1 Overview of Visualization

WILLIAM J. SCHROEDER and KENNETH M. MARTIN
Kitware, Inc.

1.1 Introduction

In this chapter, we look at basic algorithms for scientific visualization. In practice, a typical algorithm can be thought of as a transformation from one data form into another. These operations may also change the dimensionality of the data. For example, generating a streamline from a specification of a starting point in an input 3D dataset produces a 1D curve. The input may be represented as a finite element mesh, while the output may be represented as a polyline. Such operations are typical of scientific visualization systems that repeatedly transform data into different forms and ultimately transform it into a representation that can be rendered by the computer system.

The algorithms that transform data are the heart of data visualization. To describe the various transformations available, we need to categorize algorithms according to the *structure* and *type* of transformation. By *structure*, we mean the effects that transformation has on the topology and geometry of the dataset. By *type*, we mean the type of dataset that the algorithm operates on.

Structural transformations can be classified in four ways, depending on how they affect the geometry, topology, and attributes of a dataset. Here, we consider the topology of the dataset as the relationship of discrete data samples (one to another) that are invariant with respect to geometric transformation. For example, a regular, axis-aligned sampling of data in three dimensions is referred to as a *volume*, and its topology is a rectangular (structured) lattice with clearly defined neighborhood voxels and samples. On the other hand, the topology of a finite element mesh is represented by an (unstructured) list of elements, each defined by an ordered list of points. Geometry is a specification of the topology in space (typically 3D), including point coordinates and interpolation functions. Attributes are data associated with the topology and/or geometry of the dataset, such as temperature, pressure, or velocity. Attributes are typically categorized as being scalars (single value per sample), vectors (n-vector of values), tensor (matrix), surface normals, texture coordinates, or general field data. Given these terms, the following transformations are typical of scientific visualization systems:

- *Geometric transformations* alter input geometry but do not change the topology of the dataset. For example, if we translate, rotate, and/or scale the points of a polygonal dataset, the topology does not change, but the point coordinates, and therefore the geometry, do.
- *Topological transformations* alter input topology but do not change geometry and attribute data. Converting a dataset type from polygonal to unstructured grid, or from image to unstructured grid, changes the topology but not the geometry. More often, however, the geometry changes whenever the topology does, so topological transformation is uncommon.
- *Attribute transformations* convert data attributes from one form to another, or create new attributes from the input data. The structure of the dataset remains unaffected.

Text and images taken with permission from the book *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 3rd ed., published by Kitware, Inc. http://www.kitware.com/products/vtktextbook.html.

Computing vector magnitude and creating scalars based on elevation are data attribute transformations.

- *Combined transformations* change both dataset structure and attribute data. For example, computing contour lines or surfaces is a combined transformation.

We also may classify algorithms according to the *type* of data they operate on. The meaning of the word "type" is often somewhat vague. Typically, "type" means the type of attribute data, such as scalars or vectors. These categories include the following:

- *Scalar algorithms* operate on scalar data. An example is the generation of contour lines of temperature on a weather map.
- *Vector algorithms* operate on vector data. Showing oriented arrows of airflow (direction and magnitude) is an example of vector visualization.
- *Tensor algorithms* operate on tensor matrices. One example of a tensor algorithm is to show the components of stress or strain in a material using oriented icons.
- *Modeling algorithms* generate dataset topology or geometry, or surface normals or texture data. "Modeling algorithms" tends to be the catch-all category for algorithms that do not fit neatly into any single category mentioned above. For example, generating glyphs oriented according to the vector direction and then scaled according to the scalar value is a combined scalar/vector algorithm. For convenience, we classify such an algorithm as a modeling algorithm because it does not fit squarely into any other category.

Note that an alternative classification scheme is to refer to the topological type of the input data (e.g., image, volume, or unstructured mesh) that a particular algorithm operates on. In the remainder of the chapter we will classify the type of the algorithm as the type of attribute data on which it operates. Be forewarned, though, that alternative classification schemes do exist and may be better suited to describing the true nature of the algorithm.

1.1.1 Generality Vs. Efficiency

Most algorithms can be implemented specifically for a particular data type or, more generally, for treating any data type. The advantage of a specific algorithm is that it is usually faster than a comparable general algorithm. An implementation of a specific algorithm may also be more memory-efficient, and it may better reflect the relationship between the algorithm and the dataset type it operates on.

One example of this is contour surface creation. Algorithms for extracting contour surfaces were originally developed for volume data, mainly for medical applications. The regularity of volumes lends itself to efficient algorithms. However, the specialization of volume-based algorithms precludes their use for more general datasets such as structured or unstructured grids. Although the contour algorithms can be adapted to these other dataset types, they are less efficient than those for volume datasets.

The presentation of algorithms in this chapter favors more general implementations. In some special cases, authors will describe performance-improving techniques for particular dataset types. Various other chapters in this book also include detailed descriptions of specialized algorithms.

1.1.2 Algorithms as Filters

In a typical visualization system, algorithms are implemented as filters that operate on data. This approach is due in some part to the success of early systems like the Application Visualization System [2] and Data Explorer [9] and the popularity of systems like SCIRun [37] and the Visualization Toolkit [36] that are built around the abstraction of data flow. This abstraction is natural because of the transformative nature of visualization. The basic idea is that two types of objects—data objects and process objects—are connected together into visualization pipelines.

The process objects, or filters, are the algorithms that operate on the data objects and in turn produce data objects as output. In this abstraction, filters that initiate the pipeline are referred to as *sources*; filters that terminate the pipeline are known as *sinks* (or *mappers*). Depending on their particular implementation, filters may have multiple inputs and/or may produce multiple outputs.

1.2 Scalar Algorithms

Scalars are single data values associated with each point and/or cell of a dataset. Because scalar data is commonly found in real-world applications, and because it is so easy to work with, there are many different algorithms to visualize it.

1.2.1 Color Mapping

Color mapping is a common scalar visualization technique that maps scalar data to colors and displays the colors using the standard coloring and shading facilities of the graphics library. The scalar mapping is implemented by indexing into a *color lookup table*. Scalar values serve as indices into the lookup table.

The mapping proceeds as follows. The lookup table holds an array of colors (e.g., red, green, blue, and alpha transparency components or other comparable representations). Associated with the table is a minimum and maximum *scalar range (min, max)* into which the scalar values are mapped. Scalar values greater than the maximum range are clamped to the maximum color, and scalar values less than the minimum range are clamped to the minimum color value. For each scalar value s_i, the index i into the color table with n entries (and 0-offset) is given by Fig. 1.1.

A more general form of the lookup table is called a *transfer function*. A transfer function is any expression that maps scalar value into a color specification. For example, Fig. 1.2 maps scalar values into separate intensity values for the red, green, and blue color components. We can also use transfer functions to map scalar data into other information, such as local transparency. A lookup table is a discrete sampling of a transfer function. We can create a lookup table from any transfer function by sampling the transfer function at a set of discrete points.

Color mapping is a 1D visualization technique. It maps one piece of information (i.e., a scalar value) into a color specification. However, the display of color information is not limited to 1D displays. Often the colors are mapped onto 2D or 3D objects. This is a simple way to increase the information content of the visualizations.

The key to color mapping for scalar visualization is to choose the lookup table entries carefully. Fig. 1.3 shows four different lookup tables used to visualize gas density as fluid flows through a combustion chamber. The first lookup table is grey-scale. Grey-scale tables often provide better structural detail to the eye.

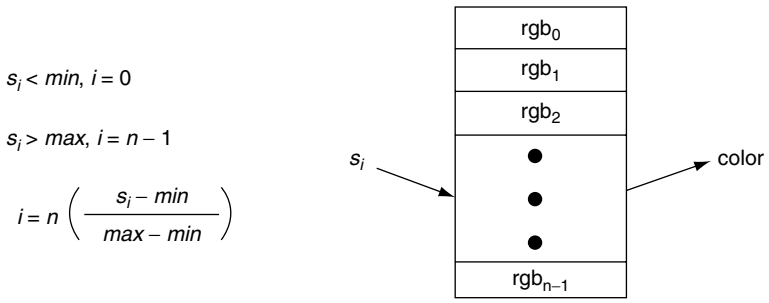


Figure 1.1 Mapping scalars to colors via a lookup table.

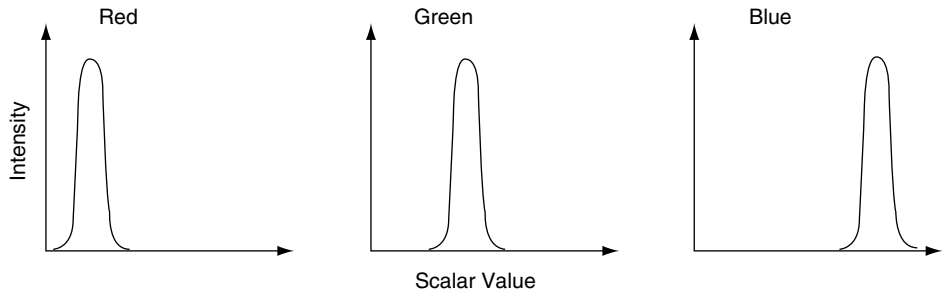


Figure 1.2 Transfer function for color components red, green, and blue as a function of scalar value.

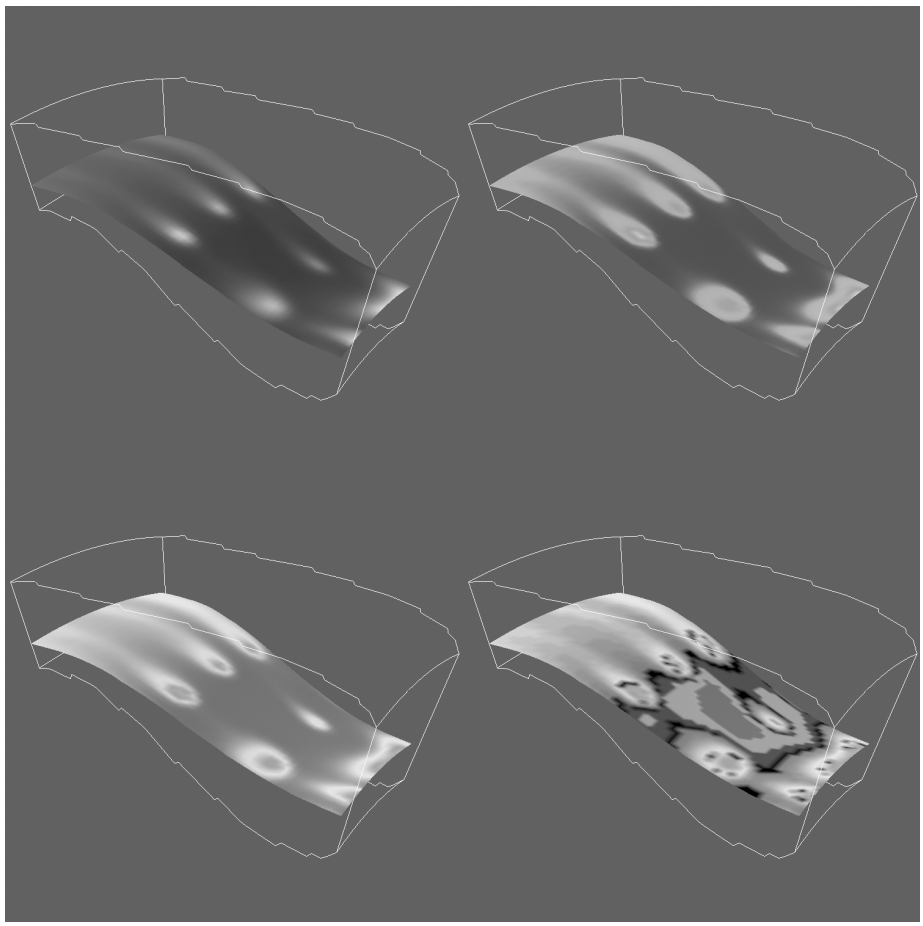


Figure 1.3 Flow density colored with different lookup tables. (Top left) Grey-scale; (top right) rainbow (blue to red); (lower left) rainbow (red to blue); (lower right) large contrast. (See also color insert.)

The other three images in Fig. 1.3 use different color lookup tables. The second uses rainbow hues from blue to red. The third uses rainbow hues arranged from red to blue. The last image uses a table designed to enhance contrast. Careful use of colors can often enhance important features of a dataset. However, any type of lookup table can exaggerate unimportant details or create visual artifacts because of unforeseen interactions among data, color choice, and human physiology.

Designing lookup tables is as much an art as it is a science. From a practical point of view, tables should accentuate important features while minimizing less important or extraneous details. It is also desirable to use palettes that inherently contain scaling information. For example, a color rainbow scale from blue to red is often used to represent temperature scale, since many people associate blue with cold temperatures and red with hot temperatures. However, even this scale is problematic: a physicist would say that blue is hotter than red, since hotter objects emit more blue (i.e., shorter-wavelength) light than red. Also, there is no need to limit ourselves to "linear" lookup tables. Even though the mapping of scalars into colors has been presented as a linear operation (Fig. 1.1), the table itself need not be linear; that is, tables can be designed to enhance small variations in scalar value using logarithmic or other schemes.

1.2.2 Contouring

One natural extension to color mapping is *contouring*. When we see a surface colored with data values, the eye often separates similarly colored areas into distinct regions. When we contour data, we are effectively constructing the boundary between these regions. A particular boundary can be expressed as the n-dimensional separating surfaces

$$F(\overline{x}) = c \tag{1.1}$$

between the two regions $F(\overline{x}) < c$ and $F(\overline{x}) > c$, where c is the *contour value* and $\overline{x}$ is an n-dimensional point in the dataset. These two regions are typically referred to as the *inside* or *outside* regions of the contour.

Examples of 2D contour displays include weather maps annotated with lines of constant temperature (isotherms) or topological maps drawn with lines of constant elevation. 3D contours are called *isosurfaces* and can be approximated by many polygonal primitives. Examples of isosurfaces include constant medical image intensity corresponding to body tissues such as skin, bone, or other organs. Other abstract isosurfaces, such as surfaces of constant pressure or temperature in fluid flow, may also be created.

Consider the 2D structured grid shown in Fig. 1.4. Scalar values are shown next to the points that define the grid. Contouring always begins when one specifies a contour value defining the contour line or surface to be generated. To generate the contours, some form of interpolation must be used. This is because we have scalar values at a discrete set of (sample) points in the dataset, and our contour value may lie between the point values. Since the most common interpolation technique is linear, we generate points on the contour surface by linear interpolation along the edges. If an edge has scalar values 10 and 0 at its two endpoints, for example, and if we are trying to generate a contour line of value 5, then edge interpolation computes that

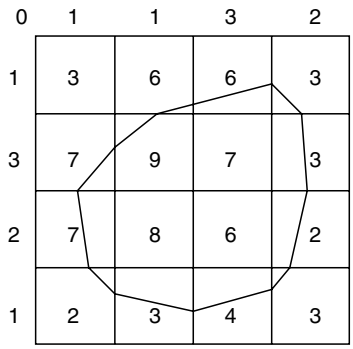


Figure 1.4 Contouring a 2D structured grid with contour line value = 5.

the contour passes through the midpoint of the edge.

Once the points on cell edges are generated, we can connect these points into contours using a few different approaches. One approach detects an edge intersection (i.e., the passing of a contour through an edge) and then "tracks" this contour as it moves across cell boundaries. We know that if a contour edge enters a cell, it must exit a cell as well. The contour is tracked until it closes back on itself or exits a dataset boundary. If it is known that only a single contour exists, then the process stops. Otherwise, every edge in the dataset must be checked to see whether other contour lines exist.

Another approach uses a divide-and-conquer technique, treating cells independently. This is called the *marching squares* algorithm in 2D and the *marching cubes* algorithm [23] in 3D. The basic assumption of these techniques is that a contour can pass through a cell in only a finite number of ways. A case table is constructed that enumerates all possible topological *states* of a cell, given combinations of scalar values at the cell points. The number of topological states depends on the number of cell vertices and the number of inside/outside relationships a vertex can have with respect to the contour value. A vertex is considered inside a contour if its scalar value is larger than the scalar value of the contour line. Vertices with scalar values less than the contour value are said to be outside the contour. For example, if a cell has four vertices and each vertex can be either inside or outside the contour, there are $2^4 = 16$ possible ways that the contour passes through the cell. In the case table, we are not interested in where the contour passes through the cell (e.g., geometric intersection), just how it passes through the cell (i.e., topology of the contour in the cell).

Fig. 1.5 shows the 16 combinations for a square cell. An index into the case table can be computed by encoding the state of each vertex as a binary digit. For 2D data represented on a rectangular grid, we can represent the 16 cases with a 4-bit index. Once the proper case is selected, the location of the contour line/cell edge intersection can be calculated using interpolation. The algorithm processes a cell and then moves, or *marches*, to the next cell. After all the cells are visited, the contour will be completed. In summary, the marching algorithms proceed as follows:

1. Select a cell.
2. Calculate the inside/outside state of each vertex of the cell.
3. Create an index by storing the binary state of each vertex in a separate bit.
4. Use the index to look up the topological state of the cell in a case table.
5. Calculate the contour location (via interpolation) for each edge in the case table.

This procedure will construct independent geometric primitives in each cell. At the cell

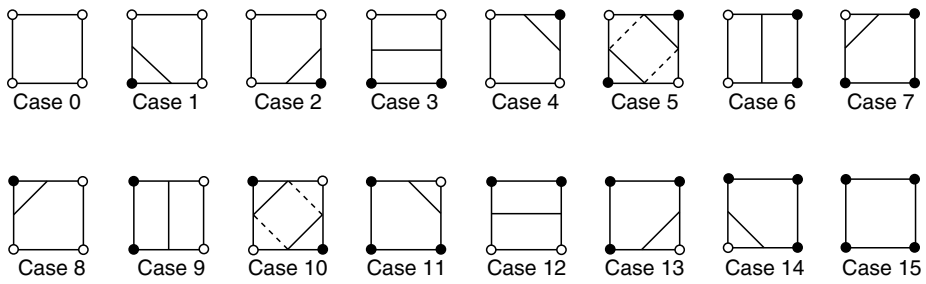


Figure 1.5 Sixteen different marching squares cases. Dark vertices indicate scalar value is above contour value. Cases 5 and 10 are ambiguous.

boundaries, duplicate vertices and edges may be created. These duplicates can be eliminated by use of a special coincident point-merging operation. Note that interpolation along each edge should be done in the same direction. If it is not, numerical round-off will likely cause points to be generated that are not precisely coincident and will thus not merge properly.

There are advantages and disadvantages to both the edge-tracking and the marching cubes approaches. The marching squares algorithm is easy to implement. This is particularly important when we extend the technique into three dimensions, where isosurface tracking becomes much more difficult. On the other hand, the algorithm creates disconnected line segments and points, and the required merging operation requires extra computation resources. The tracking algorithm can be implemented to generate a single polyline per contour line, avoiding the need to merge coincident points.

As mentioned previously, the 3D analogy of marching squares is marching cubes. Here, there are 256 different combinations of scalar value, given that there are eight points in a cubical cell (i.e., 2^8 combinations). Figure 1.6 shows these combinations reduced to 15 cases by arguments of symmetry. We use combinations of rotation and mirroring to produce topologically equivalent cases. (This is the so-called marching cubes case table.)

An important issue is *contouring ambiguity*. Careful observation of marching squares cases 5 and 10 and marching cubes cases 3, 6, 7, 10, 12, and 13 show that there are configurations where a cell can be contoured in more than one way. (This ambiguity also exists in an edge-tracking approach to contouring.) Contouring ambiguity arises on a 2D square or the face of a 3D cube when adjacent edge points are in different states but diagonal vertices are in the same state.

In two dimensions, contour ambiguity is simple to treat: for each ambiguous case, we implement one of the two possible cases. The choice for a particular case is independent of all other choices. Depending on the choice, the contour may either extend or break the current contour, as illustrated in Fig. 1.8. Either choice is acceptable since the resulting contour lines will be continuous and closed (or will end at the dataset boundary).

In three dimensions the problem is more complex. We cannot simply choose an ambiguous case independent of all other ambiguous cases. For example, Fig. 1.9 shows what happens if we carelessly implement two cases independent of one another. In this figure we have used the usual case 3 but replaced case 6 with its *complementary* case. Complementary cases are formed by exchanging the "dark" vertices with "light" vertices. (This is equivalent to swapping vertex scalar value from above the isosurface value to below the isosurface value, and vice versa.) The result of pairing these two cases is that a hole is left in the isosurface.

Several different approaches have been taken to remedy this problem. One approach tessellates the cubes with tetrahedra and uses a *marching tetrahedra* technique. This works because the marching tetrahedra exhibit no ambiguous cases. Unfortunately, the marching tetrahedra algorithm generates isosurfaces consisting of more triangles, and the tessellation of a cube with tetrahedra requires one to make a choice regarding the orientation of the tetrahedra. This choice may result in artificial "bumps" in the isosurface because of interpolation along the face diagonals, as shown in Fig. 1.7. Another approach evaluates the asymptotic behavior of the surface and then chooses the cases to either join or break the contour. Nielson and Hamann [28] have developed a technique based on this approach that they call the *asymptotic decider*. It is based on an analysis of the variation of the scalar variable across an ambiguous face. The analysis determines how the edges of isosurface polygons should be connected.

A simple and effective solution extends the original 15 marching cubes cases by adding additional complementary cases. These cases are designed to be compatible with neighboring cases and prevent the creation of holes in the

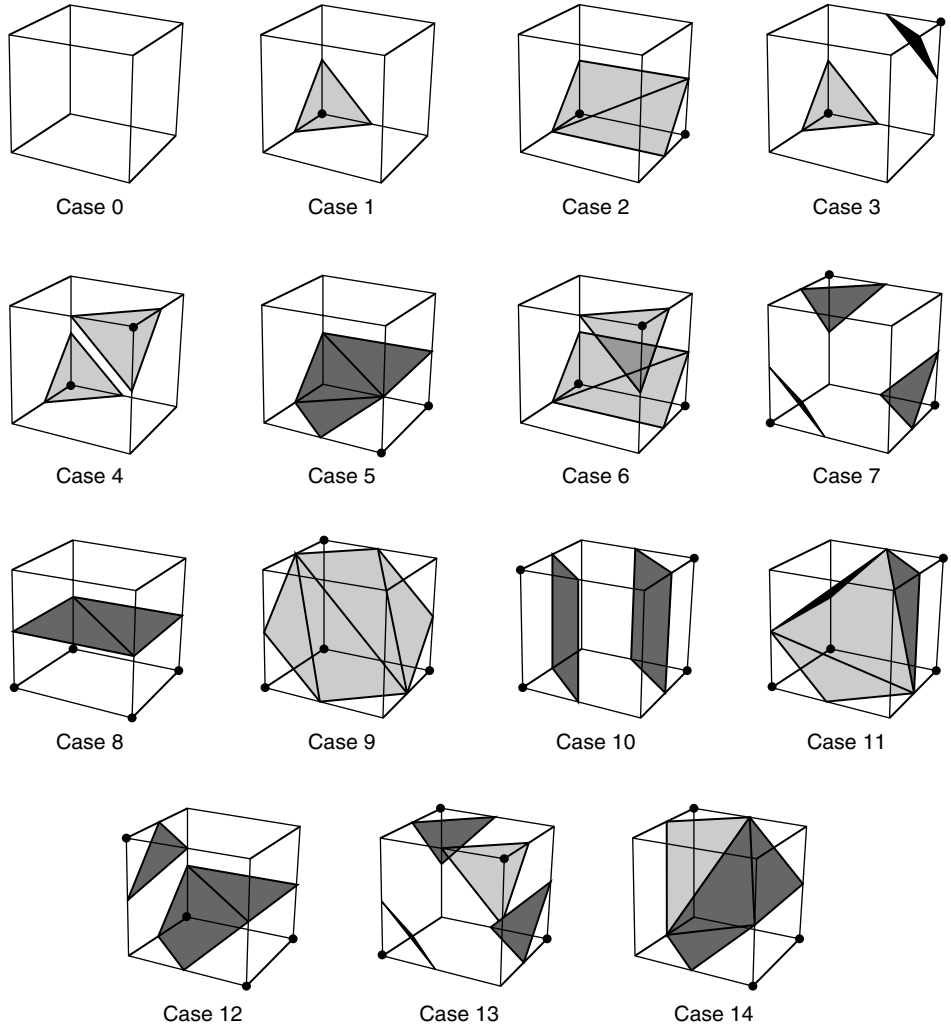


Figure 1.6 Marching cubes cases for 3D isosurface generation. The 256 possible cases have been reduced to 15 cases using symmetry. Vertices with a dot are greater than the selected isosurface value.

isosurface. There are six complementary cases required, corresponding to the marching cubes cases 3, 6, 7, 10, 12, and 13. The complementary marching cubes cases are shown in Fig. 1.10. In practice the simplest approach is to create a case table consisting of all 256 possible combinations and to design them in such a way as to prevent holes. A successful marching cubes case table will always produce manifold surfaces (i.e., interior edges are used by exactly two triangles; boundary edges are used by exactly one triangle).

We can extend the general approach of marching squares and marching cubes to other topological types such as triangles, tetrahedra, pyramids, and wedges. In addition, although we refer to regular types such as squares and cubes, marching cubes can be applied to any cell type

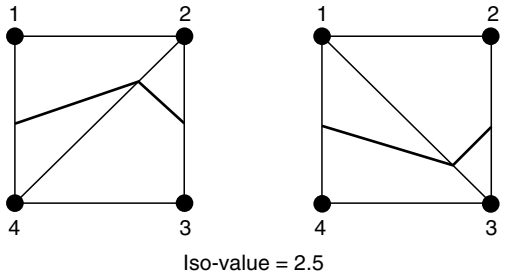


Figure 1.7 Using marching triangles or marching tetrahedra to resolve ambiguous cases on rectangular lattice (only the face of the cube is shown). Choice of diagonal orientation can result in ''bumps'' in the contour surface. In two dimensions, diagonal orientation can be chosen arbitrarily, but in three dimensions the diagonal is constrained by the neighbor.

topologically equivalent to a cube (e.g., a hexahedron or noncubical voxel).

Fig. 1.11 shows four applications of contouring. In Fig. 1.11a we see 2D contour lines of CT density value corresponding to different tissue types. These lines were generated using marching squares. Figs 1.11b through 1.11d are isosurfaces created by marching cubes. Fig. 1.11b is a surface of constant image intensity from a computed tomography (CT) x-ray imaging system. (Fig. 1.11a is a 2D subset of these data.) The intensity level corresponds to human bone. Fig. 1.11c is an isosurface of constant flow density. Figure 1.11d is an isosurface of electron potential of an iron protein molecule. The image shown in Fig. 1.11b is immediately recognizable because of our familiarity with human anatomy. However, for those practitioners in the fields of computational fluid dynamics (CFD) and molecular biology, Figs. 1.11c and 1.11d are equally familiar. As these examples show, methods for contouring are powerful, yet general, techniques for visualizing data from a variety of fields.

1.2.3 Scalar Generation

The two visualization techniques presented thus far, color mapping and contouring, are simple, effective methods to display scalar information. It is natural to turn to these techniques first when visualizing data. However, often our data are not in a form convenient to these techniques. The data may not be single-valued (i.e., a scalar), or they may be a mathematical or other complex relationship. That is part of the fun and creative challenge of visualization: we must tap our creative resources to convert data into a form on which we can bring our existing tools to bear.

For example, consider terrain data. We assume that the data are x-y-z coordinates, where x and y represent the coordinates in the plane and z represents the elevation above sea level. Our desired visualization is to color the terrain according to elevation. This requires us to create a color map—possibly using white for high altitudes, blue for sea level and below, and various shades of green and brown for different elevations between sea level and high altitude. We also need scalars to index into the color

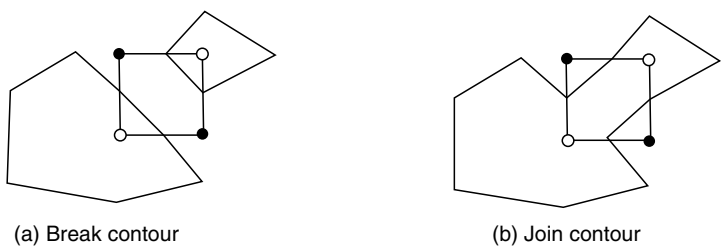


Figure 1.8 Choosing a particular contour case will (a) break or (b) join the current contour. The case shown is marching squares case 10.

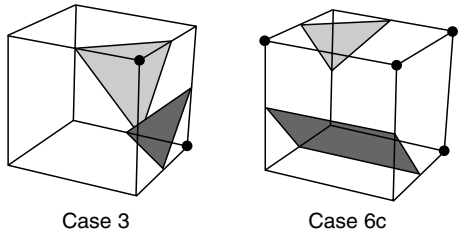


Figure 1.9 Arbitrarily choosing marching cubes cases leads to holes in the isosurface.

map. The obvious choice here is to extract the z coordinate. That is, scalars are simply the z-coordinate value.

This example can be made more interesting by generalizing the problem. Although we could easily create a filter to extract the z coordinate, we can create a filter that produces elevation scalar values where the elevation is measured along any axis. Given an oriented line starting at the (low) point p_l (e.g., sea level) and ending at the (high) point p_h (e.g., mountain top), we compute the elevation scalar s_i at point $p_i = (x_i, y_i, z_i)$ using the dot product as shown in Fig. 1.12. The scalar is normalized using the magnitude of the oriented line and may be clamped between minimum and maximum scalar values (if necessary). The bottom half of this figure shows the results of applying this technique to a terrain model of Honolulu, Hawaii. A lookup table of 256 points ranging from deep blue (water) to yellow-white (mountain top) is used to color map this figure.

Scalar visualization techniques are deceptively powerful. Color mapping and isocontour generation are the predominant methods used in scientific visualization. Scalar visualization techniques are easily adapted to a variety of situations through creation of a relationship that transforms data at a point into a scalar value. Other examples of scalar mapping include an index value into a list of data, computing vector magnitude or matrix determinant, evaluating surface curvature, or determining distance between points. Scalar generation, when coupled with color mapping or contouring, is a simple yet effective technique for visualizing many types of data.

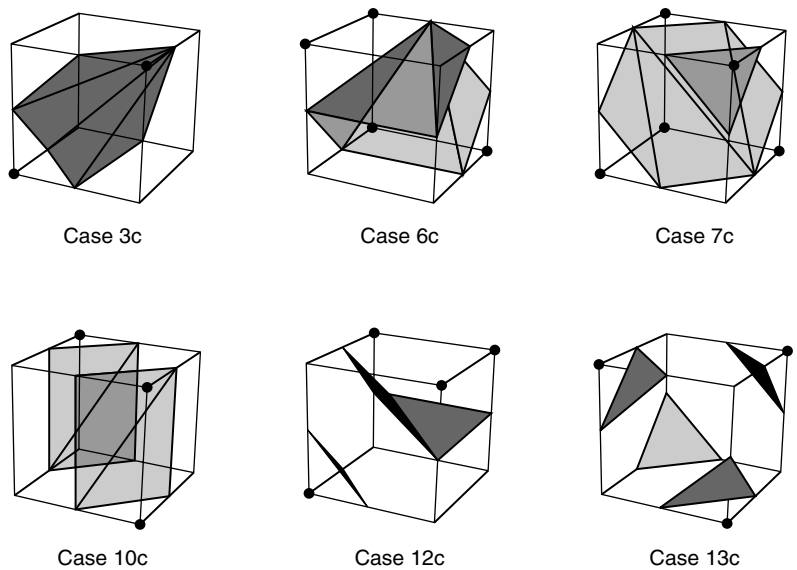


Figure 1.10 Marching cubes complementary cases.

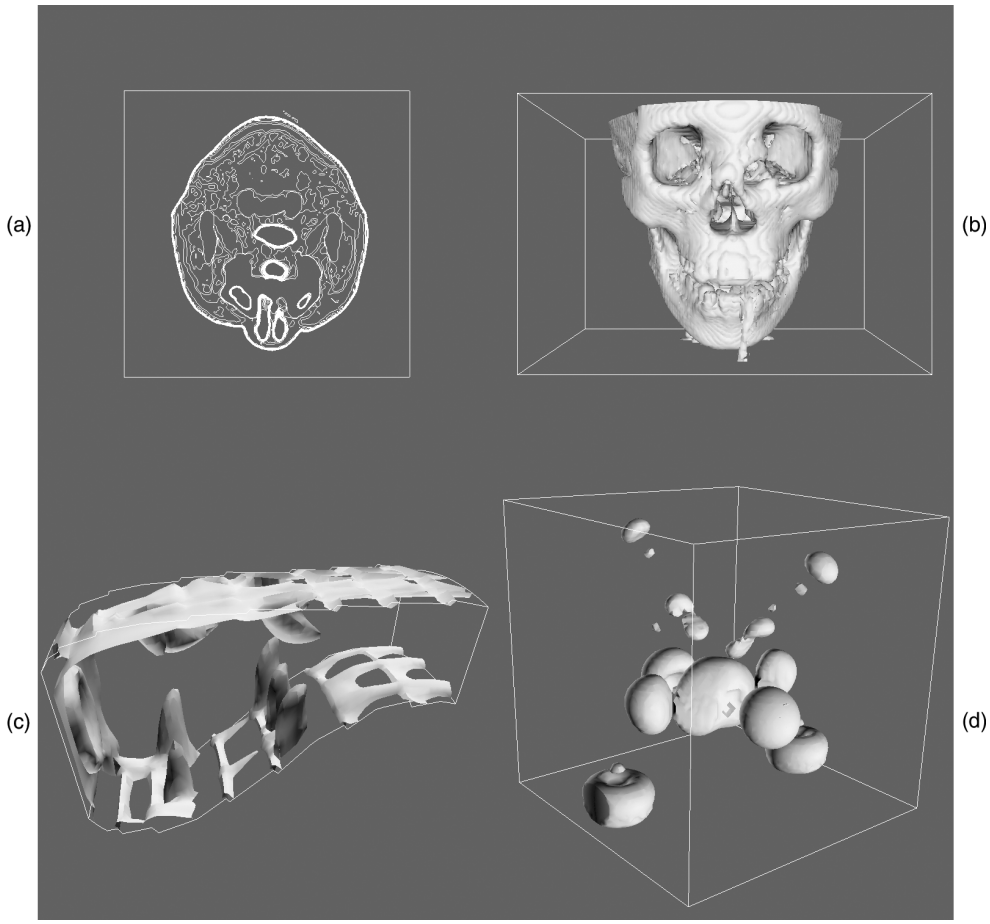


Figure 1.11 Contouring examples. (a) Marching squares used to generate contour lines; (b) marching cubes surface of human bone; (c) marching cubes surface of flow density; (d) marching cubes surface of iron–protein.

1.3 Vector Algorithms

Vector data is a 3D representation of direction and magnitude. Vector data often results from the study of fluid flow or data derivatives.

1.3.1 Hedgehogs and Oriented Glyphs

A natural vector visualization technique is to draw an oriented, scaled line for each vector in a dataset (Fig. 1.13a). The line begins at the point with which the vector is associated and is oriented in the direction of the vector components (v_x, v_y, v_z). Typically, the resulting line must be scaled up or down to control the size of its visual representation. This technique is often referred to as a *hedgehog* because of the bristly result.

There are many variations of this technique (Fig. 1.13b). Arrows may be added to indicate the direction of the line. The lines may be colored according to vector magnitude or some other scalar quantity (e.g., pressure or temperature). Also, instead of using a line, oriented

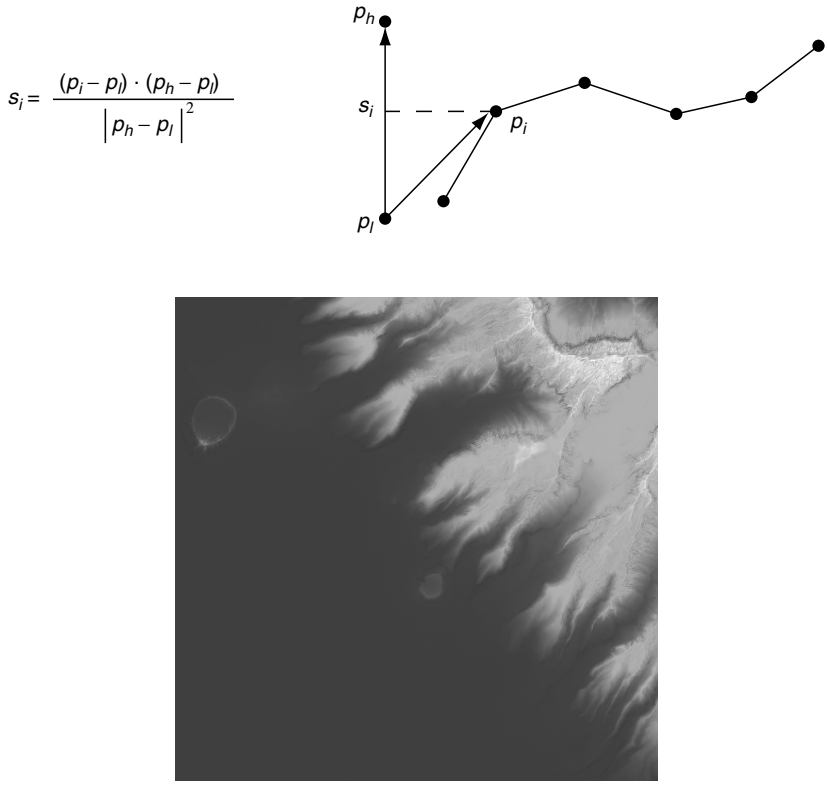


Figure 1.12 Computing scalars using normalized dot product. The bottom half of the figure illustrates a technique applied to terrain data from Honolulu, HI. (See also color insert.)

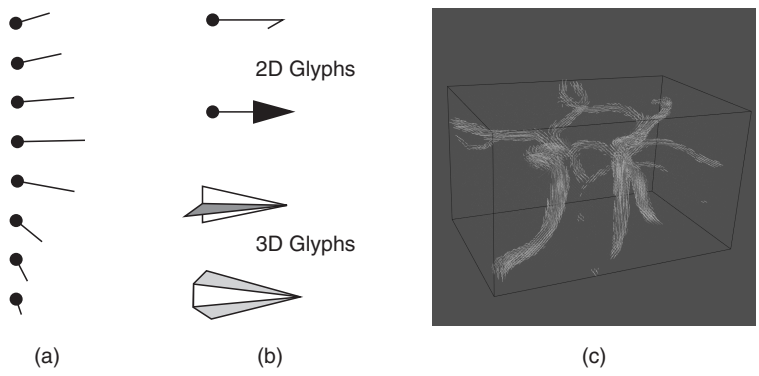


Figure 1.13 Vector visualization techniques. (a) Oriented lines; (b) oriented glyphs; (c) complex vector visualization. (See also color insert.)

"glyphs" can be used. By glyph we mean any 2D or 3D geometric representation, such as an oriented triangle or cone.

Care should be used in applying these techniques. In three dimensions it is often difficult to understand the position and orientation of a vector because of its projection into the 2D view plane. Also, using large numbers of vectors can clutter the display to the point where the visualization becomes meaningless. Figure 1.13c shows 167,000 3D vectors (using oriented and scaled lines) in the region of the human carotid artery. The larger vectors lie inside the arteries, and the smaller vectors lie outside the arteries and are randomly oriented (measurement error) but small in magnitude. Clearly, the details of the vector field are not discernible from this image.

Scaling glyphs also poses interesting problems. In what Tufte [39] has termed a "visualization lie," scaling a 2D or 3D glyph results in nonlinear differences in appearance. The surface area of an object increases with the square of its scale factor, so two vectors differing by a factor of two in magnitude may appear up to four times different based on surface area. Such scaling issues are common in data visualization, and great care must be taken to avoid misleading viewers.

1.3.2 Warping

Vector data is often associated with "motion." The motion is in the form of velocity or displacement. An effective technique for displaying such vector data is to "warp" or deform geometry according to the vector field. For example, imagine representing the displacement of a structure under load by deforming the structure. If we are visualizing the flow of fluid, we can create a flow profile by distorting a straight line inserted perpendicular to the flow.

Figure 1.14 shows two examples of vector warping. In the first example the motion of a vibrating beam is shown. The original undeformed outline is shown in wireframe. The second example shows warped planes in a structured grid dataset. The planes are warped according to flow momentum. The relative back and forward flows are clearly visible in the deformation of the planes.

Typically, we must scale the vector field to control geometric distortion. Too small a distortion might not be visible, while too large a distortion can cause the structure to turn inside out or self-intersect. In such a case, the viewer of the visualization is likely to lose context, and the visualization will become ineffective.

1.3.3 Displacement Plots

Vector displacement on the surface of an object can be visualized with displacement plots. A displacement plot shows the motion of an object in the direction perpendicular to its surface. The object motion is caused by an applied vector field. In a typical application the vector field is a displacement or strain field.

Vector displacement plots draw on the ideas in Section 1.2.3. Vectors are converted to scalars by computation of the dot product between the surface normal and vector at each point (Fig. 1.15a). If positive values result, the motion at the point is in the direction of the surface normal (i.e., positive displacement). Negative values indicate that the motion is opposite the surface normal (i.e., negative displacement).

A useful application of this technique is the study of vibration. In vibration analysis, we are interested in the eigenvalues (i.e., natural resonant frequencies) and eigenvectors (i.e., mode shapes) of a structure. To understand mode shapes, we can use displacement plots to indicate regions of motion. There are special regions in the structure where positive displacement changes to negative displacement. These are regions of zero displacement. When plotted on the surface of the structure, these regions appear as the so-called *modal* lines of vibration. The study of modal lines has long been an important visualization tool for understanding mode shapes.

Figure 1.15b shows modal lines for a vibrating rectangular beam. The vibration mode in this figure is the second torsional mode, clearly

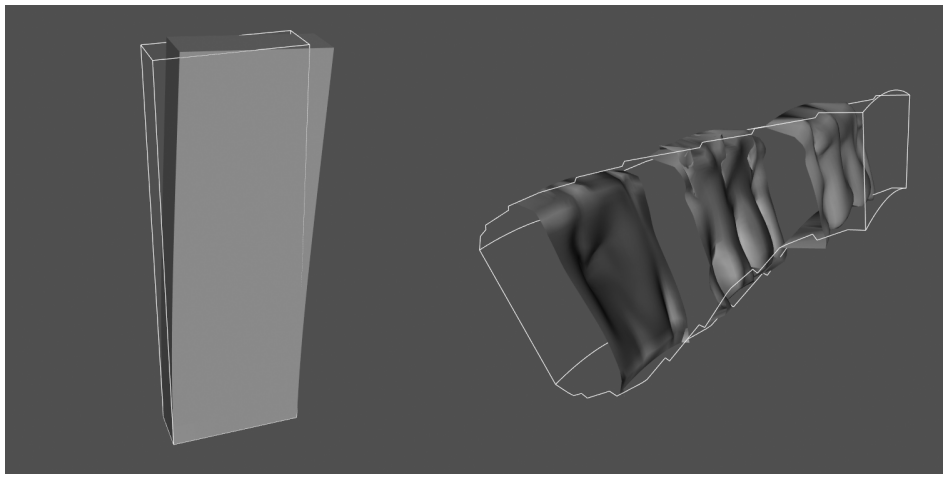


Figure 1.14 Warping geometry to show vector field. (a) Beam displacement; (b) flow momentum. (See also color insert.)

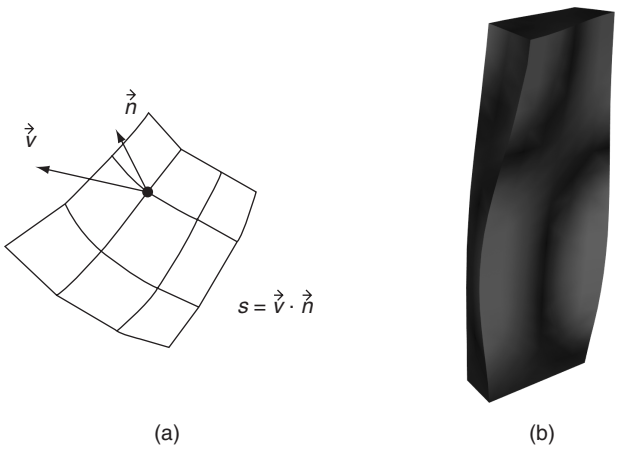


Figure 1.15 Vector displacement plots. (a) Vector converted to scalar via dot product computation; (b) surface plot of vibrating plate. Dark areas show nodal lines and bright areas show maximum motion. (See also color insert.)

indicated by the crossing modal lines. (The aliasing in the figure is a result of the coarseness of the analysis mesh.) To create the figure we combined the procedure of Fig. 1.15a with a special lookup table. The lookup table was arranged with dark areas in the center (corresponding to zero dot products) and bright areas at the beginning and end of the table (corresponding to 1 or -1 dot products). As a result, regions of large normal displacement are bright and regions near the modal lines are dark.

1.3.4 Time Animation

Some of the techniques described so far can be thought of as moving a point or object over a small time-step. The hedgehog line is an approximation of a point's motion over a time

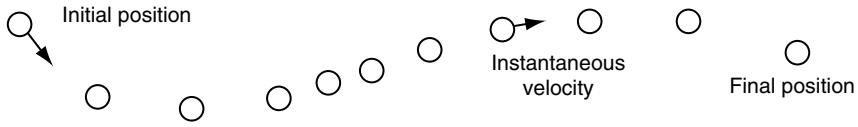


Figure 1.16 Time animation of a point *C*. Although the spacing between points varies, the time increment between each point is constant.

period whose duration is given by the scale factor. In other words, if the vector is considered to be a velocity $\vec{V} = dx/dt$, then the displacement of a point is

$$dx = \vec{V}dt \tag{1.2}$$

This suggests an extension to our previous techniques: repeatedly displace points over many time-steps. Fig. 1.16 shows such an approach. Beginning with a sphere *S* centered about some point *C*, we move *S* repeatedly to generate the bubbles shown. The eye tends to trace out a path by connecting the bubbles, giving the observer a qualitative understanding of the vector field in that area. The bubbles may be displayed as an animation over time (giving the illusion of motion) or as a multiple-exposure sequence (giving the appearance of a path).

Such an approach can be misused. For one thing, the velocity at a point is instantaneous. Once we move away from the point, the velocity is likely to change. Using Equation 1.2 assumes that the velocity is constant over the entire step. By taking large steps, we are likely to jump over changes in the velocity. Using smaller steps, we will end in a different position. Thus, the choice of step size is a critical parameter in constructing accurate visualization of particle paths in a vector field.

To evaluate Equation 1.2, we can express it as an integral:

$$\vec{x}(t) = \int_t \vec{V}dt \tag{1.3}$$

Although this form cannot be solved analytically for most real-world data, its solution can be approximated using numerical integration techniques. Accurate numerical integration is a topic beyond the scope of this book, but it is known that the accuracy of the integration is a function of the step size *dt*. Because the path is an integration throughout the dataset, the accuracy of the cell interpolation functions and the accuracy of the original vector data play important roles in realizing accurate solutions. No definitive study that relates cell size or interpolation function characteristics to visualization error is yet available. But the lesson is clear: the result of numerical integration must be examined carefully, especially in regions with large vector field gradients. However, as with many other visualization algorithms, the insight gained by using vector-integration techniques is qualitatively beneficial, despite the unavoidable numerical errors.

The simplest form of numerical integration is Euler's method,

$$\vec{x}_{i+1} = \vec{x}_i + \vec{V}_i\Delta t \tag{1.4}$$

where the position at time $\vec{x}_{i+1}$ is the vector sum of the previous position plus the instantaneous velocity times the incremental time step Δt.

Euler's method has error on the order of $O(\Delta t^2)$, which is not accurate enough for some applications. One such example is shown in Fig. 1.17. The velocity field describes perfect rotation about a central point. Using Euler's method, we find that we will always diverge and, instead of generating circles, will generate spirals.

In this chapter we will use the Runge-Kutta technique of order 2 [8]. This is given by the expression

$$\vec{x}_{i+1} = \vec{x}_i + \frac{\Delta t}{2}(\vec{V}_i + \vec{V}_{i+1}) \tag{1.5}$$

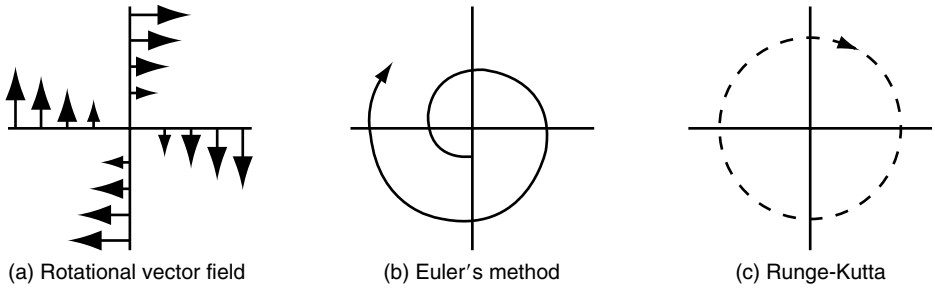


Figure 1.17 Euler's integration (b) and Runge-Kutta integration of order 2 (c) applied to a uniform rotational vector field (a). Euler's method will always diverge.

where the velocity $\vec{V}_{i+1}$ is computed using Euler's method. The error of this method is $O(\Delta t^3)$. Compared to Euler's method, the Runge-Kutta technique allows us to take a larger integration step at the expense of one additional function evaluation. Generally, this tradeoff is beneficial, but like any numerical technique, the best method to use depends on the particular nature of the data. Higher-order techniques are also available, but generally not necessary, because the higher accuracy is countered by error in interpolation function or inherent in the data values. If you are interested in other integration formulas, please check the references at the end of the chapter.

One final note about accuracy concerns: the error involved in either perception or computation of visualizations is an open research area. The discussion in the preceding paragraph is a good example of this: there, we characterized the error in streamline integration using conventional numerical integration arguments. But there is a problem with this argument. In visualization applications, we are integrating across cells whose function values are continuous but whose derivatives are not. As the streamline crosses the cell boundary, subtle effects may occur that are not treated by the standard numerical analysis. Thus, the standard arguments need to be extended for visualization applications.

Integration formulas require repeated transformation from global to local coordinates. Consider moving a point through a dataset under the influence of a vector field. The first step is to identify the cell that contains the point. This operation is a search plus a conversion to local coordinates. Once the cell is found, then the next step is to compute the velocity at that point by interpolating the velocity from the cell points. The point is then incrementally repositioned (using the integration formula in Equation 1.5). The process is then repeated until the point exits the dataset or the distance or time traversed exceeds some specified value.

This process can be computationally demanding. There are two important steps we can take to improve performance:

1. *Improve search procedures*. There are two distinct types of searches. Initially, the starting location of the particle must be determined by a global search procedure. Once the initial location of the point is determined in the dataset, an incremental search procedure can be used. Incremental searching is efficient because the motion of the point is limited within a single cell, or, at most, across a cell boundary. Thus, the search space is greatly limited, and the incremental search is faster relative to the global search.
2. *Coordinate transformation*. The cost of a coordinate transformation from global to local coordinates can be reduced if either of the

following conditions is true: the local and global coordinate systems are identical to each other (or vary by x-y-z translation), or the vector field is transformed from global space to local coordinate space. The image data coordinate system is an example of local coordinates that are parallel to global coordinates, and thus a situation in which global-to-local coordinate transformation can be greatly accelerated. If the vector field is transformed into local coordinates (either as a preprocessing step or on a cell-by-cell basis), then the integration can proceed completely in local space. Once the integration path is computed, selected points along the path can be transformed into global space for the sake of visualization.

1.3.5 Streamlines

A natural extension of the previous time animation techniques is to connect the point position $\vec{x}(t)$ over many time-steps. The result is a numerical approximation to a particle trace represented as a line.

Borrowing terminology from the study of fluid flow, we can define three related line-representation schemes for vector fields.

- *Particle traces* are trajectories traced by fluid particles over time.
- *Streaklines* are the set of particle traces at a particular time t_i that have previously passed through a specified point x_i.
- *Streamlines* are integral curves along a curve s satisfying the equation

$$s = \int_t \vec{V}\, ds, \;\; with\; s = s(x, t) \tag{1.6}$$

for a particular time t.

Streamlines, streaklines, and particle traces are equivalent to one another if the flow is steady. In time-varying flow, a given streamline exists only at one moment in time. Visualization systems generally provide facilities to compute particle traces. However, if time is fixed, the same facility can be used to compute streamlines. In general, we will use the term *streamline* to refer to the method of tracing trajectories in a vector field. Please bear in mind the differences in these representations if the flow is time-varying.

Fig. 1.18 shows 40 streamlines in a small kitchen. The room has two windows, a door (with air leakage), and a cooking area with a

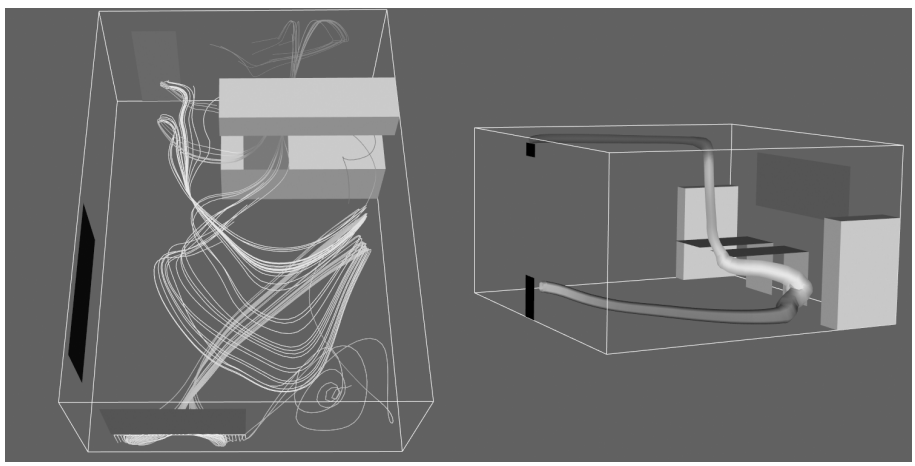


Figure 1.18 Flow velocity computed for a small kitchen (top and side view). Forty streamlines start along the rake positioned under the window. Some eventually travel over the hot stove and are convected upwards. (See also color insert.)

hot stove. The air leakage and temperature variation combine to produce air convection currents throughout the kitchen. The starting positions of the streamlines were defined by creating a *rake*, or curve (and its associated points). There, the rake was a straight line. These streamlines clearly show features of the flow field. By releasing many streamlines simultaneously, we obtain even more information, as the eye tends to assemble nearby streamlines into a "global" understanding of flow field features.

Many enhancements of streamline visualization exist. Lines can be colored according to velocity magnitude to indicate speed of flow. Other scalar quantities such as temperature or pressure also may be used to color the lines. We also may create constant-time dashed lines. Each dash represents a constant time increment. Thus, in areas of high velocity, the length of the dash will be greater relative to regions of lower velocity. These techniques are illustrated in Fig. 1.19 for air flow around a blunt fin. This example consists of a wall with half of a rounded fin projecting into the fluid flow. (Using arguments of symmetry, only half of the domain was modeled.) Twenty-five streamlines are released upstream of the fin. The boundary layer effects near the junction of the fin and wall are clearly evident from the streamlines. In this area, flow recirculation and the reduced flow speed are apparent.

1.4 Tensor Algorithms

Tensor visualization is an active area of research. However, there are a few simple techniques that we can use to visualize 3×3 real symmetric tensors. Such tensors are used to describe the state of displacement or stress in a 3D material. The stress and strain tensors for an elastic material are shown in Fig. 1.20.

In these tensors, the diagonal coefficients are the so-called normal stresses and strains, and the off-diagonal terms are the shear stresses and strains. Normal stresses and strains act perpendicularly to a specified surface, while shear stresses and strains act tangentially to the surface. Normal stress is either compression or tension, depending on the sign of the coefficient.

A 3×3 real symmetric matrix can be characterized by three vectors in 3D called the eigenvectors and three numbers called the eigenvalues of the matrix. The eigenvectors form a 3D coordinate system whose axes are mutually perpendicular. In some applications, particularly the study of materials, these axes are also referred to as the principal axes of the tensor and are physically significant. For example, if

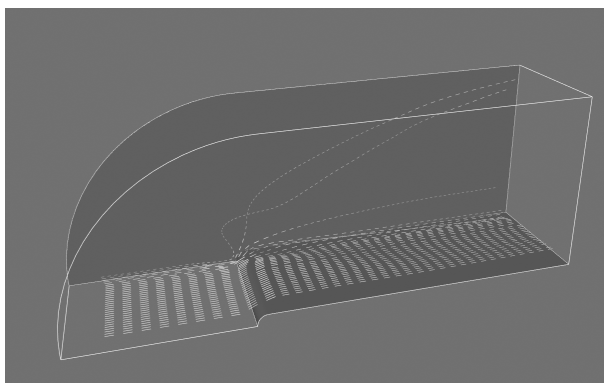


Figure 1.19 Dashed streamlines around a blunt fin. Each dash is a constant time increment. Fast-moving particles create longer dashes than slower-moving particles. The streamlines also are colored by flow density scalar.

$$\begin{bmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{bmatrix} \qquad \begin{bmatrix} \frac{\partial u}{\partial x} & \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}\right) & \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\ \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}\right) & \frac{\partial v}{\partial y} & \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) & \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) & \frac{\partial w}{\partial z} \end{bmatrix}$$

(a) (b)

Figure 1.20 (a) Stress and (b) strain tensors. Normal stresses in the *x*-*y*-*z* coordinate directions are indicated as σ_x, σ_y, σ_z, and shear stresses are indicated as τ_{ij}. Material displacement is represented by *u, v, w* components.

the tensor is a stress tensor, then the principal axes are the directions of normal stress and no shear stress. Associated with each eigenvector is an eigenvalue. The eigenvalues are often physically significant as well. In the study of vibration, eigenvalues correspond to the resonant frequencies of a structure, and the eigenvectors are the associated mode shapes.

Mathematically we can represent eigenvalues and eigenvectors as follows. Given a matrix A, the eigenvector $\vec{x}$ and eigenvalue λ must satisfy the relation

$$A \cdot \vec{x} = \lambda \vec{x} \tag{1.7}$$

For Equation 1.7 to hold, the matrix determinate must satisfy

$$\det|A - \lambda I| = 0 \tag{1.8}$$

Expanding this equation yields an n^{th}-degree polynomial in λ whose roots are the eigenvalues. Thus, there are always n eigenvalues, although they may not be distinct. In general, Equation 1.8 is not solved using polynomial root searching because of poor computational performance. (For matrices of order 3, root searching is acceptable because we can solve for the eigenvalues analytically.) Once we determine the eigenvalues, we can substitute each into Equation 1.8 to solve for the associated eigenvectors.

We can express the eigenvectors of the 3×3 system as

$$\vec{v}_i = \lambda_i \vec{e}_i, \ with\ i = 1, 2, 3 \tag{1.9}$$

with $\vec{e}_i$ a unit vector in the direction of the eigenvalue, and λ_i the eigenvalues of the system. If we order eigenvalues such that

$$\lambda_1 \geq \lambda_2 \geq \lambda_3 \tag{1.10}$$

then we refer to the corresponding eigenvectors $\vec{v}_1, \vec{v}_2$, and $\vec{v}_3$ as the *major, medium*, and *minor* eigenvectors.

1.4.1 Tensor Ellipsoids

This leads us to the tensor ellipsoid technique for the visualization of real, symmetric 3×3 matrices. The first step is to extract eigenvalues and eigenvectors as described in the previous section. Since eigenvectors are known to be orthogonal, the eigenvectors form a local coordinate system. These axes can be taken as the *minor, medium*, and *major* axes of an ellipsoid. Thus, the shape and orientation of the ellipsoid represent the relative size of the eigenvalues and the orientation of the eigenvectors.

To form the ellipsoid we begin by positioning a sphere at the tensor location. The sphere is then rotated around its origin using the eigenvectors, which in the form of Equation 1.9 are direction cosines. The eigenvalues are used to scale the sphere. Using 4×4 transformation matrices, we form the ellipsoid by transforming the sphere centered at the origin using the matrix T:

$$T = T_T \cdot T_R \cdot T_S \tag{1.11}$$

where

$$T_T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.12}$$

$$T_R = \begin{bmatrix} \cos\theta_{x'x} & \cos\theta_{x'y} & \cos\theta_{x'z} & 0 \\ \cos\theta_{y'x} & \cos\theta_{y'y} & \cos\theta_{y'z} & 0 \\ \cos\theta_{z'x} & \cos\theta_{z'y} & \cos\theta_{z'z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where T_T, T_S, and T_R are translation, scale, and rotation matrices. The eigenvectors can be directly plugged in to create the rotation matrix, while the point coordinates x-y-z and eigenvalues $\lambda_1 \geq \lambda_2 \geq \lambda_3$ are inserted into the translation and scaling matrices. A concatenation of these matrices in the correct order forms the final transformation matrix T.

Fig. 1.21a depicts the tensor ellipsoid technique. In Fig. 1.22b we show this technique to visualize material stress near a point load on the surface of a semi-infinite domain. (This is the so-called Boussinesq's problem.) From Saada [33] we have the analytic expression for the stress components in Cartesian coordinates shown in Fig. 1.21c. Note that the z direction is defined as the axis originating at the point of application of the force P. The variable ρ is the distance from the point of load application to a point x-y-z. The orientations of the x and y axes are in the plane perpendicular to the z axis. The rotation in the plane of these axes is unimportant since the solution is symmetric around the z axis. The parameter v is Poisson's ratio, which is a property of the material. Poisson's ratio relates the lateral contraction of a material to axial elongation under a uniaxial stress condition [33,35].

In Fig. 1.22 we visualize the analytical results of Boussinesq's problem from Saada. The left-hand portion of the figure shows the results by displaying the scaled and oriented principal axes of the stress tensor. (These are called *tensor axes*.) In the right-hand portion we use tensor ellipsoids to show the same result. Tensor ellipsoids and tensor axes are a form of *glyph* (see Section 1.5.4) specialized to tensor visualization.

A certain amount of care must be taken to visualize this result, because there is a stress singularity at the point of contact of the load. In a real application, loads are applied over a small area and not at a single point. Plastic behavior prevents stress levels from exceeding a certain point. The results of the visualization, as with any computer process, are only as good as the underlying model.

1.5 Modeling Algorithms

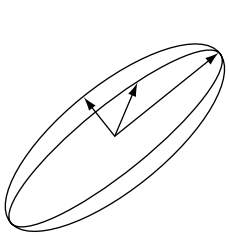
"Modeling algorithms" is the catch-all category for our taxonomy of visualization techniques. Modeling algorithms will typically transform the type of input dataset or use combinations of input data and parameters to affect their result.

1.5.1 Source Objects

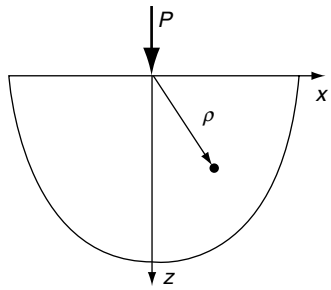
Source objects begin the visualization pipeline. Often, source objects are used to create geometry such as spheres, cones, or cubes to support visualization context, or are used to read in data files. Source objects also may be used to create dataset attributes. Some examples of source objects and their use are as follows.

1.5.1.1 Modeling Simple Geometry

Spheres, cones, cubes, and other simple geometric objects can be used alone or in combination to model geometry. Often, we visualize real-world applications such as air flow in a room and need to show real-world objects such as furniture, windows, or doors. Real-world objects often can be represented using these simple geometric representations. These source



(a) Tensor ellipsoid



(b) Point load on semi-infinite domain

$$\sigma_x = -\frac{P}{2\pi\rho^2}\left(\frac{3zx^2}{\rho^3} - (1-2\nu)\left(\frac{z}{\rho} - \frac{\rho}{\rho+z} + \frac{x^2(2\rho+z)}{\rho(\rho+z)^2}\right)\right)$$

$$\sigma_y = -\frac{P}{2\pi\rho^2}\left(\frac{3zy^2}{\rho^3} - (1-2\nu)\left(\frac{z}{\rho} - \frac{\rho}{\rho+z} + \frac{y^2(2\rho+z)}{\rho(\rho+z)^2}\right)\right)$$

$$\sigma_y = -\frac{3Pz^3}{2\pi\rho^5}$$

$$\tau_{xy} = \tau_{yx} = -\frac{P}{2\pi\rho^2}\left(\frac{3xyz}{\rho^3} - (1-2\nu)\left(\frac{xy(2\rho+z)}{\rho(\rho+z)^2}\right)\right)$$

$$\tau_{xz} = \tau_{zx} = -\frac{3Pxz^2}{2\pi\rho^5}$$

$$\tau_{yz} = \tau_{zy} = -\frac{3Pyz^2}{2\pi\rho^5}$$

c) Analytic solution

Figure 1.21 Tensor ellipsoids. (a) Ellipsoid oriented along eigenvalues (i.e., principal axes) of tensor; (b) pictorial description of Boussinesq's problem; (c) analytic results according to Saada.

objects generate their data procedurally. Alternatively, we may use reader objects to access geometric data defined in data files. These data files may contain more complex geometry, such as that produced by a 3D Computer-Aided Design (CAD) system.

1.5.1.2 Supporting Geometry

During the visualization process, we may use source objects to create supporting geometry. This may be as simple as three lines to represent a coordinate axis or as complex as tubes wrapped around line segments to thicken and enhance their appearance. Another common use is as supplemental input to objects such as streamlines or probe filters. These filters take a second input that defines a set of points. For streamlines, the points determine the initial positions for generating the streamlines. The probe filter uses the points as the position to compute attribute values such as scalars, vectors, or tensors.

1.5.1.3 Data Attribute Creation

Source objects can be used as procedures to create data attributes. For example, we

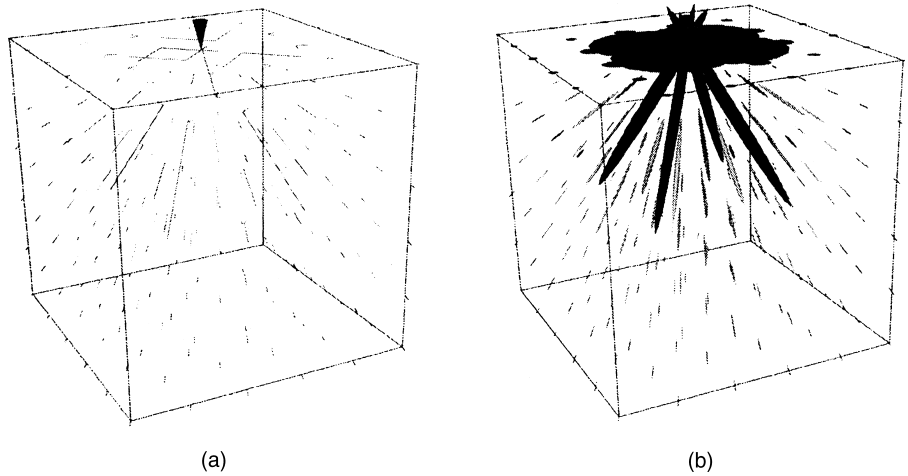


Figure 1.22 Tensor visualization techniques. (a) Tensor axes; (b) tensor ellipsoids.

can procedurally create textures and texture coordinates. Another use is to create scalar values over a uniform grid. If the scalar values are generated from a mathematical function, then we can use the visualization techniques described here to visualize the function. In fact, this leads us to a very important class of source objects: implicit functions.

1.5.2 Implicit Functions

Implicit functions are functions of the form

$$F(\overline{x}) = c \tag{1.13}$$

where c is an arbitrary constant. Implicit functions have three important properties:

- *Simple geometric description*. Implicit functions are convenient tools to describe common geometric shapes, including planes, spheres, cylinders, cones, ellipsoids, and quadrics.
- *Region separation*. Implicit functions separate 3D Euclidean space into three distinct regions. These regions are inside, on, and outside the implicit function. These regions are defined as $F(x, y, z) < 0, F(x, y, z) = 0$, and $F(x, y, z) > 0$, respectively.
- *Scalar generation*. Implicit functions convert a position in space into a scalar value. That is, given an implicit function, we can sample it at a point (x_i, y_i, z_i) to generate a scalar value c_i.

An example of an implicit function is the equation for a sphere of radius R

$$F(x, y, z) = x^2 + y^2 + z^2 - R^2 \tag{1.14}$$

This simple relationship defines the three regions $F(x, y, z) = 0$ (on the surface of the sphere), $F(x, y, z) < 0$ (inside the sphere), and $F(x, y, z) > 0$ (outside the sphere). Any point may be classified inside, on, or outside the sphere simply by evaluating Equation 1.14.

If you have been paying attention, you will note that Equation 1.14 is identical to the equation defining a contour (Equation 1.1). This should provide you with a clue as to the many ways in which implicit functions can be used. These include geometric modeling, selection of data, and visualization of complex mathematical descriptions.

1.5.2.1 Modeling Objects

Implicit functions can be used alone or in combination to model geometric objects.

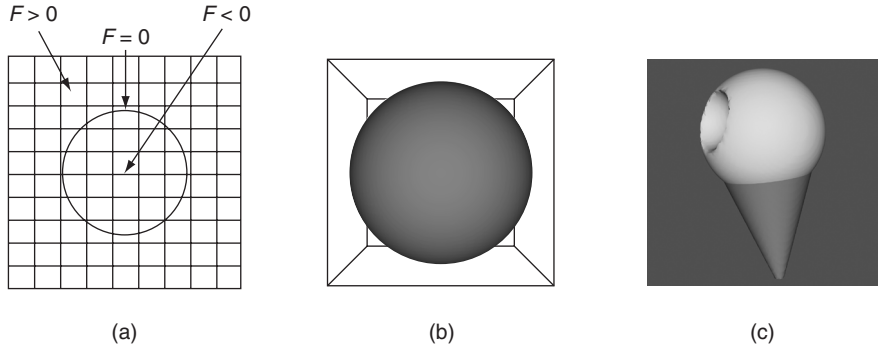


Figure 1.23 Sampling functions. (a) 2D depiction of sphere sampling; (b) isosurface of sampled sphere; (c) Boolean combination of two spheres, a cone, and two planes. (One sphere intersects the other; the planes clip the cone.)

For example, to model a surface described by an implicit function, we sample F on a dataset and generate an isosurface at a contour value c_i. The result is a polygonal representation of the function. Fig. 1.23b shows an isosurface for a sphere of radius = 1 sampled on a volume. Note that we can choose nonzero contour values to generate a family of offset surfaces. This is useful for creating blending functions and other special effects.

Implicit functions can be combined to create complex objects using the Boolean operators union, intersection, and difference. The union operation $F \cup G$ between two functions $F(x, y, z)$ and $G(x, y, z)$ at a point (x_0, y_0, z_0) is the minimum value

$$F \cup G = \min(F(x_0, y_0, z_0), G(x_0, y_0, z_0)) \tag{1.15}$$

The intersection between two implicit functions is given by

$$F \cap G = \max(F(x_0, y_0, z_0), G(x_0, y_0, z_0)) \tag{1.16}$$

The difference of two implicit functions is given by

$$F - G = \max(F(x_0, y_0, z_0), -G(x_0, y_0, z_0)) \tag{1.17}$$

Fig. 1.23c shows a combination of simple implicit functions to create an ice cream cone. The cone is created by clipping the (infinite) cone function with two planes. The ice cream is constructed by performing a difference operation on a larger sphere with a smaller offset sphere to create the "bite." The resulting surface was extracted using surface contouring with isosurface value 0.0.

1.5.2.2 Selecting Data

We can take advantage of the properties of implicit functions to select and cut data. In particular, we will use the region separation property to select data. (We defer the discussion on cutting to Section 1.5.5.)

Selecting or extracting data with an implicit function means choosing cells and points (and associated attribute data) that lie within a particular region of the function. To determine whether a point x-y-z lies within a region, we simply evaluate the point and examine the sign of the result. A cell lies in a region if all its points lie in the region.

Fig. 1.24a shows a 2D implicit function, here an ellipse, used to select the data (i.e., points, cells, and data attributes) contained within it. Boolean combinations also can be used to create complex selection regions, as illustrated in Fig. 1.24b. Here, two ellipses are used in combination to select voxels within a volume dataset. Note that extracting data

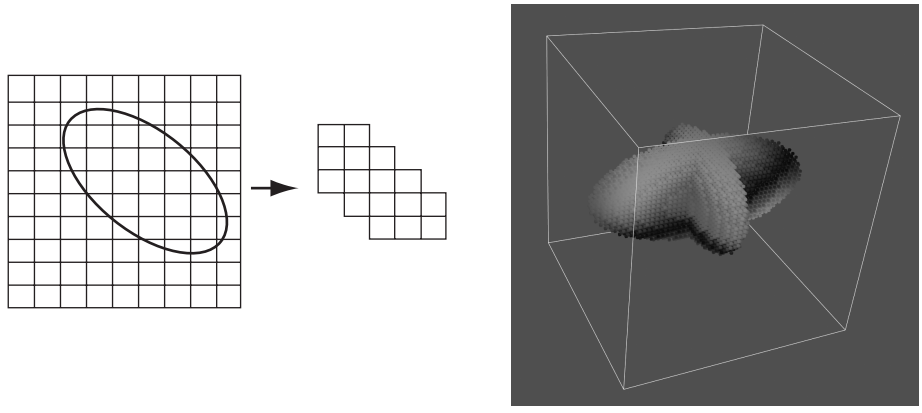


Figure 1.24 Implicit functions used to select data: (a) 2D cells lying in ellipse are selected; (b) two ellipsoids combined using the union operation used to select voxels from a volume. Voxels shrank 50%. (See also color insert.)

often changes the structure of the dataset. In Fig. 1.24 the input type is a volume dataset, while the output type is an unstructured grid dataset.

1.5.2.3 Visualizing Mathematical Descriptions

Some functions, often discrete or probabilistic in nature, cannot be cast into the form of Equation 1.13. However, by applying some creative thinking, we can often generate scalar values that can be visualized. An interesting example of this is the so-called strange attractor.

Strange attractors arise in the study of nonlinear dynamics and chaotic systems. In these systems, the usual types of dynamic motion—equilibrium, periodic motion, and quasi-periodic motion—are not present. Instead, the system exhibits chaotic motion. The resulting behavior of the system can change radically as a result of small perturbations in its initial conditions.

A classical strange attractor was developed by Lorenz [24] in 1963. Lorenz developed a simple model for thermally induced fluid convection in the atmosphere. Convection causes rings of rotating fluid and can be developed from the general Navier-Stokes partial differential equations for fluid flow. The Lorenz equations can be expressed in nondimensional form as

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= \rho x - y - xz \\ \frac{dz}{dt} &= xy - \beta z \end{aligned} \tag{1.18}$$

where x is proportional to the fluid velocity in the fluid ring, y and z measure the fluid temperature in the plane of the ring, the parameters σ and ρ are related to the Prandtl number and Raleigh number, respectively, and β is a geometric factor.

Certainly these equations are not in the implicit form of Equation 1.13, so how do we visualize them? Our solution is to treat the variables x, y, and z as the coordinates of a 3D space, and integrate Equation 1.18 to generate the system "trajectory," that is, the state of the system through time. The integration is carried out within a volume and scalars are created by counting the number of times each voxel is visited. By integrating long enough, we can create a volume representing the "surface" of the

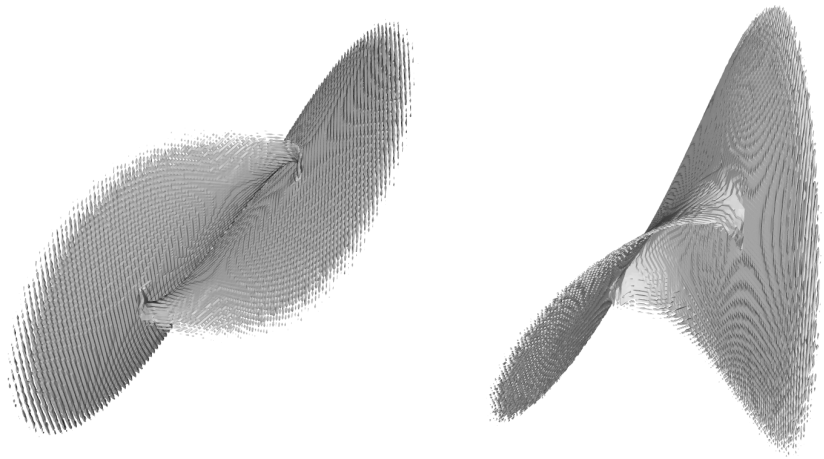


Figure 1.25 Visualizing a Lorenz strange attractor by integrating the Lorenz equations in a volume. The number of visits in each voxel is recorded as a scalar function. The surface is extracted via marching cubes using a visit value of 50. The number of integration steps is 10 million, in a volume of dimensions 200^3. The surface roughness is caused by the discrete nature of the evaluation function. (See also color insert.)

strange attractor, Fig. 1.25. The surface of the strange attractor is extracted by using marching cubes and a scalar value specifying the number of visits in a voxel.

1.5.3 Implicit Modeling

In the previous section, we saw how implicit functions, or Boolean combinations of implicit functions, could be used to model geometric objects. The basic approach is to evaluate these functions on a regular array of points, or volume, and then to generate scalar values at each point in the volume. Then either volume rendering or isosurface generation is used to display the model.

An extension of this approach, called *implicit modeling*, is similar to modeling with implicit functions. The difference lies in the fact that scalars are generated using a distance function instead of the usual implicit function. The distance function is computed as a Euclidean distance to a set of generating primitives such as points, lines, or polygons. For example, Fig. 1.26 shows the distance functions to a point, line, and triangle. Because distance functions are well-behaved monotonic functions, we can define a series of offset surfaces by specifying different isocontour values, where the value is the distance to the generating primitive. The isocontours form approximations to the true offset surfaces, but using high-volume resolution we can achieve satisfactory results.

Used alone the generating primitives are limited in their ability to model complex geometry. By using Boolean combinations of the primitives, however, complex geometry can be easily modeled. The Boolean operations union, intersection, and difference (Equations 1.15, 1.16, and 1.17, respectively) are illustrated in Fig. 1.27. Fig. 1.28 shows the application of implicit modeling to "thicken" the line segments in the text symbol "HELLO." The isosurface is generated on a $110 \times 40 \times 20$ volume at a distance offset of 0.25 units. The generating primitives were combined using the Boolean union operator. Although Euclidean distance is always a nonnegative value, it is possible to use a signed distance function for objects that have an outside and an inside. A negative distance is the negated distance of a point inside the object to the surface of the object. Using a

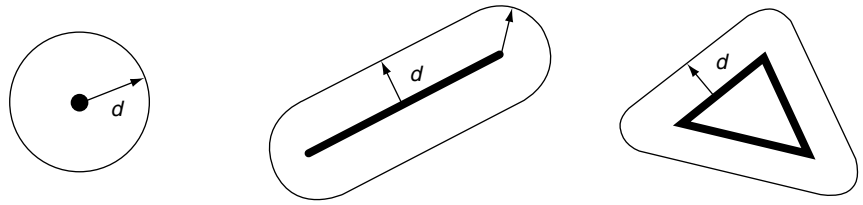


Figure 1.26 Distance functions to a point, line, and triangle.

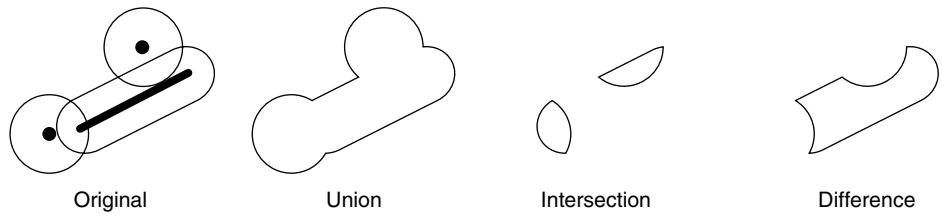


Figure 1.27 Boolean operations using points and lines as generating primitives.

Figure 1.28 Implicit modeling used to thicken a stroked font. Original lines can be seen within the translucent implicit surface.

signed distance function allows us to create offset surfaces that are contained within the actual surface.

Another interesting feature of implicit modeling is that when isosurfaces are generated, more than one connected surface can result. These situations occur when the generating primitives form concave features. Fig. 1.29 illustrates this situation. If desired, multiple surfaces can be extracted by using a connectivity segmentation algorithm.

1.5.4 Glyphs

Glyphs, sometimes referred to as icons, are a versatile technique to visualize data of every type. A glyph is an ''object'' that is affected by its input data. This object may be geometry, a dataset, or a graphical image. The glyph may orient, scale, translate, deform, or somehow alter the appearance of the object in response to data. We have already seen a simple form of glyph: hedgehogs are lines that are oriented, translated, and scaled according to the position and vector value of a point. A variation of this is to use oriented cones or arrows (see Section 1.3.1).

More elaborate glyphs are possible. In one creative visualization technique, Chernoff [6] tied data values to an iconic representation of the human face. Eyebrows, nose, mouth, and other features were modified according to financial data values. This interesting technique built on the human capability to recognize

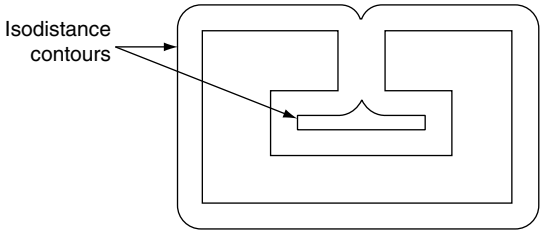


Figure 1.29 Concave features can result in multiple contour lines/surfaces.

facial expression. By tying appropriate data values to facial characteristics, rapid identification of important data points is possible.

In a sense, glyphs represent the fundamental result of the visualization process. Moreover, all the visualization techniques we present can be treated as concrete representations of an abstract glyph class. For example, while hedgehogs are an obvious manifestation of a vector glyph, isosurfaces can be considered a topologically 2D glyph for scalar data. Delmarcelle and Hesselink [11] have developed a unified framework for flow visualization based on types of glyphs. They classify glyphs according to one of three categories:

- *Elementary icons* represent their data across the extent of their spatial domain. For example, an oriented arrow can be used to represent a surface normal.
- *Local icons* represent elementary information plus a local distribution of the values around the spatial domain. A surface normal vector colored by local curvature is one example of a local icon, because local data beyond the elementary information is encoded.
- *Global icons* show the structure of the complete dataset. An isosurface is an example of a global icon.

This classification scheme can be extended to other visualization techniques such as vector and tensor data, or even to nonvisual forms such as sound or tactile feedback. We have found this classification scheme to be helpful when designing visualizations or creating visualization techniques. Often, it gives insight into ways of representing data that can be overlooked.

Fig. 1.30 is an example of glyphing. Small 3D cones are oriented on a surface to indicate the direction of the surface normal. A similar approach could be used to show other surface properties such as curvature or anatomical key points.

1.5.5 Cutting

Often, we want to cut through a dataset with a surface and then display the interpolated data values on the surface. We refer to this technique as *data cutting* or simply *cutting*. The data cutting operation requires two pieces of information: a definition for the surface and a dataset to cut. We will assume that the cutting surface is defined by an implicit function. A typical application of cutting is to slice through a dataset with a plane, and color map the scalar data and/or warp the plane according to vector value.

A property of implicit functions is to convert a position into a scalar value (see Section 1.5.2).

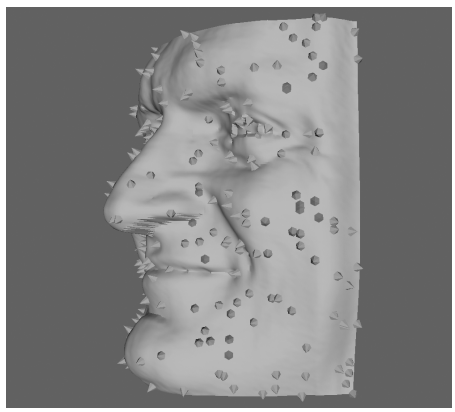


Figure 1.30 Glyphs indicate surface normals on a model of a human face. Glyph positions are randomly selected. (See also color insert.)

We can use this property in combination with a contouring algorithm (e.g., marching cubes) to generate cut surfaces. The basic idea is to generate scalars for each point of each cell of a dataset (using the implicit cut function) and then contour the surface value $F(x, y, z) = 0$.

The cutting algorithm proceeds as follows. For each cell, function values are generated by evaluating $F(x, y, z)$ for each cell point. If all the points evaluate positive or negative, then the surface does not cut the cell. However, if the points evaluate positive and negative, then the surface passes through the cell. We can use the cell contouring operation to generate the isosurface $F(x, y, z) = 0$. Data-attribute values can then be computed by interpolating along cut edges.

Fig. 1.31 illustrates a plane cut through a structured grid dataset. The plane passes through the center of the dataset with normal (−0.287, 0, 0.9579). For comparison purposes, a portion of the grid geometry is also shown. The grid geometry is the grid surface $k = 9$ (shown in wireframe). One benefit of cut surfaces is that we can view data on (nearly) arbitrary surfaces. Thus, the structure of the dataset does not constrain how we view the data.

We can easily make multiple planar cuts through a structured grid dataset by specifying multiple iso-values for the cutting algorithm. Fig. 1.32 shows 100 cut planes generated perpendicular to the camera's view plane normal. Rendering the planes from back to front with an opacity of 0.05 produces a simulation of volume rendering.

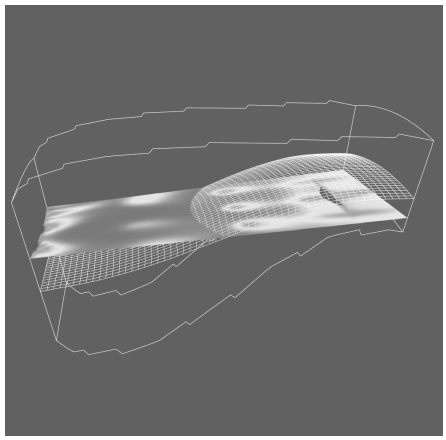


Figure 1.31 Cut through structured grid with plane. The cut plane is shown solid shaded. A computational plane of constant k value is shown in wireframe for comparison. The colors correspond to flow density. Cutting surfaces are not necessarily planes: implicit functions such as spheres, cylinders, and quadrics can also be used. (See also color insert.)

This example illustrates that cutting the volumetric data in a structured grid dataset produces polygonal cells. Similarly, cutting polygonal data produces lines. Using a single plane equation, we can extract "contour lines" from a surface model defined with polygons. Fig. 1.33 shows contours extracted from a surface model of the skin. At each vertex in the surface model, we evaluate the equation of the plane $F(x, y, z) = c$ and store the value

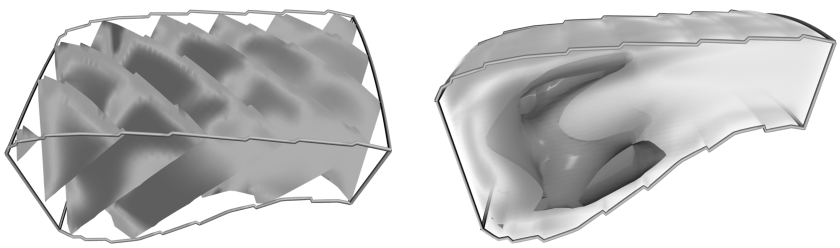


Figure 1.32 100 cut planes with opacity of 0.05, rendered back-to-front to simulate volume rendering. (See also color insert.)

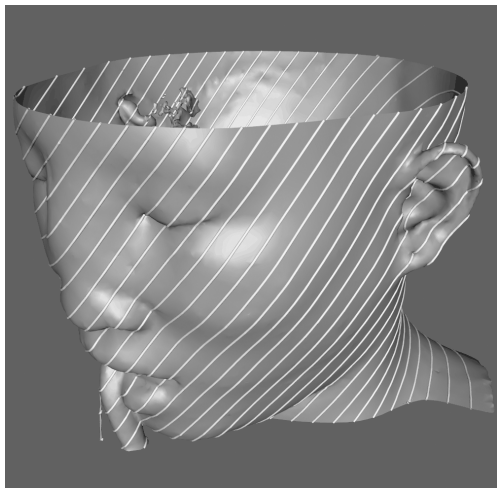


Figure 1.33 Cutting a surface model of the skin with a series of planes produces contour lines. Lines are wrapped with tubes for visual clarity. (See also color insert.)

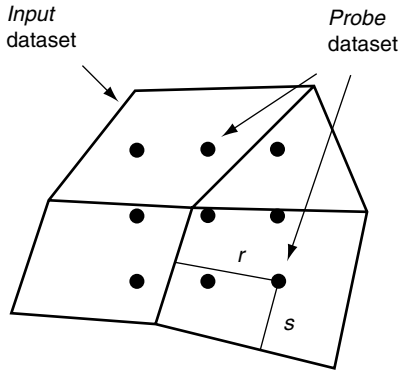


Figure 1.34 Probing data. The geometry of one dataset (*Probe*) is used to extract dataset attributes from another dataset (*Input*).

of the function as a scalar value. Cutting the data with 46 iso-values from 1.5 to 136.5 produces contour lines that are 3 units apart.

1.5.6 Probing

Probing obtains dataset attributes by sampling one dataset (the input) with a set of one or more points (the probe), as shown in Fig. 1.34. Probing is also called *resampling*. Examples include probing an input dataset with a sequence of points along a line, on a plane, or in a volume. The result of the probing is a new dataset (the output) with the topological and geometric structure of the probe dataset and point attributes interpolated from the input dataset. Once the probing operation is complete, the output dataset can be visualized with any of the appropriate techniques described previously.

As Fig. 1.34 indicates, the details of the probing process are as follows. For every point in the probe dataset, the location in the input dataset (i.e., cell, subcell, and parametric coordinates) and interpolation weights are determined. Then the data values from the cell are interpolated to the probe point. Probe points that are outside the input dataset are assigned a nil (or appropriate) value. This process repeats for all points in the probe dataset.

Probing can be used to reduce data or to view data in a particular fashion.

- Data is reduced when the probe operation is limited to a subregion of the input dataset or the number of probe points is less than the number of input points.
- Data can be visualized with specialized techniques by sampling on selected datasets. For example, using a line probe enables *x*-*y* plotting along a line, and using a plane probe allows surface color mapping or line contouring on the plane.

Probing must be used carefully or errors may be introduced. Undersampling data in a region can miss important high-frequency information or localized data variations. Oversampling data, while not creating error, can give false confidence in the accuracy of the data. Thus the sampling frequency should have a similar density as the input dataset, or if higher density, the visualization should be carefully annotated as to the original data frequency.

One important application of probing converts irregular or unstructured data to structured form using a probe volume of appropriate

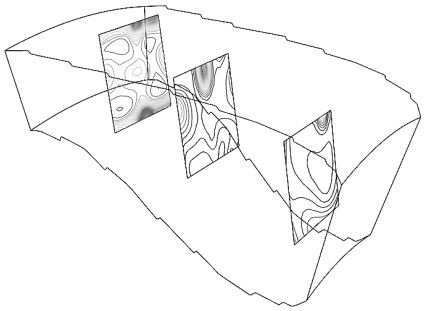


Figure 1.35 Probing data in a combustor. Probes are regular arrays of 50^2 points that are passed through a contouring filter.

resolution to sample the unstructured data. This is useful if volume rendering or another volume technique is to be used to visualize the data.

Fig. 1.35 shows an example of three probes. The probes sample flow density in a structured grid. The output of the probes is passed through a contour filter to generate contour lines. As this figure illustrates, we can be selective with the location and extent of the probe, allowing us to focus on important regions in the data.

1.5.7 Data Reduction

One of the major challenges facing the scientific visualization community is the increasing size of data. While just a short time ago data sizes of a gigabyte were considered large, terabyte and even petabyte data sizes are now available. Because the value of the visualization process is tied to its ability to effectively convey information about large and complex data, it is absolutely essential to find techniques to address this situation. A simple but effective approach is to use methods to reduce data size prior to the visualization process. The approaches taken depend on the type of data; for example, subsampling works well for structured data. Unstructured data (such as polygonal meshes) requires more sophisticated techniques. Since this topic is worth several books on its own, we present some introductory approaches to data reduction. Note that the use of probing is also an excellent data-reduction tool.

1.5.7.1 Subsampling

Subsampling (Fig. 1.36) is a method that reduces data size by selecting a subset of the original data. The subset is specified by choosing a parameter n, specifying that every nth data point is to be extracted. For example, in structured datasets such as image data and structured grids, selecting every nth point produces the results shown in Fig. 1.36. Subsampling modifies the topology of a dataset. When points or cells are not selected, this leaves a topological "hole." Dataset topology must be modified to fill the hole. In structured data, this is simply a uniform selection across the structured *i-j-k* coordinates. In structured data, the hole must be filled in by using triangulation or other complex tessellation schemes. Subsampling is not typically performed on unstructured data because of its inherent complexity.

1.5.7.2 Decimation

Unstructured data can be reduced in size by applying a variety of decimation algorithms (also known as *polygon reduction* when applied to polygonal meshes). There are several approaches to decimation based on differing operations performed on the mesh (Fig. 1.37). Vertex removal deletes a vertex and all attached cells. The resulting hole is then triangulated.

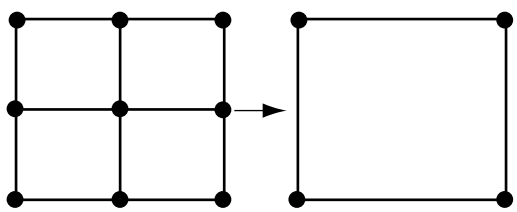


Figure 1.36 Subsampling structured data.

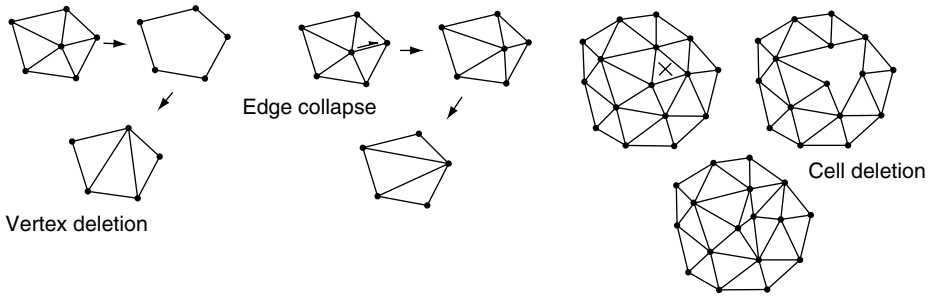


Figure 1.37 Decimating unstructured data.

Edge collapse results in merging two vertices into one. The position of the merged point is controlled by the particulars of the error metric and algorithm: choosing one of the two endpoints, or a point on the edge, is common. Some algorithms compute an optimal merge position based on minimizing error to the original data. Finally, some techniques may delete an entire cell (e.g., triangle) and attached cells and then retriangulate the resulting hole.

Decimation algorithms depend on the evaluation of an error metric to determine the operation to apply to the mesh. Simple approaches such as distance to an "average" plane work reasonably well. Probably the most widely used error metric is based on an accumulation of error represented by a quadric. The so-called quadric error metric measures the distance to a set of planes, each plane corresponding to an original triangle in the input mesh.

1.6 Bibliographic Notes

Color mapping is a widely studied topic in imaging, computer graphics, visualization, and human factors. [12,30,42]. You also may want to learn about the physiological and psychological effects of color on perception. The text by Wyszecki and Stiles [44] serves as an introductory reference.

Contouring is a widely studied technique in visualization because of its importance and popularity. Early techniques were developed for 2D data [43]. 3D techniques were developed initially as contour connecting methods [15]—that is, given a series of 2D contours on evenly spaced planes, connecting the contours to create a closed surface. Since the introduction of marching cubes, many other techniques have been implemented [13,26,28]. A particularly interesting reference is given by Livnat et al. [22]. They show a contouring method with the addition of a preprocessing step that generates isocontours in near-optimal time.

Although we barely touched the topic, the study of chaos and chaotic vibrations is a delightfully interesting topic. Besides the original paper by Lorenz [24], the book by Moon [27] is a good place to start.

2D and 3D vector plots have been used by computer analysts for many years [16]. Streamlines and streamribbons also have been applied to the visualization of complex flows [41]. Good general information on vector visualization techniques is given by Helman and Hesselink [19] and Richter et al. [31].

Tensor visualization techniques are relatively few in number. Most techniques are glyph-oriented [10, 18]. We will see more techniques in later chapters.

Blinn [3], Bloomenthal [4,5], and Wyvill [45] have been important contributors to implicit modeling. Implicit modeling is currently popular in computer graphics for modeling "soft" or "blobby" objects. These techniques are simple, powerful, and becoming widely used for advanced computer graphics modeling.

Polygon reduction is a relatively new field of study. SIGGRAPH '92 marked a flurry of interest with the publication of two papers on this topic [32, 40]. Since then a number of valuable techniques have been published. One of the best techniques, in terms of quality of results, is given by Hoppe [21], although it is limited in time and space because it is based on formal optimization techniques. Other interesting methods include those by Hinker and Hansen [20]

and Rossignac and Borel [32]. One promising area of research is multiresolution analysis, where wavelet decomposition is used to build multiple levels of detail (LODs) in a model [14]. The most recent work in this field stresses progressive transmission of 3D triangle meshes [21], improved error measures [17], and algorithms that modify mesh topology [29,36]. An extensive book on the technology is available that includes specialized methods for terrain simplification [25].

References

1. R. H. Abraham and C. D. Shaw. *Dynamics: The Geometry of Behavior*. Aerial Press, Santa Cruz, CA, 1985.
2. C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, and D. Schlegel. The application visualization system: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.
3. J. F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982.
4. J. Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*. 5(4):341–355, 1982.
5. J. Bloomenthal, *Introduction to Implicit Surfaces*. San Francisco, Morgan Kaufmann, 1997.
6. H. Chernoff. Using faces to represent points in *K*-dimensional space graphically. *J. American Statistical Association*, 68:361–368, 1973.
7. H. Cline, W. Lorensen, and W. Schroeder. 3D phase contrast MRI of cerebral blood flow and surface anatomy. *J. Computer Assisted Tomography*, 17(2):173–177, 1993.
8. S. D. Conte and C. de Boor. *Elementary Numerical Analysis*. New York, McGraw-Hill, 1972.
9. *Data Explorer Reference Manual*. IBM Corp, Armonk, NY, 1991.
10. W. C. de Leeuw and J. J. van Wijk. A probe for local flow field visualization. In *Proceedings of Visualization '93*, pages 39–45, IEEE Computer Society Press, Los Alamitos, CA, 1993.
11. T. Delmarcelle and L. Hesselink. A unified framework for flow visualization. In *Computer Visualization Graphics Techniques for Scientific and Engineering Analysis* (R. S. Gallagher, ed.). Boca Raton, FL, CRC Press, 1995.
12. H. J. Durrett. *Color and the Computer*. Boston, Academic Press, 1987.
13. M. J. Durst. Additional reference to marching cubes. *Computer Graphics*, 22(2):72–73, 1988.
14. M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings SIGGRAPH* '95, pages 173–182, 1995.
15. H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702, 1977.
16. A. J. Fuller and M. L. X. dosSantos. Computer generated display of 3D vector fields. *Computer Aided Design*, 12(2):61–66, 1980.
17. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings SIGGRAPH '97*, pages 209–216, 1997.
18. R. B. Haber and D. A. McNabb. Visualization idioms: a conceptual model to scientific visualization systems. *Visualization in Scientific Computing* (G. M. Nielson, B. Shriver, L. J. Rosenblum, eds.). IEEE Computer Society Press, pages 61–73, 1990.
19. J. Helman and L. Hesselink. Representation and display of vector field topology in fluid flow data sets. *Visualization in Scientific Computing* (G. M. Nielson, B. Shriver, L. J. Rosenblum, eds.). IEEE Computer Society Press, pages 61–73, 1990.
20. P. Hinker and C. Hansen. Geometric optimization. In *Proceedings of Visualization '93*, pages 189–195, 1993.
21. H. Hoppe. Progressive meshes. In *Proceedings SIGGRAPH '96*, pp. 96–108, 1996.
22. Y. Livnat, H. W. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm for structured and unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 2(1), 1996.
23. W. E. Lorensen and H. E. Cline. Marching cubes: a high-resolution 3D surface construction algorithm. *Computer Graphics*, 21(3):163–169, 1987.
24. E. N. Lorenz. Deterministic non-periodic flow. *J. Atmospheric Science*, 20:130–141, 1963.
25. D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. San Francisco, Morgan Kaufmann, 2002.
26. C. Montani, R. Scateni, and R. Scopigno. A modified lookup table for implicit disambiguation of marching cubes. *Visual Computer*, (10):353–355, 1994.
27. F. C. Moon. *Chaotic Vibrations*. New York, Wiley-Interscience, 1987.
28. G. M. Nielson and B. Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *Proceedings of Visualization '91*,

pages 83–91, IEEE Computer Society Press, Los Alamitos, CA, 1991.

29. J. Popovic and H. Hoppe. Progressive simplicial complexes. In *Proceedings of SIGGRAPH '97*, pages 217–224, 1997.
30. P. Rheingans. Color, change, and control for quantitative data display. In *Proceedings of Visualization '92*, pages 252–259. IEEE Computer Society Press, Los Alamitos, CA, 1992.
31. R. Richter, J. B. Vos, A. Bottaro, and S. Gavrilakis. Visualization of flow simulations. *Scientific Visualization and Graphics Simulation* (D. Thalmann, ed.), pages 161–171. New York, John Wiley and Sons, 1990.
32. J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications* (B. Falcidieno and T. Kunii, eds.), pages 455–465. Berlin, Springer-Verlag, 1993.
33. A. S. Saada. *Elasticity Theory and Applications*. New York, Pergamon Press, 1974.
34. W. Schroeder, J. Zarge, and W. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92)*, 26(2):65–70, 1992.
35. W. Schroeder. A topology modifying progressive decimation algorithm. In *Proceedings of Visualization '97*. IEEE Computer Society Press, Los Alamitos, CA, 1997.
36. W. Schroeder, K. Martin, and W. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, 3rd Edition*. Clifton Park, NY, Kitware, Inc., 2003.
37. *SCIRun: A Scientific Computing Problem Solving Environment*. Scientific Computing and Imaging Institute (SCI), http://software.sci.utah.edu/scirun.html, 2002.
38. S. P. Timoshenko and J. N. Goodier. *Theory of Elasticity*, 3rd Ed. New York, McGraw-Hill, 1970.
39. E. R. Tufte. *The Visual Display of Quantitative Information*. Cheshire, CT, Graphics Press, 1990.
40. G. Turk. Re-tiling of polygonal surfaces. *Computer Graphics (SIGGRAPH '92)*, 26(2): 55–64, 1992.
41. G. Volpe. Streamlines and streamribbons in aerodynamics. Technical Report AIAA-89-0140, 27th Aerospace Sciences Meeting, 1989.
42. C. Ware. Color sequences for univariate maps: theory, experiments and principles. *IEEE Computer Graphics and Applications*, 8(5):41–49, 1988.
43. D. F. Watson. *Contouring: A Guide to the Analysis and Display of Spatial Data*. Pergamon Press, New York, 1992.
44. G. Wyszecki and W. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. New York, John Wiley and Sons, 1982.
45. G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *Visual Computer*, 2(4):227–234, 1986.