Java Based Volume Rendering Frameworks

Ruida Cheng[a], Alexandra Bokinsky[b], Paul Hemler[c], Evan McCreedy[a], Matthew McAuliffe[a]

[a] National Institutes of Health, Bethesda, Maryland 20892;
[b] Geometric Tools, Inc.;
[c] Hampden-Sydney College, Hampden-Sydney, VA 23943

ABSTRACT

In recent years, the number and utility of 3-D rendering frameworks has grown substantially. A quantitative and qualitative evaluation of the capabilities of a subset of these systems is important to determine the applicability of these methods to typical medical visualization tasks. The libraries evaluated in this paper include the Java3D Application Programming Interface (API), Java OpenGL© (Jogl) API, a multi-histogram software-based rendering method, and the WildMagic API. Volume renderer implementations using each of these frameworks were developed using the platform-independent Java programming language. Quantitative performance measurements (frames per second, memory usage) were used to evaluate the strengths and weaknesses of each implementation.

Keywords: Visualization, rendering, model, GPU, Jogl , Java3D, Medical Imaging.

1. INTRODUCTION

The volumetric display of large datasets is still a challenging field in scientific visualization. Traditionally, a memory bandwidth bottleneck in the Java Virtual Machine has restricted 3D volume rendering in Java and, generally, it is still not possible to render highly detailed models from large datasets at interactive frame rates with a Java based approach. Several new approaches to overcome this bottleneck have been developed and this paper examines several such solutions to render high quality volume images on a graphics workstation using consumer hardware.

A number of other frameworks for biomedical visualization that already exist lie outside of the scope of this paper. Only a few have been are fully implemented in Java and are capable of taking advantage of the language's portability and ease of programming. VTK (the Visualization Toolkit) is a C++ based visualization library that has been used to develop a number of visualization tools. VolumeJ[1] is a Java based volume rendering toolkit, which has been developed using Java bindings to the VTK library. RTVR[2] is a Java based library for interactive volume rendering. It utilizes clustering techniques to render high quality volume data on low-end consumer PCs over a network connection.

This paper presents and examines four different Java based volume rendering systems. First, we demonstrate the Java3D texture-based and raycast-based volume rendering techniques. Next, we present the WildMagic volume raycasting system, which uses modern graphical processing unit (GPU) features to accelerate the rendering process. Then, we demonstrate a CPU-based Jogl multi-histogram renderer. Last, we show a Jogl OpenGL Shader Language (GLSL) based volume rendering technique. Finally, we discuss the strengths and weaknesses for each approach with respect to performance metrics and qualitative factors. We will also outline the future development path with our implementation.

2. JAVA3D-BASED VOLUME RENDERING

Java3D is a cross-platform, scene graph-based framework that enables the creation of complex visualization tools using the Java programming language. 3-D texture-based and raycast-based volume renderers have been implemented using this framework. Java3D-based volume rendering uses an image scene graph approach for rendering (Figure 1). Each node in the scene graph represents its own entity and is responsible for actions applied to its sub-graph.

Send correspondence to Ruida Cheng: E-mail: ruida@mail.nih.gov, Telephone: 301-496-5363

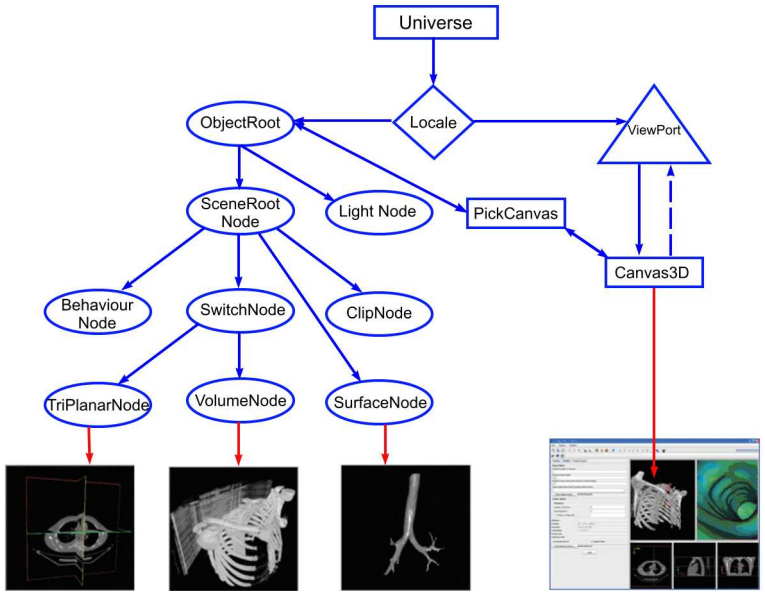


Figure 1. Image scene graph of Java3D based volume rendering.

(a) (b)

Figure 2. 3-D texture proxy geometry. (a) Axis-aligned slices and (b) viewport-aligned slices.

A brief overview of the important Java3D-based volume rendering scene graph nodes is given:

- The SceneRoot Node is the root of hierarchical tree structure that captures the elements of spatial relationships between volumetric components.
- The Tri-planar Node contains the 3-D rendering of the three primary orthogonal slices.
- The Volume Node contains the 3-D texture-based volume view or the raycast rendered image.
- The Surface Node contains a segmented iso-surface (triangle mesh).
- The Canvas3D is the GUI component associated with the native window rendering system to render the final image.

The 3-D texture-based volume renderer uses a single 3-D texture to display the image volume in either axis-aligned or viewport-aligned modes. The axis-aligned slices (Figure 2a) keep one 3-D texture block in memory, cutting planes slice through the volume in each X, Y, and Z direction. Each time the volume rotates, only one of the three directional slices is rendered, determined by the axis that is closest to the viewing direction. The viewport-aligned mode (Figure 2b) also keeps only one 3-D texture block in memory. During the volume rotation, the cutting planes cut through the volume parallel to the viewport.

The final image is composited by rendering the volume back to front. The color lookup is done by mapping the gradient magnitude filter and opacity filter to the voxel intensity. The resulting color is computed using tri-linear interpolation on the graphics hardware. The pseudocode for compositing the final image is shown in Figure 3.

```
Loop through each slice in the volume:
  Fetch the intensity value of each slice into image buffer.
  Fetch the gradient magnitude (GM) values of each slice into the GM buffer.
Loop through each voxel in the slice:
  Compute the GM value for the voxel through the GM buffer.
  Compute the voxel value through the color LUT transfer function.
  Compute the voxel opacity value through the opacity transfer function.
 Normalize the GM value and opacity value as the final opacity alpha value.
  Blend the alpha value with voxel value to composite the final voxel color value.
 End Inner Loop.
End Outer Loop.
```

Figure 3. Pseudocode for Java3D texture-based volume compositing.

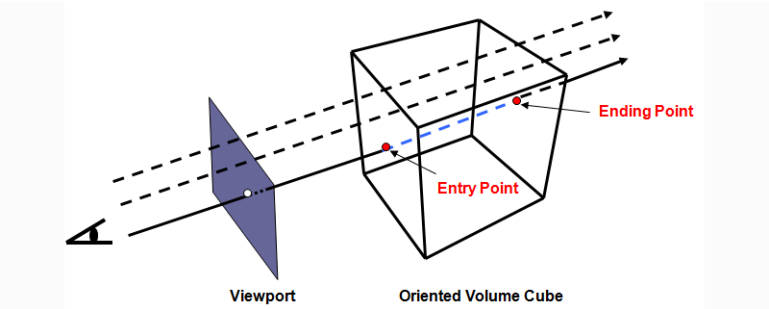


Figure 4. Raycasting with bounding box intersection.

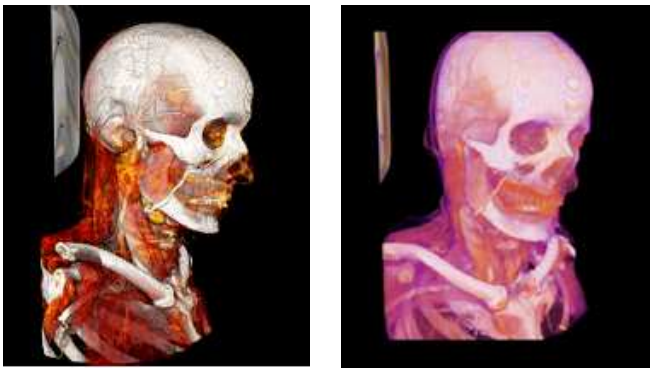
Raycast volume rendering uses the classic raycast sampling and compositing algorithm to render the volume. Parallel rays are cast from the view point through the image plane and intersect with each volume voxel. At each sampling point, the voxel color is tri-linear interpolated by performing a weighed-blending of neighboring voxels values. Rays are traced through the 3-D volume in its current orientation as determined by the oriented bounding box (Figure 4) and the rendered image is initialized to the background color. A pixel is overwritten only when a ray intersects the bounding box of the volume. Ray tracing is conducted on the line segment that interacts with the oriented bounding box. Pseudocode for the ray tracing based compositing is given in Figure 5. Java3D based volume rendering explores the high quality rendering result as shown in Figure 6.

Both Java3D-based techniques initially process the volume data entirely on the CPU before transferring them to the GPU where they are saved into texture memory. The primary issues with this rendering framework include artifacts in the displayed rendering and impediments to interactive use. Changes to the color look-up table (LUT) and opacity functions are processed on the CPU, making it necessary to regenerate and reload the texture after each change. Once the texture is generated and loaded onto the graphics card, the volume renders interactively.

Java3D-based volume rendering suffers from internal texture memory leaks due to bugs in the Java3D library.

```
Tracing the ray from front to back of intersection points:
  Interpolate the voxel's color RGB value from the 8 neighboring voxels.
  Interpolate the alpha value from the 8 corners neighbors.
  Stop tracing when the interpolated alpha value is fully opaque.
  Proceed until an intersection point at the back of the volume is reached.
Trace the ray from back to front of the intersection points.
  Blend the alpha value with the voxel RGB color value.
```

Figure 5. Pseudocode for the ray tracing algorithm.



(a) (b)

Figure 6. Java3D volume rendering. (a) Texture-based volume and (b) raycasting volume.

For example, the ImageComponent3D component replicates the texture memory during runtime execution. This drawback becomes a significant limitation for Java3D-based volume rendering applications when attempting to display large image datasets, due to the limited availability of texture memory on most systems.

An additional limitation of the Java3D framework is the limited access to the OpenGL API and the GPU shader. Versions of the Java3D library prior to 1.4 restricted access to the low-level hardware rendering pipeline. The 1.4 release added support for access to vertex and fragment shaders on the GPU. The currently supported shading languages are Cg and GLSL. The Java3D release notes indicate plans in future releases to grant access to OpenGL methods from applications that use the Java3D API. Such access would greatly increase the power of Java3D applications to directly manage the rendering pipeline.

3. WILDMAGIC GPU (CG) BASED VOLUME RENDERING

Since Java3D-based volume rendering consumes significant amounts of internal memory, and its use constrains the image data size that can be rendered, therefore necessitating the development of new methodologies to overcome this bottleneck. A Java OpenGL based solution is the key to resolving the problem. Java binding to OpenGL (Jogl) provides direct access to the rendering pipeline and empowers the programmer to directly control how the graphics are rendered. Additionally, GPU-based programmable shaders produce very efficient rendering results through the use of extensive graphics hardware optimization. WildMagic[3] is a C++ game engine recently ported to Java. It provides features ranging from low-level geometry routines to a complete high-level scene-graph and application library. It was ported to Java specifically for the shader-based rendering pipeline and shader-effects library. WildMagic is optimized for fast and efficient use of GPU memory by sharing texture and shader data.

3.1 WildMagic Library

Wild Magic library (Figure 7) layered between the Medical Image Processing, Analysis and Visualization (MIPAV) application[4] and the Jogl library to implement ray-cast based volume rendering. The two classes GPUVolumeRender and VolumeShaderEffect in the MIPAV source tree are the two workhorses of the library. GPUVolumeRender applies a VolumeShaderEffect to the proxy geometry. The VolumeShaderEffect contains the shader programs, texture images and shader related parameters for producing the ray-traced images.

3.2 Volume Rendering with WildMagic

Volume rendering is performed with a Cg fragment shader similar to the one described in Fernando et al.[5] Use of the Cg runtime library permits automatic user-interface generation based on the shading parameters. The WildMagic library provides an automatic link between the Java interface and the shader program inputs. Dynamic shader editing and re-loading are also provided by WildMagic. The WildMagic rendering pipeline for the GPU-based raycast volume is shown in Figure 8.

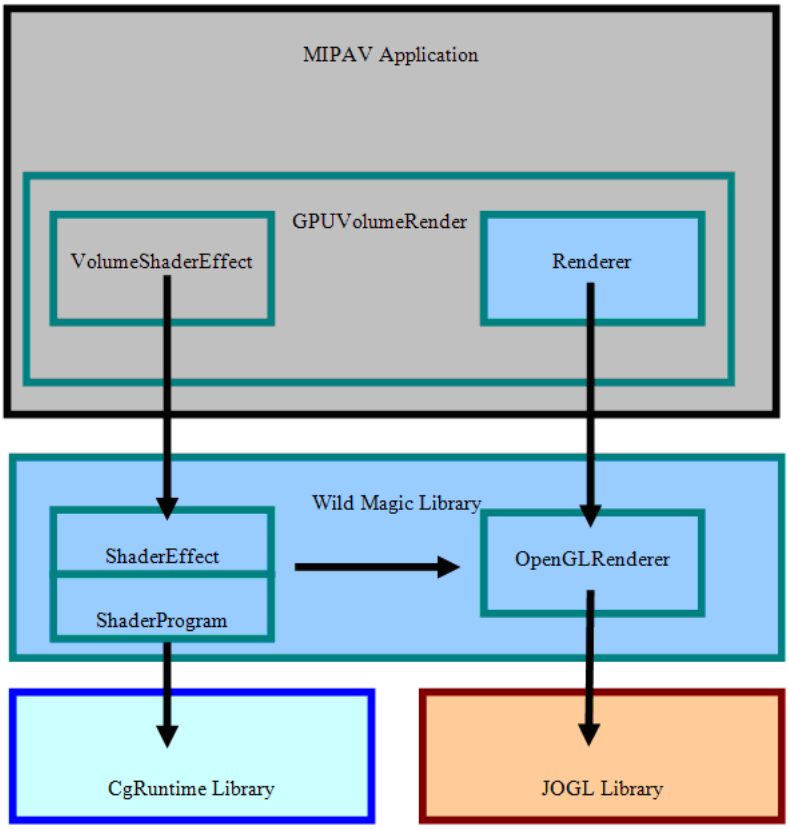


Figure 7. WildMagic library layers.

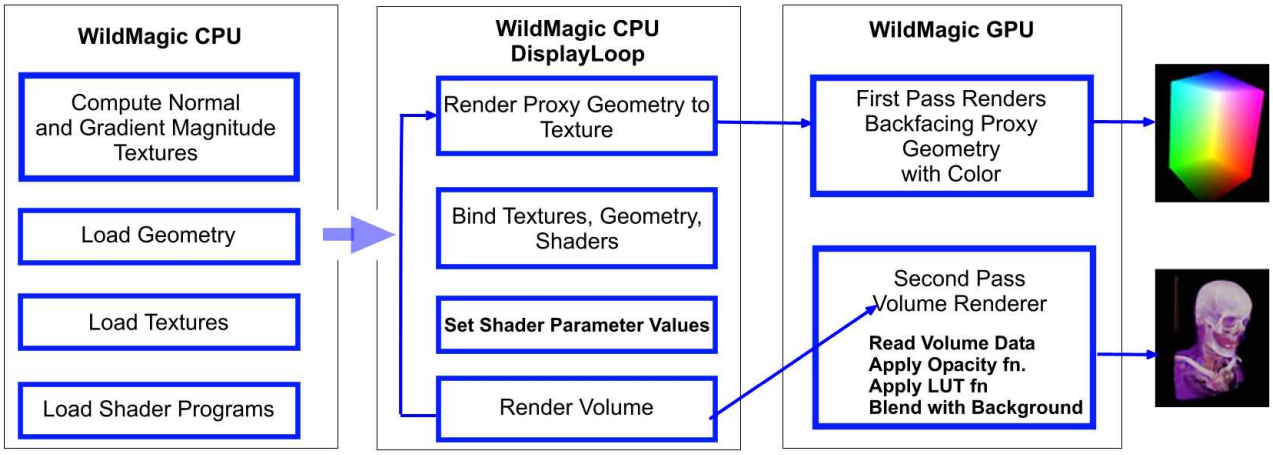


Figure 8. WildMagic rendering pipeline for the GPU-based raycast volume.

```
Determine the end points of the ray through the volume in texture coordinates.
  If user-defined clipping is active:
    Clip against the axis-aligned, view-aligned and arbitrary clip planes.
  For each sample along the ray:
    Use texture3D to read the volume data.
    Use texture1D to apply the opacity transfer function.
    If the opacity is > 0:
       Use texture1D to apply the color LUT.
       Accumulate color and opacity.
   Stopping criteria depends on the volume-rendering mode.
   Blend the resulting pixel value with the background color.
```

Figure 9. Pseudocode of the WildMagic fragment shader.

During every rendering pass, a cube proxy geometry is rendered twice. The first rendering pass renders the cube to an off-screen buffer, with the vertex colors set equal to the 3-D position on a unit cube and all front-facing polygons removed. The 3-D position is passed in as a texture coordinate. The result is saved in a texture image, SceneImage, which shows the back faces of the cube, where the color represents the 3-D position of that 'pixel' on the cube faces.

The SceneImage texture is passed to the volume fragment shader on the second rendering pass. The front-facing polygons of the cube are rendered and the 3-D position for each pixel calculated via texture-coordinates in the vertex-shader. The pixel-shader thus has both the texture-coordinates of the front-facing polygons and the back-facing polygons and can calculate a ray through the volume per pixel.

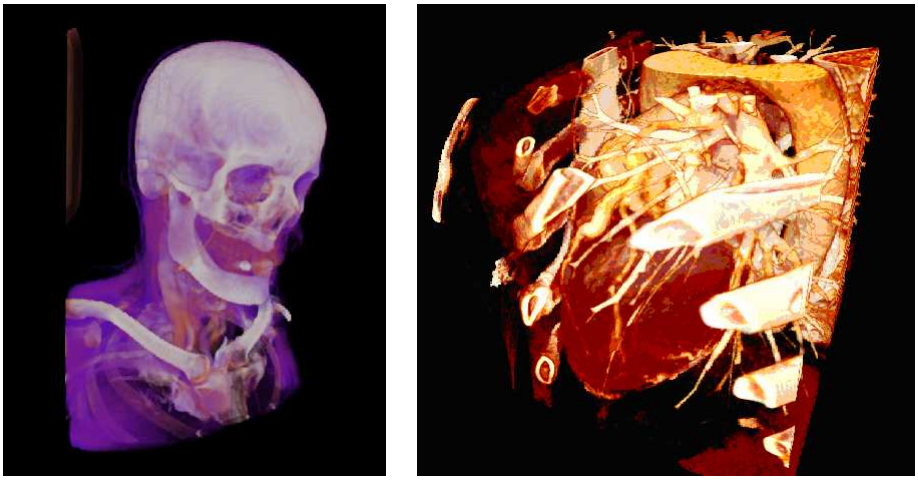
3.3 Raycasting on GPU Shaders

VolumeShaderEffect class manages several Cg shaders for volume rendering. Each of the different volume modes - maximum intensity projection (MIP), digitally reconstructed radiography (DDR), composite, surface and composite surface are implemented with different Cg shaders. The volume data, lookup table, opacity map and gradient magnitude map are stored and passed to the shader as texture images. Below, we discuss the shader programs involved in the volume raycasting on a GPU.

The vertex shader program is the same for all volume rendering modes. The input position is transformed from model space to clip space and the texture coordinates are passed to the fragment shader unmodified. All the work for raycasting the volume is done in the fragment shader. The fragment shader programs for the different volume modes follow the same algorithm. The pseudocode for the fragment shader is presented in Figure 9.

The main difference between fragment shaders for the different volume modes is in how the color is accumulated along the ray and in determining when raytracing should terminate. When applicable, early termination of ray tracing loop improves rendering performance. The Surface and Composite Surface shaders use lighting in computing the color along the ray, whereas the MIP shader determines the maximum voxel value and does not accumulate color along the ray.

The WildMagic implementation provides several improvements over the Java3D implementation. It consumes less memory on both the CPU and on the GPU. On the GPU textures are shared across all objects and renderers that access them, minimizing the use of texture-memory. In the WildMagic implementation the opacity and LUT transfer functions are applied to the data in the fragment shader. When the user modifies the transfer function the updated 1-D texture is sent to the GPU and the changes are applied interactively as the user moves the mouse. The user can see how the changes are applied to the image directly, making it simpler to achieving the desired image. In the Java3D implementation all opacity and LUT transfer function changes are done after the user releases the mouse, so the user must adjust the image without interactive visual feedback. Once the opacity and LUT transfer function changes are applied to the volume data and the 3-D texture regenerated, the Java3D version runs at 60 frames per second (FPS), faster than the WildMagic implementation which runs



(a) (b)

Figure 10. WildMagic GPU based Volume Rendering. (a) Composite rendering of head dataset and (b) composite rendering of heart dataset.

at approximately 20 FPS. The WildMagic version is significantly faster than the Java3D CPU-based raycasting implementation, however, which takes more than a second to produce the full-resolution image.

The rendering performance for the WildMagic implementation is largely independent of the size of the data volume, a benefit as there is no need to down-sample the volume data for rendering. The rendering performance depends directly on the number of pixels rendered to the screen, this is due to the fact that the raycasting is implemented as a fragment shader.

As the image size and the number of pixels rendered to the screen increases, the frame rates decrease. However, it is also true that as the image size increases the rendering quality also increases. A slower frame rate comes with the benefit of higher image quality, something that cannot be said of the non-GPU-based techniques. Figure 10 presents example visualizations, which render interactively.

4. JOGL MULTI-HISTOGRAM BASED VOLUME RENDERING

We implemented the multi-histogram volume rendering method described by Kniss. et. al[67] using the Jogl library. It's a purely CPU based approach that uses a multi-dimensional transfer function to control the opacity, boundary rendering, and color. Transfer functions map optical properties, such as the opacity LUT and color LUT to the volumetric voxel data. The quality of the final rendered image is highly dependent on how well the transfer function discriminates the optical features of interest. Traditional one-dimensional (1-D) transfer function has difficulty isolating the different features of interest due to the fact the region-of-interest (ROI) and non-ROI regions may contain the same range of data values. Typically, small changes in the 1-D transfer function can often result in large and unintuitive changes in the rendering.

The multi histogram method is based on data value, gradient magnitude, and second order directional derivatives, which facilitate improved discrimination between different material properties and boundaries. The gradient magnitude (Equation 1) depicts the local rate of change in the scalar field. The second order directional derivative (Equation 2) along the gradient direction involves the Hessian (H), a matrix of second partial derivatives.

$$f' = \| \bigtriangledown f \| \tag{1}$$

$$f'' = \frac{1}{\| \bigtriangledown f \|^2} \bigtriangledown f^T H f \bigtriangledown f \tag{2}$$

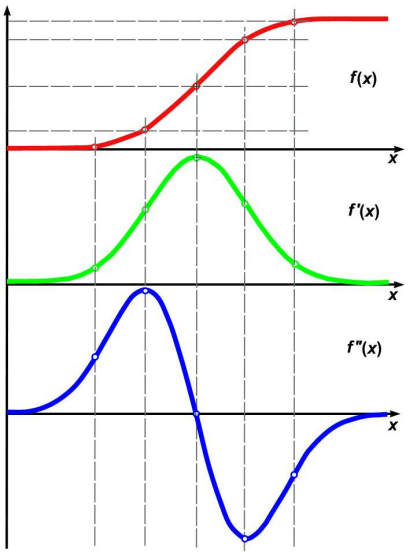


Figure 11. Gradient magnitude and second order derivatives.

At the boundary, the gradient magnitude (f') is high and the second order directional derivative (f'') is zero (zero-crossing along the second derivative measure) as shown in Figure 11. Thus, the material boundary is enhanced more than the rest of the material in the generated volume rendering.

Figure 12 demonstrates how the multi-histogram transfer function maps the traditional 1-D histogram to the log-scale 2-D histogram. The 2-D log scale histogram is shown as the magnified view on the bottom right. The horizontal axis is the data intensity distribution. The vertical axis is the gradient magnitude (f') distribution. After the 2-D log-scale histogram operation, the homogenous material (A, B, C, D regions in 1-D Histogram) is scaled down with its gradient magnitude, as shown in the oval regions (A, B, C, D) at the bottom of the multi-histogram.

The boundary between materials is scaled up, shown as the top bar of each triangle widget near the arches (E, F, G, H) of the multi-histogram. By reducing the opacity assigned to non-zero first derivative values (the color region inside each triangle widget) the material boundary can be rendered in isolation. The multi-histogram approach improves the discrimination of boundary regions.

The Jogl based multi-histogram rendering pipeline (Figure 13) generates the gradient and normal volumes from the RAW data. The volume is rendered by sampling voxels at regularly spaced intervals, then mapping the data values through the multi-dimensional color and opacity transfer functions to a color, which is rendered to the screen. The Jogl rendering context implements the 3-D texture-based view aligned rendering technique. Each slice in the volume is drawn as a polygon with multi-texture mapping. Then, Jogl based OpenGL register combiners compute the RGB channel and Alpha channel, and the RGB channel is blended using a Phong based shading module. Finally, Jogl is used to blend the alpha value and color value into the screen frame buffer. The multi-histogram method adds interactive volumetric shading to the traditional 3-D texture-based volume rendering pipeline. Figure 14 presents the pseudocode for the texture-based multi-histogram rendering. Figure 15 shows the multi-histogram volume rendering result.

Jogl multi-histogram based rendering and Java3D based volume rendering are both CPU-based 3-D texture rendering techniques. Java3D rendering is an image scene graph approach, while the multi-histogram approach uses Jogl OpenGL primitives. By comparing the rendering quality of Figure 15 and Figure 6a, it is apparent that the multi-histogram approach yields a higher quality result due to enhanced boundary classification. However, the multi-histogram approach results in a rendering which resembles a set of iso-surfaces, each with a homogeneous appearance. The Java3D approach, on the other hand, may show improved feature classification of heterogeneous tissues internal to a material boundary.

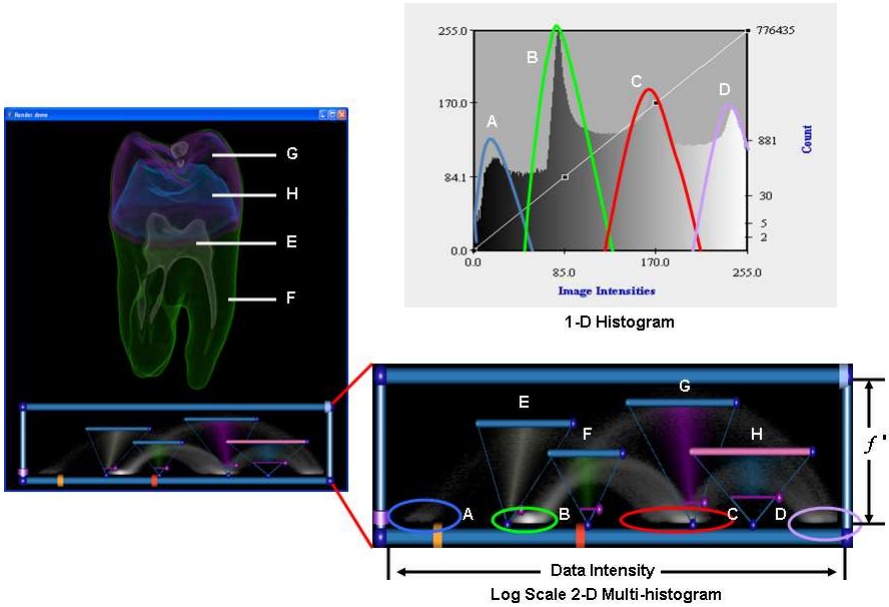


Figure 12. A demonstration of the mapping of the 1-D histogram to the log scale 2-D multi-histogram.

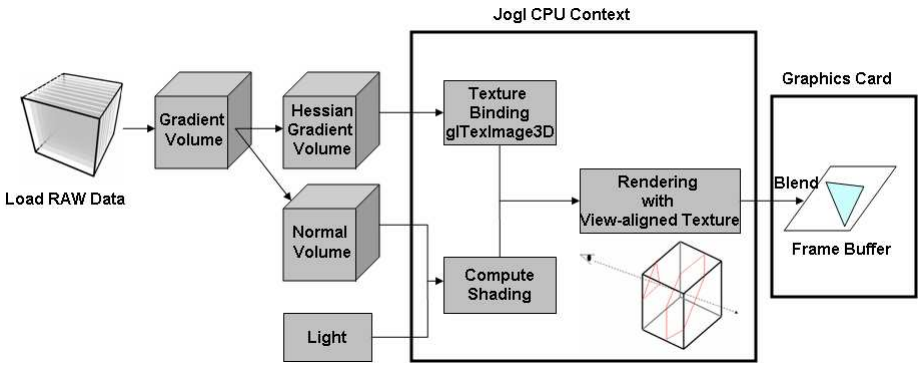


Figure 13. Jogl multi-histogram CPU-based rendering pipeline.

```
Transform the volume bounding box into view coordinates.
For each view-aligned slice in front to back order:
    Test the slice plane intersection with the edges of the bounding box.
    Compute the center of the intersected polygon.
    Tessellate the polygon into triangles.
    Draw the slice composited from the triangles, texture map it with multi-texture.
```

Figure 14. Pseudocode for view-aligned rendering.

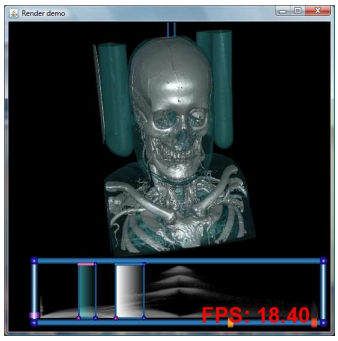


Figure 15. Jogl multi-histogram volume rendering of a head dataset.

5. JOGL GPU BASED VOLUME RENDERING

Jogl GPU OpenGL Shading Language (GLSL) based volume rendering leverages the pixel-parallelism on programmable graphics hardware. We implemented the Stegmaier et. al[8] GPU based single pass volume rendering method using Jogl and the GLSL shader with OpenGL extension.

Our implementation maps the single passing raycasting algorithm onto the programmable graphics hardware. For each pixel of the final image each single ray is traced independently through the volume. On each fragment, sampling along the ray path and accumulating the intensity and opacity are performed. The gradient magnitude filter is pre-computed using a central gradient difference or Sobel operator. All gradient components are quantized to 8-bits and stored in the RGB component of the RGBA texture already holding the scalar volume data. Gradient computation poses a significant texture memory consumption overhead, which constrains the volume rendering with larger datasets. If we were to separate the gradient pre-computing from the volume rendering procedure and load and store the gradient file on the fly, the rendering performance would improve dramatically with larger datasets.

The Jogl GLSL based volume rendering pipeline is shown in Figure 16. The Jogl CPU context loads the background data, 3-D texture data and NV fragment programs (written in assembly, specifically targeted for NVIDIA graphics cards) and binds it accordingly with OpenGL primitives. Next, the Jogl CPU-side context sends the volume data and transfer function to the GPU. The shader (NV_fragment_program2) on the GPU applies a traditional raycasting algorithm to composite the 3-D volume from front to back.

First, the interaction of the viewing ray with the volume bounding box is determined. In each step of the tracing loop the actual data value for the current sample point is fetched from the 3-D texture map and the accumulated color and opacity values for the fragment are updated.

The color computation is performed via the texture lookup, combined with alpha-blending from the 1-D transfer function. The ray terminates when a large accumulated opacity value or iso-surface threshold value is encountered. Early ray termination and empty space skipping are also used to accelerate the rendering performance on the GPU. Pseudocode for single-pass volume rendering is given in Figure 17. Figure 18 shows the Jogl GPU-based volume rendering example images.

The Jogl GLSL volume renderer, like the WildMagic GPU framework, performs volume raycasting on the GPU to generate the displayed volume. The Jogl GPU renderer, however, is more closely tied to NVIDIA graphics cards, because of the use of the NV_fragment_program2 shaders, written in assembly language. The nature of the assembly language shaders results in fast rendering of the data with a tradeoff of reduced portability. Since both rendering systems rely on calculations to be done on the GPU, the size of the datasets that they can render is limited by the amount of texture memory available on the user's GPU.

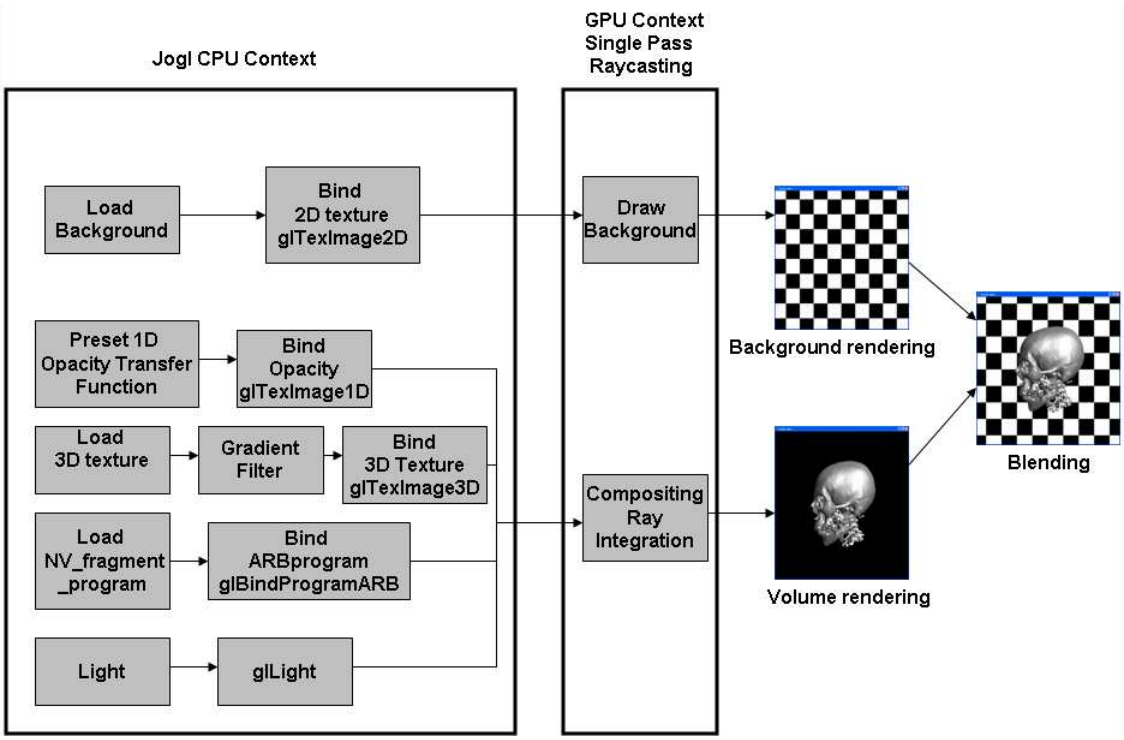
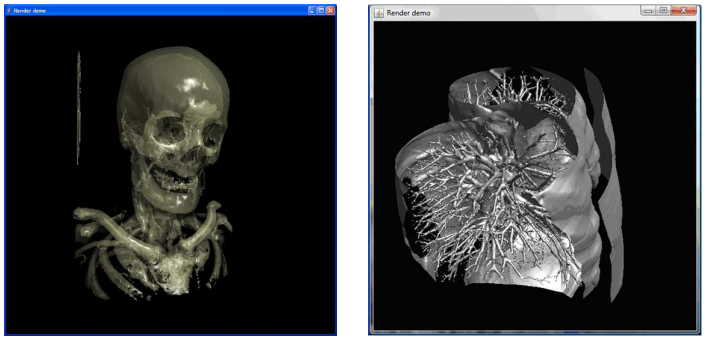


Figure 16. Jogl GPU-based single-pass raycasting pipeline.

```
Compute volume entry position.
  Compute ray of sight direction.
  Interpolate the alpha value from the 8 corners neighbors.
  While in volume:
     Lookup data value a ray position.
     Accumulate color and opacity.
     Advance along the ray.
```

Figure 17. Jogl GPU shader fragment pseudocode.



(a) (b)

Figure 18. Jogl GLSL based volume rendering. (a) Composite volume rendering and (b) iso-surface rendering.

6. PERFORMANCE METRICS

All performance measurements were conducted on PC with the NVIDIA GeForce 8800 Ultra graphics card. Listed below are the hardware specification (Table 1) and software specification (Table 2) used to test against each renderer.

Table 1. Hardware specification

PC	Dell Precision 690
Processor	Intel CPU x5355 Single Processor (4 quantra cores) 2.66 GHz (64 bit)
Memory	4 GB
Graphics card	NVIDIA GeFroce 8800 Ultra
Java VM	3 GB

Table 2. Software library specification

Software Package	Version
JDK	1.6.0_03 (64 bit)
Jogl	1.1.1 (64 bit)
Java3D	1.5.1 (64 bit)
Cg Toolkit	2.0 (64 bit)

We use the following metrics to compare the performance of different Java based volume rendering methods. For each measurement, we captured the average frame rate and memory usage when each method renders the volume with the best rendering state. Rendering performance was measured by an ad-hoc approach. A frame counter computes the frame rate at fixed time intervals. The frames per second were calculated by computing the total number of frames rendered divided by the sampling time. We measured this performance when the volume rotated, since this forces each frame to update.

Memory consumption was captured by polling the Java virtual machine. We used image volumes containing byte data with different volume sizes to run each renderer and collect the quantitative results. All of the renderings are projected to an 812×812 viewport.

7. RESULTS

Table 3 and 4 show the quantitative data collected for the frame per second and memory consumption metrics. Figure 19 shows the volume rendering quality of an abdomen dataset. Additionally, the first example image in each of the above sections explores the rendering quality of the head dataset. All the rendered images are generated using each system's composite rendering mode, except for Figure 19f.

We evaluated the rendering performance using four factors, frames per second, memory consumption, rendering quality, and rendering feature set. By comparing the quantitative data, we observe that the Jogl GLSL rendering method yields the best result with a relatively high frame rate and the lowest memory consumption. When comparing the image rendering quality, the WildMagic approach shows the best rendering and has better color classification. Comparing renderer features, the Java3D and WildMagic approaches provide more flexibility in rendering types, such as the MIP, DRR, composite, surface and composite-surface modes. The Jogl, GLSL renderer only features composite, iso-surface and MIP rendering modes, while the Jogl multi-histogram system only has a composite rendering mode.

Table 3. Quantitative data collected for frame per second

Frames Per Second (FPS)	Java3D (Texture)	Java3D (Raycast)	Multi-histo	WildMagic	Jogl (GLSL)
Heart($128^2 \times 128 \times 1$ byte)	60	9.6	58	19	27
Abdomen($256^2 \times 256 \times 1$ byte)	60	2.7	18	27	41
Stent($512^2 \times 128 \times 1$ byte)	N/A	N/A	9	22	25
Cardiac($512^2 \times 256 \times 1$ byte)	N/A	N/A	7	30	1.26
Head($512^2 \times 460 \times 1$ byte)	N/A	N/A	0.63	N/A	N/A

Table 4. Memory consumption for each visualization framework

Memory (MB)	Java3D (Texture)	Java3D (Raycating)	Multi-histo	WildMagic	Jogl (GLSL)
Heart($128^2 \times 128 \times 1$ byte)	135	151	15	18	6
Abdomen($256^2 \times 256 \times 1$ byte)	968	1188	182	80	20
Stent($512^2 \times 128 \times 1$ byte)	N/A	N/A	356	153	36
Cardiac($512^2 \times 256 \times 1$ byte)	N/A	N/A	708	296	68
Head($512^2 \times 460 \times 1$ byte)	N/A	N/A	1297	N/A	N/A

By comparing all the four factors, we consider the WildMagic GPU-based rendering as the best framework among the four renderers. The frame rate is between 19 and 30 for the different size datasets used in this paper. The memory usage is relatively low, since the gradient and normal computations only exceed the Java virtual machine memory limit with larger datasets.

The Java3D texture and raycast based renderers yield resonable quality renderings with acceptable feature classification. However, aliasing still exists as shown in Figure 6 (the under-sampling aliasing near the top of the skull). The main drawback for Java3D rendering is the redundant memory consumption. The Java3D library contains some internal memory leaks, such as the ImageComponent3D (Java3D API), which replicates the texture memory internally. This defect becomes a significant hurdle for our Java3D-based implementation.

The WildMagic GPU rendering framework has the flexibility to mimic the Java3D rendering features and implement the raycasting algorithm on the GPU. The rendering speed, quality and memory usage improve dramatically compared to the Java3D approach.

For example, as demonstrated in the head dataset (Figure 8) aliasing artifacts no longer appear near the top of the skull. The only weakness of the WildMagic approach is the memory consumption during the gradient and normal calculation. Currently, the volume gradients and normals are pre-computed by the CPU immediately followed by the rendering procedure. Separating the calculation of the gradients and normals from the rendering procedure will enable the display of larger image datasets.

As compared to the Java3D and WildMagic 1-D transfer function histogram, the Jogl multi-histogram approach provides a unique way to control the histogram classification. The multi-histogram based method makes the transfer function manipulation more intuitive. The rendering quality is also acceptable. However, since the whole multi-histogram method uses the CPU, the rendering speed is slow when it explores more detailed volume information.

The Jogl GLSL based method renders the volumetric data with relatively fast rendering speed and the lowest memory usage. The main drawback is the GLSL shader, which is written in assembly code and tied to NVIDIA graphics cards. Only a few rendering modes are available, such as MIP, 3D texture composite and iso-surface shaders. The Jogl GLSL composite mode volume in Figure 19 shows a peculiar rendering result. The 1-D transfer function limits the user's ability to make better classifications, resulting in the transparent iso-surface

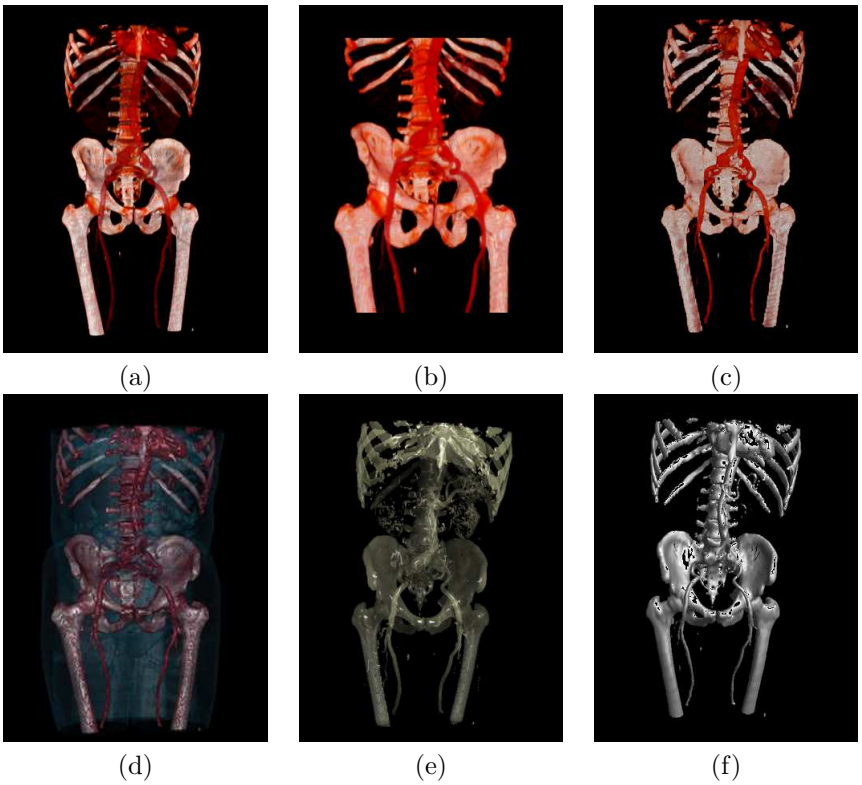


Figure 19. Volume rendering of the abdomen dataset. (a) Java3D texture rendering, (b) Java3D raycast rendering, (c) WildMagic composite rendering, (d) Jogl multi-histogram rendering, (e) Jogl GLSL composite rendering, (f) Jogl GLSL iso-surface rendering.

being visible inside the composite volume. Additionally, we illustrate the Jogl GLSL iso-surface mode with shading in Figure 19. The user can only adjust the threshold value to see the different level of details inside the volume. Early ray termination speeds up the rendering; however, this design tradeoff impacts tissue classification.

8. CONCLUSIONS

This paper has demonstrated and analyzed four innovative approaches for high-quality renderings of biomedical image datasets on commodity desktop hardware using the Java programming language. We have shown the Java based visualization libraries provide the ability for producing high quality renderings of medical image data comparable to commonly used C++ visualization libraries. Performance measures and qualitative factors have been examined to illuminate the advantages and drawbacks of each system. The WildMagic GPU-based volume renderer has been selected for future visualization development in MIPAV and an experimental version of the renderer is included in the current release of MIPAV available at `http://mipav.cit.nih.gov`.

9. FUTURE WORK

We are in the process of redesigning the MIPAV volume renderer to use the WildMagic graphics engine. WildMagic provides not only volume rendering capabilities but also a fully functional scene graph with polygon mesh surfaces, animation and a streaming system. We plan to add the two-dimensional histogram to MIPAV and to explore single-pass raycasting on the GPU further. One advantage WildMagic provides over the other renderers is the flexibility to render multiple objects in the same scene, for example polygon meshes intersecting the raycast volume data. An application of this in the current experimental WildMagic version of the MIPAV renderer includes Diffusion Tensor Visualization with polygon fiber-bundle tracts embedded in the raycast volume.

ACKNOWLEDGMENTS

We would like to give a special thanks to Olga Vovk for assisting with the preparation of figures and the formatting of this document. Also, we would like to thank Joe Kniss for providing suggestions on porting portions of his multi-histogram based volume rendering to Java.

REFERENCES

[1] Abramoff, M. and Viergever, M. A., "Computation and visualization of three-dimensional motion in the orbit," *IEEE Transactions on Medical Imaging* **21(4)**, 296–303 (2002).

[2] Mroz, L. and Hause, H., "RTVR - a flexible java library for interactive volume rendering," *IEEE Visualization* **01**, 279–286 (2001).

[3] Eberly, D. H., [*3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*], The Morgan Kaufmann Series in Interactive 3D Technology, New York, http://www.geometrictools.com/ (2006 (second edition)).

[4] MIPAV, [*Medical Image Processing, Analysis and Visualization (MIPAV)*], http://mipav.cit.nih.gov (1997). http://mipav.cit.nih.gov.

[5] Randima, F., [*GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*], Addison Wesley Professional, Reading, Mass. (2004).

[6] Kniss, J., Kindlmann, G., and Hansen, C., "Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets," *Proc. IEEE Visualization Conf.* **01**, 255–562 (2001).

[7] Kniss, J., Kindlmann, G., and Hansen, C., "Multi-dimensional transfer functions for interactive volume rendering," *IEEE Transaction on Visualization and Computer Graphics* **8(3)**, 270–285 (2002).

[8] Stegmaier, S., Strengert, M., Klein, T., and Ertl, T., "A simple and flexible volume rendering framework for graphics hardware based raycasting," in *Volume Graphics*, 187–195, Stony Brook, (New York, NY, USA) (2005).