Basic data representations

Basic data representations

Overview

This chapter will introduce you to basic 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 way VTK handles different types of grids and what you can do with it.

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

Motivation

How can we store scientific data?

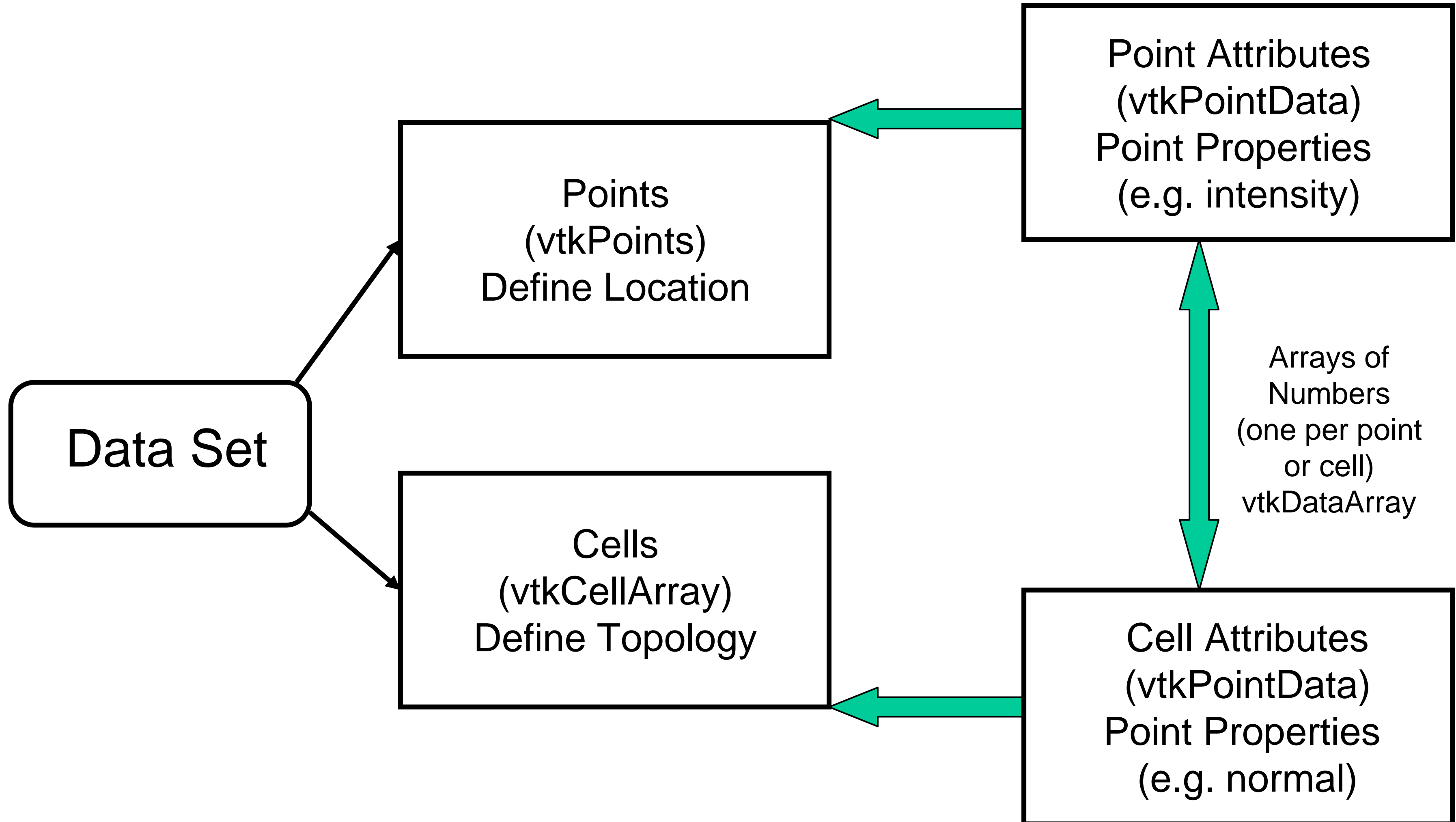
The data structure should:

- 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

Motivation

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.

Data Representation (vtkDataSet)



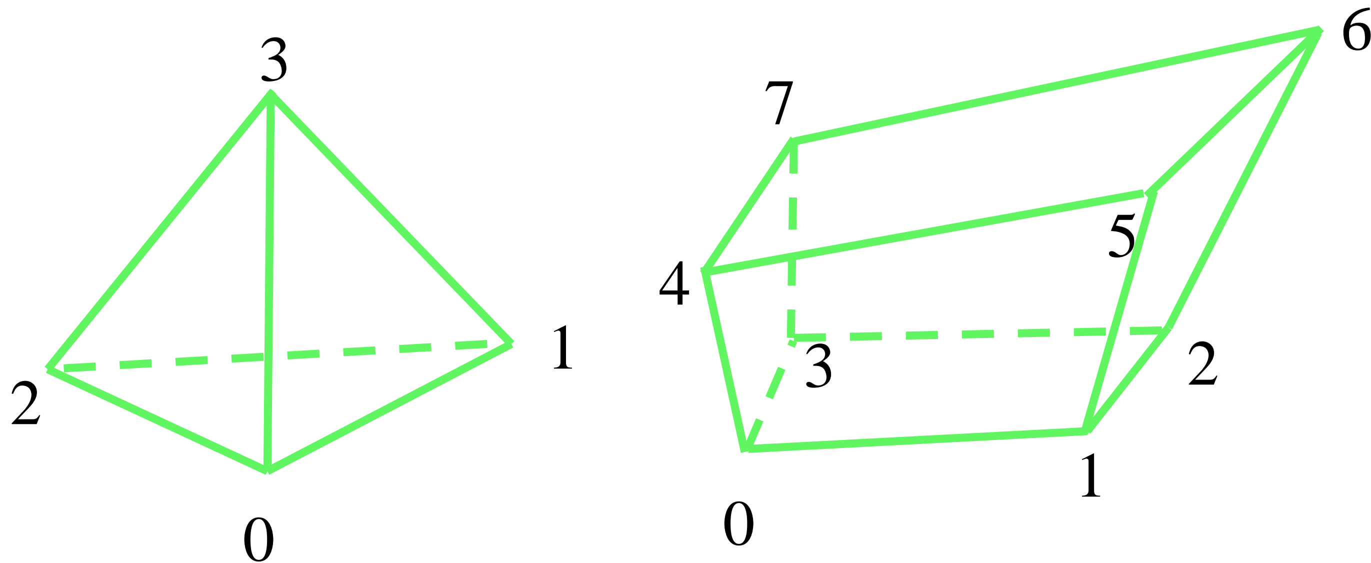
Cell Topology

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

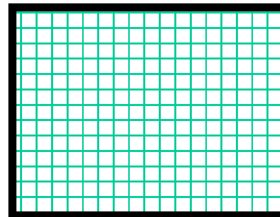
VTK provides several different grid data structures:

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

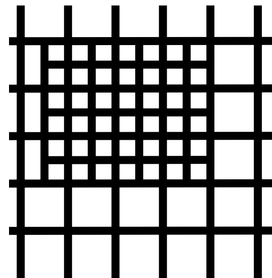
Datasets

Organizing structure plus attributes

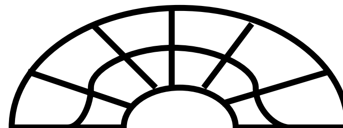
- Structured points



- Rectilinear Grid

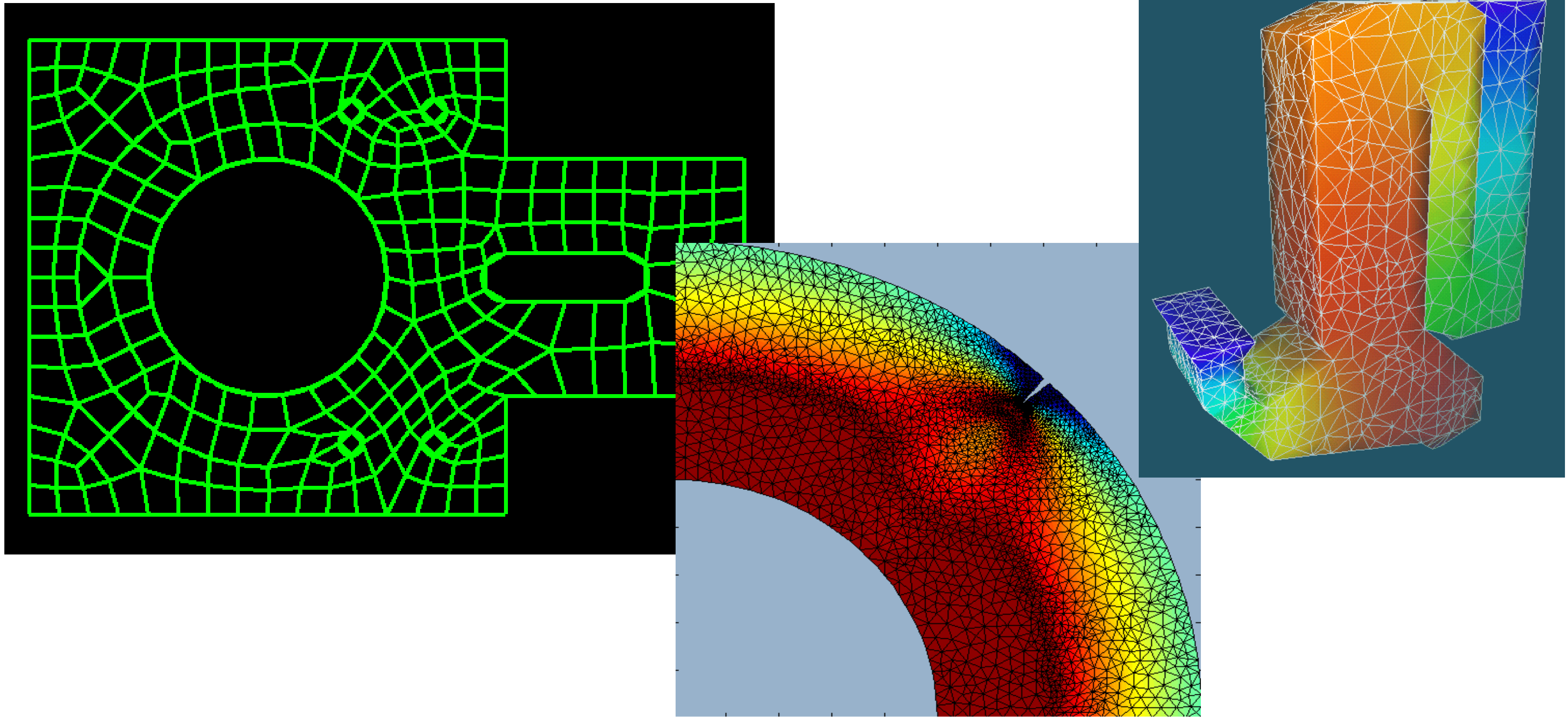


- Structured Grid

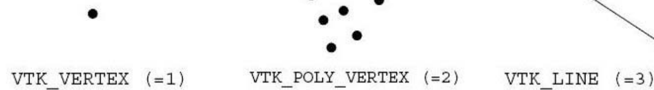


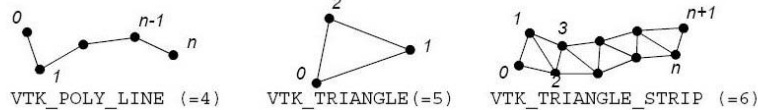
Unstructured Grid

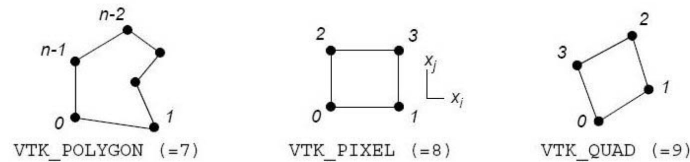
A collection of vertices, edges, faces and cells whose connectivity information must be explicitly stored

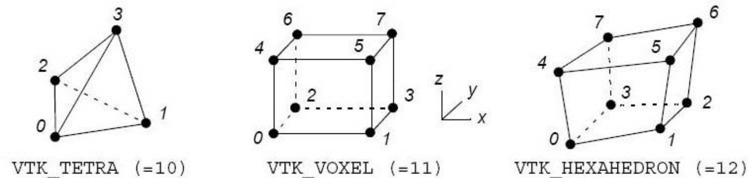


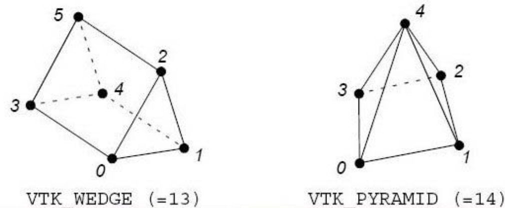
Available Cell Types in VTK











Data Attributes

Data attributes are assigned to points or cells

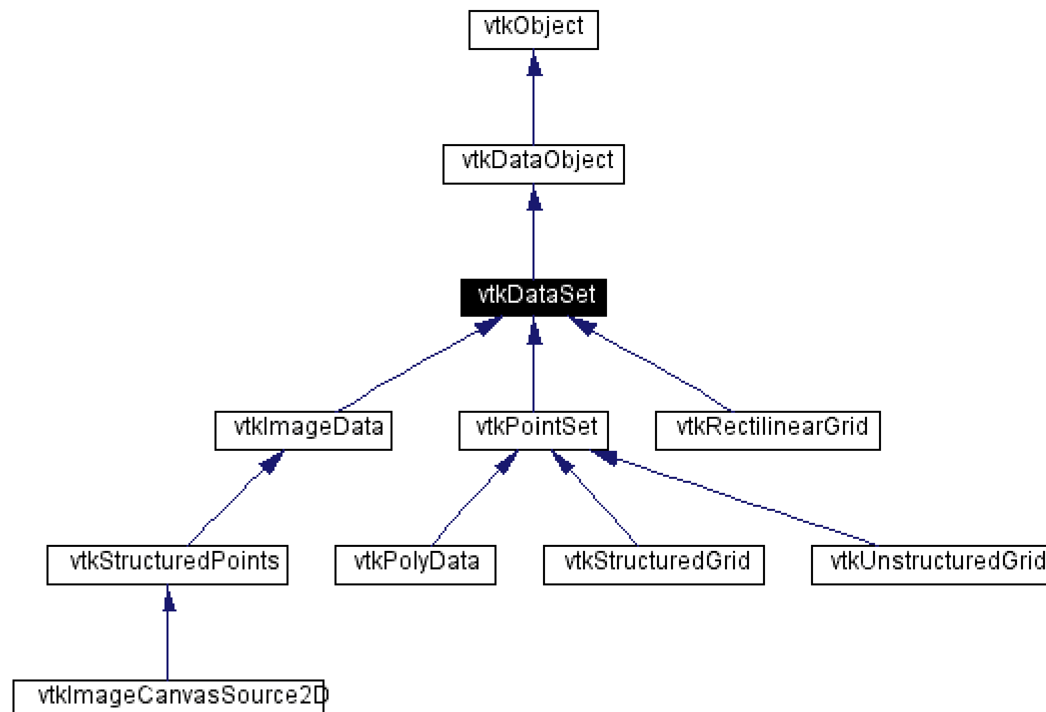
- Scalars
- Vector
  - Magnitude and direction
- Normal
  - a vector of magnitude 1
  - Used for lighting
- Texture Coordinate
  - Mapping data points into a texture space
- Tensor

VTK Hierarchy

- In order to understand VTK structure we need to look at class hierarchy
- Common functionality implemented in base-level (parent) classes
- Specialized functionality implemented in lower-level (children) classes

VTK Hierarchy (continued)

VTK class hierarchy for data structures:



Data Structure

Several member functions exist to access the cells and the vertices of the cells in the class `vtkDataSet`:

- `virtual vtkIdType GetNumberOfPoints ()=0`
- `virtual vtkIdType GetNumberOfCells ()=0`
- `virtual float *GetPoint (vtkIdType ptId)=0`
- `virtual void GetPoint (vtkIdType id, float x[3])`
- `virtual vtkCell *GetCell (vtkIdType cellId)=0`
- `virtual void GetCell (vtkIdType cellId, vtkGenericCell *cell)=0`
- `virtual void GetCellBounds (vtkIdType cellId, float bounds[6])`
- `virtual int GetCellType (vtkIdType cellId)=0`
- `virtual void GetCellTypes (vtkCellTypes *types)`
- `virtual void GetCellPoints (vtkIdType cellId, vtkIdList *ptIds)=0`
- `virtual void GetPointCells (vtkIdType ptId, vtkIdList *cellIds)=0`

Data Structure (continued)

The class `vtkDataSet` has the following protected members where it stores the data:

- `vtkCellData *CellData`
- `vtkPointData *PointData`

These arrays are then used to store all data values that are represented by this instance of `vtkDataSet` attached to the vertices or the cells itself.

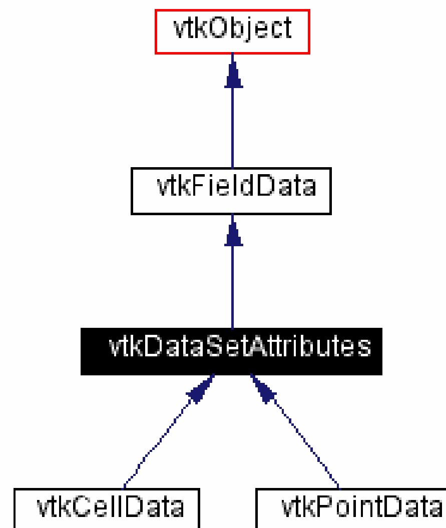
These arrays are represented by the two classes `vtkCellData` and `vtkPointData`.

Array Representation

Both classes, `vtkCellData` and `vtkPointData`, are derived from the same class: vtkDataSetAttributes

This class has the member function GetAttribute to access the array data structure:

```
vtkDataArray *GetAttribute (int attributeType)
```



Example: vtkPolyData

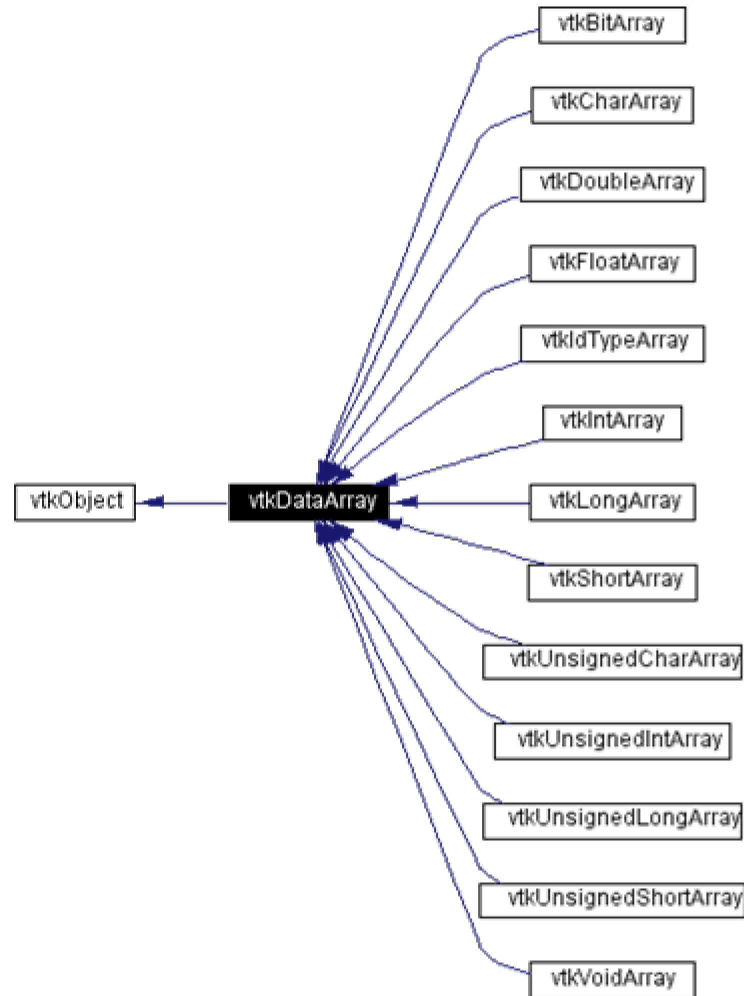
`vtkPolyData` is a complex class which has many members. The key ones are:

- Points of type `vtkPoints` – represents the geometry of the surface (i.e. the points)
- Polys of type `vtkCellArray` – represents part of the topology of the surface (i.e. the polygons)
- PointData of type `vtkPointData` – represents data associated with the points (e.g. normals, colors etc.)
- CellData of type `vtkCellData` – represents data associated with the cells (e.g. normals, colors etc.)
- Lots of other members …..

Example: vtkPolyData (continued)

- vtkDataArray is an abstract superclass for classes representing arrays of vectors called tuples (or numbers treated as vectors of length 1). Each tuple consists of a set of numbers or components.
- Derived Classes include `vtkUnsignedCharArray`, `vtkShortArray`, `vtkFloatArray`, `vtkDoubleArray` etc.
- Can function either as a dynamic array (lower performance) or a fixed length array
- All data in VTK is stored ultimately in one of the many derived classes of vtkDataArray, e.g. in the case of vtkImageData the intensities are stored in a vtkDataArray having dimensions equal to the number of voxels and vector length typically equal to 1 (3 for color images, Number of Frames for multiframe data such as fMRI cardiac etc.)

Class Hierarchy



A specific example: vtkFloatArray

Option 1 – Fixed Length Array

To create a `vtkFloatArray`:

```
float t[] = { 9.0 };
vtkFloatArray *arr = vtkFloatArray::New ();
arr->SetNumberOfComponents (1);
arr->SetNumberOfTuples (20);
arr->SetComponent (10, 0, 10.0);
arr->SetTuple (11, t);
float b = arr->GetComponent (10, 0);
```

This creates an array of 20 (number of tuples) vectors each having size 1 (number of components)

We access elements in this array be using the `SetComponent` and `GetComponent` methods. All indices start at 0.

A specific example: vtkFloatArray

Option Mode 2 – Dynamic Array

To create a `vtkFloatArray`:

```
float value;
vtkFloatArray *arr = vtkFloatArray::New ();
arr->SetNumberOfComponents (1);
value = 5;
arr->InsertNextTuple (&value);
value = 10;
arr->InsertNextTuple (&value);
flaot b = arr->GetComponent (1, 0);
```

This creates a dynamic array of vectors each having size 1 (number of components). The `InsertNextTuple` command allocates memory dynamically and places the value there.

Creating Data Arrays

The following constants are often defined

```
set VTK_VOID              0        set VTK_INT              6
set VTK_BIT               1        set VTK_UNSIGNED_INT     7
set VTK_CHAR              2        set VTK_LONG             8
set VTK_UNSIGNED_CHAR     3        set VTK_UNSIGNED_LONG    9
set VTK_SHORT             4        set VTK_FLOAT           10
set VTK_UNSIGNED_SHORT    5        set VTK_DOUBLE          11
```

You can use the generic command

`vtkDataArray::CreateDataArray(type)`

to create an array of type to be specified at run-time.

Creating a Surface Manually – Step 1

```
// Create Points for a cube
vtkPoints *pt = vtkPoints::New ();
pt->SetNumberOfPoints (8);
pt->SetPoint (0, 0, 0, 0); pt->SetPoint (1, 1, 0, 0);
pt->SetPoint (2, 1, 1, 0); pt->SetPoint (3, 0, 1, 0);
pt->SetPoint (4, 0, 0, 1); pt->SetPoint (5, 1, 0, 1);
pt->SetPoint (6, 1, 1, 1); pt->SetPoint (7, 0, 1, 1);
// Create Polygons
vtkCellArray *cl = vtkCellArray::New ();
// Insert a Square
cl->InsertNextCell (4);
cl->InsertCellPoint (0); cl->InsertCellPoint (1);
cl->InsertCellPoint (2); cl->InsertCellPoint (3);
// Insert a Triangle
cl->InsertNextCell (3);
cl->InsertCellPoint (2); cl->InsertCellPoint (3);
cl->InsertCellPoint (6);
```

Creating a Surface Manually – Step 2

```
// Create the Surface
vtkPolyData *sur = vtkPolyData::New ();
// Set the points and cleanup
sur->SetPoints (pt); pt->Delete ();
// Set the polygons and cleanup
sur->SetPolys (cl); cl->Delete ();
// Create a Mapper and set its input
vtkPolyDataMapper *map = vtkPolyDataMapper::New (); map->SetInput (sur);
// Create the actor and set it to display wireframes
vtkActor act = vtkActor::New (); act->SetMapper (map);
vtkProperty *property = act->GetProperty ();
property->SetColor  (1, 1, 1); property->SetAmbient (1.0);
property->SetDiffuse 0.0;   property->SetSpecular (0.0);
property->SetRepresentationToWireframe ();
// Create the renderer
vtkRenderer *ren = vtkRender (); ren->AddActor (act);
// Set camera mode to Orthographic as opposed to Perspective
ren->GetActiveCamera ()->ParallelProjectionOn ();
```

Creating a Surface Manually – Step 3

```
// Remainder is standard window/interactor etc.
// Render Window
vtkRenderWindow *renWin = vtkRenderWindow::New ();
renWin->AddRenderer (ren);
renWin->SetSize (300, 300);

// Interactor
vtkRenderWindowInteractor *iren =
   vtkRenderWindowInteractor::New ();
iren->SetRenderWindow (renWin);
iren->Initialize ();
iren->Start ();
```

Finishing the Cube

```
// Insert rest of cube - rest of the definitions
cl->InsertNextCell (4);
cl->InsertCellPoint (0); cl->InsertCellPoint (1);
cl->InsertCellPoint (5); cl->InsertCellPoint (4);
cl->InsertNextCell (4);
cl->InsertCellPoint (0); cl->InsertCellPoint (3);
cl->InsertCellPoint (7); cl->InsertCellPoint (4);
cl->InsertNextCell (4);
cl->InsertCellPoint (1); cl->InsertCellPoint (2);
cl->InsertCellPoint (6); cl->InsertCellPoint (5);
cl->InsertNextCell (4);
cl->InsertCellPoint (4); cl->InsertCellPoint (5);
cl->InsertCellPoint (6); cl->InsertCellPoint (7);
cl->InsertNextCell (3);
cl->InsertCellPoint (3); cl->InsertCellPoint (7);
cl->InsertCellPoint (6);
```

Setting up Colors

```
// Create color array, colors stored as RGB values 0-255
char values[3];
vtkUnsignedCharArray *ar = vtkUnsignedCharArray::New ();
ar->SetNumberOfComponents (3);
ar->SetNumberOfTuples (8);
values[0] = 255; values[1] = 0; values[2] = 0; ar->SetTuple (0, values);
values[0] = 255; values[1] = 0; values[2] = 0; ar->SetTuple (1, values);
values[0] = 255; values[1] = 0; values[2] = 0; ar->SetTuple (2, values);
values[0] = 255; values[1] = 255; values[2] = 0; ar->SetTuple (3, values);
values[0] = 0; values[1] = 255; values[2] = 0; ar->SetTuple (4, values);
values[0] = 0; values[1] = 255; values[2] = 0; ar->SetTuple (5, values);
values[0] = 255; values[1] = 255; values[2] = 255; ar->SetTuple (6, values);
values[0] = 255; values[1] = 255; values[2] = 255; ar->SetTuple (7, values);

vtkPolyData *sur = vtkPolyData::New ();
sur->SetPoints (pt) ; pt->Delete ();
sur->SetPolys (cl); cl->Delete ();
// Set The Colors and Cleanup
sur->GetPointData ()->SetScalars (ar);
ar->Delete ();
```

Viewing Surface with Solid Faces

```
vtkUnsignedCharArray *ar = vtkUnsignedCharArray::New ();
ar->SetNumberOfComponents (3);
ar->SetNumberOfTuples (7);
values[0] = 255; values[1] = 255; values[2] = 0; ar->SetTuple (0, values);
values[0] = 255; values[1] = 192; values[2] = 0; ar->SetTuple (1, values);
values[0] = 255; values[1] = 128; values[2] = 0; ar->SetTuple (2, values);
values[0] = 255; values[1] = 64; values[2] = 0; ar->SetTuple (3, values);
values[0] = 255; values[1] = 0; values[2] = 0; ar->SetTuple (4, values);
values[0] = 255; values[1] = 255; values[2] = 255; ar->SetTuple (5, values);
values[0] = 0; values[1] = 255; values[2] = 255; ar->SetTuple (6, values);

vtkPolyData *sur = vtkPolyData::New ();
sur->SetPoints (pt); pt->Delete ();
sur->SetPolys (cl);  cl->Delete ();
// Add Colors to polygons not points (i.e. use Cell Data)
sur->GetCellData ()->SetScalars (ar);
ar->Delete ();
```

Viewing Surface with Solid Faces (cont.)

```
vtkFloatArray *ar = vtkFloatArray ();
float value;
ar->SetNumberOfComponents (1);
ar->SetNumberOfTuples (7);
value = 0; ar->SetTuple (0, &value); value = 0; ar->SetTuple (1, &value);
value = 2; ar->SetTuple (2, &value); value = 3; ar->SetTuple (3, &value);
value = 4; ar->SetTuple (4, &value); value = 5; ar->SetTuple (5, &value);
value = 6; ar->SetTuple (6, &value);

// Colors are Red/Green/Blue/Opacity in range 0 to 1
vtkLookupTable *lkup = vtkLookupTable::New ();
lkup->SetNumberOfTableValues (7);
lkup->SetTableRange (0, 6);
lkup->SetTableValue (0, 1.0, 1.00, 0.0, 1.0);
lkup->SetTableValue (1, 1.0, 0.75, 0.0, 1.0);
lkup->SetTableValue (2, 1.0, 0.50, 0.0, 1.0);
lkup->SetTableValue (3, 1.0, 0.25, 0.0, 1.0);
lkup->SetTableValue (4, 1.0, 0.0,  0.0, 1.0);
lkup->SetTableValue (5, 1.0, 1.0,  1.0, 1.0);
lkup->SetTableValue (6, 0.0, 1.0,  1.0, 1.0);
```

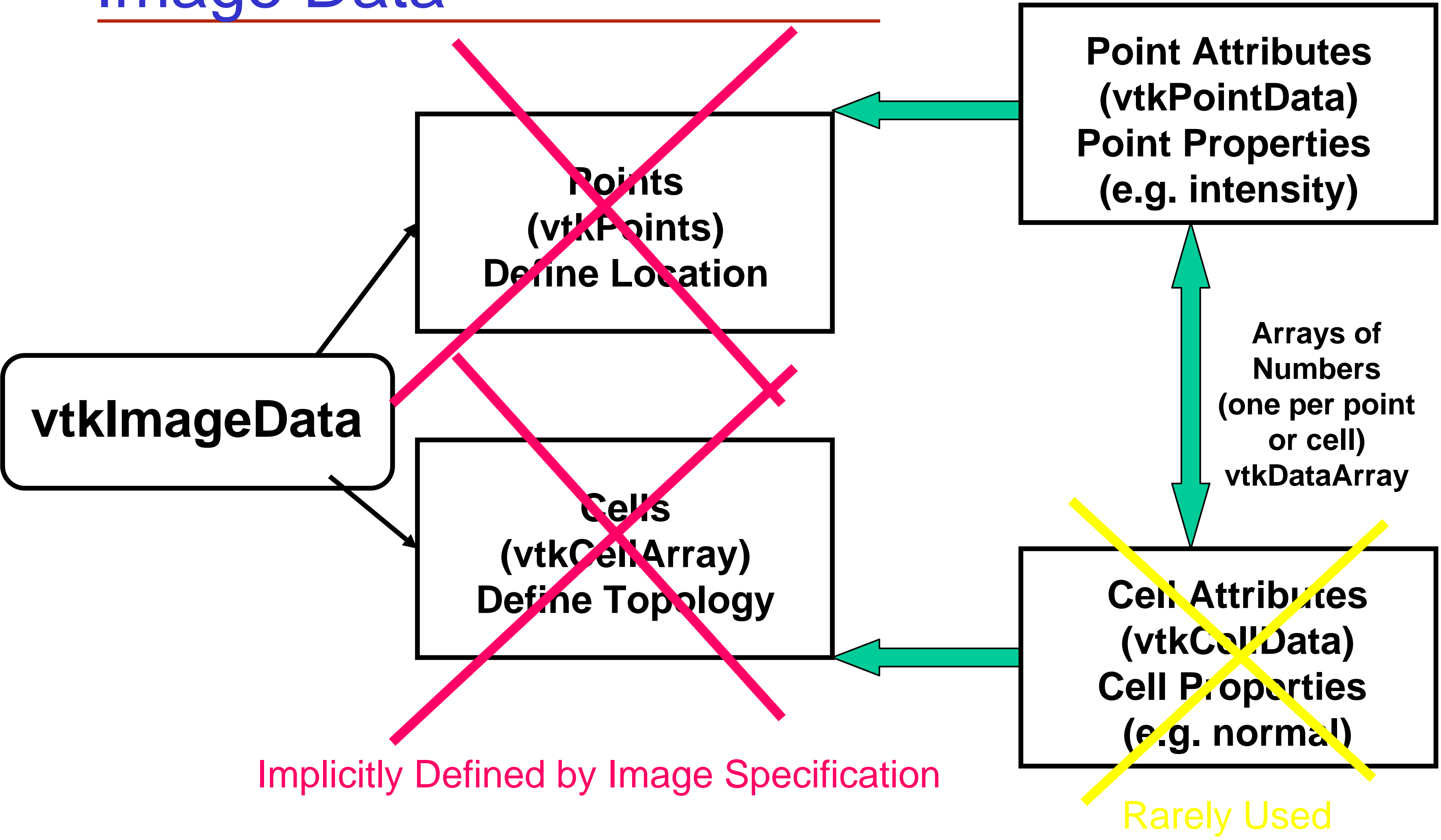
Viewing Surface with Solid Faces (cont.)

```
vtkPolyData sur
sur->SetPoints (pt);
pt->Delete ();
sur->SetPolys (cl);
cl->Delete ();

// Add Colors to polygons not points (i.e. use Cell Data)
sur->GetCellData->SetScalars (ar);
ar->Delete ();

vtkPolyDataMapper *map = vtkPolyDataMapper::New ();
map->SetInput (sur);
map->SetLookupTable (lkup);
map->SetScalarRange (0, 6);
map->SetScalarModeToUseCellData ();
```

Image Data



vtkImageData

`vtkImageData` is the basic VTK class for storing images. It is defined by 4 key elements:

- Dimensions - these define the size of the image
- Origin - the position in 3D space of point (0, 0, 0)
- Spacing - the voxel dimensions
- Scalar type - the data type of the image (e.g. float, short etc.)

An 4x4x4 image has 4x4x4=64 points and 3x3x3=27 cubic cells (both are implicitly defined)

Manually Creating an Image

```
vtkImageData *img = vtkImageData::New ();
img->SetDimensions (10, 10, 2);
img->SetOrigin (0, 0, 0);
img->SetSpacing (0.78, 0.78, 1.5);
img->SetScalarType (VTK_SHORT);
img->SetNumberOfScalarComponents (1);
img->AllocateScalars ();
```

Intensity values can be accessed using the scalar array i.e. Point 0 is (0,0,0), point 1 is (1,0,0), point 10 is (0,1,0), point 100 is (0,0,1)

```
vtkDataArray *data = img->GetPointData->GetScalars ();
data->SetComponent (10, 0, 5.0);
float v = data->GetComponent (10, 0);
float v2 = img->GetScalarComponentAsFloat (0, 1, 0, 0);
```

(this unfortunately is the nearest vtk comes to a getvoxel, no set voxel command)

The Image as a Function

- VTK treats an image as a function which takes values equal to those specified at the points and interpolates in-between
- Standard interpolation is (tri-)linear
- In some cases (`vtkImageReslice`) other interpolation schemes are available.

Image File Read/Write

- VTK supports by default a number of standard image file formats for read/write
  - Binary – `vtkImageReader`
  - JPEG – `vtkJPEGReader`, `vtkJPEGWriter`
  - PNG – `vtkPNGReader`, `vtkPNGWriter` (ppm,pgm)
  - TIFF – `vtkTIFFReader`, `vtkTIFFWriter`
  - BMP – `vtkBMPReader`, `vtkBMPWriter`
- There are local extensions for reading Analyze, Signa LX/SPR, Prism (SPECT) etc

ImageToImage Filters

- There a number of ImageToImage Filters which are analogous to the surface to surface filters we have met before.
- There are exist standard filters derived from `vtkImageToImageFilter` for among others
  - Smoothing – `vtkImageGaussianSmooth, vtkImageMedian3D`
  - Computing Gradiens/Laplacians – `vtkImageGradient, vtkImageLaplacian`
  - Fourier Operations – `vtkImageFFT, vtkImageRFFT`
  - Resampling/Reslicing – `vtkImageResample, vtkImageReslice` (vtkImageReslice on its own is reason enough to learn VTK, it implements enough operations that would take more than a year to code from scratch!)
  - Flipping, Permuting – `vtkImageFlip, vtkImagePermute`

Visualizing Images

- Images are displayed as textures mapped on polygons
- In OpenGL all textures must have dimensions that are powers of two
- Images are interpolated before display, hence some (small) loss of sharpness takes place (only visible in small images)
  - E.g. an 100x50 image will 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).

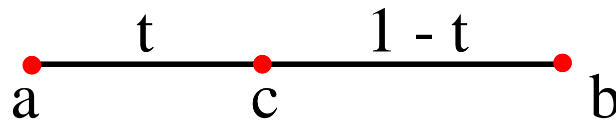
Since the data values are only known at the vertices the data set has “holes” that need to be filled in. Interpolation is a technique that does exactly that: determine a continuous function based on values that are only discretely defined.

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 $c = (1 - t) \cdot a + t \cdot b$.

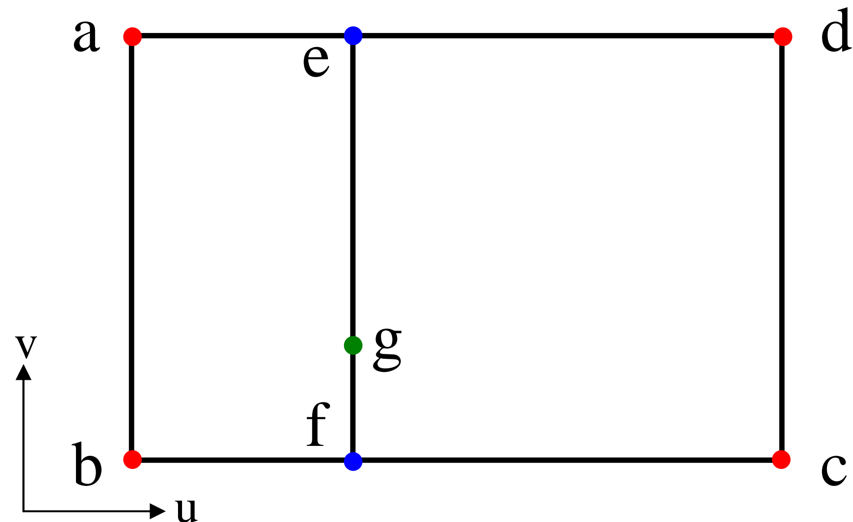
Bi-linear Interpolation

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):

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

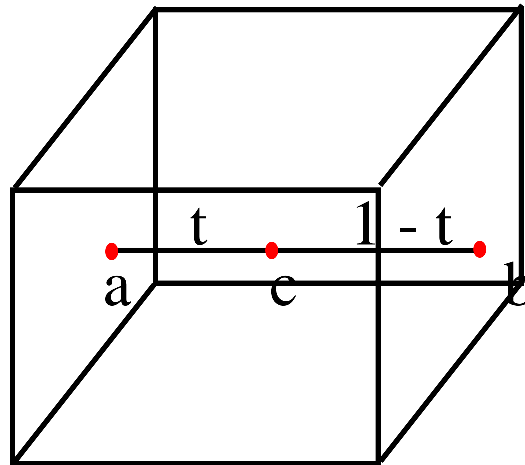
$$f = (1 - u) \cdot b + u \cdot c$$

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



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.

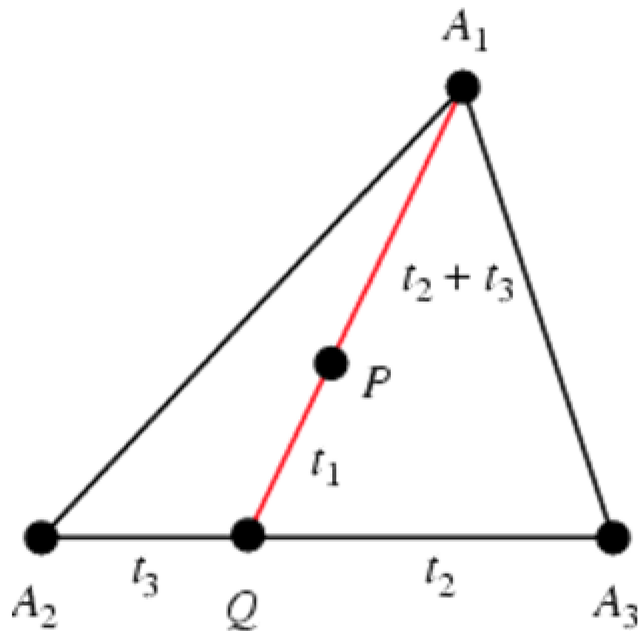
Barycentric Coordinates

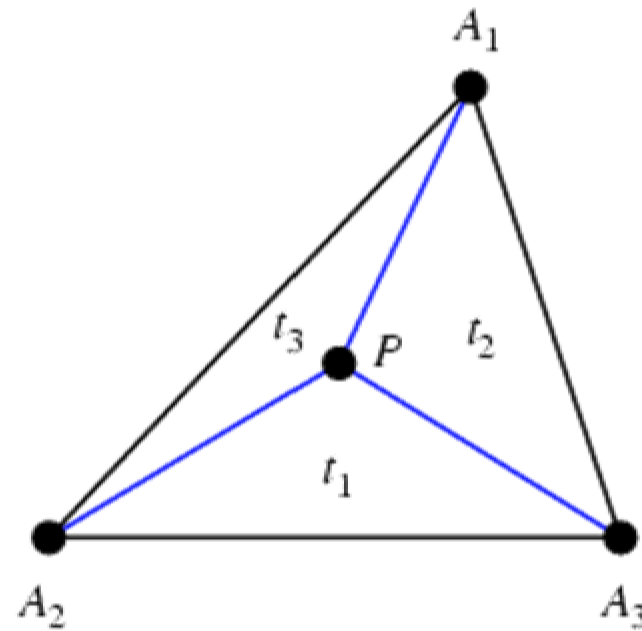
Barycentric coordinates, discovered by Möbius in 1827, define a coordinate system for points in reference to three given points, for example the vertices of a triangle.

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:

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

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:

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

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

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