Parallel Large Data Visualization with Display Walls

Luiz Scheidegger[a], Huy T. Vo[b], Jens Krüger[c], Cláudio T. Silva[b] and João L.D. Comba[d]

[a]Facebook, 1601 S California Ave, Palo Alto, CA, USA;
[b]Polytechnic Institute of New York University, Six MetroTech Center, Brooklyn, NY, USA;
[c]Saarland University, 66123 Saarbrücken, Germany;
[d]Universidade Federal do Rio Grande do Sul, Instituto de Informática - Porto Alegre/RS, Brazil

ABSTRACT

While there exist popular software tools that leverage the power of arrays of tiled high resolution displays, they usually require either the use of a particular API or significant programming effort to be properly configured. We present PVW (Parallel Visualization using display Walls), a framework that uses display walls for scientific visualization, requiring minimum labor in setup, programming and configuration. PVW works as a plug-in to pipeline-based visualization software, and allows users to migrate existing visualizations designed for a single-workstation, single-display setup to a large tiled display running on a distributed machine. Our framework is also extensible, allowing different APIs and algorithms to be made display wall-aware with minimum effort.

Keywords: display walls, visualization pipelines

1. INTRODUCTION

In recent years, scientific data has become large and complex enough to require specifically designed software tools for visualization. Moreover, the amount and complexity of different visualization techniques available introduce a bottleneck on the effectiveness of the data exploration process. *Visualization systems* simplify the creation and management of visualizations, making them accessible to a wider audience.[1–5] In a typical session in these applications, a user interacts with one or more visualization specifications, usually encoded as a set of modules and connections, referred to as *visualization pipelines*. Visualization systems adequately address the problem of managing complex visualization sessions, but most systems do not interact with tiled display walls natively. Instead, a great amount of time and effort must be spent to port existing visualizations to a multi-display setup.

As datasets increase in complexity, single-display systems become progressively less effective. Tiled displays have gained popularity as easy to build low-cost devices which can extend available screen space.[6,7] They can be constructed in several ways,[8] but in this work we restrict ourselves to arrays of high-resolution LCD panels. Such composite displays have become the consensus solution for low-cost, high resolution devices, in large part because of a discrepancy between the growth of computing power and the increase in single-display resolution. Over the last 20 years, computing power, storage density and communication bandwidth have increased by at least three orders of magnitude. Screen resolution, on the other hand, has merely doubled.[9] In addition, there is evidence that physically large displays offer psychological advantages in tasks that demand spatial orientation.[10]

Despite their low cost, tiled displays are often a scarce resource. Therefore, *efficient use* of the displays is important: if it takes a large programming or configuration effort to create a visualization in a display wall, less time is available for other users. Hence, solutions that allow easy migration to and from large displays are particularly attractive.

In this paper, we introduce PVW, a framework designed to allow parallel visualization of dataflow systems in multi-display configurations. PVW works as a plug-in to existing visualization software, and automatically converts single-display visualizations to work on tiled displays. It also manages a network distribution layer and captures user interaction events, such as mouse gestures, to allow users to naturally interact with a visualization in a tiled display environment. Our method works by modifying the structure and parameters of user-generated visualizations to port them to a tiled display.

Further author information: (Send correspondence to Luiz Scheidegger)
Luiz Scheidegger: E-mail: luiz.scheidegger@gmail.com

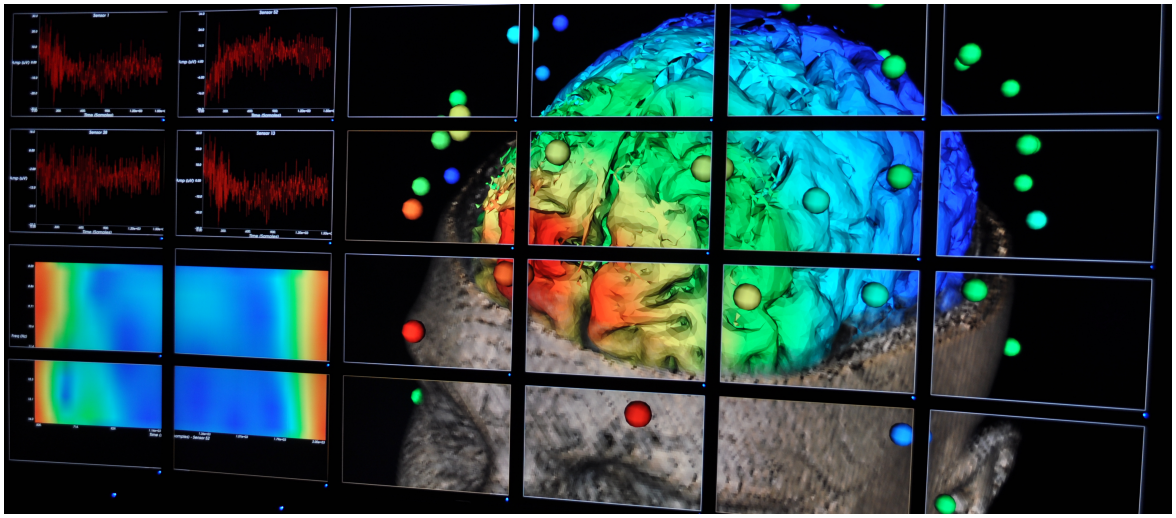


Figure 1. Display walls can be used to inspect complex data in high resolution. PVW allows users to combine the flexibility of existing visualization tools with the power of high resolution tiled displays. It supports non-uniform viewports in the display wall, which is useful to analyze multi-modal data like above.

1.1 Contributions

PVW is a simple object-oriented framework to empower parallel visualization in tiled display systems. In particular, it offers the following contributions:

- A plug-in to automatically enable visualization systems to interact with a tiled display wall;
- A simple, object-oriented framework to extend PVW to different APIs;
- A network layer to manage the display wall's distributed environment;
- User input capture to allow a natural transition between single- and multi-display interaction.

2. RELATED WORK

There are many visualization systems available,[1–3,5] as well as visualization programming libraries, such as VTK.[11] Upson et al.'s AVS is one of the pioneering systems to use pipeline-based visualizations.[4] Jankun-Kelly et al. describe a formal definition of the visualization process, providing an XML-like specification of a pipeline.[12] The ability to fully describe a visualization in this is way is important as it will later allow us to serialize pipelines before they are sent across different machines via local network.

VisTrails[5] is a scientific visualization tool that combines concepts such as a module/connection-based pipeline paradigm and a spreadsheet-like interface for multiple-view visualization. Moreover, it automatically tracks pipeline provenance through a version tree and efficiently caches intermediate computation results. Spreadsheets are very popular in the visualization literature, as they allow for simultaneous visualization of different aspects of the same data, or effective comparison between different datasets.[13] Similar ideas are also used in the information visualization community.[14] All of these works share the concept of using a visual spreadsheet to analyze two dimensions of a high-dimensional parameter space by using the spreadsheet as an orthogonal cutting plane through this space. The Hyperwall system specifically implements comparative visualization through slices of the parameter space presented across a display wall.[15] As we will show in Section 5, PVW can be used to automatically display these parameter studies on a tiled display wall. ViSUS (Visualization Streams for Ultimate Scalability)[16] is a system designed to allow progressive visualization and processing of large datasets. We use it to display extremely high-resolution images on our display wall in real time. ViSUS works by constructing a cache-oblivious multi-resolution representation of the data. This representation is kept out-of-core, and the system progressively loads only the image pixels that are visible in the current viewport. The cache-oblivious

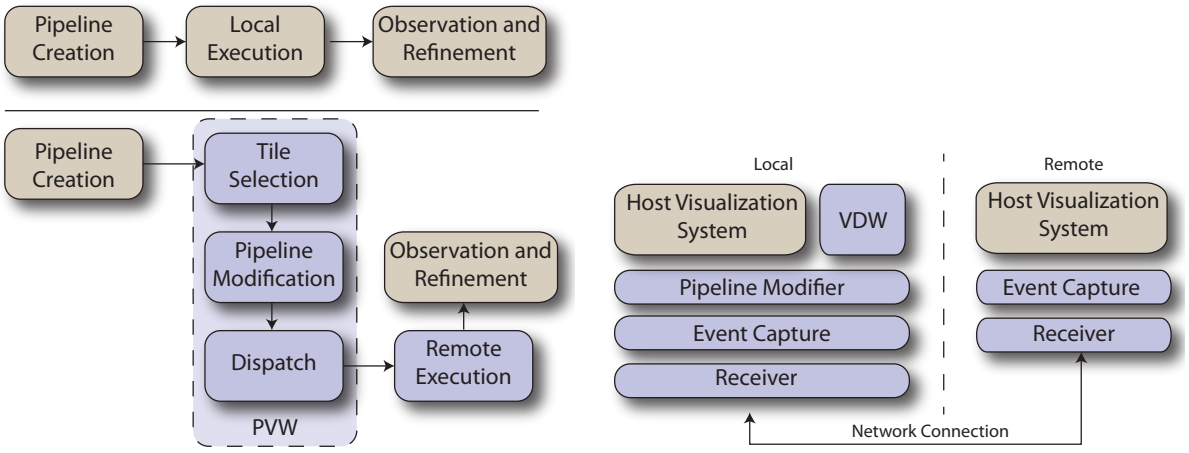


Figure 2. (a) A visualization session conforms to a simple creation, execution and observation workflow (top). PVW modifies this workflow by adding the tile selection, pipeline modification, and dispatch stages (bottom). Moreover, the local execution is replaced by a remote execution on the display wall cluster. (b) PVW is designed as a set of layers around the host system, responsible for analyzing and modifying the user's pipeline and managing a lightweight network protocol. This allows deploying PVW as a plug-in to existing tools.

layout of the data (based on a discrete space-filling curve) ensures that the loading can be done efficiently and with low latency.

There is extensive work that enables the construction and operation of display walls.[6,8,9] Several parallel visualization systems have been tailored to this specific application,[17,18] while other distributed systems can be adapted to use a high resolution tiled display.[19–21] Such distributed solutions are appealing when the applications have been written with the correct API. For general issues regarding large display technologies, software environments and applications, we refer the reader to the surveys in.[8,9]

3. PVW FRAMEWORK

In a typical session with visualization systems, a user conforms to a simple workflow: the user creates a visualization pipeline, executes this pipeline locally and observes the results in a single display. PVW modifies this workflow by adding a *tile selection*, a *pipeline modification* and a *dispatch* stage, where the local machine automatically analyzes, modifies, and finally sends the visualization pipelines for remote execution on a tiled display wall (Figure 2(a)). There is no added effort to the user, since new stages are triggered automatically.

3.1 Overview

PVW is designed as a modular, object-oriented plug-in to existing visualization systems, which we refer here as the *host* program. PVW requires a running instance of the host program in a local machine (referred to as the *master* machine), as well as separate instances of the host program in the cluster of machines driving the display wall (referred to as the *slave* machines). Several choices arise when building and configuring a cluster of machines to drive a tiled display wall, including the number of machines, the number of LCD panels, and the particular machine-panel correspondence. PVW is oblivious to these choices, as it can be customized to meet the requirements of different setups via a configuration file. Communication between the master and slave machines is done via a lightweight TCP protocol, which automatically allows a physical separation between the master machine and the slave nodes. As we will see in Section 3.4, this TCP-based protocol allows mobile devices such as smartphones to interact with PVW, enhancing user experience and navigation.

To be minimally intrusive, PVW is implemented as a set of layers around the host program (Figure 2(b)). Such layers are responsible for capturing the user's visualization pipelines, analyzing and modifying their structure and parameters, capturing input events, and managing a simple network protocol.

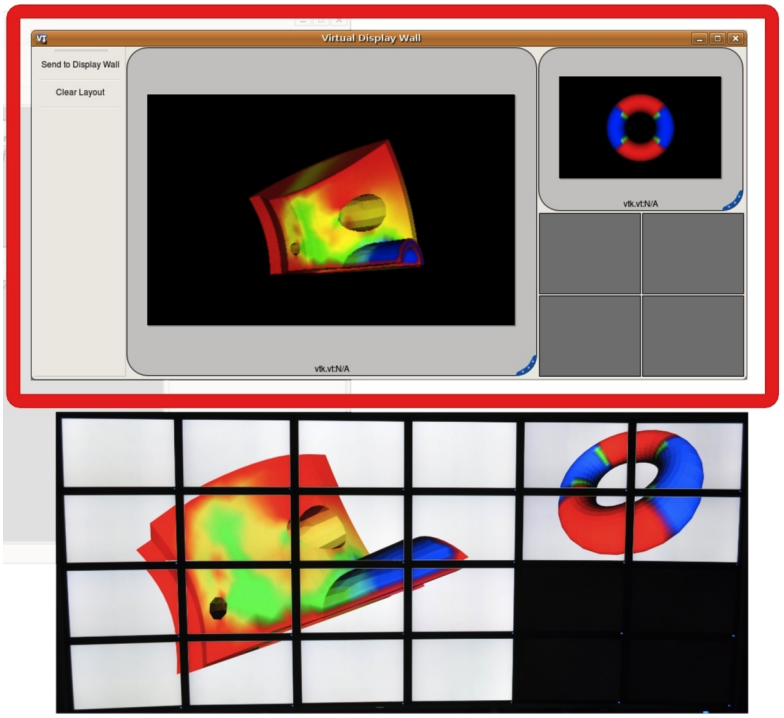


Figure 3. A simple drag-and-drop interface allows the user to select rectangular regions on the display wall in which to execute visualizations. Our system uses these regions to determine how many copies of the original pipeline must be made, as well as where in the display wall the results must be placed.

3.2 Pipeline Parsing and Modification

In order to display a visualization on a tiled display wall, the user must select which portion of the wall to occupy with the visualization. PVW natively supports non-uniform tilings of simultaneous visualizations, which is very useful for analysis of multi-modal data (see Figure 1). This specification is done via a simple graphical user interface, the *Virtual Display Wall* (VDW). The VDW turns the selection of display wall tiles into a simple drag-and-drop process, as shown in Figure 3. This specification uniquely associates a rectangular set of display wall tiles with each local pipeline. Moreover, each individual LCD panel in the wall runs one instance of the host program. To display a single visualization across multiple panels, the original pipeline must be replicated as many times as the number of tiles in the selected region, and each copy must be modified so that its viewing parameters correspond exactly to the tile's coverage of the original viewport (see Figure 4).

PVW does this by navigating the structure of the original pipeline, finding where viewport parameters need to be modified, and performing these modifications automatically, via the `PipelineModifier` class. This class has a single public method, `replicatePipelines`, which receives the original pipeline, as well as its corresponding list of display wall tiles, and returns a list of modified pipelines ready for remote execution. Moreover, it contains protected methods designed to facilitate navigation over the topology and parameters of pipelines. In effect, these methods define PVW's API.

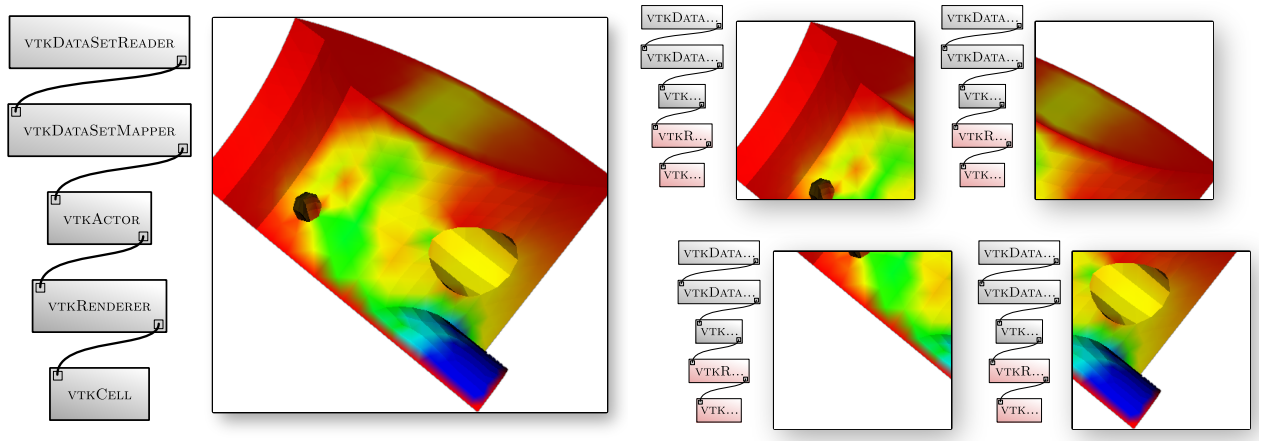
Since different visualization APIs have different callbacks and parameters, `PipelineModifer` is a purely virtual class that must be inherited for each particular API in the visualization system. Inherited classes must implement the protected method `modifyPipeline`, which receives a copy of the original pipeline, as well as a description of one of the tiles in the Display Wall, and returns the modified pipeline that can be executed on its particular display wall tile. When each modified copy is executed in its corresponding LCD panel, the result is a coherent, multi-display visualization of the original pipeline. As an example, we have implemented the `VTKPipelineModifier` and `VISUSPipelineModifier` subclasses, to support the VTK and VISUS libraries, respectively. Support for other APIs requires inheriting new classes from `PipelineModifier`. Listing 1 shows pseudo-code for the classes discussed above.

```
class PipelineModifier:
  # modifies a copy of the original pipeline for
  # execution on tile t
  void modifyPipeline(Pipeline &p, Tile t);

  Pipeline[] replicatePipelines(Pipeline p,Tile[] tls):
    Pipeline[] result = new Pipeline[]
    for tile in tls:
      Pipeline p = new Pipeline()
      modifyPipelines(&p)
      result.add(p, tile))
    return result

class VTKPipelineModifier: PipelineModifier
  void modifyPipeline(Pipeline &p, Tile t):
    # parse the input pipeline and modify its structure
    # and VTK parameters to fit only to tile t
    # ...
```

Listing 1. Public interface for the `PipelineModifier` and `VTKPipelineModifier` classes, used to modify the structure and parameters of pipelines in the display wall. PVW supports arbitrary APIs by allowing to inherit their own classes from `PipelineModifier`.



(a) (b)

Figure 4. Dataflow pipelines contain modules and parameters to display a visualization on a single screen (a) PVW creates one copy of the pipeline for each tile in the selected display wall region, and modifies its parameters (in the highlighted modules) to display portions of the original viewport (b).

The pipeline replication stage runs locally on the master machine. Once all new pipelines are ready, PVW must dispatch them to the display wall cluster for remote execution. This uses our lightweight network layer, described in the next section.

3.3 Network Layer

PVW contains a small TCP-based network protocol that provides communication between the master and slave machines. This communication is used to dispatch the modified pipelines, as well as to forward user interaction events. Although well established distributed computing solutions such as MPI exist, we found that a simple TCP messaging protocol developed specifically for PVW was a simpler solution.

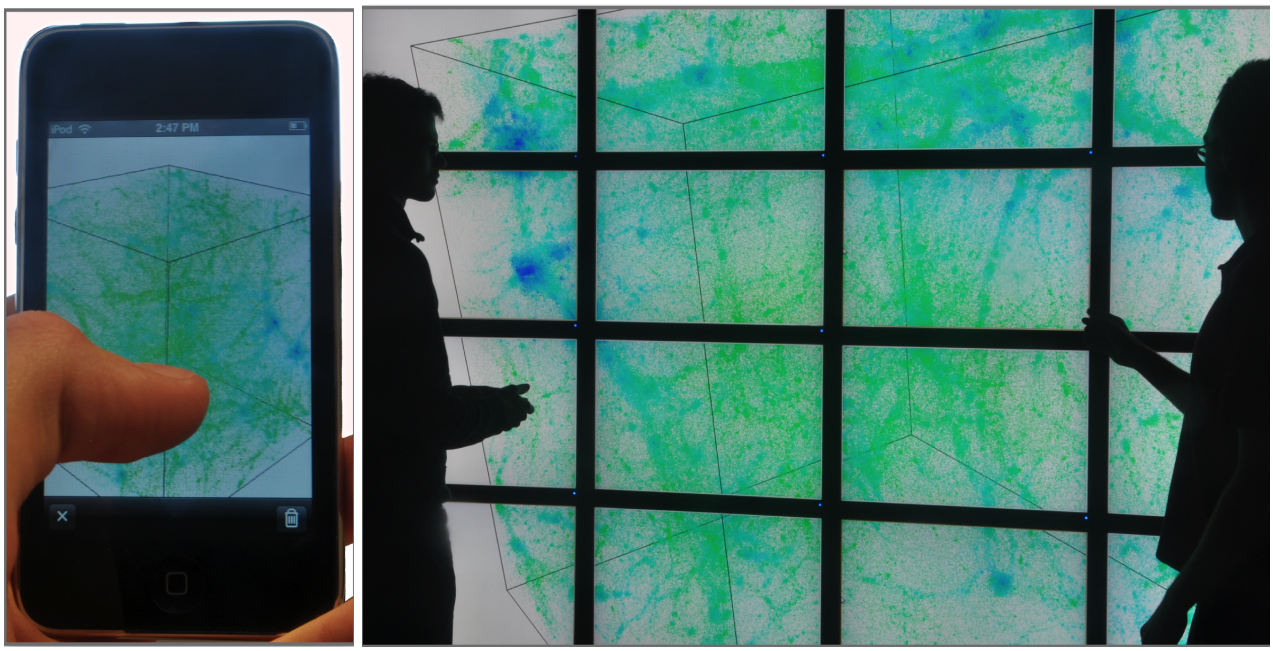


Figure 5. Using a smartphone such as an iPhone the user can control the parameters that define the visualization sent to the display wall.

The communication protocol defines two layers, the *dispatcher* and the *receiver* layers, respectively. The dispatcher layer runs on the master machine, and exists to establish a persistent TCP connection between the master and the display wall cluster. The receiver layer, on the other hand, runs underneath the host program on each node in the display wall cluster, waiting to receive the modified pipelines for execution.

After the pipeline modification stage, PVW serializes all pipelines and sends an XML message to each slave node in the remote cluster, containing the full pipeline description. The receiver layer captures these messages, and executes the modified pipelines on the host program instances running on the display wall cluster. The execution of the modified pipeline results in a multi-display image of the original visualization.

Most visualization systems offer some kind of real time interaction, where the user can navigate through the dataset being analyzed. To support interaction on the Display Wall, PVW employs an event capture layer, described below.

3.4 User Interaction

In a single-workstation, single-display setup, users typically interact with the visualization system using the mouse and keyboard. PVW natively supports these interactions using Synergy, an open source application that allows sharing of mice and keyboards across multiple displays and machines. The user interacts with the mouse on the master, while the cursor is placed directly on the display wall's tiles. The event capture layer running on the slave machines captures mouse events, and sends them to the master. The master redirects these events back to the other slave machines, ensuring that all nodes receive an instance of the event. Since all nodes run independent versions of the host program, they process events in parallel, thus allowing a consistent user interaction.

Although mouse navigation can be accomplished using Synergy, it is impractical to expect the user to sit behind the master machine, possibly far away from the display wall, to interact with the system. To circumvent this, we implemented a simple event forwarding system that allows events generated from mobile devices such as smartphones to be interpreted as mouse events, that also get processed by the event capture layer. This allows users to navigate the visualization using a handheld device, which provides the necessary freedom to be physically near the tiled display wall (Figure 5).

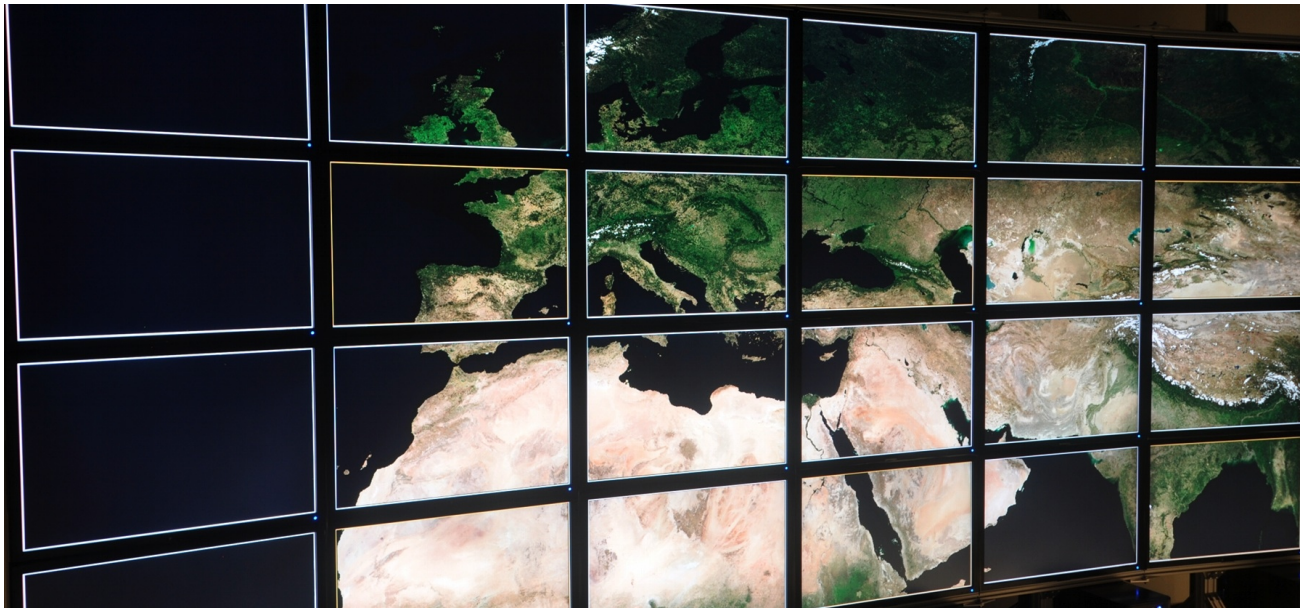


Figure 6. ViSUS allows real-time visualization and exploration of extremely high-resolution images. Here we show a composite of satellite images of the Earth totaling 13 gigabytes of data. Each display is running a separate instance of the application, with the data mounted on a shared file system.

4. EXPERIMENTAL SETUP AND RESULTS

The PVW framework is written entirely in Python, as a plug-in solution to VisTrails, a popular, open-source scientific visualization system. In conducting our experiments, we constructed a tiled display consisting of 24 30-inch LCD panels (each with a resolution of 2560×1600 pixels) arranged in a six-by-four fashion (see Figure 1). These displays are connected to six slave nodes, each of which controls a two-by-two square on the wall. All the slave nodes, as well as the master node, are identical desktop machines. The nodes are equipped with two NVidia GeForce 9800GXII GPUs, allowing each of them to connect to four separate screens.

We use this particular configuration because of its high resolution (we achieve a total of approximately 96 megapixels) and comparatively low cost (all components are consumer grade items). However, there are many possible alternative configurations for a display wall, which may include a larger or smaller number of panels as well as nodes. PVW is easily customized to fit different scenarios via a configuration file.

We implemented the `VTKPipelineModifier` and `VISUSPipelineModifier` classes, which provide display wall functionality for the VTK and VISUS libraries, respectively. VTK is a highly comprehensive visualization library, and most common visualization tasks can be performed using its functionality. VISUS is a tool that allows interactive visualization of extremely large images via a streaming approach. We were able to interact with images containing up to 300 megapixels in resolution, using VISUS (Figure 6)).

Very large images are excellent candidates to be displayed on a high resolution tiled display. We integrated ViSUS into our system by defining a small set of new pipeline modules and by writing a new pipeline modifier class. The new modules provide a thin translation layer between the ViSUS library (written in C) and the underlying visualization tool. The new class modifies the ViSUS pipelines in a similar way to what we described in Section 3.2: it transforms the viewport specified as a parameter to the ViSUS modules through a scale and a translation. This ensures that the viewport covers only the appropriate tile for each copy of the pipeline.

Since ViSUS uses the viewport information to stream coherent data from out-of-core memory, our approach will only access the parts of the data visible from the tiled display. This ensures that users can inspect arbitrarily large images on the tiled display without sacrificing interaction performance.

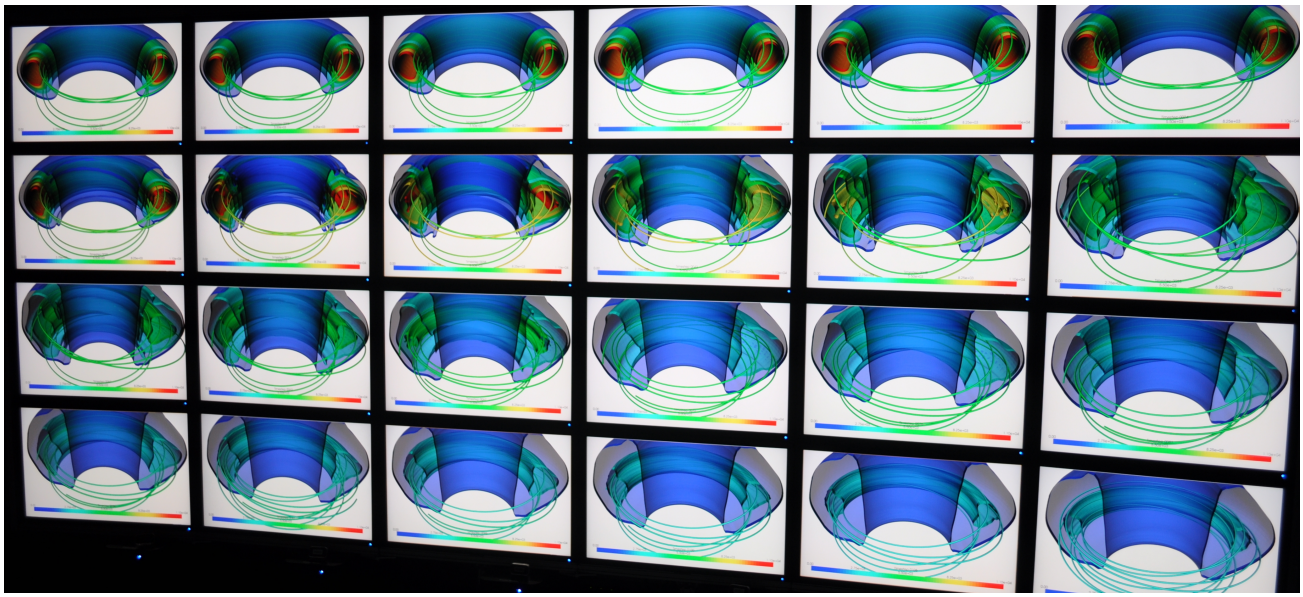


Figure 7. Parameter exploration of 24 time-steps of the Tokamak fusion simulation. The large screen allows users to inspect different parameter configurations of a dataset side-by-side in high resolution. Isosurfaces represent regions of equal temperature, with streamlines traced along the magnetic field.

5. APPLICATION SCENARIOS

Parameter explorations One notable feature present in many scientific visualization applications is the ability to conduct *parameter explorations*. A parameter exploration consists of exploring a two-dimensional slice of the parameter space by picking visualizations that span this subspace and showing them side by side. There is experimental evidence that the notion of using *small multiples* has a positive impact on the user's ability to notice trends in data.[5] A simple application of our system is to enable parameter explorations directly on the display wall, so that each parameter combination corresponds to a separate panel. In this scenario, a user can visualize simultaneously as many points in parameter space as panels present in the display wall, as can be seen in Figure 7. This feature is also present in comparable systems, such as the Hyperwall project.[15] While this is not a particularly advanced example, it shows how PVW can be orthogonally integrated with other features present in visualization tools.

Electron Microscopy Mosaics Electron microscopy mosaics combine thousands of individual microscope scans into a single high-resolution image. Using ViSUS, we experimented with a large mosaic of the retina of a rabbit's eye, obtained from scientists at the John A. Moran Eye Center. Cytological images such as these contain interesting features at many different scales, which makes them very suitable for visualization on a display wall. With this abundant screen space, features of vastly different sizes can be inspected simultaneously, facilitating the analysis process. Although it is possible to visualize this data on a single-monitor configuration, important context is lost if the user has to constantly pan and zoom the visualization to compare interesting features.

6. DISCUSSION

PVW works under the assumption that the user has already designed a set of visualizations, which must now be presented on a tiled display wall. Furthermore, we allow the user to do this in a completely transparent way; no modification of the pipelines is necessary. Instead, PVW automatically detects the API being used by the pipeline, and invokes the correct `PipelineModifier` subclass. The main advantage of this approach is that users can design pipelines on their local workstations, and migrate them to a high resolution wall only when they are satisfied with the visualization. This greatly improves usage efficiency of the display wall, which is typically a

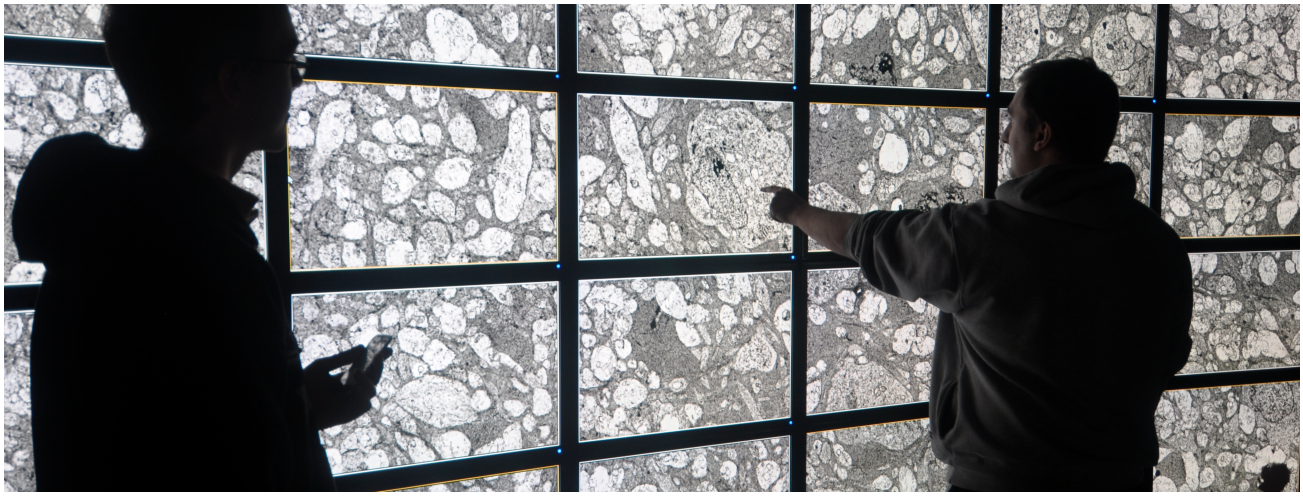


Figure 8. One challenge in interacting with retinal microscopy data is that objects of interest have different scales, and the size of the composite images is typically larger than the memory available. PVW allows a streaming technique to display results on a large wall, where scale differences are more manageable.

shared resource within an institution. With PVW, no time must be spent, while actively using the display wall, to properly configure the visualizations.

Our work is based on copying and modifying a set of visualization pipelines. An important limitation of our method is the fact that we do not explicitly address the issue of data bricking in any way. Instead, we chose to leave data management tasks to the particular algorithms that can be plugged into our system. Other issues that may help in increasing performance for large datasets, such as geometry culling against the specified viewport, are also left to the individual modules in the visualization. The main contribution of our system is the fact that if individual modules provide a solution to a particular problem on a single-screen context, our display wall extension will be able to take advantage of this solution transparently. Furthermore, since we do not modify the execution engine present in the visualization tool, the only performance overhead imposed by our system is a constant-time pipeline modification step before the visualization is performed. We can observe a strong correlation between the performance of our system and the computing requirements of individual modules used in a visualization.

PVW differs from other parallel rendering and visualization tools in several key aspects. Systems such as Chromium[20] and Equalizer[22] provide a powerful way to develop multi-machine OpenGL applications. However, these tools require that the visualization consumer write specific code to take full advantage of parallelism and tiled displays. PVW moves this burden away from the end user — as long as the visualization package being used (such as VTK) supports the `PipelineModifier` class, the end user can design visualizations with no regard for the underlying multi-machine, multi-display setup. The migration from single display to the tiled screens is done automatically. Naturally, this solution requires visualization package developers to implement the `PipelineModifier` class interface, but we believe that package developers are better suited to tackle this challenge than domain scientists producing the visualizations themselves.

The SAGE[23] system, on the other hand, relies heavily on pushing pixels through a high-speed, ultra low-latency network. This maximizes flexibility, since any application rendering to a framebuffer can be forwarded through SAGE. Its main drawback is that it requires expensive high-speed network hardware. Our design of VDW achieves a balance between flexibility, where multiple visualization systems and packages can be used, and deployment cost, as we rely on off-the-shelf hardware components for the display wall.

6.1 Synchronization and Frame-locking

Since our technique delegates the bulk of the rendering workload to the slave machines in the distributed system, synchronization issues naturally arise. For instance, depending on the viewpoint, a visualization may place a

high rendering load on some machines, while leaving others mostly idle. Without any treatment, this can lead to very noticeable frame rate artifacts between adjacent panels controlled by separate nodes.

In order to remedy this problem, we have implemented a simple frame-locking mechanism. We added hooks into the underlying visualization system's rendering loop so that each tile is first rendered to an offscreen framebuffer. The screen is not updated at this time, however. After rendering, the slave nodes send synchronization events to the server, and wait for a response before displaying their framebuffers. The server, on the other hand, will only dispatch this response once it has received synchronization events from all panels in all the slaves. At this point, all the completed framebuffers are simultaneously displayed on the screens.

Although this technique requires adding code to the visualization system's rendering loop, we believe this is not an unrealistic requirement, since virtually all modern graphics cards natively support duoble-buffered rendering. Furthermore, we have implemented the synchronization protocol through UDP, in order to minimize latency. The main drawback of this solution is that, despite all screens being refreshed simultaneously, the frame-rate of the entire visualization will be limited to that of the worst-performing node. We have conducted experiments and observed that disabling synchronization can provide a smoother interaction with the system in some situations. Therefore, we allow the user to enable or disable this feature at any time.

7. CONCLUSIONS AND FUTURE WORK

We have presented a method to seamlessly integrate scientific visualization systems with large arrays of high resolution displays. We require little configuration of the display wall, and the user does not need to directly modify his visualizations to migrate them to the tiled display. We also allow advanced users or developers to design and seamlessly integrate third-party rendering algorithms with our solution.

In the future, we would like to extend the system to work on ad-hoc projection displays and similar devices. Because the pipeline transformation approach described in Section 3.2 is general, the adjustments to the modules should be very similar. To further popularize tiled displays, we envision package writers exposing domain-specific transformation operations for their modules through our API, allowing the packages to work on a tiled display while keeping user input at a minimum.

ACKNOWLEDGMENTS

We would like to thank Allen Sanderson (SDM) for the Tokamak fusion simulation dataset, Katrin Heitmann (LANL) for the cosmology simulation dataset, and John Schreiner for the MRI dataset. We also thank the ViSUS team and the scientists at the John A. Moran Eye Center for the microscopy dataset. We thank Erik Anderson, Michael Bellem, Thomas Fogal, Juliana Freire, Phillip Mates, Valerio Pascucci and Carlos Scheidegger for their help on the development of our system. This work was supported in part by the National Science Foundation awards CNS-1153503, IIS-1153728, OCI-0904631, OCI-0906379, IIS-1045032, IIS-0844572, CNS-0751152, CCF-0702817, the U.S. Department of Energy BER and ASCR, and an NVIDIA Fellowship. João Comba is supported by CNPq (processes 200498/2010-0, 569239/2008-7, and 491034/2008-3). The work was also made possible in part by the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10 and by Award Number R01EB007688 from the National Institute of Biomedical Imaging and Engineering, as well as the Intel Visual Computing Institute. The content is under sole responsibility of the authors.

REFERENCES

[1] SCIRun: A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI).

[2] Kitware, "ParaView." http://www.paraview.org.

[3] IBM, "Ibm opendx." http://www.research.ibm.com/dx.

[4] Upson, C., Thomas Faulhaber, J., Kamins, D., Laidlaw, D. H., Schlegel, D., Vroom, J., Gurwitz, R., and van Dam, A., "The application visualization system: A computational environment for scientific visualization," *IEEE Comput. Graph. Appl.* **9**(4), 30–42 (1989).

[5] Bavoil, L., Callahan, S., Crossno, P., Freire, J., Scheidegger, C., Silva, C., and Vo, H., "Vistrails: Enabling interactive, multiple-view visualizations," in [*Proceedings of IEEE Visualization 2005*], Silva, C., Rushmeier, H., and Gröller, E., eds., 135–142, IEEE Press (2005).

[6] Humphreys, G. and Hanrahan, P., "A distributed graphics system for large tiled displays," in [*VIS '99: Proceedings of the conference on Visualization '99*], 215–223, IEEE Computer Society Press, Los Alamitos, CA, USA (1999).

[7] Funkhouser, T. and Li, K., "Large format displays," *IEEE Comput. Graph. Appl* **25**(4), 20–21 (2000).

[8] Ni, T., Schmidt, G. S., Staadt, O. G., Livingston, M. A., Ball, R., and May, R., "A survey of large high-resolution display technologies, techniques and applications," in [*Proceedings of IEEE Virtual Reality Conference*], (2006).

[9] Wallace, G., Anshus, O. J., Bi, P., Chen, H., Chen, Y., Clark, D., Cook, P., Finkelstein, A., Funkhouser, T., Gupta, A., Hibbs, M., Li, K., Liu, Z., Samanta, R., Sukthankar, R., and Troyanskaya, O., "Tools and applications for large-scale display walls," *IEEE Computer Graphics and Applications* **25**(4), 24–33 (2005).

[10] Tan, D. S., Gergle, D., Scupelli, P., and Pausch, R., "Physically large displays improve performance on spatial tasks," *ACM Trans. Comput.-Hum. Interact.* **13**(1), 71–99 (2006).

[11] Schroeder, W., Martin, K., and Lorensen, B., [*The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*], Kitware (2003).

[12] Jankun-Kelly, T. J., Ma, K. L., and Gertz, M., "A model for the visualization exploration process," in [*VIS '02: Proceedings of the conference on Visualization '02*], 323–330, IEEE Computer Society, Washington, DC, USA (2002).

[13] Levoy, M., "Spreadsheets for images," in [*SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*], 139–146, ACM, New York, NY, USA (1994).

[14] hsin Chi, E. H., Riedl, J., Barry, P., and Konstan, J., "Principles for information visualization spreadsheets," *IEEE Comput. Graph. Appl.* **18**(4), 30–38 (1998).

[15] Sandstrom, T. A., Henze, C., and Levit, C., "The hyperwall," in [*CMV '03: Proceedings of the conference on Coordinated and Multiple Views In Exploratory Visualization*], 124, IEEE Computer Society, Washington, DC, USA (2003).

[16] Pascucci, V., "ViSUS: visualization streams for ultimate scalability," Technical Report UCRL-TR-209775, Lawrence Livermore National Laboratory (LLNL), Livermore, CA (February 2005).

[17] Corrêa, W. T., Klosowski, J. T., and Silva, C. T., "Out-of-core sort-first parallel rendering for cluster-based tiled displays," *Parallel Comput.* **29**(3), 325–338 (2003).

[18] Humphreys, G., Buck, I., Eldridge, M., and Hanrahan, P., "Distributed rendering for scalable displays," in [*Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*], 30, IEEE Computer Society, Washington, DC, USA (2000).

[19] Brodlie, K., Duce, D., Gallop, J., Sagar, M., Walton, J., and Wood, J., "Visualization in grid computing environments," in [*VIS '04: Proceedings of the conference on Visualization '04*], 155–162, IEEE Computer Society, Washington, DC, USA (2004).

[20] Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P. D., and Klosowski, J. T., "Chromium: a stream-processing framework for interactive rendering on clusters," *ACM Trans. Graph.* **21**(3), 693–702 (2002).

[21] Yang, J., Shi, J., Jin, Z., and Zhang, H., "Design and implementation of a large-scale hybrid distributed graphics system," in [*EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*], 39–49, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2002).

[22] Eilemann, S., Makhinya, M., and Pajarola, R., "Equalizer: a scalable parallel rendering framework," in [*ACM SIGGRAPH ASIA 2008 courses*], *SIGGRAPH Asia '08*, 44:1–44:14, ACM, New York, NY, USA (2008).

[23] Naveen, K., Venkatram, V., Vaidya, C., Nicholas, S., Allan, S., Charles, Z., Gideon, G., Jason, L., and Andrew, J., "Sage: the scalable adaptive graphics environment."