```
P6              -- magic number
# comment       -- any number of comment lines starting with
200 300         -- image width & height
255             -- max color value
```

FIGURE 35.15 PPM P6 file header block layout.

(a) Red cube on a midgrey (0.5 0.5 0.5) background

```
000000  5036 0a33 3030 2032 3030 0a32 3535 0a7f  P6!300 200!255!!
000010   7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f   !!!!!!!!!!!!!!!!
*

009870  886d 6d92 5959 8a69 6984 7676 817d 7d80  'mm'YY'ii'vv'}}'
009880   7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f   !!!!!!!!!!!!!!!!
*

009b10  7f80 7e7e 9d41 41b5 0809 b211 11a9 2424  !!~~!AA!!!!!!$$
009c00  9f3b 3b94 5454 8b68 6886 7272 827b 7b80  !;;!TT!hh!rr!{{!
009c10   7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f   !!!!!!!!!!!!!!!!
*

009f70  7f7f 7f7f 7f82 7979 a72b 2bb9 0000 ba00  !!!!!yy!++!!!!!
009f80  00ba 0000 b801 01b7 0606 b201 01af 1d1d  !!!!!!!!!!!!!!!!
009f90  a532 3297 4d4d 8d62 6286 7272 827a 7a80  !22!MM!bb!rr!zz!
009fa0  7e7e 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  ~~!!!!!!!!!!!!!!
009fb0   7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f   !!!!!!!!!!!!!!!!
*

00a210  7f7f 7f7f 7f7f 7f7f 7f8a 6969 b014 14ba  !!!!!!!!!!ii!!!!
00a300  0000 ba00 00ba 0000 ba00 00ba 0000 ba00  !!!!!!!!!!!!!!!!
00a310  00ba 0000 ba00 00b9 0505 b60d 0daf 1d1d  !!!!!!!!!!!!!!!!
00a320  a62f 219d 4141 915b 5b88 6d6d 8279 7980  !//'AA![['mm!yy'
00a330  7e7e 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  ~~!!!!!!!!!!!!!!
00a340   7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f   !!!!!!!!!!!!!!!!
*
```

(b) Dump of start of PPM P6 red-cube image file

FIGURE 35.16 Red cube image and corresponding PPM P6 image file data: (a) red cube on a mid-gray (0.5 0.5 0.5) background, (b) dump of start of PPM P6 red-cube image file.

scanline and the number of scanlines. The maximum color value cannot exceed 255, but it may be less if less than 8 bits of color information per primary are available.

The PPM P6 data block begins with the first pixel of the top scanline of the image (upper left-hand corner), and pixel data are stored in scanline order from left to right in 3-byte chunks giving the R, G, B values for each pixel, encoded as binary numbers. There is no separator between scanlines, and none is needed, as the image width given in the header block exactly determines the number of pixels per scanline. Figure 35.16a shows a red cube on a mid-gray background, and Figure 35.16b gives the first several lines of a hexadecimal dump of the contents of the corresponding PPM file. Each line of this dump has the hexadecimal byte count on the left, followed by 16 bytes of hexadecimal information from the file, and ends with the same 16 bytes of information displayed in ASCII (nonprinting characters are displayed as !). Except for the first line of the dump, which contains the file header information, the ASCII

information is meaningless, because the image data in the file are binary encoded. A line in the dump containing only a $*$ indicates a sequence of lines all containing exactly the same information as the line above.

35.3 Research Issues and Summary

In this chapter, we have taken a quick, broad-brush look at 3-D graphics systems, from scene specification to image display and storage. The attempt has been to lay a basic foundation and to provide certain essential details necessary for further study. A practical example of the implementation and use of a 3-D graphics system appears in Chapter 43.

As research in graphics tends to be specialized, readers are directed to the research issues and summary sections of Chapter 37 through Chapter 42 of this handbook for information on important and interesting open research areas. However, we note here some broad areas of research that are both timely and important to the development of the field. In the area of rendering, there are two areas that seem to be generating much current interest: extending solutions to the global illumination problem to handle a wider variety of material types and developing nonphotorealistic techniques. The entire fields of virtual reality and volumetric modeling are just getting off the ground and promise to be strong research areas for many years. Within the subfield of computer animation, three areas are of very strong current interest. These are physically based modeling and simulation [Barzel 1992], artificial-intelligence and artificial-life techniques for character animation and choreography, and higher-order interactive techniques that exploit the capabilities of new 3-D position and motion tracking devices. Finally, in the area of modeling, there is much room for improvement in interactive modeling tools and techniques, again possibly exploiting 3-D position and motion trackers. And there is a continuing search for compact, powerful ways to represent natural forms.

Defining Terms

Affine transformation: A coordinate-system transformation where each transformed coordinate is a linear sum of the three original coordinates plus a constant.

Bidirectional reflectance distribution function: A function defined for a reflecting material that, given arrival and departure directions, gives the fraction of light energy of a particular wavelength arriving at a surface from the arrival direction that is reflected from the surface in the departure direction.

Bump map: A pattern of surface normal displacements, simulating the undulations of a bumpy surface, that is to be mapped onto a geometric surface during rendering.

Center of interest: A point in space toward which the virtual camera is always aimed.

Center of projection: The point in space at which all rays of projection for a perspective projection converge. This often is considered to be the position of the virtual camera.

Displacement map: A pattern of surface position displacements to create the undulations of a bumpy surface that is to be mapped onto a geometric surface during rendering.

Global coordinate system: A coordinate system with respect to which all other coordinate systems in the definition of a 3-D scene are defined.

Hierarchical model: A geometric model defined within a set of nested reference coordinate frames, whose references form a directed tree or acyclic graph from a set of leaf frames, where basic geometric objects are defined, to a root frame in which the entire scene is defined.

Homogeneous coordinate system: In a 3-D graphics system, this is a 4-D coordinate system into which points are transformed before being multiplied by a homogeneous transformation matrix. The system is called homogeneous because all 3-D points lie on the same hyperplane perpendicular to the fourth coordinate axis. Typically, this coordinate is 1 for all points.

Implicit surface: A surface defined implicitly as the set of points satisfying an equation of the form $F(x, y, z) = 0$.

Infinite light: A light source taken to be infinitely far from the model being illuminated, so that all of its rays reaching the model can be considered to be parallel.

Local coordinate system: A coordinate system defined with respect to some reference coordinate system, usually used to define a part or subassembly within a scene definition.

Material: The collective set of properties of the surface of an object that determines how it will reflect or transmit light.

Parametric surface: A surface defined explicitly by a set of functions of the form $X(\mathbf{u}), Y(\mathbf{u}), Z(\mathbf{u})$ that returns (x, y, z) coordinates of a point on the surface as a function of the set of parameters $\mathbf{u}$. Most commonly, $\mathbf{u} = (u, v)$, which yields a biparametric surface parametrized by the parametric coordinates u and v.

Pixel: A square or rectangular uniformly colored area on a raster display that forms the basic unit or picture element of a digital image.

Point light: A light source that radiates light uniformly in all directions from a single geometric point in space.

Projection: A transformation typically from a higher-dimensional space to a lower-dimensional space. In computer graphics the most commonly used projection is the camera projection, which projects 3-D scene geometry onto the plane of a 2-D virtual screen, as one of the key steps in the rendering process.

Raster: An array of scanlines, painted across a CRT screen, which taken together form a rectangular 2-D image. Often the term raster is used to refer to the 2-D array of pixel values stored digitally in a framebuffer.

Surface normal: A vector perpendicular to the tangent plane to a surface at a point. If the surface is planar, the three coefficients of the surface normal vector are the three scaling coefficients of the plane equation.

Texture map: A pattern of color to be mapped onto a geometric surface during rendering.

Transformation matrix: For a 3-D system, this is a 4×4 matrix that, when multiplied by a point in homogeneous coordinates, gives the coordinates of the point in a transformed homogeneous coordinate system. The 4-D homogeneous form of the matrix allows the unification of 3-D translation, rotation, scaling, and shear into one operator.

References

Bartels, R., Beatty, J., and Barsky, B. 1987. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann, Los Altos, CA.

Barzel, R. 1992. *Physically-Based Modeling for Computer Graphics*. Academic Press, San Diego.

Ebert, D. S., ed. 1994. *Texturing and Modeling: A Procedural Approach*. AP Professional, Boston.

Foley, J. and van Dam, A. 1982. *Fundamentals of Interactive Computer Graphics*. Addison–Wesley, Reading, MA.

Foley, J., van Dam, A., Feiner, S., and Hughes, J. 1990. *Computer Graphics Principles and Practice*. Addison–Wesley, Reading, MA.

Glassner, A., ed. 1990. *Graphics Gems*. Academic Press, San Diego.

Glassner, A. 1995. *Principles of Digital Image Synthesis*. Morgan Kaufmann, San Francisco.

Gonzalez, R. and Woods, R. 1992. *Digital Image Processing*. Addison–Wesley, Reading, MA.

Hill, F. 1990. *Computer Graphics*. Macmillan, New York.

Murray, J. and vanRyper, W. 1994. *Encyclopedia of Graphics File Formats*. O'Reilly, Sebastopol, CA.

Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. 1992. *Numerical Recipes in C, The Art of Scientific Computing*, 2nd ed. Cambridge University Press, Cambridge.

Rogers, D. 1985. *Procedural Elements of Computer Graphics*. McGraw–Hill, New York.

Rogers, D. and Adams, J. 1990. *Mathematical Elements of Computer Graphics*. McGraw–Hill, New York.

Russ, J. 1992. *The Image Processing Handbook*. CRC Press, Boca Raton, FL.

Watt, A. and Watt, M. 1992. *Advanced Animation and Rendering Techniques*. Addison–Wesley, Reading, MA.

Wolberg, G. 1990. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, CA.

Further Information

The reader seeking further information on three-dimensional graphics systems should refer first to Chapters 36 through 43 of this Handbook, which provide much of the detail that this overview intentionally skips. The Further Information sections of these chapters provide pointers to the best source books on each of the specialized topics covered.

Beyond this volume, the primary source book for a broad coverage of the field is *Computer Graphics Principles and Practice* by Foley, van Dam, Feiner, and Hughes, published by Addison–Wesley. For a host of practical information and implementation tips, the five-volume series *Graphics Gems*, published by Academic Press, is an invaluable source. Information on image file formats can be found in the *Encyclopedia of Graphics File Formats* by Murray and vanRyper, published by O'Reilly. Fine practical guides to image-processing techniques are *The Image Processing Handbook* by Russ, published by CRC Press, and *Digital Image Processing* by Gonzalez and Woods, published by Addison–Wesley. The mathematics of computer graphics is given a very lucid treatment in *Mathematical Elements of Computer Graphics* by Rogers and Adams, published by McGraw–Hill, and there is no better reference to practical approaches to the implementation of numerical algorithms than *Numerical Recipes* (in various computer-language editions) by Press, Teukolsky, Vetterling, and Flannery, published by Cambridge University Press. Finally, the recent two-volume set *Principles of Digital Image Synthesis* by Glassner, published by Morgan Kaufmann, provides an excellent comprehensive coverage of the theoretical groundings of the field.

Persons interested in keeping up with the latest research in the field should turn to the ACM SIGGRAPH Conference Proceedings, published each year as a special issue of the ACM journal *Computer Graphics*. Other important conferences with published proceedings are *Eurographics* sponsored by the European Association for Computer Graphics, *Graphics Interface* sponsored by the Canadian Human–Computer Communications Society, and *Computer Graphics International* sponsored by the Computer Graphics Society. The IEEE journal *Computer Graphics and Applications* provides an applications-oriented view of recent developments, as well as publishing news and articles of general interest. ACM's *Transactions on Graphics* carries significant research papers, often with a focus on geometric modeling. Other important journals include *The Visual Computer*, IEEE's *Transactions on Visualization and Computer Graphics*, and the *Journal of Visualization and Computer Animation*.

36

Geometric Primitives

Alyn P. Rockwood
Colorado School of Mines

36.1 Introduction

Geometric primitives are rudimentary for creating the sophisticated objects seen in computer graphics. They provide uniformity and standardization in addition to enabling hardware support.

Initially, definition of geometric primitives was driven by the capabilities of the hardware. Only simple primitives were available, e.g., points, line segments, triangles. In addition to the hardware constraints, other driving forces in the development of a geometric primitive have been either its general applicability to a broad range of needs or its satisfying ad hoc, but useful applications. The triangular facet is an example of a primitive that is simple to generate, easy to support in hardware, and widely used to model many graphics objects. An example of a specific primitive can be drawn from flight simulation in the case of *light strings*, which are instances of variable-intensity, directional points of light used to model airport and city lights at night. It is not a common primitive, but it is supported by a critical and profitable application.

As hardware and CPU increased in capability, the sophistication of the primitives grew as well. The primitives became somewhat less dependent on hardware; software primitives became more common, although for raw speed hardware primitives still dominate.

One direction for the sophistication of graphics primitives has been in the geometric order of the primitive. Initially, primitives were discrete or first-order approximations of objects, that is, collections of points, lines, and planar polygons. This has been extended to higher-order primitives represented by polynomial or rational curves and surfaces in any dimension.

The other direction for the sophistication of primitives has been in attributes that are assigned to the geometry. Color, transparency, associated bitmaps and textures, surface normals, and labels are examples of attributes attached to a primitive and used in its display.

This summary of graphics primitives is in rough chronological order of development which basically corresponds to increasing complexity. It concentrates on common primitives. It is beyond the scope of this review to include anything but occasional allusions to the plentiful special-purpose developments.

36.2 Screen Specification

To locate the graphics primitive in the viewing window, a local coordinate system is defined. By convention the origin is at the bottom left corner of the window. The positive x-axis extends horizontally from it, while the positive y-axis extends vertically. For 3-D objects, the z-axis is imagined to extend into the screen away from the viewer. In the 3-D case, it is necessary to transform the object to the screen via a set of viewing transformations (see Chapter 35).

Unlike pen and paper, we cannot draw a straight line between two points. The screen is a discrete grid; individual pixels must be illuminated in some pattern to indicate the desired line segment or other graphics primitive. A screen has from 80 to 120 pixels per inch, with high-resolution screens exhibiting 300 per inch. Most screens are about 1024 pixels wide by 780 pixels high. The problem of rendering a graphics primitive on a raster screen is called *scan conversion* and is discussed in Chapter 38. It is mentioned because it is closely related to the geometry of the object drawn and related drawing attributes. It is the scan conversion method that is embedded in hardware to accelerate the display of the graphics in a system. The expense and efficacy of graphics hardware depend on careful selection of the primitives for the facility desired.

36.3 Simple Primitives

36.3.1 Text

There are two standard ways to represent textual characters for graphics purposes. The first method is to save a representation of the letter as a bitmap, called a *font cache*. This method allows fast reproduction of the character on screen. Usually the font cache has more resolution than needed, and the character is downsampled to the display pixels. Even on high-resolution devices such as a quality laser printer, the discrete nature of the bitmap can be apparent, creating jagged edges. When transformations are applied to bitmaps such as rotations or shearing, aliasing problems can also be apparent. See the example in Figure 36.1.

To improve the quality of transformed characters and to compress the amount of data needed to transfer text, a second method of representing characters was developed using polygons or curves. When the text is displayed, the curves or polygons are scan-converted; thus the quality is constant regardless of the transformation (Figure 36.2). The transformation is applied to the curve or polygon basis before scan conversion. Postscript™ is a well-known product for text transferal. In a "Postscript" printer, for instance, the definition of the fonts resides in the printer where it is scan-converted. Transfer across the network requires only a few parameters to describe font size, type, style, etc. Those printers which do not have resident Postscript databases and interpreters must transfer bitmaps with resulting loss of quality and time.

Postscript is based on parametrically defined curves called Bezier curves (see Section 36.8).

Fonts are designed in the bitmap case by simply scanning script from existing print, while special font design programs exist to design fonts with curves.

FIGURE 36.1 Bitmap characters. Note jagged edges and sampling artifacts. Compare to Figure 36.2.

FIGURE 36.2 Curved representation of characters scan-converted.

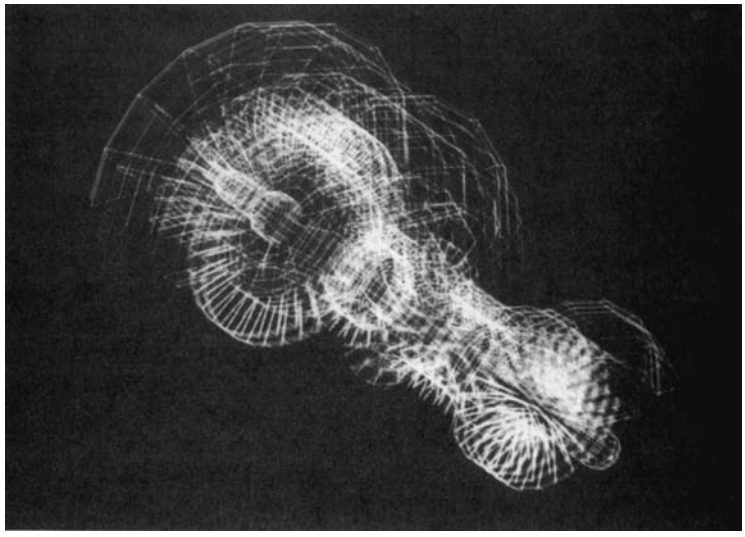


FIGURE 36.3 A polyline display of a turbine engine (Created on an Evans and Sutherland Multi-Picture System in 1986).

36.3.2 Lines and Polylines

The first and still one of the most prevalent of the graphics primitives is the *line* (actually a line segment), typically specified by "move to $(x1, y1)$" and then "draw to $(x2, y2)$" commands, where $(x1, y1)$ and $(x2, y2)$ are the end points of the line segments. The endpoints can also be given as 3-D points and then projected to the screen through viewing transforms (see Chapter 35).

The line can be easily extended to a *polyline* graphics primitive, that is to a piecewise polygonal path. It is usually stored as an array of 2-D or 3-D points. The first element of the array tells how many points are in the polyline. The following array elements are then the arguments of the line drawing commands. The first entry invokes the "move to" command; successive array entries use the "draw to" command as an operator.

Attributes for lines and polylines are used to modify their appearance without changing position. Popular attributes include *thickness, color*, and *style* (i.e., continuous, dotted, dashed, and variable dash lengths). In more advanced systems each vertex of a polyline is associated to a color and the line segment is drawn by linearly blending the one color to the other along the segment. This is called *color blending*.

Attributes are set either by extending the array to include attribute fields (polylines), or they are set *modally* (lines and polylines); that is, there is an independent command that sets the mode of the line drawing until changed explicitly by another command. Default values control the attributes that are not explicitly set. "Continuous" is the default for style, for example.

Impressive pictures can be produced from these simple line-based primitives. Figure 36.3 shows a turbine engine done with polylines.

Because of the discrete, pixel-based nature of the raster display screen, lines are prone to alias (see Chapter 39); that is, they may break apart or merge with one another to form distracting moire patterns.

They are also susceptible to jagged edges, a signature of older graphics displays. Serious line-drawing systems found it therefore important to add hardware that would "antialias" their lines while drawing them. See Chapter 39 for more details. Figure 36.3 used hardware antialiasing.

A *marker* is another primitive; it is either a point or a small square, triangle, or circle, often placed at the vertices of a polyline to indicate specific details such as distinguishing between lines of a chart. Markers are themselves usually made by predetermined lines or polylines. This unifies the display technology for the hardware, making them faster to draw. Even the point is often a very short line. A single command is then used to center the marker. Style attributes don't usually apply to markers, but color and thickness attributes can be used to advantage.

36.3.3 Elliptical Arcs

Elliptical arcs in 2-D may be specified in many equivalent ways. One way is to give the center position, major and minor axes, and start and end angles. Both angles are given counterclockwise from the major axis. Elliptical arcs may have all the attributes given to lines.

Such an arc may be a closed figure if the angles are properly chosen. In this case, it makes sense to have the ability to *fill* the ellipse with a given pattern or color using a scan conversion algorithm. Even in the case of partial arcs, the object is closed by a line between the end points of the arc so that it may be filled, if wished.

In 3-D the plane of the elliptical arc must also be specified by the unit normal of the plane. While 2-D arcs are common, 3-D arcs are limited to high-end systems that can justify the cost of the hardware. Viewing transforms must compute the arc that is the image on the screen and then scan-convert that arc (usually elliptical, since perspective takes conics to conics).

It should be mentioned that arcs can also be represented by a polyline with enough segments; thus a software primitive for the arc which induces the properly segmented and positioned polyline may be a cost-effective *macro* for elliptical arcs. This macro should consider the effects of zoom and perspective to avoid revealing the underlying polygonality of the arc. It is surprising how small a number of line segments, properly chosen, can give the impression of a smooth arc.

One of the most commonly used elliptical arcs is, of course, the circular arc and its closure, the circle.

36.4 Wireframes

Given hardware or software macro arc primitives in a system, complex curves can be generated by piecing the arcs and lines end to end. Several computer aided design (CAD) systems exist that can generate a rich set of line-based models in both 2-D and 3-D. They are called **wireframe** models. Figure 36.3 is a wireframe model of a turbine engine.

Wireframe models are popular in engineering applications, for instance, because of their visual precision. Drafting is also a natural application. They have other advantages in 3-D because of the ability to see through them to parts of the object that are behind.

Too many lines can be confusing, however; to improve wireframe models a hidden-line routine may be employed (see Chapter 38). This requires derivation of a surface implied by the line. Yet line models can be ambiguous. Figure 36.4 shows a classical example of an object for which the implied surface can be legitimately interpreted in many ways.

Finally, wireframe objects do not support realism. Most objects have a surface that is colored and reflects light according to physical laws of irradiance. This leads us to the next type of primitive.

36.5 Polygons

Closing a polyline by matching start and end points creates a **polygon**. It is probably the most commonly employed primitive in graphics, because it is easy to define an associated surface. Not all polygons have surfaces (they may be a primitive used in wireframe modeling), but most systems that admit polygons will

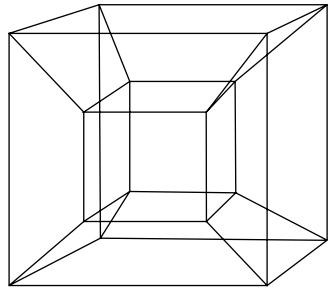


FIGURE 36.4 An ambiguous wireframe model. Where do the surfaces belong?

support both *filled* and *empty* polygons. Filling a polygon is an important example of scan conversion. There are many ways to do this (see Chapter 38). The different ways to fill become attributes of the polygon. The filled polygon is the basis for the numerous hidden-surface algorithms (see Chapter 38). Because of their usefulness for defining surfaces, polygons almost always appear as 3-D primitives which subsume the 2-D case.

The most sophisticated polygon primitive allows nonconvex polygons that contain other polygons, called *holes* or *islands* depending on whether they are filled or not. The scan conversion routine selectively fills the appropriate portions of the polygon depending on whether they are holes or islands. This complex polygon is probably made as a macro out of simple polygon primitives. Triangulation routines exist, for example, that reduce such polygons to simple triangular facets.

36.6 The Triangular Facet

If the polygon is the most popular primitive, then the simple triangle is certainly the most popular polygon. Figure 36.5 shows an object simple object composed of triangular facets.

As mentioned, the triangular facet often supports more complex polygons. It is fast to draw and supports many diverse and powerful methods (see Chapter 38). Another major advantage is that they are always flat; they do not leave the plane due to numerical problems, data errors, or nonplane-preserving transformations. At this writing, graphics workstations exist that can render 10 million triangular facets per second with hidden surfaces removed and smooth shaded display between neighboring triangles implemented. It is certain that this number will continue to increase. With such capacity several dozens of the objects in Figure 36.5 could be smoothly animated.

In order to increase the speed of polygon rendering and decrease the size of the database, triangular and quadrilateral facets can be stored and processed as meshes; i.e., collections of facets that share edges and vertices. The triangular mesh, for instance, is given by defining three vertices for the "lead" triangle, and then giving a new vertex for each successive triangle (Figure 36.6). The succeeding triangles use the last two vertices in the list with the newly given one to form the next triangle. Such a mesh contains about one-third the data and uses the shared edges to increase processing speed. Most graphics objects have large contiguous areas that can take advantage of meshing. The quadrilateral mesh requires two additional vertices be used, with the last two given, and therefore has less savings.

Because of its benefits and ubiquity, many other primitives are defined in software as configurations of the triangular facet. One example is the faceted sphere. With enough facets the sphere will look smooth. It is very commonly encountered in engineering applications and molecular modeling.

Most of the attributes associated with polygons, and triangular facets in particular, deal with the rendering of the facet (see Chapter 38), e.g., color blending between vertices, pointers into a bitmap for texturing, normal vectors (from an underlying surface), reflectance parameters, transparency, and color.

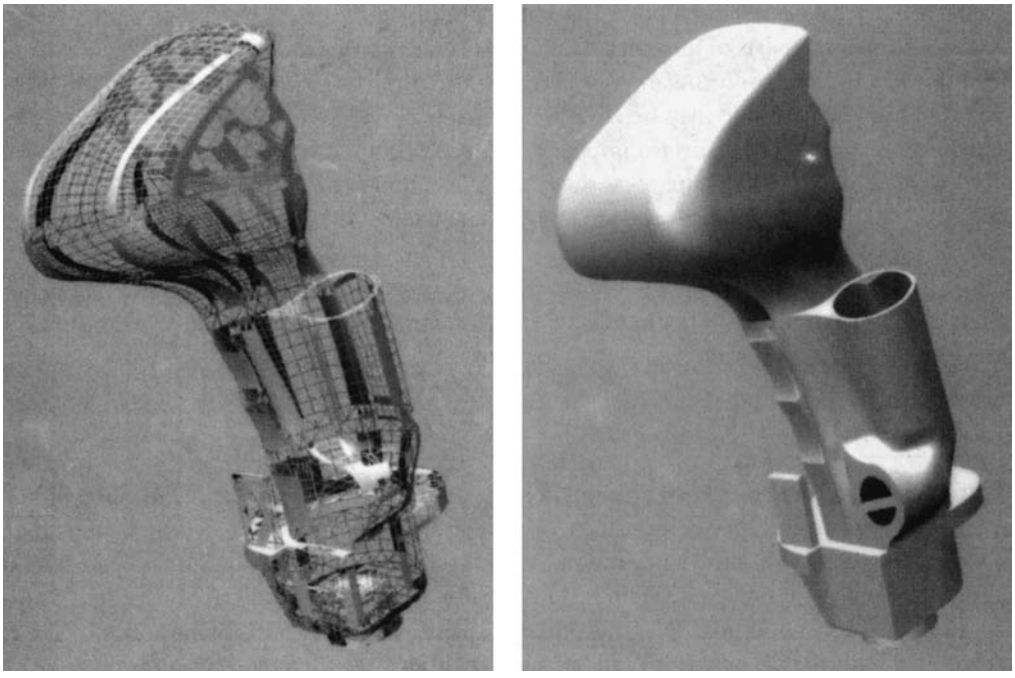


FIGURE 36.5 (See Plate 36.5 in color insert following page **29**-22.) A model composed of simple triangular facets (polygon model).

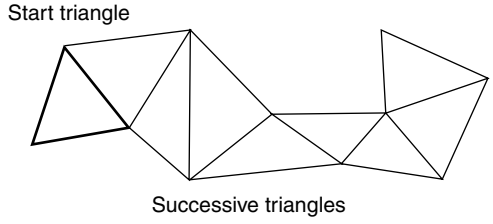


FIGURE 36.6 A mesh of triangular facets.

36.7 Implicit Modeling

36.7.1 Implicit Primitives

The need to model mechanical parts drove the definition of implicitly defined primitives. These modeling primitives naturally became graphics primitives to serve the design industry. They have since been used in other fields such as animation.

An implicit function f maps a point $\mathbf{x}$ in $\mathbf{R}^n$ to a real number, i.e., $f(\mathbf{x}) = r$. Usually $n = 3$. By absorbing r into the function we can view the implicit function as mapping to zero, i.e., $g(\mathbf{x}) = f(\mathbf{x}) - r = 0$. The importance of implicit functions in modeling is that the function divides space into three parts: $f(\mathbf{x}) < 0$, $f(\mathbf{x}) > 0$, and $f(\mathbf{x}) = 0$. The last case defines the surface of an **implicit object**. The other two cases define respectively the *inside* and *outside* of the implicit object. Hence implicit objects are useful in *solid modeling* where it is necessary to distinguish the inside and outside of an object. Modeling objects by polygons does not, for instance, distinguish between the inside and outside of the object. It is, in fact, quite possible to describe an object in which inside and outside are ambiguous. This facility to determine inside and outside

is further enhanced by using Boolean operations on the implicit objects to define more complex objects (see the subsection on CSG objects below).

Another advantage of implicit objects is that the surface normal of an implicitly defined surface is given simply as the gradient of the function: $N(\mathbf{x}) = \nabla f(\mathbf{x})$. Furthermore, many common engineering surfaces are easily given as implicit surfaces; thus the *plane* (not the polygon) is defined by $\mathbf{n} \cdot \mathbf{x} + \mathbf{d} = 0$, where $\mathbf{n}$ is the surface normal and $\mathbf{d}$ is the perpendicular displacement of the plane to the origin. The *ellipsoid* is defined by $(x/a)^2 + (y/b)^2 + (z/c)^2 - r^2 = 0$, where $\mathbf{x} = (x, y, z)$. General *quadrics*, which include ellipsoids, paraboloids, and hyperboloids, are defined by $\mathbf{x}^T M\mathbf{x} + \mathbf{b} \cdot \mathbf{x} + \mathbf{d} = 0$, where M is a 3-by-3 matrix and $\mathbf{b}$ and $\mathbf{d}$ vectors in $\mathbf{R}^3$. The quadrics include such important forms as the cylinder, cone, sphere, paraboloids, and hyperboloids of one and two sheets. Other implicit forms used are the *torus, blends* (transition surfaces between other surfaces), and *superellipsoids* defined by $(x/a)^k + (y/b)^k + (z/c)^k - R = 0$ for any integer k.

36.7.2 CSG Objects

An important extension to implicit modeling arises from applying set operations such as union, intersection, and difference to the sets defined by implicit objects. The intersection of six half spaces defines a cube, for example. This method of modeling is called *constructive solid geometry* (CSG) [Foley 1990]. All set operations can be reduced to some combination of just union, intersection, and complementation. Because these create an algebra on the sets that is isomorphic to Boolean algebra, corresponding to multiply, add, and negate, respectively, the operations are often referred to as *Booleans*.

A convenient form for visualizing and storing a CSG object is to use a tree in which the nodes are implicit objects and the branches indicate the operation. Traversal of the tree indicates the order of the binary operations and to which sets they pertain. Figure 36.7 shows a CSG tree for a simple model.

Figure 36.8 demonstrates an object made exclusively from Boolean parts of plane quadrics, a part torus, and blended transition surfaces. For any point in space it is straightforward to determine whether it is

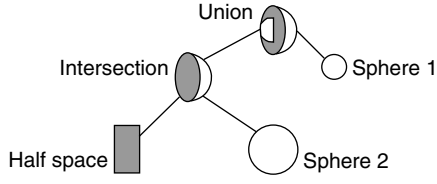


FIGURE 36.7 A CSG model and its tree.

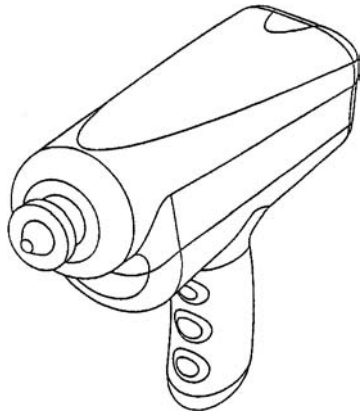


FIGURE 36.8 A solid model of a drill.

inside, outside, or on the surface of the object. This is important in determining volume, center of mass, and moments and for performing Boolean operations needed by CSG models.

Unfortunately, implicit forms tend to be difficult to render, except for ray tracing. Algorithms for polygonizing implicits and CSG models tend to be quite complex [Bloomenthal 1988]. The implicit object gives information about a point relative to the surface in space, but no information is given as to where on the surface a point is located; there is no local coordinate system. This makes it difficult to tessellate into rendering elements such as triangular facets. In the case of ray tracing, however, the parametric form of a ray $\mathbf{x}(t) = (x(t), y(t), z(t))$ composed with implicit function $f(\mathbf{x}(t)) = 0$ leads to a root-finding solution of the intersection points on the surface which are critical points in the ray-tracing algorithm. Determining whether points are part of a CSG model is simply exclusion testing on the CSG tree.

36.8 Parametric Curves

An important class of geometric primitives are formed by **parametrically defined curves and surfaces**. These constitute a flexible set of modeling primitives that are locally parametrized; thus in space the curve is given by $\mathbf{x}(t) = (x(t), y(t), z(t))$ and the surface by $\mathbf{s}(u, v) = (x(u, v), y(u, v), z(u, v))$. In this section and the next one, we will give examples of only the most popular types of parametric curves and surfaces. There are many variations on the parametric forms (see Farin [1992]).

36.8.1 Bezier Curves

The general form of a Bezier curve of degree n is

$$\mathbf{f}(t) = \sum_{i=0}^{n} \mathbf{b}_i B_i^n(t) \tag{36.1}$$

where $\mathbf{b}_i$ are vector coefficients, the *control points*, and

$$B_i(t) = \binom{n}{i} t^i (1 - t^{n-i})$$

where $\binom{n}{i}$ is the binomial coefficient. The $B_i^n(t)$ are called *Bernstein functions*. They form a basis for the set of polynomials of degree n. Bezier curves have a number of characteristics which are derived from the Bernstein functions and which define their behavior.

End-point interpolation: The Bezier curve interpolates the first and last control points, $\mathbf{b}_0$ and $\mathbf{b}_n$. In terms of the interpolation parameter t, $\mathbf{f}(0) = \mathbf{b}_0$ and $\mathbf{f}(1) = \mathbf{b}_n$.

Tangent conditions: The Bezier curve is cotangent to the first and last segments of the *control polygon* (defined by connecting the control points) at the first and last control points; specifically

$$\mathbf{f}'(0) = (\mathbf{b}_1 - \mathbf{b}_0)n \quad \text{and} \quad \mathbf{f}'(1) = (\mathbf{b}_n - \mathbf{b}_{n-1})n$$

Convex hull: The Bezier curve is contained in the convex hull of its control points for $0 \leq t \leq 1$.

Affine invariance: The Bezier curve is affinely invariant with respect to its control points. This means that any linear transformation or translation of the control points defines a new curve which is just the transformation or translation of the original curve.

Variation diminishing: The Bezier curve does not undulate any more than its control polygon; it may undulate less.

Linear precision: The Bezier curve has linear precision: If all the control points form a straight line, the curve also forms a line.

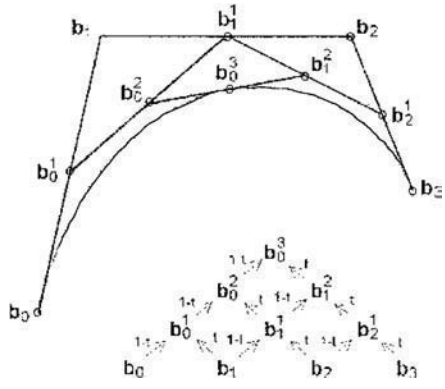


FIGURE 36.9 Bezier curve, control polygon, and de Casteljau algorithm.

Figure 36.9 shows a Bezier curve with its control polygon. Notice how it follows the general shape. This together with the other properties makes it desirable for shape design.

Evaluation of the Bezier curve function at a given value t produces a point $\mathbf{f}(t)$. As t varies from 0 to 1, the point $\mathbf{f}(t)$ traces out the curve segment. One way to evaluate Equation 36.1 is by direct substitution. This is probably the worst way. There are several better methods available for evaluating the Bezier curve. One method is the *de Casteljau algorithm*. This method not only provides a general, relatively fast and robust algorithm, but it gives insight into the behavior of Bezier curves and leads to several important operations on the curves.

To formalize de Casteljau's algorithm we need to use a recursive scheme. The control points are the input. Thereafter each point is superscripted by its level of recursion. Finally, for any point,

$$\mathbf{b}_i^j(t) = (1-t)\mathbf{b}_i^{j-1} + t\mathbf{b}_{i+1}^{j-1} \quad \text{for } i = 0, \ldots, n, \quad j = 0, \ldots, n-i$$

Note that $\mathbf{b}_0^n(t) = \mathbf{f}(t)$; it is a point on the curve.

One of the most important devices for evaluating curves is the systolic array. It is a triangular arrangement of vectors in which each row reflects the levels of recursion of the de Casteljau algorithm. The first row consists of the Bezier control points. Each successive row corresponds to the points produced by iterating with de Casteljau's algorithm.

The point $\mathbf{b}_0^3$ is the point on the curve for some value of the parameter t. Any point in the systolic array may be computed by linearly interpolating the two points in the preceding row with the parameter t; thus for example:

$$\mathbf{b}_1^2 = \mathbf{b}_1^1(1-t) + \mathbf{b}_2^1 t$$

One of the most important operations on a curve is that of subdividing it. The de Casteljau algorithm not only evaluates a point on the curve, it also subdivides a curve into two parts as a bonus. The control points of the two new curves are given as the legs of the systolic array. In Figure 36.10 is a cubic Bezier curve after three iterations of the de Casteljau algorithm, with the parameter $t = 0.5$. By using the left and right legs of the systolic array as control points, we obtain two separate Bezier curves which together replicate the original. We have subdivided the curve at $t = 0.5$.

Subdivision permits existing designs to be refined and modified; for example, by incorporating additional curves into an object. One method of intersecting a Bezier curve with a line is to recursively subdivide the curve, testing for intersections of the curve's control polygons with the line. This process is continued until a sufficiently fine intersection is attained.

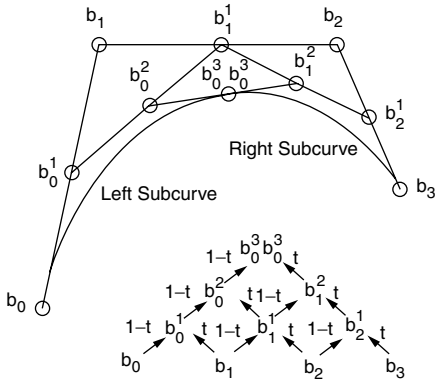


FIGURE 36.10 Subdividing the curve.

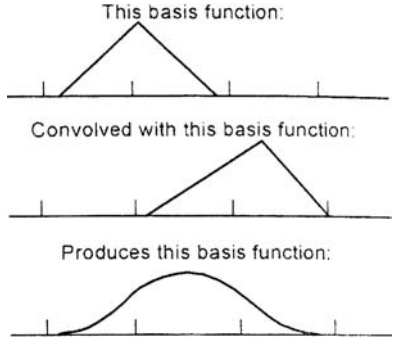


FIGURE 36.11 Defining basis functions by convolution.

36.8.2 B-Spline Curves

A single *B-spline* curve segment is defined much like a Bezier curve. It looks like

$$\mathbf{d}(t) = \sum_{i=0}^{n} \mathbf{d}_i N_i(t)$$

where $\mathbf{d}_i$ are control points, called *de Boor points*. The $N_i(t)$ are the basis functions of the B-spline curve. The degree of the curve is n.

Note that the basis functions used here are different from the Bernstein basis polynomials. Schoenberg first introduced the B-spline in 1949. He defined the basis functions using integral convolution (the "B" in B-spline stands for "basis"). Higher-degree basis functions are given by convolving multiple basis functions of one degree lower. Linear basis functions are just "tents" as shown in Figure 36.11. When convolved together they make piecewise parabolic "bell" curves.

The tent basis function (which has a degree of one) is non-zero over two intervals, the parabola is nonzero over three intervals, and so forth. This gives the region of influence for different degree B-spline control points. Notice also that each convolution results in higher-order continuity between segments of the basis function. When the control points (de Boor points) are weighted by these basis functions, the B-spline curve results.

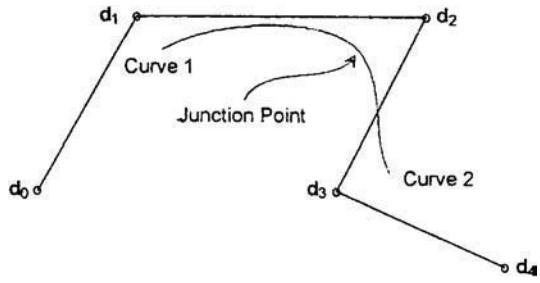


FIGURE 36.12 B-spline curve with two segments.

The major advantage of the B-spline form is in piecing curves together to form a *spline*. If two B-spline curve segments share $n-1$ control points, they will fit together at the junction point with C^{n-1} continuity. The picture in Figure 36.12 shows the case for a cubic B-spline with two segments:

- The first curve has control points $\mathbf{d}_0$, $\mathbf{d}_1$, $\mathbf{d}_2$, and $\mathbf{d}_3$.
- The second curve has control points $\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{d}_3$, and $\mathbf{d}_4$.

Instead of integrating to evaluate the basis functions, a recursive formula has been derived:

$$N_i^n(u) = \frac{u - u_{i-1}}{u_{i+n-1} - u_{i-1}} N_i^{n-1}(u) + \frac{u_{i+n} - u}{u_{i+n} - u_i} N_{i+1}^{n-1}(u)$$

where

$$N_i^0(u) = \begin{cases} 1 & \text{if } u_{i-1} \le u \le u_i \\ 0 & \text{otherwise} \end{cases}$$

The terms in u represent a *knot sequence*, the spans over which the de Boor points influence the B-spline.

More control points can be added to make a longer and more elaborate spline curve. As seen in Figure 36.12 neighboring curve segments share n control points.

It can be seen that for any parameter value u only four basis functions are nonzero; thus only four control points affect the curve at u. If a control point is moved it influences only a limited portion of the curve. This *local support* property is important for modeling.

If the first n and last n control points are made to correspond, then the curve's end points will match; it will form a closed curve. This is called a *periodic B-spline*.

Any point on the curve is a convex combination of the control points, i.e., it must be in the convex hull of the control points associated with the nonzero basis functions.

Like the Bezier curve, the B-spline curve also satisfies a variation-diminishing property, and is affinely invariant. Linear precision follows, as in the Bezier-curve case, from the convex-hull property.

The recursive form for evaluating B-splines via basis functions is seldom used in practice. The best way to evaluate a B-spline curve is to use the de Boor algorithm.

Formally, the de Boor algorithm is written as

$$\mathbf{d}_i^k(u) = \frac{u_{i+n-k} - u}{u_{i+n-k} - u_{i-1}} \mathbf{d}_{i-1}^{k-1}(u) + \frac{u - u_{i-1}}{u_{i+n-k} - u_{i-1}} \mathbf{d}_i^{k-1}(u)$$

We see that the de Boor points form a systolic array; each point is defined in terms of preceding points. Thus we may write an iterative procedure to evaluate a point on a B-spline curve in much the same way as de Casteljau's algorithm above evaluates a point on a Bezier curve. Only the weighting factors differ. The last point produced in the method is the point on the curve.

36.9 Parametric Surfaces

36.9.1 Bezier Surfaces

Imagine moving the control points of the Bezier curve in three dimensions. As they move in space, new curves are generated. If they are moved smoothly, then the curves formed create a surface, which may be thought of as a bundle of curves. If each of the control points is moved along a Bezier curve of its own, then a Bezier surface patch is created; if a B-spline curve is extruded, then a B-spline surface results (Figure 36.13).

In the Bezier case this can be written by changing the control points in the Bezier formula into Bezier curves; thus a surface is defined by

$$\mathbf{s}(u,v) = \sum_{i=0}^{n} \mathbf{b}_i(u) B_i(v) \tag{36.2}$$

Notice that we have one parameter for the control curves and one for the "swept" curve. It is convenient to write the control curves as Bezier curves of the same degree. If we let the ith control curve have control points $\mathbf{b}_{ij}$, then the surface given in Equation 36.2 above can be written as

$$\mathbf{s}(u,v) = \sum_{i=0}^{n} \left(\sum_{j=0}^{m} \mathbf{b}_{ij} B_j(u) \right) B_i(v) \tag{36.3}$$

where m is the degree of the control curves.

A surface can always be thought of as nesting one set of curves inside another. From this simple characteristic we derive many properties and operations for surfaces. Simple algebra changes Equation 36.3 above into

$$\mathbf{s}(u,v) = \sum_{i=0}^{n} \left(\sum_{j=0}^{m} \mathbf{b}_{ij} B_j(u) \right) B_i(v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{b}_{ij} B_i(v) B_j(u) \tag{36.4}$$

That is, even though we started with one curve and swept it along the other, there is no preferred direction.

The surface patch could have been written as:

$$\mathbf{s}(u,v) = \sum_{i=0}^{n} \mathbf{b}_j(v) B_j(u), \qquad \text{where} \quad \mathbf{b}_j(v) = \sum_{i=0}^{n} \mathbf{b}_{ij} B_i(v)$$

The curve is simply swept in the other direction.

The set of control points forms a rectangular control mesh. A 3-by-3 (bicubic) control mesh is shown in Figure 36.14 with the surface. There are 16 control points in the bicubic control mesh. In general there

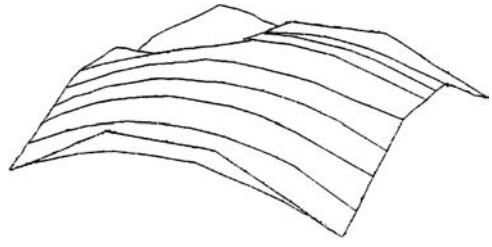


FIGURE 36.13 Sweeping a curve to make the surface.

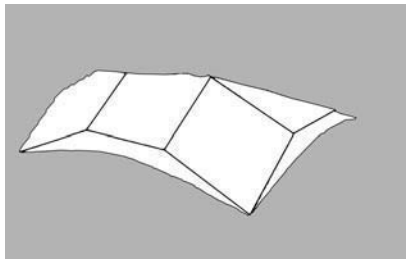


FIGURE 36.14 Bicubic surface with control mesh.

will be $(n + 1) \times (m + 1)$ control points. By convention we associate the i-index with the u-parameter, and the j-index with the v-parameter. Hence if we take:

$$\mathbf{b}_{i0}, \quad i = 1, \ldots, n$$

we get the Bezier curve

$$\mathbf{b}(u, 0) = \sum_{i=0}^{n} \mathbf{b}_{i0} B_i^n(u)$$

Each marginal set of control points defines a Bezier curve (the four border curves), and each of these curves is a boundary of the Bezier surface patch. Such a surface is shown in white in Figure 36.14.

Many of the properties of the Bezier surface are derived directly from those of the Bezier curve, especially those curves that form the boundaries of the patch:

End-point interpolation: The Bezier surface patch passes through all four corner control points.

Tangent conditions: The four border curves of the Bezier surface patch are cotangent to the first and last segments of each border control polygon, at the first and last control points. The normal to the surface patch at each vertex may be found from the cross product of the tangents.

Convex hull: The Bezier surface patch is contained in the convex hull of its control mesh for $0 < u < 1$ and $0 < v < 1$.

Affine invariance: The Bezier surface patch is affinely invariant with respect to its control mesh. This means that any linear transformation or translation of the control mesh defines a new patch which is just the transformation or translation of the original patch.

36.9.1.1 Evaluation of the Bezier Surface Patch

As with the properties described above, the evaluation of a Bezier surface patch can also be derived from the Bezier curve. If we want to evaluate a point on the patch at parameter value (u, v), we apply the de Casteljau algorithm in a nested fashion to Equation 36.3. That is, we first evaluate the control curves in the u direction, which reduce to control points in the v direction. These points are again evaluated with de Casteljau's algorithm.

36.9.1.2 Subdivision of the Bezier Surface Patch

Again, as with the Bezier curve, we can apply the de Casteljau algorithm in nested fashion to subdivide a Bezier surface patch. When a surface patch is subdivided, it yields four subpatches that share a corner at the (u, v) subdivision point. Recall that when a curve was subdivided, the new curve's control points appeared as the legs of a systolic array. In the surface case we subdivide each row of the control mesh, producing points of the systolic array for each. Each point on each leg of every row's systolic array now becomes a control point for a columnar set. A biquadratic case is shown in Figure 36.15. Here we see that three points in each row produce five after subdivision. Now we consider the points in columns, subdividing the

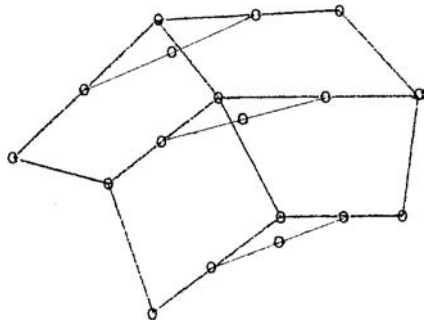


FIGURE 36.15 First-level subdivision.

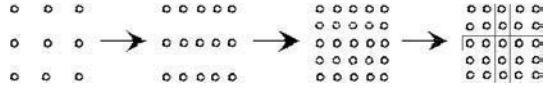


FIGURE 36.16 Control points: progression for subdivision.

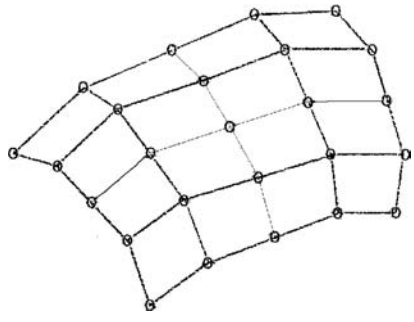


FIGURE 36.17 The subdivided patch.

columns with de Casteljau's algorithm. The points in the legs of their systolic arrays become the control points of the new subpatches. In our example rows with three control points produce five "leg" points, i.e., five columns of three points. Each column then produces five control points, so a 3-by-3 grid generates a 5-by-5 grid after subdivision. The control meshes of the four new patches are produced as shown in Figure 36.16. The central row and column of control points are shared by each 3-by-3 subpatch as shown in Figure 36.17. Note that the order of the scheme does not matter. Columns might have been taken first, and then rows.

Subdivision is a basic operation on surfaces. Many "divide and conquer" algorithms are based on it. To clip a surface to a viewing window we can use the convex-hull property and subdivision, for example. The convex-hull test can determine if a patch is entirely contained in the window. If not, it is subdivided, and the subpatches are then tested. Recursion is applied until the pixel level.

36.9.2 B-Spline Surfaces

As with the Bezier surface, the B-spline surface is defined as a nested bundle of curves, thus yielding

$$\mathbf{s}(u,v) = \sum_{i=0}^{L+n-1} \sum_{j=0}^{M+m-1} \mathbf{d}_{ij} N_i(u) N_j(v)$$

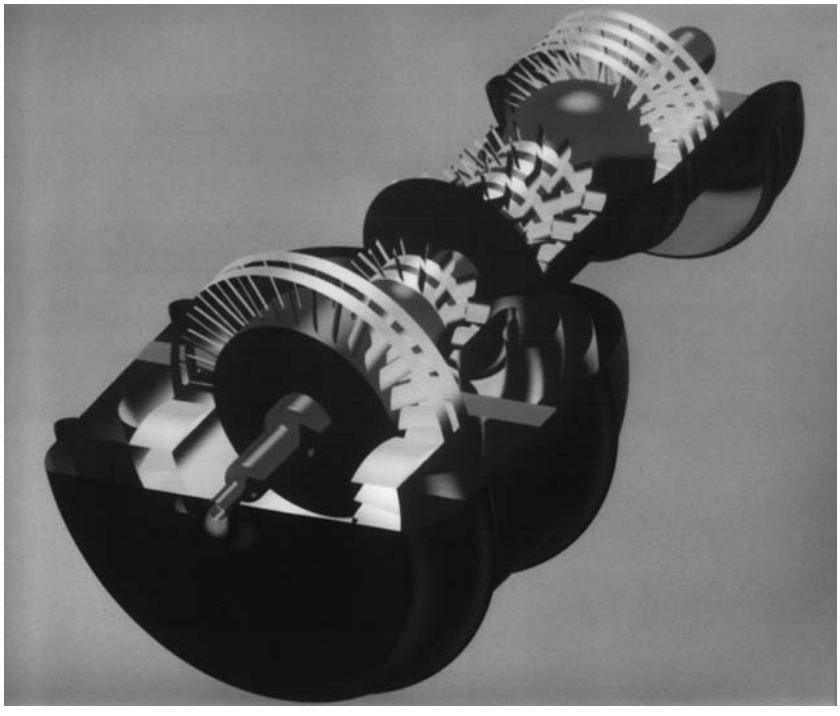


FIGURE 36.18 (See Plate 36.18 in color insert following page **29**-22.) Turbine engine modeled by B-spline surfaces.

where

n, m are the degrees of the B-splines,
L, M are the number of segments, so there are $L \times M$ patches.

All operations used for B-spline curves carry over to the surface via the nesting scheme implied by its definition. We recall that B-spline curves are especially convenient for obtaining continuity between polynomial segments. This convenience is even stronger in the case of B-spline surfaces; the patches meet with higher-order continuity depending on the degree of the respective basis functions. B-splines define quilts of patches with "local support" design flexibility. Finally, since B-spline curves are more compact to store than Bezier curves, the advantage is "squared" in the case of surfaces.

These advantages are tempered by the fact that operations are typically more efficient on Bezier curves. Conventional wisdom says that it is best to design and represent surfaces as B-splines, and then convert to Bezier form for operations. Figure 36.18 shows an object made of many surface patches.

36.10 Standards

Several movements have occurred to standardize primitives. Some of the standards grew out of the mechanical CAD/CAM industry, because of the obvious connection between it and graphics. IGES is a popular one. Although it cannot be said that IGES has become commonplace in the graphics world, it is quite often important to translate between graphics models and these standards when dealing with CAD/CAM applications. GKS [ANSI 1985], PHIGS, and PHIGS+ [ANSI 1988] are standards developed by a broad consortium of academics and industrial users. In spite of major efforts and backing, they have not been as directly successful as the standards evolved in the industry. They have, however, been an intellectual force that has influenced many of the industrial efforts.

In industry each company has developed and pushed for its particular set of graphics standards. Perhaps the most successful at this time is GL (for Graphics Library), which was developed by Silicon Graphics,

Inc., a company which based its computer workstation product on high-powered graphics. It has been licensed by IBM and many other companies. It has evolved into Open GL, which is supported by many manufacturers and threatens to become the standard.

36.11 Research Issues and Summary

There have been efforts to cast higher-order primitives like parametric surfaces into graphics hardware, but the best approach seems to be to convert these into polygons and then render [Rockwood et al. 1990]. There may be hardware support for this process, but the polygon processing remains at the heart of graphics primitives. This trend is likely to continue into the future if for no other reason than its own inertia. Special needs will continue to drive the development of specialized primitives.

One new trend that may affect development is that of volume rendering (see Chapter 41). Although it is currently quite expensive to render, hardware improvements and cheaper memory costs should make it increasingly more viable. As a technique it subsumes many of the current methods, usually with better quality, as well as enabling the visualization of volume-based objects. Volume-based primitives, i.e., tetrahedra, cuboids, and curvilinear volume cubes, will receive more attention and research.

Defining Terms

Implicit objects: Defined by implicit functions, they define solid objects of which outside and inside can be distinguished. Common engineering forms such as the plane, cylinder, sphere, torus, etc. are defined simply by implicit functions.

Parametrically defined curves and surfaces: Higher-order surface primitives used widely in industrial design and graphics. Parametric surfaces such as B-spline and Bezier surfaces have flexible shape attributes and convenient mathematical representations.

Polygon: A closed object consisting of vertices, lines, and usually an interior. When pieced together it gives a piecewise (planar) approximation of objects with a surface. Triangular facets are the most common form and form the basis of most graphics primitives.

Wireframe: Simplest and earliest form of graphics model, consisting of line segments and possibly elliptical arcs that suggest the shape of an object. It is fast to display and has advantages in precision and "see through" features.

References

Adobe Systems Inc. 1985. *Postscript Language Reference Manual*. Addison–Wesley, Reading, MA.

ANSI (American National Standards Institute). 1985. *American National Standards for Information Processing Systems — Computer Graphics — Graphical Kernel System (GKS) Functional Description*. ANSI X3.124-1985. ANSI, New York.

ANSI (American National Standards Institute). 1988. *American National Standards for Information Processing Systems — Programmer's Hierarchical Interactive Graphics Systems (PHIGS) Functional Description*. ANSI X3.144-1988. ANSI, New York.

Bezier, P. 1974. Mathematical and practical possibilities of UNISURF. In *Computer Aided Geometric Design*, R. E. Barnhill and R. Riesenfeld, Eds. Academic Press, New York.

Bloomenthal, J. 1988. Polygonisation of implicit surfaces. *Comput. Aided Geom. Des.* 5:341–345.

Boehm, W., Farin, G., and Kahman, J. 1984. A survey of curve and surface methods in CAGD. *Comput. Aided Geom. Des.* 1(1):1–60, July.

Farin, G. 1992. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, New York.

Faux, I. D. and Pratt, M. J. 1979. *Computational Geometry for Design and Manufacture*. Wiley, New York.

Foley, J. D. 1990. *Computer Graphics: Principles and Practice*. Addison–Wesley, Reading, MA.

Rockwood, A., Davis, T., and Heaton, K. 1990. *Real Time Rendering of Trimmed Surfaces*, Computer Graphics 23,3.

37

Advanced Geometric Modeling

David S. Ebert
Purdue University

37.1 Introduction

Geometric modeling techniques in computer graphics have evolved significantly as the field matured and attempted to portray the complexities of nature. Earlier geometric models, such as polygonal models, patches, points, and lines, are insufficient to represent the complexities of natural objects and intricate man-made objects in a manageable and controllable fashion. Higher-level modeling techniques have been developed to provide an *abstraction* of the model, encode classes of objects, and allow high-level control and specification of the models. Most of these advanced modeling techniques can be considered *procedural* modeling techniques: code segments or algorithms are used to abstract and encode the details of the model, instead of explicitly storing vast numbers of low-level primitives. The use of algorithms unburdens the modeler/animator from low-level control, provides great flexibility, and allows amplification of their efforts through parametric control: a few parameters to the model yield large amounts of geometric detail (Smith [1984] referred to this as *database amplification*). This amplification allows a savings in storage of data and user specification time. The modeler has the flexibility to capture the *essence* of the object or phenomena being modeled without being constrained by the laws of physics and nature. He can include as much physical accuracy, and also as much artistic expression, as he wishes in the model.

This survey examines several types of procedural advanced geometric modeling techniques, including **fractals, grammar-based models, volumetric procedural models, implicit surfaces**, and **particle systems**. Most of these techniques are used to model natural objects and phenomena because the inherent complexity of nature renders traditional modeling techniques impractical. These techniques can also be classified into **surface-based modeling** techniques and **volumetric modeling** techniques. Fractals, grammar-based models, and implicit surfaces* are surface-based modeling techniques. Volumetric procedural models and particle systems are volumetric modeling techniques.

*Implicit surfaces are rendered as surfaces, although the actual model is volumetric.

37.2 Fractals

Fractals and chaos theory have rapidly grown in popularity since the early 1960s [Peitgen et al. 1992]. Mathematicians in the late 19th century and early 20th century, including Cantor, Sierpiński, and von Koch, "discovered" fractal mathematics, but considered these formulas to be "mathematical monsters" that defied normal mathematical principles. Benoit Mandelbrot, who coined the term *fractal*, was the first person to realize that these mathematical formulas were a geometry for describing nature.

Fractals [Peitgen et al. 1992] have a precise mathematical definition, but in computer graphics their definition has been extended to generally refer to models with a large degree of *self-similarity:* subpieces of the object appear to be scaled down, possibly translated and rotated versions of the original object.* Along these lines, Musgrave [Ebert et al. 2002] defines a fractal as "a geometrically complex object, the complexity of which arises through the repetition of form over some range of scale." Many natural objects exhibit this characteristic, including mountains, coastlines, trees, plants (e.g., cauliflower), water, and clouds. In describing fractals, the amount of "roughness," "detail," or amount of space filled by the fractal can be mathematically characterized by its *fractal dimension* (self-similarity dimension), D. The fractal dimension is related to the common integer dimensionality of geometric objects: a line is 1-D, a plane is 2-D, a sphere is 3-D. Fractal objects have noninteger dimensionality. An easy way to explain fractal dimension is to define it in terms of the recursive subdivision technique usually used to create simple fractals. If the original object is subdivided into a pieces using a reduction factor of s, the dimension D is related by the power law [Peitgen et al. 1992]

$$a = \frac{1}{s^D}$$

which yields

$$D = \frac{\log a}{\log (1/s)}$$

Normal geometric objects produce fractal (self-similarity) dimensions that are integers. Fractals produce noninteger, fractional, fractal dimensions. The following examples will illustrate this. If a cube is subdivided into 27 equal pieces, each one is scaled down by a factor of one third, yielding

$$D = \frac{\log 27}{\log \left(1/\frac{1}{3}\right)} = \frac{\log 27}{\log 3} = 3$$

Conversely, a fractal object such as the *von Koch snowflake*** has a noninteger fractal dimension. The von Koch snowflake can be constructed by taking each side of an equilateral triangle, recursively dividing it into three equal pieces, and replacing the middle piece with two equal length pieces rotated to form two sides of an equilateral triangle as illustrated in Figure 37.1. Analyzing the self-similarity dimension of this object yields a noninteger value:

$$D = \frac{\log 4}{\log \left(1/\frac{1}{3}\right)} = \frac{\log 4}{\log 3} \approx 1.2618595$$

The von Koch curve has a fractal dimension between one and two, indicating that it is more space-filling than a line, but not as much as a two-dimensional object. This property is characteristic of fractal curves. By definition, a fractal has a noninteger self-similarity dimension.

Fractals can generally be classified as deterministic and nondeterministic (also called random fractals), depending on whether they contain randomness. In computer graphics, deterministic fractals are closely

*Mathematically speaking, the self-similarity must be infinite for the set to be a true fractal.

**Named for the mathematician Helga von Koch, who "discovered" it in 1904.

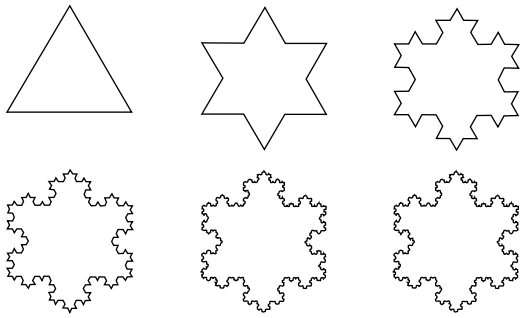


FIGURE 37.1 The von Koch snowflake after 0, 1, 2, 3, 4, and 5 iterations.

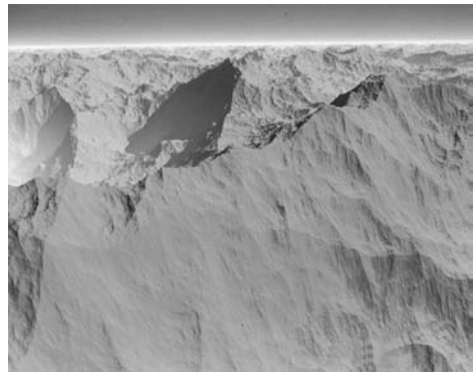


FIGURE 37.2 (See Plate 37.2 in the color insert following page **29**-22.) A fractal terrain model by Ken Musgrave. (*Slickrock* © 1992 F. Kenton Musgrave.)

related to the grammar-based L-systems described in the following section. Random fractals have been used extensively in computer graphics to model self-similar, complex, natural objects, including terrain, mountains, clouds, water, and even entire planets [Ebert et al. 2002]. Indeed, fractals are the most common technique used in graphics for modeling mountains. Most fractal terrain generation algorithms work through recursive subdivision and pseudorandom perturbation. An original surface is defined and divided equally into subparts. New vertices are added and pseudorandomly displaced from the original surface, with a displacement magnitude that decreases at each iteration. Therefore, the first iteration gives the large peaks on the surface, and later subdivisions add smaller-scale detail. A common algorithm for mountains by Fournier, Fussell, and Carpenter uses this technique by starting with a triangle, adding new vertices at the midpoint of each edge with a random height displacement, generating new triangles from these points, and repeating this subdivision with decreased height displacements at each iteration. Typically, only parameters for controlling the random number generator, the level of subdivision, and the "roughness" of the surface are needed to define extremely complex mountains and terrain. Others have used simulations of fractional Brownian motion (fBm) and statistical simulations of noise to produce the height displacements for the mountains. Musgrave [Ebert et al. 2002] uses a nonrecursive algorithm to displace the vertices of a regular grid with values iteratively calculated based on fBm and noise functions (at each iteration, the frequency is increased and the magnitude decreased) to create realistic terrain models. Recent work on fractal terrain generation has included eroding the fractal terrain to simulate natural erosion processes and the use of *multifractal* models. Multifractal terrain models allow the fractal dimension of the terrain to vary across the surface. This variation allows rougher areas and smoother areas, providing more realistic natural terrain. An example of the realistic landscapes that can be produced by these techniques can be seen in Figure 37.2.

Recent work in fractals has included the simulation of diffusion-limited aggregation (DLA) models, the previously mentioned use of multifractals, and the use of fractal models to add complex details into models. DLA is a process based on random walks (fBm motion) of particles. Several initial sticky particles are placed in space. A large number of additional particles are moved on random walks; if they touch one of the sticky particles, they stick and may become sticky also. This process continues until all the additional particles attach to or move far enough away from the original particles. DLA models are being used to model a wide range of random processes from the formation of dendrite clusters to the formation of galaxies.

There are two common applications where geometric details are added with fractals. One is the addition of realistic, detailed, fractal terrain to coarse digital elevation data to provide realistic, higher-resolution terrain models. Another is the use of fractals to add small levels of geometric detail to standard geometric models. This allows less geometry in the model, with the procedural fractal functions being applied at rendering time to add an appropriate level of detail [Hart 1995].

There are many open areas of research in fractal modeling. Better erosion models that take into account different rock hardnesses, better rain distribution models, deposition of material in addition to erosion, wind erosion, and the use of nonheight fields to allow rock overhangs will improve the realism of fractal terrains. Multifractals, diffusion-limited aggregation, and fractal detail addition are active areas of research that show great potential for geometric modeling.

37.3 Grammar-Based Models

Grammar-based models also allow natural complexity to be specified with a few parameters. Smith [1984] introduced grammar-based models to graphics, calling them *graftals.* The most commonly used grammar-based model, an L-system (named for Aristid Lindenmayer), was originally developed as a mathematical theory of plant development [Prusinkiewicz and Lindenmayer 1990]. An L-system is a formal language, a parallel graph grammar, where all the rules are applied in parallel to provide a final "word" describing the object. This parallel application of the production rules distinguishes L-systems from Chomsky grammars. Like Chomsky grammars, there are context-free L-systems (0L) and context-sensitive L-systems (1L and 2L).

Grammar-based models have been used by many authors, including Fowler, Lindenmayer, Prusinkiewicz, and Smith, to produce remarkably realistic models and images of trees, plants, and seashells. These models describe natural structures algorithmically and are closely related to deterministic fractals in their self-similarity, but fail to meet the precise mathematical definition of a fractal. Many deterministic fractals can be defined with L-systems, but not all L-systems meet the definition of a fractal.

As with most formal languages, an L-system can be described by an alphabet for the grammar, the grammar production rules, and an initial axiom. In plant modeling, alphabet symbols represent botanical structures (usually letters) and branching commands (usually "[ ]" denotes the beginning and end of a branch). We can add denotation for angular movement by introducing a "+" for clockwise rotations and "−" for counterclockwise rotations. The following simple L-system can produce a basic tree:

Alphabet:	$a, [,], +, -$
Production Rule:	$a \rightarrow a[+a]a[-a-a]a$
Initial Axiom:	a

In the L-system, each terminal symbol represents a part of the object (e.g., a branch element, internode, apex) or a directional command (e.g., turn left 30 deg) to be interpreted by a three-dimensional drawing mechanism (turtle graphics). A "word" for a tree would contain subwords describing each branch, its length, size, and branching angle, when it develops, and its connection in the tree. For example, if we interpret each "$[+a]$" as defining a right branch at 30 deg, each "$[-a]$" as defining a left branch at 30 deg,

FIGURE 37.3 Trees produced after 1, 2, and 3 derivations with the PGF software by Prusinkiewicz.

and each "a" as a tree segment (internode or apex), the above grammar can be interpreted graphically as the trees in Figure 37.3 after 1, 2, and 3 derivations and symbolically as the following:

Derivation	Word
0	a
1	$a[+a]a[-a-a]a$
2	$a[+a]a[-a-a]a[+a[+a]a[-a-a]a]a[+a]a[-a-a]a[-a[+a]a[-a-a]a$ $-a[+a]a[-a-a]a]a[+a]a[-a-a]a$

To allow more complex plant and plant growth models, L-systems have been extended to include context sensitivity, word age information, and stochastic rule evaluation. Context sensitivity allows the relationships between parts of plants to be incorporated into the model. 1L-systems have one-sided contexts: either a right or a left context, yielding production rules of the form

$$a_l < a \rightarrow F$$

and

$$a > a_r \rightarrow F$$

The production rule is applied if and only if either its right context, aa_r, is satisfied or its left context, $a_l a$, is satisfied (either if a is preceded by a_l or if a is followed by a_r). 2L-systems have both a left context and a right context, yielding production rules of the form

$$a_l < a > a_r \rightarrow F$$

Stochastic L-systems assign a probability to the application of a rule. This allows randomness into the creation of the plant, so that each plant is slightly different. Deterministic L-systems produce identical plants each time they are evaluated and would, therefore, create an unrealistic field of identical flowers. Stochastic rule evaluation is added into the grammar by associating a probability, p, with each production rule as follows:

$$a \xrightarrow{p} F$$

Parametric L-systems allow word age information and conditional expressions based on parameter values to be included into the model, permitting developmental growth models for plants. A parametric L-system has rules of the form

$$F(t) : t > 2 \rightarrow F(t-1)H(t+7)$$

where H is another parametric production rule. Parametric L-systems can be combined with stochastic L-systems and context-sensitive L-systems to achieve a powerful, flexible grammar.

Recent work in L-systems allows better developmental models, more advanced biologically based growth models, incorporation of more growth parameters, and environmental effects. Extensions of L-systems allow the death of the buds, dropping of leaves, and cutting of branches with the inclusion of erasing and

FIGURE 37.4 (See Plate 37.4 in the color insert following page **29**-22.) Horse chestnut tree created with a modified L-system that takes into account branch competition for light. (© 1995 R. Měch and P. Prusinkiewicz.)

cutting operators. Botanically based flowering structures (inflorescences), branching structures (sympodial and monopodial), and arrangement of lateral plant organs (phyllotaxis) [Fowler et al. 1992] have been incorporated into L-systems to more accurately model plants. **Tropism** effects (gravity, wind, growth toward light), pruning, amount of light, and availability of nutrients have also been incorporated into these grammars. These natural effects and growth processes not only affect the structure (topology) of the tree, but also affect the branching angles, petal and seed location and shape, and thickness of each branch segment. When generating the geometric plant model from the L-system grammar, these effects need to be included in determining the geometry and size of each structure in the plant. Figure 37.4 shows a realistic image of a horse chestnut tree generated by a modified L-system that takes into account the competition of branches for light.

Ongoing L-systems research includes environmentally-sensitive L-systems [Prusinkiewicz et al. 1995], the modeling of entire ecosystems [Deussen et al. 1998], modeling feathers [Chen et al. 2002], procedural modeling of cities [Parish and Miller 2001], and the use of L-systems for modeling other growth processes and artificial life. Additionally, better developmental models can be simulated and more accurate modeling of natural growth factors can be included.

37.4 Procedural Volumetric Models

Another procedural modeling technique, procedural volumetric modeling (also called hypertextures, volume density functions, and fuzzy blobbies), uses algorithms to define and animate three-dimensional volumetric objects and natural phenomena [Ebert et al. 2002]. These techniques have been used to model natural phenomena such as fire (Stam and Inakage), gases such as smoke, clouds, and fog (Ebert, Perlin, Sakas, Stam), and water (Ebert, Perlin). The volumetric procedures take as input a point location in space, time (if animating), and parameters that describe the object being modeled, and return the density and color of the object for that location in space. Complex volumetric phenomena can therefore be described with a few parameters.

An extremely simple procedural volumetric model for a spherical volume object is given below. This function, **spherical_pvm**, defines a sphere of radius **outer_radius**, with a solid center of radius **inner_radius**. The region between **inner_radius** and **outer_radius** is a semi-solid area that increases in density as the inner radius is approached. The following function, suggested by Perlin to create a *soft sphere* [Ebert et al. 2002], returns a density value when a point in space and the sphere definition are given as input:

```
typedef struct rgb_td { float r,g,b} rgb_td;
typedef struct xyz_td { float x,y,z} xyz_td;
typedef struct vol_td
    {
     float  density;
     rgb_td color;
    } vol_td;

vol_td spherical_pvm(xyz_td pnt,xyz_td center,float inner_radius,
                     float outer_radius, float density_factor)
{
     float outer_radius_2, inner_radius_2, distance_2;
     vol_td vol;

     distance_2 = (pnt.x - center.x)*(pnt.x - center.x)
                + (pnt.y - center.y)* (pnt.y - center.y)
                + (pnt.z - center.z)*(pnt.z - center.z);

     /* compute outer and inner radius squared values */
     outer_radius_2 = outer_radius * outer_radius;
     inner_radius_2 = inner_radius * inner_radius;

  if (distance_2 < inner_radius_2)
     { /* inside inner radius */
      vol.density =1.0*density_factor;
      vol.color.r = 1.0; vol.color.g = 0.0; vol.color.b = 0.0;
     }
  else if (distance_2 > outer_radius_2)
     { /* outside of the sphere */
      vol.density = 0.0;
      vol.color.r = 0.0; vol.color.g = 0.0; vol.color.b = 0.0;
     }
  else
     { /* in the soft area of the sphere */
      vol.density = density_factor*(distance-inner_radius_2)/
           (outer_radius_2-inner_radius_2);
      vol.color.r=vol.color.g = 1.0*vol.density;vol.color.b=0;
     }
  return(vol);
}
```

Many authors have used these techniques to describe a wide range of natural objects. Perlin has successfully created realistic rock arches, woven fabric, smoke, and fur [Ebert et al. 2002], basing his procedures on a statistical simulation of turbulence and random noise to give natural-looking complexity to the objects. Perlin's turbulence function has been used as a building block for volumetric procedural objects,

solid textures, and many other applications in computer graphics. This turbulence function defines a three-dimensional turbulence space by summing octaves of random noise, increasing the frequency and decreasing the amplitude at each step. The C function below is one implementation of Perlin's turbulence function:

```
float turbulence(xyz_td pnt, float pixel_size)
{
  float t, scale;
  t=0;
  for(scale=1.0; scale >pixel_size; scale/=2.0)
    {
      pnt.x = pnt.x/scale; pnt.y = pnt.y/scale;
      pnt.z = pnt.z/scale;
      t+=calc_noise(pnt)* scale;
    }
  return(t);
}
```

This function takes as input a three-dimensional point location in space, **pnt**, and an indication of the number of octaves of noise to sum, **pixel_size**,* and returns the turbulence value for that location in space. This function has a fractal characteristic in that it is self-similar and sums the octaves of random noise, doubling the frequency while halving the amplitude at each step. The heart of the **turbulence** function is the **calc_noise** function used to simulate uncorrelated random noise. Many authors have used various implementations of the noise function (see [Ebert et al. 2002] for several possible implementations). One implementation is the **calc_noise** function given below, which uses linear interpolation of a $64 \times 64 \times 64$ grid of random numbers**:

```
#define SIZE       64
#define SIZE_1     65
double drand48();
float calc_noise();
float noise[SIZE+1][SIZE+1][SIZE+1];

/*
 ****************************************************************
 *                          Calc_noise
 ****************************************************************
 * This is basically how the trilinear interpolation works:
 * interpolate down left front edge of the cube first, then the
 * right front edge of the cube(p_l, p_r). Next, interpolate down
 * the left and right back edges (p_l2, p_r2). Interpolate across
 * the front face between p_l and p_r (p_face1) and across the
 * back face between p_l2 and p_r2 (p_face2). Finally, interpolate
 * along line between p_face1 and p_face2.
 ****************************************************************
 */
```

*This variable name is used in reference to the projected area of the pixel in the three-dimensional turbulence space for antialiasing.

**The actual implementation uses a 65^3 table with the 64th entry equal to the 0th entry for quicker interpolation.

```
float
calc_noise(xyz_td pnt)
{
  float t1;
  float p_l,p_l2, /* value lerped down left side of face1 &
                   * face 2 */
        p_r,p_r2, /* value lerped down right side of face1 &
                   * face 2 */
        p_face1, /* value lerped across face 1 (x-y plane ceil
                  * of z) */
        p_face2, /* value lerped across face 2 (x-y plane floor
                  * of z) */
        p_final; /* value lerped through cube (in z) */

  extern float noise[SIZE_1][SIZE_1][SIZE_1];
  float        tnoise;
  register int x, y, z,px,py,pz;
  int          i,j,k, ii,jj,kk;
  static int   firstime =1;

  /* During first execution, create the random number table of
   * values between 0 and 1, using the Unix random number
   * generator drand48(). Other random number generators may be
   * substituted. These noise values can also be stored to a
   * file to save time.
   */
  if (firsttime)
    { for (i=0; i<SIZE; i++)
       for (j=0; j<SIZE; j++)
         for (k=0; k<SIZE; k++)
           {
             noise[i][j][k] = (float)drand48();
             /* A crude way to make element[64]=element[0] for
              * easier linear interpolation */
              if(i==0) noise[SIZE][j][k] = noise[i][j][k];
              if(j==0) noise[i][SIZE][k] = noise[i][j][k];
              if(k==0) noise[i][j][SIZE] = noise[i][j][k];
           }
       firsttime=0;
    }

  px = (int)pnt.x;
  py = (int)pnt.y;
  pz = (int)pnt.z;
  x = px &(SIZE-1); /* make sure the values are in the table */
  y = py &(SIZE-1); /* Effectively, replicates the table
                     * throughout space */
  z = pz &(SIZE-1);

  t1 = pnt.y - py;
  p_l = noise[x][y][z+1] +t1*(noise[x][y+1][z+1]
        - noise[x][y][z+1]);
```

```
  p_r = noise[x+1][y][z+1] +t1*(noise[x+1][y+1][z+1]
        - noise[x+1][y][z+1]);
  p_l2 = noise[x][y][z] +t1*(noise[x][y+1][z] - noise[x][y][z]);
  p_r2 = noise[x+1][y][z] +t1*(noise[x+1][y+1][z]
        - noise[x+1][y][z]);
  t1 = pnt.x - px;
  p_face1 = p_l + t1 * (p_r - p_l);
  p_face2 = p_l2 + t1 * (p_r2 -p_l2);
  t1 = pnt.z - pz;
  p_final = p_face2 + t1*(p_face1 -p_face2);
  return(p_final);
}
```

Ebert et al. [2002] have used similar functions to model and animate steam, fog, smoke, clouds, and solid marble. The turbulence function and random noise functions allow a simple simulation of turbulent flow processes. To simulate steam rising from a teacup, a volume of gas is placed over the teacup. The basic gas is defined by the following function:

```
float basic_gas(xyz_td pnt, float density, float density_scalar,
                float exponent)
{
  float turb, density;

  turb =turbulence(pnt);
  density = pow(turb*density_scalar, exponent);
  return(density);
}
```

This function creates a three-dimensional gas space controlled by the values **density_scalar** and **exponent. density_scalar** controls the denseness of the gas, while **exponent** controls the sharpness and sparseness of the gas (from continuously varying to sharp individual plumes). This function is then shaped to create steam over the center of the teacup by spherically attenuating the density toward the edge of the cup and linearly attenuating the density as the distance from the top of the cup increases, simulating the gas dissipation as it rises. The following procedure will produce an image of steam rising from a teacup, as in Figure 37.5.

FIGURE 37.5 Steam rising from a teacup. (See Plate 37.5 in the color insert following page 29-22.) (© 1991 David S. Ebert.)

```
float steam(xyz_td pnt, xyz_td pnt_world, float exponent,
            float density_scalar,xyz_td tea_center,float radius)
    {
     float    turb, dist, dist_sq, density_max, fall_off, offset;
     xyz_td   diff;

     turb = turbulence(pnt);
     density = pow(turb*density_scalar, exponent);

    /* determine distance from center of the teacup squared. */
     XYZ_SUB(diff,tea_center, pnt_world);
     dist_sq = DOT_XYZ(diff,diff);
    /* calculate relative distance from center with some
     * randomness */
     density_max = dist_sq/(radius*radius);
     density_max += .2*noise(pnt);

    /* Use a cosine function to spherically attenuate the
     * density */
     if(density_max >= .25) /* ramp off if > 25% from center */
       { /* get table index 0:RAMP_SIZE-1 */
           density_max = MAX((density_max -.25)*4/3, 1.0);
           fall_off = (cos(density_max*M_PI)+1.0)/2.0;
           density *=fall_off;
       }
    /* Use exponential attenuation to decrease density as steam
     * rises */
     dist = pnt_world.y - tea_center.y;
     if(dist > 0.0)
       { dist = (dist +noise(pnt)*.1)/radius.y;
         if(dist > .05)
           { offset = (dist -.05)*1.111111;
             offset = 1 - (exp(offset)-1.0)/1.718282;
             density = density*offset;
           }
       }
     return(density);
    }
```

These procedural techniques can be easily animated by adding time as a parameter to the algorithm [Ebert et al. 2002]. They allow the use of simple simulations of natural complexity (noise, turbulence) to speed computation, but also allow the incorporation of physically based parameters where appropriate and feasible. This flexibility is one of the many advantages of procedural techniques.

Procedural volumetric models require volume rendering techniques to create images of these objects. Traditionally, most authors have used a modification of volume ray tracing. Several authors have incorporated physically based models for shadowing and illumination into rendering algorithms for these models [Ebert and Parent 1990, Stam and Fiume 1995]. However, with the advent of programmable graphics hardware that allow user-defined programs to be executed for each polygonal pixel-sized fragment, these function can potentially be evaluated at interactive rates by the graphics hardware. Ebert et al. [2002] describe methods to adapt procedural volumetric modeling techniques to modern graphics hardware.

Procedural volumetric modeling is still an active area of research and has many research problems to address. Efficient rendering of these models is still an important issue, as is the development of a

FIGURE 37.6 A close-up of a fly through of a procedural volumetric cloud incorporating volumetric implicit models into the procedural volumetric model. (©2002 David S. Ebert.)

larger toolbox of useful primitive functions. The incorporation of more physically based models will increase the accuracy and realism of the water, gas, and fire simulations. Finally, the development of an interactive procedural volumetric modeling system will speed the development of procedural volumetric modeling techniques. The procedural interfaces in the latest commercial modeling, rendering, and animation packages are now allowing the specification of procedural models, but the user control is still lacking.

Combining traditional volumetric procedural models with implicit functions, described below, creates a model that has the advantages of both techniques. Implicit functions have been used for many years as a modeling tool for creating solid objects and smoothly blended surfaces [Bloomenthal et al. 1997]. However, only a few researchers have explored their potential for modeling volumetric density distributions of semi-transparent volumes (e.g., [Nishita et al. 1996, Stam and Fiume 1991, Stam and Fiume 1993, Stam and Fiume 1995, Ebert 1997]). Ebert's early work on using volume rendered implicit spheres to produce a fly-through of a volumetric cloud was described in [Ebert et al. 1997]. This work has been developed further to use implicits to provide a natural way of specifying and animating the global structure of the cloud, while using more traditional procedural techniques to model the detailed structure. More details on the implementation of these techniques can be found in [Ebert et al. 2002]. An example of a procedural volumetric cloud modeled using the above turbulence-based techniques combined with volumetricly evaluated implicit spheres, can be seen in Figure 37.6.

37.5 Implicit Surfaces

While previously discussed techniques have been used primarily for modeling the complexities of nature, implicit surfaces [Bloomenthal et al. 1997] (also called blobby molecules [Blinn 1982], metaballs [Nishimura et al. 1985], and soft objects [Wyvill et al. 1986]) are used in modeling organic shapes, complex man-made shapes, and "soft" objects that are difficult to animate and describe using more traditional techniques. Implicit surfaces are surfaces of constant value, **isosurfaces**, created from blending primitives (functions or skeletal elements) represented by implicit equations of the form $F(x, y, z) = 0$, and were first introduced into computer graphics by Blinn [1982] to produce images of electron density clouds. A simple example of an implicit surface is the sphere defined by the equation

$$F(x, y, z) : x^2 + y^2 + z^2 - r^2 = 0$$

Implicit surfaces are a more concise representation than parametric surfaces and provide greater flexibility in modeling and animating soft objects.

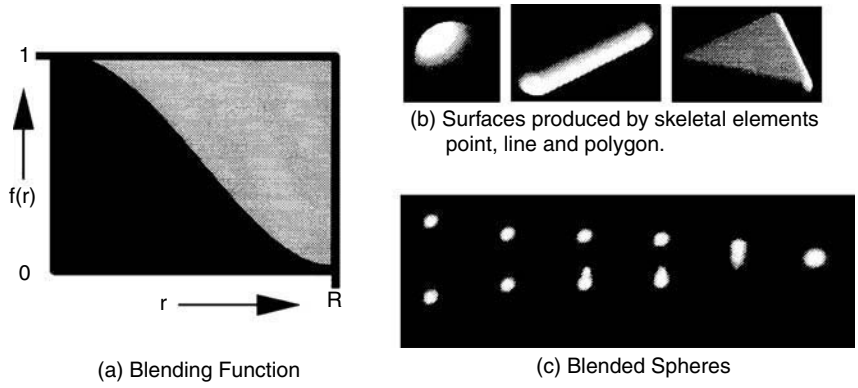


FIGURE 37.7 (a) Blending function, (b) surfaces produced by skeletal elements point, line and polygon, and (c) blended spheres. (© 1995 Brian Wyvill.)

For modeling complex shapes, several basic implicit surface primitives are smoothly blended to produce the final shape. For the blending function, Blinn used an exponential decay of the field values, whereas Wyvill [Wyvill et al. 1986, 1993] uses the cubic function

$$F_{\text{cub}}(r) = -\frac{4}{9}\frac{r^6}{R^6} + \frac{17}{9}\frac{r^4}{R^4} - \frac{22}{9}\frac{r^2}{R^2} + 1$$

This cubic blending function, whose values range from 1 when $r = 0$ to 0 at $r = R$, has several advantages for complex shape modeling. First, its value drops off quickly to zero (at the distance R), reducing the number of primitives that must be considered in creating the final surface. Second, it has zero derivatives at $r = 0$ and $r = R$ and is symmetrical about the contour value 0.5, providing for smooth blends between primitives. Finally, it can provide volume-preserving primitive blending. Figure 37.7(a) shows a graph of this blending function, and Figure 37.7(c) shows the blending of two spheres using this function. A good comparison of blending functions can be found in [Bloomenthal et al. 1997].

For implicit surface primitives, Wyvill uses procedures that return a functional (field) value for the field defined by the primitive. Field primitives, such as lines, points, polygons, circles, splines, spheres, and ellipsoids, are combined to form a basic skeleton for the object being modeled. The surfaces resulting from these skeletal elements can be seen in Figure 37.7(b). The object is then defined as an offset (isosurface) from this series of blended skeletal elements. Skeletons are an intuitive representation and are easily displayed and animated.

Modeling and animation of implicit surfaces is achieved by controlling the skeletal elements and blending functions, providing complex models and animations from a few parameters (another example of data amplification). Deformation, path following, warping, squash and stretch, gravity, and metamorphosis effects can all be easily achieved with implicit surfaces. Very high-level animation control is achieved by animating the basic skeleton, with the surface defining the character following naturally. The animator does not have to be concerned with specifying the volume-preserving deformations of the character as it moves.

There are two common approaches to rendering implicit surfaces. One approach is to directly ray-trace the implicit surfaces, requiring the modification of a standard ray tracer. The second approach is to polygonalize the implicit surfaces [Ning and Bloomenthal 1993, Wyvill et al. 1993] and then use traditional polygonal rendering algorithms on the result. Uniform-voxel space polygonization can create large numbers of unnecessary polygons to accurately represent surface details. More complicated tessellation and shrinkwrapping algorithms have been developed which create appropriately sized polygons [Wyvill et al. 1993].

Recent work in implicit surfaces [Wyvill and Gascuel 1995, Wyvill et al. 1999] has extended their use to character modeling and animation, human figure modeling, and representation of rigid objects through

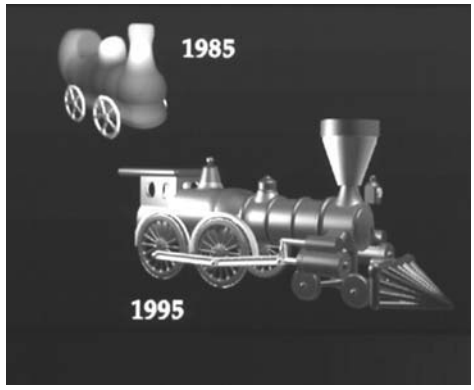


FIGURE 37.8 (See Plate 37.8 in the color insert following page **29**-22.) Ten years in implicit surface modeling. The locomotive labeled 1985 shows a more traditional soft object created by implicit surface techniques. The locomotive labeled 1995 shows the results achievable by incorporating constructive solid geometry techniques with implicit surface models. (© 1995 Brian Wyvill.)

the addition of constructive solid geometry (CSG) operators. Implicit surface modeling techniques have advanced significantly in the past 10 years, as can be seen by comparing the locomotives in Figure 37.8. The development of better blending algorithms, which solve the problems of unwanted primitive blending and surface bulging, is an active area of research [Bloomenthal 1995]. Advanced animation techniques for implicit surfaces, including higher-level animation control, surface collision detection, and shape metamorphosis animation, are also active research areas. Finally, the development of interactive design systems for implicit surfaces will greatly expand the use of this modeling technique. The use of implicit functions have also expanded to compact representations of surface objects [Turk and O'Brien 2002].

37.6 Particle Systems

Particle systems are different from the previous four techniques in that their abstraction is in control of the animation and specification of the object. A particle-system object is represented by a large collection (cloud) of very simple geometric particles that change stochastically over time. Therefore, particle systems do use a large database of geometric primitives to represent natural objects ("fuzzy objects"), but the animation, location, birth, and death of the particles representing the object are controlled algorithmically. As with the other procedural modeling techniques, particle systems have the advantage of database amplification, allowing the modeler/animator to specify and control this extremely large cloud of geometric particles with only a few parameters.

Particle systems were first used in computer graphics by Reeves [1983] to model a wall of fire for the movie *Star Trek II: The Wrath of Khan* (see Figure 37.9). Because particle systems are a volumetric modeling technique, they are most commonly used to represent volumetric natural phenomena such as fire, water, clouds, snow, and rain [Reeves 1983]. An extension of particle systems, *structured particle systems,* has also been used to model grass and trees [Reeves and Balu 1985].

A particle system is defined by both a collection of geometric particles and the algorithms that govern their creation, movement, and death. Each geometric particle has several attributes, including its initial position, velocity, size, color, transparency, shape, and lifetime.

To create an animation of a particle system object, the following steps are performed at each time step [Reeves 1983]:

1. New particles are generated and assigned their attributes.
2. Particles that have existed in the system past their lifetime are removed.

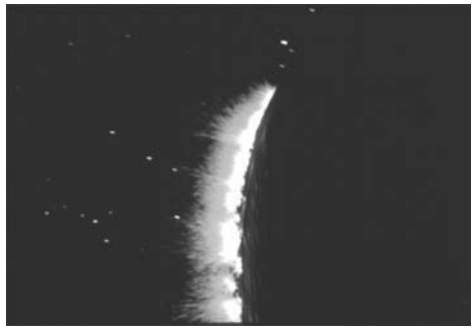


FIGURE 37.9 (See Plate 37.9 in the color insert following page **29**-22.) An image from *Star Trek II: The Wrath of Khan* showing a wall of fire created with a particle system. (© 1987 Pixar.)

3. Each remaining particle is moved and transformed by the particle-system algorithms as prescribed by their individual attributes.
4. These particles are rendered, using special-purpose rendering algorithms, to produce an image of the particle system.

The creation, death, and movement of particles are controlled by stochastic procedures, allowing complex, realistic motion to be created with a few parameters. The creation procedure for particles is controlled by parameters defining either the mean number of particles created at each time step and its variance, or the mean number of particles created per unit of screen area at each time step and its variance.* The actual number of particles created is stochastically determined to be within *mean* + *variance* and *mean* − *variance*. The initial color, velocity, size, and transparency are also stochastically determined by mean and variance values. The initial shape of the particle system is defined by an origin, a region about this origin in which new generated particles are placed, angles defining the orientation of the particle system, and the initial direction of movement for the particles.

The movement of particles is also controlled by stochastic procedures (stochastically determined velocity vectors). These procedures move the particles by adding their velocity vector to their position vector. Random variations can be added to the velocity vector at each frame, and acceleration procedures can be incorporated to simulate effects such as gravity, vorticity, and conservation of momentum and energy. The simulation of physically based forces allows realistic motion and complex dynamics to be displayed by the particle system, while being controlled by only a few parameters. In addition to the movement of particles, their color and transparency can also change dynamically to give more complex effects. The death of particles is controlled very simply by removing particles from the system whose lifetimes have expired or that have strayed more than a given distance from the origin of the particle system.

An example of the effects achievable by such a particle system can be seen in Figure 37.9, an image from the Genesis Demo sequence from *Star Trek II: The Wrath of Khan.* In this image, a two-level particle system was used to create the wall of fire. The first-level particle system generated concentric, expanding rings of particle systems on the planet's surface. The second-level particle system generated particles at each of these locations, simulating explosions. During the Genesis Demo sequence, the number of particles in the system ranged from several thousand initially to over 750,000 near the end.

Reeves extended the use of particle systems to model fields of grass and forests of trees, calling this new technique structured particle systems [Reeves and Blau 1985]. In structured particle systems, the particles are no longer an independent collection of particles, but rather form a connected, cohesive three-dimensional object and have many complex relationships among themselves. Each particle represents an

*These values can be varied over time as well.

element of a tree (e.g., branch, leaf) or part of a blade of grass. These particle systems are therefore similar to L-systems and graftals, specifically probabilistic, context-sensitive L-systems. Each particle is similar to a letter in an L-system alphabet, and the procedures governing the generation, movement, and death of particles are similar to the production rules. However, they differ from L-systems in several ways. First, the goal of structured particle systems is to model the visual appearance of whole collections of trees and grass, and not to correctly model the detailed geometry of each plant. Second, they are not concerned with biological correctness or modeling growth of plants. Structured particle systems construct trees by recursively generating subbranches, with stochastic variations of parameters such as branching angle, thickness, and placement within a value range for each type of tree. Additional stochastic procedures are used for placement of the trees on the terrain, random warping of branches, and bending of branches to simulate tropism. A forest of such trees can therefore be specified with a few parameters for distribution of tree species and several parameters defining the mean values and variances for tree height, width, first branch height, length, angle, and thickness of each species.

Both regular particle systems and structured particle systems pose special rendering problems because of the large number of primitives. Regular particle systems have been rendered simply as point light sources (or linear light sources for antialiased moving particles) for fire effects, accumulating the contribution of each particle into the frame buffer and compositing the particle system image with the surface rendered image (as in Figure 37.9). No occlusion or interparticle illumination is considered. Structured particle systems are much more difficult to render, and specialized probabilistic rendering algorithms have been developed to render them [Reeves and Blau 1985]. Illumination, shadowing, and hidden-surface calculations need to be performed for the particles. Because stochastically varying objects are being modeled, approximately correct rendering will provide sufficient realism. Probabilistic and approximate techniques are used to determine the shadowing and illumination of each tree element. The particle's distance into the tree from the light source determines its amount of **diffuse shading** and probability of having **specular highlights**. Self-shadowing is simulated by exponentially decreasing the **ambient illumination** as the particle's distance within the tree increases. External shadowing is also probabilistically calculated to simulate the shadowing of one tree by another tree. For hidden-surface calculations, an initial depth sort of all trees and a **painter's algorithm** is used. Within each tree, again, a painter's algorithm is used, along with a back-to-front bucket sort of all the particles. This will not correctly solve the hidden-surface problem in all cases, but will give realistic, approximately correct images.

Efficient rendering of particle systems is still an open research problem (e.g., [Etzmuss et al. 2002]). Although particle systems allow complex scenes to be specified with only a few parameters, they sometimes require rather slow, specialized rendering algorithms. Simulation of fluids [Miller and Pearce 1989], cloth [Breen et al. 1994, Baraff and Witkin 1998, Plath 2000], and surface modeling with oriented particle systems [Szeliski and Tonnesen 1992] are recent, promising extensions of particle systems. Sims [1990] demonstrated the suitability of highly parallel computing architectures to particle-system simulation. Particle systems, with their ease of specification and good dynamical control, have great potential when combined with other modeling techniques such as implicit surfaces [Witkin and Heckbert 1994] and volumetric procedural modeling.

Particle systems provide a very nice, powerful animation system for high-level control of complex dynamics and can be combined with many of the procedural techniques described in this chapter. For example, turbulence functions are often combined with particle systems, such as Ebert's use of particle systems for animating cloud dynamics [Ebert et al. 2002].

37.7 Research Issues and Summary

Advanced modeling techniques will continue to play an important role in computer graphics. As computers become more powerful, the complexity that can be rendered will increase; however, the capability of humans to specify more geometric complexity (millions of primitives) will not. Therefore, procedural techniques, with their capability to amplify the user's input, are the only viable alternative. These techniques will evolve in their capability to specify and control incredibly realistic and detailed models with a small number of

user-specified parameters. More work will be done in allowing high-level control and specification of models in user-understandable terms, while more complex algorithms and improved physically based simulations will be incorporated into these procedures. Finally, the automatic generation of procedural models through artificial evolution techniques, similar to those of Sims [1994], will greatly enhance the capabilities and uses of these advanced modeling techniques.

Defining Terms

Ambient illumination: An approximation of the global illumination on the object, usually modeled as a constant amount of illumination per object.

Diffuse shading: The illumination of an object where light is reflected equally in all directions, with the intensity varying based on surface orientation with respect to the light source. This is also called Lambertian reflection because it is based on Lambert's law of diffuse reflection.

Fractal: Generally refers to a complex geometric object with a large degree of self-similarity and a non-integer fractal dimension that is not equal to the object's topological dimension.

Grammar-based modeling: A class of modeling techniques based on formal languages and formal grammars where an alphabet, a series of production rules, and initial axioms are used to generate the model.

Implicit surfaces: Isovalued surfaces created from blending primitives that are modeled with implicit equations.

Isosurface: A surface defined by all the points where the field value is the same.

L-system: A parallel graph grammar in which all the production rules are applied simultaneously.

Painter's algorithm: A hidden-surface algorithm that sorts primitives in back-to-front order, then "paints" them into the frame buffer in this order, overwriting previously "painted" primitives.

Particle system: A modeling technique that uses a large collection (thousands) of particles to model complex natural phenomena, such as snow, rain, water, and fire.

Phyllotaxis: The regular arrangement of plant organs, including petals, seeds, leaves, and scales.

Procedural volumetric models: Use algorithms to define the three-dimensional volumetric representation of an object.

Specular highlights: The bright spots or highlights on objects caused by angular-dependent illumination. Specular illumination depends on the surface orientation, the observer location, and the light source location.

Surface-based modeling: Refers to techniques for modeling the three-dimensional surfaces of objects.

Tropism: An external directional influence on the branching patterns of trees.

Volumetric modeling: Refers to techniques that model objects as three-dimensional volumes of material, instead of being defined by surfaces.

References

Baraff, D. and Witkin, A. 1998. Large Steps in Cloth Simulation. *Proceedings of SIGGRAPH*, 98, 43–54, ACM Press.

Blinn, J. F. 1982. A generalization of algebraic surface drawing. *ACM Trans. Graphics*, 1(3):235–256.

Bloomenthal, J. 1995. Skeletal Design of Natural Forms. Ph.D. thesis, Department of Computer Science, University of Calgary.

Bloomenthal, J., Bajaj, C., Blinn, J., Cani-Gascuel, M.P., Rockwood, A., Wyvill, B., and Wyvill, G. 1997. *Introduction to Implicit Surfaces*, Morgan Kaufman Publishers.

Breen, D.E., House, D.H., and Wozny, M.J. 1994. Predicting the drape of woven cloth using interacting particles, pp. 365–372. In *Proc. SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994),* A. Glassner, Ed., Computer Graphics Proceedings, Annual Conf. Ser. ACM SIGGRAPH, ACM Press.

Chen, Y., Xu, Y., Guo, B., and Shum, H. 2002. Modeling and Rendering of Realistic Feathers. *ACM Transactions on Graphics*, 21(3):630–636.

Deussen, O., Hanrahan, P., Lintermann, B., Mech, R., Pharr, M., and Prusinkiewicz, P. 1998. Realistic Modeling and Rendering of Plant Ecosystems. *Proceedings of SIGGRAPH '98*, 275–286.

Ebert, D. and Parent, R. 1990. Rendering and animation of gaseous phenomena by combining fast volume and scanline a-buffer techniques. *Comput. Graphics (Proc. SIGGRAPH)*, 24:357–366.

Ebert, D. 1997. Volumetric Modeling with Implicit Functions: A Cloud Is Born, *SIGGRAPH 97 Visual Proceedings (Technical Sketch)*, 147, ACM SIGGRAPH.

Ebert, D., Musgrave, F.K., Peachey, D., Perlin, K., and Worley, S. 2002. *Texturing and Modeling: A Procedural Approach, third edition*. Morgan Kaufman Publishers, San Francisco, CA. Professional, Boston.

Etzmuss, O., Eberhardt, B., and Hauth, M. 2000. Implicit-Explicit Schemes for Fast Animation with Particle Systems. *Computer Animation and Simulation 2000,* 138–151.

Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F. 1990. *Computer Graphics: Principles and Practices*, 2nd ed. Addison-Wesley, Reading, MA.

Fowler, D.R., Prusinkiewicz, P., and Battjes, J. 1992. A collision-based model of spiral phyllotaxis. *Comput. Graphics (Proc. SIGGRAPH)*, 26:361–368.

Hart, J. 1995. Procedural models of geometric detail. In *SIGGRAPH '95: Course 33 Notes*. ACM SIGGRAPH.

Mandelbrot, B.B. 1983. *The Fractal Geometry of Nature*. W. H. Freeman, New York.

Miller, G. and Pearce, A. 1989. Globular dynamics: a connected particle system for animating viscous fluids. *Comput. and Graphics,* 13(3):305–309.

Ning, P. and Bloomenthal, J. 1993. An evaluation of implicit surface tilers. *IEEE Comput. Graphics Appl.,* 13(6):33–41.

Nishimura, H., Hirai, A., Kawai, T., Kawata, T., Kawa, I.S., and Omura, K. 1985. Object modelling by distribution function and a method of image generation (in Japanese). In *Journals of Papers Given at the Electronics Communication Conference '85*, J68-D(4).

Nishita, T., Nakamae, E., and Dobashi, Y. 1996. Display of Clouds and Snow Taking into Account Multiple Anisotropic Scattering and Sky Light, *SIGGRAPH 96 Conference Proceedings*, 379–386. ACM SIGGRAPH.

Parish, Y. and Miller, P. 2001. Procedural Modeling of Cities. *Proceedings of ACM SIGGRAPH 2001*, 301–308.

Peitgen, H.-O., Jürgens, H., and Saupe, D. 1992. *Chaos and Fractals: New Frontiers of Science*. Springer-Verlag, New York.

Plath, J. 2000. Realistic modelling of textiles using interacting particle systems. *Computers & Graphics*, 24(6): 897–905.

Prusinkiewicz, P. and Lindenmayer, A. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag.

Prusinkiewicz, P., Hammel, M., and Mech, R. 1995. The artificial life of plants. In *SIGGRAPH '95: Course Notes.* ACM SIGGRAPH.

Reeves, W.T. 1983. Particle systems — a technique for modeling a class of fuzzy objects. *ACM Trans. Graphics*, 2:91–108.

Reeves, W.T. and Blau, R. 1985. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Comput. Graphics (Proc. SIGGRAPH)*, 19:313–322.

Sims, K. 1990. Particle animation and rendering using data parallel computation. *Comput. Graphics (Proc. SIGGRAPH)*, 24:405–413.

Sims, K. 1994. Evolving virtual creatures, pp. 15–22. *Proc. of SIGGRAPH '94*, Computer Graphics Proc., Annual Conf. Series. ACM SIGGRAPH, ACM Press.

Smith, A.R. 1984. Plants, fractals and formal languages. *Computer Graphics (Proc. SIGGRAPH)*, 18:1–10.

Stam, J. and Fiume, E. 1991. A multiple-scale stochastic modeling primitive, *Proceedings Graphics Interface '91.*

Stam, J. and Fiume, E. 1993. Turbulent wind fields for gaseous phenomena, *Computer Graphics (SIGGRAPH '93 Proceedings)*, 27:369–376.

Stam, J. and Fiume, E. 1995. Depicting fire and other gaseous phenomena using diffusion processes, pp. 129–136. In *Proc. SIGGRAPH '95*, Computer Graphics Proc., Annual Conf. Series. ACM SIGGRAPH, ACM Press.

Szeliski, R. and Tonnesen, D. 1992. Surface modeling with oriented particle systems. *Comput. Graphics (Proc. SIGGRAPH)*, 26:185–194.

Turk, G. and O'Brien, J. 2002. Modelling with implicit surfaces that interpolate. *ACM Transactions on Graphics*, 21(4):855–873.

Watt, A. and Watt, M. 1992. *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, Reading, MA.

Witkin, A.P. and Heckbert, P.S. 1994. Using particles to sample and control implicit surfaces, pp. 269–278. In *Proc. SIGGRAPH '94*, Computer Graphics Proc. Annual Conf. Series. ACM SIGGRAPH, ACM Press.

Wyvill, B. and Gascuel, M.-P., 1995. *Implicit Surfaces '95, The First International Workshop on Implicit Surfaces*. INRIA, Eurographics.

Wyvill, G., McPheeters, C., and Wyvill, B. 1986. Data structure for soft objects. *The Visual Computer*, 2(4):227–234.

Wyvill, B., Bloomenthal, J., Wyvill, G., Blinn, J., Hart, J., Bajaj, C., and Bier, T. 1993. Modeling and animating implicit surfaces. In *SIGGRAPH '93: Course 25 Notes*.

Wyvill, B., Galin, E., and Guy, A. 1999. Extending The CSG Tree. Warping, Blending and Boolean Operations in an Implicit Surface Modeling System, *Computer Graphics Forum*, 18(2):149–158.

Further Information

There are many sources of further information on advanced modeling techniques. Two of the best resources are the proceedings and course notes of the annual ACM SIGGRAPH conference. The SIGGRAPH conference proceedings usually feature a section on the latest, and often best, results in modeling techniques. The course notes are a very good source for detailed, instructional information on a topic. Several courses at SIGGRAPH '92, '93, '94, and '95 contained notes on procedural modeling, fractals, particle systems, implicit surfaces, L-systems, artificial evolution, and artificial life.

Standard graphics texts, such as *Computer Graphics: Principles and Practice* by Foley, van Dam, Feiner, and Hughes [Foley et al. 1990] and *Advanced Animation and Rendering Techniques* by Watt and Watt [1992], contain introductory explanations to these topics. The reference list contains references to excellent books and, in most cases, the most comprehensive sources of information on the subject. Additionally, the book entitled *The Fractal Geometry of Nature*, by Mandelbrot [1983], is a classic reference for fractals. For implicit surfaces, the book by Bloomenthal, Wyvill, et al. [1997] is a great reference.

Another good source of reference material is specialized conference and workshop proceedings on modeling techniques. For example, the proceedings of the Eurographics '95 Workshop on Implicit Surfaces contains state-of-the-art implicit surfaces techniques.

38

Mainstream Rendering Techniques

Alan Watt
University of Sheffield

Steve Maddock
University of Sheffield

38.1 Introduction

Rendering is the name given to the process in three-dimensional graphics whereby a geometric description of an object is converted into a two-dimensional image–plane representation that looks real.

Three methods of rendering are now firmly established. The first and most common method is to use a simulation of light–object interaction in conjunction with **polygon mesh** objects; we have called this approach *rendering polygon mesh objects*. Although the light–object simulation is independent of the object representation, the combination of empirical light–object interaction and polygon mesh representation has emerged as the most popular rendering technique in computer graphics. Because of its ubiquity and importance, we shall devote most of this chapter to this approach.

This approach to rendering suffers from a significant disadvantage. The reality of light–object interaction is simulated as a crude approximation — albeit an effective and cheap simulation. In particular, objects are considered to exist in isolation with respect to a light source or sources, and no account is taken of light interaction between objects themselves. In practice, this means that although we simulate the reflection of light incident on an object from a light source, we resolutely ignore the effects that the reflected light has on the scene when it travels onward from its first reflection to encounter, perhaps, other objects, and so on. Thus, common phenomena that depend on light reflecting from one object onto another, like shadows and objects reflecting in each other, cannot be produced by such a model. Such defects in straightforward polygon mesh rendering have led to the development of many and varied enhancements that attempt to address its shortcomings. Principal among these are mapping techniques (texture mapping, environment mapping, etc.) and various shadow algorithms.

Such models are called **local reflection models** to distinguish them from **global reflection models**, which attempt to follow the adventures of light emanating from a source as it hits objects, is reflected, hits other objects, and so on. The reason local reflection models work — in the sense that they produce visually acceptable, or even impressive, results — is that in reality the reflected light in a scene that emanates from first-hit incident light predominates. However, the subtle object–object interactions that one normally encounters in an environment are important. This motivation led to the development of the two global reflection models: **ray tracing** and **radiosity**.

Ray tracing simulates global interaction by explicitly tracking infinitely thin beams, or rays, of light as they travel through the scene from object to object. *Radiosity*, on the other hand, considers light reflecting in all directions from the surface of an object and calculates how light radiates from one surface to another as a function of the geometric relationship between surfaces — their proximity, relative orientation, etc. Ray tracing operates on points in the scene, radiosity on finite areas called patches.

Ray tracing and radiosity formed popular research topics in the 1980s. Both methods are much more expensive than polygon mesh rendering, and a common research motivation was efficiency, particularly in the case of ray tracing.

For reasons that will become clear later, ray tracing and radiosity each can simulate only one aspect of global interaction. Ray tracing deals with specular interaction and is fine for scenes consisting of shiny, mutually reflective objects. On the other hand, radiosity deals with diffuse or dull surfaces and is used mostly to simulate interiors of rooms. In effect, the two methods are mutually exclusive: ray tracing cannot simulate diffuse interaction, and radiosity cannot cope with specular interaction. This fact led to another major research effort, which was to incorporate specular interaction in the radiosity method.

Whether radiosity and ray tracing should be categorized as mainstream is perhaps debatable. Certainly the biggest demand for three-dimensional computer graphics is real-time rendering for computer games. Ray tracing and radiosity cannot be performed in real time on consumer equipment and, unless used in precalculation mode, are excluded from this application. However, radiosity in particular is used in professional applications, such as computer-aided architectural design.

38.2 Rendering Polygon Mesh Objects

38.2.1 Introduction

The overall process of rendering polygon mesh objects can be broken down into a sequence of geometric transformations and pixel processes that have been established for at least two decades as a *de facto* standard. Although they are not the only way to produce a shaded image of a three-dimensional object, the particular processes we shall describe represent a combination of popularity and ease of implementation. There is no established name for this group of processes, which has emerged for rendering objects represented by a polygon mesh — by far the most popular form of representation. The generic term *rendering pipeline* applies to any set of processes used to render objects in three-dimensional graphics.

Ignoring any transformations that are involved in positioning many objects to make up a scene — modeling transformations — we can summarize these processes as follows:

Viewing transformation — A process that is invoked to generate a representation of the object or scene as seen from the viewpoint of an observer positioned somewhere in the scene and looking toward some aspect of it. This involves a simple transformation that changes the object from its database representation to one that is represented in a coordinate system related to the viewer's position and viewing direction. It establishes the size of the object, according to its distance from the viewer, and the parts of it seen from the viewing direction.

Clipping — The need for clipping is easily exemplified by considering a viewpoint that is embedded among objects in the scene. Objects and parts of objects, for example, behind the viewer must be eliminated from consideration. Clipping is nontrivial because, in general, it involves removing parts of polygons and creating new ones. It means "cutting chunks" off the objects.

Projective transformation — This transformation generates a two-dimensional image on the image or viewing plane from the three-dimensional view-space representation of the object.

Shading algorithm — The orientation of the polygonal facets that represent the object are compared with the position of a light source (or sources), and a reflected light intensity is calculated for each point on the surface of the object. In practice, "each point on the surface" means those pixels onto which the polygonal facet projects. Thus, it is convenient to calculate the set of pixels onto which a polygon projects and to drive this process from pixel space — a process that is usually called *rasterization*. Shading algorithms use a local reflection model and an *interpolative method* to distribute the appropriate light intensity among pixels inside a polygon. The computational efficiency and visual efficacy of the shading algorithm have supported the popularity of the polygon mesh representation. (The polygon mesh representation has many drawbacks — its major advantage is simplicity.)

Hidden-surface removal — Those surfaces that cannot be seen from the viewpoint need to be removed from consideration. In the 1970s, much research was carried out on the best way to remove hidden surfaces, but the Z-buffer algorithm, with its easy implementation, is the *de facto* algorithm, with others being used only in specialized contexts. However, it does suffer from inefficiency and produces aliasing artifacts in the final image.

The preceding processes are not carried out in a sequence but are merged together in a way that depends on the overall rendering strategy. The use of the Z-buffer algorithm, as we shall see, conveniently allows polygons to be fetched from the database in any order. This means that the units on which the whole rendering process operates are single polygons that are passed through the processes one at a time. The entire process can be seen as a black box, with a polygon input as a set of vertices in three-dimensional world space. The output is a shaded polygon in two-dimensional screen space as a set of pixels onto which the polygon has been projected.

Although, as we have implied, the processes themselves have become a kind of standard, rendering systems vary widely in detail, particularly in differences among subprocesses such as rasterization and the kind of viewing system used.

The marriage of interpolative shading with the polygon mesh representation of objects has served, and continues to serve, the graphics community well. It does suffer from a significant disadvantage, which is that **antialiasing** measures are not easily incorporated in it (except by the inefficient device of calculating a virtual image at a resolution much higher than the final screen resolution). Antialiasing measures are described elsewhere in this text.

The first two processes, viewing transformation and clipping, are geometric processes that operate on the vertex list of a polygon, producing a new vertex list. At this stage, polygons are still represented by a list of vertices where each vertex is a coordinate in a three-dimensional space with an implicit link between vertices in the list. The projective transformation is also a geometric process, but it is embedded in the pixel-level processes. The shading algorithm and hidden-surface removal algorithm are pixel-level processes operating in screen space (which, as we shall see, is considered for some purposes to possess a third dimension). For these processes, the polygon becomes a set of pixels in two-dimensional space. However, some aspects of the shading algorithm require us to return to three-dimensional space. In particular, calculating light intensity is a three-dimensional calculation. This functioning of the shading algorithm in both screen space and a three-dimensional object space is the source of certain visual artifacts. These arise because the projective transformation is nonlinear. Such subtleties will not be considered here, but see [Watt and Watt, 1992] for more information on this point.

38.2.2 Viewing and Clipping

When viewing transformations are considered in computer graphics, an analogy is often made with a camera, and the term **virtual camera** is employed. There are certainly direct analogies to be made between a camera, which records a two-dimensional projection of a real scene on a film, and a computer graphics system. However, keep in mind that these concern external attributes, such as the position of the camera and the direction in which it is pointing. There are implementations in a computer graphics system (notably

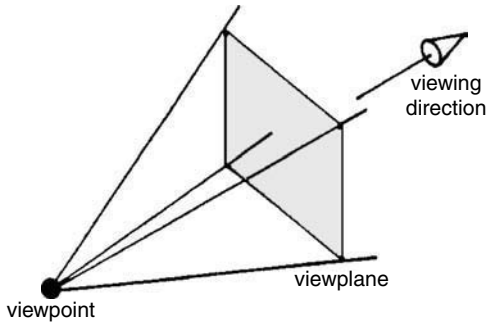


FIGURE 38.1 The three basic attributes required in a viewing system.

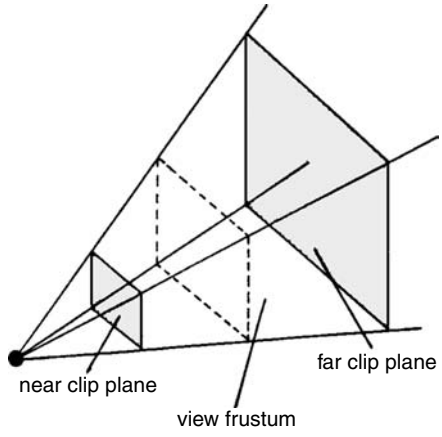


FIGURE 38.2 View frustum formed by near and far planes.

the near and far clip planes) that are not available in a camera and facilities in a camera that are not usually imitated in a computer graphics system (notably depth of field and lens distortion effects). The analogy is a general one, and its utility disappears when details are considered.

The facilities incorporated into a viewing system can vary widely. In this section, we will look at a system that will suffice for most general-purpose rendering systems. The ways in which the attributes of the viewing system are represented, which have ramifications for the design of a user interface for a viewing system, also vary widely.

We can discuss our requirements by considering just three attributes (see Figure 38.1). First, we establish a viewpoint and a viewing direction. A *viewpoint*, which we can also use as a center of projection, is a single point in world space, and a *viewing direction* is a vector in this space.

Second, we position a view plane somewhere in this space. It is convenient to constrain the view plane normal to be the viewing direction and its position to be such that a line from the viewpoint, in the viewing direction, passes through the center of the view plane. The distance of the viewpoint from the object being viewed and the distance of the view plane from the viewpoint determine the size of the projection of the object on the view plane. Also, these two distances determine the degree of perspective effect in the object projection, since we will be using a perspective projection. This arrangement defines a new three-dimensional space, known as *view space*. Normally, we would take the origin of this space to be the viewpoint, and the coordinate axes are oriented by the viewing direction.

Usually, we assume that the view plane is of finite extent and rectangular, because its contents will eventually be mapped onto the display device. Additionally, we can add *near* and *far clip planes* (Figure 38.2)

to constrain further those elements of the scene that are projected onto the view plane — a caprice of computer graphics not available in a camera.

Such a setup, as can be seen in Figure 38.2, defines a so-called *view volume*, and consideration of this gives the motivation for clipping. Clipping means that the part of the scene that lies outside the view frustum should be discarded from the rendering process. We perform this operation in three-dimensional view space, clipping polygons to the view volume. This is a nontrivial operation, but it is vital in scenes of any complexity where only a small proportion of the scene will finally appear on the screen. In simple single-object applications, where the viewpoint will not be inside the bounds of the scene and we do not implement a near and a far clip plane, we can project all the scene onto the view plane and perform the clipping operation in two-dimensional space.

Now we are in a position to define **viewing** and **clipping** as those operations that transform the scene from world space into view space, at the same time discarding that part of the scene or object that lies outside the view frustum.

We will deal separately with the transformation into the view space and clipping. First, we consider the viewing transformation. A useful practical facility that we should consider is the addition of another vector to specify the rotation of the view plane about its axis (the view-direction vector). Returning to our camera analogy, this is equivalent to allowing the user to rotate the camera about the direction in which it is pointing. A user of such a system must specify the following:

1. A viewpoint or camera position **C**, which forms the origin of view space. This point is also the center of projection (see Section 38.2.4).
2. A viewing direction vector **N** (the positive z-axis in view space) — this is a vector normal to the view plane.
3. An "up" vector **V** that orients the camera about the view direction.
4. An optional vector **U**, to denote the direction of increasing x in the eye coordinate system. This establishes a right- or left-handed coordinate system (**UVN**). This system is represented in Figure 38.3.

The transformation required to take an object from world space into view space, T_{view}, can be split into a translation T and a change of basis B:

$$T_{\text{view}} = TB$$

where

$$T = \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

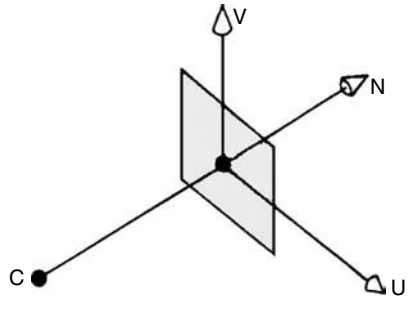


FIGURE 38.3 **UVN** coordinate system embedded in the view plane.

It can be shown [Fiume, 1989] that B is given by

$$B = \begin{bmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The only problem now is specifying a user interface for the system and mapping whatever parameters are used by the interface into **U**, **V**, and **N**. A user needs to specify **C**, **N**, and **V**. **C** is easy enough. **N**, the viewing direction or view plane normal, can be entered, say, by using two angles in a spherical coordinate system. **V** is more problematic. For example, a user may require "up" to have the same sense as "up" in the world coordinate system. However, this cannot be achieved by setting

$$\mathbf{V} = (0, 0, 1)$$

because **V** must be perpendicular to **N**. A useful strategy is to allow the user to specify, through a suitable interface, an approximate value for **V**, having the program alter this to a correct value.

38.2.3 Clipping and Culling

Clipping and culling mean discarding, at an early stage in the rendering process, polygons or parts of polygons that will not appear on the screen. Polygons that must be discarded fall into three categories:

1. Complete objects that lie outside the view volume should be removed entirely without invoking any tests at the level of individual polygons. This can be done by comparing a bounding volume, such as a sphere, with the view-volume extents and removing (or not) the entire set of polygons that represent an object.
2. Complete polygons that face away from a viewer need not invoke the expense of a clipping procedure. These are called *back-facing polygons*, and in any (convex) object they account, on average, for 50% of the object polygons. These can be eliminated by a simple geometric test, which is termed **culling** or *back-face removal*.
3. Polygons that straddle a view-volume plane must be clipped against the view frustum and the resulting fragment, which comprises a new polygon, passed on to the remainder of the process.

Clipping is carried out in the three-dimensional domain of view space; culling is performed in this space, also. Culling is a pure geometric operation that discards polygons on the basis of the direction of their surface normal compared to the viewing direction. Clipping is an algorithmic operation, because some process must be invoked that produces a new polygon from the polygon that is clipped by one of the view-volume planes.

We deal with the simple operation of culling first. If we are considering a single convex object, then culling performs complete hidden-surface removal. If we are dealing with objects that are partially concave, or if there is more than one object in the scene, then a general hidden-surface removal algorithm is required. In these cases, the event of one polygon partially obscuring another arises — a situation impossible with a single convex object.

Determining whether a single polygon is visible from a viewpoint involves a simple geometric test (see Figure 38.4). We compare the angle between the (true) normal of each polygon and a line-of-sight vector. If this is greater than 90°, then the polygon cannot be seen. This condition can be written as

$$\text{visibility} := \mathbf{N}_p \cdot \mathbf{L}_{os} > 0$$

where $\mathbf{N}_p$ is the polygon normal and $\mathbf{L}_{os}$ is a vector representing a line from one vertex of the polygon to the viewpoint. A polygon normal can be calculated from any three (noncollinear) vertices by taking a cross product of vectors parallel to the two edges defined by the vertices.

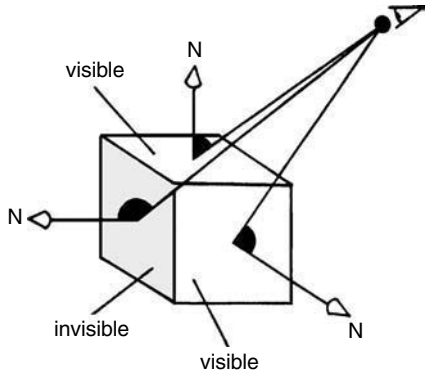


FIGURE 38.4 Back-face removal or culling.

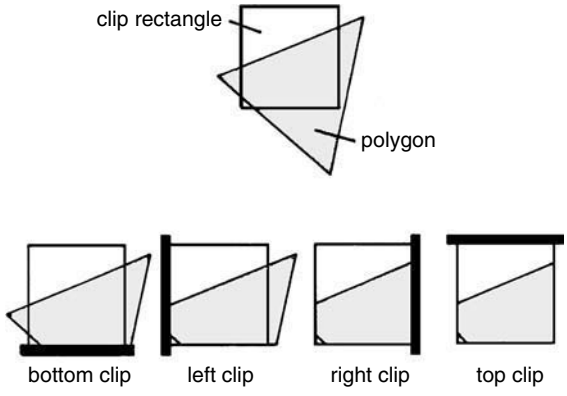


FIGURE 38.5 Sutherland–Hodgman clipper clips each polygon against each edge of each rectangle.

The most popular clipping algorithm, like most of the algorithms used in rendering, goes back over 25 years and is the Sutherland–Hodgman reentrant polygon clipper [Sutherland and Hodgman, 1974]. We will describe, for simplicity, its operation in two-dimensional space, but it is easily extended to three dimensions. A polygon is tested against a clip boundary by testing each polygon edge against a single infinite clip boundary. This structure is shown in Figure 38.5.

We consider the innermost loop of the algorithm, where a single edge is being tested against a single clip boundary. In this step, the process outputs zero, one, or two vertices to add to the list of vertices defining the clipped polygon. Figure 38.6 shows the four possible cases. An edge is defined by vertices **S** and **F**. In the first case, the edge is inside the clip boundary and the existing vertex **F** is added to the output list. In the second case, the edge crosses the clip boundary and a new vertex **I** is calculated and output. The third case shows an edge that is completely outside the clip boundary. This produces no output. (The intersection for the edge that caused the excursion outside is calculated in the previous iteration, and the intersection for the edge that causes the incursion inside is calculated in the next iteration.) The final case again produces a new vertex, which is added to the output list.

To calculate whether a point or vertex is inside, outside, or on the clip boundary, we use a dot-product test. Figure 38.7 shows clip boundary **C** with an outward normal $\mathbf{N}_c$ and a line with end points **S** and **F**. We represent the line parametrically as

$$\mathbf{P}(t) = \mathbf{S} + (\mathbf{F} - \mathbf{S})t \tag{38.1}$$

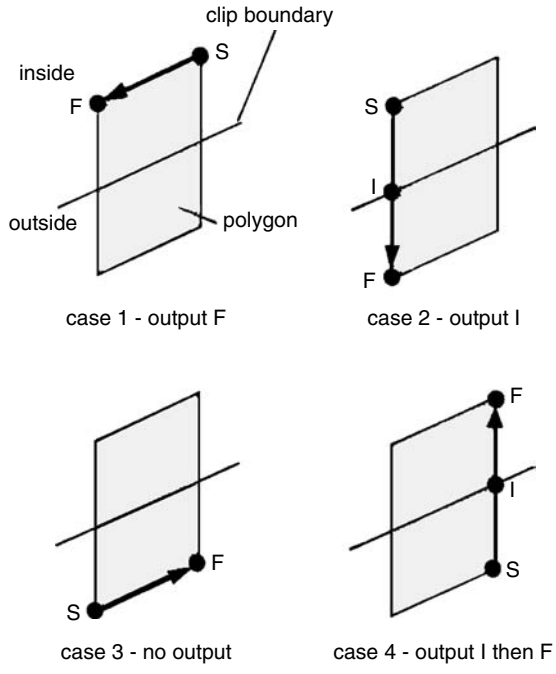


FIGURE 38.6 Sutherland–Hodgman clipper: within the polygon loop, each edge of a polygon is tested against each clip boundary.

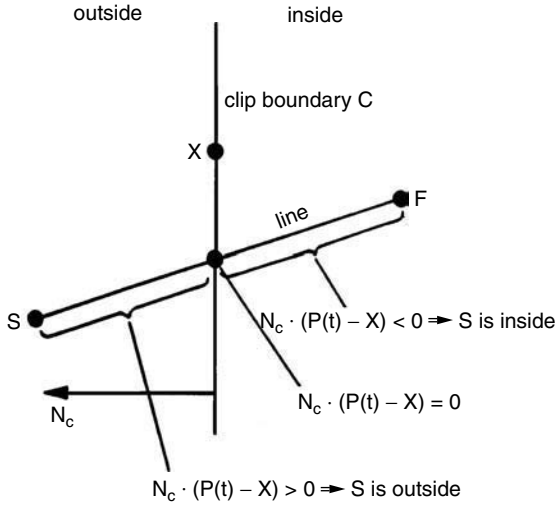


FIGURE 38.7 Dot-product test determines whether a line is inside or outside a clip boundary.

where

$$0 \leq t \leq 1$$

We define an arbitrary point on the clip boundary as **X** and consider a vector from **X** to any point on the line. The dot product of this vector and the normal allows us to distinguish whether a point on the line is

outside, inside, or on the clip boundary. In the case shown in Figure 38.7,

$$\mathbf{N}_c \cdot (\mathbf{S} - \mathbf{X}) > 0 \Rightarrow \mathbf{S} \text{ is outside the clip region}$$
$$\mathbf{N}_c \cdot (\mathbf{F} - \mathbf{X}) < 0 \Rightarrow \mathbf{F} \text{ is inside the clip region}$$

and

$$\mathbf{N}_c \cdot (\mathbf{P}(t) - \mathbf{X}) = 0$$

defines the point of intersection of the line and the clip boundary. Solving Equation 38.1 for t enables the intersecting vertex to be calculated and added to the output list.

In practice, the algorithm is written recursively. As soon as a vertex is output, the procedure calls itself with that vertex, and no intermediate storage is required for the partially clipped polygon. This structure makes the algorithm eminently suitable for hardware implementation.

A projective transformation takes the object representation in view space and produces a projection on the view plane. This is a fairly simple procedure, somewhat complicated by the fact that we must retain a depth value for each point for eventual use in the hidden-surface removal algorithm. Sometimes, therefore, the space of this transformation is referred to as *three-dimensional screen space*.

38.2.4 Projective Transformation and Three-Dimensional Screen Space

A perspective projection is the more popular or common choice in computer graphics because it incorporates foreshortening. In a perspective projection, relative dimensions are not preserved, and a distant line is displayed smaller than a nearer line of the same length. This familiar effect enables human beings to perceive depth in a two-dimensional photograph or a stylization of three-dimensional reality. A perspective projection is characterized by a point known as the *center of projection*, the same point as the viewpoint in our discussion. The projection of three-dimensional points onto the view plane is the intersection of the lines from each point to the center of projection. This familiar idea is shown in Figure 38.8.

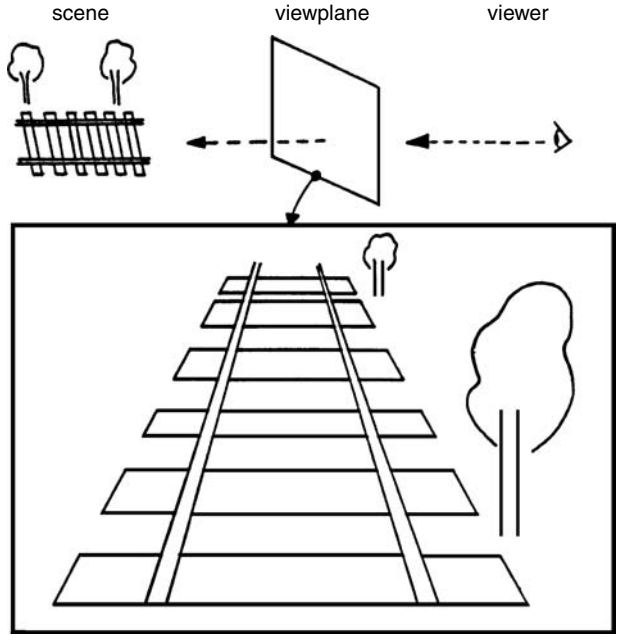


FIGURE 38.8 The perspective effect.

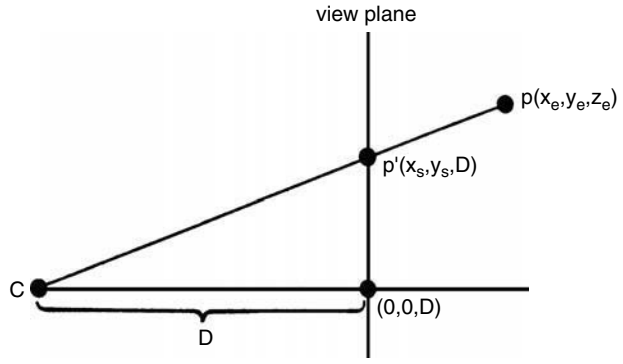


FIGURE 38.9 Perspective projection.

Figure 38.9 shows how a perspective projection is derived. Point $P(x_e, y_e, z_e)$ is a three-dimensional point in the view coordinate system. This point is to be projected onto a view plane normal to the z_e-axis and positioned at distance D from the origin of this system. Point P′ is the projection of this point in the view plane. It has two-dimensional coordinates (x_s, y_s) in a view-plane coordinate system, with the origin at the intersection of the z_e-axis and the view plane. In this system, we consider the view plane to be the view surface or screen.

Similar triangles give

$$x_s = D(x_e/z_e)$$
$$y_s = D(y_e/z_e)$$

Screen space is defined to act within a closed volume — the viewing frustum that delineates the volume of space to be rendered. For the purposes of this chapter, we will consider a simplified view volume that constrains some of the dimensions that would normally be found in a more general viewing system. A simple view volume can be specified as follows: suppose we have a square window — that area of the view plane that is mapped onto the view surface or screen — of size 2h, arranged symmetrically about the viewing direction. The four planes defined by

$$x_e = \pm h(z_e/D)$$
$$y_e = \pm h(z_e/D)$$

together with the two additional planes, called the near and far clipping planes, respectively (perpendicular to the viewing direction), which are defined by

$$z_e = D$$
$$z_e = F$$

make up the definition of the viewing frustum, as shown in Figure 38.10. Additionally, we invoke the constraint that the view plane and the near clip plane are to be coincident. This simple system is based on a treatment given in an early, classic textbook on computer graphics [Newman and Sproull, 1973].

This deals with transforming the x and y coordinates of points in view space. We shall now discuss the transformation of the third component of screen space, namely z_e. In order to perform hidden-surface calculations (in the Z-buffer algorithm), depth information must be generated on arbitrary points, in practice pixels, within the polygon by interpolation. This is possible in screen space only if, in moving

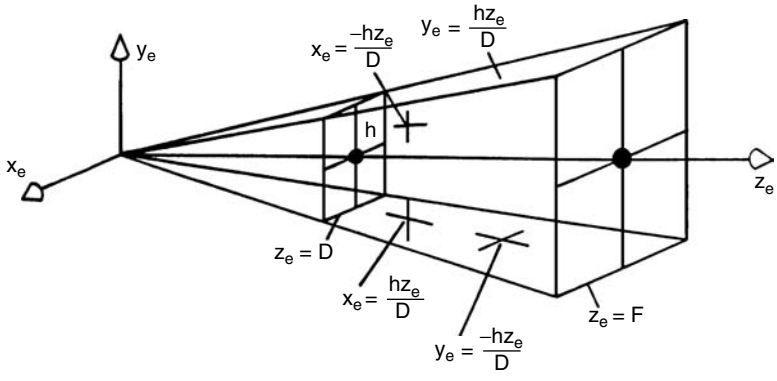


FIGURE 38.10 The six planes that define the view frustum.

from view space to screen space, lines transform into lines and planes transform into planes. It can be shown [Newman and Sproull, 1973] that these conditions are satisfied, provided the transformation of z takes the form

$$z_s = A + (B/z_e)$$

where A and B are constants. These constants are determined from the following constraints:

1. Choosing B < 0 so that as z_e increases, so does z_s. This preserves depth. If one point is behind another, then it will have a larger z_e-value; if B < 0, it will also have a larger z_s-value.
2. Normalizing the range of z_s-values so that the range z_e in [D, F] maps into the range z_s in [0, 1]. This is important to preserve accuracy, because a pixel depth will be represented by a fixed number of bits in the Z-buffer.

The full perspective transformation is then given by

$$x_s = D(x_e/(hz_e))$$
$$y_s = D(y_e/(hz_e))$$
$$z_s = F(1 - D/z_e)/(F - D)$$

where the additional constant h appearing in the transformation for x_s and y_s ensures that these values fall in the range $[-1, 1]$ over the square screen.

It is instructive to consider the relationship between z_e and z_s a little more closely; although, as we have seen, they both provide a measure of the depth of a point, interpolating along a line in eye space is not the same as interpolating along this line in screen space. As z_e approaches the far clipping plane, z_s approaches 1 more rapidly. Objects in screen space thus get pushed and distorted toward the back of the viewing frustum. This difference can lead to errors when interpolating quantities, other than position, in screen space.

38.2.5 Shading Algorithm

This part of the process calculates a light intensity for each pixel onto which a polygon projects. The input information to this process is the vertex list of the polygon, transformed into eye or view space, and the process splits into two subprocesses:

1. First we find the set of pixels that make up our polygon. We change the polygon from a vertex list into a set of pixels in screen space — through *rasterization*.

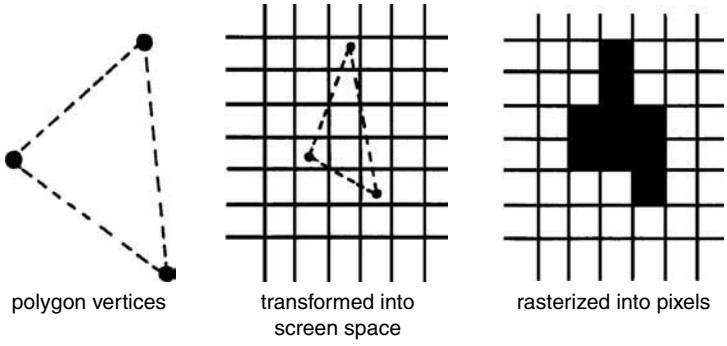


FIGURE 38.11 Different representations of a polygon in the rendering process.

2. Second we find a light intensity associated with each pixel. This is done by associating a local reflection model with each vertex of the polygon and calculating an intensity for the vertex. These vertex intensities are then used in a *bilinear interpolation* scheme to find the intensity of each pixel. By this process, we are finding the intensity of that part of the object surface that corresponds to the pixel area in screen space. The particular way in which this is done leads to the (efficiency–quality) hierarchy of **flat shading, Gouraud shading**, and **Phong shading**.

38.2.5.1 Rasterization

Rasterization, or finding the set of polygons onto which the polygon projects, must be done carefully because adjacent polygons must fit together accurately after they have been mapped into pixel sets. If it is not done accurately, holes can result in the image — probably the most common defect seen in rendering software.

As shown in Figure 38.11, the precise geometry of the polygon will map into a set of fully and partially covered pixels. We must decide which of the partially covered pixels are to be considered part of the polygon. Deciding on the basis of the area of coverage is extremely expensive and would wipe out the efficiency advantage of the bilinear interpolation scheme that is used to find a pixel intensity. It is better to map the vertices in some way to the nearest pixel coordinate and set up a consistent rule for deciding the fate of partially covered pixels. This is the crux of the matter. If the rules are not consistent and carefully formulated, then the rounding process will produce holes or unfilled pixels between polygons that share the same scan line. Note that the process will cause a shape change in the polygon, which we ignore because the polygon is already an approximation, and to some extent an arbitrary representation of the "real" surface of the object.

Sometimes called *scan-line conversion*, rasterization proceeds by moving a horizontal line through the polygon in steps of a pixel height. For a current scan line, interpolation (see Section 38.2.5.2) between the appropriate pairs of polygon vertices will yield x_{start} and x_{end}, the start and end points of the portions of the scan line crossing the polygons (using real arithmetic). The following scheme is a simple set of rules that converts these values into a run of pixels:

1. Round x_{start} up.
2. Round x_{end} down.
3. If the fractional part of x_{end} is 0, then subtract 1 from it.

Applying the rasterization process to a complete polygon implies embedding this operation in a structure that keeps track of the edges that are to be used in the current scan-line interpolation. This is normally implemented using a linked-list approach.

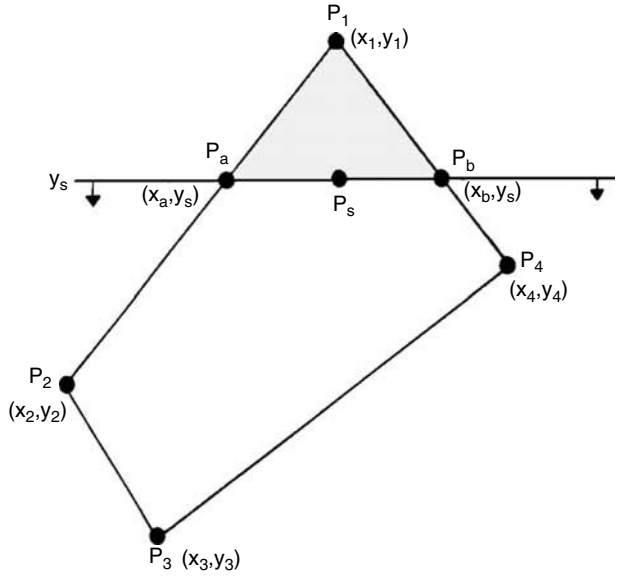


FIGURE 38.12 Notation used in property interpolation within a polygon.

38.2.5.2 Bilinear Interpolation

As we have already mentioned, light intensity values are assigned to the set of pixels that we have now calculated, not by individual calculation but by interpolating from values calculated only at the polygon vertices. At the same time, we interpolate depth values for each pixel to be used in the hidden-surface determination. So in this section, we consider the interpolation of a pixel property from vertex values independent of the nature of the property. Referring to Figure 38.12, the interpolation proceeds by moving a scan line down through the pixel set and obtaining start and end values for a scan line by interpolating between the appropriate pair of vertex properties. Interpolation along a scan line then yields a value for the property at each pixel. The interpolation equations are

$$\begin{aligned} p_a &= \frac{1}{y_1 - y_2}\,[p_1(y_s - y_2) + p_2(y_1 - y_s)] \\ p_b &= \frac{1}{y_1 - y_4}\,[p_1(y_s - y_4) + p_4(y_1 - y_s)] \\ p_s &= \frac{1}{x_b - x_a}\,[p_a(x_b - x_s) + p_b(x_s - x_a)] \end{aligned} \tag{38.2}$$

These would normally be implemented using an incremental form, the final equation, for example, becoming

$$p_s := p_s + \Delta p$$

with the constant value Δp calculated once per scan line.

38.2.5.3 Local Reflection Models

Given that we find pixel intensities by an interpolation process, the next thing to discuss is how to find the reflected light intensity at the vertices of a polygon, those values from which the pixel intensity values are derived. This is done by using a simple local reflection model — the one most commonly used is the Phong reflection model [Phong, 1975]. (This reflection model is not to be confused with Phong shading, which is a vector interpolation scheme.)

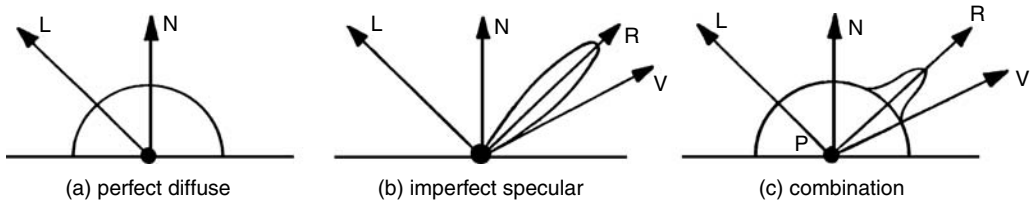


FIGURE 38.13 Components of the Phong local reflection model.

A local reflection model calculates a value for the reflected light intensity at a point on the surface of an object — in this case, that point is a polygon vertex — due to incident light from a source, which for reasons we will shortly examine is usually a point light source. The model is a linear combination of three components: **diffuse, specular**, and **ambient**. We assume that the behavior of reflected light at a point on a surface can be simulated by assuming that the surface is some combination of a perfect diffuse surface together with an (imperfect) specular or mirrorlike surface. The light scattered from a perfect diffuse surface is the same in all directions, and the reflected light intensity from such a surface is given by Lambert's cosine law, which is, in computer graphics notation

$$I_d = I_i k_d \mathbf{L} \cdot \mathbf{N}$$

where **L** is the light direction vector and both **L** and **N** are unit vectors, as shown in Figure 38.13a; k_d is a diffuse reflection coefficient; and I_i is the intensity of a (point) light source. The specular contribution is a function of the angle between the viewing direction **V** and the mirror direction **R**

$$I_s = I_i k_s (\mathbf{R} \cdot \mathbf{V})^n$$

where n is an index that simulates surface roughness and k_s is a specular reflection coefficient.

For a perfect mirror, n would be infinity and reflected light would be constrained to the mirror direction. For small integer values of n, a reflection lobe is generated, where the thickness of the lobe is a function of the surface roughness (see Figure 38.13b). The effect of the specular reflection term in the model is to produce a so-called highlight on the rendered object. This is basically a reflection of the light source spread over an area of the surface to an extent that depends on the value of n. The color of the specularly reflected light is different from that of the diffuse reflected light — hence the term *highlight*. In simple models of specular reflection, the specular component is assumed to be the color of the light source. If, say, a green surface were illuminated with white light, then the diffuse reflection component would be green, but the highlight would be white.

Adding the specular and diffuse components gives a very approximate imitation to the behavior of reflected light from a point on the surface of an object. Consider Figure 38.13c. This is a cross section of the overall reflectivity response as a function of the orientation of the view vector **V**. The cross section is in a plane that contains the vector **L** and the point **P**; thus, it slices through the specular bump. The magnitude of the reflected intensity, the sum of the diffuse and specular terms, is the distance from **P** along the direction **V** to where **V** intersects the profile.

An ambient component is usually added to the diffuse and specular terms. Such a component illuminates surfaces that, because we generally use a point light source, would otherwise be rendered black. These are surfaces that are visible from the viewpoint but not from the light source. Essentially, the ambient term is a constant that attempts to simulate the global interreflection of light between surfaces.

Adding the diffuse, specular, and ambient components (Equation 38.3), we have

$$I = k_a + I_i (k_d \mathbf{L} \cdot \mathbf{N} + k_s (\mathbf{R} \cdot \mathbf{V})^n) \tag{38.3}$$

where k_a is the constant ambient term. The expense of Equation 38.3 can be considerably reduced by making some geometric assumptions and approximations. First, if the light source and the viewpoint

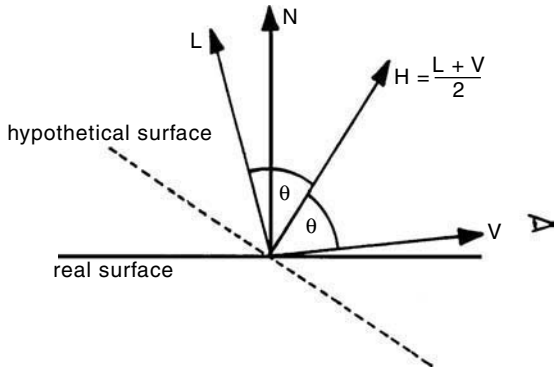


FIGURE 38.14 **H** is the unit normal to a hypothetical surface oriented in a direction that bisects the angle between **L** and **V**.

are considered to be at infinity, then **L** and **V** are constant over the domain of the scene. The vector **R** is expensive to calculate; although Phong gives an efficient method for calculating **R**, it is better to use a vector **H**. This appears to have been first introduced by Blinn [1977]. The specular term then becomes a function of **N**·**H** rather than **R**·**V**. **H** is the unit normal to a hypothetical surface that is oriented in a direction halfway between the light direction vector **L** and the viewing vector **V** (see Figure 38.14).

$$\mathbf{H} = (\mathbf{L} + \mathbf{V})/2$$

Together with a shading algorithm, this simple model is responsible for the look of most shaded computer graphics images, and it has been in use constantly since 1975. Its main disadvantages are that objects look as if they were made from some kind of plastic material, which is either shiny or dull. Also, in reality the magnitude of the specular component is not independent, as Equation 38.3 implies, of the direction of the incoming light. Consider, for example, the glare from a (nonshiny) road surface when you are driving in the direction of the setting sun: this does not occur when the sun is overhead.

We should now return to the term *local*. The reflection model is called a local model because it considers that the point on the surface under consideration is illuminated directly by the light source in the scene. No other (indirect) source of illumination is taken into account. Light reflected from nearby objects is ignored, so we see no reflections of neighboring objects in the object under consideration. It also means that shadows, which are areas that cannot "see" the light source and which receive their illumination indirectly from another object, cannot be modeled. In a scene using this model, objects are illuminated as if they were floating in a dark space illuminated only by the light source.

When shadows are added into rendering systems that use local reflection models, these are purely geometric. That is, the area that the shadow occupies on the surface of an object, due to the intervention of another object between it and the light source, is calculated. The reflected light intensity within this area is then arbitrarily reduced. When using local reflection models, there is no way to calculate how much indirect light illuminates the shadowed area. The visual consequences of this should be considered when including shadows in an add-on manner. These may detract from the real appearance of the final rendered image rather than add to it. First, because shadows are important to us in reality, we easily spot shadows in computer graphics that have the wrong intensity. This is compounded by the hard-edged shadow boundaries calculated by geometric algorithms. In reality, shadows normally have soft, subtle edges.

Finally, we briefly consider the role of color. For colored objects, the easiest approach is to model the specular highlights as white (for a white light source) and to control the color of the objects by appropriate setting of the diffuse reflection coefficients. We use three intensity equations to drive the monitor's red,

green, and blue inputs:

$$I_r = k_a + I_i(k_{dr}\mathbf{L}\cdot\mathbf{N} + k_s(\mathbf{N}\cdot\mathbf{H})^n)$$
$$I_g = k_a + I_i(k_{dg}\mathbf{L}\cdot\mathbf{N} + k_s(\mathbf{N}\cdot\mathbf{H})^n)$$
$$I_b = k_a + I_i(k_{db}\mathbf{L}\cdot\mathbf{N} + k_s(\mathbf{N}\cdot\mathbf{H})^n)$$

where the specular coefficient k_s is common to all three equations, but the diffuse component varies according to the object's surface color.

This three-sample approach to color is a crude approximation. Accurate treatment of color requires far more than three samples. This means that to model the behavior of reflected light accurately, we would have to evaluate many more than three equations. We would have to sample the spectral energy distribution of the light source as a function of wavelength and the reflectivity of the object as a function of wavelength and apply Equation 38.3 at each wavelength sample. The solution then obtained would have to be converted back into three intensities to drive the monitor. The colors that we would get from such an approach would certainly be different from the three-sample implementation. Except in very specialized applications, this problem is completely ignored.

We now discuss shading options. These options differ in where the reflection model is applied and how calculated intensities are distributed among pixels. There are three options: flat shading, Gouraud shading, and Phong shading, in order of increasing expense and increasing image quality.

38.2.5.4 Flat Shading

Flat shading is the option in which we invoke no interpolation within a polygon and shade each pixel within the polygon with the same intensity. The reflection model is used once only per polygon. The (true) normal for the polygon (in eye or view space) is inserted into Equation 38.3, and the calculated intensity is applied to the polygon. The efficiency advantages are obvious — the entire interpolation procedure is avoided, and shading reduces to rasterization plus a once-only intensity calculation per polygon. The (visual) disadvantage is that the polygon edges remain glaringly visible, and we render not the surface that the polygon mesh represents but the polygon mesh itself. As far as image quality is concerned, this is more disadvantageous than the fact that there is no variation in light intensity among the polygon pixels. Flat shading is used as a fast preview facility.

38.2.5.5 Gouraud Shading

Both Gouraud and Phong shadings exhibit two strong advantages — in fact, these advantages are their *raison d'être*. Both use the interpolation scheme already described and so are efficient, and they diminish or eliminate the visibility of the polygon edges. In a Gouraud- or Phong-shaded object, these are now visible only along silhouette edges. This elegant device meant their enduring success; the idea originated by Gouraud [1971] and cleverly elaborated by Phong [1975] was one of the major breakthroughs in three-dimensional computer graphics.

In Gouraud shading, intensities are calculated at each vertex and inserted into the interpolation scheme. The trick is in the normals used at a polygon vertex. Using the true polygon normal would not work, because all the vertex normals would be parallel and the reflection model would evaluate the same intensity at each. What we must do is calculate a normal at each vertex that somehow relates back to the original surface. Gouraud vertex normals are calculated by considering the average of the true polygon normals of those polygons that contribute to the vertex (see Figure 38.15). This calculation is normally regarded as part of the setting up of the object, and these vectors are stored as part of the object database (although there is a problem when polygons are clipped: new vertex normals then must be calculated as part of the rendering process). Because polygons now share vertex normals, the interpolation process ensures that there is no change in intensity across the edge between two polygons; in this way, the polygonal structure of the object representation is rendered invisible. (However, an optical illusion, known as *Mach banding*, persists along the edges with Gouraud shading.)

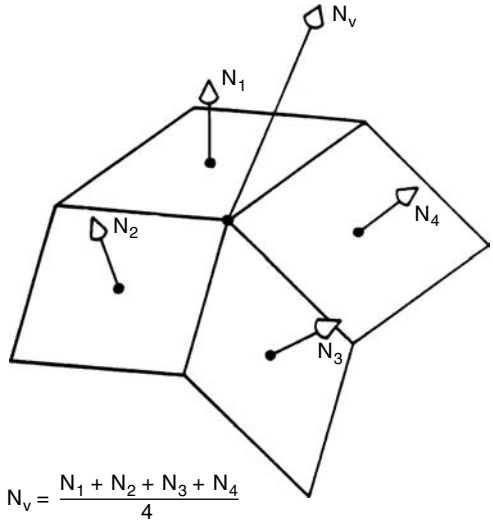


FIGURE 38.15 The concept of a vertex normal.

Gouraud shading is used extensively and gives excellent results for the diffuse component. However, calculating reflected light intensity only at the vertices leads to problems with the specular component. The easiest case to consider is that of a highlight which, if it were visible, would be within the polygon boundaries — meaning it does not extend to the vertices. In this case, the Gouraud scheme would simply miss the highlight completely.

38.2.5.6 Phong Shading

Phong shading [Phong, 1975] was developed to overcome the problems of Gouraud shading and specular highlights. In this scheme, the property to be interpolated is the vertex normals themselves, with each vector component now inserted into three versions of Equation 38.3. It is a strange hybrid, with an interpolation procedure running in pixel or screen space controlling vector interpolation in three-dimensional view space (or world space). But it works very well.

We estimate the normal to the surface at a point that corresponds to the pixel under consideration in screen space, or at least estimate it to within the limitations and approximations that have been imposed by the polygonal representation and the interpolation scheme. We can then apply the reflection model at each pixel, and a unique reflected light intensity is now calculated for each pixel. We may end up with a result that is different from what would be obtained if we had access to the true surface normal at the point on the real surface that corresponded to the pixel, but it does not matter, because the quality of Phong shading is so good that we cannot perceive any erroneous effects on the monitor.

Phong shading is much slower than Gouraud shading because the interpolation scheme is three times as lengthy, and also the reflection model (Equation 38.3) is now applied at each pixel. A good rule of thumb is that Phong shading has five times the cost of Gouraud shading.

38.2.6 Hidden-Surface Removal

As already mentioned, we shall describe the Z-buffer as the *de facto* hidden-surface removal algorithm. That it has attained this status is due to its ease of implementation — it is virtually a single *if* statement — and its ease of incorporation into a polygon-based renderer. Screen space algorithms (the Z-buffer falls into this category) operate by associating a depth value with each pixel. In our polygon renderer, the

depth values are available only at a vertex, and the depth values for a pixel are obtained by using the same interpolation scheme as for intensity in Gouraud shading.

Hidden-surface removal eventually comes down to a point-by-point depth comparison. Certain algorithms operate on area units, scan-line segments, or even complete polygons, but they must contain a provision for the worst case, which is a depth comparison between two pixels. The Z-buffer algorithm performs this comparison in three-dimensional screen space. We have already defined this space and we repeat, for convenience, the equation for Z_s:

$$z_s = F(1 - D/z_e)/(F - D)$$

The Z-buffer algorithm is equivalent, for each pixel (x_s, y_s), to a search through the associated z-values of every polygon point that projects onto that pixel to find that point with the minimum z-value — the point nearest the viewer. This search is conveniently implemented by using a Z-buffer, which holds for each pixel the smallest z-value so far encountered. During the processing of a polygon, we either write the intensity of a pixel into the frame buffer or not, depending on whether the depth of the current pixel is less than the depth so far encountered as recorded in the Z-buffer.

Apart from its simplicity, another advantage of the Z-buffer is that it is independent of object representation. Although we see it used most often in the context of polygon mesh rendering, it can be used with any representation: all that is required is the ability to calculate a z-value for each point on the surface of an object. If the z-values are stored with pixel values, separately rendered objects can be merged into a multiple-object scene using Z-buffer information on each object.

The main disadvantage of the Z-buffer is the amount of memory it requires. The size of the Z-buffer depends on the accuracy to which the depth value of each point (x, y) is to be stored, which is a function of scene complexity. Usually, 20 to 32 bits is deemed sufficient for most applications. Recall our previous discussion of the compression of z_s-values. This means that a pair of distinct points with different z_e-values can map into identical z_s-values. Note that for frame buffers with less than 24 bits per pixel, say, the Z-buffer will in fact be *larger* than the frame buffer. In the past, Z-buffers have tended to be part of the main memory of the host processor, but now graphics cards are available with dedicated Z-buffers. This represents the best solution.

38.3 Rendering Using Ray Tracing

Ray tracing is a simple and elegant algorithm whose appearance in computer graphics is usually attributed to Whitted [1980]. It combines in a single algorithm

Hidden-surface removal
Reflection due to direct illumination (the same factor we calculated in the previous method using a local model)
Reflection due to indirect illumination (i.e., reflection due to light striking) — the object which itself has been reflected from another object
Transmission of light through transparent or partially transparent objects
Shading due to object–object interaction (global illumination)
The computation of (hard-edged) shadows

It does this by tracing rays — infinitesimally thin beams of light — in the reverse direction of light propagation; that is, it traces light rays from the eye into the scene and from object to object. In this way, it "discovers" the way in which light interacts between objects and can produce visualizations such as objects reflecting in other objects and the distortion of an object viewed through another (transparent or glass) object due to refraction. Rays are traced from the eye or viewpoint, because we are interested only in those rays that pass through the view plane. If we traced rays from the light source, then theoretically we would have to trace an infinity of rays.

Ray-tracing algorithms exhibit a strong visual signature because a basic ray tracer can simulate only one aspect of the global interaction of light in an environment: specular reflection and specular transmission. Thus ray-traced scenes always look ray-traced, because they tend to consist of objects that exhibit mirrorlike reflection, in which you can see the perfect reflections of other objects. Simulating nonperfect specular reflection is computationally impossible with the normal ray-tracing approach, because this means that at a hit point a single incoming ray will produce a multiplicity of reflected rays instead of just one. The same argument applies to transparent objects. A single incident ray can produce only a single transmitted or refracted ray. Such behavior would happen only in a perfect material that did not scatter light passing through it. With transparent objects, the refractive effect can be simulated, but the material looks like perfect glass. Thus, perfect surfaces and perfect glass, behavior that does not occur in practice, betray the underlying rendering algorithm.

A famous development, called *distributed ray tracing* [Cook et al., 1984], addressed exactly this problem, using a Monte Carlo approach to simulate the specular reflection and specular transmission spread without invoking a combinatorial explosion. The algorithm produces shiny objects that look real (i.e., their surfaces look rough or imperfect), blurred transmission through glass, and blurred shadows. The modest cost of this method involved initiating 16 rays per pixel instead of one. This is still a considerable increase in an already expensive algorithm, and most ray tracers still utilize the perfect specular interaction model.

The algorithm is conceptually easy to understand and is also easy to implement using a recursive procedure. A pictorial representation is given in Figure 38.16. The algorithm operates in three-dimensional

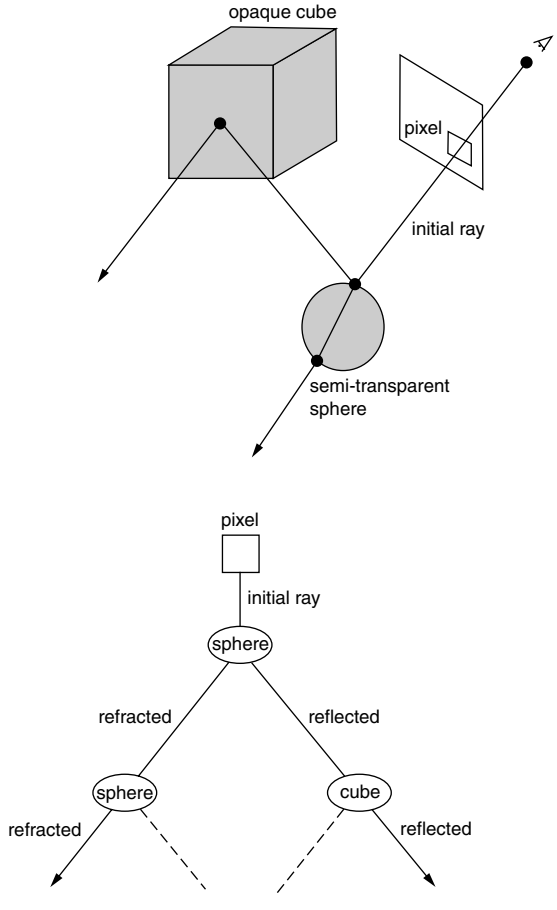


FIGURE 38.16 A representation of a ray-tracing algorithm.

world space, and for each pixel in screen space we calculate an initial ray from the viewpoint through the center of the pixel. The ray is injected into the scene and will either hit an object or not. (In the case of a closed environment, some object will always be encountered by an initial ray, even if it is just the background, such as a wall.) When the ray hits an object, it spawns two more rays: a reflected ray and a transmitted ray, which refracts into the object if the object is partially transparent. These rays travel onward and themselves spawn other rays at their next hits. The process is sometimes represented as a binary tree, with a light–surface hit at each node in the tree.

This process can be implemented as a recursive procedure, which for each ray invokes an intersection test that spawns a transmitted and a reflected ray by calling itself twice with parameters representing the reflected and the transmitted or refracted directions of the new rays. At the heart of the recursive control procedure is an intersection test. This procedure is supplied with a ray, compares the ray geometry with all objects in the scene, and returns the nearest surface that the ray intersects. If the ray is an initial ray, then this effectively implements hidden-surface removal. Intersection tests account for most of the computational overheads in ray tracing, and much research effort has gone into how to reduce this cost.

Grafted onto this basic recursive process, which follows specular interaction through the scene, are the computation of direct reflection and shadow computation. At each node or surface hit, we calculate these two contributions. Direct reflection is calculated by applying, for each light source, a Phong reflection model (or some other local model) at the node under consideration. The direct reflection contribution is diminished if the point is in shadow with respect to a light source. Thus, at any hit point or node, there are three contributions to the light intensity passed back up through the recursion:

A reflected-ray contribution
A transmitted-ray contribution
A local contribution unaltered or modified by the shadow computation

Shadow computation is easily implemented by injecting the light direction vector, used in the local contribution calculation, into the intersection test to see if it is interrupted by any intervening objects. This ray is called a *shadow feeler*. If L is so interrupted, then the current surface point lies in shadow. If a wholly opaque object lies in the path of the shadow feeler, then the local contribution is reduced to the ambient value. An attenuation in the local contribution is calculated if the intersecting object is partially transparent. Note that it is no longer appropriate to consider L a constant vector (light source at infinity) and the so-called shadow feelers are rays whose direction is calculated at each hit point. Because light sources are normally point sources, this procedure produces, like add-on shadow algorithms, hard-edged shadows. (Strictly speaking, a shadow feeler intersecting partially transparent objects should be refracted. It is not possible to do this, however, in the simple scheme described. The shadow feeler is initially calculated as the straight line between the surface intersection and the light source. This is an easy calculation, and it would be difficult to trace a ray from this point to the light source and include refractive effects.)

Finally note that, as the number of light sources increases from one, the computational overheads for shadow testing rapidly predominate. This is because the main rays are traced only to an average depth of between one and two. However, each ray-surface intersection spawns n shadow feelers (where n is the number of light sources), and the object intersection cost for a shadow feeler is exactly the same as for a main ray.

38.3.1 Intersection Testing

We have mentioned that intersection testing forms the heart of a ray tracer and accounts for most of the cost of the algorithm. In the 1980s, much research was devoted to this aspect of ray tracing. There is a large body of literature on the subject that dwarfs the work devoted to improvements in the ray-traced image such as, for example, distributed ray tracing.

Intersection testing means finding whether a ray intersects an object and, if so, the point of intersection. Expressing the problem in this way hints at the usual approach, which is to try to cut down the overall cost by using some scheme, like a bounding volume, that prevents the intersection test from searching through all the polygons in an object if the ray cannot hit the object.

The cost of ray tracing and the different possible approaches depend much on the way in which objects are represented. For example, if a voxel representation is used and the entire space is labeled with object occupancy, then discretizing the ray into voxels and stepping along it from the start point will reveal the first object that the ray hits. Contrast this with a brute-force intersection test, which must test a ray against every object in the scene to find the hit nearest to the ray start point.

38.4 Rendering Using the Radiosity Method

The *radiosity method* arrived in computer graphics in the mid-1980s, a few years after ray tracing. Most of the early development work was carried out at Cornell University under the guidance of D. Greenberg, a major figure in the development of the technique. The emergence of the **hemicube** algorithm and, later, the progressive refinement algorithm, established the method and enabled it to leave research laboratories and become a practical rendering tool. Nowadays, many commercial systems are available, and most are implementations of these early algorithms.

The radiosity method provides a solution to diffuse interaction, which, as we have discussed, cannot easily be incorporated in ray tracing, but at the expense of dividing the scene into large patches (over which the radiosity is constant). This approach cannot cope with sharp specular reflections. Essentially, we have two global methods: ray tracing, which simulates global specular reflection, and transmission and radiosity, which simulate global diffuse interaction.

In terms of the global phenomena that they simulate, the methods are mutually exclusive. Predictably, a major research bias has involved the unification of the two methods into a single global solution. Research is still actively pursued into many aspects of the method — particularly form-factor determination and scene decomposition into elements or patches.

38.4.1 Basic Theory

The radiosity method works by dividing the environment into largish elements called *patches*. For every pair of patches in the scene, a parameter F_{ij} is evaluated. This parameter, called a *form factor*, depends on the geometric relationship between patches i and j. This factor is used to determine the strength of diffuse light interaction between pairs of patches, and a large system of equations is set up which, on solution, yields the radiosity for each patch in the scene.

The radiosity method is an object–space algorithm, solving for a single intensity for each surface patch within an environment and not for pixels in an image-plane projection. The solution is thus independent of viewer position. This complete solution is then injected into a renderer that computes a particular view by removing hidden surfaces and forming a projection. This phase of the method does not require much computation (intensities are already calculated), and different views are easily obtained from the general solution. The method is based on the assumption that all surfaces are perfect diffusers or ideal Lambertian surfaces.

Radiosity, B, is defined as the energy per unit area leaving a surface patch per unit time and is the sum of the emitted and the reflected energy:

$$B_i dA_i = E_i dA_i + R_i \int_j B_j F_{ji} dA_j$$

Expressing this equation in words, we have for a single patch i

$$\text{radiosity } x \text{ area} = \text{emitted energy} + \text{reflected energy}$$

E_i is the energy emitted from a patch, and emitting patches are, of course, light sources. The reflected energy is given by multiplying the incident energy by R_i, the reflectivity of the patch. The incident energy is the energy that arrives at patch i from all other patches in the environment; that is, we integrate over the environment, for all $j (j \neq i)$, the term $B_j F_{ji} dA_j$. This is the energy leaving each patch j that arrives at patch i.

For a discrete environment, the integral is replaced by a summation and constant radiosity is assumed over small discrete patches. It can be shown that

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij}$$

Such an equation exists for each surface patch in the enclosure, and the complete environment produces a set of n simultaneous equations of the form.

$$\begin{bmatrix} 1 - R_1F_{11} & -R_1F_{12} & \dots & -R_1F_{1n} \\ -R_2F_{21} & 1 - R_2F_{22} & \dots & -R_2F_{2n} \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ -R_nF_{n1} & -R_nF_{n2} & \dots & 1 - R_nF_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \dots \\ \dots \\ \dots \\ B_n \end{bmatrix} \begin{bmatrix} E_1 \\ E_2 \\ \dots \\ \dots \\ \dots \\ E_n \end{bmatrix}$$

The $E_i s$ are nonzero only at those surfaces that provide illumination, and these terms represent the input illumination to the system. The $R_i s$ are known, and the $F_{ij} s$ are a function of the geometry of the environment. The reflectivities are wavelength-dependent terms, and the previous equation should be regarded as a monochromatic solution, a complete solution being obtained by solving for however many color bands are being considered. We can note at this stage that $F_{ii} = 0$ for a plane or convex surface — none of the radiation leaving the surface will strike itself. Also, from the definition of the form factor, the sum of any row of form factors is unity.

Since the form factors are a function only of the geometry of the system, they are computed once only. Solving this set of equations produces a single value for each patch. This information is then input to a modified Gouraud renderer to give an interpolated solution across all patches.

38.4.2 Form-Factor Determination

A significant early development was a practical method to evaluate form factors. The algorithm is both an approximation and an efficient method of achieving a numerical estimation of the result. The form factor between patches i and j is defined as

$$F_{ij} = \frac{\text{radiative energy leaving surface } \mathrm{A}_i \text{ that strikes } \mathrm{A}_j \text{ directly}}{\text{radiative energy leaving } \mathrm{A}_i \text{ in all directions in the hemispherical space surrounding } \mathrm{A}_i}$$

This is given by

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j dA_i$$

where the geometric conventions are illustrated in Figure 38.17. Now, it can be shown that this patch-to-patch form factor can be approximated by the differential-area-to-finite-area form factor

$$F_{dA_i A_j} = \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j$$

where we are now considering the form factor between the elemental area da_i and the finite area A_j. This approximation is calculated by the hemicube algorithm [Cohen and Greenberg, 1985]. The factors that enable the approximation to a single integral and its veracity are quite subtle and are outside the scope of this treatment.

A good intuition of the workings of the algorithm can be gained from a pictorial visualization (see Figure 38.18). Figure 38.18a is a representation of the property known as the *Nusselt analog*. In the example, patches A, B, and C all have the same form factor with respect to patch i. Patch B is the projection of A onto a hemicube, centered on patch i, and C is the projection of A onto a hemisphere. This property is the

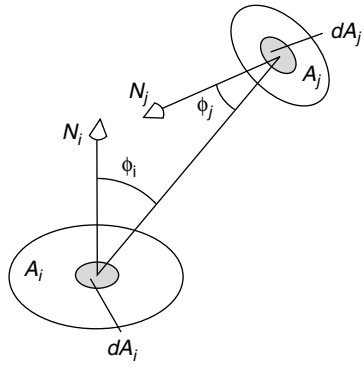


FIGURE 38.17 Parameters used in the definition of a form factor.

foundation of the hemicube algorithm, which places a hemicube on each patch i — (Figure 38.18b). The hemicube is subdivided into elements; associated with each element is a precalculated *delta form factor*. The hemicube is placed on patch i, and then patch j is projected onto it. In practice, this involves a clipping operation, because in general a patch can project onto three faces of the hemicube. Evaluating F_{ij} involves simply summing the values of the delta form factors onto which patch j projects (Figure 38.18c).

Another aspect of form-factor determination solved by the hemicube algorithm is the intervening-patch problem. Normally, we cannot evaluate the form-factor relationship between a pair of patches independently of one or more patches that happen to be situated between them. The hemicube algorithm solves this by making the hemicube a kind of Z-buffer in addition to its role as five projection planes. This is accomplished as follows. For the patch i under consideration, every other patch in the scene is projected onto the hemicube. For each projection, the distance from patch i to the patch being projected is compared with the smallest distance associated with previously projected patches, which is stored in hemicube elements. If a projection from a nearer patch occurs, then the identity of that patch and its distance from patch i are stored in the hemicube elements onto which it projects. When all patches are projected, the form factor F_{ij} is calculated by summing the delta form factors that have registered patch j as a nearest projection.

Finally, consider Figure 38.19, which gives an overall view of the algorithm. This emphasizes the fact that there are three entry points into the process for an interactive program. Changing the geometry of the scene means an entire recalculation, starting afresh with the new scene. However, if only the wavelength-dependent properties of the scene are altered (reflectivities of objects and colors of light sources), then the expensive part of the process — the form-factor calculations — is unchanged. Because the method is view-independent, changing the position of the viewpoint involves only the final process of interpolation and hidden-surface removal. This enables real-time, interactive walkthroughs using the precalculated solution, a popular application of the radiosity technique in computer-aided design (see Figure 38.20). The high-quality imagery gives designers a better feel for the final product than would be possible with simple rendering packages.

38.4.3 Problems with the Basic Method

Several problems occur if the method is implemented without elaboration. Here, we will restrict ourselves to identifying these and giving a pointer to the solutions that have emerged. The reader is referred to the appropriate literature for further details.

The first problem emerges from consideration of how to divide the environment into patches. Dividing the scene equally into large patches, as far as the geometry of the objects allows, will not suffice. The basic radiosity solution calculates a constant radiosity over the area of a patch. Larger patches mean fewer patches and a faster solution, but there will be areas in the scene that will exhibit a fast change in reflected

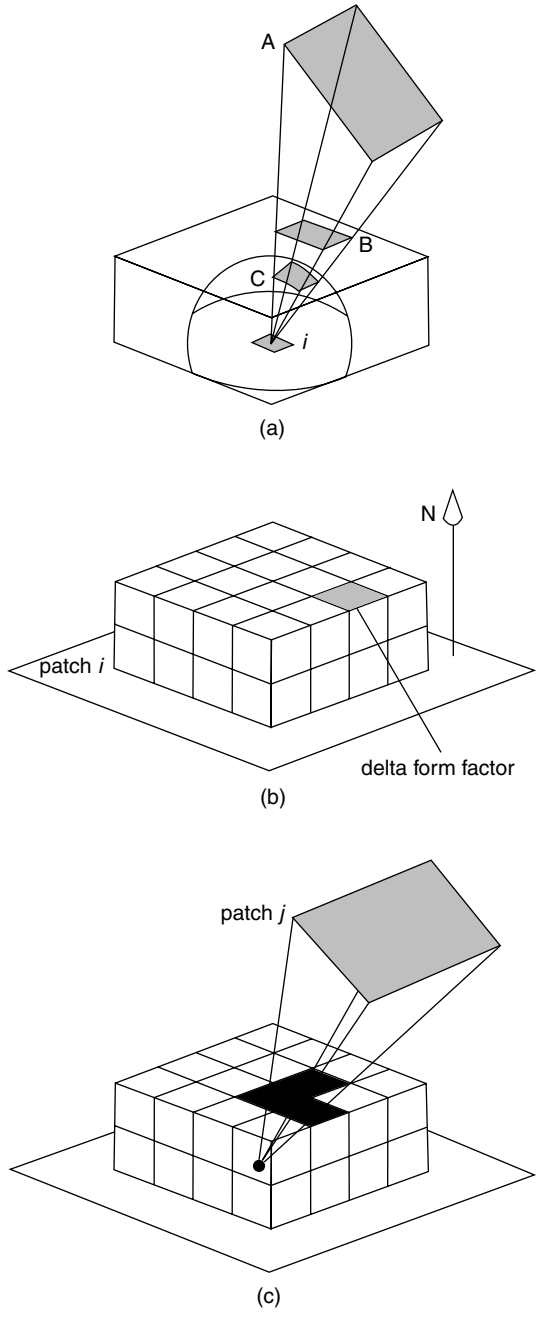


FIGURE 38.18 Visualization of the properties used in the hemicube algorithm for form-factor evaluation. (a) Nusselt analogue: patches A, B, and C have the same form factor with respect to patch i. (b) Delta form factors are precalculated for the hemicube. (c) The hemicube is positioned over patch i. Each patch j is projected onto the hemicube. F_{ij} is calculated by summing the delta form factors of the hemicube elements onto which j projects.

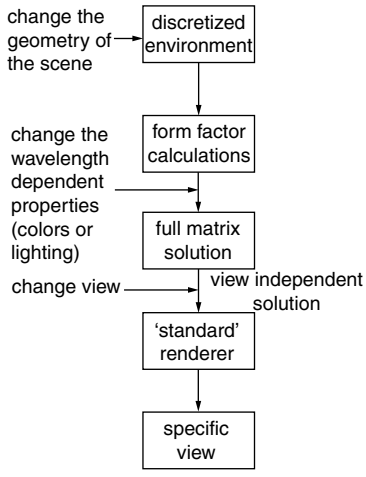


FIGURE 38.19 Processes and interactive entry points in a radiosity algorithm.

FIGURE 38.20 The interior of a car rendered using the radiosity method. (Image rendered using LightWorks, courtesy of LightWork Design.)

light per unit area. Shadow boundaries, for example, exhibit a shape that depends on the obscuring object, and small patches are required in the area of these, which will in general be curves. If the shadowed surface is not sufficiently subdivided, then the shadow boundary will exhibit steps in the shape of the (square or rectangular) patches. However, we do not know until the solution is computed where such areas are in the scene. A common approach to this problem is to incorporate a variable subdivision into the solution, subdividing as a solution emerges until the difference in radiosity between adjacent patches falls below a threshold [Cohen et al., 1986].

Another problem is the time taken by the algorithm — approximately one order of magnitude longer than ray tracing. An ingenious reordering of the method for solving the equation set, called the *progressive refinement algorithm* [Cohen et al., 1988], ameliorates this problem and provides a user with an early (approximate) image which is progressively refined, that is, it becomes more and more accurate. In a normal solution, each patch progresses toward a final value in turn. In the progressive refinement algorithm, each and every patch is updated for each iteration, and all patches proceed toward their final value simultaneously. The scene is at first dark and then gets lighter and lighter as the patches increase their radiosity. Early, approximate solutions are quickly computed by distributing an arbitrary ambient component among all patches. The early solutions are "fully" lit but incorrect. Progressive refinement means displaying more and more accurate radiosities on each patch and reducing the approximate ambient component at the same time. A user can thus see from the start some solution, which then proceeds to get more and more accurate. The process can then be interrupted at some stage if the image is not as required or has some visually obvious error.

38.5 The (Irresistible) Survival of Mainstream Rendering

Three-dimensional computer graphics have evolved significantly over the past three decades. We saw at first the development of expensive workstations, which were used in expensive applications, such as computer-aided architectural design, and which were unaffordable to home consumers. Recently, PC graphics hardware has undergone a rapid evolution, resulting in extremely powerful graphics cards being available to consumers at a cost of $500 or less. This market has, of course, been spurred by the apparently never-ending popularity of computer games. This application of three-dimensional computer graphics has dwarfed all others and hastened the rapid development of games consoles.

The demand from the world of computer games is to render three-dimensional environments, at high quality, in real time. The shift of emphasis to real-time rendering has resulted in many novel rendering methods that address the speed-up problem [Akenine-Möller and Haines, 2002; Watt and Policarpo, 2001]. The demand for ever-increasing quality led to a number of significant stages in rendering methodology. First, there was the shift of the rendering load from the CPU onto graphics cards. This add-on hardware performed most of the rendering tasks, implementing the mainstream polygon rendering described previously. Much use was made of texture mapping hardware to precalculate light maps, which enabled rendering to be performed in advance for all static detail in the environment.

At the time of writing (2003), consumer hardware is now available which is powerful enough to enable all rendering in real time, obviating the need for precalculation. This makes for better game content, with dynamic and static objects having access to the same rendering facilities.

Although graphics cards were at first simply viewed as black boxes that received a massive list of polygons, recent developments have seen cards with functionality exposed to the programmer. Such functionality has enabled the per-pixel (or per-fragment) programming necessary for real increases in image quality. Thus, we have the polygon processing returned to the control of the programmer and the need for an expansion of graphics APIs (such as OpenGL) to enable programmers to exploit this new functionality. Companies such as NVIDIA and ATI have thrived on offering such functionality.

One of the enduring facts concerning the history of this evolution is the inertia of the mainstream rendering methodology. Polygon meshes have survived as the most popular representation, and the rendering of polygons using interpolative shading and a set of light sources seems as firmly embedded as ever.

38.6 An OpenGL Example

We complete this chapter with a simple example in OpenGL that renders a single object: a teapot (see Figure 38.21). Two points are worth noting. First, polygons become triangles. Although we have consistently used the word *polygon* in this text, most graphics hardware is optimized to deal with triangles. Second, there is no exposure to the programmer of processes such as rasterization. This pixel process is invisible, although we briefly discussed in Section 38.5 that new graphics cards are facilitating access to pixel processing.

FIGURE 38.21 A teapot rendered using OpenGL.

```
// a1.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#include "math.h"

#include <gl/gl.h>
#include <gl/glu.h>
#include <gl/glut.h>

#define WINDOW_WIDTH 500
#define WINDOW_HEIGHT 500

#define NOOF_VERTICES 546
#define NOOF_TRIANGLES 1008

float vertices[NOOF_VERTICES][3] = {...};
int triangles[NOOF_TRIANGLES][3] = {...};
float vertexNormals[NOOF_VERTICES][3] = {...};

void loadData() {
  // load data from file
}

// ************************************************

void initLight0(void) {
  GLfloat light0Position[4] = {0.8,0.9,1.0,0.0};
                              // Since the final parameter is 0, this is a
                              // directional light at 'infinity' in the
                              // direction (0.8,0.9,1.0) from the world
                              // origin. A parameter of 1.0 can be used to
                              // create a positional (spot) light, with
                              // further commands to set the direction and
                              // cut-off angles of the spotlight.
  GLfloat whiteLight[4] = {1.0,1.0,1.0,1.0};
  GLfloat ambient[4] = {0.6,0.6,0.6,1.0};
  glLightfv(GL_LIGHT0, GL_POSITION, light0Position);
  glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, whiteLight);
  glLightfv(GL_LIGHT0, GL_SPECULAR, whiteLight);
}

void init(void) {
  glClearColor(0, 0, 0, 1); // Screen will be cleared to black.
  glEnable(GL_DEPTH_TEST);  // Enables screen depth tests using z buffer.
  glEnable(GL_CULL_FACE);   // Enable culling.
  glCullFace(GL_BACK);      // Back-facing polygons will be culled.
                            // Note that by default the vertices of a
                            // polygon are considered to be listed in
                            // counterclockwise order.
  glShadeModel(GL_SMOOTH);  // Use smooth shading, not flat shading.
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
                            // Both 'sides' of a polygon are filled.
```

```
                              // Thus the 'inside' of enclosed objects could
                              // be viewed.

  glEnable(GL_LIGHTING);      // Enables lighting calculations.
  GLfloat ambient[4] = {0.2,0.2,0.2,1.0};
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient);
                              // Set a global ambient value so that some
                              // 'background' light is included in all
                              // lighting calculations.
  glEnable(GL_LIGHT0);        // Enable GL_LIGHT0 for lighting calculations.
                              // OpenGL defines 8 lights that can be set and
                              // enabled.
  initLight0();               // Call function to initialise GL_LIGHT0 state.
}

void displayObject(void) {
                              // First the material properties of the object
                              // are set for use in lighting calculations
                              // involving the object.
  GLfloat matAmbientDiffuse[4] = {0.9,0.6,0.3,1.0};
                              // A mustard color.
  GLfloat matSpecular[4] = {1.0,1.0,1.0,1.0};
  GLfloat matShininess[1] = {64.0};
  GLfloat noMat[4] = {0.0,0.0,0.0,1.0};
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, matAmbientDiffuse);
                              // Ambient and diffuse can be set separately,
                              // but it is common to use
                              // GL_AMBIENT_AND_DIFFUSE to set them to the
                              // same value.
  glMaterialfv(GL_FRONT, GL_SPECULAR, matSpecular);
  glMaterialfv(GL_FRONT, GL_SHININESS, matShininess);
  glMaterialfv(GL_FRONT, GL_EMISSION, noMat);

  glPushMatrix();
  glRotatef(-90,1,0,0);
  glBegin(GL_TRIANGLES);      // The object is defined using triangles,
                              // rather than GL_QUADS
  for (int i=0; i<NOOF_TRIANGLES; i++)
                              // NOOF_TRIANGLES is a constant but could be a
                              // variable set when the data is read from
                              // file.
  // Here we send three vertices for each triangle, although
  // there are OpenGL commands to deal with triangle strips
  // and with sending a display list of data.
    for (int j=0; j<3; j++) {
                              // The attributes for a vertex are sent before
                              // the vertex itself.
                              // Here we send the vertex normal, but texture
                              // coordinates could also be sent.
      glNormal3-D(vertexNormals[triangles[i][j]][0], // x of vertex normal
                  vertexNormals[triangles[i][j]][1], // y of vertex normal
                  vertexNormals[triangles[i][j]][2]);// z of vertex normal
      glVertex3-D(vertices[triangles[i][j]][0],      // x of vertex
                  vertices[triangles[i][j]][1],      // y of vertex
                  vertices[triangles[i][j]][2]);     // z of vertex
    }
```

```
  glEnd();                    // Finish sending triangles
  glPopMatrix();
}

void viewingSystem(int w, int h) {
  glViewport(0, 0, (GLsizei)w,(GLsizei)h);
                              // The viewpoort is set to occupy the whole
                              // screen window display area,
                              // OpenGL maintains two matrices for viewing
                              // and model transformation.
                              // First we deal with the projection matrix.
  glMatrixMode(GL_PROJECTION);
                              // The GL_PROJECTION matrix becomes current.
  glLoadIdentity();           // It is set to the identity matrix.
  gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
                              // gluPerspective is a routine defined in the
                              // glu utility library that is included in
                              // OpenGL distributions. It creates a symmetric
                              // perspective-view frustum and multiplies the
                              // current (GL_PROJECTION) matrix by it.
                              // Parameter one is the angle of the field of
                              // view. Parameter two is the aspect ratio of
                              // the frustum. Parameter three is the distance
                              // of the near clipping plane from the
                              // viewpoint. Parameter four is the distance of
                              // the far clipping plane from the viewpoint.
                              // Screen coordinate (0,0) will be at the
                              // center of the viewport.

  glMatrixMode(GL_MODELVIEW);
                              // The GL_MODELVIEW matrix is now made current.
  glLoadIdentity();           // It is set to the identity matrix.
  gluLookAt(3.0,5.0,4.0, 0.0,0.0,0.0, 0.0,1.0,0.0);
                              // The utility routine gluLookAt defines a
                              // viewing transformation matrix and
                              // multiples it to the right of the current
                              // (GL_MODELVIEW) matrix.
                              // The eye position is set at (3.0,5.0,4.0).
                              // The next three parameters specify a point
                              // along the desired line of
                              // sight - here we are looking at the origin of
                              // the world coordinate system.
                              // The final three parameters indicate the
                              // direction that is 'up' in the
                              // viewing volume - these values are
                              // automatically adjusted so that the
                              // up direction is perpendicular to the line of
                              // sight.
}

void display(void) {
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
                              // Each time the display function is called the
                              // relevant buffers must be cleared.

  displayObject();            // The object (or set of objects) is displayed.
```

```
  glFlush();                // Forces previously issued OpenGL commands to
                            // begin execution.
}


// The following main method makes use of the glut utility library to open
// a simple console window for display purposes. glut is a window-system-
// independent toolkit, written by Mark Kilgard, which is commonly used for
// simple OpenGL applications. For further information, see Kilgard, M.,
// "OpenGL Programming for the X Window System," Addison-Wesley, 1996.

int main(int argc, char** argv) {
  loadData();               // Load data from file into vertices and
                            // triangles arrays. Could also be used to
                            // calculate vertex normals which are needed
                            // for subsequent lighting calculations.
  glutInit(&argc, argv);    // Initializes glut.

  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA | GLUT_DEPTH);
                            // Decide on display modes needed. Here, we
                            // specify that we want a single-buffered
                            // window, rgba color mode, and a depth buffer.
  glutInitWindowSize(WINDOW_WIDTH,WINDOW_HEIGHT);
                            // Size in pixels of the window.
  glutInitWindowPosition(0,0);
                            // Screen location of upper left corner of
                            // window.
  glutCreateWindow(argv[0]);// Creates a window with an OpenGL context.
  Init();                   // Initialize our use of OpenGL.
  glutDisplayFunc(display); // The function display will be called whenever
                            // glut determines that the contents of
                            // the screen window need to be redisplayed,
                            // e.g., when a window is uncovered after being
                            // covered by another window.
  glutReshapeFunc(viewingSystem);
                            // If the window is resized, e.g., enlarged, the
                            // viewingSystem function is called.
  glutMainLoop();           // Last thing that is done. Now the window is
                            // shown on screen and event processing begins,
                            // i.e., continuous loop processing events such
                            // as resize window until program is
                            // terminated.
  return 0;
}
```

Defining Terms

Ambient: The constant term used in local reflection models to simulate global illumination. It illuminates parts of a surface which cannot be seen from the light source but which are visible from the viewpoint.

Antialiasing: The term given to measures designed to eliminate defects in computer graphics images, the most common being the effect produced when a curved edge in the image is displayed as jagged due to the finite extent of the pixels. The term is somewhat inappropriate, as *aliasing* is a specific signal-processing term and many computer graphics defects are not aliases in that sense.

Clipping: It is common to define a view frustum, a volume emanating from the viewpoint, against which objects are clipped. For example, objects behind the viewer must be eliminated from consideration.

Culling: A process to remove whole polygons that cannot be seen from the viewpoint and therefore do not need to be considered by the hidden-surface removal algorithm.

Diffuse: Local reflection models separately evaluate a diffuse, a specular, and an ambient component. The diffuse component is the light that is reflected from a point equally in all directions, simulating a matte or plastic-like surface.

Flat shading: A shading option in which all the pixels for a polygon are allocated the same shade — there is no variation within a polygon. This makes the underlying polygonal structure visible.

Global reflection models: Models that attempt to model indirect illumination. At a surface point, they consider both light coming directly from light sources and light that has been reflected from other objects.

Gouraud shading: An interpolative shading method for polygons in which the parameter that is interpolated is the reflected light intensity at the vertices.

Hemicube: An algorithmic device that enables an efficient calculation of the form-factor values in the radiosity method. A hemicube is an efficient approximation to a hemisphere.

Hidden-surface removal: The general algorithm that removes hidden surfaces and deals with such cases where one object partially obscures another. In general, a hidden-surface algorithm will eliminate those fragments of a polygon that are not visible because they are behind another object.

Local reflection models: Models that simulate the reflection of light incident directly on an object from a light source. Unlike global models, they take no account of indirect light reflected from another object.

Phong shading: An interpolative shading method in which the parameter that is interpolated is the vertex normal at the vertices. This produces an interpolated normal for every pixel onto which the polygon projects, and a reflection model is evaluated at each pixel in the image plane.

Polygon mesh: The most common form of object representation is to build a set of planar facets or polygons that (approximately, in general) represent the surface of an object.

Progressive refinement: An elaboration of the original radiosity method that enables a visualization of the solution to emerge as the equations are being solved. The solution, originally approximated with an ambient term, is gradually made more and more accurate.

Projective transformation: Usually the final geometric transformation, it produces the perspective foreshortening desired in most applications.

Radiosity: A global reflection model that divides the scene into large elements called patches, calculates a parameter that reflects the geometric relationship between all pairs of patches, and sets up a system of equations whose solution yields a constant radiosity for each patch. These radiosity values are input to a Gouraud-type interpolation to produce a rendered image. The geometric extent of the patches means that the basic method can deal only with diffuse interaction.

Ray tracing: A global reflection model that casts infinitesimally thin rays into the scene from each pixel and follows or traces these as they are reflected and transmitted by objects that they encounter. Such a ray tracer can find out only about specular interaction.

Shading algorithm: A general term that describes that part of the process that makes a geometric description look like a solid, three-dimensional object.

Specular: The specular component is the light reflected from a point in the mirror direction, as if the surface were a perfect mirror. In practice, this component is empirically spread to simulate a practical glossy surface.

Viewing transformation: Generates a representation of the object or scene as seen from the viewpoint of an observer positioned somewhere in the scene and looking toward some aspect of it.

Virtual camera: A common analogy used for the series of geometric transformations that form a two-dimensional image on the view surface. These transformations have the same effect as a (perfect) pinhole camera. The analogy is particularly useful in animation where the camera is to be choreographed.

References

Akenine-Möller, T., and Haines, E. 2002. *Real-time Rendering, 2nd Ed*. A.K. Peters, Ltd.

Blinn, J. 1977. Models of light reflection for computer synthesized pictures. pp. 192–198. *Comput. Graphics* (*Proc. SIGGRAPH*).

Cohen, M.F., and Greenberg, D.P. 1985. A radiosity solution for complex environments. pp. 31–40. *Comput. Graphics* (*Proc. SIGGRAPH*).

Cohen, M.F., Greenberg, D.P., and Immel, D.S. 1988. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Graphics and Applications,* 6(2): 26–35.

Cohen, M.F., Chen, S.E., Wallace, J.R., and Greenberg, D.P. 1988. A progressive refinement approach to fast radiosity image generation. pp. 75–84. *Comput. Graphics* (*Proc. SIGGRAPH*).

Cook, R.L., Porter, T., and Carpenter, L. 1984. Distributed ray tracing. pp. 137–145. *Comput. Graphics* (*Proc. SIGGRAPH*).

Fiume, E.L. 1989. *The Mathematical Structure of Computer Graphics*. Academic Press, San Diego, CA.

Gouraud, H. 1971. Illumination for computer generated pictures. *Commun. ACM* 18(60): 628–678.

Newman, W., and Sproull, R. 1973. *Principles of Interactive Computer Graphics*. McGraw-Hill, New York.

Phong, B.T. 1975. Illumination for computer generated pictures. *Commun. ACM* 18(6): 311–317.

Sutherland, I.E., and Hodgman, G.W. 1974. Re-entrant polygon clipping. *Commun. ACM* 17(1): 32–42.

Watt, A., and Policarpo, F. 2001. *3-D Games, Real-time Rendering and Software Technology: Volume 1*. Addison-Wesley, Reading, MA.

Watt, A., and Watt, M. 1992. *Advanced Animation and Rendering Techniques*. ACM Press, New York.

Whitted, T. 1980. An improved illumination model for shaded display. *Commun. ACM* 26(6): 342–349.

Woo, M., Neider, J., Davis, T., and Shreiner, D. 1999. *OpenGL Programming Guide, 3rd Ed*. Addison-Wesley, Reading, MA.

Further Information

The References section comprises mostly the original sources of the algorithms that are commonly incorporated in rendering engines. A would-be implementer, however, would be best directed to a general textbook, such as [Watt and Watt, 1992], or the encyclopedic *Computer Graphics: Principles and Practice* by Foley et al.

Undoubtedly, the best source of rendering techniques and their development is the annual ACM SIGGRAPH conference (proceedings published by ACM Press). Browsing through past proceedings gives a feel for the fascinating development and history of image synthesis. Indeed, in 1998, ACM SIGGRAPH published the book *Seminal Graphics: Pioneering Efforts that Shaped the Field*, edited by Wolfe, which includes many of the pioneering rendering papers listed in the References section.

39

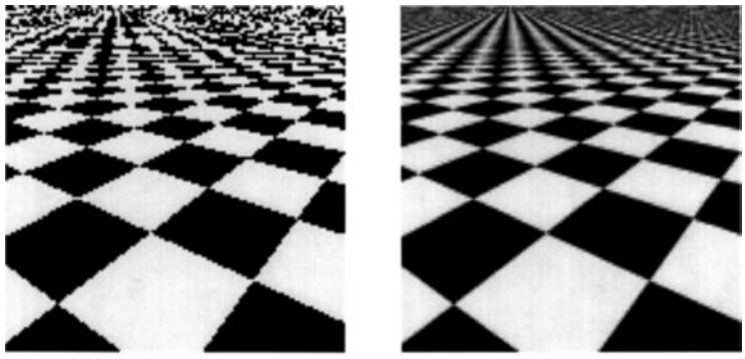
Sampling, Reconstruction, and Antialiasing

George Wolberg
City College of New York

39.1 Introduction

This chapter reviews the principal ideas of sampling theory, reconstruction, and antialiasing. Sampling theory is central to the study of sampled-data systems, e.g., digital image transformations. It lays a firm mathematical foundation for the analysis of sampled signals, offering invaluable insight into the problems and solutions of sampling. It does so by providing an elegant mathematical formulation describing the relationship between a continuous signal and its samples. We use it to resolve the problems of image reconstruction and aliasing. Reconstruction is an interpolation procedure applied to the sampled data. It permits us to evaluate the discrete signal at any desired position, not just the integer lattice upon which the sampled signal is given. This is useful when implementing geometric transformations, or warps, on the image. Aliasing refers to the presence of unreproducibly high frequencies in the image and the resulting artifacts that arise upon undersampling.

Together with defining theoretical limits on the continuous reconstruction of discrete input, sampling theory yields the guidelines for numerically measuring the quality of various proposed filtering techniques.



(a) (b)

FIGURE 39.1 Oblique checkerboard: (a) unfiltered, (b) filtered.

This proves most useful in formally describing reconstruction, aliasing, and the filtering necessary to combat the artifacts that may appear at the output.

In order to better motivate the importance of sampling theory and filtering, we demonstrate its role with the following examples. A checkerboard texture is shown projected onto an oblique planar surface in Figure 39.1. The image exhibits two forms of artifacts: jagged edges and moiré patterns. Jagged edges are prominent toward the bottom of the image, where the input checkerboard undergoes magnification. It reflects poor reconstruction of the underlying signal. The moiré patterns, on the other hand, are noticeable at the top, where minification (compression) forces many input pixels to occupy fewer output pixels. This artifact is due to aliasing, a symptom of undersampling.

Figure 39.1(a) was generated by projecting the center of each output pixel into the checkerboard and sampling (reading) the value of the nearest input pixel. This point sampling method performs poorly, as is evident by the objectionable results of Figure 39.1(a). This conclusion is reached by sampling theory as well. Its role here is to precisely quantify this phenomenon and to prescribe a solution. Figure 39.1(b) shows the same mapping with improved results. This time the necessary steps were taken to preclude artifacts. In particular, a superior reconstruction algorithm was used for interpolation to suppress the jagged edges, and antialiasing filtering was carried out to combat the symptoms of undersampling that gave rise to the moiré patterns.

39.2 Sampling Theory

Both reconstruction and antialiasing share the twofold problem addressed by sampling theory:

1. Given a continuous input signal $g(x)$ and its sampled counterpart $g_s(x)$, are the samples of $g_s(x)$ sufficient to exactly describe $g(x)$?
2. If so, how can $g(x)$ be reconstructed from $g_s(x)$?

The solution lies in the frequency domain, whereby spectral analysis is used to examine the spectrum of the sampled data.

The conclusions derived from examining the reconstruction problem will prove to be directly useful for resampling and indicative of the filtering necessary for antialiasing. Sampling theory thereby provides an elegant mathematical framework in which to assess the quality of reconstruction, establish theoretical limits, and predict when it is not possible.

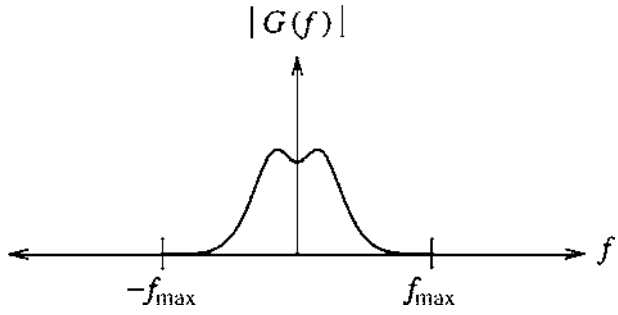


FIGURE 39.2 Spectrum $G(f)$.

39.2.1 Sampling

Consider a 1-D signal $g(x)$ and its spectrum $G(f)$, as determined by the Fourier transform:

$$G(f) = \int_{-\infty}^{\infty} g(x)e^{-i2\pi fx}\,dx \tag{39.1}$$

Note that x represents spatial position and f denotes spatial frequency.

The magnitude spectrum of a signal is shown in Figure 39.2. It shows the frequency content of the signal, with a high concentration of energy in the low-frequency range, tapering off toward the higher frequencies. Since there are no frequency components beyond $f_{\max}$, the signal is said to be **bandlimited** to frequency $f_{\max}$.

The continuous output $g(x)$ is then digitized by an ideal impulse sampler, the comb function, to get the sampled signal $g_s(x)$. The ideal 1-D sampler is given as

$$s(x) = \sum_{n=-\infty}^{\infty} \delta(x - nT_s) \tag{39.2}$$

where δ is the familiar impulse function and T_s is the sampling period. The running index n is used with δ to define the impulse train of the comb function. We now have

$$g_s(x) = g(x)s(x) \tag{39.3}$$

Taking the Fourier transform of $g_s(x)$ yields

$$G_s(f) = G(f) * S(f) \tag{39.4}$$

$$= G(f) * \left[\sum_{n=-\infty}^{n=\infty} f_s\delta(f - nf_s)\right] \tag{39.5}$$

$$= f_s \sum_{n=-\infty}^{n=\infty} G(f - nf_s) \tag{39.6}$$

where f_s is the sampling frequency and $*$ denotes convolution. The above equations make use of the following well-known properties of Fourier transforms:

1. Multiplication in the spatial domain corresponds to convolution in the frequency domain. Therefore, Equation 39.3 gives rise to a convolution in Equation 39.4.
2. The Fourier transform of an impulse train is itself an impulse train, giving us Equation 39.5.
3. The spectrum of a signal sampled with frequency f_s ($T_s = 1/f_s$) yields the original spectrum replicated in the frequency domain with period f_s Equation 39.6.

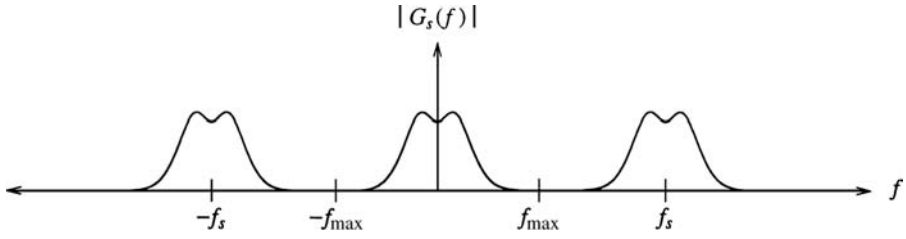


FIGURE 39.3 Spectrum $G_s(f)$.

This last property has important consequences. It yields a spectrum $G_s(f)$, which, in response to a sampling period $T_s = 1/f_s$, is *periodic in frequency* with period f_s. This is depicted in Figure 39.3. Notice, then, that a small sampling period is equivalent to a high sampling frequency, yielding spectra replicated far apart from each other. In the limiting case when the sampling period approaches zero ($T_s \to 0$, $f_s \to \infty$), only a single spectrum appears — a result consistent with the continuous case.

39.3 Reconstruction

The above result reveals that the sampling operation has left the original input spectrum *intact*, merely replicating it periodically in the frequency domain with a spacing of f_s. This allows us to rewrite $G_s(f)$ as a sum of two terms, the low-frequency (baseband) and high-frequency components. The *baseband* spectrum is exactly $G(f)$, and the high-frequency components, $G_{\text{high}}(f)$, consist of the remaining replicated versions of $G(f)$ that constitute harmonic versions of the sampled image:

$$G_s(f) = G(f) + G_{\text{high}}(f) \tag{39.7}$$

Exact signal reconstruction from sampled data requires us to discard the replicated spectra $G_{\text{high}}(f)$, leaving only $G(f)$, the spectrum of the signal we seek to recover. This is a crucial observation in the study of sampled-data systems.

39.3.1 Reconstruction Conditions

The only provision for exact **reconstruction** is that $G(f)$ be undistorted due to overlap with $G_{\text{high}}(f)$. Two conditions must hold for this to be true:

1. The signal must be bandlimited. This avoids spectra with infinite extent that are impossible to replicate without overlap.
2. The sampling frequency f_s must be greater than twice the maximum frequency f_{max} present in the signal. This minimum sampling frequency, known as the **Nyquist rate**, is the minimum distance between the spectra copies, each with bandwidth f_{max}.

The first condition merely ensures that a sufficiently large sampling frequency exists that can be used to separate replicated spectra from each other. Since all imaging systems impose a bandlimiting filter in the form of a point spread function, this condition is always satisfied for images captured through an optical system. Note that this does not apply to synthetic images, e.g., computer-generated imagery.

The second condition proves to be the most revealing statement about reconstruction. It answers the problem regarding the sufficiency of the data samples to exactly reconstruct the continuous input signal. It states that exact reconstruction is possible only when $f_s > f_{\text{Nyquist}}$, where $f_{\text{Nyquist}} = 2f_{\text{max}}$. Collectively, these two conclusions about reconstruction form the central message of sampling theory, as pioneered by Claude Shannon in his landmark papers on the subject [Shannon 1948, 1949].

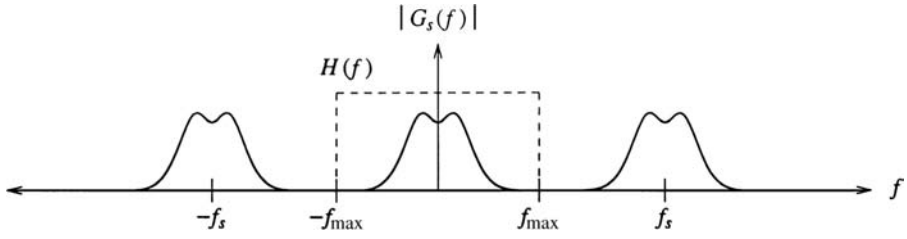


FIGURE 39.4 Ideal low-pass filter $H(f)$.

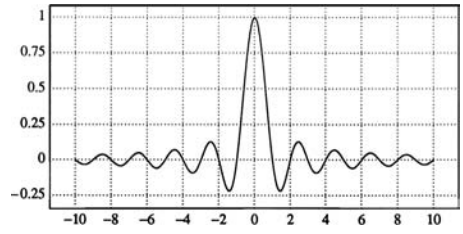


FIGURE 39.5 The sinc function.

39.3.2 Ideal Low-Pass Filter

We now turn to the second central problem: Given that it is theoretically possible to perform reconstruction, how may it be done? The answer lies with our earlier observation that sampling merely replicates the spectrum of the input signal, generating $G_{\text{high}}(f)$ in addition to $G(f)$. Therefore, the act of reconstruction requires us to completely suppress $G_{\text{high}}(f)$. This is done by multiplying $G_s(f)$ with $H(f)$, given as

$$H(f) = \begin{cases} 1, & |f| < f_{\max} \\ 0, & |f| \geq f_{\max} \end{cases} \tag{39.8}$$

$H(f)$ is known as an *ideal low-pass filter* and is depicted in Figure 39.4, where it is shown suppressing all frequency components above $f_{\max}$. This serves to discard the replicated spectra $G_{\text{high}}(f)$. It is ideal in the sense that the $f_{\max}$ cutoff frequency is strictly enforced as the transition point between the transmission and complete suppression of frequency components.

39.3.3 Sinc Function

In the spatial domain, the ideal low-pass filter is derived by computing the inverse Fourier transform of $H(f)$. This yields the sinc function shown in Figure 39.5. It is defined as

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \tag{39.9}$$

The reader should note the reciprocal relationship between the height and width of the ideal low-pass filter in the spatial and frequency domains. Let A denote the amplitude of the sinc function, and let its zero crossings be positioned at integer multiples of $1/2W$. The spectrum of this sinc function is a rectangular pulse of height $A/2W$ and width $2W$, with frequencies ranging from $-W$ to W. In our example above, $A = 1$ and $W = f_{\max} = 0.5$ cycles/pixel. This value for W is derived from the fact that digital images must not have more than one half cycle per pixel in order to conform to the Nyquist rate.

The sinc function is one instance of a large class of functions known as cardinal splines, which are interpolating functions defined to pass through zero at all but one data sample, where they have a value

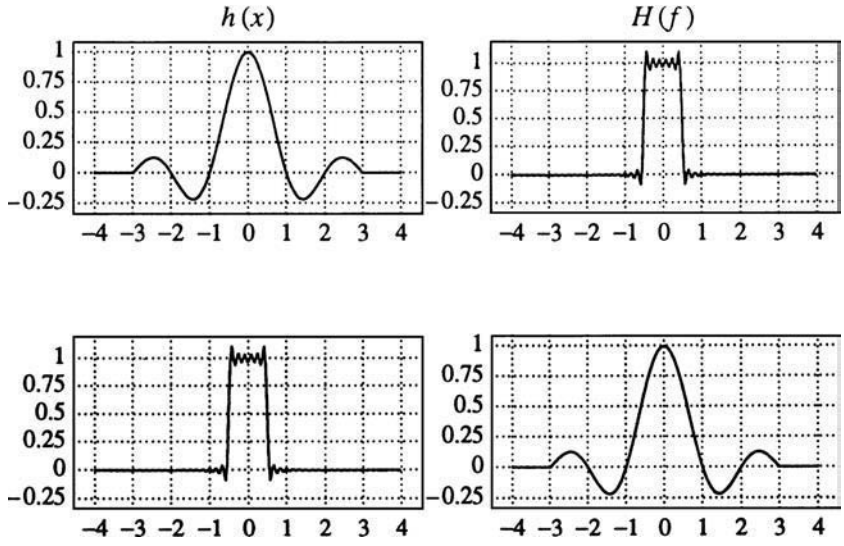


FIGURE 39.6 Truncation in one domain causes ringing in the other domain.

of one. This allows them to compute a continuous function that passes through the uniformly spaced data samples.

Since multiplication in the frequency domain is identical to convolution in the spatial domain, sinc (x) represents the convolution kernel used to evaluate any point x on the continuous input curve g given only the sampled data g_s:

$$g(x) = \text{sinc}\,(x) * g_s(x) = \int_{-\infty}^{\infty} \text{sinc}\,(\lambda)\, g_s(x - \lambda)\, d\lambda \tag{39.10}$$

Equation 39.10 highlights an important impediment to the practical use of the ideal low-pass filter. The filter requires an infinite number of neighboring samples (i.e., an infinite filter support) in order to precisely compute the output points. This is, of course, impossible owing to the finite number of data samples available. However, truncating the sinc function allows for approximate solutions to be computed at the expense of undesirable "ringing," i.e., ripple effects. These artifacts, known as the **Gibbs phenomenon**, are the overshoots and undershoots caused by reconstructing a signal with truncated frequency terms. The two rows in Figure 39.6 show that truncation in one domain leads to ringing in the other domain. This indicates that a truncated sinc function is actually a poor reconstruction filter because its spectrum has infinite extent and thereby fails to bandlimit the input.

In response to these difficulties, a number of approximating algorithms have been derived, offering a tradeoff between precision and computational expense. These methods permit local solutions that require the convolution kernel to extend only over a small neighborhood. The drawback, however, is that the frequency response of the filter has some undesirable properties. In particular, frequencies below $f_{\max}$ are tampered, and high frequencies beyond $f_{\max}$ are not fully suppressed. Thus, nonideal reconstruction does not permit us to exactly recover the continuous underlying signal without artifacts.

39.3.4 Nonideal Reconstruction

The process of nonideal reconstruction is depicted in Figure 39.7, which indicates that the input signal satisfies the two conditions necessary for exact reconstruction. First, the signal is bandlimited, since the replicated copies in the spectrum are each finite in extent. Second, the sampling frequency exceeds the Nyquist rate, since the copies do not overlap. However, this is where our ideal scenario ends. Instead of

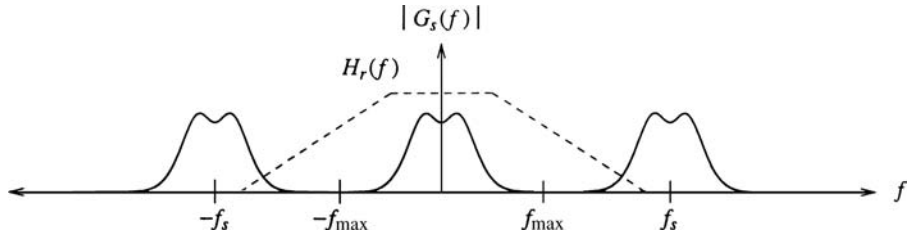


FIGURE 39.7 Nonideal reconstruction.

using an ideal low-pass filter to retain only the baseband spectrum components, a nonideal reconstruction filter is shown in the figure.

The filter response $H_r(f)$ deviates from the ideal response $H(f)$ shown in Figure 39.4. In particular, $H_r(f)$ does not discard all frequencies beyond f_{max}. Furthermore, that same filter is shown to attenuate some frequencies that should have remained intact. This brings us to the problem of assessing the quality of a filter.

The accuracy of a reconstruction filter can be evaluated by analyzing its frequency-domain characteristics. Of particular importance is the filter response in the passband and stopband. In this problem, the **passband** consists of all frequencies below f_{max}. The **stopband** contains all higher frequencies arising from the sampling process.

An ideal reconstruction filter, as described earlier, will completely suppress the stopband while leaving the passband intact. Recall that the stopband contains the offending high frequencies that, if allowed to remain, would prevent us from performing exact reconstruction. As a result, the sinc filter was devised to meet these goals and serve as the ideal reconstruction filter. Its kernel in the frequency domain applies unity gain to transmit the passband and zero gain to suppress the stopband.

The breakdown of the frequency domain into passband and stopband isolates two problems that can arise due to nonideal reconstruction filters. The first problem deals with the effects of imperfect filtering on the passband. Failure to impose unity gain on *all* frequencies in the passband will result in some combination of image smoothing and image sharpening. Smoothing, or blurring, will result when the frequency gains near the cutoff frequency start falling off. Image sharpening results when the high-frequency gains are allowed to exceed unity. This follows from the direct correspondence of visual detail to spatial frequency. Furthermore, amplifying the high passband frequencies yields a sharper transition between the passband and stopband, a property shared by the sinc function.

The second problem addresses nonideal filtering on the stopband. If the stopband is allowed to persist, high frequencies will exist that will contribute to aliasing (described later). Failure to fully suppress the stopband is a condition known as **frequency leakage**. This allows the offending frequencies to fold over into the passband range. These distortions tend to be more serious, since they are visually perceived more readily.

In the spatial domain, nonideal reconstruction is achieved by centering a finite-width kernel at the position in the data at which the underlying function is to be evaluated, i.e., reconstructed. This is an interpolation problem which, for equally spaced data, can be expressed as

$$f(x) = \sum_{k=0}^{K-1} f(x_k)h(x - x_k) \tag{39.11}$$

where h is the reconstruction kernel that weighs K data samples at x_k. Equation 39.11 formulates interpolation as a convolution operation. In practice, h is nearly always a symmetric kernel, i.e., $h(-x) = h(x)$. We shall assume this to be true in the discussion that follows.

The computation of one interpolated point is illustrated in Figure 39.8. The kernel is centered at x, the location of the point to be interpolated. The value of that point is equal to the sum of the values of the

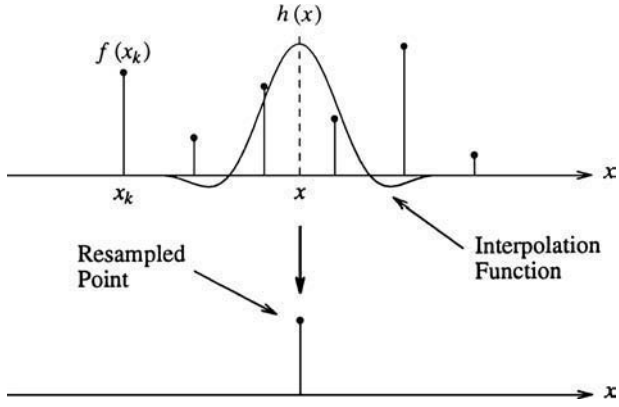


FIGURE 39.8 Interpolation of a single point.

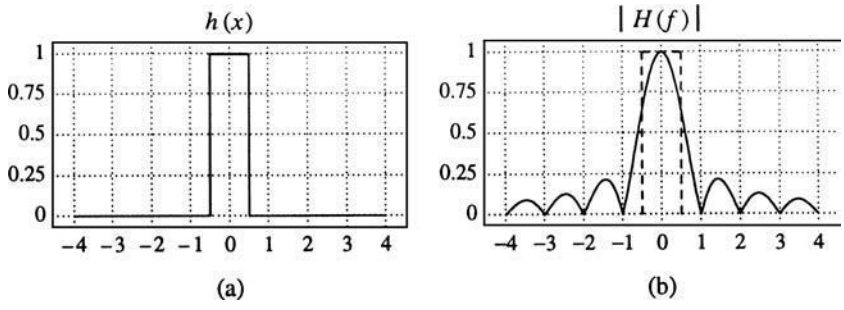


FIGURE 39.9 Box filter: (a) kernel, (b) Fourier transform.

discrete input scaled by the corresponding values of the reconstruction kernel. This follows directly from the definition of convolution.

39.4 Reconstruction Kernels

The numerical accuracy and computational cost of reconstruction are directly tied to the convolution kernel used for low-pass filtering. As a result, filter kernels are the target of design and analysis in the creation and evaluation of reconstruction algorithms. They are subject to conditions influencing the tradeoff between accuracy and efficiency. This section reviews several common nonideal reconstruction filter kernels in the order of their complexity: box filter, triangle filter, cubic convolution, and windowed sinc functions.

39.4.1 Box Filter

The box filter kernel is defined as

$$h(x) = \begin{cases} 1, & -0.5 < x \leq 0.5 \\ 0, & \text{otherwise} \end{cases} \tag{39.12}$$

Various other names are used to denote this simple kernel, including the sample-and-hold function and Fourier window. The kernel and its Fourier transform are shown in Figure 39.9.

Convolution in the spatial domain with the rectangle function h is equivalent in the frequency domain to multiplication with a sinc function. Due to the prominent side lobes and infinite extent, a sinc function

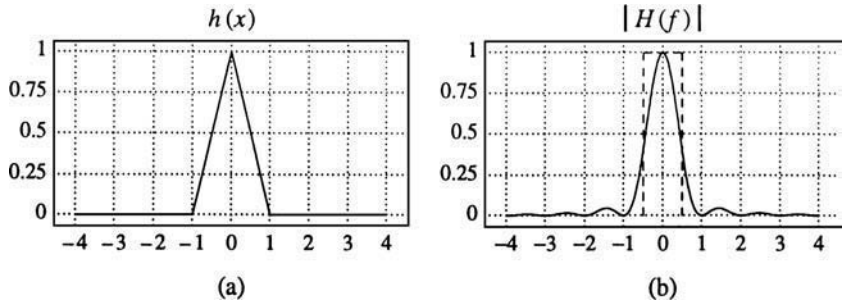


FIGURE 39.10 Triangle filter: (a) kernel, (b) Fourier transform.

makes a poor low-pass filter. Consequently, this filter kernel has a poor frequency-domain response relative to that of the ideal low-pass filter. The ideal filter, drawn as a dashed rectangle, is characterized by unity gain in the passband and zero gain in the stopband. This permits all low frequencies (below the cutoff frequency) to pass and all higher frequencies to be suppressed.

39.4.2 Triangle Filter

The triangle filter kernel is defined as

$$h(x) = \begin{cases} 1 - |x|, & 0 \le |x| < 1 \\ 0, & 1 \le |x| \end{cases} \tag{39.13}$$

The kernel h is also referred to as a tent filter, roof function, Chateau function, or Bartlett window.

This kernel corresponds to a reasonably good low-pass filter in the frequency domain. As shown in Figure 39.10, its response is superior to that of the box filter. In particular, the side lobes are far less prominent, indicating improved performance in the stopband. Nevertheless, a significant amount of spurious high-frequency components continues to leak into the passband, contributing to some aliasing. In addition, the passband is moderately attenuated, resulting in image smoothing.

39.4.3 Cubic Convolution

The cubic convolution kernel is a third-degree approximation to the sinc function. It is symmetric, space-invariant, and composed of piecewise cubic polynomials:

$$h(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1, & 0 \le |x| < 1 \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a, & 1 \le |x| < 2 \\ 0, & 2 \le |x| \end{cases} \tag{39.14}$$

where $-3 < a < 0$ is used to make h resemble the sinc function.

Of all the choices for a, the value -1 is preferable if visually enhanced results are desired. That is, the image is sharpened, making visual detail perceived more readily. However, the results are not mathematically precise, where precision is measured by the order of the Taylor series. To maximize this order, the value $a = -0.5$ is preferable. A cubic convolution kernel with $a = -0.5$ and its spectrum are shown in Figure 39.11.

39.4.4 Windowed Sinc Function

Sampling theory establishes that the sinc function is the ideal interpolation kernel. Although this interpolation filter is exact, it is not practical, since it is an infinite impulse response (IIR) filter defined by a slowly converging infinite sum. Nevertheless, it is perfectly reasonable to consider the effects of using a truncated, and therefore finite, sinc function as the interpolation kernel.

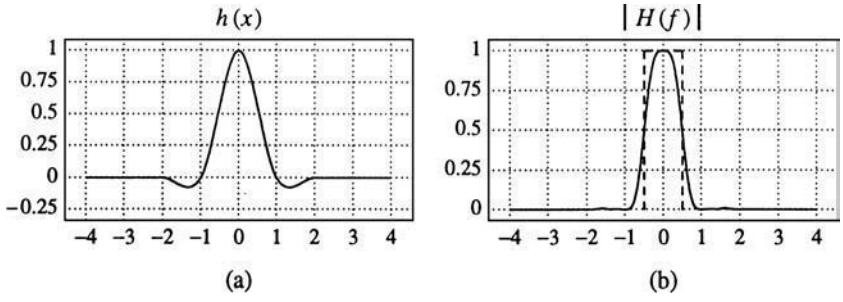


FIGURE 39.11 Cubic convolution: (a) kernel ($a = -0.5$), (b) Fourier transform.

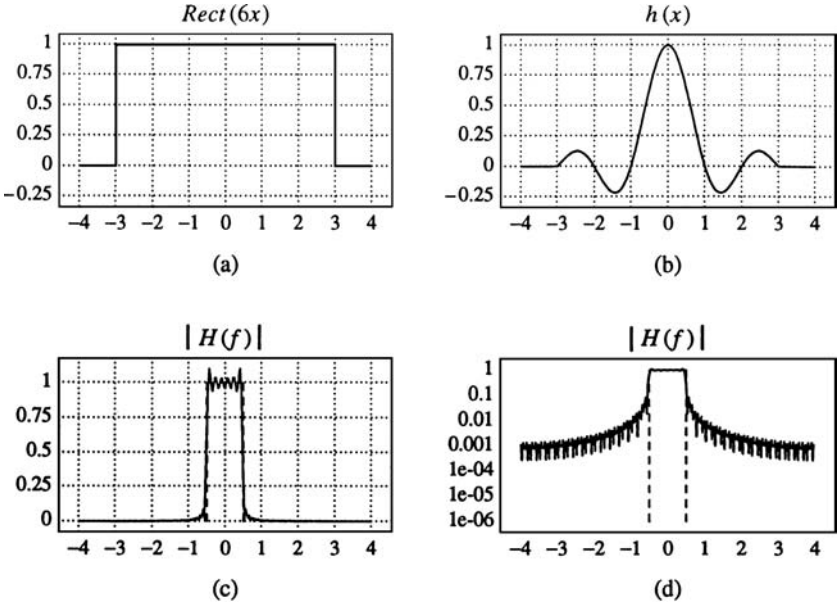


FIGURE 39.12 (a) Rectangular window; (b) windowed sinc; (c) spectrum; (d) log plot.

The results of this operation are predicted by sampling theory, which demonstrates that truncation in one domain leads to ringing in the other domain. This is due to the fact that truncating a signal is equivalent to multiplying it with a rectangle function Rect(x), defined as the box filter of Equation 39.12. Since multiplication in one domain is convolution in the other, truncation amounts to convolving the signal's spectrum with a sinc function, the transform pair of Rect(x). Since the stopband is no longer eliminated, but rather attenuated by a ringing filter (i.e., a sinc), the input is not bandlimited and aliasing artifacts are introduced. The most typical problems occur at step edges, where the Gibbs phenomena becomes noticeable in the form of undershoots, overshoots, and ringing in the vicinity of edges.

The Rect function above served as a window, or kernel, that weighs the input signal. In Figure 39.12(a), we see the *Rect* window extended over three pixels on each side of its center, i.e., Rect($6x$) is plotted. The corresponding windowed sinc function $h(x)$ is shown in Figure 39.12(b). This is simply the product of the sinc function with the window function, i.e., sinc (x) Rect($6x$). Its spectrum, shown in Figure 39.12(c), is nearly an ideal low-pass filter. Although it has a fairly sharp transition from the passband to the stopband, it is plagued by ringing. In order to more clearly see the values in the spectrum, we use a logarithmic scale for the vertical axis of the spectrum in Figure 39.12(d). The next few figures will use this same four-part format.

Ringing can be mitigated by using a different windowing function exhibiting smoother falloff than the rectangle. The resulting windowed sinc function can yield better results. However, since slow falloff requires larger windows, the computation remains costly.

Aside from the rectangular window mentioned above, the most frequently used window functions are Hann, Hamming, Blackman, and Kaiser. These filters identify a quantity known as the ripple ratio, defined as the ratio of the maximum side-lobe amplitude to the main-lobe amplitude. Good filters will have small ripple ratios to achieve effective attenuation in the stopband. A tradeoff exists, however, between ripple ratio and main-lobe width. Therefore, as the ripple ratio is decreased, the main-lobe width is increased. This is consistent with the reciprocal relationship between the spatial and frequency domains, i.e., narrow bandwidths correspond to wide spatial functions.

In general, though, each of these smooth window functions is defined over a small finite extent. This is tantamount to multiplying the smooth window with a rectangle function. While this is better than the Rect function alone, there will inevitably be some form of aliasing. Nevertheless, the window functions described below offer a good compromise between ringing and blurring.

39.4.5 Hann and Hamming Windows

The Hann and Hamming windows are defined as

$$\text{Hann/Hamming}(x) = \begin{cases} \alpha + (1-\alpha)\cos\frac{2\pi x}{N-1}, & |x| < \frac{N-1}{2} \\ 0, & \text{otherwise} \end{cases} \tag{39.15}$$

where N is the number of samples in the windowing function. The two windowing functions differ in their choice of α. In the Hann window $\alpha = 0.5$, and in the Hamming window $\alpha = 0.54$. Since they both amount to a scaled and shifted cosine function, they are also known as the raised cosine window. The Hann window is illustrated in Figure 39.13. Notice that the passband is only slightly attenuated, but the stopband continues to retain high-frequency components in the stopband, albeit less than that of Rect(x).

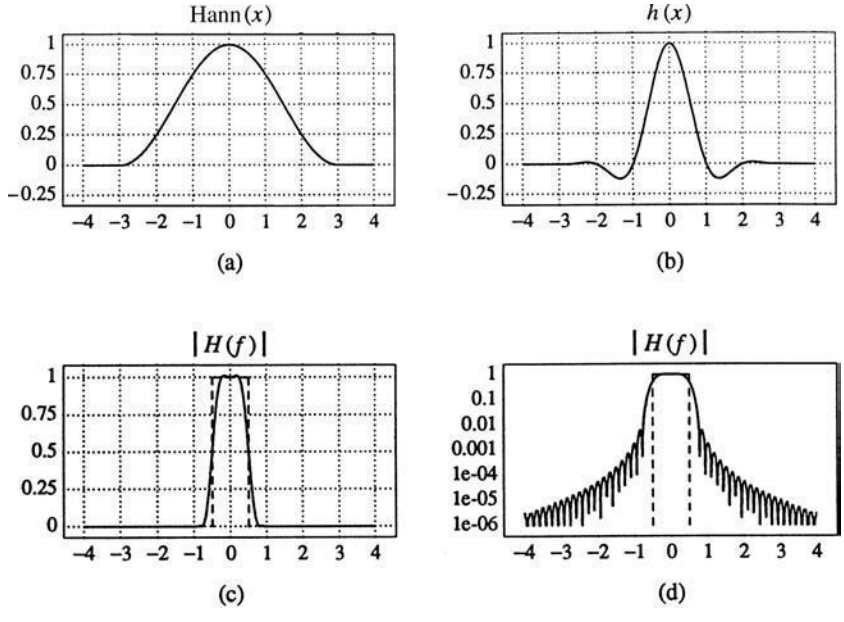


FIGURE 39.13 (a) Hann window; (b) windowed sinc; (c) spectrum; (d) log plot.

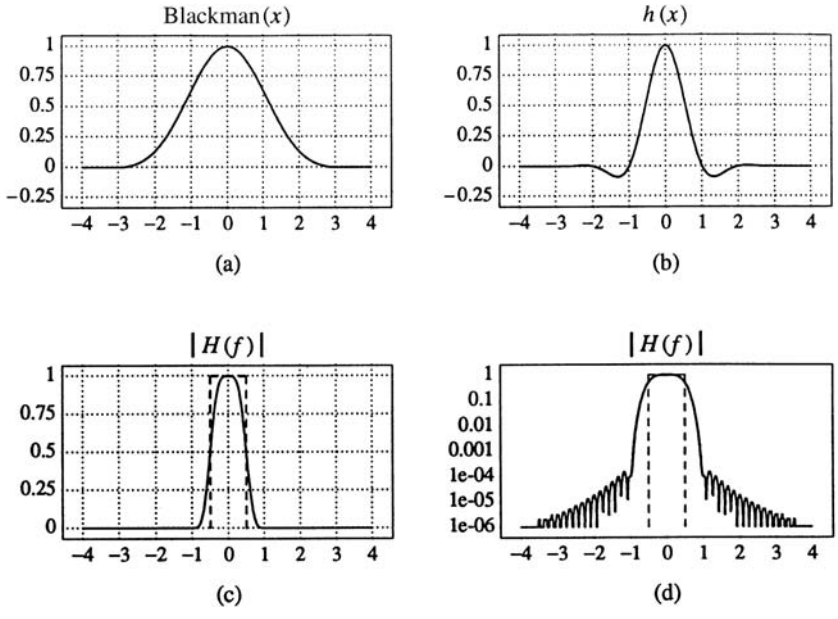


FIGURE 39.14 (a) Blackman window; (b) windowed sinc; (c) spectrum; (d) log plot.

39.4.6 Blackman Window

The Blackman window is similar to the Hann and Hamming windows. It is defined as

$$\text{Blackman}(x) = \begin{cases} 0.42 + 0.5\cos\frac{2\pi x}{N-1} + 0.08\cos\frac{4\pi x}{N-1}, & |x| < \frac{N-1}{2} \\ 0, & \text{otherwise} \end{cases} \tag{39.16}$$

The purpose of the additional cosine term is to further reduce the ripple ratio. This window function is shown in Figure 39.14.

39.4.7 Kaiser Window

The Kaiser window is defined as

$$\text{Kaiser}(x) = \begin{cases} \frac{I_0(\beta)}{I_0(\alpha)}, & |x| < \frac{N-1}{2} \\ 0, & \text{otherwise} \end{cases} \tag{39.17}$$

where I_0 is the zeroth-order Bessel function of the first kind, α is a free parameter, and

$$\beta = \alpha\left[1 - \left(\frac{2x}{N-1}\right)^2\right]^{1/2} \tag{39.18}$$

The Kaiser window leaves the filter designer much flexibility in controlling the ripple ratio by adjusting the parameter α. As α is increased, the level of sophistication of the window function grows as well. Therefore, the rectangular window corresponds to a Kaiser window with $\alpha = 0$, while more sophisticated windows such as the Hamming window correspond to $\alpha = 5$.

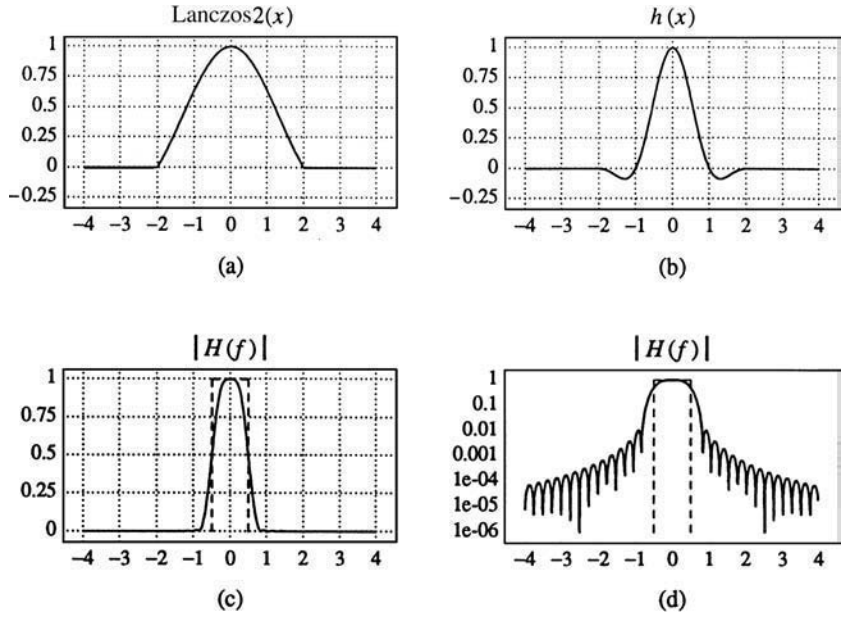


FIGURE 39.15 (a) Lanczos2 window; (b) windowed sinc; (c) spectrum; (d) log plot.

39.4.8 Lanczos Window

The Lanczos window is based on the sinc function rather than cosines as used in the previous methods. The two-lobed Lanczos window function is defined as

$$\text{Lanczos2}(x) = \begin{cases} \frac{\sin(\pi x/2)}{\pi x/2}, & 0 \leq |x| < 2 \\ 0, & 2 \leq |x| \end{cases} \tag{39.19}$$

The Lanczos2 window function, shown in Figure 39.15, is the central lobe of a sinc function. It is wide enough to extend over two lobes of the ideal low-pass filter, i.e., a second sinc function. This formulation can be generalized to an N-lobed window function by replacing the value 2 in Equation 39.19 with the value N. Larger N results in superior frequency response.

39.5 Aliasing

If the two reconstruction conditions outlined earlier are not met, sampling theory predicts that exact reconstruction is *not* possible. This phenomenon, known as **aliasing**, occurs when signals are not bandlimited or when they are undersampled, i.e., $f_s \leq f_{\text{Nyquist}}$. In either case there will be unavoidable overlapping of spectral components, as in Figure 39.16. Notice that the irreproducible high frequencies fold over into the low frequency range. As a result, frequencies originally beyond $f_{\max}$ will, upon reconstruction, appear in the form of much *lower* frequencies. In comparison with the spurious high frequencies retained by nonideal reconstruction filters, the spectral components passed due to undersampling are more serious, since they actually corrupt the components in the original signal.

Aliasing refers to the higher frequencies becoming aliased and indistinguishable from the lower-frequency components in the signal if the sampling rate falls below the Nyquist frequency. In other words, undersampling causes high-frequency components to appear as spurious low frequencies. This is depicted in Figure 39.17, where a high-frequency signal appears as a low-frequency signal after sampling it too sparsely. In digital images, the Nyquist rate is determined by the highest frequency that can be

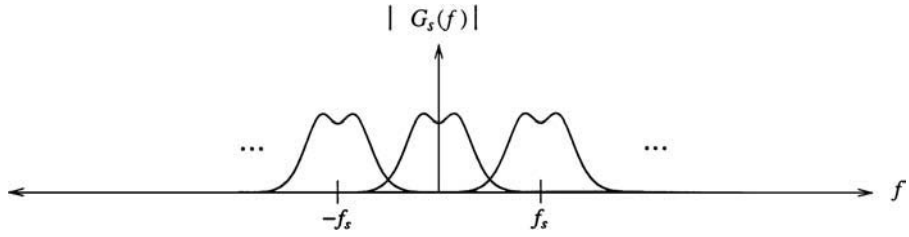


FIGURE 39.16 Overlapping spectral components give rise to aliasing.

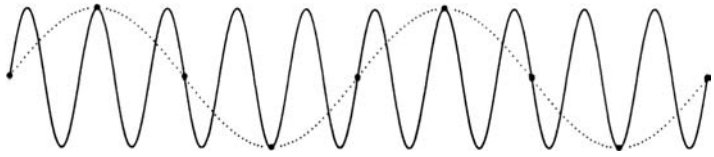


FIGURE 39.17 Aliasing artifacts due to undersampling.

displayed: one cycle every two pixels. Therefore, any attempt to display higher frequencies will produce similar artifacts.

There is sometimes a misconception in the computer graphics literature that jagged (staircased) edges are always a symptom of aliasing. This is only partially true. Technically, jagged edges arise from high frequencies introduced by inadequate reconstruction. Since these high frequencies are not corrupting the low-frequency components, no aliasing is actually taking place. The confusion lies in that the suggested remedy of increasing the sampling rate is also used to eliminate aliasing. Of course, the benefit of increasing the sampling rate is that the replicated spectra are now spaced farther apart from each other. This relaxes the accuracy constraints for reconstruction filters to perform ideally in the stopband, where they must suppress all components beyond some specified cutoff frequency. In this manner, the same nonideal filters will produce less objectionable output.

It is important to note that a signal may be densely sampled (far above the Nyquist rate) and continue to appear jagged if a zero-order reconstruction filter is used. Box filters used for pixel replication in real-time hardware zooms are a common example of poor reconstruction filters. In this case, the signal is clearly not aliased but rather poorly reconstructed. The distinction between reconstruction and aliasing artifacts becomes clear when we notice that the appearance of jagged edges is improved by blurring. For example, it is not uncommon to step back from an image exhibiting excessive blockiness in order to see it more clearly. This is a defocusing operation that attenuates the high frequencies admitted through nonideal reconstruction. On the other hand, once a signal is truly undersampled, there is no postprocessing possible to improve its condition. After all, applying an ideal low-pass (reconstruction) filter to a spectrum whose components are already overlapping will only blur the result, not rectify it.

39.6 Antialiasing

The filtering necessary to combat aliasing is known as **antialiasing**. In order to determine corrective action, we must directly address the two conditions necessary for exact signal reconstruction. The first solution calls for low-pass filtering *before* sampling. This method, known as **prefiltering**, bandlimits the signal to levels below $f_{\max}$, thereby eliminating the offending high frequencies. Notice that the frequency at which the signal is to be sampled imposes limits on the allowable bandwidth. This is often necessary when the output

sampling grid must be fixed to the resolution of an output device, e.g., screen resolution. Therefore, aliasing is often a problem that is confronted when a signal is forced to conform to an inadequate resolution due to physical constraints. As a result, it is necessary to bandlimit, or narrow, the input spectrum to conform to the allotted bandwidth as determined by the sampling frequency.

The second solution is to point-sample at a higher frequency. In doing so, the replicated spectra are spaced farther apart, thereby separating the overlapping spectra tails. This approach theoretically implies sampling at a resolution determined by the highest frequencies present in the signal. Since a surface viewed obliquely can give rise to arbitrarily high frequencies, this method may require extremely high resolution. Whereas the first solution adjusts the bandwidth to accommodate the fixed sampling rate f_s, the second solution adjusts f_s to accommodate the original bandwidth. Antialiasing by sampling at the highest frequency is clearly superior in terms of image quality. This is, of course, operating under different assumptions regarding the possibility of varying f_s. In practice, antialiasing is performed through a combination of these two approaches. That is, the sampling frequency is increased so as to reduce the amount of bandlimiting to a minimum.

39.6.1 Point Sampling

The naive approach for generating an output image is to perform **point sampling**, where each output pixel is a single sample of the input image taken independently of its neighbors (Figure 39.18). It is clear that information is lost between the samples and that aliasing artifacts may surface if the sampling density is not sufficiently high to characterize the input. This problem is rooted in the fact that intermediate intervals between samples, which should have some influence on the output, are skipped entirely.

The Star image is a convenient example that overwhelms most resampling filters due to the infinitely high frequencies found toward the center. Nevertheless, the extent of the artifacts is related to the quality of the filter and the actual spatial transformation. Figure 39.19 shows two examples of the moiré effects that can appear when a signal is undersampled using point sampling. In Figure 39.19(a), one out of every two pixels in the Star image was discarded to reduce its dimension. In Figure 39.19(b), the artifacts of undersampling are more pronounced, as only one out of every four pixels is retained. In order to see the small images more clearly, they are magnified using cubic spline reconstruction. Clearly, these examples show that point sampling behaves poorly in high-frequency regions.

Aliasing can be reduced by point sampling at a higher resolution. This raises the Nyquist limit, accounting for signals with higher bandwidths. Generally, though, the display resolution places a limit on the highest frequency that can be displayed, and thus limits the Nyquist rate to one cycle every two pixels. Any attempt to display higher frequencies will produce aliasing artifacts such as moiré patterns and jagged edges. Consequently, antialiasing algorithms have been derived to bandlimit the input *before* resampling onto the output grid.

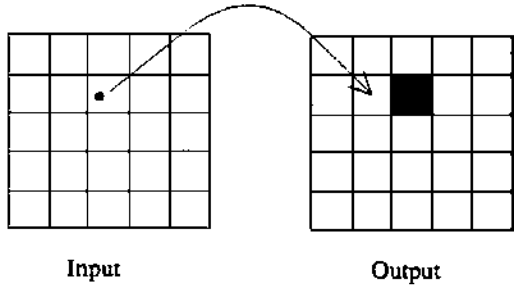


FIGURE 39.18 Point sampling.

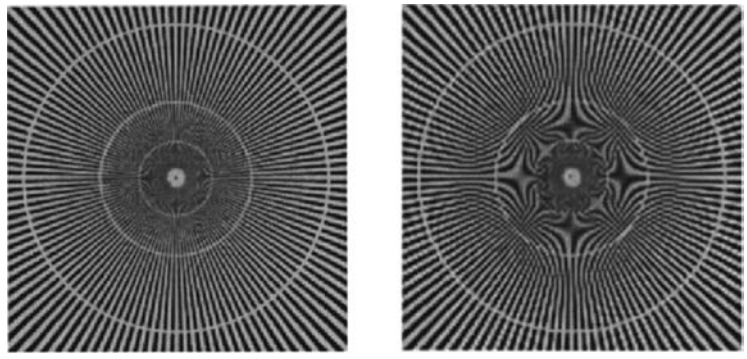


FIGURE 39.19 Aliasing due to point sampling: (a) 1/2 and (b) 1/4 scale.

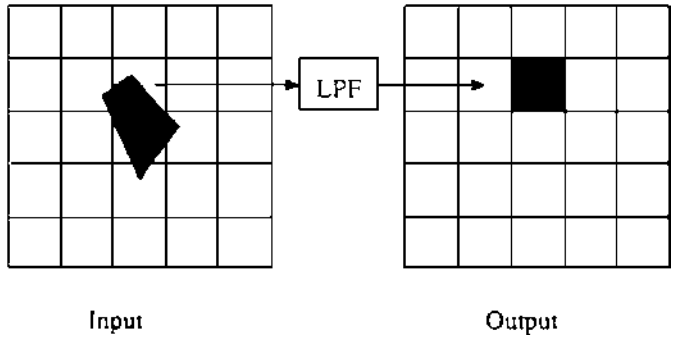


FIGURE 39.20 Area sampling.

39.6.2 Area Sampling

The basic flaw in point sampling is that a discrete pixel actually represents an area, not a point. In this manner, each output pixel should be considered a window looking onto the input image. Rather than sampling a point, we must instead apply a low-pass filter (LPF) upon the projected area in order to properly reflect the information content being mapped onto the output pixel. This approach, depicted in Figure 39.20, is called **area sampling**, and the projected area is known as the **preimage**. The low-pass filter comprises the prefiltering stage. It serves to defeat aliasing by bandlimiting the input image prior to resampling it onto the output grid. In the general case, prefiltering is defined by the convolution integral

$$g(x, y) = \int\int f(u, v)h(x - u, y - v)\, du\, dv \tag{39.20}$$

where f is the input image, g is the output image, h is the filter kernel, and the integration is applied to all $[u, v]$ points in the preimage.

Images produced by area sampling are demonstrably superior to those produced by point sampling. Figure 39.21 shows the Star image subjected to the same downsampling transformation as that in Figure 39.19. Area sampling was implemented by applying a box filter (i.e., unweighted averaging) to the Star image before point sampling. Notice that antialiasing through area sampling has traded moiré patterns for some blurring. Although there is no substitute for high-resolution imagery, filtering can make lower resolution less objectionable by attenuating aliasing artifacts.

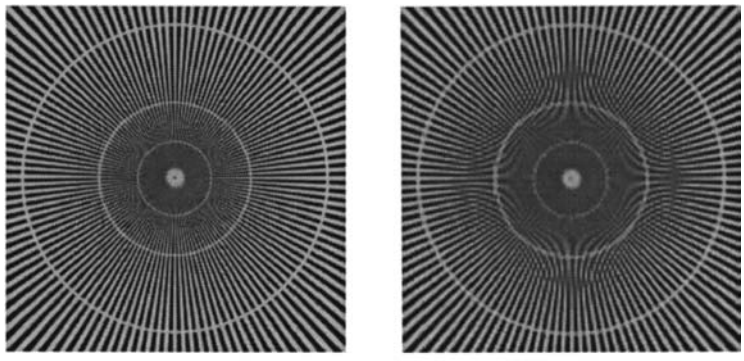


FIGURE 39.21 Aliasing due to area sampling: (a) 1/2 and (b) 1/4 scale.

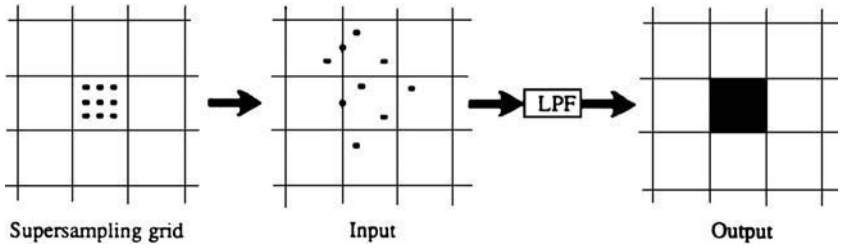


FIGURE 39.22 Supersampling.

39.6.3 Supersampling

The process of using more than one regularly spaced sample per pixel is known as **supersampling**. Each output pixel value is evaluated by computing a weighted average of the samples taken from their respective preimages. For example, if the supersampling grid is three times denser than the output grid (i.e., there are nine grid points per pixel area), each output pixel will be an average of the nine samples taken from its projection in the input image. If, say, three samples hit a green object and the remaining six samples hit a blue object, the composite color in the output pixel will be one-third green and two-thirds blue, assuming a box filter is used.

Supersampling reduces aliasing by bandlimiting the input signal. The purpose of the high-resolution supersampling grid is to refine the estimate of the preimages seen by the output pixels. The samples then enter the prefiltering stage, consisting of a low-pass filter. This permits the input to be resampled onto the (relatively) low-resolution output grid without any offending high frequencies introducing aliasing artifacts. In Figure 39.22 we see an output pixel subdivided into nine subpixel samples which each undergo inverse mapping, sampling the input at nine positions. Those nine values then pass through a low-pass filter to be averaged into a single output value.

Supersampling was used to achieve antialiasing in Figure 39.1 for pixels near the horizon. There are two problems, however, associated with straightforward supersampling. The first problem is that the newly designated high frequency of the prefiltered image continues to be fixed. Therefore, there will always be sufficiently higher frequencies that will alias. The second problem is cost. In our example, supersampling will take nine times longer than point sampling. Although there is a clear need for the additional computation, the dense placement of samples can be optimized. Adaptive supersampling is introduced to address these drawbacks.

39.6.4 Adaptive Supersampling

In **adaptive supersampling**, the samples are distributed more densely in areas of high intensity variance. In this manner, supersamples are collected only in regions that warrant their use. Early work in adaptive supersampling for computer graphics is described in Whitted [1980]. The strategy is to subdivide areas between previous samples when an edge, or some other high-frequency pattern, is present. Two approaches to adaptive supersampling have been described in the literature. The first approach allows sampling density to vary as a function of local image variance [Lee et al. 1985, Kajiya 1986]. A second approach introduces two levels of sampling densities: a regular pattern for most areas and a higher-density pattern for regions demonstrating high frequencies. The regular pattern simply consists of one sample per output pixel. The high density pattern involves local supersampling at a rate of 4 to 16 samples per pixel. Typically, these rates are adequate for suppressing aliasing artifacts.

A strategy is required to determine where supersampling is necessary. In Mitchell [1987], the author describes a method in which the image is divided into small square supersampling cells, each containing eight or nine of the low-density samples. The entire cell is supersampled if its samples exhibit excessive variation. In Lee et al. [1985], the variance of the samples is used to indicate high frequency. It is well known, however, that variance is a poor measure of visual perception of local variation. Another alternative is to use contrast, which more closely models the nonlinear response of the human eye to rapid fluctuations in light intensities [Caelli 1981]. Contrast is given as

$$C = \frac{I_{\max} - I_{\min}}{I_{\max} + I_{\min}} \tag{39.21}$$

Adaptive sampling reduces the number of samples required for a given image quality. The problem with this technique, however, is that the variance measurement is itself based on point samples, and so this method can fail as well. This is particularly true for subpixel objects that do not cross pixel boundaries. Nevertheless, adaptive sampling presents a far more reliable and cost-effective alternative to supersampling.

39.7 Prefiltering

Area sampling can be accelerated if constraints on the filter shape are imposed. Pyramids and preintegrated tables are introduced to approximate the convolution integral with a constant number of accesses. This compares favorably against direct convolution, which requires a large number of samples that grow proportionately to preimage area. As we shall see, though, the filter area will be limited to squares or rectangles, and the kernel will consist of a box filter. Subsequent advances have extended their use to more general cases with only marginal increases in cost.

39.7.1 Pyramids

Pyramids are multiresolution data structures commonly used in image processing and computer vision. They are generated by successively bandlimiting and subsampling the original image to form a hierarchy of images at ever decreasing resolutions. The original image serves as the base of the pyramid, and its coarsest version resides at the apex. Thus, in a lower-resolution version of the input, each pixel represents the average of some number of pixels in the higher-resolution version.

The resolution of successive levels typically differs by a power of two. This means that successively coarser versions each have one-quarter of the total number of pixels as their adjacent predecessors. The memory cost of this organization is modest: $1 + 1/4 + 1/16 + \cdots = 4/3$ times that needed for the original input. This requires only 33% more memory.

To filter a preimage, one of the pyramid levels is selected based on the size of its bounding square box. That level is then point sampled and assigned to the respective output pixel. The primary benefit of this approach is that the cost of the filter is constant, requiring the same number of pixel accesses independent of the filter size. This performance gain is the result of the filtering that took place while creating the

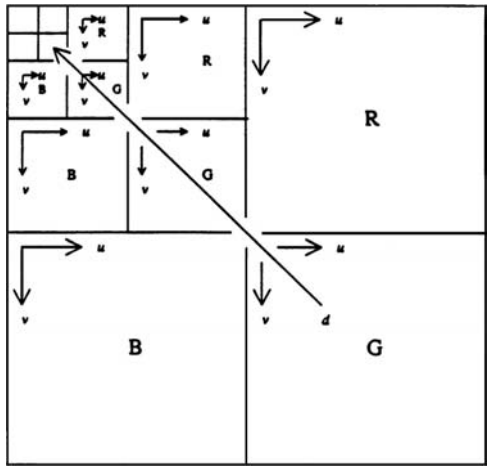


FIGURE 39.23 Mip-map memory organization.

pyramid. Furthermore, if preimage areas are adequately approximated by squares, the direct convolution methods amount to point-sampling a pyramid. This approach was first applied to texture mapping in Catmull [1974] and described in Dungan et al. [1978].

There are several problems with the use of pyramids. First, the appropriate pyramid level must be selected. A coarse level may yield excessive blur, while the adjacent finer level may be responsible for aliasing due to insufficient bandlimiting. Second, preimages are constrained to be squares. This proves to be a crude approximation for elongated preimages. For example, when a surface is viewed obliquely the texture may be compressed along one dimension. Using the largest bounding square will include the contributions of many extraneous samples and result in excessive blur. These two issues were addressed in Williams [1983] and Crow [1984], respectively, along with extensions proposed by other researchers.

Williams proposed a pyramid organization called *mip map* to store color images at multiple resolutions in a convenient memory organization [Williams 1983]. The acronym "mip" stands for "multum in parvo," a Latin phrase meaning "much in little." The scheme supports trilinear interpolation, where both intra- and interlevel interpolation can be computed using three normalized coordinates: u, v, and d. Both u and v are spatial coordinates used to access points within a pyramid level. The d coordinate is used to index, and interpolate between, different levels of the pyramid. This is depicted in Figure 39.23.

The quadrants touching the east and south borders contain the original red, green, and blue (RGB) components of the color image. The remaining upper left quadrant contains all the lower-resolution copies of the original. The memory organization depicted in Figure 39.23 clearly supports the earlier claim that the memory cost is 4/3 times that required for the original input. Each level is shown indexed by the $[u, v, d]$ coordinate system, where d is shown slicing through the pyramid levels. Since corresponding points in different pyramid levels have indices which are related by some power of two, simple binary shifts can be used to access these points across the multiresolution copies. This is a particularly attractive feature for hardware implementation.

The primary difference between mip maps and ordinary pyramids is the trilinear interpolation scheme possible with the $[u, v, d]$ coordinate system. Since they allow a continuum of points to be accessed, mip maps are referred to as pyramidal parametric data structures. In Williams's implementation, a box filter was used to create the mip maps, and a triangle filter was used to perform intra- and interlevel interpolation. The value of d must be chosen to balance the tradeoff between aliasing and blurring. Heckbert suggests

$$d^2 = \max\left(\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2, \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right) \tag{39.22}$$

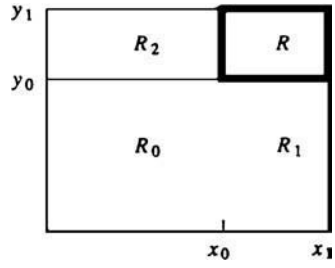


FIGURE 39.24 Summed-area table calculation.

where d is proportional to the span of the preimage area, and the partial derivatives can be computed from the surface projection [Heckbert 1983].

39.7.2 Summed-Area Tables

An alternative to pyramidal filtering was proposed by Crow [1984]. It extends the filtering possible in pyramids by allowing rectangular areas, oriented parallel to the coordinate axes, to be filtered in constant time. The central data structure is a preintegrated buffer of intensities, known as the **summed-area table**. This table is generated by computing a running total of the input intensities as the image is scanned along successive scanlines. For every position P in the table, we compute the sum of intensities of pixels contained in the rectangle between the origin and P. The sum of all intensities in any rectangular area of the input may easily be recovered by computing a sum and two differences of values taken from the table. For example, consider the rectangles R_0, R_1, R_2, and R shown in Figure 39.24. The sum of intensities in rectangle R can be computed by considering the sum at $[x1, y1]$, and discarding the sums of rectangles R_0, R_1, and R_2. This corresponds to removing all areas lying below and to the left of R. The resulting area is rectangle R, and its sum S is given as

$$S = T[x1, y1] - T[x1, y0] - T[x0, y1] + T[x0, y0] \tag{39.23}$$

where $T[x, y]$ is the value in the summed-area table indexed by coordinate pair $[x, y]$.

Since $T[x1, y0]$ and $T[x0, y1]$ both contain R_0, the sum of R_0 was subtracted twice in Equation 39.23. As a result, $T[x0, y0]$ was added back to restore the sum. Once S is determined, it is divided by the area of the rectangle. This gives the average intensity over the rectangle, a process equivalent to filtering with a Fourier window (box filtering).

There are two problems with the use of summed-area tables. First, the filter area is restricted to rectangles. This is addressed in Glassner [1986], where an adaptive, iterative technique is proposed for obtaining arbitrary filter areas by removing extraneous regions from the rectangular bounding box. Second, the summed-area table is restricted to box filtering. This, of course, is attributed to the use of unweighted averages that keeps the algorithm simple. In Perlin [1985] and Heckbert [1986], the summed-area table is generalized to support more sophisticated filtering by repeated integration.

It is shown that by repeatedly integrating the summed-area table n times, it is possible to convolve an orthogonally oriented rectangular region with an nth-order box filter (B-spline). The output value is computed by using $(n+1)^2$ weighted samples from the preintegrated table. Since this result is independent of the size of the rectangular region, this method offers a great reduction in computation over that of direct convolution. Perlin called this a selective image filter because it allows each sample to be blurred by different amounts.

Repeated integration has rather high memory costs relative to pyramids. This is due to the number of bits necessary to retain accuracy in the large summations. Nevertheless, it allows us to filter rectangular or elliptical regions, rather than just squares as in pyramid techniques. Since pyramid and summed-area tables both require a setup time, they are best suited for input that is intended to be used repeatedly, i.e., stationary background scenes or texture maps. In this manner, the initialization overhead can be amortized over each use.

39.8 Example: Image Scaling

In this section, we demonstrate the role of reconstruction and antialiasing in image scaling. The resampling process will be explained in one dimension rather than two, since resampling is carried out on each axis independently. For example, the horizontal scanlines are first processed, yielding an intermediate image, which then undergoes a second pass of interpolation in the vertical direction. The result is independent of the order: processing the vertical lines before the horizontal lines gives the same results.

A skeleton of a C program that resizes an image in two passes is given below. The input image is assumed to have **INheight** rows and **INwidth** columns. The first pass visits each row and resamples them to have width **OUTwidth**. The second pass visits each column of the newly formed intermediate image and resamples them to have height **OUTheight**:

```
INwidth   =  input image width (pixels/row);
INheight  =  input image height (rows/image);
OUTwidth  =  output image width (pixels/row);
OUTheight =  output image height (rows/image);
filter    =  convolution kernel to use to filter image;
offset    =  inter-pixel offset (stride);

allocate an intermediate image of size OUTwidth by INheight;

offset = 1;
for(y=0; y<INheight; y++) {      /* process rows */
     src = pointer to row y of input image;
     dst = pointer to row y of intermediate image;
     resample1-D(src, dst, INwidth, OUTwidth, filter, offset);
}

offset = OUTwidth;
for(x=0; x<w; x++) {             /* process columns */
     src = pointer to column x of intermediate image;
     dst = pointer to column x of output image;
     resample1-D(src, dst, INheight, OUTheight, filter, offset);
}
```

Function **resample1-D** is the workhorse of the resizing operation. The inner workings of this function will be described later. In addition to the input and output pointers and dimensions, **resample1-D** must be passed **filter**, an integer code specifying which convolution kernel to apply. In order to operate on both rows and columns, the parameter **offset** is given to denote the distance between successive pixels in the scanline. Horizontal scanlines (rows) have **offset** $=$ **1** and vertical scanlines (columns) have **offset** $=$ **OUTwidth**.

There are two operations which **resample1-D** must be able to handle: magnification and minification. As mentioned earlier, these two operations are closely related. They both require us to project each output sample into the input, center a kernel, and convolve. The only difference between magnification and minification is the shape of the kernel. The magnification kernel is fixed at $h(x)$, whereas the minification kernel is $ah(ax)$, for $a < 1$. The width of the kernel for minification is due to the need for a low-pass filter to perform antialiasing. That filter now has a narrower response than that of the interpolation function. Consequently, we exploit the following well-known Fourier transform pair:

$$h(ax) \longleftrightarrow \frac{1}{a} H\left(\frac{f}{a}\right) \tag{39.24}$$

This equation expresses the reciprocal relationship between the spatial and frequency domains. Notice that multiplying the spatial axis by a factor of a results in dividing the frequency axis and the spectrum values

by that same factor. Since we want the spectrum values to be left intact, we use $ah(ax)$ as the convolution kernel for blurring, where $a > 1$. This implies that the shape of the kernel changes as a function of scale factor when we are downsampling the input. This was not the case for magnification.

A straightforward method to perform 1-D resampling is given below. It details the inner workings of the **resample1-D** function outlined earlier. In addition, a few interpolation functions are provided. More such functions can easily be added by the user:

```
#define PI            3.14159265358979311160E0
#define SGN(A)        ((A) > 0 ? 1 : ((A) < 0 ? -1 : 0 ))
#define FLOOR(A)      ((int) (A))
#define CEILING(A)    ((A)==FLOOR(A) ? FLOOR(A) : SGN(A)+FLOOR(A))
#define CLAMP(A,L,H)  ((A)<=(L) ? (L) : (A)<=(H) ? (A) : (H))

resample1-D(IN, OUT, INlen, OUTlen, filtertype, offset)
unsigned char *IN, *OUT;
int INlen, OUTlen, filtertype, offset;
{
    int i;
    int left, right;  /* kernel extent in input */
    int pixel;        /* input pixel value */
    double u, x;      /* input (u) , output (x) */
    double scale;     /* resampling scale factor */
    double fwidth;    /* filter width (support) */
    double fscale;    /* filter amplitude scale */
    double weight;    /* kernel weight */
    double acc;       /* convolution accumulator */

    scale = (double) OUTlen / INlen;

    switch(filtertype) {
    case 0: filter = boxFilter; /* box filter (nearest
                                   neighbor) */
         fwidth = .5;
         break;
    case 1: filter = triFilter; /* triangle filter (linear
                                   intrp) */
         fwidth = 1;
         break;
    case 2: filter = cubicConv; /* cubic convolution
                                   filter */
         fwidth = 2;
         break;
    case 3: filter = lanczos3;  /* Lanczos3 windowed sinc
                                   function */
         fwidth = 3;
         break;
    case 4: filter = hann4;     /* Hann windowed sinc
                                   function */
         fwidth = 4;            /* 8-point kernel */
         break;
    }

    if(scale < 1.0) {     /* minification: h(x) -> h(x*scale)*
                             scale */
```

```
        fwidth = fwidth / scale;   /* broaden filter */
        fscale = scale;            /* lower amplitude */
    } else     fscale = 1.0;

  /* project each output pixel to input, center kernel, and
     convolve */
      for(x=0; x<OUTlen; x++) {
         /* map output x to input u: inverse mapping */
         u = x / scale;

         /* left and right extent of kernel centered at u */
         if(u - fwidth < 0) {
                left = FLOOR (u - fwidth);
         else left = CEILING(u - fwidth);
         right = FLOOR(u + fwidth);

         /* reset acc for collecting convolution products */
         acc = 0;

         /* weigh input pixels around u with kernel */
         for(i=left; i <= right; i++) {
                pixel = IN[ CLAMP(i, 0, INlen-1)*offset];
                weight = (*filter)((u - i) * fscale);
                acc += (pixel * weight);
         }

         /* assign weighted accumulator to OUT */
         OUT[x*offset] = acc * fscale;
    }
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * boxFilter:
 *
 * Box (nearest neighbor) filter.
 */
double
boxFilter(t)
double t;
{
      if((t > -.5) && (t <= .5)) return(1.0);
      return(0.0);
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * triFilter:
 *
 * Triangle filter (used for linear interpolation).
 */
double
triFilter(t)
double t;
{
      if(t < 0) t = -t;
      if(t < 1.0) return(1.0 - t);
```

```
	return(0.0);
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * cubicConv:
 *
 * Cubic convolution filter.
 */
double
cubicConv(t)
double t;
{
	double A, t2, t3;

	if(t < 0) t = -t;
	t2 = t * t;
	t3 = t2 * t;

	A = -1.0;	/* user-specified free parameter */
	if(t < 1.0) return((A+2)*t3 - (A+3)*t2 + 1);
	if(t < 2.0) return(A*(t3 - 5*t2 + 8*t - 4));
	return(0.0);
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * sinc:
 *
 * Sinc function.
 */
double
sinc(t)
double t;
{
	t *= PI;
	if(t != 0) return(sin(t) / t);
	return(1.0);
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * lanczos3:
 *
 * Lanczos3 filter.
 */
double
lanczos3(t)
double t;
{
	if(t < 0) t = -t;
	if(t < 3.0) return(sinc(t) * sinc(t/3.0));
	return(0.0);
}

/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * hann:
```

```
 *
 * Hann windowed sinc function. Assume N (width) = 4.
 */
double
hann4(t)
double t;
{
      int N = 4; /* fixed filter width */

      if(t < 0) t = -t;
      if(t < N) return(sinc(t) * (.5 + .5*cos(PI*t/N)));
      return(0.0);
}
```

There are several points worth mentioning about this code. First, the filter width **fwidth** of each of the supported kernels is initialized for use in interpolation (for magnification). We then check to see if the scale factor **scale** is less than one to rescale **fwidth** accordingly. Furthermore, we set **fscale,** the filter amplitude scale factor, to 1 for interpolation, or **scale** for minification. We then visit each of **OUTlen** output pixels, and project them back into the input, where we center the filter kernel. The kernel overlaps a range of input pixels from **left** to **right.** All pixels in this range are multiplied by a corresponding kernel value. The products are added in an accumulator **acc** and assigned to the output buffer.

Note that the **CLAMP** macro is necessary to prevent us from attempting to access a pixel beyond the extent of the input buffer. By clamping to either end, we are effectively replicating the border pixel for use with a filter kernel that extends beyond the image.

In order to accommodate the processing of rows and columns, the variable **offset** is introduced to specify the interpixel distance. When processing rows, **offset = 1.** When processing columns, **offset** is set to the width of a row.

This code can accommodate a polynomial transformation by making a simple change to the evaluation of **u.** Rather than computing **u = x**/scale **,** we may let u be expressed by a polynomial. The method of forward differences is recommended to simplify the computation of polynomials [Wolberg 1990].

The code given above suffers from three limitations, all dealing with efficiency:

1. A division operation is used to compute the inverse projection. Since we are dealing with a linear mapping function, the new position at which to center the kernel may be computed incrementally. That is, there is a constant offset between each projected output sample. Accordingly, **left** and **right** should be computed incrementally as well.
2. The set of kernel weights used in processing the first scanline applies equally to all the remaining scanlines as well. There should be no need to recompute them each time. This matter is addressed in the code supplied by Schumacher [1992].
3. The kernel weights are evaluated by calling the appropriate filter function with the normalized distance from the center. This involves a lot of run-time overhead, particularly for the more sophisticated kernels that require the evaluation of a sinc function, division, and several multiplies.

Additional sophisticated algorithms to deal with these issues are given in Wolberg [1990].

39.9 Research Issues and Summary

The computer graphics literature is replete with new and innovative work addressing the demands of sampling, reconstruction, and antialiasing. Nonuniform sampling has become important in computer graphics because it facilitates variable sampling density and it allows us to trade structured aliasing for noise. Recent work in adaptive sampling and nonuniform reconstruction is discussed in Glassner [1995]. Excellent surveys in nonuniform reconstruction, which is also known as scattered-data interpolation, can be found in Franke and Nielson [1991] and Hoschek and Lasser [1993]. These problems are also of direct

consequence to image compression. The ability to determine a unique minimal set of samples to completely represent a signal within some specified error tolerance remains an active area of research. The solution must be closely coupled with a nonuniform reconstruction method. Although traditional reconstruction methods are well understood within the framework described in this chapter, the analysis of nonuniform sampling and reconstruction remains challenging.

We now summarize the basic principles of sampling theory, reconstruction, and antialiasing that have been presented in this chapter. We have shown that a continuous signal may be reconstructed from its samples if the signal is bandlimited and the sampling frequency exceeds the Nyquist rate. These are the two necessary conditions for image reconstruction to be possible. Since sampling can be shown to replicate a signal's spectrum across the frequency domain, ideal low-pass filtering was introduced as a means of retaining the original spectrum while discarding its copies. Unfortunately, the ideal low-pass filter in the spatial domain is an infinitely wide sinc function. Since this is difficult to work with, nonideal reconstruction filters are introduced to approximate the reconstructed output. These filters are nonideal in the sense that they do not completely attenuate the spectra copies. Furthermore, they contribute to some blurring of the original spectrum. In general, poor reconstruction leads to artifacts such as jagged edges.

Aliasing refers to the phenomenon that occurs when a signal is undersampled. This happens if the reconstruction conditions mentioned above are violated. In order to resolve this problem, one of two actions may be taken. Either the signal can be bandlimited to a range that complies with the sampling frequency, or the sampling frequency can be increased. In practice, some combination of both options is taken, leaving some relatively unobjectionable aliasing in the output.

Examples of the concepts discussed thus are concisely depicted in Figure 39.25 through 39.27. They attempt to illustrate the effects of sampling and low-pass filtering on the quality of the reconstructed signal and its spectrum. The first row of Figure 39.25 shows a signal and its spectrum, bandlimited to 0.5 cycle/pixel. For pedagogical purposes, we treat this signal as if it were continuous. In actuality, though, it is a 256-sample horizontal cross section taken from a digital image. Since each pixel has four samples contributing to it, there is a maximum of two cycles per pixel. The horizontal axes of the spectrum account for this fact.

The second row shows the effect of sampling the signal. Since $f_s = 1$ sample/pixel, there are four copies of the baseband spectrum in the range shown. Each copy is scaled by $f_s = 1$, leaving the magnitudes intact. In the third row, the 64 samples are shown convolved with a sinc function in the spatial domain. This corresponds to a rectangular pulse in the frequency domain. Since the sinc function is used here for image reconstruction, it must have an amplitude of unity value in order to interpolate the data. This forces the height of the rectangular pulse in the frequency domain to vary in response to f_s.

A few comments on the reciprocal relationship between the spatial and frequency domains are in order here, particularly as they apply to the ideal low-pass filter. We again refer to the variables A and W as the sinc amplitude and bandwidth. As a sinc function is made broader, the value $1/2W$ is made to change, since W is decreasing to accommodate zero crossings at larger intervals. Accordingly, broader sinc functions cause more blurring, and their spectra reflect this by reducing the cutoff frequency to some smaller W. Conversely, narrower sinc functions cause less blurring, and W takes on some larger value. In either case, the amplitude of the sinc function or its spectrum will change. That is, we can fix the amplitude of the sinc function so that only the rectangular pulse of the spectrum changes height $A/2W$ as W varies. Alternatively, we can fix $A/2W$ to remain constant as W changes, forcing us to vary A. The choice depends on the application.

When the sinc function is used to interpolate data, it is necessary to fix A to 1. Therefore, as the sampling density changes, the positions of the zero crossings shift, causing W to vary. This makes the amplitude of the spectrum's rectangular pulse change. On the other hand, if the sinc function is applied to bandlimit, not interpolate, the input signal, then it is important to fix $A/2W$ to 1 so that the passband frequencies remain intact. Since W is once again varying, A must change proportionately to keep $A/2W$ constant. Therefore, this application of the ideal low-pass filter requires the amplitude of the sinc function to be responsive to W.

In the examples presented below, our objective is to interpolate (reconstruct) the input, and so $A = 1$ regardless of the sampling density. Consequently, the height of the spectrum of the reconstruction filter changes. To make the Fourier transforms of the filters easier to see, we have not drawn the frequency

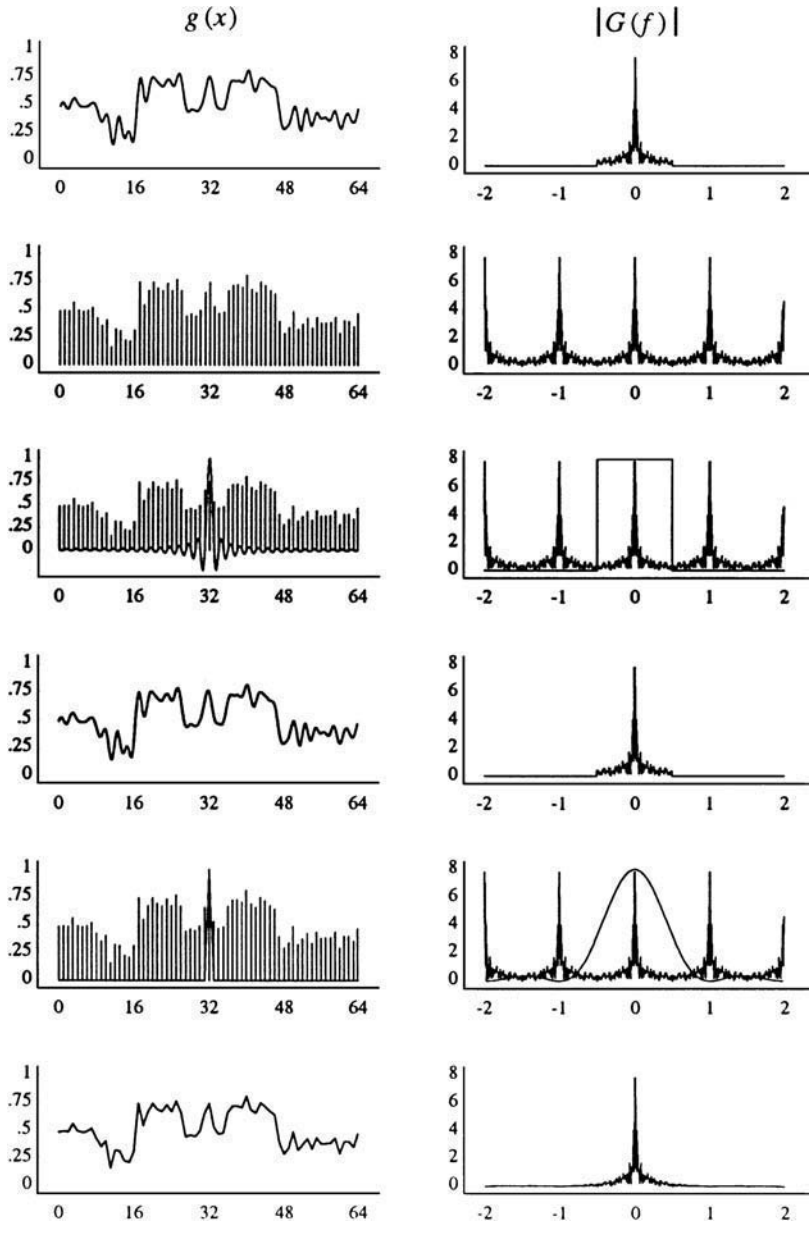


FIGURE 39.25 Sampling and reconstruction (with an adequate sampling rate).

response of the reconstruction filters to scale. Therefore, the rectangular pulse function in the third row of Figure 39.25 actually has height $A/2W = 1$. The fourth row of the figure shows the result after applying the ideal low-pass filter. As sampling theory predicts, the output is identical to the original signal. The last two rows of the figure illustrate the consequences of nonideal reconstruction filtering. Instead of using a sinc function, a triangle function corresponding to linear interpolation was applied. In the frequency domain this corresponds to the square of the sinc function. Not surprisingly, the spectrum of the reconstructed signal suffers in both the passband and the stopband.

The identical sequence of filtering operations is performed in Figure 39.26. In this figure, though, the sampling rate has been lowered to $f_s = 0.5$, meaning that only one sample is collected for every

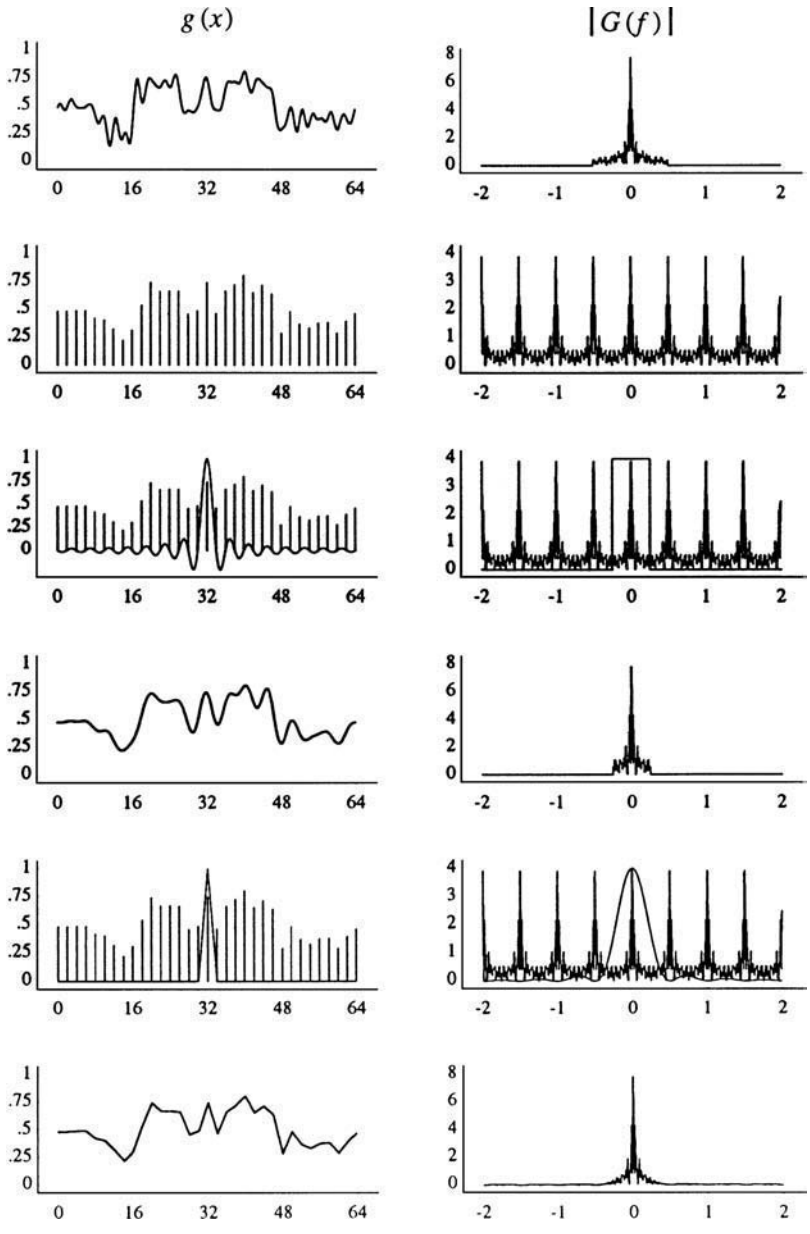


FIGURE 39.26 Sampling and reconstruction (with an inadequate sampling rate).

two output pixels. Consequently, the replicated spectra are multiplied by 0.5, leaving the magnitudes at 4. Unfortunately, this sampling rate causes the replicated spectra to overlap. This, in turn, gives rise to aliasing, as depicted in the fourth row of the figure. Applying the triangle function to perform linear interpolation also yields poor results.

In order to combat these artifacts, the input signal must be bandlimited to accommodate the low sampling rate. This is shown in the second row of Figure 39.27, where we see that all frequencies beyond $W = 0.25$ are truncated. This causes the input signal to be blurred. In this manner we have traded aliasing for blurring, a far less objectionable artifact. Sampling this function no longer causes the replicated copies to overlap. Convolving with an ideal low-pass filter now properly isolates the bandlimited spectrum.

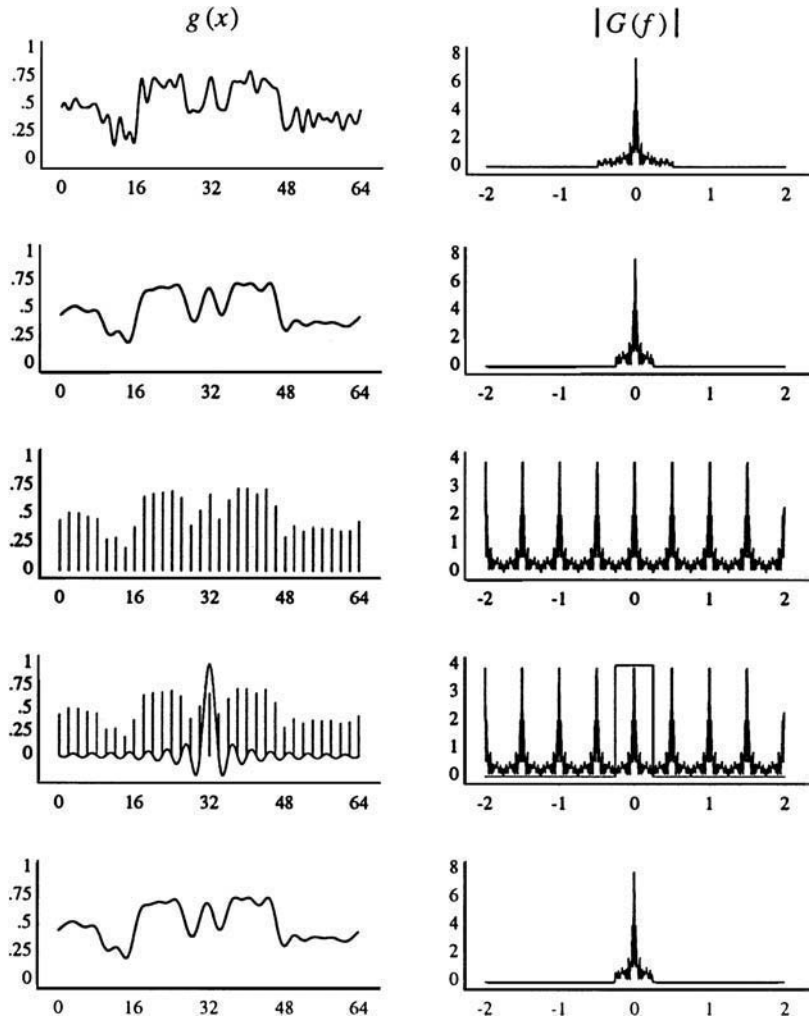


FIGURE 39.27 Antialiasing filtering, sampling, and reconstruction stages.

Defining Terms

Adaptive supersampling: Supersampling with samples distributed more densely in areas of high intensity variance.

Aliasing: Artifacts due to undersampling a signal. This condition prevents the signal from being reconstructed from its samples.

Antialiasing: The filtering necessary to combat aliasing. This generally requires bandlimiting the input before sampling to remove the offending high frequencies that will fold over in the frequency spectrum.

Area sampling: An antialiasing method that treats a pixel as an area, not a point. After projecting the pixel to the input, all samples in the preimage are averaged to compute a representative sample.

Bandlimiting: The act of truncating all frequency components beyond some specified frequency. Useful for antialiasing, where offending high frequencies must be removed to prevent aliasing.

Frequency leakage: A condition in which the stopband is allowed to persist, permitting it to fold over into the passband range.