The Visualization Pipeline

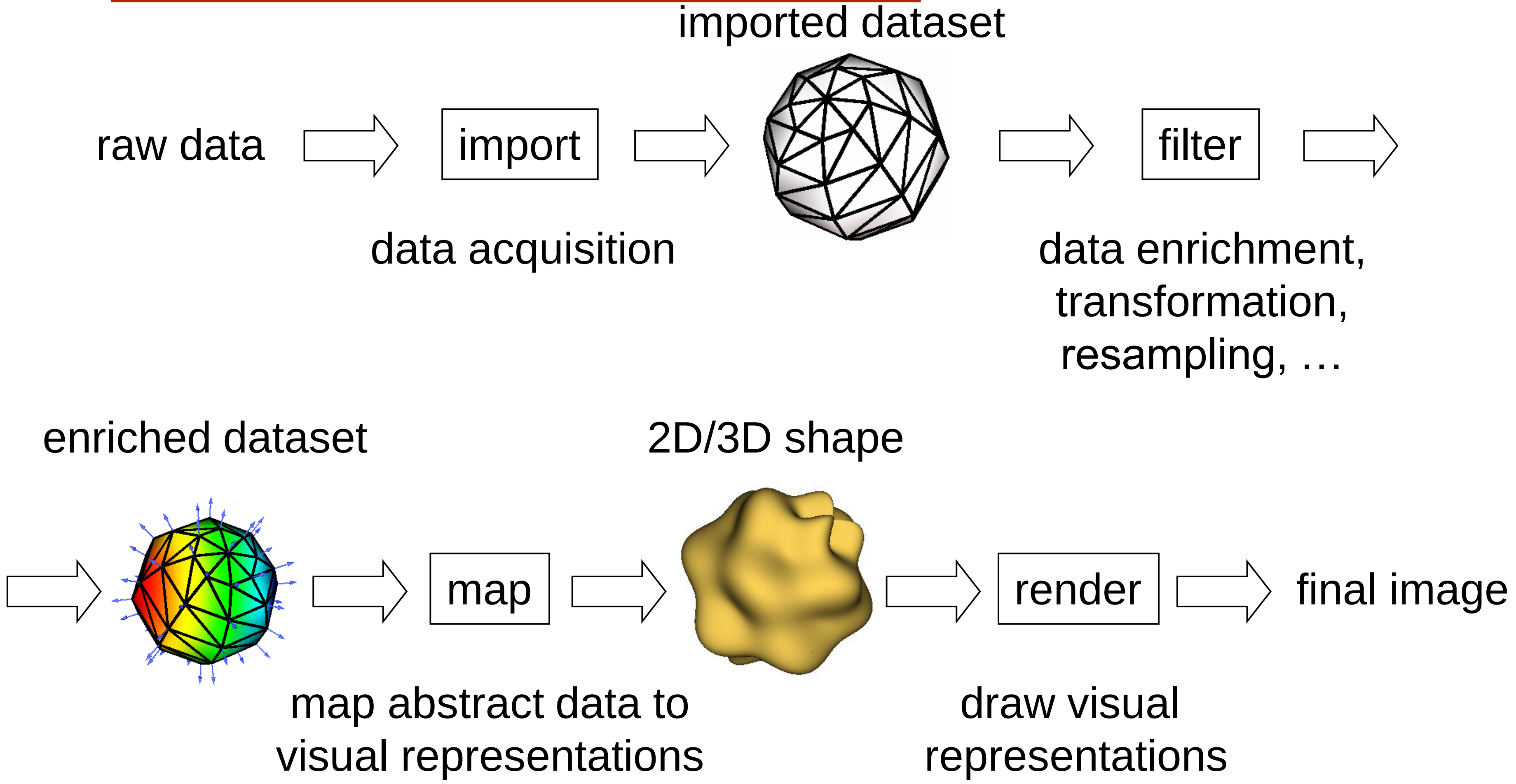
Motivation

The role of visualization is to create images that convey various types of insight into a given process. The visualization process consists of the sequence of steps, or operations, that manipulate the data produced by the process under study and ultimately deliver the desired images. On both the conceptual and the design level, this divide-and-conquer strategy in designing visualizations allows one to manage the complexity of the whole process. On the implementational level, this strategy allows us to construct visualizations by assembling reusable and modular data processing operations, much as in the field of software engineering.

Motivation

Given this modular decomposition, the visualization process can be seen as a pipeline consisting of several stages, each modeled by a specific data transformation operation. The input data flows through this pipeline, being transformed in various ways, until it generates the output images. Given this model, the sequence if data transformations that take place in the visualization process is often called the visualization pipeline.

Visualization Pipeline


imported dataset
raw data
import
filter
data acquisition
data enrichment, transformation, resampling, …
enriched dataset
2D/3D shape
map
render
final image
map abstract data to visual representations
draw visual representations

Images courtesy of Alexandru Telea

Visualization Pipeline (continued)

Importing Data

First, we have to import the data. This implies finding a representation of the original information we want to investigate in terms of a data set, be it continuous or discrete. Practically, importing data means choosing a specific dataset implementation and converting the original information to the representation implied by the chosen dataset. Ideally, this is a one-to-one mapping or data copying.

Visualization Pipeline (continued)

Importing Data (continued)

It is important to realize that the choices made during data importing determine the quality of the resulting images, and thus the effectiveness of the visualization. For example, changing the underlying grid structure from quads to triangles changes the interpolation method which for some visualization algorithm changes the resulting image. For this reason, the data importing step should try to preserve as much of the available input information as possible, and make as few assumptions as possible about what is important and what is not.

Visualization Pipeline (continued)

Data Filtering and Enrichment

We have to decide which are the data’s important aspects, or features, we are interested in. In most cases the imported data is not one-to-one with the aspects we want to get insight into. We must distill our raw data sets into more appropriate representations, also called **enriched datasets**, which encode our features of interest. This process is called **data filtering** or **data enriching**. On the one hand, data is filtered to extract relevant information. On the other hand, data is enriched with higher-level information that supports a given task.

Visualization Pipeline (continued)

Data Filtering and Enrichment (continued)

For example, medical specialists are usually interested in seeing only specific anatomical structures related to a certain condition, which are a subset of the entire dataset they obtain from scanning devices, such as CT or MRI scanners. (Note that volumetric representations of such scans already are in a filtered form based on the X-ray data.)

Visualization Pipeline (continued)

Mapping Data

The filtering operation produces an enriched dataset that should directly represent the features of interest for a specific exploration task. Once we have this representation, we must map it to the visual domain. We do this by associating elements of the visual domain with the data elements present in the enriched dataset. This step of the visualization process is called **mapping**. The visual domain is a multidimensional space whose axes, or dimensions are those elements we perceive as quasi-independent visual attributes, such as shape, position, size, color, texture, illumination, and motion.

Visualization Pipeline (continued)

Mapping Data (continued)

Typically, a visual feature is a colored, shaded, textured, or animated 2D or 3D shape. Data mapping is probably the operation in the visualization pipeline that is most characteristic for the visualization process as it influences the resulting image more than any other step. There are many different mapping techniques the visualization can be based on, which we will illustrate in the following chapters by introducing various visualization algorithms.

Visualization Pipeline (continued)

Rendering Data

The rendering operation is the final step of the visualization process. Rendering takes the 3D scene created by the mapping operation, together with several user-specified viewing parameters such as the viewpoint and lighting, and renders it to produce the desired images. In typical visualization applications, viewing parameters are considered part of the rendering operation. This allows users to interactively navigate and examine the rendered result of a given visualization. Indeed if the viewpoint changes but the 3D scene produced by the mapping stays the same, all we have to do is render the scene anew with the new viewing parameters, which is a relatively cheap operation.

Visualization Pipeline (continued)

Implementation Perspective

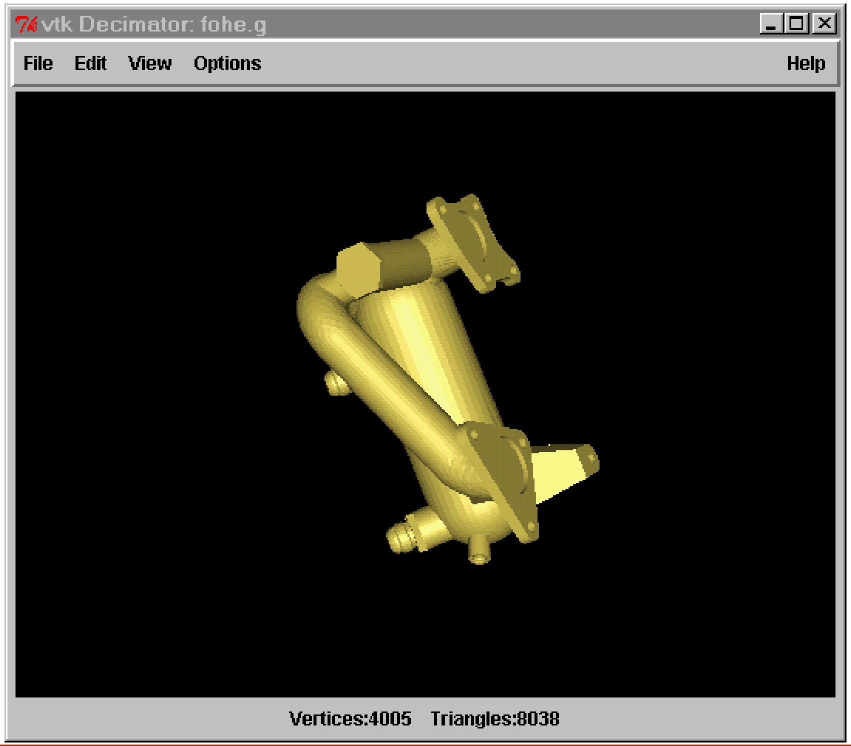
Since the individual steps of the visualization pipeline can be encapsulated in individual modules, many visualization software packages, such as VTK, provide **filters** that can take data as input and convert it in some way. For example, VTK implements generic `setInput` and `getOutput` methods so that the filters can be combined in almost any way.

Each filter implements a method called `Execute`. This method does all the computations. It can be invoked manually or automatically by the following filter whenever that filter needs its input.

Visualization Pipeline (continued)

Implementation Perspective (continued)

All filters combined than produce an interactive visualization of the input data:



Vertices:4005 Triangles:8038

Visualization Pipeline (continued)

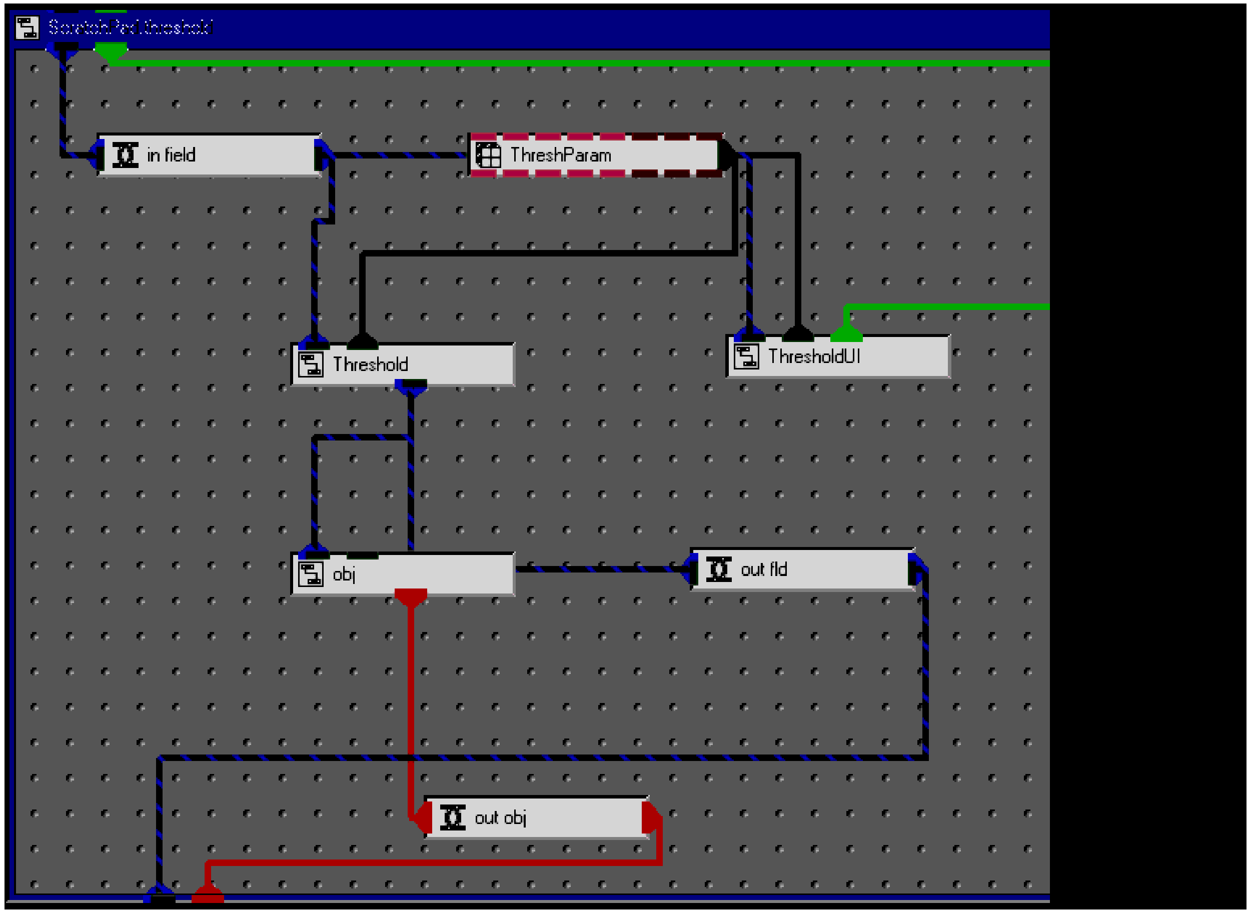
Implementation Perspective (continued)

Multiple visualization applications based on VTK are available, such as VolView or ParaView. Some approaches even allow a user to interactively set up the visualization pipeline to create the resulting visualization, for example VISSION by Telea and van Wijk. A similar approach is followed by AVS and Aviso/Amira, even though not based on VTK, where the user can combine different building blocks to form the application network.

Visualization Pipeline (continued)

Implementation Perspective (continued)

AVS:



Visualization Pipeline (continued)

Implementation Perspective (continued)

Other software packages implement the different modules as algorithms that can manipulate the data or generate new data or generate a visual representation of the data. For example, FAnToM developed by researchers at the Universities of Kaiserslautern and Leipzig follows this approach. Initially developed for vector data, it now supports various different types of data, such as scalar, vector, and tensor data.

Implementation Perspective: VTK

- Object oriented programming
- VTK pipeline
- Example

Object Oriented Programming

VTK uses object oriented programming

- Impossible to Cover in 10 minutes … but

- Traditional programming (i.e. C, Fortran, Matlab) is procedural:

  If we have matrix a

  to print it we do   print(a), disp(a) etc.,

- Data/Procedures are separate, data is dumb

Object Oriented Programming (continued)

- In OOP data is “intelligent” i.e. data structures encapsulate procedures i.e.

- If we have matrix a

  to print it we do a.print()

- Procedures are embedded within the data structure definition (called methods)
- Lots of good reasons for this …

Object Oriented Programming (continued)

Data structures are classes, e.g. in C++

```
class Matrix3x3 {
	float a[3][3];

public:
	void Print();
	void Invert();
	void Load(char* filename);
	etc.
}
```

To use:

```
Matrix3x3 a;
a.Load("a.matr");
a.Print();
```

Object Oriented Programming - Inheritance

Consider now need to change file format for matrix:

INHERIT new class myMatrix3x3 from Matrix3x3 and override Load()

```
class myMatrix3x3 : public Matrix3x3 {

public:
    void Load(char* filename);
}
```

To use:

```
myMatrix3x3 a;
a.Load("a.matr");
a.Print();
```

This calls the Print function from Matrix3x3 class

Pipeline Concept

VTK follows a pipeline concept where (almost) every class represents a filter with input and output. These filters concatenated then form a pipeline that renders the resulting image.

Each filter implements a method called `Execute`. This method does all the computations. It can be invoked manually or automatically by the following filter whenever that filter needs its input.

For example, the vtkRendererWindow, that actually displays the results invokes the `Execute` method whenever it does not have any input yet or the input is no longer current.

VTK Filter Objects

Each element in the pipeline is considered a filter with input and output. As mentioned previously, the filter is executed using the method `Execute`.

In addition, a method `Update` exists which checks if the filter's internal data is still current. If it is not, for example because the filter's input has changed, the Execute method is called to bring the filter up-to-date.

VTK Filter Objects (continued)

VTK uses reference counting for its filter objects. This means that the number of references (pointers) to a filter object is counted. Hence, a filter object should not be deleted from memory by simply invoke the C++ delete command. Instead, use the method `Delete` that is implemented by the filter object. This then decrements the reference counter. Once the reference counter becomes zero the object is removed from memory (otherwise there would still be a reference to the filter object; using it would cause a segmentation fault if the filter object would have been deleted).

Connecting Filter Objects

In order to set up the pipeline, i.e. a concatenation of a series of filters, the filters need to be connected in some way. Therefore, every filter has methods to retrieve the output and set the input:

`SetInput():`	set the input of a filter (the input does not need to be available/computed yet)
`GetOutput():`	get the output from a filter (the output does not need to be available/computed yet)

To connect to filters `filterA` and `filterB` we simply need to issue:

```
filterA.SetInput (filterB.GetOutput ())
```

Types of In- and Output

VTK supports multiple input as well as multiple output. In the same way, a filter object may require more than one input depending on its purpose. Therefore, VTK provides the methods `SetInputs` and `GetOutputs`, so that multiple in- and output can be handles by the filter object.

In addition, the output of a filter object can be used more than once. The output can be used as input for as many subsequent filter objects as necessary for the application.

Types of Filter Objects

Different types of filter objects are available in VTK:

- Data objects
- Process objects

Each type of object serves a different purpose.

Data Objects

Data objects, represented by the abstract class `vtkDataSetSource`, provides information. This information can be either generated by the object or retrieved from somewhere. For example, a data object can load a given data set from the hard drive and provide it to the subsequent filter object.

Several different data objects are provided by VTK already for generating simple objects or reading files in various file formats.

Process Objects

A process object, represented by the abstract class `vtkProcessObject`, modifies its input data in some way.

Lots of process objects are available in VTK for transforming data in various ways. For example, the `vtkTransformFilter` applies a transformation matrix to the input. This can be used for scaling or moving geometric objects that were previously generated by a data object.

Implementing Your Own Filter Object

VTK allows you to implement your own filter object that either generates data (data object) or modifies data (process object). Such a filter can then be added to the pipeline just like any other filter.

When implementing the filter, the main algorithm is to be implemented in the `Execute` method. In addition, you have to make sure that the methods `SetInput` and `GetOutput` accept and provide the correct data.

Execution of the Pipeline

VTK uses implicit control of the visualization pipeline (or network if more complex) execution. Execution occurs when output is requested from an object. Key methods here are the two methods Update and Execute. The Update method is generally initiated when the user requests the system to render a scene. As part of the process the actors send a Render() method to their mappers. At this point network execution begins. The mapper invokes the Update() method in its input(s). These in turn recursively invoke the Update() method on their input(s). This process continues until a source object (data object) is encountered.

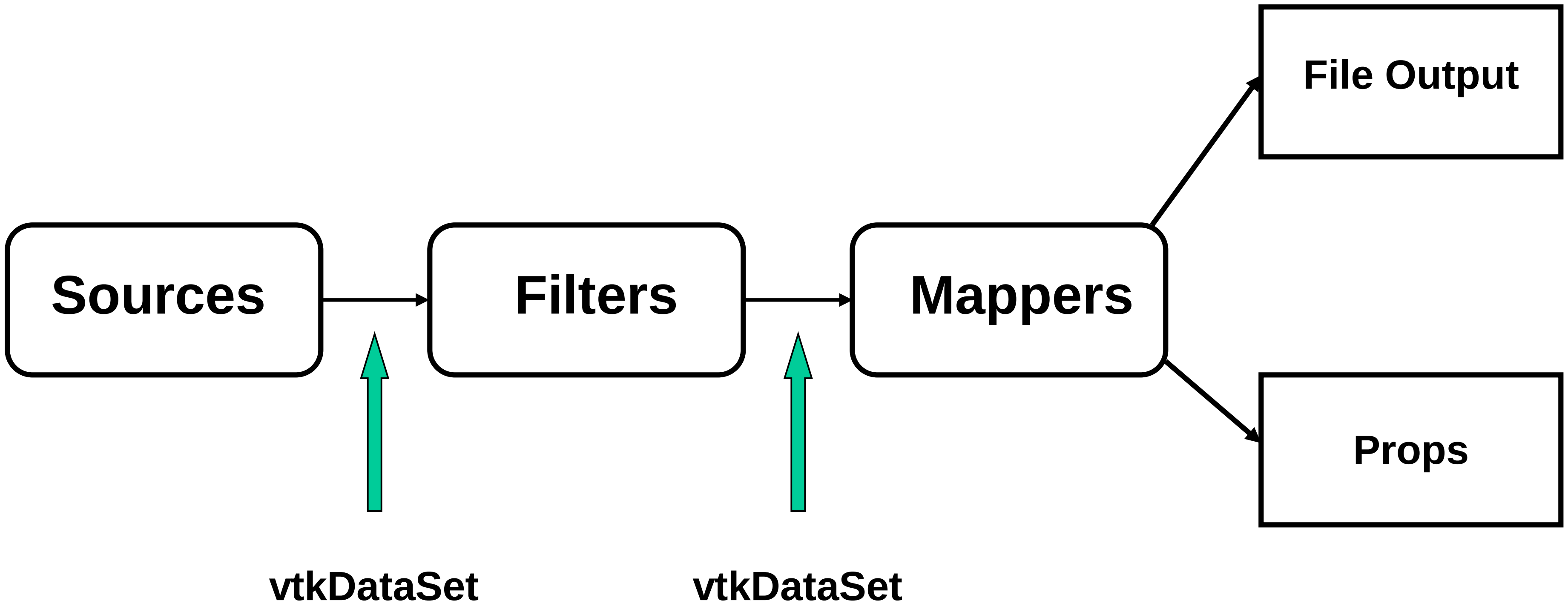
Execution of the Pipeline

At this point the source object compares its modified tome to the last time executed. If it has been modified more recently than executed, it re-executes via the Execute() method. The recursion then unwinds with each filter compring its input time to its execution time. Execute() is called where appropriate. The process terminates when control is returned to the mapper.

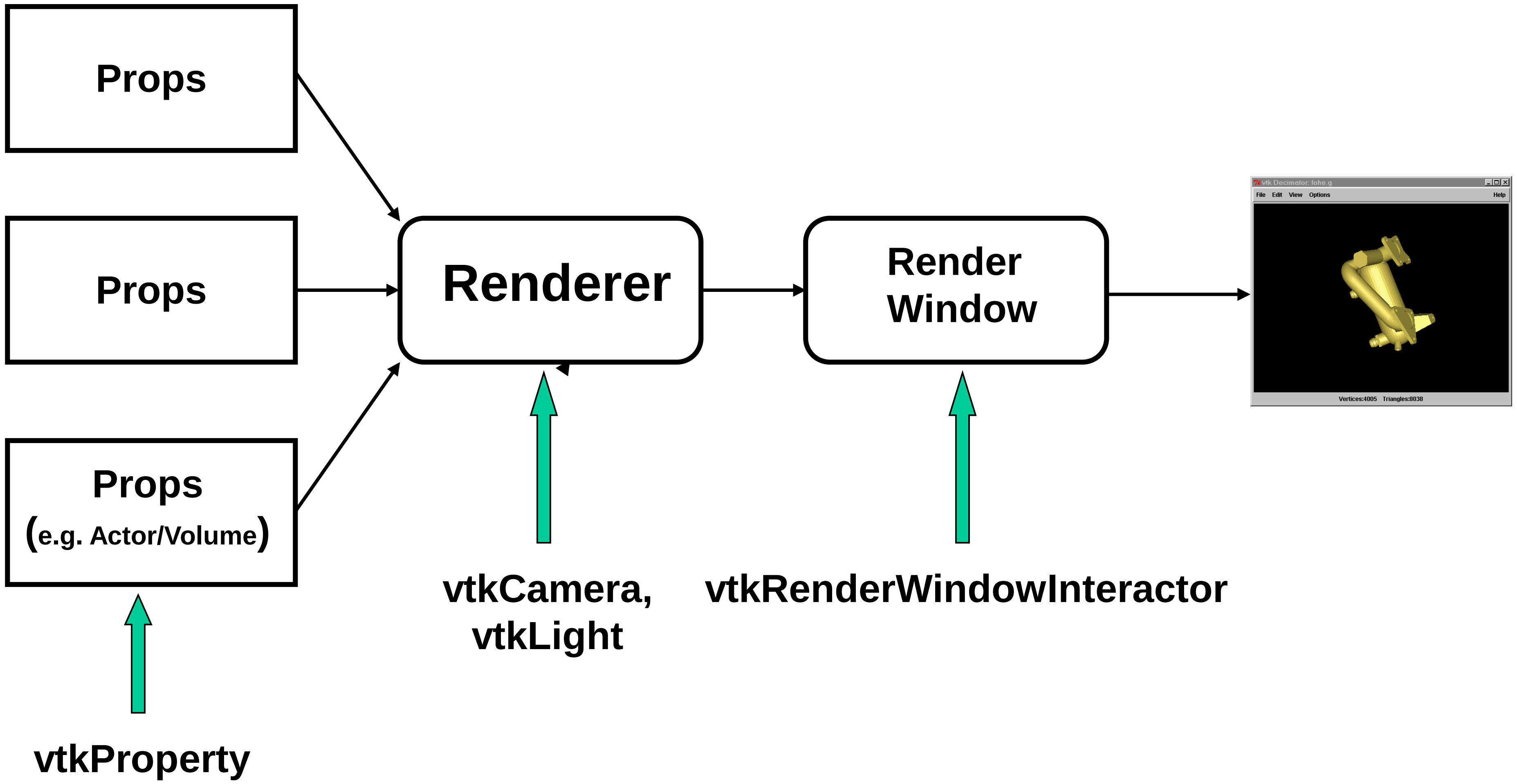
Loops

Filter objects can be concatenated in such a way that loops occur in the pipeline. VTK supports such a configuration; however, the loop is only executed once an update request is issued.

VTK Pipeline



VTK Pipeline (continued)

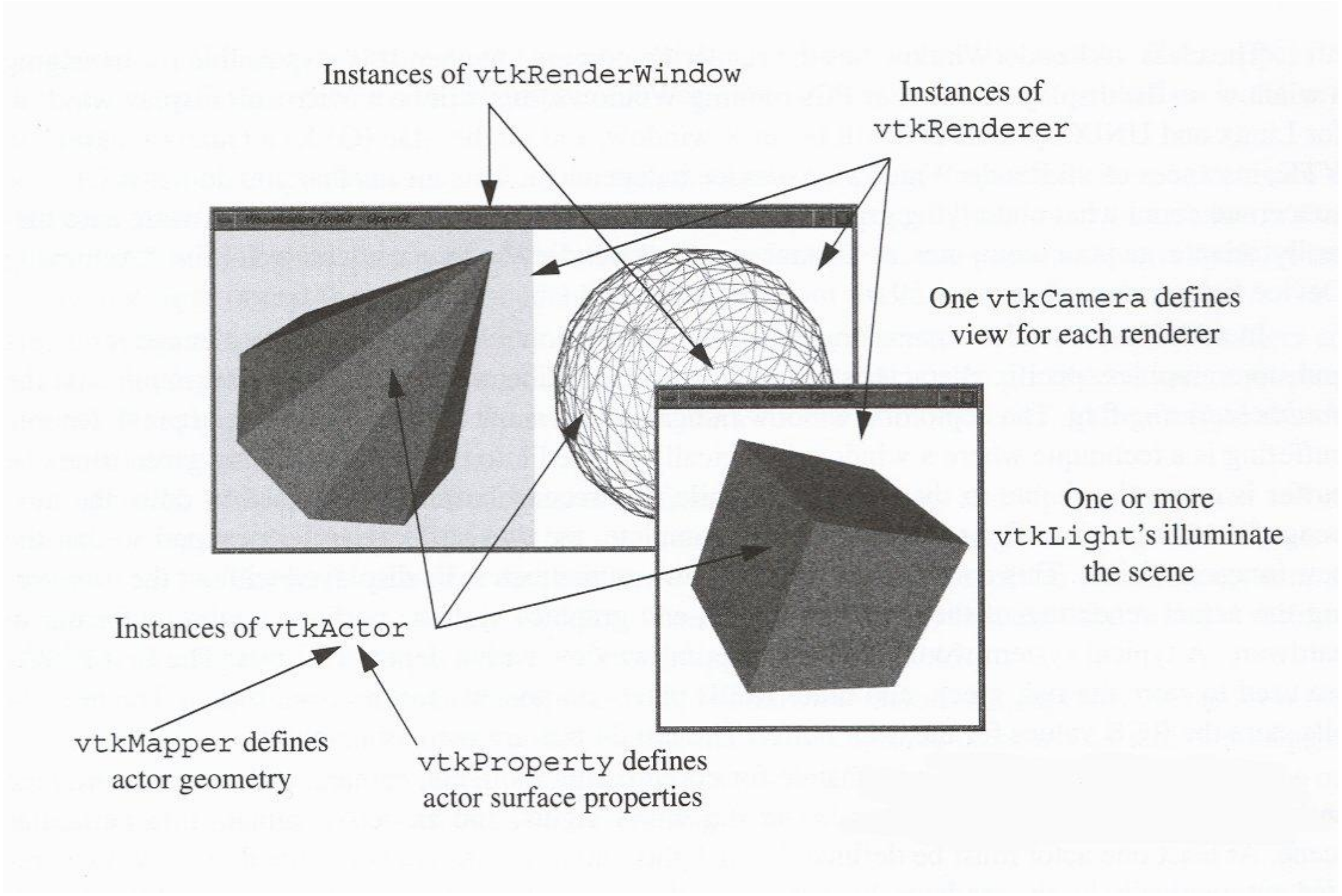


The Graphics Model

In order to generate an image using VTK several different elements are involved:

1. vtkRenderWindow: manages a window on the display device; one or more renderers draw into an instance of vtkRenderWindow.
2. vtkRenderer: coordinates the rendering provess involving lights, cameras, and actors.
3. vtkCamera: defines the view position, focal point, and other viewing properties of the scene.
4. vtkLight: a source of light to illuminate the scene
5. vtkActor: represents an object rendered in the scene, both its properties and position in the world coordinate system.
6. vtkProperty: defines the appearance properties of an actor including color, transparency, and lighting properties such as specular and diffuse. Also representational properties like wireframe and solid surface.
7. vtkMapper: the geometric representation for an actor. More than one actor may refer to the same mapper.

The Graphics Model (continued)


Instances of vtkRenderWindow
Instances of vtkRenderer
One vtkCamera defines view for each renderer
One or more vtkLight's illuminate the scene
Instances of vtkActor
vtkMapper defines actor geometry
vtkProperty defines actor surface properties
Visualization Toolkit - OpenGL

Example: cone

Source code:

```
vtkConeSource *cone = vtkConeSource::New();
  cone->SetHeight( 3.0 );
  cone->SetRadius( 1.0 );
  cone->SetResolution( 10 );
vtkPolyDataMapper *coneMapper =
   vtkPolyDataMapper::New();
  coneMapper->SetInput( cone->GetOutput() );
vtkActor *coneActor = vtkActor::New();
  coneActor->SetMapper( coneMapper );
vtkRenderer *ren1= vtkRenderer::New();
  ren1->AddActor( coneActor );
  ren1->SetBackground( 0.1, 0.2, 0.4 );
```

Example: cone (continued)

Source code (continued):

```
vtkRenderWindow *renWin = vtkRenderWindow::New();
  renWin->AddRenderer( ren1 );
  renWin->SetSize( 300, 300 );
vtkRenderWindowInteractor *iren =
  vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);
vtkInteractorStyleTrackballCamera *style =
    vtkInteractorStyleTrackballCamera::New();
  iren->SetInteractorStyle(style);
iren->Initialize();
  iren->Start();
```

Example: cone (continued)

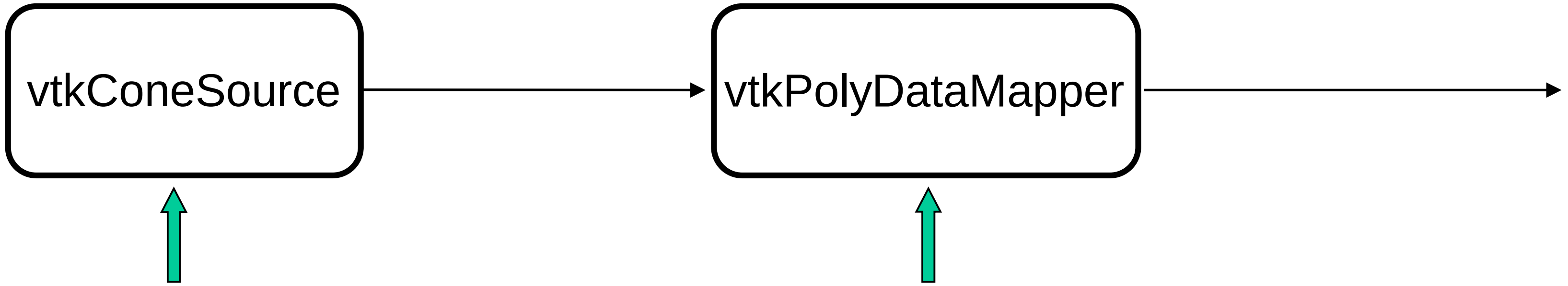
Source code (continued):

```
cone->Delete();
coneMapper->Delete();
coneActor->Delete();
ren1->Delete();
renWin->Delete();
iren->Delete();
style->Delete();

return 0;
```

Example: cone (continued)

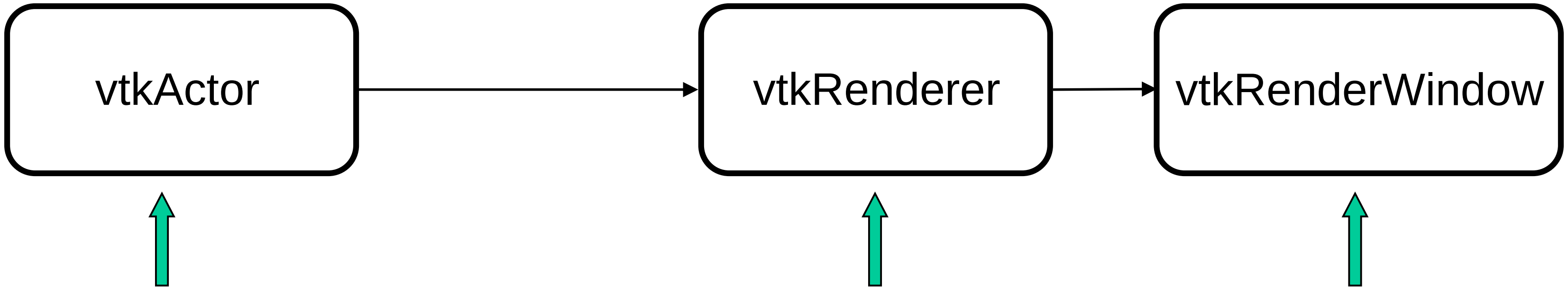
The VTK pipeline:



First, we create an instance of vtkConeSource and set some of its properties. The instance of vtkConeSource "cone" is part of a visualization pipeline (it is a source process object); it produces data (output type is vtkPolyData) which other filters may process.

In this example we terminate the pipeline with a mapper process object. (Intermediate filters such as vtkShrinkPolyData could be inserted in between the source and the mapper.) We create an instance of vtkPolyDataMapper to map the polygonal data into graphics primitives. We connect the output of the cone souece to the input of this mapper.

Example: cone (continued)



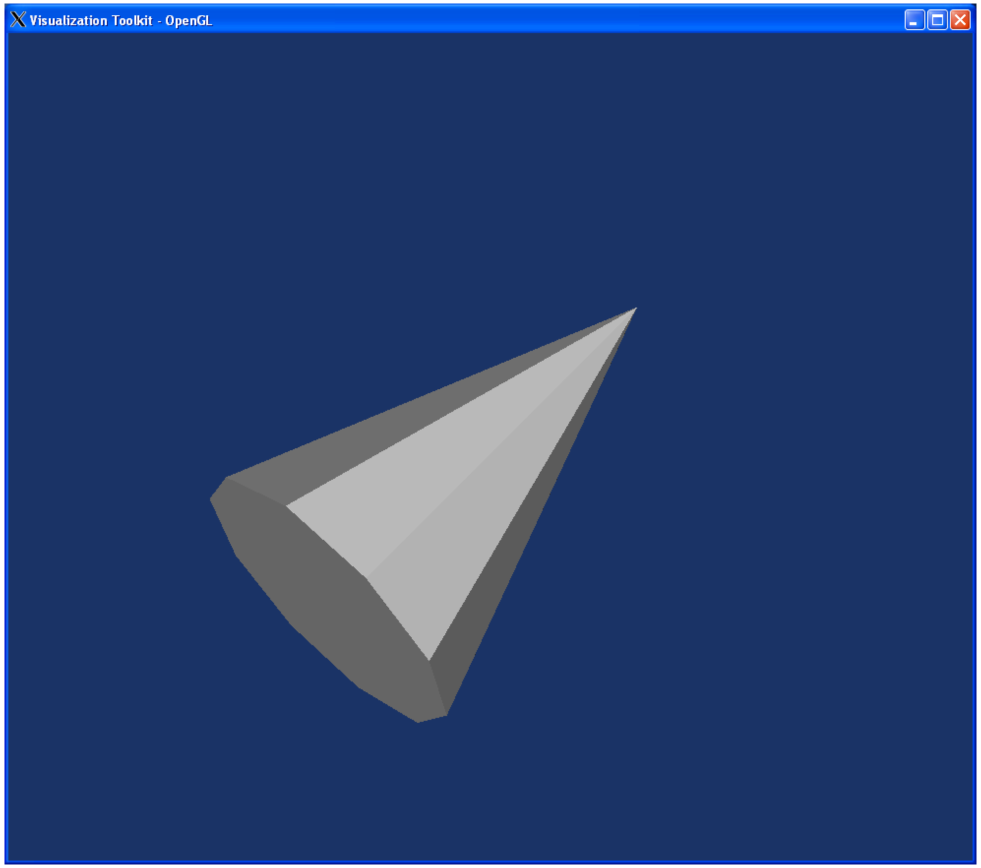
Create an actor to represent the cone. The actor orchestrates rendering of the mapper's graphics primitives. An actor also refers to properties via a vtkProperty instance, and includes an internal transformation matrix. We set this actor's mapper to be coneMapper which we created above.

Create the Renderer and assign actors to it. A renderer is like a viewport. It is part or all of a window on the screen and it is responsible for drawing the actors it has. We also set the background color here.

Finally we create the render window which will show up on the screen. We put our renderer into the render window using AddRenderer. We also set the size to be 300 pixels by 300.

Example: cone (continued)

Resulting rendering



Example: Mace