

The Visualization Pipeline

The Visualization Pipeline

Outline

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actually displays the results invokes the `Execute` method whenever it does not have any input yet or the input is no longer current.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

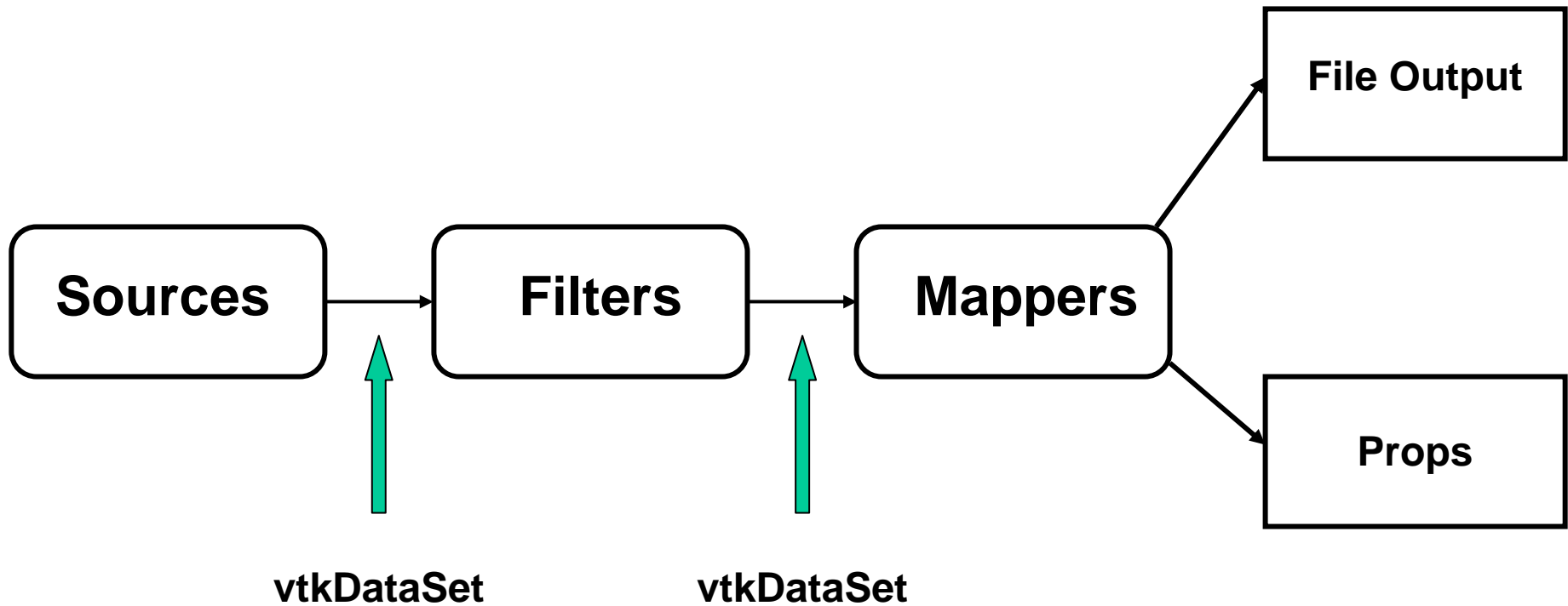
Execution of the Pipeline

At this point the source object compares its modified time 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 comparing its input time to its execution time. `Execute()` is called where appropriate. The process terminates when control is returned to the mapper.

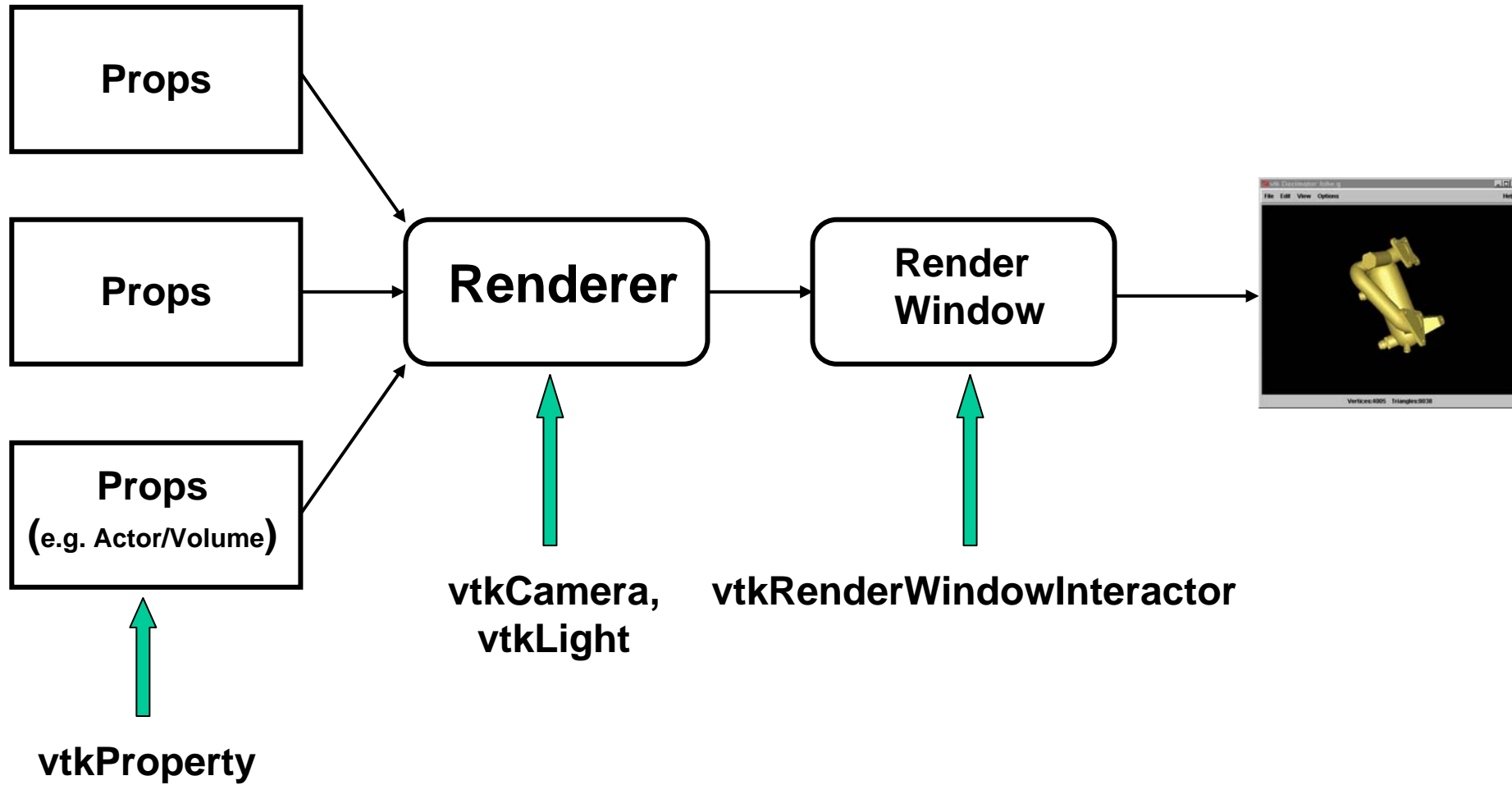
Loops

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)

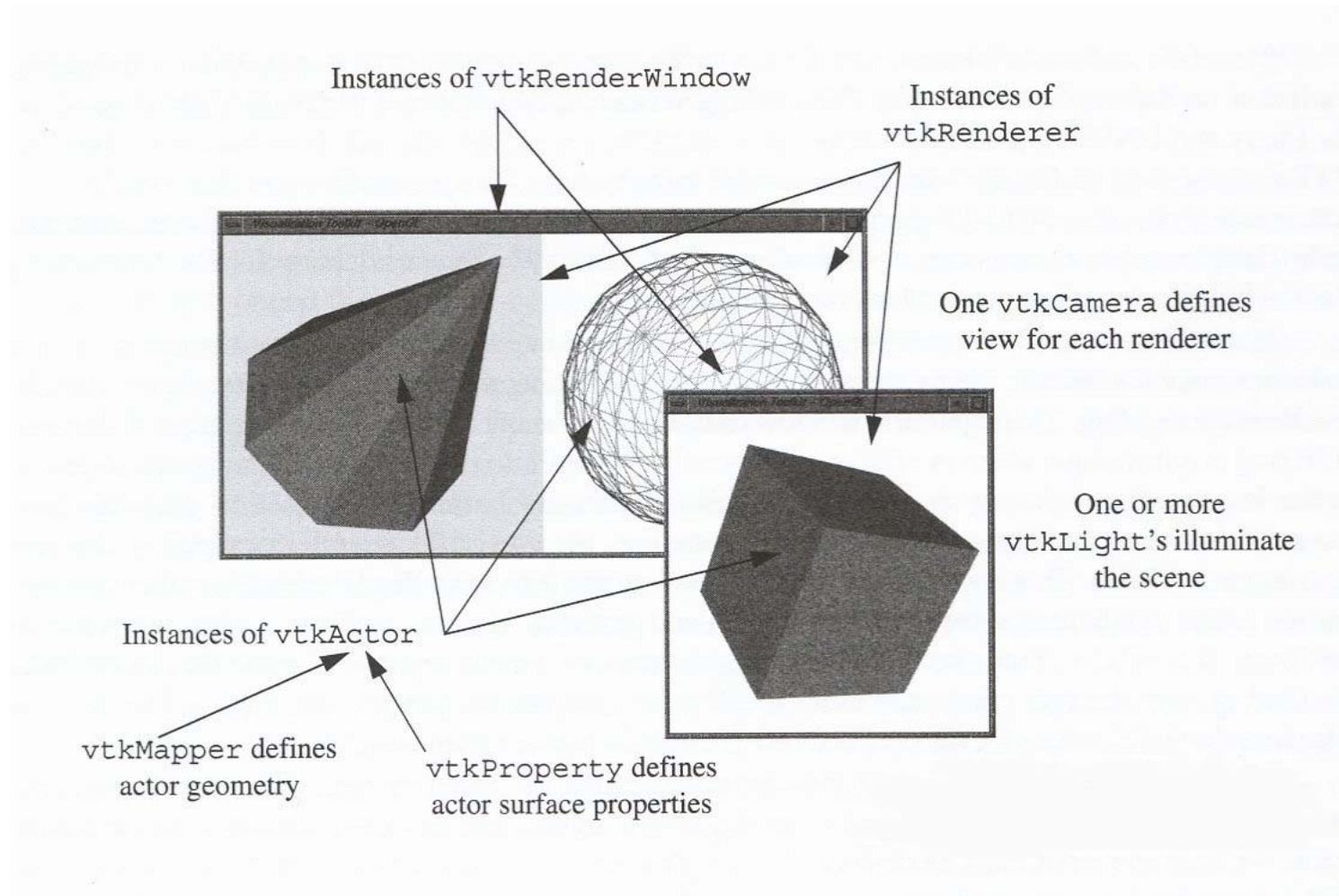


The Graphics Model

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 process 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)



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();
```

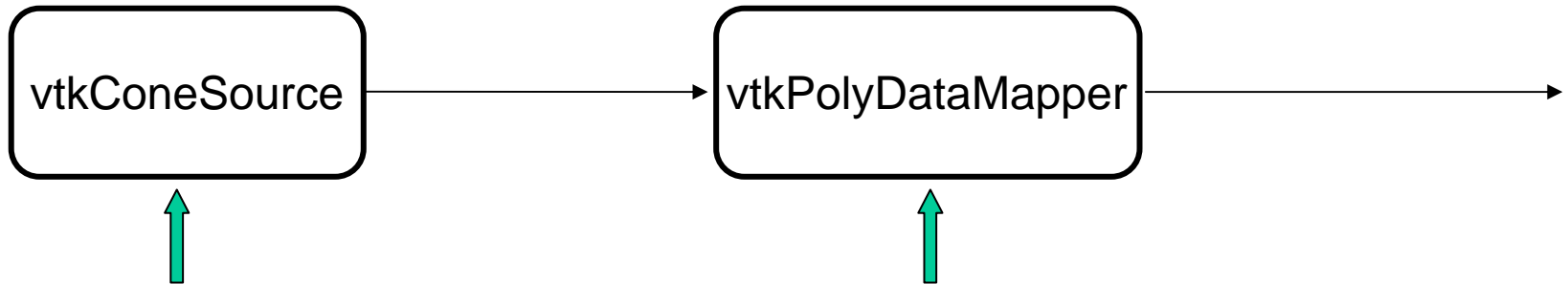

Example: cone (continued)

Source code (continued):

```
cone->Delete();  
coneMapper->Delete();  
coneActor->Delete();  
ren1->Delete();  
renWin->Delete();  
iren->Delete();  
style->Delete();  
  
return 0;
```

Example: cone (continued)

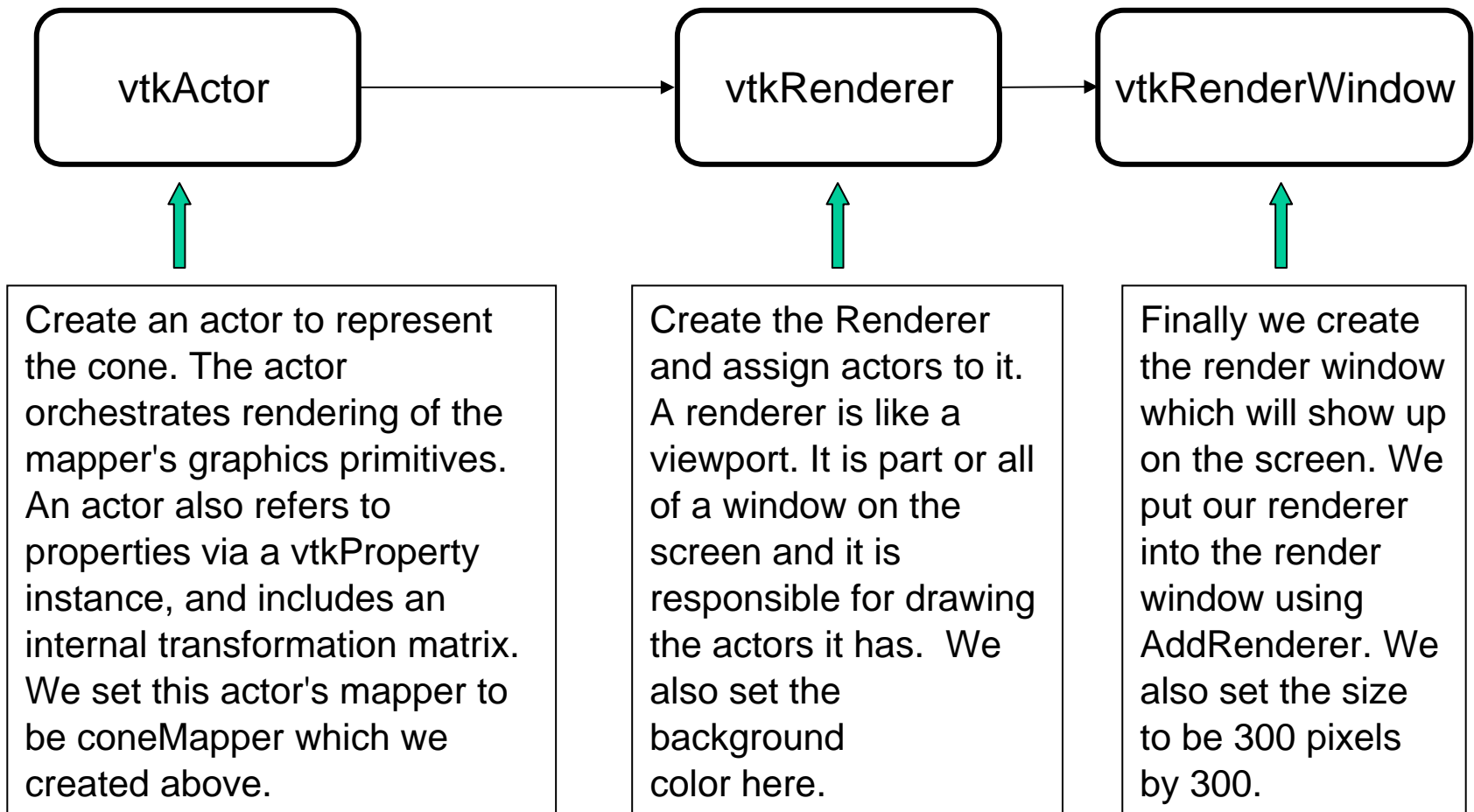
The VTK pipeline:



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.

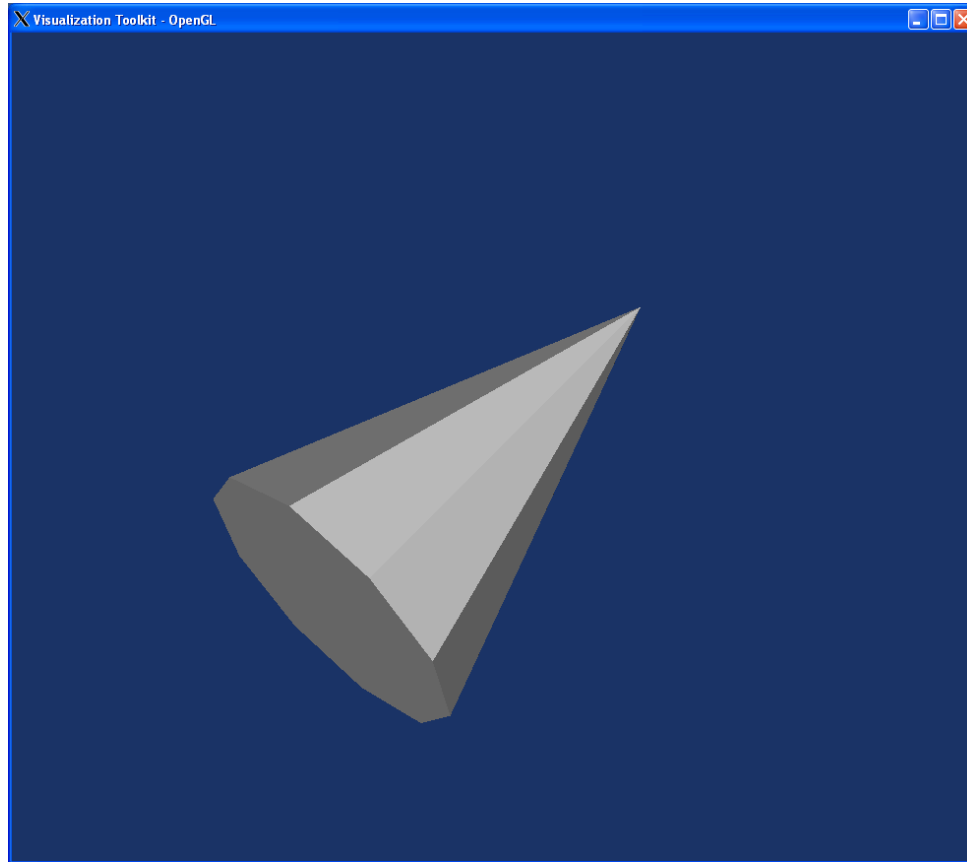
In this example we terminate the pipeline with a mapper process object. (Intermediate filters such as `vtkShrinkPolyData` could be inserted in between the source and the mapper.) We create an instance of `vtkPolyDataMapper` to map the polygonal data into graphics primitives. We connect the output of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

Example: cone (continued)



Example: cone (continued)

Resulting rendering



Example: Mace

Example: Model
