\_radius, 0]) \* [0.25, 0.5]  
  
# window settings  
fullscreen = False  
# fullscreen resolution can only be known after initialising the screen  
if not fullscreen:  
 resolution = np.array([1000, 500])  
window\_caption = "Pool"  
fps\_limit = 60  
  
# table settings  
table\_margin = 40  
table\_side\_color = (200, 200, 0)  
table\_color = (0, 100, 0)  
separation\_line\_color = (200, 200, 200)  
hole\_radius = 22  
middle\_hole\_offset = np.array([[-hole\_radius \* 2, hole\_radius], [-hole\_radius, 0],  
 [hole\_radius, 0], [hole\_radius \* 2, hole\_radius]])  
side\_hole\_offset = np.array([  
 [- 2 \* math.cos(math.radians(45)) \* hole\_radius - hole\_radius, hole\_radius],  
 [- math.cos(math.radians(45)) \* hole\_radius, -  
 math.cos(math.radians(45)) \* hole\_radius],  
 [math.cos(math.radians(45)) \* hole\_radius,  
 math.cos(math.radians(45)) \* hole\_radius],  
 [- hole\_radius, 2 \* math.cos(math.radians(45)) \* hole\_radius + hole\_radius]  
])  
  
# cue settings  
player1\_cue\_color = (200, 100, 0)  
player2\_cue\_color = (0, 100, 200)  
cue\_hit\_power = 3  
cue\_length = 250  
cue\_thickness = 4  
cue\_max\_displacement = 100  
# safe displacement is the length the cue stick can be pulled before  
# causing the ball to move  
cue\_safe\_displacement = 1  
aiming\_line\_length = 14  
  
# ball settings  
total\_ball\_num = 16  
ball\_radius = 14  
ball\_mass = 14  
speed\_angle\_threshold = 0.09  
visible\_angle\_threshold = 0.05  
ball\_colors = [  
 (255, 255, 255),  
 (0, 200, 200),  
 (0, 0, 200),  
 (150, 0, 0),  
 (200, 0, 200),  
 (200, 0, 0),  
 (50, 0, 0),  
 (100, 0, 0),  
 (0, 0, 0),  
 (0, 200, 200),  
 (0, 0, 200),  
 (150, 0, 0),  
 (200, 0, 200),  
 (200, 0, 0),  
 (50, 0, 0),  
 (100, 0, 0)  
]  
ball\_stripe\_thickness = 5  
ball\_stripe\_point\_num = 25  
# where the balls will be placed at the start  
# relative to screen resolution  
ball\_starting\_place\_ratio = [0.75, 0.5]  
# in fullscreen mode the resolution is only available after initialising the screen  
# and if the screen wasn't initialised the resolution variable won't exist  
if 'resolution' in locals():  
 white\_ball\_initial\_pos = (resolution + [table\_margin + hole\_radius, 0]) \* [0.25, 0.5]  
ball\_label\_text\_size = 10  
  
# physics  
# if the velocity of the ball is less then  
# friction threshold then it is stopped  
friction\_threshold = 0.06  
friction\_coeff = 0.99  
  
# menu  
menu\_text\_color = (255, 255, 255)  
menu\_text\_selected\_color = (0, 0, 255)  
menu\_title\_text = "Pool"  
menu\_buttons = ["Play Pool", "Exit"]  
menu\_margin = 20  
menu\_spacing = 10  
menu\_title\_font\_size = 40  
menu\_option\_font\_size = 20  
exit\_button = 2  
play\_game\_button = 1  
  
# in-game ball target variables  
player1\_target\_text = 'P1 balls - '  
player2\_target\_text = 'P2 balls - '  
target\_ball\_spacing = 3  
player1\_turn\_label = "Player 1 turn"  
player2\_turn\_label = "Player 2 turn"  
penalty\_indication\_text = " (click on the ball to move it)"  
game\_over\_label\_font\_size = 40

## cue.py

import math  
  
import numpy as np  
import pygame  
  
import config  
import event  
import gamestate  
import physics  
  
  
class Cue(pygame.sprite.Sprite):  
 def \_\_init\_\_(self, target):  
 pygame.sprite.Sprite.\_\_init\_\_(self)  
 self.angle = 0  
 self.color = config.player1\_cue\_color  
 self.target\_ball = target  
 self.visible = False  
 self.displacement = config.ball\_radius  
 self.sprite\_size = np.repeat(  
 [config.cue\_length + config.cue\_max\_displacement], 2)  
 self.clear\_canvas()  
  
 def clear\_canvas(self):  
 # create empty surface as a placeholder for the cue  
 self.image = pygame.Surface(2 \* self.sprite\_size)  
 self.image.fill((200, 200, 200))  
 self.image.set\_colorkey((200, 200, 200))  
 self.rect = self.image.get\_rect()  
 self.rect.center = self.target\_ball.ball.pos.tolist()  
  
 def update(self, \*args):  
 if self.visible:  
 self.image = pygame.Surface(2 \* self.sprite\_size)  
 # color which will be ignored  
 self.image.fill((200, 200, 200))  
 self.image.set\_colorkey((200, 200, 200))  
  
 sin\_cos = np.array([math.sin(self.angle), math.cos(self.angle)])  
 initial\_coords = np.array([math.sin(self.angle + 0.5 \* math.pi), math.cos(self.angle +  
 0.5 \* math.pi)]) \* config.cue\_thickness  
 coord\_diff = sin\_cos \* config.cue\_length  
 rectangle\_points = np.array((initial\_coords, -initial\_coords,  
 -initial\_coords + coord\_diff, initial\_coords + coord\_diff))  
 rectangle\_points\_from\_circle = rectangle\_points + self.displacement \* sin\_cos  
 pygame.draw.polygon(self.image, self.color,  
 rectangle\_points\_from\_circle + self.sprite\_size)  
  
 self.points\_on\_screen = rectangle\_points\_from\_circle + self.target\_ball.ball.pos  
 self.rect = self.image.get\_rect()  
 self.rect.center = self.target\_ball.ball.pos.tolist()  
 else:  
 self.clear\_canvas()  
  
 def is\_point\_in\_cue(self, point):  
 # this algorithm splits up the rectangle into 4 triangles using the point provided  
 # if the point provided is inside the triangle the sum of triangle  
 # areas should be equal to that of the rectangle  
 rect\_sides = [config.cue\_thickness \* 2, config.cue\_length] \* 2  
 triangle\_sides = np.apply\_along\_axis(  
 physics.point\_distance, 1, self.points\_on\_screen, point)  
 calc\_area = np.vectorize(physics.triangle\_area)  
 triangle\_areas = np.sum(  
 calc\_area(triangle\_sides, np.roll(triangle\_sides, -1), rect\_sides))  
 rect\_area = rect\_sides[0] \* rect\_sides[1]  
 # +1 to prevent rounding errors  
 return rect\_area + 1 >= triangle\_areas  
  
 def update\_cue\_displacement(self, mouse\_pos, initial\_mouse\_dist):  
 displacement = physics.point\_distance(  
 mouse\_pos, self.target\_ball.ball.pos) - initial\_mouse\_dist + config.ball\_radius  
 if displacement > config.cue\_max\_displacement:  
 self.displacement = config.cue\_max\_displacement  
 elif displacement < config.ball\_radius:  
 self.displacement = config.ball\_radius  
 else:  
 self.displacement = displacement  
  
 def draw\_lines(self, game\_state, target\_ball, angle, color):  
 cur\_pos = np.copy(target\_ball.ball.pos)  
 diff = np.array([math.sin(angle), math.cos(angle)])  
  
 while config.resolution[1] > cur\_pos[1] > 0 and config.resolution[0] > cur\_pos[0] > 0:  
 cur\_pos += config.aiming\_line\_length \* diff \* 2  
 pygame.draw.line(game\_state.canvas.surface, color, cur\_pos,  
 (cur\_pos + config.aiming\_line\_length \* diff))  
  
 def is\_clicked(self, events):  
 return events["clicked"] and self.is\_point\_in\_cue(events["mouse\_pos"])  
  
 def make\_visible(self, current\_player):  
 if current\_player == gamestate.Player.Player1:  
 self.color = config.player1\_cue\_color  
 else:  
 self.color = config.player2\_cue\_color  
 self.visible = True  
 self.update()  
  
 def make\_invisible(self):  
 self.visible = False  
  
 def cue\_is\_active(self, game\_state, events):  
 initial\_mouse\_pos = events["mouse\_pos"]  
 initial\_mouse\_dist = physics.point\_distance(  
 initial\_mouse\_pos, self.target\_ball.ball.pos)  
  
 while events["clicked"]:  
 events = event.events()  
 self.update\_cue(game\_state, initial\_mouse\_dist, events)  
 # undraw leftover aiming lines  
 self.draw\_lines(game\_state, self.target\_ball, self.angle +  
 math.pi, config.table\_color)  
  
 if self.displacement > config.ball\_radius+config.cue\_safe\_displacement:  
 self.ball\_hit()  
  
 def ball\_hit(self):  
 new\_velocity = -(self.displacement - config.ball\_radius - config.cue\_safe\_displacement) \* \  
 config.cue\_hit\_power \* np.array([math.sin(self.angle), math.cos(self.angle)])  
 change\_in\_disp = np.hypot(\*new\_velocity) \* 0.1  
 while self.displacement - change\_in\_disp > config.ball\_radius:  
 self.displacement -= change\_in\_disp  
 self.update()  
 pygame.display.flip()  
 self.target\_ball.ball.apply\_force(new\_velocity)  
 self.displacement = config.ball\_radius  
 self.visible = False  
  
 def update\_cue(self, game\_state, initial\_mouse\_dist, events):  
 # updates cue position  
 current\_mouse\_pos = events["mouse\_pos"]  
 displacement\_from\_ball\_to\_mouse = self.target\_ball.ball.pos - current\_mouse\_pos  
 self.update\_cue\_displacement(current\_mouse\_pos, initial\_mouse\_dist)  
 prev\_angle = self.angle  
 # hack to avoid div by zero  
 if not displacement\_from\_ball\_to\_mouse[0] == 0:  
 self.angle = 0.5 \* math.pi - math.atan(  
 displacement\_from\_ball\_to\_mouse[1] / displacement\_from\_ball\_to\_mouse[0])  
 if displacement\_from\_ball\_to\_mouse[0] > 0:  
 self.angle -= math.pi  
  
 game\_state.redraw\_all(update=False)  
 self.draw\_lines(game\_state, self.target\_ball, prev\_angle +  
 math.pi, config.table\_color)  
 self.draw\_lines(game\_state, self.target\_ball, self.angle +  
 math.pi, (255, 255, 255))  
 pygame.display.flip()

## event.py

import numpy as np  
import pygame  
  
  
class GameEvent():  
 def \_\_init\_\_(self, event\_type, data):  
 self.type = event\_type  
 self.data = data  
  
  
def set\_allowed\_events():  
 # only allow keypress events to avoid waisting cpu type on checking useless events  
 pygame.event.set\_allowed([pygame.KEYDOWN, pygame.QUIT])  
  
def events():  
 closed = False  
 quit = False  
  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 closed = True  
 if event.type == pygame.KEYDOWN:  
 if event.key == pygame.K\_ESCAPE:  
 quit = True  
  
 return {"quit\_to\_main\_menu": quit,  
 "closed": closed,  
 "clicked": pygame.mouse.get\_pressed()[0],  
 "mouse\_pos": np.array(pygame.mouse.get\_pos())}

## gamestate.py

import itertools  
import math  
import random  
from enum import Enum  
  
import numpy as np  
import pygame  
import zope.event  
  
import ball  
import config  
import cue  
import event  
import graphics  
import table\_sprites  
from ball import BallType  
from collisions import check\_if\_ball\_touches\_balls  
  
  
class Player(Enum):  
 Player1 = 1  
 Player2 = 2  
  
  
class GameState:  
 def \_\_init\_\_(self):  
 pygame.init()  
 pygame.display.set\_caption(config.window\_caption)  
 event.set\_allowed\_events()  
 zope.event.subscribers.append(self.game\_event\_handler)  
 self.canvas = graphics.Canvas()  
 self.fps\_clock = pygame.time.Clock()  
  
 def fps(self):  
 return self.fps\_clock.get\_fps()  
  
 def mark\_one\_frame(self):  
 self.fps\_clock.tick(config.fps\_limit)  
  
 def create\_white\_ball(self):  
 self.white\_ball = ball.BallSprite(0)  
 ball\_pos = config.white\_ball\_initial\_pos  
 while check\_if\_ball\_touches\_balls(ball\_pos, 0, self.balls):  
 ball\_pos = [random.randint(int(config.table\_margin + config.ball\_radius + config.hole\_radius),  
 int(config.white\_ball\_initial\_pos[0])),  
 random.randint(int(config.table\_margin + config.ball\_radius + config.hole\_radius),  
 int(config.resolution[1] - config.ball\_radius - config.hole\_radius))]  
 self.white\_ball.move\_to(ball\_pos)  
 self.balls.add(self.white\_ball)  
 self.all\_sprites.add(self.white\_ball)  
  
 def game\_event\_handler(self, event):  
 if event.type == "POTTED":  
 self.table\_coloring.update(self)  
 self.balls.remove(event.data)  
 self.all\_sprites.remove(event.data)  
 self.potted.append(event.data.number)  
 elif event.type == "COLLISION":  
 if not self.white\_ball\_1st\_hit\_is\_set:  
 self.first\_collision(event.data)  
  
 def set\_pool\_balls(self):  
 counter = [0, 0]  
 coord\_shift = np.array([math.sin(math.radians(60)) \* config.ball\_radius \*  
 2, -config.ball\_radius])  
 initial\_place = config.ball\_starting\_place\_ratio \* config.resolution  
  
 self.create\_white\_ball()  
 # randomizes the sequence of balls on the table  
 ball\_placement\_sequence = list(range(1, config.total\_ball\_num))  
 random.shuffle(ball\_placement\_sequence)  
  
 for i in ball\_placement\_sequence:  
 ball\_iteration = ball.BallSprite(i)  
 ball\_iteration.move\_to(initial\_place + coord\_shift \* counter)  
 if counter[1] == counter[0]:  
 counter[0] += 1  
 counter[1] = -counter[0]  
 else:  
 counter[1] += 2  
 self.balls.add(ball\_iteration)  
  
 self.all\_sprites.add(self.balls)  
  
 def start\_pool(self):  
 self.reset\_state()  
 self.generate\_table()  
 self.set\_pool\_balls()  
 self.cue = cue.Cue(self.white\_ball)  
 self.all\_sprites.add(self.cue)  
  
 def reset\_state(self):  
 # game state variables  
 self.current\_player = Player.Player1  
 self.turn\_ended = True  
 self.white\_ball\_1st\_hit\_is\_set = False  
 self.potted = []  
 self.balls = pygame.sprite.Group()  
 self.holes = pygame.sprite.Group()  
 self.all\_sprites = pygame.sprite.OrderedUpdates()  
 self.turn\_number = 0  
 self.ball\_assignment = None  
 self.can\_move\_white\_ball = True  
 self.is\_game\_over = False  
 self.potting\_8ball = {Player.Player1: False, Player.Player2: False}  
 self.table\_sides = []  
  
 def is\_behind\_line\_break(self):  
 # 1st break should be made from behind the separation line on the table  
 return self.turn\_number == 0  
  
 def redraw\_all(self, update=True):  
 self.all\_sprites.clear(self.canvas.surface, self.canvas.background)  
 self.all\_sprites.draw(self.canvas.surface)  
 self.all\_sprites.update(self)  
 if update:  
 pygame.display.flip()  
 self.mark\_one\_frame()  
  
 def all\_not\_moving(self):  
 return\_value = True  
 for ball in self.balls:  
 if np.count\_nonzero(ball.ball.velocity) > 0:  
 return\_value = False  
 break  
 return return\_value  
  
 def generate\_table(self):  
 table\_side\_points = np.empty((1, 2))  
 # holes\_x and holes\_y holds the possible xs and ys of the table holes  
 # with a position ID in the second tuple field  
 # so the top left hole has id 1,1  
 holes\_x = [(config.table\_margin, 1), (config.resolution[0] /  
 2, 2), (config.resolution[0] - config.table\_margin, 3)]  
 holes\_y = [(config.table\_margin, 1),  
 (config.resolution[1] - config.table\_margin, 2)]  
 # next three lines are a hack to make and arrange the hole coordinates  
 # in the correct sequence  
 all\_hole\_positions = np.array(  
 list(itertools.product(holes\_y, holes\_x)))  
 all\_hole\_positions = np.fliplr(all\_hole\_positions)  
 all\_hole\_positions = np.vstack(  
 (all\_hole\_positions[:3], np.flipud(all\_hole\_positions[3:])))  
 for hole\_pos in all\_hole\_positions:  
 self.holes.add(table\_sprites.Hole(hole\_pos[0][0], hole\_pos[1][0]))  
 # this will generate the diagonal, vertical and horizontal table  
 # pieces which will reflect the ball when it hits the table sides  
 #  
 # they are generated using 4x2 offset matrices (4 2d points around the hole)  
 # with the first point in the matrix is the starting point and the  
 # last point is the ending point, these 4x2 matrices are  
 # concatenated together  
 #  
 # the martices must be flipped using numpy.flipud()  
 # after reflecting them using 2x1 reflection matrices, otherwise  
 # starting and ending points would be reversed  
 if hole\_pos[0][1] == 2:  
 # hole\_pos[0,1]=2 means x coordinate ID is 2 which means this  
 # hole is in the middle  
 offset = config.middle\_hole\_offset  
 else:  
 offset = config.side\_hole\_offset  
 if hole\_pos[1][1] == 2:  
 offset = np.flipud(offset) \* [1, -1]  
 if hole\_pos[0][1] == 1:  
 offset = np.flipud(offset) \* [-1, 1]  
 table\_side\_points = np.append(  
 table\_side\_points, [hole\_pos[0][0], hole\_pos[1][0]] + offset, axis=0)  
 # deletes the 1st point in array (leftover form np.empty)  
 table\_side\_points = np.delete(table\_side\_points, 0, 0)  
 for num, point in enumerate(table\_side\_points[:-1]):  
 # this will skip lines inside the circle  
 if num % 4 != 1:  
 self.table\_sides.append(table\_sprites.TableSide(  
 [point, table\_side\_points[num + 1]]))  
 self.table\_sides.append(table\_sprites.TableSide(  
 [table\_side\_points[-1], table\_side\_points[0]]))  
 self.table\_coloring = table\_sprites.TableColoring(  
 config.resolution, config.table\_side\_color, table\_side\_points)  
 self.all\_sprites.add(self.table\_coloring)  
 self.all\_sprites.add(self.holes)  
 graphics.add\_separation\_line(self.canvas)  
  
 def game\_over(self, p1\_won):  
 font = config.get\_default\_font(config.game\_over\_label\_font\_size)  
 if p1\_won:  
 text = "PLAYER 1 WON!"  
 else:  
 text = "PLAYER 2 WON!"  
 rendered\_text = font.render(text, False, (255, 255, 255))  
 self.canvas.surface.blit(rendered\_text, (config.resolution - font.size(text)) / 2)  
 pygame.display.flip()  
 pygame.event.clear()  
 paused = True  
 while paused:  
 event = pygame.event.wait()  
 if event.type == pygame.QUIT or event.type == pygame.KEYDOWN or event.type == pygame.MOUSEBUTTONDOWN:  
 paused = False  
 self.is\_game\_over = True  
  
 def turn\_over(self, penalize):  
 if not self.turn\_ended:  
 self.turn\_ended = True  
 self.turn\_number += 1  
 if self.current\_player == Player.Player1:  
 self.current\_player = Player.Player2  
 else:  
 self.current\_player = Player.Player1  
 if penalize:  
 self.can\_move\_white\_ball = True  
  
 def check\_potted(self):  
 self.can\_move\_white\_ball = False # if white ball is potted, it will be created again and placed in the middle  
 if 0 in self.potted:  
 self.create\_white\_ball()  
 self.cue.target\_ball = self.white\_ball  
 self.potted.remove(0)  
 self.turn\_over(True)  
 if 8 in self.potted:  
 if self.potting\_8ball[self.current\_player]:  
 self.game\_over(self.current\_player == Player.Player1)  
 else:  
 self.game\_over(self.current\_player == Player.Player1)  
  
 def check\_remaining(self):  
 # a check if all striped or solid balls were potted  
 stripes\_remaining = False  
 solids\_remaining = False  
 for remaining\_ball in self.balls:  
 if remaining\_ball.number != 0 and remaining\_ball.number != 8:  
 stripes\_remaining = stripes\_remaining or remaining\_ball.ball\_type == BallType.Striped  
 solids\_remaining = solids\_remaining or not remaining\_ball.ball\_type == BallType.Striped  
 ball\_type\_remaining = {BallType.Solid: solids\_remaining, BallType.Striped: stripes\_remaining}  
  
 # decides if on of the players (or both) should be potting 8ball  
 self.potting\_8ball = {Player.Player1: not ball\_type\_remaining[self.ball\_assignment[Player.Player1]],  
 Player.Player2: not ball\_type\_remaining[self.ball\_assignment[Player.Player2]]}  
  
 def first\_collision(self, ball\_combination):  
 self.white\_ball\_1st\_hit\_is\_set = True  
 self.white\_ball\_1st\_hit\_8ball = ball\_combination[0].number == 8 or ball\_combination[1].number == 8  
 if ball\_combination[0].number == 0:  
 self.white\_ball\_1st\_hit\_type = ball\_combination[1].ball\_type  
 else:  
 self.white\_ball\_1st\_hit\_type = ball\_combination[0].ball\_type  
  
 def check\_pool\_rules(self):  
 if self.ball\_assignment is not None:  
 self.check\_remaining()  
 self.check\_potted()  
 self.first\_hit\_rule()  
 self.potted\_ball\_rules()  
 self.on\_next\_hit()  
  
 def on\_next\_hit(self):  
 self.white\_ball\_1st\_hit\_is\_set = False  
 self.turn\_ended = False  
 self.potted = []  
  
 def potted\_ball\_rules(self):  
 if len(self.potted) > 0:  
 # if it wasnt decided which player goes for which type of balls  
 # and the player potted the balls exclusively of one color (excluting white balls)  
 # then it is decided based on which players turn it is right now and which type  
 # of balls he potted  
 potted\_stripe\_count = len([x for x in self.potted if x > 8])  
 potted\_solid\_count = len([x for x in self.potted if x < 8])  
 only\_stripes\_potted = potted\_solid\_count == 0 and potted\_stripe\_count > 0  
 only\_solids\_potted = potted\_stripe\_count == 0 and potted\_solid\_count > 0  
  
 if only\_solids\_potted or only\_stripes\_potted:  
 selected\_ball\_type = BallType.Striped if only\_stripes\_potted else BallType.Solid  
 if self.ball\_assignment is None:  
 # unpacking a singular set - SO MACH HACK  
 other\_player, = set(Player) - {self.current\_player}  
 other\_ball\_type, = set(BallType) - {selected\_ball\_type}  
 self.ball\_assignment = {self.current\_player: selected\_ball\_type, other\_player: other\_ball\_type}  
 self.potting\_8ball = {self.current\_player: False, other\_player: False}  
 elif self.ball\_assignment[self.current\_player] != selected\_ball\_type:  
 self.turn\_over(False)  
 else:  
 self.turn\_over(False)  
  
 def first\_hit\_rule(self):  
 # checks if the 1st white ball hit is the same as the players target ball type  
 # for example if the current player hits a striped ball with the whit ball  
 # but he should be potting solid balls, it is next players turn and he can move the white ball  
 if not self.white\_ball\_1st\_hit\_is\_set:  
 self.turn\_over(True)  
 elif self.ball\_assignment is not None:  
 if not self.white\_ball\_1st\_hit\_8ball and self.ball\_assignment[  
 self.current\_player] != self.white\_ball\_1st\_hit\_type:  
 self.turn\_over(True)  
 # checks if the 8ball was the first ball hit, and if so checks if the player needs to pot the 8ball  
 # and if not he gets penalised  
 elif self.white\_ball\_1st\_hit\_8ball:  
 self.turn\_over(not self.potting\_8ball[self.current\_player])

## graphics.py

import numpy as np  
import pygame  
  
import config  
import event  
  
  
class Canvas:  
 def \_\_init\_\_(self):  
 if config.fullscreen:  
 config.set\_max\_resolution()  
 self.surface = pygame.display.set\_mode(config.resolution, pygame.FULLSCREEN)  
 else:  
 self.surface = pygame.display.set\_mode(config.resolution)  
 self.background = pygame.Surface(self.surface.get\_size())  
 self.background = self.background.convert()  
 self.background.fill(config.table\_color)  
 self.surface.blit(self.background, (0, 0))  
  
  
def add\_separation\_line(canvas):  
 # white ball separation line  
 pygame.draw.line(canvas.background, config.separation\_line\_color, (config.white\_ball\_initial\_pos[0], 0),  
 (config.white\_ball\_initial\_pos[0], config.resolution[1]))  
  
  
def create\_buttons(text, text\_font, text\_color\_normal, text\_color\_on\_hover):  
 # this function generates button objects using button text, button font, normal colour and color of the text when  
 # the mouse is hovering over the button  
  
 button\_size = np.array([text\_font[num].size(text[num]) for num in range(len(text))])  
 # generating button objects  
 buttons = [  
 # text when mouse is outside the button range  
 [text\_font[num].render(text[num], False, text\_color\_normal[num]),  
 # text when mouse is inside the button range  
 text\_font[num].render(text[num], False, text\_color\_on\_hover[num])]  
 for num in range(len(text))]  
  
 screen\_mid = config.resolution[0] / 2  
 change\_in\_y = (config.resolution[1] -  
 config.menu\_margin \* 2) / (len(buttons))  
 screen\_button\_middles = np.stack((np.repeat([screen\_mid], len(buttons)),  
 np.arange(len(buttons)) \* change\_in\_y), axis=1)  
  
 text\_starting\_place = screen\_button\_middles + [-0.5, 0.5] \* button\_size  
 text\_ending\_place = text\_starting\_place + button\_size  
  
 return buttons, button\_size, text\_starting\_place, text\_ending\_place  
  
  
def draw\_main\_menu(game\_state):  
 buttons, button\_size, text\_starting\_place, text\_ending\_place = create\_buttons(  
 [config.menu\_title\_text] + config.menu\_buttons,  
 [config.get\_default\_font(config.menu\_title\_font\_size)] + [  
 config.get\_default\_font(config.menu\_option\_font\_size)] \* 3,  
 [config.menu\_text\_color] \* 4,  
 [config.menu\_text\_color] + [config.menu\_text\_selected\_color] \* 3)  
 draw\_rects(button\_size, buttons, game\_state, text\_starting\_place, emit=[0])  
 button\_clicked = iterate\_until\_button\_press(buttons, game\_state, text\_ending\_place,  
 text\_starting\_place)  
  
 return button\_clicked  
  
  
def iterate\_until\_button\_press(buttons, game\_state, text\_ending\_place, text\_starting\_place):  
 # while a button was not clicked this method checks if mouse is in the button and if it is  
 # changes its colour  
 button\_clicked = 0  
 while button\_clicked == 0:  
 pygame.display.update()  
 user\_events = event.events()  
 # the first button is the title which is unclickable, thus iterating from 1 to len(buttons)  
 for num in range(1, len(buttons)):  
 if np.all((np.less(text\_starting\_place[num] - config.menu\_spacing, user\_events["mouse\_pos"]),  
 np.greater(text\_ending\_place[num] + config.menu\_spacing, user\_events["mouse\_pos"]))):  
 if user\_events["clicked"]:  
 button\_clicked = num  
 else:  
 game\_state.canvas.surface.blit(  
 buttons[num][1], text\_starting\_place[num])  
 else:  
 game\_state.canvas.surface.blit(  
 buttons[num][0], text\_starting\_place[num])  
 if user\_events["closed"] or user\_events["quit\_to\_main\_menu"]:  
 button\_clicked = len(buttons)-1  
 return button\_clicked  
  
  
def draw\_rects(button\_size, buttons, game\_state, text\_starting\_place, emit=list()):  
 # drawing a rectangle around the button text  
 for num in range(len(buttons)):  
 game\_state.canvas.surface.blit(  
 buttons[num][0], text\_starting\_place[num])  
 # emit contains indexes of buttons which do not need a rectangle around them  
 if not num in emit:  
 pygame.draw.rect(game\_state.canvas.surface, config.menu\_text\_color,  
 np.concatenate((text\_starting\_place[num] -  
 config.menu\_spacing, button\_size[num] +  
 config.menu\_spacing \* 2)), 1)

## main.py

import pygame  
  
import collisions  
import event  
import gamestate  
import graphics  
import config  
  
was\_closed = False  
while not was\_closed:  
 game = gamestate.GameState()  
 button\_pressed = graphics.draw\_main\_menu(game)  
  
 if button\_pressed == config.play\_game\_button:  
 game.start\_pool()  
 events = event.events()  
 while not (events["closed"] or game.is\_game\_over or events["quit\_to\_main\_menu"]):  
 events = event.events()  
 collisions.resolve\_all\_collisions(game.balls, game.holes, game.table\_sides)  
 game.redraw\_all()  
  
 if game.all\_not\_moving():  
 game.check\_pool\_rules()  
 game.cue.make\_visible(game.current\_player)  
 while not (  
 (events["closed"] or events["quit\_to\_main\_menu"]) or game.is\_game\_over) and game.all\_not\_moving():  
 game.redraw\_all()  
 events = event.events()  
 if game.cue.is\_clicked(events):  
 game.cue.cue\_is\_active(game, events)  
 elif game.can\_move\_white\_ball and game.white\_ball.is\_clicked(events):  
 game.white\_ball.is\_active(game, game.is\_behind\_line\_break())  
 was\_closed = events["closed"]  
  
 if button\_pressed == config.exit\_button:  
 was\_closed = True  
  
pygame.quit()

## physics.py

import math  
  
import numpy as np  
  
import config  
  
  
def point\_distance(p1, p2):  
 dist\_diff = p1 - p2  
 return np.hypot(\*dist\_diff)  
  
  
def distance\_less\_equal(p1, p2, dist):  
 # does distance comparisons without calculating square roots  
 dist\_diff = p1 - p2  
 return (dist\_diff[0] \*\* 2 + dist\_diff[1] \*\* 2) <= dist \*\* 2  
  
  
def ball\_collision\_check(ball1, ball2):  
 # distance check followed by checking if either of the balls are moving  
 # followed by vector projection check, to see if both are moving towards  
 # each other  
 return distance\_less\_equal(ball1.pos, ball2.pos, 2 \* config.ball\_radius) and \  
 np.count\_nonzero(np.concatenate((ball1.velocity, ball2.velocity))) > 0 and \  
 np.dot(ball2.pos - ball1.pos, ball1.velocity - ball2.velocity) > 0  
  
  
def collide\_balls(ball1, ball2):  
 point\_diff = ball2.pos - ball1.pos  
 dist = point\_distance(ball1.pos, ball2.pos)  
 # normalising circle distance difference vector  
 collision = point\_diff / dist  
 # projecting balls velocity ONTO difference vector  
 ball1\_dot = np.dot(ball1.velocity, collision)  
 ball2\_dot = np.dot(ball2.velocity, collision)  
 # since the masses of the balls are the same, the velocity will just switch  
 ball1.velocity += (ball2\_dot - ball1\_dot) \* collision  
 ball2.velocity += (ball1\_dot - ball2\_dot) \* collision  
  
  
def triangle\_area(side1, side2, side3):  
 # used to determine if the user is clicking on the cue stick  
 # herons formula  
 half\_perimetre = abs((side1 + side2 + side3) \* 0.5)  
 return math.sqrt(half\_perimetre \* (half\_perimetre - abs(side1)) \* (half\_perimetre - abs(side2)) \* (  
 half\_perimetre - abs(side3)))  
  
  
def rotation\_matrix(axis, theta):  
 # Return the rotation matrix associated with counterclockwise rotation about  
 # the given axis by theta radians.  
 axis = np.asarray(axis)  
 axis = axis / math.sqrt(np.dot(axis, axis))  
 a = math.cos(theta / 2.0)  
 b, c, d = -axis \* math.sin(theta / 2.0)  
 aa, bb, cc, dd = a \* a, b \* b, c \* c, d \* d  
 bc, ad, ac, ab, bd, cd = b \* c, a \* d, a \* c, a \* b, b \* d, c \* d  
 return np.array([[aa + bb - cc - dd, 2 \* (bc + ad), 2 \* (bd - ac)],  
 [2 \* (bc - ad), aa + cc - bb - dd, 2 \* (cd + ab)],  
 [2 \* (bd + ac), 2 \* (cd - ab), aa + dd - bb - cc]])  
  
  
def line\_ball\_collision\_check(line, ball):  
 # checks if the ball is half the line length from the line middle  
 if distance\_less\_equal(line.middle, ball.pos, line.length / 2 + config.ball\_radius):  
 # displacement vector from the first point to the ball  
 displacement\_to\_ball = ball.pos - line.line[0]  
 # displacement vector from the first point to the second point on the  
 # line  
 displacement\_to\_second\_point = line.line[1] - line.line[0]  
 normalised\_point\_diff\_vector = displacement\_to\_second\_point / \  
 np.hypot(\*(displacement\_to\_second\_point))  
 # distance from the first point on the line to the perpendicular  
 # projection point from the ball  
 projected\_distance = np.dot(normalised\_point\_diff\_vector, displacement\_to\_ball)  
 # closest point on the line to the ball  
 closest\_line\_point = projected\_distance \* normalised\_point\_diff\_vector  
 perpendicular\_vector = np.array(  
 [-normalised\_point\_diff\_vector[1], normalised\_point\_diff\_vector[0]])  
 # checking if closest point on the line is actually on the line (which is not always the case when projecting)  
 # then checking if the distance from that point to the ball is less than the balls radius and finally  
 # checking if the ball is moving towards the line with the dot product  
 return -config.ball\_radius / 3 <= projected\_distance <= \  
 np.hypot(\*(displacement\_to\_second\_point)) + config.ball\_radius / 3 and \  
 np.hypot(\*(closest\_line\_point - ball.pos + line.line[0])) <= \  
 config.ball\_radius and np.dot(  
 perpendicular\_vector, ball.velocity) <= 0  
  
  
def collide\_line\_ball(line, ball):  
 displacement\_to\_second\_point = line.line[1] - line.line[0]  
 normalised\_point\_diff\_vector = displacement\_to\_second\_point / \  
 np.hypot(\*(displacement\_to\_second\_point))  
 perpendicular\_vector = np.array(  
 [-normalised\_point\_diff\_vector[1], normalised\_point\_diff\_vector[0]])  
 ball.velocity -= 2 \* np.dot(perpendicular\_vector,  
 ball.velocity) \* perpendicular\_vector

## table\_sprites.py

import numpy as np  
import pygame  
  
import config  
import gamestate  
  
  
class Hole(pygame.sprite.Sprite):  
 def \_\_init\_\_(self, x, y):  
 pygame.sprite.Sprite.\_\_init\_\_(self)  
 self.image = pygame.Surface(  
 (2 \* config.hole\_radius, 2 \* config.hole\_radius))  
 # color which will be ignored  
 self.image.fill((200, 200, 200))  
 self.image.set\_colorkey((200, 200, 200))  
  
 pygame.draw.circle(self.image, (0, 0, 0),  
 (config.hole\_radius, config.hole\_radius), config.hole\_radius, 0)  
 self.rect = self.image.get\_rect()  
 self.rect.center = (x, y)  
 self.pos = np.array([x, y])  
  
  
# this class holds properties of a table side line, but doesn't actually  
# draw it  
class TableSide():  
 def \_\_init\_\_(self, line):  
 self.line = np.array(line)  
 self.middle = (self.line[0] + self.line[1]) / 2  
 self.size = np.round(np.abs(self.line[0] - self.line[1]))  
 self.length = np.hypot(\*self.size)  
 if np.count\_nonzero(self.size) != 2:  
 # line is perpendicular to y or x axis  
 if self.size[0] == 0:  
 self.size[0] += 1  
 else:  
 self.size[1] += 1  
  
  
# draws the yellow part of the table  
class TableColoring(pygame.sprite.Sprite):  
 def \_\_init\_\_(self, table\_size, color, table\_points):  
 pygame.sprite.Sprite.\_\_init\_\_(self)  
 self.points = table\_points  
 self.image = pygame.Surface(table\_size)  
 self.image.fill(color)  
 color\_key = (200, 200, 200)  
 self.image.set\_colorkey(color\_key)  
 pygame.draw.polygon(self.image, color\_key, table\_points)  
 self.rect = self.image.get\_rect()  
 self.rect.topleft = (0, 0)  
 self.font = config.get\_default\_font(config.ball\_radius)  
 # generates text at the bottom of the table  
 self.target\_ball\_text = [self.font.render(config.player1\_target\_text, False, config.player1\_cue\_color),  
 self.font.render(config.player2\_target\_text, False, config.player2\_cue\_color)]  
  
 def redraw(self):  
 self.image.fill(config.table\_side\_color)  
 color\_key = (200, 200, 200)  
 self.image.set\_colorkey(color\_key)  
 pygame.draw.polygon(self.image, color\_key, self.points)  
  
 def update(self, game\_state):  
 self.redraw()  
 self.generate\_top\_left\_label(game\_state)  
 self.generate\_target\_balls(game\_state)  
  
 def generate\_target\_balls(self, game\_state):  
 # draws the target balls for each players  
 if game\_state.ball\_assignment is not None:  
 start\_x = np.array([config.table\_margin + config.hole\_radius \* 3,  
 config.resolution[0] / 2 + config.hole\_radius \* 3])  
 start\_y = config.resolution[1] - config.table\_margin - self.font.size(config.player1\_target\_text)[1] / 2  
 # the text needs to be moved a bit lower to keep it aligned  
 self.image.blit(self.target\_ball\_text[0], [start\_x[0], start\_y + config.ball\_radius / 2])  
 self.image.blit(self.target\_ball\_text[1], [start\_x[1], start\_y + config.ball\_radius / 2])  
 start\_x += self.font.size(config.player2\_target\_text)[0]  
 for ball in game\_state.balls:  
 do\_draw = ball.number != 0 and ball.number != 8  
  
 # draw to player holds the players which the balls will be added to  
 draw\_to\_player = []  
  
 # sorts the balls into their places  
 if do\_draw:  
 if game\_state.ball\_assignment[gamestate.Player.Player1] == ball.ball\_type:  
 draw\_to\_player.append(1)  
 else:  
 draw\_to\_player.append(2)  
  
 if ball.number == 8:  
 if game\_state.potting\_8ball[gamestate.Player.Player1]:  
 draw\_to\_player.append(1)  
 if game\_state.potting\_8ball[gamestate.Player.Player2]:  
 draw\_to\_player.append(2)  
  
 # draws the balls  
 for player in draw\_to\_player:  
 # player-1 because lists start with 0  
 ball.create\_image(self.image, (start\_x[player - 1], start\_y))  
 start\_x[player - 1] += config.ball\_radius \* 2 + config.target\_ball\_spacing  
  
 def generate\_top\_left\_label(self, game\_state):  
 # generates the top left label (which players turn is it and if he can move the ball)  
 top\_left\_text = ""  
 if game\_state.can\_move\_white\_ball:  
 top\_left\_text += config.penalty\_indication\_text  
 if game\_state.current\_player.value == 1:  
 top\_left\_rendered\_text = self.font.render(config.player1\_turn\_label + top\_left\_text,  
 False, config.player1\_cue\_color)  
 else:  
 top\_left\_rendered\_text = self.font.render(config.player2\_turn\_label + top\_left\_text,  
 False, config.player2\_cue\_color)  
 text\_pos = [config.table\_margin + config.hole\_radius \* 3,  
 config.table\_margin - self.font.size(top\_left\_text)[1] / 2]  
 self.image.blit(top\_left\_rendered\_text, text\_pos)

# Testing

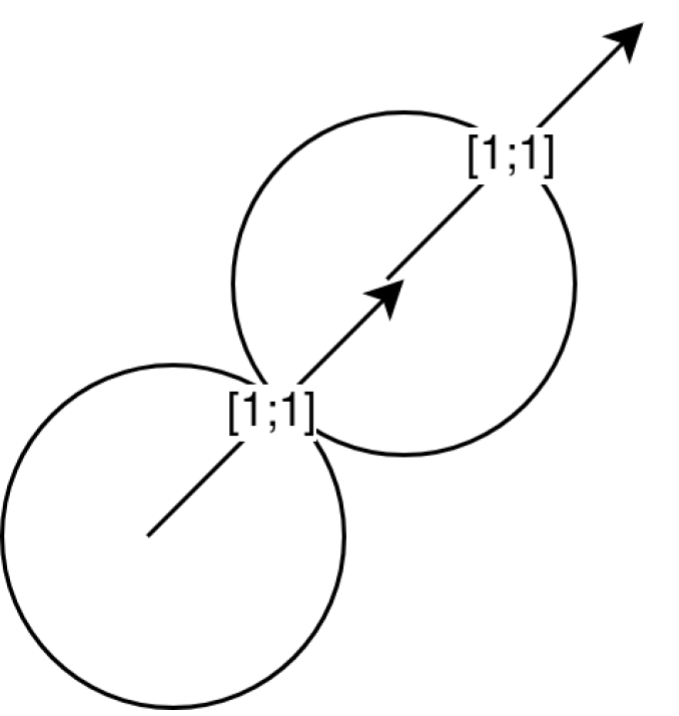
## White box testing (Unit testing)

In my game there are two main things to test - the process of the physical simulation, meaning the mathematics behind the collisions, collision detection and others, and the user interface. To test the mathematics, I’ve made a set of tests in a form of a script, which, when executed, runs the main program with the test input data. These tests are considered white box testing, and, more specificaly, unit testing, because I am testing specific functions of the program (which I wrote myself). To make the job of testing the game with every change easier, I have made unit tests which are executed on my code every time I make a commit to github. The unit tests are included with the game in the folder “tests”, the tests contain some input, which is used with the functions of the game which are responsible for a specific task. Lets look at a particular example:

def test\_point\_distance1(self):  
 assert physics.point\_distance(np.array([0, 0]), np.array([3, 4])) == 5

This function tests the function responsible for calculating the distance between two points. The function uses the function to calculate the distance between points (0,0) and (3,4), and compares it to 5. If the function does not return 5, the test returns an exception. Here is another, more complicated example of an edge case test:

def test\_movement4(self):  
 ball1.set\_velocity((1, 1))  
 ball2.move\_to(-fortyfive\_degree\_position)  
 ball2.set\_velocity((1, 1))  
 assert not physics.ball\_collision\_check(ball1, ball2)



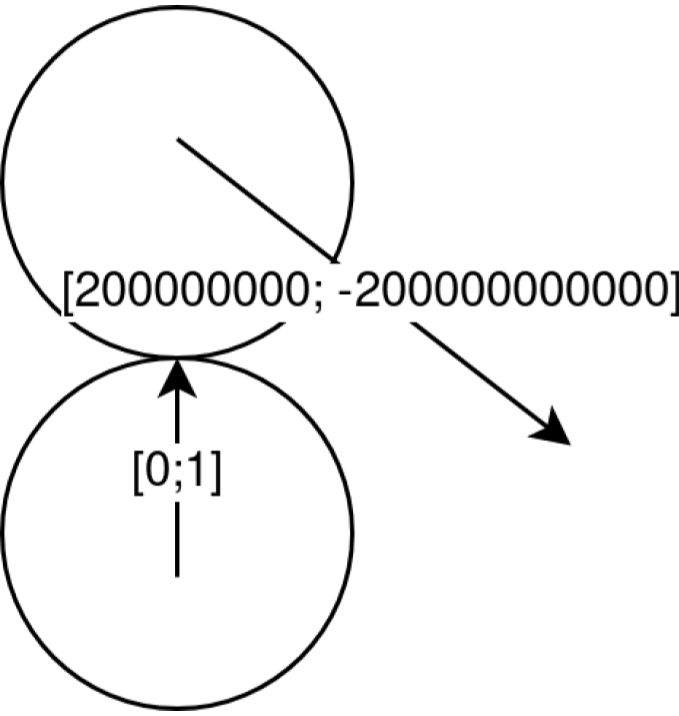
test1

The test sets the balls such that they are touching, however, both balls are moving with the same velocity. That means there is no collision.

This is an example of an extreme test:

def test\_movement10(self):  
 ball1.set\_velocity((0, 1))  
 ball2.move\_to((0, ball\_radius \* 2))  
 ball2.set\_velocity((200000000, -200000000000))  
 assert physics.ball\_collision\_check(ball1, ball2)

The values in the test are exaggerated so that we know that the game can handle extreme values.



test2

### To run the tests

The file contains many tests, including matrix transformation tests, line to ball collision tests and the triangle area test. Here is the file itself:

import itertools  
import math  
  
import numpy as np  
  
from pool import ball  
from pool import physics  
from pool import table\_sprites  
from pool.config import ball\_radius  
  
  
class TestPointDistance():  
 def test\_point\_distance1(self):  
 assert physics.point\_distance(np.array([0, 0]), np.array([3, 4])) == 5  
  
 def test\_point\_distance2(self):  
 assert physics.point\_distance(  
 np.array([0, -10]), np.array([0, -10])) == 0  
  
 def test\_point\_distance3(self):  
 assert physics.point\_distance(  
 np.array([10, 0]), np.array([-10, 0])) == 20  
  
  
class TestTriangleArea():  
 def test\_triangle\_area1(self):  
 assert physics.triangle\_area(1, 1, 0) == 0  
  
 def test\_triangle\_area2(self):  
 assert physics.triangle\_area(3, 4, 5) == 0.5 \* 3 \* 4  
  
  
class TestRotationMatrix():  
 def test\_rotation\_matrix1(self):  
 for x, y, z in itertools.product(np.linspace(-1, 1, 5), repeat=3):  
 if x != 0 or y != 0 or z != 0:  
 assert np.all(physics.rotation\_matrix(  
 np.array([x, y, z]), 0) == np.identity(3))  
  
 def test\_rotation\_matrix2(self):  
 for x, y, z in itertools.product(np.linspace(-1, 1, 5), repeat=3):  
 if x != 0 or y != 0 or z != 0:  
 assert np.all(np.round(physics.rotation\_matrix(  
 np.array([x, y, z]), 2 \* np.pi), 10) == np.identity(3))  
  
 # noinspection PyTypeChecker  
 def test\_rotation\_matrix3(self):  
 for angle in np.linspace(-np.pi \* 2, np.pi \* 2, 100):  
 assert np.all(np.round(physics.rotation\_matrix(np.array([0, 0, 1]), angle), 10) == np.round(np.array(  
 [  
 [math.cos(angle), -math.sin(angle), 0],  
 [math.sin(angle), math.cos(angle), 0],  
 [0, 0, 1]  
 ]), 10))  
  
  
table\_side\_1 = table\_sprites.TableSide([[0, 0], [10, 10]])  
table\_side\_2 = table\_sprites.TableSide([[0, 0], [10, 0]])  
ball1 = ball.Ball()  
ball2 = ball.Ball()  
  
ball1.move\_to((0, 0))  
ball1.set\_velocity((1, 1))  
  
fortyfive\_degree\_position = np.array(  
 [np.sin(np.pi / 4), np.cos(np.pi / 4)]) \* ball\_radius \* 2  
  
  
class TestBall():  
 # distance tests will check if the ball distance function is working  
 # properly  
  
 def test\_distance1(self):  
 ball2.move\_to((ball\_radius \* 2, 0))  
 ball2.set\_velocity((0, 0))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_distance2(self):  
 ball2.move\_to((0, ball\_radius \* 2 + 1))  
 ball2.set\_velocity((0, 0))  
 assert not physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_distance3(self):  
 ball2.move\_to((1, ball\_radius \* 2))  
 ball2.set\_velocity((0, 0))  
 assert not physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_distance4(self):  
 ball2.move\_to((ball\_radius \* 2 - 1, 0))  
 ball2.set\_velocity((0, 0))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_distance5(self):  
 ball2.move\_to((1, 0))  
 ball2.set\_velocity((0, 0))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_distance6(self):  
 ball2.move\_to(fortyfive\_degree\_position)  
 ball2.set\_velocity((0, 0))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 # these will check that any balls that are moving away from each other  
 # will not collide  
 def test\_movement1(self):  
 ball1.set\_velocity((1, 1))  
  
 ball2.move\_to((-ball\_radius, 0))  
 ball2.set\_velocity((0, 0))  
 assert not physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement2(self):  
 ball1.set\_velocity((1, 1))  
  
 ball2.move\_to(fortyfive\_degree\_position)  
 ball2.set\_velocity((1, 1))  
 assert not physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement3(self):  
 ball1.set\_velocity((1, 1))  
  
 ball2.move\_to(fortyfive\_degree\_position)  
 ball2.set\_velocity((0.9, 1))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement4(self):  
 ball1.set\_velocity((1, 1))  
  
 ball2.move\_to(-fortyfive\_degree\_position)  
 ball2.set\_velocity((1, 1))  
 assert not physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement5(self):  
 ball1.set\_velocity((1, 1))  
  
 ball2.move\_to(-fortyfive\_degree\_position)  
 ball2.set\_velocity((1.1, 1))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement6(self):  
 ball1.set\_velocity((0, 1))  
  
 ball2.move\_to((ball\_radius \* 2, 0))  
 ball2.set\_velocity((0, 0))  
 assert not physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement7(self):  
 ball1.set\_velocity((0, 1))  
  
 ball2.move\_to((0, ball\_radius \* 2))  
 ball2.set\_velocity((200000000, 0))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement8(self):  
 ball1.set\_velocity((0, 1))  
  
 ball2.move\_to((0, ball\_radius \* 2))  
 ball2.set\_velocity((200000000, 0.9))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement9(self):  
 ball1.set\_velocity((0, 1))  
  
 ball2.move\_to((0, ball\_radius \* 2))  
 ball2.set\_velocity((200000000, 1))  
 assert not physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement10(self):  
 ball1.set\_velocity((0, 1))  
  
 ball2.move\_to((0, ball\_radius \* 2))  
 ball2.set\_velocity((200000000, -200000000000))  
 assert physics.ball\_collision\_check(ball1, ball2)  
  
 def test\_movement11(self):  
 # stationary balls do not collide to conserve unnecessary computations  
 ball1.set\_velocity((0, 0))  
  
 ball2.move\_to((0, ball\_radius \* 2))  
 ball2.set\_velocity((0, 0))  
 assert not physics.ball\_collision\_check(ball1, ball2)  
  
 # reflection line tests  
 # the lines should only reflect balls coming from one direction so the  
 # balls wouldn't get stuck in the reflection lines  
 def test\_line\_ball\_collision\_check1(self):  
 ball1.move\_to(table\_side\_1.middle +  
 np.array([1., -1.]) \* (ball\_radius \*\* 0.5))  
 ball1.set\_velocity([1, -1])  
 assert physics.line\_ball\_collision\_check(table\_side\_1, ball1)  
  
 def test\_line\_ball\_collision\_check2(self):  
 ball1.move\_to(table\_side\_1.middle +  
 np.array([1., -1.]) \* (ball\_radius \*\* 0.5))  
 ball1.set\_velocity([-1, 1])  
 assert not physics.line\_ball\_collision\_check(table\_side\_1, ball1)  
  
 def test\_line\_ball\_collision\_check3(self):  
 ball1.move\_to(table\_side\_1.middle + 1)  
 ball1.set\_velocity([1, -1])  
 assert physics.line\_ball\_collision\_check(table\_side\_1, ball1)  
  
 def test\_line\_ball\_collision\_check4(self):  
 ball1.move\_to(table\_side\_1.middle + ball\_radius)  
 ball1.set\_velocity([-1, 1])  
 assert not physics.line\_ball\_collision\_check(table\_side\_1, ball1)  
  
 def test\_line\_ball\_collision5(self):  
 ball1.move\_to([5, ball\_radius])  
 ball1.set\_velocity([1, -1])  
 assert physics.line\_ball\_collision\_check(table\_side\_2, ball1)  
 physics.collide\_line\_ball(table\_side\_2, ball1)  
 assert np.all(np.around(ball1.velocity, 0) == [1., 1.])  
  
 def test\_line\_ball\_collision6(self):  
 ball1.move\_to(table\_side\_1.middle +  
 np.array([1., -1.]) \* (ball\_radius \*\* 0.5))  
 ball1.set\_velocity([1, -1])  
 assert physics.line\_ball\_collision\_check(table\_side\_1, ball1)  
 physics.collide\_line\_ball(table\_side\_1, ball1)  
 assert np.all(np.around(ball1.velocity, 0) == [-1., 1.])

Here is me running the tests on my machine on the lates pool build.

PYTHONPATH=./pool py.test --cov=.  
========================= test session starts ==========================  
platform linux -- Python 3.6.4, pytest-3.2.3, py-1.5.2, pluggy-0.4.0  
rootdir: /home/max/code/python/pool, inifile:  
plugins: cov-2.5.1, asyncio-0.8.0  
collected 31 items  
  
tests/test\_physics.py ...............................  
  
----------- coverage: platform linux, python 3.6.4-final-0 -----------  
Name Stmts Miss Cover  
-------------------------------------------  
pool/\_\_init\_\_.py 0 0 100%  
pool/ball.py 120 84 30%  
pool/collisions.py 26 18 31%  
pool/config.py 61 5 92%  
pool/cue.py 100 80 20%  
pool/event.py 18 12 33%  
pool/gamestate.py 210 170 19%  
pool/graphics.py 49 38 22%  
pool/main.py 31 31 0%  
pool/physics.py 44 7 84%  
pool/table\_sprites.py 77 56 27%  
-------------------------------------------  
TOTAL 736 501 32%  
  
  
====================== 31 passed in 0.39 seconds =======================

Tests for other builds and instructions on how to run the tests yourself can be found on [the github repository page](https://github.com/max-kov/pool). The records of how the build passes the tests are available on [github](https://github.com/max-kov/pool/commits/master) or on [Travis-ci](https://travis-ci.org/max-kov/pool/builds).

## Other tests

Other features of the game, like the rules of the game, cannot be easily tested with maths, for that purpose, I have created a test table where I actually play out the scenarios described in-game.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test description | Expected output | Test type | Result Reference |  |
| Putting the white ball on top of another ball | Shouldn’t be possible | Erroneous | The game stops moving the ball when it come to contact with another ball |  |
| Pressing ESC in-game | Return to the main menu | Typical | The game returns to the main menu |  |
| Pressing ESC in the main menu | Exit the game | Typical | The game terminates |  |
| Trying to extend the cue past what the game allows | The game should limit the cue extension | Extreme | The game stops the cue extending at a certain point |  |
| Clicking the cue without extending it | The ball shouldn’t move | Extreme | The ball moves towards the cue |  |

# Evaluation

## Objective Analysis

## User Feedback

## Analysis of User Feedback

## Possible Extensions

1. take away the dispalacement vector of the first ball from the second one [↑](#footnote-ref-47)