An Image-Based Modelling Approach To GPU-based Unstructured Grid Volume Rendering

Naeem Shareef,[1] Teng-Yok Lee,[1] Han-Wei Shen,[1] and Klaus Mueller[2]

[1]Department of Computer Science & Engineering, The Ohio State University
[2]Department of Computer Science, Stony Brook University

Abstract

Unstructured grid volume rendering on the GPU is an ongoing research goal. The main difficulty is how to map the irregular data efficiently to the graphics hardware. Grid cells cannot be directly processed by the pipeline since polyhedral cells are not basic graphics primitives. Thus, approaches that render directly from the grid must try to overcome inherent computational bottlenecks such as cell ordering.

We present a novel image-based approach that allows for flexible sample placement and an efficient mapping to graphics hardware that can leverage current trends in hardware performance. The unstructured grid is replaced by a semi-regular data structure we call the Pixel Ray Image (PRI), which consists of a pixelized projection plane and samples defined on sampling rays that intersect the grid. Each ray is interpreted as a linear spline of scalar values. The knot count is reduced by taking advantage of coherence in the sampling direction using spline simplification. The final representation is stored in a packed fashion onto 2D textures and uploaded once to texture memory prior to rendering. We detail how to perform efficient rendering utilizing shader programming and a slice-based rendering scheme that is similar to those used by hardware-accelerated rendering approaches for regular grids.

Categories and Subject Descriptors (according to ACM CCS): I.3.0 [Computer Graphics]: General; I.3.1 [Computing Methodologies]: Computer Graphics-Hardware Architecture; I.3.3 [Computer Graphics]: Picture/Image Generation-Viewing algorithms; I.3.3 [Computer Graphics]: Picture/Image Generation-Display algorithms; G.1.1 [Numerical Analysis]: Interpolation

1. Introduction

Todays PC graphics hardware has shown to be an ideal platform for volume rendering of regular grids. The reason is that a regular grid volume dataset naturally maps to a stack of 2D textures or a 3D texture where interpolation of the scalar field within the grid cells is performed efficiently by texture mapping hardware, a well optimized operation for some time now. On the other hand, unstructured grid volume rendering on graphics hardware has been particularly challenging due to the irregular organization of the data samples. Nevertheless, the unprecedented performance gains of graphics hardware on the PC over recent years makes it a preferred platform for many visualization tasks today, and this technology can also greatly increase the usability of unstructured grid data in key areas such as the sciences, engineering, and medicine.

The complexity of the unstructured grid poses unique challenges when trying to map rendering tasks to graphics hardware. Current approaches differ in how they utilize the available functionality in the graphics pipeline. Cell projection methods [ST90] [RKE00] [CICS05] [WMFC02] submit the cells to the pipeline one at a time and use the hardware to interpolate values across cell faces. Raycasting methods [WKME03] [WMKE04] store the grid onto textures and use shader programs to traverse the cells. Plane methods (slicing or sweeping) [YRL*96] [Wes01] [BG05] operate over groups of cells as the sweep plane traverses the data.

All direct volume rendering approaches for unstructured grids must overcome two computational bottlenecks, i.e. cell ordering and per cell processing, both of which are not easily solved or accelerated in graphics hardware. Since the set of available hardware primitives are limited to polygons only, with no direct support for polyhedra, the grid cells are usually decomposed into their cell faces. Most approaches as-

© The Eurographics Association 2006.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

sume that the grid has been tessellated into tetrahedrons in order to leverage efficient processing on a per triangle basis. The drawback is that their performance does not scale well with the tetrahedron count due to both of the aforementioned bottlenecks. Also, grids with a mix of cell types usually experience a substantial increase in their cell count when decomposed into tetrahedrons. To alleviate the problem, one solution is to apply a grid simplification algorithm in order to reduce the cell count. These methods utilize coherency found in the scalar field, but introduce approximation errors in the process. Another solution is to resample the unstructured grid into a regular grid and then use a fast hardware-accelerated approach for rendering, such as 3D texture volume rendering. This can introduce a significantly larger sample count when compared to the original data sample count as well as approximation errors. With limited texture memory continuing to be an issue, the size of the regular grid volume usually needs to be reduced.

The idea behind our approach comes from the area of image-based modelling and rendering where a large and complex scene is re-represented with view-dependent information. The simpler representation allows for fast rendering on graphics hardware. We construct a data structure that is similar to the Layered Depth Image (LDI) [SGHS98], which is used to represent polygonal scenes. The LDI is a point-based representation, where samples are collected at ray-polygon intersections on sampling rays cast into the scene from a chosen view direction. The rays are allowed to enter and exit the scene so that each sampling ray can collect all ray-polygon intersections along its path, which are then stored in depth order in the final data structure. The scene may then be rendered from other view directions much faster than rendering directly from the original scene. In addition, the semi-regular organization of the data samples allows for efficient storage and retrieval of the data.

We present a data structure called the Pixel Ray Image (PRI). Parallel sampling rays are projected into the unstructured grid volume and samples are located and collected into the final data structure. In addition to the location of a sample, a reconstructed scalar value is computed and stored as well. The list of samples on a sampling ray define a 1D scalar function, represented as a linear spline. The flexibility of placing samples allows for easy resampling of a grid with a mix of cell types. In addition, far fewer samples are needed than with a regular resampling of the grid. Scalar coherency found in many scalar fields can be used to further reduce the sample count. The PRI is easily rendered with a slicing approach using only 2D textures.

2. Related Work

One of the first algorithms to render unstructured grid data on traditional graphics hardware is the well known Projected Tetrahedra (PT) approach [ST90]. For each view, the tetrahedrons are sorted and then projected to the screen one at a time. The first version of the algorithm assumes a preclassification of the volume data with a color stored at each grid vertex. The rasterization hardware interpolates these colors across the triangle faces. Roettger [RKE00] was able to improve on this with new hardware functionality by interpolating scalars and using pre-integrated transfer functions. To address the bottleneck due to projection and tetrahedron classification, Wylie et al [WMFC02] uses the vertex shader to perform these operations. Weiler et al [WKME03] also move operations to the shader program and are able to make use of display lists and vertex arrays. Work by Callahan et al [CICS05] addresses the bottleneck due to depth sorting by computing a partial ordering of the tetrahedrons on the CPU and then use the GPU to help finish the sort. Raycasting approaches [WKME03] [WMKE04] store the grid in texture memory and perform cell traversal for each ray in the shader program. Limited texture memory only allows for the rendering of small grids since geometry as well as cell connectivity information consume a good amount of storage. Slice-based methods [YRL*96] [Wes01] [BG05] sweep the grid with a plane and process the cells that intersect the plane. Extra data structures are usually needed to hold information about which cells intersect the plane in the current iteration [YRL*96] [BG05]. Grid simplification [CCM*00] [UBF*05] reduces the number of cells in the grid using edge collapse operations or other cell merging techniques based upon coherency in the scalar field. Cignoni et al. [CFM*04] compute a level-of-detail (LOD) representation over a hierarchy of simplified grids in order to render large grids. Of course, all of these methods tradeoff approximation error with faster rendering.

An alternative approach to render the unstructured grid on graphics hardware is to resample the grid into a regular grid. The sample spacing is determined by the size of the smallest cell in order to sufficiently resample the dataset. This can cause a large increase in the number of samples. Westermann [Wes01] stores the samples computed from slicing the unstructured grid and then renders the volume as a regular grid. Leven et al [LCCK02] use an LOD approach by constructing a hierarchy of simplified regular grids and use 3D texture volume rendering on a chosen level.

Our approach stems from image-based and view-dependent techniques popular in traditional polygonal scene rendering. For example, the approaches by [OB99] [PZvBG00] use three LDI oriented to be perpendicular to each other to ensure a sufficient sampling of the polygonal scene. In the context of volume rendering, Srivastava [SCM03] pre-compute and store regularly spaced samples computed from a software raycasting of a regular grid volume, where each sample holds its location on the ray and a reconstructed scalar value. Their data structure allows for fast transfer function changes at a fixed viewpoint on the CPU if the sample count is small. To reduce the sample count for storage purposes, consecutive runs of samples that approximate a line in the scalar domain are collected into lin-

© The Eurographics Association 2006.

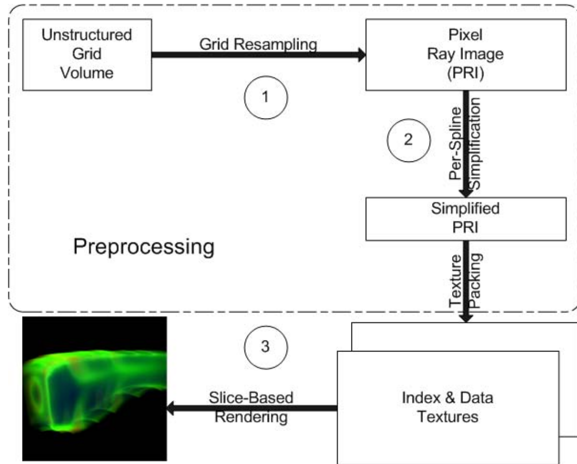


Figure 1: *Our three step PRI rendering pipeline. In the preprocessing phase, the volume grid is resampled onto sampling rays (step 1), which are reduced in size using a linear approximation (step 2). The resulting PRI is stored onto textures and rendered from arbitrary viewpoints on the GPU (step 3).*

ear segments using a spline construction approach based on a greedy algorithm that may insert new samples into the final linear spline.

3. The Pixel Ray Image (PRI)

The PRI is a data structure that consists of two components. The first is a 2D image plane, called a *sampling plane*, that has finite dimensions and is defined in space using camera parameters: 1) a position outside the unstructured grid, 2) an orientation with an up vector, and 3) a view direction that points towards the grid. A 2D coordinate system is defined on the plane where the origin is located at the lower left corner. The plane is pixelized with a regular grid of pixel centers that are located at integer offsets from the origin.

The second component is a collection of 1D scalar functions, one function per sampling plane pixel. Lets assume the unstructured grid is convex. If a pixel ray, called a *sampling ray*, intersects the grid, it will enter and exit the grid at two boundary faces. The intersection of a ray with the volume defines a 1D scalar function, $\{\forall(t_{enter} \leq t \leq t_{exit})|((f(t) = s)\}$, where t_{enter} is the entry location of the ray into the grid, t_{exit} is the exit location, t is the distance of a point on the ray from the sampling plane, and s is the reconstructed scalar value at location t. The function is represented as a linear spline with two knots located at the t_{enter} and t_{exit} locations plus zero or more knots between them. If the scalar field is assumed to be linear within the grid cells, then knots may be located at the intersections between the sampling ray and cell faces. Otherwise, samples can be located uniformly along the ray, as is typically done in raycasting, and the scalar field is reconstructed within the cell using an appropriate reconstruction filter. Thus, each knot is defined by the value pair (s,t), where s is a scalar value and t is the knot‘s distance to the sampling plane. A segment of a spline is defined by two consecutive knots (s_k,t_k) and (s_{k+1},t_{k+1}), where $t_k < t_{k+1}$, and the scalar field is assumed to change linearly over the segment. All knots on a particular sampling ray are maintained in a single list. For a non-convex unstructured grid, a sampling ray may enter and exit the grid more than once. In order to define the regions outside the volume along the sampling ray, we introduce an additional knot at those knots located at the boundary of the grid. If a knot, say (s_b,t_b), lies on the boundary of the grid then an additional knot is inserted into the knot list with knot value $(0,t_b)$. Thus, portions of the ray that lie outside the grid are bounded by consecutive pairs of knots with a multiplicity of two.

Dataset	Size	time	$\#k/spl$
Blunt Fin	187395	29.20	80
Combustor	215040	86.04	63
Liquid Oxygen Post	513375	32.84	85
SPX	12936	4.39	27

Table 1: *Test datasets with their sizes in number of tetradrons and initial PRI construction time (seconds) using depth peeling. The PRIs all have a pixel resolution of 256 $\times$ 256. The last column reports the average number of knots per spline.*

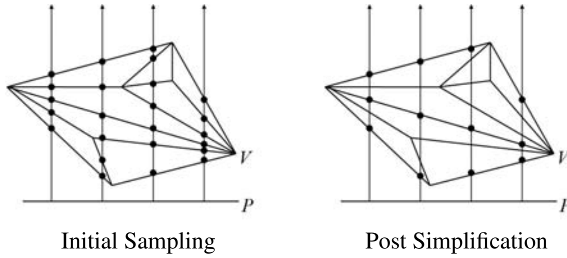
The PRI replaces the original unstructured grid volume. Our rendering pipeline consists of three steps, shown in Fig. 1, where the first two steps are done once as a preprocessing step to construct the PRI. The first step resamples the grid onto the sampling rays. In order to obtain a sufficient resampling of the grid, the pixel resolution of the sampling plane should be chosen so that multiple rays pierce every cell face. In the second step, the knot count is reduced on a per spline basis using a linear spline simplification algorithm. In the third step the entire collection of splines are packed onto a 2D texture and rendered on the GPU. We describe each step in the following sections.

4. Spline Construction

There are many choices for the location and sampling direction of the PRI about the volume dataset. We restrict our space of candidate view choices to those whose sampling directions point towards the center of the grid. We define a tight axis-aligned bounding box around the volume and choose one of the box faces as the sampling plane and use the associated axial direction as the sampling direction.

Since we assume the scalar field is linear within grid cells, we locate knots on each sampling ray at the ray-cell face intersections. This process can be accomplished with a soft-

© The Eurographics Association 2006.



Initial Sampling | Post Simplification

Figure 2: *Let V be an unstructured grid and P be the sampling plane of the PRI. The PRI is constructed by first defining samples on the intersections between a ray and the grid cell faces (left). Spline simplification is used to reduce the knot count by identifying spans of nearly linear scalar values (right).*

ware raycaster that projects a sampling ray from each pixel. When the ray intersects any grid cell face along its path, the raycaster computes the distance, t, from the originating pixel to the intersection location and reconstructs the scalar field, s, across the cell face. The resulting value pair, (s,t) is inserted into the knot list associated with the spline. Alternatively, we use depth peeling to quickly compute the knot values on graphics hardware. It is a multi-pass approach that renders layers of cell faces in depth sorted order. Table 1 shows spline construction times over a variety of volume datasets and the average number of splines per PRI computed for each dataset.

5. Data Reduction

The number of samples defined in the initial construction step will be greater than the number of samples in the original unstructured grid but less than the number of samples that would be defined in a resampling of the unstructured grid onto a regular grid. The linear spline representation provides flexibility in the placement of the knots that define a 1D scalar function. Many volume datasets contain scalar fields with regions that are nearly homogeneous and that change smoothly. Since these smooth regions may span across groups of grid cells, many of the sampled knots may be redundant. We identify scalar coherency on each linear spline as a run of consecutive knots whose scalar values are nearly collinear.

In order to reduce the total knot count after the initial spline construction step, spline simplification is used to identify spans of nearly collinear runs of scalar values within a defined error threshold in order to remove the redundant internal samples and preserve those at the endpoints of the run, illustrated in Fig. 2. We use a knot removal algorithm with the desirable property that the resulting list of knots is a subset of the original list. It is an iterative approach that is based on a 1D version of the mesh simplification algorithm for terrain data by Duchaineau [DWS*97]. At each iteration the algorithm considers all consecutive and non-overlapping pairs of linear spline segments. For each segment pair, it will merge the two segments into a larger single segment by removing the middle knot if each knot from the original spline that lie within the span of the pair of segments is within a pre-chosen vertical distance, i.e. the scalar domain, away from the candidate segment. The simplified spline that results at the end a particular iteration contains the same knots in the previous iteration with at most half of them removed by the merges. The algorithm stops when none of the spline segment pairs can be merged anymore.

Since the PRI is a resampling of the unstructured grid, call it V, we measure the approximation error of the PRI, call it P. Using an approach similar to [CCM*00], the sampling error is computed by evaluating vertices from grid V in PRI P. We compute the peak signal to noise ratio (PSNR) using the following equation for the mean square error (MSE):

$$MSE(V,P) = \frac{\sum_{\forall t \in V} \varepsilon_P(t)^2 |t|}{\sum_{\forall t \in V} |t|}$$

where t is a tetrahedron in V, the value $|t|$ is the volume of tetrahedron t, and

$$\varepsilon_P(t) = max_{\forall v \in t} |S(v) - S_P(v)|.$$

The error function $\varepsilon_P(t)$, computed over all four vertices of tetrahedron t, is the maximal residual error on tetrahedron t. The value $S(v)$ is the scalar value at v in the original grid and value $S_P(v)$ is reconstructed from PRI P by trilinear interpolation, as described in Section 6.2. Table 2 shows the average numbers of knots per spline after the simplificaiton algorithm is run on four error threshold values, where scalar values are in the range [0 ... 1]. Note that an error threshold of 0.0 results in the removal of superfluous knots. These may represent samples that lie in a constant or linearly changing region in the scalar field.

Fig. 3 shows plots of the PSNR under six error thresholds, i.e. .05, .04, .03, .02, .01, and 0.0, where each result is plotted as an icon in the figure from left to right, respectively. The horizontal axis represents the fraction of the average number of knots per spline in the PRI between pre and post spline simplification. The graphs illustrate a sharp rise in the error cost as the error threshold is relaxed, i.e. from right to left. As a result, a suitable tradeoff between error cost and data size may be easily determined.

6. Rendering

In this section, we present how the PRI is stored onto textures and then present a rendering approach using slice-based rendering and trilinear interpolation to compute the volume integral using post-classification. A trilinearly interpolated sample can be computed in a straightforward manner from the PRI in a shader program, yet this implementation is expensive on current GPU technology due to costly searches

© The Eurographics Association 2006.

Dataset / Threshold	0.00	0.01	0.02	0.03
Blunt Fin	49	9	6	6
Combustor	53	15	11	10
Liquid Oxygen Post	55	9	7	7
SPX	20	7	5	4

Table 2: *The average number of knots per spline over different simplification thresholds.*

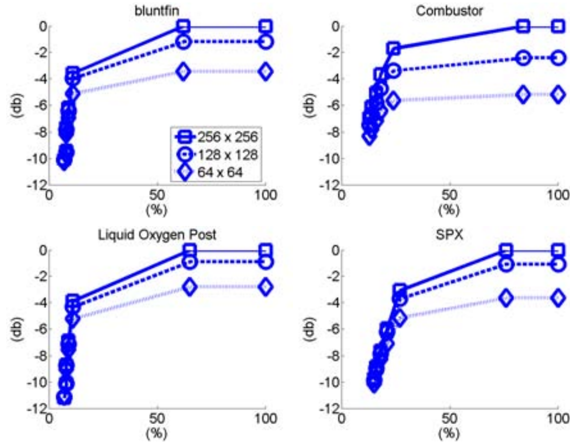


Figure 3: *The plots show PSNR comparisons between the unstructured grid and the PRI over different error thresholds and different PRI sampling resolutions.*

and interpolation. We present two acceleration schemes to overcome these bottlenecks.

6.1. Texture Formats

The PRI is represented in texture form as two 2D textures: a 2D array of index values called the *index texture* that references to another 2D texture called the *knots texture*, which holds the entire collection of per-pixel splines. Using a pre-determined traversal of the pixels in the sampling plane of the PRI, e.g. row-order or column-order, the per-pixel splines are laid out onto the knots texture in a packed fashion.

Each texel in the index texture holds an index into the knots texture that references the first knot in the linear spline associated with the corresponding pixel in the sampling plane. Thus, the index texture has the same resolution as the PRI pixel plane. Each texel plane maps to the following information: (r,g,b,a)=>$(s',t',\#knots,depth)$, where the first two channels hold the two dimensional address of the first spline knot, channel b holds the number of knots in the spline, and the alpha channel holds the normalized distance of the last knot of the spline from the sampling plane. Texels with a zero value for channel b correspond to sampling rays that do not intersect the grid. Thus, a pixel fragment on a slicing plane can be quickly identified to lie outside the unstructured grid with two checks: 1) its z location overlaps a spline with zero knots, i.e. the sampling ray does not enter the volume grid, or 2) its distance to the sampling plane is less than the distance to the first knot or greater than the distance to the last knot of the spline. In our implementation, the knots texture is stored as two separate textures with float format to ensure high precision, where one holds the scalar values and the other holds the distance values of the knots.

6.2. A Basic Slice-based Rendering Algorithm

We use a slice-based approach to compute renderings at arbitrary views. A stack of parallel quads are textured with scalar information and, after classification, the fragments are composited in depth order, *front-to-back* or *back-to-front*, to the final image. In an axis-aligned scheme, the stack of slices are positioned to be perpendicular to the sampling direction of the PRI. In a view-aligned scheme, the stack is positioned to be always parallel to the image plane. In either case, a scalar value must be reconstructed from the PRI at each pixel fragment. Take a pixel fragment whose location is (x,y,z) in the coordinate space of the PRI construction. Three steps are required to reconstruct a trilinearly interpolated scalar value at this location:

Step 1: Parallel project the position of the pixel fragment to the PRI‘s sampling plane by dropping the z coordinate and identify the four pixels on the sampling plane that surround the projection point (x,y), i.e. the projection point‘s four nearest neighbors, as shown in Fig. 4, right.

Step 2: Search each of the four linear splines associated with the neighborhood of pixels in Step 1 to find the spline segment that overlaps the pixel fragment‘s z location.

Step 3: Trilinear interpolation is performed by four linear interpolations in the z direction using the four spline segments found in Step 2, followed by bilinear interpolation in the x and y directions.

In an axis-aligned slicing scenario, at least three PRI‘s are needed to allow for rendering from arbitrary views. Three PRI‘s may be constructed such that their sampling directions are oriented to be perpendicular to each other. A single PRI is chosen, the one that is most parallel with the user’s view direction, to compute a rendering. On the other hand, in a view-aligned slicing scenario, at least one PRI is required to be able to render from arbitrary views.

A basic rendering algorithm to perform either axis- or view-aligned slicing is to perform reconstruction, classification, and alpha blending in the shader program. To perform Step 1 in an axis-aligned slicing scenario, the index texture is texture mapped to each rendering slice. The texture coordinates at the four corners of each slice are set to $(0,0)$, $(0,h)$, (w,h), and $(w,0)$, where w and h are the pixel dimensions of the index texture. This avoids the need to inverse map a pixel

© The Eurographics Association 2006.

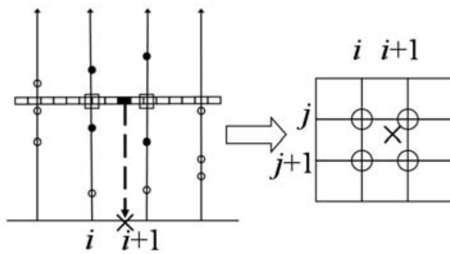


Figure 4: *A pixel fragment at (x, y, z), shown in black on the left, is reconstructed from the PRI by first projecting the point to the sampling plane. The projection point will map to four neighboring pixels, whose associated splines are used to compute a trilinearly interpolated scalar value.*

fragment from eye space to object space. In a view-aligned slicing scheme, the inverse mapping is done with OpenGL's *TexGen* operation, similar to shadow mapping. The shader program retrieves the four nearest texels in the index texture and accesses each associated linear spline in the knots texture via dependent texturing. Since the knots of each spline are sorted in depth, the search for the spline segment in a spline whose span includes the pixel fragment's z location is performed with a binary search. Finally, the shader program computes the trilinear interpolation in Step 3. We ran this algorithm on a current GPU platform and found that the search time contributed to a considerable computational bottleneck. With current performance trends in GPU technology, this approach could be viable on future graphics cards. In the following section, we present an approach to overcome this cost that requires at least three PRIs whether using either of the axis- or view-aligned slicing paradigms. Thus, the main advantage of this straightforward approach is that only a single PRI is needed at a minimum when using view-aligned slicing.

6.3. Accelerating Search With An Index Texture Cache

The basic algorithm requires four spline searches per pixel fragment to compute a trilinearly interpolated scalar value. We assume a *front-to-back* traversal of the slice stack in our discussion, but the method we present here also applies to the *back-to-front* case as well. To address this computational bottleneck we use an incremental traversal of all the splines in the PRI by progressively advancing the rendering slices. The spline segments that are required to reconstruct the pixel fragments in the current slice will either be the same spline segments used to reconstruct pixel fragments in the previous slice or spline segments that are at a further distance away from these segments. Thus, the spline segments are processed in depth order. If the next slice requires a spline segment not used in the previous slice, then a linear search will quickly find the required segment by starting from the segment processed in the previous slice. If locations of spline segments used in a previous slice are cached, then determining spline segments needed for the next slice will not require a search from scratch.

We implement this cache with a 2D texture called the *cache index texture*, which has the same resolution as the PRI's pixelized sampling plane. The cache index texture is initialized with the contents of the index texture in order to cache the first knots of each spline. Each slice in the rendering stack is then rendered with a two-pass approach. In the first pass, the cache index texture is updated with the knot indexes that are correct for the current slice. The camera is set to coincide with the PRI and the cache index texture is specified as a rendering target. The cache index texture is mapped to the slice via projective texture mapping using orthogonal projection and in the sampling direction of the PRI. Linear search is applied at each texel to find the appropriate spline segment. In order to accomplish this, two textures are used to perform *ping pong rendering*. In the second pass, the updated cache index texture is used to render the slice from the user's view using dependent texturing as described before.

6.4. Accelerating Search For Large Screen Resolutions

When the user's screen resolution exceeds the PRI's sampling plane resolution, a block of pixel fragments on a rendering slice will fall within the same neighborhood of four splines when projected to the PRI's sampling plane. If the reconstructed scalar values at the four neighboring splines are available, then the scalar values for all the projected fragments within this neighborhood can be computed more efficiently via bilinear interpolation. The magnification filter available in the standard graphics pipeline provides this operation. To accomplish this we introduce an additional texture, called the *interpolation texture*, which stores a reconstructed scalar value per spline after applying just linear interpolation on a segment in the z direction. This is easily added to the two-pass algorithm presented in the previous section. In the first pass, after the cache index texture is computed for the current slice, it is rendered to the interpolation texture, defined as a rendering target. The z distance of the slice fragment is used to access the corresponding spline segment indexed by the cache index texture and also to compute linear interpolation across the segment. This value is rendered to the interpolation texture. In the second pass, the interpolation texture is texture mapped to the rendering slice and rendered to the screen. In the axis-aligned slicing case, this operation computes an accurate reconstruction of the slice from the PRI because the fragments of a slice lie on a plane that is always perpendicular to the sampling rays, i.e. all the fragments have the same z value, similar to the shear warp approach [LL94]. In the view-aligned slicing case, there is the possibility that fragments may be reconstructed from incorrect splines, as shown in Fig. 5. This

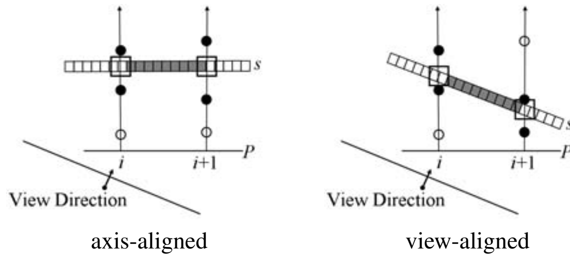


Figure 5: *Trilinear interpolation may be computed faster for pixel fragments (gray) on a rendering slice that fall between the same neighboring splines, e.g. pixels i and $i+1$ (in 2D), if the reconstructed scalar values in the z direction are cached and then re-used in a bilinear interpolation in the x and y direction. The figure on the left illustrates that this is correct for axis-aligned slicing. The figure on the right illustrates a case where inaccurate interpolation values may be computed in view-aligned slicing if the slice orientation causes the slice to coincide with an incorrect spline segment.*

introduces a small approximation error in exchange for a noticeable speedup.

7. Results

We show results of our rendering algorithm using a PC with an Intel P4 1.99 GHz CPU, 768 MB of system memory, and a nVidia GeForce 6600 GT graphics card. For all results shown, the PRI sampling resolution is 128 × 128 with no spline simplification. The renderings are computed using view-aligned slicing with a stack of 128 slices.

Fig. 6 shows rendering times of our algorithms on the four test datasets over increasing screen resolutions. The top left graph shows rendering times using the basic algorithm, while the remaining three graphs show rendering times using the interpolation texture and then with and without acceleration due to the cache index texture. In either case, a considerable speedup is achieved compared to the basic algorithm. Fig. 7 shows the rendering times for PRIs with different sampling resolutions. The benefit of the acceleration method of Sec. 6.4 decreases when the sampling resolution approaches the screen resolution.

We show renderings of our test datasets in Figs. 8, 9 and 10. To illustrate the importance of correct scalar interpolation, we render the *Blunt Fin* dataset in Fig. 8, using nearest neighbor interpolation (left) where only the closest spline is used to renconstruct at a pixel fragment. The basic and the hardware-accelerated algorithms drastically lower aliasing artifacts due to trilinear interpolation. Fig. 9 illustrates the difference in image quality over different sampling resolutions in the PRI. The smooth transition in image quality suggests that the PRI may be useful in the context of a LOD representation. To illustrate the effectiveness of our empty

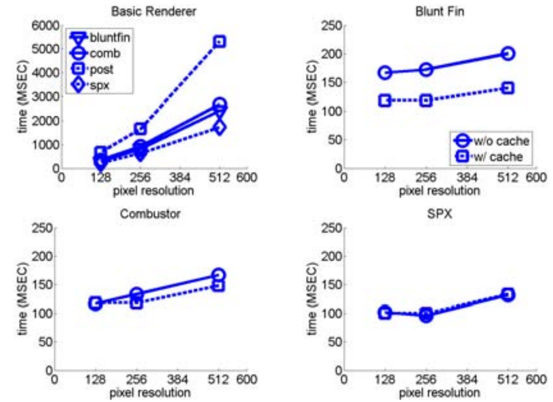


Figure 6: *The top left graph shows rendering times using the basic algorithm. The other three graphs show speedups using our two acceleration methods.*

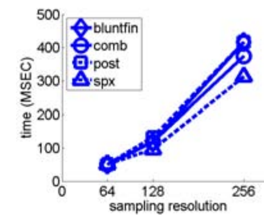


Figure 7: *The effect of the acceleration method of section 6.4 with a screen resolution of 512 × 512 on PRI sampling resolutions of 64 × 64, 128 × 128, and 256 × 256.*

region identification scheme, Fig. 10 shows the *SPX* dataset, which contains a hole in the grid, rendered without the extra knots to indicated empty regions (left) and with the knots (right). Note that the renderings of the *Combustor* and *SPX* datasets are rendered at views that are off the sampling direction.

8. Conclusion

We presented a novel data structure to resample and replace the cumbersome unstructured grid volume. The semi-regular nature of the data structure allows for flexible sample placement and, similar to grid simplification, a reduction in the sample count according to coherency in the scalar field. The data structure is easily stored onto 2D textures and rendered using sliced-based volume rendering. The results presented here illustrate the advantages of this approach in terms of a compact data size and rendering speed.

In future work, we would like to address a number of issues. We used a straightforward way to determine the sampling direction for the PRI. Other sampling directions may be better candidates in terms of sampling accuracy, data size, and scalar coherency. We will explore ways to identify op-

© The Eurographics Association 2006.

timal sampling direction(s) for a dataset. A multi-resolution scheme may be introduced to the data structure to provide a LOD to tradeoff image quality versus rendering speed. We are also interested in how to leverage scalar coherency across sampling rays. We are exploring the use of function quantization to represent the splines in a codebook of functions. The data structure also lends itself to a common framework in which time-varying grids whose structure may change across time steps can be re-represented and rendered efficiently under a single paradigm.

9. Acknowledgement

This work was supported by NSF CAREER grant ACI-0093157, NSF-ITR grant ACI-0325934, DOE Early Career Principal Investigator Award DE-FG02-03ER25572, NSF Career Award CCF-0346883 and NSF RI CNS-0403342. The authors would like to thank the reviewers for their helpful comments.

References

[BG05] BENOLKEN P., GRAF H.: Direct volume rendering of unstructured grids in a pc based vr environment. In *The Journal of WSCG* (2005), vol. 13, pp. 25–32.

[CCM*00] CIGNONI P., COSTANZA D., MONTANI C., ROCCHINI C., SCOPIGNO R.: Simplification of tetrahedral meshes with accurate error evaluation. In *IEEE Visualization '00* (2000).

[CFM*04] CIGNONI P., FLORIANI L. D., MAGILLO P., PUPPO E., SCOPIGNO R.: Selective refinement queries for volume rendering of unstructured tetrahedral meshes. In *IEEE Transactions on Visualization and Computer Graphics* (2004), vol. 10, pp. 29–45.

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L. D., SILVA C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. In *IEEE Transactions on Visualization and Computer Graphics* (2005), vol. 11, pp. 285–295.

[DWS*97] DUCHAINEAU M., WOLINSKY M., SIGETI D., ALDRICH C., MINEEV M.: Roaming terrain: Real-time optimally adapting mesh. In *IEEE Visualization '97* (1997).

[LCCK02] LEVEN J., CORSO J., COHEN J., KUMAR S.: Interactive visualization of unstructured grids using hierarchical 3d textures. In *Proc. of Symp. on Volume Visualization '02* (2002), pp. 37–44.

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In *ACM SIGGRAPH '94* (1994).

[OB99] OLIVEIRA M., BISHOP G.: Image-based objects. In *Proc. of Symp. on Interactive 3D Graphics* (1999), pp. 191–198.

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: Surface elements as rendering primitives. In *ACM SIGGRAPH '00* (2000).

[RKE00] ROETTGER S., KRAUS M., ERTL T.: Hardware-accelerated volume and isosurface rendering based on cell-projection. In *IEEE Visualization '00* (2000).

[SCM03] SRIVASTAVA V., CHEBROLU U., MUELLER K.: Intractive transfer function modification for volume rendering using compressed sample runs. In *Computer Graphics International 2003* (2003), pp. 8–13.

[SGHS98] SHADE J., GORTLER S. J., HE L.-W., SZELISKI R.: Layered depth images. In *ACM SIGGRAPH '98* (1998).

[ST90] SHIRLEY P., TUCHMAN A.: A polygonal approximation to direct scalar volume rendering. In *Proc. of San Diego Workshop on Volume Visualization* (1990), pp. 63–70.

[UBF*05] UESU D., BAVOIL L., FLEISHMAN S., SHEPHERD J., SILVA C. T.: Simplification of unstructured tetrahedral meshes by point- sampling. In *Proc. of International Workshop on Volume Graphics 2005* (2005), pp. 157–165.

[Wes01] WESTERMANN R.: The rendering of unstructured grids revisited. In *Proc. of Symp. on Volume Visualization '01* (2001).

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-based ray casting for tetrahedral meshes. In *IEEE Visualization '03* (2003).

[WMFC02] WYLIE B., MORELAND K., FISK L. A., CROSSNO P.: Tetrahedral projection using vertex shaders. In *Proc. of IEEE Volume Visualization and Graphics Symposium 2002* (2002), pp. 7–12.

[WMKE04] WEILER M., MALLON P. N., KRAUS M., ERTL T.: Texture-encoded tetrahedral strips. In *Proc. of Symp. on Volume Visualization '04* (2004), pp. 71–78.

[YRL*96] YAGEL R., REED D. M., LAW A., SHIH P.-W., SHAREEF N.: Hardware-assisted volume rendering of unstructured grids by incremental slicing. In *Proc. of Symp. on Volume Visualization 1996* (1996).

© The Eurographics Association 2006.