KVS: A simple and effective framework for scientific visualization

Naohisa Sakamoto* and Koji Koyamada

Institute for Liberal Arts and Sciences, Kyoto University

*naohisa@viz.media.kyoto-u.ac.jp

Received: September 30, 2014; Accepted: February 3, 2015; Published: May 31, 2015

Abstract. In this paper, we propose a visualization development framework that is a C++ class library for easily and efficiently rendering three-dimensional datasets, such as numerical simulation results, medical image datasets, and measurement datasets. Our framework provides a modular programming environment that supports the construction and execution of a visualization pipeline and allows the user to readily implement custom visualization algorithms using our simple module-based execution model. In addition, we also provide a simple implementation of a visualization environment that can handle multiple volumes and semi-transparent polygons in a single scene, which we call a fused visualization environment. Although many visualization software packages have been proposed thus far, such an environment has not previously been supported because of the visibility ordering problem. To confirm the effectiveness of our proposed visualization framework, we demonstrate several visualization applications implemented using this framework.

Keywords: Scientific visualization, C++ class library, Modular programming, Visualization pipeline, Volume rendering, Fused visualization

1. Introduction

With the emphasis on big data analysis in recent years, scientific visualization is playing an increasingly important role and is becoming a crucial technology for many scientists. In a Visualization Science Computing workshop report published by the US National Science Foundation [1], scientific visualization was identified as a key technology for solving problems in various fields of scientific research, and the conceptual model of scientific visualization systems called Visualization Idioms [2] was first introduced. This model is used to manage the input, transformation, display and recording of data and is known as one of the foundations of the concept of a visualization pipeline [3]. Currently, the visualization pipeline
concept serves as a basic architecture for the development of visualization applications and has been adopted in many applications, such as AVS [4], ParaView [5] and VisIt [6], that are used by scientists in physics, biology, astrophysics, and oceanography. For users of visualization applications, the visualization pipeline is a useful and intuitive dataflow architecture for rapid data visualization. Furthermore, for visualization application developers, an application programming interface (API) such as a C++ class library that supports the visualization pipeline is favorable environment for the efficient development of visualization software.

Many visualization APIs that support the visualization pipeline have been released to date. The visualization toolkit (VTK) [7] is one of the most famous visualization software systems and is widely used in many fields. VTK supports many data structures and visualization algorithms for various numerically simulated and measured datasets, and VTK also provides an object-oriented programming environment written in the C++ programming language. In comparison with the use of low-layer graphics APIs, such as OpenGL [8] and CUDA [9], which are implemented based on the C programming language, VTK enables the rapid and facile development of advanced visualization systems. Voreen [10] is an open-source C++ framework for GPU-based volume rendering. Voreen also provides a useful visual programming environment for visualization users. This framework primarily supports structured grid datasets such as medical images and does not support unstructured grid data generated through numerical simulations based on approaches such as the finite element method.

Although these APIs are highly functional for the development of visualization applications, they have not been as commonly used to implement novel visualization algorithms as are low-layer graphics APIs. The implementation of original visualization algorithms using these APIs is sometimes cumbersome. In some cases, it may be necessary to modify the rendering pipeline after gaining an understanding of the rendering mechanism to implement a novel rendering algorithm. Table 1 summarizes the usage frequencies of the graphics APIs that have been used in IEEE Visualization papers over the past half decade. In this table, the category “Unknown/Others” represents a number of papers in which the graphics APIs are not clearly identified or in which other APIs are used to implement the proposed technique. Many papers that are included in this category propose original visualization algorithms based directly on C++, CUDA or low-layer graphics APIs without using a specific visualization API. From these results, it is evident that visualization researchers prefer low-layer graphics APIs, such as OpenGL and CUDA, over visualization APIs, such as VTK and Voreen, for implementing their own original visualization algorithms. However, for application developers, such visualization APIs, in which the visualization pipeline is supported and many basic visualization algorithms are implemented, are suitable for the development of advanced visualization systems and highly practical applications.
Meanwhile, modular visual programming environments, such as AVS, ParaView and VisIt, are also widely used by visualization users as rapid prototyping environments. However, these environments do not provide sufficient support for the advanced demands of visualizing large-scale and complex datasets. Although fused visualization environments are expected to be a key technology for comparative visualization and integrated multi-field visualization, the development of this technology is listed among the current unsolved problems in the field of scientific visualization research [11], and such an environment has yet to be achieved. Many problems with the potential to act as bottlenecks in the development of interactive rendering remain unresolved, especially those related to the fused visualization of multiple-volume datasets, including multivariate datasets with volume rendering and volume datasets with semi-transparent polygons.

To overcome these difficulties, we propose a simple and effective framework that allows for modular programming based on the visualization pipeline and fused visualization using state-of-the-art rendering techniques without visibility ordering for multiple volumes and semi-transparent polygons. The main contributions of this paper are as follows:

- We provide many basic visualization algorithms implemented as modules and a simple execution model of these modules for the effective construction and execution of a visualization pipeline.
- We provide several state-of-the-art rendering techniques implemented as renderer modules and a simple implementation of a fused visualization environment based these renderers.
- We present two example applications: the implementation of a new visualization algorithm and the development of a fused visualization environment in AVS.
- We have released our framework as an open-source C++ class library and have freely provided several documents and example programs at the following web address: https://code.google.com/p/kvs/
2. Methods

We propose a simple and effective visualization framework, which we call the Kyoto Visualization System (KVS). In this section, we first explain the concepts of objects and modules, which represent data for visualization and visualization algorithms, respectively. Next, we describe the visualization pipeline and its construction and execution. After discussing the scene graphs produced and the screen structure provided by KVS, we finally introduce a simple implementation of a fused visualization environment using our state-of-the-art rendering techniques.

2.1. Object

In KVS, the target dataset for visualization is called an ‘object,’ and these objects are classified into geometric objects and volumetric objects. Geometric objects are called ‘geometry objects’ in KVS and include points, lines and polygons, whereas volumetric objects are called ‘volume objects’ and include both structured and unstructured volumes (Figure 1). Each object is represented as a data container class in KVS and is defined by inheriting a base class. The base class contains xform information that includes scalings, translations and rotations that can be controlled via mouse interaction.

![Figure 1: Object types supported in KVS.](image)

- **Geometry objects**

In KVS, a geometry object is an object that is either a point object, a line object, or a polygon object (Figure 2). A point object is an object that is represented as a set of points, with coordinates and colors for each vertex. A line object is an object that is represented as a set of line segments and contains data related to the connections between vertices in addition to the data included in a point object. A polygon object is an object that is represented as a set of triangular faces and includes the normal vectors for each vertex or face in addition to the data included in a line object.

![Figure 2: Geometry objects.](image)
In KVS, the data that describe objects, such as the vertex coordinates and colors, the connections and the normal vectors, are represented in an array structure with an arbitrary number of elements in a shared memory region. Based on this data structure, KVS provides two types of copy methods for objects, called shallow copy and deep copy, as follows:

```java
// Shallow copy from object1 to object2
Object object2 = object1;

// Deep copy from object1 to object3
Object object3 = object1.clone();
```

By using the operator `=` on an object, the user can copy it to another object of the same object type as a shallow copy operation without any memory allocation. By contrast, the `clone()` method of the object returns the same object actually copied over. In this case, the copied object does not depend on the original object in memory. The user can copy an object to another object as a deep copy operation by using `clone()` and the `=` operator. These two types of copy methods allow for the simple reduction of the duplicated memory region for the target object and the efficient execution of the visualization pipeline.

**Volume object**

A volume object is an object composed of nodes and cells. A node is a vertex of a cell, and a physical quantity, which is represented as a scalar, vector or tensor, is defined on each node. In KVS, based on the cell shape, volume objects are divided into two types, known as structured volume objects and unstructured volume objects. A structured volume object is an object that is represented as a set of hexagonal cells and has a grid structure that defines the alignment of the nodes. There are three types of grid structure available for structured volume objects: uniform, rectilinear and curvilinear (Figure 3).

By contrast, an unstructured volume object is represented as a set of arbitrarily shaped cells. Currently, KVS supports the following cell types: tetrahedral, quadratic tetrahedral, hexahedral, quadratic hexahedral, pyramid, and prism cells (Figure 4). An unstructured volume object also contains cell connectivity data for each cell in addition to the coordinates and physical quantities defined for each node. These data are also represented in an array structure, as in the case of geometry objects. Therefore, the operator `=` and the clone method...
are also available for copying volume objects in either shallow or deep copy operations.

![Unstructured volume object]

Figure 4: Unstructured volume object.

2.2. Modules

In KVS, each visualization method is implemented as a module, and these methods are divided into four types of processes: importers, filters, mappers and renderers.

- Importers
  An importer module represents the first process that is performed to import the input data into the visualization process (Figure 5). In general, the target data are often a dataset that is output from numerical simulations or consists of measurements that may originate from any of a disparate variety of scientific domains. Therefore, to integrally process these data using a given visualization pipeline, it is necessary to map them to a dataset that can be handled using KVS. In the current implementation, importer modules that can read a data file and map it to the supported objects described in the previous section are provided.

![Importer module]

Figure 5: Importer module.

- Filters
  A filter module represents a data transformation process, such as clipping, denoising, resampling, interpolation or segmentation (Figure 6). Applying such a filtering process may allow the user to focus on important or interesting regions. In such a process, an object transformation of either the ‘geometry-to-geometry’ or ‘volume-to-volume’ type is executed. For example, the clipping filter module provided in KVS outputs a clipped volume object based on the region of interest that is specified for the input volume object.

![Filter module]

Figure 6: Filter module.

- Mappers
A mapper module represents a process for converting data into graphical primitives that can be directly drawn in a subsequent rendering process (Figure 7). In the mapping process, color and opacity values will be assigned to a volume object based on physical quantities through the application of a transfer function. In this process, an object transformation of the ‘volume-to-geometry’ type is performed. For example, the isosurface mapper module provided in KVS outputs an extracted surface polygon based on the isovalue specified for the input volume object.

![Mapper module](image)

**Figure 7: Mapper module.**

*Renderers*

A renderer module represents the final process that is performed to draw graphical primitives on the image plane. In the rendering process, image data are generated and drawn into a frame buffer on a GPU (Figure 8). Although a ‘geometry-to-image’ transformation can be executed using the standard APIs provided in OpenGL, it is necessary to provide a renderer for transformations of the ‘volume-to-image’ type; such transformations are typically referred to as direct volume rendering. In KVS, we provide several volume renderer modules for both structured and unstructured volume objects. For example, in the hardware-accelerated ray-casting renderer module that is provided as a renderer class of KVS, an image is directly generated from an input structured volume object.

![Renderer module](image)

**Figure 8: Renderer module.**

### 2.3 Visualization pipeline

Each module has an input and an output port for an object. The ports of each module can be connected interchangeably, thus allowing the user to easily build a visualization pipeline using KVS (Figure 9).

![Visualization pipeline](image)

**Figure 9: Visualization pipeline.**

*Execution model*

Each module has an execution method named ‘exec,’ which takes an input object as an argument value and returns an output object as a return value. The module can be executed by
calling the exec method as follows:

```java
Module module = CreateModule();
Object input = GetInputObject();
Object output = module.exec(input);
```

To implement an original visualization technique based on KVS, the user simply needs to override the exec method inherited from the base class provided for each type of module (Figure 10).

![Figure 10: Implementation of an original module.](image)

- **Pipeline execution**
A visualization pipeline embodies a data flow network by connecting executable modules in a directed graph. The ability to manage the module execution on the graph is also important for the development of effective visualization applications. In KVS, we provide a simple implementation of the visualization pipeline execution, which can be separated from the construction of the visualization pipeline. An example of the execution is provided as follows:

```java
function Execute(Object object, ModuleList modules)
    for each module in modules
        object = module.exec(object);
    end for
    return object;
end function
```

In this pseudocode, an input object named `object` and a module list named `modules` are specified as arguments of `Execute()`. This function sequentially executes each module in the module list, which can be regarded as a visualization pipeline, and returns the object that passed through the pipeline.

### 2.4. Scene graph
A scene graph is a structure that is employed to represent a graphical scene. To illustrate the construction of a scene graph, we provide a simple example of a scene graph structure composed of objects, a camera and a light (Figure 11). The objects that are transformed through
the visualization pipeline are managed as a tree structure in the scene graph. The camera and light are also managed as nodes of the scene graph and possess xform information for user interaction as does an object. Although the renderer is implemented as a module, there is a renderer assigned to each object, which is managed in the scene graph with the object.

Figure 11: Scene graph in KVS.

The scene represented in the figure, which is composed of several objects with renderers that are represented by an object tree, a camera and a light, can be drawn in the rendering process as follows:

```plaintext
for each frame
    UpdateXform( camera );
    UpdateXform( light );
    for each object in objects
        UpdateXform( object );
        Renderer renderer = GetRenderer( object );
        renderer.exec( object, camera, light );
    end for
end for
```

where objects represents the object tree; UpdateXform() is a function that updates the xform information of the specified object, camera, or light; and GetRenderer() returns the renderer assigned to the specified object. Through the execution of all renderers, all registered objects in the scene are drawn.

2.5. Screen

Screen is a class for displaying a window on a desktop and depends on the operating system (OS). In KVS, we can use GLUT [12] and Qt [13], which are OS-independent window APIs, to display the rendering results on a desktop. Although the event types are the same between GLUT and Qt, the event handling styles are different because GLUT is implemented using C and Qt is implemented using C++, which is an object-oriented programming environment. To overcome this difficulty, we provide integrated APIs for screen display and event handling that are supported on both GLUT and Qt.
The screen class also has a scene graph for managing scene components, such as objects, cameras, and lights. By using `registerObject`, the user can register an object with a renderer to the scene via the screen class, and the object can then be automatically managed as a component of the scene graph.

```java
Screen screen;
screen.registerObject( object1, renderer1 );
screen.registerObject( object1, object2, renderer2 );
```

where `object1` is registered as a child of the root of the scene graph represented in Figure 11 and `object2` is registered as a child of `object1`.

### 2.6. Fused visualization

An effective rendering technique for fused visualization that is known as particle-based rendering (PBR) [14] has been developed. Based on a stochastic approach, PBR can integrally treat volumes and polygons without visibility sorting. This technique calculates the brightness as the expected value of the luminosity from a ray segment and can improve the accuracy of the volume rendering, which is integral to the repetition process. This process is implemented based on the ensemble averaging of the brightness values. The rendering framework of a scene for fused visualization using PBR can be described as follows:

```java
FrameBufferObject fbo;
for each frame
    UpdateXform( camera );
    UpdateXform( light );
    for each repetition in ensemble averaging
        fbo.bind();
        for each object in objects
            UpdateXform( object );
            Renderer renderer = GetRenderer( object );
            renderer.exec( object, camera, light );
        end for
        fbo.unbind();
```
In the pseudocode above, a framebuffer object named \texttt{fbo} is used for multipass rendering in the process of fused visualization using PBR. \texttt{fbo} is enabled or disabled using \texttt{bind} and \texttt{unbind}. Then, the rendered framebuffer data are accumulated into \texttt{fbo} using \texttt{add}. Finally, the accumulated data in \texttt{fbo} are ensemble averaged and then drawn using \texttt{average} and \texttt{draw}, respectively.

In KVS, the framework described above is implemented as a paint event named compositor. KVS also provides several renderers that can support particle-based rendering for each object. Users can easily apply the fused visualization mechanism by replacing the default paint event defined in the scene with the compositor.

```c
Compositor compositor;
compositor.setNumberOfRepetitions( n );

Screen screen = CreateScreen();
screen.setEvent( compositor );
screen.registerObject( object1, renderer1 );
screen.registerObject( object2, renderer2 );
```

where \(n\) is the number of repetitions used for ensemble averaging in the PBR process. Through modification of this value, the rendering quality and speed can be controlled.

3. Results

In this section, we describe several example implementations of visualization programs to illustrate the effectiveness of KVS. We also demonstrate two example applications: the implementation of a new visualization algorithm and the construction of a fused visualization environment in AVS. The performance evaluations were conducted on a PC equipped with an Intel Core i7 (2.9 GHz) CPU, 8 GB of RAM and an Intel HD Graphics 4000 (1024 GB) graphics card.

3.1. Module execution

- Isosurface extraction

Isosurface extraction [15] is one of most popular methods of visualization. To extract an isosurface as a polygon object from a volume object, KVS provides a mapper module class named \texttt{kvs::Isosurface}. The user can set up a class to be used for isosurface extraction as follows:

```c
kvs::Isosurface* isosurface = new kvs::Isosurface();
isosurface->setIsolevel( isolevel );
```
where `isolevel` is a scalar value that is used for extracting the surfaces and `tfunc` is a transfer function consisting of a color table and an opacity table. The isosurface is then extracted as a polygon object named `polygon` from a volume object named `volume` as follows:

```cpp
kvs::PolygonObject* polygon = isosurface->exec(volume);
```

Figure 13 shows the rendering results for a polygon object extracted from a turbulent combustion simulation dataset represented as a structured volume object with a resolution of 480x720x120. The isolevel for isosurface extraction can be modified using the `setIsolevel` method. In addition, for comparison, we present the rendering image and execution time results of isosurface extraction using KVS and VTK [7] in Figure 14. In VTK, isosurfaces can be extracted using the `vtkContourFilter` and `vtkPolyDataMapper` classes. From the results, it is evident that the processing cost of KVS for isosurface extraction is 1.3x higher than that of VTK, although the image quality is almost the same.

![Figure 13: Rendering results for the isosurfaces of a turbulent combustion simulation dataset.](image)

![Figure 14: Results of isosurface extraction using KVS and VTK.](image)

- **Volume rendering**

  Volume rendering is a technique for directly generating images from a 3D volume dataset and is widely used in various fields, such as medicine and engineering [16]. Many volume rendering techniques for volume datasets of these types have been proposed to date. Most are based on a density emitter model proposed by Sabella and can be categorized into slice-based approaches and ray-casting approaches. KVS provides a hardware-accelerated ray-casting
rendering class named `kvs::glsl::RayCastingRenderer` for structured volume objects [17]. A transfer function can be directly specified to the renderer using `setTransferFunction`. An example of the code for renderer setup is provided below:

```cpp
kvs::TransferFunction tfunc;
kvs::glsl::RayCastingRenderer renderer =
    new kvs::glsl::RayCastingRenderer();
renderer->setTransferFunction( tfunc );
```

Then, the instanced renderer can be registered with the target structured volume object, named `object`, to the screen as follows:

```cpp
screen.registerObject( object, renderer );
```

Figure 15 shows the rendering results for the turbulent combustion simulation dataset used for isosurface extraction in the previous example. The user can focus on a region of interest by interactively modifying the transfer function using the `setTransferFunction` method of the renderer. In addition, we compare the volume rendering results obtained using KVS and VTK [7] in terms of the image quality and the rendering speed in frames per second (fps) in Figure 16. For volume rendering in VTK, we use a GPU-accelerated volume renderer class named `vtkGPUVolumeRayCastingMapper`. From the results, it is evident that KVS performs 1.7x faster than VTK while producing the same image quality.

![Volume rendering results for the turbulent combustion simulation dataset.](image)

Figure 15: Volume rendering results for the turbulent combustion simulation dataset.
Visualization pipeline

Modules for the construction of a visualization pipeline can be set up using the pipeline module class, which is named `kvs::PipelineModule`. The following code fragments are setup examples for a isosurface mapper module and a polygon renderer module. The user can specify parameters to each module using the set methods of its class.

```cpp
kvs::PipelineModule<kvs::Isosurface> mapper;
mapper()->setIsolevel( isolevel );
mapper()->setTransferFunction( tfunc );

kvs::PipelineModule<kvs::PolygonRenderer> renderer;
renderer()->enableShading();
```

In KVS, the `kvs::VisualizationPipeline` class is provided. Each pipeline module can be connected using the connect method of the visualization pipeline class. Once the pipeline has been constructed and connected to the modules, the user can execute the pipeline using the `exec` method of the visualization pipeline class as a trigger. This mechanism allows for lazy evaluation of the visualization pipeline because the module connections and the pipeline execution can be separately processed in the source code.

```cpp
std::string filename("dataset.dat");
kvs::VisualizationPipeline pipeline( filename );
pipeline.connect( mapper ).connect( renderer );
pipeline.exec();
```

3.2. Fused Visualization

We implement a fused visualization environment [18], which can handle multiple semi-transparent geometry objects and volume objects in a single scene. KVS provides several renderers [14,18] for each object as follows:

- For point objects:
  `kvs::glsl::ParticleBasedRenderer`
  `kvs::StochasticPointRenderer`

- For line and polygon objects:
  `kvs::StochasticLineRenderer`
  `kvs::StochasticPolygonRenderer`

- For structured volume objects (with a uniform grid):
  `kvs::StochasticUniformGridRenderer`
Using these renderers, the user can visualize multiple objects with renderers in a single scene. The following code provides an example of the fused visualization of an unstructured volume object and the polygon object representing its external faces with the semi-transparent attribute extracted using a mapper module, e.g., kvs::ExternalFaces, which is provided in KVS.

```cpp
ekvs::ExternalFaces* mapper = new kvs::ExternalFaces();
kvs::PolygonObject* polygon = mapper->exec(volume);
polygon->setOpacity(0.5);

kvs::StochasticPolygonRenderer* renderer1 =
new kvs::StochasticPolygonRenderer();
kvs::StochasticTetrahedraRenderer* renderer2 =
new kvs::StochasticTetrahedraRenderer();

screen.registerObject( polygon, renderer1 );
screen.registerObject( volume, renderer2 );
```

To apply a multi-pass rendering framework for PBR, it is necessary to create a compositor class named kvs::StochasticRenderingCompositor and set it to the screen as a paint event. The number of repetitions n, which is equivalent to a number of ensembles, can be specified to the compositor using setNumberOfRepetitions().

```cpp
kvs::StochasticRenderingCompositor compositor;
compositor.setNumberOfRepetitions( n );
screen.setEvent( &compositor );
```

Figure 17 shows the rendering result of a computational structural mechanics computation of a V6 engine. This dataset is represented as a multivariate unstructured volume object composed of 282K tetrahedral cells. We visualize the distribution of the magnitudes of the displacement values and the von Mises stress values as semi-transparent boundary surfaces represented as a polygon object. We use kvs::StochasticTetrahedraRenderer for the volume object and kvs::StochasticPolygonRenderer for the polygon object.
Figure 17: Fused visualization result for a V6 engine.

Figure 18 shows the rendering results for a numerical simulation of a rotating sphere inside a high-speed unidirectional air flow. This dataset is represented as a time-varying unstructured volume object composed of 28M tetrahedral cells and 20M prismatic cells. Using \texttt{kvs::glsl::ParticleBasedRenderer}, the user can simultaneously and interactively visualize this entire dataset in a single scene. The particles used for this renderer are generated before the rendering process using a particle generation class provided in KVS, such as \texttt{kvs::CellByCellMetropolisSampling} class, and animation rendering can be performed using a timer-event function provided in KVS. The average rendering speed for this dataset was found to be 5.9 fps.

![Figure 18: Time-varying visualization results (t: time step).](image)

### 3.3. Application examples

KVS has previously been used in several visualization projects. Here, we describe two visualization applications implemented using KVS.

- **Particle-based surface rendering**
  Particle-based surface rendering (PBSR) [19] is a technique developed by Hasegawa and Tanaka. This technique can stochastically and uniformly generate particles on each triangular patch of a polygon object based on its opacity. The authors have also extended this technique to the visualization of a point cloud dataset represented as a point object [20].

  Figure 19 shows the rendering results obtained using PBSR for medical image datasets and a laser-scanned dataset. For PBSR in KVS, the particle generation techniques are implemented as a mapper module and the generated particles are represented by a point object. Thus, the user can easily and rapidly visualize the generated particles using the particle-based renderer provided in KVS.
Fused visualization environment in AVS

We are constructing a fused visualization environment in AVS/Express in cooperation with Cybernet Inc. Our rendering environment based on KVS is integrated as one of the rendering engines implemented in AVS/Express. Therefore, AVS users can use our fused visualization functions in the traditional manner, without any changes to the interfaces of their applications. Moreover, users can construct their visualization pipelines interactively using a visual programming editor called a network editor by using AVS modules and user interfaces.

Figure 20 depicts an example of the fused visualization environment in AVS. The rendering result for the V6 engine dataset represented as an unstructured volume object and its boundary surfaces with the semi-transparent attribute are shown on the rendering screen.

To evaluate our visualization system, we conducted questionnaire studies among the users who used KVS to develop the aforementioned visualization applications. We received the following comments from two respondents:

Pros:

– A visualization application using state-of-the-art techniques can be easily developed without the need for higher skills of computer graphics programming.
– Because visualization algorithms are implemented as modules in the visualization pipeline, it is easy to add our original filter and mapper modules into the pipeline.
– It is possible to efficiently visualize our datasets by connecting our original modules to the existing modules in KVS.
Through only the integration of the particle datasets, fused visualization can be performed using the particle-based renderers in KVS.

Cons:

- It is difficult to extend an existing module through class inheritance.
- Modification of the data in an object is sometimes cumbersome because of the strong encapsulation of the data in the class to ensure the safety of the data.
- It is somewhat troublesome to integrate and share the data of an object in KVS with our application because the method of data management is different.

4. Discussion

Although the modules implemented in KVS have a simple execution mechanism in which only an execution method, ‘exec,’ is implemented, it is necessary for the user to manage the input/output object resources of each module. Therefore, there is the possibility of memory leaks, even though the data that are stored in objects, such as coordinate values, can be shared among objects through the shallow copy mechanism. In essence, the framework must manage data resources and schedule the execution of modules in the visualization pipeline. In the previous section, we presented an example implementation of a visualization pipeline in KVS, in which, although lazy evaluation can be performed, object resources cannot be flexibly managed. We believe that the visualization pipeline can be extended to this type of flexible pipeline mechanism based on our simple execution model in KVS.

In this paper, we presented several examples focused on two typical visualization algorithms, isosurface extraction and volume rendering. In addition, many other basic algorithms, such as contour slices, streamlines, line integral convolution, and glyphs, are implemented in KVS. The ability to create a fused visualization environment that can efficiently handle multiple objects is one of the key characteristics of our framework because this capability has not been provided in previous visualization APIs or applications. In particular, we believe that a mechanism for the facile construction of a fused visualization environment through the addition of a compositor to the screen, without depending on module and pipeline execution, is a considerable advantages in the development of visualization applications. From the experimental results, we can confirm that not only a multivariate unstructured volume object with a semi-transparent polygon object describing its surface but also a time-varying unstructured volume object composed of several types of cells can be efficiently visualized without any prominent artifacts.

In addition, several visualization applications based on KVS were demonstrated in this paper. From the results obtained in the questionnaire survey regarding KVS, we confirmed that our modular programming architecture is useful and effective for the development of custom user applications. We also received comments that the extension of existing modules and the modification of object data are difficult. In the current implementation of KVS, the safety of each class and the robustness with respect to modifications of the class are improved by encapsulating the data. However, the resulting lack of flexibility or extendability can sometimes pose a problem in application development. Although it is generally difficult to balance safety and flexibility, we will investigate the possibility of improving the extendabil-
ity of the existing classes by introducing state-of-the-art software design patterns for scientific visualization.

5. Conclusion

In this paper, we proposed a simple and effective framework that allows for module-based programming based on the visualization pipeline concept as well as fused visualization using new rendering technologies without visibility ordering for multiple volumes and polygons. We also demonstrated examples of applications developed using KVS. In future work, we intend to apply our framework for large-scale and time-varying visualization by extending the fused visualization environment and improving the visualization pipeline for implementation in a parallel environment. We will also investigate a new interaction framework that is suitable for the development of visual analytics applications and introduce it into KVS.

Acknowledgment

This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT) through a Grant-in-Aid for Research Programs on Climate Change Adaptation (RECCA) and by the Japan Science and Technology Agency (JST) through the A-STEP project ("The research and development of fusion visualization technology," AS2415031H). We thank Prof. Satoshi Tanaka and Dr. Kyoko Hasegawa at Ritsumeikan University for providing the rendering images for particle-based surface rendering and Dr. Hideo Miyachi and Mr. Kenichiro Uchida at Cybernet Inc. for providing the images of the fused visualization environment in AVS/Express.

References


