Intro

My senior project is a game called *Naturey Snake*. It is developed for the iOS platform and optimized for the iPhone 6S. It is a remastered version of the Classic Snake game. The environments have been enhanced to be soothing and relaxing to the player. The player is a “snake” in first person’s point of view. The goal is to survive and get the highest score possible. If the player runs into a wall or their own tail, the round is over. The player has to find and consume the banana in the game level to increase their score. Every time a banana is consumed, the player’s tail will gain a segment. Therefore, the higher the player’s score, the longer the player’s tail, and it becomes increasingly difficult to avoid crashing into the tail. The player is given the ability to jump to avoid running into their tail.

Features

This iOS application is meant for players that want a game they can whip out of their pocket and play for short intervals. It is made to be relaxing and soothing for the player to enjoy. The environments are detailed and the music suggests a very peaceful environment. In the daytime, the music is a soothing score from the game *Final Fantasy XIII*, and in the nighttime, it is a soundtrack of birds and other creatures in the forest. The scenery includes trees, mountains with lava erupting out of them, and rain sprinkling down from the sky. Depending on the time of day the player decides to play, it is either morning, with a gorgeous blue sky, or it is nighttime, with a mysterious planetary horizon.
Early Development

There are two options for graphics rendering on the iOS platform. The first option is using the Metal API. Doing it this way will allow for much higher performance since Metal is a very low level API with very little overhead. In fact, Metal has substantially lower overhead than OpenGL. The second option for implementing a game like this is using SceneKit. It is an Apple proprietary framework that turns graphics rendering into a tree hierarchy using nodes. I chose the second option, because of two reasons. First of all, SceneKit is very well documented and has been around since 2012, which means there are a lot of help on StackOverflow. The Metal API was introduced in September of 2014, the documentation is sparse and there is barely any discussions online to help in one’s project development. Secondly, SceneKit’s learning curve is nowhere near as high as Metal’s. Learning how to use Metal effectively to maximize on performance has a higher complexity than learning OpenGL. Therefore, we will be using SceneKit.

Technical Difficulties

Using Gyroscope to Rotate Camera

We may face some difficulties in trying to rotate the camera using gyroscope data. There are two fields in an SCNNode from SceneKit that one can modify to achieve a change in rotation. The first is the Euler Angle and the second is the rotation matrix. Changing the Euler Angle of the camera node does not change the rotation of the camera permanently. It temporarily allows the user to turn, but it snaps back to the center in a short while. The next thing we may try is modifying the rotation matrix directly, but that raises a problem where the camera moves away from the origin. If the camera is away from the origin, the rotation will be with respect to the origin thus making the user rotate around the origin instead of rotating in place. We can achieve the proper rotation that we want by taking an extra step.

The way we solve the camera issue is by using the gyroscope data from the iPhone to rotate an arbitrary object, which the camera is locked on looking at. We first translate the
arbitrary object a constant amount, then use the iPhone’s gyroscope data to rotate it with respect to the origin. Since we had our camera locked onto this arbitrary object, the camera will continue looking at it. Therefore, my camera will rotate in place looking at the object it is fixed on. Having this arbitrary object also eases the implementation of another feature. How does one get the camera to move in the direction it is looking at (moving forward)? What we can do now is have the arbitrary object’s coordinates minus the camera’s coordinates. Now, we have a vector in the direction the camera is looking at, multiply that vector by a small constant to move in that direction.

**The Mini-Map Feature**

Another problem we may have when implementing this game has to do with the mini-map. Since we have a main view, it displays the entire scene from the point of view of the camera. The mini-map is in charge of showing the game level in a top-down view. There may be other ways to go about completing this mini-map, but we will accomplish this by adding a sub-view to the main view that is going to be located in the lower left corner of the screen.

To create a top-down view of the game level, we create a separate view, a sub-view, from the main top level view. This sub-view should have its own camera and its own copy of any moving objects. The camera needs to be put directly above the (0, 0, 0) location in the game world, and rotated so it is looking straight down. By doing that, the sub-view will now display the
game level from a top-down perspective. The static objects such as the ground and the fences are initialized at the start of the application, so the sub-view would be able to see them too. However, moving objects in the game world need to have an instance in the main view and an instance attached to the sub-view. For example, if the player gains a tail segment or the player is moving around, the location updates in the main view is not visible to the sub-view. Therefore, the instances of moving objects in the main view need to be updated as well as the instances in the sub-view. Now, all that is left is to place the sub-view, which is the mini-map, in a location that is unobtrusive to the gameplay. There may be other ways to implement a mini-map, but this way works well, so this will be our solution.

**Implementation of the Tail**

The last interesting challenge we will overcome is related to the implementation of the tail. When the player eats a banana, the player gains a tail segment to the end of their tail. Also, the player moves faster and faster as he/she collects more bananas. The problem arises when we try to update the location of the tail segments in the game world. Since there is a set rate to how fast every non-static game object updates, the increase in the player’s movement speed may cause an overlap with the player’s location in the game world when the tail segments update. As in the player’s position might not have moved away yet, but the tail segment already updated to the player’s position. Since the player loses if the player comes in contact with their tail, the player would lose illegitimately in this instance. We can use the distance between the player’s location to the location of the tail segment directly behind the player to remedy this issue.
The way we solve this tail segment overlap issue is by using the distance between the camera and the tail segment directly before it. The tail segment that is directly before the camera’s location is updated by giving it the camera’s previous location, so the tail will always be following the camera. What we need to do is have a check to make sure that the tail segment’s new location is a safe distance away from the camera’s current position so overlap does not happen. We will use a variable to store the safe distance floating point value so on every update, we can compare the tail’s segment’s new location to that value. If it is further than or equal to the safe distance, you can update, otherwise, wait until the next update. We now have solutions to all those technical problems we had before.

**Particles**

Implementing particles in our game is rather straightforward. SceneKit allows for the creation of particle systems by adding an SCNParticleSystem to an SCNNode. We need to modify the various settings of the particle system to get the visuals that we desire for our game. For example, there are settings such as acceleration, life span, birth rate, and etc. When we eat a banana in game, particles explode from the eaten banana. The particles naturally spawn and accelerate in random directions starting at a given location, but we do not want the particles to be flying randomly. Since the particles spawn from the location of the eaten banana, that also happens to be the same coordinates as the location of the player. The player will not see many particles if the particles were to fly away in random directions. Therefore, we want to direct them to fly in the direction that the player is heading towards. We can accomplish that by using the same method as moving the camera forward. We take the vector of the direction the camera is moving in, turn that into a unit vector, multiply it by a constant, and set that final vector to the acceleration of the particle system. With that done, the particles will now shoot off in the direction the camera is moving in thus allowing the player to see more particles.