Sumanth Patil, Fahim Sajjad, Qiwei Wang
Dr. Peter Keleher
Final Project - “Lavýrinthos”
December 1st, 2021

Lavýrinthos: Development and Implementation

Overview and Idea

After the final project was introduced to us, all the team members were all interested and preferred to make a video game. Since there were so many asteroids games created by other students, our team decided to make something innovative and new.

Sumanth first came up with the idea of a maze game, he talked about how the game could be having the player control the character to avoid or kill monsters and collect all the gold coins, the goal of the game is to collect as much gold as possible and to get to the exit (the door) of the maze. Fahim and Qiwei were both intrigued by Sumanth’s idea and then started to pitch in ideas. Firstly, we had to come up with a name for our game. After doing research on different mazes and the history of them, we found out that the maze originated in Greek mythology, where the maze is designed by the architect Daedelus to hold a monster called the “Minotaur”. In the story, humans are sent to the maze and eventually will be killed by the monster, but the bravest heroes can kill the monster and escape the maze. Inspired greatly by the story, we decided to name our game as “the Maze” in Greek, which is “Lavýrinthos”.

With a name of the game on the table, we then moved to discussing what our game would be like. As our minimal project goals, we believe that we need to have a character to control with buttons (left, right, up and down). Then we would have different levels of mazes, using a randomization algorithm. We would also have monsters. The monsters are also an inspiration from the mythology, the “Minotaur”. At last, we will have gold coins as treasures for players to collect and the door for players in order to progress to the next level of the game. All “treasures”,

coins, must be collected in order for the player to enter the door to the next level. After having a basic idea of what the game would look like with minimal goals. We took another step up and had a few stretch goals. When we designed the game, we would really put ourselves in the Greek mythology as we were the humans trapped in the maze. Therefore, we thought for the stretch goals we can reduce the visibility of the player. For instance, we can limit the player's visibility by only showing a little piece of the maze at a time, so that the player can only see limited areas around the character that he/she controls. In addition, to make the game more controllable, we decided that if we had time, we could improve the control system from only buttons, to joysticks as another movement option. The player can have whichever movement control they want at the game settings. Finally, there's a "slay the monster" scene in the mythology, and we wished that the hardest goal would be enabling our characters to have weapons and kill the monsters they encounter.

Choosing a specific library to use for the game was fairly easy. Choosing which parts of the project to use SpriteKit on was a bit more difficult during development, but this will be expanded upon later. With the project parameters set, and the external library decided, we started to work.

Early Structure and Maze Engine

We realized quickly that coding without a plan was a plan for failure. We wanted to have the game mechanism perfected on paper before we moved forward with programming. This stage had MANY changes. The first topic of discussion was the maze engine. There were a couple thoughts on this: 1) randomization algorithms for mazes are exceedingly difficult to implement 2) post randomization we'd need to write logic to find an optimal level finish position, coordinates for treasures and monsters, coordinates for monster movement, etc.

Fulfilling all these tasks without a set maze for each level would require much more effort, a bad return on investment, than creating our own set levels. **So this became our first change in project scope: randomized levels will be replaced by 3 preset levels.** We decided on 3 for the time being because we thought we would have to manually set the levels using a 2D array similar to how 2048 was implemented in projects 1-3.

As aforementioned, we were unsure where exactly we would need to use SpriteKit. At first, we believed we could set the mazes up manually using SwiftUI shapes, and just use SpriteKit for the monster and player character nodes. We first attempted to create maze objects using the Rectangle SwiftUI objects to form boxes without a fill. This worked, and we were able to create a maze by hiding certain edges. The problem however arises when we want the player character to bounce off or be blocked by walls. We realized, even before implementing a character, that this is not possible using SwiftUI only, and so the walls/maze blocks would also need to be constructed using SwiftUI. In other words, in addition to the nodes which would comprise the door, monsters, and player, we would need to develop the maze itself as a large series of nodes. This is exactly what we did. We create a sprite node for each block in a 16 x 20 grid and upload these sprites accordingly to a 2D array which is displayed on the screen with coordinates passed in as parameters to the MazeBlock Struct as shown to the right. These xcoord and ycoord parameters would then be used to add this struct block as a node to the scene at a specific point. This way we could manually create a maze.

```
struct MazeBlock {
    let xcoord: CGFloat?
    let ycoord: CGFloat?

    init(x: CGFloat, y: CGFloat) {
        xcoord = x
        ycoord = y
    }
    let block = SKSpriteNode(imageNamed: "wall")
}
```

We realized fairly quickly that developing 3 levels or more like this would

result in several thousands of lines of code. Time and effort which was not as rewarding as it may have seemed earlier. We needed a better method of creating and implementing our mazes. The solution we found was to take a file and use a bufferreader to go character by character and determine what belongs in each coordinate of the grid. Similar to a battleship game setup logic. The same concept could be used in Swift to develop nodes automatically. The only pre-build work we would have to do as a team is to write the logic for which character corresponds to what type of node. The position could be set for each node and the node could be added to the game scene.

```
func setMaze() {
    if let levPath = Bundle.main.path(forResource: "level\(levelNum)", ofType: "txt"){
        if let levS = try? NSString(contentsOfFile: levPath, encoding: String.Encoding.ascii.rawValue) {
            let lines = levS.components(separatedBy: "\n") as [String]
            for (row, line) in (lines.reversed().enumerated()) {
                for (col, tilecode) in line.enumerated() {
                    var pos = CGPoint(x: (40 * col) - 300, y: (40 * row) - 350)          ⚠ Variable 'pos' was never mutated; consider changing to 'let' constant

                    if (tilecode == "w") {
                        let node = SKSpriteNode(imageNamed: "wall")
                        node.position = pos
                        node.name = "wall"
                        node.physicsBody = SKPhysicsBody(rectangleOf: node.size)
                        node.physicsBody?.categoryBitMask = CollisionType.wall.rawValue
                        node.physicsBody?.isDynamic = false

                        addChild(node)
                    }
                    else if (tilecode == " ") {
                        let node = SKSpriteNode(imageNamed: "space")
                        node.position = pos
                    }
                    else if (tilecode == "c"){
                        var coin = SKSpriteNode(imageNamed: coinTextureAtlas.textureNames[0] as! String)  2 ⚠ Forced cast of 'String' to same type has no effect
                        coin.name = "coin"
```

By treating the entire file as a string and separating lines by the “\n” character, we could get a line of characters to read and develop nodes from. So depending on what character was read, we place a coin, space, wall, or door sprite in that position. All we had to do after this logic was developed 16 x 20 grid files.

Once we were able to create the maze, adding monsters and the player character was simple. We just created them as nodes for each level and added them in different positions to fit with each level.

Player and Monster Movement

This part of development started with a scope change: discard the attack and health features of the game. It became too difficult to implement a weapon and health as they would also need to be represented as sprites, and animated separately. Instead, in order to make the game more akin to a retro game like Mario, we decided if the player character runs into or touches a monster, they will automatically die and reset the level. This was also closer to the Greek story the game draws inspiration from.

One of the most difficult parts of the project was implementing player movement. Unlike monster movement which was built into the game using a sequence of node animations (coordinate movements), player movement is obviously controlled by the user. This means the player must move on indication or user action. This would require an action event, a listening event, and then a resulting game scene event.

We used the "touch.location(in:self)" feature of SpriteKit to locate where the user was touching. If the area of a button contains the coordinates of the touch then we can set the selectedButton to that respective button, and use the override touchesBegan and touchesEnded methods to successfully call a GameScene action.

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let touch = touches.first  { ...         }
}

override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let touch = touches.first { ...         }
    selectedButton = nil
}
func moveRight(){
    character.run(SKAction.moveBy(x: 40, y: 0, duration: 0.5))
}

func moveLeft(){
    character.run(SKAction.moveBy(x: -40, y: 0, duration: 0.5))
}

func moveUp(){
    character.run(SKAction.moveBy(x: 0, y: 40, duration: 0.5))
}

func moveDown(){
    character.run(SKAction.moveBy(x: 0, y: -40, duration: 0.5))
}
```

One obstacle we ran into after implementing the player movement was the player falling through the rest of the nodes. Basically, since we hadn't implemented collision interaction yet, the player was not stopping when it hit a wall. This was expected. What was unexpected, however, is the player moving downward without touching the down button at all. After a bit of research, we realized the issue was in the physics gravity. Once we had all the nodes set and turned them into physics bodies, after the movement stage, they started to sink due to the physicsWorld gravity. So we had to completely disable gravity using physicsWorld.gravity = CGVector(dx: 0, dy: 0). This sets the gravitational acceleration in either direction to 0.

Collision Handling

The next step once we had the player and monster movement down was to create collision interaction. Since we already set every node up as a physicsBody, we were able to assign a raw collision value to each. We set up a function to track any collisions between nodes. By passing in the collision as a parameter we could check if the collision was between the player character and another node or between another node and the player character. Depending on which node was the player, the opposite node would be passed to a separate collision handler function. This collision handler function accepts the node and checks its "node.name" attribute to determine whether it is a coin, monster, or a door. In the case of a wall, each wall node's collisionKeyBitmask attribute is associated with the player character's categoryBitmask value. So wall collisions are inherently taken care of. Once we determine the type of node the collision handler was passed, we can implement logic in each case as shown below.

```
func didBegin(_ contact: SKPhysicsContact) {
    print("here")
    if contact.bodyA.node == character && contact.bodyB.node != nil {•••
    else if contact.bodyB.node == character && contact.bodyA.node != nil{•••
}

func playerCollidedWithNode(node: SKNode){
    if node.name == "coin" {•••         }
    else if node.name == "monster" {•••         }
    else if node.name == "door" && numCoins == 0{•••         }
}
```

Animations

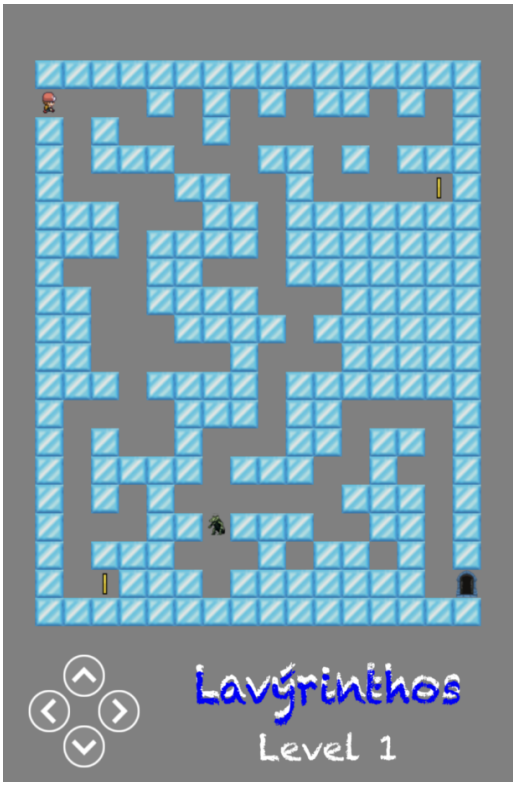
The game mechanism was complete. The last thing to do was to polish up some smaller features of the app. We wanted to add animations to our monster, character, and coin sprites. By downloading and splicing a PNG format sprite sheet for each sprite and adding them into their own TextureAtlas arrays, we were able to animate them directly during node creation using the SKSpriteNode(imageNamed: String) constructor of SKSpriteNode.

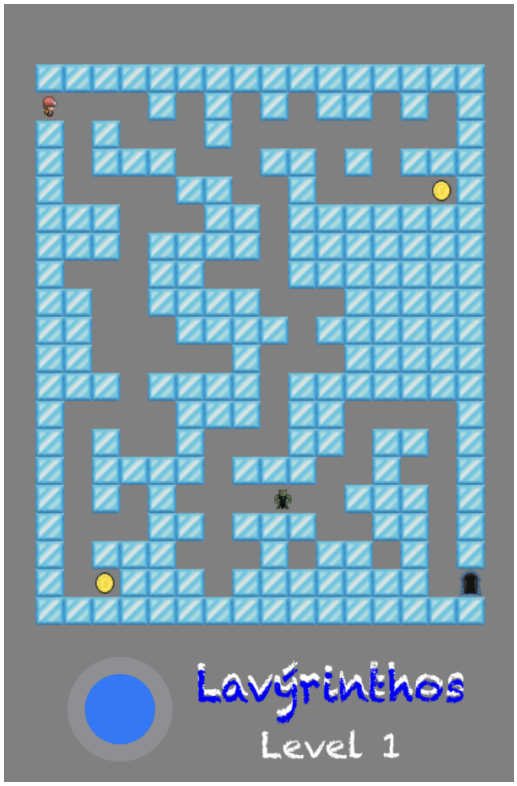
```
character = SKSpriteNode(imageNamed: charTextureAtlas.textureNames[0] as! String)
var coin = SKSpriteNode(imageNamed: coinTextureAtlas.textureNames[0] as! String)
var monster3 = SKSpriteNode(imageNamed: monsterTextureAtlas.textureNames[0] as! String)
```

Joystick Stretch Goal

One stretch goal we were able to meet is the implementation of a Joystick over buttons using the following repository: https://github.com/rebeloper/JoystickTank. This Analog Joystick was fairly simple to use. The joystick was already developed. As per instructions, all we had to do was drag the AnalogJoystick.swift file into our project and then create our own trackingHandler function to move OUR player character according to the joystick data. One unexpected behavior we had to deal with was the speed/velocity at which the player moved via the data provided by the joystick handler. So we had to multiply the speed by a factor of .05 in order to make it a usable feature. Below are screenshots of the code and UI before and after the joystick.

```
func setUpJoystick() {
    joystick.substrate.color = UIColor(Color.gray)
    joystick.stick.color = UIColor(Color.blue)
    joystick.position = CGPoint(x: -200, y: -450)
    joystick.trackingHandler = { [unowned self] data in
        self.character.position = CGPoint(x: (self.character.position.x + (data.velocity.x * 0.05)), y: (self.character.position.y +
            (data.velocity.y * 0.05)))
    }
    addChild(joystick)
}
```


Lavýrinthos
Level 1


Lavýrinthos
Level 1