View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UPCommons. Portal del coneixement obert de la UPC

Sphere–Trees Generation As Needed In Real Time To Speed Up Collision Detection

Marta Franquesa Niubó
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Email: marta@lsi.upc.edu

Omar Rodríguez González
Facultad de Ingeniería
Universidad Autónoma de San Luis Potosí
Email: omarg@uaslp.mx

Keywords: Sphere-tree, collision detection, viewing volume, graphics hardware.

Abstract: In this paper two improvements to speed up collision detection are described. Firstly, a method called *on-collide sphere-tree*, OCST for short, is presented. This approach works by detecting collisions among models with arbitrary geometry using the video card's Graphics Processing Units, GPU. While candidate parts of colliding objects are being detected, the OCST is constructed for collision evaluation in parallel, at the same time. Thus, the OCST is created in real–time. Secondly, we have tested two kinds of triangulated representation models for the same original–objects. We have evaluated triangle–soup and triangle–strip models to speed up the algorithm response when computing collisions. The method has been described, implemented and tested for the two kinds of triangulated models, and the obtained results are shown.

1 INTRODUCTION

Collision detection is a key problem in many areas of computer graphics (JTT01; LM03). Considered as a bottleneck within real-time environments, several authors have studied the detection of a collision and multiple solutions have been proposed and published.

It is well known that to compute collision detection among several objects, a bounding easy–shaped wrapper and hierarchies of them are created and used to cover each involved scene–object. These wrappers of simple shape allow us to compute intersections in a quick way, discarding collision faster than using the geometry of the original object models. The problem that arises is the efficient managing of the wrapper hierarchies. As the wrappers are usually called bounding volumes (BV), the hierarchies are called BV–trees. Examples of BV are axis–aligned–bounding–boxes (AABB), oriented–bounding–boxes (OBB) and spheres. The most solutions compute the trees in a preprocessed step and, then, traverse them in a later animation time. These approaches are cumbersome and heavy to manage in the whole process. Thus, the bottleneck of this solution lies in the time step when the new levels of the tree are created, traversed and updated. One of the most commonly BV–hierarchy model used is the sphere–tree.

The hybrid collision detection (KTAK94), refers to any collision detection method that first performs one or more iterations of approximate test to study whether objects interfere in the workspace and then, performs more accurate tests to identify the object parts causing the interference. Hubbard (Hub95) reports two phases: the *broad phase*, where approximate interferences are detected, and the *narrow phase* where exact collision detection is performed. O'Sullivan and Dingliana (O'S99; OD99) extended the classification pointing out that the *narrow phase* consists of several levels of intersection testing between two objects at increasing level of accuracy (*narrow phase: progressive refinement levels*) and, in the last level of accuracy, the tests may be exact (*narrow phase: exact level*). Franquesa and Brunet (FNB03; FNB04) divided the *broad phase* in two subphases. In the first one, tests are performed to find subsets of objects from the entire workspace where collisions can occur, rejecting at the same time, all the space regions where interference is not possible (*broad phase: progressive delimitation levels*). In the second subphase, tests determine the candidate objects that can cause a collision (*broad phase: accurate broad level*). Figure 1 summarizes the complete hybrid collision detection pipeline including all its phases.

In recent times, the availability of high performance 3D graphics cards are common in personal computers. The power and fastness of the built-in Graphics Processing Units, GPU, and its own dedicated memory is being applied to a wider variety of applications, even those that the creators did not originally intend to manage.

In this paper, two improvements are described. Firstly, a method called *on-collide sphere–tree*, OCST, is presented. This approach works by detecting collisions among models with arbitrary geometry using the video card's GPU. While candidate parts of colliding objects are detected, the OCST is constructed for collision evaluation in parallel, at the same time. Thus, the OCST is constructed in real-time. Secondly, we have tested two kinds of triangulated models for the same original–objects. We have evaluated triangle–soup and triangle–strip models to speed up the algorithm response. A triangle–strip is a set of triangles where each triangle shares two vertices with the preceding triangle. The first three indices define a triangle and then each additional index defines another triangle by using the two preceding indices. More detailed information can be found in (RFN05a; RFN05b).

As already mentioned, the whole structures involved in the hybrid collision detection phases have usually been computed as a preprocess to the simulation environment. Before entering the simulation, the structures must be loaded in core memory. We present an algorithm that does not use precomputed BV hierarchical structures, but it uses instead an octree–based sphere–tree created in real-time on needed. The detection of surface overlapping over the sphere–tree nodes is performed making use of occlusion queries by exploiting the capacities of modern graphics hardware. The algorithm is aimed at rigid objects moving in large environments. The *narrow phase* of the hybrid collision detection problem is accelerated. When many objects interact, core memory is managed more efficiently than the other preprocessed approaches. The access to secondary storage is improved when out-of-core techniques are used.

Organization: The rest of the paper is organized as follows. In section 2 the previous related work is discussed. The representation model to be used is detailed in section 3. Description of the OCST construction algorithm is in section 4. OCST collision detection in real-time are discussed in section 5. Experimental results are given in section 6 and, finally, conclusions are presented in section 7.

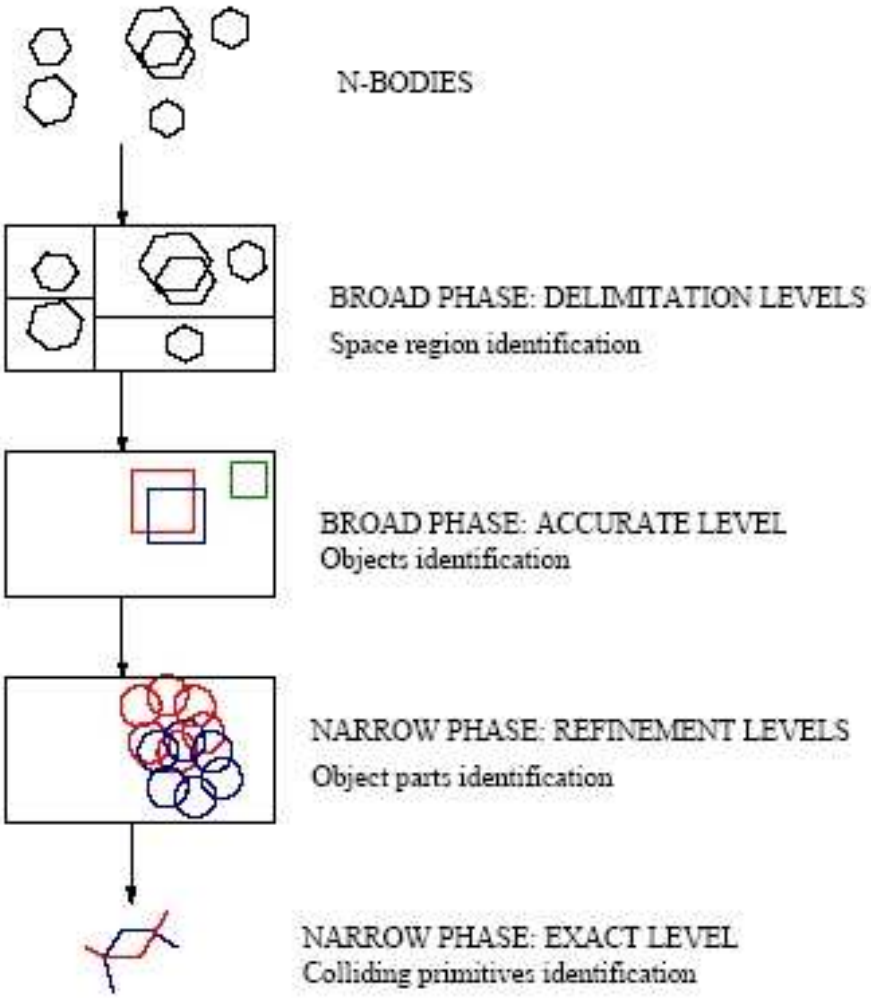

Figure 1: Collision detection pipeline

2 PREVIOUS RELATED WORK

A bounding volume hierarchy approximates a representation of an object as a hierarchical structure, known as bounding volume tree (BVtree). One of the most used BVtrees in the literature is the sphere–tree (Hub93). A sphere–tree represents an object by sets of spheres in a hierarchical way. Three methods are commonly used for the construction of a sphere–tree. The first one, consists of fitting spheres to a polyhedron and shrinking them until they just fit (RB79). The second one is based on an octree (Sam90). Thus, the octree–based sphere–trees (Hub96; OD99; PG95; PSL92) performs a recursive subdivision in 3D, creating spheres on child nodes that overlap the surface of the object. And the third and last, the medial-axis surface method (BO03; Hub95; Hub96; Qui94), uses Voronoi diagrams to calculate the object *skeleton* placing maximal sized spheres on it so the spheres fill the object.

The graphics-hardware-assisted collision detection algorithms started with Shinya and Forgue (SF91), and Rossignac *et al.* (RMS92). After them, a more efficient algorithm was proposed by Myszkowski *et al.* (MOK95) using the stencil buffer. Baciu and Wonk (BW98) were the first to use common available graphics cards to compute image–based collision detection. Vassilev *et al.* (VSC01) use a technique for collision detection in deformable objects like clothes. Kim *et al.* (KOLM03) use graphics hardware to calculate Minkowski sums to find the minimum translational vector needed to separate two interfering objects. All those algorithms involve no precomputation, but perform image-space computations that require the reading back of the depth or stencil buffer, which can be expensive on standard graphics hardware.

Govindaraju *et al.* (GRLM03) use occlusion queries to compute a potentially colliding set (PCS) in the *broad phase*, followed by exact collision in the *narrow phase*. Fan *et al.* (FWG04) use occlusion queries to fast detect collision between a convex object and an arbitrarily shaped object. The advantage of using GPU based occlusion queries is that no read back of the depth or stencil buffer is necessary to obtain results. This kind of tests are faster than image-space computations.

As pointed out in (Kor99), in order to achieve high 3D graphics performance in many applications, it is essential to use triangle strips because they can greatly speed up the display of triangle meshes. Triangle strips have been widely used for efficient rendering. It is NP-complete to test whether a given triangulated model can be represented as a single triangle strip, so many heuristics have been proposed to partition models into few long strips. In the literature we can find many approaches that treat the problem to compute triangle–strips. There are several programs available in the world wide web. One of a common software is the STRIPE[1]. This software is a tool which converts a polygonal model into triangle strips and it is freely available for non commercial use.

3 OCST REPRESENTATION MODEL

To cover each candidate object for collision, octree–based representation for sphere–trees construction is used. As it is well known, an octree is a hierarchical structure obtained subdividing recursively in 3D to form eight child nodes (Sam90; RFN05b). Each one can be represented with three colors. Black color for child nodes completely inside the subdividing object. White for child nodes completely outside. Grey for child nodes in which the frontier of the object overlaps. Grey nodes will be subdivided until a user–defined depth for the octree is reached. When the octree depth is reached, the grey nodes become leaf nodes. An octree–based sphere–tree is an octree where each node is bounded by one sphere instead of a cube. Figure 2 shows an octree–based sphere–tree representation of a dragon.

A conservative collision detection can be performed by using a sphere–tree based on octrees, with a certain depth level. The model gives enough proximity to the object's surface depending on the prespecified user–level.

The cost of creating sphere–trees can be high in terms of computing resources. Space subdivisions require floating–point operations, which are generally slow on CPU. The octree construction requires having the geometry object loaded in core memory aside the sphere–tree structure. Trying to create a sphere–tree on simulation run-time cannot be achieved using only the CPU. Therefore the construction of a sphere–tree has been treated as a precomputation step to the simulation. Having and maintaining all the sphere–tree structures in core memory when many objects are present, can be expensive during the life cycle of a simulation.

From the BVtrees construction methods, the simplicity of octree–based sphere–trees makes it good enough to implement them using graphics hardware (see section 4). The construction of sphere–trees in real-time is performed using occlusion queries. Thus, here, no precomputation is necessary, core memory is free of hierarchical

[1] http://www.cs.sunysb.edu/ stripe

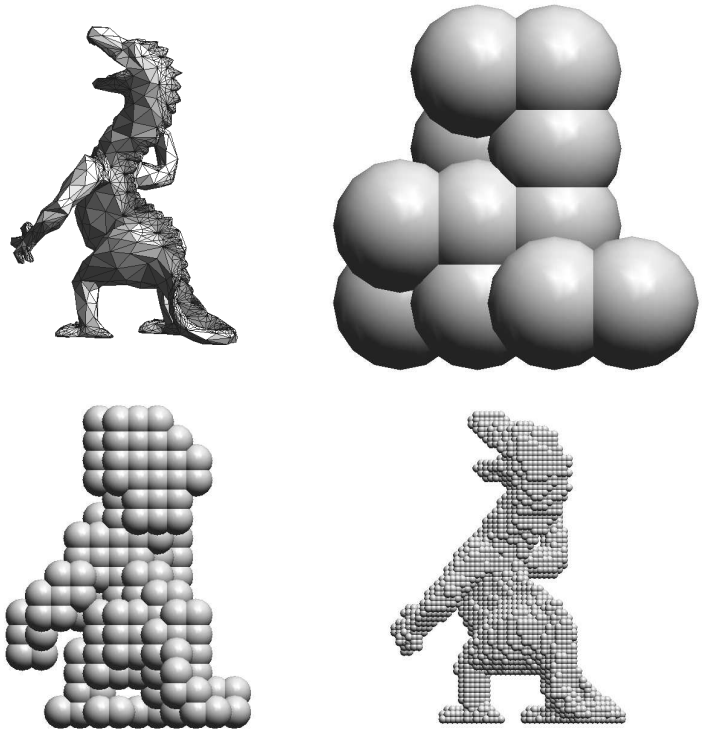

Figure 2: Original object and octree–based sphere–tree levels 2, 4 and 6

structures at the beginning of the simulation because the sphere–trees are created only on-collision when required. To preserve memory, only branches of the sphere–tree for the parts of the objects that potentially can collide are computed. Newly created branches are maintained in core memory for future use during the simulation (see section 5). As we will see in next sections, the use of triangle–strips, instead of triangle–soups, to model the scene–objects increases the efficiency of the whole collision detection system computing the wrapper model in real time on needed.

4 OCST CONSTRUCTION

Different hardware designers have made several occlusion test implementations with differences in performance and functionality. In this way, we can distinguish three types of occlusion queries The first one[2], returns a boolean answer if no incoming object fragment passes the depth test (see Figure 3, where the occlusion query will return *TRUE*). The second one[3], returns the number of fragments that pass the depth test and requires a previous boolean query to be supported by the graphic card. Thus, two queries have to be done to know the one answer. The third and most standard, GL_ARB_occlusion_query[4], is similar to the last mentioned query, but it returns the samples of object parts that occlude directly. It does not require the previous boolean query. Figure 4 shows a case of no occluded object.

The GL_ARB_occlusion_query is used in our method to avoid stalls in the graphics pipeline. This query can manage multiple queries before asking for the result of any one, increasing the overall performance.

In what follows we describe how the oclusion query works, and how our method uses of it. Let A be an arbitrarily shaped object. An OCST root node for A is constructed creating a box for A: *AABB(A)*. A bounding sphere for A is created bounding the *AABB(A)*, with its center as the center of the AABB and its radius as half the

[2]http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt
[3]http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt
[4]http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt

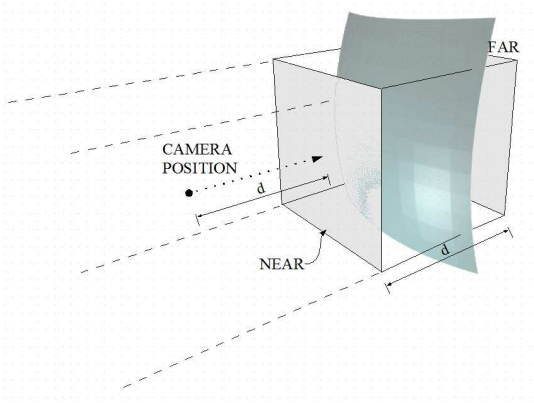

Figure 3: Occlusion queries: Some incoming object fragment passes the depth test.

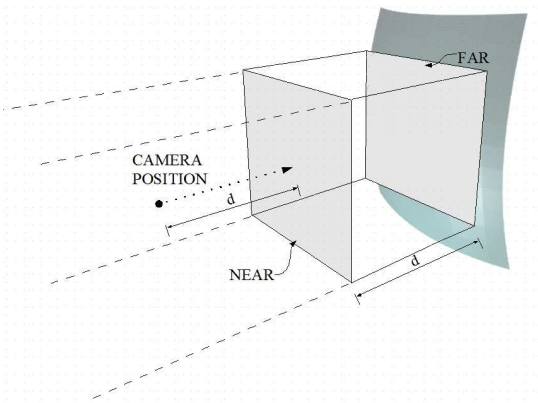

Figure 4: Occlusion queries: No incoming object fragment passes the depth test.

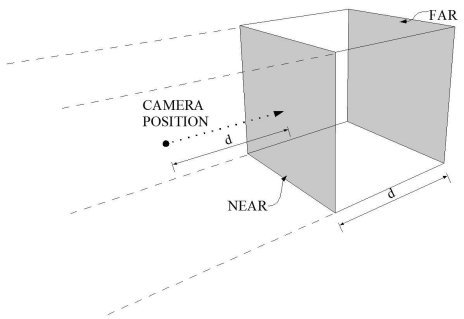

Figure 5: Viewing volume construction: One of the three viewing volumes and its camera position.

distance of the AABB extreme vertices: *S(AABB(A))*. Taking the AABB from the root node of *A*, we construct a new level for the OCST subdividing it in 3D. For each new child node, a resulting octree subdividing AABB box is assigned and an overlap test is performed to verify if it can be a grey node. Occlusion computations are performed to accelerate the overlap test for the detection of grey nodes. These computations are based in the fact that, if the surface of *A* can be viewed in at least some part from inside the AABB of an octree node, then *A* is overlapping the octree node and the node is marked grey (see Figure 3). The overlap test performs one, two or up to three occlusion queries, one for each of the main axis.

Three requirements are needed for each occlusion query (See Figure 5): *A viewing volume*, *a camera position* and *the occlusion test elements*. The viewing volume is created using an orthographic frustum view limited by the AABB box of the octree node tested. The camera position is placed outside the viewing volume, centered at a box face, looking toward the box in parallel to a main axis, and with a distance equal to the length of the box in the looking direction. The first occlusion test element (the occluder), is the AABB box of the octree node. The second occlusion test element (the possibly occluded objects), is the surface of *A* .

An occlusion query reports if one or more occluders allow the possibility that occluded objects can be seen from inside a viewing volume. In other words, if the surface of *A* can be seen from inside the AABB box (viewing volume) of the tested octree node, in at least one of the three main axis, then the surface of *A* is overlapping that octree node. If the number of samples that passed the occlusion query is greater than zero in at least one of the three queries (for *x*, *y* and *z* axis), then the surface of *A* overlaps the tested OCST node and it it is marked grey. In this case, a sphere is created bounding the AABB box of the node and is inserted on the OCST structure.

5 OCST AND REAL-TIME COLLISION DETECTION

To achieve collision detection in real-time, OCST branches are constructed for objects only when it is needed. Thus, to initialize the animation system it is only required to load the geometry of the objects into graphics card's memory, and construct a root OCST for each of them at the beginning of the simulation. The OCST root is initialized with an AABB and a bounding sphere with the center as the center of the AABB, and its radius as half the distance of the AABB extreme vertices.

Let *A* and *B* be arbitrarily shaped objects in movement. The two objects collide with each other only, if the distance between their root sphere centers is equal or less than the sum of their respective radius (See Figure 6). When a collision occurs, one level is constructed for the OCST for objects *A* and *B*. If child nodes of object *A* collide with child nodes of object *B*, an additional level is constructed only for the colliding child nodes. This process continues up to a user–defined depth for the OCST. When the depth value is reached, and two leaf nodes collide, a collision between object *A* and *B* is reported. Using a bigger depth value, the approximation to the object surface is tighter, and the collision detection is more accurate.

All hierarchies sphere centers must be updated with the objects in movement. When a new level for the OCST is created the number of updates increases. With a big user–defined depth value the maintaining cost of updating all the animation OCSTs is higher. To found the potentially colliding set, PCS, the sphere interference test

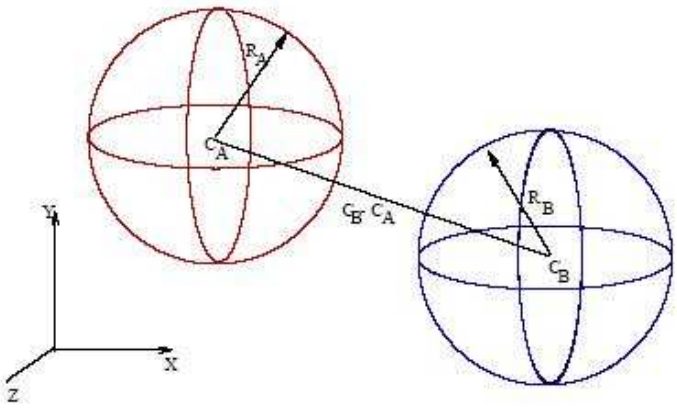

Figure 6: Spheres *A* and *B* collide *iff* $d \leq R_A + R_B$, where $d = |C_B - C_A|$.

described below is used. A list with pair–colliding spheres is computed and used to identify interfering object parts. In large environments (FN04; RFN05c), the PCS can be obtained using algorithms designed for the *broad phase* of the hybrid collision detection problem.

To increase the algorithm performance, the branches of the OCST created by older collisions are kept in core memory. These can be re-used on forthcoming collision tests. To avoid the problem of a high computing resource cost caused for hierarchies updates, a time-stamp is assigned to the deeper OCST nodes. If a complete OCST level does not participate in a collision during a certain amount of time, it is deleted from core memory and the parent initialized with its own time-stamp. This will cause an object to get back to its initial state (only the OCST root node is kept), if it is not involved in any more collisions during a certain amount of time (This is the case of the Cow and the Dragon of Figure 8).

6 EXPERIMENTAL RESULTS

In this section some relevant results of applying our method are exposed. To compare the actual results with existing others, the input data tested in other existing algorithms has been selected. Explanations about simulations and results can be found in (RFN05a). The algorithms have been implemented on a Dell Inspiron notebook with ATI Mobility Radeon 9600 graphics card with 128 MB VRAM and a Pentium M processor at 1.80 GHz. The algorithms were tested with commonly used complex models[5]. Figure 7 shows the models used.

6.1 OCST Construction Timings

The time to construct one level of an OCST is exposed in Table 1. This time is equal for the two models used, triangle–soup and triangle–strips. The results are obtained with the objects already loaded in graphics card's memory as triangles regardless of with the kind of triangulation chosen, triangle–soup or triangle–strip. The Table shows the number of triangles for each model, the time used to construct the level (in seconds) and the number of occlusion tests performed.

The complete model has to be rendered for each occlusion test. Note that the object's geometry does not affect the time of constructing one OCST new level. The algorithm performance is affected only for the number of occlusion tests and the time each one lasts. Therefore, the worst case only occurs when all occlusion tests have to be considered, for all the nodes and axis. In this case, with eight possible child nodes and three tests per each one, for a total of 24 occlusion tests, the maximum experimented time was resulted equal to 0.03 seconds. For the simplest model the construction of an OCST level using only the CPU can take from 0.03 seconds, 0.1 to 0.5 seconds for the intermediate models, and 1 second or more for the largest models. Without the use of the GPU for the construction, the object's geometry does indeed affect the algorithm performance. The optimizations such as

[5]http://isg.cs.tcd.ie/spheretree/

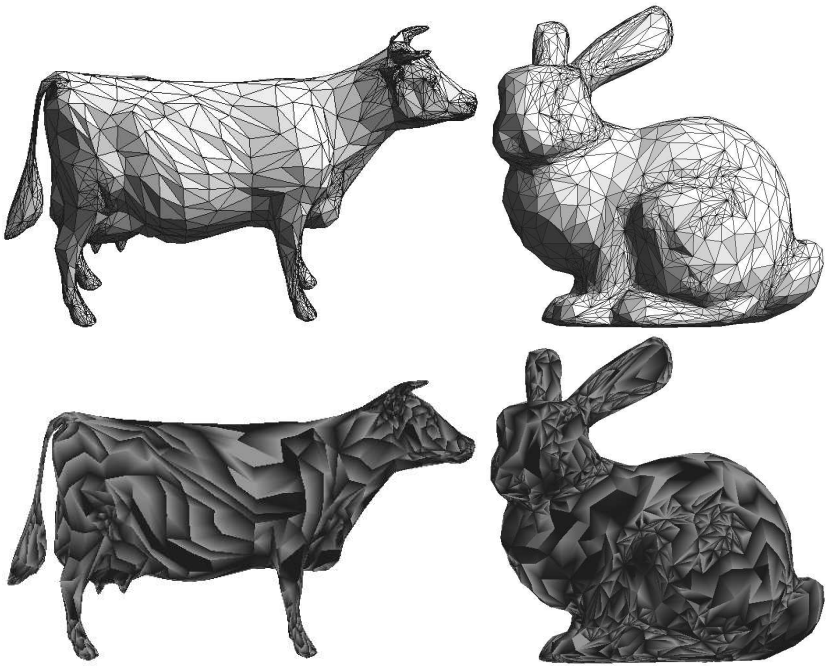

Figure 7: Two examples of input data models. Top: *Cow* and *Bunny* modeled by triangle–soup, with 5144 and 5110 triangles respectively. Down: Same input objects modeled by using triangle–strips.

Model	Triangles	Time	Occlusion
Dragon	1496	0.0099	13
Bunny	1500	0.0099	9
Cow	1500	0.0099	9
Lamp	600	0.0199	13
Dragon	5104	0.0199	13
Bunny	5110	0.0099	9
Cow	5144	0.0099	9

Table 1: OCST construction time

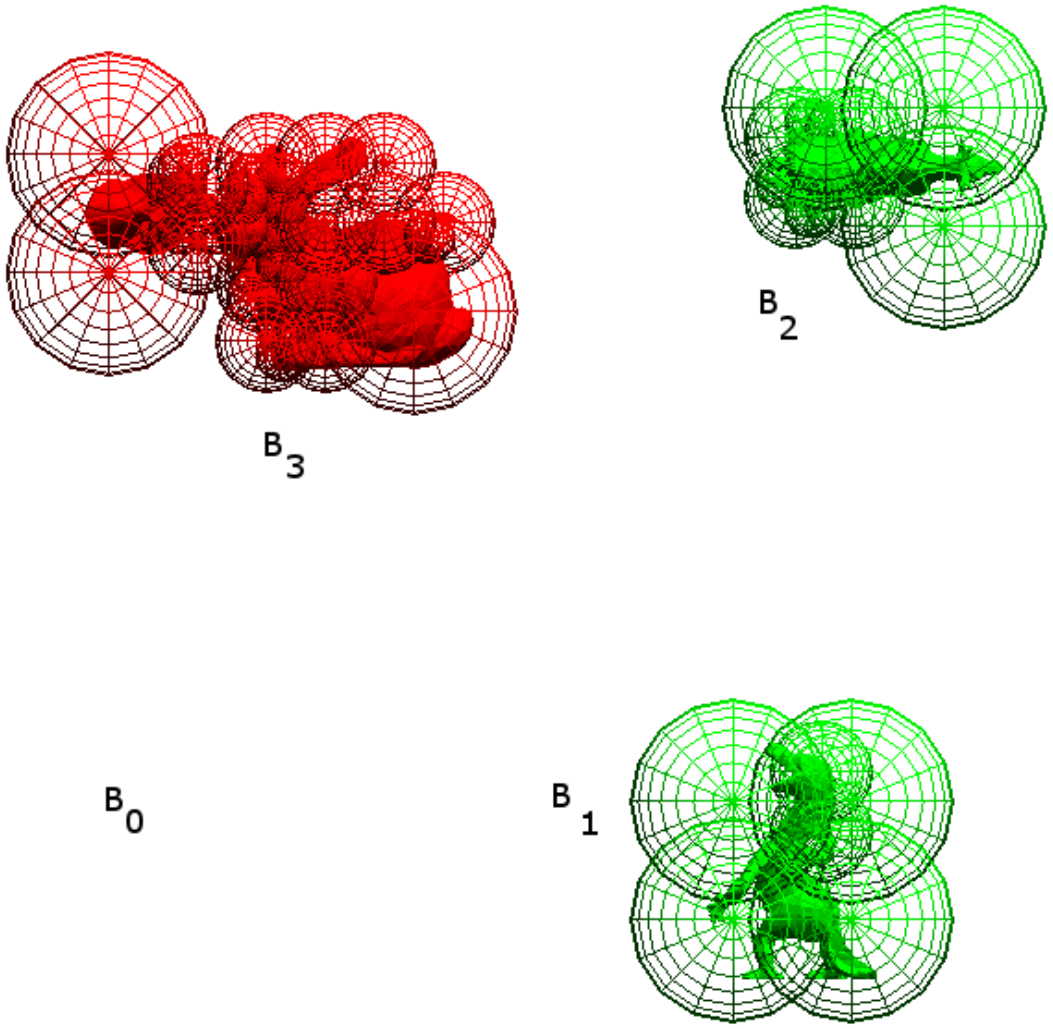

Figure 8: Example of a collision detection fragment animation: Snapshot where one object, the *Bunny* is moving and the other three objects are still.

triangle–strips have proved useful to accelerate the render of the complete model. Therefore, using triangle–strips is faster for the OCST construction, as it is shown by the results in the next section.

6.2 OCST Collision Detection Performance

The algorithms were tested with a scenario where one object follows a fixed trajectory in a 3D space. Collision occurs among the other three objects. Figure 8 shows a snapshot of an example of collision simulation: The initial location of the *Bunny* is B_0. Then following a trajectory, it passes through B_1, B_2, and B_3 (place where the snaptshot has been taken). In B_1 the *Bunny* collided with the dragon, in B_2 the *Bunny* collided with the *Cow* and in B_3 the *Bunny* is colliding with a lamp. While the collision is being detected, new levels of the respective objects trees are generated. When the collision is false, the tree is going up to the root node, deleting all the nodes. This last reason is the key why the dragon and the cow are sorrounded by big spheres, because the trees are going up. Table 2 and Table 3 show the performance for the animation, and the time taken to finish it. The tables use triangle–soup and triangle–strip respectively to render the models and represent the same animation. In the two situations, a user–defined depth level for the OCST equal to 5 is used. The results are measured in frames–per-second (FPS). The number of occlusion queries performed in each time step is also shown. Comparing the results

Time	FPS	Occlusion queries
1.00	177.64	198
2.00	244.76	700
3.00	274.45	316
4.00	245.75	694
5.00	264.47	490
6.00	283.72	294
7.00	294.71	182
8.00	268.46	754

Table 2: Animation performance using triangle–soup models. Total animation time: 8.582 sec.

Time	FPS	Occlusion queries
1.00	232.53	240
2.00	278.72	832
3.00	325.35	372
4.00	297.70	690
5.00	310.38	532
6.00	348.65	208
7.00	311.69	884

Table 3: Animation performance using triangle–strip models. Total animation time: 7.360 sec.

exposed in the tables, we can conclude that the use of triangle–strips is better than the use of triangle–soups. Note for example the number of frames per second at 4 sec. of the animation time, with 694 occlusion queries for triangle–soup and 690 queries for triangle–strip and the FPS is 245.75 in the first case and 297.70 in the second case. Then the better ratio of FPS, indicates better results for triangle–strips.
Looking at Table 2 and Table 3 and taking into account that the simulation trajectory was the same in both cases, the time of the whole animation for triangle–soup was equal to 8.582 sec, while for triangle–strip was 7.360 sec.

FPS slowed down only, when new levels for the OCSTs are generated. Although the speed of the FPS gets lower, the rate keeps on being good enough. Therefore, the animation can be maintained over 60 FPS and allows a smooth transition between frames in visual terms.

The worst case occurs when objects are moving very fast and a straight collision occurs. This situation can cause several levels of the OCST tree have to be constructed at once for each colliding object. In this case, the performance could slow down. Even so, stalls in the animation can occur, only, if the user–defined depth value is too high. Even though, these stalls are due to the high number of occlusion tests that have to be performed to construct all the OCST branches, the running time is not affected when real-time simulations are computed.

Figure 9 shows the sequence of a collision between two objects, modeled by using triangle–strips, with the OCSTs created in real-time up to level 5. The red colour indicates that a collision has been detected.

7 CONCLUSIONS

In this paper, two collision detection improvements are described. Firstly, a new method that has been conceived to speed up the collision detection pipeline has been introduced. Its application in real-time environments has been implemented using OCST. The method is fast enough to manage collision detection in real-time, as it can be seen from the experimental results exposed. The speed and efficiency obtained with our method enables us to manage many concurrent objects in a scene. Secondly, we have tested two kinds of triangulated models for the same set of original objects: triangle–soup and triangle–strip. Triangle–strip are shown to be better model in terms of sphere–tree time computing. And, as a consequence, they are better when computing collision detection.

The method's limitations are related to hardware constrictions. The overall performance is affected by several parameters. The amount and speed of the video memory built–in the graphics cards, the bus transfer speed and the clock frequency of the GPU. Other existing methods that use out-of-core algorithms in real-time could be degraded at reading time from secondary storage, and at sending time of the object's geometry to the graphic card memory.

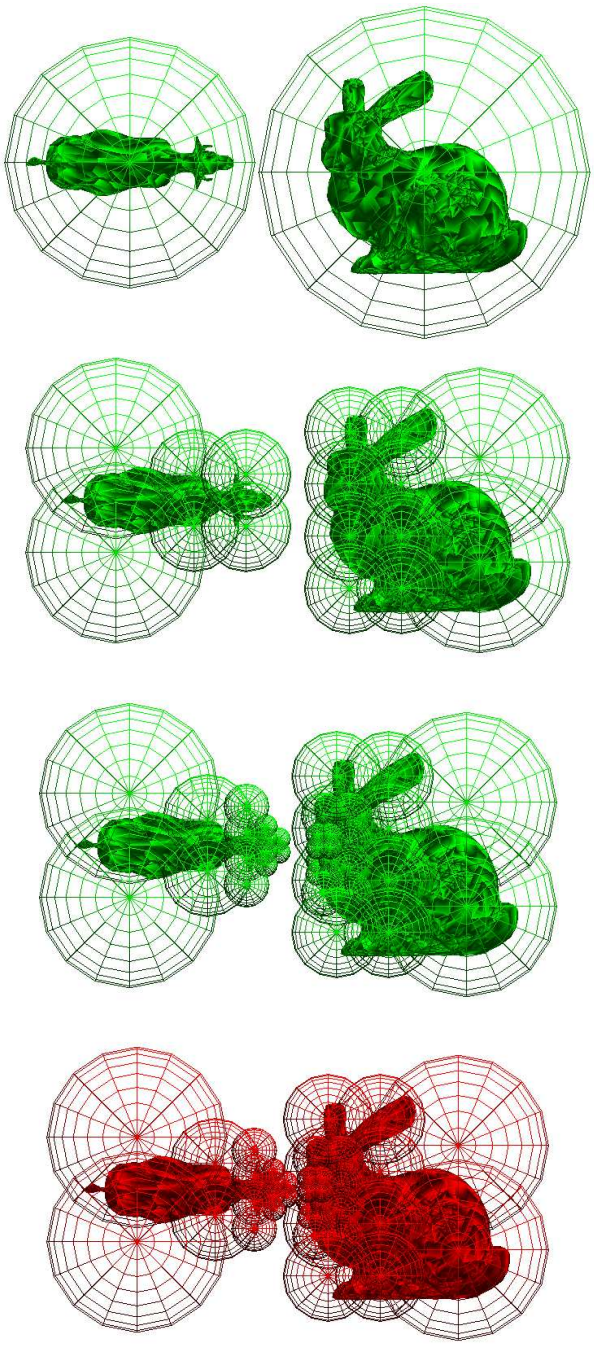

Figure 9: OCST creation up to level 5

The amount of the model representation to be generated is reduced with the use of OCST, while decreasing considerably the collision detection time without loss of accuracy.

We have detailed here, an algorithm related to the *narrow phase* of the collision detection pipeline problem. However, work related to the *broad phase* can be found in (FN04; RFN05c). We are working on bringing together both methods, so a fully functional fast collision detection system for large environments could give us better results on our application environments.

ACKNOWLEDGEMENTS

This research has been partially supported by the Ministerio de Ciencia y Tecnología under the project MAT2002-0497-C03-02 and by the CREBEC, Centre de Referencia en Enginyeria Biomèdica de Catalunya, and by the Facultad de Ingeniería de la Universidad Autónoma de San Luis Potosí under the PROMEP program.

REFERENCES

G. Bradshaw and C. O'Sullivan. Adaptative Medial–Axis Approximation for Sphere–Tree Construction. *ACM Transactions on Graphics*, 22(4), 2003.

G. Baciu and S.G. Wonk. Recode: An image–based collision detection algorithm. In *Proc. of Pacific Graphics*, pages 497–512, 1998.

M. Franquesa-Niubó. *Collision Detection in Large Environments using Multiresolution KdTrees*. PhD thesis, Universitat Politècnica de Catalunya, March 2004.

M. Franquesa-Niubó and P. Brunet. Collision Detection Using $MKtrees$. In *Proc. CEIG 2003*, pages 217–232, July 2003.

M. Franquesa-Niubó and P. Brunet. Collision Prediction Using MKtrees. In R. Scopigno and V. Skala, editors, *WSCG 2004, The 12–th International Conf. in Central Europe on Comp. Graphics, Visualization and Comp. Vision 2004*, volume 1, pages 63–70, February 2004. Plzen. ISSN 1213–6972.

Z. Fan, H. Wan, and S. Gao. Simple and rapid collision detection using multiple viewing volumes. In *VRCAI '04: Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*, pages 95–99. ACM Press, 2004.

Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 25–32. Eurographics Association, 2003.

P. M. Hubbard. Interactive collision detection. In *Proc. IEEE Symp. on Research Frontiers in Virtual Reality*, volume 1, pages 24–31, October 1993.

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

Philip M. Hubbard. Aproximating polyhedra with spheres for time–critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

P. Jimenez, F. Thomas, and C. Torras. (3D) Collision Detection: A Survey. *Computers and Graphics*, 25(2):269–285, August 2001.

Young J. Kim, Miguel A. Otaduy, Ming C. Lin, and Dinesh Manocha. Fast penetration depth estimation using rasterization hardware and hierarchical refinement. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 386–387. ACM Press, 2003.

David Kornmann. "fast and simple triangle strip generation". *VMS Finland, Espoo, Finland. Color Plates*, 1999.

Y. Kitamura, H. Takemura, N. Ahuja, and F. Kishino. Efficient collision detection among objects in arbitrary motion using multiple shape representation. In *Proceedings 12th IARP Inter. Conference on Pattern Recognition*, pages 390–396, October 1994.

M.C. Lin and D. Manocha. *Handbook of Discrete and Computational Geometry Collision Detection*, chapter 35. CRC Press LLC, 2003. To appear.

K. Myszkowski, O. G. Okunev, and T. L. Kunii. Fast collision detection between computer solids using rasterizing graphics hardware. *The Visual Computer*, 11, 1995.

C. O'Sullivan and J. Dingliana. Real–time collision detection and response using sphere–trees. In 15th Spring Conference on Computer Graphics, April 1999. ISBN: 80–223–1357–2.

C. O'Sullivan. *Perceptually–Adaptive Collision Detection for Real–time Computer Animation*. PhD thesis, University of Dublin, Trinity College Department of Computer Science, June 1999.

I.J. Palmer and R.L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 1995.

A.P. Del Pobil, M.A. Serna, and J. Llovet. A new representation for collision avoidance and detection. In *IEEE Int. Conf. on Robotics and Automation (Nice)(France)*, volume 1, pages 246–251, May 1992.

S. Quinlan. Efficient distance computation between non–convex objects. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 3324–3329, 1994. San Diego, CA.

J.O. Rourke and N. Badler. Decomposition of three–dimensional objects into spheres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–1(3):295–305, July 1979.

O. Rodríguez and M. Franquesa-Niubó. A New gpu Based Sphere–Tree Generation Method To Speed Up The Collision Detection Pipeline. Technical report, Software Dept. LSI. U.P.C., 2005. Ref: LSI–05–45–R. http: //www.lsi.upc.edu/dept/techreps/techreps.html.

O. Rodríguez and M. Franquesa-Niubó. A New Sphere–Ttree Generation Method To Speed Up The Collision Detection Pipeline. In *Proceedings of CEIG'05, September 2005. Granada. Spain*, 2005.

O. Rodríguez and M. Franquesa-Niubó. Hierarchical Structuring Of Scenes With MKTrees. Technical report, Software Dept. LSI. U.P.C., 2005. Ref: LSI–05–4–R. http: //www.lsi.upc.edu/dept/techreps/techreps.html.

Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider. Interactive inspection of solids: cross-sections and interferences. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 353–360. ACM Press, 1992.

H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990. ISBN 0-201-50255-0.

M. Shinya and M. Forgue. Interference Detection Through Rasterization. *Journal of Visualization and Computer Animations*, 2:131–134, 1991.

T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. In *Computer Graphics Forum*, volume 20(3), pages 260–267, 2001.