

# Python en calcul scientifique : visualisation avec VTK

Sylvain Faure

CNRS  
Université Paris-Sud

Laboratoire de Mathématiques d'Orsay

6-10 décembre 2010, Autrans

# En quoi consiste la visualisation ?

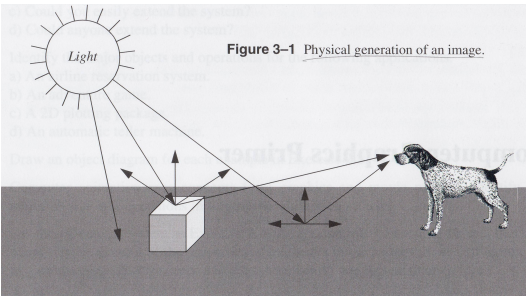
On souhaite :

- Représenter un phénomène.
- Mettre en évidence de façon visuelle un quelconque phénomène.

Dans un contexte de Calcul Scientifique, c'est la transformation de données numériques en image.

- Une simulation numérique menée sur plusieurs milliers de processeurs peut conduire à un gros volume de données.
- Il faut de l'interactivité et pouvoir envoyer ces images sur des murs d'images ou dans des salles de réalité virtuelle.

# Image physique



Le soleil émet des rayons lumineux dans toutes les directions. Certains rayons sont absorbés et nous voyons ceux qui sont réfléchis.

# Représentation d'une image physique sur un ordinateur

1/ Les procédés de type "image-order" tels que le lancer de rayons ("ray-tracing" ou "ray-casting") :

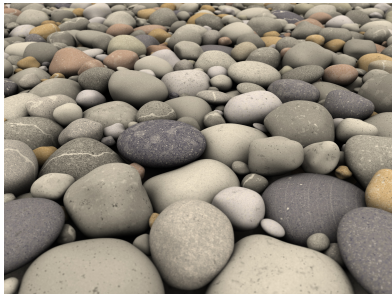
- Pour chaque pixel de l'image générée, on lance un rayon depuis le point de vue (i.e. la caméra) dans la scène 3D. Le premier point d'impact du rayon sur un objet définit l'objet concerné par le pixel correspondant.
- Des rayons sont ensuite lancés depuis le point d'impact en direction de chaque source lumineuse afin de déterminer sa luminosité. Cette luminosité combinée avec la couleur de l'objet ainsi que d'autres informations éventuelles (textures,...) déterminent la couleur finale du pixel.

Cette technique fonctionne exactement à l'inverse de la nature mais elle est nettement plus performante.

# Algorithme du lancer de rayon

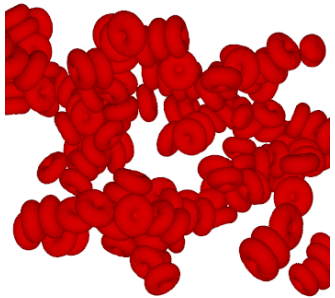
Entre la caméra et la scène on place un écran (sur lequel se trouvera l'image générée). Puisque l'écran est décomposé en pixels, on "lance" un rayon depuis la caméra par chaque pixel, et on "suit" ce rayon dans la scène. Cela revient à calculer l'équation d'une droite passant par la caméra et un pixel, puis par calcul géométrique à déterminer son intersection avec les différents objets de la scène. S'il n'intersecte rien, la couleur du pixel est celle du "fond". Sinon, on calcule la couleur de l'objet au point d'intersection. Cette couleur tient compte de l'ombre éventuelle (si la source lumineuse est cachée par un autre objet), de la couleur locale de l'objet et éventuellement des apports des autres objets de la scène (si l'objet est réfléchissant ou transparent). La détermination des ombres et des apports des autres objets nécessite de relancer d'autres rayons depuis le point d'intersection : vers les sources lumineuses pour l'ombre, grâce aux lois de Descartes pour la réflexion ou la transmission.

## Deux exemples...

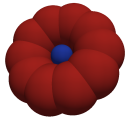


POV-Ray : <http://www.povray.org/>  
800 lignes de codes par Jonathan Hunt

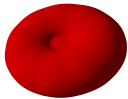
Inconvénient : le coût en temps de calcul



Simulation de globules rouges



avec VTK



avec Pov-Ray

# Représentation d'une image physique sur un ordinateur

## 2/ Les procédés de type "object-order" :

- La scène 3D est divisée en une série de coupes et chacun des voxels (cellules cubiques) de ces coupes est projeté et "écrasé" sur un plan temporaire parallèle au plan de projection. Ce dernier est ensuite combiné au plan image L l'aide d'un opérateur de composition. L'image est complète lorsque l'ensemble des voxels ont été projetés ou que celle-ci est complètement saturée et que l'ajout d'autres voxels n'aurait plus aucun effet sur l'image résultante.
- Technique beaucoup plus efficace permettant l'interactivité.

# Visualization ToolKit (VTK)

C'est une librairie écrite en C++ qui permet de visualiser des données, la première version date de 1994.

- Elle contient de nombreux algorithmes très performants agissant sur les données et plusieurs techniques de rendu (surfacique, volumique, gestion du niveaux de détails,...).
- C'est un système de type "data-flow". Le processus de visualisation utilisé par VTK est le suivant : les données brutes, introduites en début de pipeline, sont transformées et traitées par celui-ci de manière à donner une image.
- Il existe deux interfaces graphiques : Paraview et MayaVi.



<http://www.vtk.org/>  
<http://www.vtk.org/doc/nightly/html/>  
<http://www.vtk.org/mailman/listinfo/vtkusers>

# Que contient VTK ?

Le code source contient les dossiers suivants :

- *Common* : le coeur de VTK (*vtkDoubleArray*),
- *Filtering* : pour le pipeline des données (*vtkUnstructuredGrid*),
- *Rendering* : pour le rendu (*vtkRenderer*,...),
- *Graphics* : pour agir sur les données (*vtkPointDataToCellData*),
- *Imaging* : pour le traitement des images (*vtkImageShrink3D*),
- *Infovis* : pour visualiser des informations (*vtkSQLGraphReader*),
- *IO* : pour la lecture/écriture des données (*vtkXMLPolyDataReader*),
- *Views* : pour différents niveaux de vues (*vtkRenderView*),
- *VolumeRendering* : pour le rendu volumique (*vtkVolumeRayCastMapper*),
- *Hybrid* : objets complexes *Imaging* et *Graphics* (*vtkEarthSource*),
- *Widgets* : pour interagir avec les objets de la scène (*vtkBoxWidget*),
- *Parallel* : pour le parallélisme (*vtkPStreamTracer*),
- *GenericFiltering* : pour faciliter l'interaction (*vtkGenericContourFilter*),
- *Examples, Testing, Utilities, Wrapping* (*Tcl, Python* et *Java*),  
*Geovis...*

## Introduction

Les formats  
de fichiers  
VTK

Pipeline de  
visualisation

Objets VTK  
de base

TP

# VTK et le langage C++

```
#include "vtkRenderWindow.h"
#include "vtkSphereSource.h"
#include "vtkPolyDataMapper.h"
#include "vtkActor.h"
#include "vtkRenderer.h"
#include "vtkRenderWindowInteractor.h"
using namespace std;
int main () {
    vtkRenderWindow* fenetre = vtkRenderWindow::New();
    vtkSphereSource* sphere = vtkSphereSource::New();
    vtkPolyDataMapper* mappersphere = vtkPolyDataMapper::
        New();
    mappersphere->SetInput(sphere->GetOutput());
    vtkActor* acteursphere = vtkActor::New();
    acteursphere->SetMapper(mappersphere);
    vtkRenderer* ren = vtkRenderer::New();
    ren->AddActor(acteursphere);
    fenetre->AddRenderer(ren);
    vtkRenderWindowInteractor* iren =
        vtkRenderWindowInteractor::New();
    iren->SetRenderWindow(fenetre);
    iren->Initialize();
    fenetre->Render();
    iren->Start();
}
```

# VTK et le langage Java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import vtk.*;

public class VTKsphere extends JPanel {
    public VTKsphere() {
        setLayout(new BorderLayout());
        vtkCanvas fenetre = new vtkCanvas();
        add(fenetre, BorderLayout.CENTER);
        vtkSphereSource sphere = new vtkSphereSource();
        vtkPolyDataMapper mappersphere = new
            vtkPolyDataMapper();
        mappersphere.SetInput(sphere.GetOutput());
        vtkActor acteursphere = new vtkActor();
        acteursphere.SetMapper(mappersphere);
        fenetre.GetRenderer().AddActor(acteursphere);
    }

    public static void main(String s[]) {
        VTKsphere panel = new VTKsphere();
        VTKsphere panel2 = new VTKsphere();

        JFrame frame = new JFrame("VTK_Sphere");
        frame.getContentPane().setLayout(new GridLayout(2,1)
        );
        frame.addWindowListener(new WindowAdapter() {
```

# VTK et le langage Python

```
import vtk
fenetre = vtk.vtkRenderWindow()
sphere = vtk.vtkSphereSource()
mappersphere = vtk.vtkPolyDataMapper()
mappersphere.SetInput(sphere.GetOutput())
acteursphere = vtk.vtkActor()
acteursphere.SetMapper(mappersphere)
ren = vtk.vtkRenderer()
ren.AddActor(acteursphere)
fenetre.AddRenderer(ren)
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(fenetre)
iren.Initialize()
fenetre.Render()
iren.Start()
```

# Formats de fichiers *VTK*

Il existe deux familles de fichiers de données :

- Les formats de fichiers hérités du passé (“simple legacy formats”). Non abordés ici.
- Les formats *XML* (eXtensible Markup Language, successeur de l'*HTML*, c'est un métalangage permettant d'inventer à volonté de nouvelles balises).

# Fichiers *VTK* au format *XML*

## Avantages :

- Permettent le chargement sur une architecture parallèle.
- Permettent de compresser les données.

## Deux catégories :

- Structurés : ImageData (.vti), RectilinearGrid (.vtr), StructuredGrid (.vts), PImageData (.pvti), PRectilinearGrid (.pvtr), PStructuredGrid (.pvts).
- Non structurés : PolyData (.vtp), UnstructuredGrid (.vtu), PPolyData (.pvtp), PUnstructuredGrid (.pvtu).

## Fichiers *VTK* au format *XML* : *ImageData*

Les points et cellules sont définis implicitement.

```
<?xml version = "1.0"? >  
< VTKFile type = " ImageData" version = "0.1" >  
  < ImageData WholeExtent = " x1 x2 y1 y2 z1 z2"  
    Origin = " x0 y0 z0" Spacing = " dx dy dz" >  
    < Piece Extent = " x1 x2 y1 y2 z1 z2" >  
      < PointData Vectors = " velocity" Scalars = " pressure" >  
        < DataArray type = " Float32" Name = " velocity"  
          NumberOfComponents = " 3" format = " ascii" >  
          u0 v0 w0  
          ...  
        < /DataArray >
```

# Fichiers *VTK* au format *XML* : *ImageData*

```
< DataArray type = "Float32" Name = "pressure" format = "ascii" >  
  p0  
  ...  
< /DataArray >  
< /PointData >  
< CellData >  
  ...  
< /CellData >  
< /Piece >  
< /ImageData >  
< /VTKFile >
```

## Fichiers *VTK* au format *XML* : *RectilinearGrid*

Les points sont décrits par leurs coordonnées *< Coordinates >*.

```
<?xml version = "1.0"? >
```

```
< VTKFile type = " RectilinearGrid" version = "0.1" >
```

```
< RectilinearGrid WholeExtent = " x1 x2 y1 y2 z1 z2" >
```

```
< Piece Extent = " x1 x2 y1 y2 z1 z2" >
```

```
< PointData > ... < /PointData >
```

```
< CellData > ... < /CellData >
```

```
< Coordinates > ... < /Coordinates >
```

```
< /Piece >
```

```
< /RectilinearGrid >
```

```
< /VTKFile >
```

## Fichiers *VTK* au format *XML* : *StructuredGrid*

Les points sont décrits par leurs coordonnées *< Points >*.

```
<?xml version = "1.0"? >
```

```
< VTKFile type = " StructuredGrid" version = "0.1" >
```

```
< StructuredGrid WholeExtent = " x1 x2 y1 y2 z1 z2" >
```

```
< Piece Extent = " x1 x2 y1 y2 z1 z2" >
```

```
< PointData > ... < /PointData >
```

```
< CellData > ... < /CellData >
```

```
< Points > ... < /Points >
```

```
< /Piece >
```

```
< /StructuredGrid >
```

```
< /VTKFile >
```

## Fichiers *VTK* au format *XML* : *UnstructuredGrid*

Les points sont définis explicitement par `< Points >`. Les cellules sont définies par `< Cells >`.

```
<?xml version = "1.0"? >  
< VTKFile type = " UnstructuredGrid" version = "0.1" >  
  < UnstructuredGrid >  
    < Piece NumberOfPoints = " #" NumberOfCells = " #" >  
      < PointData > ... < /PointData >  
      < CellData > ... < /CellData >  
      < Points > ... < /Points >  
      < Cells > ... < /Cells >  
    < /Piece >  
  < /UnstructuredGrid >  
< /VTKFile >
```

## Fichiers *VTK* au format *XML* : *PolyData*

Les points sont définis explicitement par `< Points >`. Les cellules sont définies par `< Verts >`, `< Lines >`, `< Strips >` et `< Polys >`.

```
<?xml version = "1.0"? >
```

```
< VTKFile type = " PolyData" version = "0.1" >
```

```
< Piece NumberOfPoints = " #" NumberOfVerts = " #"
      NumberOfLines = " #" NumberOfStrips = " #"
      NumberOfPolys = " #" >
```

```
< PointData > ... < /PointData >
```

```
< CellData > ... < /CellData >
```

```
< Points > ... < /Points >
```

```
< Verts > ... < /Verts >
```

```
< Lines > ... < /Lines >
```

```
< Strips > ... < /Strips >
```

```
< Polys > ... < /Polys >
```

```
< /Piece >
```

```
< /PolyData >
```

```
< /VTKFile >
```

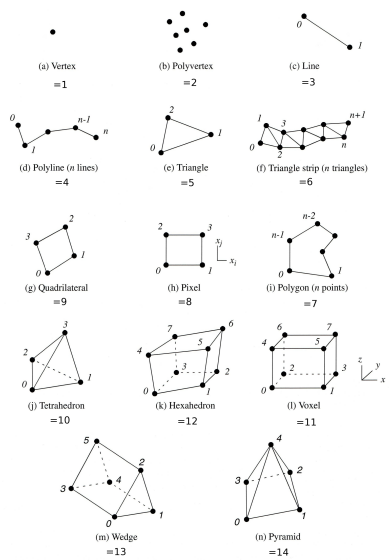


Figure 5-2 Linear cell types found in VTK. Numbers define ordering of the defining points.

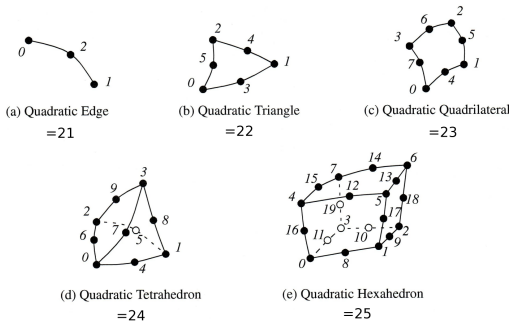


Figure 5-3 Non-linear cell types found in VTK.

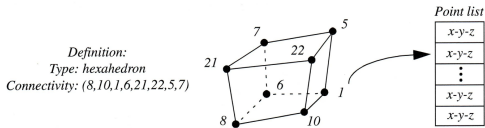


Figure 5-4 Example of a hexahedron cell. The topology is implicitly defined by the ordering of the point list.

## Fichiers *VTK* au format *XML* : *PlImageData*

Le fichier ci-dessous (*.pvti*) recolle les morceaux (*.vti*)

```
<?xml version = "1.0"? >  
< VTKFile type = " PlImageData" version = "0.1" >  
  < PlImageData WholeExtent = " 0 32 0 32 0 1"  
    GhostLevel = "0" Origin = " 0 0 0"  
    Spacing = " 0.03125 0.03125 0.0001" >  
    < PPointData >  
      < /PPointData >  
      < PCellData Vectors = " velocity" Scalars = " pressure" >  
        < DataArray type = " Float32" Name = " velocity"  
          NumberOfComponents = "3" format = " ascii" / >  
        < DataArray type = " Float32" Name = " pressure"  
          format = " ascii" / >  
      < /PCellData >
```

## Fichiers *VTK* au format *XML* : *PImageData*

```
< Piece Extent = "0 16 0 16 0 1" Source = " Gr4 - 000.001.vti" / >  
< Piece Extent = "16 32 0 16 0 1" Source = " Gr4 - 001.001.vti" / >  
< Piece Extent = "0 16 16 32 0 1" Source = " Gr4 - 002.001.vti" / >  
< Piece Extent = "16 32 16 32 0 1" Source = " Gr4 - 003.001.vti" / >  
< /PIImageData >  
< /VTKFile >
```

## Primitives graphiques

Une fois les données converties du format natif vers un format exploitable, on en extrait les caractéristiques les plus importantes sous forme de primitives :



**Polygon** — a set of edges, usually in a plane, that define a closed region. Triangles and rectangles are examples of polygons.



**Triangle Strip** — a series of triangles where each triangle shares its edges with its neighbors.



**Line** — connects two points.



**Polyline** — a series of connected lines.



**Point** — a 3D position in space.

Figure 3–19 Graphics primitives.

Cela évite de regarder les images pixels par pixels ! OpenGL (Open Graphics Library) est une interface regroupant plusieurs centaines de fonctions différentes permettant d'afficher des scènes tridimensionnelles complexes à partir de simples primitives. Elle est généralement implantée par les constructeurs

# Programmation orientée objet

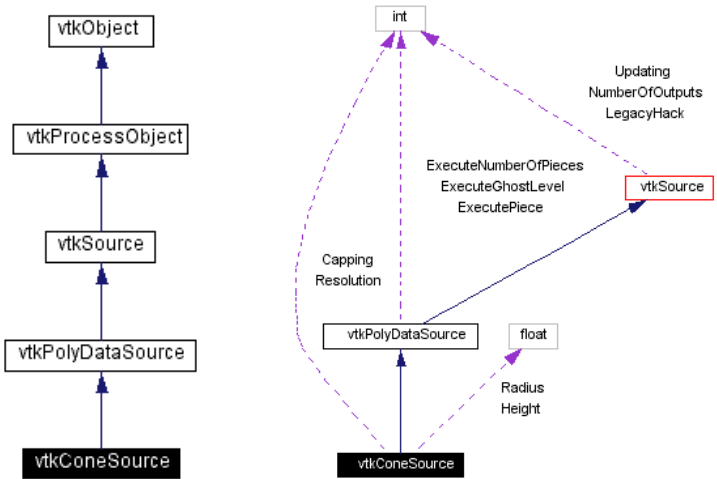
La librairie VTK est écrite en `C++` en respectant de façon rigoureuse les concepts de la programmation orientée objet.

C'est vital pour une librairie qui n'est utilisable qu'en parcourant l'arborescence des différentes classes la composant :

*[http : //www.vtk.org/doc/nightly/html/](http://www.vtk.org/doc/nightly/html/)*

- Notion d'héritage (en anglais inheritance) : permet de créer une nouvelle classe à partir d'une classe existante. La classe dérivée (i.e. la classe nouvellement créée) contient les attributs et les méthodes de sa superclasse (i.e. la classe dont elle dérive).
- Intérêt majeur : permet de définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées. On crée donc une hiérarchie de classes de plus en plus spécialisées.

# Programmation orientée objet



# Programmation orientée objet

## vtkConeSource

### Public Methods

```
virtual const char * GetClassName ()  
    virtual int IsA (const char *type)  
        void PrintSelf (ostream &os, vtkIndent indent)  
  
    virtual void SetHeight (float)  
    virtual float GetHeight ()  
  
    virtual void SetRadius (float)  
    virtual float GetRadius ()  
  
    virtual void SetResolution (int)  
    virtual int GetResolution ()  
  
        void SetAngle (float angle)  
        float GetAngle ()  
  
    virtual void SetCapping (int)  
    virtual int GetCapping ()  
    virtual void CappingOn ()  
    virtual void CappingOff ()
```

## vtkObject

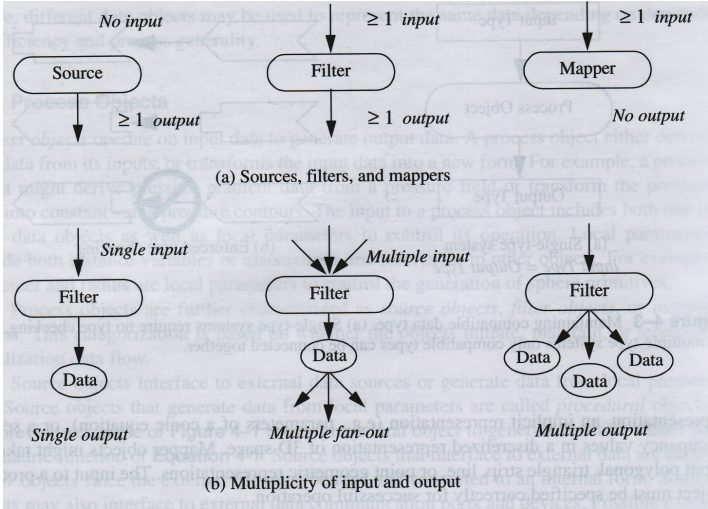
### Public Methods

```
virtual const char * GetClassName ()  
    virtual int IsA (const char *name)  
    virtual void Delete ()  
    virtual void DebugOn ()  
    virtual void DebugOff ()  
    unsigned char GetDebug ()  
        void SetDebug (unsigned char debugFlag)  
    virtual void Modified ()  
    virtual unsigned long GetMTime ()  
        void Print (ostream &os)  
        void Register (vtkObject *o)  
    virtual void UnRegister (vtkObject *o)  
        void SetReferenceCount (int)  
  
    virtual void PrintSelf (ostream &os, vtkIndent indent)  
    virtual void PrintHeader (ostream &os, vtkIndent indent)  
    virtual void PrintTrailer (ostream &os, vtkIndent indent)  
  
        int GetReferenceCount ()  
  
    unsigned long AddObserver (unsigned long event, vtkCommand *)  
    unsigned long AddObserver (const char *event, vtkCommand *)  
    vtkCommand * GetCommand (unsigned long tag)  
    void InvokeEvent (unsigned long event, void *callData)  
    void InvokeEvent (const char *event, void *callData)  
    void RemoveObserver (unsigned long tag)  
    int HasObserver (unsigned long event)  
    int HasObserver (const char *event)
```

## Pipeline et filtres : définitions

- Chaque classe de VTK représente un filtre avec entrée(s) et sortie(s).
- Le pipeline de visualisation est une succession cohérente de filtres reliés les uns aux autres permettant d'aboutir à une image.
- Exécution du pipeline : chaque filtre possède une méthode *Execute()* utilisée manuellement ou le plus souvent automatiquement. Il existe également une méthode *Update()* qui vérifie si les données intérieures au filtre sont bien à jour et les met à jour si nécessaire.
- Conséquence : un objet filtre ne doit jamais être effacé de la mémoire brutalement avec la commande *delete* du *C++*. Il faut toujours utiliser la méthode *Delete()* implanter dans le filtre.

# Pipeline et filtres : définitions



Réf. : *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics 4th Edition*  
(Kitware).

# Connecter les filtres pour former le pipeline

Pour construire le pipeline, on doit posséder des méthodes permettant de connecter les filtres :

- Pour connecter une entrée :  
*SetInputConnection(port, input)*
- Pour ajouter une entrée :  
*AddInputConnection(port, input)*
- Pour enlever une entrée :  
*RemoveInputConnection(port, input)*
- Pour récupérer une sortie :  
*GetOutputPort(port)*
- Remarque : on peut réaliser une connexion même si la sortie n'est pas encore executable (il peut par exemple manquer des données...).
- Usage classique :  
*filtre2 -> SetInputConnection(0, filtre1 -> GetOutputPort(0))*

# Connecter les filtres pour former le pipeline : ancien style

Une ancienne façon de connecter les filtres est encore bien présente dans de nombreux exemples :

*filtre2* → *SetInput(filtre1* → *GetOutput())*

Principal inconvénient : nécessite la connaissance des types de données au moment de la connection. C'est problématique quand un filtre produit une multitude de sorties car cela oblige de reformer le pipeline à chaque changement de type de sortie.

Comparaison :

- *void vtkPolyDataAlgorithm :: SetInput(int, vtkDataObject \* )*
- *virtual void vtkAlgorithm ::  
SetInputConnection(int, vtkAlgorithmOutput \* )*

# Types de filtres

Il existe deux types d'objets en VTK :

- “Data Objects” : pour charger les données à partir de multiples types de fichiers par exemple (voir *vtkDataSetSource*)
- “Process Objects” : pour modifier les données entrées et donc les utiliser !

Il est bien entendu possible d'implanter ses propres filtres à condition de définir au minimum les méthodes *SetInputConnection*, *GetOutputPort*, et *Execute...*

## Exécution et mise à jour du pipeline

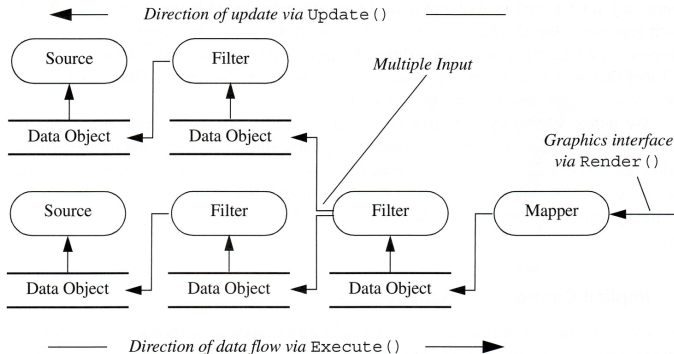
L'exécution du pipeline est contrôlée de façon implicite par VTK : chaque fois qu'une sortie est requise par un filtre. Les filtres se mettent à jour d'eux mêmes ("en remontant le pipeline et récursivement") et du coup tout le pipeline est à jour.

Avantages :

- L'utilisateur n'a pas besoin de gérer l'exécution du pipeline. Dans certains cas, il peut cependant être amené à utiliser la méthode *Update*.
- Les filtres sont dynamiques : la parallélisation est rendue plus facile.
- La mise à jour des filtres comporte des étapes qui peuvent être asynchrones

Remarque : on peut boucler un pipeline, dans ce cas il faudra utiliser *Update* pour exécuter la boucle.

## Exécution et mise à jour du pipeline



**Figure 4-12** Description of implicit execution process implemented in VTK. The Update() method is initiated via the Render() method from the actor. Data flows back to mapper via Execute() method. Arrows connecting objects indicate direction of Update() process.

Réf. : *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics 4th Edition (Kitware).*

## Exemple : visualisation d'une sphère

- On crée une sphère (*vtkSphereSource*) de centre  $(0, 1, 0)$ , de rayon 1.
- A l'aide de *vtkPolyDataMapper*, *vtkActor*, *vtkRenderer*, *vtkRenderWindow* et *vtkRenderWindowInteractor* on réalise le pipeline graphique le plus simple possible permettant de visualiser et d'interagir avec cette sphère.
- On fait varier les paramètres  $\theta$  et  $\varphi$  de la sphère afin d'améliorer sa résolution.

⇒ Exemple *SimpleSphere.py*.

## Exemple : une sphère colorée

Sylvain  
Faure

CNRS  
Université  
Paris-Sud

Laboratoire  
de Mathé-  
matiques  
d'Orsay

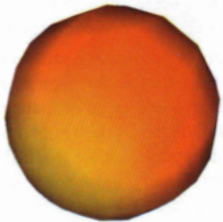
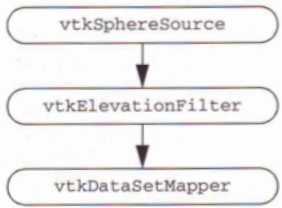
Introduction

Les formats  
de fichiers  
VTK

Pipeline de  
visualisation

Objets VTK  
de base

TP



```
vtkSphereSource *sphere = vtkSphereSource::New();
sphere->SetPhiResolution(12); sphere->SetThetaResolution(12);

vtkElevationFilter *colorIt = vtkElevationFilter::New();
colorIt->SetInputConnection(sphere->GetOutputPort());
colorIt->SetLowPoint(0,0,-1);
colorIt->SetHighPoint(0,0,1);

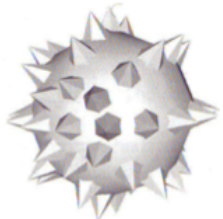
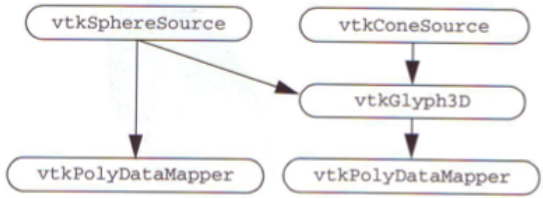
vtkDataSetMapper *mapper = vtkDataSetMapper::New();
mapper->SetInputConnection(colorIt->GetOutputPort());

vtkActor *actor = vtkActor::New();
actor->SetMapper(mapper);
```

Exemple *ColoredSphere.py*.

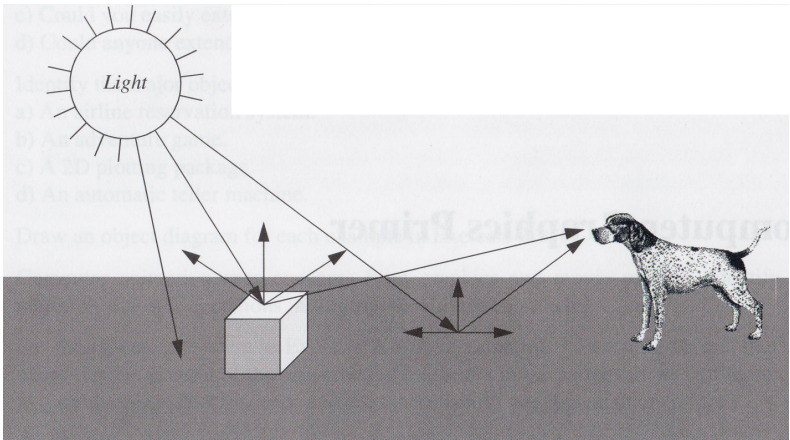
# Exemple : un pipeline avec entrées et sorties multiples.

On réalise le pipeline suivant :



⇒ Exemple *Chardon.py*.

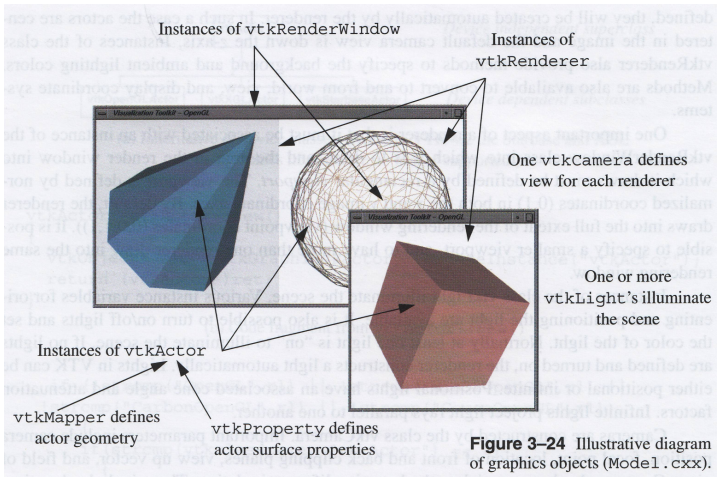
# Comme dans un film



## Les objets indispensables

- *vtkRenderWindow* : gère la (ou les) fenêtre(s) d'affichage.
- *vtkRenderer* : coordonne le rendu en utilisant les sources de lumières, les caméras et bien sûr les acteurs.
- *vtkLight* : définit une source de lumière illuminant la scène.
- *vtkCamera* : définit une source de lumière illuminant la scène.
- *vtkActor* : définit un acteur de la scène.
- *vtkProperty* : définit les propriétés d'apparences d'un acteur : couleur, transparence, comportement par rapport aux lumières.
- *vtkMapper* : définit la représentation géométrique d'un ou de plusieurs acteurs.

# Les objets indispensables



## *vtkRenderWindow*

C'est un objet abstrait spécifiant le comportement de la fenêtre d'affichage, ses principales caractéristiques sont :

- sa taille.
- sa résolution (bits par pixel).
- sa position.
- son nom.

Introduction

Les formats  
de fichiers  
VTK

Pipeline de  
visualisation

Objets VTK  
de base

TP

## *vtkRender*

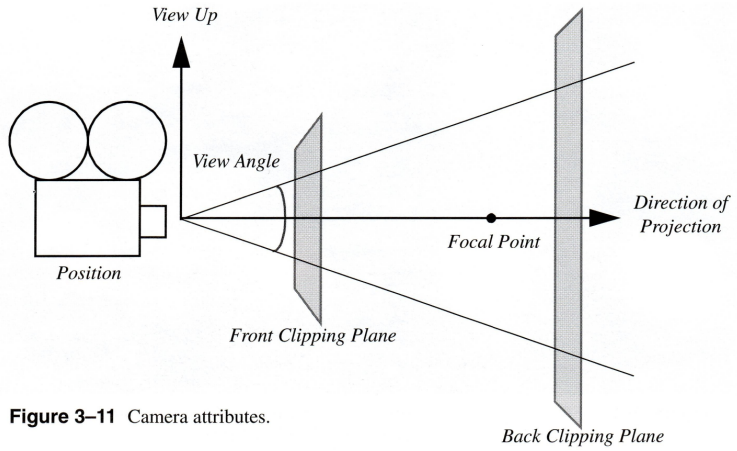
Prend en compte les lumières, caméras et acteurs afin de faire le rendu et de produire une image.

- Au moins un acteur doit être défini (lumières et caméras seront créées automatiquement si elle ne sont pas définies explicitement).
- Permet également de spécifier le fond de l'image et la lumière d'ambiance.
- Nécessite évidemment une fenêtre d'affichage.

⇒ Exemple *Rendu.py*.

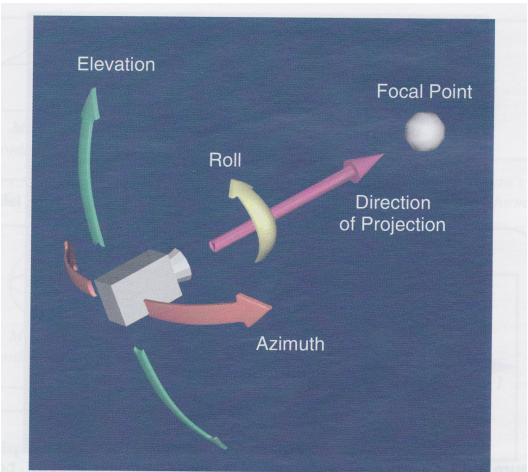
# vtkCamera

Positionne et oriente la caméra.



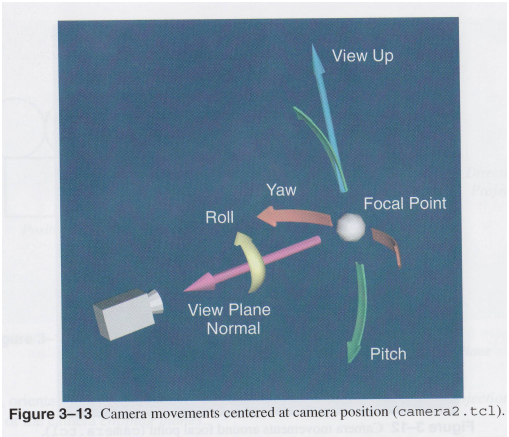
**Figure 3–11** Camera attributes.

La caméra bouge autour de la scène :



**Figure 3-12** Camera movements around focal point (`camera.tcl`).

La scène bouge autour de la caméra :



## Exemple : mouvement d'une caméra

- On affiche un cylindre (*vtkCylinderSource*) de centre  $(0, 0, 0)$ , de rayon 1, de hauteur 4 et de résolution 20.
- On définit une caméra (*vtkCamera*) placée en  $(0, 0, -10)$  et filmant dans la direction  $(0, 1, 0)$ . On prend  $(0, 0, 0)$  comme point focal.
- On associe cette caméra à l'objet *vtkRenderer* en vérifiant que la caméra est bien à l'emplacement prévu.
- On déplace la camera progressivement depuis la position  $(0, 0, -10)$  jusqu'à la position  $(0, 0, 0)$  puis on revient en  $(0, 0, -10)$
- On fait varier son paramètre "azimuth" entre 0 et 10.
- On fait varier son paramètre "roll" entre 0 et 10.
- On fait varier son paramètre "yaw" entre 0 et 5.

⇒ Exemple *Camera.py*.

## *vtkLight*

Nécessaire pour éclairer la scène, par défaut il y a une lumière d'ambiance. Il existe deux types de lumière : une lumière infinie (tout les rayons sont parallèles) et un spot de lumière (i.e. un cône de lumière). Les principaux réglages sont :

- l'orientation.
- la position.
- la possibilité d'allumer et d'éteindre.
- la couleur de la lumière.
- l'angle et le coefficient d'atténuation (pour un spot de lumière).

## Exemple : définition d'un éclairage

- On crée une sphère de centre  $(0, 1, 0)$  et de rayon 1.
- On la colorie avec la couleur  $(1, 0, 0)$ .
- On récupère la lumière ambiante associée par défaut à l'"acteur sphère" et on modifie son intensité ( $-10\%$ ,  $-20\%, \dots, -100\%$ ) avant de la ramener à son état initial.
- On crée une lumière de type lumière infini, de couleur  $(0, 0, 1)$ , et placée au même endroit que la caméra.
- On remplace la lumière précédents par une lumière de type spot, de couleur  $(0, 0, 1)$ , d'angle 5.

⇒ Exemple *Light.py*.

## *vtkRenderWindowInteractor*

Permet d'interagir avec la figure en contrôlant la caméra et les acteurs. Les principales touches permettant d'interagir :

$e$ : ferme la fenêtre	$r$ : revient à la position initiale
$s$ : mode "surface"	$w$ : mode "wireframe" (pour voir le maillage)

On bouge la caméra (mode caméra, touche  $c$ ) ou les acteurs (mode acteurs, touche  $a$ ).

Il y a aussi deux modes pour utiliser la souris : touche  $j$  pour le mode "joystick" (mouvement continu durant aussi longtemps que l'on appuie sur le bouton) et  $t$  pour le mode "trackball" (mouvement uniquement quand on appuie sur le bouton et que le pointeur bouge).

Remarque : on peut définir son propre interacteur.

⇒ Exemple *ColoredSphere.py*.

## *vtkLODActor* au lieu de *vtkActor*

Ajuste dynamiquement le niveau de détails de l'image (Level Of Details). Usage basique : *vtkActor* devient *vtkLODActor* !

Pour l'instant, une méthode très simple, basée sur *Temps\_Rendu/Nb\_Acteurs*, est utilisée pour déterminer quel niveau de détails doit être utilisé. Cela devrait bientôt être plus élaboré...

Il y a par défaut trois niveaux de détails :

- le niveau maximal.
- le niveau minimal : une simple boîte délimitant l'acteur (*vtkOutlineFilter*).
- le niveau intermédiaire : on fixe un nombre de points (positionnés aléatoirement) maximal. (*vtkMaskPoints*).

Il est possible de définir des niveaux de détails supplémentaires avec la méthode *AddLODMapper*.

⇒ Exemple *LODActor.py*.

## Exemple : lecture d'un fichier de données

- On lit le fichier *cow.g* à l'aide de *vtkBYUReader* puis on affiche la structure des données qu'il contient.
- A l'aide de *vtkPolyDataMapper*, *vtkActor*, *vtkRenderer*, *vtkRenderWindow* et *vtkRenderWindowInteractor* on réalise le pipeline graphique le plus simple possible permettant de visualiser et d'interagir avec l'objet décrit dans le fichier "cow.g".

⇒ Exemple *Vache.py*.

## *vtkTexture*

- Permet de rendre des objets plus réalistes.
- Une texture peut être un algorithme (texture procédurale) ou être constituée d'un tableau de pixels (une image bitmap par exemple) que l'on va appliquer sur une surface (ou un volume 3D).
- Il faut une transformation géométrique pour "peindre" l'objet sur lequel on veut mettre la texture.
- Il est souvent nécessaire de recourir à des algorithmes permettant de lisser l'image de façon à atténuer les artefacts liés au grossissement linéaire des pixels.

Exemple : lecture d'un fichier au format .bmp (*vtkBMPReader*) afin de l'utiliser comme texture (*vtkTexture*) sur un plan (*vtkPlaneSource*).

## *vtkOutlineFilter*

- Dessine une boîte qui délimite la zone où sont situées toutes les données.
- Variante : *vtkOutlineCornerFilter*

Exemple : Lecture du fichier *cow.g* afin d'afficher l'objet qu'il décrit et de tracer la boîte qui le contient.

⇒ Exemple *Vache\_texture\_outline.py*.

## Références : *Paraview*

- *Paraview Guide* :  
[http : // www.kitware.com/products/paraviewguide.html](http://www.kitware.com/products/paraviewguide.html)
- [http : // www.paraview.org](http://www.paraview.org)
- Mailing list  
[http : // www.paraview.org/HTML/MailingList.html](http://www.paraview.org/HTML/MailingList.html)
- Wiki (site web dynamique) :  
[http : // www.paraview.org/Wiki/ParaView](http://www.paraview.org/Wiki/ParaView)
- FAQ : [http : // www.paraview.org/Wiki/ParaView : FAQ](http://www.paraview.org/Wiki/ParaView:FAQ)

## Références : VTK

- *Visualization Toolkit Book* et *VTK User's Guide* :  
*[http : // www.vtk.org / buy – books.php](http://www.vtk.org/buy-books.php)*
- *[http : // www.vtk.org](http://www.vtk.org)*
- Mailing list :  
*[http : // public.kitware.com / mailman / listinfo / vtkusers](http://public.kitware.com/mailman/listinfo/vtkusers)*
- Wiki : *[http : // www.vtk.org / Wiki / VTK](http://www.vtk.org/Wiki/VTK)*
- FAQ : *[http : // www.vtk.org / Wiki / VTK \\_ FAQ](http://www.vtk.org/Wiki/VTK_FAQ)*
- Documents techniques (formats des fichiers de données,...) : *[http : // www.vtk.org / documents.php](http://www.vtk.org/documents.php)*

# TP planètes

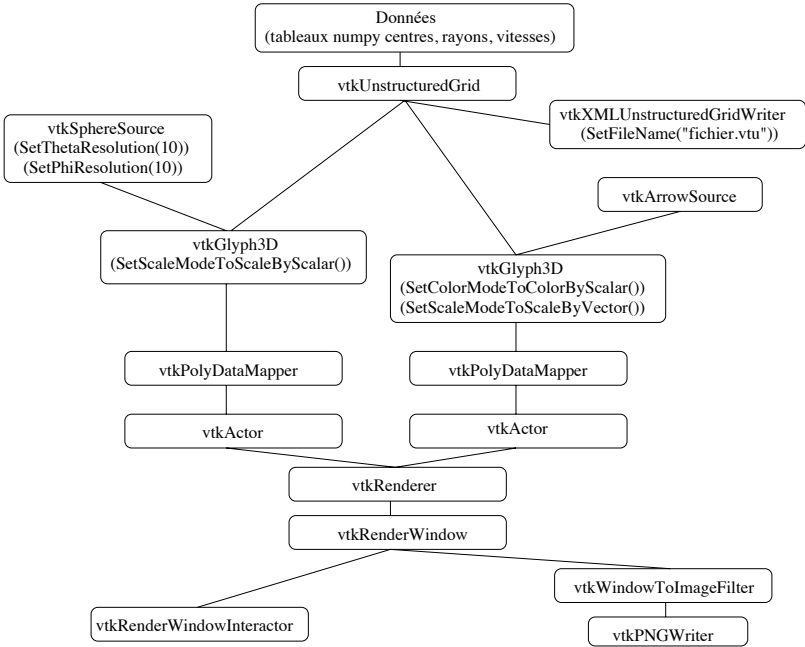
**Partie 1.** On considère les données suivantes associées à 3 sphères :

```
import vtk
import numpy
centres=numpy.array
    ([[ -2. , 1. , 4. ], [ -1. , 0. , 2. ], [ 1. , 1. , -1. ]])
rayons=numpy.array ([ 0.2 , 0.1 , 0.5 ])
vitesses=numpy.array
    ([[ 0.5 , 0. , 0.2 ], [ 0. , 0.2 , 0. ], [ 0. , -0.2 , 1. ]])
```

Créer un objet *VTK* de type *vtkUnstructuredGrid* contenant ces données.

**Partie 2.** Réaliser le pipeline graphique décrit dans le transparent suivant.

**Partie 3.** Intégrer cela dans le code simulant le système solaire.



# TP équation de la chaleur

**Partie 1.** Construire un objet *VTK* de type *vtkStructuredGrid* contenant les données issues de la résolution de l'équation de la chaleur i.e. les coordonnées des points du maillage et les valeurs de la température en chacun de ces points.

**Partie 2.** Ajouter au code de calcul le pipeline graphique décrit par le transparent suivant.

