IT 10 009
Examensarbete 30 hp
April 2010

Real-time Collision Detection with Implicit Objects

Leif Olvång

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Real-time Collision Detection with Implicit Objects

Leif Olvång

Collision detection is a problem that has been studied in many different contexts. Lately one of the most common context has been rigid multi-body physics for different types of simulations.

A popular base algorithm in this context is Gilbert-Johnson-Keerthi's algorithm for measuring the distance between two convex objects. This algorithm belongs to a family of algorithms which share the common property of allowing implicitly defined objects.

In this thesis we give a theoretical overview of the algorithms in this family and discuss things to keep in mind when implementing them. We also give a presentation of how they behave in different situations based on our experiments. Finally we give recommendations for in which general cases one should use which algorithm.

Handledare: Anders Hast
Ämnesgranskare: Bo Nordin
Examinator: Anders Jansson
IT 10 009
Tryckt av: Reprocentralen ITC

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

Collision detection is a sub-field of computational geometry which deals with the problem of detecting and classifying intersections between objects. Given a set of n shapes the problem is to construct a list of shape pairs that have nonempty intersections with each other. Additionally the solution should include the information needed to resolve each intersection. In many applications the problem should be solved for a simulation that is integrated through time, and thus falls into the the set of problems normally called dynamic problems. This can make the task easier since it lets us use temporal coherence to optimize our algorithms, but it also makes the problem harder since the simulation needs to run in real time which adds significant time constraints to the problem. Collision detection can be studied at many different levels of detail, as one normally uses several different algorithms and data structures to solve the original query given above. At the highest level the objects are many, and they can have any shape including complex nonconvex and quadratic shapes. At this level, one normally uses some kind of space partitioning algorithm combined with bounding volumes to prune the possible collision pairs. The most common algorithms for this level are different versions pf space partitioning trees such as Binary space partitioning trees (BSP-tree), Octrees, Quadtrees or K dimensional trees (Kd-trees). Another very popular algorithm to use is Sweep and Prune. After applying these kinds of algorithms we in each candidate intersection have just two objects that need to be tested. The objects can still be nonconvex or in other ways problematic. This level usually describes objects as hierarchies of simpler volumes, possibly with each shape encapsulated in a bounding volume. BSP-trees can be used for this type of convex decomposition, but which structures is used depends partly on how the lower levels of detection are handled. At the next level of detail we know that each object pair is convex, and possibly how it is represented. The algorithms this thesis deals with solve the problem at this level. Once we have solved the problem at the lowest level we propagate the results back up and combine them to a solution to the original problem.

Unlike most algorithms in computational geometry the problems that we deal with in collisions detection are not necessarily formulated in terms of vertexes or polygons. Although most collision detection algorithms deal with polygonal data this is not a strict requirement. We will examine a specific subset of collision detection algorithms, which have in common that they do not require knowledge of the internal representation of the object. All that is needed is that each object implements a common interface with the central operation being a method for querying the object for the point that is furthest in a given direction but still part of the object. This has the advantage that it lets us use the same algorithm for all objects instead of needing different algorithms for different types of objects. It also lets us divide the code into discrete sections that can be optimized individually. The main downside of being confined to a common interface is that since we do not know the specific types of shapes that are colliding, nor even their internal representation, we cannot use this information. Thus given enough time it should be possible to find a better algorithm for each object type and hence get a more efficient solution by utilizing information about the

object type and its internal representation. In practice this is impractical as the number of different configurations to optimize for quickly becomes intractable.

In this thesis we will do a survey of several algorithms that work with implicitly defined objects. To begin with we will present the theory behind each algorithm and discuss different solutions to sub-problems. We will also present a reference implementation of how the algorithms can be implemented. Finally we will use our reference implementation to construct a number of tests, present the results from said tests and use them to compare the different algorithms. The goal of the survey is to give guidance for in what cases one should use which algorithm, and illustrate common pitfalls when working with them.

1.1 Terminology

In this thesis we use terminology in a specific, but not necessarily standardized way. Therefore we now specify what we mean by certain words. The words ***volume***, ***shape***, and ***object*** are all used interchangeably to refer to a geometric description of a set of points in the space with which we are working. If it is not explicitly stated otherwise the point sets are always compact convex sets in $\mathbb{R}^3$. When referring to ***collision detection***, we mean using computational geometry to extract information about the spatial relation between two volumes, with the goal of computing the data needed to handle the collision in the specific problem. ***Intersection testing*** refers to a Boolean query that is true if and only if the intersection of two volumes is nonempty. With ***touching contact*** we mean two objects that have a nonempty intersection, but the volume of the intersection is zero. ***Penetration depth*** is used to refer to the shortest vector $\mathbf{v}$ such that if we translate A by $\mathbf{v}$ the objects A and B will be in touching contact; it should be noted that $\mathbf{v}$ is not necessarily unique. By ***contact normal*** we mean the normalized vector with the same direction as the penetration depth vector above. ***Collision plane*** is used to refer to a plane with a normal equal to the contact normal that splits the intersection of the two volumes. Finally ***contact manifold*** refers to an approximation of the intersection between the collision plane and the intersection of the two objects, usually approximated as a set of vertexes for a convex polygon.

We will also need a number of mathematical definitions. In the above definition of how we use the words shape, volume and object we mention the concept of a ***compact set***. When we use the word compact to describe a set we mean that the set is closed and bounded. Another definition we will need is that of an ***affine combination*** of a set, which has the following mathematical definition for a set X:

$$\mathbf{p} = \sum_{\mathbf{x}_i \in X} \lambda_i \mathbf{x}_i,\ 1 = \sum_{i=1}^{|X|} \lambda_i \tag{1}$$

The ***affine hull*** of a set X is the set of points that can be written as an affine combination of the vectors in X. For example the affine hulls of the different nondegenerate simplexes in $\mathbb{R}^3$ are: the affine hull of a point is the point itself, while the affine hull of a line segment,

is the line that the line segment is a part of. The affine hull of a triangle is the plane that contains the triangle, and finally the affine hull of a tetrahedron is all of $\mathbb{R}^3$.

The next definition we will need is that of a ***convex combination***, and the ***convex hull*** that it can generate. The mathematical definition of a convex combination of the set X is:

$$\mathbf{p} = \sum_{\mathbf{x}_i \in X} \lambda_i \mathbf{x}_i,\ 1 = \sum_{i=1}^{|X|} \lambda_i,\ \lambda_i \geq 0 \tag{2}$$

The convex hull of X is similarly the set of points that can be generated as convex combinations of X. An informal definition of the convex hull of a set is all the points in between the points in the original set. Therefore the convex hull of a convex set is the set itself, while for any nonconvex set it is the original set plus points that fill all the space between the points in the original set in such a way that it makes the new set convex.

A ***simplex*** is a set of points such that each point is affinely independent from all the others (i.e. it cannot be written as an affine combination of the other points in the set). A simplex in $\mathbb{R}^n$ can contain at most $n+1$ points. A subset of a simplex (also a simplex) will sometimes be referred to as a ***feature*** of the simplex. A ***polytope*** is a convex polyhedron.

1.2 Intersection Testing

The basic act of detecting intersections is the most generally applicable of the factors examined when doing collisions detection and can be used in many different kinds of systems. A few examples are physical simulations, as input to different types of decision-making systems such as triggers for AI code, computer graphics systems such as ray tracers, motion planning or automatic navigation systems (although they are probably more interested in how far apart two objects are than if they already collided). The two algorithms for solving this problem, that will be examined are: Gilbert-Johnson-Keerthi's algorithm (GJK)[1, 2], and Minkowski Portal Refinement (MPR) [3].

1.3 Contact Normal and Penetration Depth

The contact normal is not of interest quite as often as detecting intersections. It is mainly of interest in physical simulations of different kinds and in computer graphics. The penetration depth is more or less only of interest in physical simulations. We study three different strategies for contact normal computation: An adaptation of MPR that gives a heuristic estimate of the penetration depth, the Expanding Polytope Algorithm (EPA) [2], and a hybrid method based on GJK and EPA.

1.4 Common Points and Contact Manifold

A single point of intersection is needed in ray tracing for computing reflection rays and refraction rays. In physical simulations a contact manifold is needed to compute collision

constraints. The contact manifold generation algorithm we will use works through sampling the support mappings (see Section 2.2) of the two objects in the direction of the collision normal with a slight perturbation, followed by clipping the two polygons generated in this way against each other. The method is described in more detail in Section 2.6.

2 Theory

This section contains a theoretical presentation of the algorithms and the basic principles that we will use to implement the different collision queries. First we present mathematical operation of Minkowski addition that almost all of the algorithms have as their foundation. Then we introduce a method for implicitly defining our shapes called a support mapping. Finally we present each algorithm that we will use to solve our queries. Each algorithm is presented with a brief overview of how it works and what type of queries it primarily answers. Following this general overview, pseudo code and a more detailed description is presented, we also discuss what adaptations and additions can be used to extend each algorithm's basic function.

2.1 Minkowski Addition

All the algorithms in this thesis except for the contact manifold generation strategy are based on the mathematical operation ***Minkowski addition***[4]. Minkowski addition works on sets of vectors and is defined as $A \oplus B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A,\ \mathbf{b} \in B\}$. In other words the Minkowski sum of two sets A and B is a set that contains all vectors that can be constructed by adding a vector from A and a vector from B. The specific ***Minkowski sum*** used in collision algorithms is $A \oplus (-B) = A \ominus B = \{\mathbf{a} - \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$. This set has a number of interesting properties.

Primarily if the sets A and B have any point in common the origin will be in $A \ominus B$. If A and B do not have any points in common the origin will not be in $A \ominus B$.

The second useful property of $A \ominus B$, which is also easy to see, is that if the origin is not in $A \ominus B$ then the distance from the origin to $A \ominus B$ will be the distance from B to A.

The third useful property that is commonly exploited in collision algorithms is that if $\mathbf{0} \in A \ominus B$ then the shortest linear translation that will bring A and B into touching contact, is the inverted point on the boundary of $A \ominus B$ that is closest to the origin, if that point is seen as a vector. To understand this third property we need to consider how $A \ominus B$ is affected by a translation of A. All the vectors in $A \ominus B$ are based on vectors from A. If we apply a linear translation $\mathbf{v}$ to A, this vector will be added to all the points in A, but since all the points in $A \ominus B$ are the difference between vectors in A and vectors in B, therefore a translation of A will result in an equal translation of $A \ominus B$. So if we translate A by $-\mathbf{v}$ for any vector $\mathbf{v}$ on the boundary of $A \ominus B$ the origin will end up on the boundary of $A \ominus B$. The origin in $A \ominus B$ is the difference of all common points in A and B, and the boundary of $A \ominus B$ is constructed from points on the boundary of A and B. Thus if the origin is on the boundary of $A \ominus B$ then A and B will be in touching contact. Therefore a translation of A by a vector$-\mathbf{v}$ for any $\mathbf{v}$ on the boundary of $A \ominus B$ will bring A and B into touching contact, and the shortest such translation will be the shortest vector on the boundary of $A \ominus B$.

By utilizing the Minkowski addition $A \ominus B$ we can therefore simplify the problem of intersection testing, and penetration depth computation by computing them for $A \ominus B$ vs $\mathbf{0}$ instead of A vs B.

2.2 Support Mappings

Working directly with the Minkowski sum $A \ominus B$, is impossible since A and B contain an infinite number of points. To get around this problem we restrict our volumes to compact convex sets. With this limitation we could compute the boundary of $A \ominus B$ for two polyhedrons A and B explicitly by adding all vertexes in one polyhedron to all vertexes in the other and then taking the convex hull of these points. But this is not very practical as the number of points needed is $|A| * |B| = O(n^2)$ assuming $|A| \approx |B| = n$ and as computing convex hulls is an expensive operation. Additionally the convex hull method only works for polyhedrons and we want to be able to work with any convex shape. So instead of actually computing the set $A \ominus B$ explicitly we will construct it implicitly. This is done through a support mapping. A support mapping is a function $S_A : \mathbb{R}^n \to \mathcal{P}(A) \backslash \emptyset$ that given a direction $\mathbf{v}$ returns the points in the volume A that would be the first to come into contact with a plane with normal $\frac{\mathbf{v}}{\|\mathbf{v}\|}$ that moves from infinitely far away towards the center of A. Or expressed mathematically:

$$S_A(\mathbf{v}) = \{\mathbf{p} : \max\{\mathbf{v} \cdot \mathbf{a} : \mathbf{a} \in A\} = \mathbf{v} \cdot \mathbf{p}, \mathbf{p} \in A\} \tag{3}$$

Given such a support mapping for convex objects A and B, it is possible to construct a support mapping for $A \ominus B$ in the following way:

$$\begin{aligned}
S_{A \ominus B}(\mathbf{v}) &= \{\mathbf{x} : \max\{\mathbf{v} \cdot \mathbf{k} : \mathbf{k} \in A \ominus B\} = \mathbf{v} \cdot \mathbf{x},\ \mathbf{x} \in A \ominus B\} \\
&= \{\mathbf{x} : \max\{\mathbf{v} \cdot \mathbf{a} - \mathbf{v} \cdot \mathbf{b} : \mathbf{a} \in A \wedge \mathbf{b} \in B\} = \mathbf{v} \cdot \mathbf{x},\ \mathbf{x} \in A \ominus B\} \\
&= \{\mathbf{x} : \max\{\mathbf{v} \cdot \mathbf{a} : \mathbf{a} \in A\} - \min\{\mathbf{v} \cdot \mathbf{b} : \mathbf{b} \in B\} = \mathbf{v} \cdot \mathbf{x},\ \mathbf{x} \in A \ominus B\} \\
&= \{\mathbf{x} : \max\{\mathbf{v} \cdot \mathbf{a} : \mathbf{a} \in A\} - \max\{-\mathbf{v} \cdot \mathbf{b} : \mathbf{b} \in B\} = \mathbf{v} \cdot \mathbf{x},\ \mathbf{x} \in A \ominus B\} \\
&= \{\mathbf{p} - \mathbf{q} : \max\{\mathbf{v} \cdot \mathbf{a} : \mathbf{a} \in A\} - \max\{-\mathbf{v} \cdot \mathbf{b} : \mathbf{b} \in B\} = \mathbf{v} \cdot (\mathbf{p} - \mathbf{q}),\ \mathbf{p} \in A \wedge \mathbf{q} \in B\} \\
&= \{\mathbf{p} : \max\{\mathbf{v} \cdot \mathbf{a} : \mathbf{a} \in A\} = \mathbf{v} \cdot \mathbf{p},\ \mathbf{p} \in A\} \ominus \{\mathbf{q} : \max\{-\mathbf{v} \cdot \mathbf{b},\ \mathbf{b} \in B\} = -\mathbf{v} \cdot \mathbf{q},\ \mathbf{q} \in B\} \\
&= S_A(\mathbf{v}) \ominus S_B(-\mathbf{v})
\end{aligned} \tag{4}$$

Next we should consider how a support mapping is affected by a change of coordinate system. This is important because of the practice of defining each object in its own local coordinate systems and a mapping from it to the world coordinate system. So if we have an object to world translation T_A, an object to world rotation R_A and a support mapping S_A in local coordinates. Then the support mapping S_A^{world} can be defined as follows:

$$S_A^{world}(\mathbf{v}) = T_A(R_A(S_A(R_A^{-1}(\mathbf{v})))) \tag{5}$$

Since $\mathbf{v}$ is a vector it is unaffected by translations so we do not need to translate it into the local coordinate system, but it needs to be rotated into the local system, so we start by

computing $R_A^{-1}(\mathbf{v})$ which is used as input to S_A. The support mapping returns points so now we have to both rotate and translate them to the world system. Applying the object to world rotation R_A followed by the object to world translation T_A gives us the final result.

One important difference between the mathematical definition of a support mapping (Equation 3) and the practical implementations that we will later use is: the mathematical definition returns a set of points, while the actual implementations return any one point from the set in the mathematical definition. Therefore from here on we will assume that $S_X(\mathbf{v})$ returns one point instead of a set of points. In other words we redefine S_X to be a function $S_X : \mathbb{R}^n \to \mathbb{R}^n$. This has no practical effect on the definition of a support map in world space, but the definition of $S_{A\ominus B}$ can be simplified to $S_{A\ominus B}(\mathbf{v}) = S_A(\mathbf{v}) - S_B(-\mathbf{v})$.

Given the definitions above and support mappings in local coordinates for our objects we can easily construct support mappings in world coordinates for all objects and all combinations of objects of the type $A \ominus B$. Support mappings in local space can be constructed so that they can be sampled in constant time for basic shapes such as ellipsoids, boxes, discs, lines, spheres etcetera. Support mappings for polyhedrons in $\mathbb{R}^3$ that can be evaluated in $\Theta(\log(n))$ time (where n is the size[1] of the polyhedron) can be constructed using a hierarchical representation of the polyhedron similar to the one presented by Dobkin and Kirkpatrick [5]. Support mappings for more complex objects can also be constructed from support mappings of simpler objects in much the same way that we construct the support mapping of $A \ominus B$, for instance $S_{A\oplus B}(\mathbf{v}) = S_A(\mathbf{v}) + S_B(\mathbf{v})$, and $S_{\max(A,B)}(\mathbf{v}) = \max(S_A(\mathbf{v}), S_B(\mathbf{v}))$ (where max is computed as the vector that gives the maximum scalar product with $\mathbf{v}$). $A \oplus B$ results in a shape that looks like A with B swept around its surface (e.g. if A is a Box and B a sphere we get a Box with rounded edges). And $\max(A, B)$ is a mapping for the convex hull of A and B (e.g. if A is a Point and B is a Disc we get a cone). Some example support mappings in local space can be found in Table 1.

[1] Dobkin and Kirkpatrick define the size of a polyhedron as the total number of vertexes, edges and faces it contains.

Description	Support Mapping
$S_{point}(\mathbf{v})$	$\mathbf{p}$
$S_{segment}(\mathbf{v})$	$\begin{pmatrix} r_x\mathtt{sgn}(v_x) & 0 & 0 \end{pmatrix}$
$S_{rectangle}(\mathbf{v})$	$\begin{pmatrix} r_x\mathtt{sgn}(v_x) & r_y\mathtt{sgn}(v_y) & 0 \end{pmatrix}$
$S_{box}(\mathbf{v})$	$\begin{pmatrix} r_x\mathtt{sgn}(v_x) & r_y\mathtt{sgn}(v_y) & r_z\mathtt{sgn}(v_z) \end{pmatrix}$
$S_{disc}(\mathbf{v})$	$r\frac{\begin{pmatrix} v_x & v_y & 0 \end{pmatrix}}{\left\|\begin{pmatrix} v_x & v_y & 0 \end{pmatrix}\right\|}$
$S_{sphere}(\mathbf{v})$	$r\mathbf{v}$
$S_{ellipse}(\mathbf{v})$	$\frac{\begin{pmatrix} r_x^2 v_x & r_y^2 v_y & 0 \end{pmatrix}}{\left\|\begin{pmatrix} r_x v_x & r_y v_y & 0 \end{pmatrix}\right\|}$
$S_{ellipsoid}(\mathbf{v})$	$\frac{\begin{pmatrix} r_x^2 v_x & r_y^2 v_y & r_z^2 v_z \end{pmatrix}}{\left\|\begin{pmatrix} r_x v_x & r_y v_y & r_z v_z \end{pmatrix}\right\|}$
$S_{capsule}(\mathbf{v})$	$S_{segment}(\mathbf{v}) + S_{sphere}(\mathbf{v})$
$S_{lozenge}(\mathbf{v})$	$S_{rectangle}(\mathbf{v}) + S_{sphere}(\mathbf{v})$
$S_{roundbox}(\mathbf{v})$	$S_{box}(\mathbf{v}) + S_{sphere}(\mathbf{v})$
$S_{cylinder}(\mathbf{v})$	$S_{segment}(\mathbf{v}) + S_{disc}(\mathbf{v})$
$S_{cone}(\mathbf{v})$	$\mathtt{maxsupport}(S_{disc}(\mathbf{v}),\ \begin{pmatrix} 0 & 0 & height \end{pmatrix})$
$S_{wheel}(\mathbf{v})$	$S_{disc}(\mathbf{v}) + S_{sphere}(\mathbf{v})$
$S_{frustum}(\mathbf{v})$	$\mathtt{maxsupport}(S_{rectangle1}(\mathbf{v}),\ S_{rectangle2}(\mathbf{v}) + \begin{pmatrix} 0 & 0 & height \end{pmatrix})$

Table 1: Example Support Mappings

Example support mappings in local space. v_i is the i component of the unit length sample vector $\mathbf{v}$, r_i is the "radius" of the shape in dimension i (radius is used informally to mean the extent in a given axis), `sgn` is the sign function, and r is the radius of a quadratic shape. Finally `maxsupport` is a function such that given two vectors $\mathbf{x}, \mathbf{y}$ it returns $\mathbf{x}$ if $\mathbf{v} \cdot \mathbf{x} > \mathbf{v} \cdot \mathbf{y}$ otherwise it returns $\mathbf{y}$.

2.3 Gilbert-Johnson-Keerthi's Distance Algorithm

The first algorithm we present is GJK, an iterative algorithm that computes the distance between two objects A and B. Like all the algorithms used in this thesis it works through a support mapping $S_{A\ominus B}$. What GJK does is to measure the shortest directed distance from B to A, which it does by creating and iteratively updating a simplex inside $A \ominus B$. In each iteration GJK checks if the simplex contains the origin, if so $A \ominus B$ must also contain the origin, and GJK halts with a distance of $\mathbf{0}$. If the origin is not in the simplex it finds a new simplex that is at least as close to the origin as the last and still is inside $A \ominus B$. Finally it checks if the new simplex is the same distance from the origin as the last one. If it is, the simplex has reached the boundary of $A \ominus B$ and GJK halts and returns the shortest vector from the simplex to the origin.

All points returned by the support mapping will be on the boundary of $A \ominus B$. GJK finds the point in $A \ominus B$ that is closest to the origin. If the origin lies outside of $A \ominus B$ then the closest point will be on the boundary; otherwise the closest point will be the origin itself.

If we sample $S_{A\ominus B}$ between 1 and $d+1$ times in different directions (where d is the dimension of our space) we will get a simplex contained in $A \ominus B$.

Now assume we have a function $v(X)$ that computes the point $\mathbf{v}$ closest to the origin in the convex hull of a simplex X. We can then create a simplex $Y \subseteq X$, such that Y is the minimum set of vectors from X that are needed to write $\mathbf{v}$ as a convex combination. Any point on the boundary of the convex hull of X can be written as a convex combination using only the vectors of the feature that $\mathbf{v}$ is contained in (e.g. if $\mathbf{v}$ is on the face of a triangle then only the vertexes of the triangle are needed to express $\mathbf{v}$ as a convex combination). So it follows that $Y = X$ if and only if $\mathbf{v} = \mathbf{0}$ or $|X| < d + 1$. Therefore the only time we will not have room to add a new point $\mathbf{p}$ to Y in such a way that $Y \cup \{\mathbf{p}\}$ is still a simplex, is if $\mathbf{v} = \mathbf{0}$ which means that the origin was already contained in X.

Now if we compute $\mathbf{w} = S_{A\ominus B}(-\mathbf{v})$ (i.e. sample the support mapping in the direction of the origin), then $\|v(Y \cup \{\mathbf{w}\})\| \leq \|v(X)\|$ and $\|v(Y \cup \{\mathbf{w}\})\| = \|v(X)\|$ if and only if X is on the boundary of $A \ominus B$ or $\mathbf{0} \in conv(X)$ where $conv(X)$ is the set of points in the convex hull of X.

If we iteratively compute the closest point on our current simplex, throw away any points not needed to express the closest point as a convex combination and compute a new simplex by sampling in the direction of the origin from the closest point, then each simplex will be closer to the origin unless it is impossible to get closer and still be in $A \ominus B$.

The first termination condition for the iteration is: if the origin is inside the simplex. If this occurs we know that A and B have a nonempty intersection and the distance between them is $\mathbf{0}$. The second termination condition is: if closest point in iteration $i+1$ is the same distance from the origin as the closest point in iteration i (within a numerical tolerance) which tells us that the closest distance between A and B is the length of the last $\mathbf{v}$ computed.

Pseudo code for GJK can be found in Algorithm 1.

Algorithm 1 Gilbert-Johnson-Keerthi's Distance Algorithm

```
1:  function GJK(A, B, eps)                          ▷ Shortest distance from B to A
2:      d ← A.position − B.position
3:      v_0 ← vector(infinity)                       ▷ Set v_0 to be an infinitely long vector
4:      v ← A.support(d) − B.support(−d)
5:      W ← {v}
6:      while ||v_0|| − ||v|| > eps and ||v|| > eps do   ▷ Still getting closer
7:          w ← A.support(−v) − B.support(v)         ▷ Get new support point in direction of origin
8:          v_0 ← v
9:          (v, W) ← closest(W ∪ {w})                ▷ Find closest point in simplex and update simplex
10:     end while
11:     return v                                     ▷ v is the shortest distance from B to A
12: end function
```

The above iterative algorithm transforms the problem of looking for the origin in an implicitly defined convex shape to the problem of finding the point in the convex hull of a simplex that is closest to the origin. There are two general ways that the function $(v, Y) = \text{closest}(W)$ can be computed, either algebraically, or geometrically. The original GJK publication[1] uses an algebraic method called Johnson's distance algorithm which is described in section 2.3.1. Another way to solve the problem in 3D is a geometric approach that checks which Voronoi region of the simplex the origin is in followed by computing the closest point for that region and setting Y as the feature associated with said region. That algorithm is explained in section 2.3.2.

2.3.1 Johnson's Distance Algorithm

The original GJK algorithm finds the point closest to the origin in the convex hull of a simplex by expressing the problem as a series of linear equations to be solved. The reasoning behind the algorithm works as follows:

We know that the point $\mathbf{p}$ closest to the origin can be written as a convex combination of the points in the simplex X, and $\mathbf{p}$ must be perpendicular to the feature $Y \subseteq X$ that it belongs to. Unfortunately the nonnegativity constraints on λ cannot be solved by a linear system as it is a nonlinear constraint.

So instead we look at the affine hull of the simplex. Any affine hull has a unique point that is part of the hull and perpendicular to the hull. Any point $\mathbf{p}$ on the affine hull of a simplex can be written as a affine combination of the vertexes of the simplex X. The combination of the two previous statements can be written as a linear system in the following way:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ (\mathbf{x}_2 - \mathbf{x}_1) \cdot \mathbf{x}_1 & (\mathbf{x}_2 - \mathbf{x}_1) \cdot \mathbf{x}_2 & \cdots & (\mathbf{x}_2 - \mathbf{x}_1) \cdot \mathbf{x}_{n-1} & (\mathbf{x}_2 - \mathbf{x}_1) \cdot \mathbf{x}_n \\ (\mathbf{x}_3 - \mathbf{x}_1) \cdot \mathbf{x}_1 & (\mathbf{x}_3 - \mathbf{x}_1) \cdot \mathbf{x}_2 & \cdots & (\mathbf{x}_3 - \mathbf{x}_1) \cdot \mathbf{x}_{n-1} & (\mathbf{x}_3 - \mathbf{x}_1) \cdot \mathbf{x}_n \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ (\mathbf{x}_n - \mathbf{x}_1) \cdot \mathbf{x}_1 & (\mathbf{x}_n - \mathbf{x}_1) \cdot \mathbf{x}_2 & \cdots & (\mathbf{x}_n - \mathbf{x}_1) \cdot \mathbf{x}_{n-1} & (\mathbf{x}_n - \mathbf{x}_1) \cdot \mathbf{x}_n \end{pmatrix} \lambda = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{6}$$

For a simplex of n vertexes the matrix above will be of size $n \times n$. So if we solve a system like the one above for all simplexes $Y \subseteq X$ and pick the largest one for which $\lambda > 0$ for all

elements, the closest point $\mathbf{p}$ can be computed as the linear combination of the the vertexes of that simplex:

$$\sum_{i=1}^{n} \lambda_i \mathbf{y}_i = \mathbf{p},\ \mathbf{y}_i \in Y$$

Lots of the factors in the above systems figure in more then one place or more then one system. Johnson's distance algorithm exploits this by solving the system in the following way. First we use Cramer's rule to rewrite the equation as:

$$\lambda = \frac{1}{\det(\mathbf{M})} \mathbf{C}^T \mathbf{r} \tag{7}$$

Where $\mathbf{M}$ is the matrix from equation 6 above, $\mathbf{C}$ is the cofactor matrix of $\mathbf{M}$, and $\mathbf{r}$ is the vector $\begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix}^T$. As the only nonzero element of $\mathbf{r}$ is the first one, λ is the first row of $\mathbf{C}$ divided by the determinant of $\mathbf{M}$, but since the first row of $\mathbf{M}$ is all ones, the determinant of $\mathbf{M}$ is equal to the sum of the first row of $\mathbf{C}$. Which gives us the following:

$$\lambda = \frac{1}{\mathbf{C}_{1,1} + \mathbf{C}_{1,2} + \ldots + \mathbf{C}_{1,n}} \begin{pmatrix} \mathbf{C}_{1,1} \\ \mathbf{C}_{1,2} \\ \vdots \\ \mathbf{C}_{1,n} \end{pmatrix} \tag{8}$$

As $\mathbf{C}_{i,j} = (-1)^{i+j} * \det(\mathbf{M}_{i,j})$, where $\mathbf{M}_{i,j}$ is the sub matrix of $\mathbf{M}$ gained by removing the ith row and jth column, we can construct a recursive definition of $\mathbf{C}^X_{1,j}$ (the X means that the cofactor element is associated with simplex X) by expansion along the bottom row of $\mathbf{M}$. Giving us the following:

$$\mathbf{C}^X_{1,j} = \begin{cases} 1 & |X| = 1 \\ \sum_{\mathbf{x}_i \in Y} \mathbf{C}^Y_{1,i} * ((\mathbf{x}_k - \mathbf{x}_j) \cdot \mathbf{x}_i) & X = Y \cup \{\mathbf{x}_j\} \end{cases} \tag{9}$$

Where $\mathbf{x}_k$ is an arbitrary vector in X, it does not matter which one is used as long as the same one is chosen for the computation of all $\mathbf{C}^X_{1,j}$. Implementations usually pick the first element of X. By using the recursive formula above we can reuse the computations from all subsets and thereby save a large number of computations. In addition if we assign each vector in our simplex an identifier i which is constant for all GJK iterations that the vector stays in the simplex, we can reuse computations between iterations as well. We might also cache the difference vectors $\mathbf{x}_i - \mathbf{x}_j$ in a $n \times n$ matrix as they are reused in several of the computations.

The main disadvantage of Johnson's distance algorithm is that it relies on matrix inversion by cofactor expansion, which is even more numerically unstable than matrix inversion usually is. But as the systems inverted are small the instability can be manageable, at least if implemented with double precision floating point numbers. Another possible downside is

that the number of systems we need to solve is 2^{d+1} where d is the dimension of the space we are working in, but this is usually not an issue as intersection testing in dimensions above three is practically unheard of.

2.3.2 Voronoi Region Based Distance Algorithm

An alternative to Johnson's algorithm when working in 3D is to first figure out which feature (point, line, triangle, or tetrahedron) of the simplex is closest to the origin and then compute the closest point on that feature and update the simplex to that feature.

The point closest to the origin in a simplex that is a point, is trivially the point itself.

The point closest to the origin in a line segment can according to [6], be computed in the following way. Given the two endpoints of the line segment $\mathbf{a}$ and $\mathbf{b}$, let $\mathbf{c} = \mathbf{b} - \mathbf{a}$. The point $\mathbf{p}$ closest to the origin on the infinite line can be expressed as $\mathbf{p} = \mathbf{a} + t * \mathbf{c}$, and t can be computed as $t = -\frac{\mathbf{a} \cdot \mathbf{c}}{\|\mathbf{c}\|^2}$. If $t < 0$ then the closest point on the line lies before the start of the line segment, therefore the closest point on the line segment is $\mathbf{p} = \mathbf{a}$. Alternatively if $t > 1$ then the closest point on the line lies after $\mathbf{b}$ so the closest point on the line segment is $\mathbf{p} = \mathbf{b}$. Finally if $0 \leq t \leq 1$ then the closest point on the line is the same as for the line segment so the closest point is $\mathbf{p} = \mathbf{a} + t * \mathbf{c}$. Pseudo code for a version of the above is presented in Algorithm 2. It takes a line segment in the form of the two endpoints as argument and returns a tuple ($\mathbf{p}$, keep) where keep is a tuple containing either $\mathbf{a}$, $\mathbf{b}$ or both depending on which are needed to express $\mathbf{p}$ as a linear combination.

Algorithm 2 Point Closest to the Origin in a Line Segment

```
1:  function CLOSEST(a, b)                    ▷ Find the point closest to the origin on a line segment
2:      c ← b − a                             ▷ let c be the vector from a to b
3:      t ← a · c/‖c‖²                        ▷ compute the parameter t for the closest point on line a + t(b − a)
4:      if t < 0 then                         ▷ If t is less than 0 the closest point on the line is before a
5:          return (a, (a))                   ▷ The closest point on the line segment must be a
6:      else if t > 1 then                    ▷ If t is greater than 1 the closest point on the line is after b
7:          return (b, (b))                   ▷ The closest point on the line segment must be b
8:      else                                  ▷ Closest point for the line is in the line segment
9:          return (a + t ∗ c, (a, b))
10:     end if
11: end function
```

The point closest to the origin in a triangle is either the point closest to the origin on the plane that contains the triangle, or the closest point lies on one of the edges of the triangle. Therefore we can find the closest point on a triangle by checking if the origin is in the positive half space of a plane that is orthogonal to both the triangle normal and each edge. Given a triangle normal $\mathbf{n}$ an edge $\mathbf{e}$ and a point $\mathbf{p}$ on the edge, the half space test can be formulated as $(\mathbf{n} \times \mathbf{e}) \cdot \mathbf{p} < 0$. Note that we must keep track of the orientation of $\mathbf{n}$ and $\mathbf{e}$ for this to work. If we find only one such a plane then the closest point lies on the edge corresponding to that plane. If no such plane exists then the origin must lie above or below the triangle and the closest point will be an internal point of the triangle. If we find two such planes we must figure out which of the two edges corresponding to the planes is closer to the origin. The two planes must share a common vertex, let us call it $\mathbf{x}$. We check the

line from the origin against the edge that points towards the common vertex (according to the winding we chose). If these two vectors point in the same direction (i.e. $\mathbf{x} \cdot \mathbf{e} > 0$) the edge that points towards the common vertex is closest; otherwise the other edge is.

Pseudo code for computing the point closest to the origin of a triangle can be found in Algorithm 3: it assumes that the triangle is nondegenerate, it uses Algorithm 2 as a subroutine and returns the same kind of tuple except that in this case the second element of the tuple can contain up to three points.

Algorithm 3 Point Closest to the Origin in a Triangle

```
1:  function CLOSEST(a, b, c)                    ▷ Compute point closest to origin in a triangle
2:      ba ← a − b                                ▷ Compute edge from b to a
3:      ac ← c − a                                ▷ Compute edge from a to c
4:      cb ← b − c                                ▷ Compute edge from c to b
5:      n ← (b − a) × ac                          ▷ Compute normal of triangle abc
6:      front ← 0
7:      if (n × ba) · a < 0 then
8:          front ← front + 1
9:      end if
10:     if (n × ac) · a < 0 then
11:         front ← front + 2
12:     end if
13:     if (n × cb) · b < 0 then
14:         front ← front + 4
15:     end if
16:     if front = 0 then        ▷ Origin must be above or below abc so the closest point is internal
17:         n ← n/‖n‖
18:         return (n ∗ n · a, (a, b, c))
19:     else if front = 1 then                    ▷ Origin is only in front of n × ba
20:         return CLOSEST(a, b)
21:     else if front = 2 then                    ▷ Origin is only in front of n × ac
22:         return CLOSEST(a, c)
23:     else if front = 3 then                    ▷ Origin is in front of both n × ba and n × ac
24:         if a · ba > 0 then                    ▷ Origin is closer to the edge ba than to ac
25:             return CLOSEST(a, b)
26:         else                                  ▷ Origin is closer to the edge ac than to ba
27:             return CLOSEST(a, c)
28:         end if
29:     else if front = 4 then                    ▷ Origin is only in front of n × cb
30:         return CLOSEST(b, c)
31:     else if front = 5 then                    ▷ Origin is in front of both n × ba and n × cb
32:         if b · cb > 0 then                    ▷ Origin is closer to the edge cb than to ba
33:             return CLOSEST(b, c)
34:         else                                  ▷ Origin is closer to the edge ba than to cb
35:             return CLOSEST(a, b)
36:         end if
37:     else                                      ▷ Origin is in front of both n × cb and n × ac
38:         if c · ac > 0 then                    ▷ Origin is closer to the edge ac than to cb
39:             return CLOSEST(a, c)
40:         else                                  ▷ Origin is closer to the edge cb than to ac
41:             return CLOSEST(b, c)
42:         end if
43:     end if
44: end function
```

The point closest to the origin in a tetrahedron is either $\mathbf{0}$, if the origin is inside the tetrahedron, or it is equal to the point closest to one of the features that construct the tetrahedron. So our strategy for finding the closest point for a tetrahedron is to check which of the positive half spaces for the triangle faces the origin is inside, and then use that information

to compute the closest point. One problem with this approach is that given four arbitrary points we do not know what winding the faces have. To solve this we compute an internal point of the tetrahedron $\mathbf{center} = (\mathbf{a} + \mathbf{b} + \mathbf{c} + \mathbf{d}) * \frac{1}{4}$, where $\mathbf{a}, \mathbf{b}, \mathbf{c}$ and $\mathbf{d}$ are the vertexes of the tetrahedron. Given a triangle normal $\mathbf{n}$ and a point $\mathbf{x}$ in the triangle, we can test if the origin and the tetrahedron are in the same half space by checking if the following equality holds.

$$(\mathbf{n} \cdot (\mathbf{x} - \mathbf{center}) > 0) \equiv (\mathbf{n} \cdot \mathbf{x} < 0)$$

If we find that the test above holds for none of the triangle normals then the origin must be inside the tetrahedron. If the test only holds for one triangle then the closest point must be somewhere on that triangle. If we find that it holds for two or three triangles then we need additional information to determine which triangle is closest to the origin.

In the case where the origin is above two of the faces, we know that the closest point of the tetrahedron is equal to the closest point of one or both of the triangles corresponding to the faces the origin is in front of. If the closest point is on the edge between the two triangles it does not matter which one we send to the triangle subroutine so our test only has to work for cases where the closest point is inside one of the triangles but not the other. We can test if the origin is above one of the triangles by computing a new split plane that is orthogonal to both the shared edge and one of the triangle normals. We then check if the origin is above that triangle by testing if the third point in the triangle we used to construct the new split plane is on the same side of the plane as the origin. If the third triangle point and the origin are on the same side we know that the closest point is in that triangle so we call the closest point computation routine for that triangle, otherwise we call the same routine using the other triangle. Note that this only works because we already know that the origin is in front of both triangle faces.

As an example of this procedure assume that we find that the origin is above the triangles **abc** and **abd** the two common vertexes are **a** and **b**, so we compute a vector $\mathbf{n}$ that is orthogonal to both the edge **ab** and the normal of triangle **abc**, $\mathbf{n} = ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})) \times (\mathbf{b} - \mathbf{a})$. We then test $((\mathbf{a} - \mathbf{c}) \cdot \mathbf{n} < 0) \equiv (\mathbf{a} \cdot \mathbf{n} < 0)$, if this relation holds we know that **c** and the origin must be on the same side of the plane with normal $\frac{\mathbf{n}}{\|\mathbf{n}\|}$ that contains the point **a**. Which means that the distance from the origin to triangle **abc** is less or equal to the distance from the origin to triangle **abd** therefore the closest point must be part of abc. If the test does not hold we know that **abc** is not closer to the origin than **abd**, they might still be the same distance from the origin but in that case the closest point is on the edge **ab** so we can safely say that the closest point is part of **abd**.

The principle of the solution method is the same for the cases where we find that the origin is in front of three of the tetrahedrons faces. To find which of the three possible triangles is closest to the origin we use an elimination strategy. First we pick one of the edges that contains the common vertex, and a triangle that borders that edge. We use the normal of the chosen triangle together with the edge and compute a vector that is orthogonal to both.

If the third point and the origin are on the same side of the plane through the common point with a normal parallel to the computed vector then either the chosen triangle is closest to the origin or the triangle that does not border the chosen edge is closest so we repeat the procedure with one of those triangles and the edge between them. If we find that the third point in the triangle and the origin are on different sides of the plane we know that the chosen triangle is not closest to the origin so we repeat the procedure with the edge between the two remaining triangles and one of the triangles that borders it.

For example if we find that the origin is in front of the three faces **abc**, **abd** and **acd**, the common vertex is **a**. Assume the first edge triangle combination we test is **ad** together with the triangle **abd** we compute the following vector: $\mathbf{n_{ad}} = ((\mathbf{b} - \mathbf{a}) \times (\mathbf{d} - \mathbf{a})) \times (\mathbf{d} - \mathbf{a})$. We use this vector to test against the origin and the vertex **b**, by testing if $(\mathbf{n_{ad}} \cdot (\mathbf{b} - \mathbf{a}) > 0) \equiv (\mathbf{n_{ad}} \cdot \mathbf{a} < 0)$ then the closest triangle is **abd** or **abc** as **b** must be part of the closest triangle, otherwise we know that **abd** cannot be closest so either **abc** or **acd** is closest. If we now know that **abd** or **abc** is closest to the origin then we next examine the edge **ab** together with the triangle **abc** giving us the vector $\mathbf{n_{ab}} = ((\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})) \times (\mathbf{b} - \mathbf{a})$. We use $\mathbf{n_{ab}}$ to test the origin and the vertex **c**, by checking if $(\mathbf{n_{ab}} \cdot (\mathbf{c} - \mathbf{a}) > 0) \equiv (\mathbf{n_{ab}} \cdot \mathbf{a} < 0)$ then **abc** is closest as **c** must be part of the closest triangle else **abd** must be the closest triangle. Assume that we after the first test instead knew that the closest triangle is either **abc** or **acd** we instead use the triangle **acd** together with the edge **ac** to compute $\mathbf{n_{ac}} = ((\mathbf{c} - \mathbf{a}) \times (\mathbf{d} - \mathbf{a})) \times (\mathbf{c} - \mathbf{a})$. Followed by testing the vertex **d** and the origin against this vector by checking if $(\mathbf{n_{ac}} \cdot (\mathbf{d} - \mathbf{a}) > 0) \equiv (\mathbf{n_{ac}} \cdot \mathbf{a} < 0)$ then **acd** must be closest otherwise **abc** is closest.

We do not present pseudo code for the tetrahedron procedure as it would become too long and cumbersome, but MPR uses a similar split-plane strategy to the one used in the three triangle case which can be seen in Algorithm 8, and is illustrated in Figure 1.

Using the algorithms explained above, the point closest to the origin for any simplex in $\mathbb{R}^3$ can be computed by simply calling the version of `CLOSEST` that corresponds to the number of points in the simplex.

The main disadvantage of this geometric method for computing the closest point is that it is probably slower than the algebraic version, although that will depend on the implementation. One optimization that is not presented above is that if we know which vertex in our simplex we found last, we can eliminate some of the tests as we know that the closest point must be on a feature that the last point added is part of. For example, in the tetrahedron version we would only have to check three of the four triangles etcetera. Another optimization that is not present in the pseudo code for Algorithm 3 (and the tetrahedron case description) is to use a switch statement over a bit-field instead of the if statements, this will make it easier for the compiler to realize that the algorithm can be implemented using a branch table. The main advantage of this method is that its geometric nature makes it easier to understand, modify and tweak.

2.3.3 Using GJK for Intersection Testing

GJK can be used for intersection testing out of the box, but if we are not interested in the distance between the two objects a significant speedup can be gained by adding an early out as soon as we detect that the origin is on the other side of the latest support plane. The support plane for a support point $\mathbf{w} = S_{A \ominus B}(-\mathbf{v})$ is a plane with a normal parallel to $-\mathbf{v}$ that contains the point $\mathbf{w}$. Therefore we could add a check that halts the method and returns false if $\mathbf{v} \cdot \mathbf{w} > 0$, as we know that the origin is not in $A \ominus B$. This gives us a new algorithm that is called Separating-Axis-GJK (SA-GJK). Pseudo code for SA-GJK can be found in Algorithm 4.

Algorithm 4 Separating-Axis-GJK

```
1:  function SA-GJK(A,B,eps)                          ▷ Do A and B intersect
2:      d ← A.position − B.position
3:      v ← A.support(d) − B.support(−d)
4:      W ← {v}
5:      while ‖v‖ > eps do                            ▷ Origin not in simplex
6:          w ← A.support(−v) − B.support(v)
7:          if v · w > 0 then                         ▷ Origin outside support plane
8:              return false                          ▷ No A and B do not intersect
9:          end if
10:         (v, W) ← closest(W ∪ {w})                 ▷ Find closest point and update simplex
11:     end while
12:     return true                                   ▷ Yes A and B intersect
13: end function
```

2.3.4 Using GJK for Penetration Depth Computation

GJK is incapable of computing the penetration depth in its default configuration, but there is a way to modify it so that it can give an estimate of the penetration depth. Consider computing the Minkowski sum $X \oplus Y$ with X as each of our objects A , B and Y as a tiny sphere. This would give us two shapes that are very similar to A and B, just slightly larger and with rounded corners. Now if we run GJK on our original shapes A and B, giving us the distance between them, and treat any distance less then two times the radius of Y as if it was an intersection with a penetration depth of $(2r - d) * \mathbf{v}$ where r is the radius of Y, d is the length of the vector returned by GJK, and $\mathbf{v}$ is the normalization of the vector returned by GJK. What we are in fact doing is computing the intersection of $A \oplus Y$ vs. $B \oplus Y$, except we can now get a penetration depth for intersections that are shallower then $2r$. This algorithm can be combined with the early out from SA-GJK if we are not interested in the distance between nonintersecting objects. Pseudo code for how this modification is done can be seen in Algorithm 5. Unfortunately this modification can still not handle intersections deeper than $2r$ but it can detect them, so if we had an algorithm that can compute the penetration depth but that is too slow to run as a default algorithm we could pass deep intersections to that backup procedure and handle shallow intersections ourselves. The Expanding Polytope Algorithm which is presented in section 2.5, happens to be just that kind of generally applicable penetration depth computation.

Algorithm 5 Modified SA-GJK that Computes Penetration Depth

```
1: function SWEEP-SA-GJK(A, B, r, eps)
2:     d ← A.POSITION − B.POSITION
3:     v_0 ← VECTOR(infinity)
4:     v ← A.SUPPORT(d) − B.SUPPORT(−d)
5:     W ← {v}
6:     repeat
7:         w ← A.SUPPORT(−v) − B.SUPPORT(v)
8:         if v · w > 2r ∗ ||v|| then          ▷ Distance from origin to support plane is greater then 2r
9:             return (false, 0)
10:        end if
11:        v_0 ← v
12:        (v, W) ← CLOSEST(W ∪ {w})
13:    until ||v_0|| − ||v|| < eps or ||v|| < eps
14:    if ||v|| < eps then          ▷ We have a deep intersection that we cannot handle directly
15:        v ← EPA(A, B, W)                                          ▷ Call backup routine such as EPA
16:        return (true, v)                                ▷ Return depth as computed by the backup procedure
17:    else                                                ▷ A and B intersect only in the boundaries
18:        return (true, (2r − ||v||) v/||v||)
19:    end if
20: end function
```

2.4 Minkowski Portal Refinement

Minkowski Portal Refinement (MPR) first published in [3], is a different way to work with $A \ominus B$. The difference in approach compared to GJK is that MPR looks at the problem from a purely geometrical point of view. Since MPR uses a geometric approach the default algorithm needs to be reformulated to work in a different space than $\mathbb{R}^3$. We will only examine the $\mathbb{R}^3$ version here. The default query that MPR answers is: do two objects A and B intersect? MPR has the same limitations on A and B as GJK does, namely that A and B are compact convex sets that each have a support mapping.

MPR consists of three steps or phases which are listed below.

1. Construct a tetrahedron such that it has one point that is in the interior and three points that are on the boundary of $A \ominus B$, we call the triangle consisting of the three points on the boundary our portal.

2. Replace one of the points in the portal with a new one until a ray from the internal point in the direction of the origin passes through the portal, we call this ray our origin ray.

3. Generate a new portal such that the origin ray still passes through it and it is closer to the surface of $A \ominus B$ than our old portal. We repeat this step until we either find that it is impossible to reach the origin or we find the origin is inside our current tetrahedron.

We will now go through each of the steps above covering why we need that step and presenting pseudo code for how it can be done.

A tetrahedron consisting of one internal point and three points on the boundary is the basic invariant of the algorithm. It is needed for the next two steps, the internal point is used as

a source for our ray towards the origin, and gives us the direction we need to look in. The portal constructed in this step does not give us any information as the origin ray does not pass through it. An internal point $\mathbf{v}$ of $A \ominus B$ can be constructed from any two internal points from A and B by computing $\mathbf{v} = A_{internal} - B_{internal}$. Given $\mathbf{v}$ we can compute the direction of our origin ray $\mathbf{r} = -\mathbf{v}$. Since we know that we will later want the origin ray to intersect our portal it makes sense to construct it roughly in the direction of the ray so we construct our first portal point as $\mathbf{a} = S_{A\ominus B}(\mathbf{r})$. As long as the origin is not in the line from $\mathbf{v}$ with direction $\mathbf{a} - \mathbf{v}$ they will not be parallel so we find our second portal point by computing $\mathbf{b} = S_{A\ominus B}(\mathbf{v} \times \mathbf{a})$. If $\mathbf{v} \times \mathbf{a} \equiv \mathbf{0}$ we need to test if the origin is in the negative half space of the plane that contains $\mathbf{a}$ and has normal $-\mathbf{v}$. If the origin is in the negative half space then the origin is on the line segment between $\mathbf{v}$ and $\mathbf{a}$ but since the line segment is contained in $A \ominus B$ the origin must be in $A \ominus B$ so we halt and return an intersection. If the origin is in the positive half space of the support plane it cannot be in $A \ominus B$ so we halt and return a nonintersection. Our fourth point can now be computed in the direction of the plane normal that contains the triangle $\mathbf{vab}$, so we compute $\mathbf{c} = S_{A\ominus B}((\mathbf{a} - \mathbf{v}) \times (\mathbf{b} - \mathbf{v}))$, unless $(\mathbf{a} - \mathbf{v}) \times (\mathbf{b} - \mathbf{v}) \equiv \mathbf{0}$ in which case our triangle has degenerated into a line segment. Since we know that the origin is not on the line segment $\mathbf{va}$ it cannot be on the line segment $\mathbf{vb}$ as $\mathbf{b}$ must be further from $\mathbf{0}$ than $\mathbf{a}$ because $\mathbf{a}$ is as far as we can go in the direction towards the origin from $\mathbf{v}$ and still be in $A \ominus B$. Therefore we can safely return a nonintersection if $(\mathbf{a} - \mathbf{v}) \times (\mathbf{b} - \mathbf{v}) \equiv \mathbf{0}$. We now either have the tetrahedron we wanted to construct or have already determined if A intersects B. Pseudo code for the above procedure can be seen in Algorithm 6, which takes the two shapes A and B as input and returns a tuple (i, t) where $i \equiv -1$ means that A and B do not intersect, $i \equiv 1$ means that they do intersect and $i \equiv 0$ means that we do not know yet. In the case where i is 0 the return value t will be a tuple of four points where the first is internal and the other three are on the boundary of $A \ominus B$. If i is not 0 then t will be an empty tuple.

Algorithm 6 Minkowski Portal Refinement Phase One

```
 1: function MPR-Phase-One(A, B)
 2:     v ← A.position − B.position
 3:     a ← A.support(−v) − B.support(v)
 4:     if v × a ≡ 0 then
 5:         if v · a < 0 then
 6:             return (1, ())
 7:         else
 8:             return (−1, ())
 9:         end if
10:     end if
11:     b ← A.support(v × a) − B.support(−(v × a))
12:     if (a − v) × (b − v) ≡ 0 then
13:         return (−1, ())
14:     end if
15:     c ← A.support((a − v) × (b − v)) − B.support(−((a − v) × (b − v)))
16:     return (0, (v, a, b, c))
17: end function
```

In phase two we must make sure that the origin ray passes through our portal. The reason for this condition is that in phase three we will iteratively search in the direction of the portal's normal and if the origin ray does not pass through the portal we can not be sure that the

portal's normal points in the right direction. If the origin ray passes through the portal the normal of all the other triangles that make up our tetrahedron will point away from the origin ray. For the triangle **vab** for instance we get $\mathbf{n} = (\mathbf{a} - \mathbf{v}) \times (\mathbf{b} - \mathbf{v})$ followed by the test $\mathbf{r} \cdot \mathbf{n} \leq 0$. If we find that such a test fails for one of the triangles then we swap the two portal points in that triangle to invert that triangle's normal and replace the point that is not in the triangle that failed by a new point $\mathbf{x} = S_{A \ominus B}(\mathbf{n})$. If for example the test above failed such that $\mathbf{r} \cdot \mathbf{n} > \mathbf{0}$ for the triangle **vab**, we would set $\mathbf{y} \leftarrow \mathbf{b},\ \mathbf{b} \leftarrow \mathbf{a},\ \mathbf{a} \leftarrow \mathbf{y},\ \mathbf{c} \leftarrow S_{A \ominus B}(\mathbf{n})$ and repeat the tests for all the new triangles. Pseudo code for this can be found in Algorithm 7, which takes the two shapes A and B together with four points $\mathbf{v}, \mathbf{a}, \mathbf{b}, \mathbf{c}$ as input where $\mathbf{v}$ is the internal point and **abc** is the current portal. It returns a tuple $(\mathbf{v}, \mathbf{a}, \mathbf{b}, \mathbf{c})$ which is the internal point together with a portal that the origin ray passes through.

Algorithm 7 Minkowski Portal Refinement Phase Two

```
 1: function MPR-Phase-Two(A, B, v, a, b, c)
 2:     r ← −v                                   ▷ Let r be the direction of the origin ray
 3:     repeat
 4:         done ← true                          ▷ Assume that the current portal is correct
 5:         n_vab ← (a − v) × (b − v)             ▷ Compute normals of the three triangles
 6:         n_vbc ← (b − v) × (c − v)
 7:         n_vca ← (c − v) × (a − v)
 8:         if r · n_vab > 0 then                ▷ Test the triangle vab
 9:             done ← false                     ▷ The current portal is incorrect, set done to false
10:             c ← A.support(n_vab) − B.support(−n_vab)   ▷ Generate new c
11:             a, b ← b, a                      ▷ Swap a and b
12:         else if r · n_vbc > 0 then           ▷ Test the triangle vbc
13:             done ← false                     ▷ The current portal is incorrect, set done to false
14:             a ← A.support(n_vbc) − B.support(−n_vbc)   ▷ Generate new a
15:             b, c ← c, b                      ▷ Swap b and c
16:         else if r · n_vca > 0 then           ▷ Test the triangle vca
17:             done ← false                     ▷ The current portal is incorrect, set done to false
18:             b ← A.support(n_vca) − B.support(−n_vca)   ▷ Generate new b
19:             a, c ← c, a                      ▷ Swap a and c
20:         end if
21:     until done                               ▷ Repeat until the portal passes all three tests
22:     return (v, a, b, c)                      ▷ Return the final portal
23: end function
```

The third phase of MPR iteratively computes a new portal that is closer to the surface of $A \ominus B$ than the last one. This is repeated until we find that the origin is inside the current tetrahedron, that the origin is on the opposite side of the last support plane (similar to the test used in section 2.3.3), or that the origin is "close enough" to the current portal. The third case when the portal is "close enough" is needed for two reasons: If $A \ominus B$ has a quadratic surface in the direction we are searching and the origin is close to the surface of $A \ominus B$ we might need an unacceptable number of iterations to determine if A and B intersect. The second reason we need this test is if A and B are in touching contact, in which case we want the case for numerical stability so we are consistent in what we return for touching contacts (usually false since false negatives are better than false positives for physical simulation). To generate a new portal that is closer to the surface of $A \ominus B$ than the old one we first generate a point $\mathbf{p} = S_{A \ominus B}(\mathbf{n})$ where $\mathbf{n}$ is the normal of the current portal. Then a triangle made up of $\mathbf{p}$ and two of the vertexes from the current portal will be closer to the surface of $A \ominus B$ than the current portal. The reason we can be sure of this

is that **n** points in the direction of the surface and **p** will be as far as we can go in $A \ominus B$ in this direction. Now since we know that the origin ray passes through the current portal **abc** it follows that it must also pass through one of the triangles **abp**, **pbc**, or **apc**, and this triangle will fulfill the requirements for the portal update. To check which of the three triangles the origin ray passes through, we test which half space the origin is in for the three triangles **pva**, **pvb** and **pvc**. If the origin is in the positive half space of **pva** then we know that the new portal must be **abp** or **pbc**. If the origin is in the positive half space of **pvb** we know that the new portal must be either **pbc** or **apc**. Finally if the origin is in the positive half space of **pvc** we know that the new portal must be either **apc** or **abp** (see figure 1). Which side of a half space the origin is in can be computed as a scalar triple product[7]. If $\mathbf{v} \cdot (\mathbf{p} \times \mathbf{a}) > 0$ then the origin is in the positive half space of **pva**, and if $\mathbf{v} \cdot (\mathbf{p} \times \mathbf{b}) > 0$ then the origin is in the positive half space of **pvb** etcetera.

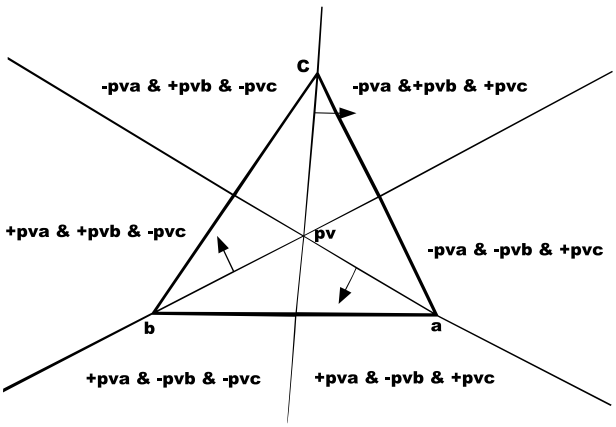

Figure 1: MPR Split Planes
The Portal seen from **p** in the direction $\mathbf{v} - \mathbf{p}$.
The three dotted lines are the half spaces that the origin is checked against.
The arrows indicate the positive side of each half space.

Combining all of this we get an algorithm for the third phase of MPR, which is presented as pseudo code in Algorithm 8: It takes two objects A and B together with four points $\mathbf{v}, \mathbf{a}, \mathbf{b}, \mathbf{c}$ such that **v** is an internal point of $A \ominus B$ and **abc** is a portal which a ray from **v** in the direction of the origin intersects. The algorithm returns a tuple (*bool*, *points*) where *bool* is true if and only if A and B intersect, and *points* is a tuple containing the final tetrahedron that was used in this phase.

Now that we have formulated all three phases we can combine them into a working whole. Pseudo code for the combination of the three phases can be found in Algorithm 9, which takes two shapes: A and B, and returns true if and only if A and B have a nonempty intersection.

The general principles of the MPR algorithm described above can be generalized to any space $\mathbb{R}^n$. It will always only need one internal point, but the number of points needed in the portal will vary with the dimensionality of the space, in such a way that the portal will consist of n points in $\mathbb{R}^n$. In addition a lot of the tests will have to be reformulated, as the cross product only exists in $\mathbb{R}^3$. But the general principles described in the text should be

Algorithm 8 Minkowski Portal Refinement Phase Three

```
1: function MPR-Phase-Three(A, B, v, a, b, c)
2:     while true do
3:         n ← (c − a) × (b − a)                ▷ Let n be the normal of the current portal
4:         if a · n > 0 then                    ▷ Check if the origin is inside the current tetrahedron
5:             return (true,(v, a, b, c))
6:         end if
7:         if a · n ≡ 0 then                    ▷ Check if the origin is in the current portal
8:             return (false,(v, a, b, c))
9:         end if
10:        p ← A.support(n) − B.support(−n)     ▷ Compute a new point p in the direction of the normal
11:        if p · n < 0 then                    ▷ Check if the origin is on the far side of the support plane
12:            return (false,(v, a, b, c))
13:        end if
14:        if v · (p × a) > 0 then              ▷ The origin is in pva:s positive half space, b must be kept
15:            if v · (p × b) < 0 then          ▷ The origin is in pvb:s negative half space, a must be kept
16:                c ← p
17:            else                             ▷ The origin is in pvb:s positive half space, c must be kept
18:                a ← p
19:            end if
20:        else                                 ▷ The origin is in pva:s negative half space, c must be kept
21:            if v · (p × c) > 0 then          ▷ The origin is in pvc:s positive half space, a must be kept
22:                b ← p
23:            else                             ▷ The origin is in pvc:s negative half space, b must be kept
24:                a ← p
25:            end if
26:        end if
27:    end while
28: end function
```

Algorithm 9 Complete Minkowski Portal Refinement

```
1: function MPR(A, B)
2:     i, v, a, b, c ← MPR-Phase-One(A, B)
3:     if i ≡ 1 then
4:         return true
5:     else if i ≡ −1 then
6:         return false
7:     end if
8:     v, a, b, c ← MPR-Phase-Two(A, B, v, a, b, c)
9:     ret, v, a, b, c ← MPR-Phase-Three(A, B, v, a, b, c)
10:    return ret
11: end function
```

applicable to any $\mathbb{R}^n$.

2.4.1 Using MPR for Penetration Depth Computation

MPR cannot compute the exact penetration depth, but it can be used as a good heuristic method for estimating the penetration depth for intersections that are not extremely deep. The estimation is done by adding two additional phases that should be computed once the basic MPR halts with a nonempty intersection. The first of these steps estimates the contact normal. This is done by continuing to iteratively advance the portal until we reach the boundary of $A \ominus B$. Once the portal reaches the boundary, the normal of the portal will be our heuristic estimate of the contact normal. The heuristic will fail if the internal point we initiated MPR with is closer to the surface of $A \ominus B$ than the origin. To minimize the risk of this happening we create the internal point from the geometric centroids [8] of each object

which should give us a internal point close to the center of $A \ominus B$. For good performance we should therefore make sure that the call `position` used in the pseudo code for MPR returns the geometric centroid of A and B respectively. Another case where the heuristic cannot give us the exact answer is if the surface of $A \ominus B$ is quadratic in the direction of the origin ray. In that case we would need an infinite number of iterations to find the exact solution which we can not afford and even then we would get numerical errors, but with a suitable numerical tolerance the heuristic will still give a good estimate. Another troublesome case is when MPR detects a collision in phase one (when the internal point and the first portal point are on opposite sides of the origin); The best estimate of the contact normal in this case is the direction of the first point. If A and B are spheres this will actually be the correct contact normal, but if they for instance are two boxes that collide corner to corner our answer will be off by forty-five degrees. Unfortunately there is not much that can be done about this without inventing a whole new algorithm.

Once we have a first estimate of the contact normal we can find the penetration depth by restarting with a slight modification of MPR. Instead of picking an internal point constructed from the geometric centroids of A and B we pick the origin as our internal point. And instead of looking in the direction of the origin from our internal point we now look in the direction of the contact normal estimate computed in the previous step. We then advance the portal until it once again reaches the boundary of $A \ominus B$. When we reach the boundary, a good estimate of the penetration depth will be the point closest to the origin on the final portal.

2.5 Expanding Polytope Algorithm

The Expanding Polytope Algorithm[2] (EPA) is an algorithm used for computing the penetration depth between two objects A and B which we already know have a nonempty intersection. The input which EPA needs is a simplex with vertexes on the boundary of $A \ominus B$ that contains the origin together with support mappings for A and B. The simplex needed as input to EPA is exactly the simplex GJK terminates with for nonempty intersections. MPR's output does not quite meet EPA's input requirements, as the output simplex does not have all of its vertexes on the boundary of $A \ominus B$. Just like GJK and MPR, EPA is an iterative algorithm that works indirectly with the objects through a support mapping, instead of requiring explicit knowledge of the objects so it fits well into the framework we are interested in constructing.

If we have a polytope in $A \ominus B$ that contains the origin we can state the following lower bound on the penetration depth between A and B: The penetration depth must be greater than or equal to the distance from the origin to the face of the polytope that is closest to the origin, since the polytope is contained inside of $A \ominus B$. As a polytope is convex and in this case it contains the origin we also know that the closest point on its closest face must be an internal point of that face (for proof see Theorem 4.8 in [2]). Now assume without loss of generality that our polytope is a convex triangle mesh (any polyhedron can be constructed from triangles), and that we have a support mapping for $A \ominus B$. The following algorithm for computing the penetration depth can be constructed.

Let μ_0 be our best known upper bound on the penetration depth, since we do not have any such bound to begin with we initialize it to infinity. In each iteration i we find the point closest to the origin on our current triangle mesh, let this point be $\mathbf{v}_i$. If we sample our support mapping in the direction of $\mathbf{v}_i$, we get a new point $\mathbf{w}_i = S_{A \ominus B}(\mathbf{v}_i)$. We can now compute a new upper bound on the penetration depth as the minimum of our old bound, and the closest point on the plane with normal $\frac{\mathbf{v}_i}{\|\mathbf{v}_i\|}$ that contains the point $\mathbf{w}_i$; in other words let $\mu_{i+1} = \min(\mu_i, \mathbf{w}_i \cdot \frac{\mathbf{v}_i}{\|\mathbf{v}_i\|})$. Once we have sampled a triangle in our polytope we need to remove it and create new triangles to fill the gap. We do this in the following way:
1) Remove all the triangles whose front sides we can see from $\mathbf{w}_i$ from our mesh, and build a line segment loop from the vertexes that were shared between triangles that face towards $\mathbf{w}_i$ and triangles that face away from $\mathbf{w}_i$.
2) Construct triangles from each pair of bordering vertexes in the line loop together with $\mathbf{w}_i$. These triangles will fill the gap where we removed triangles (see figure 2).

The reason we need to remove all the triangles we can see from $\mathbf{w}_i$ and not just the triangle that $\mathbf{v}_i$ belonged to, is that otherwise our polytope might become concave which would invalidate our penetration depth bounds. Once we have replaced our triangles and updated our bounds we are done with a iteration.

In each iteration we have the following bounds on our estimated penetration depth $\|\mathbf{v}_i\| \leq d \leq \mu_i$, where d is the actual penetration depth. As $\|\mathbf{v}\|$ is monotonically increasing and μ monotonically decreasing, we know that as the number of iterations increase the bounds

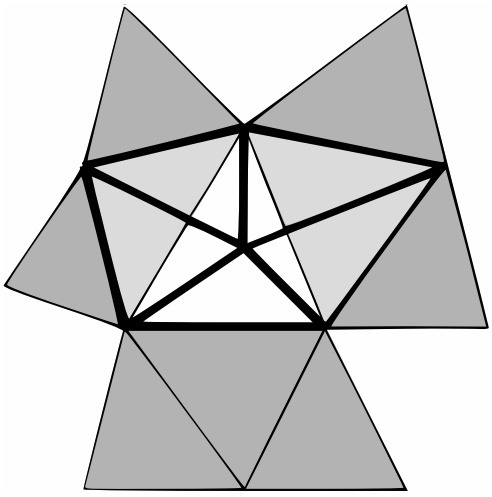

Figure 2: EPA Mesh Update
The triangle mesh seen from above $\mathbf{w}_i$. The central vertex is $\mathbf{w}_i$.
Thin lines represent old triangles. Fat lines represent the new triangles constructed.
Light triangles can be seen from $\mathbf{w}_i$, dark triangles can not.

converge on the true penetration depth. Therefore we pick the following termination condition: if $\mu_i - \|\mathbf{v}_i\| < \epsilon$ for some small epsilon then we consider $\mathbf{v}_i$ to be close enough to the actual penetration depth and return it as our answer.

The information we need for each triangle in our mesh is:

- An array vtx containing the three vertexes in the triangle is needed to construct new triangles and to compute plane normals.
- A point $\mathbf{v}$ which is the point closest to the origin for the plane the the triangle is contained in, is used as sample direction and its length determines the priority of the triangle.
- A scalar $v = \|\mathbf{v}\|$ is needed to determine the order we use to iterate over the triangles.
- Either the barycentric coordinates of $\mathbf{v}$ in the triangle or a Boolean b computed from said coordinates is needed to tell us if $\mathbf{v}$ is an internal point of the triangle.
- An adjacency list in the form of an array adj with pointers to the neighboring triangles is needed to traverse the mesh when deleting triangles that can be seen from $\mathbf{w}$ in an iteration. The array is structured such that adj_i is the triangle that shares edge $vtx_{i\oplus 1} - vtx_i$ (where $\oplus$ is addition modulo 3) with this triangle.
- An array of integers j that keeps track of which edge in which neighbor refers to us in such a way that `this` $\rightarrow$ `adj[i]` $\rightarrow$ `adj[this` $\rightarrow$ `j[i]]` $\rightarrow$ `this`. This array is also needed for the removal of triangles seen from a point $\mathbf{w}$.
- A Boolean *obsolete* that tells us if this triangle is out of date (it became obsolete during some previous pruning of our mesh). The reason we need this information is

that we will store our triangles in a heap and removing obsolete entries from a heap is too expensive.

We call the collection of information listed above for an *Entry*. The point closest to the origin in the current polytope will be the $\mathbf{v}$ of the *Entry* with the lowest v that is not *obsolete* and whose b is true. To find this *Entry* in an efficient manner in each iteration we store all our entries in a min-heap based on v and simply pop elements from the heap until we find an entry that is not *obsolete* and has b set to true.

Once we have found the entry e that contains the point closest to the origin we compute $\mathbf{w}$ and update our bounds as explained above. The front side of entry e can always be seen from the $\mathbf{w}$ computed by sampling in the direction of $e.\mathbf{v}$, so we mark e as obsolete. We then recursively traverse our mesh starting with each entry adjacent to e. For each of these entries we check if it can be seen from $\mathbf{w}$. If so we mark it as obsolete and recursively try its neighbors until we find an entry that can not be seen from $\mathbf{w}$. Once an entry t that we can not see is found we know that we will have to add a new triangle $(t.vtx_{i\oplus 1}, t.vtx_i, \mathbf{w})$ where i is the edge of t that borders the deleted triangles. We add an element (t, i) to a list that describes the line segment loop that encircles the triangles we will delete. An entry t can be see from a point $\mathbf{w}$ if and only if $\mathbf{w}$ is contained in the positive half space of the plane that contains t. This can be tested by checking if $\mathbf{w} \cdot \frac{t.\mathbf{v}}{\|t.\mathbf{v}\|} \geq \|t.\mathbf{v}\|$ or more efficiently as $\mathbf{w} \cdot t.\mathbf{v} \geq \|t.\mathbf{v}\|^2$.

Pseudo code for this flood fill algorithm can be found in Algorithm 10, which takes an entry e, the edge index i that borders the deleted region and the point $\mathbf{w}$. It returns a set of tuples (t, i) such that t is an entry that borders the deleted region and i is the index of the edge in t that borders the deleted region. The $\cup$ operation in Algorithm 10 means concatenation of two ordered sets and not union.

Algorithm 10 Flood Fill Silhouette

```
1:  function SILHOUETTE(e, i, w)
2:      if ¬(e.obsolete) then
3:          if e.v · w < ‖e.v‖² then                    ▷ e can not be seen from w
4:              return {(e, i)}
5:          else                                        ▷ e is visable from w
6:              e.obsolete ← true
7:              A ← SILHOUETTE(e.adj_{i⊕1}, e.j_{i⊕1}, w)
8:              B ← SILHOUETTE(e.adj_{i⊕2}, e.j_{i⊕2}, w)
9:              return A ∪ B
10:         end if
11:     end if
12: end function
```

As `silhouette` traverses the mesh in a counter clockwise direction, the triangles that it returns will be ordered counterclockwise around the deleted region. If we find that we want to delete any entry that can be seen from $\mathbf{w} = S_{A\ominus B}(e.\mathbf{v})$ we can form a counter clockwise loop of triangles bordering the deleted region E by computing:

$$E = \texttt{silhouette}(e.adj_0, e.j_0, \mathbf{w}) \cup \texttt{silhouette}(e.adj_1, e.j_1, \mathbf{w}) \cup \texttt{silhouette}(e.adj_2, e.j_2, \mathbf{w})$$

Once such a loop of triangles have been constructed, a triangle mesh that fills in the deleted region can be constructed in the following way. For each element (e_x, i) of E, construct a new entry $y_x = Entry(e_x.vtx_{i\oplus 1}, e_x.vtx_i, \mathbf{w})$, and connect y_x and e_x such that $y_x.adj_0 = e_x$ and $e_x.adj_i = y_x$. We also need to connect the new entries to each other such that $e_x.adj_1 = e_{x\oplus 1}$ and $e_{x\oplus 1}.adj_2 = e_x$, where $\oplus$ is addition modulo $|E|$. To connect two entries means to set their adj and j fields to appropriate values. Once all the new entries have been constructed, they need to be pushed onto the heap. When this is done we have completed one iteration of EPA.

There is one initiation problem with EPA that we have ignored so far. Namely that the simplex that EPA gets as input might not be a tetrahedron, it could also be a single point, line segment, or triangle. If the simplex has fewer than four points we need to expand it so that we can create our triangle mesh. The different cases of this expansion are covered below. The initiation procedures use Algorithm 11, which takes the four vertexes of a tetrahedron and returns true if and only if the origin is in the convex hull of the tetrahedron.

If the initial simplex is a point, we know that the point must be the origin itself as the simplex must contain the origin. But the vertexes of the initial simplex must be on the surface of $A \ominus B$ which means that the origin is on the surface of $A \ominus B$ so A and B are in touching contact. The penetration depth of a touching contact is always $\mathbf{0}$; thus the solution is to halt and return the zero vector.

If the initial simplex is a line segment consisting of the two endpoints $\mathbf{a}$ and $\mathbf{b}$ we can blow it up to a tetrahedron by doing the following. First find the coordinate axis $\mathbf{e}$ that is closest to being orthogonal to $\mathbf{d} = \mathbf{b} - \mathbf{a}$. The coordinate axis $\mathbf{e}$ in question will be the axis associated with the element of $\mathbf{d}$ with the smallest absolute value. The vector computed as $\mathbf{v}_0 = \mathbf{e} \times \mathbf{d}$ will be a good sampling direction for a new point. If we rotate $\mathbf{v}_0$ by $\frac{2}{3}\pi$ radians around $\mathbf{d}$ we will get a second suitable sample direction $\mathbf{v}_1$. Similarly we compute $\mathbf{v}_2$ as a $\frac{2}{3}\pi$ radian rotation of $\mathbf{v}_1$ around $\mathbf{d}$. Next we compute three more points $\mathbf{x}_0 = S_{A\ominus B}(\mathbf{v}_0)$, $\mathbf{x}_1 = S_{A\ominus B}(\mathbf{v}_1)$, and $\mathbf{x}_2 = S_{A\ominus B}(\mathbf{v}_2)$. The triangle $\mathbf{x}_0\mathbf{x}_1\mathbf{x}_2$ splits the line segment in two parts, and the origin is on the line segment; therefore the origin must be in one of the two tetrahedrons $\mathbf{a}\mathbf{x}_0\mathbf{x}_1\mathbf{x}_2$ or $\mathbf{b}\mathbf{x}_0\mathbf{x}_1\mathbf{x}_2$. We use Algorithm 11 to compute which one. The tetrahedron that contains the origin is used to build our initial triangle mesh.

If the initial simplex is a triangle $\mathbf{abc}$ we simply construct two new points $\mathbf{x}$ and $\mathbf{y}$ by sampling the support mapping in the positive and negative direction of the triangle normal. The origin has to be in one of the three tetrahedrons $\mathbf{abxy}$, $\mathbf{acxy}$ or $\mathbf{bcxy}$; use Algorithm 11 to check which and use that one as our initial tetrahedron.

The initiation procedures described above are constructed to maximize the probability that the origin will not be on the surface of the initial tetrahedron.

Pseudo code for EPA can be found in Algorithm 12, which takes two shapes A and B together with a simplex X, and returns the penetration depth between A and B.

EPA's strength is that it can theoretically compute the penetration depth to within an arbitrary precision. The main downside of EPA is that it often runs into numerical hurdles,

Algorithm 11 Origin in Tetrahedron Test

```
1: function originInTet(a, b, c, d)
2:     n_0 ← (b − a) × (c − a)
3:     if n_0 · a > 0 ≡ n_0 · d > 0 then
4:         return false
5:     end if
6:     n_1 ← (c − b) × (d − b)
7:     if n_1 · b > 0 ≡ n_1 · a > 0 then
8:         return false
9:     end if
10:    n_2 ← (d − c) × (a − c)
11:    if n_2 · c > 0 ≡ n_2 · b > 0 then
12:        return false
13:    end if
14:    n_3 ← (a − d) × (b − d)
15:    if n_3 · d > 0 ≡ n_3 · c > 0 then
16:        return false
17:    end if
18:    return true
19: end function
```

Algorithm 12 Expanding Polytope Algorithm

```
1: function EPA(A, B, X)
2:     touching, Q ← init-mesh(X)                ▷ Create a triangle mesh from our input simplex
3:     if touching then                          ▷ Check if the initiation detected touching contact
4:         return 0
5:     end if
6:     μ ← infinity                              ▷ Initialize the upper bound to infinity
7:     entry ← pop(Q)
8:     repeat
9:         if ¬(entry.obsolete) then
10:            v ← entry.v                       ▷ Let v be our estimated penetration depth
11:            w ← A.support(v) − B.support(−v)
12:            μ ← min(μ, w · v/‖v‖)             ▷ Update our upper bound
13:            if entry.v < μ then
14:                entry.obsolete ← true
15:                E ← ∅
16:                for i = 0, 1, 2 do            ▷ Use silhouette to delete entries and find bordering entries
17:                    E ← E ∪ silhouette(entry.adj_i, entry.j_i, w)
18:                end for
19:                T ← make-entries(E, w)        ▷ Create entries from the elements of E
20:                for t ∈ T do
21:                    if t.b then               ▷ Check that the point closest to the origin in t is an internal point
22:                        push(Q, t)
23:                    end if
24:                end for
25:            end if
26:        end if
27:        entry ← pop(Q)
28:    until entry.v ≥ μ or Q = ∅
29:    return v
30: end function
```

thus finding high precision penetration depths can come at the cost of complete failures. But in cases where one wants to compute the penetration depth with high precision and a small percentage of complete failures is an acceptable price, EPA is a good choice. Other downsides of EPA are that it is both tricky to implement and a relatively slow algorithm. It becomes even harder to implement if computational speed is of high importance. So if exactness in penetration depth computation is not as important as speed, it is probably a

better choice to go with a faster heuristic method.

2.6 Perturbed Sampling Manifold

Unlike the rest of the algorithms in this thesis which are previously published work, the manifold generation algorithm is one of our own devising. The principle of the procedure is to use the support mapping and contact normal for two objects that are known to have a nonempty intersection to generate a contact manifold. If we know that the objects A and B intersect and that they have a contact normal $\mathbf{n}$, then we know that $\mathbf{x}_0 = S_A(-\mathbf{n})$ will be the point in A that is furthest from the collision plane (in the negative half space). Similarly we know that $\mathbf{y}_0 = S_B(\mathbf{n})$ will be the point that is furthest in the positive half space of the collision plane and still in B. Now consider a small perturbation $\mathbf{v} = \mathbf{n} + \mathbf{w}$ where $\mathbf{w}$ is a short vector that is orthogonal to $\mathbf{n}$. Depending on the shape and orientation of A, the point $\mathbf{x}_1 = S_A(-\mathbf{v})$ will have different characteristics. If A has a quadratic surface in the direction of $\mathbf{n}$, then $\mathbf{x}_1$ will be a point close to $\mathbf{x}_0$. If A is not a quadratic shape, and $\mathbf{n}$ points towards a face of A, then $\mathbf{x}_1$ and $\mathbf{x}_0$ will probably be two different vertexes of that face. Similarly if A is not a quadratic shape and has an edge in the direction of $\mathbf{n}$, then $\mathbf{x}_0$ and $\mathbf{x}_1$ will probably be the endpoints of the edge. Finally if A is not a quadratic and it has a vertex in the direction of $\mathbf{n}$, then $\mathbf{x}_0 = \mathbf{x}_1$ if the perturbation $\mathbf{w}$ is small enough. The same relations hold for the points generated from the support mapping of B. If we construct a small perturbation vector $\mathbf{w}_0$ that is orthogonal to $\mathbf{n}$ and rotate $\mathbf{w}_0$ around $\mathbf{n}$ by $i\frac{2\pi}{k}$ radians, where k is the number of samples we want, and $1 \leq i \leq k$ is the index of a sample, we will get a series of perturbation vectors $\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_k$. If we add each of these to the contact normal we will get a series of sampling directions $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_k$ such that the convex hull of the points generated by sampling once for each of these directions will be an approximation of the feature of the shape in the direction of the contact normal.

If we generate the above approximation points for both shapes and project them on the collision plane we will get two polygons describing the silhouettes of the two objects on the collision plane. If we clip the two polygons against each other the resulting polygon will be an approximation of the contact manifold.

Pseudo code for the sampling of a feature from an object is shown in Algorithm 13, which takes a shape X, the contact normal $\mathbf{n}$, a perturbation vector $\mathbf{w}$ orthogonal to $\mathbf{n}$, and a point $\mathbf{p}$ in the collisions plane. The algorithm returns a set of points S in the plane $\mathbf{n}_1\mathbf{e}_1 + ... + \mathbf{n}_d\mathbf{e}_d = 0$ such that $S \oplus \{\mathbf{p}\}$ is the silhouette of the feature from X in the direction $\mathbf{n}$ projected on the collision plane.

Now that we have the two polygons X and Y we need to clip them against each other, our strategy for doing this is done in two steps. The first of these steps is done in the following way. For each vertex of X we check if it is inside Y; if so we know that it will be a vertex of the clipped polygon so we insert it into set Z. We then test each vertex of Y against X in the same way and add any points inside X to Z. The second step is to check if any edge from X intersects any edge from Y, in which case the point of intersection will be a vertex

Algorithm 13 Sample Feature by Perturbation

```
1: function SampleFeature(X, n, w, p, nr)
2:     R ← Matrix(2π/nr, n)        ▷ Create a matrix that rotates a point by 2π/nr radians around n
3:     S ← ∅
4:     for i ← 1, 2, ..., nr do
5:         x ← X.support(n + w) − p        ▷ Sample and subtract the point from the collision plane
6:         x ← x − n(n · x)        ▷ Project x on the plane through the origin with normal n
7:         X ← X ∪ {x}        ▷ Add x to the return set if it does not already exist
8:         w ← Rw        ▷ Compute new perturbation vector by rotating around n
9:     end for
10:    return S
11: end function
```

of the contact manifold so we add that point to Z. Once these two steps are completed, Z will be our approximation of the contact manifold.

To check if a point $\mathbf{p}$ in the same plane as a polygon P is inside the polygon: Compute the "center" of P as $\mathbf{c} = \frac{1}{|P|}\sum \mathbf{p}_i,\ \mathbf{p}_i \in P$. For each edge $\mathbf{e}_i = \mathbf{p}_{i\oplus 1} - \mathbf{p}_i$ compute a vector that is orthogonal to both the polygon normal $\mathbf{n}$ and $\mathbf{e}_i$ as $\mathbf{u}_i = \mathbf{n} \times \mathbf{e}_i$. Thereafter construct vectors $\mathbf{v}_i = \mathbf{p}_i - \mathbf{p}$, and $\mathbf{w}_i = \mathbf{p}_i - \mathbf{c}$. Compute $\mathbf{x}_i = (\mathbf{v}_i \cdot \mathbf{u}_i)(\mathbf{w}_i \cdot \mathbf{u}_i)$; $\mathbf{p}$ will be on the "inside" of edge $\mathbf{e}_i$ if and only if $\mathbf{x}_i \geq 0$. Test if $\mathbf{x}_i < 0$ for any i; if so $\mathbf{p}$ is not in P, otherwise $\mathbf{p}$ must be in P.

To check of if two edges from the different polygons intersect we first compute the shortest line between the infinite lines that contain the two edges. Given two lines $P_a = \mathbf{p}_1 + \mu_a(\mathbf{p}_2 - \mathbf{p}_1)$ and $P_b = \mathbf{p}_3 + \mu_b(\mathbf{p}_4 - \mathbf{p}_3)$, the shortest distance between them can be found by minimizing

$$\|\mathbf{p}_1 - \mathbf{p}_3 + \mu_a(\mathbf{p}_2 - \mathbf{p}_1) - \mu_b(\mathbf{p}_4 - \mathbf{p}_3)\|^2 \tag{10}$$

Another way to find the shortest line is to find the line that is orthogonal to both of the original lines.

$$(\mathbf{p}_1 - \mathbf{p}_3 + \mu_a(\mathbf{p}_2 - \mathbf{p}_1) - \mu_b(\mathbf{p}_4 - \mathbf{p}_3)) \cdot (\mathbf{p}_2 - \mathbf{p}_1) = 0 \tag{11}$$

$$(\mathbf{p}_1 - \mathbf{p}_3 + \mu_a(\mathbf{p}_2 - \mathbf{p}_1) - \mu_b(\mathbf{p}_4 - \mathbf{p}_3)) \cdot (\mathbf{p}_4 - \mathbf{p}_3) = 0 \tag{12}$$

Both of these formulations lead to the same equations, and as long as the lines are not parallel the solution will be unique. The exact method for solving the above problem for μ_a and μ_b can be found in reference [9]. To check if two edges intersect we first compute the lines that contain the edges. Thereafter we compute the shortest line segment between these lines. Once we have this line segment, $(\mathbf{p}_a, \mathbf{p}_b)$ we can check if the edges intersect. The edges intersect if and only if $\mathbf{p}_a = \mathbf{p}_b \wedge 0 \leq \mu_a, \mu_b \leq 1$.

There are a few downsides to the perturbed sampling manifold method. The first one is that a small change in the direction of the contact normal can result in a large change in the contact manifold, which is undesirable in some situations such as when stacking

objects in a physical simulation. Another problem arises for large quadratic shapes, as even a small perturbation can lead to sample points "far apart" if the object is large enough. Finally the methods for checking if points from X are inside Y or the other way around can give false positives for some polygons, which causes the generation of incorrect manifold approximations. The advantages of this method compared to methods that collect and cache contact points between frames (using temporal coherence to construct the manifold), is that it always constructs an up-to-date manifold. An example of where this can be an advantage is the manifold of a box sliding across a plane: the faster the box is sliding, the larger the temporal manifold becomes. The perturbation method does not suffer from this problem.

3 Implementation

This section will present the reference implementation that will be used for the tests in the next section. The goal is to motivate and explain how and why the implementation was made. The focus of this thesis is on working with implicit objects. Therefore an object oriented language is suitable for our reference implementation. We also focus on algorithms and methods usable in "soft real-time" (wall clock time is roughly equal to simulation time but we do not set hard deadlines), so we want to have relatively high performance. Therefore we choose to work in C++.

3.1 Vector Library

The collision algorithms studied in this thesis are dependent on efficient 3D vector computation. But efficient implementation of a vector library is outside the scope of this thesis. Therefore instead of trying to implement a rudimentary vector library as part of the thesis we choose to use Sony's vector math library which has been released as open source under the BSD license and is hosted as part of the Bullet Physics SDK since version 2.55.[2] Although Sony's vector library comes with both C and C++ interfaces, we use the C++ version as we want to implement our implicit objects as a class interface. The library has several different architectural versions: scalar (single precision), Power-PC Altivec, Cell SPU, and x86 SSE. Unfortunately it does not come with a double precision (DP) version for the scalar implementation. A double precision version would be useful as a baseline for detecting when errors are caused by numerical problems. Therefore we added a double precision version of the portable scalar implementation. The double precision version works exactly as the single precision one, and was created from the scalar version by simply changing the types to `double` and replacing standard library function calls (e.g. `sinf(a)`, replaced by `sin(a)`). Since the elements of a vector or matrix can be in either single or double precision we also `typedef` our own scalar type `Scalar`, that is set as `float` when working in single precision and `double` when working in double precision.

3.2 Base Classes

In this section we will go over how the basic building blocks or the reference implementation are constructed and how they fit together as a hierarchy.

3.2.1 Shape

The goal of having a simple common interface for all objects we want to do collision detection on, can be fulfilled by demanding that all shapes implement a common interface. We define

[2] Sony's Vector math Library can be downloaded from this URL:
http://sourceforge.net/project/showfiles.php?group_id=147573

this common interface as an abstract class that all objects must inherit from. We call this base class for `Shape`.

The class `Shape` demands that any inheriting class implements the three methods listed below:

- `Vector3 support(const Vector3 &v)`:
  The `support` function must return a point $\mathbf{p}$ in the object X such that $\mathbf{p} \cdot \mathbf{v} \geq \mathbf{x_i} \cdot \mathbf{v}$, $\forall \mathbf{x_i} \in X$. In other words it is the support mapping described in Section 2.2. It is the central component of all our algorithms.

- `void getAABB(Vector3 &minimi, Vector3 &maximi)`:
  The `getAABB` function must set `minimi` and `maximi` such that they define an axis aligned bounding box (AABB) that completely contains the object. This function is not needed for any of the algorithms covered in the algorithm section, but it is helpful when defining a broad phase pruning step for culling collision tests. It is used by the physics simulation that is part of the test suit. The function is not strictly needed, as an AABB can be constructed by sampling in the positive and negative axis directions, but constructing it in this way can be very inefficient compared to computing the AABB with explicit knowledge of what a specific shape is.

- `bool quadric(const Vector3 &v)`:
  The function `quadric` should return `true` if the object has a quadratic surface in direction $\mathbf{v}$. The reason that this function is needed is that the perturbed sampling manifold method runs into numerical problems when trying to compute a manifold for some quadratic surfaces. For convenience sake, `Shape` has a default implementation of this function that returns `false` for all direction so that the function only needs to be overloaded for quadratic shapes.

3.2.2 Objects

The `Shape` interface makes it possible to sample the support function for objects so we can run our algorithms, but we cannot take any actions based on the results as it is still impossible to translate or rotate an object through the `Shape` interface. To solve this limitation we construct another abstract class, that gives a `Shape` a position, an orientation and defines methods for modifying these. We call this new class `Object`. An `Object` is a subclass of `Shape` that contains two member variables: Its position given as a world to object translation `m_x`, which is stored as a three dimensional vector. And its orientation given as a object to world rotation `m_q`, which is stored as a quaternion.

In addition `Object` defines constructors, setters, getters and the following member functions:

- `void move(const Vector &v)`:
  This function moves the object by adding v to its position `m_x+=`v.

- `void rotate(const Quat &q)`:
  This function rotates the object by quaternion multiplication `m_q*=q`. In addition it makes sure that the updated m_q is normalized.

- `void rotate(const Vector3 &w)`:
  This function applies a rotation of $\|\mathbf{w}\|$ radians around the axis $\frac{\mathbf{w}}{\|\mathbf{w}\|}$, and normalizes the new orientation to unit length.

It should be noted that the fields `m_x`, and `m_q`, are public so a user does not have to go through the interface above. The reasoning behind this is that there are no illegal values for the orientation and position, so limiting access to the variables does not serve any purpose in itself.

We then construct a number of unmovable basic primitives: `CPoint`, `CSphere`, `CDisc`, `CSegment`, `CRectangle`, `CEllipse`, `CBox`, `CEllipoid`, and `CPolyhedron`. The C in the beginning of each name stands for centered. They are all subclasses of `Shape` with support mappings defined similar to what is shown in Table 1, with the difference that the origin is the geometric centroid[8] of the shape's volume (the exception is `CPolyhedron` where for simplicity's sake the origin is the average of the vertexes that make up the polyhedron).

Once we have the unmovable versions of the basic primitives we use them to construct movable versions of the same primitives: `Point`, `Sphere`, `Disc`, `Segment`, `Rectangle`, `Ellipse`, `Box`, `Ellipsoid` and `Polyhedron`. These are simply defined as subclasses of `Object`, with an additional member field of the appropriate unmovable type, and a support mapping defined as in Equation 5. In addition to these basic primitives we create classes for the combined shapes given in Table 1, `Capsule`, `Lozenge`, `RoundBox`, `Cylinder`, `Cone`, `Wheel`, and `Frustum`, except that we make sure that the origin of the local space once again is the geometric centroid of the shape.

Finally we create three general shape combination classes:
`SumObject`, which is an object of the type $A \oplus B$, which is constructed from pointers to two already constructed `Shape` instances.
`MaxObject`, which is the convex hull of two already defined `Shape` instances.
`CSO`, which is an object of the type $A \ominus B$ for two already defined `Shape` instances A and B.
This collection of class definitions give us a broad base of shapes to test our algorithms with.

3.3 Gilbert Johnson Keerthi's Algorithm

The reference implementation of the GJK based algorithms are written as a collection of c-style functions in the `namespace GJK`. Internally the functions all work with a `struct gjkData`. The reason for using this structure instead of a class with member functions is to avoid forcing the user to create an extra object instance just to compute collision data. To represent a simplex in the case that the user wants to access the final simplex, the GJK functions use a simple `struct Simplex`. A `Simplex struct` contains an array of four three-dimensional vectors and a bit field that specifies which of the array indexes are in use. The GJK based functions that were implemented are the following:

- `bool distance(const Object *a, const Object *b, Vector3 *dist, Simplex *simplex_out=0, const Vector3 *guess=0)`:
  The `distance` function measures the distance between two Objects `a` and `b`. It returns `true` if and only if `a` and `b` intersect. If `a` and `b` do not intersect it sets `dist` to the shortest distance vector from `b` to `a`. If `simplex_out` is not `NULL` the final simplex is written to `simplex_out`. If `guess` is not `NULL` the initial search direction will be the vector to which `guess` points, instead of the vector from `b` to `a`'s center.

- `bool sa_gjk(const Object *a, const Object *b, Simplex *simplex_out=0, const Vector3 *guess=0)`:
  The `sa_gjk` function is an implementation of the separating axis version of GJK. It returns `true` if and only if the two objects `a` and `b` have a nonempty intersection. If `simplex_out` is not `NULL` the final simplex is written to `simplex_out`, and if `guess` is not `NULL` the vector `guess` points to is used as the initial search direction.

- `bool collision(const Object *a, const Object *b, Vector3 *depth, Simplex *simplex_out=0, const Vector3 *guess=0, const Scalar mu=0.05)`:
  The `collision` function is an implementation of the swept shape version of GJK, it uses EPA as a backup routine for deep intersections. It returns `true` if and only if the two objects `a` and `b` are within a distance `2*mu` of each other. If it returns `true`, then `depth` is set to the penetration depth between `a` and `b`. If `simplex_out` is not `NULL` the final simplex is written to `simplex_out`. If `guess` is not `NULL` then `guess` is used as the initial search direction. Finally `mu` is the size of sweep margin we want to use.

The internal data type `struct gjkData` is in actuality a class in the sense that it has its own methods. Everything in `gjkData` is public so it has no private access encapsulation and the constructor only handles part of the initialization thus the user must know what he is doing to use it. It defines the following nontrivial methods:

- `void addPoint(const Vector3 &p)`:
  The `addPoint` method adds a point p to the current simplex. It does not check if there is room for a new point; thus if a user tries to add a point to a simplex that already has four points the fourth point will be overwritten.

- `Vector3 closest()`:
  The `closest` method is the central functionality of the `gjkData` struct. It computes the point closest to the origin in the current simplex. It uses the Voronoi region based method presented in Section 2.3.2 for doing this. The reason that it uses this algorithm instead of Johnson's algorithm is that we found that Johnson's algorithm ran into numerical problems when computing with single precision. These numerical problems were manageable for `distance` and for `sa_gjk`, but when implementing the swept collision routine the numerical imprecision can result in a situation where we know that the two objects do not intersect but we cannot tell if they are within `2*mu` of each other. For this reason we use the Voronoi region based algorithm which does not suffer from this numerical problem.

3.4 Minkowski Portal Refinement

The implementations of the MPR based algorithms are also constructed as c-style functions in the `namespace MPR`. The MPR based functions are:

- `Vector3 penDepth(const Object *a, const Object *b, const Vector3 &dir)`:
  The `penDepth` function returns the penetration depth for the two objects `a` and `b` in the direction `dir`. The vector dir does not have to be normalized. The only precondition for the method is that the intersection between a and b must be nonempty.
- `bool intersection(const Object *a, const Object *b, Vector3 *pen_depth=0)`:
  The `intersection` function returns `true` if and only if the objects `a` and `b` have a nonempty intersection. If `pen_depth` is not `NULL` a heuristic estimate of the penetration depth is written to the vector `pen_depth` points to.

Just like in the GJK implementation these functions use an internal `struct mprData` that keeps track of the internal point and portal, and defines methods for the basic operations.

- `bool findPortal()`:
  The `findPortal` method implements the second phase of MPR as described in Algorithm 7, but instead of returning the internal point and portal it updates the values in the `mprData` instance for which it is called. If `findPortal` returns `true` then the origin ray passes through the portal; otherwise the max number of iterations have been reached without finding a legal portal.
- `bool refinePortal()`:
  The `refinePortal` method implements the third phase of MPR as described in Algorithm 8. But instead of returning the simplex it updates the `mprData` instance for which it was called. If `refinePortal` returns `true` then the origin is in the simplex (in other words we have a collision), and if `refinePortal` returns `false` the origin is either in the portal and we have a touching contact or it is on the far side of the last support plane (therefore the objects do not collide).

- `Vector3 findCollisionNormal()`:
  The `findCollisionNormal` method advances the portal to the boundary of $A \ominus B$ and returns the portal normal as an approximation of the collision normal. The precondition for this method is thus that the origin is inside of $A \ominus B$. The method has a maximum iteration limit so for quadratic shapes the returned normal can be a quite rough approximation.

3.5 Expanding Polytope Algorithm

Similar to the structure used previously the implementation of EPA is a c-style function in the `namespace EPA`:

- `bool penDepth(const Object *a, const Object *b, const Simplex *simplex)`:
  This function returns `true` if it manages to compute a penetration depth vector that is not the zero vector (in other words it returns false if it finds that `a` and `b` are in touching contact). The preconditions for the function are the following: `a` and `b` must have a nonempty intersection, the convex hull of `simplex` must contain the origin, and the vertexes of `simplex` must be on the boundary of $A \ominus B$.

3.6 Manifold Construction

The manifold construction is also constructed as c-style functions, the job of the manifold construction is to fill out a struct Collision that describes a collision. A Collision struct is defined as follows:

```
struct Collision
{
  Vector3 n;       //the collision normal
  Vector3 p[16];   //the contact points in world-space
  int     nrp;     //the number of contact points in p
  Scalar  d;       //the depth of intersection
  int     idA;     //identifier of object a in collision
  int     idB;     //identifier of object b in collision
  //note: idA and idB are not set by the manifold construction
}
```

The function for manifold construction is the following:

- `void buildManifold(const Object *a, const Object *b,`
  `const Vector3 *depth, Collision *C)`:
  This function fills in all fields of `Collision` object `C`, except for the object identifiers `idA` and `idB`. The precondition for `buildManifold` is that `a` and `b` have a nonempty intersection and that `depth` is the penetration depth vector for the collision between `a` and `b`.

4 Results

This section will present the results from the different tests that were performed. All tests were run on a 2.4GHz Intel Core 2 Duo with 4MB L2 cache and 4GB of RAM.

4.1 Randomized Benchmark

Our first test suit was a randomized benchmark. The focus of the benchmark is to measure the number of iterations and run-time that the different algorithms use for a number of test cases in an attempt to find trends in their behavior. The benchmark is run using both double precision (DP), and SSE implementations of the vector library, so that we can estimate the effects of the number format on the behavior of the algorithms. The random configurations will of course not be representative of any real use for our algorithms but will hopefully give guiding information for many real problems. The reason for using a random benchmark instead of multiple real world applications is simply one of convenience as we do not have access to suitable real world applications or even knowledge about enough suitable areas to test.

The benchmark uses five different shapes which are listed below, sizes are given in engine distance units so only the proportions are of interest.

- A Box with dimensions $2 \times 2 \times 2$ is used as an example of a cheap polygonal object.
- An Ellipsoid with dimensions $1 \times 2 \times 1.5$, should give an indication of how well the algorithms handle purely quadratic shapes.
- A Round-Box constructed from a $2 \times 2 \times 2$ Box and a sphere with radius 0.5, should illustrate the behavior of objects composed from a combination of flat surfaces and quadratic surfaces.
- A Polyhedron constructed from the convex hull of 50 random points on a unit sphere, used to test how the algorithms handle expensive[3] support mappings and more complex polygonal models.
- A cone with base radius 1 and height 2, is used in comparison to the ellipsoid to test if different types of quadratic surfaces differ in behavior.

An image of the five different shapes we used can be seen in Figure 3.

The benchmark tests six different types of object combinations, Box vs. Box, Ellipsoid vs. Ellipsoid, Box vs Ellipsoid, Round-Box vs. Round-Box, Polyhedron vs. Polyhedron and Cone vs. Cone. A configuration consists of randomly generating a position and orientation for two objects withing a given cube of space. By scaling the size of the cube in which we

[3]Fifty vertexes would normally not be that expensive but since the support mapping is naively implemented as a linear search it is quite expensive compared to the other shapes tested.

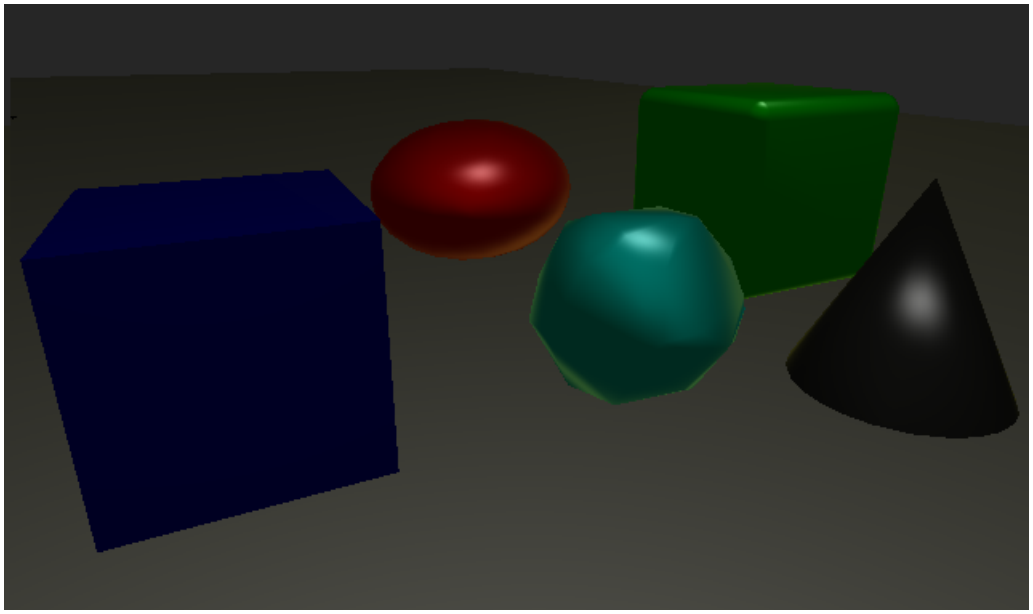

Figure 3: Test Shapes

The test shapes that where used in our benchmark. Box (blue), Ellipsoid (red) Round-Box (green), Polyhedron (cyan), Cone (black).

generate the positions, we can control the probability of intersections and the average penetration depth. Each configuration is tested for intersection using both MPR and separating axis GJK. Distance for each configuration is measured with default GJK. Finally penetration depth is computed using the MPR based heuristic, and a combination of SA-GJK and EPA for each configuration. To get a good statistical base for the measurements each test is computed as the average of a million random configurations. The same random positions were reused for all of the algorithms. But a different position sequence was used for DP and SSE as switching floating point model requires a recompile of the library. For each test the following properties were measured:

Average Iteration Count:
The average iterations or the "average work" can be computed by counting the number of support calls that each algorithm uses on average for a configuration. This measurement does not give us any useful information about how different algorithms compare to each other in overall efficiency as we ignore how costly the work in each iteration is. However it does give us a hint of how the algorithms scale with the cost of the support mapping, and how troublesome the algorithm finds different shape combinations.

Maximum Iteration Count:
As all of our algorithms are iterative in nature they can get stuck in infinite loops because of numerical errors. While some such errors can be detected, this is not the case for all such errors. To make sure that we do not get stuck in this way it is common to set a maximum iteration count for an algorithm. We would like to know how high such a cutoff needs to be for the different algorithms. To estimate this we set a very high cutoff and computed the average of the 1000 highest iteration counts (once again by counting support calls) that completed under our cutoff value. This value can also be compared to the average iteration count to estimate the standard deviation in the iteration count for an algorithm and shape configuration.

Numerical Failures:
Due to the random nature of the benchmark it is impossible for us to know the correct result for a test configuration. Since we do not have a trusted algorithm for verification it is impossible for us to measure the size of our numerical errors. What we can measure is the frequency of catastrophic failures for which the algorithms give no meaningful answer at all (if it for instance got stuck in an infinite loop) or give answers that are so bad that it is obvious that they cannot be true (for instance if the penetration depth vector $\mathbf{x}$ points in the opposite direction from the vector from B to A, i.e. $(\mathbf{a} - \mathbf{b}) \cdot \mathbf{x} < 0$).

Average Run-time:
We would also like to compare the practical run-time cost of the different test cases. We do this by computing the average run-time for each test case. Unfortunately we can not measure the range that the run-time falls in as the time is low enough that the reliability of anything except the average is compromised by noise from the operating system and other background applications.

Spatial Properties of Configurations:
To give a general overview of the average configurations that are generated, Figure 4 shows the average intersection depth as measured by both MPR-extension and EPA (nonintersecting objects were included in the average as if they had a intersection depth of zero). Figure 5 shows the average distance between objects, measured with GJK (intersecting objects are included in the average using a distance of zero).

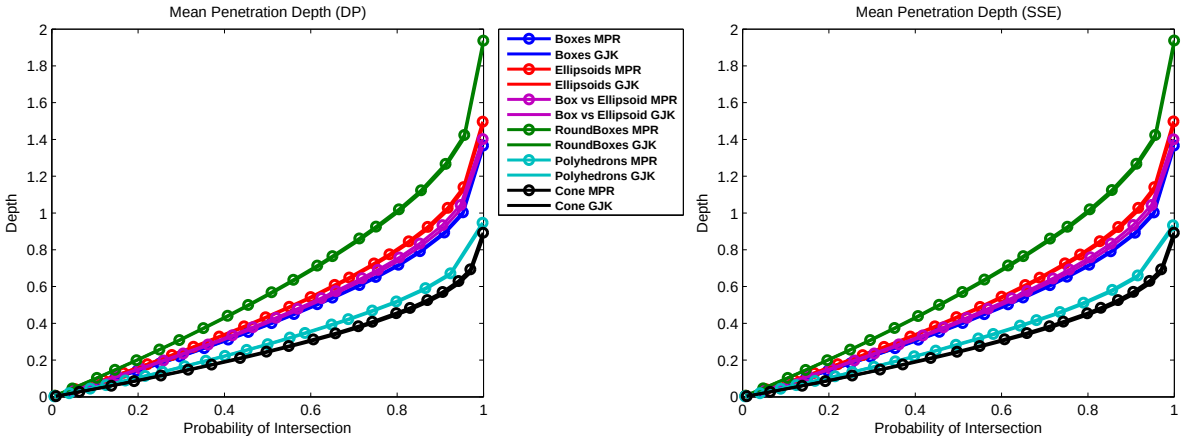

Figure 4: Mean Penetration Depth for the Test Configuration

Plots the mean penetration depth against the probability of intersection. Left plot shows results for DP implementation, right plot shows results for SSE implementation. Note that the difference between measuring algorithm and numerical implementation is too small to measure.

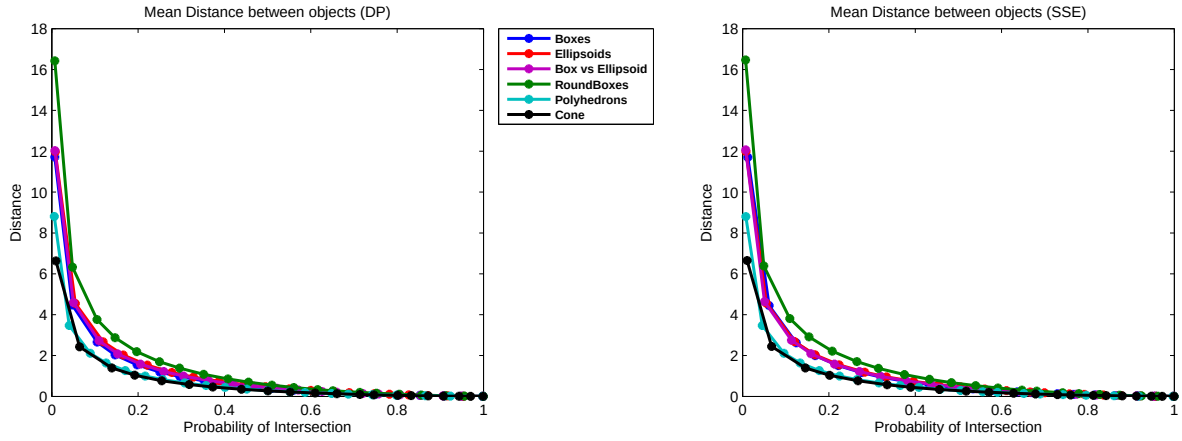

Figure 5: Mean Distance Between Objects for the Test Configuration

Plots the mean distance between objects against the probability of intersection. Left plot shows results for DP implementation, right plot shows results for SSE implementation. Note that the difference between numerical implementation is too small to measure.

4.1.1 Intersection Testing

We will now examine how the algorithms for intersection testing behave in our benchmark. When testing for intersections we compare SA-GJK to MPR. The different properties we tested will each be presented in a figure. Each figure shows which object configuration and intersection algorithm was used for each line in the legend. For example the line marked "Boxes MPR" shows the result of intersection tests between two boxes using MPR, and the line marked "Box vs Ellipsoid GJK" shows the results of testing a Box and an Ellipsoid using SA-GJK.

Average Iteration Count:
Figure 6 indicates that neither algorithm needs very many support calls to determine if two objects intersect in the average case. We also note that the choice of floating point model does not seem to affect the number of support calls needed by either SA-GJK or MPR. MPR seems to use roughly one support call less than GJK; the reason for this is that MPR's first point is computed without using the support mapping for the objects, so in practice they use a similar number of iterations for the same object configuration. The last thing that is noteworthy from this test is that it seems to be slightly harder for MPR to detect collisions between objects that are composed mostly of quadratic surfaces (Box vs Ellipsoid, Ellipsoid vs Ellipsoid, and Cone vs Cone). SA-GJK does not show this clear separation of the tests in two groups.

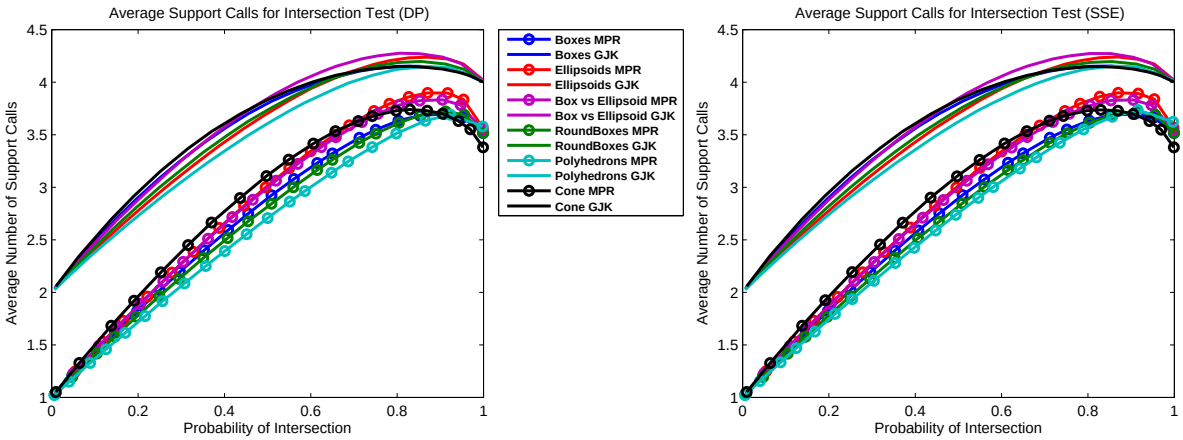

Figure 6: Mean Iterations used by Intersection Test

The average number of support calls needed to determine intersection. The left figure uses double precision floating point vectors, the right one uses SSE based vectors.

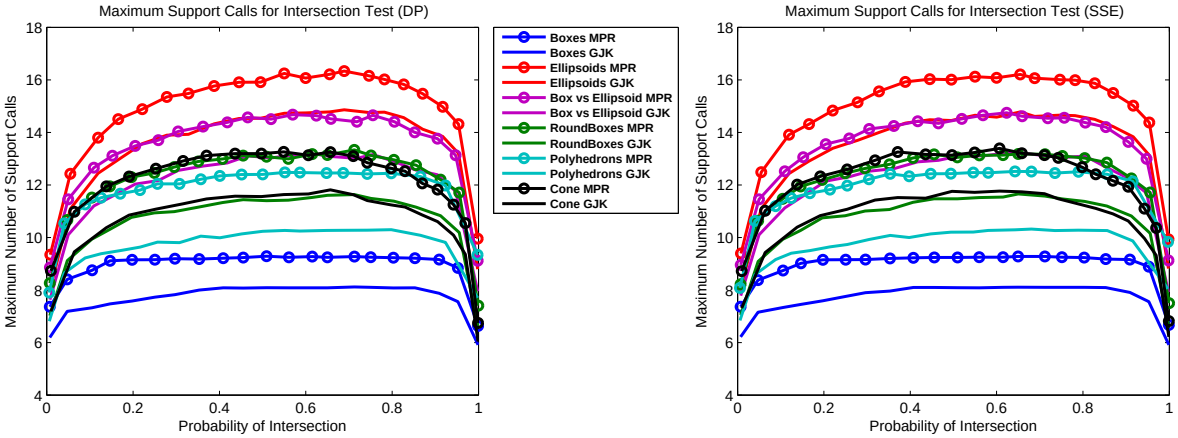

Figure 7: Maximum Iterations used by Intersection Test

The Maximum number of support calls needed to determine intersection. The left figure uses double precision floating point vectors, the right one uses SSE based vectors.

Maximum Iteration Count:

Figure 7 shows the worst case behavior for the two algorithms instead. Here we can once again see no noticeable difference between single precision and double precision floating point models. But we can easily see that MPR uses roughly two to three more iterations than GJK in the worst case for all shape configurations tested. We also note that the order of the configurations is the same for GJK and MPR when ordered on number of iterations used in the worst case.

Numerical Failures:

We do not present any plots with frequency of numerical failures as they were too infrequent to measure for all intersection tests.

Average Run-time:

The next property we test is the average time needed for the different algorithms and vector libraries. The results from this test can be seen in Figure 8. The first thing we note is that using the SSE implementation of the vector library instead of the double precision version, gives a speedup somewhere between 0% and 25% for all tested object types except the random polyhedrons and cones which become slower. The reason SSE does not speed up the run-time for Polyhedrons and cones is that the run-time cost for the support mappings of these two object types are dominated by the computation of one or more dot products. SSE does not include any instructions for computing the sum of the elements in a "SSE register" (although such instructions exist in later versions of SSE) so the naive implementation of a

dot product using SSE is roughly twice as expensive as a compiler optimized scalar implementation. If one wants to compute a lot of dot products the problem can be circumvented by computing four products in parallel, but we did not implement this version so we do not know how its run-time cost compares to the scalar version. What we did instead was to force the polyhedron support mapping to use the scalar version even when using SSE vectors. As the dot product is not as dominant to the cost of computing the cones support mapping we ignored the problem there.

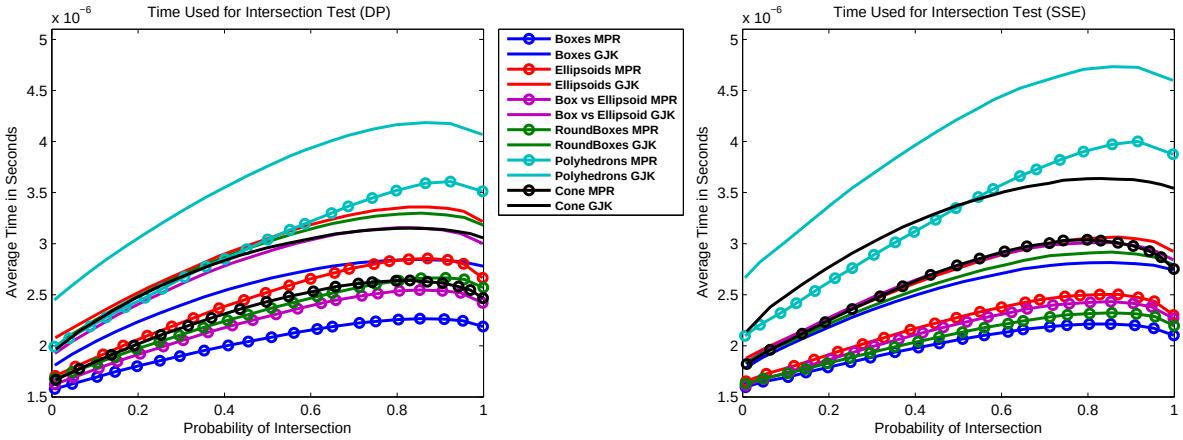

Figure 8: Mean Time used by Intersection Test

Average Run-time needed to determine intersection for different object and algorithm configurations. The left figure uses double precision floating point vectors, the right one uses SSE vectors.

We can also see that MPR in all cases was faster than GJK for the same object configuration. Another observation is that the cost of evaluating the support mapping dominates the cost of computation as the random polyhedrons test is significantly more expensive than all the other tests, even though it uses the same number of support calls on average (or even noticeably fewer in some cases). Finally we note that the time needed by GJK is proportional to the time needed by MPR for all cases. Thus the trend that MPR has a harder time working with quadratic surfaces than GJK, which we noted by counting the number of iterations used, does not seem to have any significant effect on the computation time.

4.1.2 Penetration Depth Computation

We also want to study the behavior of the penetration depth computation algorithms. As the penetration depth algorithms use the results of the intersection tests as input, the support calls and time needed for intersection testing are included in the following test results.

Average Iteration Count:
Figure 9 shows the number of support calls needed on average when computing the penetration depth for the different tests. First we note that MPR uses more iterations than EPA to compute the intersection depth, even though they use the same error tolerance. The reason for this is MPR's two pass algorithm which advances the portal to the surface of the shape twice. The reason that MPR does not use twice as many support calls is that it only advances one triangle of a tetrahedron, while EPA advances many triangles in a triangle mesh. We can also see that the floating point model used does not seem to affect the number of iterations needed. If we look closer at the different objects with quadratic surfaces we see that both MPR and EPA use the most iterations on ellipsoids, followed by ellipsoid vs box and cones. Both algorithms use about the same number of iterations for ellipsoids but differ on the other two shape configurations. Specifically MPR uses roughly the same number of iterations for box vs ellipsoid as it does for cones, whereas EPA needs significantly fewer iterations for cones than for box vs ellipsoid. This would seem to indicate that MPR has a harder time with cones, cylinders, round frustums and similar shapes than EPA does but to state that this is definitely the case we would need more test data.

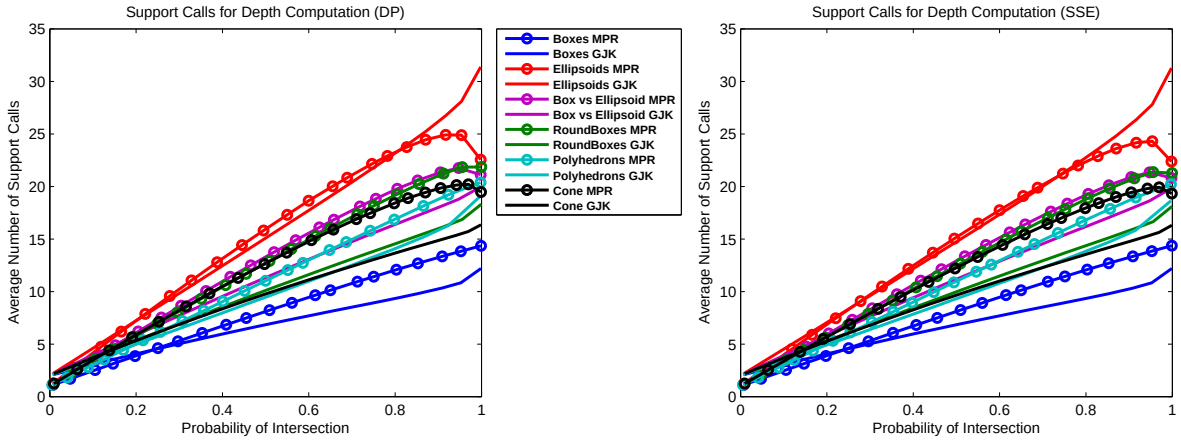

Figure 9: Mean Iterations used by Depth Computation

The average number of support calls needed to determine penetration depth . The left figure uses double precision floating point vectors, the right one uses SSE based vectors.

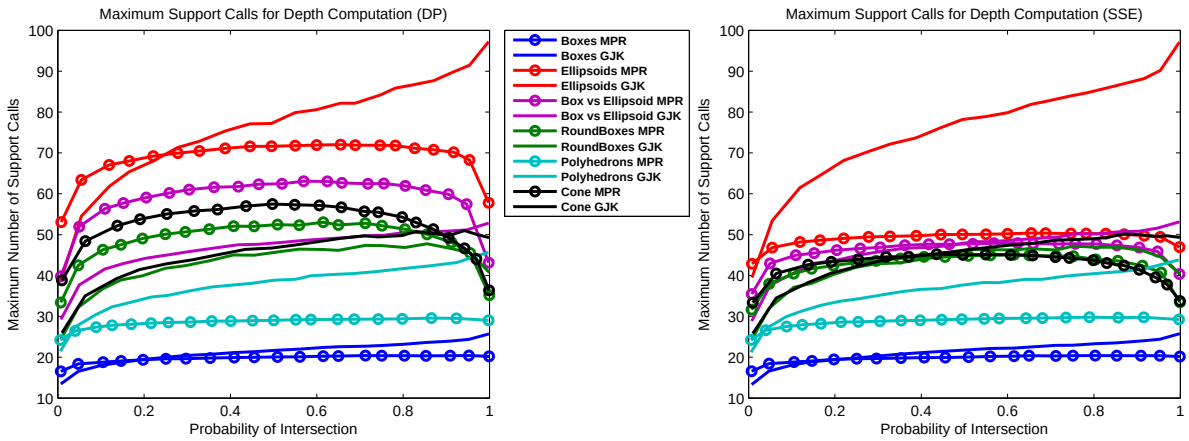

Figure 10: Maximum Iterations used by Depth Computation

The Maximum number of support calls needed to determine penetration depth . The left figure uses double precision floating point vectors, the right one uses SSE based vectors.

Maximum Iteration Count:

Figure 10 shows that even though the average case did not differ noticeably between DP and SSE implementations the worst case behavior for the MPR based routine is significantly changed when switching from one to the other. We believe the cause of this is the termination condition that says that if the new vertex found in this iteration is already in the portal (within a numerical tolerance) we halt. With lower numerical precision the number of possible vertexes that we can represent decreases so the algorithm will tend to halt earlier. EPA does not have this type of halting condition so its behavior is not affected in the same way. But as the large change in worst case behavior does not seem to affect the average case it seems safe to assume that these cases are very uncommon. We also note that although computing the penetration depth between two cones is easier in the average case than computing the penetration depth for two RoundBoxes, the reverse is true in the worst case.

Numerical Failures:

Figure 11 plots the frequency of numerical failures in depth computations. As can be seen the error frequency varies dramatically between algorithms, shapes and numerical models. MPR has a significantly lower error frequency than GJK+EPA for all cases because of its heuristic nature. We also note that the random polyhedron and the box are strongly affected by a switch from double to single precision floating point numbers. A detail that we do not have a good explanation for is that the failure frequency for quadratic shapes is actually higher with DP than with SP. If we look at the graphs as a whole we can see that for all tests except the random polyhedrons with SP GJK+EPA the error rate is less then 2 times in a thousand. Although this error rate is rather high it should be acceptable in a continuous

simulation as the likelihood of consecutive failures is low. Therefore as long as our simulation can handle some failures these methods should work fine; otherwise we probably need to look elsewhere for penetration depth algorithms.

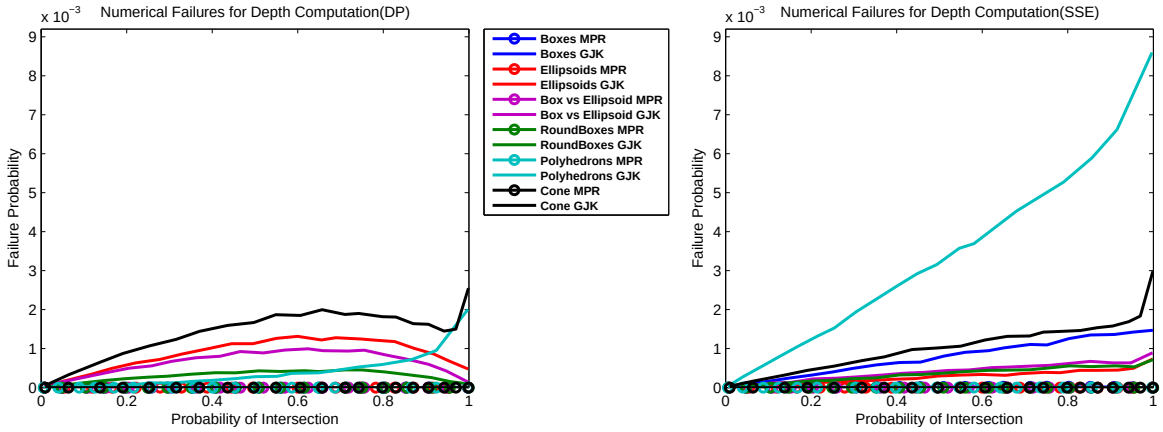

Figure 11: Failure Frequency for Depth Computation

The frequency of numerical failures for depth computation plotted against the probability of intersection. The left figure uses double precision floating point vectors, the right one uses SSE based vectors.

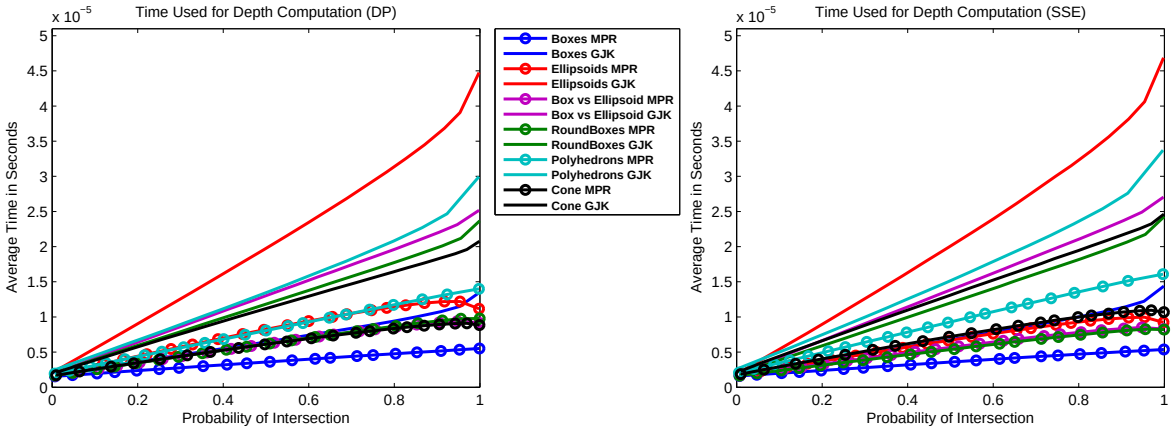

Figure 12: Mean Time used by Depth Computation

Average Run-time needed to determine penetration depth for different object and algorithm configurations. The left figure uses double precision floating point vectors, the right one uses SSE vectors.

Average Run-time:
Figure 12, shows the average timing results for the penetration depth computations. The first thing to note in this graph is that even though the previous test showed that MPR's two pass solution uses more iterations than EPA it is still faster in all of the tests. Although if we had more complex shapes, the increased cost of calling the support mapping would make GJK+EPA a more attractive method. We have not conducted enough tests to determine how complicated the shapes need to be for GJK+EPA to become faster than MPR but since MPR only uses slightly more iterations but is quite a bit faster in time the shapes would probably have to be very costly for GJK+EPA to win out in time complexity. We can also see that using SSE instead of the scalar implementation of the vector library does not give us the same performance boost. Instead of gaining up to 25% as we did for intersection tests we are gaining at most a few percent. Finally we note that all of the EPA based tests have a steep increase in time for the last five percent of the graphs, which indicates that EPA increases in cost more for deep intersections than the MPR based heuristic does. The same increase was present in the iteration count measurements above.

4.1.3 Distance Measurement

In this section we will look at the behavior of GJK when used for measuring the distance between two nonintersecting objects. We have two main reasons for this examination. Continuous collision detection schemes advance time as far as it is safe and therefore need to know the distance between objects that are likely to collide in the future. To estimate the relative cost of this kind of approach we need to compare the costs of distance measurements with the cost of intersection testing and penetration depth computation. The second reason for studying the cost of distance computation is that we want to examine the swept-shape version of GJK (see Section 2.3.4), but because of the random nature of our benchmark it can not fairly be tested directly (the number of test configurations that would end up with intersections in the swept region would simply be too low). Instead we will look at the cost of measuring the distance between two objects so that we can compare it to the cost of using EPA, or the depth computation extension of MPR.

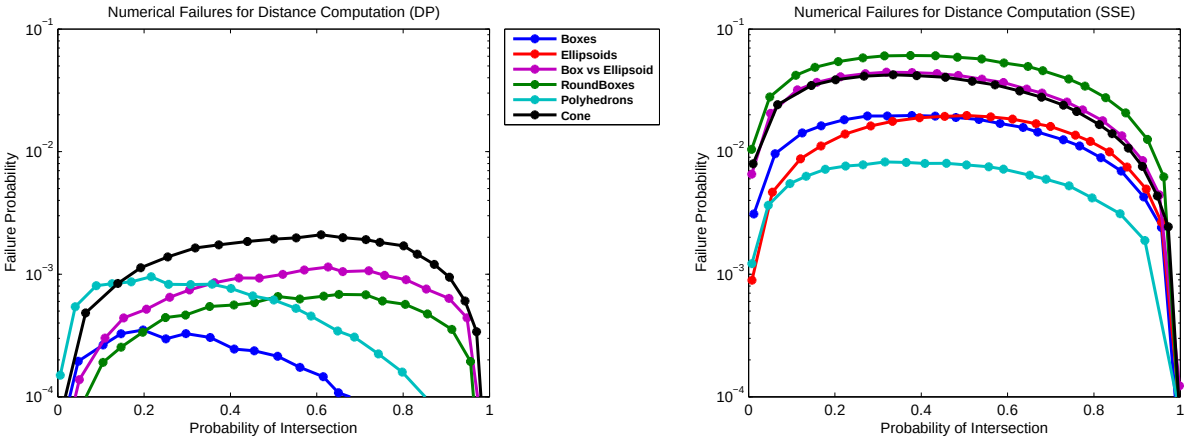

Figure 13: Failure Frequency for Distance Computation

The frequency of numerical failures for distance computation plotted against the probability of intersection. Note the logarithmic scale of the y-axis. The left figure uses double precision floating point vectors, the right one uses SSE based vectors.

Numerical Failures:

Our implementation of GJK is very sensitive to the numerical precision when used for distance computations. Thus before we look at the data for distance computations we present plots of the failure frequency. Figure 13 shows a plot of these frequencies for DP and SSE implementations, note the logarithmic scale of the y-axis. We have not found exactly what causes this huge loss of stability when switching to SSE vectors, but the errors occur somewhere in our Voronoi based method of computing the point closest to the origin in a simplex. We do not know if these errors are specific to out implementation, inherent to the Voronoi based method, or inherent to the general problem. Although we think

the general problem is numerically tricky, the huge problems seen here are specific to our implementation. In any case we judged that tracking down the error would be too time consuming and instead use this test as an example of the risks with using a lower numerical precision. For these reasons we will present test data for both the SSE cases and the DP cases of all tests in this section but in general we will restrict our comments to the DP results.

Average Iteration Count:

An plot of the average iteration count used by GJK can be seen in Figure 14. Notice the clear separation between quadratic surface shapes and polyhedral shapes. This indicates that the complexity of computing the distance between two shapes is proportional to the resolution of the shapes for lack of a better word or the error tolerance for quadrics. We can also note that measuring the distance between objects is significantly harder than determining if they intersect as the curves are decreasing instead of increasing. The reasons for the initial increasing portion of the curves is the relative error tolerance we use as a termination condition; with an absolute error tolerance the plots would be closer to monotonically decreasing. If we ignore the easy Polyhedral cases (that would become significantly harder with more detailed models) the average number of iterations needed to compute the distance between two shapes seems to be roughly 2-18 compared to 2-4 for both SA-GJK based and MPR based intersection testing, 2-32 iteration for depth computation with SA-GJK+EPA and 2-25 iteration for depth computation with MPR's heuristic.

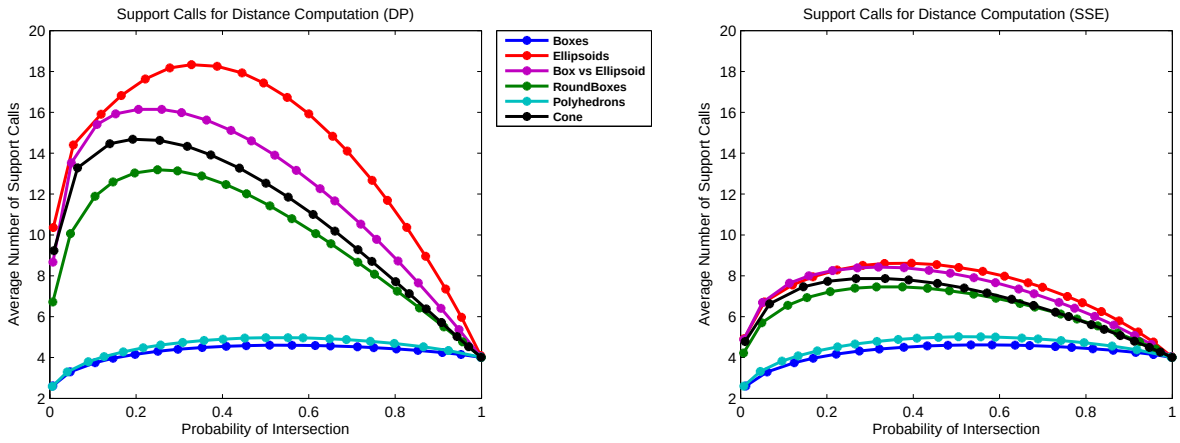

Figure 14: Mean Iterations used by Distance Computation

The average number of support calls needed to determine the shortest distance between objects. The left figure uses double precision floating point vectors, the right one uses SSE based vectors.

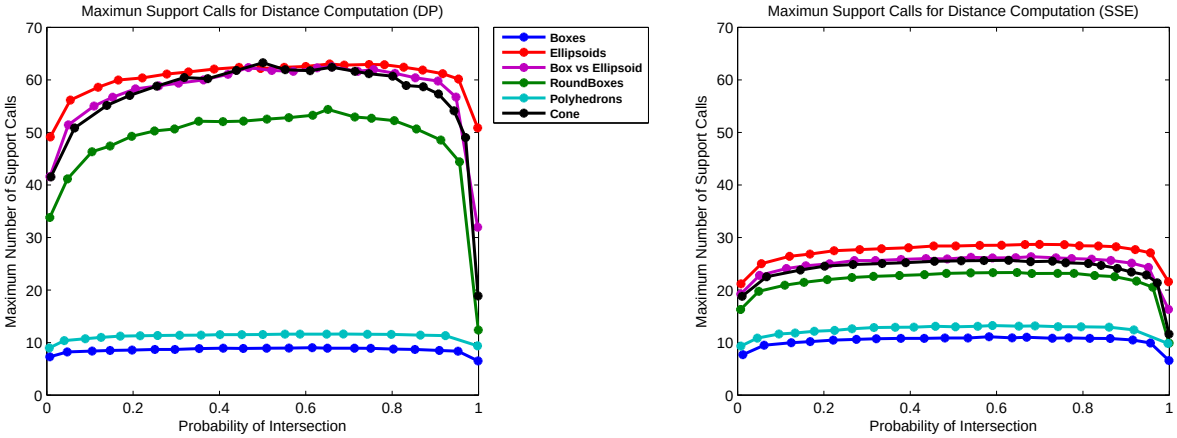

Figure 15: Maximum Iterations used by Distance Computation

The maximum number of support calls needed to determine the shortest distance between objects. The left figure uses double precision floating point vectors, the right one uses SSE based vectors.

Maximum Iteration Count:

The separation between quadric shapes and polyhedral shapes becomes even more noticeable when we study the maximum iterations count instead of the average. As can be seen from Figure 15 configurations involving Ellipsoids or Cones need almost six times as many iterations as those involving Boxes or Polyhedrons. If we compare this to the numbers from intersection testing and penetrations depth computation: We see that the distance computation uses anything from 10-60 iterations. Intersection testing with SA-GJK uses 6-14 iterations and 8-16 with MPR. Penetration depth computation plus intersection testing uses 20-70 with MPR and 15-98 with GJK+EPA. We also note that the internal order of the shapes based on complexity is the same for all three tests. In order of decreasing complexity we have: Ellipsoids, Ellipsoid vs Box, Cones, RoundBoxes, Polyhedrons, Boxes.

Average Run-time:

The run-time for the distance computation, which is shown in Figure 16 indicated that unlike the intersection case test where the support mapping time dominated the computation time completely the average time is affected both by the complexity of the shape and the cost of the support mapping. As an example of this we can look at the three curves for the ellipsoids, polyhedrons and boxes: The boxes and polyhedrons use roughly the same number of iterations but the polyhedron is twice as expensive in time because of its costly support mapping. The support mapping for the polyhedron is more expensive than the ellipsoid one but measuring the distance between two ellipsoids still takes more time on average because of its more complicated shape. If we continue our comparison with intersection testing, penetration depth computation and distance measurement we see that the average

run-time for computing the distance varies from $0.2*10^{-5}$ to $1.2*10^{-5}$ seconds. Intersection testing with MPR takes between $0.16*10^{-5}$ and $0.35*10^{-5}$ seconds, compared to SA-GJK which needs between $0.18*10^{-5}$ and $0.42*10^{-5}$ seconds on average. MPR based depth computation varies between $0.2*10^{-5}$ and $1.5*10^{-5}$ seconds while SA-GJK+EPA based depth computation spans from $0.2*10^{-5}$ to $4.5*10^{-5}$ seconds.

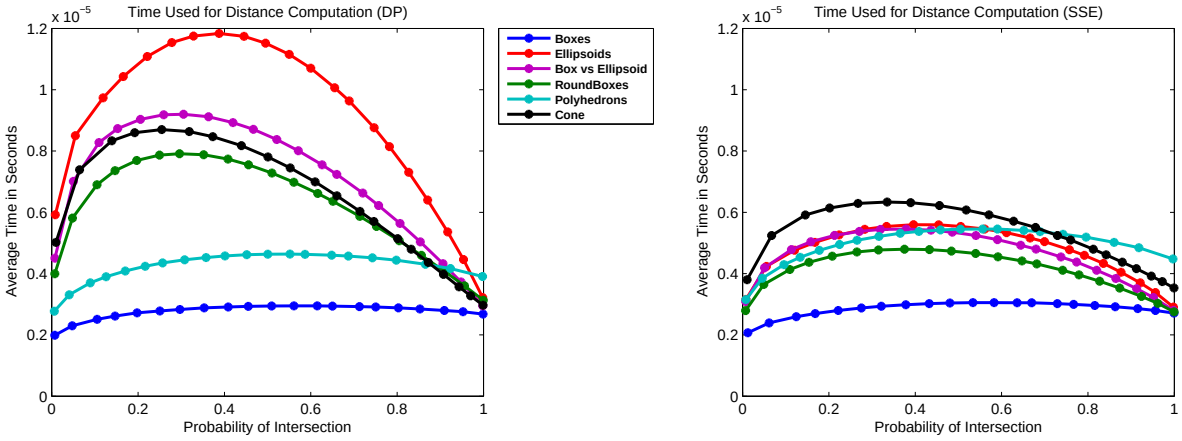

Figure 16: Mean Time used by Distance Computation

Average Run-time needed to determine the shortest distance between two objects. The left figure uses double precision floating point vectors, the right one uses SSE vectors.

4.1.4 Measuring Differing Results

The final randomized test we perform is a comparison between the different algorithms results on an intersection test. The goal of this test is to see if we can measure cases where neither algorithm suffers a catastrophic failure but each gives a different result (and thus one of them is wrong). The frequency of such cases would give us a general impression of the reliability of the algorithms. We will perform this test with and without penetration depth computation following the basic intersection test. Normally penetration depth computation would not change the result of the intersection testing, but the reference implementation is constructed in such a way that if the penetration depth algorithm detects any numerical problems it will invalidate the intersection test (since we prefer false negatives to false positives). The penetration depth addition will let us estimate if the two algorithms find the same configurations troublesome. It will also let us compare the frequency of such failures to the frequency of catastrophic failures measured in Section 4.1.2.

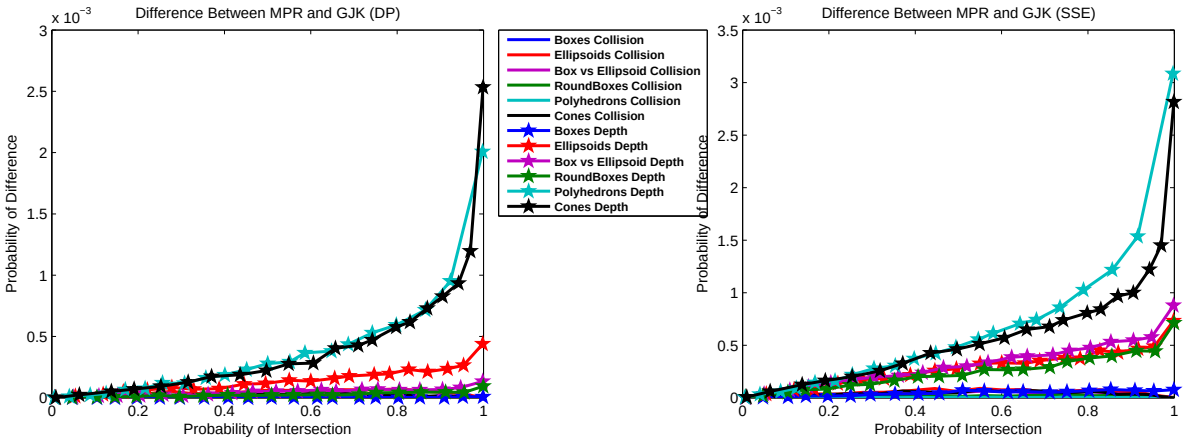

Figure 17: Difference Frequency for MPR Versus GJK

An estimation of how likely it is that different algorithms disagree. Curves marked 'collision' indicate a disagreement between SA-GJK and MPR. Curves marked 'depth' indicate a disagreement between SA-GJK+EPA and MPR+depth finding extension. The left figure was constructed using double precision vectors, the right one with SSE vectors.

The estimated probability that they will disagree given the probability of an intersection can be seen in Figure 17. Looking at the figures it seems that intersection testing suffers from no differing results at all. However this is not quite true: by examining the base data that we plot it was noted that when performing intersection testing MPR and SA-GJK disagree roughly 5-10 times in a million, but since we only tested a million configurations this should be seen as a very rough estimate. In either case they do differ but very rarely compared to the frequency of numerical failures in the penetration depth algorithm. The choice of floating point model only seems to affect two of the tested shape combinations, namely RoundBoxes and Ellipsoid versus Box. In general the frequency of disagreements

is of the same magnitude as the numerical failure frequency measured for EPA in Section 4.1.2. This observation tells us that in general MPR and EPA do not have trouble with the same configurations. The notable exception to the above is that the high frequency of failures for Polyhedrons when using the SP floating point model that was seen in Figure 11 is not present in Figure 17. This seems to indicate that both MPR and EPA have trouble measuring the penetration depth of intersecting Polyhedrons when limited to SP floating point numbers.

4.2 Rigid Body Simulation Tests

The first thing we want to test is the effect that the choice of collision algorithms has on the stability of a rigid multibody simulation. To be able to do this we need a constraint solver that is capable of enforcing collision constraints. Unfortunately there is no simple and elegant or even correct way to formulate a collision law for multiple simultaneous collisions [10]. Since the implementation of a collision constraint solver is not the focus of this thesis several compromises where made in the construction of our constraint solver. In its basic shape our solver is similar to the impulse based iterative solver described by Bender [11], but there are several differences. The most important of these is that the reference simulator treats both collisions and contacts as collisions. A consequence of treating contacts as collisions is that the simulator cannot correctly stop objects from sliding down a slope as explained by Mirtich and Canny [12]. Another simplification in the reference solver is that the impulse iteration loop does not prevent interpenetration. Instead interpenetrations are handled by first constructing a contact graph similar to the one used by Guendelman et al. [13]. Once the contact graph has been constructed the interpenetrating objects are projected apart, starting at the root level of the contact graph and working upwards; the projection is done in such a way that only the destination node (object) is moved. To prevent the objects from shaking, the projection only moves the objects a fraction of the total interpenetration depth each time-step. This interpenetration followed by projection algorithm is of course not physically correct, but by combining it with a dampening factor for the whole system it gives visually acceptable results. Since the objective is to test the effect the choice of collision algorithms has on the simulation stability and not to correctly simulate the physics involved, these limitations are not a primary concern as long as we avoid tests that measure factors directly affected by the limitations.

This gives us the following steps for advancing the simulation time by a time-step Δt:

1. Integrate object positions forward in time by the time-step.
2. Check for collisions, and compute contact points.
3. Iteratively apply impulses until the relative motion at each contact point projected on the contact normal is negative. Or until a fixed maximum number of iterations have been used. Whichever comes first.
4. Sort collisions into a contact graph and project interpenetrations apart based on the contact graph.
5. Go to step 1.

To make the simulation deterministic we use a fixed time-step for the actual time-integration, and only display the results once the simulation is up to date with wall clock time. This lets us have a deterministic simulation where perceived velocities are not dependent on the hardware resources available. But this approach has the downside that if we add so many

objects that computing a time-step takes more than Δt time, the whole simulation will grind to a halt.

4.2.1 Stacking Boxes

To test how the choice of collision algorithms affects the simulation's stability we tried to stack as many boxes as possible using various simulation configurations and collisions algorithms. A stack of objects was judged stable if it had not collapsed after 5 minutes. The dimensions of boxes used were $4 \times 2 \times 4$ units. The stacks were created such that the boxes to begin with were face to face in y direction but randomly perturbed by at most ± 0.2 units in the x and z axes. We used three different collisions detection schemes in the test: the Swept GJK with EPA as backup algorithm described in Section 2.3.4, a combination of SA-GJK and EPA, and MPR with the depth-finding extension from Section 2.4.1. The quality of the constraint solver was varied by letting it use a maximum of 20, 40 or 60 impulse iterations. Each algorithm and constraint solver combination was run using both double precision (DP) floating point vectors and SSE vectors. For each test we measured two things: How large a stable stack of boxes could we maintain, and what was the average time spent in computing the collision data for one collision (in other words the average time spent in collision routines divided by the number of boxes for a frame). The timing values are rough estimates and should be seen as guidelines to the relative cost of the different schemes rather than exact timings. The results of the test can be seen in Table 2. As can be seen from the results, the constraint solver quality is significantly more important than the choice of collision algorithm when it comes to stacking. The choice of collision algorithm is roughly as important as the floating point model used. If we look at the timings for this test, compared to the depth computation times for box vs box tests from the randomized benchmark: We see that GJK+EPA needed around 10^{-5} seconds, and MPR roughly $0.5 * 10^{-5}$ seconds to compute the penetration depth. Therefore the contact information computation needs less than $0.8 * 10^{-5}$ seconds for two boxes. The above calculation assumes that the simpler case where we only calculate the penetration depth without doing any simulation will be at least as efficient as the corresponding part of the full simulation.

4.2.2 Stacking Round-Boxes

In addition to stacking boxes, we also tried stacking RoundBoxes. The test was constructed in the same way as for the normal boxes. The results from this test can be seen in Table 4.2.2. The constraint solver is still the dominating factor, but interestingly we managed to construct taller stacks with rounded boxes than with normal ones. This seems to indicate that although the algorithms we test behave in a similar way to each other their behavior differs noticeably from shape to shape. From a correctness point of view quadratic shapes seem to be easier to handle than polygonal, we find this slightly surprising. We assume that it is the contact manifold generation that handles quadratic shapes better. Since the error probability for boxes was lower than the error probability for RoundBoxes in the

randomized benchmark this seems to be reasonable. However it was not possible for us to verify this as we do not have an alternative method for manifold generation. Finally we note that the GJK based methods manage to stack at least one box more than the MPR based method in eight cases of twelve, whereas for normal boxes this only occurred in three cases of twelve, which indicates that MPR performs better with polyhedral shapes. The average run-time for computing the penetration depth of intersecting RoundBoxes from the randomized benchmark was: At most $2 * 10^{-5}$ seconds for GJK+EPA. While for MPR it was at most 10^{-5} seconds. By subtracting these times from the total simulation time we see that the manifold construction uses at most $1.7 * 10^{-5}$ seconds for RoundBoxes, which is roughly twice the time for normal boxes. This increase is proportional to the one for computing the penetration depth, assuming once again that the shared parts perform no worse in the randomized benchmark than they do in the full simulation.

Algorithm(impulse iterations)	DP stack	SSE stack	DP time	SSE time
Swept GJK+EPA (60)	17	16	$1.6 * 10^{-5}$	$1.4 * 10^{-5}$
SA-GJK+EPA (60)	17	17	$1.7 * 10^{-5}$	$1.4 * 10^{-5}$
MPR (60)	17	17	$1.3 * 10^{-5}$	$1.2 * 10^{-5}$
Swept GJK+EPA (40)	14	14	$1.7 * 10^{-5}$	$1.4 * 10^{-5}$
SA-GJK+EPA (40)	14	14	$2.0 * 10^{-5}$	$1.4 * 10^{-5}$
MPR (40)	14	13	$1.3 * 10^{-5}$	$1.3 * 10^{-5}$
Swept GJK+EPA (20)	10	9	$1.9 * 10^{-5}$	$1.4 * 10^{-5}$
SA-GJK+EPA (20)	11	10	$1.6 * 10^{-5}$	$1.4 * 10^{-5}$
MPR (20)	10	10	$1.3 * 10^{-5}$	$1.2 * 10^{-5}$

Table 2: Stacking Boxes

Tallest stable stack that could be constructed with different levels of numerical precision and number of impulse iterations. For a stack to be judged stable it had to stand for more than 5 minutes. All tests were run with a update frequency of 200Hz.

Algorithm(impulse iterations)	DP stack	SSE stack	DP time	SSE time
Swept GJK+EPA(60)	19	17	$3.2 * 10^{-5}$	$2.5 * 10^{-5}$
SA-GJK+EPA(60)	19	19	$3.2 * 10^{-5}$	$2.4 * 10^{-5}$
MPR(60)	18	18	$2.7 * 10^{-5}$	$2.3 * 10^{-5}$
Swept GJK+EPA(40)	16	15	$3.2 * 10^{-5}$	$2.6 * 10^{-5}$
SA-GJK+EPA(40)	16	16	$3.5 * 10^{-5}$	$2.5 * 10^{-5}$
MPR(40)	15	15	$2.7 * 10^{-5}$	$2.3 * 10^{-5}$
Swept GJK+EPA(20)	11	10	$3.0 * 10^{-5}$	$2.5 * 10^{-5}$
SA-GJK+EPA(20)	11	10	$3.0 * 10^{-5}$	$2.5 * 10^{-5}$
MPR(20)	10	10	$2.6 * 10^{-5}$	$2.3 * 10^{-5}$

Table 3: Stacking Round-Boxes

Tallest stable stack that could be constructed with different levels of numerical precision and number of impulse iterations. For a stack to be judged stable it had to stand for more than 5 minutes. All tests were run with a update frequency of 200Hz.

4.2.3 Visual Observations

In addition to the more objective tests we also subjectively studied how the simulation behaved visually, which can be just as important as the "correctness" if the objective of the simulation is to display visually plausible results to a viewer. None of the algorithms had any noticeable artifacts from intersection testing, which is to say that even though all the algorithms probably miss detecting collisions they always caught them in subsequent time steps before the results became noticeable. We could also not notice any false positives in the intersection testing so it seems that if the goal of a simulation is visual acceptability both GJK and MPR perform well and the choice should probably be made based on efficiency instead of correctness.

When it came to collision normal and depth computations the differences were more noticeable. In the average case the GJK+EPA based algorithms gave slightly better results, but when EPA failed, it failed a lot worse than the MPR based heuristic. A rough description of how the two algorithms fail would be that MPR tends to guess wrong in the direction that stacks are already tipping, which makes stacks sway back and forth in an almost walking fashion until the stack collapses. MPR does not seem to have any problems with bouncing or sliding objects, nor does it give many noticeable artifacts for less organized piles of objects. EPA on the other hand gives more stable stacks as long as the constraint solver manages to do its job well. However given bad input EPA can generate normals that are very far from the correct normal, which makes objects jump or slide in noticeably "wrong" directions. Another error that was caused by EPA but not by MPR occurred when sliding a cube down a wedge shaped slope. When the cube was close to the top of the slope, EPA sometimes gave an incorrect normal that made the cube slide up the slope instead of down. The incorrect normal is caused by numerical errors that only occur when using SP floating point numbers. The incorrect normal causes the projection code to project the box up the slope, so this behavior would not occur with a better constraint solver, but the underlying problem would. All of EPA's problems are numerical in nature so they should be possible to eliminate or at least alleviate given enough time and work. But with equal or just slightly more implementation time for GJK and EPA than for MPR, the GJK and EPA combination suffers from fewer but worse artifacts, whereas MPR has more small errors but less big ones. It should be noted that most of MPR's failures are probably caused by the method's heuristic nature and not by numerical errors so they cannot be mitigated by additional implementation time in the same way that EPA's problems can.

By rendering markers for the generated contact points we were able to visually study the behavior of the manifold construction method. In the general the manifold construction works satisfactory, however it seems to perform better for slightly rounded objects than for purely polygonal objects, as was shown in Sections 4.2.1 and 4.2.2. There were however two general cases that it performed noticeably worse for, the most problematic of the two was cones, cylinders or similar objects that combine quadratic surfaces with sharp discontinuities. For such shapes the manifold construction behaved so badly that we had to construct a special case that only generates one contact point for each collision. This solution is probably

unnecessarily drastic, but since our primary goal is to evaluate the method's shortcomings not to circumvent them, we found this solution satisfactory for now. The second case that can generate weird contact points is unorganized piles of quadratic shapes, in which case the contact points flicker a lot and at least part of the time seem to be very far from where they should. We have not been able to determine if this is caused by the normal computation or the contact point generation but for stable contact manifold generation it would have to be solved.

5 Conclusion and Future Work

5.1 Intersection Testing

If we are only interested in detecting collisions, both SA-GJK and MPR do the job well. Neither algorithm suffers from noticeable numerical problems. The only difference between them is the computational speed, a contest that MPR definitely wins in our implementation. With a better distance computation subroutine for GJK the difference could be evened out somewhat but each iteration will always be more expensive for GJK than MPR. The reason for this is that GJK in each iteration finds the optimal search direction for the next iteration, this computation is significantly more expensive than just figuring out which triangle of three possibilities a ray passes through. In the collisions detection case the extra work to find the optimum does not pay off. The only case we can envision where SA-GJK might actually be preferable would be if we had shapes with extremely high cost support mappings, as the maximum iteration count is lower for GJK than for MPR (though the average is more or less the same).

5.2 Depth Computation

When we are interested in computing the penetration depth as well as detecting intersections the question of which algorithm is the most suitable becomes less clear cut. If speed and visual fidelity is the main focus of the simulation, our tests indicate that MPR followed by the extension for heuristically determining the penetration depth is the best solution. If correctness is the main focus of the simulation, GJK combined with EPA seems to be the strongest candidate. The main downside of using GJK combined with EPA is that they probably need double precision floating point numbers to achieve the higher level of correctness that they potentially can deliver. In most cases DP excludes using SIMD solutions to speed up the computation so the speed penalty could be significant. If shallow intersections are dominant in the simulation one might consider the swept shape modification to GJK as a initial depth estimate. However from our tests it seems that this scheme is slower than MPR and both are heuristic methods so we would probably recommend using the MPR based schemes over swept GJK with EPA fallback for most cases. The final possibility is a preemptive method based on measuring the distances between objects with GJK and advancing time exactly to the moment of the next intersection. We can not say very much about this scheme as we have not implemented it, but from our tests of GJK it seems that this method would also need double precision floats to function well, but it is potentially the most correct way to handle the problem.

5.3 Manifold Generation

Our experiments with generating a contact manifold by perturbed sampling of the support mapping seem to work for most cases. The method has problems with cones, cylinders

and similar shapes that combine quadratic surfaces with flat surfaces, specifically if they meet at sharp angles. The reference implementation is also quite slow. Our conclusion is that the basic concept of perturbed sampling has merit, but that our method and/or its implementation will need to be reformulated to give acceptable performance and data.

5.4 Future Work

In this thesis we have mainly studied simple mathematical shapes for which we could construct constant time support mappings. In practice, models are most often represented as irregular polyhedral data. The support mappings for this type of objects are generally handled by hill-climbing algorithms combined with caching the last returned point. Unfortunately hill-climbing cannot guarantee a better time complexity than linear. Combined with caching of the last returned point it results in close to constant look-up time as long as we have good temporal coherence. Unfortunately all simulations do not have good temporal coherence. Even in multibody physics simulations that normally have very strong temporal coherence, multiple simultaneous collisions can invalidate the caching completely. Therefore to efficiently handle polyhedral data in the general case, we would need a hierarchical data structure that could give us logarithmic time support mappings for general polytopes. As mentioned earlier something similar to Dobkin-Kirkpatrick hierarchies should be able to handle this demand. However algorithms for constructing the data structure from a point-list, in addition to an algorithm for sampling the support mapping in an efficient manner, would need to be developed.

All the algorithms we have presented in this thesis only work for convex geometries. As we have said earlier any nonconvex geometry can be approximated to within an arbitrary precision as the union of many convex shapes. While this is true it does not give us a data structure for representing such complex objects, nor does it give us efficient algorithms for checking intersections between two such objects. These problems have of course been solved before, but we are not aware of any survey that compares different methods. An even more interesting problem is that given such a data structure for complex objects, and an arbitrary polyhedral model represented as for instance a list of triangles: Can we construct an efficient method for decomposing the nonconvex object described by the triangle list into a close to minimal number of convex shapes? We believe that constructing an algorithm for finding the optimal decomposition is probably a waste of time as the problem is extremely difficult (probably NP-hard), but it should be possible to construct a heuristic method for solving the problem. Relevant questions are: How should such a heuristic be constructed? How favorable results can be achieved in general? And finally, how does the relation solution-quality versus time-consumption for different such heuristics look?

6 Acknowledgments

I would like to thank Martin Ericsson and Henrik Meijer for their invaluable assistance in finding errors in both the code and this rapport. I would also like to thank Vicki Olvång for her help with proofreading the text for grammatical errors. Finally I would like to thank my supervisor Bo Nordin for his patience in not giving up on me even though the progress on this thesis was at times extremely slow.

References

[1] E. Gilbert, D. Johnson, and S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Journal of Robotics and Automation*, vol. 4, no. 2, pp. 193–203, 1988.

[2] G. van den Bergen, *Collisions Detection in Interactive 3D Environments*, D. Eberly, Ed. Morgan Kaufmann Publishers, 2003.

[3] G. Snethen, "Xenocollide: Complex collision made simple," in *Game Programming Gems 7*. Course Technology, 2008.

[4] E. Weisstein. Minkowski sum. MathWorld–A Wolfram Web Resource. Retrived 2010-01-20. [Online]. Available: http://mathworld.wolfram.com/MinkowskiSum.html

[5] D. P. Dobkin and D. G. Kirkpatrick, "A linear algorithm for determining the separation of convex polyhedra," *Journal of Algorithms*, vol. 6, no. 3, pp. 381–392, September 1985.

[6] E. Weisstein. Point-line distance–3-dimensional. MathWorld–A Wolfram Web Resource. Retrived 2010-01-20. [Online]. Available: http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html

[7] ——. Scalar triple product. MathWorld–A Wolfram Web Resource. Retrived 2010-01-20. [Online]. Available: http://mathworld.wolfram.com/ScalarTripleProduct.html

[8] ——. Geometric centroid. MathWorld–A Wolfram Web Resource. Retrived 2010-01-20. [Online]. Available: http://mathworld.wolfram.com/GeometricCentroid.html

[9] P. Bourke. The shortest line between two lines in 3d. Retrived 2010-01-20. [Online]. Available: http://local.wasp.uwa.edu.au/~pbourke/geometry/lineline3d/index.html

[10] A. Chatterjee and A. Ruina, "A new algebraic rigid body collision law based on impulse space considerations," *Journal of Applied Mechanics*, vol. 65, no. 4, pp. 939–951, 1998.

[11] J. Bender and A. Schmitt, "Constraint-based collision and contact handling using impulses," in *In Proceedings of the 19th international conference on computer animation and social agents*, Geneva (Switzerland), July 2006, pp. 3–11.

[12] B. Mirtich and J. Canny, "Impulse-based simulation of rigid bodies," in *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*. New York, NY, USA: ACM, 1995, pp. 181–ff.

[13] E. Guendelman, R. Bridson, and R. Fedkiw, "Nonconvex rigid bodies with stacking," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*. New York, NY, USA: ACM, 2003, pp. 871–878.