Distributed Visualization Framework Architecture

Oleg Mishchenko, Sundaresan Raman and Roger Crawfis, *Member, IEEE*

Abstract—An architecture for distributed and collaborative visualization is presented. The design goals of the system are to create a lightweight, easy to use and extensible framework for reasearch in scientific visualization. The system provides both single user and collaborative distributed environment. System architecture employs a client-server model. Visualization projects can be synchronously accessed and modified from different client machines. We present a set of visualization use cases that illustrate the flexibility of our system. The framework provides a rich set of reusable components for creating new applications. These components make heavy use of leading design patterns. All components are based on the functionality of a small set of interfaces. This allows new components to be integrated seamlessly with little to no effort. All user input and higher-level control functionality interface with proxy objects supporting a concrete implementation of these interfaces. These light-weight objects can be easily streamed across the web and even integrated with smart clients running on a user's cell phone. The back-end is supported by concrete implementations wherever needed (for instance for rendering). A middle-tier manages any communication and synchronization with the proxy objects. In addition to the data components, we have developed several first-class GUI components for visualization. These include a layer compositor editor, a programmable shader editor, a material editor and various drawable editors. These GUI components interact strictly with the interfaces. Access to the various entities in the system is provided by an AssetManager. The asset manager keeps track of all of the registered proxies and responds to queries on the overall system. This allows all user components to be populated automatically. Hence if a new component is added that supports the IMaterial interface, any instances of this can be used in the various GUI components that work with this interface. One of the main features is an interactive shader designer. This allows rapid prototyping of new visualization renderings that are shader-based and greatly accelerates the development and debug cycle.

Index Terms— Distributed Systems, collaborative visualization, Visualization, interaction, volume rendering.

1 Introduction

We present a distributed visualization framework. The goal in creating the framework was two-fold: first, create a system that can reduce the time for implementing new visualization algorithms and thus reduce the time for verifying research ideas, and second, make sure that the system is easily extensible. In brief, we tried to create a system that can be used as a test bench for research in visualization.

Our open source framework can be used for creating both single user and collaborative visualization applications. To demonstrate the potential of the system, we have developed several applications using this system, such as the Volume Compositor. Volume Compositor is a volume rendering application providing shader program support, layer support and a high quality user interface. In volume rendering and scientific visualization, the majority of the algorithm implementations are using new hardware capabilities for maximum performance. Shader programs are used to employ programming capabilities of modern graphics hardware. That led us to including a shader editor to the Volume Compositor's user interface. This allows faster shader development for shader-based algorithm implementations. Instead of recompiling and executing a program, the developer can see the results of modification immediately as s/he commits the changes in the shader editor.

Another goal was to make the design flexible, both for developing standalone as well as distributed applications and algorithms. We decided to split the system into a set of components that allow the system to be distributed and allow easy code maintenance. Major criterion in splitting was to keep the components loosely coupled. Compared to monolithic systems, separate components may require a large number of interconnections between them and thus make the system overly complex, directly violating our design goal. By keeping coupling low, we achieve easier system modification and make the system distributed.

2 Related Work

There are different dimensions in collaborative visualization. Roughly collaborative visualization systems can be subdivided by several criteria:

Type of visualization:
- synchronous
- asynchronous

Kind of display systems that are utilized:
- Standard displays
- Large Displays
- Immersive systems
- Handheld devices

Location:
- Same location
- Different locations

We should emphasize that distributed visualization does not automatically mean collaborative. Making the system distributed in many cases only implies improving the performance by utilizing resources of more than a single machine. For a detailed coverage of the field we suggest the following excellent review by Brodlie et al. [3].

Historically, a number of well-known visualization frameworks have been made distributed. Usually, as in case with AVS [19] this includes execution of some parts of the system on remote machines. Later on, support for collaboration has been added. VTK was extended to provide support for collaboration in CAVE-type environments [4][11]. IBM Data Explorer [1] and later OpenDX[13], supports execution of some of its modules on the remote machines. Collaborative capabilities were added to Iris Explorer [7] with modules from the COVISA [21] project. Another well-known visualization system with support for collaborative and distributed visualization is VisTrails [2].

Recently, research in developing web-based collaboration tools includes Many Eyes [20] and Swivel [17]. They employ an asynchronous collaboration model. Typical collaboration sessions consist of creating/modifying visualization, uploading them to the website such that later users can modify the visualization, or discuss it and leave comments and annotations. Other systems, such as [22] are utilizing new web2.0 technologies (for example, AJAX [8]) without altering the interactive visualization paradigm.

The term grid computing has become increasingly popular recently. It refers to the use of multiple interconnected computers for computation as opposed to performing the same tasks on a single supercomputer. The gViz project [5] is aimed towards adding grid-enabled collaboration support to Iris Explorer and pV3 [14]. OptIPuter [18] provides collaboration for multi-gigabyte earth science datasets. RAVE [9] is another grid based visualization system.

Our system can be described primarily as a synchronous distributed collaboration one; however, we also show how we can employ asynchronous scenarios for our system.

The rest of the paper is organized as follows: in section 3 we describe system architecture, in section 4 we talk about Volume Compositor, application built on top of the framework. Section 5 covers additional use cases for the framework. Section 6 covers details on incorporating VTK plug-ins and Python scripts into the system. Section 7 describes layers, and section 8 provides additional implementation details.

3 System Architecture

The system consists of three kinds of concrete entities, a light-weight object definition (proxies), communication and synchronization mediators, and implementation objects, such as OpenGL drawables. Figure 1 shows a rough view of the interrelationships between these entities.

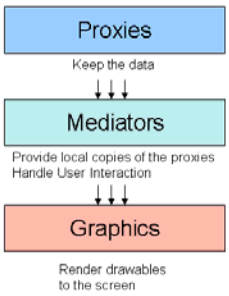


Figure 1. Main entity types in the system.

The proxy objects are shared among different machines, different software architecture configurations and with the user-interface components. Any entity that the developer/researcher would like to share is a proxy, including cameras, drawables, transfer functions and lights. As such, we call these entities Assets. Distribution of the proxies is done with the aid of an AssetManager class which is an object pool for all the shared proxies in the system. If a particular remote machine needs an instance of a proxy, it gets the (remote) reference from the AssetManager. Synchronization, in case of object modification (for example, changing camera parameters), is done by subscribing to change events associated with each proxy. In this regard, each proxy object can be viewed as a publisher to which interested parties can subscribe.

The Mediator entities provide a clean separation of concerns from the underlying graphical elements and the communication and synchronization with the proxy objects. They reside on the remote machines and each remote client will have its own instance. This differs from the proxy instances which (logically) are global across all machines. Mediators are implemented using a decorator pattern, where each graphic entity is wrapped with a mediator and mediators communicate among themselves to determine the amount of synchronization that is needed. Synchronization with the proxy objects is done lazily.

The graphics entities, like the mediator entities, reside on each remote machine. Classes at this level deal directly with graphics library, for example VTK, OpenGL or DirectX by providing the necessary graphics library calls.

The architecture provides a clear separation of responsibilities between different parts of the system, similar to a model-view-controller (MVC) architecture – with proxies being the model, mediators being the controller, and the graphics tier being the view.

4 Volume Compositor

Using the framework mentioned in the previous section, we developed the Volume Compositor, a multilayer volume rendering application. It contains a 3D rendering and a 2.5D compositing engine, flexible user interface, and distribution capabilities. By referring to the compositing engine as 2.5D we emphasize the use of layers and layer compositing. Such use can be thought of as an extension to the concept of compositing, from the general meaning associated with Porter and Duff composing operators, to a more general and more flexible set of functions.

The concept of layer is central to this application. By using the term *layer*, we refer to a 2D container for storing an image, on which a general compositing operator is defined. All drawables are rendered to one or more layers. The composition order can be changed and the layers can be exported and imported in a format readily utilized in a 2D paint package. A number of tasks for volume exploration are performed by processing the layers instead of applying more expensive 3D rendering techniques, thus combining the aspects of image-based rendering. Examples include two level volume rendering, masking and magnification.

4.1 User Interface

In this section, we describe Volume Compositor's user interface. Before proceeding, a quick overview of visualization workflow in the system is necessary. Visualization session usually consists of the following steps. First, the user loads the datasets s/he is interested in. Next, the appropriate drawables (geometric primitives) are created – these could be boxes or planes. Then, region(s) of interest (ROI) are selected; in most cases ROI is a rectangular box. Then, the user creates a set of layers. A drawable can be rendered to multiple layers. Finally, the user specifies which material is applied to which of the drawables and layers. The complexity of the process is minimized by a number of predefined templates that the user can select from. The user interface gives full control of all the elements described above.

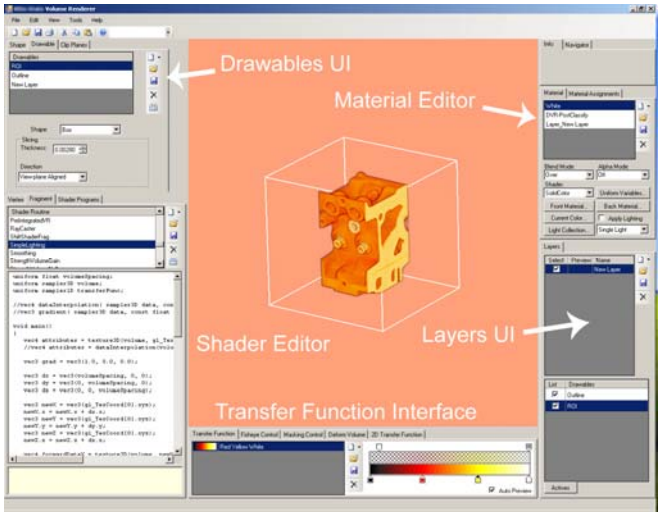


Figure 2. Volume Compositor interface screenshot.

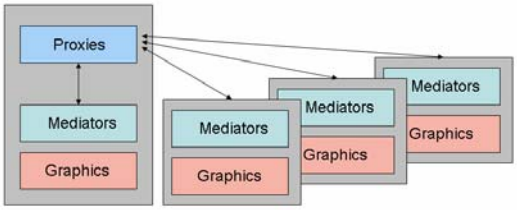
Anything that can be rendered in our system is a drawable. Currently, we have the following types of drawables: volumetric regions, boundaries, outlines, and planes. Depending on the selected geometric primitive, the user can modify the size and orientation of a drawable. All the above transformations can be done both via

changing the properties in the corresponding properties tab, or by direct manipulation with mouse and keyboard.

Each drawable has a Material property. Material specifies how the drawable is rendered. Material has a shader or a color and blending properties. We have implemented a set of predefined materials and corresponding shaders in our system. Material editor is used for material creation and editing. Material assignments are done with the Drawable UI component. Material assignment sets up the correspondence between the materials and drawables, and materials and shaders. A material can be shared across any number of drawables. Material editor also allows the user to assign uniform variables for the shaders.

The shader editor allows the user to modify and compile shaders on the fly. The results of the modification of a shader program are presented right after the user commits any changes. This makes shader development interactive, thus greatly increasing developer's productivity. The key benefit of adding shader control to the system is that it breaks the standard rewrite/recompile cycle, when any change to a shader program requires restarting the system. Currently, shader development in our system is similar to other shader development tools, like Shader Designer [16] or Render Monkey [15].

Layers are one of the key elements of the system. The layers and the user interface for layer manipulations resemble that of Photoshop. Layers can be turned on/off, created and deleted. The user can change the order in which the layers are composited, as well as the opacity of each layer. Each layer is also a drawable, which allows applying of materials (and thus shaders) to it.



Fugure 3. Distributed clients access the proxies remotely.

Development within our system can be also described in terms of declarative programming. Certain properties are exposed in the IDE and can be modified without writing the code.

OpenGL lights are used in a number of shaders. The light control allows the user to create and manipulate lights. The user can turn the lights on and off and change light direction. Same light can be used in several shaders. The user can also create light collections with any number of lights as well as modify each light's individual parameters. Transfer function interface allows creating and modifying 1D and 2D transfer functions.

5 USE CASES

In this section we describe different types of use cases that can be solved by our system. We explain the versatility of the architecture and show how it gives the researchers necessary flexibility in developing new applications.

5.1 Single User Visualization

We concentrate on the description of our system as a framework for distributed collaborative visualization research. However, it is important to keep in mind that the framework can also be successfully employed in single user scenarios for visualization research and development. The Volume Compositor, briefly described in previous section, is an example of this.

Development of new visualization algorithms is aided by providing feature-rich and easy to use interfaces as well as giving the user freedom to develop new algorithms at different levels. For example, one can use our API in C# or Python for fast prototyping of new applications, rapidly develop shaders with the shader editor, and create libraries of materials and shaders. The user can also save different visualization states to be loaded at a later time.

5.2 Collaborative Visualization

In collaborative visualization, several users work on the same visualization. There are different dimensions of collaboration. The users may simultaneously interpret the visualization; they may simultaneously modify and tune visualization parameters. They may work in the same room or in different cities. In some cases, there is no need for collaboration to happen at the same time. How do all these issues impact the collaboration process? What are the exact needs of the users to maximize the efficiency? We explore these questions in the next paragraphs, highlighting different scenarios and showing how our system is utilized in these cases. We also provide an analysis of the architectural and implementation trade-offs.

Our system provides basic synchronization and distributed functionality. The system currently provides synchronous display and control on any number of remote clients. The maximum number of clients is bounded only by the performance of the hardware and network connection properties. The system supports either a client-server model or peer-to-peer communication. Either of these can be extended to some sort of a mixed communication or hierarchical control.

There are several scenarios for collaborative visualization with our system; we start from the very simple and obvious ones, and continue with more advanced examples. System architecture gives us enough flexibility to allow easy implementation of the scenarios described below. In all examples we assume that there is only one copy of each proxy object in the system. When this is not the case, we specifically mention this.

5.3 Real-Time Collaboration

In our first scenario single workstation and one or more PDAs are utilized. This scenario is particularly useful when the workstation provides output to a big tiled display or other immersive environment such as a CAVE. Each collaborator has a PDA. The PDA is a Windows CE device and is used to control input parameters and perform user interaction, while the workstation is used as a major graphics rendering back-bone and is a high end graphics system. The PDA has two user interface components: camera controller and a transfer function editor. Camera control UI component allows zooming, panning and rotation. The transfer function editor allows use and modification of 1D or 2D transfer functions. Both UI components can be interacted with either PDA keys or with a stylus. The visualization changes interactively as the user modifies camera or transfer function parameters at the PDA. The PDA does not require high-end graphics capabilities.

Visualization systems are increasingly used for teaching in different areas. One such area is medical education. For example, volume rendering system can provide initial training for medical students. Instead of using real cadavers, the students explore the volumetric datasets interactively on the computer.

Our scenario is an interactive learning session, when a group of students is working on the sample visualization project. There is a single instructor's workstation and up to 20 students' machines. The instructor acts as an administrator, having full control over the visualizations produced by the students. Instructor sees every student's visualization in a separate window and uses the full set of UI components to control materials, shaders, cameras and layers of each of the projects. The user interface allows instructor either to modify the student's project concurrently or to lock it to prevent possible collisions. The visualization session consists of the set of assignments that students should complete. For the first one the students produce a rendering of the particular bone. While exploring the dataset, the instructor views all of the projects that the students

are working on, and can modify them, if necessary. The next task is more complex – the students are asked to modify the visualization in a way that the soft tissue around the bone is also visible. Finally, the students create the volumetric cut. Instructor has control not only of individual machines but can also perform group tasks, such as saving the snapshots of all the students' projects.

5.4 Asynchronous Collaboration

We have described the cases when the users are working synchronously. However, this is not always necessary. Moreover, there are cases when asynchronous collaboration may be preferable over synchronous one. An example of such a case is collaborative work on medical or technical illustration. An example is shown in Figure 4. The first user have rendered the foot and committed the changes. The second user exported the layers to Photoshop and added the labels to create an illustration.

One more application is segmentation – several researchers may work together to segment a dataset. It is not always necessary for their collaboration to happen in real time. Assume the following scenario. A researcher in New York is working on segmentation with his or her colleague in LA. Researcher in New York provides initial segmentation and notifies researcher in LA. Researcher in LA checks the results, modifies the visualization project and notifies researcher in New York of the change. The process is iterative: they keep working on the project until they achieve the final segmentation.

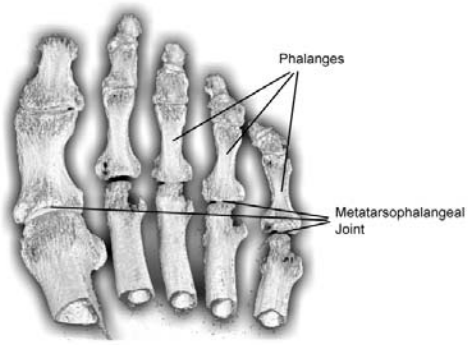


Figure 4. Medical illustration created with our framework. Foot and the drop shadow are rendered to the different layers. Blur operator was applied to the shadow to make it look soft. The layers were exported to Photoshop where labels were added.

5.5 Implementation Details

Our typical approach to ensure real-time synchronization currently implemented in the system is to keep only one copy of each proxy object in a global AssetManager residing only at one machine and allow all other machines to access proxies remotely. This ensures that all the machines see the same state of the proxy objects. Asynchronous collaboration is supported in the following way. A new instance of AssetManager is created at each of the machines. The proxies for these machines are created by cloning corresponding objects from the original AssetManager. This is illustrated in Figures 5a and 5b. Originally machine A's AssetManager was used by both machine A and remotely by machine B. Asynchronous collaboration is supported by creating an additional AssetManager residing on machine B. Machine B starts listening only to updating events generated locally. Machine B, however, still can access proxies residing in A's AssetManager and vice versa. Switching back to synchronized collaboration is done by substituting the local AssetManager by the original remote one from machine A. This example is a special case of a general m-n relationship with m AssetManagers and n machines that use them. In this general case AssetManagers may be organized in a hierarchy, such as in Figure 5d. In the figure AssetManagers at the bottom are synchronized through the topmost AssetManager.

The sets of proxy objects when there are multiple AssetManagers may be either disjoint or overlapping. Disjoint AssetManagers can keep different types of proxies, for example a separate one may be designated to manage volume proxies, while another is used for materials and shaders. Case when the AssetManagers contain copies of the same objects is shown in Figures 5a and 5b.

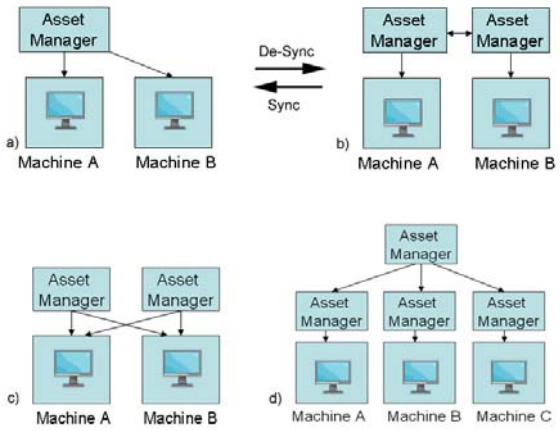


Figure 5. AssetManagers in distributed scenarios.

Up to this point we haven't discussed where the datasets are being stored. In case of very large datasets this becomes an important question. Traditional solution to this problem is creating some sort of a centralized data server. Main advantage of using centralized data server is the achieved flexibility. With the data server there is no need to send the whole dataset over the network; this may be beneficial when dataset is very large, for example in the case of seismic data. We achieve this by implementing a volume handler (mediator) with this functionality. The object of the handler class resides on the data server machine. The proxy objects and AssetManager don't have to be modified. When the dataset is rendered on the client machine, the request for data goes from the graphics level to the local mediator object, which, in turn, gets it from the handler on the server. The connections between data server and clients are managed by the handler and proxies and AssetManager are not involved. The handler supports data compression and level-of-detail data processing.

One of the issues that arise when talking about distributed applications is robustness, i.e. the ability of the application to perform properly if one or more of the clients crashes. In case of a single AssetManager such as in Figure 5a), non responsive clients do not affect the rest of the clients. For example, if machine B in Figure 5a) froze, application at the machine A would still run. This is accomplished by using a one-way asynchronous communication from the proxies to the mediator level (as shown in Figure 1), thus non responsive clients don't bring the system to a halt.

With multiple AssetManagers the scenarios become slightly more complicated; however in such cases we also have more flexibility – a machine that has its own AssetManager may remain active when other stop their execution. The following example shows in detail one of the "crashing" scenarios. Consider two systems, A and B, as shown in figure 5c). There are some proxies in A's AssetManager that are used by B, and there are some proxies in B's AssetManager that are used by A; the sets of proxies are disjoint. Whenever one of A or B terminates or freezes, the other one has to terminate, because the remote proxies are no longer available. Straightforward solution is achieved by enforcing proxies' overlap, i.e. for any remote proxy a local copy should be created.

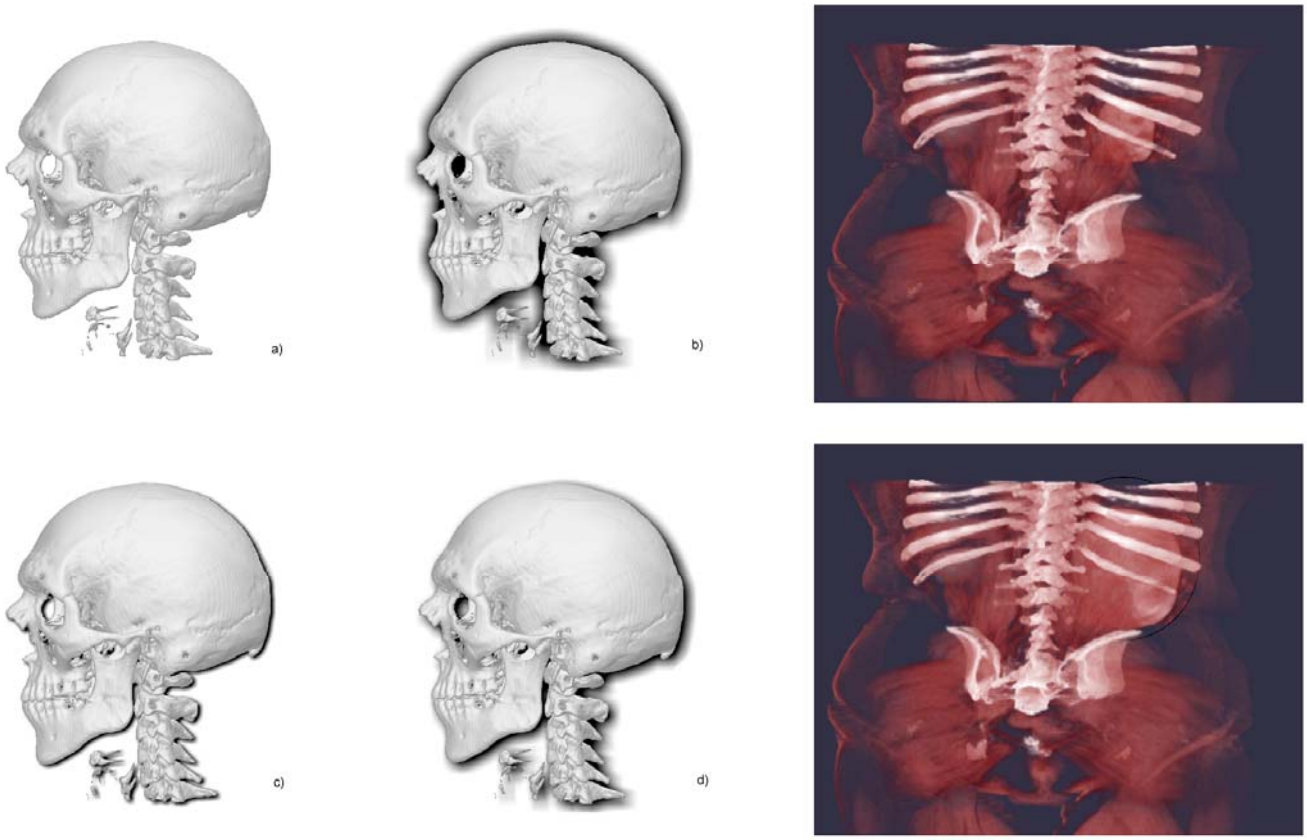


Figure 6. Examples of the use of layers. a) Skull rendering. b) Skull rendered with a halo. Skull and halo are rendered to separate layers. Halo is produced by blurring a 2D shadow. c) and d) 2D-drop shadows. Right: kidney magnification. Right top: torso dataset with a non-magnified kidney. Right bottom: kidney is rendered to a separate layer, thus when it is magnified, the ribs are not distorted.

EXTENSIONS

5.6 VTK-ITK Plugins

VTK is well known system for scientific visualization. It consists of a set of core libraries written in C++ and a number of higher level libraries, written in Tcl/Tk and Python. The VTK operates in terms of a pipeline. The users create modules that are used for processing data as it goes down the pipeline. In our framework, we include VTK and ITK [10] functionality at the graphics entity level, instead of using directly OpenGL or Direct3D. Proxy objects for those are pipeline scripts or programs, for instance in Python.

5.7 Scripting, Python and Animation

One of the big issues in current visualization toolkits is the complexity of setting up the system parameters. To get a visualization that suits the user, sometimes hours of tedious parameter-tweaking is necessary. Saving visualization projects is not the best solution, because in this case we save just the current state of the system. It's preferable to be able to somehow record the way the project was created and modified. One way to tackle this problem is to record the steps necessary to create a particular visualization, so that later it can either be easily redone or modified. To save a series of steps that lead to the particular state, we use Python scripts. This solution has the major advantage. Clear, intuitive and concise syntax of Python commands allows its use by non-programmers. Python commands are converted to the corresponding API calls through a simple wrapper class.

Animations may greatly increase the expressiveness of visualization. There are two ways of creating animations in our system. The first involves creating images of different time steps and then merging them into the single animation in a way similar to Adobe Image Ready; the other involves using Python scripts. With scripts, the user is able to specify what and how specific parameters change, as well as the appropriate timing. We prefer the second way, because for long animations it is much less time consuming. Moreover, when scripting is used, any new user interface can access the system through scripts. Scripts have access to all data in the AssetManager and the core interface types. Notice that one can still use system's C# API for this purpose.

6 LAYERS

The use of layers is an important feature of our system. The layers employ the same idea as the layers in Photoshop. Thorough coverage of the layers is beyond the scope of the paper, thus in this section we show some results done with the use of layers. In Figure 6 on the left the skull is rendered without and with the drop shadows and a halo. On the right we demonstrate the use of a layer based magnification. Kidney is rendered to a separate layer and thus is magnified without distorting the ribs.

7 SYSTEM IMPLEMENTATION

7.1 .NET Platform

Our implementation is based upon the C# programming language and the .NET framework [12]. First introduced in 2000, .NET provides the following features to the applications: interoperability, language independence, common language runtime and a rich set of standard libraries. The Common Language Runtime (CLR) allows code written in different .NET languages to be compiled to an intermediate language that is interpreted by the platform. Just-in-time compilation is used to improve performance.

.NET Remoting is a technology that allows distributed applications to communicate by using the references to so-called remote objects. .NET hides from the developer the complexity of the underlying network protocols. The framework automatically generates object stubs and provides communication mechanisms. From a developer's standpoint, the use of remoting is quite straightforward: after proper initialization, remote objects can be used as if they were local. All the proxies in our system support remoting. This allows distributed applications to operate with the same proxies, thus achieving synchronization.

Some care needs be taken when making proxies remote. For example, making a volumetric dataset a proxy, we may force one or more copies of the dataset across the network, which in case of large datasets is not desirable. As discussed in the Use Cases section, one solution is to create a separate data server. The proxy merely indicates the basic properties of the dataset. The underlying mediators can then reference data from a local store or stream it from the data server.

Performance is always an issue in graphics and visualization. When designing the system, performance was not the major concern, however. The design goals were to create scalable, easy to distribute, maintain and modify visualization system with extensive use of shaders and convenient user interfaces. Still, we achieve good performance results. These are due not to a set of optimizations, for example, but to overall system architecture which ensures such things as lazy initialization, minimizing the number of events thrown and processed, updating objects only when necessary (lazy updates). Some of the performance benefits come from the fact that we use layers and perform image manipulation instead of more computationally expensive volume rendering in a number of cases. In total, for many test cases the overall performance was better with our new system.

7.2 System Design

There are a number of key classes and interfaces that we will describe in detail to provide a better understanding on how the system operates. As mentioned in the "Architecture" section, the system has three different kinds of entities. The top proxy tier provides object sharing and remoting functionality, the mediator tier works with local copies of the proxies, and the graphics tier provides the necessary graphics library calls. Major classes and relationships between them are shown in Figure 7. The relationship between various classes is the same across the three tiers; at each tier they implement the same interfaces. For example, IMaterial is implemented by MaterialProxy, MaterialHandler and MaterialGL. Class diagrams for the key interfaces and proxy classes are shown in Figure 8. For example, at the proxies' tier, ViewProxy contains one or more LayerProxies; each LayerProxy has a collection of drawable proxies, each of them, in turn, has a MaterialProxy. The same relationship is followed by mediators' tier (handler classes), as well as by the graphics tier.

7.3 Classes and Interfaces

Every object that should eventually be rendered to the framebuffer implements an IDrawable interface. The key method in the

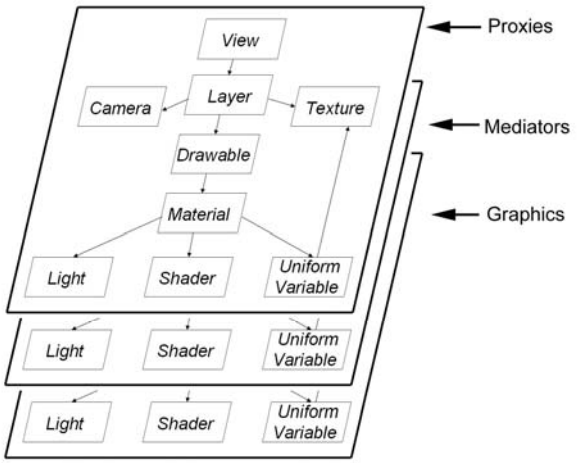


Figure 7. Relationships between classes in the three tiers are the same.

interface is Render(). Examples of drawables are volumetric regions, slice planes and outlines. In the corresponding proxies these methods are stubbed, however; the real rendering happens in the graphics tier.

All the proxies in our system are called assets and implement IAssetManaged interface. The AssetManager class stores references to every proxy. The AssetManager acts as a server, and thus at any time the necessary proxy can be accessed by calling AssetManager.Instance.getItem(). Each asset is referenced by its type and string id. For example, the main camera may be retrieved with the code:

AssetManager.Instance.GetItem(typeof(Camera), "Main Camera").

MaterialProxy, MaterialHandler and MaterialGL classes implement the IMaterial interface. The methods of IMaterial include MakeActive() and Deactivate() for enabling the material just before rendering, and disabling it after rendering has been completed. Each material has a corresponding shader that is used to render the material. The system provides a lengthy set of predefined shaders and materials. Shader program contains a set of vertex, fragment and geometry routines. We associate uniform variables with a material, not a shader. This means that the same shader for volume rendering, for example, may be used to render two different volumes with different transfer functions. The volume rendering shader is the same, while the uniform variables (volumes and transfer functions) are different.

A view contains any number of layers. Each view has its own collection of references to the layers and includes a camera for compositing. A layer can be thought of as compositing container of drawables as well as IBR-like image. From an implementation standpoint, a layer produces a 2D texture. LayerProxy contains a list of drawables to be rendered, a material for the layer and a texture proxy to render into. Layer is a 2D texture and can be used in any material requiring a texture.

Textures are employed extensively in the system. They are used for representing transfer functions, layers and volumes. Transfer functions may be one or two-dimensional. Layers are represented as two-dimensional textures, and volumes as three-dimensional textures. Texture proxies keep information about texture format and size. A texture handler takes care of binding/unbinding of textures to texture units. TextureGL provides simple enable/disable functionality. The mediator (handler) in this case provides both the synchronization as well as pushes the data into OpenGL.

There are a number of classes that represent basic mathematical and geometric objects: regions of interest, planes, points, scalars, etc. These provide a consistent framework for user manipulation and are

the building blocks of most drawables. A volumetric drawable contains a region of interest. A slice plane contains a plane as well as the region of interest. If two drawables share the same mathematical entity, any change in this entity is reflected in both drawables.

Lights are used in a variety of shaders. Though it is possible to set up light parameters directly in every shader, it is much more convenient to create a light or light collection, and use gl_Light[] references in the shader to get the corresponding light parameter. We use slice-based volume rendering in the system, employing 3D textures. We also have implemented a one-pass raycaster.

7.4 Remoting

The following classes are used to provide remoting functionality. MarshalByRefObject is a .NET framework class that enables access to remote objects in applications that support remoting. SharedObject is the base class for all the proxies in the system. Each shared object extends MarshalByRefObject class and implements the IAssetManaged interface. Thus, each shared object can be accessed remotely as an asset in the AssetManager.

We describe the use of remoting in our system with the following scenario. There are two machines, A and B. Machine A is being used as a remote display. It displays whatever is being rendered at the machine B. The machine A does not need any user interface, except for selecting the network address of the machine B to setup connection. When the application at the machine A is started, it gets the remote reference of the AssetManager. Once this is done, a local view is created using remote ViewProxy object returned by the AssetManager. Then machine A subscribes to the view changing events of the ViewProxy on machine B. The first time the view is processed on machine A, it creates any necessary mediators for its components (for example, layers) recursively. These are then associated with their corresponding proxies. Subsequently, any changes to one of these proxies notifies its mediator, which sets a dirty bit. It also notifies its proxy container which will eventually instruct the view to re-render. During the rendering any mediator whose dirty bit is set will re-sync with its proxy. Thus a minimal set of objects must be synchronized and rendered. Each of the local mediators on machine A (the objects from the middle tier) is created automatically via lazy initialization from the corresponding proxies. This example shows how our system's architecture provides a straightforward and concise way of building distributed visualization applications, minimizing programmer's efforts.

7.5 Graphics tier

Our system architecture allows a developer to switch from current implementation with OpenGL to any other graphics library, for example Direct3D. This can be done by substituting OpenGL calls with corresponding Direct3D ones. The number of GL classes is small and the functionality is very well encapsulated and modular. Since the three entities have a low coupling, the transition from one graphics library to another can be done easily. This allowed us to add high level drawables with VTK to our lower-level OpenGL drawables. Likewise, we have experimented with migrating the graphics tier to a Linux cluster controlling a power-wall.

8 Conclusion

We presented a new framework for single-user and collaborative visualization. The major design goals were providing an easy to use, scalable, distributed system that can be used for visualization research. We showed how the system can be used for a variety of visualization scenarios. The system is designed in a way that allows researchers to easily add new functionality, either by directly implementing new classes in C#, or by using a scripting language like Python. We hope that our system will become a standard workbench for research in scientific visualization.

References

[1] G. Abrams and L. Trenish, "An Extended Data-Flow Architecture for Data Analysis and Visualization," Proc. IEEE Visualization 1995, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 263-270.

[2] L.Bavoil, S.P. Callahan, P.J.Crossno, J. Freire, C. E. Scheidegger, C.T. Silva, and H. T. Vo. VisTrails: enabling interactive multiple-view visualizations *In Proceedings of IEEE Visualization*, 2005, Vis'05, 135-142.

[3] K. W. Brodlie, D. A. Duce, J. R. Gallop, J. P. R. B. Walton, J. D. Wood (2004) Distributed and Collaborative Visualization *Computer Graphics Forum* 23 (2) , 223–251

[4] Cruz-Neira, C., Sandin, D. J., DeFanti, T. A., Kenyon, R. V., and Hart, J. C. 1992. The CAVE: audio visual experience automatic virtual environment. *Commun. ACM* 35, 6 (Jun. 1992), 64-72.

[5] gViz http://www.comp.leeds.ac.uk/vvr/gViz/

[6] Steven P. Callahan, Juliana Freire, Emmanuele Santos, Carlos Scheidegger, Claudio Silva, Huy T. Vo, VisTrails: *Visualization Meets Data Management.* SIGMOD 2006, June 27-29, 2006, Chicago, IL

[7] Foulser, D. 1995. IRIS Explorer: a framework for investigation. *SIGGRAPH Comput. Graph.* 29, 2 (May. 1995), 13-16.

[8] Garrett J.J. Ajax: A New Approach to Web Applications. 2005 http://www.adaptivepath.com/ideas/essays/archives/000385.php

[9] Grimstead, I. J., Avis, N. J., and Walker, D. W. 2004. Automatic Distribution of Rendering Workloads in a Grid Enabled Collaborative Visualization Environment. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (Nov.06-12, 2004). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC

[10] ITK Toolkit http://www.itk.org/

[11] Leigh J., Rajlich P., Stein R., Johnson A., DeFanti T. LIMBO/VTK: A Tool for Rapid Tele-Immersive Visualization, CDROM proc. Of IEEE Visualization '98, Research Triangle Park, NC, October 18-23, 1998.

[12] .NET Framework http://msdn2.microsoft.com/en-us/netframework/default.aspx

[13] OpenDX. http://www.opendx.org/

[14] pV3 http://raphael.mit.edu/pv3/pv3.html

[15] Render Monkey http://ati.amd.com/developer/rendermonkey/index.html

[16] Shader Designer http://www.typhoonlabs.com/

[17] Swivel http://www.swivel.com/

[18] Taesombut, N., Wu, X., Chien, A. A., Nayak, A., Smith, B., Kilb, D., Im, T., Samilo, D., Kent, G., and Orcutt, J. 2006. Collaborative data visualization for earth sciences with the OptIPuter. *Future Gener. Comput. Syst.* 22, 8 (Oct. 2006), 955-963.

[19] Upson, C., Faulhaber, T., Kamins, D., Laidlaw, D. H., Schlegel, D., Vroom, J., Gurwitz, R., and van Dam, A. 1989. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Comput. Graph. Appl.* 9, 4 (Jul. 1989), 30-42.

[20] Viegas, F.B., Wattenberg, M. , van Ham, F., Kriss, J., McKeon, M. 2007. ManyEyes: a Site for Visualization at Internet Scale. *IEEE Transactions on Visualization and Computer Graphics*, 13 (6), pp 1121-1128.

[21] Wood, J., H. Wright, and K. Brodlie. 1997. *Collaborative Visualization*. Proceedings, IEEE Information Visualization '97, Pheoniz, Oct. 19-24, 1997, pp. 253-260.

[22] Zhao Y., Hu C., Huang Y., Ma D. "Collaborative Visualization of Large Scale Datasets Using Web Services," p. 62, *Second International Conference on Internet and Web Applications and Services (ICIW'07)*, 2007.

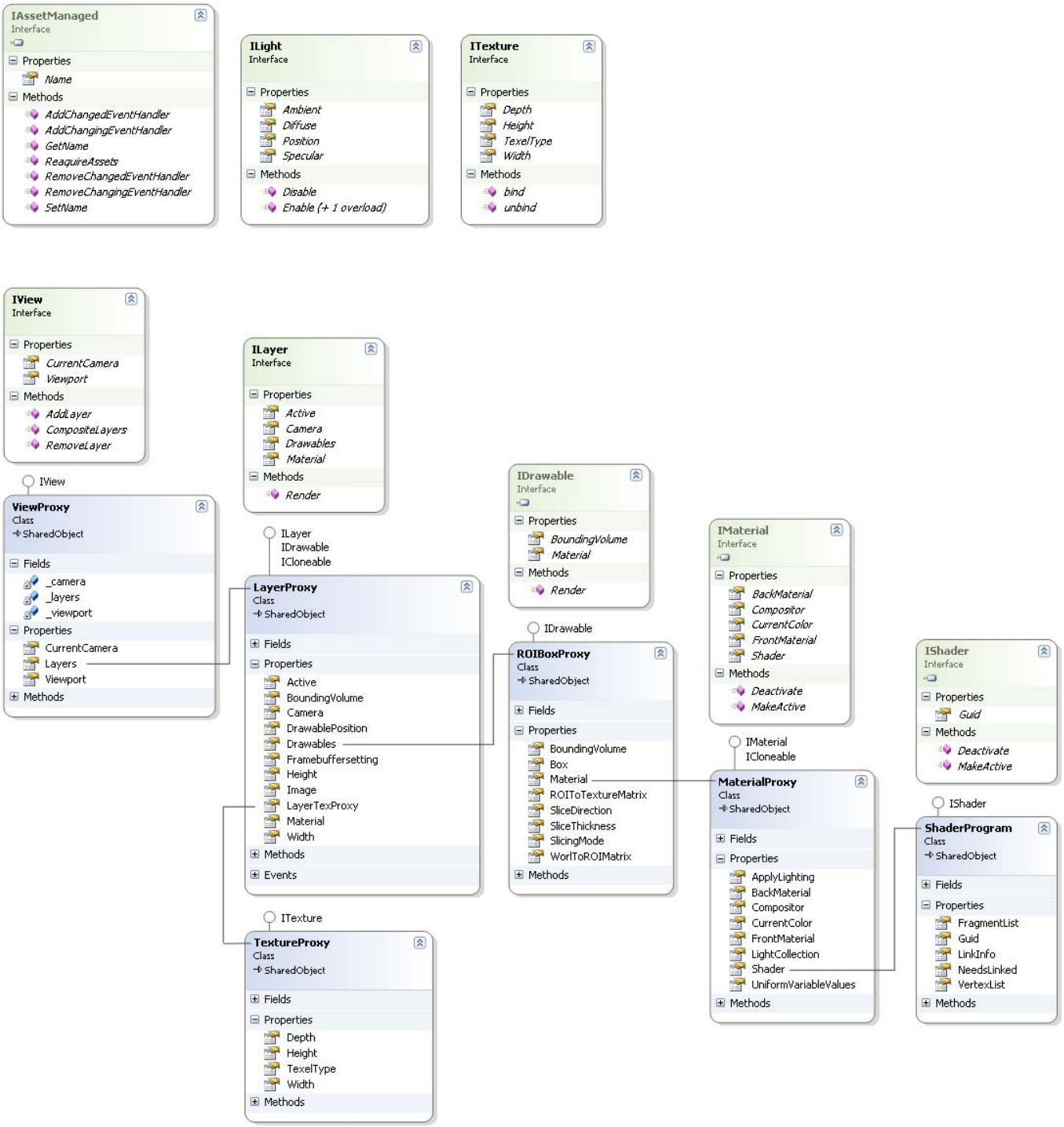


Figure 8. Major interfaces and proxy classes of our system.