A Shader-Based Parallel Rendering Framework

Jérémie Allard* Bruno Raffin†

ID-IMAG, CNRS/INPG/INRIA/UJF
Grenoble - France

ABSTRACT

Existing parallel or remote rendering solutions rely on communicating pixels, OpenGL commands, scene-graph changes or application-specific data. We propose an intermediate solution based on a set of independent graphics primitives that use hardware shaders to specify their visual appearance. Compared to an OpenGL based approach, it reduces the complexity of the model by eliminating most fixed function parameters while giving access to the latest functionalities of graphics cards. It also suppresses the OpenGL state machine that creates data dependencies making primitive re-scheduling difficult.

Using a retained-mode communication protocol transmitting changes between each frame, combined with the possibility to use shaders to implement interactive data processing operations instead of sending final colors and geometry, we are able to optimize the network load. High level information such as bounding volumes is used to setup advanced schemes where primitives are issued in parallel, routed according to their visibility, merged and re-ordered when received for rendering. Different optimization algorithms can be efficiently implemented, saving network bandwidth or reducing texture switches for instance.

We present performance results based on two VTK applications, a parallel iso-surface extraction and a parallel volume renderer. We compare our approach with Chromium. Results show that our approach leads to significantly better performance and scalability, while offering easy access to hardware accelerated rendering algorithms.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics

Keywords: Distributed Rendering; Shaders; Volume Rendering

1 INTRODUCTION

Context. In recent years, graphics clusters have become a platform of choice for high performance scientific visualization. The goal is to interconnect multiple PCs through a dedicated network to aggregate the power of their disks, memories, CPUs and GPUs. Such cluster can drive from a single screen to a display wall of several projectors allowing a higher brightness, resolution and display size. The difficulty is then to develop software solutions to efficiently take advantage of such platforms.

Graphics applications, like scientific visualization ones, are often structured as a pipeline. A raw data set is read either from disk or from a live simulation application. It is processed to obtain a graphical representation. This scene is then rendered to an image that is finally displayed. Each of these stages can be distributed differently on a cluster. For example, the final display can be connected to a single host, or each host can drive one video projector of a large display wall. Each host can render the pixels it displays, or it can render pixels to be displayed by different hosts. In this case pixels have to be communicated on the network and recomposed to form the final images [**?**]. This scheme is usually called a sort-last approach [**?**]. The graphics objects can also be produced with a different distribution scheme than the one adopted for rendering (sort-first approach). When the higher-level stage is distributed, multiple data streams have to be merged, and similarly scatter operations are required when distributing lower-level stages. The scalability of the system depends on the overhead introduced by these operations and the volume of communications.

Implementing higher level communications is generally more dependent on the application. Sort-last solutions are highly generic, as pixel format does not change. Sort-first frameworks depend on the rendering API used in the application to describe the scene. Most applications use OpenGL.

Humphreys et al. [**?**] proposes a framework called Chromium. Chromium uses a protocol for distributing OpenGL commands. However, due to OpenGL's history and the requirement to support legacy applications, these commands are numerous and often redundant. For example, OpenGL supports both immediate-mode primitives and retained-mode objects (using several concepts such as display lists, vertex array, buffer objects). Immediate-mode rendering does not allow to easily detect changes between frames, introducing duplicated communications or a high computational overhead to detect unchanged primitives. Similarly, high level informations such as bounding boxes are not available. It makes it difficult to perform efficient scatter operations (i.e. frustum culling) to different render hosts. Moreover, as OpenGL is based on a sequential state machine, commands must respect a strict ordering. Merging multiple streams together requires Chromium to track state changes and to use special commands defining the relative ordering of each stream.

All these constraints are driven by the need to support legacy OpenGL applications. In the context of scientific visualization where performance is critical and the final rendering is often handled by a shared toolkit, considering alternative solutions to an OpenGL based protocol is relevant.

In the rest of this paper we will refer to the programs used in the rendering stage of the pipeline as *renderers*, and the programs responsible to create the graphics objects will be called *viewers*.

Contribution. We propose a sort-first retained-mode parallel rendering framework called *FlowVR Render*. Instead of relying on OpenGL commands, we define a *shader based protocol* using independent batches of polygons as primitives. This protocol offer the following benefits:

- Shaders are used to specify the visual appearance of graphics objects. They require only a few parameters and not the full complexity of the fixed-function OpenGL state machine. It leads to a simpler protocol that does not have to manage state tracking.
- Shaders enables to easily take advantage of all features offered by programmable graphics cards.

*e-mail: Jeremie.Allard@imag.fr
†e-mail:Bruno.Raffin@imag.fr

- Primitives are un-ordered unless explicitly stated by viewers (for transparent objects or user-interface overlays for instance). It enables renderers to optimize the local rendering order, reducing shader and texture switches for example. It also eases merging and reordering primitives coming from multiple streams.

- A higher level information such as bounding-boxes or changes since the last frame can be directly specified by viewers, avoiding unnecessary processing and network traffic. In most cases this information is already available (visualization toolkits often detect changes between frames to recompute only the affected data).

We base our design and implementation on FlowVR [**?**], a middleware for modular data-flow based interactive applications. It provides the environment to develop, launch and execute a distributed application on a cluster. It also allows to express complex collective communications patterns and coupling policies.

Compared to an OpenGL based protocol, our approach requires modifying the applications to issue FlowVR Render primitives. However this drawback is limited as scientific visualization codes usually rely on a reduced set of graphics primitives. Porting their rendering API to FlowVR Render is usually not too difficult. To demonstrate this, we modified the rendering code of VTK [**?**], one of the most widely used visualization toolkit, to transparently support FlowVR Render.

Rendering on a display wall with a varying number of parallel viewers and renderer for iso-surface extraction and volume rendering was tested both with VTK+Chromium and VTK+FlowVR Render. FlowVR Render leads to a higher performance and a better network scalability as the number of viewers and/or renderers increases. The shader based protocol also enables to implement hardware-accelerated volume rendering algorithms that achieve high quality rendering [**?**, **?**].

Organization. After presenting the related works in section 2, we detail the protocol as well as the basic filtering operations used by FlowVR Render in section 3. Communication patterns for parallel rendering will be discussed in section 4. Their usage is presented and tested using two applications in section 5. Finally results and future works are discussed in section 6.

2 Background and Related Work

In this section, we discuss rendering APIs before giving an overview of existing parallel rendering approaches. The section ends with a discussion on the benefits of using shaders for visualization algorithms.

2.1 Rendering APIs

OpenGL [**?**] is currently the predominant graphics rendering API and is available for most platforms and graphics cards. It is originally based on a immediate-mode model where the application sequentially specifies the objects in the scene. A large state machine is used to handle rendering parameters such as transformation matrices, lights, material properties, etc. Graphics cards and hardware platforms have evolved significantly since OpenGL introduction in 1992. To match this evolution an extension mechanism is used for hardware vendors to expose additional functionalities corresponding to new features or optimizations. Periodically, a new OpenGL version standardizing useful extensions is released by the Architecture Review Board (ARB). The latest version, OpenGL 2.0, was released in September 2004 and supports recent features such as non-power-of-two textures and high-level programmable shaders.

Pure immediate-mode rendering requires re-issuing all graphics primitives for each frame and thus can not benefit from inter-frame coherence. It introduces a high overhead on the CPU and the CPU to GPU communication bus. As the graphics cards performance improved, it became a serious performance bottleneck. To reduce this bottleneck, OpenGL has evolved to support a number of mechanisms to optimize data uploads to the graphics card, ranging from the original display-list mechanism to compiled vertex arrays and the recent vertex/pixel buffer objects extensions.

Each new OpenGL version is backward compatible with all previous versions. As a consequence many deprecated or duplicate functionalities exist such as the different ways to specify vertex data or the fixed function pipeline superseded by programmable shaders [**?**]. This significantly increases the complexity of OpenGL implementations. An effort to remove legacy commands and states from OpenGL's core API was originally proposed for OpenGL 2.0. This proposal was not accepted but led to a derived API for embedded systems, OpenGL ES [**?**].

In opposite to immediate-mode rendering, retained-mode APIs [**?**, **?**] manage a scene description the application creates and then updates at each frame (often in the form of a hierarchical scene graph). This model is today commonly used by scene graph libraries on top of OpenGL.

2.2 Parallel Rendering

Different approaches have been proposed for cluster based parallel rendering [**?**]. Sort-last parallelism [**?**] relies on hardware or software compositors to combine pixels computed on different graphics cards [**?**, **?**]. This approach has also been used for load balancing on display walls [**?**].

Sort-first approaches distribute graphics primitives to several rendering nodes. Sorting is generally based on each node's view frustum. Pixels are either directly displayed or sort-last techniques are used to recombined them. Chromium [**?**] proposes an OpenGL based protocol that enables sort-first parallel rendering. Legacy OpenGL applications can be executed on a display wall without recompilation. Chromium intercepts primitives issued by the application and sends them to the rendering hosts. It relies on advanced caching, compression and state tracking mechanisms to save network bandwidth. Chromium also proposes a set of OpenGL extensions for ordering several primitive streams in order to enable a parallel primitive generation, but at the price of the OpenGL compatibility.

Other approaches work at a higher level. Scene graphs rely on a partial graph duplication on rendering nodes and coherency protocols [**?**], to balance network traffic, duplication of data and computations. However, the user control over the parallelization scheme implemented is usually limited. Scalable scientific visualization often requires a finer control on data movements and to combine different parallelization schemes to be efficient.

2.3 Shader-Based Visualization

Procedural shaders have been extensively used in offline software-based rendering [**?**] to specify the visual appearance of graphics objects. Initially hardware systems only supported fixed functionalities that were programmed through a set of parameters or switches. New generations of graphics cards are now able to execute full programmable shaders [**?**, **?**]. Recent models [**?**] support high-level shaders [**?**, **?**] with method calls, loops, conditionals, and full 32-bit floating point precision for computation and textures/buffers.

Shaders provide additional flexibility and precision allowing graphics cards to execute algorithms that only the CPU could execute before. Such algorithms include high-quality lighting models (per-pixel evaluation, shadows), tone shading [**?**], or volume rendering [**?**]. For instance, obtaining colors from raw data by applying a transfer function can now be done within a shader.

2.4 Summary

Using recent OpenGL buffer objects and hardware shaders, applications can efficiently take advantage of advanced graphic cards features. However, the OpenGL state machine and the numerous duplicated functionalities that OpenGL still supports significantly increase the complexity and hinder the effectiveness of any implementation, in particular parallel ones.

Using shaders, parts of the data processing visualization pipeline is now executed by the graphics card, distributing the load between the CPU and the GPU. This changes the level of information sent to the graphics cards, which can receive general datasets instead of only final colors. Tuning the parameters of the processing implemented by shaders only requires updating a few values instead of re-uploading the complete dataset. In a distributed context where graphics primitives are issued on one machine and rendering is performed on a distant one this can drastically change the performance of the pipeline.

We propose to develop a pure shader based sort-first protocol dedicated to high performance parallel rendering.

3 FLOWVR RENDER MODEL

In this section we present the FlowVR Render framework and its different components. A viewer program describes *primitives* sent to a renderer program using a specific *protocol*. Complex parallel rendering architectures can be built combining various viewers and renderers mapped on different nodes of a cluster. *Filters* are used to implement advanced routing functions on primitives.

3.1 Graphics Primitives

A scene is described as a set of independent *primitives*. Each primitive contains the description of a batch of polygons and their rendering parameters (see figure 1). To reduce the number of these parameters, as well as to take advantage of the advanced features of recent graphics cards, we use *shaders* to define each primitive's appearance.

Large resources such as textures or vertex attributes are often used by several primitives in the scene. To avoid duplicating these resources in each primitive, we define *resources*. A resource can encapsulate a shader, texture, vertex buffer or index buffer. It is identified by an unique *ID*. Each primitive refers to the IDs of all the resources it uses. Notice that it introduces a one-way dependency between primitives and resources, but not between primitives.

The framework must ensure that IDs are globally unique even in a distributed context. Possible methods include using a specific host, which can introduce a bottleneck for large applications, or using local information unique to each host. In our implementation we use 64-bits IDs by appending an atomic counter local to each host with the IP address of the host.

A *name* can be specified for each primitive for identification in error messages, visual selection feedback, or filters based on name patterns.

Each primitive also stores its *bounding box*. This information is required to optimize communications using frustum culling. It is useful to specify this information in the primitive as it is costly to recover from the other parameters (especially when using shaders that can change vertex positions).

Some rendering algorithms require primitives to be processed in a given order. FlowVR Render provides a mechanism to enforce this ordering by associating each primitive with an *order* value. This number defines a partial ordering between primitives. A primitive with a lower order value must be rendered before a primitive with a higher value. Primitives with the same value can be rendered in any order. Different values will mainly be used when rendering order can affect the final image like for transparent primitives

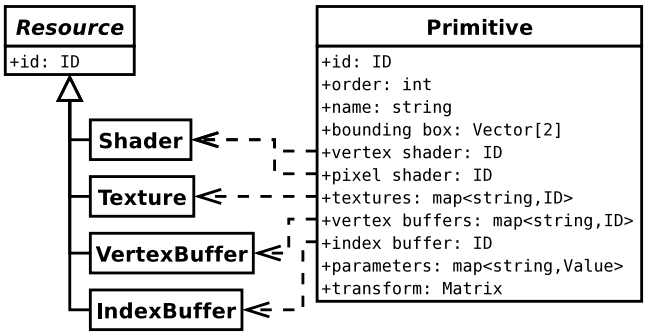


Figure 1: Simplified UML schema of a primitive.

or user-interface overlays. For a given order value, renderers can re-order primitives to improve performance. For instance, primitives can be sorted front-to-back to take advantage of fast Z-culling, primitives with similar parameters such as textures and shaders can be gathered to reduce state switching overheads. This approach enables to easily implement performance optimization compared to the strict ordering defined by an immediate-mode command stream. This strict ordering can however still be achieved by assigning a different order value to each primitive.

Global parameters such as camera position, z clipping distances, or background colors are directly set at the renderer statically or upon reception of user's interaction events. A viewer can also override these parameters using a primitive with the predefined ID *ID_CAMERA*.

3.2 Communication Protocol

In graphics scenes some data are static, i.e. they do not change between frames, while others are dynamic. One important optimization is to describe static information only once, and then transmit changes at each frame. To achieve this goal, renderers maintain the current description of the scene, and viewers describe the changes to this description. Each change is encapsulated in a *chunk*. At each frame, a viewer sends one *message* containing a list of chunks. A renderer waits to receive one message. A chunk can correspond to a:

- Creation of a new resource ;
- Destruction of a resource ;
- Update of a resource ;
- Creation of a new primitive ;
- Destruction of a primitive ;
- Modification of a primitive's parameter.

This protocol is purely one-way, from viewers to renderers. In particular, IDs are generated by the viewers and not the renderers. This property is useful when the viewers and renderers are not tightly synchronized (for example storing messages on disk and replaying them later).

3.3 Filters

Parallel rendering schemes require filtering message streams to distribute data between several viewers and/or renderers. For that purpose we introduce *filters* that implement the processings necessary for advanced routing networks. We define in the following some

common filters. Complex assemblies of viewers, filters and renderers are presented in section 4.

To support scene descriptions distributed on multiple viewer modules, it is necessary to be able to merge several messages together. In our model all primitives are independent and use globally unique IDs. As a consequence this operation consists in a simple gather, appending messages from all streams together.

For sort-first rendering with a single viewer, a simple broadcast can be used to send messages to all renderers. For highly dynamic scenes, messages can be filtered by removing (*cull*) all changes not affecting the local view frustum of each renderer. The bounding box information is readily available to test for intersection with the frustum, enabling to simply discard hidden primitives, without any effects on other primitives as they are independent. The only difficulty concerns resources (textures, vertex data), as they can be shared by several primitives. A simple algorithm consists in sending all resources referenced by each visible primitive, provided it wasn't already sent. For very large textures or meshes, a more complex implementation may only send visible parts of each resource, but recovering this information is costly.

Thanks to the one-way and retained-mode nature of our model, another operation that is very efficient and easy to implement is to allow for different frequencies between viewers and/or renderers. In particular, this enables multiple asynchronous visualization of the scene (two distant displays in a collaborative application, or a low-performance console view not affecting the performance of the main display for instance). Another more advanced application consists for each viewers in having a specific update frequency (low-frequency for large objects or remote viewers, high-frequency for local viewers or interactive updates such as camera manipulation). Filters implementing this strategy simply require either inserting empty messages or appending several messages together.

Other operations can be implemented depending on the applications. It is for instance possible to design filters altering the appearance of the scene for stylized rendering [**?**, **?**], splitting the objects for an exploded view [**?**], or writing the scene's 3D description in a file [**?**]. Compared to the traditional approach of modifying OpenGL commands, implementing these operations using our framework is easier as higher-level information is available. Also only a handful set of chunk types are used, compared to the hundreds of different OpenGL commands.

4 System Architecture

Data-flow based architectures, where several data streams are processed through a network of filters, have been successfully applied both in visualization and distributed rendering applications [**?**, **?**, **?**]. It can be applied at different stages in the application, from inputs events to final pixels composing. This section presents the design of the data-flow network used to transmit graphics primitives.

4.1 General Design

FlowVR Render architecture follows a data-flow graph. The source nodes of the graph are viewers that produce scene description messages as described in section 3.2. These messages travel through a network of filters that are responsible for implementing the necessary operations as described in section 3.3. Once data have been redistributed and processed by this network, it is used by the destination nodes, the renderers, to render the scene.

We consider modular visualization applications, where objects in the same scene are handled by completely different programs. As FlowVR Render primitves are a-priori independent, merging several data streams has a small overhead. This architecture provides two main advantages. First it favors code reuse, allowing to choose the right visualization toolkit or library for each task instead of reimplementing it in a single environment, and second it permits to choose a different distribution strategy for each object, allowing a fine performance tuning.

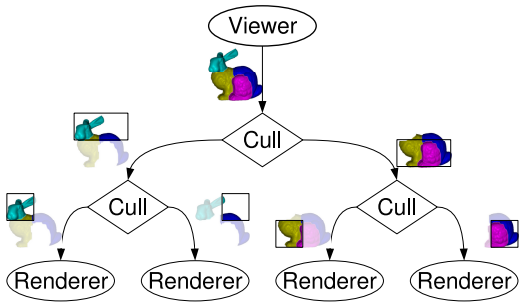


Figure 2: Sort-first distributed rendering. For small or static objects a broadcast tree can be used instead of frustum culling filters.

4.2 Distribution Strategies

Distributed rendering is a typical problem where no generic solution exists. Several factors (size and time dependency of datasets, computational power, network speed, number of pixels) are involved in determining the best distribution scheme between sort-first [**?**], sort-last, hybrid, or input-event based distribution. Testing different strategies should thus be as simple as possible. Moreover, as the same program can be used in situations requiring different distirbution strategies, changing the distribution scheme should require minimal modifications to the program. Data-flow based architectures propose an elegant solution to this issue as changing of strategy often consists in simply changing the position of each element, eventually adding some filters. The modular design of applications in FlowVR Render lets multiple schemes be simultaneously executed for different parts of the scene. The rest of this section will concentrate on the basic patterns implementing each distribution scheme. Using several schemes in a single application is then possible by combining these patterns together.

In a multiple display system (cave or display wall for example), if one viewer is connected to several renderers through frustum-culling filters as shown in figure 2 we have a sort-first distribution model. Notice that for better scalability we can use a two-level culling filter tree: first splitting horizontally then vertically. Depending on the size and dynamicity of the primitives, the cull filter can either simply cull on an primitive level or split each triangle individually. The first option is often preferred as it is simpler to implement, especially when the frustum is not static, and the spared network and GPU bandwidth might not counter-balance the additional CPU cost of a more precise culling method. To avoid any network communication of the graphics primitive, the viewer can also be replicated locally on each rendering node (figure 3).

When using multiple viewers, each one describing a part of the scene, merge filters are responsible for combining their messages together (figure 4). This scheme is useful for parallel data extraction applications, which we will show in section 5.2. It is also necessary for recombining parts of an application that uses different distribution schemes. In this case, the different streams are merged together before being forwarded to each renderer.

We now consider remote rendering systems. The renderer can be placed at the same location as the viewer, pixels being communicated to the remote site (figure 5(a)). It can alternatively be moved to the display location. In this case, the graphics primitive descriptions are transmitted instead (figure 5(b)). A more interesting case is when both are combined (figure 5(c)). In a scene composed of streamlines and volumetric rendering for instance, the streamlines

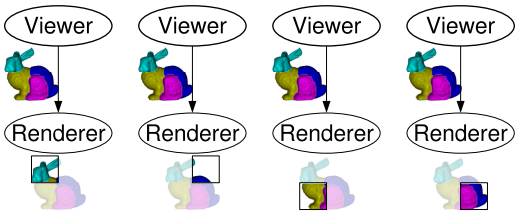


Figure 3: Replication: Viewer application is replicated on each rendering node. If the application is non-determinist then communications will be necessary to keep copies synchronized.

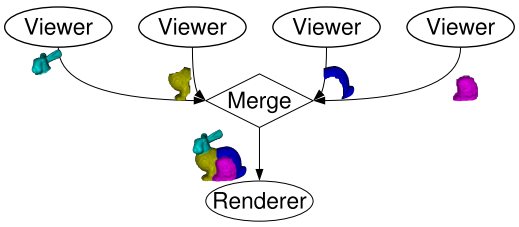


Figure 4: Parallel scene description: To distribute the load in viewers, several viewers can describe parts of the scene which are then merged together.

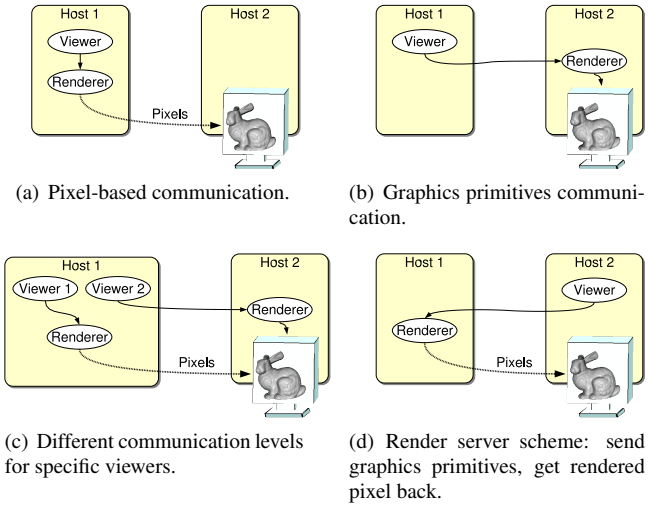
could be transmitted over the network and rendered with a high resolution, while the volume image will be rendered locally and transmitted at a lower resolution. Another combination happens when the viewer and renderer are distant but the final display is on the viewer side (figure 5(d)). This scheme corresponds to the *render server* system, where rendering is provided as a distant shared resource. By using existing image composing tools [**?**, **?**] we can use several servers in parallel for increased performance.

4.3 Interactions and Scene Management

Having a distributed scene description controlled by unrelated programs can lead to difficulties in obtaining a coherent manipulation interface. In monolithic applications, this is often done through the scene graph structure, where the user can change parameters of the nodes (visibility, wireframe, ...) and control transformation nodes with special widgets to move/rotate/scale objects. To implement this within our framework we face two issues: as the communication protocol is strictly one-way, how can viewers get the user's actions; and as the scene description model forbids dependencies between primitives, how can we specify that a group of primitives should be moved through a common transform node.

User inputs depend on the interaction devices (mouse, keyboard, VR tracking devices, ...). Reading inputs data is done by the renderers for mouse/keyboard inputs, or by special tools for other devices. As the data is small, broadcasting the inputs over the network is simple [**?**, **?**]. However interactions are often expressed in terms of objects in the scene (i.e. a user selected object with a specific ID). As we already have a globally unique ID associated with each primitive, renderers can output events containing these IDs. Viewers can then use this information to manage interactions.

As described at the beginning of this section, basic interactions (moving objects, changing rendering attributes, ...) are often not handled by the objects themselves but by other components in the scene (widgets and transform nodes). The same functionality can be achieved by adding custom filters in the data-flow network to transparently change the affected parameters. If required, additional *widgets* viewer programs can be introduced, adding objects in the scene and retrieving interaction events on these objects, eventually sending resulting movements to the transform filters. This architecture can be seen as the data-flow based mapping of the equivalent scene-graph hierarchical structure. Thus while the scene description model does not allow for interdependencies between primitives, we can implement the equivalent functionalities by adding filters in the network.



(a) Pixel-based communication.

(b) Graphics primitives communication.

(c) Different communication levels for specific viewers.

(d) Render server scheme: send graphics primitives, get rendered pixel back.

Figure 5: Remote rendering.

4.4 Implementation

FlowVR Render architecture can be implemented using many different communication API, such as TCP connections or CORBA objects. We chose to use FlowVR [**?**] as it provides a clean abstraction to design interactive data-flow applications. At the lowest level it transparently uses either TCP connections for network communications or shared-memory for local communications. It also provides tools to launch and control distributed applications, as well as generating large filters networks using scripts.

Current implementation of FlowVR Render can be downloaded from `http://flowvr.sf.net/render/`.

5 Applications

In this section we present experimental results using FlowVR Render, VTK (version 4.2) and Chromium (version 1.8). VTK and Chromium have been chosen because they are probably the most used and advanced tools publicly available today in their category. We used them without in-depth code tunings and optimizations. A version of VTK has been tuned for Chromium [**?**], but it is not used here as it is not publicly available.

While we present performance comparisons between Chromium and FlowVR Render, the reader should keep in mind that their conditions of use are different. Chromium supports all OpenGL applications without modification, while FlowVR Render proposes a new shader based rendering framework that requires application to be adapted.

Tests were performed on the GrImage[1] platform composed of a cluster of 16 Dual-Opteron 2.0 GHz having 2 GB of memory each. Cluster nodes are interconnected by a gigabit Ethernet network. Each node uses an NVIDIA Geforce 6800 Ultra graphics cards to drive a 4×4 display-wall with a total resolution of 4096×3072.

[1] `http://www.inrialpes.fr/grimage/`

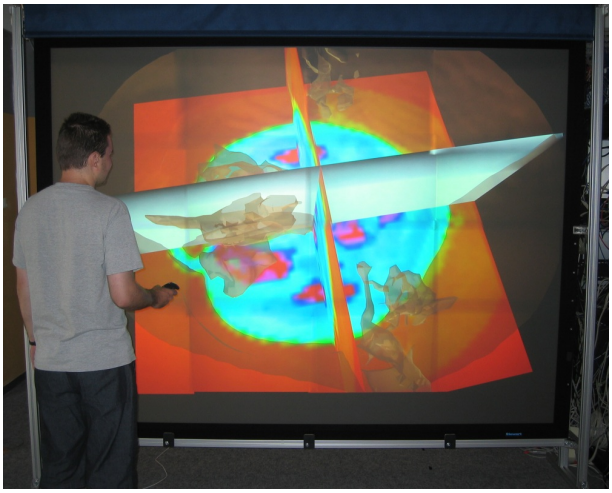


Figure 6: An example VTK application rendering on the display wall with FlowVR Render.

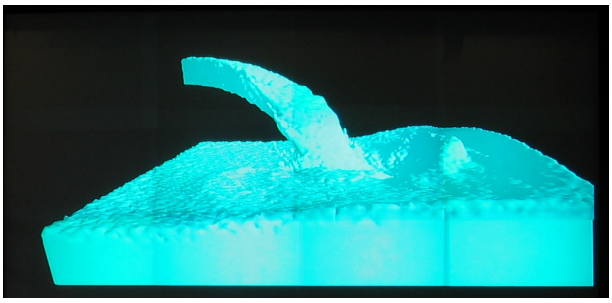
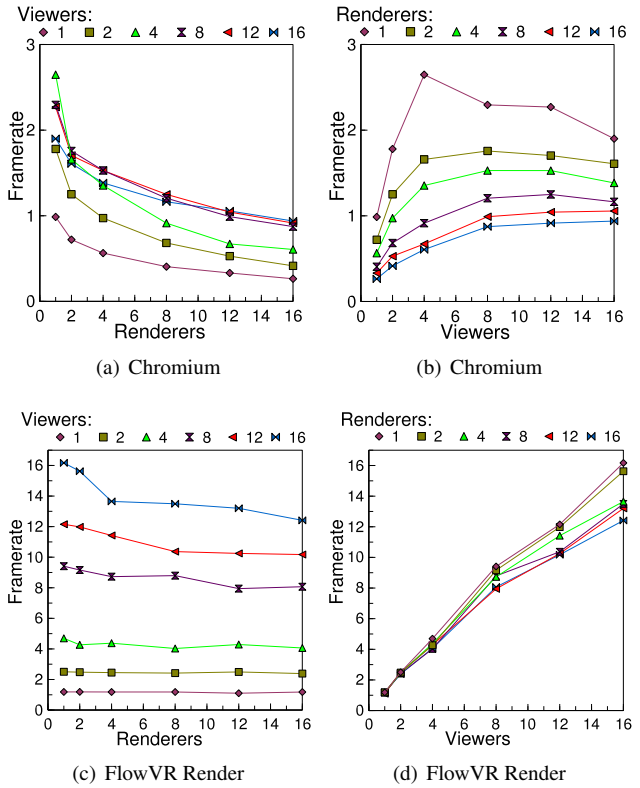


Figure 7: Iso-surface extracted from a frame of a time-varying fluid simulation dataset.



(a) Chromium (b) Chromium

(c) FlowVR Render (d) FlowVR Render

Figure 8: Parallel iso-surface extraction with sort-first rendering, using Chromium (a)-(b) or FlowVR Render (c)-(d). Scalability regarding the number of renderers is presented on the left, while scalability regarding the number of viewers is shown on the right.

5.1 VTK and FlowVR Render Integration

Several powerful open-source visualization toolkits are available such as OpenDX [**?**] or VTK [**?**]. We developed a library that transparently replaces VTK rendering classes to use FlowVR Render instead of OpenGL. VTK uses a small set of rendering classes to draw images, meshes (with a list of points, lines and triangles) or volumes. The original rendering classes use OpenGL immediate mode commands or display lists. These are complex classes to handle all possible combinations of vertex data as well as legacy OpenGL drivers. The new FlowVR Render classes are simpler as they often only consist in encapsulating the raw data into FlowVR Render resources and selecting the right shader. Unmodified VTK applications can then be rendered on a display-wall as shown in figure 6. Developing this library only took a couple of weeks for one programmer, most of which was spent learning VTK.

5.2 Parallel Iso-surface Extraction

To compare the performance of our framework with Chromium we implemented a parallel iso-surface extraction application. We used a 3D fluid simulation dataset of $132 \times 132 \times 66$ cells for 900 timesteps (one timestep is shown in figure 7). The application interactively extracts and displays an iso-surface (containing approximately 100000 triangles) for each timestep. The iso-surface extraction is parallelized by splitting the dataset into blocks, each one assigned to a different viewer.

Figure 8 presents Chromium and FlowVR Render performance results depending on the number of renderers as well as the number of iso-surface extraction viewers. FlowVR Render outperforms Chromium and shows a better scalability, both while increasing the number of renderers and the number of viewers. FlowVR Render achieves 12 frames per second with 16 data viewers and 16 renderers to display the result on the 4×4 display-wall. Chromium performance is probably affected by the high overhead related to culling and stream merging operations.

Using pixel shaders can also greatly improve visual quality. For this application, a classic OpenGL-based lighting is evaluated per vertex, while it is computed per pixel with shaders. The resulting surface appears smoother.

5.3 Volume Rendering

The second test pushes further the use of shaders to highlight the benefits of our approach. For that purpose we focus on sort-first parallel volume rendering. Notice that usually sort-last approaches have proved more efficient than sort-first approaches [**?**]. We show that using hardware shaders can significantly improve performance of sort-first algorithms based on the following points:

- Due to the massively parallel nature of todays GPUs, pixel shaders have access to more important resources, both in terms of memory bandwidth and computing power [**?**].
- Shaders are able to apply transfer functions to raw volumetric data to obtain the final color and opacity. This allows to send the raw data once, and then only update the transfer function when it is modified by the user. For time-varying datasets this is also interesting as the raw data can be 4 times smaller

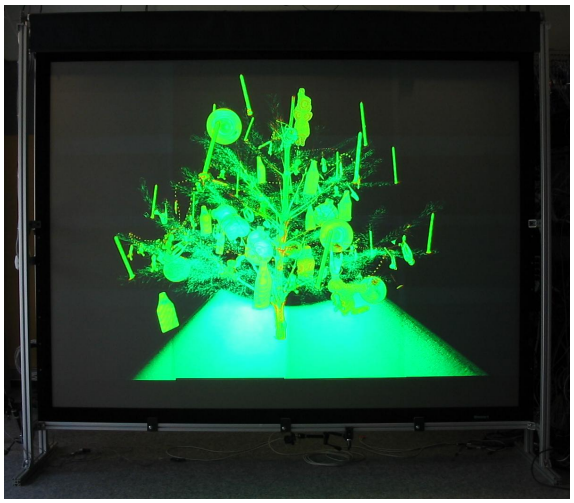


Figure 10: Volume rendering on the display wall.

than final colors and opacity data (1 value per voxel versus 4 values).

- Using pre-integrated transfer functions [**?**] and adaptive sampling steps [**?**], a shader can create a very high quality image while using fewer steps through the volume, allowing to use larger datasets.

By default VTK uses a slice-based 2D texturing approach for hardware volume rendering. We implemented a new VTK volume rendering class using shaders. A pixel shader is used to cast a ray through the volume and accumulate color and opacity using either blending, additive, or maximum value composing. As this composing is done in temporary registers instead of the frame buffer, it stays in full 32-bit precision and it saves the bandwidth required to write and read-back framebuffer values in traditional multi-pass approaches.

To implement the pre-integrated transfer function, we use a 2D texture that, given the previous and current density value, stores the color and opacity produced by integrating all intermediate densities in the transfer function. This greatly improves visual quality for high frequency transfer functions and allows for much larger sampling steps for smoothly varying datasets.

As this application is only limited by the fill-rate of the graphics cards, we used a simple broadcast distribution scheme where everything is sent to all renderers.

Renderings obtained using the VTK original slice-based 2D texturing and FlowVR Render-based raycasting shader with and without preintegration are shown in figure 9. The data set is a $512 \times 512 \times 512$ Christmas tree [**?**]. Performance results are presented in table 1. As a comparison, VTK 2D texturing implementation achieved 0.18 frames per second on one display. This is mostly due to the fact that without shaders full-color textures must be used instead of the raw grayscale texture. In our case this means that VTK had to reupload the data at each frame as it does not fit inside the graphics card.

Rendering on the display wall instead of only one display does not introduce significant overhead as the transfered data is small for most frames (camera position and transfer function). We even obtain better performance on the display-wall as coherency between neighbor pixels is higher. It leads to a more efficient texture caching inside the graphics cards.

Notice that a higher framerate may be obtained during interactions (when the camera is moving for example), by decreasing the rendering resolution in addition to the sampling resolution. It permits to obtain fluid movements while keeping a reasonable quality.

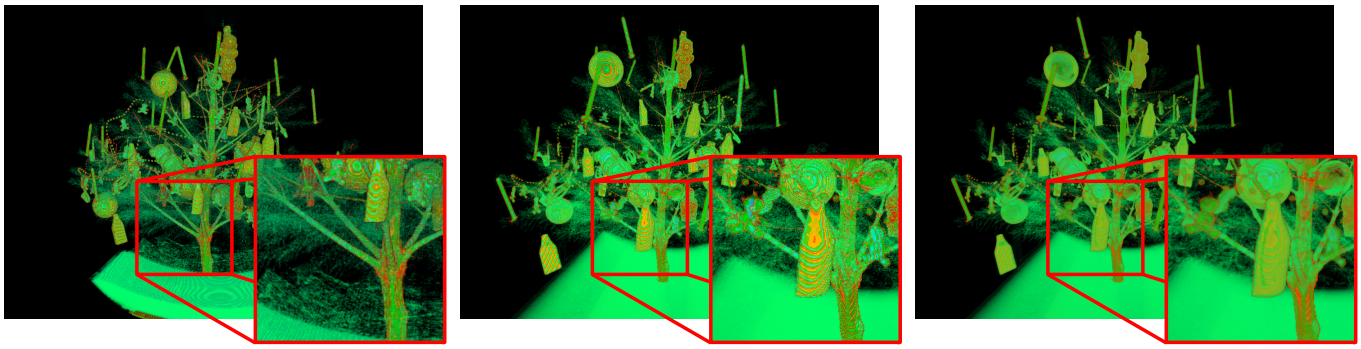
6 Conclusion

In this paper, we presented a novel parallel and remote rendering framework using a scene abstraction based on batches of polygons and shaders. This framework proved to be efficient and scalable while using simple enough concepts to be easily extensible. Although not directly compatible with existing applications in opposite to Chromium, the porting effort should usually be limited. In the case of visualization applications using a common toolkit this effort only has to be made once, with the additional benefit of providing access to advanced features using shaders.

The iso-surface extraction test application showed that the FlowVR Render approach outperforms Chromium regarding performance and scalability. The volume rendering application showed that using shaders and a communication protocol based on incremental changes significantly reduces the amount of data communicated over the network. Using raw data sets instead of final colors and geometry also reduces the memory requirements on the graphics card, allowing larger datasets on each node. In particular, we achieved an interactive rendering of a $512 \times 512 \times 512$ dataset on a 4096×3072 display wall.

Notice that the experiments presented focused on direct rendering on a display wall. FlowVR Render can also be used for remote rendering in conjunction with sort-last algorithms.

Future works will address the design of a more complete toolkit to manage user interactions. We will also extend FlowVR Render to support a dynamic level-of-detail. Viewers will send several versions of primitives (either reducing the number of vertices or the shaders quality), and renderers will adapt the rendering resolution and quality of the objects depending on the desired performance.



(a) VTK 2D texture slices (b) Raycasting shader (c) Raycasting shader with preintegration

Figure 9: Christmas tree dataset with different rendering methods (512 sampling steps per pixel).

Method	Sampling steps	Framerate on 1 display Resolution 1024×768	4×4 display-wall 4096×3072	4×4 display-wall 2048×1536	4×4 display-wall 1024×768
Raycast Shader	512	1.16	2.25	5.40	8.21
Pre-Integrated Raycast Shader	512	1.10	2.04	4.97	7.70
Pre-Integrated Raycast Shader	200	2.79	5.14	12.44	19.11

Table 1: Volume rendering performances with a $512 \times 512 \times 512$ dataset.