Texture Tile Visibility Determination For Dynamic Texture Loading

Michael E. Goss * and Kei Yuasa †
Hewlett-Packard Laboratories

ABSTRACT

Three-dimensional scenes have become an important form of content deliverable through the Internet. Standard formats such as Virtual Reality Modeling Language (VRML) make it possible to dynamically download complex scenes from a server directly to a web browser. However, limited bandwidth between servers and clients presents an obstacle to the availability of more complex scenes, since geometry and texture maps for a reasonably complex scene may take many minutes to transfer over a typical telephone modem link.

This paper addresses one part of the bandwidth bottleneck, texture transmission. Current display methods transmit an entire texture to the client before it can be used for rendering. We present an alternative method which subdivides each texture into tiles, and dynamically determines on the client which tiles are visible to the user. Texture tiles are requested by the client in an order determined by the number of screen pixels affected by the texture tile, so that texture tiles which affect the greatest number of screen pixels are transmitted first. The client can render images during texture loading using tiles which have already been loaded. The tile visibility calculations take full account of occlusion and multiple texture image resolution levels, and are dynamically recalculated each time a new frame is rendered. We show how a few additions to the standard graphics hardware pipeline can add this capability without radical architecture changes, and with only moderate hardware cost. The addition of this capability makes it practical to use large textures even over relatively slow network connections.

Categories and Subject Descriptors: I.3.1 **[Computer Graphics]** Hardware Architecture—*graphics processors;* I.3.2 **[Computer Graphics]** Graphics Systems—*distributed/network graphics;* I.3.7 **[Computer Graphics]** Three-Dimensional Graphics and Realism—*Color, shading, shadowing, and texture;* C.2.4 **[Computer-Communication Networks]** Distributed Systems—*distributed applications*

* goss@hpl.hp.com
† ky@jp.hpl.hp.com

1998 Workshop on Graphics Hardware Lisbon Portugal

1. INTRODUCTION

Images in the form of texture maps are frequently used to add detail to surfaces of geometric models used in computer graphics[2]. This allows the addition of considerable perceived surface detail without an increase in actual geometric data size or complexity. The most commonly used form of texture map stores the texture image as a multiple resolution image pyramid, called a MIP map [13]. For a typical image rendered from a scene data base, only a portion of the texture map elements are required, since not all resolution levels of the MIP map are required, and not all surfaces are visible in the image.

In a complex scene with many texture maps, the texture image data may represent a substantial percentage of the storage required for the database, often exceeding the size of the geometric information. In some cases, such as VRML [12] scenes used to display panoramas, the geometry is almost incidental, and the texture imagery is the important content (the NASA Mars Pathfinder VRML panoramas are an example of this type of use [9]). When the scene database to be rendered must be loaded over a relatively slow network connection, it is desirable to minimize the amount of data transmitted by sending only the texture data required for the requested view, and then incrementally loading additional data as required. For many applications, it is even acceptable to display the scene using lower resolution textures while loading additional images. No matter how large the textures used for the scene, the number of texture pixels which have any effect on the rendered image is closely related to the size of that image, so the amount of texture data actually required may be considerably less than the full texture data specified.

If the scene geometry and the texture map sizes and transformation parameters are known, it is possible to determine which texture tiles are required for a particular view. The information required is generated during the rendering of a scene, but current graphics pipeline architectures do not provide a way for an application to recover this information. In this paper we describe extensions to a conventional graphics pipeline architecture (such as the OpenGL pipeline [10]) which allow efficient gathering of this data. By dividing each resolution layer of each texture into tiles, and counting the number of visible pixels which reference each unloaded tile, we can determine which tiles have the greatest effect on the current view. Tiles can then be fetched from the server based on tile visibility in the scene.

1.1 Related Work

Prior work has been done by other researchers using recovery of surface visibility information from the frame buffer of a graphics

pipeline. An unmodified graphics pipeline can be easily used to determine surface visibility in a scene by assigning a unique color to each scene element, rendering the scene using these colors, and then examining the frame buffer pixels. For example, this method has been used to compute surface visibility for Hemi-Cube radiosity form factors [1,3]. A similar technique could be used to determine texture tile visibility by filling each tile with a unique color. This would, however, require rendering each frame an extra time with different texture images, and would also require a search of the frame buffer afterwards. In an interactive system, the reduction of the frame rate by a factor of two would usually be unacceptable. The method we present in this paper avoids this overhead by modifying the hardware pipeline so that neither re-rendering nor searching the frame buffer is required.

Deferred shading [5,8] is another area related to the work we present. A conventional graphics pipeline shades each pixel as it is rasterized. Even if the shading operation is performed after the depth test, another object closer to the viewer may later overwrite the earlier shading results. Deferred shading techniques postpone shading calculations for a pixel until the visibility of the pixel is completely decided (usually at the completion of rasterization). At the point where the pixel is shaded, visibility has been completely determined, and so final texture references for the pixel are also known. Full deferred shading requires sufficient storage per pixel for full shading information (lighting, texturing, etc.), and so requires a large increase in frame buffer size, or rendering of the scene in several pieces using a tiled frame buffer. The work we describe in this paper implements a lower cost solution than deferred shading, suitable for implementation in consumer-level graphics accelerators. We have described elsewhere a different texture tile loading method more suitable for use in a deferred shading architecture [14].

2. THE TILED TEXTURE IMAGE PYRAMID

In order to avoid waiting for a complete texture to be loaded, the texture must be subdivided into sections which can be loaded independently. In addition, if a portion of a texture or an entire texture is visible in a scene but at reduced resolution, it may be desirable to load a smaller, lower resolution version of the texture initially, and load the full resolution version later or not at all. Little extra overhead would be involved if a progressive image format were used, such as a tiled version of progressive JPEG or a wavelet-based method. Without a progressive format, however, the worst case (loading rather than synthesizing all MIP resolution levels) involves loading only 33% more data than just the full resolution image, much of which can usually be done in the background.

We base our texture tiling scheme on the existing MIP map image pyramid. Each level of the MIP map pyramid is a rectangular image. The size in each direction is equal to some power of two. Tiles are square, and each image in the pyramid is divided equally into tiles. Some tiles may be partially empty for image levels which are smaller than one tile in either dimension.

The texture tiling scheme used here is intended to be compatible with the *FlashPix* image format [6] and *Internet Imaging Protocol* (IIP) standards maintained by the Digital Imaging Group (DIG) consortium [7]. FlashPix stores an image as a tiled multi-resolution pyramid. Tiles are compressed using the JPEG compression standard. IIP allows image tiles to be downloaded on demand from a server by a client.

2.1 Tiled Pyramid Initialization

To initialize the tiled pyramid structure for a texture, we need to know only the size of the full resolution image and the top (1×1 pixel) level of the pyramid. With this information, a skeleton texture pyramid can be constructed. This pyramid contains the storage for the texture tile image pixels, which have not yet been loaded. A single bit *tile-present* flag is stored for each tile. These flags are initialized to false for all tiles except for the top level. At the top level, the single pixel value is stored, the *tile-present* flag is set true, and the "tile loading and synthesis" procedure described below is performed for the top level tile.

2.2 Tile Loading and Synthesis

Until all tiles are present in a texture map pyramid, there will be some gaps. These gaps will gradually be filled by tiles loaded from the server system, and also by synthesis of low resolution tiles from multiple high resolution tiles (a miniature version of the current synthesis of MIP map levels from the entire texture image).

Whenever a new tile is loaded into a texture map pyramid, tiles at other resolutions which cover overlapping areas (with regard to the base image) are potentially updated. Whenever the tile-present flag changes from false to true for a tile (the "current" tile), the following procedure is applied:

1. If the current tile is not at the highest resolution level, and if any of the (usually) four tiles covering the same area at the next higher resolution level (the "children" of the current tile) have a tile-present value of false, fill those child tiles with interpolated or pixel-replicated values from the current tile (do not mark those tiles as present). Recursively repeat this step for any child tiles which are filled.
2. If not at the lowest resolution (1×1 pixel) level of the pyramid, and if the lower resolution parent tile has tile-present value of false, examine the siblings of the current tile (the other tiles at the current level which are children of the same parent). If all other siblings have tile-present set to true, set the parent tile-present value to true and synthesize the lower-resolution parent from the children (similar to the normal MIP map synthesis). Then recursively execute this step for the parent.

In order to have some texture detail present in the initial rendered image, an application may wish to pre-load one tile for each texture at the lowest resolution which has a full tile present. The steps above will then synthesize all the lower resolution levels in the pyramid and mark them present, and will fill the higher resolution levels with usable interpolated or replicated texture pixel values without marking them as present.

The filling of absent higher resolution tiles (step 1 above) requires some overhead during initialization and whenever a tile is loaded

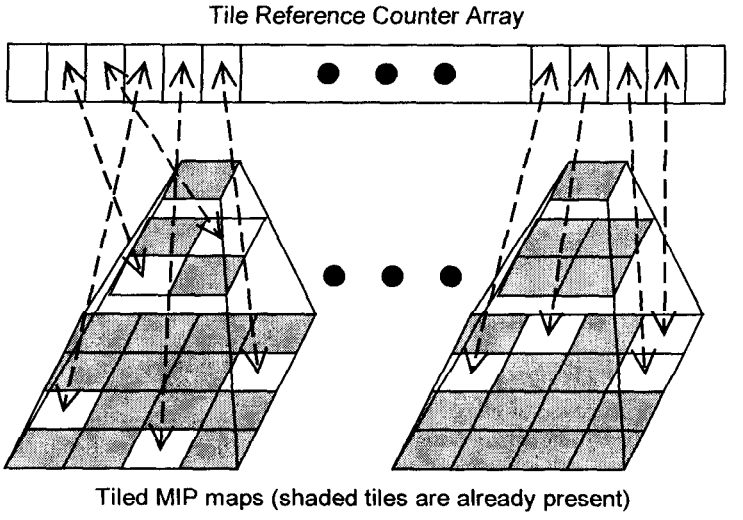

Figure 1 - Tile Reference Counter Array

over the network. No additional hardware is required, since this process can be performed easily by software. The alternative to this data extrapolation would be to add a mechanism to the texture mapping process to redirect texture pixel fetches from an absent tile to the next lower resolution tile which is present. This per-pixel overhead would require additional hardware complexity and would slow the texture mapping process.

3. TEXTURE TILE VISIBILITY DETERMINATION

When a textured surface A is invisible in a scene due to occlusion by another surface B, it is still possible that A will be rasterized (and the associated texture tiles used) if A is sent through the graphics pipeline before B. Some or all of the texture pixels (and tiles) used by A may have no effect on the final rendered image.

It is possible to take occlusion into account when gathering information on texture tile usage by keeping some simple information in the frame buffer for each pixel while rendering a scene, and also storing a small amount of information with each texture tile in addition to the tile-present bit discussed above. To keep track of which tiles are needed for a full-resolution rendering of a scene, we add a tile reference counter array, and a reference counter index for each pixel.

3.1 The Tile Reference Counter Array

This array contains one counter for each possible texture tile being tracked. Each reference counter must have sufficient bits to count up to the number of pixels in the frame buffer (20 to 22 bits for most current systems). Each tile which has not yet been loaded is assigned a non-zero *tile-index* number, which uniquely identifies the tile across all tiles of all texture images in use, and is the index into the reference counter array. This index is stored with the tile in the texture memory. Tiles which have already been loaded (tile-present set to true) are assigned a 0 index, as are tiles for which an index is not available yet (if the reference counter array is full). Strategies to deal with situations when too many texture tiles need to be tracked are presented later (section 3.6). These strategies will not change the basic method.

Figure 1 shows the relationship of unloaded tiles to entries in the tile reference counter array. Shaded tiles have already been loaded.

3.2 The Pixel Texture Tile Index

Each pixel in the frame buffer will have the usual R, G, B, and Z (depth) values, and in addition will also have an unsigned integer index into the counter array (Figure 2). During rasterization, this texture tile index will be set to the index value of a texture tile used to color the pixel. If the pixel is filled by an untextured object, the index is set to 0. A non-zero index will therefore indicate that a not-yet-loaded texture tile was used to color the pixel. For the moment assume that only one tile index per pixel is required, corresponding to OpenGL texture sampling mode "GL_NEAREST_MIPMAP_NEAREST" or "GL_NEAREST" (see section 3.7 below for interpolated or multi-texture cases).

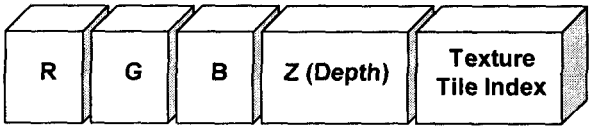

Figure 2 - Frame Buffer Pixel Format

At the start of rendering of a frame (when the Z buffer clear is performed), all the pixel texture tile index values are set to 0, which indicates the pixel does not reference any texture tile being tracked. All the reference counters in the array are set to zero (the value of the counter at index 0 can be set to the number of pixels in the frame buffer for later use as a consistency check).

Some graphics architectures have resorted to the subdivision of the screen into relatively small, separately rendered "buckets" or "chunks" in order to reduce the size of the frame buffer [4]. The method we describe here is completely compatible with such a screen subdivision. When using buckets, the tile reference counter array is zeroed only at the start of the frame, while the pixel texture tile indices are reset with the Z buffer at the start of rendering of each bucket.

3.3 Per-Pixel Rendering Operations

The image is rendered using a standard Z-buffered rasterizer, with a modification to the pixel storage operation at each pixel. The texture sampling operation will supply the texture tile index number along with the sampled texture value (an index of 0 is used for untextured pixels). When a new pixel value is stored as the result of a successful Z test (and any stencil test, stippling, etc.), additional operations are performed:

1. The tile reference counter indicated by the index previously stored in the pixel (old index) is decremented by one.
2. The tile reference counter indicated by the index supplied by the rasterizer (new index) is incremented by one.
3. The new index is stored for the pixel.

These steps can be skipped if the old index and new index are equal.

This procedure in effect maintains a count of the number of visible pixels which are affected by each non-loaded texture tile. Whenever a pixel affected by a tile is written into the frame buffer, the count for that tile is incremented. If the pixel is later overwritten by a pixel associated with another tile (or no tile), that counter is decremented, and the counter for the new tile is incremented.

3.4 Per-Frame Rendering Operations

As mentioned above, the reference counter array and the pixel tile index values must be initialized at the start of a frame. When rendering of the image is complete, the frame buffer contains the best quality version of the image using the currently available texture tiles. The reference counters indicate how many pixels would be affected by each tile which has not yet been loaded, using the most desirable resolutions (rather than the resolutions actually used to generate the image). The reference counts can be used to influence the order in which tiles are loaded in order to best improve the quality of frames rendered later, since the tiles having the highest reference counts will affect the most pixels.*

A scan of the reference counter array at the end of a frame will yield the indices of one or several non-loaded texture tiles which affect the most pixels on the screen. The application would typically request these tiles next from the server. While waiting for these tiles, the application can continue rendering frames for the user and responding to user inputs. As tiles arrive and are loaded into the texture image pyramids, the scene can be re-rendered to get an updated set of tile reference counters to determine subsequent tiles to be loaded. Since rendering and tile loading proceed in parallel, changing view parameters may change the set of tiles used, so the queue of tiles to be loaded should be updated each time a request is to be made to the server.

Notice that the number of reference counter array entries to be scanned becomes smaller as more tiles are loaded. Whenever a tile is loaded, its tile index is set to zero. At that time, the tile with the largest index can be reassigned to the index of the loaded tile, decreasing the size of the array to be scanned by one entry. Also notice that the size of the array is normally much smaller than the frame buffer, and so this part of the operation is more efficient than methods which require the application to gather visibility statistics by scanning all of the frame buffer pixels.

3.5 Hardware/Software Task Subdivision

This entire algorithm can, of course, be implemented in software, in which case the use of an index array can be supplanted by the use of pointers to possibly gain some simplicity (this was done in the test implementation used to generated the images in the figures). The main motivation behind this design, however, was a system which could be added to existing rendering hardware without greatly increasing the complexity. The additional hardware components required are the following:

1. Tile reference counter array (with capability to transfer entries to main memory)
2. Tile index stored in each pixel (no path to main memory required)
3. Tile index and tile-present bit stored in texture memory for each tile
4. Rasterizer hardware to perform per-pixel operations

3.6 Tile Index Management

As mentioned above, it is possible that there will not be sufficient entries in the tile reference counter array for all non-loaded tiles. This can be dealt with during index assignment in a variety of ways which involve retaining counter information over multiple frames. One simple method is to render multiple frames, with a different group of tiles assigned non-zero counter indices each frame, until all tiles have been processed. The counts from the sequence of frames are retained, and are searched for the largest values only after all tiles are processed. The sequence is then repeated with any updated tile indices due to loaded tiles.

For example, assume that the highest possible counter index is 3, and that 5 texture tiles (A, B, C, D, E) are currently not present (unrealistically small numbers are used for clarity of the example). The tile reference counts can be accumulated over two frames. In the first frame, the tile indices assigned are A=1, B=2, C=3, D=0, E=0. After rendering, the counter values are saved, and the second frame is rendered with tile index assignments A=0, B=0, C=0, D=1, E=2. This results in a reference count for each tile.

* If counter 0 was initialized to the number of pixels in the frame buffer and the other counters were initialized to 0, the sum of all the counters after rendering should be the number of pixels in the frame buffer, a useful consistency check during hardware or software development.

This method has the advantage of working with a relatively small reference counter array, minimizing hardware costs. It does, however, provide data which may be slightly out of date if the viewing parameters change over the frames used, and it may result in slower tile loading or a less optimal tile loading order.

3.7 Multiple Texture Samples Per Pixel

If texture pixel interpolation is used, or if multiple textures are used per pixel, several texture tile indices may be generated for a single pixel. In this case, the texture sampling hardware must select only one of the indices for the pixel out of all non-zero values generated. This will result in some texture tiles not having all their affected pixels counted. This is only a temporary problem, since eventually the tile that does get counted at a pixel will be loaded and its index will become zero, allowing another of the tiles to be counted, etc. The main observable result will be a possible sub-optimal order of tile loading. It would be possible to calculate a more accurate count by other methods such as storing multiple tile indices per pixel, but this would greatly increase hardware cost.

The multi-frame method described in the previous section could also be adapted for the multiple texture per-pixel case. If multiple textures are mapped to the same surface, a non-zero index would only be assigned to one of those textures at a time, a different one each frame. Over a number of frames equal to the number of textures per surface, all the tile references would be counted.

4. TIMING AND DEPENDENCY ANALYSIS

We do not at this point have a sufficiently low level hardware design to perform a detailed timing analysis. It is possible, however, to analyze the interactions of the architectural changes with the standard graphics pipeline and determine where the additional functions can be performed in parallel with standard functions. This analysis is divided into two parts, per-pixel effects and per-frame effects.

4.1 Per-Pixel Effects

Each time a pixel (whether textured or not) passes the Z-buffer depth test during rasterization of a polygon, the following additional operations must be performed for texture tile visibility determination:

1. Fetch (old) texture tile index for the pixel from the frame buffer.
2. Decrement (old) indexed counter.
3. Fetch (new) texture tile index for replacement pixel from the texture memory.
4. Increment (new) indexed counter.
5. Store (new) texture tile index for the pixel.

These steps must be performed for untextured as well as textured pixels, since an untextured pixel may overwrite a textured pixel. We assume for this analysis that texture image access is not performed until after the depth test.

If sufficient frame buffer memory bandwidth is available, step (1) can be performed in parallel with the fetch of the stored depth (Z) value at the pixel (the tile index is simply discarded if the depth test fails). Step (2) can be performed in parallel with the texture image access, since it does not require the new tile index. The time required should be less than the texture access time, since the complexity is far less.

For non-textured pixels, step (3) involves merely supplying a new tile index of zero. For textured pixels, step (3) requires fetching a texture tile index stored with the texture tile pixels being accessed. This may require time for an additional memory access, or it could be done in parallel with the fetch of the texture pixel RGB values if memory bandwidth allows.

Steps (4) and (5) can be performed simultaneously once step (3) is complete, and require relatively little time. Given sufficient memory bandwidth for steps 1-3, the additional time required for the entire per-pixel process would be the longer of the times for steps 4 and 5.

4.2 Per-Frame Effects

Before every frame, the reference counter array and the pixel texture tile indices must be cleared. It is possible to perform these steps in parallel with the Z-buffer clear operation if the memory architecture allows, in which case no additional time is required.

4.3 Tile Loading Effects

Whenever the application is ready to request a tile from the server, it must read the reference counter array from the hardware at the conclusion of a frame (since tile load times may span several frames, this would not necessarily be done every frame). The application would then scan the array to choose a tile or tiles to request. In a multi-threaded application, once the array has been read the other processing can occur interleaved with or in parallel with sending data for the next frame to the graphics accelerator.

Whenever a tile arrives from the server, the application must perform the tile loading and synthesis procedure (section 2.2). In a multithreaded application, any interpolation, replication, and synthesis can occur interleaved with or in parallel with the rendering of the current frame. Loading of pixels into texture memory and changing tile indices would typically occur at the end of the current frame.

5. RESULTS

Texture tile visibility determination has been implemented in software in order to test the algorithms and simulate the visual effects. The test implementation was built as a set of modifications to Mesa, a freely-available, unofficial implementation of the OpenGL graphics pipeline [11]. Plate 1 shows a sequence of images generated using the software implementation. The bright yellow areas show pixels affected by the latest tile loaded. The video accompanying this paper shows the algorithm in operation over a longer sequence of frames.

These images were generated using a tile size of 32×32 pixels, at a frame buffer resolution of 512×384 pixels. To make the yellow highlights clearly visible, a single texture sample was used per pixel, using nearest neighbor sampling at the texture level closest to the calculated level (OpenGL texture sampling mode "GL_NEAREST_MIPMAP_NEAREST").

For the software implementation, tile loading was simulated to simplify implementation. Actual performance measurements would not have been possible even if tiles were loaded over the network, since the software implementation is not optimized, and variations in network traffic would make consistent results hard to achieve without a dedicated network. A hardware implementation or more extensive, optimized software implementation would be necessary to gather performance data.

The accompanying video tape shows two animation sequences, "Cubes," a simple scene with two cubes each having one texture map, and "Home," a more complex environment. All images were rendered off-line using the software test implementation. Each sequence is shown first with yellow highlights for pixels shaded by the most recently loaded tile, and then again without the highlights as a user would actually view the scene. At each change of view point (every half second in the videos), one tile is loaded (3KB uncompressed).

The Cubes scene database contains only two cubes (12 polygons) and two texture maps. The mandrill is a 512×512 image, the garage is a 1024×512 image. Notice that as the cubes rotate and the field of view changes, the MIP map levels in use change and the texture tile loading adapts to the tiles used in the last frame.

The Home scene database contains 50538 polygons, the vast majority of which are textured, using 22 RGB texture images with total size approximately 17MB (uncompressed). The initial scene has loaded one 32×32 pixel tile for each of the 22 textures, for a total of about 67KB, about 0.4% of the total pixels. At the last frame, 100 additional tiles have been loaded (300KB), for a total of only about 2.2% of the size of the entire texture data. The image quality is reasonably good and the wait time is drastically shorter than the time required to load the entire texture database. Plate 2 shows some sample frames from the Home video.

6. CONCLUSIONS

We have described additions to the standard computer graphics hardware pipeline which would allow dramatic decreases in the time a user waits to see rendered images of complex textured scenes when the scene textures must be downloaded over a network. The initial textured image of the scene will be displayed with minimal waiting time by initially loading only one tile per texture. If the majority of the polygons in the scene do not use the maximum texture resolution levels or if many textured surfaces are occluded (quite possibly the normal case), the final image at best resolution will also be displayed much more quickly. The user is also able to interact normally with a scene while additional texture tiles are loaded, and viewpoint and occlusion changes are taken into account dynamically as tiles are loaded.

The hardware modifications we describe could be incorporated into a consumer-level graphics card without major cost increases, and would afford great benefits even in an optimized software renderer. The addition of this capability makes it practical to use large textures even over relatively slow network connections.

7. ACKNOWLEDGEMENTS

Brian Paul (and other Mesa contributors) contributed greatly to this work by writing and making available to the public the Mesa graphics library, which was an excellent base for the software implementation of the texture tile reference counting method.

REFERENCES

1. Daniel R. Baum, Holly E. Rushmeier, and James M. Winget, "Improved Radiosity Solutions Through the Use of Analytically Determined Form-Factors," *Computer Graphics*, Vol. 23, No. 3, July 1989 (Proceedings of *SIGGRAPH '89*), pp. 325-334.
2. James F. Blinn and Martin E. Newell, "Texture and Reflection in Computer Generated Images," *Communications of the ACM*, Vol. 19, No. 10, Oct. 1976, pp. 542-546.
3. Michael F. Cohen and Donald P. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments," *Computer Graphics*, Vol. 19, No. 3, July 1985 (Proceedings of *SIGGRAPH '85*), pp. 31-40.
4. Michael Cox and Narendra Bhandari, "Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC," *Proceedings: 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 1997, pp. 25-34.
5. Michael Deering et al., "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics," *Computer Graphics*, Vol. 22, No. 4, August 1988 (Proceedings of *SIGGRAPH '88*), pp. 21-30.
6. Eastman Kodak Co., *FlashPix Format Specification*, version 1.0.1, July 1997 (http://www.digitalimaging.org).
7. Hewlett-Packard Co., Live Picture Inc. and Eastman Kodak Co., *Internet Imaging Protocol*, version 1.0.5, October 1997 (http://www.digitalimaging.org).
8. Steven Molnar, John Eyles, and John Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics*, Vol. 26, No. 2, July 1992 (Proceedings of *SIGGRAPH '92*), pp. 231-240.
9. NASA, *Mars Pathfinder Virtual Reality Models and Animations of the Pathfinder Mission*, http://mars.jpl.nasa.gov/vrml/vrml.html
10. OpenGL ARB, *OpenGL Reference Manual*, 2nd ed., R. Kempf and C. Frazier eds., Addison-Wesley, 1997.
11. Brian Paul, *The Mesa 3-D Graphics Library*, http://www.ssec.wisc.edu/~brianp/Mesa.html
12. VRML Consortium, *The Virtual Reality Modeling Language*, International Standard ISO/IEC 14772-1:1997.
13. Lance Williams, "Pyramidal Parametrics," *Computer Graphics*, Vol. 17, No. 3, July 1983 (Proceedings of *SIGGRAPH '83*), pp. 1-11.
14. Kei Yuasa and Michael E. Goss, "Lazy Texture Loading for Network Graphics Objects" (unpublished manuscript), January 1998.