

Real-Time Volume Rendering of Time-Varying Data Using a Fragment-Shader Compression Approach

Alécio P. D. Binotto
UFRGS

João L. D. Comba
UFRGS

Carla M. D. Freitas *
UFRGS

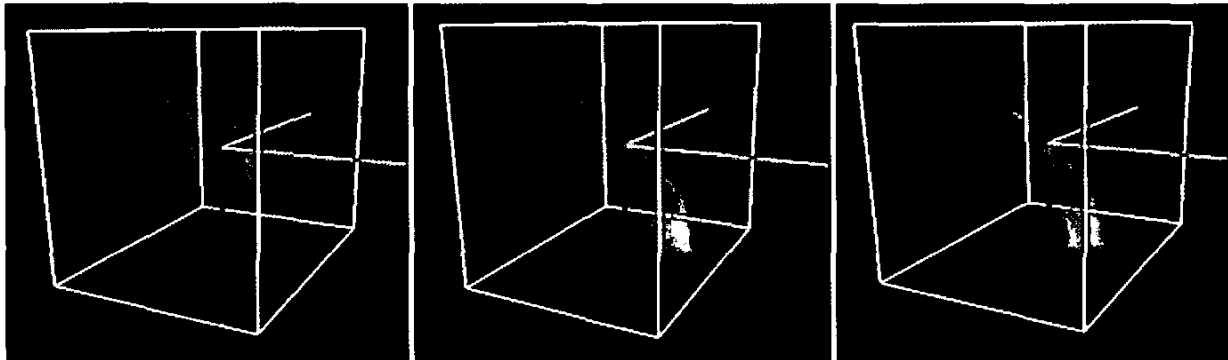


Figure 1: Three time instances of a volumetric fluid decompressed and visualized using the fragment shader.

Abstract

The recent advance of graphics hardware allowed real-time volume rendering of structured grids using a 3D texturing approach. The next challenging problem is to extend the algorithms to time-varying volumetric data (4D functions), which consume more storage and are not directly supported in current graphics hardware. In this work we present a new visualization technique that includes (1) a compression scheme of sparse 4D functions into 3D textures, and (2) a visualization algorithm that decompress the stored data from the 3D textures using the programmability of fragment shaders, allowing real-time visualization of such data. We illustrate the system in action with datasets resulting from computational fluid dynamics simulations.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations, Geometric algorithms, languages, and systems

Keywords: Compression, Volume Rendering, Graphics Hardware

*Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brasil. e-mail:(abinotto, comba, carla)@inf.ufrgs.br

IEEE Symposium on Parallel and Large-Data
Visualization and Graphics 2003,
October 20-21, 2003, Seattle, Washington, USA
0-7803-8120-3/03/\$17.00 ©2003 IEEE

1 Introduction

Volumetric datasets are common in many scientific visualization applications for representing physical phenomena. The advances in acquisition instruments and procedures in the last years substantially increased the size of acquired volumetric data, motivating several work in compression techniques. From a visualization perspective, a good compression technique [Brodie and Wood 2001] is the one that keeps good image quality while keeping high compression rates and low decompression time [Nguyen and Saupe 2001]. One possible approach is to decompose the volumetric data into smaller sub-volumes. In many applications, such as medical imaging, only selected parts of the volume have relevant data or need to be displayed, and compression techniques with this approach have been reported recently [Nguyen and Saupe 2001] [Rodler 1999].

The compression is even more critical in time-varying volumetric datasets (4D datasets), such as the ones obtained in Computational Fluid Dynamics (CFD). The direct volume rendering of such data is computationally very expensive, but it has shown promising improvements with the advances of recent graphics processor units (GPUs). In this paper we present a novel approach to perform volume rendering of time varying data that takes advantage from the support to 3D textures and hardware programmability at fragment level of the GPU. We use a compression strategy based on a hierarchical decomposition of space that stores several instances of volumetric data into 3D textures. The decompression is performed by the fragment shader (entirely in the GPU) during rendering (figure 1).

The paper is organized as follows. We start with a brief motivation for this work, describing the project where this research is being applied. We follow with a review of related work. In Section 4, we present the new features of recent GPUs, especially the programmability aspects of GPUs necessary to understand the compression algorithm presented in Section 5. The description of the visualization model used along the proposed technique is presented in Section 6. Section 7 discusses the results of our work, followed by conclusions and directions for future work.

2 Motivation

The motivation for an efficient way to compress and visualize time-varying data came in the MAPEM project (Environmental Monitoring of Off-Shore Oil Prospecting Activities and Exploration). This project is currently being developed at our University in cooperation with the Brazilian Institute of Oil, under supervision of the Brazilian environmental agency (IBAMA). The project goal is to establish a methodology for evaluating the impact in the marine ecosystem of cuttings discharged during oil well drilling activities. Seabed samples are collected before and after drilling, and are submitted to laboratory and statistical analyses by different groups (chemistry, biology, geosciences and statistics).

The discharge simulation is a useful way to understand the results obtained in the laboratory analyses. Visualizing the simulation adds another dimension to evaluate the data, not only concentrations at seabed, but the entire discharge process. It can be useful, for instance, to explain the effect of the water column and currents in the trajectory of fluid and cuttings, and why some areas receive materials. Approximations of the discharge process were obtained using a simulator called OOC [Brandsma and Smith 1999], that allows the user to specify input parameters describing the discharge process (pipe location and orientation, salinity, temperature, currents, drilling fluid composition, etc). OOC produces as output an estimated dispersion of discharged materials, as well as the accumulated deposition of cuttings on the seabed. These data are represented in uniform 2D planar sections along the water column, and can be sampled at any instance of time during the simulation. The dynamics of cuttings dispersion through the water column and their deposition on the seabed produce a very sparse dataset with high spatial and temporal coherence.

3 Related Work

Many data representation and compression techniques have been reported to improve volume rendering techniques [Kaufman 1991]. Many of these techniques use hierarchical data structures to accelerate rendering time of static volume data [Levoy 1990][Laur and Hanrahan 1991], while others focus on efficient data storage and transmission instead of run time optimizations [Muraki 1993][Luo et al. 1996]. These approaches rely on the spatial coherence property of many volume datasets, i.e., on the fact that adjacent voxels often have similar values. The use of 3D texturing hardware has also significantly accelerated volume rendering [Cabral et al. 1994][Yagel et al. 1996][LaMar et al. 1999][Guthe et al. 2002]. This approach is based on the fact that volume data can be represented in a 3D texture, sampled by a sequence of slicing planes orthogonal to the viewing direction. Texturing hardware is used to map colors and opacities based on the voxels values to each slicing polygon.

For volumetric datasets larger than the available texture memory, decomposition into sub-volumes (which may imply a compression scheme) must be applied to the data. Boada et al. [Boada et al. 2001] described a texture-based volume rendering technique that encodes octrees using 3D textures. The octree construction and storage is based on two criteria, data homogeneity and data importance. The method is efficient for rendering large datasets, but does not address time-varying data (same happens in [Cabral et al. 1994][LaMar et al. 1999]).

The fact that voxel information does not change drastically in time (temporal coherence) plays an important role in the compression of time-varying data. Westermann [Westermann 1995] has used this characteristic to compress each time instance volume using wavelet transforms. The multiscale components of each volume are examined regarding time evolution and additional processing results in a multiresolution representation for each wavelet coeffi-

cient. Rendering is performed through a multiresolution visualization framework [Westermann 1994].

In [Shen et al. 1999], an efficient structure called the TSP-tree (Time-Space Partitioning) was created to support visualization of temporal series of volumetric data using a ray-casting algorithm. The TSP-tree uses a modified octree to represent a time-varying volume both in temporal and spatial domains. The difference between a TSP-tree and a regular octree is that a regular octree node contains spatial information and a TSP tree node maintains both spatial and temporal information about the underlying data in the sub-volume. To store the temporal information, each TSP tree node itself is a binary tree, which divides recursively the entire time interval represented in the dataset. In a follow-up work, Ellsworth et al. [Ellsworth et al. 2000] developed a texture-based volume rendering technique based on TSP-trees.

Besides the speed-up obtained by using 3D-texturing, modern graphics chips are now providing programmable features that can be used in new visualization algorithms. Engel et al. [Engel et al. 2001] proposed a texture-based volume rendering algorithms that uses the dependent-texture features of recent boards to compute an approximation of the volume rendering integral. The viewing-aligned slices are texture-mapped into scalar values obtained from the volume dataset. During rasterization, two pairs of scalar values sampled from adjacent slices are used to as indexes into a third (dependent) texture that contains pre-integrated colors and opacities.

Recent work from Guthe et al. [Guthe et al. 2002] and Kraus and Ertl [Kraus and Ertl 2002] use the programmable flexibility of graphics hardware. In [Guthe et al. 2002] a volume is stored as an octree. The original volume is divided in cubic blocks, and wavelet filters are applied to each block. The recursive grouping-and-filtering of the blocks until a single block is obtained results in a hierarchical, multi-resolution representation of the volume. Rendering is accomplished by selecting from the octree only the set of nodes that provide a good approximation of the original volume from the specified viewpoint. Each block is loaded in a 3D-texture and view-aligned slices are displayed in back-to-front order.

Kraus and Ertl [Kraus and Ertl 2002] apply volume rendering to 2-, 3- and 4-D datasets. They represent four-dimensional functions using texture memory as an index grid and corresponding data blocks. This representation requires two three-dimensional lookups to be performed using two adjacent cells in the index data grid. The results are further used in another two trilinearly interpolated texture lookups to find the final value of a pixel. Since our approach is also based on indexing blocks of data in texture memory, differences to the Kraus and Ertl's approach are pointed out in section 5.

4 Graphics Hardware

The advance of graphics hardware in the last few years made it possible to incorporate many parts of the graphics pipeline that were previously only implemented in software. The resulting generation of graphics cards, called graphics processor units (GPUs), not only have higher performance (measured by higher fill rates or triangles rendered per second), but also add newer ways of generating complex graphics effects.

A fundamental aspect in this process is the programmability of recent GPUs (figure 2). Previous generations of graphics boards were able to implement the entire graphics pipeline in hardware, but using a fixed-function pipeline. The need for small variations on how computation was done in the pipeline led to many special modes that were too cumbersome to be incorporated into the existing APIs (such as OpenGL or DirectX). A much more elegant and orthogonal solution would be to make small parts of the pipeline programmable, increasing the expression power of the GPU to the limits imposed by the languages used to program the hardware. In

this model, a user-defined program is load into the graphics board to control the behavior of specific parts of the pipeline. Early languages described to program the hardware were assembly languages, but recently higher-level shading languages such as Cg (C for graphics) are available [NVIDIA-Corporation 2003].

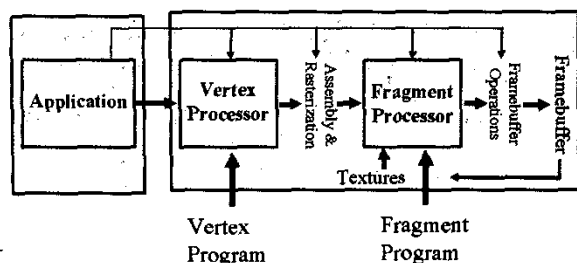


Figure 2: Programmable Graphics Processor Units (GPUs). The graphics pipeline can be programmed at the vertex and fragment levels.

Two parts of the pipeline were made programmable, involving computations at a vertex and fragment levels. Computations at vertex level are done early in the pipeline, involving vertex transformations, illumination calculations and texture coordinate generations. A vertex program can be loaded at the GPU at any instance, and works with one vertex at a time, with no deletion or creation of vertices. Applications that benefit from vertex programs are usually related to the modeling of shapes (deformations, interpolation, morphing, skinning) or complex illuminations models.

Later in the pipeline there is another part that allows modifications on the final pixels written into the frame buffer. A fragment, also referred as a pixel with information (colors, texture coordinates, fog, etc), can be manipulated in many ways. In addition, advanced texturing hardware allows now to perform a texture access from inside a fragment program. Texture results can be used in some arithmetic operations, with the result being used to perform another texture access (dependent texturing). Current graphics hardware allows several texture accesses to be performed in the fragment shader, with limited number of dependent operations. Longer fragment programs with many texture accesses are usually very expensive for current graphics hardware.

5 Compression Algorithm

The compression mechanism was designed in such way that time-varying data can be stored into graphics hardware memory (3D textures), and decompressed during rendering using fragment shaders. The method achieves higher compression rates when volumetric data are sparse and presents high coherence (spatial or temporal), which was the case in our CFD datasets.

The input to the algorithm consists of several time instances, each represented by one volumetric dataset. Only scalar values are used because in our application we are only concerned with accumulation values at each cell of the volumetric dataset.

5.1 Indexing Mechanism

The compression approach consists in creating an *indexing* mechanism in a two-step process. In the first step, a regular decomposition of each volume into sub-volumes is performed, resulting in a coarser volume that will be stored in an *index structure*. In this structure, homogeneous and non-homogeneous (to be refined) regions will be represented in a different way. For homogeneous regions, the index will contain mappings from scalar to color values

(RGBA). For non-homogeneous regions, it will contain the address to a *refinement structure*, generated in a second step of the algorithm.

We use a 2D example to illustrate the approach (Figures 3 and 4). In Figure 3(a) we have an input 8x8 grid, which is subdivided into a coarser 4-4 grid that is stored in the index structure (Figure 3(b)). For each non-homogeneous cell in the coarser grid, a refinement until the grid cell level is stored in the refinement structure (Figure 3(c)). In Figure 4 we illustrate the addition of a second 8x8 grid, its storage in the index (Figure 4(b)) and refinement (Figure 4(d)) structures. Note that refinements are re-used if they were already represented in the refinement structure.

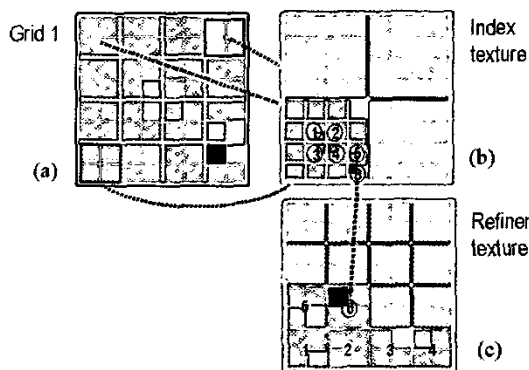


Figure 3: 2D example of the compression scheme: (a) original grid representing one time instance, (b) index structure and (c) refinement structure.

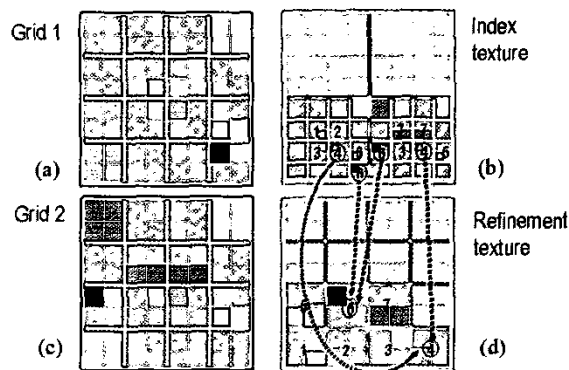


Figure 4: 2D example of the compression scheme: (a) grid representing first time instance, (b) index structure already presenting the entries corresponding to the second grid (c) and (d) refinement structure.

5.1.1 First Phase: Homogeneous Data

The first-step consists of the decomposition of each volume into a coarser grid defined by a certain level. We do not use an octree because its adaptive nature would require a more advanced indexing scheme that would require a longer fragment program.

Each cell in the coarser grid is analyzed with respect to its homogeneity (same intensity value for all voxels). If it is homogeneous, the corresponding cell in the index grid is set to the intensity value

and stored in the index structure as the 4-tuple (R, G, B, A) corresponding to the transfer function associated with the scalar value - see Figure 3(b).

5.1.2 Second Phase: Non-Homogeneous Data

For non-homogeneous regions, we represent the entire sub-volume into a refinement structure, keeping in the index structure a reference that allows it to be accessed. Since only one level of indirection is used, and that all regions to be refined have constant size, it suffices to use as reference the origin of the refined region in the refinement structure. This origin is stored at each cell of the coarser grid in the index structure (Figures 3 and 4).

The process is repeated for every volume in a time-varying dataset. After processing all time instances, the index structure will contain one sub-volume for each time-instance, and the refinement structure will contain sub-volumes for non-homogeneous regions. In order to explore spatial and temporal coherence, refinement information is reused using a hashing scheme. Figure 4(b) shows the sharing of non-homogeneous nodes by two time instances. Note that collisions must be checked using individual comparisons for all voxels in a sub-volume.

5.2 Representation using 3D textures

The index and refinement structures are stored in hardware using two 3D textures (the index and refinement textures). The texture format used is RGBA with 32 bits, each component clamped into a $[0..1]$ interval. The graphics hardware used was an ATI 9700 Pro, that has 128MB of memory and up to 64 instructions fragment programs. The dimension d of the 3D texture was chosen to be 128 (a 128^3 volume), enough to store the input data generated by the CFD simulator used in our tests. Also, it represents the largest power-of-two dimension that can be used if two 3D textures are necessary (8MB for each texture).

Consider a d^3 volume representing a time instance. The volume is divided into a coarser grid defined by a user-defined level s . The resulting coarser cell is defined by a d_i^3 volume, where $d_i = 2^s$. The number of coarser cells that are stored in the index structure is $n_i = (d/d_i)^3$. The refinement cell is defined by a volume d_r^3 , where $d_r = d/d_i$. The number of refinement cells that are stored in the refinement structure is $n_r = (d/d_r)^3$.

For example, consider a 128^3 time instance subdivided into level 4. The coarser cell side is $d_i = 2^4 = 16$ leading to a 16^3 sub-volume and $(128/16)^3 = 512$ cells to be stored in the index structure (512 time instances). The refinement cell has side $d_r = 128/16 = 8$ leading to a 8^3 sub-volume and $(128/8)^3 = 4096$ cells available in the refinement structure.

Changing the coarser level used to create the index structure affects the amount of storage used in the index and refinement structures. The choice of which coarser level to be used depends on the input data properties (coherence and sparseness) and can be chosen to give better compression rates. Figure 5 shows a 3D example for the proposed compression scheme.

Central to the decompression algorithm to be described in section 6 is the problem of distinguishing if a coarser volume in the index structure is refined or not. Setting the alpha channel of values stored in the index structure to 1 indicates that a refinement is used (alpha less than 1 otherwise). The distinction between the two cases is done in the fragment program by checking the alpha value corresponding to the value returned from the index texture.

5.3 Comparison: Adaptive Texture Maps

Our technique applied to time-varying data is similar to the Adaptive Texture Maps proposed by Kraus and Ertl [Kraus and Ertl

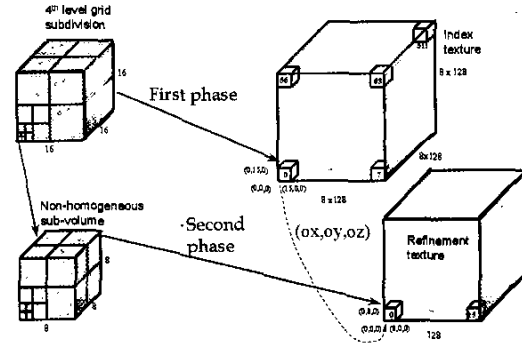


Figure 5: Two-indexed storage technique using 3D textures.

2002]. When applied to four-dimensional functions, they also use only the first texture as an index. However, their technique is not based on a subdivision of the four-dimensional domain. Their indexing scheme requires that each cell of the index data (and therefore one entire data block) covers all possible values of one of the coordinates, i.e., they have only one subdivision and the index texture stores only addresses and not primitive values as our method does. Finally, their approach is based on storing sub-volumes along with a scale factor, while in ours the original data values are kept also for non-homogeneous sub-volumes.

The coarser subdivision and further refinement of non-homogeneous nodes, as well as the use of a hashing scheme provide an efficient compression method. In addition, it allows taking advantage of the programming flexibility of recent hardware, resulting in real-time rendering of dynamic data, as will be explained in the next section.

6 Fragment Shader Decompression

The volumetric visualization uses a sequence of parallel slices orthogonal to the viewing direction that sample the 3D texture in back-to-front fashion as in the method described in [Engel et al. 2001]. Currently we are not using the pre-integrated calculation used for shading, but this will be incorporated in the future.

In each rendering step we first select which time instance t we want the corresponding volume to be visualized. The sequence of slices used to sample the volume generate fragments that will go through a decompression stage inside the fragment shader.

For each fragment generated, the first task in the decompression is to identify where in the index structure the information regarding the time instance t is stored. We separate this calculation in two parts: (1) static (base) passed as parameter to the fragment shader and (2) dynamic (displacement) that is computed inside the fragment shader. The base corresponds to the origin ($origin_i$) of the coarser volume associated with time t in the index structure, which can be computed using the following formula:

$$\begin{aligned} origin_i.x &= (t \bmod d_i)d_r \\ origin_i.y &= (t \div d_i)d_r \\ origin_i.z &= (t \bmod d_i^2)d_r \end{aligned} \quad (1)$$

This origin is passed to the fragment shader as a parameter, along with fragment texture coordinates (s, t, r, q) and handles to the two 3D textures (IndexTexture and RefinementTexture). The dimension of the coarser cell side d_i is defined as a constant inside the fragment program to reduce the instruction count of the fragment program, but could also be passed as a parameter if necessary.

The decompression performed inside the fragment shader involves a sequence of actions that can be enumerated as follows:

1. **Compute the index address:** This involves adding the origin passed as parameter to a displacement that locates the relative position of the current fragment inside the index texture.

These parameters are used by the fragment shader to calculate the index used to access the index texture.

$$index_i.xyz = origin_i.xyz + (s,t,r) \bmod d_i \quad (2)$$

2. **Access index texture:** Uses the address calculated before and retrieves a RGBA value.

$$value_i.rgba = tex3D(IndexTexture, index_i.xyz) \quad (3)$$

3. **Evaluate if a refinement was used:** If $value_i.a < 1$, no refinements are necessary and $value_i$ is returned as the fragment color. Otherwise, a refinement was used and computation continues.

4. **Compute the refinement address.** The color returned in the index structure represents the origin of the refinement volume in the refinement texture. The displacement inside the refinement requires a *mod* operation with the side of each refined cell (actually *fmod* since index values are normalized).

$$index_r.xyz = index_i.xyz + fmod((s,t,r), d_r/d) \quad (4)$$

5. **Access refinement texture:** Uses the address calculated before and retrieves a RGBA value.

$$value_r.rgba = tex3D(RefinementTexture, index_r.xyz) \quad (5)$$

6. **Return the fragment color:** Returns $value_r$ if a refinement was used ($value_i$ otherwise).

In figure 6 we have the general scheme of the decompression. Appendix A gives the entire Cg code used in the fragment shader.

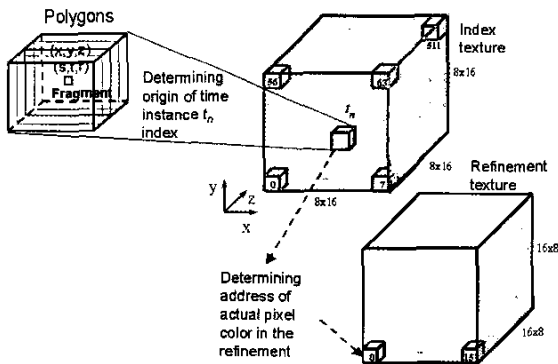


Figure 6: General scheme of the decompression for visualization.

7 Results

The case study was run on a PC system with a Pentium IV 2.0 with 640MB and a graphics board ATI 9700 Pro. Datasets come from simulations of a discharge process using the OOC Model [Brandsma and Smith 1999]. We used three datasets each one with 36 time instances, each defined by a volume of size 128^3 (each dataset comprises more than 300MB of storage). Due to a non-disclosure agreement, simulations reported here were slightly modified from the ones used in the MAPEM project without changing the characteristics of the generated volumes. Table 1 shows the characteristics of these datasets regarding sparseness. Temporal coherence results are exactly the same in this case since temporal differences for our data occur only in non-empty regions.

Table 1: Datasets characteristics.

| Dataset | Spatial sparseness(%) |
|---------|-----------------------|
| 1 | 99.92 |
| 2 | 99.65 |
| 3 | 99.51 |

Table 2 shows results obtained from the compression method applied to all three datasets with subdivisions at the third, fourth and fifth level.

Table 2: Compression results.

| Dataset | Subdivision level | Index Texture (cells) | Refinement Texture (cells) | Compression ratio |
|---------|-------------------|-----------------------|----------------------------|-------------------|
| 1 | 3 | 73728 | 5160960 | 57.69:1 |
| 1 | 4 | 589824 | 1374208 | 153.76:1 |
| 1 | 5 | 4718592 | 452864 | 58.39:1 |
| 2 | 3 | 73728 | 22396928 | 13.43:1 |
| 2 | 4 | 589824 | 5451776 | 49.98:1 |
| 2 | 5 | 4718592 | 2472704 | 41.99:1 |
| 3 | 3 | 73728 | 42336256 | 7.12:1 |
| 3 | 4 | 589824 | 7036928 | 39.59:1 |
| 3 | 5 | 4718592 | 3265536 | 37.82:1 |

The proposed method was integrated with a visualization tool of the discharge process (showing the bathymetry data and accumulations on the seabed, pipe position and orientation, etc). Figure 7(a)-(d) shows four snapshots of this tool. Since the index and refinement structures are 3D textures, we could visualize then to inspect how the space is being occupied (figure 7(e)-(h)).

The rendering speed measured in frames-per-second (FPS) is dependent on the number of polygons used to sample the 3D texture, as well as the windows size. Table 3 shows different FPS rates according to these parameters for the visualization of the third dataset, subdivided until the fourth level.

Table 3: FPS visualization rates

| Nr. of polygons | Image size | FPS |
|-----------------|------------|-------|
| 200 | 128^2 | 33.76 |
| 200 | 256^2 | 8.65 |
| 200 | 512^2 | 4.23 |
| 500 | 128^2 | 14.66 |
| 500 | 256^2 | 4.00 |
| 500 | 512^2 | 1.30 |

8 Conclusions and Future Work

The main contribution of this paper is a new approach to compress time-varying data into 3D textures that allows real-time visualiza-

tion by performing a decompression step inside the graphics hardware using fragment shaders. In order to achieve fast visualization, we chose a compression mechanism that was simple enough that would allow decompression using simpler fragment shaders (with limited dependent operations). This was possible because our datasets were highly sparse and have high temporal coherence.

The proposed mechanism is general enough to use as many textures as are available in the hardware, allowing the grid subdivision process to stop at the most adequate level, depending on data characteristics. Future work includes exploring other compression approaches since longer and faster fragment programs are becoming available, and testing higher-dimensional functions such as lightfields, following Kraus and Ertl [Kraus and Ertl 2002].

8.0.1 Acknowledgements

We would like to thank the reviewers for the detailed suggestions and comments. In particular, one reviewer suggested improvements in our Cg code which allowed us to reduce the instruction count of the final program to 16 instructions. This work was partially supported by grants from CNPq and FAPERGS (Brasil).

References

- BOADA, I., NAVAZO, I., AND SCOPIGNO, R. 2001. Multiresolution volume visualization with texture-based octree. *The Visual Computer* 17, 3, 185–197.
- BRANDSMA, M., AND SMITH, J. 1999. *Offshore Operators Committee Mud and Produced Water Discharge Model Report and User Guide*. ExxonMobil Upstream Research Co.
- BRODLIE, K., AND WOOD, J. 2001. Recent advances in volume visualization. *Computer Graphics Forum* 20, 2, 125–148.
- CABRAL, B., CAM, N., AND FORAN, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *IEEE 1994 Volume Visualization Symposium Proceedings*, IEEE Computer Society Press, IEEE Visualization Annual Symposium Series, IEEE Computer Society, 91–98.
- ELLSWORTH, D., CHIANG, L.-J., AND SHEN, H.-W. 2000. Accelerating time-varying hardware volume rendering using tsp trees and color-based error metrics. In *IEEE 2000 Volume Visualization Symposium Proceedings*, IEEE Computer Society Press, IEEE Visualization Annual Symposium Series, IEEE Computer Society, 119–128.
- ENGEL, K., KRAUS, M., AND ERTL, T. 2001. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Eurographics/ACM 2001 Workshop on Graphics Hardware Proceedings*, ACM SIGGRAPH, Eurographics/ACM SIGGRAPH Graphics Hardware Workshop Series, Eurographics/ACM SIGGRAPH, 15–22.
- GUTHE, S., WAND, M., GONSER, J., AND STRASSER, W. 2002. Interactive rendering of large volume data sets. In *IEEE 2002 Visualization Conference Proceedings*, IEEE Computer Society Press, IEEE Visualization Annual Symposium Series, IEEE Computer Society, 53–60.
- KAUFMAN, A. 1991. *Volume Visualization*. IEEE Computer Society Press.
- KRAUS, M., AND ERTL, T. 2002. Adaptive texture maps. In *Eurographics/ACM 2002 Graphics Hardware Proceedings*, ACM SIGGRAPH, Eurographics/ACM SIGGRAPH Graphics Hardware Workshop Series, Eurographics / ACM SIGGRAPH, 01–10.
- LAMAR, E., B.HAMANN, AND JOY, K. 1999. Multiresolution techniques for interactive texture-based volume visualization. In *IEEE 1999 Visualization Conference Proceedings*, IEEE Computer Society Press, IEEE Visualization Annual Symposium Series, IEEE Computer Society, 355–361.
- LAUR, D., AND HANRAHAN, P. 1991. Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *Proceedings of SIGGRAPH 1991*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 285–287.
- LEVOY, M. 1990. Efficient ray tracing of volume data. *ACM Transactions on Graphics* 9, 3, 245–261.
- LUO, J., WANG, X., CHEN, C., AND PARKER, K. 1996. Volumetric medical image compression with three-dimensional wavelet transform and octave zerotree coding. In *Visual Communication and Image Processing 1996*, SPIE, SPIE 2727 Proceedings, SPIE, 579–590.
- MANTYLA, M. 1988. *An Introduction to Solid Modeling*. Computer Science Press.
- MURAKI, S. 1993. Volume data and wavelet transform. *IEEE Computer Graphics and Applications* 13, 4, 50–56.
- NGUYEN, K. G., AND SAUPE, D. 2001. Rapid high quality compression of volume data for visualization. In *Eurographics 2001 Proceedings*, Blackwell, Eurographics Annual Conference Series, Eurographics, C49–C56.
- NVIDIA-CORPORATION. 2003. *CG Toolkit: A Developer's Guide to Programmable Graphics*. [http://developer/nvidia.com/Cg](http://developer.nvidia.com/Cg).
- RODLER, F. 1999. Wavelet based 3d compression with fast random access for very large volume data. In *1999 Pacific Graphics Proceedings*, Pacific Graphics Proceedings Annual Conference Series, ACM, 108–117.
- SHEN, H.-W., L.-J.CHIANG, AND MA, K.-L. 1999. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *IEEE 1996 Volume Visualization Symposium Proceedings*, IEEE Computer Society Press, IEEE Visualization Annual Symposium Series, IEEE Computer Society, 371–377.
- SILICON-GRAPHICS. 2002. *OpenGL 1.2 specification*. <http://www.opengl.org/developers/documentation/OpenGL12.htm>.
- WESTERMANN, R. 1994. A multiresolution framework for volume rendering. In *ACM 1994 Workshop on Volume Visualization Proceedings*, ACM SIGGRAPH, Computer Graphics Proceedings Series, ACM Press, 51–58.
- WESTERMANN, R. 1995. Compression domain rendering of time-resolved volume data. In *IEEE 1995 Visualization Conference Proceedings*, IEEE Computer Society Press, IEEE Visualization Annual Symposium Series, IEEE Computer Society, 168–175.
- YAGEL, R., D.M.REED, LAW, A., SHIH, P.-W., AND SHAREEF, N. 1996. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *IEEE 1996 Volume Visualization Symposium Proceedings*, IEEE Computer Society Press, IEEE Visualization Annual Symposium Series, IEEE Computer Society, 55–62.

Appendix: Fragment Shader in Cg

```

struct vert2frag {
    float4 texCoord : TEX0;
};
struct frag2frame {
    float4 color : COLOR0;
};
#define TEX_DIM 128
#define DR 8
frag2frame
main( vert2frag IN,
    uniform float3 origin,
    uniform sampler3D texIndex,
    uniform sampler3D texRefinement) : COLOR
{
    frag2frame OUT;
    OUT.color = f4tex3D(texIndex, origin.xyz + IN.texCoord / DR);
    float3 refIndex = OUT.color.xyz + fmod(IN.texCoord, DR / TEX_DIM);
    if (OUT.color.w == 1.0)
        OUT.color = f4tex3D(texRefinement, refIndex);
    return OUT;
}

```

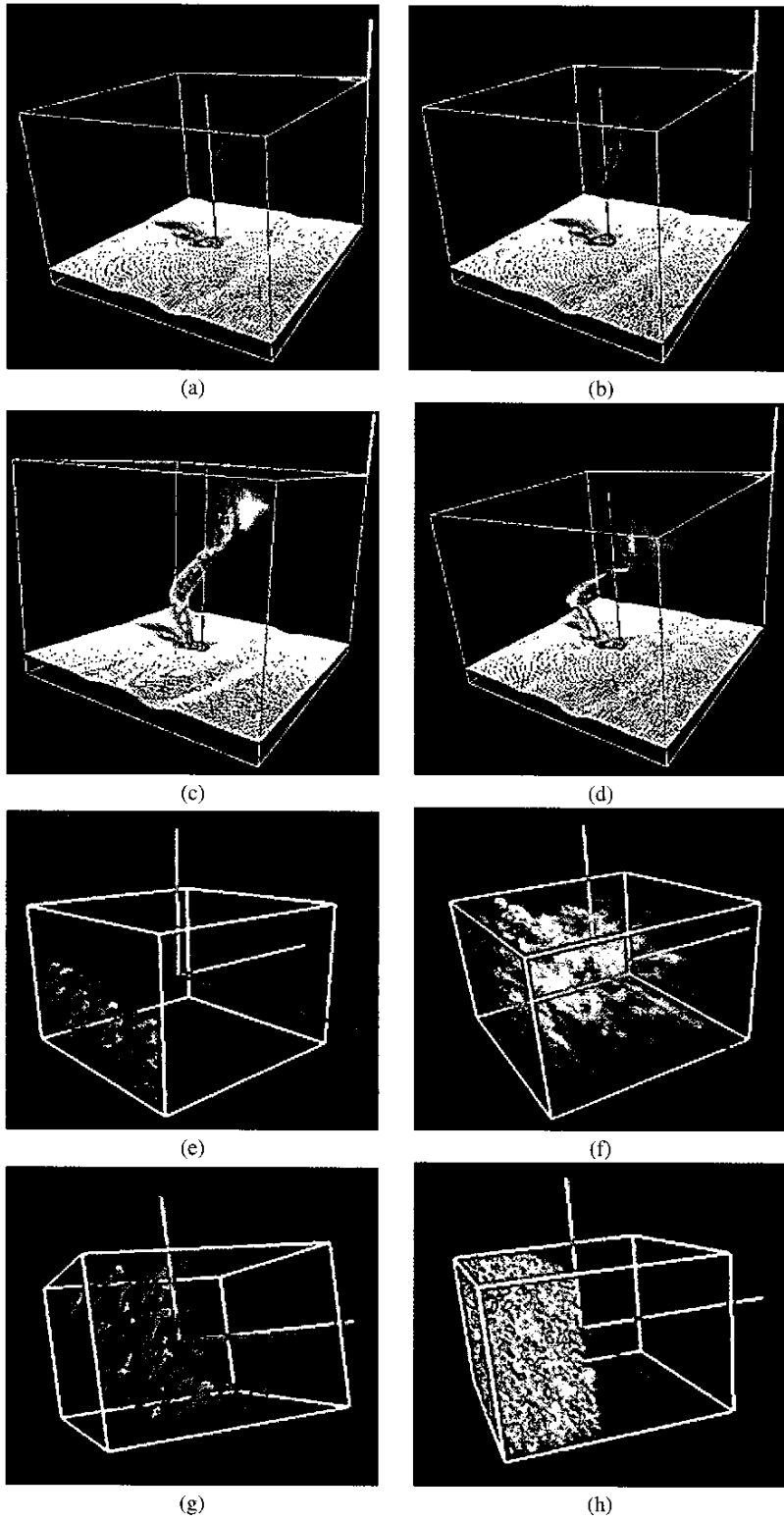


Figure 7: (a)-(d) Another example of results obtained using our fragment-shader decompression approach. (f)(h) Visualization of the index texture and its corresponding refinement texture(g)(i)

8