

# Interactive Display Of Very Large Textures

David Cline<sup>1</sup>

Parris K. Egbert<sup>2</sup>

Computer Science Department, Brigham Young University

## Abstract

Large textures cause bottlenecks in real-time applications that often lead to a loss of interactivity. These performance bottlenecks occur because of disk and network transfer, texture translation, and memory swapping. We present a software solution that alleviates the problems associated with large textures by treating texture as a bandwidth-limited resource rather than a finite resource. As a result the display of large textures is reduced to a caching problem in which texture memory serves as the primary cache for texture data, main memory the secondary cache, and local disk the tertiary cache. By using this cache hierarchy, applications are able to maintain real-time performance while displaying textures hundreds of times larger than can fit into texture memory.

**CR Categories:** I.3.7 [ComputerGraphics]: Picture/Image Generation—texture

**Keywords:** Texture Caching, Bandwidth-Limited Resource, Texture Mapping, Real-Time Display, Interactivity

## 1 Introduction

A texture map, or texture, is an image used to provide surface detail in a computer generated scene. At the most basic level, textures supply a scene with “visual interest”. On a much deeper level, textures provide important visual clues to the user and help increase the perceived realism of the environment. In some cases, such as [19], textures replace complex geometry. Image based rendering techniques, such as Quicktime VR [6], take this idea to its limit, reducing a scene to just a few primitives that serve as projection screens for textures.

When textures form an important part of scene content, reduction in texture detail may mean a loss of scene comprehension. In addition, interactivity may be important to scene understanding, so that when the frame rate drops, so does the scene’s utility. Some high-end platforms are designed to display very large textures in real time, but the cost of such hardware makes it prohibitively expensive for most users. Mid-range to low-end graphics workstations can usually draw some textured objects in real time, but they are not generally equipped to deal with large textures directly. This paper presents a software approach to texture caching designed to allow interactive display of very large textures on mid to low-end workstations. The new approach has as its basis the following goals:

- Very large textures should be supported, larger than can be held in either texture or main memories.
- Scene startup time should be as fast as possible, and independent of the textures present.
- The system should be implementable on existing low-end graphics workstations using current graphics API’s.
- The difference in frame rate between an application using texture caching and the same application using static textures that all fit into texture memory should be negligible.

- Maintaining an interactive frame rate should take precedence over instantaneous detail.
- The system should perform reasonably well when textures must be fetched over a slow network.
- The system should form a general framework for a wide variety of applications that require large textures.

### 1.1 Related Work

Most research in texture mapping to date has focused on four areas: anti-aliasing techniques, acceleration of texture rendering, applying texture mapping to new rendering problems, and texture synthesis. Since the development of low-cost rendering hardware, however, increasing the size of textures that can be practically rendered has become a popular avenue for investigation. Such research is needed because the demand for texture processing has more than kept pace with hardware advances. This trend does not show signs of letting up, so for the foreseeable future texture memory alone will not be sufficient to handle users’ wants.

The remainder of this section discusses techniques designed to overcome the hardware-imposed upper limit on texture size. Included among these are texture compression and texture caching algorithms. Other areas of interest that will be discussed are progressive image transmission and methods for dealing with bandwidth-limited resources.

**Texture compression.** Ideally, texture compression should take place in texture memory, effectively increasing its size. By using compression the hardware can accommodate larger textured scenes without swapping. One shortcoming of algorithms that keep textures compressed in texture memory is that they require hardware support, so the compression algorithm must be built into the hardware. Although not usually considered to contain texture compression facilities, Iris GL [12] provides three rudimentary examples of texture compression in texture memory. These include selection of texel size, texture lookup tables, and detail textures. Another example of hardware texture compression is found in the Talisman architecture. Talisman [21] employs a hardware-based compression scheme similar to JPEG that keeps textures in a compressed state in texture memory.

Software approaches to texture compression cannot increase the effective size of texture memory. Instead, they allow more texture to be held in main memory, reducing the need for virtual memory swapping. To be effective, software-based texture compression must allow for fast texture decoding, so that decompression time does not dominate rendering activities. Beers et al. [1], and Levoy and Hanrahan [10] advocate the use of *vector quantization* to compress textures held in main memory precisely because of decoding speed.

**Texture caching.** Even with compression, texture data may be larger than the memory capacity. When this occurs, some form of caching must be used to manage texture data. A good portion of the texture caching algorithms described in the literature are application-specific measures designed to solve a particular caching

<sup>1</sup>cline@neptune.cs.byu.edu, <http://orca.cs.byu.edu/~cline>

<sup>2</sup>egbert@cs.byu.edu

problem. For example, Quicktime VR [6] cuts texture panoramas into vertical strips for caching purposes. Cohen-Or et al. [8] describe a system that caches photo-textured terrain data. The data is fetched at different resolutions based on the distance from the viewer. Oborn [14] also uses a distance metric to determine the texture level of detail that will be fetched. In another terrain viewing application, Lindstrom et al. [11] use the angle at which textures are viewed to reduce texture requests over using a simple distance metric. Blow [2, 3] gives an overview of several texture caching schemes currently in use in the video game industry. He also discusses the implementation details of the caching system used to manage large terrain textures in a video game setting.

Besides problem-specific texture caching solutions, some work has been done to provide general support for very large textures in non real-time applications. Pharr et al. [16] use a caching scheme that dices textures into small tiles that are paged into memory as needed. Using this architecture, their ray tracing software can efficiently render scenes with textures many times larger than the texture cache. SGI's ImageVision library [13] uses a similar caching system based on cutting images into "pages" that can be manipulated efficiently.

Unfortunately, none of the above mentioned systems serves as a general framework for the display of large textures in interactive environments. One system that is designed for real-time display of large textures is SGI's InfiniteReality Engine [17]. It uses a virtual texture scheme called a "clip-map" to deal with textures too large to fit in memory. Unlike most texture caching systems, the clip-map algorithm does not cut a large texture into tiles. This has the benefit that since there are no texture tiles, scene geometry does not have to be diced along tile boundaries. Although the clip-map algorithm can produce impressive results, it has several drawbacks that make it a bad choice for current low-end hardware. First, it requires a large hardware texture capacity.<sup>3</sup> Second, an efficient implementation of clip-mapping for real-time display is likely to require some hardware assistance, so it is probably the wrong choice for a software add-on. Finally, the clip-map algorithm relies on fast local disk access to retrieve texture data, and so does not serve as a framework for texture caching in low bandwidth situations.

**Topics related to texture caching.** A successful texture caching strategy for low-end systems needs to utilize available bandwidth effectively. Thus, progressive image transmission algorithms and methods designed to deal with limited bandwidth situations are related to the display of very large textures. This is especially true for textures stored remotely and accessed across a slow network.

Hill et al. [9] describe an early attempt to use progressive image transmission. Their system was designed to send Landsat images over a slow modem connection. It worked by selectively refining user-specified areas of interest. In more recent work, Strobel et al. [20] suggest a method to store and retrieve large image databases based on wavelet transforms.

Bukowski and Sequin [4] faced a more dynamic data stream than still images. They designed a viewing program to display information from a fire simulation. The program allowed users to generate queries for a fire simulation running on a remote machine. A "bandwidth manager" was used to optimize data requests.

<sup>3</sup>Clip-mapping as described in [17] requires more than 22 million texels, or about 85 MB of texture memory for a 32-bit color implementation. This number may be reduced to under 7 million texels for 640 × 480 graphics screens, however.

## 2 Typical Texture Loading

The common approach to texture mapping in interactive applications is to treat texture memory as a finite resource, and texture images as atomic resources. Such a texture paradigm leads to two very specific texture loading practices. First, because texture memory is viewed as a finite resource, there is an upper limit on the number of bytes of texture that a scene may contain. Second, since scene designers view texture images as atomic resources, they usually store each image in one piece on disk, and often do not store MIP maps on disk at all, preferring to let the viewing program create the MIP maps when loading textures. At run time, the viewing program loads every texture image in a scene before any interaction takes place, and the user must wait while the textures are loaded. Figure 1 shows the common approach to texture mapping in real-time applications.

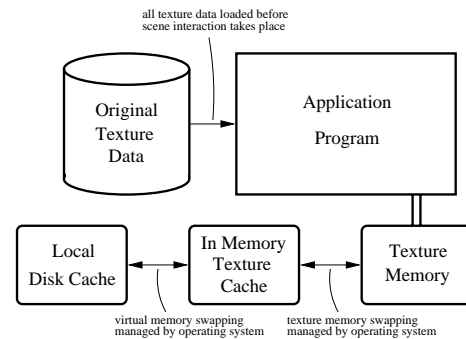


Figure 1: Typical texture loading for real-time applications

### 2.1 Performance Bottlenecks

Treating texture as a finite and atomic resource is conceptually simple and works well in many situations, but it breaks down as texture size increases. When a scene's texture memory requirements begin to exceed texture memory size, four performance bottlenecks appear that can prevent real-time interaction. They are

- *Disk or network transfer.* If large textures must be loaded from disk or across a network before interaction takes place, the user may be subjected to a significant initial wait. In addition, a disk access causes the process which initiated it to block until the data transfer completes; consequently, the user cannot interact with a process while it accesses the disk.
- *Texture translation.* If a texture image is stored on disk in a compressed format it must be decompressed before use. Additionally, odd sized textures must be resized to have widths and heights that are powers of 2. Finally, MIP maps may be created from the original texture images. Since these steps may be CPU intensive, they can prevent real-time interaction with the scene even if the viewing program is designed to allow interaction while loading textures.
- *Texture memory thrashing.* Most hardware texture mapping systems place textures in dedicated texture memory. If texture memory requirements exceed texture memory size, textures are swapped to main memory by the operating system. When large amounts of texture must be transferred from main to texture memory for each frame of an interactive application, the frame rate can drop drastically.

- *Virtual memory swapping.* If texture memory requirements exceed main memory size, the operating system swaps textures to disk. When a texture that has been swapped to disk is requested, the interactive program must wait while the texture is reloaded into main and texture memories.

Because the common approach to texture mapping in interactive applications creates performance bottlenecks, it is difficult to accommodate very large textures in real-time scenes without increasing texture memory size. Our solution to the problem of large textures, outlined in the next section, bypasses the operating system’s texture caching policies by explicitly creating a hierarchy of texture caches, with a separate process handling each cache.

### 3 An Approach to Texture Caching

This section presents an overview of our approach to texture caching. At the core, we base our strategy on the assumption that instantaneous texture detail (detail in the current frame) can be sacrificed in favor of a higher frame rate. Thus we do not guarantee that rendered frames will have all the desired texture detail immediately. The ability to sacrifice instantaneous detail differentiates our approach from texture caching algorithms designed simply to accelerate rendering of desired frames. These algorithms, while useful, are essentially virtual memory schemes. Our algorithm goes beyond simple memory management to deal with bandwidth concerns related to real-time display.

The remainder of this section describes the three essential pieces of our algorithm. They are the pre-processing of texture images, the use of texture servers and regulation of texture flow.

#### 3.1 Mip-Map Caching

Although the original purpose of the MIP map [22] was to reduce the cost of antialiasing, it can also be used for caching purposes. In order to do this, the MIP map must be precomputed and stored on disk. By loading coarse versions of the textures initially, then refining those textures as time progresses, startup time for programs that use large textures can be significantly reduced. In addition, if texture detail is loaded on an as-needed basis, some parts of the MIP map may never be fetched during a given run of the program, reducing total disk access.

Taking into account only the need to load coarse versions of images before more detailed versions, one might consider making a separate image file for each level of the MIP map and then load these files as needed. Unfortunately, this approach is ineffective for higher levels in the MIP map because each successive level is four times larger than its parent, and the highest MIP-map level is as large as the original image. What is needed is a mechanism to load manageable pieces of texture from anywhere in the MIP map on an as-needed basis. The route our algorithm takes is to cut the large MIP map into a quadtree [18] of uniform texture tiles. Uniform tiles have several advantages over non-uniform sized tiles, namely, they can be expected to have fairly consistent load times, they simplify memory management issues, and they help prevent memory fragmentation. For the remainder of this paper, the word “quadtree” refers to the quadtree MIP map unless otherwise stated. Figure 2 shows a quadtree MIP map.

#### 3.2 The Texture Servers

One of the biggest obstacles to texture caching for real-time display is the blocking of a process when data is requested from the disk. Because of this blocking, most real-time applications do not access the disk extensively during interaction. Instead, the data that a scene

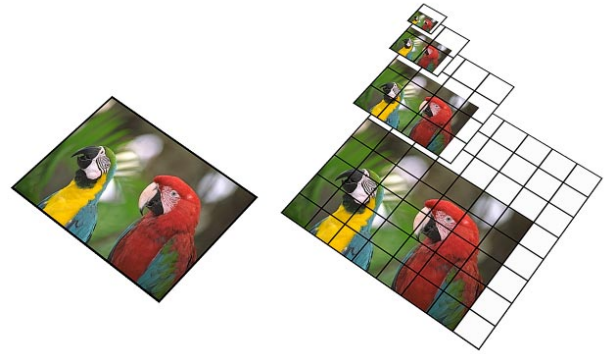


Figure 2: Original image (left) and quadtree MIP map of the same image (right). Empty tiles are not stored.

needs is loaded before interaction begins, and there is a delay between scenes while new data is loaded. Pausche [15] used a clever trick to get around stopping interaction between scenes. Instead of pausing, he required the user to fly through a long corridor containing minimal texture. The system then had time to swap textures for the new scene into texture memory. For our texture caching system, however, we needed to make a general framework that would allow continuous texture loading simultaneous to normal scene interaction. To prevent blocking during interaction, our system employs an additional process called the Texture Server. The Texture Server handles texture loading and decompression, and manages an in-memory texture cache.

A problem similar to disk blocking can occur when texture data resides across a slow network. If textures from the network are cached on a local disk, texture access time becomes bimodal. That is, access to textures already in the local disk cache is fast, but access to textures that must be downloaded over the network is slow, and the Texture Server may end up waiting for the network when it could be filling requests for textures stored locally. The solution to this problem is to employ a third process, the Meta-texture Server, to download textures over a slow network.

By using a Texture Server and Meta-texture Server, a three-level hierarchy of texture caches emerges, each level tended by a separate process. The application program regulates texture memory, the Texture Server takes care of the main memory cache, and the Meta-texture Server tends the local disk cache. Figure 3 shows how the application and texture servers work together.

#### 3.3 Texture as a Bandwidth-Limited Resource

All computers have limited bandwidth. By bandwidth we mean time-constrained resources, such as CPU cycles per second, disk access rates, and network speed. Any real-time application requires some percentage of the computer’s bandwidth to run. If *texture flow* (loading texture from disk or over a network, decompressing textures, copying textures and texture definition) consumes too much bandwidth, real-time performance will be lost. For this reason the application program and the texture servers must work together to constrain texture flow so that sufficient bandwidth is available for scene interaction.

The application program’s main responsibility with respect to texture flow is to prevent texture memory thrashing. Thus, the application may only use as much texture at a time as can fit into texture memory. In addition, the application must decide what textures to discard when texture memory is full. When a desired piece of texture is not resident in texture memory, the application requests it

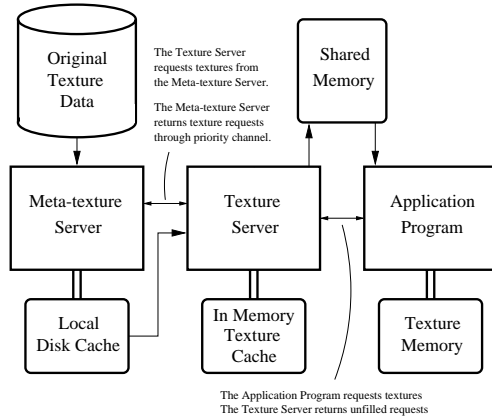


Figure 3: Texture management using texture servers. Bold arrows indicate texture flow. Other arrows indicate inter-process communication.

from the Texture Server, in the meantime drawing the scene with a coarse version of the desired texture if one is available. The application must be careful to limit the rate at which it requests texture fragments, as the Texture Server may get bogged down processing old requests. Since it usually cannot request every texture that it wants, the application collects potential requests in a priority queue, called the “pre-queue”, as it draws the scene. At the end of each frame, the pre-queue is used to fill the Texture Server’s request queue, and then is cleared.

The Texture Server has several responsibilities related to texture flow. To prevent virtual memory swapping, the Texture Server limits the size of the in-memory texture cache. In addition, the Texture Server must limit the rate at which it loads textures, decompresses them, and transfers them to the application. If the Texture Server works too quickly it will consume needed machine bandwidth, reducing interactivity.

The Meta-texture Server may begin to use excessive machine bandwidth in some circumstances. For example, If the application program requires data from across the network, the Meta-texture Server must make sure not to saturate network bandwidth, preventing vital non-texture data from reaching the application program. Another example of the Meta-texture Server using too much bandwidth is if it is given the task of decompressing texture files as they come across the network. Compute-intensive image decompression may consume needed CPU bandwidth, preventing interactivity.

## 4 Implementation

This section describes our implementation of an application program, Texture Server, and Meta-texture Server. The application, a program to view phototextured terrain, uses a quadtree based terrain decimation scheme called Q-morphing [7] that meshes well with the texture caching system. As mentioned in Section 3.1, the MIP-map tiles for the terrain texture are computed in advance and stored on disk.

### 4.1 Finding the Right Texture Tile

Graphics APIs that provide MIP-mapping generally compute the MIP-map level that will be used for rendering automatically. Our system cannot use this facility, however. Instead, we explicitly compute the appropriate MIP-map level to use for each polygon or group

of polygons to be rendered. This is done in a three step process. First, the *screen radius* of the polygon is calculated. Second, the *texture radius* of the polygon is calculated for a texture tile somewhere in the quadtree. Finally, the algorithm ascends or descends the quadtree until the polygon’s texture radius matches its screen radius as closely as possible.

**Computing the screen radius.** The *screen radius* of a polygon is an estimate of the polygon’s rendered size. It is defined as the length of the semi-minor axis of the projected bounding circle of the polygon. A good approximation to the screen radius of a polygon, or group of polygons, is derived in [11]. We use a slightly modified version of this formula:

$$r_s = \frac{h}{\psi} \times \frac{r \cos \phi}{d} \quad (1)$$

where

- $r_s$  = screen radius of the polygon
- $h$  = height of viewport in pixels
- $\psi$  = vertical field of view angle in radians
- $r$  = world radius of polygon (maximum distance from centroid to a vertex in world space)
- $\phi$  = angle between polygon normal and ray from polygon centroid to viewpoint
- $d$  = distance from viewpoint to polygon centroid

It may be inefficient to find a separate texture tile for each polygon, so one tile may be used for a group of polygons. The screen radius of a group of semi-coplanar polygons is defined as:

$$r_s = \frac{h}{\psi} \times \frac{r \cos(\max(0, \phi - v))}{d} \quad (2)$$

where  $v$  is the maximum, or mean, deflection angle between the group’s mean normal and the polygon normals of the group, and other variables are as defined above.

**Computing the texture radius.** The *texture radius* of a polygon is an estimate of the amount of texture that the polygon covers, and is defined as the maximum distance in texels from the centroid of the polygon to any vertex in texture space. Assuming texture coordinates do not change, the texture radius of a polygon on a particular texture tile is constant and need be computed only once.

**Traversing the quadtree.** Given the texture radius of a polygon for a reference texture tile,  $r_{ref}$ , and the screen radius of the same polygon,  $r_s$ , the next task is to find the appropriate level of the quadtree to use for rendering. This is done by ascending or descending the quadtree until  $r_t$ , the texture radius of the polygon, matches most closely with its screen radius. More precisely, the correct tile is found when  $r_t \leq r_s$  and  $2r_t > r_s$ . Note that the texture radius doubles for each level of descent in the quadtree, and halves for each level of ascent. Thus the appropriate level of the quadtree is  $\lfloor \log_2(r_s/r_{ref}) \rfloor$  levels below the reference tile, counting negative values as levels above.

### 4.2 Texture Coordinates

**Ascending and descending the quadtree.** For this discussion, assume that the origin of a texture is the bottom left corner, and texture coordinates  $u$  and  $v$  increase to the right and up respectively. Children in the quadtree are numbered from left to right and bottom to top as follows:

2	3
0	1

Ascending or descending the quadtree requires a modification to a point's texture coordinates. This modification is based on the point's current texture coordinates and its location in the parent or child level. Pseudocode for the texture coordinate modification is given in Figure 4.

```

TextureTile *Ascend( float &U, float &V, int ChildNum, TextureTile *ThisTile )
{
    U *= 0.5;
    if ( ChildNum == 1 OR ChildNum == 3 )
        U += 0.5;
    V *= 0.5;
    if ( ChildNum == 2 OR ChildNum == 3 )
        V += 0.5;
    return Parent( ThisTile );
}

TextureTile *Descend( float &U, float &V, TextureTile *ThisTile )
{
    int ChildNum=0;
    U *= 2.0;
    if ( U > 1.0 )
        U -= 1.0;
        ChildNum += 1;
    V *= 2.0;
    if ( V > 1.0 )
        V -= 1.0;
        ChildNum += 2;
    return Child( ChildNum, ThisTile );
}

```

Figure 4: Texture coordinate modification for ascending and descending the quadtree

Unfortunately, descending the quadtree may cause a single polygon to span multiple texture tiles, requiring a polygon split. To avoid polygon splitting, our terrain viewing application divides the terrain into small "gridlets" that are aligned with the nodes in the texture quadtree. Decisions about which texture tile to use are made on a per-gridlet basis, so no splitting is necessary. A similar partitioning of geometry is not practical for all applications, however, so geometry splitting may be a necessity in some cases. The benefit of polygon splitting is that it allows the quadtree MIP map to be applied to arbitrary polygonal geometries. Splitting may be done at scene creation time or at render time. Splitting at render time has the advantage that it can be done adaptively, producing fewer polygons than splitting done at scene creation time. The disadvantage, of course, is the computational cost of the splitting.

### 4.3 Texture Loading

To load textures, the application program requests image tiles from the Texture Server. If the application were to request textures as fast as the need arose, the Texture Server would get bogged down processing old requests. For this reason, the application program prioritizes requests for texture, sending only the most urgent requests to the Texture Server. This prioritization is done by means of a priority queue called the "pre-queue". Each time the renderer asks for a piece of texture that is not available, a request for that texture is placed in the pre-queue based on a priority that will be explained later. At the end of each frame, the requests in the pre-queue are emptied into the Texture Server's actual request queue up to some maximum depth. The pre-queue is then cleared before the start of the next frame. When a texture request is sent to the Texture Server,

the requested tile is marked as "pending". Texture requests that were pushed out of the pre-queue or that did not make it into the Texture Server's request queue will be regenerated during the next frame. Figure 9 gives pseudocode for the generation of texture requests.

The maximum depth of the request queue going to the Texture Server should be kept as short as possible while still allowing for efficient bandwidth use. This is consistent with Bukowski and Sequin's idea of running a bandwidth-limited resource in "starvation mode" [4]. We tried maximum request queue depths of 1 and 2 with similar results in both cases. The request queue going to the Texture Server was implemented using a datagram socket in our system.

To fill a request for texture, the Texture Server loads the desired image and constructs it in a shared memory segment. The application then copies the image and defines it as a texture. We control which texture is overwritten in texture memory by using a texture handle stack. At texture definition time, a handle is popped off the stack and given to the texture tile. When the texture tile is discarded, its handle is pushed back onto the handle stack.

To keep the texture server from working too quickly, we only provide one location in shared memory where the requested image may be placed. Hence, the Texture Server must wait for the application to release shared memory before it can construct another image. Pseudocode for the Texture Server's main loop is given in Figure 6. Since our system only allows one texture to be defined during a given animation frame, a good portion of texture flow regulation becomes choosing the right texture tile size. We found  $256 \times 256$  tiles to be the best suited for the configurations with which we experimented. Figure 5 gives texture definition times for the Reality Engine and O2.

	Definition Type	RE2		O2	
		gray	color	gray	color
Iris GL	Fast Define	< 0.1	< 0.1	-	-
	No MIP map	11.4	18.2	12.2	50.2
	With MIP map	208.0	216.0	264.0	354.0
OpenGL	No MIP map	4.5	19.9	13.8	32.5
	With MIP map	8.1	26.1	22.9	48.3
Software	Resampling	4.5	11.3	5.0	12.8

Figure 5: Measured times for definition of  $256 \times 256$  texture tiles on a Reality Station 10000 and O2. Both computers have a single R10000 processor. Times are given in milliseconds. The first three rows of the table show times for Iris GL texture definition. The next two give data for OpenGL. The last row of the table shows the time required to create MIP-map images using a simple box filter in software.

We set the maximum queue depth for requests going to the Meta-texture Server to 1. The rationale behind this restriction is that any extra requests in the Meta-texture Server's queue will likely be obsolete before they can be filled. To pass data to the Texture Server, the Meta-texture Server copies files to the disk cache, and then returns the request to the Texture Server through a priority channel. In this way a request from the Meta-texture Server need not be placed in the rear of the Texture Server's request queue. The main loop of the Meta-texture Server is given in Figure 7.

We did not want the application program to have to know in advance which requests can be filled by the Texture Server, and which must go to the Meta-texture Server, so a feedback mechanism is employed which returns requests to the application program unfilled when the Meta-texture Server's queue is full. To process a returned request, the application simply clears the requested texture's pending flag so that it can be requested again. By using the feedback mechanism, the application can make arbitrary texture requests without clogging the queue to the Meta-texture Server. Pseudocode for the application program is given in Figure 8.

```

TextureServerMainLoop()
{
    repeat
        /* GET NEXT TEXTURE REQUEST */
        Wait for message from Application or Meta-texture Server.
        if ( there is a request from the Meta-texture Server )
            Read request from Meta-texture Server.
            gCanSendMetaRequest = TRUE;
        else
            Read request from Application.

        if ( request == "quit" ) return;

        /* FILL THE REQUESTS THAT WE CAN */
        if ( requested bitmap in memory or the disk cache )
            Load requested bitmap if not in memory.
            Wait until shared memory available.
            Construct bitmap in shared memory.
            Release shared memory to application program.

        /* PASS ALONG REQUESTS NOT FILLED */
        else
            if ( gCanSendMetaRequest == TRUE )
                Send current request to Meta-texture Server.
                gCanSendMetaRequest = FALSE;
            else
                Return request to application.

        /* DISCARD IN-MEMORY BITMAPS */
        while ( bitmap memory usage > MAX_bitmap_mem_usage )
            Discard least significant bitmap. /* LRU */
    }
}

```

Figure 6: Texture Server main loop

```

MetaTextureServerMainLoop()
{
    repeat
        Read request from Texture Server.
        if ( request == "quit" ) return;
        Copy requested file or URL to disk cache.
        Send request to Texture Server's priority channel.
        while ( disk cache size > MAX_disk_cache_size )
            Delete least significant bitmap file. /* FIFO */
    }
}

```

Figure 7: Meta-texture Server main loop

#### 4.4 Caching Algorithms

Unlike caching problems in which all requested data must be fetched before computation can take place, our texture caching system relies on the fact that the scene is still useful even if its textures momentarily contain less than full detail. Even so, the most important caching choice that the application program needs to make is which texture tile to request, that is, what priority algorithm to use for the pre-queue. Our approach to this algorithm is to treat texture retrieval as a progressive transmission problem. Thus the application does not request a texture tile until its parent is resident in texture memory or currently pending. In order to keep scene texture quality as uniform as possible, the priority for texture requests is based on the distance between the requested tile and its ideal counterpart in the quadtree. The priority is given by

$$p_r = \frac{r_s}{r_t} \quad (3)$$

where  $p_r$  is the tile's request priority,  $r_s$  the screen radius of a polygon requesting the tile, and  $r_t$  the polygon's texture radius in the

```

/* To be executed after each frame redraw */

ServeTextures()
{
    /* DEFINE TEXTURES */
    while ( there is a texture in shared memory )
        Pop texture handle from handle stack.
        Define texture tile from shared memory using
        popped texture handle.
        Attach texture tile to proper quadtree location.
        Clear texture tile's pending flag.
        Release shared memory to Texture Server.

    /* CLEAR PENDING FLAGS */
    while ( returned request queue not empty )
        Clear pending flag for returned request.

    /* SEND REQUESTS */
    while ( pre-queue not empty AND Texture Server request queue not full )
        Place head of pre-queue in Texture Server request queue.
        Set texture tile's pending flag.
        Clear pre-queue.

    /* DISCARD TILES */
    while ( texture memory usage > MAX_texture_mem_usage )
        Discard least significant texture tile. /* modified LRU */
        Push discarded tile's texture handle on handle stack.
    }
}

```

Figure 8: Pseudocode for the application program

```

TextureTile *TileToUse( TextureTile *DesiredTile, TextureCoordinates &C )
{
    float p;
    TextureTile *T;
    p = PolygonScreenRadius / PolygonTextureRadius;
    T = DesiredTile;
    while ( T not in texture memory )
        if ( T not pending AND ( Parent(T) in texture memory or pending ) )
            place request for T in pre-queue with priority p.
            p *= 2;
            T = Parent( T );
            Modify C using Ascend().
    return T;
}

```

Figure 9: Finding the right texture

requested texture tile. Given the desired texture tile for a polygon, the actual tile used in rendering is found by ascending the quadtree until a resident tile is found, placing requests in the pre-queue as needed. Figure 9 gives pseudocode for this operation.

Determining which texture tiles to discard from the three texture caches (texture memory, main memory, and local disk) is important, but not as vital as choosing which texture to fetch. We use a modified LRU discarding rule for tiles in texture memory that favors keeping tiles based on their distance from the root. The priority is given by

$$p_d = f(l + 1) \quad (4)$$

where  $p_d$  is the discard priority for the tile,  $f$  the number of frames since the tile was last used, and  $l$  the tile's quadtree level, counting root level as zero. For the other two caches, the in-memory cache and the local disk cache, LRU and FIFO are used respectively.

## 4.5 Texture Compression

Image compression in our system plays two important roles. First, it serves to increase the effective size of the main memory and disk caches. Second, it increases the effective transfer rate of texture images over a slow network. In both cases, however, the main purpose of compression is to increase the flow rate of textures from secondary storage to texture memory without degrading real-time performance.

One of our main design goals was to make a texture caching system compatible with a wide variety of existing hardware. For this reason we could not expect to achieve compression for textures stored in texture memory. The main memory cache and the local disk cache, on the other hand, are more readily accessible and can be used to hold compressed textures.

We experimented with two types of compression, JPEG and Vector Quantization (VQ) [1]. Figure 10 gives CPU usage for decompression of JPEG and VQ texture tiles. For the single processor systems that we were using, decompression complexity turned out to be a big issue. JPEG decoding is quite CPU intensive, and produces a visible stutter in frame rate each time a texture tile is expanded. To lessen this effect, the Meta-texture server decompresses JPEG textures when they are put into the disk cache. Vector Quantization, on the other hand, is particularly efficient at decompression, so the system keeps these textures compressed in the main memory cache.

	RE2		O2	
	gray	color	gray	color
PPM	0.5	6.2	0.6	7.2
JPEG	17.2	32.2	19.6	37.5
VQ	0.6	1.8	0.7	2.2

Figure 10: CPU usage for decompression of  $256 \times 256$  images on the RE2 and O2. Times are in milliseconds. JPEG measurements are based on a library from the Independent JPEG Group.

## 5 Results

We tested our sample application running a Texture Server and Meta-texture Server on two platforms, an SGI Reality Station (RE2), and an SGI O2. The RE2 was equipped with 16 MB of texture memory and a single R10000 processor. The O2 also had a single R10000 processor, and used 16 MB of RAM as texture memory.<sup>4</sup> The main memory cache for both systems was constrained to 100 MB. Frame rate and MSE trials were calculated for a 1383 frame flyby of Park City, Utah. The original texture image for the dataset is  $6121 \times 7651$  pixels in size. The quadtree MIP map contains  $965 \times 256 \times 256$  texture tiles. Disk usage for the texture tiles exceeds 60 MB.

**Trial 1—Frame rate.** To get a feel for how much frame rate would degrade while running the Texture Server, we ran the Park City animation twice, using PGM versions of the texture files.<sup>5</sup> On the first pass, we started the animation with only the root texture loaded, and allowed normal texture loading. On the second pass, however, we killed the Texture Server to prevent texture loading during the animation. By comparing the frame rates for the two passes, we were able to measure the frame rate degradation caused by running the Texture Server. Frame rate results for the Park City flyby are given in Figure 11. As evidenced by the figure, the Texture Server has only a minimal impact on a program's interactivity.

<sup>4</sup>The O2 uses a unified memory architecture, and has no dedicated texture memory.

<sup>5</sup>We did not use the Meta-texture Server for this timing trial. This reflects actual practice since the Meta-texture Server should not be used unless texture access characteristics become strongly bimodal.

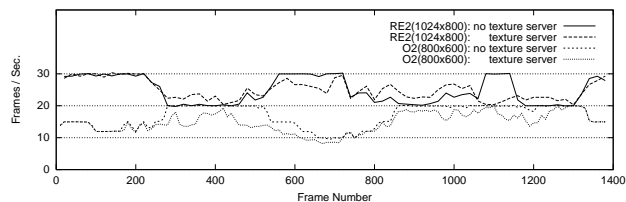


Figure 11: Frame rates for RE2 and O2 during Park City flyby showing the frame rate degradation caused by the Texture Server. The important thing here is not the actual frame rate achieved, but the drop in frame rate caused by running the Texture Server.

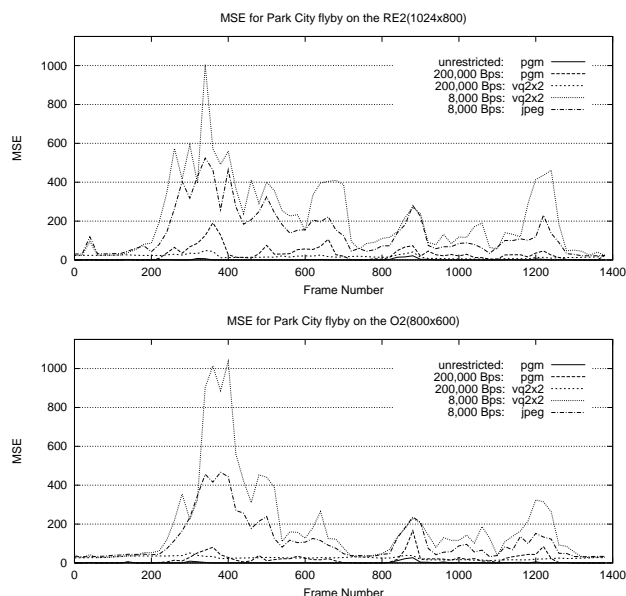


Figure 12: Mean squared error for frames extracted from the Park City flyby. Compression for vector quantized tiles was 3.8:1. Average compression for JPEG was 7.4:1. Bandwidth limits correspond roughly to a T1 line and an ISDN line.

**Trial 2—Mean Squared Error.** An important indicator of the quality of our system is an error metric between desired frames and actual frames from the same camera path under differing disk bandwidths and compression schemes. For comparison purposes we used the mean squared error per color component, not considering untextured background pixels. MSE results are given in Figure 12. Figure 13 demonstrates the image quality degradation that occurs when disk bandwidth is constrained to simulate a slow network.

The largest practical dataset that we have made to date covers a  $51 \times 98$  km region of the Wasatch Front in Northern Utah. Figure 14 shows screen captures from this scene. The quadtree MIP map for the scene contains  $60,309 \times 256 \times 256$  image tiles. Total disk usage for the PGM tiles is 3.68 gigabytes. Since the grayscale texel size for the RE2 is 2 bytes, its 16 MB texture cache can hold at most 128 of the texture tiles. The texture quadtree for the scene is 471 times larger than texture memory capacity, and would not be displayable at all using traditional texture mapping methods.

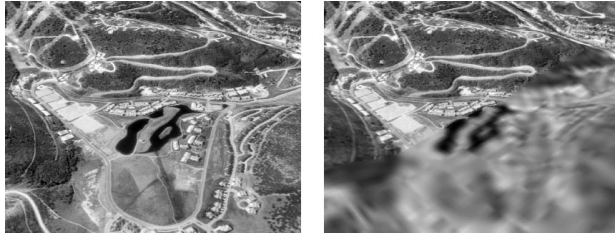


Figure 13: A frame from the Park City flyby under differing bandwidths. Unrestricted disk bandwidth with PGM textures (left). Bandwidth restricted to 8000 bytes/sec with JPEG textures (right).

## 6 Future Work

There are a number of open issues with respect to texture caching that we have not explored.

Our implementation does not provide trilinear MIP-mapping capability. Future implementations may provide trilinear filtering by defining a standard MIP map for each texture tile.

Image compression remains an area for further research. Thus far we have only considered two image compression schemes, JPEG and VQ. Other compression algorithms such as GIF or CCC encoding [5] may prove useful. We have also not considered progressive image encoding algorithms, which may be able to leverage the information of parent texture tiles to build their children.

One thing we have not discussed is how to quickly map the quadtree of textures onto arbitrary polygonal topologies. When a polygon spans more than one texture tile, it must be split. If real-time performance is to be maintained for arbitrary topologies, the modeler cannot simply dice polygons so that each piece covers a single leaf texture.

Prediction is another area in which additional work could be done. By using appropriate prediction, the bandwidth available for texture flow could be more efficiently utilized, increasing the quality of rendered images.

Finally, we have not thoroughly explored caching and priority algorithms. There may be more effective criteria than the ones we are currently using to determine which texture tiles to fetch and which to discard.

## 7 Conclusion

We have developed a texture caching system suitable for interactive display of very large textures. The new approach eliminates the performance bottlenecks associated with large textures by treating texture as a bandwidth-limited resource, allowing an application to maintain real-time performance while rendering textures nearly five hundred times larger than the hardware texture cache. For some datasets, the textures that the system can display in real time are larger than can be feasibly displayed at all using traditional texture mapping methods. The approach is completely software based and can be impemented on existing low-end graphics workstations.

## Acknowledgements

Thanks to Kirk Duffin for his work in building the terrain datasets. Thanks also to the USGS for providing the data. This work was funded in part by a grant from the Utah State Center of Excellence.

## References

- [1] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from Compressed Textures. In *SIGGRAPH 96 Conference Proceedings*, pages 373–378, 1996.
- [2] Jonathan Blow. Implementing a Texture Caching System. *Game Developer*, pages 46–56, April 1998.
- [3] Jonathan Blow. A Texture Cache. In *Proceedings of the 1998 Computer Game Developers Conference*. Bolt Action Software, 1998.
- [4] Richard Bukowski and Carlo Sequin. Interactive Simulation of Fire in Virtual Building Environments. In *SIGGRAPH 97 Conference Proceedings*, pages 35–44, 1997.
- [5] Graham Campbell, Thomas A. DeFanti, Stephen A. Joyce, Lawrence A. Leske, John A. Lindberg, and Daniel J. Sandin. Two Bit/Pixel Full Color Encoding. In *Computer Graphics (SIGGRAPH 86 Conference)*, volume 20, pages 215–219, 1986.
- [6] Shenchang Eric Chen. Quicktime VR – An Image-Based Approach to Virtual Environment Navigation. In *SIGGRAPH 95 Conference Proceedings*, pages 29–38, 1995.
- [7] David Cline. Interactive Display of Very Large Textures. *Master's Thesis, Brigham Young University, Provo UT 84602*, 1998.
- [8] Daniel Cohen-Or, Eran Rich, Uli Lerner, and Victor Shenkar. A Real-Time Photo-Realistic Visual Flythrough. *IEEE Transactions on Visualization and Computer Graphics*, pages 255–265, 1996.
- [9] F. S. Hill, Sheldon Walker Jr., and Fuwen Gao. Interactive Image Query System using Progressive Transmission. In *Computer Graphics (SIGGRAPH 83 Conference)*, volume 17, pages 323–330, 1983.
- [10] Marc Levoy and Pat Hanrahan. Light Field Rendering. In *SIGGRAPH 96 Conference Proceedings*, pages 31–42, 1996.
- [11] Peter Lindstrom, David Koller, Larry F. Hodges, William Ribarsky, Nick Faust, and Gregory Turner. Level-of-Detail Management for Real-Time Rendering of Phototextured Terrain. <file://ftp.gvu.gatech.edu/pub/gvu/tr/95-06.ps.z>. *GIT-GVU Technical Report*, 95-06, 1995.
- [12] Patricia McLendon. *Graphics Library Programming Guide*. Number 007-1702-010. Silicon Graphics, 1992.
- [13] Jacke Neider and CS Tillman. *ImageVision Library Programming Guide*. Number 007-1387-020. Silicon Graphics, 1992.
- [14] Shaun M. Oborn. UTAH: The Movie. *Master's Thesis, Utah State University, Logan UT*, 1994.
- [15] Randy Pausch, Jon Snoddy, Robert Taylor, Scott Watson, and Eric Halesline. Disney's Aladdin: First Steps Towards Story Telling in Virtual Reality. In *SIGGRAPH 96 Conference Proceedings*, pages 193–203, 1996.
- [16] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *SIGGRAPH 97 Conference Proceedings*, pages 101–108, 1997.
- [17] Montrym John S., Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A Real-Time Graphics System. In *SIGGRAPH 97 Conference Proceedings*, pages 293–301, 1997.
- [18] H. Samet. The Quadtree and Related Hierarchical Data Structures. In *ACM Comp. Surveys*, volume 16, pages 178–260, 1984.
- [19] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *SIGGRAPH 96 Conference Proceedings*, pages 75–82, 1996.
- [20] Norbert Strobel, Sanjit K. Mitra, and B. S. Manjunath. An Approach to Efficient Storage, Retrieval, and Browsing of Large Scale Image Databases. *Proceedings of the SPIE*, 2606:324–335, 1995.
- [21] Jay Torborg and James T. Kajiya. Talisman: Comodity Realtime 3D Graphics for the PC. In *SIGGRAPH 96 Conference Proceedings*, pages 353–363, 1996.
- [22] Lance Williams. Pyramidal Parametrics. In *Computer Graphics (SIGGRAPH 83 Conference)*, volume 17, pages 1–10, 1983.

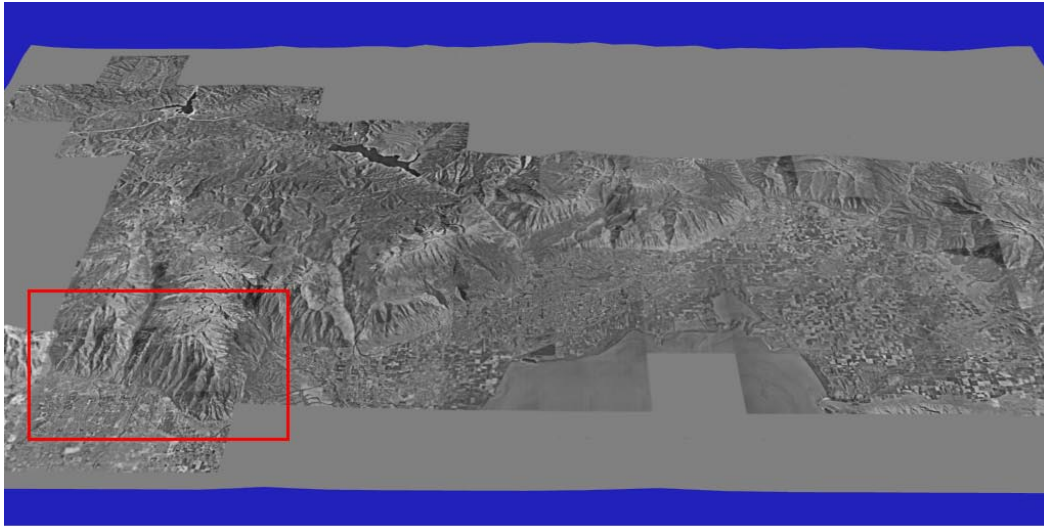


Figure 14: Frames from a flythrough of the Wasatch Front dataset. The images were captured at a resolution of  $1024 \times 512$  pixels. Rectangles show the approximate framing of successive viewpoints.