Norwich University of the Arts

BSc Games Development

I610

How can a N-Body Simulation be created in Unreal Engine 4 and how can it be optimized through the use of Octree Partitioning Algorithms?

Technical Report

Daniel George Mark Dore

4,636 words

13th November 2020

Abstract

Game Engines are development environments designed to allow people to design and develop their own video games. Modern game engines have become more powerful as they have evolved, utilizing different API's and middleware to perform tasks and calculations required by the user. Often, they use physics engines to create realistic physics for their games. This report will investigate how a game engine can be used to fill the role of a simulation engine by creating a high cost, calculation heavy virtual simulation – in this case a N-Body Simulation. We then investigate what optimization methods are available to us to reduce the cost of running the simulation, such as parallel processing or other algorithms. Finally, we discover how the Barnes-Hut algorithm can be implemented by using a spatial partitioning algorithm to optimize the simulation and increase performance.

Contents

1 Introduction . . . 3
1 Introduction . . . 3
1.2 Unreal Engine 4 . . . 3
2 Literature Review . . . 3
2.1 Parallel Processing . . . 4
2.2 Octrees and the Barnes-Hut Algorithm . . . 4
3 Implementation of a N-Body Simulation . . . 5
3.1 Controllers and Bodies . . . 5
3.2 Interaction . . . 6
3.3 Body Setup . . . 6
3.4 Array Loop . . . 6
3.5 Creating Calculations . . . 7
4 Implementation of TOctree . . . 7
4.1 Elements and Sematics . . . 7
4.2 Adding Elements . . . 8
4.3 Generating the Octree . . . 8
5 Methodology . . . 8

6 Data Analysis 9
7 Conclusion 11
8 Further Improvements 11
9 Interactive Toy 11
10 References 12

List of Figures

1 A N-Body Simulation, where N equals 10,000. Source: devpack (2015) 3
2 Results from their simulation after implementation of loop unrolling Source: Nyland, Harris and Prins, (2009) 4
3 Example of an octree. Each node has either zero or eight children 5
4 Example of the Barnes-Hut Algorithm. The scene is recursively divided until only one particle is located in each bound. Source: David (2014) 5
5 The simulation's change in Frames Per Second over the 30 second period 10
6 The amount of RAM used by the simulation over the 30 second period 10
7 The percentage load of the CPU over the 30 second period 10
8 The percentage utilization of the CPU for the simulation over the 30 second period 11

1 Introduction

The aim of this technical report is to explore how the power of Unreal Engine 4 (UE4) can be used as a pseudo-simulation engine to simulate the N-body problem, what optimization techniques can be included to improve the simulations performance and how effective the techniques are. Two versions of the simulation will be created - one without any optimization techniques and one which includes already developed optimization methods - with the data gathered from each simulation compared with each other, to see how effective the optimization method is.

Finally, I will explore ways how a N-body simulation can be used as a game feature, or modified into an interactive toy. This paper will mainly focus on how the UE4 game engine can be used as a pseudo-simulation engine, but will have a brief experimentation into game features This section introduces the main elements that the report will cover - such as what a N-Body Simulation is and what Unreal Engine is.

1.1 N-Body Problem

As stated by Rubinsztej (2018), the N-body simulation is 'the problem of how to describe the motion of a number, n, different objects interacting with each other gravitationally'. The problem is mainly used to calculate the movements of celestial bodies, such as our solar system - our solar system would be a seven-body problem. Solutions have been found to the two-body problem and to the restricted three-body problem - where one mass is static and placed in the center - but answers to a problem with N amounts of bodies is difficult. The two, three and N-body problems can be calculated and simulated through the use of N-body simulations. N-Body Simulations have few uses - with their main use is, as stated above, to calculate the movements of celestial bodies in space. They can be used whenever a large amount of objects move around based on the other objects.

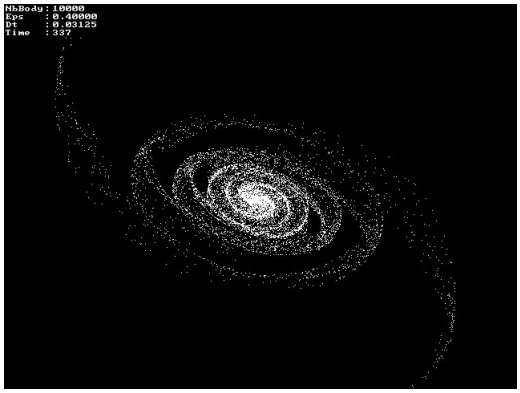

Figure 1. A N-Body Simulation, where N equals 10,000. Source: devpack (2015)

1.2 Unreal Engine 4

Unreal Engine 4 is a 3D game engine, developed by Epic Games (2019). Games created with UE4 can be exported to a wide range of platforms - including current and next generation consoles - in addition to virtual reality support.

Games created using UE4 can be created in Blueprint - a visual scripting system based on 'using a node-based interface to create gameplay elements from within Unreal Editor' (Epic Games) - or traditional scripting through C++ - with added UE4 macros. Software can be created fully in Blueprint, fully in C++ or a mix of both. The power of the engine does not differ between scripting methods - allowing complex games to be created regardless of skill.

Even though UE4 is mainly a game engine, due to the PhysX 3.3 (NVidia, 2013) physics engine for physics calculations, it can be used to create complex scientific calculations. PhysX allows for 'accurate collision detection as well as simulate physical interactions between objects in the world' (Epic Games), making it a good candidate for simulations.

2 Literature Review

This section features a review of literature pieces that are important to this technical report. This review will start with articles on parallel processing, then going into detail into other optimization methods, such as octrees.

2.1 Parallel Processing

Parallel Processing is the method of breaking up larger, more complex problems into smaller calculations which are processed by multiple microprocessors - reducing calculation time. Supercomputers use parallel processing to perform massive amounts of instructions and calculations in a small amount of time. Currently, parallel processing is very uncommon in low to mid engine devices, due to the limited amount of microprocessors per system. However, software can utilize the power of parallel processing by harnessing the devices graphics processing unit as a secondary microprocessor to process instructions.

2.1.1 Algorithms and Architecture

Parhami (2006) looks into the advantages and disadvantages that are introduced with parallel processing, the architecture of parallel processing and how it can be implemented to increase a software's performance. Parahmi details multiple variations of parallel processing architecture - including distributed-memory architecture. Additionally, they state that the main issue of parallel processing is 'the way the computational load is divided between multiple processors' (p. 8). Due to the large amount of processes that are created per second in a game engine, this could perhaps cause a game to slow down if the instructions are not correctly divided across the multiple processors.

2.1.2 Using CUDA for simulations

Nyland, Harris and Prins (2009) shows the advantages of using parallel programming by creating their own N-body simulation using NVidia's CUDA application processing interface. Their report states the calculations behind the simulation, how CUDA uses parallel processing to create a computational tile - where each interaction between two bodies 'are evaluated in sequential order' while separate rows are evaluated in parallel' (page 681).

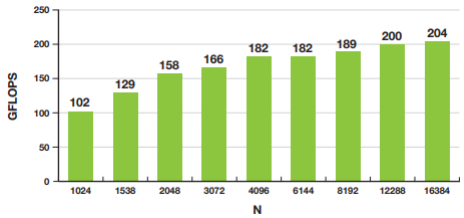

Figure 2. Results from their simulation after implementation of loop unrolling. Source: Nyland, Harris and Prins, (2009)

Their results conclude that implementing CUDA into their simulation 'achieved 163 gigaflops' with a simulation of '16,384 bodies' (p. 687), showing that parallel processing, when implemented correctly, can increase a software's performance massively. Additionally, through further optimization with loop unrolling, they were able to further improve the power of their simulation, calculating 204 gigaflops for 16,384 bodies (see Figure 2). Nyland, Harris and Prins' simulation could be improved today by recreating the simulation while using a modern graphics processing unit. Due to the difference in clock speed between the 8-series architecture and the 200-series architecture, the simulation could be pushed to simulate more bodies at once.

2.2 Octrees and the Barnes-Hut Algorithm

In programming, trees are specialized data structures which store their data in a hierarchically with parent and children nodes. Each node in a tree contains data, and may or may not have a child node. Quadtrees and octrees are versions of trees, where each node an

exact amount of children - four and eight respectively. Octrees are commonly used in 3D spatial representation, where a scene is divided into eight octants. Each octant is then given an node in the tree. The scene can then be further subdivided into eight smaller chunks, causing the divided node to have eight children.

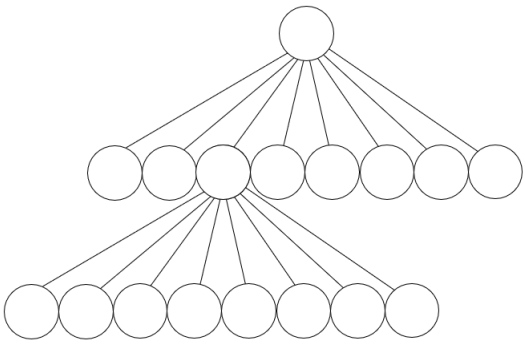

Figure 3. Example of an octree. Each node has either zero or eight children

Barnes and Hut (1986) shows how an octree can be used to optimize an N-body simulation by 'subdiving space into cubic cells, each which is recursively divided into eight subcells when more than one particle is found to occupy the same cell' (Barnes and Hut, 1986). In their experimentation, the scene of the N-body simulation was divided up through the use of an octree, where if more then one particle was contained in the same node at the same time it would be subdivided further until each particle was in a unique node.

In their investigation, Barnes and Hut describe that a standard n-body simulation processes 1/2 N (N - 1) calculations per timestep simulated, resulting in calculations rapidly increasing relative to N-1 increasing. From this, implemented an octree spatial partitioning algorithm to divide up simulation space into octants - with the division being recursive, repeating the division until only one body is located in each leaf node. The algorithm then groups particles together based on their distance from each other, with force calculations only created on each particle in each group. If a particle is outside the grouping, no calculation is generated. This allowed the calculations per timestep to be reduced to O(N log N).

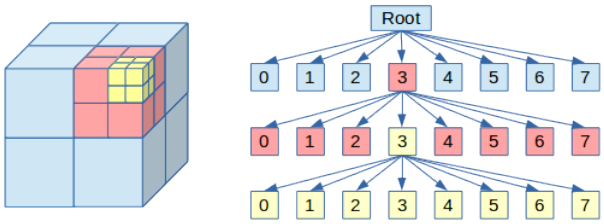

Figure 4. Example of the Barnes-Hut Algorithm. The scene is recursively divided until only one particle is located in each bound. Source: David (2014)

3 Implementation of a N-Body Simulation

This section features a overview on how we implemented the simulations, which were created in UE4.23 (Epic Games, 2017), combining a large amount of C++ scripting with small amounts of Blueprint where needed. The first, unoptimized simulation created features few to none optimization techniques, which will be expanded upon through the implementation of an octree spatial partitioning algorithm for the second, optimized simulation. Results will be recorded and compared to one another, to see how great of a benefit the optimization provides the simulation.

3.1 Controllers and Bodies

Using a blank C++ Unreal project, we began by implementing the two classes of objects that our simulation will use. One class is the objects that will represent the bodies which will interact gravitationally with each other in the simulation (CelestialBody), while the other class is the main simulation controller (SimController).

Unlike other N-Body Simulations we used an APawn class for our gravitational objects, which allowed for greater collision detection and interactiv-

ity while requiring slightly more processor load and random-access-memory when compared to particles or images. The properties of the CelestialBody objects are randomly set between specified parameters on object creation – these parameters can be changed inside the Unreal Editor to suit the user. These include mass and colour. They also include velocity and force vector properties, which are used along with the gravitational constant for the gravitational calculations, so are not randomly set on spawn.

The SimController class is used to allow the user to view and interact with the simulation, so we used another APawn class for our controller. Using a single object to control the whole simulation allowed the simulation to stay relatively simple, with any interactivity or calculations either triggered or processed by the controller. The SimController spawns the starting amount of CelestialBody objects onto the scene on creation, with the starting amount spawned equal to the set N number – which can be changed inside the editor. Additionally, all spawned bodies pushed to a TArray, allowing any body to be accessed at any time.

The octree will use the same spawning methods as the un-optimized simulation with some modifications, such as pushing to the octree instead of the TArray.

3.2 Interaction

We allow the user to manipulate the simulation through rotating around the center point of the simulation, pausing the simulation through a Heads-Up-Display button and displaying the simulation information direct to the user. The SimController features a CameraComponent attached to a SpringArmComponent, which allows for rotation on the pitch and yaw of the simulation along with modifying zoom levels between set bounds, allowing for simple navigation. A simple Heads-Up-Display displays the frames-per-second and N number of the simulation. Each body in the simulation contains a Boolean, which is set if the simulation is paused or not. The state of the Boolean is changed through a button in the Heads-Up-Display and depending on if the simulation is paused or not, force calculations will be processed.

```
// Pause or Un-pause the Simulation
void ASimController::PauseSimulation()
{
 if (bIsPaused == true)
 {
  bIsPaused = false;
  for (int i = 0; i < BodiesArray.Num(); i++)
  {
   BodiesArray[i]->bIsPaused = false;
  }
 }
 else
 {
  bIsPaused = true;
  for (int i = 0; i < BodiesArray.Num(); i++)
  {
   BodiesArray[i]->bIsPaused = true;
  }
 }
}
```

If the simulation was to be created into an interactive toy, we would add the ability to select a body in the simulation, attach it to the mouse cursor, push all body information to the Heads-Up-Display while still allowing for force calculations with each other body.

3.3 Body Setup

We set the static mesh and it's material instance of each CelestialBody object in the SimController Class and pass it to each body when it is spawned – as pointers can only be set in the scene of a spawned object. Additionally, when the material instance is applied it takes in the CelestialBody's colour variable, allowing for random colours between each body. We also set the static mesh of the body to be the root component for the octree implementation. This allows the spatial partitioning to get the center of the body more accurately.

3.4 Array Loop

In order to calculate the force between two unique CelestialBody objects, we create a function that is called on every timestep of the simulation. This function loops through every index of the TArray that contains references to every CelestialBody that has been

spawned in the simulation. We then loop through each index of the TArray again, checking to see if the two indexes are equal or unique.

```
for (int i = 0; i < BodiesArray.Num(); i++)
{
 for (int j = 0; j < BodiesArray.Num(); j++)
 {
  if (BodiesArray[i] != BodiesArray[j])
  {
   BodiesArray[i]->AddForceToBodies(BodiesArray[j]);
  };
 };
};
```

Instead of calculating every force calculation in the SimController class, we pass through reference to the second body to the primary body and calculate the forces between in the CelestialBody class – eliminating the need to get two lots of properties.

Calculating in this way generates $(N-1)^2$ force calculations – for example if N was 1000, each timestep we would generate 998,001 calculations. We then optimized this slightly through removing duplicate pairs and inversing forces through changing how we loop through the array, changing the amount of calculations per timestep to $\frac{1}{2}N(N-1)$ – this reduces force calculations by over half.

```
for (int i = 0; i < BodiesArray.Num(); i++)
{
 for (int j = i; j < BodiesArray.Num(); j++)
 {
  if (BodiesArray[i] != BodiesArray[j])
  {
   BodiesArray[i]->AddForceToBodies(BodiesArray[j]);
  };
 };
};
```

The amount of force calculations generated would be substantially reduced with the implementation of the octree - these loops through the array would be removed in its implementation.

3.5 Creating Calculations

As we now have reference for two unique bodies, we create a function that calculates the gravitational force between the two referenced bodies, updates the body's force property and modify the body's velocity based on the new force amount. The function begins by calculating the distance between each body on each axis, followed by the Euclidian distance between each body. Next, we calculate the forces applying on each body by using Newton's law of universal gravitation, given by –

$$F = \frac{G * M_a * M_b}{D^2} \tag{1}$$

Where G is the Gravitational Constant, M is the mass of each body and D is the distance. The function then calculates the total forces applying onto the body in each axis direction, given by –

$$F_a = \frac{NF * AD_a}{D} \tag{2}$$

Where NF is the forces calculated above, a is the current axis, AD is the distance on the a axis and D is the distance between the two objects. The function then inverses the axis distances and repeats the calculation to find the inverse force due to the removal of duplicate pairs.

4 Implementation of TOctree

This section features a brief overview on how we implemented an octree into our simulation, using the TOctree class featured in the UE4, similarly to how StenBone implemented his in their SpacePartitonerUsingTOctree repository. However, due to the limited amount of documentation on how TOctrees are used in the Unreal Engine C++ API, we were able to implement an octree, but could not modify it to our specifications.

4.1 Elements and Sematics

We began the octree implementation by creating the two structs required by TOctree. These structs are required to initialize the tree on spawn, all they hold all the information the tree needs. FOctreeElement holds all the information about an element placed into the octree – such as the object class and the bounds of the supplied object - while FOctreeSematics holds all the data about the construction of the octree. We set the octree's max elements per leaf – or how many elements can be placed in a single node - to one and the max inclusive elements per node – or how many child nodes each internal node has. This gives our tree the information to recursively divide until each element in the tree is in a unique leaf node.

We then initialize the TOctree by adding it to the scene through the SimController. We create a small function that creates a new bounding box for the spawn range of the controller, spawns the TOctree class into the scene, and initialize it with the bounding box.

4.2 Adding Elements

Following this, we create a function to add elements to the octree, which we created out of two smaller functions. The first function creates a bounding box for a supplied CelestialBody component bounds, then calls the second function which makes a FOctreeElement struct containing the supplied body and bounding box. Finally, we add the new element to the octree, which is then used in octree generation. We call this function whenever a new CelestialBody is spawned by the SimController, allowing the octree to generate force calculations as soon as the body is spawned.

4.3 Generating the Octree

Now that elements can be added to the TOctree, we create a function that is called on every timestep of the simulation to build the TOctree – by putting each element added into a unique leaf node based on its position to the center of the octree. For every node of the octree, the function checks if the current node has children or not or currently has an element in it or not. Based on these results, it either puts the element in the empty node, moves down a node level or creates children for the node and place the new element and the node element into one of the new leaves.

Based on what state a Boolean is, the function then draws each of the nodes bounds through DrawDebugBox. These debug boxes allow the user to see the octree node bounds. We only draw the nodes that contain elements, to reduce the number of bounding boxes the function needs to draw per timestep. We then allow the tree to be re-built every timestep. However, as there is no function in the TOctree class, the function we created instead destroys the spawned octree and spawns a new one right after. Every CelestialBody object in the BodiesArray TArray is then passed in, allowing the octree to rebuild.

We could not finish the implementation due to the limited documentation and inheritance issues. We will use this partially implemented simulation to gather data from and compare to the un-optimized simulation to see how much of an affect the octree can have on performance.

5 Methodology

In this section, we explain an overview on what types of data we are gathering, the methods we used to record the data and why the data is important to test for. We gather and record data from both the un-optimized simulation and the octree implemented simulation, compare the results against each other and analyse them in greater detail in the Data Analysis section.

For each of the two simulations, we will record data for two different N amounts – an average amount of bodies and large, stress test amount. We will run the simulations for thirty seconds, record data every second and then generate a time-graph for each individual simulation, allowing us to see what changes

over time. We will also calculate the average for each data type, allowing to compare between each graph. These results will be quantitative, allowing us to get an accurate correlation on how the implementation of the octree modifies the simulations performance. Even though the octree implementation is partially incomplete, we will still gather data for both simulations to analyse what effect the octree could have when fully implemented.

We measured 4 different units to see what affect they have in the simulation – Frames Per Second, CPU Load, CPU Utilization and Random-Access Memory used. The following sections will provide a brief overview on what each unit of measurement is and why it is important to test for.

The Frames Per Second (FPS) of a program is 'the number of consecutive full-screen images that are displayed each second. ' The higher the frame rate of a simulation or video game, the greater the number of images created per second and, in turn, the smoother the video output' (techterms, 2015). Generally, any modern program which has an average of 60fps or higher is considered acceptable performance wise. We gathered the FPS of the simulation as it is directly tied to the FPS, meaning the lower the FPS is, the longer amount of time between timesteps. We used the in-engine FPS calculation (1 divided by the delta time) and display it to the SimController's widget to get an accurate result.

The Central Processing Unit (CPU) is the primary component of a computer system, where any instructions created by the system are processed. Modern CPU's contains 2 to as many as 32 cores, which allow CPU's to process multiple instructions at once. The Load of a CPU is the measure of how many tasks are waiting in the CPU's kernel to be processed. The higher a CPU's load, the more processes waiting to be processed. A CPU's utilization is the usage of a processor at a specific point of time, measured as a percentage. Basically, the higher a processors utilization, the more of the processor that is processing instructions. We gathered the clock speed and processor utilization as they can affect the simulations performance – the higher the clock speed the more force calculations can be processed, which will directly impact the utilization of the CPU. We used NZXT's CAM software to record the CPU load, while using Task Manager to record the overall utilization.

The Random-Access-Memory (RAM) is another primary component of a computer system, where short term data is stored until it is no longer in-use or needed. This includes programs currently in use, any data in use by these programs or data to be processed by the CPU. Modern computer systems usually have around 8 to 16 gigabytes of RAM but can have more. We gathered how much RAM the simulation uses as it can also affect the simulations performance. Each CelestialBody object spawned in the simulation uses a small amount of RAM, with more bodies increasing the amount of RAM needed by the simulation. We used NZXT's CAM software to record the amount of RAM used.

6 Data Analysis

In this section, we will analyse the data we gathered from all our testing. As stated in our methodology, we recorded data for both simulations twice, once with an average amount of 1000 bodies and once with a high amount of 2500 bodies. We recorded the data values every second for thirty seconds, which will allow us to generate a time-graph for each simulation over the run time. The system specifications that we ran these tests on follows:

Intel Core i7 8700k @ 4415Mhz

NVidia GeForce GTX 1080 @ 1809

32GB DDR4 RAM @ 1066Mhz

2TB HDD

We also calculated the mean of each unit of each simulation, which will allow us to see the overall value of each unit. We will then use these in the comparisons, which will the allow us to see if the simulation could improve by finishing the octree implementation.

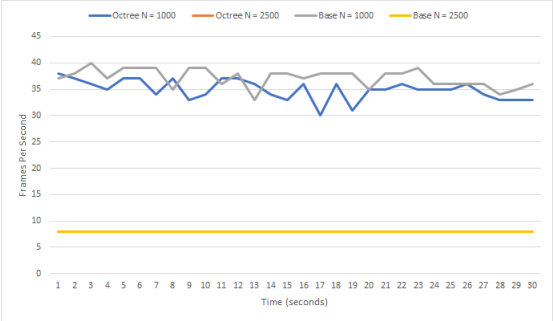

Figure 5. The simulation's change in Frames Per Second over the 30 second period

Our first figure shows the Frames Per Second of each variant of our simulation over 30 seconds. As shown in the figure on the simulations which had 1000 bodies, the FPS fluctuated over the testing period - with the highest point being around 40 FPS. Additionally, the simulation with the octree has slightly less FPS than the base simulation, with a drop around 3 frames on average, showing that the octree has a small effect on the simulation's performance. The figure also shows that both simulations with 2500 bodies had a steady FPS of only 8. We theorized that this is due to how many bodies are in the scene, increasing the number of objects that are rendered.

From these results, we gathered that before we could increase the amount of bodies spawned in a live simulation, we need to implement an optimization method to reduce the amount of bodies rendered each timestep. These optimizations include a culling method, where bodies outside for the players view frustum or behind other objects are not rendered.

Figure 6. The amount of RAM used by the simulation over the 30 second period

Our second figure shows the amount of RAM being used by the simulation over the same 30 second period. As shown in the figure, the octree implemented simulation requires more RAM to use – around 524 megabytes more. Additionally, the amount used slowly increased over the period, with a higher increase on the 1000 body simulation. On the other hand, the amount of RAM used by the base simulations reduces instead of increases. The results also show that there is only a small amount of difference in RAM use between a 1000 body simulation and a 2500 body one – requiring around 56 megabytes more on average.

From these results, we gathered that implementing an octree into our simulation increases our RAM used only slightly. However, how we have currently implemented it is flawed – we are destroying and spawning it on every timestep, which requires an increasing amount of RAM every time. This flaw could be remedied by re-building the octree on each timestep instead of spawning a new one, but currently TOctree does not include the functions to complete this.

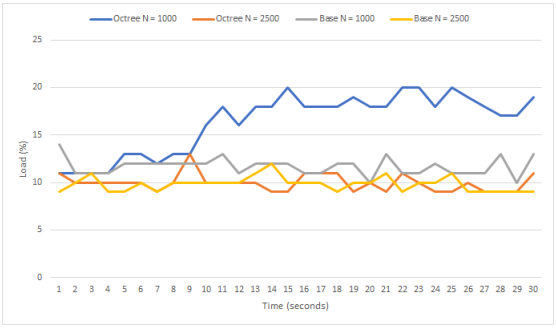

Figure 7. The percentage load of the CPU over the 30 second period

Our third figure shows the percentage load that the simulation has on the CPU over the same 30 seconds. The results show that the load percentage is consistent around each variant, with the exception being the octree 1000 body simulation – with an average load of 16% compared to 9%, 9% and 11% of the other variants – showing that more calculations were waiting to be processed in the kernel.

From these results, we gathered that the implemented octree increases the amount of load placed on

the CPU. Additionally, it shows that there is a direct coloration between an increase of FPS and CPU load – the higher the FPS, the shorter amount of time between timesteps and the more calculations created each second. This is complemented by the 2500 body simulations, where the reduced FPS creates less calculations per second and, in turn, less CPU load.

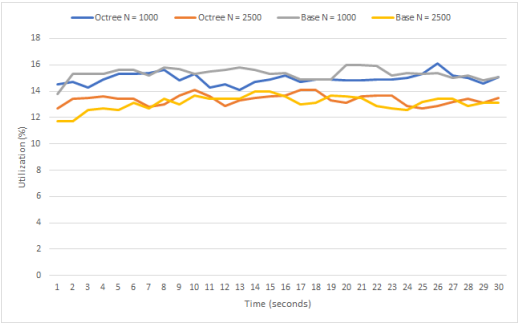

Figure 8. percentage utilization of the CPU for the simulation over the 30 second period

Our fourth figure shows the percent utilization of the CPU by the simulation over the same 30 seconds. The results show that the 1000 body simulations utilized slightly more of the available CPU than the 2500 body variants – around 2% more over their 2500 body variants. Additionally

7 Conclusion

In this section, we conclude our findings and evaluate what we have created for this report. We successfully answered one of our report's questions - which was to create a working N-Body Simulation in Unreal Engine 4 - but we were unsuccessful in completing our reports second question – to optimize the simulation through implementing a spatial partitioning octree.

We were successful in implementing a TOctree into our simulation but was unsuccessful in modifying it to our own needs. This is primarily due to the lack of documentation on the matter, but the limited amount of functions provided by the TOctree class is also a factor. We could bypass the use of a TOctree by creating our own octree generation, which would require further investigation into UE4.

From our testing results, we found that to create a high-object program in UE4 – such as a N-Body Simulation - we would need to implement a rendering optimization to reduce the amount of objects rendered per frame to increase the frame rate. We also found that the current way we have implemented the TOctree is flawed, as every time the object is deleted and respawned, it increases the amount of RAM used.

8 Further Improvements

In this section, we follow our conclusion by briefly discussing what areas we could improve upon on in the simulation. These include, but are not limited to:

TOctree Implementation – By fully implementing the TOctree, we could change how the force calculations are generated from $\frac{1}{2}N(N-1)$ to $O(NlogN)$ by grouping bodies together based on their position, calculated by the octree.

Octree Implementation – We could create our own octree generating class to replace the TOctree, which would give us the ability to re-build the tree without needing to destroy and respawn it on every timestep.

Implement game optimization methods – We could research and implement game optimization methods, such as culling, to improve the simulations performance.

9 Interactive Toy

In this section, we explore features we could add to a finished simulation to convert it into an interactive toy.

Clickable planets – Each body could have a struct which contains a wider range of randomized variables (such as temperature, name, climate etc). This struct could then be displayed on the widget attached to the SimController. These include, but are not limited to:

Drag-able Bodies – The user could click on any body and attach it to the mouse. The user could then drag it around the screen, while it still simulated its physics with the other bodies.

More interaction – The N value could be changed during gameplay, allowing the user to easily change the amount of bodies in the scene. This would be created through another widget, similarly to a pause menu.

10 References

Epic Games. (2019) Unreal Engine 4 [Game Engine]

Rubinsztejn, A. (2018) 'What is the N-Body Problem', Available at: https://gereshes.com/2018/05/07/what-is-the-n-body-problem/ (Accessed: 11 November 2020)

devpack. (2015) 'n-body-cosmos', Available at: https://github.com/devpack/nbody-cosmos (Accessed: 11 November 2020)

NVidia Corporation. (2013) PhysX 3.3 [Physics Engine]

Epic Games. 'Chaos Physics', Available at: https://docs.unrealengine.com/en-US/Engine/Physics/index.html (Accessed: 11 November 2020)

Parhami, B. (2006) *Introduction to Parallel Processing: Algorithms and Architecture*, Available at: https://books.google.co.uk/books?hl=en&lr=&id=iNQLBwAAQBAJ&oi=fnd&pg=PA3&ots=Xdj9aJU7fV&sig=wY4gLA-0ie1gxqzm6wr0wPC6Z94&redir_esc=y#v=onepage&q&f=false (Accessed: 11 November 2020)

Nyland, L., Harris, M. and Prins, J. (2007) *Fast N-Body Simulation with CUDA*, Available at: https://www.rath.us/m368k/hw5/gpu-gems-3--ch-31-N-body.pdf (Accessed: 11 November 2020)

Barnes, J. and Hut, P. (1986) *A heirarchical o(N log N) force calculation algorithm*, Available at: http://www-inf.telecom-sudparis.eu/COURS/CSC5001/new_site/Supports/Projet/NBody/barnes_86.pdf (Accessed: 11 November 2020)

David. (2014) 'Advanced Octrees 1: preliminaries, insertion strategies and maximum tree depth', Available at: https://geidav.wordpress.com/2014/07/18/advanced-octrees-1-preliminaries-insertion-strategies-and-max-tree (Accessed: 11 November 2020)

StenBone. (2015) 'UE-Space-Partioner-Using-TOctree', Available at: https://github.com/StenBone/UE_Space_Partioner_Using_TOctree (Accessed: 11 November 2020)

techterms. (2015) 'FPS', Available at : https://techterms.com/definition/fps (Accessed: 11 November 2020) NZXT. () CAM [Software]