EG UK Theory and Practice of Computer Graphics (2008)
Ik Soo Lim, Wen Tang (Editors)

A Benchmarking Framework for Static Collision Detection

Engin Deniz Diktas and Ali Vahit Sahiner

Computer Engineering Department, Bogazici University, Istanbul/Turkey †

Abstract

Performance of static collision detection queries depends on the type of the hierarchy chosen as well as the relative positioning of the colliding objects. In order to evaluate the performance of bounding volume hierarchies, relevant criteria that affect the query performance need to be determined and the sample space should be generated accordingly. In this paper we present a benchmarking framework for evaluating the performance of various static collision detection algorithms. In this framework, instances of a moving rigid object are placed on the surface of another instance of the same object fixed at a certain position, where the contact occurs for the first time. Then by offsetting the surface inwards (outwards) we generate new surfaces that are at a certain fixed negative (positive) distance to the original surface. Placing the moving object on these offset surfaces makes the object penetrate (approach) the fixed object at a fixed distance. For offset surface generation we create a signed distance field and run marching cubes algorithm on it.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1. Introduction

Collision detection methods play an important role in virtual reality applications, real-time simulations and computer games. It is a well-known fact that no single method gives best results for all cases. Therefore benchmarking different collision detection methods is important in deciding which type of method is best suited for a particular application.

A benchmarking algorithm should be both comprehensive (it should include a broad spectrum of contact scenarios) and unbiased (it should not be sensitive to specific scenarios). To achieve this we have to identify which parameters play important role in the performance of a collision detection algorithm. One such important parameter is the distance of one object to the other in cases where the objects do not collide. The reason for this is that although collision will not be detected, the algorithm still spends time for checking intersections between bounding volumes at certain depths, if the objects are close enough to each other. In fact for decreasing positive distance values, all collision detection algorithms will spend more and more time due to increased number of intersecting bounding volume primitive pairs.

Another important parameter is the amount of penetration of the moving object into the fixed one. Since static collision detection algorithms check for collision at a certain time instant, the objects might be at any general position with respect to each other. Therefore they might be penetrating each other by an arbitrary amount, especially in a scene where objects move with large relative velocities. If the penetration of objects is tolerable, then there is also a need to evaluate the performance of static collision detection methods in cases where objects penetrate each other.

In this paper, we present a framework for generating positions of a moving object at a certain fixed penetration and distance with respect to another object. When generating these positions we first generate a surface offset from the original object-surface. Then we find the first time of contact between this surface and the moving object by performing a dynamic (continous) collision detection query using a sphere-tree fitted to the object. To generate an offset surface we construct a signed distance field and run the marching cubes algorithm.

The paper is structured as follows: Section 2 reviews related work. Section 3 explains the details of our benchmarking framework by explaining how the penetration depth is fixed how the offset surfaces are generated and how the

† This work is supported by B.U. Research Fund BAP 07A104

search space is sampled. Section 4 gives the results of collision detection queries using the positioning produced by our framework. Finally Section 5 summarizes our benchmark framework and intended future work.

2. Related work

Although there has been significant work for developing collision detection algorithms, less emphasis is placed on their systematic evaluation. Many authors test their collision detection algorithms for specific scenarios. One of the earliest frameworks was developed by [Zac98], but since during the collisions objects penetrate considerably, this framework does not guarantee relevant results. In [OL03] a set of specific scenarios (a torus falling down a spiral peg, a spoon in a cup and a soup of numbers in a bowl) are used to test the proposed collision detection algorithms. In [vdB97] a pair of models are positioned randomly inside a cube, where the probability of intersection is controlled by the size of the cube. The problem with this type of testing method is again that there is no systematic way of distinguishing the different relative positions of objects. In [CRM02] the authors use different scenes in a probabilistic motion planner for testing the collision detection queries. The problem with this approach is that this benchmarking method handles only scenarios in the realm of motion planning.

The most recent work for a systematic benchmark is given in [TWZ07], where the proposed method generates a number of positions and orientations for a predefined distance. The distance computation algorithm utilizes binary search.

3. Benchmarking method

In our framework the aim is to evaluate the effectiveness of the bounding volume tree and how it is traversed. Therefore we do consider only pairwise collision detection scenarios and related algorithms. The performance of a collision detection algorithm depends mainly on the following factors:

- Relative orientation of the objects
- Relative position: Distance or penetration
- Topology of objects in close proximity
- Relative sizes of the objects
- Type of the bounding volume

In order to measure the effects of a particular factor, we need to fix the contribution of the remaining ones.

3.1. Search Space

An object has 3 translational and 3 rotational degrees of freedom (dof), the total dof of the moving object with respect to the fixed object is 6. Therefore benchmarking requires sampling of a 6-dimensional search space. Since this space is very large, we need to reduce its size by employing a clever sampling strategy that selects most relevant positions.

The set of relevant sample points might change from one scenario to another. But two criteria common to many scenarios are the *distance* between non-colliding objects and the *penetration* for intersecting ones. For non-colliding but sufficiently close objects, the collision detection algorithm spends more and more time as the distance between the two objects decreases. This is because the overlapping bounding volume primitives at nodes close to the root penetrate each other more and more, causing more nodes to be tested for intersection.

For colliding objects the amount of penetration determines the number of bounding volume primitives tested. If the penetration is zero, i.e. if the object just touches the surface of the other object, then the collision test will reach a pair of leaves each of which contain a triangle intersecting the other one. As the penetration depth increases we expect the collision test to take less and less time, since the probability of detecting a pair of intersecting triangles early during the tree traversal is high. This is due to the increased number of intersecting bounding volume primitive-pairs and triangle-pairs.

The query performance is not affected only by the penetration depth (distance), but also by the relative geometrical properties in close proximity, such as surface curvature, topology etc.

A good example of this is the well known benchmark scenario where a sphere falls through the center of the torus. As mentioned in [Got00], this corresponds to the *parallel close proximity scenario*, whereas the same sphere approaching the outer ring of the torus would constitute a *transverse close proximity scenario*. Gottschalk points out that in parallel close proximity cases OBBs outperform other types of bounding volumes. In our benchmarking framework, fixing the distance of the sphere to the torus and moving the sphere at certain locations allow us to analyze the performance due to the topology in close proximity, isolating the effects of distance.

3.2. Penetration depth computation

Computation of the penetration depth for two intersecting bodies is not as straightforward as it seems. Penetration depth is a directional property, it depends on *how* the moving object penetrates the fixed one. To visualize this consider the situation depicted in Figure 1. Assume two spherical bullets, one shot from left and the other one shot from the top into a rectangular block. Also assume that the two bullets have been shot with such initial velocities, that they end up at the same location inside a wooden block. In such a case the amount of penetration is larger for the bullet shot from the left compared to the other one shot from the top. In our case we only consider snapshots of the moving object at certain instants, and we do not know or assume anything related to how the object ended up at its final location at that specific

Figure 1: *Penetration depth depends on direction.*

instant. Therefore we either need to consider every possibility how the object ended up at that location or make an assumption about how the object approaches its final location.

Our assumption is that the object reaches its final location moving along the *normal* at the point P on the original surface. Point P is the point closest to the contact point C on the offset surface as shown in Figure 2. Since offset surfaces are obtained by shifting all points on the original surface along the negative normal direction at that point, the vector connecting the contact point on the offset surface to the closest point on the original surface will be collinear with the normal direction. So the assumption that the penetrating object reaching the contact point on the offset surface along that normal direction leads to a consistent measure of penetration depth. In Figure 2 below, the penetration depth is equal to the length $|CP|$. Note also that the vector $\overrightarrow{CP}$ is collinear with the normal vector $\vec{n}$ at the point P and this is the direction along which the object is assumed to move.

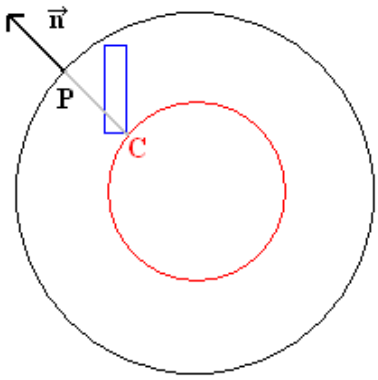

Figure 2: *Penetration depth is equal to* $|\overrightarrow{CP}|$.

3.3. Generating offset surfaces

There are different methods for generating offset surfaces. In our framework we construct an accurate signed distance map for the original object. In this map, the iso-value 0 corresponds to the original surface.

For given positive (outward-offset) / negative (inward-offset) values we extract the iso-surfaces by employing the well known marching cubes algorithm [LC87]. Since an iso-surface has the same value value throughout the whole distance map, all points on that offset surface have the same minimum distance to the original surface. Placing the moving object on the surface of this offset-surface so that it just touches it, corresponds to a penetration depth equal to the iso-value.

One short-coming of using signed distance fields for constructing the offset surfaces is that the objects must be closed 2-manifolds and objects must also be smooth enough to be able to construct well-behaving distance fields. Figure 3 shows examples of the extracted surfaces offset: the original surface is shown in the middle, the first top two surfaces are obtained by offsetting the original one inwards, and the bottom two surfaces by offsetting it outwards.

3.4. Positioning the moving object

Having generated the offset surfaces, the next question is how to place the moving object on an offset surface. One idea is to place one of its extreme vertices on any point on the offset surface. But there may be intersections at locations other than the selected point of contact and identifying such cases is extremely time-consuming.

Finding locations of the moving object that contact the offset surface only at a single point is essentially the problem of dynamic collision detection. In dynamic collision detection the aim is to find the first time and contact of two moving objects during a finite or infinite time interval. If we assume the moving object to be approaching the fixed one with a constant linear velocity vector, dynamic collision detection query provides us the first time and point of contact.

For this purpose we use sphere-trees, which enables us to use ray-casting. Here the relative velocity of the second object with respect to the first one is computed first. With this transformation, the first object can be considered to be stationary in its own frame of reference while the second one moves with a fixed linear relative velocity. In such a case, the second sphere produces a cylindrical surface as a result of sweeping its surface along the relative velocity direction.

Using Minkowski sums we can reduce the moving object to a single point by growing the stationary one. Then the problem in object space is reduced to the problem of finding the intersection of a ray with a sphere in the Minkowski space. This is shown in Figure 4. Note that in this figure the ray starts at point C_2 and the first intersection of that ray with the grown sphere is C_2'. If the direction vector of the ray, i.e. the velocity of the moving sphere, is of unit length, then the length $|C_2C_2'|$ gives the first time the two spheres intersect.

For dynamic collision detection we initialize a global variable T_f for keeping track of the first time of collision to the *last* time of collision of the root spheres and employ the following recursive rule:

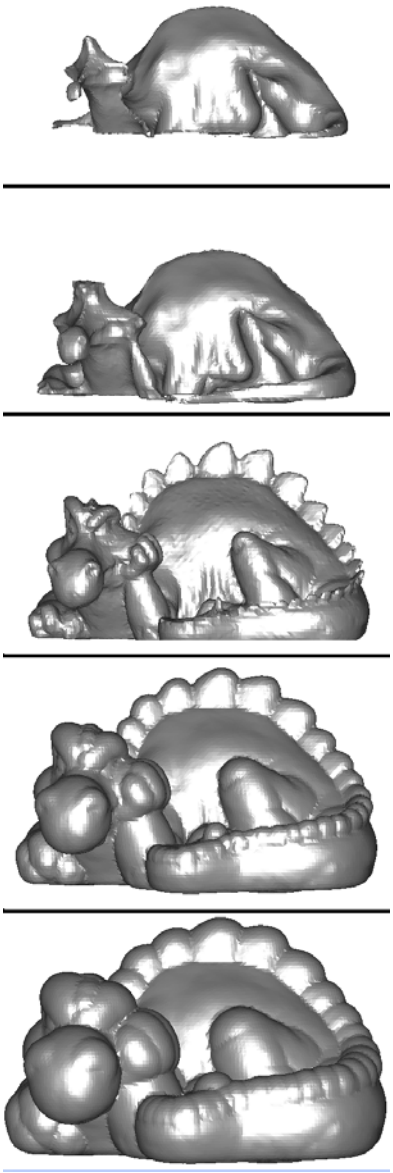

Figure 3: *Surfaces obtained by offsetting the object inwards and outwards.*

GetFirstTimeOfCollision(N_1, N_2)
IF FirstTime(BS_{N_1}, BS_{N_2}, V_{rel}) > T_f then return
IF N_1 and N_2 are parents.

Set the node with smaller sphere to be leaf

ELSE IF N_1 or N_2 is parent

Let the node parent node be M_1, the other one be M_2

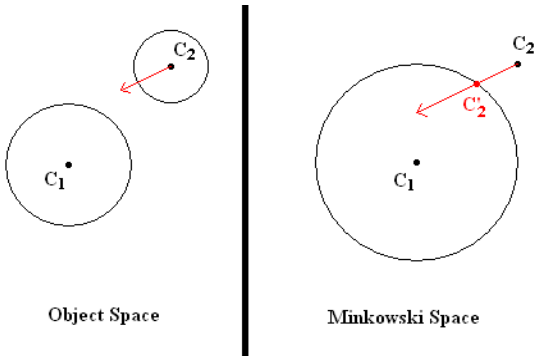

Figure 4: *Finding the first time of intersection of two spheres.*

Let the children of M_1 be C_1 and C_2
Compute T_1 = FirstTime(BS_{C_1}, BS_{M_2}, V_{rel})
Compute T_2 = FirstTime(BS_{C_2}, BS_{M_2}, V_{rel})
IF $T_1 < T_2$

Call GetFirstTimeOfCollision(C_1, M_2)
Call GetFirstTimeOfCollision(C_2, M_2)

ELSE

Call GetFirtTimeOfCollision(C_2, M_2)
Call GetFirtTimeOfCollision(C_1, M_2)

ELSE IF N_1 and N_2 are leaves

Let T_i and Tr_j be triangles assigned to the nodes N_1 and N_2, respectively
FOR each triangle pair (Tr_i, Tr_j)

T_f = min(T_f, FirstTime(Tr_i, Tr_j, V_{rel}))

where BS_{N_i} is the bounding sphere of the i^{th} node, FirstTime(BS_{N_1}, BS_{N_2}, V_{rel}) is the first time of collision of the bounding spheres with the second one approaching the first one with the relative velocity V_{rel}, and FirstTime(Tr_i, Tr_j, V_{rel}) is the first time of collision of the triangles Tr_i and Tr_j with the second one approaching the first one with the relative velocity V_{rel}.

Figure 5 shows the dynamic collision detection algorithm applied to two spherical objects: the red sphere is fixed, while the blue one moves along a line connecting its center with the center of the fixed one. The instants shown are those instants when the algorithm updates the variable T_f. At every such instant the pair of contacting triangles are also shown as filled triangles. Note that the last instant is the output of the algorithm. Figure 6 shows the locations (green points in the figure) of the center of the second sphere at every update instant.

When generating the contact points on the offset surfaces we proceed as follows: we place both objects with their origins placed at the origin of the global frame of reference.

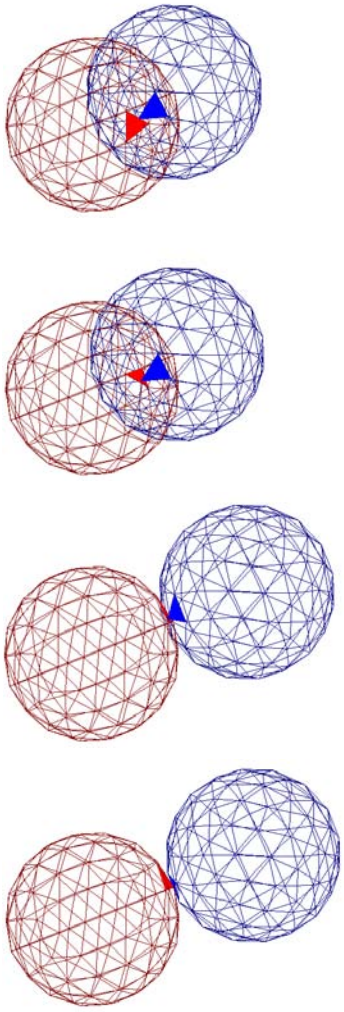

Figure 5: *Instants during dynamic collision query.*

Then we translate the second object along a certain direction until there is no intersection between their bounding volumes at the root level. Then we assign to the second object the negative of that direction as the approach velocity. Then we perform a dynamic collision detection query to find out the exact time of first contact with the foregoing parameters. For generating the set of directions we sample the unit dual sphere with a regular grid, although a stochastic method could be used as well. The result of this process is shown on an example in Figure 7 for the original object moving on the surface of an outward-offset surface. First contact occurs in a region enclosed by the green box. A close-up view of this region is given in Figure 8.

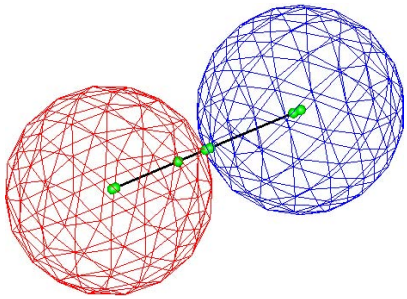

Figure 6: *Origins of the blue sphere checked during dynamic collision detection query.*

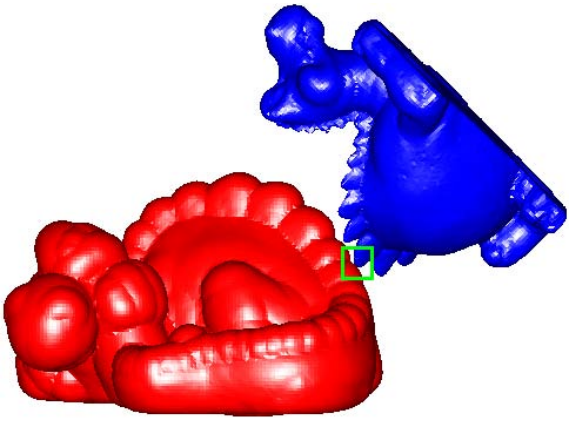

Figure 7: *Original object moving on a surface offset outward.*

3.5. Sampling the search space

Since the moving object can be at any relative orientation with respect to the fixed one, there are 3-dof for the rotational search space. Also, since the moving object can approach the fixed one from any direction, the search space of the directions is all vectors connecting the origin of the dual space to all points on the surface of the unit dual sphere, which is a 2-dimensional space. So for each offset surface we must sample a 5-dimensional space. An example of this sampling strategy is shown in Figure 9 in two dimensions for convenience: the original surface is the black circle, while the moving object is a rectangle. The red and blur circles are surfaces obtained by offsetting the original (black) circle inwards and outward, respectively. Our algorithm, generates a set of positions for different orientations of the moving rectangle on both the original and offset surfaces.

4. Experiments

When performing the collision test we measure the performance of three types of different bounding volumes:

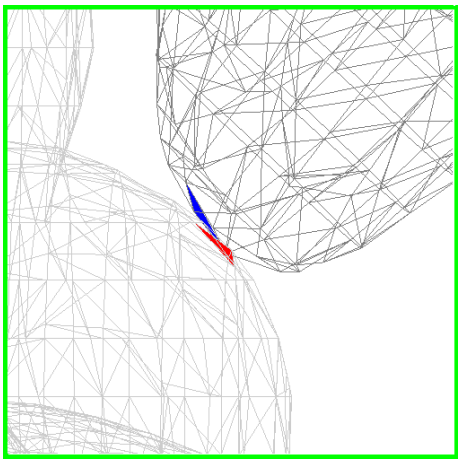

Figure 8: *Close-up view of the contacting triangles.*

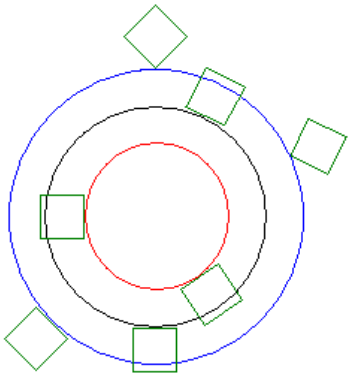

Figure 9: *2D analogy example of positioning the object on the surfaces.*

OBB-trees and AABB-trees, sphere-trees fitted to the EG07 Dragon† model consisting of 240057 vertices and 480076 triangles. The tests are run on a 1.83 GHz dual-core laptop computer with 2GB RAM running under Windows XP.

For generating the positions of the moving object we run our benchmark for three different sets of sample sets. Each set consists of different amounts of penetration and distance. The penetration and distance values are expressed as percentage of the maximum possible penetration. Note that the maximum possible penetration is the minimum value of the signed distance map.

Figure 10 shows average collision query times for each bounding volume tree for a set of penetrations and distances ranging from 0 to 25% of the maximum penetration. The plot verifies that the collision detection algorithms spend more time around the boundaries compared to other locations. The query time quickly drops to zero for increasing distance values, while it approaches a fixed constant value after some penetration.

In the second sample set we generate a coarse set of penetration values up to about 80% of the maximum penetration depth possible to see if the behavior of the collision detection algorithm changes with increasing penetration depth. As can be seen in Figure 11, the query performance practically remains the same throughout large amounts of penetration.

In the third sample set we apply the collision detection algorithms in a range of $\pm 10\%$ of maximum penetration depth around the boundary. As can be seen in Figure 12 there is irregular behavior around the boundary. This shows the power of our benchmarking algorithm because this behavior depends on the distance and penetration and could not be observed with a conventional grid-based algorithm. At this point, a more refined sample set around the boundary can be generated to capture the performance variations in the close proximity of the boundary.

Looking at all plots, we can also conclude that in general all bounding volumes tend to undergo a similar query performance changes, but for the current model OBB-trees perform much better on the average compared to the other two bounding volumes. The performance variation of OBB-tree is also much less compared to the other two. So for this 3d model it is reasonable to use OBB-trees when performing static collision detection queries.

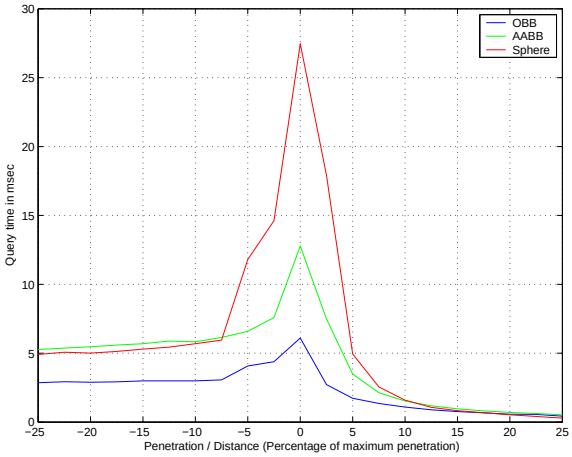

Figure 10: *Average times for each distance and bounding volume.*

5. Conclusions

The proposed benchmarking method successfully generates a set of positions for objects both at a certain distance (for non-contacting objects) and at a certain penetration depth

† http:www.cgg.cvut.cz/eg07/index.php?page=dragon

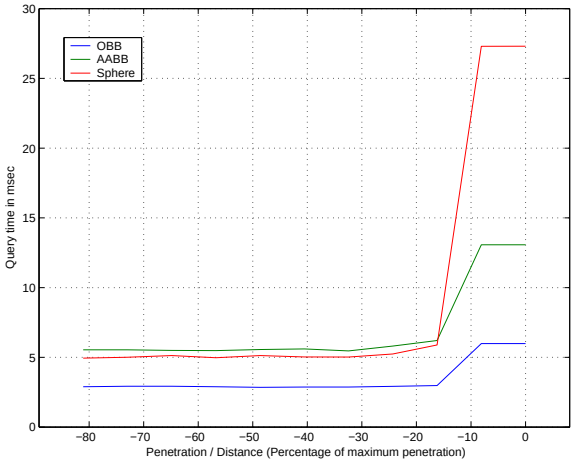

Figure 11: *Average times for large penetrations.*

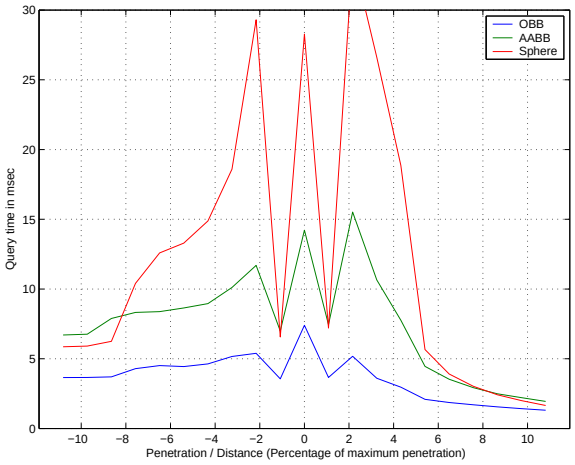

Figure 12: *Average times for small penetrations and distances around the boundary.*

(intersecting objects). By generating refined sample sets at certain distance and penetration depth ranges, we can evaluate the collision detection algorithms much more reliably. Obtained performance figures are not global as in grid-based methods but local and firmly tied to the distance and penetration parameters. This allows much better analysis of performance results.

To increase positioning accuracy we can generate more accurate offset surfaces. The accuracy of the offset surfaces depends on the resolution of the underlying distance map: for more accurate positioning, a denser signed distance map need to be constructed. In addition to this, if there is a high probability of offset surfaces getting extremely large, the offset surfaces can be generated separately by breaking the original model into parts and individual offset surface patches can be considered one by one at a time.

One of the shortcomings of the proposed method is that the model needs to be a closed surface, which is a necessary request for correct construction of the signed distance map. Another shortcoming is that there is a probability of missing some contact configurations since we employ dynamic collision detection queries for finding the exact time of first contact. Note that a complicated object of positive genus may have other interesting contact configurations inside its *tunnels*.

As part of our future work we want to eliminate these short-comings and to extend the framework to handle dynamic collision detection queries as well.

References

[CRM02] CASELLI S., REGGIANI M., MAZZOLI M.: Exploiting advanced collision detection libraries in a probabilistic motion planner. In *WSCG* (2002), pp. 103–110.

[Got00] GOTTSCHALK S.: Collision queries using oriented bounding boxes. phd thesis, university of north carolina. department of computer science, 2000.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 163–169.

[OL03] OTADUY M. A., LIN M. C.: Clods: dual hierarchies for multiresolution collision detection. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 94–101.

[TWZ07] TRENKEL S., WELLER R., ZACHMANN G.: A benchmarking suite for static collision detection algorithms. In *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)* (Plzen, Czech Republic, 29 January–1 February 2007), Skala V., (Ed.), Union Agency.

[vdB97] VAN DEN BERGEN G.: Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools 2*, 4 (1997), 1–13.

[Zac98] ZACHMANN G.: Rapid collision detection by dynamically aligned DOP-trees. In *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98* (1998), pp. 90–97.