Implementing a Texture Caching System

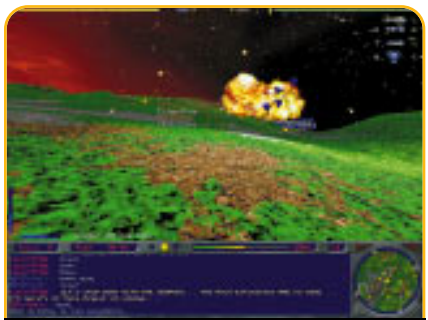
by Jonathan Blow

Texture caching systems are designed to overcome the texture budget limitations of 3D games. Only the textures required to display the current scene are held in RAM. When new textures need to appear in the scene, they are loaded from a larger and slower repository, or they are dynamically generated.

For example, textures can be pulled from disk into system RAM or downloaded from system RAM into the video RAM of a 3D accelerator. Textures can be dynamically generated by combining illumination maps with unlit source textures.

QUAKE was one of the first games to implement a texture caching system that interacts closely with the 3D pipeline to cache graphics in an efficient manner (see References). DOOM cached textures as well, but its system was more of a solid-state approach, as was the data caching scheme in the 2D side-scroller ABUSE. The source code to both ABUSE and DOOM is now available; see the References at the end of this article.

This article is broken into two parts. First, we'll discuss the nature of texture maps and the issues involved in implementing a texture cache. Then, we'll look at some concrete implementations of caching systems used in games that are currently under development.



WULFRAM, the multiplayer tank from Bolt Action Software.

Textures and MIP-mapping

Texture storage is all about MIP-maps. MIP-maps are prefiltered versions of a texture map stored at varying resolutions. To simplify this discussion, we will focus on MIP-maps that are square and are a power-of-two in width (1×1, 2×2, 4×4,). We will speak of a MIP-map level (or MIP-level) as a nonnegative integer that describes the resolution of a MIP-map: a texture at MIP-map level n is 2^n texels square. MIP-level 0 is the smallest size at 1×1 texels, increasing with conceptually no upper bound (though we might voluntarily choose one to ease implementation) (Figure 1).

Jonathan Blow is vice president of software development at Bolt Action Software, a San Francisco-based game developer. He can be reached at jon@bolt-action.com

The reader should note that this MIP-level numbering convention is different from the most commonly used notation, in which MIP-level 0 is the texture at its maximum resolution, perhaps 128×128, level 1 is reduced by one step (that is, 64×64), and so on. That convention doesn't make any sense when you're deep into texture caching: what is the maximum resolution of a dynamically generated plasma fractal?

Let's take a look at the memory required to store textures. Every (uncompressed) MIP-map level of a texture requires four times as much RAM as the level below it. A texture at level n uses

$$Size(n) = \left(2^n\right)^2 = 2^{2n}$$

texels worth of RAM. Storing all MIP-map levels from 0 to n requires

$$SizePlus(n) = \sum_{i=0}^{n} Size(i)$$

Now suppose we have a texture at maximum detail and we want to store all MIP-maps down to level 0. How much extra memory does this require? In other words, how big is SizePlus(n) relative to Size(n)? We can figure this out using some standard power series diddling.

$$s = SizePlus(n-1) = \sum_{i=0}^{n-1} Size(i) = \sum_{i=0}^{n-1} 2^{2i}$$

$$2^2 s = 2^2 \sum_{i=0}^{n-1} 2^{2i} = \sum_{i=0}^{n-1} 2^{2(i+1)}$$

$$2^2 s = \sum_{j=1}^{n} 2^{2j} = \sum_{j=0}^{n-1} 2^{2j} + 2^{2n} - 1$$

$$2^2 s = s + 2^{2n} - 1$$

$$3s = 2^{2n} - 1$$

$$s = \left(2^{2n} - 1\right)/3$$

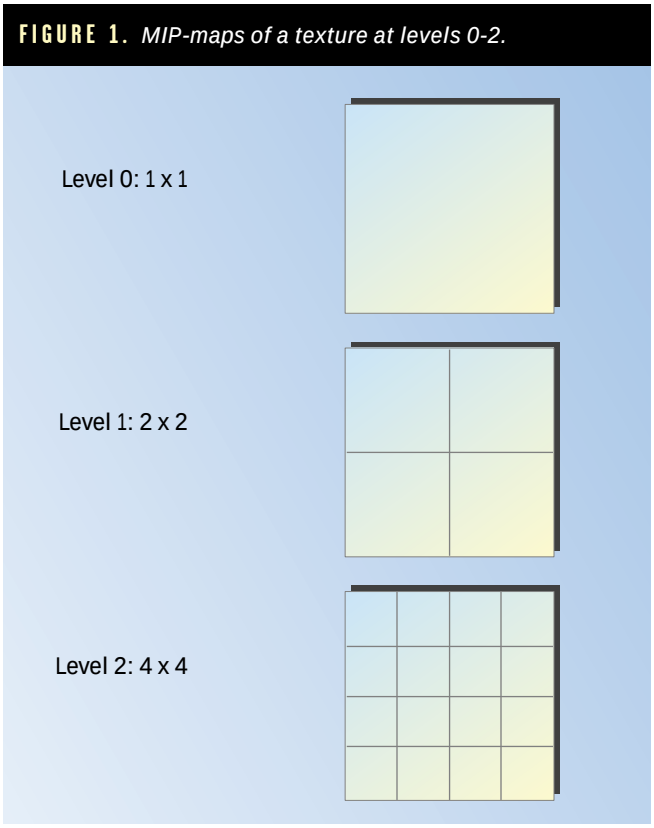
$$SizePlus(n-1) = \left(Size(n) - 1\right)/3$$

Since the amount of memory required to store a texture grows with 2^n as you climb the MIP-map ladder, you need to be careful about holding resolutions that are only as large as you really need. Conversely, because the required memory shrinks as you decrease MIP-map level, storing all the detail levels that are smaller than the level that you really need requires only one-third more memory.

Decisions Must Be Made

To build a system that's good at texture management, you want to keep only the necessary textures in RAM at only the necessary detail levels. The set of necessary textures will change from frame to frame based on what the 3D pipeline decides to do. To be effective, our caching system must predict and accommodate the needs of the pipeline. A good cache will be specifically designed for a particular application. (Does the cache need to handle dynamic or static textures or both? How many textures and at what sizes?) See Figure 2.

FIGURE 1. *MIP-maps of a texture at levels 0-2.*



Our system will have to perform the following tasks. There are a variety of ways to approach each problem, each with its own advantages and drawbacks.

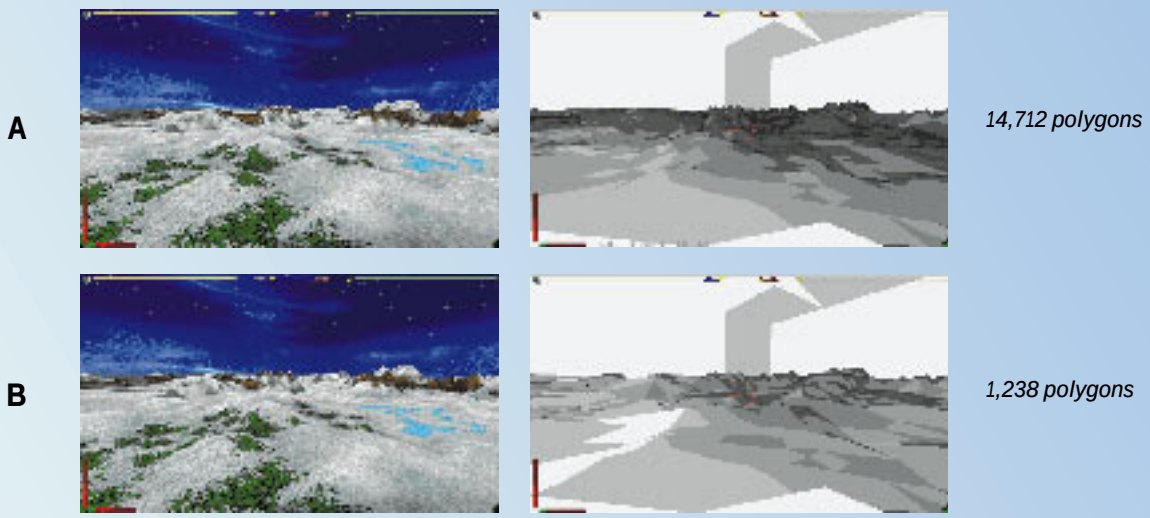
GENERATE TEXTURE REQUESTS. In order to fetch textures into RAM, the main 3D pipeline must tell the cache which textures it needs. Typically, this communication takes place in one of two ways. When the pipeline decides to emit a polygon using a particular texture, it calls a procedure to request that texture from the cache; this is known as pipeline hooking. Zoning, on the other hand, divides the world into zones; the pipeline predicts which textures it will need based on the position of the viewpoint, then requests those textures from the cache in batches.

Zoning generally requires preprocessing to determine visibility between spatial regions; however, it can be less expensive than pipeline hooking during run time because it does not require per-polygon tests. Pipeline hooking will generally require vigorous prefetching, but it is more versatile in that it places fewer restrictions on the application as a whole.

The concept of zoning applies itself more naturally to occluded indoor environments (such as QUAKE's) than to outdoor scenes (such as TERRA NOVA's).

DETERMINE THE DETAIL LEVEL. To operate efficiently, the cache should only retrieve detail levels that are required to draw the scene. We can achieve a precise solution by computing the texture gradients for each polygon that we emit and using those gradients, along with the nearest and furthest

FIGURE 2. *MIP-map level distribution for a typical scene from WULFRAM.*



On the left is the actual rendered image; on the right is a gray-coded version, with MIP-maps at level 0 drawn in dark gray, ranging up to level 7 drawn in white. Version A of the scene has detail reduction turned off; version B has it turned on. Note that A generally contains textures that are much smaller than B (the image on the right is darker). The point of this example is to highlight the interaction between cache design and pipeline design. The detail reduction system in WULFRAM strongly affects its texture usage statistics, changing the job that the cache must perform. A scheme that is memory-efficient, but behaves poorly with many small textures, would not suit applications with scenes resembling A, but would be fine for B.

vertices of the polygon, to tell us which MIP-map levels are needed. Alternatively, finding a conservative estimation involves cheaper computations than those required for the precise solution scheme. We can use a polygon's nearest vertex and a per-polygon precomputed coefficient to find a conservative upper bound on the necessary MIP-level. Or we can simply dodge the problem by always fetching textures at full resolution. We use the caching mechanism only to decide which textures are necessary.

Superficially, the conservative estimation scheme seems more attractive than attempting to find a precise solution because of our tendency to play bean counter with CPU cycles spent per polygon. However, since the conservative estimation scheme will generally request higher MIP-levels than the precise solution scheme, it places more load upon the cache. When comparing these methods in the WULFRAM engine, we found that on average the conservative estimation scheme would request textures one or two MIP-levels higher than would the precise solution scheme. This meant that the texture cache was fetching and synthesizing textures that were about eight times as large as they needed to be, slowing texture construction to an unacceptable rate. The frame rate also became very jumpy because texture-building costs are less evenly distributed than per-polygon costs.

However, the effect of the MIP-map level decision is application-dependant, and the conservative estimation scheme could be better than the precise solution scheme in some cases, especially if polygon counts are very high compared to the frequency of cache misses.

FILL THE CACHE. When the pipeline needs a texture, the cache makes it available. A synchronous fetching scheme momentarily pauses execution of the main program while the texture is being placed in the cache. Asynchronous fetching, on the other hand, allows execution of the main program to proceed in parallel with the cache filling process.

If textures are being dynamically generated into the cache in a CPU-bound manner, asynchronous fetching will only result in performance improvements on a multiprocessor machine. When a cache fill involves reading from external storage such as a bus-mastering hard drive controller, however, an asynchronous fetch won't actively consume CPU cycles.

EMPTY THE CACHE. The cache is of finite size. When it fills up, we must discard textures that are no longer needed. The games surveyed in the latter half of this article use a few different methods for emptying the cache. One common term we'll use is LRU, meaning Least Recently Used. In an LRU scheme, cache elements are marked with timestamps indicating the last time they were used. The element with the oldest timestamp is discarded.

MANAGE MEMORY. The texture caching system must find available memory for new textures efficiently. Also, if precious cache space is to be effectively utilized, fragmentation must be kept to a minimum. For a memory management scheme to be effective, it must be designed around permitted texture dimensions and the application's typical usage statistics. Developers are using a huge number of approaches to memory management; I've outlined several in the survey.

Potential Enhancements to Base Functionality

Once a caching system is in place, many things can be done to improve performance. Here are a few possibilities:

PREFETCHING. Fetching a texture invariably incurs a computational cost; requests for new textures will often occur in bursts, resulting in uneven demand on the CPU. To maintain the frame-rate level, fetch some textures before they're actually needed (during a lull in the handling of cache misses), thereby spreading each burst across more frames.

If texture fetching occurs asynchronously, it's possible that a texture won't have arrived in the cache by the time we need to draw it. In this case, we typically draw some sort of stand-in for that texture, which results in a loss of image quality. Prefetching textures, however, minimizes the possibility that our cache will be missing textures.

An easy way to prefetch textures is to bias MIP-level computations (recall the precise solution and conservative estimation schemes) so that larger MIP-maps are requested slightly early. In the very common case where the viewpoint is moving forward, this has the effect of automatically prefetching higher-resolution MIP-maps of visible textures.

COMPRESSION. If the cache system manipulates compressed textures, throughput requirements (and CPU requirements due to copying) will be reduced, and a cache of a given size will be able to hold more textures. However, this idea has many drawbacks. A software rendering system typically needs to manipulate uncompressed textures. Hardware accelerators use many different types of compression, so textures will often need to be uncompressed before they are sent to hardware (especially when using an abstracted 3D API such as OpenGL). The cost of decompressing your textures will usually outweigh the compression's initial benefits.

ADVANCED HIDDEN SURFACE ELIMINATION. Rendering scenes of high depth complexity using the painter's algorithm or a depth buffer places an unnecessary load on the texture cache; we will often load the cache with textures that are actually invisible because they're on polygons that are occluded by other polygons. A reasonable form of occlusion culling could reduce cache load tremendously for certain types of scenes.

Generating Lightmaps

Many upcoming games use "illumination maps" (or "lightmaps," the term used in the rest of this article). Lightmaps are a good example of dynamic texture generation: a source texture will often be combined with a lower-resolution lightmap to produce a shaded texture, which is then used on polygons in the scene. The shaded texture remains in the cache as long as it is being displayed. When a polygon's lighting changes, its texture is invalidated and recomputed using a new lightmap. The generation of the shaded texture can consume an appreciable number of CPU cycles, since effects such as bilinear filtering are often used to compensate for the lightmap's lower resolution (Figure 3).

In the survey of game engines in the latter half of this article, we'll speak of "lumel ratio." A lumel is one pixel of a lightmap, and the lumel ratio is a lumel's width divided by the width of a texel from the texture map to which the lightmap is being applied.

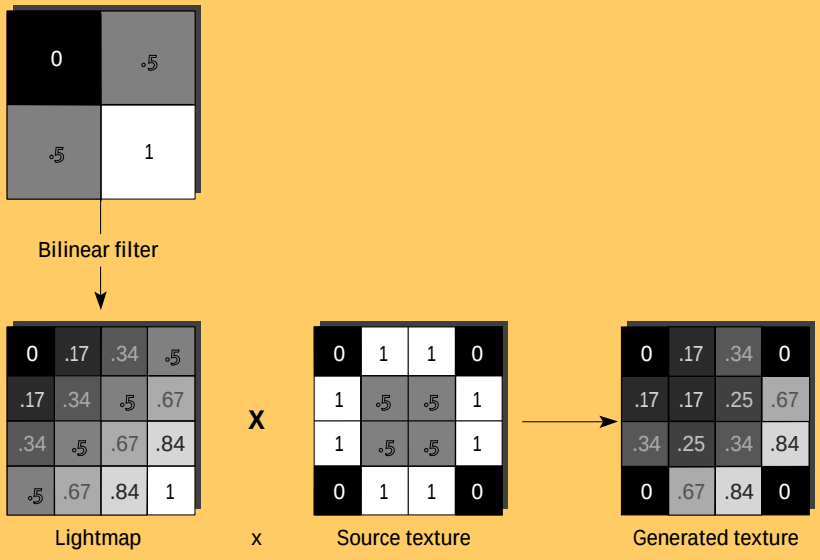


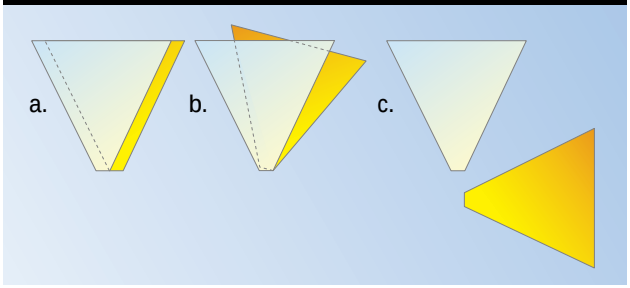
FIGURE 3. *A lightmap at level 1 is bilinear filtered up to level 2 so that it can be combined with a source texture at level 2 to generate the resultant texture. The lumel ratio in this example is 2:1.*

Performance Patterns

We would do well to note some basic truths about texture-caching systems. If the viewpoint and all objects in the scene are stationary, cache misses will be at a minimum because the scene will be the same from frame to frame. In this case, we only need to fetch new textures for animated polygons or when the cache is too small and we are forced to discard live textures. If we consider both these circumstances to be rare, then the cache is basically unstressed with a stationary viewpoint.

When the viewpoint moves linearly, small wedges of the frustum will come into view, and some textures that were already visible will shift to a higher MIP-map level (Figure 4a). This shift causes texture fetches, putting some load on the cache. The more quickly the viewpoint moves, the heavier the load on the cache, because more of those events are happening each frame. Backward motion places especially high stress on the cache because it introduces new textures at their highest detail levels (textures coming into view from the sides will generally be needed at intermediate detail levels).

FIGURE 4. *Regions of space revealed by a viewpoint moving in various ways: a is linear motion (sideways); b is rotational motion; and c is teleportation. The blue trapezoid represents the position of the view frustum before movement; the gold trapezoid is the frustum after movement.*



Rotating the viewpoint stresses the cache even more. At each frame, a wedge comes into view that widens with distance from the viewpoint (Figure 4b). Generally, the new volume of space revealed by a rotation will be much larger than the volume revealed through linear movement, resulting in a corresponding increase in cache events.

Long-distance "teleportation" of the viewpoint is the nastiest type of movement in terms of stressing the cache — the newly revealed area could potentially consist of the entire frustum (Figure 4c.)

Quality Loss

If cache filling occurs asynchronously and a texture isn't ready when we need it, we'll typically use a stand-in for the texture, such as a lower-resolution MIP-map. In this case, we suffer a loss in image quality. We can categorize potential magnitudes of quality loss in the same way that we just categorized cache stress: according to viewpoint movement.

A stationary scene should incur no quality loss in the steady state, again provided that no exceptional circumstances exist. Most linear motion in games is forward, and forward motion causes the fewest problems — when we're nearing a texture and it switches MIP-map levels from n to n+1, it does so at the point where there is almost no visual difference between level n and level n+1. This is the whole point of MIP-maps; if the texture arrives a frame or two late, the impact on the scene is minuscule.

Handling linear motion that causes new textures to come into view (say, sideward or backward motion) isn't too difficult because those newly-appearing textures are easy to prefetch. If a polygon comes into view at time t, then at time t-1 it was probably just outside the view frustum and was rejected during clip testing. This brings the polygon quickly to the attention of a properly concerned rendering engine.

Rotating the viewpoint can be slightly more difficult because the wedge widens as distance from the viewpoint increses. Nevertheless, in practice, a rotating viewpoint won't cause severe problems if adequate prefetching is in place.

Remember, however, that the quicker the motion, the greater the quality loss, because the area of newly-exposed polygons (which are now carrying erroneous textures) will be larger. Teleportation, therefore, is problematic because all textures will be unexpected.

These same principles regarding a moving viewpoint also apply to moving objects in a scene. Note, however, that rotating objects impose little stress on the cache. Textures on rotating objects first come into view edge-on, which means they are only necessary at minimal detail levels. As the polygon spins into view, the necessary MIP-map level rises in a continuous manner, just as it does for textures nearing a forward-moving viewpoint.

Survey of Engines in Progress

In the next section, we examine texture caching techniques in use by games and engines currently in development. Because these descriptions are only snapshots of prerelease software, they may not be accurately representative of the games in their final forms.

The main purpose of these descriptions is to provide examples of the design decisions that developers have employed to suit the games they are making. Because these game engines are very different from each other, it isn't useful to see these descriptions as "feature lists" in comparing engines to determine which is "better." All the developers involved have been very kind in sharing information about their systems and should be appropriately thanked.

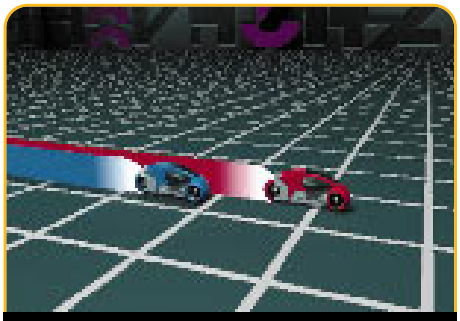
GOLGOTHA. According to Trey Harrison at Crack Dot Com, GOLGOTHA uses 16-bit textures that are initially JPEG-compressed; they are uncompressed at the beginning of each level, so the rendering engine only sees them as uncompressed. When unpacked, the textures are stored in the native formats of the current display (on 3Dfx Voodoo Graphics cards, for example, opaque textures are stored as 565 RGB, textures with holes are stored as 1555 ARGB, and textures with a full alpha channel are 4444 ARGB.)

Because of the high resolution and large number of textures in the game, textures reside primarily on disk and are cached into texture RAM. The game uses about 700 textures, typically 256×256 in size, requiring around 100MB of storage once they are uncompressed. Textures are power-of-two in width and height, but not necessarily square. The smallest handled MIP-map level is 16×16.

To determine which resolution of a texture it needs, GOLGOTHA uses a conservative estimation scheme based on the closest (1/z) of the polygons being displayed. When a new MIP-map is required, it's loaded asynchronously from disk (by a separate thread) into a

Golgotha, Crack Dot Com.

The Crystal Space engine by Jorrit Tyberghein.



The Hyper3D engine by John McCarthy.

temporary holding area in system RAM. At a fixed point in the rendering cycle, textures are downloaded from system RAM into texture RAM and removed from the holding area, eliminating concerns over the threadsafeness of 3D hardware APIs.

Until a texture arrives in memory, the MIP-map at the closest available resolution will be used for rendering. Some MIP-maps will always be available since the lowest resolution of each texture is always kept in texture RAM. Currently, the engine doesn't prefetch textures, although the final version of the game may implement this feature.

Texture memory is organized as a linked list of free space (used when finding memory for a new texture) and a linked list of allocated space (used when deciding which textures to discard from the cache). An LRU scheme is used to throw out textures when the cache is full.

Memory management becomes "a bit more ugly" under higher-level APIs such as OpenGL and Direct3D. With these APIs, the engine will discover that the cache is nearly full when an allocation request fails; textures must then be freed in a slightly blind manner because it's not possible to know what effect their deallocation will have on the fragmentation of the heap.

In order to keep the frame rate level, the main thread is limited in the number of textures that it can request per frame. The limit is imposed on both the number of textures and the total bytes requested and is adjustable based on system speed.

CRYSTAL SPACE. This is a freeware game engine under the GNU license. It was written by Jorrit Tyberghein. Crystal Space textures are 8 bits in depth and a power-of-two in both width and height, but not necessarily square. Most textures, however, tend to be square, typically 128×128 or 64×64. Four MIP-map levels are supported.

The engine uses lightmaps, which can be RGB or monochrome with a lumel width of 16:1 (see "Generating Lightmaps" for a definition of lumel). All source textures and static lightmaps are stored permanently in system RAM. The purpose of the texture cache is to manage the storage of textures generated by combining source textures with lightmaps. When a texture is built for a specific polygon, bilinear interpolation is used to expand the polygon's lightmap up to the necessary resolution; the source texture is tiled until it is the proper size. Since those 64×64 or 128×128 source textures can wrap across a polygon several times, the cached textures can be "very large." Since Crystal Space currently uses a software renderer, graphics hardware cannot impose a maximum limit on texture sizes.

The texture cache is of a fixed size; when it's full, an LRU scheme is used to discard textures until there is enough space. At present, C++ memory management (`new` and `delete`) is used for the texture cache, though this may be replaced with a custom memory manager in the near future.

Hyper3D. This engine, written by John McCarthy, is made to render exterior scenes and space environments, with an emphasis on non-static objects. Textures are of arbitrary sizes and arbitrary multiple-of-8 bit depths. Each "color" of a texture is an 8-bit channel representing an arbitrary property; for example, a texture containing RGB color, alpha, and bump map information is stored in "40-bit color." Each channel is stored contiguously (a buffer consisting of two RGB texels would be stored as RRGGBB, rather than RGBRGB). The frame buffer is also divided into channels or "output banks." This memory organization simplifies the use of texture mapping operations to create complex effects. The banks of the frame buffer are combined into native format when it's time to display each frame. Because of this per-frame translation, the color model produced by mapping operations is not restricted to RGB; HSV textures can readily be used. Vertex-based lighting is supported, but because of the customizable screen mixing, lightmaps can also be used; they are treated just like any texture, then used to scale color in the final buffer synthesis.

Because of the high resolution and depth of textures, they are stored on disk and demand-loaded into system RAM when the renderer tries to use them. Loading occurs synchronously at a fixed point in the update cycle; the number of texture loads per frame is capped at a maximum (adjustable) value to keep frame rate level.

When textures need to be displayed but are not yet loaded, the renderer draws a flat-shaded polygon instead. As Hyper3D author John McCarthy says, "Frame rate was more important than perfectly correct scenes."

DESCENT 3 from Outrage Entertainment.

KAGE, Terminal Reality Inc.

Wulfram, Bolt Action Software.

The texture cache is a linked list of 256×256-byte blocks. The cache manager fits textures together within these blocks. To reduce fragmentation, textures are rounded up to the nearest 8-texel boundary in x and y (so a 73×131 texture will take the same amount of space as a 79×133 texture: they will both occupy an 80×136 aperture). When a new texture won't fit into an available block, textures are discarded based on their reference counts (textures being used 0 times are thrown out first, then textures being used 1 time, and so on). If the heap becomes too full or very fragmented, currently-used textures will have to be discarded and reloaded during the next frame. These discards tend to reduce fragmentation.

DESCENT 3. According to Jason Leighton at Outrage Entertainment, DESCENT 3's textures are stored in 16-bit color; they are all square and power-of-two in width. QUAKE-style lightmaps are used, using lumels at a size of 16:1. Because there are many lightmaps and each lightmap is small, the caching system must efficiently handle large numbers of small textures, in addition to being effective for larger textures.

The main purpose of DESCENT 3's caching mechanism is to efficiently use the texture RAM of a 3D accelerator. All textures from the current game level are held in system RAM, with the cache pulling them into texture RAM as needed.

The square, power-of-two texture geometry allows the system to manage texture RAM without the possibility of fragmentation. Texture memory is divided into an ordered series of blocks, each of which stores textures of a given MIP-map level. There are seven such regions, with the smallest storing 2×2 textures and the largest storing 128×128 textures. Suppose that the renderer needs to use a new 16×16 MIP-map, but the 16×16 region is full. First, the system checks the 16×16 region to see if any textures can be discarded. If so, the new texture is simply uploaded over the old one. If not, the 16×16 cache region needs to grow. It can grow to the left, taking space away from the 8×8 region, or to the right, stealing space from the 32×32 region (Figure 5). Heuristics are used to make the right decision about which way to grow.

Note that when a region grows to the right, it steals only one texture from the neighboring region and gets enough space to hold four of its own textures. When growing to the left, a region must consume four of its neighbor's textures to produce space for only one of its own. When memory is stolen from a region, any textures residing within the reassigned memory are discarded.

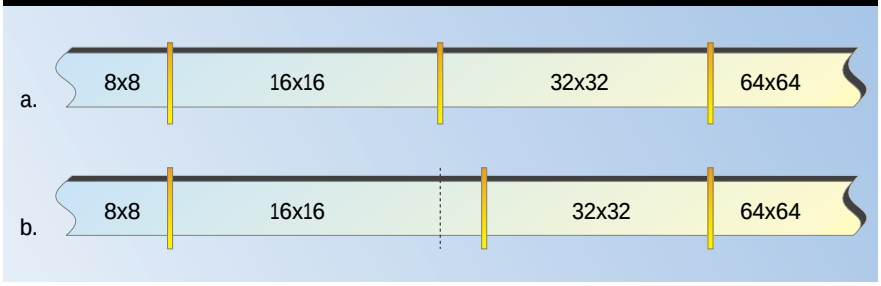
This memory management scheme requires direct control over a single, continuous memory aperture. Many hardware-specific APIs (such as 3Dfx's Glide) support this type of access naturally. However, when using higher-level APIs such as OpenGL or Direct3D, such direct control is lost, and this sort of scheme becomes difficult or impossible.

KAGE. Terminal Reality Inc. designed this an engine to render arbitrary polygonal scenes generated in editors such as 3D Studio MAX. According to Paul Nettle, KAGE uses lightmaps with a lumel width of 4:1. It uses a precise solution scheme to determine the MIP-map levels of polygons in the scene.

Textures are of unconstrained size, but their width or height cannot exceed 256 texels. The cache is stored as an array of pages. Each page is a two-dimensional image and is indexed by a list of occupied space. Within a page, textures are lined up from left to right.

Each page stores textures only within a certain range of texture heights. The global page array can be constructed so that each page height is a multiple of n. (If n is 8, there will be pages that are 8 texels high, pages that are 16 texels high, and so on. Since the height is capped at 256, there would be 32 different page heights.) Textures are effectively rounded up to the nearest multiple of n in height and stored in the appropriate page; the gap between the bottom of an allocated texture and the

FIGURE 5. *DESCENT 3's memory management scheme. a: Memory is partitioned into contiguous regions based on texture size. b: The 16x16 region has grown by taking space from the 32x32 region.*



bottom of a page is wasted memory. In practice, the amount of waste produced is very small.

When it is time to allocate space for a surface, the system searches pages of the appropriate size for a "best fit." This search is capable of examining continuous groups of allocated areas and gaps between them to determine whether that area should be cleared for the new texture (by comparing the combined most-recently-used values of allocated shards and taking the lowest).

WULFRAM. In this game from Bolt Action Software, textures are 8 bits deep, square, and power-of-two in width. At any given time, the game will have random access to around 2,800 textures, about 1,500 of which are generated at the beginning of each level. Most textures are 128×128, and all MIP-map levels of every texture are stored on disk — the size of the texture store is around 58MB. The caching system's main emphasis is on caching textures from disk in system RAM.

Detail reduction is employed heavily in landscape rendering using an algorithm based on Peter Lindstrom's SIGGRAPH paper (see References). To accommodate this detail reduction, the engine dynamically generates textures that are composites of the source textures in the texture store. Monochrome lightmaps are also used; lumel width varies within the scene, but the average is around 16:1.

A dependency management scheme is employed to synchronize the fetching of multiple textures from disk to build a composite. MIP-maps that are members of visible composites are locked in system RAM so that they are readily available when the composite needs to be resized or broken into smaller pieces.

Prefetching is used in several places. Animations (such as explosion bitmaps) are fetched several frames ahead, and the first few frames of each animation are always locked in system RAM at their maximum detail levels. Sky textures near the edges of the view frustum, as well as ground textures very close to the viewpoint, are prefetched.

MIP-map levels 0-3 of all textures are kept in system RAM at all times. The fetching of textures from disk is performed asynchronously, and whenever a texture is not yet ready, its highest available MIP-maps are used.

The texture cache is not limited in size, but it's generally kept small by a cache sweeper, which looks at 1/16 of the textures in the cache at each frame and discards those that have not been used for 300 milliseconds. After some reflection, this appears to be a less than optimal cache design, but it's adequate for present purposes. The texture cache generally occupies between 2MB and 4MB of system RAM when resolution is set to 640×480. ■

Acknowledgements

Many thanks to Trey Harrison at Crack Dot Com, Crystal Space author Jorrit Tyberghein, John McCarthy, Jason Leighton at Outrage Entertainment, and Paul Nettle at Terminal Reality Inc. for their indispensable help in supplying information for this article.

REFERENCES

Caching graphics in a game is certainly not a new practice. See Jonathan Clark, "Object Cache Management," *Game Developer*, February/March 1996, for a discussion of data caching in the 2D game ABUSE; this article will serve as a good introduction for those not used to thinking about caching.

For implementations of simpler caching schemes used by successful games, the reader is referred to the source code to ABUSE (www.crack.com/games/abuse) and DOOM (ftp://ftp.idsoftware.com/idstuff/source).

More information about KAGE can be found at www.terminalreality.com/engine/kage.html. For other technical info and some research papers that were influential in KAGE's design, see www.grafix3d.dyn.ml.org.

Crystal Space executables and source code can be found at www.geocities.com/SiliconValley/Horizon/3856.

Information about GOLGOTHA, including demo executables, can be found at www.crack.com/games/golgotha.

Technical descriptions of WULFRAM's implementation can be found at www.bolt-action.com.

General information about DESCENT 3 can be found at Outrage Entertainment's web site, www.outrage.com. At present, there is not much technical information, but the site promises to include developer comments in the future.

John McCarthy's home page, including some tidbits related to Hyper3D, can be found at www.geocities.com/SiliconValley/Peaks/6846. He can be contacted at John@McCarthy.net.

A good explanation of lightmaps can be found in Michael Abrash's "Quake's Lighting Model: Surface Caching," *Dr. Dobb's Sourcebook* #260, November/December 1996.

Zen of Graphics Programming (Second Edition) (The Coriolis Group, 1996) by Michael Abrash is considered a classic.

You can get details about computing texture gradients in screen space in Chris Hecker's "Perspective Texture Mapping Part 1: Foundations," *Game Developer,* April/May 1995.

Hin Jang's "Tri-Linear MIP Mapping," available at www.scs.ryerson.ca/~h2jang/gfx_c.html, contains a good introduction to MIP-mapping.

Peter Lindstrom's "Real-Time, Continuous Level of Detail Rendering of Height Fields" (*SIGGRAPH 96 Conference Proceedings*) outlines an algorithm that's handy for landscape rendering.

Paul Nettle's "The KAGE Surface Caching Mechanism" is available at www.terminalreality.com/engine/kage.html

Discrete Mathematics and its Applications (McGraw-Hill, 1991) by Kenneth H. Rosen is a good text for helping you figure out power series.

Functions used to determine texture detail levels can be found in section 3.8.1 of *The OpenGL Graphics System: A Specification (version 1.1)* by Mark Segal and Kurt Akeley. It's available at www.sgi.com/Technology/OpenGL/glspec1.1/glspec.html.

Choosing which elements of a cache to discard is a well-studied subject. For starters, look up LRU in *Operating System Concepts* (Third Edition) (Addison-Wesley, 1991) by A. Silberschatz, J. Peterson, and P. Galvin.