Introduction to Visualization ToolKit

Stefano Perticoni – s.perticoni@scsitaly.com
Index

- General Introduction
- Data Structures
- Filtering
- Rendering
- Strategies and optimizations
What is VTK

• History - born in 1993 as example code from the visualization textbook. Ever since then it has grown via open source services funded model
• Kitware hosts the project and are primary developers
• VTK is a library - something that applications use
What can VTK do for me?

- SCI VIS — 2 to 4D data processing and (volume) rendering
- image processing
- text analysis and information visualization
- charting/plotting
- GUI support
What VTK can do for me
What VTK can do for me
What VTK can do for me
Kitware

- Site
- Cmake
- ITK
- Paraview
- VolView

www.kitware.com
Characteristics

VKT is a C++ library

• FREE
• Open Source
• Cross Platform
• Extensible
• More then 600 classes
• Documented
• Dashboards
Characteristics (2)

• Structure:
  object oriented c++ core
  interpreted wrappings

• Interpreted layer generated automatically by VTK wrapping process
Characteristics of VTK

**Application**
C++, Java, Tcl, Python

**VTK**
New classes defined by the developer

**Graphic Libraries**
OpenGL, XGL, Starbase, Mesa, ...

**S.O.**
Windows, Linux, Irix, ...

High level programming
Creation of applications

Low level programming
Extending the library
**Data**

**Information**
One or more values that vary in a certain domain

**Discretization or sampling**
Domain partitioning in cells and measure values corresponding to the vertices. (and/or cells)

**Data**
Discrete representation of the information

**Structure**

**Geometry**
vertices property

**Topology**
cells property

**Attributes**
Whole measures

**DataSet**
G T A
Attributes

• Association
  – Points attributes
  – Cells attributes

• Type
  – Scalars (max 4 components)
  – Vectors (3 components)
  – Tensors rank 3 (9 components)
  – Normal (3 components)
  – Texture Coordinates (max 3 components)
  – Fields (n*m components)

• Representation
  – char .... double
Dato->GetPointData()->GetScalars()->GetValue(1);
Data types

- vtkStructuredPoints
  - (vtkImageData)

- vtkRectilinearGrid

- vtkStructuredGrid

- vtkPolyData

- vtkUnstructuredGrid
vtkStructuredPoints

Geometry and Topology (voxel) are both implicit and are determined using Origin, Dimensions, and Spacing.

Sample C++ code that creates a StructuredPoints

```cpp
vtkStructuredPoints *sp = vtkStructuredPoints::New();
sp->SetOrigin (0,0,0);
sp->SetDimensions(3,3,3);
sp->SetSpacing (1,1,1);

vtkFloatArray *fa = vtkFloatArray::New();
for(i=0; i<27; i++)
    fa->InsertValue( i, i );
SP->GetPointData()->SetScalars( fa );
```

(a) Structured Points
Sample Python code that creates a StructuredPoints

```python
sp = vtk.vtkStructuredPoints()
sp.SetOrigin    (0,0,0)
sp.SetDimensions(3,3,3)
sp.SetSpacing   (1,1,1)

fa = vtk.vtkFloatArray()
for i in range(0,27):
    fa.InsertValue( i, i )

sp.GetPointData().SetScalars( fa )
```

vtkRectilinearGrid

- Implicit Topology (hexahedron)
- Geometry obtained combining values of X,Y,Z coordinates specified using three arrays.

vtkFloatArray *fa = vtkFloatArray::New();
fa->InsertValue( 0, 0 );
fa->InsertValue( 1, 1 );
fa->InsertValue( 2, 3 );
fa->InsertValue( 3, 6 );

vtkRectilinearGrid *rg = vtkRectilinearGrid::New();
rg->SetDimensions (4,4,4);
rg->SetXCoordinates(fa);
rg->SetYCoordinates(fa);
rg->SetZCoordinates(fa);
vtkRectilinearGrid

- Implicit Topology (hexahedron)
- Geometry obtained combining values of X,Y,Z coordinates specified using three arrays.

```python
fa = vtk.vtkFloatArray()
fa.InsertValue( 0, 0 )
fa.InsertValue( 1, 1 )
fa.InsertValue( 2, 3 )
fa.InsertValue( 3, 6 )

rg = vtk.vtkRectilinearGrid()
rg.SetDimensions(4,4,4)
rg.SetXCoordinates(fa)
rg.SetYCoordinates(fa)
rg.SetZCoordinates(fa)
```
vtkStructuredGrid

• Implicit Topology – (hexahedron)
• Explicit Geometry

```cpp
tvtkPoints *p = vtkPoints::New();
p->InsertNextPoint( 0,0,0 );
p->InsertNextPoint( 0,0,0 );
p->InsertNextPoint( 1,0,0 );
p->InsertNextPoint( 1,0,0 );
p->InsertNextPoint( 0,1,0 );
p->InsertNextPoint( 0,1,0 );
p->InsertNextPoint( 1,1,0 );
p->InsertNextPoint( 1,1,0 );
p->InsertNextPoint( 0,0,1 );
p->InsertNextPoint( 0,0,1 );
p->InsertNextPoint( 1,0,1 );
p->InsertNextPoint( 1,0,1 );
p->InsertNextPoint( 0,1,1.5 );
p->InsertNextPoint( 0,1,1.5 );
p->InsertNextPoint( 1,1,2 );
p->InsertNextPoint( 1,1,2 );
```

```cpp
tvtkStructuredGrid *sg = vtkStructuredGrid::New();
sg->SetDimensions (2,2,2);
sg->SetPoints(p);
```
vtkStructuredGrid

- Implicit Topology – (hexahedron)
- Explicit Geometry

\[
p = \text{vtk.vtkPoints()}
\]
\[
p.\text{InsertNextPoint}( 0,0,0 )
\]
\[
p.\text{InsertNextPoint}( 1,0,0 )
\]
\[
p.\text{InsertNextPoint}( 0,1,0 )
\]
\[
p.\text{InsertNextPoint}( 1,1,0 )
\]
\[
p.\text{InsertNextPoint}( 0,0,1 )
\]
\[
p.\text{InsertNextPoint}( 1,0,1 )
\]
\[
p.\text{InsertNextPoint}( 0,1,1.5 )
\]
\[
p.\text{InsertNextPoint}( 1,1,2 )
\]

\[
sg = \text{vtk.vtkStructuredGrid()}
\]
\[
sg.\text{SetDimensions} \ (2,2,2)
\]
\[
sg.\text{SetPoints}( p )
\]
vtkPolyData
vtkPolyData

- Geometry and Topology both explicit
- **Cells** are subdivided in four classes: Verts, Lines, Polys, Strip

```cpp
tvtkCellArray *quad = vtkCellArray::New();
quad->InsertNextCell( 4 );
quad->InsertNextCell( 4 );
quad->InsertCellPoint( 3 );
quad->InsertCellPoint( 2 );
quad->InsertCellPoint( 6 );
quad->InsertCellPoint( 7 );
// create polyline cell with indexes 0,1,5,4,0
```

```cpp
tvtkPolyData *pd = vtkPolyData::New();
pd->SetPoints( p );
pd->SetPolys ( quad );
pd->SetLines ( polyline );
```
quad = vtk.vtkCellArray()
quad.InsertNextCell( 4 )
quad.InsertCellPoint( 3 )
quad.InsertCellPoint( 2 )
quad.InsertCellPoint( 6 )
quad.InsertCellPoint( 7 )

polyline = vtk.vtkCellArray()
polyline.InsertNextCell( 5 )
polyline.InsertCellPoint( 0 )
polyline.InsertCellPoint( 1 )
polyline.InsertCellPoint( 5 )
polyline.InsertCellPoint( 4 )
polyline.InsertCellPoint( 0 )
vtkPolyData

```python
pd = vtk.vtkPolyData()
pd.SetPoints(p)
pd.SetPolys(quad)
pd.SetLines(polyline)
```
(e) Polygonal Data
vtkUnstructuredGrid
vtkUnstructuredGrid

- Geometry and Topology both explicit
- Celle can be 0,1,2 or 3D

```c
vtkIdList *il = vtkIdList::New();
il->InsertNextId( 1 );
il->InsertNextId( 2 );
il->InsertNextId( 3 );
il->InsertNextId( 7 );
```

```c
vtkUnstructuredGrid *ug = vtkUnstructuredGrid::New();
ug->SetPoints( p );
ug->InsertNextCell( VTK_TETRA, il );
```

see vtkCellType.h
vtkUnstructuredGrid

- Geometry and Topology both explicit
- Cells can be 0, 1, 2 or 3D

```python
il = vtk.vtkIdList()
il.InsertNextId( 1 )
il.InsertNextId( 2 )
il.InsertNextId( 3 )
il.InsertNextId( 7 )

ug = vtk.vtkUnstructuredGrid()
ug.SetPoints( p )
ug.InsertNextCell( vtk.VTK_TETRA, il )

see vtkCellType.h
```
Cell types

- **0d**
  - Vertex
  - PolyVertex

- **1d**
  - Line
  - PolyLine

- **2d**
  - Triangle
  - TriangleStrip
  - Quadrilateral
  - Pixel
  - Polygon

- **3d**
  - TetraHedron
  - Pyramind
  - HexaHedron
  - Voxel
  - Wedge
Cell types

- Non linear Cells

- Quadratic Edge
- Quadratic Triangle
- Quadratic Quadrilateral

- Quadratic Tetrahedron
- Quadratic Hexahedron
Data querying

• Geometry
  – GetNumberOfPoints, GetPoint, FindPoint
  – GetCenter, GetBounds, GetLength,

• Topology
  – GetNumberOfCells, GetCell, FindCell, IntersectWithLine
  – GetPointCells, GetCellPoints, GetCellNeighbors

• Attributes
  – GetScalarRange
  – GetScalar, GetVector ....
  – EvaluatePosition
Supported formats

• **Reader/Writer** – works only on one data
  – Native VTK format (ASCII, Binary, XML)
  – Images: BMP, JPEG, TIFF, PNG, PNM, RAW (also 3D), DEM, GESigna
  – Surfaces: STL, MCubes, PLY
  – Volumes: Plot3D, SLC, UGFacet
  – Other: Particles

• **Importer/Exporter** – works only on the scene
  – Import : 3DS, VRML
  – Export : IVO, OBJ, OOGLE, RIB, VRML
Data Import

Strategies:

• “ASCII ART”
  – The VTK ASCII format is really simple, in some cases you have only to add a header to the data and transform it in VTK.

• Create VTK data programmatically
  – If you are able to write a program that is able to read the data to be imported, can be created a VTK data type as seen in the previous slides (Programmable Source)

• Build a Reader
  – In case of frequent usage, building a reader is the best way to proceed, but also the more expensive. At the end it can be donated to the community.
Pipeline

data-flow paradigm

• Create a visualization using VTK means:
  – Find out in the VTK libraries the necessary filters
  – Link them together (this is called **pipeline**)
    In simple cases the pipeline will be a linear chain, while in more complex cases it can be a graph.

• The pipeline ends with a Window object
  Showing this window, we will see the first result of the elaboration; you can then pass to the interactive phase that allows you:
  – Change the object’s properties or how they are linked
  – Evaluate the obtained result eventually go back to the previous value.

• No more code is strictly required.
  (execution demand driven)
Filters

- A **Filter** is an object that can elaborate a data, in particular receive a data from its **input**, elaborate it considering its **parameters**, gives the result using its **output**.

In some cases, filters don’t have inputs (**Readers**, **Source**) or don’t have the output (**Writer**, **Mapper**)

- **Multiple Input / Output**
- **Multiple Fan-Out**

- **Reader**
  - Read a data from a file

- **Filter 1**
  - Apply a certain transformation

- **Filter 2**
  - Apply a second transformation

- **Developer doesn’t create data**
Mapper and Actors

- In general a chain of filters end with two objects: the **Mapper** and the **Actor**.
- The **Mapper** specify interface between data and graphics primitives.
- The **Actor** represents one of the objects shown into the window. The Actor is always linked to a Mapper.
The pipeline visualization happen using the following objects:

- The **Renderer** receives one or more actors and represents “the visualized scene”.
- The **RenderWindow** represents the window that you see on the screen and contains the scene.
- The **RenderWindowInteractor** add the interactivity, the possibility to manage the Mouse events. By default the interactor allows you to change the scene point of view.
Complete Pipeline

Visualization Pipeline

Graphics Pipeline

Reader
Filter
Mapper
Actor
R
RW
RWI
Objects Created Implicitly

- Reader
- Filter
- Mapper
- Actor
- Transform
- Property
- Camera
- R
- Light
- RW
- InteractorStyle
- RWI

Manage the visual aspect of the actor
Manage the perspective
Manage the actor’s position
Light color, type, position
How to react to user commands
Techniques and Optimization
Techniques and Optimization

- Filter’s connection
- Reference Counting
- Pipeline execution
- Data sharing
- Caching
- Streaming
- Multithreading
Filters connection

Let’s see what there’s behind this diagram

**STEP 1:**
create A,
A create its own Output

**STEP 2:**
create B,
B create its own Output

**STEP 3:**
Link B with A,
B->SetInputConnection(A->GetOutputPort())
What happen if A is deleted?

- Each VTK object has internally a counter called “Reference Count” (RC), that keep trace of how many other objects have a pointer to it.
- This counter can be changed using two methods Register and Unregister.
- Normally these methods are invoked as side-effect of New, Delete and SetXYZMethod.
New and Delete

- **New and Delete** are invoked by an entity external to the object. It's implicit that if this entity creates the object, than want to use it, so objects are created with RC=1.

```
vtkFilter::New()
{Register, ...}
```

- When the created the object don't want to use it anymore, will call the **Delete** method of the object:

```
vtkFilter::Delete()
{
    Unregister,
    if( RC == 0 ) delete this
}
```
SetInputConnection

• All methods that links together objects (like SetInputConnection) have a prefix “Set” and are created starting from macros.

vtkFilterXYZ::SetInputConnection(vtkAlgorithmOutput *Input)
{
    ...
    I->Register(NULL);
    ...
    this->Modified();
}
Scenario

(1) A → Output A: RC=1
(2) A → Output A: RC=2
(3) A → Output A
(4) Output A: RC=1

(5) DataSet → B
    DataSet 2 → B

(6) DataSet → B: RC=0
    DataSet 2 → B
Pipeline execution

Pipeline execution is "On Demand" and is triggered by an "Update" request.

- Modified Time (MT)
- Executed Time (ET)
- Update
- Execute
Pipeline execution

1. **Filter::Update()**
   
   \[
   \text{Input->Update();}
   \]
   
   \[
   \text{if( Input->ET > ET || MT > ET ) Execute();}
   \]

2. **DataSet::Update()**

   \[
   \text{if( Source) Source->Update();}
   \]

3. **Reader::Update()**

   \[
   \text{if( MT > ET ) Execute();}
   \]

4. **Reader::Update()**

   \[
   \text{if( MT > ET ) Execute();}
   \]
Data sharing

- The Reader terminated its execution and has “filled” G,T,A of its Output.
- Now the filter can Execute. For example the Filter create new attributes.
- Geometry and Topology are then the same on the dataset1, so are good also for the dataset2, we don’t need to copy. In dataset2 will be only the pointers to G and T.
- G and T are portions of shared data, the RefCounting take care of their time life.
Caching

We don’t need this data anymore
Streaming

- There are algorithms that can execute on a single portion of the entire data.
- In this case the Pipeline execution happen in more than one phase, each time a different portion of the data is produced.
Multi-threading

VTK supports two types of multi-threading:

A) Parallelism of filters. The Exec is multi-threaded.

B) Parallelism between Filters. In case of elaborations using streaming, filters works on different portions of the same data.
Tools

- User Guide
- Examples (http://www.vtk.org/Wiki/VTK/Examples)
- Help
- Sources
- Wiki
- Mailing List
- Git / DashBoard / BugList
Thank you 😊
Credits 😊

Silvano Imboden

s.imboden@cineca.it