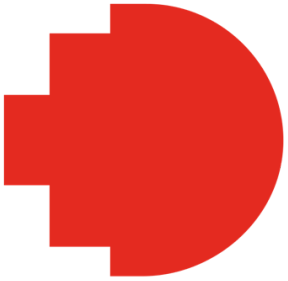


RMIT UNIVERSITY

COSC2659 - iOS Development
Assignment 2

By:
Nguyen Phuc Cuong – s3881006

Introduction **3**

Functions **3**

How to play 3

Main functions 4

Main menu 4

Option 4

Tutorial view 6

Gameover view 7

Leaderboard view 7

Game view 8

DragView 9

PhysicsHandler 13

Initializers: 13

Trajectory splitter: 13

Trajectory calculator: 14

Extra features: 15

Saving user’s data 15

Difficulty setting 15

Video demonstration link: **15**

Introduction

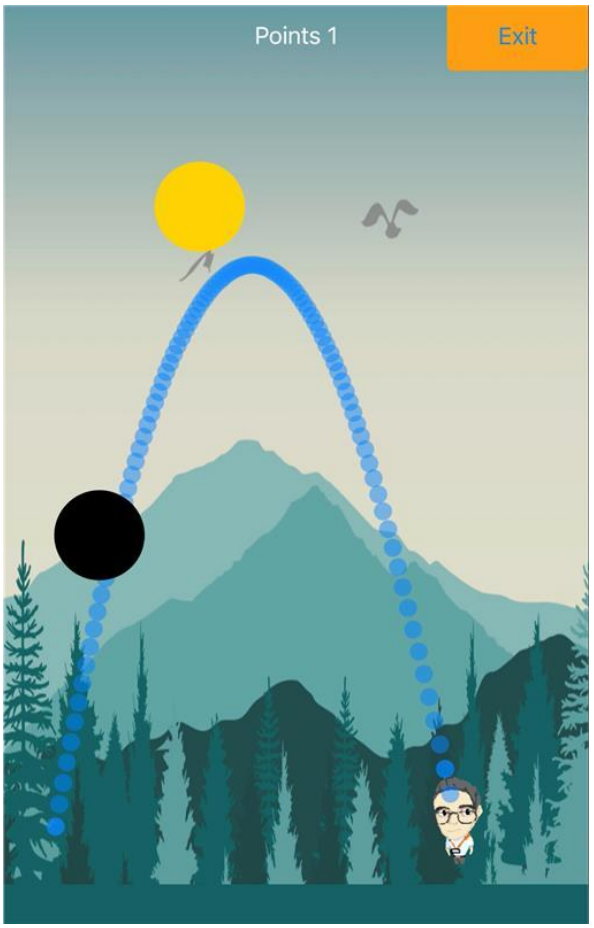
The game I made is called Jump Master, which is a jumping game that utilises the touch detection feature of smartphones that help launch a character into a direction to collect points and avoid obstacles.

The game is heavily inspired by Angry Bird in terms of the physics and launching characters aspect.

The main simulator that is used for testing and running the simulated version of the game is Iphone 11 Pro

Functions

How to play



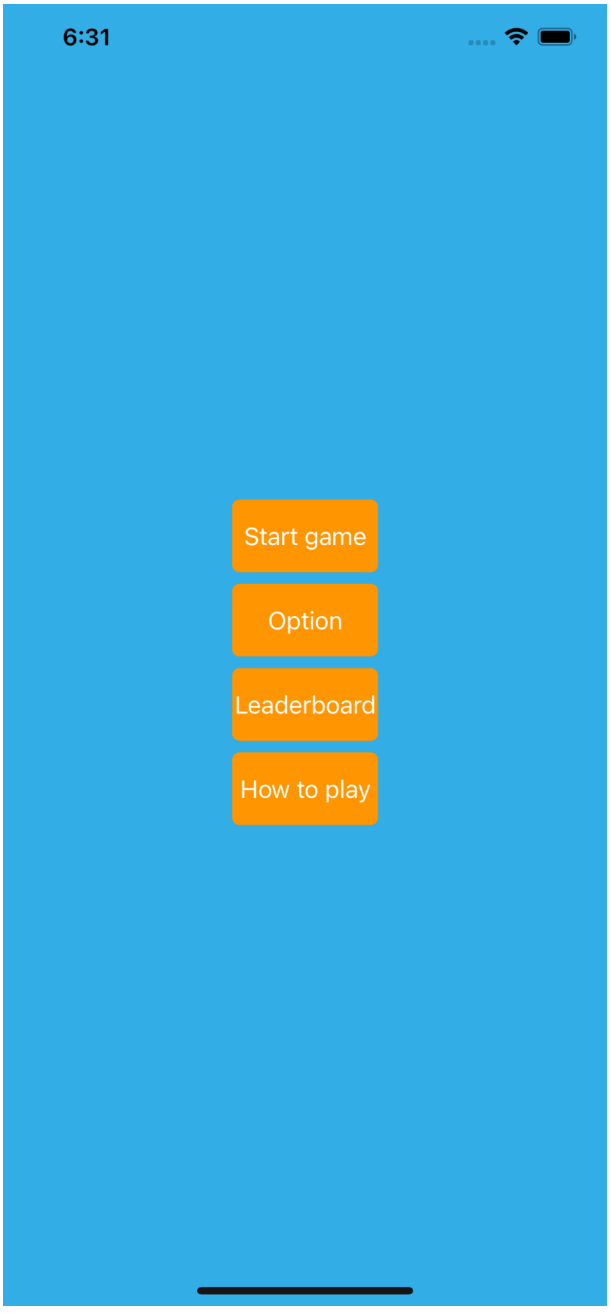
The game is played by dragging a character back from the launching direction, similar to that of a slingshot. Once the character is released, the player is launched in the opposite direction.

The blue line indicates the trajectory of the character's flight and is calculated and displayed in real time. This feature can be disabled by turning off easy mode.

Main functions

Main menu

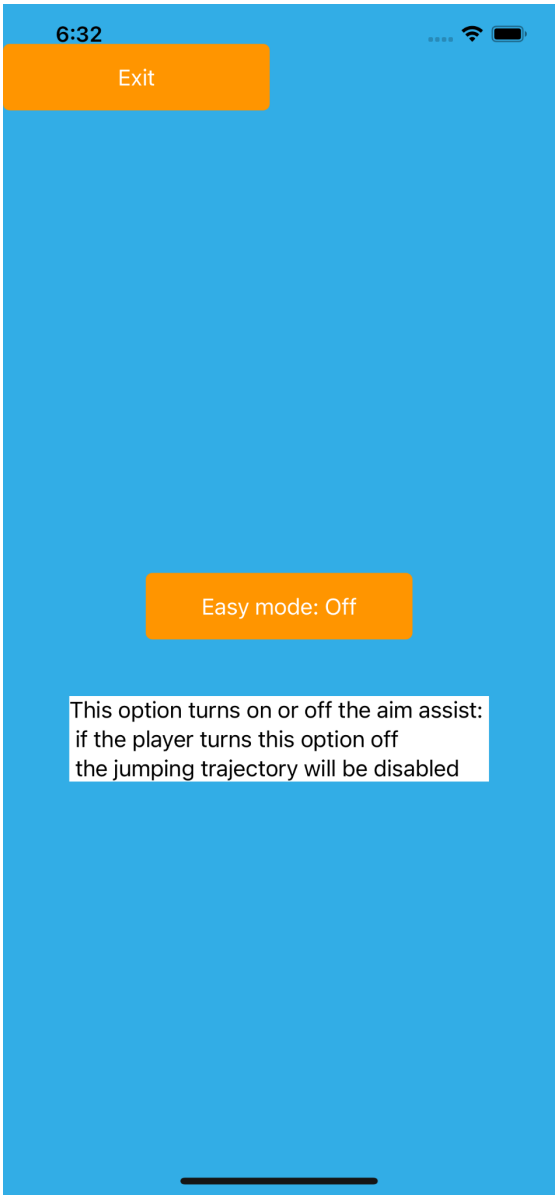
The main menu display the navigation for the other View that are Start Game, Option, Leaderboard and How to play



Option

The option menu allows the user to choose whether to play with or without the trajectory guide that assists the player in aiming.

The user can exit this view by pressing the exit button at the top left corner of the screen



The toggling easy mode is done by using @Binding for a boolean which will be used in the main game view.

```
        .foregroundColor(.orange)
Button("Easy mode: \((easyMode) ? "On" : "Off")"){
        easyMode.toggle()
}
```

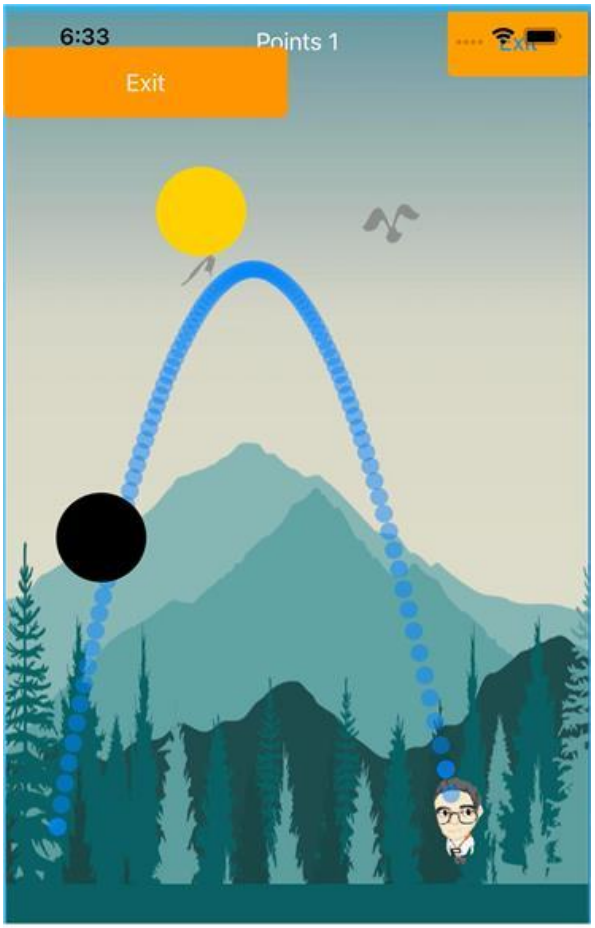
In the game view, the objects that are used for displaying the trajectory will check whether easy mode is true or false.

```
ForEach(physics.finalMovement){position in
    Circle()
        .fill(.blue)
        .frame(width: 64, height: 64)
        .scaleEffect(0.2)
        .offset(position.end)
        .opacity((easyMode) ? (isDragging ? 0.5 : 0.0) : 0.0)
}
```

If true then the objects will have the opacity of 0.5 and if false, the objects will have the opacity of 0.0.

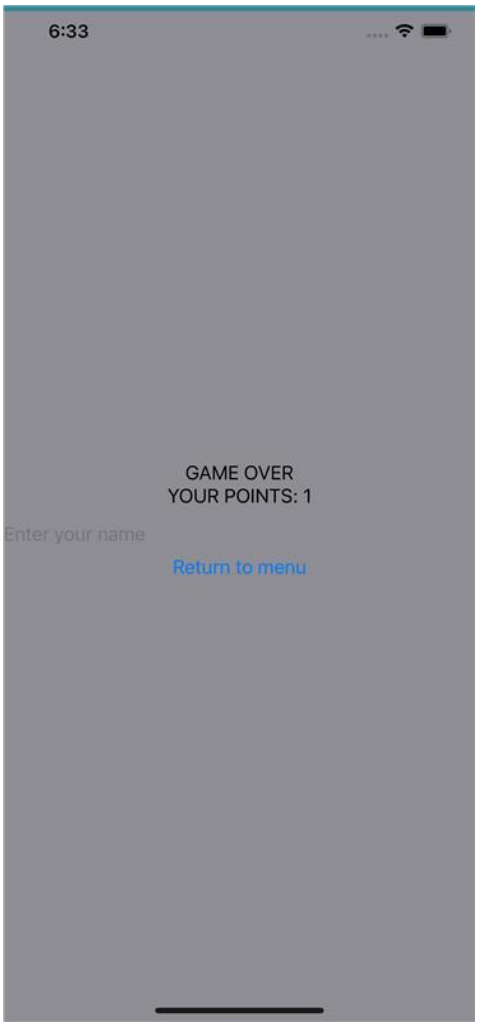
Tutorial view

The tutorial view (HowToPlayView) shows the player how the game is played



Tap on the character and hold
the blue path is the direction which
they're launched. Release to launch
collect the yellow balls to get points
careful not to touch the black ones

Gameover view



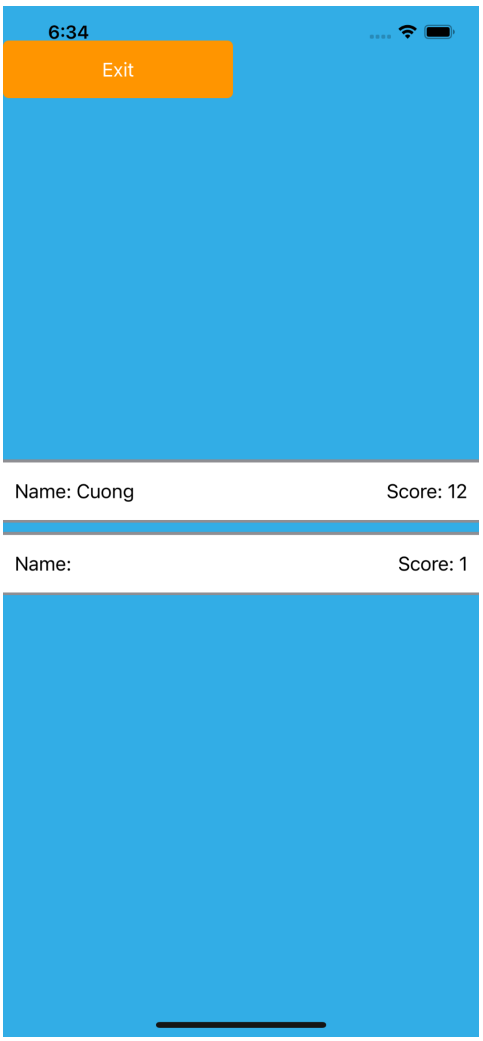
This view display the player's final points after hitting an obstacle that is the black ball

Once the player type in their name and press the "Return to menu" button, their scores will be saved in the leaderboard as well as in UserDefaults and will be displayed in the leaderboard view

```
Button("Return to menu"){
    gameOver = false
    gameView = false
    highscores[name] = points
    highscores[name] = points
    UserDefaults.standard.set(highscores,forKey: "highscores")
    UserDefaults.standard.set(0, forKey: "scores")
    UserDefaults.standard.set(0, forKey: "currentPositionWidth")
    UserDefaults.standard.set(-(UIScreen.main.bounds.height/2 - 250), forKey: "currentPositionHeight")
    points = 0
}
```

Leaderboard view

The leaderboard view display the points in the previous attempt at playing the game



The leaderboard uses @Binding to display all the scores after the player loses and use ForEach to display all the previous attempts

```
VStack{
    ForEach(highscore.sorted(by: >), id: \.key) { key, value in
        ZStack{
            Rectangle()
                .frame(width: UIScreen.main.bounds.width, height: 55)
                .foregroundColor(.gray)
            Rectangle()
                .frame(width: UIScreen.main.bounds.width, height: 50)
                .foregroundColor(.white)
            HStack{
                Spacer()
                    .frame(width: 10)
                Text("Name: \(key)")
                Spacer()
                Text("Score: \(value)")
                Spacer()
                    .frame(width: 10)
            }
        }
    }

}
```

Game view

The game view utilises a view that groups all the views that make up the game, named EnvironmentManager. The EnvironmentManager handles displaying the playing

environment including the point balls, the obstacle balls, the ground, the background, the points and the DragHandler.

To exit this view, the user can press the exit button located in the top right corner of the screen

DragView

This view mainly handles the player moving animation, the launching trajectory displaying, the points handling, the losing handling and is used to call the main physics handler of the player as well as calculating the launching force.

The following image shows the codes for handling the user's dragging gesture for launching the player character

```
let dragGesture = DragGesture()
    .onChanged { value in
        if (!onClick){
            prevMouse = CGSize(width: value.translation.width + accumulated.width, height: value.translation.height + accumulated.height)
            onClick = true
        }
        opacityHandler = 100

        currentMouse = CGSize(width: value.translation.width + accumulated.width - prevMouse.width
                              , height: value.translation.height + accumulated.height - prevMouse.height)
        offset = CGSize(width: accumulated.width + currentMouse.width, height: accumulated.height + currentMouse.height)
        if (!physics.finalMovement.isEmpty){
            physics.clearMovement()
        }
        physics.launch(endPosition: CGSize(width:prevPos.width - (value.translation.width*1.5 + accumulated.width - prevMouse.width), height: prevPos.height - (value.translation.height*1.5 + accumulated.height -
            prevMouse.height)))
        isDragging = true

    }
    .onEnded { value in
```

This block of code calculates the distance between the current mouse and the mouse when clicking the player character. A vector from the current mouse position to the previous mouse position will be drawn to calculates the magnitude of the launching force

This is implemented to assist the player's aiming activity using visual queues.

```
.onEnded { value in


    isDragging = false

    onClick = false
    opacityHandler = 0.0
    var jump = 0
    var motionCount = 0
    var death = false
    for motion in physics.finalMovement{
        motionCount += 1
        withAnimation(.linear(duration: durationTimer).delay(delayTimer)){
            if (jump < 3){
                standing = 0.0
                startJumping = 1.0
            }
            else if (jump >= 3 && jump < physics.finalMovement.endIndex-1){
                startJumping = 0.0
                jumping = 1.0
            }
            else {
                jumping = 0.0
                standing = 1.0
            }

            for balls in pointBalls.positions{
                if (MovementHandler.getDistant(vector: MovementHandler.getVector(current: motion.end, end: balls.position)) <= 97.0 && !balls.added){
                    hitCount += 1
                    balls.added = true
                    points += 1;
                    if let killBallsData = try? encoder.encode(killBalls){
                        UserDefaults.standard.set(killBallsData, forKey:"killBalls")
                    }
                    if let pointBallsData = try? encoder.encode(pointBalls){
                        UserDefaults.standard.set(pointBallsData, forKey:"pointBalls")
                    }
                    withAnimation(.linear.delay(delayTimer)){
                        playSound(sound: "points", type: "mp3")
                    }

                }


            }
            for kills in killBalls.positions{
                if (MovementHandler.getDistant(vector: MovementHandler.getVector(current: motion.end, end: kills.position)) <= 35.0 && !kills.added){
                    death = true
                    playSound(sound: "oof", type: "mp3")
                    break;
                }
            }


            offset = motion.end
        }
```

This block of codes is used to calculate and assign the player's position accordingly to the calculated physics

For clarity purpose, this code will be split into small section for further explanation

```
for motion in physics.finalMovement{
    motionCount += 1
    withAnimation(.linear(duration: durationTimer).delay(delayTimer)){
        if (jump < 3){
            standing = 0.0
            startJumping = 1.0
        }
        else if (jump >= 3 && jump < physics.finalMovement.endIndex-1){
            startJumping = 0.0
            jumping = 1.0
        }
        else {
            jumping = 0.0
            standing = 1.0
        }

        for balls in pointBalls.positions{
            if (MovementHandler.getDistant(vector: MovementHandler.getVector(current: motion.end, end: balls.position)) <= 97.0 &&
                !balls.added){
                hitCount += 1
                balls.added = true
                points += 1;
                if let killBallsData = try? encoder.encode(killBalls){
                    UserDefaults.standard.set(killBallsData, forKey:"killBalls")
                }
                if let pointBallsData = try? encoder.encode(pointBalls){
                    UserDefaults.standard.set(pointBallsData, forKey:"pointBalls")
                }
                withAnimation(.linear.delay(delayTimer)){
                    playSound(sound: "points", type: "mp3")
                }

            }


        }
        for kills in killBalls.positions{
            if (MovementHandler.getDistant(vector: MovementHandler.getVector(current: motion.end, end: kills.position)) <= 35.0 &&
                !kills.added){
                death = true
                playSound(sound: "oof", type: "mp3")
                break;
            }
        }


        offset = motion.end
```

The physics in this project is composed entirely of vectors. Gravity, bounce, the curve in trajectory is done by splitting one big vector into smaller vectors which physics can then be applied on and create a curve in the moving trajectory. The physics calculation will be done in the PhysicsHandler and MovementHandler section of the report

After that, the trajectory will be checked whether they hit a point ball or an obstacle ball that will determine whether the player will receive another point or will be considered as failing.

Once all the point ball is collected, all the balls will be reset and their positions will be randomised with the following code:

```
if (hitCount >= pointBalls.positions.endIndex && motionCount >= physics.finalMovement.endIndex && !gameOver){
    killBalls.restart()
    killBalls.checkFair(player: physics)
    pointBalls.restart()
    killBalls.checkFair(score: points)
    pointBalls.checkFair(score: points)
    hitCount = 0
    if let killBallsData = try? encoder.encode(killBalls){
        UserDefaults.standard.set(killBallsData, forKey:"killBalls")
    }
    if let pointBallsData = try? encoder.encode(pointBalls){
        UserDefaults.standard.set(pointBallsData, forKey:"pointBalls")
    }
}
```

```
                Image("Sprite-2")
                    .resizable()
                    .frame(width: 64, height: 64)
                    .scaleEffect(1)
                    .offset((isDragging) ? prevPos : offset)
                    .opacity(standing)
                Image("Sprite-3")
                    .resizable()
                    .frame(width: 64, height: 64)
                    .scaleEffect(1)
                    .offset((isDragging) ? prevPos : offset)
                    .opacity(startJumping)
                Image("Sprite-4")
                    .resizable()
                    .frame(width: 64, height: 64)
                    .scaleEffect(1)
                    .offset((isDragging) ? prevPos : offset)
                    .opacity(jumping)

                if (!physics.finalMovement.isEmpty){
                    ForEach(physics.finalMovement){position in
                        Circle()
                            .fill(.blue)
                            .frame(width: 64, height: 64)
                            .scaleEffect(0.2)
                            .offset(position.end)
                            .opacity((easyMode) ? (isDragging ? 0.5 : 0.0) : 0.0)
                    }
                }

                Circle()
                    .fill(.red)
                    .frame(width: 64, height: 64)
                    .scaleEffect(isDragging ? 1 : 0)
                    .offset(offset)
                    .opacity(0.5)
                    .gesture(dragGesture)
```

This code block displays the trajectory balls and the player position as well as the jumping animation by switching each image opacity when it's appropriate.

PhysicsHandler

This class is the main physics handler that calculates all the physics needed for the animation.

This section will be split into 3 main parts, the initializers, the trajectory splitter, the trajectory calculators

Initializers:

```
init(position:CGSize) {
    self.movement = MovementHandler.init(current: position, end: position, id: 0)
    self.otherFactor = []
    self.movementDivided = []
    self.finalMovement = []
    self.obstacles = []
}
```

```
func addFactor(factor: MovementHandler){
    self.otherFactor.append(factor)
}
```

The 2 functions above is used for initialising the physics handling object

Trajectory splitter:

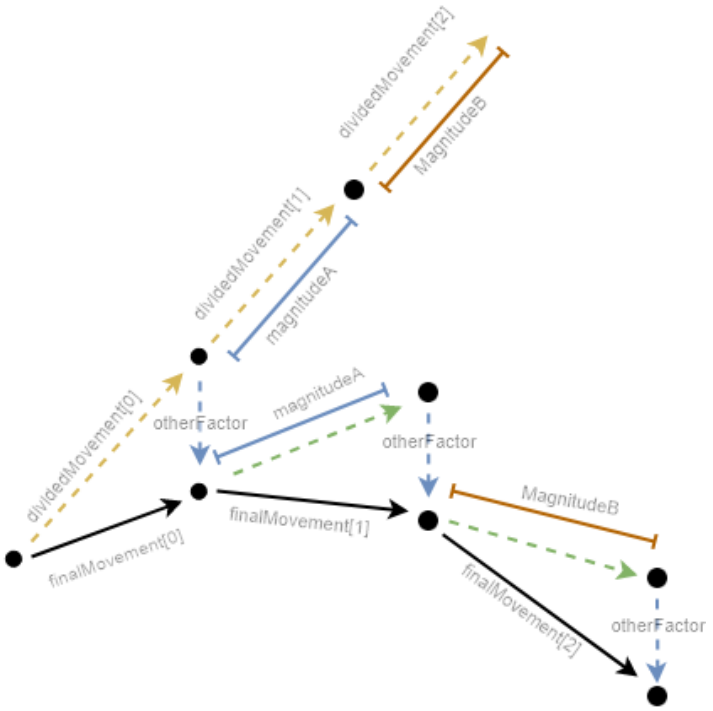
```
func getMovement(count: Int){
    if (count < 20){
        //Add movement divider
        let newMovement = MovementHandler.init(
            current:
                (self.movementDivided.endIndex == 0) ? self.movement.current : self.movementDivided[self.movementDivided.endIndex-1].end,
            end: MovementHandler.addVector(first:
                                           MovementHandler.divideVector(vector:
                                                                        MovementHandler.getVector(current:
                                                                                                  self.movement.current,
                                                                                                  end:
                                                                                                  self.movement.end),
                                                                        divideBy: 20),
                                           second:
                                            (self.movementDivided.endIndex == 0) ? self.movement.current : self.movementDivided[self.movementDivided.endIndex-1].end
                                           ),
            id: self.movementDivided.endIndex
        )
        movementDivided.append(newMovement)
        //End
        let _ = print(self.movementDivided.endIndex-1)
        self.getMovement(count:count+1)

    }
}
```

The getMovement function splits the initial movement vector into 10 smaller movement vectors. The function uses recursion and once all the vectors has been made, it will exit the function

Trajectory calculator:

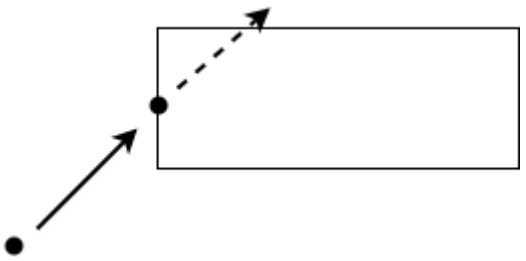
```
func applyFactor(){
    var touchStop = false
    var wallBounce = false
    var dividedMotion: [MovementHandler] = []
    for dividedMovement in self.movementDivided{

        let motion = MovementHandler.init(current:
                                            (dividedMovement.id == 0) ?
                                            (dividedMovement.current):
                                            (self.finalMovement[self.finalMovement.endIndex - 1].end),
                                          end:
                                            (dividedMovement.id == 0) ?
                                            (dividedMovement.end)
                                            :
                                            (dividedMovement.changeVectDirectionWithReturn(reference: self.finalMovement[self.finalMovement.endIndex-1]).end),
                                          id: finalMovement.endIndex)
        motion.duration = 0.002
```

The applyFactor function applies all other factors that could affect the movement trajectory that include gravity and wall bounce. The following graph will help visualise the calculation process

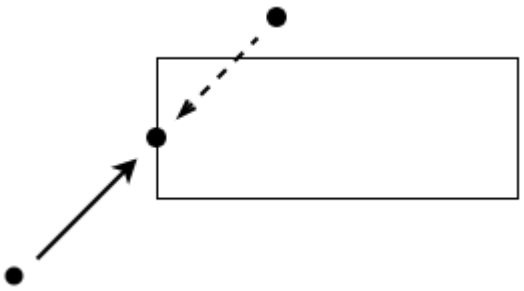


A small vector that's parallel to the previous vector will be drawn, the magnitude of the initial movement vector will be applied to that vector to change the moving trajectory

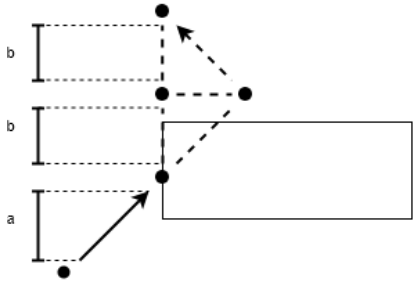
The bouncing physics will be demonstrated in the following graphs:

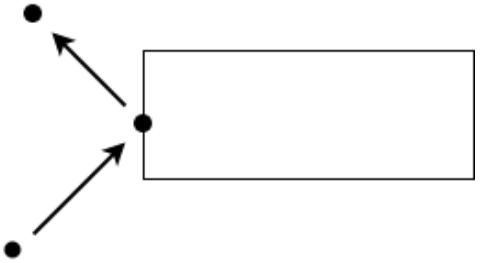


When the vector hits a wall, a smaller parallel vector will be drawn until it hits said obstacle.



A vector will then be drawn moving back towards the point of collision to get the bouncing off direction.





The direction which is not directly against the obstacle will continue its path with direction and angle taken into consideration. After that, that vector will be set back to where the player would bounce off

Extra features:

Saving user's data

User's data are saved into UserDefaults

Difficulty setting

The user can turn on or turn off easy mode. Easy mode will display the trajectory of the launch and if it's disabled, the user will have to guess where they lands

Video demonstration link:

https://rmiteduau-my.sharepoint.com/:v:/g/personal/s3881006_rmit_edu_vn/EXGuuZyv6FhFme8RVztzhRgBeq5LmrQ54_NpnCNQALY9mg?e=GKS7aj