– COS 426 Final Project Report Spring 2021 –

Turtle Trek: An Infinite Swimmer

Natalie O'Leary and Isabel Greene
Professor: Felix Heide

May 10, 2021

Abstract

This project aims to build upon the infinite runner style games with obstacles and coins to collect. We hypothesized that we could adapt the Minecraft style infinite terrain generation to create a compelling underwater environment that is less constrained than an infinite runner, but is still rife with obstacles and collectibles. This style of game also allows for enemies with more interesting movement patterns, rather than just obstacles that come directly toward the player.

1. Introduction

The goal of this project was to create a more flexible and dynamic version of an infinite runner style game. Rather than having a straight track along which to run, we wanted to have a character that could go left and right as much as they want as well as up and down. We decided that under water was the ideal location for a game like this, since a swimming animal has fewer limitations on directions for motion. For this reason, we chose a turtle as our game avatar, and we decided that the premise of the game would be for the turtle to collect it's babies, as though they had just hatched and entered the ocean for the first time. In order to add difficulty to the game, we added obstacles and enemies that float around in the water as well. If a player hits and obstacle or enemy, they lose a life. The foundation of our infinite terrain implementation comes from a tutorial by Josh Marinacci which uses simplex noise to generate randomly elevated terrain [4]. We adapted the methods from the Dreamworld project from COS 426 Spring 2020 in order to dynamically generate blocks of this terrain in order to give the appearance of infinite terrain [3]. What our project does that others don't is a more rigorous interaction with the environment. While the Dreamworld simulation generates beautiful images, the avatar character passes through everything in the scene with no accounting for collisions. Detecting collisions in this situation is difficult because everything moves relative to the turtle, while the turtle is actually staying still. This is likely why previous iterations of this method have not implemented any collision detection. We hope to create a more stimulating game environment by having prizes and enemies come from all directions that the player can interact with. This ability to interact with the environment and objects creates a more immersive and realistic game experience.

2. Methodology

2.1. Infinite Terrain Generation

We used a Mincraft style implementation to give the illusion of infinite terrain in our game by having our avatar sit in the middle of the simulation and never move except to rotate in different directions. The terrain generates itself around the avatar and moves in the opposite direction of whatever way the avatar is facing. This was achieved with a few key steps.

2.1.1. Turtle Movement Our main turtle model comes from a Mesh created by DigitalLife3D and published on sketchfab.com [2]. The turtle animation which comes from the models GLTF file, is run continuously on a loop while the turtle sits in the middle of the simulation, which gives the illusion of constant forward movement. The turtle is fixed at (0,0,0) however, it is able to rotate around that point. These rotations are triggered by a press of the up, down, left or right key on the keyboard. We did this by creating eventlisteners that checked for keyup and keydown events. When a key is pressed, we set a global variable to denote the time at which the key was pressed, and when a key was released, we then set a boolean denoting that the key was pressed to false. Then, each time the turtle was updated, we used the keypresses to determine turtle rotation and terrain position. When up or down keys were pressed, the turtle was tilted up or down in place, however it is actually the terrain below that moves up or down, while the turtle remains in place. When the left or right arrows are pressed, the turtle is once again rotated in place, this time around the y axis, and the terrain below is rotated the opposite way, to give the illusion that the turtle is turning and going forward in a new direction, this method was adapted from the Dreamworld project [3].

2.1.2. Simplex Noise Terrain The generation of realistic looking terrain dynamically was done using a method proposed by Joshua Marinacci [4]. We used the SimplexNoise package from npm to generate flat noise planes. We then stacked these noise planes on top of each other to create interesting patterns. Then we used this output as a heightmap, to map from noise areas to y values for the terrain. This is done by changing the z values of the vertices according to the heightmap values, which translates into upward y values in the scene. We also added some vertex jitter in the x and z values to give the scene a more jagged and unorganized look. We wanted to achieve a coral reef style terrain, so the flat parts of the noise generated terrain are tan and sand textured, while the elevated terrain is blue and purple, to mimic underwater plants and rocks. We mapped color to the faces of the plane by determining the maximum value of the z vertices for this plane and setting the color accordingly, with the purple colors getting lighter as the z value got higher.

2.1.3. Terrain Manager The terrain manager controls the movement of the generated terrain blocks. We created 9 of the blocks described in the previous section, which are integrated seamlessly together using an offset values in the noise function, as if each of the 9 planes was actually part of one larger plane with continuous position values for each section of the plane. As the up, down, left and right keys are pressed to move the turtle, what is actually updated is the x,y and z values stored in the scene. These values are then used by Terrain Manager to move the pieces of the terrain in the corresponding direction. When the blocks of terrain have shifted too far in the x or z direction, they are then removed and replaced with new blocks on the opposite side that they disappeared over.

2.2. Plant, Obstacle and Baby Generation

All obstacles, objects and prizes in the scene are children of the Terrain Plane objects, so that they will move at exactly the same speed as the terrain, so that from turtle POV, it appears as though they are remaining in place while we travel toward them. The baby model was taken from poly-google.com [1]. For the babies and obstacles, this was done by simply creating group classes and adding them to each Terrain Plane, generating their positions randomly relative to the parent, so they appear scattered in the scene. The enemies are also generated in a similar way, but in order to make them more difficult to avoid, we also created a small animation loop such that as they move toward the turtle, they also swim in a tight circle. There is also an enemy that remains in place but circles the turtle in the center. This enemy can never actually collide with the turtle, however it gives the illusion of imminent danger to the game player. In order to include plants and ocean rocks in the scene, we needed to map them to the ground of the simulation, rather than having them float in the scene. We used the same heightmap that we used to generate the terrain in order to determine the appropriate y value at which to place a new plant on the ground according to its x and z values.

2.3. Collision Detection

Detecting collisions in this scene was challenging, as the objects move with the terrain planes, so their world coordinates are changing, but their local coordinates remain static as if they are not moving at all. We first attempted to implement collisions by creating bounding boxes around the objects that are update as the object moves, however this was unsuccessful at detecting collisions because the positions were not able to be quickly and accurately updated according to the world position of the obstacles. What ended up working was creating an actual box mesh around the turtle and around each object in the scene. We set the visibility of these boxes to false so that they were invisible in the scene. However, we used these boxes as a bounding box for our raycaster, which detects collisions on each update call by shooting rays from the center of the object to each vertex in the cube. This RayCaster method was adapted from a threejs tutorial by stemkoski on GitHub [6]. If the ray intersects with any object in the object collision list before reaching the perimeter of the box, this indicates a collision. For baby turtle collisions, this means that the turtle disappears from the scene and is placed behind our turtle, as though it is now following it's mother. This also triggers the turtle score count to be incremented, which appears in the upper left corner of the screen. For collisions with enemies, the life counter is decremented, and updated in the left corner of the screen as well. A collision with an extra life object will increment the life counter again.

2.4. Caustics

Caustics describe the refracted and reflected light patterns beneath the surface of water. In order to create the illusion that the turtle is underwater, rather than floating in the air, we want to implement these caustics in our simulation. We were informed of the possibility of this undertaking by an article written by Evan Wallace [7], in which he includes a live demo of interactive caustics. However, these caustics are hard coded into the scene, and only work for a small box with a single sphere in the scene. This would obviously not work for our moving and changing randomly generated terrain. So, Martin Renou proposes a different method in his article "Real Time Rendering of Water Caustics" [5]. He proposes creating a shadow map of your environment to determine object depth, before mapping the caustics texture onto the scene. His scene does not have moving parts in it however. We have yet to implement this part of the simulation at the time of writing, however we hope to be able to incorporate this method into our final demo.

3. Results

We set out to create a reactive, collector style game in an infinite terrain and we believe that we achieved this goal very well. We successfully created objects and obstacles floating in the infinitely generated terrain to collect and avoid. We also made the game reactive to the different types of objects, to trigger different reactions including score increment, life decrement and life decrement. At the time of writing, we have not yet successfully implemented caustics, however we believe that this is possible for our final implementation, which would create a unique and interesting visual environment that has not yet been done to our knowledge.

4. Discussion

The approach we took is a novel way to implement an infinite runner style game, and we are very pleased with the way that it turned out. The caveat of this method is that the objects all need to move in a predictable manner, since they need to be wired onto the terrain in order to move toward the turtle, while still being able to be dodged or navigated toward by the turtle. This limited our options in terms of enemy movement. Ideally we would have liked to be able to implement enemies that swam directly toward the turtle that we could dodge, however they could only move toward the turtle at the same speed as the rest of the scene, or swim in random or circular patterns while moving toward the turtle in the scene. To directly move an object toward the turtle would mean the turtle could not avoid it, since the turtle doesn't really move. However, we believe that the random movement of the enemies does provide an interesting and difficult enough obstacle to avoid. Another aspect that we may need to test and change is game speed. Since it is just the two of us, we were unsure of game difficulty, however this could be increased by changing the game speed. The game as it currently stands is relatively slow, and a faster speed could make it more challenging for players. We learned a lot about Threejs models and GLTF animations in doing this project. We also learned about GLSL shaders and how they work in our implementation of caustics.

5. Conclusion

As stated earlier, we achieved or initial MVP and then some, and we believe we created a successful and interesting game. Both the visuals and the game dynamics are compelling and interesting. At this time, we believe that future additions such as caustics and game speed changes would be beneficial for future iterations of this game. We would like to not only change game speed, but to include a difficulty setting in the game that changes the game speed depending on what level the player selects at the beginning of the game.

6. Contributions

Natalie implemented:

1. Adaptation of the Terrain Manager from Dreamworld [3]
2. Obstacle and item generation
3. Collision Detection
4. Enemy generation and animation
5. life icon drawings and displays on the screen

Isabel implemented:

1. Object and turtle animations
2. plant generation
3. turtle rotations and movement
4. Start Screen and buttons

References

[1] P. by Google, "Turtle," October 2017, https://poly.google.com/view/fklSEvGm1Q8.

[2] DigitalLife3D, "Model 50a - hatchling hawksbill sea turtle," SketchFab, 2018, https://sketchfab.com/3d-models/model-50a-hatchling-hawksbill-sea-turtle-e69458ef8176402d919df421e444da86.

[3] E. P. L. Y. Lauren Johnston, Joanna Kuo, "Dreamworld: Meditation simulation," GitHub, May 2020.

[4] J. Marinacci, "Low poly style terrain generation," medium, November 2018, https://medium.com/@joshmarinacci/low-poly-style-terrain-generation-8a017ab02e7b.

[5] M. Renou, "Real-time rendering of water caustics," Medium, August 2020, https://medium.com/@martinRenou/real-time-rendering-of-water-caustics-59cda1d74aa.

[6] stemkoski, "Collision-detection," July 2013, https://github.com/stemkoski/stemkoski.github.com/blob/master/Three.js/Collision-Detection.html.

[7] E. Wallace, "Rendering realtime caustics in webgl," Medium, January 2016, https://medium.com/@evanwallace/rendering-realtime-caustics-in-webgl-2a99a29a0b2c.