Collision Detection Algorithm for Deformable Objects Using OpenGL

Shmuel Aharon and Christophe Lenglet

Imaging and Visualization Department,
Siemens Corporate Research, Princeton, NJ
aharon@scr.siemens.com

Abstract. This paper describes a collision detection method for polygonal deformable objects using OpenGL, which is suitable for surgery simulations. The method relies on the OpenGL selection mode which can be used to find out which objects or geometrical primitives (such as polygons) in the scene are drawn inside a specified region, called the *viewing volume*. We achieve a significant reduction in the detection time by using a data structure based on an AABB tree. The strength of our method is that it doesn't require the AABB hierarchy tree to be updated from bottom to top. We are using only a limited set of bounding volumes, which is much smaller than the object's number of polygons. This enables us to perform a fast update of our structure when objects deform. Therefore, our approach appears to be a reasonable choice for collision detection of deformable objects.

1 Introduction

Many interactive virtual environments, such as surgery simulations, need to determine if two or more surfaces are colliding. That is, if there are surfaces that are touching and/or intersecting with each other. Finding the exact locations of these areas is a key process in this kind of application. For realistic interactions/simulations these calculations require good timing performance and accuracy.

Collision detection algorithms have been published extensively. The most general and versatile algorithms are based on bounding volume hierarchies to detect collisions between polygonal models. These algorithms are primarily categorized by the type of bounding volume that is used at each node of the hierarchy tree. That is, axis aligned bounding boxes (AABB) [1], object oriented bounding boxes (OOBB) [2], or bounding spheres [3], [4], [5]. The main limitation of these algorithms is that for deformable objects, one needs to update (or re-build) the hierarchy trees at every step of the simulation. This is a time-consuming step that significantly reduces the efficiency of these algorithms.

As far as we can tell, there are only two algorithms, known today, that are using graphics hardware acceleration for collision detection. One of them is the approach of Hoff et al. [6], which is limited to collisions between two-dimensional objects, or to some specialized three-dimensional scenes, such as those whose objects collide only in a two-dimensional plane.

The second approach, suggested by Lombardo et al. [7], uses the OpenGL selection mode to identify collisions between polygonal surfaces. However, it is limited to the collisions between a deformable polygonal surface and an object with a very simple shape, such as a cylinder or a box. Furthermore, the performance of this algorithm significantly decreases with the increase of the object's number of polygons. Therefore, it is limited to objects with relatively small number of polygons.

This paper presents a new approach for collision detection using Open GL which allows a fast and accurate way to detect the collisions between the individual polygons of deformable objects.

2 The Collision Detection Algorithm

The collision detection algorithm suggested here detects collisions between a specified object, the reference object, and all other objects in the scene. We are assuming that each object is built from a set of polygons, which are either triangles or quadrangles. Following is a description of the various steps needed to complete a collision query with our algorithm.

The algorithm is based on rendering the scene in selection mode, using an orthographic camera, with OpenGL graphics. Selection is a mode of operation for OpenGL, which automatically tells which objects in the scene are drawn inside a specified region, called the *viewing volume*. Before rendering it is necessary to provide a "name" for each object/primitive in the scene. After rendering in selection mode, OpenGL will return the list of all the "names" of the objects/primitives that are drawn inside the viewing volume. Further details about the OpenGL selection mode can be found in [7], [9].

2.1 Bounding Volumes Hierarchy Creation

As a pre-processing step, it is necessary to divide the surface of each object into a set of Axis Aligned Bounding Boxes (AABB), in such a way that each bounding box contains no more than a specified number of polygons, and each polygon belongs to one and only one bounding box. We start building an AABB tree using the method suggested in [1], with the following modifications.

Only the root and the leaves of the tree are used, ignoring all its internal nodes. This allows us to perform very fast updates since there is no need to consider the complete tree structure. We stop subdividing a bounding box when the number of polygons it contains is below a specified threshold.

It may happen that the final bounding box has large empty spaces. This will be the case if it contains polygons that are not continuously connected in space. To prevent this, further subdivide it into two bounding boxes as described in [1]. If the resulting two bounding boxes are completely separated - keep them, otherwise - keep their parent bounding box.

2.2 The Collision Query

Step 1: Find the objects' bounding boxes that intersect with the global bounding box of the reference object. This is done by defining the global bounding box of the reference object as the OpenGL *viewing volume*, and rendering all the bounding boxes of all other objects, defined as triangle strips, in selection mode, using a different "name" for each one. We use triangle strips, which are the most optimized OpenGL primitive, to ensure high performance [8].

Collisions can occur only in the area of the bounding boxes that intersect with the global bounding box of the reference object. Therefore, only these bounding boxes will be processed in the next step. If there is no bounding box that intersects the global bounding box of the reference object, then there is no collision and the algorithm stops.

Step 2: Find the bounding boxes of the reference object that intersect with the bounding boxes found in the previous step. This is done, similar to step 1, by defining the OpenGL *viewing volume* as one of the bounding boxes found in the previous step, and render all the bounding boxes of the reference object, defined as triangle strips, in selection mode, using a different "name" for each one. Repeat this process using all of the bounding boxes that have been found in the previous step.

The goal of this step is to have, for each bounding box from an object that is potentially involved in a collision, the list of all bounding boxes from the reference object that intersect with it, if any.

Step 3: Find the list of all the reference object's polygons that have potential collisions with an object(s) in the scene. For a given object bounding box *B*, found in step 1, render in selection mode, all the polygons contained in the bounding boxes from the reference object that found in step 2 to intersect *B*.

This allows a major computational saving since it can greatly reduce the final number of polygon-polygon intersection checks. In order to minimize the number of *viewing volume* definitions, this step can be combined with step 2.

Step 4: Find all the polygons, from all objects, that intersect with the bounding boxes of the reference object. These polygons have potential collisions with the reference object. To do this define, as was done in step 3, the OpenGL *viewing volume* as one of the reference object bounding boxes that has known intersections with one or more objects' bounding boxes. Then render in selection mode, all the polygons from the objects' bounding boxes that were found to intersect with this reference object bounding box, using a different "name" for each polygon. Repeat this procedure for all the reference object bounding boxes.

The result of this processing provides the following for each reference object bounding box:

- The list L_r^i of the potentially colliding polygons inside this bounding box (i) of the reference object (found in step 3).
- The list L_{ri}^{jk} of the polygons potentially colliding with polygons of L_r^i inside the bounding boxes (k) from object (j) of the scene (found in step 4). Where j goes from 1 to the number of scene's object (excluding the reference object), and k goes from 1 to the number of bounding boxes for object j.

This limits the number of polygons that have possible collisions, and hence significantly reduced the number of polygon-polygon intersection tests.

Step 5: Find the polygons of the reference object that are colliding with polygons from an object, or objects, and the list of these polygons. For every polygon, P, in a L_r^i list, with i going from 1 to the number of reference object bounding boxes, find whether or not this polygon really intersects the polygons from the L_{ri}^{jk} lists. To do so, define the polygon's *viewing volume* to be a tightly fitting volume around the polygon *P* (see section 2.3 for details). Then render in selection mode all the polygons in a L_{ri}^{jk} list, giving a different name to each of them. Every polygon that is found inside the specified polygon's *viewing volume* is actually intersecting the given polygon, *P*. The accuracy of this detection algorithm is limited to how accurate the polygon's *viewing volume* actually limits the region that this polygon occupies in the world, and can be made as good as possible with no additional cost. Repeat this step for all the non-empty L_{ri}^{jk} lists, and for all the L_{r}^{i} lists.

This step provides the list of all the polygons from the reference object and the polygons from the scene's object(s) that intersect. This is the desired result of the collision detection algorithm.

2.3 Defining the Polygon's *Viewing Volume*

The goal is to define a small region tightly covering a given polygon, *P*, as the OpenGL *viewing volume*. Then render all other polygons of interest in selection mode to find out if they are drawn inside the specified polygon *viewing volume*, which means that they are intersecting with it. The accuracy of this detection method is defined by the accuracy of the *viewing volume* definition, and how well it really describes the region that this polygon occupies in the world.

The rendering is done using an orthographic camera. In this case the OpenGL *viewing volume* is a rectangular parallelepiped (or more informally, a box), defined by 6 values representing 6 planes named left, right, bottom, top, near and far. Below are the steps to define a polygon's *viewing volume*.

First, find the polygon's two-dimensional bounding box that resides in its plane. Then specify the viewing volume around this bounding box. That is define the left, right, bottom, and top planes of the viewing volume as the four edges of this bounding box. Next define the depth of the volume to be a very small number, ε, which specifies the required accuracy (we found that ε=0.001 gives good results). That is, specify the near and far planes to be $\frac{-\varepsilon}{2}$, and $\frac{\varepsilon}{2}$ from the polygon's plane along the polygon's normal.

If the polygon *P* under consideration is a triangular polygon one need to add two clipping planes to limit the viewing volume to a pyramid, tightly fitted around the polygon, as shown in Fig. 1.

This method can be easily adapted for quadrangular polygons. In this case, three clipping planes might be needed to limit the viewing volume to the quadrangle edges.

2.4 Updating the AABB Structure for Object's Deformations

When dealing with deformable objects it is necessary to update the Axis Aligned Bounding Boxes (AABB) structure after every step of the simulation. As mentioned before, we only keep the root of an AABB tree (the global bounding box of an object), and its leaves. Due to the relatively small number of bounding boxes that need to be updated, this task is performed rather quickly. Every bounding box is re-fitted around all the polygons that it contains. We further optimized this step by using the Streaming SIMD Extensions (SSE) provided by Intel processors since the release of the Pentium III processor (see [10] for details). This allows us to perform the update of the AABB structure twice faster.

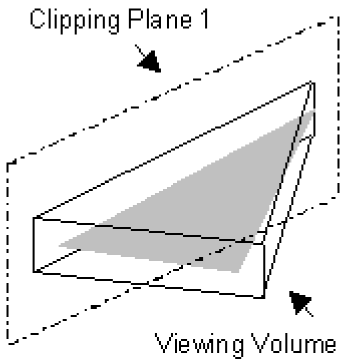

Fig. 1. Viewing Volume fitting a triangular polygon (shaded). For clarity only one clipping plane is shown

3 Results

We performed several tests of our collision detection algorithm in order to evaluate its performance, the effect of the maximum number of polygons allowed in a bounding box, and the effect of object's number of polygons, on the detection time. The tests were done on an Intel Pentium III 930 MHz processor with a Matrox Millennium G450 32 MB graphics card. All reported values are the average of a set of about 2000 collisions with approximately 10-15 colliding polygons. We tested the algorithm using a model of a scalpel consisting of 128 triangles, and one of the following polygonal models: Face with 1252 triangles, Teapot - 3752 triangles, Airways – 14436 triangles, Colon – 32375 triangles, and the Spinal Column with the Hips – 51910 triangles. The last three models were generated from clinical CT data.

3.1 Effect of Number of Polygons per Bounding Box on the Performance

The effect of the number of polygons allowed in a bounding box on the performance of our collisions detection method is shown in Fig. 2, for objects with different number of polygons.

As expected, increasing the number of polygons in a bounding box will decrease the collision detection performance, since it will require to render large number of polygons for every box that has potential collision polygons and to perform many polygon-polygon intersection tests. On the other hand, having a small number of polygons in a bounding box, will lead to too many boxes for each mesh. Hence, increasing the number of boxes that have to be tested and will also decrease the efficiency of the detection algorithm.

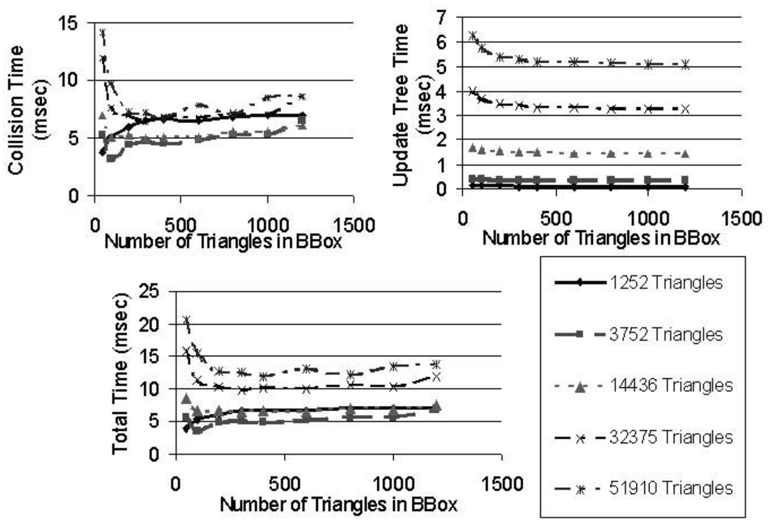

Fig. 2. Effect of the number of polygons in a bounding box (BBox) on the performance of the collision detection (upper-left), the update of the AABB tree structure (upper-right), and the total iteration time (bottom) for objects with different number of triangular polygons

However, as can be seen in Fig. 2, the performance of the collision detection is not very sensitive to the selection of the number of polygons in a bounding box. Therefore, a simple rule of thumb can be used to specify this number. That is, having about 300-600 polygons in a bounding box will have the best performance for large objects (more than 5000 polygons) and 50-150 polygons per bounding box for small objects (less than 5000 polygons).

It is worth mentioning that changing the number of polygons in a bounding box, hence the number of bounding boxes we have, almost doesn't affect the performance of updating the AABB structure. This is not surprising since no matter how many bounding boxes we have, we need to process all the polygons within them, that is all the object's polygons.

3.2 Effect of the Object's Number of Polygons on the Performance

The effect of the object's number of polygons on the collision detection performance is shown in Fig. 3.

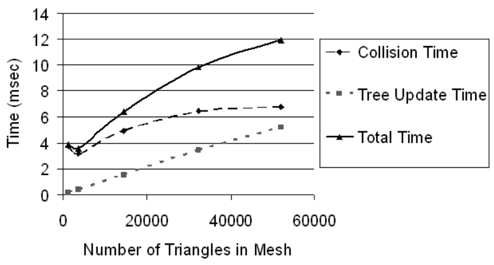

Fig. 3. Effect of the object's number of polygons on the collision detection performance

As can be seen in Fig. 3, the collisions detection time increases with the total number of polygons. However, the increase rate is far below a linear increase rate (i.e. O(n)). This means, that our algorithm can handle efficiently deformable models with a large number of polygons without a huge penalty in its performance. This is the main strength of the algorithm we presented here.

3.3 Performance Evaluation

Our goal was to compare the performance of our method to others. However, not all of the implementations were available to us. Therefore we performed a ballpark evaluation using public benchmark information [11]. Using the benchmark information we were able to estimate the difference in performance between the machines used to provide timing information of the various collision detection methods. Although this gives only ballpark estimates, it is sufficient to provide an idea of how well our algorithm performs in comparison to others.

The Object Oriented Bounding Box (OOBB) method [2], used by RAPID, is very efficient in the collision query. However, it is necessary to rebuild the OOBB tree or refit it at every step when objects deformed. This is a time consuming task (in the order of tens of milliseconds for objects with couple of thousands of polygons, see [1]) that makes this method not suitable for use with deformable objects.

The AABB method is also a very fast method for collision query, which can be updated for object’s deformation [1]. However, updating its tree structure is still the bottleneck for this method. As reported in [1] (and estimated for our machine) it takes about 1.5 milliseconds to update the AABB tree for an object with 3752 polygons, while our modified AABB structure can be updated in 0.39 milliseconds. The cost of the tree update increases significantly with the number of polygons using the AABB method. Therefore, although the AABB tree performs a collisions query much faster than our method, the overall performance for each iteration is slower than ours, in particular for large objects.

Finally, Brown et. al. [5] suggested a method to update the bounding sphere tree method reported by Quinlan [4]. They reported a time of about 0.02 milliseconds per triangle for the tree structure updates. That is, it will be in the order of 20 milliseconds for an object with 1000 triangles, while with our method we can updated 50000 triangles in about 5 milliseconds. This again implies that our method is much more efficient for large deformable objects.

4 Conclusions

We have presented an algorithm for collision detection between polygonal deformable objects using Open GL. The algorithm is based on the OpenGL selection mode combined with a structure of axis aligned bounding boxes. It performs well on deformable objects with a large number of polygons, with a relatively small cost in performance when increasing the number of polygons. This method is particularly suitable for surgery simulations, where fast interaction is essential.

References

1. Van Den Bergen G.: Efficient collision detection of complex deformable models using AABB trees. Journal of Graphics Tools (USA), vol. 2, no. 4, p. 1-13, 1997.
2. Gottschalk S., Lin M.C., Manocha D. OBB Tree: a hierarchical structure for rapid interference detection. Proceedings of 23rd International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'96), New Orleans, LA, USA, 4-9 Aug. 1996.
3. Larsen E., Gottschalk S., Lin M.C., and Manocha D. Fast distance queries with rectangular swept sphere volumes. Proceedings 2000 ICRA. IEEE International Conference on Robotics and Automation, vol.4, San Francisco, CA, USA, 24-28 April 2000.
4. Quinlan S. Efficient distance computation between non-convex objects. Proceedings of the 1994 IEEE International Conference on Robotics and Automation, vol. 4, San Diego, CA, USA, 8-13 May 1994.
5. Brown J., Sorkin S., Bruyns C., Latombe JC., Montgomery K., and Stephanides M.: Real-Time Simulation of Deformable Objects: Tools and Application, Computer Animation, Seoul, Korea, November 7-8, 2001.
6. Hoff K.E., Zaferakis A., Lin M.C., and Manocha D. Fast and simple 2D geometric proximity queries using graphics hardware. Proceedings of the 2001 symposium on Interactive 3D graphics, p. 145-148, ACM Press New York, NY, USA, 2001.
7. Lombardo J.C., Cani M.P., and Neyret F. Real-time collision detection for virtual surgery. Proceedings Computer Animation, Geneva, Switzerland, p.82-90, 26-29 May 1999.
8. Evans F., Skiena S., and Varshney A.: Optimizing Triangle Strips for Fast Rendering. Proceedings of IEEE Visualization 96, pp. 316-326, 27 October – 1 November 1996.
9. Woo M., Neider J., Davis T., and Shreiner D.: OpenGL Programming Guide. Third Edition. Addison-Wesley, Massachusetts, USA, 2000.
10. Intel Corporation: Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler. January 1999.
11. SPEC CPU95 Benchmark. Standard Performance Evaluation Corporation. www.spec.org.