Hybrid Visualization Approach for Vortex Method Simulations

Jim Geiger[1], Peter Bernard[1,2], Pat Collins[1]

[1] VorCat, Incorporated, Rockville, Maryland, USA
[2] University of Maryland, College Park, Maryland, USA

KEYWORDS*:*
Main subject(s): *Computer Visualization*
Fluid: *Aerodynamics*
Visualization method(s): Hierarchical Z-Buffer Visibility, Level of Detail
Other keywords: Vortex Methods, Octree

ABSTRACT: This paper presents a novel technique for rendering simulated incompressible flows produced by VorCat, a patented commercial implementation of the grid-free vortex method. VorCat is specifically designed to treat high Reynolds number, turbulent, incompressible flows of engineering interest. Computational elements consist of straight vortex tubes combined to form filaments, and finite thickness, thin, unstructured, triangular vortex sheets, several layers deep, covering solid surfaces. The sheet mesh provides an efficient vehicle for accommodating steep vorticity gradients in the near-wall region and permits accurate evaluation of viscous diffusion and surface vorticity production. To the extent that the vortex tube elements mimic the actual tube-like vortical structures of turbulence, the efficiency and physical plausibility of the VorCat algorithm is enhanced. Visualization of turbulent flow simulations made with VorCat often requires rendering millions of distinct vortex tubes appearing in each frame. A brute force approach, even with state-of-the-art computer graphics hardware and virtually unlimited memory, still requires several minutes to render an image. However, the application of various scene simplification and visibility techniques, originating in terrain visualization and CAD applications, produces meaningful images at near-interactive frame rates. In our approach, tube-like representations of the vortices are triangulated and placed into a spatial octree database. After view frustum culling, the hierarchical Z-buffer (HZB) algorithm is used to eliminate triangles that are occluded. Finally, an appropriate resolution of each tube is selected based on the viewpoint. This approach is gaining popularity in the field of visualizing massive polygonal databases and is particularly well suited for our representation of vortex elements.

0.0 Introduction

One of the most important research problems faced in the analysis and visualization of massive datasets, particularly those encountered in computational fluid dynamics problems, is the development of methods for efficiently storing, approximating, and rendering large amounts of data. A number of such methods have been developed for the purpose of addressing particular characteristics of datasets in fields ranging from computer aided design (CAD) systems in which a building model can have millions or even billions of polygons, to terrain visualization systems for which sub-meter resolution data for entire countries is now readily available. This paper describes an approach for rendering the vorticity structures produced by our patented turbulence modeling software (VorCat) based on combining several ideas from recent developments in the visualization of large polygonal databases. The first section below describes the VorCat methodology in terms of the “output” data that we want to visualize and understand. While we are not the first to consider vorticity and turbulence utilizing the graphics primitives discussed herein, there is as yet no commercial graphics application capable of handling the huge amount of data produced by VorCat. The second section outlines particular requirements for the visualization system to be used as a

practical analysis tool for scientists and engineers with low-cost, windows based computers on their desk. The third section describes the data structures, the visibility and culling algorithms, and the level-of-detail features of the rendering system. Finally we present specific implementation issues, show some results, and describe future work.

1.0 Background

1.1 VorCat

The VorCat software[1] is an approach to Computational Fluid Dynamics (CFD) based on the gridfree vortex method. Such schemes use freely convecting and interacting vortex elements as the primary basis for representing the flow field, in place of the more common use of discrete velocity and pressure fields defined on a fixed numerical mesh. The development of vortex methods and VorCat in particular has been spurred by a number of conceptual and practical advantages that such schemes have in the modeling and simulation of turbulent flow. Besides their gridfree status, which is a significant advantage in modeling complex engineering geometries, they offer a natural and efficient representation of turbulent flow processes that are essentially vortical in nature. For example, they can resolve sharp flow features that would otherwise be smoothed on numerical meshes.

The primary gridfree computational elements in VorCat consist of straight vortex tubes [Figure 1] combined to form filaments.

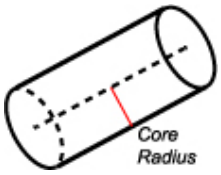

Figure 1: *Vortex Tube*

These are accompanied by a thin, unstructured mesh of triangular vortex sheets, several layers deep, covering solid surfaces. The latter provides an efficient and accurate means of accommodating steep vorticity gradients in the near-wall region out of which new vortex tube elements are produced in the flow. The use of tube-like elements as against other alternatives used in vortex methods (e.g., vortex blobs [1]), is motivated by their natural representation of tube-like vortices that are fundamental to the physics of turbulence. This includes the small-scale vortex tubes whose stretching carries energy to dissipation scales, but also the quasi-streamwise vortices that control the dynamics of the turbulent boundary layer.

Among gridfree elements, vortex tubes provide unmatched numerical stability under stretching and provide an opportunity to use hairpin removal and vortex reconnection algorithms [2] as a de facto subgrid scale model with which to limit resolution without harm to the physics.

Turbulent flow simulations using VorCat are in the category of Large Eddy Simulations (LES) in the sense that the intent is to mimic the dynamically significant features of the flow in the energy containing range of scales, while using the subgrid model to accommodate the influence of the smaller dissipation scale motions. Even with limited scale resolution and the intrinsic efficiencies of directly representing vortical motions via vortex elements, it is often the case that a very large number of vortex elements are needed to adequately represent turbulent flows. Typically several million vortices will appear in the solution of engineering problems, and to fully reap the advantages of the technique (e.g., physical insights into flow structure and evolution), it is necessary to have tools for the visualization of these large, dense fields of vortices.

[1] VorCat turbulence modeling software is protected under U.S. Patent 6512999.

1.2 Previous Work

Initial investigations into vortex methods were limited to two-dimensional flows in view of the substantial computational requirements of such techniques. Most significantly, vortex methods require the solution of an N-body problem due to the mutual influence of the vortices in establishing the dynamical behavior of the whole. With increasing computational power, algorithms for three-dimensional flows were subsequently developed. These most commonly used closed vortex loops, composed of a number of individual tubes, as the gridfree elements. Because of the extreme expense of computing with such objects, simulations tended to be of a modest scale, usually including no more than a few thousand tube-like vortices.

Visualizations of many three-dimensional calculations typically show the vortex tubes as lines in a 3D plot. For example, the development of a vortex ring [3], a turbulent filament undergoing rapid stretching [4] and a model of a jet flow [5]. Such pictures are extremely suggestive of the underlying physics and valuable in an analysis of the phenomena.

Recent years have seen the incorporation into vortex methods of fast solvers, such as the Fast Multipole Method (FMM) [6] for the rapid solution of the N-body problem. The result is calculations in three dimensions with many more vortical elements than previously. In particular, for VorCat, which has been a leader in the development of large-scale turbulent flow simulations, many millions of vortices are economical and necessary to include in the simulations. Simple plots of lines representing vortices are of very limited use when the number of elements is large and when the flow is turbulent so that the flow structures of interest are complex. Thus, the need to develop visualization tools uniquely catering to the peculiarities of large systems of tube-like elements has been a priority of VorCat. We now describe the visualization scheme we have developed in tandem with the VorCat code.

2.0 Preliminaries

2.1 Terms

The VorCat modeling methodology discusses tubes as the graphical primitive characterizing vorticity; however the remainder of the discussion will focus on *vortex filaments* and their associated triangulated representation. These are simply chain-like collections of vortex tubes. The current modeling scheme limits filaments to twenty tubes.

2.2 Database

The stability and evolution of the VorCat modeling software will continue to produce larger and larger datasets. Currently, we are challenged to handle approximately 1.2 million filaments per scene. At an average of four tubes per filament, and sixteen triangles per tube[2], the rendering system may theoretically need to handle up to 77 million triangles per. It is rare however that a user wants to see all of the filaments at once. Typically, between fifty and eighty percent are filtered out by some set of associated parameters. Also, reduced resolution versions of the filament structures need to be maintained. The set of resulting primitives is referred to as the *database*.

The database also includes the triangles representing the body geometry over which the flow is being studied. Typically however, the size of this geometry is less than 10,000 triangles and is therefore omitted from further discussion.

2.3 Frame Rate

Interactive visualization of such massive polygonal databases is rarely possible at truly interactive frame rates such as 10-20 frames per second (fps), not to mention game-like frame rates of 30-60fps. Moreover, our implementation of quad-buffered stereoscopic viewing (double-buffered left eye and double-buffered right eye) effectively cuts any frame rate in half. At present we are

[2] More triangles may be required depending upon on how the tubes are connected.

achieving less than one frame per second despite the huge advances we've made with the approach described in this paper. As graphics hardware becomes faster and faster however, with particular focus on large vertex databases and hardware support for some operations described in this paper, we fully expect to achieve at least 5 FPS. In fact, since the beginning of the project, two product lines of PC graphics cards have been introduced with partial hardware support for one of our algorithms[3].

2.4 Platforms

A key requirement for the visualization system is that it be available on desktop workstations and laptops. The development and target platforms are Windows XP/2000 based machines with nVidia and ATI graphics cards and at least 1.2 GB of main memory. The current graphics API is OpenGL although DirectX is a likely candidate for future development. For stereoscopic support, a quad-buffered stereo card such as the ATI is required.

2.5 Visualization System Features

While many features have been considered, several are crucial to our visualization application. First, users need the ability to interactively manipulate the color scheme for the filaments. Second, users need the ability to filter out filaments based on strength, axis orientation, spin, and length. Third, users need the ability to select individual vortex filaments and query for statistical information. These requirements make it necessary to hold much of the data in memory, including the original information from which the database is built.

3.0 Visualization System Architecture

The visualization software architecture is described in four steps. First we describe the creation of the polygonal database from the modeling data and the primary data structure used throughout the discussion. Then the rendering algorithm is introduced with particular attention to visibility culling and the Hierarchical Z-buffer. Finally, we describe the level-of-detail portion of the rendering algorithm, which employs approximations of filaments in cases where full resolution is not necessary or even practical.

3.1 Polygonal Database Construction

The first pre-computation step in our architecture is to build the database from the vorticity data. This task is straightforward but time consuming. Using a list of vortex filaments tubes with connectivity information, vertices, spin, and strength, we construct a triangulation of tubes representing this data. We also compute normals and store orientation information for lighting and filtering. The triangulated filaments are shown in Figure 2.

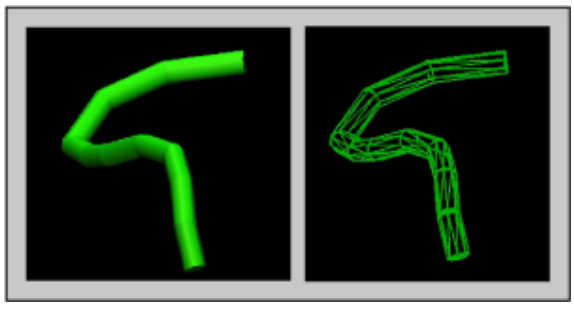

Figure 2: Vortex filament rendered as surface (left) and its triangulation (right).

[3] The ATI Radeon Pro series includes Hyper Z -- a technology that makes Z-buffer bandwidth consumption more efficient with harware implementation of the Hierarchical Z-buffer. nVidia's Lightspeed Memory Architecture, used in geForce3 and above, includes hardware level support for z-occlusion queries as well as z-occlusion culling.

3.2 Octree Construction

The *octree* spatial subdivision, or just octree, is a very popular data structure for large, complex polygonal databases. In general, an octree is constructed by enclosing the entire scene in a minimal axis-aligned box. The rest of the procedure is recursive in nature, and starts by checking whether the box contains less than a threshold number of primitives. If it does, the algorithm binds the primitives to the box and then terminates the recursion. Otherwise, it subdivides the box along its main axes using three planes, thereby forming eight boxes. Each new box is tested and possibly subdivided again into 2x2x2 smaller boxes. This process continues until each box contains fewer than the threshold number of primitives, or until the recursion has reached a specified deepest level. Such boxes are normally referred to as nodes. The two types of nodes in an octree are internal nodes, which contain the position of the planes dividing the subspaces represented by the nodes, and leaf nodes, which contain the primitives themselves.

Our implementation of the octree structure assumes that the nodes are cubes. The threshold number of primitives per node (node size) remains adjustable and should vary by dataset and, in particular, the density of structures. Smaller node sizes result in more subdivision and therefore deeper trees. The practical impact of this trade-off is on rendering speed and memory consumption.

The actual construction of the tree is then a simple recursive procedure. A triangle is considered inside of a node if its center is inside of the node. The root node of the tree contains all of the triangles in the database. If the number of triangles is less than the node size, we exit. Otherwise we recursively subdivide the node into eight child nodes and distribute the triangles, based on the center, to the eight child nodes after each subdivision. If a triangle overlaps nodes, it is placed in both. This is done until each node has contains fewer polygons than the node size. One obvious weakness of this structure is that some large cubes will have few primitives while other very small cubes will be full. The problem becomes apparent during the rendering step of our approach but is beyond the scope of this paper.

3.3 Culling - Hierarchical Visibility Algorithm

The critical step for the rendering performance is the elimination of things that are not visible. Vortex filaments may be outside of the viewing area, behind other filaments, or too far away to be seen. Traditional view frustum culling, which eliminates the contents of octree nodes outside of the viewing frustum, produces little to no overall improvement for a given frame as most of the filaments lie in dense vorticity structures in the critical areas of the flow (i.e. the region of primary interest). Traditional Z-buffer culling does not account for many of the filaments hidden inside or behind these structures. Ray-casting methods used to determine visibility of octree nodes also do not result in adequate culling as they don't exploit image-space coherence. The Hierarchical Z-buffer Visibility (HZV) algorithm was developed to address these and other problems [8]. This approach exploits both object-space coherence, through the use of an octree such as the one described above, and image-space coherence, with a Z-buffer algorithm described in this section. The HZV algorithm has another major component that exploits temporal, or frame-to-frame, coherency. While it has not yet been implemented in our system, a discussion of this component is found in Section 6. Following is a description of part of the algorithm and our implementation.

3.3.1 Algorithm Description

Once the octree has been created as described in Section 3.1, each frame is rendered in approximately front-to-back order by calling the procedures outlined in Figure 3. Nodes that are outside the view frustum are immediately discarded. The first step determines whether the node's bounding box, i.e. its corresponding cube in the octree, is visible with respect to the Z-pyramid. The Z-Pyramid and visibility tests are described in the following sections. If the node is occluded, we do not need to process the contents of that node any further, since its contents do not contribute to the final image. Otherwise, we tile the primitives associated with the node into the Z-pyramid and then process each of the node's children, if it has any, in front-to-back order using this same recursive

procedure. When recursion finishes, all visible primitives have been tiled into the Z-pyramid, and a standard Z-buffer image of the scene has been created.

```
RenderScene( OctreeNode Root )
{
        clear image buffer to background

        clear z-buffer to far clipping plans

        ProcessOctreeNode(Root)
}


ProcessOctreeNode( OctreeNode N )
{
        if IsCubeVisible(N) returns FALSE
                then return

        for each primitive P associated with N
                tile P into the z-buffer

        for each child C of N in front-to-back order
                ProcessOctreeNode(C)

}
```

Figure 3: Pseudo-code for Hierarchical Z-Buffer rendering.

The HV algorithm performs occlusion culling very efficiently because it only traverses visible octree nodes and their children, and it only renders the primitives in visible nodes. This can save much of the work in scenes that are densely occluded.

3.3.2 Z-Pyramid

Now we will describe how the Z-pyramid is maintained and then how it is used to accelerate culling. The finest, highest resolution, level of the Z-pyramid is simply a standard Z-buffer. At all other levels, each z-value is the farthest z in the corresponding 2x2 window of the adjacent finer level. Therefore each z-value represents the farthest z for a square region of the screen. To maintain a Z-pyramid, whenever a z-value is overwritten in the Z-buffer it is propagated through the coarser levels of the Z-pyramid. This is done recursively until the top of the image pyramid is reached, where only one z-value remains. Figure 4 demonstrates this on a 4x4 region of the Z-buffer.

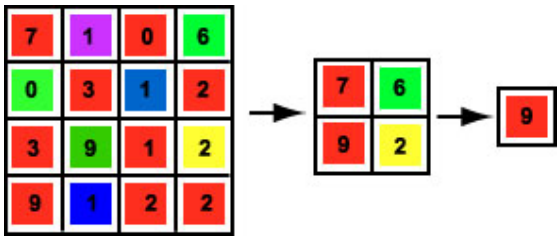

Figure 4: The Z-buffer Pyramid.

3.3.3 Hierarchical Culling

Next, we describe how hierarchical culling of octree nodes is done. To determine whether a node is visible, the front faces of its bounding box are scan-converted[4] and tested against the Z-pyramid. The node is occluded if all of its front faces are occluded by the Z-pyramid. To establish whether an individual face is occluded, we begin at the coarsest Z-pyramid cell that encloses the face's screen projection. The face's nearest depth within the node is then compared to the Z-pyramid value, and if it is farther, the face is occluded. For densely occluded scenes, this procedure often culls an

[4] Presently the scan conversion is done in software. This bottleneck is addressed in Section 4.1.

occluded face with a single depth comparison. When this initial test fails to cull a face, its visibility can be definitively established by recursively traversing from the initial Z-pyramid cell down to finer levels in the Z-pyramid. Additional depth comparisons at the encountered child cells must be performed. If this subdivision procedure does not ultimately find a visible sample on the face, the face is occluded. In scenes where the bounding boxes of octree nodes overlap deeply on the screen, the hierarchical procedure can establish visibility much more efficiently than can conventional Z-buffering. Pseudo-code for the visibility is presented in Figure 5.

```
IsCubeVisible( OctreeNode N )
{
        if N is completely outside the viewing frustum
                then return FALSE

        if viewpoint is inside N
                then return TRUE

        if  N intersects the near face of the viewing
frustum
                then return TRUE

        for each front face F of N
        {
                if  F is visible at one or more pixels
                        then return TRUE
        }

        return FALSE

}
```

Figure 5: Pseudo-code for the visibility test.

3.4 Multi-Resolution Construction

Another common approach to the performance improvement of rendering scenes with a large number of polygons, such as our datasets comprised of millions of tubes, is to use simpler versions of an object as it makes less and less of a contribution to the rendered image. The hierarchy of simpler objects is referred to as levels of detail (LOD). In general, an LOD algorithm consists of three major parts, namely LOD generation, selection, and switching. The generation depends primarily on the physical phenomena being conveyed. The selection is determined by a variety of factors including distance to the object, guaranteed frame rates, and relevance of a particular polygon or polygonal structure. Finally, the process of switching between levels of detail at rendering time can result in visual artifacts that are potentially distracting or confusing. Here we describe our approach to each of these aspects.

3.4.1 Level of Detail Generation

The first step in designing the LOD rendering component is to define approximations of the filament structures. In the context of our octree data structure, this means we choose different versions of an octree node to render depending on its distance from the viewer. While there are many physical characteristics that could be considered in developing such LODs, we have focused most of our attention on the actual pixels being sent down the graphics pipeline rather than physical relationships between the filaments contained in a given octree node. These relationships are thoroughly considered and better optimized in the modeling software itself. A discussion of these optimizations is available in [9]. In particular, the processing and memory resources required to accurately characterize, or generalize, localized turbulence phenomena are beyond platform capabilities of our visualization software.

We define three distinct representations of an octree node filled with filaments. The first, full resolution representation consists of filaments constructed from eight sided, tube-like glyphs. The

second representation consists of variable width lines. The width is an adjustable parameter for runtime performance tuning. The final representation consists of a single weighted average of the vortex tubes comprising the filaments in the node. In particular, we try to capture the average strength and shape the filaments in the box by constructing a new filament whose endpoints, length (number of tube segments), and color reflect the median of the node's contents. Originally, we had implemented a more elaborate scheme of removing filaments in a node by considering that vortex tubes have cancellation effects upon one another based on their "spin." However this cancellation is now performed in the modeling step to further improve other performance bottlenecks. The current implementation does not use the last (simplest) LOD. It is of negligible visual contribution and we are presently considering alternatives.

3.4.2 Selection – Projected Area Based

Given that different LODs of a node exist, it is necessary to choose which one to render for a given viewpoint. Typically this selection is based on either the distance from the viewpoint to the node, or on the projected area of the bounding volume of the node. Our implementation uses the second approach as the scenes are usually compact and the distance to the viewpoint is largely invariant. Practically, this amounts to computing the area of a projected cube based on the viewpoint of the user. We observe that a single lighted filament consisting of four tubes is meaningful as a non-degenerate, projected polygon only in cases where it occupies at least 30x30 screen pixels (enough to perceive shading). The next resolution levels are assigned to node projection areas above 20x20, 10x10, etc. Higher resolution displays may have more levels. Finally, a node projected onto the screen with area less than 2x2 pixels is rendered at the lowest resolution. An algorithm to quickly compute the projected area of a node is described in Section 4.2.

3.4.3 Switching – Discrete Geometries

In many fields where LOD techniques are employed, the crucial aspect of the implementation is *how* to switch between LODs. That is, when the viewpoint changes and the selection criteria ask for a different representation of a given node, how do we render the new one? The problem is usually the noticeable change of switching from one LOD to another and how it changes our perception of the scene. Many techniques have been developed to address these problems but are beyond our immediate necessity of achieving practical frame rates. Thus we use the simplest of all approaches: the so-called discrete geometry LOD algorithm. When a change of LOD is requested for a given node, we simply render the new one. This is particularly well suited for our application as it allows us to reuse the vertices that we store for the full resolution representation of the filament. Alternatives often require storing additional vertices, normals, etc. While this simplification introduces an error equal to the core radius of the filament, the rendered image is unaffected as the error is less than a single pixel value.

4.0 Implementation Issues

4.1 Hierarchical Z-Buffer

Among the many implementation details associated with the Hierarchical Z-buffer culling algorithm, the primary focus was to reduce the amount of time spent querying the depth buffer. When we started the project, hardware support for the Z-Pyramid scheme was limited to an HP extension to OpenGL and an ATI hardware implementation called "Hyper Z." Because we use nVidia graphics hardware, our implementation was completely in software, with the exception of the bottom of the pyramid -- the hardware z-buffer. Subsequently, nVidia has introduced an OpenGL extension to their hardware support for Z-buffer occlusion testing on the geForce3+ cards. This feature provides a very simple occlusion query for rasterized primitives and effectively eliminates the need for the pyramid. The OpenGL code in the following fragment demonstrates how the extension is used.

```
// turn off drawing and z-buffer upadates
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);

for (i = 0; i < octNodeCount; i++)
{
        glBeginOcclusionQueryNV(occlusionQueries[i]);
        // render bounding box  -- note that here we used to scan-convert
        // and manually update the Z pyramid
        glEndOcclusionQueryNV();
}

// at this point, if possible, go and do some other computation
// turn back on the drawing and z-buffer
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);

// get the results
for (i = 0; i < octNodeCount; i++)
{
        glGetOcclusionQueryuivNV(occlusionQueries[i],
                GL_PIXEL_COUNT_NV, &pixelCount);

        if (pixelCount > 0)
        {
                // this node is visible so render its
                // contents or continue traversing
        }
}
```

Figure 7: OpenGL code fragment for visibility test.

4.2 Level of Detail

The second part of the LOD algorithm is the selection of which LOD to render. In our implementation, this selection is based on the projected screen space of the octree node. Computing the exact projection however for thousands of nodes per frame is very costly. A good approximation is sufficient however as an error of several pixels does not have a noticeable effect in the rendered scene. One such approximation scheme, developed by Schmalsteig et al [7], quickly estimates the projection based on look-up tables. Figure 8 depicts three views of a bounding box in a perspective projection and is representative of the possible number of visible faces of a projected cube.

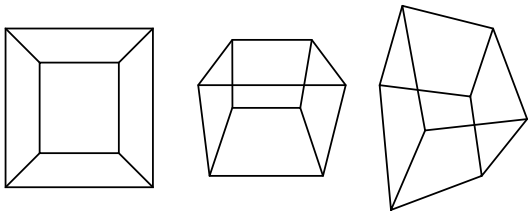

Figure 8: Octree node projections: one, two, or three faces may be visible.

These views are combined with the three possible types of viewpoints to generate a look-up table (LUT) containing the characteristic hull polygons. Then, based on a given viewpoint, an appropriate hull is selected from the LUT. Before the area of the hull is computed, its vertices are projected into screen space. Finally, a contour integral is applied to the projected vertices to gain a very fast approximation of the projected screen space of the cube.

Note that this step could be eliminated completely by using the scan converted node faces obtained in the part hierarchical Z-buffer algorithm implementation. However plans are already underway to eliminate the section including the software scan-conversion of the node face. The previous section describes this in more detail.

5.0 Results

We include here some images of vortex tubes computed in a simulation of turbulent flow associated with a tripped, zero-pressure-gradient boundary layer developing over a flat plate. The simulation begins from an impulsive start, and an equilibrium boundary layer appears containing a laminar flow immediately downstream of the trip, followed by transition to a fully turbulent state. Figure 9 is a view of the transition region where vortex tubes are observed to undergo significant reorientation from the spanwise to the streamwise directions. This is evidently the culminating downstream effect of instabilities (reminiscent of Tollmien-Schlichting waves) that appear behind the trip. The dense covering of tubes represents both the natural viscous thickening of the boundary layer that causes vorticity to rise above the underlying wall mesh, but also the effect of turbulent ejection events that cause wall vorticity to move outwards. Figure 10 is a detail of a turbulent structure appearing further downstream in the boundary layer. It shows much similarity to the eddy structure seen via smoke-visualization in physical experiments.

Each scene is comprised of nearly 900,000 vortex filaments and was rendered in about 1.9 seconds. More than 90% of the triangles were culled with the approach described in this paper.

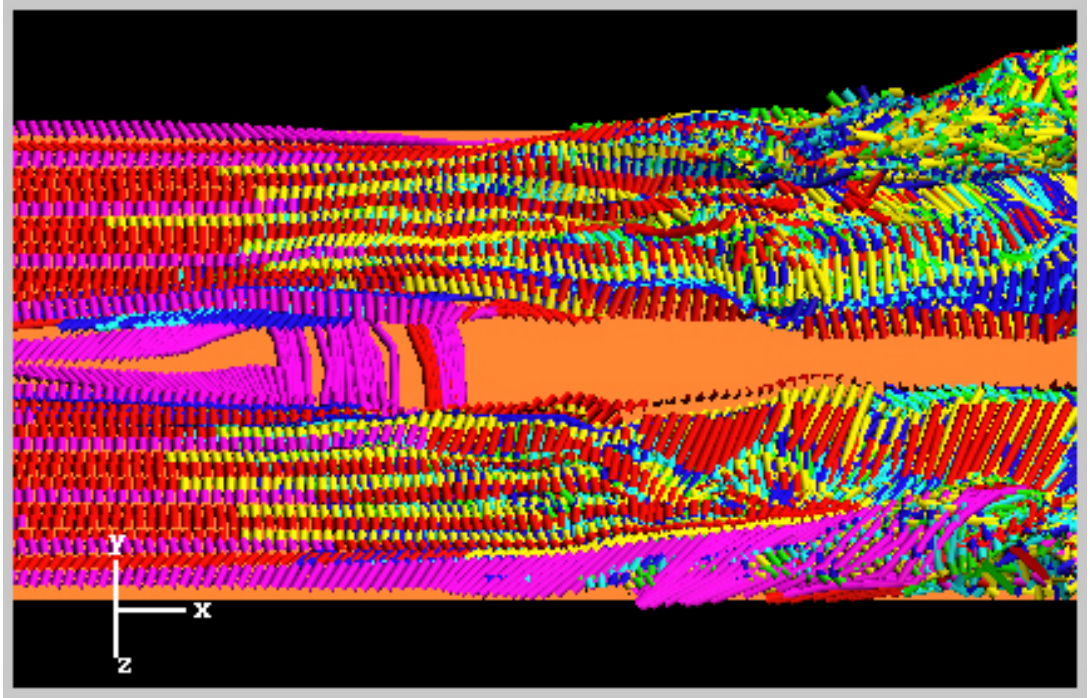

Figure 9

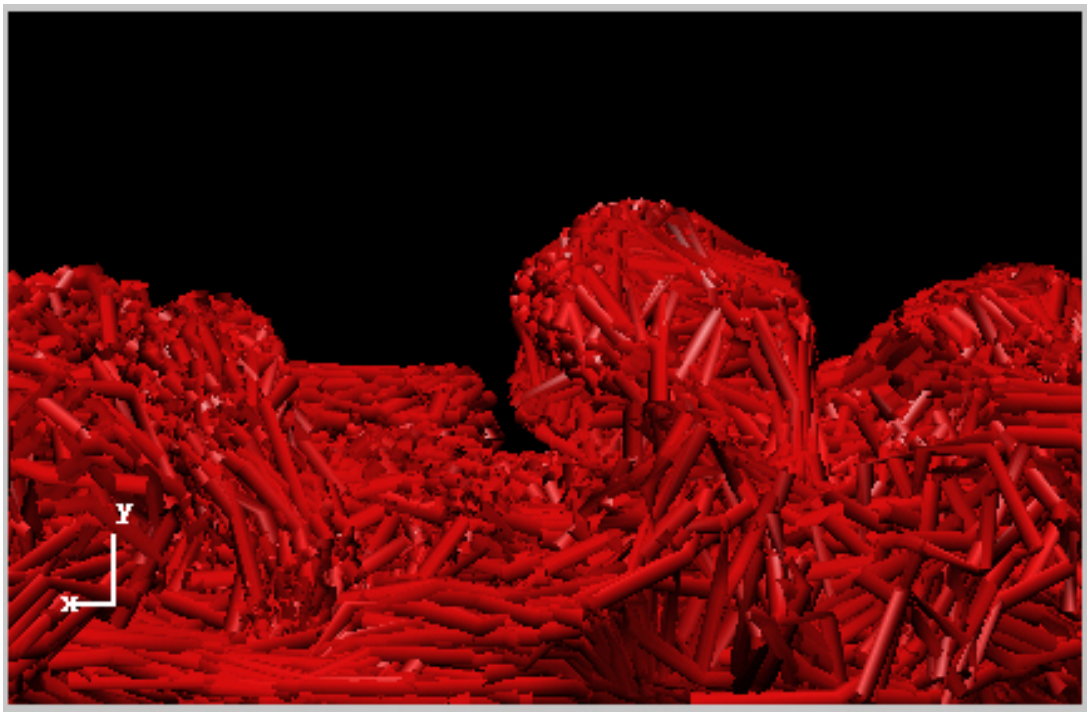

Figure 10

6.0 Future Work

There are two immediate areas for improvement. The first is to fully implement the hardware supported z-buffer occlusion test as outlined in Section 4.1. A significant performance increase should be realized with the elimination of all of the depth buffer reads and software scan conversions required by the Z pyramid maintenance. While this will introduce a hardware dependency (currently it's only available on nVidia cards), we fell that it is not a serious restriction. The two major PC card manufacturers, nVidia and ATI, are both enhancing their z-buffer support and will likely converge on their offerings.

The second improvement is related to the original HZB algorithm and is still applicable to our hybrid approach. While the current implementation exploits both image-space and object-space coherency, it does not consider frame-to-frame coherency. The HZB algorithm addresses this as well. Conceptually, octree nodes that were visible in one frame tend to be visible in the next. Consider a *sequence* of frames to be a set of images generated by moving the viewpoint (e.g., translating or zooming). Typically, such a sequence differs very little from frame to frame. The first frame of a sequence is generated with the previously described HZB implementation. After the first frame, a list of octree nodes that were visible in that frame is created by testing nodes for visibility against the Z-pyramid. Subsequent frames are generated with the following two-pass algorithm. In the first rendering pass, primitives associated with nodes on the visible node list are rendered by Z-buffer hardware. Then, the Z-buffer of the partially rendered scene is read back from the hardware, and a Z-pyramid is built from this Z-buffer. In the second rendering pass, the standard HZB algorithm is run in software, traversing the octree from front to back but skipping nodes that have already been rendered. This second pass fills in any missing parts of the scene. The final step is to update the list of visible octree nodes.

A third area for improvement is to exploit more physical properties of the flow when generating and selecting LODs. The current implementation does not allow sufficient memory or processor time to do anything except the generation already described in Section 3.4. Introduction of the hardware Z-buffer occlusion tests, as outlined in Section 4.1, presents opportunity to compute other properties

while waiting for the depth test results. During this is step in the rendering is precisely the right time to do the work necessary for LOD selection improvement.

7.0 Acknowledgements

The research reported here was supported by the Department of Energy SBIR program and the Advanced Technology Program at the National Institute of Standards and Technology.

8.0 References

[1] Puckett, E. G. (1993) Vortex methods: an introduction and survey of selected research topics. In "Incompressible computational fluid dynamics: trends and advances", edited by M. D. Gunzburger and R. A. Nicolaides, Cambridge University Press, Cambridge, 335 - 407.

[2] Chorin, A. J. (1993) "Hairpin removal in vortex interactions II,"J. Comput. Phys., Vol.107, 1 - 9.

[3] Leonard, A. (1975) "Numerical simulation of interacting, three-dimensional vortex filaments," Lec. Notes in Phys., Vol. 35, 245 - 250.

[4] Chorin, A. J. (1982) "Evolution of a turbulent vortex," Communications in Mathematical Physics, 83, pp517-535

[5] Meiburg, E. and Martin, J. (1991) "Three-dimensional vorticity dynamics in jets," Lec. in Applied Math. Vol. 28, pp. 467 - 479.

[6] Greengard, L. and Rokhlin, V. (1987) "A fast algorithm for particle simulations," J. Comput. Phys. Vol. 73, 325-348.

[7] Schmalstieg, D. and Tobler, R (1999) “Fast Projected Area Computation for Three-Dimensional Bounding Boxes,” Journal of Graphics Tools, 4(2):37-43.

[8] Greene, N., Kass, M. and G. Miller (1993) “Hierarchical Z-Buffer Visibility,” Proceedings of SIGGRAPH93.

[9] Bernard, P.S. and Wallace,J.M., *Turbulent Flow: Measurement, Analysis and Prediction*. John Wiley and Sons, Inc., 2002.

[10] Baxter, W., Sud, A., Govindaraju, N., Manocha, D. (2002) “GigaWalk: Interactive Walkthrough of Complex Environments,” University of North Carolina Technical Report TR02-013.

[11] Samet, H., *Applications of Spatial Data Structures*, Addison-Wesley, 1989.