Ibero-American Symposium on Computer Graphics - SIACG (2006)
P. Brunet, N. Correia, and G. Baranoski (Editors)

GPU-based evaluation of boolean operations on triangulated solids

C.J. Ogayar[1] F.R. Feito[1] R.J. Segura[1] and M.L. Rivero[2]

[1]Departamento de Informática, E.P.S. de Jaén. Universidad de Jaén. {cogayar,ffeito,rsegura}@ujaen.es
[2]Departamento de Informática, E.U.P. de Linares. Universidad de Jaén. mlina@ujaen.es

Abstract

This paper presents a robust method for evaluating boolean operations on triangle meshes. It is based on a fast and reliable point-in-solid algorithm that works with B-Rep representations. With this method, several boolean operations can be performed with almost the same processing cost as a single boolean operation. Moreover, the presented approach can take advantage of the modern programable graphics hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I.3.5 [Computer Graphics]: Boundary representations I.3.5 [Computer Graphics]: Geometric algorithms, languages, and systems

1. Introduction

Triangle meshes have multiple applications. They are used in CAD, virtual reality and videogames for representing polyhedral solids. This representation scheme is a standard in several areas, due to its simplicity. A triangle mesh can represent almost any object with a given level of detail. Moreover, it can be directly processed by the graphics hardware.

This paper presents an algorithm for evaluating boolean operations on triangle meshes. The method presented here is based on a classical approach for B-Reps [MT83, PK89]. The boolean evaluation method performs several optimizations in order to simplify the process. It is based on a triangle classification approach which is reduced to a point classification problem. The point classification algorithm used takes advantage of current programable graphics hardware (GPU).

2. Background

In order to perform the boolean evaluation of two solids A and B, three separate set, called subdivision regions, must be calculated: $A-B$, $B-A$, $A\bigcap B$. This operation is indispensable for evaluating boolean operations on B-Rep solids [Sha01]. When the subdivision is performed, a resulting region is selected depending on the desired boolean operation. For example, to evaluate a boolean difference only the $A-B$ region is selected and processed. There are several techniques for performing the subdivision of a triangle mesh. They are based on using section lines [MT83] or triangle trimming [PK89]. The method presented here uses the second approach, that is, it divides all triangles from one solid that intersect the other solid.

The rapid evolution of graphics hardware in the recent years is allowing a great increment in the performance of graphics applications. First, the GPU can be used in a wide variety of computer graphics problems [BFHSFHH*04, Lef04]. These problems can be resolved with high performance solutions based on features implemented in the hardware. Second, the programmable units of the GPU allow us to accelerate several operations which are applied to graphical models. Current programable GPUs can be used to solve generic problems. When using the GPU, the algorithms can be parallelized in order to improve their performance. Furthermore, the GPU can take charge of some portions of a program, saving CPU processing time for other purposes. It is important to note that graphics processors are optimized to work with data related to classic rendering purposes. Therefore the data used in a general purpose algorithm must be converted in order to be correctly processed by the GPU.

The method presented here uses a GPU-based point-in-solid test algorithm in order to accelerate the most time-consuming process of the method. This point-in-solid test algorithm is suitable for B-Rep representations based on planar faces [FT97, SFRTO*05]. This solution is robust and does not have degenerated cases, which are present in other algorithms such as crossing-count [Oro98, OSF05]. The boolean evaluation method presented here classifies the triangles of each solid using several optimizations. First, it only performs the classification of a selected triangle set. Second, it reduces the triangle classification to a point classification problem, which is obviously much easier. Finally, the point classification is carried out using a GPU based implementation [OJSF06]. This last step is the most time-demanding process, so a GPU based implementation allows us to achieve a better global performance.

3. Fundamentals

Basically, the algorithm has three main steps. The first step consists of intersecting the two solids involved in the boolean operation. The second step performs the classification of the triangle set resulting from the previous step. The final step selects a triangle set and builds a new solid.

First, the solids are transformed into the euclidean space and expressed using the same reference coordinate system. A decomposition of each solid is performed in order to obtain a resulting triangle set that meets several conditions: each triangle of a solid must not intersect with other triangle, and it must be completely inside, completely outside or completely on the boundary of the other solid. To summarize the whole process in a sentence, each triangle from one object is intersected with all triangles of the other solid, and the appropriate tessellation is performed when there is an intersection.

When the triangulation of each solid is adapted to the intersection regions with the other solid, the classification of each triangle is performed. Every triangle from solid A is logically on the boundary of A, and the same applies for B. To test whether a triangle from A is inside, outside or on the boundary of B, a point-in-solid test is carried out using the triangle barycenter. This simplification is based on the fact that every triangle is entirely contained or not in the solid to be tested, due to the triangulation performed in the first step. The method used for the point-in-solid test is presented in [SFRTO*05], and is implemented using the GPU (see Appendix). The result of this test determines the classification of the triangle with respect to the other solid, that is, inside, outside or on the boundary.

When all triangles have been classified, a subset that meets several conditions can be selected for each boolean operation (see section 6). Figure 1 shows a diagram of the entire procedure. The operations involved in the presented method are robust and reasonably simple. Moreover, the second step, which is the most time consuming process, can be implemented on the GPU.

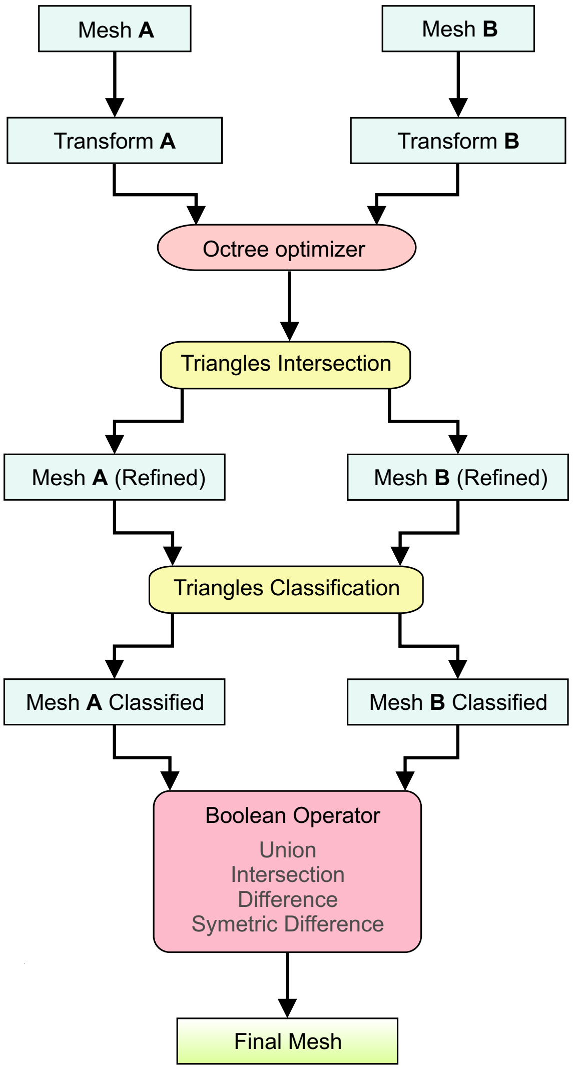

Figure 1: *The boolean evaluation process.*

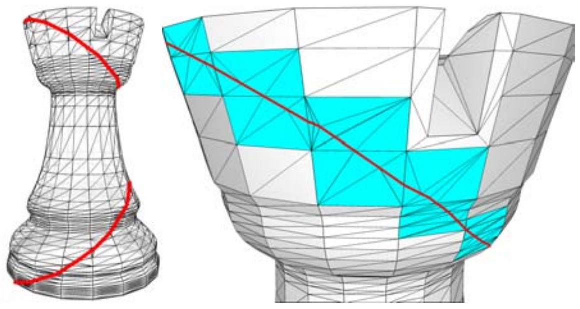

Figure 2: *A tessellation produced for adapting the rook triangulation for a classification with a sphere.*

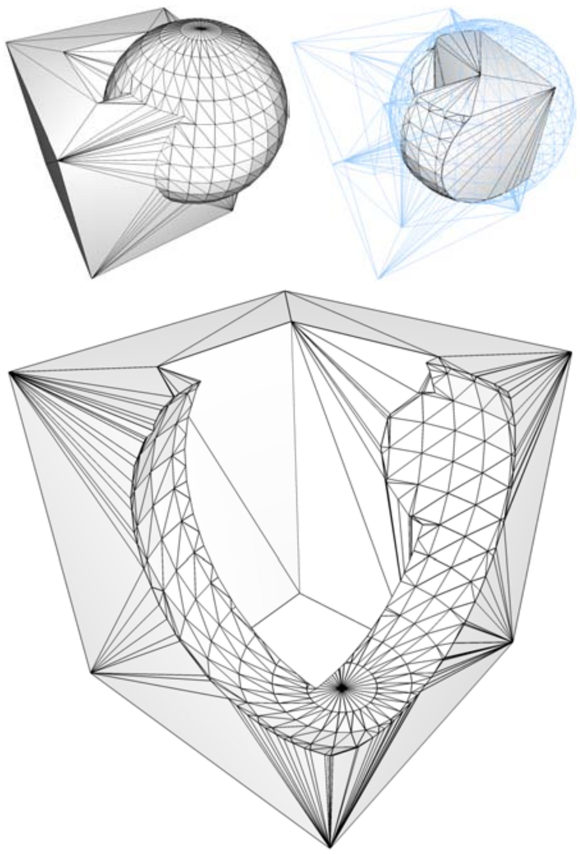

Figure 3: *Mesh refinement for performing a boolean difference on two solids.*

4. Mesh refinement

The mesh refinement process is the first step of the presented evaluation of a boolean operation. The goal is to obtain a resulting triangle set that meets the following conditions. Each resulting triangle of the decomposition of a solid must not intersect other triangle from the other solid. Besides, each resulting triangle must be completely inside, completely outside or completely on the boundary of the other solid. This resulting triangle set allows us to classify every triangle from each solid with respect to the other solid. These classifications can be performed using a single point from each triangle, more concretely, the barycenter is used for each triangle inclusion test. As the triangle inclusion test is reduced to a point-in-solid test, the process is more efficient than other boolean evaluation methods [RF04], which often rely on mesh connectivity information and edge subdivisions.

During the mesh refinement each triangle from one mesh is intersected with every triangle from the other mesh. With this process two new meshes are obtained. The triangulation of these meshes is adapted to the intersection zone between the two original solids. Figure 3 shows a sample. Note that with the new triangulations, the classification of each triangle can be reduced to a classification of a point included in that triangle. Unfortunately, this process is too complex to be implemented on the actual GPU, so it is implemented on the CPU instead.

With each triangle-triangle intersection new vertices are introduced into the B-Rep structure. This applies for both meshes during the intersection process. Those new vertices logically belong to the frontiers of both meshes, and they are the base for the new triangulations. Each triangle affected by an intersection must be decomposed into new triangles (and vertices). Figure 2 shows how the triangles are divided into smaller faces when they are over the intersection region.

The bottleneck of this stage is the triangle-triangle intersection algorithm, because it must be performed for every combination of two triangles from both meshes. To solve this a spatial optimization method must be used. In our implementation, an octree has been used. This octree stores in each leaf node the identifiers of the triangles of each mesh that intersect the volume represented by that node. Note that the octree is the same for both meshes, so it is adjusted to cover all triangles involved in the boolean evaluation. When a triangle-triangle test is to be performed, the nodes of the octree where the triangle to be intersected is included have a list of triangles from the other mesh that are also included in those nodes. With this optimization the triangles intersection step cost is dramatically reduced.

The mesh refinement increases the number of vertices and triangles. The intersection and tessellation of a triangle with a solid can be carried out in two forms. The first consist of calculating the intersections between the triangle and each face of the solid. With each intersection, new vertices are created, and the triangle must be tesselated correctly. This process is repeated for every face of the solid taking into account the new triangles created in each step. With this approach several intermediate tessellations are obtained before reaching the final result [RF04]. The second method consists of calculating the intersection of the triangle with all the faces from the solid with a single step. This is the approach implemented in this work. All the vertices created with the intersections are included in the original triangle to be tested, therefore a Delaunay [Lis94] triangulation can be applied in order to obtain the tessellation. The triangles from this tessellation replace the original triangle in the resulting mesh.

Attention must be paid to intersections between coplanar triangles because they can lead to special cases. In this work Möller's algorithm [Mol97] has been used, which efficiently resolves possible issues. Nevertheless it needs the Delaunay algorithm in order to complete the triangulation. Other methods such as [RF04] can complete this process with a single step, but with the drawback of having some special cases.

5. Triangle classification

The triangle classification is the second step of the boolean evaluation process. There are two triangulated meshes as a result of the previous step. These meshes are conveniently tesselated so that every triangle from both meshes is totally inside, outside or on the boundary of the other mesh. This allows us to determine the classification of each triangle using only a single point located on the triangle. This simplification makes the algorithm extremely simple to implement and gives an excellent performance.

The problem of triangle classification is reduced to a point-in-solid test. To test whether a triangle from A is inside, outside or on the boundary of B, a point-in-solid test is carried out using the triangle barycenter. For this purpose we have used a GPU based point-in-solid implementation [OJSF06]. This algorithm is very efficient and does not require precalculated data. Other choices for this purpose are BSP based point-in-solid test or a method based on the Jordan Curve Theorem [OSF05]. In any case, our GPU implementation is robust and very efficient. In the appendix a summary of our point-in-solid method is presented.

When the entire triangle set from each mesh is classified with respect to the other mesh (with *in*, *out* or *on* state), new solids can be obtained as a result of a boolean operation. To build a new mesh using a boolean operation, a new set of triangles must be selected from the two previously classified triangle sets. The new set of triangles must verify several conditions which depend on the boolean operation used (see section 6). This classification method allows us to calculate both regularized and not regularized boolean operations [RR99].

5.1. Optimized classification

When the intersection of two solids is calculated (the first step), there are triangles of a solid that are not divided because they are totally inside or outside the other solid from the beginning. Those triangles are located together in groups, which represent a part of the surface of the solid. These zones of the surface are normally surrounded by the triangles involved in the tessellation, which are located over the intersection line between the two solids. If the connectivity information between triangles is calculated, we can build groups of triangles that are separated by the triangles that form the tessellation zone. These groups are totally inside or outside the other solid, and they can be classified using a simple point-in-solid test.

Figure 4 shows the tessellations carried out for two intersecting solids. The figure on the right shows how each group is painted with a different color, representing a part of the solid that can be classified as a single point. With this optimization, the number of points to be classified greatly decreases. As the point-in-solid test is the most time demanding task of the boolean evaluation solution, this optimization is indispensable in order to achieve a good performance.

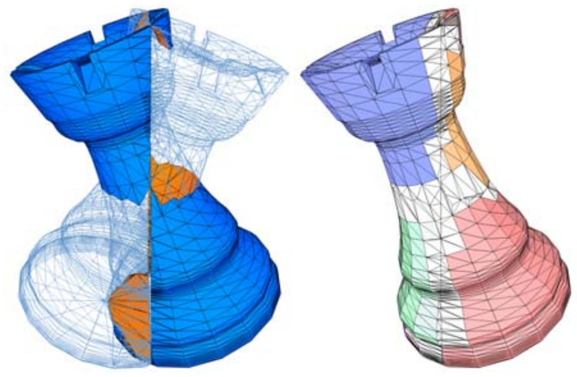
Figure 4: *A rook A - rook B boolean operation (left). Groups of triangles obtained with the tessellation of a solid (right). Each group is presented with a different color. The triangles that belong to the intersection zone do not have any color because they cannot be classified as a group.*

6. Boolean evaluation

This is the final step of the boolean evaluation solution. At this point the solids involved in the boolean operation are triangulated so that every triangle from each solid is completely outside, completely inside or on the boundary of the other solid. The triangles from each solid are also classified with respect to the other solid. This last step consists of selecting a triangle set that meets several conditions (see bellow) and building a new solid with that triangle set.

Let T_A be the set of triangles from the tessellation of the mesh A, and T_B the set of triangles from the tessellation of the mesh B. Let T^{-n} be a set of triangles with their normals inverted and *n(t)* the normal vector of the triangle *t*. Each boolean operation is defined with several conditions that the triangles must verify:

{A⋂B}: $\{T_A inB\} \bigcup \{T_A onB\} \bigcup \{T_B inA\}$. Intersection.

{A⋂*B}: $\{T_A inB\} \bigcup \{T_B inA\} \bigcup \{T_A onB / n(t_i) = n(t_j); t_i \in T_A, t_j \in T_B\}$. Regularized intersection.

{A⋃B}: $\{T_A outB\} \bigcup \{T_B outA\}$. Union.

{A⋃*B}: $\{T_A outB\} \bigcup \{T_B outA\}$. Regularized union. It is the same as {A⋃B}.

{A−B}: $\{T_A outB\} \bigcup \{T_B^{-1} inA\}$. Difference.

{A−*B}: $\{T_A outB\} \bigcup \{T_B^{-1} inA\} \bigcup \{T_A onB / n(t_i) \neq n(t_j); t_i \in T_A, t_j \in T_B\}$. Regularized difference.

{A⊖B}: $\{T_A outB\} \bigcup \{(T_A inB)^{-1}\} \bigcup \{T_B outA\} \bigcup \{(T_B inA)^{-1}\}$. Symmetric difference. It is the same as (A−B) ⋃ (B−A).

{A⊖*B}: $\{T_A outB\} \bigcup \{(T_A inB)^{-1}\} \bigcup \{T_B outA\} \bigcup \{(T_B inA)^{-1}\}$. Regularized symmetric difference. It is the same as {A⊖B} and also (A−B) ⋃ (B−A).

In order to perform a boolean operation, a triangle set that meets the corresponding conditions must be selected. Regularized operations are preferred because they produce correct solids. For example, to evaluate a regularized boolean difference, the following triangle set is selected: the triangles from A that are outside B, the triangles from B that are inside A (reversing their normals), and the triangles from A that are on the boundary of B and have a different normal vector. With this triangle set a B-Rep that represent the new solid is built.

The final result will be obtained by copying the triangles that meet the conditions of the boolean operation into a new mesh. Note that all boolean operations are performed with the same refined meshes obtained after the tessellations of the original solids. That is, all boolean operations between two solids share the results from the two first steps of the algorithm, which includes the mesh refinement and the triangle classification steps. The only difference is the third operation, that is, the final triangle set selection. For this reason, all possible boolean evaluations can be obtained with almost the same processing cost as a single one, because the last step of the algorithm has a very low processing cost (only a selection and a triangle set copy operation). This can be useful in some situations, such as an optimized CSG evaluation.

7. Implementation and results

The presented method can be implemented in several forms. As the whole process is divided into three steps, each one can be implemented using several approaches. For example, the mesh refinements can be built using other triangulation algorithm than the method presented here, and the point-in-solid algorithm can be GPU-based or not. Nevertheless, we present a robust and reasonably efficient implementation, which intensively uses the GPU, in addition of some optimizations.

The first step of the boolean evaluation method consists of performing an intersection between the two objects, followed by a triangulation of the result. The bottleneck of this stage is the triangle-triangle intersection algorithm, because it must be performed for every combination of two triangles from both meshes. This leads to a $O(n^2)$ algorithm which is very inefficient. To solve this, a single octree classifier is used for the transformed meshes. The octree stores in each leaf node the identifiers of the triangles of each solid that intersect the volume represented by that node. With this optimization the triangle-triangle intersection step cost is greatly reduced. With an octree depth of 7 or 8 the performance is dramatically improved, having a reasonably low structure calculation time. This is the most important optimization of the first step of the boolean evaluation method.

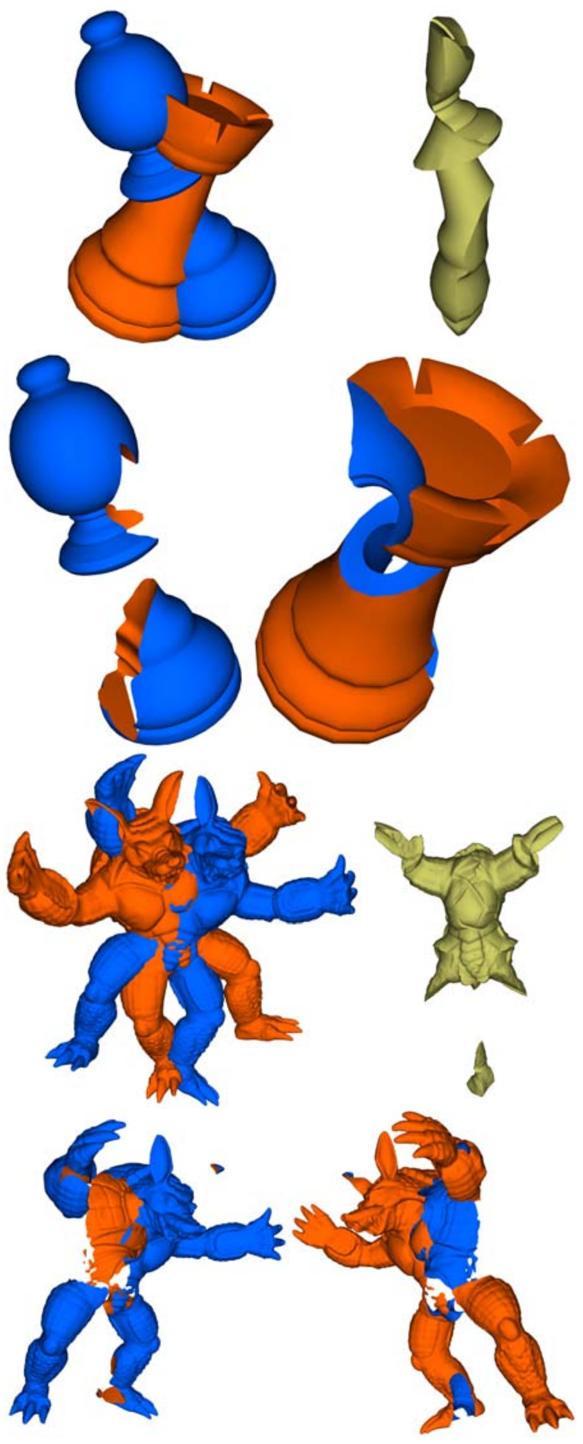
Figure 5: *Some boolean operations. The first four figures show (front left to right, top to bottom) Bishop⋃Rook, Bishop⋂Rook, Bishop−Rook and Rook−Bishop respectively. The last four figures show (front left to right, top to bottom) Armadillo A⋃Armadillo B, Armadillo A⋂ArmadilloB, Armadillo A−Armadillo B and Armadillo B−Armadillo A respectively.*

	Triangles	3D Studio Max 8	New (CPU)	New (GPU)
Rook	2256	0,5	0,8666	0,2203
Bishop	5814	2,2	3,6656	0,7708
Knight	15350	15,0	9,5663	1,7180
Golf Ball	46205	86,0	63,6881	6,7503
Armadillo	150000	1125,0	694,8313	102,5001

Table 1: *Times in seconds for several intersections between two instances of the same solid. The other boolean operations present very similar times, so they are not included. The translations and rotations used to setup the solids are the same for all operations. The times for 3DStudio Max 8 have an error* $|e| < 0.25$ *seconds.*

The GPU based point-in-solid test described in the Appendix is used for classifying the triangle set of each refined mesh. As our algorithm reduces the triangle classification to point classification, the barycenter of each triangle is used to classify it. Floating point precision is very important to keep the robustness of the method. This is because nearly zero point-in-solid tests can determine that a point and the associated triangle are exactly on the surface. If the precision for the epsilon used in the operations is not enough, problems arise when intersecting coplanar faces from both solids.

The last step of the algorithm produces a new solid as the result of the boolean operation. This new solid is represented by a mesh that is built using a select triangle set that meets the boolean operation conditions (see section 6). When using regularized operations, the resulting solid is represented by a valid B-Rep.

The data structure used to represent meshes is very simple. The vertices are stored as an array of vectors. The triangle list is an array of integer-based indices which reference the associated vertices of each triangle. During the mesh refinement new vertices and triangles are inserted into the structure. There can be duplicated vertices which must be removed in order to ensure that the resulting mesh will be a valid representation of a solid. When removing duplicated vertices its linked triangles must be adjusted to reference the correct vertices. If several boolean operations are planed to be carried out, the duplicated vertices can be removed only after the final operation in order to improve the performance.

To test the method several meshes have been used. The meshes have different polygon quantities and different topological properties. The tests have been carried out with an Intel Pentium 4 3.4GHz with 1Gb of memory using SSE2. The GPU used is a NVidia GeForce 7800GTX 256Mb with PCI-Express x16. The implementation of the algorithm has been completely written using C++ and NVidia CG [KF05, FK03].

Table 1 shows the performance times of the tests. As expected, the algorithm depends on the number of polygons. The performance of the GPU-based implementation of the algorithm is better than the CPU-based implementation. The CPU version is better for low polygon meshes, because this version does not need the initialization of any structures and buffers. However, the negative influence of the factors that affect the GPU version is reduced when the number of polygons increases. The performance of the GPU is higher than the CPU because of its specialized parallelism. Table 1 shows how the higher the count of polygons the greater the advantage of using the graphics hardware.

Figure 5 shows some examples of boolean operations between triangles meshes using our implementation. Table 1 shows performance tests for some boolean operations. A comparison with the standard boolean operator of 3DStudio Max 8 is also presented. Unfortunately, we do not have detailed information of how 3DSMax performs the boolean evaluations, therefore the comparison is presented for illustrative purposes. Note that 3DSMax presents a performance issue when processing very high detailed meshes. The proposed algorithm times include the entire process as described in previous sections, including the octree creation, the mesh refinement, the triangle classifications (which involves the GPU based point-in-solid tests in the GPU based version), the boolean evaluation and the final mesh creation.

8. Conclusions

We have presented an efficient and robust method to evaluate boolean operations on triangle meshes. The method can be implemented in several ways. Nevertheless, the solution presented here uses the modern programable GPU to enhance the performance. The data structure needed to represent the objects is the simplest form of B-Rep, therefore almost any 3D file format can be used without the need of performing complex data conversions.

9. Acknowledgements

We wish to thank the reviewers for their helpful comments. This work has been partially funded by the Ministry of Science and Technology of Spain and the European Union by means of the ERDF funds, under the research project TIN2004-06326-C03-03. Some meshes used in this work have been downloaded from http://graphics.stanford.edu.

References

[BFHSFHH*04] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., HANRAHAN, P. Brook for GPUs: Stream Computing on Graphics Hardware. *SIGGRAPH 2004 Paper Talk*, August 12, 2004.

[FT97] FEITO, F., TORRES, J.C. Inclusion test in general polyhedra. *Computer & Graphics*, 21, pp. 23-30, 1997.

[FK03] FERNANDO, R., KILGARD, M. J. The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. *NVidia, Addison-Wesley*, 2003.

[KF05] KILGARIFF, E., FERNANDO, R. The Geforce 6 Series GPU Architecture, in GPU Gems 2. *Nvidia, Addison Wesley*, 2005.

[Lef04] LEFOHN, A. GPU Data Formatting and Addressing. *GPGPU, SIGGRAPH 2004*, 2004.

[Lis94] LISCHINSKI, D. Incremental Delaunay triangulation. *in Graphics Gems IV*, pp 47-59. Academic Press, 1994.

[MT83] MÄNTYLÄ, M., TAMMINE, M. Localized set operations for solid modelling. *Computer & Graphics*, 17 (3), 1983.

[Mol97] MÖLLER, T. A Fast Triangle - Triangle Intersection Test. *Journal of Graphics Tools*, vol. 2, pp 25-30. A.K. Peters, 1997.

[ME93] MORELAND, K., ANGEL, E. The FFT on a GPU. *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*, pp. 112-120, 1993.

[OSF05] OGAYAR, C. J., SEGURA, R.J., FEITO, F. R. Point in Solid Strategies. *Computers & Graphics*, 29, No. 4, 2005.

[OJSF05] OGAYAR, C. J., JIMÉNEZ, J.J., SEGURA, R.J., FEITO, F. R. Inclusión de Puntos en Sólidos mediante el uso de Hardware Gráfico Programable. *Congreso Español de Informática Gráfica, CEIG 2005.*

[OJSF06] OGAYAR, C. J., JIMÉNEZ, J.J., SEGURA, R.J., FEITO, F. R. Point in Solid Test on the GPU. *Submited to Computer Graphics Forum*, 2006.

[Oro98] O'ROURKE, J. Computational Geometry in C (second edition). *Cambridge University Press*, 1998.

[PK89] PILZ, M., KAMEL, H.A. Creation and boundary evaluation of CSG models. *Engineer Computer*, 5, pp. 105-118, 1989.

[PBMH*02] PURCELL, T. J., BUCK, I., MARK, W. R., HANRAHAN, P. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3), pp. 703-712, 2002.

[PDCJH*03] PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., HANRAHAN, P. Photon Mapping on Programable Graphics Hardware. *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*, pp. 41-50, 2003.

[RF04] RIVERO, M., FEITO, F.R. Refinamiento de mallas triangulares. Aplicación para el cálculo de operaciones booleanas en 3D. *CEIG 2004*, Sevilla, 2004.

[RR99] ROSSIGNAC, J.R., REQUICHA, A.R. Solid Modeling. *In J. Webster, editor, Encyclopedia of Electrical and Electronics Engineering*, Webster, John Wiley & Sons, 1999.

[SFRTO*05] SEGURA, R., FEITO, F.R., RUIZ DE MIRAS, J., TORRES, J.C., OGAYAR, C. J. An Efficient Point Classification Algorithm for Triangle Meshes. *Journal of Graphics Tools*, 10(3), pp. 27-35, 2005.

[Sha01] SHAPIRO, V. Solid Modeling *in Handbook of Computer Aided Geometric Design, G. Farin, J. Hoschek, M.S. Kim*, Elsevier Science, 2001.

Appendix: **Point-in-solid test**

Feito [FT97] proposes a point-in-solid algorithm for polyhedral representations based on the study of signs. The performance of this method is similar to the performance of the crossing-count approach. However, Feito's point-in-solid algorithm does not have degenerated cases. Segura [SFRTO*05] extends the algorithm with optimizations for triangle meshes using minimal topological information. Ogayar [OJSF05, OJSF06] adapt the algorithm to be implemented on GPU. In this section we present a summary of this graphics hardware based method. The boolean evaluation method presented in this paper uses this GPU based point-in-solid implementation in order to improve the global performance.

The implementation is carried out using OpenGL and NVidia CG [KF05, FK03]. This allows rapid development and the possibility of using the advanced features of current GPUs [ME93, PBMH*02, PDCJH*03]. In short, point-in-solid algorithm involves the following steps [FT97, OSF05, OJSF05, OJSF06]:

1. A simplicial covering of the solid is built [FT97, OSF05]. This structure is composed of tetrahedra. For each triangle of the solid a tetrahedron is built using the triangle vertices and an additional point, which is the same for all tetrahedra. A good choice for this point is (0,0,0).
2. The inclusion of the point in each tetrahedron is calculated. Note that tetrahedra can overlap. The result for the point in tetrahedron test will be -1 if the point is inside or +1 if the point is outside the tetrahedron.
3. The inclusion value for all tetrahedra is accumulated. If the sum is ≥ 1 the point is inside the solid. Otherwise, the point is outside.

As stated previously, if the point is on an edge that connects the reference point with a vertex from the solid, two values must be associated with that edge. This is a special case. Also, if the point is on a face shared between two tetrahedra the inclusion state is divided by two.

In order to achieve the GPU-based implementation, the algorithm is divided into two steps. This is because there are serious limitations in current graphics hardware, that is, the graphics pipeline is too inflexible. Note that in order to render a graphic primitive using a programmable GPU, one vertex shader and one pixel shader are used in that order. In the first step a vertex shader is used to compute the inclusion of the point in each tetrahedron of the solid covering. The pixel shader performs a simple data conversion. The result is stored in a buffer, typically the framebuffer. In the second step, another pixel shader computes the final sum of the values calculated in the previous step. At this point, no vertex shader is used. In the following section, the details of each step are explained.

First Step

The first step of the algorithm uses a vertex shader to perform the point in tetrahedron test for all tetrahedra of the simplicial covering of the solid. The pixel shader writes in the framebuffer what it receives from the vertex shader. In a standard rendering process the vertex shader transforms vertices and the pixel shader writes data in the framebuffer using interpolated values that depend on the current graphic primitive (such as triangles). With this inclusion algorithm, the purpose is to execute the vertex shader once per tetrahedron and to write the result onto the framebuffer. This is achieved using the following operations:

1. The data of tetrahedra associated with the triangle mesh are calculated and converted to an adequate format for the GPU. An OpenGL display list has been used to codify tetrahedra as vertices with attributes, so that one vertex and its attributes in the graphics pipeline will actually contain the entire data for one tetrahedron. This data is sent to the graphics hardware. For each of these OpenGL vertices a vertex shader is executed in order to calculate the point-in-tetrahedron algorithm. As the vertex actually contains the data of a tetrahedron, each vertex shader execution resolves a point-in-tetrahedron test. Algorithm from 6 shows appin structure which defines semantics for the attributes of a vertex, that is, the purpose of each vertex attribute in the vertex shader. Therefore three tetrahedron points can be codified using the normal, the color and the texture coordinates of the vertex. The fourth point of all tetrahedra is fixed as the origin of coordinates. The real position of the vertex is used to place it in the viewport according to its vertex index. The viewport is therefore used as if it were a standard 2D array. Finally, the z coordinate of the transformed vertex is used to store the sign of the corresponding tetrahedron.
2. A 2D array of point primitives is rasterized in the viewport with Standard OpenGL commands activating the point-in-tetrahedron vertex shader. A point primitive will generate only one pixel. Consequently a ratio of one-to-one is achieved between elements in the vertex shader and elements in the pixel shader. In this way, the resulting pixels in the framebuffer will be aligned with an n*n matrix, n power of 2. The reason for this is explained is the second step of the algorithm. The inclusion state for a tetrahedron in [-1,1] range is stored in the framebuffer. Note that the framebuffer is configured in a floating point format, so the stored values are not clamped to [0,1].

To sumarize this entire process as a whole: one tetrahedron is codified as a vertex with attributes; then it is introduced into the graphics pipeline and processed with the vertex and pixel shaders. Finally, the result is stored in the framebuffer. A one-to-one correspondence between the tetrahedra and resulting pixels is maintained. Due to the fact that a floating point framebuffer can contain up to 128 bits per texel, four 32 bits values can be used in each shader. This allows us to perform the processing for four points in a single shader execution, and takes advantage of the streaming nature of the GPU.

At the end of the first step of the algorithm the framebuffer contains the inclusion state for all tetrahedra. The second step of the algorithm performs the final sum of the data. This process can be performed in two ways: copying the framebuffer content to main memory in order to handle the special cases in the CPU, or processing the data in the GPU using a pixel shader.

Special cases are handled in the GPU as follows: The point-in-tetrahedron state of a point is 0 (inside) or 1 (outside). For the rest of the cases special codes must be used. However, this codes cannot be used for performing the final sum at the end of the algorithm. Therefore, this step must be performed in the CPU. The problem is how to detect the special case at the GPU level. In the first step which involves the vertex shader that performs the point-in-tetrahedron test, this vertex shader must calculate the final position of the OpenGL point in the framebuffer. The solution to the detection of special cases is to translate the resulting position of the vertex to an unused location of the frame buffer, for example, the upper right corner. This position does not correspond to any tetrahedron, it is only used as a flag. This step is only performed when a special case is detected at the point-in-tetrahedron test. When the flag is activated a special case has occurred. The flag is checked by performing a 1x1 pixel transfer from the framebuffer to CPU memory. This checking is performed only once and after all tetrahedra have been processed. When a special case is detected, the first step of the algorithm must be repeated deactivating the special cases checking system to avoid the translation of some vertices to the flag position. The second step of the algorithm must be completed using the CPU. The frequency of special cases is very low. Therefore the average performance is not severely affected.

```
#define TETRA_OUTSIDE 0
#define TETRA_INSIDE 1
#define TETRA_FACE 0.1
#define TETRA_EDGE1 0.2
#define TETRA_EDGE2 0.3
#define TETRA_EDGE3 0.4

struct appin {
  float3 vertexData : POSITION;
  float3 tetraVertex1 : NORMAL;
  float3 tetraVertex2 : COLOR0;
  float3 tetraVertex3 : TEXCOORD0;
};

struct vout {
  float4 position : POSITION;
  float4 color : COLOR0;
  float4 signs : COLOR1;
};

vout tetraVertexProgram4 (
  appin IN,
  uniform float3 testPoint,
  uniform float3 testPoint2,
  uniform float3 testPoint3,
  uniform float3 testPoint4,
  uniform float3 v0,
  uniform float4x4 ModelViewProjectionMatrix )
{
  vout OUT;
  OUT.signs = IN.vertexData.z;
  OUT.color = float4 (
    tetraPoint ( IN.tetraVertex3, IN.tetraVertex2,
      IN.tetraVertex1, testPoint ),
    tetraPoint ( IN.tetraVertex3, IN.tetraVertex2,
      IN.tetraVertex1, testPoint2 ),
    tetraPoint ( IN.tetraVertex3, IN.tetraVertex2,
      IN.tetraVertex1, testPoint3 ),
    tetraPoint ( IN.tetraVertex3, IN.tetraVertex2,
      IN.tetraVertex1, testPoint4 )
    );
  OUT.position = mul ( ModelViewProjectionMatrix,
    float4 ( IN.vertexData.x, IN.vertexData.y, 0.0, 1.0 )
    );
  return OUT;
}
```

Figure 6: *CG Vertex shader for calculating the inclusion state of 4 points in a tetrahedron. The tetrahedron data is specified as a vertex with attributes.*

```
float4 accum ( float2 texcoord : TEX0,
  uniform samplerRECT img : texunit1 ) : COLOR
{
  float4 a, b, c, d;
  a = f4texRECT ( img, texcoord );
  b = f4texRECT ( img, texcoord + float2(0,1) );
  c = f4texRECT ( img, texcoord + float2(1,0) );
  d = f4texRECT ( img, texcoord + float2(1,1) );
  return a+b+c+d;
}
```

Figure 7: *The* accum *CG pixel shader calculates the addition of the values of the framebuffer in the second step.*

Second Step

In the second step no vertex shader is used, only a pixel shader. This step consists of a sum of the floating point values stored in the framebuffer, which is considered a typical 2D array. To achieve this, several operations must be performed:

1. The result of the first step of the algorithm must be available as a texture. This can be carried out with a render-to-texture capability of the GPU in the first step, that is, the framebuffer will actually be a texture. If this feature is not available in a given graphics hardware a copy operation between the framebuffer and a separated texture is adequate although slower. The texture contains the inclusion states for every tetrahedron, and is available to be used in the pixel shader.
2. A rendering process is started. The vertex shader is deactivated and texturing is enabled.
3. A quad primitive is drawn. The size of the quad is n/2 x n/2 for a n x n texture. The texture is mapped so that it entirely covers the new quad. This is achieved by adjusting the coordinate textures of the quad. The source texture is a special texture which contains floating point values, so the texture coordinates are specified in pixels instead of standard [0,1] format. In this way the top right corner of a 512x512 texture is (511,511) instead of (1,1). The algorithm is easier to implement if the framebuffer has a size of nxn, and n a power of 2. This can usually mean a little wasted space for most cases, but the performance tends to be faster because the shader is simpler.
4. At the pixel level of the rendering pipeline, the pixel shader is executed for each pixel of the resulting image. With the drawing of the quad with the mapped texture, each pixel of the resulting buffer will be mapped to a 2x2 texture region in the pixel shader. This pixel shader will access its four corresponding texels from the texture and it will add them all. The result of the sum will be stored in the framebuffer. Algorithm from 7 shows this process.
5. This process is actually a buffer reduction using a sum operator. Each resulting pixel is in fact the sum of four

texels of the source texture. With this method, an iteration of the graphic pipeline performs a buffer reduction of n/2 x n/2. This process is repeated swapping buffers until the resulting buffer has a size of 1x1, which contains the final result of the sum.

For each iteration of this step the input buffer is reduced to an output buffer by a factor of 2. This factor can be increased by implementing loops inside the pixel shader. In this way each sum involves more than four values, so fewer executions of the pixel shader are needed. Moreover the sizes of the buffers must be adjusted properly. Nevertheless it is convenient to use all available graphics processor pipes to maximize the parallelism of the GPU, so for the most cases it is better to maintain the reduction factor at 2. The buffers used have a 128 bits floating format. This allows us to store up to four 32 bits floating point numbers per buffer position. Therefore four point-in-tetraheron results can be stored in each buffer element. The reduction of n^2 elements using 4x4 partial sums is performed in $O(log(n))$ steps.