TRIANGLECASTER

Extensions To 3D-Texturing Units For Accelerated Volume Rendering

Gunter Knittel
Hewlett-Packard Laboratories, Visual Computing Department, knittel@hpl.hp.com

ABSTRACT

We discuss hardware extensions to 3D-texturing units, which are very small but nevertheless remove some substantial performance limits typically found when using a 3D-texturing unit for volume rendering. The underlying algorithm uses only a slight modification of existing methods, which limits negative impacts on application software.

In particular, the method speeds up the compositing operation, improves texture cache efficiency and allows for early ray termination and empty space skipping. Early ray termination can not be used in the traditional approach.

Simulations show that, depending on data set properties, the performance of readily available, low-cost PC graphics accelerators is already sufficient for real-time volume visualization. Thus, in terms of performance, the TRIANGLECASTER-extensions can make dedicated volume rendering accelerators unnecessary.

CCS Categories and Subject Descriptors: I.3.1 **[Computer Graphics]**: Hardware Architecture - *graphics processors;* I.3.3 **[Computer Graphics]**: Picture/Image Generation - *display algorithms*

Additional Keywords and Phrases: graphics hardware, 3D-texture mapping, volume rendering, ray casting

1 INTRODUCTION

With the advent of 3D-texturing systems the idea was born to use the 3D-texturing hardware for volume rendering [1]. For this purpose, the data set (containing either the raw data for post-interpolation look-up or pre-classified and pre-shaded $RGB\alpha$-voxels) is loaded into the texture memory. A series of equidistant planes parallel to the view plane are cut through the volume, and processed in back-to-front order. Each cut is decomposed into triangles, which are scan-converted and "textured" using the volume data set as texture. All planes are blended together in some way to give the final image.

Conceptually, this method is equivalent to (orthographic) ray-casting, provided the pixel centers on the triangles are the intersection points of the rays with the planes.

1.1 Advantages

In general, 3D-texturing based volume rendering (henceforth called *TVR*) is an attractive idea since most of the required hardware units are already there: the texture memory as storage for the volume data set and arithmetic units for the mapping of the discrete 3D data set on arbitrarily oriented planes. In many graphics systems, the compositing (blending) operation is also implemented in hardware. Due to the back-to-front processing order, no opacity component needs to be stored in the frame buffer, which keeps memory costs and bandwidth requirements low. Thus, hardware support for basic volume rendering comes at no extra costs. In conjunction with versatile programming libraries such as the OpenGL Volumizer [14], interactive volume rendering finally becomes generally available to the graphics community.

Compared to a prominent dedicated volume rendering accelerator, the announced VolumePRO [11], TVR shows a number of additional advantages:

- ❑ Arbitrary perspective projections, instead of only parallel projections, can be generated. This is useful for walk-throughs and stereoscopic projections. Artifacts resulting from the non-uniform sampling rate along the rays can be avoided by using concentric spheres instead of parallel planes, and by approximating these spheres by a larger number of triangles.
- ❑ The sampling rate is arbitrary in all directions, with full hardware support.
- ❑ Polygonal and volume graphics can easily be combined.

1.2 Disadvantages

However, compared to VolumePRO and other volume rendering architectures such as VIRIM [7] or VOGUE [9], TVR shows two major disadvantages:

- ❑ no or limited support for classification and shading operations on a per-raypoint basis, and
- ❑ compromised rendering speed.

Most of the recent work in this area has been done to overcome the classification and shading limitations of existing machines. In [16], the achievable performance using pre-classification and -shading is evaluated. In [17], gradient shading is accomplished by interpolating the raypoint value and its surface normal, the latter from pre-computed voxel gradients, and by storing these quantities in the frame buffer. The final colors are obtained by copying each pixel onto itself while applying an appropriately initialized color matrix. The work in [6] also uses the 3D-texturing hardware to interpolate density and gradient components, whereas shading is done on the CPU. Even shadows can be included into texture-based volume rendering, as reported in [3]. The main focus of this work, however, is the limited achievable performance (for a short digression into inexpensive hardware support for classification and shading, see section 7). It is true that the original papers reported high frame rates (up to 10 frames/s for 512×512×64 volumes) [4],[5],[18], however, they used expensive, high-end graphics hardware. The goal of this work is to achieve high rendering speed for low-cost, potentially PC-based systems.

The performance of TVR is limited by a number of factors:

- The compositing operation causes excessive frame buffer traffic. For each raypoint, one read and one write operation must be done. A potentially existing frame buffer cache is most likely no remedy: accesses to the same screen location are separated by too many accesses to other pixels (potentially the entire screen is processed before a particular pixel is visited again). Thus, a frame buffer cache only allows us to perform these accesses in bursts.
- A large amount of data traffic occurs between the texture memory and the texturing unit, which makes the use of a texture cache mandatory. However, due to the plane-oriented processing order and the low temporal coherence of its access pattern we cannot achieve a satisfactory hit rate.
- Early ray termination can not be used due to the back-to-front and plane-oriented processing order. Therefore, we cannot take advantage of large opaque structures in the data set to increase rendering speed.
- Empty space skipping is also problematic due to the plane-oriented processing order. The OpenGL Volumizer API [14] proposes to subdivide the volume, and to render only the non-empty sub-volumes. However, this causes the number of triangles to increase, and may cause bottlenecks in other parts of the pipeline.

It should be noted that early ray termination and empty space skipping offer the biggest speed-up potential for many data sets.

Another disadvantage is the limited arithmetic precision and possibly large accumulated rounding error due to the limited width of the color components in the frame buffer.

All these disadvantages can be alleviated using the techniques as described in this work. In section 2, the underlying algorithmic changes are explained. The resulting hardware extensions are described in section 3. Section 4 covers issues concerning the mixed rendering of polygonal and volume data. Results obtained from a software simulation are given in section 5.

2 ALGORITHM

The work presented here relies on the assumption of parallel planes instead of concentric spheres. Conceptually, the processing is the same as for traditional systems. However, the (virtual) screen is divided into small tiles. Each screen tile is further subdivided into triangles, here called *S-triangles*. Examples are shown in Figure 1.

The projection of a given S-triangle on all planes yields a set of triangles which are herein called *P-triangles*.

Instead of the commonly used plane-at-a-time order, the algorithm processes all P-triangles associated with a given S-triangle. After completion, the algorithm steps to the next S-triangle until the virtual screen is finished.

This small modification shows a number of substantial advantages:

- a small memory unit on the same chip as the texturing unit can be used as frame buffer for one complete S-triangle. All compositing operations are then performed completely on-chip, which yields a dramatic speed-up.
- It is then possible to provide storage for α-values, so that front-to-back processing can be used without increasing the size of the external frame buffer. Then, we can use early ray termination in units of S-triangles: whenever all pixels of an S-triangle are opaque, no more P-triangles need to be processed.
- Due to the small number of entries in the on-chip buffer, we can use more bits per component to provide the needed numerical precision.
- The small size gives us the opportunity for an additional optimization: we can use a dual-ported memory, one write-port and one read-port, so that reading raypoint p and writing raypoint q can take place at the same time.
- The efficiency of the texture cache can be increased substantially, due to the better locality of the accesses to the volume data set.
- Most (low-end) graphics systems have a combined memory for z, RGBα and textures. The on-chip memory frees up a large amount of bandwidth, which can be used for accessing the volume data set.

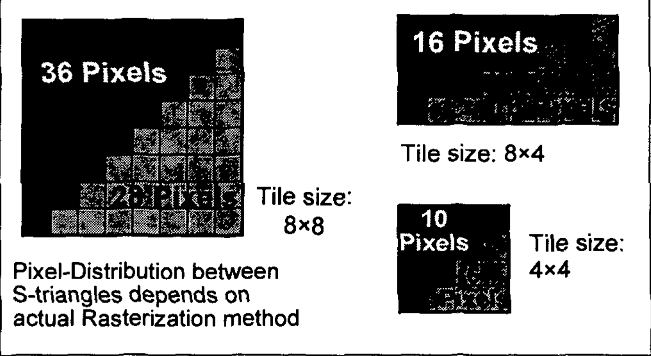


Figure 1: Possible Tile Sizes and S-Triangles

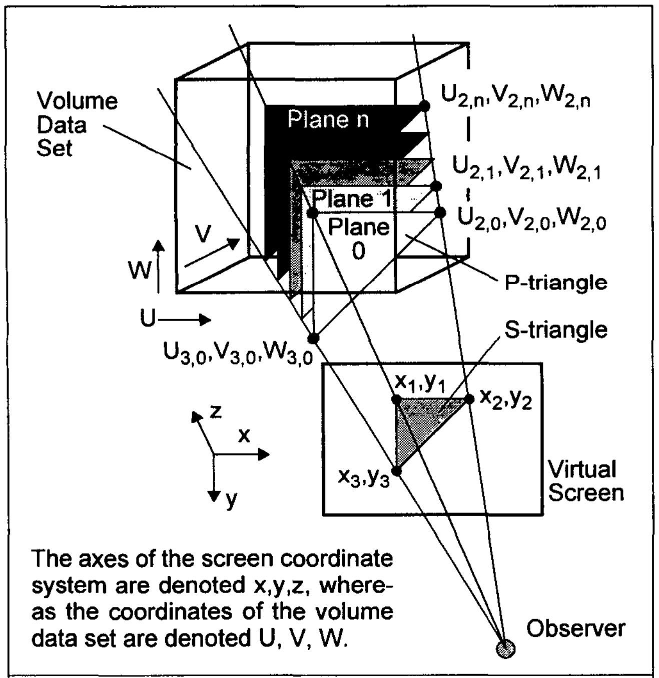
The on-chip memory and the associated arithmetic units are herein collectively called **Compositing Buffer** (see section 3.2), and represent part of the proposed hardware extensions.

However, the generation of all those P-triangles must not create a performance limitation at other parts of the pipeline. Table 1 lists the required triangle generation rates for different resolutions under the assumption of an 8×8 pixel tile size, 10 frames per second and no empty space skipping or early ray termination. Any of these rates may already exceed the capabilities of the target graphics system.

Planes	Screen Resolution		
	256×256	512×512	1k×1k
256	5MT/s	20MT/s	80MT/s
512	10MT/s	40MT/s	160MT/s
1k	20MT/s	80MT/s	320MT/s

Table 1: Required Triangle Generation Rates

However, the special mode of operation as shown in Figure 2 simplifies the generation of the P-triangles very much.



Volume Data Set
Plane n
Plane 1
Plane 0
$U_{2,n},V_{2,n},W_{2,n}$
$U_{2,1},V_{2,1},W_{2,1}$
$U_{2,0},V_{2,0},W_{2,0}$
P-triangle
S-triangle
$U_{3,0},V_{3,0},W_{3,0}$
W
V
U
z
x
y
x_1,y_1
x_2,y_2
x_3,y_3
Virtual Screen
The axes of the screen coordinate system are denoted x,y,z, whereas the coordinates of the volume data set are denoted U, V, W.
Observer

Figure 2: Casting Triangles through the Volume

Obviously, the P-triangles on the different planes all share the same screen coordinates and rasterization parameters, with the exception of the texture coordinates of the vertices. However, these can be found by adding a constant offset when proceeding from one plane to the next, assuming equidistant planes. Written equationally, this gives:

$$\begin{aligned} U_{k,n+1} &= U_{k,n} + \Delta U_k \\ V_{k,n+1} &= V_{k,n} + \Delta V_k \\ W_{k,n+1} &= W_{k,n} + \Delta W_k \end{aligned} \qquad (1)$$

In (1), $k = \{1, 2, 3\}$ denotes the triangle vertex and n is the plane number. Thus, to generate the texture coordinates of P-triangle $n+1$ from the parameters of P-triangle n, nine additions must be performed.

Depending on the actual implementation of the rasterizer, it may only be necessary to generate the texture coordinates of one vertex per triangle, reducing the number of adds to three.

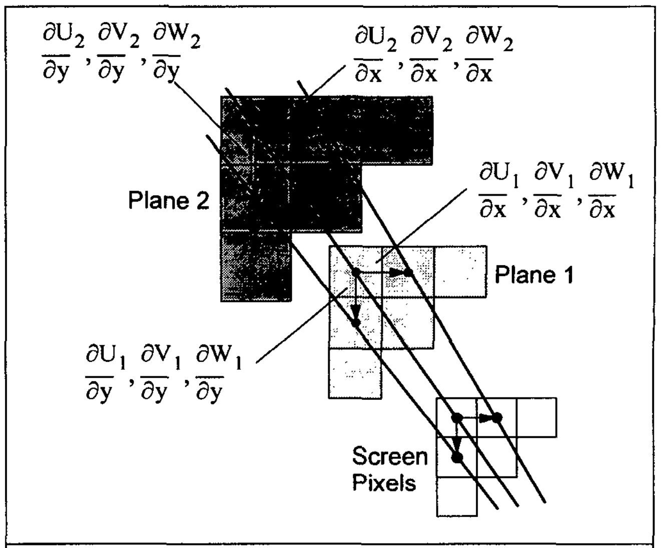
For perspective projections, the derivatives of the texture coordinates with respect to the screen coordinates are not constant from one plane to the next. Thus, these parameters must be updated as well. However, these updates are again linear increments as illustrated in Figure 3.

Thus, updating the derivatives is given by:

$$\frac{\partial U_{n+1}}{\partial x} = \frac{\partial U_n}{\partial x} + \Delta\left(\frac{\partial U}{\partial x}\right) \qquad \frac{\partial U_{n+1}}{\partial y} = \frac{\partial U_n}{\partial y} + \Delta\left(\frac{\partial U}{\partial y}\right) \qquad (2)$$

and accordingly for the *V*- and *W*-derivatives. This operation involves six additions. Thus, generating the next P-triangle requires only 9 or 15 additions.

Generating the P-triangles can therefore be done by a small dedicated hardware unit, herein called **Triangle Generation Unit** (see section 3.1), which is the second hardware extension proposed in this work.



$\frac{\partial U_2}{\partial y}, \frac{\partial V_2}{\partial y}, \frac{\partial W_2}{\partial y}$
$\frac{\partial U_2}{\partial x}, \frac{\partial V_2}{\partial x}, \frac{\partial W_2}{\partial x}$
Plane 2
$\frac{\partial U_1}{\partial x}, \frac{\partial V_1}{\partial x}, \frac{\partial W_1}{\partial x}$
Plane 1
$\frac{\partial U_1}{\partial y}, \frac{\partial V_1}{\partial y}, \frac{\partial W_1}{\partial y}$
Screen Pixels

Figure 3: Updating the Texture Coordinate Derivatives

Using this scheme, the host or the geometry accelerator generates only the **front triangle** of a given S-triangle and the increments in (1) and (2). The front triangle is the frontmost P-triangle which cuts the (bounding volume of the) data set. Additionally, the farthest P-triangle having points in common with the data set (called the **back triangle**) is determined. The number of P-triangles from the front to the back triangle is used as a counter value. Processing of the actual S-triangle is terminated when this counter has expired or all pixels are opaque.

2.1 Generation of the Front and Back Triangles and Empty Space Skipping

The method proposed here uses the graphics pipeline and can be classified as a bounding hull scan conversion algorithm (distantly related to PARC [15]). However, a small modification to the z-buffer circuitry can again provide a substantial speed-up.

The method uses a polygonal hull of the non-zero elements in the data set. If the data set contains different materials, each material should have its own hull. The bounding hull should be defined using only a small number of triangles, typically several hundred. It must be built once per data set, or optionally after each re-classification. This can be done using the Marching-Cube-algorithm and mesh simplification methods afterwards. For some data sets, however, the hull may simply be the bounding box of the entire data set, eliminating the possibility of this kind of empty space skipping in this case.

For each frame, the bounding hull is rendered twice into the z-buffer at virtual screen resolution, using the standard “less than” z-compare operator in the first pass and the “greater than” operator in the second. After each pass, the z-buffer is read back. The CPU finds the nearest (farthest) value in each S-triangle, which after perspective correction and rounding defines the plane of the front (back) triangle.

The z-buffer circuitry could be modified such that the reads are done in units of screen tiles, and that it determines the closest z-value (or farthest, respectively) in each S-triangle by itself using the existing z-comparator. This reduces the data traffic back to the CPU by a factor of 32 in case of 8×8 tiles.

It should be noted that we can not jump over internal empty spaces using this method.

2.2 Depth-Cueing

Assumed the rasterizer/texturing unit is capable of weighting the texture color $R_T G_T B_T$ of a pixel by its interpolated (shaded) color $R_I G_I B_I$, i.e.

$$R = R_T \cdot R_I \quad G = G_T \cdot G_I \quad B = B_T \cdot B_I \tag{3}$$

then this feature can be used for depth-cueing with no further hardware expenses and only a slight performance loss. For this purpose, the CPU computes the depth-cueing factors, x- and y-derivatives and all increments from one plane to the next for the front triangle. The Triangle Generation Unit interpolates the depth-cueing parameters and hands them to the rasterizer in the color channels for interpolation.

In this way it is even possible to simulate the effect of distance-dependent color shifts. This scheme can be used for TVR as well.

3 THE TRIANGLECASTER EXTENSIONS

Figure 4a shows a typical midrange graphics system for workstations or PCs. High-performance versions have multiple pipelines and some sort of interconnection structure. Geometry accelerators may or may not be present. Figure 4b shows the same pipeline with the TRIANGLECASTER hardware extensions supporting the algorithm outlined above.

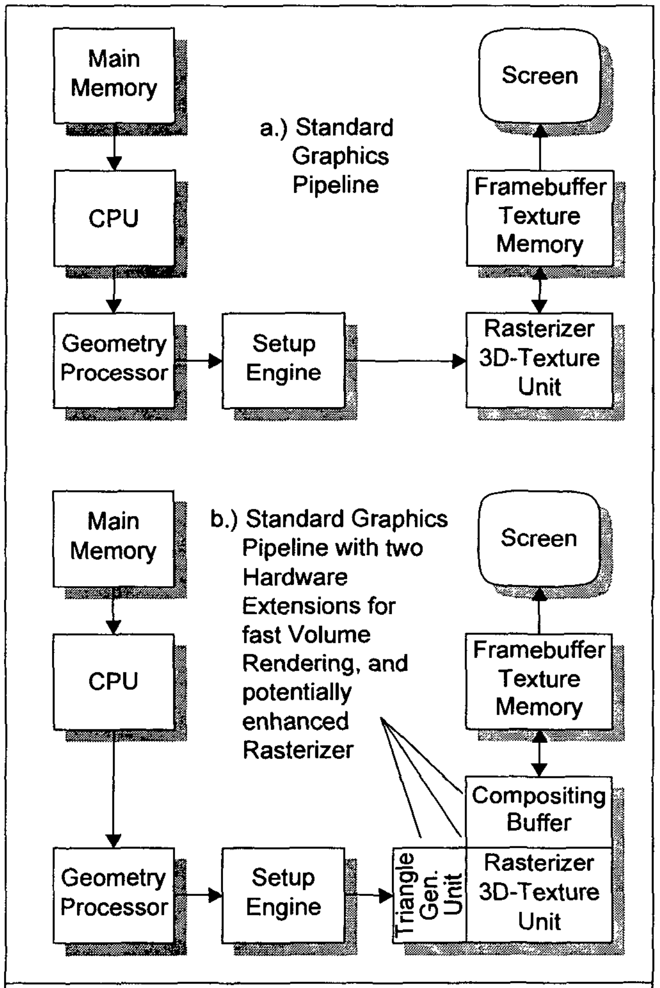

Main Memory
a.) Standard Graphics Pipeline
Screen
CPU
Framebuffer Texture Memory
Geometry Processor
Setup Engine
Rasterizer 3D-Texture Unit
Main Memory
b.) Standard Graphics Pipeline with two Hardware Extensions for fast Volume Rendering, and potentially enhanced Rasterizer
Screen
CPU
Framebuffer Texture Memory
Compositing Buffer
Geometry Processor
Setup Engine
Triangle Gen. Unit
Rasterizer 3D-Texture Unit

Figure 4: The TRIANGLECASTER Hardware Extensions

3.1 Triangle Generation Unit

A simplified block diagram of the Triangle Generation Unit is shown in Figure 5. Depending on the desired triangle generation rate, the triangle parameters can be computed sequentially, in parallel, or partially sequentially as in this example, which uses three adders. Additionally, there are three multiplexers, a control unit, a counter, three sets of increment registers and three sets of result registers.

For a given S-triangle, all needed rasterization and texturing parameters of the front triangle are generated by the host CPU or dedicated hardware units (geometry or setup processor) and written into the result registers. The increments of the texture coordinates and their derivatives are also computed and written into the increment registers.

After having received the counter value, the control unit transfers the parameters of the front triangle to the rasterizer/texturing unit, generates the next P-triangle and decrements the counter. If the counter value reached zero, the processing of the current S-triangle is terminated. Otherwise the control unit waits until the rasterizer/texturing unit is ready for the next P-triangle.

Termination of the current S-triangle can also occur if the Compositing Buffer detects that all P-triangles behind the one it has just finished are invisible. In this case, the Compositing Buffer activates the signal *TT* (see also Figure 7).

In Figure 5 we assumed that the generation of the texture coordinates for one vertex per P-triangle is sufficient. The result register files hold additional data, in case parameters such as screen coordinates must be reloaded for every P-triangle.

Obviously, this unit requires very little chip space, even if a higher degree of parallelism is required to meet the performance figures given in Table 1.

3.2 Compositing Buffer

Regardless of how the shading and classification have been done (either during pre-processing or on-the-fly), pixels (raypoints) coming to this stage are defined by their color and opacity. Processing is done in front-to-back order to take advantage of early ray termination (on a per-triangle-basis).

Instead of an alpha-component, the accumulated translucency

$$\Omega_n = \prod_{i=0}^{n} (1 - \alpha_i) \qquad 0 \le \alpha_i \le 1 \tag{4}$$

is maintained in the on-chip memory. Thus, for the processing of the *n*-th resample location (a raypoint on the *n*-th plane), the following operations must be done:

$$\begin{aligned} C_{new} &= C_{old} + C_n \cdot \alpha_n \cdot \Omega_{old} \\ \Omega_{new} &= \Omega_{old} \cdot (1 - \alpha_n) = \Omega_{old} - \alpha_n \cdot \Omega_{old} \end{aligned} \tag{5}$$

In (5), *C* can be *R*, *G* or *B*, and C_n represents the color components of the resample location. Accordingly, α_n is the opacity of the raypoint. The quantities with index "*old*" are taken out of the Compositing Buffer and replaced by their updated versions with index "*new*". The operations in (5) can be carried out in a pipeline, since the processing order as shown in Figure 6 eliminates all data dependencies.

A simplified block diagram of the Compositing Buffer is shown

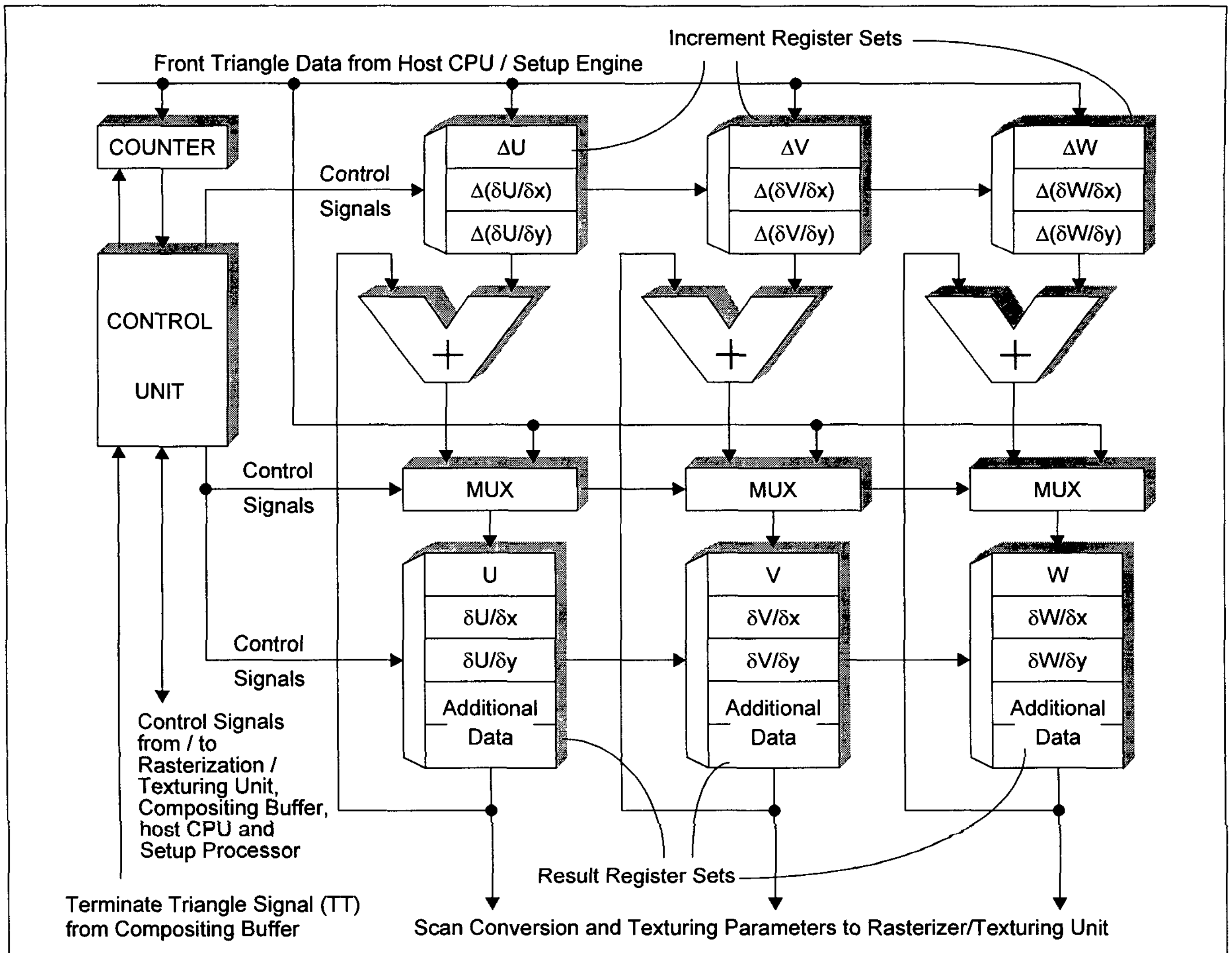

Increment Register Sets
Front Triangle Data from Host CPU / Setup Engine
COUNTER
CONTROL UNIT
Control Signals
ΔU
Δ(δU/δx)
Δ(δU/δy)
ΔV
Δ(δV/δx)
Δ(δV/δy)
ΔW
Δ(δW/δx)
Δ(δW/δy)
+
MUX
U
δU/δx
δU/δy
Additional Data
V
δV/δx
δV/δy
W
δW/δx
δW/δy
Control Signals from / to Rasterization / Texturing Unit, Compositing Buffer, host CPU and Setup Processor
Result Register Sets
Terminate Triangle Signal (TT) from Compositing Buffer
Scan Conversion and Texturing Parameters to Rasterizer/Texturing Unit

Figure 5: Schematic Diagram of the Triangle Generation Unit (simplified)

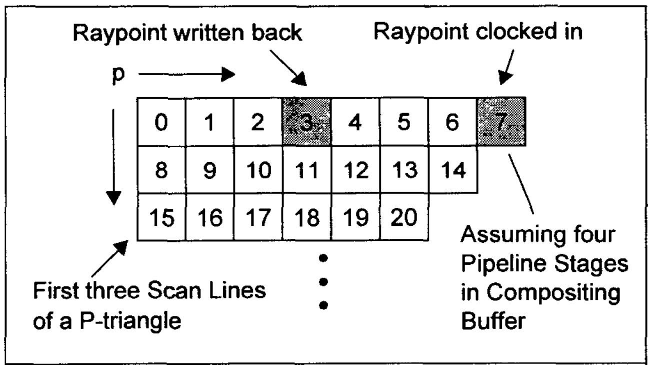


Figure 6: Processing Order (Enumeration) of Raypoints

in Figure 7. The index n denoting the plane number has been omitted (it is assumed that all raypoints currently in the pipeline are on the same P-triangle). The parameter p is the pixel number within the actual P-triangle, as depicted in Figure 6. Due to the pipelined architecture, there are always a number of pixels simultaneously under construction. It may never happen, however, that pixel p is read for P-triangle $n+1$ while or before pixel p of P-triangle n is completed. Given a certain number of pipeline stages, this sets a lower limit for the size of the S-triangles.

Early ray termination is facilitated using the circuitry at the bottom of Figure 7. The user can specify a threshold value ε for the translucency Ω along the rays. If Ω falls below ε for a given ray, all further raypoints are considered invisible. The S-triangle can be terminated if this condition occurs for all of its rays (pixels).

Every newly generated Ω-value is compared to this threshold. If it is less, a corresponding bit in the opaque-pixels register is set, and remains set until the current S-triangle is finished. This register is reset at the beginning of a given S-triangle. If all pixels in the current S-triangle have a set bit, every pixel is opaque, and thus, the Compositing Buffer sends a signal (labeled *TT*) to the Triangle Generation Unit to terminate this S-triangle and to start the next one.

The triangle-oriented approach bears the disadvantage that opaque pixels are still processed until the very last pixel of the associated S-triangle becomes opaque. However, using the individual opaque-pixel signals (OP_n) from the opaque-pixels register, the rasterizer/3D-texturing unit could be modified such

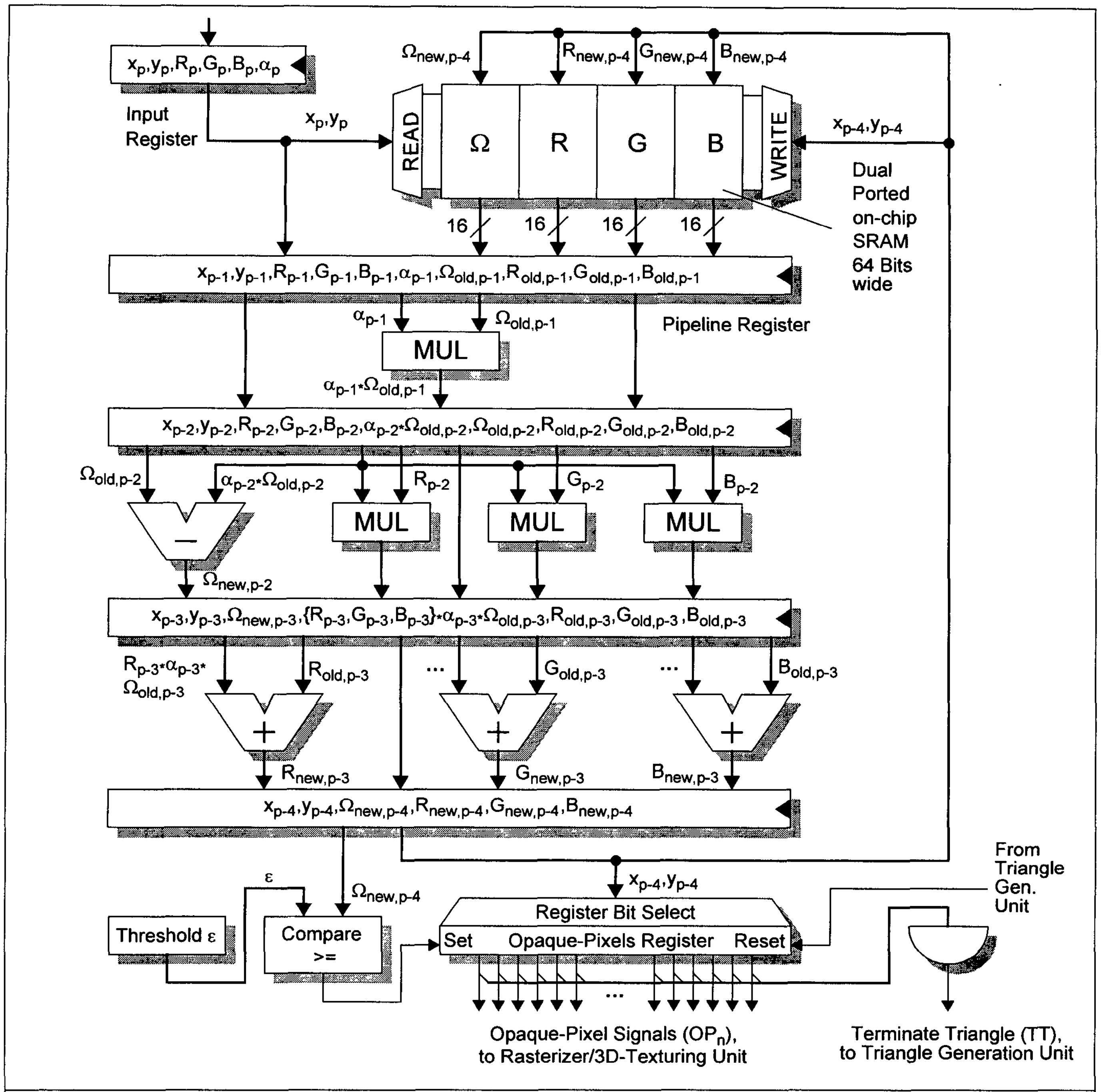

$x_p, y_p, R_p, G_p, B_p, \alpha_p$
Input Register
x_p, y_p
$\Omega_{new,p-4}$
$R_{new,p-4}$
$G_{new,p-4}$
$B_{new,p-4}$
READ
Ω R G B
WRITE
x_{p-4}, y_{p-4}
Dual Ported on-chip SRAM 64 Bits wide
16 16 16 16
$x_{p-1}, y_{p-1}, R_{p-1}, G_{p-1}, B_{p-1}, \alpha_{p-1}, \Omega_{old,p-1}, R_{old,p-1}, G_{old,p-1}, B_{old,p-1}$
α_{p-1}
$\Omega_{old,p-1}$
Pipeline Register
MUL
$\alpha_{p-1} * \Omega_{old,p-1}$
$x_{p-2}, y_{p-2}, R_{p-2}, G_{p-2}, B_{p-2}, \alpha_{p-2} * \Omega_{old,p-2}, \Omega_{old,p-2}, R_{old,p-2}, G_{old,p-2}, B_{old,p-2}$
$\Omega_{old,p-2}$
$\alpha_{p-2} * \Omega_{old,p-2}$
R_{p-2}
G_{p-2}
B_{p-2}
MUL MUL MUL
$\Omega_{new,p-2}$
$x_{p-3}, y_{p-3}, \Omega_{new,p-3}, \{R_{p-3}, G_{p-3}, B_{p-3}\} * \alpha_{p-3} * \Omega_{old,p-3}, R_{old,p-3}, G_{old,p-3}, B_{old,p-3}$
$R_{p-3} * \alpha_{p-3} * \Omega_{old,p-3}$
$R_{old,p-3}$
$G_{old,p-3}$
$B_{old,p-3}$
$R_{new,p-3}$
$G_{new,p-3}$
$B_{new,p-3}$
$x_{p-4}, y_{p-4}, \Omega_{new,p-4}, R_{new,p-4}, G_{new,p-4}, B_{new,p-4}$
ε
$\Omega_{new,p-4}$
Threshold ε
Compare >=
x_{p-4}, y_{p-4}
Register Bit Select
Set Opaque-Pixels Register Reset
From Triangle Gen. Unit
Opaque-Pixel Signals (OP_n), to Rasterizer/3D-Texturing Unit
Terminate Triangle (TT), to Triangle Generation Unit

Figure 7: Compositing Buffer (Number and Structure of Pipeline Stages may vary)

that it either skips over opaque pixels entirely, or at least avoids cache fill operations in case of texture cache misses for these pixels.

A traditional raycasting accelerator, however, may also not reach optimum speed-up from early ray termination: it suffers from the possibly large number of pipeline stages between the raypoint generator and the compositing unit. There may be a large number or even all remaining raypoints of a prematurely terminated ray already in the pipeline. Then, no or only small performance gains can be obtained from this technique.

Once a given S-triangle is finished, the contents of the on-chip frame buffer are written into the external frame buffer for display. The on-chip frame buffer is then initialized for the next S-triangle ($\Omega = 1, R = G = B = 0$).

The needed capacity of the on-chip dual-ported SRAM is 512 Bytes, assuming that an entire 8×8 pixel tile can be cached and that each component is 16 bits wide. It should be noted, however, that a blending stage and possibly a frame buffer cache may be needed for the standard rendering functionality anyway (e.g., for transparent or multi-textured objects). In this case, the additional chip space requirements for the TRIANGLECASTER-extensions may be close to negligible.

4 MIXING GEOMETRY AND VOLUMES

We only consider opaque geometrically defined objects in this work. The general processing order is outlined below.

1. The volume's bounding hull is rendered into a far and near buffer to determine the front and back triangles as described in section 2.1.
2. All the geometry is rendered in the standard way with z-buffering enabled.
3. Z-tests remain enabled with "less-than"-operator, but z-buffer updates are disabled.
4. All P-triangles are rendered as explained throughout the paper, but additionally with z-values attached to them (see below).

During step 4., however, the operation of the rasterizer must be modified in the following way. Whenever a z-value in the z-buffer is smaller than that of a pixel (raypoint) of a P-triangle, the corresponding color values in the frame buffer (belonging to the geometric object) are blended with the values in the Compositing Buffer using $\alpha=1$, instead of the raypoint colors. This pixel is then marked by a set opaque-pixel flag (OP_n), which can be used to exclude it from further processing. Note that if the entire P-triangle is obscured by geometry, processing of the S-triangle is ended due to early ray termination.

However, two issues need additional consideration. First, for perspective projections of polygonal primitives, the rasterizer usually interpolates $z' = z/w$, with $w = z/z_E + 1$ and $\{0,0,-z_E\}$ being the eye point. For correct z-compares, the z-values of the P-triangle vertices must also be perspectively transformed. Clearly, this would not be feasible given the high number of P-triangles. However, since the planes through the volume are parallel to the screen, all P-triangle vertices on a given plane have the same z-value, and therefore transform to the same perspective z-value z'. Thus, all z'-values for a given view can be precomputed and stored in a table to which the Triangle Generation Unit has access (e.g., in the frame buffer). It performs one table look-up per P-triangle using the plane number as index, and passes the z'-value along with $dzx' = dzy' = 0$ to the rasterizer.

Second, the need to perform z-tests introduces additional frame buffer traffic compared to pure volume rendering. In many cases, however, the geometry covers only small portions of the screen, as in the case of a scalpel or a prosthetic device in a medical data set. Then, performing z-tests for all P-triangles is a waste of bandwidth. This can be avoided by a small change in the operation mode of the z-buffer circuitry. For each front triangle, the rasterizer checks whether all z-values still have their initial value. If so, no z-tests must be done for all remaining P-triangles of that S-triangle.

5 RESULTS

A software simulator, containing z-buffer, Gouraud shader, 3D texturing unit and the TRIANGLECASTER-extensions was written to evaluate cache efficiency, memory traffic and achievable performance. We assumed an 8×8 pixel tile size. The memory in the Compositing Unit was implemented as a direct mapped cache instead of an SRAM as shown in Figure 7. This was done because then the same memory can be used as a frame buffer cache when comparing the TRIANGLECASTER-extensions to the traditional method (see section 5.1). The frame buffer cache has 64 entries, each 64 bits wide, to store the *RGBΩ*-components of all pixels in a tile (or two S-triangles) in 16-bit precision.

An eight-bank, direct-mapped texture cache was implemented, which allows the parallel access to any (cached) set of 2×2×2 voxels for tri-linear interpolation. The texture cache can either hold 512 or 4096 voxels. Voxels are 16 bits wide for a texture cache size of 1K or 8KByte. One cache line in each bank holds 8 voxels. Cache lines are updated individually in chunks of 16 Bytes, accordingly.

The test data sets were taken from the HP Voxelator CD: MRBrain (256×256×109, courtesy of UNC Chapel Hill) and Engine (256×256×110). Screen resolution was 256×256, except where noted otherwise in column 1 (see tables on next page). The distance from one plane to the next was 0.75 volume data set grid units. Both data sets have been rendered using a texture cache size of 1K and 8KByte, and early ray termination on and off. For early ray termination, only the *TT*-signal was used. Thus, invisible raypoints were processed in full until the entire S-triangle was opaque. A pixel was considered opaque when its translucency Ω fell below 1/255.

The results using the TRIANGLECASTER-extensions are summarized in Table 2.

We counted texture (volume) cache hits and misses in the following way. If for a given raypoint k of the eight surrounding voxels needed for tri-linear interpolation have been found in the cache, the hits have been increased by k, and the misses by $8-k$. Thus, the following relation must always be satisfied: *Raypoints* $* \; 8 =$ *Hits* $+$ *Misses*. For each miss, however, 8 voxels or 16 bytes are fetched from texture memory since a cache line holds 8 voxels. Thus, the number of bytes read from texture memory equals *Misses* $* \; 16$. The number of processed S-triangles is given by the projection of the bounding hull of the data set on the screen. For evaluation purposes, we used very tight bounding hulls, which consist of volume cell faces. They were automatically generated using a threshold value, which caused some noise to be included. The projection of the bounding hulls and the resulting set of S-triangles can be seen in Figure 8a and d.

Using pure volume rendering and the TRIANGLECASTER-extensions, there are no reads from the frame buffer. For each S-triangle, about 32 *RGB*-triples are written to the frame buffer on average.

The images in Figure 8b, c, e and f were generated by post-interpolation look-up, using only the interpolated function value as index to obtain color and opacity. No attempt was made to include more sophisticated classification and shading techniques (see section 7). The MRBrain data set contains index bits for four different materials, which were used to select one of four lookup-tables.

Note that the performance figures are the same for the opaque and translucent view of each data set if early ray termination is disabled. Therefore, they are not listed separately in Table 2.

5.1 Traditional Method

The tests have also been run in the traditional way, i.e., back-to-front and one plane at a time. For fairness, we used the same tight bounding hull for empty space skipping, and the same triangulation across the planes as in the TRIANGLECASTER-version. Thus, these tests produced the same number of triangles and raypoints as shown in Table 2 with early ray termination disabled. The results are given in Table 3. Again, opaque and translucent views have the same performance.

Dataset	Fig.	Volume Cache Size	Early Ray Term.	P-Triangles Total	S-Tri.	Raypoints (Tri-linear Inter-polations)	Volume (Texture) Cache Hits	Volume (Texture) Cache Misses	Bytes read from Texture Memory	Bytes written to Frame Buffer	Net Bandwidth needed for 30f/s (MB/s)
MRBrain	8b 8c	1KB	No	196,999	1,816	6,300,309	49,129,695	1,272,777	20,364,432	174,120	616
		8KB	No	196,999		6,300,309	49,266,118	1,136,354	18,181,664		551
opaque	8b	1KB	Yes	57,307		1,833,306	14,303,929	362,519	5,800,304		179
		8KB	Yes	57,307		1,833,306	14,425,592	240,856	3,853,696		121
trans-lucent	8c	1KB	Yes	133,323		4,263,945	33,268,087	843,473	13,495,568		410
		8KB	Yes	133,323		4,263,945	33,396,372	715,188	11,443,008		349
512^2	(8c)	8KB	Yes	481,865	7,093	15,415,406	121,888,266	1,434,982	22,959,712	680,544	709
1024^2	(8c)	8KB	Yes	1,765,039	27,661	56,425,601	448,042,360	3,362,448	53,799,168	2,654,472	1,694
Engine	8e 8f	1KB	No	120,801	1,343	3,785,584	28,246,026	2,038,646	32,618,336	128,688	982
		8KB	No	120,801		3,785,584	29,258,582	1,026,090	16,417,440		496
opaque	8e	1KB	Yes	30,148		953,630	7,132,961	496,079	7,937,264		241
		8KB	Yes	30,148		953,630	7,425,640	203,400	3,254,400		101
trans-lucent	8f	1KB	Yes	120,794		3,785,518	28,245,518	2,038,626	32,618,016		982
		8KB	Yes	120,794		3,785,518	29,258,074	1,026,070	16,417,120		496

Table 2: Performance Measurement Summary with TRIANGLECASTER-Extensions

Dataset	Fig.	Volume Cache Size	Planes	Raypoints (Tri-linear Inter-polations)	Volume Cache Hits	Volume Cache Misses	Bytes read from Texture Memory	Bytes read from Frame Buffer	Bytes written to Frame Buffer	Net Bandwidth needed for 30f/s (MB/s)
MRBrain	8b 8c	1KB	253	6,300,309	47,727,514	2,674,958	42,799,328	19,167,456	18,904,236	2,426
		8KB	253	6,300,309	47,877,794	2,524,678	40,394,848	19,167,456	18,904,236	2,354
512^2	(8c)	8KB	253	23,802,207	186,814,336	3,603,320	57,653,120	72,151,536	71,412,348	6,037
1024^2	(8c)	8KB	253	90,020,181	714,397,410	5,764,038	92,224,608	274,692,288	270,069,492	19,110
Engine	8e 8f	1KB	233	3,785,584	26,779,705	3,504,967	56,079,472	11,772,768	11,578,164	2,383
		8KB	233	3,785,584	27,279,610	3,005,062	48,080,992	11,772,768	11,578,164	2,143

Table 3: Performance Measurement Summary using traditional texture-based Volume Rendering

6 DISCUSSION

The most striking result is that all 256^2-views in Table 2 could be generated by low-cost PC hardware at 30frames/s if all speed-up methods offered by the TRIANGLECASTER-extensions are used (speaking only in terms of performance, since most of the PC graphics accelerators do not yet support 3D texturing. However, the step from tri-linear texture filtering to 3D texturing is not a dramatic one). The maximum numbers across the table for 256×256 screens are 8KByte texture cache, 512Byte frame buffer cache, 14MByte addressable texture memory, 4M P-triangles/s, 128M tri-linear interpolations per second, and 1GByte/s memory bandwidth. Most of these numbers are met or exceeded by typical PC graphics accelerators [2],[12],[13]. The required P-triangle rate can easily be achieved by inclusion of a Triangle Generation Unit (see section 3.1). At the same time, this would be the only major addition to the consumed silicon area besides an extended blending stage as discussed in section 3.2.

The most dramatic improvement over traditional texture-based volume rendering comes from the almost complete elimination of the frame buffer traffic, which was the main motivation for this work. This becomes most apparent for higher screen resolutions, as shown for 512×512 and 1024×1024 pixel screens. Next, the texture memory traffic is reduced by about 50% on average (compare Table 2, test runs without early ray termination, to Table 3).

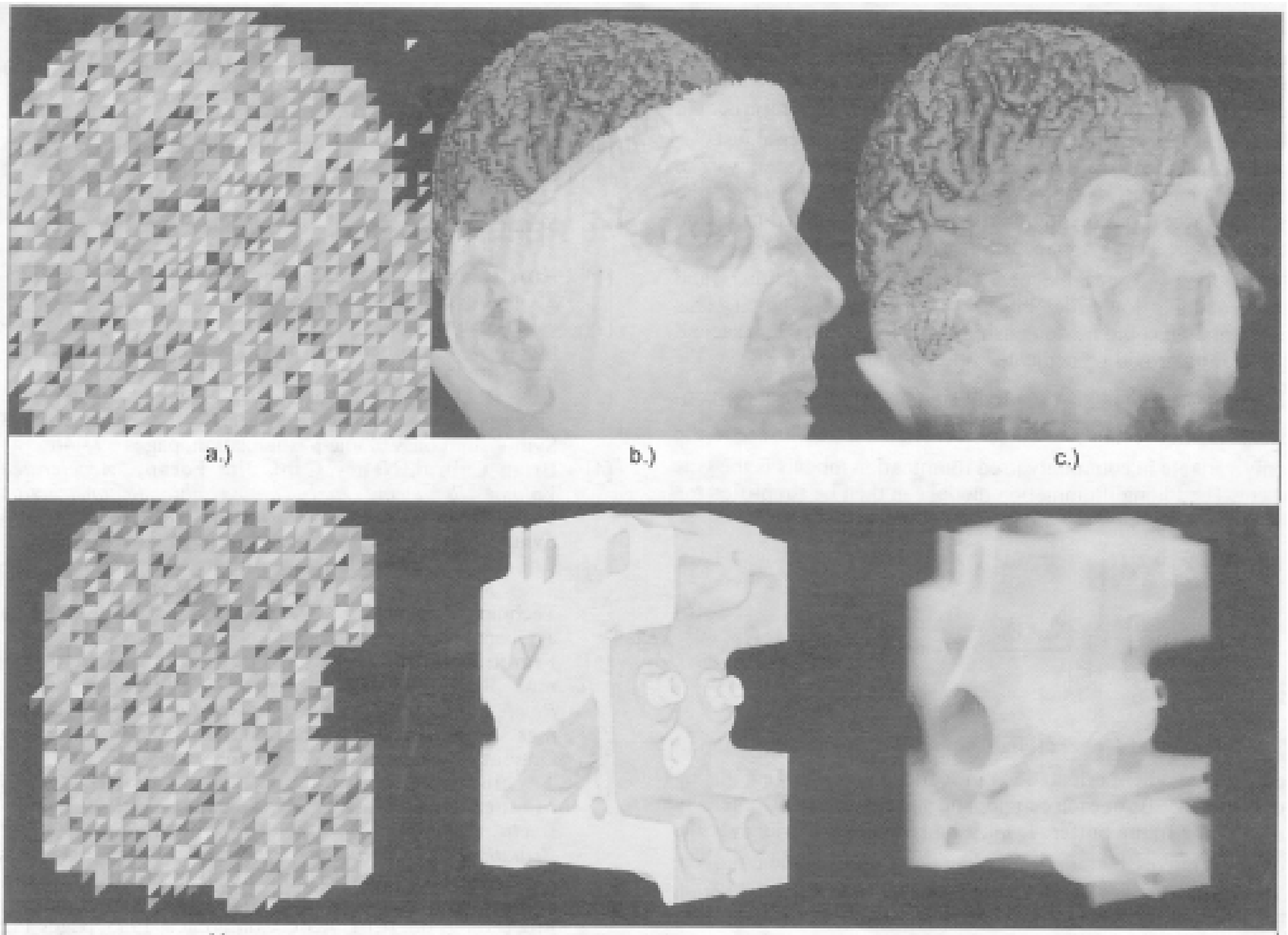


Figure 8: Images of the Test Data Sets

Finally, early ray termination can further reduce the rendering costs enormously, in our examples by as much as 75%, depending of course on viewing parameters and data set characteristics.

On the other hand, traditional texture-based volume rendering poses serious bandwidth requirements on the graphics system, as detailed in Table 3. For 256×256 images, we can achieve a 21.2-fold reduction in overall traffic in the best case (Engine, opaque, 8KByte cache), while we still achieve a 2.4-fold reduction ~~in~~ the worst case (Engine, translucent, 1KByte cache).

For 512×512 and 1K×1K images, the required bandwidth might be out of reach for all but the most expensive machines using the traditional method.

With the TRIANGLECASTER-extensions, however, the required memory bandwidth is already provided by mid-range PC systems. In case of a 512×512 (1K×1K) screen, the image of Figure 8c would require 14M (53M) P-triangles/s, and 462M (1.7G) tri-linear interpolations per second for 30frames/s. This might currently be out of reach for PC graphics accelerators as well, however, the bottlenecks using the TRIANGLECASTER-method now occur *on-chip*, where they can be tackled much easier.

7 REAL-TIME CLASSIFICATION AND SHADING

The prior work as listed in section 1.2 focussed on improving existing designs. However, for new designs, there is a whole range of possible solutions from very low-cost implementations to expensive gradient filters and Phong shaders. The following methods are targeted mainly at low-cost systems and can complement the TRIANGLECASTER-extensions for a complete volume rendering accelerator.
Classification and, in cost-efficient solutions, also shading are usually done table-based, so the task is to make pointers available and to provide on-chip or external look-up tables.
The most basic method (as used in this work) to determine the color C and opacity α of a raypoint is to use the interpolated function value F as index into an $RGB\alpha$–table. Given 8-bit quantities, this requires 1KByte (on-chip) storage.
A more versatile, two-dimensional opacity look-up table is accessed using the gradient magnitude G (or some measure of it) and the function value [10]. For low-cost implementations, gradient components are precomputed at the grid locations and stored together with the function value F as 32-bit voxels (e.g., $G_zG_yG_xF$ in 8 bit precision each) in the texture memory. The interpolated gradient components are squared and added (if the

true magnitude is desired, a fast and compact square root unit as described in [8] can be used). G^2 and F are then used as pointers into an adjusted opacity table. If three square units are already too expensive, the sum G^* of the components (L_1-norm) could be used as index. This reduces the hardware expenses to just two adders at the costs of increased classification uncertainty. The opacity table can be stored in the frame buffer, requiring one external access per raypoint. Alternatively, a reduced-precision table can be kept on-chip, using a number of high-order bits (e.g., 5 for a 1KByte table) of G, G^2 or G^* and F as pointers, and the remaining low-order bits for bi-linear interpolation. This interpolation could be done using existing circuitry in a second pass, or by providing dedicated interpolation units.

For low-cost shading, white light sources at infinity and a constant viewing vector $\vec{V}$ are assumed. Then, for a given scene, the only variable in commonly used illumination models is the gradient. The Phong illumination model can then be simplified to:

$$K = C \cdot k_d \cdot \sum_m (\vec{G} \bullet \vec{L}_m) + k_s \cdot \sum_m (\vec{G} \bullet \vec{H}_m)^n = C \cdot I_d + I_s$$
$$\text{where} \quad \vec{H}_m = \frac{\vec{V} + \vec{L}_m}{|\vec{V} + \vec{L}_m|} \quad \text{and } K,C = \{R,G,B\} \tag{6}$$

The sums I_d and I_s over all light sources $\vec{L}_m$ can be precomputed for all gradient orientations in reduced precision, e.g., using only the 5 MSBs of each component, and stored as a 3D intensity map in the frame buffer. I_s must be recomputed whenever the observer moves, both I_d and I_s whenever a light source is moved. This limits the number m of light sources in practice, although we can exploit the fact that the I_s-elements are mostly zero except within a narrow cone for each light source.

During rendering, the high-order bits of the gradient components are used to access this map. The remaining low-order bits could be used to tri-linearly interpolate the light intensity, again either using the existing circuitry or dedicated units. This method requires access to either one or eight data elements per raypoint. Finally, the reflected raypoint color K is computed using three multipliers and three adders.

It should be noted that modern multi-texturing units may provide two four-channel tri-linear interpolators, potentially with a cache in front of both. These interpolators could be placed into a pipeline for concurrent interpolation of voxels, opacity and intensity elements. Depending on the cache-structure, a high raypoint processing rate might still be achievable despite the additional table look-ups.

8 CONCLUSIONS

As the simulations show, a 3D-texturing unit equipped with the TRIANGLECASTER-extensions can provide real-time volume rendering at very low additional costs. Further low-cost extensions for classification and shading as outlined in the previous section can be used for sophisticated volume visualization in a standard, surface-oriented graphics system. Then, dedicated volume rendering accelerators are no longer necessary, even more so because the combined rendering of polygonal and volumetric objects is much easier in such an integrated system.

9 ACKNOWLEDGMENTS

The author would like to thank Tom Malzbender, Fred Kitson and Byron Alcorn for supporting this work. The suggestions of the anonymous reviewers have been very helpful in preparing the final paper. The use of the opaque-pixel flags has been inspired by one of the comments in the reviews.

10 REFERENCES

[1] **Kurt Akeley,** *"RealityEngine Graphics"*, Proceedings SIGGRAPH 93, pages 109 - 116

[2] **ATI Technologies Inc.,** *"ATI RAGE 128 Chip Information"*, http://www.atitech.com/ca_us/technology/hardware/rage128.html

[3] **Uwe Behrens, Ralf Ratering,** *"Adding Shadows to a Texture-Based Volume Renderer"*, Proceedings 1998 Symposium on Volume Visualization, pages 39 - 46

[4] **Brian Cabral, Nancy Cam, Jim Foran,** *"Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware"*, Proceedings 1994 Symposium on Volume Visualization, pages 91 - 97

[5] **Timothy J. Cullip, Ulrich Neumann,** *"Accelerating Volume Reconstruction with 3D Texture Hardware"*, Technical Report TR93-027, University of North Carolina at Chapel Hill, 1993

[6] **Frank Dachille, Kevin Kreeger, Baoquan Chen, Ingmar Bitter, Arie Kaufman,** *"High-Quality Volume Rendering Using Texture Mapping Hardware"*, Proceedings 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware, pages 69 - 76

[7] **T. Günther, C. Poliwoda, C. Reinhart, J. Hesser, R. Männer, H.-P. Meinzer, H.-J. Baur,** *"VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine"*, Proceedings 9th Eurographics Workshop on Graphics Hardware, 1994, pages 103 - 108

[8] **Günter Knittel,** *"Proven - Prompt Vector Normalizer"*, Proceedings 6th IEEE ASIC Conference, 1993, pages 112 - 115

[9] **Jürgen Hesser, Reinhard Männer, Günter Knittel, Wolfgang Straßer, Hanspeter Pfister, Arie Kaufman,** *"Three Architectures for Volume Rendering"*, Proceedings 1995 Eurographics Conference, Computer Graphics forum, Vol. 14, No. 3, pages 111 - 122

[10] **Marc Levoy,** *"Display of Surfaces from Volume Data"*, IEEE CG & A, Vol. 8, No. 3, 1988, pages 29 - 37

[11] **Mitsubishi Electric,** *"Volume PRO™ Technology at a Glance"*, http://www.3dvolumegraphics.com/3dvolumegraphics/product/index.htm

[12] **NVIDIA,** *"RIVA TNT2 Product Description"*, http://www.nvidia.com/3Dgraphics/features.html

[13] **S3 Inc.,** *"Savage4 - AGP4x for the Volume Mainstream PC Markets"*, http://www.s3.com/savage4/index.htm

[14] **Silicon Graphics, Inc.,** *"OpenGL® Volumizer Programmer's Guide"*, 1998

[15] **Lisa M. Sobierajski, Ricardo S. Avila,** *"A Hardware Acceleration Method for Volumetric Ray Tracing"*, Proceedings '95 Visualization Conference, pages 27 - 34

[16] **Allen Van Gelder, Kwansik Kim,** *"Direct Volume Rendering with Shading via Three-Dimensional Textures"*, Proceedings 1996 Symposium on Volume Visualization, pages 23 - 30

[17] **Rüdiger Westermann, Thomas Ertl,** *"Efficiently Using Graphics Hardware in Volume Rendering Applications"*, Proceedings SIGGRAPH 98, pages 169 - 177

[18] **Orion Wilson, Allen Van Gelder, Jane Wilhelms,** *"Direct Volume Rendering via 3D Textures"*, Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, 1994