3st 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 third lab of the Information Visualization class, with the objective to understand how glyphing, picking, unstructured grid and scalar association to vectors and grids work.**

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 **glyphing, picking**, **unstructured Grid** and **scalar association to vectors** and **grids.**

II. DEVELOPMENT

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

- Glyphing
- Picking
- Unstructured Grid
- Scalar association to vectors and grids
- HedgeHog

A. *Glyphing*

For the first exercise, it was asked to use the ***vtkGlyph3D*** class from **VTK**, to learn more about glyphing and how we can represent data using geometric representations or glyphs. For this we needed to represent a **sphere** with **cones** placed at each point of the sphere's polygonal model. To accomplish this we developed 2 functions, one to render a sphere and another to render a cone. Then, in our main function, we added the following lines:

```
glyph = vtkGlyph3D()
glyph.SetSourceConnection(coneSource.GetOutputPort())
glyph.SetInputConnection(sphereSource.GetOutputPort())
glyph.SetScaleFactor(0.25)
glyph.SetVectorModeToUseNormal()
```

Where, it first creates a glyph generator that copies the cone to each point of the input data. Then it specifies the points from the sphere as the locations for the glyphs and scales the glyphs to 25% of their default size. Finally it orients the glyph based on the normals of the sphere's surface. It was also asked to understand the methods ***SetScaleFactor*** and ***SetVectorModeToUseNormalMethods***, and concluded that the first one controls the size of the glyphs (in this case the cones), the second one orients the glyphs based on the **normals** of the input data. We obtained the following results

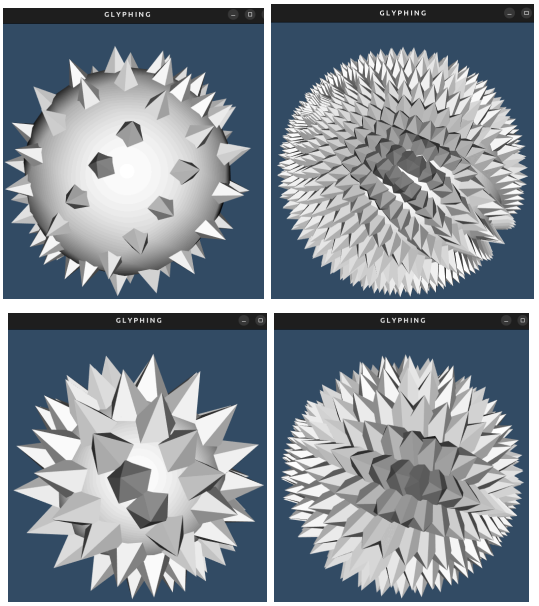

Fig. 1 - Results obtained by executing the first exercise with different values for the **Scale Factor, Theta and Phi resolution:** the first image (0.25, 10, 10), the second image (0.25, 30, 30), the third image (0.5, 10, 10) and the fourth image (0.5, 20, 20).

B. *Object Picking*

On this exercise it was requested to add an Object **Picker** to the interactor of the previous example to allow the selection of points of the model. To accomplish this it was required to render a new sphere but set it's visibility to **False,** so it will only display when a pick event is triggered, it was also necessary to create a callback function to do something when the pick event is triggered, and we developed the following:

```
def pointer_callback(picker: vtkPointPicker, event):
    pickPosition = picker.GetPickPosition()
    print(f"Picked point coordinates: {pickPosition}")

    selectedSphereActor.SetPosition(pickPosition)
    selectedSphereActor.SetVisibility(True)
    selectedSphereActor.GetProperty().SetColor(1.0, 0.0, 0.0)
    renderWindow.Render()
```

This function updates a red sphere's position when a point is picked, where it first gets the picked point's coordinates, then it moves the red sphere to that point, makes it visible and updates the window. It also prints in the terminal, the position of the sphere.

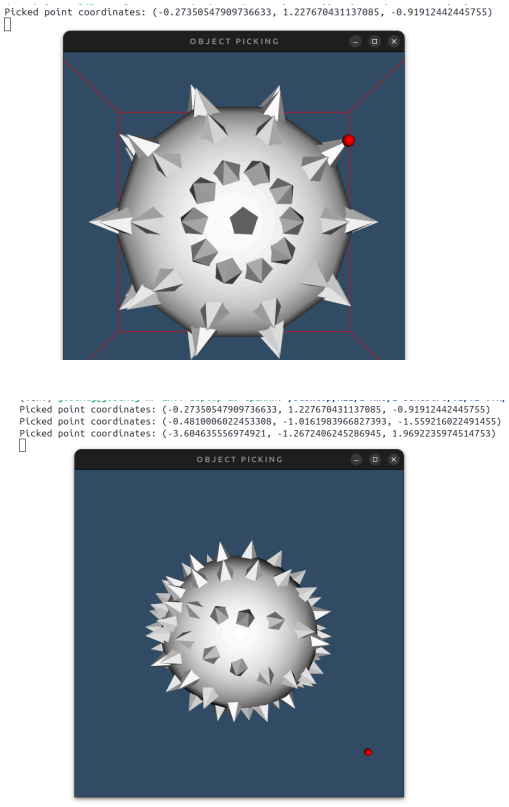

Fig. 2 - Result of the execution of the second exercise

C. *Display Coordinates*

An additional exercise to the object picking topic, was instead of displaying the coordinates in the terminal, display them in the renderer window. To complete this, we used a **Text Mapper** and a **2D Actor** to visualize the coordinates of the selected point in the renderer window. Like the sphere the text first has is visibility set to False since no point picker event was triggered. When an event of this type is triggered, some changes were implemented in the callback function, where it first converts the picked position to a string **(x, y, z)** with two decimal places, then it updates the text to show the coordinates, makes it visible, moves it slightly offset from the picked point and finally it updates the renderer window:

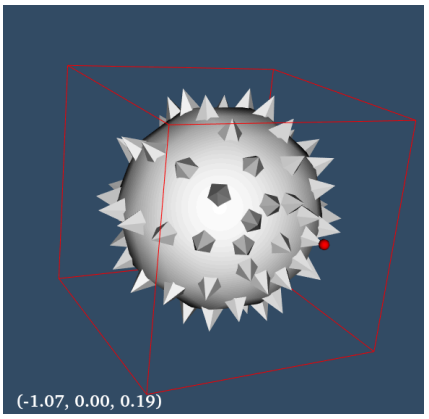

Fig. 3 - Result of the execution of the shading exercise

As you can see in the image above the text has a **bold** weight and we also changed the size of the font, by using the **SetFontSize** and **SetBold** methods of a text property.

D. *Unstructured Grid*

For this exercise a file with some code was provided that makes use of the vtk's **Unstructured Grid.** It creates an unstructured grid with only one cell, a tetrahedra. When we executed the file, we obtained this result:

Fig. 4 - Result of the execution of the code provided by the professor.

It was requested to, instead of displaying a tetrahedron with four vertices (**VTK_TETRA**), create individual cells for each vertex using **VTK_VERTEX**, where each point is treated as its own cell. The loop **for i in range(len(coords))** creates these cells by associating each coordinate with a **VTK_VERTEX**. We also modified the actor's properties to set the color and the point size to five, making the points more visible. The background color of the renderer was adjusted to a light gray for better contrast. These changes result in a visualization showing clearly defined red points instead of a connected tetrahedron:

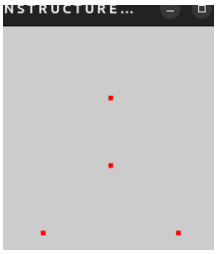

Fig. 5 - Results obtained by executing the ***ugrid.py*** file

E. Scalar association to vectors and grids

In this section, the exercise required creating an unstructured grid with four points and associating each point with a vector and a scalar value. The vectors, represented as ***(1, 0, 0)***, ***(0, 1, 0)***, ***(0, 0, 1)*** and ***(1, 1, 1)***, were stored in a ***vtkFloatArray*** with three components. These vectors were used to orient cones at the grid points, visualized using the ***vtkGlyph3D*** class. Additionally, scalar values ***(0.1, 0.3, 0.5, 0.8)*** were assigned to each point to color the cones based on the scalar data.

The code first creates an unstructured grid and adds the four points as separate ***VTK_VERTEX*** cells. It then defines two ***vtkFloatArray*** objects: one for the vectors and one for the scalars, associating them with the grid points. A ***vtkGlyph3D*** object generates cones at each point, oriented by the vector data and scaled by the scalar data. The cone's colors vary based on the scalar values, and the visualization is rendered with a light gray background. We obtained the following result:

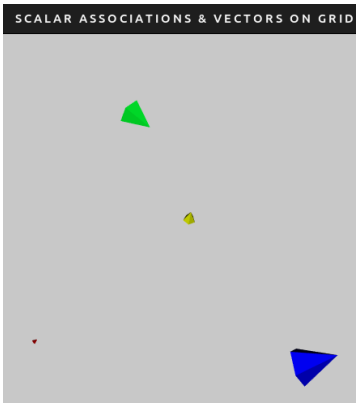

Fig. 6 - Results obtained by executing the ***scalar_associations_vectors_grids.py*** file

F. HedgeHog

For the final exercise, it was asked to explore the visualization of vector data using the ***vtkHedgeHog*** class. Unlike the previous exercise, where the glyphs (cones) were used to represent vectors, ***vtkHedgeHog*** represents vector data as line segments. The task was to adapt the previous exercise to use ***vtkHedgeHog***.

The code was modified to replace ***vtkGlyph3D*** with ***vtkHedgeHog*** for vector representation. This class was configured to use the vector data from the unstructured grid and scaled using a **scale factor** of 0.3, to control the length of the line segments. We obtained the following results:

Fig. 7 - Results obtained by executing the ***edge_hog.py*** file