ACSGM – An adaptive CSG meshing algorithm

W. Purgathofer R.F. Tobler T.M. Galla

Abstract

We present a new algorithm, called ACSGM (which is short for *Adaptive* CSG *Mesher*), that converts scenes in CSG representation into a boundary representation (BREP) based on flat triangles. The algorithm is based on the *marching cube* algorithm, but instead of working at a fixed resolution, the size of the cubes used in the meshing process is changed *adaptively*.

Quite different from the marching cube algorithm, which calculates the vertices of the triangles of the final mesh using *linear interpolation*, ACSGM uses *ray casting* for this computation. This approach not only produces *exact vertices* but provides some additional information (e.g. the normal vectors in these vertices) that can be used to generate a more accurate approximation of the CSG object by the final mesh.

Introduction

CSG is a widely used modeling paradigm, but some algorithms require a scene description based on a polygonal mesh (e.g. radiosity). Due to this fact a lot of meshing algorithms which convert a scene description given as CSG data structure into various kinds of meshes do exist.

Most of these converters will create a complete mesh for a model according to the estimated resolution requirements of the application software (e.g. the algorithms described by Boender [1]). If the application software changes its requirements a completely new mesh needs to be generated.

Since this results in a waste of computation power, because previously generated meshes are discarded, we decided to design a meshing algorithm that can *incrementally change the mesh resolution* according to the desired accuracy of the generated mesh. As an example, the application software can request a global or local limit on the maximum distance of the mesh to the surface of the original object. If the requirements change, the incremental algorithm will then change the previously generated mesh to meet the new criteria.

Marching cube for CSG

Applying the standard marching cube algorithm as proposed by Lorensen and Cline [4] to CSG objects has several major drawbacks:

- A CSG object is a discrete case of volume data representation, where the only possible values for the volume data are 0 (for a point outside the object) and 1 (for a point inside the object). Therefore it is not possible to create exact intersection points by interpolation of the volume data along an edge. The only possibility is to use the *midpoint of the edge* as intersection point, which is a rather rough approximation.
- The standard algorithm is not able to represent a CSG object (or a part of the object (i.e. any node in the CSG tree)) in cases, where the object (or the part of it) lies completely within one marching cube.
- The marching cube algorithm is only able to distinguish between the inside and the outside of the object, but it is not able to handle objects made up of different materials correctly (i.e. represent the materials boundaries in a correct manner).

One goal in the development of ACSGM was to overcome these drawbacks. The next section describes how this goal can be achieved.

Key ideas

This section deals with the key ideas of the enhancement of the standard marching cube algorithm.

Hierarchical structure

Since the standard marching cube algorithm only works at one level of detail it is not possible to create a finer grained mesh for parts of the object where an *approximation error* is very visible because these parts are close to the observer or very likely to occur (e.g. at places of high curvature of the object's surface). To overcome this drawback we use a *hierarchy of cubes with different sizes* instead of only one level of fixed size cubes to create meshes with different *levels of detail*. Each cube's edge has a length that is exactly $\frac{1}{2}$ of the length of a cube at the lower level of detail.

The criterion for the subdivision of one voxel will be explained in the next section.

The hierarchical structure used in ACSGM is based on an *octree*. Each voxel represents one marching cube and generates its own mesh. The sub-voxels of a given voxel contain a finer grained mesh.

This hierarchical approach has the following major advantages:

- The used resolution changes according to the curvature of the object. Thus *no computation power* is wasted.

- A *controlled refinement* of the mesh in particular places of the object is possible. Therefore the accuracy of parts of the generated mesh can be increased according to the needs of the specific application.

Adaptive marching cube

Due to the ispiration by the work of Campell and Fussell [2], ACSGM uses an *adaptive version* of the standard marching cube algorithm. Instead of using one cube of fixed size that "marches" over the whole object, we use the voxels of an octree to create the mesh. Each voxel is used to create a small part of the final mesh (its own *local mesh*).

This local mesh represents the part of the CSG object, that lies within the voxel.

After all local meshes have been created these meshes are put together to form a global mesh that is a BREP of the original CSG object. In order to facilitate the task of the combination of the local meshes, the level of detail between two neighboring cubes should not differ by more than one level.

If the mesh created by one voxel is not accurate enough (i.e. the mesh is only a poor approximation of the original object surface), then the voxel is divided into 8 sub-voxels, and each of the sub-voxels creates its own local mesh. The decision whether the mesh is accurate enough is primarily based on the following simple rules:

- If the material of the object changes more than once along any edge of the cube, or more than one intersection between one edge of the cube and the object takes place, the approximation that can be created by this cube is inaccurate in any case.
- If the distance between the object's surface and the mesh exceeds a specific maximum, a further subdivision has to take place.
- If the bounding box of any CSG node of object is completely within the cube, the mesh is inaccurate for sure, since this special node will not be represented by the final mesh.

Additionally some *application specific* classification functions for the accuracy of the resulting mesh can be introduced:

CSG **modeler:** For this type of application, a *viewpoint dependent* refinement of the mesh might be desirable. Therefore an additional criterium to refine the part of the mesh that are closer to the observer to a higher level of detail can be employed in the algorithm.

Radiosity: When generating meshes for radiosity, a good approach might be to base the refinement on the BF criterium introduced by Hanrahan, Salzman and Aupperle [3].

Surface flow: For applications that visualize surface flow, an appropriate criterium might be to refine parts that cause extreme turbulences of the flow (i.e. numerical instabilities in the solution).

Thus a *controlled refinement* of specific parts of the mesh is possible. Due to the flexibility of the algorithm, this refinement can be driven by the application's needs.

Ray casting for intersections

Instead of classifying each vertex of the marching cube, we *intersect each edge of the cube with the object* using ray tracing. The advantages of this approach are discussed in the next sections.

Exact intersections: Using the exact intersections provided by the ray caster enhances the quality of the final mesh compared to the standard marching cube algorithm, which uses the midpoints of the intersecting edges.

Surface correction: The standard marching cube performs very badly in situations where *polyhedral* CSG *objects* have to be meshed. In some of these situations it would even be possible to obtain a correct representation of the object using a *boundary representation based on flat polygons.* In order to provide a better result in these situations, we decided to incorporate a technique we call *surface correction* into our algorithm.

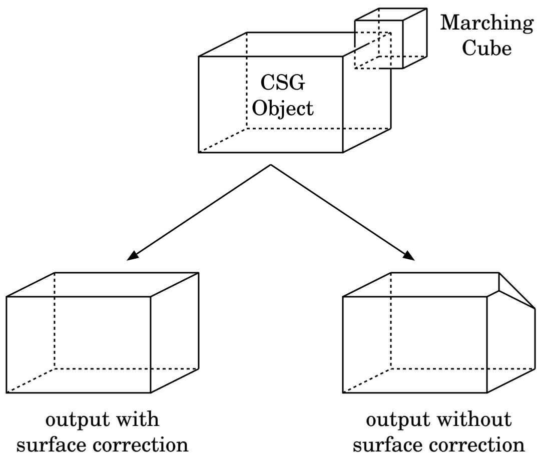


Figure 1

Figure 1 shows an example, where the standard marching cube algorithm would cut off the vertex of the cube, whereas ACSGM is able to reconstruct the correct surface of the object due to surface correction.

A detailed description how surface correction is incorperated into the algorithm will be given later on.

Multiple materials: Using ray casting for the computation of intersections has the enormous advantage that we are able to consider material boundaries. Thus we are not bound to a binary classification like the one made by the standard marching cube algorithm.

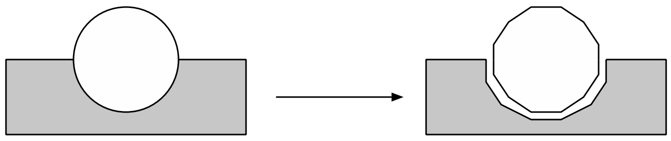


Figure 2

Using ray casting enables us to deal with objects made up of *multiple materials*, since the ray caster generates a *hit at every material boundary* (i.e. at every *transition between different materials*). Therefore, we are able to represent the material boundaries, that are present in the CSG object, in the final mesh. Figure 2 shows an example where a sphere has been united with a box. Since both CSG primitives are made up of different materials, the whole CSG object contains a material boundary. Since ACSGM uses ray casting for the computation of the intersection points, this material boundary will be present in the generated mesh (right part of figure 2).

Basic algorithm

Computation of intersection points

In order to obtain vertices for the triangle mesh, each edge of the current cube is intersected with the CSG object using ray casting. The ray caster produces a *hit* at every *material transition* along the edge of the cube. If more than one intersections occur along one edge of the cube, the *midpoint* of all these intersection points is taken as vertex at the current level of detail. Since this is only an approximation that discards features of the CSG object that are small compared to the size of the marching cube, a further refinement has to take place.

After the computation of the vertices some *material information* has to be assigned to them. To be able to perform a correct assignment, it has to be distiguished between *enter hits* (i.e. hits where a

CSG object is entered) and *leave hits* (i.e. hits where the edge leaves a CSG object). In the first case, the material of the CSG object, that is *entered* by the edge at the point of the hit, is taken as the material of the vertex. In the latter case, the material of the object that is *left* is chosen. In case of several intersection points along the edge, the material of the first and the last intersection point is assigned to the vertex that is obtained by computing the midpoint of all intersection points.

In the last step of this part of the algorithm a *value* is assigned to the vertex, depending on the number of intersections for this specific edge. If the number of intersections is odd, the value of the vertex is 1, if it is even, a value of 0 is assigned to the vertex.

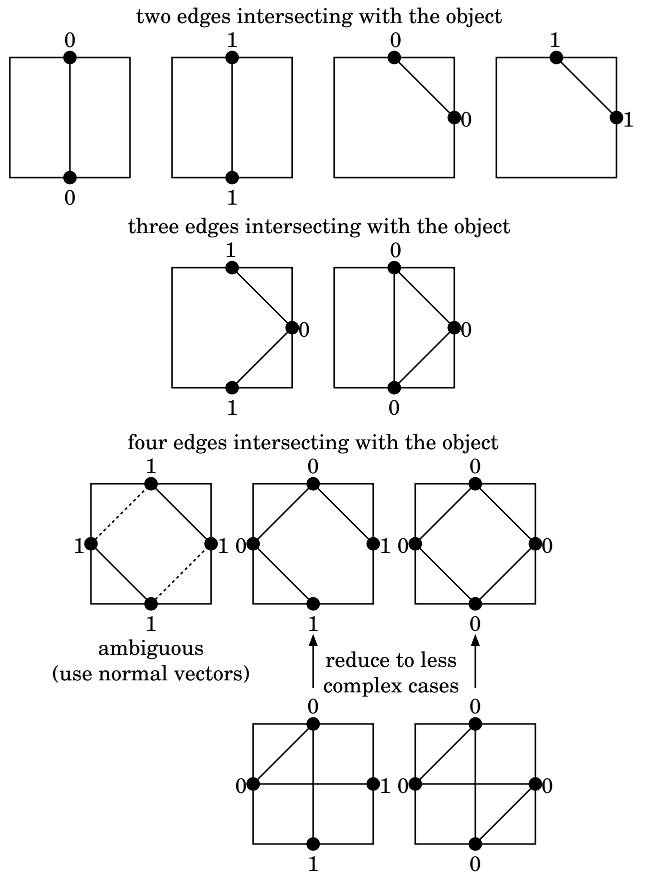
Connecting the intersection points

We consider the intersection points of every edge of a face (each face has a minimum of 0 and a maximum of 4 intersection points) and connect these points according to the classification shown in figure 3 depending on the value of each point.

From the five possible cases with 4 intersection points, two of them, which are rather complex (the connections of the lines would make up a bowtie polygon), can be reduce to simpler ones as indicated by two arrows in figure 3. The mistake that is definitively made by this simplification is either corrected by one of the following steps of the algorithm, which is called *surface correction* or by a further subdivision of the cube.

The classification shown in figure 3 is unambiguous except for one case, where intersections are encountered with all four edges, and the intersection point of each edge has the value 1. This ambiguity is solved by taking the normal vectors in the intersection points (a piece of information provided by the ray caster) into account.

When two points are connected to form one line, the material that is common to *both points* is chosen and assigned to both of the points. In the case that the two points have both materials in common, two lines will be created out of the single line, and each line will get one of the two materials. Since one intersection point can be endpoint of two lines, ACSGM creates two instances of the point in this case, where each of the two has its own material. Therefore a line that is created in this way has a defined single material.



two edges intersecting with the object
0
1
0
1
0
1
0
1
0
0
three edges intersecting with the object
1
0
0
0
1
0
four edges intersecting with the object
1
0
0
1
1
0
1
0
0
0
1
1
0
ambiguous
(use normal vectors)
reduce to less
complex cases
0
0
0
1
0
0
1
0

Figure 3

Afterwards these lines are split into two parts by an additional point that is calculated by *intersecting* the *tangential planes* in the endpoints of the line with the plane that is defined by the *face of the cube* that contains this line.

In cases, where the normal vectors in the endpoints of the line are linear dependent, no split point is generated. Figure 4 illustrates this fact.

The generation of this split point is the part of the *surface correction* incorporated in the presented algorithm.

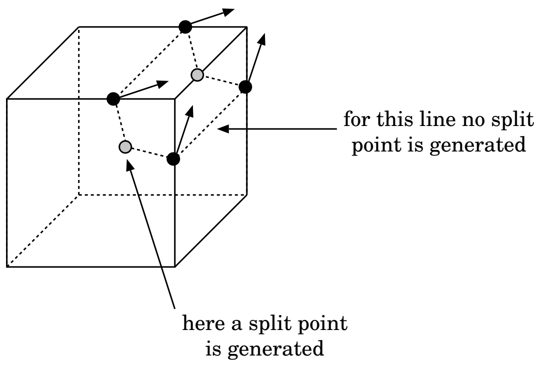


Figure 4

Building temporary polygons

In the next step the single lines are concatenated to form polygons. Since we want polygons with a well defined material, only lines consisting of the *same material* are connected.

Normally the concatenation of several lines to polygons is not easy to accomplish, since the case where two or more polygons share one edge and therefore have one line in common, has to be treated in a special way. Due to the previous step all *lines that are common to two polygons* have been *duplicated*, because the two endpoints have had *both materials in common.* This holds for polygons that consist of the same material as well, since in that case the vertices of the common line have both materials in common, too. The only difference in this case is, that actually all four materials are equal, but nevertheless two lines are created. Thus it is sufficient to connect lines with the same endpoints and the same material in those endpoints.

Splitting polygons into triangles

In general the vertices of the polygons created in the previous section *will not lie on one plane* and therefore it is required to split these polygon into several flat triangles, which is done by the computation of the so called *complex split point.*

The computation of the intersection point is based on the following equation:

$$N \cdot \vec{x} \;=\; \vec{c} \qquad (1)$$

where N is a $n \times 3$ matrix with the normal vectors in the different intersection points as rows. The vector $\vec{c}$ consists of the constants from the equations for the different tangential planes. Since this linear equation system is in the most cases *overdetermined* (there are only 3 unknowns and n linear equations), we use *singular value decomposition* to obtain the *least-squares solution* as described by Press, Flannery, Teukolsky and Vetterling [5].

In some special cases (i.e. when the tangential planes in the intersection points are parallel) the *normal vectors* in the intersection points are *linear dependent* and therefore the different equations are linear dependent as well. This may lead to an *underdetermined system* which of course has no unique solution. If this is the case, we have to distinguish between two different cases:

- If all linear equations are mutually linear dependent, the midpoint of all vertices is used to split the polygon into triangles.
- If there are exactly two linear independent equations, the midpoint of all *line split points* is taken as complex split point (figure 5).

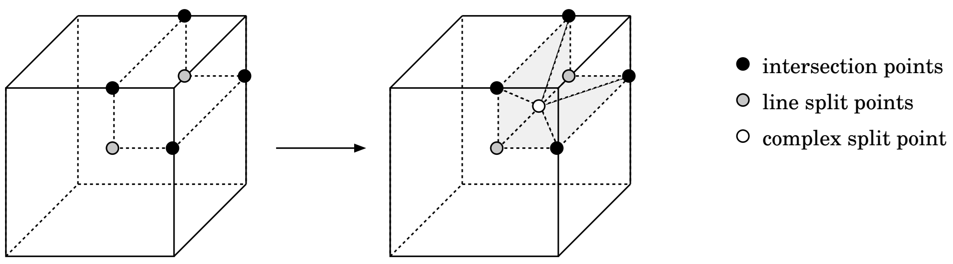


Figure 5

The choice of complex split point is the second part of the *surface correction.*

Combining the local meshes

In order to obtain a mesh that is free from T-vertices, we have to ensure, that the faces of neighboring cubes that touch each other, have exactly the same intersection points. Even the lines resulting from connecting these intersection points must be the same.

This requirement is obviously fulfilled for cubes of the same size, since both cubes intersect the *same edges* with the object resulting in the *same intersection points* with the *same classification*. Therefore neighboring cubes of equal size do not lead to any problem.

If the neighboring cubes are of different size, the situation is not that rosy, since in this case in general the intersection points will not be the same. Thus the larger cube *discards its own intersection points and adopts the intersection points* of all smaller cubes that share a part of the specific face with it. Thus it is guaranteed in a rather simple way that neighboring cubes always have exactly the same intersection points on a face they share with each other and that the resulting lines are the same as well.

The only thing left to consider is the fact that by inheriting intersection points (and their connections) from a neighboring cube, it might happen that some of these points lie exactly on one edge of the face, where the inheritance takes place. Therefore they do not only affect the face where the intersections has taken place, but the neighboring face along the specific edge as well. If the *number of points that lie on the edge stays the same* (i.e. no additional points that lie on the edge have been inherited), the new lines can be adopted without any further adaptations, since in this case, the intersection points must be the same. This is guaranteed, since we use exact intersections.

If the *number increases* (a decrease in the number of intersection point is not possible), we have to modify the lines of the neighboring face (i.e. move their vertices) in order to obtain a valid mesh. Since the number of intersections between two neighboring cubes can only differ by the amount of 1 (this is obvious, because the level difference of two neighboring cubes must not exceed one, and only one intersection point per edge is generated), only two different cases for each edge (one with a vertex with odd value, the other one with a vertex with an even value) have to be considered. Figure 6 shows both of them.

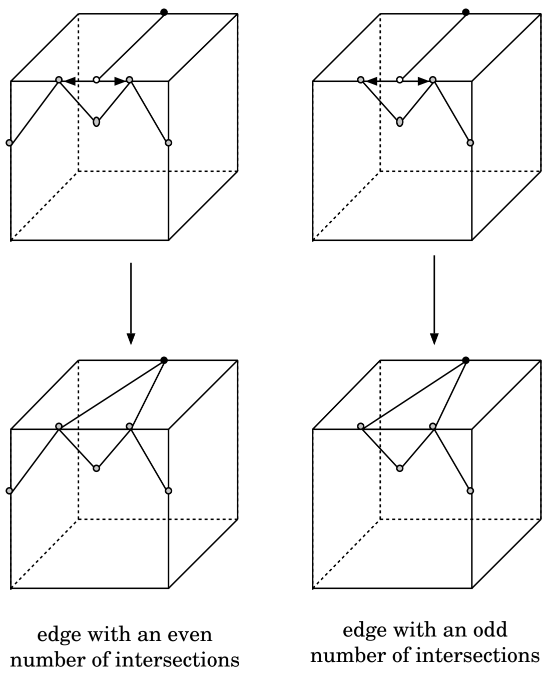


Figure 6

In the last step all local meshes (of the desired resolution) are combined to form the global mesh. If a part of the object has not been refined to this desired resolution, the local mesh at a lower level of detail is taken (which has a sufficient accuracy). Due to the patching described above all local meshes fit together and no T-vertices are produced.

Implementation

ACSGM was developed under IRIX 5.3 on a Silicon Graphics workstation using the SGI Delta/C++ compiler. Since the algorithm does not require any specific hardware, it can easily be ported to any other platform.

The software environment we used for the implementation of the algorithm was VEGA (**V**ienna **E**nvironment for **G**raphics **A**pplications), a C++ library developed by Tobler, Löffelmann and Purgathofer [6], that greatly facilitates most of the common tasks in computer graphics.

Most of the data structures used by ACSGM were based on BABEL (**B**uilding **A** **Be**tter **L**anguage). BABEL again is a C++ class library, that was developed by Tobler, Löffelmann and Galla [REFRENCE], which enhances C++ by some special modeling classes.

Results

In order to point out the advantages of surface correction, a plain cube was used as input to the algorithm. For the visualization perspective projection was chosen.

Meshes created with surface correction turned on are compared with meshes created without surface correction.

The right part of figure 7 shows a level 1 mesh consisting of 8 polygons, that was generated without surface correction. Obviously this mesh only provides a very poor approximation of the original object, since all vertices are cut off.

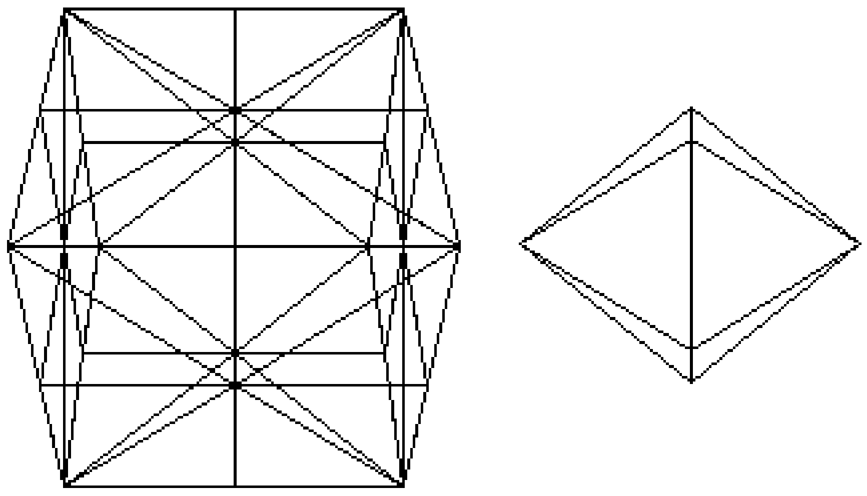


Figure 7

The left part of figure 7 shows a level 1 mesh which was generated with surface correction turned on. This mesh, which is made up of 48 polygons, provides a *perfect approximation* of the original CSG object (i.e. the maximal distance between the polygons of the mesh and the surface of the original object is 0).

Summary and further outlook

The presented meshing algorithm produces a triangular mesh out of a CSG object using an adaptive hierarchical marching cube algorithm based on an octree.

Although the new algorithm is very well suited for applications which need an early (rough) approximation of the CSG model which can be incrementally refined, several improvements are possible.

Since ACSGM produces obtuse triangles in some cases, these have to be dealt with in the application software, or an additional step of mesh relaxation can be appended.

A suitable mesh relaxation could also be used to increase the accuracy of mesh vertices located at G^1 discontinuities of the objects, since these vertices will normally be approximated according to the intersection of the tangential planes in the vicinity of the discontinuity.

In order to speed up the algorithm, it might be desirable to delay the use of the ray caster till a specific level of detail has been reached, since the computation of intersection points between the edges of the voxel and the CSG object is a rather time consuming task.

Another possible enhancement as far as execution time is concerned is to prevent the mesh generation in cases where it is known, that the result will be inaccurate and that a finer subdivision has to take place anyway.

References

[1] E. Boender, *Finite Element Mesh Generation from CSG Models*, PhD thesis, Technical University Delft, 1992.

[2] A.T. Campell and D.S. Fussell, Adaptive Mesh Generation for Global Diffuse Illumination, *Computer Graphics*, **24**, 4, [155–164], August 1990.

[3] P. Hanrahan, D. Salzman, and L. Aupperle, A Rapid Hierarchical Radiosity Algorithm, *Computer Graphics*, **25**, 4, [197–206], July 1991.

[4] W.E. Lorensen and H.E. Cline, Marching Cubes: A High Resolution 3D Surface Construction Algorithm, *Computer Graphics*, **21**, 4, [163–169], July 1987.

[5] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recepies in C*, Press Syndicate of the University of Cambridge, 1988.

[6] R.F. Tobler, H. Löffelmann, and W. Purgathofer, VEGA - Vienna Environment for Graphics Applications, In *Winter School of Computer Graphics, Plzen*, February 1995.