Physic Engine for Spherical Particles

Studente/i	Relatore
Ravani Davide	**Schärfig Randolf**
	Correlatore
	-
	Committente
	Schärfig Randolf
Corso di laurea	Modulo
Ingegneria informatica	**C10662**
Anno	
2023	

Data

31 agosto 2023

Indice

Elenco delle figure

Abstract

In this thesis physically based particle simulators were analysed and based on the findings a solution was implemented that can handle huge amounts of sphere particle interactions in real-time harnessing the capabilities of modern GPUs. The program efficiently models dynamic interactions and behaviours among particles, focusing on realistic particle motions, forces, and collisions. By skilfully utilizing GPU resources, it becomes a potent tool for understanding and visualizing intricate particle behaviours within the accelerated realm of parallel computing. This program offers a comprehensive approach to exploring particle dynamics in diverse scenarios.

Capitolo 1

Introduction

The goal of this thesis is to implement a realtime particle physics simulation. Given that the calculations involved are computationally extremely expensive, the code was implemented to take advantage of the GPUs highly parallel nature.

The program that was developed to achive this goal makes use of a combination of GPU compute shaders 4.3.2 and buffer objects 4.3.1 to handle the behavior of the particles in the simulation. By exploiting the parallel nature of GPU computations as well as the optimized search for collisions, the simulation can efficiently handle a large number of particles.

The collision detection mechanism implemented in this project is a grid partitioning approach, boosting efficiency in identifying potential particle collisions. This strategy divides a part of the simulation space into a 3D grid, so particles in the same or neighboring cell can be checked for collisions. Applying this type of optimization is possible to minimize the number of collision checks required. Starting from a complexity of $O(n^2)$ this can be considerably optimized, as described in 4.2.

The results demonstrate that by simplifying the fundamental issue of finding the colliding spheres and implementing the collision detection along with the physical calculations on the high-performance GPU, real-time simulation of scenes containing 10-thousands of spheres is possible.

Capitolo 2

State of the art

Particle simulation on GPUs has revolutionized various fields, driven by the computational power and parallelism these processors offer. This chapter explores the state of the art techniques and advancements in particle simulation, with a focus on NVIDIA Flex, a cutting-edge particle-based simulation framework designed to harness the power of GPUs.

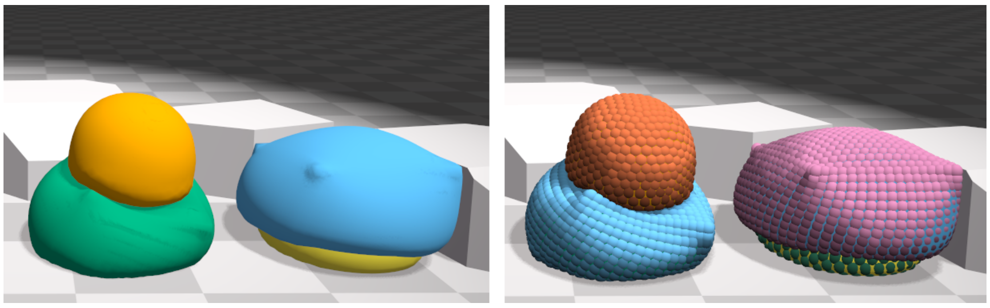
2.1 GPU Parallelism and Particle Simulation

Modern GPUs are designed for parallelism, containing thousands of cores optimized for performing the same operation on multiple data pointers simultaneously. This architectural feature aligns perfectly with particle simulations, which involve numerous particles interacting concurrently. The CUDA and OpenCL programming models provide efficient ways to leverage this parallel processing capability for simulations.

2.2 NVIDIA Flex: An Overview

NVIDIA Flex is a GPU-accelerated particle-based simulation custom library for real-time applications. It was developed to enable the creation of realistic and dynamic simulations of materials, fluids, soft bodies, and more. Flex provides a high-level API that simplifies the integration of particle-based effects into interactive applications, making it a valuable tool for simulations, games and visual effects.

One of NVIDIA Flex's standout features is its ability to produce real-time simulations with interactive responsiveness. It's particularly valuable in the gaming and entertainment industries, where dynamic and visually stunning effects are essential. For instance, in games, Flex can simulate complex fluid behaviors like splashing water, interactive smoke, and dynamic cloth interactions, all while maintaining playable frame rates.



Source: `https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/flex/manual.html`
Figura 2.1: NVIDIA Flex particles example

NVIDIA Flex isn't limited to the entertainment sector. Its capabilities extend to a variety of applications. Its adaptability highlights the versatility of particle-based simulations and the power of GPU acceleration.
Flex provides several solver modules for simulating different materials and behaviors, allowing developers to choose the best-fit solver for their specific simulation needs. Furthermore, Flex's architecture is designed to accommodate custom solvers, enabling developers to extend its functionality, for example providing springs simulation to connect particles together. This flexibility makes it a valuable asset for researchers and developers pushing the boundaries of particle simulation.

2.3 Challenges and Future Developments

While NVIDIA Flex significantly simplifies particle simulation on GPUs, challenges remain. Real-time simulation of complex scenes still requires careful optimization, and the balance between simulation fidelity and computational efficiency is an ongoing concern. As GPUs continue to evolve, with enhancements like tensor cores and dedicated ray tracing hardware, we can expect even more realism and accuracy in particle simulations.
NVIDIA Flex stands as a testament to the potential of combining GPU acceleration with particle-based simulation. The fact that it can provide real-time interactive simulations across multiple industries is proof of how powerful GPUs are and how developers are willing to push boundaries. As technology advances and GPUs become even more powerful, the future of particle simulations, bolstered by frameworks like Flex, promises to be truly remarkable.

Capitolo 3

Background

In this chapter we will discuss the physics related to the particle simulation. Starting with how we can work with discrete time steps introduced by the execution of the code in Section 3.4, we will continue describing the forces that are part of the system in Section 3.1. After that, a general description of the types of collision, elastic and inelastic will be discuss in Section 3.2. At the conclusion of the section, we will observe the collision of a sphere with planes and other spheres in Section 3.3.1 and Section 3.3.2.

3.1 Force system

In the intricate space of our universe, there exists a force that is both omnipresent the force of gravity. This phenomenon is the master conductor of cosmic orbitation and the choreography that governs interactions among particles on Earth. In this chapter as we unveil or refresh the intricate role of gravity.

3.1.1 Pull of Earth's Gravity

Gravity, as described by Isaac Newton's law of universal gravitation, is the force that draws objects towards one another based on their masses and the distance between them. On Earth's surface, the force of gravity is what keeps us grounded, gives weight to objects, and shapes the natural world around us.
The acceleration due to gravity on Earth, often denoted as G, is approximately $9.81\frac{m}{s^2}$. This value represents the rate at which objects accelerate towards the Earth when in free fall. In simpler terms, if you were to drop an object, its velocity would increase by $9.81\frac{m}{s}$ for every second it falls.

3.1.2 Force application

To comprehend the effects of gravity in particle simulation, it's crucial to understand the relationship between mass, acceleration, and force. Newton's second law of motion, $F =$

$m * a$, elucidates this connection. In the context of gravity on Earth, where acceleration is G, the formula simplifies to $F = m * G$.
For instance, if you were to simulate a particle with a mass of 2 kg, the gravitational force acting on it would be:

$$F = 2kg * 9.81\frac{m}{s^2} = 19.62N$$

This force of 19.62 N would be the weight of the particle, which is the force with which it's pulled towards the Earth.
In particle simulation, representing the effects of gravity involves incorporating this gravitational acceleration G into the equations that govern the motion of particles. For instance, when calculating how a particle's velocity and position change over time, the gravitational force can be included as an external force.
If the particle's mass is m and you're using a time step of Δt in your simulation, the update formulas for velocity and position could be:

$$\vec{v\prime} = \vec{v} + G * \Delta t$$

$$\vec{x\prime} = \vec{x} + \vec{v\prime} * \Delta t$$

These formulas account for how the particle's velocity increases due to gravity and how its position changes over time.
After some time, the velocity of a sphere could reach very low numbers, moving less than a millimiter per second. To reduce computational wasted time, with almost non existing movement, it is useful to introduce a threshold below which velocity is truncated to $\vec{0}$.

3.2 Types of collision

Collisions, in the realm of physics, are fundamental interactions where two or more bodies come into contact and exchange momentum and energy. These collisions can be broadly classified into two categories: elastic and inelastic collisions. These terms describe how objects react when they collide and forces, energy and momentum are conserved.

3.2.1 Elastic Collisions

In an elastic collision, both momentum and kinetic energy are conserved. This means that before and after the collision, the total momentum and total kinetic energy of the system remain unchanged. During an elastic collision, objects bounce off each other without any loss of energy. Billiard balls are a classic example of elastic collisions. When two billiard balls collide, they transfer momentum and energy between them, but the total energy of the system remains constant.

$$v_1\prime = v_1 - \frac{2m_2}{m_1 + m_2} \frac{(v_1 - v_2) \cdot (x_1 - x_2)}{\|x_1 - x_2\|^2} (x_1 - x_2)$$

Imagine a billiard ball hitting another one. Upon impact, they bounce off each other and continue moving without any noticeable loss of speed. In this case, the collision is elastic because both momentum and kinetic energy are conserved.

3.2.2 Inelastic Collisions

In an inelastic collision, momentum is conserved, but kinetic energy is not. When two objects collide inelastically, they stick together or deform upon impact, resulting in a loss of kinetic energy. This often occurs when objects become entangled or stick together due to the forces involved in the collision.

$$J_n = \frac{m_1 m_2}{m_1 + m_2}(1 + C_R)(v_2 - v_1) \cdot n$$

$$\Delta v_1\prime = \frac{J_n}{m_1} n$$

A classic example of inelastic collision is a car accident, where all the energy is dissipated in various forms.

3.2.3 Real-world Limitations

In the real world, it's practically impossible to have a perfectly elastic or perfectly inelastic collision. This is due to several factors:

Friction Even smooth surfaces have microscopic imperfections that cause friction. This friction dissipates some energy as heat during collisions, making them partially inelastic.

Deformation Even in elastic collisions, there's a tiny amount of deformation due to the compression of surfaces at the point of contact. This deformation results in some loss of kinetic energy.

Energy Dissipation Collisions often involve the conversion of some kinetic energy into sound, vibrations, and other forms of energy that cannot be fully accounted for.

External Forces External forces such as air resistance can influence the behavior of colliding objects, making it difficult to isolate the collision dynamics.

3.2.4 Coefficient of restitution

Another fundamental factor that must be included in the collision equation is the coefficient of restitution (COR). This allows to calculate the energy that is loss during the collision, resulting in a lower speed after the collision. In the case of a perfectly elastic collision this factor is equal to 1.0, in the inelastic case instead, it is 0.0. The great majority of collisions in the real world have a COR between these two values.

In summary, elastic and inelastic collisions provide a fundamental framework for understanding how objects interact during collisions. While the theoretical concepts of perfect elasticity and perfect inelasticity are useful for modeling, the real world's complexities, such as friction, deformation, and energy dissipation, prevent collisions from being entirely elastic or inelastic.

3.3 Collision detection and response

3.3.1 Plane collision

In the dynamic realm of particle simulation programming, the quest for realistic and accurate interactions leads us to see the reqiurements for a correct collision implementation. Among the fundamental collision scenarios, the simulation of sphere-plane collision emerges as a crucial feature. Allowing particles, represented as spheres, to interact with flat surfaces brings a sense of physical realism to virtual environments, making the spheres bounce off a surface. This chapter embarks on an in-depth exploration of the mathematics and mechanics that underlie the sphere-plane collision detection and response, diving into the complexities of its theory.

Collision Detection

The journey of sphere-plane collision begins with the process of collision detection, where the simulation determines whether a sphere and a plane intersect. This determination is facilitated by evaluating the distance between the sphere's center and the plane's surface, taking advantage of the plane's mathematical equation: $Ax + By + Cz + D = 0$. In this equation, (A, B, C) constitutes the plane's normal vector ($\vec{n}$), while D represents a constant term that offsets the plane.

Mathematically, the distance (d) between a point (x, y, z) and the plane can be calculated using the formula:

$$d = \frac{|Ax + By + Cz + D|}{\sqrt{A^2 + B^2 + C^2}}$$

The collision is detected if the calculated distance is equal or less than the radius of the sphere.

Collision Response

Upon the successful detection of a collision, the simulation advances to the collision response phase. This phase is in charge of determining the subsequent behavior of both the sphere following the collision. The underlying principle governing this process is the conservation of momentum and energy, which is pivotal in creating realistic interactions within the simulated environment. The plane is approximated to have an infinite mass, so any math is necessary since the surface should not move.
Consider the sphere's velocity vector (Vx, Vy, Vz). The collision response process computes the post-collision velocity by reflecting this velocity vector across the plane's surface. This reflection involves two primary components: one that is perpendicular to the plane's normal vector and another that is parallel to it. The perpendicular component is reversed to emulate the bounce off the plane, while the parallel component remains unchanged.
Mathematically, the reflection of a vector ($\vec{v}$) off a plane with a normal vector ($\vec{n}$) is represented as:

$$\vec{v}\prime = \vec{v} - 2 * (\vec{v} \cdot \vec{n}) * \vec{n}$$

To introduce the notions of elasticity and inelasticity, which influence the energy exchange during collisions, the concept of the coefficient of restitution (COR) comes into play (more information atabout COR 3.2.4). COR, adjusts the post-collision velocities accordingly. The formula for computing the post-collision velocities while incorporating COR is as follows:

$$k = \frac{m * |\vec{v}|^2}{2}$$

$$\vec{v}\prime = \sqrt{\frac{k * COR * 2}{m}}$$

The relationship between COR and kinetic energy is worth to note.

3.3.2 Sphere collision

Continuing our exploration of collision interactions within particles, we now turn our attention to the main topic: sphere-sphere collision. This feature allows particles, to dynamically interact with one another, resulting in a more intricate and immersive simulation. This chapter delves deep into the mathematical foundations of sphere-sphere collision detection and response.

Collision Detection

The saga of sphere-sphere collision unfolds with collision detection, where the simulation determines whether two spheres have intersected. With some basic algebra knowledge this part is quite easy to understand. This is accomplished by analyzing the distance between the centers of the two spheres and comparing it to the sum of their radii. If the distance is less than or equal to this sum, a collision is detected, and the simulation proceeds to the collision response phase.

Mathematically, the distance (d) between two spheres, each with centers $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$, is calculated using the Euclidean distance formula:

$$d = |p_2 - p_1|$$

A collision occurs if $d \leq r_1 + r_2$, where r_1 and r_2 are the radii of the two spheres.

Collision Response

Upon confirming a collision, the simulation proceeds to the collision response phase, where the spheres' post-collision velocities are calculated. Like other collision scenarios, the conservation of momentum and energy remains paramount in ensuring realism.

For two spheres with velocities $v_1 = (v_{1x}, v_{1y}, v_{1z})$ and $v_2 = (v_{2x}, v_{2y}, v_{2z})$, the collision response calculates the updated velocities considering both spheres' masses. The approach typically involves solving a series of equations that account for the conservation of momentum along with the concept of the coefficient of restitution (COR) (described in Section 3.2.4).

The resulting post-collision velocities are computed using the following formulas:

$$\vec{n} = \frac{p_2 - p_1}{|p_2 - p_1|}$$

$$m_{eff} = \frac{m_1 * m_2}{m_1 + m_2}$$

$$v_{imp} = \vec{n} \cdot (v_2 - v_1)$$

$$energy = (1 + COR) * m_{eff} * v_{imp}$$

$$v_1\prime = v_1 + \frac{energy}{m_1}\vec{n}$$

$$v_2\prime = v_2 - \frac{energy}{m_2}\vec{n}$$

Firstly it is calculated the normal of the plane between the colliding spheres, then the mass efficiency (it is a ratio between the masses of the tow spheres), and the impulse velocity of the colliding spheres. With these we can calculate the output energy, and then adding to one and substracting the delta velocity to the other the output velocity can be calculated.

3.4 Time step

In computer graphics, delta time (often abbreviated as "dt") refers to the time elapsed between frames in a real-time simulation or rendering application. It plays a crucial role in ensuring consistent and predictable behavior of graphics and physics simulations across different hardware configurations. The concept of delta time is closely tied to the frame rate at which the graphics engine is operating.
The primary reason delta time is used is to achieve frame rate independence. Different hardware configurations can render frames at varying rates, and without accounting for these differences, simulations might run too fast or too slow on different systems. Here's why delta time is necessary:

Frame Rate Variability Computer graphics applications, especially real-time ones like games and simulations, need to run on a different hardware components. The performance of these setups can vary significantly, leading to varying frame rates. If an application were designed to work at a fixed frame rate, it would run too fast on powerful hardware and too slow on weaker systems. By using delta time, developers can account for these variations and ensure that the simulation progresses at a pace appropriate for the hardware's capabilities.

Frame Rate Independence Delta time allows graphics simulations to be independent of the frame rate. When animations and physics calculations are tied to a fixed frame rate, they become choppy on systems that can't consistently achieve that frame rate. Delta time ensures that movements and changes in the simulation are smooth and proportional to the actual time that has passed, resulting in consistent visuals across different hardware setups.

Physics Simulations Physics simulations involve solving equations that describe the behavior of objects in a virtual environment. These equations often rely on time-dependent variables. Using a fixed time step in simulations can lead to instability and inaccuracies in calculations, especially for fast-moving or complex scenarios. Delta time addresses this issue by allowing simulation updates to be scaled based on the actual time elapsed, maintaining the stability and accuracy of the simulation.

Slowdowns and Speedups Without delta time, running an application on a slower system could lead to the simulation appearing in slow motion, while running it on a faster system could cause everything to move too quickly. Delta time allows the simulation to adapt to the system's performance, preventing these undesirable slowdowns or speedups. The physics are also strictly related to the delta time. To provide realistic movement of the spheres based on the elapsed time between the frames, the velocities and accelerations must e multiplicated by the delta time. If this operation

> would not be done, the sphere would immediately disappear by being moved too far compared to the passed time.

Delta time is calculated as the difference in time between the current frame and the previous frame. Between the two frames every operation (rendering, physics update, settings update, ...) must be included to have the total delta time. This value is then used to scale the rate of change of various simulation parameters. For example, to update a sphere's position, you would multiply its velocity by the delta time and add it to the current position.
In summary, delta time is a critical concept in computer graphics that addresses the challenges posed by varying hardware performance and frame rates. By incorporating delta time into simulations, graphics applications can achieve frame rate independence, smooth animations, accurate physics, and consistent experiences across a wide range of hardware configurations.

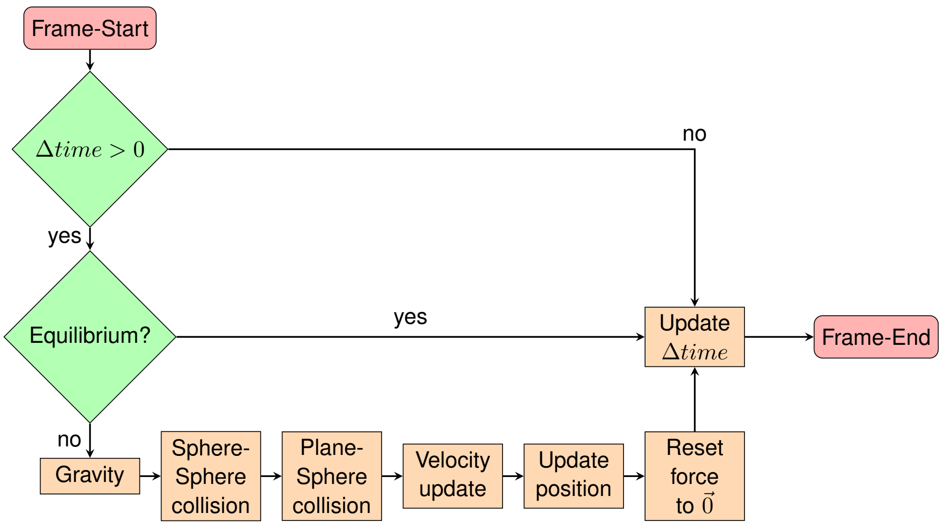
Capitolo 4

Implementation

In this chapter we will see the key details regarding the implementation of the software. Starting with the update loop (Section 4.1), that describe the order of the processes to update the position of a sphere. Continuing with the description of various possible algorithms to improve computational efficiency (Section 4.2). Ending with some details and techniques regarding GPU (Section 4.3).

4.1 Update loop

We start by giving a general overview of the update loop of a sphere. We will see the importance of the order of the processes to compute the new position.
The first check to be done is about the delta time. In the very first cycle this will be zero, after that it is updated and every cycle will not skip the first if statement. Shortly after that, we need to check if each sphere is in equilibrium. Meaning it does not have velocity and the sum of the forces is zero.



If the sphere is not in equilibrium, to the sphere is applied the gravitational force (Section 3.1), checked for collisions (Sections 3.3.1 and 3.3.2), and so its velocity and position updated. In the end the sum of the forces is reset to zero.

4.2 Space partitioning

In the pursuit of performance in particle simulations, optimizing collision detection is one of the focal point, since it is the most expensive operation to be performed (see Section 4.2.1). Among these interactions, sphere-sphere collisions stand as a critical element. Facilitating the interaction between spherical particles in a dynamic environment necessitates thoughtful implementation for accuracy and efficiency. This chapter delves into the intricacies of optimizing sphere-sphere collision detection, exploring techniques that enhance the realism and computational performance of the simulation.

4.2.1 Computational complexity

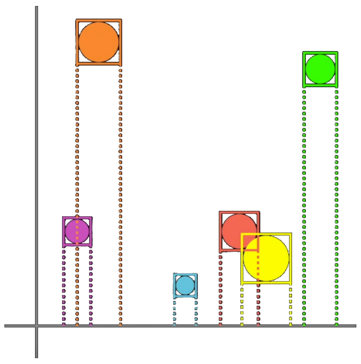
The journey of optimizing sphere-sphere collision detection begins with understanding the complexity of interactions. When simulating numerous particles, a brute-force approach to collision detection—comparing each particle pair can be computationally intensive. As the number of particles increases, the number of comparisons grows quadratically ($O(n^2)$), causing a performance bottleneck.
To overcome this challenge, spatial partitioning techniques come into play. Octrees, k-d trees, sweep and prune, and grid-based methods divide the simulation space into smaller regions, allowing for efficient culling of particle pairs that are far apart. This reduces the number of pairwise checks and enables focusing on potential collisions within the same or neighboring partitions.
Trees are by far the solution with best results by reducing the amount of memory required and having the fasted access to all the data ($O(\log(n))$), but as a drawback they require more time and klnowledge to build a functional structure. The main alternative is grid partitioning. It offers an easier structure, compared to trees, but require some more memory and computational time.

4.2.2 Optimization Techniques

4.2.2.1 Sweep and Prune

This technique sorts particles along a specific axis and efficiently identifies potential collisions by checking overlaps only along that axis, reducing the number of comparisons. Sweep and Prune (SAP) is a sorting algorithm commonly used in particle simulation programs, physics engines, and video games to efficiently determine potential collision pairs

among a large number of objects. The algorithm reduces the number of pairwise collision checks required by quickly identifying objects that are likely to collide, based on their positions along a certain axis.
Being able to sort only ona single axis in our case with three dimensions either the algorythm is applied for all three axis each new frame or use some variants that allow for multiple dimension sorting.
The core idea behind the SAP algorithm is to sort objects along a specific axis and then perform a one-dimensional sweep along that axis to identify potential collisions. It's essential to consider only the interactions between objects that have overlapping intervals along the chosen axis.



Source: `https://www.youtube.com/watch?v=eED4bSkYCB8`
Figura 4.1: Sweep and prune algorithm example

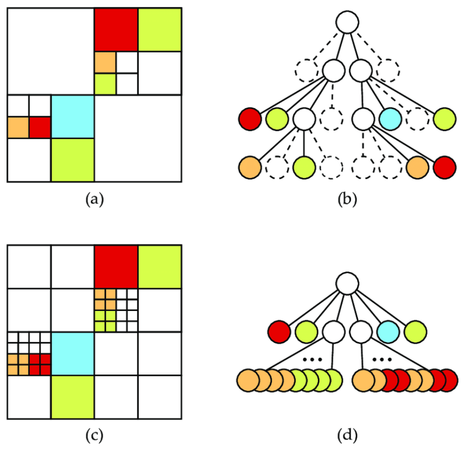
4.2.2.2 Octree

An Octree is a tree-based spatial data structure that partitions three-dimensional space into a hierarchical arrangement of cubical regions known as "octants". Each octant can be further subdivided into eight smaller octants, creating a recursive structure that allows for efficient organization, spatial querying, and collision detection in three-dimensional environments.
Here is a description of the structure of an octree:

Root Node The root of the Octree represents the entire simulation space. It is a single octant that covers the entire volume.

Subdivision Octants are subdivided into eight smaller octants if they contain a certain number of particles or objects. This subdivision continues recursively until a desired level of detail or a minimum size is reached.

Leaf Node Leaf nodes of the Octree are the smallest subdivisions that cannot be further divided. Each leaf node contains particles or objects that fall entirely within its boundaries.



Source: www.researchgate.net/figure/Different-variants-of-octrees-are-shown-Interpreting-nodes-of-an-octree-as-single_fig3_325261315

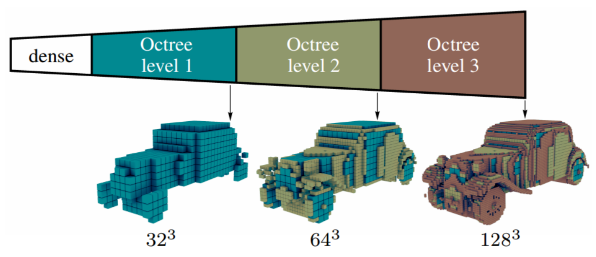
Figura 4.2: Octree space partitioning visualization

When performing collision detection, you start at the root node and recursively traverse the Octree to the appropriate branch based on the position of the query point or the particles involved. During traversal, you determine whether an octant contains potential collision pairs or relevant particles by considering factors like distances or interaction radii. If an octant is deemed "far enough" from the query point or particle, you can stop traversing that branch, effectively eliminating the need to perform detailed checks within that region.
Some considerations about the octrees:

Sparse Distributions Octrees are particularly useful for simulations where particles are not evenly distributed throughout the space. They help efficiently manage regions with varying particle densities.

Efficient Queries Queries like finding nearby particles or objects become efficient due to the hierarchical nature of the Octree. You only need to traverse relevant branches.

Leaf Node Leaf nodes of the Octree are the smallest subdivisions that cannot be further divided. Each leaf node contains particles or objects that fall entirely within its boundaries.



Source: `https://lmb.informatik.uni-freiburg.de/people/tatarchm/ogn/`
Figura 4.3: Octree level of details

More detailed information on this topic can be found in [1].

4.2.2.3 Grid partitioning

Grid partitioning is a spatial data structure and computational technique employed in particle simulations to efficiently manage and process large numbers of particles within a confined space. By organizing the simulation space into a grid structure, grid partitioning enhances the overall performance and scalability of particle simulations.
At its core, grid partitioning divides the simulation space into a regular grid composed of cells. Each cell has a fixed size, ensuring uniformity throughout the simulation area. The particles within the simulation space are assigned to the cells based on their current positions. This spatial organization enables more localized and efficient computations by focusing computations on particles that are likely to interact due to their proximity.

The simulation space is divided into a grid of cells, where each cell covers a fixed portion of the total area. The number of cells and their dimensions influence the trade-off between memory overhead and computational efficiency. This result in a limited number of possible cells based on the memory available.
Each cell in the grid is assigned a unique index, typically in a multi-dimensional array. This index allows for quick access to particles within a specific cell, facilitating rapid neighbor searches and interaction checks. So, particles are assigned to cells based on their spatial coordinates. This assignment is often accomplished by calculating the grid indices corresponding to the particles' positions, usually the center of the particle is used as a reference of the position.
Having the coordinate of a cell, and so the index of the container cell, means we can search also neighbor cells. For a given particle, neighboring cells in all directions are examined to identify particles that may interact with the target particle. This drastically reduces the number of potential interactions that need to be evaluated, leading to substantial computational savings. Once neighboring particles are identified using the grid structure, collision detection algorithms and interaction computations can be applied selectively to the relevant particle pairs. This targeted approach significantly reduces the number of unnecessary

computations.

Being probably the most simple space partitioning algorithm, does not come for free. An impartart drawback is the scalability. Let's do some math, let the space have a total of 27 cells, being divided as a 3x3x3 grid. Inside our simulation we have 1000 particles, this means we might need to save all the particles inside one single cell, and this could happen for all the cells. For each cell we should then store 1000 integers. Multiplying that by the number of cells results in 27000 integers, $4bytes * 27cells * 1000spheres = 108000B = 108KB$. Scaling this to have a better result, 10x10x10 grid and 10000 particles, the same calculation returns $4bytes * 1000cells * 10000spheres = 40000000bytes = 40GB$. To make this worse, most likely the majority of the cells will not contain any sphere. Because, the simulation at one point will reach a stable state and all the spheres will concentrate on the bottom.

Beside this potential problem, this is the solution I chose to implement. Mainly because building a stable and working tree structure on GPU memory was out of my reach, in the available time with my current knowledge.
The implementation is not particularly diffucult, but it requires some caution especially if it runs in parallel instances of the compute shader. Starting with the data storage I created an SSBO called "Grid" that contains all the indecies of the spheres inside each cell, and an "Atomic Counter Buffer" ("Grid Counter") that contains the number of spheres inside each cell. The atomic counter buffer is a particular buffer type provided by OpenGL that offers functions to update and get the values from/to concurrent programs.

Here is a little example of usage of the atomic counter buffer. In this example we loop through the neightbor cells of the cell that contain the current sphere, and check if the is a collision.

```
int i = int(gl_WorkGroupID.x);

void checkSphereCollision()
{
    for(int cell : neighbor_cells)
    {
        for (int c = 0; c < atomicCounter(counters[cell]); c++)
        {
            int b = cells[cell * spheres_count + c];
            if (b != i)
            {
                if (length(position[b] - position[i]) <= radius[i] +
    radius[b])
                {
                    sphereBounce(i, b);
                }
            }
        }
    }
}
```

4.3 GPU details

4.3.1 Transformation matrices storage

Storing data on the GPU is a task a little bit different from a more common CPU program. In fact during the execution of a programm running on a GPU it is not possible to allocate new memory. This must be done on CPU side providing the required memory size (generally in bytes) and the data that should be stored in that same space. Modifying the size allocated during the shaders execution can not be done, GPU are tought to be highly optimized for speed at the cost of flexibility.
Genrally VBO (Vertex Buffer Object) works just fine, but if the data needs to be modified this can not be done with a VBO. The best alternative is a buffer thought specifically for compute shaders, Shader Storage Buffer Object (SSBO). SSBO is a buffer type within the OpenGL graphics API that allows shaders to read from and write to buffer objects in a more flexible and unordered manner compared to traditional textures or uniform buffers.
SSBOs are particularly useful for scenarios where multiple shader invocations (such as compute shaders) need to share data between them or update data in a non-sequential order. These buffers can store various types of data, including structured data, arrays, and even user-defined structures. It is up to the programmer to make sure that everything works fine. If each thread writes into its own memory block, explicit synchronization can and should be avoided for optimal speed. If the memory is accessed randomly, OpenGL provides ways for synchronizing memory access via memory barriers or atomic operations.
SSBOs provide a way to achieve inter-shader communication and synchronization, but they don't guarantee any particular order of execution between different shader invocations. This allows for parallelism and flexibility, but it also means that extra care must be taken when designing algorithms to ensure correctness.

In the case of a particle simulation, various data must be stored like: positions, dimension, velocities, forces, and masses. These data types can be stored as flat number, vectors or matrices. Saving on memory a 4x4 matrix for each sphere is a bit of waste of memory. Because being spheres rotation does not matter and the scaling is equal in all 3 dimension, like this parts of the matrix would always be filled with zeros and with number repetition on the main diagonal. A solution to store only the minimum data without any losses is to save in a vec4 position and dimension, $(pos_x, pos_y, pos_z, dim)$.
A similar thing can be done with forces and masses, storing them in a single vec4, (f_x, f_y, f_z, m).

Velocity requires a inverse point of view, because being a vector can be easly stored as a vec3. But often is required to know only the direction or the magnitude of an object. These two require more math to be calculated from the vector itself rather can calculating the resulting veloctity from these two.

$$|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

$$\vec{u_v} = \frac{\vec{v}}{|\vec{v}|}$$

$$\vec{v} = |\vec{v}| * \vec{u_v}$$

In this case we prefer to store more data to reduce the amount of computation with need to do multiple times for every frame.

4.3.2 Compute Shader

Traditionally, GPUs (Graphics Processing Units) were mainly used for creating visuals. But with compute shaders, GPUs have taken on a new role beyond graphics. Compute shaders break free from graphics and let GPUs handle tasks like simulations and data processing.
In the world of computer graphics and computation, GLSL (OpenGL Shading Language) compute shaders are powerful tools. They let developers tap into the potential of parallel processing on modern graphics hardware. In this chapter, we will see GLSL's compute shaders architectural structure, tasks, and things to watch out for.

4.3.2.1 Architecture

Compute shaders strategically leverage the parallel architecture inherent to modern GPUs. At their core, compute shaders represent programs orchestrated to function across an array of threads, each provided with the capability to execute discrete computations. These threads, unite into workgroups, form the bedrock of parallel execution. A single workgroup works with a multitude of threads, which collaboratively engage in concurrent operations, capitalizing on the innate parallel power of GPUs.
The execution of a compute shader commences with the process known as dispatch. The function glDispatchCompute() is called up, setting the number of workgroups required for the task. Each workgroup operates upon a designated segment of the computation.

4.3.2.2 Concurrency

Harnessing compute shaders efficaciously demands familiarity with the hardware architecture. Memory access patterns, thread divergence, and data dependencies emerge as key performance influencers. Optimizations leveraging local memory and memory contention become imperatives to unravel the full performance potential. Profiling tools and GPU

debugging environments emerge as indispensable allies for pinpointing performance bottlenecks, even simply displaying a sphere with a certain color can be seen as a type of debugging tool.
In a workgroup, threads collaborate and synchronize through the medium of shared memory. This shared memory transmutes into a conduit of communication, allowing threads to exchange essential data. Yet, to ensure harmonious and coherent data sharing, prudent synchronization mechanisms such as barriers, or manually built alternatives 4.3.4, lay the foundation for consistent results.

4.3.2.3 Applications

Compute shaders' versatility finds manifestation across diverse domains. Their prowess reigns supreme in the realm of simulations, where they simulate intricate systems as fluid dynamics, particle simulations, or raytracing. Lately, compute shaders carve a niche in machine learning, offering a potent engine for neural network inference, courtesy of their parallel prowess. Despite their general-purpose nature, compute shaders logically are still a main part of graphics rendering. They contribute to post-processing effects, generate data for graphics pipelines, ... This harmonious synergy transcends the conventional boundaries of graphics and computation, having fantasy as the only limit.
In the specific case of this application, a particle simulation, a compute shader is dispatched for every sphere present in the scene. Allowing each sphere to compute its own position, velocity, acceleration and collisions.

4.3.3 Instancing

Imagine having a scene were there are only a few objects to draw, for each one of them you need to call a draw command. Even in an older and simpler hardware that scene could be rendered without any major problem with a stable and good framerate. Now, let's make that scene quite bigger. For example, it could become a forest, thousands of trees (or more) should be drawn one by one. Making it potentially quite hard even for modern hardware to keep up with the workload.
Let's do some simple math to understand how much it cost to draw a scene with a minimum framerate of 30 FPS. 1000 trees x 30 FPS = 30000 draw calls each second. To this we should also add the time shaders need to calculate materials and lighting, position relative to the camera, apply transformations, ... As it is easily understandable the computation required scales up quickly.
So, there's a solution to reduce the number of calls? The answer is yes, and it is called "Instanced rendering". The idea is simple but require some understanding. Instead of having to load the same mesh multiple times and applying to them different transformations and attributes relative to each tree, only one mesh is loaded, and the number of instances

required is equal to the number of transformations. Like this, if we go back to the previous forest example we reduce the number of call per frame to only 1 and to 30 each second with a minimum 30 FPS.
OpenGL provides two different methods for instanced rendering replacing glDrawArray and glDrawElements with glDrawArraysInstanced and glDrawElementsInstanced. For our example let's say we decided to work with glDrawArrays, this would have been a part of our code before instanced rendering:

```
for(unsigned int i = 0; i < amount_of_models_to_draw; i++)
{
    DoSomePreparations(); // bind VAO, bind textures, set uniforms etc.
    glDrawArrays(GL_TRIANGLES, 0, amount_of_vertices);
}
```

After having implemented instanced rendering simplifies to:

```
glBindVertexArray(VAO);
glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100);
```

Were the required parameters are: the kind of primitive to render (ex. GL_TRIANGLES most commonly), the starting index of the array, the number of indices to be rendered for both glDrawArray and glDrawArraysInstanced. Additionally for glDrawArraysInstanced we need the number of instances we want to render.
This is very cool but, how does OpenGL knows when the data of an instance finishes and start the next one? OpenGL needs an attribute to be enabled, in this case is the VertexAttribArray. During the binding of the data this attribute need to be enabled and requires some parameters.

```
unsigned int instanceVBO;
glGenBuffers(1, &instanceVBO);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec2) * 100, &translations[0],
    GL_STATIC_DRAW);
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void
    *)0);
glVertexAttribDivisor(2, 1);
```

First, glEnableAttribPointer is called passing as parameter the index of the buffer on which you want to activate the attribute. Secondly, with glVertexAttribPointer we provide some information about how data is stored: index of the attribute, number of components (for example if we have stored vec2 is 2), the type of the data stored (GL_FLOAT, GL_BYTE, ...), if the data should be normalized, offset between consecutive attributes and then an offset from the starting pointer. At last, glAttribDivisor providing the same index and the number of elements that need to pass between each instance. Now, all the data is loaded into the GPU and it is ready to be transformed to display some results. Inside the vertex shader is where these data are converted, here is an example of the shader before instanced rendering:

```glsl
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out vec3 fColor;

uniform vec2 offsets[100];

void main()
{
    vec2 offset = offsets[gl_InstanceID];
    gl_Position = vec4(aPos + offset, 0.0, 1.0);
    fColor = aColor;
}
```

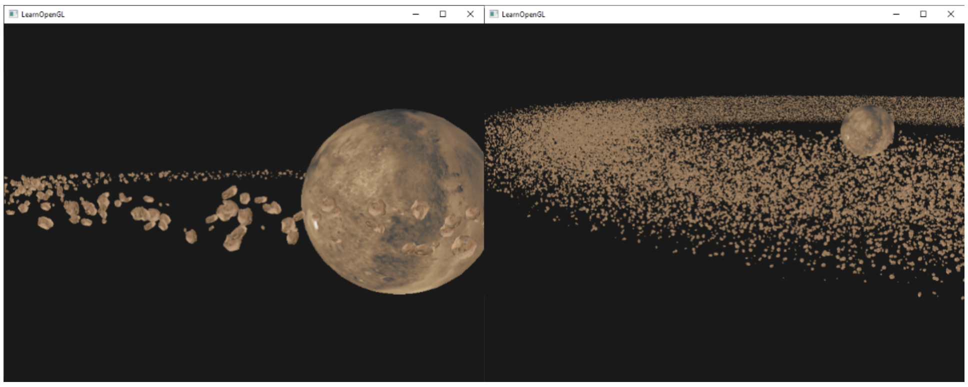
As we can see the offsets are stored inside an uniform of vec2. We are able to get the correct element from the array making use of gl_InstanceID, but this is potentially inconvenient. If the number of elements stored inside the uniform would grow it might reach the limit of the capacity that the shader is able to handle, causing a loss of data and the consequent wrong execution of the code. Having enabled the previous attribute, we can finally make use of it loading into the shader only the minimum required data.

```glsl
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aOffset;

out vec3 fColor;

void main()
{
    gl_Position = vec4(aPos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

The previous uniform is now loaded as an input and the code gets even simpler and cleaner than before. Instanced rendering is always used when a lot of the same model needs to be rendered with different transformations, because having to do only one single call instead of multiple thousands is a remarkable improvement even for modern hardware.

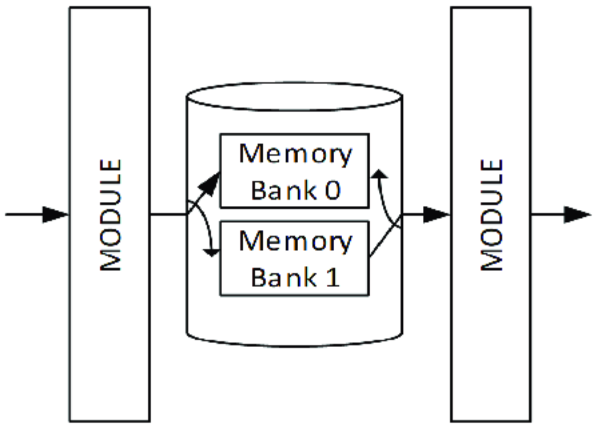


Source: `https://learnopengl.com/Advanced-OpenGL/Instancing`
Figura 4.4: Instancing asteroids example

Code examples in this chapter: `https://learnopengl.com/Advanced-OpenGL/Instancing`

4.3.4 Ping-pong buffer

Double buffering is a technique often employed in various topic in computer science. In the case of computer graphics, it is commonly used to prevent visual artifacts (ex. flickering and tearing) and provide smoother animations.
The ping-pong buffer is a particular type of double buffer. Usually in a double buffer the data is swapped between the two buffers, in this case instead, only the pointers are swapped.



Source: `www.researchgate.net/figure/Ping-pong-buffer-Double-buffering_fig3_347478182`
Figura 4.5: Ping-Pong buffer example

In this case, the two buffers have two different tasks. Let buffer 1 be named B1 and buffer 2 B2. At the start B1 and B2 contain the same exact data.

After the first instances of the computer shader are dispatched, B2 will contain data of the "next" frame. This data is generated from the "previous" frame B1. What is contained inside B2 at the start is no more important since it was swapped after the movement of the spheres was already computed.
B1, apart from being a source of the future data, is used by the GPU to render the spheres. The buffers are binded to an ID when they are first created, after that it is posible to bind them to another ID. Providing the target (GL_SHADER_BUFFER in this case), the new index and the buffer that we want to bind.

```
bool check = false;
void Mesh::pingPongBufferSwap() {
    if (check) {
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 3, vboTransform);
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, ssboTransform);

        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 5, vboVelocity);
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 6, ssboVelocity);
    }
    else {
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 3, ssboTransform);
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 4, vboTransform);

        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 6, vboVelocity);
        glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 5, ssboVelocity);
    }

    check = !check;
}
```

Implementing the ping-pong buffer in this case was needed to prevent possible concurrent read/write of the sphere's position and velocity. Having each sphere's position, velocity and force calculated by different instances of the same compute shader, we cannot know in what order are they starting and at what point are they checking for collisions.
Having an instance of the compute shader run for each sphere in a crucial part to improve the computational speed of the program so a solution like the ping-pong buffer is a must have to keep the entire data stable and correct.

Capitolo 5

Performance test

Since the start of the development, I knew that having a stable program with each component ruinning correctly without any error was the most important thing.

All the development and tests were run on a desktop computer having as GPU an NVIDIA RTX 3060.

The starting engine as a mix of the one from the Computer Graphics course and Virtual Reality Course (thanks to Bodino A. and Garatti A. for letting me use parts of the updated engine). The very first result was a scene with many sphere floating without moving in the air.

In this version the program was running at around 150 fps with 8000 spheres. Any type of optimization was implemented.

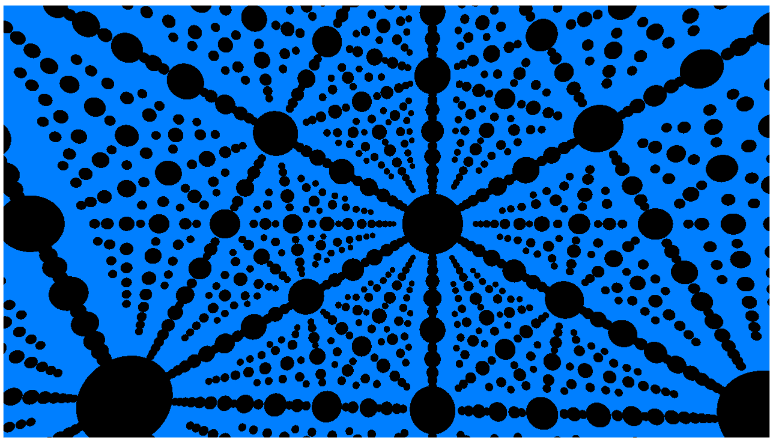


Figura 5.1: Starting point with spheres

With a base program to work on, the next step was to preparing the software to support instanced rendering (IR). The full jump directly to IR would have been to risky to do all at once, becase being a tecnique that I have never used I did not know which could have been the problems and how to debug.

So, I decided to load a single sphere mesh but all the required matrices. This resulted in an improvement with the framerate, increasing to 260 fps with the same exact result.

Now, it was time to implement IR. I didn't have any major problem, and the result were quite promising. The exact same scene was scaled up to 27000 spheres with still an important 175 fps. The details of IR can be found at 4.3.3.

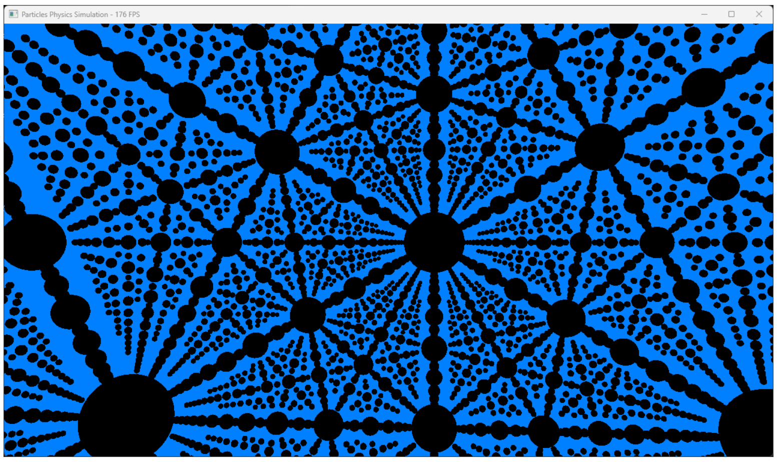


Figura 5.2: Sphere count increase with instanced rendering

Completed the basic rendering it was time to move the spheres, learning from zero how compute shaders work was not exactly the easiest thing ever. But once I understood how to dispatch and the concept of workgroup, the implementation of the sphere's movement was straight forward. The physics implemented is available at 3.1 and details of compute shader at 4.3.2.

Professor Schärfig at this point, suggested that it would be useful to implement a ping-pong buffer to better manage the update of position (for more 4.3.4) and reduce the number of triangles for each sphere, because in the final product spheres will be very small and thaving 1000 triangles or 30 per sphere would not give any graphical improvement. Instead, it would only make the program run slower.

The next step was to make the spheres bounce on planes, the theory of the math behind this can be found at 3.3.1. I started with 5 simple planes creating a shape of an open box.

Here I had the first real problem, some spheres sometimes would be stuck into a plane because the output velocity vector would not allow it to move far enough to not detect on the next frame the same collision. The solution to this was to check for the collision for the next frame, if there was one it could have been computed before the collision actually happened resulting the expected behaviour.

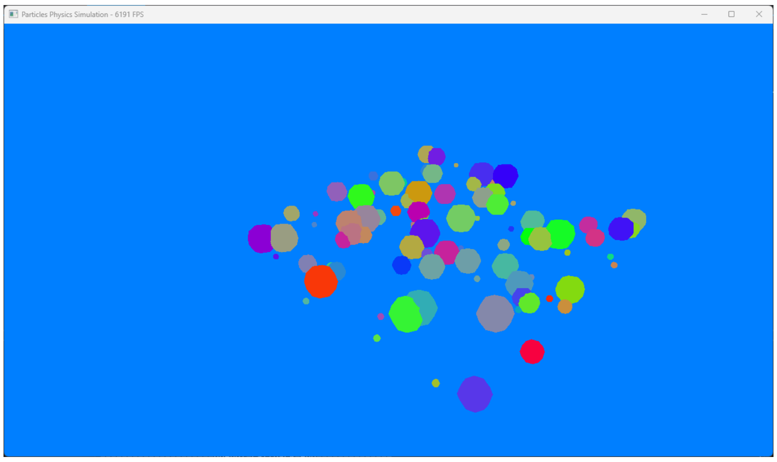


Figura 5.3: Spheres standing still on a plane

Another problem I noticed was that spheres were moving too fast, that was related to the frequancy of the dispatch of the compute shader. To make it stable so every frame would have the same impact on the movement I added a delta time value. This consist of the time between two frames, multiplying the acceleration and velocity by this factor make each fram equal to the others. For more information on this see 3.4.
It was finally time to add the sphere-sphere collision. Having the math sorted out 3.3.2, I knew that the same problem that occured with plane collision probably would have happened here again, so I preventively implemented it in the same way.
An unexpected behaviour appeard, spheres should stack on top of each others when equilibrium is reached. But instead they would slowly go through the spheres stanting still on the plane, without generating any collision response. With the help of professor Schärfig, we found out that there was a problem with the sum of the velocities after testing for all the collisions. Running out of time for the submittion, I implemented a temporary fix. Basically if after colliding with another sphere the sphere would collide again the position does not get updated. A better solution for this would be to sum the direction of all the collision and create a plane on which to make the sphere bounce.
As expected the framerate had a big drop, with 10000 spheres it was running on about 20 fps.
To solve this lack of performance some kind of space partitioning 4.2 was required. The selected algorithm was a simple grid cell partitioning. Making shure this was working as expected was a bit more complicated. The solution I found was to pair the ID of a cell with a color, the moving spheres would then change color based on the container cell.

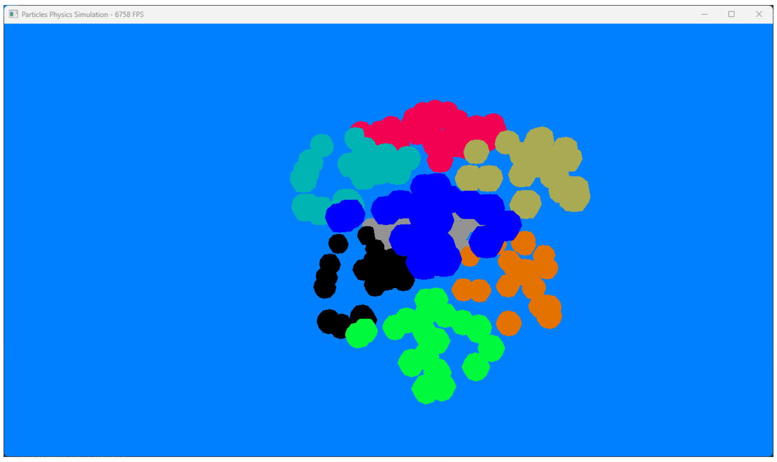


Figura 5.4: Debugging grid partitioning

Now that everything is implemented here is the final result, 50000 spheres bouncing and stacking on top of each others with a good framerate of about 45 fps. The program is also able to handle 100000 spheres with some framerate limitations, about 20 fps. The implementation of the space partitioning algorithm improved the performance of the simulation of a factor of 10, a very good result.

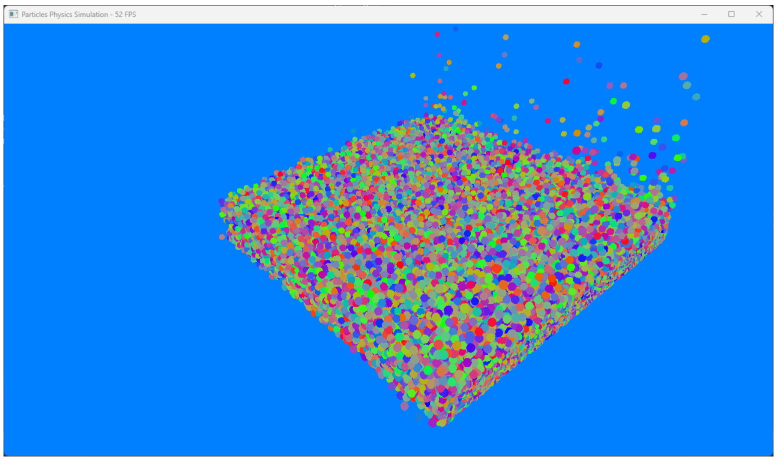


Figura 5.5: Final result with almost full equilibrium

Conclusion

Coming to an end with this project, I would say the result can be used for a couple of interesting considerations, one about the tecnology and one more personal.

First, it is surprising how far did the computational power of GPUs arrived. Even being alone developing a project on GPU I think the result is quite good. Comparing the final result to more professional solutions developed by more people with more time, it is promising. It requires more work to be more generalized, like being able to import a scene from an external file or have lights and materials to have a better visuals.

On a more personal note, I am happy with the work I have done. I think the organization of the project went well, I was able to stick to most of the plan. I have learned a lot, from more technical stuff related to GPU to more thoughtful decisions to make before starting to implement. A wrong decision made without thinking too much can cause a lot or rewriting of code that was considered good, this is also applicable to memory management.

Bibliografia

[1] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional, 2005.

[2] Xinlei Wang, Yuxing Qiu, Stuart R. Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. A massively parallel and scalable multi-gpu material point method. *ACM Trans. Graph.*, 39(4), aug 2020.

[3] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Trans. Graph.*, 33(4), jul 2014.

[4] Marco Santos Souza, Tiago Nobrega, André Ferreira Bem Silva, and Diego D. B. Carvalho. A rigid body physics engine for interactive applications. 2011.

[5] Russell M Templet. Game physics: An analysis of physics engines for first-time physics developers, 2020.

[6] Nvidia flex. `https://developer.nvidia.com/flex`.

[7] Gpu-based adaptive octree construction algorithms. 2008.

[8] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Chapter 1 octree textures on the gpu. 01 2005.

[9] Opengl wiki. `https://www.khronos.org/opengl/wiki/`.

[10] Lighthouse3d.com - opengl atomic counters. `https://www.lighthouse3d.com/tutorials/opengl-atomic-counters/`.

[11] Learnopengl - instancing. `https://learnopengl.com/Advanced-OpenGL/Instancing`.

[12] Wikipedia - inelastic collision. `https://en.wikipedia.org/wiki/Inelastic_collision`.

[13] Wikipedia - elastic collision. `https://en.wikipedia.org/wiki/Elastic_collision`.

[14] Wikipedia - coefficent of restuition. https://en.wikipedia.org/wiki/Coefficient_of_restitution.

[15] Stackexchange - game development - ball bounce in 3d. https://gamedev.stackexchange.com/questions/133599/ball-bounce-in-3d.

[16] Reducible. Youtube - building collision simulations: An introduction to computer graphics. https://www.youtube.com/watch?v=eED4bSkYCB8.

[17] Pezzza's Work. Youtube - writing a physics engine from scratch. https://www.youtube.com/watch?v=lS_qeBy3aQI.

[18] Pezzza's Work. Youtube - writing a physics engine from scratch - collision detection optimization. https://www.youtube.com/watch?v=9IULfQH7E90.

[19] Sebastian Lague. Youtube - i tried creating a game using real-world geographic data - in which a cube becomes a sphere. https://www.youtube.com/watch?v=sLqXFF8mlEU&t=36s.