

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)



Hierarchical Hash Grids for Coarse Collision Detection

Florian Schornbaum

Studienarbeit

Hierarchical Hash Grids for Coarse Collision Detection

Florian Schornbaum

Studienarbeit

Aufgabensteller: Prof. Dr. Ulrich Rde
Betreuer: Klaus Iglberger, M. Sc.
Bearbeitungszeitraum: 15.01.2009 – 05.10.2009

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.

Erlangen, den 05.10.2009

.....

Abstract

In any large-scale simulation that is dealing with a huge number of objects, the broad-phase collision detection becomes a crucial part for achieving high performance. The focus of this paper will be on developing a time-efficient as well as memory-efficient implementation of a data structure called hierarchical hash grid, which is especially well-suited for coarse collision detection in simulations involving very large numbers of objects. As part of the *pe* physics engine, the hierarchical hash grid enabled the realization of a massively parallel rigid body simulation involving more than one billion objects ($1.14 \cdot 10^9$).

Further key features of the presented implementation are not only an average-case computational complexity of order $O(N)$ as well as a space complexity of order $O(N)$, but also a short actual runtime combined with low memory consumption. Moreover, the hierarchical hash grid has the ability to, at runtime, automatically and perfectly adapt its data structures to the objects of the underlying simulation. Also, arbitrary objects can be added and removed in constant time, $O(1)$. In the course of this paper, the necessary theoretical background is presented, detailed information on the actual implementation is given, and a number of benchmarks are performed in order to evaluate the runtime performance.

Contents

1	Introduction	1
2	Conceptual Design	2
2.1	Intersection Detection	2
2.2	Coarse Collision Detection & Bounding Volumes	2
2.3	Uniform Grids	4
2.4	Grid Hierarchies	7
2.5	Hashed Storage & Infinite Grids	9
2.6	Updating the Data Structure	11
2.7	Rotating Objects & Resizing Bounding Volumes	12
3	Implementation	18
3.1	Data Structures for the Grid	18
3.2	Hash Calculation	23
3.3	Assigning Objects to the Hierarchical Hash Grid	26
3.4	Assigning an Object to a Particular Grid	28
3.5	Removing Objects	30
3.6	The Detection Step	30
3.7	The $O(N^2)$ Bound	32
3.8	Drawbacks & Worst Case Scenarios	32
3.9	Computational Effort & Space Complexity	34
4	Benchmarks	36
4.1	General Performance Scaling	37
4.2	The Hierarchy Factor & Number of Objects per Cell	39
4.3	The Update Phase	40
4.4	Worst Case Scenarios	40
4.5	The <i>pe</i> Physics Engine	41
5	Conclusion	45
5.1	Performance & Scaling Characteristics	45
5.2	Final Words	45

1 Introduction

Collision detection systems provide the basis for simulating physically correct object-object interactions. They are necessary for determining the contact points between all interacting objects. Once all points of contact have been found, this information can be used in other parts of the simulation framework in order to further deal with the detected collisions and calculate an appropriate response. To very efficiently find all pairs of potentially colliding objects is the task of the broad-phase collision detection, which is also referred to as coarse collision detection opposed to fine collision detection. When dealing with a huge number of objects, the coarse collision detection becomes a crucial part in designing fast and efficient simulation systems.

The work related to this paper is the result of the extension of the *pe* physics engine by a new coarse collision detection algorithm. The *pe* physics engine is mainly developed and maintained by Klaus Iglberger [IR09a, IR09b]. It aims at realistic, i.e., physically correct rigid body simulations. One of the most outstanding features of the *pe* framework is its parallelization, which enables massively parallel simulations. The goal is to provide a framework that is capable of performing large-scale simulations that involve millions or even billions of objects. Such simulations require an efficient coarse collision detection algorithm that is especially well-suited for these huge numbers of objects. Moreover, since the parallelization is based on the continuous exchange of objects between the parallel processes/domains, the data structures that are used for the (coarse) collision detection must be able to efficiently handle the addition and removal of arbitrary objects at runtime.

The focus of this paper will be on developing a solution for coarse collision detection that is based on a hierarchy of grids, with each grid being realized based on hashed storage. The data structure that manages these hierarchical hash grids and all the related algorithms that operate on it will be referred to as the “hierarchical hash grid”. The ideas and concepts of [Eri05] and [ESHD05] serve as a basis for the work that is presented in this paper. Some ideas are adopted, some are discarded, and some new concepts are developed, all motivated by the goal of creating a coarse collision detection method that combines the benefits of all the underlying ideas: linear performance scaling and a low memory consumption, both independent of the spatial distribution of the objects, object movement, and the size and geometry of each object. Consequently, the hierarchical hash grid will be capable of dealing with a huge number of objects. Additionally, the requirements of the parallelization of the *pe* physics engine will be met, i.e., the insertion and the removal of arbitrary objects will be possible in constant time, $O(1)$.

This paper is divided into three parts. The first part deals with the conceptual design of the hierarchical hash grid and presents the necessary theoretical background. In the second part, detailed information on the actual implementation is given. The implementation is based on C++. However, all statements and code snippets are easily transferable to other low-level programming languages. The third and final part of this paper is about analyzing the results of various benchmarks, i.e., evaluating the general performance and the performance scaling of the hierarchical hash grid for different benchmark scenarios – one scenario being a comparison within the framework of the *pe* physics engine against an already implemented sweep and prune algorithm.

2 Conceptual Design

2.1 Intersection Detection

All the algorithms and data structures that are presented in this paper are based upon a simulation system that is characterized by the following two properties:

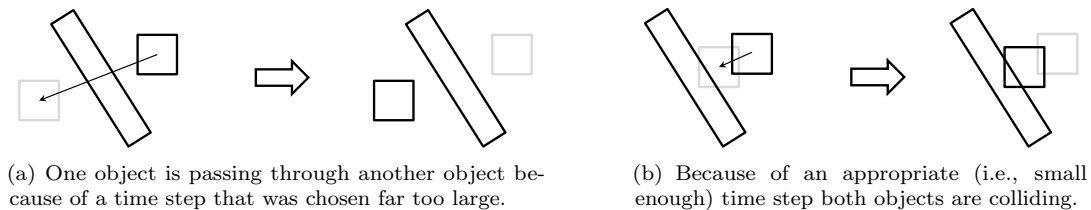
1. The simulation is carried out on the basis of time steps.

In order to detect every collision among all the objects in the simulation, the full continuous motion of each object in-between two time steps is not taken into account. In general, such a dynamic collision detection is very complex, i.e., time-consuming and thus expensive.

Therefore, detecting collisions is based upon a (much) cheaper static collision detection. This only involves detecting intersections between objects at discrete points in time. Obviously, if intersections are found, they have to be treaded accordingly. Consequently, objects are no longer following a continuous but a time-discrete motion pattern. This directly results in time steps which – depending on the velocities and sizes of the simulated objects – have to be chosen very small in order to prevent “tunneling”, i.e., objects passing each other. Therefore, over the period of one time step the movement of each object must be less than its spatial extent (see Figure 1).

2. In-between two time steps, all objects in the simulation are moving simultaneously, and afterwards they are all tested for intersections at the same time. This leads to the fact that object pair checking is commutative.

Since detecting collisions is achieved by detecting intersections, the terms “collision detection” and “intersection detection” are used synonymously throughout the rest of this paper. Hence, if two objects are said to collide, this always means both objects are intersecting at a certain point in time.



(a) One object is passing through another object because of a time step that was chosen far too large.

(b) Because of an appropriate (i.e., small enough) time step both objects are colliding.

Figure 1: Tunneling as a result of an inappropriate choice for the time step of a simulation

2.2 Coarse Collision Detection & Bounding Volumes

In practice, finding every pair of intersecting objects proves to be difficult:

- The more complex the geometry of two objects, the more expensive it gets to check whether both objects are intersecting each other.
- In any large-scale simulation, during every single time step, each object won’t collide with most of all the other objects since most objects are simply too far apart to result in an intersection.

Taking both these points into consideration while designing collision detection systems is crucial for good performance. Not taking care of these points will always lead to an unnecessarily high computational effort, and thereby to bad performance.

Therefore, in order to overcome the problem arising from very complex object geometries, bounding volumes – a common method for accelerating object intersection tests – are introduced. Bounding volumes are closed volumes that completely contain the associated objects. Additional important features are: for any given object, the bounding volume is fast to compute, easy to update and well-suited for efficient intersection tests. Hence, the reason why they are used when it comes to collision detection is pretty obvious: in order to very quickly rule out a potential collision between two objects, which is the case every time the corresponding bounding volumes are not intersecting. The two most common and best suited bounding volumes related to collision detection are:

1. The bounding sphere (see Figure 2(b)), which is represented by a center point and a radius.

Two bounding spheres are intersecting if and only if the distance between their centers is less or equal to the sum of their radii (see Code Snippet 1).

2. The axis-aligned bounding box, also known as AABB (see Figure 2(c)), which can be represented by two vertices – containing the minimum and maximum coordinate values, respectively.

Two AABBs are not intersecting if an intersection is impossible due to at least one pair of their vertices' coordinate values (for a better understanding see Code Snippet 2).

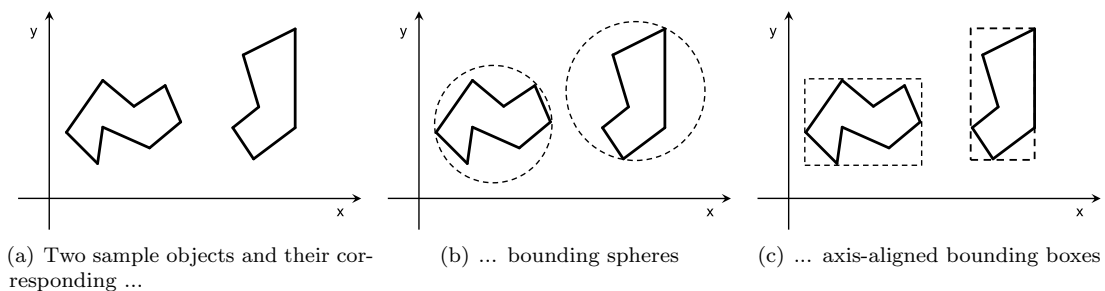


Figure 2: Illustration of different types of possible bounding volumes

Checking two objects based on their exact – and thus potentially highly complex – geometry is referred to as fine collision detection. In contrast, every approach that uses some sort of early elimination in order to avoid fine collision detection for as many pairs of objects as possible is called coarse collision detection.

Code Snippet 1: Function to check whether two bounding spheres are overlapping

```
bool overlap( BS& a, BS& b ) {
    Vector distance = a.center - b.center;
    double radius_sum = a.radius + b.radius;
    if( ( distance.x * distance.x + distance.y * distance.y +
          distance.z * distance.z ) <= radius_sum * radius_sum ) return true;
    return false;
}
```

Code Snippet 2: Function to check whether two AABBs are overlapping

```

bool overlap( AABB& a, AABB& b ) {
    if( a.min.x > b.max.x || a.min.y > b.max.y || a.min.z > b.max.z ||
        a.max.x < b.min.x || a.max.y < b.min.y || a.max.z < b.min.z ) return false;
    return true;
}

```

A fairly naive approach to coarse collision detection that involves bounding volumes is the simulation-global all-pair object test (see Code Snippet 3). In this approach, each object – admittedly only its corresponding AABB, but still – is checked against every other object in the simulation. This, of course, leads to quadratic complexity, $O(N^2)$. Thus, a simulation-global all-pair object test is clearly not feasible for large numbers of objects (see benchmark in Section 4.1).

Code Snippet 3: The simulation-global all-pair object test $\rightarrow O(N^2)$

```

for( int i = 0; i < N; ++i ) {
    for( int j = i+1; j < N; ++j ) {
        if( overlap( object[i].aabb, object[j].aabb ) )
            /* fine collision detection for object[i] and object[j] */
    }
}

```

As already mentioned at the beginning of this section, collisions only occur between spatially adjacent objects. Therefore, the goal while designing an efficient coarse collision detection system has to be to exploit this spatial locality in order to develop an algorithm that possesses an average-case computational complexity that is superior to any procedure of order $O(N^2)$. To achieve that goal, various different approaches exist, such as octrees, binary space partitioning (BSP trees), or the sweep and prune algorithm (see Section 4.5). Another approach – the approach taken in this paper – are uniform grids.

2.3 Uniform Grids

The basic idea behind uniform grids, or simply just grids as they are referred to throughout the rest of this paper, is relatively simple: the simulation space is uniformly subdivided into cubic cells and the objects are, based on their bounding volumes, assigned to these cells. In this way, spatial partitioning is achieved. Since all grid cells are cubes, all their edges are of equal length. This edge length is referred to as the size of the cell. Likewise, the size of an object is, in a very similar manner, defined based on the object’s bounding volume. Thus, depending on the type of the bounding volumes that are used, the size of an object is referring to either the diameter of its corresponding bounding sphere or the length of the longest edge of its corresponding AABB.

When it comes to inserting objects into grids – i.e., assigning objects to grid cells – two different strategies exist:

1. If the bounding volume of an object overlaps multiple grid cells, then this object is assigned to all of these cells (see Figure 3(a)).
2. The cell size of the grid is ensured to be larger than the largest object (see Figure 3(b)). As a result, it is sufficient to assign every object to one single cell only (see Figure 5(a)).

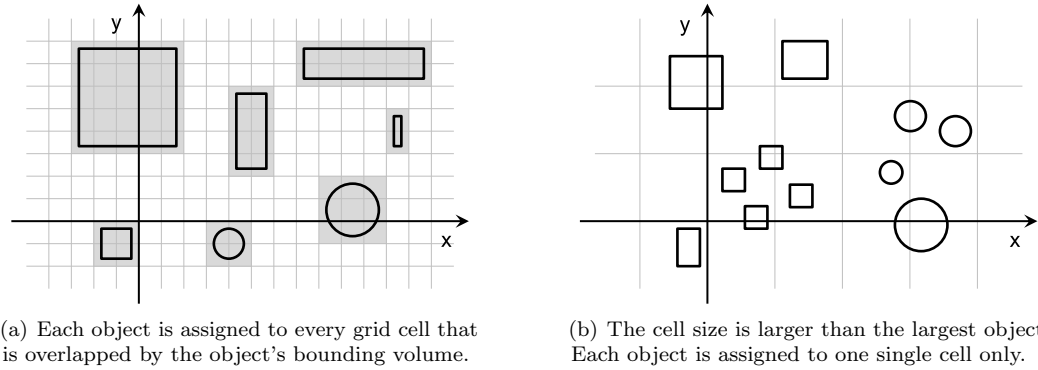


Figure 3: Two different strategies for inserting objects into grids

In order to assign an object to all of its overlapping cells (see Figure 3(a)), first of all, all these cells must be identified based on the object's bounding volume. This is not too hard to achieve as long as AABBs are used. However, when it comes to bounding spheres, this task turns out to be slightly more complicated – especially in 3D. Once all objects are assigned to grid cells, all occurring collisions can be detected by iterating through all the cells that are occupied by objects, followed by pairwise tests within each of these cells. Of course, even when using grids – along with either of both object insertion strategies – testing two objects still means, first of all, checking if the corresponding bounding volumes are intersecting. If and only if this is the case, objects are finally checked by means of fine collision detection. The strategy of assigning an object to all the grid cells that are overlapped by the object's bounding volume, however, has two major drawbacks:

1. The same collision between two particular objects might be detected multiple times.

Clearly, during one time step, two objects have to be checked against each other only once. If, however, two objects are assigned to several identical cells, these two objects are subject to being checked against each other multiple times. Preventing multiple identical intersection tests requires some additional treatment. This has to be done explicitly and cannot be achieved implicitly through a clever iteration strategy.

2. It is impossible to choose a universally well-suited cell size.

Some objects of the simulation might be much larger than most of all the other objects. As a result, very large objects might overlap a lot of grid cells. This leads to an increased computational effort, which, of course, is undesirable. In summary, any cell size that is much smaller than some/all objects in the simulation (see Figure 4(a)) should be avoided.

On the other hand, if the grid's cell size is adapted to the largest objects in the simulation, there might be other objects that are much smaller. This could result in a lot of objects being assigned to the same cell (see Figure 4(b)), which, of course, is also undesirable since within every cell the computational effort is, due to performing an all-pair test (cf. Code Snippet 3), always quadratic in the number of objects.

The other insertion strategy would be to ensure that the grid cells are larger than the largest object of the simulation (see Figure 3(b)), so that objects may be assigned to one single cell only (see Figure 5(a)). As a result, finding all occurring collisions is reduced to only checking each object against the objects that are stored in the same or in any of the directly adjacent cells. If two objects are spatially separated by one or more cells, an intersection becomes geometrically impossible.

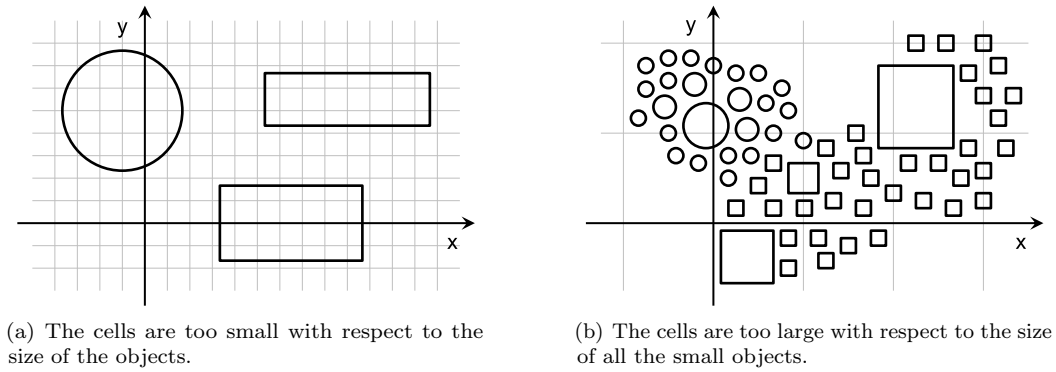


Figure 4: Issues related to the cell size

This is pretty obvious when dealing with bounding spheres. An object is assigned to a cell based on its bounding sphere center. Since, by definition, the cell size is ensured to be larger than the diameter of any bounding sphere in the simulation, two spheres which are spatially separated by one or more cells can never intersect.

When it comes to AABBs, this is slightly more complicated. The point of reference used to calculate the cell association has to be identical for every object. For instance, the center of the AABB or either of both vertices can be used, but they must never be mixed. Figure 5(a), for example, uses the lower corner of the AABB in order to assign objects to cells. In Figure 5(b), using the lower corner as point of reference leads to object A being assigned to cell number 2 and object B being assigned to cell number 1. If the upper corner is used, object A is assigned to cell number 4 and object B is assigned to cell number 3. In either case, both cells are adjacent and thus the intersection is detected. If, however, object A would be assigned to cell number 4 based on the upper corner of its AABB and object B would be assigned to cell number 1 based on the lower corner of its AABB, the intersection of both objects could not be detected if only adjacent cells are tested.

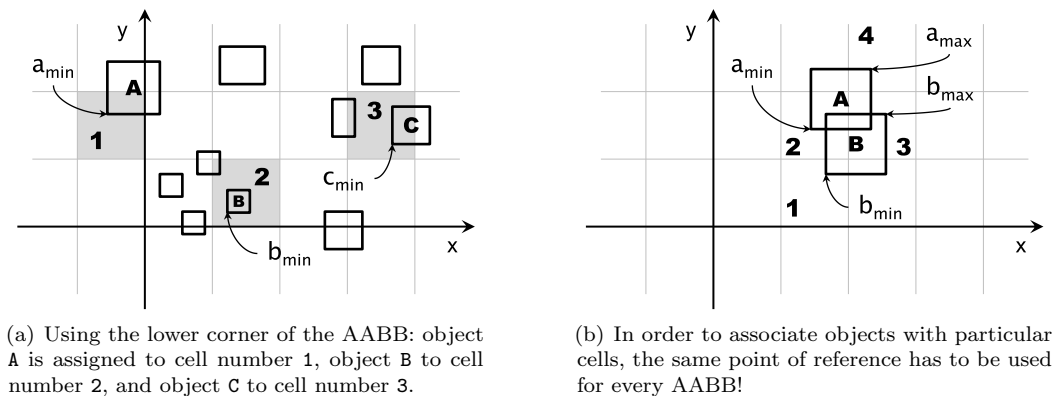


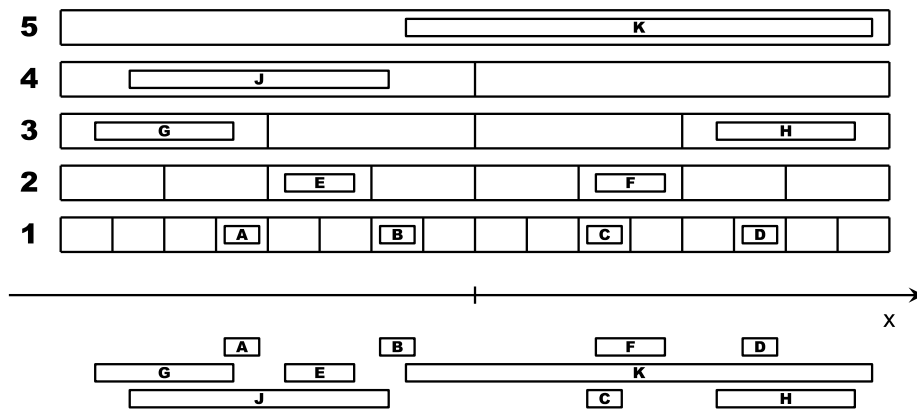
Figure 5: Issues related to assigning objects to one single cell

Naively checking each object against the objects that are stored in the same or in any of the directly adjacent cells results in detecting every intersection twice. However, by using a clever iteration scheme, this problem can be avoided implicitly and thus without additional cost (a more detailed explanation will follow in Section 3.6). Therefore, the strategy of choice for inserting an object into a grid is to assign the object to one single cell only and to simultaneously ensure that the

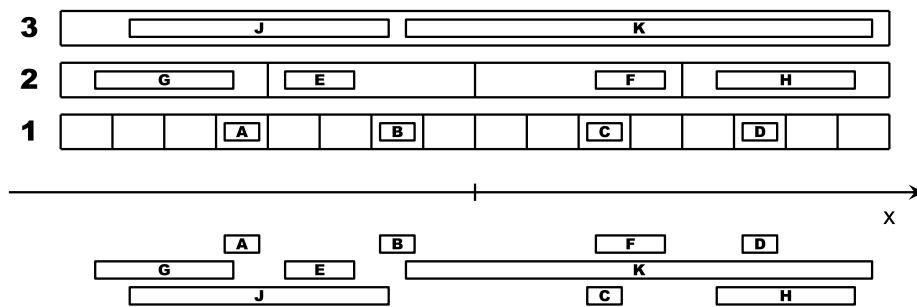
cells of the grid are always larger than the largest object in the simulation. However, one problem still remains: if the objects in a simulation vary strongly in size, there might be a lot of very small objects, many of them which could end up being assigned to the same cell (see Figure 4(b)).

2.4 Grid Hierarchies

In order to solve the problem arising from the fact that many small objects could get assigned to the same cell, the simulation space is not only divided into just one single grid, but into several spatially superimposing grids – thereby creating a hierarchy of grids. Although all of these grids are subdividing the very same space, each grid has a different and thus unique cell size. As a result, for every object, there exists a grid with properly-sized cells. In order to being able to efficiently make use of this hierarchy, all the grids are sorted in ascending order with respect to the size of their cells. Consequently, every object is assigned to one single grid in the hierarchy only, which happens to be the first grid with a cell size larger than the size of the object (see Figure 6).



(a) Some one-dimensional objects are assigned to a grid hierarchy consisting of five grids. In this hierarchy, the cells of two successive grids double in size.



(b) Some one-dimensional objects are assigned to a grid hierarchy consisting of three grids. In this hierarchy, the cells of two successive grids quadruple in size.

Figure 6: Objects are, with respect to their size, assigned to a hierarchy of grids

Thus, even if the objects of a simulation vary strongly in size, there is no danger of many small objects being assigned to the same cell. Small objects are stored in grids with appropriately sized small cells, whereas large objects are stored in grids with appropriately sized large cell. But objects of very different sizes are never assigned to the same grid – which results in a manageable number of objects within each cell of every grid.

Once all objects are assigned to suitable grids, the grid hierarchy can be used to detect all the occurring collisions. However, when using such a grid hierarchy, it is not sufficient any more to only check an object against other objects in the same grid. Obviously, now every grid in the hierarchy must be considered, which means every object must also be checked against the objects that are stored in other grids. However, since object pair checking is commutative and all objects in the simulation are tested simultaneously, when an object is processed, it is only checked against objects that are stored in grids with larger sized cells. As a result, detecting the same intersection twice is implicitly prevented for all objects of two different grids (see Figure 7 and Algorithm 1). Of course, each object is still checked against other objects in the same grid. Slightly altering the procedure outlined in Algorithm 1 with regard to the final layout of the data structure will, even for objects within the same grid, implicitly avoid detecting the same intersection twice. As already mentioned at the end of Section 2.3, this will be explained in detail in Section 3.6.

Algorithm 1: Finding all occurring collisions when using a hierarchy of grids

```

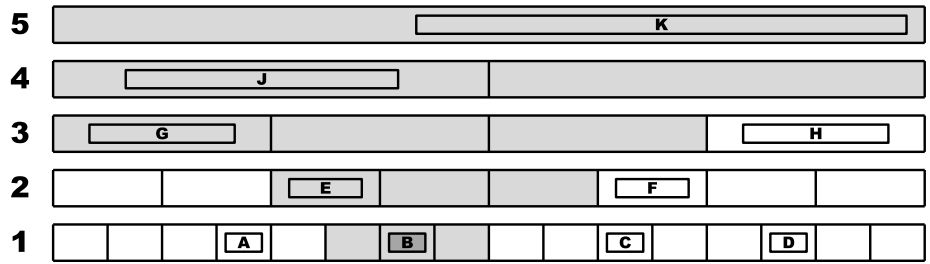
1 foreach object x of the simulation do
2    $G \leftarrow$  the grid that  $x$  is assigned to
3    $C_G \leftarrow$  the cell within  $G$  that  $x$  is stored in
4   detect all intersections of  $x$  within grid  $G$  by...
5   ...checking  $x$  against all other objects within  $C_G$  and...
6   ...checking  $x$  against any object that is stored in a cell that is directly adjacent to  $C_G$  and...
7   ...at the same time avoiding to detect the same intersection twice
8   forall grids  $H$  with cells larger than the cells of  $G$  do
9      $C_H \leftarrow$  the cell within  $H$  that  $x$  would be associated with
10    detect all intersections of  $x$  within grid  $H$  by...
11    ...checking  $x$  against all objects within  $C_H$  and...
12    ...checking  $x$  against any object that is stored in a cell that is directly adjacent to  $C_H$ 
13  end
14 end

```

Clearly, the larger the ratio between the cell size of two successive grids, the more objects are potentially assigned to one single cell, but overall fewer grids have to be used. On the other hand, the smaller the ratio between the cell size of two successive grids, the fewer objects are assigned to one single cell, but overall more grids have to be created. Hence, the number of objects that are stored in one single cell is inversely proportional to the number of grids which are in use. Unfortunately, minimizing the number of objects that are potentially assigned to the same cell and at the same time also minimizing the number of grids in the hierarchy are two opposing goals. Consequently, designing such a hierarchy of grids raises some obvious questions:

- For any given set of objects, how many grids should be created to be part of the hierarchy?
- For each grid, how exactly should the size of the cells be chosen?
- Is the cell size of one grid dependent on the cell size of all the other grids? And if so, what is their relation?

These questions will be answered when looking at the actual implementation in Section 3.3. Additionally, benchmarks concerning this topic can be found in Section 4.2.



(a) When object B is processed, it is checked against all the objects that are stored in the grey colored cells – that is, objects E, G, J, and K.



(b) When object F is processed, it is checked against the objects H, J, and K. However, F is not tested against object C since grid 1 possesses smaller cells than grid 2 and thus the intersection between C and F is already found when C is processed.

Figure 7: Collision detection when using a hierarchy of grids

2.5 Hashed Storage & Infinite Grids

One issue that has not been addressed at all until this point is the extent of the grid. When dealing with a lot of small and widely scattered objects, which are assigned to a grid with correspondingly small cells, plenty of cells – and thus a very large grid – are required. In general, whenever objects in a simulation are located far apart relative to their size – and thus also relative to the corresponding cell size of the grid – a lot of grid cells are required. This is also true if, during a simulation, some objects travel great distances and thereby cover a lot of space.

In any case, all these scenarios result in a very large grid, which will undoubtedly exhaust main memory resources. For example, if the memory requirement of each single grid cell would be a few bytes and if 10.000 cells would be required in each dimension, this would, in a three-dimensional simulation, already sum up to 10^{12} cells, and thus the data structure for this grid alone would claim memory in the range of several terabytes. A possible solution could be to spatially restrict the simulation domain and to simultaneously ensure that initially – but also after every time step – objects do not end up outside of this confined space. If objects leave the simulation space that is covered by the grid, the response could be the termination of the program in combination with an appropriate error notification. Such a solution, however, is anything but desirable and would result in certain simulations being impossible to realize.

As a consequence, representing grids as dense arrays is not feasible, and in order to conserve memory another approach must be taken. Like in many other applications, the solution to this problem is the usage of hashed storage. Some sort of hashing is introduced in order to map the infinite number of cells, which originated by uniformly subdividing the unlimited simulation space into cubes, to a finite set of buckets. Occurring hash collisions may be handled with traditionally used and well-known methods like open addressing or separate chaining (the latter is often also referred to

as open hashing or closed addressing). However, since objects are always checked against each other based on their bounding volumes first, hash collisions can be safely ignored. If two spatially distant objects are mapped to the same hash bucket, comparing their corresponding bounding volumes won't result in finding an intersection, and thus, as expected, both objects won't be subject to fine collision detection.

The first implementation to cross one's mind will most likely look like a straightforward realization of a hash table. The grid is represented by a linear array and thereby corresponds to a hash table with a limited number of buckets. Afterwards, a hash function that maps any cell in the simulation space to one of these hash buckets is constructed. In fact, this straightforward approach is very well suited for implementing hashed storage as long as for the insertion of objects into grids the strategy of assigning an object to each cell that is overlapped by the object's bounding volume is used. However, as already stated at the end of Section 2.3, the strategy of choice for inserting objects into grids is to assign each object to one single cell only, since this approach allows for a collision detection scheme that implicitly – without requiring any extra treatment – avoids finding identical intersections multiple times. In the course of searching for all occurring collisions, this strategy heavily relies on continuous access to directly adjacent grid cells. Since when using such a straightforward hash table implementation objects are randomly mapped to identical hash buckets, it is not possible to precompute neighborhood relationships. Consequently, for each object, the hash values of the directly adjacent cells would have to be calculated – by evaluating the hash function – at runtime. This computational effort seems to be somehow unpleasant and leads to the assumption that a straightforward hash table implementation is not the best option for realizing hashed storage. At least, not if the clearly favored object insertion strategy of assigning each object to one single cell only is used.

The solution – maintaining the benefits of hashed storage and at the same time eliminating the just illustrated problems – is geometrically based hashing. The idea is to use modulo operations in order to spatially map entire blocks of connected cells to the origin of the coordinate system. Throughout the rest of this paper, this block of cells at the origin of the coordinate system that is filled with all the objects of the simulation is referred to as hash grid. The key feature, and ultimately the advantage, of hash grids is the fact that two adjacent cells that are located anywhere in the simulation space are mapped to two cells that are still adjacent in the hashed storage structure. For a better understanding, the concept of hash grids is illustrated in Figure 8.

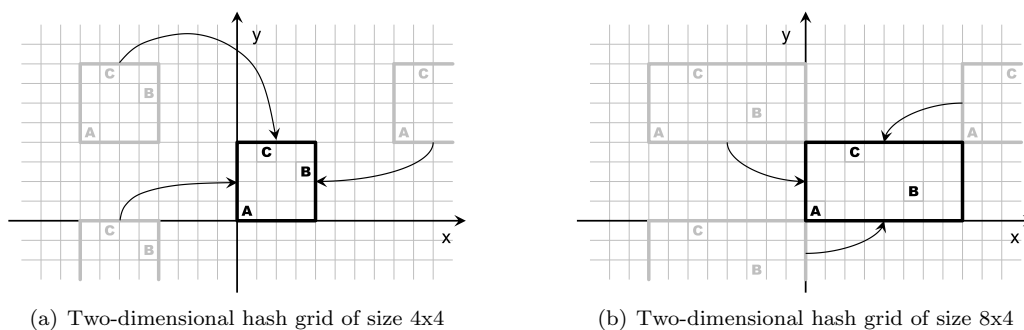


Figure 8: Illustration of geometrically based hashing in hash grids

To ensure fast direct access to adjacent cells at runtime, an index to these neighboring cells is initially precomputed for every cell in the hash grid (obviously, periodic boundary conditions have

to be used for all the cells at the edge). Once this is done, in order to detect all occurring collisions, hash grids can be used in the exact same manner as grids that are implemented on the basis of dense arrays. Hash collisions are – as already mentioned – handled implicitly by always checking the bounding volumes first before performing fine collision detection. The only two differences between hash grids and grids based on dense arrays are, obviously, the way objects are inserted, and the fact that by using hash grids, spatially infinite grids that have very limited memory requirements can be realized. Of course, there will always be limitations to the “infinity of the grid” due to the finite precision with which computers generally represent numbers. The resulting effects and a more detailed study on this topic can be found at the end of Section 3.2.

The big advantage of hash grids, namely, the limited memory requirements due to the limited number of cells being needed, raises an obvious question: for any given simulation, how many cells are actually required? Clearly, the number of required cells is somehow dependent on the number of objects. This ratio of cells to objects is referred to as the grid density. Obviously, the fewer cells are allocated – i.e., the fewer hash buckets are used – the fewer memory is required, but the more likely objects are mapped to the same cell. On the other hand, the more cells are allocated the less likely objects are inserted into the same cell. Consequently, minimizing memory consumption and at the same time also minimizing the number of objects that are potentially assigned to the same cell are, unfortunately, two opposing goals. Therefore, the performance of any hash grid implementation will depend on the following two aspects:

1. The size of the hash grid, i.e., the number of actually allocated cells (e.g., 4x4 vs. 8x8).
2. But also the cell distribution within the hash grid, meaning, the number of cells in each direction in space (e.g., 4x4 vs. 8x2 – both consisting of 16 cells).

More detailed studies on these two topics will follow when looking at the actual implementation in Sections 3.2, 3.3, 3.4, and 3.8. Apart from the number of actually allocated cells and the cell distribution within the hash grid, the performance of every hash grid implementation will, of course, also largely depend on the size of the cells in relation to the size of the inserted objects.

2.6 Updating the Data Structure

By now, most of the necessary aspects concerning the conceptual design of a hierarchical hash grid have already been discussed. In summary, issues that have been resolved so far are:

- The way objects are stored in grids.

Each object is assigned to one single cell only, while at the same time it is ensured that the size of the grid’s cells is larger than the bounding volume of the object (see Section 2.3).

- The way the data structure handles objects that greatly vary in size.

A hierarchy of superimposing grids enables each object to be assigned to a grid with suitably sized cells – meaning, cells that are neither too small nor too large (see Section 2.4).

- The way grids are represented.

Hashed storage (spatial hashing based on modulo operations) leads to a conceptually infinite grid. Adjacent cells anywhere in the simulation space are mapped to still adjacent cells in the hash grid (see Section 2.5).

- The way the data structure is used in order to find all the occurring collisions.

Within each grid, the detection strategy is based on checking directly adjacent cells. Between different grids, it is based on, for each object, only checking grids with larger sized cells (see Section 2.4 and Algorithm 1).

Yet, there is no mechanism for changing the cell association of an object. After being inserted into a grid, objects are stuck in whatever cell they are initially stored in. However, objects moving around in space are the very essence of every physics simulation. Consequently, there has to be some mechanism to, in every time step, adapt the data structure to the constantly changing spatial distribution of the objects. In fact, two approaches that realize such an update strategy exist:

1. All objects are simultaneously removed from the data structure, and immediately afterwards they are reinserted based on their current locations – which may or may not have changed since the last time step.

This admittedly naive approach obviously leads to a consistent state, and it has one major advantage: it is very easy to implement. Removing all objects is equivalent to clearing all grid cells – which can be done extremely efficiently as long as one keeps track of all the object-occupied cells while inserting objects. Moreover, reinserting an object is not different from initially inserting an object, hence the exact same methods may be used.

2. For every object, the hash value is recomputed based on the object’s current spatial location. If this new hash value is identical to the hash value of the previous time step, the object remains assigned to its current grid cell. Only if the hash value changes, the cell association has to be changed, too – meaning, the object has to be removed from its currently assigned cell and stored in the cell that corresponds to the new hash value.

Removing a specific object from a grid cell is – no matter how objects are stored – more expensive than just clearing a cell and thereby removing all of its objects. Hence, the performance of this update strategy largely depends on the number of objects that actually change their cell association. However, as already mentioned in Section 2.1, over the period of one time step, the movement of each object will be less than its spatial extent, and therefore also less than the spatial extent of the corresponding grid cell. In fact, the movement of an object usually only equates to a fraction of the object’s size. As a result, objects that remain stored in their currently assigned cells are far more likely than objects that change their cell association. Therefore, this approach turns out to be the strategy of choice for updating the data structure. For a more detailed study, see the benchmarks in Section 4.3.

Consequently, when using hierarchical hash grids, the coarse collision detection is always separated into two different stages: the update phase followed by the detection step. In every time step, the data structure is, first of all, updated as just described in order to adapt to the current object distribution. Afterwards, all the occurring collisions are detected by using the at this point up-to-date data structures.

2.7 Rotating Objects & Resizing Bounding Volumes

In the course of updating the data structure, apart from the fact that objects in physics simulations are moving around in space and thus are constantly changing their location, the rotation and – if the

simulation allows objects to be resilient – potential deformation of an object must also be considered. In the case of rigid body simulations, deformations are by definition impossible and thus can be safely ignored. In either case, the rotation – and, if allowed, the deformation – of an object could alter the size of its bounding volume and thus potentially lead to the object having to be removed from its current grid and reassigned to another grid with suitably sized – i.e., larger or possibly also smaller – cells. Hence, in contrast to object movement, object rotation and deformation not only affect the cell association but also the grid association of an object.

Obviously, if an object is only rotated, the diameter of its corresponding bounding sphere remains unchanged. Once initially computed, the bounding sphere – i.e., the bounding sphere’s center point – is, in every time step, moved together with the enclosed object. If the underlying simulation only involves rigid bodies, the radius of each object’s bounding sphere remains constant throughout the entire simulation. Thus, once inserted, each object remains associated with the same grid. However, even if this aspect seems to be desirable, bounding spheres have one major disadvantage: for any given object, the diameter of the corresponding minimal bounding sphere[†] – which will be referred to as the size of the bounding sphere (following the definition of the size of an object in Section 2.3) – is always greater than or equal to the length of the longest edge of the corresponding AABB – referred to as the size of the AABB. Proof:

Given two points p_a and p_b , which are w.l.o.g. the two points farthest apart from each other within the object’s geometry, let s be the distance between these two points.

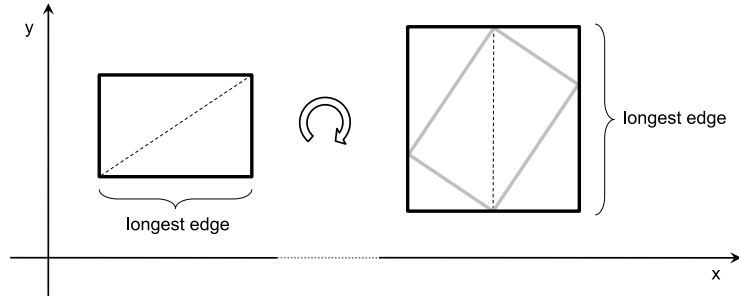
Hence, s specifies an upper limit for the size of the AABB. This upper limit is reached if and only if the straight line connecting p_a and p_b is axially aligned. In any other case, the size of the AABB will most likely be smaller than s .

On the other hand, s specifies a lower limit for the diameter of the corresponding bounding sphere. Obviously, the diameter cannot be smaller than s . It might, however, be larger. Consequently, for any given object, the size of the corresponding AABB is always smaller than or equal to the size of the corresponding bounding sphere. \square

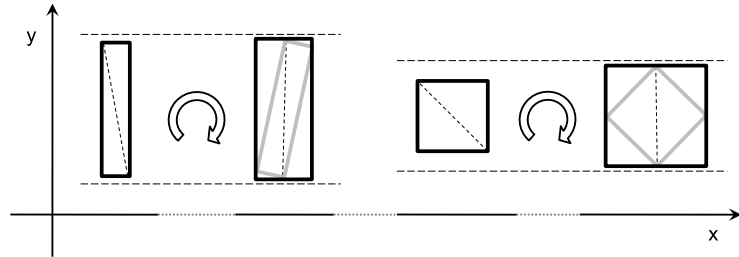
In 3D, the worst case is an axis-aligned cube, the corresponding bounding sphere of which is $\sqrt{3} \approx 1.73$ times larger than the corresponding AABB. In general, whatever the geometry of an object, if the AABB instead of the most likely larger bounding sphere is used as bounding volume, the object is possibly inserted into a grid with smaller sized cells – thereby reducing the average number of objects that are stored in one single cell. As a result, the bounding volume of choice for implementing a hierarchical hash grid is the axis-aligned bounding box.

However, if axis-aligned bounding boxes are used, the rotation of an object will most certainly result in a change in the size of its corresponding AABB. For any rectangular cuboid, the greatest possible factor by which the size of its corresponding AABB can change is given by the ratio of the length of the space diagonal to the length of the longest edge (see Figure 9(a)). Clearly, if two objects are of different shape, there may also be differences in how much their respective AABBs are varying in size (see Figure 9(b)). Thus, in order to further study the consequences of rotating objects, it is important to specify an upper bound – that is, the greatest possible factor by which the size of the AABB of any arbitrary object can change. Intuitively, this upper bound should be found in the case of a cube, and any other rectangular cuboid should correspond to an AABB that is – when arbitrarily rotating the cuboid – varying less in size.

[†]smallest possible sphere enclosing the object geometry



(a) Given any rectangular cuboid C , the greatest possible factor by which the size of its corresponding AABB can change is given by C being initially aligned to the coordinate axes and then rotated to the point where the space diagonal lines up with one of these axis – thereby producing the longest edge that is possible for any AABB that is corresponding to C .



(b) The amount by which the size of an AABB can change largely depends on the geometry (shape) of the corresponding object. The largest amount by which the size of the AABB of any object can change – that is, the upper bound for the resizing of any AABB – is given by a cube.

Figure 9: Changes in the size of the AABB resulting from rotation of the corresponding object

In order to verify this intuitive assumption, the ratio of the length of the space diagonal to the length of the longest edge has to be studied. If and only if maximizing this ratio for any arbitrary rectangular cuboid leads to a cube, the assumption is proven to be valid. Consequently, a formal verification may look like as follows:

Given any rectangular cuboid with edge lengths $x > 0$, $y > 0$, and $z > 0$, the length of the space diagonal is specified as

$$d = \sqrt{x^2 + y^2 + z^2} \quad . \quad (2.7.1)$$

Without loss of generality, let x be the length of the longest edge of the cuboid:

$$x \geq y \geq z > 0 \quad \implies \quad 2x \geq y + z \quad (2.7.2)$$

Hence, the side condition:

$$g(x, y, z) = y + z - 2x \quad (2.7.3)$$

Maximizing the ratio of the length of the space diagonal to the length of the longest edge

$$\frac{\sqrt{x^2 + y^2 + z^2}}{x} \quad (2.7.4)$$

is equivalent to maximizing

$$f(x, y, z) = \frac{x^2 + y^2 + z^2}{x^2} = 1 + x^{-2}y^2 + x^{-2}z^2 \quad . \quad (2.7.5)$$

Maximizing $f(x, y, z)$ subject to side condition $g(x, y, z)$:

$$\vec{\nabla} f(x, y, z) = \lambda \vec{\nabla} g(x, y, z) \quad (2.7.6a)$$

$$\begin{pmatrix} -2x^{-3}(y^2 + z^2) \\ 2x^{-2}y \\ 2x^{-2}z \end{pmatrix} = \lambda \begin{pmatrix} -2 \\ 1 \\ 1 \end{pmatrix} \quad (2.7.6b)$$

Solving this system of linear equations assuming that $\lambda = 0$ leads to:

$$y = 0 \text{ and } z = 0 \implies \text{contradicting } y > 0 \text{ and } z > 0 \implies \lambda \neq 0$$

Solving 2.7.6b assuming that $\lambda \neq 0$ results in:

$$\begin{aligned} 2x^{-2}y &= 2x^{-2}z \implies y = z \\ -2x^{-3}(y^2 + y^2) &= -4x^{-2}y \\ -4x^{-3}y^2 &= -4x^{-2}y \\ x = y &\implies x = y = z \quad \square \end{aligned}$$

As a result, cubes are, indeed, for any arbitrary rectangular cuboid, maximizing the ratio of the length of the space diagonal to the length of the longest edge. Thus, in a three-dimensional simulation, $\sqrt{3} \approx 1.73$ is the greatest possible factor by which the size of any object's AABB can change. In the following, the greatest possible factor by which the size of a particular object's AABB can change is referred to as the object's AABB scale factor. This factor, obviously, is dependent on the specific geometry of an object (see Figure 9(b)). For example, the AABB scale factor of a cuboid, the length of one edge of which is two times the length of both the other edges (e.g., $x = y$ and $z = 2x$) is equal to $\sqrt{1.5} \approx 1.22$, whereas the AABB scale factor of a cube is equal to $\sqrt{3} \approx 1.73$.

Taken by itself, these numbers are very abstract. However, the AABB scale factor enables answering a crucial question concerning the consequences of rotating objects. The question being: given a certain hierarchy of grids and a particular AABB scale factor, under the assumption that the size of an object is statistically independent of the cell size of any grid in the hierarchy, is it possible that an object, once inserted into a grid, will never be able to leave this grid and thus will always remain assigned to it, regardless of how the object is rotated?

As Section 3.3 will show, the grid hierarchy is constructed such that the cell size of any two successive grids differs by a constant factor c . Hence, the cell size c_k of grid k can be specified as

$$c_k = c_0 \cdot c^k \quad (2.7.7)$$

Thus, all objects with a size in-between $c_0 \cdot c^{k-1}$ (included) and $c_0 \cdot c^k$ (excluded) are assigned to grid k . Objects smaller than $c_0 \cdot c^{k-1}$ are stored in grid $k - 1$, objects larger than or equal to $c_0 \cdot c^k$ are stored in grid $k + 1$.

Given the AABB scale factor x of an object (see 2.7.8a), and under the assumptions that $x < c$ and that the AABB of an object can – by rotating the object – take on any size in-between its minimum ($size_{min}^{AABB}$) and maximum value ($size_{max}^{AABB}$), the set of all objects that will – regardless of how they are rotated – always remain assigned to grid k is specified by inequality 2.7.8b.

$$x = \frac{\text{size}_{max}^{AABB}}{\text{size}_{min}^{AABB}} \quad (2.7.8a)$$

$$c_0 \cdot c^{k-1} \cdot x \leq \text{size}_{max}^{AABB} < c_0 \cdot c^k \iff c_0 \cdot c^{k-1} \leq \text{size}_{min}^{AABB} < \frac{c_0 \cdot c^k}{x} \quad (2.7.8b)$$

Clearly, if the AABB scale factor x of an object is greater than or equal to c , then this object can always be rotated in such a manner that it has to change its grid association. However, if $x < c$, then there is always the possibility that the size of the object is related to the cell sizes of the grid hierarchy in such a way that the object will – regardless of how it is rotated – always remain assigned to the same grid (see inequality 2.7.8b). For example, given a three-dimensional simulation and a grid hierarchy in which the cells of two successive grids double in size (i.e., $c = 2$ & $1 \leq x \leq \sqrt{3}$), any object could, as far as only the AABB scale factor is concerned, possibly be inserted into a grid that the object would never be able to leave. Whether an object of this simulation would, through rotation, be able to change its grid association or not would always depend on both the size and the shape of the object – but it would never depend on the shape alone.

That all being said, there are two further aspects that should be mentioned as well when studying the consequences of rotating objects and objects changing grids:

1. The computational effort for removing an object from its cell in one grid and then inserting it into the cell of another grid is practically identical to changing the cell association within a grid. As described in Section 2.6, the latter may also result from moving an object around in the simulation space.
2. As explained in Section 2.1, the time step of the simulation has to be chosen small enough so that no collision is missed because of objects passing each other. The same holds for rotations: in-between two time steps, each object will only rotate by a small fraction. Thus, in-between two time steps, the size of an object's AABB will also only change by a small fraction. Consequently, changes in the grid association of a certain object will happen rarely. For other objects they may – as discussed before – not even be possible, regardless of how these objects are rotated. All in all, taking into account everything that was mentioned so far, changes in the grid association of an object can, in general, be expected to be unlikely.

Algorithm 2: Update phase

```

1 foreach object of the hierarchical hash grid do
2   if the cells of the grid that the object is currently assigned to are still suitably sized then
3     recompute the hash
4     if the hash value changed since the last time step then
5       remove the object from its current cell...
6       ...and insert it into the cell associated with the new hash value
7     end
8   else
9     remove the object from its current grid...
10    ...and insert it into a grid with suitably sized cells
11  end
12 end

```

In summary, axis-aligned bounding boxes – and not bounding spheres – are used for the implementation of the hierarchical hash grid. In every time step, the update phase adapts the data structure to the current object distribution by taking care of moved, rotated, and deformed objects. If rigid body dynamics are used, deformations become – as already mentioned – impossible. Consequently, the final update phase will look like as outlined in Algorithm 2.

At this point, the update strategy of first removing all objects from the data structure and then afterwards reinserting them (see Section 2.6) should be mentioned once again. This strategy is not only very easy to implement but it also implicitly takes care of object movement, rotation, and deformation because the reinsertion of each object is always based on its current AABB. However, since changes in the cell or even the grid association of an object are relatively rare, and objects that remain stored in their cells are far more common, the update strategy as outlined in Algorithm 2 will, in general, be the better choice. For further comparison, see the benchmarks in Section 4.3.

3 Implementation

3.1 Data Structures for the Grid

The simulation space is – as described in Section 2.3 – uniformly subdivided into an infinite number of cubic cells, which are represented by a hash grid (see Section 2.5). The hash grid is implemented based on a linear array of cells. Indexing follows the intuitive pattern that is illustrated in Figure 10.

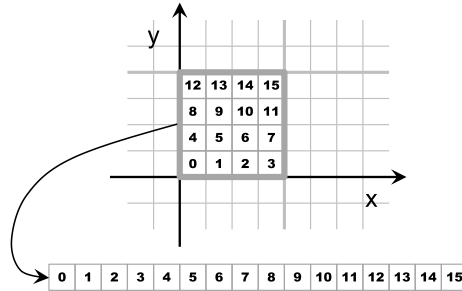


Figure 10: A two-dimensional hash grid represented by a one-dimensional linear array. Indexing in combination with a three-dimensional hash grid follows the exact same pattern.

As already discussed at the end of Section 2.5, the more cells are allocated the better because the probability of objects being assigned to the same cell of the hash grid is decreasing – except that the more cells are used the more memory is required. Thus, an obvious goal for implementing the data structure of a hash grid must be to keep the memory requirements of one single cell as low as possible. This means reducing the data to the point where no redundant information is stored. Basically, every cell must consists of ...

- ... references to all adjacent cells in order to ensure fast access to these cells at runtime and ...
- ... a container for storing the objects that are assigned to the cell.

Each cell in a three-dimensional hash grid has 26 distinct and unique neighbors. If pointers would be used in order to directly access these adjacent cells, each cell would have to store its own array of pointers. However, if not pointers but offsets are used to refer to the neighboring cells, it shows that these offsets are – as already mentioned in Section 2.5 – identical for all the inner cells of a hash grid (see Figure 11).

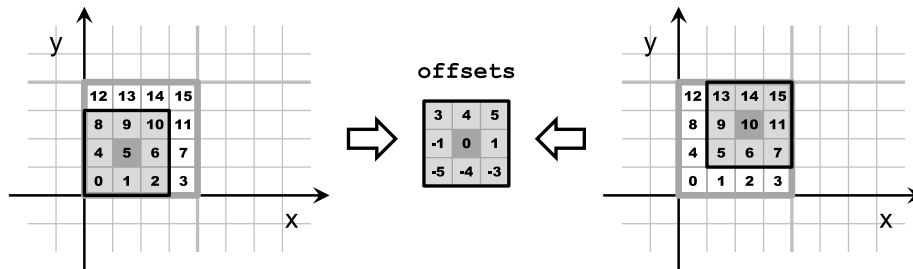


Figure 11: The offsets to the directly adjacent cells are identical for all inner cells of a hash grid.

Thus, it is sufficient to allocate one single, grid-global array that is initialized once when the grid is created and contains offsets that are valid for all the inner cells. As a consequence, instead of storing the offsets directly within the cells themselves, each inner grid cell only requires a pointer to this offset array. Thus, in order to store offset information, no additional memory has to be allocated for all the inner cells because each of these cells points to the same generally valid, grid-global offset array. Of course, in the case of an outer cell, the offset array has to be specially allocated and is only valid for this one particular cell (see Figure 12).

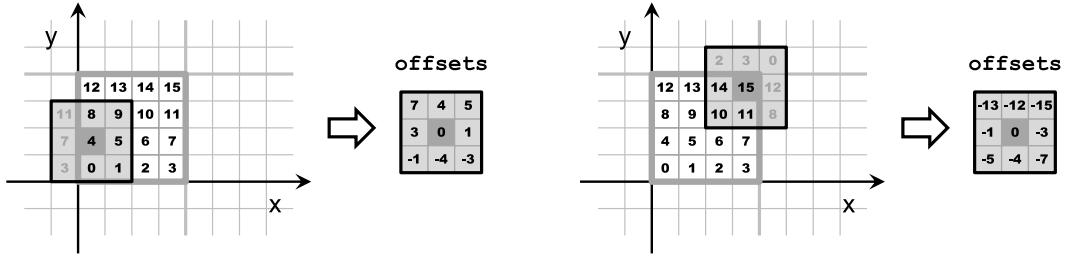


Figure 12: Each outer cell possesses a unique set of offsets for its neighboring cells.

Consequently, the required memory for storing all neighborhood relationships – which means storing a pointer in each cell and allocating all the necessary offset arrays – is mainly dependent on the number of outer cells in the hash grid. Given a hash grid of dimensions $x \times y \times z$, the number $n(x, y, z)$ of outer cells is specified by equation 3.1.1a. Hence, the percentage $p(x, y, z)$ of outer cells in a hash grid can be calculated by dividing $n(x, y, z)$ by the number of all the cells in the grid (see 3.1.1b). From this it follows that the more cells are used in each dimension the smaller becomes the number of cells at the edge of the hash grid (see 3.1.1c).

$$n(x, y, z) = 2xy + 2y(z - 2) + 2(x - 2)(z - 2) \quad (3.1.1a)$$

$$p(x, y, z) = \frac{n(x, y, z)}{xyz} = \frac{2xy + 2y(z - 2) + 2(x - 2)(z - 2)}{xyz} \quad (3.1.1b)$$

$$\begin{aligned} \lim_{\substack{x \rightarrow \infty \\ y \rightarrow \infty \\ z \rightarrow \infty}} p(x, y, z) &= \lim_{z \rightarrow \infty} \lim_{y \rightarrow \infty} \lim_{x \rightarrow \infty} \frac{2xy + 2y(z - 2) + 2(x - 2)(z - 2)}{xyz} \\ &= \lim_{z \rightarrow \infty} \lim_{y \rightarrow \infty} \frac{2y + 2(z - 2)}{yz} = \lim_{z \rightarrow \infty} \frac{2}{z} = 0 \end{aligned} \quad (3.1.1c)$$

The percentage of outer cells in a particular grid can be calculated by substituting the dimensions of this grid into equation 3.1.1b. For some of the most common sized hash grids, this results in:

$$\begin{aligned} p(\underbrace{16, 16, 16}_{4 \cdot 10^3 \text{ cells}}) &= 33\% & p(\underbrace{32, 32, 32}_{33 \cdot 10^3 \text{ cells}}) &= 18\% \\ p(\underbrace{64, 64, 64}_{262 \cdot 10^3 \text{ cells}}) &= 9.1\% & p(\underbrace{128, 128, 128}_{2 \cdot 10^6 \text{ cells}}) &= 4.6\% \end{aligned}$$

Consequently, allocating only one offset array that is valid for all the inner cells of a hash grid will efficiently reduce the required memory. Moreover, this effect will carry even more weight the larger the grid becomes.

In order to make a final estimation of the memory that is required for managing the neighborhood relationships, two aspects have yet to be considered:

1. The size of the pointer that is stored directly within each cell.

Depending on the underlying system architecture, this pointer will, for example, require 4 bytes on a 32-bit and 8 bytes on a 64-bit system.

2. The size of the offset array that is allocated only once for all inner and individually for all outer cells.

Because the offset to the cell itself – which obviously will always be equal to 0 – is also stored in the offset array (cf. Figure 11 and Figure 12), in a three-dimensional simulation this array will consist of 27 values. Storing this one offset is, of course, redundant. However, since the overhead of storing one additional value hardly carries any weight at all, but on the other hand nicely simplifies the detection algorithm (see Section 3.6 and Algorithm 4), the offset array will always also store the reference to the cell itself. Consequently, depending on the size of the signed integer type that is used, the offset array will, for example, require 108 bytes in combination with 32-bit and 216 bytes in combination with 64-bit integers. However, for most simulations, 32-bit integers are absolutely sufficient since even within a hash grid consisting of slightly more than 2 billion cells they allow to address each cell correctly. This also means that 32-bit integers are more than enough for representing offsets in hash grids of dimensions up to $1024 \times 1024 \times 1024$ cells.

Finally, taking into account everything that was discussed so far, the memory that is required for managing the neighborhood relationships within a given hash grid is specified by

$$m(x, y, z) = xyz \times \text{bytes}_{\text{pointer}} + [1 + n(x, y, z)] \times \text{bytes}_{\text{offset array}} \quad . \quad (3.1.2)$$

As already mentioned at the beginning of this section, another important aspect that must be considered in order to conserve memory is the implementation of a memory-friendly mechanism for storing objects in a grid cell. Therefore, every object has to be represented by some kind of unique identifier, which almost always will be the pointer to the object itself. As for the mechanism for storing the objects – that is, storing the object identifiers – the data structure of each cell must meet a certain set of conditions:

- Adding an object to a grid cell must be a fast operation, which means that very few instructions should be required for assigning an object to a specific cell. Moreover, the corresponding computational complexity must be of order $O(1)$.
- The same holds true for removing an object from a grid cell, which happens every time an object has to change its cell or grid association resulting from movement, rotation, or deformation (see Sections 2.6 and 2.7).
- Moreover, since iterating through all of the objects that are stored in a grid cell is the essential part of the detection step (see Algorithm 1 in Section 2.4), the complexity of this operation must be of order $O(N)$. Again, very few instructions should be required, which means the computational overhead should be as low as possible.
- Last but not least, each cell must be able to store any arbitrary number of objects. Normally, it is possible to very well estimate an upper bound for the number of objects that might have

to be stored in one single cell (see Section 3.9), so that, as a result, it becomes very unlikely that more objects than specified by this bound are assigned to the same cell. In general, however, there is no limit to the number of objects that could get associated with the same cell (see Section 3.8).

A static array would perfectly meet all of these conditions, except for the last. Hence, it is desirable that the selected data structure is based upon a static array, which is extended by the ability of dynamically increasing the size of the underlying array whenever necessary. Fortunately, the vector implementation of the standard library of C++ exactly matches these requirements [Jos99]. Since, as just mentioned, it is possible to very well estimate an upper bound for the maximum number of objects that have to be stored in the same cell (see Section 3.9), dynamic resizing of the underlying array – and therefore potentially costly reallocation operations, which generally involve the entire storage space to be copied to a new location – can be prevented almost completely by initially reserving sufficient storage capacity when the vector is created (\rightarrow `std::vector::reserve`). Furthermore, because the underlying linear array allows for consecutive memory access, iterating through all of the objects that are stored in a vector yields high performance. Inserting an object into a vector corresponds to storing this object at the end of the array, directly after the previously inserted object (\rightarrow `std::vector::push_back`). Moreover, the initially reserved capacity will almost always be sufficient so that resizing the array will virtually never be necessary (see Figure 13).

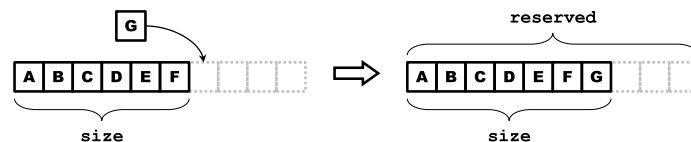


Figure 13: Inserting an object into a vector that possesses sufficient storage capacity

In order to ensure constant time ($\rightarrow O(1)$) removal for all objects, each object must, upon being inserted into the vector, memorize its index in the array, which, at that moment, is equivalent to the current size of the vector (\rightarrow `std::vector::size` – see Code Snippet 4). Consequently, since the order in which the objects are stored in the vector is irrelevant, removing an object becomes shifting the most recently inserted object from the end of the vector (\rightarrow `std::vector::back` & `std::vector::pop_back`) to the location – which, as just explained, was saved upon insertion – of the object that is going to be removed (see Figure 14 and Code Snippet 5).

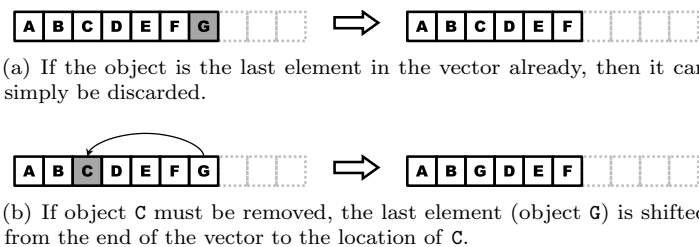


Figure 14: Removing an object from a vector

Code Snippet 4: Inserting an element into a vector so that removal in constant time is guaranteed

```
element->position = vector->size();
vector->push_back( element );
```

Code Snippet 5: Removing an element from a vector (computational complexity of order $O(1)$)

```

if( element->position == vector->size() - 1 ) {
    vector->pop_back();
}
else {
    last_element = vector->back();
    vector->pop_back();
    last_element->position = element->position;
    (*vector)[ element->position ] = last_element;
}

```

Another important aspect that must be considered when developing a memory-friendly mechanism for storing objects is the fact that, at any given time step, many cells within each hash grid will be empty with no object assigned to them at all. In fact, for every hash grid, there exists a corresponding upper bound for the number of cells that are allowed to be filled with objects, and once this limit is reached, the size of this hash grid is automatically increased (for further details see Section 3.4). As a consequence, depending on the configuration of the hash grid, objects might be assigned to no more than 10% of all grid cells – quite possibly even less. Thus, if every cell would incorporate its own container that is ready to store objects – i.e., a vector that was initialized with a sufficient amount of memory for storing several objects – a lot of memory would be wasted.

Consequently, just as with managing the offset array, each cell only stores a pointer to an object container – that is, a pointer to a `std::vector` data structure – and not the container itself. As soon as the first object is assigned to a cell, the container is allocated and initialized – and the moment the last object is removed from a cell, the container is destroyed, the allocated memory is freed, and the pointer is reset to null. Hence, this pointer being valid (i.e., not being set to null) is an indication of whether any objects are assigned to the corresponding cell or not.

Taking into account everything that was discussed so far, the memory consumption for managing object storage within a hash grid will most likely slightly fluctuate in-between different time steps, and it consists of a pointer that is stored in each cell and the memory that is required for all the currently allocated object containers. The memory consumption of such an object container depends on the system architecture, the representation of the object identifier, the configuration of the hash grid – i.e., the amount of initially reserved storage capacity – and the specific implementation of the `std::vector` data structure. Usually, this requires memory within the range of 64 to 256 bytes.

Consequently, the data structure for the cells of a hash grid (cf. Code Snippet 6) consists of a pointer to an offset array, a pointer to an object container, and an additional index that is required for realizing an efficient detection algorithm (for further explanations see Sections 3.4 and 3.6).

Code Snippet 6: Final data structure for the cells of a hash grid

```

struct Cell {
    std::vector<ObjectType>* object_container;
    offset_t* neighbor_offset_array;
    index_t occupied_cells_index;
};

```

For example, given a 64-bit architecture, and under the assumptions that the data structure of each cell requires 24 bytes, that 100 000 objects are distributed to 25 000 different cells in the same

hash grid (with no more than 8 objects being assigned to one cell), that the hash grid consists of 128^3 cells ($128 \times 128 \times 128 \approx 2 \cdot 10^6$ cells), that 32-bit offsets are used, and that each object container initially requires 256 bytes (thus being able to store more than 8 objects without the need to enlarge the underlying array), the entire required memory sums up to 64 MB:

$$\begin{aligned} \text{memory} &= xyz \times \text{bytes}_{\text{cell}} + [1 + n(x, y, z)] \times \text{bytes}_{\text{offset array}} + 25\,000 \times \text{bytes}_{\text{object container}} \\ &= 128^3 \times 24 \text{ bytes} + [1 + 96\,776] \times 27 \times 4 \text{ bytes} + 25\,000 \times 256 \text{ bytes} \\ &= 67\,183\,564 \text{ bytes} \approx 64 \text{ MB} \quad \square \end{aligned}$$

If the worst case scenario – at least as far as memory consumption is concerned – occurs, that is, every object is assigned to a different cell – which means 100 000 object containers need to be allocated – the required memory increases to 82 MB.

3.2 Hash Calculation

Hash grids apply – as described in Section 2.5 – geometrically based hashing, which means modulo operations are performed on each coordinate value of an object’s position in order to map objects that are located anywhere in the simulation space to one coherent block of cells. Assuming the grid is infinite in size, converting the position of an object into the corresponding cell index is realized by, for each dimension of the coordinate system, dividing the object’s position by the cell size of the grid (cf. Code Snippet 7). Obviously, even if the grid would be truly infinite in size, this calculation would lead to problems for negative values – that is, small negative as well as small positive values would both be mapped to the index “0”, thus creating a non-uniform mapping.

Code Snippet 7: Naive cell index calculation (in x-axis direction, allowing negative indices)

```
int x_cell_index = static_cast<int>( x / cell_size );
```

As a consequence, the index calculation in hash grids distinguishes between positive and negative coordinate values. Additionally, an included modulo operation ensures geometrically based hashing (see Code Snippets 8 and 9). For a better understanding, this approach is illustrated for both situations – that is, positive and negative values – in Figure 15.

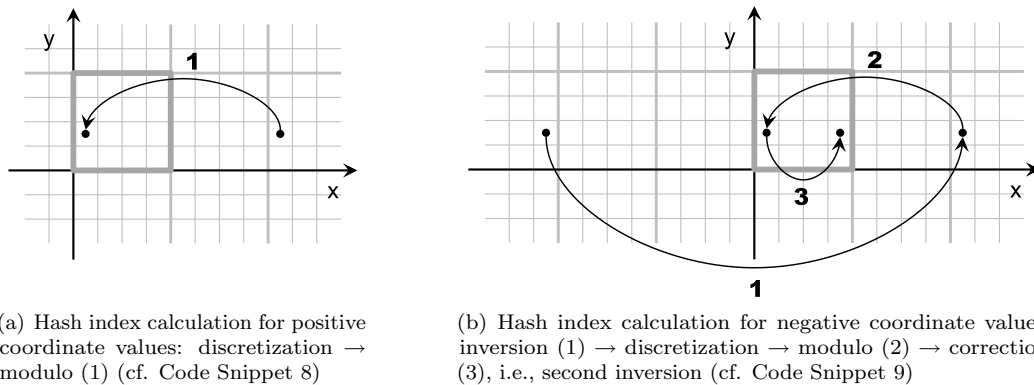


Figure 15: Illustration of modulo based hashing in x-axis direction – hashing in the other coordinate directions is, of course, carried out analogously.

Code Snippet 8: Hash index calculation for positive coordinate values

```
double i = x / cell_size;
size_t x_hash = static_cast<size_t>( i ) % x_cell_count;
```

Code Snippet 9: Hash index calculation for negative coordinate values

```
double i = ( -x ) / cell_size;
size_t x_hash = x_cell_count - 1 - ( static_cast<size_t>( i ) % x_cell_count );
```

Once a hash index is calculated in each coordinate direction, the final hash – that is, the cell index in the linear array of cells that is representing the grid (see Figure 10 at the beginning of Section 3.1) – can be calculated as outlined in Code Snippet 10.

Code Snippet 10: Calculation of the final hash value (i.e., the index in the grid’s cell array)

```
size_t hash = x_hash + y_hash * x_cell_count +
              z_hash * x_cell_count * y_cell_count;
```

However, the approach described so far has one downside: evaluating the hash for one single object in a three-dimensional simulation involves three floating point divisions and three modulo operations, both which are – on almost every current hardware architecture – extremely costly compared to additions, subtractions, or multiplications. Since calculating the hash of an object is one of the most essential and therefore mostly performed operations in the whole algorithm, this problem should definitely be addressed. Fortunately, it can be solved by ...

- ... replacing the floating point divisions with multiplications.

In order to calculate hash indices, instead of using divisions, the coordinate values are multiplied by the inverse value of the cell size. This inverse value can, and of course should, be calculated only once when the grid is created.

- ... realizing the modulo calculations with fast bitwise operations.

The modulo calculations only operate on already discretized – and thus integral – values. Moreover, as long as only integer powers of two are considered, it holds that

$$v \text{ AND } (2^n - 1) \equiv v \text{ mod } 2^n \quad \forall v \geq 0, v \in \mathbb{N}, n \geq 0, n \in \mathbb{N} \quad . \quad (3.2.1)$$

In order to benefit from this mathematical equivalence, the spatial dimensions of the hash grid must be restricted to powers of two. This prerequisite, however, does not restrict the functionality of the hierarchical hash grid in any way. (Yet, even though extremely unlikely in combination with the detection algorithm presented in this paper, this might lead to cache thrashing, which can be avoided by introducing some sort of padding to the cell array [Hoi01]. Of course, in that case, the calculation of the hash value (cf. Code Snippet 10) and the initialization of all offset arrays must be adapted)

Consequently, the final source code looks like as outlined in Code Snippet 11.

Examining Code Snippet 11 even makes it possible to study the “infinity of the grid” and its limitations that arise from the finite precision with which computers generally represent numbers.

Code Snippet 11: Final hash calculation using only fast arithmetic operations

```

inverse_cell_size = 1 / cell_size;
x_hash_mask      = x_cell_count - 1;

[...]

if( x < 0 ) {
    double i = ( -x ) * inverse_cell_size;
    x_hash = x_cell_count - 1 - ( static_cast<size_t>( i ) & x_hash_mask );
}
else {
    double i = x * inverse_cell_size;
    x_hash = static_cast<size_t>( i ) & x_hash_mask;
}

```

Assuming the floating point numbers that are used are compliant with IEEE 754, 32-bit single precision numbers are capable of accurately storing ~ 7 and 64-bit double precision numbers are capable of accurately storing ~ 15 decimal places. This enables a simple estimation: in order to unambiguously assign an object to a grid cell, the result of the implicit division of each coordinate value by the cell size must be – before being discretized – accurately representable for each digit to the left of the decimal point. Hence, it must hold that

$$\frac{\text{coordinate value}}{\text{cell size}} < 10^x \left\{ \begin{array}{ll} x = 7 & \text{if 32-bit} \\ x = 15 & \text{if 64-bit} \\ x = 33 & \text{if 128-bit} \end{array} \right\} \text{ floating point numbers are used.} \quad (3.2.2)$$

In other words, depending on the bit length of the floating point numbers that are used, for each dimension of the coordinate system, the coordinate value of an object’s position must not be greater than 10^x times the cell size of the corresponding grid. For a better understanding, some examples of simulations that are – in combination with a certain floating point type – theoretically possible (in practice, there may be additional limitations because of further restrictions in other parts of the simulation):

- Using 32-bit floating point numbers, a hierarchical hash grid is, in principle, suitable for simulations involving objects of the size of fine sand (0.2 mm) within a cubical simulation space with an edge length of 4 km.
- If 64-bit floating point numbers are used, objects of the size of a proton ($1.5 \cdot 10^{-15}$ m) can be simulated within a cubical space with an edge length of 3 m.
- In combination with 128-bit floating point numbers, even simulating objects as tiny as protons within a huge cubical space with an edge length of $3 \cdot 10^{18}$ m (~ 317 light-years) becomes theoretically possible.

Last but not least, the result of the division of the coordinate value by the cell size – which is accurately representable for each digit to the left of the decimal point – is converted to an unsigned integer. Therefore, the bit length of this integer data type must be equal to or greater than the bit length of the involved floating point numbers. For example, if 32-bit floating point numbers are used, the integer data type may consist of 4 bytes – 64-bit floating point numbers would, however, already require the usage of integers of at least 8 bytes.

3.3 Assigning Objects to the Hierarchical Hash Grid

The data structure of the hierarchical hash grid consists of a list of multiple hash grids sorted in ascending order by the size of their cells – thus realizing a grid hierarchy as described in Section 2.4. Initially, however, this list – which in C++ may be implemented using the list data structure of the standard library (\rightarrow `std::list` [Jos99]) – is empty. When the first objects are actually assigned to the hierarchical hash grid, only then suitable hash grids are created and added to the list.

Additionally, besides this list, if there are objects that cannot be assigned to any hash grid, there must also be a global container for storing these objects. Such a global object container is only necessary if the targeted simulations contain objects that are infinite in size – which means, without spanning the whole simulation space, straight lines or curves in 2D and flat or curved planes in 3D. When assigning objects to the hierarchical hash grid, when removing objects, during the update phase, and during the detection step, the objects stored in this global object container must always be treated separately. This special treatment is neither complicated nor hard to implement. Therefore, this paper will – in every presented algorithm, in every figure, and in all explanations – only concentrate on objects that are finite in size and thus it will not go into any further details regarding the issue of infinite objects.

As for assigning objects to the hierarchical hash grid, the implemented algorithm distinguishes between two different states:

1. The data structure is still empty. Thus, no hash grid does yet exist.
2. The data structure is not empty anymore – at least one hash grid already exists.

When the first object is assigned to the data structure, a new hash grid with suitably sized cells must be created, the object must be inserted, and the grid must be added to the – still empty – grid list. As already mentioned in Section 2.7, the grid hierarchy is constructed such that the cell size of any two successive grids differs by a constant factor – in the following referred to as the hierarchy factor $f_{hierarchy}$. As a result, the cell size c_k of grid k can be expressed as

$$c_k = c_0 \cdot f_{hierarchy}^k \quad . \quad (3.3.1)$$

Thus, there must be a cell size c_0 underlying all hash grids. In practice, the first object that is assigned to the data structure determines the cell size of the first hash grid that is created. The cell size of any subsequently created grid is then based on the cell size of this initial grid. Given the size of the first object that is assigned to the data structure, the cell size of the first grid that is added to the hierarchical hash grid is specified as

$$cell\ size_{initial\ grid} = size_{first\ inserted\ object} \cdot \sqrt{f_{hierarchy}} \quad . \quad (3.3.2)$$

Thus, the size of the first inserted object lies exactly midway in-between the upper and the lower bound for object sizes that are associated with this particular hash grid – which means, objects $\sqrt{f_{hierarchy}}$ times smaller than the first object are still stored in this grid, and objects $\sqrt{f_{hierarchy}}$ times larger are just too big in order to be associated with this grid and therefore are the smallest objects to be stored in the subsequent grid.

Additionally, the initial dimensions for this newly created hash grid must be determined. In general, as long as no assumptions can be made about the spatial distribution of the objects in

the simulation, an equal number of cells should be used in each coordinate dimension. However, there are simulations where such a uniform cell distribution proves to be a rather unfavorable choice (for further details see Section 3.8). Moreover, as discussed in Section 3.2 concerning the efficient calculation of hash values, the number of cells in each coordinate dimension must be equal to a power of two – thus enabling fast bitwise operations for realizing the modulo calculations. Since the size of a hash grid can be increased at runtime in order to adapt to the number of currently inserted objects (further details will follow in Section 3.4), $16 \times 16 \times 16$ is a suitable choice for the initial size of a newly created hash grid – it already consists of four thousand cells, yet only requires a few hundred kilobytes of memory.

Finally, once the first hash grid is created, the corresponding first object can be inserted. The actual insertion of an object into a hash grid, however, is covered in detail in a separate section of this paper (see the following Section 3.4).

If an object is inserted into a hierarchical hash grid that contains a non-empty list of grids, it makes no difference whether just one or already several hash grids form the current hierarchy. Furthermore, the hierarchy does not have to be dense, which means, if not every valid cell size that can be generated is required, some in-between grids are not created. Consequently, the cell size of two successive grids differs by a factor of $f_{hierarchy}^x$, with x being an integral value that is not necessarily equal to 1 (see Figure 16).

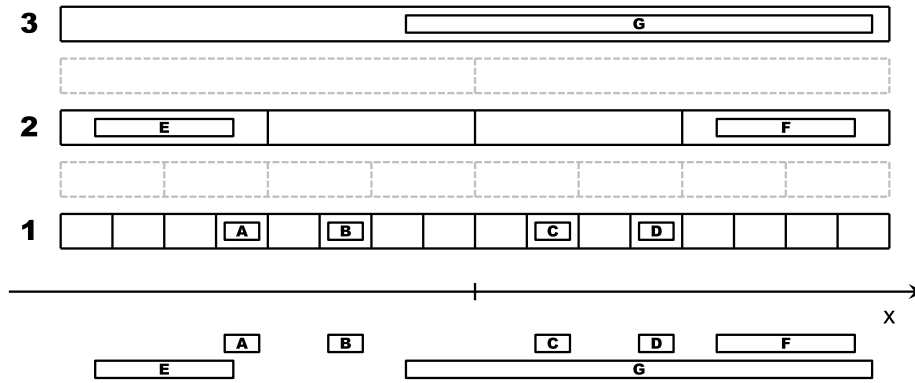


Figure 16: If, due to the spatial extents of the simulated objects, not all cell sizes that are theoretically possible (here: $f_{hierarchy} = 2$) are required, some in-between grids do not have to be created.

A particular grid is said to be suitable for an object – which means the object must not be assigned to any other grid – if

$$\frac{cell\ size}{f_{hierarchy}} \leq size_{object} < cell\ size \quad . \quad (3.3.3)$$

If an object is about to be added to the hierarchical hash grid and a suitable grid does not yet exist in the hierarchy, a new hash grid with appropriately sized cells must be created, the object must be assigned to this grid, and the grid must be inserted at the right place in the grid list. Therefore, taking into account everything that was discussed so far, the final algorithm for assigning an object to the hierarchical hash grid looks like as outlined in Algorithm 3.

Consequently, only those hash grids that are actually required will be created and added to the grid list – thus generating a perfectly adjusted hierarchy. In other words, this means the data structure of the hierarchical hash grid is, at runtime, automatically adapted to the objects in the

Algorithm 3: Final algorithm for assigning an object to the hierarchical hash grid

```

1 if no hash grid yet exists then
2   create a new hash grid with ...
3      $cell\ size = size_{object} \cdot \sqrt{f_{hierarchy}}$ 
4   ... afterwards insert the object into the grid
5   ... and finally add this newly generated grid to the grid list
6 else
7    $x = 0$ 
8   for  $G = \text{grid with the smallest cells}$  to  $\text{grid with the largest cells in the grid list}$  do
9      $x = \text{cell size of } G$ 
10    if  $size_{object} < x$  then
11       $x = x \div f_{hierarchy}$ 
12      if  $size_{object} < x$  then
13        while  $size_{object} < x$  do  $x = x \div f_{hierarchy}$ 
14        create a new hash grid with ...
15           $cell\ size = x \cdot f_{hierarchy}$ 
16        ... afterwards assign the object to the grid
17        ... and finally insert this newly generated grid directly before G into the grid list
18      else
19        insert the object into the already existing grid G
20      end
21      the object has been inserted into a hash grid with suitably sized cells, ...
22      ... thus abort further execution  $\rightarrow$  return
23    end
24  end
25  while  $size_{object} \geq x$  do  $x = x \cdot f_{hierarchy}$ 
26  create a new hash grid with ...
27     $cell\ size = x$ 
28  ... afterwards insert the object into the grid
29  ... and finally add this newly generated grid as the last element to the grid list
30 end

```

simulation. Furthermore, the hierarchy factor $f_{hierarchy}$ and the initial spatial dimensions of a newly created hash grid are configuration parameters that should be easily modifiable at compile time, thus ensuring the possibility of further adjusting the hierarchical hash grid to any simulation scenario, however unusual or atypical it may be.

3.4 Assigning an Object to a Particular Grid

Inserting an object into a particular hash grid is separated into two steps. First, the hash value is calculated, and then afterwards the object is assigned to the corresponding cell. If this cell is already occupied by other objects, which means the pointer to the object container holds a valid address and thus the container itself is properly initialized, then the object is simply added to this

already existing object container.

If, however, an object is about to be stored in a cell that is still empty, then the object container is not yet available and therefore, first of all, must be created (i.e., allocated) and properly initialized (i.e., sufficient initial storage capacity must be reserved, see Sections 3.1 and 3.9). Furthermore, the cell must be inserted into a grid-global list in which all cells that are currently occupied by objects are recorded. This list is essential for realizing an efficient detection step, the performance of which does not decrease if the number of available allocated cells increases (for further explanations see Section 3.6). The requirements for the data structure for this list are identical to the requirements for the object container of a cell (cf. Section 3.1). Thus, when using C++, the `std::vector` data structure would once again be an ideal choice, this time for implementing the grid-global list that records all object-occupied cells. Additionally, as with objects that are about to be assigned to an object container, in order to ensure constant time removal, each cell must, upon being inserted into this list, memorize its index in the internal array of the vector (see Section 3.1 & Code Snippet 4). The necessity of being able to save this index within each cell is the reason for the final data structure for grid cells that was already introduced with Code Snippet 6 in Section 3.1.[†]

As mentioned in the previous Section 3.3, in order to handle an initially unknown and ultimately arbitrary number of objects, each hash grid, starting with a rather small number of cells at the time of its creation, must have the ability to grow as new objects are inserted. Therefore, if by inserting an object into a hash grid the associated grid density – that is the ratio of cells to objects – drops below a certain point, or in other words, if the number of objects grows larger than a certain percentage of the number of cells, the following actions are taken:

1. All objects are temporarily removed from the grid. This can be realized very efficiently by using the grid-global list that records all the currently object-occupied cells.
2. All current data structures – that is, all offset arrays, all object containers, and the array of cells itself – are deleted. Afterwards, the number of cells in each coordinate dimension is doubled and corresponding new data structures are allocated and properly initialized. Thus, in 3D, the total number of cells is increased by a factor of eight, while at the same time the number of cells in each coordinate dimension is still guaranteed to be a power of two.
3. Finally, all previously removed objects are reinserted according to their new hash values.

Just like the hierarchy factor $f_{hierarchy}$ and the initial size of a newly created hash grid, the maximum number of objects – in proportion to the current number of cells – that are allowed to be stored in a hash grid is another configuration parameter that should be easily modifiable at compile time.

Consequently, as for the actual dimensions of the underlying hash grid (i.e., the number of allocated cells), different grids in the hierarchy are completely independent of each other. Thus, a particular hash grid will consist of a greater number of cells than other grids if the corresponding object sizes are occurring more frequently. This means that not only just those grids that are actually required in the hierarchy are created (see the previous Section 3.3), but also within each hash grid only as many cells as are necessary for storing all objects and at the same time ensuring a certain minimal grid density are allocated. Thus, the hierarchical hash grid is not only automatically, but also perfectly adjusted to the underlying simulation.

[†]If the update strategy of in every time step completely clearing the data structure and afterwards reinserting all objects would be used, saving this index within each cell would not be necessary since for applying this update strategy the targeted removal of specific cells is not required anymore.

3.5 Removing Objects

If the strategy of updating each object individually is applied, every change in the cell or grid association of an object resulting from movement, rotation, or deformation requires not only a mechanism for adding objects to different cells, possibly even in different grids, but also a mechanism for efficiently removing specific objects from their currently associated grid cells. Targeted object removal is also required if the underlying simulation allows objects to be destroyed at runtime, regardless of which algorithm was chosen for updating the data structure.

As already described in Section 3.1 and illustrated in Figure 14, efficiently deleting an object from an object container depends on the object to memorize, upon insertion, its location within the internal array of the `std::vector` data structure. Moreover, additionally saving the grid and the cell association of an object – which means, saving both the current hash value of the object and the pointer to the corresponding hash grid data structure – nicely simplifies the update phase. However, depending on the actual implementation of the update procedure, these two parameters are not necessarily required. They are essential, though, if a constant time ($\rightarrow O(1)$) algorithm for completely removing a random object from the hierarchical hash grid is required – possibly because the simulation allows objects to be destroyed at runtime.

If the last object is removed from a particular cell, the object container must be destroyed in order to conserve memory, and the pointer must be reset to null in order to indicate that the cell is again empty. Additionally, the cell must be removed from the grid-global list that records all object-occupied cells in the exact same manner as objects are removed from an object container.

However, in contrast to the adaptive enlargement resulting from the insertion of more and more objects, the size of a hash grid is not decreased when objects are removed. In other words, the hierarchical hash grid grows in size as more objects are inserted, yet it does not shrink when objects are removed. The automatic enlargement of a hash grid is required in order to adapt the data structure to any given object distribution at the beginning of a simulation. The removal of a significant large number of objects, however, is very unlikely. Moreover, implementing both mechanisms, one for automatic increase and one for automatic decrease, bears the risk of a constantly oscillating data structure – that is, a continuous change between the creation/increase and the destruction/decrease of certain individual hash grids.

3.6 The Detection Step

For any object distribution of a given simulation, to very efficiently detect all the occurring collisions is the core purpose of the hierarchical hash grid. Algorithm 1 in Section 2.4 already outlined how a hierarchy of grids can be used in order to find all these collisions based on checking each object individually. The final algorithm, however, utilizes the fact that all objects in the simulation are tested simultaneously. Therefore, the intra-grid detection procedure is based on cell-wise instead of object-wise collision checks. Thus, for all the objects that are stored in the same grid, detecting the same intersection twice is implicitly avoided by ...

- ... testing objects within the same cell according to Code Snippet 3 and ...
- ... checking each cell not against all but only against one half of the directly adjacent cells. Meaning, for example, that in 3D each cell is tested only against the first 13 of all its 26 neighboring cells (for a better understanding, see Figure 17).

As for the hierarchy, since when an object is processed it is only checked against objects that are stored in grids with larger sized cells, detecting the same intersection twice is implicitly prevented for all objects of two different grids (see Section 2.4 and Figure 7). In order to detect all collisions of a particular object x within another grid H , first the object's corresponding cell association C_H (i.e., its hash value) must be computed. Afterwards, x must be checked against each object that is stored in C_H or in any of its directly adjacent cells. This can be achieved by testing x against every object of each cell that is addressed by the offset array of C_H . In other words, since one entry in this array is referring back to the cell itself (cf. Figures 11 and 12), each cell in grid H that is relevant for detecting all collisions of x – that is, C_H and each of its directly adjacent cells – can be accessed very easily by simply iterating over all entries in the offset array of C_H . As a consequence, no distinction between the cell C_H and its neighboring cells is required (see Algorithm 4).

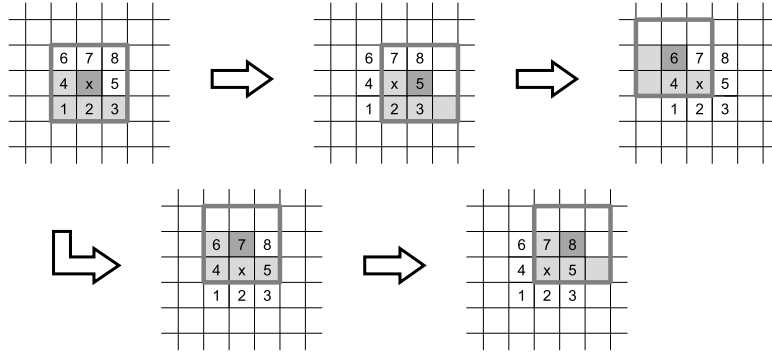


Figure 17: In each sketch, the dark-colored cell is tested against all the light-colored cells. Thus, when cell x is processed, it is checked against the cells 1, 2, 3, and 4. However, cell x is tested against the cells 5, 6, 7, and 8 not until these cells themselves are processed.

Algorithm 4: Final algorithm for the detection step of the hierarchical hash grid

```

1 for  $G \leftarrow$  grid with the smallest cells to grid with the largest cells in the hierarchy do
2   foreach cell  $C_G$  of grid  $G$  that is occupied by objects do
3     perform pairwise object checks within  $C_G$  ( $\rightarrow$  Code Snippet 3)
4     for the first half of all directly adjacent cells  $C_A$  do
5       check all objects that are stored in  $C_G$  against all objects that are stored in  $C_A$ 
6     end
7   end
8   forall objects  $x$  that are stored in grid  $G$  do
9     forall grids  $H$  with cells larger than the cells of  $G$  do
10       $C_H \leftarrow$  the cell within  $H$  that  $x$  would be associated with
11      detect all collisions of  $x$  within grid  $H$  by...
12      ...checking  $x$  against all objects within  $C_H$  and...
13      ...checking  $x$  against any object that is stored in a cell that is directly adjacent to  $C_H$ 
14      ( this is achieved by simply checking  $x$  against every object of each cell...
15      ...that is addressed by the offset array assigned to  $C_H$  )
16    end
17  end
18 end

```

Thanks to the list that keeps track of every object-occupied cell, it is never necessary – neither in the detection step nor in the update phase (cf. Algorithm 2 in Section 2.7) – to iterate through all the cells of a hash grid. Consequently, allocating a greater number of grid cells only affects the required memory, but has no negative impact on the runtime. In fact, allocating more grid cells might even increase the performance, since fewer objects might get mapped to the same cell.

3.7 The $O(N^2)$ Bound

The development of the hierarchical hash grid that is presented in this paper is motivated by the goal of implementing an efficient coarse collision detection algorithm that is suitable for a huge number of objects. If, however, the simulation only consists of a very small number of objects, simply checking each object against every other object (cf. Code Snippet 3) proves to be faster than involving the far more complex mechanisms of the hierarchical hash grid. In other words, despite its quadratic complexity, as long as there are only a couple of objects present in the simulation, the naive approach of conducting pairwise checks for all existing objects will always result in the best runtime performance (cf. benchmarks in Section 4.1, Figure 20 and Section 4.5, Figure 28).

As a consequence, a threshold is introduced, and as long as there are fewer objects added to the hierarchical hash grid than specified by this threshold value, no hierarchy of hash grids is constructed and thus no detection algorithm based on grids is applied. Instead, all objects are simply stored in a static array. Moreover, in order to detect all occurring collisions, a global all-pair test according to Code Snippet 3 is performed. However, the moment this threshold is exceeded, all objects are removed from the array and finally added to the grid hierarchy – thereby creating the initial set of hash grids which are adapted to the just inserted objects (see Sections 3.3 and 3.4). Once this has happened, the hierarchical hash grid is activated irrevocably, which means the coarse collision detection phase will continue to use the hierarchical hash grid, even if removing objects from the simulation might cause the total number of objects to again drop below the threshold.

3.8 Drawbacks & Worst Case Scenarios

Sadly, since the size of an object is defined by the length of the longest edge of its corresponding axis-aligned bounding box, if the geometry of an object is unevenly distributed, many of such objects might have to be stored in the same grid cell. The longer or the flatter certain objects are in shape, the more of these objects could be assigned to one single cell (see Figure 18). Ultimately, there is no

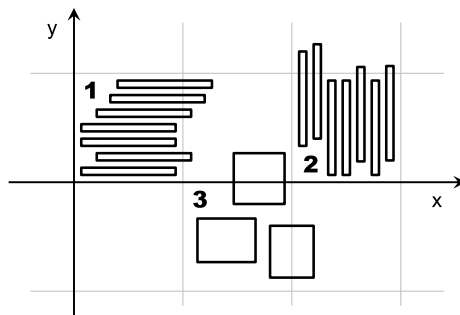


Figure 18: Since the objects assigned to cells 1 and 2 are longer in shape than the objects assigned to cell 3, cells 1 and 2 have to store seven objects each, whereas cell 3 only stores three.

limit to the number of objects that might get associated with the same grid cell. As a consequence, although extremely unlikely and virtually impossible, it is theoretically possible that all objects of a simulation might have to be stored in one single grid cell. Meaning, in the worst case scenario the detection step of the hierarchical hash grid is basically degenerated to a simulation-global all-pair object test which results in quadratic computational complexity (cf. Code Snippet 3).

Another factor that influences the overall performance of the hierarchical hash grid is the spatial distribution of the simulated objects in relation to the dimensions of the used hash grids. If it is known in advance that throughout the entire course of a certain simulation the objects will be unevenly distributed over the simulation space – maybe because in a particular three-dimensional simulation object movement is restricted to only two coordinate directions – it might be a good idea to adapt the initial dimensions of a newly created hash grid to that effect (see Figure 19). However, as long as no assumptions can be made about the spatial distribution of the objects, using an equal number of cells in each coordinate direction is the most logical choice.

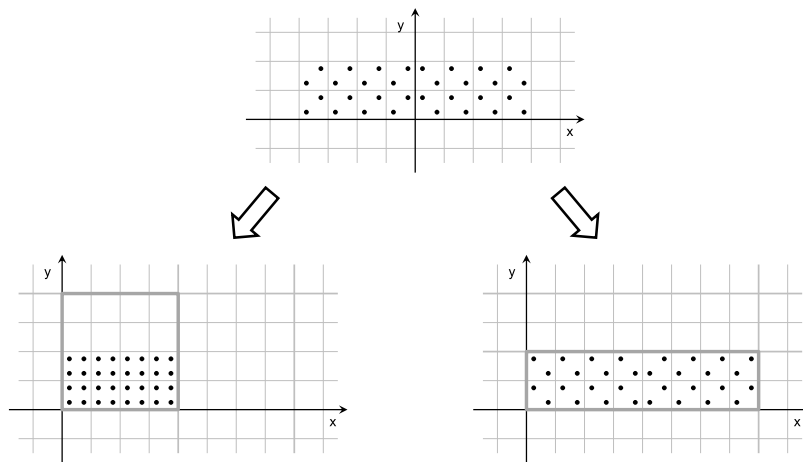


Figure 19: A hash grid of size 4×4 causes the given set of spatially unevenly distributed objects to be assigned to 8 different cells. A subsequent detection step would require 368 checks of different object pairs. However, a hash grid of size 8×2 causes the same objects to be assigned to all 16 available cells – leading to only 176 required object pair tests.

It is worth mentioning that in the final implementation, every hash grid must consist of at least four cells in each coordinate direction. Therefore, the hash grid of size 8×2 in Figure 19 would be invalid for any real application. In order for the detection step that was described in Section 3.6 to work correctly – i.e., in order to be able to implicitly guarantee that each pair of potentially intersecting objects is checked only once – every cell in a two-dimensional hash grid must have eight unique neighbors. In a three-dimensional hash grid, every cell must have 26 unique neighbors. Because of periodic boundary conditions (see Section 2.5), this means that there must be a minimum number of three cells in each coordinate direction. Moreover, since the spatial dimensions of a hash grid are restricted to powers of two (see Section 3.2, Equation 3.2.1), every hash grid must consist of at least four cells in each coordinate direction. In Practice, the initial size of a newly created hash grid is, however, already considerably larger – e.g., $16 \times 16 \times 16$ (see Section 3.3).

3.9 Computational Effort & Space Complexity

As for the required memory, the worst case scenario would be the allocation of a separate hash grid for each object in the simulation. This, of course, is only necessary if the objects strongly vary in size, so that no two objects can be assigned to the same grid. Since each newly created hash grid requires a constant amount of memory (see Section 3.3), the space complexity of the hierarchical hash grid clearly is of order $O(N)$. Moreover, if multiple objects are assigned to the same grid, the size of this hash grid – that is, the number of allocated cells – is always directly proportional to the total number of inserted objects (see Section 3.4).

As for studying the average-case computational complexity of all involved algorithms, the following two assumptions are made:

1. No matter what the number of objects in the simulation, there is a constant upper bound for the number of hash grids that might have to be created.

This proves to be a valid assumption. After all, the cell size grows exponentially (cf. Equation 3.3.1 in Section 3.3). For example, if a given hierarchy consists of 10 different hash grids and if the cells of two successive grids double in size, then the cells of the “largest” grid in this hierarchy are already 512 times larger than the cells of the “smallest” grid. If the same hierarchy consists of 100 hash grids, this factor increases from 512 to $6 \cdot 10^{29}$. Thus, in order to create all these 100 hash grids, the largest object in the simulation must be $6 \cdot 10^{29}$ times larger than the smallest one. Additionally, all object sizes that lie in-between must also exist, otherwise some hash grids might not even have to be created (see Section 3.3 and Figure 16). Thus, in practice, no hierarchy will ever consist of more than a few dozen hash grids, and therefore, in a simulation with N objects, the number of hash grids can be assumed to be of order $O(1)$ – instead of, in the worst case, being of order $O(N)$.

2. There is a constant upper bound for the number of objects that might get associated with the same grid cell.

As explained in the previous Section 3.8, in the worst case scenario, all objects must be stored in the same cell. In general, however, if the simulated objects are not extraordinarily long or flat in shape, no more than just a few objects will get assigned to the same cell. A good estimation for such an upper bound is given by Equation 3.9.1a. A fairly pessimistic and thus even better estimation is given by Equation 3.9.1b.

$$object\ bound_{cell} = 2 \cdot f_{hierarchy}^d \quad \begin{cases} d = 2 & \text{for two-dimensional simulations} \\ d = 3 & \text{for three-dimensional simulations} \end{cases} \quad (3.9.1a)$$

$$object\ bound_{cell} = 4 \cdot f_{hierarchy}^3 \quad (\text{for three-dimensional simulations only}) \quad (3.9.1b)$$

Since it is very unlikely that more objects than specified by this bound are assigned to the same cell, these estimations can, and should, be used as a guideline to how much initial storage capacity should be reserved for a newly created object container. For example, given a three-dimensional simulation and a grid hierarchy in which the cells of two successive grids double in size, initially reserving enough storage space for 16 objects will, in general, be a good choice in order to almost completely preclude the necessity for increasing the capacity of an object container at runtime.

Taking these assumptions into account, the average-case computational complexity of adding a random object to the hierarchical hash grid is of order $O(1)$ – since adding an object involves finding the appropriate grid (that is, iterating through all existing grids – the number of which is limited by a constant upper bound – and potentially creating a new one), afterwards calculating the hash value and finally directly accessing the corresponding cell in order to insert the object. Thus, in the first time step of the simulation, initially adding all objects to the hierarchical hash grid – thereby creating perfectly adapted data structures – is of order $O(N)$.

Deleting a random object is also of order $O(1)$ – since every object stores all the information that is required for directly accessing its current hash grid and its current grid cell (see Section 3.5). Moreover, as already explained in Section 3.1 and illustrated in Figure 14, objects can be removed from an object container in constant time.

Updating an object requires to compare the size of its bounding volume – which, due to rotation or deformation (see Section 2.7), may have changed since the last time step – to the cell size of the hash grid that the object is currently assigned to. Afterwards, if the current grid is still appropriate, the hash value is recomputed in order to check whether the current cell association is still valid. Both checks are, of course, of order $O(1)$. If the grid or the cell association of an object must be changed, the object must be removed from its current grid/cell and inserted into another grid/cell – both operations which are of order $O(1)$ (cf. the previous two paragraphs). Thus, the average-case computational complexity of updating all the objects that are assigned to the hierarchical hash grid (\rightarrow update phase, see Algorithm 2) is of order $O(N)$.

As for the detection step, the number of all object-occupied cells clearly is of order $O(N)$. Moreover, all these cells can be accessed directly through grid-global lists (see Section 3.4). Finding all collisions that involve one particular object requires checking this object against a constant number of cells in the same grid and against a constant number of cells in every other grid of the hierarchy (see Section 3.6). As just explained, in the average-case scenario a constant upper bound can be assumed for both the number of objects that are stored in every single cell and for the number of hash grids that form the hierarchy. Thus, the procedure for finding all collisions that involve one particular object is bounded by a constant number of operations. Consequently, taking all these statements into account, the average-case computational complexity of the detection step of the hierarchical hash grid – which is the core algorithm of the data structure that is responsible for finding all occurring collisions – is of order $O(N)$.

Bottom line, the implementation features an overall average-case computational complexity of order $O(N)$ as well as a space complexity of order $O(N)$. Starting with the next section, some selected benchmarks will, in practice, confirm the runtime behavior that was just discussed in theory.

4 Benchmarks

The specifications of the environment in which all benchmarks are carried out are as follows:

- The system is built around an Intel® Core™2 Duo E6400 CPU that is running at 2.13 GHz. The main memory consists of 2 GBytes DDR2-SDRAM.
- A 64-bit build of openSUSE 10.3 (Linux kernel version 2.6.22.19) is used as operating system.
- All programs are compiled with g++ (GCC) version 4.2.1, using the following compilation flags: `"-Wall -Wextra -Winline -Wshadow -Woverloaded-virtual -ansi -pedantic --param inline-unit-growth=150 --param max-inline-insns-single=500 --param large-function-growth=600 -O3 -DNDEBUG"`

In the scenario that underlies every benchmark that follows, all objects are randomly and homogeneously distributed within a confined simulation space. Unless explicitly specified otherwise, this space possesses the shape of a cube. Furthermore, periodic boundary conditions apply. Thus, if an object leaves the simulation space at one side, it will reenter on the other side. However, no fine collision detection will be active. After a potential intersection is detected, no further actions will be taken. Therefore, objects will intersect and pass each other – without any collision response.

Every object in this simulation moves in a random direction at a random velocity, the maximum value of which is bounded by the size of the object, i.e., in-between two time steps, no object will be able to move a distance that is greater than, for example, 20% of its size. Of course, objects that move significantly less or objects that do not move at all are equally probable – after all, the velocity of an object is randomly and homogeneously chosen from an interval that ranges from zero to the maximum velocity that is allowed for this object.

Additionally, the size of the simulation space is adapted to the number of currently simulated objects. As a result, the spatial distribution of the objects – i.e., the object density – remains constant throughout an entire benchmark. Thus, the average number of intersections per object does not change if the number of objects is increased. Of course, because of periodic boundary conditions and homogeneously distributed objects, this average number of intersections per object also remains constant in-between different time steps of a running simulation.

The simulated objects are always rectangular cuboids. Depending on the benchmark scenario, they are either all cubes and thus of the same size or, upon being created, the size of each object is randomly and homogeneously chosen from a predefined interval – so that, depending on the hierarchy factor (cf. Section 3.3 and Equation 3.3.1), multiple grids must be created. By changing the size of the objects, the object density of a particular benchmark, and thus also the average number of intersections per object within this benchmark scenario, can be adjusted very precisely.

As for the data structure of the hierarchical hash grid, the following configuration applies:

- A newly created hash grid will consist of 4096 cells ($16 \times 16 \times 16$).
- Upon initialization, the object container of a cell will reserve enough memory so that up to 16 objects can be stored before the underlying array must be resized.
- The moment the number of objects that are assigned to one grid exceeds one-eighth of the number of cells, the size of this hash grid is increased as described in Section 3.4 – that is, the

number of cells in each coordinate direction is doubled, thereby increasing the total number of cells by a factor of eight.

- Unless explicitly specified otherwise, the cell size of two successive grids differs by a factor of two ($\rightarrow f_{hierarchy} = 2$).

4.1 General Performance Scaling

In this first series of benchmarks, the basic performance scaling behavior of the hierarchical hash grid is analyzed and compared with the naive approach of a simulation-global all-pair object test. For the scenario that was chosen, in-between two time steps, no object moves a distance that is greater than 20% of its size. Moreover, the object density is adjusted so that in average there are always two intersections per object – meaning, in every time step, the number of detected potential collisions is twice the number of currently simulated objects.

As was expected, the simulation-global all-pair object test shows quadratic complexity, whereas the hierarchical hash grid shows almost perfect linear scaling (see Figure 20). Multiple grids – a result of simulating objects whose size differs by up to a factor of 12 – lead, except for a minor impact on the absolute runtime, to the same linear scaling behavior. Moreover, the effects described in Section 3.7 can be observed: if the simulation only consists of a small number of objects, conducting a simulation-global all-pair object test is faster than involving the far more complex mechanisms of the hierarchical hash grid. Since in this and every other benchmark that follows both the number of objects and the measured corresponding runtimes vary greatly, a double logarithmic scale is used in all the graphs.

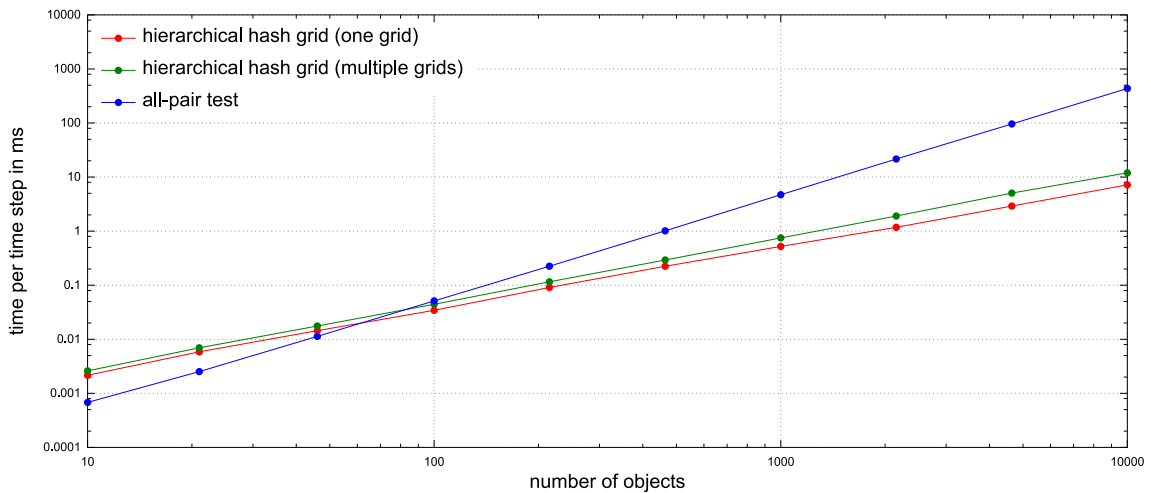


Figure 20: Performance scaling of the hierarchical hash grid (\rightarrow linear complexity) in comparison to a simulation-global all-pair object test (\rightarrow quadratic complexity).

For further benchmarks, the number of simulated objects is increased up to one million. Since a simulation-global all-pair object test would lead to disastrous runtime behavior (a single time step can be expected to require more than one hour), further comparisons are dropped and only the performance scaling behavior of the hierarchical hash grid is studied.

As for the setup phase of the hierarchical hash grid, even slightly superlinear scaling can be observed (see Figure 21). The impact of having to create multiple grids instead of only one is hardly

noticeable, and the actual time that is required for adding one million objects and thereby generating and initializing a perfectly adapted data structure (see Sections 3.3 and 3.4) is rather remarkable.

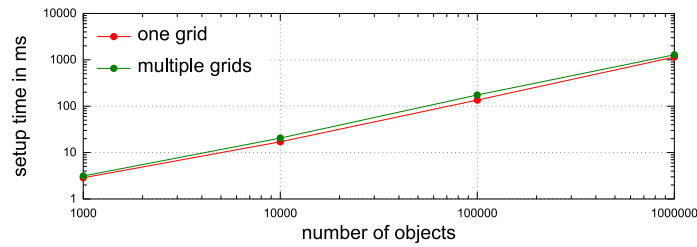


Figure 21: Required time for setting up the hierarchical hash grid, i.e., for initially adding all objects and thereby creating a perfectly adapted data structure.

In the last benchmark of this section (see Figure 22), three different things can be observed:

- Whereas the right half of the graph shows almost perfect linear scaling, the scaling in the left half is clearly worse than linear. This behavior can be explained by all object data and the data structures of the hierarchical hash grid completely fitting into the cache for small numbers of objects. As more objects are simulated, more data must be exchanged with the main memory. In this area of transition between cache and main memory, the scaling is slightly worse than linear. For large numbers of objects, the data transfer between the processor and the main memory is completely dominating memory latency and bandwidth. At this point, the hierarchical hash grid re-establishes its nearly perfect linear scaling characteristics. This behavior can also be observed in the benchmarks in Sections 4.3 and 4.4.
- The object density, and thus the average number of collisions per object, noticeably influences the performance. The more collisions occur, the more time is required in the detection step.
- The same holds true for the number of grids that form the hierarchy: the more different grids have to be used, the more time is required in the detection step.

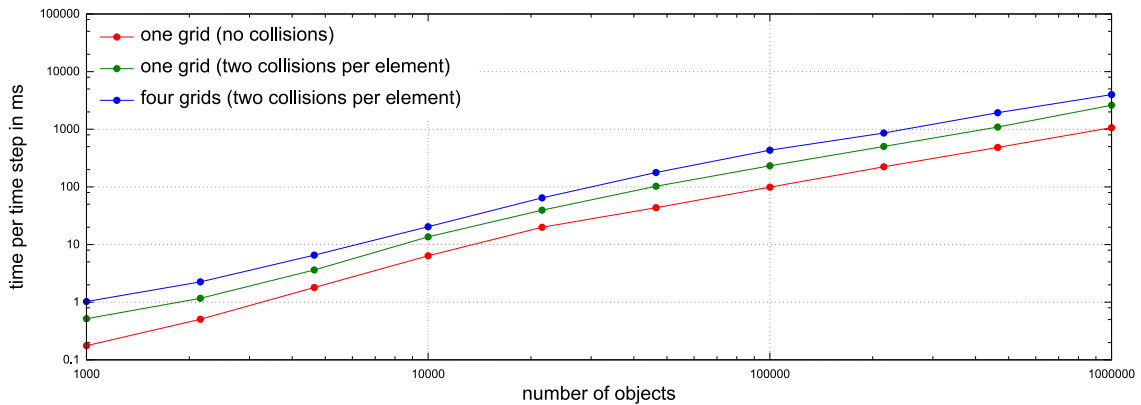


Figure 22: Performance scaling of the hierarchical hash grid for large numbers of objects in different simulation scenarios.

The short time that is actually required for simulating large numbers of objects is also worth mentioning. Even with the rather weak system that is used for all the benchmarks (see Section 4), the hierarchical hash grid only needs 10 to 20 milliseconds in order to process a simulation that involves ten thousand objects that cause twenty thousand potential intersections.

4.2 The Hierarchy Factor & Number of Objects per Cell

In this section, the influence of the hierarchy factor and thus also the influence of the number of objects per cell is studied. For this purpose, in both of the following series of benchmarks, once again, object movement is restricted to 20%, and an object density that leads to an average number of two intersections per object is chosen.

If all objects are of the same size, the following rule of thumb applies: the smaller the hierarchy factor, the smaller the size of the grid's cells in relation to the size of the objects (see Section 3.3 and Equation 3.3.1). Consequently, fewer objects are assigned to the same cell and thus the overall performance increases (see Figure 23).

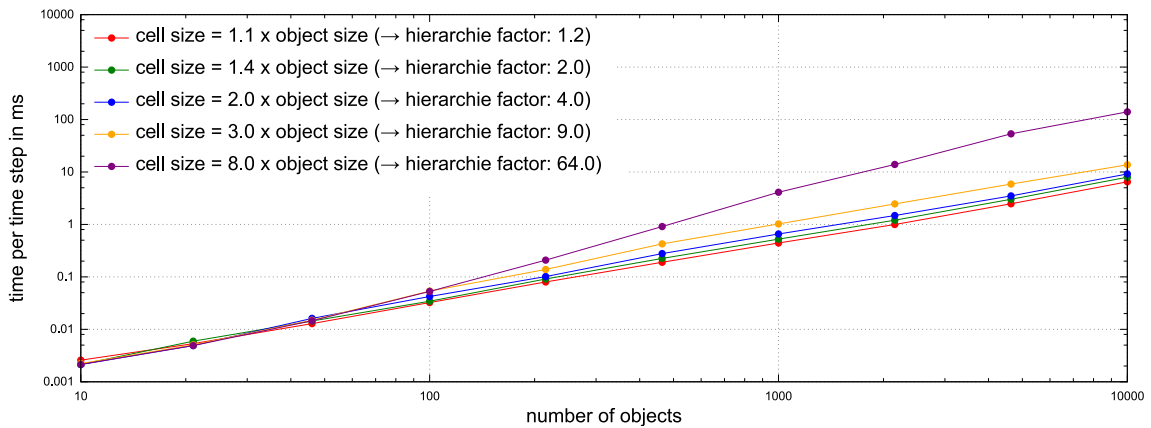


Figure 23: Performance scaling for different hierarchy factors and a simulation that only involves equally sized cubes (\rightarrow the less objects per cell, the better the performance).

If a simulation involves objects that vary in size, the observation that was already described in Section 2.4 applies: minimizing the number of objects that are potentially assigned to the same cell and at the same time also minimizing the number of grids in the hierarchy are two opposing goals. The smaller the hierarchy factor, the fewer objects are assigned to one single cell, but more grids have to be created. On the other hand, the larger the hierarchy factor, the fewer grids have to be used, but more objects are assigned to the same cell (see Figure 24). In general – after evaluating a number of different scenarios – the best choice seems to be a hierarchy factor that is equal to 2.0.

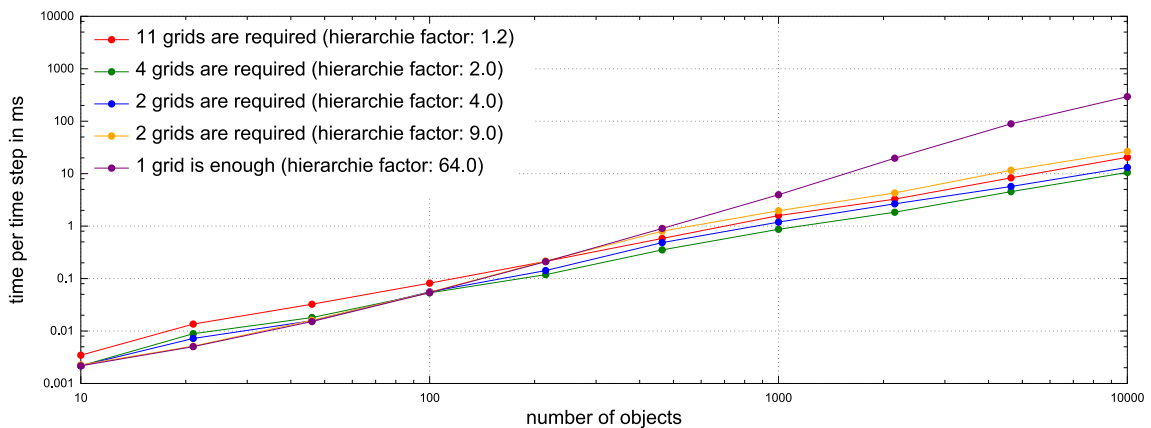


Figure 24: Depending on the hierarchy factor of the hierarchical hash grid, the same underlying simulation requires a different number of grids to be created.

4.3 The Update Phase

The benchmark in this section compares the two update strategies that were first introduced and explained in Section 2.6. In the scenario that was chosen for this purpose, once again, in every time step, there are twice as many detected potential collisions as there are objects in the simulation.

Since the performance of the strategy of updating each object individually depends on the number of objects that actually have to be updated, two series of measurements are made: one with normal (20%) and one with abnormal movement (in every time step, all objects change their cell association \rightarrow because of too much and thus invalid object movement, this scenario is impossible in any properly working simulation). The performance of the strategy of simultaneously removing all objects from the data structure, and immediately afterwards reinsert them, is, of course, independent of the degree of object movement. Consequently, under normal simulation conditions, updating each object individually proves to be 40% faster in average (see Figure 25).

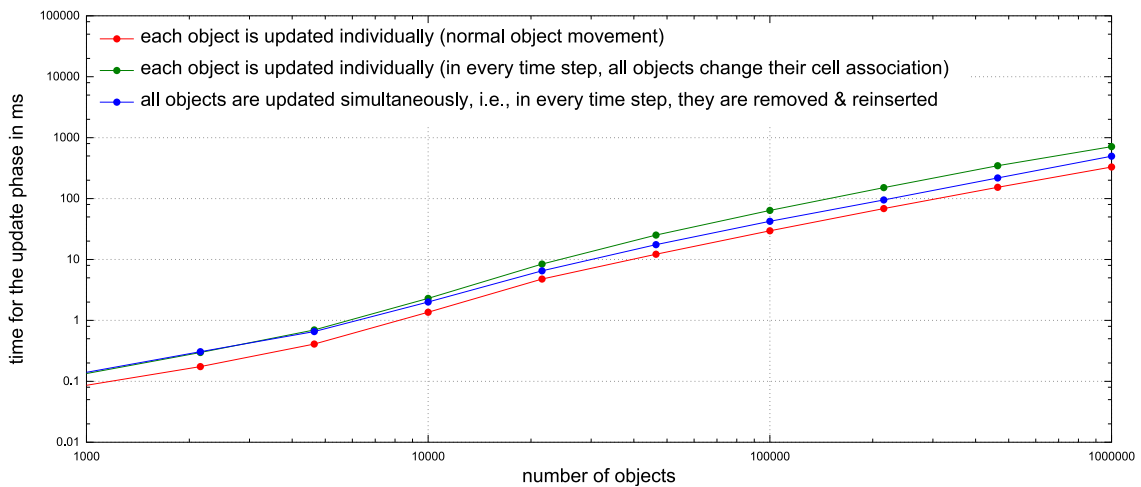


Figure 25: Under normal and valid simulation conditions, updating each object individually is faster than simultaneously removing and reinserting all objects.

Thus, as long as only the performance of the update phase is considered, updating each object individually as outlined in Algorithm 2 must be the update strategy of choice. However, depending on the object density – i.e., the number of potential intersections per object – the update phase may account for only 30% (quite likely even less) of the total time that is required for the coarse collision detection – as opposed to the detection step, which may account for 70% or more. Furthermore, the coarse collision detection itself may account for only a small fraction of the whole simulation framework. Consequently, even if updating each object individually is significantly faster than removing and reinserting all objects, this difference may hardly be noticeable in the final simulation.

4.4 Worst Case Scenarios

The benchmark in this section quantitatively determines the negative impact of the two worst case scenarios that were presented in Section 3.8. Once again, all the simulations that were built for this purpose feature twice as many potential collisions as there are objects in the simulation. Moreover, in-between two time steps, no object moves a distance that is greater than 20% of its size.

The results (see Figure 26) can be summarized as follows:

- Dealing with many long or flat-shaped objects does have a significant impact on the performance. If, for example, all the objects in a simulation are unusually long-shaped ($10 \times 1 \times 1$), every simulation step requires, with otherwise identical conditions, up to five times the time.
- Even if the spatial distribution of the simulated objects considerably differs from the cell distribution of the underlying hash grid, the impact on the performance is rather moderate. A significant impact occurs only if the objects are extremely unevenly distributed with respect to the coordinate axes. For example, when using an equal number of hash cells in each coordinate direction, the performance will drop by 50% only if there are 1024 times as many objects in x-axis as there are in y-axis and z-axis direction.
- In either case, for large numbers of objects, a nearly perfect linear scaling can be observed.

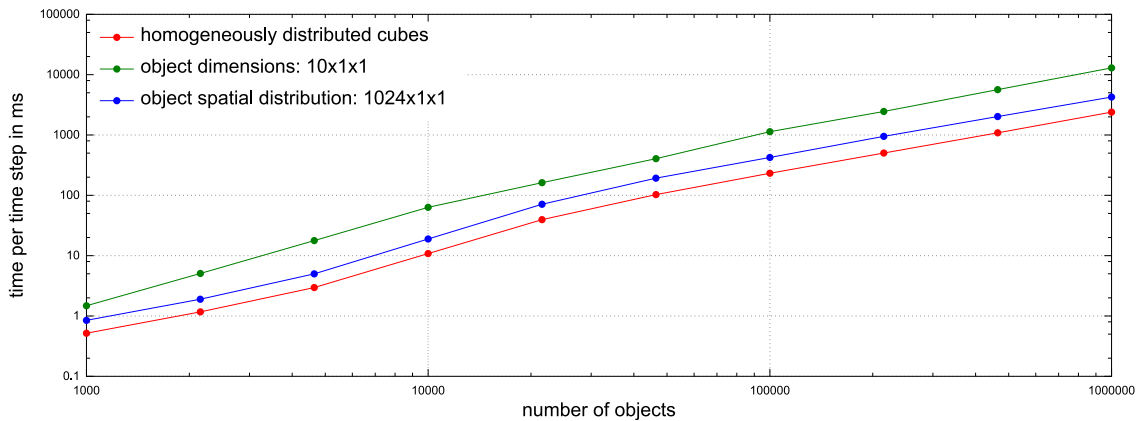


Figure 26: With otherwise identical conditions, a lot of unusually long or flat-shaped objects do have a significant impact on the performance – as well as extremely unevenly distributed objects.

4.5 The *pe* Physics Engine

This last series of benchmarks studies the performance of the hierarchical hash grid implementation within the framework of the *pe* physics engine [IR09a, IR09b]. For this purpose, the fine collision detection of the *pe* physics engine was deactivated – but otherwise the functionality of the framework remained unchanged. During the course of the following benchmarks, the performance of the hierarchical hash grid is compared with the performance of the two other coarse collision detection algorithms that are already implemented in the *pe* physics engine:

- The exhaustive search algorithm, which is a simulation-global all-pair object test.
- The sweep and prune algorithm, which is based on the concepts presented by Kenny Erleben et al. in “Physics-based Animation” [ESHD05].

The general idea behind sweep and prune is to sort all objects based on the vertices of their AABBs along each coordinate axis. After that, all overlapping AABBs – i.e., all potentially intersecting objects – can be found in linear time, $O(N)$. The performance of this method is mainly determined by the time that is required for maintaining the sorting along the coordinate axes. Maintaining this sorting can be achieved very efficiently (best case: $O(N)$) if in-between two time steps the spatial distribution of the objects hardly changes. In other

words, the higher the temporal-spatial coherence, the better the performance of the sweep and prune algorithm.

However, since the complexity of inserting a new object is of order $O(N)$, and since initially all objects are consecutively added to the *pe* framework, the setup phase has an expected quadratic $O(N^2)$ running time (cf. Figure 27) – clearly, further optimizations of the implementation of the sweep and prune algorithm should be able to reduce these complexities to $O(\log N)$ and $O(N \log N)$, respectively.

The first benchmark in this section (see Figure 27) analyzes the time that is required for initially adding all objects to the *pe* framework. As with all the benchmarks in the previous sections, an object density that leads to an average number of two intersections per object is chosen. The hierarchical hash grid shows, as already seen without the *pe* framework in Figure 21 in Section 4.1, perfect linear scaling. The sweep and prune implementation, however, results in a quadratic running time for the initial setup phase. Thus, since the setup phase already requires one minute for a simulation that involves ten thousand objects, the sweep and prune implementation in its present form is, just like exhaustive search in general, not feasible for simulations that involve huge numbers of objects – i.e., one hundred thousand, one million, or even more objects.

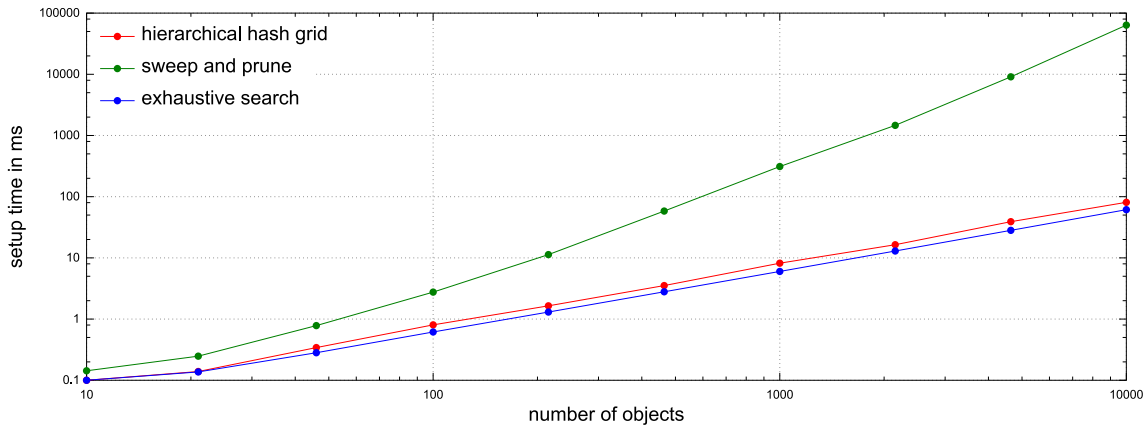


Figure 27: As far as the initial setup phase is concerned, within the *pe* physics engine, the implementation of the hierarchical hash grid shows perfect linear scaling with virtually no overhead, whereas the sweep and prune implementation results in clearly inferior quadratic scaling.

In all of the following benchmarks, the three coarse collision detection algorithms are compared based on the time that is required for one time step, regardless of the time that is required for the initial setup phase. The first series of measurements involves the usual scenario: an average number of two potential intersections per object with, in-between two time steps, no object moving a distance that is greater than 20% of its size. The results (cf. Figure 28) are as follows:

- As expected, exhaustive search results in quadratic runtime behavior.
- In the mid-section of the graph, the hierarchical hash grid shows almost perfect linear scaling. For small numbers of objects, the effects of the approach that is described in Section 3.7 can be observed: instead of hierarchical hash grids, a more efficient simulation-global all-pair object test is used. The scaling in the third section of the graph, which is clearly worse than linear, can, once again, be explained with effects that are caused by the transition from cache to main memory (the same applies to both graphs in Figure 29 and 30). For larger numbers of objects

(not shown in the graph), the hierarchical hash grid implementation of the *pe* physics engine re-establishes nearly perfect linear scaling characteristics.

- This first scenario offers too little temporal-spatial coherence for the sweep and prune algorithm to be efficient. In fact, the scenario of this first benchmark results in a devastating drop in the performance of the sweep and prune algorithm, so that even a simulation-global all-pair object test leads to superior runtime characteristics.

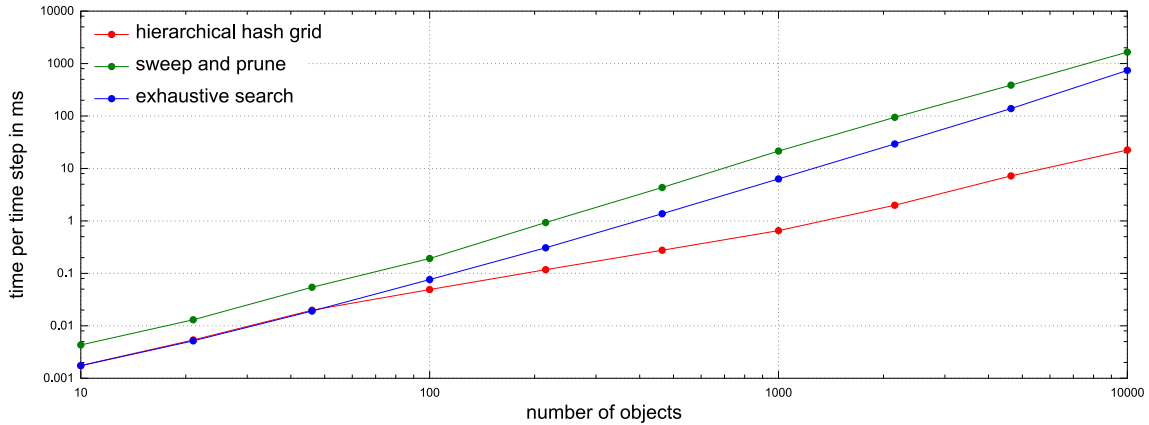


Figure 28: Whereas the hierarchical hash grid shows very good scaling characteristics regardless of the underlying temporal-spatial coherence, the sweep and prune algorithm has to deal with a devastating drop in the performance.

The scenario of the next benchmark is identical to the scenario of the last benchmark, with one exception: object movement is restricted significantly, so that high temporal-spatial coherence can be achieved. In-between two time steps, no object is allowed to move a distance that is greater than 1% of its size – most objects move even less. As a result, the performance of sweep and prune is significantly improved, yet the performance scaling is still clearly worse than linear (see Figure 29).

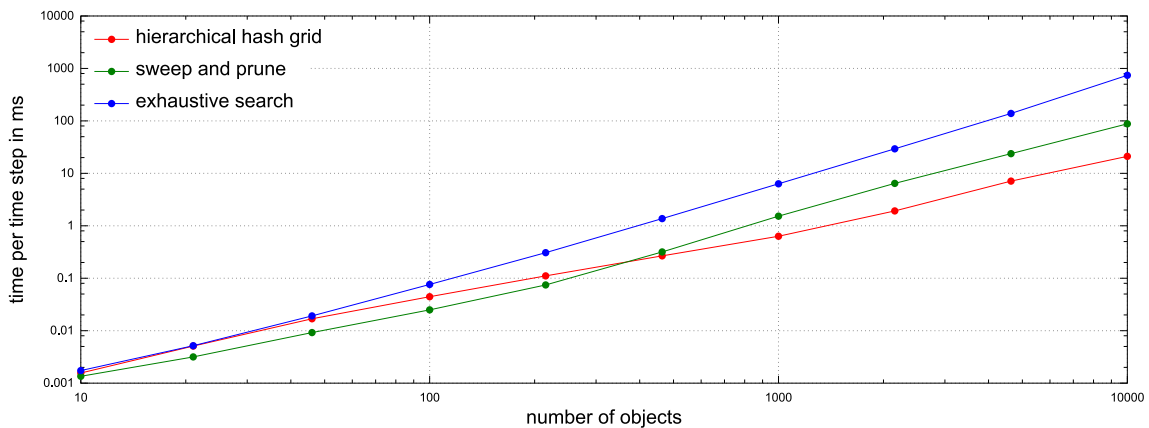


Figure 29: High temporal-spatial coherence results in a significantly better performance of the sweep and prune algorithm, whereas the performance of the hierarchical hash grid remains unchanged (yet still beating sweep and prune by up to a factor of five for large numbers of objects).

For the last benchmark (see Figure 30), the spatial distribution of the objects as well as the object movement restrictions remain unchanged. However, the size of each object is reduced by a factor of 10^5 . Thus, even if all objects are still moving, they are tremendously far apart. As a

result, no collisions occur (in fact, they are virtually impossible) and the spatial distribution of the objects can be considered as being constant (best case scenario as far as temporal-spatial coherence is concerned). Thus, sweep and prune can reach its full potential because in-between two time steps the internal data structures remain almost unchanged with virtually no need for resorting. Consequently, the sweep and prune implementation is two to three times faster than the hierarchical hash grid. Moreover, it shows the same nearly perfect linear scaling characteristics (worse than linear scaling in the last third of the graph \rightarrow transition from cache to main memory). However, the hierarchical hash grid, even if being defeated by sweep in prune in this scenario, still shows – just like in all the other benchmarks – high performance.

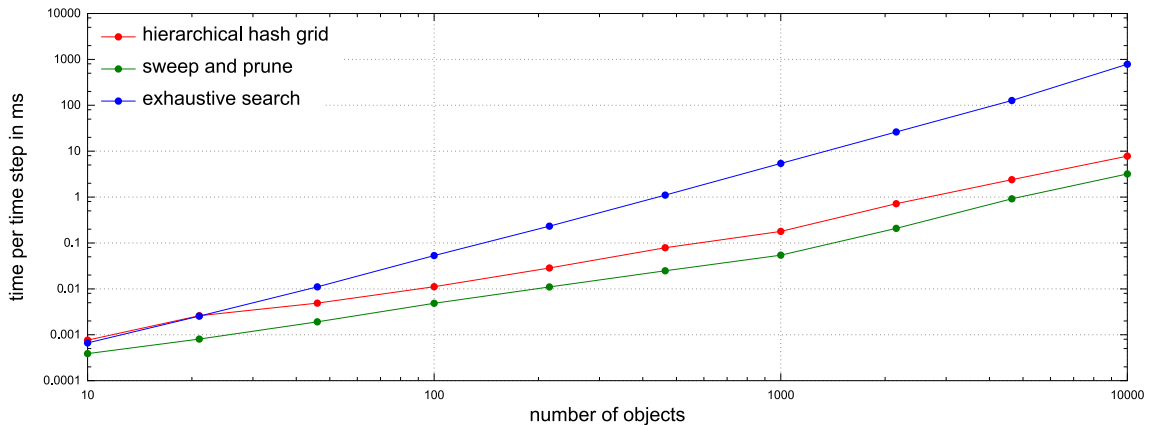


Figure 30: A simulation scenario with no occurring collisions and an extremely high temporal-spatial coherence enables sweep and prune to reach its full potential. Even if being beaten in this scenario, the hierarchical hash grid still shows high performance and the same nearly perfect linear scaling characteristics.

This section only presented some selected benchmarks that compare the hierarchical hash grid against a sweep and prune implementation. However, several more benchmarks with different scenarios (many vs. few moving objects, strongly vs. slightly varying object sizes, ...) have been performed, with the result that depending on the particular scenario, either the hierarchical hash grid or the sweep and prune implementation achieves the better performance. As long as both algorithms show linear scaling (cf. Figure 30), their actual runtime always only differs by a constant factor. Consequently, both coarse collision detection schemes generally offer high performance, with the result that they are both perfectly-suited for large numbers of objects – with sweep and prune being especially well-suited for scenarios that contain only few moving objects. If, however, a simulation contains moderate, or even high, object movement that results in a rather low temporal-spatial coherence (see Figures 28 and 29), sweep and prune implementations suffer from a devastating drop in the performance, whereas the hierarchical hash grid always maintains perfect linear scaling characteristics and therefore leads to a vastly superior performance in such simulations.

5 Conclusion

5.1 Performance & Scaling Characteristics

The hierarchical hash grid that was developed in this paper shows excellent performance and scaling characteristics in large-scale simulations. This becomes particularly apparent in comparison with sweep and prune – an algorithm that is very popular for coarse collision detection, not least because of its very good performance in many scenarios (especially large-scale simulations that contain only few moving objects). As shown in various different benchmarks, a clear advantage of the hierarchical hash grid and one of its most important features is the fact that there exist no scenarios that lead to a devastating drop in the performance. Whereas sweep and prune, for example, is extremely sensitive to object movement and temporal-spatial coherence – in the worst case scenario, this will result in massive performance drops combined with quadratic performance scaling – the hierarchical hash grid proves to be a perfect all-round solution, with no devastating drops in the performance, regardless of the underlying simulation. Even if worst case scenarios cause clearly noticeable drops in the performance, these drops are always characterized by a constant factor. Consequently, the hierarchical hash grid maintains linear performance scaling behavior, regardless of the underlying simulation scenario – the worst case scenario being a simulation that involves a high collision density (i.e., a large average number of collisions per object) and only consists of unusually long-shaped objects that vary greatly in size.

Another decisive advantage that is especially important in combination with the *pe* physics engine [IR09a, IR09b]: whereas, for example, standard sweep and prune implementations are not suited for object insertion and removal and therefore perform rather badly in these situations[†], the hierarchical hash grid enables the insertion and removal of arbitrary objects at runtime in constant time, $O(1)$. This excellent performance pays off in combination with the parallelization of the *pe* physics engine, which is based on the continuous exchange of objects between the parallel processes.

5.2 Final Words

For future work, further benchmarks with varying simulation scenarios should allow an even better understanding of the performance scaling characteristics of the hierarchical hash grid – thereby potentially enabling further optimizations. Apart from that, the hierarchical hash grid in its current form meets all the requirements that were set at the beginning of the development process: it is a coarse collision detection framework that shows high performance and linear performance scaling behavior, that possesses a memory-friendly implementation, and that is therefore capable of handling a huge number of objects. So far, the largest simulation using the *pe* physics engine in combination with the hierarchical hash grid implementation that is presented in this paper consisted of 1.14 billion interacting rigid bodies (\rightarrow 9120 processes on 9120 processor cores, with 125 000 rigid bodies per process). It was performed in May 2009 on the HLRB-II supercomputer at the Leibniz Computing Center (Leibniz-Rechenzentrum, LRZ) in Munich [LRZ].

[†]A rather recent paper by Daniel J. Tracy et al. [TBW09] introduces new features to sweep and prune, which are primarily aimed at improving the performance of object insertion and removal.

References

- [Eri05] C. Ericson, *Real-Time Collision Detection*, Morgan Kaufmann, January 2005.
- [ESHD05] K. Erleben, J. Sporring, K. Henriksen, and H. Dohlmann, *Physics-based Animation*, Charles River Media, August 2005.
- [Hoi01] A. Hoiisie, *Performance Optimization of Numerically Intensive Codes*, Society for Industrial and Applied Mathematics (SIAM), March 2001.
- [IR09a] K. Iglberger and U. Rde, *Massively parallel rigid body dynamics simulations*, Computer Science – Research and Development 23 (2009), 159-167.
- [IR09b] K. Iglberger and U. Rde, *The pe Rigid Multi-Body Physics Engine*, Technical report, University of Erlangen-Nuremberg, Computer Science Department 10 (System Simulation), May 2009.
- [Jos99] Nicolai M. Josuttis, *The C++ Standard Library – A Tutorial and Reference*, Addison-Wesley, September 1999.
- [LRZ] Leibniz Computing Center (Leibniz-Rechenzentrum), <http://www.lrz-muenchen.de>.
- [TBW09] Daniel J. Tracy, Samuel R. Buss, and Bryan M. Woods, *Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal*, IEEE Virtual Reality Conference (2009), 191-198.