BOCST: Branch On-collide Sphere-trees

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

Marta Franquesa Niubó
Departament de Llenguatges
i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona, España
marta@lsi.upc.edu

October 13, 2005

Abstract

In this paper, a fast sphere-tree generation method used for collision detection called Branch On-collide Sphere-trees is proposed. Using the video card graphic processing unit (GPU), a sphere-tree is constructed in real-time inside an animation. With this method, the core memory usage is minimized because no pre-computed data is loaded at any time during simulation life cycle.

With our method, real-time conservative collision detection is achieved using the GPU, core memory is managed efficiently and the error is lowered using fast-construction sphere-tree structures.

Keywords: sphere-tree construction, collision detection, viewing volume, graphics hardware

1 Introduction

Collision detection has been considered as a bottleneck within real-time environments, because the high CPU usage involved. At animation run-time, the CPU is needed to update the bounding volume (BV) hierarchies loaded in core memory in a preprocessed step. When many objects are in the scene, the cost to update all the needed hierarchies are high in computer resources. Even so, having loaded in core memory all the BV hierarchies from the used models can imply high memory requirements.

Several authors have studied the key areas for collision detection usage [LG98, JTT01]. Proposals for the collision detection problem has been based over the Kitamura *et al.* [KTAK94] hybrid collision detection work. 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 *narrow phase* with the *narrow phase: progressive refinement levels* and the *narrow phase: exact level*. Franquesa and Brunet [FNB03, FNB04] extended the *broad phase* with the *broad phase: progressive delimitation levels* and *broad phase: accurate broad level*. For more information in the hybrid collision detection problem refer to [RF05c].

The power and fastness of the recent video card GPUs and its own dedicated memory has been applied to a wider variety of applications. Using the video card GPU, we have developed a method to detect collision detection in simulation run-time without the need of precomputed hierarchies. In this paper, we focus in the *narrow phase: progressive refinement levels*. However, work related to the *broad phase* has been researched in [RF05a].

Main contribution: In a common simulation, the structures involved in the hybrid collision detection phases (space subdivision hierarchies and BV hierarchies) have been approached as a construction preprocess to the simulation run-time. Before entering the simulation, all the structures involved must be loaded into core memory. In simulation run-time, the updates to the BV hierarchies take place according to the position and orientation of the moving objects.

We present a method that doesn't use precomputed BV hierarchical structures, instead uses a fast sphere-tree created in real-time called Branch On-collide Sphere-trees (BOCST for short). The method uses GPUs occlusion queries to construct the hierarchy in real-time and is oriented for moving rigid objects. With our method, the *narrow phase* of the hybrid collision detection problem is accelerated. When many objects are present inside the simulation, the core memory can be managed more efficiently because the nature of the BOCST.

Organization: The rest of the paper is organized as follows. In section 2 we discuss the state of the art in related areas. The description of the representation model to be used is detailed in section 3. Generation methods for the BOCSTs are explained in section 4. Simulations, performance and results are given in section 5 and conclusions are presented in section 6.

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 [PG95, Hub96, OD99] 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 [Qui94, Hub95, Hub96, BO03], 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 [MM91], 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.

Due that no previous papers mentioned above solved the problem to construct a BVtree in real-time for the use of arbitrarily shaped objects, we present a new method that efficiently achieve this task in real-time. Then, Rodríguez and Franquesa [RF05b] used the GPU to construct an octree-based sphere-tree so no precomputation is required. 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.

3 BOCST Description

A BOCST, is a sphere-tree constructed in real-time via the graphics card GPU. It has the following features:

- Based on a sphere-tree hierarchy
- Constructed in animation run-time when is needed
- It has a low memory stamp and it's structure can be created/deleted as needed
- Constructed using the video graphics GPU

- The updated cost in animation run-time of the structure is kept to minimum

The BOCST is constructed as needed. That means that at the beginning of the animation only the root node of the BOCST exists in memory. When an initial collision is detected, the required branches of the involved trees are created. In this way the core memory is managed efficiently. Therefore, the tree is going up when it is needed. Thus, if in a certain amount of time, a branch of the tree is not involved in any collision test, the branch is deleted from memory. Then, the space in memory is the minimum needed at any time.

The construction takes place using the object's geometry used by the graphics card. The model is loaded in the video card memory. So the model doesn't need to be loaded at core memory at any moment. Using GPU occlusion queries, the tree is constructed while a collision is detected up to an user-defined level.

Compared to precomputed structures, that require that the entire tree nodes exists in core memory for each object, the BOCST only has the tree nodes needed to detect a collision. With this, less CPU usage is needed to perform updates to the BOCST structure.

3.1 Representation Model

Each node of a BOCST is represented as:

- A pointer to the parent node
- A List of pointers to the children nodes
- A pointer to the original object
- An integer n, where 2^n represents the dimensions of a bounding cube for the node
- The radius of the bounding sphere
- The center of the bounding sphere

The BOCST structure is maintained in core memory and updated as needed. The dynamic structure can increase or decrease in size depending on the frame-collision coherency inside the animation. A time-stamp is assigned to the deeper BOCST nodes. If a complete BOCST level doesn't 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 that a BOCST gets back to its initial state (only the root node). Then, the whole system runs faster because discarded branches are not carried out by the animation.

4 BOCST Generation

Two methods have been developed and tested to create BOCSTs. The first is based on classic octrees and the second one is based on BONOs. The last one, shows major improvements in speed and memory management in real-time construction inside an animation (see subsection 4.3).

To construct a BOCST, OpenGL occlusion queries must be supported by the graphics card. The occlusion query operation is performed via the GPU and its result is read and processed by the methods.

4.1 Occlusion Query Operations

Different hardware designers have made various occlusion test implementations with differences in performance and functionality. In this way, the first occlusion query that we can find[1], returns a boolean answer if any incoming fragment passes the depth test. The second one[2], returns the number of fragments that pass the depth test but requires that the first query be supported by the graphic card. The third and most standard, GL_ARB_occlusion_query[3], is similar to the last named query, but does not require the first query to be available.

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.

4.2 Octree-based BOCST (O-BOCST) Generation

An octree is a tree of degree eight which represents the space occupied by objects contained in a space. A detailed description and operations involving octrees can be found in [BJN92].

4.2.1 O-BOCST Construction

Let A be an arbitrarily shaped object. A BOCST root node for A is constructed creating an axis-aligned bounding box (AABB) for A. A bounding sphere for A is created bounding the AABB from A, with its center as the center of the AABB and its radius as half the distance of the AABB extreme vertices. Taking the AABB from the root node of A, we construct a new level for the BOCST 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.

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

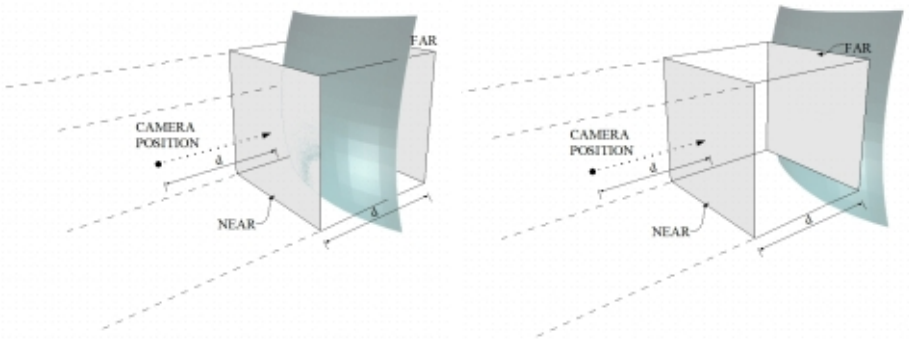

Figure 1: Viewing volume construction and occlusion query test: left, the occlusion query returns the number of samples that passes the test inside the viewing volume; right, the occlusion query returns zero

To accelerate the overlap test for the detection of grey nodes, occlusion queries are performed. Based in the observation 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.

The overlap test performs one, two or up to three occlusion queries for each of the main axis. The requirements for each occlusion query are:

- a viewing volume
- a camera position
- 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 that occluded objects can be seen from inside a viewing volume. That is, 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. Figure 1 illustrates one of the three possible viewing volumes and its camera position and the.

Algorithm 1 illustrates the overlap test. If the number of samples that passed the occlusion query is greater than zero in at least one of the three queries,

```
set occlusion query buffer;
render AABB box of the tested octree node;
clear depth buffer;
disable color and depth buffer;
disable cull face;
while not overlap found do
    select one of the three frustum views;
    set occlusion query;
    render surface of A;
    end occlusion query;
    get occlusion query results;
    if samples passed the test > 0 then
        overlap with surface of A is found;
    else
        if all queries finished then
            no overlap found;
        else
            select next frustum view;
        end if
    end if
end while
enable cull face;
enable depth and color buffer;
```

Algorithm 1: GPU based overlap test

the surface of A overlaps the tested octree node and it's a grey node. If it's a grey node, a sphere is created bounding the AABB box of the node and is inserted on the BOCST.

4.3 BONO-based BOCST (B-BOCST) Generation

Similar concepts for the construction of O-BOCSTs are used to construct the B-BOCST.

The branch-on-need octree (BONO) was introduced by Whilhelms and van Gelder in [WvG92]. The BONO is a data structure based on octrees that delay the subdivision of the space until absolutely necessary. The BONO associates each node to a conceptual region and an actual region. The BONO strategy is that the lower subdivision in each branching direction always covers the largest possible exact power of two. Whit this, BONOs only requires integer space subdivisions so simple shift operations are only required. Using BONOs, the CPU dependent subdivisions are accelerated. For more information about BONOs, see [WvG92].

4.3.1 B-BOCST Construction

The process to construct a B-BOCST is similar to the process to construct an O-BOCST, but a BONO method is used instead of an octree (see section 4.2). The BONO method is well known so here we will detail only some implementation aspects to consider in the usage of BONOs.

Let A be an arbitrarily shaped object. A BOCST root node for A is constructed creating an axis-aligned bounding box (AABB) for A. A bounding sphere for A is created bounding the AABB from A, with its center as the center of the AABB and its radius as half the distance of the AABB extreme vertices. Taking the AABB from the root node of A, we construct a new level for the BOCST subdividing it in 3D with the BONO method. For each new child node, an AABB box is assigned and an overlap test is performed to verify if it can be a grey node.

The same algorithm 1 is applied for the BONO-based overlap test to detect grey nodes.

The B-BOCST has several advantages over the O-BOCST:

- BONOs are used as the sphere-tree, resulting in faster construction times
- Faster construction time, because only integer operations are required for the BONO constructio
- Less BOCST nodes and lower memory requirements, due to the nature of the BONO method
- Easier to maintain in run-time animation

However, there are some disadvantages on the B-BOCST method:

- In some cases, the BONOs behave equals to an octree
- Less spheres means increased error in object tightness

However, this disadvantages can be easily resolved as we'll see on later sections (see subsection 5.3).

4.4 BOCST Construction Considerations

With an octree-base BOCST, a bounding cube covering O (with O = any object) is created with dimensions d_t, with $d_t = Max(Proj(AABB(O), d_x, d_y, d_z))$. Independently of the value of d_t, an O-BOCST is always constructed the same.

With B-BOCST, the construction varies depending on the value of d_t because a space subdivisions with the BONO algorithm occurs at 2^n (where

Method	O-BOCST	B-BOCST		O-BOCST	B-BOCST	
Levels	6	6	7	6	6	7
d_t	514 $(2^n + 2)$			768 $(2^n + 2^{n-1})$		
Total Nodes	4457	1068	4453	4457	2406	10065
Nodes $l = 1$	1	1	1	1	1	1
Nodes $l = 2$	8	2	2	8	4	4
Nodes $l = 3$	42	9	9	42	16	16
Nodes $l = 4$	193	44	44	193	99	99
Nodes $l = 5$	819	192	192	819	436	436
Nodes $l = 6$	3394	820	820	3394	1850	1850
Nodes $l = 7$			3385			7659
d_t	896 $(2^n + 2^{n-1} + 2^{n-2})$			1023 $(2^{n+1} - 1)$		
Total Nodes	4457	3324	13787	4457	4404	18120
Nodes $l = 1$	1	1	1	1	1	1
Nodes $l = 2$	8	7	7	8	8	8
Nodes $l = 3$	42	30	30	42	43	43
Nodes $l = 4$	193	142	142	193	192	192
Nodes $l = 5$	819	593	593	819	812	812
Nodes $l = 6$	3394	2551	2551	3394	3348	3348
Nodes $l = 7$			10463			13716

Table 1: Number of spheres for the *bunny* with $n = 9$ (l = number of constructed levels)

$n = nbits(d_t) - 1$ and $nbits()$ = number of bits of the integer part of d_t). This causes that when $(2^n + 2) < d_t < (2^{n+1} - 1)$ (being $n = 1...x$), the B-BOCST representation takes less spheres to cover the original object than the O-BOCST with the same depth level. When $d_t = 2^{n+1}$, the B-BOCST behaves similar to the O-BOCST with the same depth level. Table 1 and table 2 show the number of spheres for some examples.

In the case that $(2^n + 2) < d_t < (2^{n+1} - 1)$, the B-BOCST requires less spheres to cover the original object, the construction is faster and less memory is required for the BOCST. In the other hand, the tightness error increases. See section 5.2 for a description of the error measure in different cases.

A trade-off is caused between fastness and object tightness. This can be corrected inside the animation, varying the value of the user defined level depending of the value of d_t for each O in the scene. For more information in animation performance and real-time BOCST corrections see subsection 5.3.

5 Simulations and Results

This section is divided in several categories: input data description shows the model properties used in all simulations; error description explains the

Method	O-BOCST	B-BOCST		O-BOCST	B-BOCST	
Levels	6	6	7	6	6	7
d_t	514 $(2^n + 2)$			768 $(2^n + 2^{n-1})$		
Total Nodes	2216	546	2265	2216	1219	5113
Nodes $l = 1$	1	1	1	1	1	1
Nodes $l = 2$	8	2	2	8	5	5
Nodes $l = 3$	26	9	9	26	14	14
Nodes $l = 4$	102	27	27	102	59	59
Nodes $l = 5$	406	101	101	406	221	221
Nodes $l = 6$	1673	406	406	1673	919	919
Nodes $l = 7$			1719			3894
d_t	896 $(2^n + 2^{n-1} + 2^{n-2})$			1023 $(2^{n+1} - 1)$		
Total Nodes	2216	1716	7079	2216	2232	9281
Nodes $l = 1$	1	1	1	1	1	1
Nodes $l = 2$	8	6	6	8	8	8
Nodes $l = 3$	26	19	19	26	26	26
Nodes $l = 4$	102	83	83	102	100	100
Nodes $l = 5$	406	310	310	406	401	401
Nodes $l = 6$	1673	1297	1297	1673	1696	1696
Nodes $l = 7$			5363			7049

Table 2: Number of spheres for the *dragon* with $n = 9$ (l = number of constructed levels)

method used to calculate the error in a BV hierarchy according to the model it covers; O-BOCST vs B-BOCST simulations shows the performance in a real-time animation for each method; and finally, the results subsection compares all the results for both methods.

5.1 Input Data Description

Common data found on the web has been used to test the methods[4]. A process of stripification[5] has been applied to this models, so an optimized model results in better performance inside the animation. Figure 2 shows some of the models used in the tests.

5.2 Error Description

To measure the tightness for the spheres to the object surface of each method an error has been calculated. The error is related to the accuracy of the conservative collision detection, and is computed as follows:

- The error is fixed measuring the distance between the surfaces contained inside a bounding sphere and the bounding sphere

[4]http://isg.cs.tcd.ie/spheretree/
[5]http://www.cs.sunysb.edu/ stripe/

Figure 2: Example of input models: left to right: *bunny* with 5110 triangles, *dragon* with 5104 triangles, *lamp* with 600 triangles and *cow* with 5144 triangles

- The maximum error is equal to 1.0, being this case when the root sphere is tangent to the object surface: $MaxError = diam(sphere_{root})$
- $Error = Max(dist(O_i, sphere_i))/MaxError$, where $sphere_i$ = BVtree leaf nodes and O_i = objects inside $sphere_i$
- The minimum and average error is calculated too
- The error is determined only with the leaf nodes of the BVtree

To compute a correct value for the error, a complete octree-based and BONO-based BOCST have been constructed and leaf nodes have been analyzed. Some results are showed in figure 3 and figure 4.

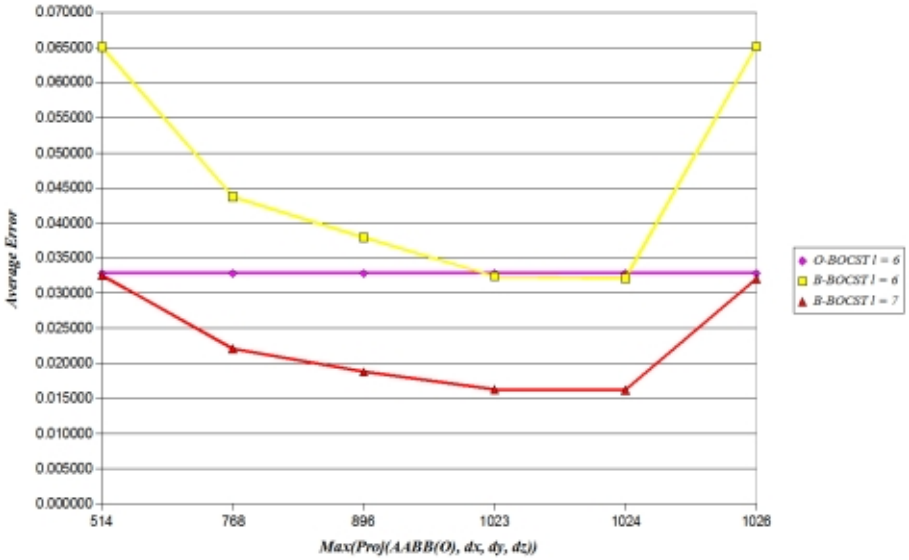

Figure 3: Average error for the bunny (l = user-defined depth level)

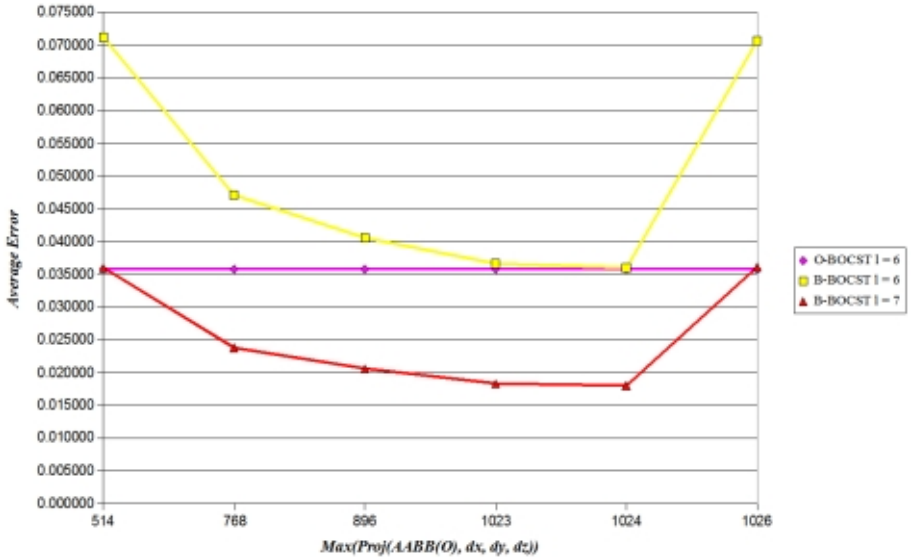

Figure 4: Average error for the dragon (l = user-defined depth level)

5.3 BOCST Simulation Parameters

Several parameters affect the performance of the simulations. Some of them are:

- *Collision depth*: number of levels to construct while the method detects a collision. In other words, the hierarchy is generated up to the *collision depth level if the interference between objects has not been discounted before.*
- *Max. time*: time-stamp for the deeper nodes of the BOCST.
- *Anim. frames*: number of frames for a fixed animated route.
- d_t: maximum dimension for the bounding box of O.

The *collision depth*, *max. time* and *anim. frames* are user-defined. The value for d_t is variable depending on the object dimensions.

The *collision depth* parameter affects on the performance and the accuracy of the algorithm. With a low value, the accuracy is poor but the performance is good, because less levels have to be constructed. With a high value, the accuracy is good but the performance is poor.

The *Max. time* parameter affects on the performance, the usage of core memory and collision-coherency. With a low value, the performance is better and the core memory usage is low because less spheres need to be maintained in animation run-time, but the collision-coherency is poor, which causes that

	O-BOCST	B-BOCST
Coll. depth	5	5, 6
Max. time	2.0	2.0
Anim. frames	2200	2200
d_t	514, 832, 1024	514, 832, 1024

Table 3: Parameter values

levels used in the animation have to be constructed continuously. With a high value, the performance diminishes and the core memory increases, but the collision-coherency is better.

The *Anim. frames* parameter affects on the performance too. With a low value, the animation takes less steps to finish, so the objects move faster. This causes that a faster moving object collide with another, and several levels of the BOCST have to be constructed at once, which can cause that a stall in the animation occurs. With a high value, the objects movement in the animation is slow, but levels of the BOCST are constructed one at a time with no penalties in the performance.

The d_t parameter is variable and affects on the performance and the accuracy of the BOCST methods (see section 4). To test with several scenarios, we have modified the value of d_t scaling all objects to known dimensions. When $(2^n + 2) < d_t < (2^{n+1} - 1)$ (being $n = 1...x$), less spheres are constructed for the B-BOCST compared to the O-BOCST with the same depth level (l) but the accuracy is lower for the first. When $d_t = 2^{n+1}$, the B-BOCST and the O-BOCST behaves similar so the performance is low but the accuracy increases. To alleviate this problem, the following considerations are taken into account for the B-BOCST method:

- If $(2^n + 2) \leq d_t < (2^n + 2^{n-1})$ then $l = l + 1$
- If $(2^n + 2^{n-1}) \leq d_t \leq (2^{n+1})$ then $l = l$

5.4 BOCST Performance and Results

To compare the results of both BOCST methods, the following parameter values in table 3 are used.

For both BOCST methods, optimized model are used (see section 5.1). An animation is setup and it moves an object with 5000 triangles over three objects on a precalculated trajectory. Table 4, table 5 and table 6 shows the animation results. The tables show the time both methods took to finish it, the frames per second (FPS) in each animation second and the number of occlusion queries performed by each method.

	O-BOCST $l = 5$ 6.869 segs.		B-BOCST $l = 5$ 6.479 segs.		B-BOCST $l = 6$ 6.609 segs.	
Time	FPS	Occ. queries	FPS	Occ. queries	FPS	Occ. queries
1.0	192.81	0	260.74	0	258.74	0
2.0	310.38	524	328.34	368	295.41	702
3.0	337.66	254	364.64	64	354.65	166
4.0	355.29	144	370.26	0	367.27	0
5.0	371.63	22	371.63	0	369.63	0
6.0	370.63	0	366.27	28	367.27	28

Table 4: Animation results with $d_t = 514$

	O-BOCST $l = 5$ 7.120 segs.		B-BOCST $l = 5$ 6.970 segs.		B-BOCST $l = 6$ 7.7512 segs.	
Time	FPS	Occ. queries	FPS	Occ. queries	FPS	Occ. queries
1.0	243.76	148	247.75	86	234.53	50
2.0	274.45	902	298.40	644	227.77	1270
3.0	329.67	412	336.66	208	262.74	1064
4.0	304.39	588	312.38	412	303.39	588
5.0	340.66	328	353.65	130	266.73	908
6.0	368.63	8	348.30	132	370.26	0
7.0	335.33	742			329.67	326

Table 5: Animation results with $d_t = 832$

	O-BOCST $l = 5$ 7.881 segs.		B-BOCST $l = 5$ 7.250 segs.		B-BOCST $l = 6$ 8.752 segs.	
Time	FPS	Occ. queries	FPS	Occ. queries	FPS	Occ. queries
1.0	177.64	178	225.77	234	185.63	70
2.0	279.72	828	296.70	742	209.79	1410
3.0	311.38	388	319.36	514	228.77	1402
4.0	290.71	740	303.70	518	303.39	538
5.0	314.69	532	336.33	344	234.77	1246
6.0	347.31	208	333.67	384	216.57	1454
7.0	323.68	754	330.34	406	345.65	226
8.0					288.42	848

Table 6: Animation results with $d_t = 1024$

6 Conclusions

The BOCST methods to approximate objects in 3D has been presented in this paper. Compared with others, the BOCST is faster and better. It is created as it is required in the collision detection animation run-time and not in a preprocessed step, like other methods. The subtrees of the BOCST actual node are computed only for the parts of the objects that are involved in a collision. The method has been tested for a variety of data sets and scenarios with different performance parameters.

In the future, we will continue researching with improved BVtree methods for better object tightness to achieve better collision detection. As well, collision prediction and exact collision detection are pending subjects in this paper.

Acknowledgements

This research has been partially supported by the Ministerio de Ciencia y Tecnología under the project MAT2002-0497-C03-02 and the Facultad de Ingeniería of the Universidad Autónoma de San Luis Potosí under the PROMEP program.

References

[BJN92] P. Brunet, R. Juan, and I. Navazo. Octree representations in solid modeling. *Progress in Computer Graphics*, 1:164–215, 1992.

[BO03] G. Bradshaw and C. O'Sullivan. Adaptative Medial–Axis Approximation for Sphere–Tree Construction. *ACM Transactions on Graphics*, 22(4), 2003.

[BW98] G. Baciu and S.G. Wonk. Recode: An image–based collision detection algorithm. In *Proc. of Pacific Graphics*, pages 497–512, 1998.

[FNB03] M. Franquesa-Niubo and P. Brunet. Collision detection using *MKtrees*. In *Proc. CEIG 2003*, pages 217–232, July 2003.

[FNB04] M. Franquesa-Niubo 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.

[FWG04] 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.

[GRLM03] 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.

[Hub93] P. M. Hubbard. Interactive collision detection. In *Proc. IEEE Symp. on Research Frontiers in Virtual Reality*, volume 1, pages 24–31, October 1993.

[Hub95] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, September 1995.

[Hub96] Philip M. Hubbard. Aproximating polyhedra with spheres for time–critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[JTT01] P. Jimenez, F. Thomas, and C. Torras. (3d) collision detection: A survey. *Computers and Graphics*, 25(2):269–285, August 2001.

[KOLM03] 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.

[KTAK94] 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.

[LG98] M.C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In *Proc. of IMA Conference on Mathematics of Surfaces*, 1998.

[MM91] Shinya M. and Forgue M. Interference detection through rasterization. *Journal of Visualization and Computer Animations*, 2:131–134, 1991.

[MOK95] K. Myszkowski, O. G. Okunev, and T. L. Kunii. Fast collision detection between computer solids using rasterizing graphics hardware. *The Visual Computer*, 11, 1995.

[OD99] 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.

[O'S99] 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.

[PG95] I.J. Palmer and R.L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 1995.

[Qui94] 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.

[RB79] 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.

[RF05a] O. Rodríguez and M. Franquesa. 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.

[RF05b] O. Rodríguez and M. Franquesa. A new gpu based sphere-tree generation method to speed up the collision pipeline. In *Proc. CEIG 2005*, pages 137–145, September 2005.

[RF05c] O. Rodríguez and M. Franquesa. A new sphere-tree generation method to speed up the collision detection pipeline. Technical report, Software Dept. LSI. U.P.C., 2005. Ref: LSI–05–23–R. http: //www.lsi.upc.edu/dept/techreps/techreps.html.

[RMS92] 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.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990. ISBN 0-201-50255-0.

[VSC01] T. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. In *Computer Graphics Forum*, volume 20(3), pages 260–267, 2001.

[WvG92] J. Wilhelms and A. van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.