CPS 533 Scientific Visualization

Wensheng Shen

Department of Computational Science
SUNY Brockport
Chapter 8: Advanced Data Representation

- Geometric relationships
- Topological relationships
- Computational methods for cells and datasets
8.1 Coordinate systems

- Global coordinate system
- Dataset coordinate system
- Structured coordinate system
Global coordinate system

- The global coordinate system is a Cartesian, three-dimensional space. Each point is expressed as a triplet of values \((x,y,z)\) along the \(x\), \(y\), and \(z\) axes. This is the same system that was described in Chapter 3.

- The global coordinate system is always used to specify dataset geometry (i.e., the point coordinates), and data attributes such as normals and vectors. We will use the word “position” to indicate that we are using global coordinates.
The dataset, or local, coordinate system is based on combined topological and geometric coordinates. The topological coordinate is used to identify a particular cell, and the geometric coordinate is used to identify a particular location within the cell. In this context, “location” refers to local or dataset coordinates.

The topological coordinate is an “id”, a unique, nonnegative integer number referring to either a dataset point or cell. For a composite cell, we use an additional “sub-id” to refer to a particular primary cell that composes the composite cell. The sub-id is also unique and nonnegative. The id and sub-id together select a particular primary cell.

To specify a location within the primary cell, we use geometric coordinates. These geometric coordinates, or parametric coordinates, are coordinates “natural” to the particular topology and dimension of a cell.

Every cell type will have its own parametric coordinate system.
For a polyline cell type, we can specify the position of a point by indicating (1) the polyline cell id, (2) the primary cell (i.e., line) sub-id, and (3) the parametric coordinate of the line.

Since the line is a one-dimensional object, the parametric coordinate is based on the one-dimensional parameter \( r \), then any point along the line is given by a linear combination of the two end points of the line \( x_i \) and \( x_{i+1} \)

\[ x(r) = (1 - r)x_i + rx_{i+1} \]

where the parametric coordinate \( r \) is constrained between \((0,1)\). In this equation we are assuming that the sub-id is equal to \( i \).

The number of parametric coordinates corresponds to the topological dimension of the Cell. Three-dimensional cells will be characterized by the three parametric coordinates \((r, s, t)\). For cells of topological order less than three, we will ignore the last \((3-n)\) parametric coordinates, where \( n \) is the topological order of the cell. Each parametric coordinate is ranged between \((0, 1)\).
Structured coordinate system

- Structured coordinate system is based on the i-j-k indexing scheme, and it is a natural way to describe components of a structured dataset. We can specify points, lines, surfaces, and volumes by fixing some indices and allowing others to change within a limited range. If we fix three indices, we specify a point; if we fix two indices, we specify a line; if we fix one indices, we specify a surface; if we allow three indices to change, we specify a volume.

- The structured coordinate system is generally used to specify a region of interest (ROI).

- There is a simple relationship between the point and cell id of the dataset coordinate system and the structured coordinate system.
We can obtain a point id $p_{id}$ given the indices $(i_p, j_p, k_p)$ and dimensions $(n_x, n_y, n_z)$,

$$p_{id} = i_p + j_p n_x + k_p n_x n_y$$

Where $0 \leq i_p < n_x$, $0 \leq j_p < n_y$, $0 \leq k_p < n_z$. We can use this id to index into an array of points or point attribute data. This equation implicitly assumes an ordering of the points in topological space. A similar relationship exists for cell id's

$$cell_{id} = i_p + j_p (n_x - 1) + k_p (n_x - 1)(n_y - 1)$$

We can notice that there are one fewer cells along each topological axes than there are points.
8.2 Interpolation functions

- Computer visualization deals with discrete data. The data is either supplied at a finite number of points or created by sampling continuous data at finite number of points. But we often need information at positions other than these discrete point locations. In this case, we need to interpolate data from known points to some intermediate point using interpolation functions.

- Interpolation functions relate the values at cell points to the interior of the cell. We assume that information is defined at cell points, and that we must interpolate from these point. We can express the result as a weighted average of the data values at each cell point.
The interpolation functions are a linear combination of the data values at the cell points. To interpolate data from the cell point \( p_i \) to a point \( p \) that is inside the cell, we need

1. the data value at each cell point,
2. the parametric coordinates of the point \( p \) within the cell, and
3. the cell type including interpolation functions.

\[
d = \sum_{i=0}^{n-1} W_i \cdot d_i
\]

Where \( d \) is the data value at the interior cell location \( (r,s,t) \), \( d_i \) is the data value at the \( i^{th} \) cell point, and \( W_i \) is a weight at the \( i^{th} \) cell point. The interpolation weights are functions of the parametric coordinates \( W_i = W(r,s,t) \). In addition, because we want \( d = d_i \) when the interior point coincides with a cell point, we can place additional constraints on the weights.
The additional constraints are

\[ W_i = 1, \; W_j = 0 \text{ when } p = p_i \text{ and } i \neq j \]

The interpolated data value \( d \) should be no smaller than the minimum \( d_i \) and no larger than the maximum \( d_j \), that is

\[ \sum W_i = 1, \; 0 \leq W_i \leq 1 \]

\[ d(r) = (1-r)d_0 + rd_1 \]

The interpolation functions are of a characteristic shape. They reach their maximum value \( W_i = 1 \) at cell point \( p_i \) and are zero at all other points. It can be used to interpolate any data value defined at the cell points to any other point within the cell. We have only to define the specific interpolation function \( W_i \) for each cell type.
Specific forms

For each primary cell type, we define the parametric coordinate system and interpolation function. Composite cells use the interpolation functions and parametric coordinates of their composing primary cells. The only difference between primary and composite cells is that composite cells use the additional sub-id to specify a particular primary cell.
**Vertex:** vertex cells do not require parametric coordinate or interpolation functions since they are zero-dimensional. The single weighting function is \( W_0 = 1 \).

\[ W_0 = 1 \]

\[ x_0 \]

\[ r = 0 \]

\[ r = 1 \]

\[ x_1 \]

**Line:** the line is described using the single parametric coordinate \( r \).

\[ W_0 = (1-r)(1-s) \]

\[ W_1 = r(1-s) \]

\[ W_2 = (1-r)s \]

\[ W_3 = rs \]

**Pixel:** the pixel is described using the two parametric coordinate \((r,s)\). The pixel edges are constrained to be parallel to the global coordinate axes. The interpolation functions for a pixel is called *bilinear interpolation.*
Quadrilateral: the quadrilateral is described using the two parametric coordinates \((r, s)\)

Triangle: the triangle is characterized using the two parametric coordinates \((r, s)\).
The polygon is characterized using the two parametric coordinates \((r,s)\). The parametric coordinate system is defined by creating a rectangular oriented along the first edge of the polygon. The rectangle also must bound the polygon. The problem of polygon is that we don’t know how many vertices define the polygon. As a result, it is not possible to create general interpolation functions explicitly. Instead, we use a function based on weighed distance squared from each polygon vertex. The weighted distance squared Interpolation functions work well in practice, but there are rare cases where points topologically distant from the interior of a polygon have an undue effect on the polygon Interior. These situations occur only if the polygon is concave and wraps around on itself.

\[
W_i = \frac{\left(\frac{1}{r_i}\right)^2}{\sum \left(\frac{1}{r_i}\right)^2}
\]

Region of strong influence

*Potential problem with distance-based Interpolation functions.*
The tetrahedron is described using the three parametric coordinates \((r,s,t)\).

\[
\begin{align*}
W_0 &= (1-r)(1-s)(1-t) \\
W_1 &= r(1-s)(1-t) \\
W_2 &= (1-r)s(1-t) \\
W_3 &= rs(1-t) \\
W_4 &= (1-r)(1-s)t \\
W_5 &= r(1-s)t \\
W_6 &= (1-r)st \\
W_7 &= rst
\end{align*}
\]

The voxel is described using the three parametric coordinates \((r,s,t)\). The voxel edges are constrained to lie parallel to the global coordinate axes. The interpolation function is often called \textit{trilinear interpolation} function.
The hexahedron is described using the three parametric coordinates \((r, s, t)\)

\[
\begin{align*}
W_0 &= (1-r)(1-s)(1-t) \\
W_1 &= r(1-s)(1-t) \\
W_2 &= rs(1-t) \\
W_3 &= (1-r)s(1-t) \\
W_4 &= (1-r)(1-s)t \\
W_5 &= r(1-s)t \\
W_6 &= rst \\
W_7 &= (1-r)st
\end{align*}
\]
8.3 Coordinate transformation

Coordinate transformation is a common visualization operation. This may be either transformation dataset coordinates to global coordinates, or global coordinates to dataset coordinates.
To transform between dataset coordinates and global coordinates, we start by identifying a primary cell using the cell id and sub-id. Then the global coordinates are generated from the parametric coordinates by using the interpolation function. Given cell points $p_i = p_i(x_i, y_i, z_i)$

$$p = \sum_{i=0}^{n-1} W_i(r_0, s_0, t_0) p_i$$

where the interpolation weights $W_i$ are evaluated at the parametric coordinate $(r_0, s_0, t_0)$.

*Iso-parametric* interpolation is the interpolation of using the same order interpolation functions for both data and cell geometry;

*Super-parametric* interpolation is the interpolation when the order of the interpolation functions for geometry is greater than those used for data.

*Sub-parametric* interpolation is the interpolation when the order of the interpolation functions for geometry is less than those used for data.

We always use the iso-parametric interpolation for visualization applications.
Global to dataset coordinates

To transform from dataset coordinates to global coordinates, we must:
1. Identify the particular cell \( C_i \) that contains the global point \( p \).
2. Solve the equation for the parametric coordinates of \( p \).

\[
p = \sum_{i=0}^{n-1} W_i(r_0, s_0, t_0) p_i
\]

To identify the cell \( C_i \), we must do some form of searching. A simple but inefficient approach is to visit every cell in a dataset and determine whether \( p \) is inside any cell. If so, then we have found the correct cell and stop the search. Otherwise, we check the next cell in the list.

The search process can be accelerated by spatially organized structures such as an octree or three-dimensional hash table.

We have to solve the interpolation function for the parametric coordinates of \( p \) analytically or numerically.
Interpolation for a line and any linear combinations takes the following form:

\[
\begin{align*}
    r &= \frac{x - x_0}{x_1 - x_0} = \frac{y - y_0}{y_1 - y_0} = \frac{z - z_0}{z_1 - z_0}.
\end{align*}
\]

Vertices, lines, triangles, and tetrahedra use linear combinations. The quadrilateral and hexahedron interpolation functions are nonlinear because they are products of linear expressions for the parametric coordinates. The interpolation functions for pixels and voxels are nonlinear as well, but because of their special orientation with respect to the x, y, and z coordinate axes, we can solve them exactly.

To solve the interpolation functions for parametric coordinates we must use nonlinear techniques for the solution of a system of equations. A simple and effective technique is Newton’s method.
To use Newton’s method, we begin by defining three functions for the known global coordinate \( p = p(x, y, z) \) in terms of the interpolation functions \( W_i = W_i(r,s,t) \)

\[
x - \sum W_i x_i = f(r,s,t) = 0
\]

\[
y - \sum W_i y_i = g(r,s,t) = 0
\]

\[
z - \sum W_i z_i = h(r,s,t) = 0
\]

Expanding the functions using a Taylor’s series approximation,

\[
f = 0 = f_0 + \frac{\partial f}{\partial r} (r-r_0) + \frac{\partial f}{\partial s} (s-s_0) + \frac{\partial f}{\partial t} (t-t_0) + ...
\]

\[
g = 0 = g_0 + \frac{\partial g}{\partial r} (r-r_0) + \frac{\partial g}{\partial s} (s-s_0) + \frac{\partial g}{\partial t} (t-t_0) + ...
\]

\[
h = 0 = h_0 + \frac{\partial h}{\partial r} (r-r_0) + \frac{\partial h}{\partial s} (s-s_0) + \frac{\partial h}{\partial t} (t-t_0) + ...
\]
Using an iterative procedure, we can solve the equation for the parametric coordinates:

\[
\begin{bmatrix}
    r_{i+1} \\
    s_{i+1} \\
    t_{i+1}
\end{bmatrix} =
\begin{bmatrix}
    r_i \\
    s_i \\
    t_i
\end{bmatrix} - 
\begin{bmatrix}
    \frac{\partial f}{\partial r} & \frac{\partial f}{\partial s} & \frac{\partial f}{\partial t} \\
    \frac{\partial g}{\partial r} & \frac{\partial g}{\partial s} & \frac{\partial g}{\partial t} \\
    \frac{\partial h}{\partial r} & \frac{\partial h}{\partial s} & \frac{\partial h}{\partial t}
\end{bmatrix}^{-1}
\begin{bmatrix}
    f_i \\
    g_i \\
    h_i
\end{bmatrix}
\]

The interpolation functions are well behaved, Newton’s method surely converges, and it converges quadratically.
8.4 Computing derivatives

The interpolation functions enable us to compute data values at arbitrary locations within a cell. They also allow us to compute the rate of change, or derivatives, of data values. For example, given displacements at cell points we can compute cell strains and stresses, or given pressure values, we can compute the pressure gradient at a specified location.
Using geometric arguments, we can compute the derivatives in the r parametric

\[
\frac{ds}{dr} = \frac{(s_1 - s_0)}{1} = s_1 - s_0
\]

where \(s_i\) is the data value at point \(i\). In the local coordinate system \(x'\), which is parallel to the \(r\) coordinate system (that is, it lies along the vector \(x_1-x_0\)), the derivative is

\[
\frac{ds}{dx'} = \frac{(s_1 - s_0)}{l}
\]

Derivative due to chain rule:

\[
\frac{d}{dr} = \frac{d}{dx'} \cdot \frac{dx'}{dr}
\]

\[
\frac{d}{dx'} = \left(\frac{d}{dr}\right) \frac{1}{\frac{d}{dr} x'}
\]

\[
\frac{d}{dr} x' = \frac{d}{dr} \left(\sum_{i=0}^{1} W_i \cdot x_i'\right) = -x_0' + x_1' = l
\]
To convert the derivative in the $x'$ coordinate system into the global $x$-$y$-$z$, we create a unit vector $v$ as

$$v = \frac{x_1 - x_0}{|x_1 - x_0|}$$

where $x_0$ and $x_1$ are the locations of the two end points of the line. Then the derivatives in the $x$, $y$, and $z$ directions can be computed by taking the dot products along the axes.

$$\frac{ds}{dx} = \left(\frac{s_1 - s_0}{l}\right)v \cdot (1,0,0)$$

$$\frac{ds}{dy} = \left(\frac{s_1 - s_0}{l}\right)v \cdot (0,1,0)$$

$$\frac{ds}{dz} = \left(\frac{s_1 - s_0}{l}\right)v \cdot (0,0,1)$$

To summarize this process, derivatives are computed in the local $r$-$s$-$t$ parametric space using cell interpolation. These are then transformed into a local $x'$-$y'$-$z'$ Cartesian system. If the $x'$-$y'$-$z'$ system is not aligned with the global $x$-$y$-$z$ coordinate system, another transformation is required to generate the result.
The $3 \times 3$ matrix $J$ is called the Jacobian matrix, and it relates the parametric coordinate derivatives to the global coordinate derivatives. We solve for the global derivatives by taking the inverse of the Jacobian matrix.

\[
\frac{\partial}{\partial x_i} = J^{-1} \frac{\partial}{\partial r_i}
\]
The inverse of the Jacobian always exists as long as there is a one-to-one correspondence between the parametric and global coordinate systems. This means that for any \((r,s,t)\) coordinate, there is only one corresponding \((x,y,z)\) coordinate.

In our one-dimensional example, the derivatives along the line were constant. However, other interpolation functions may yield non-constant derivatives. Here, the Jacobian is a function of position in the cell and must be evaluated at a particular \((r,s,t)\) coordinate value.
8.5 Topological operations

- Topological operations include cell operations and dataset operations. In other words, topological operations provide information about the topology of a cell or dataset. Examples of these operations include obtaining the topological dimension of a cell, or accessing neighboring cells that share common edges or faces.

- We might use these operations to decide whether to render a cell or to propagate particles through a flow field.

- Manifold topology describes a region surrounding a point that is topologically connected. A region around the point is topologically equivalent to a small disk in two-dimension or a ball in three-dimension.

- Topology that is not manifold is termed nonmanifold.
If the local neighborhood around a vertex is topologically a 2D disk (i.e., a small disk can be placed on the surface without tearing or overlapping), then the surface is manifold at that vertex. In 2D, if every edge of a two-dimensional cell is used by exactly one other cell, then the surface is locally manifold. In three dimensions, if every face of a 3D cell is used by exactly one other cell, then the region is locally manifold.

0D point 1D line 2D triangle 3D tetrahedron

A simplex of dimension n is the convex region defined by a set of n+1 independent points. A vertex, line, triangle, and tetrahedron are simplices of dimension 0, 1, 2, 3, respectively.
Cell operations

- Cell operation returns information about the topology of a cell, the topology order of the cell or the topology of the cell boundary. Given a cell \( C_i \) of topological dimension \( d \), the cell is composed of boundary cells of topological order \( d-1 \) and lower. For example, a tetrahedron is composed of four two-dimensional triangles, six one-dimensional edges, and four zero-dimensional vertices. Cell operation returns the number of boundary cells as well as the ordered list of points that define each bounding cell.

- Cell operation returns the closest boundary cell of dimension \( d-1 \) given the parametric coordinates of the cell.

- Cell operation can decompose a cell into simplices. Every cell can be decomposed into a collection of simplices. By operating on the simplex decomposition rather than the cell itself, we can create algorithms that are independent of cell type. In we want to intersect two datasets of different cell type, we would have to create methods to intersect every possible combination of cell without simplex decomposition. With simplex decomposition, we can create a single intersection that operates on only the limited set of simplices. The significant advantage of the approach is that as new cells are added to the visualization system, only the cell object must be implemented, and no other objects need to modified.
The closest boundary cell operation is implemented by partitioning each cell into various regions. To determine the closest boundary cell we need only to identify the parametric region that the point lies in, and then return the appropriate boundary cell. This example shows how to determine the closest boundary of a quadrilateral cell.

Edge 0: \( r > s \) and \( r < 1 - s \)
Edge 1: \( r > s \) and \( r > 1 - s \)
Edge 2: \( r < s \) and \( r > 1 - s \)
Edge 3: \( r < s \) and \( r < 1 - s \)
Dataset operations

- Dataset operation returns information about the topology of a dataset or topological information about the adjacency of cells, such as determining the neighbors of a cell or returning a list of all cells that use a particular point.
- A neighbor of a particular cell $C_i$ is simply a cell that shares one or more points in common with $C_i$. A vertex neighbor is a neighbor that shares one or more vertices. An edge neighbor is a neighbor that shares one or more edges. A face neighbor is a cell that shares vertices that defines one of the faces of the cell. A face neighbor is also an edge neighbor, and an edge neighbor is also a vertex neighbor.
The adjacency operators are simple set operations. For a particular cell $C_i$ defined by points

$$C_i = \{p_1, p_2, ..., p_n\} = P$$

and a point list $\overline{P} = (\overline{p_1}, \overline{p_2}, ..., \overline{p_n})$ with $\overline{P} \subset P$ where $\overline{p}$ typically corresponds to the points defining a boundary cell of $C_i$; the neighbors of $C_i$ are the adjacency set $A(C_i, \overline{P})$.

The adjacency set of $C_i$ is simply the intersection of the use sets for each point, excluding The cell $C_i$.

$$A(C_i, \overline{P}) = \left( \bigcap_{i=1}^{n} U(\overline{P}_i) \right) - C_i$$

The adjacency set represents a variety of useful information. In a manifold object represented by a polyhedra, for example, each polygon must have exactly one edge neighbor for each of its edges. Edges that have no neighbors are boundary edges; edges that have more than one edge neighbor represent nonmanifold topology. Datasets that consist of three-dimensional cells (e.g., unstructured grids) are topologically consistent only if, for each cell, there is exactly one face neighbor for each face. Faces that have no neighbors are on the boundary of the dataset. More than one face neighbor implies that the neighbors are self-intersecting (in 3D space).
8.6 Searching

Searching is an operation to find the cell containing a specified point p, or to locate cells or points in a region sounding p. Searching operations include streamline generation to find the starting location within a cell, probing to interpolate data values from the containing cell, and collision detection to evaluate cells for intersection.

We have two options when applying spatial search structures, inserting points or cells into the search structure. Inserting cells may waste memory resources and extra processing time. Inserting points allows us to search for both points and cells. Cells can be found by using p to index into the appropriate bucket. The closest point(s) pi to p are then located. Using the topological adjacency operator to retrieve the cells using point pi, we can then search these cells for the cell containing p. This method could have potential problem, since the closest points may not be used by the cells containing p. (a) points are associated with appropriate bucket. Point p is used to index into bucket, and closest point(s) pi is found. Cells using pi are evaluated for the cell containing p. (b) sometimes closest points pi are not used by cells containing p.
8.7 Cell/line intersection

- Intersection of a line with a cell can be used to interactively select a cell from the rendering window, to perform ray-tracing for rendering, or to geometrically query data. In VTK, each cell must be capable of intersecting itself against a line.

- **Vertex**: Project point onto ray, distance to line must be within tolerance, \( t \) must lie between [0,1].

- **Line**: 3D line intersection, distance between lines must be within tolerance, \( s,t \) must lie between [0,1].

- **Triangle**: Line/plane intersection, intersection point must lie in triangle, \( t \) must lie between [0,1].
Quadrilateral
line/plane intersection
intersection point must lie in quadrilateral
t must lie between [0,1]

Pixel
line/plane intersection
intersection point must lie in pixel (uses efficient in/out test)
t must lie between [0,1]

Polygon
Line/plane intersection
Intersection point must lie in polygon (uses ray tracing
For polygon in/out)
T must lie between [0,1]

Intersection point
(minimum t value)

Tetrahedron
intersect each (triangle) face
t must lie between [0,1]

Hexahedron
intersect each (quadrilateral) face
since face may be non-planar, project previous result onto hexahedron surface
t must lie between [0,1]

Voxel
Intersect each (pixel) face
T must lie between [0,1]
There is a close correspondence between scalar data and colors. An image is a regular, two-dimensional array of points. The points define pixels, which in turn form a two-dimensional structured points dataset. Images are frequently stored as a pair of dimensions along with data values. The data values may be one of black and white (e.g., bitmap), grayscale, or color ( pixmap). Bitmaps and grayscale images can be directly cast into the form of single-values scalar data. Pixmaps consist of three values per pixel of red, green, and blue, sometimes alpha opacity. Pixmaps cannot be directly cast into scalar form.

To accommodate color data, special types of scalar objects need to be created, such that a request for data at a particular point must return a single scalar value. A mapping from RGB or RGBA color coordinates to a single scalar value is required.

\[
Y (\text{luminance}) = 0.30 \, R + 0.59 \, G + 0.11 \, B
\]

\[
Y (\text{luminance}) = A(0.30 \, R + 0.59 \, G + 0.11 \, B)
\]

This mapping is very simple, where R --- red, G --- green, B --- blue, A --- transparency
8.9 Special techniques for structured points

- A significant attraction of the structured points dataset is the speed and simplicity of computation.
- Coordinate transformation
- Derivative computation
- Topology
- Searching
Structured point coordinate transformation: given a point \( p \) we can find the structured coordinates by performing three division operations. Taking the integer of floor function Yields the structured coordinates. Taking the fractional part of the result yields the parametric coordinates of the cell. The point ids and cell ids can be found as follows:

\[
p_{id} = i_p + j_p n_x + k_p n_x n_y
\]

\[
cell_{id} = i_p + j_p (n_x - 1) + k_p (n_x - 1)(n_y - 1)
\]
Using finite difference to compute derivatives on structured points.

Derivative computation

\[ g_x = \frac{d(x_0 + \Delta x, y_0, z_0) - d(x_0 - \Delta x, y_0, z_0)}{2\Delta x} \]

\[ g_y = \frac{d(x_0, y_0 + \Delta y, z_0) - d(x_0, y_0 - \Delta y, z_0)}{2\Delta y} \]

\[ g_z = \frac{d(x_0, y_0, z_0 + \Delta z) - d(x_0, y_0, z_0 - \Delta z)}{2\Delta z} \]

This central difference. At the boundary of the dataset, one-sided differences is preferred. We can use these equations to compute the derivatives at each cell point, and then use the cell interpolation functions to compute the derivative at the point inside the cell.
For structured datasets, it is possible to determine vertex, edge, or face neighbors using simple constant time operations, given cell ids. We do this by combining the operations of division and modulo.

\[
\begin{align*}
    i &= \text{id modulo } (n_x - 1) \\
    j &= \text{id/}(n_x - 1) \text{ modulo } (n_y - 1) \\
    k &= \text{id/}((n_x - 1)(n_y - 1))
\end{align*}
\]

Face neighbors are determined by incrementing one of the \( i, j, \) or \( k \) indices. Edge neighbors are determined by incrementing any two indices, while vertex neighbors are found by incrementing all three indices.

**Topology**

\[
0 \leq i < (n_x - 1)
\]

\[
0 \leq j < (n_y - 1)
\]

\[
0 \leq k < (n_z - 1)
\]
Given a point \( p(x,y,z) \) we can determine the cell containing \( p \) by using the equations given below to generate the structured coordinates \((i,j,k)\), which can then be converted to cell id (i.e., dataset coordinates)

\[
\begin{align*}
    i &= \text{floor}((x-x_0)/(x_1-x_0)) \\
    j &= \text{floor}((y-y_0)/(y_1-y_0)) \\
    k &= \text{floor}((z-z_0)/(z_1-z_0))
\end{align*}
\]

\[
    cell_{id} = i_p + j_p (n_x - 1) + k_p (n_x - 1)(n_y - 1)
\]

To find the closest point \( p \), we compute the structured coordinates by rounding to the nearest integer value,

\[
\begin{align*}
    i &= \text{int}((x-x_0)/(x_1-x_0)) \\
    j &= \text{int}((y-y_0)/(y_1-y_0))
\end{align*}
\]

searching \( k = \text{int}((z-z_0)/(z_1-z_0)) \)
For unstructured dataset, the complexity of traversing the cell array and retrieving adjacency information is at least $O(n^2)$, since a time complexity of $O(n)$ is required to implement any operation to retrieve vertex, edge, or face neighbors, due to the downward hierarchy shown in (b). We can improve this by allowing upward hierarchy shown in (a). The solution to this problem is to extend the unstructured data structure with cell links. The cell links array is a list of lists of cells that use each point and corresponds to the upward links of figure (c). The cell links array transforms the hierarchical structure of Figure 5.10 into a ring structure. Cells reference their composing points, and points in turn reference the cells that use them.
Figure 5.10 The vtkUnstructuredGrid data structure
Complete unstructured data representation including link lists. There are $m$ cells and $n$ points. The $n$ structures in the link list are lists of cells that use each vertex. Each link list is variable in length.

In vtk, the unstructured data structure is implemented by the four classes, `vtkPoints`, `vtkCellArray`, `vtkCellTypes`, and `vtkCellLinks`. At a minimum, the points and cells are represented using `vtkPoints` and `vtkCellArray`. If random access or extra type information is required, then the object `vtkCellTypes` is used. An instance of `vtkCellLinks` is created if adjacency information is required.
8.10 Implementation in vtk

\begin{verbatim}
type = GetDataObjectType()
Return the type of dataset, \textit{vtkPolyData, vtkImageData, vtkStructuredGrid, vtkRectilinearGrid, vtkUnstructuredGrid}.

numPoints = GetNumberOfPoints()
Return the number of points in the dataset.

numCells = GetNumberOfCells()
Return the number of cells in the dataset.

GetPoint(ptId, x)
Given a point id, return the \((x, y, z)\) coordinates of the point.

cell = GetCell(cellId)
Given a cell id, return a pointer to a cell object.

type = GetCellType(cellId)
Return the type of the cell given by cell id.

GetCellTypes(types)
Return a list of types of cells that compose the dataset.

cells = GetPointCells(ptId)
Given a point id, return the cells that use this point.
\end{verbatim}
GetCellPoints(cellId, ptIds): Given a cell id, return the point ids defining the cell.

GetCellNeighbors(cellId, ptId, s, neighbors): Given a cell id and a list of points composing a boundary face of the cell, return the neighbors of that cell sharing the points.

cellId = FindCell(x, cell, cellId, tol2, subId, pcords, weights): Given a coordinate value x, an initial search cell defined by cell and cellId, and a tolerance measure (squared), return the cell id and sub-id of the cell containing the point and its interpolation function weights. The initial search cell is used to speed up the search process when the position x is known to be near the cell. If no cell is found, cellId<0 is returned.

pointData = GetPointData(): Return a pointer to the object maintaining point attribute data. This includes scalars, vectors, normals, tensors, and texture coordinates, as well as any other data arrays that the field carries.

cellData = GetCellData(): Return a pointer to the object maintaining the cell attribute data. This includes scalars, vectors, normals, tensors, and texture coordinates, as well as any other data arrays that the field carries.

Bounds = GetBounds(): Get the bounding box of the dataset.

Length = GetLength(): Return the length of the diagonal of the bounding box of the dataset.

Center = GetCenter(): Get the center of the bounding box of the dataset.

Range = GetScalarRange(): A convenience method to return the (minimum, maximum) range of the scalar attribute data associated with the dataset.

dataSet = NewInstance(): Make a copy of the current dataset, a “virtual” constructor.

CopyStructure(dataSet): Update the current structure definition with the supplied dataset.

**Dataset abstraction**
type = GetCellType(): return the type of the cell: one of the twelve vtk cell types or empty cell type.
dim = GetCellDimension(): return the topological definition of the cell.
order = GetInterpolationOrder(): return the degree of the interpolating polynomial of the cell. The twelve cell types are all degree 1.
numberPoints = GetNumberOfPoints(): return the number of points that define the cell.
points = GetPoints(): return a list of point ids defining the cell.
numberEdges = GetNumberOfEdges(): return the number of edges in the cell.
edge = GetEdge(i): return a pointer to a cell that represents an edge of the cell given an edge id.
numberFaces = GetNumberOfFaces(): return the number of faces is a cell.
face = GetFace(i): return a pointer to a cell that represents a face of the cell given a face id.
inOutStatus = CellBoundary(subId, pcoords, pointIds)
   given a cell subid and parametric coordinates, return a list of point ids that define the closest boundary face of the cell. Also return whether the point is actually in the cell.
inOutStatus = EvaluatePosition(x, closestPoint, subId, pcoords, weights, dist2)
   given a point coordinate x, return the sub-id, parametric coordinates, and interpolation weights of the cell if x is inside the cell. The position closestPoint is the closest point on the cell to x (may be the same) and dist2 is the squared distance between them. It returns an inOutStatus indicating whether x is topologically inside or outside the cell. The point may satisfy parametric coordinate conditions but may lie off the surface of the cell, i.e., above a polygon. We use both inOutStatus and dist2 to determine whether point is both topologically and geometrically in the cell.
evaluateLocation(subId, pcoords, x, weights): given a point location (i.e., sub-id and parametric coordinates), return the position x of the point and the interpolation weights.

Cell abstraction
Contour(value, cellScalars, locator, verts, lines, polys, inputPointData, outputPointData): given a contour value and scalar values, at the cell points, generate contour primitives (vertices, lines, or polygons with associated points and attribute data values). The points are placed in a locator object which merges coincident points, and the attribute data values are interpolated from the inputPointData to the outputPointData.

Clip(value, cellScalars, locator, cells, inputPointData, outputPointData, insideOut): given a contour value and scalar values at the cell points, clip the cell to generate new cells of the same topological dimension as the original cell. The points are placed in a locator object which merges coincident points, and the attribute data values are interpolated (or copied) from the inputPointData to the outputPointData. The clipped cells are placed in the cells list.

Derivatives(subId, pcoords, values, dim, derivs): given a cell location (i.e., subid and parametric coordinates) and data values at the cell points, return dim*3 derivatives (i.e., corresponds to the x, y, and z directions times dimension of data)

inOutStatus = IntersectWithLine(p1, p2, to1, t, x, pcoords, subId): given a finite line defined by the two points p1 and p2 and an intersection tolerance, return the point of intersection x. The parametric coordinate t along the line and cell location at the point of intersection is also returned. Returns a nonzero if intersection occurs.

Triangulate (index, ptIds, points): decompose the cell into simplices of dimension equal to the topological cell dimension. The index is an integer that controls the triangulation if more than one triangulation is possible. The simplices are defined by an ordered list of point ids and their corresponding coordinates.

bounds = Getbounds(): Return the bounding box of the cell.

Cell abstraction
**CopyScalarsOn() / CopyScalarsOff()**: turn on/off boolean flag controlling copying of scalar data from input to output of filter.

**CopyVectorsOn() / CopyVectorsOff()**: turn on/off boolean flag controlling copying of vector data from input to output of filter.

**CopyNormalsOn() / CopyNormalsOff()**: turn on/off boolean flag controlling copying of normal data from input to output of filter.

**CopyTensorsOn() / CopyTensorsOff()**: turn on/off boolean flag controlling copying of tensor data from input to output of filter.

**CopyTextureCoordsOn() / CopyTextureCoordsOff()**: turn on/off boolean flag controlling copying of texture coordinates data from input to output of filter.

**CopyAllOn() / CopyAllOff**: turn on/off all boolean flags controlling copying of all attribute data from input to output of filter.

**PassData(pointData)**: transfer all point attribute data (pointData) to the output according to the copy flags listed previously.

**CopyAllocate(pointData)**: Initialize and allocate storage for point-by-point copy process.

**CopyData(pointData, fromId, toId)**: given point data and a specific point id, copy the point attribute data(pointData) to the output point.

**InterpolateAllocate(pointData)**: initialize and allocate storage for point-by-point interpolation process.

**InterpolatePoint(pointData, toId, ptIds, weights)**: Given input point data and a list of points and their interpolation weights, interpolate data to the specified output point.

**InterpolateEdge(pointData, toId, p1, p2, t)**: from an edge defined by the two points p1 and p2, interpolate the pointData at the edge parametric coordinate t and copy the interpolated attribute data to the output point ptId.

**Point and cell attribute abstraction**
**NullPoint(int ptId)**: set the data value(s) of the specified output point id to a null value.

**SetScalars() / Get Scalars**: set/return scalar data. The GetScalars() method may return a Null value, in which case the scalars are not defined.

**SetVectors() / GetVectors()**: set/return vector data. The GetVectors() method may return a NULL value, in which case the vectors are not defined.

**SetNormals() / GetNormals()**: set/return normal data. The GetNormals may return a NULL value, in which case the normals are not defined.

**SetTensors() / GetTensors()**: set/return tensor data. The GetTensor() may return a NULL value, in which case the tensors are not defined.

**SetTextureCoords() / GetTextureCoords()**: set/return texture coordinate data. The GetTextureCoords() may return a NULL value, in which case the texture coordinates are not defined.

Point and cell attribute abstraction
In vtk, **vtkPointLocator()** class is designed for searching dataset points. vtkPointLocator is implemented as a regular grid of buckets. The number of buckets can be user-specified, or automatically computed based on the number of dataset points. The closest point to a point \( p \) can be determined using vtkPointLocator. In the first step, the bucket containing \( p \) is found using the appropriate insertion. Next, the list of points in this bucket is searched to determine the closest point. Since points in the neighboring buckets may be closer, and final search of neighboring buckets is necessary. In the above, initial search in bucket results in point \( a \), and search must extend beyond local bucket as a function of search radius \( R \), resulting in point \( b \). Once all neighbors within this distance are searched, the closest point is returned.
Cell List Structure (Union)

if parent octant:
  IN / OUT flag
if terminal octant:
  list of cells

level of octree, \( l \)

number of terminal octants, \( n_T \)
\[ n_T = 8^l \]

number of octants, \( n_O \)
\[ n_O = \sum_{i=0}^{l} 8^i \]

number of parents, \( n_P \)
\[ n_P = n_O - n_T \]

Structure of spatial search structure vtkCellLocator. The data structure represents a uniformly subdivided octree.
Vtk provides four classes to perform actor, point, cell, and world point picking: vtkPicker, vtkPointPicker, vtkCellPicker, and vtkWorldPointPicker.

**vtkPicker** intersects a ray defined from camera position to a screen (i.e., pixel coordinate) against the bounding box of all pickable and nontransparent actors. (An actor is pickable if its pickable instance variable is true). The result of the vtkPicker pick operation is to return a list of the actors whose bounding box is intersected. The actor closest to the camera position is also returned.

**vtkPointPicker** intersects the ray against the points defining each actor, and returns the point coordinate closest to the camera position, as well as the actor that the point belongs to.

**vtkCellPicker** intersects the ray with the cells defining each actor, and returns the point of intersection, as well as the actor that the cell belongs to.

**vtkWorldPointPicker** returns the \((x, y, z)\) coordinate value of a pick in the rendering window. To determine the information, it combines the display \((x, y)\) values with the z-buffer depth values.
(a) vtkPicker

vtkCellPicker

vtkWorldPointPicker
sphereActor->SetPosition(picker->GetPickPosition());

if(picker->GetPointId() >=0)
{
    cout<<“point id: “<< picker->GetPointId() << “\n”;
    cellsActor->VisibilityOn();
    plateActor->VisibilityOff();
    cells->Initialize();
    cells->Allocate(100);
    cells->SetPoints(plateOutput->GetPoints());

    plateOutput->GetPointCells(picker->GetPointId(), cellIds);
    for(i=0; i<cellIds.GetNumberOfIds(); i++)
    {
        cellId = cellIds.GetId(i);
        plateOutput->GetCellPoints(cellId, ptIds);
        cells->InsertNextCell(plateOutput->GetCellType(cellId), ptIds
    }
}
else
{
    cellsActor->VisibilityOff();
    plateActor->VisibilityOn();
}
renWin->Render();
// Loop over all input points, interpolating source data;

for (ptId=0; ptId<numPts; ptId++)
{
    // get the xyz coordinate of the point in the input dataset;
    x = input->GetPoint(ptId);

    // Find the cell that contains xyz and get it
    cell = source->FindAndGetCell(x, NULL, -1, to12, subId, pcoords, weights
    if(cell)
    {
        // Interpolate the point data;
        outPD->InterpolatePoint(pd, ptId, &(cell->PointIds), weights);
    }
    else
    {
        outPD->NullPoint(ptId);
    }
}