Talisman: Commodity Realtime 3D Graphics for the PC

Jay Torborg James T. Kajiya
Microsoft Corporation

ABSTRACT

A new 3D graphics and multimedia hardware architecture, code-named Talisman, is described which exploits both spatial and temporal coherence to reduce the cost of high quality animation. Individually animated objects are rendered into independent image layers which are composited together at video refresh rates to create the final display. During the compositing process, a full affine transformation is applied to the layers to allow translation, rotation, scaling and skew to be used to simulate 3D motion of objects, thus providing a multiplier on 3D rendering performance and exploiting temporal image coherence. Image compression is broadly exploited for textures and image layers to reduce image capacity and bandwidth requirements. Performance rivaling high-end 3D graphics workstations can be achieved at a cost point of two to three hundred dollars.

CR Categories and Subject Descriptors: B.2.1 [Arithmetic and Logic Structures]: Design Styles - Parallel, Pipelined; C.1.2 [Processor Architectures]: Multiprocessors - Parallel processors, Pipelined processors; I.3.1 [Computer Graphics]: Hardware Architecture - Raster display devices; I.3.3 [Computer Graphics]: Picture/Image Generation - Display algorithms.

INTRODUCTION

The central problem we are seeking to solve is that of attaining ubiquity for 3D graphics. Why ubiquity? Traditionally, the purpose of computer graphics has been as a tool. For example, mechanical CAD enhances the designerÕs ability to imagine complex three dimensional shapes and how they fit together. Scientific visualization seeks to translate complex abstract relationships into perspicuous spatial relationships. Graphics in film-making is as a tool that realizes the vision of a creative imagination. Today, computer graphics has thrived on being the tool of choice for augmenting the human imagination.

However, the effect of ubiquity is to promote 3D graphics from a *tool* to a *medium*. Without ubiquity, graphics will remain as it does today, a tool for those select few whose work justifies investment in exotic and expensive hardware. With ubiquity, graphics can be used as a true medium. As such, graphics can be used to record ideas and experiences, to transmit them across space, and to serve as a technological substrate for people to communicate within and communally experience virtual worlds. But before it can become a successful medium, 3D graphics must be universally available: the breadth and depth of the potential audience must be large enough to sustain interesting and varied content.

How can we achieve ubiquity? There are a few criteria: 1) hardware must be so inexpensive that anyone who wants it can afford it, 2) there must be a minimum level of capability and quality to carry a wide range of applications, and 3) the offering must carry compelling content. This paper will treat the first two problems and a novel hardware approach to solving them.

There are two approaches to making inexpensive graphics hardware. One approach is to make an attenuated version of conventional hardware. In the next section we make an analysis of the forces driving the cost of conventional graphics architectures. By mitigating some of these costs, one may obtain cheaper implementations with more modest performance. Over a dozen manufacturers are currently exploring this approach by cutting down on one or another cost factor. The risk of this approach, of course, is that each time one cuts cost, one also cuts performance or quality.

An alternative approach is to look to new architectures that have a fundamentally different character than the conventional graphics pipeline. This is an approach pioneered at the high end by the Pixel Planes project [Fuc89], PixelFlow [Mol92], and various parallel ray tracing machines [Nis83, Pot89]. At the low end, Nvidia [Nvi95] is offering such a different architecture. We present an architecture that very much is in the spirit of this latter path, delivering a high performance, high quality graphics system for a parts cost of $200 to $300.

The second criterion, quality, must be evaluated in terms of the applications and content to be executed by the machine. Here we make a fundamentally different assumption from that underlying the conventional graphics pipeline. We believe that the requirements and metric of performance for a ubiquitous graphics system to be much different than that for a system designed primarily for mechanical CAD. In MCAD the ability to accurately and faithfully display the shape of the part is a strict requirement. The metric of performance is often polygons per second, but ultimate result is frame rateÑa low-cost system will display at a much slower rate than a high-performance system, but both will be able to display the shape accurately with the exactly the same image. One of our central assumptions is that in applications and content for ubiquitous graphics this situation is reversed. In a system to be used as a medium, rather than as a tool, the ability to smoothly convey motion, to be synchronized with sound and video, and to achieve low-latency interaction are critical requirements. We believe the fidelity of the shapes, the

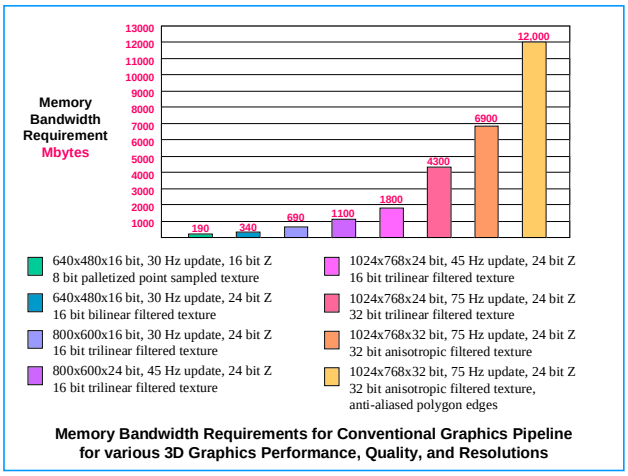
jaytor@microsoft.com kajiya@microsoft.com

precise nature of their geometric relationships, and image quality are performance metrics. In our architecture we have striven to make it possible for one to always be able to interact in real-time, at video frame rates (e.g. 72-85 Hz). The difference between high-cost and low-cost systems will be in the fidelity and quality of the images.

FUNDAMENTAL FORCES

A graphics system designer struggles with two fundamental forces: memory bandwidth, and system latency. To achieve low-cost, a third force looms large: memory cost.

Space considerations do not allow us to detail all the bandwidth requirements for a conventional graphics pipeline. The considerations are straightforward: for example, simple multiplication shows display refresh bandwidth for a 75 Hz, 640x480x8 frame buffer requires 23MB per second, while that for 1024x768x24 requires 169 MB per second. If we add the requirements for z-buffering (average depth complexity of 3 with random z-order), texture map reads with various antialiasing schemes (point sample, bilinear, trilinear, anisotropic), and additional factors imposed by anti-aliasing, we obtain the following chart:



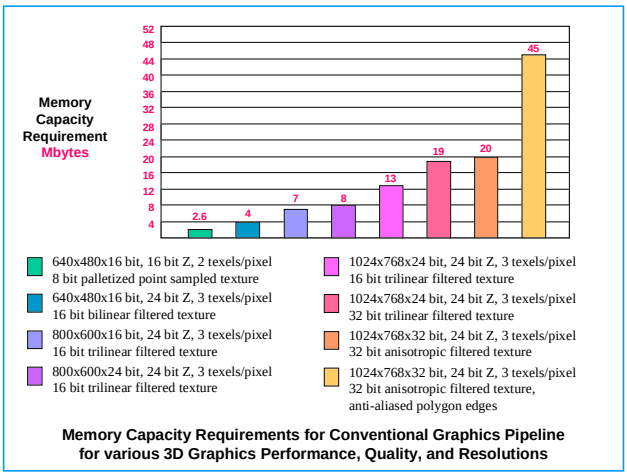
Memory Bandwidth Requirements for Conventional Graphics Pipeline for various 3D Graphics Performance, Quality, and Resolutions

Memory bandwidth is a key indicator of system cost. The left hand two columns indicate where current 3D accelerators for the PC are falling. A full up SGI RE2—a truly impressive machine—boasts a memory bandwidth of well over 10,000 MB per second. Its quite clear that SGI has nothing to fear from evolving PC 3D accelerators, which utilize traditional 3D pipelines, for some time to come.

The second force, system latency, is handled mainly through careful design of the basic algorithms of the architecture, as well as careful pipelining to mask memory latencies.

The third force, memory cost, traditionally has not been of great concern to high-end systems because achieving the aggregate bandwidth has required large amounts of memory. The next chart shows the results of calculating memory requirements for a conventional graphics pipeline with different levels of performance.

Over the last two decades, the drop in price per bit of semiconductor memory has been phenomenal. A look at an early DRAM vs. today's reveals interesting trends.



Memory Capacity Requirements for Conventional Graphics Pipeline for various 3D Graphics Performance, Quality, and Resolutions

Note that although capacity has improved tremendously, latency and bandwidth have not made similar improvements. There is every indication that these trends will continue to hold.

These charts suggest that achieving high-quality imagery using the conventional graphics pipeline is an inherently expensive enterprise. Those who maintain that improvements in CPU and VLSI technology are sufficient to produce low-cost hardware or even software systems that we would consider high-performance today, have not carefully analyzed the nature of the fundamental forces at work.

DRAM Technology Improvements

	1976	1995	Change	Change per Year
Access Time	350 ns	50 ns	7X	10%
Bandwidth (per data pin)	2 Mb/sec	22 Mb/sec	11X	12%
Capacity	4 Kbit	16 Mbit	4096X	50%
Cost per MByte	$ 16,500	$ 23	720X	40%

IMAGE PROCESSING AND 3D GRAPHICS

Although the conventional graphics pipeline uses massive amounts of memory bandwidth to do its job, it is equally clear that much of this bandwidth is creating unused, if not unusable, capacity. For example, the conventional pipeline is fully capable of making every frame a display of a completely different geometric model at full performance. The viewpoint may skip about completely at random with no path coherence at all. Every possible pixel pattern may serve as a texture map, even though the vast majority of them are perceptually indistinguishable from random noise. A frame may be completed in any pixel order even though polygons tend to occupy adjacent pixels.

In our architecture we have sought to employ temporal coherence of models, of motion, of viewpoint, and spatial coherence of texture and display. We have found that this approach greatly mitigates the need for large memory bandwidths and capacities for high-quality systems.

A fundamental technique we have used repeatedly is to replace image synthesis with image processing. That image processing

and 3D graphics have always had an intimate theoretical relationship, is evident to anyone perusing the contents of a typical SIGGRAPH proceedings. Even in high-quality off-line rendering, image processing and composition has served essential functions for many years. But, with a few exceptions like the Pixar Image Computer [Lev84], Regan's image remapping system [Reg94], and the PixelFlow architecture [Mol92] this relationship has not extended into the physical embodiment of hardware.

In a sense, one can view texture mapping as an example of marrying images and 3d graphics early in the pipeline. Segal, et. al. [Seg92] have shown that texture mapping, especially when considered in context with multiple renderings can simulate many lighting effects. We have adopted this idea for the real-time context, calling it multi-pass rendering.

Image compositing and image morphing have been long used in the utilization of temporal coherence—at least in software systems, [Coo87, Che94, Che95, McM95]. Our architecture extends these ideas into the real-time hardware domain, for the case of affine image transformations.

HARDWARE ARCHITECTURE

There are four major concepts utilized in Talisman, these are:

- Composited image layers with full affine transformations.
- Image compression.
- Chunking.
- Multi-pass rendering.

Composited Image Layers

The Talisman hardware does not incorporate a frame buffer in the traditional sense. Instead, multiple independent *image layers* are composited together at video rates to create the output video signal. These image layers can be rendered into and manipulated independently. The graphics system will generally use an independent image layer for each non-interpenetrating object in the scene. This allows each object to be updated independently so that object update rates can be optimized based on scene priorities. For example, an object that is moving in the distant background may not need to be updated as often, or with as much accuracy, as a foreground object.

Image layers can be of arbitrary size and shape, although the first implementation of the system software uses only rectangular shaped layers. Each pixel in a layer has color and alpha (opacity) information associated with it so that multiple layers can be composited together to create the overall scene.

Several different operations can be performed on these image layers at video rates, including scaling, rotation, subpixel positioning, and skews (i.e., full affine transformations). So, while image layer update rates are variable, image layer transformations (motion, etc.) occur at full video rates (e.g. 72 to 85 Hz), resulting in much more fluid dynamics than can be achieved by a conventional 3D graphics system that has no update rate guarantees.

Many 3D transformations can be simulated by 2D imaging operations. For example, a receding object can be simulated by scaling the size of the image. By utilizing 2D transformations on previously rendered images for intermediate frames, overall processing requirements are significantly reduced, and 3D rendering power can be applied where it is needed to yield the highest quality results. Thus, the system software can employ temporal level of detail management and utilize frame-to-frame temporal coherence.

By using image layer scaling, the level of spatial detail can also be adjusted to match scene priorities. For example, background objects (e.g., cloudy sky) can be rendered into a small image layer (low resolution) which is then scaled to the appropriate size for display. By utilizing high quality filtering, the typical low resolution artifacts are reduced.

A typical 3D graphics application (particularly an interactive game) trades off geometric level of detail to achieve higher animation rates. The use of composited image layers allow the Talisman system to utilize two additional scene parameters—temporal level of detail and spatial level of detail—to optimize the effective performance as seen by the user. Further, the Talisman system software can manage these trade-offs automatically without requiring application support.

Image Compression

Talisman broadly applies image compression technology to solve these problems. Image compression has traditionally not been used in graphics systems because of the computational complexity required for high quality, and because it does not easily fit into a conventional graphics architecture. By using a concept we call chunking (described below), we are able to effectively apply compression to images and textures, achieving a significant improvement in price-performance.

In one respect, graphics systems *have* employed compression to frame buffer memory. High end systems utilize eight bits (or more) for each of three color components, and often also include an eight bit alpha value. Low end systems *compress* these 32 bits per pixel to as few as four bits by discarding information and/or using a color palette to reduce the number of simultaneously displayable colors. This compression results in very noticeable artifacts, does not achieve a significant reduction in data requirements, and forces applications and/or drivers to deal with a broad range of pixel formats.

The compression used in Talisman is much more sophisticated, using an algorithm similar to JPEG which we refer to as TREC to achieve very high image quality yet still provide compression ratios of 10:1 or better. Another benefit of this approach is that a single high quality image format (32 bit true color) can be used for all applications.

Chunking

A traditional 3D graphics system, or any frame buffer for that matter, can be, and usually is, accessed randomly. Arbitrary pixels on the screen can be accessed in random order. Since compression algorithms rely on having access to a fairly large number of neighboring pixels in order to take advantage of spatial coherence, and only after all pixel updates have been made, the random access patterns utilized by conventional graphics algorithms make the application of compression technology to display buffers impractical.

This random access pattern also means that per-pixel hidden surface removal and anti-aliasing algorithms must maintain additional information for every pixel on the screen. This dramatically increases the memory size requirements, and adds another performance bottleneck.

Talisman takes a different approach. Each image layer is divided into pixel regions (32 x 32 pixels in our reference implementation) called *chunks*. The geometry is presorted into

bins based on which chunk (or chunks) the geometry will be rendered into. This process is referred to as *chunking*. Geometry that overlaps a chunk boundary is referenced in each chunk it is visible in. As the scene is animated, the data structure is modified to adjust for geometry that moves from one chunk to another.

While chunking adds some upstream overhead, it provides several significant advantages. Since all the geometry in one chunk is rendered before proceeding to the next, the depth buffer need only be as large as a single chunk. With a chunk size of 32 x 32, the depth buffer is implemented directly on the graphics rendering chip. This eliminates a considerable amount of memory, and also allows the depth buffer to be implemented using a specialized memory architecture which can be accessed with very high bandwidth and cleared instantly from one chunk to the next, eliminating the overhead between frames.

Anti-aliasing is also considerably easier since each 32 x 32 chunk can be dealt with independently. Most high-end graphics systems which implement anti-aliasing utilize a great deal of additional memory, and still perform relatively simplistic filtering. By using chunking, the amount of data required is considerably reduced (by a factor of 1000), allowing practical implementation of a much more sophisticated anti-aliasing algorithm.

The final advantage is that chunking enables block oriented image compression. Once each 32 x 32 chunk has been rendered (and anti-aliased), it can then be compressed with the TREC block transform compression algorithm.

Multi-pass Rendering

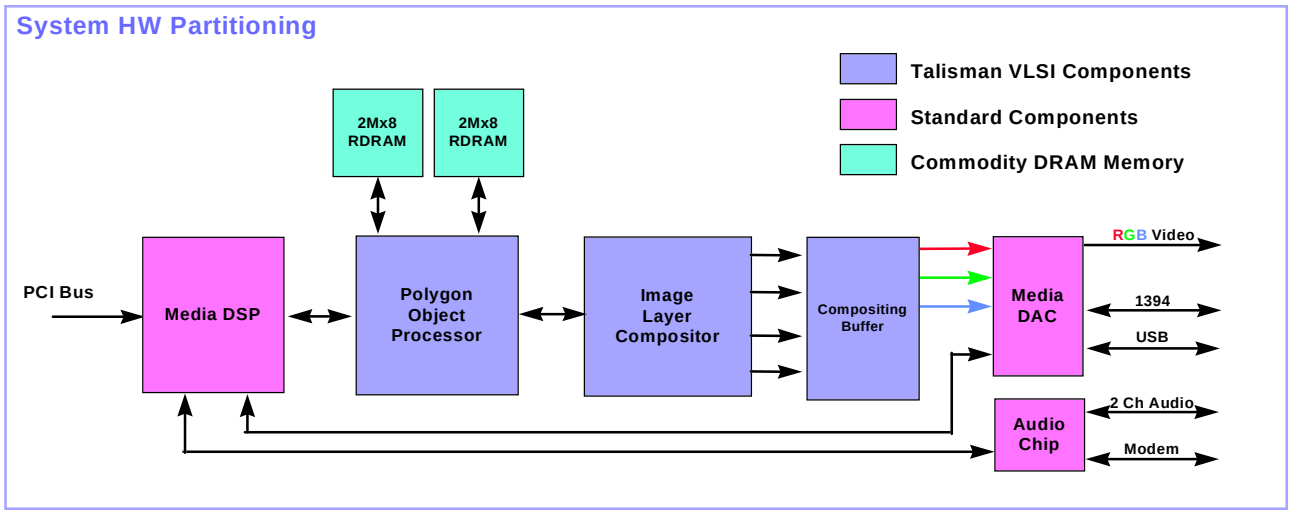
One of the major attractions of the Talisman architecture is the opportunity for 3D interactive applications to break out of the late 1970's look of CAD graphics systems: boring lambertian Gouraud-shaded polygons with Phong highlights. Texture mapping of color improves this look but imposes another characteristic appearance on applications. In the 1980's, the idea of programmable shaders and procedural texture maps[Coo84, Han90] opened a new versatility to the rendering process. These ideas swept the off-line rendering world to create the high-quality images that we see today in film special effects.

By reducing the bandwidth requirements using the techniques outlined above, Talisman can use a single shared memory system for all memory requirements including compressed texture storage and compressed image layer storage. This architecture allows data created by the rendering process to be fed back through the texture processor to be used as data in the rendering of a new image layer. This feedback allows rendering algorithms which require multiple passes to be implemented.

By coupling multi-pass rendering with a variety of compositing modes, texture mapping techniques [Seg92], and a flexible shading language, Talisman provides a variety of rendering effects that have previously been the domain of off-line software renderers. This includes support of functions such as shadows (including shadows from multiple light sources), environment mapped reflective objects, spot lights, fog, ground fog, lens flare, underwater simulation, waves, clouds, etc.

REFERENCE HARDWARE IMPLEMENTATION

The Talisman architecture supports a broad range of implementations which provide different performance, features, rendering quality, etc. The *reference* implementation is targeted at the high-end of the consumer PC market and is designed to plug into personal computers using the PCI expansion bus. This board replaces functionality that is typically provided by a Windows accelerator board, a 3D accelerator board, an MPEG playback board, a video conferencing board, a sound board, and a modem.



The reference hardware consists of a combination of proprietary VLSI devices and commercially available components. The VLSI components have been developed using a top-down modular design approach allowing various aspects of the reference implementation to be readily used to create derivative designs.

The reference implementation uses 4 Mbytes of shared memory implemented using two 8-bit Rambus channels. The Rambus memory provides higher bandwidth than traditional DRAM at near commodity DRAM pricing. This shared memory is used to store image layers and texture data in compressed form, DSP code and data, and various buffers used to transfer data between processing subsystems. A 2MB configuration is also possible, although such a configuration would have lower display resolution and would have other resource limitations.

The Media DSP Processor is responsible for video codecs, audio processing, and front-end graphics processing (transformations, lighting, etc.). The reference HW implementation uses the

Samsung MSP to perform these functions. The DSP combines a RISC processor with a specialized SIMD processor capable of providing high performance floating point and integer processing (>1000 MFLOPS/MOPS). A real-time kernel and resource manager deals with allocating the DSP to the various graphics and multimedia tasks which are performed by this system.

The Polygon Object Processor is a proprietary VLSI chip which performs scan-conversion, shading, texturing, hidden-surface removal, and anti-aliasing. The resulting rendered image layer chunks are stored in compressed form in the shared memory.

The Image Layer Compositor operates at video rates to access the image layer chunk information from the shared memory, decompress the chunks, and process the images to perform general affine transformations (which include scaling, translation with subpixel accuracy, rotation, and skew). The resulting pixels (with alpha) are sent to Compositing Buffer.

Memory Use - Typical Scenario				**Net Memory Requirements**
Image Layer Data Storage				
Display Resolution	*1024*	*x*	*768*	
Average Image Layer Size	*128*	*x*	*128*	
Average Image Layer Depth Complexity	*1.7*			
Image Layer Data Compression Factor	*5*			
Image Layer Memory Management Overhead	*51 bytes*	*per*	*32x32 chunk*	
Memory Allocation Overhead	*4 bytes*	*per*	*128 bytes*	
Total Image Layer Data Storage Requirements				1,171,637 bytes
Display Memory Management	64 bytes	per	image layer	5,222 bytes
Texture Data Storage				
Number of Texels	*4,000,000*		*texels*	
Percent Texels with Alpha	*30%*			
Avg. Number of Texture LODs	*6*			
Texture Data Compression Factor	*15*			
Total Texture Data Storage Requirements				1,415,149 bytes
Command Buffers				53,248 bytes
Audio Output Buffer				2,450 bytes
Audio Synthesis Data				32,768 bytes
Wav Table Buffer				524,800 bytes
Media DSP Program and Scratch Mem				524,288 bytes
Total				**3,729,563 bytes**

Image layer chunk data is processed 32 scan lines at a time for display. The Compositing Buffer contains two 32 scan line buffers which are toggled between display and compositing activities. Each chip also contains a 32 scan line alpha buffer which is used to accumulate alpha for each pixel. The Video DAC includes a USB serial channel (for joysticks, etc.), and an IEEE1394 media channel (up to 400 Mbits/sec. for connection to an optional break-out box and external A/V equipment), as well as standard palette DAC features.

A separate chip is used to handle audio digital to analog and analog to digital conversion.

The table above indicates the total memory usage for a typical 3D application scenario. For the same scenario, the memory bandwidth requirements are shown in the following table.

Memory Bandwidth - Typical Scenario	
Pixel Rendering (avg. depth complexity 2.5)	32.4 Mbytes/sec
Display Bandwidth	130.0 Mbytes/sec
Texture Reads	
Texels per Pixel (anisotropic filtering)	*16*
Texture Cache Multiplier (avg. texel reuse)	*2.5*
Texture Read Bandwidth	58 Mbytes/sec
Polygon Command (30,000 polygons/scene)	61.0 Mbytes/sec
Total 3D Pipeline Bandwidth	281.4 Mbytes/sec

POLYGON OBJECT PROCESSOR

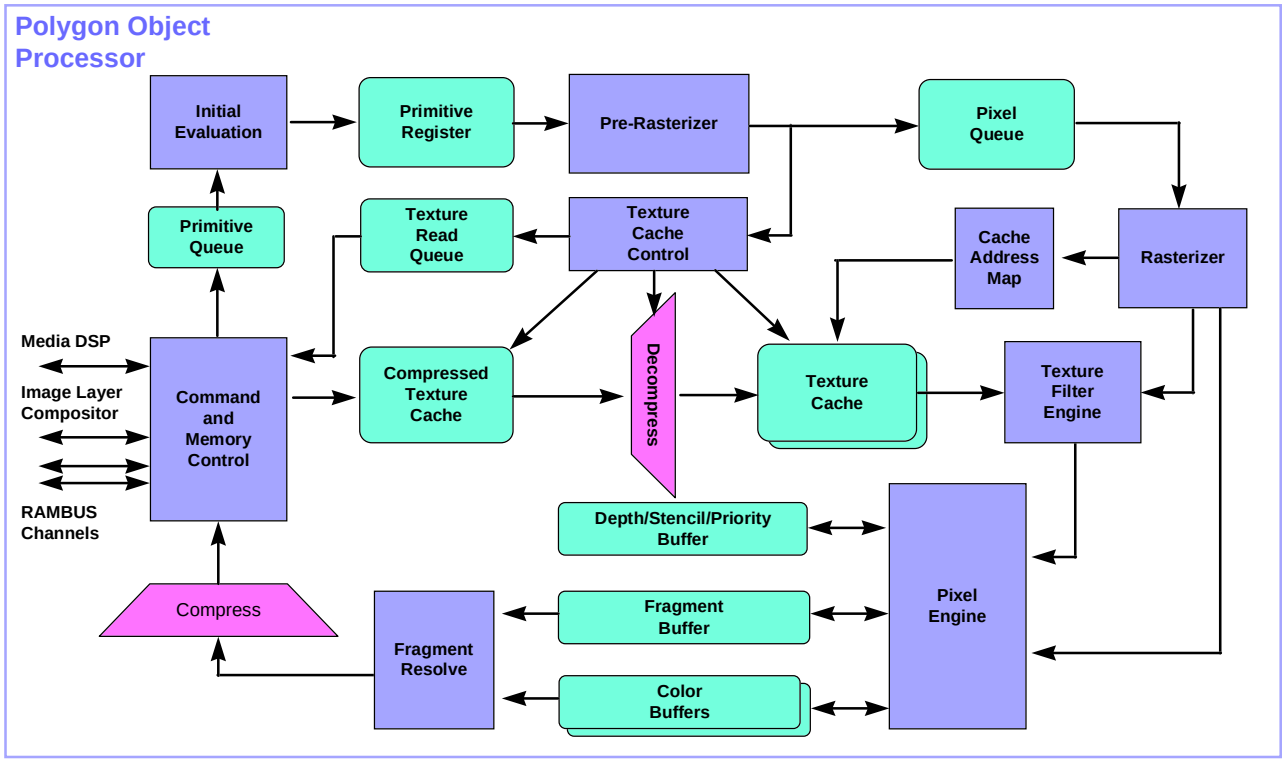
The Polygon Object Processor is one of the two primary VLSI chips that are being developed for the reference HW implementation.

Unique Functional Blocks

Many of the functional blocks in the Polygon Object Processor will be recognized as being common in traditional 3D graphics pipelines. Some of the unique blocks are described here. The operation of this chip is provided later in the paper.

Initial Evaluation - Since polygons are processed in 32 x 32 chunks, triangle processing will typically not start at a triangle vertex. This block computes the intersection of the chunk with the triangle and computes the values for color, transparency, depth, and texture coordinates for the starting point of the triangle within the chunk.

Pixel Engine - performs pixel level calculations including compositing, depth buffering, and fragment generation for pixels which are only partially covered. The pixel engine also handles z-comparison operations required for shadows.



Fragment Resolve - performs the final anti-aliasing step by resolving depth sorted pixel fragments with partial coverage or transparency.

Coping with Latency

One of the most challenging aspects of this design was coping with the long latency to memory for fetching texture data. Not only do we need to cope with a decompression step which takes well over 100 12.5 ns. cycles, but we are also using Rambus memory devices which need to be accessed using large blocks to achieve adequate bandwidth. This results in a total latency of several hundred cycles.

Maintaining the full pixel rendering rate was a high priority in the design, so a mechanism that could ensure that texels were available for the texture filter engine when needed was required. The basic solution to this problem is to have two rasterizers - one calculating texel addresses and making sure that they are available in time, and the other performing color, depth, and pixel address interpolation for rendering. While these rasterizers both calculate information for the same pixels, they are separated by up to several hundred cycles.

Two solutions were considered for this mechanism - one was to duplicate the address calculations in both rasterizers; the other was to pass the texture addresses from the first rasterizer (called the Pre-Rasterizer in the block diagram) to the second rasterizer using a FIFO.

In this case, texture address calculation logic in the rasterizers is fairly complex to deal with perspective divides and anisotropic texture filtering (discussed later). To duplicate this logic in both rasterizers required more silicon area than using the pixel queue, so the latter approach was chosen.

Die Area and Packaging

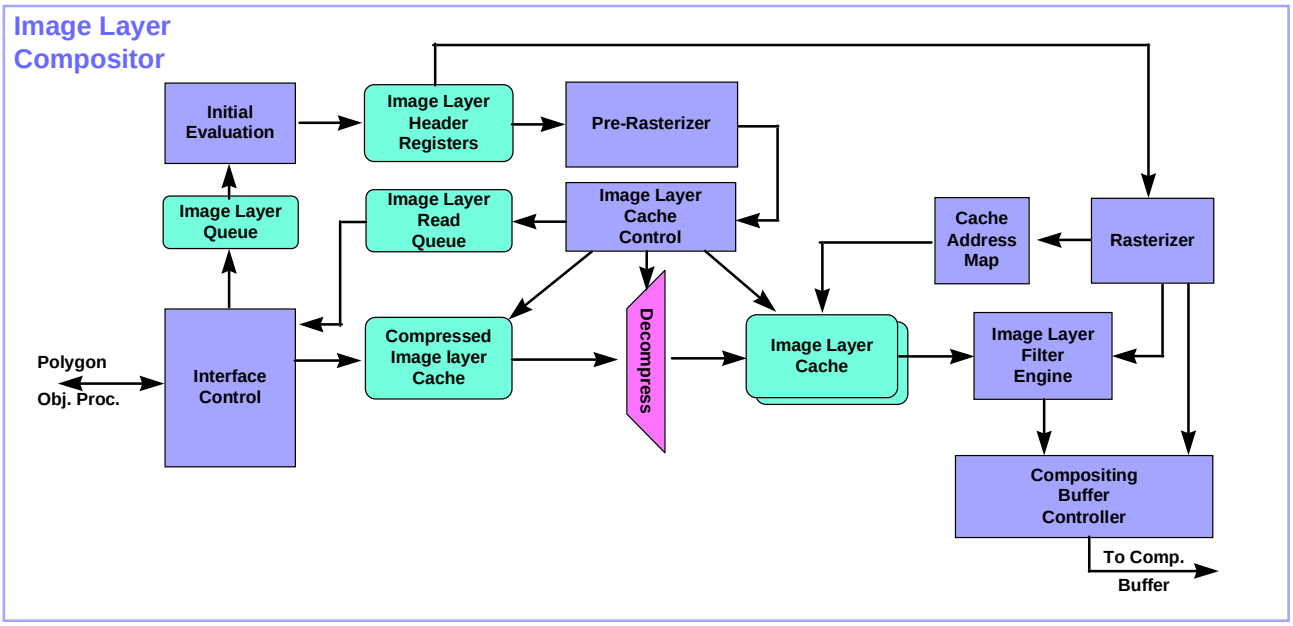
The total die area of the Polygon Object Processor is shown in the following table. The die area figures shown here are estimates since the layout of this part was not complete at the time of paper submission.

POP Area Calculation			0.35 Micron
Functional Block	**Gates**	**RAM bits**	**Total Area**
RAC Cell			5.17
Memory Interface	4,500	12,288	1.77
Input Logic	10,044	0	1.09
Setup Logic	30,920	0	3.92
Scan Convert	125,510	57,760	18.38
Texture Lookup	83,450	0	8.87
Pixel Logic	86,090	137,216	20.03
Cache Logic	42,000	71,680	10.91
Compression Logic	33,120	32,896	14.62
Decompression Logic	47,000	16,000	6.02
			90.77
Testability Gates	50,000		6.55
Interblock Routing Area			9.73
Core Area			107.05
I/O Cells Area			21.69
Total Area			**128.75**

The Polygon Object Processor is implemented using an advanced 0.35 micron four layer metal 3.3 volt CMOS process. The die is mounted in a 304 pin thermally-enhanced plastic package.

IMAGE LAYER COMPOSITOR

The Image Layer Compositor is the other custom VLSI chip that is being developed for the reference HW implementation. This part is responsible for generating the graphics output from a collection of depth sorted image layers.



Comparison with Polygon Object Processor

You will notice that this block diagram is similar in many ways to the Polygon Object Processo. In fact, many of the blocks are identical to reduce design time. In many ways, the Image Layer Compositor performs the same operations as triangle rasterization with texture mapping.

In addition to the obvious differences (no depth buffering, anti-aliasing, image compression, etc.) there are a couple of key differences which significantly affect the design:

Rendering Rate - the Image Layer Compositor must composite the images of multiple objects at full video rates with multiple objects overlapping each other. To support this, the rendering rate of the Image Layer Compositor is eight times higher than the Polygon Object Processor.

Texture/Image Processing - the sophistication of the image processing used by the Image Layer Compositor is significantly reduced in order to keep silicon area to a reasonable level. Instead of performing perspective correct anisotropic filtering, this chip performs simple bi-linear filtering and requires only linear address calculations (since perspective transforms are not supported).

These differences significantly affect the approach used to deal with memory latency. The rasterizer in the Image Layer Compositor is significantly simpler due to the simplified image processing, and the higher pixel rate requires the pre-rasterizer to be much further ahead of the rasterizer. As a result, the Image Layer Compositor eliminates the Pixel Queue and simply recalculates all the parameters in the second rasterizer.

Die Area and Packaging

The total die area of the Image Layer Compositor is shown in the following table. The die area figures shown here are estimates since the layout of this part was not complete at the time of paper submission.

ILC Area Estimate			0.35 Micron
Functional Block	Gates	RAM bits	
Interface Controller	2,290	2,048	1.40
Initial Evaluation	10,514	0	1.35
Header Registers	7,550	2,048	1.25
Pre-Rasterizer	21,018	0	2.65
Rasterizer	19,019	0	2.35
Cache Logic	41,350	71,636	18.30
Decompressor	86,000	25,856	21.00
Filter Engine	21,608	0	4.10
Compositing Buffer Controller	1,200	0	0.25
Testability Gates	50,000		6.60
Interblock Routing Area			4.65
Core Area			63.90
I/O Cells Area			16.99
Total Area			**80.89**

The Image Layer Compositor is implemented using an advanced 0.35 micron four layer metal 3.3 volt CMOS process. The die is mounted in a 304 pin thermally-enhanced plastic package.

OPERATION

This section describes the overall operation of the system and discusses some of the key features.

Objects and Image Layers

As in a traditional 3D graphics system, objects are placed in the virtual environment by the application specifying their position, orientation, and scale relative to the coordinate system of the virtual environment. The transform engine uses this information, in conjunction with the viewpoint specification to construct the synthetic scene.

In this system, however, independent objects are rendered into separate image layers which are composited together at video rates to create the displayed image. Independent objects can be described directly by the APIs (in the case of the Windows PC, this is done using DirectDraw and Direct3D [Mic95]), or can be calculated automatically based on object hierarchy and bounding boxes. The latter approach will likely be used for virtual reality environments described using behavior languages where the relative motion of objects can be predicted.

In our implementation, the host processor maintains control of real-time display operation. The object database is maintained in host memory, and primitive descriptions are passed to the graphics system as needed for rendering. Unlike a traditional 3D graphics system, the entire display is not updated at the same time. Each image layer can be updated based on scene priorities.

As previously discussed, an affine image transformation can be applied to each image layer at video rates as it is composited. This affine transformation is used to simulate a 3D transformation of the object. The appropriate affine transform and the geometric and photometric error that results is computed based on a least square error of selected points within the object.

The host software maintains a priority list of image layers to be updated based on perceptible error and object priorities. More error is allowed for objects that are not considered primary, allowing higher quality to be maintained for the important elements in the scene. For those objects that are not updated, the system will always try to produce the best possible result using affine transforms on previously rendered image layers.

Image layer transforms can be processed considerably faster than re-rendering the geometry - typically 10 to 20 times faster. Since the frame to frame changes of animated objects are typically small, an affine transformed image of a 3D object can be used for several frames before incurring enough distortion to require re-rendering. This gives a tremendous performance advantage to the front end of the graphics pipeline; and since the polygon rendering requires significantly more processing resources, the net result is significantly higher price-performance.

Image Layer Chunking

As previously noted, all image layers are processed as 32 x 32 pixel regions known as chunks. Prior to sending graphics rendering primitives to the Talisman hardware, the host processor must sort the geometry for each image layer into these independent regions.

After some experimentation, we determined that it was more efficient to sort into 64 x 64 regions and allow the hardware to process this geometry for each of the four 32 x 32 chunks in the region.

Sorting into these 64 x 64 regions can be accomplished using a variety of algorithms. In the simplest case, a binary sort is performed based on bounding volumes using an algorithm similar to polygon scanline techniques (active lists, etc.). More advanced algorithms utilize incremental algorithms based on the temporal behavior of the environment. We have found that we can achieve full performance using static algorithms on a mid-range personal computer.

Primitive Rendering

The Talisman software provides the capability to render independent triangles, meshed triangles (strips and fans), lines, and points. All of these primitives are converted to triangles for rendering by the Polygon Object Processor. Triangle rendering provides numerous simplifications in the hardware since it is always planar and convex.

All coordinate transformations, clipping, lighting, and initial triangle set-up is handled by the Media DSP using 32 bit IEEE floating point.

During scan conversion, the Polygon Object Processor uses the linear equation parameters generated by the Media DSP to determine if the triangle is visible in the current chunk. The edge equations are also stored in the Primitive Registers until required by the Pre-Rasterizer and Rasterizer.

As previously discussed, rasterization is split into two sections which are separated by several hundred clock cycles. This separation allows the first section (the Pre-Rasterizer) to determine which texture blocks will be required to complete rendering of the triangle. This information is sent to the Texture Cache Controller so that it can fetch the necessary data from the common memory system, decompress it, and move it into the specialized high-speed on-chip memory system used by the texture filtering engine, as described below.

The second section, the Rasterizer, calculates the color, translucency, depth, and coverage information, and passes this to the Pixel Engine where it can be combined with the texture information to determine the output pixel color.

Texture Mapping

Texture data is stored in the common memory system in the TREC compressed image format. 8x8 blocks are grouped together in 32x32 chunks for memory management purposes, although each 8x8 block is individually addressable. The 32x32 chunks are identical in format to the image layer chunks, allowing textures and image layers to be used interchangeably (although textures are generally mip-mapped).

Data is fetched from the common memory in blocks called Memory Allocation Units. Each MAU is 128 bytes, allowing high bandwidth utilization from the Rambus DRAMs to be achieved.

As texture blocks are needed, they are fetched into a compressed texture cache in the Polygon Object Processor. The compressed texture cache holds sixteen MAUs. Holding texture data in compressed form increases the effective size of the compressed texture cache by the compression factor (typically 15 times).

Texture blocks are decompressed as required by the texturing engine and placed in an on-chip decompressed texture cache. A total of sixteen 8x8 texture blocks are cached in RGB$\propto$ format. The texture cache allows each texel to be used for an average of 2.5 pixel calculations.

A texture filter kernel is generated on the fly for each pixel, depending on the texture resolution, offset, orientation, and Z-slope. The texture processor supports anisotropic filtering with up to 2:1 anisotropy at full pixel rendering rates. Higher anisotropy is supported at lower rendering rates. The resulting filter quality is considerably better than the tri-linear filtering that is commonly used in high-end graphics workstations, and will result in sharper looking images. Tri-linear filtering is also available at full pixel rendering rates.

Hidden Surface Removal

The Polygon Object Processor has an on-chip 32 x 32 pixel depth buffer (with 26 bits per pixel) used to store the depth of the closest primitive to the eye point at each pixel location. The 26 bit depth value uses between 20 and 24 bits for depth with the remaining 2 to 6 bits used for priority and/or stencil. Priority is used to eliminate depth buffering artifacts due to coplanar objects. Priority is tagged per surface, and indicates which surface should be considered as closer to the viewpoint, if the depth of the two surfaces at the specific pixel location are within a certain depth tolerance of each other. The depth tolerance is a

fixed value which is based on the overall accuracy of the graphics pipeline with regards to computing pixel depth values.

The Polygon Object Processor also supports translucent triangles, translucent textures, and triangle edge anti-aliasing, all of which fall outside of normal depth buffer operations. To properly compose pixels which are only partially covered, or have an alpha value less than 1.0, the Talisman system has special anti-aliasing hardware, which is described below.

Anti-Aliasing

One of the significant advantages of the chunking architecture used by Talisman is that high quality anti-aliasing can be implemented cost effectively, without the need for large memory systems. The algorithm we have implemented is compatible with depth buffering and translucent surfaces.

The color buffer always stores the pixel value for the front most, fully opaque, fully covered pixel at each pixel location in the 32x32 chunk. Pixels with partial geometric coverage or which are translucent are contained in a fragment buffer (the basic anti-aliasing algorithm is loosely based on the A-Buffer algorithm described by L. Carpenter [Car84]). Each entry in the fragment buffer provides the color, alpha, depth, and geometric coverage information associated with these pixels. Multiple fragment buffer entries can be associated with a single pixel location within the 32x32 chunk. Memory is managed within the fragment buffer using a linked list structure.

To represent the geometric coverage of a pixel by a polygon edge, a coverage mask is used. In the Talisman system, a 4x4 mask is used, which effectively divides each pixel into 16 sub-pixels. The Talisman hardware does not store an individual entry for each sub-pixel, as do many high end graphics systems of this quality, but instead stores a simple coverage mask which tags which virtual sub-pixels the fragment entry would cover. The 4x4 coverage mask is generated using an algorithm similar to the algorithm described by A. Schilling [Sch91] in order to provide improved dynamic results for near horizontal and near vertical edges.

All fragments generated by the scan converter are tested against the 32x32 pixel depth buffer prior to being sent to the fragment buffer. This eliminates storage and subsequent processing of any fragments which are obscured by a nearer fully covered, fully opaque pixel.

To further reduce the number of fragments that are stored and subsequently processed, the Pixel Engine implements a merging operation for complimentary fragments. In many cases, at a single pixel location, sequential fragments are from the mating edges of adjacent polygons which represent a common surface within an object. In these cases, the pixel depth and color are often virtually identical, with the only difference between the two fragments being the coverage masks. The Pixel Engine compares the incoming fragment to the most recently received fragment at that pixel location, and if the depth and color are within predefined limits, combines the fragments by ORing the incoming fragment coverage mask with the stored fragment coverage mask, and stores the result in place of the stored coverage mask. The rest of the incoming fragment data is discarded, eliminating the use of another fragment entry. When this operation occurs, the resulting mask is tested to see if full coverage has been reached. If full coverage is achieved, the pixel is moved out of the fragment buffer and placed in the 32x32 pixel depth buffer, freeing up additional fragment memory.

Once all the polygons for the chunk are rendered, the anti-aliasing engine resolves the remaining pixel coverage values performing front-to-back compositing of each pixel which has one or more fragments associated with it. As a chunk is being resolved, the Rasterizer, Pixel Engine, etc. are rendering the next chunk. As fragments are freed up by the resolve process, they are added to the free fragment list where they can be used by the next chunk being rendered.

Shadows

The Pixel Engine supports a full range of compositing functions which are useful for multipass rendering. One of the most common uses is for the generation of real-time shadows. The Polygon Object Processor implements a shadow buffer algorithm that determines which parts of the object are in shadow from a given light source. This is done in three passes, assuming a single light source.

In the first pass, a *shadow buffer* is generated by rendering the scene from the point of view of the light source [Wil78, Ree87]. The shadow buffer generated from this step is a depth buffer, where each pixel depth location is set to the average depth of the two closest surfaces to the light source. This is done to eliminate the possibility of the front surface casting a shadow on itself due to accuracy errors in the algorithm, and was first described by A. Woo [Woo92]. This shadow buffer is then saved in the common memory system.

In the second pass, the scene is rendered from the eyepoint, generating a normally lit scene from the single light source.

In the third pass, the scene is rendered from the eye point, to generate the shadow contribution to the image. As the scene is rendered, the shadow buffer is accessed using the texture map addressing hardware, and the shadow buffer depth is compared with a projection of the surface depth in the shadow buffer coordinate system. If the surface depth is behind the shadow buffer depth, it is in shadow. For each pixel in the eyepoint scene, a 4x4 or 8x8 set of nearest shadow buffer locations are visited, each producing an *in-front* or *behind* result. A trapezoid weighted filter is multiplied against the results of the shadow depth compares, giving a resulting shadow factor, ranging from 0 to 1.0. The trapezoid filter gives a soft, anti-aliased edge to the shadow in the resulting scene. The shadow factor is used to attenuate the pixel intensity from the scene rendered in pass two.

Display Image Generation

A data structure defining the image layer display list (sorted in depth order) is stored in shared memory and is traversed every frame time to determine how the image layers are displayed. This data structure can be modified by the host at any time to control the temporal behavior of each image layer (and hence each independently rendered object).

Image layer compositing is performed 32 scanlines at a time. The Image Layer Compositor traverses the image layer data structure (sorted in front to back order), performing image transforms on those chunks that are visible in each 32 scanline band.

The Compositing Buffer is a specialty memory device developed for the Talisman architecture. This part has two 32 scanline color buffers (24 bit RGB) and a 32 scanline alpha buffer. One of the color buffers is used for compositing into while the other

is being used for streaming the video data out to the monitor. The two buffers ping-pong back and forth so that as one scanline region is being displayed, the next is being composited. The single 32 scanline alpha buffer is used only for compositing and is cleared at the start of each new 32 scanline region.

The Image Layer Compositor (which does the image layer chunk addressing, decompressing, and image processing), passes the color data and alpha data to the Compositing Buffer for each pixel to be composited. Since the sprites are processed in front to back order, the alpha buffer can be used to accumulate opacity for each pixel, allowing proper anti-aliasing and transparency.

FUNCTIONALITY AND PERFORMANCE

The features and major performance goals for the reference implementation are:

- Single PCI board implementation with audio, video, 2D and 3D graphics.
- High resolution display capability - 1344 x 1024 @ 75 Hz.
- 24 bit true-color pixel data at all resolutions for maximum image fidelity.
- Optimized for 3D animation at full refresh rates (75 Hz) using a combination of image layer animation and 3D rendering. Scene complexity of 20,000 to 30,000 rendered polygons or higher can be supported. This is comparable to a 3D graphics workstation capable of 1.5 to 2 million polygons per second.
- Polygon Object Processor pixel rendering rate of 40 Mpixels per second with anisotropic texturing and anti-aliasing. Image Layer Compositor pixel compositing rate of 320 Mpixels/sec.
- Very high quality image generation incorporating anisotropic texture filtering, subpixel-filtered anti-aliasing, translucent surfaces, shadows, blur, fog, and custom shading algorithms.
- Front-end geometry processor to off-load transformations, clipping, lighting, etc.
- Full resolution (720 x 486) MPEG-2 decode, as well as other video codecs. Video can be used as surface textures, and can be combined with graphics animations.
- Base system has two-channel 16-bit audio inputs and outputs with DSP based MIDI synthesis (wave table and other mechanisms supported), 3D spatialization, and digital audio mixing. Other audio processing is also supported.

CONCLUSIONS

The Talisman architecture demonstrates how a fresh look at 3D animation hardware can result in dramatic improvements in price-performance. The system described in this paper has a bill of materials of $200 to $300, yet can achieve performance and quality comparable or superior to high-end image generators and 3D graphics workstations.

The first reference implementation is a high-end PC board level system with adequate functionality to meet a broad range of applications. The goal of the Talisman architecture is to significantly improve the quality, performance, and integration of media technologies on the PC. This first implementation is intended to be a realization of this goal at a price point that is viable for consumer applications, and to be a reference from which derivative designs can be created.

Although the reference implementation or other products based on this technology are not yet on the market, we believe that products based on the architecture described in this paper will have retail street prices of $200 to $500.

The Talisman architecture has been fully simulated and Verilog models are complete. We expect prototype implementations of this hardware by late this year.

ACKNOWLEDGMENTS

The authors would like to thank the entire Talisman research and develop team for their contributions to this program. We would specifically like to thank Jim Blinn, Joe Chauvin, Steve Gabriel, Howard Good, Andrew Glassner, Kent Griffin, Bruce Johnson, Mark Kenworthy, On Lee, Jed Lengyel, Nathan Myhrvold, Larry Ockene, Bill Powell, Rob Scott, John Snyder, Mike Toelle, Jim Veres, and Turner Whitted for their contributions to the HW architecture, algorithms, and demos, although many others also contributed.

REFERENCES

[Ake88] Akeley, K. and T. Jermoluk, "High Performance Polygon Rendering", *Proceedings of SIGGRAPH 1988* (July 1988), p239-246.

[Ake93] Akeley, Kurt, "Reality Engine Graphics", *Proceedings of SIGGRAPH 1993* (July 1993), p109-116.

[Car84] Carpenter, L. "The A-Buffer, an Anti-Aliased Hidden Surface Method", *Proceedings of SIGGRAPH 1984*, (July 1984), p103-108.

[Che94] Shenchang Eric Chen, Lance Williams, View interpolation for image synthesis, *Proceedings of SIGGRAPH 93*, (August 1993), pp. 279-288.

[Che95] Shenchang Eric Chen, QuickTime VR—an image based approach to virtual environment navigation, *Proceedings of SIGGRAPH 95*, (August 1995), pp. 29-38.

[Coo84] Cook, R, "Shade Trees", *Proceedings of SIGGRAPH 1984*, July 1984, p223-231.

[Coo87] Cook, R., L. Carpenter, E. Catmull, "The REYES Image Rendering Architecture, *Proceedings of SIGGRAPH 1987* (July 1987). p95-102

[Fuc89] Fuchs, H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, L. Isreal, "Pixel Planes 5: A Heterogenous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Proceedings of SIGGRAPH 89*, p79-88.

[Han90] Hanrahan, P. and J. Lawson, "A Language for Shading and Lighting Calculations", *Proceedings of SIGGRAPH 1990*, August 1990, p289-298.

[Lev84] Levinthal, A., T. Porter, "Chap - a SIMD Graphics Processor", *Proceedings of SIGGRAPH 84*, p77-82.

[Mic95] Microsoft, "DirectDraw API Specification" and "Direct3D API Specification", Microsoft Corporation, Redmond WA, 1995.

[McM95] Leonard McMillan, Gary Bishop, Plenoptic modeling: an image-based rendering system, *Proceedings of SIGGRAPH 95*, (August 1995), pp. 39-46.

[Mol91] Molnar, S., "Image Composition Architectures for Real-Time Image Generation", *PhD Dissertation*, University of North Carolina, 1991.

[Mol92] Molnar, S., J. Eyles, J. Poulton, "PixelFlow: High Speed Rendering Using Image Composition", *Proceedings of SIGGRAPH 1992* (July 92), p231-240

[Nis83] H. Nishimura, H. Ohno, T. Kawata, LINKS-1: a parallel pieplined multimicrocomputer system for image creation, Proceedings of the 10th Symposium on computer architecture (1983), pp.387-394

[Nvi95] Nvidia, various press releases on the Nvidia NV1 Multimedia Accelerator, Nvidia Corporation, Sunnyvale CA, 1995.

[Pot89] Michael Potmesil and Eric Hoffert, The PixelMachine: a parallel image computer, Proceedings of SIGGRAPH 89, (July 1989), pp. 69-78.

[Reg94] Regan, M. and R. Pose, "Priority Rendering with a Virtual Reality Address Recalculation Pipeline", *Proceedings of SIGGRAPH 1994* (July 94), p. 155-162.

[Ree87] Reeves, W. , D. Salesin, R. Cooke, "Rendering Anti-aliased Shadows with Depth Maps", *Proceedings of SIGGRAPH 87*, p283-291.

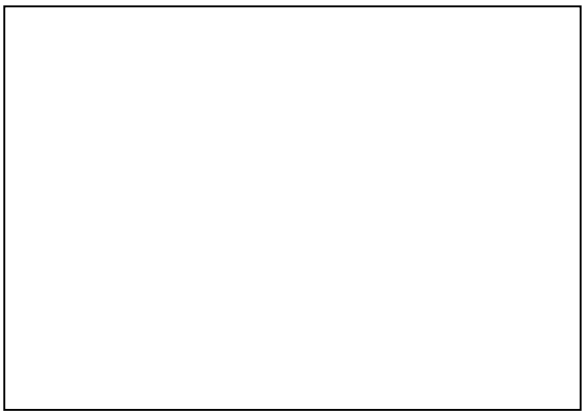
[Sch91] Schilling, A. "A New Simple and Efficient Anti-aliasing with Subpixel Masks", *Proceedings of SIGGRAPH 1991* (July 1991), p133-141.

[Seg92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, Paul Haeberli, Fast shadows and lighting effects using texture mapping, *Proceedings of SIGGRAPH 92*, (July 1992), pp. 249-252.

[Wil78] Williams, L., "Casting Curved Shadows on Curved Surfaces", *Proceedings of SIGGRAPH 78*, p270-274.

[Woo92] Woo, A. in "The Shadow Depth Map Revisited", in *Graphics Gems*, edited by D. Kirk, Academic Press, Boston, 1992, p338-442.

SAMPLE



This sample image, and the one shown on the frontispiece of the 1996 SIGGRAPH Proceedings were generated using a bit and cycle accurate simulator of the Talisman reference hardware. Both of these images are single frames from an animation that will be rendered in realtime on the Talisman hardware.