GJK for deformable object collision detection

Maher Hatab, Abderrahmane Kheddar

GJK for Deformable Object Collision Detection

Maher Hatab*
Universite d'Evry Val d'Essonne
Evry, France

Abderrahmane Kheddar†
JRL - CNRS
Tsukuba, Japan

Abstract *- GJK is one of the main methods for distance calculations between convex objects. In this article, the adaptation of GJK for deformable object collision detection is proposed. Although the original method is only suited for distance calculations, the proposed method is capable of finding the colliding triangle pairs between two continuously deforming virtual objects in real time. Furthermore, it handles all deformation types at no extra time cost and it allows the client application to add, or remove triangles from the considered object meshes at run time with no extra overhead for the collision detection. The proposed method is very flexible in many aspects, making it an ideal choice for virtual reality and haptic applications.*

I. INTRODUCTION

Computer simulated scenes gained in complexity and realism over the years. It is crucial for real time applications to develop fast and reliable algorithms, capable of providing high fidelity multimodal rendering at interactive rates. This is even more crucial in interactive simulations such as the ones using haptic feedback. Physically based simulations are gaining in acceptability because they use rigorous mathematical basis to drive graphical animation. In all these applications, collision detection (CD) is an important unavoidable step, but it is still the bottleneck process of any implementation. There are several important methods that have been proposed for CD with rigid or deformable models and in robotic planning. We refer the reader to the following surveys [12, 19].

One of the main advantages in targeting a standard CD module is its potential to be ported to a dedicated hardware card which will free developers in various domains from this recurrent problem. Recent methods are trying to use nowadays GPU computation capabilities [8, 14]. The problem is that as far as we focus on particular category of objects, any method would not open hardware implementation perspectives. The hardware implementation of the hidden surface removal algorithm (the well known Z-buffering) was made possible because the algorithm revealed to be generic, simple and applicable to any kind of objects. It is in this spirit that we are developing research on CD.

The rest of the paper is organized as follows. In the next section, we start by a brief reminder of the main collision detection methods. Next we explain in greater details the GJK algorithm and some of its modifications that are directly related to our method. Then we introduce the cut procedure. This simple procedure is at the heart of the improved version of GJK, which is presented in section 4. In section 5 we show how to include the combined GJK of section 4 in a recursive procedure, capable of finding the colliding triangle pairs of the objects in a similar manner to bounding volume hierarchies. We finish the article by the implementation results and performance comparison against the AABB method and a discussion of the obtained figures. We conclude by interesting future enhancements of the proposed algorithm.

*e-mail: hatab@iup.univ-evry.fr
†email: kheddar@ieee.org

II. RELATED WORK

Traditionally CD occurs in two main phases (or passes): (i) an acceleration phase in which objects shapes are approximated with a succession of rough to relatively more tiny bounding volumes, called bounding volume hierarchy (BVH) and (ii) an exact, precise collision checking computation that takes place at the lower level of the BVH and made on the actual object's shape. None of existing papers demonstrates a clear and plausible relation between the nature of objects, the number of vertices and the recommended level of the BVH trees. There is indeed an optimum, but very dependent on various parameters.

Bounding volumes hierarchies (BVH) have been used for solid objects CD long ago. Spheres [9], OBBs [7], $k-$DOPS [13] are among the proposed basic shapes to enclose the objects. Strategies for tree construction such as bottom-up, top-down, or insertion [6] produce variably fit hierarchies, traversed using depth first, level first, bigger volume first, or interruptible strategies. The latter is used for time critical applications [10, 17].

The adaptation of BVH to deformable objects mainly focused on refitting previously constructed hierarchies instead of completely rebuilding them. Thus Van Den Bergen [20] proposed a way to refit the AABB hierarchy which is made possible by the preparation of an AABB for each object with respect to its own reference. The tree starts by refitting the leaves, containing one polygon each, and then refits the higher levels of the tree by adjusting the size of each AABB to contain its child boxes. He also proved that refitting the tree can produce less optimized hierarchies than rebuilding them, but the overall CD process is up to 15 times faster. More recently, [15] considered several factors in building and traversing BVH for deformable object collision detection, and James and Pai [11] proposed to adjust the sphere hierarchies using the reduced deformable models. We refer the reader to this recent survey [19] on deformable objects CD techniques.

The first collision detection techniques only operated on convex objects since they present many exploitable advantages. The two main convex-based methods are the GJK [5] and the LC [16]. GJK finds the separation distance between the convex hulls of two point clouds at linear time. LC on the other hand only operates on convex hulls, and that at constant time.

It walks on the surface of the hulls jumping between vertices, edges and faces in order to converge to the minimum separation distance. Convex hulls are rarely used as bounding volumes in real time application involving deformable objects, due to the extensive building time they require. Furthermore, the object decomposition into its convex subparts is not unique and usually requires a large number of objects to completely cover the original ones. One solution is to implement acceleration levels, capable of selecting pairs of convex subparts for the exact CD process [2, 4] but are not suited for deformable objects. The next section contains a more detailed explanation of GJK's general algorithm and its main modifications that have been proposed over the years.

[18] developed a fast triangle-triangle test that we chose to use in our implementations.

Our Contribution

We introduce an adaptation of the known GJK in conjunction with a recursive procedure for deformable object collision detection. Its main characteristics are the following:

- It runs in real time for a pair of meshes of moderate sizes while **deterministically** finding the colliding triangle pairs.
- **No restrictions** are imposed on the deformation types or amounts, furthermore, such changes do not require additional computation time for CD.
- Its performance is not affected by **triangle creation or removal**. In fact, the client application can have a giant triangle pool, and can choose at each time step the triangles for which the CD should be performed.

III. GJK BASICS

The original GJK [5] is capable of finding the separation distance between the convex hulls of a couple of point clouds in case the objects are disjoint. Even though the convex hulls are not actually built, the algorithm can find the separation distance in $\mathcal{O}(n+m)$ where n and m are the clouds' point counts. Internally, it uses simplices, and a distance sub-algorithm for finding the minimum distance between them. It starts by choosing a support direction, and then finds the support vertices in that direction on both objects. The newly found vertices are added to the simplices, and the simplices results form the new support direction to be used in the next iteration. The algorithm iterates in that manner until the minimum separation distance is found. The termination condition is a comparison between the current iteration's simplices result and the minimum theoretical distance at that iteration, while accounting for computational errors. When the objects are colliding, the method can find an upper bound for the minimum penetration distance. We refer the reader to the original paper for more details.

A first modification was proposed in [1] where the simplices and the distance sub-algorithm were replaced by a formula for guessing $iteration_{i+1}$ support direction based on the $iteration_i$ support direction and support vertices. The main gain is the ability to find the new direction without having to solve the simplices. The termination condition was also changed to accommodate for the simplices removal.

Another modification was brought by [3] where many optimizations were introduced. We are particularly interested by the separating axes terminating condition. In fact, Van Den Bergen proposed to terminate the GJK iterations when the support direction is a separating direction, i.e. the projections of both objects on this direction are disjoint. This terminating condition can be used when the exact value of the separation distance is not needed, allowing early termination of the iterative procedure.

In summary, the overall structure of GJK based methods is presented in algorithm 3.1. The **SupportVertex(**VL**,** *SupDir***)** function finds VL's vertex in *SupDir*'s direction. It runs in $\mathcal{O}(n)$, where n is VL vertex count. It can be brought down to a constant time if the convex hull is used. The **SupportDirection(***SupDir***,** v_1**,** v_2**)** finds the support direction based on the previous support direction, and on the current support points. As explained above, several criteria were used over the years for obtaining the next support direction and for termination condition.

Algorithm 3.1: General GJK algorithm

input : VL_1 and VL_2 are two point lists
output: minimum separation/penetration distance or a predicate of collision

```
1  begin
2      v1 ← any point of VL1;
3      v2 ← any point of VL2;
4      SupDir ← v2 − v1;
5      Terminate ← 0;
6      while Terminate ≠ 1 do
7          v1 ← SupportVertex(VL1, SupDir);
8          v2 ← SupportVertex(VL2, −SupDir);
9          SupDir ← SupportDirection(SupDir, v1,
           v2);
10         Terminate ← CheckTermination(Result);
11 end
```

Now that we reviewed the related existing methods, we will introduce our algorithm. The main idea is to safely eliminate at every step, the triangles that lie far from the collision region. The triangle elimination algorithm is described in the next section. In fact, you will notice that it has a common part with the general GJK of algorithm 3.1. We will show how to exploit this feature for finding all colliding triangle pairs between the objects.

IV. THE CUT PROCEDURE

The main idea of the proposed method is to eliminate the triangles that are completely outside the potential collision region. A potential collision region exists for each direction. The ideal direction which yields the smallest collision region is usually referred to as minimum penetration direction. Finding such a direction is time consuming, and its good cutting performance does not compensate for the big amount of time needed for the search. It is also effective to use any direction, if it can be easily found and if it allows the removal of non colliding triangles.

A fast way for identifying a potential collision region is by isolating the space delimited by the maximum extents of both

objects on the considered direction. The main function of our method (algorithm 4.1) operates as follows: given TL_1 and TL_2 two triangle lists, and given *CutDir*, the cut direction going from TL_1 to TL_2, it starts by finding the support vertex v_1 of TL_1 in the direction *CutDir*, and the support vertex v_2 of TL_2 in the direction $-$*CutDir*. It removes from TL_1 the triangles having all vertices located before v_2 in *CutDir*'s direction, and it removes from TL_2 the triangles having all vertices located after v_1 in *CutDir*'s direction.

Algorithm 4.1: CutAlongDirection

```
   input  : CutDir is the cut's direction
   input  : TL1 and TL2 are two triangle lists
   output : TL1 and TL2 only contain the potential collision
            region's triangles
1  begin
2      v1 ← SupportVertex(CutDir, TL1);
3      v2 ← SupportVertex(−CutDir, TL2);
4      for every polygon t1 of TL1 do
5          if All t1's vertices are before v2 on CutDir then
6              RemovePolygonFromList(t1, TL1);
7      for every polygon t2 of TL2 do
8          if All t2's vertices are after v1 on CutDir then
9              RemovePolygonFromList(t2, TL2);
10 end
```

The above algorithm runs in $\mathcal{O}(n+m)$ where n and m are TL_1 and TL_2 vertex counts. If the provided meshes present a high vertex sharing level among neighboring polygons, then the vertex number is smaller and the performance is optimal. We are currently investigating the use of adapted data structures for accelerating this process. At the first glance, this expensive operation seems only profitable if it removes a relatively large number of triangles, and thus, one might think that it might slow down the overall process if the objects are deeply inter penetrating (since in such situations, many cuts will not be able to remove a big percentage of the existing triangles). We found out in our experiments, that it is always advantageous to perform the cut, even for deeply inter penetrating objects. The related graph and discussion are presented in section 6 of this paper. This is furthermore practical, since the client application doesn't have to implement a cut decision making scheme any more, thus avoiding further complexity and sources of error.

A sample cut is shown in figure 1. In the first part, the initial positions of the objects are shown, along with the maximum extents of both objects with respect to a chosen cut direction. In the second, we can see the remaining triangles of both objects after the cut is performed.

Both presented algorithms need to perform support vertex calculations. The combined GJK with the cut code of algorithm 4.2 operates as follows: once the support vertices corresponding to the support direction are found, we can use them to perform the cut operation by using the support direction as the cut direction. This reduces the triangles count and accelerates the support vertex operation for the following iterations.

The **CutList(*TL*, *v*)** function performs the elimination of the triangles having all vertices before v. The function does not need to have the cut direction as an input since the dot product for all the vertices was already calculated by the support

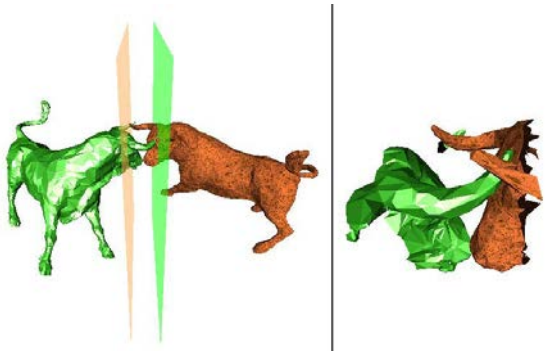

Figure 1: L_1 and L_2 cut with respect to *CutDir*

vertex function. It only needs to compare the saved dot product result with the one stored with v. Although it is not shown in the above algorithm, the code terminates anytime a cut removes all the triangles from a list. For faster performance, our implementation uses a vertex list that contains the triangle lists vertices. The support vertex operation is performed on the vertex list rather than on the triangle list since less dot product operations have to be operated. The formula shown in line 15 of algorithm 4.2 is the one introduced by [1]. The choice of the terminating condition will be explained in the next section.

Algorithm 4.2: Combined GJK with cut

```
   input  : TL1 and TL2 are two triangle lists
   input  : VL1 and VL2 are the point lists of TL1 and TL2
   output : L1 and L2 only contain their common region's
            triangles
1  begin
2      v1 ← any point of VL1;
3      v2 ← any point of VL2;
4      SupDir ← v2 − v1;
5      cut1 ← true;
6      cut2 ← true;
7      while cut1 = true and cut2 = true do
8          v1 ← SupportVertex(VL1, SupDir);
9          v2 ← SupportVertex(VL2, −SupDir);
10         SupDir ← SupportDirection(SupDir, v1,
           v2);
11         cut1 ← CutList(TL1, v2);
12         cut2 ← CutList(TL2, v1);
13         temp = v1 − v2;
14         normalize(temp);
15         SupDir = SupDir − temp × (temp · SupDir) × 2
16 end
```

V. THE RECURSIVE ALGORITHM

The combined GJK algorithm presented above gives a predicate if the convex hulls of the triangles collide. It also keeps in the final triangle lists, only those situated in the collision region. At this point, it is possible to perform a split of both triangle lists in the same manner used for building top-down bounding volume hierarchies. Once the two pairs of lists are produced, we can perform pair wise checks between the produced lists using the combined GJK method in a recursive

manner. In fact, the corresponding operation is an on line partial built of the hierarchy. Every iteration, many non-colliding triangles are removed, and the remaining ones are split into two, and so on, until the lists contain one triangle each. In this case, a triangle-triangle test decides whether the triangle pair should be added to the results list.

Algorithm 5.1 presents the corresponding procedure.

This algorithm has the same overall structure as the recursive collision detection procedure for BVH but presents some key modifications. First we check TL_1 and TL_2 for collision. If they do not collide, we stop the process at this level. If they do collide and they can be split, we check the produced splits against each others using recursive function calls. On the other hand, if the current nodes are leaves, all the contained triangles are checked for collision, and the found colliding pairs are added to the results list. Typically each leaf only contains one triangle.

Algorithm 5.1: RecursiveCheck

```
   input : TL1 and TL2 are triangle lists.
   output: Obtain RL, the list of TL1's and TL2's colliding
           triangle pairs
 1 begin
 2     if CombinedGJKcollide(TL1, TL2) then
 3         if Split(TL1, TL11, TL12) = false And
           Split(TL2, TL21, TL22) = false then
 4             Check all triangles of TL1 against all
               triangles of TL2
 5         else
 6             RecursiveCheck(TL11, TL21);
 7             RecursiveCheck(TL12, TL21);
 8             RecursiveCheck(TL11, TL22);
 9             RecursiveCheck(TL12, TL22);
10 end
```

In algorithm 5.1, when a split operation is not successful, all the triangles of the original list TL_1 will be placed in TL_{11}. TL_{12} will be empty. When **RecursiveCheck(**TL_1**,** TL_2**)** is called on an empty list, it will exit directly with a negative result. This handles the case where only one list could be split.

It is not crucial for the **CombinedGJKcollide()** to remove all non colliding triangles. Those will most probably be removed at later iteration, or in the worst case, during the triangle-triangle test. For this reason, it is not critical to use a non-optimal terminating condition in algorithm 4.2. In fact, the iterations for that algorithm terminate when it fails to perform a list cut. Although it might be possible to remove more triangles if the procedure is not stopped, we chose to stop it because from the moment where a cut is unsuccessful, the next ones have a high probability of failing again. The time consumed by the cut operation is not balanced out by a reduction of the number of triangles, and the cut is actually slowing the overall process instead of accelerating it. Stopping the iterative process will avoid such situations.

If all cuts are unsuccessful, then the algorithm is actually performing the complete construction of the hierarchy plus the CD. Fortunately, such cases never occur. Even if the root nodes present a high inter penetration level, it regresses while descending the tree due to the split operation. The regression speed has the same relation to the configuration of the objects as the depth of tree traversal in standard BVHs.

The split is the operation of separating the original list into two sub-lists. Several splitting heuristics were proposed over the years. In general, they compute a threshold on the splitting direction, and then put all the triangles having their barycenter before this value in the first sub-list, and the others in the second one. A common choice of threshold is the median of all the polygon points. This produces balanced lists. The choice of the splitting direction is also an important one and comes just after the cut in the *dir* direction. Due to the nature of the cut, the remaining triangles usually form oblong shapes. Any orthogonal direction to *SupDir* would make a good splitting direction.

As one can see, the algorithm operates in a single shot: it takes the two triangles lists, and operates on the current positions of their vertices. If the objects deform, the algorithm does not need to perform additional operations. Even if polygons are added or removed from the original objects, the algorithm finds the colliding triangle pairs at that iteration, since it considers the polygons currently in the list, and not those of previous iterations.

VI. RESULTS AND DISCUSSIONS

The tests were conducted on an Intel Centrino 1.7 GHz with 1Gbytes of RAM and an ATI X700 graphics card.

The aim of our first experiments was to determine the applicability of the cut with respect to the percentage overlap region. In other words, we wanted to verify if we should always perform a cut, even when it has a low probability of actually eliminating triangles. As a fast indicator of such a probability, we considered the ratio of the projections on the cut direction (*dir*) of:

- The common region of the two objects delimited by the max_1 and max_2, the maximum values of O_1 and O_2 in dir and $-dir$ respectively.
- The object that need to be cut, delimited by *max* and *min*, the maximum and the minimum of O in dir.

Please note that all the above *max* values are the same that need to be found to perform the cut and thus do not present an extra cost. The *min* search on the other hand, is an additional operation, and will be the major factor for the high computational cost of the decision making process.

We implemented the above ratio in a test version, and allowed the cut to be performed if the overlap is less than the T_{max} . While moving the object pair, we varied T_{max} between 40%, 55%, 70%, 85% and 100% for each position of the objects (where 100% means that the cut is always performed), and we plotted the time needed in each case to find the colliding triangle pairs. The corresponding graph is graph 2.

It is very clear from the graph that it is always beneficial to perform the cut. Except for very few exceptions, the 100% curve is always the fastest. This result can be explained by the fact that the cut operation consists of two main parts: The dot products, and the list split which only consists of pointer assignments. The dot products are by far, the most time consuming. The modified version with the implemented threshold

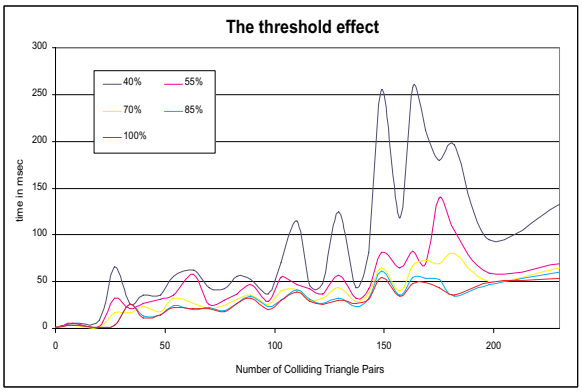

Figure 2: The effect of the threshold on the CUT operation

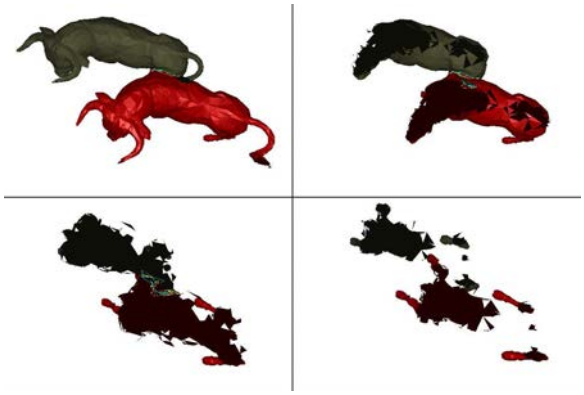

Figure 3: 4 instances of random triangle removal

contains more dot products. In fact, a decision can be made once the maximum and minimum were calculated, while the cut only required calculating the max. To be able to make the decision, the procedure has already consumed the most of the time that the cut would have needed, that performing the cut at this stage will only be a waste if fails to remove any triangles.

The above result is only valid of course, if our decision making criteria is used. However, we did not find a faster choice for the modified GJK method that is capable of justifying the implementation of decision module.

The next test investigates the effect of the on line triangle removal from the objects while performing the collision detection. To be able to compare and scale up the results, we decided to disable the physical engine, and to fix the positions of a pair of bulls in a high collision setup. Such a configuration is non realistic because the physical engine will not allow the inter penetration to happen. Please note that the objects will not be moving with respect to each other, but their triangles will be randomly removed until there are no more left. The plotted timings are the needed ones to find all the colliding triangle pairs. Since the algorithm does not require any additional time to update the hierarchy, those timing show the needed time to perform the complete collision detection process. Initially, there are 258 colliding triangle pairs, but the number decreases when a colliding triangle is removed. The X axis indicates the number of remaining triangle in both objects. Since we are removing the same quantities of triangles at each step, and the objects had initially the same number of triangles, then the number of triangles of each object at each given X value is half of that value. Figure 3 shows 4 instances with decreasing triangles and their collisions, while graph 4 shows the timings plot.

The first observation from graph 4 is the big fluctuations occurring at many instances. Although it might seem contradictory that removing triangles from a pair of fixed position objects is slowing the collision detection process, it has a simple explanation: removing triangles has the same effect as changing configuration. A new configuration implies a new set of cutting directions and iterations, which might be slower than the first one if it follows a more problematic path. This is actually the same effect that involves the standard bounding volume hierarchies, and that makes their behavior highly dependent on their configuration at each iteration. The sudden drop at 8000 triangles happened when most of the colliding triangles were

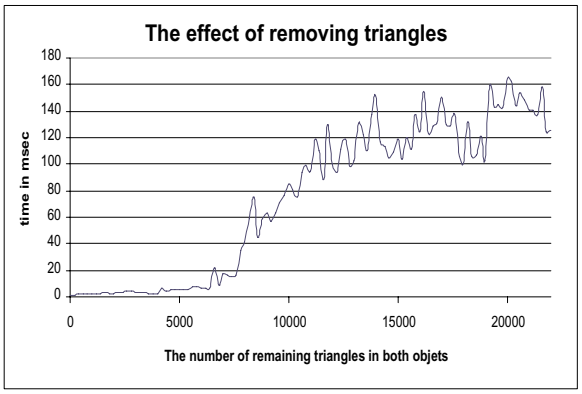

Figure 4: The effect of randomly removing triangles

removed.

Due to the properties of the algorithm and the implementation, the effect of adding triangles is almost exactly the inverse of graph 4 and will not be plotted again.

The next two tests involve two pairs of continuously deforming objects. At every iteration, we applied a wave function to deform all the triangles of both objects. In addition, we were moving the objects with a haptic device to obtain different degrees of collision. These tests were meant to compare the performance of our method to the AABB hierarchies. Since we were unable to find an implementation of the AABB where it is possible to deform the meshes and to update the hierarchies accordingly, we implemented the AABB as presented in [20]. It was brought to our attention that our implementation might be slower than other existing AABB implementations, but the faster implementations we found did not take the deformations into account, and had to rebuild the hierarchies. Graph 5 shows the performance of AABB and the modified GJK on objets of about 1500 triangles each. Graph 6 shows the performance on the pair of bulls of about 13000 triangles each.

As can be seen, the modified GJK is faster than our implementation of the AABB method. The green curve represents the AABB performance including the time it took to refit the hierarchies and is basically a shift of the blue curve that represents the AABB performance while excluding the refit operation.

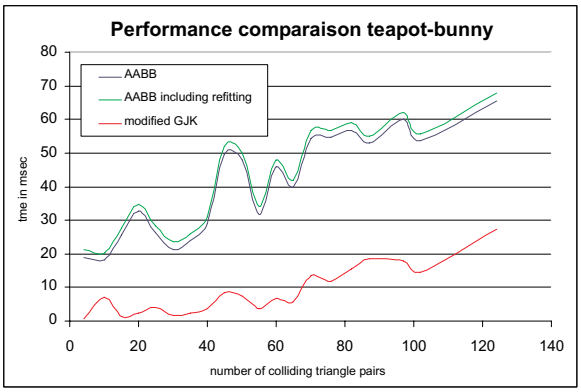

Figure 5: Speed comparaison between AABB and the modified GJK while operating on 1500 triangles objects

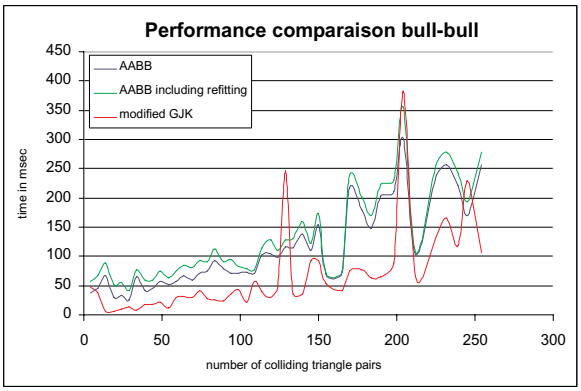

Figure 6: Speed comparaison between AABB and the modified GJK while operating on 13000 triangles objects

VII. CONCLUSION

We presented a modification to the known GJK method, enabling it to be used for exact collision detection by finding all colliding triangle pairs. The new method is suitable for continuously deforming objects, and for adding or removing triangles at run time. It is capable of satisfying high demanding virtual reality application such as haptic simulations dealing with moderate sized objects and requiring the exact list of colliding triangles. Next we will be investigating the use of this method for multi-resolution collision detection: an increase in the resolution can be regarded as the removal of few rough triangles, and the addition of many fine ones. Since the method can easily handle both cases, it is a very good candidate.

ACKNOWLEDGEMENTS The work presented in this paper is sponsored by a ph.D grant to Mr. Hatab offered by the Lebanese National Council for Scientific Research.

REFERENCES

[1] K. Chung and W. Wang. Quick elimination of non-interference polytopes in virtual environments. In *Proceedings of the Eurographics workshop on Virtual environments and scientific visualization '96*, pages 64–73, London, UK, 1996. Springer-Verlag.

[2] J. D. Cohen, M. C. Lin, D. Manocha, and M. Ponamgi. I-collide: an interactive and exact collision detection system for large-scale environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff., New York, NY, USA, 1995. ACM Press.

[3] G. V. den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *J. Graph. Tools*, 4(2):7–25, 1999.

[4] S. A. Ehmann and M. C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. In A. Chalmers and T.-M. Rhyne, editors, *EG 2001 Proceedings*, volume 20(3), pages 500–510. Blackwell Publishing, 2001.

[5] E. Gilbert, D. Johnson, and S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of robotics and Automation*, 4(2):193–203, apr 1988.

[6] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987.

[7] S. Gottschalk, M. Lin, and D. Manocha. Obbtree: a hierarchical structure for rapid interference detection. In *SIGGRAPH'96 Conference Proceedings, Computer Graphics annual conference series*, pages 171–180, New Orleans, aug 1996.

[8] N. Govindaraju, S. Redon, M. Lin, and D. Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop*, 2003.

[9] P. Hubbard. Approximating polyhedra with spheres for time critical collision detection. *ACM Transactions on Graphics*, 15(3):179–209, jul 1996.

[10] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[11] D. L. James and D. K. Pai. Bd-tree: output-sensitive collision detection for reduced deformable models. *ACM Trans. Graph.*, 23(3):393–398, 2004.

[12] P. Jimnez, F. Thomas, and C. Torras. 3D Collision Detection: A Survey. *Computers and Graphics*, 25(2):269–285, apr 2001.

[13] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[14] D. Knott and D. Pai. Cinder: Collision and interference detection in real–time using graphics hardware, 2003.

[15] T. Larsson and T. Akenine-Moller. Collision detection for continuously deforming bodies. In *Eurographics 2001*, pages 325–333, 2001.

[16] M. C. Lin and J. F. Canny. Efficient algorithms for incremental distance computation. In *IEEE International Conference on Robotics and Automation*, volume volume 2, pages 1008–1014, 1991.

[17] C. Mendoza and C. O'Sullivan. An interruptible algorithm for collision detection between deformable objects. In *Workshop On Virtual Reality Interaction and Physical Simulation*, Pisa, nov 2005.

[18] T. Moller and B. Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphic Tools*, 2(1):21–28, 1997.

[19] M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, and W. Strasser. Collision detection for deformable objects. In *Eurographics State-of-the-Art Report (EG-STAR)*, pages 119–139. Eurographics Association, Eurographics Association, 2004.

[20] G. van den Bergen. Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, 2(4):1–13, 1997.