Data Representation

Overview

This chapter will introduce you to data representations used for Scientific Visualization.

We will discuss different grid structures and ways to represent data using these grid structures. In addition, you will learn more about the different VTK file formats.

Finally, we will briefly discuss interpolation strategies for deriving data values at locations where no data value is provided directly by the data representation and discuss some advanced data representations.

Continuous Data

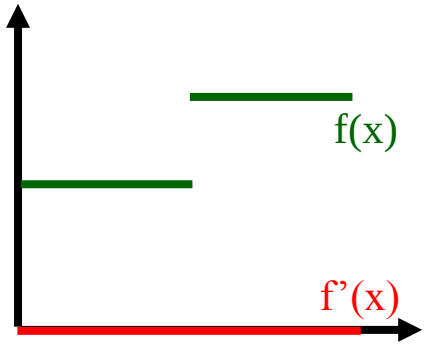
Mathematically, continuous data can be modeled as a function

$$f: D \rightarrow C$$

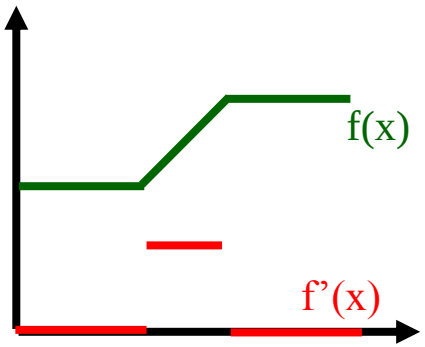
Where $D \subset R^d$ is the function domain and $C \subset R^c$ is the function codomain, respectively. In a related terminology, f is called a d-dimensional, or d-variate, c-valued function. In other words, if we write $f(x)=y$, where $x \in D$ and $y \in C$, this actually means $f(x_1,...,x_d)=(y_1,...,y_c)$. In visualization applications, f or its sampled counterpart is sometimes called a **field**.

Continuous Data (continued)

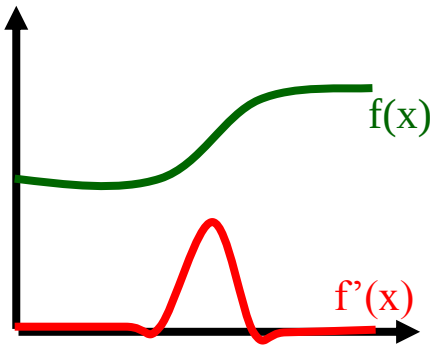
The function values are usually called dataset attributes. The dimensionality c of the function's codomain C is also called the attribute dimension. The attribute dimension c typically ranges from 1 to 4 (it can be significantly higher as we will see in Information Visualization).



Discontinuous function



C^0 continuous function



higher order C^k continuous function

Sampled Data

Most of the time, visualization data is not provided in a continuous form. Two operations relate sampled data and continuous data:

- **sampling**: produce sampled data from continuous form
- **reconstruction**: recover/approximate continuous data

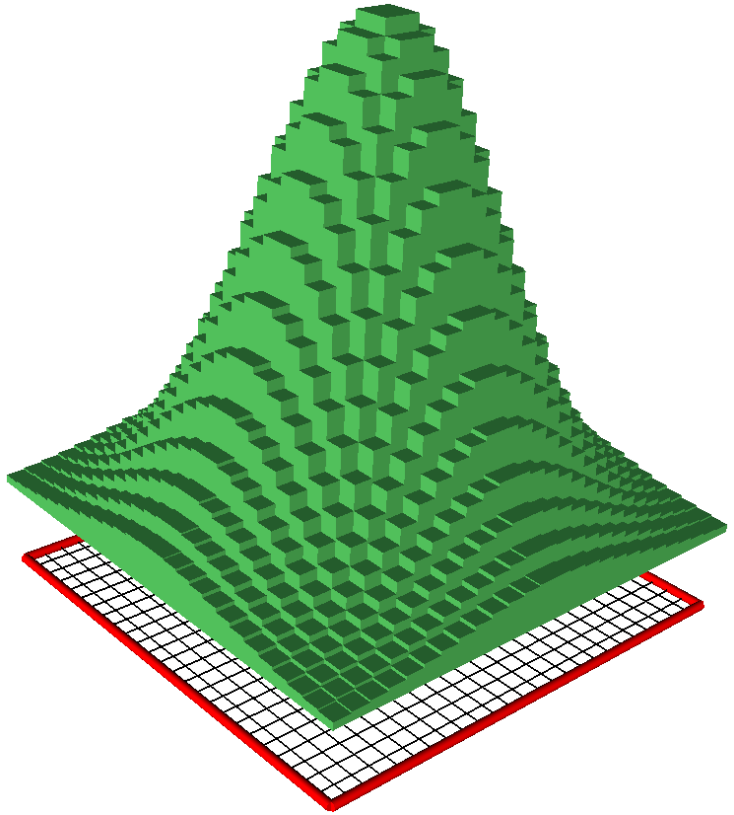


Image courtesy of Alexandru Telea

Sampled Data (continued)

How can we store scientific data?

The data structure should:

- Cover an area or volume with a set of data points
- Allow easy and fast access to the data
- Re-usable, i.e. not applicable to one example only
- Be flexible

Sampled Data (continued)

Inspired by finite element analysis, often different kinds of cells are used where the data points are located at the vertices of these cells or the entire cell.

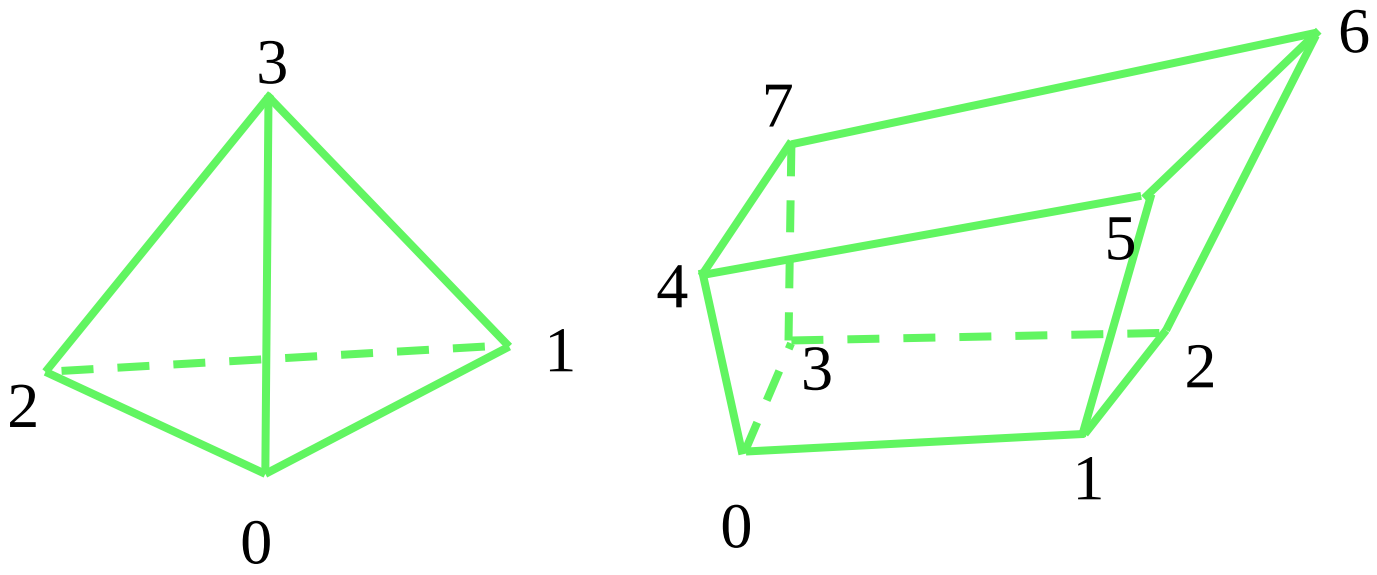
Cell Topology

Cells specify the topology of the covered area. Different cell types can occur:

- Polygon
- Tetrahedron
- Hexahedron
- Triangle
- Line
- etc.

Cells

- Cell is defined by an ordered list of points
  - Triangle, quadrilateral points specified counter clockwise
  - Others as shown
- Data is attached to entire cell or vertices of the cell



VTK Dataset Types

VTK provides several different grid data structures:

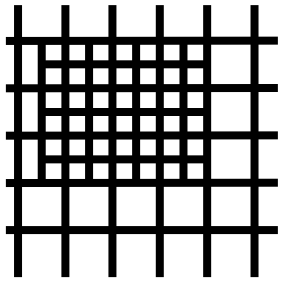
- vtkImageData
- vtkStructuredPoints
- vtkRectilinearGrid
- vtkStructuredGrid
- vtkPolyData
- vtkUnstructuredGrid

Datasets

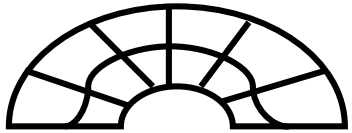
Organizing structure plus attributes

- Structured points

- Rectilinear Grid

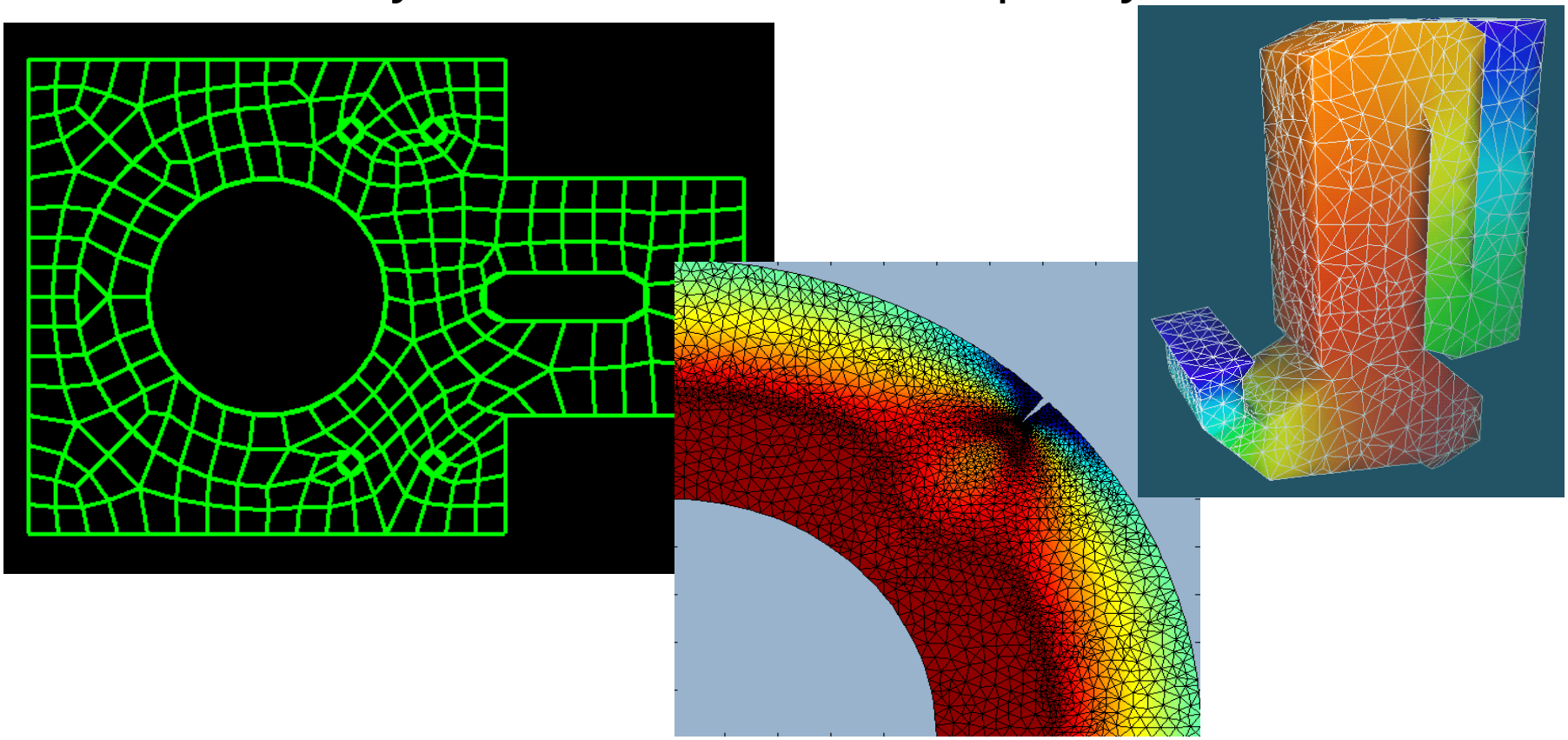


- Structured Grid

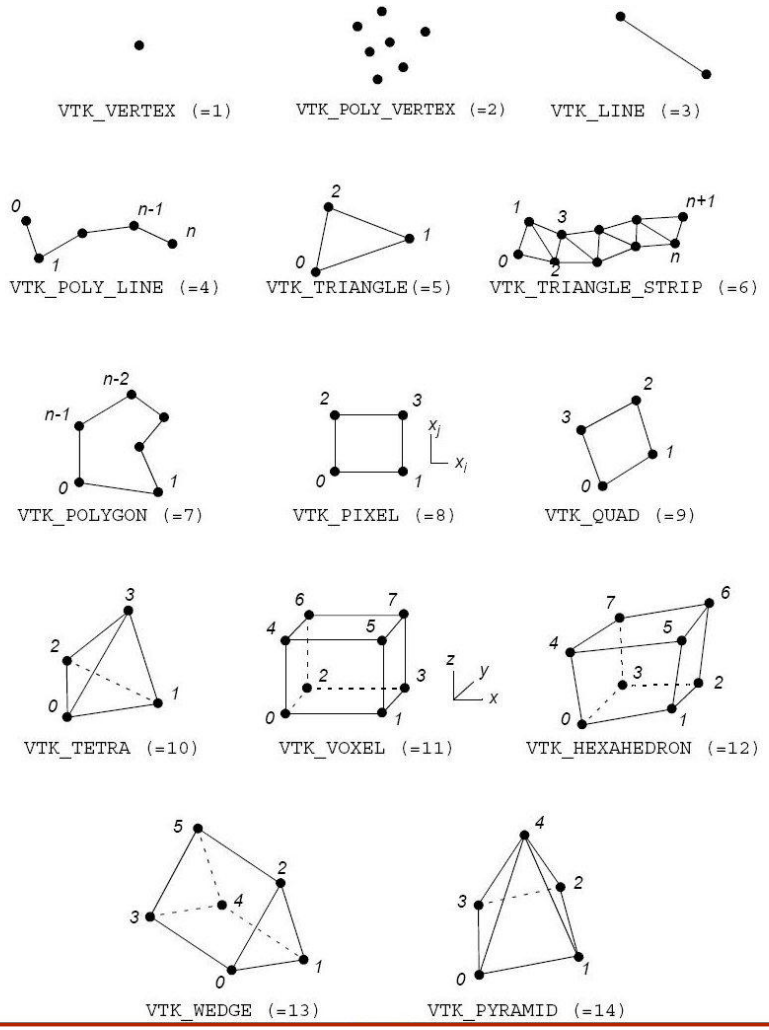


Unstructured Grid

A collection of vertices, edges, faces and cells whose connectivity information must be explicitly stored



Available Cell Types in VTK



Data Attributes

Data attributes are assigned to points or cells

- Scalars
- Vector
  - Magnitude and direction
- Normal
  - a vector of magnitude 1
  - Used for lighting
- Texture Coordinate
  - Mapping data points into a texture space
- Tensor

VTK File Format

There are two different styles of file formats available in VTK. The simplest are the legacy, serial formats that are easy to read and write either by hand or programmatically. However, these formats are less flexible than the XML based file formats described later in this section. The XML formats support random access, parallel I/O, and portable data compression and are preferred to the serial VTK file formats whenever possible.

We will focus on the ASCII legacy file format as it is easier to parse and read in our own software.

VTK File Format (continued)

The legacy VTK file formats consist of five basic parts:

- The first part is the file version and identifier. This part contains the single line: # vtk DataFile Version x.x.
- The second part is the header. The header consists of a character string terminated by end-of-line character \n. The header is 256 characters maximum.
- The next part is the file format. The file format describes the type of file, either ASCII or binary. On this line the single word ASCII or BINARY must appear.

VTK File Format (continued)

- The fourth part is the dataset structure. The geometry part describes the geometry and topology of the dataset. This part begins with a line containing the keyword DATASET followed by a keyword describing the type of dataset. Then, depending upon the type of dataset, other keyword/data combinations define the actual data.
- The final part describes the dataset attributes. This part begins with the keywords POINT_DATA or CELL_DATA, followed by an integer number specifying the number of points or cells, respectively. (It doesn't matter whether POINT_DATA or CELL_DATA comes first.) Other keyword/data combinations then define the actual dataset attribute values (i.e., scalars, vectors, tensors, normals, texture coordinates, or field data).

VTK File Format (continued)

Example

vtk DataFile Version 2.0

Really cool data

ASCII | BINARY

DATASET *type*

...

POINT_DATA *n*

...

CELL_DATA *n*

...

Header

Title

Type

Geometry/topology

Dataset attributes

VTK File Format (continued)

The type of the dataset can be one of the following:

STRUCTURED_POINTS

STRUCTURED_GRID

UNSTRUCTURED_GRID

POLYDATA

RECTILINEAR_GRID

FIELD

The number of data items *n* of each type must match the number of points or cells in the dataset. (If *type* is FIELD, point and cell data should be omitted.)

VTK File Format (continued)

Before describing the data file formats please note the following.

- *dataType* is one of the types `bit`, `unsigned_char`, `char`, `unsigned_short`, `short`, `unsigned_int`, `int`, `unsigned_long`, `long`, `float`, or `double`. These keywords are used to describe the form of the data, both for reading from file, as well as constructing the appropriate internal objects. Not all data types are supported for all classes.
- Indices are 0-offset. Thus the first point is point id 0.
- If both the data attribute and geometry/topology part are present in the file, then the number of data values defined in the data attribute part must exactly match the number of points or cells defined in the geometry/topology part.
- Cell types and indices are of type int.
- The geometry/topology description must occur prior to the data attribute description.

VTK File Format (continued)

Structured Points

The file format supports 1D, 2D, and 3D structured point datasets. The dimensions *nx, ny, nz* must be greater than or equal to 1. The data spacing *sx, sy, sz* must be greater than 0. (Note: in the version 1.0 data file, spacing was referred to as “aspect ratio”. ASPECT_RATIO can still be used in version 2.0 data files, but is discouraged.)

```
DATASET STRUCTURED_POINTS
DIMENSIONS nx ny nz
ORIGIN x y z
SPACING sx sy sz
```

VTK File Format (continued)

Structured Grid

The file format supports 1D, 2D, and 3D structured grid datasets. The dimensions *nx, ny, nz* must be greater than or equal to 1. The point coordinates are defined by the data in the POINTS section. This consists of *x-y-z* data values for each point.

```
DATASET STRUCTURED_GRID
DIMENSIONS nx ny nz
POINTS n dataType
p0x p0y p0z
p1x p1y p1z
...
p(n-1)x p(n-1)y p(n-1)z
```

VTK File Format (continued)

Rectilinear Grid

A rectilinear grid defines a dataset with regular topology, and semi-regular geometry aligned along the *x-y-z* coordinate axes. The geometry is defined by three lists of monotonically increasing coordinate values, one list for each of the *x-y-z* coordinate axes. The topology is defined by specifying the grid dimensions, which must be greater than or equal to 1.

```
DATASET RECTILINEAR_GRID
DIMENSIONS nx ny nz
X_COORDINATES nx dataType
x0 x1 ... x(nx-1)
Y_COORDINATES ny dataType
y0 y1 ... y(ny-1)
Z_COORDINATES nz dataType
z0 z1 ... z(nz-1)
```

VTK File Format (continued)

Polygonal Data

The polygonal dataset consists of arbitrary combinations of surface graphics primitives vertices (and polyvertices), lines (and polylines), polygons (of various types), and triangle strips. Polygonal data is defined by the POINTS VERTICES, LINES, POLYGONS, or TRIANGLE_STRIPS sections. The POINTS definition is the same as we saw for structured grid datasets. The VERTICES, LINES, POLYGONS, or TRIANGLE_STRIPS keywords define the polygonal dataset topology. Each of these keywords requires two parameters: the number of cells *n* and the size of the cell list *size*. The cell list size is the total number of integer values required to represent the list (i.e., sum of *numPoints* and connectivity indices over each cell). None of the keywords VERTICES, LINES, POLYGONS, or TRIANGLE_STRIPS is required.

```
DATASET POLYDATA
POINTS n dataType
p0x p0y p0z
...
p(n-1)x p(n-1)y p(n-1)z
```

VTK File Format (continued)

VERTICES *n size*

$numPoints_0$, i_0, j_0, k_0, ...

$numPoints_1$, i_1, j_1, k_1, ...

...

$numPoints_{n-1}$, i_{n-1}, j_{n-1}, k_{n-1}, ...

LINES *n size*

$numPoints_0$, i_0, j_0, k_0, ...

$numPoints_1$, i_1, j_1, k_1, ...

...

$numPoints_{n-1}$, i_{n-1}, j_{n-1}, k_{n-1}, ...

VTK File Format (continued)

POLYGONS *n size*

$numPoints_0$, i_0, j_0, k_0, ...

$numPoints_1$, i_1, j_1, k_1, ...

...

$numPoints_{n-1}$, i_{n-1}, j_{n-1}, k_{n-1}, ...

TRIANGLE_STRIPS *n size*

$numPoints_0$, i_0, j_0, k_0, ...

$numPoints_1$, i_1, j_1, k_1, ...

...

$numPoints_{n-1}$, i_{n-1}, j_{n-1}, k_{n-1}, ...

VTK File Format (continued)

Unstructured Grid

The unstructured grid dataset consists of arbitrary combinations of any possible cell type. Unstructured grids are defined by points, cells, and cell types. The CELLS keyword requires two parameters: the number of cells *n* and the size of the cell list *size*. The cell list size is the total number of integer values required to represent the list (i.e., sum of *numPoints* and connectivity indices over each cell). The CELL_TYPES keyword requires a single parameter: the number of cells *n*. This value should match the value specified by the CELLS keyword. The cell types data is a single integer value per cell that specified cell type (see vtkCell.h or **Figure 2**).

```
DATASET UNSTRUCTURED_GRID
POINTS n dataType
p0x p0y p0z
p1x p1y p1z
...
p(n-1)x p(n-1)y p(n-1)z
```

VTK File Format (continued)

CELLS *n size*

$numPoints_0$, i_0, j_0, k_0, l_0, ...

$numPoints_1$, i_1, j_1, k_1, l_1, ...

$numPoints_2$, i_2, j_2, k_2, l_2, ...

...

$numPoints_{n-1}$, i_{n-1}, j_{n-1}, k_{n-1}, l_{n-1}, ...

CELL_TYPES *n*

$type_0$

$type_1$

$type_2$

...

$type_{n-1}$

VTK File Format (continued)

Field

Field data is a general format without topological and geometric structure, and without a particular dimensionality. Typically field data is associated with the points or cells of a dataset. However, if the FIELD *type* is specified as the dataset type, then a general VTK data object is defined. Use the format described in the next slides to define a field.

VTK File Format (continued)

Dataset Attribute Format

The *Visualization Toolkit* supports the following dataset attributes: scalars (one to four components), vectors, normals, texture coordinates (1D, 2D, and 3D), tensors, and field data. In addition, a lookup table using the RGBA color specification, associated with the scalar data, can be defined as well. Dataset attributes are supported for both points and cells.

Each type of attribute data has a *dataName* associated with it. This is a character string (without embedded whitespace) used to identify a particular data. The *dataName* is used by the VTK readers to extract data. As a result, more than one attribute data of the same type can be included in a file. For example, two different scalar fields defined on the dataset points, pressure and temperature, can be contained in the same file. (If the appropriate *dataName* is not specified in the VTK reader, then the first data of that type is extracted from the file.)

VTK File Format (continued)

Scalars

The variable *numComp* is optional—by default the number of components is equal to one. (The parameter *numComp* must range between (1,4) inclusive; in versions of VTK prior to vtk2.3 this parameter was not supported.)

`SCALARS` *`dataName dataType numComp`*

s_0

s_1

...

s_{n-1}

VTK File Format (continued)

Vectors

VECTORS *dataName dataType*

v_{0x} v_{0y} v_{0z}

v_{1x} v_{1y} v_{1z}

...

$v_{(n-1)x}$ $v_{(n-1)y}$ $v_{(n-1)z}$

Normals

Normals are assumed normalized.

NORMALS *dataName dataType*

n0x n0y n0z

n1x n1y n1z

...

n(n-1)x n(n-1)y n(n-1)z

VTK File Format (continued)

Texture Coordinates

Texture coordinates of 1, 2, and 3 dimensions are supported.

`TEXTURE_COORDINATES` *dataName dim dataType*

$t_{00}\ t_{01}\ \ldots\ t_{0(dim-1)}$

$t_{10}\ t_{11}\ \ldots\ t_{1(dim-1)}$

$\ldots$

$t_{(n-1)0}\ t_{(n-1)1}\ \ldots\ t_{(n-1)(dim-1)}$

VTK File Format (continued)

Tensors

Currently only 3x3 real-valued, symmetric tensors are supported.

TENSORS *dataName dataType*

$t^0_{00}\ t^0_{01}\ t^0_{02}$
$t^0_{10}\ t^0_{11}\ t^0_{12}$
$t^0_{20}\ t^0_{21}\ t^0_{22}$
$t^1_{00}\ t^1_{01}\ t^1_{02}$
$t^1_{10}\ t^1_{11}\ t^1_{12}$
$t^1_{20}\ t^1_{21}\ t^1_{22}$
…
$t^{n-1}_{00}\ t^{n-1}_{01}\ t^{n-1}_{02}$
$t^{n-1}_{10}\ t^{n-1}_{11}\ t^{n-1}_{12}$
$t^{n-1}_{20}\ t^{n-1}_{21}\ t^{n-1}_{22}$

VTK File Format (continued)

Example: dipol.vtk

```
# vtk DataFile Version 2.0
vtk output
ASCII
DATASET STRUCTURED_GRID
DIMENSIONS 10 10 10
POINTS 1000 float
-1 -1 -1 -1 -1 -0.8 -1 -1 -0.6 -1 -1 -0.4 -1 -1 -0.2 …
POINT_DATA 1000
VECTORS vectors float
-1 1 0 -1 1 0
-1 1 0 -1 1 0
…
```

Visualizing Images

- Images are displayed as textures mapped on polygons
- In OpenGL all textures must have dimensions that are powers of two
- Images can be interpolated before display, hence some (small) loss of sharpness takes place (only visible in small images)
  - E.g. an 100x50 image can be resampled to 128x64 before display
- Similar issues for color display, i.e. scalars vs. lookup tables as in surfaces
- We will examine image display later in this course

Data in-between

Let us assume the data values are given at the vertices of a triangular or rectangular grid (the two most common cases).

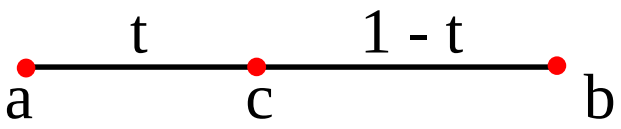
Since the data values are only known at the vertices the data set has “holes” that need to be filled in. Interpolation is a technique that does exactly that: determine a continuous function based on values that are only discretely defined.

Interpolation

Several interpolation methods are available that can be applied here. We will cover the linear case; however, other interpolation techniques, such as Hermite interpolation, can be used as well. See the course Computer Graphics II for more details on interpolation.

Linear Interpolation

In 1-D, linear interpolation is equivalent to a weighted average of two points connected by a straight line:



The value for the point in question can then be computed as $c = (1 - t) \cdot a + t \cdot b$.

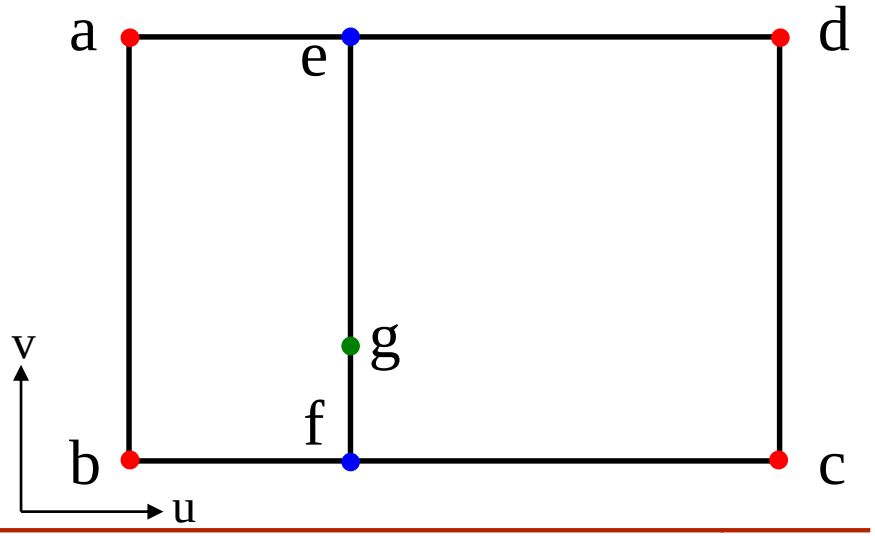
Bi-linear Interpolation

Bi-linear interpolation can be used for rectangular cells. The interpolation process is simply applied several times. First, an interpolated value along one edge is computed; another one is determined at the parallel edge. Then, the linearly interpolated value between the previously determined ones is computed resulting in the final value g at (u,v):

$$e = (1 - u) \cdot a + u \cdot d$$

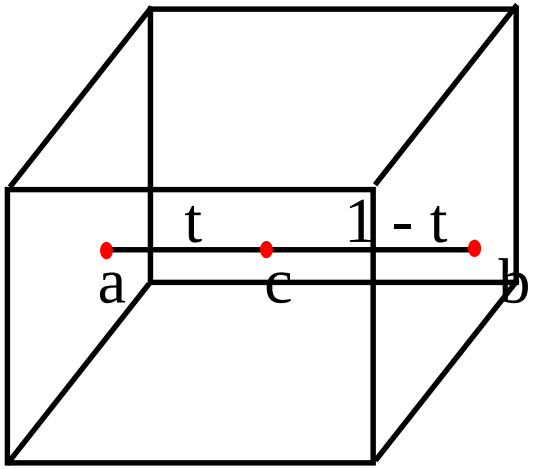
$$f = (1 - u) \cdot b + u \cdot c$$

$$g = (1 - v) \cdot f + v \cdot e$$



Tri-linear Interpolation

Tri-linear interpolation is suitable for cuboid-shaped cells. The interpolation process is applied to two parallel faces just like in the 2-D case for interpolation as seen before. After that, the resulting data value is derived by linearly interpolating between the two values we just computed:



Interpolation in Triangles

When using triangles as the basic grid element, linear interpolation can be used directly without applying several linear interpolations as a sequence. Just like in the 1-D linear interpolation, weights are determined so that the resulting value can be computed as the weighted average between the data values at the vertices of the triangle.

$$v = t_1 \cdot v_1 + t_2 \cdot v_2 + t_3 \cdot v_3$$

In order to determine these weights barycentric coordinates can be used.

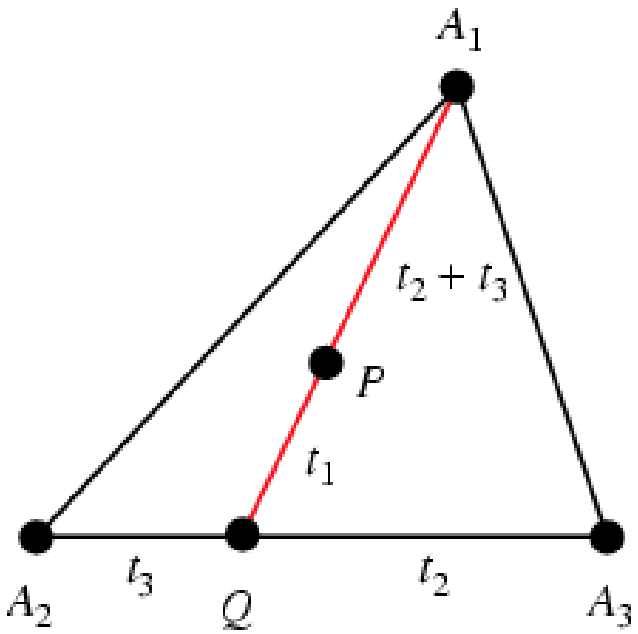
Barycentric Coordinates

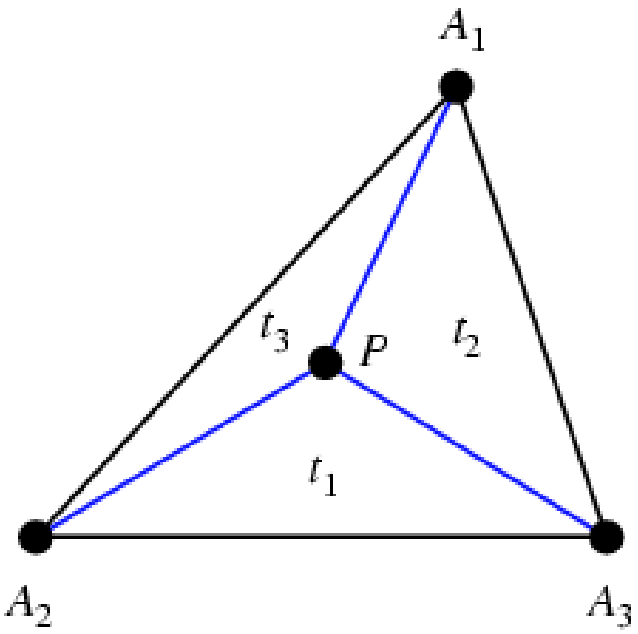
Barycentric coordinates, discovered by Möbius in 1827, define a coordinate system for points in reference to three given points, for example the vertices of a triangle.

To find the barycentric coordinates for an arbitrary point P, find t_2 and t_3 from the point Q at the intersection of the line A_1P with the side A_2A_3, and then determine t_1 as the mass at A_1 that will balance a mass $t_2 + t_3$ at Q, thus making P the centroid (left figure). Furthermore, the areas of the triangles ΔA_1A_2P, ΔA_1A_3P, and ΔA_2A_3P are proportional to the barycentric coordinates t_1, t_2, and t_3 of P.

In case of $t_1 + t_2 + t_3 = 1$ we speak of homogeneous barycentric coordinates (this is what we will use).

Barycentric Coordinates (continued)





Barycentric Coordinates (continued)

Computing homogeneous barycentric coordinates

In order to compute homogeneous barycentric coordinates, a system of linear equation needs to be solved:

$$t_1 + t_2 + t_3 = 1$$

$$A_1 \cdot t_1 + A_2 \cdot t_2 + A_3 \cdot t_3 = P$$

This looks similar to what we are looking for, i.e. we can use as weights t_1, t_2, and t_3 to form the weighted average and compute the interpolated value:

$$v = t_1 \cdot v_1 + t_2 \cdot v_2 + t_3 \cdot v_3$$

Cramer's Rule

For solving the system of linear equation Cramer's rule usually results in better performance compared to Gaussian solvers. Systems of linear equations can be solved using Cramer's rule and computing determinants:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ \vdots \\ d_n \end{pmatrix}$$

$$D := \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}, \quad D_k := \begin{vmatrix} a_{11} & \cdots & a_{1(k-1)} & d_1 & a_{1(k+1)} & \cdots & a_{1n} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & \cdots & a_{n(k-1)} & d_n & a_{n(k+1)} & \cdots & a_{nn} \end{vmatrix}$$

Then, the solution of the system of linear equations can be computed:

$$x_k = \frac{D_k}{D} \quad \text{for} \quad 1 \le k \le n$$

Interpolation in Tetrahedra

The same scheme that was used for interpolation in triangles can be applied to tetrahedra. The only difference is that the system of linear equations consists of more equations due to the higher dimensionality:

$$t_1 + t_2 + t_3 + t_4 = 1$$

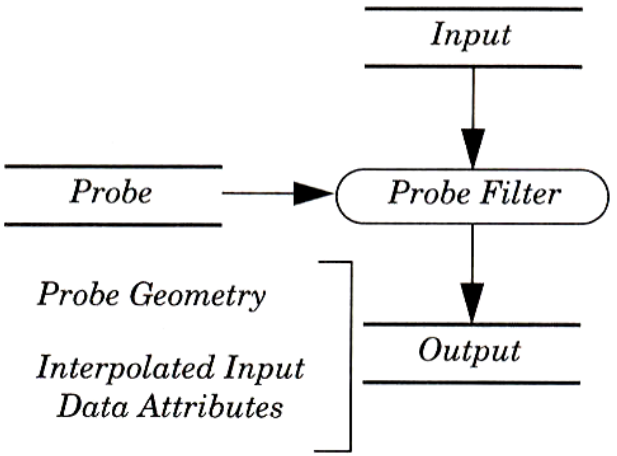
$$A_1 \cdot t_1 + A_2 \cdot t_2 + A_3 \cdot t_3 + A_4 \cdot t_4 = P$$

$$v = t_1 \cdot v_1 + t_2 \cdot v_2 + t_3 \cdot v_3 + t_4 \cdot v_4$$

Modeling algorithms

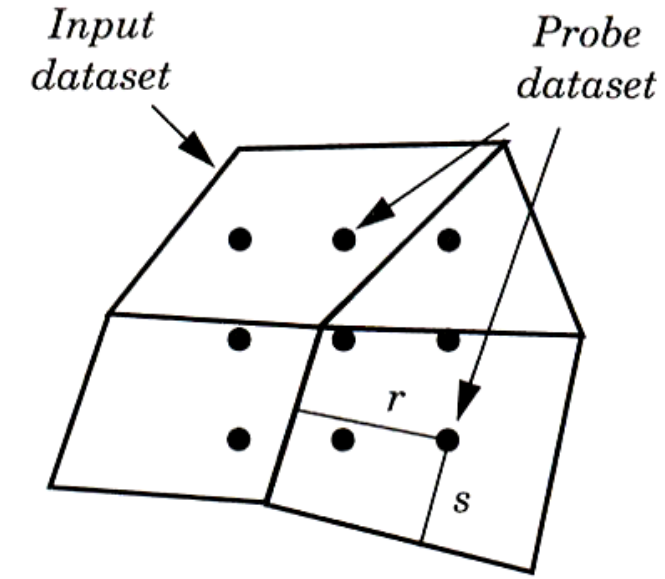
Probing/Resampling

Probing obtains data set attributes by sampling one data set (the *input*) with a set of points (the *probe*). The result of probing is a new data set (the *output*) with the topological and geometric structure of the probe data set, and point attributes interpolated from the input data set.



Modeling algorithms

For every point in the probe data set, the location in the input data set (i.e. cell, sub-cell, and parametric coordinates) and interpolation weights are determined. Then the data values from the cell are interpolated to the probe point.



Modeling algorithms

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

Modeling algorithms

One important application of probing converts irregular or unstructured data to structured form using a volume of appropriate resolution as a probe to sample the unstructured data. This is useful if we use volume rendering or other volume visualization techniques to view our data.

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)

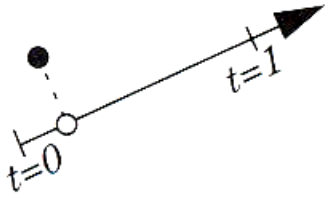
Cell / line intersection

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.

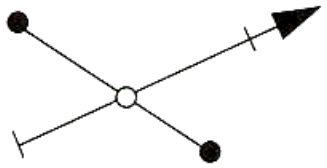
In VTK, every cell must be capable of intersecting itself against a line. The following approaches are used for each cell type.

Searching (continued)



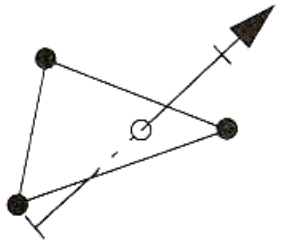
Vertex

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



Line

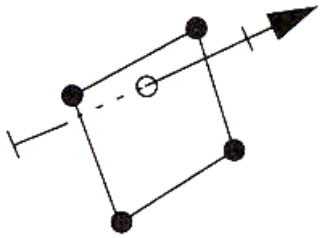
- 3D line intersection
- distance between lines must be within tolerance
- s,t must lie between [0,1]



Triangle

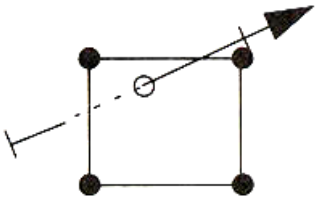
- line/plane intersection
- intersection point must lie in triangle
- t must lie between [0,1]

Searching (continued)



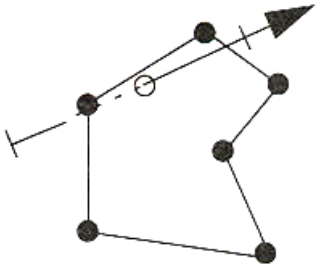
Quadrilateral

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



Pixel

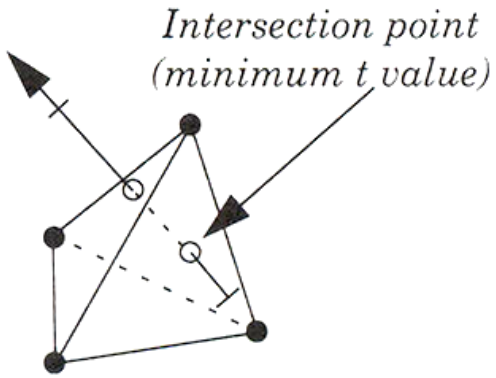
- line/plane intersection
- intersection point must lie in pixel (uses efficient in/out test)
- t must lie between [0,1]



Polygon

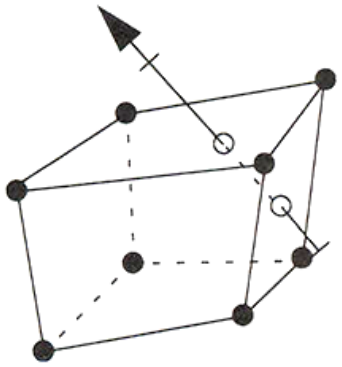
- line/plane intersection
- intersection point must lie in polygon (uses ray casting for polygon in/out)
- t must lie between [0,1]

Searching (continued)



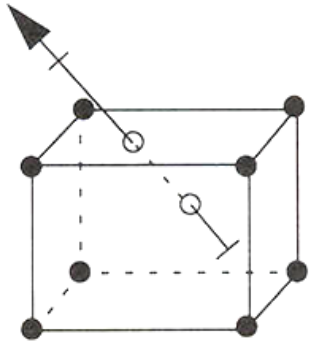
Tetrahedron

- 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*.