Chapter 2

Objects

2.1 Introduction

There are several ways to represent objects in a computer graphics scene. The most common choice is to represent only the surface, and to represent it with a polygon mesh, but there are other approaches, each with their own advantages and disadvantages. Even a polygon mesh may be represented in different ways depending on the application.

2.2 Polygon meshes

A polygon mesh is a simple, yet flexible way of representing a surface. The heart of the representation is a list of *vertex coordinates* for a set of points in space, and a list of polygons that connect the points. The polygons are mostly *triangles*. Polygons with more than three corners can be useful during interactive modelling, but they are almost always split into triangles before rendering.

During transformation and rendering, it is useful to be able to tell the inside from the outside of a polygon, and for that reason, you usually stick to a consistent *winding order* for the vertices that make up a polygon. The most common choice is to traverse the vertices in a counter-clockwise order seen from the outside of the object. If after transformation to camera coordinates the vertices for a polygon are traversed in the opposite direction, that polygon is viewed from the back. This is used in hardware rendering to perform *back face culling*, an operation that can eliminate half of the polygons from further processing and save a considerable amount of work for the renderer. This works for opaque objects, where only the outside is visible. Transparent objects need to be handled differently, but back face culling saves a lot of work where it can be used.

In addition to the vertex coordinates and the triangle list, it is useful to store the *surface normals* at each vertex. To model smoothly curved surfaces, the vertex normal may differ from the true surface normal of the triangle, and it may be different for adjacent triangles sharing the same vertex.

If all you store is a vertex list, a list of normals and a triangle list, what you get is called a *triangle soup*, a data structure that is sufficient for rendering but contains too little information for interactive modelling and progressive level of detail.

To perform modelling operations on a mesh that change its structure, like subdivision or simplification, you need information about *adjacency* in the mesh, to find the answer to questions like:

- Which faces are adjacent to this face?
- Which faces are using this vertex?
- Which edges connect to this vertex?
- Which faces are to either side of this edge?
- Which edges border this face?

In a polygon soup, you would need to search the entire list of faces to answer that kind of questions. To avoid lengthy searches through every element for every operation you perform, some sort of indexing or linking of neighbouring elements is required. A very useful and commonly used data structure for this situation is the *half-edge mesh.* The good ideas behind it are far from obvious, and it wasn't until 1998 that it was first described in scientific literature.

The main idea is that you split each edge into two *directed edges*, one to either side, to make the direction of the edge consistent with the winding order of the polygon on the corresponding side of the edge. The main data structure is the list of half-edges, where for each edge you store pointers to the following information:

- The face it belongs to
- The vertex where it ends
- The next half-edge in the polygon
- The half-edge on the other side of the edge

There are two other lists as well: a list of faces and a list of vertices. For each vertex you store its coordinates, its normal and any other per-vertex data you need, plus a pointer to any one of the half-edges that emanate from

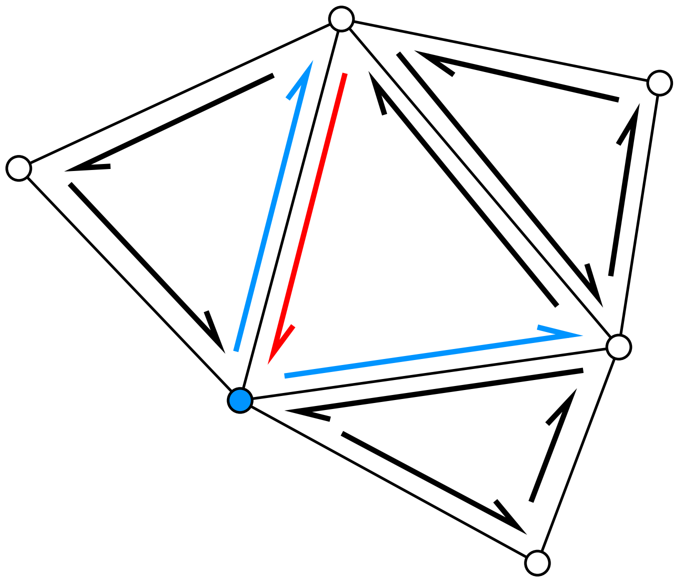

Figure 2.1: The half-edge mesh representation. Each half-edge (highlighted in red in the figure) references the vertex it points to, its opposing half-edge and the next half-edge around the winding order of the polygon (highlighted in blue).

it. For each face, you store only a pointer to any one of the half-edges that border it.

Some variations of the half-edge representation store more data for each edge or for each face to speed up some operations, but the basic data structure described above contains all the information required to both edit and render the mesh in a reasonably efficient manner.

2.3 Subdivision surfaces

Using a structured polygon mesh, such as the half-edge mesh, it is relatively easy to subdivide either the entire mesh or some selection of polygons into several smaller polygons.

Global subdivision schemes, where all polygons in the mesh are subdivided, can be used to create smooth looking models even from relatively jagged, low resolution meshes. This method has seen a lot of use in recent years, because it provides a way of creating objects with smooth, curved surfaces while keeping the ease of use that comes from having a polygon mesh model that is both easy to understand and straightforward to edit. One such subdivision method is the Catmull-Clark method. It deserves special mention because it was the first method that was proposed, the first method that was widely adopted in the graphics industry, and also because it is still in common use. Most modern applications use various improvements and variations of the method to allow for explicit creases and folds and to behave somewhat differently for some special cases, but the differences are small. Some entirely different subdivision methods exist as well, but they are all variations on the same theme: you start with a polygon mesh and create a subdivided mesh with more vertices and more polygons that are computed from the coarser mesh, and the process can be repeated a number of times to create a surface that looks very smooth indeed.

Catmull-Clark subdivision is performed as follows:

1. For each face, add a new *face vertex* at a position which is the average of all original n vertices for the face.

2. For each edge, add an *edge vertex* at the average position of the two neighboring face vertices and the two original endpoints of the edge.

3. For each original vertex P, take the average F of all n recently created face vertices for faces touching P, and take the average R of all n *edge midpoints* for edges touching P, where each edge midpoint is the average of its two endpoint vertices. (Note that this is different from the edge vertex.) *Move each original vertex* to the point $(F + 2R + (n-3)P)/n$. This is a weighted average of the positions of P, R and F with the respective weights $(n-3)/n$, $2/n$ and $1/n$.

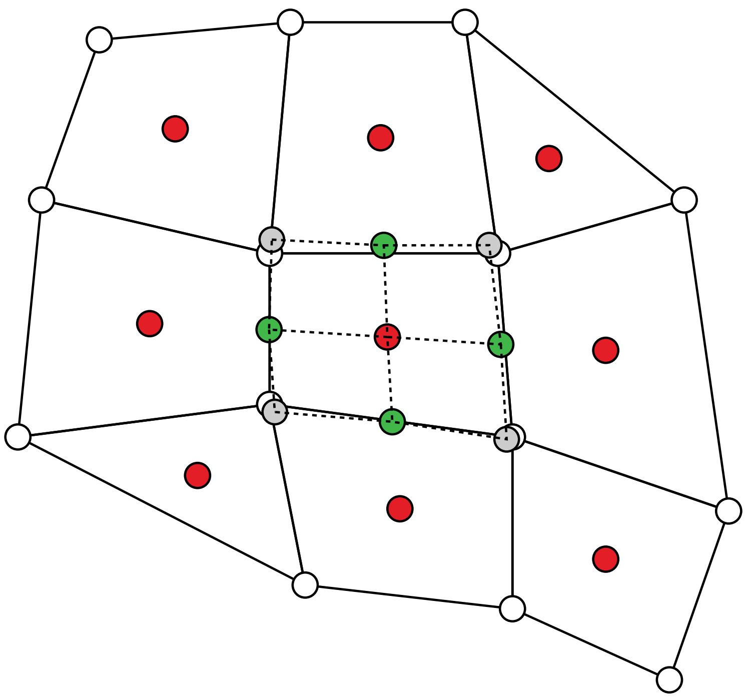

Figure 2.2: Catmull-Clark subdivision. Original mesh: white vertices, solid lines. New face vertices: red. New edge vertices: green. Moved vertices: gray. Subdivision of the center face: dashed lines.

4. Create *new edges* from each newly created face vertex to each of the newly created edge vertices for edges surrounding the face.

5. Define *new faces* as enclosed by the new edges.

The subdivision looks fairly complex when presented like this, but its principle is simple: Create one new vertex in the center of each face, create one new vertex for each edge, move the original vertices a bit and create n new faces for each original face with n vertices. An overview of the process is presented in Figure 2.2.

Because of how Catmull-Clark subdivision is performed, the input mesh can contain arbitrary polygons, but the subdivided mesh will consist only of quadrilaterals, which will typically not be planar. The new mesh will be smoother than the old mesh, because the averaging of neighboring points to create each new point will reduce rapid local variations and round off sharp corners. The smoothing effect is very strong in the first few iterations, but then there are no sharp edges left, and the surface approaches its *limit surface*, an ideal smooth surface which for the Catmull-Clark subdivision scheme is well defined and has a closed form parametric expression.

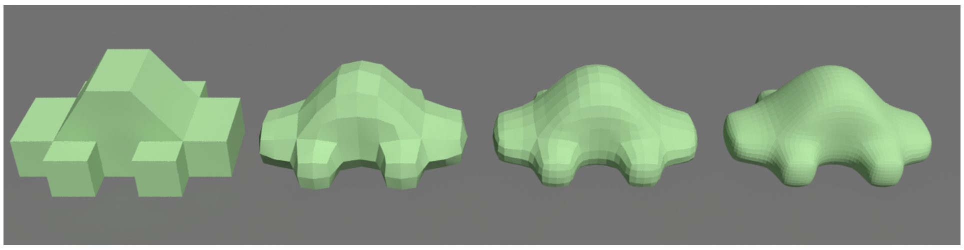

Figure 2.3: From left to right: original mesh, one, two and three iterations of Catmull-Clark subdivision.

Evaluating it directly requires more work than performing a few steps of successive subdivision until the surface looks smooth enough, but merely knowing that there is a well defined limit surface is useful. The limit surface is continuous everywhere, and has a continuous derivative (”is smooth”) everywhere, except at vertices where more or less than four faces meet. These points are called *extraordinary points*.

Creating a coarse, ”boxy” model and relying on automatic subdivision to smooth out the unwanted sharp edges is sometimes called *box modelling*. Figure 2.3 shows an example of Catmull-Clark subdivision applied in several iterations to a coarse polygon model.

Selective subdivision schemes, where only some polygons are subdivided in each step, can be used as an efficient modelling tool, where the user starts out with a coarse mesh to model the overall structure of the object and refines it to model successively finer detail where needed. The process can be organized to maintain the previous mesh in the successive steps of refinement, and make it possible to backtrack to a more coarse model. This is a convenient way of creating models with a built-in dynamic *level of detail* (LOD) for real time rendering. This modelling method is commonly referred to as *hierarchical subdivision surfaces*, or HSDS for short.

2.4 Parametric curves

Curves are one-dimensional shapes, paths through space, which is not the same kind of objects as polygon meshes which describe surfaces in space. However, curves are very important for 2D graphics, where they are used to describe both lines and the outline of closed shapes, and they are heavily used as a tool in 3D graphics to describe the profile of an extruded object or the outline of a lathed object, and perhaps most importantly to specify paths for animation. Parametric curves are also quite a lot easier to explain and to understand than parametric surfaces, and once you understand them, parametric surfaces are really only an extension. Therefore, we begin by describing parametric curves, more specifically an important class of parametric curves called *Béziér curves.*

A parametric curve in two dimensions is defined as a vector-valued function of one parameter:

$$\mathbf{p}(u) = \begin{bmatrix} x(u) \\ y(u) \end{bmatrix} \tag{2.1}$$

In the 3D case, the vector has a third component $z(u)$. The principle is the same, and the description below is valid for both the 2D case and the 3D case.

From calculus, we know that a reasonably well-behaved (infinitely differentiable) function can be approximated by polynomials. Polynomials are easy to compute and has simple derivatives. We choose to create parametric curves where the components, $x(u)$ etc., are polynomials in u. We also choose to restrict the degree of those polynomials to 3. If complicated curves need to be represented, we split them up into several segments and represent each segment with a third degree curve. This is not an obvious choice, but long experience has shown that it is a good one. Third degree curves, *cubic curves*, are used a lot in computer graphics.

For easy interactive drawing, we want to specify the starting point and the endpoint of the curve, like we would for a straight line. We also want some degree of control over the curvature, and we can get that by specifying the *tangent* of the curve. For a third degree curve, specifying the endpoints and the tangent at both endpoints gives us enough constraints to compute the equation for the curve. This also gives the user reasonably close and intuitive control over the curve segment, and provides enough flexibility.

Béziér curves are far from an obvious idea. Skipping some of the motivation for exactly why we do it like this, a Béziér curve is defined by four *control points*, $\mathbf{p}_0$ through $\mathbf{p}_3$, we assume that $0 \leq u \leq 1$, and the constraints for the curve are:

$$\begin{array}{rcl} \mathbf{p}(0) & = & \mathbf{p}_0 \\ \mathbf{p}(1) & = & \mathbf{p}_3 \\ \mathbf{p}'(0) & = & 3(\mathbf{p}_1 - \mathbf{p}_0) \\ \mathbf{p}'(1) & = & 3(\mathbf{p}_3 - \mathbf{p}_2) \end{array} \tag{2.2}$$

In words, this means that the curve starts at $\mathbf{p}_0$ for $u = 0$, ends at $\mathbf{p}_3$ for $u = 1$, has a tangent 3 times the vector from $\mathbf{p}_0$ to $\mathbf{p}_1$ at $u = 0$, and a tangent 3 times the vector from $\mathbf{p}_2$ to $\mathbf{p}_3$ at $u = 1$. Asserting a third degree polynomial for the function and taking its derivative with respect to u to compute the tangent, we get:

$$\begin{array}{rcl} \mathbf{p}(u) & = & \mathbf{A}u^3 + \mathbf{B}u^2 + \mathbf{C}u + \mathbf{D} \\ \mathbf{p}'(u) & = & 3\mathbf{A}u^2 + 2\mathbf{B}u + \mathbf{C} \end{array} \tag{2.3}$$

Putting the constraints into this yields the equation system:

$$\begin{aligned}\mathbf{p}(0) &= \mathbf{D} &&= \mathbf{p}_0\\ \mathbf{p}(1) &= \mathbf{A}+\mathbf{B}+\mathbf{C}+\mathbf{D} &&= \mathbf{p}_3\\ \mathbf{p}'(0) &= \mathbf{C} &&= 3(\mathbf{p}_1-\mathbf{p}_0)\\ \mathbf{p}'(1) &= 3\mathbf{A}+2\mathbf{B}+\mathbf{C} &&= 3(\mathbf{p}_3-\mathbf{p}_2)\end{aligned} \tag{2.4}$$

Solving the equation system gives us:

$$\begin{aligned}\mathbf{A} &= \mathbf{p}_0+3\mathbf{p}_1-3\mathbf{p}_2+\mathbf{p}_3\\ \mathbf{B} &= 3\mathbf{p}_0-6\mathbf{p}_1+3\mathbf{p}_2\\ \mathbf{C} &= -3\mathbf{p}_0+3\mathbf{p}_1\\ \mathbf{D} &= \mathbf{p}_0\end{aligned} \tag{2.5}$$

Putting A, B, C and D into the original equation gives us an equation for the curve segment:

$$\mathbf{p}(u) = (-\mathbf{p}_0+3\mathbf{p}_1-3\mathbf{p}_2+\mathbf{p}_3)u^3+(3\mathbf{p}_0-6\mathbf{p}_1+3\mathbf{p}_2)u^2+(-3\mathbf{p}_0+3\mathbf{p}_1)u+\mathbf{p}_0 \tag{2.6}$$

Rearranging by collecting terms for each of the control points $\mathbf{p}_i$ yields:

$$\mathbf{p}(u) = (-u^3+3u^2-3u+1)\mathbf{p}_0+(3u^3-6u^2+3u)\mathbf{p}_1+(-3u^3+3u^2)\mathbf{p}_2+(u^3)\mathbf{p}_3 \tag{2.7}$$

And, finally, re-writing the polynomials gives us the form that is usually used when presenting Béziér curves:

$$\mathbf{p}(u) = (1-u)^3\mathbf{p}_0+3u(1-u)^2\mathbf{p}_1+3u^2(1-u)\mathbf{p}_2+(u^3)\mathbf{p}_3 \tag{2.8}$$

This can be expressed a lot more compactly as a sum with symbolic names for the polynomials:

$$\mathbf{p}(u) = \sum_{i=0}^{3} B_i(u)\mathbf{p}_i \tag{2.9}$$

where

$$\begin{aligned}B_0(u) &= (1-u)^3\\ B_1(u) &= 3u(1-u)^2\\ B_2(u) &= 3u^2(1-u)\\ B_3(u) &= u^3\end{aligned} \tag{2.10}$$

In this form, the equation can be interpreted as a weighted sum of the control points, with weights being determined by the interpolating functions $B_i(u)$. Figure 2.4 shows these functions on the interval $0 \leq u \leq 1$. Points $\mathbf{p}_0$ and $\mathbf{p}_3$ are the only ones that have non-zero weights at the endpoints

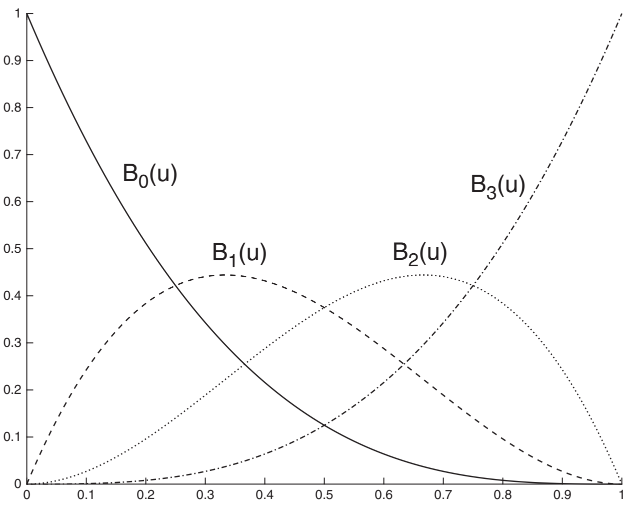

Figure 2.4: Béziér polynomials plotted over the interval $0 \leq u \leq 1$.

of the curve, and the points $\mathbf{p}_1$ and $\mathbf{p}_2$ influence the curve most strongly around the middle of the interval.

Looking at the polynomials, we can see that the sum of all weights is exactly 1 everywhere:

$$\sum_{i=0}^{3} B_i(u) = 1 \tag{2.11}$$

Furthermore, all weights are positive in the interval $0 \leq u \leq 1$. This means that the point $\mathbf{p}(u)$ is always in the *convex hull* of the control points. The convex hull in 2D is the convex quadrilateral that can be drawn with the control points as corners. In 3D, the convex hull is a tetrahedron with the control points as vertices. This makes it possible to create a very simple test during rendering for whether a Béziér curve segment is visible or not, and if it might extend across the border of the rendering window: if the convex hull is entirely inside the window, the entire curve is visible, and if the convex hull is entirely outside the window, the curve is not visible. If the convex hull is partially inside the window, the curve is potentially visible and might need clipping. This convex hull property of a Béziér curve segment is very useful, and it is the main reason why we chose the seemingly arbitrary factor 3 when we set the constraints for the derivatives at each endpoint in Equation 2.2.

2.5 Parametric surfaces

Parametric surfaces are a very wide class of objects. Their fundamental advantage is that they are capable of describing smoothly curved surfaces without approximating them with polygons. By describing, say, a sphere as a set of polygons, you need to decide how many vertices you should create to approximate the smooth surface, but the right choice for the number of vertices depends on many factors, for example the resolution of the output image, how close the camera is to the object, how well lit the sphere is, what kind of material it has, and how much effort you can afford to spend on rendering the sphere. You often need a variable resolution for the mesh model. Parametric surfaces solve this problem by describing not a fixed set of vertices, but a resolution-independent equation for all points on the surface. The partitioning into polygons, the *tessellation* of the surface, can be left for later, and the same object description can be used to create both high and low resolution polygon meshes.

The surface coordinates for a sphere in parametric form could look like this:

$$\mathbf{p}(\varphi,\theta) = \begin{bmatrix} R\cos\varphi\sin\theta \\ R\sin\varphi\sin\theta \\ R\cos\theta \end{bmatrix} + \mathbf{p}_0 \tag{2.12}$$

There are two different kinds of parameters in this equation. The *creation parameters* or the *attributes* are the sphere's radius R and center $\mathbf{p}_0$, and they determine the size and position of the sphere. The *surface parameters* are φ and θ, and by varying those in the ranges $0 \leq \varphi \leq 2\pi$, $0 \leq \theta \leq \pi$ you trace out every point on the surface. Usually, you want parameters in the range 0 to 1 rather than having irrational numbers like π in your loop limits, so a better parametric equation for a sphere would be:

$$\mathbf{p}(u,v) = \begin{bmatrix} R\cos(2\pi u)\sin(\pi v) \\ R\sin(2\pi u)\sin(\pi v) \\ R\cos(\pi v) \end{bmatrix} + \mathbf{p}_0 \tag{2.13}$$

Many useful objects can be described in exact form with reasonably simple equations like this. Spheres, cylinders and a few other simple shapes are important because they are used a lot in manufacturing, and they are often best described in parametric form.

By an extension to the idea behind parametric curves, we can describe a surface with the same kind of interpolating polynomials. A *Béziér patch* can be described by the equation:

$$\mathbf{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i(u)B_j(v)\mathbf{p}_{ij} \tag{2.14}$$

where the polynomials B_i and B_j are the same polynomials as for the Béziér curves, and the set of 4×4 control points $\mathbf{p}_{ij}$ form a control mesh. The interpretation of the control points is similar to that for curves, but not quite as straightforward, and a Béziér patch is not quite as easy to use as a modelling tool as a Béziér curve is. Other kinds of parametric surface patches exist which have a different interpretation of their control points and which are easier to manipulate during modelling. The most popular variant are so-called *NURBS surfaces*. NURBS is short for *non-uniform rational B-spline*. It would take us a bit too far into theory to explain them in full here, but they are fundamentally quite similar to Béziér surfaces. With an understanding of Béziér curves and Béziér patches it is not a lot of work to learn more about NURBS patches, as well as any other kind of parametric curve or surface based on control points and interpolating functions.

2.6 Swept surfaces

Real world objects which are manufactured by extrusion or lathe can be described by a 2-D *profile* which is then swept along a line or in a circle around a rotation axis. A common name for this kind of surfaces, both in mathematics and graphics, is *swept surfaces*. The profile is usually specified as a parametric curve. When generating polygon mesh objects with this method, the number of steps along each of the directions (along the profile and along the sweep path) can be specified independently to control the detail and complexity of the generated mesh. The sweep path can also be an arbitrary curve, not just a line or a circle, and the profile can change gradually across the sweep. Two swept surfaces are shown in Figure 2.5: the base of the column is a surface of revolution (a profile swept along a circle), and the column is a linear extrusion (a profile swept along a straight line).

2.7 Implicit surfaces

The object representations presented above have all been *explicit*, in the sense that they store or compute the coordinates for points on the surface. A common way of representing shapes in mathematics is *implicit* equations, which work the other way around: an equation tells whether a certain point is on the surface, but there is no direct and obvious way of generating only those points that are on the surface. Continuing the example with a sphere, for which the parametric equation was given in Equation 2.13, an implicit equation for the same sphere would be:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = R^2 \tag{2.15}$$

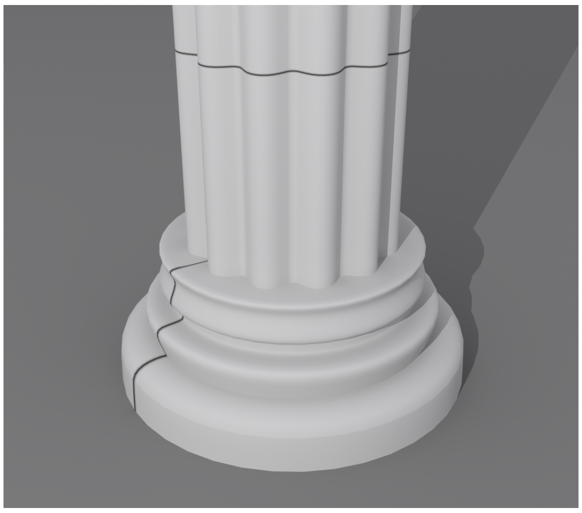

Figure 2.5: Two swept surfaces. The profiles are indicated by the black stripes.

Another, more common formulation is to define a function $F(x, y, z)$ that is zero on the surface:

$$F(x, y, z) = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2 = 0 \tag{2.16}$$

This formulation has the advantage that the sign of the function indicates whether a point is inside or outside the surface. A negative sign for F means that the point (x, y, z) is inside the sphere, and a positive sign means that the point is outside. The gradient of the function points away from the object, and the gradient at the surface is parallel to the surface normal. The value of the function can be used to compute the distance to the surface, which can be a very useful piece of information for computations in simulation and rendering. The implicit form also makes it considerably easier to compute the intersection between a ray and the surface, a computation which is essential to the various rendering methods based on ray tracing.

Some rendering methods, like ray tracing, are very well suited to rendering implicit surfaces directly without subdividing them into polygon meshes. For real time rendering, an implicit equation is not a convenient representation of a surface, but it can be very useful as a complement where available, for example to perform collision checks for animation.

A special case of implicit surfaces is called *metaballs* or *metaparticles.* Their implicit function F is defined as a sum of functions that depend on the distance from the current point $\mathbf{p}$ to a set of points or *particles*, $\mathbf{p}_i$:

$$F(\mathbf{p}) = \sum_i D(|\mathbf{p} - \mathbf{p}_i|) - T \tag{2.17}$$

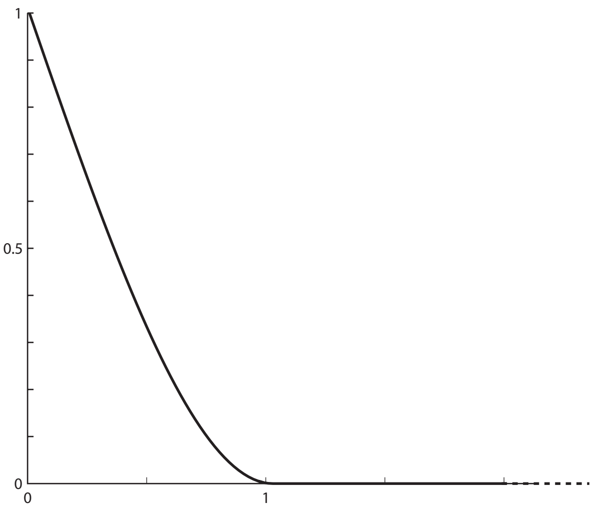

Figure 2.6: A suitable distance function for a metaballs object

where T is some threshold value and D is some distance function that typically starts out at 1, has has a small region of linear decline near the origin and drops to zero beyond a certain distance, like the function in Figure 2.6.

The metaball surface is defined as the set of points where $F = 0$, like other implicit surfaces. The difference is that F is computed by a sum of several smaller contributions from nearby particles, making it possible to model objects of very irregular shape. Metaball surfaces can be used to represent water and other liquids, which are often simulated as particles of relatively large size rather than at a microscopic level. The "correct" level for a water simulation would be to simulate individual water molecules, but that is way too much work, enough to be out of reach for the foreseeable future. Water simulations on today's computers may involve millions, even billions of particles, but in real life, even a single glass of water contains around 10^{25} molecules.

2.8 Constructive Solid Geometry

The term *Constructive Solid Geometry*, or CSG for short, refers to a kind of compound objects which are defined as *set operations* between other objects. The set operations are union, difference and intersection, and their meanings can be explained by a simple diagram, see Figure 2.7. The *union* of two objects is the volume covered by either of the objects, Their *intersection* is the volume covered by both of the objects, and the *subtraction*, one object minus another, is the volume covered by the first object that is also not covered by the second object. All these operations require that both

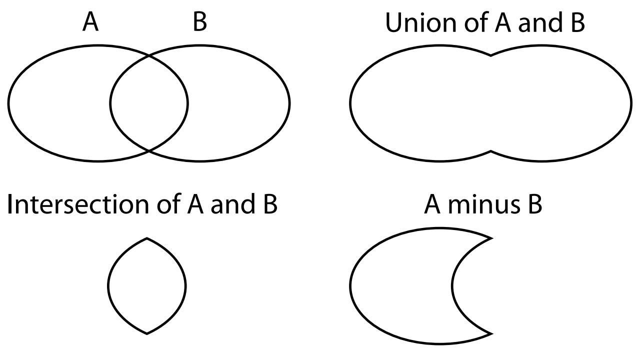

Figure 2.7: Set operations for CSG

objects have a closed surface, such that they have a well defined "inside" and "outside".

CSG operations are very useful for replicating many operations used in the manufacture of real world objects, like cutting, drilling and welding. CSG modeling is often used in computer aided design (CAD), but it can be a useful tool in all kinds of 3-D modeling.

This kind of modeling is particularly simple to implement for implicit objects. Determining whether a point is inside the compound object is just a matter of checking for all its components and using simple logic, e.g. "inside the union of A and B" is equivalent to "inside A or inside B".

Most 3-D modeling software tools can perform CSG operations also between polygon meshes. The algorithms for such operations are quite tricky to figure out and to implement, because you need to compute lots of intersection lines between polygons, cut the models up to create new vertices and new edges along the intersections, and make sure the new compound object is assembled correctly.

2.9 Volumetric objects

Implicit surfaces cross the boundary to a full-fledged *volumetric* representation of objects, seeing how they use a function F that is defined for all points in space, not only points on the surface. Some objects do not have a clearly defined surface, like smoke, fog and fire, and some that do have a surface are transparent with a varying density on the inside. To represent such objects well, we need a volumetric function $F(x, y, z)$. Sometimes you can find a mathematical function that does the job, and that is a viable solution for objects with a simple internal structure or some random-looking things like clouds and fire. However, there are many cases where such functions may be unreasonably hard or even impossible to find. Therefore, volumetric data is often stored as *sampled volumes*, similarly to digital images. The concept of a *pixel* which represents an intensity sample in 2D image space is carried over to *voxels* which typically represent a density sample in 3D space.

Sampled density volumes are routinely collected in the field of medical imaging, by means of X-ray tomographs and magnetic resonance cameras. Visualizing such density volumes is one important application of computer graphics, and how to do it well is a very active research topic. The amount of data represented by a sampled voxel volume is a problem even for modern computers. While an image of resolution 1000×1000 pixels requires only a few megabytes of storage, a volume of resolution $1000 \times 1000 \times 1000$ voxels require several gigabytes. One single volume pretty much fills all the available memory in a modern computer, and handling several gigabytes of data efficiently for real time rendering is tricky. Storing, transmitting and archiving lots of data volumes also present real problems for the current generation of computers.