

## Introduction

---

This document describes a variety of performance concerns in relation to Isekai Castle Crusaders. Three performance issues that could impact a reverse tower defense game include an abundance of entities spawned, unoptimized routes for pathfinding, and unnecessary if-else statement branching.

## Abundance of Entities Spawned

---

One issue that could arise is the issue of spawning too many objects, whether they be ally troops, enemy troops, or towers. Since the map is larger than the view camera of the player, objects are more likely to cover a larger region of the map than the camera itself which could mean unnecessarily rendering assets that aren't even visible to the player.

To address this, I could implement conditional rendering based on the position of an object relative to the area of the player camera. In a 3D space, this would be calculated using a frustum, a portion of a cone or pyramid cut off from the top at a parallel conic section. This technique is known as frustum culling. In a 2D world such as my own game, using the camera's view area as a region to render objects would suffice.

## Unoptimized Routes for Pathfinding

---

In a game where entities need to calculate a path to follow, an algorithm must be able to calculate a path within fractions of a second. Naive solutions that run in  $O(V^2)$  often aren't suitable for even smaller maps with a 20 x 20 grid.

For this reason, I'll most likely need to use Dijkstra's or, even better, A\*. Both of these algorithms, when implemented optimally, perform better than a naive approach and result in time complexities better than  $O(V^2)$ . Thankfully, Godot has a built in object that handles these which I can use for my project.

## Unnecessary If-Else Statement Branching

---

In beginner-level games, we often take simple approaches to handle logic for state. For example, to handle movement, most tutorials implement an if-else statement within the physics rendering function of an object. This is suboptimal for handling complex branches of state, and we've seen how this can affect performance (looking at you Yandere Dev) when scaled to handle more than just movement.

To counter this problem, I'll be implementing a finite state machine for checking the state of an object and only executing a code segment when a condition is met. For example, if I want to calculate damage an entity deals to another entity, I would only need to check this when the "Attack" state is in action. By encapsulating logic into state, we can avoid as much conditional logic as we need to.

## Sources

---

- [1] <https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling>
- [2] [https://en.wikipedia.org/wiki/Viewing\\_frustum](https://en.wikipedia.org/wiki/Viewing_frustum)
- [3] <https://www.geeksforgeeks.org/dsa/a-search-algorithm/#>
- [4] [https://docs.godotengine.org/en/stable/tutorials/navigation/navigation\\_introduction\\_2d.html](https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_introduction_2d.html)
- [5] <https://gameprogrammingpatterns.com/state.html>

## AI Usage

---

N/A