Point in Solid Tests for Triangle Meshes. Comparative Study.

Carlos J. Ogayar; Rafael J. Segura; Francisco R. Feito

Dpto. de Informática - Universidad de Jaén - 23071 - Jaén - España
{cogayar,rsegura,ffeito}@ujaen.es

Abstract
This document presents a comparative study of several point inclusion tests for triangle meshes. We also discuss some issues about the usefulness of each method depending on the situation, taking into account memory and CPU usage. The goal is to compare the performance of well known spatial classification algorithms with a modified implementation of Feito-Torres, optimized to work only with triangle meshes. A description of studied methods is also presented, with implementation issues and time tables showing the performance of each algorithm. The tests highlight both weak points and virtues of each approach.

1. Introduction

There are lots of applications which use triangle meshes due to their simplicity and versatility. Every object can be represented or approximated using triangle meshes, even more when using 3D hardware acceleration. Point in solid test is a very common task in Computer Graphics (face and edge classification, collision detection, CSG, etc.). Several well known methods are used with B-rep representations (specially triangle meshes) like Jordan curve theorem based algorithms (which use ray-casting); other methods work with a spatial structure like BSP, octree and grid. Each algorithm has different efficiency, complexity and memory requirements, so no one should be considered as the best for all situations. One of the goals of this document is to compare the most used point inclusion algorithms with a Feito-Torres[2] modified implementation applied to triangle meshes.

2. Commonly used algorithms survey

2.1. Jordan curve theorem based algorithm

The most used schemes for both 2D and 3D point inclusion testing are based in the Jordan curve theorem, due to its simplicity and the fact that no precomputed data are needed. This method works by casting a ray from the point to be tested following a random direction and therefore intersecting the solid several times (or none). The result of the test depends on the intersections count parity. However, sometimes this process must be repeated (changing the direction of the ray) when a special case is detected, i.e., when a vertex or edge intersection, a face included point or a coplanar face are detected in the process. As it should be expected, the used ray-triangle intersection algorithm will clearly affect the overall performance of the method. You are encouraged to use the algorithm from Badouel[1] or Möller-Trumbore[7]; both use barycentric coordinates.

2.2. Spatial segmentation based point in solid tests

There are other methods based on a spatial structure like grid, BSP or octree. They have a weak point: preprocessing, which usually supposes high memory and CPU requirements.

2.2.1. Grid

A grid based algorithm uses an uniform space segmentation, which results in a voxel space. Voxel is short for 'volumetric pixel', and each voxel represents a regular volume of space. These cells can be totally or partially included in the solid. In order to get enough precision in the solid representation, a high definition grid is needed; in addition, partial inclusion states must be taken into account. When using binary voxels (only total inclusion/exclusion allowed) aliasing appears, specially at low resolution levels. Nevertheless, the main problems of using a grid for inclusion testing are the choice of the voxel inclusion test algorithm, and the waste of space due to uniform (non adaptive) segmentation. As an advantage, the grid traverse algorithm is very fast.

2.2.2. Octree

Problems arising with grids may be partially solved using an octree [8]; moreover, an octree can be considered as a particular case of a grid. This time, the structure is adapted to the solid spatial distribution (it is adaptive). The octree construction begins with the bounding cube (not box) of the solid. This is the first octant (the root node), which is subdivided into eight suboctants. This process is repeated until each octant is completely inside or completely outside of the solid, or other condition is achieved (e.g.: a given depth is reached). Of course, the inclusion state of each octant must be calculated using some method, like Jordan based algorithm. This method greatly saves memory, and above all, reduces the number of inclusion tests while building the structure, because there are less octants to test than voxels in the equivalent grid. As with grids, there are problems related to aliasing and the performance of the point inclusion test used in octant processing. The octree traversing is not as efficient as the grid traversing; however, there are efficient algorithms which dramatically reduce this difference.

2.2.3. BSP

BSP (Binary Space Partitioning) [3, 6] consists of consecutive space subdivisions using planes. The subdivision is done recursively, so each subspace is divided again, giving as a result a spatial classifying tree. Two types of BSP are used in Computer Graphics: axis aligned BSP and polygon aligned BSP. The first one is not very suitable for inclusion tests, because it does not offer an exact representation of the solid (it has the same problems as octrees and grids). On the other hand, the second one is appropriate for these tests, because it uses the planes where solid faces are included as splitter planes. If a polygon is intersected by the current separating plane during the subdivision process, it must be divided, and each side has to be added to the corresponding subspace. This task is repeated until all the planes containing a face of the solid have been used. Obtaining an efficient BSP tree is a time and memory demanding task. To classify a point as inside or outside a solid using a BSP, it is enough to traverse the tree testing the position of the point with regard to the plane stored in each node. This process is recursively repeated until a leaf node is reached.

2.3. Feito-Torres algorithm

Both Jordan based and spatial classification based algorithms have an extra cost associated with the complexity of the triangle mesh. This is because of the special cases (non-valid intersections) in the first ones and the memory requirements in the second ones. Feito-Torres method[2] does not need any precomputing and it does not depend on the shape of the solid. On the other hand, there are some optional optimizations (related to precomputing) which dramatically enhance the performance. In addition, Feito-Torres algorithm can easily treat convexities, holes and multiresolution representations.

Feito-Torres works by considering the solid as made up of original tetrahedra and calculating the sum of the signs from the original tetrahedra where the given point is inside. Next, the triangle adapted version is summarized. For each triangle of the solid a tetrahedron is built using the origin of the coordinate system as an additional vertex; this tetrahedron is called an original tetrahedron. Then, all the original tetrahedra that contain the point to be tested are selected (tetrahedra may overlap), and their signs are calculated. Taking the sum of the signs, the point is classified as inside (sign>0) or outside the solid (sign<=0). Original tetrahedra are used because the sign of a tetrahedron is computed by solving a determinant, and the determinant related to an original tetrahedron is very fast to solve. Please, check [2] for details.

3. Implementation

The main goal of this paper is to compare the results of some algorithms under the same conditions. For this reason, an homogeneous implementation was done (all code is C++). There are some object oriented data types and structures, which are common to all algorithms; moreover, all of them use the same macros and tools. The implemented and tested algorithms are Jordan based method, octree-optimized Jordan based method, Feito-Torres, optimized Feito-Torres and a BSP based inclusion algorithm. No other spatial subdivision algorithms were selected (Grid and Octree), because they do not give exact results, unless combined with something else.

3.1. Jordan based algorithm

In order to apply Jordan curve theorem, Möller-Trumbore[7] ray-triangle intersection algorithm has been used, but it was modified to detect all special cases: a ray-vertex or ray-edge intersection is found, or the ray and the triangle are coplanar. This variant introduces a little performance penalty, but it is necessary in order to keep the algorithm robustness. Each time a special case appears, all the process has to be done again using a different random ray. Indeed, this is the weak point of Jordan based algorithm, which constrains to use a high numerical precision.

The bottleneck of Jordan based method is the ray-triangle intersection algorithm. In order to enhance the performance, an octree has been used as a spatial optimizer in the optimized version. Note that this octree is only used to discard unnecessary ray-triangle intersections as it is usually done in a ray-tracing task, so do not mistake it for the inclusion based octree described before. In fact, both octree implementations are the same, but this time, each leaf node keeps a list of triangles that intersect the related octant[5]. Of course, it is possible for a triangle to be in several lists, but it does not matter. To traverse the octree, Gargantini traversal method[4]

			Feito-Torres (opt)		Jordan+octree6		BSP	
	vertices	triangles	time	memory	time	memory	time	memory
Celtic Cross	1849	2366	<0.001s	61Kb	0.079s	135Kb	0.001s	408Kb
Hydrant	15822	5786	<0.001s	150Kb	0.188s	270Kb	1.282s	4.6Mb
Battery	4763	9522	<0.001s	285Kb	0.281s	401Kb	1.500s	720Kb
Mobile Phone	13025	25946	0.015s	674Kb	0.578s	281Kb	54.514s	158Mb
Golfball	23370	46205	0.031s	1.2Mb	1.188s	861Kb	72.484s	2.7Mb
V8 engine	76214	152553	0.132s	3.9Mb	4.469s	2.1Mb	514.141s	576Mb
Sculpture	139217	277512	0.235s	7.2Mb	6.687s	1.4Mb	442.109s	328Mb
Seat	282535	564960	0.502s	14.7Mb	15.219s	3.5Mb	mem_error	mem_error
Dragon	437645	871414	0.782s	22.5Mb	27.766s	4.2Mb	mem_error	mem_error

Table 1: *Models data and their associated preprocessing cost. Time unit is second.*

is used; with this method, the list of triangles to be intersected is dramatically reduced; moreover, the deeper the octree, the smaller the triangle list. Indeed, with an efficient octree traversal method, it is much better to use an octree instead of a grid, because it saves a lot of memory.

3.2. BSP based algorithm

BSP based algorithm implementation is quite conventional. The planes where solid faces are located are used to split the space. If a dividing plane cuts any triangle, its associated plane is added in both resulting subspaces; this produces a huge tree when processing complex models, specially when concavities are found. The implemented BSP construction algorithm uses no recursion; it uses a stack instead, which stores the branching state at every moment. This iterative version is, by far, faster than the recursive version, and it also uses less memory. The first implemented version of the algorithm (which was recursive) collapsed all memory resources with a not too complex solid. Moreover, the implemented iterative BSP navigation is faster too.

3.3. Feito-Torres algorithm

The implementation of Feito-Torres algorithm uses only an auxiliary function which computes the inclusion of a point in an original tetrahedron. The main function calculates the sum of the signed volumes of all the original tetrahedra which include the given point. The optimized version has a data structure where much of the required data during the inclusion testing are stored for each original tetrahedron; there is no need for recomputing these data when changing the point to be tested.

4. Results

The tests were done with 9 models, with a polygonal complexity in the range from ∼2500 to ∼875000 triangles. Some of them have lots of concavities and holes, which make them very suitable to probe all solid shape dependent methods. Table 1 shows the characteristics of the used models. The algorithms were tested on a 2GHz Intel Pentium 4 PC with 512MB of RAM, working with Windows XP, which is a standard home computer at this moment. Table 2 shows the results.

- Jordan based algorithm: This algorithm does not work well when there is a high vertices and polygons density, specially when concavities appear, as in V8 model (see tables 1 and 2). In situations like these, intersection computations must be repeated changing the outgoing vector (the direction of the ray) until no special cases are detected. The octree-optimized version (we used a depth value of 6 in the tests) dramatically enhanced the performance, greatly improving execution times, although it needed some precomputing time and a moderate amount of memory.

- BSP based algorithm: With no doubt, this is the fastest spatial classification algorithm. The problem is the extremely high preprocessing cost when dealing with very complex models, both in time and memory (some tests collapsed the system, even with an iterative imple-

	Feito	Feito(op)	Jordan	Jdn(op)	BSP
Celtic Cross	0.875s	0.063s	0.969s	0.021s	0.001s
Hydrant	2.188s	0.172s	2.407s	0.031s	0.002s
Battery	3.578s	0.375s	3.907s	0.029s	0.007s
Mobile Phone	9.672s	1.359s	10.953s	0.172s	0.041s
Golfball	17.907s	2.406s	19.841s	0.062s	0.367s
V8 engine	55.152s	7.469s	639.841s	0.359s	0.167s
Sculpture	119.372s	13.910s	129.712s	1.031s	0.008s
Seat	237.971s	31.409s	264.324s	1.219s	error
Dragon	336.879s	45.942s	3759.402s	1.781s	error

Table 2: *CPU times for a 1000 point inclusion test. Time unit is second.*

	Feito	Feito(opt)	Jordan	Jordan(opt)	BSP
Efficiency	good (6)	quite good (7)	moderate (5)	very good (8)	excellent (9)
Preprocessing time	none (9)	lowest (8)	none (9)	moderate (5)[1]	very high (2)[2]
Preprocessing memory	none (9)	moderate (5)	none (9)	moderate (6)[1]	very high (2)[2]
Solid shape dependency	none (9)	none (9)	moderate (5)	moderate (5)	extreme (0)
Scalability (best case)	excellent (9)	excellent (9)	quite good (7)	very good (8)	excellent (9)
Scalability (worst case)	excellent (9)	excellent (9)	poor (2)[3]	poor (3)[3]	very poor (1)[4]
Difficulty of implementation	very low (9)	low (8)	low (7)	normal (6)	normal (5)

Quality ranking values range from 0 (worst) to 9 (best)
[1] It depends on the depth of the octree — [2] It also depends on the shape of the solid
[3] Due to special cases in intersection computations — [4] Due to concavities and holes

Table 3: *Features of the implemented point in solid methods.*

mentation). BSP is very solid shape dependent.

- Feito-Torres algorithm: According to the tests results, this method appears to be the most stable, scalable and efficient point inclusion algorithm without any preprocessing. Test times are slightly better than the best cases of Jordan based algorithm; about -10%. Feito-Torres is solid shape independent, so it maintains a constant linear relation with the number of polygons, no matter the number of concavities or holes the model could contain. Optimized version of Feito-Torres greatly lowers execution times (about -85% from the standard version with almost all solids) with an insignificant preprocessing time cost, although a moderate amount of memory is required.

As a conclusion, a comment about each algorithm suitability must be done (see Table 3). Feito-Torres is the best algorithm without precomputing. Intermediate solutions in resource requirements are the optimized Feito-Torres and the octree based Jordan (with a medium depth). Without memory limits and lots of points to be tested (which compensates preprocessing), BSP is by far the best choice.

What is clear is the inverse relation between memory use (preprocessing cost, in general) and performance when classifying points. The fastest algorithms need a data structure, sometimes a huge structure, whereas algorithms which work without precomputing or extra memory are not so fast. Consequently, memory requirements are a key restriction for choosing an inclusion method.

5. Conclusions

This work presents an evaluation of several point inclusion methods for triangle meshes, plus a review of the capabilities of each one. Feito-Torres algorithm is very robust and stable, moreover, it does not depend on the shape of the solid. Indeed, it appears to be more efficient than the Jordan based algorithm as long as no precomputing is done. Bearing in mind implementation easiness, Feito-Torres is an excellent alternative to current existing algorithms.

Acknowledgements

This work has been partially granted by the Ministry of Science and Technology of Spain and the European Union by means of the ERDF funds, under the research project TIC2001-2099-C03-03.

References

1. Badouel, D. An Efficient Ray-polygon Intersection, in Graphic Gems. *Andrew S. Glassner, Academic Press*, pp. 390–393, 1990. 1

2. Feito, F.R. and Torres, J.C. Inclusion Test for General Polihedra. *Computer & Graphics*, **21**(1):23–30, 1997. 1, 2

3. Fuchs, H., Kedem, Z.M., and Naylor, B.F. On Visible Surface Generation by A Priori Tree Structures. *Computer Graphics (SIGGRAPH '80 Proceedings)*, pp. 124–133, 1980. 2

4. Gargantini, I. and Atkinson, H.H. Ray Tracing an Octree: Numerical Evaluation of the First Intersection. *Computer Graphics Forum*, **12**(4), 1993. 2

5. Glassner, A.S. Space subdivision for fast ray tracing. *IEEE Comput. Graph. Appl.*, **4**(10):15–22, October 1984. 2

6. James, A. Binary Space Partitioning for Accelerated Hidden Surface Removal and Rendering of Static Environments. *Ph.D. Thesis, University of East Anglia*, August 1999. 2

7. Möller, T. and Trumbore, B. Fast, minimum storage ray-triangle intersection. *Journal on Graphics Tools*, **2**(1):21–28, 1997. 1, 2

8. Samet, H. The Design and Analysis of Spatial Data Structures. *Addison-Wesley, Reading, Massachusetts*, 1989. 2