Software Virtual Textures

J.M.P. van Waveren

February 25th, 2012

Abstract

Modern simulations increasingly require the display of very large, uniquely textured worlds at interactive rates. In large outdoor environments and also high detail indoor environments, like those displayed in the computer game RAGE, the unique texture detail requires significant storage and bandwidth. Virtual textures reduce the cost of unique texture data by providing a sparse representation which does not require all of the data to be present for rendering, while leaving the majority of the texture data in highly compressed form on secondary storage. Virtual textures not only provide for a reduction of memory requirements but also improved rendering performance through a reduction in both graphics driver and hardware state changes because many surfaces can use a single virtual texture without the need for per surface texture selection. Several practical examples are discussed to emphasize the challenges of implementing virtual textures in software without special graphics hardware support and viable solutions are presented. These include solutions for address translation, texture filtering, oversubscription, LOD snapping, compression, caching, and streaming.

1. Introduction

In computer graphics, texture mapping is used to add surface detail to geometric primitives. Even though texture mapping enables efficient management and rendering of surface detail, unique texture data requires significant storage and bandwidth. Therefore, a small set of textures is often re-used and it is common practice to use tiling textures that can be repeated many times on a single surface. Multiple textures can also be blended together at run-time to composite a variety of detail from a small set of source textures. These can all be considered specialized forms of texture compression that place a significant burden on the content generation pipeline. By providing a sparse representation, a virtual texture significantly reduces the cost of unique texture data.

A sparse representation of a texture describes a layout in memory that does not require all of the data to be present for rendering. In the context of a mip-mapped texture, a sparse representation can allow different parts of the image to be available at different resolutions. This is important because in any given rendered scene, only a sparse sampling of the data is used anyway, even if all the data is present. Exploiting this sparsity through a virtual texture involves a pool of physical texture pages and a page table that maps virtual addresses to physical ones, similar to a virtual memory system.

A virtual texture, however, differs from other forms of virtual memory in two key ways. First, with a texture it is possible to fall back to slightly blurrier data without stalling execution. A standard virtual memory system must stop the executing process until the system loads in the requested data. Second, lossy compression of the data is perfectly acceptable for most uses. These two key differences make a virtual texture amenable to trade-offs in quality that do not sacrifice performance and allow a significant reduction in memory requirements.

A virtual texture allows not only for a significant reduction in memory requirements but it also allows all surfaces to be drawn in a single batch of geometry. Different surfaces can use different parts of the same virtual texture and there is no need for per surface texture selection. When virtual textures are applied universally, geometry is typically only broken up into multiple batches for culling granularity. This can significantly improve the rendering performance because of the improved culling and the reduction in both graphics driver and hardware state changes. Virtual textures also make it possible to build uniquely textured static worlds with per texel pre-computed lighting stored directly in the texture data which can further reduce the number of rendering passes and state changes.

Virtual textures provide significant benefits such as a reduction of memory requirements and improved rendering performance. However, implementing virtual textures in software, without special graphics hardware support, involves various challenges. This paper discusses these challenges and viable solutions are presented that trade performance, quality and memory requirements.

2. Previous Work

For terrain rendering there are numerous approaches to managing very large texture data sets by adaptively selecting the texture level of detail (LOD) while selecting or constructing the geometry level of detail. Some of these approaches use pre-calculated geometry LODs with baked in texture LODs [1, 2, 3, 4], while others dynamically generate the geometry LOD and in the process assign different texture LODs [5, 6]. None of these approaches implement virtual textures through per fragment texture address translation. Instead, different texture coordinates are supplied with different pre-calculated versions of the geometry, or geometry is generated on the fly with new texture coordinates.

The clip-map [7, 8, 9, 10, 11, 12] is one of the first effective schemes for virtual textures with per fragment texture address translation. The clip-map, introduced by Tanner et al. in 1998 [7], consist of a stack of images similar to a mip-map hierarchy. However, whereas mip-map levels cover the whole texture with images of increasing size, the clip-map uses fixed size levels that cover a decreasing area around a single focus point that is placed somewhere on the texture. Using a region of interest around a single focus point significantly simplifies the mapping of texture data to geometry and the texture address translation during rendering has minimal complexity. However, the single focus point of a clip-map limits this texture management scheme to environments with a natural spatial correlation between the texture data and the geometry, like for instance mostly flat contiguous terrain.

Recent virtual texture systems are more flexible and mimic the virtual memory management of an operating system on modern CPUs [16, 17, 18, 19, 20, 21, 22, 23, 24]. Textures are divided into small pages that are automatically cached and loaded into video memory as required for rendering. These systems use real-time per fragment texture address translation through page tables and/or mapping textures. Because there is not necessarily a natural correlation between the geometry and texture pages, these virtual texture systems require feedback from the rendering pipeline to determine which pages need to be resident [13, 14, 15]. This paper describes the challenges of implementing this class of virtual texture systems in software without special graphics hardware support.

There are also software systems for generic virtual memory management on graphics hardware like GLIFT [25]. Such systems do not automatically exploit the specific properties of virtual textures that make them amenable to trade-offs between quality and performance. However, a system like GLIFT can be used as a foundation to build a virtual texture system upon. Instead of using a more generalized sub-system like GLIFT, the focus here is on virtual texture systems that map directly to the underlying hardware.

3. Rendering With Software Virtual Textures

3.1 Address Translation

A virtual texture is divided into small pages that are loaded into a pool of resident physical pages as required for rendering. These small pages are square blocks of texels, typically on the order of 128 x 128. The pool with physical pages is a fully resident texture that is logically subdivided into such square blocks of texels. While a virtual texture can be very large (say a million pages) and is never fully resident in video memory, the texture that holds the pool of physical pages is fully resident but much smaller (typically only 4096 x 4096 texels or 1024 pages). Virtual texture pages are mapped to physical texture pages, and during rendering virtual addresses need to be translated to physical ones.

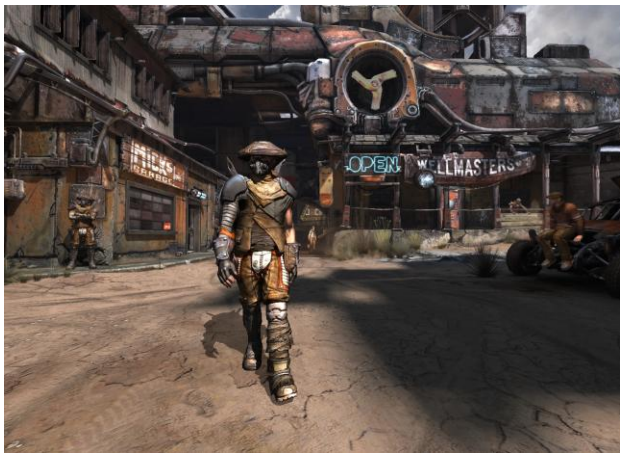


Figure 1: Scene with virtual textures applied to all geometry.

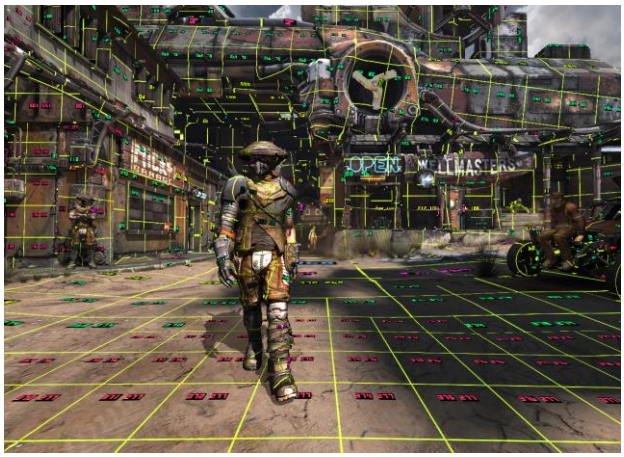


Figure 2: Same scene with highlighted virtual texture pages.

In its simplest form, the virtual to physical translation is equivalent to finding the desired level of detail (LOD) by using the virtual texture address to walk the quad-tree that represents the mip hierarchy of the currently resident texture pages. Every node in the quad-tree provides a scale and bias that will convert a virtual address inside a virtual page to a physical address inside a physical page. The scale is the ratio between the size of the virtual mip level and the size of the physical texture. The bias is the offset to the *physical* page in the physical texture, minus the scaled offset to the *virtual* page in the virtual mip level. While walking the quad-tree to find the desired LOD at a given virtual address, either the scale and bias for the desired LOD are found, or the quad-tree terminates early and the scale and bias at the final node are used for the address translation. In the latter case, the address translation falls back to a page from a coarser mip level because the texture page for the desired finer mip level is not yet available in the pool of physical pages. Figure 3 shows an overview of the virtual to physical address translation and the calculation of the scale and bias.

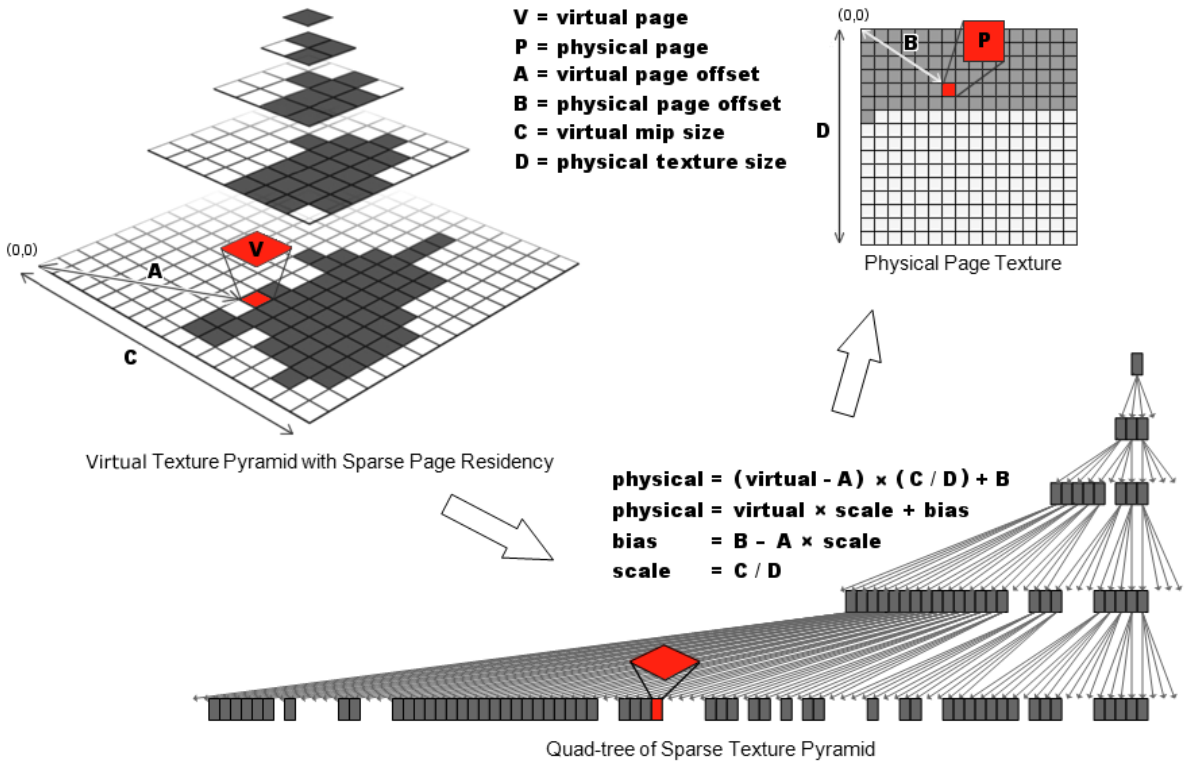


Figure 3: Virtual to physical address translation.

The full calculation of the scale and bias (taking page borders into account) can be found in appendix A. Using only the quad-tree data structure with currently resident texture pages allows for a minimal memory page table, but it has the worst case access latency because it requires a dependent read for each finer level of detail accessed. Instead of walking the quad-tree, the address translation can be implemented in various ways that trade performance and memory.

A straightforward approach to implementing the virtual to physical translation is looking up the scale and bias in a mip-mapped texture with one texel per virtual page. In effect this mip-mapped texture stores the complete quad-tree data structure with a node for every virtual texture page whether it is resident or not. A regular lookup into this page table texture allows the texture hardware to be used to compute the nearest texel of the nearest mip level that corresponds to the virtual texture page for the desired LOD at a given virtual address. The texture lookup is biased with the base-two logarithm of the page width to account for the size difference between the virtual texture and the page table texture. The scale and bias retrieved from this page table texture can be used directly to map a virtual address to a physical one. A texel of this texture will store a scale and bias for a texture page from a coarser mip if the desired finer mip is not yet available in the pool of physical pages.

If the virtual and physical texture are both square, there only needs to be a single scale value for both axes. However, to map virtual pages to arbitrary physical pages, there need to be two bias values (S-bias and T-bias). The scale and bias need to be stored with at least 16 bits of precision to be able to support reasonably large virtual textures (>= 1024 x 1024 pages with borders). This

means that the scale and bias can be stored as 32-bit floating-point (FP32) but not 16-bit floating-point (FP16) because there are not enough bits in the FP16 mantissa. The simplest implementation results from storing the three values (ST-scale, S-bias, T-bias) in a four component FP32 (FP32x4) texture. The implementation in a fragment program of the virtual to physical translation using such a FP32x4 page table texture is trivial and consists of no more than a texture lookup and a multiply-add as shown in appendix A.1. Unfortunately, for a reasonably large virtual texture the FP32x4 page table texture consumes a fair amount of memory. For instance, for a virtual texture with 1024 x 1024 virtual pages, the page table texture takes up 21.33 MB of memory.

To reduce the memory requirements, the page table can be split into two textures. The first texture is mip-mapped with one texel per virtual page. Once again a regular lookup into this texture allows the texture hardware to be used to compute the nearest texel of the nearest mip level that corresponds to the virtual texture page for the desired LOD at a given virtual address. Instead of storing a scale and bias, each 2-byte texel of this texture contains the (x,y) coordinates of the physical page to be used for the virtual page. A texel of this texture will point to a physical page from a coarser mip if the desired finer mip is not yet available. The second texture is a non-mip-mapped FP32x4 texture with one texel per *physical* page. A texel from this texture contains the scale and bias (ST-scale, S-bias, T-bias) necessary to map a virtual texture coordinate to a physical texture coordinate for that page. The implementation in a fragment program of the virtual to physical translation using a page table and a separate mapping texture to store the scale and bias is shown in appendix A.2. This approach saves memory by storing the floating-point scale and bias in a much smaller (typically 32 x 32) texture with one texel per *physical* page, while allowing access to this texture at the cost of a fully resident much larger (typically 1024 x 1024) texture with only two bytes per *virtual* page. This memory optimization costs the latency of a dependent texture read, but it saves 8x the memory compared to storing the floating-point scale and bias in a texture with one texel per virtual page. With only 2 bytes per virtual page and 16 bytes per physical page, the page table textures consume about 2.66 MB of memory for a virtual texture with 1024 x 1024 pages.

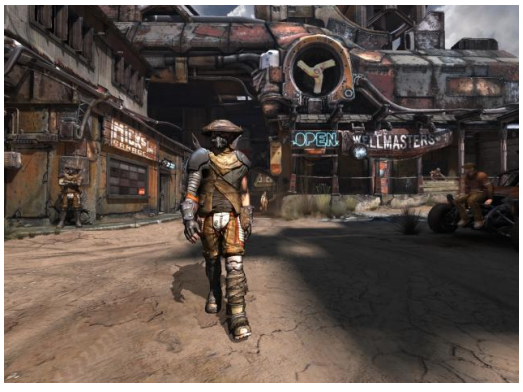


Figure 4: A scene with virtual textures applied to all geometry.

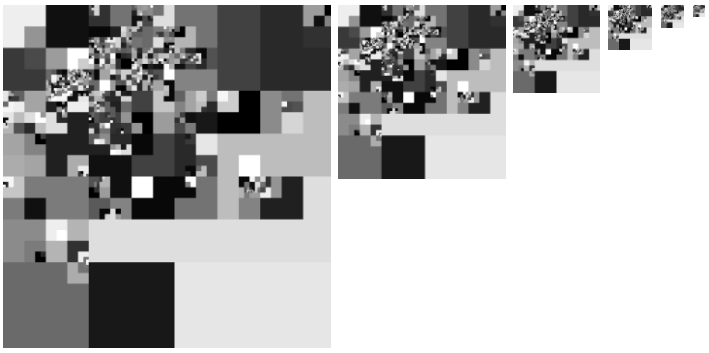


Figure 5: Page table with 2 bytes per virtual page of the virtual texture applied to static geometry in the scene.

On some graphics hardware fetching from a texture with more than 32 bits per texel is expensive and/or limits the number of fragments that can be processed in parallel. Instead of using a single FP32x4 mapping texture, the scale and bias can be stored in three separate single-component

FP32 textures: one for the ST-scale, one for the S-bias and one for the T-bias. The implementation in a fragment program using three FP32x1 textures to store the scale and bias is shown in appendix A.3.

On some platforms the performance is limited by the number of textures used. Using three separate single-component FP32 textures may therefore not result in optimal performance. To reduce the number of bound textures, the scale and bias can be stored in two 16-bit per component fixed-point textures: one single-component texture to store the ST-scale and one two-component texture to store the S-bias and T-bias. Some care is necessary to make good use of all the available bits in order to allow virtual to physical translations from very large virtual textures. The implementation using two 16-bit fixed-point mapping textures is shown in appendix A.4. The virtual to physical translation no longer consists of just texture lookups and a multiply-add to apply the scale and bias. Two additional multiplications and a subtract are necessary to first move the 16-bit fixed-point values into the appropriate range.

Instead of storing a scale and bias in textures, the virtual to physical mapping can also be calculated in a fragment program based on the coordinates and mip level of a physical page. The coordinates and mip level of a physical page can be stored in a single mip-mapped texture which avoids the latency of a dependent texture read to fetch a scale and bias. Here also, a regular lookup into this texture allows the texture hardware to be used to compute the nearest texel of the nearest mip level that corresponds to the virtual texture page for the desired LOD at a given virtual address. The coordinates of the physical page retrieved from this texture are used to calculate the offset to the top-left corner of the physical page in the physical texture. To calculate the complete physical address, the offset within the virtual page needs to be scaled and added to the top-left corner of the physical page. This offset within the virtual page needs to be calculated for the correct mip level which is done by first multiplying the virtual texture coordinates with the width in pages of the physical page's mip level (virtual pages wide / 2^mip) and then taking the fraction. This fraction is scaled into the correct range before being added to the physical page offset.

When using an 8-bit per component RGBA page table texture the physical page coordinates can be stored in the first two components. Instead of storing the mip level in one of the other components, the 16-bits of the last two components combined can be used to store the width in pages of the physical page's mip level. The 16 bits are enough to support very large virtual textures of 1024 x 1024 pages with over 10 mip levels. The implementation in a fragment program using an 8-bit per component RGBA texture is shown in appendix A.5. The 8-bit components are converted to the floating-point range [0, 1] in hardware before they become available in the fragment program. Unfortunately this conversion may be hardware specific. On some graphics hardware the 8-bit components are directly converted to FP32 through a division by 255. On other hardware the 8-bit components are first converted to FP16 and then to FP32. The calculation of the virtual to physical translation in the fragment program has to account for any hardware specific conversion of the texture data when converting the physical page coordinates and the width in pages of the physical page's mip level back to integer values. Using an 8-bit per component RGBA texture, the page table ends up consuming 5.33 MB of memory for a virtual texture with 1024 x 1024 pages.

To reduce the memory footprint the page table can be stored as a 5:6:5 RGB texture. The physical page coordinates are stored in the 5-bit components and the base-two logarithm of the width in pages of the physical page's mip level is stored in the 6-bit component. The exp2() function is used to calculate the actual width in pages of the physical page's mip level in the fragment program. The 5:6:5 values are converted to the floating-point range [0, 1] in hardware before they become available in the fragment program. Unfortunately, just like the 8-bit to floating-point conversion, this conversion is hardware specific as well. On some hardware, the 5:6:5 values are first converted to 8-bits per component by replicating the high order bits into the low order bits. For instance, for a 5-bit value all bits are first copied to the most significant bits of the 8-bit value and then the most significant 3 bits are replicated to the least significant 3 bits of the 8-bit value. On some hardware, the 8-bit values are then converted to FP16 through a division with 255 before being converted to FP32, while on other hardware the 8-bit values are directly converted to FP32. On yet other hardware, the 5:6:5 values are converted directly to floating-point through bit replication without first going to an 8-bit value. The fragment program for the virtual to physical translation needs to take all these hardware differences into account. To complicate things further the exp2() function is no more than an approximation on some hardware even when the exponent is an exact integer. For instance, exp2(10) may not be exactly 1024. As such, a tiny value is added to the exponent such that 2^exponent is always larger than 2^int(exponent) but never larger than 2^int(exponent) + 1. The floor() function is then used on the result of the exp2() function to get the exact integer power of two. The implementation in a fragment program using a 5:6:5 RGB page table can be found in appendix A.6.

The performance of these different implementations of the virtual to physical translation varies per platform. The implementation using a single 8-bit per component RGBA page table is one of the fastest approaches on all platforms because it does not involve a dependent texture read and uses a fair trade between memory bandwidth and compute. Unfortunately, page table sizes add up, in particular if there are multiple large virtual textures active at the same time. Alternatively, the 5:6:5 RGB page table uses half the memory and requires less memory bandwidth at the cost of several fragment program instructions. On platforms where compute is costly compared to memory bandwidth and latency, the 8:8 page table with separate mapping textures can deliver optimal performance.

The virtual to physical translation using a mip-mapped page table texture is a lot faster than using a quad-tree structure with a dependent read for each finer level of detail accessed. However, compared to storing only the quad-tree, page table updates are much more expensive when using a mip-mapped page table texture. Consider when the first page of a virtual texture is mapped: the entire page table texture must be populated with a single texel value. When the next finer page is mapped in, one quarter of the texels must be updated, and so on. Fortunately large page table updates happen infrequently.

Instead of using a quad-tree page table or a page table texture, a hash table can be used to provide a middle ground between access latency, memory footprint and compute. Only resident texture pages are stored in the hash table. A virtual page is found in the hash table with a hash key calculated from the mip level and (x,y) coordinates of the page. A good hash key function in combination with a small hash table that, for instance, is only twice as large as the number of

physical pages (say 2048 entries), typically results in very few collisions, allowing the lookup of most pages with a single memory access. The spatial index of a page in the virtual texture quad-tree, modulo the hash table size can be used as a hash key. However, better results are achieved if the (x,y) coordinates of a virtual page are first remapped within the mip level such that pages that are close to each other in the quad-tree do not map to the same hash table entry.

The hash table does not provide an automatic mechanism to fall back to a texture page from a coarser mip if a desired finer mip is not yet available in the pool of physical pages. Instead, when the desired page is not found in the hash table, the hash key for the next coarser page will have to be calculated in an attempt to fetch the next coarser page from the hash table. If the next coarser page is also not resident this process will have to be repeated until a valid page is found. On average, when most desired pages are resident, the hash table access latency is much better than a quad-tree page table. However, in the worst case, when few or no pages are resident, multiple hash keys have to be calculated and multiple memory accesses are required. The memory footprint of a hash table is small and only a small factor larger than a quad-tree page table. Even though a hash table provides an interesting middle ground between a quad-tree page table and a page table texture, on most hardware the direct access of a page table texture is preferred and with only 2 bytes per virtual page, the memory requirements of a page table texture are usually acceptable.

3.2 Texture Filtering

One of the unfortunate complexities of virtual textures without special graphics hardware support is that the texture unit, being unaware of the actual texture pages, cannot filter across page boundaries. Texture pages that are adjacent in virtual texture space do not necessarily map to physical pages that are next to each other, let alone close to each other in the physical texture. Instead of using the filter hardware, it is too costly to implement texture filtering completely in a fragment program.

It is possible to use bi-linear filtering while forcing edge texels to be equal at the expense of some fragment program instructions. However, addresses that are between physical pages cannot be used which means that such "clamped" pages really have an effective width that is one texel less than the physical page width, with a half-texel no-man's-land around the perimeter. These clamped pages also have a fundamental flaw in texel shifting at mip level transitions that show up as objectionable seams. In order to properly support hardware bi-linear filtering, each physical texture page must have a border of texels around it.

Implementations of virtual textures in software, without special graphics hardware support, are also not able to transparently support tri-linear filtering. A straightforward way to allow for hardware accelerated tri-linear filtering is to store one mip level for the texture with physical pages but this comes at the expense of a 25% increase in memory footprint and an increase in compute and bandwidth to populate this mip level. When using a block compressed physical texture there is also not necessarily a smooth transition from the coarser virtual mip to the coarser physical mip. For instance, if texture pages have a 4-texel border then the first mip level of the physical texture is populated with pages that are half the size with a 2-texel border and a 2-texel offset which results in different block compression artifacts compared to the virtual mip.

Another way to implement tri-linear filtering is to access two virtual pages during rendering, determining the LOD fraction between them and computing the weighted average. Even with a mip-mapped page table texture implementation, the cost of a single virtual to physical translation carries significant overhead. It is interesting to note however, that some of the overhead of a second translation is hidden behind the cost of the first one. Nevertheless tri-linear filtering using two virtual to physical translations is typically still fairly expensive.

To improve the visual fidelity, anisotropic filtering tends to be more important than tri-linear filtering. Hardware accelerated anisotropic filtering can be supported if the page border is wider than 1 texel. As such, a 4-texel border is typically used around each physical page. In other words, if a page is 128 x 128 texels then the payload is 120 x 120 texels surrounded by a 4-texel border with texels that are replicated from adjacent virtual pages. The 4-texel border maps well to the 4x4 block size of a hardware supported compression format (DXT, S3TC, ETC, etc.) and allows for reasonable quality anisotropic filtering with a maximum anisotropy up to 8.

Ideally, the *virtual* texture coordinate is used to compute the anisotropic footprint and the physical pages are sampled with a texture fetch that explicitly specifies the derivatives. This requires scaling the derivatives to factor in the different scales that texture coordinates will have when they come from different mip levels. The scale factor is the ratio between the size of the virtual mip level and the size of the physical texture. When using a virtual to physical translation with one or more mapping textures, this scale factor has to be stored separately using an additional texture component. In the case of a FP32x4 mapping texture, the scale factor can be stored in the unused component. When using three FP32x1 or two 16-bit fixed-point mapping textures, an additional texture or texture component has to be added to store the scale factor. No additional data needs to be stored when the virtual to physical mapping is calculated in the fragment program using either an 8:8:8:8 or 5:6:5 page table texture. In these cases, the sale factor can be calculated in the fragment program as shown in appendix A.5 and A.6.

Calculating and scaling the derivatives adds fragment program complexity and on various hardware a texture fetch with explicit derivatives is more expensive which may make this solution unattractive from a performance standpoint. Instead, hardware accelerated anisotropy on the *physical* texture coordinate with implicitly computed derivatives can be used. This results in erroneous anisotropic footprints for quad-fragments that cross virtual page boundaries because the physical texture space is discontinuous at page boundaries. Texture pages that are adjacent in virtual texture space do not necessarily map to physical pages that are next to each other, let alone close to each other. However, even though for virtual page crossings the derivatives may become way too large with an arbitrary direction, the anisotropic footprint is still bounded to a single physical texture page when the maximum anisotropy is less-equal to twice the border width. The erroneous footprints at page boundaries are a reasonable performance vs. quality trade-off on most hardware. While the artifacts increase with the maximum anisotropy, even with higher anisotropy settings the quality is actually surprisingly good and the erroneous footprints are only noticeable under significant magnification.

Unfortunately, when using hardware accelerated anisotropy on the *physical* texture coordinate on older ATI/AMD hardware, the sample positions at virtual page crossings chosen by the hardware are outside the page boundaries and produce objectionable artifacts despite a maximum

anisotropy setting less-equal to twice the page border width. On this hardware, anisotropic filtering with a non-mip-mapped physical texture walks through texture space assuming an ideal mip level instead of the actual mip level and as a result the steps can be way too large for the case of virtual page crossings. Fortunately, on most ATI/AMD hardware explicitly calculating the derivatives, scaling them and using a texture fetch with explicit derivatives has good performance and this is the preferred method anyway.

Normally, when fetching texture data from a mip-mapped texture, the anisotropic footprint is sampled using texels from multiple mip levels. Even when an additional mip level is provided for the physical texture to allow tri-linear filtering, the page table texture is point-sampled using a regular texture lookup unaware of the anisotropic texture fetch that follows. The page table texture must be point-sampled to retrieve individual page mappings without mangling the data by blending between adjacent but independent page mappings. Using the texture hardware to point-sample a page table texture does not necessarily result in a mapping to a physical texture page with the appropriate texture detail for the anisotropic texture fetch that follows. Without any adjustments, the page table lookup returns a mapping to a physical texture page that is typically too coarse and provides too little detail. To provide additional texture detail for the anisotropic texture fetch, the page table lookup can be biased with the negative base-two logarithm of the maximum anisotropy. This allows the anisotropic texture fetch to work with additional texture detail on surfaces at an oblique angle to the viewer where the sampled footprint is maximized (anisotropic). However, this can cause noticeable shimmering or aliasing on surfaces that are orthogonal to the view direction where the sampled footprint is minimal (isotropic). To improve the quality, the texture LOD of the page table texture lookup can be calculated in the fragment program based on the anisotropy. The calculated LOD can then be explicitly passed to the page table texture lookup. The calculation of the texture LOD is shown below.

```
const float maxAniso = 4;
const float maxAnisoLog2 = log2( maxAniso );
const float virtPagesWide = 1024;
const float pageWidth = 128;
const float pageBorder = 4;
const float virtTexelsWide = virtPagesWide * ( pageWidth - 2 * pageBorder );

float2 texcoords = virtCoords.xy * virtTexelsWide;

float2 dx = ddx( texcoords );
float2 dy = ddy( texcoords );

float px = dot( dx, dx );
float py = dot( dy, dy );

float maxLod = 0.5 * log2( max( px, py ) );  // log2(sqrt()) = 0.5*log2()
float minLod = 0.5 * log2( min( px, py ) );

float anisoLOD = maxLod - min( maxLod - minLod, maxAnisoLog2 );
```

Obviously calculating the texture LOD adds significant computational complexity to the fragment program. It is interesting to note, however, that compared to using a constant LOD bias, some performance is gained back due to better texture cache usage. For surfaces that are mostly orthogonal to the view direction, the calculated LOD causes a mip level to be selected

where the texture samples are closer to each other. Nevertheless selecting a mip level based on the anisotropic footprint may be unattractive from a performance standpoint because of the additional fragment program complexity. Instead, using a maximum anisotropy of 4 and a page table texture lookup biased with a constant negative 2 typically results in a reasonable trade between quality and performance where surfaces at an oblique angle to the viewer are significantly sharper while the shimmering or aliasing that appears on surfaces orthogonal to the view direction is usually not objectionable.

3.3 Rendering from Multiple Virtual Addresses

There is often a desire to supply different texture data for the same model to customize it or to indicate damage, teams, etc. In conventional API usage, one would do this with a separate set of skin textures or a texture array containing all possible skins. With virtual textures, skins can allocate their own unique space within a single large virtual texture, and supply a base texture (s,t) address per skin that is used to offset the virtual texture coordinates in a vertex program. Damage skins require multiple virtual to physical translations per fragment in order to blend from multiple texture sources. This turns out to be less expensive than one might expect because the cost of the second translation hides nicely behind the first. Modest use of two virtual to physical translations per fragment can be very effective to display localized damage while maintaining performance.

3.4 Feedback Rendering

While a sparse representation makes it possible to render with a partially resident texture, feedback is necessary for determining which parts of the texture need to be resident. Texture feedback is rendered to a separate buffer that, for the virtual texture pages used in the current scene, stores the virtual page coordinates (x,y), desired mip level, and virtual texture ID (to allow multiple virtual textures). This information is then used to make those texture pages resident that are needed to render the scene. The feedback can be rendered in a separate rendering pass or to an additional render target during an existing rendering pass. An advantage of rendering the feedback is that the feedback is properly depth tested, so the virtual texture pipeline is not stressed with requests for texture pages that are ultimately invisible. When a separate rendering pass is used it is fine for the feedback to be rendered at a significantly lower resolution (say 10x smaller). A fragment program that can be used to render to a feedback buffer is shown in appendix B.

Figure 6: Scene with virtual textures applied to all geometry.

Figure 7: Feedback buffer for the same scene.

Only the texture coordinates and not the actual texture data are used in the feedback rendering pass which means that alpha tested and transparent surfaces are considered completely opaque. As a matter of fact it is generally not possible to use alpha tested or transparent virtual texture data to generate feedback because this texture data may not be resident and proper feedback has to be generated first before such texture data becomes resident.

To properly pull in texture data that is visible through an alpha tested or transparent surface, any surfaces that are not completely opaque could be rendered randomly every so many frames to the feedback buffer. Similarly, when a surface uses multiple virtual texture sources, these sources could be alternated every render frame such that over time all the necessary texture data is pulled in. Unfortunately, this has the tendency to destabilize the virtual texture pipeline because a different set of texture pages is requested every frame even when the scene does not change. The pages that are requested one frame may end up replacing the pages that were requested for the same surface the previous frame. As a result, the system may never stabilize and pages may be continuously replaced without ever pulling in the highest detail texture data necessary for rendering the scene.

When a surface uses multiple virtual texture sources, one solution is to alternate the different texture sources in screen space, where every other pixel of the feedback buffer pulls texture data from a different source. When rendering from two different sources, this results in a simple checkerboard pattern but more complex patterns can be used when rendering from more than two sources. This approach may increase the chance of undersampling the feedback when a surface is very small and covers very few pixels of a relatively small feedback buffer, but this turns out not to be a noticeable problem in practice.

A similar approach can be used for alpha tested or transparent surfaces where every other pixel of the feedback buffer covered by such a surface is considered either fully transparent or fully opaque. When a simple checkerboard pattern is used to alternate between fully transparent and fully opaque, not all the texture data may be pulled in for a scene with multiple layers of alpha tested or transparent surfaces stacked on top of each other. Using more complex patterns and a different pattern per alpha tested or transparent surface can alleviate this problem. This is similar

to rendering with screen-door transparency or alpha-to-coverage which on some hardware could actually be used for this purpose if the 'alpha channel' of the feedback buffer is not used to store feedback data.

On newer graphics hardware that supports atomic operations it is possible to implement per pixel linked lists. Such linked lists are also known as structured append and consume buffers. Being able to create linked lists allows an arbitrary number of data elements to be tracked per pixel. Virtual texture feedback data for each layer of transparency or each different virtual texture source used per surface can simply be appended to the chain of data stored per pixel. At the cost of some complexity, this approach reduces the chance of undersampling the feedback when an alpha-tested or transparent surface is very small and covers very few pixels of a relatively small feedback buffer.

The results of the feedback rendering pass are analyzed in a separate process. This process could stall and wait for the feedback render but it is typically fine to use a frame old data and incur a frame of latency. The feedback analysis walks the screen buffer and condenses the page information into a list with unique pages. Effectively, the feedback analysis creates the quad-tree that represents the mip-map hierarchy with all the pages that need to be resident to properly render the current scene. The analysis process sorts the pages on priority. First, the priority is set such that the farther away the desired mip level is from the actual resident mip level, the higher the priority. Second, the priority increases as the number of samples for a particular page increases in the feedback buffer. The virtual texture system uses the sorted pages to maintain residency of already resident visible pages and to first stream in the non-resident pages that will most improve the visual quality of the currently rendered scene.

3.5 Oversubscription

In any system with virtual storage, the condition can arise where physical storage is insufficient to hold the working set, and thrashing occurs. This is a case where a virtual texture system can cope without losing performance. The number of resident pages that were seen in the previous frame's feedback is tracked. If that number is greater than a high water mark, the system is considered oversubscribed and the LOD bias used when generating feedback, is incremented. If the number is less than a low water mark, the system is considered undersubscribed and the LOD bias used when generating feedback is decremented. The dynamic feedback LOD bias is always clamped to avoid it ever becoming negative. This mechanism backs off of detail, without thrashing, for views where enough detail cannot be supplied, but then adds the detail back as soon as the system is not strained.

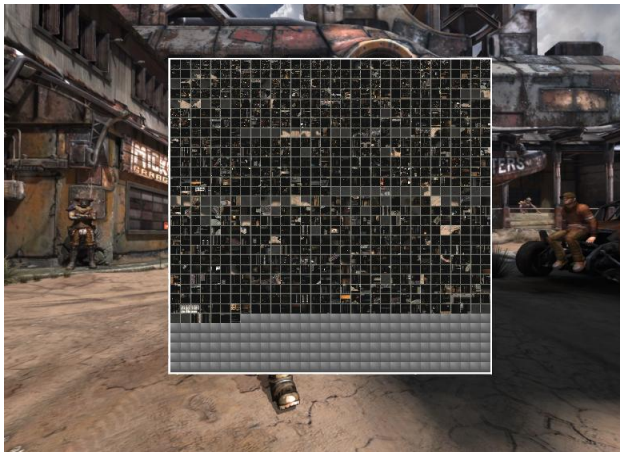


Figure 8: Physical texture with 32 x 32 pages.
Undersubscribed system but already at full resolution.

Figure 9: Physical texture with 8 x 8 pages.
Oversubscribed system with everything blurry.

When rendering from mip-mapped textures there should, in theory, not be a need for more texels than about 4 times the screen resolution. However, even when there are enough physical pages to hold more than 4 times the screen resolution (taking page borders into account), the system may still end up in an oversubscribed state. The reason for this is a combination of texture page sizes and the way geometry is unwrapped onto the virtual texture. Objects that are relatively small or thin in texture space and have multiple sides that face in different directions can cause significant overhead when all sides are unwrapped onto the same texture page. Only a few sides are ever visible at the same time while such objects require whole texture pages to be pulled in where most of the texels on the pages may not be visible. Using smaller texture pages may reduce the number of texels that are pulled in but not needed. However, smaller pages waste relatively more space for borders and more memory is needed for larger page tables. Using pages of 128 x 128 texels is usually a reasonable trade.

3.6 LOD Snapping / Texture Popping

It will happen sometimes that, despite all efforts, a significant latency will be incurred between the time that a texture page is needed and the time that the data for it is available. This can result in an unpleasant "pop" when the desired LOD for a page is off by more than one level and the right LOD suddenly becomes available. The straightforward approach, if tri-linear filtering with two virtual to physical translations were employed, would be to include some delay in the transition from coarser to finer mip level when the desired level is not sufficiently close to the currently displayed level. The delay could be encoded into the page table or mapping textures. When bi-linear or anisotropic filtering with a single mip level is used, a different approach is required. Instead of using tri-linear filtering during rendering, the physical pages can be continuously updated with some sort of blend between the coarser and finer mip.

Waiting until the finer mip level becomes available and phasing it in by starting with an upsampled version of the coarser mip level seems like a plausible solution. However, this does not completely remove texture popping because the sample positions change suddenly when the upsampled image is mapped. It is not possible to create a texture that is twice the resolution of another texture that when both rendered with bi-linear magnification between texel centers

produce the exact same bi-linear filter patterns. Bi-linear magnification effectively creates a triangle wave pattern between texel centers that cannot be replicated with a texture that is twice the resolution and also rendered with bi-linear magnification. The texture of twice the resolution typically shows a pattern with the triangle wave tops and bottoms cut off or some other approximation. The result is a visible "pop" from the bi-linearly magnified coarser mip to the finer mip created through upsampling the coarser mip.

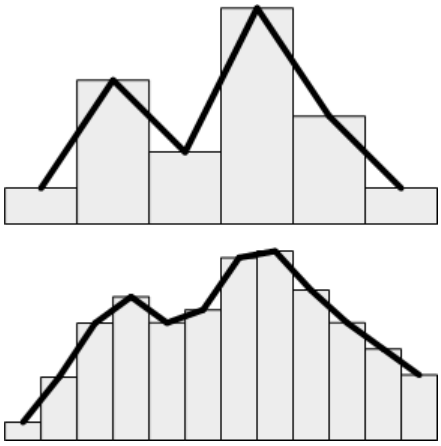


Figure 10: Bi-linear filtering between texel centers at resolution N and 2x N.

The better solution is to create a physical page for the finer mip from a coarser mip through upsampling *before* the coarser mip is magnified. The finer mip created from the coarser mip through upsampling will come in gradually as the viewpoint approaches the textured surface. Without visibly popping, the finer mip that was created from the coarser mip through upsampling can then be blended towards the actual finer mip when it becomes available. This will not have the abrupt transition because only the texel colors change, not the location of the samples.

Various interesting algorithms can be used to create a finer mip level by upsampling a coarser mip. For instance, a straightforward bi-cubic filter can be used. However, more complicated edge enhancing filters can be used as well. Creating finer mips through upsampling as soon as they are desired for rendering can also be used to create interesting detail if there is no original texture content available [35].

4. Storage & Streaming

4.1 Storage & Compression

To accommodate for universally applied virtual textures, 3 physical page textures are used that store the same physical pages but different data. These 3 textures store a total of 10 channels per texel. There is one texture that stores a diffuse map using the YCoCg color space which is converted back to RGB during rendering in the fragment program. There is also one texture that stores the X and Y components of a tangent space normal map where the Z component of a normal is derived in the fragment program during rendering. A power map is stored in one of the channels of the same texture. Finally, there is one texture that stores a specular map, either as monochrome or RGB color. This texture also stores a 1-bit alpha channel (cover mask).

The physical textures in video memory can be stored uncompressed or DXT compressed [30, 31]. DXT compression is desired to reduce video memory usage and bandwidth requirements during rendering. Three DXT compressed textures are used to store the physical page textures: one DXT1 for the specular map with alpha [32], one DXT5 for the YCoCg diffuse map [33], and one DXT5 for the tangent space normal map [34] and the power map. Uncompressed, each physical page consumes 192 kB, while compressed to DXT format a physical page takes up 40 kB.

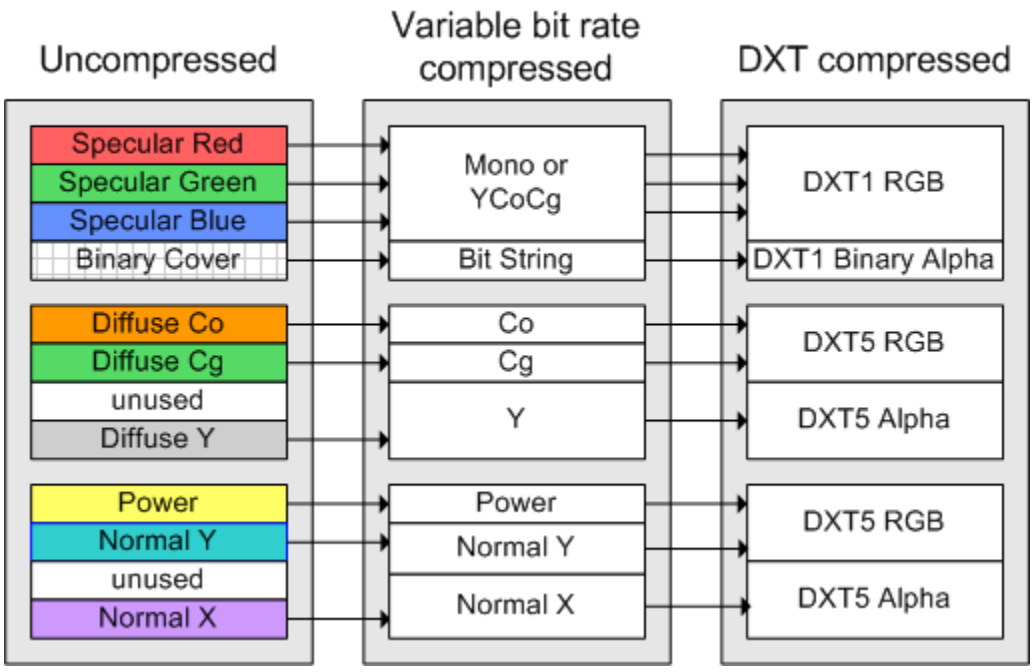


Figure 11: Virtual texture channel layout.

Pages on disk can be stored uncompressed, DXT compressed or in a variable bit rate compression format such as a JPEG-like DCT based format [27] or HD-Photo [28] / JPEG-XR [29]. High compression ratios are necessary to be able to store very large uniquely textured worlds on the limited storage space available on optical disks like a DVD. High compression

ratios are also desired to reduce the bandwidth requirements for streaming the texture data off of slow storage devices. As such, a variable bit rate compression format is used to store the texture pages on disk which reduces most texture pages to between 1 and 6 kB.

With a 4-texel inset border for texture filtering and a physical page size of 128 x 128 texels this leaves only 120 x 120 texels worth of payload per page. As such, the border required for proper texture filtering winds up costing approximately 12% in additional texels. The border texels need not be stored on disk, but as a practical matter, it is far less complicated to have pages be fully independent and actually store the additional 12% texels on disk.

4.2 Improving Compression Ratios

Before the texture data is compressed to disk, it can be processed in several ways to improve the compression ratio. For instance, the diffuse map may be sub-sampled to 4:2:0 YCoCg based on the compression algorithm used. Similarly the specular map can be either converted to monochrome if there is no significant color information, or the YCoCg color space can be used with 4:2:0 sub-sampling.

Additional savings are possible when using a pre-computed lighting solution with an approximation for real-time specular reflection. For instance, an off-line radiosity based global illumination algorithm can be used to calculate the static lighting in the environment where only the specular reflections are calculated real-time using some approximation. For such an approximation to look right, the specular map is downscaled by the (off-line calculated) shadow to reduce specular reflections in shadowed areas. This reduces the dynamic range of the specular map in shadowed areas and improves the compression ratio. Sometimes specular map texel values can be omitted completely. If the incoming light at a unique texel in the environment is almost the same from all directions (same color and intensity) then a constant specular contribution can be moved to the pre-lit "diffuse" map and the specular values for the texel need not be stored.

When pre-computed lighting is used, the normal X and Y components can also be scaled with the specular intensity to improve the compression ratio by reducing the dynamic range. After all, when using pre-computed lighting the surface bumpiness is, for the most part, already baked into the diffuse map, and otherwise only visible when there are dynamic specular reflections. Reducing the magnitude of the X and Y components causes the run-time derived Z component to increase in magnitude which means the tangent space normal will converge to pointing straight out of a surface such that the perceived bumpiness decreases as the specular reflection decreases. Reducing the perceived bumpiness based on the specular lighting comes at a slight loss in quality but the loss in quality is usually small while the storage savings are significant.

4.3 Visibility-based Optimizations

Large virtual textures reduce the cost of unique texture data but in a very large virtual world there will often be many areas that are inaccessible to the viewer, or that can be viewed only from a distance. In particular in a computer game there will be many surfaces that can never be seen up close. For this reason it can be very useful to perform an off-line brute force analysis of

scene visibility, capturing omni-directional virtual texture feedback data and marking can-be-seen data for all possible viewer positions. In a computer game this brute force visibility determination can, for instance, be based on a sampling of floor surfaces of the player's navigation space (configuration space).

First of all it is useful to capture per-texel visibility. To reduce storage requirements the visibility can be tracked as a bit per 4x4 texels in the virtual texture. This visibility information can then be used to eliminate whole pages that can never be seen. In other words, a virtual texture is not only sparsely resident in video memory, the virtual texture may also be sparsely populated with source data. The visibility information can furthermore be used to blur out invisible portions of pages in order to improve their compression.

Secondly, data can be collected to optimize texel area allocation for bits of geometry that are contiguous in texture space, often called "charts" in an "atlas" in literature. Instead of capturing per-texel visibility, the chart ID, number of samples seen, minimum derivative, minimum ratio and maximum ratio of the derivatives are recorded. This information is then used to scale objects in texture space such that all objects take up a reasonable amount of space on the virtual texture, not just based on the number of texels per world unit but also based on visibility. This not only avoids having a soda can with a 20x higher texel density than the counter that it sits on, but also allows scaling down charts on the virtual texture for objects that can never be seen close up. When the very large texture atlas is first built, there is no visibility-based information available to determine how much texture space charts should take up, and some estimates based on the world-space area are used for each chart. After finest-LOD data is collected on a per-chart basis, the atlas is re-optimized based on the additional information.

4.4 Layout and Locality

The layout of texture data on virtual textures is of critical importance for a real-time application, because seek times of optical disks are on the order of 100+ ms and similar latencies are to be expected when streaming texture data over the Internet. On systems with a local hard disk, the danger of going without needed data for long periods of time can be dramatically reduced. Unfortunately, not all systems have a hard disk, or a limited amount of hard disk space is available to cache texture data.

There are several metrics that are important when optimizing the layout of virtual textures. It is important to have a consistent texel density such that virtual texture space is not wasted for LOD texels that are never visible. As described in the previous section, the texel allocation of geometry "charts" can be optimized based on brute-force collected visibility information.

The charts are then placed on the virtual texture such that the least amount of space is wasted. The charts may have odd shapes that only fit well together in certain relative positions. The orientation of charts can also be changed to where they fit better. However, when virtual addressing is performed with only a scale and bias this requires changing the texture coordinates on the geometry to account for orientation changes. Rotating charts also makes the optimization process significantly more complicated.

While minimizing the amount of wasted space is important, it is also important to optimize the locality of "charts" on the virtual texture such that any rendered scene uses the least number of texture pages. This is a hard optimization problem and sometimes conflicting with minimizing the wasted space on a virtual texture. Optimizing for both locality and the least amount of wasted space is a combination of a mesh parameterization and a bin-packing problem which is NP-hard. A reasonable solution can be found by first spatially sorting the charts with 3D Binary Space Partitioning and then walking the tree breath first while allocating 2D texture space with a space filling curve for 2D locality.

Once all charts are optimally placed on a virtual texture, it is important to optimize for locality of texture pages on optical disks such that any rendered scene requires the least amount of seeking when streaming in the required texture data. Previous work [26] suggests laying out pages on disk in quad-tree order possibly interleaving multiple mip levels. This works well for a mostly flat contiguous terrain. However, the situation is vastly more complex for virtual textures that are applied to arbitrary 3D environments.

4.5 Streaming and Caching

Caches can be used to significantly reduce the latency when streaming frequently used texture pages. When used to store recently seen texture pages, a cache in system memory is very effective in reducing the latency for updating physical pages with texture data from previously viewed areas, for instance when the view suddenly rotates back to where the viewer came from. When a texture page is streamed from an optical disk to system memory it can be written back to a hard disk cache such that future access is both faster and does not suffer as much from very long seek times. For high performance streaming, multiple software threads are typically used to retrieve texture pages from relatively slow storage devices like hard disks or optical disks, where a separate thread is implemented for each storage device.

With compression ratios in the 10:1 up to 20:1 range between variable bit rate compression and DXT fixed block compression, it is important to keep the data fully compressed for as long as possible to make optimal use of the available memory. Texture data is stored in 4 distinct locations: optical disk, hard disk, system memory, and physical page textures in video memory. The data remains compressed using a variable bit rate compression format in all the cache levels except the physical page textures.

Like all virtual memory systems, a virtual texture system can have memory areas that are "pinned down", i.e. texture pages that are always resident and cannot be replaced. For instance, the texture page that represents the coarsest mip level of a virtual texture is usually locked in the physical page textures to make sure there is always some texture data to fall back to when the view point suddenly changes to a scene with all new texture data.

5. Physical Page Updates

5.1 Pipeline Overview

Figure 12 shows an overview of the pipeline for physical page updates. Texture feedback for a scene is rendered to a small screen buffer. The color values of this screen buffer are used to store the virtual texture pages for the pixels of the currently rendered scene. From this buffer the feedback analysis creates a sorted list with texture pages that need to be resident. The sorted pages from the feedback analysis are used to first update the non-resident pages that will most improve the visual quality of the currently rendered scene. Residency is maintained for visible pages that are already present in the physical textures.

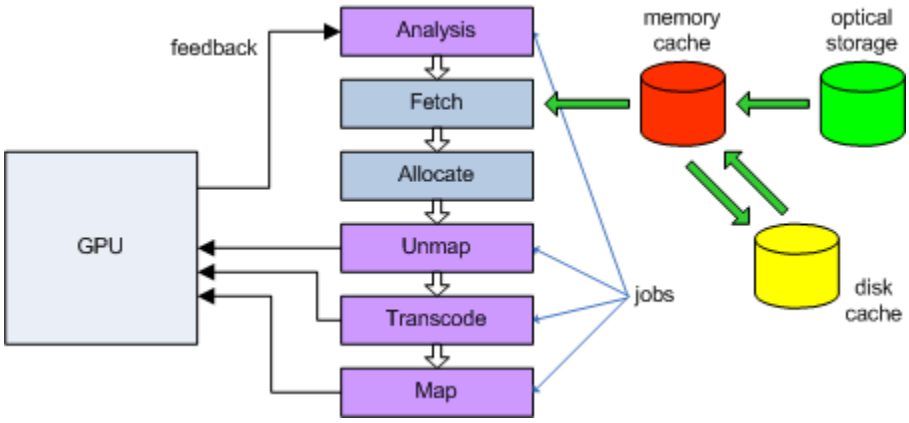


Figure 12: Physical page update pipeline.

For each sorted page, the system tries to fetch the compressed texture data from cache. The cache returns the compressed texture data in case of a cache hit and the cache schedules misses to load in the background. For a texture page that has compressed texture data available, the system allocates a new physical page. Before using the new physical page it is unmapped, to make sure the GPU stops using the page and drops down to a coarser parent page. The compressed texture page is then transcoded such that it can be used for rendering on the GPU. Only after transcoding the full texture page it is mapped such that the GPU will start rendering from it.

5.2 Replacement Policy

When a new physical page needs to be allocated to store a requested virtual page, all the physical pages are sorted based on priority and the physical page with the highest priority is used for replacement. The priority is first based on the LOD level of the page such that finer mips will be replaced first. This means that the quad-tree that represents the mip-hierarchy with resident pages gradually changes structure by removing and adding leaf nodes. Next, the page priority for replacement increases as the number of rendered frames increases since the page was last used. In other words, the finest mip level pages are replaced with a least recently used (LRU) replacement policy.

5.3 Transcode

Variable bit rate compression with very high compression ratios is used to store very large uniquely textured worlds on the limited storage afforded by optical disks, and also to reduce bandwidth requirements when streaming texture data from optical disks. Fixed bit rate DXT compressed textures are used on the GPU for their memory bandwidth and footprint benefits as well. When a new texture page is streamed in from disk or fetched from cache it has to be transcoded to DXT format from a JPEG-like format or from HD-Photo / JPEG-XR. This transcode process is a significant computational load and is performed in parallel through a parallel job processing system that offloads the jobs to whatever processors are available and suitable to the task. To improve load balance two jobs are added for each page that needs to be transcoded where each job transcodes part of the page data. During normal operation about 8 to 16 pages are transcoded per render frame which results in 16 to 32 jobs. When transcoding 16 pages per frame at 60 Hz with 128 x 128 texels per page, this results in a throughput of about 15 MegaTexels / second (MT/s). On systems with direct access to texture memory the transcode jobs can write the DXT compressed texture blocks directly to video memory.

5.4 Asynchronous Updates

On systems with direct access to texture memory, the paging system operates asynchronously to the GPU. Every frame, the system determines whether or not more texture pages need to be made available and the pages that will be reused are immediately unmapped. This causes the GPU to begin using the coarser parent page. The page's texels are not overwritten until the page has been unmapped and the new page is not mapped until all its texels are present. This process ensures that even though page tables and physical pages are updated continuously and asynchronously, the consistency of the virtual texture is maintained. There is the danger that the texture caches are not coherent with texture memory, but in practice, operations that flush or invalidate the texture cache occur often enough that erroneous behavior is never observed.

6. Results

6.1 Address Translation

The virtual to physical translation performance has been measured on several platforms using the fragment program shown in appendix C. For this performance test, a single screen aligned quad is rendered with 100 resident texture pages that cover the whole screen at a resolution of 1280 x 720.

Table 1: Virtual to physical translation performance.

page table / mapping texture formats	page table size for 1024x1024 pages	ATI Radeon X1900	NVIDIA GeForce 7800	NVIDIA GeForce 8800 Ultra
8:8 + FP32x4	2.66 MB	1.52 ms (1.20 ms)	1.30 ms	0.32 ms
8:8 + 3 x FP32x1	2.66 MB	1.35 ms (1.32 ms)	0.94 ms	0.37 ms
8:8 + UINT16x1 + UINT16x2	2.66 MB	1.44 ms (1.15 ms)	0.91 ms	0.32 ms
8:8:8:8	5.33 MB	1.08 ms (0.95 ms)	0.91 ms	0.26 ms
5:6:5	2.66 MB	1.08 ms (1.00 ms)	1.00 ms	0.29 ms

On older ATI/AMD hardware the *virtual* texture coordinate is used to compute the anisotropic footprint and a texture fetch with explicit derivatives is used. This is necessary to avoid objectionable artifacts that appear when hardware accelerated anisotropy on the *physical* texture coordinate with implicitly computed derivatives is used. The performance on ATI/AMD hardware using implicitly computed derivatives is shown between parentheses in table 1 for comparison.

6.2 Page Table Updates

A page table update consists of a partial page table texture update and possibly a single texel update in any mapping textures that are used. Updating mapping textures is no more than a trivial single texel write which is not very expensive. However, page table updates can be much more costly based on the virtual page that is being mapped. When a virtual page is mapped, the page table texel from the mip level that corresponds to the virtual page and its mip level is updated first. Next, power of two square areas of texels are updated in any finer mip levels of the page table texture that correspond to virtual pages that store finer image detail for the virtual page that is mapped. For instance when the very first coarsest virtual texture page is mapped in, all texels in the complete mip-map hierarchy of the page table are set to the same value. When the next finer page is mapped in, one quarter of the texels must be updated, and so on.

The performance of these updates is proportional to the number of texels written to the square page table texture areas which in return is related to the mip level of the virtual page that is mapped. On systems with direct memory access the performance is only limited by the speed at which memory can be written. However, on systems where the only access to video memory is through an API like OpenGL or DirectX the page table updates face significant API overhead. Current APIs do not support sub-image "memset" operations, only sub-image data copies. In order to effectively "memset" a square texel area of a page table mip level, the same texel value

is first written many times to a block of main memory and this data is then transferred through the API to update a square texture area in video memory.

6.3 Feedback Analysis

The performance of the feedback analysis is proportional to the size of the feedback buffer. A smaller feedback buffer improves the performance of the feedback analysis but it results in a coarser sampling of the texture data where virtual pages may not be immediately considered visible. In practice there are no noticeable visual anomalies from using a feedback buffer that is more than a 10x factor smaller than the resolution at which the scene is rendered.

The feedback analysis could be implemented on the GPU. However, there is usually enough of CPU time available, in particular on for instance the IBM Cell SPEs. On this platform and multi-core PCs, the feedback analysis only consumes in the order of 0.5 milliseconds worth of CPU time for a feedback buffer of 80x60 pixels. This performance is achieved by using a hash table to quickly find the parent of a page in the quad-tree that is built to extract an unique list of pages from the feedback buffer.

6.4 Storage

The texture pages used in the system are 128 x 128 texels with a 4-texel inset border and a 120 x 120 texel payload. In order to optimally use the available video memory, the texture pages with borders are a power of two as opposed to the payload being a power of two. On some hardware the maximum size of the physical textures is a power of two (like 4096 x 4096) and with a power of two page size of 128 x 128 an exact number of pages can be fit on the physical textures without wasting any space.

The physical texture size is typically 4096 x 4096 texels which is equal to 32 x 32 texture pages. When stored DXT compressed in video memory, the three 4096 x 4096 physical textures (storing 10 channels) together consume 40 MB. Support for larger physical textures (16k x 16k texels) is available on newer hardware while older hardware may not support textures larger than 4096 x 4096 texels. To increase the number of physical pages on older hardware, a 3D texture could be used. This would make the virtual to physical translation only slightly more complicated but a lot of hardware does not support anisotropic filtering on 3D textures which makes this solution unattractive. Instead, a cube map texture could be used but that would make the address translation significantly more complicated. The easiest way to increase the number of physical pages is to create multiple separate physical page pools in the form of multiple sets of physical textures. However, this requires different geometry to allocate pages from different pools and not every rendered scene may end up with an even balance of geometry such that all pools fill up equally.

The virtual to physical translations that use bytes to store the physical page coordinates allow for a maximum of 256 x 256 physical pages which is equivalent to a physical texture of 32k x 32k texels when the texture page size is 128 x 128 texels. The virtual to physical translation using a 5:6:5 page table can only support up to 32 x 32 physical pages which accumulates to a physical texture size of 4096 x 4096 when the texture page size is 128 x 128 texels.

Virtual textures on the other hand can be as large as 240k x 240k texels (where 240k = 2048 * payload size) in which case a 2-byte or 5:6:5 page table texture consumes 10.66 MB of memory. Instead of the maximum size, 120k x 120k virtual textures are typically used to reduce the page table texture memory. For a 120k x 120k virtual texture the page table texture consumes 2.66 MB of memory.

In the computer game RAGE most environments use a single 120k x 120k virtual texture for all static geometry and another 120k x 120k virtual texture for the dynamic objects in the environment. Some very large outdoor environments use multiple virtual textures for the static geometry. Each virtual texture has its own page table but multiple virtual textures can map to the same pool of physical pages but they can also map to separate pools of physical pages.

An uncompressed fully populated 120k x 120k virtual texture with 12 channels per texel (of which 2 unused) requires 225 GB of storage without borders and 256 GB with borders. Compressed to DXT format such a virtual texture still takes up 53 GB. For most static environments in the computer game RAGE, the 120k x 120k virtual texture is not fully populated. A lot of the finest mip texture pages are never visible and need not be stored. Furthermore, any part of a page that is never visible at full resolution can be blurred to significantly improve the variable bit rate compression ratio. Table 2 shows the sizes of the virtual textures used for the static geometry in several environments of the computer game RAGE.

Table 2: Virtual texture sizes (10 components per texel).

environment	# pages	uncompressed	DXT	JPEG-like	HD-Photo
Wellspring	112102	21023 MB (96 b/t, 192 kB/p)	4379 MB (20 b/t, 40 kB/p)	334 MB (1.5 b/t, 3.1 kB/p)	170 MB (0.8 b/t, 1.6 kB/p)
Bash TV	115879	21731 MB (96 b/t, 192 kB/p)	4526 MB (20 b/t, 40 kB/p)	358 MB (1.6 b/t, 3.2 kB/p)	205 MB (0.9 b/t, 1.8 kB/p)
RC-Bomb Base	244357	45826 MB (96 b/t, 192 kB/p)	9545 MB (20 b/t, 40 kB/p)	777 MB (1.6 b/t, 3.2 kB/p)	416 MB (0.9 b/t, 1.8 kB/p)

In table 2, the average bits/texel (b/t) and average page size in kiloBytes (kB/p) is shown between parentheses.

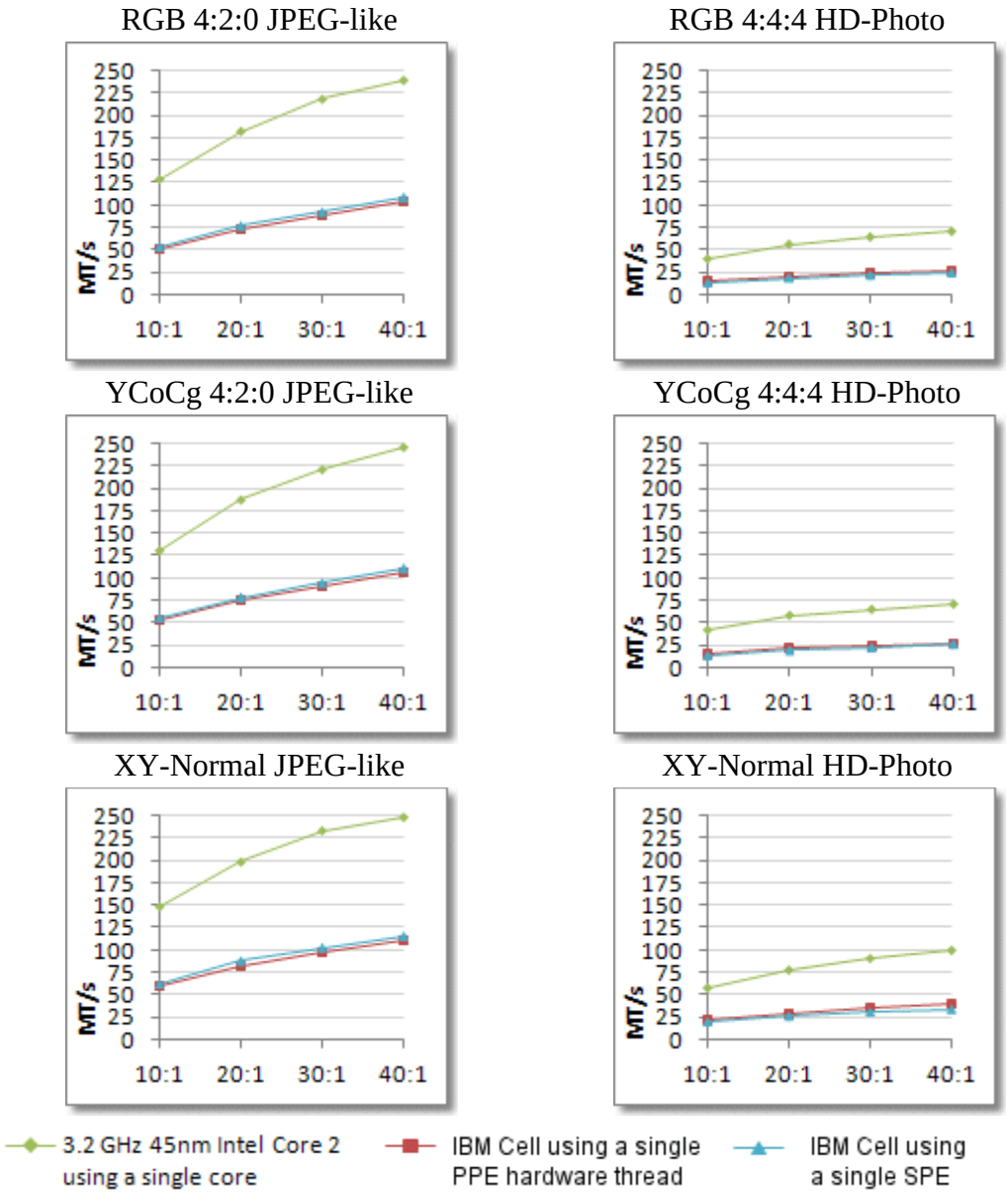
6.5 Transcode

Transcoding texture pages is a significant computational load and is performed in parallel through a parallel job processing system that offloads the jobs to whatever processors are available and suitable to the task. On newer graphics hardware the texture page transcoding can be implemented to run on the GPU. On systems with few CPU cores this can offload a significant computational load to the GPU allowing the CPU cores to be used for other purposes. However, older graphics hardware may not have the necessary capabilities to implement transcoding, or the GPU is already fully occupied with rendering and there is no time available for the additional computational load. On such systems the transcoding is implemented to run on the available CPU cores, like the hardware threads of the Cell PPE and the Cell SPEs. Table 3 shows the DXT compression performance on several different processors expressed in MegaTexels per second (MT/s).

Table 3: DXT compression performance.			
type	**3.2 GHz 45nm Intel Core 2 using a single core**	**3.2 GHz IBM Cell using a single PPE hardware thread**	**3.2 GHz IBM Cell using a single SPE**
DXT1 - RGB Specular + Binary Alpha	252 MT/s	160 MT/s	334 MT/s
DXT5 - YCoCg Diffuse	128 MT/s	75 MT/s	150 MT/s
DXT5 - XY-Normal + Power	203 MT/s	115 MT/s	240 MT/s

The following graphs show the variable bit rate decompression performance on several different processors expressed in MegaTexels per second (MT/s) for a range of compression ratios.

Variable bit rate decompression performance.

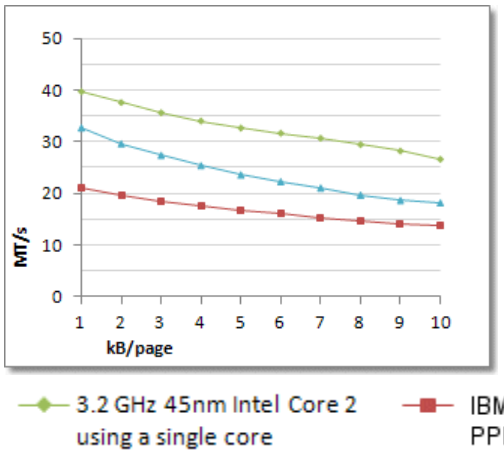


It is clear that the HD-Photo decompression is more costly than decompressing the JPEG-like format. However, HD-Photo generally allows the texture data to be compressed to almost half the size of JPEG-like data while maintaining equal fidelity and usually reducing or removing objectionable compression artifacts. As such, different compression ratios need to be considered when comparing the performance between HD-Photo and the JPEG-like format. For instance, it is reasonable to compare 20:1 JPEG-like compressed data with 40:1 HD-Photo compressed data. It is also important to note that 4:2:0 sub sampling is used for the JPEG-like compression of RGB and YCoCg data. Therefore, only half the data is pushed through the JPEG-like decoder as opposed to the 4:4:4 sampled data that is pushed through the HD-Photo decoder.

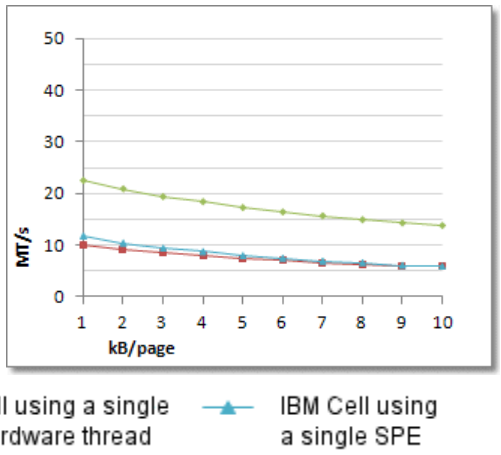
The following graphs show the complete transcode performance on several different processors.

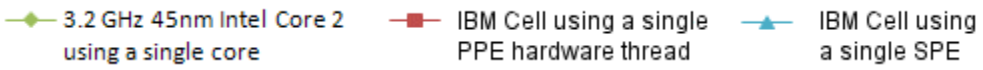
Transcode performance (10 components per texel).

JPEG-like -> DXT



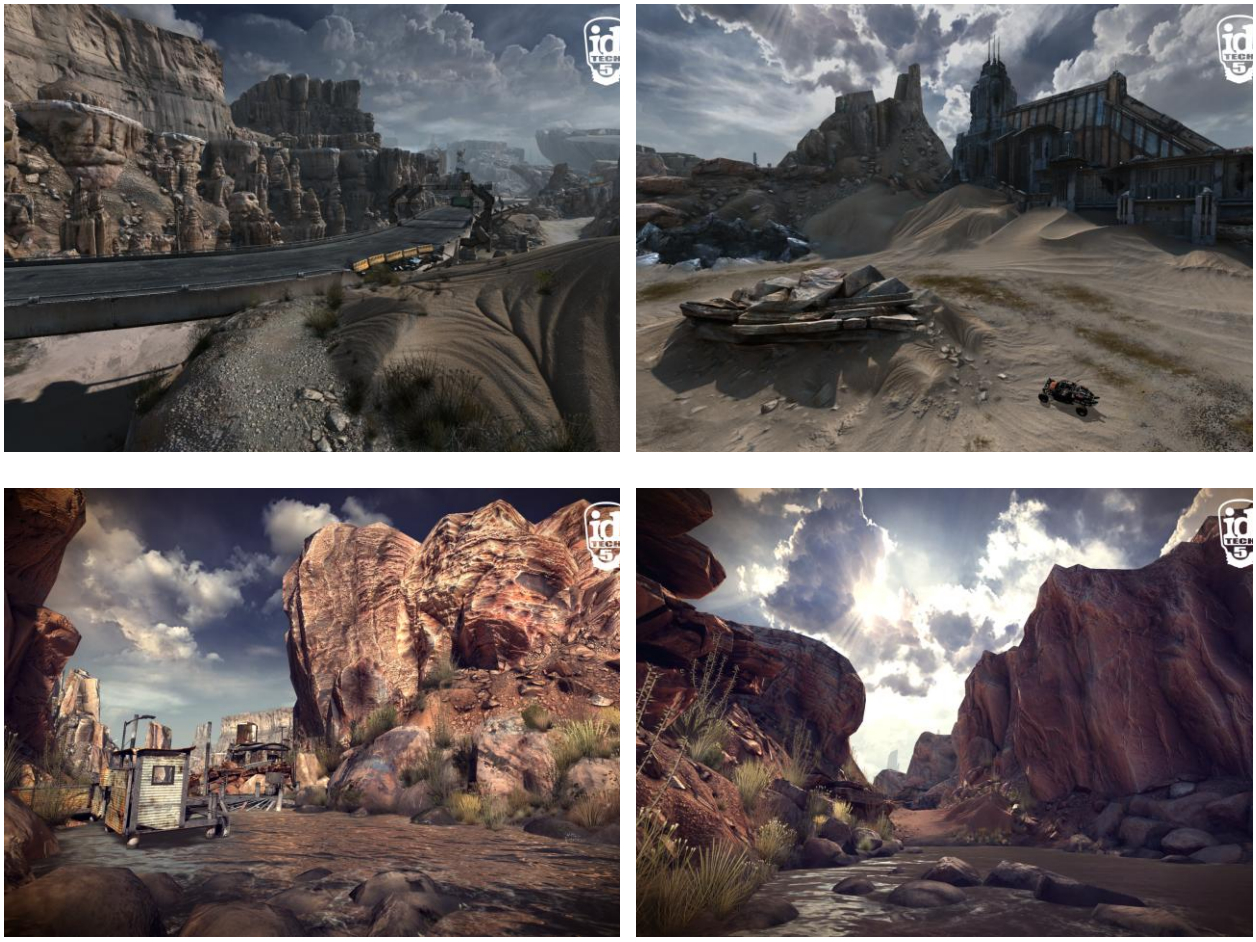
HD-Photo -> DXT





6.6 Visuals

A virtual texture system using the described solutions has been successfully implemented and employed in the computer game RAGE. This game runs at high frame rates (60 FPS) on a variety of graphics hardware as used on various platforms such as the Microsoft Windows PC, Apple Macintosh PC, Xbox 360, and the PlayStation 3. Below are some outdoor environment shots from the computer game RAGE.



Environment shots from RAGE.

7. Conclusion

Virtual textures differ from other forms of virtual memory because first, it is possible to fall back to slightly blurrier data without stalling execution, and second, lossy compression of the data is perfectly acceptable for most uses. Implementations of software virtual textures exploit these key differences between virtual textures and other forms of virtual memory to maintain performance and reduce memory requirements at the cost of quality. Implementing virtual textures without special hardware support is challenging, involves many subtleties, and inevitably comes down to finding the right trade between performance, memory requirements, and quality. The solutions presented here allow the implementation of very large virtual textures at high performance with very little loss in quality while requiring a minimal memory footprint. A virtual texture system using the described solutions has been successfully implemented and employed in the computer game RAGE which runs on a variety graphics hardware such as available on various platforms like the Microsoft Windows PC, Apple Macintosh PC, Xbox 360, and the PlayStation 3.

8. Future Directions

A reactive system for texture feedback works well because a virtual texture system is inherently latency tolerant. However, the separate feedback rendering pass comes at a cost and it is desirable to combine it with a normal rendering pass. When combining the feedback pass with a normal rendering pass it is important that the feedback is properly depth tested to avoid stressing the virtual texture pipeline with requests for texture pages that are ultimately invisible. If the rendering pipeline implements a separate explicit depth-only pass then it is worthwhile to consider writing out feedback with this depth pass. On many platforms a depth-only pass benefits from double speed depth writes and such a rendering pass usually ends up being geometry (vertex) limited. By writing out feedback with the depth pass, the depth will no longer be written at double speed and additional fragment cost is added to calculate the feedback such that the rendering pass may become fragment limited. Based on the amount of geometry that is rendered and the specific GPU performance characteristics this can be a positive trade because there is no longer a separate feedback pass that is almost always geometry (vertex) limited due to the small size of the feedback buffer. When using multiple render targets it is also possible to render depth, color and feedback in a single pass. This has the additional benefit of being able to use the anisotropic LOD calculation for both the feedback and the page table lookup.

Texture filtering is something that is implemented in fixed function hardware even on current and future programmable graphics hardware because performing texture filtering completely in software (i.e. a fragment program) comes at a significant cost. One of the unfortunate complexities of virtual textures without special graphics hardware support is that the texture unit, being unaware of the actual texture pages, cannot filter across page boundaries. If, however, the virtual to physical translation is implemented in hardware, proper texture filtering can be implemented in hardware by performing an address translation for each texel that is needed by the texture filter. This means that a hardware implementation uses 8 virtual to physical translations for a texture fetch with tri-linear filtering and there may be up to 8 different texture pages touched if the texture location is on a page corner of the nearest coarser mip level. An anisotropic texture fetch may require even more virtual to physical translations.

When a regular virtual memory system is already implemented in hardware, it is desirable to take advantage of such a system to implement a virtual texture system in hardware as well. The virtual memory pages can be used to store texture pages and the virtual memory page tables can be used for the virtual to physical translation of texture data. However, one of the complications is that a virtual memory system has no knowledge of the relationship between texture pages in the mip hierarchy and cannot automatically fall back to a page from a coarser mip if the finer mip is not yet available. When a page for a finer mip is not available, the virtual memory system could simply return a page fault and a texture fetch can be retried with a higher minimum LOD clamp. However, the next attempt may also fail and several retries may be necessary at a performance cost. Another solution could be to store additional information in the page tables of the virtual memory system to allow the virtual to physical translation to automatically fall back to a page from a coarser mip if the finer mip is not yet available. The downside of this approach is that this bloats the page tables that may also be used for data other than textures where this information has no meaning and is simply a waste.

Instead of storing additional information in the page tables, it is also an option to maintain a "min LOD" texture that stores the minimum LOD that is resident in physical memory for each page of the virtual texture. Obviously, such a texture would have to be kept perfectly synchronized with the page tables of the virtual memory system to avoid page faults. A texture lookup into this "min LOD" texture can be used to retrieve the LOD to which the actual virtual texture lookup needs to be clamped to avoid page faults. Unfortunately, a tri-linear and, in particular, an anisotropic texture lookup on a virtual texture may cross several page boundaries on several mip levels. As such, the "min LOD" texture would have to be sampled with a "maximum filter" with an appropriate footprint that corresponds to the footprint of the virtual texture lookup that follows. This requires a "maximum texture filter" to be implemented in hardware and a fast way to determine the sample footprint into the "min LOD" texture based on the footprint of a virtual texture lookup.

Such a "min LOD" texture can also be used to blend in finer mips to avoid LOD snapping or visible "popping" of texture detail when, despite all efforts, there is a significant latency between the time that a page is needed and the time that the data for it is available. When using a "min LOD" texture, the texels of this texture can be gradually adjusted when a new page is mapped to transparently blend in the new page.

9. References

1. Integrated Multiresolution Geometry and Texture Models for Terrain Visualization
   Konstantin Baumann, Jürgen Döllner, Klaus Hinrichs
   Joint Eurographics-IEEE TCVG Symposium on Visualization, May 2000
   Available Online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.5001
2. Texturing Techniques for Terrain Visualization
   Jürgen Döllner, Konstantin Baumann, Klaus Hinrichs
   IEEE Visualization, pp. 227-234, 2000
   Available Online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.4992
3. Rendering massive terrains using chunked level of detail control
   Thatcher Ulrich
   SIGGRAPH Course Notes, 2002
   Available Online: http://tulrich.com/geekstuff/sig-notes.pdf
4. View-Dependent Rendering of Multiresolution Texture-Atlases
   Henrik Buchholz, Jürgen Döllner
   IEEE Visualization, p. 28, 2005
   Available Online: http://www.hpi.uni-potsdam.de/doellner/publications/year/2005/919/BD05.html
5. Terrain Rendering at High Levels of Detail
   Jonathan Blow
   Bolt Action Software, May, 2000
   Available Online: http://number-none.com/blow/papers/terrain_rendering.pdf
6. Adaptive 4-8 Texture Hierarchies
   Lok M. Hwa, Mark A. Duchaineau, Kenneth I. Joy
   IEEE Visualization, pp. 219-226, October 2004
   Available Online: http://www.llnl.gov/tid/lof/documents/pdf/310387.pdf
7. The Clipmap: A Virtual Mipmap
   Christopher C. Tanner, Christopher J. Migdal, Michael T. Jones
   Proceedings of SIGGRAPH 98, pages 151-158, July 1998
   Available Online:
   http://www.cs.virginia.edu/~gfx/Courses/2002/BigData/papers/Texturing/Clipmap.pdf
8. Clip-mapping on the GPU
   Roger Crawfis, Eric Noble, Michael Ford, Frederic Kuck, Eric Wagner
   The Ohio State University, OSU-CISRC-4/07-TR24, April 2007
   Available Online: ftp://ftp.cse.ohio-state.edu/pub/tech-report/2007/TR24.pdf
9. Hardware-Independent Clipmapping
   Antonio Seoane, Javier Taibo, Luis Hernández
   The 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, 2007 (WSCG 2007)
   Available Online: http://wscg.zcu.cz/WSCG2007/Papers_2007/full/A89-full.pdf

10. Clipmap-based Terrain Data Synthesis
Malte Clasen, Hans-Christian Hege
SimVis 2007
Available Online: http://www.zib.de/clasen/download/ClipmapSynthesis.pdf
11. Terrain Rendering using Spherical Clipmaps
Malte Clasen, Hans-Christian Hege
EuroVis 2006 Proc. Eurographics / IEEE VGTC Symposium on Visualization, pp. 91-98, 2006.
Available Online: http://www.zib.de/clasen/download/SphericalClipmaps_Electronic.pdf
12. Virtual Texture: A Large Area Raster Resource for the GPU
Anton Ephanov, Chris Coleman
Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2006
Available Online: http://lists.modsim.org/devrim_extras/2006IITSEC_VTPaper_2.pdf
13. Texture Tile Visibility Determination for Dynamic Texture Loading
Michael E. Goss, Kei Yuasa
EUROGRAPHICS/SIGGRAPH 1998 Workshop on Graphics Hardware, pp. 55-60, Lisbon Portugal, Aug./Sept. 1998
Available Online:
http://www.hpl.hp.com/research/mmsl/publications/3d/texturetilevisibility.pdf
14. Interactive Display Of Very Large Textures
David Cline, Parris K. Egbert
Proceedings of the conference on Visualization, pp. 343-350, October 1998
Available Online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.4571
15. A Perceptually-Based Texture Caching Algorithm for Hardware-Based Rendering
Reynald Dumont, Fabio Pellacini, James A. Ferwerda
Proceedings of the 12th Eurographics Workshop on Rendering Techniques, pp. 249-256, 2001
Available Online: http://www.cs.dartmouth.edu/~fabio/papers/textures01.pdf
16. Adaptive Texture Maps
Martin Kraus, Thomas Ert
Graphics Hardware, pp. 1?10, 2002
Available Online: http://www.vis.uni-stuttgart.de/eng/research/pub/pub2002/hwws02-kraus.pdf
17. Unified Texture Management for Arbitrary Meshes
Sylvain Lefebvre, Jérome Darbon, Fabrice Neyret
Institut National de Recherche en Informatique et en Automatique (INRIA), May 2004
Available Online: http://www-evasion.imag.fr/Publications/2004/LDN04/RR-5210.pdf
18. TileTrees
Sylvain Lefebvre, Carsten Dachsbacher
Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - 2007
Available Online: http://www-sop.inria.fr/reves/Basilic/2007/LD07/LD07.pdf

19. Advanced Virtual Texture Topics
Martin Mittring
Advances in Real-Time Rendering in 3D Graphics and Games Course SIGGRAPH 2008
Available Online: http://portal.acm.org/citation.cfm?id=1404435.1404438
20. Sparse Virtual Textures
Sean Barret
Game Developer Conference 2008
Available Online: http://silverspaceship.com/src/svt/
21. Virtual Texturing
Andreas Neu
RWTH Aachen University, April 2010
Available Online: http://www.graphics.rwth-aachen.de/uploads/media/neu_virtual_texturing_high_03.pdf
22. Implementing Virtual Texturing
Martin Andersson
Lulea University of Technology, May 2010
Available Online: http://epubl.ltu.se/1404-5494/2010/009/LTU-HIP-EX-10009-SE.pdf
23. Virtual Texture Mapping 101
Matthaus G. Chajdas, Christian Eisenacher, Marc Stamminger, Sylvain Lefebvre
GPU Pro, section 3.4, July 1 2010
Available Online: http://www.akpeters.com/product.asp?ProdCode=4728
24. Accelerating Virtual Texturing Using CUDA
Charles-Frederik Hollemeersch, Bart Pieters, Peter Lambert, Rik van de Walle
GPU Pro, section 10.2, July 1 2010
Available Online: http://www.akpeters.com/product.asp?ProdCode=4728
25. Glift: Generic Data Structures for Graphics Hardware
Aaron E. Lefohn
Ph.D. dissertation, Computer Science Department, University of California Davis, September 2006
Available Online: http://graphics.idav.ucdavis.edu/~lefohn/work/dissertation/
26. Geospatial Texture Streaming From Slow Storage Devices
J.M.P. van Waveren
Intel Software Network, June 2008
Available Online: http://softwarecommunity.intel.com/articles/eng/3865.htm
27. Real-Time Texture Streaming & Decompression
J.M.P. van Waveren
Intel Software Network, March 2007
Available Online: http://softwarecommunity.intel.com/articles/eng/1221.htm
28. HD Photo: a new image coding technology for digital photography
Sridhar Srinivasan, Chengjie Tu, Shankar L. Regunathan, Gary J. Sullivan
Proc. SPIE, Vol. 6696, September 2007
Available Online: http://spie.org/x648.html?product_id=767840

29. JPEG XR image coding system - Image coding specification
ISO/IEC FDIS 29199-2 | Draft ITU-T Rec. T.832
PEG document WG 1 N 4918, February 2009
Available Online: http://www.iso.org/iso/catalogue_detail.htm?csnumber=51609
30. Compressed Texture Resources (Direct3D 9)
Microsoft Developer Network
MSDN, November 2007
Available Online: http://msdn2.microsoft.com/en-us/library/bb204843.aspx
31. Block Compression (Direct3D 10)
Microsoft Developer Network
MSDN, November 2007
Available Online: http://msdn2.microsoft.com/en-us/library/bb694531.aspx
32. Real-Time DXT Compression
J.M.P. van Waveren
Intel Software Network, October 2006
Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/324337.htm
33. Real-Time YCoCg-DXT Compression
J.M.P. van Waveren, Ignacio Castaño
NVIDIA, October 2007
Available Online: http://developer.nvidia.com/object/real-time-ycocg-dxt-compression.html
34. Real-Time Normal Map DXT Compression
J.M.P. van Waveren, Ignacio Castaño
NVIDIA developer site, March 2008
Available Online: http://developer.nvidia.com/object/real-time-normal-map-dxt-compression.html
35. Image Upsampling via Texture Hallucination
Yoav HaCohen, Raanan Fattal, Dani Lischinski
International Conference on Computational Problem-Solving (ICCP), 2010
Available Online: http://www.cs.huji.ac.il/~yoavhacohen/upsampling/

Appendix A - Virtual to physical translations

```
/*
    Copyright (c) 2012, Id Software LLC, a ZeniMax Media Company.
    Written by: J.M.P. van Waveren

    Permission is hereby granted, free of charge, to any person
    obtaining a copy of this software and associated documentation
    files (the "Software"), to deal in the Software without
    restriction, including without limitation the rights to use,
    copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of the Software, and to permit persons to whom the
    Software is furnished to do so, subject to the following
    conditions:

    The above copyright notice and this permission notice shall be
    included in all copies or substantial portions of the Software.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
    EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
    OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
    NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
    HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
    WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    OTHER DEALINGS IN THE SOFTWARE.
*/

// constants
const int pageWidth = 128;      // 128 x 128 texels per page
const int pageBorder = 4;       // 4-texel inset border
const int physPagesWide = 32;   // 4096 x 4096 texels per physical texture
const int virtPagesWide = 1024; // 120K x 120K texels per virtual texture

// derived constant used to scale an offset within a virtual page to an offset within a physical page
const float pageFracScale = ( pageWidth - 2.0f * pageBorder ) / ( pageWidth * physPagesWide );

unsigned int physX, physY;      // coordinates of the physical page (in whole pages)
unsighed int physMipLevel;      // mip level of the virtual page stored in the physical page
unsigned int virtX, virtY;      // coordinates of the virtual page (in whole pages)

float physTexelsWide = physPagesWide * pageWidth;
float virtLevelPagesWide = virtPagesWide >> physMipLevel;

// The scale and bias that can be used to translate a virtual address to a physical address.
float scaleST = virtLevelPagesWide * ( pageWidth - 2.0f * pageBorder ) / physTexelsWide;
float biasS = ( physX * pageWidth + pageBorder ) /
               physTexelsWide - scaleST * virtX / virtLevelPagesWide;
float biasT = ( physY * pageWidth + pageBorder ) /
               physTexelsWide - scaleST * virtY / virtLevelPagesWide;

// Scale factor applied to the virtual texture coordinate derivatives
// used for anisotropic texture lookups.
float derivativeScale = pageFracScale * virtLevelPagesWide;

// Physical page coordinates converted to 8-bit values that can be used directly in the fragment
// program to lookup the scale and bias from a mapping texture with one texel per physical page.
unsigned char texelPhysX = (unsigned char)( ( physX / ( physPagesWide - 1.0f ) ) * 255.0f + 0.01f );
unsigned char texelPhysY = (unsigned char)( ( physY / ( physPagesWide - 1.0f ) ) * 255.0f + 0.01f );
```

A.1 - FP32x4 page table

```
// The page table is a four component 32-bit floating-point texture storing
// the scale in the first component and the S,T bias in the last two components.
//
// float pageTable.texel[4] = { scaleST, derivativeScale, biasS, biasT };
//
// For a virtual texture with 1024 x 1024 pages the page table = 21.33 MB.

static float2 VirtualToPhysicalTranslation( sampler2D pageTable, float2 virtCoords ) {
  float4 scaleBias = tex2D( pageTable, virtCoords );
  return virtCoords * scaleBias.x + scaleBias.zw;

  // float derivativeScale = scaleBias.y;
}
```

A.2 - 8:8 page table + FP32x4 mapping texture

```
// The page table is an 8-bit per component unsigned integer luminance-alpha texture
// storing the X,Y coordinates of the physical pages to be used for the virtual pages.
// Also using a single 4 component 32-bit floating-point mapping texture with the scale
// stored in the first component and the S,T bias stored in the last two components.
//
// unsigned char pageTable.texel[2] = { texelPhysX, texelPhysY };
// float physPageScaleBias.texel[4] = { scaleST, derivativeScale, biasS, biasT };
//
// For a virtual texture with 1024 x 1024 pages and physical textures with 32 x 32 pages
// the page table = 2.66 MB and the mapping texture is 16 kB.

static float2 VirtualToPhysicalTranslation( sampler2D pageTable,
                                            sampler2D physPageScaleBias, float2 virtCoords ) {
  half2 physicalPage = tex2D( pageTable, virtCoords ).xw;
  float4 scaleBias = tex2D( physPageScaleBias, physicalPage );
  return virtCoords * scaleBias.x + scaleBias.zw;

  // float derivativeScale = scaleBias.y;
}
```

A.3 - 8:8 page table + 3/4 x FP32x1 mapping textures

```
// The page table is an 8-bit per component unsigned integer luminance-alpha texture
// storing the X,Y coordinates of the physical pages to be used for the virtual pages.
// Also using three single component 32-bit floating-point mapping textures.
// One to store the scale and two for the S,T bias.
//
// unsigned char pageTable.texel[2] = { texelPhysX, texelPhysY };
// float physPageScale.texel[1] = { scaleST };
// float physPageBiasS.texel[1] = { biasS };
// float physPageBiasT.texel[1] = { biasT };
// float virtCoordScale.texel[1] = { derivativeScale };
//
// For a virtual texture with 1024 x 1024 pages and physical textures with 32 x 32 pages
// the page table = 2.66 MB and the mapping textures are 12 kB.

static float2 VirtualToPhysicalTranslation( sampler2D pageTable,
                                            sampler2D physPageScale,
                                            sampler2D physPageBiasS,
                                            sampler2D physPageBiasT, float2 virtCoords ) {
  half2 physicalPage = tex2D( pageTable, virtCoords ).xw;
  float scaleST = tex2D( physPageScale, physicalPage ).x;
  float biasS = tex2D( physPageBiasS, physicalPage ).x;
  float biasT = tex2D( physPageBiasT, physicalPage ).x;
  return virtCoords * scaleST + float2( biasS, biasT );

  // float derivativeScale = tex2D( virtCoordScale, physicalPage ).x;
}
```

A.4 - 8:8 page table + UINT16x1/2 + UINT16x2 mapping textures

```
// The page table is an 8-bit per component unsigned integer luminance-alpha texture
// storing the X,Y coordinates of the physical pages to be used for the virtual pages.
// Also using two unsigned fixed-point 16-bit per component mapping textures.
// One luminance texture with a single component to store the scale
// and one luminance-alpha texture with two components to store the S,T bias.
//
// unsigned char pageTable.texel[2] = { texelPhysX, texelPhysY };
// unsigned short physPageScale.texel[1/2] = {
//     (unsigned short)( scaleST * ( 65535.0f / 32.0f ) + 0.5f ),
//     derivativeScale
// };
// unsigned short physPageBias.texel[2] = {
//     (unsigned short)( ( biasS + 30.0f ) * ( 65535.0f / 32.0f ) + 0.5f ),
//     (unsigned short)( ( biasT + 30.0f ) * ( 65535.0f / 32.0f ) + 0.5f )
// };
//
// For a virtual texture with 1024 x 1024 pages and physical textures with 32 x 32 pages
// the page table = 2.66 MB and the mapping textures are 6 kB.

static float2 VirtualToPhysicalTranslation( sampler2D pageTable,
                                            sampler2D physPageScale,
                                            sampler2D physPageBias, float2 virtCoords ) {
  half2 physicalPage = tex2D( pageTable, virtCoords ).xw;
  float4 scaleST = tex2D( physPageScale, physicalPage ) * 32.0;
  float4 biasST = tex2D( physPageBias, physicalPage ) * 32.0;
  return virtCoords * scaleST.x + biasST.xw - 30.0;

  // float derivativeScale = scaleST.y;
}
```

A.5 - 8:8:8:8 page table

```
// The page table is in 8-bit per component RGBA format with the physical page coordinates
// stored in the first two components and the scale stored in the last two components.
//
// unsigned short scale = ( virtPagesWide * 16 ) >> physMipLevel;
// unsigned char pageTable.texel[4] = { physX, physY, scale & 0xFF, scale >> 8 };
//
// For a virtual texture with 1024 x 1024 pages the page table = 5.33 MB.

static float2 VirtualToPhysicalTranslation( sampler2D pageTable, float2 virtCoords ) {
  // constants
  const float pageWidth = 128;        // 128 x 128 texels per page
  const float pageBorder = 4;         // 4-texel inset border
  const float physPagesWide = 32;     // 4096 x 4096 texels per physical texture

  // Multiplier to undo the hardware specific conversion from an
  // unsigned byte [0, 255] to a floating-point value in the range [0, 1].
#if defined( GEFORCE_7800 ) || defined( RADEON_X1900 )
  const float floatToByte = 255.0 + 1.0 / 256.0;
#else
  const float floatToByte = 255.0;
#endif

  // derived constants
  const float pageFracScale = ( pageWidth - 2 * pageBorder ) / pageWidth / physPagesWide;
  const float borderOffset = pageBorder / pageWidth / physPagesWide;
  const float4 texScale = {
     floatToByte / physPagesWide,
     floatToByte / physPagesWide,
     floatToByte * 1.0 / 16.0,
     floatToByte * 256.0 / 16.0
  };
  const float 4 texBias = { borderOffset, borderOffset, 0, 0 };

  // virtual to physical translation
  float4 physPage = tex2D( pageTable, virtCoords ) * texScale + texBias;
  float2 pageFrac = frac( virtCoords * ( physPage.z + physPage.w ) );
  return pageFrac * pageFracScale + physPage.xy;

  // float derivativeScale = pageFracScale * ( physPage.z + physPage.w );
}
```

A.6 - 5:6:5 page table

```
// The page table is in RGB 565 format with the physical page coordinates stored
// in the 5-bit components and the log2 of the width in pages of the physical
// page's mip level stored in the 6-bit component.
//
// unsigned short pageTable.texel[1] = {
//    ( ( physX * 32 / physPagesWide ) << 11 ) |
//    ( ( log2( virtPagesWide ) - physMipLevel ) << 5  ) |
//    ( ( physY * 32 / physPagesWide ) << 0 )
// };
//
// For a virtual texture with 1024 x 1024 pages the page table = 2.66 MB.
//
// On platforms with an accurate implementation of exp2(), the 1.0 / 4096.0
// addition can be removed and the floor() does not need to include the 'y'
// component which removes a dependency and improves performance.

static float2 VirtualToPhysicalTranslation( sampler2D pageTable, float2 virtCoords ) {
  // constants
  const float pageWidth = 128;        // 128 x 128 texels per page
  const float pageBorder = 4;         // 4-texel inset border
  const float physPagesWide = 32;     // 4096 x 4096 texels per physical texture

  // Multipliers to undo the hardware specific conversion from an
  // 5-bit or 6-bit value to a floating-point value in the range [0, 1].
#if defined( GEFORCE_7800 )
  const float floatToPagesWide = ( 255.0 + 1.0 / 256.0 ) / 256.0 * physPagesWide;
  const float floatToMip = ( 255.0 + 1.0 / 256.0 ) / 256.0 * 64.0;
#elif defined( RADEON X1900 )
  const float floatToPagesWide = ( 31.0 + 1.0 / 33.0 ) / 32.0 * physPagesWide;
  const float floatToMip = ( 63.0 + 1.0 / 65.0 );
#else
  const float floatToPagesWide = 255.0 / 256.0 * physPagesWide;
  const float floatToMip = 255.0 / 256.0 * 64.0;
#endif

  // derived constants
  const float pageFracScale = ( pageWidth - 2 * pageBorder ) / pageWidth / physPagesWide;
  const float borderOffset = pageBorder / pageWidth / physPagesWide;
  const float4 texScale = {
    floatToPagesWide,
    floatToMip,
    floatToPagesWide,
    0
  };
  // small value added to the exponent so we can floor the inaccurate exp2() result
  const float4 texBias = { 0, 1.0 / 4096.0, 0, 0 };

  // translation
  half4 physPage = tex2D( pageTable, virtCoords ) * texScale + texBias;
  physPage.y = exp2( physPage.y );
  physPage = floor( physPage );     // account for inaccurate exp2() and strip replicated bits
  float2 pageFrac = frac( virtCoords * physPage.y );
  return pageFrac * pageFracScale + physPage.xz / physPagesWide + borderOffset;

  // float derivativeScale = pageFracScale * physPage.y;
}
```

Appendix B - Feedback Fragment Program

```
uniform float virtualTextureID;
uniform float feedbackBias;      // log2( feedbackWidth / windowWidth ) + dynamicLODbias

struct PS_IN {
  float4 texcoord0 : TEXCOORD0;
};

struct PS_OUT {
  float4 feedback : COLOR;
};

void main( PS_IN fragment, out PS_OUT result ) {
  const float maxAniso = 4;
  const float maxAnisoLog2 = log2( maxAniso );
  const float virtPagesWide = 1024;
  const float pageWidth = 128;
  const float pageBorder = 4;
  const float virtTexelsWide = virtPagesWide * ( pageWidth - 2 * pageBorder );

  float2 texcoords = fragment.texcoord0.xy * virtTexelsWide;

  float2 dx = ddx( texcoords );
  float2 dy = ddy( texcoords );

  float px = dot( dx, dx );
  float py = dot( dy, dy );

  float maxLod = 0.5 * log2( max( px, py ) );   // log2(sqrt()) = 0.5*log2()
  float minLod = 0.5 * log2( min( px, py ) );

  float anisoLOD = maxLod - min( maxLod - minLod, maxAnisoLog2 );
  float desiredLod = max( anisoLOD + feedbackBias, 0.0 );

  result.feedback.xy = fragment.texcoord0.xy * virtPagesWide;
  result.feedback.z = desiredLod;
  result.feedback.w = virtualTextureID;
}
```

Appendix C - Virtual Texture Fragment Program (Bilinear-Anistropic)

```
uniform sampler2D physicalTextureDiffuse;
uniform sampler2D physicalTextureSpecular;
uniform sampler2D physicalTextureNormal;
uniform sampler2D pageTable;

struct PS_IN {
  float4 texcoord0 : TEXCOORD0;
};

struct PS_OUT {
  float4 color : COLOR;
};

void main( PS IN fragment, out PS OUT result ) {
  float3 physCoords = VirtualToPhysicalTranslation( pageTable, fragment.texcoord0.xy );

#if defined( RADEON_X1900 )
  float2 dx = ddx( fragment.texcoord0.xy ) * physCoords.z;
  float2 dy = ddy( fragment.texcoord0.xy ) * physCoords.z;

  half4 diffuseYCoCg  = tex2D( physicalTextureDiffuse,  physCoords.xy, dx, dy );
  half4 specularRGB   = tex2D( physicalTextureSpecular, physCoords.xy, dx, dy );
  half4 normalXY      = tex2D( physicalTextureNormal,   physCoords.xy, dx, dy );
#else
  half4 diffuseYCoCg  = tex2D( physicalTextureDiffuse,  physCoords.xy );
  half4 specularRGB   = tex2D( physicalTextureSpecular, physCoords.xy );
  half4 normalXY      = tex2D( physicalTextureNormal,   physCoords.xy );
#endif

  half3 diffuse;
  diffuseYCoCg.z = 1.0 / ( ( diffuseYCoCg.z * 255.0 / 8.0 ) + 1.0 );
  diffuseYCoCg.xy *= diffuseYCoCg.z;
  diffuse.r = dot4( diffuseYCoCg, half4(  1.0, -1.0,                  0.0, 1.0 ) );
  diffuse.g = dot4( diffuseYCoCg, half4(  0.0,  1.0, -0.5 * 256.0 / 255.0, 1.0 ) );
  diffuse.b = dot4( diffuseYCoCg, half4( -1.0, -1.0,  1.0 * 256.0 / 255.0, 1.0 ) );

  half3 normal = normalXY.wyz * 2.0 - 1.0;
  normal.z = 1.0 - sqrt( dot( normal.xy, normal.xy ) );

  result.color.rgb = diffuse.rgb + normal.xyz * half3( 0.707, 0.707, 0 ) * specularRGB.rgb;
}
```

Appendix D - Virtual Texture Fragment Program (Trilinear-Anisotropic + Feedback)

```
uniform sampler2D physicalTextureDiffuse;
uniform sampler2D physicalTextureSpecular;
uniform sampler2D physicalTextureNormal;
uniform sampler2D pageTable;     // assumed to be FP32x4

uniform float virtualTextureID;
uniform float feedbackBias;      // log2( feedbackWidth / windowWidth ) + dynamicLODbias

struct PS_IN {
  float4 texcoord0 : TEXCOORD0;
};

struct PS_OUT {
  float4 color : COLOR0;
  float4 feedback : COLOR1;
};

void main( PS_IN fragment, out PS_OUT result ) {
  const float maxAniso = 4;
  const float maxAnisoLog2 = log2( maxAniso );
  const float virtPagesWide = 1024;
  const float pageWidth = 128;
  const float pageBorder = 4;
  const float virtTexelsWide = virtPagesWide * ( pageWidth - 2 * pageBorder );

  float2 texcoords = fragment.texcoord0.xy * virtTexelsWide;

  float2 dx = ddx( texcoords );
  float2 dy = ddy( texcoords );

  float px = dot( dx, dx );
  float py = dot( dy, dy );

  float maxLod = 0.5 * log2( max( px, py ) );   // log2(sqrt()) = 0.5*log2()
  float minLod = 0.5 * log2( min( px, py ) );

  float anisoLOD = maxLod - min( maxLod - minLod, maxAnisoLog2 );

  float4 virtCoordsLod1 = float4( fragment.texcoord0.x, fragment.texcoord0.y, 0.0, anisoLOD - 0.5 );
  float4 virtCoordsLod2 = float4( fragment.texcoord0.x, fragment.texcoord0.y, 0.0, anisoLOD + 0.5 );

  float4 scaleBias1 = tex2Dlod( pageTable, virtCoordsLod1 );
  float4 scaleBias2 = tex2Dlod( pageTable, virtCoordsLod2 );

  float2 physCoords1 = fragment.texcoord0.xy * scaleBias1.x + scaleBias1.zw;
  float2 physCoords2 = fragment.texcoord0.xy * scaleBias2.x + scaleBias2.zw;

#if defined( RADEON_X1900 )
  float2 dx = ddx( fragment.texcoord0.xy );
  float2 dy = ddy( fragment.texcoord0.xy );

  float2 dx1 = dx * scaleBias1.y;
  float2 dy1 = dy * scaleBias1.y;

  float2 dx2 = dx * scaleBias2.y;
  float2 dy2 = dy * scaleBias2.y;

  half4 diffuseYCoCg1 = tex2D( physicalTextureDiffuse,  physCoords1.xy, dx1, dy1 );
  half4 specularRGB1  = tex2D( physicalTextureSpecular, physCoords1.xy, dx1, dy1 );
  half4 normalXY1     = tex2D( physicalTextureNormal,   physCoords1.xy, dx1, dy1 );

  half4 diffuseYCoCg2 = tex2D( physicalTextureDiffuse,  physCoords1.xy, dx2, dy2 );
  half4 specularRGB2  = tex2D( physicalTextureSpecular, physCoords1.xy, dx2, dy2 );
  half4 normalXY2     = tex2D( physicalTextureNormal,   physCoords1.xy, dx2, dy2 );
#else
  half4 diffuseYCoCg1 = tex2D( physicalTextureDiffuse,  physCoords1.xy );
  half4 specularRGB1  = tex2D( physicalTextureSpecular, physCoords1.xy );
  half4 normalXY1     = tex2D( physicalTextureNormal,   physCoords1.xy );

  half4 diffuseYCoCg2 = tex2D( physicalTextureDiffuse,  physCoords2.xy );
  half4 specularRGB2  = tex2D( physicalTextureSpecular, physCoords2.xy );
  half4 normalXY2     = tex2D( physicalTextureNormal,   physCoords2.xy );
#endif
```

```
  float trilinearFraction = fract( anisoLOD );

  half4 diffuseYCoCg  = lerp( diffuseYCoCg1, diffuseYCoCg2, trilinearFraction );
  half4 specularRGB   = lerp( specularRGB1, specularRGB2, trilinearFraction );
  half4 normalXY      = lerp( normalXY1, normalXY2, trilinearFraction );

  half3 diffuse;
  diffuseYCoCg.z = 1.0 / ( ( diffuseYCoCg.z * 255.0 / 8.0 ) + 1.0 );
  diffuseYCoCg.xy *= diffuseYCoCg.z;
  diffuse.r = dot4( diffuseYCoCg, half4(  1.0, -1.0,                  0.0, 1.0 ) );
  diffuse.g = dot4( diffuseYCoCg, half4(  0.0,  1.0, -0.5 * 256.0 / 255.0, 1.0 ) );
  diffuse.b = dot4( diffuseYCoCg, half4( -1.0, -1.0,  1.0 * 256.0 / 255.0, 1.0 ) );

  half3 normal = normalXY.wyz * 2.0 - 1.0;
  normal.z = 1.0 - sqrt( dot( normal.xy, normal.xy ) );

  result.color.rgb = diffuse.rgb + normal.xyz * half3( 0.707, 0.707, 0 ) * specularRGB.rgb;

  result.feedback.xy = fragment.texcoord0.xy * virtPagesWide;
  result.feedback.z = max( anisoLOD + feedbackBias, 0.0 );
  result.feedback.w = virtualTextureID;
}
```