Correct Shading of Regularized CSG Solids Using a Depth-Interval Buffer

Jaroslaw R. Rossignac and Jeffrey Wu

ABSTRACT A convenient interactive design environment requires efficient facilities for shading solid models represented in CSG. Shading techniques based on boundary evaluation or ray casting that require calculations of geometric intersections are too inefficient for interactive graphics when CSG primitives with curved (parametric) surfaces are involved. Projective approaches, where the primitive surfaces are scan-converted using standard hardware-supported graphic functions are preferred. Since not all the points of the faces of a CSG primitive lie on the CSG solid, scan conversion must be combined with a procedure that tests the produced 3D surface-points against the original CSG expression. Point classifications against primitives defined by arbitrary curved boundaries may be performed, without geometric intersections, through depth-comparisons at each pixel. This approach has been implemented for the Pixel-Power machine by researchers at UNC. It deals with complex CSG trees by converting CSG expressions into sum-of-product form and repeatedly scan-converting the primitives of each product. The Trickle algorithm, which considerably reduces the number of scan-conversions in the general case has been developed at IBM Research and presented elsewhere. This paper discusses several recent improvements to the original Trickle algorithm. The overall algorithm has been simplified. The scan-conversion process and the point classification tests have been modified to correctly handle cases where several primitive faces coincide within an arbitrary numerical resolution. These enhancements are not only necessary for on/on cases in regularized Boolean expressions, but also for processing pairs of faces near their common edges. Finally, we point out that a simple two-pass extension of the trickle algorithm using an auxiliary shadow buffer suffices to compute directly from CSG shaded images with shadows.

1 Introduction

Mechanical parts commonly designed in CAD systems are seldom polyhedral and only in rare cases can be expressed as extrusions of 2D contours. Three-dimensional design techniques are thus necessary. The most popular technique for interactively designing models of 3D mechanical parts is Constructive Solid Geometry (abbreviated CSG), where designers construct solid models by combining sub-solids or parameterized primitive volumes through regularized Boolean expressions [1]. Such CSG specification is typically parsed and stored in a CSG tree, or more precisely a binary directed rooted acyclic graph. The internal nodes of the graph correspond to regularized set theoretic Boolean operators (union, intersection, or difference) and define sub-solids. The root defines the entire solid—typically a semi-algebraic three-dimensional r-set, which could be empty or composed of disconnected volumes. Terminal leaves of the graph represent parametric primitive shapes, which in traditional CSG-based systems were restricted to be intersections of planar or simple quadric half-spaces, such as cylinders or spheres. The graph is not Necessarily a tree since the same sub-solid may be used several times in the final Boolean expression. Although the graph can always be expanded into a tree, a systematic expansion should be avoided, especially when dealing with CSG definitions containing many instances of complex CSG sub-solids.

Often CSG graphs also contain linear transformations (often restricted to rigid body motions) that define the relative position of children nodes with respect to the parent node's local coordinate systems. The effect of these transformations are usually combined during tree traversal and propagated all the way down to the primitives to establish their final positions. The correct processing of these transformations is trivially combined with the approach described here and will not be discussed any further. We shall simply assume that the final position of each primitive instance is known whenever necessary.

CSG graphs can be conveniently edited by simply changing the Boolean expression or the primitives' type, position, orientation, or size parameters. Other representations, such as a boundary graph, are much more difficult to edit without endangering their validity. CSG is thus the preferred medium for performing an incremental (trial-and-error) design process. Interactive editing requires interactive feedback to guide the designers towards the desired solution. Interactive graphic from CSG is thus an essential component of the design process and shaded images have become the de facto standard for visualizing solid models. Several techniques for shading CSG solids have been reported.

1.1 Boundary Tesselation

CSG graphs may be converted into a boundary representation by boundary evaluation procedures [2] which computes edges and vertices by intersecting surfaces. Then, the bounding faces may, for example, be tesselated and rendered as a triangular mesh. The boundary evaluation is usually very time consuming, especially if higher degree implicit or parametric surfaces are involved. Tesselation is also delicate, because one must ensure that no cracks or overshooting occurs near the intersection edges.

1.2 Primitives' Tesselation

To avoid dealing with complex and expensive surface intersection and boundary tesselation procedures, many commercial solid modellers tesselate the primitives prior to the boundary evaluation. Good accuracy requires a large number of facets, and thus boosts up the cost of boundary evaluation, without even guaranteeing topological consistency. Furthermore, independent tesselations of coincident curved faces of different primitives may not be aligned properly creating models that topologically do not correspond to the designer's intent and that may even be invalid models for solids, due to numerical problems.

1.3 Ray-casting on CSG

Ray casting can be used directly on CSG, thus bypassing the expensive and delicate boundary evaluation, because the original 3D CSG expressions may be localized to each single ray, in which case it combines one-dimensional intervals obtained as intersections of the ray with the primitive solids [3]. Using these 1D models, the first point (along the viewing direction) on the ray that lies on the actual intersection of the ray with the solid can be easily computed. Efficient direct approaches for shading from CSG through ray-casting have been developed in software. They have been optimized for facetted models [4,5,6]. A hardware implementation exists for Boolean combinations of quadric half-spaces [7,8]. However, ray-casting involves computing a large number of ray/surface intersections and becomes particularly inefficient when higher degree algebraic or parametric surfaces are involved.

1.4 Projective Methods with Software Classification

Since, for parametric formulations, surface evaluation is faster than ray/surface intersection calculation, scan conversion techniques with (adaptive) tesselation of primitive surfaces have been used for shading boundary models. A hardware depth-buffer can be used for automatically selecting the visible faces. Because the faces of a CSG solid are not directly available, the depth-buffer visibility test must be combined with a trimming process that selects the portions of primitive faces that lie on the solid.

A software implementation of this selection has been combined with a depth-buffer test in [9] and works as follows. A point P on a front-facing face of a primitive A is first compared to the depth stored in the z-buffer of the corresponding pixel. If P is in front of what is stored in the z-buffer, it is 'classified', i.e., tested, against the CSG graph. Points on the boundary of the solid are rendered into the z-buffer. Outside points out are discarded. A point inside the solid will be automatically rejected by the z-buffer visibility test. Therefore, one can improve performance and avoid testing P against certain primitives in the graph, by classifying P against the I-zone of A, which is the intersection of a subset of the nodes of the original CSG graph [10]. If P lies inside the I-zone of A, it lies on the final solid or inside it. The software selection described in [9] classifies points against solid primitives by evaluating, at the tested points, the implicit functions that define the half-spaces bounding the primitive volumes. (Typically, a primitive is the intersection of such half-spaces. For example, a truncated cylinder is the intersection of one quadric half-space with two planar half-spaces.) These classification results are then combined up the tree according to the Boolean expression of the I-zone of the particular primitive on which the classified point lies.

1.5 Projective Methods with Z-buffer Classification

When sculptured primitives are used in the CSG expression (for example, primitives defined in terms of their boundary composed of trimmed NURBS surfaces), no set of implicit equations are available for classifying points against the primitive. As mentioned earlier, software implementation of such classification (for example through ray-casting [11]) would be far too expensive for graphics. A convenient alternative is to use depth comparisons and primitive boundary scan-conversion to classify points.

During scan-conversion, surface points are generated, which project onto individual pixels along the viewing direction. The depth of the 3D points (computed along the viewing direction away from the viewer) may be stored in the z-buffer memory associated with the corresponding pixel. A 3D point whose depth is stored in some pixel's z-buffer may be classified against a primitive by scan-converting the boundary of that primitive and computing the parity of the number of layers of the primitive's surface that are behind the point being tested. (One needs only to compare depth values of surface points projecting on the same pixel as the tested point with the value stored in the z-buffer. Each time the scanned point depth is larger than the stored one, a binary flag associated with that pixel is inverted.) Note that this process may be used to classify in parallel a large number of points, as long as they project on different pixels. This technique is described in [12] and mathematically justified in [13].

To classify a point against a CSG expression, it is not sufficient to classify the point against all the primitives. Point-primitive classification results must be combined according to a Boolean expression. For some simple Boolean expression, such as an intersection, no storage is necessary because the result can be formulated as the conjunction of Boolean results. The classification algorithm may process the primitives in any order and stop as soon as one of these results is FALSE. (This would be the case when, for example, the point was out of a primitive in a Boolean intersection.) If all the primitives are processed an no FALSE result is found, the point is inside the solid defined by the Boolean expression.

Unfortunately, the evaluation of more complicated CSG expression may require a large amount of temporary storage for intermediate binary results. Usually a stack mechanism is used for the temporary storage. The required stack depth may reach the depth of the CSG graph.

Since the amount of memory per pixel is limited, one cannot use a stack of arbitrary depth at each pixel. Yet, we want to perform classification operations in parallel for all pixels, so as to minimize the number of required primitive scan-conversions.

A technique that circumvents this memory limitation converts the CSG expression into a much larger (sum-of-product) form [14] in which primitive instances can be duplicated many times, appearing in several products. Techniques for eliminating redundant (empty) products have been discussed in [14].

Primitive faces are first trimmed against the appropriate products using repeated primitive scan-conversions. The resulting trimmed faces are then merged using a final depth buffer for selecting the front-most faces among all the products. Note that products can interfere and thus a front face of a product needs not lie on the solid. The z-buffer is used, as in [9], for both visible surface selection and for discarding faces interior to the solid.

This paper pertains to the implementation of this projective approach. It focuses on correct algorithms for computing the visible front-faces of a product, given that depth-buffer comparisons are performed with limited resolution and that one needs to correctly handle situations where faces of several primitives overlap or where the ray of a pixel intersect two adjacent faces very close to their common edge. In both cases, due to scan-conversion round-off errors, we cannot rely on the computed depth values, but must still produce a picture that corresponds to a regularized version of the CSG expression. A solid is regularized when it is equal to the topological closure of its interior with respect to the three-dimensional Euclidean space. Regularized solids have no dangling edges or faces. Thus, faces or edges that lie on several primitives, but are not bounding any three-dimensional volume in the final result, should not be displayed.

The next section presents a new algorithm for shading CSG solids and briefly discusses its historic evolution. Then we point out the accuracy and regularization problems and explain how we solve them. We also point out a simple extension of our algorithm that produces shaded images with shadows directly from CSG, without computing any silhouette edges that are usually required to define the limits of shadow volumes. In the final section, we demonstrate in detail the progress of the trickle algorithm on simple products with coincident faces. The appendix shows the content of the three buffers as the trickle algorithm progresses through the a real CSG object with many coincident faces and a non-convex primitive.

2 The Trickle Algorithm

Researchers at UNC [14] have implemented a hardware algorithm for trimming primitive faces by comparing them to all the front and back faces of all the primitives in a product. A variation of this approach was also reported in [4]. The comparisons are done independently at each pixel and involve depth tests, masks, counters, and logical bit operations at each pixel. The algorithm has been efficiently implemented on the Pixel-Power graphic system that has one processor with local memory at each pixel [14].

Researchers at IBM [15] have developed a more efficient algorithm for processing products, called the Trickle algorithm. It requires, in general, a much smaller number of primitive scan-conversions and of buffer-merging operations than the UNC algorithm and may be better suited for implementation on emerging commercial graphic workstations, because it only requires simple extensions of existing programmable depth-buffer functions. Both the UNC and the IBM algorithms handle non-convex primitives by producing and trimming the successive layers of a primitive's faces.

The strength of the trickle algorithm lies in the fact that it processes the primitive faces of a product in a front-to-back (away from the viewer) order independently at each pixel. This ordering permits to stop the processing of a product as soon as a visible point (or the background) has been reached at each pixel. Furthermore, while moving 'deeper' (away from the viewer) from one primitive-face layer to another at a given pixel, the trickle algorithm skips primitive-face layers that clearly cannot lie on the product because they are out of at least one primitive in the product.

A drawback of the trickle algorithm is that it uses three depth and intensity buffers (the standard depth and intensity buffers used for visible surface selection, plus a new depth-interval buffer, abbreviated DIB, composed to two depth buffers and two intensity buffers). Note that the UNC algorithm only requires two additional buffers and a counter. However, the trickle algorithm may be configured to run in four passes, once for each quadrant of the screen. Splitting the screen into four quadrants provides enough buffers for the trickle algorithm, even with the standard configuration of one depth and two intensity buffers commonly available on graphic workstations. Unfortunately, this four-pass approach requires that intensity buffers be sometimes used as depth-buffers, which is currently impossible with commonly available commercial graphic systems.

In this paper, we report a new version of the Trickle algorithm, which was originally published in [15]. A very high-level view of the trickle algorithm follows:

```
initialize buffer 1
for each product P do:
    compute the front of P in buffer 2
    merge the result into buffer 1 selecting the visible faces
```

Buffer 1 (depth and intensity) is used to select the visible surfaces of a union of products. The visible front faces of each product are computed one after the other using a DIB as summarized below. The result is stored in buffer 2.

```
initialize buffer 2 to the background
while not done, circulate through the primitives Q of P and do:
    compute into buffer 3 the next face of Q that lies behind buffer 2
    At pixels where the next face of Q is front-facing copy buffer 3 into buffer 2
```

The details and advantages of the Trickle algorithms are described in details in the companion paper [15]. We can only briefly mention that the trickle algorithm works because, when a point in buffer 2 immediately precedes in depth a front face of a primitive Q, it lies out of Q, and thus out of the product. Furthermore, the interval between the tested point and the corresponding front point on Q is also out of Q and thus out of the product (see Figure 1).

Z1, the point of the boundary of Q that lies immediately behind the point Z2 stored in buffer 2 is computed using buffer 3. If Z2 lies on a front facing portion of the boundary of Q, the segment separating Z1 and Z2 lies out of Q and the trickle algorithm advances buffer 2 to Z3 bypassing any point r that may lie between Z2 and Z3.

We propose below a pseudo-code presentation of a new version of the trickle algorithm. We have slightly simplified the algorithm of [15] by integrating the initialization steps into the main loop. The main improvement is in the carefully enhanced depth-tests that provide correct treatment of all singular (so called 'on/on') cases that involve coincident primitive faces and pixels in the vicinity of the projection of silhouette edges.

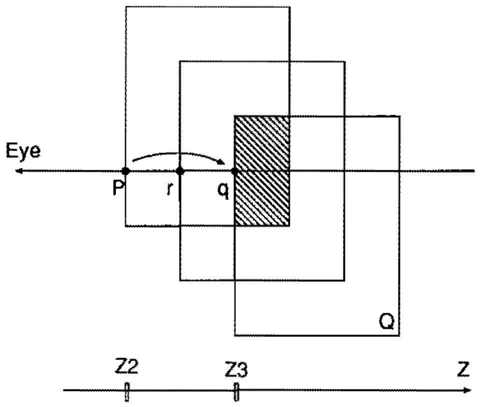


Fig. 1. Move forward: Point p lies on a primitive in the current product and its depth is stored in Z2. The primitive Q is scan-converted and the depth of point q on Q is stored into Z3 because point q is the first point on Q (in terms of depth) to be hidden by p. Point q is a front point for Q. Therefore, the segment (p,q) is out of Q and thus out of the product. The algorithm moves forward by storing the depth and intensity of p into Z2 and I2. Note that point r, on a different primitive in the product is skipped because it lies between p and q and thus is not on the product.

After an initialization of buffer 1, i.e., depth buffer Z1 and intensity buffer I1 are set to maximum depth and background color, (lines 1-3), the algorithm processes each product. For each product, buffer 2 is initialized to minimum depth and background color (lines 5-7). Then, the visible front faces of the product are computed and stored in the depth buffer Z2 and intensity buffer I2 (lines 9-31). Finally, the result is merged into buffer 1 (lines 33-35) wherever the current product's front is in front of previously processed products.

To compute the visible front faces of a product, we proceed as follow. A counter k is first initialized to -1 (line 9). It will be used to count how many primitives of the current product have been processed without affecting buffer 2. If there is only one primitive in the product, we need to scan it twice to properly produce the effect of regularization. If all primitives have been processed in this manner, there is no need for any further scan-conversion for this product, because we already have its visible faces in buffer 2 and we

```
FOREACH x IN pixels DO initialize final buffer
    {Z1[x] = infinity;          back plane
    I1[x] = black;}             background color
FOREACH P IN products DO
    {FOREACH x IN pixels DO initialize product buffer
        {Z2[x] = 0;                 init buffer 2
        I2[x] = black;}

    k=-1;
    UNTIL(k==NumberOfPrimitivesIn(P)) REPEAT
        {k++;                       count useless passes
        change=0;                   set if any pixel has changed
        Q= NextPrimitiveInTheCircularListOfPrimitivesIn(P);
        FOREACH x IN pixels DO initialize Z3, I3, and ff
            {Z3[x] = infinity;
            I3[x] = black;
            IF IsPositive(Q) THEN ff[x]=1 ELSE ff[x]=0;}
        FOREACH F IN faces(Q) DO
            {FOREACH x IN PixelsVisitedByScanconverting(Q) DO
                Z=DepthOfScanconvertedPointOn(F,x);
                I=IntensityReflectedByScanconvertedPointOn(F,x);
                IF (Z2[x] < Z - eps < Z3[x])
                    {Z3[x]= Z;
                    IF (IsFrontFacing(F)) I3[x]=I;}
                    IF (Z2[x] < Z - eps) ff[x]=!ff[x];}
        FOREACH x IN pixels DO move back if bad point
            {IF(ff[x] AND Z2[x] != Z3[x])
                {Z2[x] = Z3[x];
                I2[x] = I3[x];
                change = TRUE;}}
        IF (change) k= 0;}
    FOREACH x IN pixels DO merge into final z-buffer
        {IF(Z2[x] < Z1[x] + eps)
            {Z1[x] = Z2[x];
            I1[x] = I2[x];}}}
```

exit the UNTIL loop (lines 10-31). In the worst case, the trickle algorithm scan converts each primitive of the product roughly as many times as there is layers in all the primitives of that product. (Each layer may produce a tentative point for buffer 2, and it may be required to scan all but one primitives of the product to declare that the point in buffer 2 is out of the product.) However, in the average case, the algorithm stops the loop very early. Note that k is initialized to -1 (line 9) to ensure proper processing of products with a single primitive.

In the loop, we use a change flag (initialized line 12 and updated line 30) to see if we have advanced buffer 2 and if we should reset k to zero (line 31).

We execute the inside of the loop (lines 13-31) for primitives Q in the product. We circulate through the list of primitives in a consistent way (line 13). We first initialize buffer 3 and the front facing mask (lines 14-17). The mask is set to one if the current primitive is positive i.e., bounded. (We assume that each primitive has a bounded boundary. Primitives with a bounded interior are called positive. Others are called negative. In fact, if the leaves of the original CSG graph are bounded volumes, negative primitives correspond to leaves that have been subtracted an odd number of times.) The front facing flag ff is used to decide, at each pixel, whether the algorithm should advance buffer 2 or not (line 27).

Then we scan-convert faces of the current primitive Q (lines 18-25). For each pixel covered by the projection of Q, we update Z3 and I3 where appropriate using the depth and intensity of the scanned surface points (lines 22-24). The update takes place only if the scanned Z value lies between the depth stored in buffer 2 and 3 (line 22). The small *eps* value subtracted before the test is used to ensure correct treatment of coincident faces (see next section). Furthermore, I3 is only updated if the scanned surface is front-facing, so that we do not store intensities of back facing faces, should they overlap with front-facing ones near silhouette edges (see next section).

The front facing flag is flipped for each face of Q that passes behind Z2 (line 25). Again, the *eps* value is used to ensure that scan conversion accuracy and round-off errors do not lead to incorrect results.

Finally, once the entire primitive Q has been scan-converted, we advance by copying buffer 2 into buffer 3 wherever the front flag ff is set (line 27-31), i.e., wherever the points stored in buffer 2 are out of Q. Note that these points are replaced by front-facing points on Q (if any), which have been computed in buffer 3.

The additional condition, Z2[x] != Z3[x] (line 27), is necessary to ensure proper treatment of points outside of the projection of positive primitives. For these points ff=1, but Z2 and Z3 are both equal to the maximum depth, and the algorithm does not progress.

Line 33, an epsilon is used to prevent color mixing between overlapping faces of different products.

3 Treatment of Singularities

Singular situations occur when two different faces cover the same pixel and have the same—or almost the same—depth at points that project onto that pixel. Such situations occur in the mathematical (exact) model when faces of several primitives overlap. They also occur in the discretized graphic model near edges that connect a front and a back face or at constrictions (thin walls whose depth is less than the depth resolution of the z-buffer). A method for computing the correct picture in all these singular cases must be sufficiently robust to handle depth-errors due to the round-off errors of scan-conversion.

In this section, we discuss these singular cases and show how they are handled by our algorithm.

3.1 Silhouette Edges Yield on/on Cases

Different faces of the same primitive typically do not coincide. However, if a layer of the primitive has depth smaller than the depth resolution of the z-buffer, it will be processed as if it was flat. (Layers are more precisely defined in [15]. They correspond to the disjoint segments obtained by intersecting a ray parallel to the z-axis with the primitive's volume.) Furthermore, even for large primitives, we can locally have a situation where the primitive appears flat. For example, while approaching a silhouette edge, the abutting two faces of the same primitive (one front-facing and one back-facing) are arbitrarily close to each other in depth. At that edge both front and back faces have the same depth. If a pixel very close to the edge's projection on the screen is visited by the scan-conversion, both faces would have the same integer-rounded depth-value at that pixel. Thus, for that particular pixel, the primitive appears as a flat (zero-depth) degenerate solid. Since the trickle algorithm proceeds independently for all pixels, it must correctly handle such degenerate primitives, otherwise cracks or dangling edges may appear. The initialization (line 9) of the counter k to -1 ensures that products with a single primitives will be scan-converted twice, thus giving a chance for the trickle algorithm to produce a tentative point in buffer 2 during the first scan-conversion, and then to classify it as out during the second scan-conversion.

3.2 Need for Tolerance

Rotations used to position primitives introduce round-off errors in the coefficient of their bounding planes or vertices. Consequently, if for example primitives are rotated to align some of their faces, the surfaces containing faces that should overlap will usually not coincide, even though theoretically should. Furthermore, scan-conversion round-off errors may result in unpredictable depth-ordering of any two theoretically coincident faces at any pixel. Since the ordering based on pure depth comparisons would not be consistent across the entire overlap region of both faces, algorithm cannot rely on the result of depth tests for the covered pixels.

Therefore, in all computations that address the problem of coincident faces, we use a small tolerance value, called '*eps*' in our algorithm. This value will ensure that two depth values that were intended to be equal, will be considered equal.

Of course, choosing *eps* too large may result in treating as equal two values which were not intended to be equal. In such cases, the algorithm will produce a picture, that corresponds to a regularization 'modulo *eps*' of the solid, i.e., will remove shallow parts of the model and will display the correct faces everywhere else.

Treatment of on/on cases involves neighborhood evaluation [2]. Since we are testing faces front-to-back, only the neighborhood behind the face is relevant. (The neighborhood indicates whether there is material, with respect to the product, behind the face. If there were material in front of the face, the trickle algorithm would have stopped sooner at that pixel and would never have reached that face. Therefore, if there is material of the product behind the face, the face point is on the product and is the most-front point visible through the corresponding pixel.) We can thus use a technique proposed in [9] and test a point positioned behind the scan-converted surface by a small distance, *eps*. So, we subtract *eps* from the depth of the scan-converted point before comparing it to the tested point, whose depth is stored in Z2 (line 22).

The distance *eps* is chosen so as to exceed the combined effect of the depth-buffer limited resolution and of the round-off errors of the depth calculation during surface scan-conversion. When too large an *eps* value is used, details, such as shallow features, could disappear because their front-facing and back-facing faces would be considered coincident. To keep *eps* small and still correctly process the coincident face on/on cases, the staring depth along each scan-line must be accurate as elaborated in the next subsection.

3.3 Scan-conversion Consistent with the Supporting Surface

Scan-conversion procedures, which compute surface points for all the pixels covered by a face, produce approximate (truncated) depth values, because the z-buffer contains integer values. Consequently, the depth of a point that lies on the overlapping portion of two coplanar faces differs depending on which face is used (i.e., scan-converted) to produce the point. Therefore, to ensure that intended coincidences are correctly treated, we consider that two points that project onto the same pixel are identical if their depths differ by less than a small *eps* value computed from the size of the scene and the z-buffer resolution.

If a face is at a steep angle relative to the z axis, the sampled depth value may vary widely over the width of the pixel. To handle coincident faces correctly, it is important that every face be sampled at exactly the same point within the pixel (e.g., at the center of the pixel). If two faces are coincident, the sampling of depth values for the pixels covered by both faces must yield the same depth for both faces at any pixel.

We have obtained excellent results by computing the depth for the starting pixel of each horizontal span, not using a z-increment along the leading edge (as it is the case in most scanning algorithms), but precisely as the exact depth of the surface point that projects onto the center of the pixel. Consequently, within the numerical accuracy of the computation of the z-increment and its use along a scanline, the depth for all covered pixels will be correct with respect to the scanned surface, and will thus be the same for all faces lying in that surface.

3.4 Parity-based Point/Primitive Classification

In the original version of the Trickle algorithm [15], face trimming (i.e. testing face points) against a primitive Q in a product was done by checking whether the first face of Q encountered behind the tested point is front-facing or back-facing. Near the edges of Q, or if Q is flat within the depth buffer resolution, both front-facing and back-facing faces may have the same depth (due to truncated depth calculations). To solve the problem, we use the parity of the number of faces behind the tested point (line 25), as suggested in [14]. (An even number of faces of Q behind a point implies that the point is outside of Q.)

3.5 Scan-conversion Consistent with Primitive Boundary

The trickle algorithm requires that every pixel covered by the projection of a primitive be covered by a number of front faces that equals the number of back faces of that primitive. This requirement obviously indicates that primitives' boundaries should be valid two-cycles (no interior or dangling faces), which is the case theoretically in the definition of CSG. However, scan-conversion procedures dealing with two adjacent faces of the same primitive do not automatically ensure such parity. For example, the Bresenham or anti-aliasing algorithms do not.

A simple interpolation of the depth value over a span (horizontal row of pixels covered by the face projection) may lead to a wrong calculation of the depth at the end of the span if the end only partially covers the pixels. (The depth would be extrapolated using the slope of the plane containing the face, even though the actual pixel's center is not covered by the face.)

Consider a pixel whose rectangular region is traversed by the projection of an edge between a front and a back face, but whose center is not covered by these faces. If the scan-conversion algorithm visits that pixel for these two faces, a depth value will be computed for the center of the pixel. No matter how big the depth resolution or the *eps* tolerance value, one can choose the slope of the two faces such that the depth at the pixel's center computed for the front face exceeds the computed depth of the back face. This overshoot can lead to incorrect pictures near silhouette edges of primitives. The solution we have implemented simply processes, during scan-conversion, only those pixels whose centers are covered by the face.

Shadows are handled by using an auxiliary 'shadow' z-buffer that selects the surface portions visible from the light source. It is constructed by running the above trickle algorithm (without computing any intensity information) in the coordinate system that positions the eye at the light source. Then the standard trickle algorithm is run again from the eye orientation, except that, while computing the intensity reflected by visible points, their distance to the light is compared to the distance stored in the shadow buffer to establish if the surface point is visible from the light source, i.e., is lighted or is in the shadow of some other surface closer to the light source.

The above approach requires that during the final scan-conversion (after the shadow buffer has been computed) the coordinates of surface points be expressed in both coordinate systems (the coordinate system aligned with the viewer and the one aligned with the light source). To ensure that all pixels covered by the scan-converted surface are correctly processed, the scan-conversion uses increments in the viewer's coordinate system. These increments may be mapped into increments in the light coordinate system through a constant matrix so as to speed the scan-conversion process up.

The aliasing effect in the shadow buffer can be significantly accentuated if the surface upon which the shadow is projected is orthogonal to the viewing direction and nearly parallel to the light direction. Some techniques for dealing with such visual artifacts are discussed in [11].

An example, presented in the appendix, shows the result of this two pass algorithm.

4 Constructive Examples

Let us illustrate how the new trickle algorithm works by considering several examples. In these examples we will only concentrate on the computation of the visible front faces of simple products, because the sum (or union) of products is performed by the standard z-buffering hidden surface removal approach.

4.1 Intersection of Two Simple Primitives

Consider two convex primitives, A and B, of Figure 2. Our algorithm proceeds as follows. Before we start scan-converting primitives, we initialize the pixels' buffers memory. Z2 is initialized (line 6) to 0 (for simplicity, we assume that the world lies on the positive side of the Z=0 plane), and I2 to the background color (line 7), which we call 'black'. The count k of redundant passes is initialized to -1.

Primitive A appears first in the circular list of the primitives in our product, so we scan-convert it first. Since A is positive (and thus bounded), the parity bit flag 'ff' for each pixel is initialized to 1, so that points stored at pixels not covered by the projection of A will be properly treated as being exterior to A. Z3 is initialized to 'infinity' (a number representing the maximum depth of the z-buffer) and I3 is set to black (lines 15-17).

During the scanning of the faces of A, we find that, at least for the pixels we consider here, we have the following relation: $Z2 < F1 < F4 < Z3$. The test (line 22) succeeds, for small enough *eps*, and thus the depth of F1 is stored in Z3 and the color of F1 is stored in I3. In short: we are storing the front face of A in buffer 3. Both F1 and F4 are behind Z2, and thus the parity flag ff is toggled twice during the scan-conversion of A and remains 1 (the tested point in Z2 is outside of A and should be replaced by points further back).

During the update steps (lines 26-30), the contents of Z3 and I3 which corresponds to face F1 are copied into Z2 and I2. The count k of redundant passes is reset to 0, since we have updated some pixels. Note that pixels outside the projection of A now contain in buffer 2 the background depth and color.

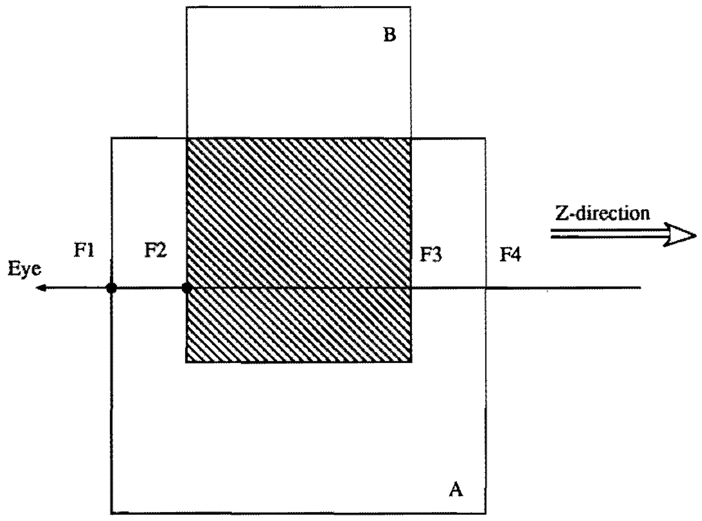


Fig. 2. Intersection of two convex primitives: The product is the intersection of two convex primitives, A and B. A has a front face F1 and a back face F4. B has a front face F2 and a back face F3. The other faces of A and B do not project onto the pixels of interest in this example and will thus not be considered. The eye is on the left and thus the z-axis direction is horizontal and left-to-right. We consider a particular pixel symbolized by a horizontal line through the eye. The face of the product that is visible through the pixel is the front face F2 of B.

Now we scan primitive B. We increase k to 1. The parity flag is also set to 1 and Z3 is initialized to 'infinity'. We find that: $Z2 < F2 < F3 < Z3$, so the depth and intensity of F2 are copied into Z3 and I3. Once more Z2 is less than F2 and F3, so, the parity flag is toggled twice. (The tested point is out of B.) The content of Z2 and I2 are overwritten with data from face F2 and the counter k is reset to 0 again. The algorithm has moved forward and F2 is stored in buffer 2.

The next primitive must be considered, but first we increase k to 1 and set the parity flag at each pixel to 1. Going through the circular list of the primitives in the product, we are back to primitive A. Scan-converting A again, we find that: $F1 < Z2 < F4 <$

$Z3$. Face F1 is not considered, because it does not lie behind Z2. The depth of F4 is stored in Z3, but, since F4 is back-facing, the color of F4 is not copied into I3! Only one face of A, F4, is behind Z2, so the parity flag is toggled only once and is set to 0. Consequently, Z2 is not changed and the counter k is not reset and is still 1.

The next primitive is B. K is incremented to 2. The parity flag is set to 1. Z2 contains the depth of F2 and scan-converting B we find that: $Z2 < F3 < Z3$. The depth of F3 is copied into Z3, but the color is not copied into I3 because F3 is back-facing. Only F3 is behind Z2 so the parity flag is toggled to 0 at each pixel covered by the projection of F3. Z2 is not changed and k remains at 2. Buffer 2 still contains the front face, F2, of B.

Since k equals the number of primitives in the product, we stop the loop (line 10). The product's visible front face is in buffer 2. The contents of Z2 and I2 are copied into the display buffers Z1 and I1, wherever Z2 is less than Z1, so as to merge this product with other products of the disjunctive form.

4.2 Coincident Face and One Non-convex Primitive

Now consider the intersection of a non-convex primitive A with a convex primitive B, as shown Figure 3. A has faces F1, F3, F4, and F5 and B has faces F2 and F6. The front face F1 and F2 are coincident.

As before, we first initialize buffer 2.

We start by scanning A. As in the previous example, the parity flag is set because A is positive. Because $Z2 < F1 < F3 < F4 < F5$, F1 is stored in buffer 3. Since there are 4 faces behind Z2, ff is set to 1 and F1 is copied into Z2 and I2 (lines 26-30).

Now, when we scan B, the face F2 is coincident with F1, and we have $F2 < Z2 + eps < F6$. With only 1 face of B that is behind Z2, the parity flag is toggled to 0 and Z2 remains unchanged. The counter k is incremented to 1.

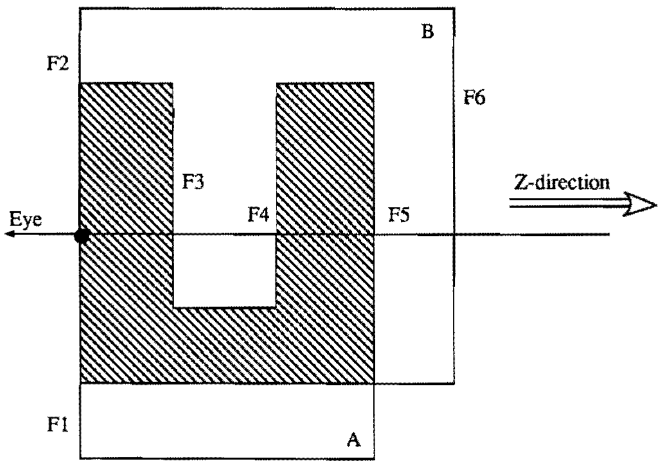


Fig. 3. A non-convex primitive: A has two front faces, F1 and F4, and two back faces, F3 and F5. B has a front face F2 and a back face of F6. The product is defined as the intersection of A and B.

When we scan A again, and find that $Z2 < F3 < F4 < F5$. The depth of F3 is stored in Z3, but I3 is not changed because F3 is back-facing. With an odd number of faces of A behind Z2, the parity flag is toggled to 0 and Z2 remains unchanged. The counter K is incremented to 2, which equal to the number of primitives in the product, and thus the loop stops. Buffer 2 contains F1, which is the front of the product, and can be merged into buffer 1.

4.3 Coincident Face and an Unbounded Primitive

Consider, the difference of one convex primitive, A, with another convex primitive, B, of Figure 4. The product is the intersection of A with the complement of B, which is a negative unbounded primitive. The front/back orientation of the faces of B are inverted because it is complemented.

Initialization takes place as usual.

We scan A, as in the previous examples, and F1 is stored in buffer 2.

Now, we scan B. Because B is negative, the parity flag ff is initialized to 0. We find that, $F2 < Z2 + eps < F3 < Z3$. Therefore, F3 is stored in buffer 3, including the intensity, because F3 is the front-facing (remember that B is negative). With 1 face of B behind Z2, the flag ff is toggled to 1 at the pixels visited by the scan-conversion: F2 is outside of B and we move forward. Buffer 3 is copied into buffer 2.

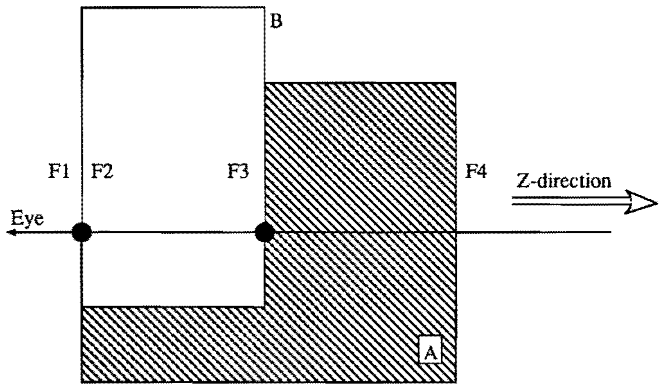


Fig. 4. Subtraction of simple primitives: In the product A-B, the front face, F1, of A coincides with the front face F2 of B. Note that, because B is negative in the product, F2 is treated as a back face, because we are intersecting A with the complement of B. The visible face of the product is F3, the original back face of the non-complemented primitive B.

When we scan A again and set the parity flag to 1, only F4 is behind Z2, therefore, the parity flag is toggled once and set to 0. Consequently, Z2 remains unchanged. The same happens as we scan B for the second time. Then the algorithm stops for this product, since k has reached 2. Buffer 2, containing F3, is merged into buffer 1.

4.4 Primitives with Internal Cracks

Finally, we look at the subtraction of one non-convex primitive, B, (which has an internal crack) from another non-convex primitive, A, (which also has an internal crack, which furthermore coincides with the crack in B) depicted in Figure 5. Such cracks are produced when, in the model—or because of the limited depth resolution—two faces of the same primitive are coincident. These situations also appear close to silhouette edges, as discussed earlier.

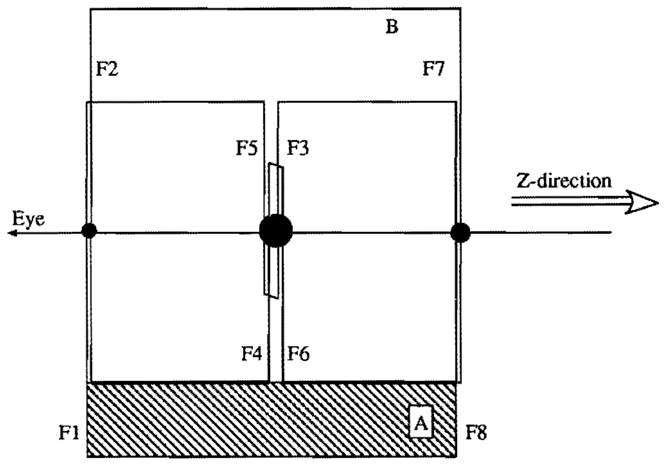


Fig. 5. Coincident cracks: A has faces F1, F4, F5, F7. B has faces F2, F3, F6, F8. The non-regularized difference A-B, that projects onto the pixels considered here is the empty set.

After scanning A, as in the previous examples, F1 is stored in buffer 2.

When we scan B, we set the parity flag to 0 since B is negative. F2 is coincident with F1, so $F2 < Z2 + eps < F3 = F6 < F8 < Z3$. The depth of either F3 or F6 is copied into Z3, depending on the order in which faces of B are scan-converted. However, the intensity of F3 is saved in I3, because F6 is backfacing.

Since there are 3 faces of B behind Z2, the parity flag is toggled to 1 and buffer 3 is copied into buffer 2.

We scan A again. The parity flag is set. F4 is coincident with F5 and with Z2. We have, $F1 < F4 = F5 < Z2 + eps < F7 < Z3$. F7 is written into Z3. Since only F7 is behind Z2, the parity bit is reset to 0 and Z2 remains unchanged. Buffer 2 still contains F3.

Now scanning B again, F6 and F3 are coincident with Z2. We have: $F2 < F3 = F6 < Z2 + eps < F8 < Z3$. F8 is written into buffer 3. Since only one face (F8) is behind Z2, the parity flag is toggled once to 1 and F8 is copied into buffer 2.

Scanning A again, the last face of A, F7 is coincident with F8. We have: $F1 < F4 = F5 < F7 < Z2 + eps < Z3$. There is no face of A behind Z2. Therefore, nothing is copied into Z3 and the parity flag remains set. Thus the background color (black) and maximum depth (infinity), with which buffer 3 was initialized are copied into buffer 2.

A further pass causes no change and the process stops when the count reaches 2.

5 Conclusion

To display regularized CSG solids using a multiple depth buffer algorithm, singular cases, where two faces have the same depth at some pixel, must be handled properly. These coincident-face situations do not only happen because the CSG primitives have been positioned with two-dimensional contacts along their boundaries. The situations also happen when surface points on constrictions or on sharp corners near silhouette edges project onto the same pixel and have depth-values that are sufficiently close to be rounded by the scan-conversion process to the same integer Z value.

First, the authors have decided to use a small *eps* tolerance value to remove the effects of round-off errors during scan-conversion. For example, faces that were designed to coincide, will, even if the actual depth may differ at some pixels. To keep *eps* small, relative to the size of the model, we have improved scan-conversion, so that it produces actual surface depths for all the visited pixels. This way, if two faces that overlap in space are scan-converted independently, the pairs of values generated for all the pixels covered by both faces will be equal, except for a very small round-off error that may occur during the depth increment cumulation.

To ensure that the scan-conversion may be used safely for point-in-primitive classification, we have modified the scan-conversion to guarantee that only pixels whose centers are covered by a face are visited during that face's scan-conversion.

To produce a picture that is correct with respect to the regularized interpretation of the CSG expression, toleranced depth tests are used within the trickle algorithm to remove dangling faces or edges that would otherwise appear.

Shadows, which may exhibit a fair amount of aliasing, may be produced using a simple two-pass algorithm with an auxiliary shadow-buffer.

Acknowledgements

The original idea of combining a z-buffer depth-test with a CSG classification was described in [9]. The work at IBM on z-buffer techniques for shading from CSG was particularly inspired by the work of J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs, at UNC reported in [14] and by some of Prof. F. Jansen's work [4]. It started as a collaboration between N. Gharacharoloo, C. Zoulas, and J. Rossignac at IBM Research, who were later joined by F. Jansen, visiting IBM from Delft Technical University in the Netherlands, and by D. Epstein. Our approach was originally focused on an improvement to the UNC algorithm, developed by Rossignac and based on the fact that the visible front-faces of products of convex (or z-connected) primitives or half-spaces can be computed in one pass by scan-converting only once each primitives' front faces into one depth and intensity buffer; by scan-converting only once each primitive's back faces into another depth-buffer; and by combining these two buffers. Several improvements were made to this approach in order to reduce the average complexity of the algorithm in terms of the number of primitives in the CSG tree. The work culminated in the trickle algorithm developed by D. Epstein, F. Jansen, and J. Rossignac and reported in [15]. The present paper relates recent progress made by its authors on simplifying the trickle algorithm and enabling it to correctly handle important singular situations and to deal with the round-off errors due to the z-buffer and scan-conversion limited resolutions.

References

[1] Requicha, A. A. G. and Tilove, R. B.: Mathematical foundation of constructive solid geometry: General topology of closed regular sets, *Tech. Memo. No. 27a, Production Automation Project, Univ. of Rochester*, June 1978. (Available from CPA, 304 Kimball Hall, Cornell University, Ithaca, New York 14853.)

[2] Requicha, A. A. G. and Voelcker, H. B.: Boolean Operations in Solid Modelling: Boundary Evaluation and Merging Algorithms, *Proceedings of the IEEE*, Vol. 73, No. 1, January 1985.

[3] Roth, S. D.: Ray casting for modeling solids, *Computer Graphics and Image Processing*, vol. 18, no. 2, pp. 109-144, February 1982.

[4] Jansen, F. W.: Pixel-Parallel hidden-surface algorithm for Constructive Solid Geometry, *Proceedings Eurographics'86*, pp. 29-40, Elseviers Science Publishers, Amsterdam. 1986.

[5] Jansen, F. W.: CSG hidden-surface algorithms for VLSI hardware systems, in *Advances on Graphics Hardware I* W. Strasser (ed.), Springer Verlag, 1987.

[6] Jansen, F. W.: Solid modelling with faceted primitives, Delft University Press, *Doctoral Dissertation*, Technische Universiteit Delft, The Netherland, September 1987.

[7] Kedem, G. and Ellis, J. L.: The ray-casting machine, *Proc. ICCD'84*, pp. 533-538, October 1984.

[8] Kedem , G. and Ellis, J.: The Ray Casting Machine Prototype, *International Conference on Parallel Processing for Computer Vision and Display*, University of Leeds, UK, January 1988.

[9] Rossignac, J. R. and Requicha, A. A. G.: Depth buffering display techniques for constructive solid geometry, *IEEE, Computer Graphics and Applications*, vol. 6, no. 9, pp. 29-39, September 1986.

[10] Rossignac, J. R. and Voelcker, H. B.: Active Zones in CSG for Accelerating Boundary Evaluation, Redundancy Elimination, Interference Detection, and Shading Algorithms, *ACM Transactions on Graphics*, vol. 8, no. 1, pp. 51-87, January 1989.

[11] Woodward, C.: Methods for computer-aided design of free-form objects, *PhD dissertation, Mathematics and Computer Science Series number 56*, Helsinki University of Technology, Finland. 1990.

[12] Goldfeather, J., Hultquist, J. P. M. and Fuchs, H.: Fast Constructive Solid Geometry Display in the Pixel-Power Graphics System, *ACM SIGGRAPH'86 Proc.*, Computer Graphics, vol. 20, no. 4, August 1986.

[13] Tilove, R. B.: Set membership classification: A unified approach to geometric intersection problems, *IEEE Trans. on Computers*, vol. C-29, no. 10, pp. 874-883, October 1980.

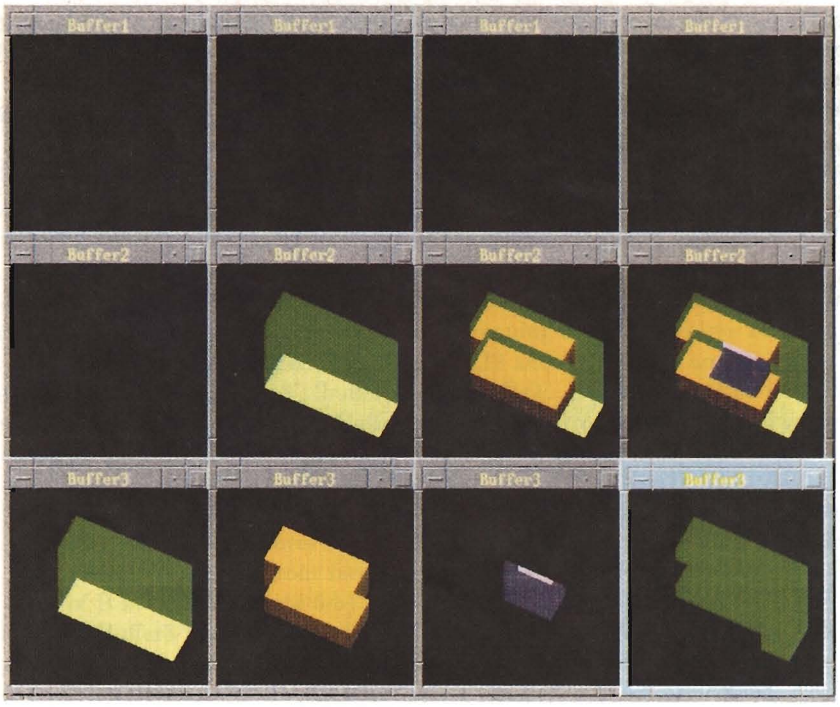
[14] Goldfeather, J., Molnar, S., Turk, G. and Fuchs, H.: Near Real-Time CSG Rendering using Tree Normalization and Geometric Pruning, *IEEE Computer Graphics and Applications*, vol. 9, no. 3, pp. 20-28. May 1989.

[15] Epstein, D., Jansen, F. and Rossignac, J.: Z-buffer rendering from CSG: The Trickle Algorithm, *IBM Research Report RC15182*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, December 1989.

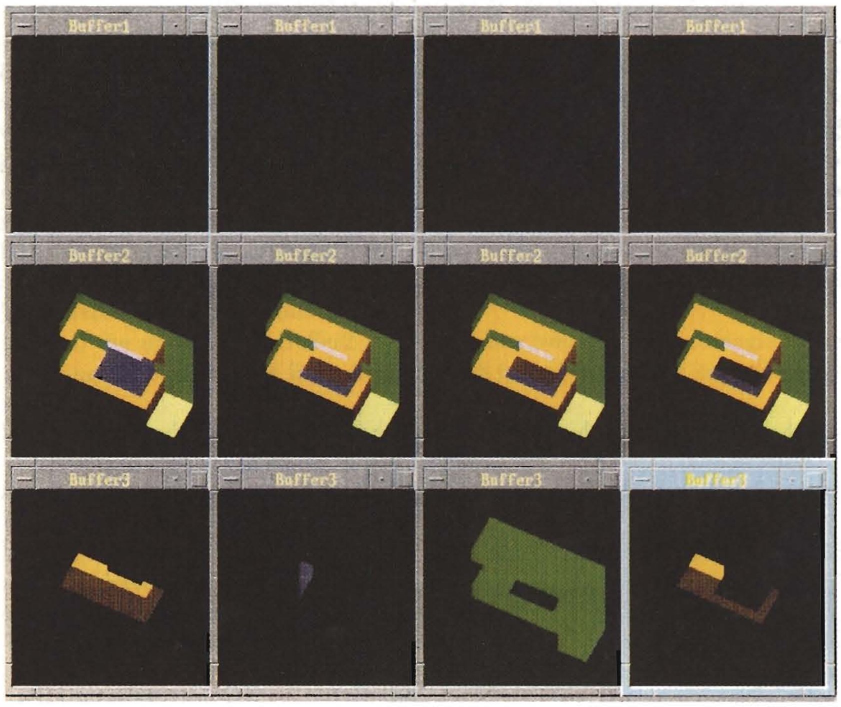
Appendix A: A Case Study

The following sequence of pictures show how the regularized CSG expression $(A - (B \cup C)) \cup (B \cap D)$ is processed by the new trickle algorithm to generate a correct shaded image. Primitive A is the large green block; B is a non-convex primitive composed of two rectangular brown blocks. Primitive C is a purple block and D is a light blue block. A and B share coincident faces at the front and back sides. B and C share coincident faces at the back end, and B and D shard coincident faces at the right end. The expression is first converted into sum of products form. We obtain two products: $A \cap \bar{B} \cap \bar{C}$ and $B \cap D$. Each picture shows a series of time steps in the running of the algorithm. For each time step, the contents of the 3 buffers (Z3,I3 and Z2,I2 and Z1,I1) that are used are shown in three windows called 'buffer3', 'buffer2', and 'buffer1'. Buffer3 shows the result of the current primitive's scan-conversion. Buffer2, shows the current result of merging the scan conversion previously obtained in Buffer3 with the previous state of Buffer2. Buffer1 stores the resulting products of the CSG sum of product forms.
We first scan A into buffer3. (Plate 6-a) Then buffer2 is updated with the contents of buffer3 and we see A's front faces in buffer2. Then we scan B into buffer3. Because the front flag is reversed and B's front face is coincident with A's front face, we see only the back faces of B in buffer3. (Plate 6-b).
In Plate 6-c, Buffer3 with B's back faces is merged into Buffer2. We see the back-faces merged into the image of A in Buffer2 creating the two shelves. The updating is not a simple merge, only those pixels for which the front flag ff is set are copied. Note that there is the dangling back faces of B which are coincident with the back face of A. It will disappear later to produce the correct image of a regularized solid. C, which is also negative, is scanned into Buffer3. Only the parts of the back faces of C that are behind Buffer2 appear in Buffer3.
In Plate 6-d, the faces of C are updated into Buffer2 creating the notch. Part of the back face of C is coincident with a back face of B; when this occurs the color of the visible face depends on the order in which the primitives are processed. The lightly colored area is from the coincident face of C that was behind Buffer2. Note the dangling backfaces of C are also present. We scan A again into Buffer3, and only the backfaces of A behind buffer2 are visible.
In Plate 6-e, the updating of Buffer2 by Buffer3 have trimmed away the dangling back faces of B from the image in Buffer2. We scan B into Buffer3 and we see only a small part of B's backfaces appearing in Buffer3.
In Plate 6-f, the updating of Buffer2 by Buffer3 has overwritten the dangling backface of C restoring the back faces of B. A dangling back face of B has been created. We scan C again.
In Plate 6-g, the updating of Buffer2 by Buffer3 has caused no change in the image. We scan A again and see the backfaces of A in Buffer3.
In Plate 6-h, the updating of Buffer2 by Buffer3 has trimmed the last dangling back face of B from the image in Buffer2. We scan B again.
In Plate 6-i, a repeated scanning of C has not changed the contents of Buffer2. So we know that we have computed the correct visible front faces of the product in Buffer2 and we copy it into Buffer1. (Plate 6-j,k)
In 6-k, we start processing the second product. B is scanned into Buffer3. Buffer3 is updated into Buffer2 in Plate 6-l. We scan D into Buffer3 and see the part of D that is behind the front faces of B.

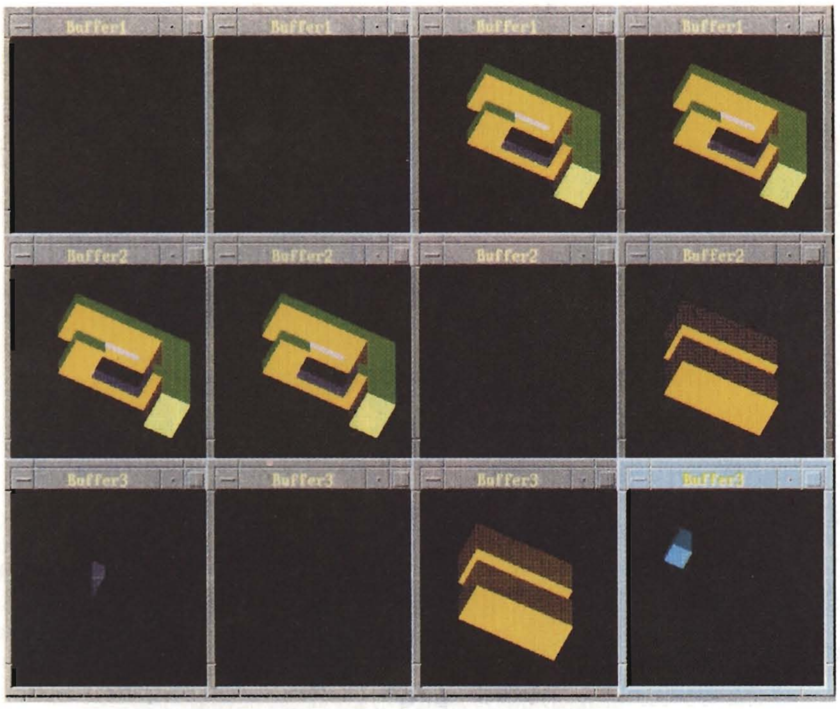
In Plate 6-m, we see the result of the update of Buffer2. Note that one of the faces of the images belongs to B. Scanning B into Buffer3, we see the backface of B that is behind the image in Buffer2.
In Plate 6-n, the image in Buffer2 is trimmed by the contents of Buffer3 to produce a block. We scan D into Buffer3.
In Plate 6-o,p, because the passes over B and D have caused no change in Buffer2, we merge Buffer2 into bufferl to get the final image.
In Plate 6-q, we apply the shadow algorithm to produce the shadows on the final image.



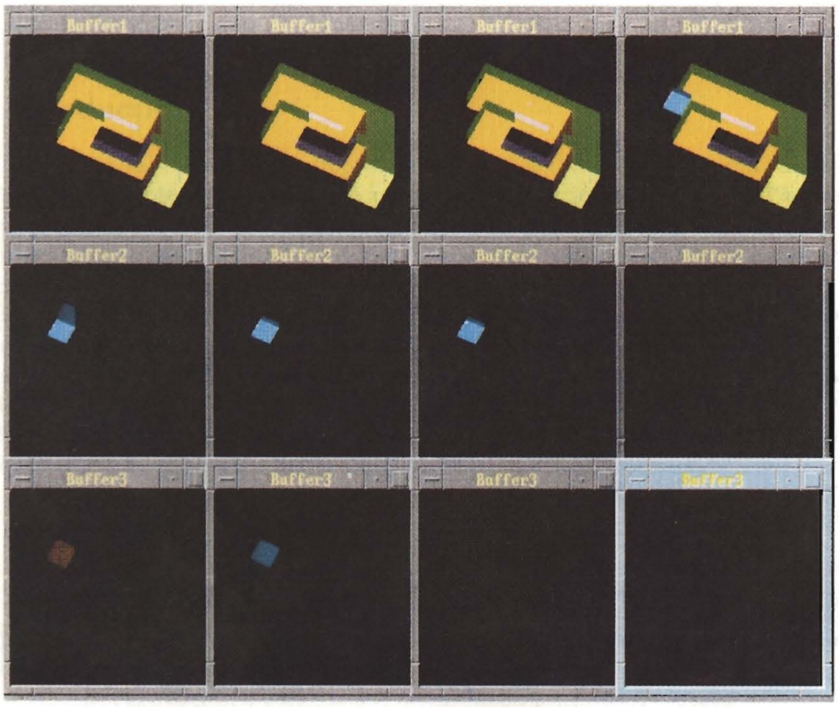
6 - a 6 - b 6 - c 6 - d



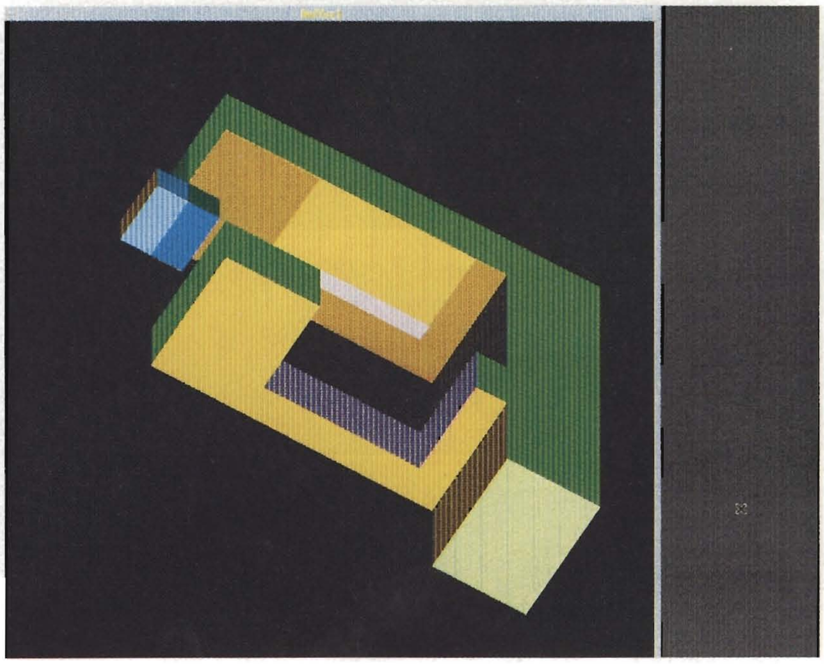
6 - e 6 - f 6 - g 6 - h



6 - i　　6 - j　　6 - k　　6 - l



6 - m　　6 - n　　6 - o　　6 - p



6 - q