GDAŃSK UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS

Student's name and surname: Mateusz Zych
ID: 131662
Second cycle studies
Mode of study: Full-time studies
Field of study: Informatics
Specialization: Algorithms and Internet Technologies

MASTER'S THESIS

Title of thesis: Efficient algorithms for collision detection in three-dimensional space

Title of thesis (in Polish): Wydajne algorytmy wykrywania kolizji w trójwymiarowej przestrzeni

Supervisor *signature*	Head of Department *signature*
dr hab. inż. Dariusz Dereniowski	

Date of thesis submission to faculty office:

GDAŃSK UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS

STATEMENT

First name and surname: Mateusz Zych
Date and place of birth: 04.11.1991, Gdańsk
ID: 131662
Faculty: Faculty of Electronics, Telecommunications and Informatics
Field of study: informatics
Cycle of studies: postgraduate studies
Mode of studies: Full-time studies

I, the undersigned, agree/do not agree* that my diploma thesis entitled: Efficient algorithms for collision detection in three-dimensional space
may be used for scientific or didactic purposes.[1]

Gdańsk, ...
signature of the student

Aware of criminal liability for violations of the Act of 4th February 1994 on Copyright and Related Rights (Journal of Laws 2006, No. 90, item 631) and disciplinary actions set out in the Law on Higher Education (Journal of Laws 2012, item 572 with later amendments),[2] as well as civil liability, I declare that the submitted diploma thesis is my own work.

This diploma t hesis has never before been the basis of an official procedure associated with the awarding of a professional title.

All the information contained in the above diploma thesis which is derived from written and electronic sources is documented in a list of relevant literature in accordance with art. 34 of the Copyright and Related Rights Act.

I confirm that this diploma thesis is identical to the attached electronic version.

Gdańsk, ...
signature of the student

I authorise the Gdańsk University of Technology to include an electronic version of the above diploma thesis in the open, institutional, digital repository of the Gdańsk University of Technology and for it to be submitted to the processes of verification and protection against misappropriation of authorship.

Gdańsk, ...
signature of the student

*) delete where appropriate

[1] Decree of Rector of Gdańsk University of Technology No. 34/2009 of 9th November 2009, TUG archive instruction addendum No. 8.

[2] Act of 27th July 2005, Law on Higher Education:

Art. 214, section 4. Should a student be suspected of committing an act which involves the appropriation of the authorship of a major part or other elements of another person's work, the rector shall forthwith order an enquiry.

Art. 214 section 6. If the evidence collected during an enquiry confirms that the act referred to in section 4 has been committed, the rector shall suspend the procedure for the awarding of a professional title pending a judgement of the disciplinary committee and submit formal notice of the committed offence.

ABSTRACT

Wide variety of applications utilizes 3D graphics, thus providing understanding of algorithms used in 3D graphics allows large group of programmers to choose suitable algorithms and data structures for their applications, such that computing resources will be used efficiently. Usually rendering 3D graphics requires affordable simulation of physics, thus this master's thesis covers efficient collision detection algorithms, which are fundamental building blocks of physics simulation responsible for maintaining solidity of objects. Notably strategies used in broad phase of collision detection are reviewed. Theoretical analysis is supplemented by results of empirical tests, from which recommendations regarding algorithm's suitability for various usage scenarios are reasoned.

Designing efficient and robust collision detection algorithms is challenging, because requirements and constraints imposed on physics engines are very rigorous. It is common that real time 3D graphics requires detecting collisions between thousands of objects in few milliseconds.

Recapping theoretical knowledge, all broad phase algorithms perform essentially the same task – they prune unnecessary intersection tests by maintaining additional data structure allowing fast collision detection. The BVH broad phase organizes objects into a hierarchy in which each node is understood as a BV encapsulating object assigned to that node and all objects assigned to its child nodes. Such construction allows effective pruning – when two BVH nodes do not intersect, intersections between children nodes do not have to be checked. The octree broad phase, classified as a spatial partitioning method, encloses all objects into a cube and slices it recursively along planes defined by X, Y and Z axes, creating an octree. Then each object is assigned to the smallest node possible, which encapsulates it entirely. Pruning comes from the fact that objects from one particular octree node can intersect only with objects inserted to its child nodes or direct ancestors of that node. A spatial sorting algorithm, the Sweep and Prune broad phase maintains sorted projections of objects' endpoints onto chosen axes. The SAP broad phase prunes all object pairs, which are not overlapping on all projections.

Results of empirical tests show that the SAP broad phase is the most efficient, with the exception of scenes with large density of objects. The BVH broad phase is almost as efficient as the SAP algorithm and has very consistent performance regardless of input geometry characteristics. Interestingly results obtained by utilizing the octree broad phase, which was added to physics engine in order to improve its collision detection performance, are significantly worse, but still much better than naive algorithm. Additional testing revealed that numerous objects straddling partitioning planes greatly reduce performance.

Interesting conclusion comes from observing maintenance cost of additional data structures required by broad phase algorithms and their performance benefits. The BVH and SAP broad phases have very good performance in terms of pruning intersection tests, but their maintenance cost is quite high. The octree broad phase has much lower maintenance cost, but the performance of pruning was greatly diminished. Thus, great way of continuing this work would be refining BVH or SAP broad phases to minimize their maintenance overhead without losing their effectiveness and performance.

Keywords:

Collision detection, Broad phase, Bounding volume hierarchy, Hgrid, Octree, Sweep and prune

STRESZCZENIE

Różnorodność aplikacji wykorzystujących grafikę 3D jest ogromna, zatem zapewnienie zrozumienia algorytmów używanych w grafice 3D pozwala dużej grupie programistów wybierać odpowiednie algorytmy i struktury danych do ich aplikacji, w celu efektywnego wykorzystania zasobów obliczeniowych. Zazwyczaj generowanie grafiki 3D wymaga symulowania fizyki, więc praca magisterska opisuje wydajne algorytmy wykrywania kolizji, które zapewniają nieprzenikalność obiektów, a także są elementarną częścią symulacji fizyki. W szczególności wykonano przegląd strategii wykorzystywanych w algorytmach broad phase. Rozważania teoretyczne zostały uzupełnione wynikami empirycznych testów, na których oparto rekomendacje algorytmów odpowiednich dla różnych scenariuszy użycia.

Projektowanie wydajnych i niezawodnych algorytmów detekcji kolizji jest wyzwaniem technicznym, ponieważ wymagania i ograniczenia nałożone na silniki fizyki są wyjątkowo rygorystyczne. Często grafika 3D generowana w czasie rzeczywistym wymaga detekcji kolizji pomiędzy tysiącami obiektów w czasie kilku milisekund.

Podsumowując wiedzę teoretyczną, wszystkie algorytmy broad phase wykonują jedno zadanie – odrzucają niepotrzebne testy kolizji dzięki utrzymywaniu dodatkowej struktury danych pozwalającej na szybką detekcję kolizji. BVH broad phase organizuje obiekty w hierarchię w której każdy wierzchołek jest utożsamiany z BV otaczającym obiekt przypisany do tego wierzchołka oraz wszystkie obiekty należące do wierzchołków potomnych. Taka konstrukcja pozwala wydajnie odrzucać niepotrzebne testy kolizji – gdy dwa wierzchołki BVH nie kolidują, to kolizje pomiędzy potomnymi wierzchołkami nie muszą być sprawdzane. Octree broad phase, zaliczany do metod partycjonujących przestrzeń, zamyka wszystkie obiekty w sześcian i przecina go rekurencyjnie zgodnie z płaszczyznami wyznaczonymi za pomocą osi X, Y oraz Z, tworząc drzewo ósemkowe. Następnie każdy obiekt jest przydzielany do najmniejszego wierzchołka, który w pełni otacza dany obiekt. Odrzucanie testów kolizji wynika z faktu, że obiekty należące do wierzchołka drzewa ósemkowego mogą przecinać tylko obiekty należące do wierzchołków potomnych lub bezpośrednich przodków tego wierzchołka. Algorytm przestrzennego sortowania, SAP broad phase utrzymuje posortowane projekcje końców obiektów zrzutowanych na wybrane osie. Wszystkie pary obiektów nieprzecinające się na wszystkich projekcjach, są odrzucane.

Wyniki testów empirycznych pokazują, że SAP broad phase jest najwydajniejszy, z wyjątkiem scen z dużą gęstością obiektów. BVH broad phase jest niemal tak samo szybki jak SAP broad phase, a jego wydajność jest bardzo dobra niezależnie od charakterystyki geometrii kolizji. Rezultaty uzyskane przez octree broad phase, który został dodany do silnika fizycznego w celu poprawienia wydajności detekcji kolizji, okazały się gorsze, lecz nadal znacznie lepsze od algorytmu naiwnego. Dodatkowe testy ujawniły, że obiekty przecinające płaszczyzny partycjonowania znacząco obniżają wydajność.

Interesujące wnioski wysunęły się po analizie koszu utrzymania dodatkowych struktur danych wymaganych przez algorytmy broad phase oraz ich zalet z punktu widzenia wydajności. Algorytmy broad phase SAP i BVH bardzo szybko odrzucają niepotrzebne testy kolizji, natomiast ich koszty utrzymania są duże. Octree broad phase jest znacznie tańszy w utrzymaniu, ale odrzucanie testów kolizji okazało się niewydajne. Zatem kontynuując tę pracę wato udoskonalić algorytmy SAP lub BVH w celu zmniejszenia ich kosztów utrzymania, zachowując przy tym ich wydajność.

Słowa kluczowe:

Wykrywanie kolizji, Broad phase, Bounding volume hierarchy, Hgrid, Octree, Sweep and prune

TABLE OF CONTENTS

LIST OF IMPORTANT DESIGNATIONS AND ABBREVIATIONS

SAP – Sweep and Prune
BVH / T – Bounding Volume Hierarchy / Tree
HGrid – Hierarchical Grid
Octree – Oct Tree
TOI – Time of Impact
BV – Bounding Volume
AABB – Axis Aligned Bounding Box
OBB – Oriented Bounding Box

1. INTRODUCTION, OBJECTIVES AND AIMS OF THESIS

3D graphics is one of the most widely used technique of computer visualization. Video games, 3D modeling, special effects in movies, physics simulations, animated films and many others rely on fast and good quality 3D computer graphics extensively. Therefore, demand on a software and a hardware allowing achieving that is huge. Base for that software and hardware are algorithms, which design is usually the deciding factor of feasible level of quality and efficiency of a whole system.

The following paragraph was written based on [5, 10]. Process of generating 3D graphics is a pipeline consisting three main phases:

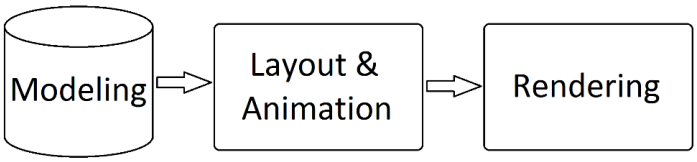


Figure 1.1. 3D graphics pipeline

- Modeling is a process of describing the shape of an object. Usually model of an object is acquired using various scanning techniques or created manually by an artist. Basically models are files stored in nonvolatile memory containing vertex data, which define shapes of polygons, colors of their sides, normal vectors, material properties and other.
- Layout and animation is a process of creating transformation for an object in form of a model matrix. This transformation includes laying out (placing) object in a world and animating its movements. A layout and an animation are both object's transformations, but the difference between them is substantial. An animation usually tries to create illusion of object's natural movements, for example walk performed by a human being. On the other hand, laying out object is relatively simple transformation, which moves object from an origin of the world to its desired position, angle and scale. Because of complicated movements required to animate an object various techniques where developed, including inverse kinematics, key framing and motion capture. In inverse kinematics, objects are moving relative to each other, constrained by limited movements of virtual joints. Key framing is a technique in which objects movements are approximated using interpolation between key frames, usually created by an artist. Lastly, motion capture allows transferring movements of real life objects into virtual world. Such a task is accomplished by tracking movements of an object using specialized sensors, which track markers attached to a moving object. Lifelike transformation of an object has to not only contain layout and animation transformations, but also has to take into account relationships between objects. For example, movement of an object can be blocked if it is surrounded by other objects, shape of an object can change after collision with other object and so on. Managing transformation of every object in a world is done by a physics engine. This software module utilities numerous algorithms allowing its efficient operation and is a key part of an application utilizing computer generated 3D graphics.
- Rendering is final process that results in a visual representation of a model, created in a modeling phase, transformed by data prepared in a layout and animation phase. In the rendering phase, geometry, texture and transformation data is processed in the rendering pipeline resulting 2D displayable image – the frame buffer.

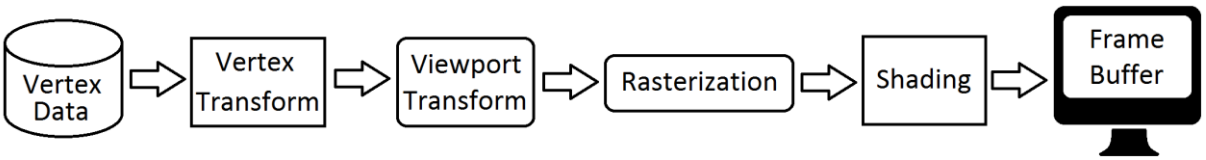


Figure 1.2. The rendering pipeline; figure created based on [6]

Initially vertex data is transformed by a model-view-projection matrix, then vertex coordinates are normalized and transformed by the viewport transformation. Geometry processing is then followed by rasterization and shading. Rasterization is a decision problem determining which pixels of a 2D image corresponds to object's geometry. Finally, shading assigns desired color to each pixel of a 2D image. In rendering phase all data prepared in previous stages is used to produce result – the frame buffer filled with colors. Rendering pipeline usually requires huge amount of simple calculations, thus efficiency of this phase is mainly dictated by raw computing power of parallel machine in form of a GPU. Many calculations are executed by specialized hardware blocks processing chunks of data. Because of that, not much improvement can be done by software rather than hardware. This can be seen in substantial performance gap between mobile and desktop GPUs.

1.1. Collision detection system architecture and its role in a physics engine

The section 1.1 was created based on chapters 3, 4, 6 and 9 from [2]. The layout and animation phase is managed by a physics engine. This software module allows creating an illusion of realistic, lifelike objects in a virtual world. Generally, physics engines perform physics simulation, which enables objects to move according to movement defined by its animation, but also take into account various object's and world properties. Such properties usually include shape, mass and its center, coefficient of friction of an object's material and force of gravity present in the world. To accomplish such a complex task, physic engines are divided into separated modules responsible for key phases of physics simulation.

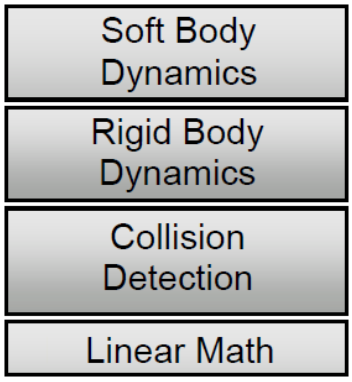


Figure 1.3. Physics engine architecture; figure created based on the first scheme from the chapter 3 in [2]

Linear Math is the most low-level module (other modules use its functionality), supplying implementations of following features:

- fundamental mathematical operations and data types:
  - vectors with operations including calculating dot and cross products
  - matrices with operations including calculating transposition matrix, inverse matrix and determinant of a matrix
  - Cartesian and barycentric coordinates with operations and conversions between them
  - lines, rays, half-spaces and various polygons including triangles and quads with constructing functions from proper data
- geometric queries:
  - triangle orient queries
  - point plane query
  - polygon / polyhedral convexity query
- geometric algorithms:
  - calculating convex hull using quick hull or Andrew's algorithm
  - determining the minimum distance between two convex sets using GJK algorithm

The number of needed mathematical data types with operations, geometric queries and algorithms varies from project to project, because it depends on chosen methods. Above list contains what is needed in usual case. Worth mentioning is the fact that vast majority of geometric queries are implemented using few mathematical operations and usually are used as a basis for geometric algorithms.

Collision detection is key task performed by a physic engine, because it allows objects to interact properly with each other maintaining illusion of rigidity of objects. Without collision detection characters in games would penetrate through walls, bullets fired from guns would not hurt opponents and so on. As usual, detecting collisions is a process divided into separate phases, which create coherent pipeline.

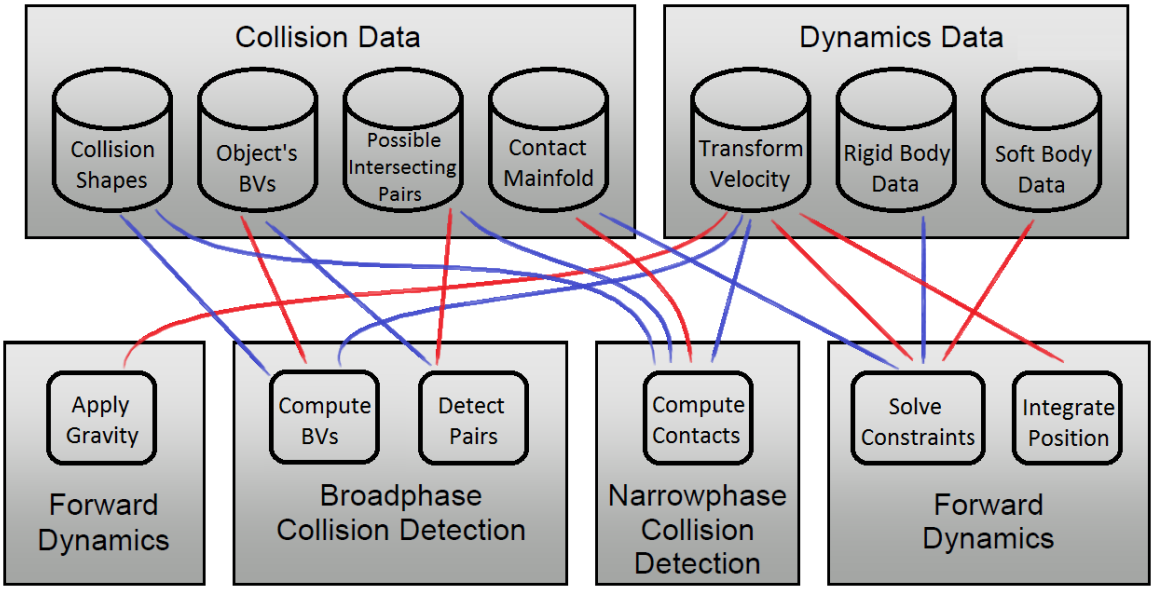


Figure 1.4. Physics engine pipeline with data flow;
figure created based on the second scheme from the third chapter in [2]

The whole physics simulation is executed stepwise with short time interval, usually lasting 16ms. In such a step physics pipeline is run. Firstly, on each object in the world the force of gravity is applied, transforming their velocities. This is followed by a collision detection phase divided into broad and narrow phases. Generally, collision detection searches though space of all pairs of objects to find all intersecting pairs of objects. In some cases, in addition to finding all pairs of colliding objects, collision detection system will calculate contact manifold and predict time of impact (TOI). Contact manifold is a section of a space defined by the set of joint points of contacting objects. Term time of impact describes expected time after which two objects approaching each other will collide. In a typical case objects collide with few other objects in their local surroundings, consequently collision detection is divided into two phases. The broad phase eliminates as many unnecessary collision tests as possible, then in the narrow phase object's geometry is used to perform intersection tests. Such a strategy is usually very efficient, because typically, number of actual collisions is very small fraction of all possible collisions and broad phase eliminates large number of unnecessary intersection tests. In broad phase geometry of bounding volumes (BVs) are used for finding pairs of potentially intersecting objects. This strategy improves search for colliding pairs of objects, because when two BVs are not intersecting objects in their interiors are also not intersecting. For BVs to be profitable, their complexity has to be smaller than object's geometry. Example of very common BV is axis aligned bounding box (AABB). Encapsulating objects in BVs improves detection of intersecting pairs by a constant factor, but to have truly fast collision detection broad phase utilize specialized algorithm, which allow collision detection to have smaller computational complexity in average case. Compared to exhaustive search though whole space of objects pairs, which has quadratic complexity in terms of number of objects in the world, broad phase algorithm should be much faster. After computing BV of each object and detecting all intersecting pairs in broad phase, collision detection system moves to the narrow phase. Main task of narrow phase is to check for intersections of potentially colliding pairs. In opposite to broad phase, which check for possible collisions using BVs, narrow phase utilizes more geometrically complex collision shapes to find actual collisions. Therefore, fast collision detection system should prune possible collisions pairs set to minimum in order to minimalize amount of work in narrow phase. Worth mentioning is a fact, that narrow phase is usually quite large module in terms of number of algorithms required to be implemented. The reason for this collision matrix, which contains algorithms for collision tests between every type of object supported in physics engine, is quadratic growth of the number of collision algorithms in terms of number of supported types of collision shapes. In the final step of physic engine pipeline, constraints of objects movements are solved and new objects positions are calculated from their velocities. Constraints can be modeled as predefined ways in which a pair of objects can move relatively to each other, effectively decreasing degree of freedom of object's movement. Constrains are also created as a collision response, which is way of reacting to collision with another object. Such response could cause soft object to deform, solid object to fall apart, block its movement or bounce it back, depending on properties of an object. Solving constraints requires data of soft and rigid bodies, including their mass, inertia, flexibility and so on. In the last phase, a physics pipeline a model of Newton's dynamics is used to calculate final positions of objects using calculus.

Rigid Body Dynamics is a module allowing objects in the world to be represented as solid bodies. It is implemented on top of functionality provided by collision detection module, adding new type of objects and solver module for its constraints. This task requires implementation of a physics model

of a rigid body involving calculations of forces acting on bodies and consequently their velocities with respect to their masses. Thanks to invariant shapes and impenetrability of rigid bodies, they can be represented only by their origin, shape and mass. Rigid bodies are used to simulate dynamics of a terrain, buildings, rocks, walls and so on. Compared to soft bodies, rigid bodies are much cheaper in terms of amount of computing power required to be properly simulated.

An extension of a rigid body dynamics is soft body dynamics allowing modeling bodies, which can alter their shape depending on a situation, meaning that any two points in object can change their relative distance to each other. Such bodies are notably used as a representation of cloths, ropes, hair, surfaces of a fluids, elastic volumetric bodies and deformable bodies. There are various approaches to simulate soft body dynamics, which usually are approximation of a correct physical behavior or even are based on simple methods producing visually acceptable results. The following description of methods used to represent soft bodies is based on [12]. In the spring-mass model, soft bodies are represented as a graph of nodes being characterized by mass and edges linking nodes via springs obeying some version of Hook's law. This graph can be a polygonal mesh serving as a surface of an object or can model internal structure of an object. Another method, which is more physically accurate, is finite element simulation in which shape of an object is filled with small solid elements fitting together. Simulation of body dynamics calculates stresses and strains in each element with respect to properties of object's material.

1.2. Collision detection requirements

Numerous applications are using 3D computer graphics, which created great demand for collision detection systems. These applications set very rigorous requirements for collision detection systems. Following requirements are cited from the second chapter in [8].

Quite large fraction of applications need 3D computer graphics generated in real time, which means that application has to generate one frame per very short time interval, usually lasting 16ms. Not only entire 3D graphics pipeline has to be executed in that small period, including physics calculations, collisions detection, rendering, but also logic of an application. Usually 10-30% of available time is dedicated to collision detection, which is even smaller amount of time. In addition to short period available for collision detection, also consistency and worst case scenarios in detecting collisions are very important, because application should generate frames without sudden drops. Worth mentioning is a fact that despite of exponential growth of hardware computing power, relative amount of time intended for collision detection is rather constant and complexity of virtual worlds also increases. This suggests that rigorous requirements for collision detection systems will not disappear.

Another very significant requirement for collision detection system is an ability to work efficiently even for large number of objects. In modern simulations, world can contain thousands of objects and collision detection system is expected to handle them reasonably well. Not only operating on large number of objects is desired, but also handling objects with great geometric complexity is a must. Additionally majority of scenes contain objects with large disproportion of their relative sizes and fast moving objects.

In addition to performance and environment simulation requirements there are equally important requirements concerning numerical and geometrical robustness and correctness. All geometric algorithms are calculated with some level of numerical accuracy and precision, resulting from utilizing

approximation algorithms and numerical errors. This can lead to catastrophic events in simulations, where even slighted errors can cause interpenetrating of objects, collapsing under the ground, and so on. Generally, numeric and geometric stability of an algorithm is desired or in some cases required.

Lastly memory consumption of collision geometry and additional data structures should be minimal, or at least kept under some predefined limit. Memory constrains are not that strong for personal computers, but game consoles and mobile devices usually have limited amount of memory. It is also worth mentioning that amount of memory intended for collision detection system is usually fraction of a memory available on a device. As these devices are improving in time, their memory capacity grows as well, however proportion of memory intended for collision detection to total memory available on a device stays constant.

These requirements impose strain on collision detection algorithms, demanding their efficiency, versatility and robustness.

1.3. Goal and scope of the work

Considering vastness of topics involving collision detection, it has been agreed that this work will focus on the review of broad phase algorithms. The main goal is to analyze strategies used in broad phase algorithms by familiarizing with available literature. Additionally, properties and suitability for various applications of these algorithms will be considered. Complementary to theoretical analysis, numerous practical tests verifying their correctness will be implemented and executed.

“Real-Time Collision Detection” by Christer Ericson was chosen as a main source of theoretical knowledge, which is widely respected position, by some considered the best collision detection related source. To perform empirical tests existing physics library had to be chosen in order to focus on studying broad phase algorithms instead of time consuming implementation of a whole physics engine from the start. The Bullet physics library was chosen because of its clear and powerful object oriented interface. In favor of the Bullet library was its zlib license and usage in numerous commercially successful projects.

Empirical tests were designed to compare suitability of broad phase algorithms for various environment configurations. Unfortunately, the Bullet library supports only BVT, SAP and brute force algorithms, making spatial partitioning algorithms not available. In order to compare all types of broad phases, implementation of the Octree based algorithm will be added to the Bullet library. The Octree algorithm was chosen form several spatial partitioning schemes because according to [8] it is suitable for dynamic scenes in opposite to Binary Spatial Partitioning (BSP) trees, which are primarily used for static terrain.

2. OVERVIEW OF BROAD PHASE ALGORITHMS

In order to have truly fast collision detection, proper broad phase algorithm has to be utilized. Before in depth description of broad phase algorithms, it is extremely important to realize general limitations of optimizations and consider relationships between environment and algorithm properties.

The efficiency of a broad phase algorithm can be evaluated using two distinct metrics: how fast set of potentially intersecting object pairs can be created and how many pairs are in that set. There are two extreme solutions and definitely both of them are not be optimal. On one side of a spectrum, a set of potentially intersecting object pairs can be filled with all possible pairs of objects. This results in relatively quick creation of the set, but will cause narrow phase to be extremely slow. On the other side of a spectrum, there is much better approach, in which every pair of objects is tested for potential collision using BVs of these objects. This approach will result the quickest narrow phase, but the broad phase itself will be very slow. Optimal broad phase should be quick enough to do not slow down whole collision detection, but effective enough to create reasonably small set of object pairs. To some extent this could be a tradeoff, between how small the set is created and the time spend in a broad phase. Example of such a relationship is a broad phase algorithm trimming number of unnecessary intersection tests at the cost of using some data structure organizing objects spatially. Because of imbalance between testing intersection of BVs and testing intersection of objects geometry, broad phase algorithms creating fairly small sets are desired. Generally, in order for a broad phase algorithm to benefit collision detection, it has to be more efficient than brute force algorithm testing all possible pairs of objects.

For most algorithms additional data structures has to be initialized and maintained as the world changes and they should not generate such a cost, that brute force algorithm is faster. In some cases, algorithm resulting larger set of an object pairs with cheaper data structures will yield better results than algorithm with expensive data structures creating small set of object pairs. This balance is affected by the algorithm itself and the ratio between cost of BVs intersection test and object's geometries intersection test. Additional data structures and calculations have to prune enough unnecessary tests to result performance gains, otherwise they will slow down collision detection and make code unnecessary complicated.

Properties of environment in which collision detection is performed can have impact on the performance. The significance of that impact depends on various factors including broad phase algorithm, number of objects, diversity of objects sizes, velocities of objects, density of objects in the world and others. For some specific configurations, one broad phase will be more suitable than the other will and even can be tuned to perform better than in an average case. In order to specify which factors have large influence and which have not, understanding of underlying strategies used in broad phase algorithms is crucial.

All broad phase algorithms are built on top of strategy utilizing special property of data structure organizing objects spatially, which allows it to prune as much unnecessary intersection tests as possible. A number of broad phase algorithms have been proposed over the years with different types of data structures expressing spatial arrangement of objects. Predominant types of techniques elaborated in literature are Bounding Volume Hierarchies, Spatial Partitioning and Spatial Sorting.

2.1. Bounding volume hierarchies

The following description of bounding volume hierarchies is created based on the chapter 6 from [8]. Bounding volumes hierarchies as the name suggests is a group of methods utilizing some variant of a bounding volume hierarchy to accelerate the collision detection. Such a data structure organizes objects into a hierarchy, in which each node has assigned BV containing all objects belonging to the node. In each node, all of its objects are split against children of that particular node. There are many possible variants of such a hierarchy differing in creation rules. In the simplest variant, root node contains all objects present in the world and world's BV. Then collection of these objects is split against children nodes until one object is present in a node, called leaf node.

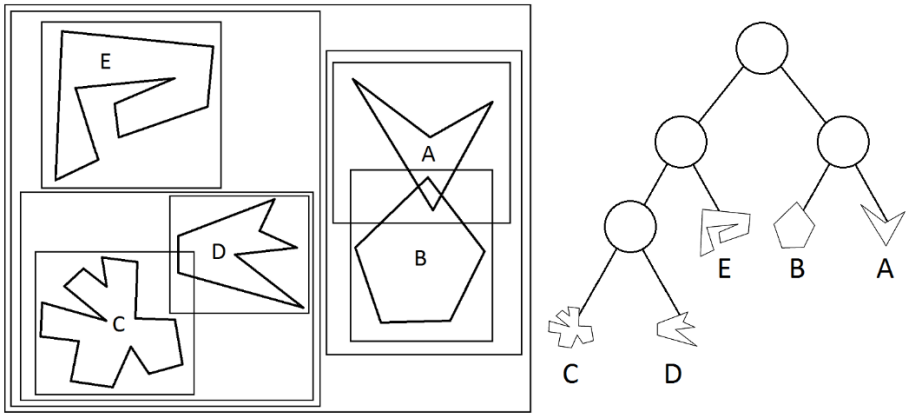


Figure 2.1. Example of binary bounding volume hierarchy; figure based on the figure 6.1 from [8]

Creation rules can assume some limitations on depth of the BVH, which cause leafs to have collection of objects instead of single object. This limit can be dynamically adjusted and have different values in different sub-trees. Additionally, number of node children can vary and be dynamic. For example, higher node can have more children than lower nodes. Also building hierarchy can be performed in number of different ways. Most of them can be categorized as a top-down, bottom-up or insert constructions. Algorithms also vary in splitting rules, describing how to decide which object should be assigned to which child and traversal rules, which organize hierarchy exploring. All these differences create quite large space of possible broad phases using hierarchies, but all of them utilize the most valuable property of BVs. When a node of a BV is tested for intersection and the result of a test is negative - intersection is not present, all children of that node do not have to be examined for intersecting. Such pruning of a node's sub-tree can reduce broad phase complexity in average case. BVT not only prune large number of unnecessary intersection tests, but also avoids expensive tests on complex object's geometry thanks to utilizing BVs. BVHs can also be used in narrow phase as a representation of complex object's geometry. Also, it can be beneficial for an application to use two different BVHs. The first one used for expressing spatial arrangement of static terrain and the second one storing temporary positions of dynamic objects. This strategy tries to exploit temporal coherence of a scene, by assuming that objects do not change their states rapidly.

2.1.1. Desired characteristics

The first step needed to understand strategies used in algorithms building and traversing BVHs is to list desired properties of data structure itself:

- When traversing deeper into a BVH distance between objects contained in that nodes should decrease, that is objects form nodes within a relatively small sub-tree should be near each other. As a consequence of that relation, sizes of BVs should decrease, as a hierarchy is traversed from root to leaves.
- Volume of each node's BV and summary volume of BVs in a hierarchy should be minimal. Tightness of BVs minimizes number of false positive intersection tests, in which BVs are intersecting but underlying geometry of objects' are not.
- Majority of attention should be allocated on upper parts of a hierarchy, since cutting off sub-trees corresponding to nodes located higher in a hierarchy is much more profitable compared to cutting off sub-trees corresponding to nodes located in lower parts of hierarchy. The reason for that is upper sub-trees contains much larger number of objects than lower sub-trees.
- Balance of BVH should be maintained with respect to node and geometrical structure. Balancing node structure ensures that height of a hierarchy is close to logarithmic in terms of number of nodes in a hierarchy and nodes are distributed evenly between sub-trees. Geometrical balance of a hierarchy is responsible for dividing evenly objects and their total volume between sub-trees. Balance is a hierarchy is vital to performance gains arising from utilizing BVH, because it allows maximizing pruning capabilities.
- Flexibility of adapting BVH to static terrain and dynamic objects is desired. For static terrain, hierarchy should be able to be constructed in a preprocessing step and use much more accurate and precise construction methods. On the other side of a spectrum, for dynamic objects, hierarchy should be fast enough to insert and remove objects in real time. It is also desired that building strategy could be adapted to scenario between these extreme situations. For example, when BVH is built for a terrain in the next level in some video game, the loading screen should not keep player waiting too long, but take enough time to allow effective pruning.

The above list of desired characteristics was based on the section 6.1.1 from [8] and is not exhaustive. Some specific applications can have additional requirements and as a consequence of that additional or little bit different characteristics would be desired.

2.1.2. Design issues

One of the most fundamental decision affecting characteristics of a BVH is a strategy managing degree of nodes. Interesting relationship can be specified between tree degree and its height, assuming constant number of nodes.

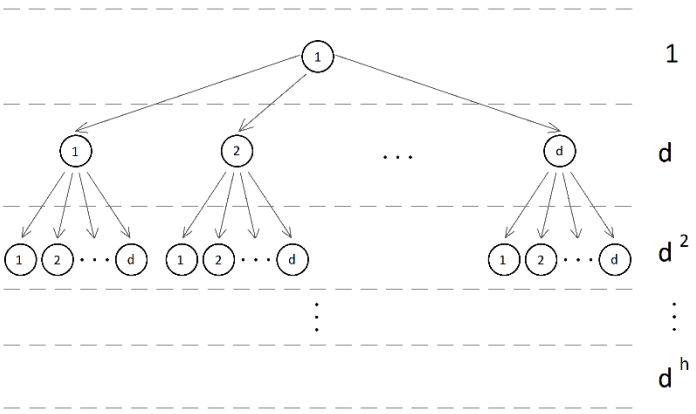


Figure 2.2. Visualization of d-nary tree and number of nodes at each level

D-nary tree is a type of a tree in which every node has up to d children (its degree is not higher than d+1). Relationship between degrees of d-nary tree, its height and number of nodes in a tree can be expressed with following equation.

$$\boldsymbol{n = \sum_{i=0}^{h} d^i = \frac{d^{h+1} - 1}{d - 1} = \frac{dl - 1}{d - 1}} \tag{1.1}$$

n – number of nodes in a tree [nodes]

d – number of children in a node [nodes]

h – height of a tree [edges]

l – number of leafs in a tree [nodes]

Assuming that the number of nodes in a tree is constant, equation (1.1) shows that relationship between tree degree and its height is inversely proportional. Relation in (1.1) proves correctness of intuitive thinking, that trees of higher degrees have smaller height. That positive impact of increasing degree of a tree on its height minimizes root-to-leaf traverse time. Formula (1.1) can also be transformed to formula (1.2), which shows that the percentage of number of internal nodes in total number of nodes in BVH decreases as degree of a tree increases.

$$\boldsymbol{\frac{n - l}{n} = \frac{l - 1}{dl - 1} = \frac{d^h - 1}{d^{h+1} - 1}} \tag{1.2}$$

n – number of nodes in a tree [nodes]

l – number of leafs in a tree [nodes]

d – number of children in a node [nodes]

Minimalizing amount of internal nodes in a tree reduces amount of construction work required to build a BVH. However not all properties of high degree BVH have positive impact on their performance. Increasing degree of BVH increases complexity of work that has to be executed in each node, when traversing and building this data structure. Building BVH with high degree nodes is more complex than building BVH with low degree nodes, because all possible arrangements of splitting collection of objects in each node between its child nodes grow exponentially. Traversing BVH is also more costly for high degree BVH, since when two BVHs are tested for collisions in each node all of its children has to be checked for intersection. According to the section 6.1.3 from [8], not much research has been conducted on optimal degree of BVHs, but practical implementations shows that binary trees are sufficient.

2.1.3. Building Strategies

The following section was written based on the chapter 6.2 from [8]. Arraigning data in a tree usually is utilized because of logarithmic complexity of traversing form root of a tree to its leaf, which logically represents some method of extracting valuable information out of that data structure. In order to maintain that property, tree has to be balanced, otherwise it can lose that property. That situation can be observed in binary search trees (BSTs), where special rotation operations are present, ensuring that data structure is balanced. That is not the case with BVHs, because rotation operations are too expensive. Because of that, it is vital for construction algorithms to produce balanced trees, allowing effective pruning. Unfortunately, lack of balancing and the fact that node arrangement is direct representation of arrangement of object in a world, logarithmic height of a tree cannot be assured, but is expected in general case. Methods creating BVH can be categorized into three major schemes:

- Top-down construction starts with set of all objects and split it against child nodes of a root node. Then for each subset algorithm is calculating BV and assigns it to proper child node. Then whole process repeats itself until subsets contain single object or some predefined limit.

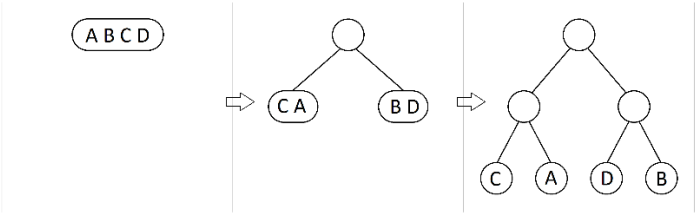


Figure 2.3. Top-down construction of a tree; figure based on the figure 6.2 (a) in [8]

  The biggest advantage of this method is ease of implementation, which main task is to loop through each object in a set and evaluate to which subset object should be assigned. However, these methods, in general case, do not yield best results.

- Bottom-up construction approaches problem in the opposite way. The algorithm starts with all objects assigned to individual sets and then merges them into larger sets. Sizes of these sets are equal or smaller than degree of nodes in BVH. The procedure repeats merging sets until singe set containing all objects is created and assigned to the root node.

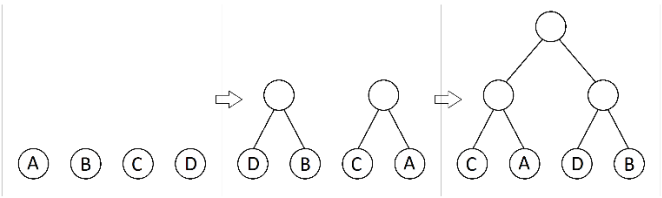


Figure 2.4. Bottom-up construction of a tree; figure based on the figure 6.2 (b) in [8]

  Compared to top-down method, constructing BVH using bottom-up scheme, in general case, creates BVH with better node organization, but is more difficult to implement.

- Insertion construction is a method, which divides construction into discrete insert operations, which add node to proper place in a BVH. Insert operation chooses node placement based on minimization of some cost measurement. Building BVH starts with empty BVH, which grows as objects are added to the tree one at a time.

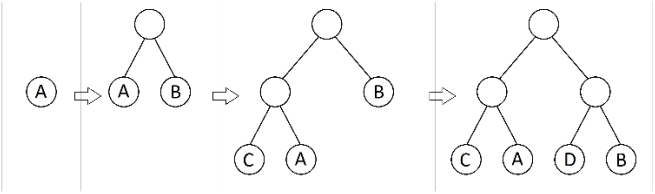


Figure 2.5. Construction of a tree using insertion operations; figure based on the figure 6.2 (c) in [8]

Differing property of insertion method is its ability to start building BVH without knowledge of entire input compared to top-down and bottom-up methods, which require all objects to start the algorithm. This can benefit startup time of collision detection system and exploit temporal coherency of scene or parts of a scene.

The top-down BVT construction can be implemented as a recursive procedure, which starts by creating new node in a tree, then builds BV of objects assigned to this node. Finally, if lower limit of the number of objects per node has been satisfied algorithm assigns collection of objects to just created node, otherwise algorithm divides collection of objects into partitions and calls itself for left and right child with corresponding partition. Listing 2.1 illustrates how such a procedure could be implemented using C++ language.

```
template <unsigned int MIN_NUM_OF_OBJS_PER_NODE = 1>
void TopDownConstructionBVT(BVTNode*& bvtNode, const Collection<Object>& objects)
{
    bvtNode = new BVTNode();
    bvtNode->leftNode = nullptr;
    bvtNode->rightNode = nullptr;
    bvtNode->objects = objects;
    bvtNode->boundingVolume = ComputeBoundingVolume(bvtNode->objects);

    if (objects.Size() <= MIN_NUM_OF_OBJS_PER_NODE)
    {
        bvtNode->type = NodeType::LEAF;
    }
    else
    {
        bvtNode->type = NodeType::INTERNAL;

        Pair<Collection<Object>> objectsPartitions = PartitionObjects(objects);

        TopDownConstructionBVT(bvtNode->leftNode, objectsPartitions.left);
        TopDownConstructionBVT(bvtNode->rightNode, objectsPartitions.right);
    }
}
```

Listing 2.1: Logic of top-down construction algorithm in C++; code based on the listing form the section 6.2.1 in [8]

Aside from computing BV for collection of objects only partitioning of objects decides how good or bad top-down construction is performing. Because of that, extra care should to be put in designing good objects partitioning algorithm. Assigning n objects to k partitions is equivalent to n-element variation of k-element set with repetitions allowed, but fortunately, suboptimal solution is good enough. Partitioning can de designed to provide desired characteristics of BVH:

- minimal sum of volume of children BVs to decrease likelihood of intersection between BVs
- minimal maximal volume of children BVs to improve behavior in the worst-case scenarios
- minimal intersection volume or even maximal separation between children's BVs to minimize probability of intersection between BVs

- split objects against children equally to provide best hierarchy balance

Usually god hierarchy balance is most valuable. Consequently, methods partitioning set of objects using hyper-plane are most common. Hyper-plane separates objects into two partitions. The first partition contains objects located in the positive half-space and the second partition contains objects located in the negative half-space. Objects straddling hyper-plane have to be assigned based on other criterion. One solution is to split straddling objects, but this could immensely increase number of objects. Also splitting objects themselves is quite computationally complex. Another method splits objects based on their centroids. That way, objects can always be assigned to one of two partitions and it is ensured that objects will not protrude by more than half of their size.

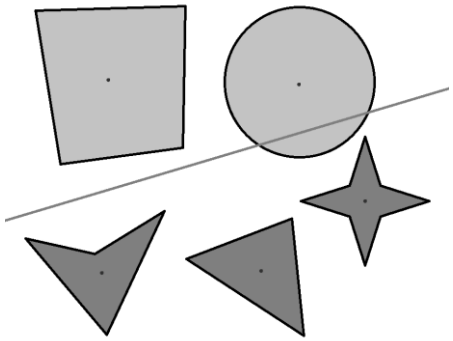


Figure 2.6. Example of splitting line in 2D partitioning objects into light gray and dark gray partitions

For clarity, partitioning is split into two steps choosing separating axis and position along that axis. Choosing separating axis can be accomplished utilizing some variant of hill climbing algorithm, but in practice, this method is too expensive. Typically, separating axis is chosen form small predefined set of presumably good axes, which commonly consists:

- Axes of global coordinate systems
- Axes of objects' local coordinate systems
- Axes associated with BVs of objects, like axis of OBBs
- Axes from parent's set or oriented BVs containing objects
- Axis along two most distant points in defining objects
- Axis along which variance is the largest

Most of mentioned axes are included, because they are easy to create and perform computations on. Second step after proper axis has been chosen is to choose position along that axis. Similarly choosing optimal position is has to be done as an iteration through predefined set of points including:

- Median of the centroid coordinates (object median)
- Mean of the centroid coordinates (object mean)
- Mean of the extent of BV projections (spatial median)

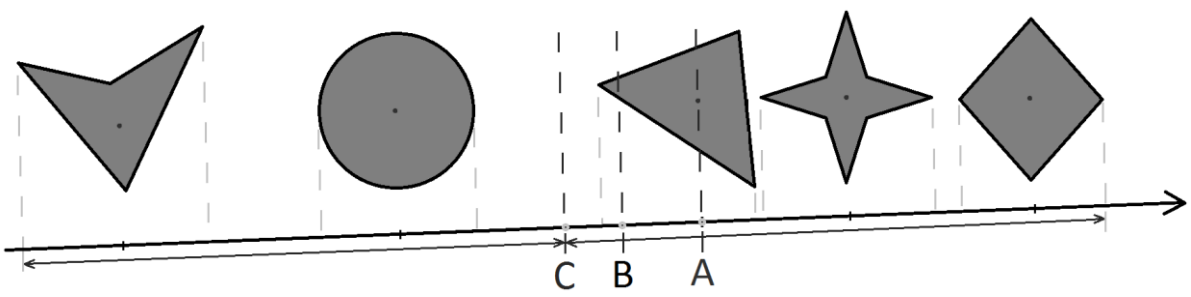


Figure 2.7. Examples of splitting using (A) object median, (B) object mean and (C) spatial mean; figure based on the figure 6.3 from [8]

Of course, there are many other approaches to selecting splitting hyper-plane based on some statistical properties or using some iterative methods. Concluding, top-down construction is quite fast to build BVH and easy to implement, but does not yields best hierarchies.

Alternative method to top-down construction is bottom-up construction, which can be implemented as an iterative procedure. In the initial step, the algorithm encapsulates all objects in BVs and sets them as leafs of BVH. Then procedure iteratively merges these objects into sets using a clustering rule, also known as a merging criterion, and builds for them BV. This iterative process stops when all objects are in one common set, which becomes root of BVH. It is possible to stop prematurely creating forest instead of one tree. Listing 2.2 shows example implementation of bottom-up construction in C++ language.

```
void BottomUpConstructionBVT(BVTNode*& bvtNode, const Collection<Object>& objects)
{
    Collection<BVTNode*> nodesCollection;

    for (int i = 0; i < objects.Size(); ++i)
    {
        BVTNode* node = new BVTNode();

        node->type = NodeType::LEAF;
        node->leftNode = nullptr;
        node->rightNode = nullptr;
        node->objects = Collection<Object>::OneItemCollection(objects[i]);
        node->boundingVolume = BoundingVolume(node->objects);

        nodesCollection.Insert(node);
    }

    while (nodesCollection.Size() >= 2)
    {
        Pair<BVTNode*> nodesToMerge = FindNodesToMerge(nodesCollection);

        BVTNode* node = new BVTNode();

        node->type = NodeType::INTERNAL;
        node->leftNode  = nodesToMerge.left;
        node->rightNode = nodesToMerge.right;
        node->objects = node->rightNode->objects + node->leftNode->objects;
        node->boundingVolume = BoundingVolume(node->objects);

        nodesCollection.Remove(nodesToMerge.left);
        nodesCollection.Remove(nodesToMerge.right);
        nodesCollection.Insert(node);
    }

    assert(nodesCollection.Size() == 1);
    bvtNode = nodesCollection[0];
}
```

Listing 2.2: Example of bottom-up construction algorithm in C++; code based on the listing from the section 6.2.2 in [8]

Listing 2.2 presents general idea behind bottom-up construction, but is not optimal algorithm. Refined version comes from the observation that finding nodes to merge and recomputing BV in each step can be reduced to maintaining priority queue of tuples. This priority queue is just refined set of nodes to merge, consisting of tuples instead of actual nodes. Tuples themselves are composited from nodes and

nodes that are best choice for merging with them. In tuples, there is also kept a volume of bounding box for merged node, which is used as a priority in the queue. Thanks to implementing priority queue as a binary search tree, finding nodes to merge becomes very cheap pop operation from the top of queue. Independent from scheme used in the bottom-up algorithm, clustering rules play important role in every bottom-up construction. Most common clustering rule chooses nodes to merge, which has minimal BV encapsulating sum of their sets of objects. Other popular merging criterion assumes that the closer the BVs of two nodes are the more probable is that their common BV will be minimal. Consequently, that merging criterion chooses the nearest nodes, which can be efficiently found using k-dimensional trees. Other merging strategies can also be used, notably utilizing minimum spanning tree. In such an approach, objects become nodes in fully connected graph with weights on edges proportional to distances between BVs of nodes. For such constructed graph, MST is constructed using Prim's or Kruskal's algorithm. By iteratively removing edges from MST, connected components of MST become clusters of objects.

Insertion construction is an approach, which splits construction of BVH into insert operations. At first BVH is empty, then, as objects are added to the BVH by insertion operation, hierarchy grows. Each insert operation searches through set of nodes and chooses that node, which minimizes insertion cost and adds new node to BVH. This scheme has several parts, which are not specific and can be implemented in different ways. First ambiguous part regards creating set of nodes to search. Simplest approach to this problem is to create search set from encountered nodes during traversing BVH from root to leaf, descending to node, which BV encapsulating its set of objects summed with new object will be smaller. More expensive approach utilizes priority queue with all nodes in BVH and traverses hierarchy according to best node on top of queue. Second unclear part of insertion schema is the insertion cost. Most common method of calculating an insertion cost of the new object at an insertion node is to sum the volume of bounding box encapsulating said object and the increase in volume of insertion node's ancestors' bounding boxes, caused by hypothetical insertion. Last unobvious part is actual addition of an object to BVH at specified node. Adding object to binary BVH can be realized by creating one leaf node containing object itself and one parent node.

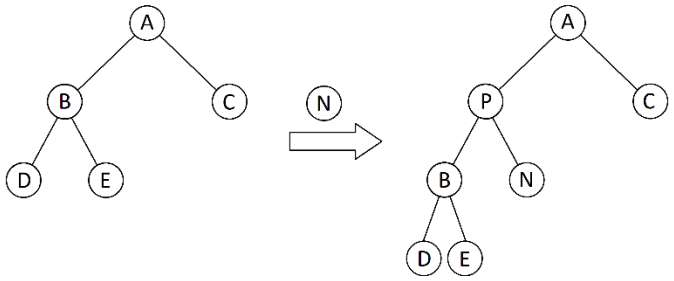


Figure 2.8. Insert object into BVH creates leaf node N and parent node P

Assuming that previously mentioned methods are used in insertion operation listing 2.3 presents example of insert operation implemented in C++.

```
void InsertConstructionBVT(BVTNode*& bvtNode, const Object& object)
{
    Collection<Object> objectCollection = Collection<Object>::OneItemCollection(object);
    Collection<BVTNode*> rootLeafPath;

    BVTNode* insertNode = bvtNode;
    while (insertNode != nullptr)
```

```
    {
        BoundingVolume left(insertNode->leftNode->objects + objectCollection);
        BoundingVolume right(insertNode->rightNode->objects + objectCollection);
        if (left < right)
        {
            insertNode = insertNode->leftNode;
        }
        else
        {
            insertNode = insertNode->rightNode;
        }
        rootLeafPath.Insert(insertNode);
    }

    BVTNode* bestNode = bvtNode;
    for (unsigned int i = 0; i < rootLeafPath.Size(); ++i)
    {
        if (InsertCost(rootLeafPath[i], object) < InsertCost(bestNode, object))
        {
            bestNode = rootLeafPath[i];
        }
    }

    AddToBVT(bestNode, object);
}
```

Listing 2.3: Example of insert operation implemented in C++

Insertion algorithm inserts objects into a BVH depending on objects themselves. This property allows BVH to express spatial arrangement, but also makes structure of a hierarchy input dependent. This can cause algorithm to create, for some inputs, degenerated hierarchies. Such degenerated BVH prunes unnecessary intersection very ineffectively. In order to avoid this situation, it is best if input objects are randomly shuffled. The insertion creation is an approach allowing on-line adding and removing nodes. This unique property makes this approach very suitable to create and update BVHs holding dynamic objects.

2.1.4. Hierarchy Traversal

A lot of attention has been dedicated to designing good quality BVHs, but not much was said about how BVH is used to detect colliding objects. The reason for extensive description of building BVHs is inability to optimize collision detection in a meaningful way by poorly built BVHs. Also, previous knowledge is quite useful to design methods utilizing built BVH to accelerate collision detection. It is vital to declare that when a BV is not intersecting with BV of a node in BVH, also none of its children intersects with that BV. Pruning whole sub-trees is what makes BVH suitable for collision detection. Conventionally collision detection utilizing BVHs are built on top of procedure, which takes two BVHs and finds all collisions between them, but does not find collisions inside them. Listing 2.4 presents declaration of that procedure in C++.

```
void BVTCollision(Collection<Pair<Object&>>& collisions, BVTNode* nodeA, BVTNode* nodeB);
```

Listing 2.4: BVTCollision procedure declaration

The BVTCollision is very powerful and flexible procedure, because it can be used for collision detection between two objects represented as BVHs. Assuming these BVHs are not self-intersecting, the BVTCollision will detect all collisions. This assumption is fairly reasonable, because objects collide with other objects rather than with themselves. Even though the BVTCollision procedure detects only

collisions between two BVHs and omits collisions inside each BVH, it can be used to detect internal collisions of one BVH. To achieve this, BVH has to be passed to the BVTCollision as both hierarchy A and hierarchy B. In addition to this unusual calling, the BVTCollision procedure itself has to be modified by adding additional check for intersection occurring between the same objects. Typically implementing the BVTCollision procedure is reduced to obtaining pairs of objects, in which the first element is a leaf form BVH A and the second element form BVH B, and checking them for intersection. With that definition of the BVTCollision procedure, one could argue that this method is not different form brute-force exhaustive search. That arguing would be true, but the BVTCollision procedure is always refined form this basic version by early pruning of nonintersecting pairs of objects. How effective this pruning is depends on not only how well built a BVH is, but also by traversing scheme and descend rules.

The following description of traversing rules is based on the beginning of the chapter 6.3 from [8]. A traversing rule, as the name suggests is a rule, which defines how nodes of a tree are visited. Result of a traversing rule is a well-order on a set of nodes. There are two extreme strategies of visiting nodes: uninformed traversing and informed traversing. Uninformed traversing is a strategy, which visits nodes regarding only structure of a tree and does not take into consideration any information related to nodes itself. Examples of such traversing, presented on figure 2.9, are deep first search (DFS) and breadth first search (BSF). DFS is a method, which starts traversing from root of a tree and proceeds deeper into a tree, until leaf node is encountered. In that moment traversing deeper is not possible, therefore algorithm backtracks traversing until parent of a first unvisited node is found and proceeds deeper into unvisited nodes. BFS also starts form root of a tree, but traverse visiting all nodes on the current level before traversing deeper into a tree.

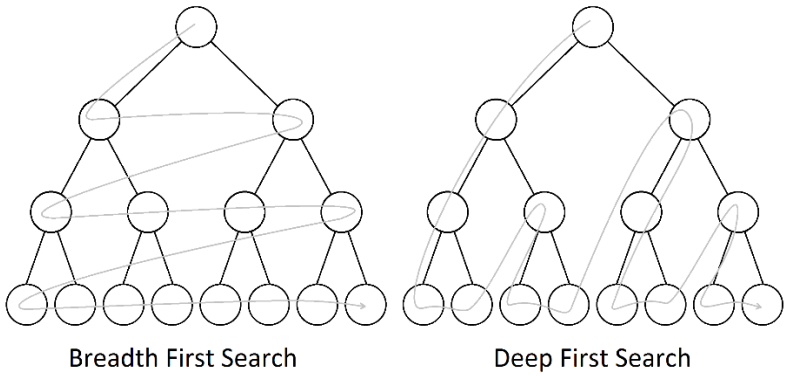


Figure 2.9. Uninformed binary tree traversing;
figure based on figures 6.4 (a) and (b) from the chapter 6.3 in [8]

An informed traversing uses entirely different strategy for visiting nodes. Instead of traversing regarding structure of a tree, informed traversing estimates, using heuristic rules, which nodes are most promising to visit. Example of informed traversing is greedy algorithm known as best first search, which in each iteration visits node currently estimated as best to visit. Best first search starts by creating priority queue of nodes, which in the beginning contains only root of a tree. Then in each iteration, the top node is popped and visited. Then all children of visited node are pushed into the queue. Iterations are repeated until predefined search succeeds achieving goal of search or search fails without achieving goal, when priority queue becomes empty. Value of priority in queue corresponds to rating of a node calculated by a heuristic. Figure 2.10 presents example of successful execution of best first search. White nodes are

visited, light gray nodes are considered by the algorithm, but not visited and dark gray nodes are not considered by the algorithm.

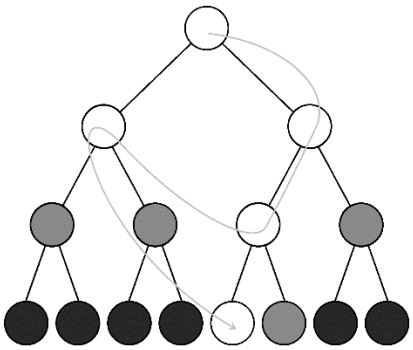


Figure 2.10. Informed binary tree traversing; figure based on the figure 6.4 (c) from the chapter 6.3 in [8]

The breadth first search is method primary used is situations, where collision detection can be interrupted or even stopped. In that situation, collision detection should be prioritized by the size of colliding objects, because penetrating of small object is more difficult to be noticed. Because BVs of nodes in lower parts of a BVH are smaller, compared to BVs in higher parts of that BVH, hierarchy should be traversed from top to bottom. Such traversing can be implemented as BFS. DFS however, is used in remaining situations, but usually is also enhanced by some simple heuristic, which helps it to choose traverse direction.

The BVTCollision procedure detects collisions between two BVHs by traversing both of them to generate pairs of objects and checks them for intersection. This procedure has to not only choose traversing scheme of BVHs individually, but also organize traversing both BVHs, relative to each other. This relative traversing is known as a descend rule and is very important from performance point of view, because affects order in which BVs are checked against other BVs for intersections, which directly translates to pruning factor. The section 6.3.1 from [8] describes several fundamental descend rules:

- Descend one BVH before the other. This descend rule is inefficient in cases where BVH of a small object is descended before BVH of a large object and the small object is positioned near the large object, for example when player's character is walking through a city. The reason for inefficiency in that scenario is all bounding volumes of the small object will overlap with bounding box assigned to root node of the large object's BVH. This overlapping will be the cause of the absence of pruning and the BVTCollision will be recursively called for each node in BVH of the small object. Alternative for that scenario is to descend BVH of the large object first, then descend BVH of the small object. This strategy is better, but still not ideal. The reason for that is leafs in BVH of the large object can have objects that are smaller than the small object, thus leafs in BVH of the large object will overlap BV of the root node in BVH of the small object.
- Descend BVHs alternatively or descend both BVHs simultaneously. To address mentioned problems one could try to balance descending into BVHs by switching hierarchy to descending or descend both hierarchies at the same time. Difference between these two is descending both hierarchies simultaneously will traverse to leafs using smaller number of recursive calls compared to descending BVHs alternatively. These strategies are definitely more efficient, resulting more pruning than descending one BVH before the other, but still could suffer the same problem, because of potential difference between sizes of bounding boxes on the same level in BVHs.

- Descend hierarchy with larger bounding volume of current node. To avoid problems with disproportionate BVHs descend rule can compare BVs of currently visiting nodes from both BVHs and descend BVH with currently bigger BV. This evaluation allows dynamically switch between BVH and balance sizes of visiting BVs, which directly results in effective pruning. To effectively exploit balancing, evaluation of BV's size has to regard proper indicators. Most common are volume, surface area and max dimension of BV. However, evaluating sizes introduces overhead, which has to be justified by appropriate pruning.

The following description of the BVTCollision procedure was written based on the section 6.3.2 from [8]. Implementation of the BVTCollision depends on traversing scheme and descend rule utilized internally. Because DFS is the most common traversing scheme in collision detection systems utilizing BVH broad phase, all presented implementations of the BVTCollision will use DFS as a traversing scheme. Depending whether descend rule descends BVHs sequentially, by switching between them, or descends both BVHs simultaneously, the BVTCollision have to be implemented slightly different. Strategy managing switching between BVHs in the BVTCollision can be abstracted away using the DescendBVT procedure, which returns value indicating whether the algorithm should descend to BVH A or B. The listing 2.5 presents generic implementation of the BVTCollision in C++ language with DFS traversing scheme.

```
template <DescendBvtFunction DescendBVT>
void BVTCollision(Collection<Pair<Object&>>& collisions, BVTNode* nodeA, BVTNode* nodeB)
{
    if (!BVOverlap(nodeA->boundingVolume, nodeB->boundingVolume)) // Prunning subtrees
    {
        return;
    }

    if (IsLeaf(nodeA) && IsLeaf(nodeB)) // Possible intersecting pair
    {
        collisions.Insert(MakeObjectsPairFromTwoLeafs(nodeA, nodeB));
    }
    else
    {
        switch (DescendBVT(nodeA, nodeB)) // Choose BVT to descend
        {
            case Descend::A:
            {
                BVTCollision<DescendBVT>(collisions, nodeA->leftNode, nodeB);
                BVTCollision<DescendBVT>(collisions, nodeA->rightNode, nodeB);
                return;
            }
            case Descend::B:
            {
                BVTCollision<DescendBVT>(collisions, nodeA, nodeB->leftNode);
                BVTCollision<DescendBVT>(collisions, nodeA, nodeB->rightNode);
                return;
            }
        }
    }
}
```

Listing 2.5: BVTCollision with DFS traverse and sequential BVH descend rule; code based on the first listing from the section 6.3.2 in [8]

The procedure from the listing 2.5 takes as input current nodes in BVH A and B and returns collection of object's pairs, which are possibly colliding. First operation executed by the BVTCollision is to check whether BVs of current nodes are overlapping. If answer is negative, then two sub-trees corresponding

to current nodes are pruned, because whole sub-trees cannot intersect. This conditional statement is responsible for informed nature of traversing in the BVTCollision and is expected to reduce large amount of work. In the next step, both nodes are tested for being leafs. If that condition is satisfied, and from previous check it is known that BV of these leafs are intersecting, objects from these leafs are paired and inserted into returned collection. However, if that condition is not satisfied, it is known that current call to the BVTCollision is inside at least one BVH and descending closer to leafs is needed. Descend rule is expressed as an implementation of the DescendBVT procedure deciding to which BVH the algorithm should descend. Finally, procedure calls itself with descendants of current node from chosen BVH. It is clearly seen that this procedure generates pairs, in which left item is always leaf node form BVH A and right item is always leaf from BVH B. In worst case, when all BV are overlapping each other resulting no pruning, number of pairs will be equal to number of leafs in BVH A multiplied by number of leafs in BVH B, which is number of possible pairs. The size of a call stack, in worst-case scenario, will be equal to height of higher BVH.

Implementations of descend rules are provided to the BVTCollision as a definition of the DescendBVT. Described earlier sequential descend rules can be implemented as shown on listing 2.6.

```
template <const Descend DescendFirst>
Descend DescendOneBeforeOther(const BVTNode* const bvtA, const BVTNode* const bvtB)
{
    switch (DescendFirst)
    {
        case Descend::A:
        {
            return IsLeaf(bvtA) ? Descend::B : Descend::A;
        }
        case Descend::B:
        {
            return IsLeaf(bvtB) ? Descend::A : Descend::B;
        }
    }
}

Descend DescendLargerVolume(const BVTNode* const bvtA, const BVTNode* const bvtB)
{
    if (IsLeaf(bvtA))
    {
        return Descend::B;
    }
    else if (IsLeaf(bvtB))
    {
        return Descend::A;
    }
    else
    {
        return (bvtA->boundingVolume >= bvtB->boundingVolume) ? Descend::A : Descend::B;
    }
}
```

Listing 2.6: Implementations of descending rules in C++;
code based on the second listing from the section 6.2.2 in [8]

The DescendOneBeforeOther procedure has to versions, one descending BVH A before BVH B and second with reverse order of descending. The template parameter controls which version of the procedure is active. Logic of this descend rule chooses first BVH until leafs are reached, and then chooses second BVH. The DescendLargerVolume procedure, when possible, chooses BVH with currently larger BV. Designing implementation of descend rule one have to support cases when

algorithm reached leaf nodes and descending is not possible, otherwise memory access violation errors could occur.

Simultaneous descending in the BVTCollision using DFS traversal is implemented slightly different from code presented in the listing 2.5, because more cases have to be considered, resulting four recursive calls. The listing 2.7 presents code adjusted to handle simultaneous descending.

```
void BVTCollision(Collection<Pair<Object&>>& collisions, BVTNode* nodeA, BVTNode* nodeB)
{
    if (!BVOverlap(nodeA->boundingVolume, nodeB->boundingVolume)) // Prunning subtrees
    {
        return;
    }

    if (IsLeaf(nodeA))
    {
        if (IsLeaf(nodeB)) // Possible intersecting pair
        {
            collisions.Insert(MakeObjectsPairFromTwoLeafs(nodeA, nodeB));
        }
        else // Descend BVT B
        {
            BVTCollision(collisions, nodeA, nodeB->leftNode);
            BVTCollision(collisions, nodeA, nodeB->rightNode);
        }
    }
    else
    {
        if (IsLeaf(nodeB)) // Descend BVT A
        {
            BVTCollision(collisions, nodeA->leftNode, nodeB);
            BVTCollision(collisions, nodeA->rightNode, nodeB);
        }
        else // Descend both BVT simultaneously
        {
            BVTCollision(collisions, nodeA->leftNode, nodeB->leftNode);
            BVTCollision(collisions, nodeA->leftNode, nodeB->rightNode);
            BVTCollision(collisions, nodeA->rightNode, nodeB->leftNode);
            BVTCollision(collisions, nodeA->rightNode, nodeB->rightNode);
        }
    }
}
```

Listing 2.7: C++ implementation BVTCollision utilizing simultaneous descending and DFS traversing; code based on the listing from the section 6.3.3 in [8]

The first operation in the procedure form the listing 2.7 is conditional statement checking for overlapping of BVs, which hopefully result in pruning. The remaining part of BVTCollision is dedicated to handling all possible cases. In the first case nodes from both BVHs are leafs and objects from these nodes are added to collection of possible collisions. In the second and the third cases, one node is a leaf and the other node is not. Both of these cases are handled by descending into BVH of the internal node. In the final case both nodes are not leafs and the algorithm descends both BVHs. Because the BVTCollision have to generate all possible pairs of objects from BVH A and B for all possible pairs of children nodes of current nodes BVTCollision have to be called. The main advantage of simultaneous descending compared to sequential descending is reaching leaf nodes in fewer recursive calls. Simultaneous descending descends both BVHs by one level calling the BVTCollision four times with all combinations of child nodes from both BVHs. However, sequential descending calls two times the BVTCollision with child nodes from first the BVH and then for each of these two calls, calls the BVTCollision another two

times with child nodes from the second BVH, resulting in total of six calls. In fact, it can be estimated that for two full binary trees, which all BVs are overlapping simultaneous descending reaches leaf nodes in two thirds of calls required by the sequential descending. However, usually the worst case scenarios are not representative and effective pruning of descending to BVH with larger BV of current node can be more beneficial. Unfortunately, theoretical proof for superiority of one descending scheme over the other is very difficult, because pruning factor depends greatly on geometrical properties of a scene. Only statistical analysis can show which solution would be better.

2.2. Spatial Partitioning

The spatial Partitioning is a group of methods, in which the world space is divided into partitions, exploiting the fact that objects can collide only in their local surroundings. Spatial partitioning is a bit different approach form bounding volume hierarchies. Both methods organize objects spatially, but based on different determinants. Spatial partitioning divides space itself based on some predefined scheme. On the other hand, BVH divides objects into clusters and then encapsulates them with BVs. In spatial partitioning, space division depends only on the partitioning scheme, disregarding geometry of objects. This gives an edge to partitioning scheme in terms of constructing space division, because it is known in advance how space will be partitioned. On the other hand, spatial partitioning is not suitable for some configurations of objects geometry, resulting in performing worse than BVH, which can dynamically adapt to objects geometry. Conventionally methods of spatial partitioning are classified into two distinct groups: uniform grids and hierarchical grids. Uniform grid is a partitioning in which space is divided into subspaces, creating regular tessellation. Most useful uniform grids are regular square girds for a plane and cube grids for a 3D space. Main advantages of these solutions are their simplicity and ease of assigning objects to partitions.

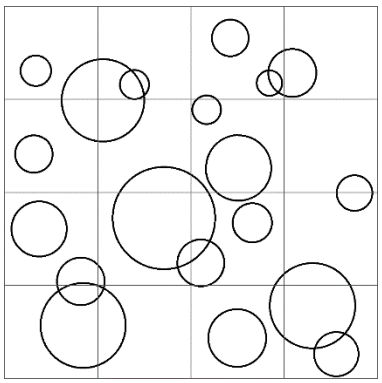


Figure 2.11. Example of square uniform grid in 2D world

Uniform grids accelerate broad phase by utilizing very powerful property of spatial partitioning, which states that objects can intersect if and only if objects are assigned to common cell. In response to the need for a spatial partitioning working efficiently with objects of diverse sizes, hierarchical grids where created. This group of methods places a number of grids dividing the same space, stacked on top of each other. These grids are sorted by their granularity creating a hierarchy. Grids with largest cell sizes occupy highest levels in the hierarchy and grids with smallest cell sizes occupy lowest levels. Prime examples of a hierarchical grid are an octree, for 3D space, and a quadtree, for 2D space. In these trees, root node contains bounding box of all objects in the world. Then each node is created by dividing box into four boxes in quadtree and eight boxes in octree, which has twice smaller size.

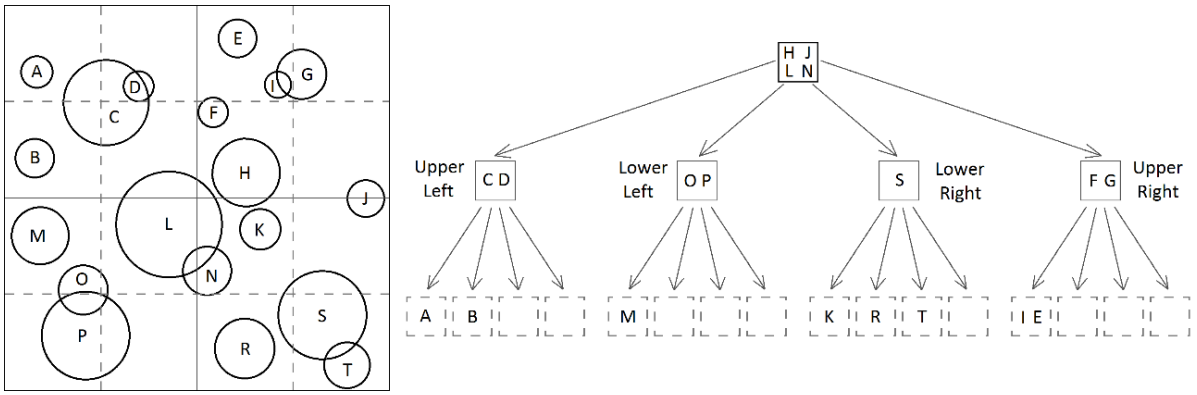


Figure 2.12. Example of quadtree in 2D world

There are numerous strategies, which manage creation of hierarchical grids and assignment of objects to nodes. Most fundamental creation rule of hierarchical grids is the spatial partitioning scheme. It decides how space is divided into cells and what the shape of a cell is. As mentioned earlier, most common cell shapes are squares and cubes, but there are other useful possibilities, like arbitrary polyhedrons. In most cases spatial partitioning scheme ensures that sum of children subspaces is equal to parent subspace. This is very important property, because it allows hierarchy grid to accelerate broad phase. Quite important is proper selecting number of children. In most practical usages, number of children is fixed. Most popular are binary, quad and octrees. Managing depth of a tree is usually limited by predefined constant. Large number of hierarchical grids have this predefined depth limit, because number of leafs grows exponentially as depth of a hierarchy increases. Despite of predefined depth limit, actual depth can be dynamic and grow as needed, obviously, until the limit is not exceeded. Such a dynamic depth is a result of dynamic building of a hierarchical grid, in which nodes are created as needed. In opposition to dynamically built hierarchy is a creation of a whole data structure initially and filling it with objects afterwards. Middle ground solution would be initially creating upper parts of a hierarchy and creating them afterwards as needed. One of the most performance affecting rules are assignment rules, which decide how objects should be assigned to nodes. On the one side of a spectrum, there is an assigning rule stating that object is assigned to the lowest node, whose cell fully encloses BV of that object. The main advantage of this rule is very fast inserting, deleting and updating of an object in a hierarchy. On the other side of a spectrum, there is an assignment rule, which allows an object to always be assigned to a leaf of a hierarchy. However, this methods has its drawback, because one object has to be assigned to all leafs that their cells intersect with object's BV. This could lead to multiplication of objects and is considered ineffective for dynamic scenes, because as object move through space all of its assignments have to be updated. Previous method does not suffer from this problem, but has its drawbacks too. Potentially many objects are straddling separating planes and have to be inserted into hierarchy much higher than their size suggests. This could lead to many objects accumulating in high nodes, which will decrease collision detection performance. Hierarchy grids accelerate broad phase by cutting off whole sub-trees, which prunes unnecessary intersection tests, similarly to how BV hierarchy accelerates broad phase. Cutting off sub-trees is possible, because objects assigned to a node in a hierarchy grid can only intersect with objects assigned to nodes, which are direct ancestors and children of that particular node.

2.2.1. Uniform Grid

The following description of uniform grids is based on the chapter 7.1 from [8]. The simplest space partitioning is a uniform grid. This broad phase algorithm comes from an observation that objects collide only in their local surroundings. The closer two objects are to each other, the more likely they are to collide. Therefore, partitioning scheme in uniform grids divides space into equal size squares or cubes, depending on number of space dimensions. Assignment rule in this broad phase is also quite simple. An object is assigned to all cells, which overlaps. This rule in practice is even more simplified by assigning object to all cells, which its BV overlaps. Thanks to very simple partitioning scheme and assignment rule, calculating range of indices of cells, which an object overlaps, can be obtained by trivial division the world coordinates of minimal and maximal points defining object's AABB by the size of a grid. An implementation of a procedure detecting all colliding objects is shown in the listing 2.8.

```
void GridBroadPhase(Collection<Pair<const Object&>>& collisions, Grid grid)
{
    for (const Cell& cell : grid)
    {
        for (unsigned int i = 0; i < cell.objects.Size(); ++i)
        {
            for (unsigned int j = 0; j < cell.objects.Size(); ++j)
            {
                if (i != j)
                {
                    const Object& objectI = cell.objects[i];
                    const Object& objectJ = cell.objects[j];

                    if (BVOverlap(objectI.boundingVolume, objectJ.boundingVolume))
                    {
                        collisions.Insert(Pair<const Object&>(objectI, objectJ));
                    }
                }
            }
        }
    }
}
```

Listing 2.8: Simple collision detection using grid broad phase

At first procedure might give impression that it is very inefficient, but when size of a grid is chosen properly, the GridBroadPhase can boost collision detection performance significantly. The algorithm iterates through all cells in a grid an in each cell applies brute force checking all possible pairs of objects for intersection. Ideally, grid should divide space in such a way, that only few objects are overlapping one cell. In that case, two most internal loops, iterating through objects overlapping current cell, are reduced to few checks. The most external loop, iterating through all cells in grid, should be only loop contributing to the cost of the GridBroadPhase. However, size of a grid can be estimated to be roughly equal to number of objects present in the world divided by average number of objects overlapping one cell. When number of cells exceeds number of objects, then there must be empty cells, which can be ignored by maintaining additional list with references to all non-empty cells or flag embedded in each cell, indicating if cell is empty or not. Therefore, the GridBroadPhase is expected to have linear complexity, but only if size of a grid is chosen properly, otherwise the number of objects per cell increases and algorithm's complexity becomes quadratic again.

Choosing proper size of a grid is very important, because it allows detecting collisions efficiently. Grid size can be incorrect due to several manners:

- Grid is too fine. Scheme A in the figure 2.13 shows that size of cells are too small, because objects are overlapping too many cells. This becomes a problem when scene contains dynamic objects, because movement of this object causes quite expensive update operation, which has to remove object reference form many cells and then insert object reference to many cells.
- Grid is too coarse. Scheme B in figure 2.13 shows that size of cells are too large, because too many objects are overlapping the same cells. This will cause increase in the GridBroadPhase complexity form linear to quadratic, because all objects, from common cell, have to be tested for intersection against each other. Scheme C in the figure 2.13 shows also grid with too large cells, but because of complexity of object's geometry. Solution in that situation is to split large and complex objects into convex polygons and adjust cell size accordingly.
- Grid is too fine and too coarse at the same time. Scheme D in the figure 2.13 presents situation, which cannot be handled by uniform grid efficiently, because cells are too small for largest object and too large of remaining objects. Therefore, proper size of grid is impossible to be chosen properly. Solution in that situation is to use hierarchical grid or other broad phase.

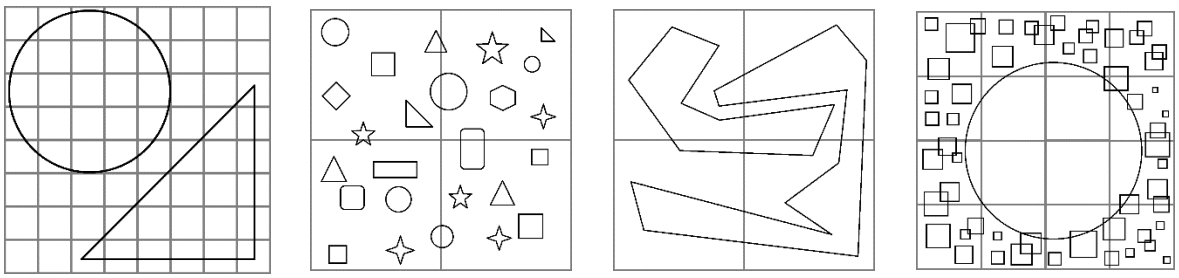


Figure 2.13. Incorrectly chosen sizes of a grid; figure based on the figure 7.1 from the chapter 7.1 in [8]

Usually size of cells in a grid is chosen to be large enough to fit the largest object rotated at any angle, because this assures that each object can overlap only eight cells in 3D space and four cells in 2D space simultaneously.

Implementations proposed in the following section are based on sections 7.1.3 and 7.1.4 in [8]. Storing uniform grid and information about which object overlaps which cells can be done using array of cells, which contains referenced to lists of objects overlapping particular cell.

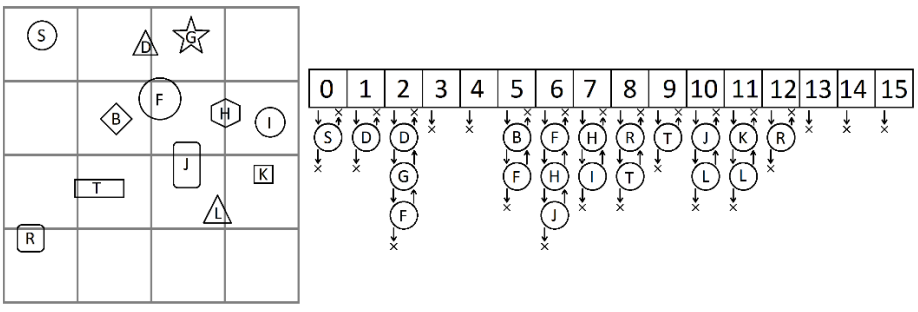


Figure 2.14. Uniform grid stored as an array of lists

This data structure is quite efficient, because inserting, deleting and updating objects are operations characterized by constant time complexity. In addition to that, when cells have proper size, lists of objects are kept short. Only disadvantage of this method is its memory requirements. In most cases, world density is not that high, resulting in numerous empty cells. For large grids, the number of empty

cells becomes huge overhead wasting memory. For these cases instead of regular array hash table can be used. The figure 2.14 shows rather dense world – only one third of elements contains empty list, so this particular situation will not benefit from using hash table.

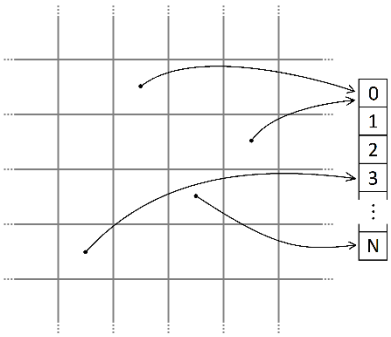


Figure 2.15. Uniform grid stored as a hash table; figure based on the figure 7.2 from chapter 7.1 in [8]

The figure 2.15 presents most important idea behind representing uniform grid as a hash table. Instead of mapping one cell to one element of an array, cells are mapped to specific element of an array, which has fixed size N. The reversed statement is not true, that is one element of an array can store data of more than one cell. Another consequence of that is grid becomes conceptual and does not reside in the memory. The only data stored in the memory are objects mapped to proper element of a hash table, thus size of a grid does not affect performance of collision detection. The mapping of cells to hash table elements is defined by a hash function.

```
template <unsigned int HASH_TABLE_SIZE>
unsigned int HashFunction(Cell cell)
{
    const int primeX = 0x8da6b343;
    const int primeY = 0xd8163841;
    const int primeZ = 0xcb1ab31f;

    const int n = (cell.coords.X * primeX +
                   cell.coords.Y * primeY +
                   cell.coords.Z * primeZ) % HASH_TABLE_SIZE;

    return (n < 0) ? (n + HASH_TABLE_SIZE) : n;
}
```

Listing 2.9: Example of hybrid between multiplicative and modular hash function; code based on the listing from the chapter 7.1.3 in [8]

The listing 2.9 presents very simple hash function, which multiplies each coordinate of a cell by an arbitrarily chosen prime number and calculates mod operation on sum of these products to reduce value to range from zero to the size of a hash table minus one. That calculated hash can be used to address hash table directly. A quality hash function has to distribute keys evenly across all hashes. It is recommended to incorporate more sophisticated hash function, which performs better than the hash function presented on the listing 2.9. Unfortunately designing such mappings will not be covered here, but can be found in any hash table related book. Because of not unique mapping of elements from hash table to cells, conflicts can occur and have to be solved. There are two major solutions, allowing solving key conflicts, notably open and closed hashing. Open hashing resolves conflicts by arranging conflicted hashes into lists. These lists, which are elements of a hash table, contain nodes, which are constructed form key and data. In case of representing grid as a hash table, node key should contain coordinates of

a cell and node data should contain elements of the array from previous solution, which are lists of objects. The figure 2.16 presents this data structure visually.

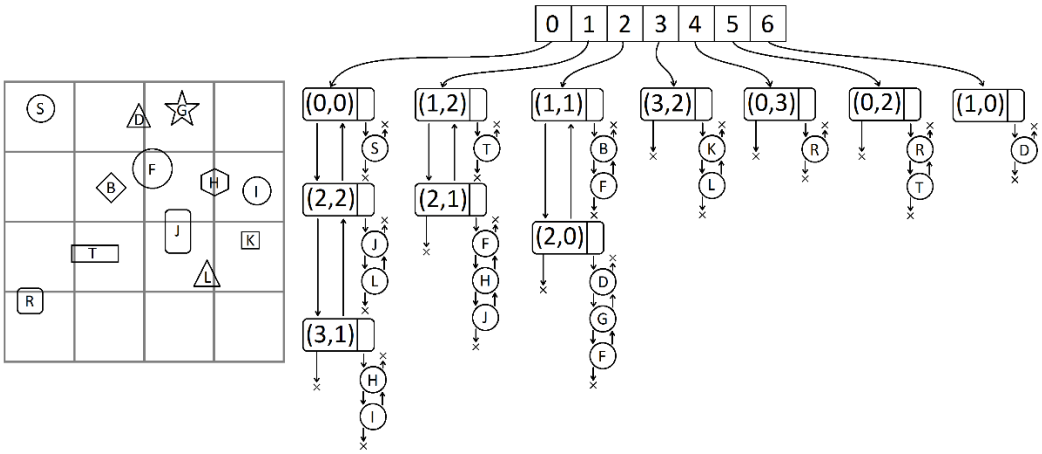


Figure 2.16. Hash table with open hashing representing uniform grid

Open hashing can be considered as too complicated in terms of clarity of code, because in this solution, grid is represented by an array with lists of lists as elements. Closed hashing is alternative solution in which, conflicts are solved by occupying other elements of a hash table, which are free. The most basic method called linear probing, when encounters conflict, searches through next elements of a hash table until finds free element and occupies it. When the end of a hash table is reached, search continues from the beginning of a hash table. The figure 2.17 shows the same world as figures 2.16 and 2.14 with grid represented as a hash table with linear probing.

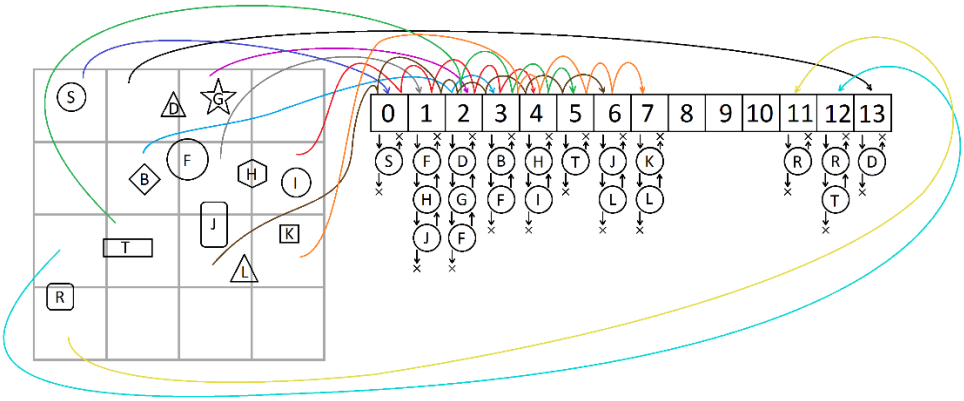


Figure 2.17. Hash table with closed hashing representing uniform grid

It can be immediately observed on the figure 2.17 that for an array of the size 14 with hash function from the listing 2.9 distribution of keys across hash space is poor, resulting many searches for next free element is hash array. The figure 2.17 is an excellent example of how important is good quality hash function and proper choice of size of a hash table. Ideally for closed hashing load factor should not be bigger that 50%. Load factor can be easily calculated using formula (1.3).

$$\boldsymbol{L = \frac{n}{N}} \qquad (1.3)$$

L – load factor [%]

n – number of elements in hash table [element]

N – size of hash table [element]

Deleting elements from a hash table with closed hashing cannot be done in ordinary manner, because searching for an object would be stopped prematurely, when encountered deleted object. Therefore deleting elements from a hash table has to be done lazily, with objects elements marked deleted for insertion and occupied for search operation. It is also important to mention that algorithm detecting all possible collision pairs cannot distinguish to which cells object actually belong. The only information that can be extracted is that objects cannot belong to the same cells when occupy different hashes. Because of that, the algorithm detecting possible collisions checks all possible pairs between object belonging to cells with common hash instead of one cell.

2.2.2. Hierarchical Grid

The description of hierarchical grids is based on the chapter 7.2 from [8]. In order to handle objects with varying sizes hierarchical grids were introduced. They effectively resolve the problem with impossible choice of proper size of uniform grid cells by applying number of uniform grids with various cell sizes on top of each other, covering the same area. This set of hierarchies is sorted by the size of their cells, forming hierarchy. A grid with the smallest cells is located on the lowest level and a grid with the largest cells is located on the highest level. Hgrids do not assume any particular ratio between sizes of cells in different grids, but usually cells of a grid at a level k are twice the size of cells at the level k-1. Example of an hgrid with such a ratio is presented in the left side of the figure 2.18. It is important to realize that this ratio causes cells to grow exponentially as levels increase, thus larger ratios makes this grow even faster, which makes them rather impractical. Because ratio has to be larger than one, ratio smaller than two cannot be an integer. Integer ratios have unique property, making ancestor cells a sum of their descendant's cells, which makes integer ratios most desirable. In addition, integer ratios allow hgrids to be represented as a forest. Visual example of such a representation is presented in the right side of the figure 2.18. This representation allows utilizing wide range of techniques and knowledge applicable to trees and graphs in general.

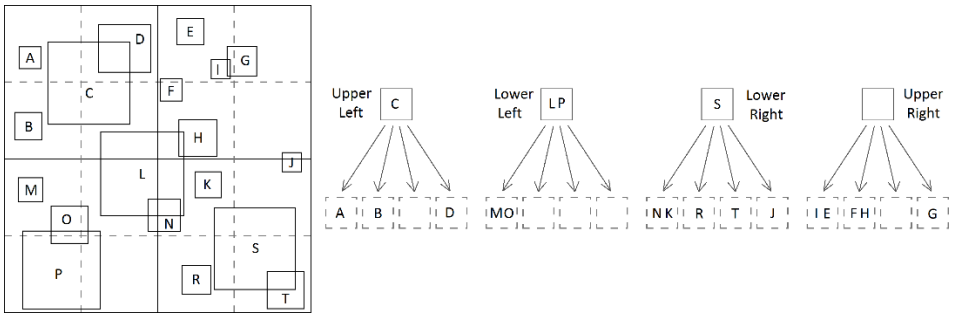


Figure 2.18. Hgrid, represented as a forest, partitioning 2D world

However, hgrids in do not have to be considered forests and following algorithms operating on hgrids will not assume they are.

Hierarchical grids were designed to eliminate problem of large objects covering numerous cells at the same time. Therefore, insertion of an object into hgrid assigns it to the one particular cell at one particular level, making insertion a fast operation. This particular insertion algorithm makes hgrid an efficient data structure particularly suitable for detecting collisions between dynamic objects. The reason for that is, when moving object crosses trough scene, its assignment to cell has to change. Because in hgrid only one assignment per object has to be maintained, fast moving objects are updated efficiently.

The insertion itself is split into two parts: choosing level at which object is inserted and choosing cell at that level to which object is assigned. An object is inserted at the lowest level, which contains cells large enough to accommodate BV of the object. This should not be confused with inserting object at the smallest cell, which encapsulates entire BV of the object, because a BV of the object can protrude outside of a cell. Subsequently algorithm calculates which cells encapsulates center of object's BV and chooses that cell, which is located at the chosen level. Valuable attribute of such an assignment is that each object is guaranteed to cover at most eight cells in 3D space. Example of constructed hgrid via insertion operations can be seen in the figure 2.18. Also the listing 2.10 shows logic of insertion operation and how it can be implemented in C++.

```
void InsertObjectToHGrid(HierarchicalGrid& hierarchicalGrid, Object* const object)
{
    const float diameterOfObjectBV = object->boundingVolume->radius * 2.0f;

    // Calculate level and cell, at which object will be inserted.
    unsigned int cellLevel = 0;
    float cellSize = hierarchicalGrid.MIN_CELL_SIZE;

    // Go up until BV does not fit into cell.
    while (cellSize < diameterOfObjectBV)
    {
        cellSize *= hierarchicalGrid.CELL_TO_CELL_RATIO;
        ++cellLevel;
    }

    // Insert object into list corresponding to intertion cell's bucket.
    Cell cell(object->boundingVolume->center / cellSize);
    object->bucket = hierarchicalGrid.hashTable.ComputeHashBucket(cell, cellLevel);
    object->level = cellLevel;
    object->nextObject = hierarchicalGrid.hashTable[object->bucket];
    hierarchicalGrid.hashTable[object->bucket] = object;
}
```

Listing 2.10: Implementation of object insertion operation on a hierarchical grid in C++; code based on the listing from the section 7.2.1 in [8]

Procedure from the listing 2.10 finds proper insertion level by iterating, from bottom, levels of a hierarchy, until the diameter of object's BV is larger than the size of a cell at the current level. A cell to which object will be assigned is chosen by dividing position of the center of insert object's BV by the size of cells at chosen level, to convert world coordinates to cell coordinates. Such obtained indices and the level of a cell are enough to define to where object should be inserted. In this example, actual storage for hgrid is a hash table. The idea behind this method of storing hgrid in a hash table is to make coordinates of a cell and its level, a key in a hash table and calculate bucket of a cell using hash function. This is nearly identical as storing uniform grid in a hash table, but the difference is in the hash key. The hash key in a uniform grid does not contain level of a cell, but in hgrid does. Key conflicts can be solved by open or closed hashing. For implementation in the listing 2.10 closed hashing was chosen to maintain simplicity. The ComputeHashBucket procedure expresses closed hashing, because it not only computes hash of a cell, but also searches for free bucket until one is found. To complete insertion, object is assigned to the bucket by adding it at the beginning of a list of objects belonging to cells with common bucket.

Collision detection accelerated by a hierarchical grid is typically done by implementing procedure, which finds all collisions between one particular object and all objects in a hierarchical grid. Before detailed description of this algorithm, it is important to mention that, the insertion operation allows BV's of objects to protrude from cells which they are assigned to. When object is inserted into a cell,

which encapsulates center of its BV and BV of this object is smaller or the same size as cell itself, then in worst-case scenario, object would protrude from its cell exactly by its radius. This protruding area is marked by the green dashed line in the figure 2.19.

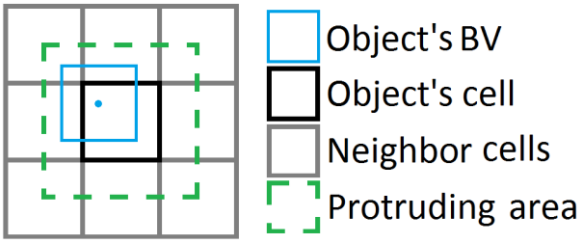


Figure 2.19. Protruding area overlapping neighbor cells

Because of this protruding area, algorithm detecting collisions between input object and objects residing in a hgrid have to take into consideration objects from neighbor cells. In order to find range of neighbor cells, which needs additional checking, an overlapping area needs be calculated. All objects, which centers are encapsulated by the overlapping area might collide with the input object. The overlapping area and the collision between input object (marked by its blue BV) and the object from overlapping area (marked by its green BV) are shown in the figure 2.20.

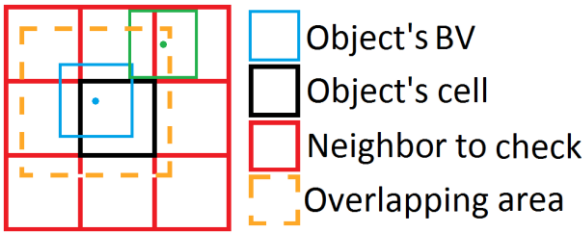


Figure 2.20. Overlapping area for object's BV, defining neighbor cells, which need additional checking

Overlapping areas are always specific for a BV of an object, because its center is defined by the center of object's BV. The size of overlapping area is a sum of the diameter of object's BV and the diameter of cells at particular level of hgrid. The reason for that is protrude area extends by half of cells size outside each cell in each direction, so object's BV has to be enlarged by this margin. The overlapping area then can be transformed into overlapping range of cells that have to be checked for overlapping with the input object. The algorithms calculating range of neighbor cells that needs to be checked divides world coordinates of overlapping area by the size of cells, to convert them to cells coordinates and then uses floor function on each coordinate of minimal point and ceil function on maximal point defining overlapping area. Because, all cells intersecting with the overlapping area need to be additionally tested, fractional part of minimal point coordinates needs to be cut off and maximal point coordinates need to be complemented to whole numbers. This algorithm defining range of cells can be seen in the listing 2.11. Knowing that additional cells are needed to participate in detecting collisions using hgrid data structure, the algorithm itself can be explained. The listing 2.11 shows example implementation of this algorithm written in C++ language.

```
Collection<Pair<const Object* const>>
DetectObjectCollisionsAgainstHGrid(HierarchicalGrid& hierarchicalGrid, Object* object)
{
  // Collection of collisions - algorithm output.
  Collection<Pair<const Object* const>> collisions;

  const BoundingVolume* const boundingVolume = object->boundingVolume;
```

```
  // Iterate through levels.
  float cellSize = hierarchicalGrid.MIN_CELL_SIZE;
  for (unsigned int cellLevel = 0; cellLevel < hierarchicalGrid.MAX_LEVELS; ++cellLevel)
  {
    // Find range of cells.
    const float halfRangeSize = boundingVolume->radius + cellSize / 2.0f;
    const float oneOverCellSize = 1.0f / cellSize;

    int X0 = int(floor((boundingVolume->center.X - halfRangeSize) * oneOverCellSize));
    int Y0 = int(floor((boundingVolume->center.Y - halfRangeSize) * oneOverCellSize));
    int Z0 = int(floor((boundingVolume->center.Z - halfRangeSize) * oneOverCellSize));

    int X1 = int(ceil((boundingVolume->center.X + halfRangeSize) * oneOverCellSize));
    int Y1 = int(ceil((boundingVolume->center.Y + halfRangeSize) * oneOverCellSize));
    int Z1 = int(ceil((boundingVolume->center.Z + halfRangeSize) * oneOverCellSize));

    // Iterate through cells, that needs to be checked.
    for (int x = X0; x < X1; ++x)
    {
      for (int y = Y0; y < Y1; ++y)
      {
        for (int z = Z0; z < Z1; ++z)
        {
          Cell cell(Coordinates(x, y, z));
          int bucket = hierarchicalGrid.hashTable.ComputeHashBucket(cell, cellLevel);

          // Check neighbor cell, whether contains neighbor objects
          // possibly intersecting with the object.
          const Object* const neighbor = hierarchicalGrid.hashTable[bucket];
          while (neighbor)
          {
            if(neighbor != object && BVOverlap(boundingVolume, neighbor->boundingVolume))
            {
              collisions.Insert(Pair<const Object* const>(object, neighbor));
            }
          }
        }
      }
    }

    cellSize *= hierarchicalGrid.CELL_TO_CELL_RATIO;
  }

  return collisions;
}
```

Listing 2.11: Implementation of collision detection between object and hierarchical grid in C++; code based on the last listing from the section 7.2.1 in [8]

Collision detection algorithm utilizing hgrid iterates through levels of hierarchy, from bottom to top, and on each level finds range of cells, which might contain objects possibly colliding with the input object. After determination, which cells need checking, algorithm iterates though them and checks every object assigned to them for collision with the input object.

Choosing proper sizes of cells in hgrid can greatly affect performance of collision detection, thus it is important to properly adjust sizes of cell in hgrid at each level. In practice, strategy of choosing size of cells at the lowest level to be large enough to fit the smallest object rotated at any angle and calculating cell sizes at remaining levels using cell to cell ratio works well.

2.2.3. Octree

The following description of the octree broad phase is based on the section 7.3.1 from [8]. The octree broad phase utilizes a type of spatial partitioning, in which space is recursively divided into axis-aligned cubes. Data structure expressing that recursive partitioning into cubes is an octree. In an octree, each node represents axis-aligned cube, which is divided along x, y and z axis creating eight sub-cubes, which are represented as child nodes. Each child cube is exactly two times smaller than the parent cube, that means ratio between lengths of edges of the parent cube and a child cube is two. The figure 2.21 presents spatial partitioning represented by an octree.

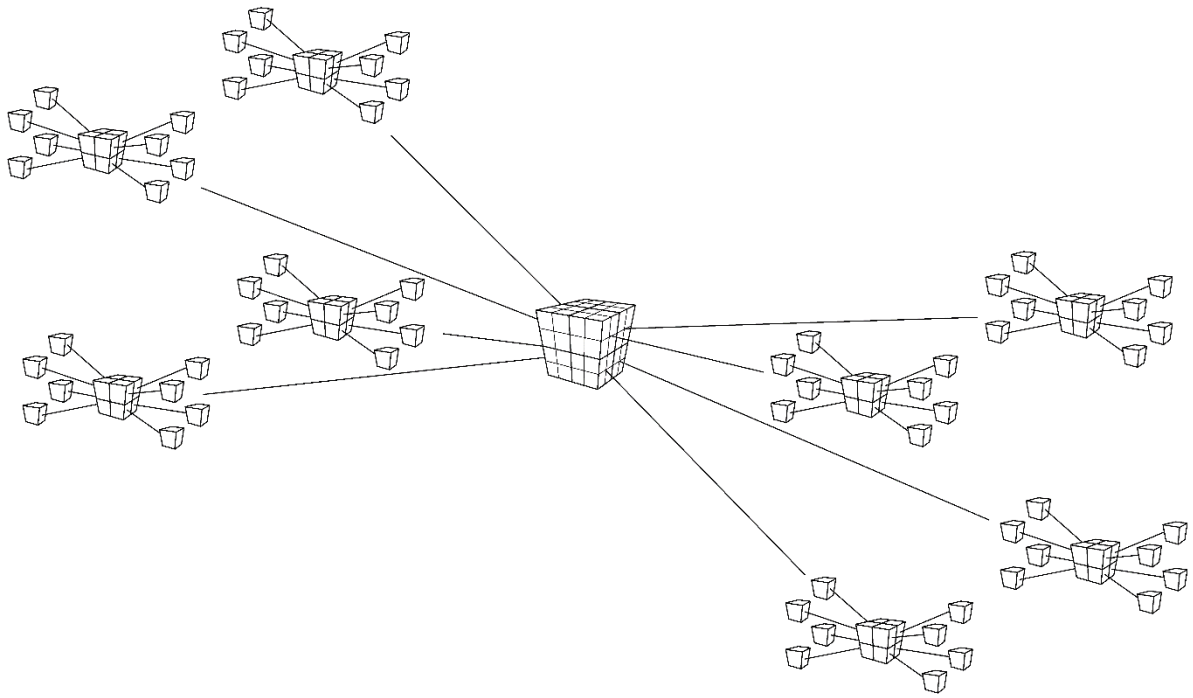


Figure 2.21. Octree's spatial partitioning

The root of an octree is a box containing all objects and its size is usually selected to encapsulate entire world rotated at any angle. Dimensions of other nodes are calculated using rule that halves the dimensions of a parent node to get dimensions of its child nodes at each level. The analogous data structure to the octree in 3D space is the quadtree in 2D space. The quadtree, as the same suggests, is a tree with nodes, which each has four children instead of eight children. Also with each node is associated axis-aligned square instead of a cube. An example of a quadtree can be seen in the figure 2.12. Both octrees and quadtrees have desirable feature of a parent node's volume being sum of child nodes' volume, which makes them quite appealing in terms of pruning unnecessary collision tests.

The most important factor deciding of how profitable is the octree broad phase is an assigning rule, which specifies to which node each object belongs. There are two major strategies described in the section 7.3.2 from [8]:

- Assign an object to each leaf node, which overlaps object's BV. Algorithm achieving that assigning can start by assigning all objects to the root node. Then recursively split objects by octree's X partitioning plane, then split these two sets by octree's Y partitioning plane and finally split four sets by octree's Z partitioning plane. However, some objects might straddle between partitioning planes.

In that cases object is assigned into both sets. Testing whether objects straddles plane can be executed by checking if any two vertices defining object are on opposite sides of the plane. This assignment rule is more suitable for representing static part of the virtual world, because duplicating objects along large number of nodes increases number of operations required to update object's assignment after it changed position. This is not the problem, when objects are static, because their assignments do not have to be updated. This solution can be refined even further by splitting objects instead of duplicating them. In that situation, performance of collision detection is not reduced by additional geometry testing. However, splitting objects is expensive operation and has to be done in the preprocessing stage. For splitting objects, the Sutherland-Hodgman algorithm can be used.

- Assign object to the smallest possible volume, which encapsulates BV of an object. Alternative approach to duplicating or splitting an object, is to assign it to a node, whose partitioning plane cuts this object, implying it to straddle. The advantage of this solution is no overhead of updating object's assignment. However, in this assigning rule objects are inserted into an octree higher than its size would suggest. The most problematic are objects straddling partitioning planes belonging to nodes located high in an octree, because regardless of their size, they are inserted in the octree near or even in the root node. The figure 2.22 visualizes problem of straddling objects in a quadtree. The most extreme examples of that problem are objects J, N and H, which have to be inserted into root of the quadtree, even if their sizes suggest that they could be inserted into a leaf node.

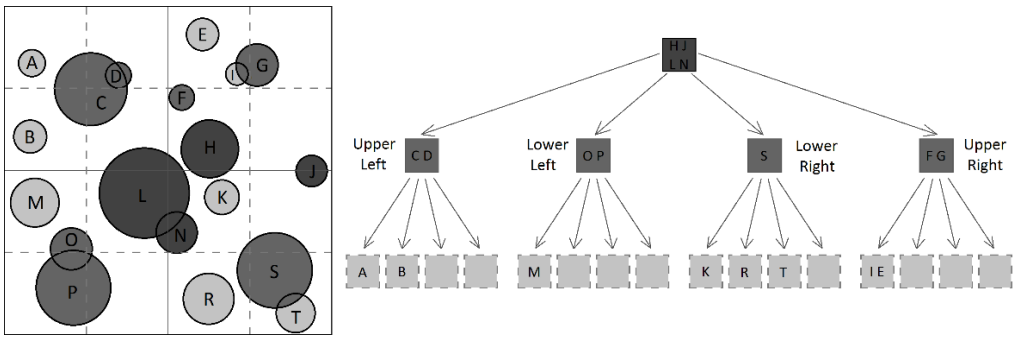


Figure 2.22. Straddling objects inserted into quadtree higher than their sizes justifies

Recursive division of space has to stop at some point and there are numerous criteria according to which recursion stops:

- When all objects are not able to descend deeper into an octree. Some assigning rules do not allow object to descent into an octree unless an object meets some requirement. Recursion stops naturally at some point, when all objects do not meet this requirement.
- When the number of children is smaller than predefined value. Building very high tree is not desirable, because traversing from the top to the bottom becomes expensive operation. Therefore, premature stopping of building octree when nodes contain predefined small number of objects is a good idea.
- When height of an otree exceeds predefined limit. This criterion also tries to prevent too extensive growth of an octree, but using different metric. Instead of considering number of children in a node, a criterion oversees height of an octree and stops building when the tree is too high. This could be beneficial in situations, when objects are clustered in small section of space and further spatial partitioning does not yields any benefits.

- When length of edges, surface area or volume become smaller than minimal value. Premature exit based on size of sub-cubes can be beneficial in situations, when assigning rules allows objects to always descent deeper into an octree and the size of the octree has to be managed. This fixed limit has to be adjusted properly to objects' geometry.

The proposed here methods of implementing the octree data structure are based on the description in the section 7.3.2 from [8]. An implementation of an octree can represent this data structure as an array of nodes or as variables representing nodes with links to other nodes, scattered around large block of memory. The array representation of an octree is equivalent to representation of binary tree as an array. All nodes are stored in the continuous block of memory and accessing child node of particular parent node is done by calculating unique index of child node in an array using formula (1.4).

$$\boldsymbol{Index(n, m) = 8n + m} \tag{1.4}$$

n – index of a parent node

m – number of a child node

The array representation is very good in terms of cache friendliness and traversing speed. However, this speed comes with cost of memory consumption, because array representation supports only storing complete trees. Alternative solution is to represent each node as a variable with links to other nodes. In C++ language node can be structure similar to shown in the listing 2.12.

```
struct OctreeNode
{
    static const int NUMBER_OF_CHILDREN = 8;

    Vector3     center;                       // Center point of axis-aigned bounding box
    float       halfSize;                     // Radius of axis-aigned bounding box
    OctreeNode* children[NUMBER_OF_CHILDREN]; // Links to child nodes
    Object*     objectList;                   // Link to list of objects assigned to node
};
```

Listing 2.12: Structure of an octree's node; code based on the listing from the section 7.3.1 in [8]

This representation allows storing only needed nodes of an octree, thus saving memory. However, additional memory has to be dedicated to storing links to child nodes. In practice representing an octree as nodes with links is much more memory efficient and the degradation of performance caused by traversing hierarchy using links is not substantial. Also, allocating nodes in predefined pool of memory helps greatly with caching, reducing scattering nodes across global memory. Apart from method of representing an octree, there is a building strategy, which decides when nodes of an octree should be created. Two major strategies will be described: preallocating all nodes in advance and dynamic grow of an octree. In order to minimize execution time of insertion operation, all nodes can be created when program starts. Major disadvantage of approach is slow startup time of an application. In addition, this building strategy combined with node-link representation consumes the largest amount of memory. Dynamic grow of an octree is an opposite approach, in which allocation of the nodes is delayed until node is actually needed. This building strategy allows to shorten startup time and combined with link-node representation reduce memory consumption. This benefit comes with a price of elongated time required to insert an object into an octree.

After description of assignment rules, representation methods and building strategies example implementation of insertion operation on an octree represented as nodes with links with dynamic octree building can be presented.

```
void Insert(OctreeNode* octreeNode, Object* object)
{
    bool straddle = false;
    unsigned int index = 0;
    Vector3 offset(-octreeNode->halfSize / 2.0f);

    for (unsigned int i = 0; i < Vector3::SIZE; ++i)
    {
        float delta = object->boundingVolume->center[i] - octreeNode->center[i];

        if (std::abs(delta) < object->boundingVolume->radius[i])
        {
            straddle = true;
        }

        if (delta > 0.0f)
        {
            index |= 0x1 << i;
            offset[i] *= -1;
        }
    }

    // 0 -> 0b000 -> |  left | down |  far |
    // 1 -> 0b001 -> | right | down |  far |
    // 2 -> 0b010 -> |  left |  up  |  far |
    // 3 -> 0b011 -> | right |  up  |  far |
    // 4 -> 0b100 -> |  left | down | near |
    // 5 -> 0b101 -> | right | down | near |
    // 6 -> 0b110 -> |  left |  up  | near |
    // 7 -> 0b111 -> | right |  up  | near |

    if (!straddle) // Proxy fully contained by the node.
    {
        if (octreeNode->children[index] == nullptr) // Create new node as needed.
        {
            octreeNode->children[index] = new OctreeNode();
            octreeNode->children[index]->center = octreeNode->center + offset;
            octreeNode->children[index]->halfSize = octreeNode->halfSize / 2.0f;
            for (int i = 0; i < OctreeNode::NUMBER_OF_CHILDREN; ++i)
            {
                octreeNode->children[index]->children[i] = nullptr;
            }
            octreeNode->children[index]->objectList = nullptr;
        }

        Insert(octreeNode->children[index], object);
    }
    else // Proxy outside the node.
    {
        object->nextObject = octreeNode->objectList;
        octreeNode->objectList = object;
    }
}
```

Listing 2.13: Implementation of object insert operation on an octree in C++;
code based on third and fourth listings from the section 7.3.2 in [8]

In the first step, the Insert procedure, presented in the listing 2.13, calculates straddle, index and offset variables. The straddle variable is a flag, indicating whether an object is encapsulated by any of sub-cubes entirely or straddles any partitioning plane. Thus, a value of this flag can be obtained by checking if an object does not straddle any partitioning plane. Such test is sufficient, because it is assumed that testing object is encapsulated by parent node's cube. If this would not be true, it would not descend into the parent node. Testing whether an object straddles partitioning plane can be implemented by checking

if radius of object's BV is shorter than distance between center of object's BV and partitioning plane. The figure 2.23 visualizes logic of the algorithm calculating straddle flag.

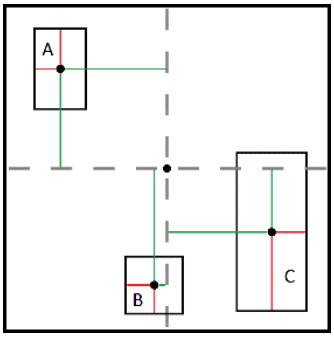


Figure 2.23. The upper left child encapsulates the object A and objects B and C are straddling partitioning lines

Basically the algorithm tests if all green lines outgoing from object's center are longer than red lines outgoing from object's center. Green lines are stored in the delta variable in the listing 2.13 and red lines are obviously radiuses of objects' BVs. Obtaining these lengths is done by accessing proper components of vectors, because partitioning planes, volumes of nodes and object's BV are axis aligned. The second variable, called in the listing 2.13 index, is used to access node, which encapsulates object if the straddle flag is false. Obtaining index can be done by checking signs of delta variables. Then sign of each delta is converted into bit flag and written into the index, which finally constructs value in binary code. Detailed table explaining this conversion can be seen in the comment in the listing 2.13. The last variable, which is calculated, is called offset. This variable stores vector by which center of parent's cube has to be moved to become center of child's cube. Only one offset is needed, because only one object is inserted into the octree at the time, thus it can only descend into only one node. Magnitudes of offsets along each axis of the offset vector are equal to half of parent cube's radius. The direction of offset vector should also be chosen according to signs of each delta, thus value of the index also determines direction of the offset. After all these variables have been calculated, the algorithm checks if the object can be descendent deeper into the octree or straddles partitioning plane and has to be assigned at the beginning of the list of objects belonging to the current node. Descending deeper into the octree is preceded by creating new node if it has not been created previously. Descending itself is regular recursive call into proper child node.

Detecting collisions accelerated by an octree, similarly to detecting collisions using other data structures is focused on effective pruning of unnecessary collision tests. Similarly to other data structures accelerating broad phase, also octree has powerful property allowing it to omit great number of redundant tests. Consider a node in an octree. Objects assigned to that node can collide only with objects assigned to the node itself, ancestors and descendants of that node. The figure 2.24 presents an example world, with marked nodes from the quadtree that needs testing against node containing object S.

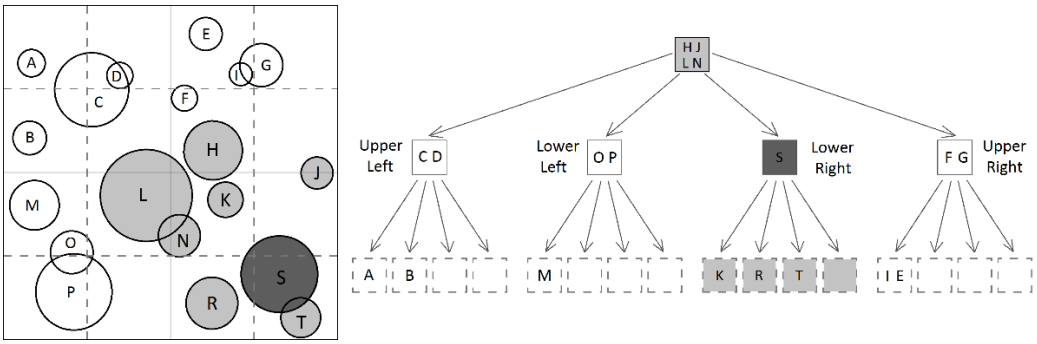


Figure 2.24. Visualization of nodes needing testing when objects from example node are checked

The reason, why only ancestors and descendants have to be checked is a combination of two facts. Firstly, ancestors and descendants of particular node are only nodes, which overlap that node. This can be seen on the left side of figure 2.24. Secondly, objects are always assigned to all nodes on a particular level, which they overlap, independently from assigning rule. Because collisions can occur only between objects from overlapping nodes only ancestors and descendants have to be taken into consideration. Understanding how octree accelerates collision detection, procedure detecting collisions can be explained. The listing 2.14 presents example implementation of the DetectCollisions procedure implemented in C++ language.

```
void DetectCollisions(OctreeNode* octreeNode, Collection<OctreeNode*>& ancestors,
                                              Collection<Pair<Object*>>& collisions)
{
 ancestors.PushBack(octreeNode);

 for (int i = 0; i < ancestors.Size(); ++i)
 {
 for (Object* objectA = ancestors[i]->objectList; objectA; objectA = objectA->nextObject)
  {
   for (Object* objectB = octreeNode->objectList; objectB; objectB = objectB->nextObject)
   {
    if (*objectA == *objectB)
    {
     break;
    }
    else
    {
     if (BoundingVolume::Overlap(objectA->boundingVolume, objectB->boundingVolume))
     {
      collisions.Insert(Pair<Object*>(objectA, objectB));
     }
    }
   }
  }
 }

 for (int i = 0; i < OctreeNode::NUMBER_OF_CHILDREN; ++i)
 {
  if (octreeNode->children[i] != nullptr)
  {
   DetectCollisions(octreeNode->children[i], ancestors, collisions);
  }
 }

 ancestors.PopBack();
}
```

Listing 2.14: Example implementation of procedure detecting collisions utilizing an octree in C++; code based on the last listing from the section 7.3.2 in [8]

Procedure from the listing 2.14 takes as an input, a node in an octree and a collection of ancestor nodes, sorted from top to bottom, which contains all nodes on the path from root to the input node's parent. Then procedure detects collisions between objects assigned to the input node and all its ancestors by checking all possible pairs of objects. In order to simplify code and detect collisions between objects assigned to the input node, ancestors of the input node contains the input node itself. When pair of object's BVs overlap, then pair of possible colliding objects is detected and that pair is inserted into collection of possible collisions, which is an output from the DetectCollisions procedure. Subsequently for all child nodes of the input node DetectCollisions procedure is recursively called. The reason for that is it is assumed that procedure from listing 2.14 detects all collisions between objects assigned to the input node and descendants of the input node. Because of that, the DetectCollisions procedure can be used to detect collisions between objects assigned to child nodes of the input node and objects assigned to descendant nodes. This solution allows detecting collisions between objects belonging to the input node and its ancestors explicitly and between its children implicitly. An important fact to realize is that objects assigned to nodes from sibling sub-trees cannot collide, thus calling recursively the DetectCollisions procedure only for child nodes is sufficient. Finally, the DetectCollisions procedure removes the input node from collection of ancestors to clean up collection and gets it ready to be used by other nodes. The last piece of information concerning functional aspect of collision detection is a method of detecting collisions in the whole octree. This can be achieved by simply calling the DetectCollisions procedure with the root node of an octree as an input node and with empty ancestors and collisions collections.

Performance of collision detection accelerated by an octree is dictated by computational complexity of the procedure detecting collisions. In the worst case scenario, when all objects straddle partitioning planes of the root node, procedure is characterized by the quadratic time complexity. However, when lists in each node are kept short and it can assumed that they are bounded by a constant value, then time complexity is reduced significantly. The total time complexity of detecting all collisions in an octree is equal to the time complexity of one call, excluding any recursive calls, of the DetectCollisions procedure multiplied by the total number of recursive calls. Assuming that lists of objects are bounded by a constant value, time complexity of one call of the DetectCollisions procedure becomes linear in terms of number of input node's ancestors. The number of ancestors of any node is always smaller than height of the octree. Height of a *d*-nary tree can be calculated using formula (1.5).

$$\boldsymbol{h = \log_d l} \tag{1.5}$$

h – height of d-nary tree [nodes]

d – number of child nodes per node [nodes]

l – number of leafs in d-nary tree [nodes]

In one call of the DetectCollisions procedure, the number of recursive calls is equal to the number of children per node. Because collision detection starts from calling the DetectCollisions procedure with the root node of an octree as input node, it is clear that the total number of calls to the DetectCollisions procedure is equal to the number of nodes in an octree. To calculate number of nodes in a d-nary tree one could use formula (1.1). Combining formula (1.5) and formula (1.1) and knowing that octree is an 8-nary tree, estimated time complexity of the DectectCollisions procedure can be calculated.

$$h \cdot n = \log_8 l \, \frac{8^{\log_8 l + 1} - 1}{7} = \log_8 l \, \frac{8l - 1}{7} \qquad (1.6)$$

h – height of an octree [nodes]

n – number of nodes in an octree [nodes]

l – number of leafs in an octree [nodes]

The formula (1.6) shows that time complexity of the DetectCollisions procedure is *O(l·log(l))* in terms of number of leafs in an octree. In well-built octree the number of objects should be proportional to the number of leafs in an octree, which is equal to number of smallest cells. Therefore, complexity of the DetectCollisions procedure can be finally estimated to be *O(n·log(n))* in terms of number of objects.

2.3. Spatial Sorting

Spatial Sorting is a group of broad phase algorithms, which utilize some variant of a collection containing objects sorted spatially. This data structure allows expressing spatial arrangement of objects without negative consequences of keeping information about assignment of objects to cells. This lack of cells is the main difference between spatial sorting and spatial partitioning. The most fundamental differences between methods classified as a spatial sorting, are ways in which spatial arrangement is expressed and the type of data structures used to store spatially sorted objects. Sometimes spatial sorting methods are also called coherence exploiting methods, because they utilize the fact that in short time interval scene is not changing drastically. Thanks to that property, collection of objects sorted with respect to the beginning of short time interval is partially sorted compared to the collection of objects sorted with respect to the end of that time interval. Therefore, preserving previous content of that collection allows using it as an input for updating collection, for example by the insertion sort. Spatial partitioning methods usually do not handle clustered objects, because assumption of temporal coherence of the scene breaks in that scenario. When objects are clustered together small changes in objects' positions causes significant changes in their order. In that case, the previous state of sorted collection cannot be treated as an approximation of the next state of that collection. In other words sorted collection form moment ago is not partially sorted current collection.

2.3.1. Sweep And Prune

The following description of the Sweep And Prune algorithm is based on the chapter 7.5 from [8]. The predominant method of spatial sorting is the Sweep And Prune method, also called the Sort And Sweep method, in which spatial arrangement is expressed as projections of objects onto chosen axes. These methods vary in terms of number of projections, algorithms used to choose axes for these projections, chosen data structure storing projections and sorting algorithm used to maintain these representations of projections. The figure 2.25 presents the SAP method in 2D space with one, arbitrary chosen, projection axis.

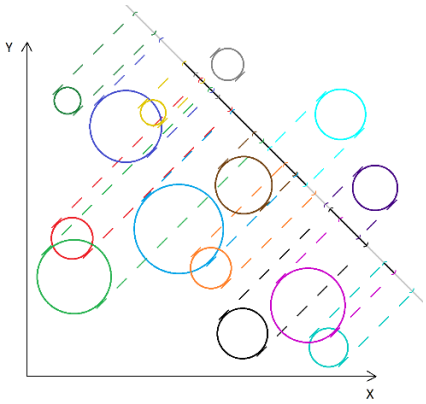


Figure 2.25. Example of sweep and prune on a plane utilizing one projection axis

Sweep and prune methods utilize the fact that, in order for two objects to intersect, they have to overlap on all projections used by this method. Because in the figure 2.25 SAP algorithm uses only one projection axis, all objects that are suspected for collisions, from one projection, have to be checked. Fortunately, increasing the number of projections decreases number of false positive results, but at the expense of additional computations and memory required to maintain these extra projections. Obviously proper choice of axes for projections is very important, because its impact on pruning factor is extremely significant. Quite profitable method of selecting projection axes is to favor axes with greatest object's intervals variance, which can be calculated as a variance of set of centers of object's intervals. It is also important to mention that grouping objects causes projection to yield poor results, because minimal changes in scene causes large changes in order of object's projections. This situation breaks the assumption, that scene is characterized by the temporal coherence. Unfortunately clustering is quite common situation, for example when objects settle on some kind of terrain, they tends to cluster on Y axis. Example of such clustering can be noticed in the figure 2.26, where nearly every cube is settled on the plane perpendicular to the Y axis.

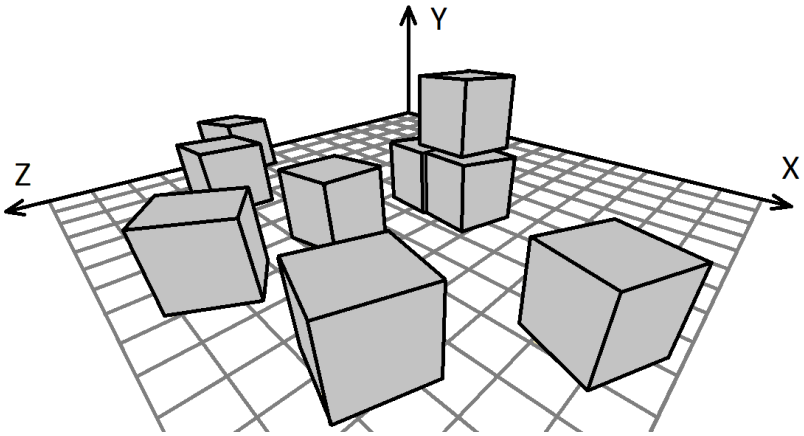


Figure 2.26. Objects clustering on the y axis due to settling on the terrain

Choosing axis, on which clustering does not occur can help with particular geometry configuration, but it is not a general solution.

In the SAP algorithm, the projection is a sorted collection of endpoints indicating the beginning or the end of object's interval. Projections of objects onto axis can be used to create object's intervals. However, calculating object's intervals can be simplified by projecting objects' BVs instead of objects' actual geometry. Popular choice of BV type and projection axes are AABB and coordinate axes,

because this approach allows to greatly simplify projection operation, reducing it to extracting coordinates of minimal and maximal points defining AABB. Intervals located on all axes belonging to one particular object form an AABB of that object. This is obviously the case, when intervals are created by projecting object's AABB onto axes, but it is also true for intervals created by projecting the geometry of an object itself or its BV, not necessarily AABB. The projection is a sorted collection, in which interval endpoints have ascending order with respect to endpoint's coordinate occupied on the projection axis. The following descriptions of implementations are based on the section 7.5.1 from [8]. The most common way of implementing projection is to model it as a list of endpoints. The figure 2.27 presents projections represented as lists.

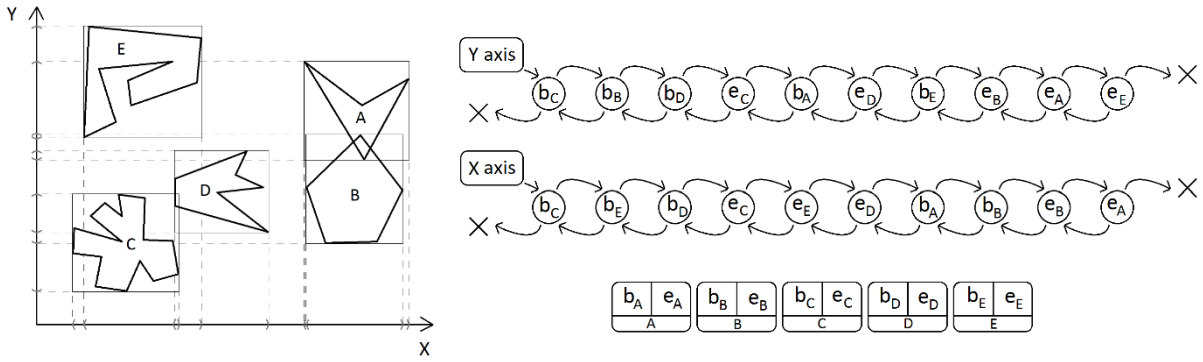


Figure 2.27. Example projections onto x and y axes represented as double linked lists

To represent these projections, one sorted list of interval endpoints per projection axis has to be created. In addition to linked lists, an array of bounding boxes, which links endpoints of object's intervals, has to be created. The listing 2.15 shows implementation of these data structures in C++ language.

```
struct Interval
{
    EndPoint* begin;
    EndPoint* end;
};

struct AABB
{
    Vector3<Interval> intervals;
    Object*           object;
};

enum class EndPointType : bool { BEGIN, END };

struct EndPoint
{
    AABB*        aabb;
    EndPoint*    nextEndPoint;
    EndPoint*    prevEndPoint;
    float        coordinate;
    EndPointType type;
};

struct SweepAndPrune
{
    AABB*              objectsAABBs;  // Array of object's AABBs
    Vector3<EndPoint*> projListHeads; // Heads of projection lists
};
```

Listing 2.15: Projections represented as linked lists in C++;
code based on the first listing from the section 7.5.1 in [8]

Implementation from the listing 2.15 is very explicit, which makes it easy to understand, but also is quite ineffective, because multiple data structures express one logical data structure. Better approach, than presented in the listing 2.15, is to merge all this data structures into one array, which would contain all linked lists. Elements of that array would be object's AABBs containing minimal and maximal points defining AABB. These points would be created as merged minimal or maximal endpoints. The figure 2.28 presents visual representation of refined projection model.

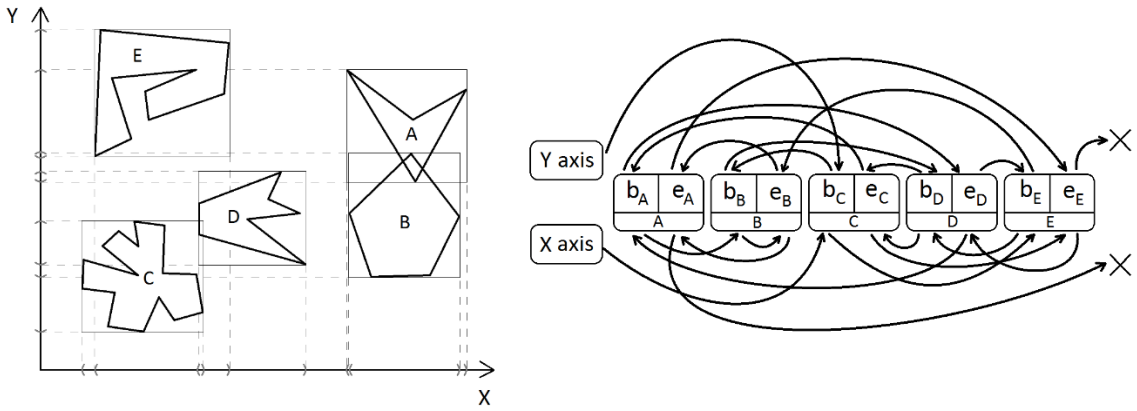


Figure 2.28. Refined SAP data structure modeled as array of object's AABBs containing lists for all projections

In the right side of the figure 2.28, an array of object's AABBs with links is presented. Despite the lack of visual appeal and clarity, this representation is much more effective for a computer to handle. The listing 2.16 shows refined version of the code implemented in C++ language.

```
struct AABB
{
    Point   min;
    Point   max;
    Object* object;
};

enum class PointType : bool { BEGIN, END };

struct Point
{
    Vector3<Point*> nextPoint;
    Vector3<Point*> prevPoint;
    Vector3<float>  coordinates;
    PointType       type;
};

struct SweepAndPrune
{
    AABB*           objectAABBs;   // Array of object's AABBs
    Vector3<Point*> projListHeads; // Heads of projection lists
};
```

Listing 2.16: Projections merged into array of object's AABBs;
code based on the fourth listing from the section 7.5.1 in [8]

The refined version of the code in the listing 2.16 allocates only one block of memory, which contains three lists for all axes with shared nodes combined into AABBs of all objects. What is interesting is links of these lists can point only to members of elements of the array, which minimizes spreading of list nodes across memory. In addition to merging interval's endpoints into points and allocating all shared elements of the lists in an array, the link inside of points pointing back to object's AABB was removed.

Removing this link was possible because of continuous characteristics of array's memory and proper packing of structures. Minimal points are always placed before maximal points, therefore object's AABB variable can be obtained by offsetting address of maximal element by the size of the point structure or just reinterpreting minimal element as AABB variable. Listing 2.17 shows example implementation of obtaining object's AABB from point.

```
AABB* GetAABB(Point* point)
{
    return reinterpret_cast<AABB*>(point->type == PointType::BEGIN ? point : point - 1);
}
```

Listing 2.17: Projections merged with AABBs of objects;
code based on the fifth listing from the section 7.5.1 in [8]

Summing up, there are numerous advantages of merging data into one array, including decreased memory consumption, substantially improved caching and less complex code.

Before the description of operations on the presented SAP data structure, a short description of sentinel element has to be provided. Sentinel element is special kind of element, which contains two nodes, one placed at the beginning and other one placed at the end of each projection list. They are introduced to simplify code operating on lists and also allows code to be slightly faster, by omitting checking for the beginning or the end of a list. Building array of object's AABBs containing all projection lists begins with creating sentinel element and later adding next elements of the array. The figure 2.29 shows initialized array with links pointing to valid elements.

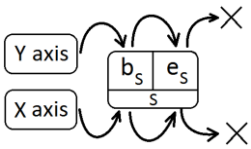


Figure 2.29. Initialized array with only sentinel element

Example implementation in C++ of procedure initializing the array using sentinel element can be seen in the listing 2.18.

```
void InitializeSAP(SweepAndPrune& sap)
{
        sap.objectAABBs = Collection<AABB>::OneItemCollection(AABB());

        AABB& sentinel = sap.objectAABBs[0];

        sentinel.min.type = PointType::END;
        sentinel.min.coordinates = Vector3<float>(std::numeric_limits<float>::lowest());
        sentinel.min.prevPoint = Vector3<Point*>(nullptr);
        sentinel.min.nextPoint = Vector3<Point*>(&sentinel.max);

        sentinel.max.type = PointType::BEGIN;
        sentinel.max.coordinates = Vector3<float>(std::numeric_limits<float>::max());
        sentinel.max.prevPoint = Vector3<Point*>(&sentinel.min);
        sentinel.max.nextPoint = Vector3<Point*>(nullptr);

        sap.projListHeads = Vector3<Point*>(&sentinel.min);
}
```

Listing 2.18: Implementation of procedure initializing array of object's AABBs using sentinel element;
code based on the sixth listing from the section 7.5.1 in [8]

The sentinel element is conveniently set as a first element of the array. Coordinates of sentinel's minimal node are set to lowest possible values to always stay at the beginning of all lists and its type is set to be

end of object's interval to do not cause any reports of collisions. On the other hand, sentinel's maximal node is set to be begin of object's interval to avoid any reports of collisions. To assure that sentinel's maximal node is always at the end of all lists its coordinates are set to the largest possible value.

Knowing that sentinel element contains nodes guarding endpoints of projection lists, operations of adding an object into SAP data structure can be described. In the first step of adding an object, its AABB is created and inserted into the array of object's AABBs. In the next step for each projection axis new minimal and maximal points defining object's intervals are inserted into projection lists, by finding their proper positions in lists and executing required operations on node's links. Finding proper position is required because after insertion all projection lists have to be sorted by endpoint's coordinates. The listing 2.19 shows an example implementation of insert operation in C++ language.

```
// Assumed that objectAABB has proper link to object and
// coordinates of minimal and maximal points are set to valid values.
void InsertObjectIntoSAP(SweepAndPrune& sap, const AABB& inputObject)
{
    sap.objectAABBs.Insert(inputObject);

    AABB& objectAABB = sap.objectAABBs[sap.objectAABBs.Size() - 1];

    objectAABB.min.type = PointType::BEGIN;
    objectAABB.max.type = PointType::END;

    for (unsigned int axis = 0; axis < 3; ++axis)
    {
        Point* point = sap.projListHeads[axis];

        // -+ -> +-----+ -> +-
        //  |    |point|    |
        // -+ <- +-----+ <- +-      -+ -> +----+ -> +-----+ -> +-
        //                      =>   |    |AABB|    |point|    |
        //       +----+ -> X        -+ <- +----+ <- +-----+ <- +-
        //       |AABB|
        //  X <- +----+

        // Insert minimum point into all projection lists.
        while (point->coordinates[axis] < objectAABB.min.coordinates[axis])
        {
            point = point->nextPoint[axis];
        }
        objectAABB.min.prevPoint[axis] = point->prevPoint[axis];
        objectAABB.min.nextPoint[axis] = point;
        point->prevPoint[axis]->nextPoint[axis] = &objectAABB.min;
        point->prevPoint[axis] = &objectAABB.min;

        // Insert maximum point into all projection lists.
        while (point->coordinates[axis] < objectAABB.max.coordinates[axis])
        {
            point = point->nextPoint[axis];
        }
        objectAABB.max.prevPoint[axis] = point->prevPoint[axis];
        objectAABB.max.nextPoint[axis] = point;
        point->prevPoint[axis]->nextPoint[axis] = &objectAABB.max;
        point->prevPoint[axis] = &objectAABB.max;
    }
}
```

Listing 2.19: Insert object operation implemented in C++;
code based on the seventh listing from section 7.5.1 in [8]

Order of object's intervals endpoints implies that projection axis is divided into two types of intervals: active intervals and inactive intervals. Active intervals are that parts of projection axis, where

objects are overlapping. In the figure 2.25, active intervals were marked using black line. Inactive intervals are parts of the projection axis where no overlapping occurs. Inactive intervals are marked with light gray line in the figure 2.25. The interesting parts of projection axis are active intervals, because they generate pairs of objects that are suspected for collisions. In order to detect collisions for all projection axes collection of pairs of objects overlapping on that particular projection has to be created. Then, to find the final collection of pairs of objects suspected for colliding a common subset of pairs has to be obtained from collections of pair of objects overlapping each on each projection. The listing 2.20 shows logic of the algorithm implemented in C++.

```
Collection<Pair<Object*>> CollisionDetectionSAP(const SweepAndPrune& sap)
{
 Vector3<Collection<Pair<Object*>>> overlappingObjects;

 for (unsigned int axis = 0; axis < 3; ++axis)
 {
  overlappingObjects[axis] = OverlappingObjectsOnProjectionAxis(sap.projListHeads[axis]);
 }

 return overlappingObjects.X && overlappingObjects.Y && overlappingObjects.Z;
}
```

Listing 2.20: Detecting colliding objects as and operation between overlapping objects on each partitioning axis

To find overlapping pairs of objects on a projection, corresponding axis has to be swept from beginning to end, which translates to iterating through the collection of endpoints, omitting sentinel nodes. In the beginning of the algorithm, empty collection of active objects has to be created. Then in each iteration, whenever the beginning of object's interval is encountered, for all active objects, pair containing active object and encountered object is created and added to the list of overlapping objects on particular projection. Lastly, encountered object becomes active object. However, when an iteration encounters the end of object's interval, encountered object is removed from the set of active objects. Each interval of projection axis, which has been swept with more than one active object is an active interval. The listing 2.21 shows example implementation of a procedure finding all pairs of objects, which are overlapping on a projection.

```
// Assumes projection sorted in ascending order by points positions on projection axis.
Collection<Pair<Object*>>
OverlappingObjectPairsOnProjectionAxisSAP(const Vector3<Point*>& ProjListHeads)
{
    Collection<Pair<Object*>> overlapping;
    Collection<Object*> activeObjects;

    for (Point* point = ProjListHeads.X->nextPoint.X;
         point->nextPoint.X != nullptr;
         point = point->nextPoint.X)
    {
        Object* object = GetAABB(point)->object;

        switch (point->type)
        {
            case PointType::BEGIN:
            {
                for (unsigned int j = 0; j < activeObjects.Size(); ++j)
                {
                    overlapping.Insert(Pair<Object*>(object, activeObjects[j]));
                }
                activeObjects.Insert(object);
                break;
            }
```

```
            case PointType::END:
            {
                activeObjects.Remove(object);
                break;
            }
        }
    }

    return overlapping;
}
```

Listing 2.21: Procedure finding all pair of objects overlapping in projection;
code based on the seventh listing from the section 7.5.1 in [8]

Algorithms from listings 2.20 and 2.21 are very easy to understand, but certainly are not efficient, because all projection axes are being iterated through and collections of object pairs suspected for colliding are created from scratch. Alternative method of detecting collisions is to maintain collection of overlapping objects for each projection, as projection lists changes. This is different from previous algorithm, which takes input data, processes it and returns output. The alternative algorithm reduces input data to convenient data structure and maintains its state by refining it content whenever its state becomes obsolete. In case of the SAP algorithm this data structure is a vector of three collection of pairs of objects, representing all pairs of overlapping objects on each projection. The listing 2.22 presents an example implementation of procedure updating state of pairs of overlapping objects on all projections.

```
void SyncStateOfOverlappingObjectPairsOnProjectionAxesSAP(SweepAndPrune& sap)
{
  // Just inserted object's AABB.
  AABB& objectAABB = sap.objectAABBs[sap.objectAABBs.Size() - 1];

  for (unsigned int axis = 0; axis < 3; ++axis)
  {
    for (Point* point = sap.projListHeads[axis]; ; point = point->nextPoint[axis])
    {
      if (point->type == PointType::BEGIN)
      {
        // left case form the figure 2.30
        if (objectAABB.max.coordinates[axis] < point->coordinates[axis])
        {
          break;
        }

        // cases other than right form the figure 2.30
        if (GetAABB(point)->max.coordinates[axis] > objectAABB.min.coordinates[axis])
        {
          if (AABB::Overlap(&objectAABB, GetAABB(point)))
          {
            Pair<Object*> overlappingPair(objectAABB.object, GetAABB(point)->object);
            sap.overlapingObjectsPairsOnAxes[axis].Insert(overlappingPair);
          }
        }
      }
    }
  }
}
```

Listing 2.22: Procedure finding all pair of objects overlapping in projection;
code based on the seventh listing from the section 7.5.1 in [8]

The procedure shown in the listing 2.22 should be called as the last operation in the InsertObjectIntoSAP procedure to maintain collection of colliding objects. The maintenance in case of inserting new object's AABB is reduced to detecting which objects in the world overlap newly added object on particular

projection. To obtain such an information, each axis is swept from its beginning and each object whose endpoint is currently visited is checked for overlapping with inserted object. Because projection list contains endpoints of object's intervals, iterating through projection list can be interpreted as iterating through all object's in the world twice. Therefore checking for overlapping between current object and inserted object has to be done when visiting the beginning or the end of object's interval. In the listing 2.22 it was chosen to check for overlapping objects when the beginning of object's interval is encountered. However not all objects has to be checked, which is presented in the figure 2.30.

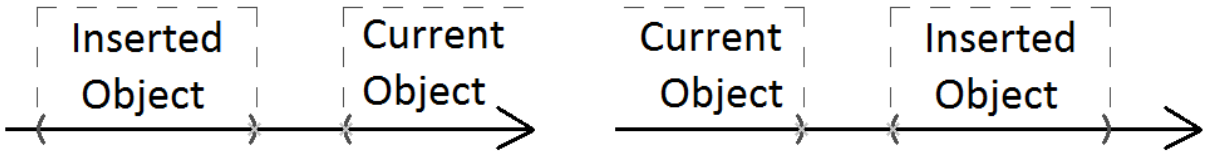


Figure 2.30. Two cases of interval alignment without overlapping

The figure 2.30 shows two cases in which objects cannot overlap, thus checking for overlap is unnecessary. In addition to that optimization, premature exit from the loop iterating through projection list is possible. Because objects are checked for overlapping when beginning of object's interval is encountered when case presented on the left side of the figure 2.30 is encountered it is no longer needed to iterate though projection list and loop can exit prematurely. The reason for that is there are two types of objects that can be encountered when loop passed the end of inserted object's interval: objects whose interval begins after the end of inserted object's interval, which obviously cannot collide with inserted object, and objects whose interval begins before the end of inserted object's interval, which were handled before. Iterating through projection list can also end when the end of it is encountered, which should be regular condition terminating loop. However, the sentinel node assures that path, which breaks loop prematurely is always executed, thus regular condition stopping loop is not needed. After projections on each axis have been updated, common set of overlapping pairs has to be created by previously introduced procedure.

The operation of deleting an object is relatively simple to implement. Operations required to be executed for object's deletion on the SAP data structure are straightforward: endpoints of the object's interval has to be removed from all projection lists, object's AABB has to be removed from the array of object's AABBs and lastly all pairs of overlapping objects containing object, which is deleting has to be removed. Much more complex procedure has to be developed in case of updating SAP data structure after an object changed its position in the world. Similarly, to the insertion operation, the update operation also can be reduced to updating state of projection lists in the array of object's AABBs and the collections of pairs of overlapping objects. General scheme of updating algorithm is trying to move object's intervals endpoints from previous positions in left and right directions to find proper positions in each of the projection list. While endpoints are moving, algorithm tracks all encountering endpoints, which require modifying collection of colliding pairs of objects. The listing 2.23 shows example implementation of that algorithm in C++ language.

```
// Assumes that coordinates of AABB are already updated.
// Function updates projection lists and collection of colliding objects.
void UpdateObjectAABBPositionSAP(SweepAndPrune& sap, AABB* objectAABB)
{
    // For all three axes corresponding to projection lists.
    for (unsigned int axis = 0; axis < 3; ++axis)
```

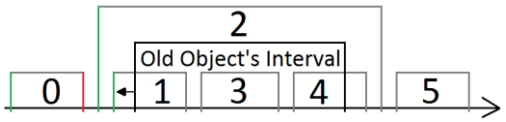
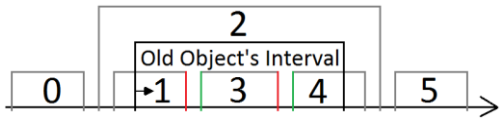
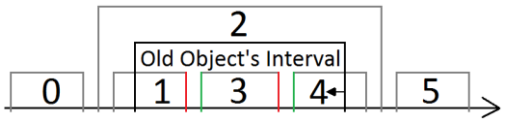
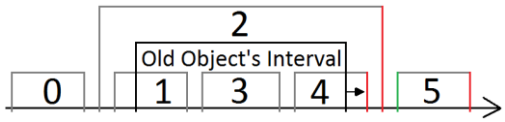
```
{
    Point& minAABB = objectAABB->min;
    Point& maxAABB = objectAABB->max;

    // Roaming pointer iterates through projection list,
    // searching for insertion position.
    Point* roamingPointer = nullptr;

    // Try to move minimum point to the left. Move roaming pointer until
    // the minimum point coordinate is larger than roaming point coordinates.
    // This position is the insertion position. While doing so,
    // keep track of update status of any passed points.
    for (roamingPointer = minAABB.prevPoint[axis];
        minAABB.coordinates[axis] < roamingPointer->coordinates[axis];
        roamingPointer = roamingPointer->prevPoint[axis])
    {
        AABB* roamingAABB = GetAABB(roamingPointer);
        Pair<Object*> objectPair(roamingAABB->object, objectAABB->object);

        if (roamingPointer->type == PointType::END)
            if (AABB::Overlap(roamingAABB, objectAABB))
                if (!sap.collidingObjectsPairs.Contains(objectPair))
                    sap.collidingObjectsPairs.Insert(objectPair);
    }
    if (roamingPointer != minAABB.prevPoint[axis])
        MovePointAfterDestAtAxis(&minAABB, roamingPointer, axis);

    // Try to move maximum point to the right.
    for (roamingPointer = maxAABB.nextPoint[axis];
        maxAABB.coordinates[axis] > roamingPointer->coordinates[axis];
        roamingPointer = roamingPointer->nextPoint[axis])
    {
        AABB* roamingAABB = GetAABB(roamingPointer);
        Pair<Object*> objectPair(roamingAABB->object, objectAABB->object);

        if (roamingPointer->type == PointType::BEGIN)
            if (AABB::Overlap(roamingAABB, objectAABB))
                if (!sap.collidingObjectsPairs.Contains(objectPair))
                    sap.collidingObjectsPairs.Insert(objectPair);
    }
    if (roamingPointer != maxAABB.nextPoint[axis])
        MovePointAfterDestAtAxis(&maxAABB, roamingPointer, axis);

    // Try to move minimum point to the right.
    for (roamingPointer = minAABB.nextPoint[axis];
        minAABB.coordinates[axis] > roamingPointer->coordinates[axis];
        roamingPointer = roamingPointer->nextPoint[axis])
    {
        AABB* roamingAABB = GetAABB(roamingPointer);
        Pair<Object*> objectPair(roamingAABB->object, objectAABB->object);

        if (roamingPointer->type == PointType::END)
            if (sap.collidingObjectsPairs.Contains(objectPair))
                sap.collidingObjectsPairs.Remove(objectPair);
    }
    if (roamingPointer != minAABB.nextPoint[axis])
        MovePointAfterDestAtAxis(&minAABB, roamingPointer, axis);

    // Try to move maximum point to the left.
    for (roamingPointer = maxAABB.prevPoint[axis];
        maxAABB.coordinates[axis] < roamingPointer->coordinates[axis];
        roamingPointer = roamingPointer->prevPoint[axis])
    {
        AABB* roamingAABB = GetAABB(roamingPointer);
        Pair<Object*> objectPair(roamingAABB->object, objectAABB->object);

        if (roamingPointer->type == PointType::BEGIN)
            if (sap.collidingObjectsPairs.Contains(objectPair))
```

```
                    sap.collidingObjectsPairs.Remove(objectPair);
            }
            if (roamingPointer != maxAABB.prevPoint[axis])
                MovePointAfterDestAtAxis(&maxAABB, roamingPointer, axis);
        }
    }
```

Listing 2.23: Procedure updating SAP data structures after object changed position; code based on the eighth listing from the section 7.5.1 in [8]

The most outer loop allows to update all collections of overlapping pairs of objects for each axis. In each iteration of that loop, endpoints of object's interval are attempted to move. Because there are two ways in which endpoints can be moved and there are two endpoints in every interval, each iteration tries to move endpoints four times. The table 2.1 presents all attempts in each iteration.

Table 2.1. Directions in which endpoints can move.

	Left direction	Right direction
Minimal endpoint		
Maximal endpoint		

Each figure in the table 2.1 shows in which direction, which endpoint of object's interval is moved. Also all relevant minimal endpoints, marked using green lines, and maximal endpoints, marked using red lines, are distinguished and have to be considered whether they do not introduce new overlapping pair or makes existing collision disappear. List of all cases is the following:

- Minimal endpoint moved in the left direction:
  - Encountering minimal endpoints of intervals corresponding to objects 0, 1 or 2 do not provide additional information, because whether they do or do not overlap interval of updated object depends on the position of its maximal endpoint. Therefore, this case is solved when maximal endpoint is moved to the left and encounters minimal endpoint.
  - Encountering maximal endpoint require checking for overlapping, because it is possible for interval of object 0 to overlap interval of updated object.
- Minimal endpoint moved in the right direction:
  - Encountering minimal endpoints of intervals corresponding to objects 3 or 4 do not provide additional information, because whether they do or do not overlap interval of updated object depends on the position of its minimal endpoint. Therefore, this case is solved when minimal endpoint is moved to the right and encounters maximal endpoint.
  - Encountering maximal endpoints of intervals corresponding to objects 1 or 3 implies that they do not overlap updated object and all pairs containing updated object and any object from mentioned set have to be removed.

- Maximal endpoint moved in the left direction:
  - Encountering minimal endpoints of intervals corresponding to objects 3 or 4 implies that they do not overlap updated object and all pairs containing updated object and any object from mentioned set have to be removed.
  - Encountering maximal endpoints of intervals corresponding to objects 1 or 3 do not provide additional information, because whether they do or do not overlap interval of updated object depends on the position of its maximal endpoint. Therefore, this case is solved when maximal endpoint is moved to the left and encounters minimal endpoint.
- Maximal endpoint moved in the right direction:
  - Encountering minimal endpoint require checking for overlapping, because it is possible for interval of object 5 to overlap interval of updated object.
  - Encountering maximal endpoints of intervals corresponding to objects 2, 4 and 5 do not provide additional information, because whether they do or do not overlap interval of updated object depends on the position of its minimal endpoint. Therefore, this case is solved when minimal endpoint is moved to the right and encounters maximal endpoint.

Obviously each of minimal and maximal endpoints belonging to one object will be moved only in one direction in each projection, thus after each loop moving roaming pointer in the listing 2.23, endpoint belonging to updated object can be moved to its proper position in the projection list. The procedure handling moving that endpoint from outdated to proper position is presented in the listing 2.24.

```
void MovePointAfterDestAtAxis(Point* point, Point* destn, unsigned int axis)
{
    // -+ -> +-----+ -> +-     -+ -> +-----+ -> +-
    //  |    |destn|    |  ...  |    |point|    |
    // -+ <- +-----+ <- +-     -+ <- +-----+ <- +-
    //                      |
    //                      V
    // -+ -> +-----+ -> +-----+ -> +-     -+ -> +-
    //  |    |destn|    |point|    |  ...  |    |
    // -+ <- +-----+ <- +-----+ <- +-     -+ <- +-

    // Unlink point
    point->prevPoint[axis]->nextPoint[axis] = point->nextPoint[axis];
    point->nextPoint[axis]->prevPoint[axis] = point->prevPoint[axis];

    // Insert point after dest.
    point->prevPoint[axis] = dest;
    point->nextPoint[axis] = dest->nextPoint[axis];
    dest->nextPoint[axis]->prevPoint[axis] = point;
    dest->nextPoint[axis] = point;
}
```

Listing 2.24: Procedure moving particular point after destination point implemented in C++; code based on the last listing from the section 7.5.1 in [8]

It is important to realize that objects can be transformed in three ways. They can be translated (moved), rotated and scaled. Because of that, it is incorrect to assume that size of object's interval cannot change, therefore endpoints can move to opposite or matching directions. This could potentially break the assumption that minimal endpoint has smaller coordinate than maximal endpoint coordinate. To avoid this situation algorithm tries to move endpoints outwards, which is guaranteed to do not swap endpoints order. After that attempt, endpoints are moved safely inwards. This is the reason why maximal endpoints of objects 2, 4 and 5 and minimal endpoint of object 5 do not have to be considered when minimal

endpoint of updated object's interval is moved in the right direction. They simply cannot be encountered, because updated object's interval will not be flipped. Analogously minimal endpoints of objects 0, 1 and 2 and maximal endpoint of object 0 were not considered when maximal endpoint of updated object's interval was moved to the left.

Presented here algorithms maintaining SAP data structures implicitly realize insertion sort. This sorting algorithm is quite effective in case of already or nearly sorted lists, with expected linear time complexity in terms of number of elements in the list. In addition to that when scene is not dynamic very little work is required to detect collisions. However, objects' clustering breaks the assumption of temporal coherence of the scene, causing the performance of insertion sort to degrade to quadratic time complexity.

3. BULLET LIBRARY DESCRIPTION AND OCTREE IMPLEMENTATION

The theoretical analysis of an algorithm is extremely powerful tool, which allows studying fundamental properties of an algorithm. Calculating time and space complexities allows understanding its behavior for the large amount of data. Algorithm complexities are commonly calculated for the worst, average and the best case scenarios, because algorithms can process input data differently depending on specific properties of an input data. However calculating algorithm complexity, especially in the average case, can be very difficult to perform. Additionally, in the average case scenario, it is assumed that each possible input data can occur with equal probability, which can be substantially inaccurate, compared to real world distribution of an input data. In order to measure algorithm's behavior in the real world scenarios, usually empirical tests are performed. These tests can focus on many aspects of an algorithm, like behavior in typical cases, commonly occurring in an application. The theoretical analysis can also provide knowledge about properties of an input data, which cause changes in the performance of an algorithm. How strongly these properties affect the performance can be measured by empirical tests. In addition to that, empirical tests can be treated as verification of conclusions drawn from the theoretical analysis. Therefore, it is helpful to think of theoretical analysis and empirical tests not as opposite approaches, but as complementary tools, which boost their value, when combined.

In order to perform empirical tests working implementation of a physics engine is required. As stated in the introduction, implementing physics engine yourself is very time consuming work, thus making it impractical. The best way to perform empirical tests is to choose an existing implementation of a physics engine and use it as foundation for testing framework. Because performance of algorithms used in a physics engine will be subject of measurements, there are requirements for physics engine. First of all physics engine should have as many broad phase algorithms as possible. Secondly, physics engine should have license allowing its use for educational purposes and open-source license is also a strong point. Lastly, it is desirable for a physics engine to be relatively easy to work with. The Bullet Library satisfies all of these requirements, thus it was chosen for this project.

3.1. Bullet Library description

The Bullet Library is an open source, platform independent physics engine written in portable C++03. Availability of the Bullet Library under free and open-source zlib license allows its wide commercial and non-commercial use across whole graphics industry, including 3D rendered movies, video games, special effects, CAD applications and even benchmarks. The most spectacular examples

are Rockstar Advanced Game Engine (RAGE) used in Grand Theft Auto IV and V by Rockstar Games [1] and How To Train Your Dragon by Dreamworks [7]. The Bullet Library was designed to be modular and very easy to use, which is extremely valuable properties of any software. The modularity of the Bullet Library is very useful, because allows usage of only needed parts. When only collision detection module is useful, an application can incorporate only that module, leaving physics simulation to other software. This modular architecture can be seen in the introduction chapter in the figure 1.3. It is not obvious that real world implementation of a physics engine would incorporate this architecture, which shows only required dependencies. Another strong point of the Bullet Library is its ease of use and modularity inside each module, resulting from usage of object oriented interfaces and inheritance across whole physics engine. The Bullet Library is an excellent example of proper usage of C++ language. Interfaces and classes abstract unnecessary details of implementations, making the code very simple to work with and easy to understand, but at the same time very efficient, thanks to the explicit control over memory and code available in C++ language. To make these advantages more apparent and support mentioned claims, the listing 3.1 shows complete example of physics simulation written in 73 lines of code.

```
#include <iostream>
#include "btBulletDynamicsCommon.h"

btRigidBody* CreateBody(btCollisionShape* shape, btVector3& origin, btScalar mass)
{
  btTransform transform;
  transform.setIdentity();
  transform.setOrigin(origin);
  btDefaultMotionState* motionState = new btDefaultMotionState(transform);

  btVector3 localInertia(0.0, 0.0, 0.0);
  if (mass != 0.0) // Static object do not have mass.
  {
    shape->calculateLocalInertia(mass, localInertia);
  }

  btRigidBody::btRigidBodyConstructionInfo info(mass, motionState, shape, localInertia);
  return new btRigidBody(info);
}

int main()
{
  btCollisionConfiguration* collisionConfig = new btDefaultCollisionConfiguration();
  btDispatcher* dispatcher                  = new btCollisionDispatcher(collisionConfig);
  btBroadphaseInterface* broadPhase         = new btDbvtBroadphase();
  btConstraintSolver* constraintSolver      = new btSequentialImpulseConstraintSolver();

  btDynamicsWorld* dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher, broadPhase,
                                                               constraintSolver, collisionConfig);
  dynamicsWorld->setGravity(btVector3(0.0, -10.0, 0.0));

  btRigidBody* ground = CreateBody(new btBoxShape(btVector3(40.0, 4.0, 40.0)),
                                   btVector3(0.0, -40.0, 0.0), 0.0);
  dynamicsWorld->addRigidBody(ground);

  btRigidBody* ball = CreateBody(new btSphereShape(1.0), btVector3(0.0, 0.0, 0.0), 1.0);
  dynamicsWorld->addRigidBody(ball);

  for (unsigned int step = 0; step < 6; ++step)
  {
    dynamicsWorld->stepSimulation(1.0 / 60.0, 10);

    for (unsigned int i = 0; i < dynamicsWorld->getNumCollisionObjects(); ++i)
```

```
    {
      btCollisionObject* object = dynamicsWorld->getCollisionObjectArray()[i];
      btTransform& transform = object->getWorldTransform();

      std::cout << "[" << i << "] -> (";
      std::cout << transform.getOrigin().getX() << " ";
      std::cout << transform.getOrigin().getY() << " ";
      std::cout << transform.getOrigin().getZ() << ")";
      std::cout << std::endl;
    }
  }

  for (unsigned int i = 0; i < dynamicsWorld->getNumCollisionObjects(); ++i)
  {
    btCollisionObject* object = dynamicsWorld->getCollisionObjectArray()[i];
    btRigidBody* body = btRigidBody::upcast(object);
    if (body != nullptr && body->getMotionState())
    {
      delete body->getMotionState();
    }
    dynamicsWorld->removeCollisionObject(object);
    delete object;
  }

  delete dynamicsWorld;
  delete constraintSolver;
  delete broadPhase;
  delete dispatcher;
  delete collisionConfig;
}
```

Listing 3.1: Complete example of physics simulation implemented using the Bullet Library; code based on the Hello World example from the chapter 3 in [3]

It is remarkable how piece of software can be designed in such a way that very simple and clear code can perform extremely complicated task.

The following description of the Bullet library is mainly based on the source code analysis done by the author. However, the Bullet library documentation [4] generated from the source code also helped to develop this description. Lastly, the user manual available with the Bullet library [2] aided with description of more general concepts.

The Bullet Library consists of numerous classes and interfaces grouped into modules. In the most general view, presented in the figure 1.3, of the Bullet Library there are four major modules: the Linear Math, the Bullet Collision, the Bullet Dynamics and the Bullet Soft Body. These modules can be interpreted as layers, because each module is implemented in such a way that it depends on lower modules and is unaware of higher modules. For example, the Bullet Dynamics module creates simulation for rigid bodies utilizing collision detection provided by the Bullet Collision and mathematical entities defined in the Linear Math module. Obviously, the Bullet Dynamics module is independent from the Bullet Soft Body module. This layer architecture not only provides very clear separation of tasks performed by each layer, but also allows code to be naturally reused. However, layered architecture can cause abstraction to be overgrown, which makes code difficult to understand and slow to execute. That is not the case with the Bullet Library, because implementation of each layer is a set of few interfaces and classes with very thin abstraction. Thanks to optimizations present in C++ compilers, the thin abstraction nearly disappears in the runtime making code not only very fast, but also easy to understand. Because each module has only few classes and interfaces necessary to understand it is quite easy grasp the idea of how library works.

The most low-level layer is the Linear Math module, which provides availability of mathematical entities used in physics simulation. The figure 3.1 shows internal architecture of that module.

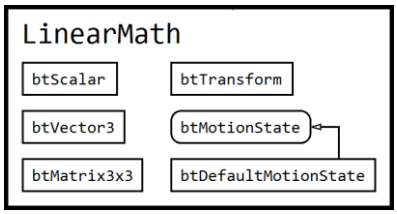


Figure 3.1. The architecture of the Linear Math module

Obviously, the figure 3.1 shows only most important entities, but at the same time shows enough to use and understand the Linear Math module.

- The btScalar is a type, which represents single value in floating point format. One could say that btScalar is unnecessary, because C++ provides float data type. This is not true, because btScalar can have single or double precision depending on client requirements. Additionally, the C++ language specification does not provide any assurance about exact precision of float and double data types. Thanks to abstraction of the btScalar, underlying type of calculations can be changed with ease. For example, some platforms provide 80-bit and 128-bit floating-point data types, which can be used as underlying data type for the btScalar.
- The btVector3 and btMatrix3x3 are data types representing vector and matrix respectively. As the names suggest the sizes of these data structures are fixed. In order to provide uniform precision across all core data types, these structures are aggregates of the btScalar type. What interesting about btVector3 and btMatrix3x3 is that their internal implementations can use SIMD instructions.
- The btTransform, as the name suggests, is a data type representing transformation of some object in the world. As mentioned before there are three types of transformation: translation, rotation and scale. The rotation and scale transformations can be represented as three by three matrix. The translation transformation is problematic, because only four by four matrix can incorporate translation. In order to minimize size of the btTransform structure it was chosen to store translation as a separate vector with three components.
- The btMotionState is an interface, which allows storing and updating snapshot of object's transformation. This is very useful data type for algorithms, which interpolate motion of an object. The btDefaultMotionState is a basic implementation of the btMotionState interface. It allows storing object's transformation in the beginning and the end of the motion interval and transformation of object's center of mass.

The Bullet Collision module is built on top of the Linear Math module and provides functionality of collision detection. Additionally, the Bullet Collision module allows obtaining information about contact points between colliding objects and expected time of impact between objects approaching each other. The figure 3.2 presents the architecture of the Bullet Collision module.

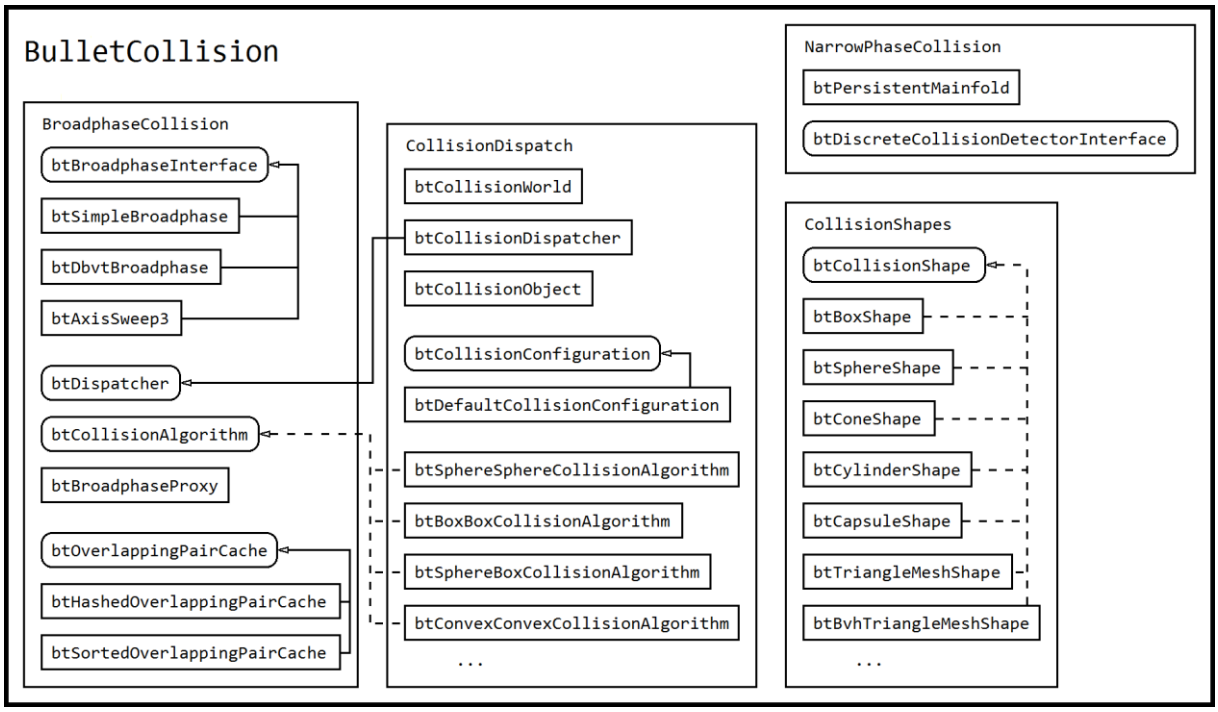


Figure 3.2. The architecture of the BulletCollision module with its sub-modules

The Bullet Collision module is quite complicated, because it needs to provide narrow and broad phases of collision detection and interfaces through which data is exchanged between the Bullet Collision module and a client code. Because of mentioned essential complexity four sub-modules where introduced to divide functionality.

- The Collision Shapes module defines data types representing shapes of objects based on which narrow phase of collision detection is performed. All these data types implement the btCollisionShape interface, which was designed to be used internally, by the Bullet library, and externally, by applications utilizing the Bullet Library. An example of the btCollisionShape, the btBoxShape and the btSphereShape usage can be seen in the listing 3.1. It is also interesting to note that classes in the Collision Shapes module create quite large hierarchy, which is an exception, because other hierarchies in the Bullet library are very flat, not exceeding three levels. The reason why all collision shapes create such large hierarchy is that mathematical entities are defined using previously defined mathematical entities. The inheritance diagram of collision shapes reflects that fact.
- The Narrow Phase Collision module implements interface through which data describing contact points and time of impact from narrow phase of collision detection can be obtained:
  - The btPersistentMainfold is a data type, which provides interface through which contact points can be obtained.
  - The btDiscreteCollisionDetectorInterface is an interface providing results of time of impact calculations.

  Apart from interfaces, the Narrow Phase Collision module contains various utilities, which are not presented in the figure 3.2, for sake of clarity.
- The Broad Phase Collision module provides set of interfaces and classes for broad phase collision detection, including utility cases and well as and interfaces designed to be used by applications utilizing the Bullet library.

- The btBroadphaseInterface is the most important interface, because it provides all operations required form a broad phase algorithm. In the Bullet library, there are three implementations of that interface: the btSimpleBroadphase, which is brute force algorithm checking all possible pairs of objects used mostly for debugging purposes, the btDvbtBroadphase, which is implementation of dynamic BVH and the btAxisSweep3, which is implementation of SAP method utilizing three arbitrary aligned axes.
- The btOverlappingPairCache is an interface for storing pairs of overlapping objects. This collection of overlapping pairs of objects is considered cache, because its state persists between runs of discrete collision detections and is updated only when needed. Additionally, the btOverlappingPairCache is a data type which btBroadphaseInterface returns as a result of broad phase collision detection.
- The btSortedOverlappingPairCache and btHashedOverlappingPairCache are two available implementations of the btOverlappingPairCache interface. The first implementation uses regular array as an internal storage for pairs of overlapping objects and sorts it when necessary. The second implementation also uses array as an internal storage, but also utilizes hashing algorithm to perform much faster search operations.
- The btBroadphaseProxy is a class representing simplified version of object's collision geometry, usually called bounding volume, utilized by the implementation of the btBroadphaseInteface to test whether objects are colliding or not. The btBroadphaseInteface implementation is unaware of the btCollisionObject and the btCollisionShape data types and interacts only with the btBroadphaseProxy data type. The Bullet library chose AABBs for BVs, in the btBroadphaseProxy implementation, because of they are very fast to create and produce acceptable level of tightness.
- The btCollisionAlgorithm in an interface, which defines required capabilities from an algorithm detecting collision between two objects in the narrow phase. Firstly, it needs to check whether two objects are intersecting. Secondly, it needs to calculate time of impact between pair of approaching object. Lastly, it needs to return all contact points between intersecting objects. The btCollisionAlgorithm uses utility class called btMainfoldResult, which is wrapper around the btPersistentMainfold from the Bullet Narrow Phase module, to return result of a collision algorithm.
- The btDispatcher is an interface designed to organize narrow phase of a collision detection. The core functionality provided by the btDispatcher is processing all pairs of overlapping objects in order to calculate whether pair returned by a broad phase is actually intersecting or not. In each step of processing pairs, the btDispatcher chooses proper implementation of the btCollisionAlgorithm from collision matrix, based on the btCollisionShape of both objects.

- The Collision Dispatch module is mainly responsible for dispatching collision data, thus manages the process of collision detection. The module itself consists of two major parts: a set of interfaces and classes organizing the process of collision detection and an implementation of, mentioned in the introduction, collision matrix in form of numerous collision algorithms.
  - The btCollisionObject is a data type representing an object placed in the virtual world, which can collide with other objects. In terms of implementation, the btCollisionObject is mainly an aggregate of various properties, which are need for proper collision detection. Notably the

btCollisionObject aggregates collision geometry in form of btCollisionShape, position in the world as the btTransform and bounding volume as the btBroadphaseProxy.

- The btCollisionWorld is the most important class from the point of view of application developers, because it provides interface and implementation for an object representing virtual world of objects on which the collision detection algorithms are applied. The most important capabilities of the btCollisionWorld are storing a collection of btCollisionObjects and performing discrete collision detection between them. In order to achieve that, the btCollisionWorld aggregates implementations of the btBroadphaseInterface to isolate all possible collision pairs and the btDispatcher to process all pairs of objects from broad phase in the narrow phase. Aside from detecting collisions between objects, the btCollisionWorld supports ray testing and collision queries concerning particular object. Overall, the btCollisionWorld can be understood as a wrapper of all entities in the Bullet Collision module, representing whole collision detection system and performing core calculations of virtual world simulation without any modeling physics.
- The btCollisionDispatcher is an implementation of the btDispatcher interface, allowing proper processing of the possible collision pairs in the narrow phase. As expected form an implementation of the btDispatcher interface, the btCollisionDispatcher dispatches all possible colliding objects pairs stored in the btOverlappingPairCache by calling the btCollisionAlgorithm on each one of them. Because, the btCollisionDispatcher stores collection of available btCollisionAlgorithms, it is able to find proper collision algorithm for pair of objects based on their types. Lastly, the btCollisionDispatcher stores all btPersistentMainfolds for all intersecting pairs of objects.
- The btCollisionConfiguration and the btDefaultConfiguration are an interface and an implementation of that interface respectively. The main purpose of these data types is to provide standard way of configuring the btCollisionDispatcher object. It mainly provides set of available btCollisionAlgorithms and allocation pools in which btCollisionAlgorithms and btPersistanceMainfolds are stored.
- The numerous implementations of the btCollisionAlgorithm form a collision matrix. As described earlier, the collision matrix defines which collision algorithm will be used, in the narrow phase, for detecting collision between pair of objects. In the Collision Dispatch module, there are numerous implementations of collision algorithms ranging from extremely simple ones like the btShpereSphereCollisionAlgorithm to much more advance algorithms like the btConvexConvexCollisionAlgorithm.

Considering the essential complexity of collision detection system and its impact on performance of whole physics engine, it can be said that the Bullet Collision module has very clean design, allowing developers familiar with theory and algorithms concerning collision detection to be instantly familiar with the module. The abstraction build by interfaces, classes and sub-modules allow working with the Bullet Collision module without knowledge of its internals and expand acquaintance with the implementation as needed.

The most low-level module that offers physics simulation is the Bullet Dynamics module, which defines fundamental concepts, like rigid bodies, constraints and solvers for these constraints. The goal of the Bullet Dynamics module is to provide virtual environment for rigid bodies, in which they can move,

according to Newton's laws of motion, collide with each other and react to these collisions, while maintaining invariant shape. The figure 3.3 presents the most fundamental elements of the Bullet Dynamics module.

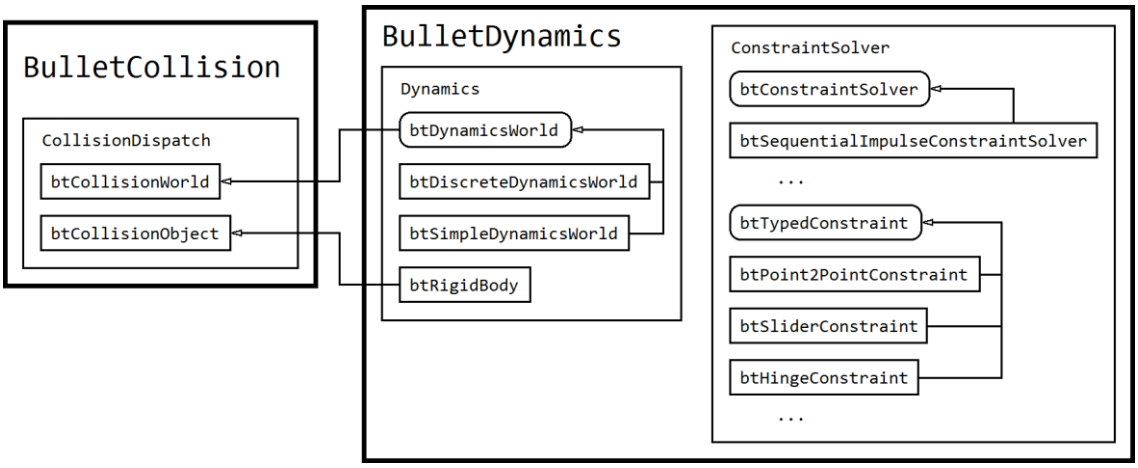
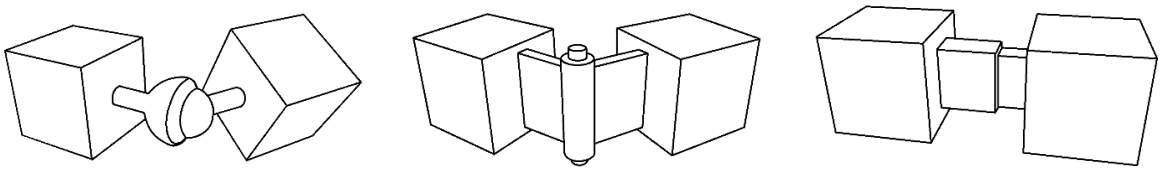


Figure 3.3. The architecture of the Bullet Dynamics module, including sub-modules

- The Dynamics module provides classes and interfaces allowing interaction with rigid body simulation.
  - The btRigidBody is a class providing abstraction for an object, which shape is invariant, even after collision with another object. The btRigidBody class inherits part of its interface form the btCollisionObject class and implementation of that part of the interface. This could imply poor design of the btRigidBody class, but it is not. The most important reason for that is the correctness of thinking that a rigid body is a kind of collision object. Secondly, the btRigidBody class does not overrides, partially or entirely, the implementation of the btCollisionObject, but extends it, by building additional functionality and reusing the implementation of the btCollisionObject. The inherited and additional functionalities of the btRigidBody are various methods providing access to necessary properties of a rigid body, like applying forces and torque to an object. Because the btRigidBody is a class representing an object in the physics simulation, it holds various data members, which correspond to mentioned earlier properties of a physics object, which include object's mass, angular and linear velocities and active forces.
  - The btDynamicsWorld is an interface providing abstraction for an object representing virtual world with simulation of physics, containing rigid bodies, which move according to Newton's laws of motion. Obviously, to be consistent with mentioned laws of motion, detecting collisions is required, thus the btDynnamicsWorld inherits collision detection capabilities from the btCollisionWorld. Apart from collision detection, the btDynamicsWorld provides proper physics simulation, thus defines an interface, which allows interaction with physics simulation. Notably, the btDynamicsWorld interface supports manipulating the collection of btRigidBodies present in the world and applying the btTypedConstraints to particular objects. Additionally the btDynamicsWorld manages which type of the btConstraintSolver is used to resolve constraints. The btDiscreteDynamicWorld is default implementation of the btDynamicWorld interface, therefore inherits the btCollisionWorld's interface and implementation, including data members. The btSimpleDynamicsWorld is another implementation of the btCollisionWorld, used exclusively for testing and debugging purposes. Similar to the btCollisionWorld, the

btDiscreteDynamicsWorld can be understood as a wrapper of all entities in the Bullet Collision and Dynamics modules, representing whole physics simulation system, performing essential calculations of virtual world simulation including modeling physics.

- The Constraint Solver module defines classes actually performing calculations simulating physics and resolving constraints.
  - The btTypedConstraint is a data type representing some constraint of object's motion, which has to be respected by a pair of btRigidBodies. In the figure 3.4, there are presented three constraints: the btPoint2PointConstarint, the btHingeConstraint and the btSliderConstraint.



Point to point constraint | Hinge constraint | Slider constraint

Figure 3.4. Examples of motion constraint available in the Bullet library; figure based on figures 3, 4 and 5 from chapter 7 in [2]

Constraints assure proper motion of bodies, which have to maintain specific relative positions. Example of such objects could be character's limbs connected to torso or a door attached to its frame.

  - The btConstraintSolver is very simple interface providing notion of an object solving some constraint. The most straightforward implementation of that interface is the btSequentialImpulseConstraintSolver utilizing the Gauss-Seidel method, which allows solving a square system of linear equations. However, the most important responsibility of the btSequentialImpulseConstraintSolver is to manage contact constraints. These constraints are generated, by the btSequentialImpulseConstraintSolver, for each contact point occurred in the collision as object's response to a collision. Later all those constraints are solved by the btSequentialImpulseConstraintSolver, allowing application to maintain illusion of object's solidity. Apart from managing contact constraints, the btSequentialImpulseConstraintSolver manages friction and rolling-friction constraints.

It is worth mentioning that the Bullet Dynamics module also supports multi-bodies, which organize multiple bodies in a hierarchy and connect them by joints. Support for multi-bodies are organized in form of separate sub-module, called the Featherstone module. Origin of the name comes from the most fundamental algorithm, which calculates forces acting on a structure of links and joints in a multi-body, called Featherstone's algorithm. The most notable example of a multi-body is a model of a human body, in which limbs, torso and head are connected through joints, represented as constraints.

The most high-level module is the Bullet Soft Body, which provides simulation of soft bodies. Usually soft bodies are utilized for simulation of objects like hair and cloth of a character, ropes present in the environment, blades of a grass, plant stems and various bodies made out of elastic materials. As expected, the Bullet Soft Body module continues to support concepts introduced by the Bullet Collision and Dynamics modules, extending their capabilities of soft body simulation. The architecture of the Bullet Soft Body module is presented in the figure 3.5.

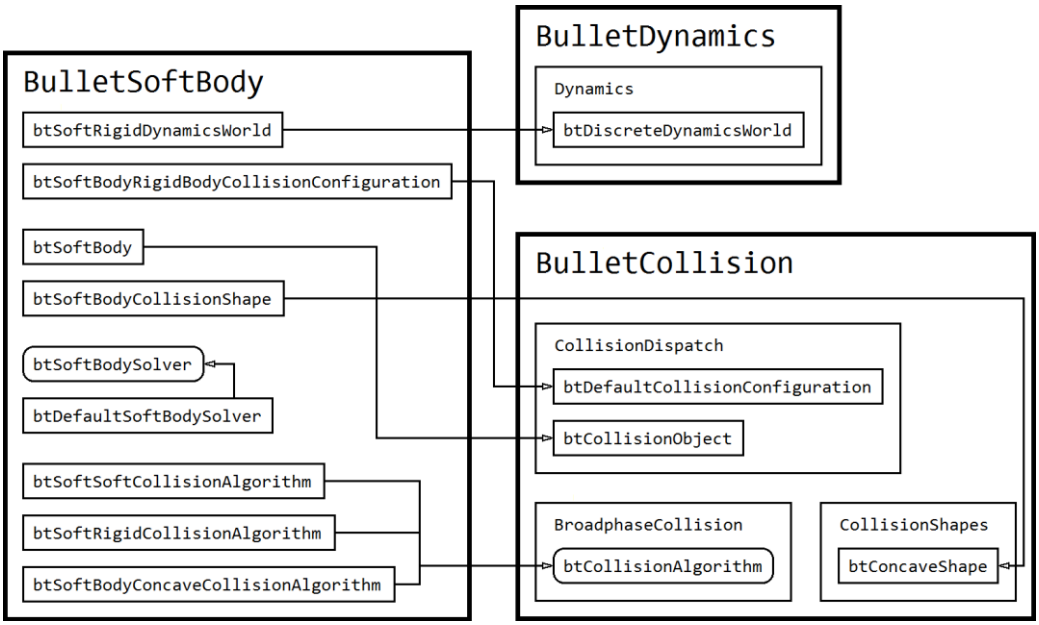
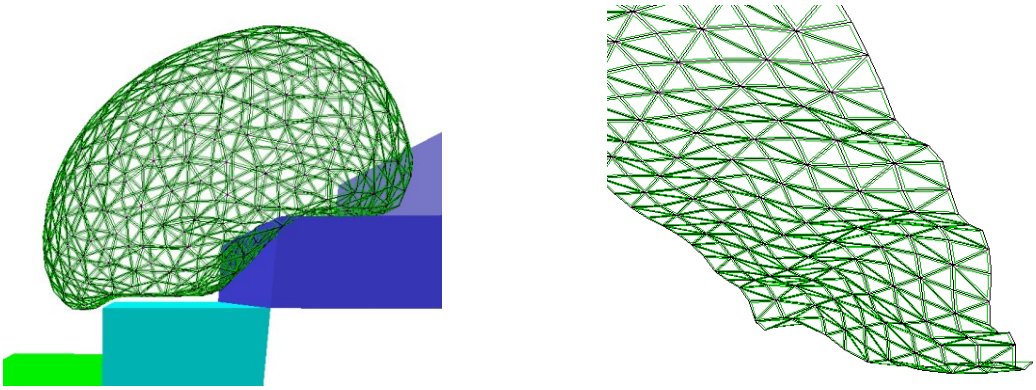


Figure 3.5. The architecture of the Bullet Soft Body module

- The btSoftBody is class representing collision object, which shape can change depending on environmental factors, for example, forces applied to structure of an object. In most basic principle, a soft body is built as a multi-body in which component parts have some degree of freedom, but unlike multi-bodies, soft bodies can have arbitrarily complex internal structure of links and nodes instead of only hierarchical structure present in multi-bodies. That is the reason why soft bodies have to store internally coordinates of each component instead of single btWorldTransofrmation. Obviously, coordinates of soft body components can have much lower precision, because only relative changes to origin, specific for particular soft body, have to be stored. Soft bodies can be divided into two distinct groups: volumetric soft bodies and flat soft bodies. The figure 3.6 presents example of volumetric and flat soft bodies form demos supplied with the Bullet library.



Volumetric soft body Flat soft body

Figure 3.6. Examples of volumetric and flat soft bodies from demos available with Bullet's source code

- The btSoftRigidDynamicsWorld represents physics simulation, with collision detection, of virtual world containing objects with rigid and soft bodies. Because the btSoftRigidDynamicsWorld inherits physics simulation from the btDiscreteDynamicsWorld and collision detection from the btCollisionWorld, support for btSoftBodies has to be added in such a way that soft bodies interact properly with rigid bodies. To ensure that, in addition to the btSoftSoftCollisionAlgorithm providing

collision detection between two soft bodies, the btSoftRigidCollisionAlgorithm was added to the Bullet Soft Body module. There is also available collision algorithm for soft body and triangle meshes called the btSoftBodyConcaveCollisionAlgorithm.

Analyzing architecture of a software allows understanding its structure. It provides knowledge of project decomposition into distinct modules, providing their functionalities. Additionally, it allows grasping dependencies between modules and their further decomposition into classes. Lastly, analyzing architecture allows understanding how classes are connected to each other via inheritance and aggregation relations. However, to fully understand how software works, it is essential to know how objects interact and cooperate with each other. In order to keep this description brief, only two scenarios will be covered: adding new rigid body to dynamics world and performing one full step of discrete physics simulation.

One of the most elementary functionalities in the Bullet library is an ability to insert the btRigidObject into the btDiscreteDynamicsWorld. The method providing implementing insertion operation is the addRigidBody(), which schema is presented in the figure 3.7.

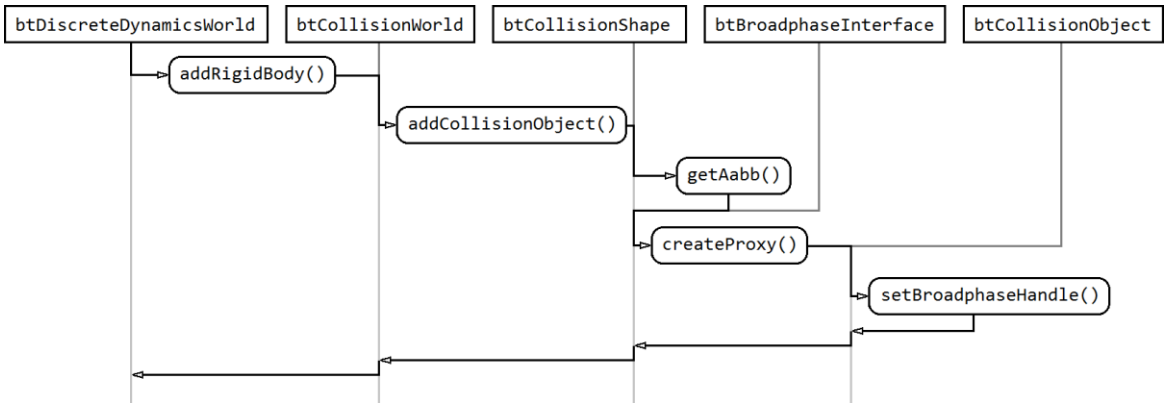


Figure 3.7. Cooperation schema of the addRigidBody() method

The addRigidBody() is very good example of object's cooperation and division of functionalities between modules. In the first step of this method, the rigid body is added to the collection of all rigid bodies present in the dynamic world. Then addCollisionObject() is called, which adds collision object to collision world. Analyzing deeper, the addCollisionObject() inserts collision object into collection of all collision objects present in the world and creates proxy, from AABB calculated by an implementation of the btCollisionShape interface. Creating proxy for the collision object is handled by an implementation of the btBroadphaseInterface. Lastly, handle to created proxy is saved in the collision object. Note that each task is divided into distinct operations, which are handled by objects, which have access to all necessary data to perform particular task. This makes not only code faster by minimizing indirections and data transfers, but also cleaner and more readable. For example when proxy for a broad phase is created, the collision shape calculates the AABB for it because, it stores all geometry data associated with the object.

Arguably, the most interesting method in the Bullet library is the stepSimulation() from the btDiscreteDynamicsWorld class, which performs discrete physics simulation. This method implements physics engine pipeline presented in the figure 1.4 from the introduction chapter. The cooperation schema of the stepSimulation() is presented in the figure 3.8. The stepSimulation() is quite complex method, thus in order to preserve clarity, sub-routine performing collision detection will be presented on a separate schema.

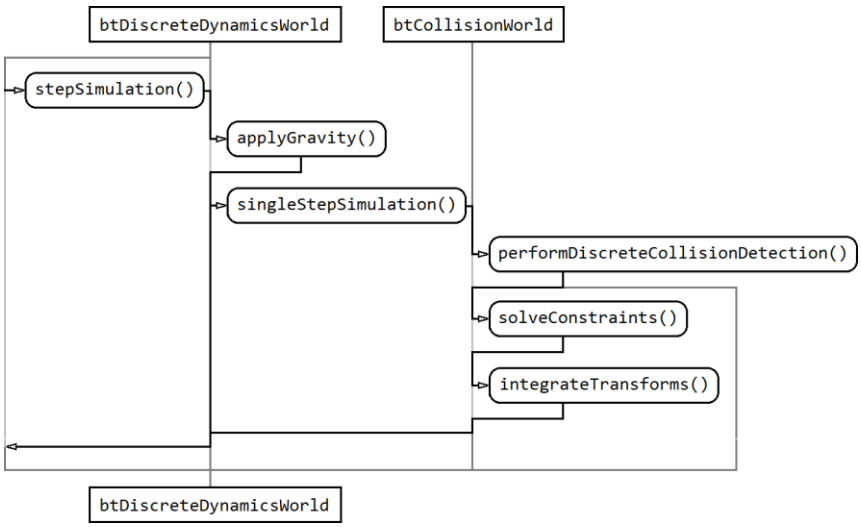


Figure 3.8. The schema presenting calling tree of the addRigidBody() method

The first procedure called in a physic simulation applies force of gravity onto bodies present in the world. Then, the stepSimulation() executes single step of a simulation in a loop. This is the reason why this simulation can be described as discrete, because it performs simulation in a step manner. The number of iterations in the mentioned loop depends on the duration of the time, which needs to be simulated, and the duration time of a single step. Analyzing deeper, the single step of a simulation consists of detecting collisions, solving constraints and integrating transformations. Again, the collision detection procedure will be described separately. Obviously solving constraints not only includes constraints imposed by the application programmer, but also constraints generated automatically by the constraint solver, including contact constraints maintaining solidity of objects and friction constraints improving accuracy of a simulation. The last operation finalizes simulation by integrating acceleration and then velocity of each object to calculate its world transformation. Returning to the collision detection procedure, named in the Bullet library performDiscreteCollisionDetection(), it is important to mention that algorithm used to detect collisions consists of three steps. Firstly, all objects' AABBs are updated to reflect current objects' transformations. Then an implementation of the btBroadphaseInterface computes all potentially overlapping pairs of objects based on their AABBs. Lastly, the btCollisionDispatcher iterates through pairs of objects returned by the broad phase performing narrow phase of collision detection. The figure 3.9 presents schema visualizing cooperation between objects in the collision detection procedure.

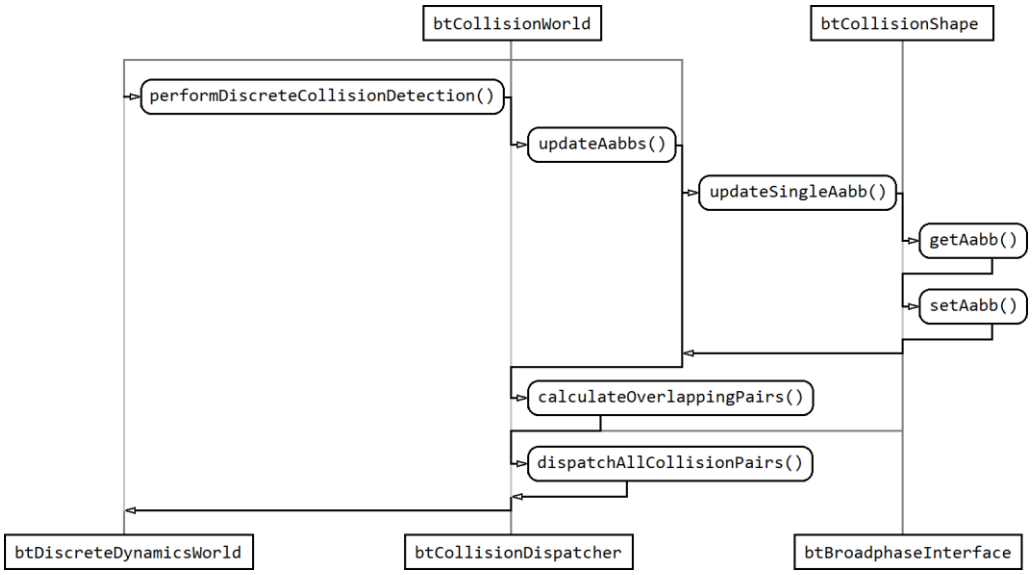


Figure 3.9. The cooperation schema of the performDiscreteCollisionDetection() procedure

Updating object's AABB is reduced to getting AABB form object's collision shape, regarding its current world transformation, and notifying implementation of the btBroadphaseInterface about updating object's AABB. The narrow phase is executed as the dispatchAllCollisionPairs() procedure, which schema is presented in the figure 3.10.

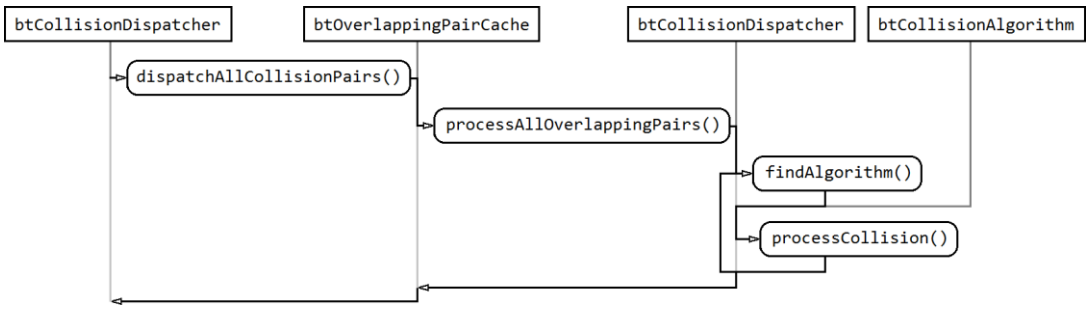


Figure 3.10. The cooperation schema of the dispatchAllCollisionPairs() procedure

The execution of the narrow phase is implemented as for each loop on the btOverlappingPairCache. In each iteration of the loop, proper btCollisionAlgorithm is chosen by the btCollisionDispatcher, according to collision matrix. Then chosen collision algorithm is called producing the btPersistentManifold, through which contact points of colliding objects can be obtained.

In retrospect, it is clear that the Bullet library is well designed piece of software. Its modular architecture allows adapting this library into much larger projects and explicit implementation with adequate amount of abstraction allows wide variety of developers to use library without specialist knowledge.

3.2. Octree broad phase in the Bullet library

Understanding the Bullet library internals and theoretical analysis of the octree broad phase is finally consolidated in the implementation of the btBroadphaseInterface, which utilizes an octree data structure. The schema visualizing architecture of the octree implementation is presented in the figure 3.11.

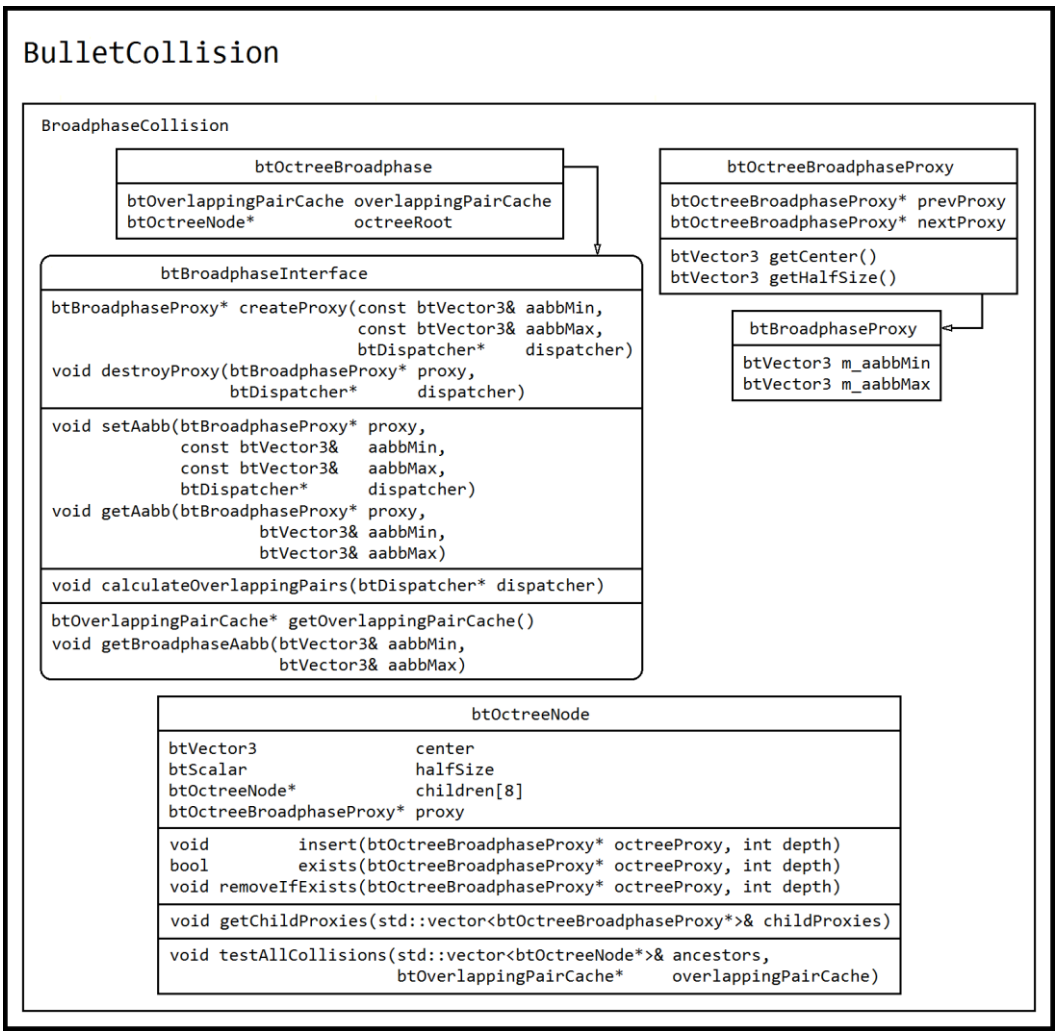


Figure 3.11. The architecture of the octree broad phase implementation

The implementation of an octree broad phase consists of three classes. The btOctreeBroadphase manages execution of a broad phase by implementing interface inherited from the btBroadphaseInterface. The btOctreeNode is a data structure, which represents node of an octree and implements operations, which can be executed on a tree data structure, like inserting and removing nodes. The btOctreeBroadphaseProxy represents object through which collision objects are visible to broad phase. By inheriting data members form the btBroadphaseProxy class, the octree proxy can be understood as AABB of a collision object. At the same time, the btOctreeBroadphaseProxy represents node in a list of objects assigned to particular node of an octree.

The facade behind which whole functionality of a broad phase is hidden is defined by the btBroadphaseInterface. The most important methods of that facade can be seen in the figure 3.11.

- The method createProxy(), which can be seen in the figure 3.7, is responsible for creating broad phase proxy for a collision object. In the btOctreeBroadphase implementation, it creates new instance of the btOctreeBroadphaseProxy class and returns its address through pointer data type. It is important to understand that the createProxy() method is called once in a lifetime of a btCollisionObject.
- The setAabb() method updates AABB of a collision object, stored in the broad phase proxy belonging to that object. The first parameter of the setAabb() method is a broad phase proxy, which needs to be updated. The second and third parameters define updated AABB, regarding current world transformation of a collision object. Unlike the createProxy(), the setAabb() can be called multiple times, while a btCollisionObject is alive. The reason for that is the setAabb() is called once for each object in each step of a simulation, which can be seen in figures 3.8 and 3.9. The octree broad phase implements the setAabb() method in three steps. Firstly, it removes broad phase proxy form the octree, if it is already inserted. The second step overwrites old AABB with the current one. Lastly, updated broad phase proxy is inserted into the octree data structure. Removing and inserting operations are necessary, and cannot be replaced with just updating AABB, because updating object's AABB can cause change position of the broad phase proxy in the octree.
- The destroyProxy() method signalizes that a collision object with particular proxy has been removed from a collision world. The implementation of the destroyProxy(), not only calls destructor of the btOctreeBroadphaseProxy and frees allocated memory for it, but also removes that proxy from octree data structure. The destroyProxy() should free all resources acquired in the createProxy().
- The getAabb() is very simple getter, which extracts and returns AABB stored in the btBroadphaseProxy.
- The getOverlappingPairCache() is another getter, which returns collection of pairs of overlapping objects. This method should be understood as a way of returning results of calculations performed in the broad phase by the btBroadphaseInterface.
- The getBroadphaseAabb() is last getter in the btBroadphaseInterface, which simply returns AABB, which encapsulates all collision objects present in a collision world. In the octree broad phase implementation, returned AABB is a box associated with the root node of an octree.
- The calculateOverlappingPairs() is a method, which fills the btOverlappingPairCache with pairs of objects, which AABB intersect. As stated earlier, in the chapter 2.2.3, in order to find all collisions in an octree procedure from listing 2.14 has to be called with root of the octree as an input node and with empty ancestors and collision collections. This is exactly what body of the calculateOverlappingPairs() contains. Firstly, the overlapping pair cache is emptied, because it is a collision collection. Secondly, a collection for ancestors is created with height of an octree elements reserved. Lastly, the testAllCollisions() method from the btOctreeNode class is called.

The facade of the btOctreeBroadphase is implemented plainly and explicitly thanks to availability of operations implemented by the btOctreeNode class. All these operations are presented in the bottom of the figure 3.11.

- The insert() operation, as the same suggests, inserts particular octree proxy into the list associated with proper octree node. Its implementation is nearly identical to procedure Insert() from the listing

2.13. The only difference is additional depth limit preventing octree proxies to descend deeper into the octree, above that limit.

- The exists() and the removeIfExists() operations are very similar to insert() operation, because both of them uses the same algorithm defining descending path for a proxy. This descending path is used to check if particular proxy exists in an octree and to remove particular proxy from an octree if it is already inserted, respectively.
- The getChildProxies() is very simple method, which returns collection of all proxies assigned to nodes, which form sub-tree of a particular node. The implementation simply adds proxies assigned to immediate child nodes to shared collection, passed in a method parameter, and calls itself recursively for each immediate child node.
- The testAllCollisions() detects all collisions between objects assigned to the node on which operation is performed and objects assigned to all descendants of that node. Implementation of that method can be seen in the listing 2.14.

Design decisions, which were made regarding properties of an octree implementation, described in the chapter 2.2.3, are explicitly expressed in operations of the btOctreeNode class. All these properties are the same as in the case of example implementation from the chapter 2.2.3. Assignment rule inserts object into smallest possible node, which box encapsulates AABB of an object. The octree data structure is represented using linked nodes scattered across block of memory. Building an octree is performed dynamically, as needed, until object cannot descend lower due to straddling partitioning plane, too large AABB, or exceeding predefined height limit of an octree. Besides the implementation of described operations on an octree, the btOctreeNode class also defines data required by each node of an octree. The structure of that node can be seen in the listing 2.12 and in the figure 3.11.

The last building block of the broad phase implementation utilizing octree data structure is the btOctreeBroadphaseProxy class. As mentioned earlier, this data type represents an object based on which broad phase algorithm checks whether collision objects are intersecting or not. In the Bullet library underlying geometry of a proxy is an AABB, thus the btOctreeBroadphaseProxy inherits data members form the btBroadphaseProxy class, which represent minimal and maximal point of the AABB. Because one btOctreeNode can encapsulate many btBroadphaseProxies, each octree node has to be capable of storing collection of proxies. In order to achieve that, the btBroadphaseProxy is used as an underlying data type, which represents an element of a list and the btOctreeNode stores head of that list. In the btBroadphaseProxy implementation, a double linked list is used, which can be recognized by links to next and previous elements of a list presented in the top-right corner of the figure 3.11.

That completes description of the octree broad phase implementation, which makes impressions that this implementation is quite simple. That impression is not false, because besides code of core algorithms presented in the second chapter, the remaining code just organizes abstraction and interfaces. Additionally the whole implementation takes under one thousand lines of code. Lastly incorporating an octree broad phase into a collision detection system is as simple as passing an instance of the btOctreeBroadphase class to the constructor of the btDiscreteDynamicsWorld class. These facts suggest that the octree implementation makes quite complex problem very easy to understand.

4. EMPIRICAL TESTS EVALUATING BROAD PHASE ALGORITHMS

As stated before, evaluating broad phase algorithm for collision detection by theoretical analysis is not feasible. Therefore, benchmarking tool performing empirical tests has to be utilized. Unfortunately, there is no tool available, which provides customizable physics simulation with configurable broad phase algorithm and allows measuring various metrics concerning performance and effectiveness of a collision detection system. However, as explained in the previous chapter, the Bullet library provides API, which allows client applications to create physics simulation. Thanks to the Bullet library, a benchmarking tool, which would execute tests evaluating behavior of a broad phase algorithm in some predefined scenarios, can be created.

4.1. Benchmarking tool and supervising script

The benchmarking tool, created for the purpose of empirical tests, evaluating broad phase algorithms has to be supervised by a script managing tests execution. The reason for that is multiple factors can affect performance of collision detection algorithms, thus empirical evaluation of broad phase algorithms has to consider these factors. That implies that the supervisor script performs multiple passes in which configuration of a simulation feeds the benchmarking tool, which collects data based on which charts visualizing results are generated. The flow of a data in a single pass performed by the supervising script can be seen in the figure 4.1.

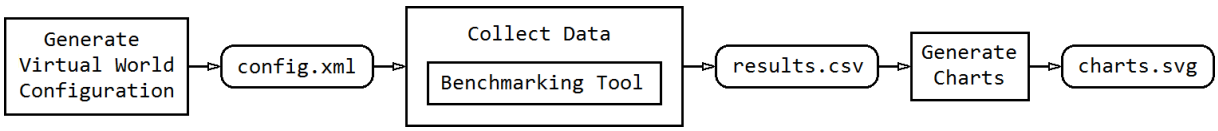


Figure 4.1. The flow chart of single empirical test evaluating broad phase algorithm

In the first step, the supervising script generates XML configuration file called config.xml, which consists of properties defining a virtual world. The idea behind configuring a virtual world is to investigate how various factors affect collision detection. Below there is a list of all properties which can be configured.

- The broad phase algorithm used in the collision detection. There are four broad phase algorithms available, including three algorithms representing each group described in the second chapter and one algorithm representing naive approach. Full list of broad phase algorithms consists of,
  - the brute force broad phase algorithm (the btSimpleBroadphase);
  - the broad phase utilizing dynamic bounding volume hierarchy (the btDvbtBroadphase);
  - the sweep-and-prune broad phase algorithm (the btAxisSweep3);
  - the octree based broad phase algorithm (the btOctreeBroadphase).

  Availability of this option enables comparing broad phase algorithms against each other.
- The number of collision objects. Thanks to this parameter, computational complexity of a broad phase algorithm can be approximated. Additionally it allows minimizing measurement error by supplying more work to an algorithm, thus lengthening computation time.
- The world density, that is expected number of objects per one unit volume cube. The world density is very important factor, because it strongly affects frequency with which objects collide. Thanks to this parameter, the average number of collisions can be kept relatively steady through a whole experiment.

- The velocity of collision objects. Because moving objects disrupt coherence of a scene, this parameter can be used to investigate algorithm susceptibility to the lack of coherence.
- The fraction of stationary objects. The idea behind this parameter is to simulate presence of non-moving objects, like buildings and trees present in real world scenes, and verify whether an algorithm is able to take advantage of that type of scene.
- Types of collision shapes. This parameter provides method of affecting complexity of a narrow phase, thanks to which time in the narrow phase is lengthen. This allows investigating the effectiveness of a broad phase algorithm, which is number of false positive collisions reported, by emphasizing narrow phase's contribution to overall collision detection process. The available collision shapes are boxes, spheres, cones, cylinders and capsules.
- The relative size of objects. As stated in the second chapter, the extreme sizes of objects can be difficult to handle for some types of broad phase algorithms, for example methods based on a uniform grid. On the other hand, some algorithms are designed to handle objects with disproportional sizes. It is worth investigating whether broad phase performance is affected by uniformity of objects' sizes.
- The terrain present in a world. Most of 3D video games take place in an open world with ground represented as triangle mesh. This parameter allows verifying behavior of a broad phase algorithm in scenarios with terrain.

After the configuration file has been generated, the benchmarking tool is lunched. Firstly, the benchmarking tool prepares simulation based on properties from the configuration file. Then, it runs the simulation, while collecting data. Finally, the collected data, from a single pass, is written into CSV file. The schema presenting generation of a results.csv file can be seen in the figure 4.2.

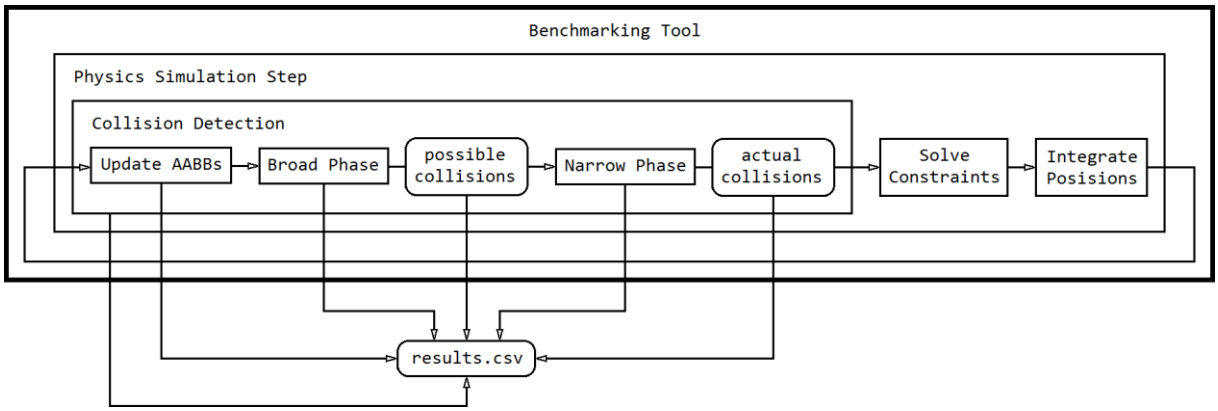


Figure 4.2. Collecting data by the benchmarking tool

The benchmarking tool collects six quantities, which describe performance and effectiveness of collision detection. The first three quantities concern computation time of three stages of a collision detection, shown in the figure 4.2. The forth quantity is computation time of the whole collision detection, which is just collective sum of computation times of each mentioned stage. The last two quantities describe the number of collisions between object's AABBs reported by the broad phase and the number of collisions actually occurred, which is reported by the narrow phase. Each measured quantity corresponds to a value extracted from the Bullet library.

- The value of "Update AABBs" quantity is an execution time of the updateAabbs() method.
- The value of "Broad Phase" quantity is an execution time of the calculateOverlappingPairs() method.

- The value of “Narrow Phase” quantity is an execution time of the dispatchAllCollisionPairs() method.
- The value of “Collision Detection” quantity is a collective sum of execution time of previous quantities, which is an execution time of the performDiscreteCollisionDetection() method.
- The value of “Possible Collisions” quantity is equal to number of intersecting pairs of AABBs in the btOverlappingPairCache after broad phase.
- The value of “Actual Collisions” quantity is equal to the number of the btPersistentMainfolds stored in the btCollisionDispatcher. However, only manifolds having contact points are counted.

In the last step, the supervising script converts collected data into numerous charts and saves them as SVG files. That action finishes current pass and causes supervising script to start new pass with different configuration. The supervising script stops when all relevant configurations have been measured.

Understanding configuration capabilities of the supervising script and quantities collected by the benchmarking tool, performed tests can be described. Each test configures physics simulation with standard set of parameters, besides one parameter, which influence on collision detection is currently investigated. Such a parameter takes in each pass one particular value from predefined range of possible values. Then, values of mentioned quantities are investigated, while value of examined parameter changes through predefined range. The standard set of parameters consists of,

- one thousand collision objects present in the world;
- object sizes are chosen randomly using beta distribution with parameters alpha and beta set to 100;
- collision shapes are chosen randomly with uniform distribution from set of following shapes: box, sphere, cylinder, cone and capsule;
- world density equal to 0.05 objects per unit volume cube;
- objects choose velocities randomly, using uniform distribution with average velocity equal to 40 units per one simulation step, in every tenth step of the simulation;
- no terrain and force of gravity are present in the world.

The values chosen for the default configuration were selected to be representative for typical collision detection scenarios. Obviously, each application is different, thus it is strongly recommended to test collision detection algorithms in scenarios representative for that particular application before making final decision. Lastly, it is important to mention that the benchmarking tool collects data from 400 simulation steps lasting 16 ms and averages recorded values of mentioned quantities. In order to make measurements independent from effects of caches present in modern CPUs, before collecting data benchmarking tool executes 200 “warm up” simulation steps.

4.2. Measurements results and interpretation

One of the most important property of an algorithm is its behavior for large input data. The first test investigates that behavior for available, in the benchmarking tool, broad phase algorithms.

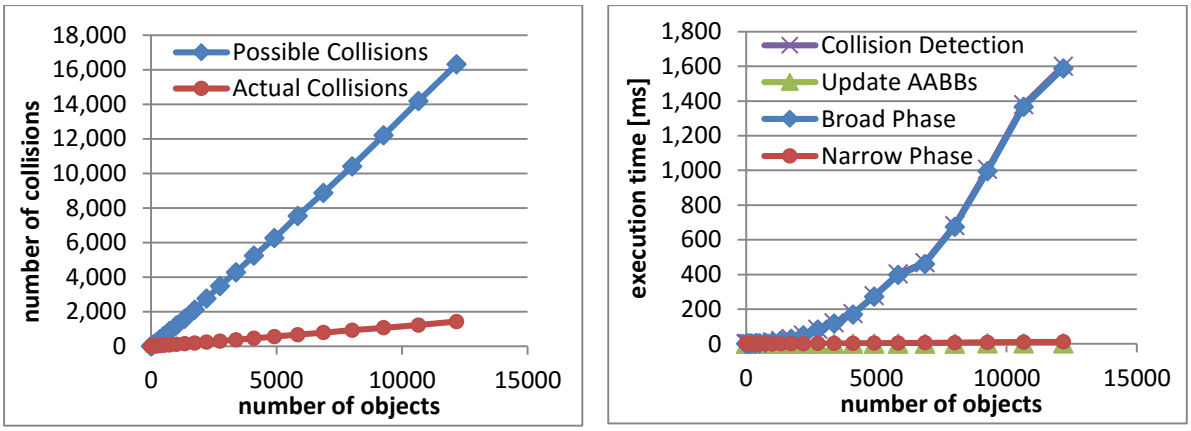


Figure 4.3. Effectiveness and performance of brute force broad phase as function of number of objects

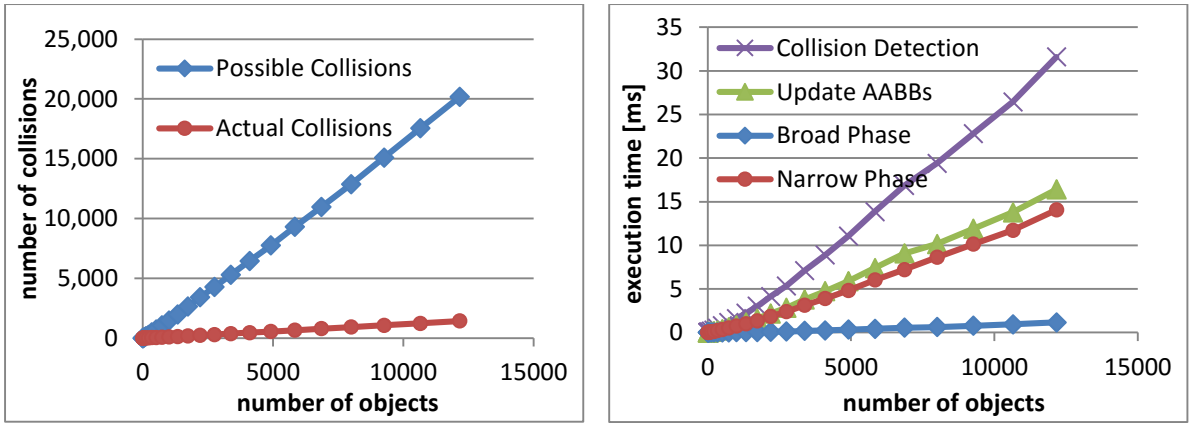


Figure 4.4. Effectiveness and performance of BVH broad phase as function of number of objects

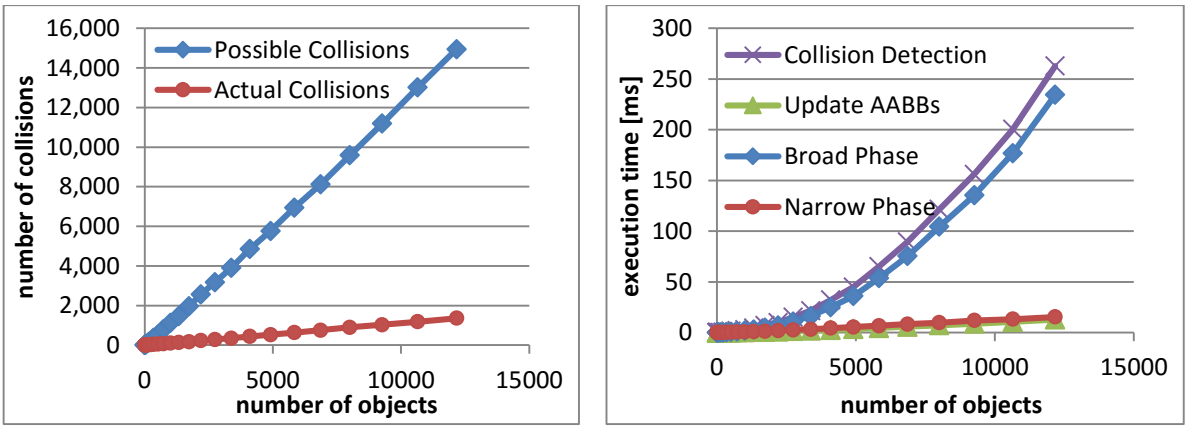


Figure 4.5. Effectiveness and performance of Octree broad phase as function of number of objects

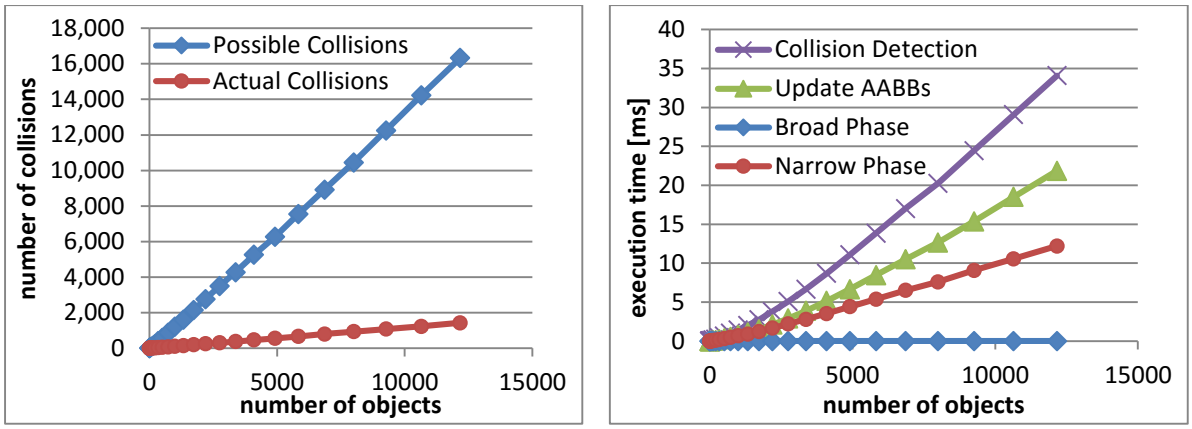


Figure 4.6. Effectiveness and performance of SAP broad phase as function of number of objects

The brute force approach, presented in the figure 4.3, reveals the problem with testing all possible collisions - the number of tests performed in the broad phase is quadratic in terms of number of objects in the world. This problem makes performance of a whole collisions detection system bound by the broad phase, which in the case of a brute force approach is very slow. It is also worth mentioning that utilizing AABBs in the broad phase and collision geometry in the narrow phase is much better than just testing collisions using collision geometry. This can be seen in charts on the left side of figures 4.3 – 4.6, which show that number of collisions reported by the broad phase is linear in terms of number of objects in the world instead of quadratic. The best results were obtained using SAP and BVH broad phases. In both cases, maintaining additional data structures provided huge performance gains, making performance of a whole collision detection system bounded by the narrow phase and updating objects AABBs. More importantly, the time complexity of these methods is between quasi-linear and linear, in contrast to quadratic time complexity of the naive approach. It is also worth mentioning that in case of SAP broad phase Broad Phase execution time is always zero, because performDiscreteCollisionDetection() is an empty procedure. Implementation of the SAP broad phase in the Bullet library maintains algorithm output rather than recalculates it from scratch, thus it is no surprise that all calculations required to detect collisions are performed in the updateAabbs() method. Its code is equal to the code presented in the listing 2.23 with additional maintaining of the collection with all overlapping pairs of AABBs. Despite very similar performance of BVH and SAP broad phases, measurable difference between these algorithms can be observed. The figure 4.7 compares performance of overall collision detection between BVH and SAP broad phases.

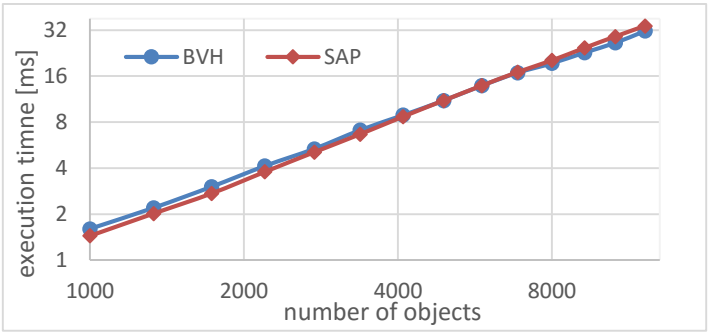


Figure 4.7. Comparison between collision detection with BVH and SAP broad phases

It can be noticed that SAP broad phase performs better, when number of objects do not exceed five thousand objects. When number of objects is larger than that, the BVH broad phase becomes more profitable. The Octree broad phase performed much worse than SAP and BVH broad phases, but also significantly better than brute force algorithm.

An investigation to find the root cause of the Octree broad phase's poor performance started by profiling the benchmarking tool while executing described test. This is standard procedure performed when understanding performance impact of parts of a code is needed. Using this method, parts of a code, which are executed the longest, can be located. The figure 4.8 visualizes collected profiling data from the callgrind tool in form of call map.

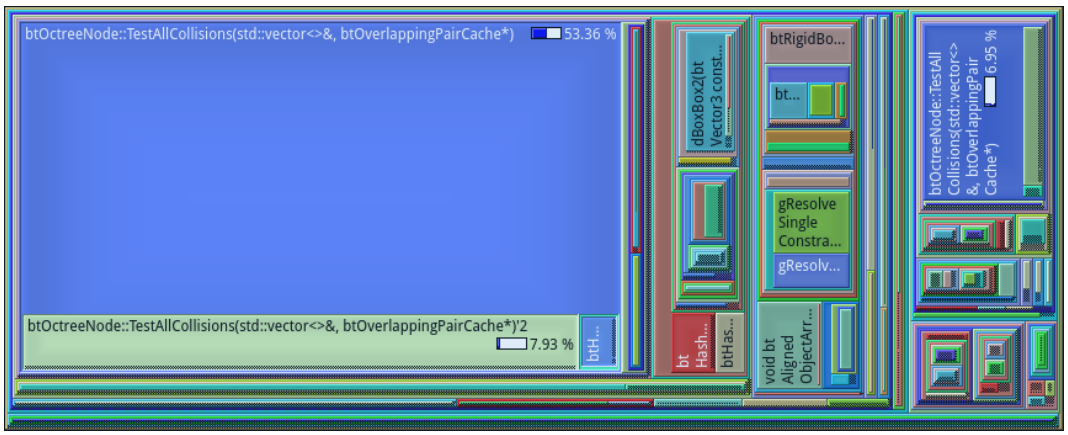


Figure 4.8. Call map of the benchmarking tool utilizing Octree broad phase

It can be easily seen, that in the figure 4.6 there are three large rectangles, which represent calls to the TestAllCollisions() method from the btOctreeNode class. According to data collected by the callgrind tool, roughly 60% of overall time is spend inside that method. As stated in the third chapter, this method reports all pairs of objects, which AABBs are intersecting. The code of that method can be seen in the listing 2.14. Time complexity of the algorithm from the listing 2.14 has been analyzed in the second chapter and states that algorithm has linearithmic computational complexity in terms of number of objects in the world. However, that is the case only when all lists of objects associated with nodes are bounded by a constant value, which means that lists are negligibly short. This assumption was broken in tested scenarios, because lists become significantly long. Obviously, that caused the performance of the Octree broad phase is be poor, which can be observed in the figure 4.5. The curve representing execution time of the Octree broad phase resembles quadratic growth rather than linearithmic growth. The figure 4.9 presents chart revealing average length of lists at particular level of an octree.

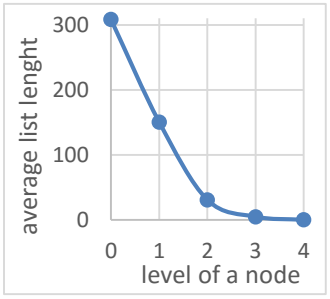


Figure 4.9. Relationship between level of a node and length of list assigned to it

The underlying cause of the large number of objects accumulating in top parts of an octree is the problem of straddling objects, which was described in the second chapter. It can be easily grasped that straddling is a big problem, when one would considered number of separating planes at each level of an octree. At level zero, the number of straddling planes is equal to three – they correspond to planes defined by X, Y and Z axes. Unfortunately, the number of separating planes doubles at each level. This shows the fundamental problem occurring in the data structures used in spatial partitioning based methods. They do not adjust to the collision geometry of objects, making difficult for objects to fit into these data structures. Methods organizing collision geometry spatially, like BVH broad phase, build data structures based on the collision geometry, thus this problem is not existent. Building and maintaining data structures organizing collision geometry spatially is more complex than building and maintaining spatial partitioning methods. However, results of empirical tests in the figure 4.4 shows that building and maintaining BVH is definitely cost effective.

In the large number of applications utilizing physics simulation, objects participating in a collision detection have different sizes. Good example of an application with objects of various objects' sizes is a game in an open world, in which buildings and vehicles are much larger than characters, gun bullets or vehicle shards. In order to test influence of uniformity of objects' sizes on a performance of broad phase algorithms, empirical tests were performed in which size of an object was chosen randomly using beta distribution. The figure 4.10 presents graphs of beta distribution with both alpha and beta parameters equal to the name of the series.

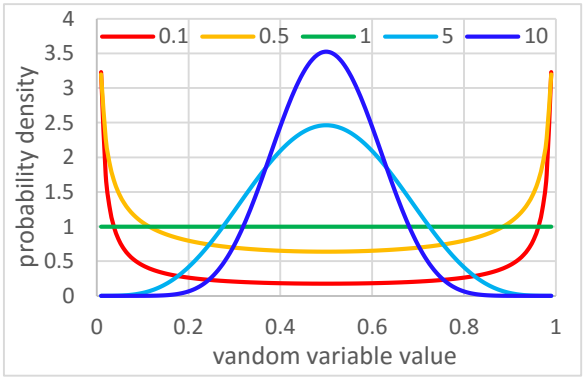


Figure 4.10. Beta distribution with different values of alpha and beta parameters

The beta distribution was chosen, because it allows changing distribution of objects' sizes from extreme to uniform, depending on values of alpha and beta parameters. Setting both parameters to value smaller than one causes objects' sizes to be extreme. By simple assigning value larger than one to both parameters, objects' sizes become uniform. Even uniform distribution can be represented by the beta distribution, when alpha and beta parameters are equal to one. Results of tests measuring performance and effectiveness of a collision detection system for different degrees of uniformity of objects' sizes are shown in figures 4.11 – 4.14.

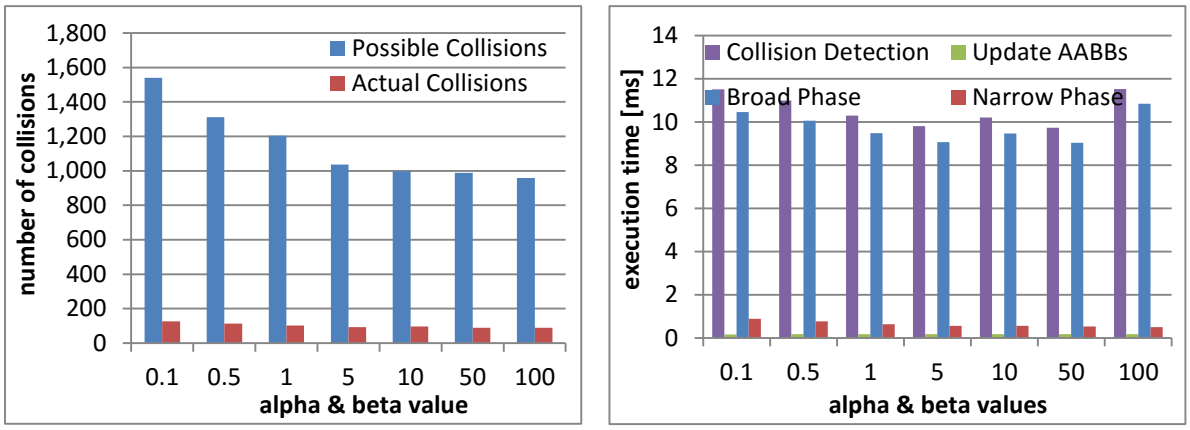


Figure 4.11. Effectiveness and performance of brute force broad phase as a function of uniformity of objects' sizes

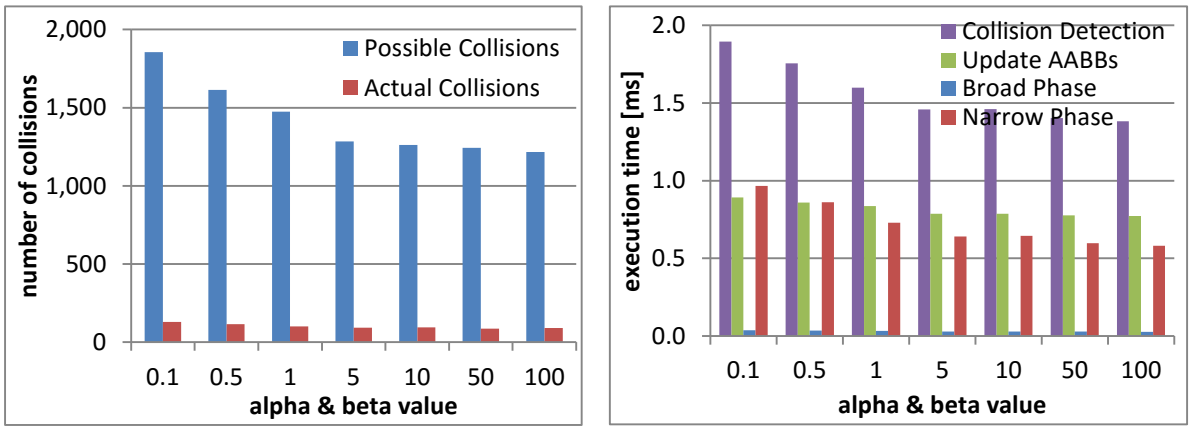


Figure 4.12. Effectiveness and performance of BVH broad phase as a function of uniformity of objects' sizes

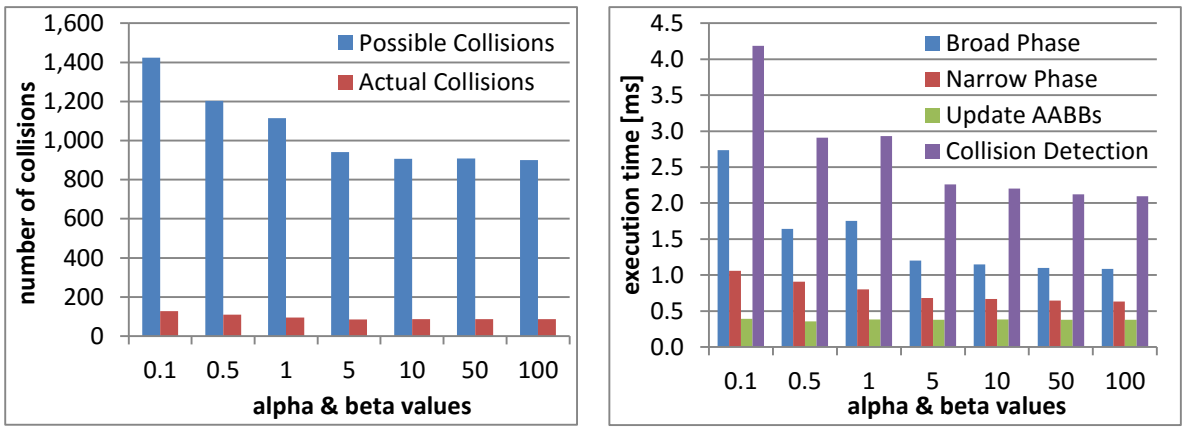


Figure 4.13. Effectiveness and performance of Octree broad phase as a function of uniformity of objects' sizes

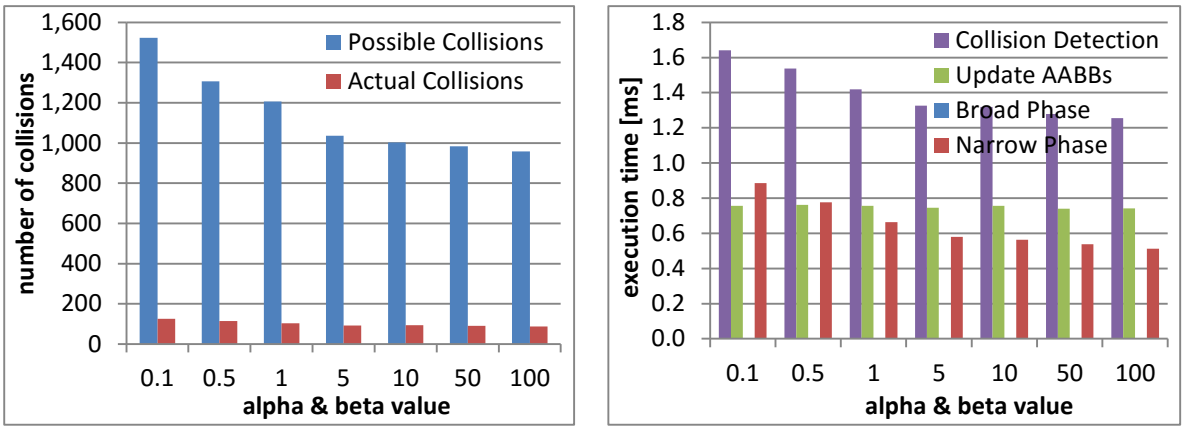


Figure 4.14. Effectiveness and performance of SAP broad phase as a function of uniformity of objects' sizes

As expected, the brute force broad phase is unaffected by a degree of objects' sizes uniformity. Interestingly, the number of possible collisions is noticeably higher when objects' sizes are extreme. The reason for that is the size of an object is chosen randomly and independently for each dimension. Therefore objects can be very thin, which causes decrease in the tightness of their AABBs. The figure 4.15 shows example of AABB encapsulating very thin box.

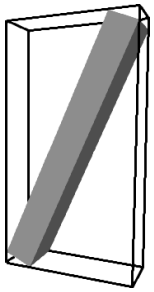


Figure 4.15. AABB has poor tightness when encapsulates rotated thin object

Reduced tightness of AABB caused drop in performance of a narrow phase in each test, which can be observed in the left side of figures 4.11 – 4.14. This conclusion comes from the fact that, the number of possible collisions is the most important factor influencing execution time of a narrow phase. Lastly, the lack of tightness in thin objects caused increase in execution time of the Octree broad phase. Because objects in the Octree broad phase are seen through BVs, the lack of tightness caused objects to be inserted in higher nodes of an octree, negatively affecting performance of procedure detecting collisions. Aside from the Octree broad phase, the BVH and SAP broad phases are not impacted by uniformity of objects' sizes, which match conclusion from theoretical analysis.

Another popular type of scenes in applications involves some percentage of static objects, which have to be considered when detecting collisions, like buildings, trees or rocks. It is worth investigating whether collision detection system can take advantage of this input data's property. Results of executed tests with various fractions of stationary objects are presented in figures 4.16 – 4.19.

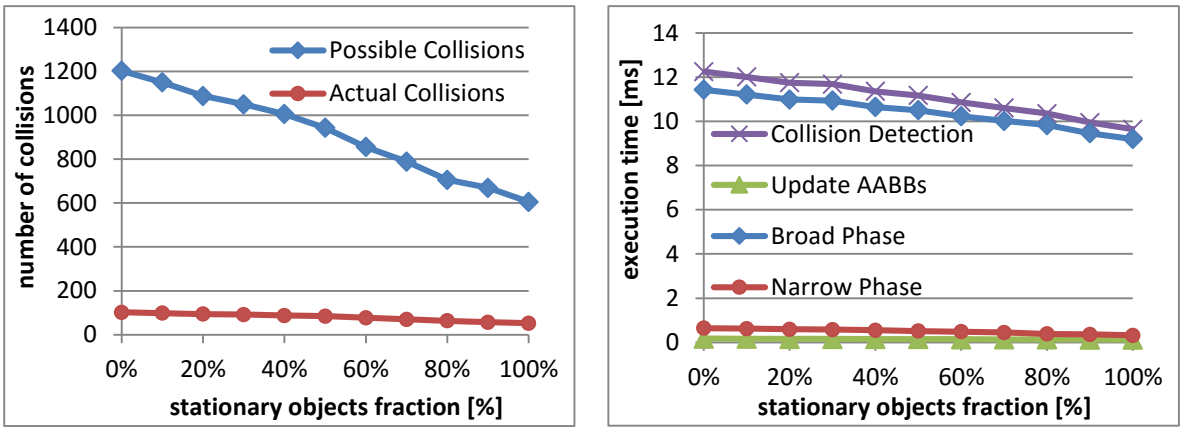


Figure 4.16. Effectiveness and performance of brute force broad phase as a function of stationary objects fraction

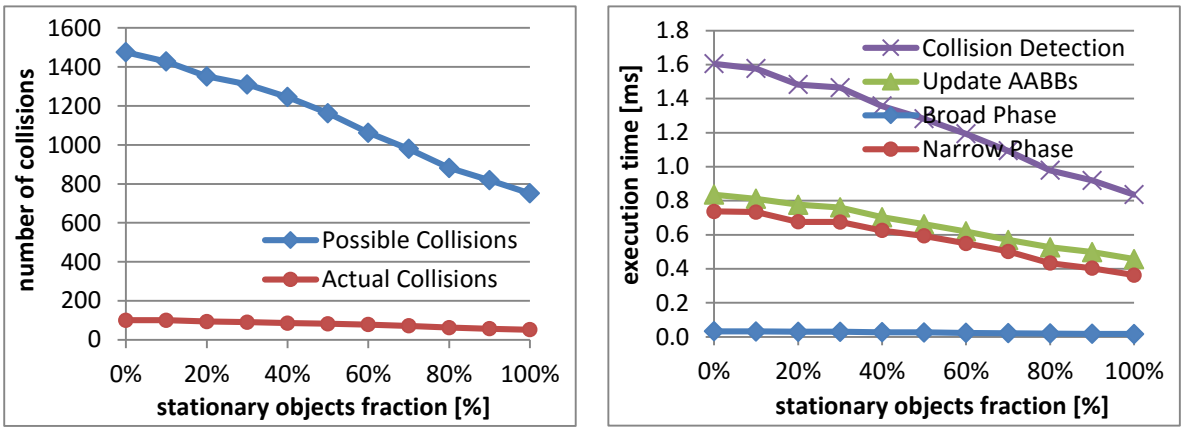


Figure 4.17. Effectiveness and performance of BVH broad phase as a function of stationary objects fraction

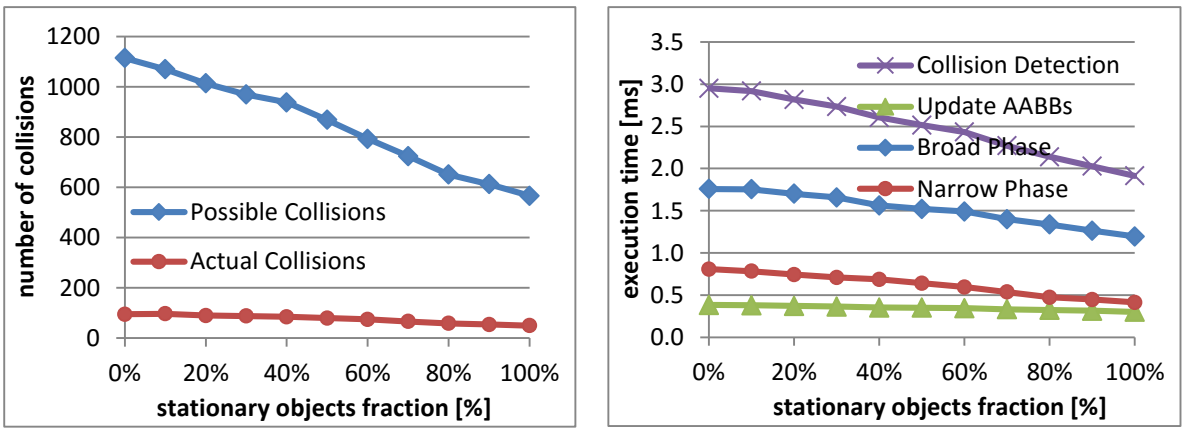


Figure 4.18. Effectiveness and performance of Octree broad phase as a function of stationary objects fraction

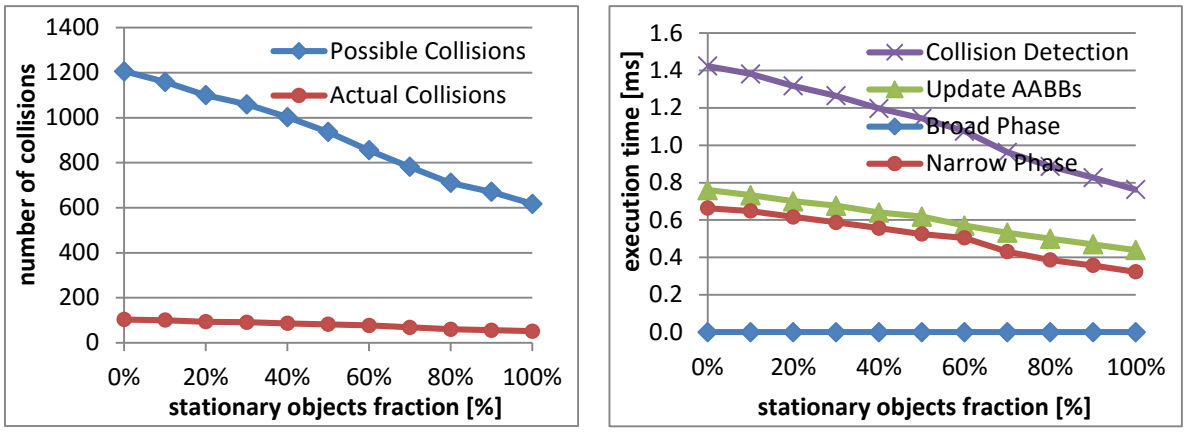


Figure 4.19. Effectiveness and performance of SAP broad phase as a function of stationary objects fraction

As expected, the percentage of stationary objects affects significantly execution time of a collision detection system in all executed cases. It can be noticed in the left side of figures 4.16 – 4.19, that the number of stationary objects is inversely proportional to the number of expected and possible collisions. The reason for that is when there is less and less movement in a scene the probability of collision between objects decreases. Because the number of possible collisions translates directly to the number of tests performed in a narrow phase, the computation time of the narrow phase is also inversely proportional to the number of stationary objects. The amount of work dedicated to updating AABBs also decreases, when more and more objects stop moving, because updating procedure is called less often. Regarding the execution time of a broad phase, it is clearly visible in the right side of figures 4.16 – 4.19 that it is affected by the stationary objects' fraction parameter. However, theoretical analysis of broad phase algorithms suggests otherwise. For instance, brute force broad phase algorithm always performs exhaustive number of tests, thus should not be affected by stationary objects' fraction parameter. The same argument can be made for octree and BVH based broad phase algorithms. The most likely explanation for this behavior is some sort of optimization performed by the library itself, which is not explicitly expressed in the code nor the documentation. Additionally, it is important to mention that the size of the space representing all possible collisions can be reduced by omitting tests between objects, which position did not change from the last update. It is possible to take advantage of this effect, but described broad phase algorithms in the second chapter do not incorporate such an optimization. Investigating effect of the stationary objects' fraction parameter on the broad phase's execution time requires in depth analysis of the Bullet Library, which is outside the scope of master's thesis topic.

For large group of applications, like video games and flight simulators, it is crucial to detect collisions between fast moving objects. In order to investigate influence of object's velocity on broad phase algorithm's performance and effectiveness a series of test with objects moving at different speeds were performed.

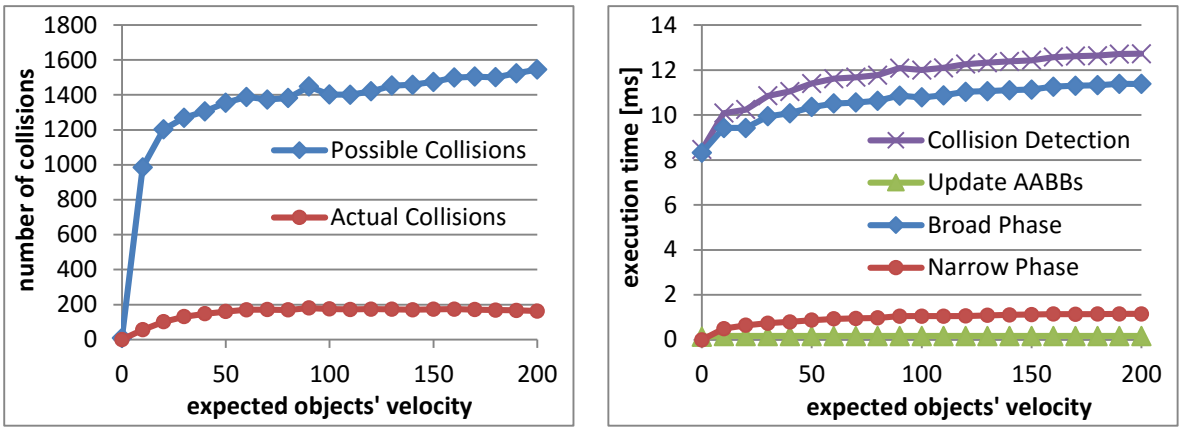


Figure 4.20. Effectiveness and performance of brute force broad phase as a function of expected objects' velocity

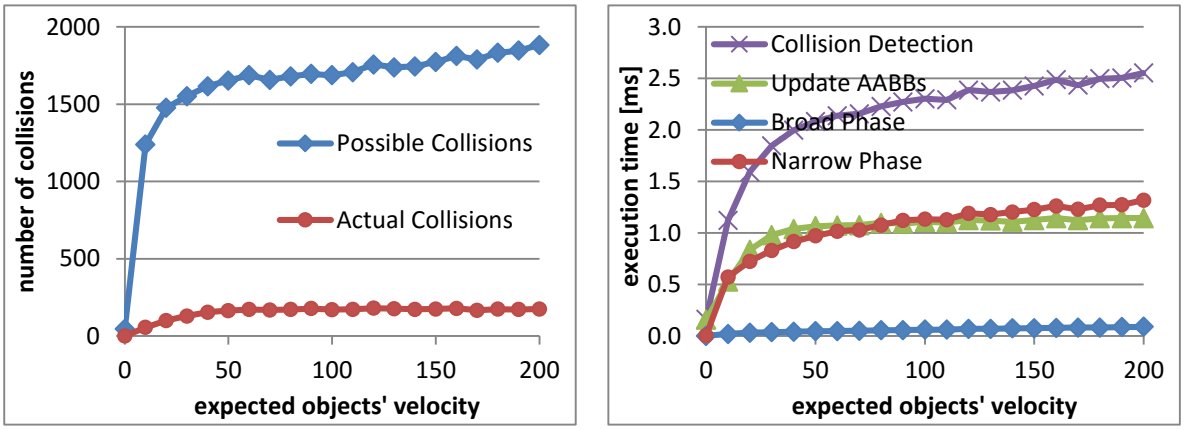


Figure 4.21. Effectiveness and performance of BVH broad phase as a function of expected objects' velocity

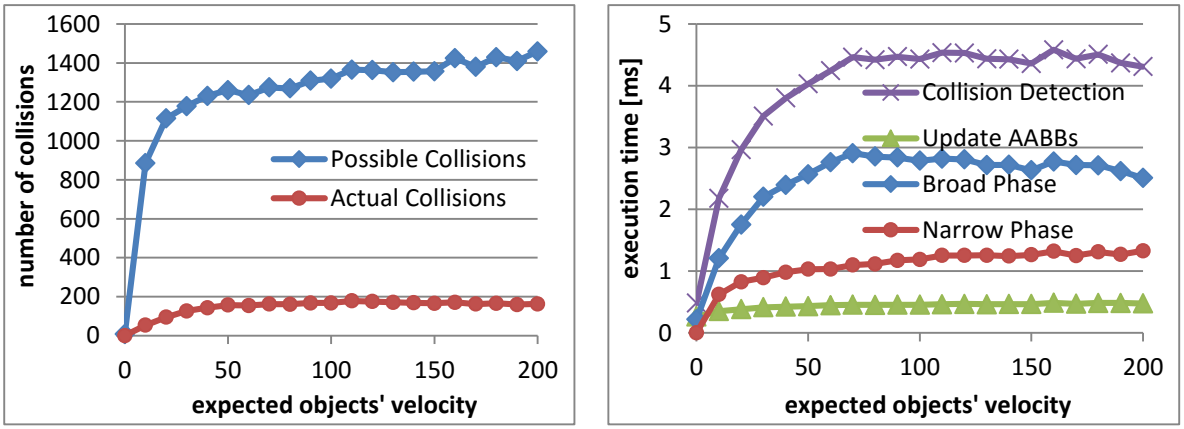


Figure 4.22. Effectiveness and performance of Octree broad phase as a function of expected objects' velocity

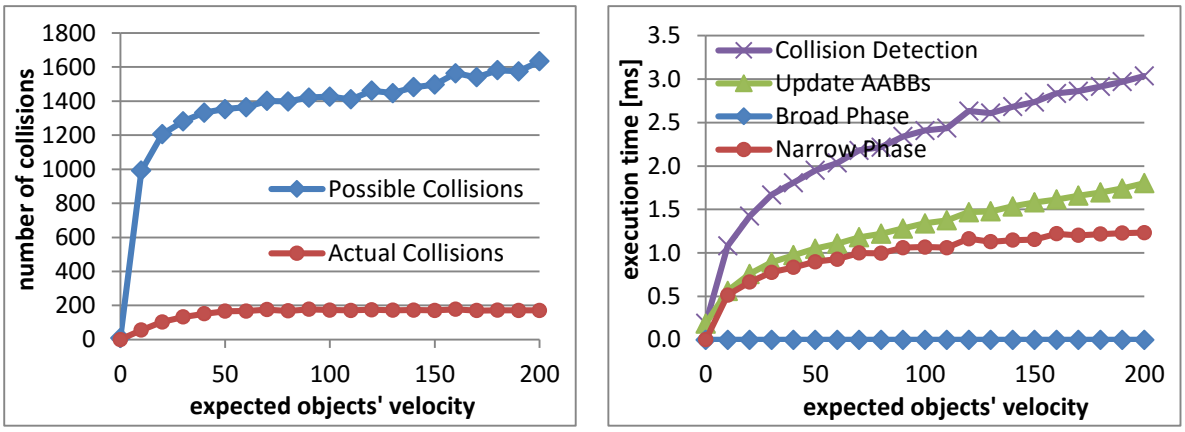


Figure 4.23. Effectiveness and performance of SAP broad phase as a function of expected objects' velocity

Not surprisingly, results presented in figures 4.20 – 4.23 confirm that the faster objects move the more difficult is to detect collisions between them. Because objects move in random directions, increasing expected objects' velocity also increases probability of collision occurring between two objects, which can be observed on the left side of figures 4.20 – 4.23. As before, the number of possible collisions directly translates to performance of the narrow phase. Also recalculating objects' AABBs is required more often, when objects move faster, which explains increase in execution time of Update AABBs phase. However, the most interesting effect appearing in figures 4.20 – 4.23 is the reduction of strength with which increasing the expected objects' velocity parameter affects the performance of a collision detection system. In case of the SAP broad phase, the expected objects velocity parameter has quite strong impact on performance of a collision detection, compared to an octree based broad phase for instance. This matches with theoretical analysis of the SAP broad phase, from which it is known that this method is very susceptible to lack of coherence in a scene. Regarding execution time of broad phase itself, the situation is very similar, when stationary objects fraction parameter was analyzed. The execution time of broad phase in all tests is definitely affected by the expected objects' velocity parameter, which is not expected regarding theoretical analysis of broad phase algorithms in the second chapter. For instance, the execution time of the octree broad phase depends on two factors: length of lists associated with octree's nodes and the height of an octree. It however should not depend on expected objects' velocity. Again, finding optimization causing observed dependency in the Bullet library is outside the scope of this master's thesis.

Scenes rendered using 3D graphics are characterized by various degrees of density, that is the average number of objects per unit sized volume. For instance, an open world game has order of magnitude smaller density compared to a particle simulation. Results of tests examining influence of world's density on performance and effectiveness of broad phase algorithms are presented in figures 4.24 – 4.27.

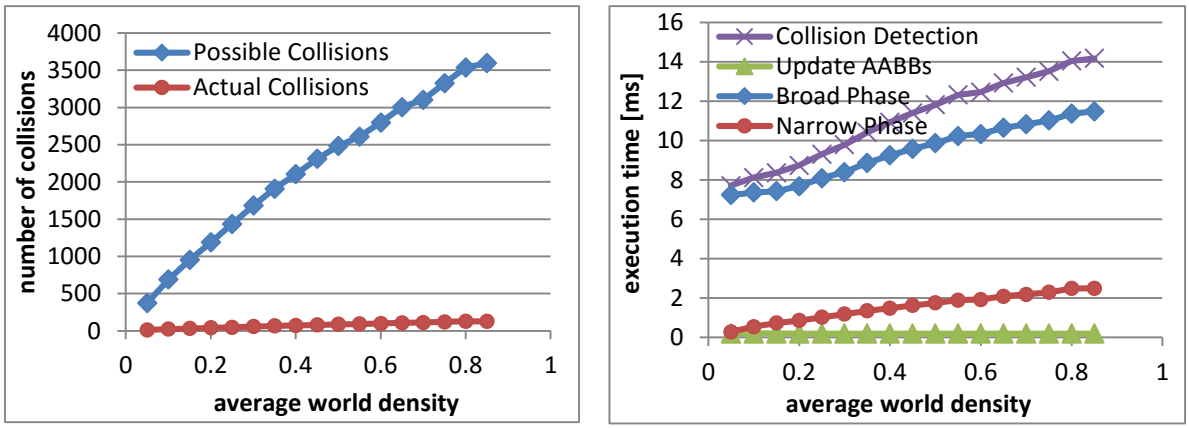


Figure 4.24. Effectiveness and performance of brute force broad phase as a function of average world density

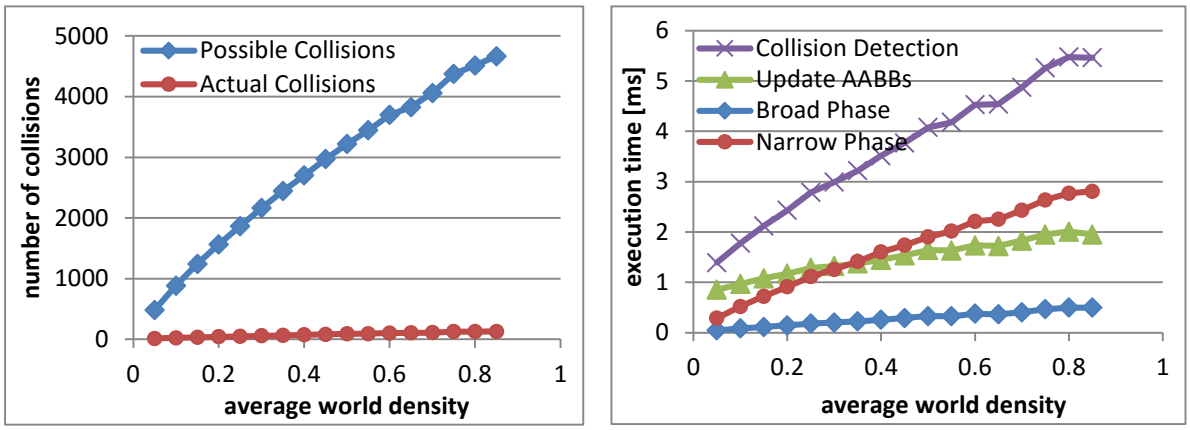


Figure 4.25. Effectiveness and performance of BVH broad phase as a function of average world density

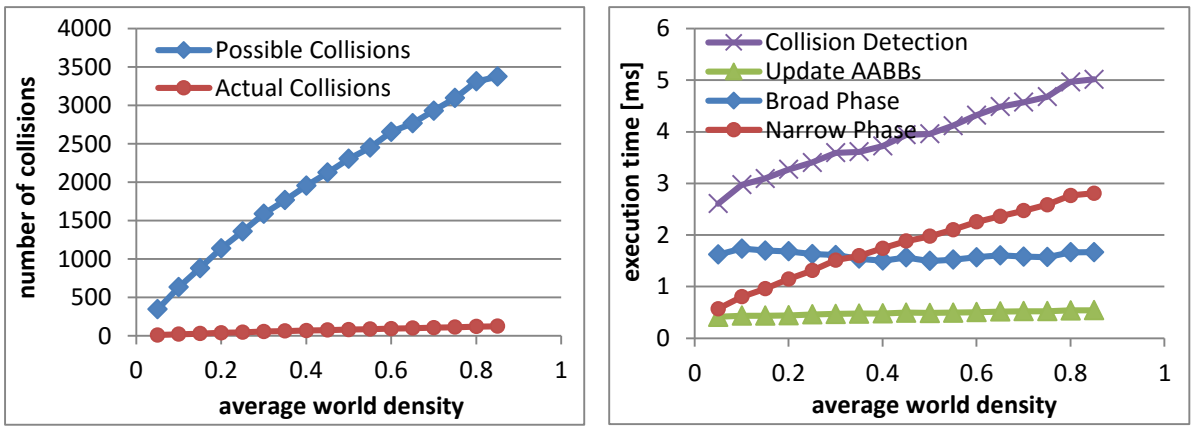


Figure 4.26. Effectiveness and performance of Octree broad phase as a function of average world density

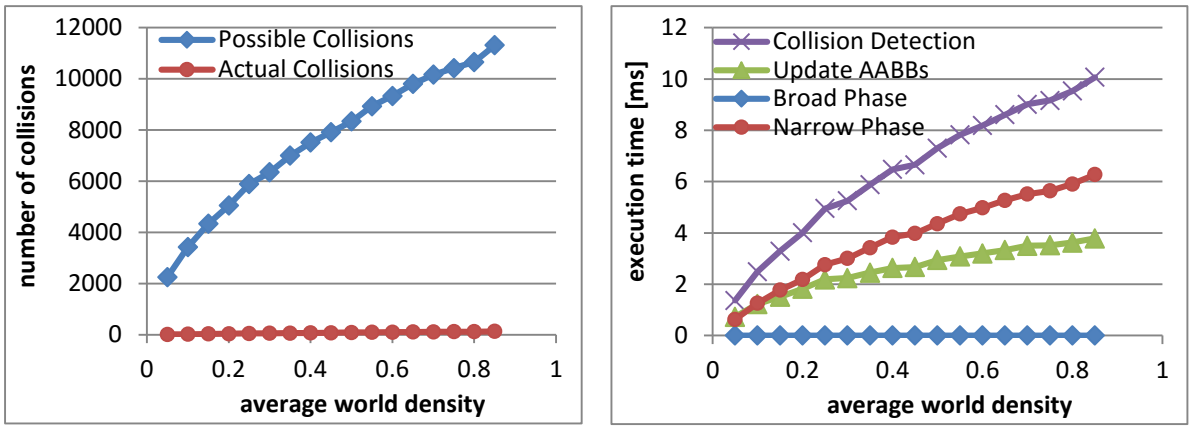


Figure 4.27. Effectiveness and performance of SAP broad phase as a function of average world density

Collected results obviously confirm that the denser the world is, the more frequently objects collide with each other, which can be seen in the left side of figures 4.24 – 4.27. The rate with which the average world density affects the number of actual collisions is much weaker, compared to the impact of increasing the number of objects, which can be seen in the left side of figures 4.3 – 4.6. The reason for that is, in 3D space, a solid object cannot contact all objects in a virtual world at once, which means it can contact only objects from its local surroundings. Therefore making virtual world denser will not increase drastically the number of actual collisions. This cannot be said about the number of collisions between AABBs, because in scenes with large densities, the lack of BV's tightness causes broad phase to report many false positive collisions. Therefore, the execution time of the narrow phase increased as the world become denser in all test cases. In the test case utilizing brute force broad phase algorithm, the execution time of updating AABBs were unaffected by the world density parameter, which can be observed in the right side of the figure 4.24. This result is expected, because no additional data structure accelerating broad phase execution had to be maintained and only one AABB per object has to be calculated. Regarding the execution time of the brute force broad phase itself, its susceptibility to world density is definitely baffling. The only factor, which should affect performance of the brute force broad phase, is the number of objects present in the world. Again, searching for optimization allowing the Bullet library to shorten the execution time for a sparse world is outside the scope of this master's thesis topic. In the test case, which utilizes BVH based broad phase, the performance of a broad phase itself was moderately affected by the world density. It is known, from the theoretical analysis, that the only factors affecting performance of the BVH broad phase are the number of nodes in a BVH and the pruning factor. Because increasing world density causes AABBs to be closer together, the probability of intersection between two AABBs increases, effectively decreasing the pruning factor. Thus, it is expected that the world density affected the broad phase's performance. The density of the world also affected updating objects' AABBs phase. From description in the second chapter, it is known that BVH stores BV for each node in the tree, which represents not only individual objects, but also groups of objects. Moreover, inserting object into a BVH requires resizing all BV associated with all direct ancestor nodes if they do not encapsulate child nodes after an insertion. In a dense world, BVs containing multiple objects will have smaller volume compared to BVs in a sparse world, assuming the same object sizes and deterministic algorithm calculating BV. Thus, it is more likely for moving object to trigger large number of recalculations of AABBs, when AABBs are smaller. This would explain why updating AABBs is slower,

when the world is denser. Moving on to the test case, which utilizes an octree based broad phase, it can be seen in the figure 4.26, that broad phase and update AABBs are unaffected by the world density. Because test cases assume constant number of objects in the world and change world density, the size of the world had to be adjusted accordingly. Despite the fact that a sparse world implies larger size of the world compared to dense world, the octree partitions space proportionally. That fact in conjunction with the assumption that the distribution of objects in the world is uniform allows concluding that algorithm creates roughly the same octree structure with lists of the similar length regardless of the world density, which explains the invariant performance of the octree broad phase and the update AABBs phase. Lastly, in the test case, utilizing SAP broad phase, updating AABBs becomes more expensive as the world density increases. The reason for that is, in a dense world, it is much more probable for even a small change in object's position to trigger reordering of object's endpoints in a projection, compared to sparser world. On other words, in a dense world it is more difficult to maintain scene coherence.

Variety of applications utilizing 3D graphics is enormous, like video games, flight simulators, CAD applications and CGI animation software. All of these applications render models with diverse level of geometrical complexity. Thus is it important to test how collision geometry influences performance and effectiveness of a collisions detection process. To accomplish that, tests with only one type of collision shape were performed. In addition to standard shapes like box, sphere, cylinder, cone and capsule, compound shapes were additionally tested. Compound shapes are represented by a set of basic shapes, which are merged together into one potentially concave shape. Compound shapes can be understood as CSG (Constructive Solid Geometry) shapes created using only union operation. Before tests results will be described, brief description of the collision geometry used in tests has to be provided. As mentioned earlier size of objects was chosen randomly using beta distribution. In order to ensure repeatability of performed tests and for sake of analysis of test results, the figure 4.28 presents expected size in each dimension of each shape used in tests.

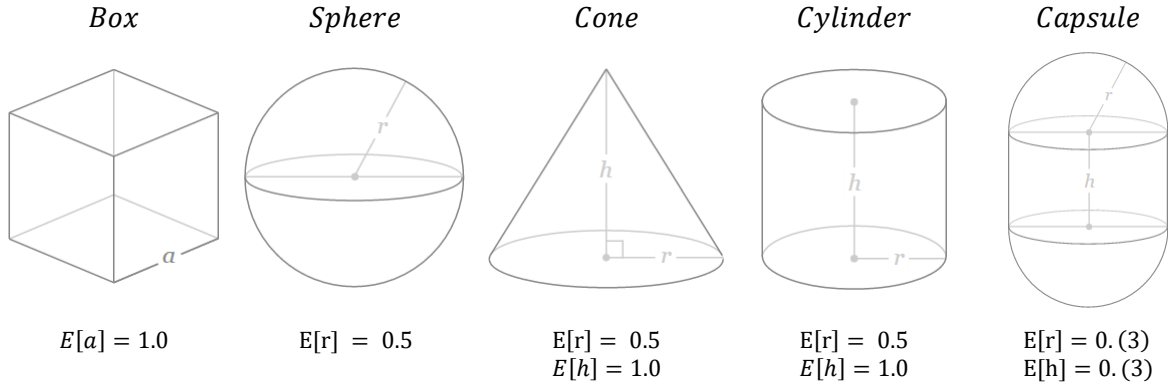


Figure 4.28. Expected sizes of basic shapes used in tests in each dimension

All these averages were selected so that the size of an AABB encapsulating shape will have roughly the same size in all cases. As mentioned before, apart from basic shapes, presented in the figure 4.28, in performed tests compound shapes were also used. They were created by choosing randomly nine shapes from the following set: box, sphere, cylinder, capsule and cone. Size of each element shape was chosen similarly to basic shapes, except that average sizes were four times smaller. Then these nine objects were distributed evenly on the surface of a sphere, which radius was equal to 0.25. Lastly,

each object was translated outwardly by vector, which magnitude was chosen randomly using normal distribution with the average magnitude equal to 0.5. Test results are presented in figures 4.29 – 4.32.

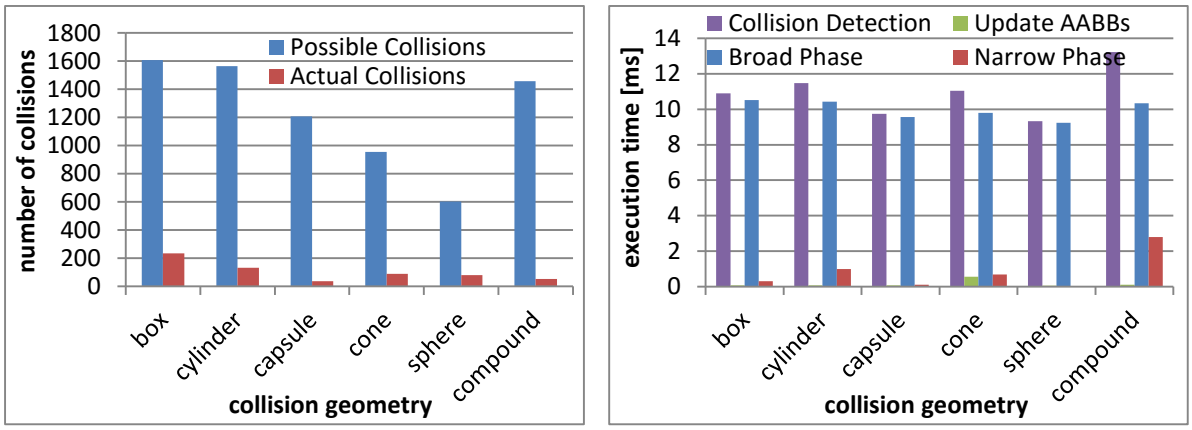


Figure 4.29. Collision geometry impact on effectiveness and performance of brute force broad phase

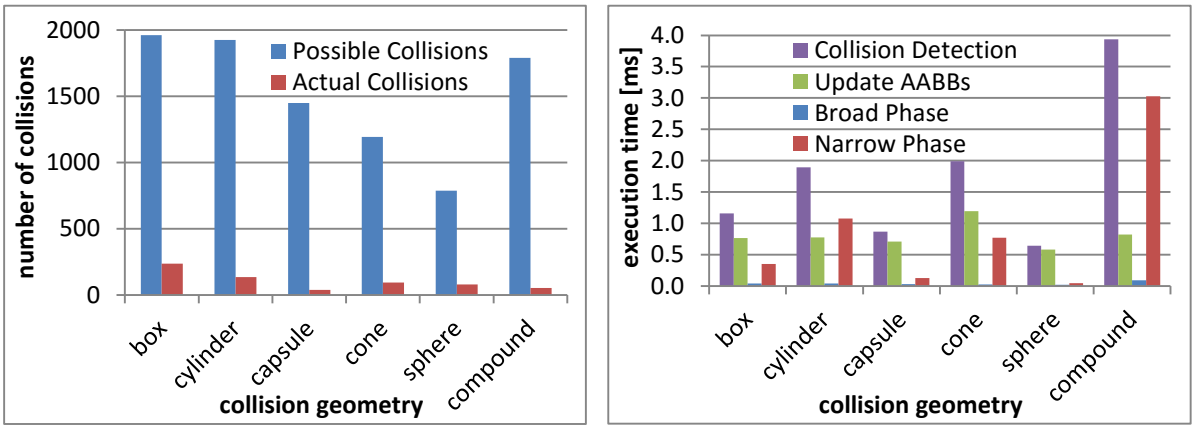


Figure 4.30. Collision geometry impact on effectiveness and performance of BVH broad phase

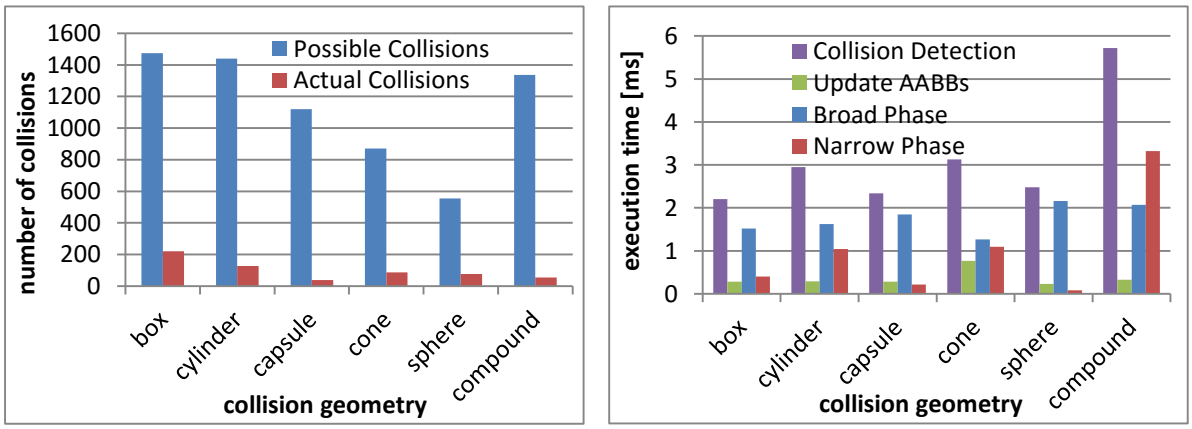


Figure 4.31. Collision geometry impact on effectiveness and performance of Octree broad phase

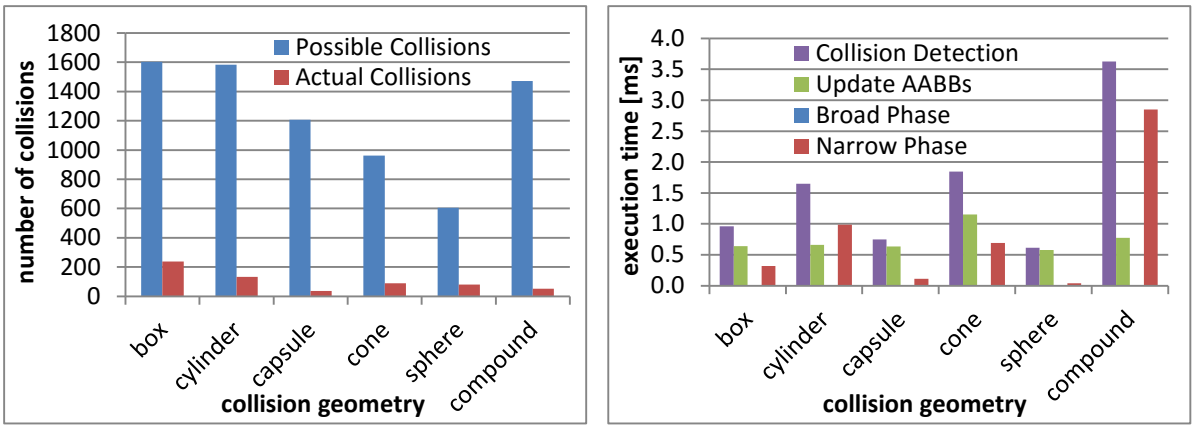


Figure 4.32. Collision geometry impact on effectiveness and performance of SAP broad phase

The number of actual collisions varies across different types of collision geometry. Collision shapes can be ordered by the number of actual collisions as follows: box, cylinder, cone, sphere, compound and capsule. This agrees with intuitive thinking that increasing surface area of an object's shape will increase the probability of a collision between objects, because when collision shapes are ordered by the average surface area, the ordering is almost the same. The last statement refers to only basic shapes, which expected surface area can be calculated easily. Regarding compound shape, the expected surface area is quite challenging to calculate due to complicated algorithm creating compound objects. This is not a problem, because it can be easily observed that the amount of joined volume in a compound shape is usually quite large and the element shapes are four times smaller than basic shapes, resulting in small surface area. Thus, it is not surprise that the number of actual collisions is small. Explaining the number of possible collisions is much more challenging, because at least two factors affect this quantity. Obviously, the tightness of AABB has significant influence on number of possible collisions. Because, as mentioned before, sizes of AABB for basic shapes are similar, the tightness of AABB can be approximated by shape's volume. That means list of basic shapes ordered by the BV tightness is as follows: box, cylinder, sphere, capsule and cone. However, just because tightness of AABBs is high does not automatically mean that the number of collisions between AABBs will be low, because underlying shape of an object affects how that object will react to collisions. Thus, AABBs movements are dependent on encapsulated collision shape. Because of that, probability of collision between AABBs can changed significantly. In order to confirm that this theory is correct, tests, which results are presented in figures 4.29 – 4.32, were performed once again, but with one crucial change - solidity of objects was disabled. Disabling objects solidity was achieved by commenting out code in the btSequentialImpulseConstraintSolver, which is responsible for generating contact constraints. That way collision detection is unaffected and performance results will be reliable. Thanks to the penetrability of objects, movement of all AABBs is the same, in contrast to previous tests in which AABBs were moving differently, depending on collision geometry. Results regarding number of collisions for all broad phases are presented in the figure 4.33.

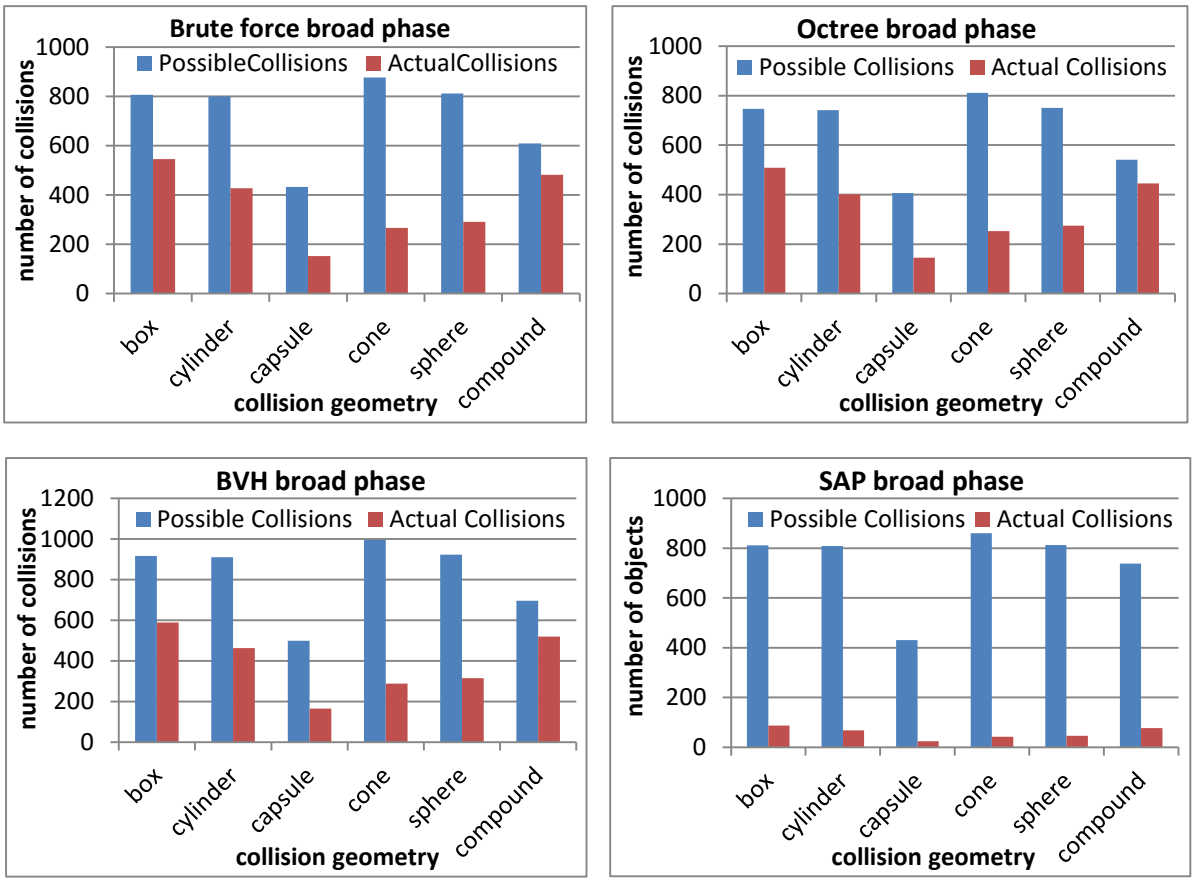


Figure 4.33. Impact of AABB tightness on effectiveness of collision detection

Results confirm that indeed collision geometry does affect the number of possible collisions, because when differences between movements of AABBs disappeared, the possible collisions quantity become invariant regardless of collision geometry type. That means that the significant factor affecting the number of possible collisions in figures 4.29 – 4.32 was movement of AABBs arising from a type of collision shape. Obviously the surface area of AABBs and its tightness stills plays important role, thus it is expected that in case of capsule, the number of possible collisions is smaller, compared to other basic shapes. The reason for that is depth and height of capsule are expected to be two thirds, which significantly alters surface area of its AABB. Regarding the compound shape, the algorithm calculating its AABBs is a compromise between speed and tightness. It iterates through corner points of AABBs encapsulating element shapes and choses the most extreme points, thus it produces results strongly dependent on positions of element shapes. Answering fundamental question about the relationship between collision shape and its movement, which would explain why certain types of shapes are prone to collisions and other are not, requires precise physics model and stochastic analysis, which is outside the scope of this master's thesis topic. Moving on to the execution time of the update AABBs phase, there is not much differences between different types of collision geometry, apart from the case of cone shape. As it turns out, the Bullet library implements creating AABB for a cone using general algorithm for convex shapes, which is clearly not optimal in case of a cone. In addition, the execution times of broad phase algorithms themselves are unaffected by the type of collision geometry, which is an expected result, because objects in broad phases are handled using AABBs. Lastly, the execution time of a narrow phase is definitely affected by the type of collision geometry. This is also expected, because

performance of a narrow phase is dictated by the number of pairs suspected for collision reported by the broad phase and cost of finding intersection points between pairs of objects, which are tested in the narrow phase. For instance, the numbers of collisions between AABBs encapsulating compound, box and cylinder shapes are very similar, but the execution times of the narrow phase are vastly different. Obviously, the execution times are longer for shapes with more complex geometry, thus narrow phase with compound shapes is the longest and narrow phase with boxes is the shortest. However, in case of capsule shape, despite that it has larger number of possible collisions compared to the case of cone shape, it also has shorter narrow phase. The reason for that finding intersection points for two capsules is cheaper compared to finding intersection points for two cones. To give and idea why that is the case, one could consider much more simple test, which just checks for intersection and does not find intersection points. In order to test whether two capsules are intersecting, comparing sum of radiuses of both capsules with distance between nearest centers of capsule's hemispheres is sufficient. Testing intersection between two cones is not that straightforward. Lastly, in the case of the sphere shape, the number of possible collisions was the lowest and the algorithm finding contact points between two spheres is quite cheap, thus the execution time of the narrow phase was the shortest.

4.3. Conclusions

All algorithms have various properties and computer science is mainly about creating algorithms and understanding its properties. From engineering perspective, properties of an algorithm are its strengths and weaknesses, in context of algorithm's usage. What engineering focuses on is emphasizing algorithm strengths and hiding its weaknesses by understanding algorithm's properties. From this perspective, results of empirical tests allowed to draw very interesting conclusions, which illustrates landscape of broad phase algorithms and their influence on performance of collision detection.

Firstly, the SAP broad phase was the most efficient algorithm and the BVH broad phase was little less efficient in vast majority of cases. However, the SAP broad phase did not cope with scenes characterized by very low coherence, achieving results comparable to naive approach. Additionally in all executed test cases, the BVH broad phase was second most efficient algorithm, with relatively small difference to the most efficient algorithm, or was the most efficient algorithm. That said the BVH broad phase has one more disadvantage over the SAP broad phase, which is the amount of effort needed to develop an implementation in terms of time and complexity. Obviously, utilizing some preexisting physics library can solve this problem. Therefore, if an application contains scenes with low coherence and drops in the performance of that application are not acceptable, for instance a video game designed for Virtual Reality (VR), the most appropriate choice is to use the BVH broad phase. In other cases, the SAP broad phase should be better solution.

Regarding attempts to improve the collision detection process, empirical tests clearly show that for the best algorithms like SAP and BVH, the broad phase itself is extremely well optimized and has very small contribution to overall collision detection. The update AABBs phase is much more interesting, because in all tests its contribution to collision detection was significant with the exception of all test cases utilizing brute force broad phase. The reason for that is, in case of brute force broad phase only objects' AABBs are updated and no additional calculations are performed. In cases of BVH, Octree and SAP broad phases the update AABBs phase in addition to updating objects' AABBs, maintains data structures required by broad phase algorithms. Fox example, updating AABB of an object in the Octree

broad phase will also execute delete and insert operations on the octree. The empirical results show that in most cases roughly half of the cost associated with the collision detection is dedicated to maintaining data structures for SAP and BVH broad phases. That was the basic idea for implementing the Octree broad phase. Spatial partitioning algorithms have much lower cost of maintaining theirs data structures compared to algorithms, which build their data structures based on collision geometry. The reason for that is, in spatial partitioning algorithm only partitioning scheme decides how space is divided and collision geometry is ignored. Unfortunately, savings in the update AABBs phase were not cost effective, because cost of the broad phase increased significantly. This however does not indicates that another broad phase algorithm with lower maintenance cost and comparable broad phase efficiency to BVH and SAP cannot be created. It would be interesting to experiment with an Octree broad phase, which would divide objects into smaller pieces or testing various hashing strategies used the HGrid broad phase.

Performed empirical tests provide very useful insights regarding factors affecting collision detection and magnitude of their impact. Obviously, the most influential factor was the number of objects. Its impact reflected computational complexity of particular broad phase algorithm. The second most influential factor was the world density, which affected mostly narrow phase, causing its execution time to grow in a linear fashion with quite steep slope. The expected objects' velocity quantity has moderate influence on the collision detection process compared to increasing world density or number of objects. When objects start moving, performance and effectiveness decreases rapidly, but as objects exceeded certain speed, the influence becomes much weaker. The stationary objects fraction also moderately affects collision detection. The effectiveness and performance of a collision detection is inversely proportional to the percentage of stationary objects in a linear fashion with mild slope. The last factor affecting collision detection, more specifically the narrow phase, is the collision geometry. When the geometrical complexity of objects increases calculating contact points becomes very expensive and can become major bottleneck of a whole collision detection. Lastly, the uniformity of objects sizes does not affects tested broad phase algorithms and collision detection as a whole, because they were designed to handle various objects' sizes. Therefore, artists should focus on creating scenes with the smallest number of objects possible. Additionally they should pay close attention to keeping density of objects in scenes small. Lastly, it is vital for them to create objects with as little geometrical complexity as possible.

5. SUMMARY

In retrospection, the most important goal of reviewing efficient collision detection algorithms and various techniques used in their broad phases was achieved and results of that work are presented in detail in the second chapter. The task to design and implement empirical tests allowing measuring quantities concerning collision detection algorithms in controlled environment, was completed. The description of these tests and obtained results can be found in the fourth chapter. Additionally the conclusions section of the fourth chapter describes recommendations concerning choosing suitable broad phase algorithm and collision geometry.

Regarding the goal of attempting to improve performance of collision detection algorithms, the goal was not met by creating brand new algorithm. The reason for that is optimizing software can also be done by locating the most expensive parts of a code and then optimizing them. Thus, optimizing collision detection algorithms should be focused on the most expensive stages of collision detection. The first step moving towards that goal was accomplished thanks to results of empirical tests, which shown that broad phases' data structures are expensive to maintain. Thanks to that conclusion, direction in which future work should be headed was established.

The implementation of chosen broad phase algorithm – the octree based broad phase was added to the Bullet library mostly for the purpose of empirical tests. However, because spatial partitioning methods are characterized by relatively low maintenance cost, there was hope for this method to compete with broad phases already implemented in the Bullet library. Unfortunately empirical tests shown that reducing maintenance cost is not optimal, in the case of the octree broad phase, due to increase in cost of pruning collision tests. Despite poor performance results delivered by the the octree broad phase, its implementation allowed to draw interesting conclusions. Strategies used in broad phase inherently have to optimize the balance between maintenance cost of data structures used to accelerate collision detection and performance of pruning unnecessary collision tests. Essentially additional maintenance work has to be cost effective. The optimal balance between maintenance cost and performance of pruning is shifted towards methods with very fast pruning and more expensive maintenance cost than required by the octree broad phase, like SAP and BVH broad phases.

All broad phase algorithms described in the second chapter were groups of methods sharing common strategy rather than rigid list of instructions and could be altered by tweaking various functionalities used by them. For instance, different methods of calculating insertion cost in a BVH could vastly change its structure and maintenance cost. Another tweak involving BVH is to store one BVH for static objects, which does not need to be maintained, and another one for dynamic objects. Obviously all these tweaks increase performance of collision detection only in certain scenarios and always applying them would harm efficiency. Furthermore, scenes tested in empirical tests were designed to represent collision geometry encountered in most popular applications, but cannot be adequate for all types of applications. Therefore improving collision detection algorithms could be focused on adjusting functionalities used by collision detection algorithms to suit particular application instead of building new algorithms. It would be worth to consider implementing all broad phase algorithms and functionalities required by them and create testing framework choosing automatically which algorithm with which configuration of functionalities yields the best performance results for particular application.

BIBLIOGRAPHY

1. Erwin Coumans: *AAA Titles using Bullet?*, http://www.bulletphysics.org/Bullet/phpBB3/viewtopic.php?p=11971&f=&t=#p11971, (date of access 05.28.2016)
2. Erwin Coumans: *Bullet 2.83 Physics SDK Manual*, https://github.com/bulletphysics/bullet3/raw/master/docs/Bullet_User_Manual.pdf, (date of access 05.28.2016)
3. Erwin Coumans: Bullet 2.83 Quickstart Guide, https://github.com/bulletphysics/bullet3/blob/master/docs/BulletQuickstart.pdf, (date of access 05.30.2016)
4. Doxygen Bullet Documentation, http://bulletphysics.org/Bullet/BulletFull/index.html, (date of access 05.28.2016)
5. Richard S. Wright, Nicholas Haemel, Graham Sellers, Benjamin Lipchak: *OpenGL SuperBible, Fifth Edition, Comprehensive Tutorial and Reference*, Addison-Wesley Proffessional, 2010.
6. Song Ho Ahn: *OpenGL Transformation*, http://www.songho.ca/opengl/gl_transform.html, (date of access 05.28.2016)
7. *PDI Dreamworks Megamind, Shrek 4 and 'How to train your dragon' are using Bullet*, http://bulletphysics.org/wordpress/?p=241, (date of access 05.28.2016)
8. Christer Ericson: *Real-Time Collision Detection, The Morgan Kaufmann Series in Interactive 3-D Technology*, CRC Press, 2004.
9. Thorben Linneweber: *Sweep and Prune*, http://jitter-physics.com/wordpress/?tag=sweep-and-prune, (date of access 05.28.2016)
10. *Wikipedia, 3D computer graphics*, https://en.wikipedia.org/wiki/3D_computer_graphics, (date of access 05.28.2016)
11. *Wikipedia, Collision detection*, https://en.wikipedia.org/wiki/Collision_detection, (date of access 05.28.2016)
12. *Wikipedia, Soft body dynamics*, https://en.wikipedia.org/wiki/Soft_body_dynamics, (date of access 05.28.2016)

LIST OF FIGURES

LIST OF TABLES

STRESZCZENIE

Popularność wizualizacji komputerowych opartych o grafikę 3D w ciągu ostatnich dwudziestu pięciu lat eksplodowała i w obecnych czasach jest uznawana za standard. Grafika 3D jest oczywiście wykorzystywana w grach komputerowych, filmach animowanych czy programach typu CAD, jednakże grafika 3D znalazła również znacznie mniej oczywiste zastosowania takie jak obrazowanie medyczne, symulatory lotu czy symulacje komputerowe. Przytłaczająca większość aplikacji generuje grafikę 3D modelując fizykę ciał poruszających się w wirtualnej przestrzeni, więc wymaga użycia silnika fizycznego zapewniającego ruch obiektów zgodny z zasadami dynamiki Newtona. Fundamentalną częścią silnika fizycznego jest moduł wykrywający kolizję pomiędzy obiektami. Z racji tak dużej popularności grafiki 3D, kluczowe jest, aby algorytmy detekcji kolizji działały wydajnie i stabilnie, zapewniając wysoką precyzję i wiarygodność symulacji fizyki. Ponieważ rozległość tematyki związanej z detekcją kolizji jest ogromna, ta praca magisterska skupia się na kluczowym elemencie detekcji kolizji, jakim jest broad phase. Oczywiście analiza teoretyczna algorytmów zawarta w pracy jest wsparta testami empirycznymi, których wyniki dostarczają interesujących wniosków.

Zagadnie detekcji kolizji jest wyzwaniem pod względem technicznym, ponieważ wymagania, które stawia się przed aplikacjami korzystającymi z grafiki 3D są bardzo rygorystyczne. Niewątpliwie jednym z największych wyzwań jest detekcja kolizji w aplikacjach generujących grafikę 3D w czasie rzeczywistym. Oczekiwane jest, że moduł detekcji kolizji będzie wstanie wykryć kolizje pomiędzy tysiącami obiektów w czasie rzędu kilku milisekund. Dla części aplikacji istotny jest nie tylko średni czas wykonania algorytmu, ale również czas wykonania w przypadkach pesymistycznych. Idealnym przykładem takich aplikacji są gry pisane z myślą o hełmach wirtualnej rzeczywistości, ponieważ każde spowolnienie działania programu powoduje utratę płynności animacji wywołującą chorobę lokomocyjną. Algorytmy detekcji kolizji spotykają się również z wieloma ograniczeniami, takimi jak ograniczona precyzja obliczeń numerycznych oraz wielkość dostępnej pamięci operacyjnej. Oczekuje się, że mimo błędów obliczeń numerycznych algorytm zachowa stabilność numeryczną. Istotne jest również minimalizowanie zużytej pamięci przez struktury danych wykorzystywane przez algorytmy.

Główną funkcjonalnością modułu detekcji kolizji jest informowanie obiektów o zaistniałych kolizjach. Dzięki tej informacji obiekty mogą reagować na kolizje zapewniając, że nie przenikają się nawzajem, a także mogą się odkształcać i rozpadać. Ponieważ teoretycznie każdy obiekt w wirtualnej przestrzeni może kolidować ze wszystkimi innymi obiektami, a w rzeczywistych aplikacjach jedynie mały procent obiektów koliduje ze sobą w danym momencie, to wykrywanie kolizji dzieli się na dwie fazy: broad phase i narrow phase. Algorytm broad phase wykorzystuje specjalną strukturę danych, odzwierciedlającą ułożenie obiektów w przestrzeni, za pomocą której bardzo szybko odrzuca ogromną liczę par obiektów o których wiadomo, że nie kolidują. Dzięki tej technice algorytm narrow phase wykrywa kolizje pomiędzy bardzo małą liczbą par obiektów. Pierwsza grupa algorytmów broad phase charakteryzuje się wykorzystaniem hierarchicznej struktury danych, w której każdy wierzchołek jest utożsamiany z BV otaczającym grupę obiektów. Korzeniem BVH jest wierzchołek grupujący wszystkie obiekty, natomiast każdy liść BVH zawiera pojedynczy obiekt. Zazwyczaj BVH jest drzewem binarnym, w którym każdy wierzchołek dzieli wszystkie enkapsulowane obiekty na dwa klastry, które są enkapsulowane przez wierzchołki potomne. Opisana organizacja obiektów w BVH pozwala wydajnie odrzucać niepotrzebne testy kolizji, ponieważ gdy dwa wierzchołki nie przecinają się, to oznacza, że

kolizje pomiędzy wierzchołkami potomnymi również nie występują. Wynika to z faktu, że w BVH wszystkie BV należące do wierzchołków potomnych są enkapsulowane przez BV wierzchołka, który jest ich przodkiem. Drugą grupą algorytmów broad phase są algorytmy wykonujące tzw. spatial partitioning, które głównie różnią się od algorytmów BVH kryterium za pomocą którego dzielą przestrzeń. Algorytmy wykonujące spatial partitioning dzielą przestrzeń według predefiniowanego schematu, natomiast algorytmy BVH dzielą przestrzeń w zależności od geometrii kolizji. Sztandarowym przykładem algorytmu wykonującego spatial partitioning jest Octree broad phase, w którym cała przestrzeń jest zamknięta w sześcianie dzielonym rekurencyjnie, na osiem sześcianów o dwukrotnie krótszych krawędziach, wzdłuż płaszczyzn partycjonowania wyznaczonych za pomocą osi X, Y oraz Z tak, aby otrzymać drzewo ósemkowe. Do opisanej struktury danych wstawia się obiekty zgodnie z następującą zasadą: obiekt wstawia się do możliwie małego wierzchołka octree, którego sześcian całkowicie zawiera BV wstawianego obiektu. Struktura danych octree pozwala ograniczyć testy kolizji obiektów enkapsulowanych przez wierzchołek octree do testowania obiektów przypisanych do bezpośrednich przodków tego wierzchołka, wszystkich jego potomków oraz jego samego. Takie ograniczenie jest możliwe, ponieważ tylko sześciany należące do wymienionych wierzchołków mają jakiekolwiek części wspólne oraz wiadomo, że obiekty muszą być enkapsulowane przez sześciany wierzchołków, do których są przypisane. Trzecią i ostatnią grupą algorytmów broad phase są algorytmy wykonujące tzw. spatial sorting, które wyróżniają się wykorzystaniem struktury danych, która zawiera obiekty posortowane zgodnie z ich ułożeniem w przestrzeni. Algorytmy wykorzystujące spatial sorting są czasem nazywane algorytmami wykorzystującymi chwilową koherencję sceny. Najpopularniejszą metodą wykonującą spatial sorting jest SAP, w którym utrzymuje się projekcje końców obiektów na wybrane wektory kierunków. Algorytm SAP odrzuca niepotrzebne testy kolizji dzięki kluczowej właściwości wspomnianych projekcji – gdy obiekty nie przecinają się na projekcjach, to jest pewne, że nie przecinają się w przestrzeni wirtualnej. Zatem algorytm SAP zwraca tylko te pary obiektów, które przecinają się na wszystkich projekcjach utrzymywanych przez algorytm.

W celu uzupełnienia teoretycznej analizy algorytmów, została uruchomiona seria empirycznych testów oceniających wydajność i skuteczność algorytmów broad phase. Wyniki testów pokazały, że algorytm SAP jest najwydajniejszy we wszystkich przypadkach z wyjątkiem scen o bardzo dużej gęstości obiektów, w których wydajność algorytmu SAP jest bardzo słaba. Algorytm BVH również okazał się wydajny ustępując nieznacznie algorytmowi SAP we wszystkich przypadkach oprócz scen o bardzo dużej gęstości obiektów, w których poradził sobie najlepiej. Bardzo dużą zaletą algorytmu BVH była przewidywalna wydajność niezależnie od konfiguracji sceny, która jest szczególnie istotna w aplikacjach wirtualnej rzeczywistości. Algorytm Octree okazał się znacznie mniej wydajny od algorytmów SAP i BVH, lecz nadal znacznie lepszy od algorytmu naiwnego. Przyczyną obniżonej wydajności algorytmu Octree była liczność obiektów, które przecinały płaszczyzny partycjonowania sześcianów, a w konsekwencji musiały zostać umieszczone znacznie wyżej w drzewie ósemkowym niż by ich wielkość na to wskazywała. Wspomniany problem występuje w każdej metodzie partycjonującej przestrzeń, ponieważ geometria kolizji obiektów musi się wpasowywać w geometrię podziału przestrzeni. Z kolei zaletą metod partycjonujących przestrzeń jest bardzo niski koszt utrzymania struktur danych, ponieważ ich geometria nie zależy od geometrii kolizji obiektów. Testy empiryczne pokazały, że metody utrzymujące kosztowniejsze, lecz odrzucające większą liczbę niepotrzebnych testów kolizji, struktury danych są bardziej opłacalne.