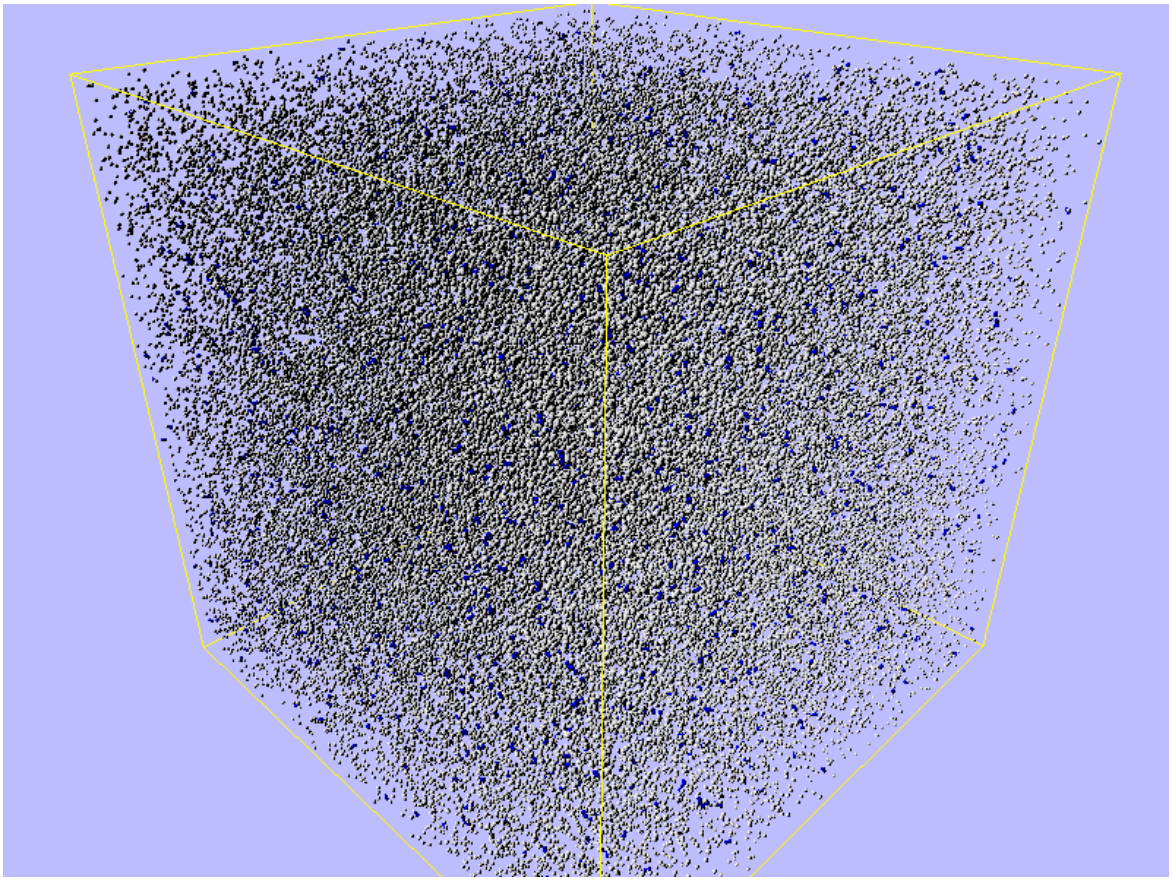
CHALMERS



Fast GPU-based Collision Detection

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages, and Logic

LIN LOI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, December 2010

Fast GPU-based Collision Detection

Lin Loi

Examiner: Ulf Assarsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden December 2010

1 Abstract

In this thesis, fast collision detection algorithms on the Graphics Processing Unit (GPU) and CPU are implemented and performance is evaluated. Many collision detection algorithms are surveyed and discussed and choices of algorithms chosen are motivated.

In this thesis, fast large scale pruning of tens of thousands of potential collisions are made in interactive frame rates. Also exact collision detection algorithms are implemented to find intersection and distance between pair of objects.

Keywords: collision detection, broad-phase, narrow-phase, mid-phase, algorithms, GPU, geometric algorithms

2 Sammanfattning

I detta examensarbete implementeras snabba kollisionsdetekterings-algoritmer för både grafik processor enheten (GPU) och CPU, för att sedan evaluera prestandan. Många kollisionsdetekterings-algoritmer undersöktes och diskuterades och val av algoritmer är motiverade.

I detta examensarbete utförs storskalig gallring av tio-tusentals potentiella kollisioner i interaktiva bildfrekvenser. Dessutom implementeras exakt kollisionsdetektering för att hitta skärning och avstånd mellan par av objekt.

3 Introduction

The purpose of collision detection is to detect collisions between objects in a virtual environment. The field is well-used in areas such as games where objects interact with each other and collisions between 3D objects have to be detected in order to make a realistic and convincing simulation.

Because collision detection algorithms are run during the whole simulation and other parts such as: physical response and rendering, which also requires much computational time, collision detection algorithms have to be fast. Because new graphics hardware have enabled the programmer to write programs that can be run on the graphics hardware's multi-core processors, collision detection algorithms can be parallelized to achieve more speed-up.

Collision detection systems can be divided into three major phases: Broad-phase collision detection, mid-phase collision detection, and narrow-phase collision detection. Each phase's purpose is briefly explained in upcoming subsections without going into details about what algorithms are used.

3.1 Broad-phase Collision Detection

The main purpose of this phase is to do a coarse level prune of objects that cannot collide in the virtual environment. Objects in the virtual environment are enclosed with bounding volumes which simplifies intersection testing. Then the bounding volumes of the objects are tested against each other to see whether they intersect or not. If the bounding volumes do not intersect, one can know for sure that a collision did not occur. If the bounding volumes intersect each other, a finer level of collision detection is needed. The end-result of this phase is a set of potentially colliding objects in the virtual environment.

3.2 Mid-phase Collision Detection

In this phase a set of potential colliding objects are received and a finer level of intersection tests are made. Depending on the complexity of the objects, there might be a hierarchy of bounding volumes which need to be tested for intersection. If the objects in the virtual environment are simple objects (like convex objects with low tessellation), a hierarchy of bounding volumes is not needed and this phase can be omitted. If instead the objects are complex (like non-convex and/or highly tessellated), a hierarchy of bounding volumes is needed. If the latter case holds, each potentially colliding pair is tested for intersection against their respective bounding volume hierarchy. As result, pairs of potentially colliding pairs (which are bounding volumes intersecting each other) are produced.

3.3 Narrow-phase Collision Detection

When potentially colliding objects reach this level, exact collision detection is done on the geometry of each pair. As an end-result the system reports that a collision has occurred, or a collision has not occurred between the pair of objects. If there is a collision and the collision detection system is part of a physics engine system, narrow-phase also has to report the colliding pairs together with the contact points. This will then be used in the collision response part to calculate contact forces (small impulses) to determine physically correct movement of the objects. The collision response part is not within the scope of this report. For further detail see: [Mirtich 1996] and [Eberly 2003].

4 Scope

Because collision detection is a large research field, it is infeasible to cover all major parts within this field. Also due to a tight time frame, choices have to be made on what parts are to be excluded. Parts that are not within the scope of this thesis are: Mid-phase collision detection, continuous collision detection, and collision detection of deformable objects.

5 Previous Work

5.1 Broad-phase Collision Detection

5.1.1 Spatial Data Structures

Octrees In an octree (in 3D) or quadtree (in 2D) (explained in [Akenine-Möller et al. 2008], [Ericson 2005], and [van den Bergen 2004]), space is recursively subdivided forming hierarchy of AABB boxes. The union of this hierarchy forms a large AABB enclosing the environment. An octree is created by first forming a minimum AABB (the root of the hierarchy) enclosing the environment. The AABB of the root is further subdivided in the xy-, yz-, and xz-plane giving 8 AABB children. This is repeated until some stopping criteria are reached. An example of stopping criteria can be that a certain depth of the tree is reached or the current AABB have less than a certain amount of primitives. Notice that the union of children forms their parent node. An octree is different from a hierarchical grid (see next section) in the sense that a hierarchical grid do not need to have an AABB which encloses the whole environment. In other words an octree has a root node.

If the scene to be subdivided is a static scene, top-down octree creation is most straight forward [Ericson 2005]. If this is the case, all primitives are divided into eight children cells and this is then repeated until a stopping criterion is met. If a primitive straddles into several cells, it can be divided or duplicated into the cells it intersects. In the case when objects are straddling cells and inserted to intersecting cells, an array of primitives are maintained and referenced from a node to avoid duplicating objects and consume more memory [Ericson 2005]. If instead primitives are to be divided, they have to reference their original primitives in the same array.

In the case when the scene is dynamic, inserting objects to cells it intersects will lead to complicated updating of the octree as the objects are moving to other cells or starting to straddle other cells. A solution to this is to insert objects into cells that can fully contain the object. By doing this, the update is simplified, but the disadvantage is that objects straddling cells can be added to a cell that is too large, and lead to less good collision culling. A solution to this is to use a *loose octree* instead.

Another variation of the octree (which consider the fact that objects straddling between AABBs will be in a level where the AABBs is large enough for these objects), are the *loose octree* by Ulrich in [DeLoura 2000]. The loose octree relaxes the condition that the cells do not overlap each other. In loose octrees a cell is some factor larger than the minimum AABB of a normal octree. By extending the extents of a cell, objects can now be inserted to a deeper level and therefore lead to better collision culling. The disadvantage is that some more cells have to be checked if an object overlaps the common loose regions. Another advantage is that insertion depth can be calculated when the cell is extended with a factor of 2, leading to fast insertions and deletions.

Hierarchical Grids Before a description of the ideas of hierarchical grids are made; let us discuss the non-hierarchical grids (uniform grids) and its advantages and disadvantages. Uniform grids are a subdivision method to divide space into a grid with cells. All cells are equally large. By dividing an environment into a grid and categorizing objects into cells using their world space coordinates, collision detection is only performed on objects that are located in the same cell.

Often (but not necessarily needed), objects are enclosed in bounding volumes (BVs), such as bounding spheres or axis-aligned bounding boxes. The reason is that intersection testing will be simplified. If a pair of BVs are in the same cell and their BV intersects each other, the underlying objects are passed to narrow-phase collision detection algorithms (see section 5.2) to do more exact collision detection.

There are several ways to determine which cells an object should belong to. If a sphere is used as BV, the center of the sphere determines which cell the object should belong to. If an AABB is used as BV, one of the end-points can be used. By just using one point to determine which cell an object should belong to does not consume as much memory as to add the object to a cell whenever it intersects another cell. If the radius or extents are not saved (due to little memory), neighbouring cells' objects also have to be checked. This method is preferred when there is little memory. Another method is to store an object in cells that its BV intersects. This consumes more memory compared to the first method mentioned, but there is no need to check neighbouring cells.

The problem of uniform grids is to choose an appropriate grid size, if there are objects of varying sizes. A finer grid (smaller cells) will give good collision culling for smaller objects but larger objects will straddle many cells and updates will take more time. If the grid is very coarse, many small objects can be in the same cell leading to more BV tests.

There are several ways to represent grids. [Ericson 2005] has a good presentation of methods. One way is to represent the grid as an array of lists. Objects mapping to the same array position is inserted to a list. Insertion of an object into a list can be made $O(1)$ if objects are added to the head of the list. Deletion is $O(n)$ (n is the total amount of objects) if object do not have a reference to the list, and $O(1)$ if such reference exists to the list (double-linked list to be more precise). The disadvantage of this method is that if the grid has very high resolution (many cells), much memory is needed to represent the grid. Another better way is to use a hash table. By using a hash function to map an object to an array position, the constraint of having a finitely large world is relaxed. The problem is that collisions of array elements can occur, meaning objects belonging to different cells are mapped to the same array position. This is solved by using *separate chaining* [Drake 2005]. The hash table will then store a map inside each array position, where the key is the cell position and value is a double-linked list. Another method exist called *open addressing* [Drake 2005]. Also insertion of an object will be slightly more complicated. First an array position is computed by using the hash function and this takes $O(1)$. Then the map must be accessed to search for a matching key. This search takes $O(l)$ if a linear search of keys are performed to find a matching cell position, or $O(log\ l)$ (using binary search) if the map is sorted, where l is number of keys. Then the object is deleted from the double-linked list in $O(n)$ or $O(1)$ as mentioned before. The advantage of using hash tables are that there is no fixed amount of cells in an environment because a cell will be hashed to an array position. Another advantage is that this method uses less memory. A disadvantage might be the choice of a bad hash function, leading to many collisions.

When objects are moving in the environment, they can go from one cell to another or start to straddle between cells, updates are then needed. In the case when a point from a BV is used to determine a cell to insert into, and this point has moved to another cell, a removal in the corresponding double-linked list has to be made. This will take $O(1)$ if the object has a reference to the list element. This list element must then be unlinked from the list. The point is then inserted to a new cell and the list element is linked to an existing or new list. In the case of using the whole BV to determine the cells intersected, there is a possibility that this BV straddles another cell during update. If this is the case, a new list element must be created and inserted to the correct cell's list. This method of updating is useful when the objects are static, leading to small amounts of list updates, removals and insertions. In the case when many objects are moving it is better to clear all cells and rehash all objects. This is a better choice because if many objects are moving, the number of list operations will be increased.

As mentioned before, uniform grids have problems handling an environment with objects of varying sizes. For that purpose, hierarchical grids are a better choice. A hierarchical grid consists of several levels of grids with different resolution. Going from a level i with cell size c, to a level i+1 will double the cell size to $2c$. So the lowest level has the smallest cell size and the next level's cell size is twice as large, and so on. In [Ericson 2005], objects are inserted to a level where the cell size is large enough to contain the BV of the object, and into a cell where the point of a BV is contained within. Therefore insertion starts in the lowest level and continues up until a level with large enough cell size is found. The lowest level's cell size is as large as the smallest objects of the environment. Because the object is only inserted to a cell that contains the BV's point, neighbouring cells must be checked for intersection with this BV, because the BV can straddle into other cells. In 3D the maximum number of cells to be checked against are eight. See figure 1 for a clarification on how the grid is traversed when finding potential intersections.

Brian Mirtich In [Mirtich 1997], described two different methods to hash objects, which were used in the dynamics simulator *impulse* [Mirtich 1996]. The first method presented, inserts a BV to cells that intersects the object and into a level where the BV can be contained in a cell. See figure 2. Comparing figure 2 to figure 1 one can see that fewer cells are checked, but more time must be spent to determine intersection of a BV to cells, to insert the object to correct cells.

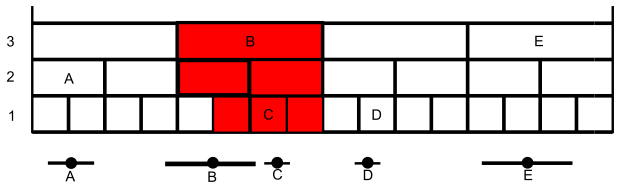


Figure 1: *One-dimensional hierarchical grid with BV center points inserted to a level where BV fits the cell size. The red cells are the cells needed to be checked to find intersections to C.*

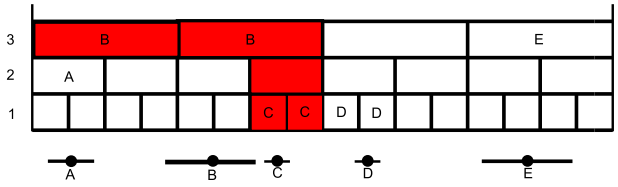


Figure 2: *One-dimensional hierarchical grid with a BV is inserted to a level where the BV intersects a cell and is large enough to be contained in it. The red cells are the cells needed to be checked to find intersections to C.*

In the second method, an object's BV is inserted to a level k where cells can contain the BV plus to all *higher* levels where a cell at level $k+1$ encloses a cell directly above it at level k. Two objects, A and B are close to each other if they share a common cell and the maximum of the lowest level of A and B are shared by both A and B. A reference to the lowest cell level can be stored for an object to speed up lookup. See figure 3 for an explanation. This method reduces the number of cells checked to find close pairs, but consumes more memory because a pair data-structure must be used to track a counter for pairs of objects. Also work has to be done to update the hash table when objects are moving because now, not only one level needs to be updated for an object, but also the levels above it. The pair tracking data-structure also needs to be updated, decrementing the object pair counter when a BV moves to another cell. Mirtich showed that this method spends fewer cycles than the previous method (around 6-7 times less). The scenes for the performed test are from [Mirtich 1996].

In [Le Grand 2007], uniform grids are parallelized and bounding spheres are the used bounding volume. The grids/cells are set to be larger than the largest bounding volume. Because of this constraint an object can intersect a maximum of eight cells in 3D. Another consequence of this is that the maximum amount of array elements needed to store cell intersections are calculated as 8*N in 3D, where N is the number of objects. This array is referred as cell-id in the article and it stores an id telling which cells a specific object intersects. Each object is then processed in parallel to determine the

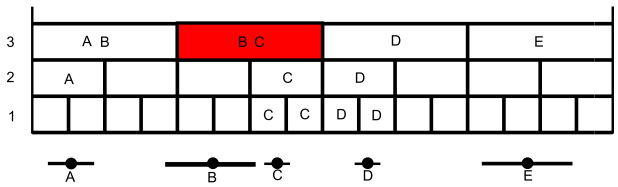


Figure 3: *One-dimensional hierarchical grid with BVs inserted to all cells large enough to contain this BV and to higher levels. The red cells are the cells needed to be checked to find intersections to C.*

cells they intersects and writes that to the corresponding array position along with an object id. If an object intersects less than eight cells, invalid array elements are marked as 0xFFFFFFFF (all bits of a 32 bit word set to one).

Each object also has some control bits used in the end of the algorithm to process objects in cells not neighbouring each other to ensure no redundant tests. The newly created cell-id array is then sorted and same cell-ids are grouped together. After this, the sorted cell-id array needs to be inspected to look for cell-id transitions, so one can know where to start collision tests.

The algorithm requires setting some hard constraints on the properties of the objects such that they intersect a maximum of eight cells in 3D. The algorithm is very useful and fast when dealing with simulations where the scene consists of objects with uniform size. An extension to the method would be to support objects that can intersect an arbitrary amount of cells by having a pre-pass function to count the number of cells each object intersects and then performing a prefix sum [Blelloch 1990] to determine starting offsets for each object to write the cells it is intersecting [Kalojanov and Slusallek 2009]. Another remark is that one thread processes objects belonging to same collision cell and thus work load distribution can be very uneven for scenes where many objects belong to the same cell, whereas other cells contain only a few objects.

Binary Space Partitioning Tree A BSP-tree, first introduced in [Fuchs et al. 1980], is a more general tree compared to previous mentioned trees, because partitioning planes can be arbitrarily defined. Kd-trees are a special case of BSP-trees, in which partitioning planes are axis aligned planes. As the name says, a BSP-tree partitions space into two halves: a positive and negative half-space (the BSP-tree is a binary tree).

There are several ways to choose a partitioning plane. One way is to have a predefined set of planes to choose from. Another way is to choose a plane aligned to the coordinate axis (kd-tree), or picking planes from faces of an object, also called a *polygon aligned BSP-tree* or *auto-partitioning* [Ericson 2005]. Planes can also be picked arbitrarily (called general or arbitrary BSP-tree). Another method to pick good partitioning planes is presented in [Naylor 1993].

In [Fuchs et al. 1983] the BSP-tree is used to cull away polygons of an object which are not visible to the viewer, and the described BSP-tree is a *node-storing* BSP-tree [Ericson 2005], where partitioning planes are planes coinciding with a polygon of an object. In a node-storing BSP-tree, every internal node stores a polygon (implicitly defines a plane by polygons normal and translation). The tree is created by randomly picking a polygon as an internal node and then test each polygon of the object against this newly picked polygon, and assign polygons to a left and a right sub-tree depending on if a polygon is on the positive half-space of the plane or in the negative half-space of the plane. Polygons coinciding with the dividing plane are on the same node and polygons straddling a plane are divided. This process is repeated for newly created sub-trees until some stopping criteria, like tree depth or a certain amount of primitives left.

Another variant of the BSP-tree and more useful for collision detection is the *leaf-storing BSP-tree* [Ericson 2005]. This type of BSP-tree stores polygons in the leaves of the tree and internal nodes are arbitrary planes or polygons from objects. If a polygon aligned variant is chosen, polygons should be marked so they are not chosen again when picking a plane as an internal node. Collision queries are done by traversing the BSP tree with a query to determine which half-space the query belongs to and do this down to leaf level and then test the polygons in the leaf. If the query straddles internal node, both sub-trees must be traversed.

Yet another useful BSP-tree variant described in [Ericson 2005], is the *solid-leaf BSP-tree*. A half-space of a dividing plane tells whether it is part of the object or outside of the objects. By intersecting all half-spaces which are part of the objects, the original object will be formed. Only the leaf nodes store the information of whether the half-space is outside of the object or inside. Choosing a partition plan can be arbitrary or polygon aligned, but it is important that all polygons of an object are chosen as a partitioning plane.

A collision query of a point on a BSP-tree is very simple. The point is tested against the root node first to determine the half-space to continue on. This is repeated recursively until a leaf node is reached. Now the point is inside the object if the leaf node represents the inside of an object, and outside if the node represents outside of object. If a point is on the plane, both sub-trees are traversed recursively and if leaf nodes are different, the point is on the boundary. Otherwise it is inside or outside the object.

Continuous collision detection of a point on a BSP-tree can also be done between frames. If it turns out that in frame n the point is outside an object, but in frame $n+1$, it is inside the object, a bounded line have been formed from frame n to $n+1$. Now *time of impact* can be determined by finding a point on this bounded line, which intersects the boundary of a partitioning plane.

Collision detection of a bounding volume P on a BSP-tree can also be done. The process which is described in [Melax 2000], is similar to the point-BSP-tree query. By forming the Minkowski sum (addition) (see section 5.2.3) of P and the BSP-tree, P will be reduced to a point p and the previous method can be used to detect a collision. Because the Minkowski sum is an addition of all points from P with all points on all partitioning planes of the BSP-tree, the BSP-tree planes have to be shifted in the normal direction of the planes with a width of the bounding volume (for example: The radius if the bounding volume is a sphere). Mentally the Minkowski sum can be seen as sweeping P on planes of the BSP-tree. Just shifting the planes of the BSP-tree is not enough because this will include too much space (can be seen mentally that the sweep of P is not the same as shifting the planes in the normal direction), so *bevel planes have to be added*. In [Melax 2000], bevel planes are added to neighbouring planes if the outer angle is greater than 90 degrees. This reduces the error but does not completely remove it.

In [Luque et al. 2005], a *semi adjusting BSP-tree* is proposed to handle dynamic objects in a scene. Partitioning planes are chosen from a predefined set of planes and each of them are evaluated with a goodness criteria and the best one is picked as partitioning plane. The goodness criteria depend on 3 factors (taken from [Luque et al. 2005], where p is a partitioning plane):

- population(p): Number of objects tested against partitioner p.
- balance(p): number of objects in the positive half-space and negative half-space. The ratio of the smaller one over the larger one.
- redundancy(p): number of objects straddling partitioner p.

These criteria can easily be determined by maintaining a sorted list of objects for a partitioner p. Also a set of operators on these BSP-trees are proposed, which are applied in a certain order during update of the tree. The proposed operators are: *split*, *shift-split*, *merge*, *balance*, and *swap*. The split operator determines a plane to split objects into two half-spaces. Given a candidate normal the split is orthogonal to the normal and this plane is located where the redundancy is the lowest and the balance the best. The split is applied when a population of a node becomes larger than a threshold. The shift-split operator is the same as split operator, but a partitioning plane is just shifted along its parent node's normal with a constant and a plane that satisfies the goodness criteria is chosen. The merge operator removes a partitioner and merges the list of a leaf node with an internal node. This is done when objects leaves one half-space, in which a half-space becomes empty or falls below a threshold. The balance operator shifts a partitioning plane along a direction which satisfies the goodness criteria (especially balance criteria). The swap operator removes a partitioner that is unbalanced and uses one of its sub-trees (the one with highest population) as new root and inserts objects from the other half space which belonged to the removed partitioner.

These operators are scheduled by a scheduler, which calculates the number of operators that should be applied, and the rest is deferred to next update of the tree. Update of the BSP-tree is not done every time an object changes position but it is done in some regular interval and the scheduler schedules and defers operators to be applied in this update and other operators to be applied later on.

5.1.2 Sweep and Prune

Sweep and Prune or *Sweep and sort* [Lin 1993] and [Witkin et al. 2001], is an $O(nlog^2n+k)$ algorithm in worst case and an $O(n+k)$ algorithm when frame to frame coherency is exploited [Lin 1993], where n is the number of objects and k is the number of pairs overlapping. Different kinds of bounding volumes can be used, but the most used ones are bounding spheres and AABBs. Spheres are suitable because one can easily find extreme points along a sweeping axis. The same applies for AABBs when any of x-,y-, or z-axis is the sweeping axis. Variable size AABB can also be used but a fixed size AABB is preferred. A method to create variable size AABBs is proposed in [Lin 1993], by using the Lin-Canny Algorithm 5.2.1 to search for six vertices on a convex polyhedron (the convex hull must be used if polyhedron is non-convex), which gives the shortest distance to six boundary walls (set to maximum and minimum values in each axis). Because polyhedron is convex, frame to frame coherency can be exploited by using the max (or min) vertex from previous frame (for the interested axis), as a starting vertex for the distance calculation. The same method can be applied to other algorithms such as the GJK algorithm 5.2.3, the V-Clip algorithm 5.2.1 or any other shortest distance algorithm. Another way to find max (or min) vertex is to use *hill climbing* 5.2.3 on the max (or min) vertex for an axis from a previous frame to start a search for a new max (or min) vertex in current frame. This method also exploits the frame to frame coherency.

In [Akenine-Möller et al. 2008], it is said that a fixed size AABB used for the sweep and prune algorithm gives better performance than a variable size AABB, so focus will be on this kind of AABBs when describing the idea of the algorithm. The reason for this, is that the amount of swapping needed during the sort of AABBs (due to objects can rotate and a Non-fixed size AABB have to be recomputed) are reduced.

By using a bounding volume that is aligned in the x-, y-, and z-axis it is sufficient to detect interval overlaps in these three axes to deduce a bounding volume overlap. To deduce that no overlap had occurred it suffices to find that intervals of two bounding volumes do not overlap in one of the axis. By solving the one dimensional interval overlap problem for all three axes one can find overlapping pairs of AABBs.

The idea of the sweep and prune algorithm is the following: Consider an AABB to be bounded in the interval $[b_i, e_i]$ for one of the three axes, where b_i and e_i are the endpoints of an interval belonging to the i:th AABB and $0 \leq i < n$. Given n objects and $2n$ endpoints, these endpoints are sorted in ascending order. The sorted list (or array) is then traversed (swept) and collision pairs are found. When a b_i endpoint is received while traversing the list, the

corresponding AABB is inserted to an active collision list. If e_i is received, the i:th AABB is removed from the list of active collisions. Let us say that a b_i is received and the i:th AABB is in the active collision list. Now a b_j is received, where $i \neq j$, then the j:th AABB is added the active collision list. As long as a beginning endpoint is received when traversing the swept list, the corresponding AABB overlaps all AABBs in the active collision list.

In an environment where small movements exist for objects, frame to frame coherency can be exploited. Because of this, changes of intervals are small so the swept list is almost sorted. By using insertion sort to a nearly sorted list, the expected sorting time can be reduced to $O(n)$.

By keeping track of a bookmarking flag for interval pairs in each frame, the frame to frame coherency is exploited. Whenever insertion sort swaps place of two endpoints the corresponding flag is toggled.

Implementation details are discussed in [Ericson 2005] and [van den Bergen 2004]. The sweep and prune algorithm can be integrated with uniform grids to speed up the algorithm [Tracy et al. 2009]. The authors analyzed the algorithm and came to the conclusion that the swapping behaviour of the algorithm is the factor affecting the performance the most. By using uniform grids to divide space into cells and apply the sweep and prune algorithm in these cells, the clustering behaviour can be reduced. The authors also suggest a new data structure to maintain collisions: The *Segmented Interval List*. This list is a hybrid of the array based method and the list based method (explained in [Ericson 2005]). The structure maintains an array that contains chunks. Chunks are double linked lists. Each chunk has an array of intervals and a set of *checkpoints*. A checkpoint contains intervals that spans over this chunk but this chunk do not contain any of the endpoints of an interval. Checkpoints are references to an object's ID, where it's minima is on the same chunk or *one chunk* to the left and the maxima is in a chunk to the right. This proposed method and combination with uniform grids speeds up the algorithm in a scene with many objects and relatively small amount of them are moving. This proposed extension of sweep and prune with uniform grids, where cells of a grid is independent of another cell, makes the algorithm a good candidate for parallelization.

5.1.3 Combining Spatial Subdivision with Sweep and Prune on GPU

As mentioned in previous section combining these two algorithms is a good candidate for parallelization. In more recent research [Liu et al. 2010] it has been done on the GPU. The main differences compared to [Tracy et al. 2009] is the fact that instead of launching the sweep and prune algorithm on every cell, which can give a very uneven work load distribution between thread, sweep and prune is deferred until the end after performing a two level spatial subdivision along other than the best sweeping axis of sweep and prune.

The work uses Principal Component Analysis [Jolliffe 2002] to determine the best sweeping axis and the scene is then subdivide with planes parallel to this best sweeping axis. A first level of subdivision is performed and objects are mapped to cells. An object that straddles between cells forms a new cell containing this object along with other objects straddling this newly created cell. Each cell is then given a unique id. The cells are then shifted along the best sweeping axis to ensure that objects in different cells will not be processed by parallel sweep and prune. This shifting operation reduces the clustering behaviour mentioned earlier, plus it reduces the swapping behaviour of an insertion sort if incremental updates are made.

A second level subdivision is also performed for every first level cell to further cull away objects. An object is assigned a bit-string that indicates which cells it intersects and by bit-wise ANDing these bit-strings between two objects, one can easily determine if two objects share the same second level subdivision cell.

The main algorithm is the parallelized sweep and prune algorithm. The algorithm itself is not changed very drastically. The difference is that instead of an insertion sort, a fast radix sort (such as [Satish et al. 2009] [Merrill and Grimshaw 2010]) is used to sort the intervals. Another difference is that only the starting points of the intervals are sorted. So if a key-value sort is used, the keys will be all the starting points, and as value all the ending points. Then for each sorted key, the corresponding value is picked which marks where the ending point is. Next, keys after the current key is picked and compared to the current value (the end-point), and an overlap has occurred whenever the starting point is smaller than current ending point. See pseudo code 1 below or [Liu et al. 2010] for a clarification.

Algorithm 1 The parallel sweep and prune algorithm. BV stands for bounding volume.

Input: Objects: $O : \{O_1, O_2, ..., O_n\}$, O_i has a starting point s_i and an ending point e_i
Input: Overlapping pairs set: S. (Pre-condition: $S = \emptyset$)
Output: overlap set: S, containing pairs of BVs intersecting
radix sort with $< key, value >:< s_i, e_i >, 1 \leq i \leq n$
do in parallel:
for each $s_i, 1 \leq i \leq n$,
while $s_j \leq e_i, i < j$ **do**
  if BV_i intersects BV_j **then**
    $S = S \cup \{O_i, O_j\}$
  end if
end while
end for each

The report also discusses a work load distribution heuristic to even out the workload of threads. In some scenes, performing sweep and prune for some objects might involve a larger amount of potentially colliding objects to test against. By assigning more threads to objects that might have more potentially colliding tests, the uneven work load can be reduced. The amount of threads assigned to an object is determined by its size. The larger an object is, the more likely it will have more potentially colliding tests, because it is more likely that objects are within the sweeping interval of this larger object. So the larger the object is, the more thread it gets assigned.

5.2 Narrow-phase Collision Detection

There exist many different kinds of narrow-phase collision detection algorithms. Examples are: Triangle-triangle intersection algorithms, feature-based algorithms, separating axis algorithms and simplex based algorithms.

The surveyed algorithms are very fast and can exploit the frame to frame coherence that can exist between objects. Triangle-triangle intersection tests can also be used, but using it without a bounding volume hierarchy is very time consuming, requiring $O(n^2)$ triangle-triangle tests, where n are the number of triangles in the scene. Also in this thesis, mid-phase collision detection is not surveyed, so tirangle-triangle intersection tests are left out in this section.

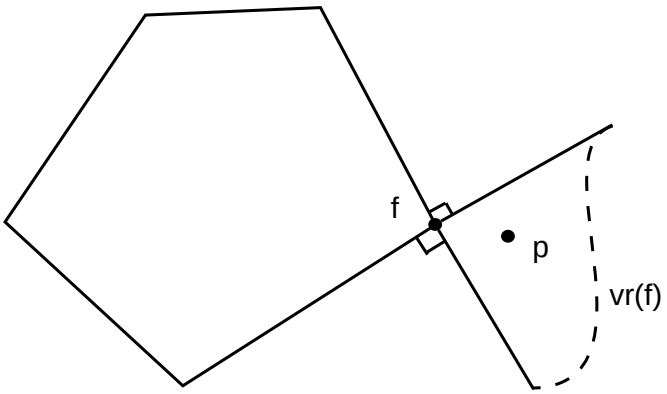


Figure 4: *Convex polygon with a feature f and a point p, located inside the voronoi region $vr(f)$ of f ($p \in vr(f)$). Vertex f is the closest feature to p.*

5.2.1 Feature-based Algorithms

Here two algorithms will be presented to detect collisions: The Lin-Canny algorithm [Lin 1993] [Cohen et al. 1994] and the V-Clip algorithm [Mirtich 1998]. A similarity for these two algorithms is that they consider the *features* of a convex polyhedron, which is the vertices, edges and faces that build up the polyhedron. They also use voronoi regions to determine whether two features are the closest feature pair. Because these algorithms use features from two convex polyhedra to determine the shortest distance, they are restricted to convex polyhedral objects (objects consisting of vertices, edges and faces). So both the Lin-Canny algorithm and the V-Clip algorithm do not support implicit surfaces (surfaces that are defined by parameters. For example $x^2 + y^2 + z^2 = 1$ is a sphere centered at the origin with radius 1), which the GJK algorithm (see section 5.2.3) does. An advantage is that the algorithms are really fast and by exploiting frame to frame coherency an almost constant running time can be reached for both algorithms, according to [Lin 1993] (Lin-Canny algorithm) and [Mirtich 1998] (V-Clip).

The Lin-Canny algorithm As mentioned before the Lin-Canny algorithm, which is included in the collision detection library I-COLLIDE [Cohen et al. 1995] is a feature-based algorithm. The Lin-Canny algorithm uses voronoi regions to determine closest feature pairs of two convex polyhedra. See figure 4 for a clarification. As seen from the figure: A point p is inside a voronoi region $vr(f)$ of the feature f. This vertex feature f, is the closest feature for the point p. Calculating the distance is then trivial. To find the closest feature between two convex polyhedra, six cases have to be considered (where V stands for a vertex, E for an edge, and F for a face): $V_A - V_B$, $V_A - E_B$, $V_A - F_B$, $E_A - E_B$, $E_A - F_B$ and $F_A - F_B$. The notations are from [Lin 1993], and the closest feature pair of convex polyhedron A and B are desired. Let $vr(f)$ denote the voronoi region for a feature f.

Case $V_A - V_B$: If V_A is inside vr(V_B) and V_B is inside vr(V_A), V_A and V_B are the closest feature pair. If V_A is not inside vr(V_B), a new pair of features are tested. In this cases, V_A is tested with E_B, where E_B is the violated constraint from vr(V_B).

Case $V_A - E_B$: Two conditions must hold here: V_A must be inside vr(E_B), and if that is the case a closest point p in E_B is calculated. If p is inside vr(V_A) then $V_A - E_B$ is the closest feature pair. Otherwise update with a new pair of features.

Case $V_A - F_B$: This case is similar to $V_A - E_B$ except now we are dealing with a face F_B. V_A is tested against the voronoi regions of F_B. If V_A is within the voronoi regions of the face, a closest point on F_B is determined and this point has to be inside $vr(V_A)$. If V_A violates any voronoi region constraints, the corresponding violated constraint's feature (an edge) is picked as next feature. One also has to consider the case where V_A can be inside B or on the other side (not intersecting B) of B. The algorithm has to check that V_A is not inside B. If V_A is inside B, a linear search (in the number of features of B) is performed to find the closest feature f to V_A, and continues from there.

Case $E_A - E_B$: Closest point P_A on E_A and closest point P_B on E_B are found. If P_A is inside vr(E_B) and P_B is inside vr(E_A) then $E_A - E_B$ are the closest feature pair. If this is not the case, one of the edges will choose one of its neighbouring features (a face or an edge).

Case $E_A - F_B$: Two cases can occur. If E_A is parallel to F_B then three conditions must hold for $E_A - F_B$ to be the closest feature pair:

- E_A must intersect vr(F_B).
- $F'_B s$ face-normal must lie between $E'_A s$ neighbouring faces' normal.
- E_A must lie above F_B.

If E_A and F_B are not parallel, then one endpoint of E_A will be closer to the face. In that case, V_A (which is one of $E'_A s$ endpoints) is tested against the face (the $V_A - F_B$ case). Otherwise E_A is tested against the closest edge E_B of F_B.

Case $F_A - F_B$: If F_A and F_B are parallel, an overlap check is performed using the edges of F_A and F_B. If they are parallel and the projection of one face down on the other face overlaps, and both faces are above each other relative to their normal, then $F_A - F_B$ is the closest feature pair. If they are parallel but the projection does not overlap each other a liner search on the closest pair of edges from F_A and F_B are performed and those features will continue the algorithm. If the faces are not parallel, an edge or vertex of F_A that is contained in F_B is returned as potential closest feature. If it is an edge the $E_A - F_B$ case is performed and if a vertex the case $V_A - F_B$ is performed. If those cases succeed a closest feature pair is found. If this is not the case, the closest edges are found and the algorithm continues.

The analysis in [Lin 1993] states that the algorithm will converge in $O(N_A \cdot N_B)$ at most, where N_A and N_B is the number of features for object A and object B respectively. Lin also states that empirically, the algorithm's running time is not worse than constant time. To further increase speed, coherency is exploited: The closest feature pair from the previous frame is used as initialization for the current frame. Considering this, the algorithm converges in almost constant time. Because the running time is almost constant in scenes where coherency is high, one good application of the algorithm will be within the field of animation, where abrupt changes of a polyhedron are very unlikely.

The above description of the algorithm just describes the general idea. For more details see [Lin 1993].

In [Ponamgi et al. 1995], an extension to the original algorithm is made, which adds the support of detecting penetrations between polyhedra. The idea is to create so called *pseudo internal voronoi regions*, which are regions internal to the polyhedron. These regions are then used to detect whether a penetration has occurred between two polyhedra. The pseudo internal voronoi regions are created by first finding the weighted average of all vertices (the centroid), and then planes are created by extending edges toward the centroid of the polyhedron.

Collision detection between polyhedra now extends to checking these pseudo internal voronoi regions. If a candidate feature fails the constraint for being above a face, then the internal voronoi regions of the face are checked. If these constraints hold, we now know that the candidate feature is inside the polyhedron, meaning a penetration has occurred. If any of the constraints for the pseudo internal voronoi regions fails, the algorithm progresses as normal.

An alternative to the Lin-Canny algorithm is Brian Mirtich's V-Clip algorithm [Mirtich 1998]. It is similar in the sense that it also uses voronoi regions to find the shortest distance. A major difference is that the V-Clip algorithm performs clipping against voronoi regions to obtain better robustness in degeneracy cases. Mirtich also compared the V-Clip algorithm against the Lin-Canny algorithm and Cameron's enhanced GJK (see section 5.2.3 and [Cameron 1997] for Cameron's enhanced GJK), and showed that the V-Clip algorithm is faster and more robust. Mirtich also claims that the Lin-Canny algorithm cannot handle the case when polyhedron A and B are initialized in an already intersected state. The Lin-Canny algorithm will then cycle forever.

As mentioned before for the Lin-Canny algorithm, coherence can also be exploited by V-Clip to get an almost constant running time.

5.2.2 Separating Axis Algorithms

Separating Axis Theorem The separating axis theorem (SAT) algorithm described in [Gottschalk et al. 1996], is an algorithm to find a separating axis for two convex polyhedra. If such an axis exists, then the polyhedra do not intersect with each other. Possible separating axis for polyhedron A and polyhedron B are: A face-normal of A, a face-normal of B and the cross product of an edge from A and B. To see whether a potential separating axis d is a separating axis, A's and B's *support vertex* on direction d are first found. A support vertex SV of A in a direction d is defined as:

$$SV(d, A) = max\{d \cdot (x - o) : x \in A\} \tag{1}$$

where o is the origin. To check whether d is a separating axis, intervals $I_A = [I_{A_{min}}, I_{A_{max}}] = [SV(-d, A), SV(d, A)]$ and $I_B = [I_{B_{min}}, I_{B_{max}}] = [SV(-d, B), SV(d, B)]$ are found. The direction d is a separating axis if:

$$I_{A_{max}} < I_{B_{min}} \; or \; I_{A_{min}} > I_{B_{max}} \tag{2}$$

Time complexity of the SAT algorithm is $O(N_A(H_A + H_B) + N_B(H_A + H_B) + E_A E_B(H_A + H_B))$, where N_A is the number of face-normals for A, N_B is the number of face-normals for B, E_A is the number of edges for A and E_B is number of edges for E_B. H_A and H_B is the time it takes to find the extreme intervals for a direction. If a linear search of vertices in A and B are done, then H_A and H_B are N_A respective N_B. One can construct a BSP-tree to speed up the search of extreme vertices [Eberly 2003]. If this is done then the search will be reduced to the height of A's respective B's tree. In the case of a BSP-tree $H_A = log_2(N_A)$ and $H_B = log_2(N_B)$. The given time complexity is the worst case, where no separating axis is found and intersection is detected. As mentioned before temporal coherency is exploited to get a speedup: The separating axis from last frame is used for the current frame.

Separating Vector Algorithm The separating Vector (CWSV) algorithm [Chung 1996] [Chung and Wang 1996a] by Chung and Wang, is very similar to the SAT algorithm. The CWSV algorithm tries to find a separating axis between polyhedron P and Q just like the SAT algorithm, but the way to find a separating axis vector is different. In the initial phase an arbitrary vector S_i is chosen and equation 1 is used to determine the support vertices $p_i \in P$ in the direction S_i and $q_i \in Q$ in the direction $-S_i$. Then S_i is a separating axis if:

$$S_i \cdot (q_i - p_i) > 0 \tag{3}$$

If equation 3 holds, then a separating axis is found. If equation 3 does not hold then a new direction S_{i+1} have to be determined. Let $r_i = (q_i - p_i)/||q_i - p_i||$. Then S_{i+1} is defined as:

$$S_{i+1} = S_i - 2(r_i \cdot S_i)r_i \tag{4}$$

In other words S_{i+1} is the reflected vector of S_{i+1}, given the normal vector $r_i^{\perp}$. See figure 5 for clarification.

The algorithm stops when the origin O is contained in the Minkowski sum (see section 5.2.3) $M = Q + (-P)$, which means that a collision has occurred. Let $m \in M$ and the vector w is a separating axis if:

$$m \cdot w \geq 0 \tag{5}$$

is true. The vector w cannot be found if a collision has occurred. This can be seen geometrically: let $r_i = m_i/||m_i|| = (q_i - p_i)/||q_i - p_i||$, where $0 \leq i \leq k$ (k can be seen as a maximum threshold of iterations). All r_i will then lie on the unit sphere. If all r_i can be divided by a plane (which is passing through the origin), such that all r_is are in one side of the plane, then no intersection has occurred. If i = k and a plane cannot be found then collision has occurred.

As mentioned before, the separating axis from the previous frame is cached and reused. Also last support vertices are cached and used as a starting point of search, when new support vertices for S_{i+1} are needed. Because of convexity a local search is sufficient according to Chung and Wang. They also say that using temporal and geometric coherency, an expected constant running time is reached. According to them, the worst time complexity is: $O((log(n) + log(m) + k) * k)$ where n and m is the number of vertices for each polyhedron and k is the number of iterations performed.

In [van den Bergen 1999a], Bergen claims that the proof of convergence is incorrect for the CWSV algorithm in [Chung 1996] and in [Chung and Wang 1996a]. Bergen showed that calculating a new S_{i+1} may not give an axis closer to a separating axis:

$$S_{i+1} \cdot w \geq S_i \cdot w \tag{6}$$

for a separating axis vector w. What equation 6 says is: For the next iteration, the new potentially separating axis vector S_{i+1} is a *better* or an *equally good* vector as the previous one. In [van den Bergen 2004] he also claims that the CWSV algorithm performs better than the ISA-GJK algorithm (see section 5.2.3) on an average, but the ISA-GJK algorithm has a tighter worst case bound compared to the CWSV algorithm.

The CWSV algorithm is part of the collision detection library Q-COLLIDE (Quick Collision Detection Library) [Chung 1996] [Chung and Wang 1996a] [Chung and Wang 1996b].

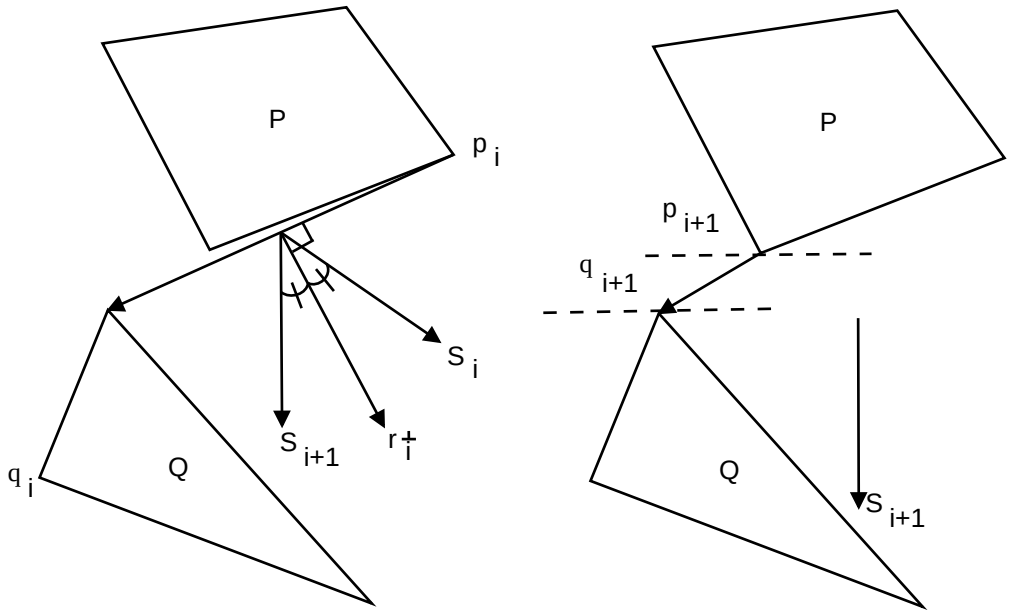


Figure 5: *S_i is reflected on normal vector $r_i^{\perp}$ to get S_{i+1}, which is a separating axis.*

5.2.3 Simplex Based Algorithms

In this section, a simplex based algorithm and its improvements are described. The algorithm described is the GJK algorithm. The GJK algorithm is called a simplex based algorithm because it uses *simplices* inside a convex polytope to determine a shortest distance or finding a separating axis.

The GJK algorithm The Gilbert-Johnson-Keerthi distance algorithm (GJK) is a versatile algorithm that can be used to compute the shortest distance between two convex polytopes and it can also be modified to determine a separating axis. The GJK algorithm can also be used for implicit surfaces like cones, cylinders and spheres (see [van den Bergen 2004] and [van den Bergen 1999a]). The GJK algorithm can also be combined with the EPA algorithm (Expanding-Polytope Algorithm) to determine penetration depth [van den Bergen 2004], which is useful by a collision response part of a physics engine. The GJK algorithm is used in the open-source physics engine Bullet (Bullet homepage) and in the collision detection library SOLID (Software Library for Interference Detection: SOLID homepage).

A general description of the algorithm will now be given. For the mathematics see [Gilbert et al. April 1988]. The algorithm determines the shortest distance between two convex objects A and B by forming the Minkowski sum $M = A + (-B)$:

$$M = \{a - b : a \in A, b \in B\} \tag{7}$$

Then the shortest distance is found from the origin to M. An intersection has occurred if the *origin is contained in M*. The Minkowski sum can be seen as reflecting B to get -B and then -B's reference point is *swept* along A's surface. See figure 6. Notice that the distance (d) in upper left figure is the same as the distance in the lower figure.

As with the SAT algorithm, the GJK algorithm needs to find a point, which has the property:

$$S_C(v) \in C \; such \; that \; v \cdot S_C(v) = max\{x \cdot v : x \in C\} \tag{8}$$

given that v is a direction vector. Support mapping of the Minkowski sum for object A and B are defined as (for convex objects):

$$S_{A-B}(v) = S_A(v) - S_B(-v) \tag{9}$$

The support point (or vertex) for a *polytope* is defined as:

$$S_A(v) \in vert(A), \; v \cdot S_A(v) = max\{x \cdot v : x \in vert(A)\} \tag{10}$$

See figure 7 for an execution of the algorithm on an example. A remark is that the simplices that can be formed in 2D are: 0-simplex (point), 1-simplex (bounded line) and 2-simplex (triangle). In 3D, the simplices are the same as for the 2D case, plus the 3-simplex (tetrahedron). The main idea is to search for a shortest distance on simplices, and because of convexity the algorithm will converge to a point on M, which has the shortest distance to the origin. This distance is the same as the shortest distance from A to B. See algorithm 2 (taken from [Van den Bergen 1999b]) to compute the shortest distance. In algorithm 2, $conv(S)$ is the *convex hull* of the set S and $v(S)$ is the vector from O to the closest point on simplex S.

The difference $||v|| - \mu$ is a threshold to deal with floating point imprecision. In line 9 of algorithm 2, the set $(W \cup \{\mathbf{w}\})$ will contain at most d+1 vertices from M (triangle in 2D and tetrahedron in 3D).

In line 11 of algorithm 2, one have to find a vector v from O to the convex hull of the union of W (vertices forming simplex) and a new support point **w**. This part of the algorithm is called the *distance sub-algorithm* in [Johnson 1987] and [Gilbert et al. April 1988]. This problem can be solved algebraically or geometrically. In [Gilbert et al. April 1988], it is solved algebraically by solving a *system of linear equations*. The idea is that the vector from the

Algorithm 2 GJK

1: $W \leftarrow \emptyset$
2: $\mathbf{v} \leftarrow$ arbitrary point in M
3: $\mu \leftarrow 0$
4: $close\_enough \leftarrow$ **false**
5: **while** not $close\_enough$ and $\mathbf{v} \neq \mathbf{0}$ **do**
6: $\quad \mathbf{w} \leftarrow S_M(-\mathbf{v})$
7: $\quad \delta \leftarrow \mathbf{v} \cdot \mathbf{w}/||\mathbf{v}||$
8: $\quad \mu \leftarrow max(\mu, \delta)$
9: $\quad close\_enough \leftarrow ||\mathbf{v}|| - \mu \leq \epsilon$
10: $\quad$ **if** not $close\_enough$ **then**
11: $\quad\quad \mathbf{v} \leftarrow v(conv(W \cup \{\mathbf{w}\}))$
12: $\quad\quad W \leftarrow$ smallest $X \subseteq (W \cup \{\mathbf{w}\})$ such that $\mathbf{v} \in conv(X)$
13: $\quad$ **end if**
14: **end while**
15: **return** $||v||$

origin to p is perpendicular to the affine hull of X. This defines a system of linear equations, which is solved to obtain p. This p can then be expressed with barycentric coordinates of the simplices. If these coordinates do not sum up to one, p will not be on the convex hull of simplices. Trying to solve the problem geometrically requires the use of voronoi regions (see section 5.2.1 and [Ericson 2005]) to find a feature on the simplex that gives the shortest distance.

In [Gilbert et al. April 1988], searching for a support point for a given direction is a linear search of vertices from A and B giving a linear complexity. In sections 5.2.3 and 5.2.3 this can be improved. This improvement together with improvements from 5.2.3 and 5.2.3 will give an empirically almost constant running time.

Enhanced the GJK algorithm Further improvements of the GJK algorithm can be made. In [Cameron 1997] [Van den Bergen 1999b], improvements are presented.

Hill climbing can used for convex polyhedron to speed up support point searching: The support point from the last iteration, together with its neighbouring vertices are tested against a new direction to look for a new support point. If no neighbouring vertices are better than this vertex a support point is found. If conversely, some neighbouring vertices gives a better result compared to this vertex, the neighbouring vertex that gives the best result (among other neighbours) is chosen as the best support point for this iteration. In the worst case, the algorithm has to search through all vertices, but this is very unlikely in practice.

Another improvement is to reduce the number of valid subsets of W when looking for X. Because one knows that the new subset X will contain $\mathbf{w}$, the number of subsets to search for are reduced [Van den Bergen 1999b].

ISA-GJK Bergen in [Van den Bergen 1999b] presented yet another improvement of the GJK algorithm: The Incremental Separating Axis-GJK (ISA-GJK) algorithm. The algorithm is modified to look for a separating axis instead of calculating the shortest distance. See algorithm 3, which is a restatement from the report.

The test on line 5 for algorithm 3 is a separating axis if it holds (which means that the origin is not contained in M).

As with the SAT algorithm, the Lin-Canny algorithm, the V-Clip algorithm and the Chung-Wang Separating vector algorithm, temporal coherency can be exploited to get an almost constant running

Algorithm 3 ISA-GJK

1: $\mathbf{v} \leftarrow$ arbitrary vector
2: $W \leftarrow \emptyset$
3: **repeat**
4: $\quad \mathbf{w} \leftarrow S_M(-\mathbf{v})$
5: $\quad$ **if** $\mathbf{v} \cdot \mathbf{w} > 0$ **then**
6: $\quad\quad$ **return false**
7: $\quad$ **end if**
8: $\quad \mathbf{v} \leftarrow v(conv(W \cup \{\mathbf{w}\}))$
9: $\quad W \leftarrow$ smallest $X \subseteq (W \cup \{\mathbf{w}\})$ such that $\mathbf{v} \in conv(X)$
10: **until** $\mathbf{v} = \mathbf{0}$
11: **return true**

time [Van den Bergen 1999b]. As with the SAT algorithm, the separating axis from previous frame can be used for the current frame as an initialization of the vector v. This version is faster because the length of v is no longer needed, removing a square-root calculation. Bergen performed experiments on the performance of the ISA-GJK algorithm, compared to the Lin-Canny algorithm and the ISA-GJK algorithm was roughly five times faster. Also the ISA-GJK algorithm detected a collision which the Lin-Canny algorithm missed.

5.3 Non-convex Collision Detection

Algorithms presented above considered objects to be convex. In a simulation, objects can also be non-convex. An object O is *non-convex* if there exist a point p on a bounded line $a(1-t)+tb$, where $a, b \in O$ and $t \in \{0, 1\}$, such that $p \cap O = \emptyset$.

A method to make a collision detection system that support non-convex objects, are to decompose the objects into convex parts and run algorithms suggested in 5.2 to do a more exact collision detection of the convex parts. Below are some algorithms that do this.

Convex decomposition of non-convex objects One method to make collision detection support non-convex objects is to decompose the object into convex parts. By doing that, collision detection will be between convex parts, and algorithms presented in 5.2 will still work.

Another approach is to use bounding volume hierarchies to enclose primitives of the polyhedron down to primitive level. Each leaf node will consist of a primitive (in this case a triangle), which is convex.

A method used in [Quinlan 1994] uses a hierarchy of *bounding spheres* to decompose a non-convex polyhedron into convex parts. The method uses the divide and conquer paradigm to build the hierarchy. The algorithm starts by enclosing every triangle into bounding spheres. These spheres will be the leaves of the bounding sphere hierarchy. After doing this, two sets of spheres are created. The sets are created by using the local coordinate system of the polyhedron and find the mean of the longest axis and then divide the spheres into two parts, using the centers of the spheres. The algorithm recursively divides spheres into two smaller sets containing spheres, until one sphere is left. Then the algorithm creates a parent node (bounding sphere that encloses the two sets) and do this recursively. The root node will be a sphere containing all leaves of the tree and an internal node is a sphere that entirely encloses its children. The same can be applied to other bounding volumes to create hierarchies.

Another approach is presented in [Ponamgi et al. 1995]. Here a polyhedron can be divided into: *Hull features*, *concavity features* and *boundary features*. A feature is part of a *hull feature set* if the

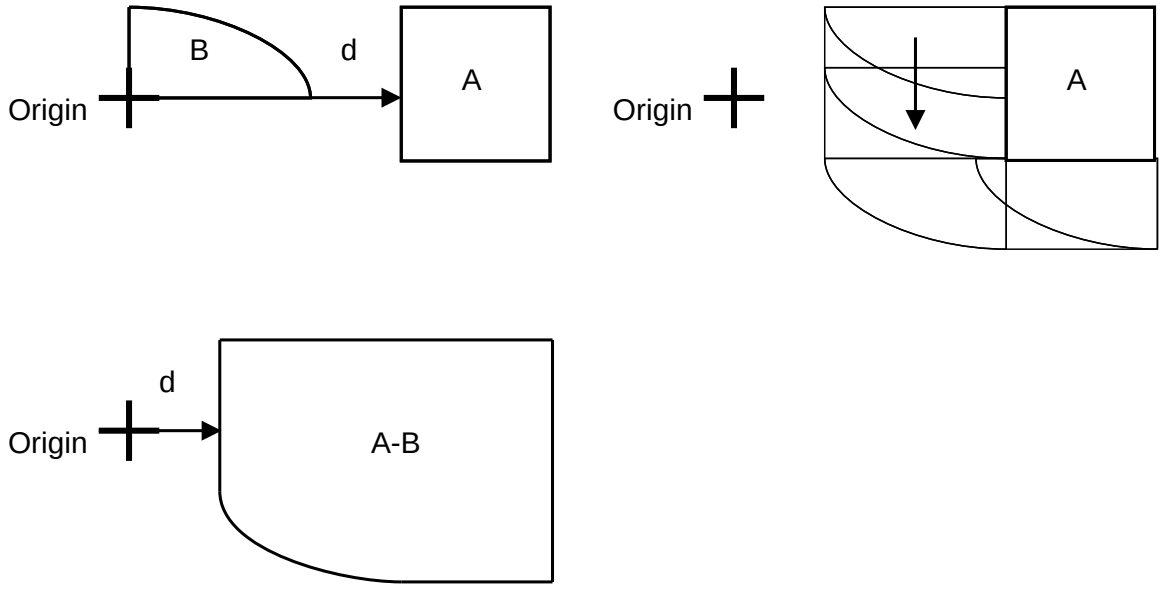


Figure 6: *Upper left: A and B are separated from each other. Upper right: -B is swept on A. Lower: The Minkowski sum A-B is formed and the shortest distance is d (same as upper left figure).*

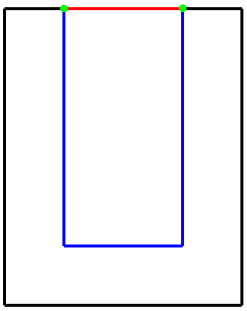


Figure 8: *Red part is the 'cap' of the concave polygon. Blue parts are the features belonging to the concavity and black parts together with the red part are the features on the convex hull. The green vertices belong to the boundary feature set.*

feature also exists in the features of the convex hull of the polyhedron. A feature is part of a *concavity feature set* if it is not part of the convex hull. Vertices and edges that are in the border of hull features and concavity features is part of the *boundary feature set*. To distinguish which parts of the convex hull that belongs to the polyhedron, and which parts that are only part of the convex hull and not part of the polyhedron, so called "caps" are created.

The idea is similar to the previous approach but the main difference is that the convex hull of an object is determined and then 'caps' are created to distinguish where the polyhedron is concave. The cap can be seen as an area on the convex hull where the convex hull is not part of the polyhedron. See figure 8.

To detect collisions on the concavity feature sets, a hierarchy of axis aligned bounding boxes are created. Now if a penetration of the convex hull is detected, then a collision has occurred, but if instead, an intersection of a cap is detected, further checking has to be made. In this case the AABB hierarchy is traversed, and at each level of the hierarchy, a hierarchical sweep and prune (see section 5.1.2 for sweep and prune) is used to detect which AABBs are intersecting. As an end result pairs of potentially colliding primitives are tested for exact collision detection by using for example the extended Lin-Canny algorithm (see section 5.2.1)

6 Results and Discussion

The algorithms were implemented on an AMD X2 5600+ CPU, 4 GB memory, with Windows 7 64-bit, Visual Studio 2008, and on an NVIDIA GeForce GTX 470 GPU using CUDA.

6.1 Broad-phase Collision Detection

The chosen algorithm is a spatial subdivision with sweep and prune to cull away objects.

6.1.1 Implementation Details

The spatial subdivision part is similar to [Le Grand 2007]. The main difference is that in this thesis, support for arbitrary sized objects is added similar to [Kalojanov and Slusallek 2009] by performing a pre-pass to count the number of cells an AABB (chosen as bounding volume) intersects and then perform a prefix sum to determine the offset for threads to start writing AABB cell intersections. Also, the best sweeping axis is chosen along x-,y-, or z-axis by calculating the variance of the AABB centers and pick the axis with highest variance as best sweeping axis. This calculation can be neglected if one has prior knowledge of the scene, such that the objects are uniformly distributed in the scene and moves randomly. If this is the case it is better to randomly pick one of the three coordinate axes as best sweeping axis.

In this thesis, only one level of subdivision is performed. This is simple enough and works well. The parallel sweep and prune algorithm from [Liu et al. 2010] is called on the best sweeping axis, so subdivision will be performed only on the other two axes, leading to a 2D spatial subdivision. Each AABB is then processed to determine which cells it intersects and writes that to an array, which will be sorted. After performing the radix sort, objects intersecting the same cells will be grouped together and one has to perform parallel sweep and prune on them. Before the parallel sweep and prune

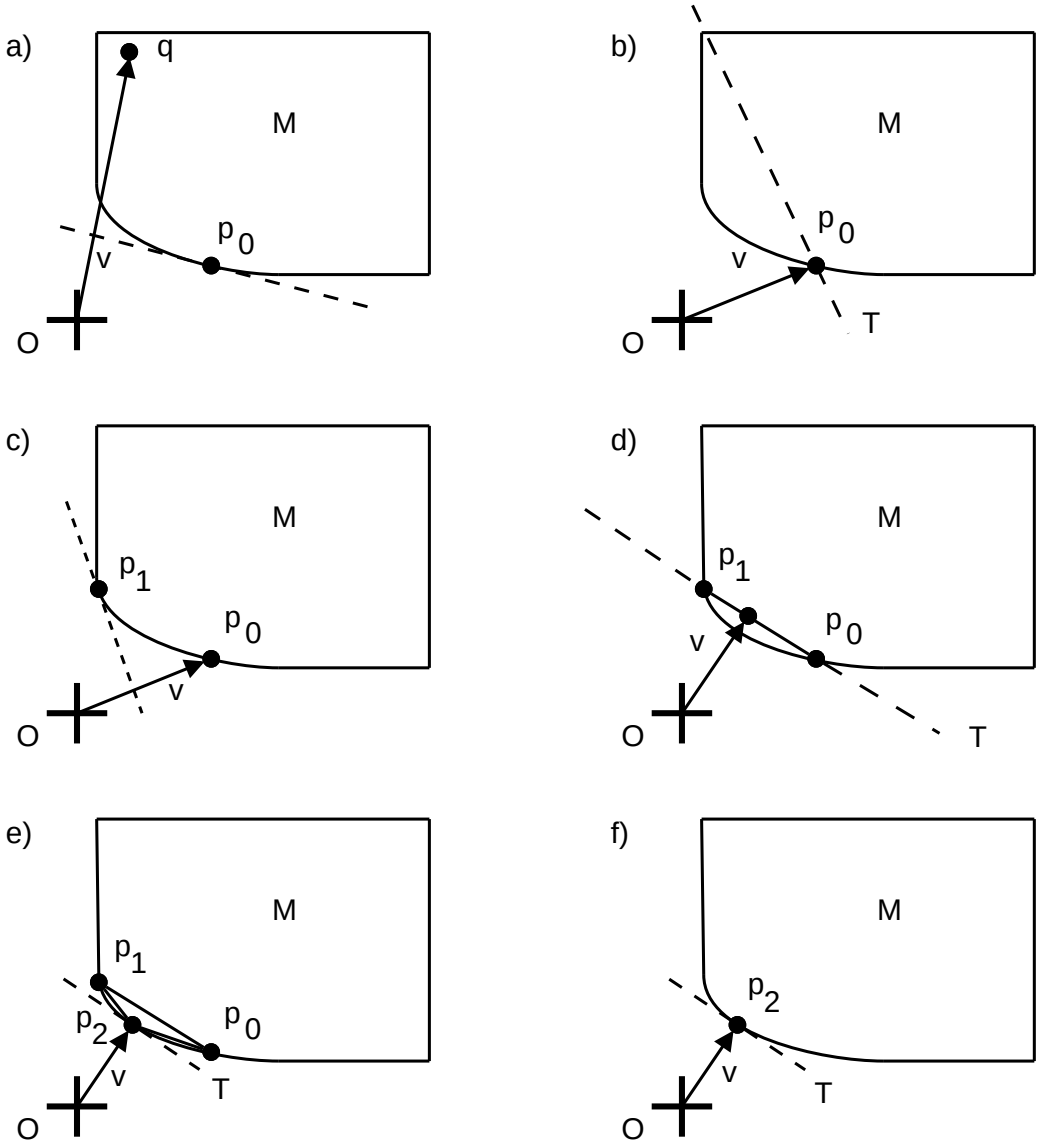


Figure 7: *a) An arbitrary point q inside M is chosen and $p_0 = S_M(-v)$ is found. b) $G = \{p_0\}$ and vector v from O to p_0 is formed. The plane T (at p_0 with normal v) still has points from M in both half spaces. c) $p_1 = S_M(-v)$ is found and $G = \{p_0, p_1\}$. d) A point that gives the shortest distance $||v||$ is found. The plane T still has points from M in both half spaces. e) Do $p_2 = S_M(-v)$ and $G = \{p_0, p_1, p_2\}$. In e) all points of M is on, or on the other side of T. So in f) the shortest distance between A and B are $||v||$. Equivalently: In f), getting a new support point in direction $-v$ will return p_2 (the same point from last iteration).*

algorithm is invoked, one has to invalidate cells that only contain one AABB. After this, all objects are shifted along the best sweeping axis. The amount shifted is determined by the size of the scene along the best sweeping axis multiplied with the cell id to ensure complete separation between cells.

In this thesis, an upper bound on maximum number of collision pairs that a specific AABB can have is determined. This upper bound is derived from the number of AABBs there are in a cell. By knowing this, the number of maximum potential collisions in that cell is determined by a geometric sum. This worst case upper bound guarantees the generality of the algorithm, but does not work so well in practice because this worst case upper bound is never to be reached. With the limited amount of memory on the graphics hardware and for larger scenes, there is not enough memory to allocate for such an array. Instead one can set a more empirical value on the number of maximum collisions that one AABB can have.

After the shifting, parallel sweep and prune is performed and objects that are shown to intersect along best sweeping axis are further tested for intersection in the other two axes (looking for a separating axis). If intersection has occurred, the AABB pairs are written to a collision pair array. This array is then used by the ISA-GJK algorithm to perform more exact collision detection.

Spatial subdivision and the parallel sweep and prune were implemented using the Thrust library [Hoberock and Bell 2010]. In an earlier implementation, an own prefix sum and radix sort algorithm were implemented (based on [Sengupta et al. 2008] and [Satish et al. 2009] along with the fast broad-phase algorithm, but profiling showed that the radix sort was the bottleneck of the application, so a change to the Thrust library was made due to the fact that Thrust has Duane Merrill's fast radix sort algorithm [Merrill and Grimshaw 2010]. Thrust is a good library to implement algorithms fast, but makes it hard to know which kernel calls actually invoke the code that oneself has written.

6.2 Narrow-phase Collision Detection

The chosen algorithms were GJK and ISA-GJK. The Lin-Canny algorithm is not chosen because Mirtich in [Mirtich 1998] showed that it was slower than the V-Clip algorithm. Also in [Van den Bergen 1999b] Bergen showed that the ISA-GJK algorithm was five times faster than the Lin-Canny algorithm. So the V-Clip algorithm is a better alternative than the Lin-Canny algorithm. The Chung-Wang Separation Vector algorithm is also a good candidate and very fast. It was shown in [van den Bergen 2004] that the Chung-Wang Separating Vector algorithm was faster than the ISA-GJK algorithm on average, but the ISA-GJK algorithm has a tighter worst case bound. So the ISA-GJK algorithm is preferred when considering this. The SAT algorithm is not chosen because it performs poorly for highly tessellated objects, because of the increased amount of potentially separating axes. In worst case all these axes have to be checked. Also the GJK algorithm and the ISA-GJK algorithm have support for general convex objects. Considering this the GJK algorithm and the ISA-GJK algorithm are a better choice than the V-Clip algorithm (which only works on polyhedra).

6.2.1 Implementation Details

There are several ways to implement the ISA-GJK and GJK algorithms. Instead of doing as in [Van den Bergen 1999b], one can start by sampling four different directions to get four support points, forming a tetrahedron. This is done before entering the loop. It should not be a big performance difference by choosing one or the other.

Hill climbing is used to speed up the search for a support point. In [Van den Bergen 1999b], hill climbing integrated into the Dobkin-Kirkpatrick hierarchy [Dobkin and Kirkpatrick 1990], obtain an expected worst case support point search of $O(log(n))$. In [Ericson 2005], hill climbing can be speeded up by adding artificial neighbours. Artificial neighbours also remove the possibility of getting an infinite loop when looking for a support point (see figure 9 for an explanation). Using hill climbing with artificial neighbours might be slower for highly tessellated convex polyhedron, but more time-saving to implement.

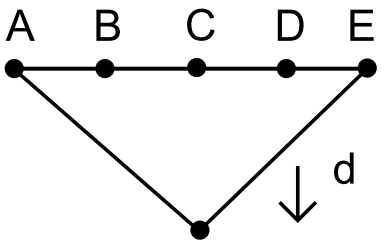


Figure 9: *Example where a support point search given a direction d can fail. For example when starting from C and picking D as the next best vertex, D can pick C again and this can be repeated, and one can get an infinite loop. In this case a constraint can be added, saying that one cannot visit a vertex that has already been visited. In the 2D case this will work out, but in 3D this can lead to closing into a vertex, making it impossible to pick a neighbouring vertex because they were chosen already in some previous iteration. In this case the algorithm fails. If an artificial neighbour is added, which is not coplanar to any neighbour, this problem can be solved and finding a support point can also be speeded up.*

Another important remark considers the simplex solver in the GJK algorithm. Bergen uses Johnson's distance algorithm which sets up a system of linear equations and solves it by using Cramer's rule, getting a point on the simplex giving the shortest distance to the origin. Bergen claims that this is the most robust way of solving the shortest distance problem from the origin to a point on a simplex set. In [Ericson 2005], a voronoi simplex solver is used instead. By looking at the voronoi regions of the simplex set one can determine which feature will give the shortest distance to the origin. This method has the advantage that one will see the problem in a geometric way, simplifying coding and debugging. Bergen's method might be faster but is harder to understand.

Another remark is on the implementation details of the voronoi simplex solver, mentioned in [Ericson 2005]. Ericson checks all voronoi regions, but this is actually not needed. In [Van den Bergen 1999b], theorem 2 states that the newly added support point for this iteration, will be in simplex set X. Therefore this newly added support point must be needed for the shortest distance calculation in the next iteration of the algorithm. This reduces the number of voronoi regions tests needed to look for. In the case of a tetrahedron, [Ericson 2005] will result in fourteen voronoi region checks (4 vertices, 4 faces, and 6 edges) but using theorem 2 from [Van den Bergen 1999b], this will be reduced to seven region checks (1 vertex, 3 faces, and 3 edges), half of the regions.

In this thesis, the better voronoi simplex solver (using theorem 2 from [Van den Bergen 1999b]) is implemented.

The GJK algorithm and the ISA-GJK algorithm can also be implemented on the GPU. Because the GJK algorithm relies heavily on support point search in every iteration and maintaining a neighbouring map to perform hill climbing on, it is hard to make an efficient implementation for the GPU. Also the ISA-GJK algorithm and the GJK algorithm are inherently serial algorithms, so parallelizing it

is not a trivial task. There are methods to do this. The biggest concern is whether the convex objects fits the GPU's on-chip shared memory. If it does, support mapping can be implemented by performing a dot product on all vertices with a direction and then store it on shared memory and then perform an on-chip max-prefix sum calculation to find a support vertex. If the convex object does not fit in shared memory, one have perform a prefix sum after writing the result of the dot product into the slower device memory.

Another method to port the GJK algorithm to the GPU is to create a cube map of a convex object and then sample from it [Sathe and Lake 2006]. The method uses the texture memory of the GPU to sample for a support vertex. The cube map is created by having six textures, one for each face of the cube. This cube can be seen as a virtual AABB. This method only works for rigid objects. The virtual AABB encloses the object in such a way that the object can have any orientation. Each face of the cube map is divided into cells. If the texture is of resolution s x t, there will be s cells along one axis and t cells along the other axis. These cells are mapped to a texture. When all cells for six virtual AABBs are created, rays are shot from the center of the object to the center of these cells, and the hit-point on the object is calculated and stored in the six textures.

The texture memory is cached and reads with locality are fast. When a support vertex for a given direction is desired, the given direction is mapped to one of the cube map's face, and a corresponding texel is looked up.

An important remark of this method compared to a serial implementation, is that it is approximate. It is approximate because it shoots rays from the center of the polyhedron to a cell, and if the grid size has to low resolution, some faces of the polyhedron will be missed.

The ISA-GJK algorithm is chosen as the narrow-phase collision detection algorithm because it requires less iterations in practice compared to the GJK algorithm [Van den Bergen 1999b] (due to the case of early termination when a separating axis is found). The GJK algorithm is the preferred algorithm when there is a collision response part after narrow-phase collision detection, because the GJK algorithm can be used to determine contact point(s).

6.3 Experiment

In this section, the whole collision detection system is benchmarked on scenes consisting of different amount of objects. The scene consists of a certain amount of convex objects and they are uniformly distributed and are assigned a random velocity vector. The number of cells are chosen to be $\lceil \frac{N}{64000} \rceil$, which is from [Liu et al. 2010] and works well for this scene. The scene consist of triangulated spheres with low tessellation (80 triangles), which does not affect the broad-phase collision detection algorithm's performance because in the broad-phase only AABBs are considered. Figure 10 shows the timing when cell size varies with the number of objects versus fixed cell size.

The choice of cell size is not trivial. The broad-phase does not perform so well with to small cell sizes. As seen from 10, a cell size that varies with the amount of objects in the scene, gives better performance. The fixed cell size version of the benchmark used a cell size of three in x-,y-, and z-axis and objects are triangulated spheres with a radius of one. The fixed cell size does not depend on the number of objects in the scene. With this setting all AABBs can intersect a maximum of four cells. The faster timing graph has a cell size that varies depending on the number of objects in the scene and as shown from the graph the varying cell size configuration is faster than the fixed cell size. A reason for this, is the increased number of objects that straddles between cells for the smaller cell size. When objects straddle between cells, they must be duplicated when workspace shifting is performed to ensure that collisions are not missed. More clearly: An object straddling between cells has a world space coordinate and when workspace shifting is performed this AABB must be shifted with different values along the best sweep axis, which will create two or more AABBs which are actually enclosing the same object.

Next, the narrow-phase collision detection algorithm ISA-GJK is benchmarked. In this thesis a narrow-phase collision detection algorithm is called for potentially colliding pairs (which the broad-phase collision detection algorithm finds), so it suffices to test the performance for one pair of object. Figure 11 shows timing of the ISA-GJK algorithm when hill climbing is enabled and disabled. It is clearly shown in the figure that hill climbing speeds up the algorithm a lot. When no hill climbing is performed a linear search on the number of vertices is performed and as triangle count rises, the ISA-GJK algorithm takes more time to find a support vertex. The reason that the intersected state takes more time than the non-intersecting state is because of more iterations of the while loop (see algorithm 3).

Timing for the whole system can be seen in figure 13. One can see that the parallel part scales well, but ISA-GJK does not. This is due to the increased amount of potentially colliding pairs that the broad-phase returns (see figure 12).

7 Conclusion

In this thesis, a two-phase collision detection pipeline is implemented and benchmarked. Fast large-scale collision culling of tens of thousands of objects are achieved by executing parallel algorithms on the GPU. Also fast exact collision detection algorithms are implemented on the CPU to find a separating axis between two convex objects.

8 Future Work

The implemented broad-phase algorithm is a simplified version of the one discussed in [Liu et al. 2010], so adding methods from that report will be interesting, just to see how much of an improvement one can get. In this thesis the Thrust library is used to enhance productivity. Thrust is an excellent library for this purpose, but sometimes it feels inflexible. There are no possibilities to fine-tune the code. For example: Launch configurations cannot be changed and support for CUDA streams and asynchronous CUDA functions are currently not supported. So future work will remove the dependency to Thrust and replace it with own written code.

For the narrow-phase, an improvement of support vertex search can be made. Instead of simple hill climbing, one can implement a Dobkin-Kirkpatrick hierarchy [Dobkin and Kirkpatrick 1990] integrated with hill climbing to further speed up support vertex searching. Another improvement would be to add artificial neighbours to avoid termination problems when performing hill climbing.

Also implementing the GJK algorithm on the GPU can make an improvement to the collision detection pipeline, because less data needs to be transferred back to the CPU.

A big improvement would be to add mid-phase collision detection to the collision detection pipeline, or more specifically adding a parallel bounding volume hierarchy [Lauterbach et al. 2009] . By adding such a phase, support for deformable meshes follows. It is then unclear, whether the GJK algorithm is a good candidate for the narrow-phase, because at the bottom level of the hierarchy will be triangles, so a good narrow-phase algorithm candidate would be a fast triangle-triangle intersection/distance test. This choice boils

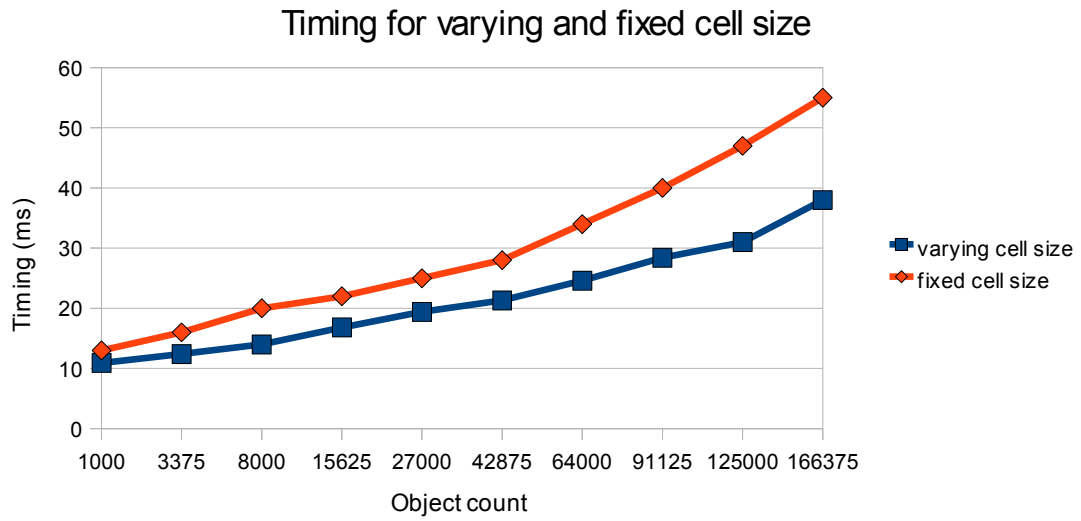


Figure 10: *Timing for different number of object counts. Notice the non-linear x-axis, which is cubic.*

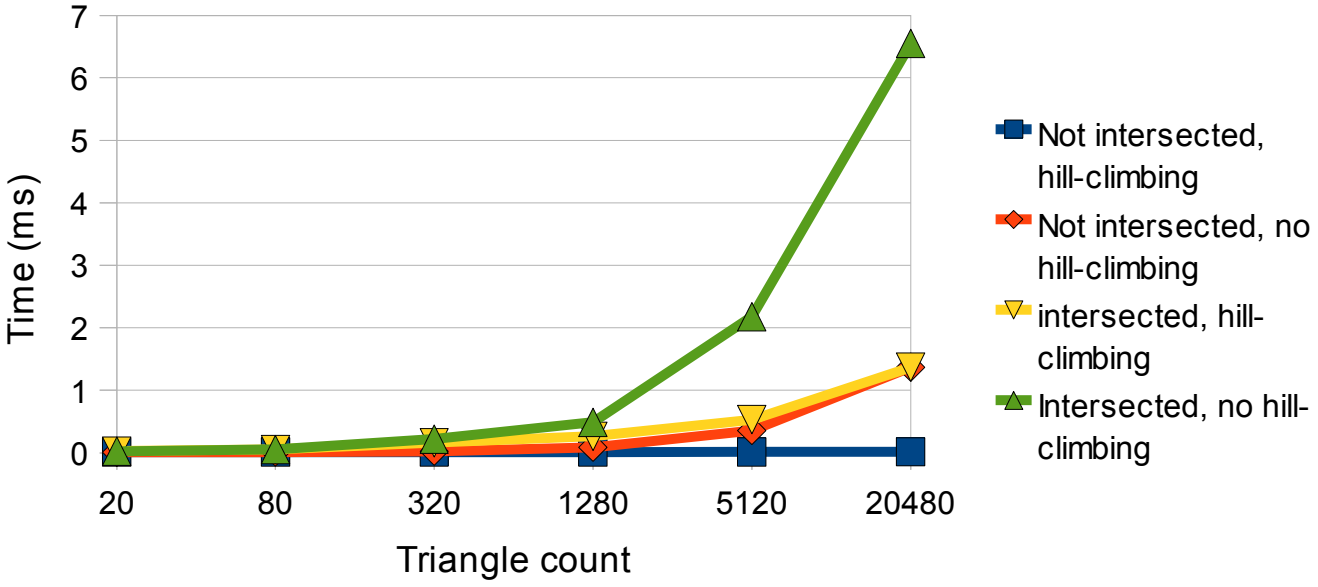


Figure 11: *Timing for the ISA-GJK algorithm.*

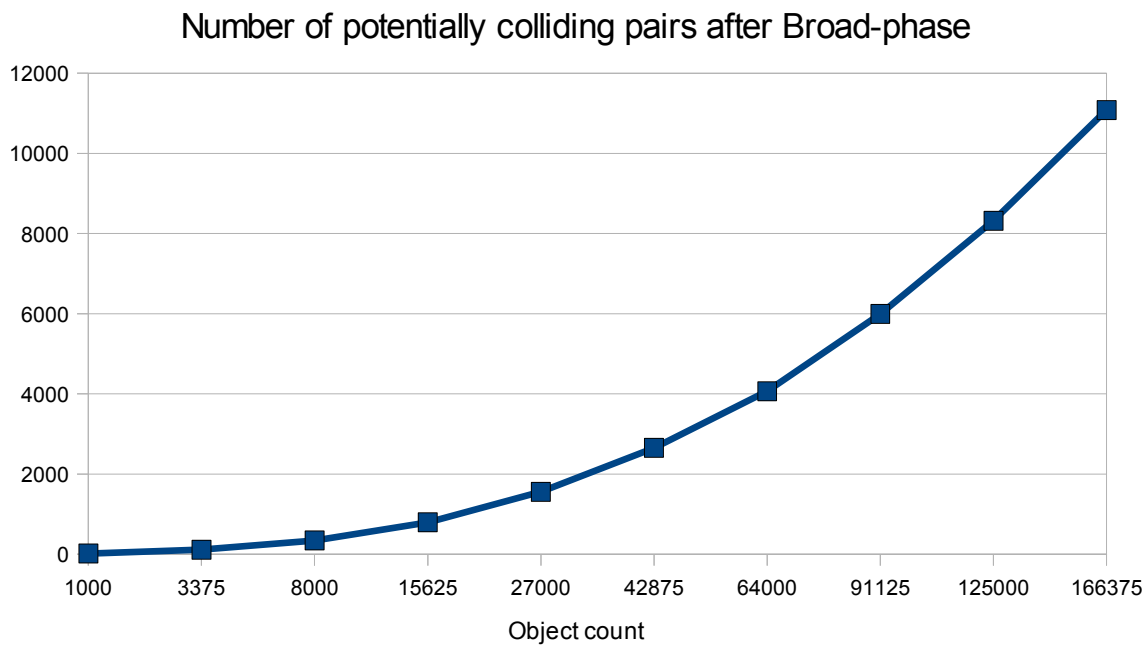


Figure 12: *Potentially colliding pairs that broad-phase outputs with different object counts.*

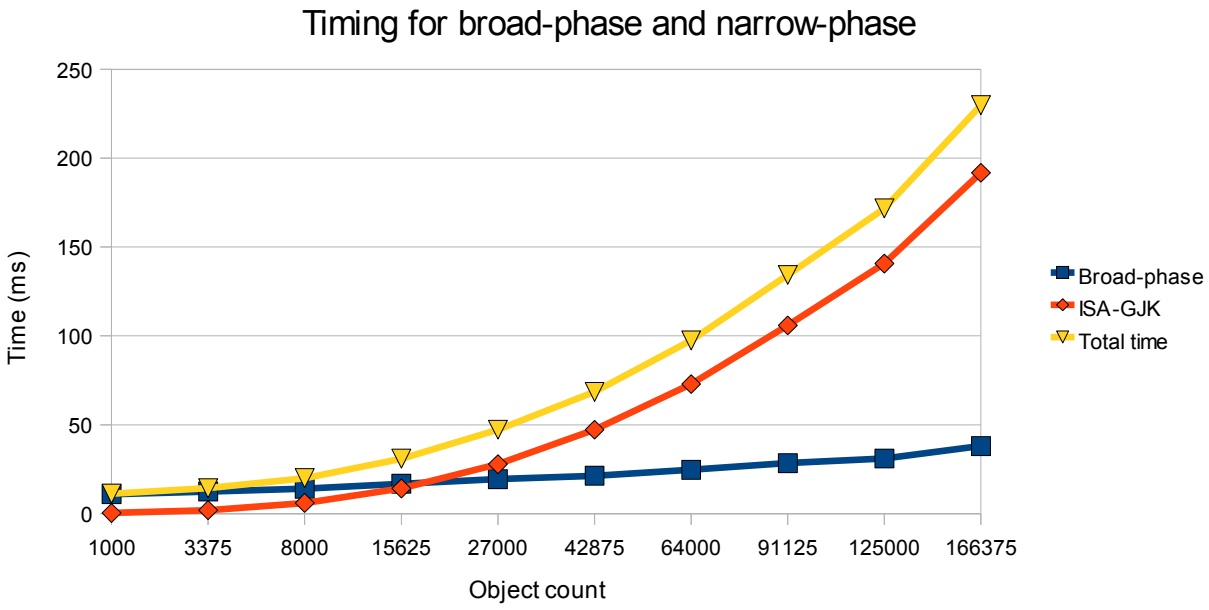


Figure 13: *Timing for broad-phase and narrow-phase collision detection.*

down to whether the GJK algorithm is faster than more specialized triangle-triangle algorithms when primitives are triangles instead of whole convex objects.

9 Acknowledgements

I want to thank Ulf Assarsson for his supervision. I also want to thank Fuchang Liu for discussions on his broad-phase collision detection algorithm, and the Thrust user group for helping out with Thrust related problems.

References

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

BLELLOCH, G. E. 1990. Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov.

CAMERON, S. 1997. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In *Proceedings of International Conference on Robotics and Automation*, 3112–3117.

CHUNG, K., AND WANG, W. 1996. Quick collision detection of polytopes in virtual environments. In *ACM Symposium on Virtual Reality Software and Technology 1996*, ACM, Hong Kong.

CHUNG, K., AND WANG, W. 1996. Quick elimination of non-interference polytopes in virtual environments. In *Proceedings of the Eurographics workshop on Virtual environments and scientific visualization '96*, Springer-Verlag, London, UK, 64–73.

CHUNG, K. 1996. An efficient collision detection algorithm for polytopes in virtual environments. Tech. rep., Department of Computer Science, University of Hong Kong.

COHEN, J., LIN, M. C., MANOCHA, D., AND PONAMGI, M. K. 1994. Interactive and exact collision detection for large-scaled environments. Tech. rep., ACM SIGGRAPH.

COHEN, J. D., LIN, M. C., MANOCHA, D., AND PONAMGI, M. K. 1995. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *In Proc. of ACM Interactive 3D Graphics Conference*, 189–196.

DELOURA, M. 2000. *Game Programming Gems*. Charles River Media, Inc., Rockland, MA, USA.

DOBKIN, D. P., AND KIRKPATRICK, D. G. 1990. Determining the separation of preprocessed polyhedra: a unified approach. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, Springer-Verlag New York, Inc., New York, NY, USA, 400–413.

DRAKE, P. 2005. *Data Structures and Algorithms in Java*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

EBERLY, D. H. 2003. *Game Physics*. Elsevier Science Inc., New York, NY, USA.

ERICSON, C. 2005. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann, January.

FUCHS, H., KEDEM, Z. M., AND NAYLOR, B. F. 1980. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 124–133.

FUCHS, H., ABRAM, G. D., AND GRANT, E. D. 1983. Near real-time shaded display of rigid objects. *SIGGRAPH Comput. Graph. 17*, 3, 65–72.

GILBERT, E., JOHNSON, D., AND KEERTHI, S. April 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. IEEE, vol. 4 (2), 193–203.

GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D. 1996. OBB-Tree: a hierarchical structure for rapid interference detection. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 171–180.

HOBEROCK, J., AND BELL, N., 2010. Thrust: A parallel template library. Version 1.3.0.

JOHNSON, D. W. 1987. *The optimization of robot motion in the presence of obstacles*. PhD thesis, Ann Arbor, MI, USA.

JOLLIFFE, I. T. 2002. *Principal Component Analysis*. Springer, New York, NY, USA.

KALOJANOV, J., AND SLUSALLEK, P. 2009. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, HPG '09, 23–28.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. *Comput. Graph. Forum 28*, 2, 375–384.

LE GRAND, S. 2007. *GPU Gems 3: Broad-Phase Collision Detection with CUDA*. Addison-Wesley Professional.

LIN, M. C. 1993. Efficient collision detection for animation and robotics. Tech. rep.

LIU, F., HARADA, T., LEE, Y., AND KIM, Y. J. 2010. Real-time collision culling of a million bodies on graphics processing units. *ACM Trans. Graph. 29* (December), 154:1–154:8.

LUQUE, R. G., COMBA, JO A. L. D., AND FREITAS, C. M. D. S. 2005. Broad-phase collision detection using semi-adjusting BSP-trees. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 179–186.

MELAX, S. 2000. Dynamic plane shifting BSP traversal. In *Graphics Interface*, 213–220.

MERRILL, D., AND GRIMSHAW, A. 2010. Revisiting sorting for GPGPU stream architectures. Tech. Rep. CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA.

MIRTICH, B. V. 1996. *Impulse-based dynamic simulation of rigid body systems*. PhD thesis.

MIRTICH, B. 1997. Efficient algorithms for two-phase collision detection. Tech. rep.

MIRTICH, B. 1998. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics 17*, 177–208.

NAYLOR, B. 1993. Constructing good partitioning trees. In *Graphics Interface*, 181–191.

PONAMGI, M., MANOCHA, D., AND LIN, M. C. 1995. Incremental algorithms for collision detection between solid models. In *SMA '95: Proceedings of the third ACM symposium on Solid*

modeling and applications, ACM, New York, NY, USA, 293–304.

QUINLAN, S. 1994. Efficient distance computation between non-convex objects. In *In Proceedings of International Conference on Robotics and Automation*, 3324–3329.

SATHE, R., AND LAKE, A. 2006. Rigid body collision detection on the GPU. In *ACM SIGGRAPH 2006 Research posters*, ACM, New York, NY, USA, SIGGRAPH '06.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore GPUs. *Parallel and Distributed Processing Symposium, International 0*, 1–10.

SENGUPTA, S., HARRIS, M., AND GARLAND, M. 2008. Efficient parallel scan algorithms for GPUs. NVIDIA. Tech. rep.

TRACY, D. J., BUSS, S. R., AND WOODS, B. M. 2009. Efficient large-scale sweep and prune methods with AABB insertion and removal. In *VR '09: Proceedings of the 2009 IEEE Virtual Reality Conference*, IEEE Computer Society, Washington, DC, USA, 191–198.

VAN DEN BERGEN, G. 1999. *Collision Detection in Interactive 3D Computer Animation*. PhD thesis.

VAN DEN BERGEN, G. 1999. A fast and robust GJK implementation for collision detection of convex objects. *J. Graph. Tools 4* (March), 7–25.

VAN DEN BERGEN, G. 2004. *Collision Detection in Interactive 3D Environments*. Elsevier Science Inc.

WITKIN, A., BARAFF, D., AND KASS, M., 2001. Physically based modeling course notes.