Design and Implementation of a Visualization Software Framework

Anonymous cvm submission

Paper ID 0019

Abstract

Research related software often consists of individual isolated prototype applications. Small proof of concept applications are usually enough for demonstrating new algorithms. The unification of new research algorithms into a cohesive software framework has its advantages. Adding new features to an existing pipeline reduces implementation overhead. The researcher is 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. We describe the design and implementation of our feature rich flow visualization software framework, and discuss the effectiveness and scalability of our approach. It is a system we have been developing for four years. Our exposition here goes beyond what a typical research paper is able to present.

1. Introduction

Computational fluid dynamics (CFD) is a rapidly developing tool for fluid analysis. With wide reaching industrial applications such as automotive, aerospace, weather, and medical, it’s an increasingly important tool 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 issue providing visualization methods for the evaluation and correlation of simulation results. Bridging the gap to current visualization software techniques with latest research 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 terms of three stages:

1. Pre Processing: A volumetric or surface mesh is generated to model the physical object. This mesh is generated from computer aided design (CAD) geometry utilizing mesh generation and manipulation tools. Boundary conditions and fluid properties are then 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 complex simulation data, a comprehensive and versatile visualization framework is needed. It is the focus of this paper to describe the design and implementation of a generic visualization framework which provides engineers with effective solutions for the visualization of CFD simulation data. By drawing on recent developments in the game and hardware industry, and by combining several scientific visualization techniques our visualization framework yields following benefits:

- Our framework handles large unstructured real world CFD data fast and efficiently.
- A smooth and efficient threaded user interface, even when processing large data.

Figure 1. The CFD simulation pipeline is comprised of a pre processing stage, the simulation stage, and the post processing stage. This illustration shows the inputs and outputs to each stage, and highlights where in each stage the different tasks reside.
2. Flow Visualization Classification

Flow visualization is one of the classic subsets of data visualization. The techniques in this field of study can be classified into four general categories: direct, geometric, texture-based and feature-based flow visualization [5] [2] [4] [3] [1].

Direct Flow Visualization Techniques: Common approaches visualizing flow include placing arrow glyphs at each sample point to depict the associated vector field and color mapping. These direct techniques are able to make flow visualization universally and intuitively understandable. It has been used in many applications of CFD flow visualization. Direct techniques can suffer from a lack of visual coherence and might suffer from visual complexity and occlusion. See Figure 2 for examples of direct flow visualization.

Geometric Flow Visualization Techniques: A typical example of geometric techniques is the streamline. A trajectories are computed from an initial location within the domain (seeding point) using integration techniques such as the Runge-Kutta integration scheme. The resulting geometric objects from these trajectories are then rendered. Due to its comparatively accurate and coherent result, geometric flow visualization has been widely used to visualize almost all types of CFD data. However, visual clutter and occlusion can stem if a poor seeding strategy is employed in a 3D domain. See Figure 2 for examples of streamlines and stream surfaces.

Texture Flow Visualization Techniques: This approach renders with convolved textures to reflect the local properties of the vector field. Texture based techniques provide a dense and coherent visualization reflecting the local properties of the vector field, even in areas of complex flow. It has been successfully applied to 2D and 2.5D data, but can suffer from visual complexity and occlusion when applied to 3D volumetric flow data. For examples of texture based techniques, refer to Figure 2.

Feature Flow Visualization Techniques: Feature based techniques extract subsets of data which are deemed interesting by user. The visualization is then based on these extracted subsets rather than the whole dataset. This provides a more suggestive and efficient visualization. There is significant computational cost when dealing with the complexity of feature extraction, especially when it is applied to visualize the flow based on 3D unstructured CFD meshes. Refer to Figure 2 for examples of feature based flow visualization.

Although our framework is suitable for implementing all the categories described in this section, our current framework focuses on geometric techniques.
3. System Overview

Our software application system is divided into three main subsystems; the GUI, the Logic, and the Services. See figure 4. We divide the system into smaller easier to maintain subsections 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 specified 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 essentially an interface for features or algorithms within our framework. 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 encapsulate the stream surface algorithm.

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 4, 5, and 6.

4. GUI Subsystem Design

The GUI is divided into three main frames. The center frame for rendering the current list of visualization objects, the left frame for listing the current visualization objects, and the right frame which lists the available options for the currently selected visualization object. Refer to Figure 3.

IMAGE: Subsystem breakdown. e.g. observer pattern list of frame objects. Everything is a vis object with an associated gui frame.

5. Logic Subsystem Design

IMAGE: Workflow of typical algorithm. e.g. get data, seedcurve, surface. IMAGE: Subsystem breakdown. create vis object of algorithm from factory, store in ob...
server/render list, pass data and params to vis object, vis object gens vis data, vis data rendered in render loop.

factory and aggregate design patterns. Everything is a vis object with an associated gui frame.

6. Services Subsystem Design

The Service subsystem includes a data store, a vis object factory, a render object store, a color map store, an interaction store, and a locator for these items.

Some objects or systems in a game tend to get around, visiting almost every corner of the codebase. It’s hard to find a part of the game that won’t need a memory allocator, logging, the file system, or random numbers at some point. Systems like those can be thought of as services that need to be available to the entire game.

This is the Service Locator pattern in a nutshell: it 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).

7. Discussion and Evaluation

8. Conclusions

We have presented the design and implementation of research based software visualization system. We have discussed our design decisions and the associated motivation for those decisions. And although we have focused on flow visualization specific software, we believe the principles outlined here can be applied in a more general way to other similar projects. The result of incorporating research related algorithms into a cohesive scalable software system brings both advantages and disadvantages. Benefits include a rich visualization feature set and robustness while disadvantages include all those tasks inherent in commercial software development such as a steep learning curve and project maintenance.

References


