Crowded Game Project Final Report

Zhengyuan Chen(7541236), Xinliang Lu(0822760),

Shaopeng Wang(1833545), Sichun Wu(9877886), Chengkai Zhu(1749145)

February 13, 2023

Abstract

This report introduces a game called “Purge” that features 5000 zombies with individual navigation abilities and collision detection. The development of the game included the creation of the map, NPC characters, HUD, and game logic. Optimization techniques were employed to address any frame rate issues. The experiment results show that the game demo consistently performs above 20 frames per second, and performance data is presented. The aim of the game is to successfully incorporate 5000 NPCs into a fun and playable experience.

1 Introduction

In this report, we present a game named *Purge*[1], which contains up to 5000 Zombies (Agents) in the game scene. Our objective is to not only enhance the enjoyment factor, but also ensure a seamless gaming experience by maintaining a satisfactory frame rate. The game was developed using the Godot engine 3.51[1]. As normally two cores are used for running a game, it is a big challenge for the CPU to simulate large crowds with collision and navigation. In the following sections, we will explain the problem we encountered in detail and present a comprehensive strategy employed to resolve them.

2 Related work

With the advancements in the gaming industry and crowd simulation algorithms, it has become possible to simulate large crowds in games. One notable example of this is *Assassin’s Creed: Unity* [5], which simulates massive crowds and allows players to interact with up to 10K agents in certain scenes. The developers achieved this through the implementation of a system called “bulk”, which contains thousands of cheap meshes and a pool of pre-spawned NPCs. When the player gets close to the cheap mesh, the mesh will be replaced with an NPC from the pool. NPCs at a far distance from the player are represented with lower resolution and basic reactions, while those closer to the player have a more advanced appearance and animation. The game only displays the NPCs that are in front of the camera. More recently, the Niagara system in *Unreal Engine*[2] has made it possible to display millions of characters in real-time, making simulation much easier through the particle generation system, which allows artists to create functionality without programming. Additionally, García-García, et al.[4] presented a serious game for the simulation of massive evacuations, serving as a useful planning and training tool for event organizers.

[1]Published at: `https://github.com/Sweet-john/Crowd_game`

Sophisticated technology now exists to effectively manage big crowd simulations in 3D games. But none of them actually have that many instances of individual agents at the same time. In order to get rid of the possibility that GPU could be the bottleneck, we decided to choose a 2D game engine for further experiments. To experiment with the possibility of creating a lightweight, easy-to-operate and fun 2D crowded game, we chose the Godot engine which could code with scripts for development.

3 Overall gameplay

In our game, the object for the player is to score as many points as possible before all the zombies are gone. After the zombies spawn, they will wait for a certain amount of time before setting off for one of the evacuation points on the map, then disappearing near the evacuation point. The player can score points by getting close to zombies to eliminate them. When the number of zombies in the map falls below a certain value, the game ends. The positive feedback the player receives is the increase in points gained by eliminating the zombies.

The map is designed to fit over five thousand agents, and these agents will be able to interact with each other as well as with the map structure. Figure 1 is the map of this game. The size of the map is approximately 23300*20600. The map is quite large because we consider the enemy density as the most important factor. As we randomly spawn these agents across the whole map, keeping these agents from taking over the screen, and preventing them from colliding with each other when they spawn will take a huge amount of processing resources. We sized the map so that no more than 80 agents will be spotted by the player at the same time. And the game will maintain a steady frame rate at the agent spawning phase. This map consists of over 200 obstacles with collision enabled, and one navigation mesh that covers the whole map. There are also some structures with mechanics. Four zombie evacuation points (Figure 3) are put at the four corners of the map, where the zombies will despawn. We also set up 10 portals (Figure 2) on the map which provides player more mobility and enable players to move quickly around the map.

As the player cannot see the map in its entirety, the gameplay loop is focused on the exploration of the map. There are a lot of things that players can discover and master during the play-through. For example, the mechanics behind the portal, the location of the evacuation point and the pattern of zombies travelling to different evacuation points. Players will be able to gain a deeper understanding of the map mechanics by constantly trying, and be able to get a higher score on their next attempt.

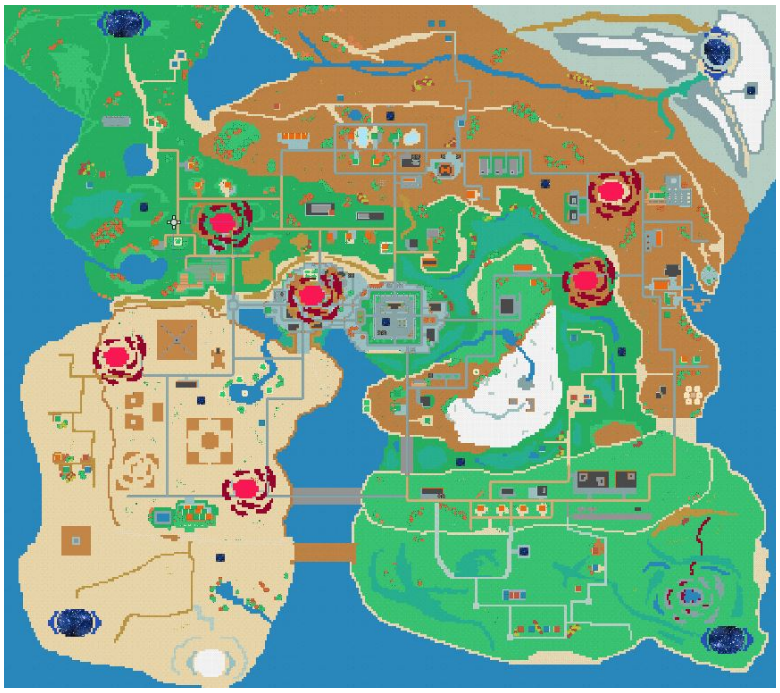

Figure 1: World map

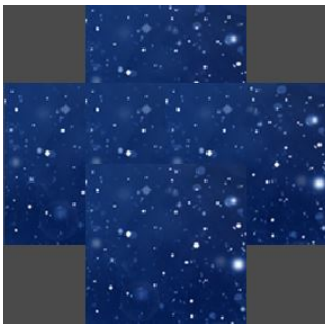

Figure 2: Transport area

Figure 3: Evacuation areas

4 Exploration

Navigating and moving large crowds with collisions at the same time is a huge consumption of CPU. Once we drop 1.2K agents in the game, the frame rate drops to below 15FPS. Thus, we made an assumption that frame drop was caused by the large number of nodes and heavy calculation for collisions (Figure 4). Therefore, we draw up two ways of solutions: reduce the number of nodes or reduce processing times. Here we split our solutions into two branches. This section is mainly about reducing the number of nodes. In section 5, we will discuss how we reduced processing times.

4.1 Reduce the number of nodes

Like the developer log of bittersweet birthday[3], we tried to register manually the agents to the navigation server (Figure 5). Since we used only one node to spawn zombies instead of instancing four nodes each, the process time was greatly shortened (Figure 6) with 6400 agents.

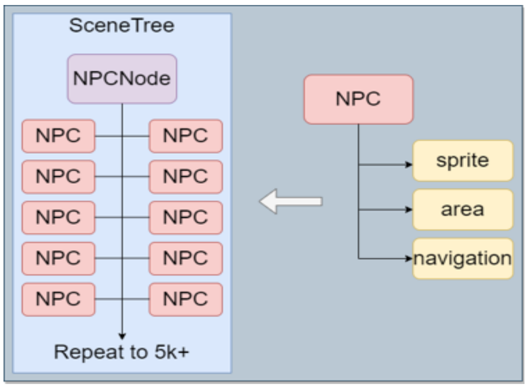

Figure 4: Scene Tree without bullet system

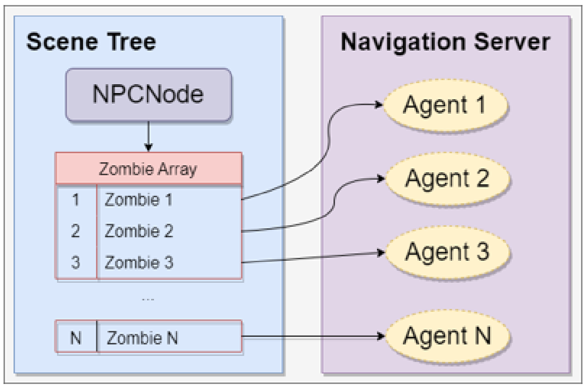

Figure 5: Scene Tree with bullet system

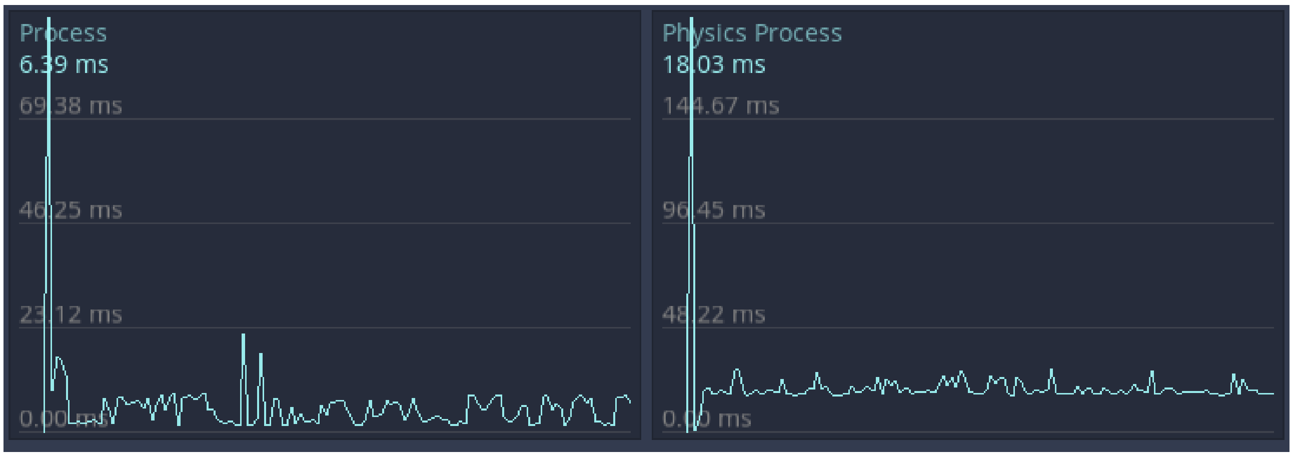

Figure 6: Process/Physics Process Time over Time

4.2 Navigation with RVO

Godot Engine uses RVO for collision avoidance, the navigation server will receive the properties of agents to confirm their conditions such as velocity and position. After the RVO process, their safe velocity will be returned to each of them using a callback. Godot 3.5 has introduced this feature, however, as it is an interim version, we cannot get the correct callback every time we need it, where the structure of the old *Navigation2D* node is to blame. Thus if we want to have the accurate velocity, we have to use an *Agent2D* node or refresh the server every frame, obviously, both of the solutions will add too much overhead.

4.3 Plan on hold

The bullet system can truly reduce the loading time and processing time, but it is not suitable for the following algorithm. Setting *Agent2D* as individual nodes is also a solution, however, it is doubtful whether it is worth restructuring the whole project for this purpose. Since trying this has taken too much time, we sadly set this conception aside.

5 Methods

Asynchronous NPC Generation Each NPC is an independent instance in our game, which means it not only needs to request a new memory space but also initializes both the animation and sound effects resources. When 5K NPCs allocate memory and register resources together, the bursting computational demands may cause a massive frame drop, or even crash the game. So instead of putting all agents in the scene simultaneously, we set a countdown animation at the beginning of the game, and use this waiting screen (Figure 7) to spawn about 500 NPCs at each frame, which results in lowering the lags.

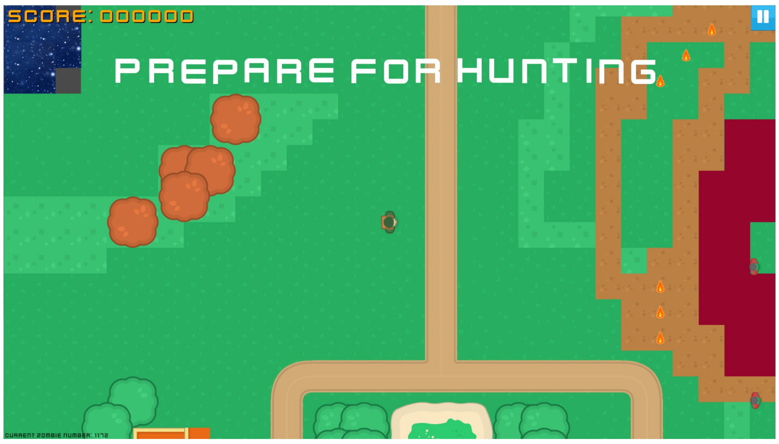

Figure 7: Waiting screen

Decreasing Physics Computation Rate In the final version of our demo, we further lower the physics frame rate from 60 times per second down to 10. This is because all physical interactions are dependent on the physics computation, including collision detection and bouncing. After making this change, we eliminated the heavy computation coursing by collision, and successfully maintain the frame rate at 40FPS.

Reducing Path Planning A-star algorithm is used by the game engine when calling the path to the destination, and this is computationally problematic because its time complexity increases exponentially as the path nodes grow, which means calculating the path for each NPC will bear a heavy consequence. So we reduced the number of path plannings as much as possible, by initializing it for each NPC when they are spawned and only replanning the path when one's collision counts above 50 times.

Reducing Collision As we all know, dealing with the collision between objects is power-demanding for the gaming engine since it needs to repeatedly calculate the bouncing reaction for each object. Thus, we improve the collision Logic, which means when the system detects collisions between NPCs, we make them stop. This is quite intuitive. Instead of bouncing around, the method can avoid further collision for the whole crowd. When an NPC stops, a timer will start ticking. The waiting time is related to the distance from the target, which is also straightforward. This feature results in letting NPCs line up to pass the crowded area (Figure 8) instead of blocking the narrow path.

Optimizing Evacuation Behavior We set the destination of an NPC as the evacuation area closest to it. Since we only have four evacuation areas on the map but many NPCs, a large group of NPCs will go in one direction. If the NPC at the beginning of a line is waiting, others behind it can not move in their direction, which causes congestion. We solved this problem by letting the NPC closest to the evacuation area move first if collisions happen. This ensures more reasonable behaviors.

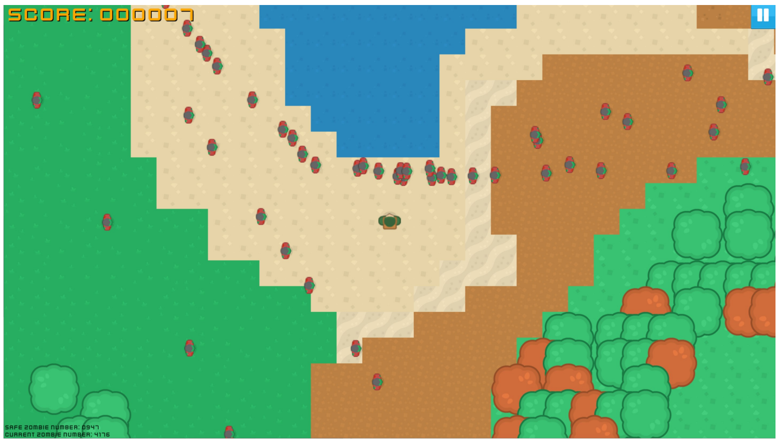

Figure 8: NPCs line up to pass

Other Optimizations We cleaned up the code, avoiding new or free variables during runtime to bypass the garbage collection. And we split the moving and collision control into two parts by putting the former part in actual frames and the latter part in physics frames. This is because the actual frame runs at 60FPS, and it can make the movement smoother.

6 Experiment and results

We have tested our game on a PC with a 64-bit Windows 10 Home system, an AMD Ryzen 7 5800U 8-core processor with Radeon Graphics 1.90 GHz, an NVIDIA GeForce RTX 3050 Laptop GPU, and 16 GB of RAM.

To evaluate the game's performance, we measured the change in frame rate over a 3-minute period from the beginning of the game. Results are displayed in Figure 9 and Table 1. There is a significant descending frame rate at the beginning of the game. This is due to the rapidly increasing NPC number. The frame rate dropped below 25FPS in the first 57 seconds. When the NPC number reached the highest, the frame rate hit the bottom which is 9FPS. At this moment, we observed a little lag on the screen. Other times our game can be perceived to run smoothly.

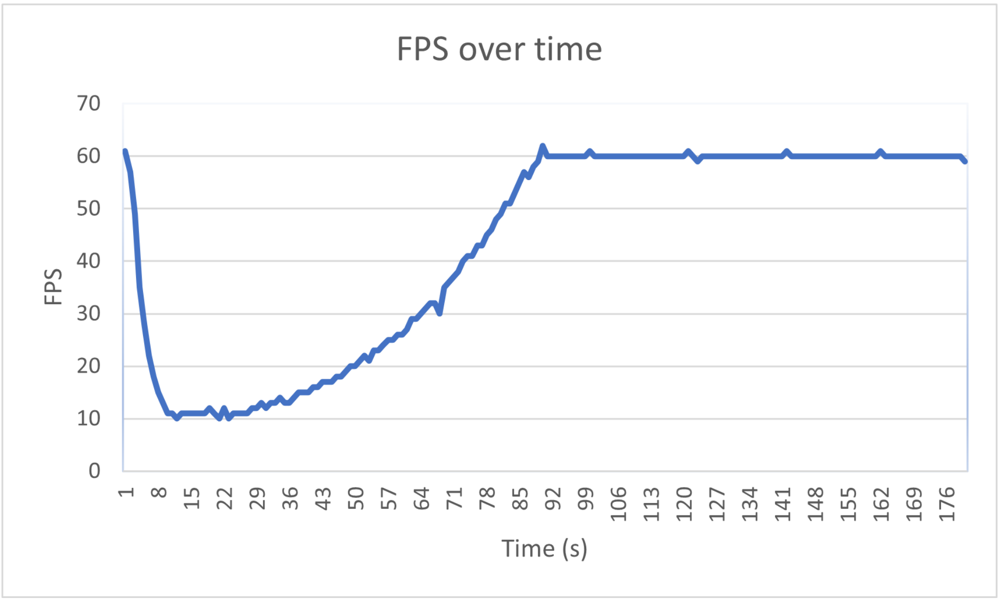

Figure 9: Frame rate change over time

Total frames	Time (s)	Min	Max	Average
7769	180	9	62	43.161

Table 1: Summary of frame rate

7 Discussion

7.1 Result discussion

Apart from the smoothness of the game, we observed some performances that could affect the game experience. The first one is the average time to finish one round. According to play test, player usually need about 10 minutes to reach the end game. When the current zombie number drops to 300, which takes around 4 minutes, it would be hard for the player to find the rest zombies. This is because the NPCs are stuck at some area inside the large map. Due to the time limitation, we did not optimize this problem. To help players find the right direction easier, we can put more transport areas on the map, and maybe create a minimap to show the position of the player and transport areas.

The other important feature is the performance of NPCs. The A-star algorithm is not the ideal method to guide the zombies. However, the RVO method limited the agent number to 1200, otherwise, we can use RVO to get more realistic movements. With the updated version of the Godot engine, we can try to make use of the bullet system for path planning, and even add animations on NPCs. According to the feedback, we could also use kNN queries in collision avoidance to prevent unnecessary computations. This is one of our future research directions.

7.2 Game engine assessment

Game Engine Comparison Godot is an open-source game engine that is free and accessible to indie game developers. It has a simple and user-friendly interface and a scripting language, GDScript, which is easy to learn and use. Additionally, Godot has an active and supportive community of developers and users who contribute to its development and share resources. However, it is less mature compared to Unity and Unreal Engine and supports fewer platforms. Unity and Unreal Engine have a more established ecosystem of tools and plugins. But they also have a steeper learning curve compared to Godot.

2D Game Development Unreal Engine is more focused on 3D game development and may require more effort to create 2D games. Unity may result in performance issues when used for 2D game development, especially for more complex and resource-intensive games. Compared to these two engines, Godot has a more dedicated 2D physics engine and supports features like sprite animation and tile maps, making it an efficient option for 2D game development. So it became popular among 2D game developers.

Coding For the coding part, Godot not only provides script language support but also exposes its APIs for C# and C++, however, its library files supporting C# and C++ are not up-to-date, making programming in C# unachievable. So instead of using other high-performance languages to deal with the gaming logic, we can only bear its built-in slow script language. In addition, although the engine contains rich APIs and functions, some of its critical features (like navigation and polygon mask) are marked as deprecated and are with defects. This results in devoting lots of time to debugging and replacing the existing codes.

8 Conclusion

In this report, the game "Purge" was presented, which simulates 5000 zombies simultaneously in the game scene. The game was developed using the Godot engine and the problem of handling a large number of NPCs was addressed. Various methods were implemented to resolve the frame rate drop problem. A navigation mesh was generated to let NPCs find the shortest path to their destination. Congestion was handled by prioritizing the movement of NPCs at the front of the queue. The game was tested and the performance was evaluated by measuring the change in frame rate. The results show a good performance of the game. We also mentioned the limitations of our game and how we can improve it in the future. Hope that we would have the chance to update our work and introduce the game to the public.

9 Acknowledgement

We would like to thank Prof. Roland for his lectures and for helping with our project.

References

[1] Godot. `https://godotengine.org/`.

[2] Unreal engine. `https://www.unrealengine.com/`.

[3] World Eater Games. Howto: Drawing a metric ton of bullets in godot. `https://worldeater-dev.itch.io/bittersweet-birthday/devlog/210789/howto-drawing-a-metric-ton-of-bullets-in-godot`.

[4] César García-García, José Luis Fernández-Robles, Victor Larios-Rosillo, and Hervé Luga. Alfil: A crowd simulation serious game for massive evacuation training and awareness. *International Journal of Game-Based Learning (IJGBL)*, 2(3):71–86, 2012.

[5] Ubisoft. Assassin's creed: Unity. `https://en.wikipedia.org/wiki/Assassin%27s_Creed_Unity`, 2014.