A Fast Method for Local Penetration Depth Computation

Stephane Redon and Ming C. Lin

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

This paper presents a fast method for determining an approximation of the local penetration information for intersecting polyhedral models. As opposed to most techniques, this algorithm requires no specific knowledge of the object's geometry or topology, or any pre-processing computations. In order to achieve real-time performance even for complex, non-convex models, we decouple the computation of the local penetration directions from the computation of the corresponding local penetration depths: for any pair of intersecting objects, we partition the penetrating zones into coherent regions, and we determine for each of these regions a local penetration direction. *Then, for each of these regions, we estimate a local penetration* depth *in the previously computed penetration direction. This method has been implemented and tested on a* 2.0 *GHz Pentium PC with a NVIDIA GeForce FX* 5900 *graphics card with AGP 4× and 768MB of RAM. The results indicate that a meaningful local penetration information can be computed for complex models and challenging intersection scenarios within a few milliseconds.*

1 Introduction

The penetration depth information, *i.e.* the minimum translation required to separate two objects, is often useful for computing an appropriate collision response, or for backtracking simulation to the first time of contact. The exact penetration depth, however, is typically very difficult to compute, and several approximate methods have been introduced in the past. Although efficient, these methods often require some specific knowledge of the objects geometries (e.g. convexity) or topologies (e.g. closed manifold), or special pre-processing computations [Cam97, GS87, KR92, Ber01, KLM02, OJSL04]. Moreover, some of these methods might not always be fast enough for real-time interaction with complex models.

In this paper, we present a fast method that computes an approximation of the local penetration information and requires no pre-processing nor specific knowledge of the objects' geometry and topology.

Essentially, we simplify the problem by attempting to compute meaningful *local* penetration information. Moreover, we decouple the computation of the local penetration directions and the corresponding local penetration depths: first, we partition the penetrating zones into coherent regions, and we determine for each of these regions a local penetration *direction*. Then, for each of these regions, we estimate a local penetration *depth* along the computed penetration direction.

We do not attempt to compute the exact penetration depth, but to provide estimates of local penetration depth *as efficiently as possible*. For this reason, we make several simplifying assumptions

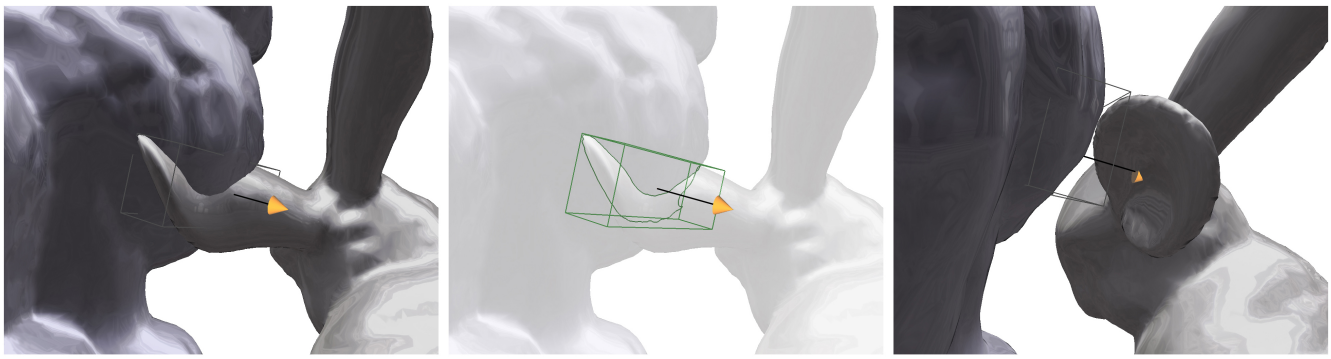

Figure 1: Two polyhedral bunnies (48,000 triangles each) intersect. Our algorithm clusters the intersection segments into a single intersection curve (left, highlighted in the center part), and determines the corresponding penetration direction and depth, which allows to separate the bunnies (right), in less than three milliseconds.

and design heuristics which work best when these assumptions are applicable, and still provide meaningful estimates in degenerate or complex cases (e.g. non-closed objects). We believe that, although not exact, our algorithm provides a sufficiently good approximation of the local penetration directions and depths, especially when interactive performance is crucial.

The overall pipeline of our algorithm consists of three stages:

- **Intersection segments clustering**. We assume that the collision detection algorithm determines a list of intersection segments. The first stage clusters these intersection segments into intersection curves, which define locally coherent penetration regions.
- **Intersection curves fitting and clustering**. When the intersection segments have been clustered into intersection curves, we perform a *final clustering* and estimate an local penetration direction for each intersection curve.
- **Local penetration depth computation**. For each intersection curve and its corresponding local penetration direction, we determine a local penetration depth, *i.e.* the minimum signed distance required to locally separate the objects in this direction.

Figure 1 shows the penetration information computed by our algorithm on two intersecting Stanford bunnies (48,000 triangles each). In this example, the penetration information is computed in less than three milliseconds.

Next, we present our algorithm in more detail. Section 2, 3 and 4 detail its three stages. Section 5 discusses some implementation details and presents some benchmarks and results, while Section 6 discusses the limitations of our approach and concludes with some future research directions.

2 Intersection segments clustering

2.1 Overview

This section describes the first stage of our algorithm to compute the local contact information. We assume that the collision detection algorithm has output a list of n_s *intersection segments*, where each segment corresponds to a non-degenerate intersection between two non-coplanar triangles. This assumption corresponds to the usual output of a triangle/triangle intersection test [Mol97]. More precisely, each intersection segment is described by its two endpoints $\mathbf{p}_1$ and $\mathbf{p}_2$.

In the first stage of our algorithm, we attempt to determine locally coherent penetration regions by clustering these intersection segments into intersection curves. Since we do not make any assumptions on the object's topology, there is no guarantee that the intersection segments *should be* or even *can be* clustered into coherent intersection curves. However, this is the case for classic models consisting of manifold surfaces and even for "*soups of polygons*". Thus, we design a technique that works well in practice for most cases.

Even for simple objects, the problem of clustering the intersection segments into intersection curves is difficult, mainly because we do not assume any knowledge of the object geometry or topology. In particular, the intersection curves could be open, if the intersecting objects themselves are not closed, or contain loops. Moreover, due to finite precision computations, two adjacent intersection segments on an intersection curve might not have a common endpoint. Also, some numerical issues in the collision detection stage might lead to some missing intersection segments, and thus result in incomplete sets of intersection segments for the penetration depth computation stage.

The simple yet effective technique we propose proceeds as follows. Assume $c-1$ curves have already been determined, and we want to determine a c-th curve from the remaining intersection segments, i.e. the *free* intersection segments. One of these free intersection segments is picked up and arbitrarily oriented, i.e. the *beginning* and the *end* of the segment are arbitrarily chosen. This segment is the first one in the c-th curve, and is also the *current end segment* of the curve. We build the curve by repeatedly appending a free intersection segment to the current end segment of the curve. Precisely, we append the free intersection segment for which one of the two endpoints is the closest to the *end endpoint* of the current end segment. This closest segment is removed from the list of the free segments, and becomes the new current end segment. The construction of the c-th curve continues as long as there exists a free intersection segment that is *sufficiently* close to the current end segment. When the construction of the c-th curve stops, the algorithm stops, if there is no free intersection segment left. Otherwise, a new free intersection segment is picked up and arbitrarily oriented, and a new curve, the $c+1$-th one, is constructed.

As can be seen in the following pseudo-code, this algorithm is simply composed of two loops:

```
curveIndex=1

WHILE there is a free intersection segment {

    // begin the curve curveIndex

    PICK UP a free intersection segment S

    // build the curve by appending free
    // intersection segments

    DO {

        ADD S to the curve curveIndex
        SET currentEndSegment = S
        SET S = the closest free intersection
                segment to currentEndSegment

    }
    WHILE a close enough segment S was found
```

```
        // proceed to the next curve

        curveIndex=curveIndex+1

    }
```

Despite its simplicity, a naive implementation of this algorithm would yield a *quadratic* complexity in the number of intersection segments, essentially due to the proximity query step, which determines the closest segment to another segment. Another reason for the quadratic complexity is in the management of the list of free intersection segments, when proper care is not taken. This is not practical when many intersection segments have been output by the collision detection functions, as we must perform the penetration depth estimation in real time for many interactive applications (e.g. games, VR, etc).

In order to achieve a nearly linear runtime complexity in the number of intersection segments, we use a combination of a hashtable and a heap. The hashtable is used to perform the proximity queries in nearly linear time, while the heap is used to manage the list of free intersection segments. The rest of the section describes how to carefully design these data structures to achieve a linear-time performance. To enhance the presentation clarity, we use pseudo-code or even C++ code bits throughout the description.

Assume an intersection segment is stored in the following class:

```
class cTContact {

public :

    cVector3f intersectionPoint[2];
    cVector3f contactNormal;
    float     penetrationDepth[2];

};
```

where `intersectionPoint` stores the two (known) endpoints of the intersection segment, `contactNormal` will contain the estimated contact normal, and `penetrationDepth` will contain the estimated penetration depth for the two endpoints of the intersection segment. The class `cVector3f` implements a 3-dimensional vector of floats. In the following, we assume that the n_s intersection segments are numbered from 0 to $n_s - 1$.

2.2 The segment heap

In order to perform constant-time access and deletion of an intersection segment from the list of free intersection segments, the heap is implemented as a *doubly-linked list stored in an array*. Precisely, each element of the array is an object of the class `cHeapCell`:

```
class cHeapCell {

public :

    cTContact *contactPointer;

    int       parentIndex;
```

```
    int       childIndex;

};
```

where `contactPointer` is a pointer to an intersection segment, `parentIndex` is the index of the parent (previous) cell in the doubly-linked list, and `childIndex` is the index of the child (next) cell in the doubly-linked list.

During the initialization, the n_s intersection segments are stored in the heap as follows. For each i, $0 \leqslant i < n_s$, the i-th intersection segment is stored in the i-th element of the array. Its `childIndex` value is set to $i-1$ and its `parentIndex` value is set to $i+1$. By convention, an element index equal to -1 is invalid, and is used to denote the beginning and the end of the heap. We use the n_s-th element of the array as the root of the heap, whose child is always the top element of the heap during the clustering. When the heap is initialized, the `childIndex` value at the root of the heap is thus set to $n_s - 1$, while its `parentIndex` value is set to -1 and its `contactPointer` value is set to `NULL`.

During clustering, when the i-th intersection segment must be removed from the heap, the `contactPointer` value of the i-th element of the array is set to `NULL`, to indicate that the i-th intersection segment is no longer free, and the `parentIndex` and `childIndex` values are used to remove the element from the doubly-linked list in constant time. Figure 2 shows an example of this heap structure for six free intersection segments at the initialization (Fig. 2b), and after one free intersection segment has been removed from the heap (Fig. 2c). Note that the heap is empty when the `childIndex` value of the root of the heap header becomes -1.

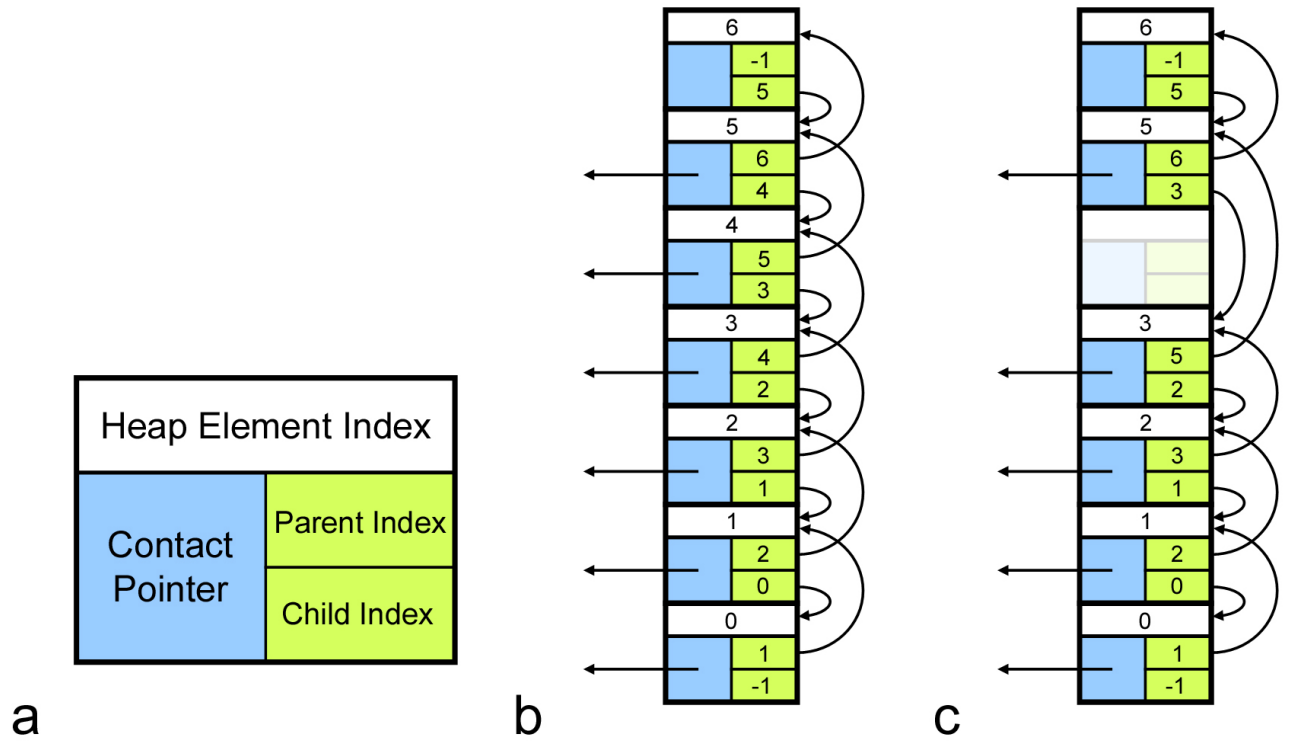

Figure 2: An example of the heap structure for six free intersection segments. (a) a single heap element; (b) the heap at the initialization; (c) the heap after the fourth free intersection segment has been removed.

2.3 The hashtable

To perform the proximity queries between the endpoints of the intersection segments, we use a spatial partitioning method. Basically, we subdivide the region surrounding the intersection segments into uniform grids and we store the non-empty space cells in a hashtable, to reduce the memory consumption. Although the method is well-known, we carefully design the data structures so as to be able to perform insertion in the hashtable in constant time, and the proximity queries in nearly constant time.

Consider again the list of n_s intersection segments, numbered from 0 to $n_s - 1$. Roughly, we begin by specifying an axis-aligned three-dimensional grid which encloses the endpoints of the intersection segments. The limits of the grid correspond to the smallest axis-aligned bounding box which encloses the intersection segments, and the number of subdivisions in the grid is chosen by a heuristic (*cf.* Section 5). We would like to restrict the proximity queries to those endpoints which are contained in the same grid cells. More precisely, when a proximity query is performed to determine the closest free intersection segment to the *end* of the current end segment, the grid is used to perform the search on those segments for which one of the endpoints at least has been stored in the same grid cell.

Instead of allocating memory for all the grid cells, however, we only store the grid cells which contain at least one endpoint of an intersection segment in a *hashtable*, implemented as an array of *hash cells*. Precisely, we define a `cHashCell` class:

```
class cHashCell {

public :

    int segmentIndex;
    int segmentExtremity;
    int cell[3];
    int nextIndex;
    int lastIndex;

};
```

where `segmentIndex` is the index indicating the location of the intersection segment in the heap (remember that the knowledge of this index allows to remove the segment from the list of free intersection segments in constant time), `segmentExtremity` is either 0 or 1 depending on which endpoint of the intersection segment has been stored in the cell, `cell[0]`, `cell[1]` and `cell[2]` contains the three integer coordinates of the grid cell which contains the beginning or the end of the intersection segment.

The two integers `nextIndex` and `lastIndex` are used to handle *collisions in the hashtable*, which occur when two endpoints have to be stored in the same grid cell, or when the hash function returns the same hashtable location for two different grid cells. Precisely, they allow us to build linked lists which store the endpoints located in the same hash cell. During clustering, when a proximity query has to be performed, the linked list which contains all the endpoints stored in the hash cell is traversed to determine the closest endpoint.

Because we can not know in advance how many collisions might occur for each cell, we use a single pre-allocated array to store all possible collisions. We thus pre-allocate *two* arrays to implement the hashtable structure - one for the original hashtable, and one to handle collisions:

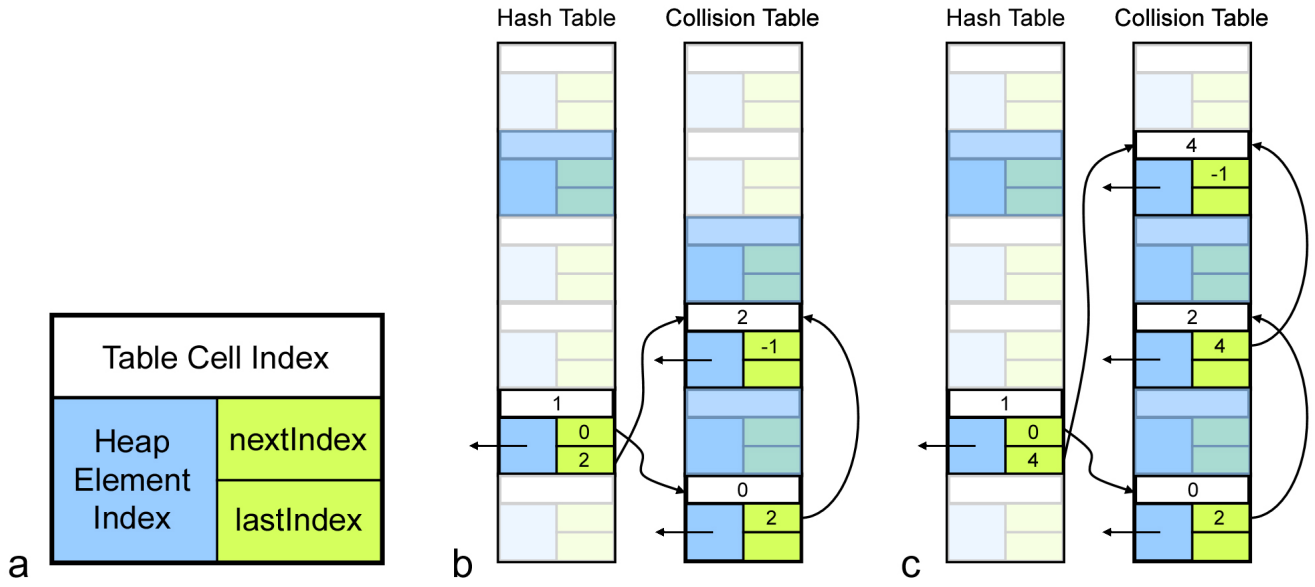

Figure 3: Constant-time insertion in the hashtable at location 1. Before the insertion, the hashtable contains three endpoints at the position 1, one in the hash table and two in the collision table. The non-empty cells that are not relevant to the example have been shadowed. (a) A table element (hash table or collision table); (b) the hash table and the collision table before insertion; (c) an endpoint has been inserted in the hashtable at location 1. Because this location was already containing three elements, the reference to the endpoint is stored in the first free collision cell. Note how the pointer to the end of the linked list is updated from 2 to 4.

```
// the hashtable

cHashCell hashTable[HASHTABLESIZE];

// the collisions table

int       nCollisions;
cHashCell collisionTable[COLLISIONTABLESIZE];
```

The integer `nCollisions` contains the number of collisions that have occurred when inserting endpoints in the hashtable, while the array `collisionTable` fills up as the number of collisions in the hashtable increases. The `nextIndex` values in the hash cells and the collision cells allow to build and traverse the linked lists of endpoints stored in the grid cell, while the `lastIndex` value of the hash cells always point to the last element of their associated linked list, to perform constant-time insertion. Figure 3 shows an example of insertion in the hashtable. The second hash cell (position 1), where the endpoint is to be inserted, already contains three endpoints: one in the hash cell, and two in the collision table (positions 0 and 2). The third element (position 2) of the collision table is the current end of the linked list of endpoints. The `lastIndex` value of the second hash cell indicates the location of this current end, thus allowing us to perform the insertion in constant-time, by filling up the collision table.

2.4 Clustering algorithm

Using these data structures, we can now perform the clustering of the intersection segments in nearly linear time in the number or intersection segments. When a new set of intersection segments has to be clustered, each segment is stored in the heap, while each of its endpoints is stored in the hashtable. When the algorithm tries to determine the closest free endpoint to a specific endpoint `e`, the hashtable is used to traverse the list of endpoints which are contained in the same grid cell as `e` and determine the closest one. Assuming each grid cell contains at most a few endpoints, the clustering is performed in linear time.

3 Intersection curves fitting and clustering

When the first stage of our algorithm completes, the intersection segments have been clustered into intersection curves. We then proceed to the second stage of our algorithm, in which we perform a *final clustering* and estimate an *average penetration direction* for each intersection curve.

This second stage is performed using *oriented bounding boxes* (OBBs). For each of the intersection curves output by the first stage of our algorithm, we determine a fitting OBB using principal component analysis [GLM96]. For each intersection curve, a 3×3 covariance matrix of the positions of the curve's endpoints is computed. This matrix is diagonalized using an iterative method [PTVF92], and the eigenvectors of the covariance matrix indicate the OBB axes. The OBB center and dimensions are then computed such as to obtain the smallest OBB which has these axes and contains the curve's endpoints.

In a first step, the OBBs are used to perform a final clustering of the intersection curves into larger intersection curves. This final clustering helps connecting curve segments that were not connected during the previous stage, either because of missing intersection segments or because two endpoints that should have been connected were stored in two different grid cells. Basically, we repeatedly merge two intersection curves as long as the distance between their fitting OBBs is below some threshold ϵ_{OBB} (*cf.* Section 5). Whenever two curves are merged into a new curve, we determine a new fitting OBB around the new curve. The process stops when no curves can be merged. Although this step is quadratic in the number of intersection curves, the number of curves at the beginning of stage two is typically very small (as opposed to the number of intersection segments), thus merging curves is typically very fast.

When the final set of intersection curves is obtained, we define the average penetration direction for each curve to be the axis of the fitting OBB which corresponds to the smallest dimension of the OBB (or alternatively the axis with the smallest associated eigenvalue). The rationale for this choice is that, for simple intersection geometry and a small amount of penetration, the actual penetration direction (*i.e.* the one corresponding to the direction of the minimal translation required to separate the objects) is well approximated by the normal of the plane fitting the intersection curve (*cf.* Section 5).

4 Local penetration depth computation

As stated in the introduction, the efficiency of our approach comes from the fact that we address the problem of estimating the minimal penetration depth by decoupling the estimation of the penetration *direction* and the determination of the corresponding *directional penetration depth*. The first two stages of our algorithm are crucial in helping to determine potential local coherent penetration

regions and corresponding penetration directions, by making use of the only knowledge available: the intersection segments.

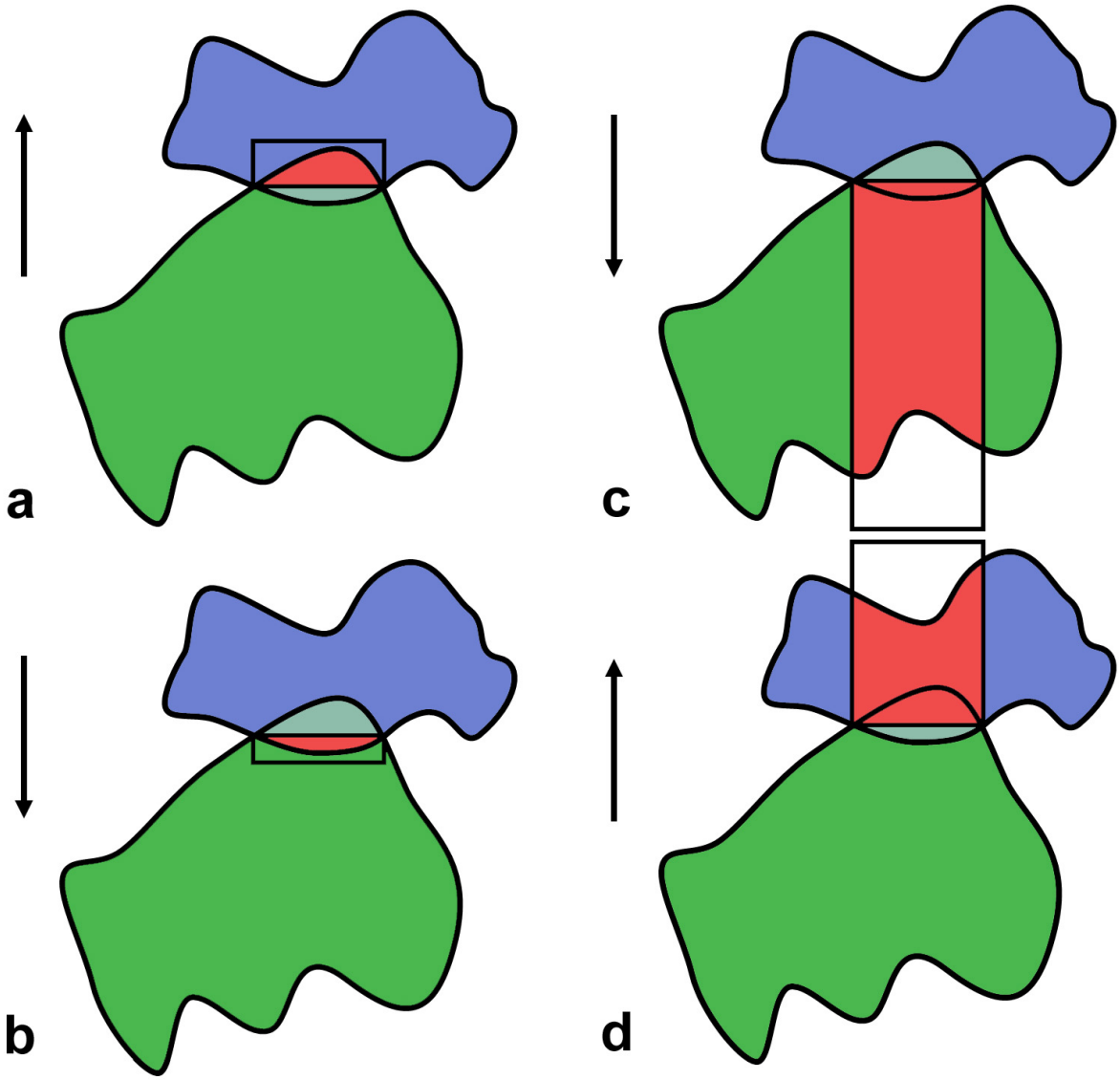

Figure 4: The penetration depth is determined by rendering both objects in both potential penetration directions. (a) First object, direction $+\mathbf{n}$. (b) second object, direction $-\mathbf{n}$. These first two renderings enable us to determine the penetration depth in direction $\mathbf{d} = +\mathbf{n}$. (c) first object, direction $-\mathbf{n}$. (d) second object, direction $+\mathbf{n}$. These last two renderings allow us to determine the penetration depth in direction $\mathbf{d} = -\mathbf{n}$. The actual penetration depth is the smallest of the two rendering passes.

The third and final stage of our algorithm simply determines a *directional penetration depth* for each intersection curve. When the directional penetration depth has been determined for each curve, the penetration depth *per intersection segment endpoint* is set to be the directional penetration depth of the belonging curve, offset by the signed distance of the endpoint to the curve's fitting plane.

In order to estimate the directional penetration depths, we use the graphics hardware. Essentially, for a given pair of objects and a given intersection curve, we render the objects in the penetrating region and perform readbacks of the z-buffer to determine the maximal penetration depth at pixel resolution, in the direction previously determined.

Assume the penetration direction is $\pm\mathbf{n}$. The direction sign is still undetermined because we do not know which one of the two directions $+\mathbf{n}$ or $-\mathbf{n}$ would require the smallest translation to separate the objects. We thus estimate the penetration in both directions and select the smallest one.

Assume we begin by estimating the penetration depth in direction $\mathbf{d} = +\mathbf{n}$. We proceed in two stages. First, we set the center of the viewing frame to be the center of the OBB which fits the intersection curve, and we set the view direction to be $\mathbf{d}$. We then render the first object and read the z-buffer back into a first two-dimensional array. Then, we reverse the view direction to $-\mathbf{d}$ and render the second object, and read the z-buffer back into a second two-dimensional array. In each case, the view limits are set such as to bound the intersection curves and contain the whole object behind the fitting plane (*cf.* Figure 4). The directional penetration depth is set to the maximum of the pairwise differences of the two penetration depth arrays. The efficiency of this last stage essentially depend on the complexity of the objects to render, and the size of the z-buffer region used to determine the penetration depth. The size of the z-buffer also determines the precision of the penetration depth determination (*cf.* Section 5).

5 Implementation and Results

5.1 Parameters

Our algorithm uses one tunable parameter per stage, whose value can be set depending on the application, the objects, and the desired performance. In the following, we indicate how these parameters might be set.

- **Number of grid cells .** The first parameter occurs in the first stage of our algorithm. If the number of cells is too small, then the performance of the first stage tends to become quadratic because of the proximity query step. On the opposite, if the total number of grid cells is too high, then two segments which belong to the same curve might not be connected (especially if some segments are missing). As this is partially corrected in the second stage through the final curves clustering. In order to determine the number of grid cells automatically, we note that a curve is mono-dimensional by definition, and thus there is on average a linear relationship between the number of intersection segments n_s and the number of grid cells for each side of the bounding box. Consequently, we express the number of grid cells for each side n_g as a linear function of the number of intersection segments: $n_s = kn_g$ (total number of cells in the grid is n_g^3). Our preliminary tests have shown that a good value for k is 50 for our benchmarks.

- **OBB/OBB distance threshold.** This parameter determines when two intersection curves should be merged. If the threshold is too large, there is a risk that two curve segments won't be detected as belonging to the same curve. However, if the threshold is too small, two intersection curves might be merged without a need for it. In both cases, the quality of the penetration depth direction is degraded. Depending on the application, this parameter might be used to control the locality of the contact information. Assuming that, on average, at most one intersection segment is locally missing in a curve, the distance threshold can be set to the average length of an intersection segment, which can be precomputed from the average size of a triangle in the models involved in the application.

- **Z-buffer size.** Both the precision and the performance of the third stage depend on the size of the region used to render the objects and determine the local penetration depth. If this region is too small, there is a risk that the penetration depth won't take into account

the sharp features of the penetration regions. On the opposite, if this region is too large, it might take too long to read back. In our tests, we have used a constant buffer size of 64x64 (128x128 for the entire test, which involves four renderings). We are currently investigating different methods to determine the region size adaptively. One of the methods can be simply setting it proportionally to the projected size of the penetration region, in order to have a constant precision when the objects are rasterized. We note that if reading-back the depth-buffer becomes a bottleneck, occlusion queries can be used to perform the depth estimation directly on the graphics hardware [OJSL04].

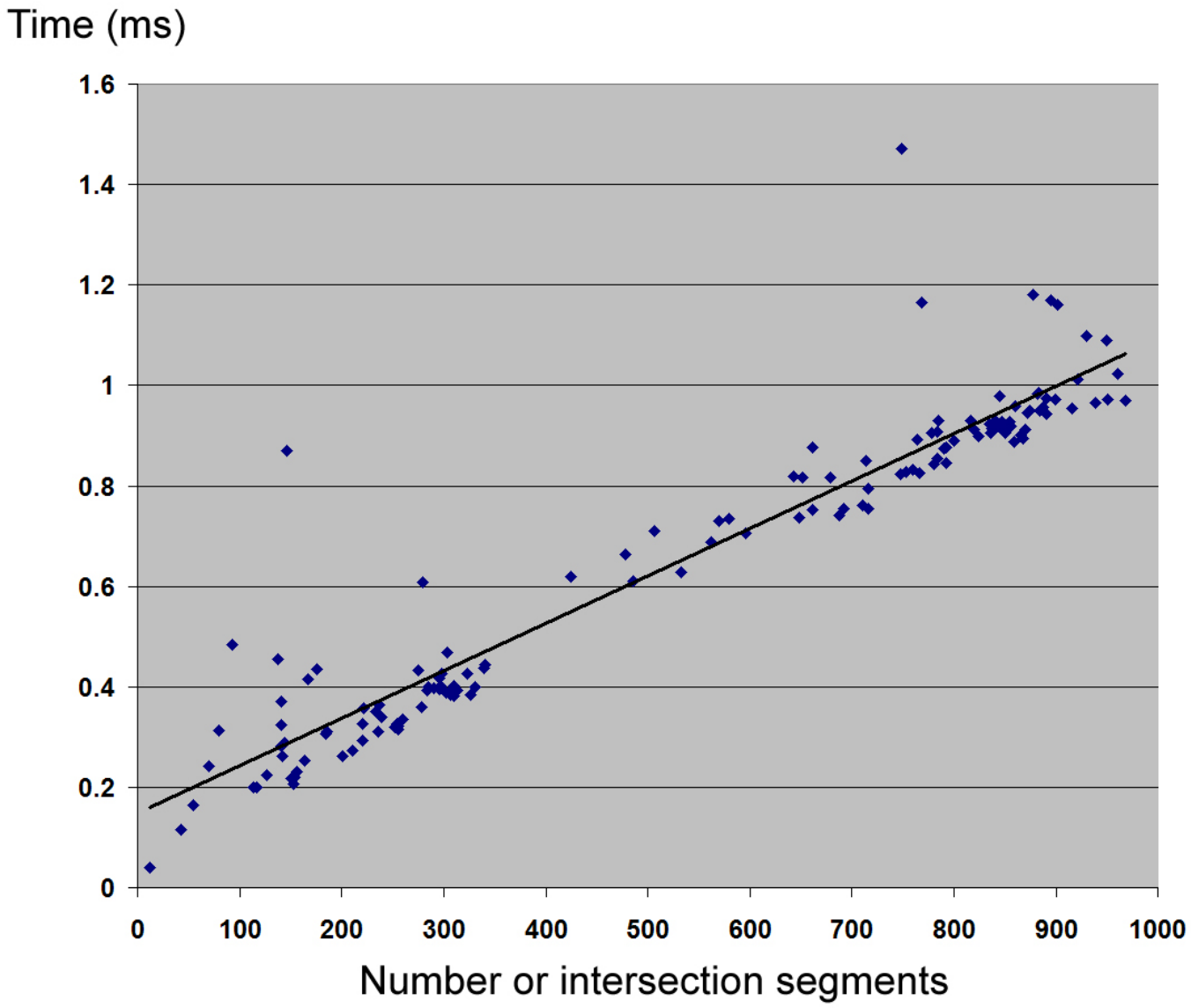

Figure 5: Total time (in msec) spent in clustering the intersection segments, depending on the number of intersection segments. Carefully designed data structures allow us to perform the clustering in linear time.

5.2 Benchmarks

This section presents various results obtained with our heuristic-based algorithm. The benchmarks have been carried out on a 2.0GHz Pentium PC with a NVIDIA GeForce FX 5900 graphics card, AGP4x, and 768MB of RAM. Several polyhedral models have been used for the tests, and we report the most challenging ones performed on a pair of Stanford bunnies (48,000 triangles each, *cf.* Figure 1)and a pair of Stanford dragons (about 300,000 triangles each, *cf.* Figure 7.c). These complex models give rise to several interesting intersection scenarios.

Figure 7 describes three such general scenarios. In all cases, our algorithm determines the appropriate intersection curves provides meaningful contact information, which allow us to locally separate the objects. The intersection curves within the OBBs are displayed, as well as the local penetration vectors. We note that, even in case of multiple intersection regions (Fig. 7(a)) or in case of degenerate intersection curves (such as the open curve in Fig. 7(b)), our algorithm is able to return an useful penetration information.

In order to determine the performance of our algorithm in practice, we have generated random intersecting configurations for the two bunnies, and measured in each case the time taken by each stage.

Figure 5 reports the time required to cluster the intersection segments into intersection curves (in msec), depending on the number of intersection segments. Figure 5 clearly indicates that the carefully implemented data structures allow us to perform the clustering in linear time, depending on the number of intersection segments. Moreover, the timings in Figure 5 also show the efficiency of the first stage: one thousand intersection segments can be clustered in roughly one millisecond.

The timings of the second stage do not exhibit any simple structure, because they depend on the number of segments per initial curve, on the number of initial curves, and on the number of curves after the final clustering. For the same set of intersecting configurations, we have observed that the average time required by the second stage is 0.33 msec, with a worst case slightly less than 1 msec.

Figure 6 shows the time required by the third and last stage of our algorithm (in msec), depending on the number of intersection curves to process. The graph clearly reports the expected linear performance (when the depth-buffer size and the pair of objects do not change). On average, the time taken by stage three for the two Stanford bunnies is about 2 msec per intersection region. Note that this includes the time to set the buffers, the view-points, render the objects, read the depth-buffer back, and perform the per-pixel depth computation.

6 Conclusion

We have described a fast method to determine meaningful local penetration directions and depths very efficiently. Our algorithm has a few limitations:

- **Locality.** Our algorithm only determines local penetration information, which could be a potential problem for consistency in collision response. But, a hybrid approach of using previous history can be incorporated to alleviate the problem.
- **Image-precision errors**. Our algorithm makes use of the graphics hardware to determine the directional penetration depth, up to pixel resolution. This can prevent fine, sharp features to be taken into account when determining the penetration depth.
- **Discrete Approximation.** As mentioned, our algorithm is not exact. In particular, there is no guarantee that the local penetration can be completely removed by only using the local penetration information returned by an approximation algorithm.

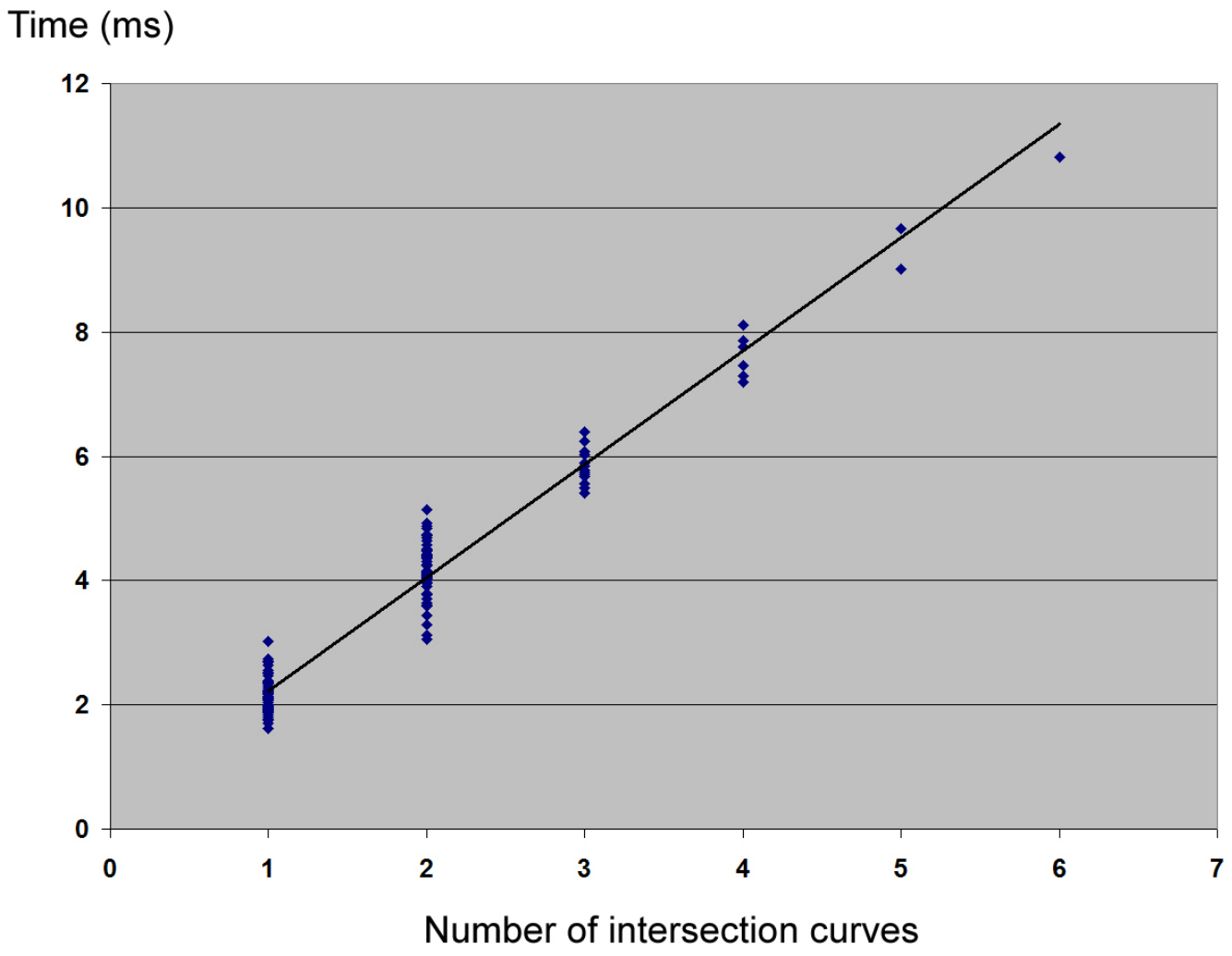

Figure 6: Total time (in msec) spent in the third stage of our algorithm, the determination of the local penetration depth, depends on the number of intersection curves to process. When the size of the depth-buffer is fixed, the computation time per intersection curve is nearly constant.

In practice, however, we observed that our algorithm returns meaningful and useful penetration information even for difficult intersection scenarios involving complex, general (non-convex) objects. Furthermore, the algorithm was able to determine local penetration information within just a few milliseconds.

Acknowledgements

The authors wish to thank Stanford University for the bunny and dragon models, and Miguel A. Otaduy for insightful discussions and his reviewing of a first version of this paper. This project is supported in part by the Office of Naval Research VIRTE Program, National Science Foundation, U.S. Army Research Office, and Intel Corporation.

References

[Ber01] BERGEN, G. van den. Proximity queries and penetration depth computation on 3d game objects. Game Developers Conference,2001. 1

[Cam97] CAMERON, S. 1997. Enhancing GJK: Computing minimum and penetration distance between convex polyhedra. Proceedings of International Conference on Robotics and Automation, 3112- 3117. 1

[GLM96] GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. In SIGGRAPH 96 Conference Proceedings, Annual Conference Series. ACM SIGGRAPH, Addison Wesley, August 1996. 8

[GS87] GUIBAS, L., AND SEIDEL, R. 1987. Computing convolutions by reciprocal search. Discrete Comput. Geom 2, 175-193. 1

[KR92] KAUL, A., AND ROSSIGNAC, J. 1992. Solid-interpolating deformations: construction and animation of pips. Computer and Graphics 16, 107-116. 1

[KLM02] KIM, Y. J., LIN, M. C., AND MANOCHA, D. 2002. DEEP: Dualspace Expansion for Estimating Penetration depth between convex polytopes. In IEEE Conference on Robotics and Automation. 1

[Mol97] MOLLER, T. A Fast Triangle-Triangle Intersection Test. Journal of Graphics Tools, vol. 2, no. 2, pp. 25-30, 1997. 2

[OJSL04] OTADUY, M. A., JAIN, N., SUD, A., AND LIN, M. C. Haptic Display of Interaction between Textured Models. In Proceedings of IEEE Visualization Conference. Austin, Tx. 2004. 1, 11

[PTVF92] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T. AND FLANNERY, B. R. Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, Cambridge, second edition, 1992. 8

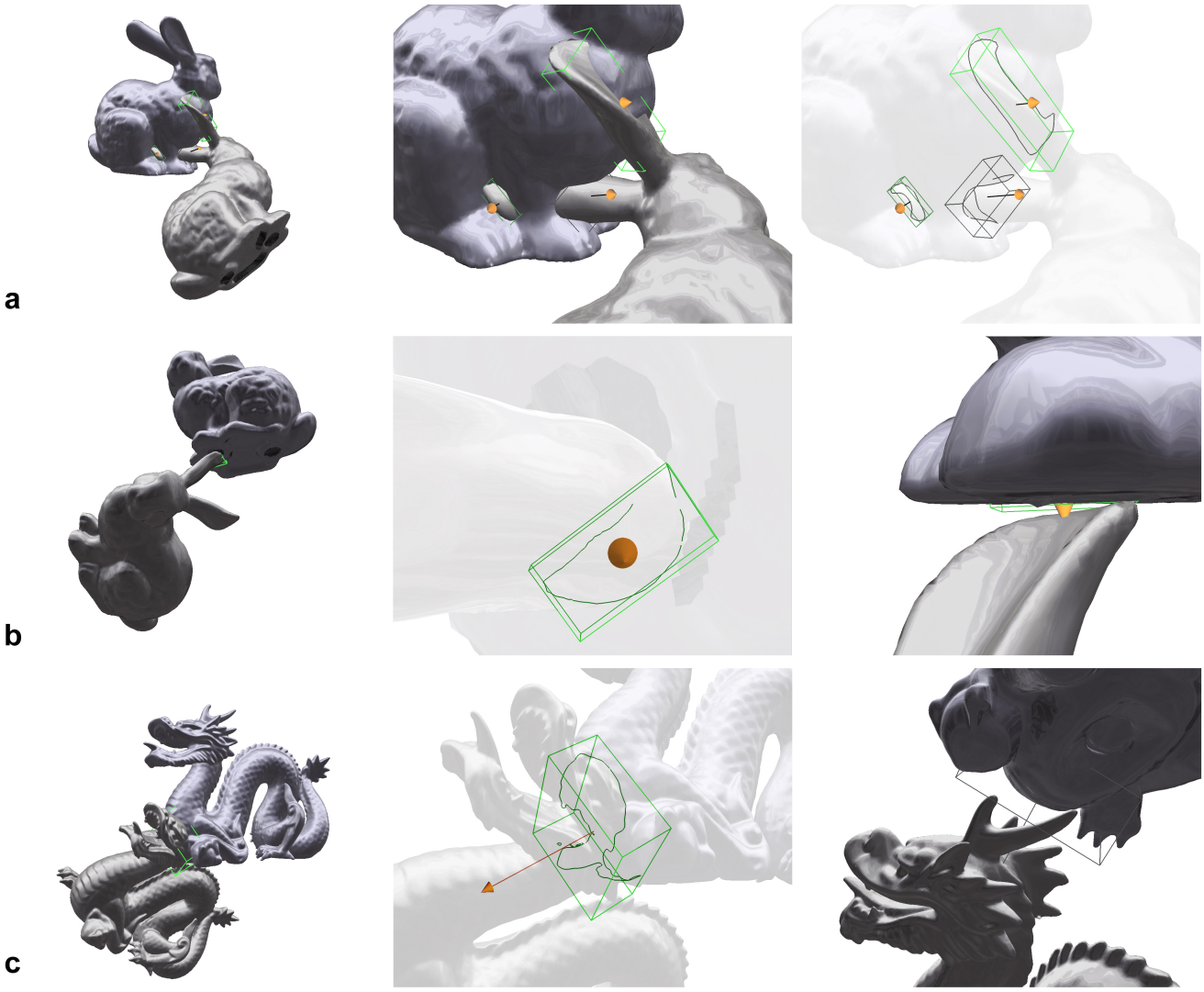

Figure 7: Three penetration depth scenarios. a: Even in case of a complex intersection scenario, our algorithm determines a satisfying partitioning of the intersection segments, and the corresponding local penetration information. b: Holes in the base of the bunnies (left) lead to open intersection curves appropriately detected by our algorithm (center), still allowing to precisely separate the objects (right). c: Two dragon models (300,000 triangles each) intersect (left). Our algorithm clusters 1,491 intersection segments in a single curve (center) and determines a penetration information which allows to precisely separate the objects (right), in 9.5 milliseconds.