GPGPU Enabled Ray Directed Adaptive Volume Visualization for High Density Scans

James H. Money
Idaho National Laboratory
Idaho Falls, Idaho
james.money@inl.gov

Marko Sterbentz
University of Southern California
Los Angeles, California
sterbent@usc.edu

Nathan Morrical
University of Utah
Salt Lake City, Utah
bitinat2@isu.edu

Thomas Szewczyk
Idaho National Laboratory
Idaho Falls, Idaho
thomas.szewczyk@inl.gov

Landon Woolley
Brigham Young University - Idaho
Rexburg, Idaho
woolleylandon@gmail.com

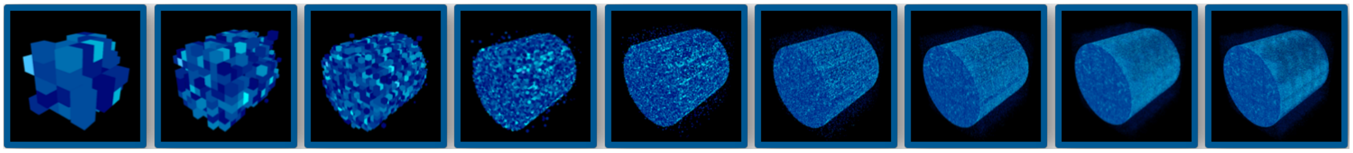


Figure 1: Unity 3D based adaptive open source volume visualizer using the Gray Rot dataset rendered at increasing levels of detail using the hierarchical Z-Order data format.

ABSTRACT

This paper presents an open source implementation of a volume visualizer capable of rendering large scale simulated and tomographic volume datasets on everyday commodity hardware. The ability to visualize this data enables a wide range of analysis in many scientific disciplines, such as medicine, biology, engineering, and physics. The implementation presented takes advantage of a hierarchical space filling curve to efficiently query only a subset of the data required for accurate visualization. A major advantage of this implementation is that it is open source, and works on a wide range of visualization devices by using the Unity game engine, a cross platform engine capable of targeting over 25 platforms [24]. This work's unique contribution is utilizing a space filling curve for multiple bricks in the volume to permit adaptive streaming at different resolutions.

CCS CONCEPTS

• **Human-centered computing → Visualization toolkits**; *Scientific visualization*; *Information visualization*; • **Computing methodologies → Parallel algorithms**; **Ray tracing**;

KEYWORDS

exascale data, volume rendering, hierarchical rendering, scientific visualization, parallel computing, ray tracing

ACM Reference Format:
James H. Money, Marko Sterbentz, Nathan Morrical, Thomas Szewczyk, and Landon Woolley. 2018. GPGPU Enabled Ray Directed Adaptive Volume Visualization for High Density Scans. In *PEARC '18: Practice and Experience in Advanced Research Computing, July 22–26, 2018, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3219104.3219105

1 INTRODUCTION

Interacting with large simulated and tomographic volume datasets can be difficult, and often stands in the way of a volume visualizer's [11] adoption. Doctors and scientists often resort to looking at volumetric data slice by slice. While this is a proven method for volume analysis, information about a third dimension is inherently lost during visualization. Presenting volumetric data in a three dimensional fashion is much more natural, and can lead to additional discoveries through a more informal and engaging visualization experience. However, difficulties arise when volumetric data becomes too prohibitively large to process in core, and poorly coalesced volumetric sampling per pixel can quickly hinder render performance.

As the resolution and size of modern scientific simulations and experiments increases, it is crucial for the performance of a volume visualizer to scale only on display size and not on the size of the dataset being shown. By enforcing performance to scale only on display size, hardware restrictions like on board GPU memory can be circumvented. This in turn allows volume visualization to be delivered to a broader audience, since volumes can then be visualized on consumer grade visualization equipment such as laptops, virtual reality headsets, and mobile devices.

The overarching goal of this paper is to demonstrate a repeatable volume visualization technique based on state of the art results using commodity game engine technology for use by a wide range of audiences including immersive environments, such as a Cave Automatic Virtual Environment (CAVE) [4]. In immersive environments,

PEARC '18, July 22–26, 2018, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-6446-1/18/07…$15.00
https://doi.org/10.1145/3219104.3219105

it is quite common to see volumes limited to 512x512x512 to permit performance. The solution described in this paper eliminates this issue. This paper is divided into two separate components. First, the conversion of the raw 3D volume to a streaming based format is developed that is optimized for rendering inside Unity 3D [24], the commodity game engine utilized in this paper. Secondly, the volume renderer is developed using a custom shader to perform ray marching [20] that can quickly ingest the volume information from the first phase.

2 RELATED WORK

Direct volume visualization by way of ray casting is a very popular technique for scientific visualization. One of the major challenges with this technique is managing and rendering increasingly large simulated and scanned datasets. Many large scale visualization systems utilize bricking and ray guided techniques [3, 5, 6, 9] in order to reduce the amount of data required to be in GPU memory at once by retrieving and only operating on data that is currently visible on screen. This bypasses the memory limitations on many devices and allows for large scale data to be rendered on commodity hardware in a scalable, output-sensitive manner. An extensive review of the state of the art algorithms and techniques developed are found in [2], including these methods above. Prior examples of scans at lower resolutions using commodity game engines can be found in [1]. Additionally, related work using a standard Z-order curve is found in [25].

Since the regions of the data to be rendered are modified in real-time by the user, I/O processing can often be a highly detrimental bottleneck in exascale applications, with Peterka et al. [22] showing that many applications spend upwards of 90% of their rendering time in the I/O processing stage. Furthermore, as brick sizes get smaller, the number of I/O calls increases, slowing down the rendering [6]. Finding an optimal brick size can aid in speeding up the rendering. In addition, data representations like hierarchical Z-order curves preserve the spatial locality of data while also allowing for hierarchical access [21] and enabling only the required data at a specified level of detail to be retrieved.

3 IMPLEMENTATION OVERVIEW

Our implementation is broken up into three sections. First, we discuss an efficient way to decompose a given dataset into a set of smaller, locality preserving cubes to facilitate out of core rendering. Then, an implementation of an efficient data layout to optimize data access on disk during visualization is covered. Finally, how this optimized data layout can be used to query and render data in an output-sensitive manner is discussed.

4 DATA STORAGE AND LAYOUT

To achieve scalable volume rendering, the layout and storage of a given volume must be efficient. Following many current approaches, we perform an object space decomposition called "bricking", which partitions the volume into more manageable sub regions. These regions help facilitate out of core approaches, since bricks can be loaded and rendered as required without having to stream the volume in its entirety.

To efficiently access data on disk, we additionally optimize the data layout of each brick. In general, reading small bits of data at randomly scattered positions is inefficient compared to reading larger chunks in a continuous layout. Therefore, the data is ordered within each brick along a space filling curve, which helps improve coalesced reads at runtime. Specifically, the HZ-order curve proposed by Pascucci and Frank [21] is used, which allows for hierarchically progressive reads. This HZ-order curve naturally maps to a Z-order curve which can be adaptively modified to restrict sampling to certain resolution levels during visualization.

4.1 Bricking

The volume is partitioned in a way that satisfies a set of constraints. First, each brick must be a power of two cube in order to allow HZ-ordering. Second, a maximum brick size is allowed to be specified in order to accommodate different GPU texture memory constraints. Finally, the minimum brick size is maximized for optimal HZ-ordering while also minimizing padding. For convenience, these parameters are left up to the user. A future paper will discuss various optimization schemes for this method.

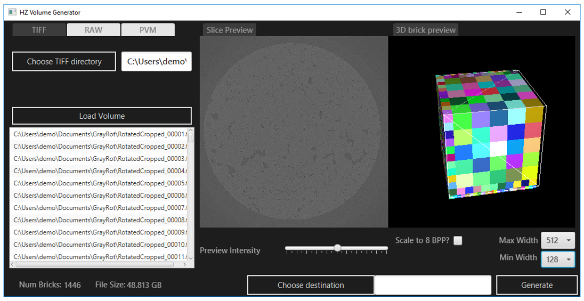
To partition the data, a 3D uniform grid is allocated, where each voxel in the grid contains a position, a size, and a flag. The size for each voxel is initialized as a minimum brick width, and the flag is initialized as true, meaning that voxel should be assumed to be a final brick. The dimensions of this grid are found by dividing the size of the original volume along each axis by the minimum brick width. This method employed is similar to ideas used in [6, 9].

In order to provide efficient computational time for generating the bricks, we utilize GPGPU algorithms to speed computation. We focus on OpenCL [8] as the language of choice, vice CUDA [19], to enable cross-platform capability for non-nVidia cards. Original tests on CPU only algorithms showed over a two day computation time for larger volumes, while utilizing OpenCL allowed our computational time to reduce to under one hour. We will examine this in more detail in the results section.

For each voxel in parallel, we iteratively check to see if the current voxel's size is less than a maximum brick size. If the current size is equal to or larger than the maximum brick size, that thread is terminated, locking in that voxel as a final brick. Otherwise, we check to see if $any(voxelPosition >= (vs - vs\%i))$, where voxelPosition is a three component position of the current voxel, and vs is a three component vector containing the sizes of the entire volume along each axis. The region defined in size by $(vs - vs\%i)$ is guaranteed to be perfectly divisible by voxels twice as large as the current voxel's size. If the current voxel is outside that region, it is unmergable with any neighboring voxels, so the thread is terminated, locking that voxel in as a final brick.

If the current voxel lies within this region, we check to see if $all(voxelPosition\%i == 0)$, meaning that the current voxel lies on the corner of a soon to be merged voxel. If this is not the case, $voxel.flag$ is marked as false, meaning the current voxel has been merged and should be removed. Otherwise, the voxel's width in each axis, as well as i, is increased by a power of two, and then the loop iterates.

Finally, we perform a parallel stream compaction to keep only the uniform grid voxels whose flag is set to true. Pseudocode for



C:\Users\demo\

C:\Users\demo\Documents\GrayRot\RotatedCropped_00001.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00002.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00003.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00004.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00005.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00006.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00007.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00008.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00009.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00010.t
C:\Users\demo\Documents\GrayRot\RotatedCropped_00011.t

Max Width: 512
Min Width: 128

Num Bricks: 1446 File Size: 48.813 GB

Figure 2: The HZ Generator tool with the bricking scheme for Gray Rot shown on the right.

this algorithm can be seen in Algorithm 1. A visualization of this algorithm can be seen in Figure 2.

Algorithm 1: PartitionVolume

```
for each voxel in the uniform grid do in parallel
    int3 voxelPosition = get_global_id();
    int3 vs = get_entire_volume_size();
    voxel.size = minBrickSize;
    voxel.delete = false;
    for (int i = 2; voxel.size < maxBrickSize; i «= 1) do
        if (any( voxelPosition >= (vs - vs % i) )) then
            break;
        end
        voxel.size «= 1;
        if (any( voxelPosition%i !=0)) then
            voxel.delete = true;
        end
    end
end
```

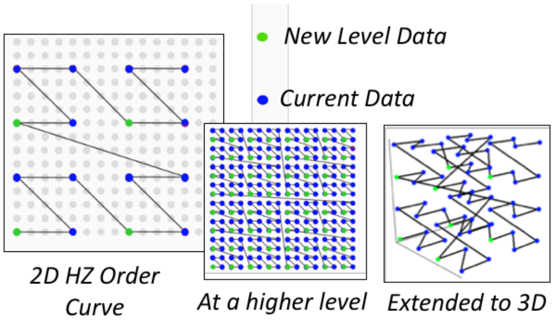
4.2 Curving

Many large scale scientific simulations and experiments produce massive volumes of data in a row major fashion. When sampling these volumes during rendering, row major layouts tend to produce a large number of seek and read operations, which is detrimental to coalesced memory reads.

This scattered sampling can be reduced by ordering voxels along a space filling curve, which preserves spatial locality. When deciding on a curve, concern must be taken into how well a mapping preserves locality, as well as how easy it is to both encode and decode keys.

One such curve which allows constant time key encoding and decoding while preserving spatial locality is the Z-order curve proposed by Morton [18]. Computing the Z-order index given quantized Cartesian coordinates is as simple as interleaving the bits, as shown in Algorithm 2. For more details on how this algorithm works, please refer to the original paper by Morton [18], or to Karras [10] for more explanation on the bit shifting operations shown in Algorithm 2.

Figure 3: The left and center images show the HZ-order curve at low and higher levels of detail in 2D. The right image is the extension to 3D for a cube.



Although Z-order retains spatial locality of the input data, it does so only at full resolution, and does not support hierarchical access. So instead, we use a hierarchical Z-order variant proposed by Pascucci and Frank [21]. The HZ-order curve organizes a given Z-ordered dataset by levels corresponding to a subsampling tree, in which each level doubles the number of points in one dimension. An example of multiple levels of the HZ-order curve are shown in Figure 3.

This hierarchical Z-order curve allows us to stream and query data in an output-sensitive manner while still preserving spatial locality. Pseudocode for mapping Cartesian coordinates to Z-order is found in Algorithm 2 and an optimized version of the Z-order to HZ-order can be seen in Algorithm 3.

Algorithm 2: Naive Cartesian to Z-order

```
unsigned int zId = 0;
for (int i = 0; i < numLevels; i++) do
    zId |= (x & 1 « i) « i | (y & 1 « i) « (i + 1) | (z & 1 « i) « (i + 2);
end
```

Algorithm 3: Z-order to HZ-order

```
int last_bit_mask = 1 « (3 * numLevels);
zId |= last_bit_mask;
zId /= z & -z;
hz = zId » 1;
```

4.3 Apportioning

For optimal read and write times, we curve each brick in the partitioned volume in parallel. Voxels in a particular brick are reordered in parallel in high bandwidth GPU memory. Once curved, each brick can be written to disk using only one seek.

Each block is stored in its own file, and can be allocated before apportioning. In addition, a metadata file is generated which contains information on block sizes and positions. This file is used by the Unity 3D volume renderer to stream in contents.

5 VOLUME RENDERING

Our direct volume rendering approach makes use of volume ray casting, implemented within the fragment shader on a per data brick basis. The entirety of the volume data set is mapped to fit within a unit cube bounding volume. A direct volume rendering approach was chosen due to the need for a minimization of rendering artifacts, as well as the need for hierarchical visualization of the data. Indirect or hybrid volume rendering methods, such as the shear-warp technique [14], would produce lower quality results and introduce visual artifacts when rendering at lower levels of the HZ-order curve.

Ray casting is typically limited due to the need for the entirety of the data set to be available within GPU memory. Our approach mitigates this issue by curving the data into a hierarchical Z-order, allowing only the specific parts of the data that are needed, or possible due to device limitation, to be loaded into the GPU's memory.

5.1 Ray casting

The backbone of our data rendering process is a ray marching technique, which can be seen in Algorithm 4. This inherently parallel algorithm produces high quality results for a large variety of data sets, and allows for changes in the data to be viewed quickly without the need for additional heavy processing.

Rays are projected through the volume and intersect with the data represented by voxels. A specified step size is used to determine the points along the ray at which samples of the data are taken. These intensity values are mapped to a specific color and alpha value based on a predetermined one-dimensional transfer function and are then composited using a standard front to back blending technique. The ray marching is terminated early in two primary cases. The first occurs when the current ray sampling position is outside of the cube. The second case is triggered if the composited fragment color value has reached the maximum alpha value, in which case further sampling along the ray has no practical effect.

Algorithm 4: Ray Casting

```
input  : Ray origin, ray direction
output : Fragment color
ray = IntersectAABB(rayOrigin, rayDirection);
if clipping plane is enabled then
    ray = ClippingPlane(ray, plane);
end
foreach step along the ray do
    intensityValue = SampleIntensity(rayPosition);
    intensityColor = TransferFunction(intensityValue);
    fragColor = FrontToBackBlend(intensityColor);
    rayPosition = rayPosition + rayStep;
    if rayPosition is outside bounding cube or fragAlpha > 1.0
      then
        break;
    end
end
```

5.2 Analysis tools

In order to facilitate more in-depth visual analysis of the data, we added functionality including a one-dimensional transfer function and a clipping plane tool to the volume renderer.

5.2.1 One-dimensional transfer function. Transfer functions are a common way to quickly isolate and highlight different voxel values and allow for the data to be analyzed according to the intensity values of the data set. The input into a one-dimensional transfer function [15] is a single intensity value, which is then mapped to a corresponding RGBA color value. The transfer function is defined by two sets of control points: one for mapping color values, and one for mapping alpha values. Voxels that are not explicitly mapped are determined by interpolating between the control points.

5.2.2 Clipping plane. Depending upon the orientation of the clipping plane relative to the volume, the clipping plane will cull data points from the render pass by modifying the ray that is currently being cast. This enables a cross-sectional view of the data volume and allows the internal voxel data to be examined. Algorithm 5 shows the method utilized for the clipping plane. The variable t declared on line 2 is the distance along the ray at which the intersection between the ray and the plane occurs.

Algorithm 5: Clipping Plane

```
   input  : Ray, plane
   output : Updated ray
1  denom = dot(planeNorm, rayDir);
2  t = 0.0;
3  if denom != 0.0 then
4      t = dot(planeNorm, planePos - rayStart) / denom;
5  end
6  if plane is not facing camera and t < 0.0 then
7      discard fragment;
8  end
9  if plane is facing camera and t > rayLength then
10     discard fragment;
11 end
12 if t > 0.0 and t < rayLength then
13     if plane is facing camera then
14         rayStart = rayStart + rayDirection * t;
15     end
16     else
17         rayStop = rayStart + rayDirection * t;
18     end
19 end
```

5.3 Rendering multiple data bricks

In order to render the data bricks that make up the volume, the bricks are transformed from their native voxel space into the space of the 1x1x1 bounding cube that is used to contain the volume. It is assumed that the bounding cube has its bottom left corner placed at the origin in world space.

A metadata file, in standard JSON format [7], is used to provide mandatory information about the volume, including size of the volume in voxel space, the total number of bricks that comprise the volume, the number of bytes per voxel value, and the minimum and maximum brick size. Information about individual bricks is also provided, including the corresponding file name, the brick's size, and its position in voxel space. Pseudocode for this process can be found in Algorithm 6. The final position and scale of each data brick are computed on lines 7 and 8, respectively.

Algorithm 6: Rendering Multiple Bricks

```
input  : Brick position from metadata
output : Location and scale of brick within
1 foreach brick in volume do
2     brickPosition = (0,0,0);
3     volumeCornerVoxelSpace = boundingCubeCenter -
        volumeCenter;
4     brickPositionVoxelSpace = ReadMetadata();
5     brickPositionVoxelSpace = volumeCornerVoxelSpace +
        brickPositionVoxelSpace;
6     brickOffsetVoxelSpace = brickSize / 2.0;
7     brickPosition = brickPosition + (brickPositionVoxelSpace +
        brickOffsetVoxelSpace) / maxGlobalSize;
8     brickScale = brickSize / maxGlobalSize;
9 end
```

5.4 Utilizing HZ-Order

In order to utilize HZ-ordering, the current position of the ray march in Cartesian coordinates must be converted to the correct index in the HZ-order file format using the method in Algorithm 7. This enables each brick to be rendered at the desired level of detail. In turn, this allows for the volume data to be loaded in only as required by the current level of detail. We achieve this through a four step process.

First, the Z-order index is computed by interleaving the bits of the Cartesian coordinates, as previously mentioned. Then, rather than converting directly to HZ-order, the second step is to compute a masked Z-order index that is based on the desired level of detail. The pseudocode for the Z-order index masking can be found in Algorithm 7. The while loop on line 2 computes the total number of HZ-order levels that a brick has in total, based on its dimensions. Lines 5 and 6 determine the appropriate mask that will effectively quantize the Z-order index to the current level of detail the brick is being rendered at.

Third, the HZ-order index is computed based on the masked Z-order index, as shown in Algorithm 7. In the final step, the computed HZ-order index can be used to index into the brick's data. Depending upon how the data is stored in memory, additional conversions to the HZ-order index may be necessary.

6 RESULTS

We consider the efficiency and speed of the volume renderer and the volume partitioning scheme in this section for several volumes.

Algorithm 7: Level of Detail Z-order Index Masking

```
input  : Render level
output : The masked Z-order Index
1 totalLevels = -1;
2 while brickSize »= 1 do
3     totalLevels = totalLevels + 1;
4 end
5 zBits = totalLevels * 3;
6 zMask = -1 » (zBits - 3 * currentLevel) « (zBits - 3 *
    currentLevel);
7 maskedZIndex = zIndex & zMask;
```

Table 1: 3D scan volume information on dimensions and bit depth

	X	Y	Z	Bit depth
Visible Human	256	256	128	8
Sinus CT Scan	323	397	499	8
Gray Rot	2833	2872	2715	16

Table 2: Video card specification comparison.

	Year	CUDA Cores	Memory(Gb/s)
Quadro 6000	2010	448	144
GTX 980 TI	2015	2816	336
Quadro m6000	2016	3072	317

We examine three volumes of interest: (1) The Visible Human head scan [16] (2) Sinuses from a computed tomography (CT) scan and (3) The Gray Rot dataset. Each of these volumes are shown in Figure 4.

The sinus CT scan includes a volume near the limiting size for software such as Virtual Reality User Interface's (VRUI) [12] volume visualization tool [13]. The scan was collected from a team members doctor's office using a MiniCAT system [23]. Inside the Idaho National Laboratory CAVE environment, this volume suffers from processing speed limitation in the transfer functions. The Gray Rot dataset is an X-ray tomographic scan of NBG-18 graphite at a final resolution of $2833x2872x2715$ and is approximately 42GB in size on disk. Note that this scan is not possible to display in the CAVE at this time without either rescaling or insets of smaller resolution. Our goal, as originally stated, is to render a scan such as Gray Rot, in its native resolution. A summary of the scans is shown in Table 1 including x, y, and z dimensions, along with original bit depth of the image captured.

We consider three different video cards in this study. First is an older nVidia Quadro 6000 card. The second card is a nVidia Quadro m6000. Finally, we compare with the nVidia Geforce GTX 980 TI, which is a common card for gaming systems. The specifications of the cards including CUDA core counts and memory bandwidth are found in Table 2.

First, we consider the pre-processing conversion from raw data scan volumes to HZ-order volumes for each of the scans in Table 3. Note that the first two conversions of the Sinus CT and Visible

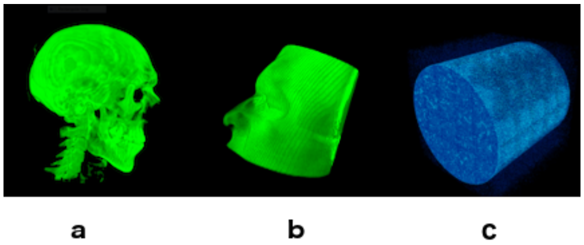


Figure 4: The three data sets used in this paper. a) The Visible Human head scan. b) Sinuses from a computed tomography (CT) scan. c) Gray Rot dataset.

Table 3: Speed comparison across various GPUs and datasets for conversion from raw scan to HZ order bricked volume in seconds.

	Quadro 6000	GTX 980 TI	Quadro m6000
Visible Human	0.306	0.243	0.115
Sinus CT	4.386	2.039	0.754
Gray Rot	66823 ≈ 18.6h	4311.151 ≈ 71.85m	2454.435 ≈ 40.91m

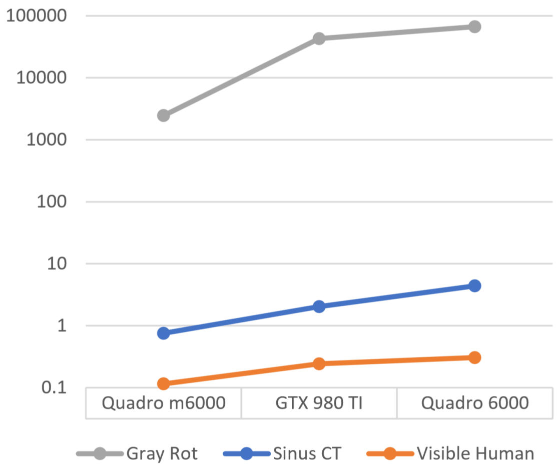


Figure 5: Graph of HZ-order conversion computation time for various datasets and cards. This shows linear speedup across the datasets and cards. Note, the time axis is in the logarithmic scale.

Human datasets complete in less than 5 seconds using all of the cards. However, the method here shows processing speed slowdown on the Gray Rot dataset, but using the Quadro m6000 GPU, the conversion time is faster than the original data collection scan time of over one hour. In this case, it is plausible to complete the conversion in near real-time as the bits are retrieved from the scanner. In Figure 5, we see that the speedup is at least linear for the computation time among the various graphics cards and datasets.

Table 4: Frames per second for rendering the volume on a desktop monitor.

	Quadro 6000	GTX 980 TI	Quadro m6000
Visible Human	192	195	198
Sinus CT	191	195	193
Gray Rot - Level 3	72	122	144
Gray Rot - Level 9	64	122	144

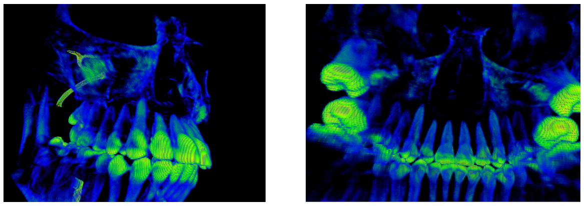


Figure 6: Rendering of an additional skull dataset when using the HTC Vive.

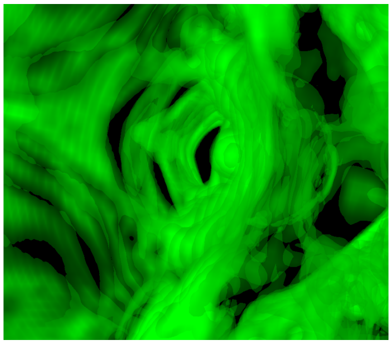


Figure 7: Rendering of Visible Human skull inside the volume.

Some results achieved by utilizing the volume visualization technique on the datasets with the three video cards are shown in Table 4. In terms of frames per second (FPS), the rendering speed on a desktop monitor varies from 64 FPS to 198 FPS depending on the card and size of the dataset. Note that the average person cannot perceive more than 60 FPS in stereo, but for motion in virtual reality systems, we generally strive for 90 FPS. With this in mind, another test case was performed using the HTC Vive headset with the nVidia GTX 980 TI GPU and a locked 90 FPS was achieved for the Visible Human head dataset as well as for an additional skull scan dataset which is depicted in Figure 6.

It is also worth noting that we are aiming to release this software in Fall 2018, and it will be available at https://github.com/idaholab.

7 FUTURE WORK

The ultimate goal is to be able to render exascale data in an output-sensitive manner on any type of device, regardless of memory and processing capacity. To this end, it is necessary to adjust the level of detail of each data brick depending on the location of the brick relative to the camera. In this way the bricks that are closer to the user's eye are more detailed, while bricks that are occluded are rendered at lower levels of detail. This method will reduce the amount of volume data that is required to be buffered over to the GPU, while still producing a high-fidelity image with relevant data to analyze.

Our approach will be to first determine distances from the camera to the front and back sides of the volume, as well as determine how far each of the bricks is from the camera. These distances will then be normalized to an interval of [0, 1] based on the total distance. Lastly, the Z-order render level for each brick can be calculated with a simple linear interpolation based on the bricks' relative distances from the camera.

In order for there to be greater control and modularity with regards to the rendering pipeline, compute shaders will be utilized for the ray-based calculations and general memory management. A brick cache will be used for storing data about bricks currently in GPU memory and at what level of detail they are currently stored as described in [6]. During each frame, the data in the brick cache can be used to determine which data needs to be uploaded to the GPU. The necessary data is then uploaded to a brick buffer, which is a one-dimensional array on the GPU that contains the data values for all of the bricks that currently contribute to the rendering pass. Stream compaction can be used to re-order the data within the brick buffer and remove old data that is no longer used.

We will also consider machine learning algorithms for the implementation of the bricking scheme thresholds. Combining this with optimizations for space and computation time, will help reduce the disk storage required in experimental scans.

Finally, the streaming solution from end-to-end will be developed using the Scientific & Intelligence Exascale Visualization Analysis System (SIEVAS) [17]. This will permit brick level stream controls without physically co-locating the datasets with the tool. Additionally, modules will be developed to permit stream processing in-situ of the data as they are acquired at the scanning location.

ACKNOWLEDGMENTS

We would like to thank Joshua Kane, Idaho National Laboratory, for his support in example datasets including the Gray Rot dataset. We would also like to thank Idaho National Laboratory and the Center for Advanced Energy Studies for their ongoing support for this work.

REFERENCES

[1] Tod T Amon, Edward S Jimenez, and Kyle R Thompson. 2016. Game Development Engines as an Ideal Platform for Exploring 3D Virtual Reality Based Visualizations Constructed from Computed Tomography Data. In *25th ASNT Research Symposium*. 18–22.

[2] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. 2015. State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum* 34, 8 (2015), 13–37. https://doi.org/10.1111/cgf.12605

[3] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. 2009. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. ACM, 15–22.

[4] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. 1992. The CAVE: Audio Visual Experience Automatic Virtual Environment. *Communication of the ACM* 35, 6 (5 1992), 64–72. https://doi.org/10.1145/129888.129892

[5] Klaus Engel. 2011. CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard pcs. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE, 123–124.

[6] Thomas Fogal, Alexander Schiewe, and Jens Krüger. 2013. An analysis of scalable GPU-based ray-guided volume rendering. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*. IEEE, 43–51.

[7] Internet Engineering Task Force. 2018. RFC 8259 - The JavaScript Object Notation. https://tools.ietf.org/html/rfc8259. Accessed: 2018.

[8] The Khronos Group. 2018. OpenCL. https://www.khronos.org/opencl/. Accessed: 2018.

[9] Markus Hadwiger, Johanna Beyer, Won-Ki Jeong, and Hanspeter Pfister. 2012. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2285–2294.

[10] Tero Karras. 2012. Thinking parallel, part III: tree construction on the GPU. *Parallel Forall, Dec* 19 (2012), 8.

[11] Arie Kaufman. 1990. Volume visualization. *The visual computer* 6, 1 (1990), 1–1.

[12] Oliver Kreylos. 2016. Virtual Reality User Interface. http://idav.ucdavis.edu/~okreylos/ResDev/Vrui/. Accessed: 2018.

[13] Oliver Kreylos, Gunther H Weber, E Bethel, John M Shalf, Bernd Hamann, and Kenneth I Joy. 2002. Remote interactive direct volume rendering of AMR data. (2002).

[14] Philippe Lacroute and Marc Levoy. 1994. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM, 451–458.

[15] Marc Levoy. 1988. Display of surfaces from volume data. *IEEE Computer Graphics and Applications* 8, 3 (1988), 29–37.

[16] William E Lorensen. 1995. Marching through the visible man. In *Proceedings of the 6th Conference on Visualization'95*. IEEE Computer Society, 368.

[17] James H Money and Thomas Szewczyk. 2017. Live Integrated Visualization Environment: An Experiment in Generalized Structured Frameworks for Visualization and Analysis. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. ACM, 29.

[18] Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).

[19] nVidia Corporation. 2018. CUDA Zone. https://developer.nvidia.com/cuda-zone. Accessed: 2018.

[20] Steven Parker, Michael Parker, Yarden Livnat, P-P Sloan, Charles Hansen, and Peter Shirley. 1999. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics* 5, 3 (1999), 238–250.

[21] Valerio Pascucci and Randall J Frank. 2003. Hierarchical indexing for out-of-core access to multi-resolution data. In *Hierarchical and Geometrical Methods in Scientific Visualization*. Springer, 225–241.

[22] Tom Peterka, Hongfeng Yu, Robert Ross, Kwan-Liu Ma, and Rob Latham. 2009. End-to-end study of parallel volume rendering on the ibm blue gene/p. In *International Conference on Parallel Processing, 2009*. IEEE, 566–573.

[23] Xoran Technology. 2018. MiniCat Scanner. https://xorantech.com/products/minicat/. Accessed: 2018.

[24] Unity Technologies. 2018. Unity - Multiplatform. https://unity3d.com/unity/features/multiplatform. Accessed: 2018.

[25] Junpeng Wang, Fei Yang, and Yong Cao. 2017. A cache-friendly sampling strategy for texture-based volume rendering on GPU. *Visual Informatics* 1, 2 (2017), 92 – 105. https://doi.org/10.1016/j.visinf.2017.08.001