A Fast Method for Local Penetration Depth Computation Appendix

Stephane Redon and Ming C. Lin

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

In this appendix, we provide some details on the data structures used to cluster the intersection segments. Although the data structures are fairly classical (a heap, a grid and a hashtable), we pay attention to the implementation details to ensure that the segment clustering is performed in almost linear time.

1 Intersection segment clustering

1.1 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. 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. This is typically the case for models consisting of non-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 present 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 selected and arbitrarily oriented (i.e. the *beginning* and the *end* of the segment are arbitrarily set). 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, provided that there is no free intersection segment left. Otherwise, a new free intersection segment is selected 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

    Select 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 seg-

ment. 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 $n_s - 1$.

1.2 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*. More 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 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 1 shows an example of this heap structure for six free intersection segments at the initialization (Fig. 1b), and after one free intersection segment has been removed from the heap (Fig. 1c). Note that the heap is empty when the `childIndex` value of the root of the heap header becomes -1.

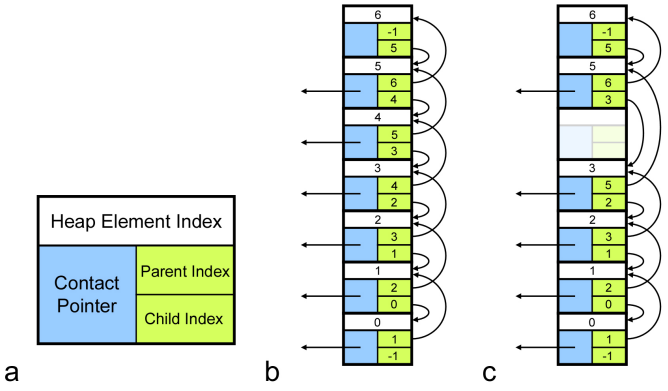

Figure 1: An example of the heap structure for six free intersection segments. (a) a single heap element; (b) the heap at the initialization; (c) the heap after the fourth free intersection segment has been removed.

1.3 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. 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*. More 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.

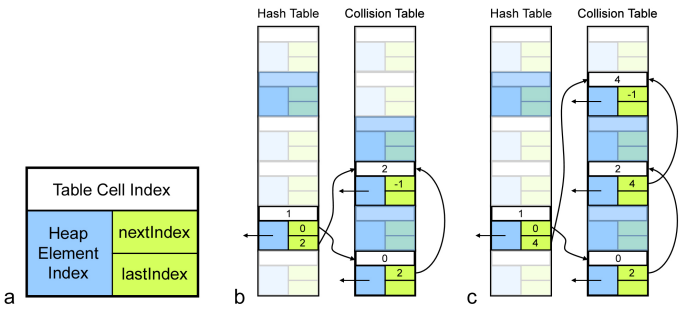

Figure 2: Constant-time insertion in the hashtable at location 1. Before the insertion, the hashtable contains three endpoints at the position 1, one in the hash table and two in the collision table. The non-empty cells that are not relevant to the example have been shadowed. (a) A table element (hash 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 already contained 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 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. More 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:

```
// 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 2 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.

1.4 Clustering algorithm

Using these data structures, we can now perform the clustering of the intersection segments in nearly linear time in the number of 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.