Advanced Data Representations

Advanced Data Representations

Overview

This chapter is basically devoted to searching in different type of grids. In particular, how to find the cell contains a given point or how to intersect cells with lines. In addition, better approaches are discussed for applications, such as streamlines.

Searching

Finding the cell containing a point p

To find the cell containing p, we can use the following naïve search procedure. Traverse all cells in the dataset, finding the one (if any) that contains p. To determine whether a cell contains a point, the cell interpolation functions are evaluated for the parametric coordinates (r,s,t). If these coordinates lie within the cell, then p lies in the cell. The basic assumption here is that cells do not overlap, so that at most a single cell contains the given point p.

Searching

These naïve procedures are unacceptable for all but the smallest data sets, since they are of order $O(n)$, where n is the number of cells or points. To improve the performance of searching, we need to introduce supplemental data structures to support spatial searching. Such structures are well-known (see Computer Graphics II) and include octress or kd-trees.

The basic idea behind these spatial search structures is that the search space is subdivided into smaller parts, or buckets. Each bucket contains a list of the points or cells that lie within it. Buckets are organized in structured fashion so that constant or logarithmic time access to any bucket is possible.

Searching (continued)

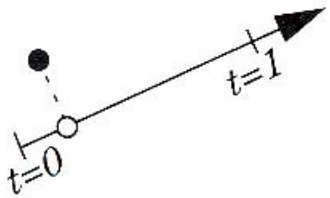
Cell / line intersection

An important geometric operation is intersection of a line with a cell. This operation can be used to interactively select a cell from the rendering window, to perform ray-casting for rendering, or to geometrically query data.

Often, curves are approximated by line segments (i.e. a linear spline) so that intersection with a line can be used for computing the intersection with a curve.

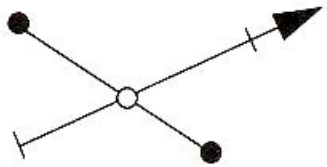
In VTK, every cell must be capable of intersecting itself against a line. The following approaches are used for each cell type.

Searching (continued)



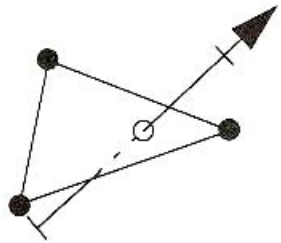
Vertex

- project point onto ray
- distance to line must be within tolerance
- t must lie between [0,1]



Line

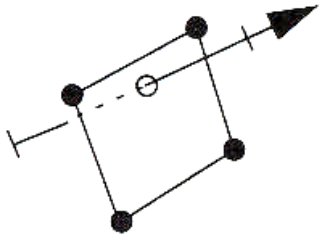
- 3D line intersection
- distance between lines must be within tolerance
- s,t must lie between [0,1]



Triangle

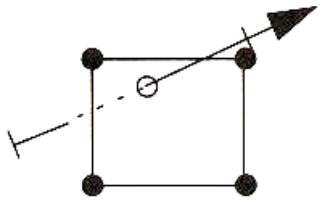
- line/plane intersection
- intersection point must lie in triangle
- t must lie between [0,1]

Searching (continued)



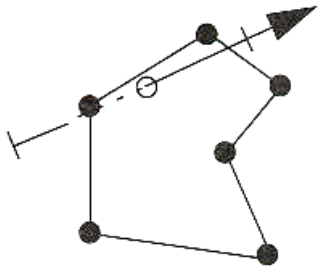
Quadrilateral

- line/plane intersection
- intersection point must lie in quadrilateral
- t must lie between [0,1]



Pixel

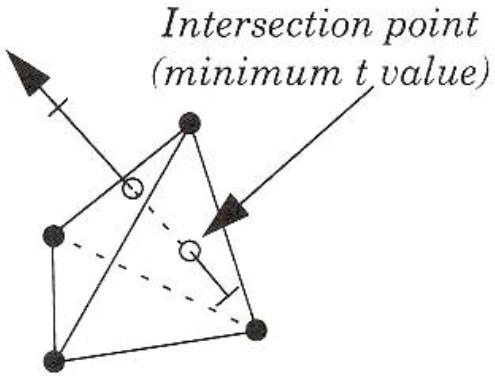
- line/plane intersection
- intersection point must lie in pixel (uses efficient in/out test)
- t must lie between [0,1]



Polygon

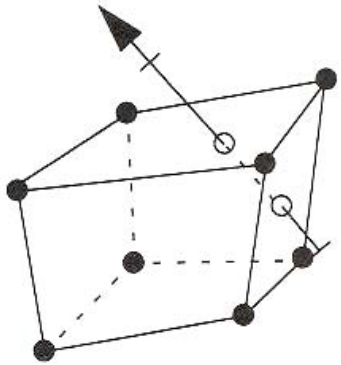
- line/plane intersection
- intersection point must lie in polygon (uses ray casting for polygon in/out)
- t must lie between [0,1]

Searching (continued)



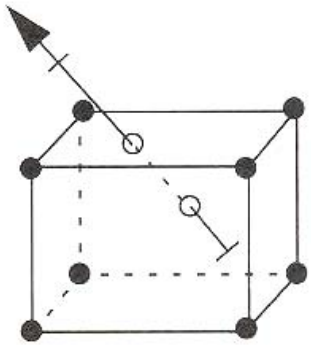
Tetrahedron

- 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]

Searching (continued)

Special search techniques for structured points

Searching in structured point data sets is particularly easy due to the equidistant arrangements of grid points. In order to find the cell containing the point p = (x, y, z) we can exploit this:

$$i = int\ ((x\text{-}x_0)/(x_1\text{-}x_0))$$

$$j = int\ ((y\text{-}y_0)/(y_1\text{-}y_0))$$

$$k = int\ ((z\text{-}z_0)/(z_1\text{-}y_0))$$

Thus, we get the cell id by simply rounding to the nearest integer value.

Searching (continued)

Topological operations

Many visualization algorithms work more efficiently if more information about the grid structure is available. For example, a streamline integrator that traverses one cell after another. Once a cell is left by the streamline the integration process continues at the next, neighboring cell. One way would be to use the search data structure to identify this next cell. However, it is much more efficient if the cells know their neighbors directly.

This connectivity information within the cells is called the *topology* of the grid. Hence, operations that provide this neighborhood information are called *topological operations*.

Searching (continued)

Searching with VTK

The Visualization Toolkit provides two classes to perform searches for data set points and cells. These are `vtkPointLocator` and `vtkCellLocator`. (Both of these calsses are subclasses of `vtkLocator`, which is an abstract base class for spatial search objects). `vtkPointlocator` is used to search for points and, if used with the topological data set operator `GetPointCells ()`, to search for cells as well. `vtkCellLocator` is used to search for cells.

Searching (continued)

vtkPointLocator

`vtkPointLocator` is implemented as a regular grid of buckets, i.e. same topology and geometry as a structured point set. The number of buckets can be user-specified, or more conveniently, automatically computed based on the number of data set points. On average, `vtkPointLocator` provides constant time access to points, However, in cases where the point distribution is not uniform, the number of points in a bucket may vary widely, giving O(n) worst-case behavior. In practice, this is rarely a problem, but adaptive spatial search structures (e.g. an octree) may sometimes be a better choice.

Searching (continued)

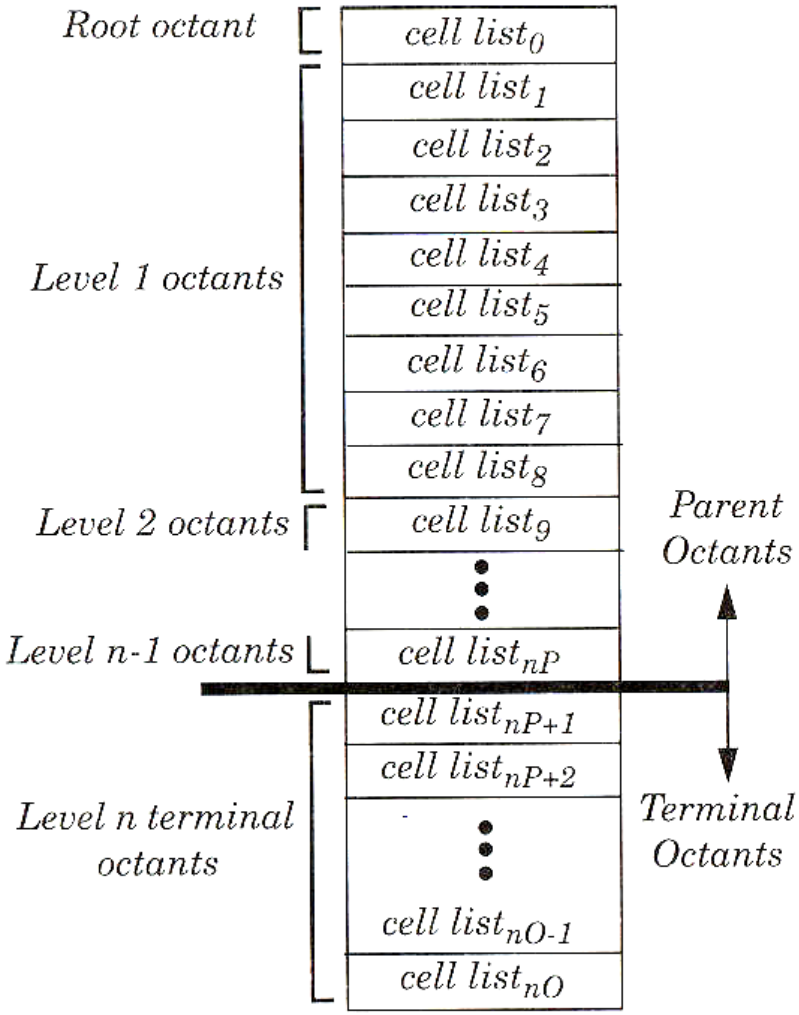
vtkCellLocator

`vtkCellLocator` is implemented as a uniformly subdivided octree with some peculiar characteristics. Conventional octree representations use upward parent and downward children pointers to track parent and children octants. Besides the required list of entities (i.e. points or cells) in each octant, additional information about octant level, center, and size may also be maintained. This results in a flexible structure with significant overhead. The overhead is the memory resources to maintain pointers, plus the cost to allocate and delete memory.

Searching (continued)

In contrast, `vtkCellLocator` uses a single array to represent the octree. The array is divided into two parts. The first part contains a list of parent octants, ordered according to level and octant child number. In the second part are the terminal, or leaf octants. The terminal octants are ordered on a regular array of buckets, just the same as `vtkLocator`. The terminal octants contain a list of entities inside the octant. The parent octants maintain a value indicating whether the octant is empty, or whether something is inside it. Because the octree is uniformly subdivided, parent-child relationships, as well as octant locations, can be computed quickly using the simple division operations.

Searching (continued)



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

$$n_P = n_O - n_T$$

Searching (continued)

The advantage of this structure is that memory can be allocated and deleted quickly. In addition, insertion into the octree is simpler than conventional octrees, The parent octants provide quick culling capabilities, since their status (empty or nonempty) allows us to stop certain types of search operations. On the downside, because the octree is uniformly subdivided, this structure is wasteful of memory resources if the data is non-uniformly distributed.

In this case, a special locator that is more optimized for a specific data set can be implemented and used.