Midnight Mountains - Final Project Write-up

David Paulk, Channing Huang, Christopher Giglio

Honor Code: This paper represents our own work in accordance with University regulations.
Signature: /s/ David Paulk, Channing Huang, Christopher Giglio

1 Introduction

1.1 Goal

The aim of our project was to implement a simple video game with a focus on quality graphics. The user's objective is to collect as many coins as possible while navigating a mountain terrain. When the user collides with the mountain, the game ends. As the user performs the coin-collecting task, the scene changes in several ways: a day-night cycle reflects the passage of time, the coins float about their spawn-points according to a periodic function, and the player gains speed as more coins are collected. Also, the light emitted by the sun, the players, and the coins affects the surface of objects in the scene such as the mountainside and the fog.

1.2 Previous Work

Our project builds on the work of several other projects. The Perlin noise generator used for generating heights for the mountains was adapted from `http://threejs.org/examples/webgl_geometry_terrain.html`. The day-night cycle that makes the sky dynamic was implemented using [a WebGL demo] (need citation) as start code. The game music was taken from the Mario videogame (need citation).

1.3 Approach

We tried to implement normative mapping using a mountain texture. Our implementation approach for adding this feature did not work because it could not be used simultaneously with the fog and continuously rendering the mountain texture as new mountain terrain is spawned proved to be computationally expensive, resulting in noticeable latency.

2   Methodology

The following subsections describe each part of the video implementation in further detail. The parts include sky rendering, coin rendering, player perspective, navigation controls, mountain terrain spawning, collision detection, and extras (i.e. game menus and audio tracks).

2.1   Sky Rendering

2.2   Coin Rendering

It is the goal of the user to collect as many coins as possible before crashing into the mountainside. A coin is represented in Javascript as a THREE.Object3D in the shape of a sphere and contains a THREE.PointLight to illuminate it, which is particularly convenient during the night when it is harder to navigate the terrain. The coins are animated by moving the x, y and z coordinates of their positions according the periodic function, relative to their spawn position.

$$x+ = sin(time * 7) * 3$$

$$y+ = cos(time * 5) * 4$$

$$z+ = cos(time * 3) * 3$$

The position update is small enough so that animating them does not make it more difficult for the user to collect coins. When the user's player collides with a coin, the score increases by one and the coin is removed from the scene. One implementation challenge that we faced was figuring out how to spawn and update coins in a way that would avoid noticeable latency. While one approach would be to limit the amount of memory occupied by each coin, this approach would prevent us from adding extra coin features such as lighting (glow). A better approach for avoiding latency that we chose was to limit the number of coins present in the scene at any point in time. We did this by first adding a maximum distance from the player that the coins can be spawned and updated at (7000 units). Second, we limited the number of coins that could be present at any time to 10, so that after 10 coins are spawned, no more can be spawned until one or more is collected by the player.

2.3   Player Perspective

2.4   Navigation Controls

The controls are implemented with a modified THREE.FirstPersonControls. The code for looking around with the mouse is maintained; however, the movement itself is overhauled. We store the player's current velocity as a vector, and accelerate or decelerate by a constant value in each direction based on input commands. The velocity is capped, and beyond this cap, the velocity vector is scaled down to fit the cap. Additionally, the player's movement speed will

decay over time if acceleration is not continued. Currently, velocity decreases by 1 percent every frame, though this is of course adjustable.

2.5 Mountain Terrain

The mountain terrain was initially based on the THREE.js WebGL Geometry Terrain demo at `http://threejs.org/examples/webgl_geometry_terrain.html`. We retain the Perlin noise generator used by the demo, but have overhauled most other aspects of the demo.

The mountain's material was made using an extended version of the MeshLambertMaterial in the THREE.js libraries. Non-photorealistic rendering of the mountains was added through toon shading and outlines. Toon shading was achieved by quantizing the $n \cdot l$ value when calculating Lambertian reflectance. Adding the outline was initially difficult, due to the many difficulties discussed in class and the need for complex ray caster operations to determine where to draw lines. However, we found a clever hack for outlining shapes that made the process significantly easier, using the method at `https://stemkoski.github.io/Three.js/Outline.html`. In order to add outlines, this hack duplicates all meshes, uses a basic black texture for the duplicates, and scales the duplicates up by a very small factor (we use 1.01). Then, we add the duplicates to the scene but only render the back side of the faces. This makes it so that the only parts of the duplicate that are visible are the parts slightly beyond crests in the mesh, exactly where we want outlines.

2.6 Mountain Terrain Spawning

We also added infinite spawning of terrain so one can fly without worrying about hitting a boundary. In order to achieve this without using too much memory or processing power, we divide up the world into square tiles 1500 by 1500 units large. In the scene, we only include tiles that are within a certain distance to the player, and we dynamically add or remove tiles as necessary as the player moves. Careful offsetting within the Perlin noise function ensures that the terrain is continuous even though each tile is generated separately.

2.7 Fog

Fog is an important part of our game, as not only does it add ambience, but it also conceals the fact that we are adding and removing chunks of terrain as the player moves. However, adding fog proved to be a harder task than expected. Although THREE.js comes with a default implementation of fog, it only allows you to set the fog to a static color. Since we have a dynamic and multicolored background due to the day/night cycle, a static color does not suffice. Thus, we manually created ShaderMaterials for the mountain terrain and outlnes with an enhanced version of fog that calculates fog color based on position, orientation, and time in the day/night cycle.

2.8   Collision Detection

There are two main types of collisions we handle: collisions with the terrain and collision with coins. Collisions with the terrain are handled simply by casting a ray under the player and checking to see if there is terrain under the player. If there is not, then that means the player has intersected the mountain and we flag that the player is dead. Collisions with coins are handled by constructing an octree using the THREE.Octree library (`https://github.com/collinhover/threeoctree`) and inserting the coin positions into the tree. The octree can then be used to look up coins close to the player, and any coin within 200 units is considered to be picked up.

2.9   Stars

Stars are created by generating points at random on the surface of a sphere, and then placing white spheres at those points. The color of the star material is varied with time in order to make the transition between day and night smooth.

2.10   Extras

2.10.1   Menus

The start, pause, and death menus are implemented in HTML and overlaid atop the renderer. To pause rendering while menus are open, we stop the requestAnimationFrame loop by interrupting the animate() function and restart the loop after menus are closed.

2.10.2   Sound

Music and sound effects were added using the HTML5 Audio API.

3   Results

4   Discussion

5   Conclusion