Case Study: Interactive Rendering of Adaptive Mesh Refinement Data

Sanghun Park[*]
CCV TICAM
University of Texas at Austin

Chandrajit L. Bajaj[†]
CCV TICAM
Department of Computer Sciences
University of Texas at Austin

Vinay Siddavanahalli[‡]
CCV TICAM
Department of Computer Sciences
University of Texas at Austin

Abstract

Adaptive mesh refinement (AMR) is a popular computational simulation technique used in various scientific and engineering fields. Although AMR data is organized in a hierarchical multi-resolution data structure, the traditional volume visualization algorithms such as ray-casting and splatting cannot handle the form without converting it to a sophisticated data structure. In this paper, we present a hierarchical multi-resolution splatting technique using k-d trees and octrees for AMR data that is suitable for implementation on the latest consumer PC graphics hardware. We describe a graphical user interface to set transfer function and viewing / rendering parameters interactively. Experimental results obtained on a general purpose PC equipped with NVIDIA GeForce card are presented to demonstrate that the technique can interactively render AMR data (over 20 frames per second). Our scheme can easily be applied to parallel rendering of time-varying AMR data.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Interactive Techniques, Graphics Data Structures and Data Types; I.3.8 [Computer Graphics]: Applications

Keywords: AMR, K-d trees, Octree, Hierarchical splatting, Texture mapping

1 INTRODUCTION

Adaptive mesh refinement (AMR) is a computational technique for improving the efficiency of numerical simulations of systems of partial differential equations. After Berger and Oliger [2] developed AMR in 1980s to simulate gas dynamics, it has become a popular technique in computational physics and various engineering fields. The basic idea of AMR is to refine, both in space and in time, regions of the computational domain where high resolution is needed to resolve developing features, while leaving the less interesting parts of the domain at lower resolutions. AMR techniques have been shown to be very successful in reducing the computational and storage requirements for solving many partial differential equations and used in various engineering applications where there are regions of greater interest such as global atmospheric modeling and numerical cosmology. For example, Bryan [3] shows how a hybrid approach of AMR can be applied to cosmological research.

Although AMR data has a hierarchical multi-resolution structure, it is impossible for traditional visualization techniques developed for simple mesh data to handle AMR data without any modification. Relatively few results have been presented on visualization of AMR data. Norman et al. [6] presented problems and solutions in storing, handling, visualizing, virtually navigating, and remote-serving data produced by large-scale AMR simulations. Weber et al. [8] introduce crack-free isosurface extraction methods from AMR data. They also present a hardware-accelerated rendering interface for previewing and cell-projection based progressive refinement rendering scheme in [7]. In another paper [9], they render AMR data using the progressive cell-projection approach and level-dependent transfer function. Even though their method can produce high quality images, it takes about 23∼115 seconds to render one image from an AMR data with a $80 \times 32 \times 32$ root-grid resolution and a three-level hierarchy.

In this paper, we describe a hierarchical multi-resolution splatting of AMR data. Our main contributions are the design and implementation of

- k-d tree and octree data structures for hierarchical storing and rendering of AMR data;
- a hardware accelerated splatting algorithm for interactive rendering of AMR data;
- a graphical user interface for determining transfer function and viewing / rendering parameters interactively;

The rest of this paper is organized as follows. Section 2 presents the implementation details of our scheme. Section 3 contains the results of our techniques regarding rendering time, image quality, and graphical user interface. Finally this paper is concluded in Section 4.

2 IMPLEMENTATION DETAILS

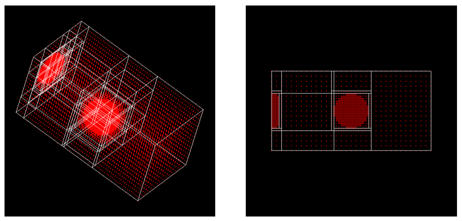
2.1 K-d Trees

A k-dimensional (k-d) tree is a data structure that splits multidimensional spaces. It is used in computer sciences during orthogonal range searching. It allows us to find the set of voxels that fall within a given block or brick in a space. Given a k-d tree of voxels, it is possible to find the resulting voxels in $O(\sqrt{n}+k)$ where n is the number of voxels and k is the number of voxels in the result. An AMR simulation algorithm generates a grid hierarchy data structure (a tree of arbitrary structure and depth) and every node and leaf of the tree is associated with a 3D grid. Since these have various shapes, sizes, and spatial resolutions, we need to convert this form to a sophisticated data structure.

AMR data is represented by $\bigcup\{f_{t,l,v}(i,j,k)\}$ where t is the timestep of time-varying AMR data, l is the refinement level and v is the index of function values. To convert the data to a k-d tree structure, the first step is to determine the minimum bounding boxes surrounding each group in the AMR data space. The voxels included in a group are connected in spatial resolution of the current level and each group may include several levels of AMR data. In the next stage, our scheme splits the bounding boxes into a set of bricks using a modified k-d tree algorithm. The generated bricks have pointers to actual function value sets in each level.

[*]e-mail:hun@ticam.utexas.edu
[†]e-mail:bajaj@cs.utexas.edu
[‡]e-mail:skvinay@cs.utexas.edu

As we mentioned, a k-d tree is not only useful for rendering, but also for storing AMR data hierarchically. In fact, the raw form of AMR data is a list of records that consist of a voxel index (i, j, k), some function values $f_{t,l,v}$, and a level l. Replacing the raw AMR data format with a k-d tree is very efficient in that it contains hierachical, multiresolution structures and we don't have to store the voxel indices any more. Figure 1 shows the partition result of test AMR data using a k-d tree. To exploit spatial coherence, our scheme constructs an octree structure for relatively large bricks.



(a) 3D view (b) 2D view

Figure 1: Different views of splitting AMR data space into a set of bricks in k-d tree structure.

2.2 Hierarchical Splatting

Splatting is an object space direct volume rendering algorithm that generates high quality images [10]. A voxel's contribution is mapped directly onto the image plane, eliminating the need for interpolation. Since only interesting voxels (weighted by the discrete voxel values) are required to be represented by a 3D kernel, it is possible to generate the final images at interactive speed. Splatting has been used in the past to handle hierarchical error and higher dimension rendering [5, 1]. This techniques allow us to render different level data sets using footprints of various sizes. By projecting the footprint to a polygon, we can exploit OpenGL 2D texture mapping hardware. Crawfis et al. propose to render each splat using texture mapping hardware [4]. This technique alleviates the CPU from the computational complexity incurred in resampling the footprint tables and compositing each splat into a frame buffer.

In our algorithm, the rendering of a selected range of isovalues, together with a transfer function is performed through splatting. While the k-d tree data structure provides an accelerated search algorithm for bricks, we further optimize our algorithm through the use of octrees at each node of the k-d tree. Each brick is represented as an octree of voxels. Each voxel is then projected by transforming its position from world coordinates to screen coordinates.

In each brick and octree, we store an isovalue code to ensure that only relevant volumes are considered for the search algorithm. This isovalue code is implemented as a 32-bit number, each bit representing the presence or absence of a range of isovalues. As a preprocessing step, we compute and store the binary code. It is computed for each level of the octree. We traverse through all voxels contained in a leaf and determine the bits in the code to be turned on to indicate the presence of at least one voxel in that range. If there are n bits for the code, and we have a range of r for the voxels in the whole volume, then each bit covers a range of $\frac{r}{n}$. Once we have obtained the code for each leaf, we recursively obtain the code for a node as the boolean *OR* of the codes of its children. The code of an octree's root is assigned to the brick containing the octree. This procedure is used for each function in the vector field. We do not perform weighting for this code. For example, we do not assign fewer bits to non-interesting regions, hence assuming the entire range to be of equal interest to user. An *OR* operator is used to build a search code when the user changes the range of important isovalues. A simple *AND* operator is sufficient to eliminate those sub-volumes whose codes do not fall under the currently selected range of isovalues. This is implemented at both the brick level and the octree level. Within an octree, each child that is not *NULL* contains such a code to help its traversal. We found this method to be considerably faster than storing min-max values. A second useful feature of the octree is the fast and natural ordering of voxels that it provides to obtain high quality images quickly. We sort bricks for each view from the k-d tree.

We have implemented a color table to be used as lookup for the transfer function. An interactive selection of opacity values for different isovalues is used. This is also extended to handle vector valued data.

```
1  Load AMR data
2  Create k-d tree and octree data structure
3  Set current transfer function and viewing / render-
ing parameters
4  Determine brick list BL according to viewing direction
5  For (Bi in BL) {
6      Splatting(Bi, lod)
7      Composite the current partial image
8  }
9  Display final image
```

Figure 2: K-d tree based splatting algorithm

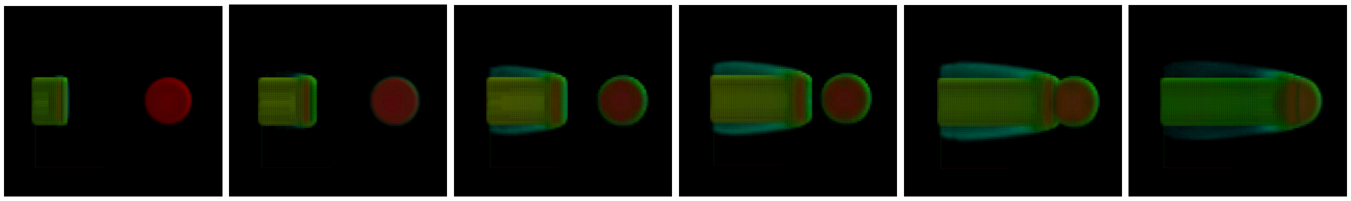
2.3 Algorithm

Figure 2 shows our rendering algorithm for splatting of AMR data. Since creating k-d trees and octrees of the given AMR data (line 2) can be done at the preprocessing stage, they don't affect the actual run-time rendering speed. Once a k-d tree is created, the brick list *BL* can be efficiently determined by traversing the k-d tree depending on the viewing direction (line 4). In this step, the bricks that don't contain any interesting voxels are not included in the brick list. A single bitwise *AND* operation is needed for checking whether the brick will be rendered or not. Then, splatting is applied to each brick B_i in the sorted list of relevant bricks and the generated partial images are composited to form a final image (lines 6 and 7). Since the nodes in the brick list have pointers to function values corresponding to several levels, the rendering level of detail *lod* should be chosen and a proper transfer function and footprint size should be used according to the *lod*. As we mentioned, it is possible to accelerate the performance of the lines 6 and 7 by texture mapping hardware.

3 RESULTS

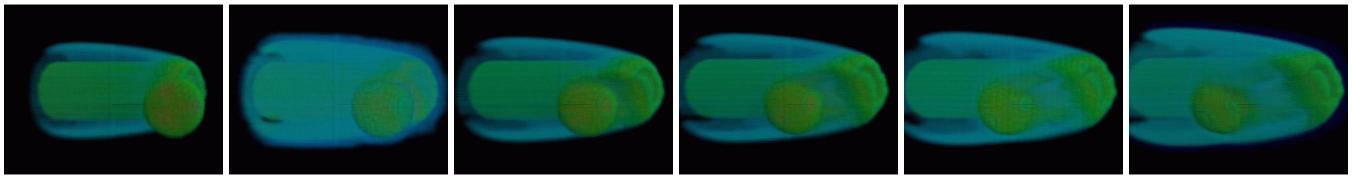
Our splatting technique was implemented on a PC, equipped with a 800 MHz Intel Pentium III Processor, 256MB main memory and a graphics card with an NVIDIA GeForce3 processor and 64MB of memory. To test our proposed algorithm, we implemented a program in OpenGL with GLUT.

Our test data is the result from a simulation of a radiative jet colliding with a dense cloud. The simulation result is stored in AMR format with a $64 \times 64 \times 128$ finest-grid resolution and a four-level hierarchy $(0 \leq l \leq 3)$. Fifteen function values are given at the nodes of the mesh in floating point format $(0 \leq v \leq 14)$. We scaled the values to range from 0 to 4095. Interesting values include energy density $(v = 0)$, mass density $(v = 4)$, electron density $(v = 6)$, and so on. The image sequences in Figure 3 and 4 were generated from



(a) $t = 11$ (b) $t = 12$ (c) $t = 13$ (d) $t = 14$ (e) $t = 15$ (f) $t = 16$

Figure 3: Images generated from mass density ($v = 4$) of test time-varying data



(a) $t = 16$ (b) $t = 17$ (c) $t = 18$ (d) $t = 19$ (e) $t = 20$ (f) $t = 21$

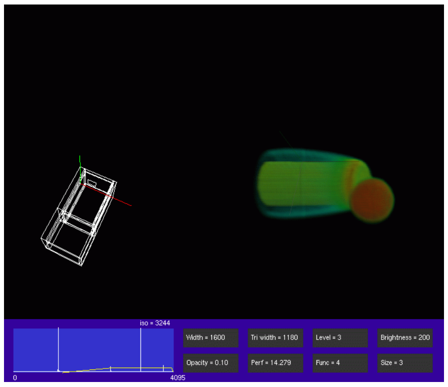
Figure 4: Images generated from electron density ($v = 6$) of test time-varying data

mass density and electron density function values of the test time-varying AMR data using our technique respectively.

Using a k-d tree data structure, we are able to effectively limit the search space. As described before, the bricks have a code representing the range of isovalues contained in the brick. This helps us to achieve considerable performance. The octree structure at each brick of the k-d tree helps to further limit the search space. It was observed that a maximum octree width of 8 was quite efficient when the number of voxels was greater than 40,000 to 50,000. Less dense volumes performed well with an octree size of 4. We show the results comparing the performance due to addition of the octrees and also compare the performance using 4, 8, and 16 as the maximum octree width. Figure 5 shows the ratios of searched voxels and search gains resulted from rendering the time-varying image sequence in Figure 3.

Nearly interactive frame rates were achieved using our implementation. Although considerable gains in limiting the search space was achieved with smaller octree width values, the search time became a bottleneck. We give both the gain in the search space limitation and the speed of rendering as comparison with different octree widths. Figure 6 shows the resulting rendering speeds and gains from generating the images of Figure 3. We define search gain and rendering gain as follows: $search\ gain = \frac{\eta_{without} - \eta_{with}}{\eta_{without}}$, $rendering\ gain = \frac{\tau_{without} - \tau_{with}}{\tau_{without}}$ where $\eta_{without}$ and η_{with} are the number of voxels searched without and with octree, respectively, similarly, $\tau_{without}$ and τ_{with} are the rendering speeds without and with octree, respectively.

The data structure combination of k-d trees and octrees result in interactive rendering of data sets as large as $64 \times 64 \times 128$ with highest resolutions. While we get very fast selection of regions in space where relevant voxels exists with our k-d trees, the octree further improves search time and gives a natural ordering of voxels for any view direction. Although the relevant bricks of the k-d tree are sorted for every viewing change, it does not prove to be a bottleneck due to the limited number of selected bricks. This is particularly true of volumetric data, where the region of interest is usually not dense throughout the object space. Hence we have partially solved the problem of quickly obtaining an ordering on the rendering primitives. This gives us significantly faster rendering speeds.



iso = 3244

Width = 1600 | Tri width = 1180 | Level = 3 | Brightness = 200

Opacity = 0.10 | Perf = 14.279 | Func = 4 | Size = 3

Figure 7: The graphical user interface for interactive rendering

Figure 7 shows the graphical user interace desiged for interactive rendering. Users can set transfer function, rendering and viewing parameters through the interface.

4 CONCLUSIONS AND FUTURE WORK

We presented a hierarchical multi-resolution splatting scheme to render AMR data interactively. Our technique constructs a k-d tree and octrees from AMR data during preprocessing. The data structures are used effectively for rendering and storing the data. Our technique takes advantage of hardware accelerated 2D texture mapping supported by NVIDIA GeForce card in implementing an interactive splatting algorithm. We designed a graphical user interface to allow users to select various viewing / rendering parameters

Frame	Num of total voxels	Num of voxels in searched bricks				Search gain (%)		
		no octree	octree 16	octree 8	octree 4	octree 16	octree 8	octree 4
(a)	1120288	26656 (2.4%)	24057 (2.1%)	14874 (1.3%)	4700 (0.4%)	9.7	44.2	82.4
(b)	1174304	48880 (4.2%)	39234 (3.3%)	17904 (1.5%)	5966 (0.5%)	19.7	63.4	87.7
(c)	1203872	58672 (4.9%)	51148 (4.2%)	31898 (2.6%)	11479 (1.0%)	12.8	44.5	80.4
(d)	1274336	87520 (6.9%)	68328 (5.4%)	39585 (3.1%)	13940 (1.1%)	21.9	32.0	84.0
(e)	1391776	143520 (10.3%)	81052 (5.8%)	37376 (2.7%)	12614 (1.0%)	43.5	73.9	91.2
(f)	1412992	164736 (11.7%)	164736 (7.2%)	47010 (3.3%)	16502 (1.2%)	38.3	71.5	89.9

Figure 5: The ratios of searched voxels and search gains in the finest level

Frame	Rendering speed (frames per second)				Rendering gain (%)		
	no octree	octree 16	octree 8	octree 4	octree 16	octree 8	octree 4
(a)	111.9	106.1	120.2	140.4	-5.2	7.4	25.5
(b)	60.6	59.7	75.3	88.3	-1.5	24.3	45.7
(c)	43.3	41.1	45.6	51.3	-5.1	5.3	18.5
(d)	29.6	29.4	54.8	35.7	-0.7	85.1	20.6
(e)	21.1	23.1	28.1	29.3	9.5	33.2	38.9
(f)	18.0	18.4	23.5	23.5	2.2	30.6	30.6

Figure 6: Rendering speeds and rendering gains in the finest level

as well as a transfer function.

An important challenge is to apply our scheme to parallel rendering of AMR data. Our k-d tree based splatting scheme has a good structure to be extended to parallel rendering. The view dependent brick list can be considered as a task pool. Assume that there is a master processor and several slave processors for this parallel scheme. The master processor assigns a task to a proper slave processor and composites partial images from slave processors according to a sorted brick order. Each slave processor loads the assigned bricks, creates partial images using splatting, and then sends them to the master processor. Another challenge is to implement an encoding and rendering method exploiting temporal coherence for time-varying AMR data. If we develop lossless or lossy compression techniques, the data can be stored in compact forms and thus rendering speed to produce videos for analyzing time-varying data can be enhanced. Finally, we can consider combining images generated from simultaneously rendering AMR and geometric models. For example, we can create an animation such that a spaceship model navigates a cosmology AMR data space. Since AMR data space is partitioned by a k-d tree, the brick that includes the spaceship can be easily detected.

Acknowledgements

This work was supported in part by NSF grants ACI-9982297, CCR-9988357, a DOE-ASCI grant BD4485-MOID from LLNL/SNL and from grant UCSD 1018140 as part of NSF-NPACI, Interactive Environments Thrust, and a grant from Compaq for the 128 node PC cluster. We thank Marcelo Alvarez and Prof. Paul Shapiro of the Galaxy Formation and Intergalactic Medium Research Group, University of Texas at Austin for allowing us to use the AMR test data. The data was from a workshop hosted by Alejandro C. Raga in Mexico City in 2002.

References

[1] C. L. Bajaj, V. Pascucci, G. Rabbiolo, and D. R. Schikore. Hypervolume visualization: A challenge in simplicity. In *Proceedings of IEEE/ACM 1998 Symposium on Volume Visualization*, pages 95–102, Oct 1998.

[2] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[3] G. L. Bryan. Fluids in the universe: adaptive mesh refinement in cosmology. *Computing in Science & Engineering*, 1(2):46–53, 1999.

[4] R. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *Proceedings of IEEE Visualization'93*, pages 261–267, Oct 1993.

[5] D. Laur and P. Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. *Computer Graphics*, 25(4):285–288, 1991.

[6] L. Norman, M., J. Shalf, S. Levy, and G. Daues. Diving deep: data-management and visualization strategies for adaptive mesh refinement simulations. *Computing in Science & Engineering*, 1(4):36–47, 1999.

[7] G. H. Weber, H. Hagen, B. Hamann, K. I. Joy, T. J. Ligocki, K.-L. Ma, and J. M. Shalf. Visualization of adaptive mesh refinement data. In *Proceedings of the SPIE (Visual Data Exploration and Analysis VIII)*, May 2001.

[8] G. H. Weber, O. Kreylos, T. J. Ligocki, J. M. Shalf, H. Hagen, B. Hamann, and K. I. Joy. Extraction of crack-free isosurfaces from adaptive mesh refinement data. In *Proceedings of the Joint EUROGRAPHICS and IEEE TCVG Symposium on Visualization*, pages 25–34, May 2001.

[9] G. H. Weber, O. Kreylos, T. J. Ligocki, J. M. Shalf, H. Hagen, B. Hamann, K. I. Joy, and K.-L. Ma. High-quality volume rendering of adaptive mesh refinement data. In *Proceedings of the 6th International Fall Workshop on Vision, Modeling, and Visualization 2001*, pages 121–128, Nov 2001.

[10] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, 24(4):367–376, 1990.