Efficient Rendering of Extrudable Curvilinear Volumes

Steven Martin* Ohio State University
Han-Wei Shen† Ohio State University
Ravi Samtaney‡ Princeton Plasma Physics Laboratory

ABSTRACT

We present a technique for memory-efficient and time-efficient volume rendering of curvilinear adaptive mesh refinement data defined within extrudable computational spaces. One of the main challenges in the ray casting of curvilinear volumes is that a linear viewing ray in physical space will typically correspond to a curved ray in computational space. The proposed method utilizes a specialized representation of curvilinear space that provides for the compact representation of parameters for transformations between computational space and physical space, without requiring extensive preprocessing. By simplifying the representation of computational space positions using an extrusion of a profile surface, the requisite transformations can be greatly simplified. Our implementation achieves interactive rates with minimal load time and memory overhead using commodity graphics hardware with real-world data.

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1 INTRODUCTION

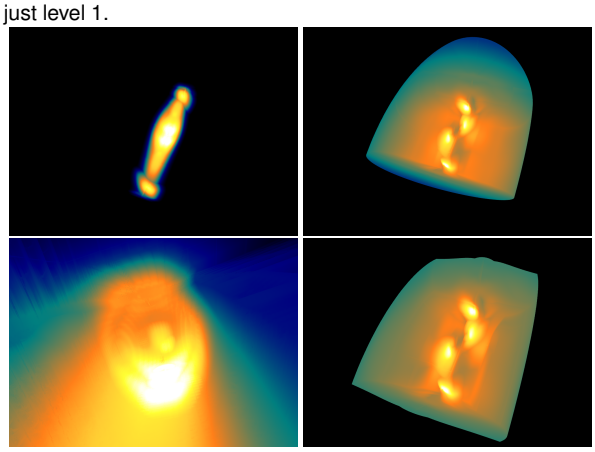
Ray casting through curvilinear adaptive mesh refinement volumes requires a large number of transformations between computational and physical space. While this is trivial for rectilinear computational spaces, curvilinear computational spaces present additional challenges. By exploiting the characteristics of a particular class of curvilinear spaces, we enable volume rendering at interactive frame rates with minimal preprocessing and memory overhead using commodity graphics hardware.

The core contribution of our technique is its representation of a curvilinear space as an extrusion of a profile surface along a curve, permitting memory and time-efficient transformations between physical space and computational space. Our technique renders the data in blocks, where each block is a curvilinear grid of cells(for example, a block may be 64x64x64 cells). Blocks can have different sampling resolutions, and information is passed to the renderer to permit block hierarchies. The borders of each block are rendered to the framebuffer using triangles with vertices specified in computational space. A vertex program is utilized to transform the vertices from computational space to physical space such that for a given computational space position in any given level, the physical space position produced is consistent. Ray-casting is performed within the fragment program for every fragment rendered, stepping the ray simultaneously through both computational space and physical space. The step size is found by computing intersections with cell boundaries in computational space then applying a minimum step length constraint. The physical space step vector is trivially determined by the direction from the camera origin to the fragment position, and the computational space step vector is found by applying a Jacobian matrix, which is easy to derive using our specialized representation, to the physical space step vector. Several constraints were considered in the design of this method. Memory-efficiency is of great importance to performance, both because of limited memory capacity and limited memory bandwidth. Additionally, any representation of computational space must be easily traversable without a significant loss in accuracy. Finally, the inexpensive computational power available on GPUs today must be exploited, despite its limitations, to be competitive with other techniques.

As data sets from simulations become larger, memory efficiency of rendering techniques becomes more important. Additionally, as computational power becomes more densely packaged in devices such as GPUs, the disparity between computation and memory performance becomes greater, necessitating the use of techniques which facilitate efficient cache utilization. Given these constraints, direct rendering techniques for curvilinear data rather than techniques that resample the data into rectilinear meshes or decompose the data into tetrahedral meshes make more practical sense.

A potential application of curvilinear adaptive mesh refinement volume rendering is exhibited in section 3. Section 4 describes the proposed specialized computational space representation. Then, section 5 provides details about the rendering process and volume data structure and details about the ray casting algorithm are described in section 5.1. Finally, results are examined examined in section 6.

Figure 1: Sample volume renderings of data set 1. The left column shows two views of one data component. The right column shows two different AMR level ranges for a different component, with the top image showing levels 0 through 1, the bottom image showing just level 1.



2 RELATED WORK

Many methods exist for volume rendering of curvilinear data. Many of them can be adapted to support adaptive mesh refinement data sets, and some can be easily implemented such that they take advantage of GPU capabilities. Four potential methods are resampling to rectilinear space, decomposition of curvilinear data into unstructured tetrahedral data, direct cell projection, and direct ray casting.

*e-mail:martinst@cse.ohio-state.edu
†e-mail:hwshen@cse.ohio-state.edu
‡e-mail:samtaney@pppl.gov

A good survey of techniques for volume rendering is provided in [5].

Techniques for volume rendering of rectilinear data are the most well-developed and tend to be the most straightforward due to the simplicity of the mesh. [9] presents a GPU ray casting implementation for rectilinear data. [11] proposes GPU data structures for efficient volume rendering of rectilinear AMR data. [4] presents acceleration structures for supporting empty space skipping and early ray termination for GPU rendering of rectilinear data. Resampling of the curvilinear data into a rectilinear mesh offers the obvious advantage of enabling these well-developed techniques at the cost of introducing extra sampling error, increasing memory consumption, and reducing potential performance due to memory bandwidth requirements. Additionally, the level of preprocessing required may be unacceptable for large time-varying datasets.

Decomposition of the curvilinear grid into an unstructured tetrahedral mesh is straightforward and enables the usage of the large body of work on unstructured tetrahedral mesh rendering. [13] proposes a point-based approach for rendering unstructured meshes. [1] and [8] propose rendering techniques for tetrahedral elements. [6] presents a technique for rendering unstructured grids using graphics hardware-assisted incremental slicing. However, in this process of decomposition, potentially useful data for accelerating rendering may be lost and, if the decomposition is done as a preprocessing step, excessive memory consumption may result. Additionally, unstructured tetrahedral meshes introduce additional challenges for the evaluation of depth-order-dependent transfer functions.

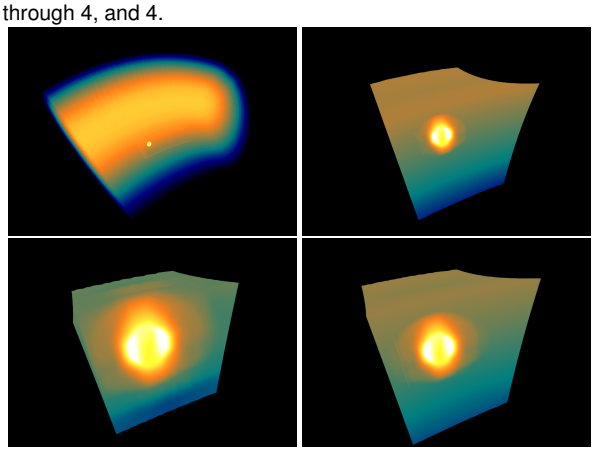
Curvilinear direct cell projection offers the potential of avoiding preprocessing and utilizing the curvilinear structure of the data to accelerate sorting and rendering. However, direct cell projection requires a significant amount of non-localized overdraw which, when implemented on modern GPUs, can reduce performance due to memory bandwidth limitations. Additionally, implementation of direct cell projection requires a significant amount of vertex data to be manipulated, further increasing the required memory bandwidth and vertex processing requirements for rendering.

Curvilinear direct ray casting offers the same potential as curvilinear direct cell projection for reduced preprocessing and data loss, while being more adaptable for implementation on modern GPUs. [12] presents a ray casting technique for direct curvilinear volume rendering and compares it to resampling to a rectilinear mesh. [3] and [2] present additional methods utilizing ray casting. [10] proposes using textures for a transformation from physical space to computational space. Our technique provides a compact representation of the mesh for transformations from physical space to computational space as well as from computational space to physical space while reducing the memory required for mesh specification by 100 times or more. Sorting, for single block non-AMR data sets, is implied and image quality can be smoothly changed to permit a user-driven compromise between speed and quality.

3 Applications

ITER ("The Way" in Latin), a joint international research and development project that aims to demonstrate the scientific and technical feasibility of fusion power, is now under construction at Cadarache, France. Refueling of ITER is a practical necessity due to the burning plasma nature of the experiment, and longer pulse durations (100 - 1000 seconds). An experimentally proven method of refueling tokamaks is by pellet injection. Pellet injection is currently seen as the most likely refueling technique for ITER. Thus it is imperative that pellet injection phenomena be understood via simulations before very expensive experiments are undertaken in ITER. The emphasis of the present work is to understand the large-scale macroscopic processes involved in the redistribution of mass into a tokamak during pellet injection. In particular, it was experimentally established that high-field-side (HFS, or inside) pellet lauches are more effective than low-field-side (LFS or outside) pellet launches. Arguably, such large scale processes are best understood using magnetohydrodynamics (MHD) as the mathematical model.

Figure 2: Sample renderings from data set 2. In clockwise direction from the top left corner are AMR levels 0 through 4, 2 through 4, 3 through 4, and 4.



There is a large disparity between the pellet size and device size. Naive estimates indicate that the number of space-time points required to resolve the region around the pellet for simulation of ITER-size parameters can exceed 10^{19}. The large range of spatial scales and the need to resolve the region around the pellet is somewhat mitigated by the use of Adaptive mesh refinement (AMR). Our approach is to employ block structured hierarchical meshes using the Chombo library for AMR developed by the APDEC SciDAC Center at LBNL.

We use data from simulations performed with an adaptive upwind conservative mesh MHD code in generalized curvilinear coordinates. A critical component is the modeling of the highly anisotropic energy transfer from the background hot plasma to the pellet ablation cloud via long mean-free-path electrons along magnetic field lines. Further details on the approach can be found in [7].

A primary scientific question is establishing the MHD mechanisms responsible for the differences in HFS and LFS pellet launches. Visualizations of the density field helps identify the extent of the migration of the ablated pellet mass along the magnetic field lines and more importantly, the transport across magnetic flux surfaces in the direction of increasing major radius. In particular volume rendering of the density field is an effective method to visualize the global mass distribution in the tokamak during pellet injection.

4 Computational Space Representation

Our technique supports rendering of volumes that can be represented via an extrusion of a planar profile surface along a curve to create the physical space grid as a function of computational space. For example, torus can be represented as planar radially sampled circle extuded along another circle. A cylinder can be represented by a radially sampled circle extruded along a line. The tokamak shape used in the MHD simulation data presented in section 3 is another potential application.

4.1 Positional transformations

Equation 1 transforms a point from computational space to physical space. This transformation is needed to derive the Jacobian, as well as to compute the distance between a given point in physical

space and a point in physical space corresponding to a given point in computational space.

$$\bar{q}(i,j,k) = \bar{p}(k) + s_u(i,j)[\hat{n}(k) \times \hat{y}] + s_v(i,j)[\hat{y}] \quad (1)$$

where

$\bar{q}(i,j,k)$ is the physical space 3D position of a point i,j,k in computational space.

$\bar{p}(k)$ is a physical space position as a function of k in computational space that is the origin of each slice plane of the extruded volume.

$\bar{n}(k)$ is a physical space unit vector as a function of k in computational space that represents the normal to each "slice" of the extruded volume.

$\bar{s}(i,j)$ is the planar profile surface.

$\bar{y}$ is the "up" vector of computational space. All points in $\bar{p}(k)$ lie within a plane with $\bar{y}$ as its normal.

The $p(k)$, $n(k)$, and $s(i,j)$ can be either derived from a given mesh, or user-specified separately. The $p(k)$ and $n(k)$ functions are represented as one-dimensional sampled data, while $s(i,j)$ is represented as two-dimensional sampled data. If it is necessary to derive these functions from a given mesh, the following process is used:

1. Find $p(k)$ as in equation 2. This is the mean of all given sample positions in physical space in slice k. It does not matter if the point is in the exact center of slice, as it is just a reference point that must lie in the plane of the slice.

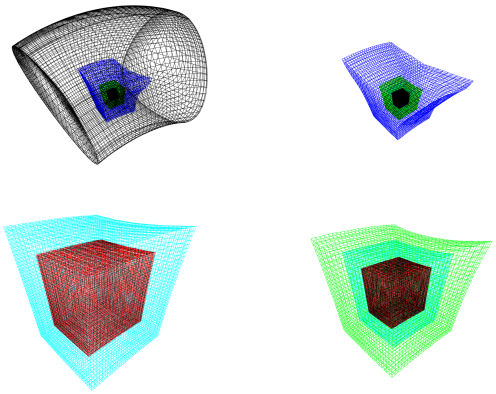
2. Find $n(k)$ via numerical methods. This is the normal to the planar slice. Random triplets of sample points are chosen in each slice, and two vectors(sharing one of the three points as a common origin) formed for each triplet. A cross product is applied on those vectors, the sign of the vector adjusted such that it faces forward from the slice(using a simple forward or backward difference reference vector), and the result accumulated into an accumulator vector. The resulting per-slice accumulated vector is normalized, resulting in an accurate slice normal.

3. Pick $\bar{y}$. All $p(k)$ should lie within a plane with normal $\bar{y}$. In our test data, $\bar{y}$ was simply the y-axis. If $\bar{y}$ is not known initially, it can be found in a manner similarly to $n(k)$, but using the $p(k)$ values instead of sample positions.

4. Find $s(i,j)$. Any k slice within the volume can be picked to form this function, and in practice $k = 0$ is used. A coordinate system defined by $p(k)$, $\bar{y}$ and $n(k)$ is defined at each slice, and the sample points are projected onto the axes of that coordinate system to find $s(i,j)$ positions, as in equation 3.

While the computational to physical space transformation is very straightforward to implement with this specialized representation, the physical space to computational space transformation is not. An analytical inverse to the above function is often impractical, and producing a 3D sampled volume in physical space mapping the positions to computational space would impart excessive memory requirements while potentially introducing inconsistencies at the edges of the valid data. However, [10] did implement the physical space to computational space transformation using 3D textures.

Figure 3: Data set 2 volume block bounding wireframes. Each vertex corresponds to a grid-centered position on the boundary. The wireframes demonstrate the curvature and non-uniform cell sizes of the curvilinear space. Level 0 has 8 distinct blocks, level 1 has 24 distinct blocks.



The proposed rendering algorithm requires only minor corrections to computational space positions. Thus, Equation 1 can be applied to find a distance between a given physical space point and a physical space point corresponding to a given computational space point which can then be used as a convergence test for a gradient descent algorithm, obviating the need for a full physical space to computational space transformation.

For each slice $k = 0..N_k - 1$,

$$p(k) = \frac{1}{N_i N_j} \sum_{i=0}^{N_i} \sum_{j=0}^{N_j} d_{\text{phypos}}(i,j,k) \quad (2)$$

where

$\bar{N_{ijk}}$ is the dimensionality of data function $d_{\text{phypos}}(i,j,k)$ in i, j, and k directions

$d_{\text{phypos}}(i,j,k)$ is the physical space position of a given point in the given data.

$$s(i,j) = \begin{bmatrix} [n(k) \times \bar{y}] \cdot [d_{\text{phypos}}(i,j,k) - p(k)] \\ \bar{y} \cdot [d_{\text{phypos}}(i,j,k) - p(k)] \end{bmatrix} \quad (3)$$

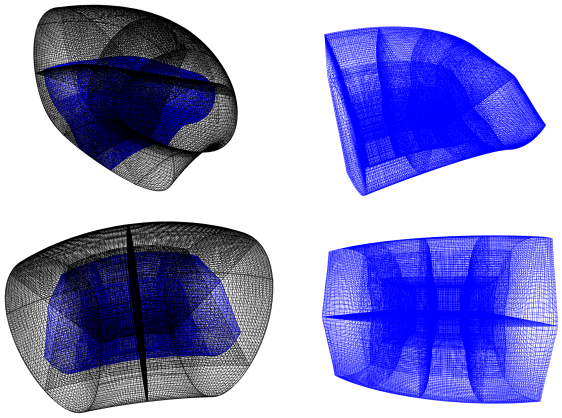
4.2 Jacobian matrices

As a ray is stepped through physical space, a corresponding ray must be stepped through computational space. Because computational space is curvilinear, a straight ray in physical space will, in general, correspond to a curved ray in computational space. While gradient descent could be applied with the computational space to physical space transformation to compute corresponding computational space points, our representation of computational space positions permits the easy computation of Jacobian matrices to transform physical space vectors to computational space vectors which can be used to directly transform steps.

Though we need the inverse Jacobian matrix to transform a vector from physical space to computational space(J^{-1}), that matrix is hard to directly compute given our representation of computational space. However, it is easy to compute J then invert that 3x3 matrix.

Equations 4, 5, and 6 form the i, j, and k columns(respectively) of the Jacobian matrix J as in equation 8. The i and j columns effectively are dealing with changes within a single slice, where as

Figure 4: Data set 1 volume block bounding wireframes. Each vertex corresponds to a grid-centered position on the boundary. The left column shows AMR levels 0 and 1, while the right column shows AMR level 1. The wireframes demonstrate the curvature and non-uniform cell sizes of the curvilinear space. Level 0 has 8 distinct blocks, level 1 has 24 distinct blocks.



the k column deals with changes between multiple slices. Because $\hat{y}$ is constant in equation 6, equation 6 can be simplified to 7.

$$J_{pc_i}(i,j,k) = [\hat{n}(k)\times\hat{y}][\frac{\delta s(i,j)}{\delta i}]_u + \hat{y}[\frac{\delta s(i,j)}{\delta i}]_v \quad (4)$$

$$J_{pc_j}(i,j,k) = [\hat{n}(k)\times\hat{y}][\frac{\delta s(i,j)}{\delta j}]_u + \hat{y}[\frac{\delta s(i,j)}{\delta j}]_v \quad (5)$$

$$J_{pc_k}(i,j,k) = \frac{\delta p(k)}{\delta k} + s(i,j)_u\frac{\delta[\hat{n}(k)\times\hat{y}]}{\delta k} + s(i,j)_v\frac{\delta\hat{y}}{\delta k} \quad (6)$$

$$J_{pc_k}(i,j,k) = \frac{\delta p(k)}{\delta k} + s(i,j)_u\frac{\delta[\hat{n}(k)\times\hat{y}]}{\delta k} \quad (7)$$

$$J(i,j,k) = \begin{bmatrix} J_{pc_i} & J_{pc_j} & J_{pc_k} \end{bmatrix} \quad (8)$$

$$J^{-1}(i,j,k) = J(i,j,k)^{-1} \quad (9)$$

Because the matrix is a full 3x3 matrix, a general determinant-based matrix inversion is used to find J^{-1}. Note that the input data should have well-formed positions, with no two sample points in computational space lying at the same physical space position, to guarantee that this matrix is always invertible.

4.3 AMR integration

Adaptive mesh refinement is supported by defining, for a given level of detail, axis-aligned cuboid regions in computational space that are said to be owned by a lower level. This enables the ray caster to easily determine whether a particular sample cell should be accumulated or not. Because consistency is required in the positional transformations between levels, the $p(k)$, $n(k)$, and $s(i,j)$ functions must be defined in a way such that their domain contains the union of all levels of detail. Because the resolution needs to be uniform for each of the functions, $p(k)$ and $n(k)$ will need to be specified at the lowest level of detail in the k direction, and $s(i,j)$ needs to be defined at the highest available level of detail.

5 Rendering

The data is stored as a hierarchy of blocks, as can be seen in figures 4 and 3. Each block is an axis-aligned cuboid in computational space with a 3D uniform grid of sample points. In physical space, the blocks will tend to be curved as shown in the figures. Associated with each block is a set of child blocks, which define their bounds in computational space. Each one of these blocks can have the currently selected data component of interest stored into a 3D texture, with single-component voxels.

Rendering is performed recursively on a per-block basis. The faces constituting the borders to the block are rendered in computational space via two triangles for each grid centered boundary cell, using a vertex program to evaluate the computational space to physical-space transformation for the vertices, and a fragment program to perform the ray casting. Back face culling eliminates the rendering of faces not facing the viewer on a per-triangle rather than per-block-side basis because the blocks may have significant curvature in physical space. The physical space and approximate computational space positions are passed to the fragment program, providing starting points for the ray to be evaluated for every fragment. Additionally, the bounds of child blocks are passed to the fragment program as well to permit AMR rendering. The depth buffer is used for tracking evaluated field values for maximum intensity projection.

With this technique, only minimal preprocessing is required. The following are some use cases and the required processing for each:

Initial load: The textures for the $p(k)$, $n(k)$, $s(i,j)$, $\frac{\delta p(k)}{\delta k}$, $\frac{\delta n(k)}{\delta k}$, $\frac{\delta s(i,j)}{\delta i}$, and $\frac{\delta s(i,j)}{\delta j}$ functions need to be created. If these functions are not already specified as part of the source data, they will require one full iteration through all data points at the lowest level of detail, and one full iteration through a single slice of the data points at the highest level of detail. Additionally, the texture that contains the field data for the currently selected component needs to be created.

Different data component selected: The single component 3D texture that contains the field data for each block needs to be reloaded with the new component.

Data modified, positions left intact: The single component 3D texture that contains the field data for each block needs to be reloaded with the new component.

Camera/view change: No re-preprocessing needs to be performed – the scene can simply be re-rendered.

Changing mesh: If the volume data positions are changed in a way that cannot be represented with a simple affine transformation, but the volume data is not, the $p(k)$, $n(k)$, $s(i,j)$, $\frac{\delta p(k)}{\delta k}$, $\frac{\delta n(k)}{\delta k}$, $\frac{\delta s(i,j)}{\delta i}$, and $\frac{\delta s(i,j)}{\delta j}$ functions need to be rebuilt. However, the 3D volume textures storing the field data do not need to be modified if that field data is not modified.

The majority of time during rendering is spent executing the fragment programs that perform ray casting. This creates potential for effective image-space parallelization.

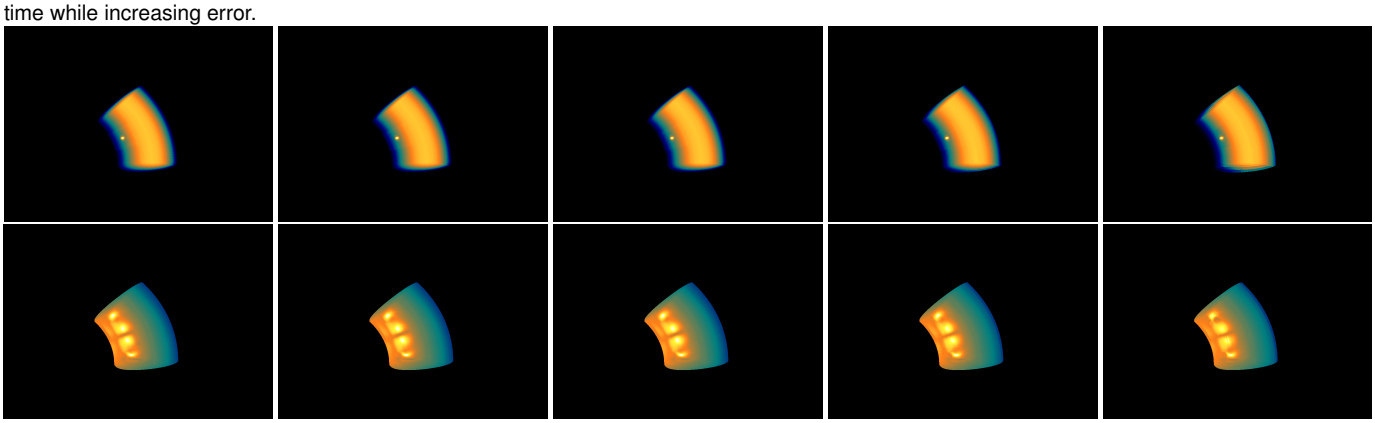
5.1 Ray Casting

Ray casting is performed for every fragment generated by the border triangle rasterization. Each of those fragments has associated physical space and approximate computational space positions interpolated between the vertices by the rasterizer that form the starting point for a ray within a given block. Each fragment program execution performs ray casting for a single ray through a block.

1. Compute the block-local computational space position.

$$p_{\text{loccom}} = \frac{p_{\text{com}} - p_{\text{blkmin}}}{p_{\text{blkmax}} - p_{\text{blkmin}}}$$

Figure 5: Volume renderings for different minimum step lengths. Each row from left to right show step lengths 0.001,0.005,0.010,0.050,and 0.100. The top row shows data set 2 and the bottom row shows data set 1. A larger minimum step length decreases required computational time while increasing error.



2. Compute the global unscaled computational space step using the Jacobian.

$$v_{\text{comstep}} = J^{-1} v_{\text{phystep}}$$

3. Compute the block-local computational space unscaled step.

$$v_{\text{loccomstep}} = \frac{v_{\text{comstep}}}{p_{\text{blkmax}} - p_{\text{blkmin}}}$$

4. Compute the scaled computational space and physical space steps(section 5.3)

5. Check whether the the computational space position of the ray lies within a child volume. If it does not, sample the field texture and accumulate the field texture using a maximum intensity projection rule. A simple list of volumes that are axis aligned cuboids in computational space, each defined by a minimum and maximum point, defines the child volumes to a given block.

6. Increment the computational space and physical space position vectors by the scaled steps

7. Compute the block-local computational space position

8. Apply the correction(section 5.2) loop to the computational space position.

9. Check if that position is within the bounds of the block. If it is not, terminate the ray loop and write the resulting color value and maximum field value to the fragment program color and depth results respectively.

10. Goto 2

5.2 Correction loop

Because the current computational space position was computed using a linear approximation(the Jacobian) of the step in computational space for the step in physical space, an error is inherent in it. The correction loop corrects this error by iteratively transforming the reverse error vectors in physical space to computational space then accumulating them with the current computational space position. This is very similar to applying gradient descent for minimization of the distance between a given physical space point and a physical space point that corresponds to a varying computational space position, except the gradient is not being computed numerically in computational space, reducing the number of texture fetches required.

1. Transform the current computational space position(p_{com}) in to a physical space position(p_{truephy}, the current "true" physical space position), using equation 1.

2. Compute the physical space correction vector using the current physical space position and the current "true" physical space position.

$$v_{\text{phycor}} = p_{\text{truephy}} - p_{\text{phy}}$$

3. If the magnitude of v_{phycor} is below a specified tolerance, break from the correction loop.

4. Transform the physical space correction vector into a computational space correction vector using the Jacobian.

$$v_{\text{comcor}} = J^{-1} v_{\text{phycor}}$$

5. Accumulate the computational space correction vector v_{comcor} with the current computational space position p_{com}. A scaling factor(γ) is applied to the correction vector to improve the rate of convergence.

To improve performance for previewing of volumes, a hard iteration limit can be applied to this correction loop. This will have some implications in accuracy, but for the purposes of previewing may be acceptable.

5.3 Step length determination

While a uniform step length can be used successfully, variable step lengths can provide for greater performance. With curvilinear data, cells may vary in size greatly, so the proper step size through the data should also vary. Our technique computes the approximate intersection points in computational space with the borders of a given cell to find the necessary step to the next cell. Performing the intersection in computational space greatly simplifies the operation, because the cells unit cubes in rectilinear space rather than six-faced curved volumes in physical space.

1. Compute the cell ceiling and floor for that step using equations. These are needed to find the intersection with neighboring cells.

$$p_{\text{comcellceil}} = \begin{bmatrix} \lceil N_i p_{\text{loccom}_i} \rceil \\ \lceil N_j p_{\text{loccom}_j} \rceil \\ \lceil N_k p_{\text{loccom}_k} \rceil \end{bmatrix}$$

$$p_{\text{comcellfloor}} = \begin{bmatrix} \lfloor N_i p_{\text{loccom}_i} \rfloor \\ \lfloor N_j p_{\text{loccom}_j} \rfloor \\ \lfloor N_k p_{\text{loccom}_k} \rfloor \end{bmatrix}$$

2. Compute the intersection with the neighboring cells in computational space to find the proper step size using the computational space step vector, $p_{\text{comcellceil}}$, and $p_{\text{comcellfloor}}$. This is done by computing the intersection with each face plane with the v_{comstep}, then using the intersection lowest positive intersection parameter. Some special cases must be handled with the intersections. If a ray is traveling parallel to or even within a given face, intersections should not be computed against that face. Also, if a ray intersection with a face would result in a time parameter of zero, the ray should not be intersected with that face. Additionally, a scaling factor greater than but close to 1 needs to be applied to the resulting intersection time to reduce the likelihood that an intersection point will lie exactly on a face.

3. Apply the minimum step length constraint to the intersection parameter. If the intersection parameter is less than the minimum step length, then it it is set to the minimum step length. This is to permit user configurability of quality.

4. Using that intersection parameter, scale the computational space and physical space step vectors(v_{comstep} and v_{phystep} into their scaled forms.

In practice, it was found that the linear approximation to a ray within a given cell did not introduce noticeable error. A future extension to this step length determination method could set a maximum step length that is a function of the curvature of the space in that cell to reduce the amount of error. Additionally, because the step length is chosen based on cell boundary intersections, only a test for whether the origin of a given ray step is within a child volume is required to support adaptive mesh refinement.

5.4 GPU implementation

Block boundaries are rasterized with OpenGL using GLSL vertex and fragment programs. The depth buffer is used for compositing the different blocks with maximum intensity projection. Equations 1, 8, and 9 are implemented within fragment and vertex programs. In total, 6 small textures are used to represent the parameters for defining the transformations between computational and physical space, and the volume samples for each block are stored in a 3D volume texture. Each block has an associated list of child block bounding regions in computational space which is passed to the vertex and fragment programs.

The positional functions $p(k)$, $n(k)$, and $s(i,j)$ can each be defined by a texture. $p(k)$ is a one-dimensional texture with resolution N_k and 3 components per texel. $n(k)$ is a one-dimensional texture with resolution N_k and 3 components per texel. It is possible to reduce $n(k)$ to two-components per texel given that $n(k)$ is a unit vector, but on current graphics hardware this would not yield any performance improvement because the memory requirements wouldn't be significantly changed, yet additional computation would be required to renormalize the values. $s(i,j)$ is a two-dimensional texture with resolution N_i x N_j, with two components per texel.

While the derivatives for the Jacobian matrices can be derived within the fragment program, a significant performance penalty was found to be incurred by the required number of texture fetches and conditionals required for handling boundary cases. Instead, the derivatives are sampled and stored in textures. The derivatives $\frac{\delta p(k)}{\delta k}$ and $\frac{\delta n(k)}{\delta k}$ are each stored in their own one-dimensional, three-component textures. $\frac{\delta s(i,j)}{\delta i}$ and $\frac{\delta s(i,j)}{\delta j}$ are combined into a single two-dimensional, four-component texture. These textures can be built directly from the $p(k)$, $n(k)$, and $s(i,j)$ textures.

6 Results

Two data sets were used for testing, both from the context of MHD simulations as discussed in section 3. Both data sets are curvilinear, utilize adaptive mesh refinement, and contain cell-centered samples.

Table 1: Set 1 blocks

Space dimensions	128x128x128
Total samples	819200
Level 0 Blocks	8x(32x32x32)
Level 1 Blocks	8x(32x32x32) 8x(16x32x32) 8x(20x32x32)

Table 2: Set 2 blocks

Space dimensions	512x512x512
Total samples	189440
Level 0 Blocks	1x(32x32x32)
Level 1 Blocks	1x(24x24x24)
Level 2 Blocks	1x(24x24x24)
Level 3 Blocks	1x(36x32x32)
Level 4 Blocks	1x(48x40x48)

Table 3: Data set memory requirements

	Set 1	Set 2
Data Component(L) Voxels	906048	205921
Profile(UV) Texels	4225	1089
Curve/Normal(XYZ) Texels	130	66

Set 1 has two levels with several blocks within each level. Table 1 lists the blocks and their dimensions, and figure 4 exhibits the block boundaries. This set is a good test case for the usage of several blocks within a single level, as well as shallow AMR.

Set 2 has one block per level, and five levels. Table 2 lists the blocks and their dimensions, and figure 3 exhibits the block boundaries. This set is a good test case for deep AMR data sets. Due to the curvilinear AMR representation used as well as the representation for the computational space to physical space transformation, memory requirements are very reasonable. As can be seen from figure 5.4, very little overhead is needed to represent the data, thus permitting increased scalability.

Figures 5 and 4 show volume rendering times for data sets 1 and 2 respectively. Set 1 requires more rendering time than set 2 due to the increased number of samples and blocks in set 1.

As can be seen in figures 6 and 7, the rendering time does not vary linearly with the resolution. It was found that the majority of time within the fragment programs, where the ray casting occurs, is being spent on texture fetches. This indicates that at lower resolutions, the caches on the GPU are not as effective at caching the texture data as they are at higher resolutions.

Also, as expected, as the minimum step length increases, the required computational time decreases due to the decreased number of steps. Figure 5 exhibits a range of step lengths for both of the test data sets.

The proposed compact representation of position greatly reduces the amount of memory bandwidth required between GPU caches and the external texture memory. Additionally, it reduces the number of texture fetches required to compute Jacobians for each ray step.

Some positional error results from the profile extrusion on AMR

Figure 8: The positional error(the difference between the original mesh position and the mesh point found with equation 1) is proportional to the point darkness in these images. From left to right, the images are of set 1 levels 0 to 1, set 1 level 1, set 2 levels 0 to 4, set 2 levels 3 to 4.

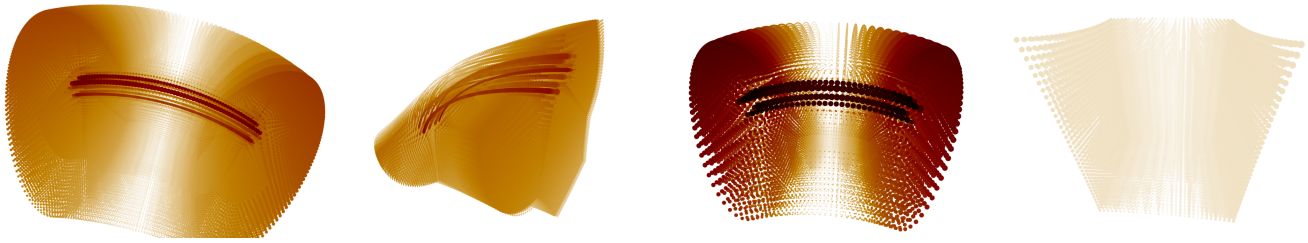


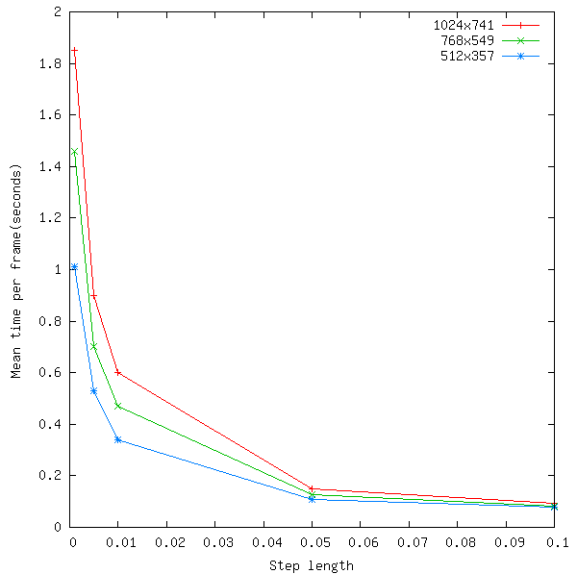
Table 4: Set 2 rendering times for different minimum step lengths and viewport resolutions.

MSL	1024x741	768x549	512x357
0.1	0.032s	0.025s	0.020s
0.05	0.049s	0.038s	0.029s
0.01	0.155s	0.112s	0.076s
0.005	0.229s	0.161s	0.106s
0.001	0.396s	0.275s	0.177s

Table 5: Set 1 rendering times for different minimum step lengths and viewport resolutions.

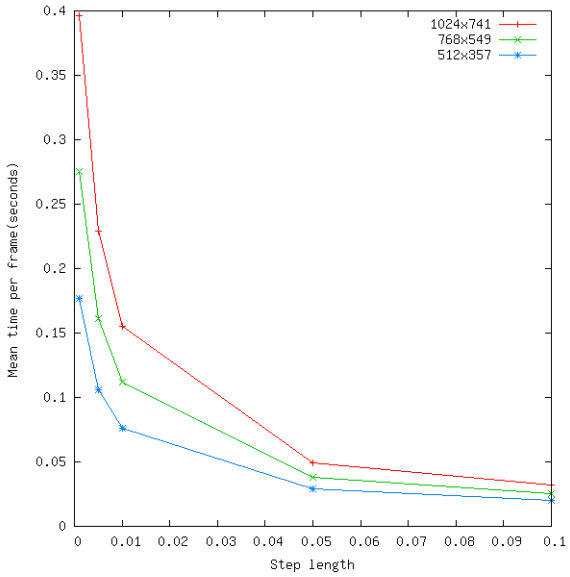
MSL	1024x741	768x549	512x357
0.1	0.095s	0.083s	0.079s
0.05	0.151s	0.127s	0.110s
0.01	0.60s	0.47s	0.338s
0.005	0.90s	0.70s	0.528s
0.001	1.85s	1.46s	1.01s

Figure 6: Data set 1 running times



data. On the test data sets, this error remained much smaller than the size of a given cell within the data. Figure 6 exhibits the positional error in the data sets.

Figure 7: Data set 2 running times



For a 1024x1024x1024 volume that fits the above constraints to be fully defined for transformations between computational space and physical space, only 4096 three-component, 2^{20} two-component, and 2^{20} four-component texels would be required – a significant improvement over a direct 3D representation which would require thousands of times more memory.

7 Conclusion

We have a presented a technique for memory-efficient and time-efficient volume rendering of curvilinear adaptive mesh refinement data within extrudable computational spaces. The volume is represented as a planar two-dimensional surface that is extruded along a profile curve. The Jacobian for points within the volume can also be easily computed using partial derivatives of these functions. This provides significant memory savings over using a a uniformly sampled volume texture to represent the transformation, in addition to reduced memory bandwidth requirements because due to more localized texture lookups. With this technique, curvilinear adaptive mesh refinement data sets with extrudable meshes can be more efficiently visualized and manipulated.

REFERENCES

[1] J. Georgii and R. Westermann. A generic and scalable pipeline for gpu tetrahedral grid rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1345–1352, 2006.

[2] L. Hong and A. Kaufman. Accelerated ray-casting for curvilinear volumes. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 247–253, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[3] L. Hong and A. E. Kaufman. Fast projection-based ray-casting algorithm for rendering curvilinear volumes. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):322–332, 1999.

[4] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[5] G. Marmitt, H. Friedrich, and P. Slusallek. Interactive Volume Rendering with Ray Tracing. In *Eurographics State of the Art Reports*, 2006.

[6] D. M. Reed, R. Yagel, A. Law, P.-W. Shin, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 55–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[7] R. Samtaney, B. van Straalen, P. Colella, and S. C. Jardin. Adaptive mesh simulations of multi-physics processes during pellet injection in tokamaks. *Journal of Physics Conference Series*, 78:2062–+, July 2007.

[8] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *SIGGRAPH Comput. Graph.*, 24(5):63–70, 1990.

[9] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 187–195, 2005.

[10] J. H. University. Visualization of time-varying curvilinear grids using a 3d warp texture yuan chen, jonathan cohen, subodh kumar.

[11] J. E. Vollrath, T. Schafhitzel, and T. Ertl. Employing Complex GPU Data Structures for the Interactive Visualization of Adaptive Mesh Refinement Data. In *Proceedings of the International Workshop on Volume Graphics '06*, 2006.

[12] J. Wilhelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri. Direct volume rendering of curvilinear volumes. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, pages 41–7, 1990.

[13] Y. Zhou and M. Garland. Interactive point-based rendering of higher-order tetrahedral data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1229–1236, 2006.