Polygonization of Implicit Surfaces with Constructive Solid Geometry

Brian Wyvill[†] Kees van Overveld[‡]

October 22, 1996

[†]Department of Computer Science, University of Calgary
2500 University Dr. NW, Calgary T2N 1N4 Canada
Email: blob@cpsc.ucalgary.ca

[‡]Department of Mathematics and Computer Science, Eindhoven University of Technology
PO Box 513; 5600 MB, Eindhoven, the Netherlands
Email: wsinkvo@win.tue.nl

1 Abstract

A polygonisation algorithm is presented which extends an existing skeletal implicit surface technique to include operations based on Constructive Solid Geometry between blended groups of implicit surface objects. The result is a surface definition (to be called *Boolean Compound Soft Object*, or *BCSO* for short) which consists of a boolean expression with union, intersection, and set difference operators. The geometric primitives that form the operands are *soft objects* bounded by the iso-surfaces resulting from suitable potential fields. These potential fields are parameterized by configurations of so called *skeletal elements*. The resulting system, unlike most CSG systems, combines blended and unblended primitives. The polygonisation algorithm produces a mesh of triangles to facilitate fast viewing and rendering.

keywords Free form surface design, implicit surfaces, constructive solid geometry, computer aided geometric design

2 Introduction

Over the last few years, two techniques for defining complex geometric objects as composites from more elementary geometric primitives have received some attention. They are Constructive Solid Geometry (CSG; [Requicha 80], [Mantyla 88]) and Implicit Surface Modeling based on soft objects (ISM; [Wyvill 86]). Both techniques have been used to construct a wide variety of non-trivial geometric objects, but due to the difference in nature, the application fields of the two methods appears to be quite different. Two earlier system that also combines CSG and skeletal implicit surface objects were developed by Geoff Wyvill [Wyvill 90], using ray tracing to both traverse the CSG tree and render the objects and also a system based on *R-functions* by Alexander Pasko et al [Pasko 95]. In this work our

approach is quite different in that we do not use ray tracing, instead we have developed a polygonizing algorithm to facilitate fast rendering and improve the design cycle of such objects. For animation and model visualization applications it is important to be able to view a model in real time. Currently this can not be done with ray traced models, thus we employ a prototyping scheme that provides for fast visualiztion even at reduced quality. As explained in the sequel, our model visualizations are not of the same quality as ray traced images, however the polygonal meshes can be produced at an arbitrary resolution and different views can be calculated in real time on modern graphics workstations.

In the table below we list some significant differences between CSG features and ISM features.

property	CSG features	ISM features	BCSO Features
primitives	geometric Primitives e.g. sphere, cone, torus	skeletal elements e.g. soft ellipsoids, soft lines, soft polygons,...	unlimited variety of ISM's either skeletal elements or blended groups thereof
junctions	unblended (non-C^1), difficult to get C^1	blended (C^1), difficult to get non-C^1	both C^0 and C^1 in one object
rendering	either output b-rep surface or ray trace while doing CSG-classification for each traced ray	either output polygon mesh or ray-trace finding intersections with ISM for each ray	CSG operations performed on-the-fly during poly-gonisation
main areas of appli-cation	CAGD, industrial design and NC-machining	computer (cartoon-type) animation, (pseudo-) biologic or organic looking shapes, e.g. also in the realm of of industrial design	traditional CSG combined with ISM

As indicated in the table the primitives in standard CSG, are a limited set of closed geometric objects such as the sphere, cone, torus etc. Extended versions can also support primitives bounded by free form surfaces, sweep surfaces, or other deformed primitives ([Crocker 87], [Jansen 87]).

Implicit surface systems also employ geometric primitives, known as skeletal elements, ([Bloomentha 90]). Skeletal elements are geometric shapes that allow easy computation of a distance to a given point in 3-space.

Junctions in the boundary between surface fragments of different CSG primitives are generally not C^1. It requires special primitives to obtain smooth blends ([Middleditc 85]). Alternatively, filleting and rounding operations may apply to the boundary representation of the CSG-object ([Chiokura 83]).

The implicit functions used in ISM that give rise to the resultant iso-surface are in general differentiable everywhere in 3-space, so the surface is smooth everywhere. Since there is no notion of explicitly represented junctions in ISM, it is not possible to get non-C^1 boundaries anywhere. (See however [Gascuel 93]).

CSG and implicit surface systems are similar in that the underlying model description has to be visualized. There are two basic approaches for each type of representation:

- For CSG systems, find a boundary representation (b-rep) and render the model as boundary

fragments (mostly converted to polygon meshes), or alternatively, the object may be ray traced while the CSG-expression evaluation takes place for each ray while being traced ([Jansen 87]).

- An implicit iso-surface, once polygonized, is just a polygon mesh that can be rendered as it stands. Alternatively, it may be rendered directly via ray tracing. Although this is often too computationally expensive for many applications, it has been demonstrated that CSG-type boolean combinations of several iso-surfaces can be obtained by evaluating the boolean expressions along with the ray intersections in a manner similar to standard CSG [Wyvill 90].

In this paper we present an extension to the uniform subdivision algorithm [Wyvill 86] to include CSG operations between blended groups of implicit surface primitives.

Geometric primitives are complete ISM's, so a virtually unlimited variety of primitives is available. Next these primitives are assembled via the usual CSG-type boolean constructors. The skeletal elements within one ISM primitive blend smoothly, so there are no visible non-C^1 junctions within those elements. On the other hand, two ISM primitives are combined in the CSG-sense, and hence a visible junction arises there. So both types of junctions are supported within one surface representation scheme. When ray tracing is employed for ISM's, CSG-type operations can be berformed on-the-fly where the operands are individual ISM's [Wyvill 90], but ray tracing is computationally expensive. On the other hand, when all ISM's are polygonized first, then CSG-type operations can be performed afterwards on the resulting polygon meshes ([Naylor 90]), since they are closed manifolds, but this has a high complexity in terms of the number of triangles in the meshes involved: the fully triangulated meshes of all input ISM's have to be available, even if a given ISM only contributes for a small fraction of its surface. Also, this strategy cannot be used if one the participating CSG-primitives (ISM's) is unbounded, as for instance when intersecting with a planar half space in order to 'cut an object in half'. In the polygonizer presented here, we perform the CSG-operations on-the-fly while polygonizing the *resulting* surface. This means that the complexity is linear in the number of triangles of the resulting surface only, even if some of the contributing ISM's would have given rise to much larger triangular meshes.

In case we use the resulting mesh for further manipulations that require a parametrised representation, we may first want to reduce the number of triangles using resampling ([Schroeder 92]), and next use e.g. *Loop patches* ([Loop 94]), so if need be, ISM's can serve in a CAGD-context.

For the rest, the combined ISM/CSG modelsserve as an extension of the variety of shapes that may be modeled with plain ISM's, so they may be applied in the same areas.

Finally, we observe that the availability of CSG-type operations may extend the versatility of volume visualisation techniques, e.g. in medical imaging where the surfaces of different structures may serve as ISM surfaces in CSG-expressions.

An example of an implicit model which uses CSG operations is shown in fig. 1, the cylinders smoothly blend together, difference operations are used to remove the insides of the cylinders and the result is intersected with two planes. The cylinder in the horizontal direction shows the original shape of the ISM cylinder primitive which has hemispherical ends.

In Section 2, we describe the algorithm for performing CSG-operations in parallel with polygonizing the appropriate parts of ISM surfaces. Although this algorithm gives an adequate and consistent polygonization for the smooth parts of the resulting surface, the non-C^1 junctions are reproduced poorly. This is due to the fact that the voxel structure that underlies our polygonization algorithm has a uniform distribution, which is adequate under the assumption that for smooth surfaces the curvature of the polygonized surface is distributed more or less uniformly over space. (In fact, it

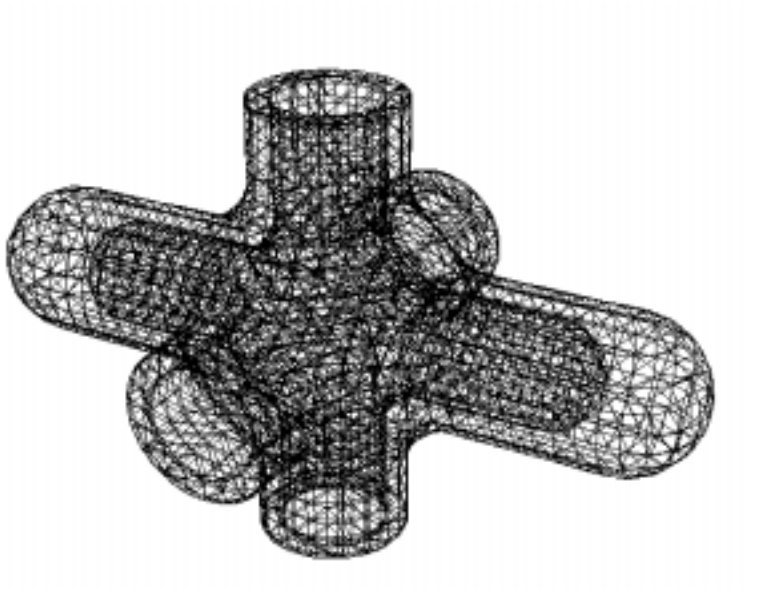

Figure 1: Blended, hollow cylinders, two of them have been intersected with orthogonal planes. The cylinders in the horizontal direction have rounded ends as they are ISM primitives

can be rather wasteful in areas of large curvature radii. Adaptive polygonization techniques (see e.g. [Bloomentha 88]) should be used to make this more efficient, but we do not focus on adaptive techniques here.) The 'smoothness'-assumption does *not* hold, however, in the vicinity of the non-C^1 junctions, and therefore severe voxelization artifacts may show up in these areas. Section 3 discusses a remedy to these problems. Some implementation aspects are the topic of Section 4. Finally, Section 5 summarizes our results.

3 Voxel-based CSG-operations

Given a scalar field function $f = f(x, y, z)$, the Uniform voxel Subdivision Algorithm of [Wyvill 86] estimates intersections of the iso-surface $\{(x, y, z)|f(x, y, z) = 0\}$, to be polygonized, with the 12 edges of a cubic voxel, on the basis of the $f(x_c, y_c, z_c)$ values, $c = 0, \cdots, 7$, in the 8 corner vertices (x_c, y_c, z_c) of that voxel (see fig. 2). Here, the front lower left corner vertex (solid circle) has $f > 0$ whereas the other corner vertices (open circles) have $f < 0$. A vertex with $f > 0$ classifies 'in' with respect to the iso-surface and a vertex with $f < 0$ classifies 'out'. In the case where an intersection of that edge with the iso-surface exists, the extreme vertices of a voxel edge are classified differently. Cases where an edge contains one intersection are indistinguishable from cases where there are any odd number of intersections. Similarly, the occurrence of an even number of intersections goes unnoticed.

On the bases of linear interpolation of the values of f in the corner vertices, the intersection points (solid squares) are estimated. (Alternatively, more sophisticated, but slower, numerical methods such as regula falsi or Newton Raphson may be used to obtain more accurate estimates of the intersections.) The shaded triangle in fig. 2 is the mesh element that originates from this voxel. Any intersections found are connected by piecewise planar surface elements (triangles), and the collection of all these triangles forms the triangulated polygon mesh that approximates the iso-surface.

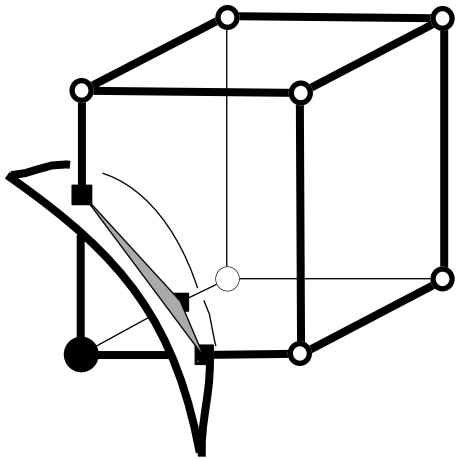

Figure 2: A cubic voxel intersected by an iso-surface.

In order to generalize towards CSG-expressions in iso-surfaces, we assume that instead of a scalar function $f(x, y, z)$, we have an $n-$ component vector function $f_j(x, y, z), j = 0, \cdots, n-1$. Each of the components f_j gives rise to its own iso-surface, each iso-surface can be seen as the boundary of one ISM primitive. The resulting surface has to be constructed such as to bound the appropriate CSG-expression in each of the ISM primitives. In the sequel, the CSG-operations are denoted as $DIFF$, $UNION$, and $INTSCT$, for difference, union, and intersection, respectively. The arguments of these operators will be either numbers of ISM primitives (the above j) or other CSG-operations.

In order to see how this works out, we study a 2-D version first (see fig.3). Here C_1 and C_2 are two iso-value contours that both intersect voxel edge A-B. They give rise to (estimated) intersections p_1 and p_2, respectively. Suppose C_1 is the boundary of ISM primitive 1 whereas C_2 bounds ISM primitive 2, and we want to polygonize the boundary of the object $DIFF(1, 2)$. It can be seen from fig. 3 that p_2 is the relevant intersection of the two.

In general, we observe that depending on the in- or out-classifications of each of the components f_j in A and B we can determine, for each of the operators $DIFF$, $UNION$, $INTSCT$, which of the intersections is the relevant one. Note that there does not always have to be a relevant intersection: if the resulting iso-surface does not pass through the edge AB, the 'relevant intersection' is not defined. The following table gives an exhaustive list of all possible in- and out-cases for two participating objects i and j.

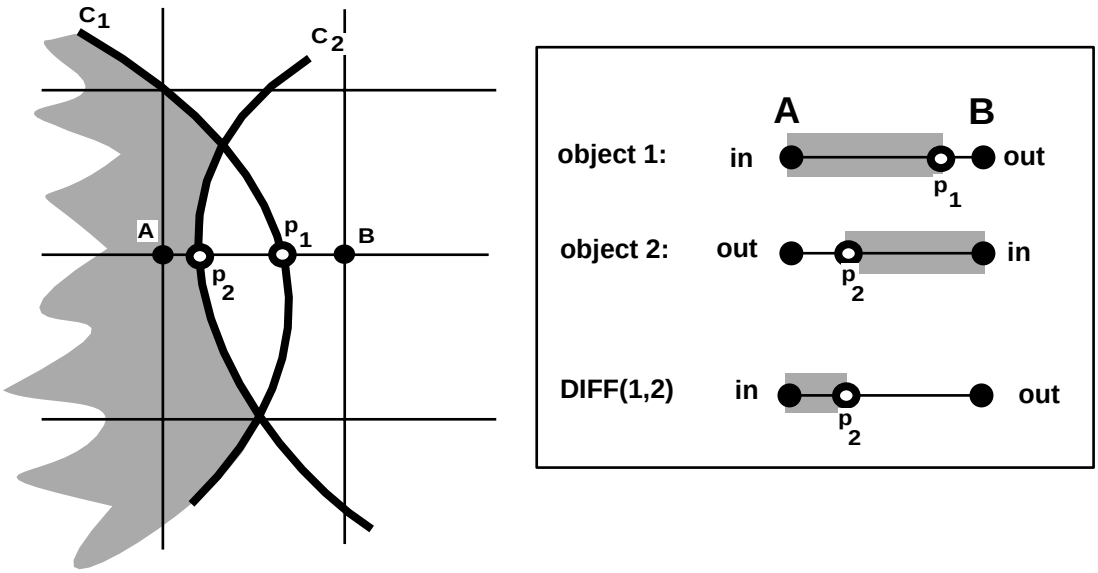

Figure 3: Using vector functions to represent several ISM objects at once.

$f_i(A)$	$f_j(A)$	$f_i(B)$	$f_j(B)$	$DIFF(i,j)$	$UNION(i,j)$	$INTSCT(i,j)$
out	out	out	out	out, -, out	out, -, out	out, -, out
out	out	out	in	out, -, out	out, p_j, in	out, -, out
out	out	in	out	out, p_i, in	out, p_i, in	out, -, out
out	out	in	in	out, -, out(*)	out, MIN, in	out, MAX, in
out	in	out	out	out, -, out	in, p_j, out	out,- ,out
out	in	out	in	out, -, out	in, -, in	out,- ,out
out	in	in	out	out, MAX, in	in, -, in(*)	out,- ,out(*)
out	in	in	in	out, -, out	in, -, in	out,p_i,in
in	out	out	out	in, p_i, out	in, p_i, out	out, -, out
in	out	out	in	in, MIN, out	in, -, in(*)	out, -, out(*)
in	out	in	out	in, -, in	in, -, in	out, -, out
in	out	in	in	in, p_j, out	in, -, in	out, p_j, in
in	in	out	out	out, -, out(*)	in, MAX, out	in, MIN, out
in	in	out	in	out, -, out	in, -, in	in, p_i, out
in	in	in	out	out, p_j, in	in, -, in	in, p_j, out
in	in	in	in	out, -, out	in, -, in	in, - , in

In the above table, the entries in the columns labeled $DIFF(i,j)$, $UNION(i,j)$, and $INTSCT(i,j)$ take the form 'cl_A, p, cl_B'. Here cl_A and cl_B are the in- or out-classifications of the resulting object in the vertices A and B, respectively. If they are equal, the intersection point is not defined, indicated by a '-' for the intersection point p. Otherwise, the intersection point is either the intersection point p_i (associated with ISM nr. i), the intersection point p_j (associated with ISM nr. j), or the minimum or maximum of these two. The minimum, $MIN = MIN(p_i, p_j)$, is the point closest to extreme A whereas $MAX = MAX(p_i, p_j)$ is the point closest to B. Some entries are labeled with an asterisk (*). For these entries, the application of the corresponding boolean operator on the coverage intervals

would really yield either zero or two intersection points. Since we have to comply with the convention that these intersection configurations go unnoticed, we have to report 'no intersection' in these cases.

Based on the table above and on a straightforward binary tree-representation of the boolean expression, an algorithm to compute the intersection point and in/out classification of the resulting surface, given the intersection points and in/out-classifications of the n ISM primitives, is readily obtained.

For a single voxel edge the algorithm in ANSI-C, it reads as follows:

```
typedef struct B_EXP {
  char kind;  /* the kind of this operator node u,i,d or n for union,
               * intersection, difference or primitive number */
  int n;      /* if kind=='n', the value of the primitive number */
  struct B_EXP *se1, *se2;
              /* if kind!='n', the first and second sub-expressions */
} B_EXP;

typedef struct {
  int A_in,B_in;/* two booleans that indicate if the two
                 * extremes are inside */
  float p;      /* a number between 0 and 1, defined if (A_in!=B_in)
                 * indicating the relative position of
                 * the surface intersection */
}SEGMENT;

SEGMENT combine(SEGMENT s[],B_EXP *e) {
SEGMENT rs,ss1,ss2; /* if the expression is an operator,
                     * ss1(2) are its operands */
  if(e->kind=='n') {
    rs=s[e->n];
    return rs;
  }
  ss1=combine(s,e->se1);
  ss2=combine(s,e->se2);
  switch(e->kind) {
    case 'u':rs=form_union(&ss1,&ss2);break;
    case 'i':rs=form_intersection(&ss1,&ss2);break;
    case 'd':rs=form_difference(&ss1,&ss2);break;
  }
  return rs;
}
```

The functions *form_union()*, *form_intersection()*, and *form_difference()* implement the instructions in the above table. Before a call to combine is made, the caller has to set up the array $s[]$ of segments, one segment for each of the ISM primitives. This means that for each of the primitives the intersection point with the current voxel edge has to be computed, as well as the in/out classification for that primitive in both extremes of the voxel edge. The segment that is returned by *combine()* contains the

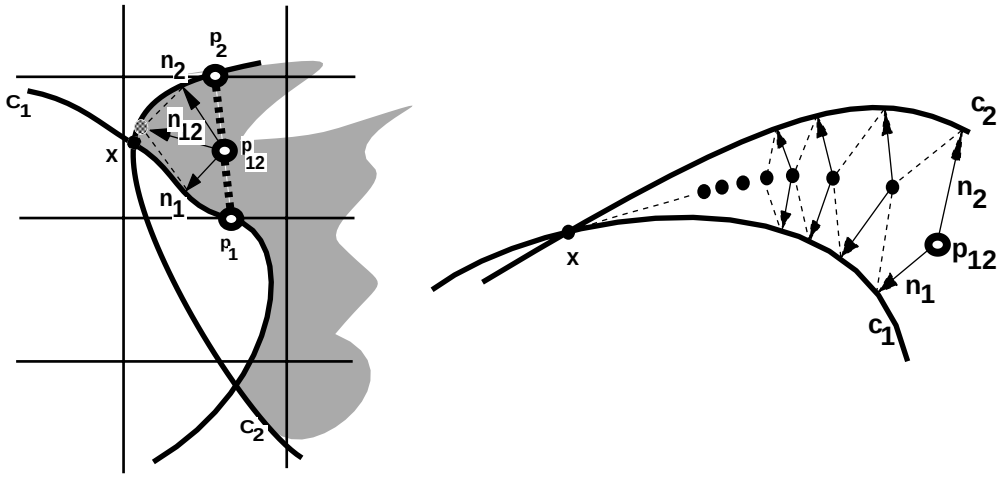

Figure 4: Approximating the intersection of two implicit contours. Left: $\mathbf{p}_{12}$ is a first guess. Right: iterating to get a more accurate approximation of $\mathbf{x}$.

intersection of the resulting boundary surface with the current voxel edge (if it exists), as well as the in/out classification in both extremes of this edge.

So an existing implementation of the uniform voxel space-subdivision algorithm can be easily extended by replacing the computation of the intersection by a loop that computes the intersections for all ISM surfaces, and next perform a call to *combine()* to have the intersection with the resulting surface computed.

For efficient evaluation of the boolean expressions the implementation of the $combine()-$ function is rather straightforward. We observe, however, that simple tests can be included to avoid evaluating sub expressions that do not contribute to the final result. For instance, computing $INTSCT(i,j)$ for two segments i and j where i is entirely outside the BCSO, yields again a segment that is entirely outside, irrespective of j. So if we haven't computed the sub expression yielding j yet, we may just as well leave it. Similar considerations hold for all other operations. These and other techniques have been well known in the field of ray tracing for CSG objects; see ([Jansen 87]) for an overview.

4 Arriving at non-smooth edges

Unlike implicit blends, CSG operations should result in sharp contours between primitives. Again we first study the problem in 2-D. Consider fig. 4. Here we have again the configuration that two iso-value contours, C_1 and C_2 intersect. The intersection point is $\mathbf{x} = (x, y, z)$, but this is of course a priori unknown and we should try to find an approximation to it. Suppose that the CSG-expression is $DIFF(2,1)$ where the interior region associated with curve C_1 is on the left of C_1 and the interior region associated with C_2 is on the right of C_2. Then the *combine()*-function from Section 2 results in the two relevant intersections between the resulting contour and the voxel edges $\mathbf{p}_1$ and $\mathbf{p}_2$, respectively. Following the uniform subdivision strategy for inferring the contour from intersection points, we would obtain the dashed line $\mathbf{p}_1\mathbf{p}_2$ as a segment of the contour. Since this is quite far from the actual intersection point, the non-C^1 junction is not very well reproduced. Instead we observe that we should find an estimate for $\mathbf{x}$ such that $f_1(\mathbf{x}) = f_2(\mathbf{x}) = 0$ where f_1 and f_2 are the scalar field functions for the two

ISM primitives with contours C_1 and C_2, respectively. Assuming we have a starting point which is not too far off, say $\mathbf{p}_{12} = \frac{\mathbf{p}_1+\mathbf{p}_2}{2}$, we can apply first order Taylor expansion to the difference $\mathbf{n}_{12} = \mathbf{x} - \mathbf{p}_{12}$.

As follows:

$$0 = f_1(\mathbf{x}) = f_1(\mathbf{p}_{12} + \mathbf{n}_{12}) \approx f_1(\mathbf{p}_{12}) + (\mathbf{n}_{12}, \nabla f_1(\mathbf{p}_{12})) \tag{1}$$

$$0 = f_2(\mathbf{x}) = f_2(\mathbf{p}_{12} + \mathbf{n}_{12}) \approx f_2(\mathbf{p}_{12}) + (\mathbf{n}_{12}, \nabla f_2(\mathbf{p}_{12})) \tag{2}$$

As we can evaluate $f_1 = f_1(\mathbf{p}_{12})$ and $f_2 = f_2(\mathbf{p}_{12})$, and similarly ∇f_1 and ∇f_2 in the same point $\mathbf{p}_{12}$, we can solve the two linear equations 1 and 2 in $\mathbf{n}_{12}$ e.g. in least squares sense. However, since we have to iterate anyway in order to find an accurate estimate for $\mathbf{x}$, due to the f being nonlinear, we proceed differently. Using 1 we can try to get an estimate $\mathbf{n}_1$ for $\mathbf{n}_{12}$ by setting $\mathbf{n}_1 = \lambda_1 \nabla f_1$, for an unknown scalar λ_1. For λ_1 we then find

$$\lambda_1 = -\frac{f_1}{(\nabla f_1, \nabla f_1)}, \tag{3}$$

so

$$\mathbf{n}_1 = -\frac{f_1 \nabla f_1}{(\nabla f_1, \nabla f_1)}, \tag{4}$$

and similarly

$$\mathbf{n}_2 = -\frac{f_2 \nabla f_2}{(\nabla f_2, \nabla f_2)}. \tag{5}$$

The two vectors $\mathbf{n}_1$ and $\mathbf{n}_2$ thus obtained are depicted in fig. 4. Since $\mathbf{n}_1$ would move $\mathbf{p}_{12} + \mathbf{n}_1$ closer to the contour C_1, and $\mathbf{n}_2$ would move $\mathbf{p}_{12} + \mathbf{n}_2$ closer to contour C_2, their sum $\mathbf{n}_{12} = \mathbf{n}_1 + \mathbf{n}_2$ is expected to move $\mathbf{p}_{12} + \mathbf{n}_{12}$ closer to the intersection of the two. So we set

$$\mathbf{p}'_{12} = \mathbf{p}_{12} + \mathbf{n}_1 + \mathbf{n}_2 \tag{6}$$

and re-iterate the previous computations until the length of the resulting displacement vector $\mathbf{n}_{12}$ drops below a given threshold. In practice, 10 iterations are sufficient for a visually smooth approximation of the contour. The right hand part of fig. 4 shows the subsequent estimates for the intersection point during a series of iterations in the case of a 'hard' configuration. We make two observations:

- *Converge* is only linear. Faster schemes could be employed, but the number of edges that have to be split (i.e. the number of edges that are in the vicinity of a non-C^1 junction) is expected to be significantly less than the total number of edges. So the additional effort would not pay off.

- When the surfaces of the two ISM primitives are intersecting under a sharp angle, convergence may need considerably more steps. In pathological cases it can even happen, if one of the functions f_j varies wildly in the vicinity of the intersection, that the 1st order Taylor approximation that underlies our intersection-finder is insufficient, in which case the algorithm is numerically unstable. This is reminiscent, however, of the numerical difficulties in finding intersections between nearly tangent curved surfaces in more general contexts. However this does not cause our algorithm to fail. A surface will still be found and a consistent mesh produced, although it will show an inexact surface representation.

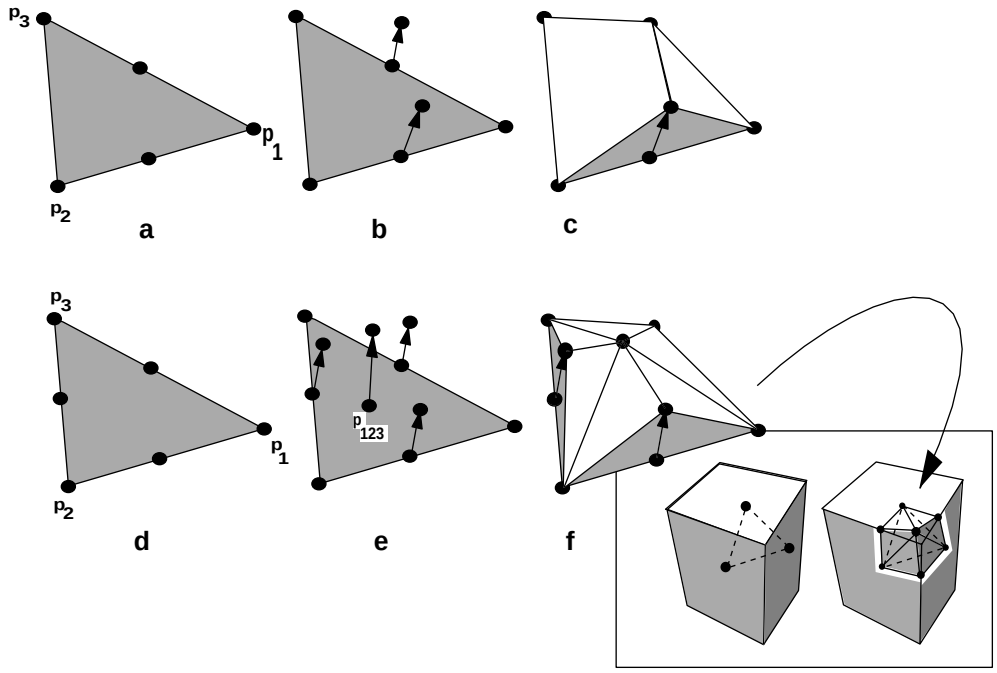

Figure 5: Splitting schemes for 3-D triangles

A more sophisticated numerical technique may find a better approximation to the non-C^1 junction, however we should not lose sight of our goal here, which is to make a reasonable prototype mesh. Ray tracing using a guaranteed method (e.g. see [Kalra 89]) would produce a better result for a single image, although as can be seen for the examples we present the artifacts are generally small. Our intent is to produce a mesh which can then be arbitrarily transformed as polygons on a graphics workstation for rendering in real time as a prototype.

Notice that this computation only needs to be done if the intersections $\mathbf{p}_1$ and $\mathbf{p}_2$ belong to the surfaces of different ISM's.

In three dimensions, we are confronted with triangles rather than line segments that may have to be split. Here, two situations have to be distinguished.

Consider fig. 5. In the top row, we see the case where in the triangle $\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ point $\mathbf{p}_1$ belongs to the surface of one ISM primitive whereas $\mathbf{p}_2$ and $\mathbf{p}_3$ together belong to another ISM primitive surface. So edges $\mathbf{p}_1\mathbf{p}_2$ and $\mathbf{p}_1\mathbf{p}_3$ are split and the split points are computed according to the above algorithm (fig. 5b). The new configuration consists of the two polygons in the diagram (fig. 5c); for a fully triangulated boundary mesh, the quadrangle should be triangulated as well.

In the bottom row, all three vertices $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ belong to different ISM surfaces. A configuration like this occurs e.g. near a corner vertex of a cube (see inset in fig. 5). In this case the triangle is to be replaced by 6 triangles, which means that we not only have to find the intersections associated with the mid points of edges, but also one additional new point that comes in the place of the mid point $\mathbf{p}_{123}$ of the triangle. This latter intersection is a three-fold intersection, i.e. a point where three ISM primitive surfaces meet. We find the latter point using exactly the same approach as we did for intersections with

two ISM primitive surfaces. Only our starting estimate will be $\mathbf{p}_{123}$ in stead of $\mathbf{p}_{12}$. So we compute

$$\mathbf{n}_1 = -\frac{f_1 \nabla f_1}{(\nabla f_1, \nabla f_1)}, \tag{7}$$

$$\mathbf{n}_2 = -\frac{f_2 \nabla f_2}{(\nabla f_2, \nabla f_2)}. \tag{8}$$

$$\mathbf{n}_3 = -\frac{f_3 \nabla f_3}{(\nabla f_3, \nabla f_3)}. \tag{9}$$

where the f and ∇f are defined in the point $\mathbf{p}_{123}$, and next we set

$$\mathbf{p}'_{123} = \mathbf{p}_{123} + \mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3. \tag{10}$$

Figure 6 shows a wheel model before and after the application of the edge enhancement described above. This image clearly shows the improvement to the edges of the model with the additional polygons.

5 Implementation aspects.

Finding the surface: A simple heuristic to obtain a starting voxel for the uniform subdivision algorithm in the case of one single ISM object is to probe the function f in the centre of one of the skeletal elements where it is certain that $f > 0$ and next move onwards in an arbitrary direction one voxel at a time until a voxel has been found that has vertices both inside and outside the ISM; this voxel is an appropriate starting position for the polygonizer. When polygonizing the surface of a combined CSG/ISM object, however, this strategy does not necessarily work. Consider the intersection of two spheres with slightly different centres; both centre points will be outside the resulting model, and if we happen to proceed in a direction away from the model, we never find the surface. A partial fix to this problem is to probe in several directions, in order to enlarge the chance to find the surface. It appears however that a 100% reliable starting strategy should probe in an infinite amount of directions since there seems to be no way for computing beforehand where the surface is. In practice, this turns out to be not much of a problem since there usually will be at least one ISM skeletal element with its centre inside the resulting surface. So probing in 2 or 3 sufficiently different directions, starting from all the skeletal elements, is a practical way to initiate the surface polygonization. Initiating the search from all of the skeletal elements is also necessary to find the surface in case it consists of several disjoint segments as described in [Wyvill 86].

Normal vectors: for illumination computations, normal vectors should be provided that are consistently oriented. For a single ISM primitive j, normal vectors always point in the direction of ∇f_j. In the case ISM j is subtracted from the current combined model, the normal vector on this part of the surface should point in the direction of $-\nabla f_j$. The $combine()$-function from Section 2 therefore could be extended in such a way that it also computes information on the orientation of the normal vector of the resulting surface as a result of the two normal vectors in the two input surfaces. This is of course easily done on the basis of the values of A_in and B_in. Notice however, that computing normal vectors is relatively expensive, and it is really only necessary to have the normal vector for the single ISM primitive that is responsible for the resulting intersection. Therefore the $combine()-$ function should rather return an identification of that ISM, together with a boolean to

indicate if the normal vector should be reversed so that the normal vector calculation can take place afterwards. We also observe that in the intersections that are computed as in Section 3, two or three normal vectors should be output, namely one for each of the intersecting ISM's, in order to obtain also visually non-C^1 junctions. Finally observe that it is advantageous to have the $combine()-$function compute the identification of the resulting ISM, so that ISM's with different surface attributes (e.g. colours) give rise to appropriately coloured parts in the surface.

6 Results; discussion; conclusion

Results

Figure 6 shows a wheel built from 14 BCSO primitives. These are combined to form seven ISM's. For example, the seven spokes are blended together with the outer torus to form a single ISM. Intersecting half planes are used to flatten the front and back and a difference operation was used to subtract a torus from the flattened surface to form the decorative groove. The colours have been chosen to illustrate the separate ISM's.

Figure 7 shows two primitive cylinders (magenta) are positioned in a cross formation. They blend with each other and the magenta spheres. The white spheres and the yellow cylinder have been added using a union operation. Two green cylinders have been blended together to form a smoothly blended T-shape ISM, that has been subtracted from inside the magenta cylinders. The cut-away is formed by taking the intersection of two half planes and forming the union with the remainder of the object after translating the cut-away portion.

The coffee grinder of figure 6 was constructed from 15 ISM's, composed of some 25 primitives. Again colours are used to distinguish between different ISM's.

The table and mirror frame were also modelled using BCSO's. Our polygonizer took about 1 minute on an SGI Indy to convert the primitive description and boolean expression to about 1.5 million triangles. There are 37 BCSO primitives in the coffee grinder (14 blending groups), 17 primitives in the table (10 blending groups) and 19 primitives in the mirror (16 blending groups). The polygons were ray traced using a locally modified version of Rayshade [Kolb 95].

The model of the American Type 4-4-0 (c. 1855) uses 237 CSG primitives and takes about 5 minutes to produce about 450,000 polygons on the SGI Indy.

It should be noted that the times given are for the manufacture of a relatively high density mesh. Lower quality meshes can be produced significantly faster. Once the polygons have been calculated different views can be chosen in real time.

7 Discussion

Earlier work has used ray tracing to compute BCSO objects. The main contribution of this research is introduce a scheme for polygonization of these models. It is therefore helpful to compare the two methods. Both ray tracing and voxel-based polygonization are methods to convert an ISM into a (explicit) coordinate representation:

- ray tracing by sampling the surface with a large set of light rays and using the collection of intersections to directly assign colours to pixels. To get an accurate rendering, the number of rays has to be at least as high as the number of pixels, but consequently sampling artifacts are not larger than pixels.

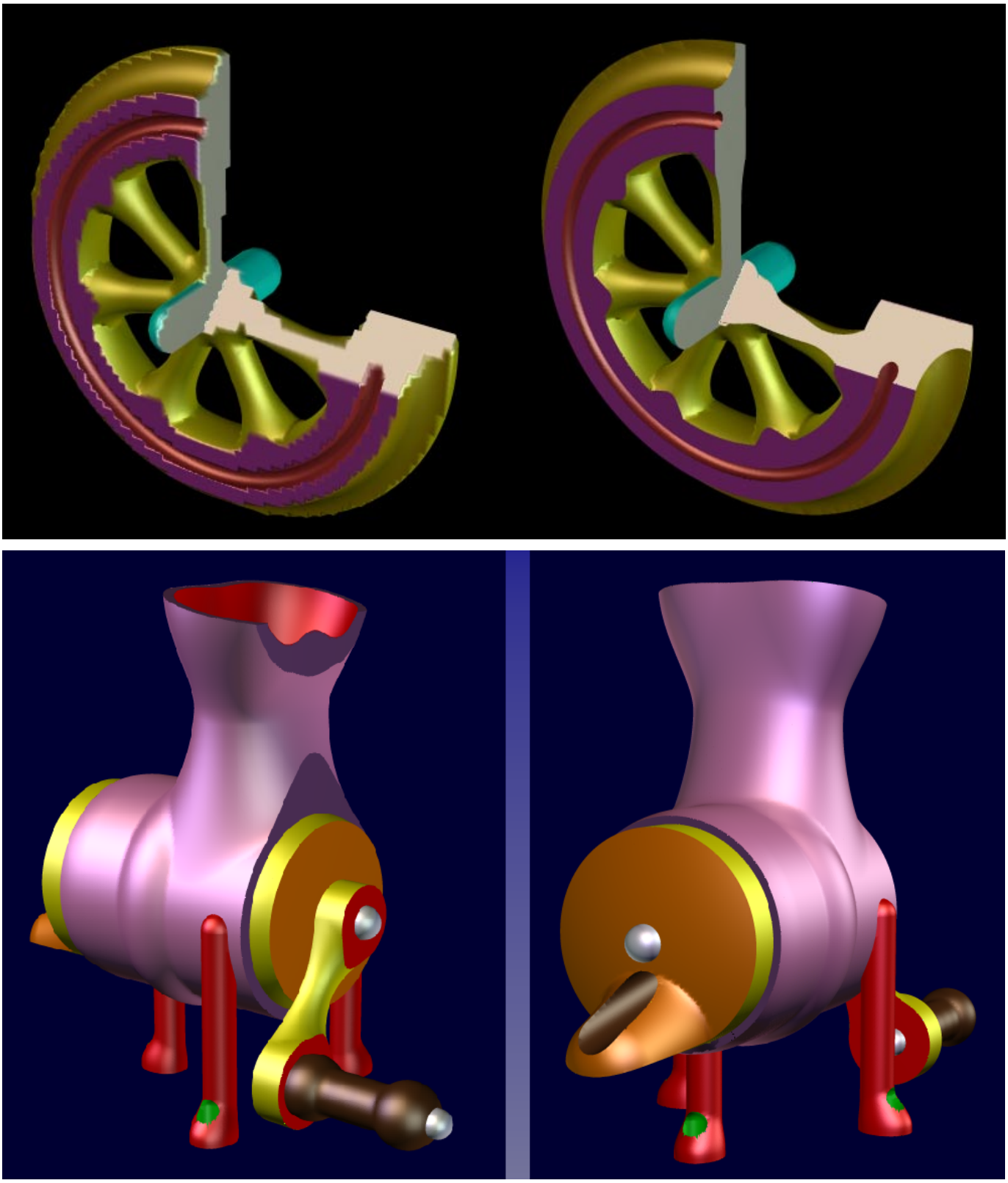

Figure 6: upper: Wheel before and after postprocessing. lower: The Canmore coffee grinder.

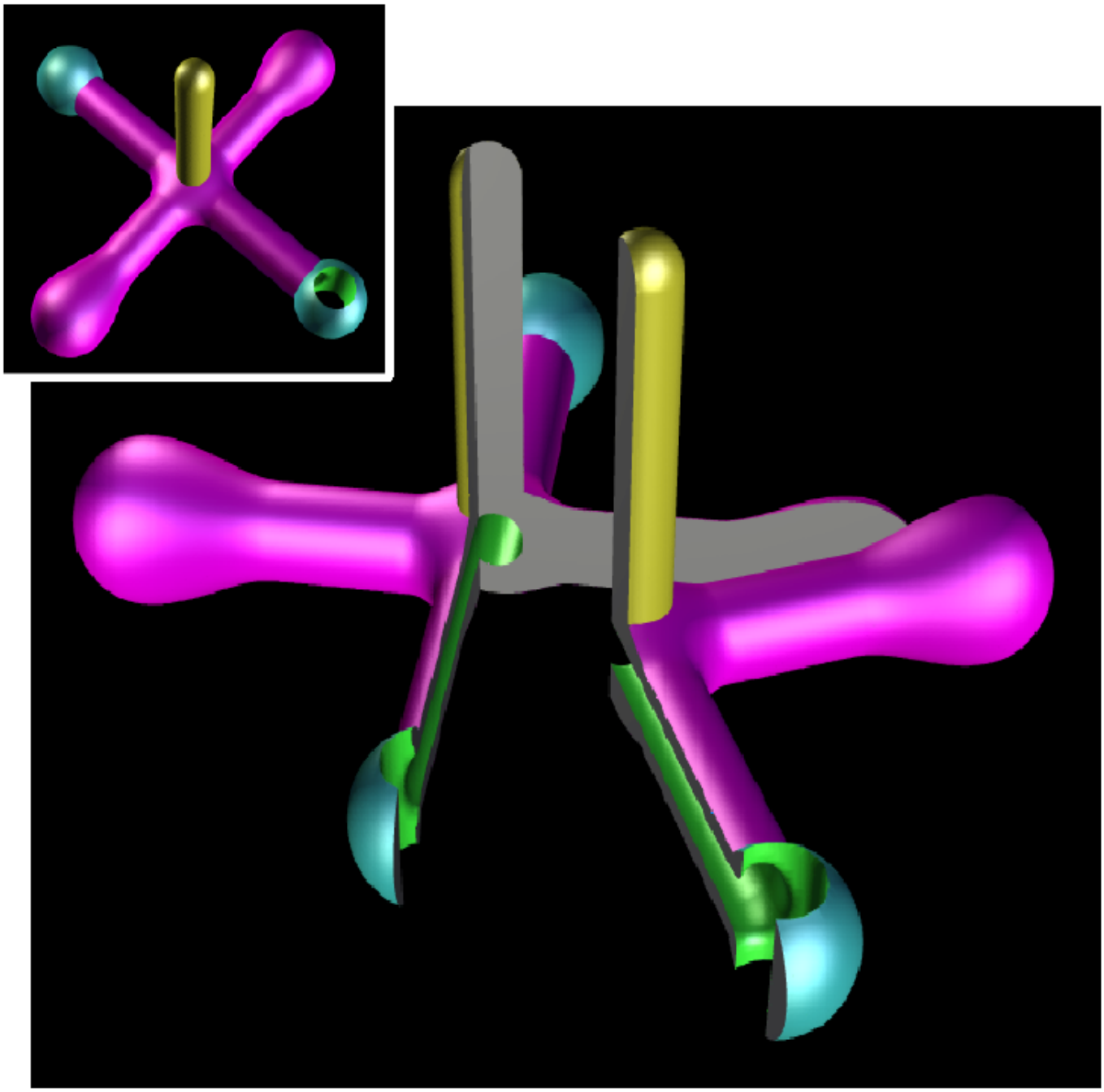

Figure 7: Combining blended primitives with CSG operations.

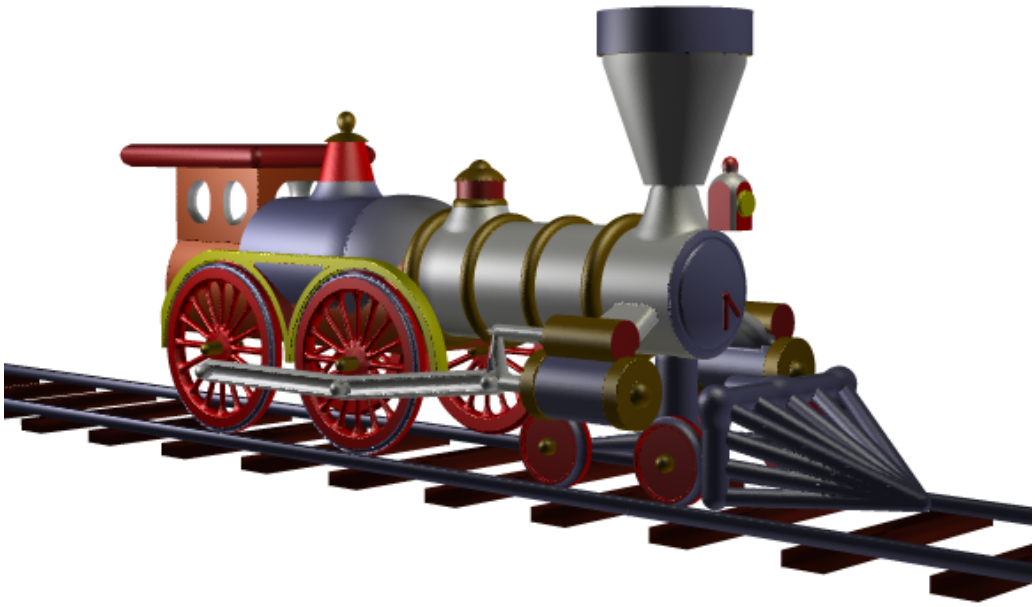

Figure 8: upper: The Canmore coffee grinder and friends. Lower: American Type 4-4-0 (c. 1855)

- voxel-based polygonization by sampling the surface on a regular grid of voxel edges, and using the collection of surface-edge intersections to build a polygon mesh. To get an adequate representation, the voxel grid only has to be dense enough to capture all geometric features of the implicit surface, but sampling artifacts may be as large as individual voxels. With respect to sampling the ISM, voxel edges essentially play the roles of the rays in ray tracing.

Due to the fundamental similarity of the two techniques, it should not come as a surprise that both can be extended to cope with CSG-expressions in ISM's. This has been well known for ray tracing where the CSG-expressions are evaluated on a per-ray base; in this paper we suggest to follow a similar technique for voxel-based polygonization where the CSG-expressions are evaluated on a per-voxel-edge base.

8 Conclusion

We propose an extension to voxel-based polygonization of ISM's which accounts for CSG-type expressions in ISM's. It converts a BCSO into a polygon mesh which displays both smooth blends, characteristic in implicit surface modelling, and non-C^1 junctions characteristic for traditional CSG.

9 Acknowledgements

The authors would like to thank the many students who have contributed towards the ongoing work on implicit surfaces. We would also like to thank Dr. Przemek Prusinkiewicz, Dr. Jules Bloomenthal and Dr. Geoff Wyvill of the University of Otago for their encouragement past and present.

This work is partially supported by the Natural Sciences and Engineering Council of Canada.

10 Glossary

BCSO: Boolean Compound Soft Object: a CSG expression with a set of ISM's as operands.

CSG: a method for combining shapes (primitives) via the boolean operations $DIFF$ (set difference), $UNION$ (set union), and $INTSCT$ (set intersection).

CSG expression: an expression in CSG operations with primitives as operands.

CSG operation: set difference, set union, or set intersection.

ISM: Implicit Surface Model.

ISM surface: the boundary of an ISM.

ISM primitive: an ISM as it occurs in boolean expressions to arrive at a BCSO.

junction: a curve on the surface of a CSG object which is the intersection curve of two of the consituent primitives.

skeletal element: a geometric primitive (e.g. point, line, torus, etc.) that parameterises a scalar function f such that the iso-surface $f = 0$ forms the ISM surface of the ISM that is defined via one or more of such elements.

skeleton: a collection of skeletal elements that form an ISM.

polygonization: the approximation of an implicit surface by a polygon mesh (usually a mesh consisting of triangles).

primitives: operands in a CSG expression.

segment: an edge of a voxel that intersects the surface of a BCSO.

voxel: a cubic volume in 3-D space which is oriented along the coordinate axes.

References

[Bloomentha 88] Jules Bloomenthal. Polygonisation of Implicit Surfaces. *Computer Aided Geometric Design*, 4(5):341–355, 1988.

[Bloomentha 90] Jules Bloomenthal and Brian Wyvill. Interactive Techniques for Implicit Modeling. *Computer Graphics*, 24(2):109–116, 1990.

[Chiokura 83] H. Chiokura and F. Kimura. Design of Solids with Free Form Surfaces. *Computer Graphics (Proc. SIGGRAPH 83)*, 17(3):289–296, July 1983.

[Crocker 87] G.A. Crocker and W.F. Rainke. Boundary evaluation of non-convex primitives to produce parametric trimmed surfaces. *Computer Graphics (Proc. SIGGRAPH 87)*, 21(4):129–136, July 1987.

[Gascuel 93] Marie-Paule Gascuel. An Implicit Formulation for Precise Contact Modeling Between Flexible Solids. *Computer Graphics (Proc. SIGGRAPH 93)*, pages 313–320, August 1993.

[Jansen 87] F.W Jansen. *Solid Modeling with Faceted Primitives*. PhD thesis, Delft University of Technology, Netherlands, September 1987.

[Kalra 89] D. Kalra and A. Barr. Guaranteed Ray Intersections with Implicit Functions. *Computer Graphics (Proc. SIGGRAPH 89)*, 23(3):297–306, July 1989.

[Kolb 95] Craig Kolb. Rayshade - a public domain ray tracer. *The Rayshade Manual*, 1995.

[Loop 94] Charles Loop. Smooth spline surfaces over irregular meshes. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 303–310. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[Mantyla 88] Martti Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland 20850, 1988.

[Middleditc 85] A. Middleditch and K. Sears. Blend Surfaces for Set Theoretic Volume Modelling Systems. *Computer Graphics (Proc. SIGGRAPH 85)*, 19(3):161–170, 1985.

[Naylor 90] B. Naylor, J. Amantides, and J. Thibault. Merging BSP trees yields polyhedral set operations. *Computer Graphics (Proc. SIGGRAPH 90)*, 224(4):115–124, August 1990.

[Pasko 95] Alexander Pasko and Vladimir Savchenko. Constructing functionally-defined surfaces. In *Implicit Surfaces '95*, pages 97–106, April 1995.

[Requicha 80] A.A.G. Requicha. Representations for Rigid Solids: Theory, Methods, and Systems. *ACM computing surveys*, 12(4):437–464, December 1980.

[Schroeder 92] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of Triangle Meshes. *Computer Graphics (Proc. SIGGRAPH 92)*, 26(2):65–70, July 1992.

[Wyvill 86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2(4):227–234, February 1986.

[Wyvill 90] G. Wyvill and A. Trotman. Ray tracing soft objects. *Proc. CG International 90*, 1990.