# Core Crawler: Intermediary Report

*By Epi Torres and Jorge Zreik*

## Abstract

Several games incorporate grappling hook mechanics or 3D procedural generation; however, few games meaningfully use both. As a result, this project develops Core Crawler, a 3D game that implements grappling hook mechanics and procedural terrain generation in a way that produces a thrilling and replayable gameplay experience. To do this, we have implemented the marching cubes algorithm for procedural terrain generation, quaternion basis changes for camera logic, and raycasting for the grappling hook so far. In the future, we will implement physical simulation using verlet integration, collisions using uniform grids, and collectible spawning using rejection sampling. This approach has shown positive results in terms of load times, FPS, and initial feedback, and we plan to continue our evaluation as we continue development.

## Introduction

Although many games that use grappling hook mechanics or 3D procedural generation exist, few games make meaningful use of both. Recognizing this great opportunity, we decided that the goal of our project would be to create a 3D game centered around both grappling hook mechanics and procedural terrain generation. Throughout this game's implementation process, we have taken an approach that encourages fast-paced gameplay and maximizes the game's replay value, making it an excellent choice for individuals interested in a thrilling gameplay experience. Nevertheless, the low-stakes nature of this game's scoring mechanics means that even casual gamers can enjoy Core Crawler as well.

This report begins by briefly describing other related projects and how they influenced some of the design choices for our project. We then explain our project's approach and how it differs from other existing projects. Afterwards, we outline the different aspects of our project's current implementation and describe the future work that we anticipate incorporating in our final game. Afterwards, we present the different evaluation strategies that we have taken and plan to take throughout the ongoing development of our project. This paper concludes with final remarks regarding both the successes and limitations of our project.

## Related Projects

Our interest in procedural generation mainly stems from a YouTube video by Sebastian Lague titled "Coding Adventure: Marching Cubes" [1]. This video explains the concept of marching cubes in a succinct and compelling way, which made us interested in using it because the algorithm is intuitive and yields surprisingly effective results. In particular, Lague used the algorithm to procedurally generate terrain from an underlying scalar field of noise. This technique was extremely useful as the terrain can be shaped however we'd like through simple mathematical manipulations of the underlying scalar field (e.g., making more noise appear in the center or stopping all noise after a certain distance from the center). Aside from this video, we also drew some inspiration from games that generated terrain using a non-uniform grid, like Townscaper, but we did not end up pursuing the techniques used in those sources.

For our grappling hook mechanic, we were mainly inspired by Feng Lee's "Attack on Titan Tribute Game" (AoTTG) [2]. This is a video game from 2013 based on the anime and manga titled "Attack on Titan," where soldiers use grappling hook-like devices to maneuver around and fight giant humanoid monsters called titans. AoTTG is a fan project that differs from later, official games because of the compelling high-skill gameplay that results from its physically simulated and fast-paced grappling mechanics. Because of this, we aim to implement a similar mechanic in our game where players can grapple to a surface, swing on the grappling hook, and influence their direction using arrow keys. Other games that include grappling mechanics like Spiderman games were also an inspiration for us, but we decided to pursue more physically realistic mechanics like those of AoTTG.

# Approach

The first part of our project's approach involves maximizing the game's replayability by procedurally generating the game's environment. This approach differs from other grappling-hook based games like AoTTG because those games typically use static environments. Whereas static environments can become boring once the player has explored the environments in their entirety, procedurally generated environments virtually ensure that no two environments are ever the same, thereby keeping the game's map fresh and exciting for the player to explore.

The second part of our project's approach is to implement simple to understand and hard to master gameplay mechanics, like a fast-paced and physically simulated grappling hook. Although many related games (like the Spiderman games) incorporate grappling hook mechanics that are easy for beginners to pick up, such an approach both requires much more developer effort and lowers the skill ceiling of the game. Thus, our project's approach of using simple yet naturally skillful gameplay would better enable us to deliver a satisfying game experience within our project's narrow timeframe.

# Current Implementation

## Procedural Terrain Generation

The two main options we considered for terrain generation were noise-based generation with marching cubes and cityscape generation using non-uniform grids. To implement marching cubes terrain generation, we would first generate a 3D scalar field using a noise function like Perlin or Simplex noise. Then, we would loop over each "cube" of vertices and create faces according to a triangulation table. To implement non-uniform grid terrain generation, we would first create a non uniform grid of quadrilaterals through a process similar to the one described by Oskar Stålberg, the creator of Townscaper [3]. We would then extrude buildings from these quadrilaterals to create a city.

These two approaches differ from each other significantly in terms of both their underlying implementations and the types of terrain that they produce. Whereas the marching cubes approach can easily generate natural looking terrain with arbitrary shapes and high degrees of verticality, the irregular grid approach can effectively produce a more geometric, city-like environment. While this would have made for an interesting map, it would have required significantly more work because we would have had to create models and textures for buildings to make the terrain interesting, as simple quadrilaterals make for dull terrain. Because of this, as well as the large amounts of existing documentation on marching cubes, we decided to pursue the first approach for our game. This approach is easier to implement and generates more interesting terrain, such as the planetary core we envisioned for our game.

To implement marching cubes, we first generate a scalar field that represents the terrain. To do so, we first generate values from noise. The type of noise we use is Simplex noise due to its efficiency and ability to generate noise in three dimensions. We use the "simplex-noise" npm package for this because generating simplex noise can require significant amounts of code and it is a standard problem. We use this package to generate increasingly finer noise that we layer on top of each other. Then we shape the noise into a planet by multiplying the generated values by each point's distance to the center in order to reduce terrain in the middle of the planet. We then interpolate the noise with the first derivative of a gaussian curve that is located at a specific radius from the center in order to generate the surface and crust of the planet.

After generating the scalar field, we implement standard marching cubes as documented by Paul Bourke [4]. This technique loops over each set of eight scalar values in a cubic arrangement. By referring to an edge table and a triangulation table, we find the faces that must be generated within the current cube. Finally, we apply color to the mesh by setting each vertex's color according to the point's distance to the center.

## Camera Logic

To implement our camera movement, we had several choices to make from a design perspective. First, we could make a first person or a third person camera. Both approaches were similar and our code was modularized in such a way that changing this decision later on would be easy. Because of this, we decided on a first person camera in order to save us the effort of making a player model, but this can be changed later on with minimal work. Second, we could make the camera's up vector constantly point in the positive Y direction, or we could lock it to point to the center of our planet. While locking the up vector to positive Y would be significantly easier, the second option would be better for our gameplay since we want gravity to pull the player away from the center of the planet. Even though the second option was a significant challenge, we decided to pursue it.

To implement our camera, we first forked Three.js' PointerLockControls example so we could base our implementation off of that logic. This module allows for first person camera movement by translating mouse movement into camera rotations; however, it assumes that the camera's up vector points in the positive Y direction. To get around this, we first rotate the camera every frame so that its up vector points to the center of the planet. We accomplish this by applying the quaternion that rotates the camera's current up vector to the desired up vector (using the functions Object3D.applyQuaternion and Quaternion.setFromVectors). Then, we essentially change the basis of the PointerLockControls' logic by first rotating the camera so that its up vector points in the positive Y direction before it applies the mouse-derived rotation and then rotating the camera by the opposite of the original rotation afterwards. This allows us to have a first person camera whose "down" direction is always in the direction of gravity.

## Grappling Hook Mechanics

Since grappling hook mechanics rely heavily on mesh intersections, we considered two possible options. The first option involved launching a 3D hook object and detecting if the hook collided with the terrain's mesh. Although this option would produce more realistic grappling hook behavior, it would also be somewhat costly to implement because the terrain's mesh is complex and irregular. The second option that we considered involved raycasting from the center of the camera to detect mesh intersections. Although this option would be less realistic than launching a 3D object, it would be much more computationally efficient because it only requires handling intersections with the terrain's mesh at a single point.

For our implementation, we decided to use the raycasting option due to its greater efficiency and ease of use. To begin implementing the grappling hook, we first created custom event handlers for the right and left mouse buttons. When the player presses the left mouse button, a ray is cast from the center of the camera along its forward vector.

If the ray intersects with the mesh within some maximum distance from the player, the game animates a line mesh from the player towards the intersection point. The line mesh eventually reaches the intersection point after a preset delay, at which point the player becomes attached to the terrain. While attached, the player's maximum distance from the intersection point is fixed. If the player releases the left mouse button while attached, the player becomes unattached and the line mesh moves back towards the player. If the player presses the right mouse button while attached, the player gains a velocity towards the intersection point.

If the ray does not intersect with the mesh within some maximum distance from the player, the line is animated so that it moves away from the player until it reaches that maximum distance and returns to the player.

# Future Work

## Physical Simulation

To physically simulate the player and their interactions with the world through a grappling hook, we saw two possible approaches: using a physics engine such as Ammo.js or implementing our own gravity and collisions. Using a physics engine would allow us to more accurately simulate the physics; however, this approach would likely come with considerable and unnecessary overhead given the relatively smaller scope of the game. Although programming physics ourselves would be infeasible for a larger game due to the overall complexity of physical simulations, the small scope of our project would allow us to create physically motivated behavior relatively easily and without increasing the number of dependencies and the amount of abstraction in our project.

Because of this, we have decided to implement simple collision detection between the player and the planet and to simulate gravity and other forces ourselves. To simulate forces, we will use verlet integration as used in the assignments of the course. To detect collisions, we plan to run checks between the player (modeled simply as a sphere) and the mesh's vertices by using a simple uniform grid to limit the amount of checks needed.

## Random Collectible Generation

One naive approach to random collectible generation would simply be to randomly generate collectible positions that lie within the radius of the planet. Although this approach would be very easy to implement, it would likely cause issues where some collectibles clip the terrain mesh or are entirely inside it. Additionally, since the density of the planet's mesh is much lower towards its center, this approach might cause issues where too many collectibles are spawned in difficult-to-reach areas with low terrain density.

A more robust option would involve using the common statistical technique of rejection sampling. Essentially, this option entails generating a position within the planet created by moving forward from the normal of a random vertex (to avoid landing inside the mesh). If this random position is too close to any vertex, we can reject it and generate a new position; this would prevent the collectibles from clipping with the terrain. If the random position is too close to the center of the planet, we can reject it as well; this would prevent collectibles from spawning in areas where the planet's density is very low. Overall, although it would be more complex to

implement than a naive approach, this rejection sampling strategy would grant us much finer control over the density of the collectibles within the 3D environment. As a result, we anticipate implementing this feature by using this type of rejection sampling strategy.

### Reach Goals

In addition to physical simulation and random obstacle generation, we hope to incorporate a main menu that allows players to see the game's credits, adjust game settings, and begin a new game session. Additionally, we hope to allow players to customize the planet by allowing them to alter different parameters for the procedural terrain generation. Another potential avenue for future work on this game could include transitioning between multiple levels based on the player's score.

## Evaluation

While working on the currently implemented features, our main evaluation strategy was to examine both the game's performance and load times. Due to its fast-paced nature, our game must be highly performant in order to produce the smooth and crisp movement necessary to create a satisfying gameplay experience. Currently, the procedural generation of the planet takes 5 seconds to load on average; additionally, we consistently achieve 60 frames per second when the planet is loaded, even with very high terrain resolution. Both of these values indicate that the game is successfully meeting our performance expectations so far. Additionally, initial feedback from others regarding the game's current state has been positive, suggesting that our project's approaches will lead to a promising final product.

As we continue implementing additional features, we will continue to evaluate their impact on the game's overall performance in order to optimize these features when necessary. Additionally, we anticipate requesting external feedback from other individuals once the game's core features have been implemented in order to better evaluate our design choices and better polish the final version of the game.

## Conclusion

Throughout this report, we have explained the ongoing process that we have been taking to reach our project's main goal of developing a game that combines procedural terrain generation and grappling hook mechanics. So far, we have implemented the procedural terrain generation using the marching cubes algorithm, the custom camera logic necessary to navigate the unique environment, and the grappling hook using raycasting. In the coming days, we plan to implement physical simulations and random collectible generation for the final version of the game. Overall, our game's current version has been highly performant and has received positive initial feedback.

In conclusion, although several other games make use of procedural generation or grappling hook mechanics, our game combines those two features to produce a very replayable and fast-paced gameplay experience.

## Contributions

So far, Jorge has implemented terrain generation and camera logic. Because of his previous interest in marching cubes and familiarity with Sebastian Lague's marching cubes project, Jorge implemented both scalar field generation and the marching cubes algorithm. Based on Epi's original experiments with Three.js'

PointerLockControls, Jorge finalized the game's camera logic through further experimentation with quaternion mechanics.

So far throughout this game's development process, Epi Torres has implemented the game's grappling hook mechanics by integrating custom event handlers and raycasting; additionally, Epi had worked on the preliminary implementation of the game's custom camera logic, which Jorge was able to finish. In the coming days, Epi will be primarily responsible for developing the random collectible generation and programming the physically motivated behavior of the grappling hook.

## Honor Pledge

We pledge our honor that this report represents our own work according to University regulations. — Epi Torres & Jorge Zreik

## Works Cited

[1] Sebastian Lague, "Coding Adventure: Marching Cubes", May 6th 2019, YouTube, https://www.youtube.com/watch?v=M3iI2l0ltbE

[2] Feng Lee, Attack on Titan Tribute Game, https://attack-on-titan-tribute-game.en.softonic.com

[3] Oskar Stålberg, "Fairly even irregular quads grid in a hex," July 7th 2019, Twitter, https://twitter.com/osksta/status/1147881669350891521

[4] Paul Bourke, "Polygonising a Scalar Field (Marching Cubes)," May 1994, http://paulbourke.net/geometry/polygonise/