The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics

Michael Deering[*] *Stephanie Winner*[†]
Bic Schediwy[‡] *Chris Duffy* *Neil Hunt*

Schlumberger Palo Alto Research, 3340 Hillview Avenue, Palo Alto, CA 94304

Abstract

Current affordable architectures for high-speed display of shaded 3D objects operate orders of magnitude too slowly. Recent advances in floating point chip technology have outpaced polygon fill time, making the memory access bottleneck between the drawing processor and the frame buffer the most significant factor to be accelerated. Massively parallel VLSI systems have the potential to bypass this bottleneck, but to date only at very high cost. We describe a new more affordable VLSI solution. A pipeline of *triangle processors* rasterizes the geometry, then a further pipeline of *shading processors* applies Phong shading with multiple light sources. The triangle processor pipeline performs 100 billion additions per second, and the shading pipeline performs two billion multiplies per second. This allows 3D graphics systems to be built capable of displaying more than one million triangles per second. We show the results of an anti-aliasing technique, and discuss extensions to texture mapping, shadows, and environment maps.

CR Categories and Subject Descriptors: B.2.1 [Arithmetic and Logic Structures]: Design Styles - pipeline. C.1.1 [Computer Systems Organization]: Single Data Stream Architectures - pipeline processors. I.3.1 [Computer Graphics]: Hardware Architecture - raster display devices. I.3.3 [Computer Graphics]: Picture/Image Generation - display algorithms. I.3.7 [Computer Graphics]: 3D Graphics and Realism - color, shading, shadowing and texture, visible lines / surface algorithms.

Additional Keywords: real-time image display, triangle processor, interpolation, hardware lighting models, shading, graphics VLSI.

* Current address: Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

† Current address: Apple Computer Inc., 20525 Mariani Avenue, Cupertino, CA 95014.

‡ Current address: Hewlett Packard Labs (3-U), 1501 Page Mill Road, Palo Alto, CA 94304.

1 INTRODUCTION

Computer graphics has become an integral component of the modern concept of a general purpose computer. The graphical metaphor is more intuitive, flexible, and efficient than the old text-only man machine interface. Also, much of the usefulness of modern computing is the ability of the computer to simulate the real world to more and more exacting degrees. The graphics interface must support this, by portraying the three dimensional dynamic world more and more accurately.

Currently affordable architectures for realistic display of three dimensional objects have severe constraints on scene complexity and display rates. The improvements needed are far beyond the factor of two or three typical of incremental improvements on existing techniques. We re-examined the whole process of image generation, looking for an improvement of at least an order of magnitude over conventional state of the art systems.

After describing the prior art, we will introduce the concept of the *Triangle Processor* and *Normal Vector Shader* chips, and show how they can be configured into a high performance 3D display system. Certain important details of the architecture and chips are examined in more depth, followed by a discussion of extensions to support more sophisticated effects, including anti-aliasing and texture maps.

2 PRIOR WORK

Most commercial 3D graphics systems are based on DRAM Z-buffers. Until very recently, almost all reasonable cost systems (less than \$100 000) rasterized polygons one pixel at a time, with read-modify-write cycle times of approximately one micro-second. Coupled with geometry transform systems capable of processing no more than 10 000 triangles a second, such systems take many seconds or even minutes to draw complex scenes containing large numbers of small triangles. This is so even though most modern systems include some form of VLSI support [2, 13]. Recently announced systems employ parallelism to rasterize polygons several pixels at a time [14]. But as detailed in Appendix A, the pixel write efficiency does not scale linearly, and overall performance is increased by less than an order of magnitude.

To get around the Z-buffer write limit, a number of massively parallel VLSI systems have been proposed. Most have an array of intelligent pixel processors, through which edges of polygons are broadcast; each pixel processor examines the polygon geometry, and sets its pixel color to the polygon's color if the polygon both contains this pixel *and* has a closer Z value than any other polygon previously containing this pixel.

One of the first such systems is Fuchs' Pixel-Planes system [5, 11]. Here a processor is dedicated to each pixel in the display screen. The advantage is that any n-sided polygon can be rendered in constant time, without regard to its area. The disadvantages include the relatively large amount of time taken to enter the geometry for each polygon (tens of microseconds), and the large amount of silicon required (several thousand chips for 1024 × 1024).

The SLAM architecture [4] combines a static RAM for 2D pixel storage with a 1D run-length pixel fill hardware unit on the same custom IC. This is like a Pixel-Planes chip with time multiplexed pixel fill hardware. While this results in better silicon density, the improvement is still not large enough to make 1024 × 1024 72-bit pixels practical, and is better suited to limited pixel depth 2D applications. Other disadvantages include the need to slice up geometry into 1D runs *outside* the SLAM chips, and rendering time complexity linearly proportional to the height of a polygon.

The Super-Buffer [6] dispenses with the 2D array concept, and uses a single scan line of pixel processors. This finally reduces the number of replicated custom chips to a reasonable count, but adds the requirement to pre-sort and buffer all polygons by their first active scan line in the displayed image. The rendering time is proportional to the height of the triangles, and (as in the SLAM system) the data for a given polygon must be externally sequenced and entered repeatedly into the chips until the polygon has been rendered. Nishizawa *et al.*[10] describe another VLSI implementation of this concept.

The lighting model is also an important issue. Most high speed hardware implementations rely upon simple linear interpolation of color between vertices. More comprehensive lighting model computations have not been integrated into the VLSI solution.

The systems discussed above all associate processors with pixels; an alternative is to associate processors with polygons. Many early flight simulator architectures had an architecture of the latter form, as well as an unpublished system of Cohen and Demetrescu. Our system fits into this category.

3 THE TRIANGLE PROCESSOR CONCEPT

The key concept is that of the *Triangle Processor*, a processor dedicated to the rasterization of a single triangle. A number of these processors are connected in series to form a *triangle pipe*, into which a raster ordered stream of "blank" pixels is fed. Each Triangle Processor is responsible for one triangle in the image. If a received pixel falls within the triangle, the processor may substitute triangle specific data; otherwise the pixel is passed along un-altered.

Each Triangle Processor contains the value of the normal to the surface and Z depth at each vertex of the local triangle, and also color data. It generates point samples of these values for each pixel position within the triangle by bi-linear interpolation between the vertex values.

Triangle Processors are designed only to overwrite received pixels that are further back in Z than the interpolated Z value for their local triangle. Thus the triangle pipe implements triangle rasterization with depth sort.

Triangle parameters are loaded by sending data down the triangle pipe in specially marked packets between scan lines of "blank" pixels. They are loaded into the first available Triangle Processor encountered.

A Triangle Processor can only handle one triangle at a time, but since most triangles cover only a relatively small portion of the image, it is not necessary that a Triangle Processor remain allocated to a single triangle for the whole image. When the last pixel in a triangle has been rendered, its processor becomes available for re-use with another triangle, whose parameters may be loaded any time after the current scan line. Thus during the processing of an image, a single Triangle Processor may be used to render a number of triangles which do not overlap in the y direction.

Because of the extremely deep pipelining, the processors at the end of the pipeline are operating several thousand cycles behind those at the front. Thus at any point in time, the pipeline contains pixels from several different scan lines, as well as the packets of new triangle data sandwiched between them. Using the same pipeline for loading the new triangle data as well as rasterizing solves the problem of synchronizing the loads to the different processors in a very elegant manner.

4 THE NORMAL VECTOR SHADER

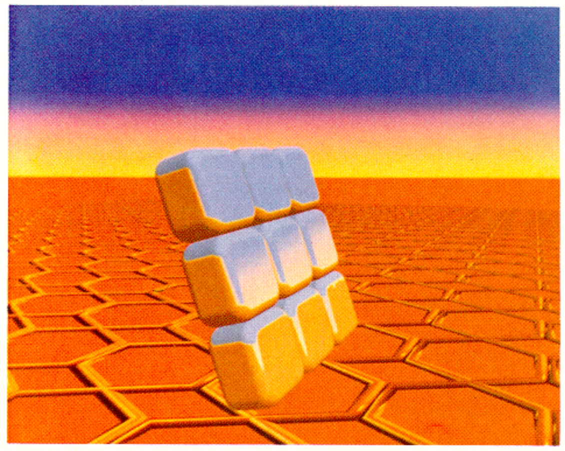

Figure 1: Nine rounded cubes with simple environment mapping.

To complete the rendering process, a shading pipeline applies a lighting model to the stream of normal vector values and other data from the triangle pipe. This second pipe is comprised of "Normal Vector Shader" (NVS) chips, which support a full multiple light source Phong illumination model, which is applied *independently* to each pixel. Since the majority of polygons are only a few pixels across, algorithms such as that of Bishop and Weimer [1], which take 10 pixels per scan line to break even, were deemed too slow. The realism of the image is further enhanced by also interpolating the viewpoint vector to each pixel within the NVS chips, as can be seen in the reflection of the environment in Figure 1 (rendered at 1280 × 1024). This technique

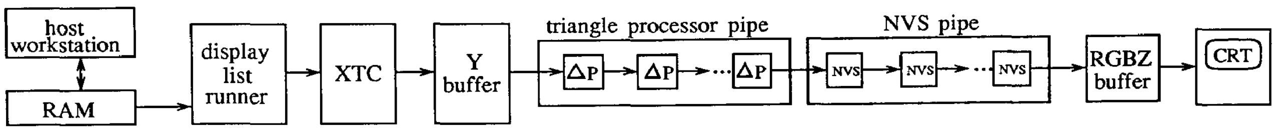

Figure 2: Block diagram of the GSP-NVS system.

avoids the "flat flash" problem associated with Phong shading of flat surfaces by planar light sources.

5 SYSTEM OVERVIEW

We now describe a complete graphics system based on "graphical signal processing" with "normal vector shading" (GSP-NVS), which achieves a processing rate of one million triangles per second. A block diagram of the overall system is shown in Figure 2. Up to the Y-buffer, the processing is fairly conventional, although the computations must be performed at a rate of several million triangles per second.

A display list runner (DLR) extracts geometry and display commands from the memory of the host workstation. The DLR handles viewing matrix manipulation, simple subroutine jumps, picks, bounding box tests, mode bit settings, and so on, but passes triangle and line stroke (vector) commands *to the next stage without interpretation*, keeping the overhead to a minimum.

The transform, clip, and set-up stage (XTC) performs the usual calculations on triangles and vectors. The resulting 448 bits of "pre-digested" triangle data are then passed to the Y-buffer. A much higher throughput (nearly a Gigaflop) than past systems is required here to support the rest of the architecture.

The Y-buffer is organized as 1024 (or more) linked lists, one for each scan line on the display. The Y-buffer input process accepts triangles from the XTC, sorts them into bins indexed by their first active scan line number, and holds them in DRAM storage until all of the triangles in a given frame have been buffered. Then the output process feeds these triangles into the triangle pipe for rasterization, but now in scan line order.

The Y-buffer also sequences the rasterizing within the triangle pipe. First, data representing all the triangles that become active on the current line are sent into the triangle pipe. Then the Y-buffer sends (1280) "blank" pixels for one scan line into the triangle pipe. This is repeated for all the scan lines in the frame (1024).

The triangle pipe accepts the triangle data, and converts the "blank" pixels into a stream of rasterized (and depth sorted) pixels in scan line order, as described above. Once the data for a triangle has been entered into the pipe, it will remain lodged in a Triangle Processor somewhere within, without need for any additional control, for as many scan lines as the triangle is tall.

The NVS pipe takes the rasterized output of the triangle pipe, and converts the surface normal vector pixels to RGB pixels. These are finally stored in an RGB frame buffer prior to display on a CRT.

6 SYSTEM PROPERTIES

The GSP-NVS system has several advantages. The amount of silicon required by the custom rendering pipelines is realistic, and indeed is less than needed elsewhere in a balanced system. This is partly due to the compact layout of the Triangle Processor cell, allowing as many as ten triangle processors per (inexpensive) chip. The hardware is not restricted to any fixed size image for output.

To a first approximation, the rasterization time for a complete image is the time taken to load all the triangle data, *plus the time to send through a screen-full of pixels*. Thus in many normal cases, the rasterization time is *independent* of the sizes of the triangles. The load time is short, because each triangle need only be entered into the pipe once, and special care has been taken to make this entry as fast as possible (8 cycles of 50ns). The time to send pixels into the pipe is also very small, at 50ns per pixel. Typical load time might be 20% of the rasterization time.

There are some disadvantages. As is the case with the Super-Buffer, a Y-buffer is required to pre-sort the geometry. Unlike the processor-per-pixel architectures, geometry overflow is possible if too many triangles are active on the same scan line. This is not a fundamental limit, as the pixels can be passed through the triangle pipe multiple times, effectively increasing the number of Triangle Processors available.

7 TRIANGLE PROCESSOR INTERNAL ARCHITECTURE

The feasibility of a machine with a processor per triangle is dependent on such processors being relatively inexpensive. We have designed a complete Triangle Processor in full custom CMOS VLSI, using less than 25 000 transistors; prototype chips containing one triangle processor have been fabricated, tested and shown to be fully operational at expected speed. Modern VLSI densities allow upwards of ten such processors to be instanced onto a single die; furthermore, as VLSI densities increase, additional Triangle Processors can be added to the chip with little effort and *without* modifying the chip pin-outs. Because these chips are made to be directly wired together in a linear pipe (with no support chips required), upwards of 1 000 Triangle Processors occupy less than a square foot of PC board space (assuming 8 processors per chip in 1.2μ technology).

Figure 3 shows the external data flow between individual Triangle Processor chips, and the internal data flow between Triangle Processors within the same chip. The x pixel location is locally generated within each chip by the G-unit; consequently, this value does not have to be passed between the chips, reducing the pin count. Another factor of two reduction in pin count is obtained by double clocking

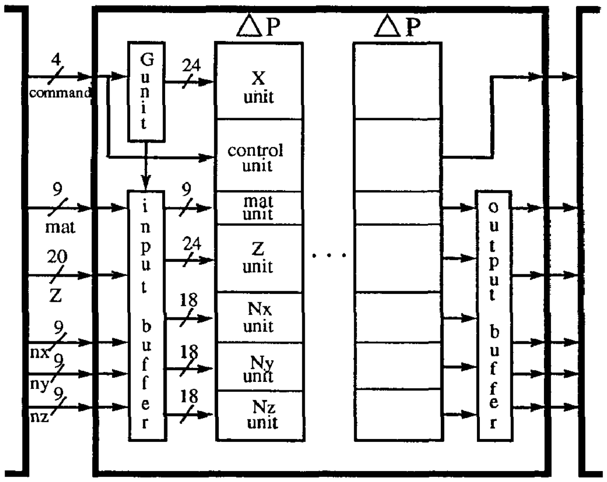

Figure 3: External data flow and internal organization of the Triangle Processors chip. (Note that only 30 data input pins are needed for the 60 signals shown, as the pins are double clocked.)

the I/O pins*, enabling the Triangle Processors and NVS chips to be housed in 68-pin leaded chip carrier packages.

Each Triangle Processor is broken down into a number of function units, also shown in Figure 3. Each unit operates on a different component of the pixel value flowing past. The X channel propagates the 24-bit fixed point number indicating the x location of the current pixel. The Z channel also propagates a 24-bit fixed point number, representing the z depth generated for the current pixel. The 18-bit N channels (N_x, N_y, and N_z) propagate the interpolated (denormalized) surface normal vector. The M channel propagates the associated "material index" (information about color and other triangle surface properties). The processor control unit interprets the command instructions, which sequence the loading of initial values and incremental parameters into the function units, and control the rasterization of the image.

7.1 The X-Unit.

Figure 4 shows the X-unit, which interpolates the points where the edges of the triangle cross each scan line in turn. Before the first scan line on which the triangle is active, x coordinates of the left and right edges of the triangle, and y-increment values are loaded. At the end of each scan line, the X-unit updates the left and right x coordinates by adding the y increment, thus obtaining the coordinates where the edges cross the *next* scan line.

Also, a counter is decremented; when the counter goes negative, either the current triangle has expired, or one of the edges has reached a vertex. In the former case, the triangle processor becomes passive, and awaits the next packet of new triangle data to arrive. If a vertex has been reached on one of the edges, new initial and incremental values are set for that edge, and also a new counter value, from *local* backup registers. Whether it is the left or right edge which changes direction is known at set-up time, and indicated by a triangle "type" bit in the data initially loaded for the triangle.

The X-unit continuously compares the pixel x coordinate (which increases on each incoming pixel) with the x positions of the left and right edges of the local triangle on the current scan line. (The values of x are generated by a single G unit on each chip, and are then pipelined through all the X-units on that chip.) Other units of the processor are signalled whenever the current pixel falls within the local triangle.

* Data is transferred on both rising and falling phases of the clock, using each pin twice per clock cycle.

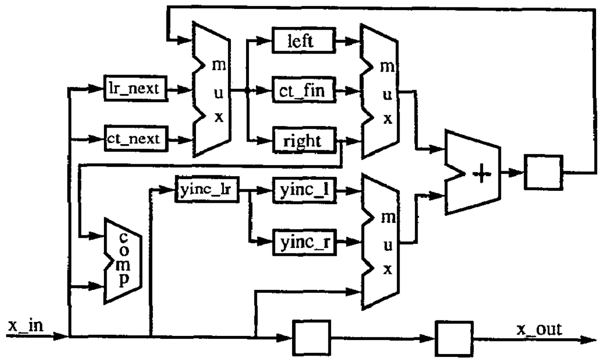

Figure 4: Schematic of the X-unit, used to track left and right edges of triangles, relative to the current scan line.

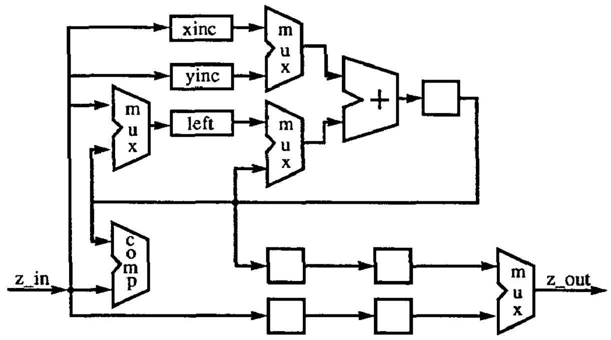

Figure 5: Schematic of the Z-unit, used to interpolate and compare Z depth values.

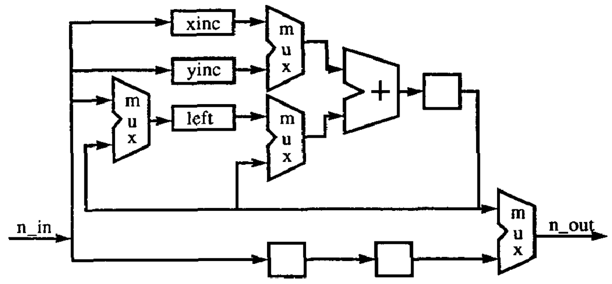

Figure 6: Schematic of the N-unit, used to interpolate surface normal vectors.

7.2 The Z-Unit.

Figure 5 shows the Z-unit schematic, which interpolates the z depth in both x and y directions across the triangle. The data loaded for a triangle before the scan line on which the triangle first becomes active is the initial z value, and also x- and y-increments. At the start of each scan line, the z value is stored in the left register. On every pixel clock cycle, the x-increment is added to the current z value, interpolating triangle surface depth from left to right across the image. At the end of each scan line, the y-increment value is added to the stored z value from the left register, and this is used for subsequent x-interpolation on the next scan line. In this way, the z depth at each pixel in the triangle is computed.

Note that the interpolation is across the *entire* scan line, regardless of where the triangle is actually active. This saves having to activate the interpolation part way through a scan line, without affecting the interpolated value within the triangle in any way*.

Each Z-unit compares the z value input from the preceding Triangle Processor to that locally computed. When the current pixel falls within the local triangle (as determined by the X-unit), *and* when the z depth of the local triangle is less than that for triangles computed by preceding Triangle Processors for this pixel, the local triangle "wins", and will be visible in the image (unless some other triangle later in the pipeline subsequently wins over it). For a winning triangle, the interpolated z depth at this pixel is propagated to the next Triangle Processor in the pipeline; otherwise, the previous best z from the preceding Triangle Processors will be passed forward unchanged.

If a pixel is unclaimed by any triangle, the z value is undefined, and is usually set to the back clipping plane value.

7.3 The N-Unit.

Figure 6 shows a schematic of an N-unit. Three such units N_x, N_y, and N_z are used to interpolate the surface normal vector **N** over the triangle. Before the triangle becomes active, an initial **N** vector is loaded, along with x- and y-increments. The x-increment updates the **N** vector across each scan line, while the y-increment updates the **N** vector at the end of each scan line, just as in the Z-unit. The output multiplexer selects the locally interpolated **N** normal vector if the local triangle pixel "wins", and is to be substituted; otherwise the input **N** normal value from the previous best triangle is propagated.

The value interpolated is the normal vector to the surface of the winning triangle at the point center of the current pixel. Implementing the interpolation in this fashion ensures that all triangles are uniformly sampled and avoids a number of aliasing artifacts and "holes" in the image.

7.4 The M-Unit.

The M-unit latches the 9-bit material index of the local triangle during load time. At each pixel, the input material index is propagated, unless the local triangle wins, in which case the stored material index is substituted.

* Small triangles having large curvatures have very large increment values; this is not a problem, as we simply subtract enough from the initial value, allowing for the overflow wrap-around which may occur, so that the value is correct within the area of the triangle.

7.5 Commands.

The Triangle Processor control unit sequences the N, Z, X, and M-units in response to commands received via the command data bus. Only nine commands are needed:

RESET	Global reset. Force all Triangle Processors to state free.
ENABLE	Global enable. Set the enable bit in all Triangle Processors.
IDLE	No operation. Used to pad processing.
NEW	Header of an eight word packet of new triangle data.
SOL	Start of line. Used to mark an eight word header starting rasterization for each line.
EOL	End of line. Used to terminate rasterization of each line.
RAZ	Rasterization command. Represents a background pixel to be processed.
RAZD	Rasterization command with data. Represents a pixel previously claimed by some triangle.
EXT	External commands. Normal Vector Shader commands and other user defined non-Triangle Processor commands.

7.6 Additional Features.

- The Triangle Processor includes support for simple 2×2 "screen-door" translucency.
- Line strokes can be displayed efficiently by converting them to parallelograms (a special case of a triangle with parallel sides, with the point cut off prematurely).
- Because the fixed-point bit position is externally selectable, the 24-bits of accuracy in x and y values allow a large variety of screen address ranges. Note that the internal registers and data paths of the Triangle Processor chip have a few more bits than the width of the data paths off-chip, to maintain interpolation accuracy, while keeping the pin-count down.
- A number of features facilitate testing of the chips. These include the ability to map out defective processors, which can reduce the cost of building the chips, as those with some fabrication defects are still usable.

8 SIMPLIFIED LIGHTING MODEL

The GSP-NVS lighting model is similar to that used in most software based polygon display systems, but is more sophisticated than any other video rate hardware systems to date. Most high speed graphics systems only apply their lighting model at the vertices of polygons, obtaining an RGB value for each vertex. These color values are interpolated across the face of the polygon (Gouraud shading). We apply our full Phong lighting model to each pixel, for inherently superior shading. Our "Normal Vector Shader" (NVS) chips implement, in silicon, all interior shading methods, and full lighting and depth cueing equations, as specified by the PHIGS+ proposal [15], except point and cone light sources. Five planar, colored light sources are simultaneously supported at full speed.

Given as input the un-normalized surface normal vector $\mathbf{N}'$, and the internally interpolated un-normalized viewpoint vector $\mathbf{V}'$, we first compute re-normalised values $\mathbf{N}$ and $\mathbf{V}$, and also a reflection vector $\mathbf{R}$:

$$\mathbf{N} = \frac{\mathbf{N}'}{\sqrt{\mathbf{N}' \cdot \mathbf{N}'}}, \quad \mathbf{V} = \frac{\mathbf{V}'}{\sqrt{\mathbf{V}' \cdot \mathbf{V}'}}, \quad \mathbf{R} = 2(\mathbf{N} \cdot \mathbf{V})\mathbf{N} - \mathbf{V}.$$

From these values, the **RGB** color is computed:

$$\mathbf{RGB} = \sum_{i=1}^{i=5} [(\mathbf{N} \cdot \mathbf{L}_i + la_i)\mathbf{mc} + (\mathbf{R} \cdot \mathbf{L}_i)^{sp}]\,\mathbf{lc}_i.$$

Here $\mathbf{L}_i$, $\mathbf{lc}_i$, and la_i are the normal to, color of, and ambient component, respectively, of the i^{th} light source, and **mc** and *sp* are the color of, and specular power of the material under illumination, looked-up with the material index of the pixel. The square root, power function, and specular reflection coefficient (not shown) are implemented via on-chip ROM tables. It is beyond the scope of this paper to describe all the details of the full lighting model, which include depth cueing, intensity of specular illumination, built-in simple environment map (including reflections – as seen in Figure 1), and hooks for texture maps, shadows, and local light sources.

9 LIGHTING MODEL CHIP

The illumination pipe of the GSP-NVS system consists of 16 identical NVS chips, with pin-outs *identical* to the Triangle Processors chip. Each is capable of applying the entire five light source lighting model to a single pixel (surface normal vector $\mathbf{N}'$ + depth z + material index M) in 16 clock cycles. Round robin scheduling enables the 16 chips in the pipe to shade a stream of pixels at the full clock rate.

Internally the NVS chips are heavily pipelined, and in fact take 64 clock cycles to shade each pixel. However, four pixels can be at various stages of shading at the same time, providing the 16 clock cycle throughput per chip. Figure 7 displays the specialized vector function units of the NVS chip, which are the key to the efficiency of the NVS pipeline (two billion multiplies per second with 16 chips). They allow for efficient area layout, as well as simplified scheduling. An alternative to building a custom lighting model engine would be to use a general purpose DSP chip. However, such chips typically contain only a single scalar function unit, and for a given area of silicon, will tend to be at least an order of magnitude slower than our custom chips. For example, the DSP32 chips employed in the Pixel Machine [9] individually run at 10 million multiply-adds per second, and it would take *at least* 200 of these chips (with several support chips each) to perform the equivalent computation.

10 TRIANGLE PIPE OVERFLOW

The previous discussion has assumed that there were a sufficient number of Triangle Processors to handle all the triangles active on a particular scan line; in other words, no overflow. Our strategy for dealing with overflow is to always detect it *before* it occurs. This allows us to split the problem into two tasks: overflow detection and overflow correction.

For detection, we must know the number of Triangle Processors in the **free** state at any time. To this end, the Y-buffer keeps a **free** counter and an **auxiliary** table. The **free** counter represents the number of Triangle Processors known to be **free** (initially all of them), and the **auxiliary** table records the number of Triangle Processors scheduled to become **free** on a particular scan line. Whenever a new triangle is to be entered into the pipe, the **free** counter is tested: if it is zero, the triangle is *not* entered, and overflow processing commences; otherwise, the counter is decremented, showing one less free Triangle Processor. When a triangle is successfully entered into the pipeline, the scan line on which that triangle ends is computed (by adding the height of the triangle to the current scan line number), and the entry in the **auxiliary** table corresponding to that line is incremented. After each scan line has been processed, the free counter is incremented by the **auxiliary** table entry for the current line, representing the number of Triangle Processors somewhere within the pipe that have just become **free**.

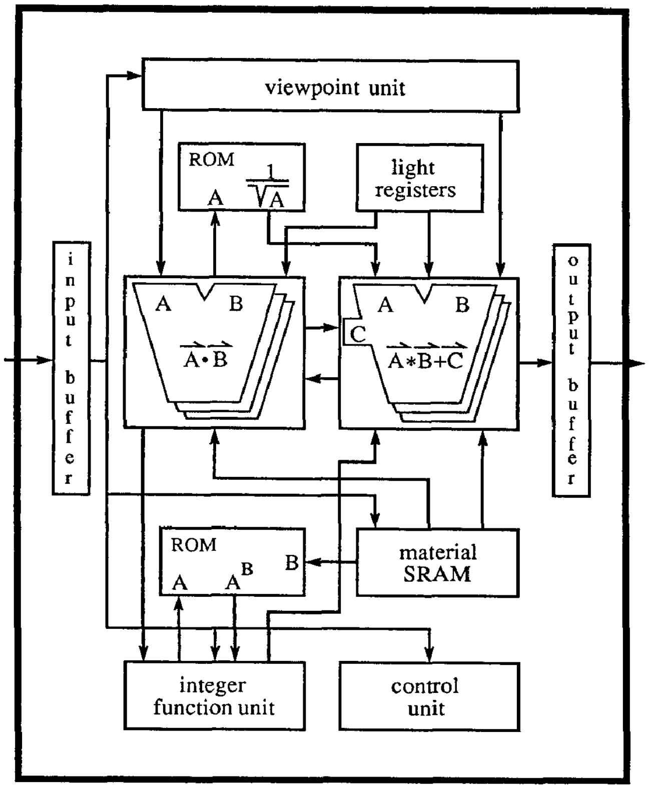

Figure 7: Block diagram of the Normal Vector Shader chip, displaying the internal function units and their (simplified) interconnection.

When an overflow condition is detected, the first action is to stop entering new triangles into the pipe, as there are no free Triangle Processors waiting for them. The overflowed triangles are left in the Y-buffer for a second rendering pass, while processing continues for triangles already in the pipe; the resulting pixels are stored in the RGBZ buffer, along with their associated z depth values, which will be used to merge later passes. Subsequent scan lines may or may not also overflow, depending upon the local triangle population growth. At the end of the frame some of the triangles have been rendered into the RGBZ buffer, while the remainder are still waiting in the Y-buffer. Another pass is made, rasterizing some or all of these overflowed triangles; the z values generated by the pipeline are compared with those stored in the RGBZ buffer from previous passes, the pixel with the nearer z value being stored in the buffer as a result. This is repeated as many times as necessary to rasterize all the triangles in the Y-buffer.

While it appears that the machine will take a factor of two hit if even one triangle overflows, and linearly more for each additional pass required, such is not the case. Overflows tend to be localized to a few areas. Hence, the second pass need not rasterize an entire frame of pixels, but only those scan lines touched by overflowed triangles. System simulations have statistically validated this, showing that small overflows tend to increase rasterization time by less than 20%. Large overflows imply large numbers of input triangles, and the extra rasterization passes are overlapped by extra XTC computation time.

One must have enough Triangle Processors in the system to match the throughput of the XTC section. For non-pathological display lists, there is a balance point where the time taken by the XTC to deliver the triangles will always exceed the time taken by the total rasterization process, including multiple overflow passes. This is because both times grow approximately linearly in the number of triangles to be processed. As a specific example, at one million renderable triangles per second, the balance point is on the order of one thousand triangle processors. The balance point also places an upper bound on the required size of the Y-buffer, limiting costs.

In support of multiple rendering passes, the Y-buffer uses a servoing control algorithm that balances overflows of the triangle pipe, the Y-buffer storage, and the XTC. For display lists small enough to be rendered at frame rates (in other words, no overflows), the Y-buffer double buffers frames of triangle data. For larger display lists, the Y-buffer switches into single buffer mode, and after a high water mark, sets the triangle pipe to rasterizing, even though the XTC is still delivering triangles for the current frame. The assumption here is that multiple passes will be required, so the Y-buffer becomes a quantized FIFO between the XTC process and the triangle pipe. Detailed systems simulations have shown this approach to be viable and capable of sustaining full performance.

11 ANTI-ALIASING

While everyone is for anti-aliasing, few wish to pay large amounts extra for it. Thus versions of the Triangle Processor with circuitry to generate pixel percent coverage data were rejected as too inaccurate when high quality was required, and too expensive when only speed was desired.

The single sample per pixel model used by the Triangle Processors avoids a number of smooth-surface aliasing artifacts in previous hardware implementations. This accuracy allows high quality anti-aliased images to be produced by a variety of oversampling techniques.

Within the system, there are two general anti-aliasing hardware support features. Where speed is essential, the system computes 960 $\times$ 1280 images, and then averages blocks of 2×2 pixels in real-time to NTSC video. When accuracy is required, a stochastic sampling is obtained by filtering together multiple renderings of the same display list, but from sub-pixel jitters in viewpoint position. By using a 24-bit per color component accumulator frame buffer, as many different pseudo-random samplings can be summed together as desired. Within each pass the pixel sample positions are coherent with each other, making the technique somewhat more restricted than software implementations of stochastic anti-aliasing [3].

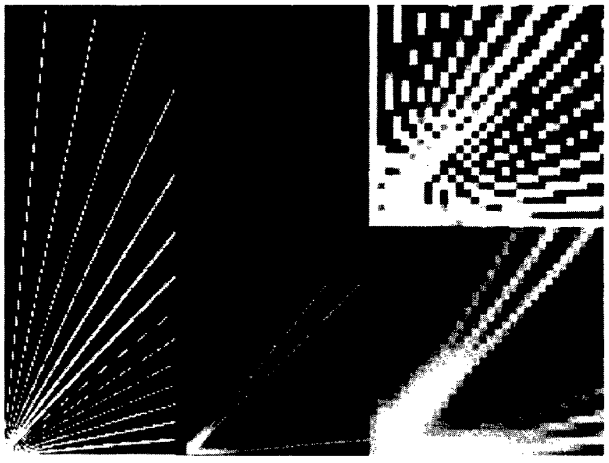

Figure 8: Demonstration of anti-aliasing technique. The image is of a fan of cylinders of random thickness, in the range of 0.6 to 1.6 pixels in width.

Figure 8 displays an example of this technique. The data base is a fan of cylinders of varying thickness, from 0.6 to 1.6 pixels in width. The leftmost image is a single sampling with no anti-aliasing, rendered at 256×96. The upper right insert is a blowup of a portion of this. The middle image is the result of 16 pseudo-stochastic samplings weighted by a Gaussian low-pass filter. To the right is the corresponding blowup of it. This technique was also employed in the making of Figure 1. Other techniques based on stochastic sampling may be implemented on a GSP-NVS system using variations of this approach.

12 SOPHISTICATED EFFECTS

Historically, hard-wired graphics has implied limited rendering models; general purpose machines have had to be used to get most of the recent improvements in graphics algorithms. While machines such as the Pixar [8] and the Pixel Machine [9] seem to cross this line, at heart they are built on top of general purpose processors. Without graphics specific VLSI, the speeds are only fast relative to full general purpose computers. The intent of this section is to show how simple enhancements of the GSP-NVS system can support some very sophisticated options, at speeds orders of magnitude faster than existing hardware. One way or another, such sophisticated options will emerge in hardware systems of the future.

Texture mapping, environment mapping, and shadowing can all be implemented in a related fashion. Each requires an additional frame store, and hardware for a number of transformation functions to generate addresses for this store from rendering information. The data output from the store forms additional lighting model input parameters. These processes must be combined with multiple anti-aliasing passes to sample the data properly.

Texture mapping requires an auxiliary texture map frame store, storing the desired texture(s) at multiple levels of resolution. Three intermediate values a, b, and c, derived from the values of the texture addresses at the corners of the triangles, must be interpolated across the faces of triangles. The final u, v texture map addresses are obtained by performing a perspective division: $u = a/c$, $v = b/c$. The texture map of the appropriate resolution is chosen as a function of the local scale, and can involve a filtered combination of two scales [7]. The three additional interpolations can be performed by a second triangle pipe in parallel with the first, interpolating a, b, and c in place of the surface normal vector. An alternative is to make two passes using a single triangle pipe, once for normals, and then once for a, b, and c, buffering the outputs until needed. The u, v image is sent as a stream of addresses for the texture frame store, resulting in an RGB pixel stream to be sent into the NVS pipe in place of the RGB color indexed by the material tag.

Environment mapping requires the vector computed as the reflectance of the viewpoint vector by the surface normal vector. This value is generated within the NVS chips as part of their normal processing, and a mapping function converts this vector into an environment map address. A simple mapping function is $u = \tan^{-1} R_y/R_x, \quad v = R_z$. The RGB output from the environment map is then averaged into the results of the regular lighting model computation, with a mixing factor proportional to surface reflectivity. (Indeed, a simplified version of environment mapping is implemented within the NVS chips for simple color ramps, as seen in Figure 1.)

For shadows, the vector from each light source to the surface must be computed for each pixel. These vectors are used to generate addresses into precomputed depth buffer tables containing the maximum distance to which the light source reaches in each direction. A comparison of the length of the vector with the stored value is used to enable or disable each light source at each pixel in the regular lighting model computation. (Reeves *et al.* [12] discuss depth buffer shadowing with anti-aliasing.)

Since these look-up process for texture, environment mapping and shadowing involve random pixel addressing, the full pipeline rate of the machine will not be sustained. However, with complex images, the rendering stage normally requires multiple passes, balancing the time taken for the table accesses which are only performed once per output pixel.

13 SIMULATORS

The development of a set of simulator tools was tightly coupled with the evolution of the GSP-NVS architecture. The simulators span the range from a high level simulation fast enough to create short video test sequences, to a low level transistor by transistor simulation of the custom graphics chips. All of these simulators maintain the same numerical precision as the real hardware. We used the simulators to verify the algorithms, and also to generate the performance numbers. All the color images in this paper were generated by the high level simulator, from very complex data bases.

Of course, the ultimate simulation is the real silicon. An incremental approach to the VLSI layout was taken, with individual function units fabricated as separate test chips. This allowed each unit to be fully verified in silicon as the design progressed. Test patterns for the design fragments, and for a complete Triangle Processor, were generated by the software simulators.

14 PERFORMANCE

The GSP-NVS concept and family of VLSI chips can be used to build a variety of high performance graphics systems. We have been designing such a system capable of sustained rendering of one million triangles per second. This system uses twenty 40-Megaflop floating point chips within the transform and clip subsystem to process 1.6 million triangles per second, ensuring adequate throughput at the triangle pipe after various overheads. The display region is split into left and right halves, each with its own 20MHz triangle and NVS pipes. This effectively halves the total time taken to load triangles and rasterize pixels. Thus triangles are loaded at 200ns each, and then the whole region is rendered at 25ns per pixel. The Y-buffer allows transformation and rasterization to be 100% overlapped most of the time.

Figure 9 shows a comparatively simple scene generated by a simulation of the GSP-NVS system. The data base used contained 24 784 triangles, of which 11 327 were visible. Rendered at a resolution of 1280 × 1024, the 1.25 million pixel image would be generated in less than one twentieth of a second. Note that the triangle load time was less than 5% of the total frame time, indicating that the system was running at only a quarter of its capacity. Simulations of larger display lists involving multiple triangle pipe overflows have shown rendering times to scale almost linearly. More aggressive uses of additional rendering pipes can lead to processing rates upwards of five million triangles per second.

15 CONCLUSIONS

We have described a highly parallel 3D graphics architecture based upon a pipeline of VLSI chips, which overcomes the frame-buffer DRAM access bottleneck present in all of today's general commercial systems. Our architecture achieves at least an order of magnitude increase in rendering rate over previous systems, so that much more complex scenes can be rendered in a reasonable time. A further pipeline of dedicated lighting model chips enhances the reality of the output with no loss in speed.

An additional improvement in speed over present systems is achieved by decoupling the transformation process from the rendering process using a large buffer; this keeps both processes running in parallel for the majority of the time. The architecture has been tuned to support the statistical mix of geometry encountered in *real* applications, rather than simplified benchmarks.

Our sampling model is built on a mathematically sound basis, similar to that used in ray-casting techniques. This avoids a number of aliasing artifacts and holes inherent in

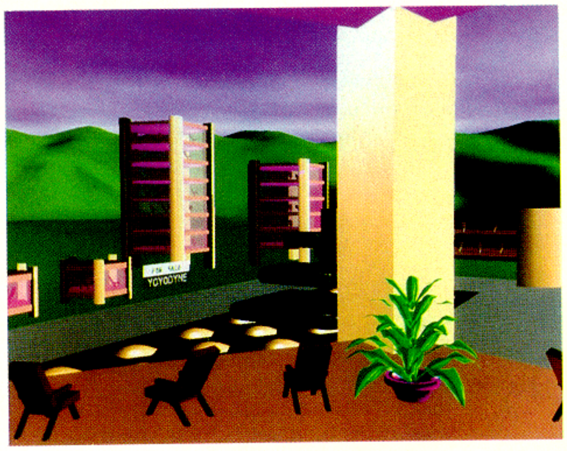

Figure 9: Example of 1280 × 1024 20Hz frame rate performance.

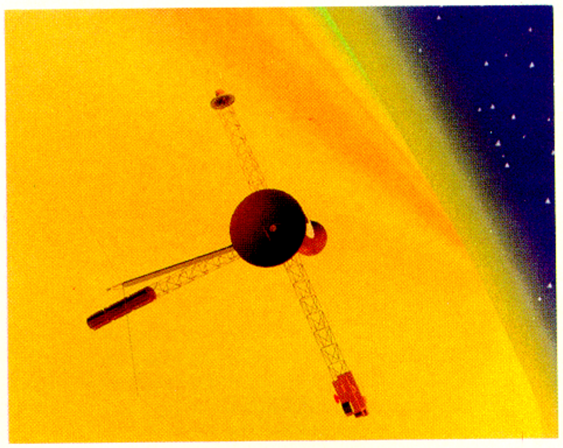

Figure 10: Example of complex mechanical CAD object. Satellite database courtesy of NASA JPL.

many other polygon fill algorithms. It also allows the system to be extended to support a number of sophisticated rendering techniques; for example we have already simulated a form of stochastic anti-aliasing.

It is no longer sufficient to concentrate on only a single aspect of the 3D graphics problem; we believe that to obtain significant improvements for *real* applications, the entire rendering process must be balanced from display list to photon. GSP-NVS represents our attempt to advance the state of the art.

ACKNOWLEDGMENTS

The authors would like to thank the Applicon hardware crew: Brian Schroeder, Tony Barkans, Roger Day, Dorothy Gordon, Tom Durant, Jorge Lach, Mark Turner, Mark Hood, Gene Pinkston, Mike Garrity, Stan Sclaroff, and Peter Masters for their efforts in the development of the system. Glen Stone was responsible for the early layout of the Triangle Processor. We would also like to thank Dick Lyon for comments on the VLSI design, Steve Rubin for the Electric VLSI design system and the support thereof, and Shane Robison for his support of us.

A WHAT'S IN A TRIANGLE ?

The GSP-NVS system supports the RISC primitive of 3D graphics: the triangle. For our system to support non-uniform rational B-splines (NURBS) as a primitive, they would have to be tessellated into triangles at extremely high speed, in excess of one million triangles per second. The computational overhead of doing this to complex objects was considered too high. The major problem with building hardware to directly tessellate NURBS is dealing with the trimming curves.

As is the case with many other complex computational systems, there is great confusion in benchmarking the performance of 3D display systems. There are many assumptions: what percent backface rejection?, clipping enabled?, what lighting model?, area of polygon?, what is a polygon? Typically the benchmark represents a "never to be exceeded" performance number. The benchmark for a particular system usually consists of identical polygons which are sized such that the transformation and clipping time exactly equals the polygon pixel fill time. This is reminiscent of similar "creative" benchmarks of identical length vectors that were used to benchmark vector stroke displays 20 years ago. They provide very little insight as to where and how much to improve the performance of 3D display systems. Thus, one of the first tasks of the GSP-NVS project was to create some more realistic benchmarks.

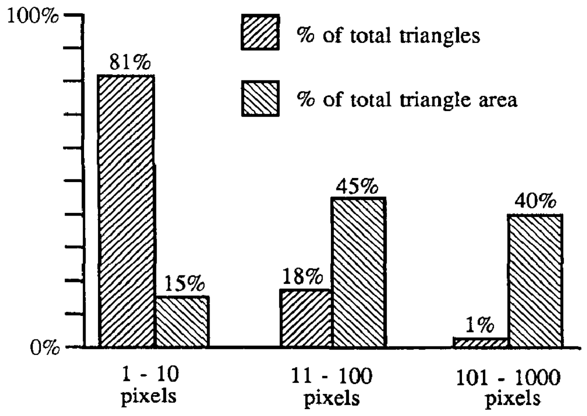

Figure 11: Percentage of triangles and triangle area for the cubes in Figure 1 sorted by the area of the triangle.

Figure 1 displays an image of nine rounded cubes, formed of 9 × 956 triangles, of which 3 986 were non back-facing and were rendered. The triangles in this figure (rendered at 480 × 484) were grouped by their displayed area into three categories, as shown in the chart in Figure 11. In each category, the left bar shows the relative *number* of triangles in the group, while the right bar shows the relative *area* contributed by triangles in that group. The three groups roughly correspond to three classes of polygons: a few large ones (the flat faces of the cubes), a number of long thin ones (the rounded edges), and a large number of very small ones (the rounded corners), of which many are sub-pixel in size. While this scene was somewhat artificial, experiments with a number of other objects from real mechanical CAD data bases bore out similar results. This chart gives part of the reason why real world applications typically run *at least* two to four times slower than the theoretical peak rate of the graphics system. The activity of the transform and clip section is proportional to the height of the left boxes in each group, while rasterization activity is proportional to the height of the right boxes. For any given triangle size, typically only one of the two processes is active. Thus most polygons are either transform bound *or* polygon fill bound, resulting in significant idle time for the other process. This is responsible for nearly a factor of two reduction is speed over the "perfect" case. Additional reductions arise from idle rasterization cycles due to long chains of triangles that are back-facing or outside the clipping window, as well as overhead for starting a new triangle, moving to the next scan line, et cetera.

An increase in graphics performance is more likely to cause users to display more complex objects, rather than the same objects faster. This is the so-called "four second" rule. Figure 10 (1280 × 1024) displays an extremely large CAD object (by present standards). It represents a 70 000 triangle simplification of a 1 000 000 triangle satellite. Two trends appear in this and other similar objects that we have examined: realistic background environments will almost always cover every pixel in the frame at least once; and more detailed, more complex objects will involve large numbers of small polygons. Furthermore, the mid-size polygons tend to be long and very thin, bordering on sub-pixel. From statistics of these objects we have concluded that pixel fill architectures using $n \times m$ block write schemes will be less effective than might be assumed. We simulated a number of different block sizes, applied to the data base of Figure 1 (without ground pattern), and the resulting write efficiencies are given in Table 1.

Total Write Efficiency	
Block Size	Pixels written/cycle
1 × 1	0.69
2 × 1	1.04
2 × 2	1.57
4 × 1	1.41
4 × 2	2.22
4 × 4	3.10
8 × 4	4.04
8 × 8	5.27

Table 1: Efficiency of various parallel write architectures.

The statistics are only for *rendered triangles*. The 1 × 1 number is less than unity because of triangles not containing valid sample points on every scan line. All these numbers would be even lower if back-facing and clipped triangles were to be counted.

References

[1] **Gary Bishop and David M. Weimer.** Fast Phong shading. Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18-22, 1986). In *Computer Graphics*, pages 103–106, 1986.

[2] **James. H. Clark.** The geometry engine, a VLSI geometry system for graphics. Proceedings of SIGGRAPH'82 (Boston, Massachusetts, July 26-30, 1982). In *Computer Graphics*, pages 127–133, 1982.

[3] **Robert. L. Cook.** Stochastic sampling in computer graphics. *ACM Transactions on Computer Graphics*, 5(1):51–72, January 1986.

[4] **Stefan Demetrescu.** High speed image rasterization using scan line access memories. In *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 221–243, Computer Science Press, 1985.

[5] **Henry Fuchs and John Poulton.** Pixel-planes: a VLSI-oriented design for a raster graphics engine. *VLSI Design*, 2(3):20–28, 1981.

[6] **Nader Gharachorloo and Christopher Pottle.** SUPER BUFFER: a systolic VLSI graphics engine for real time raster image generation. In *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 285–305, Computer Science Press, 1985.

[7] **Paul S. Heckbert.** Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986.

[8] **Adam Levinthal and Thomas Porter.** Chap - a SIMD graphics processor. Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23-27, 1984). In *Computer Graphics*, pages 77–82, 1984.

[9] **Leonard McMillan.** Graphics at 820 MFLOPS. *ESD: THE Electronic Systems Design Magazine*, 17(9):87–95, September 1987.

[10] **Teiji Nishizawa** *et al.* A hidden surface processor for 3-dimension graphics. In *Proceedings of ISSCC'88*, pages 166–167,351, 1988.

[11] **John Poulton** *et al.* PIXEL-PLANES: building a VLSI-based graphic system. In *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 35–60, Computer Science Press, 1985.

[12] **William T. Reeves, David H. Salesin, and Robert L. Cook.** Rendering antialiased shadows with depth maps. Proceedings of SIGGRAPH'87 (Anaheim, California, July 27-31, 1987). In *Computer Graphics*, pages 283–291, 1987.

[13] **Roger W. Swanson and Larry J. Thayer.** A fast shaded-polygon renderer. Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18-22, 1986). In *Computer Graphics*, pages 95–101, 1986.

[14] **John G. Torborg.** A parallel processor architecture for graphics arithmetic operations. Proceedings of SIGGRAPH'87 (Anaheim, California, July 27-31, 1987). In *Computer Graphics*, pages 197–204, 1987.

[15] **Andries van Dam** *et al.* PHIGS+ functional description rev. 2.0. July 20 1987. Jointly developed PHIGS+ specification.