Volumetric Windows: Application to Interpretation of Scientific Data, Shader-based Rendering Method, and Performance Evaluation

Christoph W. Borst[1], Vijay B. Baiyya[1], Christopher M. Best[1], Gary L. Kinsland[2]
[1]Center for Advanced Computer Studies, [2]Department of Geology
University of Louisiana at Lafayette; Lafayette, LA, USA

ABSTRACT - *We present a framework, example application, and rendering methods for volumetric windows, a 3D analog of the 2D windows metaphor. A set of parameters reflected in both implementation and user menus allows windows to behave like 3D volumetric lenses, clip windows, world-in-miniature views, and in other ways. We use programmable hardware shaders to render volumetric windows and we investigate performance of both shader and clipping plane methods. The evaluation considers different dataset and view features as well as the use of data surface tiling for speedup. We apply our work to interactive data viewer windows for managing multiple datasets or multiple views of a dataset in a scientific visualization application.*

Keywords: volumetric windows, volumetric lenses, Magic Lenses, clipping planes, shaders, 3D exploration

1 INTRODUCTION

This paper presents a system of 3D windows that we refer to as "volumetric windows", as a generalization of "volumetric lenses" from [1]. Simply stated, a volumetric window is an interactive box containing 3D information. It may act, for example, as a tool box, a lens, or a data viewer. We use the term "windows" rather than "lenses" because not all of the supported behaviors are intuitively described as lenses and because we intend the name to encompass possible behaviors that extend beyond earlier work (e.g., a window for docking other windows).

Existing 3D visualization systems often include at least one feature that could be considered a volumetric window. For example, a dataset may be rendered in an axis-aligned box with adjustable sides that restrict the visible range, or in a box that always orients itself to face the viewer (as in Crumbs [2]). Other techniques that can be considered volumetric windows are volumetric lenses [1, 3, 4], clipping volumes (e.g., [5]), and world-in-miniature techniques [6]. In the case of a volumetric lens, the environment is rendered differently in a 3D lens region than in the normal view, and this is seen as a generalization of earlier 2D lens techniques from [7], which have also been generalized to flat lenses in 3D space [8-10]. In clipping volumes, a volume can be used to "cut out" a portion of the environment. In certain world-in-miniature techniques, a small 3D duplicate of the environment is rendered in a small volume near the user, and the user can reach into this volume to interact.

In contrast to the earlier cited work, we discuss:

1) Implementation with programmable hardware shaders.

2) A combination of window rendering methods with a tiling-based speedup technique. As tiling is similar in essence to a class of widely-used speedup techniques, results can be expected to generalize to some other applications.

3) A performance analysis that evaluates both the shader-based method and a standard clipping planes method. It considers scene complexity, polygon fill demands, and tiling granularity.

4) A single framework for windows that behave like lenses, clipping volumes, or worlds-in-miniature, and that readily extends to new behaviors such as collaborative windows. Although these may be seen as types of volumetric lenses, they are usually treated individually in other work, or merely suggested as possible future variations. Our parameters guide both implementation and user control for a general window type.

2 APPLICATION AND MOTIVATION

2.1 Application Overview

Our work is motivated by an ongoing investigation of VR-based techniques for interactive exploration and interpretation of scientific datasets. In preliminary work [11, 12], we applied a prototype system to interpretation of features in topographic and associated geophysical datasets over the Chicxulub Impact Crater on the Yucatán Peninsula. The crater, now buried by carbonates that both preserved and obscured its structure, is the result of an impact with an asteroid or comet [13, 14] and is widely believed to be the "end Cretaceous event". One of the co-authors has worked with related gravity and topographic datasets since 1993.

Visualized data seen in this paper include Shuttle Radar Topography Mission (SRTM) data giving elevation grids, Bouguer gravity values krigged with Surfer software, elevation extracted from gravity surveys, a shoreline description of latitude-longitude pairs, and a profile of latitude-longitude-elevation triples collected with a GPS receiver.

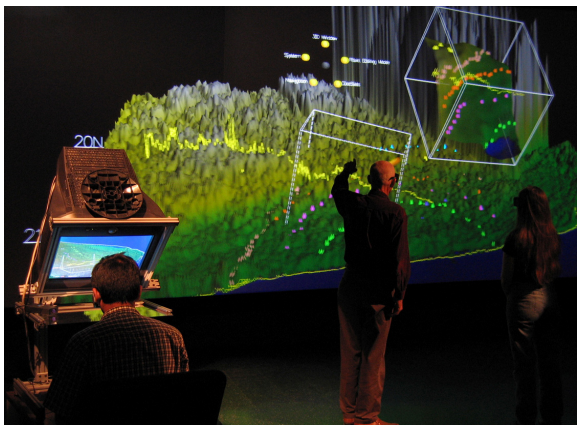

Figure 1: Networked use of our system in a mirror-based "fishtank" VR configuration (left) and a projection system.

The software supports a basic set of visual rendering options, interaction techniques, and rudimentary networked operation. It has been used with HMD, mirror-based "fishtank" [15], and large-scale projection systems. Figure 1 shows networked use of our system by a fishtank user and users of a projection system. We previously demonstrated networked use in which two users control local views independently and see pointers and view proxies representing remote users [11].

Elevation and gravity datasets seen in this paper were rendered as triangle meshes with height-based color mapping and per-vertex illumination. Texture mapping is supported to texture a mesh with satellite images or with the coloring from a second dataset. A user-selectable averaging filter allows mesh reduction for performance or user preference, and the system allows coloring and shading from the highest-detail mesh to be textured onto reduced-detail meshes to visually preserve some detail.

Users can interact with menus and volumetric windows using either ray- or point-based techniques, navigate by walking, grab and move the world, and place/edit interpretive markers on the datasets. For HMD and projection displays, a tracked gamepad is used for input, and a PHANTOM is used for the "fishtank".

2.2 Interpretation Task and Preliminary Observations

The geological interpreter's main interest was to explore these datasets and annotate extrema and ridges (e.g., to mark low points corresponding to limestone sinkholes). During this process, the user wanted to view interpretive markers remapped onto other datasets to check for consistency of features between datasets.

During interpretation with an earlier system that lacked windows, much time was spent manipulating viewpoints, changing filtering options, and switching between datasets. When examining minima, the user sometimes preferred to adjust vertical scale to a large-magnitude negative value or flip a mesh upside-down to interpret it from the bottom. Before adding volumetric windows, we observed long sequences of such manipulations as the user sought the best marker placement.

Although a user study has not yet been conducted and is beyond the goals of this paper, preliminary informal observations suggest a potential usefulness of volumetric windows. A user spends time setting up windows for viewing multiple datasets or multiple views, but this subsequently reduces frequent switching or manipulation of datasets when checking for correspondence between them. The ability to place and edit markers is enhanced because the user can reach into multiple views to adjust marker placement. Choice of preferred window parameters (discussed next) varies depending on the type of VR system used. We give examples later, after discussing the design.

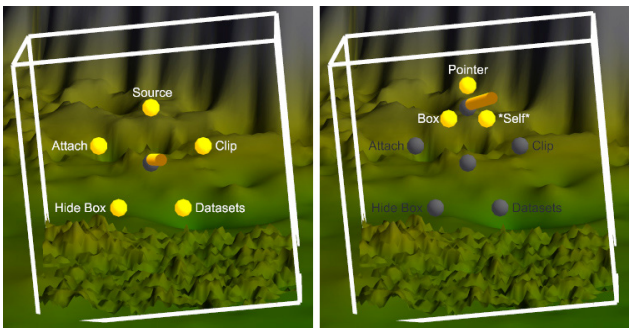

Figure 2: 3D menu system for a volumetric window

3 Volumetric Windows For Data Viewing

3.1 Design Overview

We identified a set of parameters for a 3D data viewer window that behaves in several useful ways. These parameters are:

Contents: A reference to the data/objects to be rendered in the box along with rendering options to be applied.

Source: A parameter selecting the data region represented by box contents, used to compute a geometric transform applied to contents. Three possible values are:

- *Pointer*: The box contents correspond to a region around the user's current pointer tool.
- *Box*: Another box in the scene selects the region.
- *Self*: No special control; the box's actual extents select the region.

Attachment: Attaches the box to a selected coordinate system. Possible attachments are:

- *Room*: The box maintains its position with respect to the real room in which the user is located.
- *User*: The box is attached to tracked user head or body, as sometimes found in world-in-miniature techniques.
- *Dataset*: The box is attached to an object in the scene – in this case, a dataset.

Clip: A flag to indicate whether or not the box clips the exterior environment (the interior is always clipped).

Automation: Activates algorithms that influence window or content transformations. For example, vertical data offset and scale could be automatically adjusted to make full use of the box's range. Or, an orientation coupling mechanism could keep two views aligned.

These parameters are reflected in both system implementation and menu design. A user accesses parameters with a 3D menu system, illustrated in Figure 2, in which subitems are arranged in a ring. When the box is selected and a menu button is pressed, the menu appears near the user's controller, facing the user. To avoid problems of datasets obscuring menu parts, data surface translucency is simulated for obscured items as follows: first, after other scene elements are rendered, the menu is rendered with some transparency and with depth buffering disabled, and then it is rendered again normally. The same menu system is used for other system features, including visualization options, marker editing, file browsing, and initial creation of a window.

Besides the menu, the user can interact with the volumetric windows more directly. Using a clutch button on a tracked controller or PHANTOM, combined with ray or point-based selection, a user can grab and pose a window's box as desired. If the user points to the corner of a box, indicators for resizing appear and the box can be resized with a press of the same button and controller motion.

A volumetric window becomes active (gains focus) when a user points to it or reaches into it. A change in visualization parameters is appropriately applied only to the contents of the currently active view. For example, to apply a mesh filter only to the data view in a window, the user points to that window when selecting a filter using a menu button. Certain elements such as markers and a visual pointer tool for editing them are visible in all views. So, when a user reaches into a box to place a marker,

the pointer tool also appears in any other view that represents the same region being edited in the box, and changes to marked points are reflected in all views. This occurs because all views reference the same markers and pointers as part of their contents.

3.2 Examples of Data Viewer Windows

We give examples of window behaviors using frames from a video demonstration (*http://www.cacs.louisiana.edu/~cborst/vw*). Interactions were performed in the fishtank with a PHANTOM.

Figure 3 illustrates a volumetric window with *clip* on and *source* set to *self*. This window acts like a basic interactive volumetric lens: the window filters the region inside it by applying different rendering options or displaying different data. For example, a high-resolution mesh could be seen inside the lens while a low-resolution version is seen outside. In the figure, the window shows topography in the selected region while the main view shows corresponding geophysical data (gravity).

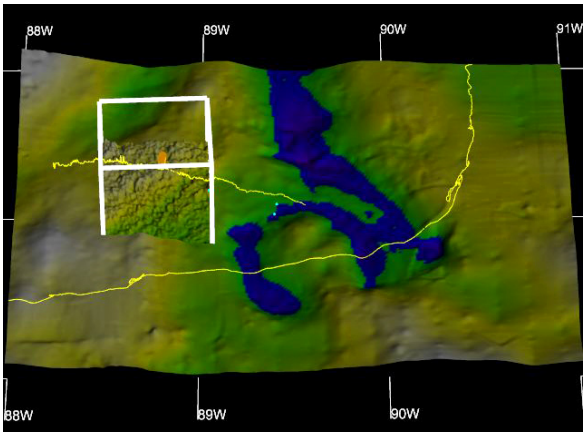

Figure 3: Volumetric window as a volumetric lens. The main view shows gravity while the lens shows SRTM data.

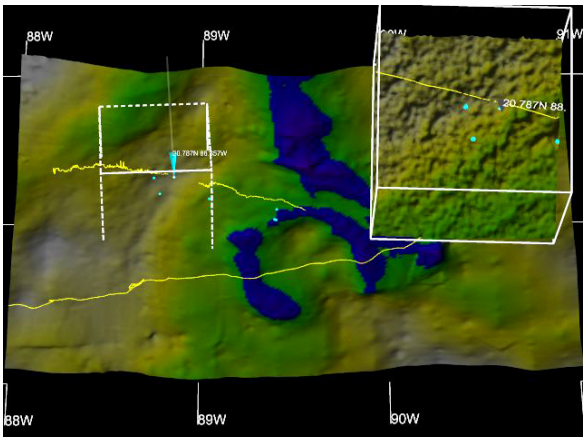

Figure 4: Volumetric window giving an alternative view of a region selected by a dotted source box

In Figure 4, the *source* has been set to *box* and the window has been scaled and posed to the right. A dotted box is drawn to illustrate the source region, and a user can reach into either the window or the source box (or main view) to perform interactions. In the figure, the user is reaching into the source box and using a point editing tool to place interpretive marks. The marks and a representation of the point editing tool are present in both boxes. This allows a user to manage two very different views of an environment in the same scene during interpretation. The specific illustrated interactions were performed in the fishtank display, where *attachment* for this type of use would normally be set to *room* so the window remains fixed at the top right of the viewing surface during transformation of the main view (translation, scale, etc.). For use with a tracked head mounted display, the corresponding effect (window appearing fixed relative to display surface) instead requires a *user* attachment. The dotted source box would typically have a *dataset* attachment, so that the volumetric window continues to represent the same dataset region. A source box can support some of the same interactions and options as a window (and therefore share implementation elements), but it has no contents of its own and simply indicates a 3D region. The source box must move when a user reaches into the associated volumetric window to grab and move an interior mesh, to reflect the change in displayed region.

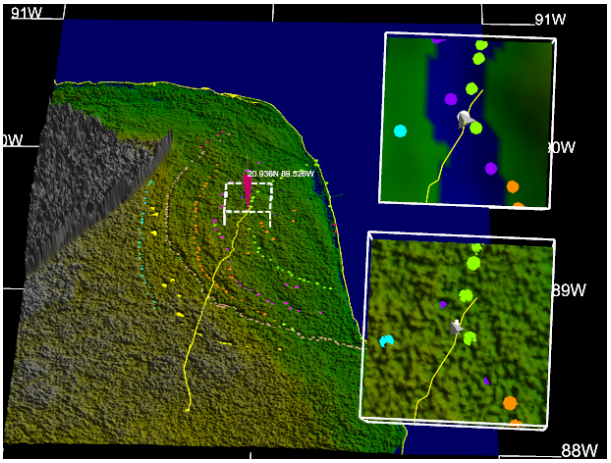

Figure 5: Source region automatically following a user's pointer during placement of markers. The main view shows topography while two windows give zoomed-in views of topography and gravity for the tracked pointer region.

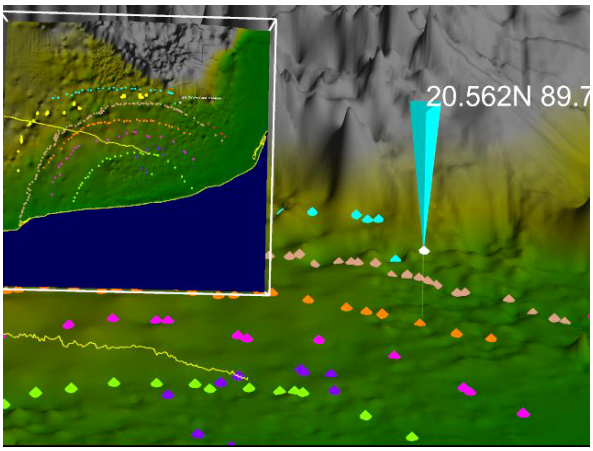

Figure 6: Volumetric window displaying a miniature 3D view of the world. The illustrated dataset contains elevation data extracted from gravity surveys.

For figure 5, *source* has been set to *pointer* (additionally, two volumetric windows are present). So, a dotted source box always

follows the user's marker editing tool so the volumetric windows at the right always display views of the specific region being interpreted. Besides showing additional datasets or viewing angles, this is useful for viewing context in the main view and a zoomed-in view of the region being interpreted in the windows (magnification). In this type of viewing, when a user reaches into a volumetric window, the source box does not move into the window itself, since resulting behavior would not be well-defined. Thus, the pointer-follow mechanism selects the source region without considering target windows. For collaborative use, the source box could follow a remote pointer (telepointer).

Figure 6 shows the volumetric window used as a miniature 3D duplicate of the world. A user can manipulate objects such as point markers both in the main view and in the window. Parameters are similar to those in Figure 4, but the source box encloses the entire dataset and the main view is zoomed in, so the window minifies a source instead of magnifying it.

4 Rendering Techniques and Performance

4.1 Volumetric Window Rendering and Data Tiling

We investigated both the use of hardware clipping planes and the use of programmable hardware shaders to render volumetric windows, as well as their combination with dataset tiling. The clipping planes method is known from early work on volumetric lenses [1]. More recently, researchers have used depth peeling [3], which simulates dual depth buffers but introduces aliasing artifacts. Simple depth buffer and stencil buffer tricks that have been used for planar lenses or windows are not applicable. Performance analysis in existing related work is minimal.

We describe the techniques for a window that has *clip* enabled. With *clip* off, simplified versions of these techniques can be used. For clarity, this section also assumes only one volumetric window is rendered. Section 4.3 considers generalizing this.

Clipping planes method: The clipping planes technique uses seven passes that can be summarized as follows: One pass renders interior window contents while clipping to the box boundary using six hardware clipping planes. The outer region (the main view) is then partitioned into six regions defined by subsets of the same planar boundaries. These can be described as left of the box (one plane), right of the box (one plane), in front of the box but not in left or right (3 planes), behind the box but not in left or right, above the box but not in any previous region (5 planes), and below the box but not in any previous region.

Shader method: The main idea of our shader technique is to use a fragment shader (pixel shader) to discard fragments that are either inside or outside the window boundaries, depending on whether the interior or exterior scene is being drawn. A vertex shader is also involved, to provide coordinates for the fragment shader in a convenient coordinate system without per-fragment transformations. We implemented shaders in GLSL, but describe them conceptually to minimize language-specific presentation.

For simplicity and efficiency of the fragment shader, assume the fragment includes a 3D attribute that is a coordinate describing its position with respect to a coordinate system having origin at the volumetric window's center and principal axes parallel to the window box edges. Assume a window size is also passed to the fragment shader (specifically, one half box size in each dimension). This shader then simply compares absolute values of coordinates against box size to determine if a fragment is inside or outside the box. Fragments not in the desired region are discarded.

To provide the needed coordinate to the fragment shader, a vertex shader sets a per-vertex attribute and standard hardware performs the linear interpolation needed to find per-fragment values from this. The vertex shader computes the per-vertex attribute by applying a transform to the regular vertex coordinate. The transform, passed by the main program, is the homogeneous transform describing the mesh coordinate system with respect to the box-centered coordinate system, where the mesh coordinate system is the local coordinate system in which mesh vertices were described when display lists were built.

Tiling techniques: Since techniques such as tiling, spatial partitioning, and bounding volumes are found in many graphics systems, it is useful to consider such a technique in an evaluation of achievable window performance. We used tiling to reduce the portion of the scene traversed by different rendering stages.

The datasets are rendered using stripified triangle meshes in display lists cached by the graphics card. Our tiling approach divides the data into a set of smaller display lists, enclosed in hierarchical bounding boxes, organized in a manner that resembles region quadtrees, as illustrated in Figure 7.

With the clipping planes approach, for each rendering pass, these bounding boxes are used to cull tiles that would be completely clipped out by the active planes. Except when a user is geometrically transforming a window box or dataset (or has linked it to a dynamic external control element such as pointer following), the culling test results are constant. So, the list of tiles remaining after culling is cached for re-use. While a user grabs and moves a window, some lists must be recomputed for each frame, incurring extra cost. In the case of pointer-following, only the list for the interior window region needs to be recalculated. Similarly, for geometric transformations that only affect one of the views, only a subset of lists needs to be recalculated.

With the shader approach, the tiling system is again used to cull out certain tiles and cache tile lists when possible. In this case, one list references tiles that lie completely inside the window: these can be rendered entirely, without the shaders. Another list references tiles intersected by window sides: these are rendered with the shaders discarding exterior fragments. For the scene outside the window (which generally differs from the interior scene and is also tiled), there is a list of tiles completely outside the window and a list of tiles intersected by box sides – the first is rendered without the shader active and the second is rendered with the shader discarding interior fragments.

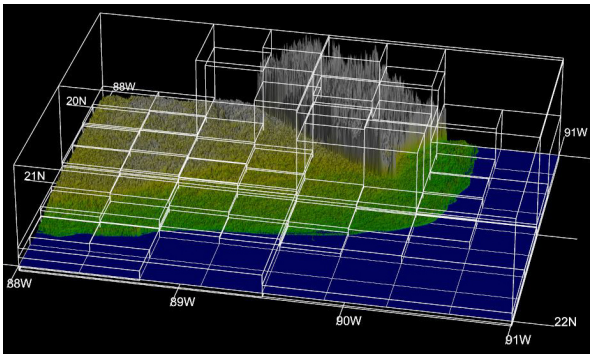

Figure 7: All bounding boxes (hierarchical) for 64-tile data

4.2 Performance Analysis

4.2.1 Methods

We evaluated performance cost of rendering a volumetric window with both clipping plane and shader approaches, and with varying dataset complexity and tiling resolution.

Our first measurements were taken with settings chosen to minimize polygon fill time: lighting was precomputed and

therefore disabled on the graphics subsystem, and no texture mapping or antialiasing features were used. This tends to emphasize performance costs of the extra passes and overhead described in Section 4.1. However, we also include a case with increased fill time as could often be found in real use, and we note differences in resulting costs of rendering windows.

The scenes used for performance evaluation are shown in Figure 8. The main view and the volumetric window have the same contents, to avoid side effects of differences in dataset complexity. The window is roughly centered on the mesh and tilted diagonally to avoid special cases where boundaries would be parallel to tile divisions or where some culling steps would eliminate all tiles. The illustrated dataset has 1801 × 1201 vertices and we also measured performance for a reduced-detail mesh with 721 × 481 vertices. We rendered all scenes on a Dell Precision 690 with two dual-core Xeon 5140 processors, 2 GB main memory, and an Nvidia-based Quadro FX 4500 graphics card. Display resolution was fixed at 1280 × 1024.

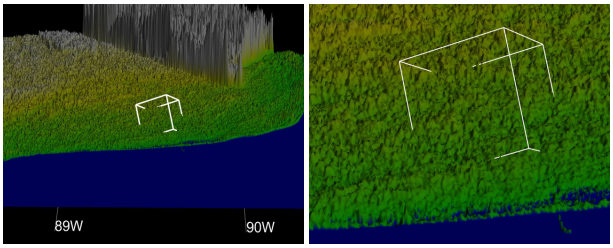

Figure 8: Zoomed-out and zoomed-in views of a window, used for performance analysis of rendering techniques

4.2.2 Measurements Obtained

For each dataset resolution and view, we present measurements of frame rate against tiling resolution for the following conditions:

Baseline: no volumetric window was rendered (effects were due to tiling only).
Shader: the window was rendered using the shader method.
Clip planes: The window was rendered with the 7-pass hardware clipping planes method.
Moving, shader: Tile lists from the culling step were not cached, to simulate the worst-case requirements for arbitrary window manipulation (this case can often be avoided in practice). The shader method was used.
Moving, clip planes: Like the previous case, but for the clipping planes method.

We first measured frame rate with varying tiling granularity for the 1801 × 1201 version of the dataset. Results are summarized in Figures 9 and 10.

As illustrated in Figures 11 and 12, we took similar performance measurements for the reduced-detail version of the data. This was to check a relationship between scene complexity and performance costs by comparison to the previous results.

Since the previous measures were taken with settings that minimize fill time (e.g., precomputed lighting), we repeated the measurements for the high-resolution dataset with 16X anti-aliasing enabled. This increases fill time, as would texturing, lighting, very high resolutions, or more complex shading. Results are shown in Figures 13 and 14.

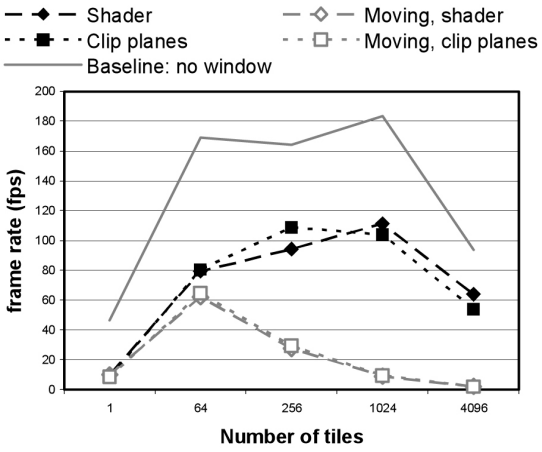

Figure 9: Performance for zoomed-in 1801 × 1201 dataset

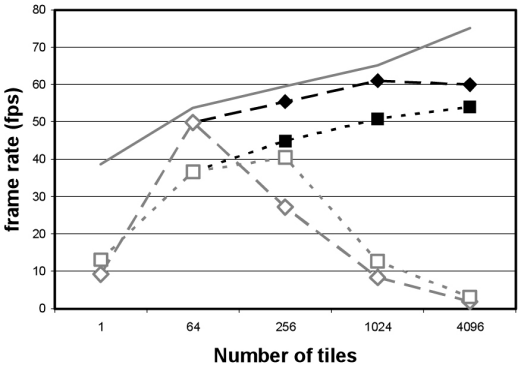

Figure 10: Performance for zoomed-out 1801 × 1201 data

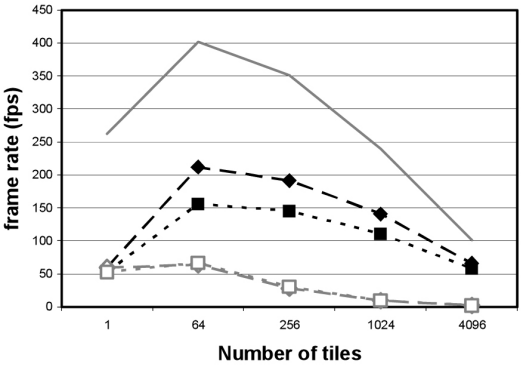

Figure 11: Performance for zoomed-in 721 × 481 dataset

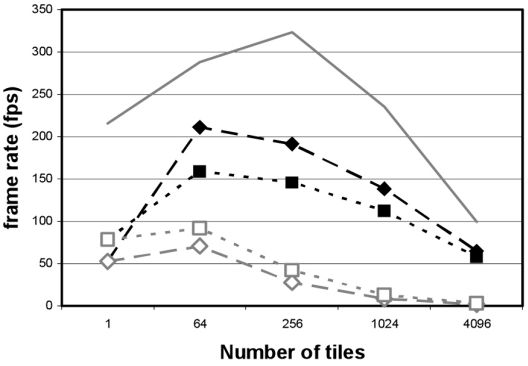

Figure 12: Performance for zoomed-out 721 × 481 dataset

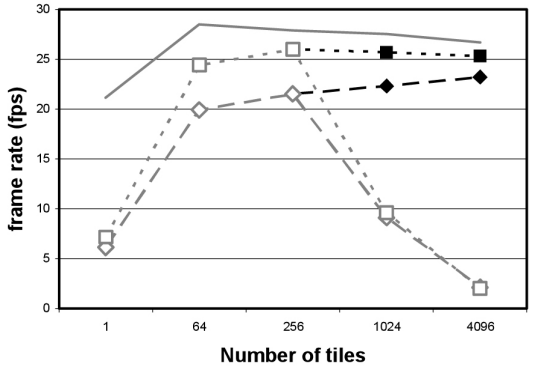

Figure 13: Performance for zoomed-in 1801 × 1201 dataset with increased fill requirements

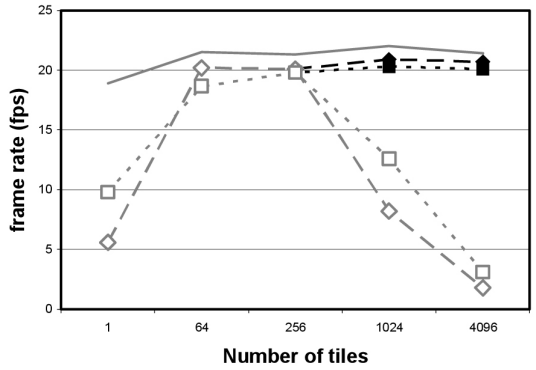

Figure 14: Performance for zoomed-out 1801 × 1201 dataset with increased fill requirements

4.2.3 Discussion of Results

Several behaviors are demonstrated by the performance plots. First, the value of tiling increases with scene complexity and with settings that would increase polygon fill time. For the reduced-detail dataset, there is no benefit to tiling beyond the minimal amount of 64 tiles (Figures 11 and 12) when the window is rendered. In fact, the overhead of increased tiling outweighs its potential benefits. Little difference can be seen for the two different views. With the higher-detail datasets (Figures 9 and 10), the benefits of increased tiling outweighed the cost, except for the finest-grained tiling. When polygon fill time was increased by anti-aliasing features of the graphics card (Figures 13 and 14), increased tiling usually increased performance or made little difference, and never decreased performance notably. Similar results should be anticipated when fill rate is substantially limited by texturing, lighting, very high resolution, or complex shading.

Consistent with this, differences in behavior between zoomed-in and zoomed-out views are explained by the increased complexity of the zoomed-out view (more of the dataset is visible). To generalize the discussion so far: usually, higher tiling resolution is preferable with increasing complexity when the increase results from increased dataset resolution, increased fill time, or wider field of view.

The improvement with tiling even in the baseline case of no window rendering may be considered counterintuitive – for example, that rendering only the main view with 256 display lists was always faster than rendering it as one display list. Contrary to a rule of thumb suggesting that reducing the number of triangle strips is desirable, plots illustrate inefficiency of long triangle strips. There are two aspects of tiling to note: first, tiling uses many smaller display lists rather than one large one; second, triangle strips are broken into shorter pieces when cut by tiling (furthermore, resulting substrips are rendered side-by-side rather than end-to-end). We investigated further by taking additional performance measures for the high-detail zoomed-out and zoomed-in cases. Specifically, we modified the baseline case (one large display list) to use shorter triangle strips to match those from tiling, but still as a single display list. Resulting performance resembled the previously plotted baseline case for coarse division, and no substantial performance drop occurred for finer division. Thus, the baseline's speedup with tiling is related to the particular organization of triangles into strips, and not due to size or number of display lists. However, the drop in baseline performance for the finest tiling (Figure 9) is likely due to extra display lists and not changes in triangle organization.

When the tiling system does not cache culling results (the *moving* cases), frame rates drop severely for fine-grained tiling due to the added overhead, while a benefit is seen for coarse tiling. As with 2D windows, one way to improve performance during window moves would be to only show a window as an outline during manipulation. Other approaches are possible, such as temporarily disabling *clip* during manipulation. Note that the plots are based on the worst case of no caching, but it is often possible to cache at least a subset of the tile lists. For example, the interior scene geometry is often fixed relative to the window during window manipulation. Furthermore, performance may be improved by exploiting properties such as coherence (cached tile lists may typically change only slightly from frame to frame) that are not considered in the current system.

Plots show that reasonable frame rates could be obtained in all tested scenes for some tiling resolution, but there is no single optimal choice of tiling resolution. A system should select tiling resolution based on dataset and rendering features and should store multiple representations if memory is available for this.

There is little existing work on performance for comparison. Ropinski and Hinrichs [3] evaluated performance of a depth-peeling technique for volumetric lenses and were able to render a lens at just above 40% of the frame rate obtained when no lens was in a scene. However, no relationship to scene characteristics was established and rendering style in their lens was simpler than for the no-lens case, so results are not conclusive.

With selection of a good tiling resolution, both the clipping plane and shader techniques evaluated here achieved frame rates above 50% of the highest rate achieved by the baseline case for the high-detail dataset. For zoomed-out high-detail and for anti-aliased cases, the added cost of introducing a window was minor when optimal tiling was used. In fact, Figures 10 and 14 show shader performance very close to the baseline case when tiling is used. For low-detail datasets, there is a more substantial slowdown from rendering a window. However, frame rates in such cases are already high enough that this slowdown does not produce undesirable results.

Finally, regarding the difference in performance between shader and clipping plane approaches: when there is a notable difference, the shader method tends to be moderately faster. However, the clipping planes method is seen to be faster in Figure 13 and in the one-tile points (leftmost points) in some plots. This result in Figure 13 may be due to an increase in fragments handled by the shader due to multisampling.

4.3 Concerning the Rendering of Multiple Windows

The rendering and use of multiple simultaneous 3D windows or lenses exhibiting *clip*-type behavior was not presented in the cited works. It has been suggested that, unlike for 2D approaches, the behavior of intersecting 3D elements may not be

as well-defined or meaningful [1]. Nonetheless, it is beneficial to render multiple windows to support more views of data, for example, as shown in Figure 5. With *clip* disabled, this is straightforward. For example, one extra pass per window is required with the clipping planes technique. However, when *clip* is enabled for the windows, a more complex approach is needed.

Consider the case of a clipping planes approach for multiple windows with *clip* on. The space external to the windows (the main view) must be partitioned into convex regions defined by at most n planar boundaries each, where n is the number of available hardware clipping planes (typically 6). For two nonintersecting windows with clipping, a separating plane can be used as the extra needed clipping plane. For the general case, regions can be meshed using algorithms from computational geometry (e.g., Delaunay tetrahedrization or minimal tetrahedrization; a minimal convex hexahedrization would be preferable). If intersecting windows are allowed, any internal region with more than n planar boundaries must also be subdivided. Besides the conceptual complexity of efficient implementation, the resulting number of rendering passes may make this impractical and limit the number of windows supported. Thus, such window behaviors are not fully supported with current systems.

The shader approach extends more readily to the case of multiple windows. A fragment can be checked against multiple window boxes by repeated application of a single-box technique, and kept or discarded based on a Boolean combination of results. Similarly, Boolean tests on results can be done in the tiling system that compares tiles against box boundaries. Full details will be addressed by future work.

5 Conclusion and Future Work

We presented interactive volumetric windows as a metaphor that encompasses a range of techniques. We developed an application to data viewer windows for interpretation of topographic and associated geophysical datasets that was motivated by observations of user behavior during an interpretation task. We note that the framework applies to other window types such as 3D toolboxes or windows that communicate information about a remote user's views and actions (for example, a window with its *source* set to follow a remote user's pointer).

We presented shader-based rendering for volumetric windows and evaluated the performance of both shader and clipping plane methods combined with dataset tiling. As tiling is representative of a class of commonly used speedup techniques, we expect results will generalize to some other applications.

Besides further technical development, evaluation of window interaction techniques is an important topic for future work. Several issues will arise, for example: Which parameter combinations are most useful to users, and for which display types (e.g., attaching to a user is probably not useful in the desktop)? What behaviors make the most sense to users in response to various interactions (e.g., when users scale the box, should contents scale with it, or should the view expand, and do expectations relate to window parameter values)? When a user reaches into a box, how should the user best specify intent to grab the box vs. intent to grab data in the box vs. intent to annotate? What are useful automations and how are they best implemented?

References

[1] J. Viega, M. J. Conway, G. Williams, and R. Pausch, "3D Magic Lenses," ACM UIST, 1996, pp. 51-58.

[2] R. Brady, J. Pixton, G. Baxter, P. Moran, C. S. Potter, B. Carragher, and A. Belmont, "Crumbs: A Virtual Environment Tracking Tool for Biological Imaging," *IEEE* Symposium on Frontiers in Biomedical Visualization, 1995, pp. 18-25.

[3] T. Ropinski and K. Hinrichs, "Real-time Rendering of 3D Magic Lenses having Arbitrary Convex Shapes," International Conference in Central Europe on Computer Graphics (WSCG), 2004, pp. 379-386.

[4] B. Fröhlich, S. Barrass, B. Zehner, J. Plate, and M. Göbel, "Exploring Geo-Scientific Data in Virtual Environments," IEEE Visualization, 1999, pp. 169-174.

[5] D. Weiskopf, K. Engel, and T. Ertl, "Interactive Clipping Techniques for Texture-Based Volume Visualization and Volume Shading," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9 (2003), pp. 298-312.

[6] R. Stoakley, M. Conway, and R. Pausch, "Virtual Reality on a WIM: Interactive Worlds in Miniature," ACM CHI, 1995, pp. 265-272.

[7] E. A. Bier, M. C. Stone, K. Fishkin, W. Buxton, and T. Baudel, "A Taxonomy of See-Through Tools," ACM CHI, 1994, pp. 358-364.

[8] M. Billinghurst, R. Grasset, and J. Looser, "Designing Augmented Reality Interfaces," *ACM SIGGRAPH Computer Graphics*, vol. 39 (2005), pp. 17-22.

[9] S. L. Stoev and D. Schmalstieg, "Application and Taxonomy of Through-The-Lens Techniques," ACM VRST, 2002, pp. 57-64.

[10] A. Fuhrmann and E. Groeller, "Real-time Techniques for 3D Flow Visualization," IEEE Visualization, 1998, pp. 305-312.

[11] C. W. Borst and G. L. Kinsland, "Visualization and Interpretation of 3-D Geological and Geophysical Data in Heterogeneous Virtual Reality Displays: Examples from the Chicxulub Impact Crater," Transactions: Gulf Coast Association of Geological Societies, 2005, pp. 23-34.

[12] C. W. Borst, G. L. Kinsland, V. B. Baiyya, A. M. Guichard, A. P. Indugula, A. V. Asutay, and C. M. Best, "System for Interpretation of 3D Data in Virtual Reality Displays and Refined Interpretations of Geophysical and Topographic Data from the Chicxulub Impact Crater," Transactions: Gulf Coast Association of Geological Societies, 2006, pp. 87-100.

[13] A. R. Hildebrand, G. T. Penfield, D. A. Kring, M. Pilkington, A. Camargo-Zanoguera, S. Jacobson, and W. V. Boynton, "Chicxulub Crater: A Possible Cretaceous/Tertiary Boundary Impact Crater on the Yucatán Peninsula, México," *Geology*, vol. 19 (1991), pp. 867-871.

[14] K. O. Pope, A. C. Ocampo, and C. E. Duller, "Mexican Site for K/T Impact Crater? ," *Nature*, vol. 351 (1991), p. 105.

[15] C. Ware, K. Arthur, and K. S. Booth, "Fish tank virtual reality," SIGCHI Conference on Human Factors in Computing Systems, 1993, pp. 37-42.