Swansea University
Prifysgol Abertawe

Cronfa
Setting Research Free

Cronfa - Swansea University Open Access Repository

This is an author produced version of a paper published in :
Design and Implementation of Interactive Flow Visualization Techniques

Cronfa URL for this paper:
http://cronfa.swan.ac.uk/Record/cronfa24673

Book chapter :
McLoughlin, T. & Laramee, R. (2012). *Design and Implementation of Interactive Flow Visualization Techniques.* Design and Implementation of Interactive Flow Visualization Techniques, InTech.

http://dx.doi.org/10.5772/35943

This article is brought to you by Swansea University. Any person downloading material is agreeing to abide by the terms of the repository licence. Authors are personally responsible for adhering to publisher restrictions or conditions. When uploading content they are required to comply with their publisher agreement and the SHERPA RoMEO database to judge whether or not it is copyright safe to add this version of the paper to this repository.
http://www.swansea.ac.uk/iss/researchsupport/cronfa-support/

6

Design and Implementation of Interactive Flow Visualization Techniques

Tony McLoughlin and Robert S. Laramee
Swansea University
United Kingdom

1. Introduction

The demand for flow visualization software stems from the popular (and growing) use of Computational Fluid Dynamics (CFD) and the increasing complexity of simulation data. CFD is popular with manufacturers as it reduces cost and the time of production relative to the expense involved in creating a real physical model. Modifications to a physical model to test new prototypes may be non-trivial and expensive. CFD solvers enable a high degree of software-based testing and refinement before creating a real physical model.

The visualization of CFD data presents many different challenges. There is no single technique that is appropriate for the visualization of all CFD data. Some techniques are only suitable for certain scenarios and sometimes an engineer is only interesting in a sub-set of the data or specific features, such as vortices or separation surfaces. This means that an effective flow visualization application must offer a wide range of techniques to accommodate these requirements. The integration of a wide variety of techniques is non-trivial and care must be taken with the design and implementation of the software.

We describe our flow visualization software framework that offers a rich set of state-of-the-art features. It is the product of over three years of development. The paper provides more details about the design and implementation of the system than are normally provided by typical research papers due to page limit constraints. Our application also serves as a basis for the implementation and evaluation of new algorithms. The application is easily extendable and provides a clean interface for the addition of new modules. More developers can utilize the code base in the future. A group development project greatly varies from an individual effort. To make this viable, strict coding standards [Laramee (2010)] and documentation are maintained. This will help to minimize the effort a future developer needs to invest to understand the codebase and expand upon it.

Throughout this chapter we focus on the design and implementation of our system for flow visualization. We address how the systems design is used to address the challenges of visualization of CFD simulation data. We describe several key aspects of our design as well as the contributing factors that lead to these particular design decisions.

The rest of this chapter is organized as follows: Section 2 introduces the reader to the field of flow visualization and provides information for further reading. Section 3 describes the user requirements and goals for our application. Section 4 provides an overview of the application design. A description of the major systems is then provided with the key classes and relationships are discussed. The chapter is concluded in Section 5. Throughout the

chapter, class hierarchies and collaboration graphs are provided for various important classes of the system.

2. Related work and background

The visualization of velocity fields presents many challenges. Not the least of which is the notion of how to provide an intuitive representation of a 3D vector projected on to a 2D image plane. Other challenges include:

- Occlusion in volumetric flow fields
- Visualizing time-dependent flow data
- Large, High-dimensional datasets – it is common place to see datasets on the Giga- and Tera-byte scale.
- Uncertainty. Due to the numerical nature of CFD and flow visualization, error is accumulated at every stage. This needs to be minimized in order to provide accurate visualization results.

This is by no means an exhaustive list but serves as a representation to give the reader a feel for the context of the system. Flow visualization algorithms can be classified into 4 sub-groups: direct, texture-based, geometric and feature-based. We now provide a description of each of these classes and highlight some of key techniques in each one.

Direct flow visualization

This category represents the most basic of visualization techniques. This range of techniques maps visualization primitives directly to the samples of the data. Examples of direct techniques are color-mapping of velocity magnitude or rendering arrow glyphs [Peng & Laramee (2009)].

Texture-based flow visualization

This category provides a dense representation of the underlying velocity field, providing full domain coverage. This range of techniques depicts the direction of the velocity field by filtering a (noise) texture according the local velocity information. This results in the texture being smeared along the direction of the velocity. Line Integral Convolution (LIC) by Cabral and Leedom [Cabral & Leedom (1993)] one seminal texture-based technique. Other texture-based variants include Image-space advection (ISA) by Laramee et al. [Laramee et al. (2003)] and IBFVS [van Wijk (2003)], both of which use image-based approaches to apply texture-based techniques to velocity fields on the surfaces of CFD meshes. It should be noted that due to the dense representation of the velocity field, texture-based techniques are more suited to 2D flow fields and flow fields restricted to surfaces. Three-dimensional variants do exist [Weiskopf et al. (2001)] but occlusion becomes a serious problem and reduces effectiveness. We refer the interested reader to [Laramee et al. (2004)] for a thorough overview of texture-based techniques.

Geometric flow visualization techniques

This category involves the computation of geometry that reflects the properties of the underlying velocity field. The geometry used is generally curves, surfaces and volumes. The geometric primitives are constructed using numeric integration and using interpolation to reconstruct the velocity field between samples.

Typically the geometry remains tangent to the velocity as in the case of streamlines and streamsurfaces [Hultquist (1992)] [McLoughlin et al. (2009)]. However non-tangential geometry also illustrate important features; streaklines and streaksurfaces [Krishnan et al. (2009)] are becoming increasingly popular. In fact this application framework was involved in the development of a novel streak surface algorithm [McLoughlin, Laramee & Zhang (2010)]. A thorough review of geometric techniques is beyond the scope of this paper and we refer the interested reader to a survey on the topic by McLoughlin et al. [McLoughlin, Laramee, Peikert, Post & Chen (2010)].

Feature-based flow visualization

Feature-based techniques are employed to present a simplified sub-set of the velocity field rather than visualizing it in its entirety. Feature-based techniques generally focus on extracting and/or tracking characteristics such as vortices, or representing a vector field using a minimal amount of information using topological extraction as introduced by Helmann and Hesselink [Helman & Hesselink (1989)]. Once again a thorough review of this literature is beyond the scope of the presented paper and we refer the interested to in-depth surveys on

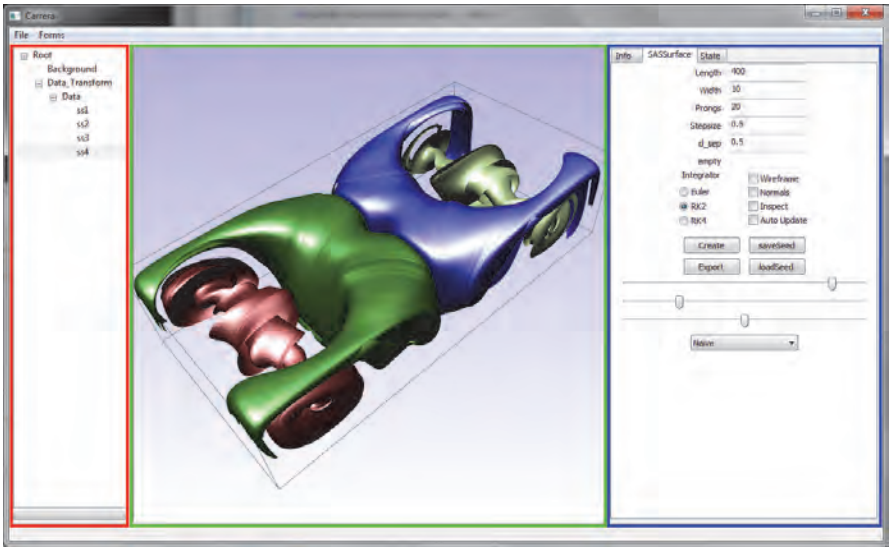

Fig. 1. A screenshot of the application showing the GUI providing controls for streamsurfaces computed on a simulation of Rayleigh-Bénard convection. The application window is split up into three distinct regions. 1. The **Application Tree** (highlighted by the red box) is used to manage the assets in the scene. 2. The **Rendering Window** (highlighted by the green box) displays the visualization results and allows the user to interactively modify the viewing position and orientation. 3. The **Asset Control Pane** (highlighted by the blue box) displays the current set of controls for the selected asset. The GUI is context sensitive for the benefit of the user and the asset control pane only displays the controls for a single tool at any given time. Should a different visualization tool be selected a new set of controls are displayed and the unrequired ones removed. This approach is adopted to provide a simple, uncluttered interface, allowing the user to focus only on the necessary parameters/controls.

feature-based flow visualization by Post et al. and Laramee et al. [Laramee et al. (2007); Post et al. (2003)].

3. System requirements and goals

Our application framework is used to implement existing advanced flow visualization techniques as well as being a platform for the development and testing of new algorithms. The framework is designed to be re-used by future developers researching flow visualization algorithms to increase efficiency and research output. Figure 1 shows a screenshot of the application in action.

Support for a wide-variety of visualization methods and tools

Our application is designed as a research platform. A variety of visualization methods have been implemented so that new algorithms can be directly compared with them. Therefore, the system is designed to be easily extensible. Some of the key flow visualization methods and features that are integrated into our application framework include the following:

1. Integral Curves (with illumination) [Mallo et al. (2005)]
2. Stream- and Pathsurfaces [McLoughlin et al. (2009)]
3. Isosurfaces [Lorensen & Cline (1987)]
4. Streaksurfaces [McLoughlin, Laramee & Zhang (2010)]
5. Slice probes
6. Line Integral Convolution (LIC) [Cabral & Leedom (1993)]
7. Critical Point Extraction
8. Parameter Sensitivity Visualization [McLoughlin, Edmunds, Laramee, Chen, Max, Yeh & Zhang (2011)]
9. Clustering of integral curves [McLoughlin, Jones & Laramee (2011)]
10. Vector field resampling
11. Image output to multiple formats
12. Integral curve similarity measures [McLoughlin, Jones & Laramee (2011)]
13. Computation of the Finite Time Lyapunov Exponent (FTLE) [Haller (2001)]

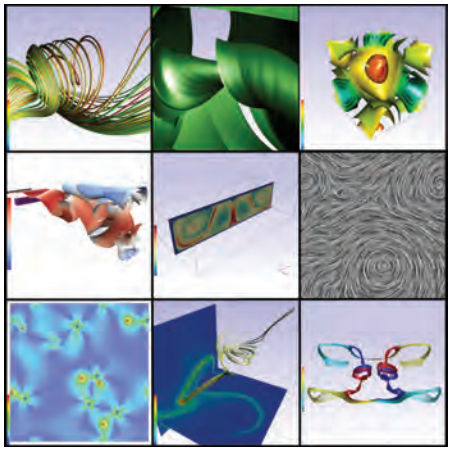

Fig. 2. Traversing left-to-right from top-to-bottom, examples of visualization techniques 1-9.

Interactivity

Users generally require flexibility over the final visualization and favor feedback as quickly as possible after modifying visualization parameters. Our system is designed to enable a high level of interaction for the user. Providing such a level of interaction allows for easier exploration of the data. The user can also tailor the resulting visualization to their specific needs. This level of interactivity is also of use to the developer. Some algorithms are inherently dependent upon threshold values and parameters. Providing the functionality for these to be modified at run-time allows the programmer to test varying values without having to modify and recompile the code. Once the final value has been found it is then possible to remove the user-option and hard code as a constant if required.

Support for large, high-dimensional, time-dependent simulations

The application is used to visualize the results of large simulations comprised of many time-steps. Not every time step has to present in main memory simultaneously. Our application uses a streaming approach to handle large data sets. A separate data management thread continually runs in the background. When a time-step has been used this manager is responsible for unloading the data for a given time-step and loading in the data for the next (offline) time-step. A separate thread in used to minimize the interruption that occurs from the blocking I/O calls. If a single threaded solution was used the system would compute the visualization as far as possible with the in-core data and then have to halt until the new data is loaded. Note that in many cases the visualization computation still out performs the data loading in a multi-threaded solution, however, the delay may be greatly reduced.

Simple API

The system is intended for future developers to utilize. In order to achieve this the system must be composed of an intuitive, modular design maintaining a high level of re-usability. Extensive documentation and coding conventions [Laramee (2010)] are maintained to allow new users to be able to minimize the overhead required to learn the system. The system is documented using the doxygen documentation system [van Heesch (197-2004)], the documentation can be found online at http://cs.swan.ac.uk/∼cstony/documentation/.

4. System design and implementation

Figure 3 shows the design of our application. The major subsystems are shown along with the relationships of how they interact with one another.

The *Graphical User Interface* subsystem is responsible for presenting the user with modifiable parameters and firing events in response to the users actions. The user interface is designed to be minimalistic. It is context sensitive and only the relevant controls are displayed to the user at any time. The GUI was created using the wxWidgets library [wxWidgets GUI Library (n.d.)]. wxWidgets provides a cross-platform API with support for many common graphical widgets – greatly increasing the efficiency of GUI programming. The *3D Viewer* is responsible for all rendering. It supports the rendering of several primitive types such as lines, triangles and quads. The 3D viewer is implemented using OpenGL [Architecture Review Board (2000)] for its platform independence. The *Simulation Manager* stores the simulation data. It stores vector quantities such as velocity and scalar quantities such as pressure. The simulation manager is also responsible for ensuring the correct time-steps are loaded for the desired time. The *Visualization System* is used to compute the visualization results. This system is comprised of several subsystems. Each major system of the application is now described in more detail.

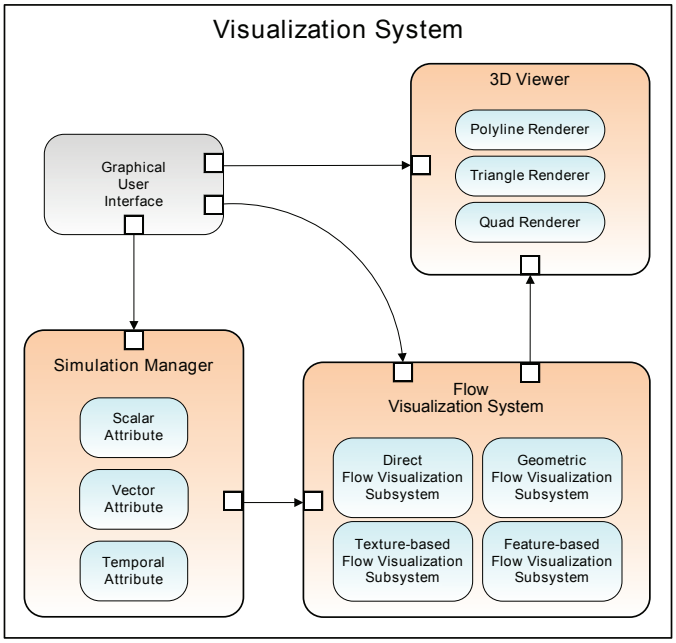

Fig. 3. An overview of our system design. This shows the major subsystems of the framework and which systems interact with one another.

4.1 Visualization system design

The visualization system is where the visualization algorithms are implemented. The application is designed to separate the visualization algorithm logic, the rendering logic, and the GUI. This allows part of the visualization system to be integrated into other applications – even if they use different rendering and GUI APIs. This system is comprised of four sub-systems.

4.1.1 Geometric flow visualization subsystem

Figure 4 illustrates the processing pipeline for the geometric flow visualization subsystem. Input and output data is shown using rectangles with rounded corners. Processes are shown in boxes.

The geometric-based visualization subsystem uses the simulation data as its main input. After the user has set a range of integration parameters and specified the seeding conditions the initial seeding positions are created. Numerical integration is then performed to construct the geometry by tracing vertices through the vector field. This is an iterative process with which an optional refinement stage may be undertaken depending on the visualization method. For example, when using streamsurfaces, extra vertices need to be inserted into the mesh to ensure sufficient sampling of the vector field. Afterwards the object geometry is output. The penultimate stage takes the user-defined parameters that direct the rendering result. Most of the implemented algorithms in our application reside within this sub-system.

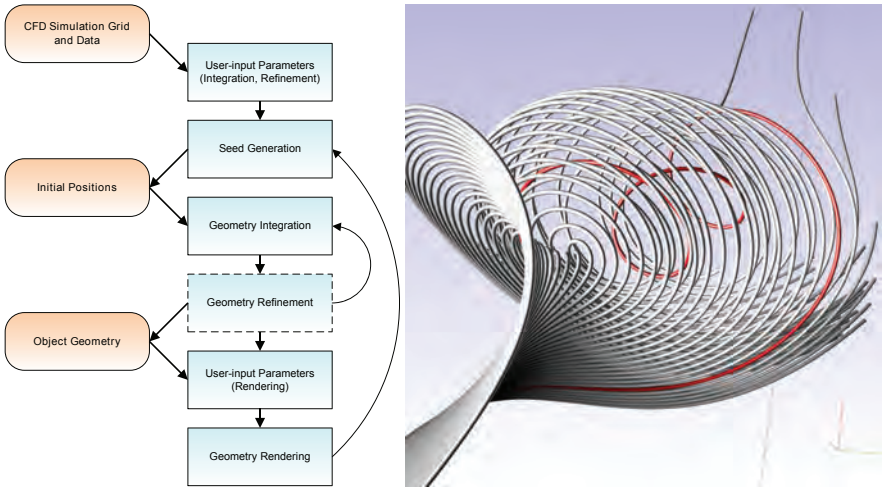

Fig. 4. (Left) The processing pipeline for the geometric flow visualization subsystem. (Right) A set of streamlines generated by the geometric flow visualization subsystem. The streamlines are rendered as tube structures to enhance depth perception and provide a more aesthetically appealing result. The visualization depicts interesting vortical behavior in a simulation of Arnold-Beltrami-Childress flow [Haller (2005)].

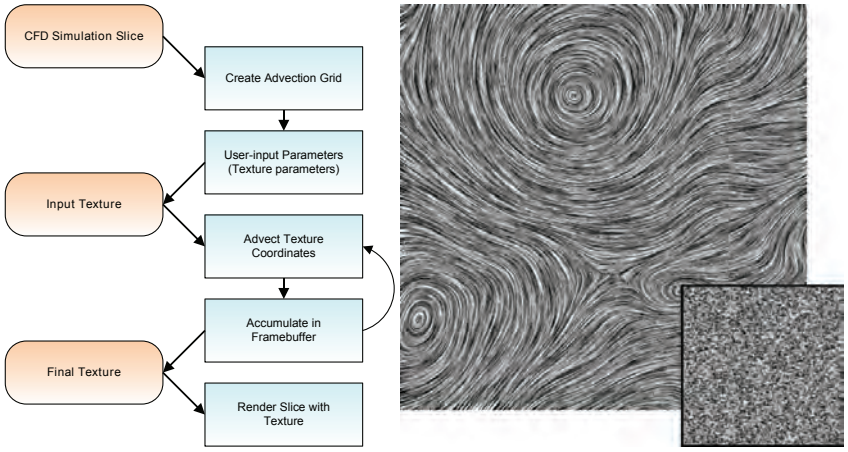

Fig. 5. (Left) The processing pipeline for the texture-based visualization subsystem. (Right) A Line Integral Convolution (LIC) visualization using the texture-based visualization system. This image was generated by 'smearing' the noise texture (inset) along the direction of the underlying vector field at each pixel. The visualization is of a simulation of Hurricane Isabel. The eye of the hurricane can be seen towards the top of the image.

4.1.2 Texture-based visualization subsystem

The texture-based visualization process (Figure 5) also takes in the simulation data as input. An advection grid (used to warp the texture) is then set up and user-parameters are specified. An input noise-texture (Figure 5 inset) is then 'smeared' along the underlying velocity field – depicting the tangent information. The texture advection is performed as an iterative process of integrating the noise texture coordinated through the vector field and accumulating the results after each integration. The resultant texture is then mapped onto a polygon to display the final visualization.

4.1.3 Direct flow visualization subsystem

The direct visualization sub-system presents the simplest algorithms. Typical techniques are direct color-coding and glyph plots. The left image of Figure 6 shows a basic glyph plot of a simulation of Hurricane Isabel. The right image includes a direct color-mapping of a saliency field showing local regions where a larger change in streamline geometry occurs.

4.1.4 Feature-based flow visualization subsystem

Feature-based algorithms may involve a lot of processing to analyze entire the simulation domain. There exists many types of feature that may be extracted (such as vortices), and each feature has a variety of algorithms to detect/extract them. In our application we implemented extraction of critical points (positions at which the velocity diminishes). The right image of Figure 6 shows a set of critical points extracted on a synthetic data set. A red highlight indicates a source or sink exists in the cell and a blue highlight indicates that a saddle point is present in the cell.

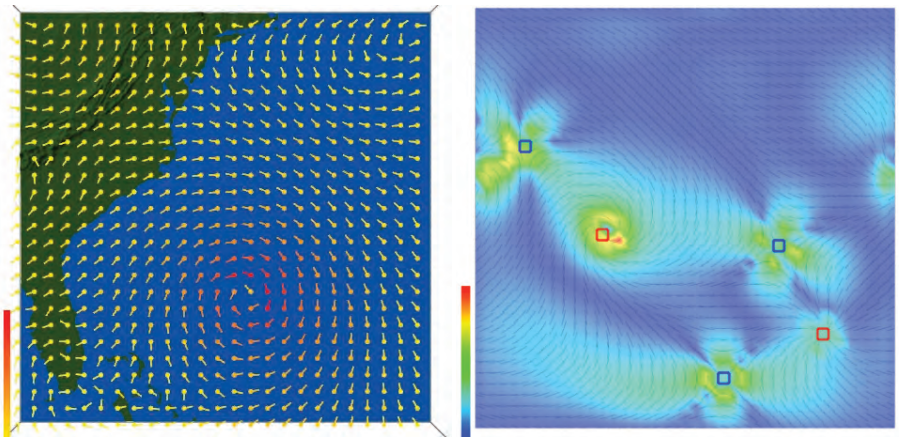

Fig. 6. Direct and feature-based visualizations. The left image shows a basic glyph plot of the velocity field of a simulation of Hurricane Isabel. The right image shows the critical points extracted on a synthetic data set. The cells that contain the critical points are highlighted. A red highlight indicates the critical point is a source or a sink and a blue highlight indicates a saddle point. This visualization also contains a direct color-mapping of a saliency field based on local changes in streamline geometry.

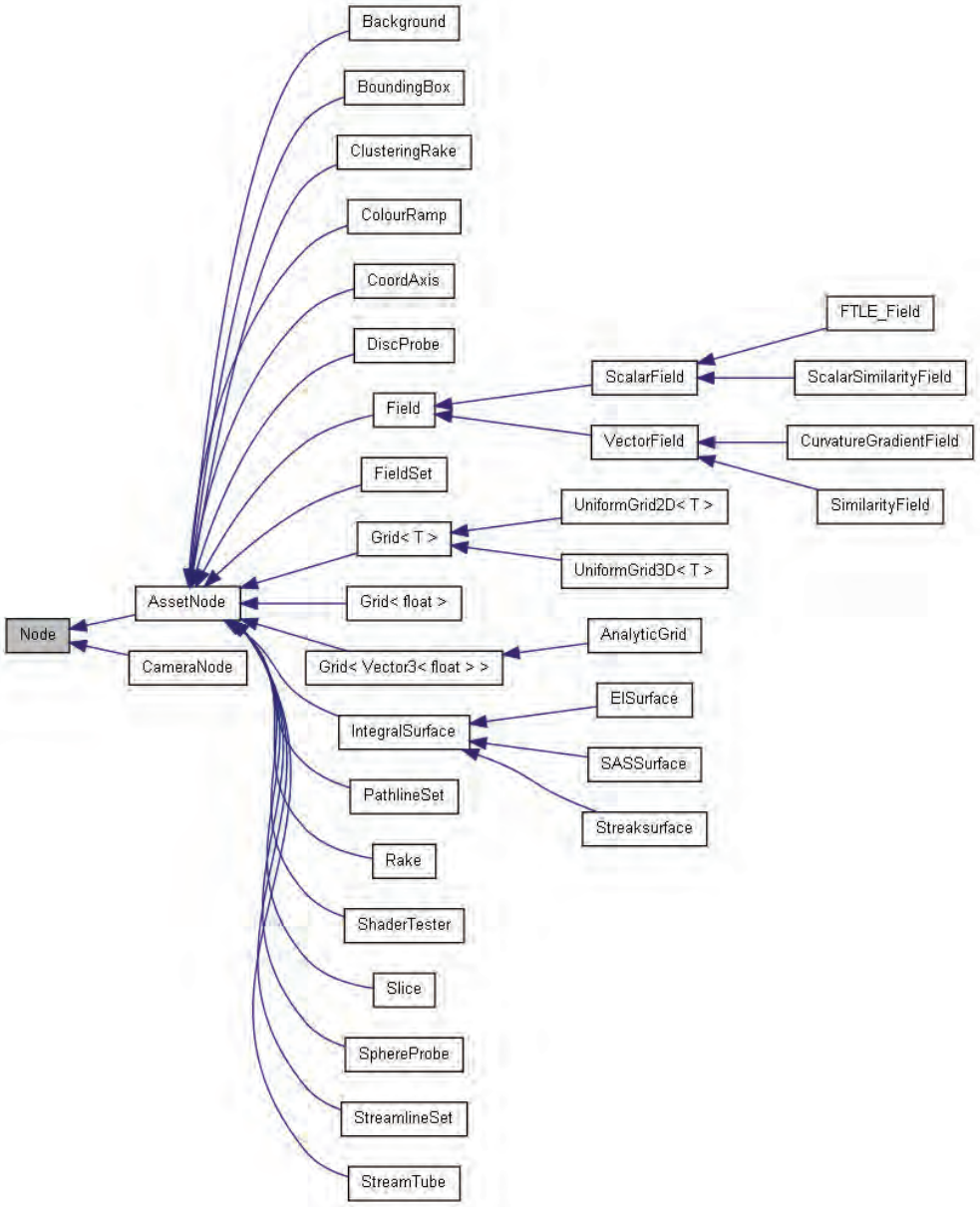

Fig. 7. Inheritance diagram for the node classes. The asset node is the interface from which all integrated visualization techniques inherit and implement.

4.2 Graphical user interface and asset management

The perfect visualization tool does not (yet) exist. Each piece of research that has been undertaken over the past several decades focuses on a specific problem. Thus, a general solution that is suitable for all visualization problems has not been discovered – and may never be found. To this end, visualization applications must support a variety of techniques in order to be useful. When referring to a visualization tool/technique in terms of our software, we refer to them as *assets*. Our asset management system is designed with the following requirements:

- A common interface for assets, simplifying the process of adding new assets in the future and ensuring the application is extendable.
- A common interface between assets and the application GUI. Again this simplifies expansion in the future and ensures a basic level functionality is guaranteed to be implemented. This also provides a consistent user interface for the user.
- Enforcing the re-use of existing code.
- The same method of adding assets for the visualization at run-time.

Fortunately the object-oriented programming paradigm and the C++ programming language provides us with a powerful set of tools to realize these requirements. The rest of this section discusses aspects of the GUI design and our framework for managing the visualization assets.

4.2.1 Application tree and scene graph

In order to provide a flexible system, that allows the user to interactively add and remove assets at run-time, we utilize a scene graph. A scene graph is a tree data structure in which all assets are represented by nodes within the tree. When a frame is rendered, a pre-order, depth-first traversal of the tree is carried out, starting from the root node. As each node is visited, it is sent to the rendering pipeline. Transformations applied to a node are passed onto it's children. We provide two node types: *Asset Nodes* and *Camera Nodes*. These are derived from a base node which provides a common interface and are not directly instantiable. The inheritance diagram for the node types is shown in Figure 7.

The tree structure used for the scene graph lends itself to be represented using a GUI tree control (see Figure 8). The tree control directly depicts all of the nodes in the scene graph and the tree hierarchy. The user manipulates the scene graph through the tree control. Assets can be added to the scene graph by selecting a node to which an asset is attached. Right-clicking upon an asset presents a context menu with an option to add a new node into the scene graph (see Figure 9). Following this option another context menu is presented with a variety of assets which the user is able to add. When an asset is selected to be added, it is inserted into the scene graph as a child node of the currently selected node (the node which was right-clicked). Removal of a node is achieved using a similar method – the right-click context menu gives the option of removing a node. When a node is removed from the scene graph all of its children are also removed. This ensures that there are no dangling pointers and acquired resources are freed. The *resource acquisition is initialization* (RAII) [Meyers (2005)] programming idiom is obeyed throughout the application to ensure exception safe code and resources are deallocated.

From a user perspective, this system allows a flexible method with which to interactively add and remove the visualization tools at run-time. The current tool set is always displayed to provide fast and easy access. From a developer perspective, this system provides a consistent

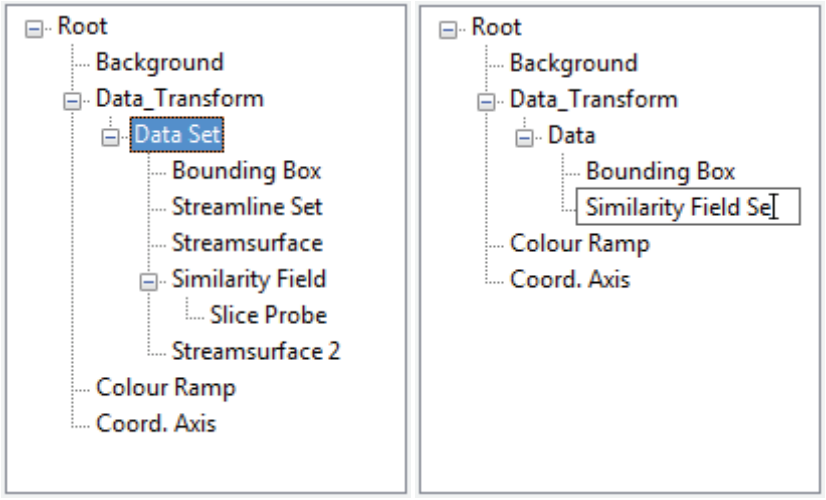

Fig. 8. Screenshots of the application tree during the run-time of different sessions. The application tree is a GUI tree control that represents the nodes in the scene graph. (Left) Several visualization assets, such as streamline sets and slice probes, are currently being employed. (Right) The user is editing the label of one of the assets.

interface. The logic for adding and removing a node is maintained in the scene graph, application tree, and node classes. It does not need implementing on a per-asset basis. When a new visualization technique is implemented, all that is required is that the developer inherits from the asset node class and provides the implementation for the pure virtual functions described by the abstract asset node class (described in more detail in Section 4.2.3). In addition to the asset node, we provide a class called *camera node* which is responsible for storing and configuring the projection and viewpoint information. We now discuss the camera node and the asset node classes in more detail.

4.2.2 Camera node

3D APIs such as OpenGL and DirectX have no concept of a camera. The viewpoint is always located at the position $(0.0, 0.0, 0.0)$ in eye-space coordinates (for a thorough discussion of coordinate spaces and the OpenGL pipeline we refer the reader to [Woo et al. (2007)]. However, the concept of a camera navigating through a 3D scene provides an intuitive description. We can give the appearance of a movable camera by moving the scene by the inverse of the desired camera transformation. For example, to simulate the effect that the camera is panning upwards, we simply move the entire scene downwards.

As outlined in Section 4.2.1, all child nodes inherit the transformations of their parent. The camera node is set as the root node in the scene graph. The inverse transformation matrix is re-computed when the camera is manipulated. All other nodes are added as a descendant of the camera node and are, therefore, transformed by it's transformation matrix. Thus, the camera parameters are the main factor for setting the viewpoint and orientation. This is in line with the camera analogy described at the beginning of this section.

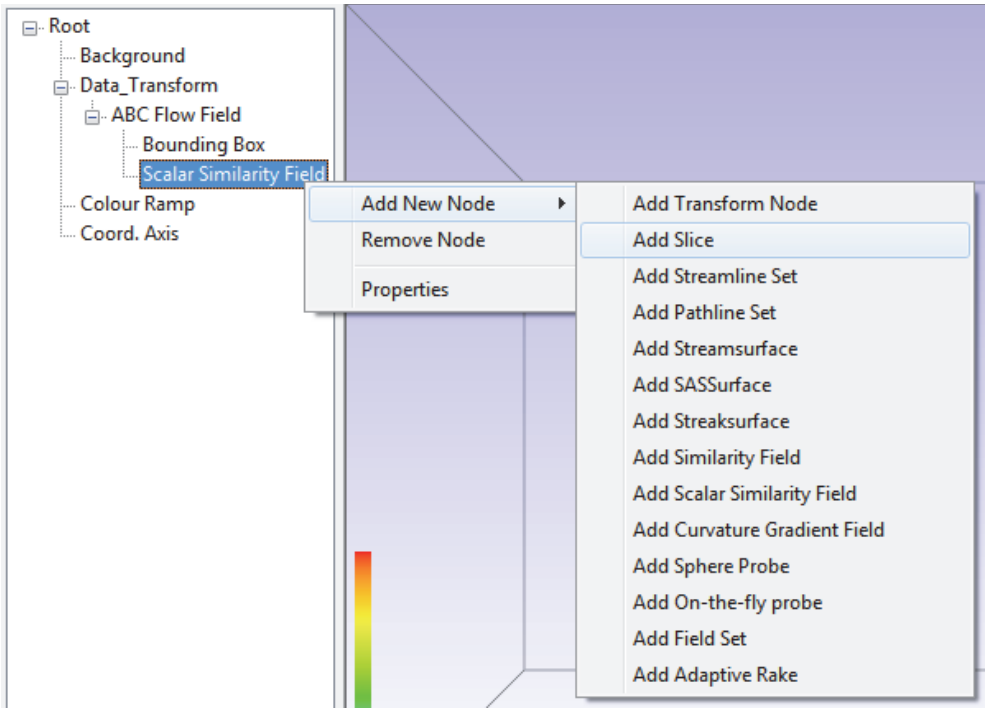

Fig. 9. The tree control is used to add new nodes into the scene graph. The user selects which node they want add an asset to. A context menu then presents the user with a list of assets. When an asset is selected it is added to the scene graph as a child node of the currently selected node.

This method can be extended to render to multiple viewports with different view points. This could be realized by maintaining a list of camera nodes, each maintaining their own set of view parameters (like using multiple cameras in real-life). For each view point, the relevant camera node can be inserted into the root node position. The scene graph is then traversed sending each node to the rendering pipeline. This allows the same scene graph to be used, the only change is the initial camera transform.

4.2.3 Asset node

Figure 10 shows the collaboration graph for the asset node class. This class is designed to provide a consistent interface for all visualization methods integrated into the application. It is an abstract class and therefore provides an interface that declares common functionality. The class provides three pure virtual function signatures:

- SendToRenderer()
- setDefaultMaterial()
- loadDefaultConfiguration()

The SendToRenderer() function issues a command to the class to send it's geometry to the 3D viewer. The setDefaultMaterial() function is an initialization function that sets

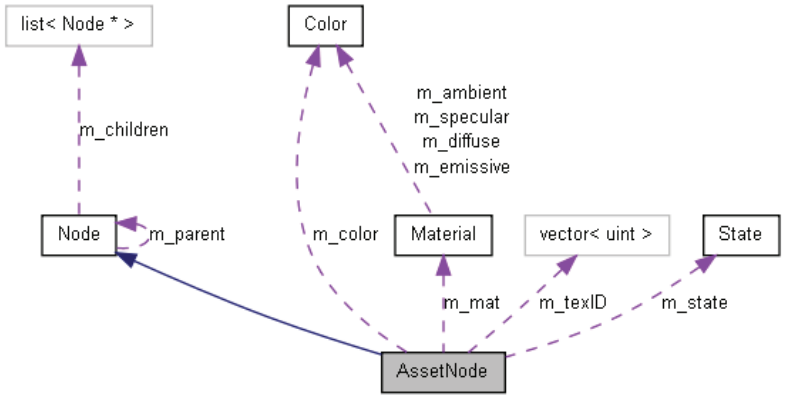

Fig. 10. Collaboration diagram for the asset node class. The boxes represent classes in our framework.

the initial/default material properties that are used by the OpenGL API. Finally the loadDefaultConfiguration() function loads the default set of parameters for the visualization method from file. The configuration files follow the INI file format [*INI File* (n.d.)]. This function is provided to ensure that all visualization methods are loaded with sensible default values (where necessary). By providing the configuration information in a file and not hard-coding it into the application brings several benefits. A change to default parameters does not result in any re-compilation bringing speed benefits during development. It also means that the end user can change the default settings without having to have or understand the source code. It also allows users on different machines to each have their own set of default parameters tailored to their requirements. It would be a simple task to allow per-user configuration files on a single machine, however we have not implemented this functionality as it is superfluous to our requirements as a research platform.

The asset node class also provides several member variables that are inherited:

- (unsigned int) m_vboID
- (unsigned int) m_indexID
- (vector⟨unsigned int⟩) m_texID
- (Material) m_mat
- (Color) m_color
- (State) m_state
- (bool) m_inited

OpenGL assigns numeric ID's to all buffers. Asset nodes provide variables to store a vertex buffer (m_vboID), an index buffer (m_indexID) and a list of textures (m_texID). More than one texture can be assigned to an asset node in order to facilitate multi-texturing. Materials are settings that affect how geometry reflects to the light within OpenGL. A material is separated into several components: *ambient*, *diffuse*, *specular* and *emissive*. The asset node provides all renderable objects with a material property. It also provides a color property, this is used in a similar fashion to material but it much more lightweight with less flexibility. OpenGL is a

state machine, where the current state affects how any primitives passed to it are rendered. Whether lighting and/or texturing is enabled are examples of some of the states used by OpenGL [Architecture Review Board (2000)]. Every asset node has a state member variable which allows the node to stores various OpenGL state settings plus other non-OpenGL state parameters. The state class is described in more detail in Section 4.3.2.

4.2.4 Asset user-interaction and the asset control pane

User specified parameters for the various visualization assets are provided through the asset control pane. When an asset is selected in the application tree control (Section 4.2.1), the asset control pane is updated. The asset control pane shows only the controls for the currently selected asset. This helps reduce clutter in the GUI and provides an easier experience for the user. The asset control panel also populates the controls with the current values of the asset, therefore the GUI always represents the correct values for the selected asset. The asset panel can be seen in the blue box of Figure 1.

The use of C++ pure virtual functions ensures that the GUI panels for all visualization assets must implement functionality to update itself according to the current state of the active asset it is controlling. The GUI panels are now discussed in more detail.

4.2.5 Asset panels

Figure 11 shows an examples of the asset panel at runtime. The left panel shows the controls displayed when a streamsurface asset is selected by the user. The right image shows the asset panel when the user has selected a different visualization asset, in this case a streamline set. Note how the streamsurface panel has now been removed and it replaced with the streamline set panel. Other relevant controls for the selected tool (such as state parameters) are neatly set in separate tabs. This has two benefits. It keeps the visualization tool parameters and the OpenGL rendering parameters for the tool separate. We can also re-use the same GUI panel for state controls as the parameters are common across all visualization methods.

Asset panel controls are event-driven. When a control is modified an event is fired which is then handled by the visualization system. The event handler typically obtains an handle to the currently selected visualization asset and calls the correct function. The visualization system is then updated and feedback is presented to the user. Asset panels utilize multiple inheritance. While multiple inheritance has its shortcomings, i.e., the diamond problem, but can provide powerful solutions to problems if used with care. Figure 12 shows the inheritance diagram for a typical asset panel (in this case the streamsurface panel). Note that only a single level of inheritance is used. Throughout the design of this system, keeping the inheritance levels as low as possible was set out as a requirement. This ensures shall depth of inheritance trees (DIT) which makes the code easier to extend, test, and maintain. All asset panels inherit from two base classes. One of these classes is unique to all derived classes and the other one is common to all derived classes. The class *CommonPanel*, as its name implies, is inherited by all asset panels. It contains information such as the string that is displayed when the panel is shown in the asset control pane and enumeration of the panel type. It also provides the signature for a pure virtual function, $UpdatePanel()$. This function is used to populate the panels controls with the correct values (by querying the currently selected asset). The second class panels inherit from are unique auto-generated classes that are output from using a GUI building tool called wxFormBuilder. The auto-generated classes provide panel layout and controls. They also provide interface for the events that are fired from that panel. The asset

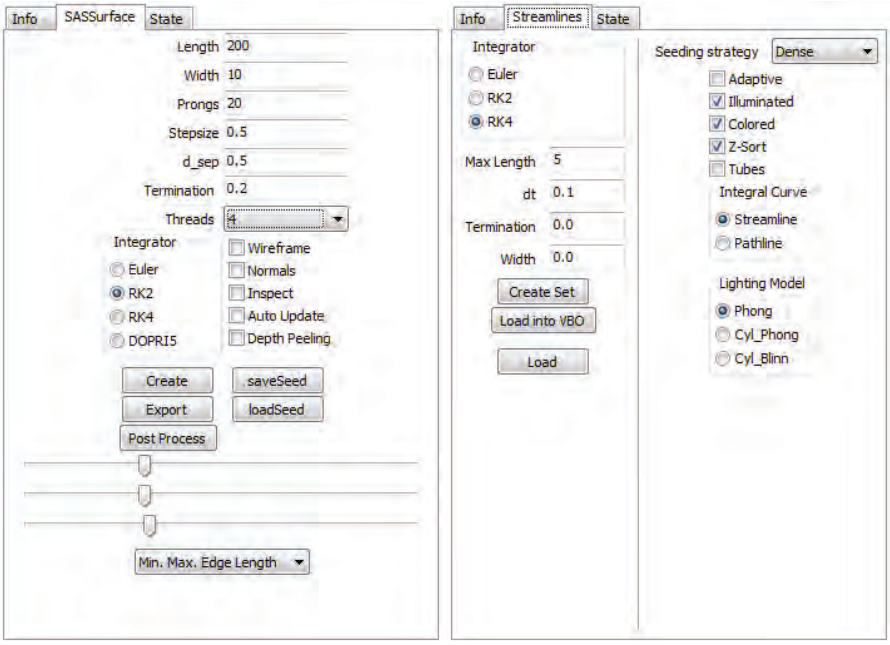

Fig. 11. Two examples of asset panels taken at runtime. The panel on the left shows the controls for streamsurfaces. When a streamsurface asset is selected in the application tree, this panel is inserted into the asset control pane. The right image shows the result of the user then selecting a streamline set asset. The streamsurface control panel is removed from the asset control pane and the streamline set control pane is inserted in its place. Only the relevant controls to the currently selected asset are displayed to the user. This leads to a less cluttered GUI and the user is not burdened with manually navigating the GUI to find the appropriate controls.

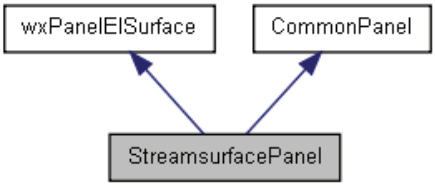

Fig. 12. Inheritance diagram for asset panel types. This example shows the streamsurface panel. Asset panels use multiple inheritance, they inherit from *CommonPanel* and another class that is auto-generated using a GUI builder. Using this method provides fast creation of GUI controls (using the form builder and generated class) and allows us to provide a common interface and behavior for all panels (using the common panel class).

panel then provides the implementation for the interface. In our system the auto-generated classes are prefixed with the letters "wx" to differentiate them from user created classes.

The asset panels are designed with both developers and users in mind. The updating panel in the asset control pane ensures that only the relevant controls are displayed. The controls are also located in the same place within the application. Therefore, the user does not have to search around for various options. Similar to the node structures (Section 4.2.1), the panels are organized in a manner that facilitates easier implementation that ensures a certain level of functionality. The use of a GUI builder greatly facilitates the developer increasing the productivity when creating GUI components.

4.3 3D viewer

This section details the 3D viewer system of our application. We discuss some key implementation details and outline how our application manages various rendering attributes such as materials and textures.

The 3D viewer system is implemented using the OpenGL API. The OpenGL API was chosen because it provides a high level interface for utilizing graphics hardware and it is platform independent. The 3D viewer is responsible for providing the feedback from the visualization software. Recall that OpenGL defines a state machine whose current rendering state affects how the primitives that are passed through the graphics pipeline are rendered. State-machines can make debugging difficult; unexpected behavior may arise simply from a state being changed that the developer is unaware of. Querying the current state may be difficult at times and almost always relies on dumping text to a console window or file. To try and alleviate this issue our system implements a wrapper around for OpenGL state machine. Our *OGL_Renderer* (OpenGL Renderer) class provides flags for the OpenGL states used within our system. Other states may be added as they are added and utilized by the system. We also provide accessor and mutator functions for retrieving and manipulating state values. Our wrapper provides several benefits:

- Breakpoints may be set to halt the program when a specific state value has been modified.
- Bounds checking may be performed as states as a sanity check, making sure no invalid values are set.
- When using an integrated development environment (IDE), the class can be queried easily and does not rely on the outputting of large volumes of text that the user has to manually search through.
- Some OpenGL code can be simplified making development easier and more efficient.
- Separating the graphics API code allows for other APIs to be used in the future if the requirement arises. This is very difficult if API specific code is embedded throughout the entire codebase.
- It aids the developer by being able to focus more on the visualization algorithms rather than the rendering component. Thus, promoting the system as a research platform.

Our system only requires a single rendering context (if multiple viewports are present, the same rendering context can be used). We utilize the Singleton design pattern [Gamma et al. (1994)] so that instantiation of the OGL_Renderer is restricted to a single instance. We note that a singleton has downsides as it is in essence a global variable. However, the OpenGL state machine is inherently global and the fact we only want a single rendering context makes a singleton suitable for our needs. In our case, a singleton provides a much cleaner solution than continually passing references around is much preferably than every object (that needs

to) storing it's own reference to the renderer object. Access to the singleton is provided by using the following C++ public static function:

```
static OGL_Renderer& OGL_Renderer::Instance()
{
   static OGL_Renderer instance;
   return instance;
}
```

The first time that this function is called, an instance of the OGL_Renderer is created and the reference to it is returned. Future calls to this function do not create a new instance (due to the static variable) and a reference to the current instance is returned.

4.3.1 Rendering

OpenGL rendering code can be ugly and cumbersome if not carefully designed. The API uses C-style syntax which does not necessarily interleave itself well with C++ code in terms of code readability. Many calls are usually made to set the OpenGL state before sending the geometry data along the rendering pipeline. Here is an example of OpenGL code that renders a set of vertices that are already stored in a vertex buffer on the GPU.

```
...
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferID);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(Vector3<float>), NULL);

glBindBuffer(GL_ARRAY_BUFFER, normalBufferID);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, sizeof(Vector3<float>), NULL);

glBindBuffer(GL_ARRAY_BUFFER, textureBufferID);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(1, GL_FLOAT, sizeof(float), NULL);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexId);

glDrawElements(GL_TRIANGLE_STRIP, numVerts,
               GL_UNSIGNED_INT, NULL);

glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, NULL);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, NULL);
...
```

This renders an indexed set of vertices as a strip of triangles with shading and texturing information. First the buffers and pointers into them are set as well. The vertices are then passed down to the rendering pipeline. The state changes are undone after rendering to put the OpenGL back into its original state. It is clear this is not the simplest code to work with. If the rendering code was merged into the visualization code, all renderable objects would possess similar code chunks. This (1) makes the code harder to read and (2) produces a lot of repetitive code throughout the codebase.

Our system segregates this type of rendering code. We provide classes such as *TriangleRenderer* and *LineRenderer* which contain utility functions that simplify the rendering process. A typical usage of the triangle renderer is shown below.

```
...
TriangleRenderer::RenderTriangle_VBO(m_vboID,
                                     m_indexID,
                                     m_numberOfIndices,
                                     TRIANGLE_STRIP);
...
```

This call to the RenderTriangle_VBO function passes in the required buffers, the number of vertices to be rendered and the rendering mode. This approach allows the developer to take advantage of code re-use and makes the code much more readable.

4.3.2 State objects

We provide a *State* class that encapsulates various OpenGL states that are utilized by our visualization assets. The state class has the following members:

- (bool) m_lighting;
- (bool) m_texturing;
- (bool) m_blend;
- (uint) m_program;
- (int) m_stateBlendSrc;
- (int) m_stateBlendDst;
- (bool) m_render;

The first three bool members are flags indicating whether the matching OpenGL state will be enabled. The m_program member is the ID of the shader program that is used to render the asset. The blend members store the blending states when blending is enabled. The final member, m_render, indicates whether the asset is rendered or ignored. This member has no counterpart in the OpenGL state machine. It is included to allow the user to disable the rendering an asset without removing it from the scene graph. The state class has a member function, *SetState()*, which is called immediately before the asset is rendered.

```
void State::SetState()
{
   OGL_Renderer& renderer = OGL_Renderer::Instance();

   if(m_lighting)  renderer.Enable(LIGHTING);
   if(m_texturing) renderer.Enable(TEXTURING);
   if(m_blend)
   {
      renderer.Enable(BLEND);
      renderer.SetBlendFunc(m_stateBlendSrc, m_stateBlendDst);
   }
   renderer.UseProgram(m_program);
}
```

After the asset has been rendered the OpenGL state is returned to its original state by calling the *UnsetState()* function of the state.

```
void State::UnsetState()
{
    OGL_Renderer& renderer = OGL_Renderer::Instance();

    if(m_lighting)  renderer.Disable(LIGHTING);
    if(m_texturing) renderer.Disable(TEXTURING);
    if(m_blend)     renderer.Disable(BLEND);
    renderer.UseProgram(NULL);
}
```

These functions greatly simplify the rendering code and aid the developer in efficiently managing the OpenGL state machine and reduce unexpected behavior arising from incorrectly configures states. The state objects are utilized every time an asset is rendered. The code segment below outlines there usage within our application framework.

```
void AssetType::SendToRenderer()
{
    if(m_state.RenderingEnabled())
    {
        m_state.SetState();
        //Rendering Code
        ...
        m_state.UnsetState();
    }
}
```

4.3.3 Material objects and lights

Within OpenGL (and other rendering APIs), the currently bound material state affects how primitives that are passed to down the rendering pipeline interact with light sources. OpenGL lighting is comprised of various terms that approximate the behavior of light in the real-world. In computer graphics and visualization the aesthetics of the final rendering result are very important for high quality results. A research paper looks more polished and professional with high quality images. We recognize this importance and provide functionality that allows the user to adjust the various lighting and material parameters at run-time.

We allow the user to interactively add and remove light sources. The type of light source and its position can also be controlled by the user. The application also allows the user to set the values of each component of the light source (ambient, diffuse and specular). Likewise with materials we allow the user to adjust each component (ambient, diffuse, specular, emission and specular power). Allowing this level of control at run-time allows the user to receive immediate feedback of the results and prevents any unnecessary recompilation and re-generation of results.

Each asset has it's material object which encapsulates the state behavior. Prior to the asset being rendered it's material is bound by the OpenGL state machine. Once again, by separating the rendering code from the visualization asset code we are promoting code-reuse and not cluttering up the visualization asset classes with rendering code.

Note we omit a through discussion of how OpenGL approximates lighting and materials. Instead we refer the interested reader to [Woo et al. (2007)].

4.3.4 Textures, texture management and texture editing

As we have previously discussed, our application has served as a research platform for flow visualization techniques. More specifically we have focused on a sub-set of flow visualization techniques that fall into the geometry-based category. These methods compute a geometry that represents some behavior of a flow field. However, by color-mapping this geometry we can depict more information about the flow behavior than the geometry alone. For example, velocity magnitude is often mapped to color.

Color-mapping can be achieved in a variety of ways. A function may be provided that maps the color, although for complex mappings defining a suitable function may be difficult. A large lookup table may be produced, this is a flexible solution but can lead to the developer producing lots of code to produce large look up tables.

Our approach to color-mapping utilizes texture-mapping. Here the texture itself is the lookup table and all we have to do is provide the texture coordinate to retrieve the desired value from the texture. Textures are a very powerful tool in computer graphics and rendering APIs readily provide functionality for various interpolation schemes which we can utilize. They are also fast as they due to their hardware support. This system is also very flexible, new color-maps (in the form of images) can be dropped into the textures folder of the application and they will automatically be loaded the next time the application is run. Management of the texture is equally simple, the texture manager simply maintains a list of textures, the user can select the texture they wish to use from the GUI and the texture manager binds that texture to the OpenGL state.

Fig. 13. Some images from an interactive session with the color map editor. The top-left image shows the initial state of the editor. The top right image shows the result when the user inserts a new sample (black) in the center of the color map. The bottom-left image shows the result after the user has updated the color of the middle sample to yellow. Finally the bottom-right image shows the effect that dragging the middle sample to right has. The color values between each sample are constructed using interpolation.

We also provide a tool that allows the user to create their own color maps. This allows the user to customize the color-mapping at run-time to ensure that the mapping adequately represents the information they wish to present. Figure 13 shows some steps of an interactive session with the editor. The editor allows the user to insert (and remove) samples along the color map. The color of the samples can be altered and the position of the sample can be updated by dragging it around in the editor window. The colors values are interpolated between samples. An up-to-date preview of the color map is always displayed within the editor.

4.4 Simulation manager

The final major system in our application is the simulation manager. The simulation manager is responsible for loading the simulation data and managing the sub-sets of simulation data when it won't fit in core memory. The simulation provides a set of classes for 2D and 3D simulations. The simulation manager handles both discretely sampled data, such as the output from CFD simulations, and analytically defined data by providing the necessary parameters to a function to compute the vector information. Flow simulations are output in a variety of file formats using both ASCII and binary file output. Our application supports a range of formats and provides a simple interface for developers to add support for more formats in future.

The simulation manager is used whenever a vector field evaluation is requested by one of the visualization assets. It is responsible for determining whether a given position lies within the domain (both spatially and temporally). If the position is determined as valid, the simulation manager populates a cell object (of the corresponding grid type) with the appropriate vector values. The cell objects also belong to the simulation manager and are used to construct the final vector value at the desired position using interpolation.

4.4.1 Large-time dependent simulation data

As previously discussed, the output from time-dependent CFD simulations can be of the order of gigabytes or even terrabytes. Thus, we have to consider out-of-core methods. Our application handles such large amounts of data by only loading a sub-set of the simulation into memory. In order to perform a single integration step, only two time-steps need to be copied to main memory. For example if each our simulation output data for every second and we need to advect a particle at $t = 3.5s$, only time-steps 3 and 4 four are needed to interpolate the required vector values.

We employ a method similar to Bürger et al. [Bürger et al. (2007)]. We allocate a number of slots equal to the number of time-steps that fit into main memory. These slots are then populated with the data from a single time-step, starting with the first time-step and proceeding consecutively. For example, if we can fit 6 time-steps into memory we allocate 6 slots and populate them with time-steps 0-5. When we have passed through a time-step it's data is unloaded and the next time-step is loaded from file in it's place. For example, if we are constructing a pathline, when $t \geq 1$, the first slot (which holds the data for timestep 0) is overwritten with the data for timestep 6 – the next unloaded time-step in the simulation. Figure 14 illustrates an example.

Conceptually, a sliding window run over the slots, with the pair of slot covered by the window being used for the current vector field evaluations. When the sliding window has passed a slot, the slot is updated with the next unloaded time-step. When the sliding window reaches the last slot it wraps around to the first slot and the cycle is repeated. The sliding

Time-step	0	1	2	3	4	5
Slot	0	1	2	3	4	5

(a) $0 \leq t < 1$

Time-step	6	1	2	3	4	5
Slot	0	1	2	3	4	5

(b) $1 \leq t < 2$

Time-step	6	7	8	9	10	5
Slot	0	1	2	3	4	5

(c) $5 \leq t < 6$

Time-step	6	7	8	9	10	11
Slot	0	1	2	3	4	5

(d) $6 \leq t < 7$

Fig. 14. These four tables show the time-steps that are loaded into the simulation manager slots for given time periods. The grey cells show the time-steps that are used to perform any vector field evaluations for the stated time period. (a) Shows the first time period ($0 \leq t < 1$). (b) shows the next time period, the two slots used in the vector field evaluation have moved over – a sliding window. The previous slot has been updated with the subsequent unloaded time-step in the simulation (slot 0 is loaded with time-step 6). (c) The slots wrap around, when the sliding window reaches the last slot it switched back to the first slot. (d) The process repeats with the new time-steps in the slots.

window transition is triggered when a time greater than the current time period covered by the window is requested by the application.

For this method to be effective the simulation manager runs in a separate thread. Disk transfer operations are blocking calls and they halt the rest of the application if a single thread is used. Moving these blocking calls to a separate thread allows the application to proceed computing visualization results while data is loaded in the background. Note, there may be times where the visualization results are computed faster than the simulation manager can load the data. If the required time-steps are not present in memory the application has no option but to halt until they have been loaded. However, even in this case the multi-threaded simulation manager reduces the number and duration of halts compared to a single-threaded solution.

Another consideration that needs to be considered is how the visualization assets are constructed. If we were to generate 10 pathlines by computing the first pathline and then the second one and so on, the simulation manager would have to load all time-steps 10 times (one for each pathline). It is much more efficient to construct all pathlines simultaneously by iterating over them and computing successive points. This ensures that they all require the same sliding window position in the simulation slots and prevents unnecessary paging of data.

5. Conclusion

In a typical research paper many implementation details have to be omitted due to space restraints. It is rare to see literature that provides an in-depth discussion concerning the implementation of an entire visualization application. This chapter serves to provide such a discussion. The chapter provides an overview of the high-level application structure and provides details of key systems and classes and the reasoning why these were designed in this way. Many topics are covered ranging from multi-threaded data management for performance gains to GUI design and implementation with considerations both the developer and the user.

We demonstrate that using a good software engineering practices and design methodologies provide an enhanced experience for both software developers and end-users of the software.

This serves as proof that research code is not restricted to small 'one-off' applications and that implementing proof-of-concept algorithms into a larger framework has many benefits – not least of which is an easier comparison to other techniques.

6. References

Architecture Review Board, O. (2000). *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*, Addison Wesley. D. Schreiner, editor.

Bürger, K., Schneider, J., Kondratieva, P., Krüger, J. & Westermann, R. (2007). Interactive Visual Exploration of Unsteady 3D Flows, *Proc. EuroVis*, pp. 251–258.

Cabral, B. & Leedom, L. C. (1993). Imaging Vector Fields Using Line Integral Convolution, *Poceedings of ACM SIGGRAPH 1993*, Annual Conference Series, pp. 263–272.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design Patters: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Haller, G. (2001). Distinguished material surfaces and coherent structures in three-dimensional fluid flows, *Phys. D* 149: 248–277.
URL: *http://portal.acm.org/citation.cfm?id=370169.370176*

Haller, G. (2005). An objective definition of a vortex, *Journal of Fluid Mechanics* 525: 1–26.

Helman, J. L. & Hesselink, L. (1989). Representation and Display of Vector Field Topology in Fluid Flow Data Sets, *IEEE Computer* 22(8): 27–36.

Hultquist, J. P. M. (1992). Constructing Stream Surfaces in Steady 3D Vector Fields, *Proceedings IEEE Visualization '92*, pp. 171–178.

INI File (n.d.). http://en.wikipedia.org/wiki/INI_file.

Krishnan, H., Garth, C. & Joy, K. I. (2009). Time and streak surfaces for flow visualization in large time-varying data sets, *IEEE Transactions on Visualization and Computer Graphics* 15(6): 1267–1274.

Laramee, R., Hauser, H., Zhao, L. & Post, F. H. (2007). Topology Based Flow Visualization: The State of the Art, *Topology-Based Methods in Visualization (Proceedings of Topo-in-Vis 2005)*, Mathematics and Visualization, Springer, pp. 1–19.

Laramee, R. S. (2010). Bob's Concise Coding Conventions (C^3), *Advances in Computer Science and Engineering (ACSE)* 4(1): 23–36.

Laramee, R. S., Hauser, H., Doleisch, H., Post, F. H., Vrolijk, B. & Weiskopf, D. (2004). The State of the Art in Flow Visualization: Dense and Texture-Based Techniques, *Computer Graphics Forum* 23(2): 203–221.
URL: *http://www.VRVis.at/ar3/pr2/star/*

Laramee, R. S., Jobard, B. & Hauser, H. (2003). Image Space Based Visualization of Unsteady Flow on Surfaces, *Proceedings IEEE Visualization '03*, IEEE Computer Society, pp. 131–138.
URL: *http://www.VRVis.at/ar3/pr2/*

Lorensen, W. E. & Cline, H. E. (1987). Marching Cubes: a High Resolution 3D Surface Construction Algorithm, *Computer Graphics (Proceedings of ACM SIGGRAPH 87, Anaheim, CA)*, ACM, pp. 163–170.

Mallo, O., Peikert, R., Sigg, C. & Sadlo, F. (2005). Illuminated Lines Revisited, *Proceedings IEEE Visualization 2005*, pp. 19–26.

McLoughlin, T., Edmunds, M., Laramee, R. S., Chen, G., Max, N., Yeh, H. & Zhang, E. (2011). Visualization of User-Parameter Sensitivity for Streamline Seeding, *Technical report*, Dept. Computer Science, Swansea University.

McLoughlin, T., Jones, M. W. & Laramee, R. S. (2011). Similarity Measures for Streamline Seeding Rake Enhancement, *Technical report*, Dept. Computer Science, Swansea University.

McLoughlin, T., Laramee, R. S., Peikert, R., Post, F. H. & Chen, M. (2010). Over Two Decades of Integration-Based, Geometric Flow Visualization, *Computer Graphics Forum* 29(6): 1807–1829.

McLoughlin, T., Laramee, R. S. & Zhang, E. (2009). Easy Integral Surfaces: A Fast, Quad-based Stream and Path Surface Algorithm, *Proceedings Computer Graphics International 2009*, pp. 67–76.

McLoughlin, T., Laramee, R. S. & Zhang, E. (2010). Constructing Streak Surfaces for 3D Unsteady Vector Fields, *Proceedings of the Spring Conference on Computer Graphics (SCCG)*, pp. 25–32.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley.

Peng, Z. & Laramee, R. S. (2009). Higher Dimensional Vector Field Visualization: A Survey, *Theory and Practice of Computer Graphics (TPCG '09)*, Cardiff, UK, pp. 149–163.

Post, F. H., Vrolijk, B., Hauser, H., Laramee, R. S. & Doleisch, H. (2003). The State of the Art in Flow Visualization: Feature Extraction and Tracking, *Computer Graphics Forum* 22(4): 775–792.
URL: *http://cs.swan.ac.uk/ csbob/research/*

van Heesch, D. (197-2004). *Doxygen, Manual for version 1.3.9.1*, The Netherlands.

van Wijk, J. J. (2003). Image Based Flow Visualization for Curved Surfaces, *Proceedings IEEE Visualization '03*, IEEE Computer Society, pp. 123–130.

Weiskopf, D., Hopf, M. & Ertl, T. (2001). Hardware-Accelerated Visualization of Time-Varying 2D and 3D Vector Fields by Texture Advection via Programmable Per-Pixel Operations, *Proceedings of the Vision Modeling and Visualization Conference 2001 (VMV 01)*, pp. 439–446.

Woo, M., Neider, J., Davis, T. & Shreiner, D. (2007). *OpenGL Programming Guide, The Official Guide to Learning OpenGL, Version 2.1*, 6 edn, Addison Wesley.

wxWidgets GUI Library (n.d.). `http://www.wxwidgets.org/`.