CMPE485 TERM PROJECT REPORT (OPTION A)

Ahmed Bera Pay - 2020400183
Mehmet Süzer - 2019400213
Repository: https://github.com/mehmetSuzer/CMPE485_Term_Project
Video: **CmpE 485 Term Project - Battlefield Simulation - Spring24**

Technical Challenges

1- Rendering large number of game objects in the scene
2- Handling large number of rigidbodies and collisions
3- Instantiating and rendering many instances of particle effects such as blood

Why Did We Choose This Topic?

In the game industry, action-adventure is one of the most popular genres. Games of this genre often require a large number of objects and effects to be rendered in a single scene. In addition, most of the objects are subject to a collision when a projectile hits. Large numbers of soldiers and projectiles on a wide battlefield have a high impact on the GPU and CPU due to heavy computations. It is common to experience FPS drops when the camera is looking at a wide area full of objects. Therefore, in this project, we decided to focus on handling a high number of objects and effects in a battlefield. Our goal is to achieve a balance between visual fidelity and performance, ensuring a smooth and immersive gameplay experience. You can follow the links below to see that handling a large number of objects in games is common problem for games:

1- Optimizing performance of many rigidbodies - Unity Forum
2- Thousands of RigidBodies in Scene - performance improvement - Unity Forum
3- How many Rigidbodies can Unity support?
4- [VC] Battle Size vs Optimization :: Mount & Blade: Warband General Discussions

Our Experiment Setup

The following components are present in the battlefield:

1- A destructible castle built out of individual bricks
2- Cannons firing cannonballs periodically
3- Particle effects (smoke, explosion, fire) triggered by cannonball impacts
4- A high number of soldiers engaging in a combat
5- Particle effects (blood, magic) triggered by weapon hits

Following parameters can be adjusted:

1- Number of bricks used to built the castle
2- Number of cannons
3- Number of soldiers
4- Activation of particle effects
5- Lifetime of particle effects
6- Number of soldiers

In order to reduce GPU usage, CPU usage, and FPS drop, we used the following optimization techniques:

1- GPU instancing
2- Object Pooling
3- Level of Detail (LOD)

Controls

It is difficult to control all these parameters by hand. Therefore, we designed a menu containing a control panel where a user can adjust the settings for the experiment easily. Note that you should start the game from **MenuScene**. When you press **Run** button, it directs you to **CombatScene**.

In order to experiment in **CombatScene**, we added some controls:

1- You can control the camera by using **WASD or Arrow** keys.

2- You can move faster by holding **Left Shift**.

3- You can look around with your mouse while holding **Right** button.

4- You can shoot cannonballs from the camera position toward the camera direction by pressing **H**.

5- You can start and stop the cannon fire by pressing **V**. It takes a while for cannons to perform their first shot.

6- You can start spawning the soldiers by pressing **B**. Once you start spawning, it cannot be undone. You need to restart the simulation.

7- You can go back to **MenuScene** by pressing **ESC**.

8- You can start and stop our FPS counter by pressing **N**. When you stop it, it prints the average FPS between the start time and the stop time to the Debug Console.

Performance Evaluation

We will utilize the profiling tools within Unity and our FPS counter to measure the following performance metrics:

1- Total Memory Usage

2- Number of Batches (Draw calls)

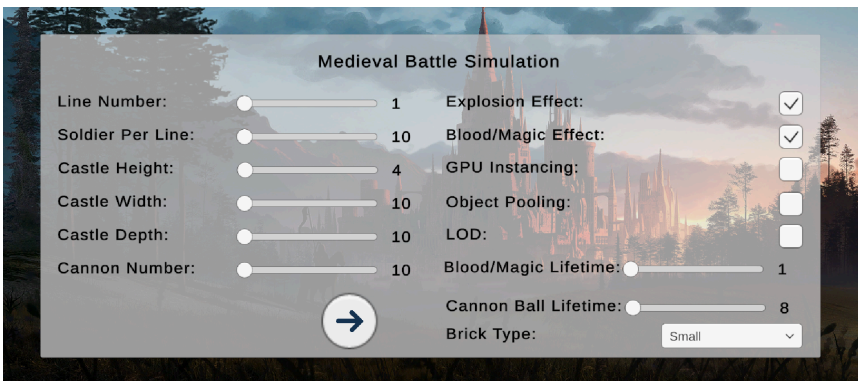
3- Number of Triangles

4- Average FPS

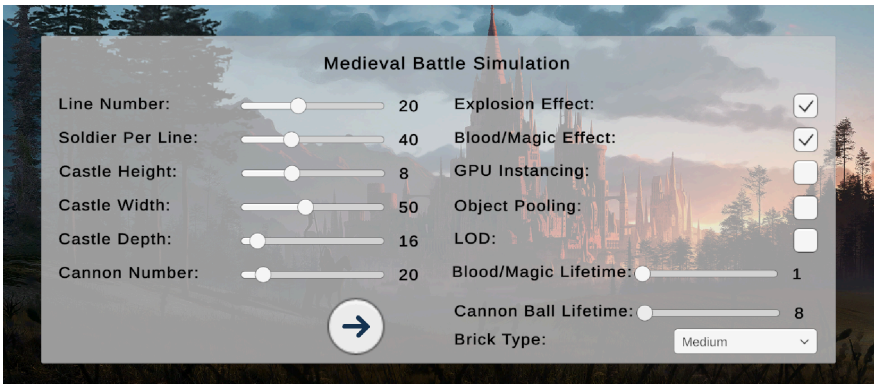
Experiments

All experiments were conducted in the same conditions. Once the construction of the castle was completed, t_start = 0, we started cannon fire(**V**), soldier spawn(**B**), and our FPS counter(**N**). The total memory usage, the number of batches, and the number of triangles were taken 15 seconds later, t_data = 15. FPS counter was stopped after 30 seconds later, t_end = 30. Therefore, average FPS was calculated for [t_start, t_end] interval.

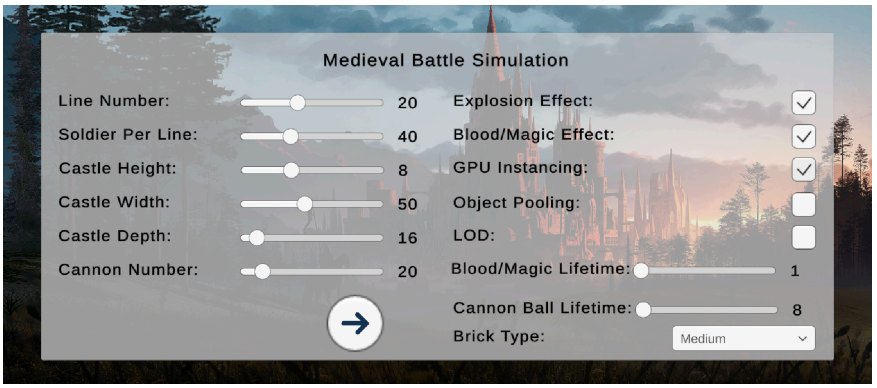
Experiment 1 Settings



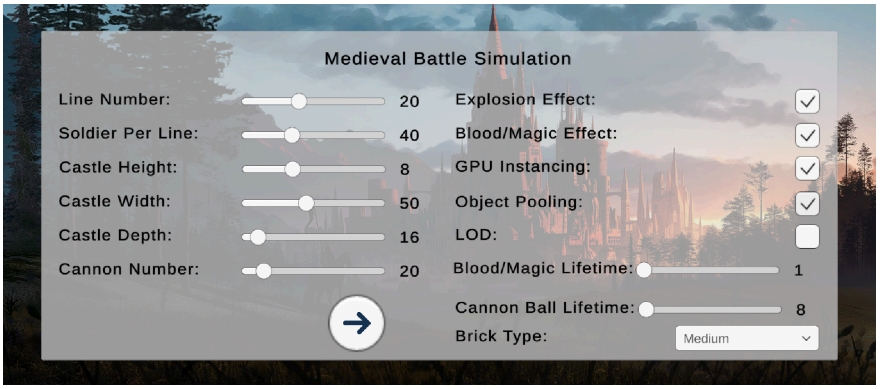
Experiment 2 Settings



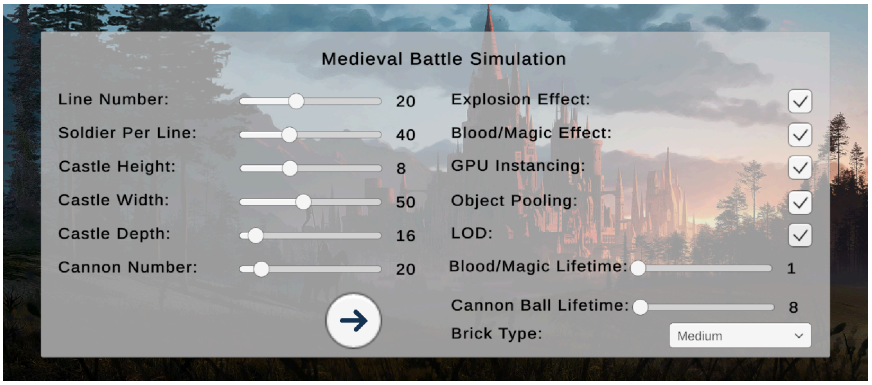
Experiment 3 Settings



Experiment 4 Settings



Experiment 5 Settings



Results

	Memory	# of Batches	# of Triangles	Avg. FPS
Exp 1	0.98 GB	953	374K	435
Exp 2	3.54 GB	11.75K	11.1M	17.92
Exp 3	3.59 GB	7.32K	11.1M	18.37
Exp 4	4.17 GB	6.95K	11.1M	18.54
Exp 5	1.71 GB	2.57K	3.4M	40.50

Conclusion

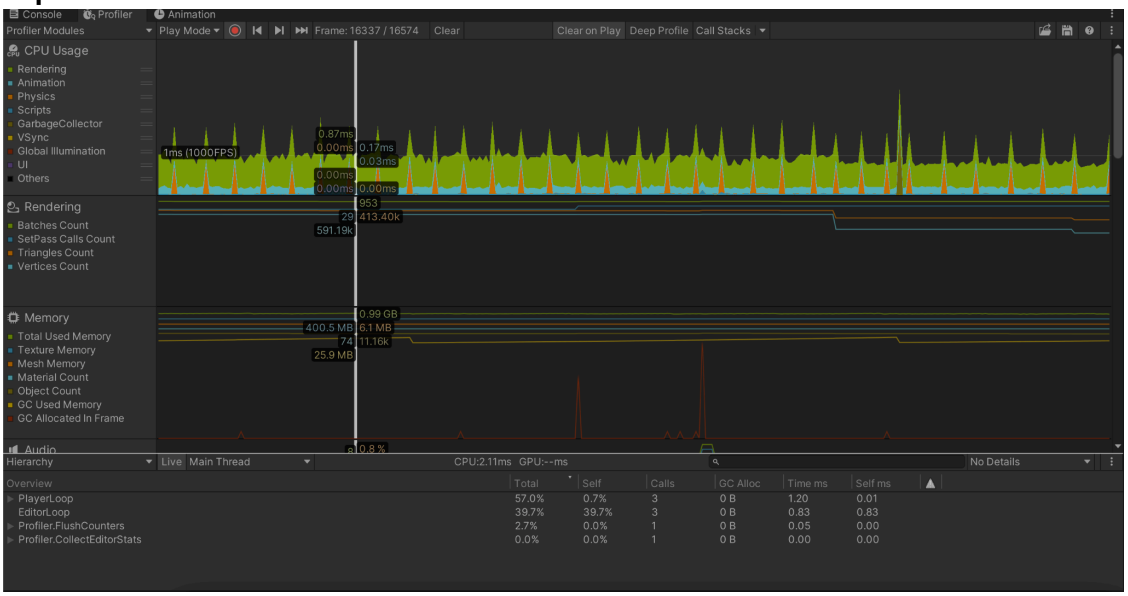
In the first experiment, the number of objects is low. Therefore, physics, animations, and render take small times, which results in high FPS and low memory usage. When the number of objects increases as in the second experiment, the memory usage increases due to the large number of triangles and animations. Average FPS, on the other hand, decreases dramatically. Frame transitions are clear to the user. In the third experiment, we enabled GPU instancing, which enables the game engine to render objects sharing the same mesh in the same draw call. The number of batches decreases by 37.7% from experiment 2 to 3. However, its effect on average FPS is not that significant. Average FPS increases by less than 1 FPS, which is about 2.5%. Since this is in the margin or error, we can conclude that GPU instancing has no major effect in our simulation. In the fourth experiment, we enabled object pooling in addition to GPU instancing. Since a large number of objects is instantiated before the simulation starts, the memory usage is higher than ever. However, its effect on the average FPS is small, as well. From experiment 3 and 4, we can conclude that rendering objects in different calls and managing memory have relatively small impact on the performance compared to physics and animations. In order to raise the average FPS, we need to reduce the cost of these. In the final experiment, we enabled LOD, too. LOD allows us to render objects that are far from the camera with less details. The results indicate significant performance improvements. The total memory usage, the number of batches, and the number of triangles are reduced by 59%, 63%, and 69.4%, respectively. We also see a huge boost on the average FPS. Therefore, we can deduce that LOD has a high impact on the game performance when the camera looks at a wide scene.

Extra

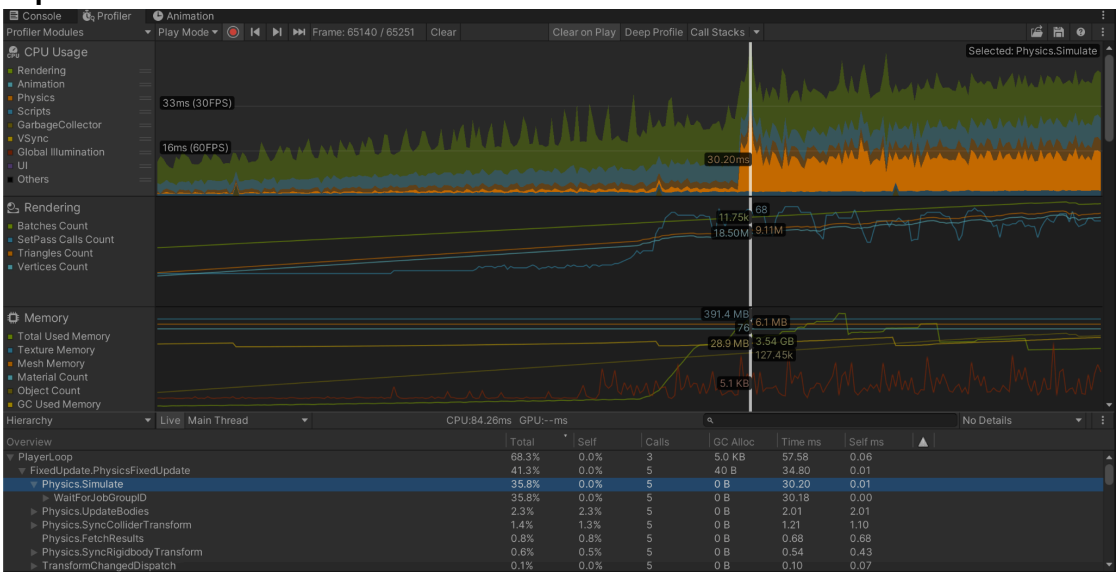
In addition to the optimization techniques listed above, we tried occlusion culling, too. Occlusion culling is a technique where objects that are not visible to the camera are not rendered by the GPU. The CPU detects triangles that are hidden by other triangles and prevents the GPU from rendering them. Occlusion culling is a useful technique for town-like scenes since many objects are hidden by buildings, whereas it reduces the performance when the scene is wide, and the game is not GPU-bounded. Our simulation contains a wide battlefield and requires high physics calculations, which means it is CPU-bounded. Therefore, we saw a less stable FPS in our simulation when we enabled occlusion culling. Since occlusion culling requires several steps like baking the scene, it hurts the repeatability of the simulation. Therefore, we didn’t add it to our menu.

Appendix

Experiment 1



Experiment 2



Experiment 3

Experiment 4

Experiment 5