FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Implementation and Evaluation of Acceleration Structures for Inhomogeneous and Polydisperse Particle Dynamics

Christian Koch

Bachelorarbeit

Implementation and Evaluation of Acceleration Structures for Inhomogeneous and Polydisperse Particle Dynamics

Christian Koch

Bachelorarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	M. Sc. S. Eibl
Bearbeitungszeitraum:	7.1.2020 – 13.7.2020

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 13. Juli 2020 .....

Abstract

Simulations of particle dynamics rely on the ability to find colliding particles in an efficient manner. Acceleration structures can be used to minimize the amount of collisions that need to be checked. This thesis will focus on implementing two acceleration structures based on spatial partitioning that are well suited for two types of scenarios. In the first scenario particles are very unevenly distributed while in the second there is a large discrepancy in the size of the particles. The first issue will be tackled by introducing an octree able to dynamically partition the simulation space based on the distribution of particles in the simulation space. The second issue will be tackled by introducing a hierarchical hash grid, managing multiple grids for corresponding particle sizes.

The data structures were created for the WaLBerla Framework, a highly parallel simulator for fluids and particles. This thesis will provide insight on the underlying ideas behind the acceleration structures and how they were implemented. Furthermore their performance during the simulation of different scenarios is analyzed to figure out in which cases they are better suited than the general purpose Linked Cells structure, which has been the main acceleration structure for WaLBerla until now. Criteria are runtime performance and their efficiency as an acceleration structure.

Contents

List of Figures

List of Tables

1 Introduction

Simulation is a concept widely used in industry and research. It can be a huge advantage to virtually simulate a project, instead of having to build it to test it in the real world. Simulation can save money and resources, it also enables the optimization of issues that would be hard to test in practice.

One kind of simulation deals with particle dynamics. In order to resolve a given scenario, for example dumping a bucket of sand, various physical effects have to be modeled and computed. These include factors like heat, movement and distortion, which generally apply to each particle individually, but also collision, which happens between two particles. Figuring out which particles collide with each other is not a trivial task though. Finding all collision pairs can prove to be computation intensive, as checking each particle against all other particles leads to $\mathcal{O}(n^2)$ checks. As this would make large scale simulations not feasible, a more efficient approach to collision detection is needed in practice.

The process is usually divided into different phases: Coarse collision detection and fine collision detection [1]. Coarse collision detection breaks down which particle pairs should be considered and which can safely be discarded. Fine collision detection then checks whether the particle pairs provided by the coarse step actually collide. In implementations of coarse collision detection the most prominent underlying concept is spatial partitioning [1], meaning that particles located in different parts of the simulation can be excluded from checking. This can achieved by using various data structures like uniform grids, BSP trees and octrees [1]. A data structure designed to implement coarse collision detection is also called acceleration structure.

Once the majority of particle pairs have been sorted out the fine collision detection can begin. Particles can have very complex shapes, leading to difficult calculation whether two complex shapes collide or not [1]. To reduce the amount of computation even more, the shapes are simplified first. Using a bounding box or sphere, each shape is modeled roughly. If the bounding models do not collide, the complex shapes cannot collide. This proves useful since finding out whether two shapes touch is a lot easier if the shapes are simple. Consequently, all pairs where the bounding models do not collide are sorted out. Finally the remaining particle pairs are checked against each other, using their actual shape.

In this thesis the focus lies on coarse collision detection and the acceleration structures used during that phase. In the WaLBerla Framework [2], a framework for massively parallel simulation, an acceleration structure named Linked Cells is in use. While the acceleration structure performs well in most scenarios, Linked Cells has two weak spots that leave room for improvement. One weak spot is handling an uneven distribution of particles. The second is handling particles with very different sizes.

In this thesis two acceleration structures, designed to complement the issues of Linked Cells, were implemented and evaluated. First of all in section 2, concepts that are necessary to understand the way the acceleration structures work will be clarified. In the beginning, the Linked Cells acceleration structure will be described, to illustrate the need for alternatives. The problematic inhomogeneous scenario will be described, including the reasons why Linked Cells struggles with it. Then, the octree based acceleration structure will be introduced, highlighting key features that help to perform well in the posed scenario. The same will be done with the polydisperse scenario and the third acceleration structure, a hierarchical hash grid. Its implementation is based on the work presented in [5]

After the fundamental concepts are clear, the actual implementation in C++ will be shown in section 3. For both acceleration structures, the three relevant functionalities will be iterated: inserting a particle, finding all collision pairs, and clearing the data structure.

In the section 4 the performance of the implemented acceleration structures will be tested. In various theoretical scenarios, they will be compared with Linked Cells to find out in which cases the new acceleration structures are advantageous. The general performance of the octree and the hierarchical hash grid will also be tested.

Appendix A and Appendix A contain the code segments that are showcased, but would disrupt the reading experience.

2 Concepts

In this section,

2.1 Simulation Environment

Acceleration structures are a small gear in the process of simulating particle dynamics. This section aims to describe the factors influencing the design of the acceleration structures. The environment in the WaLBerla Framework will be introduced shortly and terms recurring in this thesis are defined. The data structures presented in this paper were designed to complement the WaLBerla Framework [2], in short WaLBerla. Simulation in this framework is based on using time steps. The idea behind time steps is to split time into short periods and to simulate them step by step. The events happening in one second could be simulated by accumulating 1000 time steps, each 1ms long.
As reality involves time and space, simulations need a model to represent the space where all objects are located. This model uses a coordinate system to address locations. Finite space is modeled by defining a boundary, usually represented by a cuboid, specifying which coordinates are out of bounds. The valid portion is regarded as the domain.

Although parallelization is not a subject of this thesis, the fact that WaLBerla is a massively parallel framework influences the environment of the acceleration structures:
The domain is always restricted to a cuboid.
In a parallel simulation, particles would be able to leave and to join the domain. Consequently, the amount of particles is not fix and sometimes particles need to be removed from acceleration structures.

Updating the data structures each time step is necessary to ensure that the correct amount of particles is used in calculation. Additionally, particles might need to update their position in the acceleration structure. The simple approach is to remove all particles from the data structures and to reinsert all that are present in this time step. This strategy is used throughout this paper. An alternative would be to only reinsert particles that have to be stored in a different partition. Additionally particles that entered the current domain have to be inserted and ones that left the domain need to be removed. As presented in the benchmarks section 4, the main time factor is how efficient the collision detection is, but some performance could be improved by optimizing the update strategy.

Particles can have different shapes and sizes, although every shape can be put into a bounding sphere (see Figure 1). For coarse collision detection and thereby for the acceleration structures, rough models of particle shapes are sufficient. To define the size of a particle, acceleration structures use its bounding sphere instead of its actual shape. Throughout this thesis, the term *particle size* refers to the diameter of its bounding sphere.
To define the location of an object, a point in the shape is chosen, for example the center point of a sphere. Using this point, the position of an object can be expressed explicitly.

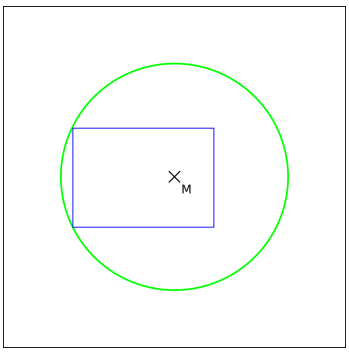

Figure 1: Bounding circle (2D-equivalent of a bounding sphere, green) with fixed center point (M) of a rectangular object (blue). It should be noted, that any efficient design would choose a center point that minimizes the radius needed

Particles are stored in a data structure separate from the acceleration structure in use. To reference which particles are stored inside the acceleration structure the index in the particle storage

is utilized, so that only one integer needs to be stored to identify a particle. This index is referred to as *particle ID*.

2.2 Linked Cells

Linked Cells [6], short LC, is a general purpose acceleration structure with a simple concept. The domain is divided into a three dimensional uniform grid of cells that store particle IDs. Its implementation is not part of this thesis as it was already functioning part of WaLBerla [2]. This thesis provides alternatives in scenarios where Linked Cells is not efficient. Linked Cells will serve as a reference for analyzing performance (see section 4).
When a particle is inserted into the structure, the cell containing the location of the particle is found. This cell stores the particle's ID, completing the insertion.

In LC each particle is assigned to a single cell and all cells are cubic. Cell size refers to the length of a side of these cubes. As all cells have the same size, this cell size can also be attributed to the whole LC.

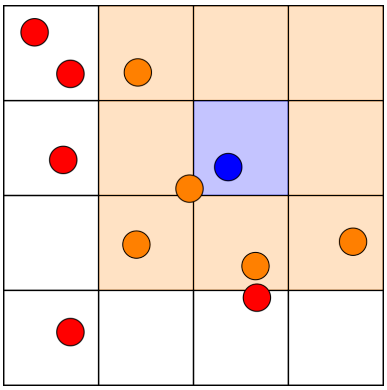

Figure 2: Blue cell checks for potential collisions with the blue particle in adjacent cells (orange).

The efficiency of collision detection is dependent on the size of the cells. As the grid is uniform, the cell size can be attributed to the whole LC structure, not just to one cell.
Collision detection only considers a particle pair if both particles are stored in the same cell or in adjacent cells. This will only work correctly if the cell size is not smaller than the diameter of the biggest particle, else some collision might not be detected (see Figure 3). However, determining the right cell size is not trivial:
The smaller it is the more efficient the collision filtering will become. Since LC stores $\mathcal{O}((\frac{1}{n})^3)$ cells, n being the cell size, the amount of cells can quickly become critical to the memory. Runtime performance is also affected by the number of cells, as all cells are iterated during collision detection.

2.3 Inhomogeneous Scenario

Inhomogeneity in particle dynamics means that the particles are very unevenly distributed across the domain. This means that many of them are in one or more dense places while there also are areas where almost no particles are present. The early stages of an explosion are an example for this condition.
This is a weak spot of LC since the cell size has to be very small to be efficient in very dense areas. A small cell size will in turn lead to lots of wasted resources managing the empty parts of the simulation. The optimal cell size (based on runtime) will often end up with lots of room for improvement when it comes to filtering collision pairs.

2.4 Octree

In the previous scenario, Linked Cells was conflicted between small cell size for filtering efficiency and large cell size, so empty parts of the domain would need less time to iterate. Unfortunately, LC

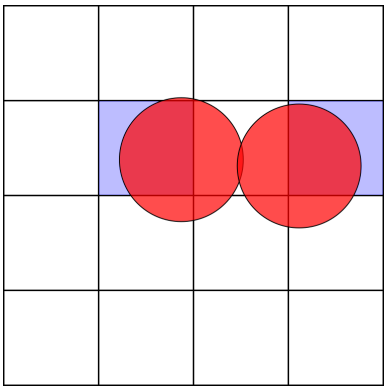

Figure 3: Undetected collision because the cell size is too small: The particles (red) are stored in non-adjacent cells (blue), but still overlap

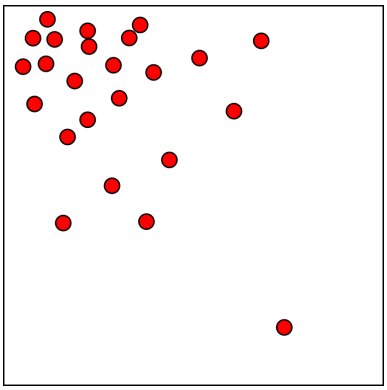

Figure 4: Inhomogeneous distribution of particles (red)

can only choose the same size for all of its cells. The idea behind this acceleration structure is to enable having cells of different sizes, managing the conflicting interests in inhomogeneous scenarios. With the use of a tree structure [4], the domain is partitioned into smaller cells when many particles are present.
Just like a square can be split into four smaller squares of equal size, a cube can be split into eight small cubes of equal size. Thus, the dynamic partitioning of the domain can be represented with an octree, a tree where all inner nodes have eight children. This tree is why the structure will be regarded as *Octree* The root node administers the whole domain, its children each administer one octant. As follows, one location is not mapped to a specific node, but it can be mapped to a specific leaf node. Consequently particle IDs may only be stored in leaf nodes.
Apart from the different structure, the concept for insertion is the same as in Linked Cells. Given the location of the particle, the corresponding leaf containing the position is found and stores the ID. The difference lies in is how the right leaf is found. To find it, the tree starts at the root. By comparing the particle position to the root's center point, the octant containing the position of the particle is found. Then the previously found node has to repeat this procedure, stopping once a leaf is found. The recurring pattern invites for a recursive implementation (see section 3).
When a node has to find the child containing the searched position, one comparison with its center point should be enough. To translate the result of the comparison to an index in the children array, the array uses a helpful layout (see Figure 5). Looking at the numbers in binary system might help to understand the idea. The numbers 0-7 can be represented in 3 bit. One for each coordinate axis. Each bit is 0 if the coordinate value is smaller than the one of the center point, else the bit is 1. This way the index encodes the position in relation to the center point.

So far there is no model that defines when a leaf has to section itself. In order to find a suitable condition for splitting into eight children, the goal of this structure should be kept in mind. In

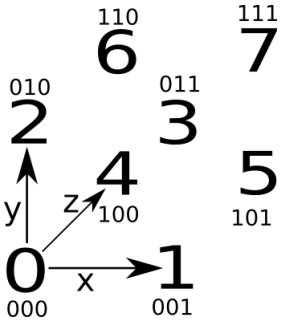

Figure 5: Layout of the children vector, including binary numbers

places with little to no particles, there should be few partitioning, leading to large leaf cells and in densely populated areas there should be lots of partitioning, resulting in small cells. A rule that enforces this idea is to split once a certain threshold of IDs is reached in a leaf node. Then a leaf will turn into an inner node and its content is transferred to the corresponding children (see Figure 6).

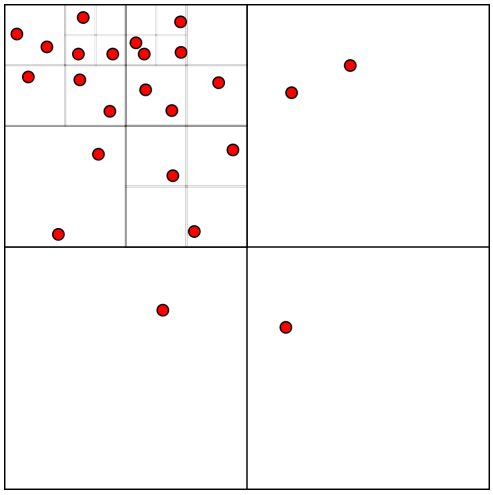

Figure 6: Quadtree (2D-equivalent of an octree), constructed by splitting when 3 particles are stored in a leaf

The most important feature of the data structure is its ability to determine collision pairs. In contrast to LC this task is a lot more complex. LC can store the cells in a linear cell array where it can be calculated which cells are adjacent (concept returning in Figure 10). Unfortunately, the final structure of the Octree is not clear during construction. Furthermore, finding adjacent nodes is not easy since some nodes are close (in the tree) and some can be located at completely different parts of the tree. The number of neighbors is unpredictable as well, due to the varying sizes of the nodes. This leaves two strategies to finding adjacent cells:
1. Nodes store which cells are adjacent to them and keep this information up to date
2. During the collision detection each node has to find its neighbors

The first strategy requires an additional array stored for each node. This however might save time compared to calculating neighborhood every time step. Since it was not clear how time consuming this part of the whole algorithm was beforehand, both strategies were implemented. Storing adjacent nodes proved to immensely boost the performance and will be used by default throughout the thesis. Nevertheless the discarded strategy will be described.
Keeping track of adjacent nodes can be done with a list of neighbors in each node. Updates to this structure have to happen when the node or one of the adjacent nodes splits, as splitting is the only operation that changes the tree structure (apart from reversing splitting which is an optional

feature). Each split will affect the node itself, its children and adjacent nodes (see Figure 7).
In the approach without the list of adjacent nodes an algorithm has to find all neighbors each

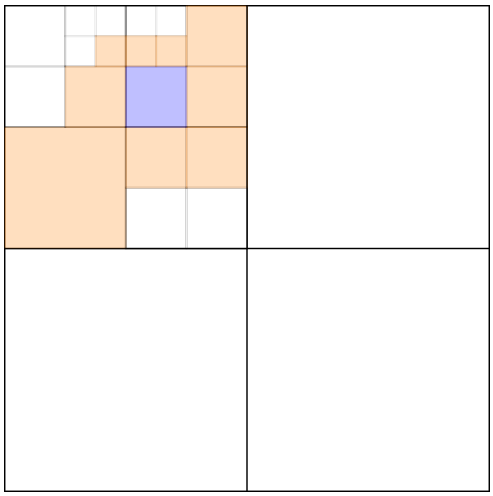

Figure 7: Adjacent nodes (orange) to splitting cell (blue), that need to be updated.

time step. Since neighbors can be almost anywhere in the tree there is no easy solution. One would be to traverse the whole tree and compare bounding boxes with all other nodes, which is rather inefficient considering a growing tree size.
A more feasible approach would be to sample positions near the node and see which nodes are near. In three dimensions twenty-six samples would be needed for all directions. However, a node can have smaller sized neighbors, resulting in possibly more than twenty-six adjacent nodes and the question how to sample. A solution would be to sample as demonstrated in Figure 8 and to descend no further than the depth of the original node. Then, all descendants of sampled nodes in the direction of the original node have to be selected.
Since this procedure turned out to be very slow in comparison to storing adjacent nodes, it will not be explained in detail in section 3.

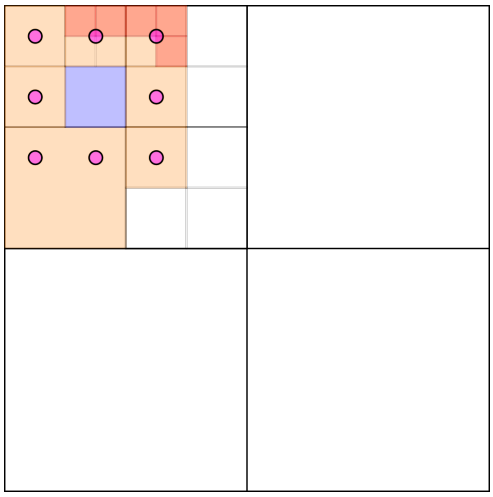

Figure 8: Sample points (pink) are used to find adjacent nodes on same or higher level (orange) as the original node (blue). If they are inner nodes, all their descendants not touching the original node are discarded (darker orange)

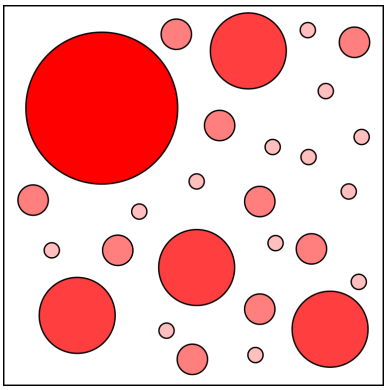

Figure 9: Polydisperse setup

2.5 Polydisperse Scenario

A second configuration which can harm Linked Cells' efficiency is a polydisperse one. This means that there are different kinds of particles, most importantly resulting in differing sizes. Simulations involving sand usually fall into this category, as grains of sand do not have a uniform size. It will become more extreme though if some stones join the sand, where size proportions of more than 100 times can easily be reached.
LC is limited to how small the cell size can get, based on the size of the biggest particle (see Figure 3). So if there is one big stone and millions of sand grains LC has to respect the big stone and will not have success at sorting out collision pairs. In an extreme case not even a single pair is discarded, resulting in $\mathcal{O}(n^2)$ collision checks.

2.6 Hierarchical Hash Grid

To provide a solution to polydisperse setups, the design for a hierarchical hash grid described in [5] has been partially implemented. This structure is will be called *Hierarchical Hash Grid*. While no new concepts were added to this data structure some features were discarded due to different requirements. In this thesis the following concepts have been put into practice.
The basic idea behind a hierarchical grid is the existence of multiple grids with different cell sizes. Each particle will be inserted into one of these grids. Which grid is chosen is determined by the particle size, meaning that big particles are inserted into grids with big cell size. The goal of this approach is to ensure that efficiently sorting out collision pairs is possible for all particles, big and small, according to their needs.
In more detail there has to be a rule which cell sizes are used for grids. They could be set up manually, but in this implementation they are determined automatically. The first particle inserted into the data structure will set the cell size of the first grid. Following grids are designed based on the first grid, with a factor scaling the cell size. This factor is called $f_{hierarchy}$. As follows, grids can have these sizes:

$$c0 \cdot f_{hierarchy}^{x}, x \in \mathbb{Z}$$

with the first cell size

$$c0 = firstParticleSize \cdot \sqrt{f_{hierarchy}}$$

While LC consists of cells that span the whole block space, hash grids consist of a smaller cube of cells which is not bound to any location. These cells are cubic and share the same size. The number of cells is defined when the grid is created and does not change. For example a cube could feature 16x16x16 cells, 4096 in total. This manageable amount of cells is chosen because it allows control of memory usage without memory usage being tied to the cell size.

To reduce the domain to this smaller cube, suitable hash functions are required to assign a particle to a cell (based on the particle's position). For the hash grid to fulfill its purpose, particles that would be in adjacent cells if the grid was global still have to be in adjacent cells after hashing.

Presuming a modulo function that can handle negative values, this design has the aforementioned property:
$h(x) = (\frac{x}{cellsize}) \bmod D$, D being the number of cells in one dimension

To enable fast hash calculation D has to be a power of two.

With hashed particle assignment, particles in different locations of the simulation can be mapped to the same cell, when both particles have the same hash value. Consequently the collision detection might find pairs of particles that are far apart. This is the main disadvantage of implementing a hash grid instead of a regular grid. It also is the reason why the number of cells should be chosen carefully since a small grid might need less memory but is in turn more prone to this problem. Similar to the cell size in LC the right balance has to be found (see section 5).

When a particle is inserted, the first step is to determine the right grid. Which grid is suited is solely dependent on the size of the particle. Each grid is designed to store particles ranging from $\frac{cellsize}{f_{hierarchy}}$ to $cellsize$ in size. If the required grid is not allocated yet, it has to be created before the particle can be inserted.

After the right grid has been chosen, the hash function can be applied to the position and it can be inserted into the correct cell. To conserve time and memory, cells are only allocated when they are needed. So, in case that the cell in question does not exist yet, it is created before inserting.

In the collision detection step all cells need to know who their neighbors are to enable fast algorithms. Each cell could store the specific indices of adjacent cells, but that would greatly increase memory consumption. A better solution is to convert these indices into offsets to the own index (see Figure 10). The advantage of this is that all inner cells of a grid share the same offset pattern. Therefore this pattern has to be stored only once and can be referenced. The outer cells of the hash grid are the only remaining issue. Since the hash grid is modulo based, adjacency-wise there is no edge to the grid. The leftmost cell is adjacent to the rightmost, top to bottom and front to back are also connected. These connections have to be translated into offset patterns. Analyzing the properties of a cube leads to twenty-seven different patterns: Eight corners, twelve edges, six sides and one for inner cells. These all have to be stored in order to be referenced.

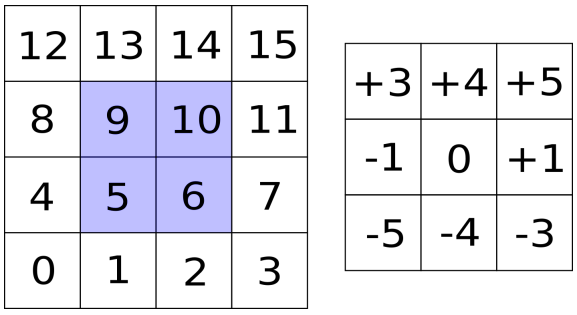

Figure 10: All inner cells (blue) share the same offset pattern (right) for adjacent cells

The order in which they are stored is designed similar to the children indices of the Octree. This time the position in the cube has three states per coordinate: low boundary, in the middle, high boundary. The Indices 0-26 can be represented by 3 ternary system digits. This way a cell's position in the cube can be encoded.
This whole procedure has to be repeated for all numbers of cells in use, as a 16x16x16 cube will have different offsets than a 32x32x32 cube. This is best done when the Hierarchical Hash Grid is built, so once a new grid is built, it can simply reference the required offset patterns needed for its grid size.
With the adjacency being available, the collision pairs can be found efficiently. Determining the particle pairs within the same grid is similar to LC and the Octree. All (occupied) cells check

for intern pairs and pairs with adjacent cells. An issue with finding collision pairs is that each collision only happens once, but can be found by both participating particles. For example (3,4) and (4,3) might be a found, so each acceleration structure has to ensure that no collision will be registered twice. Fortunately there is an elegant solution to this problem. By checking only half of the adjacent cells the pairs found can be cut in half. To ensure that only the duplicate pairs are not found any more no neighbors on opposite ends may both be left out (see Figure 11).
Additionally, all collisions between different grids have to be registered. In order to cut duplicates,

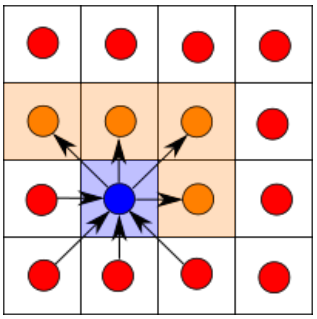

Figure 11: Each cell checks half of the adjacent cells so each pair is found exactly once

the grids are iterated from small to big cell size. When a particle checks for collision in a larger grid, it is virtually inserted there and checks within the resulting cell and all its adjacent cells - this time the pattern mentioned before is not applied. Note that the particle is not actually inserted, it rather boils down to calculating the hash value in the other grid. This way all collisions between grids are registered exactly once, as only small to big is found, not big to small.

3 Implementation

The project was developed in C++. For convenience some code snippets will be presented in pseudo code.
This section will focus on clarifying the layout of used data structures and providing the algorithms necessary to use them efficiently. Central ideas will be highlighted, concepts discussed in the previous section will only be reiterated shortly. The complexity of used methods and final algorithms will be discussed.

The data structures only store IDs of particles. It is assumed that the particles are stored in a different data structure and managing IDs is sufficient and memory friendly.
The data type size_t represents an unsigned integer and real_t a floating point number.

3.1 Octree

The layout of the octree class and the node class can be found in Appendix A.

3.1.1 Insert

To store the particle IDs there needs to be a function that places an ID into the correct node and possibly updates the tree structure. In *WaLBerla Accessor* classes are used to retrieve information about particles like position or velocity, given the ID. Determining the right leaf for the particle ID to be stored in can be achieved by a recursive method, initially called on the root. The only exception is inserting infinite particles since their IDs are stored in the Octree class. This leads to a simple insert method in the Octree class:

```
void insert(size_t id, Accessor& ac) {
   if(ac.isInfiniteSized(id)) { //paraphrased
      infinite_size_particles.push_back(id);
   }
   else {
```

```
      root->insert(id, ac, max_depth, max_ids);
   }
}
```

Nodes need the information about maximum depth and number of IDs per node, as these limits will be used in the insert process. Then the important work is done in the recursive method of the node class. As the maximum number of stacked calls is equal to the maximum depth, recursion is not much of a problem here.
The end of the recursion is quite simple. Once we reach a leaf, meaning the node is not split, the ID is inserted. Finding the way to the right leaf is similar to a search tree: Depending on the position of the particle in relation to the center of the node's bounding box, the right successor is chosen. The method utilizing binary system encoding is used. Finding the right child index is put in a helper method:

```
size_t findOctant(Vec3 p) {
   size_t index = 0;
   for(int i = 0; i < 3; i++) {
      if(p[i] >= bounding_box.center()[i]) {
         switch(i) {
            case 0:
            ret += 1;
            break;
            case 1:
            ret += 2;
            break;
            case 2:
            ret += 4;
            default:{}
         }
      }
   }
   return index;
```

The number of IDs per node should not surpass the cap given. To enforce this, a node splits into eight when insert is called on a fully occupied node. This step however may not increase the depth beyond the maximum depth. Since the maximum number of IDs only affects performance, the limit can be surpassed without severe issues. In contrast the maximum depth is critical to the validity, as it is used to ensure that the cell size of nodes will never become too small. Else the issue in Figure 3 could apply. The addition of the limit handling leads to the final insert method:

```
void insert(size_t id, Accessor& ac, size_t max_depth,
           size_t max_ids_per_node) {
   if(is_split) {
      children[findOctant(id, ac)]->insert(id, ac, max_depth, max_ids_per_node);
   }
   else if(particleIDs.size() < max_ids_per_node){
      particleIDs.push_back(id);
   }
   else if(depth < max_depth){
      split(ac);
      children[findOctant(ac.getPosition(id), ac)]->insert(id, ac, max_depth,
          max_ids_per_frame);
   }
   else {
      particleIDs.push_back(id);
   }
}
```

When a node is split, the eight children are initialized and the IDs are transferred to the children. This is the only moment when the tree changes its structure. If nodes keep track of adjacent nodes,

there has to be a second part to the split function.
After the children are initialized all adjacent nodes have to be notified of their new neighbors. Their adjacency to the currently split node has to be removed. A comparison of bounding boxes can quickly determine whether a child is still adjacent: If there is one dimension where the coordinate intervals do not touch, they are not adjacent. Then all children add their siblings to their adjacency list:

```
void split(Accessor& ac) {
   //The upper part of the function is deemed trivial:

   //Initialize children with fitting bounding boxes

   //Transfer stored particle IDs to children

   is_split = true;

   //Add for adjancency update:

   for(each child c) { //pseudo C++
      for(Frame* a : adj) {
         if(bounding boxes touch) { //paraphrased
            c->adj.push_back(a);
            a->adj.push_back(c);
         }
      }
   }
   //removing self
   for(Frame* a : adj) {
      for(int i = 0; i < a->adj.size(); i++) {
         if(a->adj[i] == this) {
            a->adj[i] = a->adj[a->adj.size()-1];
            a->adj.pop_back(); //removal in constant time
         }
      }
   }
   //child to child
   for(int i = 0; i < 8; i++) {
      for(int j = 0; j < 8; j++) {
         if(i != j) {
            child[i]->adj.push_back(child[j]);
         }
      }
   }
}
```

This method of removing in constant time was mentioned by [5].

3.1.2 Collision Check

The whole tree has to be traversed and every time a leaf is reached, it has to be processed. Iterating through all leaves could also be achieved by keeping a list of all (non-empty) leaves. Maintaining this list would only be impactful when a small IDs per node limit is used. Else empty leaves are pretty rare. Additionally, leaves make up at least 87.5% of all nodes:

$\lim_{x\to+\infty} \frac{8^x}{\sum_{n=0}^{x} 8^n} = 0.875$

In this thesis the octree was traversed with a pointer. In addition to the current node a second value is used to track how many children have already been processed. This results in the following loop layout:

```
Frame* current = root;
size_t next = 0; //equal to number of current's children that were processed already

while(current!=root || next != 8) { //loop stopping at current == root && next == 8
   if(next == 8) { //all children are done
      next = current->index+1;
      current = current->parent;
   }
   else if(current->is_split) { //descending
      current = current->children[next];
      next = 0;
   }
   else { //leaf case
      if(current->particleIDs.empty()) {
         next = current->index+1;
         current = current->parent;
         continue;
      }

      //insert collision detection here

      next = current->index+1;
      current = current->parent;
   }



}
```

These particles could be colliding with particles in the node:

- other particles in the same node
- particles in adjacent nodes
- infinite sized particles

Just like the Hierarchical Hash Grid, the Octree has to deal with the problem of pairs being detected twice. In the grid based designs, the pattern from Figure 11 can be utilized to eliminate this problem while improving the performance. It would be great if the pattern could be used for the Octree as well. Unfortunately, the pattern is valid for uniform grids, while different cell sizes can cause a pair to find found twice (see Figure 12). Then, the pattern cannot be used. Instead, as all collision pairs will be found twice, only the half where the first ID is smaller is accepted. How adjacent nodes can be found if adj is not available is described in section 2, the code can be found in Appendix A. The particle pairs are found like this:

```
for(size_t id : current->particleIDs) {
   //adjacent nodes
   //inefficient method would need to determine adjacent nodes here
   for(Node* n : current->adj) {
      for(size_t id2 : n->particleIDs) {
         if(id<id2) {
            check_collision(id, id2);
         }
      }
   }
   //same node
   for(size_t id2 : current->particleIDs) {
      if(id<id2) {
         check_collision(id, id2);
      }
   }
```

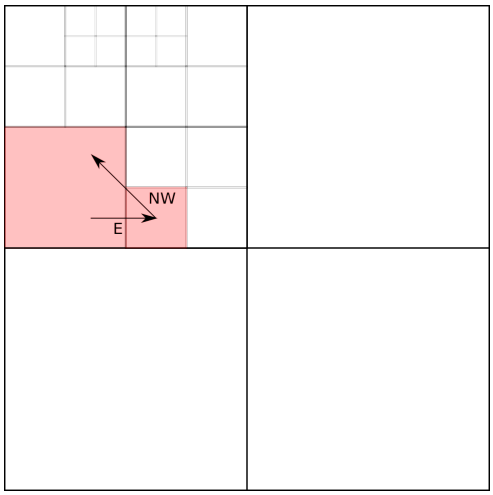

Figure 12: The pattern checking the directions (NW, N, NE, E) fails on the quadtree, although it works on grids. Between the marked cells, collisions will be detected twice

```
}
//Infinite particle collision
for(size_t id : current->particleIDs) {
   for( id2 : infiniteSizeParticles) {
      check_collision(id, id2);
   }
}
```

3.1.3 Clear

Since the strategy to clear the whole structure and to reinsert all particles is chosen in this paper, clearing is part of the routine each time step. Clearing does not aim to completely start from scratch in the next time step as particles are still located in the same area as before. Therefore the old node structure can most likely be reused. Only problem is that small changes add up over time. Over the coarse of hundreds of time steps a dense chunk of particles could have moved from left to right. If old structures are carried over, this now sparsely populated left part of the tree would still be of high depth in contrast to the actual situation.
Two different approaches can be realized to tackle this problem:
• The structure is completely built anew every X time steps.
• There is a cleanup mechanism that reverts splits once a certain threshold is reached.
In this implementation the second option is not used.
Each time step every leaf clears its vector of particle IDs.The same method of traversing as the collision check is used. The vector containing the infinite sized particles is also cleared.

3.2 Hierarchical Hash Grid

The Hierarchical Hash Grid is structured like this:

```
AABB bounding_box;
std::list<Grid*> gridList;
std::vector<size_t> infinite_size_particles;
std::vector<std::vector<offsets>> offsetList;
real_t c0;
real_t fHierarchy;
size_t dMax;
```

offsets is a struct containing an array of 27 integers. Each cell has to know the offsets to its adjacent cells. To save memory and since these offset patterns are repeating, they are stored in the base

class. *offsetList* contains patterns for all possible grid sizes up until the dMax limit. Each entry contains the 27 different patterns possible in a cyclical cuboid.
dMax limits the maximum number of cells used in hash grids. If it has the value 6 this would mean, that a hash grid can at maximum use 2^6 cells per dimension. This enables control of memory policy.

The Grid and HashCell class can be found in Appendix A

3.2.1 Insert

If a particle is not infinite in size, the right grid has to be found. Criterion is the size of the particle. If the right does not exist yet, the insertion algorithm has to make sure it is correctly initiated first. The pseudo code algorithm [5] can be found in Appendix A.

When the grid is determined, the hash value of the particles position has to be evaluated. The basic concept is to calculate $\frac{coordinate}{cellsize} \bmod N$ for each coordinate. These hash values have to be combined to an index in the linearized cell vector. Division and modulo calculation are both quite slow compared to other arithmetic operations. To speed up the hash function the division is expressed as a multiplication with the inverse and due to N being a power of 2 only the last few bits matter. Similar to mod100 cutting off all digits but the last two in the decimal system, mod4 sets all bits to zero, apart from the last two. Because of this *bitwise and* can replace the modulo operation:
$X \bmod N = X \; and \; (N-1), for X >= 0$ [5]
For negative values a small configuration of this idea is necessary [5]. Using these methods and the variables stored in the grid class the hash of a coordinate is calculated like this:

```
if(x < 0) {
   real_t i = (-x) * inv_cell_size;
   x_hash = hash_mask - (static_cast<size_t>(i) & hash_mask);
}
else {
   real_t i = x * inv_cell_size;
   x_hash = static_cast<size_t>(i) & hash_mask;
}
```

To get a useful index the hash value are weighted with powers of N:
$cellindex = xHash + yHash \cdot N + zHash \cdot N^2$
In case that the cell at the calculated index is not initialized yet, it has to be created before insertion is possible. When a cell is constructed, it needs to know which offset pattern is the correct one to reference. Due to the use of ternary system encoding of the directions the right index can be found like this:

```
//Input: size_t ind
size_t offset = 13;
if((ind & (N-1)) == 0) {
   offset -= 1;
}
else if((ind & (N-1)) == N-1) {
   offset += 1;
}
if((ind & (N*N-1)) < N) {
   offset -= 3;
}
else if((ind & (N*N-1)) >= N*(N-1)) {
   offset += 3;
}
if(ind < N*N) {
   offset -= 9;
}
else if(ind >= N*N*(N-1)) {
   offset += 9;
}
```

Finally the newly created cell has to be added to the list of occupied cells and inserted into the cell. To avoid too frequent reallocation, cells follow this allocation policy:

```
void store(size_t id) {
   if(particle_ids->size() == particle_ids->capacity()) {
      particle_ids->reserve(2*particle_ids->size());
   }
   particle_ids->push_back(id);
}
```

3.2.2 Collision Check

The following checks have to be realized for each particle:
Collision with ...

- infinite particles
- particles in the same cell
- particles in adjacent cells
- adjacent particles in higher grids

The check in higher grids requires an additional calculation of the right offset pattern if the cell is not initialized yet. This could be fixed by initializing every cell upon grid construction, resulting in significantly more memory usage.
The code was put in the Appendix A again.

3.2.3 Clear

Each time step all occupied cells remove their particle IDs and the list of occupied cells itself is cleared. In addition, the vector of infinite particles is reset as well. After many time steps the amount of unused, allocated cells might increase, taking up memory. If this is an issue a total reset every some time steps is recommended.

4 Benchmarks

When a simulation is set up, it is crucial to know which data structure should be used for optimal performance. To collect data in which cases Linked Cells, the Octree or the Hierarchical Hash Grid should be used, benchmarks can be used to compare performance. These benchmarks are scenarios that are simulated multiple times, each time with a different data structure, but with identical starting conditions. Differences in performance can then be traced back to the data structures, as they are the only difference in the setup. This way comparisons between data structures can be quantified. Important metrics are the total runtime of the benchmark, the time spent on finding collision pairs, the number of collision pairs found and the memory usage.
For a valid analysis it is required that measurements as precise as possible. A simulation on a normal computer would render the measurements vulnerable to side effects, the operating system could for example prioritize a different task - causing a delay. Solution for this issue is to use a computer specifically designated to running benchmarks, with control over the hardware and full focus on the benchmark tasks. Even this computer is not free from side effects, but the risk is minimized. All benchmarks in this thesis were performed with the benchmark machine of computer science chair 10, Friedrich-Alexander-Universität Erlangen-Nürnberg.
Hardware details:

- CPU name: Intel(R) Xeon(R) Gold 5122 CPU @ 3.60GHz
- CPU type: Intel Skylake SP processor
- CPU stepping: 4

- L1 Cache: 32 kB
- L2 Cache: 1 MB
- L3 Cache: 16 MB

The CPU frequency was set to 3.30 GHz for all benchmarks.
In this section the comparison between Octree, the Hierarchical Hash Grid and their main competitor Linked Cells is evaluated. First off there will be a homogeneous, monodisperse scenario to have a reference for scenarios with special properties. This first benchmark will be used to identify how changes in the scenario setup affect performance. The Octree is expected to perform well in inhomogeneous scenarios, so different levels of homogeneity are tested to see whether the Octree lives up to its expectations. Similarly, the Hierarchical Hash Grid should be favored in polydisperse scenarios, so there will also be tests with varying dispersity.
Goal of these benchmarks is to find out which acceleration structures perform best for given types of simulation setups, but also to provide insights on how changes in the setup translate into changes in performance. Once the comparative benchmarks are analyzed, the complexity of Octree and the Hierarchical Hash Grid will be examined, to see how well the data structures react to a growing amount of particles in a simulation. A benchmark setup will be run multiple times, but each time with more particles. From the results, conclusions about the complexity of runtime and collision pairs are deduced.

4.1 Various Scenarios

4.1.1 General Setup

Simulation setups all have the same structure. The simulation space is a cuboid, all regular particles have to be inside of this space. If a particle leaves the simulation space it is no longer relevant for the simulation. Before any simulation is happening, the starting conditions for the simulation are specified. The amount of particles defines how many particles are created. Each particle needs a shape, a starting location and a starting velocity. In this thesis, the location and velocity are usually randomized and the shape is always a sphere. The velocity is set to very low values to keep the state of a simulation relatively static. This results in a small amount of actual collisions. Few collisions are accepteable, as the scenarios are designed to demonstrate the coarse collision capabilities of acceleration structures and for that no collisions are required. The sphere is chosen because in coarse collision detection the only relevant information about the particle is the bounding radius. With randomized locations, particles could share the same spot. This can pose a problem to collision models, as usually particles would not overlap that much. To avoid effects like uncontrollable forces due to overlapping, a simple algorithm is used: Each particle is checked against all other particles upon creation. As this leads to $\mathcal{O}(n^2)$ checks, the strategy limits the amount of particles. If more are needed, the setup needs to be specified directly or overlapping has to be tolerated.
In order to keep all particles inside the simulation space, walls are put around the cuboid, modeled as infinite sized particles. To complete the physical information, the number of time steps is defined. In the following benchmarks, time steps primarily serve the purpose to measure reliable data, the physics are secondary.
While the benchmarks in this thesis don't represent real world events, physical correctness of a simulation is still guaranteed. Goal is to provide an analysis about how changes in the setup translate into performance so that conclusions can be drawn and applied to meaningful simulations.
The criteria above define which scenario is simulated. However, it is not only relevant what is simulated, but also how it is simulated. The second part of settings describes which acceleration structures were used and how they were adjusted.
In all comparative benchmarks the time optimal settings for each acceleration structure are used. The settings are optimized for runtime performance since optimizing for collision pairs is not constructive. Both grid based acceleration structures can trade off memory and runtime efficiency for less collision pairs. As optimizing for collision pairs would lead to horribly inefficient test results, runtime was deemed a better criterion for optimization. To achieve the best time performance the right balance between overhead and detection efficiency has to be found. Each benchmark was run with different settings, but only the runs with the best settings are presented.

Note on the use of $f_{hierarchy}$:
It is a setting option that could be optimized. $f_{hierarchy}$ can be used to manipulate the size of the grid cells, but this should not be common practice. The setting should be used for determining how many hash grids should be used, not to manipulate the performance of one grid. $f_{hierarchy}$ could be used in the polydisperse benchmarks to optimize the performance of multiple grids, but it was deemed more educational use a fixed value. Thus, $f_{hierarchy}$ is set to 2 for all benchmarks. More details about the optimal use of the feature can read in [5].

4.1.2 Reference Benchmark

While one benchmark run can show which structure performed best, two different benchmarks are required to gather information on how the setup affects the results. To analyze the effects on inhomogeneity and polydispersity, comparison with a homogeneous and a monodisperse benchmark is crucial. This benchmark aims at providing this context.
Linked Cells is expected to perform well, as none of its weaknesses are present in the setup. The acceleration structures will showcase how efficient they are in general.

Benchmark setup:

- Simulation space: 100x100x100
- Particles: 100000 spheres with radius 0.5
- Distribution: no bias (homogeneous)
- Time steps: 200
- Structures in use: LC, Octree, Hierarchical Hash Grid
- LC cell size: 1.3
- Octree max size: 11
- Octree max depth: 6
- Hash Grid size: 128x128x128

Results: see Table 1

Table 1: Results of the reference benchmark

Structure	**Time Total in s**	**Collision Time in s**	**Collision Pairs**
Linked Cells	5.24	4.42	1.75e8
Octree	20.8	17.9	9.04e8
Hierarchical Hash Grid	17.1	12.5	1.92e8

As expected, Linked Cells proves to be significantly faster than its competitors, the Octree being four times slower and the Hierarchical Hash Grid being more than three times slower. The collision pair efficiency of Linked Cells and the Hierarchical Hash Grid is similar.

4.1.3 Inhomogeneous Benchmark

To make a simulation inhomogeneous, the particles are not initiated at completely random positions, but in random positions within a segment of the simulation space. Particles were distributed to 3 disjoint areas, each with a different size and density. The small amount of remaining particles was distributed globally. There was no incentive to design this scenario exactly like this, the goal was to create inhomogeneity in some non-radical way.
Over time inhomogeneity will always dissolve. The longer a simulation lasts and the more velocity the particles have from the start, the more homogeneous the state of the simulation will become.

That is why a near zero velocity is used, ensuring that inhomogeneity persists throughout the simulation.
Although the Hierarchical Hash Grid was not designed to tackle inhomogeneous setups, it is still included in the benchmark because compared to Linked Cells, a hash grid has significant advantages when handling empty space.

Benchmark setup:

- Simulation space: 500x500x500
- Particles: 100000 spheres with radius 0.5
- Distribution:
  70% in 150x150x100 area, from (0,200,250) to (150,250,350)
  25% in 75x75x75 area, from (325, 150, 100) to (400, 225, 175)
  4% in 25x25x25 area, from (475,475,475) to (500,500,500)
  1% no bias
- Time steps: 200
- Structures in use: LC, Octree, Hierarchical Hash Grid
- LC cell size: 3.75
- Octree max size: 11
- Octree max depth: 8
- Hash Grid size: 128x128x128

Results: see Table 2

Table 2: Results of the inhomogeneous benchmark

Structure	**Time Total in s**	**Collision Time in s**	**Collision Pairs**
Linked Cells	15.9	13.3	7.09e8
Octree	20.6	17.3	8.40e8
Hierarchical Hash Grid	9.42	6.50	1.59e8

Even though the Hierarchical Hash Grid was not intended for this purpose, it clearly is the fastest of the three acceleration structures. The Hierarchical Hash Grid finds more than four times less pairs than its competitors. The Octree is still slower than Linked Cells.
The Hierarchical Hash Grid's ability to use a small cell size without drawbacks proves to be very useful. While Linked Cells has to manage lots of cell in sparse areas, the Hierarchical Hash Grid only considers the ones with at least one particle. Compared to the reference benchmark, Linked Cells lost its advantage over the other structures and was outperformed by the Hierarchical Hash Grid.

4.1.4 Very Inhomogeneous Benchmark

Compared to the previous benchmark, the simulation space was increased even further. The additional space will slow down Linked Cells even more. The areas with particles were set up in different locations and this time they overlap. First, an area in a corner is filled with particles and then an extended area around the corner is filled. Remaining particles are spread globally. This setup creates a dense area in one corner, even denser close to the corner, while most of the simulation remain empty.

Benchmark setup:

- Simulation space: 2000x2000x2000
- Particles: 80000 spheres with radius 0.5
- Distribution:
  10% in 40x40x40 area, from (0,0,0) to (40,40,40)
  85% in 100x100x100 area, from (0,0,0) to (100,100,100)
  5% no bias
- Time steps: 200
- Structures in use: LC, Octree, Hierarchical Hash Grid
- LC cell size: 7.0
- Octree max size: 11
- Octree max depth: 10
- Hash Grid size: 128x128x128

Results: see Table 3

Table 3: Results of the very inhomogeneous benchmark

Structure	**Time Total in s**	**Collision Time in s**	**Collision Pairs**
Linked Cells	93.0	69.1	5.06e9
Octree	19.6	17.3	9.72e8
Hierarchical Hash Grid	7.50	5.60	1.42e8

Again, the Hierarchical Hash Grid performs very well, but the Octree also shows strong improvement compared to Linked Cells. Even if the Octree cannot compete with the performance of the Hierarchical Hash Grid, its goal was reached.

4.1.5 Checkerboard Benchmark

This benchmark uses a checkerboard pattern to fill the simulation space. By doing this, half the simulation space will be covered by particles and half of it is empty. This setup is inhomogeneous, but the amount of empty space is very low compared to the other inhomogeneous benchmarks. This means that Linked Cells should be able to perform well although the setup is inhomogeneous. The structure of the checkerboard aligns with the structure of the Octree, which should result in a better performance compared to, say, a 3x3x3 board that does not align.
Due to the many disjoint areas, the randomized particle creation can be sped up enormously. A particle location does not need to be checked against all other particles, but only against those in the same area. To make sure this does not enable overlapping particles, particles cannot reach out of the boundaries of an area. A higher amount of particles than usual is used and in turn, fewer time steps were simulated.

Benchmark setup:

- Simulation space: 400x400x400
- Particles: 1000000 spheres with radius 0.5
- Distribution: checkerboard pattern, 4x4x4 "board" with 32 100x100x100 areas containing 31250 particles each
- Time steps: 50
- Structures in use: LC, Octree, Hierarchical Hash Grid

- LC cell size: 2.1
- Octree max size: 11
- Octree max depth: 8
- Hash Grid size: 256x256x256

Results: see Table 4

Table 4: Results of the checkerboard benchmark

Structure	Time Total in s	Collision Time in s	Collision Pairs
Linked Cells	19.8	14.8	4.87e8
Octree	73.6	64.9	4.12e9
Hierarchical Hash Grid	30.2	16.6	3.71e8

Although the structure of the board favors the Octree, the grid based structures are superior. Compared to the homogeneous benchmark, the Octree even performs worse in relation to its competitors.

4.1.6 Polydisperse Benchmark

Polydisperse simulations were than original issue why the Hierarchical Hash Grid was designed. The polydisperse scenarios are homogeneously distributed, so performance changes can be traced back to the polydispersity. In this scenario four different particle sizes are used, with the biggest particle being eight times bigger than the smallest. In polydisperse benchmarks, Linked Cells issue is that its cell size is tied to the size of the biggest particle. It cannot be less than the particle's diameter.
The Octree was not included in polydisperse benchmarks as there is no indicator that it would have gained any advantage over the grid based structures and it was already slower in the reference benchmark.
Benchmark setup:

- Simulation space: 200x200x200
- Particles:
  12500 spheres with radius 2 12500 spheres with radius 1 12500 spheres with radius 0.5 12500 spheres with radius 0.25
- Distribution: no bias (homogeneous)
- Time steps: 200
- Structures in use: LC, Hierarchical Hash Grid
- LC cell size: 4
- Hash Grid size: 128x128x128

Results: see Table 5

Table 5: Results of the polydisperse benchmark

Structure	Time Total in s	Collision Time in s	Collision Pairs
Linked Cells	2.87	2.64	1.12e8
Hierarchical Hash Grid	12.6	8.99	6.54e7

Linked Cells performs a lot better time wise, while the Hierarchical Hash Grid finds less collision pairs. Compared to the runtime ratio in the reference benchmark, the Hierarchical Hash Grid's runtime even got worse. Managing multiple grids requires additional effort, which might not pay off when the difference in particle size is small and the hash grids are not very filled.

4.1.7 More Polydisperse Benchmark

Compared to the previous benchmark, the discrepancy in size between biggest and smallest particles was increased to factor 40. The number of particles in each size group was also adjusted so the bigger the particle size, the fewer particles are created.

Benchmark setup:

- Simulation space: 400x400x400
- Particles:
  320 spheres with radius 20 2880 spheres with radius 2.5 12800 spheres with radius 1 64000 spheres with radius 0.5
- Distribution: no bias (homogeneous)
- Time steps: 200
- Structures in use: LC, Hierarchical Hash Grid
- LC cell size: 40
- Hash Grid size: 128x128x128

Results: see Table 6

Table 6: Results of the more polydisperse benchmark

Structure	**Time Total in s**	**Collision Time in s**	**Collision Pairs**
Linked Cells	166.0	165.8	1.30e10
Hierarchical Hash Grid	24.0	18.7	4.17e8

When the difference in particle size becomes this big, Linked Cells cannot keep up any more. Its cell size is too big to be efficient at handling all the small particles. The Hierarchical Hash Grid now shines with about seven times faster runtime and about thirty times less collision pairs.

4.1.8 Very Polydisperse Benchmark

This benchmark doesn't reveal anything new but demonstrates why certain simulations were not feasible with the use of Linked Cells. One large particle that almost reaches the whole simulation space is surrounded by small particles. This scenario is similar to sand interacting with a big rock.

Benchmark setup:

- Simulation space: 400x400x400
- Particles:
  1 sphere with radius 180
  39999 spheres with radius 0.5
- Distribution: no bias (homogeneous)
- Time steps: 10
- Structures in use: LC, Hierarchical Hash Grid

- LC cell size: 360
- Hash Grid size: 128x128x128

Results: see Table 7

Table 7: Results of the very polydisperse benchmark

Structure	Time Total in s	Collision Time in s	Collision Pairs
Linked Cells	104.4	104.4	8.00e9
Hierarchical Hash Grid	0.154	0.0965	3.01e6

As shown by the number of collision pairs, Linked Cells loses all of its purpose and is as efficient as a brute-force approach. In fact, straight up brute-force would even be faster. With use of the Hierarchical Hash Grid, this scenario does not pose a problem at all.

4.2 Performance Scaling

4.2.1 Complexity Expectation

Based on the methods implemented, theoretical expectations about the performance can be estimated. Below each expectation a small explanation will be given. The complexity is depends on the number of particles n.

Octree:

- **Single insertion**: $\mathcal{O}(log(n))$
  The insertion is called recursively until a leaf is found. The depth of the tree is $\mathcal{O}(log(n))$.
- **Collision check**: $\mathcal{O}(n)$
  As the amount of IDs per node is limited, the average amount of neighbors will not increase if the amount of particles increases. The amount of nodes grows linearly, resulting in an overall linear performance.
- **Clearing**: $\mathcal{O}(n)$
  For each clear the whole tree is traversed, resulting in a linear complexity.
- **Memory consumption**: $\mathcal{O}(n)$
  The number of nodes grows linearly and the amount of particles does too.
- **Collision pairs**: $\mathcal{O}(n)$
  Similar reasoning as for the collision check.

Hierarchical Hash Grid:

- **Single insertion**: $\mathcal{O}(1)$
  No dependency.
- **Collision check**: $\mathcal{O}(n)$, turning to $\mathcal{O}(n^2)$ the more a hash grid is filled All occupied cells are iterated and each cell checks for adjacent neighbors The amount of occupied cells is named C, and the average amount of IDs per cell A:
  Runtime $= \mathcal{O}(C + n \cdot A)$
  C will grow linearly until the hash grid becomes filled. A will also grow linearly, but starts with a value below 1. With a low saturation C will be more important, while $n * A$ will be the only relevant factor once the grid is saturated.
- **Clearing**: $\mathcal{O}(n)$
  All occupied cells are cleared. In a saturated grid this could even be regarded as $\mathcal{O}(1)$
- **Memory consumption**: $\mathcal{O}(n)$
  The amount of cell stored grows linearly until saturation, the amount of IDs stays linear.

- **Collision pairs**: $\mathcal{O}(n^2)$
  Simply $\mathcal{O}(n * A)$

4.2.2 Test Setup

Simulations with large amounts of particles can take lots of effort to compute. In these large scale simulations estimations for memory consumption or the time needed to finish the computation are vital. If it is not carefully planned, the simulation might not finish in time or lack of memory shuts the simulation down. To be able to predict behavior in large scale simulations, models are needed. For example big O notation is a common model for expressing how an increase in demands will translate into performance. To accurately predict runtime performance for a given scenario, complex formulas would be needed as there are many factors that define a simulation. As a goal, this would be aiming too high, instead this thesis will suffice with providing simple models based on big O notation.
For this analysis one benchmark will be run multiple times, where the setup stays the same, except for the amount of particles, which is doubled each run. If doubling the amount of particles results in double the runtime, a perfect $\mathcal{O}(n)$ relation between particles and runtime would be shown. To be able to attribute the change in runtime to the change in particle numbers, all other factors influencing the runtime have to stay the same for all benchmark runs. This is why the settings for the data structure do not change throughout the runs, even though an increase in performance could be achieved.
The benchmark setup needs to be able to consist of millions of particles. In previous benchmarks the strategy to forbid overlapping particles was chosen. This is an issue as the routine for checking all other particles upon creation of a particle leads to $\mathcal{O}(n^2)$ checks. As this is not feasible with this millions of particles, overlapping particles were allowed. This means that the test does not simulate a real physical application, but this is not a problem since only the collision detection is of importance. For performance evaluation no physics need to be applied, as it is not task of the data structures. Without application of physics, no change to the state of the simulation happens between time steps. Time steps are only used to average out the time needed to allocate the whole data structure in the first time step.
Other benchmarks used walls modeled as infinite particles to hinder particles from leaving the simulation. As particles are not moving, no walls are necessary. An infinite particle would interact with every particle once, causing a linear offset to the data gathered. This would complicate evaluation of the measurements and therefore infinite particles are not part of this benchmark. The setup of the benchmark is as follows:

- 10000-5120000 Particles, radius 0.5
- 1024x1024x1024 simulation space, no walls
- 10 time steps, but no physics
- Octree max depth: 10
- Octree max size: 10
- Hash Grid size: 128x128x128

4.2.3 Octree Results

Table 8 shows the data gathered by the benchmark runs. The numbers in parenthesis describe by which factor the value has changed, compared to the run with half the amount of particles. On average, these factors are slightly greater than 2, but the factors underlie a high variance. This is especially apparent in the column for the collision pairs. Even though the values are varying this much a more stable pattern can be found. If the values are compared to the run with eight times less particles more consistent factors are found. The pattern exists due to the splitting conditions of the Octree. A node will split once a certain threshold is reached, in this case upon trying to insert the eleventh particle ID. Say, the average number of IDs is currently 4. Due to the homogeneous distribution in this benchmark most nodes will contain about 4 IDs. If the amount of particles is

Table 8: Octree scaling test data: The numbers in parenthesis indicate the change compared to previous run. In box brackets the change to the run with 8 times less particles is indicated

Particles	Time Total in s	Collision Time in s	Pairs
1.0e4	0.0692 (—) [—]	0.0583 (—) [—]	2.95e6 (—) [—]
2.0e4	0.188 (2.71) [—]	0.1711 (2.94) [—]	1.12e7 (3.78) [—]
4.0e4	0.391 (2.08) [—]	0.336 (1.96) [—]	2.00e7 (1.79) [—]
8.0e4	0.692 (1.77) [10.00]	0.541 (1.61) [9.28]	2.51e7 (1.26) [8.50]
1.6e5	1.88 (2.72) [10.03]	1.61 (2.97) [9.38]	9.53e7 (3.80) [8.54]
3.2e5	4.32 (2.30) [11.06]	3.32 (2.07) [9.89]	1.66e8 (1.74) [8.32]
6.4e5	8.24 (1.91) [11.90]	5.46 (1.65) [10.11]	2.08e8 (1.26) [8.30]
1.28e6	19.8 (2.40) [10.49]	14.7 (2.69) [9.13]	7.87e8 (3.78) [8.26]
2.56e6	44.3 (2.24) [10.25]	29.1 (1.98) [8.77]	1.37e9 (1.75) [8.28]
5.12e6	86.5 (1.95) [10.49]	48.9 (1.68) [8.95]	1.69e9 (1.23) [8.12]

doubled, most nodes will have about 8 IDs. Even though the amount of particles was doubled, the structure did not change much as few split were triggered. Doubling the amount of particles once more will result in almost every node splitting. This behavior explains the varying factors and why eight times more particles result in a stable pattern. Eight times more particles will on average trigger one split for each node and the average IDs per node will be kept.
Judging by the factors in box brackets, still Table 8, eight times more particles will result in roughly 10.5 times more computation time. The Octree does not show perfect linear complexity due to the overhead that comes from the growing size of the tree structure. With higher depth, insertion and traversing the tree need more effort. Thus, it is reasonable that the Octree shows superlinear time complexity. In contrast, the amount of collision pairs looks better regarding linear complexity. While eight times more particles result in an increase greater than 8 at first, it appears to be converging against 8 the more particles are present. The efficiency of grid based coarse collision detection is lower when the grid consists of few cells. An increase in tree depth, resulting in more nodes, causes the filtering to be more efficient. Table 9 shows the number of cell to cell checks in relevant grid sizes. For 10000 particles and 10 IDs allowed per node, an 8x8x8 is not big enough, so

Table 9: Amount of neighbor relations between cells. Purpose is to show how many cell to cell collision checks are happening. As particles can collide with particles in their own cell, each cell counts as its own neighbor.

Grid Size	Adjacency relations	Increase factor
2x2x2	64	-
4x4x4	1000	15.6
8x8x8	10648	10.6
16x16x16	97336	9.14
32x32x32	830584	8.53
64x64x64	6859000	8.26
128x128x128	55742968	8.13

we can assume that the simulation started with an average depth of 4, resulting in a 16x16x16 grid. The factors in Table 9 align with the measurements in Table 8. Thus, linear scaling of the collision pairs found can be expected. Even though the number of collision pairs shows linear complexity, the collision detection step appears to have superlinear time complexity.
In conclusion the structure shows functionality of linear complexity, but due to overhead of a growing structure, the execution time is superlinear. Based on the change from first run to the last run, a long term trend is estimated:
$86.5/0.0692 = 1250$
long term change per run:
$\sqrt[9]{1250} \approx 2.209$
transformation to exponent:

$log_2(2.209) \approx 1.143$
The complexity for collision detection is calculated the same way:
$48.9/0.0583 \approx 839$
$\sqrt[9]{839} \approx 2.11$
$log_2(2.11) \approx 1.079$
In the calculations no rounded values were used.
Total time complexity: $\mathcal{O}(n^{1.143})$
Collision detection complexity: $\mathcal{O}(n^{1.079})$
Collision pairs complexity: $\mathcal{O}(n)$

4.2.4 Hierarchical Hash Grid Results

Table 10: Hierarchical Hash Grid scaling test data: The numbers in parenthesis indicate the change compared to previous run

Particles	Time Total (s)	Collision Time (s)	Pairs
1.0e4	0.0262 (—)	0.00757 (—)	6.33e3 (—)
2.0e4	0.0472 (1.80)	0.0173 (2.28)	2.55e4 (4.02)
4.0e4	0.106 (2.23)	0.0488 (2.83)	1.05e5 (4.14)
8.0e4	0.286 (2.71)	0.150 (3.08)	4.19e5 (3.98)
1.6e5	0.802 (2.81)	0.479 (3.19)	1.69e6 (4.04)
3.2e5	2.15 (2.68)	1.47 (3.06)	6.72e6 (3.98)
6.4e5	6.07 (2.83)	4.60 (3.14)	2.69e7 (4.00)
1.28e6	17.6 (2.90)	14.4 (3.13)	1.08e8 (4.00)
2.56e6	47.0 (2.67)	40.3 (2.79)	4.31e8 (4.00)
5.12e6	115.1 (2.45)	100.0 (2.49)	1.72e9 (4.00)

In the data gathered in Table 11 clear behaviors can be observed. The amount of collision pairs quadrupling every run. Apart from the deviation in line 6, the factors of both time measurements increase up to a peak and decrease over the last two runs. The peak can can explained with the saturation of the hash grid. The more cells are already allocated, the more likely it is for an insertion to hit an existing cell. This means that insertions will become faster the more the grid is filled. In the collision detection step all occupied cells are iterated. With low saturation of the grid twice the amount of particles lead to twice the amount of occupied cells. At some point, the amount of occupied cells hits its limit. In this benchmark a grid of size 128x128x128 is used, containing 2097152 cells. The last two runs happen on a hash grid with high saturation, explaining the drop in factor growth.
This decrease is only temporary though, as it only means a reduction in overhead. As observed by the collision pairs, the grid starts with a very low amount of pairs, but becomes more and more inefficient the more it is filled. So as the number of particles grows even more, the linear overhead of inserting all particles and iterating over all cells of the grid will become dominated by the $\mathcal{O}(n^2)$ increase in collision pairs found.
In conclusion the Hierarchical Hash Grid performs with $\mathcal{O}(n)$ complexity when the hash grid is not filled, but shifts more and more towards $\mathcal{O}(n^2)$ the more the grid is filled. To counteract to the quadratic complexity, a sufficient grid size has to be chosen.

4.2.5 Memory Test

To roughly measure the memory consumption a separate test was created. The test does not include a simulation as inserting particles to created a filled data structure was deemed sufficient. The measurements are very inaccurate since memory needed for the particles had to be included in the calculation and the memory display rounded the values. Nevertheless, general trends could still be estimated. Two different versions of the Hierarchical Hash Grid were measured to estimate the how much more memory a hash grid of higher size needs for the same amount of particles.

Test setup:

- 400x400x400 simulation space

- 500000-8000000 particles, radius 0.5
- Octree max depth: 8
- Octree max size: 11
- Hierarchical Hash Grid grid size: 128x128x128 / 256x256x256

Table 11: Memory test data in MiB, HHG stands for Hierarchical Hash Grid and the attached number indicates the grid size used. " " in front of a value denotes that it was rounded

Particles	**HHG7**	**HHG8**	**Octree**
$5.0e5$	35	144	80
$1,0e6$	59	200	120
$2,0e6$	100	300	200
$4.0e6$	200	500	900
$8.0e6$	300	1000	1200

The saturation of the HGG7 can be observed as the rate of growth slows down around the 2000000 particles mark. The HHG8 uses about 3-4 times more memory than the HHG. The octree does have its usual jumps in the factors.
Overall for all structures a roughly linear trend can be seen. As this matches with the expectations, it seems likely. Combined with the fact that the memory needed to store the particles was at least 5 times, the size of the structures should not become an issue.

5 Conclusion

Both acceleration structures managed to achieve their respective goals. In inhomogeneous scenarios the octree based structure performed very well compared to its competitor Linked Cells and proved to be a well suited alternative. The Hierarchical Hash Grid managed to simulate polydisperse scenarios without issues. Scenarios involving many small particles and one particle spanning the whole simulation space were the biggest problem of Linked Cells as it practically resulted in the brute-force pairing of all particles. With the Hierarchical Hash Grid these simulations can be treated like any other. In addition to the Hierarchical Hash Grid's excellent performance in the polydisperse setups, the acceleration structure even turned out to be significantly outperforming the Octree in inhomogenous scenarios. As a result the Hierarchical Hash Grid complements Linked Cells very well, as Linked Cells still performs better when confronted with plain scenarios. In practice, the Octree will likely be the second option regardless of the setup. With the combination of Linked Cells and the Hierarchical Hash Grid almost any scenario can be simulated efficiently, although there is still no solution for huge, elongated objects, the main problem of the Hierarchical Hash Grid [5].

Although the results are sufficient the data structures still leave room for improvement. This thesis did not focus on an efficient update step. By implementing a more efficient update routine, the performance of the Hierarchical Hash Grid can probably be improved slightly. The design implementation of other acceleration structures to further complement the set of structures would also be of interest. Maybe even better choices than the approaches in this thesis can can found. More detailed research into determining the right choice of acceleration structure would also be useful, as beforehand, it is not always trivial which acceleration structure will perform best, given a setup.

References

[1] C. Ericson. *Real-time collision detection.* CRC Press, 2004.

[2] C. Godenschwager, F. Schornbaum, M. Bauer, H. Köstler, and U. Rüde. A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In *Proceedings of the*

International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–12, 2013.

[3] N. M. Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012.

[4] D. P. Mehta and S. Sahni. *Handbook of data structures and applications*. Taylor & Francis, 2018.

[5] F. Schornbaum. Hierarchical hash grids for coarse collision detection. *Student Thesis, University of Erlangen-Nuremberg*, 2009.

[6] U. Welling and G. Germano. Efficiency of linked cell algorithms. *Computer Physics Communications*, 182(3):611–615, 2011.

A Additional Octree Code

The Octree is structured like this:

```
Node* root;
std::vector<size_t> infinite_size_particles; //Used for features like walls
size_t max_ids; // maximum amount of IDs a node can contain before splitting
size_t max_depth; // control of depth, necessary so cells can't become too small
AABB bounding_box;
```

The basic element of the octree is structured like this:

```
//class Node
AABB bounding_box;
bool is_split;
size_t depth;
size_t index; //child position in parent
Node* parent;
std::vector<Node*> children;
std::vector<Node*> adj; //adjacent nodes (optional)
std::vector<size_t> particleIDs;
```

This helper method was used for the search of neighbors:
x, y, z contain -1, 0 or 1, denoting the relative position of the searching node
depending on each value, four of the children might be discarded.

```
void addNodes(std::set<Node*>& adjSet, int x, int y, int z) {
   if(!is_split) {
      adjSet.insert(this);
   }
   else {
      std::vector<bool> skip;
      skip.assign(8,false);

      if(x == -1) {
         skip[0] = true;
         skip[2] = true;
         skip[4] = true;
         skip[6] = true;
      }
      else if(x == 1) {
         skip[1] = true;
         skip[3] = true;
         skip[5] = true;
         skip[7] = true;
      }
      if(y == -1) {
         skip[0] = true;
         skip[1] = true;
         skip[4] = true;
         skip[5] = true;
      }
      else if(y == 1) {
         skip[2] = true;
         skip[3] = true;
         skip[6] = true;
         skip[7] = true;
      }
      if(z == -1) {
         skip[0] = true;
         skip[1] = true;
         skip[2] = true;
```

```
            skip[3] = true;
        }
        else if(z == 1) {
            skip[4] = true;
            skip[5] = true;
            skip[6] = true;
            skip[7] = true;
        }

        for(size_t i = 0; i < 8; i++) {
            if(!skip[i]) {
                children[i]->addNodes(adjSet, x, y, z);
            }
        }
    }
}
```

This routine has to be called each time adj has to be calculated:

```
std::set<Node*> adj;

for(int x = -1; x < 2; x++) {
    for(int y = -1; y < 2; y++) {
        for(int z = -1; z < 2; z++) {
            if(x == 0 && y == 0 && z == 0) continue;

            Vec3 search = current->getBBox().center();
            search[0] += x*current->getBBox().xSize();
            search[1] += y*current->getBBox().ySize();
            search[2] += z*current->getBBox().zSize();

            Frame* s = findPoint(search, current->getDepth());
            //searches a node, but not deeper than the second argument, nullptr if out of
                bounds
            if(s != nullptr) {
                s->addNodes(adj, x, y, z);
            }
        }
    }
}
```

A Additional Hierarchical Hash Grid Code

A hash grid is structured like this:

```
std::vector<offsets>* offsetPtr;
std::vector<HashCell**> occupied_cells; // cells with at least one entry
std::vector<HashCell*> cells;
real_t cell_size;
real_t inv_cell_size; // 1/cell size, used for hashing
size_t hash_mask; //D-1, also used for hashing
size_t N; // denotes grid size
size_t nLog; // log2 of N
```

The HashCell is structured like this:

```
std::vector<size_t>* particle_ids;
offsets* offset;
```

Pseudo code algorithm determining the right grid and initiating new grids if necessary:

```
if(no hash grid exists) {
   create a new hash grid with
   cell_size = particle_size * sqrt(fHierarchy)
   and add it to the grid list

   insert particle into this grid
}
else {
   x = 0
   for(Grid G with smallest cells to grid with largest cells) {
      x = cell_size of G
      if(particle_size < x) {
         x = x / fHierarchy
         if(particle_size < x) {
            while(particle_size < x) {
               x = x / fHierarchy
            }
            create a new hash grid with
            cell_size = particle_size * sqrt(fHierarchy)
            and add it directly before G to the list
            insert particle into the new grid
         }
      }
      else {
         insert particle into existing grid G
      }
      as the particle has been inserted in any case:
      return
   }
   while(particle_size >= x) {
      x = x * fHierarchy
   }
   create a new hash grid with cell_size = x
   and add it to the list as last element
   insert particle into the new grid
}
```

back to section 3.2

Collision detection:

```
for(Grid* G : gridList) {
   for(HashCell** c1 : G->occupied_cells) {
      for(size_t p1 : *((*c1)->particle_ids)) {
         //Check infinite sized
         for(size_t p2 : infiniteSized) {
            check_collision(p1, p2);
         }
         //Check inside cell
         for(size_t p2 : *((*c1)->particle_ids)) {
            if(p1 < p2) {
               check_collision(p1, p2);
            }
         }
         //Check adjacent cells
         for(size_t i = 0; i < 13; i++) {
            HashCell* c2 = *(c1+(*c1)->offset->vals[i]);
            if(c2 != nullptr) {
               for(size_t p2 : *(c2->particle_ids)) {
                  check_collision(p1, p2);
               }
            }
         }
         //Check higher Grids
         for(Grid* G2 : gridList) {
            if(G->cell_size < G2->cell_size) {
               size_t hash = G2->getHash(ac.getPosition(p1));
               offsets* offset_pattern;
               if(G2->cells[hash] != nullptr) {
                  offset_pattern = G2->cells[hash]->offset;
               }
               else {
                  size_t N = G2->N;
                  size_t offset = 13;
                  if((hash & (N-1)) == 0) {
                     offset -= 1;
                  }
                  else if((hash & (N-1)) == N-1) {
                     offset += 1;
                  }
                  if((hash & (N*N-1)) < N) {
                     offset -= 3;
                  }
                  else if((hash & (N*N-1)) >= N*(N-1)) {
                     offset += 3;
                  }
                  if(hash < N*N) {
                     offset -= 9;
                  }
                  else if(hash >= N*N*(N-1)) {
                     offset += 9;
                  }
                  offset_pattern = &((*(G2->offsetPtr))[offset]);
               }
               for(size_t i = 0; i < 27; i++) {
                  HashCell* c2 = G2->cells[hash+offset_pattern->vals[i]];
                  if(c2 != nullptr) {
                     for(size_t p2 : *(c2->particle_ids)) {
                        check_collision(p1, p2);
                     }
                  }
```

```
            }
          }
        }
      }
    }
}
```