Voxelization of Free-form Solids using Catmull-Clark Subdivision Surfaces

Shuhua Lai and Fuhua (Frank) Cheng
Graphics & Geometric Modeling Lab, Department of Computer Science
University of Kentucky, Lexington, Kentucky 40506-0046

March 10, 2006

Abstract. A voxelization technique and its applications for objects with arbitrary topology are presented. With parametrization techniques for subdivision surfaces becoming available, it is possible now to model and represent any continuous but topologically complex object with an analytical and mathematical representation. In this paper we propose a method to convert a free-form object from its continuous geometric representation into a set of voxels that best approximates the geometry of the object. Unlike traditional 3D scan-conversion methods, our voxelization method is performed by recursively subdividing the 2D parameter space and sampling 3D points from selected 2D parameter positions. Because we can calculate every 3D point position explicitly and accurately, uniform sampling on surfaces with arbitrary topology is not a problem any more. Moreover, our discretization of 3D closed objects is guaranteed to be leak-free when a 3D flooding operation is performed. This is ensured by proving that our voxelization results satisfy the properties of seperability, accuracy and minimality. In addition, a 3D volume flooding algorithm using dynamic programming techniques is presented which significantly speeds up the volume flooding process. Hence our method is suitable for visualization of complex scene, measuring object volume, mass, surface area, determining intersection curve of multiple surfaces and even performing accurate Boolean/CSG operations. These capabilities are demonstrated by test examples shown in the paper.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - curve, surface, solid and object representations;

Keywords: voxelization, subdivision, Catmull-Clark surfaces, visualization, parametrization.

1 Introduction

Volume graphics [9] represents a set of techniques aimed at modeling, manipulation and rendering of geometric objects, which have proven to be, in many aspects, superior to traditional computer graphics approaches. The main advantages of volume graphics are: (1) Decoupling of voxelization from rendering, (2) Uniformity of representation, and (3) Support of Boolean, block and CSG operations. Two drawbacks of volume graphics techniques are their high memory and processing time demands. However, due to the progress in both computers and specialized volume rendering hardware, these drawbacks are gradually losing their significance.

To be represented by the voxel raster, a geometric object has to be subjected to a process called voxelization. This process is concerned with converting geometric objects from their continuous geometric representation into a set of voxels that best approximates the continuous object. Traditional voxelization methods (also referred to as 3D scan-conversion) mimic the 2D scan-conversion process that pixelizes (rasterizes) 2D geometric objects. Hence Traditional voxelization methods only work well for polygon based 3D objects. For surfaces with arbitrary topology, voxelization using 3D scan-conversion is not efficient, nor accurate.

Subdivision surfaces have become popular recently in graphical modeling, visualization and animation because of their capability in modeling/representing complex shape of arbitrary topology [1], their relatively high visual quality, and their stability and efficiency in numerical computation. Subdivision surfaces can model/represent complex shape of arbitrary topology because there is no limit on the shape and topology of the control mesh of a

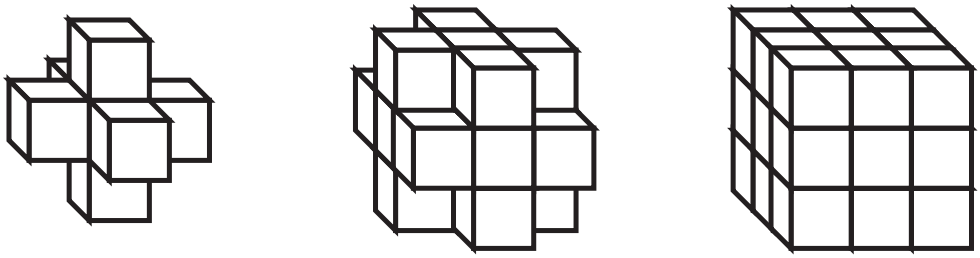


Figure 1: N-neighborhood, $N \in \{6, 18, 26\}$.

subdivision surface. With the parametrization technique for subdivision surfaces becoming available [2] and with the fact that non-uniform B-spline and NURBS surfaces are special cases of subdivision surfaces becoming known [7], we now know that subdivision surfaces cover both *parametric forms* and *discrete forms*. Parametric forms are good for design and representation, discrete forms are good for machining and tessellation (including FE mesh generation). Hence, we have a representation scheme that is good for all graphics and CAD/CAM applications.

In this paper we will propose a new voxelization method for free-form solids presented by Catmull-Clark subdivision surfaces. Different from traditional voxelization methods, our method is based on recursive sampling 2D parameter points of a surface patch. Hence it is more efficient in that it avoids direct sampling on 3D points, which is more expensive and more sensitive to numerical error.

Note that a voxelization process does not render the voxels but merely generates a database of the discrete digitization of the continuous object [8]. Some previous voxelization methods use quad-trees to store the voxelization result. Using quad-tree can save some memory spaces but may sacrifice in time when it is used for applications such as Boolean operations or intersection curves determination. With cheap and giga-byte memoery chips becoming available, time consuming is more concerned. Our method stores the voxelization result directly in a *Cubic Frame Buffer* for fast operation purposes.

2 background

2.1 3D Discrete Spaces [8]

The 3D discrete space is a set of integral grid points in 3D Euclidean space defined by their Cartesian coordinates (x, y, z), with $x, y, z \in Z$. A voxel is a unit cube centered at the integral grid point. Usually the voxel value is either 0 or 1. The voxels assigned '1' are called the 'black' voxels representing opaque objects, and those assigned '0' are the 'white' voxels representing the transparent background. Outside the scope of this paper are non-binary approaches where the voxel value is mapped onto the interval [0,1] representing either partial coverage, variable densities, or graded opacities. Due to its larger dynamic range of values, this approach may support higher quality rendering.

Two voxels are *26-adjacent* (See Fig. 1 (right)) if they share either a vertex, an edge, or a face. Every voxel has 26 such adjacent voxels: eight share a vertex (corner) with the center voxel, twelve share an edge, and six share a face. Accordingly, face-sharing voxels are defined as *6-adjacent* (See Fig. 1 (left)), and edge-sharing and face-sharing voxels are defined as *18-adjacent* (See Fig. 1 (middle)).

The prefix N is used to define the adjacency relation, where N= 6, 18, or 26. We say that a sequence of voxels having the same value (e.g., 'black') is an *N-path* if all consecutive pairs are N-adjacent. A set of voxels are *N-connected* if there is an N-path between every pair of these voxels. It is easy to see that N-connectedness is an equivalence relation. Let A, B and C be three disjoint sets of voxels. Then A is said to *N-separate* B and C if any N-path from a voxel of B to a voxle of C meets A.

2.2 Catmull-Clark Subdivision Surfaces

Catmull-Clark subdivision scheme provides a powerful method for building smooth and complex surfaces. Given a control mesh, a *Catmull-Clark subdivision surface* (CCSS) is generated by iteratively refining (subdividing) the

control mesh to form new and finer control meshes [1]. The mesh refining process consists of defining new vertices and connecting the new vertices to form new edges and faces of a new control mesh. A CCSS is the limit surface of the refined control meshes. The limit surface is called a *subdivision surface* because the mesh refining process is a generalization of the uniform B-spline surface *subdivision technique*. The *valence* of a mesh vertex is the number of mesh edges adjacent to the vertex. A mesh vertex is called an *extra-ordinary vertex* if its valence is different from four. A mesh face with an extra-ordinary vertex is called an *extra-ordinary face*. The *valance* of an extra-ordinary face is the valence of its extra-ordinary vertex. In the following, for the sake of simplicity, a mesh face and the corresponding surface patch will be treated the same and denoted by the same notation.

As we can see that the number of faces in the uniformly refined meshes increases exponentially with respect to subdivision depth. Hence it is impossible to accurately sample 3D points directly on subdivided surfaces. Fortunately, parametrization technieques have become available recently. Therefore efficient and accurate sampling for voxelization is not a problem any more. Given an extra-ordinary face $\mathbf{S}$. If the valence of its extra-ordinary vertex is n, then the surface patch corresponding to this extra-ordinary face is only influenced by $2n + 8$ control vertices. Recent parametrization techniques show that every 3D point (its position, normal and partial derivatives) on the limit surface of a patch can be explicitly and accurately calculated. We will introduce the most recent parametrization techniques for CCSS in the next section.

3 Related Work

3.1 Approaches for Voxelization

Voxelization approaches can be classified into two major types. One is the methods that extend the widely known 2D scan-line algorithms and employ numerical considerations to guarantee that no gaps appear in the the resulting discretization. As we know polygons are fundamental primitives for 3-D surface graphics in that they approximate arbitrary surfaces as a mesh of polygonal patches. Hence, early work on voxelization was focused on voxelizing 3D polygon meshes [10, 11, 12, 13, 14] by using 3D scan-conversion algorithm. Although this type of methods can be extended for voxelizing parametric curves, surfaces and Volumes [15], It is difficult to deal with non-polygonally represented surfaces with arbitrary topology.

The another widely used approaches for voxelizing free-form solids is to use spatial enumeration algorithms which employ point or cell classification methods in either an exhaustive fashion or by recursive subdivision [18, 19, 20, 21]. However, 3D space subdivision techniques for model decomposition into cubic subspaces are computationally expensive and thus inappropriate for medium or high resolution grids. In this paper we will present a technique for voxelization using recursive subdivision. Different from previous methods, our method performs recursive subdivision on 2D parameter space. Hence expensive distance computation between 3D points is easily avoided.

Just like 2D pixelizaiton, voxelization is a very useful technique for representing and modeling complex objects. Recent research work has shown many successful applications using volume graphics. For example voxelization can be used for visualization of complex objects or scene [19]. It can also be used for measuring some integral properties of solids, such as mass, volume and surface area [21]. Most importantly, it can be usde for intersection curve calculation or performing accurate Boolean operations. For example, in [20, 22], a series of Boolean operations are performed on objects represented by a CSG tree. Voxelization is such an important technique that several hardware implementations of this technique has been reported recently [16, 17].

3.2 Evaluation of a CCSS Patch

Several approaches [2, 3, 4, 5] have been proposed for exact evaluation of an extraordinary patch at any parameter point (u, v). In this paper, we will use the parametrization technique presented in [5], because this parametrization/evaluation method is numerically stable, employes less eigen basis functions and can be used for evaluating the 3D position and normal vector of any point in the limit surface exactly and explicitly. Some most related results of [5] are summarized as follows.

The parametrization/evaluation approach in [5] is presented for general Catmull-Clark subdivision surface.

That is, the new *vertex point* $\mathbf{V}'$ of $\mathbf{V}$ after one subdivision is computed as follows:

$$\mathbf{V}' = \alpha_n \mathbf{V} + \beta_n \sum_{i=1}^{n} \mathbf{E}_i + \gamma_n \sum_{i=1}^{n} \mathbf{F}_i$$

where α_n, β_n and γ_n are positive numbers and $\alpha_n + \beta_n + \gamma_n = 1$. In a general Catmull-Clark subdivision surface, all new *face points* and *edge points* are computed the same way as in an ordinary Catmull-Clark subdivision surface [1].

The parametrization and evaluation of a surface patch can be written explicitly as follows [5].

$$\mathbf{S}(u,v) = W^T \mathrm{K}^m \sum_{j=0}^{n+5} \lambda_j^{m-1} \, \mathrm{M}_{b,j} \, G. \tag{1}$$

where n is the valance of the extraordinary patch, W is a vector containing the 16 B-spline power basis functions and K is a constant diagonal matrix. G is the vector of the $2n+8$ control points of the patch. In addition, m and b are two real numbers dependent on (u,v) and can be calculated directly from (u,v) [5]. λ and $\mathrm{M}_{b,j}$ are independent of (u,v) and their exact expressions are shown in [5]. One can compute the derivatives of $\mathbf{S}(u,v)$ to any order simply by differentiating $W(u,v)$ in Eq. (1) accordingly. With the explicit expression of $S(u,v)$ and its partial derivatives, one can easily get the limit point of an extraordinary vertex in a general Catmull Clark subdivision surface:

$$\mathbf{S}(0,0) = [1, 0, \cdots, 0] \cdot M_{2,n+1} \cdot G$$

and the first and second derivatives:

$$\begin{array}{lcl}
\mathbf{D}_u & = & [0,1,0,0,0,0,0,\cdots,0] \cdot \mathrm{M}_{2,2} \cdot G \\
\mathbf{D}_v & = & [0,0,1,0,0,0,0,\cdots,0] \cdot \mathrm{M}_{2,2} \cdot G \\
\mathbf{D}_{uu} & = & [0,0,0,2,0,0,0,\cdots,0] \cdot \mathrm{M}_{2,2} \cdot G \\
\mathbf{D}_{uv} & = & [0,0,0,0,1,0,0,\cdots,0] \cdot \mathrm{M}_{2,2} \cdot G \\
\mathbf{D}_{vv} & = & [0,0,0,0,0,2,0,\cdots,0] \cdot \mathrm{M}_{2,2} \cdot G
\end{array}$$

where $\mathrm{M}_{2,n+1}$ and $\mathrm{M}_{2,2}$ are of dimension $16 \times (2n+8)$ constant matrices [5], $\mathbf{D}_u$, $\mathbf{D}_v$, $\mathbf{D}_{uu}$, $\mathbf{D}_{uv}$ and $\mathbf{D}_{vv}$ are the direction vectors of $\frac{\partial \mathbf{S}(0,0)}{\partial u}$, $\frac{\partial \mathbf{S}(0,0)}{\partial v}$, $\frac{\partial^2 \mathbf{S}(0,0)}{\partial u \partial u}$, $\frac{\partial^2 \mathbf{S}(0,0)}{\partial u \partial v}$ and $\frac{\partial^2 \mathbf{S}(0,0)}{\partial v \partial v}$, respectively. The normal vector at $(0,0)$ is the cross product of $\mathbf{D}_u$ and $\mathbf{D}_v$.

4 Voxelization based on Recursive Parameter Space Subdivision

4.1 Basic Idea

Given a free-form object represented by a CCSS (i.e. its control mesh is known) and the cubic frame buffer resosultion $M_1 \times M_2 \times M_3$, the goal is to convert the CCSS represented free-form object (i.e. continuous geometric representation) into a set of voxels that best approximates the geometry of the object. We assume each face of the control mesh is a quadrilateral and each face has at most one extra-ordinary vertex (a vertex with a *valence* different from 4). If this is not the case, simply perform Catmull-Clark subdivision on the control mesh of the CCSS twice.

With parametrization techniques for subdivision surfaces becoming available, it is possible now to model and represent any continuous but topologically complex object with an analytical and mathematical representation [2, 3, 4, 5]. In other words, any point in the surface can be explicitly calculated. On the other side, for any given parameter point (u,v) in the parameter space, a surface point $S(u,v)$ according to this paramter can be exactly computed as well. Hence, different from previous recursive voxelization methods, which voxelize solids directly in the 3D object space, our voxelization method recursively performs subidivisons and testing on the 2D parameter space.

First we consider how to voxelize a subpatch, which is a small portion of a patch. Given a subpatch of $\mathbf{S}(u,v)$ defined on $[u_1,u_2] \times [v_1,v_2]$, we voxelize it by assuming this given subpatch is small enough (hence flat enough) so that all the voxels generated from it are the same as the voxels generated using only its four corners:

$$\mathbf{V}_1 = \mathbf{S}(u_1,v_1), \quad \mathbf{V}_2 = \mathbf{S}(u_2,v_1), \quad \mathbf{V}_3 = \mathbf{S}(u_2,v_2), \quad \mathbf{V}_4 = \mathbf{S}(u_1,v_2).$$

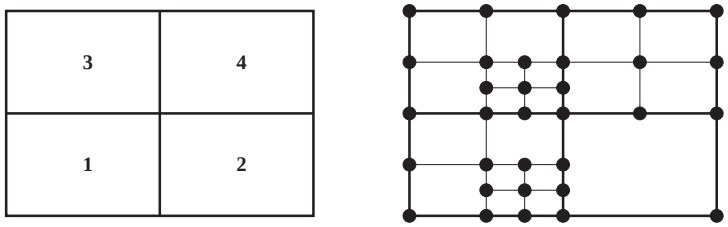


Figure 2: Basic idea of parameter space based recursive voxelization.

Usually this assumpsion does not hold. Hence a test must be done before the patch or subpatch is voxelized. It is easy to see that if the voxels generated using only its four corners are not N-neighboring ($N \in \{6, 18, 26\}$) to each other, then there exist holes between them. In this case, the patch or subpatch is still not small enough. To make it smaller, we perform a *midpoint subdivision* on the corresponding parameter space by setting

$$u_{12} = \frac{u_1 + u_2}{2} \quad \text{and} \quad v_{12} = \frac{v_1 + v_2}{2}$$

to get four smaller subpatches:

$$\mathbf{S}([u_1, u_{12}] \times [v_1, v_{12}]), \quad \mathbf{S}([u_{12}, u_2] \times [v_1, v_{12}]), \quad \mathbf{S}([u_{12}, u_2] \times [v_{12}, v_2]), \quad \mathbf{S}([u_1, u_{12}] \times [v_{12}, v_2]),$$

and repeat the testing process on each of the subpatches. The process is recursively repeated until all the subpatches are small enough and can be directly voxelized using only their four corners.

The vertices of the resulting subpatches after the recursive parameter space subdivision are then used as vertices for voxelziation that approximates the limit surface. For example, if the four rectangles in Figure 2(a) are the parameter spaces of four adjacent subpatches of $\mathbf{S}(u, v)$, and if the rectangles shown in Figure 2(b) are the parameter spaces of the resulting subpatches when the above recursive testing process stops, then 3D points will be directly *evaluated* at the 2D paramter points marked with small solid circles to form vertices for voxelization of the limit surface.

To make things simple, we first normalize the input mesh to $[0, M_1 - 1] \times [0, M_2 - 1] \times [0, M_3 - 1]$. Then for any 2D paramter point (u, v) generated from the recursive testing process (See Fig. 2), direct and exact evaluation is employed to get its 3D surface position and normal vecotr at $S(u, v)$. Now it is very esay to get the voxelized coordinates (i, j, k) from $S(u, v)$ simply by setting

$$i = \lfloor S(u, v).x + 0.5 \rfloor, \quad j = \lfloor S(u, v).y + 0.5 \rfloor, \quad k = \lfloor S(u, v).z + 0.5 \rfloor. \tag{2}$$

Once every single point marked in the recursive testing process is voxelized, the process for voxelzing the given patch or subpatch is finished. The proof about the correctness of our voxelization results will be discussed in the next section.

Since the above process guarantees shared boundary or vertex of patches or subpatches will be voxelized to the same voxels, we can perform voxelzation for free-form objects represented with a CCSS patch by patch. A point needed to be pointed out is that to avoid stack overflow, only small subpatches are fed to the recursive subdivision and testing process. This is especially true when high resolutions for the cubic frame buffer are given or some polygons are very big in the given input mesh. Generating small subpatches is not a problem for a CCSS with the parametrization techniques becoming available. For example, in our implementation, the size (in the parameter space) of subpatches fed for recursive testing is $\frac{1}{8} \times \frac{1}{8}$, i.e. each patch is divided into 8×8 subpatches before voxelization. In addition, small size of feeding subpatches ensures the assumption of our basic assumption of voxelization to be valid, because the smaller parameter size a subpatch has, the flatter the subpatch is.

4.2 Algorithms for Voxelization

Here we summarize our recursive subdivision on parameter space based voxelization method into the following algorithms: *Voxelization* and *VoxelizeSubPatch*. The parameters to these algorithms are explained as follows. S:

the input mesh which represents the surface of an object. N: a parameter used to specify the N-neighborhood relation between neighboring generated voxels. M_1, M_2, and M_3: Resoltion for the Cubic Frame Buffer. k: a parameter used to specify that each patch is divided into $k \times k$ subpatches before fed to the recursive voxelization process.

Voxelization(Mesh S, int N, int M_1, int M_2, int M_3, int k)
1. normalize S such that S is bounded by an axle-aligned cube of dimension $[0, M_1-1] \times [0, M_2-1] \times [0, M_3-1]$
2. for each patch *pid* in S
3. for $u = \frac{1}{k} : 1$, step size $\frac{1}{k}$
4. for $v = \frac{1}{k} : 1$, step size $\frac{1}{k}$
5. VoxelizeSubPatch(N, *pid*, $u - \frac{1}{k}$, u, $v - \frac{1}{k}$, v);

VoxelizeSubPatch(int N, int *pid*, float u_1, float u_2, float v_1, float v_2)
1. $(i_1, j_1, k_1) = \text{Voxelize}(S(pid, u_1, v_1))$;
2. $(i_2, j_2, k_2) = \text{Voxelize}(S(pid, u_2, v_1))$;
3. $(i_3, j_3, k_3) = \text{Voxelize}(S(pid, u_2, v_2))$;
4. $(i_4, j_4, k_4) = \text{Voxelize}(S(pid, u_1, v_2))$;
5. $\Delta_i = \max\{|i_a - i_b|\}$, with a and $b \in \{1, 2, 3, 4\}$;
6. $\Delta_j = \max\{|j_a - j_b|\}$, with a and $b \in \{1, 2, 3, 4\}$;
7. $\Delta_k = \max\{|k_a - k_b|\}$, with a and $b \in \{1, 2, 3, 4\}$;
8. if($N = 6$ & $\Delta_i + \Delta_j + \Delta_k \leq 1$) return;
9. if($N = 18$ & (($\Delta_i \leq 1$ & $\Delta_j + \Delta_k \leq 1$) | ($\Delta_j \leq 1$ & $\Delta_i + \Delta_k \leq 1$) | ($\Delta_k \leq 1$ & $\Delta_i + \Delta_j \leq 1$)) return;
10. if($N = 26$ & $\Delta_i \leq 1$ & $\Delta_j \leq 1$ & $\Delta_k \leq 1$) return;
11. $u_{12} = (u_1 + u_2)/2$; $v_{12} = (v_1 + v_2)/2$;
12. VoxelizeSubPatch($N, pid, u_1, u_{12}, v_1, v_{12}$);
13. VoxelizeSubPatch($N, pid, u_{12}, u_2, v_1, v_{12}$);
14. VoxelizeSubPatch($N, pid, u_{12}, u_2, v_{12}, v_2$);
15. VoxelizeSubPatch($N, pid, u_1, u_{12}, v_{12}, v_2$);

In the above algorithm 'VoxelizeSubPatch', positions for the four corners are directly evaluated using eq. (1), where *pid* tells us which patch we are currently interested in. And the routine 'Voxelize' voxelizes points by using eq. (2). The lines 8, 9 and 10 are used to test if voxelizing the four corners of a subpatch is enough for generating a 6-, 18- and 26-neighborhood voxelization, respectively.

5 Separability, Accuracy and Minimality

Let S be a C^1 continuous surface in R^3, we denote by $\bar{S}$, the discrete representation of S. $\bar{S}$ is a set of black voxels generated by some digitalization methods. There are three major requirements that $\bar{S}$ should meet in the voxelization process. First, *separability* [8, 13], which requires to preserve the analogy between continuous and discrete space and to guarantee that $\bar{S}$ is not penetrable since S is C^1 continuous. Second, *accuracy*. This requirement ensures $\bar{S}$ is the most accurate discrete representation of S according to some appropriate error metric. Third, *minimality* [8, 13], which requires the voxelization should not contain voxels that, if removed, make no difference in terms of separability and accuracy. The mathematic definitions for these requirements can be found in [13], which are based on [8].

First we can see that voxelization results generated using our recursive subdivision method satisfy the requirement of minimality. The reason is that voxels are sampled directly from the object surface. The termination condition of our recursive sampling process (i.e., Line 8, 9, 10 in algorithm 'VoxelizeSubPatch') and the coordinates transformation in eq. (2) guarantee every point in the surface has one and only one image in the resulting voxelization. In other words,

$$\forall\ P \in S,\ \exists\ Q \in \bar{S},\ \text{ such that }\ P \in Q. \tag{3}$$

Note here P is a 3D point and Q is a voxel, which is a unit cube. On the other hand, because all voxels are

mapped directly from the object surface using eq. (2), we have

$$\forall\ Q \in \bar{S},\ \exists\ P \in S,\ \text{ such that }\ P \in Q. \tag{4}$$

Hence no voxel can be removed from the resulting voxelization, i.e., the property of minimality is satisfied. In addition, from eq. (3) and eq. (4) we also can conclude that the resulting binary voxelization is the most accurate one with respect to the given resolution. Hence the property of accuracy is satisfied as well.

To prove that our voxelization results satisfy the separability property, we only need to show that there is no holes in the resulting voxelization. For simplisity, here we only consider 6-sperability, i.e., there does not exist a ray from a voxel inside of the free-form solide object to the outside of the free-form solide object in x, y or z direction that can penetrate our resulting voxelization without intersecting any of the black voxels. Now we prove the separability property by contradiction. As we know violating separability means there exists at least a hole (voxel) Q in the resulting voxelization that is not included int $\bar{S}$ but is intersected by S and, there must also exist two 6-neighbors of Q that are not included in $\bar{S}$ either and are on opposite sides of S. Because S intersects with Q, there exist at least one point P on the surface that intersects with Q. But the image of P after voxelization is not Q because Q is a hole. However, the image of P after voxelization must exist because of the termination condition of our recursive sampling process (i.e., Line 8, 9, 10 in algorithm 'VoxelizeSubPatch'). Moreover, according to our voxelization method, P only can be voxelized into voxel Q because of eq. (2). Hence Q cannot be a hole, contradicting our assumption. Therefore, we conclude that $\bar{S}$ is 6-separating.

6 Volume Flooding with Dynamic Programming

6.1 Seed Selection

A seed must be designated before a flooding algorithm can be applied. In 2D area flooding, a seed usually is given by users interactively. However in 3D flooding, for a closed 3D object, it is impossible for a user to designate a voxle as a seed simply by mouse-clicking because voxels inside a closed 3D object are invisiable. Hence an automatic method is needed for selecting a inside voxel as a seed for volume flooding. Once we can correctly choose a inside voxel, the by applying a flooding operation, all inside voxels can be obtained. To select a voxel as a seed for volume flooding, we must be able to tell if this voxel is located inside or outside of the 3D object. This is not a trivial problem. In the past In-Out test for voxles is efficient and not accurate [21], especially for topologically complicated 3D objects.

With the availabilty of parametrization of subdivision surfaces, we now can calculate derivatives and normals exactly and explicitly for each point located on the 3D object surface. Hence the normal for each voxel can also be exactly calculated in the voxelization process. Because the direction of normals is perpendicular to the surface and points to the ouside of the surface, the closest voxel in its oppsite direction should be located either inside or on the surface. To choose the closest voxel in its oppsite direction, we just need to calculate the dot product of the normal and one of the axle vectors. These vectors are: $\{1,0,0\}$, $\{-1,0,0\}$, $\{0,1,0\}$, $\{0,-1,0\}$, $\{0,0,1\}$, $\{0,0,-1\}$ corresponding to x, $-x$, y, $-y$, z and $-z$ direction, respectively. the dirceciton with biggest dot product is chosen for calculating a inside voxel. if the closest voxel in this chosen direction is also a black voxel (i.e., located on the 3D object surface), another point on surface has to be selected and the above process is repeated until a inside voxel is found. The found inside voxel can be designated as a seed for inside volume flooding. Similarily, an outside voxel can also be found for outside volume flooding. In this case, the seed voxel should not be chosen from the normal's oppsite direction, but along the normal's direction.

6.2 A 3D Flooding Algorithm using Dynamic Programming

In this section we only consider flooding algorithms using 6-seperability, but the idea can be applied to N-seperability. Although 6-seperability is used in the flooding process, the voxelization itself can be N-neighboring. Once a seed is chosen, 3D flooding algorithms can be performed in order to fill all the voxels that are 6-connected with this seed voxel. The simplest flooding algorithm is the *recursive flooding*, which recusively search neighboring voxels in 6 directions for 6-connected voxels. This method sounds ideally reasonable but does not work in real world because for a even very low resolution, it still causes stack overflow.

Another method can be used for flooding is called *linear flooding*, which searchs neighboring voxels that are 6-connected with the given seed voxel, linearly from the first voxel to the last voxel in the cubic frame buffer, and marks all found voxels into gray. The search process is repeated until no more white ('0') voxels is found that are 6-connected with one of the gray voxels. Linear flooding is simple and does not require any extra memory in the flooding process. However, it is very slow, especially when a high resolution is used in the voxelization process.

In many applications, 3D flooding operations are required to be fast with low extra memory consumption. To make a 3D flooding algorithm applicable and efficient, we can combine the recursive flooding and the linear flooding methods using the so called dynamic programming technique.

Dynamic programming usually breaks a problem into subproblems, and these subproblems are solved and the solutions are memorized, in case they need to be solved again. This is the essentiality of dynamic programming. To use dynamic programming in our 3D flooding algorithm, we use a sub-routine *FloodingXYZ* which marks inside voxels only in x, y and z directions from a seed voxel, and all marked voxels are memorized by pushing them into a stack called *GRAYSTACK*. Note here the stack has a limited space, whose length is specified by users. When the stack reaches its maximal capacity, no gray voxels can be pushed into it. Hence it guarantees limited memory consumption. The 3D flooding algorithm with dynamic programming can improve the flooding speed significantly. For a ordinary resolution, say, $512 \times 512 \times 512$, a flooding operation can be done almost in real time. The pseudocode for the 3D volume flooding algorithm is given as follows and the parameters (s_i, s_j, s_k) are the coordinates of the given seed voxel.

VolumeFlooding(int s_i, int s_j, int s_k)
1. FloodingXYZ(s_i, s_j, s_k);
2. loop = 1;
3. while(loop)
4. while (GRAYSTACK is not empty)
5. (i, j, k) = GRAYSTACK.Pop();
6. FloodingXYZ(i, j, k)
7. loop = 0;
8. for$(i = 0; i < M_1; i{+}{+})$
9. for$(j = 0; j < M_2; j{+}{+})$
10. for$(k = 0; k < M_3; k{+}{+})$
11. if (Voxel (i, j, k) is white and is 6-neighboring with a gray voxel)
12. FloodingXYZ(i, j, k);
13. loop = 1;

7 Applications

7.1 Visualization of Complex Scene

Ray tracing is commonly used in the field of visualization of volume graphics. This is due to its ability to enhace spatial perception of the scene using techniques such as transparency, mirroring and shadow casting. However, there is one main disadvantage for ray tracing approach: large computational demands. Hence rending using this method is very slow. Recently, surface splatting technique for point based rendering has become very popular. [23]. Surface splatting requires the position and normal of every point to be known, but not their connectivity. With explicit position and exact normal information for each voxel in our voxelization results, now it is much more easy for us to render discrete voxels using surface splatting techniques. The rendering is fast and high quality results can be obtained. For example, Fig. 3(a) and Fig. 3(e) are two given meshes, Fig. 3(b) and Fig. 3(f) are their corresponding limit surfaces. After the voxelization process, Fig. 3(c) and Fig. 3(g) are generated only using basic point rendering techniques with explicitly known normals to each voxel. While Fig. 3(d) and Fig. 3(h) are rendered by using splatting techniques. The size of cubic frame buffer used for Fig. 3(c) and Fig. 3(g) is $512 \times 512 \times 512$. And the voxelization resolution used for Fig. 3(d) and Fig. 3(h) is $256 \times 256 \times 256$. Although the resolution is much lower, we can tell from Fig. 3, that the ones using splatting techniques are more smooth and closer to the corresponding object surfaces given in Fig. 3(b) and Fig. 3(f), respectively.

7.2 Integral Properties Measurement

Another application of voxeliation is that it can be used for measuring some integral properties of solid objects. For example, mass, volume and surface area can be easily calculated. Without discretization, these integral properties are very difficult to be measured, especially for free-form solids with arbitrary topology.

Volume can be measured simply by counting all the voxels inside or on the surface boundary because each voxel is a unit cube. With efficient flooding algorithm, voxels inside or on the boundary can be precisely counted. But the resulting measurement may not be accurate because boundary voxels do not occupy all these corresponding unit cubes. Once volume is known, it is very easy to measure the mass by simply multiplying its density. In addition, surface area can also be measured similarly. But using this method would lead to big error because we do not know how surfaces pass through their corresponding voxels. Fortunately, surface area can be measured much more precisely in the voxeliztation process. As we know, during the recursive voxelization process, if the recursive process stops, all the marked parameter points of a patch or subpatch (See Fig. 2) are points used for final voxelization. Hence all these quadrilaterals corresponding to these marked parameter points can be used for measuring surface area after these marked parameter points are mapped to 3D space. The flatness of these quadrilaterals is required to be tested if high accuracy is needed. The definition of patche flatness and the flatness testing method can be found in [6].

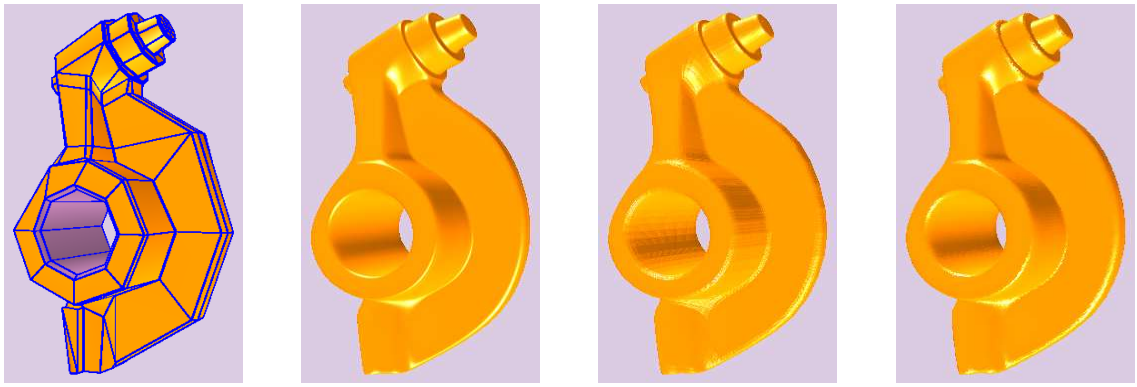
7.3 Performing Boolean and CSG Operations

The most important application of voxelization is to perform Boolean and CSG operations on free-form objects. In solid modelling, an object is formed by performing Boolean operations on simpler objects or primitives. A CSG tree is used in recording the construction history of the object and is also used in the ray-casting process of the object. Surface-surface intersection (including the in-on-out test) and ray-surface intersection are the core operations in performing the Boolean and CSG operations. With voxelization, all of these problems become much easier set operations. Examples of performing Boolean operations on two and three cows are presented in Fig. 3(j) and Fig. 3(k), respectively. A difference operation is first performed to remove some portions from each of these cows and a union operation is then performed to join them together. A mechanichal part is also generated in Fig. 3(n) using CSG operations. Intersection curves can be similarly generated by searching for common voxels of objects. The black curves shown in Fig. 3(i), Fig. 3(i) and Fig. 3(i) are intersection curves generated from two differenct objects.

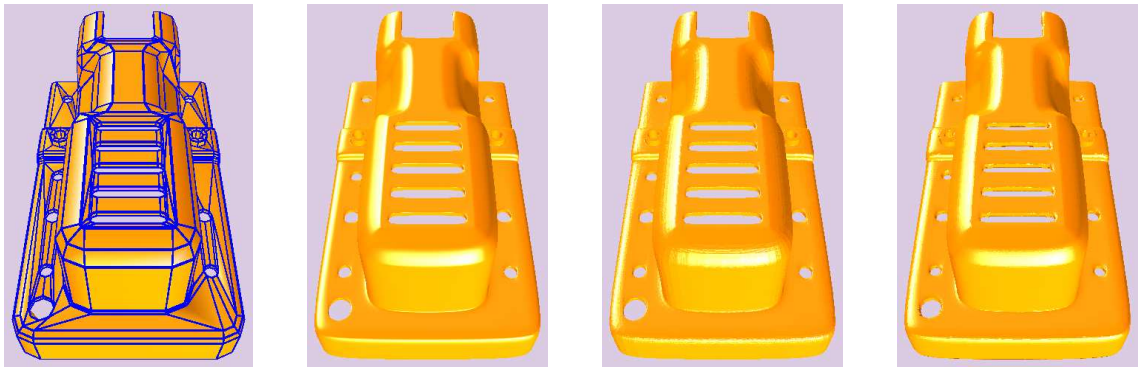
8 Summary

In this paper we propose a method to convert a free-form object from its continuous geometric representation into a set of voxels that best approximates the geometry of the object. Unlike traditional 3D scan-conversion methods, our voxelization method is performed by recursively subdividing the 2D parameter space and sampling 3D points from selected 2D parameter positions. Because we can calculate every 3D point position explicitly and accurately, uniform sampling on surfaces with arbitrary topology is not a problem any more. Moreover, our discretization of 3D closed objects is guaranteed to be leak-free when a 3D flooding operation is performed. This is ensured by proving that our voxelization results satisfy the properties of seperability, accuracy and minimality. In addition, a 3D volume flooding algorithm using dynamic programming techniques is presented which significantly speeds up the volume flooding process. Hence our method is suitable for visualization of complex scene, measuring object volume, mass, surface area, determining intersection curve of multiple surfaces and even performing accurate Boolean/CSG operations.

Acknowledgement. Data sets for Figs. 3(i) and 3(l) are downloaded from the web site:
http://research.microsoft.com/∼hoppe/ .
The data set for the cow model in Figs. 3(j), 3(k) and 3(m) is downloaded from the web site:
http://graphics.cs.uiuc.edu/∼garland/research/quadrics.html.

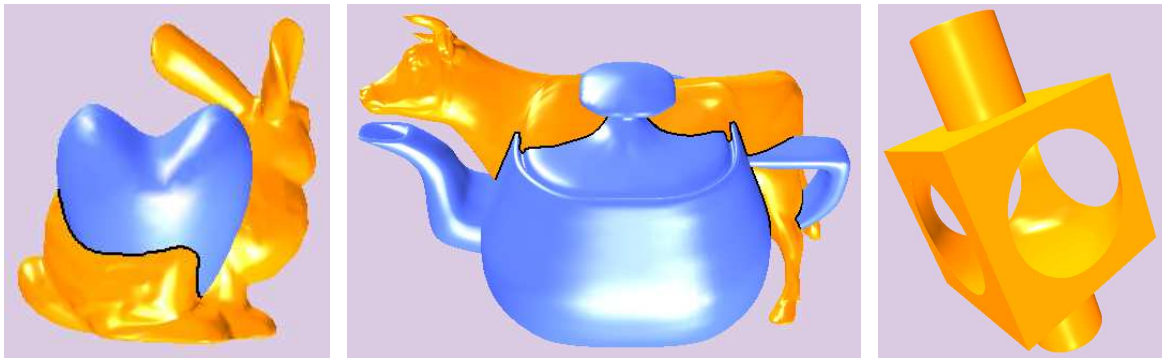


(a) Given Mesh (b) Object Surface (c) Point Rendering (d) Splatting Rendering



(e) Given Mesh (f) Object Surface (g) Point Rendering (h) Splatting Rendering

(i) Intersection Curve (j) Boolean Operations (k) Boolean Operations



(l) Intersection Curve (m) Intersection Curve (n) CSG Operations

Figure 3: Applications of Voxelization

References

[1] Catmull E, Clark J. Recursively generated B-spline surfaces on arbitrary topological meshes, *Computer-Aided Design*, 1978, 10(6):350-355.

[2] Stam J, Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values, *Proceedings of SIGGRAPH* 1998:395-404.

[3] Stam J, Evaluation of Loop Subdivision Surfaces, *SIGGRAPH'99 Course Notes*, 1999.

[4] Zorin D, Kristjansson D, Evaluation of Piecewise Smooth Subdivision Surfaces, *The Visual Computer*, 2002, 18(5/6):299-315.

[5] Shuhua Lai, Fuhua (Frank) Cheng, Parametrization of General Catmull Clark Subdivision Surfaces and its Application, *Computer Aided Design & Applications*, 3, 1-4, 2006.

[6] Shuhua Lai, Fuhua (Frank) Cheng, Near-Optimum Adaptive Tessellation of General Catmull-Clark Subdivision Surfaces, *To appear or submitted*.

[7] Sederberg TW, Zheng J, Sewell D, Sabin M, Non-uniform recursive subdivision surfaces, *Proceedings of SIGGRAPH*, 1998:19-24.

[8] Cohen Or, D., Kaufman, A., Fundamentals of Surface Voxelization, *Graphical Models and Image Processing*, 57, 6 (November 1995), 453-461.

[9] A. Kaufman, D. Cohen. Volume Graphics. *IEEE Computer*, Vol. 26, No. 7, July 1993, pp. 51-64.

[10] D. Haumont and N. Warzee. Complete Polygonal Scene Voxelization, *Journal of Graphics Tools*, Volume 7, Number 3, pp. 27-41, 2002.

[11] M.W. Jones. The production of volume data from triangular meshes using voxelisation, *Computer Graphics Forum*, vol. 15, no 5, pp. 311-318, 1996.

[12] S. Thon, G. Gesquiere, R. Raffin, A low Cost Antialiased Space Filled Voxelization Of Polygonal Objects, *GraphiCon 2004*, pp. 71-78, Moscou, Septembre 2004.

[13] Jian Huang, Roni Yagel, V. Fillipov and Yair Kurzion, An Accurate Method to Voxelize Polygonal Meshes, *IEEE Volume Visualization'98*, October, 1998.

[14] Kaufman, A., An Algorithm for 3D Scan-Conversion of Polygons, *Proc. EUROGRAPHICS'87*, Amsterdam, Netherlands, August 1987, 197-208.

[15] Kaufman, A., Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes, *Computer Graphics*, 21, 4 (July 1987), 171-179.

[16] S. Fang and H. Chen. Hardware accelerated Voxelisation. *Volume Graphics*, Chapter 20, pp. 301-315. Springer-Verlag, March, 2000.

[17] Beckhaus S., Wind J., Strothotte T., Hardware-Based Voxelization for 3D Spatial Analysis *Proceedings of CGIM '02*, pp. 15-20, August 2002.

[18] N. Stolte, A. Kaufman, Efficient Parallel Recursive Voxelization for SGI Challenge Multi-Processor System, *Computer Graphics International*, 1998.

[19] Nilo Stolte, Graphics using Implicit Surfaces with Interval Arithmetic based Recursive Voxelization, *Computer Graphics and Imaging*, pp. 200-205, 2003.

[20] T. Duff, Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry, *SIGGRAPH*, pp. 131–138, July, 1992.

[21] Lee, Y. T. and Requicha, A. A. G., Algorithms for Computing the Volume and Other Integral Properties of Solids: I-Known Methods and Open Issues; II-A Family of Algorithms Based on Representation Conversion and Cellular Approximation, *Communications of the ACM*, 25, 9 (September 1982), 635-650.

[22] S. Fang and D. Liao. Fast CSG Voxelization by Frame Buffer Pixel Mapping. *ACM/IEEE Volume Visualization and Graphics Symposium 2000 (Volviz'00)*, pp. 43-48, Salt Lake City, UT, 9-10 October 2000.

[23] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, Markus Gross, Surface Splatting, *SIGGRAPH 2001*.