CPS 533 Scientific Visualization

Wensheng Shen

Department of Computational Science
SUNY Brockport
Chapter 5: Basic Data Representation

- Characteristics of data:
  - Visualization data is discrete
  - The structure of visualization data may be regular (structured) or irregular (unstructured)
  - Visualization data has a topological dimension. What is the dimension of $y=x^2$?
  - Design criterion: compact, efficient, mappable, minimal coverage, simple.
The architecture of a dataset. A dataset consists of an organizing structure, with both topological and geometric properties, and attribute data associated with the structure.
Topology is the set of properties invariant under certain geometric transformations.

Geometry is the instantiation of the topology, the specification of position in 3D space.

Structure consists of cells and points. The cells specify the topology, while the points specify geometry. Typical attributes include scalars, normals, texture coordinates, Tensors, and user-defined data.
Cells are the fundamental building blocks of visualization systems. Cells are defined by specifying a type in combination with an ordered list of points. In other words, the ordered list and type together define the topology of the cell, and the x-y-z point coordinates define the cell geometry.

**Definition:**

Type: hexahedron
Connectivity: (8,10,1,6,21,22,5,7)

This is an example of a hexahedron cell. The topology is implicitly defined by the ordering of the point list.

Cells are the fundamental building blocks of visualization systems. Cells are defined by specifying a type in combination with an ordered list of points. In other words, the ordered list and type together define the topology of the cell, and the x-y-z point coordinates define the cell geometry. ...
Cell types:

(a) Vertex
(b) Polyvertex
(c) Line
(d) Polyline (n lines)
(e) Triangle
(f) Triangle strip (n triangles)
(g) Quadrilateral
(h) Pixel
(i) Polygon (n points)
(j) Tetrahedron
(k) Hexahedron
(l) Voxel
Mathematical representation of a cell

- A cell is an ordered sequence of points, \( C_i = \{p_1, p_2, \ldots, p_n\} \) with \( C_i \) the symbol representing the cell, \( p_i \in P \), where \( P \) is a set of \( n \)-dimensional points (here \( n=3 \)). The type of cell determines the sequence of points, or cell topology. The number of points \( n \) defining the cell is the size of the cell. A cell \( C_i \) “uses” a point \( p_i \) when \( p_i \in C_i \). Hence the “use set” \( U(p_i) \) is the collection of all cells using \( p_i \):

\[
U(p_i) = \{C_i : p_i \in C_i\}
\]

- Vertices, lines, triangles, and tetrahedron are examples of 0,1,2, and 3-dimensional cells in three-dimensional space.

- Cells can be primary or composite. Composite cells consist of one or more primary cells, while primary cells cannot be decomposed into combinations of other primary cell types.
Vertex: a primary zero-dimensional cell, defined by a single point

Polyvertex: a composite zero-dimensional cell, defined by an arbitrarily ordered list of points.

Line: a primary one-dimensional cell, defined by two points, with a direction from the first point to the second point.

Polyline: a composite one-dimensional cell consisting of one or more connected lines, which is defined by an ordered list of n+1 points, where n is the number of lines in the polygon, and each pair of points (i, i+1) defines a line.

Triangle: a primary two-dimensional cell, defined by a counter-clockwise ordered list of three points. The order of points specifies the direction of the surface normal using the right-hand rule.

Triangle strip: a composite two-dimensional cell consisting of one or more triangles. The points defining the triangle strip need not lie in a plane. The triangle strip is defined by an ordered list of n+2 points, where n is the number of triangles. The ordering of the points is such that each set of three points (i, i+1, i+2) with 0 ≤ i ≤ n defines a triangle.

Quadrilateral: a primary two-dimensional cell, defined by an ordered list of four points lying in a plane. It is a convex and its edges must not intersect. The points are ordered counterclockwise around the quadrilateral, defining a surface normal using the right-hand rule.
Pixel: a primary two-dimensional cell defined by an ordered list of four points, a special case of the quadrilateral. Each edge of the pixel is perpendicular to its adjacent edges, and is parallel to one of the coordinate axes x-y-z. The normal to the pixel is also parallel to one of the coordinate axes. The points are ordered in the direction of increasing axis coordinate, starting with x, then y. How to compute the normal of the pixel?

Polygon: a primary two-dimensional cell, defined by an ordered list of three or more points located in a plane. The polygon normal is implicitly defined by a counterclockwise ordering of its points using the right-hand rule. A polygon may not be a convex.

Tetrahedron: a primary three-dimensional cell, defined by a list of four nonplanar points. the tetrahedron has six edges and four triangular faces.

Hexahedron: a primary three-dimensional cell consisting of six quadrilateral faces, twelve edges, and eight vertices. The hexahedron is defined by an ordered list of eight points.

Voxel: a primary three-dimensional cell, a special case of hexahedron. Each face of the voxel is perpendicular to one of the coordinate x-y-z axes. The defining point list is ordered in the direction of increasing coordinate value.
5.3 Attribute data

- Scalars: scalar data is data that is single valued at each location in a dataset.
- Vectors: vector data is data with a magnitude and direction. What is the two-dimensional and three-dimensional representation?
- Normals: normals are direction vectors, and they are vectors of magnitude $|n|=1$.
- Texture coordinates: texture coordinates are used to map a point from Cartesian space into a 1-, 2-, or 3-dimensional texture space. Texture maps are regular arrays of color, intensity, and/or transparency values that provide extra detail to rendered projects. For example, to paste a picture onto one or more polygons.
- Tensor: tensors are complex mathematical generalizations of vectors and matrices. A tensor of rank k can be considered as a k-dimensional table. A tensor of rank 0 is a scalar, rank 1 is a vector, rank 2 is a matrix, rank 3 is a three-dimensional rectangular array. Tensors of higher rank are k-dimensional rectangular arrays. Two-dimensional rank 2 tensors are $3 \times 3$ matrices.
- User defined data.
5.4 Types of datasets

- Regular (structured) dataset: a dataset is regular if there is a single mathematical relationship within the composing points and cells. If the points are regular, then the geometry of the dataset is regular. If the topological relationship of cells is regular, then the topology if the dataset is regular. Regular data can be implicitly represented.

- Irregular (unstructured) dataset must be explicitly represented, it tends to be more general, but requires greater memory and computational resources.
The polygonal dataset consists of vertices, polyvertices, lines, polylines, polygons, and triangle strips.

The topology and geometry of polygonal data is unstructured.

Triangle strips are high-performing primitives. To represent n triangles, n+2 points are needed. For conventional representation, 3n points are required.

Other cell types: quadrilateral meshes, Bezier curves and surfaces, and NURBS (Non-Uniform Rational B-Splines)
A structured points dataset is a collection of points and cells arranged on a regular, rectangular lattice. The rows, columns, and planes of the lattice are parallel to the global x-y-z coordinate system. If the points and cells are arranged on a plane, the dataset is referred to as a pixmap, bitmap, or image. If the points and cells are arranged as stacked planes, the dataset is referred as a volume.

Structured points consist of lines, pixels, or voxel. Structured points are regular in both geometry and topology. The number of points in the dataset is $n_x \times n_y \times n_z$, and the number of cells is $(n_x-1) \times (n_y-1) \times (n_z-1)$.

For structured points, increasing the dimensions of an image results in an $O(n^2)$ increase in memory, while volumes require an $O(n^3)$ increase.

Structured points are used in computer graphics. Volumes are often used in medical imaging such as Computed Tomography (CT) and Magnetic Resonance Imaging (MRI).
**Rectilinear grid:** the rectilinear grid dataset is a collection of points and cells arranged on a regular lattice. The rows, columns, and planes of the lattice are parallel to the Global x-y-z coordinate system. The topology of the dataset is regular, but the geometry is only partially regular. The points are aligned along the coordinate axis, but the spacing between points are not.

**Structured grid:** a structured grid is a dataset with regular topology and irregular geometry. Regular topology means the lines are perpendicular to each other.

**Unstructured points** are points irregularly located in space. There is no topology in an unstructured point dataset, and the geometry is completely unstructured. The vertex and polyvertex cells are used to represent unstructured points.

**Unstructured grid:** in unstructured grid, both topology and geometry are completely unstructured. Finite element analysis, computational geometry, geometric modeling.
Image data (structured points)

Structured grid

Rectilinear grid

Unstructured points

Unstructured grid
How are unstructured meshes different than regular grids?

- **Regular Grids**
  - e.g., Cartesian grids, logically rectangular grids
  - mesh info accessed implicitly using grid point indices
    - Efficient in both computation and storage
  - typically use finite difference discretization

- **Unstructured Meshes**
  - mesh connectivity information must be stored
    - Incurs memory and computational cost
  - handles complex geometries and grid adaptivity
  - typically use finite volume or finite element discretization
The implementation of contiguous array. The instance variable Array is a pointer to memory of type float. The allocated length of the array is given by Size. The MaxId field is an integer offset defining the end of inserted data. MaxId is $-1$ if no data has been inserted, otherwise, MaxId is an integer value where $0 \leq \text{MaxId} < \text{Size}$. The array is dynamic, so an attempt to insert Data beyond the allocated size automatically generates a Resize() operation. The Extend field specifies the amount of additional memory that is requested during a resize operation.
Abstract/concrete data array objects

Types of visualization data: floating point, integer, byte, double precision, character string, and multidimensional identifiers. How to represent the visualization data using the representation model? --- use abstract data objects.

Abstract data objects are objects that provide uniform methods to create, manipulate, and delete data using dynamic binding.

Dynamics binding allows us to execute a method belonging to a concrete object by Manipulating that object’s abstract superclass.

We use float s = GetScalar(129) to access the scalar value at point id 129. Note: since the Virtual GetScalar(1) method returns a floating-point scalar value, each subclass of vtkScalars Must also return a floating-point value. If the subclass represents other types of data value, it must transform its data representation into a floating-point value.

Vtk uses abstract data array objects for point coordinates, scalars, vectors, normals, texture Coordinates, and tensors. The concrete subclasses based on the built-in types char, short, Int, and float.
The abstract class `vtkScalars` is a subclass of `vtkReferenceCount` and `vtkObject`. The concrete classes `vtkBitScalars`, `vtkUnsignedCharScalars`, `vtkIntScalars`, `vtkShortScalars`, and `vtkFloat Scalars` are subclasses of `vtkScalars`. 
Five datasets are implemented in **vtk**, **vtkPolyData**, **vtkStructuredPoints**, **vtkStructuredGrid**, **vtkRectilinearGrid**, and **vtkUnstructuredGrid**. The unstructured points dataset is not implemented, but can be represented by either **vtkPolyData** or **vtkStructuredGrid**.

Question: can we use **vtkUnstructuredGrid** to represent all other types of datasets?
**vtkStructuredPoints**: the simplest and most compact data representation. Both the dataset points and cells are represented implicitly by specifying the dimension, data spacing, and origin. The dimensions define the topology of the dataset, while the origin and spacing specify the geometry. Both the cells and points are numbered in the direction of increasing x, then y, and then z.

**vtkRecilinearGrid**: the topology is regular, but the geometry is semi-regular. The geometry is defined by three arrays of coordinate values along all three axes. The three coordinate arrays can be combined to determine the coordinates of any point in the dataset. In **vtk**, we represent the arrays using three instances of **vtkScalar**. The numbering of points and cells is the same as **vtkStructuredPoints**.

**vtkStructuredGrid**: the topology is regular, but the geometry is not. The geometry is represented by specifying point coordinates in the global x-y-z coordinate system. Point coordinates are represented by the abstract data class **vtkPoints** and its concrete subclasses, such as **vtkFloatPoints**. The numbering of points and cells is in the same fashion as **vtkStructuredPoints**.

**vtkPolyData**: the topology is not regular, so both topology and geometry must be explicitly represented. The point data in **vtkPolyData** is represented using the **vtkPoints** class and subclasses. The class **vtkCellArray** is used to explicitly represent cell topology. This class is a list of connectivity for each cell. The design of the **vtkPolyData** class is based on the two important requirements: to have an efficient interface to external graphics libraries, and to aggregate cells according to topology.
`vtkCellArray` structure to represent cell topology. **vtkPolyData** maintains four separate lists to Vertices, lines, polygons, and triangle strips. The vertex list represents cells of type `vtkVertex` and `vtkPolyVertex`. The lines list represents cells of type `vtkLine` and `vtkPolyLine`. The polygon List represents cells of type `vtkTriangle`, `vtkQuad`, and `vtkPolygon`. The triangle strip list Represents cells of the single type `vtkTriangleStrip`. 

<table>
<thead>
<tr>
<th>count</th>
<th>Point ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>p4</td>
<td>n</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
</tr>
<tr>
<td>p3</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
<td>p3</td>
</tr>
</tbody>
</table>
**vtkUnstructuredGrid**: vtkUnstructuredGrid is the most general dataset type in terms of its ability of representing topological and geometrical structures. Both points and cells are explicitly represented using derived classes of vtkPoints and vtkCellArray. vtkUnstructuredGrid is capable of representing all cell types. In vtkUnstructuredGrid, there is an additional class, vtkCellTypes to represent cell type explicitly. The vtkCellTypes is an array of supplemental information. For each cell, an integer flag defines the cell type, another variable is used to record the location of the cell definition in the corresponding vtkCellArray. The inclusion of vtkCellTypes class enables random access to cells.
The vtkUnstructuredGrid data structure

Points

- $x_0$
- $y_0$
- $z_0$
- $x_1$
- $y_1$
- $z_1$
- $x_{n-1}$
- $y_{n-1}$
- $z_{n-1}$

Cell array

- $n$
- $p_1$
- $p_2$
- $p_3$
- $n$
- $p_1$
- $p_2$
- $p_3$

Cell types

- type$_0$
- offset$_0$
- type$_1$
- offset$_1$
- type$_2$
- offset$_2$
- type$_{n-1}$
- offset$_{n-1}$
Dataset object in **vtk**

Dataset object diagram. This diagram shows how the five datasets are implemented in **vtk**.
Cell representation in **vtk**

Object diagram for twelve cell types (plus the empty cell type) in **vtk**. Each cell is a subclass of the Abstract class vtkCell, which specifies methods that each cell must implement. Cell topology is represented by a list of ordered point ids, and cell geometry is represented by a list of point coordinates.
Data attributes in **vtk**

- Data attributes are associated with points and cells.
- In **vtk**, data attributes are associated with points.
  - Point attributes are representationally complete.
  - Point attributes are more common than cell attributes.
  - Coding and implementation is simple.
  - Data inconsistency is avoided.

To represent point attributes, we use the organizing class vtkPointData and the data specific classes vtkScalars, vtkVectors, vtkNormals, vtkTCoords, vtkTensors, and vtkUserDefined. vtkPointData serves to coordinate the movement of data from one process object to the next.
vtkPolyData *cube = vtkPolyData::New();
vtkFloatPoints *points = vtkFloatPoints::New();
vtkCellArray *polys = vtkCellArray::New();
vtkIntScalars *scalars = vtkIntScalars::New();

for (i=0; i<8; i++) points->InsertPoint(i,x[i]);
for (i=0; i<6; i++) polys->InsertNextCell(4,pts[i]);
for (i=0; i<8; i++) scalars->InsertScalar(i,i);
cube->SetPoints(points);
points->Delete();
cube->SetPolys(polys);
polys->Delete();
cube->GetPointData()->SetScalars(scalars);
scalars->Delete();

Polygonal dataset is created by constructing pieces, points, cells, and Point attribute data, and then assembling the pieces to form the complete dataset.

(1) Create an instance of vtkPoints, and use the operator cube->SetPoints() to associate the points with the dataset.

(2) Create an instance of vtkCellArray to define topology for vertices, lines, polygons, and triangle strips. Use operators cube->SetVerts(), cube->SetLines(), cube->SetPolys(), and cube->SetStrips() to associate the cells with the dataset.

(3) Create an instance of vtkPointData to add point attribute data. Use operator pd=cube->GetPointData() to retrieve the pointer to the point attribute data. Use operators pd->SetScalars(), pd->SetVectors(), pd->SetNormals(), pd->SetTensors(), pd->SetTCoords(), and pb->SetUserDefined().
Creating a structured points dataset. Scalar data is generated from the equation for a sphere.
Creating a structured grid dataset of a semicylinder. Vectors are created whose magnitude is proportional to radius and oriented in tangential direction.
Creating a rectilinear grid dataset. The coordinates along each axis are defined using an instance of `vtkScalar`.

```cpp
vtkFloatScalars *xCoords = vtkFloatScalars::New();
for (i = 0; i < 47; i++) xCoords->InsertScalar(i, x[i]);

vtkFloatScalars *yCoords = vtkFloatScalars::New();
for (i = 0; i < 33; i++) yCoords->InsertScalar(i, y[i]);

vtkFloatScalars *zCoords = vtkFloatScalars::New();
for (i = 0; i < 44; i++) zCoords->InsertScalar(i, z[i]);

vtkRectilinearGrid *rgrid = vtkRectilinearGrid::New();
rgrid->SetDimensions(47, 33, 44);
rgrid->SetXCoordinates(xCoords);
rgrid->SetYCoordinates(yCoords);
rgrid->SetZCoordinates(zCoords);

vtkRectilinearGridGeometryFilter *plane =
    vtkRectilinearGridGeometryFilter::New();
plane->SetInput(rgrid);
plane->SetExtent(0, 46, 16, 16, 0, 43);

vtkPolyDataMapper *rgridMapper = vtkPolyDataMapper::New();
rgridMapper->SetInput(plane->GetOutput());

vtkActor *wireActor = vtkActor::New();
wireActor->SetMapper(rgridMapper);
wireActor->GetProperty()->SetRepresentationToWireframe();
wireActor->GetProperty()->SetColor(0, 0, 0);
```
vtkFloatPoints *points = vtkFloatPoints::New();
for(i=0; i<27; i++) points->InsertPoint(i, x[i]);

vtkUnstructuredGrid *ugrid = vtkUnstructuredGrid::New();
ugrid->Allocate(100);
ugrid->InsertNextCell(VTK_HEXAHEDRON, 8, pts[0]);
ugrid->InsertNextCell(VTK_HEXAHEDRON, 8, pts[1]);
ugrid->InsertNextCell(VTK_TETRA, 4, pts[2]);
ugrid->InsertNextCell(VTK_TETRA, 4, pts[3]);
ugrid->InsertNextCell(VTK_POLYGON, 6, pts[4]);
ugrid->InsertNextCell(VTK_TRIANGLE_STRIP, 6, pts[5]);
ugrid->InsertNextCell(VTK_QUAD, 4, pts[6]);
ugrid->InsertNextCell(VTK_TRIANGLE, 3, pts[7]);
ugrid->InsertNextCell(VTK_TRIANGLE, 3, pts[8]);
ugrid->InsertNextCell(VTK_LINE, 2, pts[9]);
ugrid->InsertNextCell(VTK_LINE, 2, pts[10]);
ugrid->InsertNextCell(VTK_VERTEX, 1, pts[11]);

Ugrid->SetPoints(points);
points->Delete();