2st Lab with VTK (Visualization Tool Kit)

Information Visualization

Daniel Madureira, 107603
José Gameiro, 108840

***Resumo* - This document presents a brief description of the work developed for the second lab of the class Information Visualization, with the objective to understand how projections, lighting and transformations work within VTK.**

I. Introduction

The **Visualization Toolkit** (**VTK**) is an open source software system widely used for 3D computer graphics, image processing, and scientific visualization. The report focuses on key features of VTK, including the use of **multiple actors** and **renderers** to compose complex scenes, **shading options** and **textures**. Additionally, it delves into **transformations** for modifying object orientations and positions, as well as the implementation of **observers and callbacks**.

II. Development

The lab provided by the class professor has multiple exercises that follow the following topics:

- Multiple actors
- Multiple renderers
- Shading options
- Textures
- Transformations
- Observers and callbacks

A. *Multiple actors*

For the first exercise, it was required to add multiple actors to one render. In this case, we were asked to create two identical cones with different colors. We used different approaches to manipulate the properties of each cone: In the first case we setted each property individually, while in the latter we created a new ***VTKProperty*** and after tweaking it we assigned it to the actor.. We were also asked to drop the opacity to 0.5 in both cones:

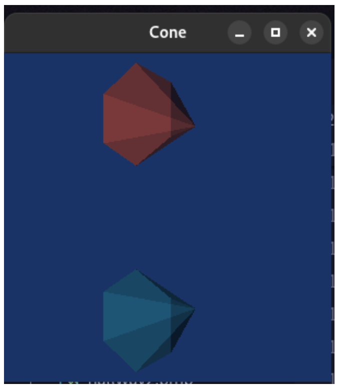

Fig. 1 - Result of the execution of the first exercise

We concluded that when we dropped the opacity, the other faces were visible as if the cones were made of acrylic. Other than that, the application of properties achieves the same result with the methods used to change it for each cone.

B. *Multiple renderers*

This exercise is a simple one: we had to display two renderers inside one window. For this, we added the cone exercise from the last lab and divided the window (set to 600 x 300 px) into two using the ***SetViewport*** method. The second renderer had its camera position rotated 90 degrees in azimuth. Finally, we changed the backgrounds for each renderer to better distinguish them:

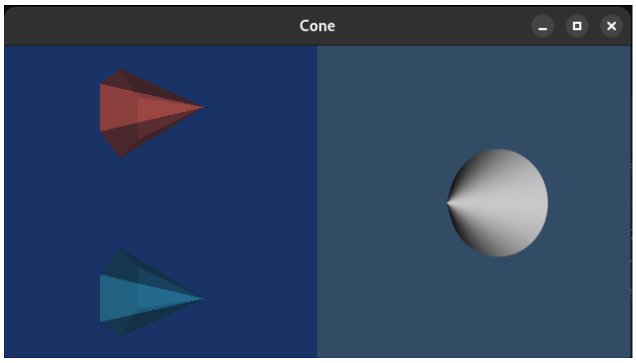

Fig. 2 - Result of the execution of the second exercise

C. *Shading options*

For this exercise, we replaced our cones with spheres. The objective was to explore and apply different shading methods offered by VTK to each sphere. To better observe the differences, we reduced the resolution of the spheres to 10 (for both phi and theta) and created four renderers, each displaying one sphere (instead of the required two renderers and two spheres). We then set one sphere without a shading option, one with ***SetInterpolationFlat***, another with ***SetInterpolationGouraud***, and the last with ***SetInterpolationPhong***:

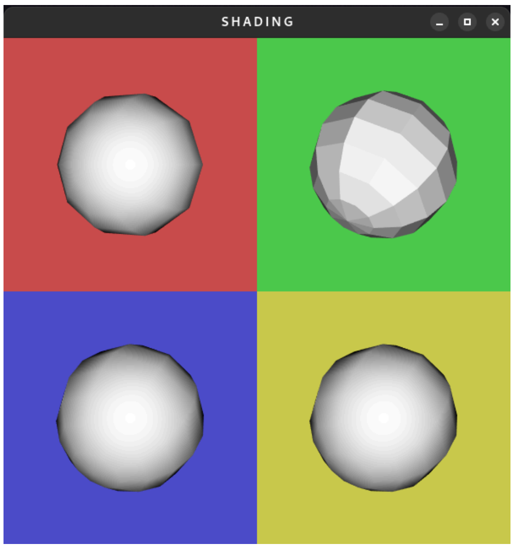

Fig. 3 - Result of the execution of the shading exercise

We observed that, by default, VTK applies Gouraud shading to its objects. We concluded that Phong and Gouraud shaders often produce similar outcomes, but Phong shading tends to create smoother lighting and more realistic specular highlights. The Flat shader, in contrast, is the most visually distinct, producing a faceted look due to uniform lighting on each polygon.

D. *Textures*

For this exercise we created a plane and applied a texture to it. In order to do so, we used the ***vtkJPEGReader*** to parse a JPEG file as a texture. Then we create the texture using ***vtkTexture*** and parsed the output from the vtkJPEGReader. The result was a plane with the image as the texture as follows:

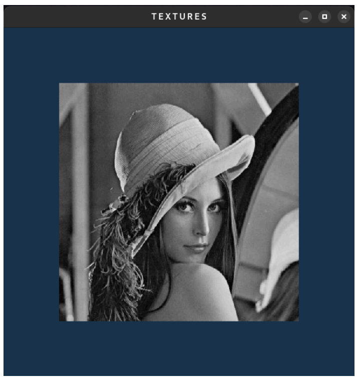

Fig. 3 - Result of the execution of the texture exercise

E. *Transformation*

This section encompasses two exercises. The first exercise involves applying custom transformations to a plane. Specifically, we use ***vtkTransform*** to define a translation vector of (0.5, 2, -1), and then apply the transformation to the plane using ***vtkTransformPolyData- Filter***. To ensure the transformation is effectively applied to the actor, we pass the filter's output to the plane's mapper. The result is a translated plane, with the sphere marking the origin (0, 0, 0):

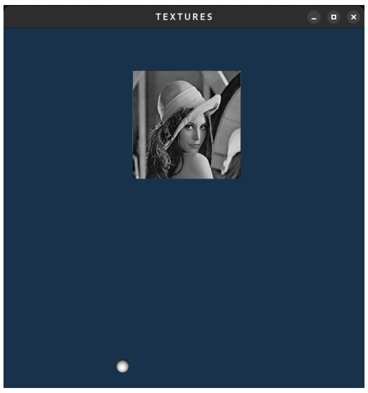

Fig. 3 - Result of the execution of the transformation exercise

The second half of the exercise was to create a cube made of planes. In order to do that we created one function to render a cube face, where a ***vtkTransform*** is applied to rotate and translate the plane. The rotation is specified as an angle (in degrees) and a 3D axis vector, while the translation moves the plane in the 3D space. The transformations are applied using the ***vtkTransformPolyDataFilter*** to the plane geometry. For the texture of each facet, a ***vtkJPEGReader*** was used to read the different images available and then with the help of the ***vtkTexture***, each image is mapped to each facet of the cube. The following image shows the results obtained.

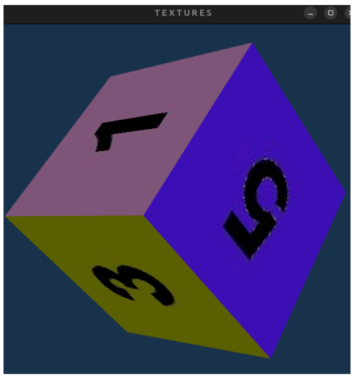

Fig. 4 - Result of the execution cube.py file

F. *Use of callbacks for interaction*

For the last part of the lab, it was asked to explore the feature of defining **Callback functions** and associate them to an object.To achieve this the class ***vtkMyCallback*** was created to the **cone.py** of the first lab, where it prints information about the event that occurred in the terminal like the id of the event or the position of the camera. This only happens when the event is triggered.

When the **cone.py** file is executed, the callback is only executed every time the rendering process finishes. If the type of the event is changed to **Start Event**, or **ResetCameraEvent,** then the callback will be triggered at the beginning of the rendering process and when the camera view is reset.