Final Project Report

Introduction

"Gyro Fighter" is our newly developed app that provides a new spin on the traditional "Asteroids" game. Instead of the player using a joystick or buttons to maneuver the ship, we utilize the core motion features of the iPhone for the tilt-to-move control of the ship. In this report we will discuss our initial ideas, goals, our dive into Spritekit, core motion controls, fine-tuning the motion, progression of game functionality, and our final product with what goals we were able to attain and what goals we had to scrap throughout development.

What Led to a Gyroscopic "Asteroids" Game

Initially, we were deciding on what unique feature to have on our app, and what kind of app it would be. After determining that we would create some kind of game, we settled on somehow using gyroscopic motion for the controls, which led us to many avenues of ideas such as a marble maze or a racing type of game. A marble maze seemed difficult, since we foresaw some challenges implementing the physics (we did not know at this point that spritekit can handle these physics quite easily). A racing game seemed difficult since we would have to implement some kind of opponents. We decided to make an asteroids-type game that would accelerate the ship with the gyroscope, as we felt we could make the best product with this idea.

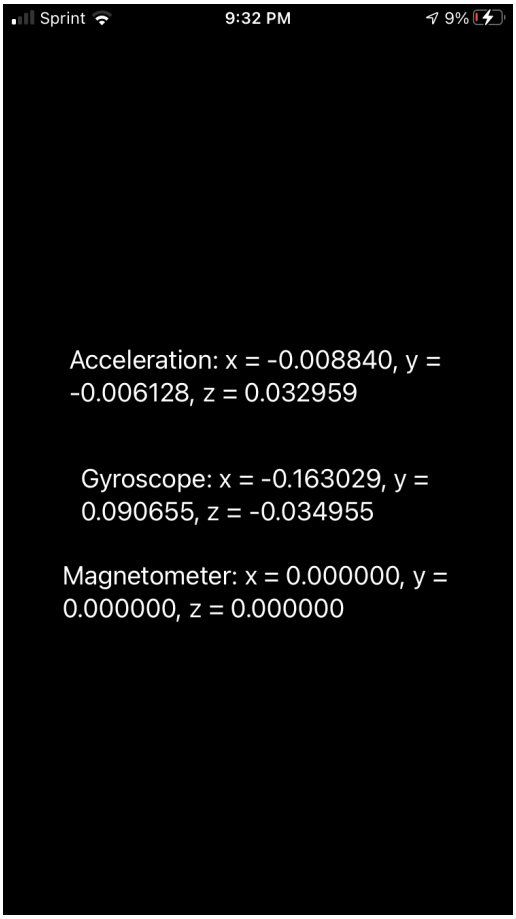
Minimum and Stretch Goals

Our initial goals were mainly to successfully implement gyroscopic controls for the game, having enemies/asteroids appear from off-screen, to keep a score, and to notify the player when the game was over. We also wanted to have the ship wrap across the screen similar to the

arcade game. Stretch goals included having the game ramp up in difficulty as time went on, having different types of enemies with different trajectories and interactions, keeping a highscores list, and aesthetic touch ups like particle effects and sleek menus.

Testing and Applying CoreMotion

To be able to properly utilize gyroscopic controls, we had to use one of Swift's features, Core Motion, to read the values of a device's accelerometer, gyroscope, and magnetometer. We created an app that read these values, and outputted them on a screen at a reasonable time interval.



We studied these output values to see how best to utilize them. We determined that the accelerometer is what we would use for the controls. The magnetometer seemed to be for an entirely different, compass-like function, and the gyroscope showed a rotation rate, which wouldn't translate well into our controls as much as the accelerometer. Even though we use the term "gyro controls" when discussing the game, it is really using the accelerometer field in core motion, not the actual gyroscope field..

To make this the base of our player movement, we had to use the values of the accelerometer to calculate an acceleration vector for the ship. To have the movement work on a simulator as well as a phone, we had different code that would have the ship move by a touch on the screen.

```
524    func getVec() -> CGVector? {
525
526        var ret : CGVector? = nil
527
528        if let accelerometerData = motionManager.accelerometerData {
529
530            let accx = (-accelerometerData.acceleration.y) + xAccelAdjust
531            let accy = (accelerometerData.acceleration.x) + yAccelAdjust
532
533            if (abs(accx) + abs(accy) >= moveThresh) {
534                // accelerate in tilted direction
535
536                player.physicsBody?.linearDamping = accel
537                ret = CGVector(dx: accx * xmult, dy: accy * ymult)
538            } else {
539                // stop accelerating, slow to halt
540                player.run(SKAction.setTexture(SKTexture(imageNamed: "ship")))
541                player.physicsBody?.linearDamping = decel
542                ret = CGVector(dx: 0.0, dy: 0.0)
543            }
544        }
545        return ret
546    }
```

Function that utilized CoreMotion values to determine vector for the ship

Movement Tweaks and Rationale

The implementation of tilt to move controls was successful, but it needed a bit more tuning. At first, the ship moved like a marble, making it very difficult to start moving, stop moving, and change directions. We fixed this using the linearDamping property on the ship's physics body. This damping value changes based on whether the ship is speeding up or slowing down. Additionally, we check to see if the ship's speed exceeds a speed limit, and clamp its velocity so that it does not get out of control. These changes make the ship more reliable to move around, while still giving the sensation that it is floating in space.

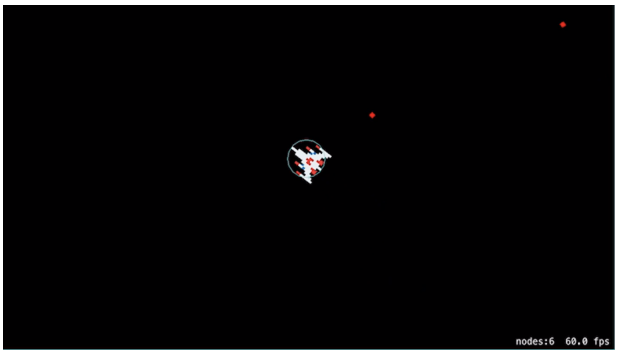
After these changes, the ship still didn't feel quite smooth and would jitter around a lot. We found that the accelerometer is a bit erratic and has lots of small random noise. We also realized that people don't really have super-steady hands, so we decided to implement a tilt threshold that would ignore tilt below a certain magnitude. This makes it much easier to stop the ship and make slighter movements.

Accelerometer Adjustments

One issue we had during testing was with keeping the phone level in order to have control over the ship. We decided to implement a function that offsets the incoming accelerometer data to balance out the current tilt. This effectively lets you reset the "neutral position" of the phone so you can hold it more comfortably. It's not perfect however, as it only works if the phone's screen is facing up at the player; it cannot be facing down (such as when lying in bed). We tried to make this function more robust, but the mathematics were a bit too complicated for us.

Bullet System

The most challenging thing about implementing a bullet system was having the bullet's start at the center position of the ship and having it shoot in a direction that depended on the rotation of the ship as well. With some trigonometry and playing around with some values, we were able to have the bullets successfully shoot from the ship. However, problems arose when attempting to have the bullet and asteroid disappear when hit. Some bullets seemed to phase through the asteroid when shooting a barrage of them at once, and the asteroid would disappear from the screen but it's physics body would still be present, causing the ship and bullet to collide and interact with it. To resolve this problem, we had to conform to an SKPhysicsContactDelegate, which allowed us to use collision bitmasks to categorize sprite interactions. We also had to thoroughly manage our sprite nodes, as it seems like the physics we delegated to our bullet seemed to apply to only the most recent bullet sprite node in the game scene. After reviewing our node management, our issues were solved, and we were successfully able to shoot and destroy asteroids with bullets. To keep the players from infinitely rapid-firing, only 3 on-screen bullets are allowed.

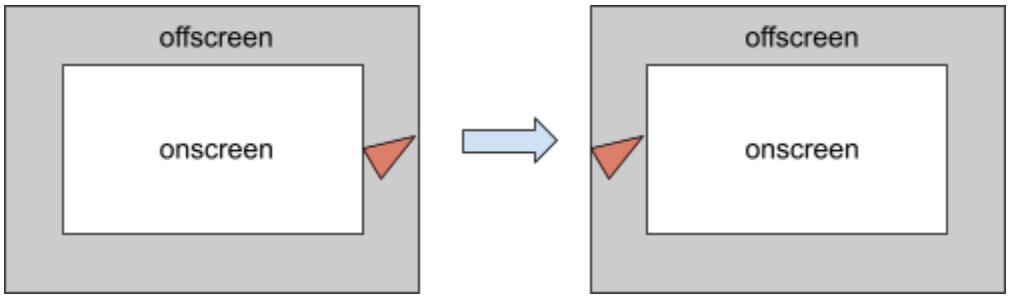


Bullets were shooting 90 degrees off during early stages of development

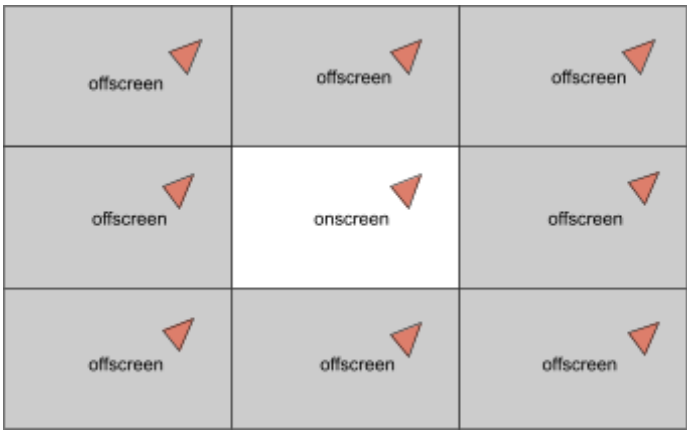
Screen Wrapping vs Walls

We had wanted to include a screen-wrapping feature as seen in the original Asteroids game, but there were several problems. There is no built-in way (that we could find) to let sprite nodes smoothly wrap from one end of the screen to the other.

One possible implementation would be to wait for the ship to completely cross one border edge, then warp to behind the opposite border edge, before emerging back on screen. This novice approach could open the door to a lot of issues. It would be possible to keep the ship completely off-screen, which generally looks unpolished and lets you get hit by unseen asteroids.



A more in-depth approach would be to have 8 “parallel universes” surrounding the main screen, each with their own copy of the ship, mirroring the same movements. When a new ship comes into view, it becomes the new player ship, and is able to fire the bullets.



There are a plethora of edge cases to check for, such as when the ship is halfway off screen and another is halfway on screen, then determining which one shoots. Implementation of

this would have required a lot of time and work that did not seem to be worth it. We scrapped the idea entirely and instead made the edges of the screen into walls that the ship could bounce off of. This turned out to be a fun way to keep things interesting, as a player retreating to safe corners would get thrust back into the action.

Game Balancing

Gyro Fighter, like Asteroids, is meant to be a difficult game. In early development, the ship would always face the same direction as the phone tilt, but this actually made the game very easy and did not require the player to move around very much. Now, the ship faces the direction of its velocity. The player only controls the acceleration, so they have to account for the fact that changing directions, and thus their aim, will take more time. This slight change makes the game much more difficult to master.

The game constantly gets harder the further along you get. The speed and frequency of the asteroids increases with player score. This keeps it from feeling monotonous and harkens back to arcade-style games. Every “game over” screen teases you with your own high score to get you to try again and get better.

Graphics

In the graphic design of our game, our goal was to make the different sprites clear and easy to see on a phone. We designed the ship to be more triangular to make it easier for the user to determine the orientation of the ship for shooting. Using two ship sprites, we were able to depict the ship firing its thrusters when moving around the screen. The hearts on the top left of the screen contrast the colors of the game so that the user can easily see the amount of lives that

they have. These hearts, along with the recentering button, were made slightly transparent so as not to obstruct the game objects. In the design of the asteroids, we created three different sprites, and then through the code, each spawn would have a different size, trajectory across the screen, and spin to make them look more realistic. For more excitement, we created explosion particle effects using SKEmitterNodes for when the ship and asteroids explode.

Ship sprites used for thruster effect in movement.

End Result

In the end, we were able to achieve most of our goals for this game. From the use of Core Motion to the implementation of the general movements of the game, we learned how essential mathematics and physics can be in game design. We also learned how in creating sprites how we must consider the overall look and feel of the game, focusing more on how the shapes, colors, and opacity of different sprites can effectively communicate what the sprites are themselves. With the final touches done, we are proud of the effort we've put in our game, and hope others enjoy it as well.