Collision Detection for Haptic Rendering

Hans Sjöberg, Olov Ylinenpää

April 23, 2009
Master's Thesis in Computing Science, 2*30 ECTS credits
Supervisor at CS-UmU: Anders Backman
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Collision detection tries to answer one or more of the following questions: *if?*, *where?* and *when?* two or more (virtual) objects are colliding. These types of questions arise in many different applications including robot motion planning, physics-based simulations, haptic rendering, virtual prototyping, interactive walkthroughs, computer gaming, and molecular modeling. Haptic rendering is the display of force to a user via a mechanical device, giving the feeling of touching and interacting with virtual objects. In this thesis we present a design and implementation of a general-purpose, object-oriented, extendable collision detection framework and a library for haptic rendering.

Contents

List of Figures

Chapter 1

Introduction

> hap·tic ['haptic]
>
> adjective technical
>
> of or relating to the sense of touch, in particular relating to the perception and manipulation of objects using the senses of touch and proprioception.
>
> ORIGIN late 19th cent.: from Greek *haptikos 'able to touch or grasp,'* from *haptein 'fasten.'*
>
> *New Oxford American Dictionary, 2nd Edition*

Haptic rendering is where one uses a force-feedback device (haptic device) consisting of motors and angular sensors to generate forces that a user can sense. This enables us to touch and feel virtual objects rendered by a computer. Haptic rendering is used in many different areas such as modeling, mechanical engineering and visualization but mainly in the medical area, where one wants to touch volumetric data from tomography or practice difficult surgical tasks. One can look at haptic rendering as consisting of two parts; the collision detection part that tries to answer *if*, *where* or *when* two or more of the haptic objects collide and the haptic rendering algorithm that calculates the force to apply on the user through the haptic device. Collision detection is a fundamental part of haptic rendering but is also used in many other areas such as computer animation, physical based modeling, molecular modeling, computer simulated environments (VR) and robot motion planning. [LM03, Eri05]

1.1 Background

VRlab is a center active in research and development in the areas of visualization, visual interactive simulation and virtual reality (VR). Connected to VRlab is a core of specialists, as well as a large network of affiliated researchers, teachers, students and collaborating companies.

osgHaptics is a high-level library developed at VRlab that is used to model haptic environments integrated in a rendering scenegraph, *OpenSceneGraph*.[1] Today, the commercial library *OpenHaptics* from Sensable Inc.[2] is used internally in *osgHaptics*. In

[1] `http://www.openscenegraph.org`
[2] `http://www.sensable.com`

spite of its name OpenHaptics is proprietary software and there exists a desire to replace it with a free[3] alternative that can be altered and expanded to fit the various needs of VRlab.

1.2 Thesis Description

1.2.1 Purpose

The purpose of this thesis is to investigate current methods for stable and efficient collision detection between different geometries for use in haptic force rendering. Different algorithms/methods for haptic rendering shall also be examined. A design and implementation shall be performed.

1.2.2 Goal

As can be seen in Figure 1.1, the goal is to design and implement a replacement for OpenHaptics consisting of two libraries, one for haptic rendering and one for collision detection. The collision detection library should be designed in such way that it easily can be extended with additional primitives and geometry descriptions for general collision detection queries. The library should be designed so that it can be used in different applications such as physics simulations, haptics, games etc. The haptic library will be built on top of *Haptik*, which is a low-level open source library that is used to communicate with the haptic device. This way the library will be independent of the haptic hardware and drivers. The functionality of the libraries should be demonstrated in an application.

The integration of the new libraries and osgHaptics is not a part of this thesis.

1.2.3 Method

In order to achieve the aforementioned goals it is necessary to perform an in-depth study in the fields of collision detection and haptic rendering. It is important to have an overview of different methods, algorithms and applications since the collision library should be designed in such way that it is easy to add future extensions. This together with the implementation is a major part of the thesis. Since a collision detection library cannot be tested by itself a number of test applications have to be written in order to verify both the correctness and usability of the library. The correctness will be determined visually, i.e. if it looks correct it is correct.

1.2.4 Report Outline

Chapter 2 gives an introduction to collision detection. A number of concepts, methods and strategies are presented. Chapter 3 describes methods and algorithms in the field of haptic rendering. Chapters 4 and 5 describes the design and implementation of the libraries for collision detection and haptic rendering. Chapter 6 discusses the result, shortcomings and future work.

[3] In the sense "free speech" rather than "free beer"

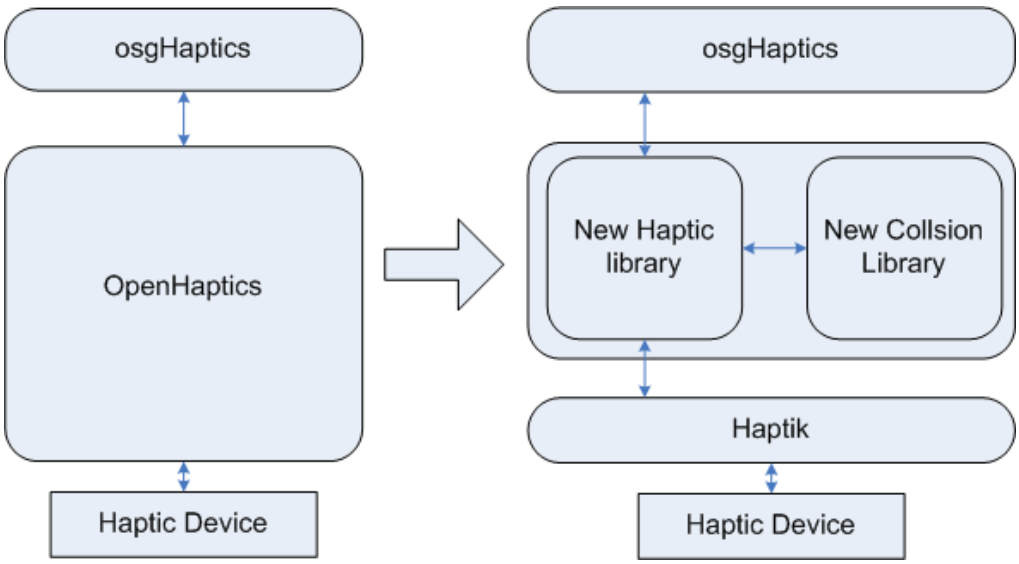

Figure 1.1: To the left, the current situation for osgHaptics is depicted. The desired configuration is shown on the right.

1.2.5 Prerequisites and Terminology

The reader is supposed to be familiar with basic data structures and algorithms like trees, linked lists and various sorting algorithms. Basic linear algebra and computer graphics knowledge is also expected. Here follows a short list of definitions that hopefully will help the reader

- Haptic rendering - Analogous to graphical rendering with the difference that virtual objects are made perceivable through touch rather than vision.
- Haptic device - A device that provides a touch sensations of interacting with virtual objects.
- Collision detection - Answer one or more of the following questions *if?*, *where?* and *when?* two or more virtual objects are colliding/overlapping.
- Temporal coherence - The existence of a correlation in time between the position of two objects.
- Polygon Soup - A way of storing polygon models, each polygon is stored separatly in a list. In contrast to polygon meshes the data structure contains no information about the topology of the model. A polygon soup of triangles is called a Triangle Soup.
- Particle system - Basically just a collection of 3D points in space. Depending on what the particle system is simulating each particle often has properties like velocity, direction, age etc.
- Convex object - A polygon where all internal angles are strictly less than 180 degrees.

1.3 Related Work

There exist numerous collision detection libraries which are more or less application specific. Some are specialized to handle only a specific type of collision query. Examples of this are RAPID[4], SOLID[5] and OPCODE[6] which only handle collisions between polygon soups. Others like Bullet[7] and OpenTissue are more of general purpose collision detection libraries. In the field of collsion detection for haptic rendering there exists a library called H-Collide.

[4] http://www.cs.unc.edu/ geom/I_COLLIDE/index.html
[5] http://www.win.tue.nl/ gino/solid/
[6] http://www.codercorner.com/Opcode.htm
[7] http://www.continuousphysics.com/Bullet/

Chapter 2

Collision Detection

Collision detection tries to answer one or more of the following questions *if?*, *where?* and *when?* two or more objects are colliding. These types of questions arise in many different applications including robot motion planning, physics-based simulations, haptic rendering, virtual prototyping, interactive walkthroughs, computer gaming, and molecular modeling. [LM03]

The most basic case is a test between two objects. *Intersection testing* is the *if*. Are the object intersecting or not at a given time? *Intersection finding* corresponds to the *where*, some applications like rigid body simulations require knowledge of a set of points where the objects are intersecting, *contact points*, a much harder query to calculate. If the objects are penetrating we might be interested in knowing the *penetration depth*. This is often defined as the minimum translational distance, i.e the length of the shortest movement vector needed to separate the objects. [Eri05]

Since having penetrating objects is not a desirable state in most cases, some applications use a different approach and try to prevent penetration before it occurs. Keeping track of the closest points of two non-intersecting objects the *time of impact* or TOI can be calculated, that is the *when*.

In general, a specialized routine must be written for every combination of geometries for each query type (e.g. box-cylinder intersection finding). This means that writing a complete collision library is an enormous task.

In the basic case we had two objects to test for collision, in a scene we might have thousands of objects moving around. At first collision detection appears to be an $O(n^2)$ problem since any two objects in a scene can collide and thus every object needs to be tested against all other. By dividing collision detection into two parts, the *broad phase* and *narrow phase* we can deal with this. The concept was first introduced by Hubbard[Hub96]. The idea is to first (in the *broad phase*) make a coarse test on simplified bodies which prunes away unnecessary pair tests. The *narrow phase* is the exact tests described earlier between object pairs whose simplified form overlapped in the *broad phase*. In the rest of this chapter we will present a number of different strategies and algorithms for both broad an narrow phase collision detection.

2.1 Separating Axis Theorem

If we can find a plane that separates a pair of objects we know for a fact that they are not colliding. This is known as a *separating plane*. In order to know that a pair *is*

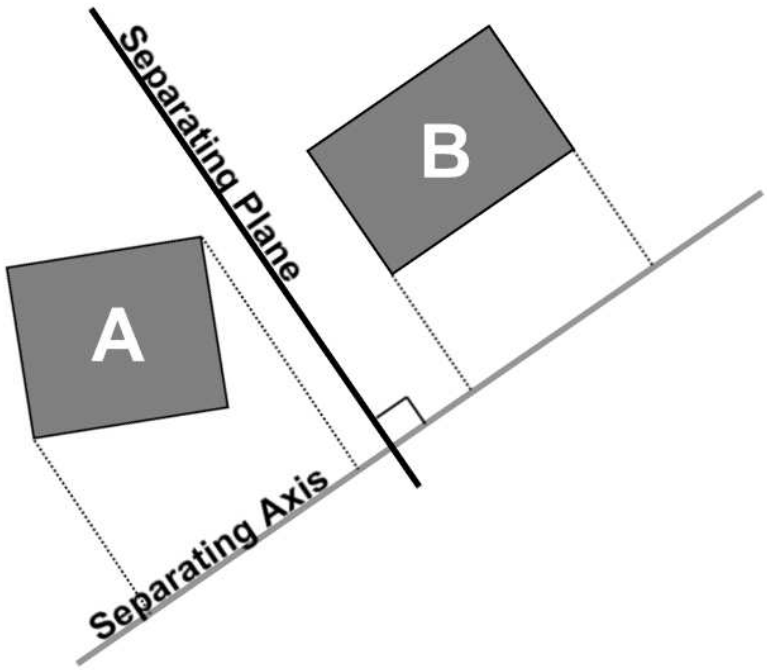

Figure 2.1: Example of the separating axis theorem in 2-D

colliding we have to check that the objects are not separated by any plane. The problem is that there exists an infinite number of possible separating planes. Fortunately there exists a theroem for convex objects that gives a finite set of planes that are sufficient to test in order to verify a collision. This theorem, The Separating Axis Theorem, also known as SAT is the basis for many separating tests. In [Ebe04] Eberle makes this formal definition.

Definition 2.1.1. Separating Axis Theorem
For any two arbitrary convex, disjoint, polyhedra A and B, there exists a separating axis where the projections of the polyhedra, which form intervals on the axis, are also disjoint. If A and B are disjoint, they can be separated by an axis that is orthogonal to the plane formed by one of the following:

1. A face normal of polyhedron A.
2. A face normal of polyhedron B.
3. A normal formed by the cross product of a pair of edges, one from A and the other from B.

This makes it easy to derive new intersection tests. Given two triangles T_1 and T_2, SAT gives a maximum of eleven axes that need to be tested for separation. Two from the normals of the triangles and nine (3×3) from the cross products of their edges. If the projections of T_1 and T_2 are separated on any of these axes, the triangles are not

intersecting. Hence it is possible to do an early exit from the intersection test as soon as one separating axis is found. For an illustration of the theorem see Figure 2.1

2.2 Bounding Volumes

Testing two complex gemetrical shapes against each other can be very expensive. In order to avoid doing these expensive tests, complex geometrical shapes can be wrapped in simpler geometrical shapes, bounding volumes, such that when testing two objects for intersection a much cheaper test with their bounding volumes can first be performed. Only if the bounding volumes overlap an exact test has to be performed. Examples of bounding volumes are Spheres, Axis-Aligned Bounding Boxes (AABBs), Oriented Bounding Boxes (OBBs) k-Discrete Oriented Polytopes (k-DOPs). Zachmann [ZL03] identifies three important properties of bounding volumes:

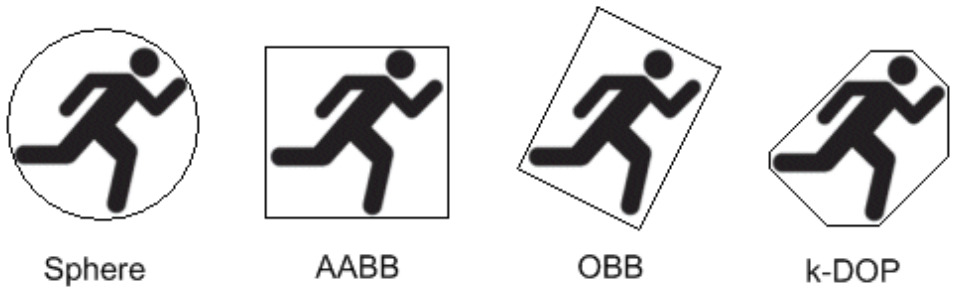

Figure 2.2: Examples of bounding volumes

1. Tightness. A tighter bounding volume will lower the number of expensive tests that needs to be performed.

2. Memory usage. The bounding volumes are stored together with the objects and should be as small as possible.

3. Number of operations needed to test against another BV. Bounding volumes are used to avoid expensive pair-wise testing so their intersection tests should be cheap.

As always there is a trade-off between these properties. In general, complex bounding volumes are more tight fitting but also use more memory and have more expensive overlap tests and vice versa. No specific bounding volume is the best choice for all situations. Figure 2.2 shows four different types of bounding volumes. They are ordered from left to right where the leftmost has the fastest test and uses the least amount of memory but has the worst fit. The rightmost bounding volume has the best fit but on the other hand has the slowest test and uses the most memory.[1]

[1]Note that this is dependent on the size of k for the k-DOP e.g. a 6-DOP corresponds to an AABB or OBB depending on the direction of the normals, but for large k the order is correct.

2.2.1 Axis-Aligned Bounding Boxes (AABB)

An Axis-Aligned Bounding Box is a rectangular box with six sides aligned with the given coordinate system. Since all AABBs in one coordinate system have the same orientation, the overlap test is simply a comparison of coordinate values along the three axes. If the values don't overlap at any of the axes the boxes are not intersecting. Construction is done by projecting the contained object on each of the three axes and finding the extreme values. AABBs are invariant to translations but not rotations since that would make the sides of the box unaligned to the coordinate axes. [AMH02]

2.2.2 Bounding Spheres

Spheres are defined by their center and radius, this gives the advantage of being invariant to rotation. The test for overlap between two bounding spheres is done by simply checking if the distance between the center points is smaller than the sum of the radii. By comparing the squared distance and radius an expensive square-root operation can be avoided. Spheres are very memory efficient, requiring to store only four floating-point values. They are generally loose fitting, especially for oblong objects. There are a number of ways to create bounding spheres. A fast simple constant time algorithm is to first create a AABB and then use the center of the AABB as center and the diagonal of the box as radius. This could give a poor fit, but can be improved by traversing over all the polygon points and find the point that have the longest distance to the center and picking that as a new radius. Möller [AMH02] and Ericsson [Eri05] present a number of more complex algorithms that generate a better fitting bounding sphere.

2.2.3 Oriented Bounding Boxes (OBB)

An Oriented Bounding Box is a rectangular box with six sides, just as the AABB, but with arbitrary orientation. The most common representation is a center point plus a rotation matrix and three half-edge lengths. When testing for intersection between OBBs the separating axis theorem described in section 2.1 is used. Each box has three unique face orientations and three unique edge directions. This gives a total of 15 potential separating axes to test (six from the faces of the boxes and nine from the combination of edges). [Eri05, GLM96]

2.2.4 k-DOPs

k-DOPs or discrete-orientation polytypes are convex polytypes whose facets are determined by half-spaces whose normals come from a fixed set of k orientations. [KHM+98] The set of normals is shared by all k-DOPs. It is often limited to the set 1,0,-1. A 6-DOP with the normals ($\pm$1,0,0), (0,$\pm$1,0), (0,0,$\pm$1) is a special case of k-DOP that corresponds to an AABB. k-DOPs are constructed in the same way as AABBs; the object is projected along the defined axes and the min and max points are stored. For an 8-DOP only 8 float values needs to be stored. k-DOPs are invariant to translations but rotations would unalign the axes and the k-DOPs must be recalculated. The intersection tests for k-DOPs are similar to the intersection tests for AABBs. All that needs to be done is to check the k/2 intervals for overlap, if the k-DOPs overlap at all of the intervals they intersect. [Eri05]

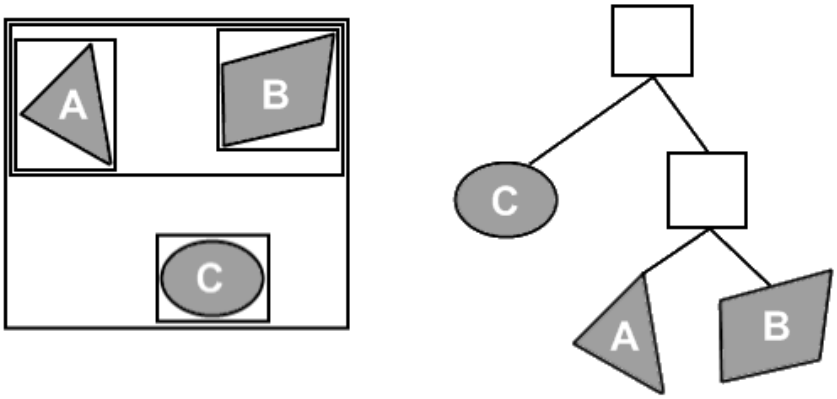

Figure 2.3: A simple bounding volume hierarchy using AABBs as bounding volumes

2.3 Bounding Volume Hierarchies

Intersection tests between complex objects consisting of many primitives can be very costly. Bounding volume hierarchies gives us a way to lower this cost. For an intersection test between two complex objects consisting of N primitives each (two triangle soups with N triangles) we must perform a total of N^2 primitive tests. Using bounding volume hierarchies this pair test can be performed in logarithmic time instead.

A bounding volume hierarchy is a tree structure containing primitives and bounding volumes. In each leaf one or more primitives are stored. Each internal node contains a bounding volume enclosing all primitives in the subtrees of the node. Note that it does not necessarily enclose the bounding volumes of the subtrees. This results in a tree structure with a root node containing a bounding volume that encloses all primitives. An example of a binary bounding volume hierarchy in 2D is shown in fig 2.3. In [ZL03] Zachmann makes the following definition of bounding volume hierarchies.

Definition 2.3.1. Bounding Volume Hierarchy
Let $O = o_1, \cdots, o_n$ be a set of primitives. A bounding volume hierarchy for O, $BVH(O)$, is defined by

1. if $|O| = e$, then $BVH(O) :=$ a leaf node that stores O and a bounding volume of O;

2. if $|O| > e$, then $BVH(O) :=$ a node v with $n(v)$ children $v_1, \cdots, v_n$ where each child v_i is a BV hierarchy $BVH(O_i)$ over a subset $O_i \subset O$, such that the union of all subsets is O. In addition, v stores a bounding volume of O.

where e is the maximum number of primitives in each leaf and $n(v)$ controls the tree's arity, which in this definition need not be constant.

When testing two complex objects with BVHs for intersection, BVH branches with non-intersecting bounding volumes can be pruned. This can reduce the number of primitive tests to $O(\log n)$. The next section explains this in more detail.

2.3.1 Traversing BVHs

Given two bounding volume hierarchies the following algorithm outline can be used to quickly discard primitive pairs which cannot intersect. [Zac02]

Algorithm 1 BVH Traversal algorithm

```
1. traverse(A, B)
2. if A and B do not overlap then
3.       return
4. end if
5.
6. if A and B are leaves then
7.       return intersection of primitives
8.               enclosed by BV(A) and BV(B)
9. else
10.      for all children A[i] and B[j] do
11.              traverse(A[i], B[j])
12.      end for
13. end if
```

In the case where only one of the objects is a BVH, lines 10 and 11 are replaced with

```
10. for all children A[i] do
11.      traverse(A[i], B)
```

Different aspects of this basic algorithm can be modified to improve performance. If two internal nodes overlap how should the algorithm descend into the subtrees? Should the trees be traversed breadth-first, depth-first or some other traversal order? According to Ericson [Eri05] depth first traversal seems to be the most popular choice in collision detection, often enhanced with some simple heuristic. We must also decide on a descent rule that decides in which order the order the trees should be descended, there are a number of possibilities

Descend one before the other. Let A be a rather small object that moves through a bigger object B. If object A is in the middle of object B it will overlap the top bounding volume of object B. Now descending A before B will be very slow since for each leaf in A, B will be traversed. This is even worse than having no hierarchy at all.

Descend the larger volume. At each step the tree with the larger bounding volume is descended into. This rule results in the largest reduction of total volume size in the following tests. Thus, it gives the lowest probability of overlap in subsequent BV overlap tests. [vdB97]

Descend alternatingly. First object A is descended into, then B, then A again. This rule is simple to implement but doesn't prune the search space as good as the previous method.

These are only a few of the possible rules. Others could be: descend based on overlap, descend based on traversal history or some kind of combination of the previous rules. Although the algorithm is formulated recursively, it is often a bad choice to implement

it with recursive calls. Apart from the incurred overhead, the often large number of primitives in the objects results in a very large amount of recursive calls, leading to stack overflow. It is often better to traverse the tree iteratively. [KED05]

2.3.2 BVH Characteristics and Cost Function

A cost function can be used to evaluate the performance of a Bounding Volume Hierarchy (equation 2.1). The function gives a measure of the total cost for an intersection test when at least one of the objects have a BVH. It has its origin in the field of ray tracing, but it is now also used for collision detection. [WHG84] [GLM96]

$$T = N_v * C_v + N_p * C_p \tag{2.1}$$

T: total cost for intersection detection.

Nv: number of bounding volume pairs tested for overlap.

Np: number of primitive pairs tested for intersection.

Cv: cost of testing a pair of bounding volumes for overlap.

Cp: cost of testing a pair of primitives for intersection.

Creating a better hierarchical decomposition of an object, i.e making the bounding volumes fit the object as tight as possible, will result in lower values for N_v and N_p. To lower the values of C_v and C_p a faster overlap test for bounding volumes and primitives must be used. Making the bounding volume test faster usually means having to use a bounding volume with looser fit, resulting in higher value for N_v and N_p. The key to a good BVH is to find a good compromise between the fit of the bounding volumes and the time for the overlap tests.

In [KPLM97] Krishnan presents an altered cost function (equation 2.2), which includes an extra term for updating the BV due to an object's motion. This equation applies when BVs are stored in world coordinates.

$$T = N_v * C_v + N_p * C_p + N_u * C_u \tag{2.2}$$

Nu: number of bounding volumes updated due to object's motion

Cu: cost of updating a BV

Other properties of the BVHs also plays a role in the performance of intersection tests. Here is a list of desirable characteristics for BVHs. [Eri05, KK86]

- The nodes contained in any given subtree should be near each other. "Nearness" is relative, but the lower down the subtree is with respect to the entire hierarchy, the "nearer" objects should be.
- Each node in the hierarchy should be of minimal volume.
- The sum of all bounding volumes should be minimal.
- Greater attention should be paid to nodes near the root of the tree. Pruning a branch here allows a larger subtree to be removed from further consideration, whereas pruning one lower down removes just a few bounding volumes and objects from further consideration.

- The volume of overlap of sibling nodes should be minimal.
- The hierarchy should be balanced with respect to both its node structure and its content. Balancing allows as much of the hierarchy as possible to be pruned whenever a branch is not traversed into.

A choice of tree arity also has to be made. Binary trees are often used. There are analytical arguments suggesting that the use of binary or possibly ternary trees is optimal. [Eri05] Empirical studies also supports the binary choice, but they are not conclusive since few experiments have been conducted with higher-degree trees. [Eri05] On the other hand Mezger[MKE03] claims that quad-trees and oct-trees are significantly faster than binary trees.

In the next section different building strategies for BVHs in which these properties play an important role are presented.

2.3.3 Building Strategies

There are three main categories of methods for constructing BVHs, bottom-up, top-down and insertion.

Bottom-up Construction

When constructing the tree bottom-up, the first step is to enclose all primitives with a bounding volume. This forms the leaf nodes of the tree. Two or more (depending on the arity of the tree) nodes are selected based on some merging criteria. These nodes are then enclosed by a bounding volume to form a new internal node. The procedure is repeated until everything has been grouped under a single node. A possible merging criteria is to select the nodes so that the volume of the new bounding volume is minimized. Another criteria is to select the nodes that are closest to each other. This often produces similar results, but since the "nearest neighbour problem" is well studied it is possible to make faster implementations with this criteria. The bottom-up method often produces better trees compared to the two other methods but it is also the most difficult to implement. [Eri05]

Top-down Construction

Top-down construction is a commonly used method,[GLM96, vdB97] probably because it's easy to implement. Most of the time it doesn't construct the best possible tree though. [Eri05]

The first step is to find a BV that encloses all primitives, this forms the root node of the tree. The next step is to split the set of primitives into two or more subsets. The subsets become the child nodes of the original node, and have their bounding volumes computed. This continues in a recursive fashion until each subset contains less then k primitives, where k is an integer > 0. The main descision a developer faces when creating the tree in top-down order apart from what bounding volumes to use is how to split the input set into new subsets [Eri05]. The most common way is to find some axis along which the primitives should be split, and on this axis find a splitting point. [AMH02] This forms a hyperplane that splits the set into two subsets.

There are an infinite number of splitting-axes and splitting-points to choose from, but some axis and point must be picked. It is theoretically possible to find the best possible axis using an iterative optimization method (e.g. hill climbing). But since this

is expensive the selection problem is often solved by picking the best axis from a number of common choices. [Eri05]

- Local x, y and z coordinate axes.
- Axes from the intended aligned bounding volume. For AABBs this is the same as above.
- Axes of the parent bounding volume.
- The axis through the two most distant points
- The axis along which variance is greatest

The same argument applies for the selection of the splitting point. The best point is selected from a list of common choices. Some examples follow

- Median of the centroid coordinates.
- Mean of the centroid coordinates.
- Median of the bounding volume projection extents.
- Splitting at k evenly spaced points along the bounding volume projection extents and picking the best one.

If the splitting-plane intersects any of the primitives they need to be taken care of. One solution is to split the primitive and assign the parts to different subsets. This makes the child bounding volumes smaller and minimizes the overlap between children. But on the other hand, any primitive that has been split can be split again resulting in a huge increase in the number of primitives. Another strategy is to partition the primitives according to which side of the plane their center point lies [GLM96]. The number of primitives will stay the same, but the bounding volume will be extended by half the width of the primitive, increasing the overlap between the bounding volumes. A similar strategy instead picks the midpoint of the primitive's projection onto the splitting-axis with the purpose of getting a balanced tree, giving good worst case performance. [vdB97]

Insertion Construction

The third method, tree-insertion, starts with an empty tree. The primitives and their BVs are inserted to the tree one at a time. The insertion position is chosen so that the total tree volume is as small as possible (one of the desirable characteristics). Very little is know about the tree-insertion method in the context of collision detection, but since it has been used with success in ray tracing it should work well with collision detection too. [AMH02]

2.3.4 Existing Implementations

Since bounding volume hierarchies have been known for a long time there exists a lot of different implementations. In the following section the more interesting will be shortly described.

Bounding volume hierarchies using spheres have been implemented by Hubbard[Hub95] and Palmer[IJP95].

Gotchalk et al. [GLM96] have implemented BVHs using OBBs, claiming that given a sufficiently large model their OBB-trees are much faster compared to the more common AABB-trees and Sphere-trees.

van der Bergen [vdB97] has been using AABBs. By testing only the first 6 of the 15 axes given by the SAT method, he has managed to get performance that is close to the OBB implementation by Gotchalk et al. When it comes to deformable objects the AABB outperforms the OBB since updating an OBB tree is significantly more complex.

Klosowski [KHM+98] has implemented hierarchies using discrete oriented polytypes (k-DOPs) with good performance in tests with real and simulated data.

2.4 Broad Phase Collision Detection

So far we have only been describing techniques how to perform and improve the performance of intersection tests between two objects, but still every possible pairwise combination of objects are tested against each other. That means

$$\frac{n(n-1)}{2} \approx O(n^2) \tag{2.3}$$

tests, where n is the number of objects. In this section three algorithms that try to improve this by pruning expensive and unnecessary pairwise tests are presented. A common name for these types of algorithms are broad-phase algorithms. [KED05]

2.4.1 Exhaustive Search

The naive way of doing broad-phase collision detection is to calculate a bounding volume for each object and test every pair of bounding volumes against each other. Pseudo code for this algorithm is presented in Algorithm 2. It might seem like a poor choice since it does not lower the number of pairwise tests. However, the algorithm has some nice features that sometimes makes it a good choice.

- Simple and fast to implement.
- Low time constant when using simple bounding volumes (Spheres, AABBs).
- Applicable to any kind of bounding volume.

Algorithm 2 Exhaustive Search

1. for each object $a \in A$ do
2. compute Bounding volume of a
3. end for
4. for each pair (a, b) where $a \neq b$ and $a, b \in A$ do
5. if $BV(a)$ overlap $BV(b)$ then
6. report overlap of (a, b)
7. end if
8. end for

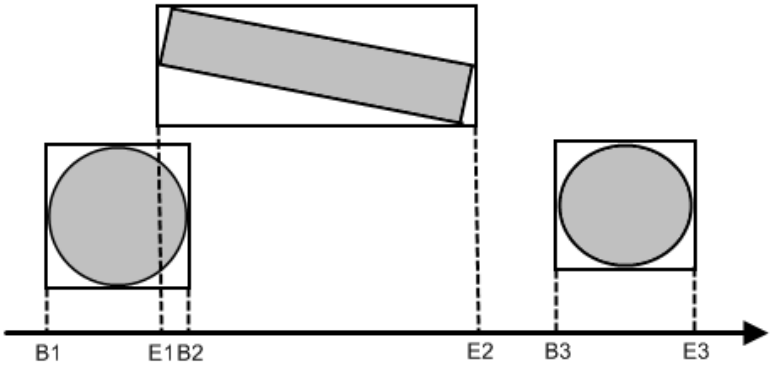

Figure 2.4: AABBs correspond to intervals $[b_i, e_i]$ when projected onto an axis.

2.4.2 Sweep and Prune

If an AABB is projected onto the x, y and z axes it will correspond to an interval [b, e] on each axis, see figure 2.4. This interval plays an important role in the sweep and prune algorithm since a pair of AABBs overlap if and only if their intervals overlap in all three dimensions. The sweep and prune algorithm was first presented by Cohen et al. in [CML+94].

For the sake of simplicity the algorithm will first be described for one dimension. Given a set of objects, the first step is to project their AABBs onto one of the axes, the endpoints of the intervals are then added to a list that is sorted in an increasing order along the axis. In the next step the list is scanned for overlapping intervals (bounding boxes). When a starting endpoint b_i is encountered for some object i, object i is added to a set of active objects. An overlap is also registered against all other objects currently in the set. When the ending endpoint e_i is encountered object i is removed from the set of active objects. Once the scan is done all existing overlaps have been registered. This can be seen as an invisible line that is swept from $-\infty$ to $+\infty$ perpendicular to the the axis.

Extending this to three dimensions is done by repeating the procedure for all three axes, keeping a counter for each pair of AABBs. The counter is increased each time a overlap between a pair has been reported. If the counter reaches three the AABBs overlap.

The sweep part runs in linear time and if we use a standard O(n log n) sorting method like quicksort or mergesort to sort the lists we get a time complexity of O(n + n log n) for the algorithm, but this can be improved.

Between two time steps in a simulation only minor changes in the order of the projected endpoints are expected. Sweep and prune takes advantage of this this temporal coherence by saving the sorted list from the previous invocation and using that as indata for the sorting algorithm. Two sorting algorithms that have linear behaviour for almost sorted indata are bubble sort and insertion sort[CML+94, KED05]. Using any of these two algorithms for sorting the list will allow sweep and prune to run in $O(n+ex+ey+ez)$

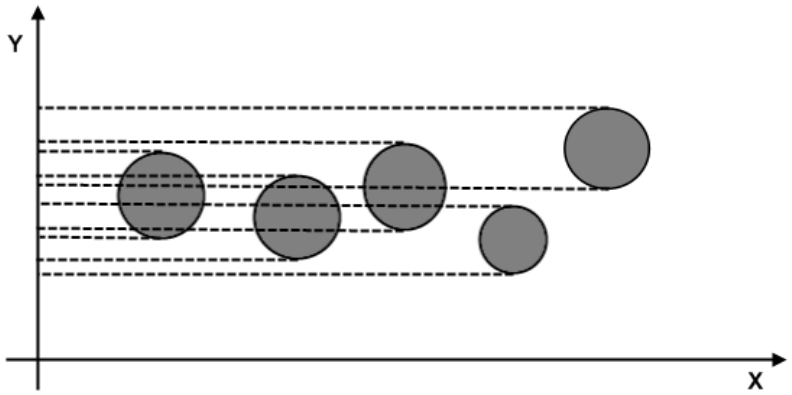

Figure 2.5: Dense clustering of endpoints along one axis may cause bad performance in the sweep and prune algorithm

time, where ex, ey, ez are the number of exchanges in along the x, y, z-axes [CML+94].

Most of the time this will be $O(n)$, with a worst case complexity of $O(n^2)$. The following is an example when the worst case could occur:

When a number of objects are placed on a horizontal plane, the projected endpoints of the objects on the vertical axis will be clustered together, see Figure 2.5. A small movement along that axis will then cause large movements in the sorted list of endpoints. The temporal coherence is lost on the y axis and the large ey term results in $O(n^2)$ complexity.

One way to avoid this clustering is to project the AABBs on the x-y, x-z, and y-z planes instead. Typically there are fewer overlaps of these 2D rectangles than of the 1D intervals. This results in fewer swaps as the objects move. [CML+94]

When using insertion sort a final optimization can be made by integrating the sorting and sweeping parts of the algorithm. While sorting, whenever b_i is swapped to the left of e_j, objects i and j are now overlapping. The counter for the pair is increased and if its value is the same as number of dimensions an overlap is reported. If e_j is swapped to the left of b_i, objects i and j no longer overlap so the counter is decreased. If the counter previously had the same value as number of dimensions the the overlap is removed. Pseudo code for this algorithm is given in Algorithm 3. [KED05]

2.4.3 Spatial Grids

Another approach to broad phase collision detection is to overlay space with a uniform grid. This forms a number of grid cells of equal size. Every object is then assigned to the cells it overlaps.

Only objects assigned to the same cell are tested for intersection, reducing the number of pairwise tests to only testing objects which are likely to overlap. “This conceptual as well as implementational simplicity has made grids a good and popular choice”. [Eri05]

One of the most important properties of the grid, having a big impact on the performance, is the size of the cells. Listed below are cases that should be avoided.

– A too fine grid, i.e the cells are too small. Objects are assigned to multiple cells

Algorithm 3 Sweep and prune

```
1. sweep-and-prune(L:List(endpoints))
2. scan = head(L)
3. while scan ≠ tail(L)
4.       mark = right(scan), left = left(scan)
5.       while scan < left do
6.            if scan = b_i and left = e_j then
7.              increment counter(i,j)
8.              if counter(i,j) = dimensions then
9.                  report overlap (i,j)
10.             end if
11.           else if scan = e_j and left = b_i then
12.             decrement counter(i,j)
13.             if counter(i,j) = dimensions − 1 then
14.                 remove overlap(i,j)
15.             end if
16.           end if
17.           scan = left, left = left(scan)
18.      end while
19.      scan =mark
20. end
```

and a lot of redundant work will be done when objects move. Each object needs to be added and removed from multiple cells.

- A too coarse grid, i.e the cells are too big, results in that many objects that are far apart are assigned to the same cell.
- A both too coarse and too fine grid. If objects in the world vary a lot in size some may be too big for the cells while others are too small, resulting in bad performance.

Using Equation (2.4) we can find the triplet that corresponds to the grid cell enclosing any point p in space.

$$\tau(p) = \left(\lfloor\frac{p_x}{r}\rfloor, \lfloor\frac{p_y}{r}\rfloor, \lfloor\frac{p_z}{r}\rfloor\right) = (\alpha, \beta, \gamma) \tag{2.4}$$

If we want the triplet to always be positive we can define a minimum point t for the world.

$$\tau(p) = \left(\lfloor\frac{p_x - t_x}{r}\rfloor, \lfloor\frac{p_y - t_y}{r}\rfloor, \lfloor\frac{p_z - t_z}{r}\rfloor\right) = (\alpha, \beta, \gamma) \tag{2.5}$$

AABBs are often represented by a max and a min point. If those points are located in different cells, the object needs to be assigned to both these cells and to all cells between them. The triplets for the cells between the max and min point can easily be found since they all satisfy the relation in Equation (2.6).

$$\tau(AABB_{min}) \leq (\alpha, \beta, \gamma) \leq \tau(AABB_{max}) \tag{2.6}$$

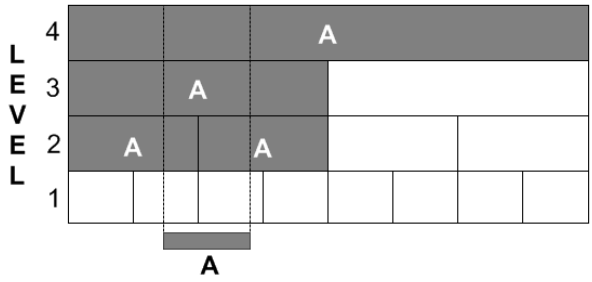

Figure 2.6: Insertion of the object A into a hierarchical grid. The shaded cells are the ones to which A has been mapped.

The naive way of implementing the grid is to use a 3D array where each element in the array corresponds to a grid cell. For each array element a pointer to a linked list of objects is stored. The big problem with this implementation is the memory requirements, since it allocates memory for all grid cells when in fact only few are used at any given time. A better solution is to use a hash table where the grid cells are mapped into a finite number of buckets. This way the storage is only dependent on the number of objects and not the size of the grid. Because of hash collisions an overlap test of the bounding volumes of objects hashed to the same bucket should still be performed.

Hierarchical Grids

One way to solve the problem with different sized objects could be to have multiple grid layers with different resolution. The layers all cover the same space and are ordered in the following way $R_1 < R_2 < ... < R_n$ where R_i is the size of the grid cells at level i. Each object is inserted into all cells it overlaps, starting at the lowest level at which the cells are large enough to contain the object. This is shown in figure (2.6).

We only want to test two objects for overlap if they are **close** to each other. Closeness is defined as:

Definition 2.4.1. Close
Boxes X and Y are **close** if and only if they overlap a common cell at level $max(level(X), level(Y))$, where $level()$ returns the lowest level the object is stored in.

Whenever an object A is inserted in a grid cell already containing another objects B and $max(Level(A), Level(B))$ corresponds to the current level in the hierarchy, a counter for the pair (A, B) is increased, and an overlap is reported. If either object is moved so they no longer share a common cell, the counter is decreased and if it reaches zero a "non-overlap" is reported. [Mir96, KED05, Eri05]

It is hard to select good values for the cell sizes and number of grid layers. A simple procedure is to let R_1 be the the smallest possible size that can encapsulate the smallest object. For each level the the size of R is doubled until R is big enough to encapsulate the biggest object. This strategy will of course be bad when there are only a few sizes of objects, some very small and some very large. [Eri05]

2.4.4 Comparison

Compared to Sweep And Prune, Hierarchical Hashing has some advantages. The time complexity is independent of the configuration of the scene while some configurations could lead to $O(n^2)$ complexity in Sweep and Prune. Another advantage is that temporal coherence does not change the time complexity of the algorithm, it only changes the workload. The time complexity of Sweep And Prune is highly dependent on temporal coherence. [KED05]

For Hierarchical Hashing to perform well we need good parameters for the number of levels and the size of the grid cells. It's also important to have a good hash table implementation. The biggest drawback is still the memory usage. According to Erleben[KED05] Hierarchical Hashing is competitive with Sweep And Prune for medium sized configurations, but as the configurations grow larger page swapping degrades the performance and Sweep and Prune becomes superior. Also, Exhaustive Search performs better than both Sweep and Prune and Hierarchical Hash Tables when the number of objects are few (<100).

Chapter 3

Haptic Rendering

"A haptic interface is a device worn by a human which provides the touch sensations of interacting with virtual objects. The device usually consists of an electro-mechanical linkage which can exert a controllable force on a human's hand.". [Zil95] The haptic device is used to perceive and interact with virtual objects and is often combined with a graphical display. Haptic rendering is analogous to graphical rendering with the difference that virtual objects are made perceivable through touch rather than vision.

Haptic interfaces have been used in a multitude of applications including surgical training, investigation of the human sense of touch, molecular docking and 3D painting and modelling. [BS01] Although there exist different types of haptic devices, we will limit this discussion to 3-DOF devices. These can be likened to 3D mice with force feedback capability. The Phantom Omni, a 3-DOF haptic device made by Sensable Inc. can be seen in Figure 3.1.

3.1 Haptic Devices

To be able to discuss haptic devices we first need to know a thing or two about the human sense of touch. Humans have two main systems for sensing touch, tactile and kinetic. The tactile system is made up by the nerve ends in the skin. With these we can sense temperature, texture and local geometry. The tactile system is sensible to vibrations up to 1000Hz. The kinetic system is made up of the receptors in the muscles, tendons and joints. With these we can feel the positions of and forces upon our limbs.

Current haptic devices often use a tool-handle approach. This means that the interaction with the virtual world is through a tool or pen. The user holds the pen and the tip of the pen called probe or Haptic Interface Point (abbreviated HIP) interacts with the virtual objects. This way the tactile system is passively satisfied. This means that the user does not directly touch the virtual objects, it is more like exploring the world with the tip of a stick.

This type of haptic device can further be classified by how many DOF[1] in/out it has. A device may be able to sense position in six DOF $(x, y, z, pitch, roll, yaw)$ but only generate force feedback in three (x, y, z).

The type of haptic device we are discussing is ground-based, that is, it is fastened to the ground. Body-based devices like haptic gloves are portable and can provide the

[1] Degrees Of Freedom

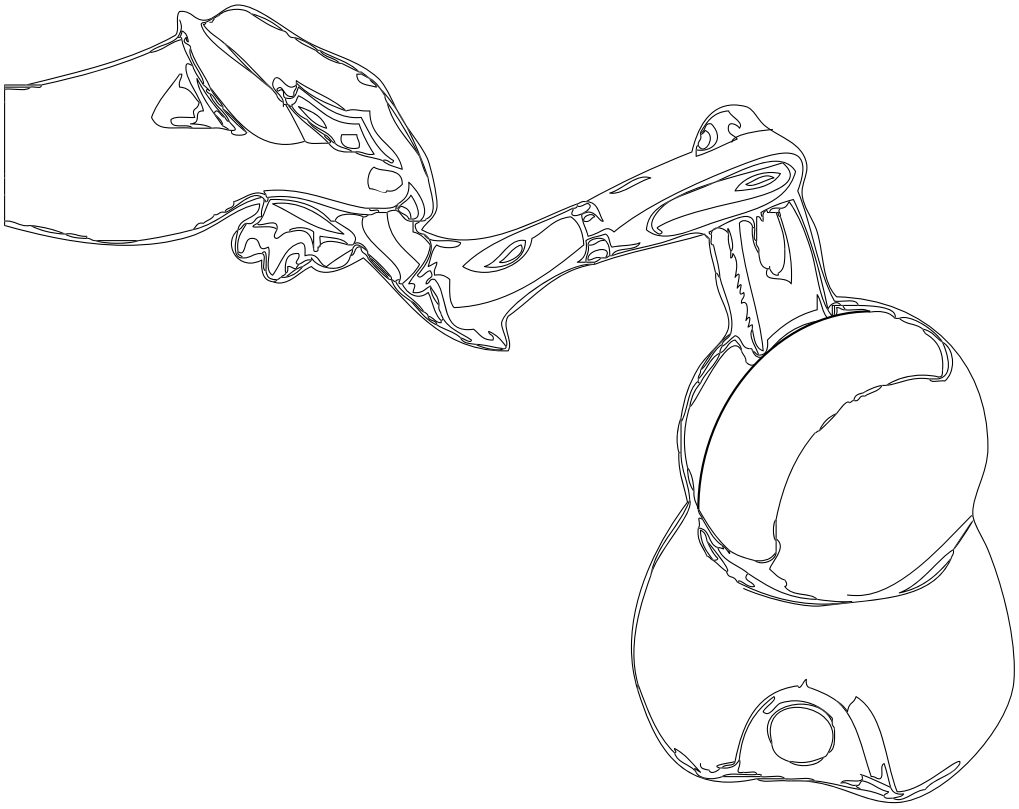

Figure 3.1: The Phantom Omni, A 3-DOF haptic device.

sensation of e.g. grasping objects but cannot stop the user from passing through virtual objects. Ground-based devices on the other hand are attached to something stable making them able to exert net forces on a user.

3.2 Rendering Algorithms

The mission of haptic rendering is to make virtual objects appear as more or less solid objects. The rendering algorithm ties together the haptic device and the virtual world. The most basic haptic property to render is the shape of a stiff object[2]. That is, we want to feel the surface of the object to explore it's shape. Later surface properties like friction, stiffness, texture etc. can be added.

We are omitting control loop algorithms for controlling the physical haptic device, the methods described here are on the level "read position, calculate reaction force".

Given the position of the haptic interface point we have to calculate the appropriate force to be applied to the haptic device to keep the HIP on the surface of objects. Due to the mechanical compliance and limited servo rate of the haptic device, the haptic interface point will already have penetrated the object when its time to calculate the reaction force.

[2]For now forgetting more imaginative things to render than "world-like" rigid bodies

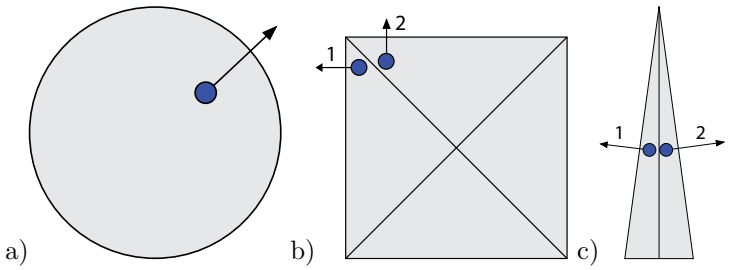

Figure 3.2: (a) The reaction force when touching a sphere can be trivially calculated by taking the vector representing the HIPs position minus the vector representing the sphere's position. (b) The direction of force may suddenly change when pushing into a subdivided object. (c) Popping through a thin object.

3.2.1 Vector-field/Penalty Methods

Vector-field methods generate a reaction force that is directly proportional to the penetration into the virtual object. That is, the reaction force is a direct function of the haptic interface point's position inside an object which means that an object is really a force-field with larger force vectors further in. For simple or "mathematically defined" geometries like spheres and planes the direction of force is easily calculated and unambiguous (e.g. $\vec{f} = [x, y, z]_{HIP} - [x, y, z]_{sphere}$), see figure 3.2a.

For more complex geometries a number of problems appear. Since the HIP will have travelled some distance into the object it is that it is not always clear towards which surface to direct the reaction force. One approach described by Massie et al. in [MS94] is to subdivide the object's internal volume and assign each sub-volume to a surface. That is, the direction of the reaction force will be the normal of the surface. For simple objects the sub-volumes can be constructed by hand fairly easy. The problems with this method are the construction of more complex objects and that sharp discontinuities in force may appear when passing from one sub-volume to another (figure 3.2b).

Another problem with vector-field methods is that thin objects cannot easily be rendered. Thin objects lack the volume to stop the HIP from passing through them. When the HIP has passed more than halfway through an object the direction of the reaction force will flip towards the other side of the object, leading to a pop-through effect [RKK97, Zil95](figure 3.2c).

It would be a nice (and basic) feature of a haptic rendering library to be able to combine objects and build larger objects out of smaller overlapping objects. It is not clear how to do this inside the vector field-model. Simply adding two overlapping force fields (objects) will producing strange effects at the intersection. Sharp corners will be rounded etc.

3.2.2 The God-object Algorithm

The problems with the vector field approach stem from (the combination of) measuring the HIPs position at a discrete rate and considering the internal volumes of objects rather than their surfaces. The fundamental problem is just that it is a *vector field* approach. We are interacting with the internal volume of an object when we want to feel its shape which is (often) really defined by its surface.

The HIP will already have penetrated the virtual object when it's time for us to

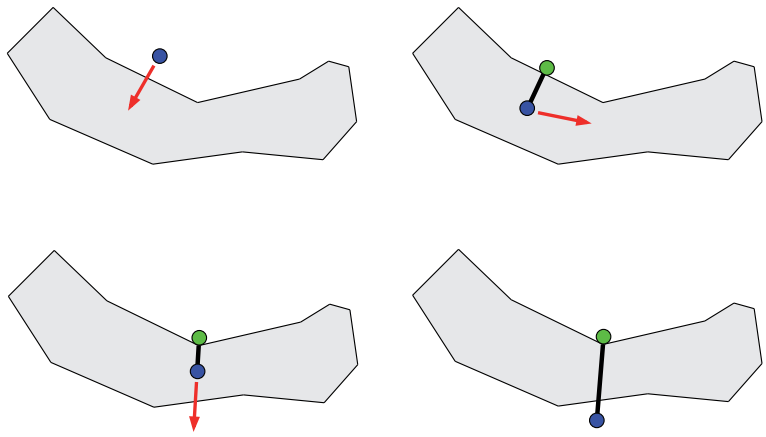

Figure 3.3: A visual description of the god-object algorithm. The IHIP stays at the surface of objects.

react. Having no history of the HIP's movement makes it hard to parry the penetration. If we could keep some kind of history of the HIPs path we would know where to direct the force.

It is time to introduce IHIP (also known as the the ideal haptic interface point, virtual proxy or god-object) which is a copy of the HIP acting as its history. Conceptually, one can think of the IHIP as being connected to the HIP through a (infinitely short) rubber band. While moving in free space the HIP and IHIP are collocated. The HIP is allowed to penetrate virtual objects while the IHIP is not. When the HIP penetrates an object the IHIP stays on the surface. The imaginary rubber band now pulls the IHIP to the point on the surface closest to the HIP and at the same time the HIP is pulled towards the surface of the object, see figure 3.3. We now have a simple algorithm outline:

- Read HIP position.
- Greedily move the IHIP as close to the HIP as possible without crossing any object surface.
- Calculate reaction force, which is a spring force between HIP and IHIP.

The only tricky part is to make the IHIP stay on the surface of objects. This can be divided into two steps: collision detection and position calculation. In each time step we are trying to move the IHIP (historic location of the HIP) to the current physical location of the HIP. We are looking for a local minimum distance since the IHIP is a history of the HIP's movement, so the IHIP should move in a straight line towards the HIP. Ruspini et al. likened this to a robot greedingly moving towards its goal . [RKK97] If the HIP is located inside an object, the line from IHIP to HIP crosses a surface since the IHIP never is inside objects. The collision detection step detects this and stops the IHIP before crossing this *constraint surface*. There can be a point on the surface that is closer to the HIP, this point is found in the position calculation step.

Finding constraint surfaces is done by sweeping the straight path from IHIP to HIP and check that path for collisions with objects. If the HIP and IHIP are modelled as points and objects as triangle meshes this will result in a line-triangle mesh collision query (a swept point is a line). This corresponds to performing the collision detection

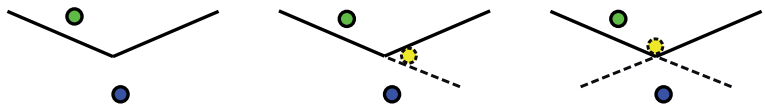

Figure 3.4: Iteratively adding constraint planes

in a fourth dimension. Techniques for speeding things up described in the collision detection chapter are of course applicable.

If there are any triangles along the path, the earliest collision is chosen and the plane defined by that triangle (that is, the plane defined by the collision point and the collision normal) is a new constraint plane. The point on the constraint plane closest to the HIP is calculated and set as the new goal for the IHIP. How the calculation is done is described later. Since the new goal might be behind other triangles the two steps are iterated, see algorithm 4. The path from the IHIP to the new goal is checked for collisions and more planes may be added to a set of constraint planes giving new goals for the IHIP, see figure 3.4.

Algorithm 4 The god-object algorithm

1. Set GOAL = HIP position
2. Check path from IHIP to GOAL for collisions.
3. If there is a collision add the plane to the set of constraint planes.
4. Calculate the new GOAL using Lagrange multipliers
5. Go to 2.
6. Set IHIP = GOAL.

At convex regions of an object the IHIP will be constrained by one plane and its motion will be limited to the plane. At concave regions the IHIP will be constrained by at most three planes. With two planes the motion is limited to the line where the two planes intersect. With three planes the motion is limited to a point. The reason for adding the planes iteratively is visualized in figure 3.4.

Note that for collision detection the shape of the object is used but when calculating the new IHIP goal the infinite plane defined by the collision point and normal, therefore the following situation occurs at convex regions of objects. The new position of the IHIP may actually be outside the object, see figure 3.5. In the next time-step the IHIP will "drop" to the surface. The time-steps and distances are small enough that this will be unnoticed by the user. [ZS95, RKK97, Zil95, BS01]

A big advantage of the god-object approach is that the IHIP can be displayed graphically to the user as the position of the HIP. Research shows that the visual sense significantly dominates the kinestethic sense of hand position and overshadows the real position of the HIP making objects feel stiffer than they really are. [SABB96]

Lagrange Multipliers

Lagrange multipliers can be used to find the extrema of multivariate function subject to one or more constraints. [GW04] The method is used to calculate the new IHIP

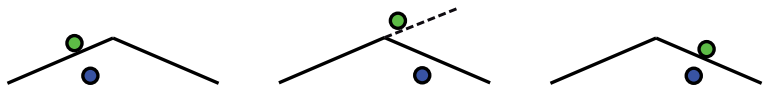

Figure 3.5: At convex portions of an object the new IHIP position might be above the surface.

position given a set of constraint planes. Equation 3.1 gives the energy in a spring of unity stiffness connecting the HIP (x_p, y_p, z_p) and IHIP (x, y, z). Minimizing the energy while observing the constraints imposed by the constraint planes gives the new position.[3] Constraint planes are added in the form shown in equation 3.2. The new position is found by minimizing L in equation 3.4 by setting all six partial derivatives of L to 0 (equation 3.4) giving an easily solvable matrix. When the IHIP is constrained by less than three planes, zeros are inserted resulting in a lower order system. [Zil95]

$$Q = \frac{1}{2}(x - x_p)^2 + \frac{1}{2}(y - y_p)^2 + \frac{1}{2}(z - z_p)^2 \tag{3.1}$$

$$A_n x + B_n y + C_n z - D_n = 0 \tag{3.2}$$

$$\begin{aligned} L = \; & \frac{1}{2}(x - x_p)^2 + \frac{1}{2}(y - y_p)^2 + \frac{1}{2}(z - z_p)^2 \\ & + l_1(A_1 x + B_1 y + C_1 z - D_1) \\ & + l_2(A_2 x + B_2 y + C_2 z - D_2) \\ & + l_3(A_3 x + B_3 y + C_3 z - D_3) \end{aligned} \tag{3.3}$$

$$\begin{bmatrix} 1 & 0 & 0 & A_1 & A_2 & A_3 \\ 0 & 1 & 0 & B_1 & B_2 & B_3 \\ 0 & 0 & 1 & C_1 & C_2 & C_3 \\ A_1 & B_1 & C_1 & 0 & 0 & 0 \\ A_2 & B_2 & C_2 & 0 & 0 & 0 \\ A_3 & B_3 & C_3 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ L_1 \\ L_2 \\ L_3 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ D_1 \\ D_2 \\ D_3 \end{bmatrix} \tag{3.4}$$

Adding Friction

Without simulation of friction, objects will feel icy and slippery. With the god-object algorithm friction can be modelled just by modifying the position of the IHIP.

Static friction (stiction) can be modelled in the following way. The force exerted on the IHIP can be estimated by the equation $f = k_p(p_{IHIP} - p_{HIP})$, where k_p is the proportional gain of the haptic controller. For each constraint plane the force can be divided into the components f_t and f_n, tangential and normal to the plane. If the surface has a static friction constant μ_s then the IHIP is in static contact if the force is inside the friction cone $\| f_t \| \leq \mu_s \| f_n \|$. When the IHIP is in static contact with any surface its position is set to same as the previous cycle. [RKK97, Zil95]

[3] To avoid falling through planes because of numerical errors the IHIP is actually placed a small delta distance above the plane.

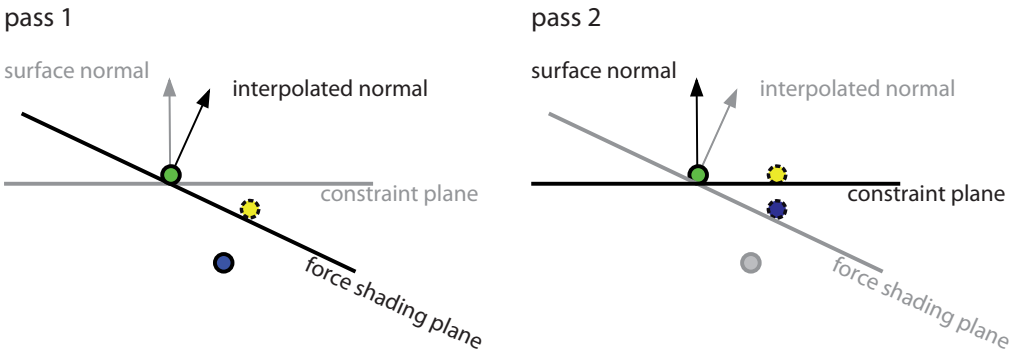

Figure 3.6: Two-pass force shading. In the first pass interpolated normals are used. In the second pass the true normals are used and the goal is the result of the first pass.

A way of modelling viscous damping and Coulomb friction can be deduced from the equation of motion of an object with mass m moving in a viscous field, along a surface that exhibits dynamic friction

$$f_t - \mu_d f_n = m\ddot{x} + b\dot{x} \tag{3.5}$$

where b is a viscous damping term, and μ_d is the coefficient of Coulomb friction. If the mass is set to zero the velocity of the object in dynamic equilibrium is given by

$$\dot{x} = \frac{f_t - \mu_d f_n}{b} \tag{3.6}$$

This limit of the IHIP s velocity is used to calculate the maximum distance it can travel in the current cycle. If it is in contact with multiple surfaces, the one with lowest velocity bound is used. If the maximum velocity is negative, the dynamic friction is enough to resist motion and the IHIP s position is not changed. If $b = 0$ the maximum velocity is unlimited. [RKK97]

Note that both friction models are independent of the motion of the HIP.

Adding Force Shading

Often we want to display objects with smooth continuous surfaces but our models are made up of connected flat polygons. Making polygonal models fine-grained enough to appear smooth is not feasible. In graphical rendering this is solved with different interpolation schemes. Either by calculating the illumination model at the vertices and interpolating the values (Gourad shading) or calculating the illumination at each pixel but interpolating the normal direction (Phong shading). [AMH02] The same approach can be applied to haptic rendering.

The normal at each vertex is calculated offline as the average of the normals of the neighbouring polygons. At runtime the normal of a point on a polygon can be calculated by taking the weighted average of the polygon's vertices exactly as in Phong shading. [Zil95, RKK97, BS99]

Just as with friction this can be accomplished in the god-object algorithm by modifying the position of the IHIP. Ruspini et al. [RKK97] describe how to integrate force shading with the god-object algorithm through a two-pass scheme. First the new position of the IHIP is calculated using constraint planes with interpolated normals. The

weights for interpolation have already been calculated in the collision detection step (the coordinates on the triangle where the line crosses the triangle). Then a second pass is performed using the true normals and the IHIP position from the first step as goal/HIP, see Figure 3.6. If the position calculated in the first pass lies above the true constraint plane, it is projected back on the plane before the second pass to make sure the final IHIP position always lies on the surface.

The distance between the IHIP and HIP might increase from force shading, implying that energy is added to the system. However, if the angle between the interpolated and true normal is small, the added energy is very small and the increase in reaction force will not be noticeable by the user.

Chapter 4

HOACollide

HOACollide is a general-purpose, object-oriented collision detection framework implemented in C++. It is extendable through dynamically loaded plug-ins.

4.1 Overview

HOACollide consists of several core parts and in this section we give a brief overview of the entire system, the purpose of the different parts and how they interact with each other. In Appendix A we provide some sample sample code that exemplify the use of the API.

Objects(things that can collide) are inserted into an *universe*, the universe acts as a controller for all the components in the library. Apart from containing all the collidable objects the universe also stores a list of pairs of objects that are in close proximity. This list is updated by a run-time exchangable *proximity detector* that implements some algorithm for broad phase collision detection. When an overlap is reported at the broad phase level (the proximity detector adds a new pair to the proximity list), a *collider* is created for the pair.

The collider is responsible for answering collision queries, e.g calculating collision data like contact points, penetration depth or boolean contact for that specific pair. The type of the collider is determined by the *geometries* of the objects, e.g. a sphere and a box will create a new sphere-box collider. Note that this is done in a dynamic fashion through plug-ins; the system has no knowledge of "boxes" or "spheres". A collider exists as long as "its" object pair overlap at the broad phase level. This allows for exploitation of temporal coherence by caching data in the collider between time steps, e.g. the last known separating plane. The universe also provides a callback mechanism that passes desired collision data back to the application level. Callbacks are set up using *object filters*.

4.1.1 Universe and the Callback Mechanism

The universe is what ties everything together and provides an API to the user of the library. This includes a callback mechanism that passes desired collision data back to the application level. In the simplest case, boolean intersection, this will only be a boolean value together with references to the colliding objects. There are currently four different events that can generate callbacks: boolean intersection, number of contact

points between two objects is greater than zero and change in broad-phase overlap state; i.e. when two objects are overlapping at the broadphase level for the first time ("Hello") and when they get separated ("Goodbye"). Whenever a *contact point callback* is triggered a collision data object is returned to the application level, containing contact point information, contact normal, penetration depth and references to the colliding objects. Apart from the collision data itself a reference to the *Collider* that calculated the information is included. This makes it possible for the application to update the position of the two objects and recalculate the collision data for the specific collision pair without having to run a full update with broadphase. This can be useful when using the library for rigid body simulations where an iterative solver is used (for example the BGF algorithm [GBF03]).

Through the use of filters, different callbacks can be registered for different combinations of events and groups of objects. *Object filters* filter *objects* with respect to specific properties of the objects. This can be the geometry type, a custom user tag or any other property of an object. It is also possible to create an catch-all filter that accepts all types of objects. When two object filters are combined they form a *pair filter*. An object pair consisting of two objects O_1 and O_2 in an universe can be tested against a pairfilter consisting of filter F_1 and F_2. If O_1 matches F_1 and O_2 matches F_2 or vice versa the object pair matches the pair filter. All callbacks has a corresponding pair filter registered to it. A callback will only be triggered when the objects matches the filter.

This gives a very flexible and fine grained controll of what objects that are allowed to collide and what callback functions should be called. A possible senario could be that the user wants to play a sound whenever *Object X* intersects a green object. This is achieved by creating two filters; one for green objects and one for the specific object *Object X* (identified by some unique property). These two filters are combined into a pair filter. The function that is responsible for playing the sound is registered in the universe as a callback function for boolean intersection events matching the pair filter.

4.1.2 Object

An object consists of a pointer to a geometry, a transformation (rotation and translation) and a bounding volume. The geometry is a representation of a shape residing in its own local coordinate system. This can be anything from a sphere represented simply by its radius to a triangle mesh with millions of triangles each represented by three float values, but the objects themselves are not aware of the type of their geometry. This abstraction enables objects to be be treated homogeneously throughout the system. The object´s bounding volume encloses the transformed geometry. Because of the previously mentioned abstraction it is the geometries that are responsible for calculating a bounding volume given a transformation matrix. Multiple objects can share the same geometry, allowing for reduction of memory usage. This reduction can be significant in scenes where there are multiple identical trimesh objects consisting of hundreds of thousands of triangles. Geometry types are implemented as plugins, allowing for easy expansion. Below is a list of geometry plugins already implemented in HOACollide.

Triangle Soup

Triangle soups are stored as a list of vertices and a list of indices into the first list, where each triplet of indices forms one triangle. An AABB hierarchy is built for each triangle soup. The hierarchy is built using a top-down approach where we use the longest side cut strategy for finding the splitting point. If all the edges are the same length the x

axis will be picked. When this algoritm performs bad and all triangles end up on the same side of the splitting plane we implemented an alternativ algorithm that selects the splitting point as the median of all the centroids. The optimal solution to this problem would have been to implement multiple strategies and for each tringle soup select the strategy that generates the smallest total volume. The choice of bounding volume was motivated by its simplicity and for its low cost for refitting when used with a deformable mesh. Although refitting is not implemented yet it is probably something that is desirable to add in the future.

Box

Boxes are stored in the halfwidth extents format, as specified by the COLLADA[1] standard. Each box is represented by a center point and halfwidth extents or radii along the x, y and z axes, Since boxes are stored in modelspace the center is always origo and no center point need to be stored. For a box only three float values are stored. Comparing this with the min-max representation of an box we are saving three float values for each box.

Sphere

Spheres are stored on the centerpoint - radius format. Since the centerpoint is always origo we don't store that. Thus, for a sphere only one float value needs to be stored.

Particle Systems

In HOACollide particle systems are very much a special case since they have no geometry and no transformation. The particles are stored as a list of positions. Due to the dynamic nature of a particle system it does not have a dynamic bounding box like the other objects. Recalculating it each time step would be expensive, and if one or more particles would get separated from the big mass the bounding volume could get very large making it useless. It is possible to set a static bounding volume of the particle system.

HOACollide has support for enclosing one object in another. Since having one object enclosed in another would make the broad phase collision very tricky, enclosed objects are never tested at the broadphase level. By adding the particle system as an object enclosed by another object in the scene it is never tested at the broadphase level and no bounding volume is needed.

In some applications e.g. SPH[2] simulations one wants to find the neighbours of each particle, where the neighbours are defined as all particles within a certain radius. HOACollide includes a routine for neighbour searching using a spatial hashing approach. Some of the tips and tricks described in by Ericson and Hjelte in [Eri05, Hje05] are used, namely the implementation of the hash table and other data structures.

Line / Swept Point

Lines or swept points are stored as six float values, three for each end's position. A line can be seen as a point swept in time. The line object was implemented beacuse it is what HOAHaptics is using as collision object when a point is used as probe.

[1] "COLLADA is a COLLAborative Design Activity for establishing an open standard digital asset schema for interactive 3D applications´´ www.collada.org

[2] Smoothed Particle Hydrodynamics, a method for simulating fluids using particles.

Capsule / Swept Sphere

Capsule or Swept Sphere is stored in the same way as the line but with an additional float value for the radius, in total seven float values. The reason why we implemented capsule is the same as for line i.e. it is needed by HOAHaptics if a sphere is used as probe object. The Capsule can be seen as a sphere swept in time.

4.1.3 Colliders and the Plugin Manager

Colliders are responsible for calculating collision data for a pair of objects e.g. contact points, penetration depth, closest features, etc. There is one collider type for each combination of geometry types that should be able to collide. The current version of HOACollide supports two types of queries, boolean and contact points. Boolean queries return true/false depending on if the objects intersect. Contact point queries give the collision normal, a subset of contact points and the penetration depth. More query types should be straight-forward to add, all that needs to be done is to implement a given interface. Colliders are compiled to shared libraries which are loaded at run-time. This makes it easy to add new colliders or replace existing ones. HOACollide has support for both Unix and Windows shared libraries (.so and .dll files).

The *Plugin Manager* is a collider factory responsible for the loading and creation of colliders. At startup it loads all the colliders found in a given directory and creates a table that maps a pair of geometry types to the constructor of the corresponding collider type. The actual filename of the shared library is used for identifying the type of collider. Filenames follow the pattern Geometry1Geometry2Collider.so,dll.

Whenever two objects intersect at the broad-phase level a collider is created (only if there exists a callback function with a matching pair filter). The universe asks the plugin manager for a new collider object, the plugin manager performs a lookup in the table and returns the correct type of collider. In the current design a new collider is created each time. A better approach would perhaps be to have a singleton collider object of each type and an optional contact information cache object that might be created for each pair. This would improve the performance when no caching is used and the number of objects is large.

4.1.4 Proximity Detector

Since we want to be able to use different algorithms for broad-phase collision detection a standardized interface for communicating with the algorithms has been defined, the proximity detector. Algorithms can be interchanged at run-time. So far, two algorithms that implement this interface have been implemented. "Exhaustive search" using AABBs and "Sweep and Prune". Exhaustive search was implemented beacuse it is very easy to implement and performs well when the number of objects are few. For the cases were we have a lot of objects we implemented the "Sweep and Prune" algorithm since it doesn't suffer from the potential memory problems that hierarchical hashing has when the number of objects is very high. The sweep and prune implementation is a combination of Algorithm 3 in section 2.4 and techniques described in [Eri05].

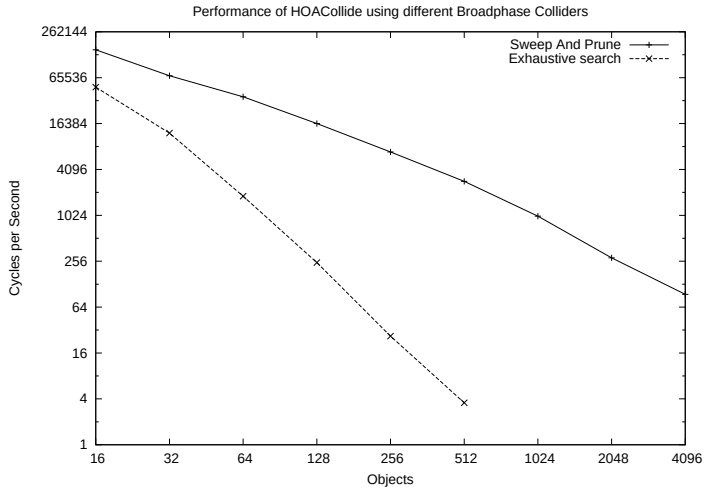

Figure 4.1: Performance of HOACollide when the number of objects in the scene is increased. Performance is measured in collision cycles per second.

4.2 Performance

To test the performance of the collision library the following scene was created. X number of objects are created inside an imaginary box and given a random velocity. Whenever a object moves outside the imaginary box its velocity is mirrored. This ensures that all objects are inside the box at all times. When the number of objects is increased so is also the volume of the box. The ratio between box size and number of objects inside it is kept constant so that we always have around 20 percent of objects colliding. To make sure only the performance of the collision library is tested no visualization of the objects is done in the test application. Performance is measured in collision cycles per second and memory usage.

By looking at the graph in (Figure 4.1) it is clear that sweep and prune outperforms exhaustive search, even when there are few objects. We expected exhaustive search to be faster for few objects due to the extra overhead of the sweep and prune algorithm. But the sweep and prune algorithm had one extra advantage we didn't think of, and that is that it only report new overlaps. The exhaustive search algorithm has no memory and reports overlaps each timestep ignoring if they where overlapping previous step. This forced us to add an extra test to verify that we hadn't already reported the overlap before creating a new collider. This extra test is probably why sweep and prune performs better than exhaustive search even for a few number of object.

In figure 4.2 a graph of the memory usage of the library is shown. When it comes to memory usage sweep and prune is using alot more memory when the number of objects is increased. Due to this we did some additional testing and found that the biggest memory thief in the sweep and prune implementation was the hash map[3] we were using as a counter table. It maps a pair of objects to the number of axes that the pair is overlapping on. In a scene of 4000 objects the map was responsible for around 90 percent of the memory usage. Due to this we tested to alter the sweep and prune implementation so that it had a staic matrix of size 4000×4000, and gave each object a

[3]The map used is the stdext hash_map distributed with visual studio by Microsoft

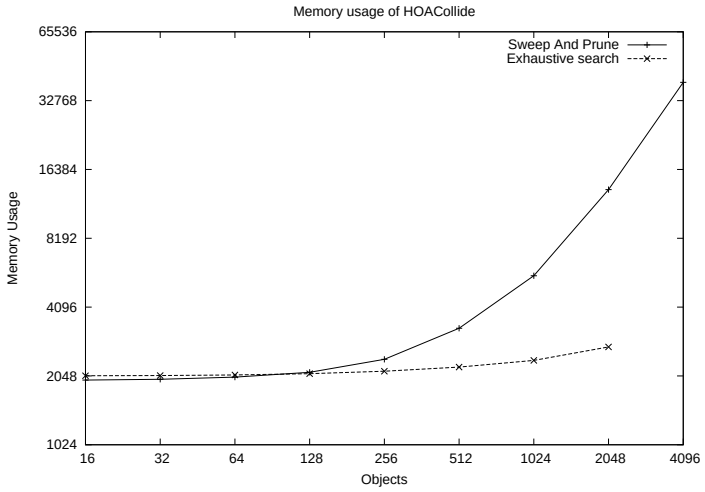

Figure 4.2: How memory usage of HOACollide is affected by the number of objects in the scene for the broadphase collision algorithms.

unique integer value between 1 and 4000. This way we could use the matrix instead of the map as counter table. This implementation was about 1.5 times faster and it used about the same amount of memory. This confirmed that the hash map implementation we used was not performing well in memory or speed. The best way to improve the memory usage and the performance of the sweep and prune algorithm would be to implement a new memory efficient hash map and replace the current one.

4.3 Test Applications

During development of HOACollide a number of test applications were written in order to verify the correctness and usability of the library. For debugging purposes it is very helpful to visualize the contact data. A viewer where objects can be moved around with the mouse was written. When two objects intersect, contact points and normals are shown (Figure 4.3). This has been the most valuable tool when writing the different colliders.

Since manually moving around a small number of objects only give an indication of the correctness and says nothing about perfomance and scalability some kind of "automated" large-scale testing had to be done. We did this by integrating HOACollide with a physics engine[4] that supports rigid bodies and particle systems. Figure 4.5 gives an example of rigid body collisions using HOACollide. Finally we implemented an application that simulate cloth using a particle system and stick constraints (Figure 4.4).

[4]Written by Markus Lindgren and Olov Ylinenpää as a part of the course "Visual Interactive Simulation" at Umeå University.

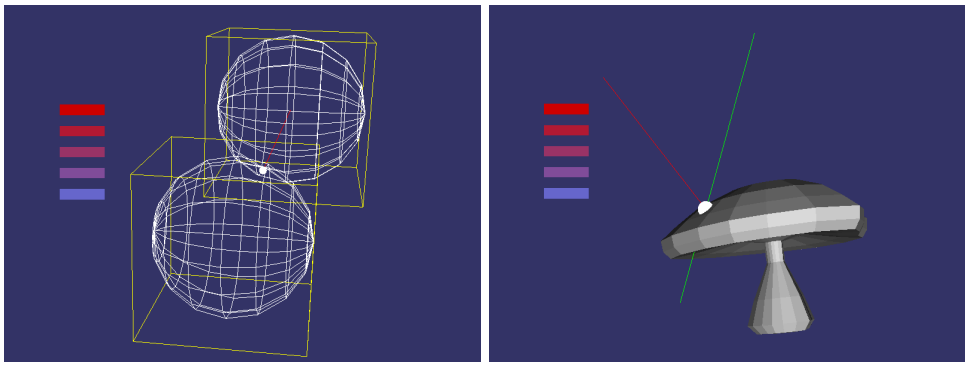

Figure 4.3: Screen captures of the viewer test application. To the left two spheres are in contact, their bounding boxes, the collision point and collision normal are shown.

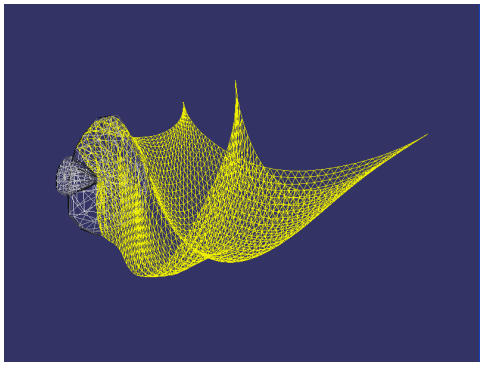

Figure 4.4: A cloth simulated by particles connected with stick constraints colliding with a triangle mesh in the form of a mushroom.

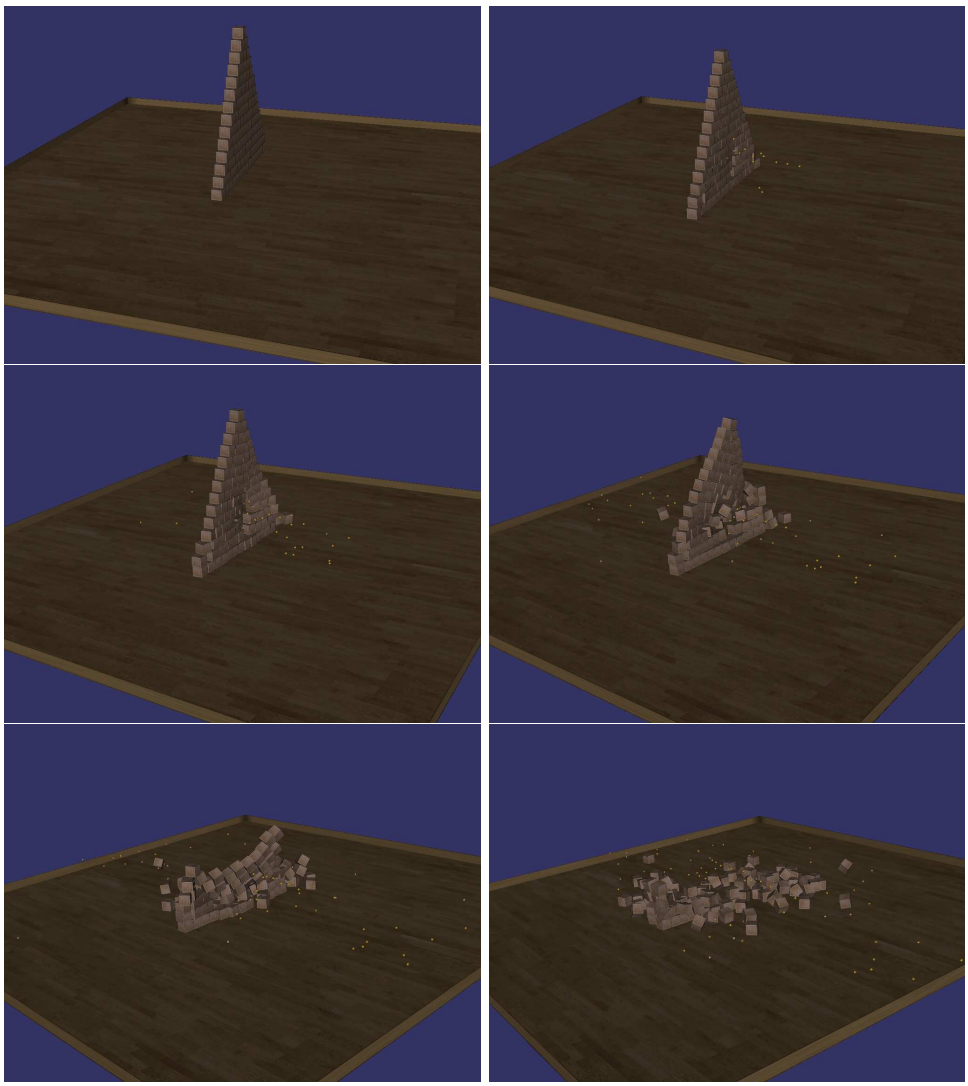

Figure 4.5: A series of screen captures from the physics simulation test application. A pyramid of boxes is bombarded with small spheres, demolishing it.

Chapter 5

HOAHaptics

HOACollide is a library for haptic rendering with 3-DOF haptic devices. It uses the god-object algorithm described in Chapter 3.2.2. The collision detection i.e. finding constraint planes is done by integration with HOACollide.

5.1 HOACollide Integration

Collision detection is done by registering a callback function in HOACollide for collisions between the probe (or rather the path swept by the probe during one time-step) and objects that we want to touch. The path swept by a point in one discrete step is a line so for collision detection a line is used. The constraint planes used in the god-object algorithm can be formed directly from the collision point and normal returned by HOACollide. If there are multiple collisions the earliest one is picked. In the current version the geometry of the probe is limited to Sphere or Point since these are the two geometries that we have implemented a swept version of, i.e Capsule and Line. All geometries that have a collider implemented for colliding with a swept point or swept sphere can be rendered haptically. Thanks to HOACollide's flexible approach, adding haptic rendering support for a new geometry type is as easy as writing a new collider.

5.2 Implementation Details

The form of the "Lagrange matrix" (see chapter 3.2.2) is known and simple to solve so we decided to solve it "analytically". Since the constraint planes are added one by one and we have to solve the equation up to three times we can reuse some intermediary results. Equation 5.1 is the equation we have to solve for the first constraint plane.

$$\begin{bmatrix} 1 & 0 & 0 & A_1 \\ 0 & 1 & 0 & B_1 \\ 0 & 0 & 1 & C_1 \\ A_1 & B_1 & C_1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ L_1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ D_1 \end{bmatrix} \tag{5.1}$$

We can rewrite this to Equation 5.2 (where $r = -A1 \times A1 - B1 \times B1 - C1 \times C1$ and $t = D1 - A1 \times x_p - B1 \times y_p - C1 \times z_p$).

$$\begin{bmatrix} 1 & 0 & 0 & A_1 \\ 0 & 1 & 0 & B_1 \\ 0 & 0 & 1 & C_1 \\ 0 & 0 & 0 & r \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ L_1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ t \end{bmatrix} \tag{5.2}$$

The new position of the IHIP is then found by solving Equation 5.2 for x, y, z. If this point also lies inside an object an additional constraint plane is added, giving equation 5.3.

$$\begin{bmatrix} 1 & 0 & 0 & A_1 & A_2 \\ 0 & 1 & 0 & B_1 & B_2 \\ 0 & 0 & 1 & C_1 & C_2 \\ A_1 & B_1 & C_1 & 0 & 0 \\ A_2 & B_2 & C_2 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ L_1 \\ L_2 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ D_1 \\ D_2 \end{bmatrix} \tag{5.3}$$

In the same way Equation 5.3 can be rewritten to Equation 5.4, where r and t are the same as in Equation 5.2.

$$\begin{bmatrix} 1 & 0 & 0 & A_1 & A_2 \\ 0 & 1 & 0 & B_1 & B_2 \\ 0 & 0 & 1 & C_1 & C_2 \\ 0 & 0 & 0 & r & s \\ 0 & 0 & 0 & s & v \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ L_1 \\ L_2 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ t \\ w \end{bmatrix} \tag{5.4}$$

When solving the three constraint plane version, Equation 5.5, we can reuse r, t, s, v, w from previous computations, see Equation 5.6.

$$\begin{bmatrix} 1 & 0 & 0 & A_1 & A_2 & A_3 \\ 0 & 1 & 0 & B_1 & B_2 & B_3 \\ 0 & 0 & 1 & C_1 & C_2 & C_3 \\ A_1 & B_1 & C_1 & 0 & 0 & 0 \\ A_2 & B_2 & C_2 & 0 & 0 & 0 \\ A_3 & B_3 & C_3 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ L_1 \\ L_2 \\ L_3 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ D_1 \\ D_2 \\ D_3 \end{bmatrix} \tag{5.5}$$

$$\begin{bmatrix} 1 & 0 & 0 & A_1 & A_2 & A_3 \\ 0 & 1 & 0 & B_1 & B_2 & B_3 \\ 0 & 0 & 1 & C_1 & C_2 & C_3 \\ A_1 & B_1 & C_1 & r & s & g \\ A_2 & B_2 & C_2 & s & v & h \\ A_3 & B_3 & C_3 & g & h & k \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ L_1 \\ L_2 \\ L_3 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ t \\ w \\ m \end{bmatrix} \tag{5.6}$$

The position of the HIP and IHIP can be read from the library and displayed graphically by the client application.

HOAHaptics utilizes an existing library called Haptik. Haptik is an open source library that acts as a hardware abstraction layer to provide uniform access to haptic devices. It does not contain any graphic primitives or rendering algorithms, only a uniform way to read the position and set the feedback force of haptic devices. Figure 5.1 shows the relation of the different libraries.

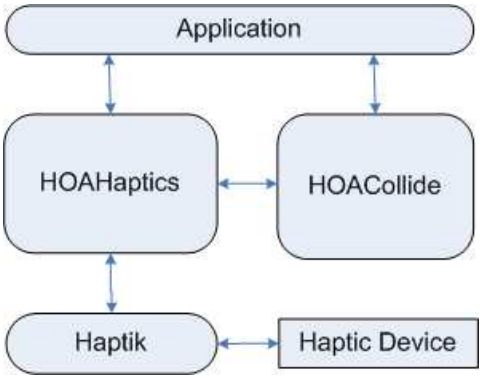

Figure 5.1: Relationship between HOAHaptics, HOACollide, Haptik and application.

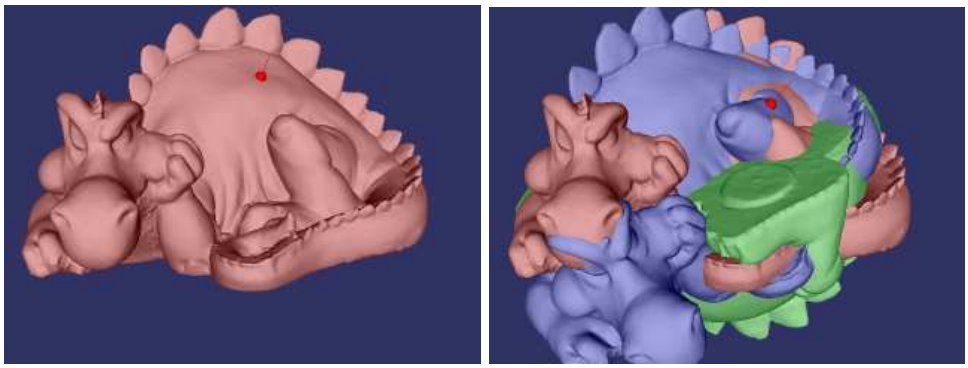

Figure 5.2: Haptic rendering using HOAHaptics, the IHIP and a visualisation of the reaction force vector can be seen. The dragon model is made up of 480 076 triangles. The right image shows haptic rendering of overlapping objects.

5.3 Result

HOAHaptics has successfully been used in an application where triangle meshes can be loaded and haptically sensed. The interaction is stable and responsive, no apparent penetration of the objects are noticed when haptically sensed. Neither friction nor force shading has been implemented yet. Figure 5.2 shows HOAHaptics in action.

Chapter 6

Conclusions

As stated earlier in the report our initial goal was to design and implement a library and API for haptic rendering and a well designed, extendable collision toolkit. Was the goal reached? The short answer to that would be yes, we have implemented an extendable collision detection library (HOACollide). We have also implemented stable haptic rendering with the HOAHaptic library, capable of rendering of models consisting of millions of triangles. The libraries could be integrated into osgHaptics replacing OpenHaptics.

During the design phase we came up with with some good and some not so good ideas. The resulting design should not be seen a final solution but as a possible first step towards a good collision detection library. To cut the development time we decided to use as much preewritten code for datastructures, visualization and memory management as possible. Being able to extend the library through plug-ins was a requirement raised by VRLab due to licensing issues and we feel that it was the right way to go.

Our conclusion regarding the design is that it is hard to construct an almighty collision detection library that can be used in all different areas. There's a trade-off between speed and how flexible the library is. Seen from a haptic perspective it might have been better to have a specialized collison library tightly integrated with the haptics but on the other hand it would be useless in other areas. In physics based simulations it could be better to merge the collision detection and physics to gain performance. We found that particle systems were the hardest to integrate in to our design because they don't fit the broadphase/narrowphase approach and the desirable collision data differs from "solid objects". The decision to use prewritten code saved us coding time but on the other hand it was what later caused the memory problems for the sweep and prune algorithm.

Some of the concepts that seem promising in the final implementation and that we would keep if we had the chance to do it again include:

- **Colliders -** Since they make the library simple to extend and improve. They give a natural place to cache data between time-steps and the ability to get hold of the object that calculated he collision data so that you can itterate it independently of the global system.

- **Filters -** Provides a neat solution for the controlling what objects that can collide and also a powerful tool for building new functionality such as grouping objects.

– **Geometry + Object -** Seems to be the most commonly used solution and saves alot of memory in scenes where the same geometry is used repeatedly.

Due to the fact that the haptic hardware rendering arrived late during the work with the thesis, most of the time was spend on development of the collision library. This is reflected both in the report and the final result. It is hard to implement something that is general-purpose without a full view of all its potential usages. Although the library is flexible and extendable it can be argued that our design is biased towards impulse based rigid body and haptic rendering using the god-object algorithm.

The amount of time/work spent on verifying the correctness of the collision detection was significant. We had to write a number of test applications and we also integrated the library in a physics simulation. But since most if these tools are resuable they should make further development of the library faster.

6.1 Future Work

Of course there are potentially an infinite number of things that could be added to the libraries. Here are some of the more interesting examples that we feel would make the library even better:

– Compound geometries i.e. being able to combine two or more geometries into a single geometry. This could be implemented either inside HOACollide or on top of it using filters.

– Adding more primitive tests, that is writing more plug-ins (a neverending task).

– Add new events that could trigger callbacks.

– No effort has been made to parallelize the library, but there should be some areas that would benefit from threading. The near field collision detection is trivially parallelizable, each collision pair can be treated independently from the other pairs.

– The ability to divide the universe into multiple broadphase regions, to make it even more flexible.

For HOAHaptics there are a number of additions that could be made to the library. The following are the ones we think are most urgent, with exception of the last item:

– Addition of friction and force shading.

– Add a framework for force effects such as vibrations, force fields, sticky surfaces..

– Add support for additional proxies such as spheres, cubes etc.

– Add support for 6-dof haptic devices.

6.2 Derived Work

In 2007 Algoryx Simulation AB was formed as an off-spring from Umeå University. They have developed a Multiphysics Toolkit called AgX. The source code of the HOACollide library was used as a starting point for the development of AgX and some design aspects still remain.

Chapter 7

Acknowledgements

The authors would like to thank the following: Our tutors Anders Backman and Kenneth Bodin. Claude Lacoursière and Daniel Sjölie at VRlab. Markus Lindgren, Patrik Berg, Tommy Löfstedt, Lisa Peterson-Berger and Sofia Kerttü for support.

References

[AMH02] T. Akenine-Möller and E. Haines. *Real-Time Rendering*. A.K. Peters Ltd., second edition, 2002.

[BS99] Cagatay Basdogan and Mandayam A. Srinivasan. Efficient point-based rendering techniques for haptic display of virtual objects. *Presence*, 8:477–491, 1999.

[BS01] C. Basdogan and M. Srinivasan. Haptic rendering in virtual environments. In K. Stanney, editor, *HandBook of Virtual Environments*, pages 117–135. CRC, 2001.

[CML+94] J. D. Cohen, D. Manocha, M. C. Lin, , and M. K. Ponamgi. Interactive and exact collision detection for large-scaled environments. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, 1994.

[Ebe04] David Eberle. Primitive tests for collision detection. Siggraph 2004 course, 2004. www.siggraph.org/s2004 (visited 20 November, 2006).

[Eri05] C. Ericsson. *Real-Time Collision Detection*. Morgan Kaufmann, San Francis, 2005.

[GBF03] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. In *ACM Trans. Graph. (SIGGRAPH Proc.)*, volume 22, 3, pages 871–878, 2003.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[GW04] David Gluss and Eric W Weisstein. Lagrange multiplier. http://mathworld.wolfram.com/LagrangeMultiplier.html (visited December 15, 2006), 2004.

[Hje05] N. Hjelte. Smoothed particle hydrodynamics on the cell broadband engine. Master's thesis, Umeå University, 2005.

[Hub95] P. Hubbard. Real-time collision detection and time-critical computing. In *Proc. 1 st Workshop on Simulation and Interaction in Virtual Environments, U. of Iowa*, 1995.

[Hub96] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, 1996.

[IJP95] R. L. Grimsdale I. J. Palmer. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105–116, 1995.

[KED05] K. Henriksen K. Erleben, J. Sporring and H. Dohlmann. *Physics-Based Animation*. Charles River Media, INC, Massachusetts, 2005.

[KHM+98] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM.

[KPLM97] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha. Spherical shells: A higher-order bounding volume for fast proximity queries. Technical report, Department of Computer Science, University of North Carolina, 1997.

[LM03] M. Lin and D. Manocha. Collision and proximity queries. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry.*, pages 787–808. Chapman & Hall/CRC, 2003.

[Mir96] Brian Vincent Mirtich. Impulse-based dynamic simulation of rigid body systems. Master's thesis, University of California, Berkeley, 1996.

[MKE03] Johannes Mezger, Stefan Kimmerle, and Olaf Etzmuß. Hierarchical Techniques in Collision Detection for Cloth Animation. *Journal of WSCG*, 11(2):322–329, 2003.

[MS94] T H Massie and J K Salisbury. The phantom haptic interface: A device for probing virtual objects. In *Proceedings of the ASME Winter Annual Meeting, Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems Vol. DSC 55-1*, pages 295–302, 1994.

[RKK97] Diego C. Ruspini, Krasimir Kolarov, and Oussama Khatib. The haptic display of complex graphical environments. *Computer Graphics*, 31(Annual Conference Series):345–352, 1997.

[SABB96] Srinivasan, M A, G L Beauregard, and D L Brock. The impact of visual information of the haptic peception of stiffness in virtual environments. In *ASME Winter Annual Meeting*, 1996.

[vdB97] Gino van den Bergen. Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, 2(4):1–13, 1997.

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984.

[Zac02] Gabriel Zachmann. Minimal hierarchical collision detection. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 121–128, New York, NY, USA, 2002. ACM Press.

[Zil95] Craig B. Zilles. Haptic rendering with the toolhandle haptic interface. Master's thesis, MIT Dep. of Mechanical Engineering, 1995.

[ZL03] G. Zachmann and E. Langetepe. Geometric data structures for computer graphics. In *SIGGRAPH '03 Proceedings. ACM Transactions of Graphics*, July 2003. Tutorial 16.

[ZS95] C. B. Zilles and J. K. Salisbury. A constraint-based god-object method for haptic display. In *Proceedings of the IEEE/RSJ International Conference on Human Robot Interaction and Cooperative Robots*, volume 3, pages 146–151, 1995.

Appendix A

HOACollide API Example

To give a feeling for the API, a very basic example of its use is shown here.

```
using namespace HOACollide;


//Broadphase callback function
void bpCallback(ContactPair cpair, BroadPhaseEvent e)
{
   ...
}
//Callback function, intersection
void boolCallback(ContactPair cpair)
{
   ...
}
//Callback function, contact points
void collisionCallback(ref_ptr<ContactData> cpoints)
{
   ...
}




int main( int argc, char **argv )
{
   //Create a collision universe
   Universe* collisionUniverse = new Universe();

   //We want to use sweep and prune for broad phase
   collisionUniverse.setBroadPhaseAlgorithm(new SweepAndPrune(collisionUniverse));


   //We need some objects to populate the universe
   Box aBox = new Box(vr::Vec3(1, 3, 2));
```

```
   Object aBoxObject = new Object(aBox);

   //Create two objects with a shared geometry
   Sphere aSphere = new Sphere(0.5);
   Object aSphereObject1 = new Object(aSphere);
   Object aSphereObject2 = new Object(aSphere);

   //Add them to the universe
   collisionUniverse.addObject(aBoxObject);
   collisionUniverse.addObject(aSphereObject1);
   collisionUniverse.addObject(aSphereObject2);


   //Register some callbacks

   //This callback is invoked each time the specific object "aBoxObject"
   //intersects with any object. Contact points are calculated
   PairFilter aPairFilter = PairFilter(ObjectFilter(aBoxObject), ObjectFilter());
   collisionUniverse->registerCallback(aPairFilter, collisionCallback);


   //This callback is invoked each time a Sphere intersects with another Sphere
   aPairFilter = PairFilter(ObjectFilter("Sphere"), ObjectFilter("Sphere"));
   collisionUniverse->registerCallback(aPairfilter, boolCallback) );


   //This callback is invoked each time any two objects get in close proximity
   collisionUniverse->registerBroadPhaseCallback(PairFilter(), HELLO, bpCallback);


   //The system is initiated and we're good to go
   while(42)
   {
      moveStuff();
      collisionUniverse.update();
   }
}
```