Article

Design of a Flow Visualisation Framework

Matt Edmunds [1,*]**, Chao Tong** [1]**, Robert S. Laramee** [1]

[1] Computer Science Dept, Swansea University, Wales, UK.

* Author to whom correspondence should be addressed; csmatti@swansea.ac.uk

Abstract: Research software often consists of individual isolated prototype applications. Small proof of concept applications are usually enough for demonstrating new algorithms. However, the unification of new research algorithms into a cohesive software framework has advantages. Adding new features to an existing pipeline reduces implementation overhead. The researcher is more able to compare and contrast existing or previous work with new research. Utilizing previously implemented techniques, researchers are able to combine visualization options in new ways that typical research prototypes cannot. The software application can be made available to the domain expert for evaluation and future use. These goals are in part realized by utilizing recent advancements in game design technology, and by leveraging features available with recent graphics hardware. Described is the design of a feature rich flow visualization software framework in more detail than a typical research paper. In contrast to most research prototypes, the system we present handles real-world simulation data sets by interfacing directly with the commercial package called tecplot. We present a description of our system in detail where as previously published work focuses on the research contribution. The effectiveness and scalability of the approach is also discussed.

1. Introduction

Computational fluid dynamics (CFD) is a rapidly developing tool for fluid analysis [1]. With wide reaching industrial applications such as automotive, aerospace, weather, and medical, its an increasingly important software application used to speed up the design process while reducing costs. With increasing data complexity and size, the ability of the user to interpret the CFD results becomes increasingly difficult. Post processing software aims to alleviate this challenge by providing visualization methods for

Figure 1. The CFD simulation pipeline is comprised of a preprocessing stage, the simulation stage, and the post processing stage. This illustration shows the inputs and outputs of each stage, and highlights where the different tasks reside.
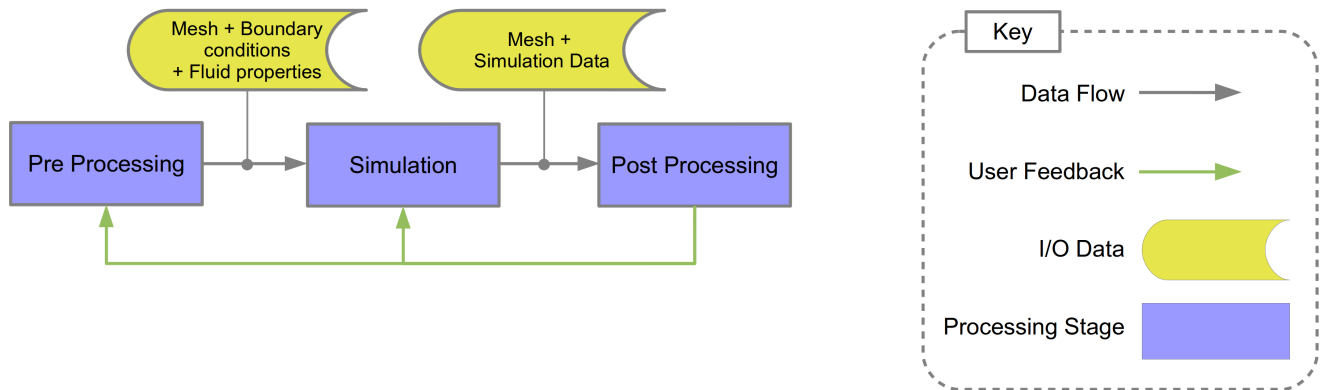
the evaluation and correlation of simulation results. Our visualizations include the latest state-of-the-art flow visualization techniques. Bridging the gap between current visualization software techniques and the latest research techniques is an important goal of our work.

Illustrated in Figure 1 is the CFD simulation pipeline which highlights the inputs, outputs, and processing steps. This process is described in three stages:

1. Preprocessing: A surface and the volumetric mesh is generated to model a physical object. This mesh is generated from computer aided design (CAD) geometry utilizing mesh generation and manipulation tools. Boundary conditions and fluid properties are defined and specified.

2. Simulation: A computational simulation of the fluid is performed using numerical methods applied to the mesh, with respect to the boundary conditions and fluid properties.

3. Post Processing: The simulation result is explored, analyzed, and visualized using a range of techniques dependent on the requirements of the CFD engineer.

In order to present a visualization toolkit which is capable of dealing with large simulation data and complex algorithms, a comprehensive and versatile visualization framework is needed. It is the focus of this chapter to describe the design and implementation of a state-of-the-art flow visualization framework which provides scientists and engineers with effective solutions for the visualization of CFD simulation data. By drawing on recent developments in the game and graphics card hardware industry, and by combining several scientific visualization techniques our visualization framework yields following benefits:

- A platform independent flexible framework which implements state of the art research algorithms in flow visualization. In particular, stream surface seeding algorithms are incorporated which are not available in previous systems such as the VTK or Visit [3].

- A framework specifically designed to enable quick integration of new algorithms for study, testing, and evaluation.

Figure 2. This top level overview of the system shows three main subsections of the software application framework. The GUI subsystem provides the user interface and all associated user interaction and feedback. The logic subsystem encapsulates the processing aspects of the framework. The Services subsystem provides system wide services for use by the other subsystems.
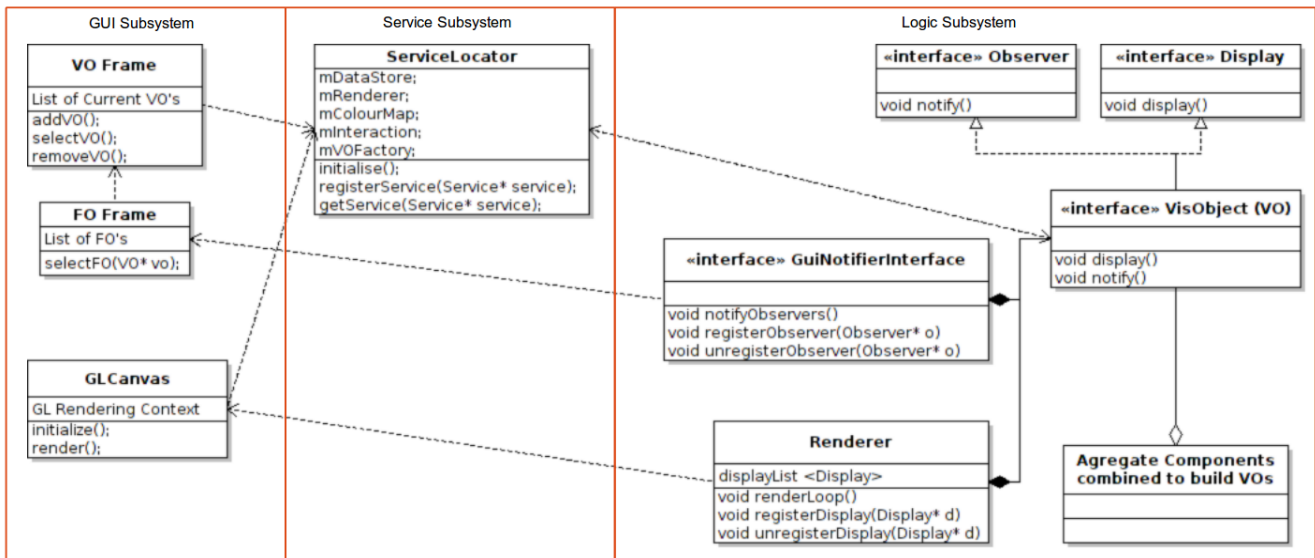
- A smooth and efficient threaded user interface, even when processing large data and complex algorithms.

- Simultaneous comparisons of different techniques.

What we have developed is a novel research prototypes for flow visualization. However, unlike most prototypes, it interfaces directly with commercial software. Thus it combines the best of both worlds: state-of-the-art flow visualization with commercial grade data sets. The visualizations are novel for flow and also offer fast performance. The flow visualization techniques are more current then previous systems such as the VTK or Para-view [3]. What we present here are the design and implementation considerations for our software. The information necessary to implement these features is missing from our previous papers [16][17] because of the strict focus on visualization and space limitations. Also, to see performance details, for example how our software performs compared to previous systems, we refer the reader to Edmunds *et al* [16][17].

The remainder of this paper is organized as follows: Section 2 outlines the software application framework. Section 3 details the design of the GUI. Section 4 describes the services subsystem. Section 5 details the logic sub system. Section 6 highlights our conclusions and proposed future work.

2. System Overview

In this section we start by discussing some design and implementation related work, we then provide an overview of our application framework and related subsystems. Laramee et al. [2] 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. [4] present a generic multi-threading architecture which enables early cancellation of the visualization thread due to user interaction without common pitfalls of multi-threading. They also present an interactive visualization toolkit, HyperMoVal [5], as an implementation of this architecture in practice.

McLoughlin and Laramee [6] describe a flow visualization software framework that offers a rich set of features. Their 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. Peng et al. [7] 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.

Our software application system is divided into three main subsystems; the GUI, the Logic, and the Services. See Figure 2. We divide the system into smaller easier to maintain subsystems using logical groupings of similar functionality. The GUI subsystem is responsible for capturing the user input for computing a visualization, displaying the resultant visualization, and supporting user interaction with the visualization. The Logic subsystem encapsulates the algorithms used to compute the visualizations. This subsystem processes the input data as specified by the user and computes the visualization based on the given user input parameters and selected algorithm. Providing a central point of storage for input and derived data is the Service subsystem.

Another feature of our framework is the Visualization Object (VO). The VO is an encapsulating aggregate of all the elements of an individual feature or individual algorithm. Some examples of features are: Interaction VO's which encapsulate the interaction with the visualization e.g. panning, zooming, and rotating. Color map VO's encapsulating color map functionality. Stream surface VO's which encapsulates the stream surface generation algorithms [8].

For each VO within our framework there is an associated Frame Object (FO); an interface for encapsulating the I/O with the VO. This design approach creates a simple interface for adding new functionality quickly and easily. We further discuss VO's and FO's within our application, along with the detailed description of the structure of our framework in Sections 3, 4, and 5.

3. GUI Subsystem Design

The GUI, shown in Figure 3, is divided into three main frames. The center frame for rendering the current list of VO's, the left frame for listing the current VO's, and the right frame area which displays the FO for the currently selected VO and lists the VO's currently available options. Refer to Figure 3.

VO's are added by right mouse clicking the parent VO category and selecting the required VO from the drop down menu. The VO categories (data, view, scene, visualization) can be seen in the left frame of Figure 3. Right mouse clicking a VO in the list enables the user to delete it. Left mouse click the listed VO and its associated FO is displayed in the right frame area. The FO lists the attributes of the VO which can be manipulated. In Figure 3 the selected VO is a seeding object. The right frame displays the the options available to the seeding object. At the top of this FO a track pad is available for seeding curve manipulation, under this we can specify type and size. We can save and load previously specified seeding curves. At the bottom of this FO we have options for the generation of stream surfaces.

Figure 3. This illustration shows the layout of the user interface.
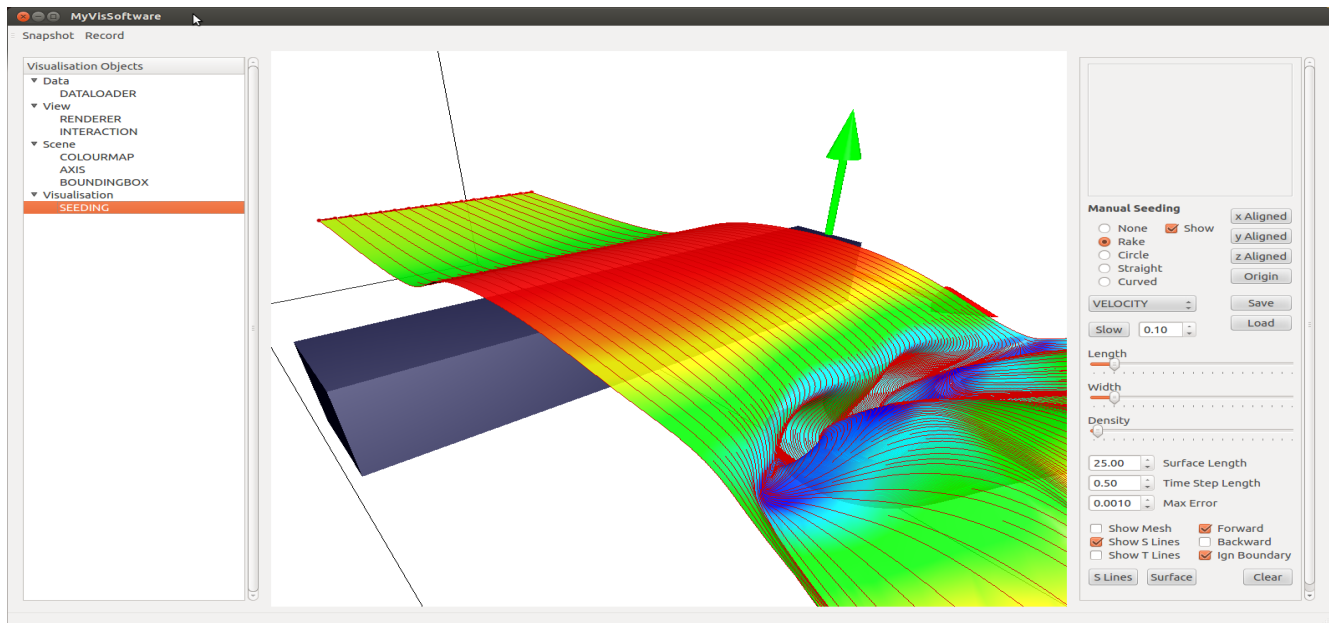
Once we add the required VO's and set FO options, we can see the results rendered in the center frame. Interaction with the visualization is achieved using the mouse. For example: to pan press hold and drag the right mouse button, to zoom press hold and drag the left mouse button, to rotate the view press hold and drag the left and right mouse buttons together.

Each of the three frames within the interface run their actions in separate threads. This allows continued interaction with the interface (such as panning and rotation of the visualization) while waiting for lengthy processes and computations (such as loading large datasets) to finish. As a result of our design approach the application of thread locks to specific VO's is trivial [9].

4. Services Subsystem Design

Some objects in an application can end up communicating with almost all other objects within the code base. It's difficult to find a component of our application that does not need access to objects which encapsulate either input or derived data at some point. Other examples are objects which encapsulate color maps or OpenGL rendering. A method of accessing these objects via interfaces is also desirable. The interface can allow swapping of a given object for an alternative, or allow intermediate objects to be installed which can perform additional tasks such as logging. Systems like these can be thought of as services that need to be available to the entire application.

The Service Locator design pattern [10] [11] [12] is used to meet these requirements in our application. The Service Locator pattern decouples code that needs a service from both who it is (the concrete implementation type) and where it is (how we get to the instance of it). The Service subsystem utilizes the Service Locator design pattern to provide a robust, type safe, central location for accessing system services.

In our services subsystem we focus on data storage and accessibility. The services we implement include; raw data storage, derived data storage, model data storage, render object storage, color map storage, interaction matrix storage, and a VO factory.

In Listing 1 we see the basic layout of the service locator class. The listing shows an initialization function, a register function, and a get function. The service locator is reliant on a service being registered with it before any other objects try to use the service. One challenge here is that a null pointer may be returned if not initialized or a service is registered. It is also important to note that the service isn't aware of the concrete service class just the service interface. The only object in the code that knows about the concrete class is the function that registers the service.

If we try to use the service before a provider has been registered it returns a null pointer. If the calling code doesn't check this we will crash the application. To address the null pointer we implement the Null Object design pattern [13] or in our case a null service. A null service inherits the same service interface but its implementation does nothing. This allows code that receives the object to safely continue on. The calling code will never know that the service isn't available, and we don't have to handle a null pointer. We are guaranteed a valid object.

Listing 1. Service Locator Pattern

```
#ifndef SERVICELOCATOR_H_
#define SERVICELOCATOR_H_

class ServiceLocator {
public:
        static void InitializeDataStore() {
                mDataStore = &mNullData;
        }

        static void Register(DataStoreInterface* service){
                if (service == nullptr) {
                        mDataStore = &mNullData;
                } else {
                        mDataStore = service;
                }
        }

        static DataStoreInterface& GetDataStore() {
                return *mDataStore;
        }

private:
        static DataStoreInterface* mDataStore;
        static NullDataStore mNullData;
```
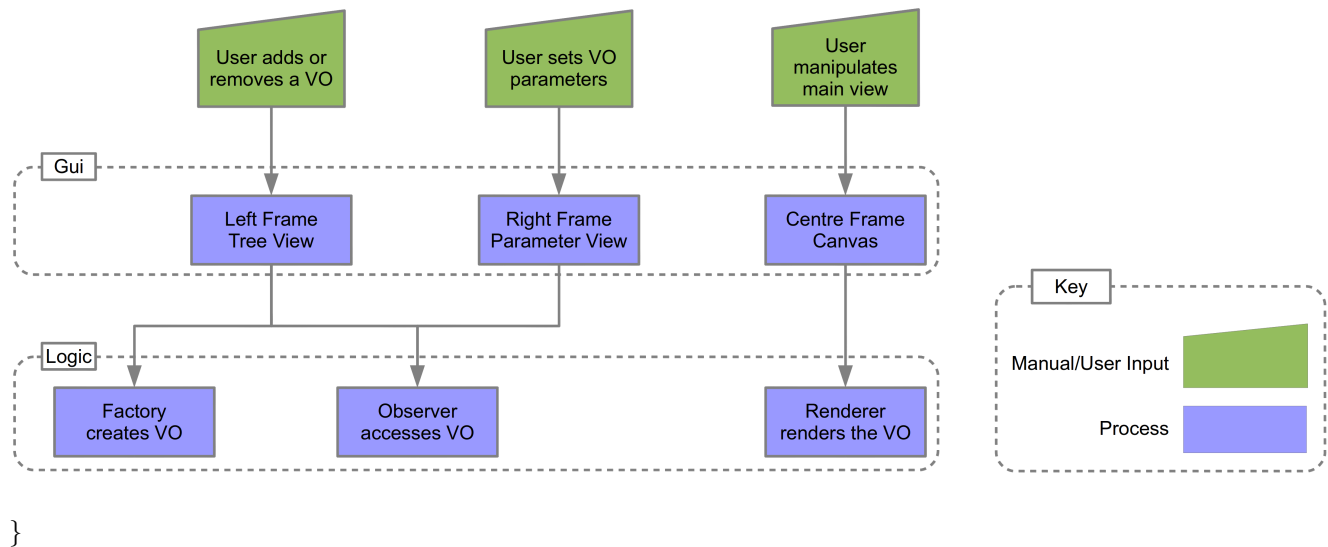
Figure 4. Communication routes from the user to the logic subsystem. User interaction with the GUI is communicated to the visualization object (VO) responsible for the selected visualization algorithm via dedicated subsets of the logic subsystem.

}

endif /∗ SERVICELOCATOR_H_ ∗/

Our service locator is initialized within the main function prior to any other calls. The initialization function registers the null objects. We can then register the any additional services as required. Registering a replacement service is a simple case of just registering the service. We don't delete any services within the service locator class, which provides us the flexibility of swapping services back and forth if required. An example of this is two interaction services storing different view matrices; which ever service is currently registered dictates from which viewpoint the visualization is viewed in the center frame.

Another important feature of this design pattern is the logging or debugging capabilities. Using a decorator design pattern [14] we can easily add an intermediate object which can output either logging or debugging information. Accessing the services is a simple matter of calling the accessor function. This returns a reference to the currently registered service object. Unregistering a service can be achieved by registering a null pointer. The service locator will then register the default null service.

5. Logic Subsystem Design

The Logic subsystem encapsulates the algorithms used to compute the visualizations. This subsystem processes the input parameters selected by the user, and computes the visualization based on the selected algorithm. Figure 4 shows the communication route from user interaction with the GUI, to the selected VO generating the visualization data. There are three main subsets of the logic subsystem responsible for manipulation of the VO. These are; the factory class, the observer class and the renderer class. In this section a discussion of the Factory and Observer design patterns takes place. This is followed by a discussion of the render loop, and finishes with an examination of the aggregate VO and its life cycle.

The Factory

There are many cases when a class needs to create several types of object. This class can have this responsibility but it has to know about each and every object type. The possibility that the class cannot anticipate all the required objects in advance is clear. The idea of the factory pattern is to encapsulate the creation of a multitude of different classes utilizing a common interface. A factory is the location of a class in the code where multiple types of object are constructed. The purpose of utilizing this pattern is to isolate the creation of objects from their usage, and to create groups of similar objects without the need to depend on their concrete classes. This has the advantage of introducing new derived types with no change to the code that leverage the base class type. Using this pattern also introduces the possibility of interchanging the concrete implementations without code modification. An example of the factory pattern can be seen in Listing 2. If the user chooses to add a new VO in the treeview, the GUI instructs the factory to create a new VO of a type indicated by a type ID. A reference to the factory is stored in, and accessed from the Service Locator.

<div align="center">**Listing 2**. Factory Pattern</div>

```
#ifndef COMPONENTFACTORY_H_
#define COMPONENTFACTORY_H_

class ComponentFactory {
public:
        ComponentFactory();
        ~ComponentFactory();

        VisObjectInterface* genObject(const VisTypeEnum& ID) {

                switch(ID) {
                        case VisTypeEnum::surfacevisobject:
                                return new SurfaceVisObject();
                                break;
                        case VisTypeEnum::axisvisobject:
                                return new AxisVisObject();
                                break;
                        default:
                                return new DefaultVisObject();
                }

        }
};

#endif /* COMPONENTFACTORY_H_ */
```
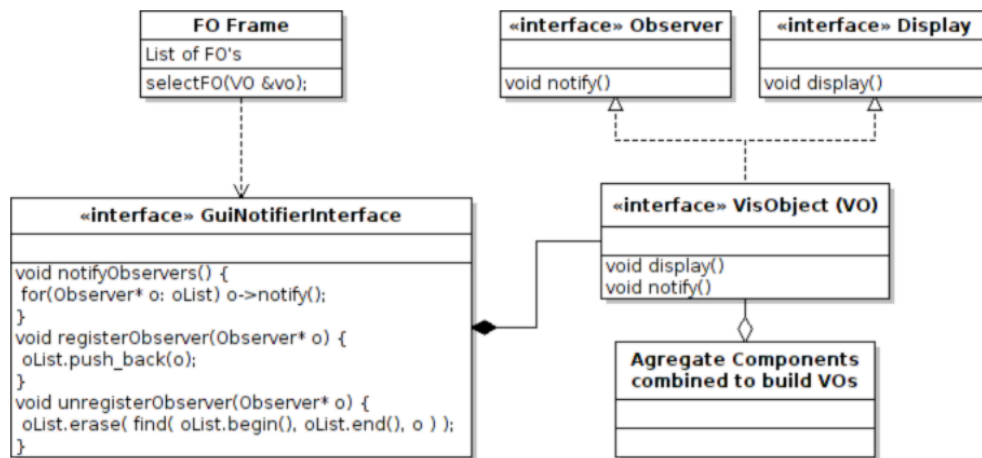
The Observer

The observer design pattern is used to communicate an objects change of state to all dependent objects. These dependent objects are then automatically updated. This pattern controls the communication between classes. An object known as the subject is able to publish a change of state. A set of observer objects which depend on the subject are registered with the subject such that they can then be automatically notified when the state of the subject changes. This design pattern provides a loose coupling between the subject and its observers. The subject has access to a set of observers that are registered at runtime. The observers must inherit an observer interface thus the base class and its functionality is not known to, or required by the subject.

Figure 5 shows the the use case in our software framework. The VO's inherit the observer interface which includes the notify function which must be appropriately overridden in the base class. The VO is registered with the subject, which in our case is the GuiNotifier class. When a frame object (FO) state changes, it calls the notifyObservers function which notifies all registered VO's that the state has

Figure 5. This UML diagram shows the layout of the observer design pattern utilized for communication between the GUI frame objects (FO's) and the visualization objects (VO's).

Figure 6. This UML diagram shows the relationship between the renderer and the visualization objects (VO's). Note the similarity with the observer design pattern shown in Figure 5
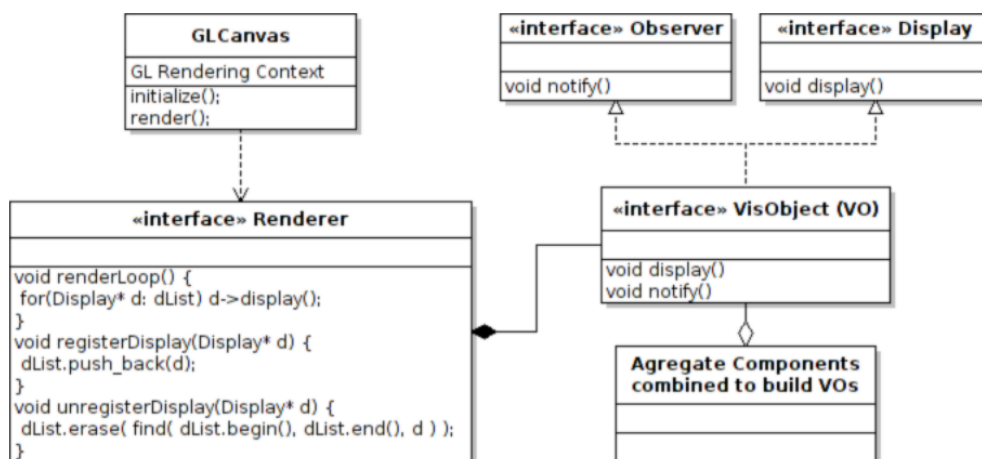

changed. The base class implementation overriding the notify function then checks for changes and updates itself accordingly where applicable.

The Rendering Loop

The GUI center frame renders the current OpenGL rendering context to its canvas for viewing by the user 3. An example of this can be seen in the following illustration 3. In Figure 6 the center frame encapsulates the GLCanvas class. When the GLCanvas class render event is triggered it calls the renderLoop function from the Renderer class.

The renderer class is designed to encapsulate the collection of OpenGL calls required to set the state of the OpenGL state machine. This class also has functionality based on the observer design pattern. VO objects are registered with the renderer, and when the renderLoop function is called by the GLCanvas class, the overridden display function called on the list of registered VO's. The detail of this is shown in the UML diagram of Figure 6.

Figure 7. The Visualization Object (VO) lifecycle.
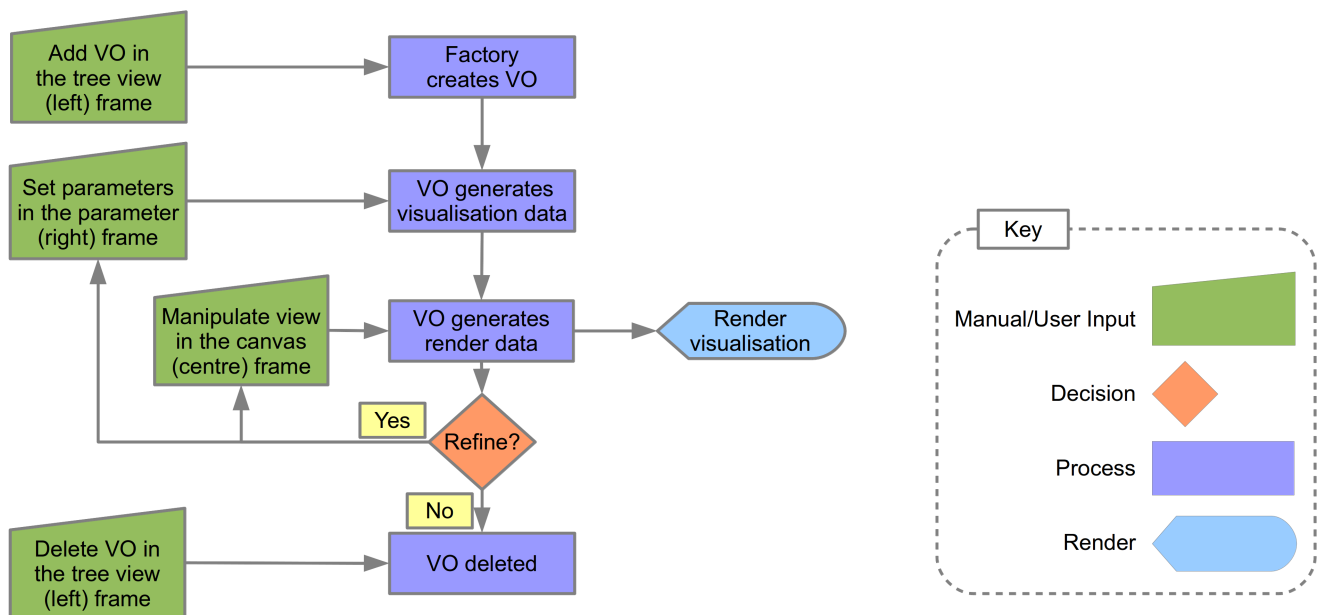
The general state of the scene is set from the renderer's associated frame object (FO). This includes features such as; perspective parameters, light source locations, lighting intensity's e.g. ambient, diffuse, and specular. The type of OpenGL Shader is also selected from this frame object. The renderer class defines how to render the scene data, whereas the VO defines what data is to be rendered.

The Renderer interface is used to provide a general interface for the specific OpenGL implementation. There are differing versions of OpenGL available supported by different specifications of hardware. The advantage of our system is the ability to interchange the Renderer class depending on the hardware support for a particular version of OpenGL. The required version of the Renderer class is registered with the Service Locator. This is also useful for testing revised OpenGL or GLSL [15] implementations.

What is GLSL: GLSL or Graphics Library Shading Language is a high level shading language which uses a syntax close to that of the C programming language. It was created for the purpose of providing developers with more precise control over the built in graphics pipeline. Previously developers would be required to use OpenGL assembly or graphics card specific programming languages to implement modifications to the fixed functionality pipeline.

In our implementation we use a combination of per pixel Phong lighting, edge highlighting using a Gaussian filter to detect sudden changes in depth, and depth peeling for order independent transparency. These algorithms are discussed in more detail in [16], [8], and [17], and in the work by Born et. al. [18] and Hummel et al. [19].

The Visualization Object (VO)

The visualization object (VO) is a key element in the functionality of this software framework. The entire framework is built around the idea of a single object interface type used for all implemented features within the application. In this subsection we describe the lifecycle of a typical VO. We then describe the aggregate style implementation of the VO's and discuss some advantages of this approach

over more traditional methods in the context of a robust visualization framework for prototyping, testing, and comparing new techniques.

The life cycle of a VO is illustrated in Figure 7. The life cycle of the VO starts with its creation as the result of user input. The factory is responsible for the generation of VO's which are then passed to the GuiNotifierInterface detailed in Figure 5. Communication with the VO can then take place. The next step is setting the parameters of the VO required as input to the particular algorithm the VO encapsulates. Once these parameters are set the visualization data is then generated.

During the visualization generation process, data external to the VO may be required. An example of this is the vector and scalar data made available by another VO (the data storage VO) registered with the service locator. This data can be accessed via the service locator interface, along with other data such as color maps etc.

Once the visualization data is generated, in the form of a triangulated mesh, the next step is to generate the render data. The mesh data is usually generated in dual arrays, one representing the vertex, normal, color data, and one representing the node indices. This data is sent directly to the VBO class encapsulating the OpenGL Vertex Buffer Object. This class registers the pointers to the data, then specifies the required rendering parameters needed by the OpenGL state machine to correctly render the mesh.

Once the visualization algorithm is rendered to the center frame, the user is able to interact with the visualization. The user can directly pan, zoom, and rotate the rendered scene by dragging the mouse over the scene in combination with the mouse buttons. The right mouse button is pressed to pan the scene, the left mouse button is pressed to zoom the scene, and both mouse buttons are used to rotate the scene. While interacting with the visualization, the user can also set new parameters for the visualization and generate a modified visualization which is then rendered to screen. See Figure 7.

The final part of the process is deletion of the VO. If the user removes the VO from the left frame 3 by selecting delete from the VO context menu, the VO is then passed back to the factory for deletion ending this VO's lifecycle. All VO objects are created and deleted by the factory. This centralized approach makes for easy debugging of erroneous VO's during the VO lifecycle.

The VO's are constructed by aggregating the set of sub classes needed to compute the required algorithm. The more traditional method of class hierarchies implement features available in a set of classes by extending them and therefore maximizing code reuse. The design of the class hierarchy is made in advance with anticipation of possible additions to the code at a later date. This approach is tried and tested and generally works well.

However, a more recent approach is being more commonly utilized particularly within the games industries; the aggregate design pattern [20] [21]. With the more traditional hierarchical inheritance approach problems can arise in a number of ways. If the insight into the type of additions to the inheritance hierarchy were not well predicted, the code base can easily end up struggling to cope with the required changes leading to major refactoring of the code base. Also with more and more features being added the interwoven complexity of the inheritance hierarchical tree can quickly become unwieldy, and difficult to maintain. These scenarios do not lend themselves to quick and simple implementations of new features.

Figure 8. This UML diagram illustrates a simplified version of a streamline tracer class and a multi threaded streamline tracer class. Rather than inherit the original streamline class and adding the multi threading implementation into the new class, we instead assemble a new class from the original components plus a new thread handling component. If threading was an original intention then the hierarchical inheritance could be designed more efficiently, but now we must refactor the code. With the component based approach foresight into how the software will involve in the future is not required. This is particularly useful in a system which is subject to adding a lot of new functionality.
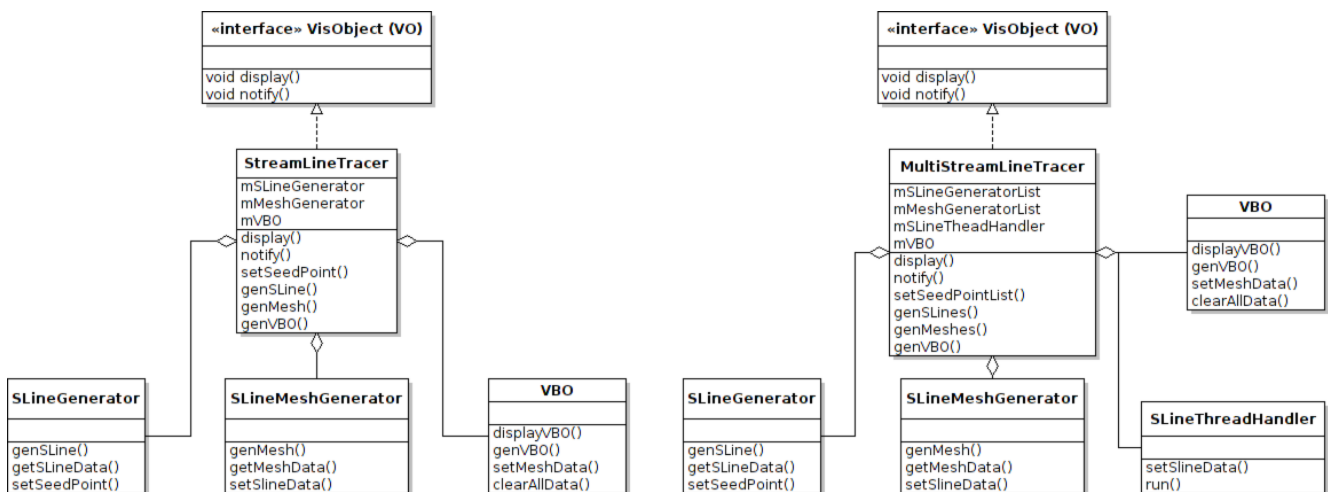

The aggregate design pattern provides an alternative approach which is more applicable to the type of framework discussed in this Paper. We need to be able to add new algorithms quickly and easily but without the issues and complexity associated with large inheritance hierarchies. Another advantage of this approach is the ability to minimize the inherited overhead of unrequired features when extending a class of which only some of the functionality is required.

The parent class is constructed from an aggregation (collection) of a set of components classes. The component classes separate all the functionality required by the system into individual components which are independent of each other. The parent class becomes a wrapper for communication between each of the individual components i.e. the parent class performs no computation or stores any data with the exception of the aggregated component classes, see Figures 8 and 9. Distinct new functionality is implemented in the system by creating a new component. New combinations of component classes are aggregated together to form new base classes providing the required system functionality.

6. Conclusions

This paper presents a comprehensive and versatile cross platform state of the art flow visualization toolkit. This framework has the flexibility to add new algorithms fast and efficiently for testing, evaluation, comparing results, and for study by the domain experts. We focus on describing the design and implementation of a generic visualization framework which provides scientists and engineers with effective solutions for the visualization of CFD simulation data.

Figure 9. This UML diagram illustrates a simplified version of a multi threaded stream ribbon tracer class and a stream surface tracer class. Some components are reused, and new components are added to provide the required functionality. In these illustrations the thread handler used in the stream ribbon tracer class is dropped from the stream surface tracer class, while the time line generator class is added for timeline refinement.
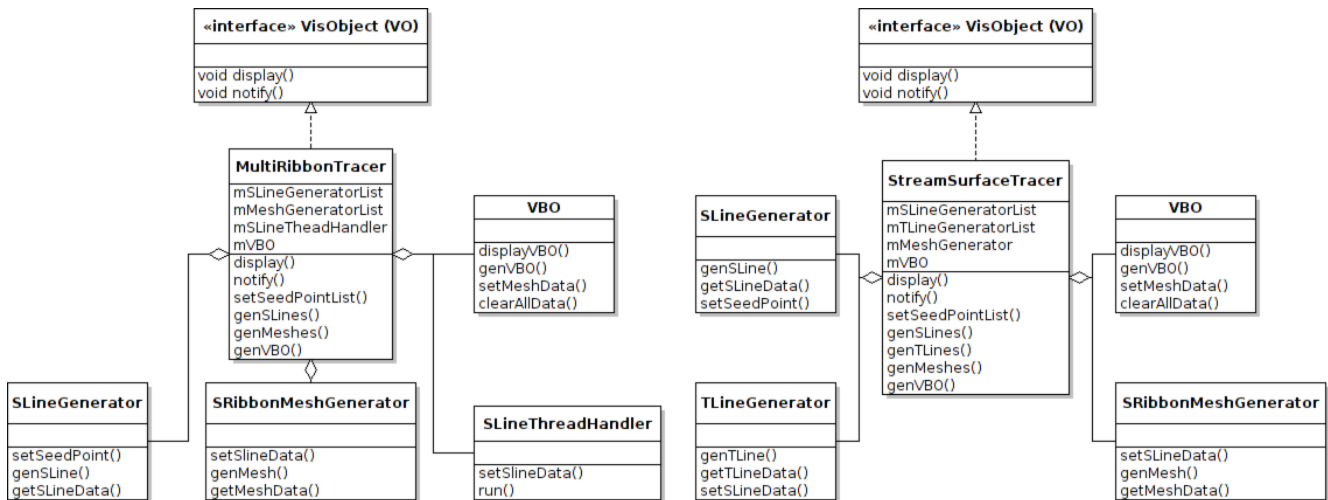
The simple and intuitive layout of the GUI is effective for reducing the software related learning curve by the domain experts. The subdivision of the GUI, Service Locater, and Logic into logical subsystems not only greatly simplifies the system, but also simplifies the communication between the different subsystems, and provides greater flexibility when accessing the raw data. The use of aggregation with the VO's proved very successful and flexible when adding new algorithms and features to the software stack.

Future work for this system includes further abstraction of the VO's and FO's to a plugin based system where new features and algorithms can by added via a plugin interface. This would further simplify implementation and remove the need to modify the main code base. Another area of further work is extending the aggregate pattern of composite components to an XML based approach for constructing the VO's. This combined with a plugin interface for the components would produce a very flexible way to add new features and algorithms to the software.

Acknowledgements

Conflict of Interest

The authors declare no conflict of interest.

References

1. ANSYS UK. Fluent Engineering Simulation. http://www.fluent.co.uk/. Accessed: April 2013.

2. Laramee, R.S.; Hadwiger, M.; Hauser, H. Design and Implementation of Geometric and Texture-Based Flow Visualization Techniques. Proceedings of the 21st Spring Conference on Computer Graphics, 2005, pp. 67–74.

3. Laramee, R.S. Comparing and evaluating computer graphics and visualization software. Software: Practice and Experience(SP&E), Vol 38, NO.7 June 2008 , pp. 735-760

4. Piringer, H.; Tominski, C.; Muigg, P.; Berger, W. A Multi-threading Architecture to Support Interactive Visual Exploration. *Visualization and Computer Graphics, IEEE Transactions on* **2009**, *15*, 1113–1120.

5. Piringer, H.; Berger, W.; Krasser, J. HyperMoVal: Interactive Visual Validation of Regression Models for Real-Time Simulation. Computer Graphics Forum. Wiley Online Library, 2010, Vol. 29, pp. 983–992.

6. McLoughlin, T.; Laramee, R.S., Design and Implementation of Interactive Flow Visualization Techniques. In *Computer Graphics*; InTech Publishers, 2012; chapter 6, pp. 87–110.

7. Peng, Z.; Geng, Z.; Laramee, R.S., Design and Implementation of a System for Interactive, High-Dimensional Vector Field Visualization. In *Horizons in Computer Science Research*; Nova Science Publishers, 2013; Vol. 8, pp. 175–200.

8. Edmunds, M.; Laramee, R.; Malki, R.; Masters, I.; Croft, T.; Chen, G.; Zhang, E. Automatic Stream Surface Seeding: A Feature Centered Approach. *Computer Graphics Forum* **2012**, *31*, 1095–1104.

9. cppreference.com. std::lock. http://en.cppreference.com/w/cpp/thread/lock. Accessed: May 2013.

10. BlackWasp. Service Locator Design Pattern. http://www.blackwasp.co.uk/ServiceLocator.aspx. Accessed: May 2013.

11. Code Project. A Basic Introduction On Service Locator Pattern. http://www.codeproject.com/Articles/18464/A-Basic-Introduction-On-Service-Locator-Pattern. Accessed: May 2013.

12. Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html. Accessed: May 2013.

13. BlackWasp. Null Object Design Pattern. http://www.blackwasp.co.uk/NullObject.aspx. Accessed: May 2013.

14. BlackWasp. Decorator Design Pattern. http://www.blackwasp.co.uk/Decorator.aspx. Accessed: May 2013.

15. OpenGL.org. OpenGL GLSL. http://www.opengl.org/wiki/OpenGL_Shading_Language. Accessed: July 2013.

16. Edmunds, M.; Laramee, R.; Chen, G.; Zhang, E.; Max, N. Advanced, Automatic Stream Surface Seeding and Filtering. Theory and Practice of Computer Graphics, 2012, pp. 53–60.

17. Edmunds, M.; Laramee, R.S.; Evans, B.; Chen, G. Stream Surface Placement for a Land Speed Record Vehicle. Technical report, The Visual and Interactive Computing Group, Computer Science Department, Swansea University, Wales, UK, 2013.

18. Born, S.; Wiebel, A.; Friedrich, J.; Scheuermann, G.; Bartz, D. Illustrative Stream Surfaces. *IEEE Transactions on Visualization and Computer Graphics* **2010**, *16*, 1329–1338.

19. Hummel, M.; Garth, C.; Hamann, B.; Hagen, H.; Joy, K. IRIS: Illustrative Rendering for Integral Surfaces. *IEEE Transactions on Visualization and Computer Graphics* **2010**, *16*, 1319–1328.

20. Nystrom, R.. Game Programming Patterns. http://gameprogrammingpatterns.com/component. html. Accessed: July 2013.

21. West, M.. Refactoring Game Entities with Components. http://cowboyprogramming.com/2007/ 01/05/evolve-your-heirachy. Accessed: July 2013.