Natural Interaction in Virtual Environments

Gabriel Zachmann
University Bonn, Informatik II
Römerstraße 164
53117 Bonn, Germany
email: zach@cs.uni-bonn.de

Abstract

This paper presents a number of algorithms necessary to achieve natural interaction in virtual environments; by "natural" we understand the use of a virtual hand as naturally as we are used to manipulate our real environment with our real hand.

We present algorithms for very fast collision detection, which is a necessary prerequisite for natural interaction. In addition, we describe a framework for preventing object penetrations while still allowing the object's motion in a physically plausible way. Finally, we explain a model for naturally grasping virtual objects without resorting to gesture recognition.

Keywords: Virtual prototyping, virtual reality, collision detection, gesture recognition, physically-based interaction.

1 Introduction

Virtual reality (VR) promised to allow users to experience and work with three-dimensional computer generated scenes just like with real environments. Yet, after 15 years of research, this has come true only to a very limited extent. Actually, in recent years, applications of VR have specifically turned away from datagloves and virtual hands towards simpler input devices and interaction paradigms. The success of VR is mostly due to other benefits, such as the complete surround-view, its use as a communication platform for reviewing new CAD models of CAE results, and 6D tracking. So while VR offers a lot of other efficient and more or less intuitive interaction paradigms today, users still cannot interact with virtual environments just like they do in the real world, that is, by grasping, pinching, pushing, etc., virtual objects with their virtual hand. This is even more surprising, since the human hand is probably our most important and by far most frequently used tool in the real world.

There are, however, a number of applications which would actually require an exact and faithful modeling of the human real hand-environment interaction. One of them is virtual assembly simulation [JJWT99, Zac99, ZR01]. In order to be able to make a correct verification of the assembly process (which is performed by human workers), one must be able to simulate the interaction between the worker's hands and the real car as complete and correct as possible. Other applications are immersive ergonomic investigations and 3D sketching. Such an interaction technique might even open up possibilities for novel types of games (other than racing, shoot 'em up, fighting, and quests).

In this paper, we present therefore a number of algorithms which are needed to solve this problem. The first are fast collision detection algorithms, because any kind of collision handling must be triggered by collision detection. Then, we turn to the problem of making objects behave in a plausible, realistic manner, so that VR users can manipulate and assemble objects even in crowded and tight virtual environments (VEs). Finally, we propose a framework which allows to grasp and manipulate objects by the virtual hand in a natural way.

2 Collision detection algorithms

Collision detection can be regarded as a pipeline of successive filters [Zac01]. This concept is somewhat similar to the concept of a rendering pipeline or visualization pipeline. The input of the collision detection pipeline is a set of objects, while the output is a set of pairs of objects (and possibly polygons).

During run-time, objects, together with their current position, are entered in the queue when they have been moved. Conceptually, the front end passes on all possible pairs of objects down the collision detection pipeline. After the front end, the collision interest matrix filters out all object pairs of no interest. After that, object pairs are filtered further by two neighbor-finding stages and finally by an exact collision test (polygonal collision algorithms can be considered another filter before the polygon-polygon test).

In the following, we will present just two algorithms for two stages of the collision pipeline. A complete

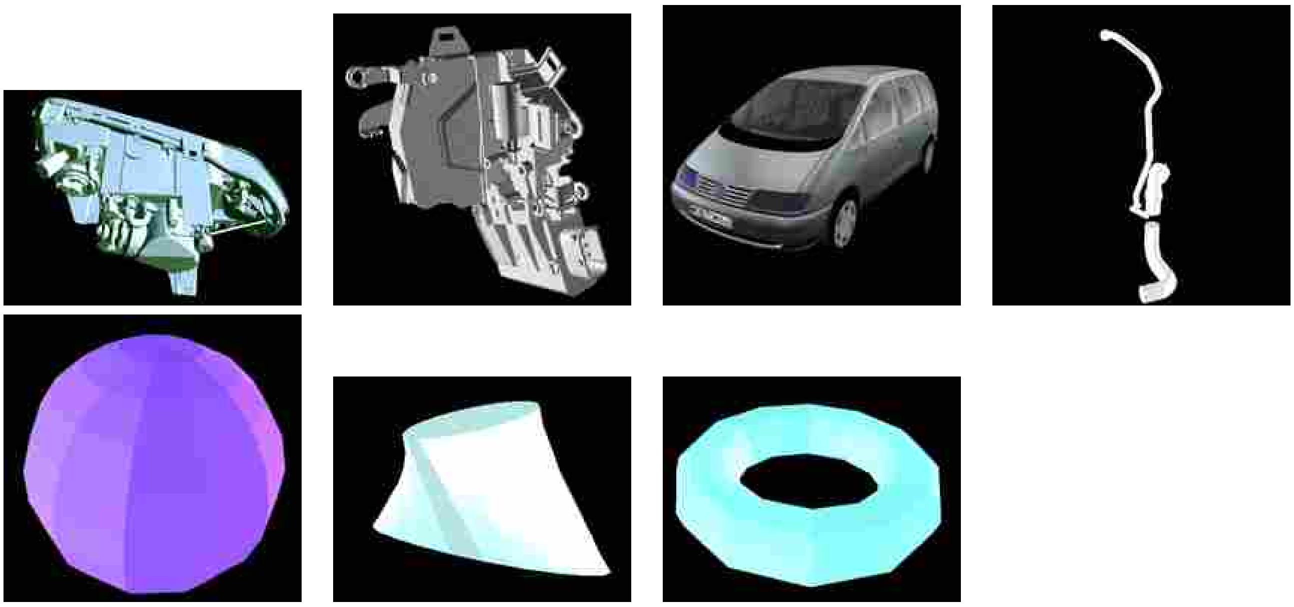


Figure 1: Our suite of test objects. They are (left to right): a car headlight, the lock of a car door, body and seats of a car, hose of a car engine, sphere, hyperboloid, torus. (Data courtesy of VW and BMW)

treatise on the collision detection pipeline and the algorithms involved in each stage can be found in [Zac00, Zac01].

2.1 DOP trees

At the last stage of the collision detection pipeline, we are given one pair of objects and have to determine their collision status. Research has shown that hierarchical algorithms can solve this problem very efficiently [Hub95, OD99, GLM96, Zac97].

Here, we will describe only one such hierarchical algorithm, namely the DOP tree. It is based on a hierarchical BV tree, where each node is enclosed by a discretely oriented polytope (DOP); see [Zac00] for further details.

As with other hierarchical collision detection algorithms, the main problem is to find a fast overlap test for transformed BV hierarchies. Given two DOP-trees O and Q, the basic step of the simultaneous traversal is an overlap test of two nodes. In order to apply the simple and very fast interval overlap test to DOP-trees, they must be given in the same space. However, at least one of the associated objects has been transformed by a rigid motion, so the DOPs of its DOP-tree are "tumbled" — in fact, in any other than the object's coordinate system the DOPs are no longer DOPs in the strict sense.

The idea is to enclose a tumbled DOP by another, "axis-aligned" DOP. We call this process *(re-)aligning*. The re-aligned DOP is, of course, less tight than the original one. On the other hand, the overlap test between two aligned DOPs is much faster than between non-aligned ones.[1]

By choosing P's object space (w.l.o.g.), we need to re-align only Q's nodes as we encounter them during traversal. We will show that this can be done by a simple affine transformation of the DOP's plane offsets. We would like to remark that, except for the interval overlap test of DOPs, the alignment algorithm works in the case of general DOPs as well.

Assume we are given a (non-aligned) DOP D of Q's DOP-tree, which is represented by $\mathbf{d}$. Assume also, that the object associated with Q has been transformed by a rotation M and a translation $\mathbf{o}$, with respect to O's reference frame. Then D is the intersection of k half-spaces

$$h_i : \mathbf{b}_i\mathbf{x} - d_i + \mathbf{b}_i\mathbf{o} \leq 0,$$

where $\mathbf{b}_i = B_i M^{-1}$ (see Figure 2).

Now suppose we want to compute the d'_i of the enclosing DOP D' of D. There is (at least) one extremal vertex P_i of the convex hull of D with respect to $\mathbf{B}_i$. This vertex is the intersection of 3 (or more) half-spaces $h_{j^i_l}, 1 \leq l \leq 3$. It is easy to see that

$$d'_i = \mathbf{B}_i \begin{pmatrix} \mathbf{b}_{j^i_1} \\ \mathbf{b}_{j^i_2} \\ \mathbf{b}_{j^i_3} \end{pmatrix}^{-1} \begin{pmatrix} d_{j^i_1} \\ d_{j^i_2} \\ d_{j^i_3} \end{pmatrix} + \mathbf{B}_i\mathbf{o} \qquad (1)$$

[1] Of course, an incremental hill-climbing overlap test, which saves closest features, could be applied to non-aligned DOPs. However, this would incur a lot of additional "baggage" in the data structures. In fact, [HKM96] have reported that it is still less efficient than brute-force re-alignment.

The correspondence established by j_l^i is the same for all (non-aligned) DOPs of the whole tree, if the following condition is met: DOPs must not possess any *completely redundant* half-spaces, i.e., all planes must be supported by at least one vertex of the convex hull of the DOP. (We do allow *almost redundant* half-spaces, i.e., planes which are supported by only a single vertex of the convex hull.) Fortunately, this condition is trivially met when constructing the DOP-tree.

The correspondence j_l^i is established in two steps: First, we compute the vertices of a "generic" DOP, constructed such that each vertex is supported by three planes (i.e., no degenerate vertices). In an intermediate correspondence c we store with each vertex P^i the three orientations $\mathbf{B}_{c_1^i}, \ldots, \mathbf{B}_{c_3^i}$ of the three supporting planes.[2] In the second step, another correspondence is calculated telling which P^i does actually support a plane of the new axis-aligned DOP D' (not all P's will do that). The first and the second correspondence together yield the overall correspondence j.

The first intermediate correspondence c has to be computed only once at initialization time,[3] so a brute-force algorithm can be used. The second intermediate one has to be computed whenever one of the objects has been rotated, but it is fairly easy to establish: at the beginning of each DOP-tree traversal, we transform the vertices of a generic DOP (see below) by the object's rotation. Then, we combine these to establish the final correspondence j_l^i.

To establish the intermediate correspondence c, we can choose any DOP satisfying the additional condition that all planes do support a non-degenerate face (i.e., all planes are non-redundant in the strict sense). We construct such a DOP in the following way: Start with the unit DOP $\mathbf{d} = (1, \ldots, 1)$. Then check that each plane satisfies the condition. If there is a plane which does not, increase its plane-offset. This algorithm should be made probabilistic so as to avoid cycles. Of course, it could still run into a cycle, but this has not happened so far in countless runs.

2.1.1 Brute-force alignment

There is another way to realign DOPs [KHM+98]. The idea is to represent DOPs by their vertices. Then, an aligned enclosing DOP of a tumbled DOP can be found trivially by transforming the vertices of the tumbled

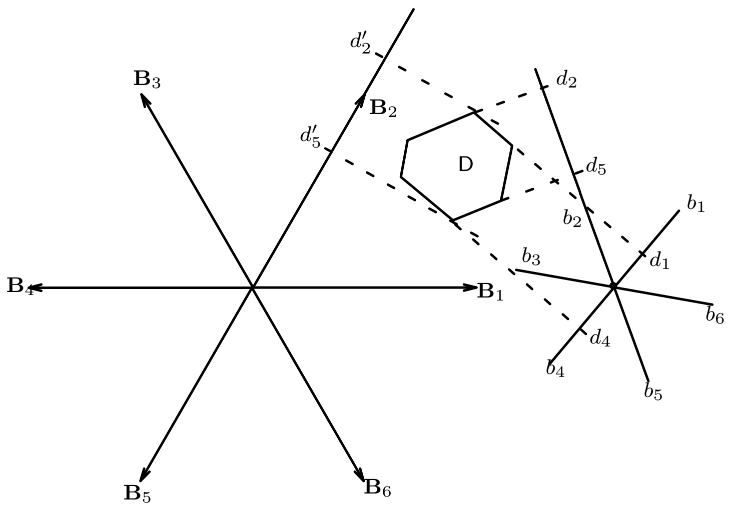


Figure 2: A rotated DOP can be enclosed by an aligned one by computing new plane offsets d'_i. Each d'_i can be computed by an affine combination of 3 $d_{j_l^i}, 1 \le l \le 3$ (2 in 2-space). The correspondence j_l^i depends only on the affine transformation of the associated object and the fixed orientations $\mathbf{B}_i$.

DOP and then computing the min/max of all vertices along each orientation.

Enclosing a non-aligned DOP by an aligned one takes

	affine trf.	vertex trf.
FLOPs	$6k$	$6k^2 + 17k$

where multiplications, additions, and comparisons count equal (the constant terms have been omitted). Both methods represent the resulting aligned DOP in "slab form". For the estimation of the brute-force method, we have assumed that DOPs consist only of triangles.

2.1.2 Comparison

Although a lot of hierarchical algorithms have been published in recent years, it has not been clear how they compare to each other. In order to optimize the collision detection pipeline, one would like to know which algorithm to use at the back end. Maybe there is no single best algorithm, and different algorithms perform best for different complexities or types of objects.

In this section, we will compare three algorithms: OBB-trees [GLM96], BV-trees of k-DOPs [KHM+98], and DOP-Trees [Zac98]. The latter two use the same type of bounding volume, but different algorithms for checking the overlap of two BVs. The former uses

[2] Because of our loose definition of *redundancy* for planes, it could happen that more than 3 planes pass through one point in space. This is no problem, though, because this just means that several vertices of the DOP will be coincident. Each of them corresponds to exactly three planes.

[3] We chose not to hard-code it, so we could experiment with different sets of orientations.

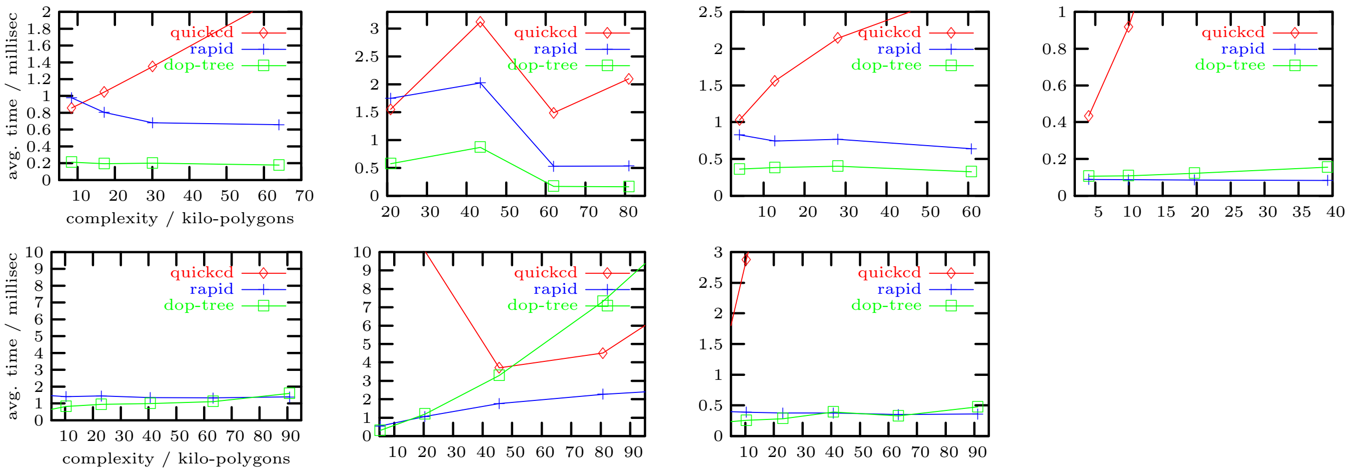


Figure 3: Results for the suite of benchmark object as shown in Figure 1 (headlight, door lock, car body, hose, sphere, hyperboloid, torus). All times have been obtained on a R10000 194MHz, averaging over distance, orientation, and object frame.

oriented bounding boxes. In contrast to [Zac98], we have determined 24 as the optimal number of orientations. For the comparison, we used the implementations Rapid and QuickCD [Got97, KHM99].

We have found, that it is extremely difficult to compare collision detection algorithms, because in general they are very sensitive to conditions and scenarios, such as the relative size of the two objects, the relative position to each other, the distance, etc. Even the orientation of the object with respect to its object frame can have a significant impact on the BV-tree and hence on the efficiency [Zac00, Fig. 3.36].

Therefore, we propose a benchmark scenario which tries to neutralize all of these effects (see [Zac00, Zac01] and `http://www.igd.fhg.de/~zach/coldet/index.html`).

We have carried out extensive experiments with both synthetic and real-world CAD objects (see Figure 1). All timings include vertex and normal transforms.

Figure 3 shows the results of our benchmark suite. The QuickCD algorithm performs much worse than Rapid and our DOP-tree. In addition, it depends much more on the number of polygons. In contrast, Rapid and DOP-tree depend very little on the number of polygons, where Rapid depends less; in some cases, collision detection time even decreases slightly as the number of polygons increases. For most objects, the DOP-tree is faster, for some it is significantly faster, while Rapid is slightly faster for others.

With one object (the door lock), there is a remarkable decrease in collision detection time; we are not sure why that is, but we suspect this is because with a finer tesselation the BV-tree construction algorithms get a better chance of producing good BV hierarchies. Another reason might be that with this particular object the bulk of the polygons are interior ones.

2.2 Convex hull test

Convex hulls are much tighter bounding volumes than bounding boxes, so they can be used as one component of the neighbor-finding stage in the collision detection pipeline to further reduce the number of polygonal object-object tests. In particular, convex hulls offer a lot of nice properties which make them attractive for collision detection [LC92, CLMP95, vdB99, GJK88, CW96].

We have chosen to use a probabilistic convex hull algorithm which trades accuracy for speed. Since in our case there is always an exact collision detection check after the convex hull test, this does not introduce wrong results.

The algorithm We have developed an algorithm which just needs the set of vertices of the convex hull and its adjacency map. The algorithm is based on the notion of linear separability: P and Q do not intersect iff there is a plane h such that all vertices of P and Q are on different sides. Such a plane is called a *separating plane*. Let $P = \{p^1, \ldots, p^n\}, Q = \{q^1, \ldots, q^m\} \subseteq \mathbb{R}^3$; then, P and Q are linearly separable, iff $\exists w \in \mathbb{R}^3, w_0 \in \mathbb{R}\, \forall i,j : (p^i, -1) \cdot (w, w_0) > 0\ ,\ (-q^j, 1) \cdot (w, w_0) > 0$.

The algorithm is based on the perceptron learning rule [HKP91]. Let $Z = \{z^k\} := \{(p^i, -1), (-q^j, 1)\}$ be the set of vertices, and w^0 an initial plane. If w^l is not a separating plane, i.e., $\exists z : z \cdot w^l < 0$, then a new plane is computed by $w^{l+1} := w^l + \eta \cdot z$. η is a "temperature" which is used for simulated annealing.

The algorithm terminates when a separating plane has been found, or when the maximum number of loops has been reached.

By saving the plane w for each pair of objects, this algorithm can be turned into an incremental algorithm very easily. In addition, checking whether or not a plane is separating can be done very quickly by hill-climbing, because the two sets of vertices are known to be convex.

We have found that a maximum number of 300 loops is sufficient, i.e., the algorithm either has determined a separating plane or it will never determine one, no matter how many more loops are being performed.

Comparison with Lin-Canny We compared the separating planes algorithm with the Lin-Canny algorithm as implemented in I_collide [CLM+]. The benchmarking procedure is comprised of two spheres, one of them stationary, the other orbiting around the first with various distances. For the separating planes algorithm, the maximum number of steps was set to 300. Times are averaged over 5,000 samples for each distance. At all distances, there were only 0–3 wrong results, except for distance 2.000060 which yielded 1256 wrong results.

Figure 4 shows how the two algorithms depend on various parameters, namely distance, number of polygons, and rotational velocity.

In addition to being about twice as fast, it seems that the separating plane algorithm is much more robust than I_collide [Zac00, Sec. 3.4.3]. Maybe these problems persist because the Voronoi regions are magnified a little bit in order to avoid other problems.

2.3 Parallelization

The collision detection pipeline offers several possibilities of parallelization: pipelining, concurrency, coarse- and fine-grain parallelization. We have not yet investigated a parallel pipeline, i.e., running each stage of the pipeline concurrently to the others. Making the whole collision detection pipeline concurrent to other modules of the system is very easy: since our framework already provides a queue at the front-end, this only needs to be implemented as a double-buffer with access control.[4]

At the back-end, several pairs of objects can be checked simultaneously, using dynamic workload allocation, which we call coarse-grain parallelization. This yields very good speedups, provided there are always enough pairs passing through the neighbor-filtering stage, i.e., the environment is dense enough. Figure 5

[4] We have, of course, implemented our collision detection module concurrently, and the results are entirely satisfying.

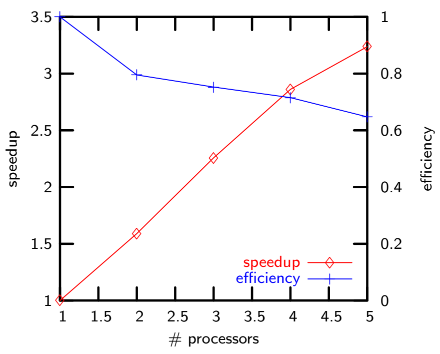


Figure 5: Coarse-grain parallelization of the back-end yields good speed-ups, if there are enough object pairs to be checked with each collision frame.

shows some timing results for a scenario where several objects are bouncing off each other inside a cube. The timing includes all stages of the collision detection pipeline, although only the back-end has been parallelized. It has been carried through on a 6-processor 194 MHz R10000 Onyx.

Sometimes, only one object pair passed through the neighbor-filter. In that case, a parallel version of the exact collision detection algorithm itself should be used. For hierarchical algorithms, a dynamic load-balancing scheme should be used, because different branches of the BV-tree need to be descended to highly different levels. The implementation must be careful, otherwise synchronization overhead will be too high.

3 Sliding contact for interactive part movement

As mentioned earlier, the sliding simulation of objects is a necessity for virtual assembly simulation. In recent years, packaging of parts in cars has become so tight, that there are many parts for which there is no assembly path such that the part never touches other parts. In fact, the part usually touches other parts over a relatively long interval along the path. And even if there would be a collision-free path — a human worker will always “create” assembly paths with touching collisions in the real world.

So, in the virtual world, we need a way to prevent parts from penetrating each other, while still allowing touching collisions such that the user can move parts in a sliding fashion along the surface of other parts.

If the VR system has no control over the user’s real hand (via force feedback), then the commonly used rigid grasping metaphor has to be changed slightly towards a less rigid one. This is a variant of grasping where the transformation from hand coordinate system to object coordinate system is no longer invariant.

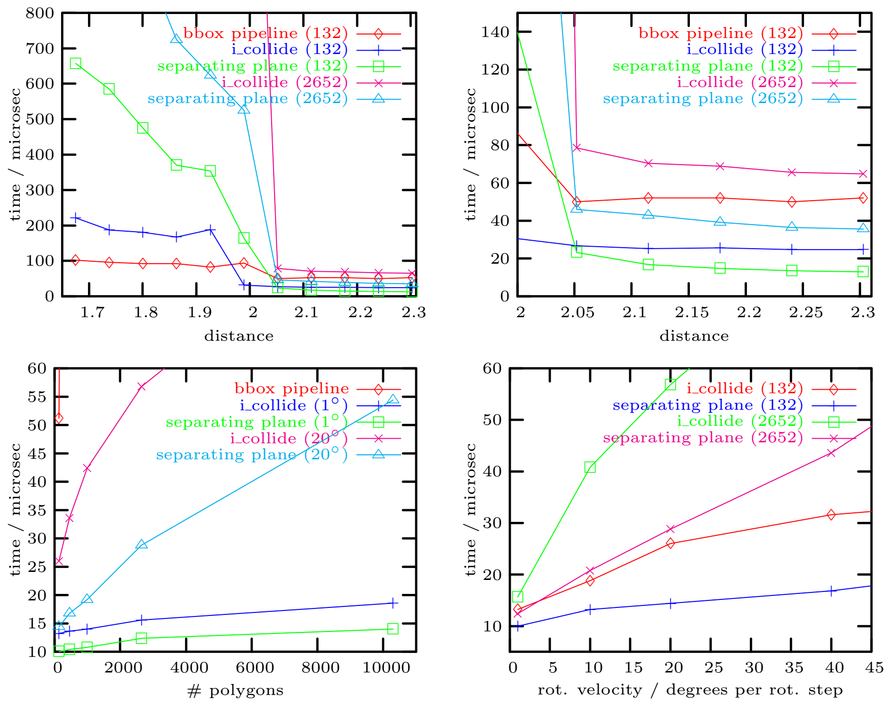


Figure 4: Comparison of I_collide and our separating planes algorithm with respect to distance, number of polygons, and rotational velocity. In the upper two graphs, the numbers in parentheses denote the number of polygons of each object. In the lower left graph, the numbers in parentheses denote rotational velocity. In the lower right graph, the numbers denote polygon count.

For the sake of clarity, let us assume that the object is collision-free at the time when it is being attached to the hand. Let us assume further that at that moment we make a copy of the object which is allowed to penetrate all other objects and which is being grasped rigidly by the hand. We will call this copy the "*ghost*" of the object. It marks the position where the object would really be if there was no collision. There are, at least, three non-rigid grasping metaphors:

- The rubber band metaphor: the object is connected to the ghost by a rubber band. This tries to pull the object as close to the ghost as possible without penetrating.

- Rubber band and spiral spring: like the plain rubber band metaphor, but the object is also connected by a spiral spring to the ghost (this is a little bit difficult to picture). In the plain rubber band metaphor, the user has no control over the orientation of the object — it is completely determined by the simulation.

- Incremental motion: when the ghost has moved by a certain delta the object will try to move about the same delta (starting from its current position). If there is a collision during that delta, then the simulation will determine a new direction. So, alternatingly the object is under the control of the user and under simulation control.

The algorithm we describe in the following allows for implementation of any of these metaphors.

If the VR system does have control over the user's real hand via force feedback, then the rigid (and more natural) grasping metaphor can be retained. Still, a simulation algorithm is needed to create forces appropriate to keep the user's hand from generating interpenetrations. The algorithm we will present below can be used to render such forces.

In the remainder of this section, we will explain that algorithm in more detail. The reader should keep in mind, that the goal was *not* to make the sliding behavior of objects as physically correct as possible. Readers interested in physically correct simulations should refer to the wealth of literature, for instance [Bar94, GVP91, Hah88, SS98, BS98].

Instead, the goal was to develop an efficient algorithm which helps the user to move the object exactly where he wants it, in minimal time, and even in closely packed environments (such as the interior of a car door).

However, we believe that our approach is more general than the one presented by [KYK98], where the approach is to constrain the motion of objects by certain faces identified through collision detection; then, the number of faces determines the number of remaining degrees of freedom (for instance, with two faces only one translational degree remains). In addition, constraining faces exhibit a certain "stickiness". The overall approach is not physically-based, but just meant as an aid for assembling simple "block-shaped" objects.

3.1 The main loop

For several reasons, the collision detection module runs concurrently in our VR system. In order to handle that, the sliding simulation is implemented as a finite state machine. This has the additional advantage, that the simulation module itself runs concurrently in our VR system. The three modules communicate with each other as depicted in Figure 7.

In the simulation, there is a so-called *collision object* (short *collobj*) which is invisible.[5] It is used to check intermediate position for collisions. The *visible object* is the one users are really seeing. It is never placed at invalid (i.e., colliding) positions. So the user only sees a valid, i.e., collision-free path of the object.

The algorithm works, simply put, as follows:

```
loop:
  while no collision
    move visible and coll.  object
    according to hand motion
  {now the coll.-object is penetrating}
  approximate exact contact point
  classify contact
  calculate new direction
```

As mentioned before, this is implemented as a finite state machine. A more detailed picture and description of that can be found in [Zac00]. There, you can also find several algorithms for fast collision detection. Figure 6 shows that in a little bit more detail.

In our algorithm, the contact approximation is done by interval bisection and a number of static collision checks. [ES99] propose a dynamic collision detection algorithm. However, it is not clear that this would really speed up the simulation in this case, since the dynamic algorithm takes about 3–5 times longer and an exact contact point is usually not needed here.

3.2 New directions

Let us assume that we have the exact contact position. Let us further assume that we need to handle only practically relevant contact situations. Then we will need to deal only with the following cases: 1 contact point, 2 contact points, and ≥ 3 contact points, which we will discuss in the following.

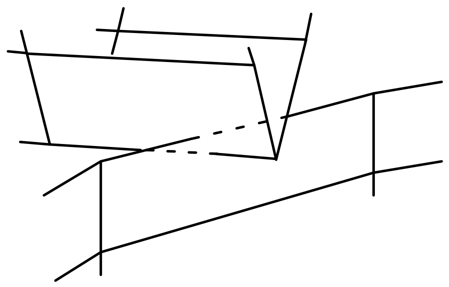


Figure 8: Contact classification can be done by looking at adjacent polygons and counting edge/face intersections.

In each case, we must be able to deal with "wrong" surface normals. In general, polygonal geometry imported from CAD programs has "random" surface normals in the sense that the vertex order is *not* consistent across adjacent polygons. But even if it were, we would have to be able to deal with such a situation, because unclosed objects (like sheet metal) does not have "inside" and "outside". With such "sheet objects" we might be colliding from either side.

Our implementation of the sliding algorithm presented here allows for arbitrarily pointing normals. They can even be "inconsistent" in the sense that adjacent polygons' normals can point on different side.

In order to be able to compute new forces, each contact point must be classified. In theory, we need to handle only two contact situations: vertex/face and edge/edge. Given one pair of touching polygons (p, q), we can determine the contact situation by the following simple procedure: let n_p be the number of polygons adjacent to p and touching q; define n_q analogously. If $n_p = n_q = 1$, then we have the edge/edge case. Otherwise, either n_p or n_q must be > 1 (but not both), and we have the vertex/face (or face/vertex) case.

In practice, a few more cases can happen. Partly, this is due to the mere approximation of the contact position, partly, it is due to non-closed geometry.[6]

If $n_p = 0$, for instance, then this polygon might be at the rim of the object. In order to decide that, we need to check if any edge of p intersects q (see Figure 8). If so, we have got the edge/edge situation. If not, then it is the vertex/face situation. Note that both $n_p = n_q = 0$ is possible. It is not sufficient to just check edge/face intersections of the two intersecting polygons (counter-example: two intersecting wedges).

[5] The geometry of the collision object is, usually, exactly the same as that of the visible object. If the scene graph API allows for it, they can share their geometry.

[6] Non-closed geometry is fairly frequent in virtual prototyping: all sheet metal is non-closed. Now imagine the possible contact situations when a pipe is to be fitted into a hole of sheet metal.

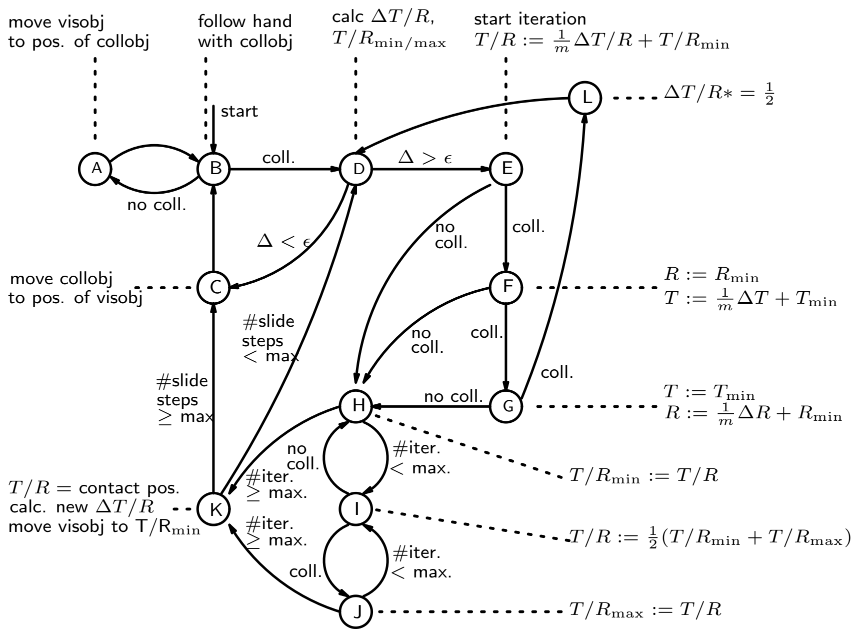

move visobj to pos. of collobj
follow hand with collobj
calc $\Delta T/R$, $T/R_{\min/\max}$
start iteration $T/R := \frac{1}{m}\Delta T/R + T/R_{\min}$
$\Delta T/R* = \frac{1}{2}$
start
coll.
$\Delta > \epsilon$
no coll.
$\Delta < \epsilon$
no coll.
coll.
move collobj to pos. of visobj
$R := R_{\min}$
$T := \frac{1}{m}\Delta T + T_{\min}$
#slide steps < max
no coll.
coll.
coll.
#slide steps ≥ max
no coll.
$T := T_{\min}$
$R := \frac{1}{m}\Delta R + R_{\min}$
no coll.
#iter. < max.
#iter. ≥ max.
$T/R_{\min} := T/R$
T/R = contact pos. calc. new $\Delta T/R$ move visobj to $T/R_{\min}$
#iter. ≥ max.
$T/R := \frac{1}{2}(T/R_{\min} + T/R_{\max})$
#iter. < max.
coll.
$T/R_{\max} := T/R$
A
B
C
D
E
F
G
H
I
J
K
L

Figure 6: The main loop of the sliding simulation is a finite state machine.

When all contact points have been classified, we can compute new forces and velocities. See [Zac00] for the mathematical details. Depending on the kind of metaphor we have chosen (see above), we must then evaluate a stopping criterion before iterating the next sliding cycle. Actually, the kind of grasping metaphor is determined by the stopping criterion. Our criterion is a combination of several sub-criteria involving the number of iterations so far and different distance measures. See [Zac00] for more details.

4 Natural grasping

Grasping objects is one of the most fundamental interaction techniques in VEs, in particular in virtual assembly simulation, which comes as no surprise since it is also one of the most frequent activities in the real world.

Previous algorithms include [KH95], which is based on the notion of a "finger-tip triangle", and [BRT96], in which a simple automaton is associated with each finger.

In order to be able to make true verifications of assemblability and serviceability, it is important, that the virtual hand grasps virtual parts just like the real hand would grasp their real counterparts. So, the VR system must make sure that the virtual fingers never penetrate parts, but still allows them to close tight around them (see Figure 9). In addition, in order to make virtual grasping natural, the VR system has to determine when an object is grabbed firmly. So, the user would not need to remember a command, and objects cannot be grabbed by the back of the hand.

Like with force-feedback devices, we need to distinguish between (at least) four types of grasping:

1. *Precision grasping* with three sub-types [Jon97]: tip pinch, three-jaw chuck, and key grasp,
2. *cigarette grasping*,
3. *3-point pinch grasping*,
4. *Power grasping* (or just *grasping*),
5. *Gravity grasping* (or *cradling*).

Precision grasping involves 2 fingers, usually the thumb and one of the other fingers; it is used for instance to grasp a screw. Cigarette grasping involves two neighboring fingers; it is usually used to "park" long thin objects, such as a cigarette or pencil. 3-point grasping involves three fingers (one of them being the thumb), giving the user a fairly firm grip, and allowing him to rotate the object without rotating the hand. Power grasping involves the whole hand, in particular the palm. With this type of grasp the object is stationary relative to the hand. Gravity grasping is actually a way of carrying an object.

For power grasping, the algorithm consists of two simple parts: clasping the fingers around the object and analyzing the contact. The former will be done by an iteration (see Figure 10), while the latter is implemented by a simple heuristic.

The position of the hand is completely specified by (M, F), where M is a matrix specifying the position of the hand root, and F is the *flex vector* (usually 22-dimensional). Given a new target hand position (M^n, F^n), the goal is to minimize $(|MM^{n-1}|, |F -$

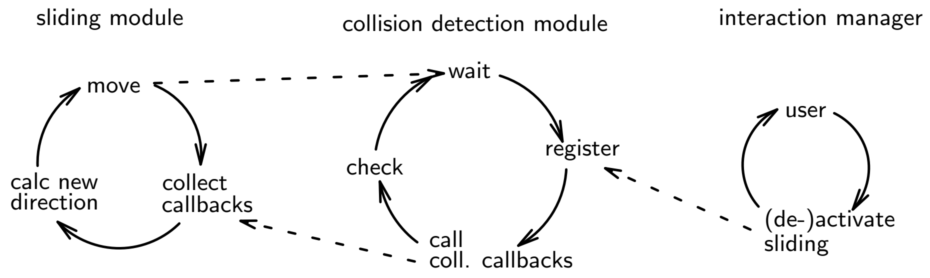


Figure 7: The physically-based simulation module for sliding runs concurrently to the other two main loops of the collision detection module and the interaction manager. Dashed arrows mark rendezvous points.

$F^n|$) such that (M, F) is collision-free. Note that the position of a finger-joint depends on its flex value and all flex values higher up in the chain and the position of the hand root. Therefore, we suspect that there are several local minima, even if we only consider flex values during optimization (and keep the position fixed). However, this should not be a problem if the minimization process is fast enough, so that consecutive collision-free hand positions are not too "distant" from each other.

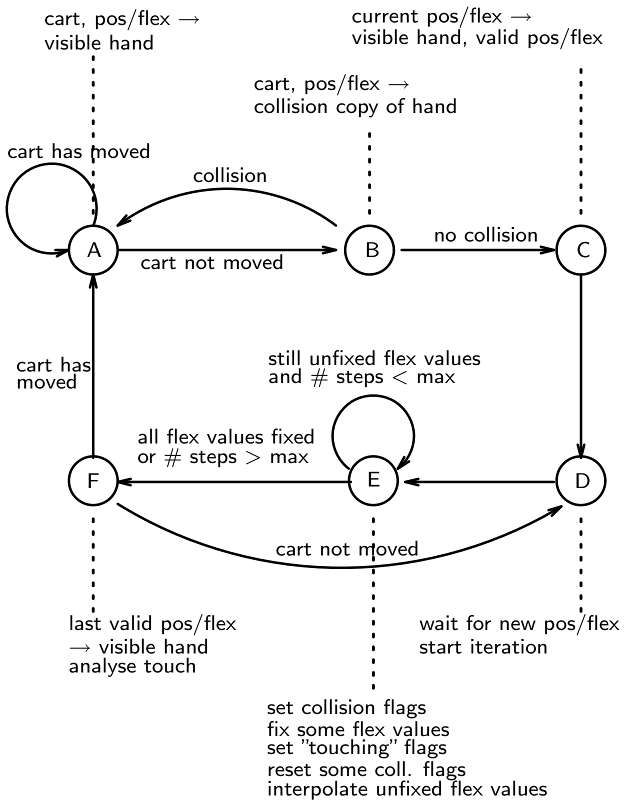
Minimization must not be done using the visible model of the hand; otherwise, the user would "witness" the process (because the renderer runs concurrently). So, a copy of the hand tree is used for collision detection, and only after minimization has finished, the new position/flex values are copied to the visible hand. This minimization should be as fast as possible, so it runs as a concurrent process in our VR system; otherwise, the user might notice considerable latency.

In order to find an optimal flex vector, our algorithm uses an iteration process interpolating non-colliding (i.e., valid), and colliding (i.e., invalid), flex values. Here, a flex value is *colliding* if its associated finger-joint is colliding *or* any finger-joint depending on it. A finger-joint J' is *depending* on a finger-joint J, if it is further down the kinematic chain, i.e., if J moves, then J' moves, too (see [Zac00] for a detailed depiction of the finite state machine of this process). Note that a finger-joint can have many depending finger-joints. (In this context, the palm is a "finger-joint" like all the others.)

During the iteration process, the position M of the hand is treated like any other flex value, i.e., it is interpolated. The only differences are that interpolation is done on matrices instead of single real numbers. The "joint" associated with it is usually the palm or the forearm.

After a few iteration steps, some flex values will be approximated "close enough" (when the range between valid and invalid flex value is small enough). Then, they will be *fixed*. Depending flex values must be considered for fixing, too: they may or may not be close enough. So, several flex values in a row may become fixed at the same time. As long as a flex value is not fixed, it will be interpolated *and* all its depending flex values.[7]

During iteration, the algorithm has to mark all finger-joints (including the palm) which are *touching* the object. If a finger-joint is not touching the object, we will call it *free*. At the beginning of the iteration, all finger-joints are free. When a finger-joint collides, the algorithm sets a collision flag for it. When that flex



cart, pos/flex → visible hand
current pos/flex → visible hand, valid pos/flex
cart, pos/flex → collision copy of hand
cart has moved
collision
no collision
cart not moved
A
B
C
cart has moved
still unfixed flex values and # steps < max
all flex values fixed or # steps > max
F
E
D
cart not moved
last valid pos/flex → visible hand analyse touch
wait for new pos/flex start iteration
set collision flags
fix some flex values
set "touching" flags
reset some coll. flags
interpolate unfixed flex values

Figure 10: A simplified overview of the algorithm for simulating natural grasping.

[7] An alternative would be not to interpolate depending flex values, but since depending finger-joints need to be checked for collision anyway, we can as well interpolate them, too. Thus, we probably achieve an optimum faster.

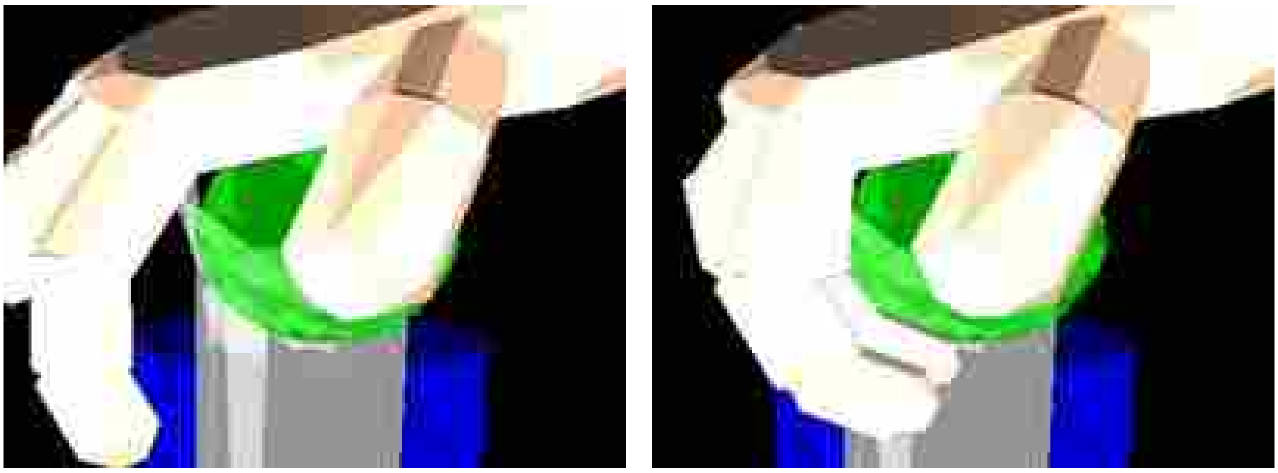


Figure 9: Natural grasping is basically a minimization problem for the flex vector under the constraint that finger-joints (and palm) must not penetrate the object.

value gets fixed, the finger-joint is marked as "touching" if the collision flag is set (the collision may have happened several iterations earlier). After a flex value has been fixed, the collision flag of all depending flex values will be cleared again. This is because possible collisions of depending finger-joints are due to motions of up-chain finger-joints and not because the depending finger-joint is touching.

After all flex values have been fixed, the second phase of the algorithm tries to analyze the type of grasp. While the grasping algorithm is in general applicable to any hierarchical kinematic chain, the analysis algorithm needs to know more about its "semantics", i.e., it has to know about a palm, it needs to know which finger-joints belong to the same finger, etc. The heuristic we have implemented is very simple:

1. only one finger-joint or palm is touching → push;
2. several finger-joints are touching, and none of them is part of the thumb, and the palm is not touching → push;

   This part of the heuristic would need to be more sophisticated if cigarette grasping should be recognized. However, this type of grasp is not needed for virtual assembly simulation.
3. one or more finger-joints is touching, one or more thumb-joints is touching, and the palm is not touching → precision grasp;
4. one or more finger-joints (possibly a thumb joint) and the palm are touching, and at least one of the finger-joints is a middle or outer joint → power grasp;
5. one or more finger-joints and the palm are touching, but all finger-joints are inner joints → push.

In our algorithm, motion of the cart is handled specially (see [Zac00] for more details). A cart motion indicates that the user's (virtual) body is changing place. Since the hand is attached below the cart, a cart motion always brings on a motion of the hand. In that case, the algorithm does not try to clasp the hand tightly around an object, because that might cause the hand to be left behind, which is probably not what the user wanted. Unfortunately, navigation might cause the hand to end up in an invalid (i.e., colliding) place, so after navigation has stopped, the clasping algorithm cannot begin until the whole hand has been moved to a collision-free place by the user.

5 Conclusion

In this paper, we have presented several algorithms which tackle various prerequisites for a solution to natural hand-object interactions in virtual environments.

One of the basic requirements is fast collision detection. In this area, we have described two algorithms: the first one solves the object-object polygonal intersection test efficiently by a DOP tree, while the second one is an efficient pre-check in the neighbor-finding stage of the collision detection pipeline exploiting properties of the convex hull and temporal coherence.

The convex algorithm depends sub-linearly on polygon count and rotational velocity with very small hidden constants. The algorithm gains additional efficiency by hill-climbing on the convex hull and by maintaining a separating plane in two different coordinate frames simultaneously. Collision detection time is of the order of microseconds when objects do not intersect. A comparison with the renowned Lin-Canny algorithm showed that the algorithm presented in this work is about 2 times faster. In addition, it seems to be more robust numerically.

The DOP tree algorithm gains its speed by an elegant way to enclose non-axis-aligned DOPs by axis-aligned ones which is in $O(k)$ while the previously proposed method is in $O(k^2)$ (k being the number of orientations). This algorithm can check a pair of objects of 50,000 polygons each within $\frac{1}{2}$–4 milliseconds.

Collision detection lends itself well to parallelization. Our collision detection module features concurrency, coarse-grain, and fine-grain parallelization. Experiments demonstrate the efficiency of the implementation.

Another requirement for natural interaction is that objects behave naturally and somehow "slide" along the surface rather than penetrating it. This kind of behavior has been implemented by a physically-based algorithm. It does not try to be physically correct, but to be as fast as possible while still providing intuitive and plausible behavior. Experiments have shown that our algorithm needs about 300 microseconds per contact point (not counting collision detection time), and it works well in practice.

Finally, we have described a general algorithm for solving the problem of natural grasping and manipulation of virtual objects with a virtual hand. This enables a user to grasp objects just like in the real world without having to resort to abstract gestures. The algorithm presses the virtual hand's fingers to an object by a minimization procedure, and then grasps it based on the analysis of the contact.

6 Future Work

In order to make the sliding simulation and natural grasping even more efficient, collision detection algorithms should provide some measure of the amount of interpenetration. This information could be used to estimate the contact points more quickly.

A particularly difficult problem for the sliding algorithm is presented by the way slide-in units are designed (for instance, car radios): these parts and their compartments are usually designed such that their respective hulls overlap precisely, so that the virtual parts will actually have a collision when in final position.

Force feedback is currently an active area of research. For simple cases, algorithms and devices are available, but as of yet, there is no device suitable for virtual assembly simulation. However, force feedback would greatly increase intuitivity and user efficiency as well as the transferability of the results obtained in interactive simulations.

Finally, it would be necessary to represent the user's hand by a deformable virtual hand. This would also include a more sophisticated model of the skin of the user's hand. In addition, efficient collision detection algorithms for deformable models are needed.

A different problem, but related to natural interaction in VEs, is tracking the user's real hand and fingers. Until now, the user is still tethered by these tracking devices. In order to increase acceptance of VR in general, it seems necessary to us to be able to precisely solve this tracking problem while allowing the user to remain untethered.

References

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Andrew Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, pages 23–34. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0. 6

[BRT96] Ronan Boulic, Serge Rezzonico, and Daniel Thalmann. Multi-finger manipulation of virtual objects. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST '96)*, pages 67–74. Hong Kong, July1-4 1996. 8

[BS98] Matthias Buck, and Elmar Schömer. Interactive rigid body menipulation with obstacle contacts. *The Journal of Visuazlization and Computer Animation*, 9:243–257, 1998. 6

[CLM+] John Cohen, Ming C. Lin, Dinesh Manocha, Brian Mirtich, M. K. Ponamgi, and John Canny. I_COLLIDE. URL `http://www.cs.unc.edu/~geom/I_COLLIDE.html`. Software. 5

[CLMP95] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *1995 Symposium on Interactive 3D Graphics*, Pat Hanrahan and Jim Winget, Eds., pages 189–196. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7. 4

[CW96] Kelvin Chung, and Wenping Wang. Quick collision detection of polytopes in virtual environments. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST'96)*, Mark Green, Ed., pages 125–131, July 1996. 4

[ES99] Jens Eckstein, and Elmar Schömer. Dynamic collision detection in virtual reality applications. In *Proc. The 7-th International Conference in Central Europe on Computer Graphics, Visualization, and Interactive Digital Media '99 (WSCG'99)*, pages 71–78. University of West Bohemia, Plzen, Czech Republic, February 1999. 7

[GJK88] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects. *Internat. J. Robot. Autom.*, 4(2): 193–203, 1988. 4

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *SIGGRAPH 96 Conference*

Proceedings, Holly Rushmeier, Ed., Annual Conference Series, pages 171–180. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996. 2, 3

[Got97] Stefan Gottschalk. Rapid library, 1997. URL `http://www.cs.unc.edu/~geom/OBB/OBBT.html`. Vers. 2.01. 4

[GVP91] Marie-Paule Gascuel, Anne Verroust, and Claude Puech. Animation and collisions between complex deformable bodies. In *Proceedings of Graphics Interface '91*, pages 263–270, June 1991. 6

[Hah88] James K. Hahn. Realistic animation of rigid bodies. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, John Dill, Ed., vol. 22, pages 299–308, August 1988. 6

[HKM96] Martin Held, James T. Klosowski, and Joseph S.B. Mitchell. Real-time collision detection for motion simulation within complex environments. In *Siggraph 1996 Technical Sketches, Visual Proceedings*, page 151. New Orleans, August 1996. 2

[HKP91] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computing*. Addison-Wesley, 1991. 4

[Hub95] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, September 1995. ISSN 1077-2626. 2

[JJWT99] Sankar Jayaram, Uma Jayaram, Yong Wang, and Hrishikesh Tirumal. VADE: A virtual assembly design environment. *IEEE Computer Graphics & Applications*, 19(6):44–50, November, December 1999. 1

[Jon97] Lynette Jones. Dextrous hands: Human, prosthetic, and robotic. *Presence*, 6(1):29–56, February 1997. 8

[KH95] R. Kijima, and M. Hirose. Fine object manipulation in virtual environments. In *Virtual Environments '95*, M. Göbel, Ed., Eurographics, pages 42–58. Springer-Verlag Wien New York, 1995. Proc's Eurographics Workshop, Barcelona, Spain, 1993, and Monte Carlo, Monaco, 1995. 8

[KHM$^+$98] James T. Klosowski, Martin Held, Jospeh S.B. Mitchell, Henry Sowrizal, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998. 3

[KHM99] Jim Klosowski, Martin Held, and Joe Mitchell. QuickCD, software library for efficient collision detection, 1999. URL `http://www.ams.sunysb.edu/~jklosow/quickcd/QuickCD.html`. Vers. 1.00. 4

[KYK98] Yoshifumi Kitamura, Amy Yee, and Fumio Kishino. A sophisticated manipulation aid in a virtual environment using dynamic constraints among object faces. *Presence*, 7(5):460–477, October 1998. 6

[LC92] Ming C. Lin, and John F. Canny. Efficient collision detection for animation, September 1992. 4

[OD99] Carol O'Sullivan, and John Dingliana. Real-time collision detection and response using sphere-trees. In *15th Spring Conference on Computer Graphics*, pages 83–92. Budmerice, Slovakia, April 1999. ISBN 80-223-1357-2. 2

[SS98] Jörg Sauer, and Elmar Schömer. A constraint-based approach to rigid body dynamics for virtual reality applications. In *Proc. VRST '98*, pages 153–161. ACM, Taipei, Taiwan, November 1998. 6

[vdB99] Gino Johannes Apolonia van den Bergen. *Collision Detection in Interactive 3D Computer Animation*. PhD dissertation, Eindhoven University of Technology, 1999. 4

[Zac97] Gabriel Zachmann. Real-time and exact collision detection for interactive virtual prototyping. In *Proc. of the 1997 ASME Design Engineering Technical Conferences*. Sacramento, California, September 1997. Paper no. CIE-4306. 2

[Zac98] Gabriel Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*, pages 90–97. Atlanta, Georgia, March 1998. 3, 4

[Zac99] Antonino Gomes de Sá, and Gabriel Zachmann. Virtual reality as a tool for verification of assembly and maintenance processes. *Computers & Graphics*, 23(3):389 –403, 1999. 1

[Zac00] Gabriel Zachmann. *Virtual Reality in Assembly Simulation — Collision Detection, Simulation Algorithms, and Interaction Techniques*. PhD dissertation, Darmstadt University of Technology, Germany, Department of Computer Science, May 2000. URL `http://web.informatik.uni-bonn.de/~zach/papers/diss.html`. Fraunhofer IRB Verlag, ISBN 3-8167-5628-X; also: `http://www.geocities.com/gabriel_zachmann/`. 2, 4, 5, 7, 8, 9, 10

[Zac01] Gabriel Zachmann. Optimizing the collision detection pipeline. In *Proc. of the First International Game Technology Conference (GTEC)*, January 2001. 1, 2, 4

[ZR01] Gabriel Zachmann, and Alexander Rettig. Natural and robust interaction in virtual assembly simulation. In *Eighth ISPE International Conference on Concurrent Engineering: Research and Applications (ISPE/CE2001)*. West Coast Anaheim Hotel, California, USA, July 2001. 1