Visualization of Time-Varying Curvilinear Grids Using a 3D Warp Texture

Yuan Chen, Jonathan Cohen, Subodh Kumar

Johns Hopkins University, Department of Computer Science
NEB 224, 3400 North Charles Street, Baltimore, MD 21209, USA
Email: {cheny,cohen,subodh}@cs.jhu.edu

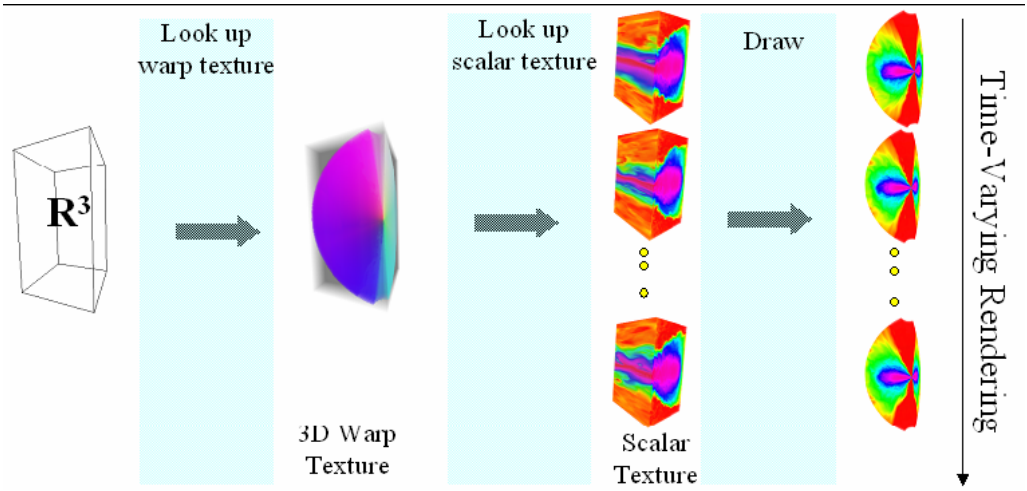

Figure 1: Fragment algorithm for rendering a time-varying curvilinear grid using a 3D warp texture.

Abstract

We present a novel scheme to interactively visualize time-varying scalar fields defined on a curvilinear grid. We create a 3D warp texture that maps points in R^3 into the grid coordinate system. At rendering time, the warping function is reconstructed at each fragment using tri-linear interpolation, and provides the 3D texture coordinates required to look up a scalar field value stored in a separate scalar texture. In essence, this approach reduces the problem of rendering a curvilinear grid to the problem of rendering a regular grid with one additional texture lookup.

Because the curvilinear grid data typically lies on a regular grid in its native space, the scalar data is easily stored in a 3D texture without the need for explicit resampling. For many time-varying data sets, the warping function itself is constant over time, so only the 3D scalar texture needs to be reloaded with each time step. Thus this factorization of the problem minimizes the bandwidth requirements for time-varying playback. We demonstrate the approach on several data sets, achieving interactive performance with low approximation error.

1 Introduction

A curvilinear (structured) grid is topologically a uniform, Cartesian grid, but is geometrically warped into R^3, a Euclidean space. For example, curvilinear grids specify a warped position for each data point and assume some interpolation function for locations between the data points. Curvilinear grids are convenient for applications such as computational fluid dynamics because they provide a regular grid space for the simulation while providing an adaptive sampling of the domain space. In Eulerian simulations, the mapping of the grid into the domain space is fixed over time, whereas in Lagrangian simulations, that mapping may change over time. Both styles of simulation are widely used

in practice.

As with regular and unstructured grids (tetrahedral meshes), curvilinear grids may be rendered using customized ray tracing [4] and cell projection techniques [11, 8]. In fact, it is even quite common to tetrahedralize the grid cells and render as an unstructured grid. This is exemplified by the fact that many papers describing unstructured grid rendering actually use test data that originated as structured grids (such as NASA's blunt fin and delta wing models). This approach has the benefit of leveraging advances in the rendering of this more general mesh structure [10] and may produce high quality images using cell projection. However, even the very fastest tetrahedral mesh rendering algorithm we are aware of currently achieves a maximum rendering throughput of 1.8 million tetrahedra per second [1]. For the largest data set we report here, containing roughly 2.3 million cells (or 11.5 million tetrahedra) we would thus expect a maximum performance of 6 seconds per frame, which is not interactive. Such an approach suffers from an inability to leverage information about the curvilinear grid's simple topological structure.

A typical, higher-performance approach to rendering curvilinear grids involves resampling the scalar data into a regular grid, either at a fixed resolution or hierarchically [6]. Such resampling admits fast, hardware-accelerated rendering via 3D textures, using either slicing and blending techniques [12] or hardware raycasting [5]. However, resampling the scalar data at each time step not only dramatically increases the total data size, but also potentially introduces unnecessary resampling artifacts. The choice of resampling resolution determines both the quality of each time step and the data transfer size for loading each time step to the graphics hardware.

In contrast to this direct resampling, we propose a novel approach that still combines regular sampling with 3D texture-based volume rendering, but with an important difference – we do not resample the scalar field except at the pixel level. By using one level of texture indirection, we effectively leverage pre-computed point location queries from Euclidean space into the grid space. For Eulerian simulation data, this *warp texture* does not change over time, and thus is easily cached in texture memory on the graphics hardware. At rendering time, we can now load the scalar data for each time step in its original grid-space sampling pattern as a 3D texture, avoiding the additional bandwidth required to maintain high quality rendering for direct resampling approaches.

Our approach has a number of desirable features:

- Applies to a general class of curvilinear grids, without any requirement for an analytical and invertible function from grid space to Euclidean space.
- Does not require run-time sorting of cells.
- Leverages the topological structure of the curvilinear grid.
- Converts the problem of rendering curvilinear grids to the problem of rendering regular grids with one level of indirection.
- Enables storage of scalar data at the original sampling rate on the graphics card and avoids resampling of scalar data until pixel shading time.

2 Algorithm

As mentioned above, our approach leverages the regular topological structure of the grid to provide a natural mapping to graphics hardware. We realize that the structured data in fact lies on a rectilinear lattice in some warped space, just not in R^3 (see Figure 2a for a 2D example). Let us denote this space by the triple (s, t, r). The scalar value is sampled on a lattice: at regular intervals in s, t, and r. Indeed, a structured grid is often specified by a three dimensional array of scalar values along with an unwarping function, or a table, that specifies the R^3 location of each lattice point. Thus the time varying scalar data itself may be directly stored as a 3D texture in that warped space – call it the grid scalar texture S. Of course, this texture is not regular in R^3 and thus may not be directly rendered as textured slices. Instead, we use a regular R^3 grid to sample the inverse of the structured grid's unwarping function. The resulting warping texture, W, effectively serves as a voxel-based parameterization of R^3. Now we can find the scalar value at any point in R^3, denoted by (u, v, w), by first looking up the warp texture $(s, t, r) = W(u, v, w)$ to find its 3D warped coordinate, and then looking up the scalar value from the grid texture: $S(s, t, r)$.

This indirect texturing approach has some inherent advantages over a standard resampling algorithm. First, for Eulerian grids, which have a con-

stant warping function over all time steps, we create a single warping texture, W, to describe the parameterization of space, and re-use it for all the time-varying scalar values. Second, the scalar data itself remains as compact as in its original form, and thus requires minimal bandwidth to load to texture memory and a minimal footprint to store there. Third, the warping texture sometimes requires less resolution to maintain good quality rendering than does the direct resampling of the scalar texture. This is due to the fact that warping functions are often largely smooth and well-approximated by the tri-linear interpolation performed on the hardware. The scalar values themselves tend to have higher frequency content and many discontinuities. Furthermore, by avoiding directly resampling the scalar field, we eliminate one resampling of this data, delaying its resampling to the stage of rendering individual pixels. Any error induced by sampling the warping function has the effect of distorting the data in space rather than changing the actual scalar values portrayed. Such a change in the nature of error has been previously deemed effective in the domain of polygon mesh simplification [2]. As a result our technique is effective not only to reduce the data size for time-varying scalar fields, but also for static scalar fields.

Crawford et al.'s recent work on visualization of Gyrokinetic data [3] also takes the approach of storing the scalar data in a 3D texture domain and warping it back into Euclidean space at run-time. However, their algorithm is rather specific to the rendering of twisted torroidal domains. It involves both resampling of the scalar data and the use of a known analytical function for the torroidal warp. Our warp texture approach does not involve such resampling and does not require an analytical representation for the warp function.

Our work is foreshadowed by the work of Rezk-Salama et al. [9], which was tuned to operate on an earlier generation of graphics hardware. They deform a uniform grid of points from a base position (our grid coordinate system) into a deformed space (our Euclidean domain), and approximate the inverse transformation of this deformation by negating the translation vectors. Rendering is performed using object-aligned slicing, where a triangle mesh with deformed texture coordinates is rendered for each slice. A number of extensions are proposed at a very high level for future hardware capabilities, including the use of indirection with 3D textures to specify the warp function. Our work provides all the important and non-trivial details of this suggested approach as well as an implementation applied to the rendering of curvilinear grids. We develop algorithms for better generation of the inverted deformation, handling of the boundary conditions, measuring approximation error, etc.

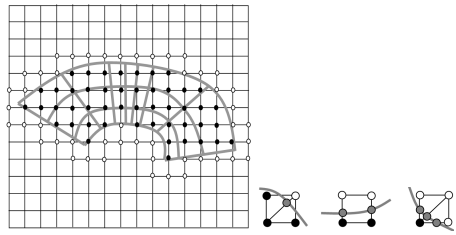

Figure 2: (a) An 8x4 curvilinear grid is shown over the regular gridlines of R^3. Black samples are centers of interior voxels, white samples those of boundary voxels, and unmarked samples those of exterior voxels. (b) Three cases of boundary matching (in 2D). The white boundary voxels are computed using extrapolation so as to match the warping function at the grey points along the structured grid boundary.

3 Warp Texture Generation

The essential data component for our rendering algorithm is the warp texture, W, a voxel-based approximation to the inverse of the curvilinear grid's unwarping function, W_i. Thus, each value in the warp texture describes a mapping back into the grid (scalar) texture domain and may be applied as texture coordinates to look up interpolated values from the original scalar data.

The generation of the warp texture is primarily a sampling process. The warp texture is defined to cover the bounding box of the curvilinear grid unwarped into the R^3 domain. (We also refer to the cells of a regular rectilinear R^3 grid as voxels.) Given some R^3 grid resolution, a voxel may be classified as interior, boundary, or exterior, according to its relationship with the warped cells of the structured grid, as shown in Figure 2a.

3.1 Sampling Interior Voxels

To compute the values of W at the centers of the interior voxels, we perform a point location query of each voxel sample (its center) within the cells of the structured grid. Such a query should report which cell, if any, contains the sample location, and where the sample is within that cell (i.e., its (s, t, r) coordinate). If there is such a cell, then the voxel is an interior voxel, and the matching grid coordinates are stored at that voxel (such as the black samples in Figure 2a).

In case the unwarp function is analytically specified and invertible, one can compute the warp function at each voxel center. In the more general case, however, when only the value of the unwarp function is provided at a set of given points in the grid space, we reconstruct the function from these samples first. This reconstruction is defined as a simple tri-linear interpolation to match hardware tri-linear filtering.

To answer the point location query, in our implementation, we tetrahedralize the structured grid. We locate the voxel center in the tetrahedron containing it (if one exists) and then use the barycentric coordinates within that tetrahedron to compute the mapping to grid space.[1]

We later use hardware tri-linear texture filtering to compute the warp function for each fragment. It is worth noting that with the prescient knowledge of this subsequent tri-linear interpolation, we might alternatively compute the voxel center sample values more intelligently to reduce the difference between the interpolant and the inverse function everywhere in between samples. However, this substantially complicates the procedure.

A variety of methods are applicable to speed up the sequence of point location queries. In order to exploit coherence, we perform the queries within the outermost grid cells first, and then walk from voxel to voxel, following stabbing lines through the tetrahedra. Spatial data structures such as octrees and k-D trees are also good candidates for this acceleration.

[1]This tetrahedralization for point-location should not be confused with algorithms that permanently tetrahedralize a structured grid as a means of performing the entire rendering process.

3.2 Extrapolating Boundary Voxels

By sampling the interior voxels, we have effectively bound their centers to their accurate positions in the grid space. This, in turn, implies that each grid sample, (s, t, r), is bound to some point $W^{-1}(s, t, r)$ in R^3, where W^{-1} is close to W_i^{-1}. However, it is especially important to ensure that the boundaries of the structured grid, which do not generally pass through the centers of voxels, are accurately rendered. We take special care at the grid boundaries. In particular, we also locate all the boundary voxels as well as reproduce a set of samples on the grid boundary precisely. Recall that a boundary voxel is one that is adjacent to an interior voxel but is not itself an interior voxel (using the convention that a voxel has 26 adjacent voxels in 3D).

We associate each boundary voxel with a lattice point on the grid boundary, as shown in Figure 2b. We choose the grid boundary point to be the intersection of the grid boundary with a line connecting the boundary voxel with one of its interior neighbors. Whenever possible, this line is parallel to one of the coordinate axes, but we use a diagonal line when no axis-aligned neighbor pairing is available.[2] For each of these intersection computations, we trace the line from the interior voxel to the boundary voxel, walking from tetrahedron to tetrahedron until the structured grid boundary intersection is found. In cases where more than one possible neighbor pairing is available, we select one arbitrarily.

Given an interior voxel, a boundary voxel, and a boundary intersection, we apply a simple extrapolation to determine the warp texture value to store at the boundary voxel.

As a result of this extrapolation, the warp texture now contains texture coordinates both inside and outside the valid grid domain. The texture coordinates outside the valid domain are used to allow us to interpolate right up to the structured grid boundary. When we come to a pixel whose interpolated value in the warp texture lookup lies outside the valid grid domain, we map that fragment to transparent black. As a result, the grid does not contribute color to any fragment outside the grid boundary.

[2]If the corners and edges of the grid domain map to sharp features in the Euclidean space, it may be advantageous to force some of the matched grid boundary points to lie along these sharp features.

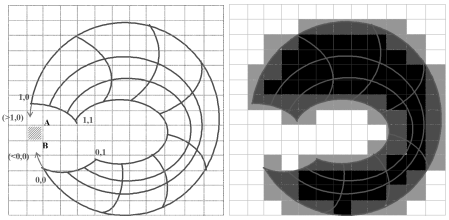

Figure 3: (a) It is possible for a voxel to be both above and below the curvilinear grid (mapped into R^3), resulting in the appearance of an extraneous grid. (b) The colors indicate how fragments within those regions are classified. White: forced to be transparent by $stencil = 0$. Light grey: eliminated due to warp values outside $[0, 1]$. Dark grey: boundary fragments with warp values in $[0, 1]$ and $stencil > 0$. Black: interior points with warp values in $[0, 1]$ and $stencil = 1$.

3.3 Stenciling Exterior Voxels

We refer to voxels that are not adjacent to an interior voxel as exterior voxels. The entire neighborhood around these exterior voxels should always be rendered as transparent black. One might like to assign to these voxels warp values outside the valid grid domain so that they too may be mapped to transparent black. However, this is not generally possible, depending on how the grid warps in R^3. For example, see Figure 3a. The warp function takes s component values greater than 1 as we proceed beyond the $s = 1$ boundary, as shown by the arrow. At the same time, s takes values less than 0 beyond the $s = 0$ boundary. This necessarily causes adjacent voxels, like A and B to get warp values greater than 1 and less than 0, respectively. Thus the interpolated values of s for points between A and B fall into the range $[0, 1]$, mapping them to points inside the structured grid.

To avoid this problem, we supplement the warping texture with a 3D binary stencil texture. This stencil texture is set to 1 at all interior voxels and 0 at all boundary and exterior voxels. At rendering time, we test the linearly interpolated value of this stencil texture, and set the pixel's output color to transparent black if the stencil texture evaluates to exactly 0 (as depicted in Figure 3b). This effectively eliminates the rendering of all exterior voxel neighborhoods, and thus we do not need to care what value is stored in the actual warp texture for exterior voxels. The extraneous grid may yet appear within a voxel if opposite boundaries pass through the same voxel. We generally avoid this by choosing a sufficient voxel resolution such that opposite boundaries are separated into separate voxels (see Section 5.3 for handling of cases where this is not possible).

This stenciling approach bears some resemblance to our prior work [6]. However, we are operating here in a different domain that requires a very different procedure. In the prior work, the scalar data is being resampled, and the regions just outside the boundary are padded with copies of the boundary scalar data, then clipped away by the stencil texture. Here we are operating on a warp texture by extrapolating the warping function to position the grid boundary in the correct location.

4 Error Computation

Apart from numerical errors, the main source of error in our technique is the approximate reconstruction of the warp-function from the samples at voxel centers. In order to measure this error, we compute the difference of the interpolant from the actual warp function at each point of R^3. For a point, p_e, in Euclidean space, we evaluate the warping texture at $W(p_e)$ using tri-linear interpolation of neighboring voxel centers to compute its corresponding point, p_g, in grid space. Now we use the original inverse warp function provided with the curvilinear grid to map p_g to its correct location, p'_e, in Euclidean space. We measure the error as $|p_e - p'_e|$, a distance of displacement in R^3. This is effectively the distance from where we are rendering a particular scalar value to where that scalar value ought to appear. If desired, this object-space distance may also be conservatively projected at runtime into screen space to indicate the warping error in pixel-sized units that account for perspective foreshortening.

It is worth noting that we have chosen to measure the error in the warp function itself. A more direct approach may be to measure the error in the resulting scalar values, if they are known in advance. This is helpful primarily if the scalar values are generally R^3-coherent – in other words, grid coordinates close in R^3 have similar scalar values. Similarly, one could measure the color error resulting from

application of the transfer function to the incorrect scalar value. However, such error measures seem less informative to us than our measurement of geometric distortion in the warping function.

5 Subtleties

The use of one level of indirection in our texture-based volume rendering algorithm gives rise to several more subtle issues that are worth discussing in more detail. These are issues related to the limited precision of the warp texture, to the polygonal slices used for rendering, and to degenerate grid specifications.

5.1 Precision Issues

When we specify texture coordinates to the graphics driver in the typical way, by binding them to individual vertices, we can send the data in a variety of formats. They ultimately arrive in floating point registers in the fragment processing unit, so we can potentially take advantage of the full floating point precision. In practice, the number of mantissa bits is a good estimate of the useful precision of a floating point texture coordinate, due to the limited range of interest, so we essentially have access to 24 bits.

However, when we specify texture coordinates as elements of an actual texture, as we do with our warp texture, our precision is limited by the supported internal texture formats. For example, if we wish to store the warp texture in a single 3D texture on our current hardware (NVIDIA GeForce 6800), we can choose from a four-channel texture with either 8 bits integer precision per channel or 16 bits floating point precision per channel.[3] The four channels may be used to store the three texture coordinate dimensions plus the stencil texture. The 8 bit option is rather small. 16 bits floating point somewhat better, although it is difficult to make good use of the exponent bits. The use of multiple textures (and thus multiple texture lookups), allows us to use integer texture formats with 16 bits per channel, which currently gives us the best quality, at the expense of more texture lookups. We can expect increased support for the 32-bit floating point texture formats in future graphics hardware, so this problem will likely go away soon.

[3] 32-bit floating point textures do not yet support trilinear interpolation.

5.2 Slicing Distance

Another interesting issue which arises in our setting is the choice of a distance to use between polygonal slices during rendering. In a standard voxel grid, all the scalar samples are equally spaced, so it is straightforward to choose a reasonable spacing of planes to incorporate all the data. In our case, however, the scalar data is still non-uniformly distributed in R^3. This makes choosing an inter-plane distance a tougher problem. One useful approach is to vary the inter-plane distance according to the portion of the space being sliced (i.e. the distance between successive pairs of planes varies). Our current approach is to incorporate progressive refinement as well as user-controlled refinement, which works reasonably well in an interactive system. Using our warp texture formulation in a hardware-accelerated ray caster may provide increased opportunities for adaptive step sizes.

5.3 Grid Degeneracies

In some cases, the warping of the grid into Euclidean space as specified by the original grid data may contain degeneracies. For example, the grid shown in Figure 3 could close into a toroidal configuration. In this case, there is no way to keep the boundaries separate during rendering. Such a degeneracy occurs, for example, in the NASA Oxygen Post data set. In this case, it is generally possible to split the original grid into multiple grids such that no single grid has a degenerate configuration. We then sort these grids with respect to each other at render time and render them in sorted order. This approach may also be used to handle curvilinear grids with multiple zones. The actual implementation of this is ongoing work for us.

6 Implementation And Results

We have implemented the warp texture generation algorithm as well as the rendering algorithm on a Windows PC equipped with 2.8 GHz Intel Xeon CPU, 2 GB RAM, AGP 8X bus, and an NVIDIA Geforce 6800 GT with 256 MB VRAM. 3D texture-based volume rendering is performed with viewport-aligned slicing. A custom fragment program uses the R^3 location as an index into the warp and stencil texture. The results of the warp texture lookups are scaled and biased, then used

to perform the scalar texture lookup. The scalar texture, which has been quantized to one byte per scalar, is then used to index a final transfer function texture, which maps the scalar value to RGBA according to a user-specified transfer function. The transfer functions may be mapped linearly or logarithmically across the scalar range, and the user may explicitly update the quantization to reflect the current range of interest.

The preprocessing time for generating the warp texture was less than 20 minutes for the highest resolution warp textures and much less for the lower resolution ones. We have not yet devoted significant effort to optimizing this stage because the time is trivial compared to the time to generate the simulation data (e.g., 50 hours times 288 CPUs, or 14,400 CPU hours, to generate the KDPhrd data set), and can be performed in parallel with the actual simulation computation if desired.

Figure 4 presents some data for several test models. The BluntFin and Tapered Cylinder (Figure 6) are well known data distributed by NASA. KDPhrd (Figure 7 is a simulation of turbulent gas around a black hole.

We see that even small resolutions for the warp texture provide reasonable accuracy. The mean error remains rather small with respect to the warp texture voxel size.

We timed the run-time performance of our warp texture volumes using a window size of 640x480, with the object just filling that window and rotating. We use `glTexSubImage3D()` to load the next time step's scalar data before rendering each frame. For consistency, we use 800 slices for all these timing tests. Because we are generally fragment bound, the frame rate generally varies with the number of fragments. We currently perform two two-channel, 16-bit integer texture lookups[4] to fetch the warp and stencil textures before looking up the scalar and transfer function in two additional lookups. The number of unnecessary fragments is directly related to the percentage of empty volume in the object's bounding box (these fragments may be killed before actually looking up the scalar data and transfer function). Due to our warp texture approach, which keeps the scalar texture size small, the texture loading is not a bottleneck. Should the simulation grid size exceed the bus bandwidth for some application, it may be more appropriate to employ a scalar field compression based on vector quantization of each voxel over some number of time steps [7].

We also present a set of visual quality comparisons in Figure 5. The two images rendered using warp textures exhibit warping error as well as some ringing artifacts at the boundary of the spherical cut-out at the core of the original domain. The images rendered using direct resampling exhibit scalar data errors that become color errors. We believe the warping error is preferable – its size shrinks with the distance of the viewer to the images, as you can see by stepping back from your paper or computer monitor. The resampling error remains objectionable even as it becomes smaller on your retina. Note that the goal of this comparison is not to compare warp resolution to direct resampling resolution. Recall that the warp texture resides in texture memory throughout visualization, with only the original scalar resolution loaded for each time step (e.g., 192x192x64), whereas the direct resampling method requires the resampled resolution to be loaded at each time step.

[4]Driver and hardware advances should soon reduce this to a single lookup.

7 Conclusions and Future Work

We have presented and demonstrated a new 3D texture-based algorithm for visualization of time-varying curvilinear grids. By factoring a static warping function out from the time-varying scalar data, we essentially convert the problem of rendering curvilinear grids to the problem of rendering regular grids with an additional level of indirection. The use of a precomputed 3D warp texture serves as an accelerated point location query from Euclidean space into grid space. Our approach eliminates the need to resample the scalar data for each time step and maintains a relatively low space requirement for storing and transmitting the data.

We are currently developing methods for assigning adaptive warp texture resolution across the Euclidean domain to reduce error in a localized fashion and also for dealing with more degenerate and multizone data sets. Although we have demonstrated the use of our warp texture in the context of slice-based volume rendering of curvilinear grids, it should be useful in other contexts as well, such as ray casting of curvilinear grids, particle and streamline tracing through curvilinear grids, etc.

Model	Grid resolution	Bounding box size	Time steps	Warp resolution	Mean error	Std. deviation	Frame rate
KDPhrd	192x192x64	117.2 x 119 x 231.8	2000	128x128x256	.002389 %	.0006710 %	31.7
				256x128x256	.002341 %	.0006433 %	25.5
				256x256x256	.002247 %	.0006955 %	21.0
BluntFin	40x32x32	22.87x8.59x5.9	1	64x64x64	.02095 %	.01853 %	75.2
				128x128x128	.008506 %	.01062 %	74.1
				256x128x128	.006773 %	.01103 %	62.5
Cylinder	64x64x32	49.5x49.5x32.28	400	128x128x64	.01938 %	.2106 %	35.3
				128x128x128	.01893 %	.02106 %	35.2
				256x256x128	.01502 %	.005253 %	25.1

Figure 4: Statistics for several data sets. Grid resolution is the number of data points in the curvilinear grid. Time steps is the number of scalar data values available per grid point. Warp resolution is the resolution of the created warp texture. Mean error and standard deviation are measured as a percentage of the bounding box diagonal (i.e., error/diagonal*100). Frame rates are for time-varying data, using 640x480 window resolution and 800 slices.

8 Acknowledgments

We would like to thank Julian Krolik and John Hawley for providing the astronomical data which motivates this work and also NASA for the blunt fin and tapered cylinder data. This work was supported in part by NSF ITR Grant AST-0313031.

References

[1] S. Callahan, M. Ikits, J. Comba, and C. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[2] J Cohen, M Olano, and D Manocha. Appearance-preserving simplification. In Michael Cohen, editor, *SIGGRAPH 98*, Annual Conference Series, pages 115–122, Orlando, FL, 1998. Addison Wesley.

[3] D Crawford, K Ma, M Huang, S Klasky, and S Ethier. Visualizing gyrokinetic simulations. In *IEEE Visualization 2004*, pages 59–66, 2004.

[4] L Hong and A Kaufman. Accelerated ray-casting for curvilinear volumes. In *IEEE Visualization '98.*, pages 247–254, 1998.

[5] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *IEEE Visualization 2003*, pages 287–292, 2003.

[6] J. Leven, J. Corso, J. Cohen, and S. Kumar. Interactive visualization of unstructured grids using hierarchical 3d textures. In *IEEE/SIGGRAPH Symposium and Volume Visualization and Graphics 2002*, pages 37–44, 2002.

[7] E. Lum, K. Ma, and J. Clyne. Texture hardware assisted rendering of time-varying volume data. *IEEE Visualization 2001*, pages 255–262 and 562, 2001.

[8] N. Max, P. Williams, and C. Silva. Cell projection of meshes with non-planar faces. In *Data Visualization: The State of the Art*, pages 157–168. Kluwer, 2003.

[9] C. Rezk-Salama, M. Scheuering, G. Soza, and G. Greiner. Fast volumetric deformation on general purpose hardware. In *2001 ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 17–21, 2001.

[10] S. Roettger, S. Guthe, A. Schieber, and T. Ertl. Convexification of unstructured grids. In *Proceedings of Workshop on Vision, Modeling, and Visualization 2004*, pages 283–292, 2004.

[11] A. Van Gelder. Rapid exploration of curvilinear grids using direct volume rendering. In *IEEE Visualization '93*, pages 70–77, 1993.

[12] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH 98*, pages 169–177, 1998.

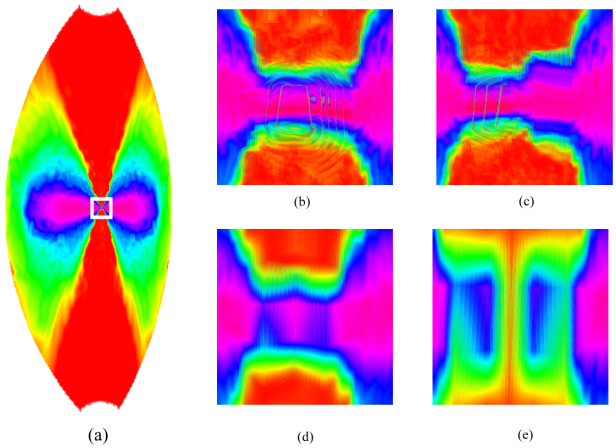

Figure 5: KDPhrd astronomical simulation data with density values logarithmically mapped to hue. (a) White square indicates zoom-in region for b through e, which is the site of the most densely space grid samples. (b) Warp texture resolution 256x256x256. (c) Warp texture resolution 128x128x128. (d) Direct resampling at resolution 256x256x256. (e) Direct resampling at resolution 128x128x128. The error for warp-based renderings is visible as geometric distortions, as opposed to the color errors present in direct resamplings.

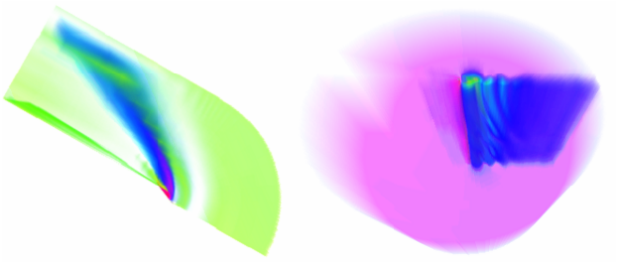

Figure 6: Rendering of the NASA BluntFin with a 128x128x256 warp texture and Tapered Cylinder with a 256x256x128 warp texture.

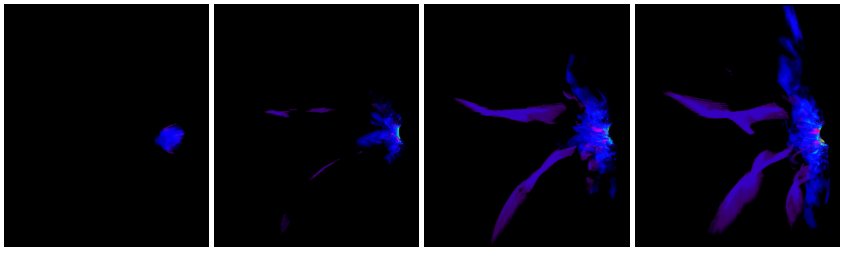

Figure 7: Several time steps in a temporal sequence from the KDPhrd astronomical simulation data. The scalar field represents the Poynting flux.