VTK Lesson 01

Introduction to VTK

112793 Cláudia Seabra, 131224 Francisco Costa
Information Visualization, 2025/26
MSc Informatics Engineering, University of Aveiro

Abstract

This lesson introduces the Visualization Toolkit (VTK), an open-source framework for scientific and 3D visualization. The lesson focus on the installation and use of VTK in Python,the construction of a basic visualization pipeline, and the rendering of simple geometric primitives. Several rendering features were explored, including background configuration, camera manipulation, object resolution, interaction mechanisms, and lighting models.

Through a series of incremental experiments, the lesson demonstrates how VTK structures its rendering workflow and how different parameters influence the visual appearance and perception of 3D objects.

Objectives

The main objective of this lesson is to become familiar with the fundamental concepts and tools offered by VTK, particularly the construction of a visualization pipeline in Python.

This report explores the creation and display of geometric primitives, modification of their proprieties, as well as the interaction with visualization windows. Additional goals include understanding how VTK handles camera control, lighting, and rendering modes. These initial exercises establish the foundation required for more advanced tasks in subsequent lessons.

VTK

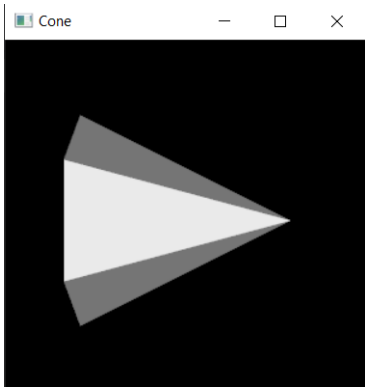
The Visualization Toolkit (VTK) is an open-source library widely used in scientific visualization, computer graphics, and image processing. It provides a modular architecture based on a pipeline model in which data objects progress through sources, filters, mappers, and renderers until they are displayed. VTK supports various geometric primitives, 3D widgets, interaction modes, and rendering techniques, making it suitable for both simple and highly complex visualization tasks. [1]

Visualization Solution

First Example

The lesson begins with the cone.py example provided. By running the script, there's a default cone rendered in a 300x300 window displayed, which rotates once before closing. The cone is generated through a standard VTK pipeline composed of:

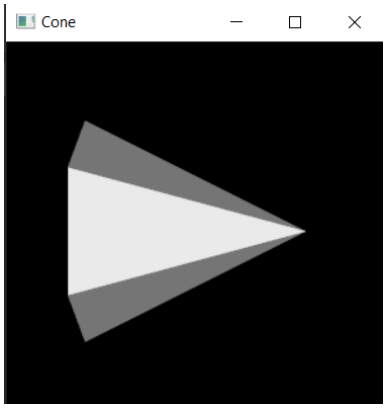
- a source (vtkConeSource)
- a mapper (vtkPolyDataMapper)
- an actor (vtkActor)
- a renderer (vtkRenderer)
- a render window (vtkRenderWindow)



Modifying the Cone Geometry

To customize the cone, the following lines were added to specify a height of 2 and a radius of 1.

```
coneSource.SetHeight(2)
coneSource.SetRadius(1)
```



These changes modify the cone's geometry while keeping the overall appearance mostly unchanged from the default example.

Resolution Tests

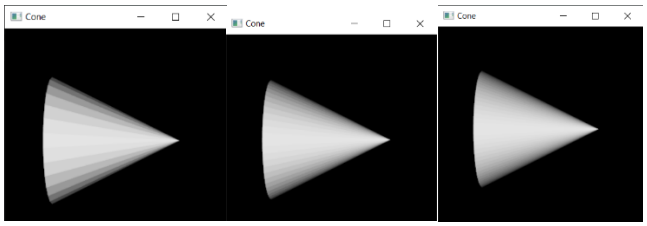
Next, the SetResolution() method was tested using values of 30, 60, and 90:

```
coneSource.SetResolution(x)
```

This method controls how many facets (triangles) make up the circular base and the surface of the cone. In other words, SetResolution() defines the number of subdivisions around the cone's circumference.

Low resolution: the cone appears faceted (polygonal), as it is composed of a small number of polygons. This results in lower computational cost and faster rendering performance.

High resolution: the cone appears smoother due to the increased number of triangles used to represent its surface. Consequently, rendering becomes slower and requires more computational resources.

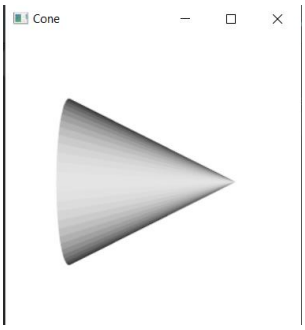


After testing, the resolution was set to 60 as a balanced value.

Background Color

The background color of the scene was modified to white:

```
ren.SetBackground(1, 1, 1)
```



Window Size

A line was added after defining the render window to explicitly set its size to 300×300, although this matches the default:

```
renWin.SetSize(300, 300)
```

Other Primitives

Sphere

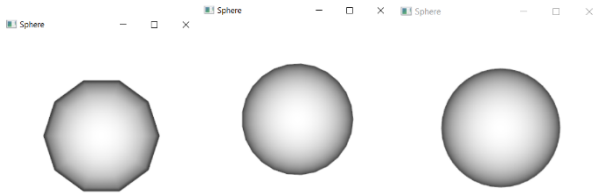
The cone was replaced by a sphere generated through vtkSphereSource, with a radius of 2:

```
sphereSource = vtkSphereSource()
sphereSource.SetRadius(2)
```

Both the phi and theta resolution were tested at 10, 25, and 40. This produced visible differences in surface smoothness. Once again, it was observed that higher resolution results in a more spherical and less faceted appearance.

```
sphereSource.SetPhiResolution(y)
sphereSource.SetThetaResolution(y)
```

The following figures represent the different values tested, but it's easier to observe when the sphere spins.

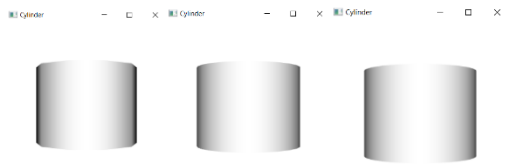


Cylinder

A cylinder of radius 2 and height 3 was also implemented:

```
cylinderSource = vtkCylinderSource()
    cylinderSource.SetRadius(2)
    cylinderSource.SetHeight(3)
```

Again, three resolutions (10, 25, 40) were applied, illustrating how mesh density influences the smoothness of the circular bases and lateral surface.



Introducing Interaction

To enable real-time interaction with the 3D scene, the original rotation loop was replaced by a vtkRenderWindowInteractor, which allows the user to rotate, zoom, and pan directly with the mouse:

```
iren = vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)
iren.Initialize()
iren.Start()
```

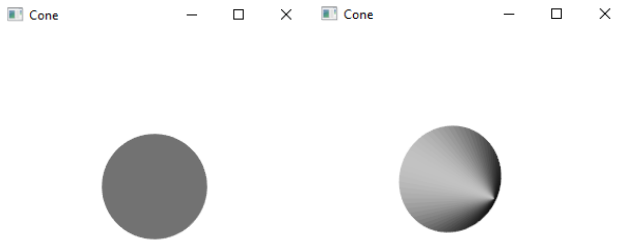
This interactor significantly improves the usability of the visualization by allowing dynamic inspection of the rendered object.

Camera Control

A custom camera was introduced to better understand how VTK handles viewpoint manipulation:

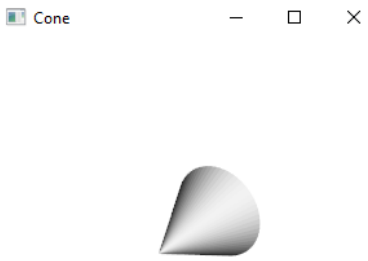
```
cam1 = vtkCamera()
cam1.SetPosition(10, 0, 0)
cam1.SetViewUp(0, 1, 0)
ren.SetActiveCamera(cam1)
```

This modification places the camera 10 units along the x-axis, looking toward the origin with a vertical orientation defined by the view-up vector. The result is a frontal perspective of the object.



When the camera was instead positioned at (10, 10, 0) with a ViewUp vector of (0, 1, 1),

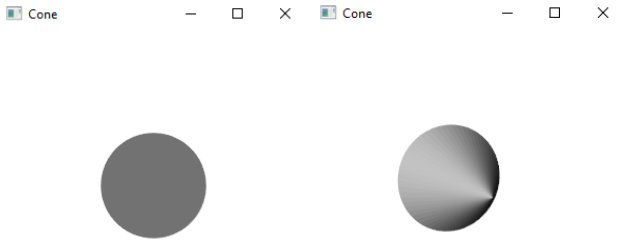
the scene appeared tilted, demonstrating how the view-up vector influences orientation around the viewing direction.



Using the Default Camera

The same configuration was applied using the renderer's default camera rather than creating a new one:

```
cam = ren.GetActiveCamera()
cam.SetPosition(10, 0, 0)
cam.SetViewUp(0, 1, 0)
```



Both approaches produce identical visual results but using GetActiveCamera() is preferable when no additional camera objects are needed.

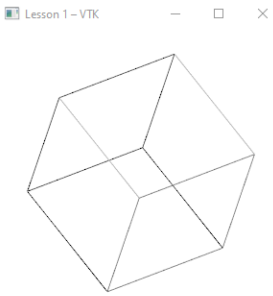
Orthographic Projection

By enabling parallel projection:

```
cam.SetParallelProjection(True)
```

The visualization changes from a perspective camera (objects shrink with distance) to an orthographic projection (no perspective distortion).

Viewing a cube in wireframe mode under orthographic projection reveals perfectly parallel edged, which allows the conclusion that this mode is suitable for engineering and architectural visualization where geometric precision is required.



Lighting

Lighting experiments were conducted to understand how illumination affects object perception.

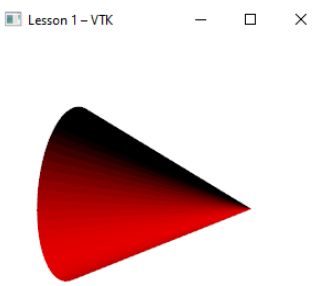
Simple Light Attached to the Camera

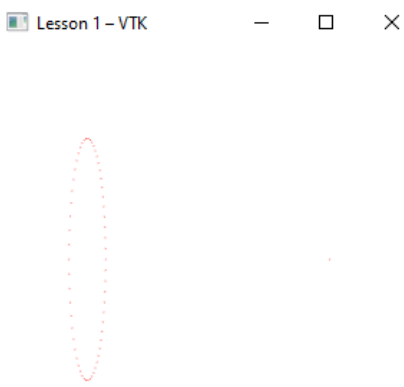
Replacing the code with:

```
cam1 = ren.GetActiveCamera()
light = vtkLight()
light.SetColor(1, 0, 0)
light.SetFocalPoint(cam1.GetFocalPoin
t())
light.SetPosition(cam1.GetPosition())
ren.AddLight(light)
```

adds a red light source to the camera, meaning the light follows the camera's position.

This produces a scene with strong red shading, where the illuminated faces appear tinted while shadows remain darker.





Actor Properties

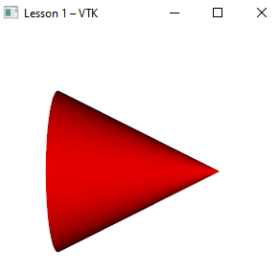
VTK actor properties were used to control visual appearance:

- Color to distinguish objects or emphasize features;
- Opacity to induce transparency and reveal internal or overlapping structures;
- Representation modes to inspect geometry or focus on structural aspects.

The cone's color was modified using:

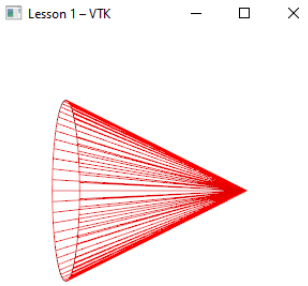
```
coneActor.GetProperty().SetColor(1, 0, 0)
```

This sets the cone to pure red.



Other visual representations were tested:

- Points mode – mesh rendered as vertex points
- Wireframe mode – only edges visible, useful for inspecting geometry.
- Surface mode – full shading (default).



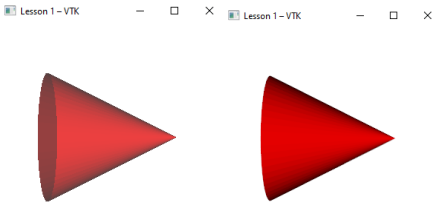
The transparency was also varied using:

```
coneActor.GetProperty().SetOpacity(value)
```

Opacity = 0: object becomes invisible.

Opacity = 0.5: object becomes semi-transparent, allowing internal geometry to be seen.

Opacity = 1: object is fully opaque (default).



These properties are essential when visualizing overlapping datasets or when highlighting layered structures.

Properties and Multi-Light Setup

Four colored lights were added around the scene, all pointing toward the origin:

- Red – position (-5, 0, 0)
- Green – position (0, 0, -5)
- Blue – position (5, 0, 0)
- Yellow - position (0, 0, 5)

To avoid code repetition, a function was implemented:

```
def addLight(renderer, color, position):
    light = vtkLight()
    light.SetColor(color)
    light.SetPosition(position)
    light.SetFocalPoint(0, 0, 0)
    renderer.AddLight(light)
```

This enabled straightforward creation of multiple light sources.

Light Markers with Spheres

To visually indicate light position, small spheres with a 0.5 radius were placed at each light's coordinates.

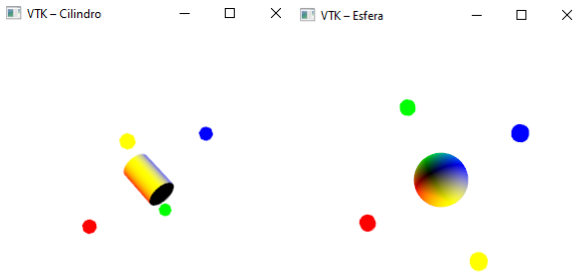
To ensure these spheres were not affected by lighting, their shading was disable:

```
sphereActor.GetProperty().LightingOff()
```

This preserves the assigned color regardless of other lights in the scene, making them accurate markers for illumination direction.

Testing with Multiple Primitives

The cone was replaced by a sphere and other shapes to observe the combined effect of the four lights. With multiple colors, the illuminated faces display gradients resulting from mixing contributions from different light sources, demonstrating how surface orientation influences perceived color.



Conclusion

The first VTK practical lesson provided a comprehensive introduction to the fundamental concepts of scientific visualization using the Visualization Toolkit. Through the exploration of geometric primitives and rendering parameters, it became clear how VTK structures its visualization pipeline and how flexible it is in adapting to different visualization needs.

By experimenting with actor properties such as color, transparency, and representation modes, a deeper understanding was gained of how objects can be visually emphasized or de-emphasized within a scene. The implementation of multiple colored light sources further illustrates how illumination can be combined to create complex shading effects and enhance perception of geometry.

Overall, this lesson established a strong foundation for future visualization tasks, such as the next two lessons. It provided not only practical skills in constructing VTK pipelines but also conceptual insights into how rendering and illumination interact to produce effective 3D visualizations.

References

[1] Inc. Kitware, "VTK: The Visualization Toolkit". [Online]. Available: https://vtk.org/