Visual Tool Kit (VTK) - Lesson 3

Miguel Pinto
Student 107449, DETI
Aveiro University
Aveiro, Portugal
miguel.silva48@ua.pt

Tiago Figueiredo
Student 107263, DETI
Aveiro University
Aveiro, Portugal
tiago.a.figueiredo@ua.pt

I. Solved Exercises

A. Glyphing

To create this visualization, we first generated a sphere using the `vtkSphereSource` class and adjusted its resolution using the `SetThetaResolution` and `SetPhiResolution` methods. The sphere served as the input for the glyphing process. Next, we created a cone using the `vtkConeSource` class, which was used as the glyph symbol.

The `vtkGlyph3D` class was then configured to combine the sphere and the cone:

- The `SetVectorModeToUseNormal` method aligned the cones with the sphere's normals, ensuring the cones pointed outward from the sphere's surface (the default behaviour is for the cones to point in positive direction of the X-axis).
- The `SetScaleFactor` method was used to adjust the size of the cones relative to the sphere (the bigger the factor, the bigger the cone glyphs, obviously).
- The sphere methods `SetThetaResolution` and `SetPhiResolution` change the amount of cones (glyphs) around the longitudinal and latitudinal axis of the sphere, respectively.

The end results can be seen in Figure 1.

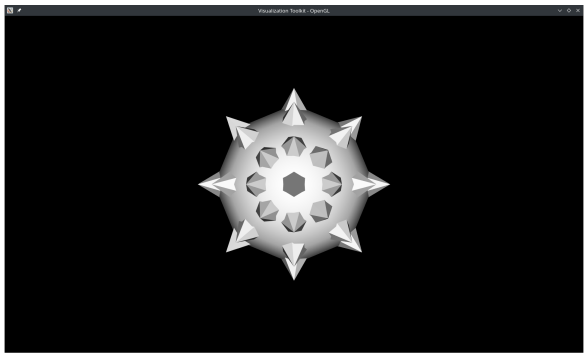

Fig. 1. Sphere with cones (glyphs) aligned to normals using `vtkGlyph3D`.

B. Object Picking

In this exercise, we extended the previous visualization by introducing object picking functionality using the `vtkPointPicker` class. A callback was associated with the picker to handle the selection of points on the sphere. Upon picking a point (using the `P` key), the program retrieves the coordinates of the selected point and updates the position of a small red sphere to indicate the selected location.

Additionally:

- The `VisibilityOn` and `VisibilityOff` methods were used to control the visibility of the red sphere, ensuring it only appears after a point is selected.
- The callback uses the `GetPickPosition` method to obtain the world coordinates of the picked point and dynamically adjust the indicator sphere's position.

The final result can be seen in Figure 2.

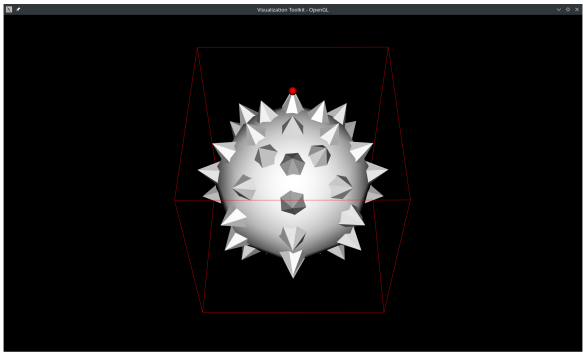

Fig. 2. Point picking with the red sphere indicating the selected location.

C. Display of Coordinates

In this exercise, we extended the point-picking functionality by adding a visual display of the selected point's coordinates directly in the renderer, using the `vtkTextMapper` and `vtkActor2D` classes to display text next to the selected point.

The implementation involved the following steps:

- The `vtkTextMapper` was used to manage the text displaying the 3D coordinates of the picked point. Its properties were customized to use a Courier font, set to bold, and center the text (as requested).
- A `vtkActor2D` was employed to render the text in the 2D overlay of the visualization.

- The callback was updated to retrieve the 2D pixel coordinates (using the `GetSelectionPoint` method) and dynamically position the text actor using the `SetPosition` method.
- The visibility of the text actor was controlled with `VisibilityOn` and `VisibilityOff`, so it only appeared after a point was picked (similarly to the red sphere).

The final result is illustrated in Figure 3.

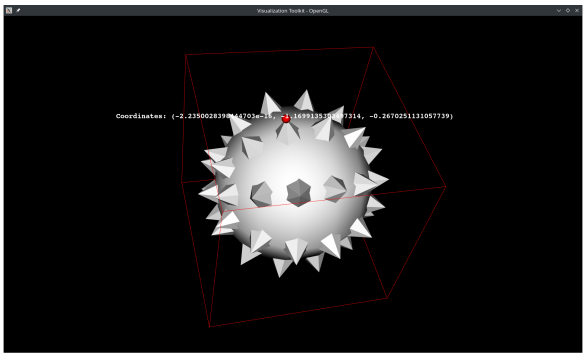

Fig. 3. Visualization of picked point coordinates displayed on the renderer.

D. Unstructured Grid

This exercise involved working with an unstructured grid in VTK to visualize vertices as opposed to a tetrahedron. Initially, the program displayed a single tetrahedron composed of four points. The following modifications were made to the code:

- The cell type was changed from `VTK_TETRA` to `VTK_VERTEX`, resulting in the visualization of separate vertices instead of the tetrahedron.
- The actor's properties were updated using the `SetColor` and `SetPointSize` methods to make the vertices more visually distinct.

The final result is shown in Figure 4 (note that the vertex size was tripled just so it could be visible in the report).

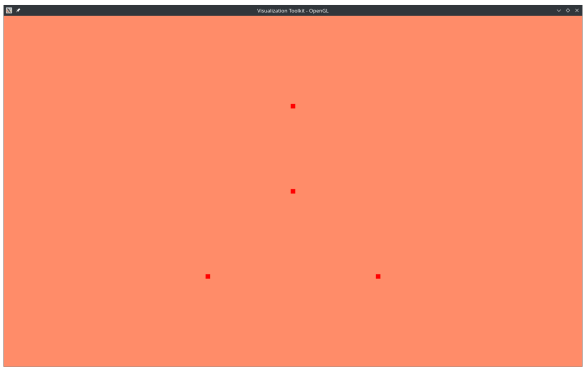

Fig. 4. Visualization of separate vertices in an unstructured grid.

E. Scalar Association to Vectors and Grids

This exercise involved associating vectorial and scalar information to the vertices of an unstructured grid and visualizing the data using glyphs (cones) oriented and scaled accordingly.

We performed the following steps to complete the exercise:

- **Vector and Scalar Data Association:**
  - A vector field was defined with directional information for each grid vertex.
  - A scalar field was created to associate a magnitude value to each vertex.
- **Glyph Visualization:**
  - Cones were used as glyphs, oriented according to the vectors and scaled by the scalar values.
  - The `vtkGlyph3D` class provided functionality for automatic alignment and scaling of the cones.
- **Visualization Options:**
  - Different methods in `vtkGlyph3D` were explored to adjust the glyphs' orientation and scaling.

The resulting visualization featured cones that represent vector direction and magnitude through their orientation and size, with scalar values additionally affecting the colour of the glyphs. The final result is shown in Figure 5.

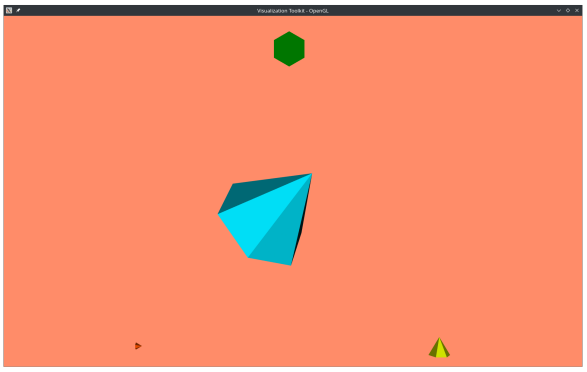

Fig. 5. Glyph visualization with orientation based on vector data and scaling/colour based on scalar values.

F. HedgeHog

This exercise focused on visualizing vector data using the `vtkHedgeHog` class, which represents vector information as line segments emanating from the grid points.

To complete it we performed the following steps:

- **Vector Data Association:**
  - The same vector data from the previous exercise was reused, associating directional information to each vertex of the grid.
  - The vectors were stored in a `vtkFloatArray` and linked to the points of the unstructured grid using `SetVectors`.
- **HedgeHog Visualization:**

  - The `vtkHedgeHog` class was used to visualize the vectors as line segments.
  - The length and orientation of each line corresponded to the magnitude and direction of the associated vector.
- **Renderer Setup:**
  - A mapper and actor were created for the `vtkHedgeHog` output.
  - The visualization was integrated into a render window with a defined background color.

The resulting visualization represented the vector field as simple line segments, providing an alternative and more minimalistic representation compared to the glyph-based approach. The final result can be observed in Figure 6.

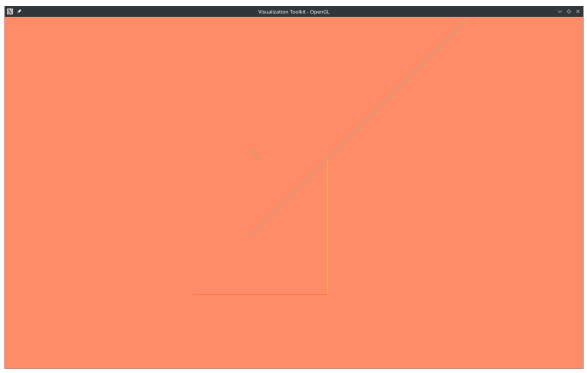

Fig. 6. Vector visualization using the `vtkHedgeHog` class. Line segments indicate vector direction and magnitude.

II. Conclusions

Through the exercises performed, we explored key functionalities of the VTK library, including glyphing, object picking, and working with unstructured grids. The progression from basic object manipulation to more advanced data representations, such as glyph scaling and HedgeHog visualizations, provided a good understanding of how to effectively map data onto 3D geometries, emphasizing both clarity and aesthetic presentation.

References

[1] VTK Documentation. Available at: https://vtk.org/doc/nightly/html/
[2] Python VTK Wrapper. Available at: https://pypi.org/project/vtk/