Multi-Level Texture Caching for 3D Graphics Hardware

Michael Cox,[*] [†] Narendra Bhandari,[†] Michael Shantz[†]

[*]MRJ/NASA Ames Research Center
Moffett Field, CA 94035

[†]Intel Microcomputer Research Labs
2200 Mission College Blvd.
Santa Clara, CA 95052

Abstract

Traditional graphics hardware architectures implement what we call the *push architecture* for texture mapping. Local memory is dedicated to the accelerator for fast local retrieval of texture during rasterization, and the application is responsible for managing this memory. The push architecture has a bandwidth advantage, but disadvantages of limited texture capacity, escalation of accelerator memory requirements (and therefore cost), and poor memory utilization. The push architecture also requires the programmer to solve the bin-packing problem of managing accelerator memory each frame. More recently graphics hardware on PC-class machines has moved to an implementation of what we call the *pull architecture*. Texture is stored in system memory and downloaded by the accelerator as needed. The pull architecture has advantages of texture capacity, stems the escalation of accelerator memory requirements, and has good memory utilization. It also frees the programmer from accelerator texture memory management. However, the pull architecture suffers escalating requirements for bandwidth from main memory to the accelerator. In this paper we propose multi-level texture caching to provide the accelerator with the bandwidth advantages of the push architecture combined with the capacity advantages of the pull architecture. We have studied the feasibility of 2-level caching and found the following: (1) significant re-use of texture between frames; (2) L2 caching requires significantly less memory than the push architecture; (3) L2 caching requires significantly less bandwidth from host memory than the pull architecture; (4) L2 caching enables implementation of smaller L1 caches that would otherwise bandwidth-limit accelerators on the workloads in this paper. Results suggest that an L2 cache achieves the original advantage of the pull architecture – stemming the growth of local texture memory – while at the same time stemming the current explosion in demand for texture bandwidth between host memory and the accelerator.

1 Introduction

Acceleration hardware for real-time 3D graphics has become mainstream. Once available primarily on high-end workstations [1, 17], 3D rasterization hardware has been available (even standard) on the PC platform since about 1995. While the performance and image quality of initial products were below the standard for the workstation market (cf. [14]), intense competition among many Independent Hardware Vendors (IHVs) has driven both performance and quality to within range of (sometimes beyond) workstation 3D graphics.

One feature of greater importance on the PC platform is texture mapping. Texture mapping is a rendering process whereby 2D images are mapped onto polygons as the polygons are rasterized into pixels. Texture mapping allows the graphics programmer to create an illusion of much greater realism in a 3D scene than would otherwise be possible simply with surface primitives such as polygons. It is well known that texture mapping is memory-intensive, both in terms of capacity and bandwidth (cf. [26]).

Workstation graphics hardware has with few exceptions been an implementation of what we call the *push architecture* (Figure 1a). Large fast memories are co-located with the acceleration hardware so that when pixels from the texture (called *texels*) are required, they can be retrieved with low latency and high bandwidth. This was the original architecture adopted by IHVs on the PC platform as well. The push architecture has an advantage that since local memory is dedicated to graphics, the memory subsystem can be designed to provide exactly the bandwidth required. The push architecture has several disadvantages. First, historically for any date *d* graphics programmers want more textures at higher resolution than can be stored by the push architecture on date *d*. Demand for textures and resolution has exceeded capacity even on the high-end InfiniteReality [17] whose common configuration includes 64 MB of local texture memory. Second, when the programmer wants to use more textures than fit in texture memory, he or she (or the software driver) must write what is essentially a segment manager. This is provably a hard problem since textures are of variable size, and is exacerbated since it also includes synchronization between CPU and accelerator. Third, the push architecture in general forces the programmer (or driver software) to download entire textures to the accelerator,[1] an inefficient use of valuable memory since generally only a fraction of the texels is required.

To stem an apparently unbounded escalation in the size of texture memory dedicated to the accelerator, Intel Corporation proposed and promoted the adoption of what we call the *pull architecture* (Figure 1b). Textures are stored in system memory, and the accelerator pulls texels from system memory to accelerator on-chip memory as required. There is no local external memory for texture. To provide the I/O bandwidth required for texturing from system memory, Intel defined and has implemented the Accelerated Graphics Port (AGP) between core logic and the accelerator [16]. Version 1.0 of this specification provides for up to 512 MB/s of bandwidth dedicated to graphics. The pull architecture has several advantages. First, it places texture in system memory, relaxing capacity constraints and thereby curbing the growth of local memory associated with the accelerator. Second, it has freed the programmer from what was previously the bin-packing problem of texture memory management. Third, the accelerator only downloads the texels (or blocks of texels) required. The pull architecture also has disadvantages. It forces system memory to be

[1] A clever scheme to “clip” the MIP pyramid to the viewing frustum has been introduced at the high end [17], but this technique saves memory only for extremely large textures (e.g. 32K x 32K), and still has the disadvantage that it requires download of more texture blocks than used.

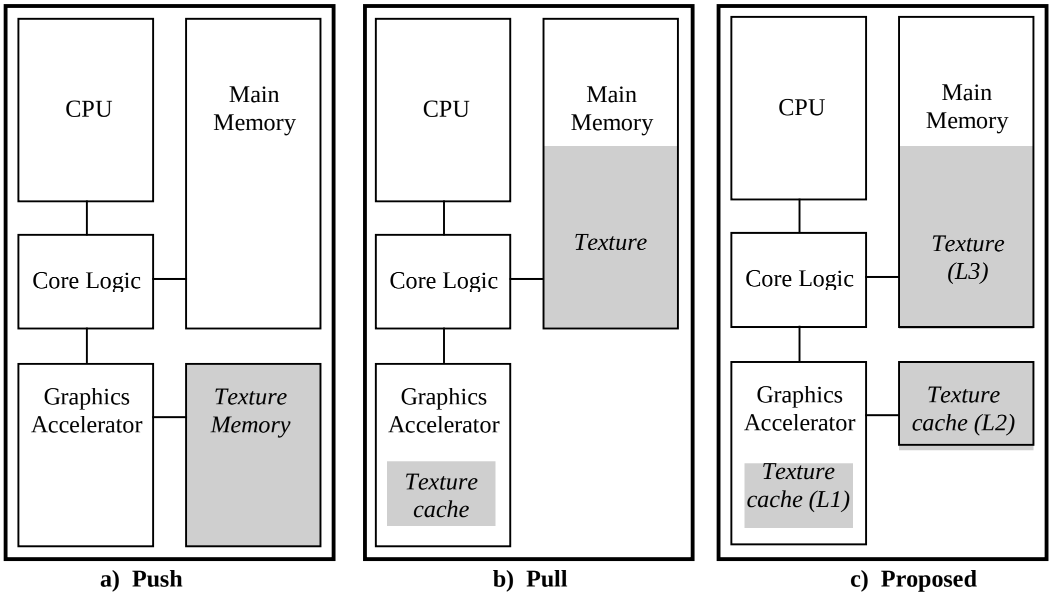

Figure1. Alternative architectures.

designed to support the bandwidth required by texture mapping. It also forces the AGP standard (and implementations) to keep pace with escalating demand for texture bandwidth. Some high-end accelerator parts designed to implement the pull architecture are even today rate-limited by their ability to retrieve texture from system memory [27].

While high-end accelerator boards continue to implement the push architecture, the bulk of IHVs have moved to the pull architecture. This has achieved the desired result of stemming the growth of local frame buffer size, but it has had the effect of moving a growing bandwidth requirement from local memory to core logic and system memory. We would like to allow the unfettered growth of texture capacity while at the same time allowing the unfettered growth of texture mapping performance. This is the goal of multi-level texture caching and the proposed architecture of Figure 1c.

2 Background

2.1 Texture Mapping

Creating a realistic image of a three dimensional scene requires projecting the geometry, which is typically composed of planar polygons, into screen space for display. To give the geometry a realistic appearance, texture maps must be mapped from texture space onto the vertices of the polygons and then perspectively projected correctly into screen space [2, 12, 13, 23]. The resulting projection in screen space is sampled at the pixel locations to generate the final display. The process of sampling polygons in screen space is referred to as scan conversion, or rasterization. The scan conversion process iteratively steps across the polygon's pixels in screen space <*x,y*>, finds the corresponding points in texture space <*u,v*>, then accesses the texture to obtain the texels for use in coloring the pixels. When the viewer is distant from the surface (*minification*), adjacent pixels may map to corresponding texels that are far apart in texture space. When the viewer is close to the surface (*magnification*), many pixel steps may be required to move from one texel to the next in texture space. Such variations in step size in texture space lead to poor locality of reference and also cause aliasing unless properly filtered textures are used [4, 5]. Texture mip maps, properly filtered and interpolated provide a well-known solution to both problems. With mip mapping, the texture is stored at many resolutions called MIP levels [31]. Each level is a one-quarter filtered image of the lower MIP level. During scan conversion the texture space step size is used to select an appropriate MIP level which will yield an approximately 1:1 mapping of texels to pixels. The ratio of texels to pixels is sometimes referred to as the *texture compression*, and is used directly to select the proper MIP level (cf. [28]). This process gives antialiased texture mapping and also has good locality of reference, but the transitions between MIP levels within large polygons are objectionable and require interpolation. Bilinear and trilinear interpolation are the standard ways to mitigate these problems (cf. [10, 28]).

2.2 Multi-Level Texture Tiling

Sequential texture memory accesses tend to be local spatially in <*u,v*> within a MIP level, and also local to a MIP level. It is well known in computer graphics that advantage can be taken of the first by storing texture images in tiles rather than linearly by <*u,v*> (cf. [3]). The second can be exploited by storing MIP levels separately and addressing texture by a new tuple <*u,v,m*> (where *m* is the MIP level). In the current paper we take advantage of separate MIP level storage as well as texture storage in tiles of tiles. These both fit within a framework that might be called *hierarchical texture storage*. For each texture used by an application in a graphics system, there is today generally assigned a unique identifier, *tid*. In many systems today, each texture is generally further partitioned into tiles (or blocks), each of which is uniquely identified within the texture by an integer (which we call *L2* for later convenience). Each of these tiles may be further partitioned into tiles (or blocks) each of which may be uniquely identified within its parent L2 block by an integer (say, *L1*). The concatenation of <*tid, L2, L1*> can then be used to identify a unique tile within a tile among all the textures employed by the application. This addressing lends itself well to 2-level texture caching, and is shown pictorially in Figure 2. In the figure, each MIP level is represented as a smaller square below and to the left of its higher-resolution parent. Within a texture, L2 block numbers are assigned sequentially from the first block of the lowest MIP level to the last block of the highest MIP level. Each new level of the MIP begins with a unique L2 block. Within an L2 block the L1 sub-blocks are assigned unique numbers only within the scope of that block. Translation from <*u,v,m*> to this tiled representation is

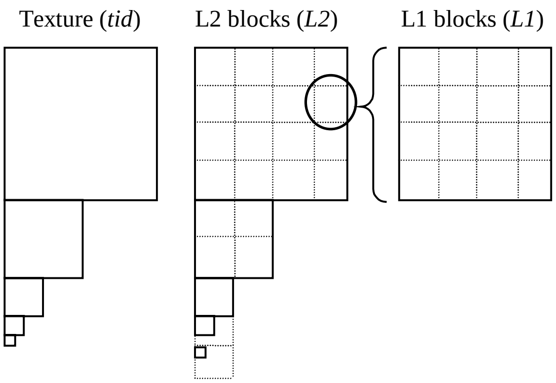

Figure 2. Hierarchical texture addressing for L2 caching. Representation of virtual texture address <*tid, L2, L1*>.

straightforward and can be done in integer arithmetic in a small number of shifts, additions, and a table look-up.

2.3 Level 1 Texture Caching

Texture caching has been proposed, studied, and previously implemented [11, 20, 26, 30]. In fact, IHVs designing for AGP texture access have no (apparent) choice but to cache textures on chip because of the high latencies of the pull architecture. Previous texture caching strategies have implicitly been Level 1 (L1) caching strategies. One previous proposal has been for disk caching of texture tiles [20]. Hakura's study of L1 texture caching provides a nice taxonomy of general issues that must be addressed [11], in particular the effects of storage format (tiled vs. linear), tile and cache-line size, cache addressing and associativity, cache size, and rasterization order. We discuss these in turn.

Talisman proposes texture tiles 8 texels by 8 texels in size (8x8) [26], Winner employs 2x2 tiles [30], and the authors are aware of commercial architectures that employ 4x4 tiles. Hakura studies the difference in hit rate and bandwidth requirements between 4x4 and 8x8 and finds that although 8x8 tiles lead to better cache hit rate, they require more download bandwidth (the larger the tile the lower the utilization). Consistent with our goal to reduce bandwidth to system memory and in order to study L2 parameters with respect to a fixed L1 cache implementation, we have restricted our attention to 4x4 L1 tiles of 32-bit texels.

Consistent with processor cache design [22], Hakura points out that the L1 cache line size need not be the same as the L1 tile size. His results show that miss rates are always higher when the line size is less than the tile size, but that for 2x2 and 4x4 tiles[2] miss rates are reduced when the line size is greater. This suggests that when one tile is downloaded, it is efficacious to download its neighbors as well. However, his results also suggest that while miss rates drop, bandwidth increases when this technique is employed. We have also restricted our attention to L1 cache lines the same size as L1 tiles.

The associativity of the Talisman L1 texture cache is not discussed in [26]. Winner describes an interesting L1 cache that takes advantage of traversal order in texture space to choose blocks for replacement, although the implementation appears to be a modified fully associative cache. Hakura studies fully, set-associative, and direct-mapped caches, and argues that 2-way set associative is of sufficient associativity to avoid conflict misses with trilinear interpolation. We follow Hakura's lead with respect to associativity, and study multi-level texture caching with a 2-way set-associative L1 cache.

Talisman proposes a 4 KB cache, Winner does not report cache size, and Hakura studies cache sizes from 4 KB to 128 KB. The latter's results suggest that in fact a 2-way set associative cache of 16 KB achieves results almost as good as a 32 KB cache of the same associativity. We follow this lead and study L1 cache sizes of 16 KB. A 16 KB cache would require about 400K gates, most of the 3D gate budget of an accelerator today for the volume market. While 16 KB will clearly be feasible in the future, a 2 KB cache is comfortably in range even at the low-end in the immediate future. We have chosen to restrict our attention to these two cache sizes – one at the "low end" (2 KB) and one at the "high end" (16 KB).

Finally, Hakura studies the effect on texture locality of rasterization by tiles, rather than in linear scanline order as is more often done. While his results do show tiled rasterization results in better texture locality, it is not always cost-effective to rasterize this way. In particular, for smaller or skinnier triangles, tiled rasterization leads to lower hardware utilization. For this reason we study multi-level texture caching assuming that primitives are rasterized in scanline order (cf. [10]).

3 Work Loads and Methodology

We have employed the *Intel Scene Manager (ISM)* [24], a scene management and rendering package that we use internally at Intel to explore new graphics algorithms and alternative hardware architectures. This package reads and manages scene databases, provides object-space visibility culling, geometry processing, and rasterization. We have instrumented ISM in order to study the texture access patterns underlying L2 texture caching (section 4.2), and have also integrated into it a transaction-accurate hardware simulator for L2 caching (section 5.3). Results reported in this paper have been measured with a screen resolution of 1024x768.

3.1 Work Loads

We have employed two workloads in the current study: the *Village* and the *City*. The Village is a database courtesy of Evans and Sutherland Corporation. The City is a database developed at UCLA. We have employed scripted animations for both: a walk-through of the Village and a fly-through of the City. Selected images from the animations are shown in Figure 12 at the end of this paper.

3.2 Statistics

To study texture access patterns we have instrumented ISM by adding calls to our own tracing library from appropriate code sites. This tracing library calculates the virtual texture address <*tid, L2, L1*> as described in section 2.2 and tracks all pixel references during each frame. We have gathered statistics for L1 tile sizes of 4x4and 8x8, and L2 sizes of 8x8, 16x16 and 32x32 texels. We have assumed that textures are stored in main memory in their original depth but are expanded to 32 bits by the accelerator for cache storage. We have also assumed that the push architecture can download and store in local memory textures in their original depth. All texture accesses have been measured with point-sampling in order to provide a picture of basic texture locality in the absence of more advanced filtering. The statistics library records textures as they are loaded and deleted, and the L2 blocks and L1 sub-blocks accessed each frame (total and those not accessed the previous frame).

3.3 Simulation

To study the L2 caching algorithm of section 5.2 we have implemented a transaction-accurate (but not cycle-accurate) simulator. This simulator implements L2 caching above a 2-way set-associative L1 cache. L1 tags are the "same" <*tid, L2, L1*> used for L2 virtual addresses. This implicitly implements the "6D blocked representation" for collision avoidance suggested by Hakura [11]. However, the calculation of L2 and L1 require a choice of tile sizes. In a real implementation, these would likely be chosen to match the sizes chosen for L2 cache. However, in

[2] Their results do not address the latter case for 8x8 tiles.

simulation this leads to different L1 cache behavior for different choices of L2 cache parameters. We have instead chosen to implement an L1 cache with a fixed tag calculation across all L2 tile sizes, in particular with L2 tiles 16x16. Finally, the animations produced by the simulator match those produced by the original rendering package.

4 Locality and Working Sets

We distinguish 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 (e.g. as within a tessellated sphere). (3) *Intra-frame locality*. Objects within a frame may share textures (e.g. street pavement, sky, bricks in a building), 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 during one frame are likely used during the next.

L1 texture caching is designed primarily for the intra-triangle working set, but can be expected to absorb some of the intra-object working set as well. The goal of L2 texture caching is to absorb L1 misses when the intra-triangle and intra-object working set exceeds L1 cache size, and to absorb the inter-object and inter-frame working set.

4.1 Expected Inter-Frame Working Set

In this section we derive an expression for expected inter-frame working set. We begin with screen resolution R. During any given frame, we may define the *depth complexity* d as the average number of pixels that are rendered for each pixel location. That is, the number of pixels N_{pix} generated during rasterization is $N_{pix} = R * d$. In general the accelerator attempts to choose a correct MIP level so that texture compression is 1:1, that is $N_{tex} = N_{pix}$. When the accelerator downloads blocks of tiled textures, it must download at least B_{min} blocks, where $B_{min} = N_{tex}$ / *(texels/block)*. However, not every texel in every block is generally used (internal fragmentation), nor is every texel used only once (repeated textures and re-use between objects). The actual number of blocks used B is a function of block *utilization*, that is $B = B_{min}$ / *utilization*. For 32-bit texels, the working set W is then B * *texels/block*, or in bytes $W = (R * d * 4)$ / *utilization*.

W is plotted in Figure 3. As can be seen, for very low utilization the inter-frame working set should require significant memory. However, for utilization at or above 25% the inter-frame working set should be under 64 MB at reasonable depth and resolution. Today's higher-end PC NT workstation designs are already configured with 64 MB of texture memory. When utilization is at or above 50% and depth is 1, the inter-frame working set should be under 16 MB. This requirement is in range of today's lower-end PC NT workstation designs. At greater depth complexity, inter-frame working set clearly increases, and it may be efficacious to support L2 texture caching by z-buffering before texture block retrieval. This is discussed further in section 6.

Table 1 shows statistics and expected inter-frame working set for the Village and City animations. As can be seen, both databases reuse textures. The Village reuses textures within frames and makes use of repeated textures. The City only repeats textures (not obvious from these statistics is that the City does not substantially reuse textures between objects).

4.2 Measured Inter-Frame Working Sets

The memory requirements of texture mapping each frame are shown in Figure 4. This figure shows the system memory in use for textures (in all architectures), the minimum local accelerator memory required by the push architecture, and the minimum local accelerator memory required by the L2 caching architecture. The push architecture requirements are based on the assumption that textures are replaced in local memory only at frame boundaries, but that the application has a perfect replacement algorithm (i.e. *that it can predict exactly the textures required in the upcoming frame*). The L2 caching architecture requirements are based on the assumption that active blocks used more than once during the frame are not replaced before they are required again.

Note that L2 caching can achieve important local memory savings over the push architecture. L2 caching requires about 3.9 MB (1.5 MB) versus about 12 MB (7.4 MB) required by the push architecture for the Village (City). These correspond to factors of between 3x and 5x savings in local memory when "minimum" statistics are compared. Note also that 16x16 L2 tiles do not require significantly more memory than 8x8 tiles but can provide some savings over memory requirements with 32x32 tiles.

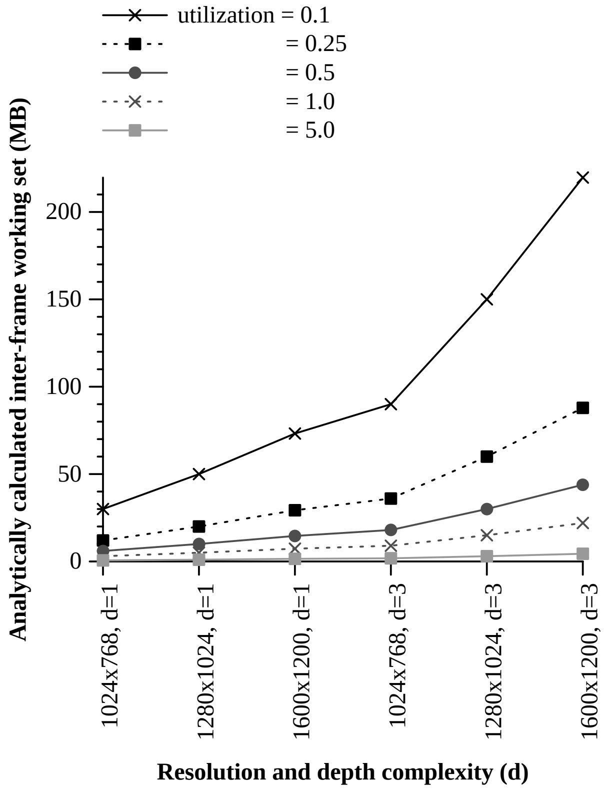

Figure 3. Expected inter-frame working set W as function of resolution R, depth complexity d, and cache block *utilization*.

	Village	City
Depth complexity, *d*	3.8	1.9
Block *utilization*	4.7	7.8
Expected working set, *W*	2.43 MB	.73 MB

Table 1. Statistics and expected inter-frame working set W of the Village and City animations (1024x768 for 16x16 L2 tiles).

Focusing specifically on 16x16 L2 tiles, Figure 5 shows the total versus the new memory required for texture blocks accessed during one frame that were not accessed the previous frame. Total memory is the sum of all active blocks multiplied by block size. New memory is the sum of new blocks accessed multiplied by block size. Note that the inter-frame working set changes only slowly for both the Village and City animations. On average only about 150 KB (40 KB) of required textures are new each frame in the Village (City).

Finally, Figure 6 shows the potential savings in texture download bandwidth that L2 caching offers. This figure shows the minimum total bandwidth required to download tiles to L1 cache, and also the minimum bandwidth required specifically to download L1 tiles that were not used in the previous frame. These numbers are conservative in that they only count once each L1 tile required during the frame. The total download bandwidth is the minimum bandwidth required by the pull architecture in the absence of L2 caching. The bandwidth to download only new L1 tiles is the minimum bandwidth required with L2 caching. Clearly L2 caching offers the potential for extremely significant savings in host memory and AGP bandwidth. Averaged over all frames, 2 MB (510 KB) of L1 tiles are hit each frame in the Village (City), while only 110 KB (23 KB) of these are new.

5 Level 2 Texture Caching

5.1 L2 Cache Organization

The problem of collisions is paramount in texture caching. Hakura has investigated the issue of collisions in L1 texture caching [11], but because L1 size is limited the expected collisions are between blocks within a single texture. L2 texture caching introduces collisions *between* textures. Consider three common cache organizations: direct-mapped, set-associative, and fully associative. With potential collisions between textures, the problem of designing a direct-mapped L2 cache is that of finding a hashing function that uniform-randomly distributes virtual blocks <*tid, L2, L1*> over a much smaller space of physical blocks. This is quite different from the problem of finding a hashing function for L1 caching, where intra-triangle locality is virtually assured by the proper choice of MIP level. Pixels are rasterized in some order within a triangle, and in general except for very large triangles there is strong pixel-to-pixel spatial coherence in texture space that can

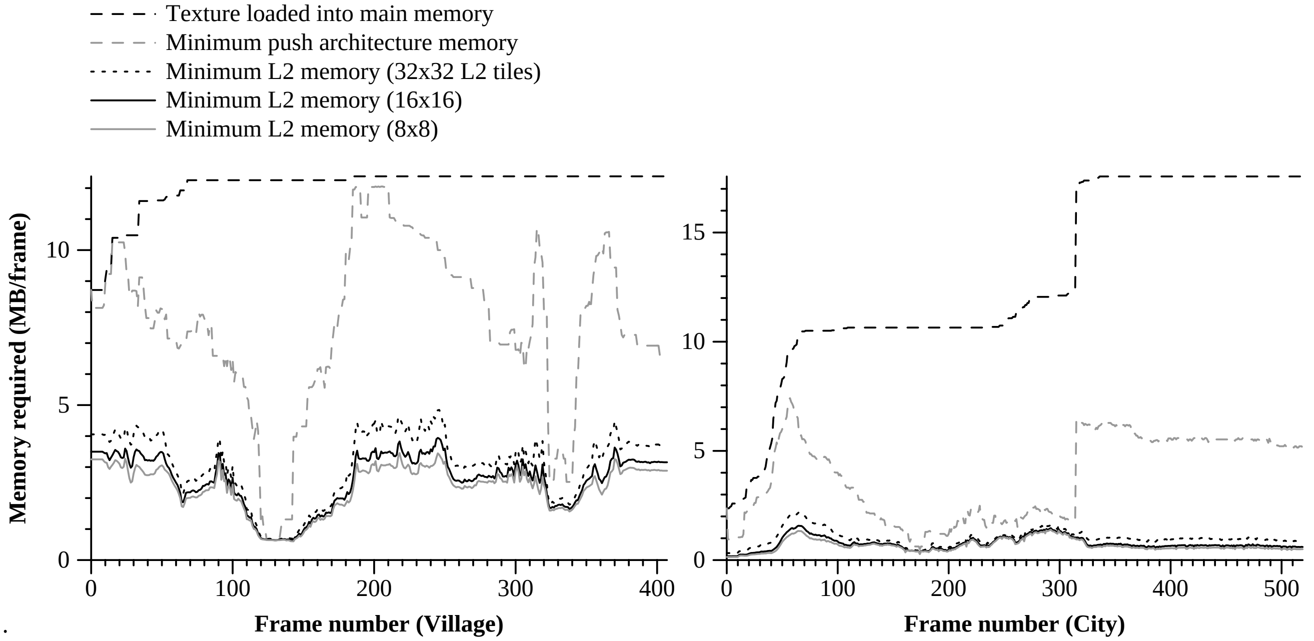

Figure 4. Minimum memory required for all textures, by the push architecture, and in an L2 cache of tiles 32x32, 16x16, and 8x8.

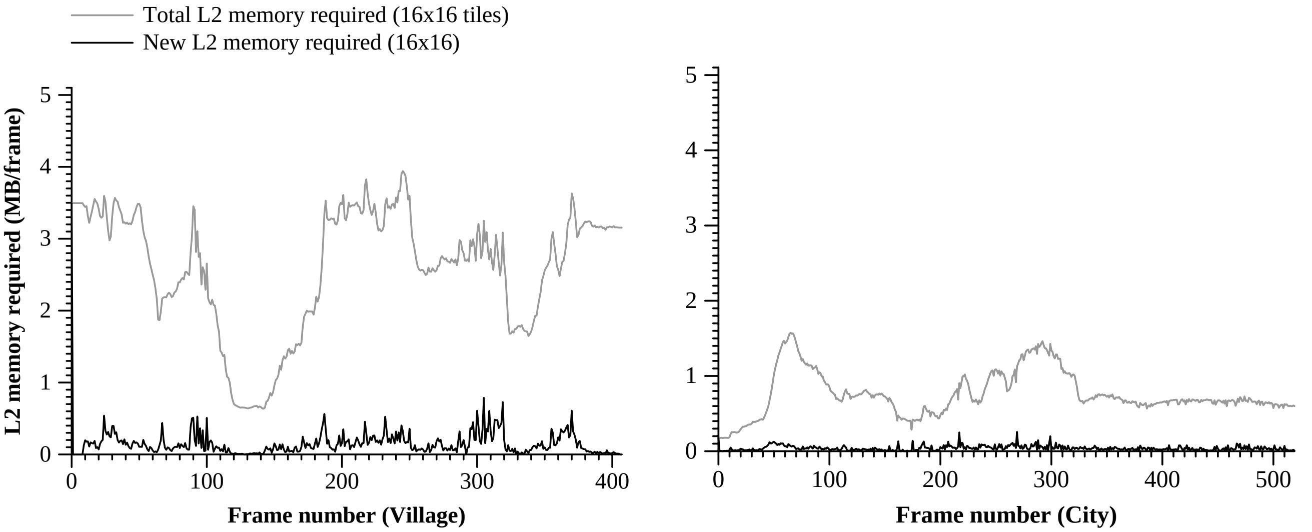

Figure 5. Total and new memory required in an L2 cache of 16x16 tiles.

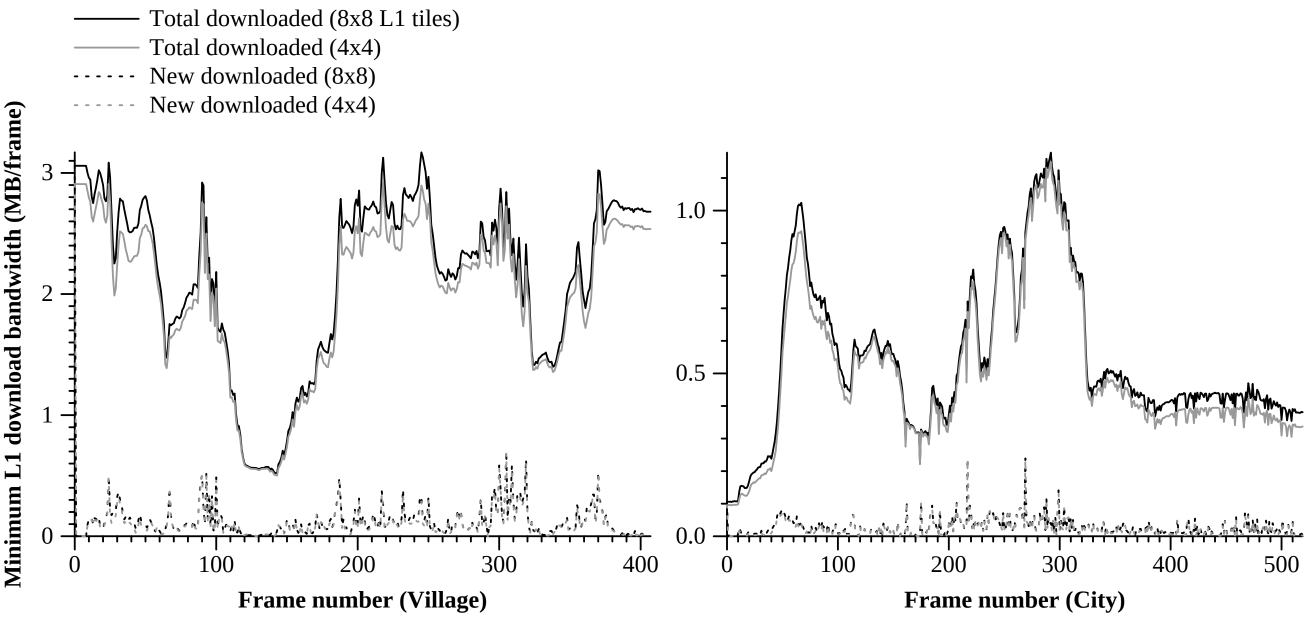

Figure 6. Minimum L1 bandwidth required based on L1 blocks hit at least once. Shown are curves for 8x8 and 4x4 L1 tiles.

be exploited to avoid collisions. Even in this simple case, 2- or 4-way set-associative L1 caches are preferred because of collisions *between* MIP levels. The goal of L2 caching is to accommodate three additional working sets: intra-object, intra-frame, and inter-frame. Intra-object working sets introduce the complication (especially for large textures) that collisions between disparate blocks within the textures must also be avoided. This is because the triangles that result from tessellation of arbitrary surfaces may be delivered to the rasterization engine in arbitrary order (e.g. consider the triangle-strip tessellation of a sphere). Intra- and inter-frame working sets introduce the complication that collisions between blocks of different textures must be avoided. This in turn introduces an implicit requirement that the hashing function be a function not only of spatial but also of temporal locality. Similarly, organizing an L2 cache set-associatively would mitigate rather than solve the problem of finding an acceptable hashing function. Both direct-mapped and set-associative caches are made more difficult by the trend in graphics hardware to support simultaneous mapping from multiple textures. Collisions can be viewed as a block replacement problem that results from restricting the placement of an L2 texture tile to a restricted set of physical locations within the L2 cache [18]. Such restriction is inherent in set-associative (and direct-mapped) caches. While it may be possible to find a hashing function that leads to good replacement behavior in a direct-mapped or set-associative L2 texture cache, we have not pursued this direction. The only apparent alternative is to design a fully associative L2 cache. The working set results of Section 4.2 suggest that an L2 cache should be on the order of MB rather than KB. A fully associative cache of this size is clearly beyond feasibility on the PC platform in the forseeable future.

Alternatively we have chosen to treat the problem of L2 texture caching as a problem of virtual memory (cf. [15, 18, 25]), in a cache organization similar in spirit to that used on the MU5 (where on-processor addresses were virtual, with translation to real addresses via "current page registers" in a Store Access Control Unit [19]). We may view each L2 tag *<tid, L2, L1>* as the address of a virtual texture block that must be mapped to a physical texture block within L2 cache memory. To proceed in this way then, we require a texture page table to translate from virtual texture addresses of the form *<tid, L2, L1>* into physical addresses within L2 cache memory. We also require a replacement algorithm. We a priori expect Least Recently Used (LRU) page replacement to be a good choice for replacing texture pages in an L2 cache. We have chosen to study L2 texture caching with LRU approximated by the "clock" algorithm (cf. [15, 25]). While more recent algorithms for approximating LRU may be less "peaky" in their behavior (cf. [8]), clock is a simple and robust algorithm that is still used in practice.

5.2 Structures and algorithms

Principal data structures and cache control are covered in this section, and shown more precisely in pseudo-code in the Appendix. Control flow in L2 caching is shown in Figure 7. When a texel is required during rasterization, the accelerator first calculates the virtual texture block *<tid, L2, L1>* that holds that texel and

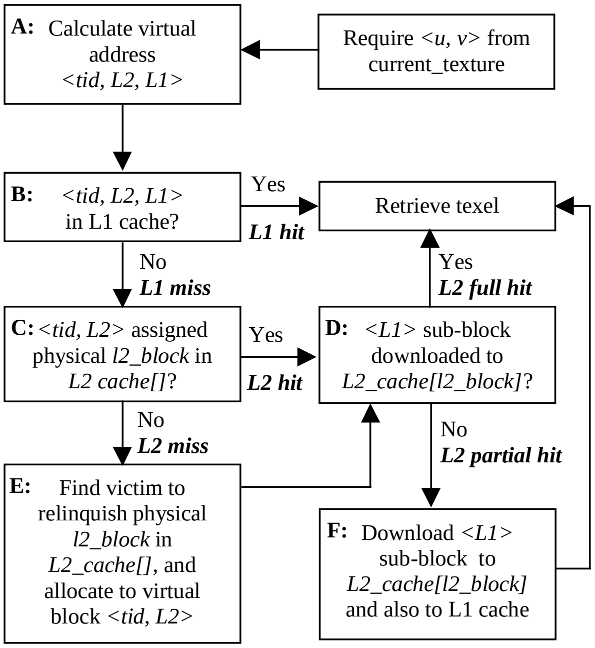
A: Calculate virtual address <tid, L2, L1>
Require <u, v> from current_texture
B: <tid, L2, L1> in L1 cache?
Yes
L1 hit
Retrieve texel
No
L1 miss
Yes
L2 full hit
C: <tid, L2> assigned physical l2_block in L2 cache[]?
Yes
L2 hit
D: <L1> sub-block downloaded to L2_cache[l2_block]?
No
L2 miss
No
L2 partial hit
E: Find victim to relinquish physical l2_block in L2_cache[], and allocate to virtual block <tid, L2>
F: Download <L1> sub-block to L2_cache[l2_block] and also to L1 cache

Figure 7. Control flow in L2 caching.

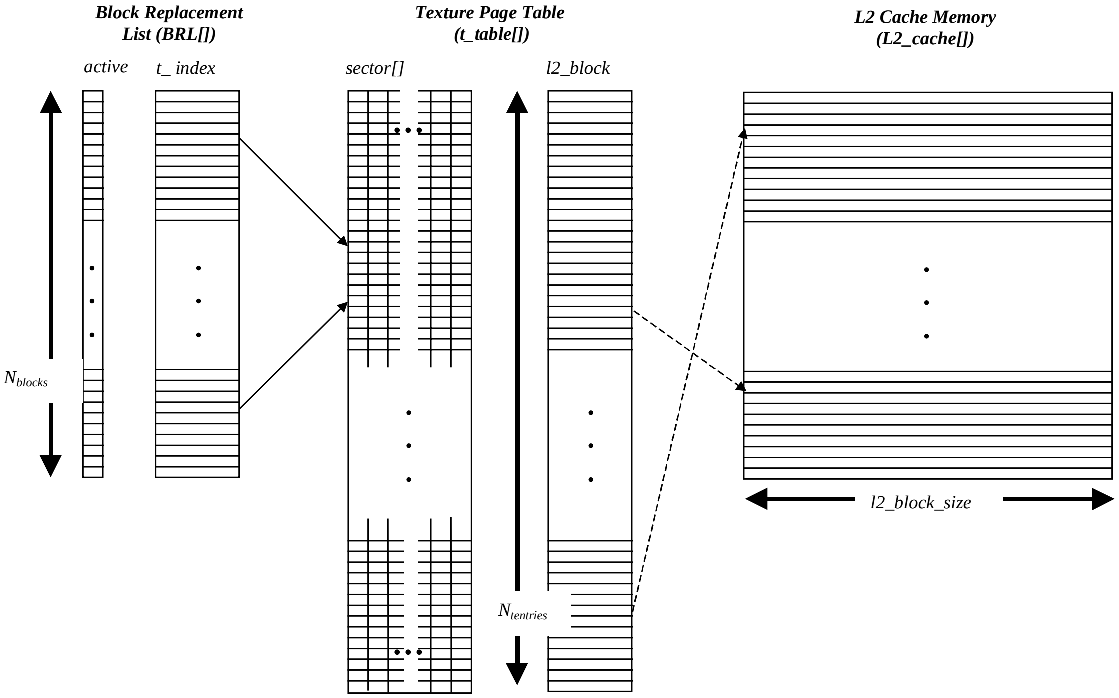
Block Replacement List (BRL[])
Texture Page Table (t_table[])
L2 Cache Memory (L2_cache[])
active
t_ index
sector[]
l2_block
N_{blocks}
$N_{tentries}$
l2_block_size

Figure 8. The primary data structures of L2 caching.

determines if the required L1 block is in L1 cache (steps A and B). If the desired block is in L1 cache the accelerator retrieves the desired texel. If not, the accelerator checks the L2 cache for the L1 block. This is done in two steps because we employ a technique referred to as *sector mapping*.[3] Rather than download each full L2 block, we download only the L1 sub-block required by each L1 miss (leaving vacant the remaining L1 sub-blocks to be downloaded on demand). We do so in order not to exceed the download bandwidth of the pull architecture (which has no L2 cache and downloads L1 blocks directly from L3 to L1). So, continuing at step C, the accelerator first determines if a physical *l2_block* has been allocated to the virtual L2 block <*tid, L2*>. At step E on a full *L2 miss* a physical block has not yet been allocated, and the accelerator finds a victim to relinquish a block currently in use. The accelerator then allocates the found *l2_block* for use by the virtual block <*tid, L2*>. If (or once) a physical block is allocated, the accelerator determines if the desired <*L1*> sub-block has yet been downloaded (step D). If not, the accelerator must download that block to L2 cache (step F). Since L2 cache memory is under the control of the accelerator and downloading from main memory passes through the accelerator itself, we take the opportunity to remove latency by downloading into L1 cache in parallel with the download to L2 cache at step F.

The principle data structures of L2 caching are shown in Figure 8. The *Texture Page Table (t_table[])* provides the means to map from virtual block <*tid, L2, L1*> to a physical L1 block in *L2 Cache Memory (L2_cache[])*. The page table must be sufficiently large to hold an entry for every <*tid, L2*> block that may be active in system memory at once.[4] Allocation of *t_table[]* space is done by host driver software.

Mapping from virtual block <*tid, L2, L1*> is done in two steps. First, an index into *t_table[]* is built from <*tid, L2*>. Second, the entry at that index is checked to determine if the <*L1*> sub-block has yet been downloaded. To build an index into *t_table[]*, we leverage on some standard machinery for managing textures. Even today the host software driver keeps track of textures as the application loads and deletes them, and informs the accelerator whenever the application changes the *current texture*. We require an extension to the accelerator's register set to identify the *current texture's* contiguous entries in the table from *tstart* to *tstart + tlen*. The *t_table[]* is then indexed by *tstart* + <*L2*>. If a physical block has been allocated to the virtual block, *l2_block* is non-zero and is the physical block number in *L2_cache[]*. This corresponds to step C of Figure 7. The *sector[]* field of each *t_table[]* entry holds a bit for each L1 sub-block. If this bit is set the corresponding L1 sub-block has been downloaded (step D).

Finally, replacement in the L2 cache is driven by the *Block Replacement List (BRL[])*. There are the same number of entries in the *BRL[]* as there are blocks in the *L2_cache[]*. The index of a given block in *L2_cache[]* is the same as its index into the replacement list. As already noted we employ the "clock" algorithm to approximate LRU (cf. [15]). The *BRL[]* serves as circular FIFO for "clock". For each entry that corresponds to a physical block in *L2_cache[]*, there is an index *t_index* into the *t_table[]* virtual block to which the physical block has been allocated, and a bit that reflects recent activity (*active*). When a victim is required, the clock algorithm marches around the *BRL[]* to find an entry with no recent activity, at the same time clearing the *active* bits of entries it encounters. During normal operation, whenever a physical *l2_block* is accessed, the accelerator sets the *active* bit corresponding to that block's entry in the *BRL[]*. The accelerator notifies a victim chosen for replacement by using *t_index* to clear the virtual block's ownership (*l2_block*). The accelerator allocates a physical block by setting *t_index* to the virtual block's index in *t_table[]* and by setting that entry's *l2_block* to the physical block's index into *L2_cache[]* and the *BRL[]*.

[3] Sector mapping was employed on the IBM System/360 Model 85 to conserve cache tag comparators in a fully associative cache [15]. Our use of sector mapping is to conserve page table entries.

[4] For example, if the system allows 32 MB of texture in system memory and L2 blocks are 16x16 texels by 32 bits, the *t_table[]* must comprise 32 K-entries.

It remains to describe how texture table entries are deallocated. This can be done easily for the current texture by iterating over the page table entries from *tstart* to *tstart+ tlen*, and for each entry with an allocated block, clearing the appropriate *BRL[]* entry and clearing the page table entry itself. This can be done by the accelerator or by the host if the appropriate data structures are memory-mapped.

5.3 Simulation Results

5.3.1 L1 cache behavior

Figure 9 shows L1 miss rates (by cache size) over the Village animation. These results corroborate Hakura's graphs that show that 16 KB caches result in hit rates almost as good as 32 KB caches [11]. Note also that even for 2 KB caches, peak miss rates do not exceed 4% for bilinear or 5% for trilinear. Average hit rates over all frames are shown in Table 2.

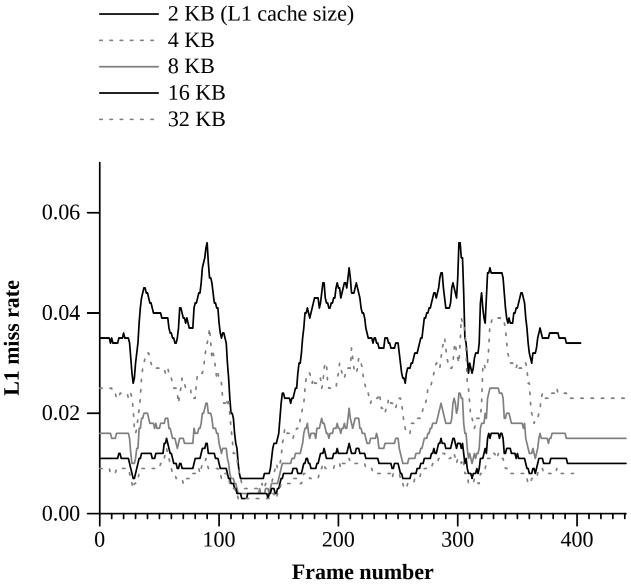

Figure 9. L1 miss rate by cache size (Village).

	L1 size (KB)	BL hit rate (%)	TL hit rate (%)
Village	2	97.16	96.34
	16	99.11	98.94
City	2	97.46	96.72
	16	99.38	99.26

Table 2 . Average L1 hit rates bilinear (BL) and trilinear (TL).

5.3.2 Bandwidth

The download bandwidths required with and without L2 cache are shown in Figure 10. Averages over all frames are shown in Table 3. The figure and table show results for trilinear animations and L2 caches of 16x16 tiles. Similar results were observed for tiles 8x8 and 32x32. These results apply to the pull and L2 caching architectures. Results are not shown for push architecture download bandwidths, as these depend on the specific replacement and packing algorithms employed by the application.

Without L2 caching, it is clear that download bandwidth requirements of the pull architecture are high, even with 16 KB L1 caches. With even a 16 KB L1 cache (but no L2 cache) the Village would require 475 MB/s average download bandwidth at 30 Hz. This exceeds the delivered bandwidth of AGP (and does not even account for peak requirements). With a 2 KB L1 cache (but no L2 cache) the Village would require sustained bandwidth of 1.6 GB/s at 30 Hz. With the same 2 KB L1 cache, even a 2 MB L2 cache would reduce Village download bandwidth requirements to an average 92 MB/s at 30 Hz. Even a 2 MB L2 cache saves the Village animation between 5x and 18x in bandwidth over a vanilla pull architecture (for 16 KB and 2 KB L1 caches, respectively).

These results in turn suggest that not only may an L2 cache be used to stem the escalating demands for main memory and AGP bandwidth, but that an L2 cache can reduce the size of L1 cache required for target performance. Of course, the difference in bandwidth must be absorbed by the local L2 cache memory system, but this is one of the goals of the proposed architecture – to move graphics bandwidth to local memory without the push architecture's escalation in capacity requirements.

Even a 2 MB L2 cache accommodates the inter-frame working

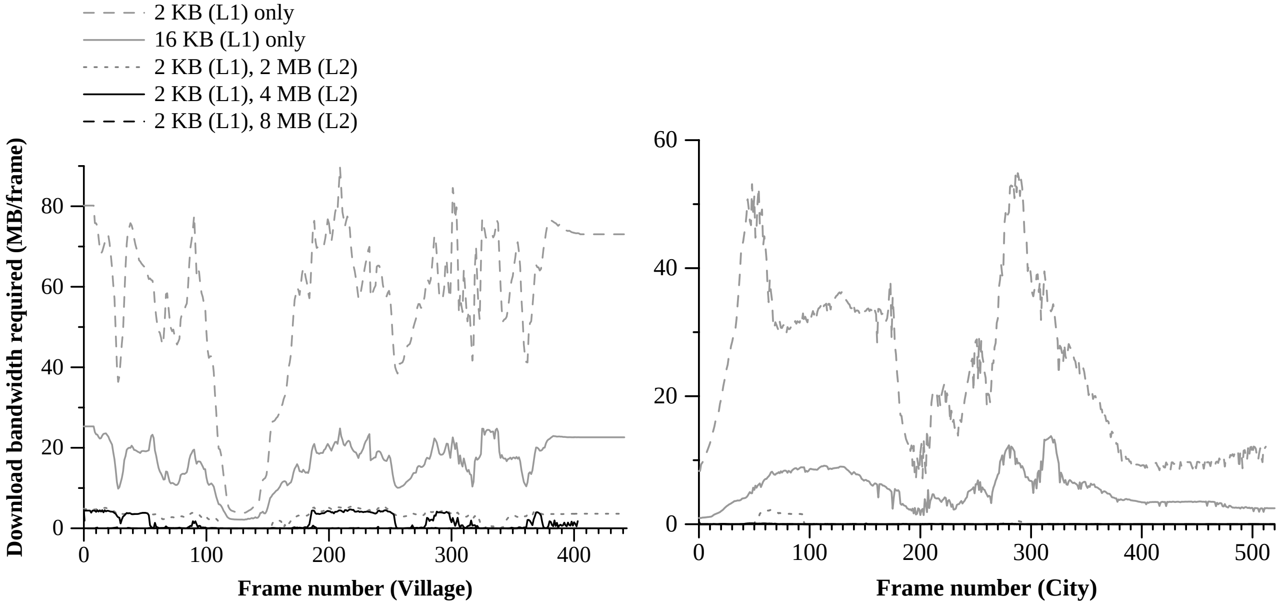

Figure 10. Download bandwidth required with and without L2 cache (2 KB and 16 KB L1 caches with no L2 cache, and 2 KB L1 cache with 2, 4, and 8 MB L2 caches of 16x16 tiles).

set of the City animation over almost all frames (with exception between frames 50 and 100). A 4 MB L2 cache accommodates the inter-frame working set about half of the time, and an 8 MB L2 cache accommodates this working set all of the time for the Village. These numbers may be compared with the maximum of the minimum inter-frame working set measured in section 4.2. The larger memory required in practice arises from approximation by LRU (and approximation of LRU by clock) versus perfect replacement. However, it is clear that even when the inter-frame working set is not always accommodated, L2 caching requires less local memory than the push architecture (which requires *minimum* 8 and 12 MB for the City and Village, respectively). At the same time L2 caching frees the application from texture memory replacement and packing.

L2 size	L1 size	Village (MB/frame)		City (MB/frame)	
(MB)	(KB)	BL	TL	BL	TL
2	2	1.90	3.05	0.01	0.17
4	2	0.09	1.65	0.01	0.01
8	2	0.03	0.04	0.01	0.01
None	2	21.20	54.60	9.20	23.90
None	16	6.70	15.80	2.30	5.40

Table 3. Average AGP and system memory bandwidth required (MB/frame) for the Village and City for bilinear (BL) and trilinear (TL) filtering, with and without L2 cache.

5.4 Implementation and Performance

5.4.1 Implementation

	L2 cache size (MB)		
	2	4	8
Page table size to support: … 16 MB host texture (KB)	64 KB		
… 32 MB host texture (KB)	128 KB		
… 64 MB host texture (KB)	256 KB		
… 256 MB host texture (KB)	1024 KB		
… 1 GB host texture (KB)	4096 KB		
BRL for *active* bits only (KB)	.25 KB	.5 KB	1 KB
BRL sans *active* bits (KB)	8 KB	16 KB	32 KB

Table 4. Memory requirements of L2 caching structures given L2 tiles of 16x16 texels.

The implementation of L2 texture caching is driven primarily by the sizes of the structures required. These in turn depend on the target texture capacity of the platform and the L2 cache and tile sizes. The memory requirements of L2 caching structures are shown in Table 4 assuming L1 tile sizes of 4x4 texels and assuming *t_table[]* and *BRL[]* entries are aligned on 16-bit boundaries. Shown are the KB of data structures required for L2 cache memories of 2, 4, and 8 MB when L2 tiles are 16x16 texels.

It is clear that to support 2+ MB of L2 cache, the cache blocks themselves must reside in external DRAM of some sort. Today's PC graphics subsystems have relatively high-bandwidth channels to local memory (SGRAM, fast SDRAM, or RDRAM) for color- and z-buffers. The continued growth of memory density encourages the use of some of this memory and bandwidth for L2 cache since screen resolution grows more slowly than memory capacity or bandwidth. Higher-end push architectures allocate a separate channel and separate memory for texture. In such architectures that channel/memory may be retargeted for L2 cache.

It is also clear from Table 4 that in the immediate future texture page tables and the *t_index* field of the block replacement list must be stored in external DRAM. We propose that both be stored in the same memory as L2 cache blocks. While the storage of page tables in external memory could significantly increase the latency of access on L1 misses, a page table *Translation Lookaside Buffer (TLB)* can effectively mitigate page table access latency. This is addressed in section 5.4.3. Since the *t_index* field of the *BRL[]* need only be read/written when a victim has been found for replacement, it is not in the critical path of the clock algorithm. Finally, storage for the *active* bits of the *BRL[]* are best left on chip since an *active* bit must be set for every L2 reference. The memory required for *BRL[] active* bits (Table 4) suggest that this may comfortably be done with on-chip SRAM.

5.4.2 Performance

In this section we develop a simple model to compare the expected performance of an L2 caching architecture to the expected performance of the pull architecture. Let the hit rate to L1 cache be h_1, and in the caching architecture the full and partial hit rates[5] to L2 cache be h_{2full} and $h_{2partial}$. Let the time to access a texel be t_1 when the texel is in L1 cache, and in the L2 caching architecture let t_{2full}, $t_{2partial}$, t_{2miss} be the times to access a texel on a full L2 hit, partial L2 hit, and L2 miss respectively. Finally let t_3 be the texel access time in the pull architecture on an L1 miss. Then the average access times A_{pull} and A_{L2} of the pull and L2 caching architectures respectively are given by:

$$A_{pull} = h_1\, t_1 + (1 - h_1)(t_1 + t_3)$$

$$A_{L2} = h_1\, t_1 + (1 - h_1)[t_1 + h_{2full}\, t_{2full} + h_{2partial}\, t_{2partial} + (1 - h_{2full} - h_{2partial})\, t_{2miss}]$$

Let us express $t_{2partial}$, t_{2full}, and t_{2miss} as functions of t_3. It is standard wisdom in graphics hardware design that the accelerator may only take 50% of host memory bandwidth. Ignoring many details, let us assume that the accelerator memory and host memory subsystems are of similar design/bandwidth, so that local accelerator memory is 2x the performance of host memory. We further assume it is possible to pipeline the algorithm in Figure 7 so that $t_{2full} \approx \frac{1}{2}\, t_3$. The main challenge in achieving this goal is in average fast access to the texture page tables. Texture table TLB results are discussed in section 5.4.3.

Continuing the derivation of a simple performance model, we assume that the cost of downloading to L1 and L2 on a partial L2 hit is comparable to the cost of downloading to L1 alone, that is $t_{2partial} \approx t_3$.

The cost of a full L2 miss is more difficult to estimate. On a full page miss, the clock algorithm must search the *active* bits of the *BRL[]* to find a victim and must read-modify-write three locations in external DRAM (the *t_index* of *BRL[]*, the *t_table[]* entry of the victim, and the *t_table[]* entry of the new owner). While it should be possible to pipeline the accesses to external memory with downloading the L1 sub-block from main memory, the clock's search of the *active* bits can be of variable cost. We have studied this cost, and have found that extreme *BRL[]* searches tend to be "peaky" – lasting only a frame or two. We have also found that if the *active* bits were searched 16 at a time (in our workloads for 2- and 4-MB L2 caches), a victim could always be found within 32 cycles. This cost is comparable to the time to download an L1 block from host memory (ignoring latency). However, we leave the cost of a full L2 miss a variable for subsequent exploration, that is $t_{2miss} \approx c\, t_3$.

Combining these results we have that

[5] We report these as L2 rates given that an L1 miss has occurred (i.e. as a conditional probability). We do so in part because unlike with processor multi-level caches, inclusion is not guaranteed (L1 block *A* that is loaded into L1 cache from L2 block *B* may remain in L1 even after *B* has been replaced in L2).

$A_{pull} = h_1 t_1 + (1 - h_1)(t_1 + t_3) = t_1 + (1 - h_1) t_3$

$A_{L2} = h_1 t_1 + (1 - h_1)(t_1 + f\, t_3) = t_1 + (1 - h_1) f t_3$

where $f = c - (c - ½)\, h_{2full} - (c - 1)\, h_{2partial}$

and $c = t_{2miss} / t_3$.

f must clearly be less than 1 for the performance of an L2 caching architecture to match or exceed that of the standard pull architecture. Let us call *f* the *fractional advantage* of the L2 caching architecture (the ratio of the L2 architecture's cost on L1 miss divided by the pull architecture's cost on L1 miss). Using measured L1 hit rates (Tables 5 and 6), and choosing a value for *c*, we can calculate the expected fractional advantage *f* of L2 caching. We assume arbitrarily that a full L2 miss costs no more than 8x the cost of downloading an L1 block to L1 cache (*c = 8*). The fractional advantage is shown in Table 7. As can be seen, even when a full L2 miss is quite expensive, we expect overall performance of the L2 caching architecture to exceed that of the pull architecture.

L2 size (MB)	L1 size (KB)	Village h_{2full} (%)		City h_{2full} (%)	
		BL	TL	BL	TL
2	2	91.03	94.41	99.89	99.31
	16	71.46	80.66	99.54	96.95
4	2	99.56	96.97	99.89	99.94
	16	98.61	89.40	99.56	99.75
8	2	99.85	99.92	99.89	99.94
	16	99.52	99.74	99.56	99.75

Table 5. Average L2 full hit rate h_{2full} (%) for the Village and City for bilinear (BL) and trilinear (TL).

L2 size (MB)	L1 size (KB)	Village $h_{2partial}$ (%)		City $h_{2partial}$ (%)	
		BL	TL	BL	TL
2	2	8.30	5.18	.10	.64
	16	26.42	17.92	.42	2.81
4	2	.41	2.81	.10	.06
	16	1.28	9.82	.41	.23
8	2	.14	.07	.10	.06
	16	.44	.24	.41	.23

Table 6. Average L2 partial hit rate $h_{2partial}$ (%) for the Village and City for bilinear (BL) and trilinear (TL).

L2 size (MB)	L1 size (KB)	Village		City	
		BL	TL	BL	TL
2	2	0.59	0.53	0.50	0.50
	16	0.79	0.59	0.50	0.51
4	2	0.50	0.51	0.50	0.50
	16	0.51	0.55	0.50	0.50
8	2	0.50	0.50	0.50	0.50
	16	0.51	0.50	0.50	0.50

Table 7. Fractional advantage *f* of L2 caching (ratio of the L2 architecture's cost on L1 miss to the pull architecture's cost on L1 miss). The cost of a full L2 miss has been assumed to be bounded by 8x the cost of downloading an L1 block (*c = 8*).

5.4.3 Texture Page Table TLB

Simulated behavior of a *Translation Lookaside Buffer (TLB)* for the *Texture Page Table* is shown in Figure 11. Replacement for multi-entry TLB's was round robin. Shown are the results with trilinear filtering and for 2 KB L1 and 2 MB L2 caches of 16x16 tiles. Average TLB hit rates for the Village and City over 411 and 525 frames (respectively) are shown in Table 8. Results for other L2 cache sizes were essentially identical.

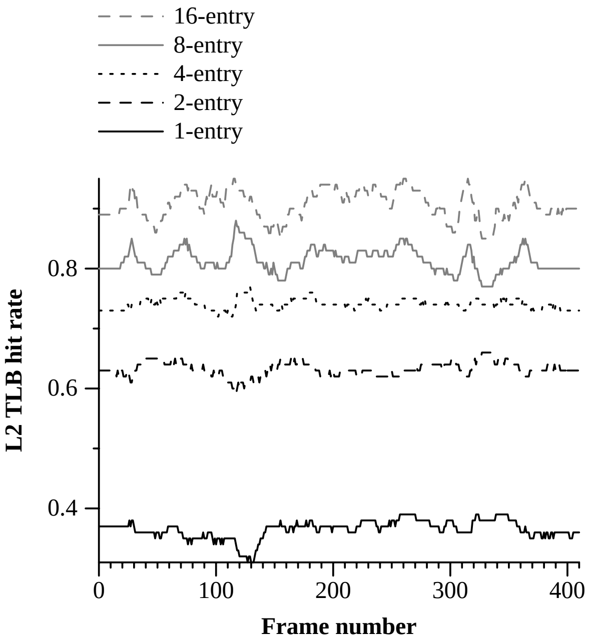

Figure 11. Texture Table TLB hit rates for the Village as a function of number of entries (tiles 16x16, trilinear filtering).

# TLB entries	Village hit rate (%)	City hit rate (%)
1	36%	36%
2	63%	63%
4	74%	75%
8	81%	82%
16	91%	92%

Table 8. Average TLB hit rates for the Village and City as a function of number of TLB entries (tiles 16x16, trilinear filtering).

6 Conclusions and Future Work

We distinguish two existing architectures. The traditional *push architecture* has co-located a large texture memory with the graphics accelerator, requiring the application to bin pack the textures required during each frame. This architecture suffers from application complexity and texture capacity limitations. The more recent *pull architecture* textures directly from main memory, relaxing capacity constraints but introducing main memory and graphics I/O bandwidth as the bottleneck and requiring their performance to escalate with a steeper graphics performance curve.

In this paper we have proposed an intermediate architecture based on multi-level texture caching. Textures are still pulled from main memory, but an L2 cache intermediates between the host and L1 cache. We have found that L2 caching may use 3x to 5x less local memory than the push architecture, and even a 2 MB L2 cache saves from 18x to 140x the download bandwidth over the pull architecture. A simple performance model suggests that L2 texture caching should also result in better performance than the pull architecture.

At least several optimizations to the current work are worth investigation. First, z-buffering before allocating and loading L2 cache blocks should reduce texture depth to something close to one, and may significantly save both local texture memory and block download bandwidth. Second, alternative algorithms to "clock" deserve investigation to avoid "peaky" behavior. Third, while expected-case analysis predicts that L2 texture caching should scale well to high-end machines and scenes, investigation with "workloads of the future" are worthy of pursuit.

Appendix

```
struct {
    Bit                 active
    int                 t_index /* zero if no block allocated */
} BRL[ $N_{blocks}$ ]          /* BRL[] is indexed by 0 ≤  k< $N_{blocks}$ */

struct {
    Bit_vector          sector[]  /* sub-block loaded flags */
    Int                 l2_block  /* zero if no block allocated */
} t_table[ $N_{tentries}$ ];    /* t_table[] is indexed by 0 ≤  k< $N_{entries}$ */

struct {
    Byte                ram[ l2_block_size ]
} L2_cache[ $N_{blocks}$ ]   /* L2_cache[] is indexed by 0 ≤  k< $N_{blocks}$ */

struct texture {
    int                 tstart
    int                 tlen
    Address            L3_start
    … additional fields required by texture-mapping …
} current_texture

int    clock_index              /* current clock index into BRL[] */
int    l2_block_size           /* size in bytes of an L2 cache block */
int    l1_block_size           /* size in bytes of an L1 cache block */
Address    l2_base_addr     /* starting address of L2 cache memory */
```

```
Calculate <tid, L2, L1>
t = current_texture.tstart + L2
addr  = l2_base_addr + (t_table[t].l2_block – 1) * l2_block_size +
          L1 * l1_block_size
test1 = <tid, L2, L1> is in L1 cache
if(  test1 ) {                                          /* L1 hit */
    Retrieve the desired texel(s)
} else {                                                /* L1 miss */
   test2 = t_table[t].l2_block is non-zero
   test3 = t_table[t].sector[L1]
   if( test2 ) {
      if( test3 ) {                                     /* L2 full hit */
         Load L1 sub-block from L2 cache at addr into L1 cache
         BRL[ t_table[t].l2_block – 1 ].active = 1
      } else {                                          /* L2 partial hit/miss */
         Load L1 sub-block from system memory into L2 cache
                at addr, and into L1 cache
         t_table[t].sector[L1] = 1
         BRL[ t_table[t].l2_block - 1 ].active = 1
   } else {                                             /* L2 full miss */
      while( BRL[ clock_index ].active ) {      /* Find a victim */
         BRL[ clock_index ].active = 0
         clock_index = (clock_index + 1) mod $N_{blocks}$
      }
      if(BRL[ clock_index ].t_index )
          Clear t_table[ BRL[ clock_index ].t_index – 1 ]
      Load L1 sub-block from system memory into L2 cache
           at addr, and into L1 cache
      BRL[ clock_index ].t_index = t + 1
      t_table[t].l2_block = clock_index + 1
      clock_index = clock_index + 1
      t_table[t].sector[L1] = 1
      BRL[ t_table[t].l2_block – 1 ].active = 1
   }
}
```

Acknowledgements

Many thanks to Evans and Sutherland Corporation for providing the Village database, and William Jepson at UCLA for the City database. Thanks to Dave Sprague of Intel for funding and useful discussion, Bob Dreyer and Kai Li for reviewing a previous draft, and Kai Li for useful discussion on user-level TLB management. Thanks also to the anonymous reviewers for their helpful comments.

References

[1] K. Akeley, “Reality Engine Graphics,” *Computer Graphics* (Proc. Siggraph), August 1993, pp. 109-116.

[2] J. Blinn and M. Newell, “Texture and Reflection in Computer Generated Images”, *Communications of the ACM*, Vol. 19, No. 10, October 1976.

[3] J. Blinn, “The Truth About Texture Mapping,” *IEEE Computer Graphics and Applications*, March 1990, pp. 78 – 83.

[4] E. Catmull, “A Subdivision Algorithm for Computer Display of Curved Surfaces,” Ph.D. dissertation, University of Utah, 1974.

[5] F. Crow, “The Aliasing Problem in Computer Synthesized Shaded Images,” Ph.D. dissertation, University of Utah, 1976.

[6] F. Crow, “Summed-Area Tables for Texture Mapping,” *Computer Graphics* (Proc. Siggraph), July 1984.

[7] M. Deering, S. Schlapp, M. Lavalle, “FBRAM: A New Form of Memory Optimized for 3D Graphics,” *Computer Graphics* (Proc. Siggraph), August 1994, pp. 167-174.

[8] Y. Denville, “A low-cost usage-based replacement algorithm for cache memories,” *ACM Computer Architecture News*, Vol. 18, No. 4, December 1990, pp. 52-58.

[9] E. Feibush, M. Levoy, and R. Cook, “Synthetic Texturing Using Digital Filters,” *Computer Graphics* (Proc. Siggraph), 14, July, 1980.

[10] J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed., Addison-Wesley, Reading MA, 1990.

[11] Z. Hakura and A. Gupta, “The Design and Analysis of a Cache Architecture for Texture Mapping,*” Proc. of the 24th International Symposium on Computer Architecture*, May 1997, pp. 108 – 119.

[12] P. S. Heckbert, *Fundamentals of Texture Mapping and Image Warping*, Master’s thesis, University of California at Berkeley, June 1989.

[13] P. S. Heckbert and H. P. Moreton, “Interpolation for Polygon Texture Mapping and Shading.” In David F. Rogers and Rae A. Earnshaw, editors, *State of the Art in Computer Graphics: Visualization and Modeling*, pp. 101-111. Springer-Verlag, 1991.

[14] B. Hook, “All I Want for Christmas ’98 is a Hardware Accelerator that Doesn’t Suck,” *Game Developer Magazine*, September 1997.

[15] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York NY, 1984.

[16] Intel Corporation, *Accelerated Graphics Port Interface Specification*, Revision 1.0, Intel Corporation, 1996.

[17] J. Montrym, D. Baum, D. Dignam, C. Migdal, “InfiniteReality: A Real-Time Graphics System,” *Computer Graphics* (Proc. Siggraph), August 1997, pp. 293-301.

[18] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Mateo CA, 1990.

[19] R.N. Ibbett and P.C. Capon, “The Development of the MU5 Computer System,” *Communications of the ACM*, Vol. 21, January 1978, pp. 13 – 24.

[20] D. Peachey, “Texture on Demand,” unpublished manuscript, Pixar, San Rafael CA, 1990.

[21] T. Porter and T. Duff, “Compositing Digital Images,” *Computer Graphics* (Proc. Siggraph), July 1984.

[22] S. Przbylski, *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kaufman, San Mateo CA, 1990.

[23] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli, “Fast Shadows and Lighting Effects Using Texture Mapping,” *Computer Graphics* (Proc. Siggraph), July 1992, pp. 249-252.

[24] M. Shantz, D. Krasnov, A. Kibkalo, A. Subbotin, F. Xie, and T. Park, “Building Online Virtual Worlds,” *Graphicon-96*, July 1-5 1996, GRAFO Computer Graphics Society, State Education Center, Saint Petersburg, Russia.

[25] A. Silberschatz, J. Peterson, P. Galvin, *Operating System Concepts*, Addison-Wesley, Reading MA, 1991.

[26] J. Torborg and J. Kajiya, “Talisman: Commodity Realtime 3D Graphics for the PC,” *Computer Graphics* (Proc. Siggraph), August 1996, pp. 353-363.

[27] Doug Voorhies, Nvidia Corporation, personal communication 1997.

[28] A. Watt and M. Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*, Addison-Wesley, Reading MA, 1992.

[29] T. Whitted, “An Improved Illumination Model for Shaded Display,” *Communications of the ACM*, Vol. 23, No. 6, June 1980.

[30] S. Winner, M. Kelley, B. Pease, B. Rivard, and A. Yen, “Hardware Accelerated Rendering of Antialiasing Using a Modified A-buffer Algorithm,” *Computer Graphics* (Proc. Siggraph), August 1997, pp. 307-316.

[31] L. Williams, “Pyramidal Parametrics,” *Computer Graphics* (Proc. Siggraph), July 1983, pp. 1-11.

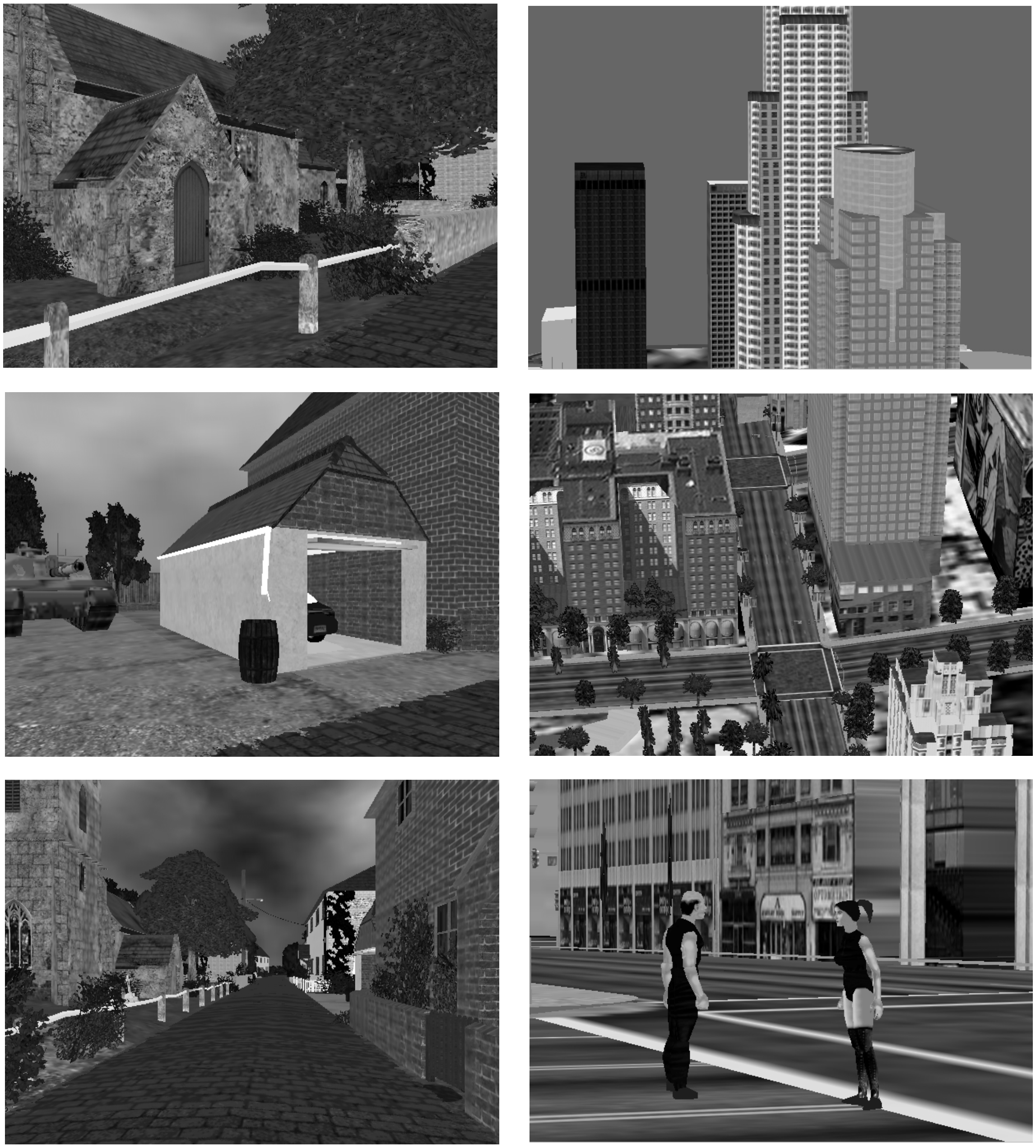

Figure 12. Snapshots from the animation work loads employed in this paper: Village (left) and City (right).