1.Introduction

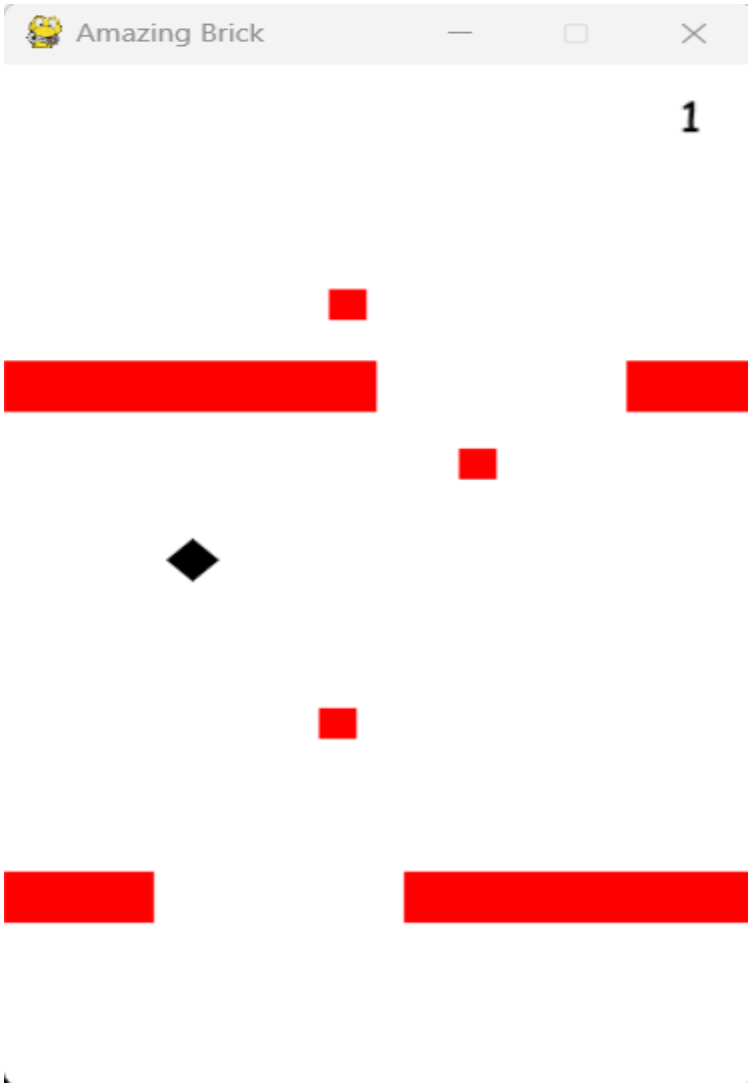
I'm developing the game Amazing brick. I have attempted to reverse engineer a fairly popular game on the mobile platform with the same name using pyGame. The game was originally developed by ketchapp.

Below are the links to the game in android and apple:

apple:
https://apps.apple.com/us/app/amazing-brick/id905455244

Android:
https://play.google.com/store/apps/details?id=com.minimalist.bricks&hl=en_US&gl=US

1.1 Screenshot of the game:



1.2 Requirements:

Python version: Python 3.10.11
Pygame version: 2.1.2

1.3 Controls:

The game can be controlled by moving the left arrow to move the brick to the left and right arrow to move the brick to the right in a projectile motion. Once the game ends and the Game Over message is displayed the player can restart the game by clicking the restart button with a mouse click.

Part 2: Game Design:

The code is developed using Model-View-Controller. The game loop initiates the model and passes it as a parameter to the controller and the view. The game has a brick which is difficult to control, when the brick is moved toward the left or the right by the player there is an upthrust to the brick which will make it move in a projectile motion. The main reason for the brick to move in a projectile motion(parabolic) is due to the downward acceleration. There are 2 types of obstacles in the game; the first consists of 2 rectangles which are separated by a small gap for the brick to pass through. The second type of obstacles are tiny square blocks which are in-between the rectangle obstacles. The player needs to overcome these obstacles one by one. There is no end to this game and the player can compete against himself in order to play better.

2.1 Game Mechanics:

1. **Projectile Motion of the brick:** Any object which has a constant velocity in the X and the Y direction will continue to move in a straight line. However, if there is an acceleration from either the X or the Y direction the motion would not continue to be straight but rather in a parabolic manner. I have set the velocity of 5 units in the x direction and 30 units in the y direction and acceleration of 4 units downwards. These Parameters can be varied to make the game easier/difficult.
2. **Rotation of the brick about its axis:** If the brick collides with the obstacle the brick will fall down rotating about its axis. This is implemented by using the various trigonometric

formulas to compute the position of each vertex of the brick. Using sin to compute the vertical change and cos to compute the horizontal change. The angular velocity of the brick is 5 degrees.

3. **Collision Detection:** This was a fairly tricky part of developing the game as the brick can collide with the multiple obstacles in the game with multiple possibilities. Each vertex of the brick can collide with any of the possible edges of the obstacles.
4. **Brick/Obstacle relative velocity:** One thing that I found perplexing while thinking about reverse engineering this game was how does the brick not move out of the window screen if it is constantly moving upwards with the player's action. This question can be answered by the concept of relative velocity. I have programmed the game such that once the brick reaches to the center of the screen, instead of the brick moving upwards the brick remains stationary and the obstacles in turn move towards the brick at the same velocity giving an illusion that the brick is moving towards the obstacle. And once the brick loses its upward velocity it will fall down and the obstacles become stationary.

There are a couple of games with the same genre developed by ketchapp like **“Stick Hero”** and **“2 Cars”,** Amazing brick is one of them. The thing that fascinates me most about these games is that even though the game looks simple it takes a certain level of practice to get better at the game. These games can get you addicted as well and can be played as a stress buster.

The game uses the usual classes to draw polygons for the brick and rectangles for the obstacles from pygame. It also uses the event listener from pygame to listen to keyboard and mouse clicks. Image class is used to add a restart button image with event listeners to restart the game. Mixer class is used to display the sound in the game. I have used the math library to do some computation for the brick/obstacle relative velocity.

Game Design Changes:

Most of the original design that I had in mind while I was implementing this game is done except for adding a catchy background in the window. I intended to fully reverse engineer the game which is done at a satisfactory level.

Adding a background for this game would make the game visually more appealing but would confuse the player thus making the game less enjoyable.

Game Development and Documentation

The game is developed using Model-View-Controller architecture:

1. Model: The model consists of all the business logic of the game. The initial positions of the brick and the obstacles. The brick is a diamond shaped figure which is referenced by its center coordinates. In each iteration of the game loop the position of the brick is updated in the model based on the input received from the user on the controller.

```
def __init__(self, window_length, window_width):
    self.window_length = window_length
    self.window_width = window_width
    self.brickPos =  [self.window_length//2, self.window_width//2]
    self.brickVelocity = [5, 30]
    self.acceleration = 4
    self.fps = pygame.time.Clock()
    self.brickColor = (0,0,0)
    self.keyPressed = None
    self.rectTop1 = -25
    self.rectTop2 = -275
    self.event = None
    pygame.init()
    pygame.mixer.init()
    self.gap1, self.gap2 = self.getRandomGap()
    self.color = self.getRandomColor()
    self.obstacleTop1 = self.rectTop1-random.randint(30,90)
    self.obstacleLeft1 = self.gap1+random.randint(0,100)
    self.obstacleTop2 = self.rectTop2+random.randint(30,90)
    self.obstacleLeft2 = self.gap2+random.randint(0,100)
    self.obstacleTop3 = 500
    self.obstacleLeft3 = self.gap1+random.randint(0,100)
    self.obstacleTop4 = self.rectTop2-random.randint(30,90)
    self.obstacleLeft4 = self.gap2-random.randint(0,100)
    self.current = "top1"
    self.score = 0
    self.gameRect1 = pygame.Rect(0, self.rectTop1, self.gap1, 25)
    self.gameRect2 = pygame.Rect(self.gap1+100, self.rectTop1, 300-(self.gap1+100), 25)
```

Figure 1: The constructor of the gameState class (model)

The top and the left part of each rectangle and square obstacles are kept in track in the model. The positions and velocity are also recorded. Initial position of the brick is at the center of the window with only the first obstacle visible to the user. The first rectangular obstacle top is at -25 and the second rectangle obstacle top is at -275. As the brick moves closer to the obstacles they are uncovered from the top of the screen.

```
def updatePositionAndScore(self):
    self.setUpObstacles()
    self.moveBrickOrObstacles()
    self.regenerateObstacle()
    self.detectCollision()
    self.updateScore()
```

Figure 2: updatePositionAndScore method in the model

This method is called in each iteration of the game loop, the setUpObstacles method will set up the updated position of the obstacles. The method **moveBrickOrObstacles()** will make a decision based on the brick position to either move brick towards the obstacle or the obstacle towards the brick based on the position of the brick.

```
def moveBrickOrObstacles(self):
    self.brickVelocity[1] = self.brickVelocity[1]- self.acceleration
    if self.keyPressed == "left":
        if(self.brickPos[0]-10 >= 0):
            self.brickPos[0] = self.brickPos[0]-self.brickVelocity[0]
        else:
            self.brickVelocity[1] = -10
        if self.brickPos[1] > 250:
            self.brickPos[1] = self.brickPos[1]-self.brickVelocity[1]
        else:
            self.moveObstacles()

    if self.keyPressed == "right":
        if(self.brickPos[0]+10 <= 300):
            self.brickPos[0] = self.brickPos[0]+self.brickVelocity[0]
        else:
            self.brickVelocity[1] = -10
        if self.brickPos[1] > 250:
            self.brickPos[1] = self.brickPos[1]-self.brickVelocity[1]
        else:
            self.moveObstacles()
```

Figure 3: moveBrickOrObstacles()

The method regenerateObstacles() will check if the obstacles have moved out of the screen, if so it will change the obstacle position to the top of the screen so that it becomes the next obstacle the player has to surpass.

```
def regenerateObstacle(self):
    if self.rectTop1 >= 500:
        self.rectTop1 = self.rectTop2 - 275
        self.gap1 = random.randint(40, 180)
        self.obstacleTop1 = self.rectTop1-random.randint(30,90)
        self.obstacleLeft1 = self.gap1+random.randint(0,100)
        self.obstacleTop3 = self.rectTop1+random.randint(30,90)
        self.obstacleLeft3 = self.gap1+random.randint(0,100)

    if self.rectTop2 >= 500:
        self.rectTop2 = self.rectTop1 - 275
        self.gap2 = random.randint(40, 180)
        self.obstacleTop2 = self.rectTop2+random.randint(30,90)
        self.obstacleLeft2 = self.gap2+random.randint(0,100)
        self.obstacleTop4 = self.rectTop2-random.randint(30,90)
        self.obstacleLeft4 = self.gap2+random.randint(0,100)
```

Figure 4: regenerateObstacles

```
def detectCollision(self):
    self.detectSquareCollision(self.gameRect5)
    self.detectSquareCollision(self.gameRect6)
    self.detectSquareCollision(self.gameRect7)
    self.detectSquareCollision(self.gameRect8)
    self.detectLeftRectangleCollision(self.gameRect1)
    self.detectLeftRectangleCollision(self.gameRect3)
    self.detectRightRectangleCollision(self.gameRect2)
    self.detectRightRectangleCollision(self.gameRect4)
```

Figure 5: detectCollision

The collision detection logic is different for the left rectangles, the right rectangles and the square obstacles. So based on the obstacle that we are checking I have written separate methods to check for the collision. Once collision is detected it will set the gameOver variable to True which is then updated in the view. This is one of the most complex pieces of logic in the game.

```
def updateScore(self):
    if self.brickPos[1] <= self.rectTop1 and self.current == "top1":
        self.score += 1
        self.current = "top2"
    if self.brickPos[1] <= self.rectTop2 and self.current == "top2":
        self.score += 1
        self.current = "top1"
```

Figure 6:updateScore

This method checks if the brick has moved passed the rectangle obstacles if so it will increment the score by 1.

2.Controller:

```
class eventHandler:
    def __init__(self,model):
        self.model = model

    def handleEvent(self, event):
        keys = pygame.key.get_pressed()
        self.model.event = event
        if keys[pygame.K_LEFT]:
            self.model.brickVelocity[1] =  30
            self.model.keyPressed = "left"
        elif keys[pygame.K_RIGHT]:
            self.model.brickVelocity[1] =  30
            self.model.keyPressed = "right"
        elif event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

Figure 7: eventHandler function

The controller gets the left or the right keys pressed by the player and passes it on to the model to move the brick accordingly.

3.View:

```
def updateView(self):
    if self.model.gameOver:
        theta = 0
        if self.tempY+10 <= 500:
            self.tempX = self.model.brickPos[0]
            self.tempY = self.model.brickPos[1]
        while self.tempY+10 <= 500:
            theta += 5
            self.surface.fill((255,255,255))
            xCh = 10-10*math.cos(theta)
            yCh = 10*math.sin(theta)
            pygame.draw.polygon(self.surface, self.model.brickColor, ((self.tempX-10+xCh, self.tempY-yCh), (self.tempX+yCh, self.tempY-10+xCh), (s
            self.drawRectangles()
            self.tempY = self.tempY + 5
            pygame.display.update()
            self.fps.tick(50)
        self.displayGameOverMessage()


    else:
        self.surface.fill((255,255,255))
        pygame.draw.polygon(self.surface, self.model.brickColor, ((self.model.brickPos[0]-10, self.model.brickPos[1]), (self.model.brickPos[0], se
        self.drawRectangles()
        self.updateScore()
```

Figure 8: updateView method

This method will update the window screen to draw the rectangle and obstacles during each iteration. It will check the gameOver variable from the model. If it is set to true it will display the“Game Over” message along with the final score and a restart button so that the user can

play again. The brick rotation logic is written in the view as it was difficult to be implemented in the model.

4.Game Loop:

```
class amazingBrick:
    def __init__(self):
        self.model = model.gameState(300, 500)
        self.controller = controller.eventHandler(self.model)
        self.view = view.gameView(self.model)
        self.fps = pygame.time.Clock()

    def startGameLoop(self):
        while True:
            for event in pygame.event.get():
                self.controller.handleEvent(event)
            self.model.updatePositionAndScore()
            self.view.updateView()
            self.fps.tick(12)
```

Figure 9: run the game loop

The game loop initiates the model,view and the controller.

Bugs:

1. Sometimes the brick passes through the square obstacles where collision is not detected.
2. The final score after the game is over sometimes is increased by 1.

Group Member Roles, Tasks, and Performance:

Work Load:

Amith:

1. Implemented the model,view and the controller and provided a connection between them.
2. Implemented the logic for collision detection and Game Over message.

Timeline:

MileStone 1:

Basic outline of the model view and controller was completed.

MileStone 2:

The game Logic is completely implemented.

Final Submission:

Implementation of the Game Over and the restart logic was implemented.

Source Code:

https://github.com/amithbv98/BrickGame

Demo Video:

https://drive.google.com/file/d/1EiEz4019h48CtMWZ7W5L5SxrHZlwe0AE/view?usp=sharing