VTK Lesson 02

Projections, lighting and transformations

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

Abstract

This lesson explores advanced concepts in 3D visualization using the Visualization Toolkit (VTK). It focuses on the implementation of multiple actors and renderers, the use of various shading techniques, texture mapping, transformations, and interaction mechanisms in 3D visualizations. Key topics covered include the manipulation of object properties, the addition of textures to geometric shapes, and the application of transformations such as translation and rotation. The lesson also introduces the use of observers and callback functions to enable real-time interaction and dynamic updates of the visualization.

Through a series of exercises, an understanding of how VTK handles multiple rendering contexts, lighting effects, and the application of visual transformations is gained.

Objectives

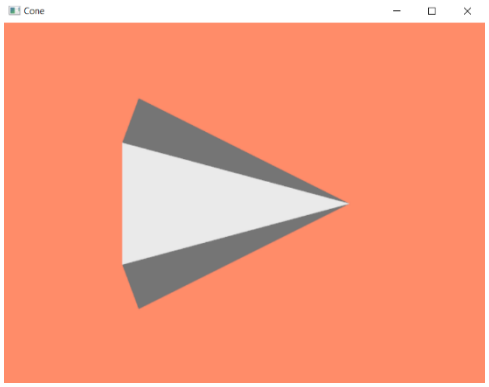
The primary objective of this lesson is to provide a deeper understanding of the tools and techniques for creating interactive and visually enhanced 3D renderings using VTK. Specific goals include:

1. Understanding how to work with multiple actors and renderers in a single window;
2. Learning various shading options and their effects on the appearance of objects;
3. Implementing texture mapping and manipulating texture properties;
4. Applying geometric transformations (translation, rotation) to objects in space;
5. Implementing observers and callbacks to enhance user interaction with visualized data.

Visualization Solution

Multiple Actors

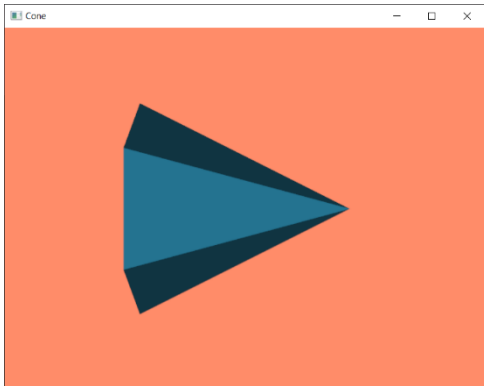
The lesson begins with modifications to the cone.py script, which initially renders a cone in a 3D scene.



The cone's appearance was adjusted using the `vtkProperty` class, which allows control over the

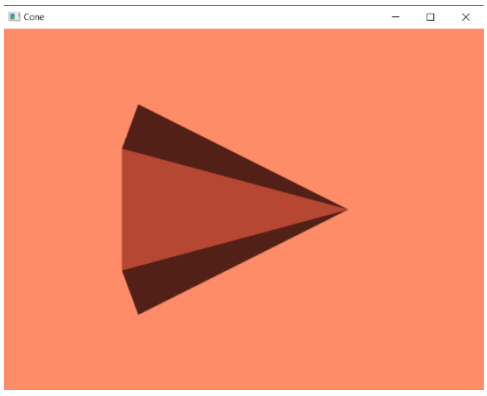
object's properties, such as color, diffuse lighting, specular highlights, and opacity.

```
coneActor.GetProperty().SetColor(0.2, 0.63, 0.79)
coneActor.GetProperty().SetDiffuse(0.7)
coneActor.GetProperty().SetSpecular(0.4)
coneActor.GetProperty().SetSpecularPower(20)
```



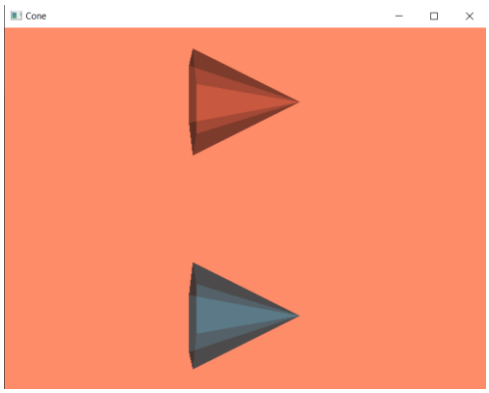
Then, a second cone was added to the code, once again using the `vtkProperty` to adjust it but by creating a property variable.

```
property = vtkProperty()
property.SetColor(1.0, 0.3882, 0.2784)
property.SetDiffuse(0.7)
property.SetSpecular(0.4)
property.SetSpecularPower(20)
```



This way, the properties of both cones are manipulated using the same `vtkProperty` object for efficient management.

After changing both cones' opacities, this was the result:



This was achieved by adding `property.SetOpacity(0.5)` to the property variable.

This method simplifies managing multiple objects with similar properties, showing that we can either modify properties individually or apply a predefined property set to multiple objects.

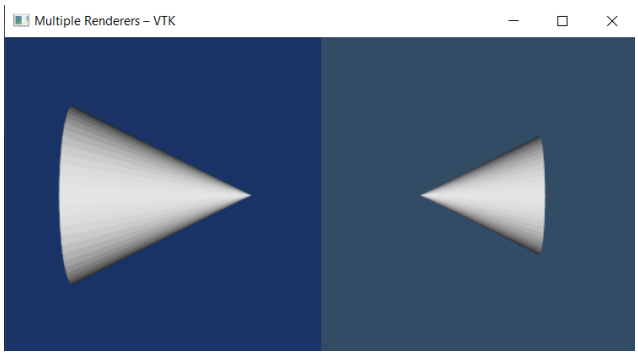
Multiple Renderers

To increase the scene's complexity, multiple renderers are used within the same window. The window is resized to 600x300 pixels, and the `SetViewport(xmin, ymin, xmax, ymax)` method defines the regions of the window where each renderer operates. The following adjustments are made:

```
ren1.SetViewport(0.0, 0.0, 0.5, 1.0)
ren2.SetViewport(0.5, 0.0, 1.0, 1.0)
```

Additionally, the camera for the second renderer is adjusted to a 90-degree azimuthal rotation to provide a different viewpoint of the same object. Different background colors are applied to each renderer to clearly distinguish the two views:

```
ren1.SetBackground(0.1, 0.2, 0.4)
ren2.SetBackground(0.2, 0.3, 0.4)
```



Shading Options

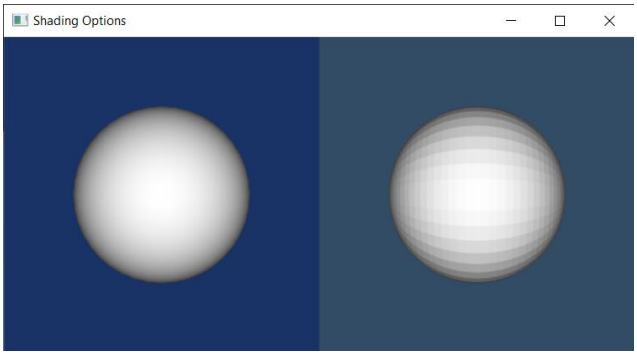
In this exercise, we rendered two spheres in the same window. The goal was to experiment with different shading techniques and observe the effects on the appearance of the spheres.

Three shading techniques were tested:

1. Flat Shading
   `(SetInterpolationToFlat())`
2. Gouraud Shading
   `(SetInterpolationToGouraud())`
3. Phong Shading
   `(SetInterpolationToPhong())`

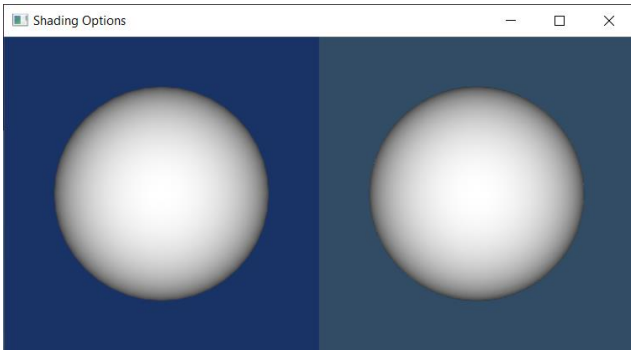
Each technique was applied to the sphere in the second viewport.

- Flat Shading:



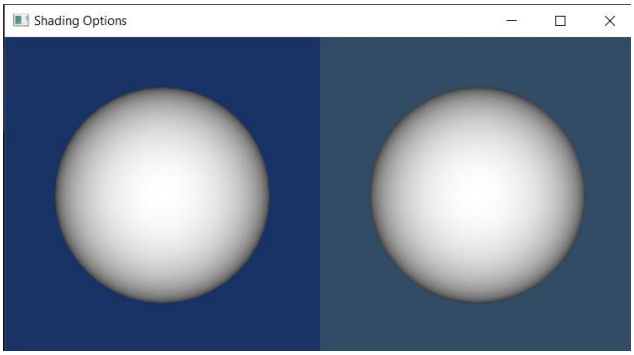
The sphere appeared faceted and angular, with abrupt transitions in lighting between polygons, giving it a blocky appearance.

- Gouraud Shading:



The sphere appeared smoother, with more natural lighting transitions due to vertex-based lighting. While the surface was softer, there were still visible edges.

- Phong Shading:



This provided the most realistic result with a smoother surface.

Textures

To introduce textures, a `vtkPlaneSource` is created, and a texture is applied to the plane. The texture is loaded from a JPEG file using the `vtkJPEGReader` class and then mapped onto the plane as follows:

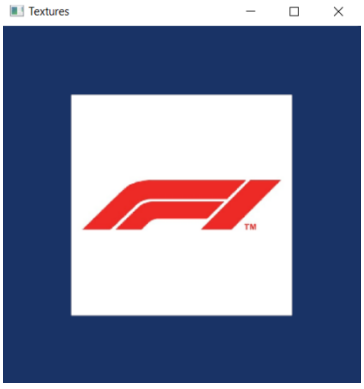
```
JPGReader = vtkJPEGReader()
JPGReader.SetFileName("formula-1-logo.JPG
")
JPGReader.Update()

texture = vtkTexture()
texture.SetInputConnection(jpg_reader.GetOu
tputPort())

plane_mapper = vtkPolyDataMapper()
plane_mapper.SetInputConnection(plane.GetOu
tputPort())

plane_actor = vtkActor()
plane_actor.SetMapper(plane_mapper)
plane_actor.SetTexture(texture)
```

In this case, the image formula-1-logo.JPG is loaded using the `vtkJPEGReader` class. This class reads the image file and prepares it for use as a texture. The texture is then applied to the plane by connecting the reader's output to the `vtkTexture` object, which is finally set as the texture for the plane actor using `SetTexture()`.



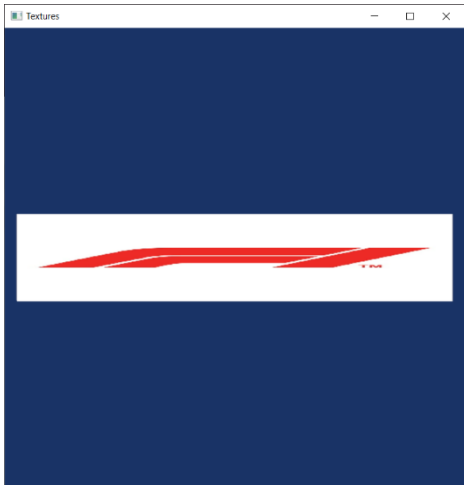
Next, the plane's size is adjusted by changing the `SetOrigin()`, `SetPoint1()`, and `SetPoint2()` methods. These parameters define the dimensions and orientation of the plane in 3D space. For instance:

```
plane.SetOrigin(0,0,0)
plane.SetPoint1(1,0,0)
plane.SetPoint2(0,1,0)
```

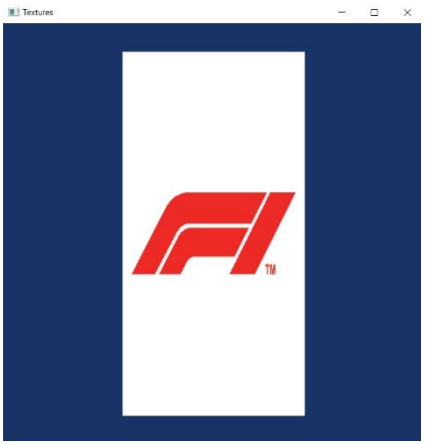
This code defines a unit plane with one corner at the origin (0, 0, 0), the second corner along the x-axis at (1, 0, 0), and the third corner along the y-axis at (0, 1, 0).

As the size of the plane changes, the texture applied to the plane behaves differently:

- When the plane is scaled larger, the texture appears stretched because the texture coordinates (which are based on the plane's dimensions) remain fixed, but the plane itself becomes larger. As a result, the texture gets stretched across a larger area.



- When the plane is made smaller, the texture appears compressed.



This effect demonstrates the importance of texture mapping and scaling in visualizations. Proper texture mapping ensures that the textures appear proportional, without being distorted or stretched unnaturally.

Transformation

To manipulate objects that do not have built-in position or orientation methods, transformations are applied using the `vtkTransform` and `vtkTransformPolyDataFilter` classes. The transformation defines a translation along the Z-axis, and this transformation is applied to the plane:

```
plane = vtkPlaneSource()
MyTransform = vtkTransform()
MyTransform.Translate(0, 0, -1)

MyFilter = vtkTransformPolyDataFilter()
MyFilter.SetTransform(MyTransform)
MyFilter.SetInput(plane.GetOutput())

planeMapper = vtkPolyDataMapper()
planeMapper.SetInput(MyFilter.GetOutput())

planeActor = vtkActor()
planeActor.SetMapper(planeMapper)
```

Six planes were defined, with a texture, rotation and transformation attributes:

```
faces = [
        ("images/Im1.jpg", (0, 0, 0), (0, 0, 0.5)),
        ("images/Im2.jpg", (0, 0, 0), (0, 0, -0.5)),
        ...
        ("images/Im6.jpg", (90, 0, 0), (0, 0, -0.5)),
    ]
```

And a for loop to render all the faces of the cube:

```
for img, rot, trans in faces:
        actor = make_plane_actor(img, rot, trans)
        renderer.AddActor(actor)
```

Calls for Interaction

An important feature of VTK is its ability to react to user interactions via callback functions. In this exercise, a custom callback is defined that prints the camera position whenever the scene is interacted with. The observer is added to the renderer, allowing real-time feedback on the interaction:

```
class vtkMyCallback(object): def init(self, renderer): self.ren = renderer

def __call__(self, caller, ev):
    print(caller.GetClassName(), 'Event Id:', ev)
    print("Camera Position: %f, %f, %f" % (self.ren.GetActiveCamera().GetPosition()[0],
self.ren.GetActiveCamera().GetPosition()[1],
self.ren.GetActiveCamera().GetPosition()[2]))

mo1 = vtkMyCallback(ren)
ren.AddObserver(vtkCommand.AnyEvent, mo1)
```

When the code is executed, every single event, like StartEvent, EndEvent and ResetCameraEvent, is logged in the command line:

```
vtkOpenGLRenderer Event Id: StartEvent
Camera Position: -1.080221, 1.145351, 2.790060
vtkOpenGLRenderer Event Id: EndEvent
Camera Position: -1.080221, 1.145351, 2.790060
vtkOpenGLRenderer Event Id: ModifiedEvent
Camera Position: -1.080221, 1.145351, 2.790060
vtkOpenGLRenderer Event Id: StartEvent
Camera Position: -1.080221, 1.145351, 2.790060
vtkOpenGLRenderer Event Id: EndEvent
Camera Position: -1.080221, 1.145351, 2.790060
```

It was also tested logging specific events like:

- EndEvent, that only logs the end of any event

```
vtkOpenGLRenderer Event Id: EndEvent
Camera Position: 0.000000, 0.000000, 3.203614
vtkOpenGLRenderer Event Id: EndEvent
Camera Position: -0.329446, 0.114133, 3.184585
vtkOpenGLRenderer Event Id: EndEvent
Camera Position: -0.654768, 0.227515, 3.127724
vtkOpenGLRenderer Event Id: EndEvent
```

- StartEvent, that only logs the start of any event

```
vtkOpenGLRenderer Event Id: StartEvent
Camera Position: 0.000000, 0.000000, 3.203614
vtkOpenGLRenderer Event Id: StartEvent
Camera Position: -0.339678, 0.156030, 3.181731
vtkOpenGLRenderer Event Id: StartEvent
Camera Position: -0.674310, 0.310810, 3.116383
vtkOpenGLRenderer Event Id: StartEvent
Camera Position: -0.999321, 0.462223, 3.008462
vtkOpenGLRenderer Event Id: StartEvent
```

- ResetCameraEvent, that only logs when the camera is reset

```
vtkOpenGLRenderer Event Id: ResetCameraEvent
Camera Position: 0.000000, 0.000000, 3.203614
vtkOpenGLRenderer Event Id: ResetCameraEvent
Camera Position: -0.998473, 2.006358, 2.289261
vtkOpenGLRenderer Event Id: ResetCameraEvent
Camera Position: -0.998473, 2.006358, 2.289261
vtkOpenGLRenderer Event Id: ResetCameraEvent
Camera Position: -0.998473, 2.006358, 2.289261
```

Conclusion

Lesson 2 builds upon the foundational knowledge of VTK and extends it with advanced techniques for 3D visualization. By exploring multiple renderers, shading options, textures, and transformations, it was possible to understand how to enhance the realism and interactivity of visualizations. Additionally, the use of callbacks for dynamic interactions demonstrates how VTK can be employed to create responsive and user-friendly visual applications.