BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING
FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE
ÚSTAV AUTOMATIZACE A INFORMATIKY

COLLISION DETECTION IN 3D SPACE
DETEKCE KOLIZE OBJEKTŮ V 3D PROSTORU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — Bc. Jan Grulich
AUTOR PRÁCE

SUPERVISOR — Ing. et Ing. Stanislav Lang
VEDOUCÍ PRÁCE

BRNO 2016

Abstrakt

Práce se zabývá detekcí kolizí v 3D simulačním prostoru. V první části jsou popsány nejpoužívanější algoritmy pro detekci, stejně jako některé knihovny hotových řešení. Druhá část práce obsahuje popis testovacího softwaru vytvořeného na základě knihovny OpenGL, včetně popisu důležitých částí. V poslední části práce jsou také prezentovány výsledky testování a porovnání vybraných algoritmů na vytvořených testovacích úlohách.

Summary

The thesis deals with collision detection in 3D simulation space. In the first part, the most used algorithms for detection are presented as well as some complete solution libraries. The second part contains the description of the testing software, which is based on OpenGL library, including the description of important segments. The final section presents some testing problems on which the chosen algorithms were tested, results and method comparison.

Klíčová slova

detekce kolizí, Gilbert-Keerthi-Johnson algoritmus, nejmenší konvexní obálka, OpenGL, dělení prostoru, obalová tělesa, hierarchie obalových těles

Keywords

collision detection, Gilbert-Keerthi-Johnson algorithm, convex hull, OpenGL, spatial partitioning, bounding volumes, bounding volume hierarchies

GRULICH, J. *Detekce kolize objektů v 3D prostoru.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2016. 68 s. Vedoucí Ing. et Ing. Stanislav Lang.

Prohlašuji, že jsem diplomovou práci *Detekce kolizí v 3D prostoru* vypracoval samostatně pod vedením Ing. et Ing. Stanislava Langa s použitím materiálů uvedených v seznamu literatury

Bc. Jan Grulich

Děkuji vedoucímu diplomové práce Ing. Stanislavu Langovi a odbornému konzultantovi Ing. Václavu Velebovi za odbornou pomoc a cenné rady při psaní této práce. Dále děkuji spolužákům za dobrou atmosféru a vzájemnou pomoc při cestě studiem a v neposlední řadě také mé přítelkyni a rodině za podporu a trpělivost.

Bc. Jan Grulich

Contents

1. The Introduction

The movement of the robot can be risky operation. As far as the robot can't distinguish the neighborhood, there will always be some danger of damaging it due to collisions. The lowering of the danger can be achieved by simulating the movement by some software, where the the risk of destroying the device is eliminated. The human eye can easily distinguish a possible collision, because the object is represented as a whole, but the computers see objects only as sets of numerical data. The aim of this thesis is presenting popular methods and algorithms for detecting collisions in 3D space, which can be used as the detection tool in a simulation software, and also the OpenGL-based simulation program implemented as visualizer of performed tests.

The overview is divided into three parts. Chapter 3 presents basic algorithms based on simple geometry and tree structures. Those algorithms are often older than the following ones. The algorithms running on convex hulls are great compromise between accuracy and cost of the algorithm and they are presented in chapter 4. There are also some complete solutions, which can be obtained as libraries or executable application. In chapter 5 those libraries are presented as well as terms of their usage.

Chapter 6 describes implemented application and the comparison of the chosen algorithms is presented in the last chapter. The direct comparison is difficult, because different algorithms are designed for different purposes. Due to this limitation, only groups of algorithms are tested if they fulfill given theoretical assumptions.

2. Theoretical Background

In this chapter, an important mathematical theory, which is used in this thesis, will be introduced. The thesis is based mainly on 3-dimensional Geometry discipline and the main goal is to explain some nontrivial mathematical constructions, not giving a complete overview of disciplines. More details can be found in referenced literature.

2.1. Convex objects

The first, and maybe the most important feature of the object is its convexity. A set S is convex if and only if every pair $(x, y) \in S$ has its connecting line segment contained in S (fig. 2.1).

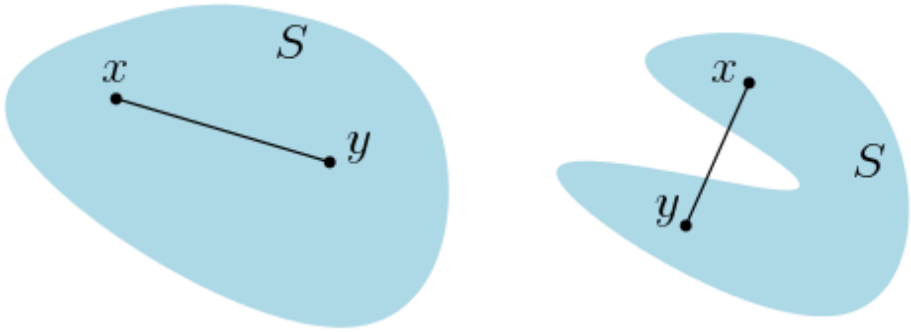

Figure 2.1: Convex and non-convex object

This definition is simple and easy to understand, but insufficient. Let S be sphere in 3-dimensional space. With this definition it is non-convex shape, so a redefinition is needed for case of two-dimensional surfaces in 3D. These surfaces are convex if the volumes they define are convex sets. With this definition, the sphere is convex object. The example of non-convex object can be torus.

2.2. Minkowski sum and difference

In this section two important operations will be described, *Minkowski sum* and *Minkowski difference*. These operations are similar to numerical sum and difference, but operands are sets instead of numbers. Let S and T be two point sets in R^n and let $\mathbf{s}$ and $\mathbf{t}$ be the relevant position vectors to sets S and T. Minkowski sum is also a set marked by $S \oplus T$ and defined as follows:

$$S \oplus T = \{\mathbf{s} + \mathbf{t} : \mathbf{s} \in S, \mathbf{t} \in T\},$$

where $\mathbf{s} + \mathbf{t}$ is the vector sum of vectors $\mathbf{s}$ and $\mathbf{t}$. The Minkowski sum is illustrated in figure 2.2

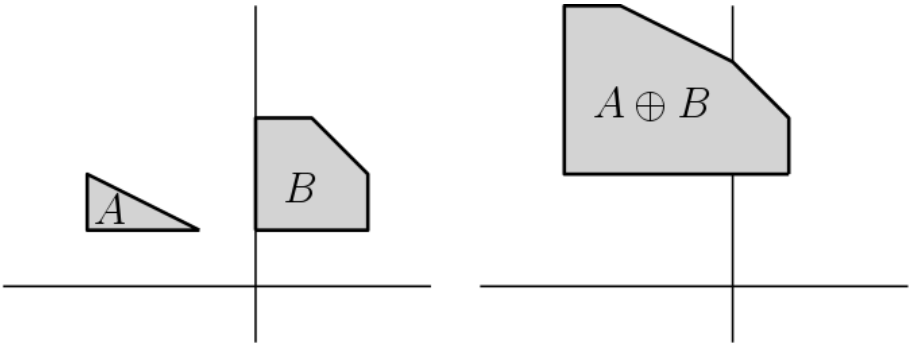

Figure 2.2: Minkowski sum of two objects

The Minkowski difference definition is similar to sum. The difference $S \ominus T$ of sets S and T is defined as follows:

$$S \ominus T = \{\mathbf{s} - \mathbf{t} : \mathbf{s} \in S, \mathbf{t} \in T\}.$$

The Minkowski difference example is in figure 2.3.

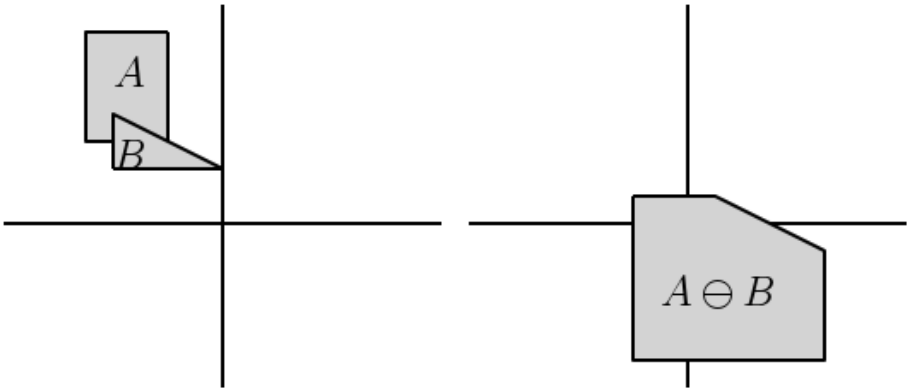

Figure 2.3: Minkowski difference of two objects

Relationship between Minkowski sum and difference can be expressed as $S \ominus T = S \oplus (-T)$, so both operations are marked as Minkowski sum. The Minkowski difference can be used for collision detection, when we need to know if two point sets have at least one point in common. Two point sets S,T collide if and only if Minkowski difference $U = S \ominus T$ contains the origin. Above that we can find in U more details. The minimal distance between S and T is equal to minimal distance between U and the origin and this feature is used in Gilbert-Johnson-Keerthi algorithm. This fact can be easily proved:

$$dist(S,T) = min\{\|\mathbf{s} - \mathbf{t}\| : \mathbf{s} \in S, \mathbf{t} \in S\} = min\{\|\mathbf{u}\| : \mathbf{u} \in S \ominus T\}.$$

2.3. Voronoi regions

A very important construction used in collision detection tests is Voronoi region. Let S be set of points in R^n. Voronoi region $V \subseteq R^n$ of $P \in S$ is a set of points closer to point P than to other points in S. For collision detection this theory has to be extended a little. Modern systems are working with 3D models, which are defined by huge amount of triangles, so regions defined only on points aren't sufficient. Let T be d-simplex defined by n+1 points $A_1, A_2, ..., A_{n+1}$ in n-D space and let P be point in same space. Voronoi regions for $A_1, A_2, ..., A_{n+1}$ can be found, in this case they would be infinite and would cover whole R^n, so P would be in one region. Scatter Voronoi region is not sufficient, because minimal distance between T and P doesn't have to be equal to minimal distance between P and points $A_1, A_2, ..., A_{n+1}$. This is the reason why *features* of polyhedron are defined. Let a feature of T be one of object's faces, edges of vertices. Voronoi region of a feature F is a set of points closer to feature F than to other features of T. The boundary planes of regions are called *Voronoi planes*. Voronoi regions of triangle's features are presented in figure 2.4 and regions of cube in figure 2.5.

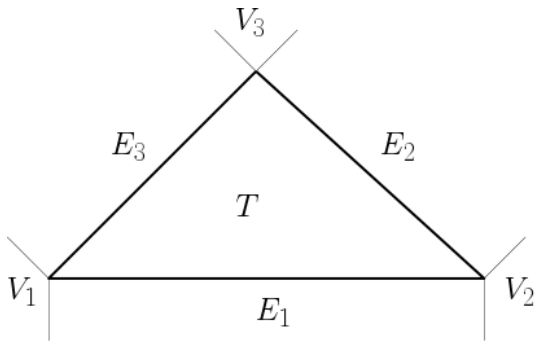

Figure 2.4: Seven Voronoi regions of triangle: the face region(T), three edge regions(E_1, E_2, E_3) and three vertex regions(V_1, V_2, V_3).

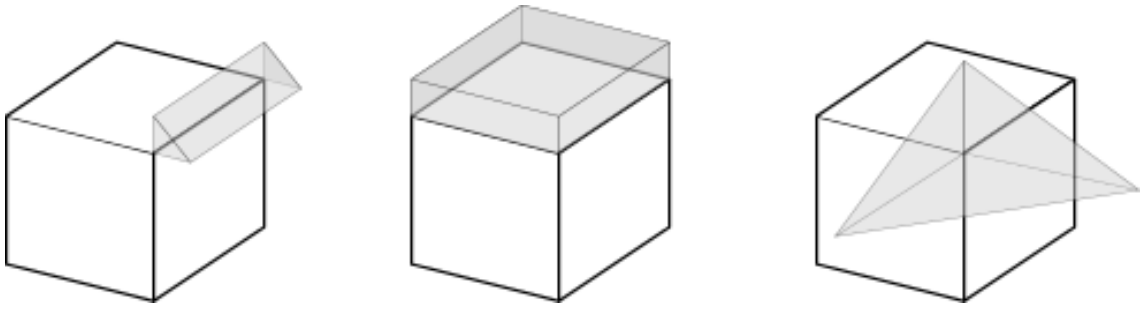

Figure 2.5: Voronoi regions of the cube: edge, face, vertex

2.4. Barycentric coordinates

To complete chapter it is also necessary to mention *barycentric coordinates*. These coordinates represent the way to parametrize the space by weighted combinations of known points. Let R, S be points forming the line segment and let u be vector $S-R$. Every point P on the segment can be expressed as $R+tu = R+t(S-R) = S(1-t)+Rt = Su+Rv$, where $u+v=1$, $0 \leq u \leq 1$, $0 \leq v \leq 1$. The example is on figure 2.6.

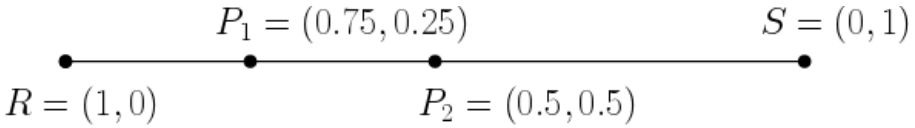

Figure 2.6: Barycentric coordinates of four points on line RS.

3. Basic collision detection methods

The most basic method for collision detection may be direct testing of two objects against each other. In practice that means testing each polygon of first object to each polygon of the second object. This approach seems to be very expensive as far as each object consists of thousands of polygons. Above that typical scene consists of many objects, so complexity can raise even in factorial way. Due to this limitation many advanced algorithms were developed to reduce complexity and computational time.

3.1. Bounding Volumes

To make tests faster, each object can be bounded by an approximate object, which is called bounding volume. A *Bounding Volume* is a simple object, such as sphere or box, which is easy to describe and its collision detection tests is cheap. Such objects can contain one or more complex objects, depending on application.
As long as we work with these approximated objects, collisions tests can be inaccurate. They can detect collisions in situations where objects don't collide but are very close to each other. Thus, for every special application, the compromise between accuracy and cost of the algorithm has to be found.
Every geometric object, which is used as the bounding volume, should satisfy following properties [7].

- Inexpensive intersection tests
- Tight fitting
- Inexpensive to compute
- Easy to rotate and transform
- Use little memory

Naturally, not all these properties can be well satisfied by one type of bounding objects, so it is necessary to determine the most suitable one to the application.

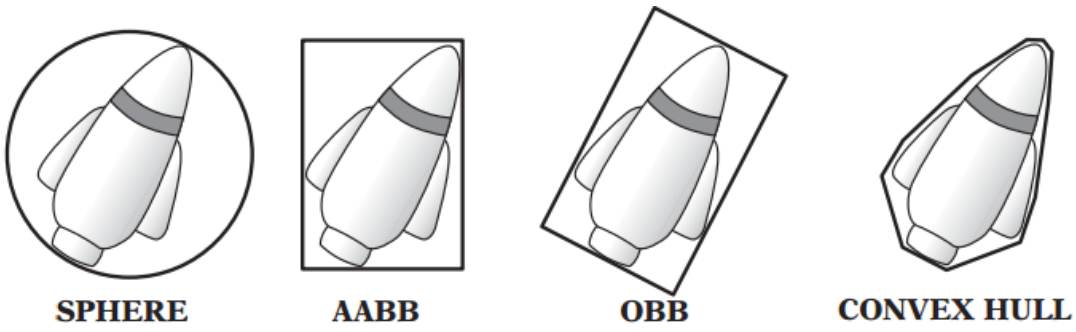

Figure 3.1: Bounding volume types [7]

Figure 3.1 illustrates objects mainly used in intersection tests, which are ordered with respect to their properties. In the next section these types will be briefly described, except for the Convex Hull, which is described in chapter 4.

3.1.1. Spheres

The sphere is the simplest commonly used bounding volume. It uses little memory and its intersection tests are very cheap, but inaccurate. It is necessary to hold only center position and radius in memory. Important feature is also its rotational invariance, which makes bounding volume independent of rotation and new position is reached only by translation.
Intersection test between two spheres is simple. Let A, B be spherical bounding volumes given by its centers c and radii r. Then the collision detection test *testSphere*(A, B) is defined as follows:

Algorithm 1 Sphere-Sphere intersection test

Precondition: A and B are spheres defined by its centers c and radii r

```
1: function testSphere(A, B)
2:     dist ← A.c − B.c
3:     sqrDist ← dot(dist, dist)
4:     radii ← A.r + B.r
5:     sqrRadii ← sqr(radii)
6:     if sqrDist ≤ sqrRadii then
7:         return True
8:     end if
9:     return False
10: end function
```

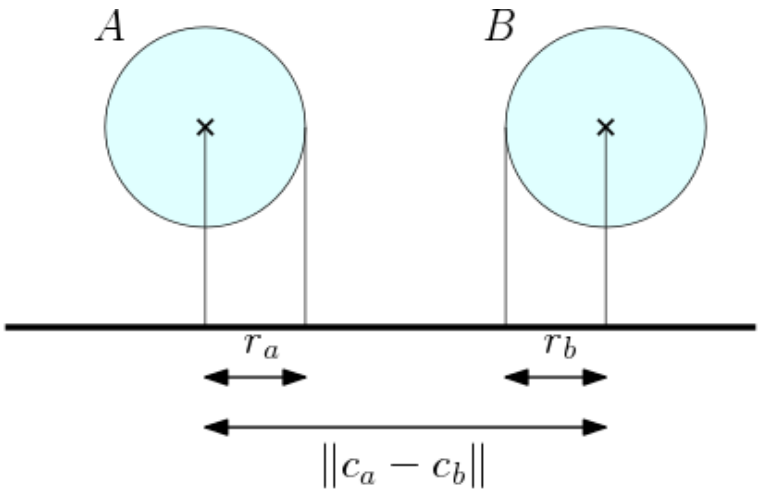

Figure 3.2: Sphere-sphere collision test

The biggest problem of spherical bounding volumes is its fitting. Finding the minimal sphere for given point set by brute-force algorithms means to examine all different combinations of four points and check if all other points fit into the sphere. This algorithm has complexity $O(n^5)$ which is inapplicable in practice. That's the reason why Welzl's iterative algorithm was developed. It processes point by point and recomputes the minimal sphere when necessary. The following pseudocode describes the algorithm, more details can be found in [3].

Algorithm 2 Minimal sphere recursive computation

Precondition: *points*, *support*

```
1: function MINIMALSPHERE(points, support)
2:     if size of points = 0 then
3:         return sphere based on support points
4:     end if
5:     i ← random index from points
6:     actualMinimalSphere ← minimalSphere(points, support)
7:     if actualMinimalSphere contains points[index] then
8:         return actualMinimalSphere
9:     end if
10:    append points[index] to support
11:    Erase points[index]
12:    return minimalSphere(points, support)
13: end function
```

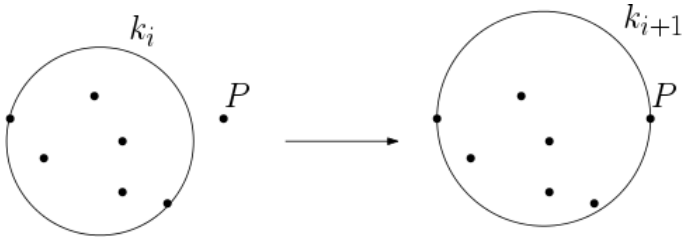

Figure 3.3: One iteration of 2D Welzl's algorithm. Point P is added to circle k_i, which makes circle k_{i+1}

Summary		
Asymptotic complexity	creating	$\mathcal{O}(n)$
	updating	$\mathcal{O}(1)$
	test	$\mathcal{O}(1)$
Memory requirements	center	3 * size of used data type
	radius	1 * size of used data type

3.1.2. Axis-aligned Bounding Boxes

The *Axis-aligned bounding box* (AABB) is also a type of bounding volume, which has cheap intersection test and uses little memory. It is six-sided box (in 3D) and each side facet's normal is oriented parallely with coordinate system's axes. The AABB is represented with its *center* and vector of *halfwidths* in all directions.

Let S be set of points in 3-dimensional space. The minimal bounding AABB is computed as follows:

Algorithm 3 AABB computing

Precondition: S is set of 3D points

1: **function** ComputeAABB(S)
2: Find minimal and maximal value on all axes
3: $center \leftarrow$ new 3D point, value in each axis is equal to $min + (max - min)/2$
4: $hw \leftarrow$ new 3D point, value in each axis is equal to $(max - min)/2$
5: $aabb \leftarrow$ new AABB structure with properties $center, hw$
6: **return** $aabb$
7: **end function**

The procedure has complexity $\mathcal{O}(n)$, due to searching for extreme values on the axes, but building the AABB can be launched in preprocessing, meaning such complexity is acceptable. During movement of the object, the AABB has to be recomputed. There are two basic approaches:

- Creating tight AABB from scratch
- Creating approximate AABB from transformed AABB

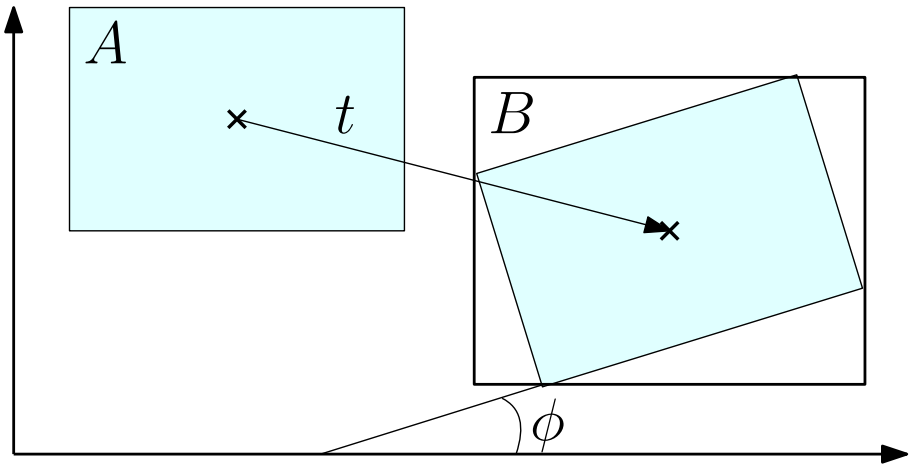

Figure 3.4: The box A is translated by vector t and rotated by angle ϕ. Then, approximate box B is computed from transformed A

First option is building the AABB using previous algorithm with new vertex coordinates. More common approach is building coarse AABB, which creates approximate

box instead of tight one. Such simplification make the intersection test less accurate, but faster. The procedure, which creates approximate AABB, is defined in following algorithm and its geometry interpretation in figure 3.4.

Algorithm 4 AABB updating

Precondition: *aabb* is AABB defined by its *center* and halfwidths hw, T is translation vector, R is rotation 3x3 matrix

```
1:  function UpdateAABB(aabb, T, M)
2:      B ← new AABB defined by center and hw
3:      i ← 0
4:      while i < 3 do
5:          B.center[i] ← T[i]
6:          B.hw[i] ← 0
7:          j ← 0
8:          while j < 3 do
9:              B.center[i] ← B.center[i] + R[i][j] * aabb.center[j]
10:             B.hw[i] ← B.hw[i] + R[i][j] * aabb.hw[j]
11:             j ← j + 1
12:         end while
13:         i ← i + 1
14:     end while
15:     return B
16: end function
```

The algorithm runs in constant time $\mathcal{O}(1)$, and is the most suitable for real-time applications. The intersection test between two AABBs is simple and fast. In each axis, the distance between centers and the sum of halfwidths is compared, and if objects doesn't intersect in one axis, they cannot collide.

Algorithm 5 AABB intersection test

Precondition: A, B are AABBs defined by *center* and hw

```
1:  function TestAABB(A, B)
2:      i ← 0
3:      while i < 3 do
4:          if abs(A.center[i] − B.center[i]) > (A.hw[i] + B.hw[i]) then
5:              return False
6:          end if
7:          i ← i + 1
8:      end while
9:      return True
10: end function
```

Summary		
Asymptotic complexity	creating	$\mathcal{O}(n)$
	updating	$\mathcal{O}(1)$ - approximate $\mathcal{O}(n)$ - accurate
	test	$\mathcal{O}(1)$
Memory requirements	center	3 * size of used data type
	halfwidths	3 * size of used data type

3.1.3. Oriented Bounding Boxes

The *Oriented bounding box* is also a six-sided box, but there are no restrictions to its faces. The box is represented similarly as AABB. The structure contains its *center* and $halfwidths$ in each axis. Because the OBB can be oriented arbitrary, it's also necessary to add three vectors, which describe the local axes.

Let S be set of points. As far as there is countless amount of OBBs, which overlap the S, some tool for finding the best suitable one is needed. Choosing OBB randomly can lead to great inaccuracies, as shown in figure 3.5.

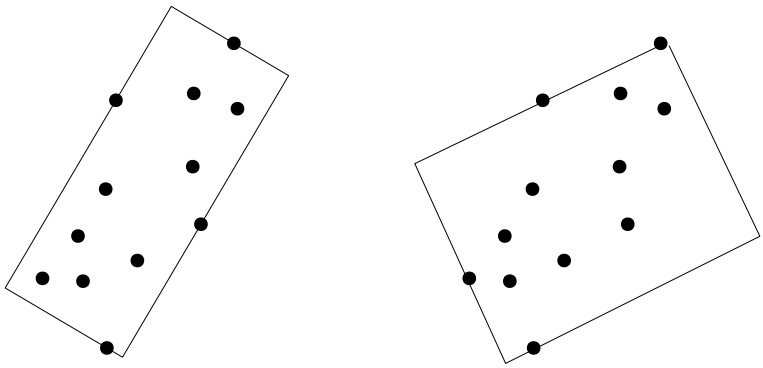

Figure 3.5: Difference between well and badly chosen OBB

Computing tight-fitting OBB is harder then it seems. Exact algorithm was presented by Joseph O'Rourke in 1985, but his algorithm has asymptotical complexity $\mathcal{O}(n^3)$. This property makes the algorithm too complex, slow and basically inapplicable. Simpler algorithms provide approximate solution of finding best-fitting OBB.

Let S be set of points in 3-dimensional space. First part of the algorithm is computing AABB for those points. There are three pairs of points lying on opposite sites of the box, those farthest apart are selected and vector v between them determines one axis and dimension of OBB. All other points are projected onto plane perpendicular to vector v. Then whole procedure repeats, but with lower dimension. On the plane 2-dimensional AABB is created and again pair of points with the same properties are selected. The second axis and dimension of the OBB is determined by this pair of projected points. Third axis can be computed as cross product of already known axes. Final step is projecting all

points into third axis and a pair of farthest points determines third dimension. The figure 3.6 displays 2D version of the algorithm.

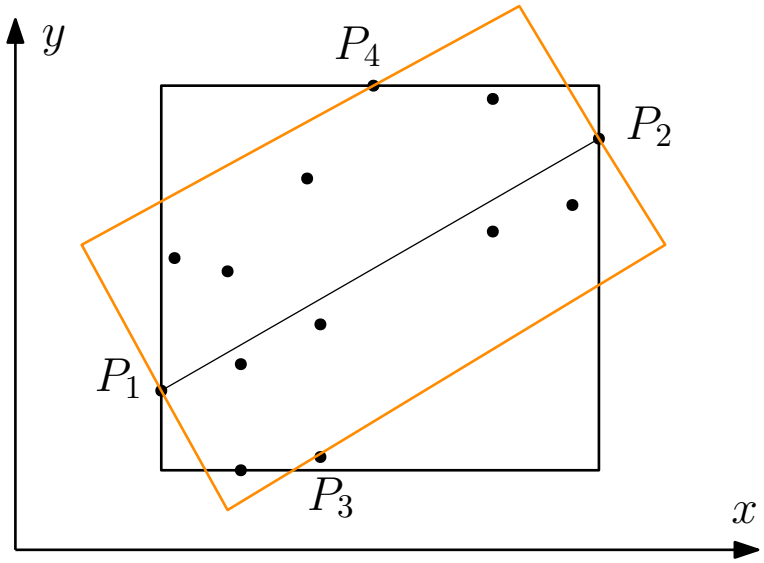

Figure 3.6: Approximate OBB computation (orange)

Instead of cheap intersection tests mentioned before, testing oriented bounding boxes is much more difficult and expensive. The test for OBBs intersection uses *separating axis theorem*, which is described in section 4.2. Basic principle is, if distance between OBB's centers is less then sum of their radii, with respect to some axis, then the boxes collide (figure 3.7).

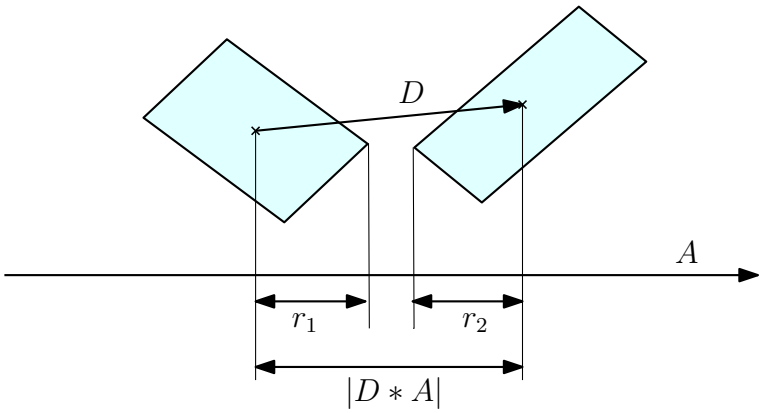

Figure 3.7: OBB itersection test

The inequality describing the problem is

$$|D * A| > r_1 + r_2,$$

where D is vector between centers, A is actual axis and r_1, r_2 are radii of tested OBBs. Determining which axes have to be tested is the last step to completing collision test. There are 15 axes, which need to be checked [6], which means solving 15 inequalities. Table in [6] contains values, which needs to be substituted into general inequality. If simple inequality is met, test is terminated and reports objects as non-intersecting. The

simplification may be performing only the first six tests (in order given by table in [6]), while other nine tests determine only 15% of non-intersecting cases.

Summary		
Asymptotic complexity	creating	$\mathcal{O}(n^3)$ - accurate $\mathcal{O}(n)$ - approximate
	updating	$\mathcal{O}(1)$
	test	$\mathcal{O}(1)$
Memory requirements	center-halfwidths- local axes representation	15 * size of used data type

3.2. Bounding Volume Hierarchies

Tests on bounding volumes are much simpler than on original meshes and performance is improved drastically. Problem is that all pairwise tests are performed at the same time, regardless of position of tested objects. To overcome this problem, *Bounding volume hierarchies* (BVH) are used. This technique divides the space into smaller parts and separates objects, which have no chance to be intersecting. The example of objects, which are creating hierarchy, is in figure 3.8. Each box overlaps two smaller boxes and whole hierarchy can be illustrated as a tree. The corresponding tree is shown on the right.

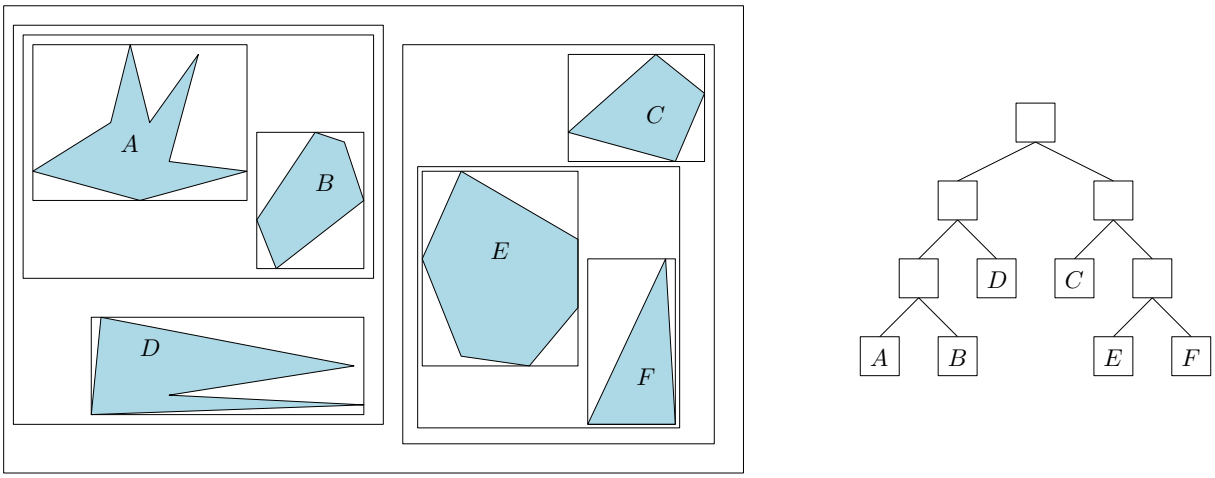

Figure 3.8: BVH of six objects

There are some defined properties[14], which BVH should satisfy.

- Tight bounding volumes
- Simple shapes
- Minimal overlap between sibling nodes
- Tree structure

- Balanced hierarchy (the example is in figure)

Some of these properties don't have to be fulfilled exactly, but they should be a main guide to completing the hierarchy. The optimal tree is built with respect to some strategy. These strategies are top-down, bottom-up and insertion. Most popular is the first strategy, top-down. It's recursive procedure and in each iteration it splits input set of primitives into two subsets, until stopping criterion is met. These criteria can be number of objects in tree's nodes, the volume of the bounding volume or depth of the tree. The number of objects in single node is mostly one, but can be chosen arbitrary. It also depends on application and complexity of the objects, because while the BVH is applied on an object with thousands of primitives and the number of objects in tree's leaves is set to one, the depth of that tree will raise rapidly.

Algorithm 6 Top-down construction

Precondition: *objects* is set of objects or primitives, *node* is defined by its bounding volume *bv*, array of objects *obj* and left/right descendant $ldes, rdes$

```
1: function TopDown(node, objects)
2:     node.bv ← bounding volume of objects
3:     if size of objects < maximal count of objects per leaf then
4:         node.objects ← objects
5:     else
6:         Split objects into two parts part1, part2
7:         TopDown(node.ldes, part1)
8:         TopDown(node.rdes, part2)
9:     end if
10: end function
```

When splitting, some primitives (like triangles) can be cut through by that plane. Two solutions are presented in [7]:

- Cut the primitive into two parts and both parts will represent new primitives. It prevents children from overlapping, but the total number of primitives can grow.
- Use primitives's centroid to determine which bounding volume will be assigned to. That solution doesn't increase number of primitives, but can cause small overlap of bounding volumes.

As mentioned, the splitting can be guided by some strategies. The basic splitting strategies are: [8][14]

- Minimize the sum of the volumes of the children - creates children as tight as possible
- Minimize the largest child - creates children of equal size
- Minimize the intersection area of children

- Maximize the separation - decreases cost of intersection test
- Combinations

While trees are built, the collision detection tests can be launched on tree pairs. To do so, it is necessary to have some rules for descending through the tree, from the top to the bottom. There are two basic approaches:

- uninformed methods
- informed methods

The uninformed methods use only hierarchy structure to determine next node. It includes *breath-first* (BFS) and *depth-first* searches (DFS). The BFS process nodes gradually, according to their depth from zero to maximal tree depth. All nodes of each depth are processed before continuing deeper into the tree. Opposite approach is DFS, which processes node's descendants before other nodes of the same depth. Both searches are illustrated in figure 3.9.

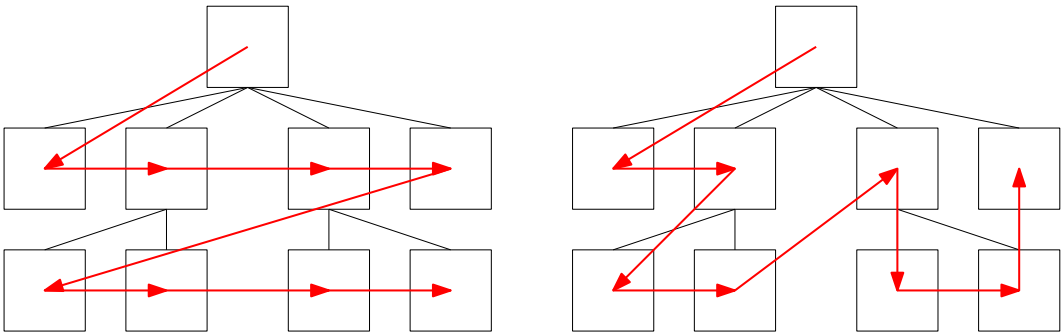

Figure 3.9: Breath-first search (left) and depth-first search(right)

Informed methods are more advanced. They examine node data to determine the next node, while the most popular is *best-first search*. It holds a queue of nodes and expands only the best current node (according to given criterion). The expanded children are added to the queue and the process repeats until stopping criteria are met. Such search is illustrated in figure 3.10.

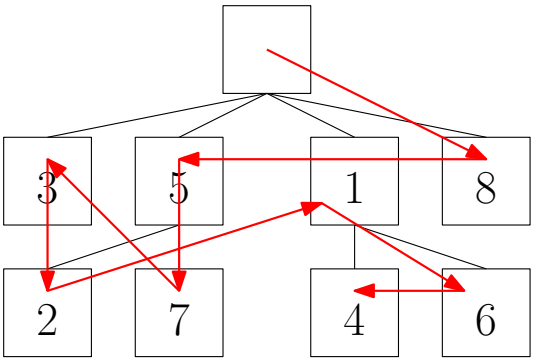

Figure 3.10: Best-first search

Last issue is determining a way, by which the tested hierarchies will be tested. That means finding a rule determining which hierarchy should descend during test. Let A, B be BVHs. Possible descent rules are[7]:

- Descend A before B (or vice versa) - descent fully into the leaves of the first hierarchy before descending into the second one. This option is not recommended, while hierarchy's root is fully inside other root.
- Descend hierarchies according to its volumes - removes problems of first rule, larger hierarchy is descended
- Descend both hierarchies simultaneously - doesn't need any computations like previous rule, but isn't so efficient.
- Combinations

Previous rules can be used for building large variety of intersection tests. There isn't any best way for designing the test, suitability fully depends on the object shape, complexity and mutual position as wall as type and quality of bounding volume hierarchy. Following algorithm describes depth-first search algorithm with "descend A before B rule":

Algorithm 7 BVH collision test

Precondition: a, b are BVHs with two descendants $ldesc, rdesc$

```
1: function BVHIntersectionTest(a, b)
2:     if no overlap between a, b then
3:         return False
4:     end if
5:     if a and b are leaves then
6:         return result of some collision test on primitives
7:     else
8:         if a is not leaf then
9:             BVHIntersectionTest(a.ldesc, b)
10:            BVHIntersectionTest(a.rdesc, b)
11:        else
12:            BVHIntersectionTest(a, b.ldesc)
13:            BVHIntersectionTest(a, b.rdesc)
14:        end if
15:    end if
16: end function
```

3.2.1. AABB trees

The most common and popular BVH type is a binary AABB tree and figure 3.8 shown exactly this one. For collision detection, two possible concepts exist. A *static AABB*

tree is computed only once and during the process the tree isn't changing. This concept is implemented in library CGAL (section 5.3). the process of building a tree is the same, as described before. More interesting is *dynamic AABB tree*, which can be used with moving objects. The most important precondition is balancing the tree as much as possible. Unbalanced trees can lead to decreasing the performance of computations performed on the tree. The most expensive operation is updating the tree. As far as all objects in the scene move, the tree can be either built from scratch or updated by special algorithm. Building the tree from scratch is rather naive and expensive way to obtain updated tree.

The same result can be acquired by fat AABB approach. Each object is bounded by AABB, which is bigger than tight-fitting one. The size is often defined to make object able to rotate in all direction within the AABB and not smaller than 105% of original size. In each step, the tight-fitting AABB is checked, if it is still contained in the fat AABB. If not, the object is removed from the tree and inserted again, represented by updated AABB.

3.3. Spatial Partitioning

As said before, pairwise testing of polyhedra is expensive. The solution of this problem, apart from BVH, can be restricting those tests to object, which are close enough. That is the idea, which was vital for developing *Spatial Partitioning*. The main idea is dividing space into smaller regions, meaning two objects can intersect if and only if they overlap the same region.

3.3.1. Grids

Basic and the most natural option is dividing space with a grid with constant cell size. Each object in the space is associated with all cells it overlaps and collision between objects A, B is possible if and only if they are associated with the same cell. Using uniformly sized grid is advantageous for two reasons. It is easy to find each cell position just with indices and also finding neighbouring cells is trivial. The only problem is determining the cell size, which is the most suitable for each problem. [7] describes four problems, which are connected with this problem, all of them are in figure 3.11.

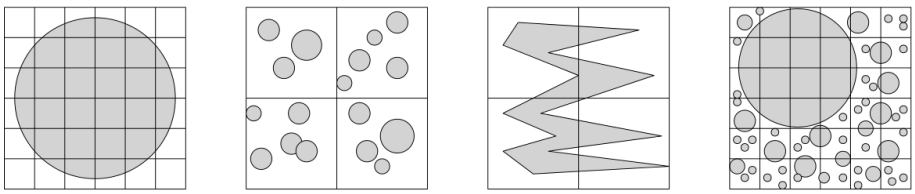

Figure 3.11: Problematic cell sizes.

- The first picture describes *fine* grid. This situation results in large number of cells associated with such large objects. Expensive recomputing of the cells when object moves and too many intersection tests have to be performed.
- The opposite problem is *coarse* grid. As can be seen in the second picture, too many objects are associated with each cell. It results in large amount of pairwise tests and benefit of spatial partitioning is minimal.
- Another case of problematic grid is in the third picture. The size is well chosen, but the object is too complex. There should be used some simplistic method like dividing the object (and then smaller cells) or bounding the object (with reduction of accuracy).
- The last issue is combination of the first and second ones. This can be solved by *Hierarchical grids*

Cells should be large enough to cover the largest objects in any rotation. There exists $n^{\frac{1}{3}}$ rule[7]. The space is divided into $k \times k \times k$ cells, where $k = n^{\frac{1}{3}}$ and n is the number of objects.

When the suitable cell size is set, the object can overlap only neighboring cells. Let's assume all objects are associated with only one cell, which contains object's AABB's minimal corner. The approach through AABBs helps to design cheap intersection algorithm. The given algorithm is for 2-dimensional case, but can be easily extended to 3D. Whole situation can be seen in figure 3.12. In the worst case, the number of tested cells is only four (eight in 3D), but ideally it can stop after one test for each object.

Algorithm 8 Intersection test of two objects represented by AABB minimal corner

```
1:  function intersectionUniformGrid
2:      test representative cell
3:      if object overlaps right cell border then
4:          test right cell
5:      end if
6:      if object overlaps lower cell border then
7:          test lower cell
8:          if object overlaps right cell border then
9:              test lower-right cell
10:         end if
11:     end if
12: end function
```

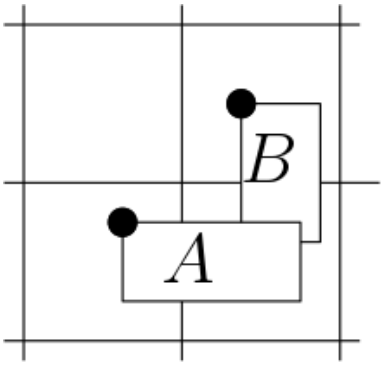

Figure 3.12: Objects in grid represented by AABB's minimal corners

3.3.2. Trees

Another approach is creating a space partitioning tree. Firstly the space is divided into regions and these regions are recursively divided until the stopping criteria are met. The section describes basic type, *octree*.

The octree is based on axis-aligned space partitioning. The main feature is that each node of the tree has eight children, called octants. Two steps of dividing are in figure 3.13.

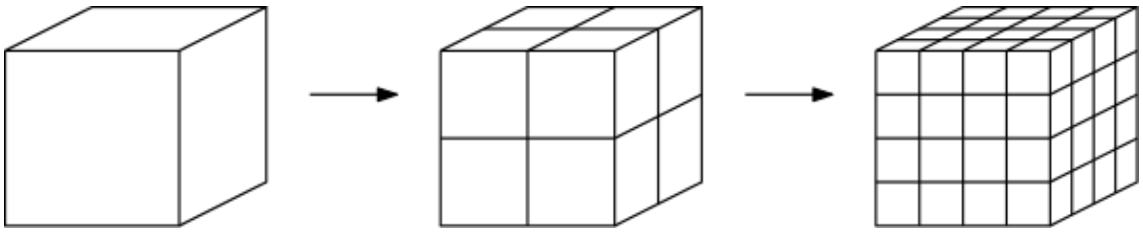

Figure 3.13: Octree

The root represents entire space and is axis-aligned cube. Children nodes are acquired by dividing the cube in half in all three axes. Each node of the tree has some attributes. *Center* of the octree is the point in the middle of the cube (where diagonals meet). It can be easily computed from parent's center, if *halfwidth* of the cube is known. Octree also holds two arrays. First array contains *pointers to children nodes* and second contains *pointers to objects*, which the node overlaps. Following algorithm describes recursive building of the octree:

Algorithm 9 Octree

Precondition: *center* is point in center of whole space, *hw* represents halfwidth of this space, *depth* is maximal depth of octree

```
1: function OCTREE(center, hw, depth)
2:     if depth < 0 then
3:         return
4:     end if
5:     n ← new node with properties center, hw
6:     i ← 0
7:     while i < 8 do
8:         c ← ith children's center
9:         children of n ← Octree(c, 0.5 * hw, depth − 1)
10:        i ← i + 1
11:    end while
12:    return n
13: end function
```

When the octree is built, objects can be inserted into. The insertion procedure has to determine minimal node overlaping whole object's bounding sphere and link object to this node. It's also recursive process, where two options are possible. If the object's bounding sphere fits into one of child nodes, whole procedure continues on this specific node. If the object overlaps more child nodes, then is assigned to actual node. Algorithm 10 presumes object representation as bounding sphere with defined center and radius. The octree is built by previous algorithm.

In this stage, the octree is built and objects are linked to corresponding nodes. Let's remind that each object is linked only with one node. Final procedure searches the octree and tests possible collisions. While processing a node, linked objects are tested with each other, as well with objects in descendant nodes. The iterative test uses static array of ancestors named *ancestors*, which is common for all iterations.

Algorithm 10 Insert object into octree

Precondition: *node* is root of octree built by previous algorithm, *object* is defined by its bounding sphere

```
1: function InsertToOctree(node, object)
2:     fits ← false
3:     i ← 0
4:     while i < 8 do
5:         if object fits into ith actual node's child then
6:             index ← i
7:             fits ← true
8:             Break
9:         end if
10:        i ← i + 1
11:    end while
12:    if fits == True then
13:        InsertToOctree(node's ith child, object)
14:    else
15:        add object to node's object array
16:    end if
17: end function
```

Algorithm 11 Test all collisions inside octree

Precondition: *tree* is a octree build by previous algorithm, *ancestors* is empty static array of (sub)octrees

```
1: function TestOctree(tree)
2:     Add tree to ancestors
3:     for i = 0 to size of ancestors do
4:         for each object o1 in ancestors[i] do
5:             for each object o2 in tree do
6:                 if o1 is equal to o2 then
7:                     break
8:                 end if
9:                 Perform some collision test between o1 and o2
10:            end for
11:        end for
12:    end for
13:    for i = 0 to 8 do
14:        TestOctree(tree.children[i])
15:    end for
16: end function
```

4. Convex hull-based methods

The best-suitable objects for fast collision detection are convex objects. They have important properties which help making fast and robust collision tests. The most important property is that local minimum is equal to global minimum, than many well-known, fast and easy to implement algorithms can be used, such as simple hill-climbing. The next advantage is existence of plane, which separates two non-intersecting objects. These properties help to design efficient algorithms to test intersection. In first section the problem of convexity is discussed and then specific convexity-based methods are described. These methods are much more accurate then methods mentioned before, they are absolutely general and exact, but precondition of convexity is strongly restrictive. The main reason is that most objects tested by these algorithms are non-convex, so some way to overcome this problem is needed. The solution is bounding the object with some convex polyhedra. This solution was presented before, in chapter *Bounding Volumes*, but there will be some difference. Instead of bounding to primitive object, the *Convex hull* will be created. Algorithm creating the convex hull is presented in next section. Another approach is dividing object into smaller pieces, which is called *Convex decomposition*. Some intersection testing algorithms can be used either on convex and non-convex objects, but in the first case they will be much faster and it is the most necessary feature when used in real-time. The example is Gilbert-Johnson-Keerthi algorithm or Chung Wang algorithm.

4.1. Convex hull algorithm

The convex hull is defined as the smallest polyhedron, which encloses given set of points. The points can be either on surface and in the interior. In this section algorithm, which is implemented in attached application, is described. The algorithm not only creates convex hull, but its outputs are also lists of neighbors for each vertex. These lists help making intersection test faster. The algorithm 12 presents whole computing process, then all parts will be described in detail.

The first step is building initial tetrahedron from given point set. The simplest solution is choosing four points randomly, which will work. The problem is the tetrahedron's volume. Randomly created tetrahedron doesn't have to overlap enough points and iterative process can be slower. To make iterative phase faster, the initial tetrahedron should be as large as possible. The first step is finding minimal and maximal points in each axis. The most distant pair makes one edge and point most distant to this edge completes the triangle. The most distant point from this triangle is the last searched vertex.
When the tetrahedron is built, assignment of points is performed. The point is assigned to face if and only if the face is visible from that point. If more faces can be seen, the point is assigned randomly. It can be proven, that random assignments don't effect algorithm's performance. In figure 4.1 a 2D version of this step is shown. Initial polygon is represented by $x_{min}, y_{min}, x_{max}$. Points inside the polygon aren't assigned, because they

cannot see sides from outside, so only four points are assigned to side x_{min}, x_{max} and single point to side x_{min}, y_{min}.

Algorithm 12 Convex hull

Precondition: S is set of 3D points

```
1: function createConvexHull(S)
2:     Build initial tetrahedron T using four points S
3:     Assign each point from S to face of T, which is visible from the point
4:     f ← all faces with assigned points
5:     while f is not empty do
6:         Find farthest assigned point eye from first face of f
7:         Find horizon of T visible from eye
8:         Add eye to T by connecting all vertices on the horizon with eye
9:         Again assign points to faces created in previous step
10:        Add faces with some assigned points to end of f
11:        Remove first face of f
12:    end while
13: end function
```

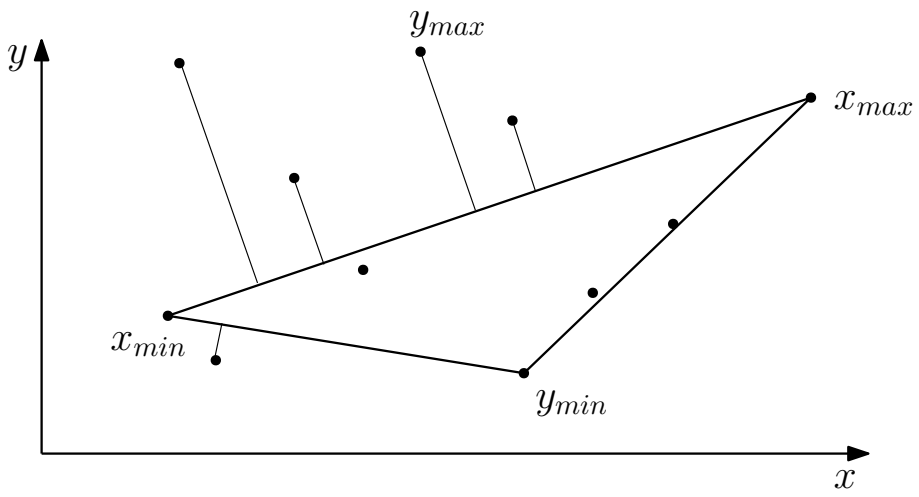

Figure 4.1: Computing maximal initial polygon

When faces with some assigned points are listed, main iterative loop begins. The faces are processed one by one, then removed from list and newly created faces are added to the end of the list (if have assigned points). Let F be face with set of assigned points S. The farthest point from the face is marked as *eye* and will be added to actual convex polyhedron. In 3D finding polyhedron's horizon isn't simple and the implemented application uses following observation. While looking on the polyhedron from *eye*, triangular faces are visible, such as in figure 4.2. Edges lying on the horizon belong to only one triangle, instead of others, which belong to pair of triangles. Vertices on horizon are connected with the *eye* and new faces are contructed. Final step is reassigning point set, which belonged

to processed face. The procedure is the same as assigning in the beginning. Then new faces with some assigned points are listed to unprocessed faces set.

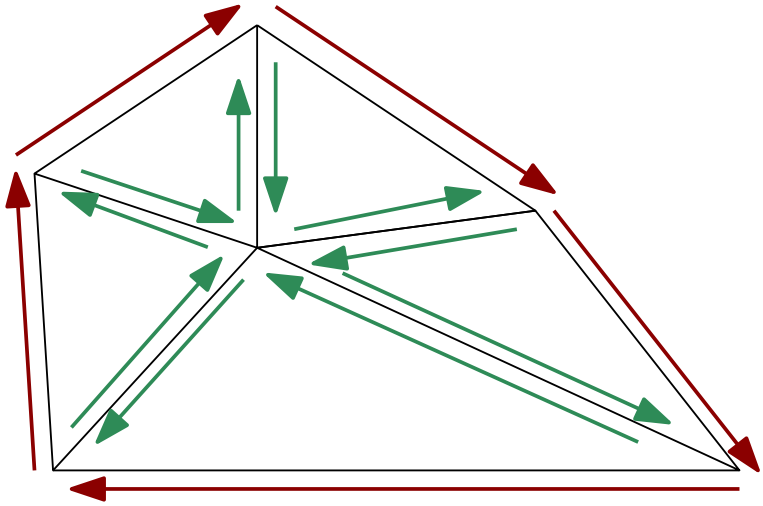

Figure 4.2: The horizon of polyhedron

4.2. Separating axis theorem

The *Separating Axis Theorem* (SAT) is a method, which determines intersecting objects. Basically, it searches for gaps between objects. The analogy is a flashlight shining on two shapes from different directions, so the shadow appears on the wall behind objects. If we can find a direction, where the shadows will be separated by gap, the objects are not intersecting. Figure 4.3 represents two different states. On the left picture the gap doesn't exist, but when the direction changes, the gap appears.

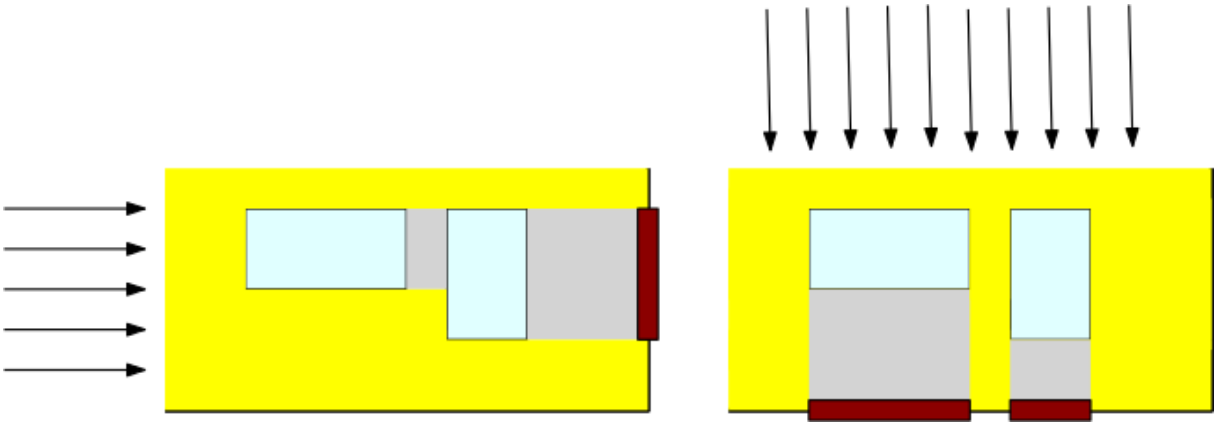

Figure 4.3: Flashlight analogy for SAT

Naturally, checking every angle and searching for a gap would be very expensive. Due to nature of the polygons, only countable amount of angles have to be checked. Those directions correspond to object faces and whole procedure consists of these steps:

Algorithm 13 SAT

```
1: function SAT intersection test(A, B)
2:     s ← all facets of A
3:     while s not empty do
4:         t ← first facet of s
5:         axis ← normal vector of t
6:         Project both objects onto the axis
7:         if Projections don't intersect then
8:             return False
9:         end if
10:    end while
11:    return True
12: end function
```

The SAT has strong restriction on convex objects. Figure 4.4 shows a situation, when nonintersecting objects can be marked as intersecting due to violating convexity condition. In that case, convex hulls of objects are tested implicitly.

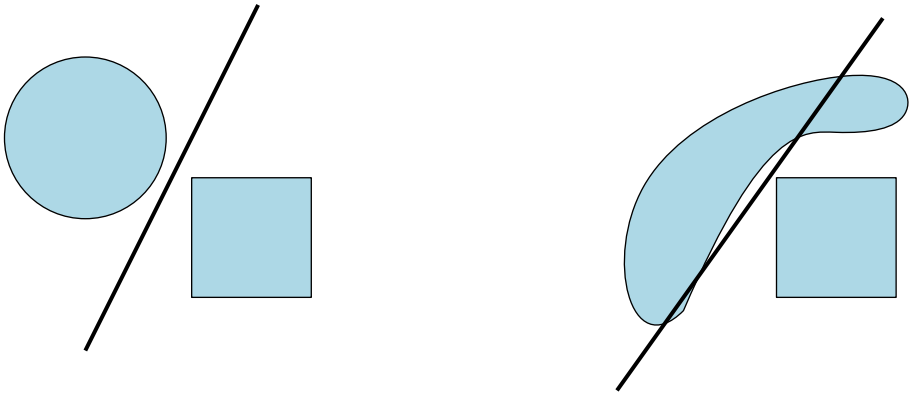

Figure 4.4: Convex and non-convex objects in SAT intersection test

4.3. Gilbert-Johnson-Keerthi

The *Gilbert-Johnson-Keerthi* (GJK) algorithm was developed for quick intersection tests, which will return not only true/false, but also minimal distance between tested objects. The main goal si computing their Minkowski difference and find its point of minimum norm P (point closest to origin). If P is origin itself, the objects are surely intersecting, and in other case distance between the objects is equal to $\|P\|$. However, explicit computing of Minkowski difference is expensive operation, so algorithm uses simplex-based implicit computations. It computes only the difference points by *support mapping* function. A support mapping function $s_A(\mathbf{d})$ computes point from object A, which is the most extreme in direction $\mathbf{d}$. Then, for Minkowski difference $C = A \ominus B$ the function is defined as follows:

$$s_{A \ominus B}(\mathbf{d}) = s_A(\mathbf{d}) - s_B(-\mathbf{d})$$

The algorithm uses one more finding known as *Carathéodory's theorem*. It says that each point from convex object $\in R^n$ can be expressed as up to $n+1$ points, so the algorithm needs to keep only those points in the worst case (four in 3D). If the simplex contains the origin, the objects are intersecting and algorithm stops, otherwise new simplex, containing points closer to origin, is computed. Whole algorithm is described in following procedure and figure 4.5.

Algorithm 14 Gilbert-Keerthi-Johnson algorithm

Precondition: A, B - convex hulls of two objects

1: **function** GJK(A, B)
2: Initialize the simplex set S from $A \ominus B$
3: Find point P closest to origin
4: If P is equal to origin, stop the test and return TRUE
5: Remove points $\in S$, which are not necessary for expressing P
6: Compute $M = s_{A\ominus B}(-P)$, where $-P$ is vector $(\vec{0} - P)$
7: If P is more extremal (or equal) than M in direction $-P$, stop the test and return FALSE
8: Add M to simplex set S and go to line 3
9: **end function**

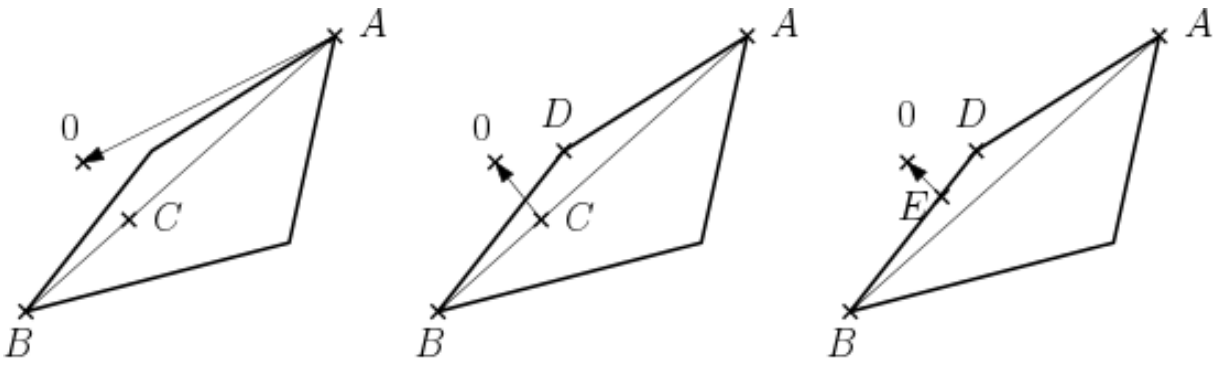

Figure 4.5: GJK algorithm

Let initial simplex S be $\{A\}$. Point closest to origin is A, so support mapping function searches in direction $-A$ and finds point B. B is added to simplex and procedure continues. New point of minimal norm is C and $s_{A\ominus B}(-\mathbf{C}) = D$. D is added to S. Point of minimal norm on triangle ABD is E, so it is not necessary to hold point A in memory and S is reduced to $\{B, D\}$. Support mapping function discovers there is no more extremal point in direction $-E$ than E itself, so the algorithm returns *false*. Minimal distance between objects is then equal to $\|E\|$.

As said before, GJK algorithm can also be used on non-convex objects and the algorithm process will be exactly the same with one exception. The support mapping function has to find the most extreme point in given direction **d**. The rate of "extremeness" can be

measured as a dot product $(P * d)$, where P is a given point and $\mathbf{d}$ search direction. When this function runs on non-convex object, all its vertices have to be tested to find the most extreme one. Time complexity is $\mathcal{O}(n)$, linear, which is unsuitable for real-time operations as far as the objects consist of thousands vertices and the test has to be performed many times per second. That's why convexity is essential for real-time applications. If the object is defined by vertices lying on convex hull and each vertex knows its neighbors, whole procedure can be handled by simple hill climbing. The largest issue of hill climbing, jamming in local minimum, is impossible due to convex hull's properties.

Searching for extreme vertices can be improved by adding some artificial edges, as shown in figure 4.6. Each vertex will have more neighbors to test, but convergence to resulting vertex will be much faster. The situation in the figure 4.6 describes searching in direction $\mathbf{d}$ from initial point A. Basic approach is to test all five vertices on the way to the most extreme point. If artificial edge e is added, only three points need to be tested and the process speeds up.

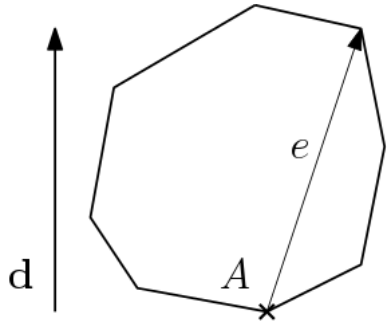

Figure 4.6: Faster hill climbing

The last issue is searching for point of minimal norm P_{min} in simplex, which means point on the simplex closest to origin. As far as the simplex can consist of up to four points (in 3D), different test have to be performed. Let S be simplex and $|S| = n$, then following cases have to be considered:

- $n = 1$: trivial case, point itself has minimal norm
- $n = 2$: simplex is a line segment. P_{min} can be one of outer points. This point remains in simplex set, the second one is erased. If P_{min} is some point between outer points, both outer points remain in simplex set.
- $n = 3$: simplex is a triangle. There are three possible cases. If P_{min} is one of the vertices, this vertex remains in the simplex set and others are erased. If P_{min} lies on a line segment, points representing the segment remain in the simplex set. If P_{min}lies inside the triangle, no point is erased.
- $n = 4$: simplex is a tetrahedron. The situation is similar to previous with one difference. If P_{min} lies on the face of the tetrahedron, only points representing this face remain is the simplex set and no point is erased if P_{min} lies inside the tetrahedron (not on surface). In that case is also P_{min} equal to origin.

4.4. Chung Wang

The *Chung Wang* algorithm was designed for testing pairs of polyhedra. It consists of two subalgorithms. The first one searches for separating vector in iterations, starting from some randomly chosen vector, called candidate vector. In each iteration the vector is updated and tested if the non-intersecting criterion is met. The second subalgorithm checks intersection criterion and stops the first algorithm if criterion is met.
The main algorithm computes new possible separating vector $\mathbf{s}_i$ and a point of each object, which is the most extreme in given way. Like the GJK, this algorithm uses support function, defined above, for obtaining those points. Then the comparison of points leads to possible no intersection report.

Algorithm 15 Chung-Wang algorithm, main loop[7]

Precondition: A, B - convex hulls of two objects

1: **function** CW(A, B)
2: $\mathbf{s}_0 \leftarrow$ random candidate vector
3: $i \leftarrow 0$
4: Point $a_i \leftarrow$ supportFunction(A, $\mathbf{s}_i$)
5: Point $b_i \leftarrow$ supportFunction(B, $-\mathbf{s}_i$)
6: **if** $(a_i * \mathbf{s}_i) < (b_i * -\mathbf{s}_i)$ **then**
7: **return** False
8: **end if**
9: $\mathbf{r}_i \leftarrow (a_i - b_i)/\|a_i - b_i\|$
10: $\mathbf{s}_{i+1} \leftarrow \mathbf{s}_i - 2(\mathbf{r}_i * \mathbf{s}_i)\mathbf{r}_i$
11: Go to line 4.
12: **end function**

5. Existiong solutions and libraries

5.1. V-HACD

Possible solution of approximate convex decomposition has been presented in [11] in 2009. It uses hierarchical segmentation approach and the dual graph. Let S be a mesh in $\mathbb{R}^3$, $\psi = \{v_1, v_2, ..., v_V\}$ its vertices and $\theta = \{t_1, t_2, ..., t_T\}$ the set of its triangles, where V is number of vertices and T number of triangles. Then the dual graph S^* of the mesh S is graph satisfying these two conditions:

- vertices of S^* correspond to triangles θ of S
- edge between two vertices of S^* exists if and only if corresponding triangles share an edge

The main phase of the algorithm is applying half-edge collapse[11] operation. The process is presented in figure 5.1. If applied to an edge (x, y), all incident edges to y are connected to x and y is removed.

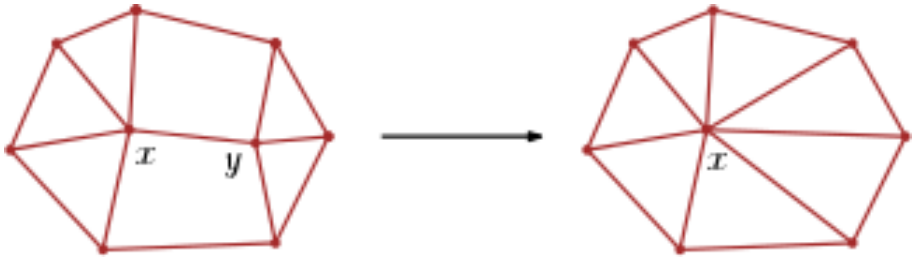

Figure 5.1: Half-edge collapse operation

To determine which vertices should be processed in each iteration, it is necessary to define *concavity* $C(S)$ of a mesh S. The concavity indicates how far the object is from its ideal convex hull, which has a concavity zero. In V-HACD it is defined as follows:

$$C(S) = arg\,max_{M \in S} \|M - P(M)\|,$$

where the point M lies on convex hull of S and $P(M)$ is its projection on convex hull in direction of normal vector.

It is also necessary to guide the process by cost function, which takes into account concavity and aspect ratio of pair of triangles. The pair with the lowest cost function is processed in each iteration. For detailed mathematical description please visit [11]. The work of the algorithm is described in figure 5.2.

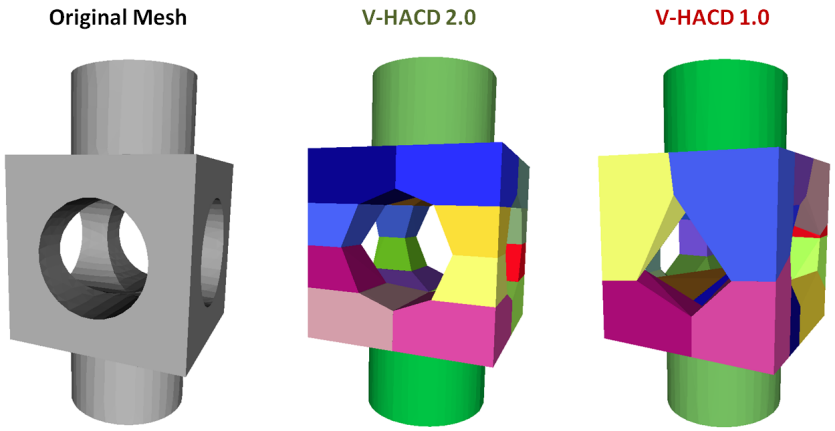

Figure 5.2: V-HACD output[10]

The libraries can be downloaded from [10] in 2.0 version under BSD license. Until now the libraries have been used in different applications:

- Unreal Engine 4
- Leadwerks
- 3DEXCITE
- DBPro

5.2. Bullet

Bullet Physics is a complete solution for use in games and robotic simulations under ZLib license which makes library free for commercial use. The library is written in open source C++ code for use on all platforms. It contains methods for discrete and continuous collision detection on both convex and non-convex objects, soft and rigid body dynamics. Although the library is a complex solution, only needed parts can be used, such as collision detection algorithms. Whole architecture of the library is presented in figure 5.3.

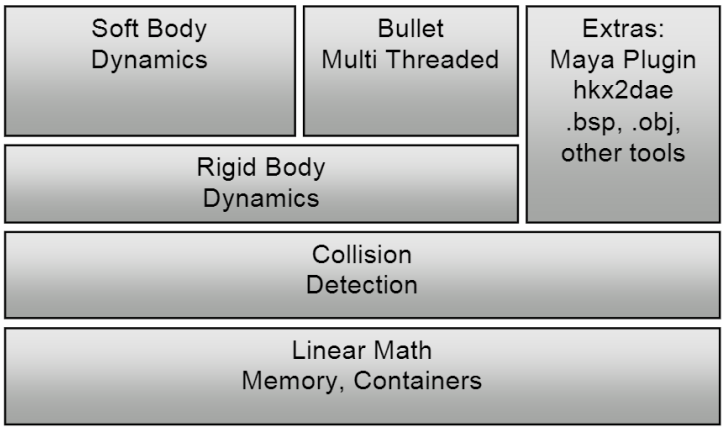

Figure 5.3: Bullet Physics library architecture[2]

Bullet library can choose from large amount of collision objects to provide best performance and quality. It is also possible to add own shapes that suit special purpose. In Bullet's manuals, the diagram is presented, which can help choosing the shape.

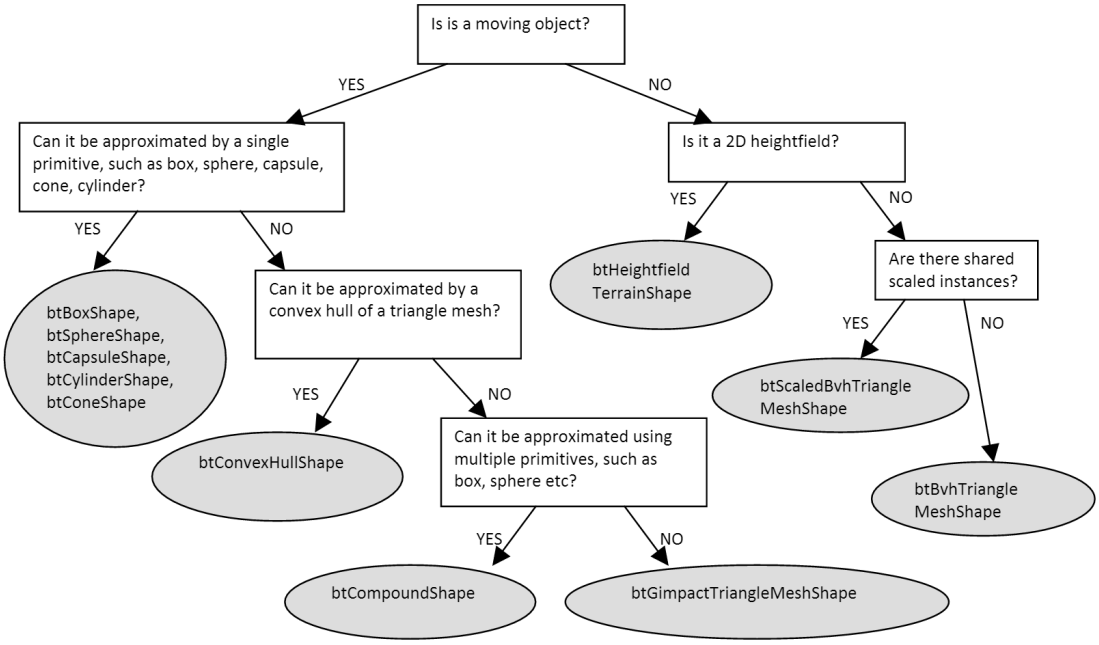

Figure 5.4: Shapes diagram[2]

There are many different collision tests prepared for different situations. This leads to rapid increase of performance, while only the most suitable tests are launched. The following table represents choosing specific test.

	box	sphere	convex,cylinder cone,capsule	compound	triangle mesh
box	boxbox	spherebox	gjk	compound	concaveconvex
sphere	spherebox	spheresphere	gjk	compound	concaveconvex
convex, cylinder, cone, capsule	gjk	gjk	gjk or SAT	compound	concaveconvex
compound	compound	compound	compound	compound	compound
triangle mesh	concaveconvex	concaveconvex	concaveconvex	compound	gimpact

Figure 5.5: Table for determining launched test[2]

The library can be downloaded from the webpage and built by CMake. Also whole detailed documentation as well as quickstart documentation can be found in [2].

5.3. CGAL

Next important library is *CGAL* (The Computational Geometry Algorithms Library). It provides huge spectrum of efficient geometric algorithms as C++ libraries in different packages.

- Arithmetic and Algebra
- Convex Hull algorithms
- Triangulations
- Mesh Generation
- Interpolation
- and many other [5]

CGAL also contains package for intersection and distance computing using AABB hierarchy structure. The main component is class *AABB_tree*, which represents constructed tree from input data. In library's documentation benchmarks can be find and will be shown in the last chapter.

The library is provided with dual licensing- open source (LGPL/GPL) and commercial one for companies and is used in large variety of interests: urban modeling, astronomy, computer graphics, image processing or games.

5.4. Other libraries

On the internet, many libraries can be found, but there are often some issues. Those libraries were created about a decade ago and are not maintained and updated anymore. Also adding them into project created in modern IDEs can cause problems and they can be substituted by libraries mentioned before.
Those libraries are:

- ColDet3D (or Claudette)
- OZCollide
- OPCODE
- V-Clip
- V-COLLIDE

6. Implementation

One part of this master's thesis is implementation of chosen algorithms. The aim of this part is to make own code, which can be used to make new libraries without limitations like licenses. The program is written in C++ language, with the help of OpenGL library, which is described in next section.
The program should satisfy following conditions:

- Input data will be represented as STL files
- The resulting program can be used in real time applications
- Output will be only TRUE/FALSE (intersecting/ not intersecting)

Let's go briefly through these conditions. The program doesn't have to be able to read all possible data files, but is dedicated only to SLT files. Using the program in real time applications predetermines used algorithms. As many operations as possible should be done in preprocessing (during data loading) to make real-time part of the program quicker. In practice that means doing operations like removing redundant data or creating needed structures (bounding objects, convex hulls etc.). The discussion about suitable algorithms is in separated section. The boolean output of the program makes real time part quicker, because it's not necessary to compute exact distance between objects, which is the most expensive operation.

6.1. C++

The instrument for creating the program is C++ programming language developed by Bjarne Stroustrup. It is imperative, generic, object oriented and compiled language, which has been developed for large variety of problems. It is direct successor of C language (C++ means incremented C). In present it is one of the most popular languages utilized with application software, client-server applications, embedded firmware or drivers.
The STD library is the set of classes and functions, which are part of the C++ standard itself, and are declared in *std* namespace. The library contains many types of containers, objects, functions to work with those objects or streams.
Drawing the data through OpenGL is quite complex problem without using other libraries so next library is GLFW. It provides basic methods, like creating a window with OpenGL context, event handling (key, mouse or joystick issues). It can be downloaded from [4] as source codes of pre-compiled binaries. It is recommended to compile the library manually, because pre-compiled libraries don't have to work properly on some operating systems.
The GLEW (*OpenGL Extension Wrangler Library*) is the last library, which can simplify working with OpenGL. The OpenGL standard is implemented on the graphic card by its manufacturer. Without GLEW, a developer has to retrieve the location of needed functions himself, which is a lot of additional work. The GLEW solves that issue and makes programming more simple and clear. The library can be downloaded from [15].

6.2. STL format

The data input of program should be .stl (*Stereolitography*) file. STL files carry the data as surface geometry of objects without any additional information like color or texture. There are two types of STL file - binary and ASCII, while binary file is more common. In both types the file carry unstructured triangulated surface by three vertices and normal per each simple triangle. All these object must be described in three-dimensional space and the values must be positive.
Naturally, a *ASCII STL* file has strict structure[13]:

solid *name*

(foreach triangle)
facet normal $n_x n_y n_z$
  outer loop
    vertex $v1_x v1_y v1_z$
    vertex $v2_x v2_y v2_z$
    vertex $v3_x v3_y v3_z$
  endloop
endfacet

endsolid *name*

Numbers v_i and n_i are floating-point numbers in format sign-mantissa-e-sign-exponent, for example 5.680200e-006 and $v_i \geq 0$.
More common type of STL file is binary STL file, which structure is as follows[13]:

UINT8[80] - describtion
UINT32 - number of triangles

(foreach triangle)
REAL32[3] - normal vector
REAL32[3] - vertex
REAL32[3] - vertex
REAL32[3] - vertex

UINT16 - attribute byte count

After writing the last triangle, the file ends. The attribute byte count is zero in standard STL format, because no additional information is added and most of the software cannot use it. The representation of REAL32 numbers is defined by IEEE floating-point number.

6.3. OpenGL

OpenGL is a standard developed and maintained by Khronos Group. It specifies graphics API and contains large variety of functions, which can manipulate graphics processing unit (GPU). The goal is to achieve hardware-accelerated rendering. In following sections, important features of OpenGL will be described, as well as representing C++ code.

6.3.1. Architecture

Rendering of objects through OpenGL is quite a complex procedure, which contains many steps. The object data are stored in computer memory and OpenGL uses the *rendering pipeline* to display those data on the screen. The pipeline is shown in figure 6.1.

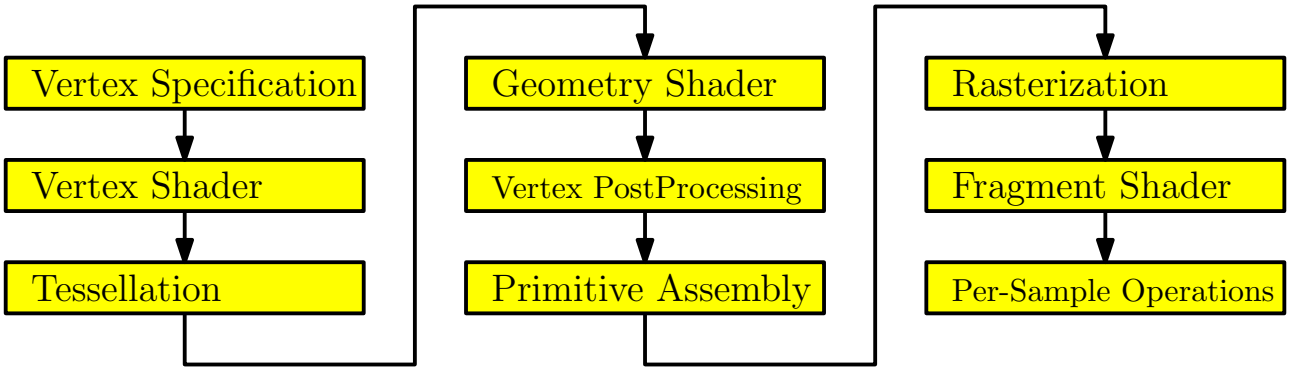

Figure 6.1: OpenGL pipeline

It is not necessary to handle all those steps, they are mainly launched by OpenGL itself. The programmable steps are: *Vertex Shader, Tessellation, Geometry Shader* and *Fragment Shader*, while vertex and fragment shaders are compulsory and two remaining steps are voluntary. In [12], the description of steps is presented:

- Vertex Specification prepares the list of vertices, which will be send to the pipeline. The vertices are clustered into *primitives*, which are triangles, lines and points. In later steps, those primitives will create a whole complex object. Raw data are stored in *Vertex Buffer Objects* (VBO), which provide methods for uploading data to the GPU. The VBOs often contain huge variety of vertex features, like position, normal, texture coordinates or color. For distinguish issues there are *Vertex Array Objects* (VAO), which define data for each vertex in the VBO.

- The vertex shader computes final coordinates of the vertices. The input of vertex shader are basically vertices in its local coordinates. Shader can make computations, like placing the vertex into world coordinates. More details, as well as minimal vertex shader, are presented in next section.

- The tessellation is a process of dividing vertex data into smaller primitives. For example, the square can be tessellated into two triangles and in that step, the dividing is performed and new vertex properties are computed.
- The geometry shader is responsible for creating the objects (triangles, lines...) from simple vertices. The output is a set of primitives.
- The most important part of Vertex post-processing is clipping of objects. Clipping cuts objects, which are on border of visible space, to fit into the space. Moreover the step contains preprocessing to primitive assembly and rasterization.
- During primitive assembly face culling is performed. It prevents non-visible primitives to be rendered. The example is rendering of the cube. Only up to three faces are visible, so others are hidden and there is no need to render them.
- The rasterization process is cutting the primitives into *fragments*. The fragment basically represents simple pixel and contains its position and another data, which are necessary in following steps.
- Second arbitrary shader is fragment shader. During this step, the color value for each pixel is computed. The fragment shader will be also described in following section.
- The last piece of the pipeline, per-sample operations, contain many user-activated tests, color blending and writing to the framebuffer.

6.3.2. Shaders

In previous section, shaders were presented. As mentioned, different types can be used depending on step in the OpenGL pipeline. Shaders are subprograms, which changes input data in some way, written in the *GLSL* (OpenGL Shading Language). GLSL is C-like language designed for vector and matrix arithmetic. Typical shader has following structure:

```
#version

in type var_name
in type var_name

out type out_var_name

void main()
{
        out_var_name = something
}
```

Keywords **in** and **out** represent input and output variables. Communication between shaders is performed only through variables, so when output/input variable names

match between two consecutive shaders, the variable is passed between them. GLSL has common data types like other C-like languages - bool, int, uint, float and double. For shading purposes there are also data types vector and matrix.
The vertex shader recomputes the local coordinates of vertices into world's coordinates. Whole shader is simple:

```
#version 330 core
layout (location = 0) in vec3 position;

uniform mat4 finalMatrix;

void main()
{
    gl_Position = finalMatrix * vec4(position, 1.0f);
}
```

The layout and location number represent data stored in VAO. Uniform is the way to pass data from program to shader. They are (in C++ meaning) static, so they are global for whole shader program (containing all shaders) and remain set until updated. In that case, transformation matrix is passed. *gl_Position* is output variable of vertex shader and its value is gained by simple matrix-vector multiplication.
The fragment shader takes a fragment as input. The fragment is defined by two coordinates on the screen and one coordinate as depth. In this step, the color of the fragment is computed. It can be done by input RGBA vector, uniform passed from the program or by input texture position vector. The following fragment shader sets the color with respect to given RGB vector.

```
#version 330 core

out vec4 color;

uniform vec3 finalColor;

void main()
{
    color = vec4(finalColor, 1.0f);
}
```

The output variable *color* represent final color of the fragment. The fragment shader is also place, where lighting of the scene is implemented. The attached program uses Phong lighting model, which consist of ambient, diffuse and specular lighting. The objects in real world aren't absolutely dark even in night. There is always some weak light, so first component cares about setting constant color. The most significant component is diffuse lighting, which represents amount of impacted light. While the object face's normal is directed more to the light source, the color becames more bright. Finally, the specular component adds bright spot, which appears on shiny objects.

6.3.3. Transformations

If the object is meant to be dynamic, that it's necessary to use transformation matrices. Only three types of transformations are needed and any movement can be described - scale, translation and rotation. Those transformations are described by matrix, which can be chained as needed.

- *Scaling* a vector means multiplying its coordinates by number. Uniform scaling keeps vector's direction and formula $S_1 = S_2 = S_3$ is satisfied. The scaling of vector (x, y, z) is defined as:

$$\begin{pmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 * x \\ S_2 * y \\ S_3 * z \\ 1 \end{pmatrix}$$

- The *translating* is simple sum of vectors, of moving the point by some vector. Let (x, y, z) be point in 3D space. Translating the point by vector (T_1, T_2, T_3) is defined as:

$$\begin{pmatrix} 1 & 0 & 0 & T_1 \\ 0 & 1 & 0 & T_2 \\ 0 & 0 & 1 & T_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} T_1 + x \\ T_2 + y \\ T_3 + z \\ 1 \end{pmatrix}$$

- The most difficult is *rotation*. Let θ be angle representing rotating along the x, y or z axis. The matrices are different according to given axis. Following equations represent rotating of point (x, y, z) along axes.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ cos\theta * y - sin\theta * z \\ sin\theta * y + cos\theta * z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 0 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} cos\theta * x + sin\theta * z \\ y \\ -sin\theta * x + cos\theta * z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} cos\theta * x - sin\theta * y \\ sin\theta * x + cos\theta * y \\ z \\ 1 \end{pmatrix}$$

6.4. Program architecture

The implementation was created in Visual Studio 2015 and is based on object-oriented programming. It contains following classes:

Class name	Class description
Camera	Processes keyboard and mouse input and computes camera properties[9]
Shader	Represents shader program containing vertex and fragment shader[9]
AABB	Axis-aligned bounding box class
OBB	Oriented bounding box class
Sphere	Sphere class
TestAABB	Contains static method for testing two AABBs
TestOBB	Contains static method for testing two OBBs
TestSphere	Contains static method for testing two spheres
TestManager	Determines which testing method should be launched on pairs determined by *testing* matrix
ConvexHull	Computes convex hull of input set of points. Returns set of points lying on the convex hull and their neighbors
GJK	Tests intersection of two convex hulls by Gilbert-Johnson-Keerthi iterative algorithm
Vertex	Represents 3D vertex structure with needed operators
Model	Contains object's drawing data, testing data, bounding volumes and convex hull
DataRepairer	Converts raw STL triangular data into set of unique vertices and corresponding indices

6.5. Main loop

The function *main* is in file *Source.cpp*. It contains the game loop, which recomputes the objects data and redraw OpenGL window. The run of main function can be represented by following diagram.

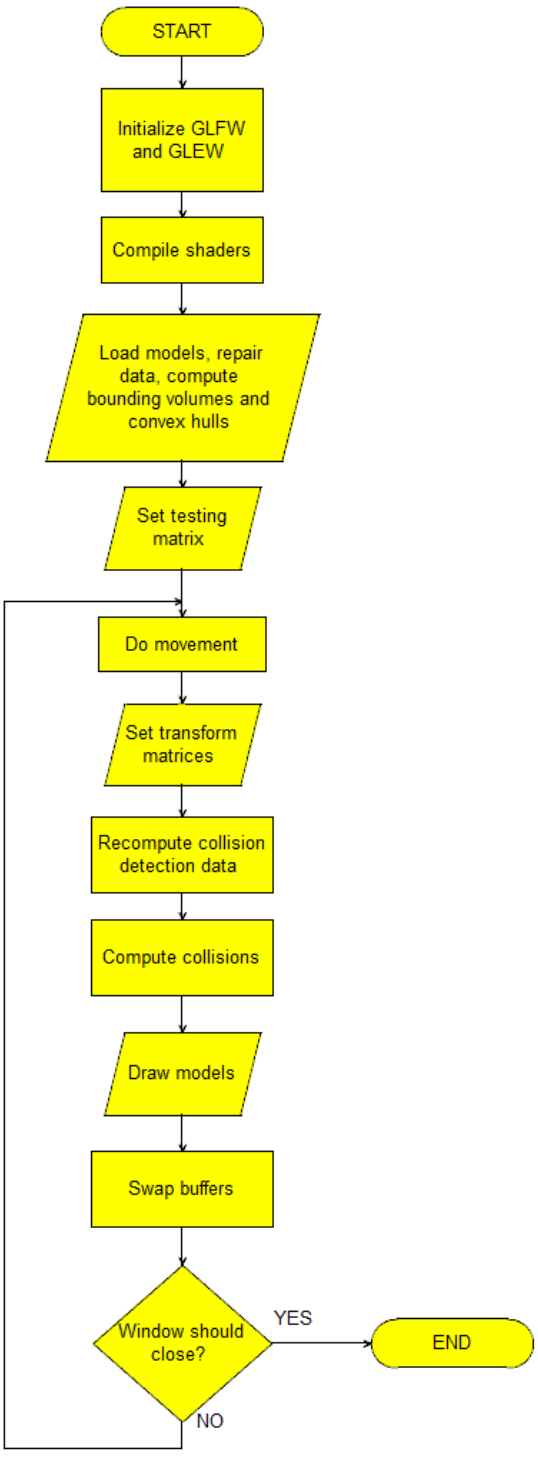

Figure 6.2: Main loop of the program

7. Methods comparison

In this final chapter, the previously mentioned methods will be placed into collision detection paradigm. Evidently all those methods are not suitable for the same detection level. Those levels are presented in first section and then some tests are presented. The tests are launched on following computer: IntelCore i3-3110M 2.40GHz, 4.00GB RAM, Intel HD Graphics 4000, Windows 10. The time measuring is issue while the non-realtime operating system is running due to system's scheduling. It can be solved by launching the application with Diagnostic tool, which is part of the Visual Studio. The diagnostic tool measures time and CPU/GPU usage only when the application runs and even gives values for each running function separately.

7.1. Collision detection phases

The collision detection is normally divided into three consecutive parts. Each part can be easily skipped, but the performance of such algorithm will be lower. Each part reduces searching space, while more accurate and expensive tests are launched. The phases are:

- **Broad phase** - The phase determines which objects will be tested. The basic approach is testing each object against every object in world, so let n be number of objects in world. Then $(n^2 - n)/2$ tests have to be performed, which leads to $\mathcal{O}(n^2)$ complexity and such complexity is (under the term of high number of objects) inappropriate for real-time applications. Due to this fact, some more advanced algorithms has to be used, which determines, which pairs should be tested, and ignores objects too far from other objects. The tests themselves aren't performed yet. Algorithms, which are suitable for broad phase, are mainly based on spatial partitioning, presented in section 3.3.

- **Mid phase** - While potential colliding pairs are determined, the basic tests are performed. Mid phase still doesn't say, if the pair of objects is colliding, but only excludes pairs, which aren't. The test should be fast and more accurate then previous phase. The simple bounding volume is the most used method for catching the false positives. The simple bounding volumes are described in section 3.1 with its presumed asymptotically and memory complexities. The mid phase and the broad phase are often connected in one algorithm. The spatial partitioning methods, presented in section 3.3 fulfill this connection, while it is working with axis-aligned bounding boxes as approximation of objects.

- **Narrow phase** - The narrow phase performs the most expensive and the most accurate methods for determining the collision. In some applications the result of previous phase is sufficient and the narrow phase is skipped. Naturally, more exact detection leads to more expensive test. The methods can be divided into two groups - applied on static and moving objects. In static case, the best method is bounding volume hierarchy, presented in 3.2. The implementation of hierarchy is in

CGAL library, as Axis-aligned bounding box tree, but all simple bounding volume hierarchies (Sphere trees, Oriented bounding box trees) can be used. Moving objects are more likely to be represented as convex objects.

7.2. Simple bounding volumes benchmarks

In section 3.1 some basic bounding volumes were presented. They belong to second phase and their task is removing the part of nonintersecting pairs of objects and the test should be as fast as possible. In this section the benchmark for bounding volumes is presented as well as reached output for each volume. The benchmark:

- The scene contains two objects A, B bounded by given volume
- The positions of object's center c are random, but fixed for all tests
- The number of steps n has the same conditions
- The object B is static, A is translating towards the B, colliding it and then moving apart in same direction. Let d be distance between object's centers. Then the translation vector t_i in step i is defined as $t_i = 2*i*(c_B - c_A)/n$, where $i = 1, 2, ..., n$.
- The rotation is random and fixed

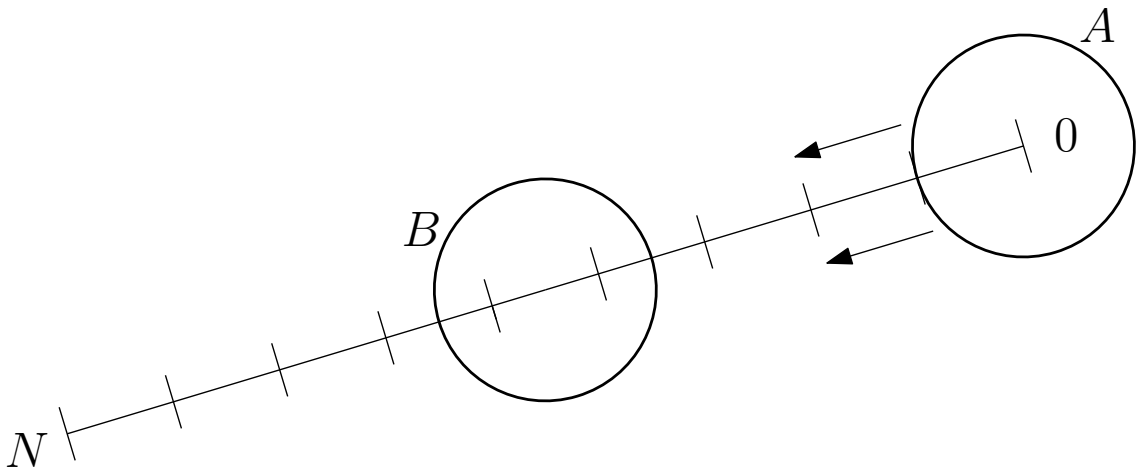

Figure 7.1: Benchmark with N steps for spherical bounding volume

The test with same objects and same fixed parameters is performed with each bounding volume. Following outputs are measured: total time, average test time, rate of positive tests and memory requirements. Interesting properties would be also build time and recomputation time, but those operations are really fast and cannot be accurately measured on non-real-time operating system. The recomputation time is mentioned only in AABB test table, because only in this case the time is significant.

The tests are performed on two pairs of objects O_1, O_2 and O_3, O_4, the objects consist of different number of triangles and have different shape.

Object	Number of triangles
O_1	12874
O_2	25864
O_3	12602
O_4	4948

The tests are launched with following properties:

- N = 20000
- The object A stays in default coordinates, object B is translated by *(0.8, 0.8, -0.8)*.
- The rotation is same for both objects and is given by 0.08° along each axis in each step.

Sphere	**Objects** O_1, O_2	**Objects** O_3, O_4
total time[ms]	9890	11242
avg. test time[μs]	4.35	4.32
rate of positives [%]	42.15	21.38
memory[byte]	28	28
OBB		
total time[ms]	10612	10408
avg. test time[μs]	4.6	5
rate of positives [%]	24.86	15.09
memory[byte]	108	108
AABB		
total time[ms]	10339	9727
avg. test time[μs]	3.75	4.1
rate of positives [%]	51.11	18.81
memory[byte]	42	42
recompute[μs]	1.75	1.2

The data were measured as expected. The sphere and axis-aligned bounding box have quick tests, but the fitting is worse. That leads to higher rate of positives. The oriented bounding box test returns the lowest rate of positives while both pairs are tested, but the tests itself is more expensive. Simple bounding volumes are rarely used as only detecting structure, but are used, as mentioned in previous chapters, with other algorithms. When they are used in some tree structure like an octree or a hierarchy, the most important property is the cost of the detecting algorithm. On the other side, when they are used as first test before more expensive one (e.g. convex hull-based methods), then the accuracy is the main parameter for lowering the count of launching the expensive test.

7.3. Static AABB trees

A part of the CGAL library is collision detection through static AABB tree structure. The input is a set of geometric data, they are converted to primitives and then a hierarchy is build. Leaves of the tree are 3D triangles of original object, so the collision detection is exact. The structure is suitable also for moving objects, but the AABB tree has to be recomputed every time the object moves. In [1] some benchmarks are performed with presented measured values. The tested object is the knot model (figure 7.2), defined by different numbers of triangles.

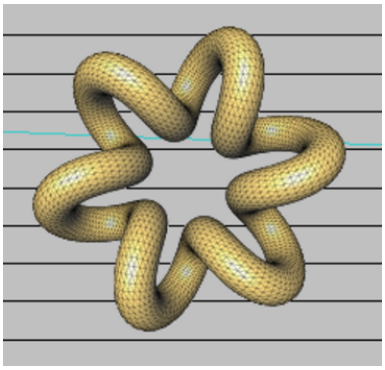

Figure 7.2: The knot model [1]

- Construction[1]

Triangles	Construction [ms]
14400	156
57600	328
230400	1141
921600	4813

- Memory consumption of a AABB tree[1]

Triangles	Memory [MB]
18400	1.1
102400	6.33
1022400	59.56
1822400	108.34

- Intersections[1] - the hierarchy is tested by following queries, numbers in the table represent number of queries performed per second.

Function	Segment	Ray	Line	Plane
AABB_tree::do_intersect()	187868	185649	206096	377969
AABB_tree::number_of_intersected_primitives()	64389	52943	54559	7906
AABB_tree::all_intersections()	46507	38471	36374	2644

The first function returns true, if the query intersect any primitive. An exact number of primitives, which are intersecting, is returned by the second function and finally list of those intersecting pairs is returned by third function.
The AABB tree implemented in CGAL is very powerful tool for exact collision detection, which is illustrated by huge number of intersecting tests per second. The weakness is the usage in the scene with many randomly moving objects, because the trees have to be recomputed or build from scratch. The weakness can be solved by lowering the accuracy and performing convex hull-based methods.

7.4. Convex Hull and Convex decomposition

The aim of this thesis is on methods, which can be used in real-time application with moving objects. Moving objects are mostly represented by convex objects, which are well suitable for fast intersection tests. The main issue is that objects mostly aren't convex, but very complex with large concavity(in sense of 5.1). Methods for overcoming this issue were presented in 4.1 and 5.1. It cannot be said, which method is better, it depends on models and their representation. Complex models can be represented by several STL files, each file representing significant part. In that case can be simple convex hull algorithm, performed on each part, sufficient. The problem appears while the whole complex model is represented by single file. Such model is shown in figure 7.3, representing nanosuit from game Crysis (displayed by Windows 3D Builder).

Figure 7.3: Original model of nanosuit, downloaded from [9]

Creating single convex hull leads to quite inaccurate model, because minimal convex hull encloses, except of the model, huge amount of empty space. In that case considering convex decomposition is in place, which is presented in section 5.1. The V-HACD creates only approximate decomposition, so the output model has slightly bigger volume than

original model, but the difference is almost unnoticeable. Also there is possible concavity, which can be problem for support mapping function in GJK algorithm, while using hill climbing algorithm. To make decomposed parts reliable, for each of them is created simple convex hull to ensure the output will be cleanly convex. The figure 7.4 shows the nanosuit model as simple convex hull and set of decomposed parts. The precision difference is significant. On the other side, computational time and number of pairwise tests increase, so finding the best compromise for a given application is essential.

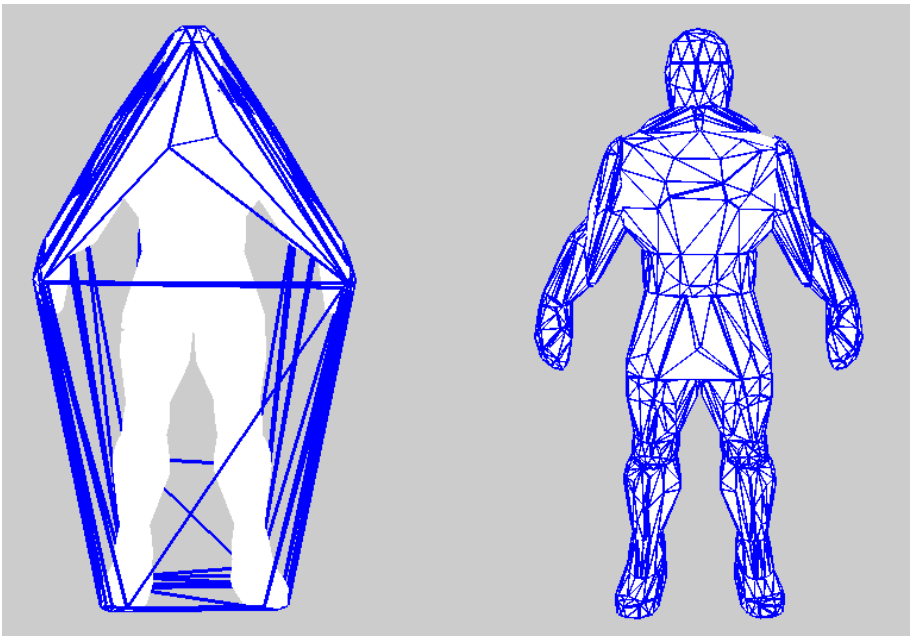

Figure 7.4: Simple convex hull (left) and decomposed hulls (right)

Creating the convex hull is quite expensive operation with worst-case complexity $O(n^2)$, but as far as it is done in data preprocessing stage, it's not an issue. Still, the process should be as quick as possible. The worst case is while all points of the object lie on convex hull, such as sphere. In that case, the points are added one by one to the hull and the time complexity grows. Following table contains data measured on eight different objects, displayed below . First column represents original number of unique vertices. On each object, the convex hull algorithm is used, which leads to geometrical simplification and lowering the number of vertices (second column). Third detail is computational time and last column contains percentage of saved memory.

object	vertices	convex hull vertices	creation time [ms]	memory saving [%]
1	6423	690	39	89.25735638
2	12928	506	29	96.08601485
3	5801	294	11	94.93190829
4	8370	348	15	95.84229391
5	1027	232	5	77.40993184
6	6302	580	35	90.79657252
7	2466	264	7	89.29440389
8	732	202	8	72.40437158
9	5000	5000	2135	0

Only low correlation between original number of vertices and computational time or memory savings results from the table. The reason is, that those values depend mainly on object geometry and number of vertices has only small influence. It can be only said, that objects with dominant spherical shape will have worse statistics than, for example, cubic shaped ones. It is proven by object 9, which is the sphere with 5000 vertices randomly generated on sphere's surface. All its vertices lie on the convex hull, there is no simplification or points erasing, so memory saving is 0%. The test performed on simple and decomposed object is presented in the next section.

7.5. GJK benchmark

In this section, two cases will be tested:

- The same movement as defined in 6.2
- Traversal through simple convex hull and through decomposed hulls.

The first test will be performed with the same properties and pairs of objects and only third collision detection phase will be launched. Measured values are: total time, average test time, rate of positives and average recompute time. The memory issue is already presented in previous section.

GJK	**Objects** O_1, O_2	**Objects** O_3, O_4
total time[ms]	20238	25296
avg. test time[μs]	18.05	19.65
rate of positives [%]	14.27	18.45
recompute[μs]	32.2	33.95

Measured values verify theoretical assumptions. The rate of returned positives is lower then while using simple bounding volume so the test is more accurate, but it is balanced by the duration.
The second test with GJK algorithm is two-phase test (OBB + GJK). The scene contains the complex object and ten small boxes moving around. The test is performed between the object and boxes, boxes aren't tested each other. The loop runs 20,000 times and

collision between object and any box is tested. The test is launched twice, the first table represents simple complex object, the second one represents decomposed objects into 21 pieces. The measured values are: total time, average test time, models loading time and rate of the positives.

OBB + GJK	**Single object**	**Decomposed object**
total time[ms]	26595	29250
avg. testing time [μs]	3.4	8.39
model loading time [ms]	3289	1678
rate of positives[%]	7.8	4.9

7.6. Robotic arm practical example

The practical example of algorithm usage may be movement of robotic arm in space with some obstacles. The collision detection starts by second phase, where objects are approximately tested by OBB test and when this test reports collision, the GJK test is performed. Objects, which passed through OBB test but not through GJK test are colored green. These objects don't collide, but are really close so green color can be considered as warning. While the GJK test is reporting intersection, the objects are colored red. The arm consists of seven objects, while first object is defined in world's coordinates and others in coordinate space of previous object. The obstacles are represented by cubes, which are rotating among near point.

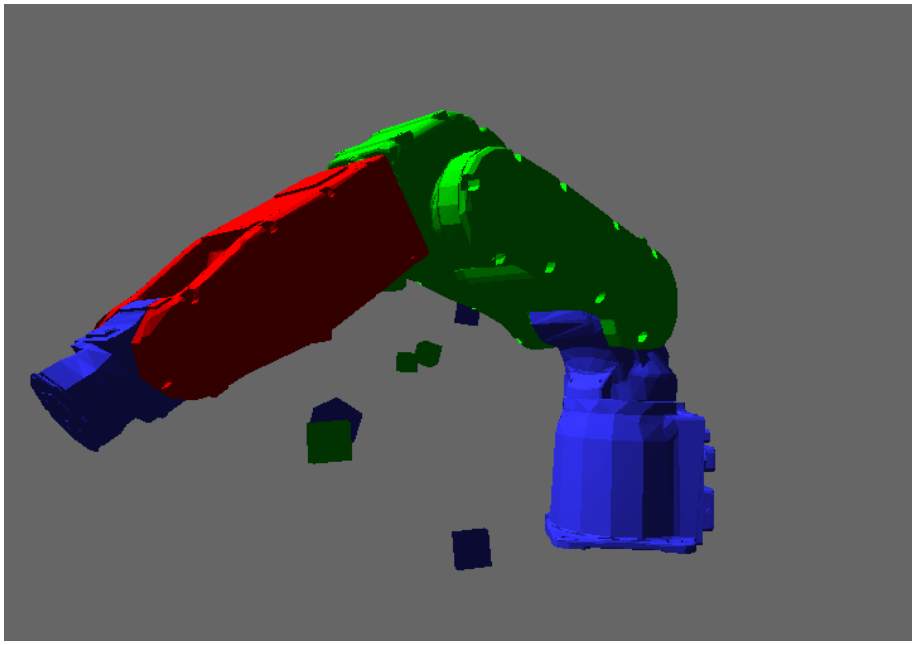

Figure 7.5: Robotic arm moving in space with obstacles

The testing scene was launched in the loop with 20.000 steps. The best illustration of program's effectiveness is following table. It contains diagnostic data of chosen functions,

where the CPU usage percentage is shown. The value represents only usage in program, not whole operating system.

Function	**CPU[%]**	**CPU[ms]**	**Description**
main	92.4	17685	main function of the program
glfwSwapBuffers	40.8	7823	swaps the front and the back buffers (GPU)
glfwPollEvents	19.51	3741	events processing (keyboard input, mouse move, window size changed etc.)
test	7.33	1405	tests data, stores outputs
Model	1.84	353	loads model and creates all additional structures (bounding volumes, convex hull)
recomputeTestingData	0.39	74	recomputes data needed for performing the test

The Model constructor is launched only once, the rest of the functions are launched in the loop, so presented data have to be divided by 20.000 to get average time of one function call.

8. Conclusion

The thesis presents the most used algorithms in collision detection field. The first part of the thesis is cleanly theoretical and contains descriptions of algorithms. For better understanding all partial algorithms are presented as pseudocodes. In the chapter "Mathematical background" overview of necessary mathematical apparatus is presented, mainly the Minkowski difference and convexity of objects. The overview itself is divided into two parts. The first part describes methods based on simple geometrical objects (sphere, axis-aligned bounding box, oriented bounding box) and tree structures(octrees, bounding box hierarchies) and the second one methods based on convex hulls(Gilbert-Keerthi-Johnson, Chung-Wang, Separating axis test). The problem of creating the convex hull is also discussed and own version of algorithm, creating the hull, is presented in chapter "Convex-hull based methods".

In the practical part of the thesis the simulation software was created, based on C++ programming language and OpenGL library. The first group of tested structures are the simple bounding volumes: sphere, axis-aligned bounding box and oriented bounding box. The results shows, that the duration of the test is almost same for all bounding volumes (units of microseconds), but the rates of positive tests vary.

The convex hull algorithm work with worst case complexity $\mathcal{O}(n^2)$, but the complexity is mainly theoretical. It becomes $\mathcal{O}(n^2)$ only if all object's vertices lie on their convex hull. The table in 7.4 shows, that ordinary objects have great memory savings, the share of vertices lying on the convex hull can only be about 4% and the build time is measured in units of milliseconds.

The GJK algorithm is tested in section 7.5 by two ways. The first test is a comparison to the simple bounding volumes and the results show expected values. The test is the most accurate, but also the most expensive. The traversal of a cube through complex object is tested by combined OBB and GJK algorithm. The OBB represents a broad phase test and determines when the expensive GJK test is launched. The object is represented as a whole and as decomposed convex parts.

The final test contains a robotic arm moving in the space with obstacles, which are represented as cubes. The aim is determining percentage usage of the CPU by testing procedure. The results shows, that only 7.33% of CPU performance, assigned for the application, was used for testing the objects.

Bibliography

[1] 3D Fast Intersection and Distance Computation (AABB Tree). cgal.org. [online]. [cit. 2016-05-24]. Available at: http://doc.cgal.org/latest/AABB_tree/

[2] Bullet User Manual and API documentation. BULLET PHYSICS. [online]. [cit. 2016-05-24]. Available at: http:bulletphysics.orgmediawiki-1.5.8index.phpBullet_User_Manual_and_API_documentation

[3] CAPENS, Nicolas. FLIPCODE.COM. Smallest Enclosing Spheres. [online]. [cit. 2016-05-24]. Available at: http://www.flipcode.com/archives/Smallest_Enclosing_Spheres.shtml

[4] Documentation. GLFW. [online]. [cit. 2016-05-24]. Available at: http://www.glfw.org/documentation.html

[5] Documentation. The Computational Geometry Algorithms Library. [online]. [cit. 2016-05-24]. Available at: http://www.cgal.org/documentation.html

[6] EBERLY, David. Geometric tools. Dynamic Collision Detection using Oriented Bounding Boxes. [online]. [cit. 2016-05-24]. Available at: https://www.geometrictools.com/Documentation/DynamicCollisionDetection.pdf

[7] ERICSON, Christer. Real-time collision detection. Boston: Elsevier, c2005. ISBN 1558607323.

[8] KLOSOWSKI James T., Martin Held, Joseph S.B. Mitchell, Henry Sowizral, and Karel Zikan. . Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. [online]. [cit. 2016-05-24]. Available at: http://cdn.intechopen.com/pdfs/34468/InTech-Bounding_volume_hierarchies_for_collision_detection.pdf

[9] Learn OpenGL. [online]. [cit. 2016-05-24]. Available at: http://www.learnopengl.com/

[10] MAMOU, Khaled. GitHub. [online]. [cit. 2016-05-24]. Available at: https://github.com/kmammou/v-hacd

[11] MAMOU, Khaled and Faouzi Ghorbel. . A simple and efficient approach for 3D mesh approximate convex decomposition. [online]. [cit. 2016-05-24]. Available at: http://www.khaledmammou.com/AllPublications/icip2009.pdf

[12] Rendering Pipeline Overview. OpenGL.org. [online]. [cit. 2016-05-24]. Available at: https://www.opengl.org/wiki/Rendering_Pipeline_Overview

[13] STL (file format). Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2016-05-24]. Available at: https://en.wikipedia.org/wiki/STL_(file_format)

[14] SULAIMAN, Hamzah Asyrani and Abdullah Bade. INTECH. Bounding Volume Hierarchies for Collision Detection. [online]. [cit. 2016-05-24]. Available at: http://cdn.intechopen.com/pdfs/34468/InTech-Bounding_volume_hierarchies_for_collision_detection.pdf

[15] The OpenGL Extension Wrangler Library. [online]. [cit. 2016-05-24]. Available at: http://glew.sourceforge.net/