Computer Modeling: New Research 2012
Guests
(Editors)

Volume 0 (2012), Number 0

Design and Implementation of a System for Interactive High-Dimensional Vector Field Visualization

Zhenmin Peng, Zhao Geng, and Robert S. Laramee

The Department of Computer Science, Swansea University, United Kingdom.
Email:{cszp, cszg r.s.laramee}@swansea.ac.uk

Abstract

Although the challenge of 2D flow visualization is deemed virtually solved as a result of the tremendous amount of effort invested into this problem, high-dimensional flow visualization, (e.g. the visualization of flow on surfaces in 3D (2.5D), the volumetric flow (3D), and flow with several attributes (nD)), still poses many challenges and unsolved problems. In this paper we describe the design and implementation of a generic framework incorporating a selection of related scientific and information visualization techniques which are designed and integrated to provide the user solutions for effective visualization of the high-dimensional CFD flow simulation data. In contrast to most research prototypes, the system we present handles real-world, unstructured simulation data. Our framework provides direct, feature-based and geometric flow visualization techniques and supports information visualization approaches, such as a tabular histogram, velocity histogram, and parallel coordinate plot. In order to enable a smooth and efficient user interaction, these visualization options are systematically combined on a multi-threading platform which ensures responsiveness even when processing large high-dimensional data.

1. Introduction

Over the last three decades, computational fluid dynamics (CFD) has developed very rapidly. Its applications range widely from the automotive industry to medicine [LEG*08]. This is because CFD *modeling* and *simulation* speed up the manufacturing process. Constructing objects and simulating experiments in a software environment is normally faster and cheaper than building and testing physical hardware counterparts in real world. As another important part of the CFD pipeline, the *visualization* process not only provides the engineer the visual result of the simulation, but also verifies or conflicts with the results expected by the engineer so that the original model design can be approved or improved. The CFD process, illustrated in Figure 1, is composed of three main stages:

1. Modeling: a 3D structured or unstructured, volumetric or surface mesh is generated to model the physical object. This procedure is based on computer aided design (CAD) modeling.
2. Simulation: a computational simulation of a fluid through the given model in the previous stage is computed in a 3D simulation environment with a set of given initial conditions.
3. Visualization: the simulation result is explored, analyzed, and visualized in different ways according to different needs.

Since the size, complexity and dimensionality of the CFD simulation data have dramatically increased in recent years, so has the need for visualization which provides quick and effective insight into the data [PL09]. In order to present a visualization toolkit which is capable of dealing with large, complicated, and high-dimensional CFD simulation data, a comprehensive and versatile visualization framework is needed. In this paper we focus on the design and implemen-

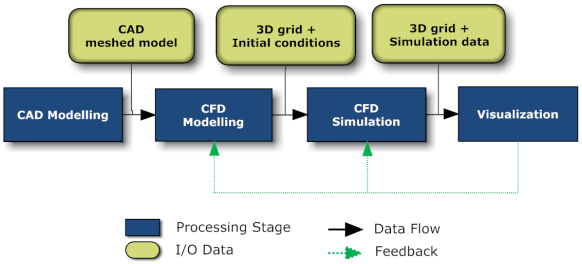

Figure 1: *The CFD process is composed of modeling, simulation, and visualization stages.*

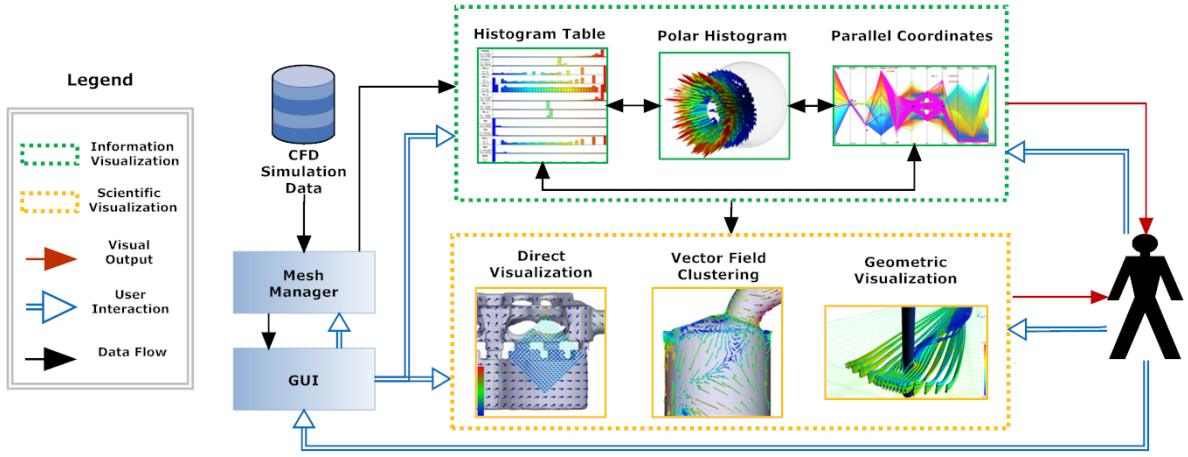

Figure 2: *An overview the application framework.*

tation of a generic visualization framework which provides the user solutions for effective visualization of the high-dimensional CFD simulation data by combining several scientific and information visualization techniques. This visualization framework yields following benefits:

- The framework handles versatile real-world, unstructured 2.5D, 3D, and *n*D CFD simulation data.
- Direct, geometric, and feature-based flow visualization techniques are integrated in order to support the CFD engineer with intuitive and rich visualizations for effective visual analysis.
- Information visualization approaches, such as tabular histogram, velocity histogram, and parallel coordinate plot (PCP), are incorporated to enable engineers to gain an in-depth analysis of the simulation data and thus focus on parts they deem interesting.
- Smooth and efficient user interaction is ensured by our multi-threading application, even when large data sets are processed.
- Our flow visualization framework is platform independent in terms of both hardware and software.
- The framework is a open source project with simple API provided. The full system document generated by Doxygen [vH] is available online (`http://cs.swan.ac.uk/~cszp/mt_docs/index.html`). Bob's coding conventions [Lar10] are applied.

We discuss the details of several aspects related to the design and implementation of our flow visualization framework. The corresponding advantages and disadvantages are also discussed. Our presentation here provides much more detail about the design and implementation of our software than a typical visualization research paper. Also, typical research prototypes are unable to process real-world CFD data (like that we present here).

The rest of the paper is organized as follows: Section 2 provides an overview of related research work. Section 3 discusses the overall design of the visualization framework while details of implementation and design of our flow visualization framework are presented in Section 4. Section 5 evaluates the design and implementation and discusses some advantages and disadvantages of the framework. Conclusions and suggestions for future work are presented in Section 6.

2. Related Work

In this section, we discuss some design and implementation related work. Baldonado et al. [WBWK00] present a set of guidelines for system designers to make the design of a system with multiple views more systematic and efficient. The first four guidelines (diversity, complementarity, parsimony, and decomposition) provide the designers with suggestions on selection of multiple views. The last four (space/time resource optimization, self-evidence, consistency, and attention management) help designers make decisions on view presentation and interaction.

Doleisch et al. [DGH03] [Dol07] describe the design and implementation of the SimVis which enables interactive visual exploration and analysis of large, time-dependent, and high-dimensional data sets resulting from CFD simulation. Weaver [Wea04] discusses the design of a multiview visualization system, Improvise, which utilizes a shared-object coordination mechanism using visual abstraction language. He also presents a cross-filtered views implementation [Wea09] based on Improvise for multiple dimensional visual analysis. Laramee et al. [LHH05] describe the design and implementation of a flow visualization subsystem which utilizes the geometric and texture-based flow visualization techniques. In order to guarantee the quick responsiveness for user interaction even when dealing with large data, Piringer et al. [PTMB09] present a generic multi-threading architecture which allows early cancellation of the visualization thread due to user interaction without common pitfalls of multi-threading. They also present an interactive visualization toolkit, HyperMoVal [PBK10], as an implementation of

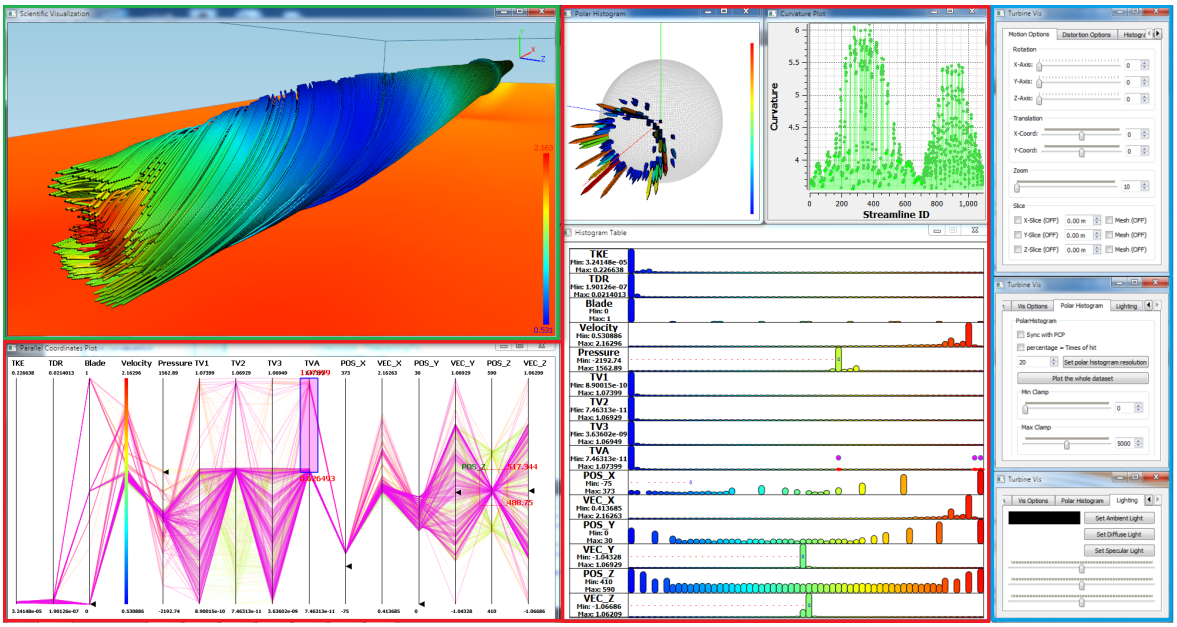

Figure 3: *A screen shot of our flow visualization framework applied to visualize the flow past marine turbines [PGSC11]. The multi-linked application GUI consists of three distinct parts: (1) Information Visualization Windows (outlined in red) providing various data drilling widgets such as histogram table, parallel coordinate plot, etc. (2) A Scientific Visualization Window (outlined in green) rendering the final visual result in 3D based on the filtered data. (2) The tabbed Console Menu (outlined in sky blue) enables the user to update the parameters intuitively and interactively.*

this architecture in practice. Fisher et al. [FDFR10] present a framework called WebCharts by which existing information visualizations can be plugged into a variety of host applications. WebCharts is like a API of visualization library which encourages greater reuse of existing visualization in host applications, so that users can do visualization locally, yet new visualizations can be updated and obtained via the API from related websites. Peng et al. [PGSC11] present a multi-linked framework which provides customized visualization techniques for engineers to gain a fast overview and intuitive insight into the flow past the marine turbine.

3. Overview of Visualization System Design

Figure 2 illustrates an overview of the framework. The input is the CFD flow simulation and associated mesh. It is characterized by the high-dimensional data which includes position, velocity and various simulation attributes such as pressure, viscosity, turbulent kinetic energy, etc. Since the mesh is often unstructured and adaptive resolution, the mesh manager is used to compute and store the mesh topology information as a preprocessing step. After the mesh adjacency construction, various information visualization approaches are employed to gain insight into the data. The histogram table provides an intuitive overview of the multi-dimensional attributes of the whole simulation. After gaining an overview based on the histogram table, the user can focus on attributes they deem interesting, while the polar histogram and PCP simultaneously depict the details of the focus attributes. The polar histogram presents an intuitive visual summary of the flow velocity distribution. The PCP highlights the relationship between CFD attributes to support exploration and analysis. Several flow visualization approaches are applied to provide rich visual analysis. Interactive glyph visualization provides a quick and direct hands-on exploration on the vector field, while a continuous representation can be obtained by a geometric flow visualization such as streamlines. The automatic vector field clustering produces intuitive and insightful images of vector fields. The user can interact between different information visualization approaches to obtain the final scientific visualization result. Figure 3 shows our application based on this framework.

4. System Design and Implementation

In following subsections, we describe two main subsystems in more detail about design and implementation: the **Information Visualization Subsystem** and the **Scientific Visualization Subsystem**. This is where the majority of our research work was done. These two subsystems involve a fairly complex set of classes and associated responsibilities. In order to elaborate these intuitively and show the framework is really working, Standard UML diagrams [Fow03] and Doxygen diagrams [vH] generated based on a project named “Marine Turbine Visualization” [PGSC11] are pro-

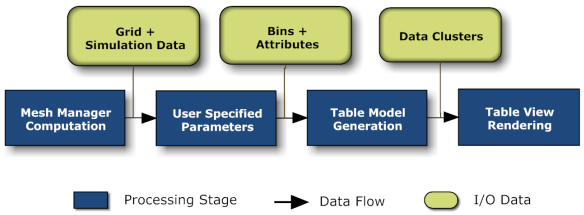

Figure 4: *The processing pipeline of the histogram table visualization design.*

vided. **User Interface Design** is also presented with details at the end of this section.

4.1. Information Visualization Subsystem

Here we detail how the information visualization subsystem is designed and implemented. Each part of the subsystem is examined with details such as the class relationship, the hierarchy of class components, the collaboration between different classes.

4.1.1. Histogram Table Visualization

In order to quickly and efficiently present a large amount of multidimensional data, it's desirable to provide a quick overview of the entire data set. For this, we incorporate a histogram table. See Figure 5. The histogram table represents the distribution of multidimensional information across the data set in an interactive visualization. Figure 4 illustrates the main processing pipeline of the histogram table visualization subsystem. The input and output data is shown in rectangles with rounded corners and processes are shown in boxes.

The mesh manager preprocesses the input CFD flow simulation data and related mesh in order to obtain the mesh topology information and the range of values for each simulation attribute. After the user specifies their input requirements for the histogram table, such as the bin number of each

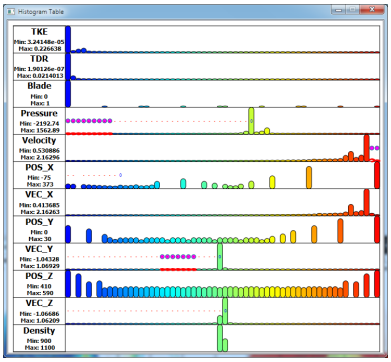

Figure 5: *A histogram table is used to provide an overview of the multidimensional information from the turbines simulation [PGSC11].*

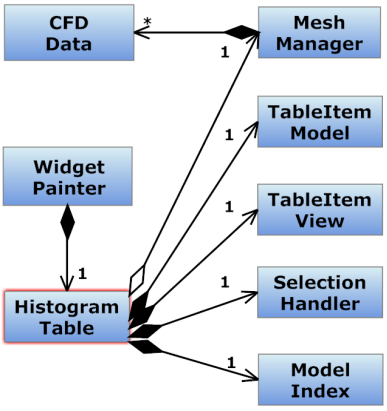

Figure 6: *A screen shot of relationship among the major components of histogram table visualization implementation. UML notation is used.*

histogram, the histogram height, the order of attributes and the color-mapping parameters, a table model is generated to group the preprocessed values into clusters. Based on the table model, the table view renders a histogram table which enables the user interaction on the table model. The user is able to select the items from the histogram table for further exploration.

In order to illustrate the class relationship among the major components of the histogram table implementation, a UML class diagram [Fow03] is drawn in Figure 6. The black diamond shape arrow indicates the *composition* which defines a "owns a" relationship while the white shows a "has a" *aggregation* relationship. A 1 or $*$ in the figure represents the multiplicity of the relationship. For example, the **TableItem Model** object is owned by the **Histogram Table** object and the relationship is one-to-one. Here we describe the major components shown in Figure 6. The **Histogram Table** is the class with the most responsibilities shown in Figure 4. The **Mesh Manager** class is responsible for processing and managing the raw **CFD data** from CFD flow simulation. All the information used in this subsystem is filtered and provided by the mesh manager. The **TableItem**

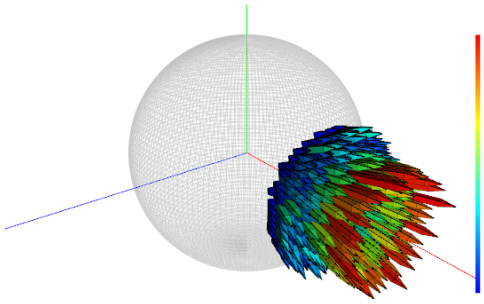

Figure 7: *A polar histogram is used to illustrate the velocity distribution of the tidal flow around the blade element.*

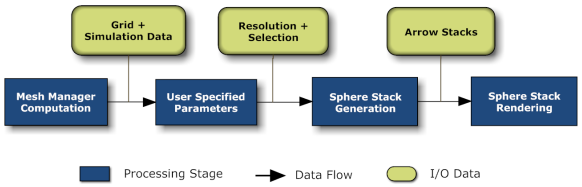

Figure 8: *The processing pipeline of the polar histogram visualization design.*

Model object is responsible for defining a table model which contains all the grouped information and delegates access to data. The **Model index** is the index object for locating data in the table model. The **TableItem View** object is used to display data from the TableItem Model. The TableItem Model and the TableItem View are constructed based on the model/view architecture. The **Selection Handler** class is responsible for tracking and managing the user selection from the table view. The **Widget Painter** class provides the general 2D rendering mechanism for the histogram table.

4.1.2. Polar Histogram Visualization

In order to visualize the direction in which the majority or minority of velocity of the whole domain or the user selected region points, we have a view called the polar histogram which is originally inspired from the global animal tracking visualization by Grundy et al. [GJL*09] for an integrated view of velocity distribution in a 3D spherical coordinate system. See Figure 7. This visualization delivers an intuitive and direction-oriented visual result of the velocity distribution. Figure 8 demonstrates the main design pipeline of this visualization subsystem. Similar to the first step of the histogram table, the preprocessed mesh and simulation data are obtained as the input by the mesh manager. The polar histogram is initialized as a sphere wireframe whose bin resolution is specified by the user. Each cell of the sphere wireframe depicts a range and frequency of velocity direction.

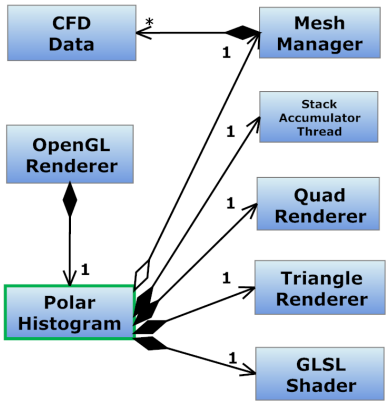

Figure 9: *A screen shot of relationship among the major components of polar histogram visualization implementation. UML notation is used.*

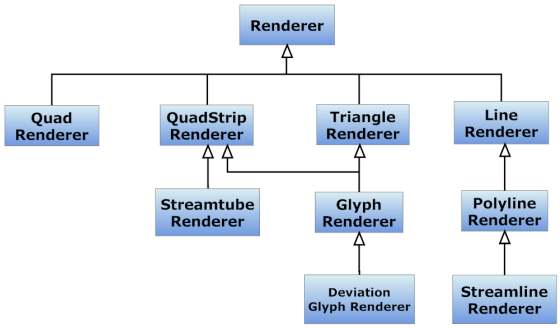

Figure 10: *A class hierarchy of object Renderer options. UML notation is used.*

Based on the user selection of the data, sphere stacks are generated on top of the corresponding sphere wireframe cells with height associated to the frequency of vectors pointing in the direction. Finally stacks are rendered with arrow tips to represent the direction.

4.1.3. Parallel Coordinate Plot

Figure 9 shows the class relationship between the major components of the polar histogram subsystem implementation, again using UML notation. As the core class **Polar Histogram** coordinates other classes to fulfill all design steps in Figure 8. A sphere wireframe with user specified bin resolution is defined to represent different ranges of velocity direction in 3D space by the **Sphere Framer** class. Based on the velocity information provided from the raw CFD data by the **Mesh Manager** class, the **Stack Accumulator Thread** class calculates the frequency of vectors pointing to each velocity direction range (cell) on the sphere wireframe. The multi-threading is applied to make this computation independent from the main working thread in order to prevent the interface from locking up. After the frequency of each cell is obtained, a stack with height associated to the frequency is defined as a cuboid with a pyramid-shaped top by the class **Stack Packer**. The **OpenGL Renderer** is responsible for general rendering of primitives such as points, lines, and polygons. Figure 10 shows our **Renderer** options displayed in the class hierarchy in which they

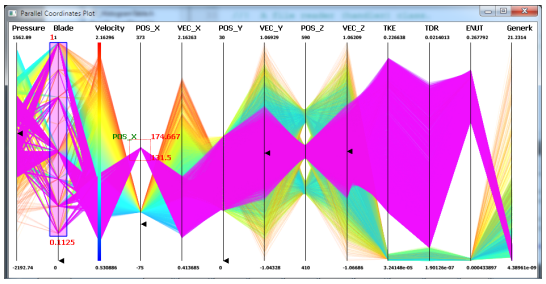

Figure 11: *A screen shot of the parallel coordinate plot visualization. The user selection is highlighted in magenta.*

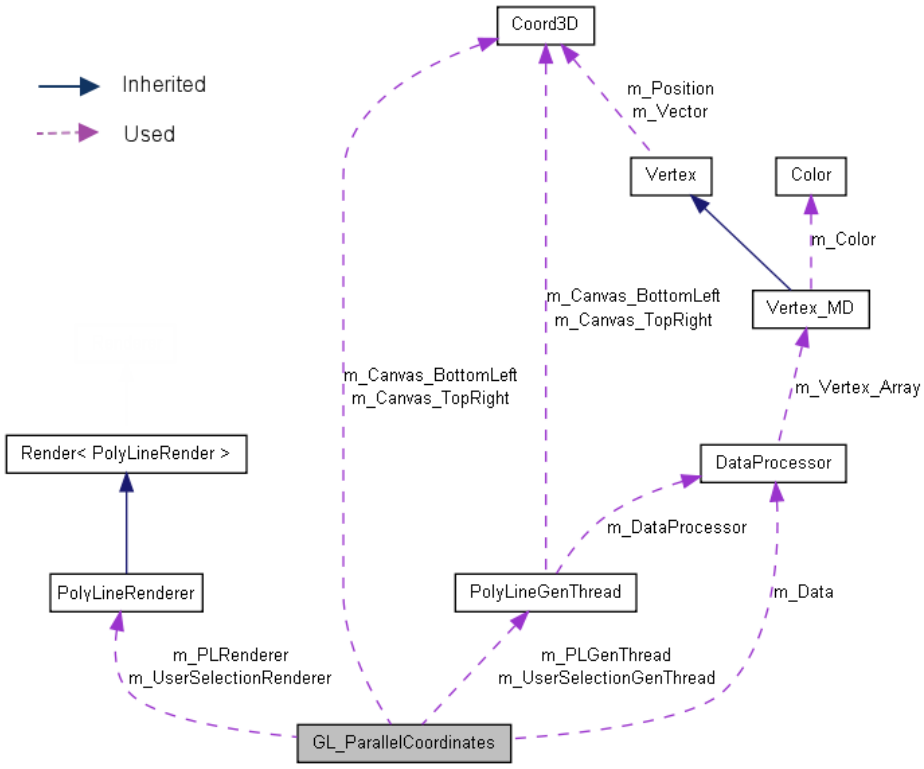

Figure 13: *A collaboration diagram generated by Doxygen for the class **GL_ParallelCoordinates** as the core class in PCP visualization.*

were designed and implemented. The hierarchy, following UML notation [Fow03], illustrates the is-kind-of relationship between rendering classes. At the top of the hierarchy the abstract base class, **Renderer**, describes attributes and behaviors that all rendering objects have, such as color, lighting, anti-aliasing etc. So there is no need to rewrite code for these in other rendering objects but simply inherit these features from this parent class. This type of design makes the project implementation more efficient and controllable. In this case, the sphere wireframe is rendered by the **Line Renderer** class and the pyramid-shaped stack is rendered by join of the **Triangle Renderer** class and the **Quad Renderer** class. In order to make the final rendering result more aesthetically pleasing, a **GLSL Shader** class is used to provide direct control of rendering effects on graphics hardware with a high degree of flexibility.

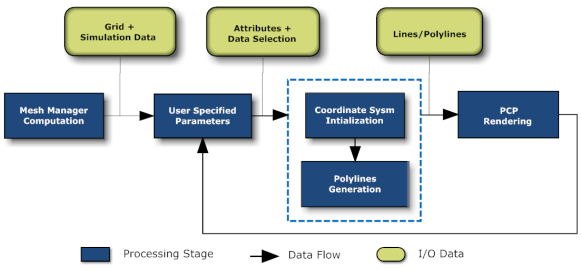

Figure 12: *The processing pipeline of the parallel coordinate plot visualization design.*

The correlation of simulation attributes of the CFD simulation dataset is deemed interesting and important by CFD engineers. Identification of these correlations and clusters from the multivariate data is the strength of Parallel Coordinate Plot (PCP) introduced by Al Inselberg [ID87]. A PCP subsystem is integrated to help the user further analyze and explore the multivariate CFD data. (Figure 11) By interacting with the PCP an intuitive pattern of attribute relationship of user selection is highlighted. Figure 12 illustrates the design pipeline of PCP subsystem. After the user brushes a portion of the data and specifies which attributes to be investigated for the initialization of PCP, polylines which reflect the correlation between each attribute are generated. Each axis represents the distribution of a specific attribute. The name of the attribute is labeled with minimum and maximum values.

In stead of using UML structure diagrams such as class diagrams in Figure 6 and 9, the implementation of PCP is illustrated using the UML behavior diagram - a collaboration diagram produced from our marine turbine project by Doxygen [vH] in Figure 13. Doxygen is a widely used documentation system for various programming languages. It can

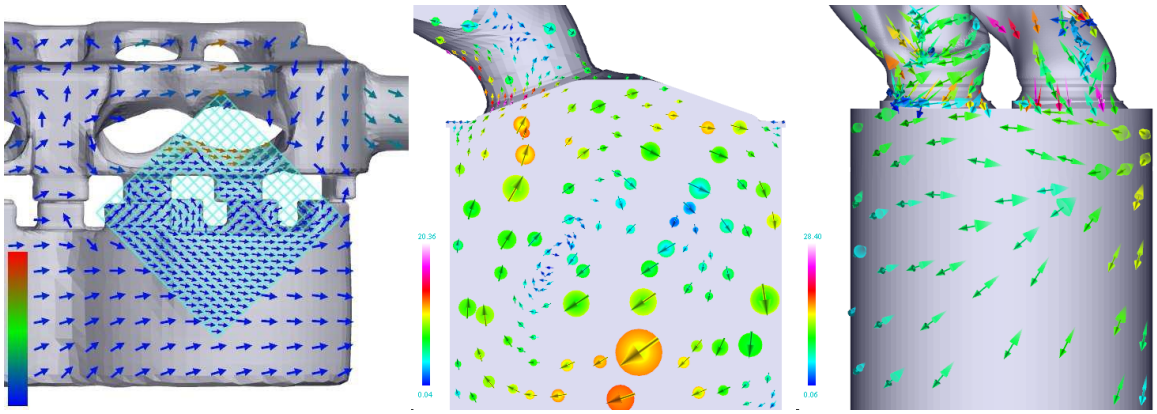

Figure 14: *Examples show direct visualization results from some our projects. (left) The glyph placement is directed by the user controlled resampling Cartesian grid on the geometry surface [PL08]. Each glyph is placed at the center of each cell to directly reveal the direction and magnitude information of the vector at that position. (middle, right) Some range glyphs are also applied along with arrow glyphs to provide statistical information of the vector field variance [PGL*11]. (middle) The ring-shaped magnitude-range glyph is used to visualize the variation in vector field magnitude within each cluster while (right) the cone-like direction-range glyph depicts the range of direction.*

produce a collaboration diagram to visualize the interactions between classes. In Figure 13 the PCP main class is presented as a filled gray box named **GL_ParallelCoordinates**. It directly employs four elements: **Coord3D**, **DataProcessor**, **PolylineGenThread**, and **PolylineRender**. The interactive collaboration between each component is detailed in following:

- **Coord3D** is a 3D coordinate class which provides methods accessing x, y, and z coordinates. Two member objects of this class, **m_Canvas_BottomLeft** and **m_Canvas_TopRight**, are used as two bounding points, bottom left and top right, to initialize the canvas region for drawing polylines by **GL_ParallelCoordinates**.
- **DataProcessor** is a data handler class that processes the raw CFD data from data files and provides all the information required by each computation and visualization component in the system. This class is the implementation of the mesh manager in Figure 12. During the data processing the multidimensional information associated with each vertex can be obtained and stored using the vertex class **Vertex_MD**. **Vertex_MD** is inherited from its parent class **Vertex** which has ability to accessing position and vector information. **Vertex_MD** also maintains other simulation attributes such as pressure, mesh resolution, etc. **Color** is used to present the color attribute of the vertex according to a given color mapping during the visualization.
- **PolylineGenThread** is mainly responsible for generating the polylines for PCP visualization. The multithreading technique is applied to accelerate the computation and make sure the polyline generation process does not freeze the main thread. In the implementation of **GL_ParallelCoordinates** two independent objects of **PolylineGenThread** are created to handle different computation requests. **m_PLGenThread** is the default working thread to generate the polylines based on the input data. **m_UserSelectionGenThread** is more specific. It focuses on the polylines selected by the user from those previously generated by **m_PLGenThread**. This implementation enables the user interactively select polylines deemed interesting for further exploration even the general polyline generation by **m_PLGenThread** is still processing.
- **PolyLineRender** is a renderer class which is inherited from **Render**. See Figure 10. This component is associated with **PolylineGenThread**. After polylines are generated and grouped by **PolylineGenThread**, **PolyLineRender** stores them into display lists for the final OpenGL rendering. **m_PLRenderer** and **m_UserSelectionRenderer** are generated for rendering different groups of polylines.

4.2. Scientific Visualization Subsystem

The scientific visualization subsystem is designed and implemented to provide the 3D spatial result of the filtered flow data from information-assisted views. Scientific techniques for flow visualization can be categorized into four groups: direct, geometric, texture-based and feature-based [PVH*03]. In our framework the direct, geometric and feature-based visualization techniques are utilized for the user to investigate the flow at different levels of abstraction. The design and implementation detail of each of these techniques is provided in following subsections.

4.2.1. Direct Flow Visualization

Direct flow visualization is the most basic and intuitive visualization category. It directly maps the visual representation

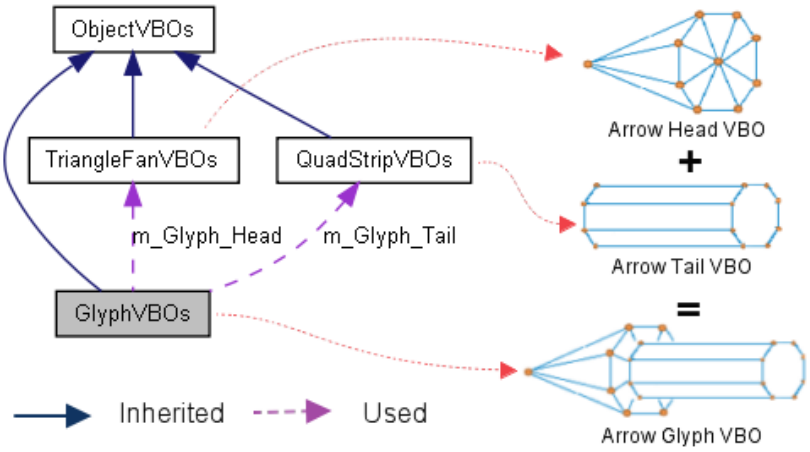

Figure 16: *(left) A collaboration diagram for the glyph VBO class. (right, top) The triangle fan VBO class is used to generate the cone-like head of the arrow glyph by tiling triangle in the fan fashion. (right, middle) The tube-like tail of the glyph is accomplished by rolling the quad strip using the quad strip VBO class. (right, bottom) by combining the previous two VBOs the final arrow glyph VBO is obtained.*

to the data samples without complex transformations or intermediate computation. Color mapping and arrow glyphs are the most representative examples of this category. Note that our design and implementation of direct flow visualization subsystem focuses on glyph-based flow visualization such as arrow glyphs rather than the color coding.

The pipeline of our direct flow visualization subsystem is shown in Figure 15. Based on the data sample the user specifies for visualization, arrow glyphs with corresponding orientation and color are generated to depict the vector field at the seeding point. Glyphs are rendered along with the original geometry to form the final result.

Direct flow visualization is fully implemented in our projects and some example results are shown in Figure 14. It has been used for 2.5D (surface-based) flow visualization [PL08] [PGL*11] and 3D flow visualization [PGSC11].

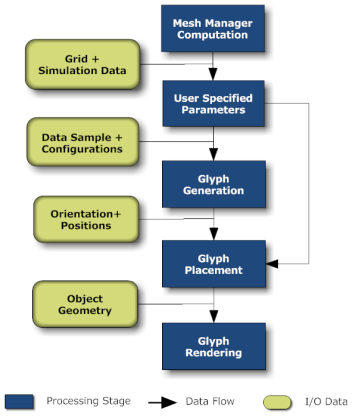

Figure 15: *The processing pipeline for the direct flow visualization subsystem.*

It is worth pointing out that although glyph flow visualization attempts to present intuitive visualization of the vector field of the sample data quickly, the visualization can still suffer from performance issues if glyphs are naively implemented for the large dataset visualization. In order to maintain the high performance of the glyph implementation, the new OpenGL extension - Vertex Buffer Object (VBO) [Ope] is used. VBO allows vertex array data to be stored in high-performance graphics memory rather than the system memory and promotes efficient data transfer, which enables substantial performance gains and more flexibility for the dynamic glyph object implementation. We don't go through the detail of VBO here. For more information the official white paper [Ope] is recommended.

In Figure 16 the collaboration diagram shows how each component VBO class is used to form the arrow glyph VBO. **GlyphVBOs**, **TriangleFanVBOs**, and **QuadStripVBOs** are classes inherited from **ObjectVBOs** class which provides the basic VBO operations such as providing attribute arrays for creating VBOs, resetting attribute arrays, etc. Based on the parent class **ObjectVBOs** the **TriangleFanVBOs** class creates VBOs in the fashion of the triangle fan while the **QuadStripVBOs** class uses quad strips. The arrow head of **GlyphVBOs** is described by the **TriangleFanVBOs** class with a cone tiled by a triangle fan which consists of 10 vertices and 16 triangles by default. See Figure 16. And the tube shaped tail is represented by the **QuadStripVBOs** with a default 8 adjacent quads, shown in Figure 16. Note that the resolution of **TriangleFanVBOs** and **QuadStripVBOs** can be customized by the user even the default values are provided. VBOs are also used with other visualization subsystems such as the following geometric flow visualization.

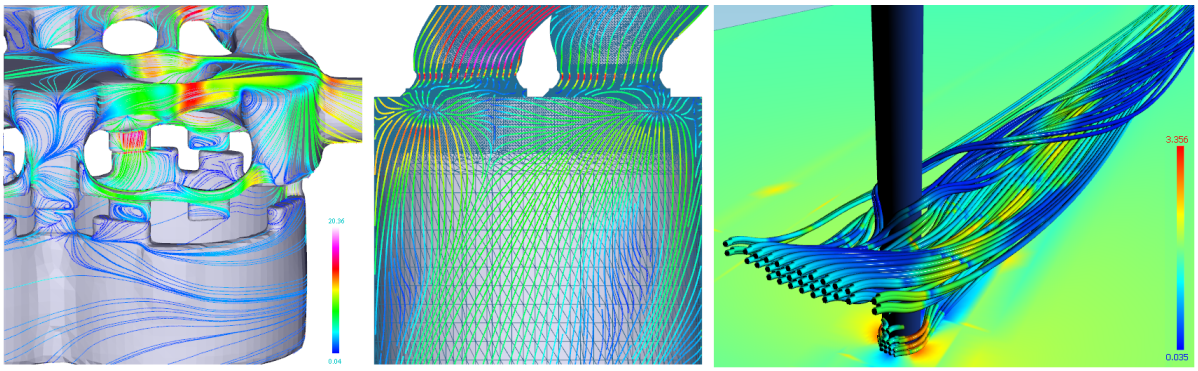

Figure 17: *Results of Geometric flow visualization implementation in our projects. (left and middle) Color coded streamlines are used to depict the underlying boundary flow characteristics. (middle) The evenly-spaced streamline technique [SLCZ09] [JL97] can be applied regardless of the associated mesh resolution. (right) Shaded streamlines deliver more depth percept for visualization.*

4.2.2. Geometric Flow Visualization

Visualization of this type generates continuous geometric objects which reflects the underlying vector field. During the computation the missing velocity field between original sparse samples can be reconstructed and applied to the geometric objects by using interpolation and integration. This visualization can provide a coherent visual result of underlying vector field. Some implementation results are shown in Figure 17. In what follows, we discuss the design and implementation of our geometric flow visualization subsystem. Note that it focuses on the geometric objects using integration such as streamlines and tubes rather than other geometric objects such as isosurfaces, since the complete coverage of the geometric techniques is beyond this work. See the survey by McLoughlin et al. [MLP*10] for more research results in this area.

Figure 18 illustrates the main processing pipeline for the geometric flow visualization subsystem. The simulation data is the input for the process. After the user specifies the data sample and the configuration attributes such as seeding requirements, the initial seeding positions are generated. Based on these seeds, the streamline is traced by the numerical integration and interpolation. The construction of the streamline is iterative and interactive: an interactive streamline adjustment step is involved at each iteration of the construction to make sure the updated user requirements can be met as soon as possible such as changing the distance between each streamline in evenly spaced streamline visualization [JL97] [SLCZ09]. After the object geometries are ready, the last step is rendering. Feedback from the rendering result helps the user to refine the configurations. The main process of the implementation step is demonstrated by the collaboration diagrams of class **StreamGenerator** and class **StreamTubeRenderThread** in Figure 19.

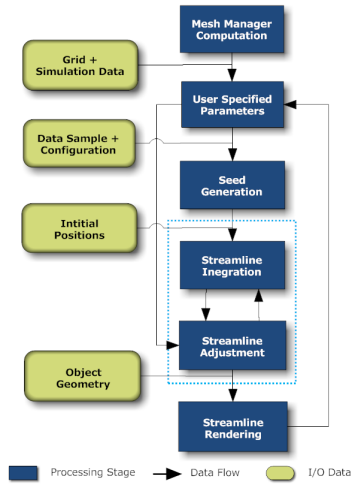

Figure 18: *The processing pipeline for the geometric flow visualization subsystem.*

The **StreamGenerator** class is designed to trace and validate all the stream seeding points and then output each streamline as an array of points. This class provides methods needed by the streamline integration and streamline adjustment stages in the design process. In Figure 19 (left) as the component classes **DataProcessor** and **Vertex_MD** have been explained in the PCP section, the functions provided here are very similar, so we focus on the other two classes which are the core components - **Integrator** and **Interpolator**.

- **Integrator**: This class is designed to provide static methods for various numerical integrations such as the first-order Euler integration and the second-order Runge-Kutta integration [PTVF07]. In the subsystem the streamlines are traced using the Euler integration by default with the user specified step sizes. After each streamline is generated by integration, a validation method is provide by **StreamGenerator** to check if the generated point is within the bounding box or not. If it is in the bounding box the iterative streamline tracing proceeds otherwise the integration is ended and the resulting integral paths are

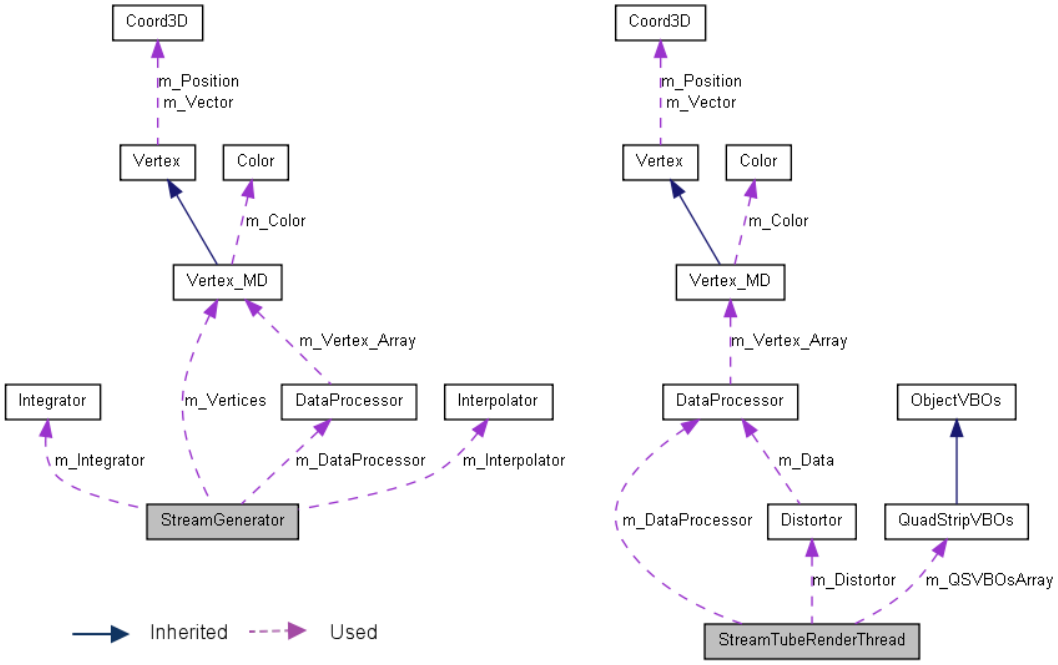

Figure 19: *Collaboration diagrams for the class* ***StreamGenerator*** *and class* ***StreamTubeRenderThread*** *in geometric flow visualization subsystem. Diagrams are generated by Doxygen.*

stored as arrays and handed over to the streamline renderer.

- **Interpolator**: This class provides various interpolation schemes in a quad or cube grid cell such as the velocity interpolation, color interpolation, mesh resolution interpolation of a point(1D), a line(2D), or a plane(3D). After each valid streamline point is obtained from the integration, the **Interpolator** class is used to interpolate its attribute values like velocity, pressure, etc. based on vertices of the cell which contains this point. By using the interpolation, vector field information needed for streamline integration can be reconstructed from the original discrete data samples.

Once streamlines are generated they are ready for rendering. In our subsystem we have two streamline rendering mechanisms: the color mapped 2D and 3D streamline rendering and the shaded 3D rendering. The former is simpler and faster but tends to suffer from visual clutter while the later is a bit more complex and slower but delivers the better depth perception of streamlines. Here we look at the shaded streamline rendering class **StreamTubeRenderThread**.

The **StreamTubeRenderThread** class is mainly responsible for extending the streamlines generated from **StreamGenerator** to shaded 3D tube-shaped streamlines for rendering. This class employs a multi-threading technique since the tube construction and rendering can be expensive enough to block the main thread. The collaboration diagram is shown in Figure 19 (right). To construct the shaded streamlines, **QuadStripVBOs** is used to generate a tube segment with default 8 adjacent quads, like the arrow tail VBO in Figure 16 (right, middle), around each streamline segment. See Figure 20. The resolution of the quads can be updated by the user. This affects the smoothness of the streamline and the tube construction performance as well. The more quads

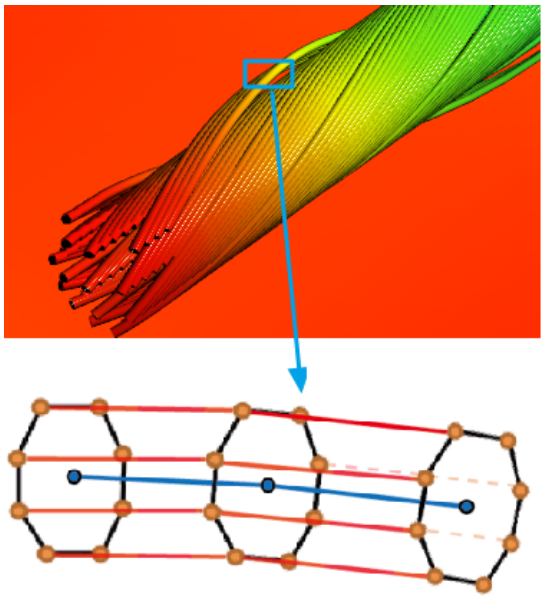

Figure 20: *This figure demonstrates how the shaded streamline is generated by a collection of quad strips.*

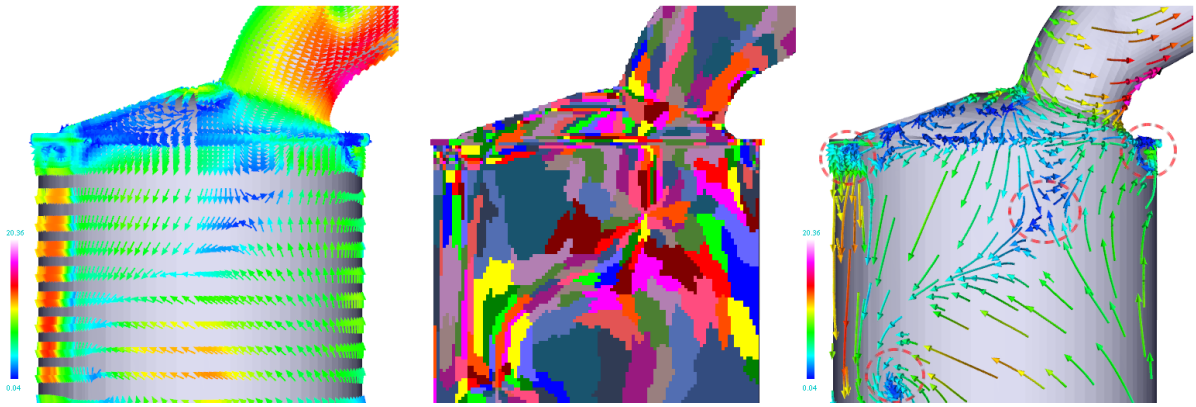

Figure 21: *A screen shot of the visual result comparison between the direct flow visualization (left) and the feature-based visualization (right). (left) The cluttered direct flow visualization can't do a good job of extracting important flow features. (middle) Vector field clusters are generated and colored in different colors. (right) Based on the generated clusters, shaded streamlets with arrow head are applied to reveal some flow features such as vortices and saddle points which are highlighted by red circles.*

used the smoother the tube geometry but the slower the tube construction. The **Distortor** class is used to transfer the position of seeding points according to the mesh resolution so that more visual detail in regions of interest can be provided by the distorted streamlines without losing visualization coherency [PGSC11].

Note that the streamline generation step and the streamline rendering step are separate. The advantage to this type of design and implementation is once streamlines are generated they can be reused again and again for different rendering options if there is no new seeding requirement from the user.

4.2.3. Feature-based Flow Visualization

Rather than directly showing the underlying vector field, feature-based flow visualization is employed to provide a 'filtered' visual result by extracting or highlighting the meaningful patterns such as vortices and saddles from the simulation data sets. Those extracted parts deemed interesting by users are defined as the *feature*. For more literature we refer the interested reader to in-depth surveys of feature-based flow visualization by Post et al. [PVH*02] and Laramee et al. [LHZP07]. In our framework we applied an image-based vector field clustering approach (IBVFC) presented by Peng et al. [PGL*11] as a feature extractor and highlighter for the flow visualization. By doing a quick and automatic hierarchical clustering of the vector field according to the user specified errors, this feature-based visualization approach is able to simplify the result with dense visualization on regions with the most suggestive and important information and remain sparse on the less important ones. An example result is shown in Figure 21. Note that in this section we only focus on this approach closely rather than other feature-based visualization techniques.

Figure 22 demonstrates the design pipeline of our feature-based flow visualization subsystem. In brief, our feature-based flow visualization subsystem simplifies the problem of clustering vector fields on surfaces by confining the clustering process to image space. After the projection to image space, clusters are only generated for visible regions of vector fields on the surfaces. Then various visualization approaches are applied to reflect the vector fields based on those clusters. The main procedures of our IBVFC visualization subsystem are shown on the right part of Figure 22: (1) project the vector field to the image plane so that attribute images are obtained, (2) detect geometric edge discontinuities based on the depth image, (3) process a bottom-up hierarchical clustering based on attribute images using a distance-metric, (4) overlay specified visual representations of the original surface geometry such as a semi-transparent representation of the surface with shading, along with (5) the image overlay glyph or/and streamline visualization which

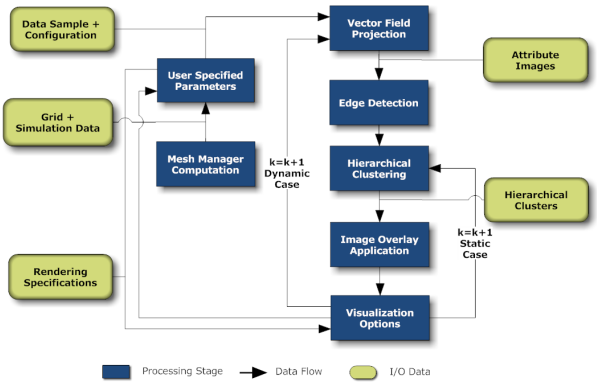

Figure 22: *The processing pipeline for the feature-based flow visualization subsystem.*

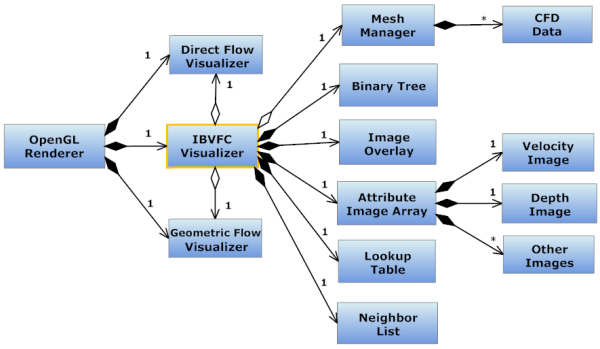

Figure 23: *The relationship among the major components of the feature based visualization implementation. UML notation is used.*

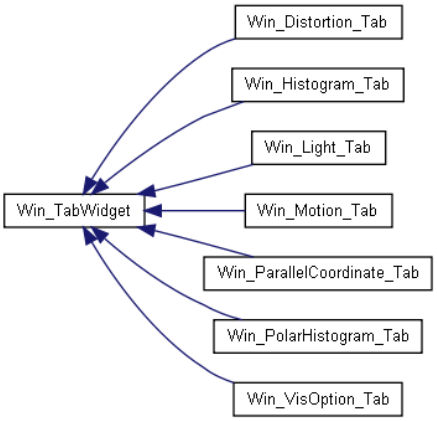

Figure 24: *A Doxygen inheritance diagram for the console menu tab window.*

are applied automatically based on the user-defined error metric and rendering specifications. Additionally, various enhancements and user options, like distance-metric weighting coefficients, can be used to customize the clustering process and thus the final visualization result. It's also worth mentioning that if the viewpoint is changed, such as cases of geometry rotation, translation, and scaling, steps 1-5 of the pipeline are necessary for the next pass, and only a subset (steps 4-7) of the algorithm is required for the static cases if the clustering distance-metric parameters are changed without changing the view-point. Each stage is described in more detail in previous research [PGL*11].

Figure 23 outlines the class relationship between the major components of the IBVFC visualization subsystem implementation using UML composition notation. The **IBVFC Visualizer** class is the core class which coordinates each component in Figure 22. The **Mesh Manager** class is responsible for handling CFD data sets and providing access to the data information. The **Attribute Image Array** class is used to store and manage the projected attribute images which simplifies the computation from 3D to 2D. This array is the data source for the IBVFC. A **Velocity Image** holds all the projected vector field information which is used to produce the clusters. The **Depth Image** stores the depth map of the projected visible portion. It is used for edge detection. Other image classes such as the **Mesh Resolution Image** in [PGL*11] can be used to add extra parameters associated with the error metric which drives the clustering process. The **Binary Tree** class is a storage class which is used to implement the hierarchal cluster structure. Each node of the binary tree represents a cluster whose error metric reflects its vector field characteristics. The storage classes, the **Lookup Table** and the **Neighbor List**, are used to accelerate the cluster searching and merging process. The **Image Overlay** class stores perceptual information such as the shading and the depth map for the final rendering. It is worthy of note that the **IBVFC Visualizer** has a "is-part-of" relationship with the **Direct Flow Visualizer** and the**Geometric Flow Visualizer** in Figure 23. This means various visualizations are plug-ins for IBVFC to reveal vector field clusters with different presentations. This design and implementation makes the system easier to extend and maintain. If there are new visualization approaches available for IBVFC we can simply plug them in. At last **OpenGl Renderer** wraps up all the rendering processes.

4.3. User Interface Design and Implementation

Design and implementation of a friendly and effective user interface (UI) is an essential part of our flow visualization framework. How to design a good UI is a classic topic in the field of human-computer interaction (HCI) community. A literature survey dedicated to the subject is beyond the scope of this work and we refer the interested readers to some influential work by Shneiderman [Shn92], Torres [Tor02], and Cooper and Reimann [CR03]. In the following content we discuss the design and implementation of important components of our framework UI: the graphical user interface (GUI), the event handler, and the multi-threading platform. Note that we use QT library [Nok] to implement each above UI component since the use of QT ensures the platform independence of the user interface. So related QT classes are also mentioned in the following subsections.

The Outer Body - Graphical User Interface (GUI) The GUI serves at the frontline of the user interaction. In our multi-linked flow visualization framework shown in Figure 3, each subwindow works as a GUI. Firstly, the tabbed console menu window (boxed in sky blue in Figure 3) is used. The tab widget not only groups the interactions for the same or similar task but also saves screen space by compacting tabs in the same window. Each tab widget of the console menu is inherited from the class **Win_TabWidget** which provides basic QT graphical control elements including check-boxes, sliders, drop-down lists, spinners and buttons. See Figure 24. The user can specify parameters to select or update analysis or visualization options by interacting

with control elements in the corresponding tab widget. For example if we want to turn on the slice function along x axis and specify the slicing position, we go to the 'Motion Options' tab then tick the check-box 'X-Slice' and update the position value by scrolling the spinner next to the check-box. Additionally, each visualization window is a GUI which supports mouse-event driven user interactions. Information visualizations such as histogram table and PCP allow the user to mouse brush or select the data deemed interesting while the scientific visualization window provides the hands-on interaction for the rendering result such as translation, rotation, and zooming.

The Brain - Event Handler After the user input obtained from GUI, the class **ControlPanel** is applied as an asynchronous callback center which handles all the corresponding events from GUI such as key presses, mouse movement, action selections, and timers and then triggers the related functions or computations. In order to implement this functionality to the **ControlPanel** class, the Signal-Slot callback mechanism from QT is used. The concept of Signal-Slot is that objects can send particular signals containing event information which can be received by other objects using special functions known as slots. The Signal-Slot mechanism is contained in all classes which are inherited from the class **QObject**. This means the mechanism can be easily accessed and used in almost every class. One more advantage about the Signal-Slot mechanism is that we can connect as many signals as we want to a single slot, and a signal can be connected to as many slots as we need. See Figure 25. This flexibility enables the **ControlPanel** coordinating the communication between multi-linked views more straightforward and efficient.

The Nervous System - Multi-threading Platform Since the size of the simulation datasets tends to be non-trivial, maintaining the smooth user interaction among the GUI and rendering a large number of selected objects simultaneously in visualization views is a challenge. In order to address this and deliver a smooth and efficient user interaction, a multi-threading platform inspired by Piringer et al. [PTMB09] is used. **QThread** class provided by QT is inherited by each visualization class so that each visualization view has its own working thread which is independent from the main working thread. Based on this, users can interact with the visualization result even as it is still being generated in the background.

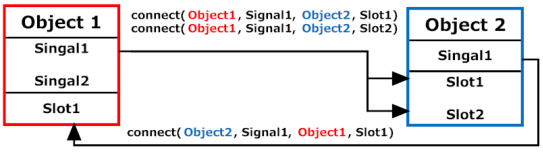

Figure 25: *A diagram shows how the QT signal-slot callback mechanism works.*

5. Discussion and Evaluation

After presenting the design and implementation of our flow visualization framework, we now discuss the advantages and disadvantages of implementing each subsystem into the framework and evaluate the framework against the benefits specified in Section 1.

Support for Versatile, Real-world, High-dimensional CFD Data sets Our framework implements a mesh manager which handles a large, varied collection of real-world CFD data sets: from small geometries to large geometries, from surface-based flow to volume-based flow, from vector field information to multidimensional data, from the automotive simulation to the marine turbine simulation data. The mesh manager is able to convert the data from an ASCII file to a binary file to accelerate the loading process of a large data set. However, the mesh manager has its disadvantages: it has not been separated from the main working thread. So when a large data set is being loaded the user interaction is interrupted until the data loading is finished.

Support for a Wide Range of Visualization Techniques A big advantage of implementing different visualization subsystems into the framework is the ability to provide various visualization options. For information visualization we implement the histogram table, the polar histogram and the parallel coordinate plot while the direct, geometric, and feature-based flow visualization techniques are employed for scientific visualization use. It is helpful to provide CFD engineers with a wide range of options since each visualization technique has its own advantages and disadvantages. The usability of this framework is approved by CFD engineers based on their domain expert reviews [PGL*11] [PGSC11]. It's also worthy of note that it is unusual to have this many visualization options integrated to a research prototype. Naturally there are also disadvantages to integrating various visualization subsystem into a framework. Firstly, our all-in-one framework involves a lot of classes, although an object-oriented design pattern is adopted to reduce the complexity and the time spent on learning and understanding the process pipeline. Secondly, large numbers of data transfered simultaneously between subsystems may freeze the user interaction. So the way to combine various visualizations needs to be more efficient, stable and robust.

Interactivity Based on our multi-threading platform, each visualization subsystem is able to deliver smooth and efficient user interactions even as the visualization result is still being generated in the background. However, multi-threading adds more complexity to the coding and debugging stages.

Platform Independence Platform independence is achieved through the use of the platform independent libraries - QT and OpenGL. QT is an open source cross-platform application framework which not only provides the platform independent GUI library but also supports OpenGL which a widely supported, platform independent graphics

library. The downside of QT in this implementation is all OpenGL related. Firstly, the QT OpenGL API is a bit different from the original OpenGL API, which may involve some learning curve. Secondly, the QT OpenGL library could conflict with other OpenGL libraries.

6. Conclusion

We describe the design and implementation of a generic framework which incorporates information and scientific visualization approaches to provide effective visual analysis of CFD flow simulation data. In contrast to most research prototypes, the system we present handles large, real-world, high-dimensional data. We discuss the design and implementation of visualization subsystems of the frame work and the user interface. We also evaluate the implementation of this framework by discussing the advantages and disadvantages against the goal of our framework. At last, we believe the principles outlines in this work can be applied to a more general way to other similar projects.

7. Acknowledgments

The authors wish to thank A, B, C. This work was supported in part by a grant from EPSRC.

References

[CR03] COOPER A., REIMANN R. M.: *About Face 2.0: The Essentials of User Interface Design*. John Wiley & Sons, 2003. 12

[DGH03] DOLEISCH H., GASSER M., HAUSER H.: Interactive Feature Specification for Focus+Context Visualization of Complex Simulation Data. In *Data Visualization, Proceedings of the 5th Joint IEEE TCVG–EUROGRAPHICS Symposium on Visualization (VisSym 2003)* (May 2003), pp. 239–248. 2

[Dol07] DOLEISCH H.: Simvis: Interactive Visual Analysis of Large and Time-Dependent 3D Simulation Data. In *WSC '07: Proceedings of the 39th Conference on Winter Simulation* (Piscataway, NJ, USA, 2007), IEEE Press, pp. 712–720. 2

[FDFR10] FISHER D., DRUCKER S., FERNANDEZ R., RUBLE S.: Visualizations Everywhere: A Multiplatform Infrastructure for Linked Visualizations. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1157 –1163. 3

[Fow03] FOWLER M.: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, third ed. Object Technology Series. Addison-Wesley, Sept. 2003. 3, 4, 6

[GJL*09] GRUNDY E., JONES M. W., LARAMEE R. S., WILSON R. P., SHEPARD E. F.: Visualization of sensor data from animal movement. *Eurographics/ IEEE-VGTC Symposium on Visualization (Eurovis) 2009, Computer Graphics Forum 28*, 2 (June 2009), 815–822. 5

[ID87] INSELBERG A., DIMSDALE B.: Parallel Coordinates for Visualizing Multi-Dimensional Geometry. In *Proceedings of Computer Graphics International (CGI '87))* (1987), pp. 25–44. 6

[JL97] JOBARD B., LEFER W.: Creating Evenly–Spaced Streamlines of Arbitrary Density. In *Proceedings of the Eurographics Workshop on Visualization in Scientific Computing '97* (1997), vol. 7, pp. 45–55. 9

[Lar10] LARAMEE R. S.: Bob's Concise Coding Conventions (C^3). In *Advances in Computer Science and Engineering (ACSE)* (February 2010), vol. 4, pp. 23–26. 2

[LEG*08] LARAMEE R. S., ERLEBACHER G., GARTH C., THEISEL H., TRICOCHE X., WEINKAUF T., WEISKOPF D.: Applications of Texture-Based Flow Visualization. *Engineering Applications of Computational Fluid Mechanics (EACFM) 2*, 3 (Sept. 2008), 264–274. 1

[LHH05] LARAMEE R. S., HADWIGER M., HAUSER H.: Design and Implementation of Geometric and Texture-Based Flow Visualization Techniques. In *Proceedings of the 21st Spring Conference on Computer Graphics* (May 2005), pp. 67–74. 2

[LHZP07] LARAMEE R., HAUSER H., ZHAO L., POST F. H.: Topology-Based Flow Visualization: The State of the Art. In *Topology-Based Methods in Visualization (Proceedings of Topo-in-Vis 2005)* (2007), Mathematics and Visualization, Springer, pp. 1–19. 11

[MLP*10] MCLOUGLIN T., LARAMEE R. S., PEIKERT R., POST F. H., CHEN M.: Over Two Decades of Geometric Flow Visualization. *Computer Graphics Forum 29*, 6 (2010), 1807–1829. 9

[Nok] NOKIA: QT. *http://qt.nokia.com/products/library (last accessed 25/05/2011).* 12

[Ope] OPEGNGL.ORG: Vertex Buffer Object Whitepaper. *http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt (last accessed 02/06/2011).* 8

[PBK10] PIRINGER H., BERGER W., KRASSER J.: HyperMoVal: Interactive Visual Validation of Regression Models for Real-Time Simulation. *Computer Graphics Forum 29*, 3 (2010), 983–992. 2

[PGL*11] PENG Z., GRUNDY E., LARAMEE R. S., CHEN G., CROFT N.: Mesh-Driven Vector Field Clustering and Visualization: An Image-Based Approach. *IEEE Transactions on Visualization and Computer Graphics (IEEE TVCG)* (2011), forthcoming. 7, 8, 11, 12, 13

[PGSC11] PENG Z., GENG Z., S.LARAMEE R., CROFT N.: *Visualization of Flow Past a Marine Turbine: The Search for Sustainable Energy*. Tech. rep., Department of Computer Science, Swansea University, UK, Dec 2011. 3, 4, 8, 11, 13

[PL08] PENG Z., LARAMEE R.: Vector Glyphs for Surfaces: A Fast and Simple Glyph Placement Algorithm for Adaptive Resolution Meshes. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2008* (Constance, Germany, 8-10 October 2008), pp. 61–70. 7, 8

[PL09] PENG Z., LARAMEE R.: Higher Dimensional Vector Field Visualization: A Survey. In *Proceedings of Theory and Practice of Computer Graphics (TPCG '09)* (Cardiff, UK, 17-19 June 2009), pp. 61–70. 1

[PTMB09] PIRINGER H., TOMINSKI C., MUIGG P., BERGER W.: A Multi-Threading Architecture to Support Interactive Visual Exploration. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1113–1120. 2, 13

[PTVF07] PRESS W. H., TEUKOLSKY S. A., VETTERLING W. T., FLANNERY B. P.: *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3 ed. Cambridge University Press, New York, NY, USA, 2007. 9

[PVH*02] POST F. H., VROLIJK B., HAUSER H., LARAMEE R. S., DOLEISCH H.: Feature Extraction and Visualization of Flow Fields. In *Eurographics 2002 State-of-the-Art Reports* (2–6 September 2002), pp. 69–100. 11

[PVH*03] POST F. H., VROLIJK B., HAUSER H., LARAMEE R. S., DOLEISCH H.: The State of the Art in Flow Visualization:

Feature Extraction and Tracking. *Computer Graphics Forum 22*, 4 (Dec. 2003), 775–792. 7

[Shn92] SHNEIDERMAN B.: *Designing the User Interface: Strategies for Effective Computer Interaction*, 2nd ed. Addison-Wesley, 1992. 12

[SLCZ09] SPENCER B., LARAMEE R., CHEN G., ZHANG E.: Evenly-Spaced Streamlines for Surfaces. *Computer Graphics Forum* (2009). forthcoming. 9

[Tor02] TORRES R. J.: *Practitioners Handbook for User Interface Design and Development*. Prentice Hall, 2002. 12

[vH] VAN HEESCH D.: Doxygen. *http://www.stack.nl/ dimitri/-doxygen/index.html (last accessed 25/05/2011).* 2, 3, 6

[WBWK00] WANG BALDONADO M. Q., WOODRUFF A., KUCHINSKY A.: Guidelines for Using Multiple Views in Information Visualization. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces* (New York, NY, USA, 2000), ACM, pp. 110–119. 2

[Wea04] WEAVER C.: Building Highly-Coordinated Visualizations in Improvise. In *Proceedings of the IEEE Symposium on Information Visualization* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 159–166. 2

[Wea09] WEAVER C.: Cross-Filtered Views for Multidimensional Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics 16*, 2 (2009), 192–204. 2