Crowd Simulation Project Report: Zombie Horde Shooter

Group 1B

Gianmarco Picarella
g.picarella@students.uu.nl
Utrecht University
Utrecht, Netherlands

Joel Brieger
j.brieger@students.uu.nl
Utrecht University
Utrecht, Netherlands

Nikos Giakoumoglou
n.giakoumoglou@students.uu.nl
Utrecht University
Utrecht, Netherlands

1 INTRODUCTION

In the project "Zombie Horde Shooter", we simulated a zombie horde with up to 20+ thousand autonomous agents trying to conquer the main city square. The player has the goal to protect the square from the zombies by means of an automatic rocket launcher. Zombies are spawned within a torus-shaped area centered around the square center and move towards it with a constant speed using a navigation mesh and avoidance system to avoid environmental obstacles and the other zombies. The player can jump from one building roof to another in order to shoot the zombies. If the number of zombies within the square is above a specific threshold then the game is lost. If the number of zombies is zero and the previous condition has not yet occurred then the game is won. The final result was a short and interactive game created with Unreal Engine 5 achieving satisfactory results in the amount of agents simulated while also preserving a playable frame-rate.

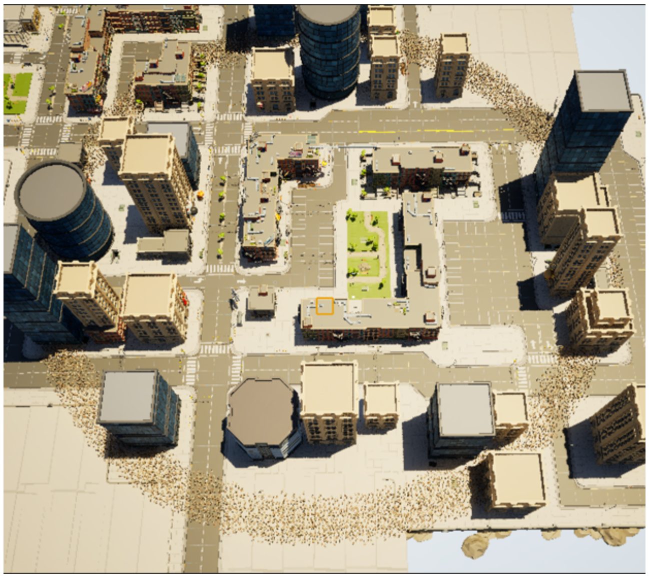


Figure 1: A top-down view of the entity spawn radius

Figure 2: Player point-of-view

2 RELATED WORK

The authors of the paper "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration" [4] introduced a system for automatic algorithm selection and configuration, enabling the determination of the most suitable nearest-neighbor algorithm for a given dataset. Based on our needs, we used a customized version of FLANN called nanoFLANN [1], which is currently one of the fastest libraries for performing fast approximate nearest-neighbour searches with kD-trees. A big element of our project was the usage of UMass [2] in Unreal Engine 5. This powerful framework is uses an Entity-Component-System (ECS) architecture to efficiently simulate thousands of agents in real-time. Fragments contain data, Processors contain the logic and entities are represented with IDs.

3 DEVELOPMENT ENVIRONMENT

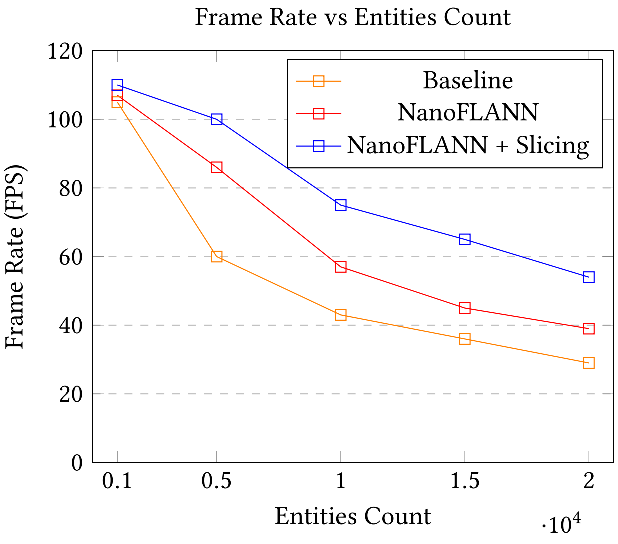
In this project we used Unreal Engine 5.3, its experimental UMass plugin to efficiently simulate crowds, and its experimental vertex animation plugin to handle zombie animations. Our codebase is compiled and edited using Visual Studio 2022 and C++ and it can be found at the following GitHub repository.

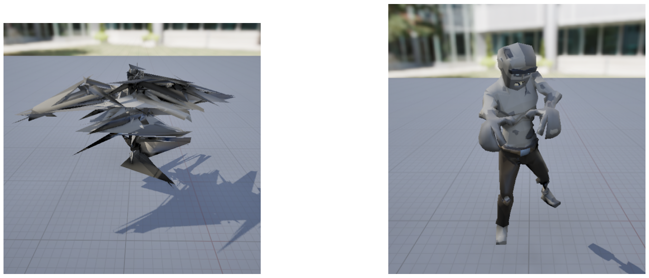
4 METHOD

In this section, we detail the implementation of the project and discuss challenges we faced as well as the methods we used to overcome them.

4.1 Efficient Avoidance

Initially, we used the built-in UMass avoidance system to handle entity avoidance. Unfortunately, such a system's performance drastically decreases as the number of simulated agents increases: with 15 thousand agents the measured frame rate is ≈ 10 FPS. Thanks to a CPU profiling session with Unreal Insights tools, we determined that the main application bottleneck was the nearest neighbours search performed for each agent. For this reason, we created our custom avoidance system based on NanoFLANN. As a result, we achieved ≈ 1x more FPS for the same amount of agents. In addition to that, we implemented a slicing strategy that distributes the nearest neighbours computation across multiple frames. Based on the amount of frame used, we trade precision for performance. The result of such computation is an approximate list of neighbours for each agent. In our tests, we sliced the computation across 3 subsequent frames. Our benchmarks show that slicing increased the frame rate by around 10-15 FPS. We run our benchmarks on a desktop computer equipped with a Ryzen 7600X, 32GB of ram and an RTX 3080. We measured the game in shipping mode with three different avoidance strategies: baseline (built-in UMass), custom avoidance and custom avoidance with slicing. The results show that nanoFLANN paired with the slicing technique is 3x faster than the baseline avoidance offered in UMass.





(a) Failed vertex animation

(b) Final vertex animation

4.2 Vertex Animations

As mentioned above Unreal Engine features an experimental vertex animation plugin [3] that automatically bakes animations into materials. It does this by first taking in a skeletal mesh with a source animation, a target output material that contains a "material attribute layer", and a target static mesh that features the original skeletal mesh in a neutral pose. With these objects in place, each frame of the animation is encoded into the vertices of the material. When done correctly, the result is an extremely performant animated mesh that is orders of magnitude less computational demanding that a regular animated skeletal mesh. Unfortunately, the process is incredibly precise and there were several failed attempts before success was achieved. Firstly, the level of detail in the static mesh and the original skeletal mesh from which its derived must be the same. Unfortunately, while the level of detail of the original skeletal mesh was low, it was still higher than required for our use case. As such a lower level of detail was created and was set as the only available level. Another issue was ensuring the material used was suitable for the process and a small amount of material customization had to occur in order to prepare it. This involved adding a material attribute layer that included the vertex animation information and blending the angle-corrected normal's with the original material texture. Ultimately the vertex animation process was successful which allowed for an enormous performance boost.

4.3 Instanced Draw Calls

Once the vertex-animated static mesh objects were in place, they were then fed into unreal engines UMass entity objects. From there it was automatically instanced once spawned. This allowed for the batching of draw calls which is a significant performance improvement when rendering multiple objects that feature the same mesh and material. This is because every draw call represents a communication between the CPU and the GPU. Without instancing each mesh and material attributes one draw call each. When rendering 30 thousand entities, that would be 60 thousand draw calls. Instancing these meshes allows for the draw calls to be batched, which reduces the number of draw calls to less than 5 thousand. Without this the game's performance would have been completely unacceptable.

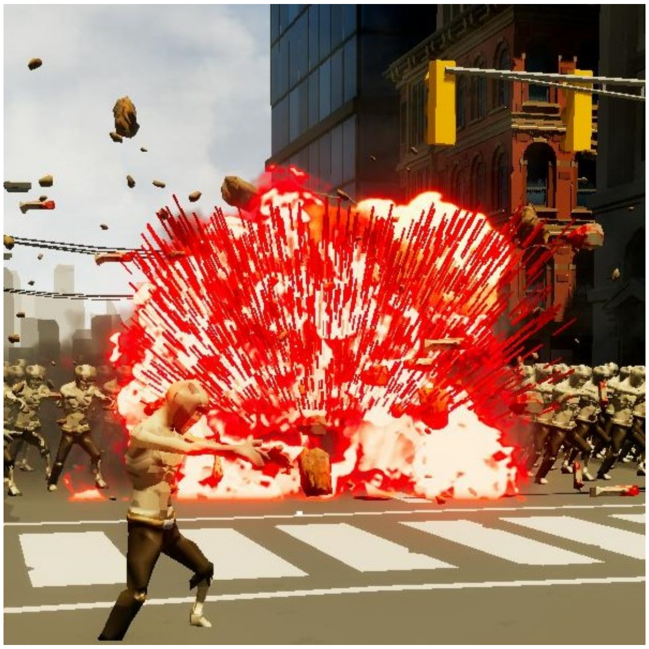


Figure 4: Visual effects used for explosions

4.4 NavMesh and Avoidance Integration

While the vertex-animated static meshes are the visual representation, algorithmically speaking, the agents are actually simple objects with a position, radius, and behaviour. The two primary behaviour functions that were implemented including world navigation on a NavMesh, and agent-to-agent avoidance using NanoFLANN. For the world navigation, each agent is provided a random target destination that is inside the central square. In the first frame, a path is calculated on this NavMesh from the agent to the target location. The agent is then moved along a path by updating is position and rotation transform. Note that since the agents do not feature a collision component of any kind, and do not simulate physics, the movement is purely transformational. That is to say that there is no physics force pushing the agent in the direction of its target. Instead its location is changed every frame to the next point on the NavMesh. Once the NavMesh calculations are complete, the updated location of the agent is read by the nanoFLANN-based avoidance system. This system first computes a kD-tree using the up-to-date location of each agent and then computes the list of nearest neighbours within a fixed radius from each agent. This information is then used by a force-based algorithm that minimizes the amount of collisions between different agents. The NavMesh and Avoidance systems follow a custom execution order that guarantees the navigation mesh system to be executed before the avoidance system. In the end, we managed to have a navigation system that can guide agents through static and dynamic obstacles.

4.5 Player Interaction and Gameplay Mechanics

In order to move from a simple observable crowd simulation to an interactive video game complete with a starting state, win state, and loss state, several key features were required. This includes a player controller, a health system, and a weapon that can be used to interact with the zombies. For the player controller, the built-in unreal engine first-person character was used. This saved valuable time as the built-in implementation and was perfectly adequate for our needs. The player character used default values for everything except the jump which was boosted enormously to allow the player to see more zombies from above. It also had the added benefit of making the game feel casual and less difficult which became crucial once the implementation of the health system was complete. The game's health system was rudimentary as this project was primarily used to display the effective crowd simulation solutions used. The system involved a basic check to count the number of zombies in a certain radius of the world origin which also happened to be the square the player is tasked with defending. Starting with 100 points of health, for every second that more than 500 agents are detected inside this radius one point is deducted. If the health reaches 0 it can be considered a loss state. If the player manages to kill all of the zombies whilst maintaining health above 0 it can be considered a win state. In order to determine the number of agents inside the square, our nanoFLANN nearest neighbour algorithm was generalized and a C++ function that accepted a radius and a location was written. This function efficiently calculated the number of agents that were inside the given radius at the given location. Originally, it was written to be used for the game's rocket launcher and was later re-used for the health system. The rocket launcher itself was a custom implementation although based on the example gun from the first person template provided by Unreal Engine. Mechanically it is very simple, upon firing the rocket launcher a slow-moving projectile is spawned and moved forward at a constant rate until it hits something physical such as a building or the ground. Note that as mentioned in section 4.4, the zombies are not considered physical and therefore cannot be directly hit by the rockets. For this reason, using a physics-based collision detection radius was also not available and a custom solution was required. Fortunately, as described above, our NanoFLANN implementation is generalizable. By passing the hit location of the rocket from its physics representation in blueprints to the C++ function, we were able to leverage nanoFLANNs performant nearest neighbour search to return all agents that were within a specified radius of the rocket explosion. These agents were then instantly destroyed and the number of these agents was passed back to unreal engine blueprints for further visual effects.

4.6 Visual Effects

Pre-made Niagara particle effect systems were used for the rocket and its explosions. For the rocket projectile, a firey trail was added that allows the player to clearly see their rocket as it moves through the air. Once the rocket hits the ground, an explosion effect is played. If it is determined that the rocket also killed at least one zombie, a basic blood and guts effect is played. Initial tests were used to scale the amount of blood and guts based on the number of zombies killed but this was dropped due to reliability issues in playing to many particle effects.

4.7 UMass and Unreal Engine Complexity

Most group members did not have any prior experience with Unreal Engine. This lack of knowledge slowed down the development process. Because of its experimental status, UMass has changed its interface many times in the last few years, making its API usage confusing at times. Intercommunication between the two is also a hard task: it requires the creation of specific subsystems that can then be accessed through Unreal blueprints. The data in such subsystems cannot be accessed concurrently from Unreal and UMass as data integrity cannot be guaranteed. Consequently, we had to implement a mutually exclusive subsystem that is run on the game thread and can be accessed by blueprints and UMass code without having to worry about concurrency.

4.8 Lack of Documentation

Both Unreal Engine and UMass lack of up-to-date documentation going through more technical aspects of the code. Because UMass is still in its experimental stage, the available documentation is mostly outdated or invalid. This is because UMass API has changed many times in the last few years. This also implies that the available online repositories cannot be used for experiments or study purposes because the used API is deprecated. In the end, our only choice was to study the internal UMass codebase and experiment all the library features by ourselves. We believe this process has been the most challenging one.

5 FUTURE WORK AND LIMITATIONS

We believe that there various performance and visual improvements that could be done to the project. In terms of performance, we are currently bounded by the UE rendering system. Decoupling the simulation and visualization code would allow to use different refresh rates thus improving the amount of entities our application can handle. We have tried to prevent characters from penetrating walls when the crowd density is high and the available space small. Our current solution to this problem is two-fold: first, we apply a bigger repulsive force for collisions between agents and walls; second, we keep the size of each narrow passage sufficiently large to avoid the issue. Despite some visual improvements, we haven't come up with a reliable solution to this problem. We believe that a different force-based algorithm is required in order to achieve this result. In terms of gameplay experience, more complex zombie behaviours and animations could also be implemented such as attacking the buildings or climbing up to reach the player.

6 CONCLUSION

The final version of our project successfully simulates 5+ thousand agents in real-time with a peak of 15 thousand agents, a maximum framerate of 110 FPS and a minimum of 35 FPS. We managed to reduce much of the computational costs by performing a repeated profile-optimize process using the tools provided by Unreal Insights. In conclusion, we managed to achieve our goal of simulating over 5 thousand agents while presenting challenging, yet fun, gameplay at playable framerates.

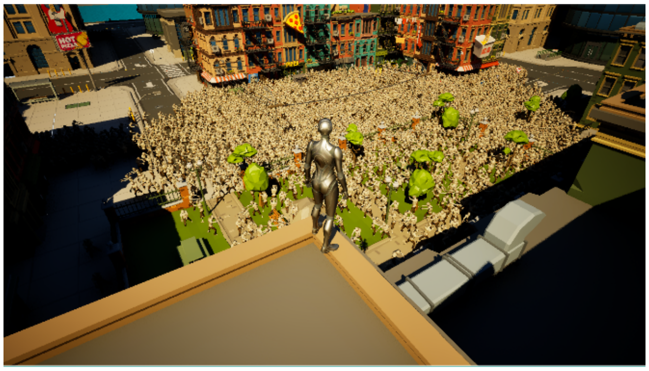


Figure 5: Group of entities in the central square

REFERENCES

[1] Jose Luis Blanco and Pranjal Kumar Rai. 2014. nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) with KD-trees. https://github.com/jlblancoc/nanoflann.

[2] Unreal Engine. [n. d.]. UMass. https://docs.unrealengine.com/5.0/en-US/overview-of-mass-entity-in-unreal-engine/

[3] Epic Games. [n. d.]. Unreal Vertex Animations. https://dev.epicgames.com/community/learning/tutorials/daE9/unreal-engine-baking-out-vertex-animation-in-editor-with-animtotexture

[4] Marius Muja and David Lowe. 2009. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. VISAPP 2009 - Proceedings of the 4th International Conference on Computer Vision Theory and Applications 1, 331–340.