Design Of A High Performance Volume Visualization System

Barthold Lichtenbelt*

Graphics Products Laboratory, Hewlett-Packard

Abstract

Visualizing three dimensional discrete datasets has been a topic of many research projects and papers in the past decade. We discuss the issues that come up when designing a whole computer system capable of visualizing these datasets in real time. We explain the three way chicken and egg problem and discuss Hewlett-Packard's effort at breaking it with the Voxelator API extensions to OpenGL. We enumerate what a good hardware design should accomplish. We discuss what system issues are important and show how to integrate volume visualization hardware in one of Hewlett-Packard's graphics accelerators, the VISUALIZE-48XP. We show why the Voxelator is an efficient and well designed API by explaining how various existing hardware engines will easily fit into the Voxelator framework.

CR Categories and Subject Descriptors: C.5.3 [Computer System Implementation] Microcomputers – Workstations; D.2.0 [Software Engineering] General – Standards; I.3.1 [Computer Graphics] Hardware Architecture – Graphics Processors; I.3.7 [Computer Graphics] Three-Dimensional Graphics and Realism – Raytracing.

Additional Keywords: volume rendering, visualization, volume accelerator, OpenGL, system design.

1 INTRODUCTION

Designing a hardware volume rendering engine is a non-trivial task. Many designs have been proposed [2, 4, 7, 8, 9, 10, 11, 12, 15, 16, 17], and a few either simulated or actually built [1, 4, 9, 10, 15]. Volume visualization is compute intensive and very demanding on system resources like CPU compute power, memory bandwidth and bus bandwidth. Visualizing a reasonable volume dataset of 64 Mbytes in real time is beyond today's desktop computers. Visualizing an ever bigger dataset of about 512 Mbytes in real time is beyond today's supercomputers. Therefore people have been designing, and building special purpose hardware that will accelerate the rendering of volume datasets, much beyond what a general CPU is capable of doing. This makes perfect sense because we expect that a well designed volume visualization *system* will outperform general CPU solutions by several orders of magnitude.

One of the problems we are faced with in designing a volume visualization system is system integration. Globally there are three crucial areas that need to be addressed in order to be able to produce a good volume visualization system. First, there should be hardware that accelerates the volume visualization. There are many problems associated with building this kind of hardware. We will not discuss these issues in this paper. Second, the hardware should interface to a computer. This includes the hardware interface itself, like bus issues. The physical size, power consumption, and heat dissipation of the hardware also falls in this category. Third, applications that allow the user to visualize their volume data need to talk to the hardware through some kind of software interface, or API (Application Programming Interface). An API hides hardware details from the application, which allows software developers to design an application that is portable across platforms.

These three problems need to be solved to design and build a high performance volume visualization system. To also make this system affordable makes it even harder. Hewlett-Packard's goal is to make volume visualization *pervasive*. With that we mean that every professional, who today uses a 3D workstation, will be able to afford to do volume visualization with that kind of a system.

In achieving this we see a three way chicken and egg problem that needs to be solved. Fast volume visualization hardware does not function without a good industry standard API, which does not function without applications that use that API, and these applications will not function without fast hardware. We wanted to break this circle and started at the API level. The reasons for starting there are:

- A good API outlives hardware. By hiding hardware details from the application, hardware designers have the freedom to change hardware from one generation to the next, without impacting the application. This is crucial to the wide spread acceptance of volume visualization. This means that the API has to be general enough to allow for several hardware generations, without limiting potential new developments and improvements in hardware design.

* Hewlett-Packard, 3404 East Harmony Road, Fort Collins, CO 80525, USA. barthold@fc.hp.com

1997 SIGGRAPH/Eurographics Workshop

- The API has to deal with both hardware on one end and applications on the other end. Both have specific constrains that govern the design and definition of the API.
- Without a standard API there is no incentive to develop applications. Applications need to be portable across platforms. A standard API provides the means for a high degree of portability.
- Without a standard API that is available on several different platforms it is hard to justify developing special purpose volume visualization hardware. A standard highly available API will help making volume visualization pervasive, which means that hardware development costs can be amortized over a much broader base of systems.
- The 3D texture mapping OpenGL extension is in widespread use today [1, 5, 6, 13, 20], but it only solves parts of the volume visualization problem. It only accelerates the rasterization stage of the volume pipeline. This will be explained in more detail in section 3.

The next section discusses requirements for a good volume visualization API. Section 3 describes the Voxelator API, our volume visualization extensions to OpenGL. Section 4 explains the concept of blocking. Section 5 discusses important factors that make for a good volume visualization system and discusses the integration of volume visualization hardware into one of Hewlett-Packard's graphics accelerators. Section 6 validates the Voxelator by discussing how existing hardware engines map to it. Section 7 and 8 discuss future directions and draw some conclusions.

2 API REQUIREMENTS

The API is the most essential part of a volume visualization system, since it enables the hardware and the applications using the hardware to work together as a volume visualization system. Therefore careful thought has to be given to its design. We used the following list of design goals in the design of the Voxelator API extensions to OpenGL. A good volume visualization API:

- is based on the industry standard API, OpenGL. We did not want to design a proprietary API. OpenGL is a widely adopted visualization API and is available on all major platforms.
- outlives a hardware design. Several generations of hardware should fit under the API. Therefore the API should not dictate one specific hardware implementation, as the 3D texture mapping extensions to OpenGL do.
- is extensible. That means that if new features need to make it into the API, there is a mechanism to do so. OpenGL provides a general extension mechanism to do just that. The next section will list some explicit examples of possible future extensions to the Voxelator.
- has low overhead. This means that the API does not neutralize the hardware performance by forcing significant software preprocessing. The API is a thin layer on top of hardware, just enough to hide hardware details from the applications.
- fits numerous hardware designs. It allows for some or all of its stages to be accelerated by hardware. This choice should be up to the hardware designer, not dictated by the API.
- optimally uses system resources. The critical resources in a volume visualization system are memory, memory bandwidth, bus bandwidth, acceleration hardware and the general purpose CPU. For example,. the API should not be designed so that it has to make an extra copy of the dataset in memory.
- provides enough flexibility for applications.
- implements the full volume rendering pipeline. With this we mean gradient computation, classification, lighting and shading, interpolation and compositing. The full pipeline will be discussed in detail in the next section.
- integrates well with the existing OpenGL geometry and imaging pipelines.
- is unambiguous. All stages in the API are well defined as is the order in which the stages are executed. The default values are defined.

Some of these design constraints conflict with each other, like the desired flexibility for applications and the defined order in which stages of the pipeline are executed. Applications want maximum flexibility, which means that they would like to re-define the order in which stages are executed. However, this will make hardware acceleration a great deal more complex. How we dealt with these issues is the topic of the next section.

3 THE VOXELATOR API

.OpenGL [19] is the standard visualization API at this moment, and will be in the foreseeable future. Therefore we chose to use OpenGL as the API of choice for our volume visualization system. Since volume visualization is a new field, OpenGL did not adequately address this topic yet. Figure 1 shows a high level overview of the OpenGL pipeline. The pixel pipeline and the geometry pipeline exist in OpenGL today. The images that are

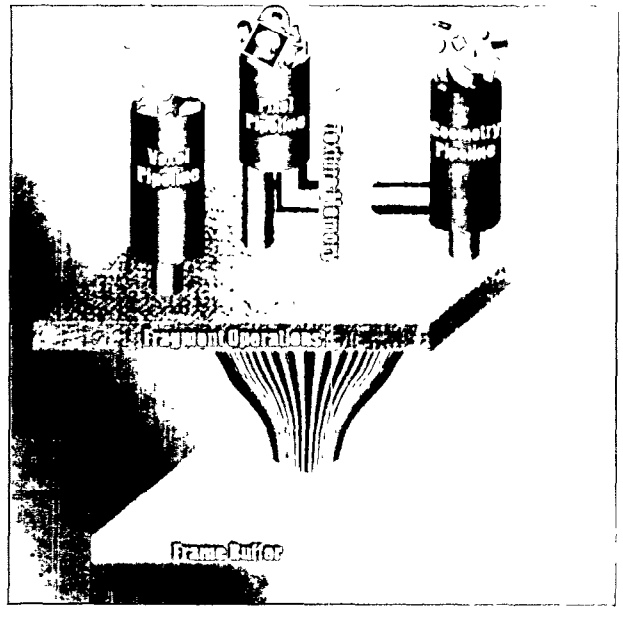

Figure 1. *The standard OpenGL pipeline plus the new voxel pipeline.*

input into the imaging pipeline either can be routed to texture memory or to the frame buffer, through the fragment operations stage. The geometry pipeline will process polygons, and optionally use the images in texture memory to texture map the polygons. Both pipelines produce *fragments*. Fragments are little data structures that contain information that is used in the fragment operations stage to decide if and how to update the frame buffer. Fragments typically contain a RGBA and a depth, or z-value. The fragment operations stage performs operations like fog, stencil test, depth buffer test, alpha test, blending and some others.

Since OpenGL does not address volume visualization other than through 3D texture mapping, we decided to explore if there is a need for a volume visualization system that does more than what you can do with 3D texture mapping today.

To get an informed opinion on this we decided to ask university and corporate laboratories, research institutes and customers about what they thought a good volume visualization system entails. We also solicited, and got, feedback by distributing the Voxelator CD-ROM at the SIGGRAPH '96 and Visualization '96 conferences. The result of the queries is a very clear answer: Yes there are things we want to do we can not do with just 3D texture mapping hardware. 3D texture mapping solves a good number of problems, but not all. Therefore we decided to add a third pipeline to OpenGL, the voxel pipeline. We refined this pipeline using the feedback we got. We show this pipeline in Figure 1.

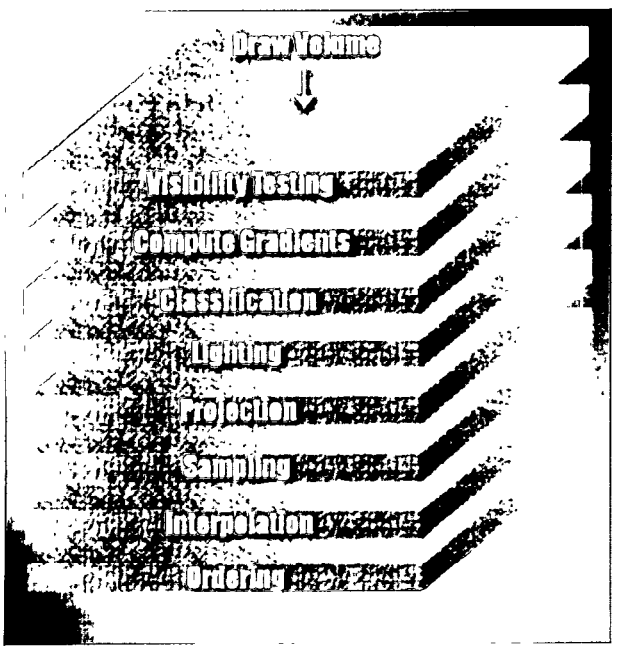

Figure 2. *The voxel pipeline.*

The voxel pipeline also produces fragments, of exactly the same format as the pixel and geometry pipelines do. Pixel, geometric and volumetric data merge at the fragment operations stage. OpenGL takes care of the problem of mixing geometric, imaging and volumetric data into one scene.

Figure 2 shows the voxel pipeline, which we call the Voxelator, in detail. The pipeline in Figure 2 is a conceptual pipeline. That means that this is the pipeline an application expects the underlying API and hardware to follow. The actual implementation of the Voxelator pipeline can be different from the one shown in Figure 2. This is up to the API and hardware designers, as long as the final rendered image is the same as the image that results if the conceptual pipeline in Figure 2 is followed.

After the application has set up the volume, rendering parameters and data format it calls glDrawVolume(). The following stages will then be performed on the data:

Visibility Testing is the stage where voxels optionally can be masked out by a bit mask supplied by the application. The opacity of each voxel that is masked will be set to zero.

Compute Gradients is the stage where the local gradient for each voxel is computed. The gradient is needed in the classification and lighting stage.

Classification is the stage where an opacity and RGB value is assigned to each voxel. The magnitude of the gradient, the voxel intensity and the index field can all be combined in a user specified way to form the input to the classification stage. The resulting value is then used as an index into a lookup table, whose output is a RGBA value. Each voxel can have an index field containing a label assigned by the application. This allows an application to do pre-segmentation on the dataset before it is rendered. The Voxelator can use that label in the rendering stage.

Lighting is the stage where the standard OpenGL lighting model is applied to the RGBA values of each voxel given the normal defined by the gradient.

Projection translates, rotates and scales the dataset using the OpenGL model, view and projection matrices as well as the viewport transform.

Sampling is the stage that determines what the distance is between two sample points, or fragments, on one ray. This allows for over and under sampling of the dataset. The application has control over this distance.

Interpolation uses an application defined interpolation method to compute the RGBA values for a sample point on a ray using the neighboring voxel values. Currently nearest neighbor and tri-linear interpolation can be specified.

Ordering determines if the fragments are generated in front-to-back or back-to-front order.

The resulting fragments are processed by the, already existing, OpenGL fragment operations stage. There fragments can be blended together, or their value tested against the previous fragment value to do Maximum Intensity Projection rendering. Finally the frame buffer will be updated after the fragment is processed.

If it is desired to have additional functionality in the Voxelator such as better interpolation filters or better gradient filters, anyone implementing the Voxelator API can use the extension mechanism of OpenGL to add that functionality to the Voxelator.

The Voxelator is not tied to one specific hardware implementation, unlike the 3D texture mapping method. In fact, the Voxelator will fit on many different hardware architectures,

including 3D texture mapping hardware. This is the topic of section 6.
The Voxelator pipeline can be divided into two sections, see Figure 2. The first five stages in the pipeline, up to the projection stage, form the first section. Everything below that forms the second section. Globally speaking the first section is responsible for the setup, transformation, classification and lighting of a dataset. The next section is responsible for the rasterization of the dataset. This is analogous to the geometry pipeline, where a

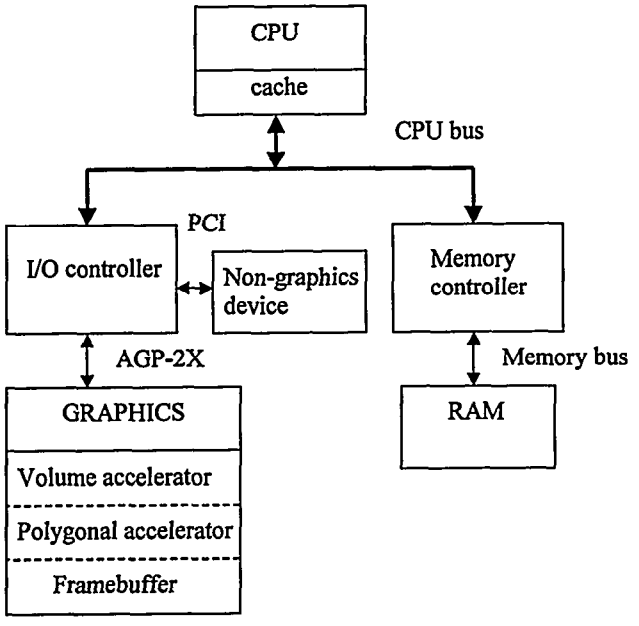

Figure 3 *General volume visualization system diagram. Several bottlenecks can be identified.*

similar division can, and often is, made. In fact, the first section is what the Voxelator adds over what you can do with the 3D texture mapping extensions to OpenGL. The Voxelator adds and allows for interactive lighting and classification, that are core functionalities of any volume visualization system. To be able to do this interactively will greatly enhance the usefulness of a volume visualization system, and thus its pervasiveness. The complete Voxelator specification can be found in [18].

4 BLOCKING

An application has to format the dataset into *blocks* before it hands the dataset to the Voxelator with glDrawVolume(). Only then is hardware acceleration guaranteed. It is still possible to render a dataset without blocking it, but this will not result in optimal rendering performance.
Blocks are sub volumes of the dataset. The size of the block is dictated by the volume visualization hardware, through the Voxelator API. The application calls glGet*() to find out what this size is. Each block has to be stored linearly in memory. One of the parameters to glDrawVolume() is a list of pointers to the blocks.
Each block has a so called *action* assigned to it. An action can be any of the following three attributes: render compressed, render uncompressed, or skip. A block can be stored in memory compressed, or non compressed. The Voxelator will decompress a compressed block right after the visibility test stage. Setting the action to 'skip' means that the Voxelator will not process that block at all. This can be used to render only part of a dataset.

Having the dataset formatted into blocks has several advantages to the overall performance of the visualization system. A block is in the optimal size for a hardware volume visualization accelerator, since the block size is dictated by the hardware. For example a hardware accelerator might only have a small on-chip cache in which to store a block, or it might have off-chip fast memory which will hold blocks of a much bigger size.
Since a block is stored linearly in the system's main memory, transferring the block over the system bus to the hardware accelerator will be processor cache efficient. If any pre-processing has to be done by the Voxelator API, which is the interface between the application and the hardware, the data will be transferred to the processor cache, processed by the CPU and transferred to the hardware. A block stored linearly in memory will prevent cache trashing.
In many computer systems data that is transferred over the system bus to any device on that bus will be loaded into the processor cache. Data can be sent by either programmed I/O or DMA to the device. Each of these I/O models has, depending on the computer system hardware, an optimal data size in terms of maximum bus bandwidth. The block size can be tuned to optimize the bus transfer rates. See also section 5 and Figure 3.
Blocks can be marked as being empty, not contributing to the rendering because the opacities of all the voxels in the block are zero, or close to zero. The application can mark a block by setting it's action attribute to 'skip'. The combination of Voxelator API and underlying hardware however could also keep an internal list of blocks that are empty and use that information for performance enhancements.
Exposing the blocking issues to the application allows for a Voxelator API implementation that does not have to make a copy of the dataset 'under the covers' in main memory. Current OpenGL 3D texture mapping implementations will make a copy of the dataset. This copy is formatted into a format suitable for the underlying 3D texture mapping hardware. The application will have it's own copy of the data stored in main memory and OpenGL will have it's own, specially formatted, copy of the same data in main memory. This can be a problem for application developers. Datasets can be very big. Having an extra copy around in memory means the system could run out of memory and start swapping, which results in a severe rendering performance penalty.
Blocks will also benefit application performance. Applications almost always want to pre-process a dataset, before it is rendered. Segmentation of a dataset is one example. As indicated before, blocks allow for efficient cache behavior, which will improve an application's pre-processing performance.
The API software driving the visualization system should have low overhead and has to be designed carefully to achieve just that. This is why we chose to expose blocking to the hardware and to the application.

We provide glu utilities to convert a dataset into blocks and to convert blocks from one size to another.

5 A COMPLETE SYSTEM

As we stated before, designing hardware that accelerates volume visualization is only part of the problem. The acceleration hardware generates fragments or maybe pixels, which need to be processed and written to a frame buffer for display. If it is desirable that the output of the volume accelerator can be rendered and properly occluded into a scene with geometric primitives, it will need to interface to the geometry hardware. It needs to get its data out of main memory by some I/O mechanism.

Then there is the memory bus and the memory system itself. The memory system should be able to supply the CPU and I/O devices with data at a high enough rate to keep them continuously busy. The I/O devices, like the graphics accelerator, use the AGP or PCI bus. The PCI bus, since it has a lower bandwidth and can serve multiple I/O devices, is used for the not so data hungry devices, like a network card. The AGP-2X bus is specifically designed for a high speed peer to peer connection. This means that only one device can use the AGP bus. This typically will be a graphics device, since these have the highest bandwidth requirements. A well balanced volume visualization system is designed so that all busses, memory system, graphics system, general CPU processing power and processor cache are optimally tuned to each other.

We will now take a closer look at the graphics device in Figure 4,

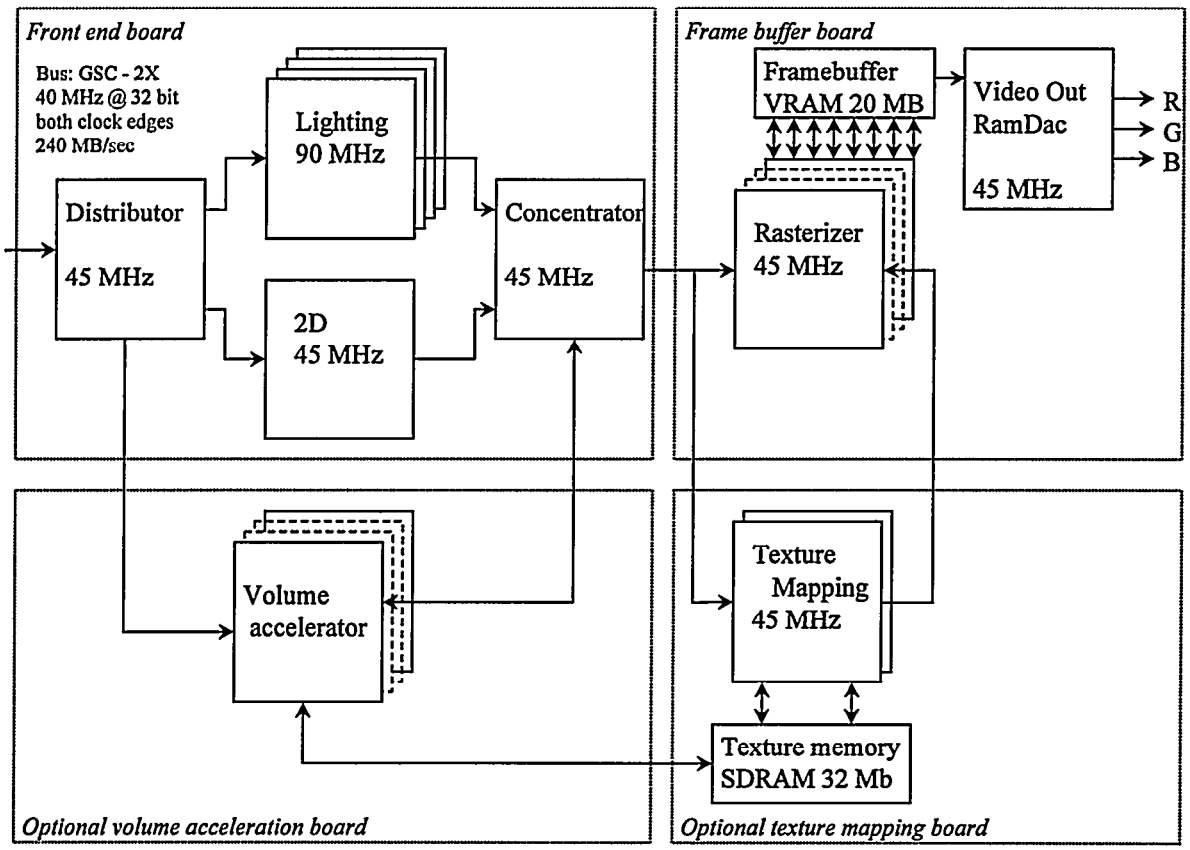

Figure 4 *Hewlett-Packard's Visualize-48XP geometry accelerator with proposed volume acceleration board*

Several researchers have pointed out [3, 12, 16] that getting the dataset out of memory at rates needed for interactive volume visualization is one of the bottlenecks of their hardware system. If we look at Figure 3 we identify several potential bottlenecks in a volume visualization system. First of all, there is the CPU bus. All data goes over this bus. Depending on the I/O model, programmed I/O or DMA, data might be fetched from memory, transferred over the CPU bus into the processor cache, written to the I/O device by the CPU over the CPU bus, or transferred directly from memory to the I/O device without CPU interference.

and more specifically at the integration issues of a volume accelerator with conventional geometric acceleration hardware. We envision a volume visualization chip that has the following properties:

- Accelerates the full Voxelator pipeline.
- Is scaleable. Adding more chips will mean that performance scales linearly.
- Processes blocks, as defined by the Voxelator.
- Has an external 16 bit lookup table for the classification stage. The classification lookup table is defined by the

Voxelator as having at least 2^16 entries. A bigger table is allowed and that choice is up to the hardware designer.

- Interfaces to some fast RAM which acts as a block cache. Block misses will initiate a transfer out of main memory.
- Generates fragments as output, which then can be blended into the frame buffer.
- Will be able to process at least 200 Million voxels per second per chipset.

Internal studies at Hewlett-Packard have shown that these goals are feasible. The block cache is extremely useful and will greatly enhance the performance of the volume accelerator. In order to process one block, the volume accelerator needs access to all 26 neighboring blocks for gradient computation and interpolation. Thus it makes sense to transfer at least 27 blocks at once from main memory to the graphics system and store them in the block cache.

Having the volume accelerator chipset generate fragments allows for correct mixing with opaque data already in the frame buffer. In order to guarantee correct rendering with opaque geometry data the application has to render its geometry data first, then render the volume. Each sample point on the ray cast through the volume is a fragment, which has a z-value. Doing a z-compare with the geometry data in the frame buffer guarantees correct blending and occlusion. Another design option would be to have an accumulation buffer closely coupled to the volume accelerator. The volume accelerator then composites the samples on a ray into the accumulation buffer. This alleviates the problem of the high bandwidth needed to and from the frame buffer in the former solution. However, proper mixing with geometric data is no longer possible, since the volume dataset is already blended into a 2D image in the accumulation buffer, before being transferred to the frame buffer.

Figure 4 shows a block diagram of one of Hewlett-Packard's current geometry accelerator solutions, the VISUALIZE-48XP. It also shows how to take that solution and add a volume accelerator to it.

The VISUALIZE-48XP is a three board system. It physically plugs into the GSC bus, which is Hewlett-Packard's proprietary bus. The GSC-2X is a 40 MHz bus with a 32 bits wide data path. Data is transferred on both clock edges, resulting in a sustained bandwidth of about 240 Mb/sec.

The distributor chip is the gateway into the system. It decides where data goes, to the 2D stage or the lighting stage. 2D data, typically X11 primitives, are accelerated by the 2D chip.

Geometric primitives are assigned to one of the four lighting chips by the distributor chip. Each lighting chip accelerates the floating point intensive operations like: Geometric transformations, lighting, depth cueing and clipping calculations. This chip heavily leverages the floating point units of the PA-RISC processor line.

The concentrator chip combines the output streams from the lighting chips as well as the output from the 2D chip. Floating point values are converted to fixed point and sent on to the rasterization and texture mapping chips.

The 10 rasterizer chips combine the functionality of a scan converter and frame buffer controller. The VISUALIZE-48XP employs a unified frame buffer architecture, which means that the same memory array stores image and overlay planes as well as z values. Screen space parallelism is used in the rasterization stage.

The two texture mapping chips look up texture values from the texture RAM which is managed as a cache. Interrupts are used to fetch texture cache blocks from the system's main memory. A separate port on the texture mapping chip is used to load texture RAM. This port bypasses the rest of the rendering pipeline, so that texture cache misses can be serviced while the rendering pipe is busy. This architecture allows the texture size to be limited only by the system's main memory size instead of the size of the local texture memory. The total texture memory is 32 Mb. However the VISUALIZE-48XP stores textures twice, to increase performance. Effectively this leaves 16 Mb of texture storage.

The total system is capable of rendering 3.9 Million triangles per second. These are 50 pixel, Gouraud shaded and z-buffered triangle strips. The VISUALIZE-48XP combined with the C180 workstation is a well balanced system with respect to the issues discussed earlier and shown in Figure 3.

The best way to integrate a volume accelerator into this system is also shown in Figure 4. The texture memory is re-used as local block storage and acts as a block cache. An interrupt is generated if a block miss occurs and the missing blocks are transferred out of main memory into the texture memory. The volume accelerator chips will generate fragments, which are passed on to the rasterizer section to be blended into the frame buffer. Some external memory is needed for the classification lookup table. Depending on the clock speed of the volume accelerator chips the GSC-2X bus will be the first performance bottleneck in the complete system, meaning that the performance for this system would top out at 250 million voxels per second.

6 MAPPING THE VOXELATOR

We carefully designed the Voxelator API so that it would not dictate how the hardware, that accelerates the Voxelator, should

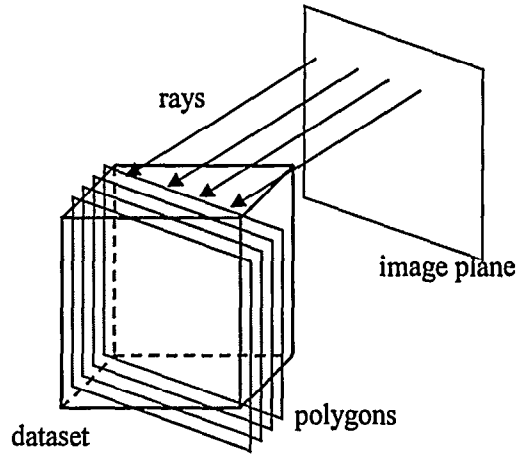

Figure 5 *3D texture mapping concept. Polygons are parallel to the image plane. Data is texture mapped onto one polygon, the texture mapped polygon is blended into the frame buffer and the next polygon processed.*

be designed and build. This also means that an application does not have to deal with hardware specific details, which in turn might mean that the application will run on one platform, but not on another because a certain hardware feature is not available. We will discuss several hardware architectures and show how they will be able to accelerate the Voxelator pipeline.

3D texture mapping hardware [1, 5, 6, 13, 20] can be used to accelerate the second section of the Voxelator pipeline, as we discussed in section 3. After the visibility testing, gradient calculation, classification and lighting is done in software, the Voxelator API can format the data into a texture map and pass that texture map, which contains classified and shaded voxels, on to the 3D texture mapping hardware. Next the Voxelator will define polygons that slice through the texture map perpendicular to the viewing direction. The 3D texture mapping hardware will then rasterize and texture map each polygon and perform the OpenGL fragment operations, like blending the data into the frame buffer. This is schematically shown in Figure 5. A potential problem can arise when the texture map is bigger than the available texture mapping memory in the system. Hewlett-Packard solved this problem with the texture caching model we discussed in section 5. If the system does not have a mechanism like texture caching the Voxelator API has to manage the texture map itself, by bricking it up into smaller chunks of data [13]. The first section of the Voxelator pipeline has to be computed on the host CPU, which is a significant part of the whole pipeline. Having to do this will be the performance bottleneck for this solution.

2D texture mapping hardware can be used if 3D texture mapping hardware is not available. It is possible to implement the second section of the Voxelator pipeline on 2D texture mapping hardware, and still get significant performance improvements over a software only solution. Instead of building a 3D texture map we can only send one slice of data at a time to the texture mapping hardware. This slice furthermore has to be parallel to the face of the dataset that is most perpendicular to the viewing direction. This is schematically shown in Figure 6. Next the Voxelator will define a polygon the slice is texture mapped onto. The fragments created will be processed by the hardware and blended into the frame buffer. Each consecutive polygon has to be sheared to correct for the offset created by having the polygons parallel to the face of the dataset instead of parallel to the image plane. The image that results will be warped and has to be un-warped to get the final image. This is very similar to the shear/warp algorithm [14].

The Cube-4 volume rendering accelerator [9, 16, 17] is a very good fit for the Voxelator pipeline. It in essence implements both sections of the pipeline, thus providing hardware acceleration for the whole pipeline, not just the rasterization stage as texture mapping hardware does. See Figure 7. This Figure gives a high level overview of the Cube-4 architecture. The power of Cube-4 lies in it's parallelism and modularity. Cube-4 works on one whole beam of voxels at a time. A beam consists of n voxels, where n is the size of the $nxnxn$ dataset being rendered. The design is highly pipelined, and after the pipeline is filled it generates n outputs every clock cycle. Thus all units shown in Figure 6 work on n voxels at the same time.

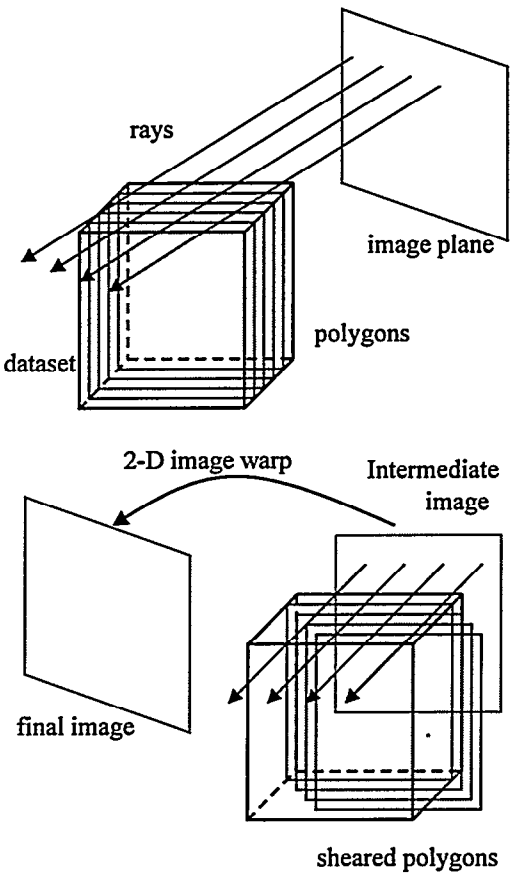

Figure 6 *2D texture mapping. (a) Top, polygons are parallel to the face of the dataset most perpendicular to the viewing direction. (b) Bottom, shearing the polygons will generate an intermediate image that will need to be warped to get the final rendering.*

We expect the minimal Cube-4 system to consist of a PCI or AGP style board, which contains the Cube-4 chips, memory for the dataset and memory for the resulting final image. By adding more memory and/or more Cube-4 chips the system will scale to the user's needs. The memory on the Cube-4 board can be viewed as a block cache. If the dataset does not fit into the Cube-4 on-board memory, parts of it can be paged in from main memory. Since Cube-4 was originally designed to hold the whole dataset in on-board memory it makes sense to define the Voxelator block size as the size of that on-board memory. The Cube-4 hardware architecture does not benefit from a smaller block size. See Figure 7. It is up to the Cube-4 hardware to define the block size, and up to the application to query that block size.
The current Cube-4 architecture does not address integration with the system's frame buffer and fragment operations stage. Cube-4 will generate the final image on-board, and will then blit that image over to the frame buffer. This prevents proper z integration. However, the Cube-4 designers are free to find a solution to this problem however they see fit to solve it. The Voxelator API does not force hardware designers into one particular solution. This is what makes the Voxelator API so powerful and flexible.

VIRIM is a fully programmable hardware engine built out of FPGAs and DSP signal processors [3, 4, 8]. It consists of two distinct parts, a rotator or resampler, and a ray-tracing unit. The rotator will resample the dataset so it aligns with the viewing plane using a programmable filter. After the alignment the ray-tracing unit will cast rays through the dataset, calculates the gradient, does the opacity mapping and the compositing. The units communicate with each other and the host processor over a VME bus. Linear scaling can be achieved by adding more rotator or ray-tracing units to the system. The system is operational and renders about 30 million voxels per second. The designers of VIRIM also recognize the memory system as the first bottleneck in their system. They propose [3] a memory system based on so called sub-cubes, similar to the blocks we propose in the Voxelator. Furthermore, just like the Cube-4 architecture, VIRIM implements both sections of the Voxelator pipeline. Since VIRIM is fully programmable it is possible to implement several different volume rendering solutions. VIRIM is perfectly suited to fit under the Voxelator API.

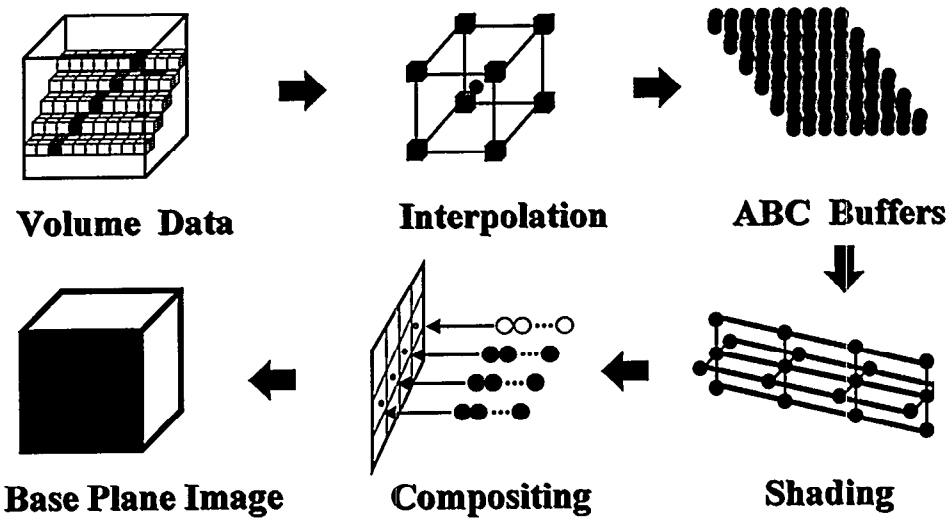

Figure 7. *Cube-4 architecture overview*

7 FUTURE DIRECTIONS

The Voxelator is the first step towards full integration of discrete data with geometric data. In the past, workstation vendors have always focused on polygonal acceleration. Geometric models are becoming bigger and more detailed, which means that the average polygon size goes down. The handling and processing of discrete data and geometric primitives has to be done at equal speeds and fully interchangeable with each other. Once a polygon reaches the size of a voxel one is processing discrete data. When a polygon becomes that small, it is better to eliminate the redundant information associated with a polygon and it's vertices and process the data directly using the volume pipeline.

The conventional geometric pipeline and the Voxelator pipeline have many operations in common, like transformation, lighting and shading, interpolation and fragment operations. A more integrated hardware design will re-use these blocks for both the volume pipeline and the geometry pipeline, saving development time, VLSI real estate, board real estate and thus ultimately cost and reliability.

The current OpenGL model does not completely address the issue of merging volume data and geometry data in 3D space. It cannot handle both geometry and volume data at the same time, and as a result the geometry data will be already collapsed into the frame buffer before the volume data is rendered, or vice versa. This limits the merging of both data types to opaque objects only. Both geometry and volume data should be able to intersect each other and the correct blending and rendering of the two should be guaranteed, regardless if each of them is transparent, semi-transparent or completely opaque.

The Voxelator pipeline currently addresses only regular gridded volume data. An obvious next step is to enhance the Voxelator pipe so it will be capable to handle rectilinear, curvilinear and maybe even completely irregular gridded volume datasets.

8 CONCLUSION

We have shown why an industry standard volume visualization API is essential for a broad adoption of this technology. By carefully considering complete system design it is possible to build a well balanced, high performance visualization system that accelerates both discrete data and geometric data.

9 ACKNOWLEDGEMENTS

I would like to thank the whole Voxelator design team who did a great job: Ken Severson, Randi Rost, Shaz Naqvi, Ales Fiala, Jeff Burrell, Russ Huonder and Dave Desormeaux. Furthermore thanks to Hanspeter Pfister for proofreading and supplying the cube-4 image in Figure 7.

References

[1] **K. Akeley.** RealityEngine Graphics. In *Computer Graphics Proceedings,* ACM SIGGRAPH, pp. 109-116, August 1993.

[2] **M. Bentum.** Interactive Visualization of Volume Data. *PhD Thesis,* University of Twente, 1995. ISBN 90-9008788-5.

[3] **M. de Boer, A. Gröpl, J. Hesser, R. Männer.** Latency- and Hazard-Free Volume Memory Architecture for Direct Volume Rendering. In *11th Eurographics Workshop on Graphics Hardware,* pp. 109-119, Poitiers, France, August 1996.

[4] **M. de Boer, J. Hesser, A. Gröpl, , T. Günther, C. Poliwoda, C. Reinhart, R. Männer.** Evaluation of a Real-Time Direct Volume Rendering System. In *11th Eurographics Workshop on Graphics Hardware,* pp. 121-131, Poitiers, France, August 1996.

[5] **B. Cabral, N. Cam, J. Foran.** Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Symposium on Volume Visualization,* pp. 91-98, Washington, 1994.

[6] **T. J. Cullip, U. Neumann.** Accelerating Volume Reconstruction with 3D Texture Hardware. In *technical report TR93-027,* University of North Carolina, Chapell-Hill, NC, 1993.

[7] **M. Doggett.** An Array Based Design for Real-Time Volume Rendering. In *10th Eurographics Workshop on Graphics Hardware,* pp. 93-101, Maastricht, The Netherlands, August 1995.

[8] **T. Günther, C. Poliwada, C. Reinhart, J. Hesser, R. Männer, H.-P. Meizner, H.-J. Baur.** VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine. In *9th Eurographics Workshop on Graphics Hardware,* pp. 103-108, Oslo, Norway, September 1994.

[9] **U. Kanus, M. Meißner, W. Straßer, H. Pfister, A. Kaufman.** Cube-4 Implementations on the Teramac Custom Computing Machine. In *11th Eurographics Workshop on Graphics Hardware,* pp. 133-143, Poitiers, France, August 1996.

[10] **G. Knittel.** A PCI-based Volume Rendering Accelerator. In *10th Eurographics Workshop on Graphics Hardware,* pp. 73-82, Maastricht, The Netherlands, August 1995.

[11] **G. Knittel.** A Scalable Architecture for Volume Rendering. In *9th Eurographics Workshop on Graphics Hardware,* pp. 58-69, Oslo, Norway, September 1994.

[12] **G. Knittel, W. Straßer.** A Compact Volume Rendering Accelerator. In *Symposium on Volume Visualization,* pp. 67-74, Washington, 1994.

[13] **T. Kulick.** Building an OpenGL Volume Renderer. http://reality.sgi.com/kulick_engr/devnews/volren/article.html. Silicon Graphics Computer Systems, Mountain View, CA.

[14] **P. Lacroute, M. Levoy.** Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Computer Graphics Proceedings,* ACM SIGGRAPH, pp. 451-458, July 1994.

[15] **J. Lichtermann.** Design of a Fast Voxel Processor for Parallel Volume Visualization. In *10th Eurographics Workshop on Graphics Hardware,* pp. 83-92, Maastricht, The Netherlands, August 1995.

[16] **H. Pfister, A. Kaufman.** Cube-4 A Scalable Architecture for Real-Time Volume Rendering. In *1996 Symposium on Volume Visualization,* pp. 47-54, San Francisco, October 1996.

[17] **H. Pfister, A. Kaufman, F. Wessels.** Towards a Scalable Architecture for Real-Time Volume Rendering. In *10th Eurographics Workshop on Graphics Hardware,* pp. 123-130, Maastricht, The Netherlands, August 1995.

[18] **R. Rost.** Volume Rendering Extensions for OpenGL. http://www.hp.com/info/voxelator. Hewlett-Packard Company, Palo Alto, CA., 1997.

[19] **M. Segal, K. Akeley.** The OpenGL Graphics System: A Specification (Version 1.1). http://www.sgi.com/Technology/openGL/spec.html. Silicon Graphics Computer Systems, Mountain View, CA., 1995.

[20] **A. Van Gelder, K. Kim.** Direct Volume Rendering with Shading via Three-Dimensional Textures. In *1996 Symposium on Volume Visualization,* pp. 23-28, San Francisco, October 1996.