# GEOMETRIC MODELING: A First Course



## 7. Application Algorithms

Unambiguous geometric models are potentially capable of supporting fully automatic algorithms for any applications that involve object geometry. However, only a few application algorithms have reached maturity and are routinely used in industry. This chapter discusses currently understood applications in graphics and simulation; mass-property calculation (i.e., volume, moments of inertia); and interference (i.e., collision) analysis. It also touches upon other applications such as planning for inspection and robotics, which are emerging from the research labs. Generally, *analysis* algorithms are well-developed, whereas *synthesis* algorithms, which require *geometric reasoning*, are not.

### 7.1 Graphics and Kinematic Simulation

#### 7.1.1 Overview

Most of the rendering software in use today operates on BReps. Simple images are produced swiftly, but photo-realistic renderings, with texture, shadows, and so on, may take several minutes or even hours per image. If objects are modeled by using Boolean operations, the BRep must be evaluated before rendering, and this is a time consuming procedure. Thus, although rendering itself is fast, the entire process suffers from what is sometimes called “the Boolean bottleneck”. Rendering methods that do not require boundary evaluation, and therefore avoid the Boolean bottleneck, are attractive for modeling systems that define objects primarily through Booleans. Graphic algorithms for BReps are covered in standard graphics texts, and are not discussed extensively here. We focus on algorithms that do not require explicit boundary information.

Graphic displays of solids and surfaces are typically produced in three styles:

- Wireframes.
- Line drawings with hidden lines removed.
- Shaded displays.

*Wireframes* – All the edges of the object are displayed, regardless of whether they are truly visible or not. For curved objects, which have few edges, displays often contain additional curves, usually called *generators*. These may be computed by intersecting the object with a set of parallel planes, or, more commonly, by tracing curves of constant parameter value in parametric surfaces. Figure 7.1.1.1 provides an example. Given parametric representations for the curves to be drawn, the display process consists of (i) stepping along the curve through suitable parameter increments, (ii) generating a piecewise linear approximation on the fly, and (iii) projecting the line segments on the screen, as discussed in Chapter 2. Parameter increments may be constant, or they may be smaller in regions of high curvature,

for better approximation and visual effect. Wireframe displays are easy to generate from BReps (especially if generators are not used), but hard to interpret for complicated objects. Rotating a wireframe helps human users perceive the represented object (or objects, if the internal representation happens to be an ambiguous wireframe).

Figure 7.1.1.1 – Wireframe display of a curved solid.

*Line drawings with hidden lines removed* – Here the invisible edges or edge segments are not displayed. The resulting images are much more intelligible than their wireframe counterparts. However, hidden-line removal requires substantial computation. For objects with curved surfaces, *profile* or *silhouette* curves are also computed and displayed, for a more realistic effect. A silhouette corresponds to the points of the object where the surface normal is perpendicular to the line of sight. If $\mathbf{p}(u,v)$ is a generic point on the surface, $\boldsymbol{n}(u,v)$ is the normal vector to the surface at $\mathbf{p}$, and $\mathbf{v}$ is the viewpoint, the equation of a silhouette is $(\mathbf{v} - \mathbf{p}) \cdot \boldsymbol{n} = 0$. This is an equation in the two parametric variables $u$ and $v$, and therefore implicitly defines a curve in parameter space. Note that silhouettes are not edges in a BRep, and are viewpoint dependent. Figure 7.1.1.2 presents a simple example that shows that some BRep edges are not displayed, because they do not correspond to discontinuities in the surface normal, and that some edges (silhouettes) that are not in the BRep are displayed. Hidden-line algorithms normally operate on boundary representations and are discussed at length in texts on Computer Graphics.

Figure 7.1.1.2 – A display with hidden lines removed.

*Shaded displays* – These are the most realistic, but also the most expensive to compute. They can be very sophisticated, with texture, reflectance effects, shadows, and so on. Figure 7.1.1.3 is an example of a high-quality shaded display of a metalic part. Often shaded displays are generated from boundary representations by *visible surface* algorithms, which are discussed in graphics textbooks. In this course we present only a very simple boundary representation algorithm, and then focus on shaded graphics algorithms for CSG, which are usually not described in the graphics texts. There are also algorithms for generating shaded displays directly from octree and voxel representations. These algorithms are simple, and especially attractive when implemented in hardware. Efficient software for rendering octrees is available commercially.

Figure 7.1.1.3 – A shaded display of an automotive part.

Kinematic simulation amounts essentially to displaying a sequence of images, or *frames*, in which an articulated object is shown in several poses along a trajectory of motion. (Here "frame" is not a coordinate system; it is used with its standard meaning in the animation field, i.e., as an individual image in an animation sequence.) Pose computation was discussed in Chapter 2. Major challenges in kinematic simulation are computational speed and bandwidth. For realistic motion effects the frame rate ideally should be at least 30 fps (frames per second). A 1,000 by 1,000 pixel display, with 24 bits of color per pixel, corresponds to 3MB (mega bytes) of data per frame. At 30 fps this is a data rate of 90 MB/sec, which is very large, and requires special hardware. In addition, motion tends to exacerbate certain imperfections in computed images. For example, if the silhouette of an object is not accuratetly approximated, the object/background boundary flickers annoyingly in the animation.

### 7.1.2 Depth Buffering

Depth- or z-buffering is a simple rendering technique, well-suited for hardware implementation, and widely used in graphic accelerators. It is especially useful for objects represented by BReps in which the faces are very simple polygons such as triangles or quadrangles. Such BReps are also called *tessellations*; if all the faces are triangular, the tessellation is called a *triangulation*.

The basic z-buffer algorithm uses two arrays `I[x,y]` and `Z[x,y]`. The indices `x,y` range over all the pixels of the display window. `I[x,y]`stores the intensity, or color value, that corresponds to pixel `x,y`, and `Z[x,y]` is the corresponding depth. The depth, or Z value, is the distance between the viewpoint and the point on the object's boundary that is being projected on the pixel—see Figure 7.1.2.1.

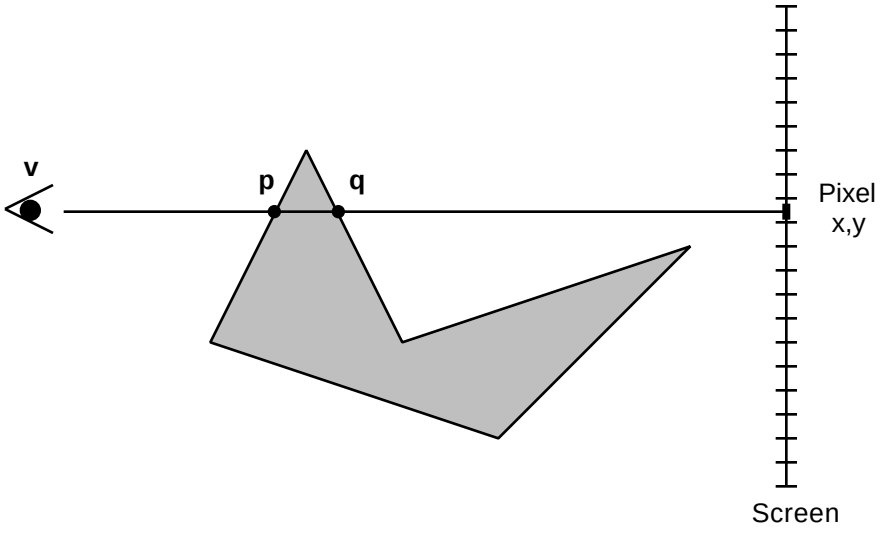


Figure 7.1.2.1 – Depth buffering geometry, illustrated in section view.

The agorithm may be summarized as follows: initialize the buffers; scan each face; check the distance between each point on the face and the viewpoint; if the distance is less than that stored in the z-buffer, overwrite the z-buffer and compute the new intensity by using information about the surface normal and the light sources. At the end of the scan the intensity buffer contains the correct image values. Enough points must be generated in each face to ensure that all the pixels covered by the face's projection have at least one corresponding face point.

In pseudo-code the algorithm may be expressed as follows.

**Depth Buffering Algorithm for BReps**

```
for each [x,y] do
     Z[x,y] = BigNumber; // "Infinite" distance
     I[x,y] = BackgroundColor;
     end; // Initialization loop
for each face F of solid S do
     // Scan convert the face
     for each point P in a dense grid on face F do
          [x,y] = ProjectOnScreen(P);
          d = Norm(V-P);
          // V is the viewpoint
          // Function Norm computes the length of a vector
          if d < Z[x,y] then
               Z[x,y] = d; // Update buffers
               N = NormalToSurface(P);
               I[x,y] = Intensity(P,N,LightSources);
               end; // if
          end; // Scan loop
     end; // Face loop
Display(I[x,y]);
end;
```

Figure 7.1.2.1 illustrates the need for depth testing. Points **p** and **q** project on the same pixel. If the z-buffer contains the depth that corresponds to **q** when **p** is reached in the face scan, then the buffers must be updated, to reflect the fact that only **p** is visible, because its depth is lower than **q**'s.

The z-buffer algorithm does not require any complicated geometric computations. For example, no intersections of surfaces or curves are computed. And it can support realistic rendering if the `Intensity` function is sufficiently elaborate. (Images produced with the algorithm as described have a jagged appearance, and should be anti-aliased. Anti-aliasing is a filtering operation necessary to combat sampling effects, and is described in standard graphics texts. It is analogous to low-pass filtering, familiar to electrical engineers.)

This z-buffer algorithm is not directly applicable to CSG representations, because the faces of an object are not explicitly available. But it can be extended to CSG by using the generate-and-test paradigm we encountered earlier, in the study of Boolean operations. We generate points on *primitive* solid faces, which are guaranteed to be larger than the faces of the object, and discard some of the points by classifying them with respect to the solid. Only those points that classify *on* the boundary are processed.

The CSG algorithm differs from its BRep counterpart only by the addition of the classification test implemented by function `ClPtSol()`. Note that it is always possible to avoid points lying on edges or vertices of the solid in the face scans. If necessary, a point can be moved randomly by a small amount, so that the perturbed point projects on the same pixel and lies in the interior of a face. This simplifies significantly the representation and combination of point neighborhoods in the point classifier. Many other techniques are available for speeding up the computations—see [Rossignac & Requicha 1986].

In pseudo-code the algorithm is as follows.

**Depth Buffering Algorithm for CSG**

```
for each [x,y] do
     Z[x,y] = BigNumber; // “Infinite” distance
     I[x,y] = BackgroundColor;
     end; // Initialization loop
for each face F of each primitive of solid S do
     // Scan convert the primitive face
     for each point P in a dense grid on face F do
          [x,y] = ProjectOnScreen(P);
          d = Norm(V-P);
          // V is the viewpoint
          // Function Norm computes the norm of a vector
          if d < Z[x,y] then
               if ClPtSol(P,S) == onS then
                    Z[x,y] = d; // Update buffers
                    N = NormalToSurface(P);
                    I[x,y] = Intensity(P,N,LightSources);
                    end; // Inner if
               end; // Outer if
          end; // Scan loop
     end; // Face loop
Display(I[x,y]);
end; // Algorithm
```

### 7.1.3 Ray Casting

Ray casting can be used with any representation scheme, but is most attractive for CSG. The basic idea is very simple: cast rays between the viewpoint and the screen pixels, and compute the first point of intersection between each ray and the object to be rendered. From information about this entry point determine the color to be displayed and write it to the appropriate pixel. In pseudo-code the algorithm is as follows.

**Ray Casting Algorithm for CSG**

```
for each Pixel P[x,y] do
     R = V - P; // Create ray from viewpoint V to pixel P
     RwrtS = ClLineSol(R, S); // Classify ray against solid S
     if RinS == 0 then I = BackgroundColor // No intersection
     else
          Q = FirstPoint(RinS); // Entry point into solid S
          N = NormalToSurface(Q);
          I = Intensity(Q, N, LightSources);
          end; // else
     Display(P, I); // Write color I on pixel P[x,y]
     end; // Ray casting algorithm
```

Whereas the z-buffer algorithm scans object faces and projects their points on the screen, ray casting scans the screen and computes the intersections of rays with the object’s faces. Unlike z-buffering, ray casting requires line/face intersections, which can be difficult to

compute when faces lie in complicated curved surfaces. Experimental results have shown that ray casting and depth buffering for CSG have comparable complexities, and that z-buffering has advantages when the object's surfaces are complex.

The efficiency of ray casting algorithms may be increased by a variety of techniques—see [Roth 1982]. For example, we can sample the screen, and cast a ray for every eighth pixel, say, in a scan line. If two consecutive rays hit the same face of the object we interpolate the intensities for the in-between pixels without casting additional rays. If two faces are hit, we subdivide the distance and cast a new ray half way between the two pixels, i.e., at a distance of four pixels from the first. Figure 7.1.3.1 illustrates the procedure. For simplicity, we assume the viewpoint is at infinity and the rays are parallel. We first cast rays 1 and 2, which hit Faces 1 and 2. Next we divide by half the distance between rays 1 and 2, and cast 3, which hits Face 1. We interpolate colors between pixels 1 and 3. Next we cast 4, which hits Face 2. We interpolate colors between 2 and 4. Finally we cast ray 5, which hits Face 1, and we assign the correct color to pixel 5. Note that this procedure is not entirely safe, because we will miss a small protrusion or depression if it is sited between two rays that hit a single face.

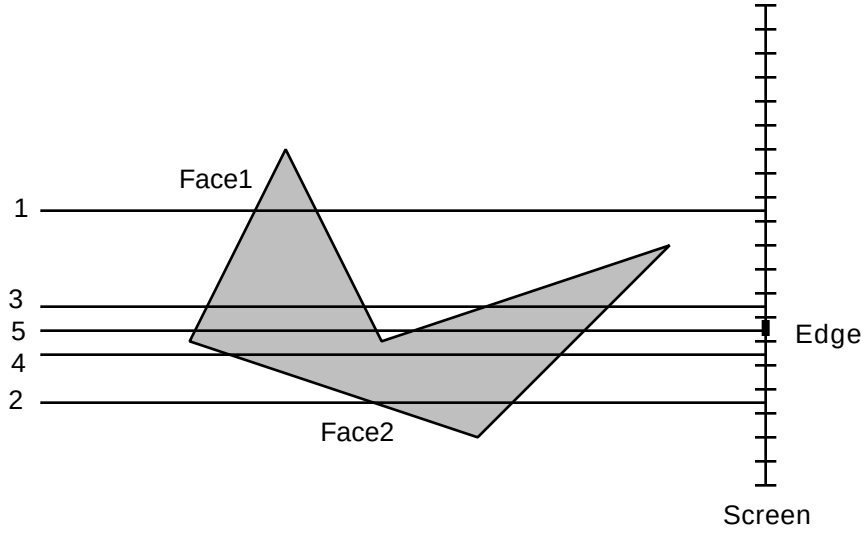


Figure 7.1.3.1 – Generating line drawings by ray casting.

Interestingly, this technique can be used to generate line drawings with hidden lines removed. In essence, we are searching for edges, because we cast rays until we find two successive rays, in adjacent pixels, that intersect different faces. If we turn on the pixels where the transitions occur, we produce a suppressed hidden-line display. In Figure 7.1.3.1, rays 4 and 5 bracket the intersection edge of Faces 1 and 2. We turn on the pixel that corresponds to ray 5. (Alternatively, we could have turned on the pixel of ray 4, or done some anti-aliasing average.) Figure 7.1.1.2 was produced precisely by this technique, using the ray casting algorithms of the PADL-2 system.

Photo-realistic images may be obtained by ray casting by using a sufficiently elaborate `Intensity` function. Figure 7.1.1.3 was generated by ray casting on a CSG representation, through a rendering module developed by the Ford Motor Corporation for the PADL-2 system.

Ray casting is a relatively simple and inherently parallel computation, which can be done by using special purpose hardware. Existing experimental systems are capable of rendering

complex objects in about 1 second [Voelcker et al ???]. A drawback of the ray casting approach is its viewpoint dependence. A change of viewpoint requires a complete recomputation. This implies that real-time rotation of shaded images produced by ray casting on CSG is not feasible in the current state of the technology, even with special hardware. In contrast, z-buffering hardware for BReps routinely supports real-time object rotation. As noted earlier, computation of BReps is relatively slow, and therefore BRep-based rendering also is slow when an object is displayed for the first time, because its boundary must be evaluated. Once the BRep is computed, however, BRep display is fast and meets real-time constraints.

### 7.1.4 Graphic Interaction

Defining objects in 3-D by using a 2-D screen is inherently difficult. The vendors of CAD/CAM systems, as well as many researchers, have spent much time and money in the design and implementation of graphic user interfaces (GUIs) for geometric modeling systems. User interfaces are critical for the industrial acceptance of CAD/CAM software. Today's commercial systems have modern interfaces with the look-and-feel users have come to expect, with liberal use of menus, point-and-click, and drag-and-drop operations.

Relative positioning is used to establish the 3-D poses of primitives and sub-objects. Typically, one picks on the screen an entity or set of entities that serve as reference for positioning others. For example, pick a planar face and position another plane parallel to the first at a given offset distance. Defining poses through geometric constraints is intuitive and very convenient.

A good GUI requires two key capabilities: picking reference geometric entities from a screen, and creating other entities in constrained poses with respect to the references. The implementation of graphic picking depends on the primary representation used in a system, and on the type of display normally presented to the user. Most of the existing modeling systems are based on BReps, and display objects as line drawings, with or without suppressed hidden lines. The entities picked in such systems normally are edges. These can be selected by using standard capabilities of graphic packages.

In CSG systems, edges cannot be picked directly because BReps are not available. Instead, picking operations return faces, which can then be used as positional references, or as means of defining edges or vertices, by intersection. Face picking on CSG representations is implemented by casting a ray from the viewpoint to the mouse position, and classifying the ray. Experimental GUIs based on CSG representations and face picking have been demonstrated, and are comparable to their BRep counterparts [Encarnação & Requicha 199?].

Geometric constraint satisfaction is a complicated problem, beyond the scope of this course. For a survey and introduction to the area, see [Hoffmann ???].

## 7.2 Mass Property Calculation

### 7.2.1 Applications and Definitions

Mass properties such as weight and moments of inertia are important in many applications, as the following examples show. The most fundamental equation of dynamics, Newton's equation $F = ma$, requires knowledge of the mass of the moving object. The weight of an

object is critical in aerospace products such as those intended to fly inside satellites. The weight is less critical but still very important for the automobile industry, since it has a direct bearing on fuel consumption.

The center of gravity of an object is essential for stability studies. An object resting on a horizontal plane is stable if a vertical line through the center of gravity intersects the interior of the 2-D convex hull of the points of contact between the object and the plane—see Figure 7.2.1 for a 2-D example, in which the convex hull of the contact points is 1-D.

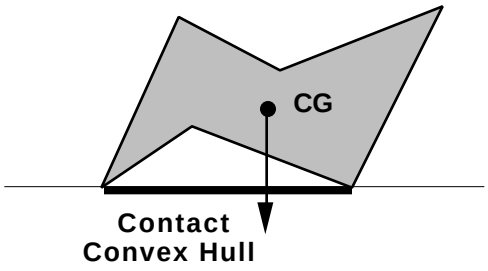


Figure 7.2.1 – Stability check in 2-D

The dynamics of a rotating body is studied through equations of motion in which the moments of inertia play a role analogous to that of mass in Newton's equation. In addition, the moments and principal axes of inertia of an object provide a coarse characterization of its shape and can be used for discrimination in pattern recognition and computer vision. Several object recognition algorithms, especially in 2-D, use moments as one of the features that help distinguish the objects from one another.

Volume, mass, center of gravity, and moments of inertia of homogeneous objects are all defined by integrals of the form

$$I = \int_V f(\mathbf{p})\rho\, dv,$$

where $\mathbf{p}$ is a generic point of the solid $V$, $dv$ is the volume differential, $\rho$ is the density, and $f$ is a polynomial function. The following table shows some examples.

| **Function** $f$ | **Entity** |
|---|---|
| 1 | Mass $M$ |
| $x/M$ | $X$ coordinate of CG |
| $x^2 + y^2$ | Moment of inertia about $Z$ |
| $xy$ | Product of inertia |

There is an area of numerical analysis that is concerned precisely with the calculation of integrals. It is called *numerical integration* or numerical quadrature. However, numerical integration typically addresses the problem of computing integrals in which the domain of integration (the volumetric solid $V$) is geometrically simple and the function $f$ is complicated. In geometric modeling we normally face just the opposite situation of a simple, polynomial $f$, but a complex domain $V$.

Integral evaluation is additive, in the sense that the integral over the union of two disjoint domains $A$ and $B$ is simply the sum of the integrals over $A$ and over $B$. This implies that

mass properties of objects represented by spatial decompositions can be computed by evaluating the contributions of individual cells in the decomposition and adding the results. If the cells have simple geometry, the integrals can be evaluated easily by closed form expressions. As a simple example, let $f = z$, and let $V$ be a unit cube aligned with the principal axes and with left lower back vertex at $\mathbf{p} = (a, b, c)$; then

$$\int_a^{a+1} dx \int_b^{b+1} dy \int_c^{c+1} z\,dz = [(c+1)^2 - c^2]/2 = c + \tfrac{1}{2}.$$

Standard calculus texts show how to find the closed-form expressions for the other integrals of interest for mass-property calculation over cubical domains. Therefore, spatial occupancy enumerations, octrees and other decompositions into cubes or cuboids are especially convenient for mass property calculation.

### 7.2.2 CSG Algorithms

The divide and conquer paradigm we have been exploiting for designing classification algorithms for CSG does not work for mass property calculation. The mass properties of two subtree objects cannot be combined to produce the mass properties of their Boolean combination. For example, the volume of the intersection of two objects cannot be expressed in terms of the volumes of the arguments.

Therefore we must convert CSG into other representations that are more suitable for mass properties. We can evaluate the boundary and then use the BRep algorithms discussed in the next section. But it is easier to convert CSG into an approximate spatial decomposition and then add the contributions of each cell. There are several attractive options [Lee & Requicha 1982]:

- Spatial Enumeration
- Ray Representation
- Octree

To construct an approximate spatial enumeration we classify every point in a dense 3-D grid. If the point is in the solid we associate with it a filled cubical cell. Figure 7.2.2.1 illustrates this conversion procedure in a 2-D example. Instead of generating the points in a regularly-spaced grid it is better to generate them randomly within each corresponding cell, as shown in the figure. Randomness avoids systematic errors. Imagine, for example, that the left boundary of the object in the figure was moved right by, say, one third of a cell width. If the points were on a regular grid, all the cells would classify inside the solid and we would make a relatively large error. With random points, some cells would classify in and others out, giving a better approximation for the volume. This approach is known in the numerical analysis literature as a Monte Carlo procedure. The error associated with a Monte Carlo computation can be estimated by standard statistical techniques. Error estimates are very useful to guide a user in the selection of a suitable level of subdivision for the problem being solved. Smaller cells increase the accuracy of the result, at the expense of increased computation. Unfortunately the complexity of the algorithm increases linearly with the number of cells $N$, whereas the accuracy increases slower, with the square root of $N$ [Lee & Requicha 1982].

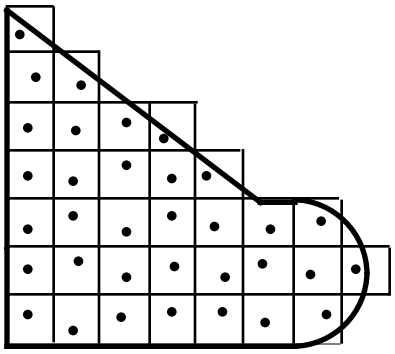

Figure 7.2.2.1 – Spatial enumeration conversion by point classification

For a ray representation we classify parallel rays (i.e. lines) in a 2-D grid to produce a set of columns of square cross-section. Again, it is advantageous to cast rays randomly to avoid systematic errors and to be able to use Monte Carlo error estimates. Figure 7.2.2.2 illustrates the procedure in a 2-D example. Ray representations are much more concise that spatial enumerations, and are considerably faster to compute.

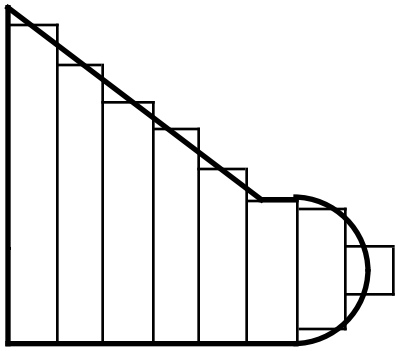

Figure 7.2.2.2 – Ray representation conversion by line classification. For clarity, the classified lines within the columns are not shown.

Let us turn now to CSG to octree conversion. We need to classify entire cells, which may be large. For a large cell, we cannot simply classify the cell center (or a random point in the cell) because there is no guarantee that the whole cell will have the same classification as the selected point. Misclassification of large cells will produce intolerable errors. Classifying the vertices of a cell also is not safe. Unfortunately, to be sure that a cell is entirely inside or outside a solid we have to do something equivalent to intersecting the cell with the solid and evaluating the boundary of the result. This is too expensive for practical use, since many cells have to be tested.

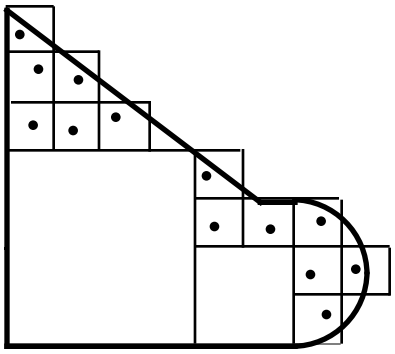

Figure 7.2.2.3 – Octree conversion by cell classification

(To be continued, but not this year...)