Improving Rendering by Texture Map-Based Triangle Strips

Yu Yang, Tulika Mitra and Zhiyong Huang

Department of Computer Science, School of Computing
National University of Singapore, Singapore 117543
{yangyu, tulika, huangzy}@comp.nus.edu.sg

Abstract

Improving the rendering performance is a basic problem for computer graphics system. In this paper, we are aiming to investigate the impact on the rendering performance of some geometry compression methods. These compression methods are devised to optimize the use of the vertex cache. We will study how it interacts with the on-chip texture cache. Based on the study, one simple method of improving rendering by texture map-based triangle strips is proposed to balance the use of the vertex and texture caches. We have conducted the experiments to show the effectiveness of this method.

1. Introduction

The development of computer graphics applications is quickly increasing and it is common to find systems which will result in complex models of millions polygons with texture mapping. In many cases, the full detailed geometry and texture maps must be sent down to graphics hardware for rendering. As the users demand ever larger and more realistic 3D models, the transmission time, rendering time and storage requirements grow rapidly. Thus, real-time graphics hardware is increasingly facing a memory bus bandwidth bottleneck in which a large amount of data cannot be sent fast enough to the graphics pipeline for rendering [1].

Limited memory bandwidth is a barrier to increasing PC graphics performance. The memory interface gets inundated with multiple, continuous, high bandwidth demands such as pixel writes, pixel reads, display refresh, AGP bus transactions, and texture reads. Unfortunately, end users notice a slowdown in graphics performance when one of their multiple demands gets bottlenecked by the memory interface.

In this paper, we are aiming to investigate the impact on the rendering performance of some geometry compression methods that are devised to optimize the use of the vertex cache. We will study how they interact with the on-chip texture cache. Based on the study, one simple method of improving rendering by texture map-based triangle strips is proposed to balance the use of the vertex and texture caches. We have conducted the experiments to show the effectiveness of this method.

2. Background and Related Work

To reduce the bottleneck effect which appears as an obstacle to the increasing needs for fast rendering, the rendering process must be carefully examined. The traditional OpenGL polygon-rendering pipeline consists of geometry processing, rasterization and image composite [2]. In the geometry processing stage, input data will go through transformation, shading, primitive assembly, visibly culling and projection. In this stage, the information needs to be processed is the geometry data, which includes the vertex position, face information, and normal vectors. In the rasterization stage, the raster images (e.g. texture, bump and environment maps) will be transmitted from system memory to graphics processor, with the use of a texture image cache to reduce the texture image bandwidth. In the image composite stage, it will process the z-buffer of each pixel and then write them to memory.

2.1. Geometry Compression

One solution for reducing the bandwidth is to compress the static geometry as an offline pre-process [3]. This strategy exploits the idea of using vertex cache. By using a relative small size of cache to hold frequently referenced vertices, savings are generated because to render the same object, fewer vertices are needed to pass through to the graphics subsystem.

Some basic and popular geometry compression methods are taken for experiment. One is generalized triangle strip; the other is generalized triangle mesh. Triangle strip is a very special way of organizing triangles.

Considering the following triangulation shown in Figure 1(a), to maximize the use of the available data bandwidth, it is desirable to order the triangles so that consecutive triangles share an edge. Using such an ordering, only the incremental change of one vertex per triangle need to be specified, potentially reducing the rendering time by a factor of three by avoiding redundant lighting and transformation computations. Besides, such an approach also has obvious benefits in compression for storing and transmitting models. To allow greater freedom in the creation of triangle strips, a "swap" command permits one to alter the FIFO queuing

discipline in a triangle strip as shown in Figure 1(b), the triangle strip can extend further by taking the sequence of (1 2 3 SWAP 4 5 6). The swap command gives greater freedom in the creation of triangle strips at the cost of one bit per vertex. This form of a triangle strip that includes swap command is referred to as a generalized triangle strip [4].

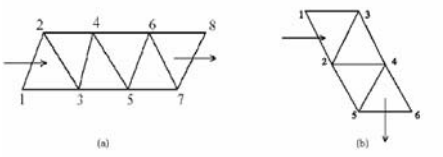


Figure 1. Triangle strips

The concept of a generalized triangle strip structure allows for compact representation of geometry while maintaining a linear data structure. By confining itself to the linear strips, the generalized triangle strip leaves a potential factor of two in the space occupied.

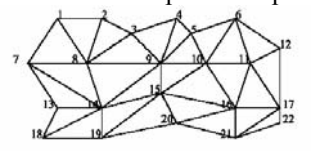


Figure 2. Generalized triangle mesh

While the geometry in Figure 2 can be represented by one triangle strip, many of the interior vertices appear twice in the strip. This is inherent in any approach wishing to avoid references to the old data. A generalized technique is employed to address this problem. The old vertices are explicitly pushed into a queue, and then implicitly referenced from the queue in the future when the old vertex is desired again. This queue is referred to as the mesh buffer. The combination of generalized triangle strips and mesh buffer references is referred to as a generalized triangle mesh [3].

2.2. Texture Mapping

Texture mapping can substantially enhance the realism and visual complexity of computer generated imagery [5]. Two characteristics of texture mapping are: (1) texture images often require large amounts of memory, and (2) it requires many calculations and texture lookups. These characteristics cause it to be the main performance bottleneck in graphics pipeline. For each screen pixel that is textured, the calculations consist of generating texture addresses, filtering multiple texture samples to avoid aliasing artifacts, and modulating the texture color with pixel color. Since the number of pixels that are texture mapped can be quite large (typically tens to hundreds of millions per second), and each one requires multiple texture lookups (usually 8), the memory bandwidth requirements to texture memory can be very large (typically several gigabytes per second). In addition, to achieve the high clock rates required in graphics pipeline, low latency access to texture memory is needed.

For example, the approximate bandwidth requirement for a professional application running full-screen at a resolution of 1,280×1,024, and drawing a complex trilinear-textured scene filling the graphics window is 1,280 × 1,024 × (16 bytes + 32 bytes) × 60 fps × 3 = 11.32 GB/sec. Assuming that 3 out of every 4 texel fetches can be satisfied from the texture cache, the bytes transferred from memory to GPU arising from texture fetches would be reduced by about 75%. This may appear somewhat aggressive. However, considering that the neighboring pixels can easily share a significant number of the same texels and a texture surface also typically covers a reasonable screen area in terms of pixels, if the texture cache is large enough, texel reuse will be significantly increased. Its impact on the bandwidth requirement is significant. Using the above illustration, the bytes transferred from memory become: 1,280 × 1,024 × (16 bytes + 8 bytes) × 60 fps × 3 = 5.66 GB/sec. The bandwidth requirement is reduced to nearly a third and goes from being beyond the limit of traditional memory controller architectures to something actually achievable.

One proposed approach is to use an SRAM cache with each fragment (screen pixel) generator [6]. The premise is that there is a substantial amount of locality of reference in texture mapped scene. By using the small size of SRAM texture cache, lower latency of access to texture memory and higher bandwidth can be achieved. There are three factors important to texture cache behavior (1) the representation of texture images in memory, (2) the cache organization, and (3) the rasterization order on screen.

2.3. The Problem of Geometry Compression

Different triangles traverse orders will definitely affect the access patterns of the texture images. In Figure 3 for example, because each triangle in geometry will be mapped to a certain part of the texture images, the traversal order of T1-T2-T3-T4 will generate different access sequence to the texture images compared to that of T1-T2-T4-T3. The texture cache is used to store a small amount of texture image data for further references. If the triangle traverse orders are rather random, it may generate a large amount of cache misses and thereby worsen the rendering performance.

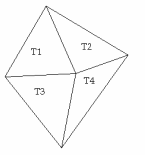


Figure 3. Traverse orders for triangles

The triangle traversal orders are determined by the geometry compression scheme because these compression schemes exploit a special way of arranging triangles to be sent for rendering. This order is important to different caches used on the graphics chip, so the compression scheme should be aware of the utilization of those caches.

The geometry compression schemes usually ignore the importance of the texture cache; they focus only on the vertex cache. For example, if a triangle strip happens to be mapped to different texture images or many distant parts of one image, the triangle rendering order will not give a good rendering performance.

There are four types of locality in texture mapping: (1) *Intra-triangle locality*. Pixels within a triangle naturally share blocks of texture. (2) *Intra-object locality*. Graphics objects generally comprise multiple triangles. Triangles within an object naturally share blocks of texture. (3) *Intra-frame locality*. Objects within a frame may share textures, especially as hardware becomes more common that supports multiple textures applied to the same object. (4) *Inter-frame locality*. Generally the viewpoint moves only incrementally between frames. Texture blocks employed in one frame are likely used again in the next one. The texture cache implemented in software environment is designed primarily for the intra-triangle working set, but can be expected to absorb some of the intra-object working set as well.

Different triangle traverse orders result in different cache hit ratio. Normal sequence of rendering just follows the face sequence of an object. This sequence is fairly random. It has not taken any of cache consideration into account. Thus it has a relatively lower hit rate. While the triangle strip or generalized triangle mesh explore the triangles in a way that the triangles within a strip are adjacent, the access to their corresponding textures will most probably hit the textures left in the cache, which have just been accessed.

3. Texture Map-based Triangle Strips and Implementation

Based on the above analysis, we present a simple solution by introducing the texture maps into the meshes: the texture map-based triangle strips (Figure 4).

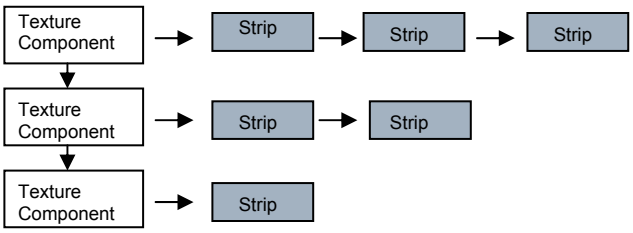


Figure 4. A texture map-based triangle strip

The idea is simple. It introduces one level above any existing geometry compression scheme. Now, the triangles are not processed in one list. First, they will be organized in groups by the texture maps applied to them. In each group, the triangles that are texture mapped by only one image are ordered by a data representation of the geometry compression scheme, e.g., the triangle strips. This way, we can balance the use of the texture caching and vertex caching.

We have implemented the method using C++ and OpenGL and running the experiments on a Pentium IV PC machine with 256MB memory and 1.6GB harddisk.

First, we implemented a 3-D polygonal graphics pipeline. This is responsible for geometry, clipping, and lighting of vertices, rasterization, shading, texture mapping and Z-buffering. The pipeline is similar to the one described in [6]. Specifically, the texture mapping implementation is based on mipmap method [12] using OpenGL [8]. Since the pipeline is implemented in software, it is easy to experiment with different rasterization order of triangles. Second, we implemented a function to trace the Open GL calls that are made by a graphics application running in real-time. This was done using the Mesa (http://www.mesa3d.org/), an open source 3-D graphics library with an API which is similar to that of OpenGL. It is easy to explore into all graphics application function calls and discover the process of how rendering is taking place. Third, we implemented a trace-driven cache simulator that can model different cache sizes and queuing disciplines. Whenever the software based fragment generator accesses texels from the memory, all the accessing records will be kept and later passed to the cache simulator. The cache simulator runs after the graphics pipeline.

4. Experiment Study

First, the geometry compression is carried out on the 3D objects as an offline pre-process. Next, the visiting order of triangles is extracted from the compressed geometry data for future analysis of the traversal order effects. Then the compressed data are sent to the 3-D graphics pipeline of the simulation environment discussed above. When the object is rendered on the screen, each generation of screen pixel needs to lookup in the texture space for RGBA values and interpolates them. The modified graphics library will generate the texel-pixel mapping and produce a list of texture address corresponding to each screen pixel. Finally this mapping will be passed to the third component which is the cache simulator. Since the cache simulator is software based, the cache hit ratio can be easily examined under different cache sizes and queuing disciplines.

4.1. Texel-Pixel Mapping

By observing the texel-pixel mapping, we can extract the texture access pattern of the rendering process. We want to study how different visiting orders of triangles will affect the texture and pixel access.

Looking into the source code of Mesa, the rasterization procedure is conducted like this: first, do a scan conversion over the screen space, for each pixel that is visible, a function call will calculate its RGBA value. This value comes from the interpolation of several texels' RGBA values. The code is modified so that each time a screen pixel is scanned and displayed, a pair of texel-pixel mapping is written to an external file. Here is one example of this kind of pair:

Texel s = 0.534613 t = 0.213296,
Texel s = 0.534663 t = 0.218398,
Pixel x = 326 y = 215,

which means that to render the pixel located at the coordinate of (326,215), texel (s,t) = (0.534613,0.213296) and (0.534633,0.218398) will be referenced for their color values. The texel coordinates here do not mean there is a exact map in the texture place. It still needs texel interpolation. Thus for each coordinate there may still have several more texel lookups. Assuming that there is a texture map, 800 pixels wide (east to west) by 600 pixels tall (north to south), to look up a value in the map, the longitude is scaled to the range 0 to 1 in step of 800 and the latitude to the range 0 to 600. the integer parts of these numbers, call them I_u and I_v, give the coordinates of the upper left of the 2 × 2 pixel region that must be fetched for interpolation. The interpolation amount comes from the fractional parts of the scaled u and v values.

4.2. Geometry Compression

The geometry compression methods are generalized triangle strip and generalized triangle mesh. For generalized triangle strip, the optimized algorithm by Evans et al. is adapted [4]. The compression result is very impressive. The following table shows the number of vertex count for each swap and vertex sent to the renderer.

Data File	Plane	Skyscraper	Foot	Triceratops	Porsche
Vert. No.	1,508	2,022	2,154	2,832	5,247
Tris. No.	2,992	3,692	4,202	5,660	10,425
Cost	3,509	4,616	5,102	6,911	12,367

Table 1. The cost of triangle strip algorithms

The algorithm ideally will result in a cost of one vertex per triangle. However, in experiment, a cost of around 1.1 to 1.25 vertex per triangle is achieved. For the generalized triangle mesh, the implementation from Chow is adapted [9]. By increasing the buffer size from a capacity of 2 vertices to 16 vertices, the cost will go down to a theoretical minimum of 0.5 vertex per triangle. In practice the result turns out to be an average of 0.67 vertex per triangle with an average of 43% vertex reuse. The comparison of the cost per triangle between two algorithms is illustrated in Figure 5.

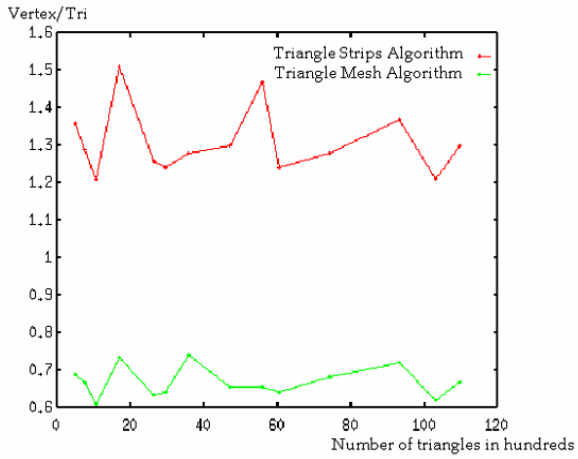


Figure 5. Comparison between two algorithms in term of vertex to triangle ratio

4.3. Texture Cache Size

Working set size is a measure of the amount of data that is actively in use at a particular time. Most applications have a hierarchy of working sets [10]. Suppose the screen resolution is R and the depth complexity is d which represent the average number of pixels that are rendered for each pixel location. Thus, the number of pixel N_{pix} generated during rasterization is N_{pix} = R * d. In this experiment, the MIP level is chosen to be 1:1, so $N_{pix} = N_{tex.}$ The working size W is given by N_{tex} * texel size.

The following figure shows how the cache behaves as the cache size changes.

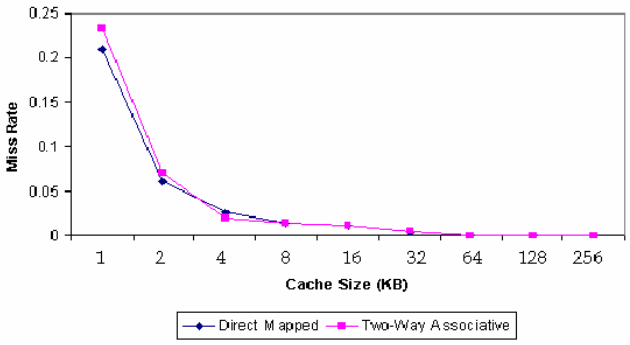


Figure 6. Texture cache miss rate under different cache size

The triangles in a generalized mesh appear close to each other and grow in a patch pattern because the implementation uses a progressive growth method [9].

A larger buffer size implies that more of the textures that are previously transmitted will be re-used. But there is also a cost-efficient issue to be considered. Implementing very large texture cache on graphics chip is much more expensive than memory on board. Taking this into consideration, a texture cache size which results in about 75% hit ratio is adapted in NVIDIA's graphics card design [11].

4.4. Cache Queuing Disciplines

Experiment studies were carried out on two types of queuing disciplines for maintaining the buffer:

(1) First-in, first-out (FIFO) - this implies that there is no rearrangement of the textures in the buffer. FIFO is the easiest to implement in hardware, and would thus be preferable if the performance is comparable.

(2) Least recently used (LRU) – LRU dynamically rearranges the texture in the buffer, by loading a texture that was used most recently into the spot in the buffer that holds the most recently admitted texture. The least recently used texture is eliminated when a new texture is added to the queue. LRU provides the benefit that popular textures are held in the buffer in the hope that they will likely be used in the near future.

The results of running test on two queuing disciplines with different cache size are presented in Figure 7.

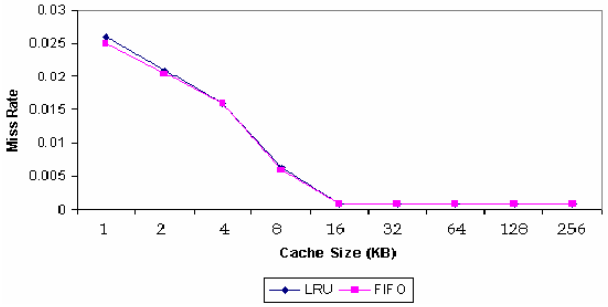


Figure 7. Texture cache miss rate under LRU and FIFO

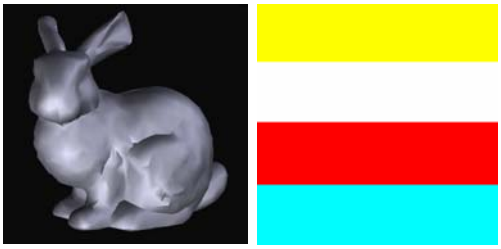
The above figure shows the cost of the LRU and FIFO queuing disciplines versus different cache size. As can be seen the advantages to be gained from larger buffer sizes starts diminishing beyond a buffer size of about 16KB. For buffer sizes less than 16KB, LRU performs better than FIFO scheme by a factor of about 5%.

4.5. Triangle Strip vs. Texture-Based Method

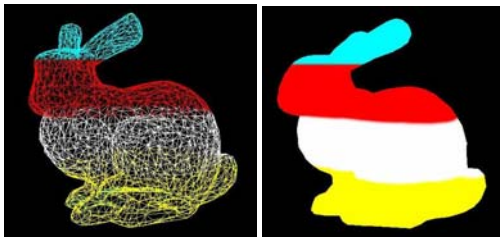
We carried out the follow-up experiment on the texture cache size range from 1KB to 64KB using LRU replacement policy. We compared our proposed algorithm and the triangle strip algorithm. We omit the generalized triangle mesh algorithm because it is just a special type of triangle strip and they will have similar result.

In the comparative study, we looked into two types of objects. One type of objects is constructed with one connected component, like the Stanford bunny. The other type of objects is constructed with some separated components. For example, the foot object we used in the experiment, the skeletons and bones of the foot are naturally separated.

Figure 8 shows an example of texture mapping on the Stanford bunny model. The original model (Figure 8 (a)) is mapped with a texture map of four colors (Figure 8 (b)). The rendering result is shown in wireframe and smooth shading respectively (Figure 8(c)). The performance study result using the texture cache miss rate as the metric is shown in Figure 9. We compared the proposed method, where the triangle strips are organized in four bins respective to the four colors in the texture map, with the strip compression method, where the triangle strips are organized as only one sequence. We can see the cache miss rate of the proposed method, indicated by "texture" in the figure, is lower than the strip compression method, indicated by "strip" in the figure, due to balance the use of the texture and vertex caches of the proposed method.



(a) Original model (b) Texture map



(c) Texture mapping result in wireframe and smooth shading

Figure 8. Texture mapping on Stanford bunny

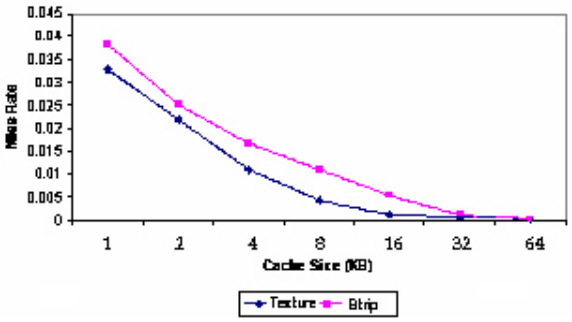


Figure 9. Texture cache miss rate of bunny object

Similarly, we have conducted another experiment on a foot model (Figure 10) and the performance study result using the texture cache miss rate is shown in Figure 11.

Figure 10. The foot skeleton

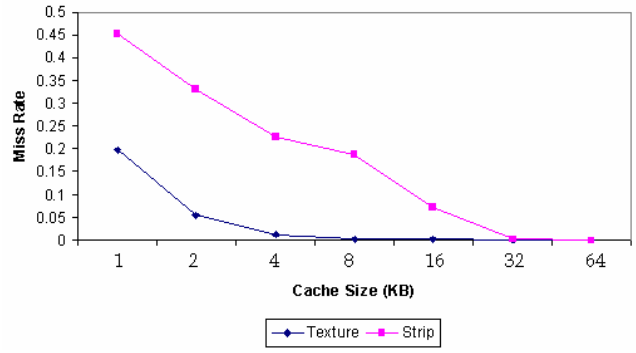


Figure 11. Texture cache miss rate of foot object

From the above results, we can see that for the Stanford bunny like models, the proposed method has about 10%~15% less texture cache miss rate than the triangle strip algorithm. For the foot like models, it has about 100%~300% less. To gain this better performance in texture caching performance, we need to pay a cost of a lower vertex caching performance. For the first type of models, the proposed method has about 1% more in the vertex caching cost than the triangle strip algorithm. For the second type of models, it has about 10% more. Since the texture information contributes a higher weight to the traffic transferred on the memory interface, overall the proposed method is still better than the triangle strip algorithm.

5. Conclusion and Future Work

We have studied the rendering performance of two geometry compression methods in texture mapping. The analysis and experiment results showed that the compression methods alone are not optimal. A simple method of improving rendering by texture map-based triangle strips was proposed and implemented.

Texture cache is very useful in reducing the memory bandwidth and improves rendering performance. Besides the rasterization order, there are two more factors that will affect the texture cache behavior for us to explore in future:

(1) The cache organization. Study the effect by adapting a single texture cache and two level caching strategies.

(2) The representation of texture in storage place. Study what kind of addressing scheme will be most effective for fetching of textures.

6. References

[1] Sun Microsystems, Inc. The UPA Bus Interconnect. "Ultra1 – Creator3D Architectural Technical Whitepaper", www.sun.com/desktop/whitepaper/Ultra1, 1996.

[2] J. Foley, A. van Dam, S. Feiner, J. Hughes. "Computer Graphics Principles and Practice", Second Edition, Addison-Wesley Publishing Company, Inc, 1990.

[3] M. Deering. "Geometry Compression", *SIGGRAPH'95*, August 1995, pages 13-20.

[4] F. Evans, S. Skiena, A. Vashney. "An Optimizing Triangle Stripes for Fast Rendering", *Proc. Visualization '96*, 1996, pages 319-326.

[5] P. S. Heckbert. "Fundamentals of Texture Mapping and Image Warping", University of California at Berkeley, June 1989, pages 321-326.

[6] K. Akeley. "Reality Engine Graphics", *SIGGRPH'93*, September 1993, pages 109-116.

[7] J. F. Blinn. "The Truth about Texture Mapping", *IEEE Computer Graphics and Applications,* March 1990, pages 78-83.

[8] M. Segal, and K. Akeley. "The OpenGL Graphics System: A Specification", Version 1.2.1, Silicon Graphics, Inc., April 1999.

[9] M. Chow, "Optimized geometry compression for real-time rendering", *Proc. Visualization'97*, 1997, pages 415-421.

[10] E. Rothberg, J. P. Singh, and A. Gupta. "Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors", *Proceedings of the 20th Annual International Symposium on Computer Architecture,* May 1993, pages 14-26.

[11] NVIDIA® Corporation. "Lightspeed Memory Architecture II Tech Brief", http://www.nvidia.com/_view.asp?PAGE=geforce4, April 2002.

[12] L. Williams. "Pyramidal Parametrics", *Proceedings of SIGGRAPH '83.* July 1983, pages 1-11.