Virtual Texturing in Software and Hardware

SIGGRAPH 2012 Courses

Los Angeles, August 5-9 2012

Juraj Obert
Advanced Micro Devices

J.M.P. van Waveren
ID Software

Graham Sellers
Advanced Micro Devices

Abstract

The objective of this course is to introduce Partially Resident Textures (PRTs), a new GPU feature for virtual texturing, and contrast them with traditional, software-based, methods used for virtual texturing. PRTs are currently available in the Southern Islands (Radeon HD 7xxx) family of graphics processors.

Software-based virtual texturing methods have been used in computer games since 2007. The technology was popularized by John Carmack (ID Software) and ID Software's implementation of virtual texturing is termed Megatexture. The first title featuring virtual texturing was Enemy Territory: Quake Wars (Splash Damage), followed by other titles such as Brink (Splash Damage) or RAGE (ID Software).

The basic idea of virtual texturing is simple — instead of maintaining a separate texture for each object rendered on the screen, all textures are stored in a massive "virtual texture". The size of the virtual texture is in the order of billions of texels and each object is assigned unique virtual texture coordinates from the virtual texture. When used in a shader, the virtual texture coordinates are translated into physical texture coordinates, which are used to access the physical texture that contains the working set of all required tiles.

Existing approaches implement the entire virtual texturing algorithm in software. The software is required to update the page table (yet another texture used for translating virtual texture coordinates into physical ones), perform address translation (by dependent texture lookups in a shader) and deal with hardware differences when it comes to supported texture types, formats and filtering modes. The first part of the course will outline this process and discuss difficulties encountered when deploying this technology in RAGE.

Partially Resident Textures are a new hardware technology that provides direct hardware support for virtual texturing. PRTs eliminate the need for maintaining the page table (the hardware does it) and address translation (the hardware does it) as well as provide support for all texture types/formats and filtering modes (again the hardware does it all). The second part of the course will describe the hardware architecture as it relates to PRTs.

The third part of the course will introduce the new AMD_sparse_texture OpenGL extension that exposes PRTs to software applications. We will present several PRT use cases, a tech demo and discuss strengths and weaknesses of this technology. The course will conclude with a discussion of the limitations present in today's PRT hardware and future plans for PRT development.

About the Authors

Juraj Obert
Advanced Micro Devices
juraj.obert@amd.com

Juraj Obert is a software engineer in the OpenGL driver team at AMD, where he focuses on performance tuning, feature development and software architecture. He has over 10 years of experience in graphics programming and holds a PhD in Computer Science from the University of Central Florida. His research work was previously published at both SIGGRAPH and Eurographics conferences.

J.M.P. van Waveren
ID Software
mrelusive@idsoftware.com

J.M.P. van Waveren studied computer science at Delft University of Technology in the Netherlands. He has been developing technology for computer games for over a decade and has been involved in the research for, and development of various triple-A game titles such as: Quake III Arena, Return to Castle Wolfenstein, DOOM III and RAGE.

Graham Sellers
Advanced Micro Devices
graham.sellers@amd.com

Graham Sellers is the manager of the OpenGL driver team at AMD. He represents AMD at the OpenGL ARB and Khronos Group and is responsible for the design and implementation of new features in AMD's OpenGL implementation, including extensions and new versions of the OpenGL API. He is the author of or a contributor to over 20 OpenGL extensions, many of which are now part of the core API specification. He is listed as a contributor to OpenGL from version 3.2 onwards. He is also a co-author of the OpenGL SuperBible and the upcoming edition of the OpenGL Programming Guide. He holds a Masters' degree in Engineering from the University of Southampton, UK.

Course Outline

10 minutes: Introduction
Juraj Obert

1. Introduction to the course and its goals, course overview and introducing all speakers
2. Introduction to virtual texturing

25 minutes: Challenges of Software Virtual Texturing
J.M.P. van Waveren

1. Virtual texturing in RAGE
2. Different virtual to physical translations
3. Texture filtering
4. Feedback rendering

25–30 minutes: Hardware Virtual Texturing — Partially Resident Textures
Juraj Obert

1. Hardware architecture
2. Driver support (OpenGL, DX)
3. OpenGL AMD_sparse_texture extension (Part 1)

20 minutes: Demo & Future development
Graham Sellers

1. OpenGL AMD_sparse_texture extension (Part 2)
2. Tech demo
3. Future development

5–10 minutes: Conclusion and Discussion
Juraj Obert, J.M.P. van Waveren, Graham Sellers

Contents

Chapter 1

Introduction

These course notes describe software and hardware virtual texturing methods available to graphics developers in early 2012. This course is divided into 3 main parts — Virtual Texturing in Software, Virtual Texturing in Hardware, and Demo/Future Development. The objective of the course it to familiarize the reader with existing software techniques, introduce (in detail) Partially Resident Textures (a new hardware technique for virtual texturing) and provide a comparison between hardware and software techniques. At the end of the course, the reader should have enough knowledge to understand the strengths and weaknesses of all approaches.

1.1 Virtual Texturing in Software

1.1.1 Virtual Textures

Virtual texturing refers to a texturing technique in which multiple object textures are stored in one massive texture called the *virtual texture*. The size of the virtual texture generally exceeds the available storage space[1] in modern GPUs (RAGE used virtual textures that contain $128k \times 128k$ texels). When rendering using a virtual texture, *only parts of it are made resident* on the GPU and accessed from shaders.

For the purpose of this course, the regions of the texture resident in GPU memory at any given time are termed *the working set*. In order to allow applications to selectively upload texture regions to the GPU, the virtual texture is subdivided into *virtual tiles (pages)*. The working set is the set of tiles resident in the GPU memory and it always is a subset of the set of all tiles in the entire virtual texture.

[1]The terms GPU memory and storage space will be used interchangeably throughout this course. Both refer to the memory the GPU is able to access and render from. For high-performance texturing purposes, local GPU memory is used almost exclusively by most applications.

1.1.2 Texture Coordinates

Surfaces of objects referencing the virtual texture are parameterized using *virtual texture coordinates.* Virtual texture coordinates refer to texels in the virtual texture. At rendering time, virtual texture coordinates are translated to *physical texture coordinates* which refer to texels in the working set (i.e. the virtual tiles that are resident in the GPU memory at the time the shader is invoked). In GPU terms, the working set is stored in the GPU memory as a *physical texture.* The physical texture is smaller than the virtual one and serves as a container for tiles that need to be accessible by the current draw call.

Since the texture coordinate space of the virtual texture is different from the texture coordinate space of the physical texture, a mapping must exist between virtual and physical texture coordinates. This mapping is stored in another texture termed the *page table* and the conversion of virtual texture coordinates to physical texture coordinates is termed *virtual-to-physical address translation.*

1.1.3 Rendering

When rendering with virtual textures, the address translation is typically performed inside a shader by a lookup into the page table texture. The input to the lookup is a set of virtual texture coordinates and the result of the lookup is a set of physical texture coordinates. Once obtained, the physical textures coordinates are used to fetch texture data from the physical texture.

The entire pipeline is depicted in Figure 1.1. Notice that the rendering pass is preceded by a feedback pass that determines which texture pages are required to be resident for the next frame. The feedback pass is traditionally implemented on the GPU and requires a readback operation that transfers the IDs of required pages to the CPU. In the analysis pass, the application determines which pages need to be uploaded and which are already resident based on what it knows about the last rendered frame. The non-resident pages that are required for the next frame are then uploaded to the GPU and the page table is updated. Finally, the frame is rendered using the virtual texture.

1.2 Virtual Texturing in Hardware

While sounding fairly easy in theory, implementing virtual texturing in software comes with a great deal of issues. The issues encountered when developing RAGE will be described in detail in Chapter 2. For now, let us mention some just some of them:

- How to deal with different texture filtering modes across virtual tiles? (nearest, linear, aniso, etc.)
- How to deal with mipmapped textures?

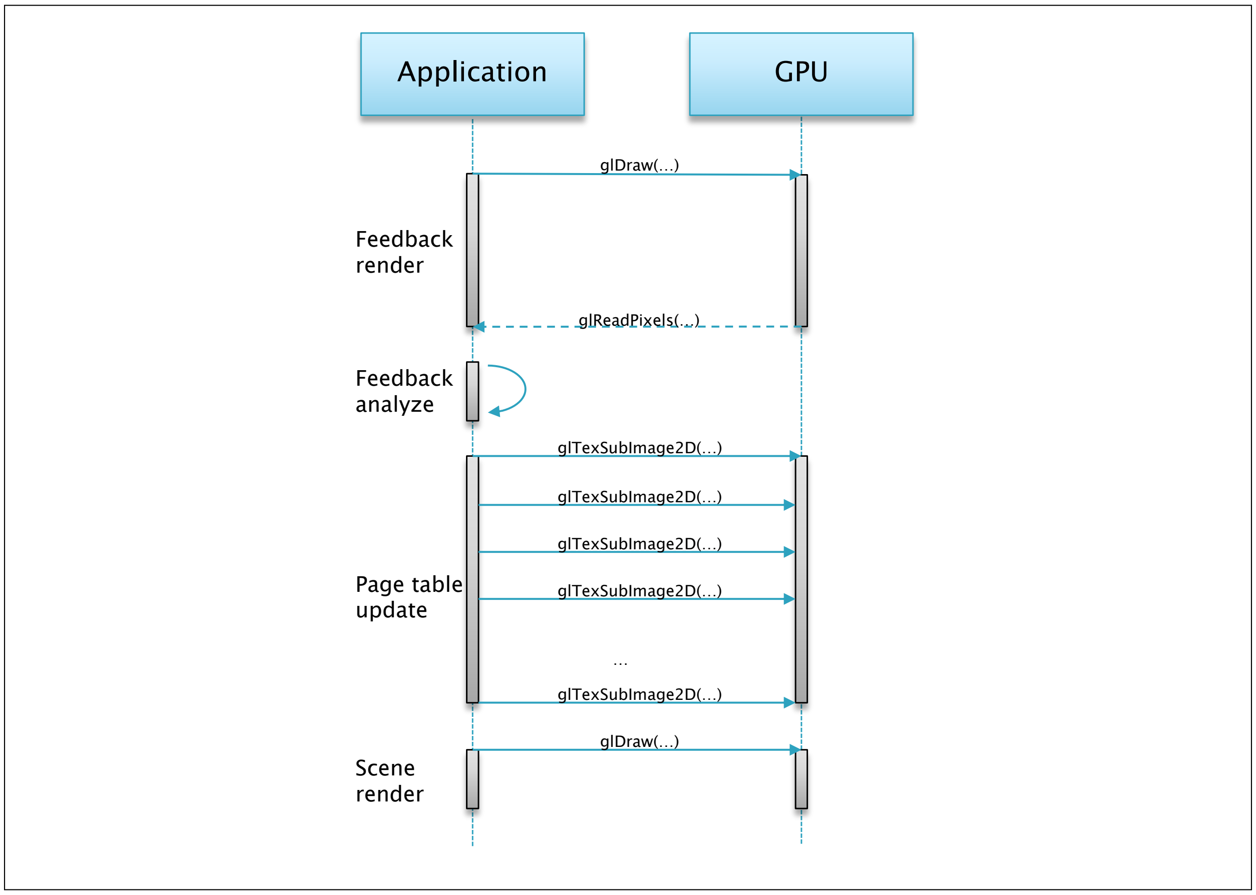


Figure 1.1: Virtual Texturing pipeline.

- How to deal with different texture formats?
- How to deal with special texture types? (such as cubemaps)
- How to deal with multiple platforms?

While almost all of these issues can be resolved on the software side to a quite high level of satisfaction, dealing with all of them requires a non-negligible amount of effort and should not be required on modern GPUs. We will discuss these issues in detail in the next chapter.

1.2.1 Partially Resident Textures

In order to diminish the amount effort game developers need to put into the software implementation of virtual texturing, independent hardware vendors are now moving toward supporting virtual texturing directly in hardware. AMD's Partially Resident Textures (PRTs) represent the first implementation of virtual texturing directly in hardware.

Compared to existing implementations of virtual texturing in software, PRTs provide several advantages, among which are:

- Support for all texture types, formats and filtering modes
- Elimination of dependent texture fetches during rendering
- Support for mipmapped textures (all types)
- Access to tile residency information directly from shaders
- Support for massive texture size

In Chapter 3, we will describe the hardware support for PRTs as it exists in the Southern Islands family of AMD GPUs (Radeon HD 7xxx Series). We will then discuss how the PRT functionality is exposed to developers (OpenGL AMD_sparse_texture extension) and provide examples in the form of screenshots and sample source code.

The rest of this document is organized as follows. In Chapter 2, we describe the implementation of software virtual texture as used in RAGE. We discuss the main challenges encountered when working with software virtual textures. In Chapter 3, we introduce Partially Resident Textures and give an overview of the hardware architecture that supports them. We discuss how PRTs eliminate many of problems inherent to software virtual textures. Finally, in Chapter 4, we propose several PRT use cases, describe the AMD_sparse_texture OpenGL extension and provide outlook on future hardware development.

Chapter 2

Challenges in Software Virtual Texturing

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 (see Figure 2.1), 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.

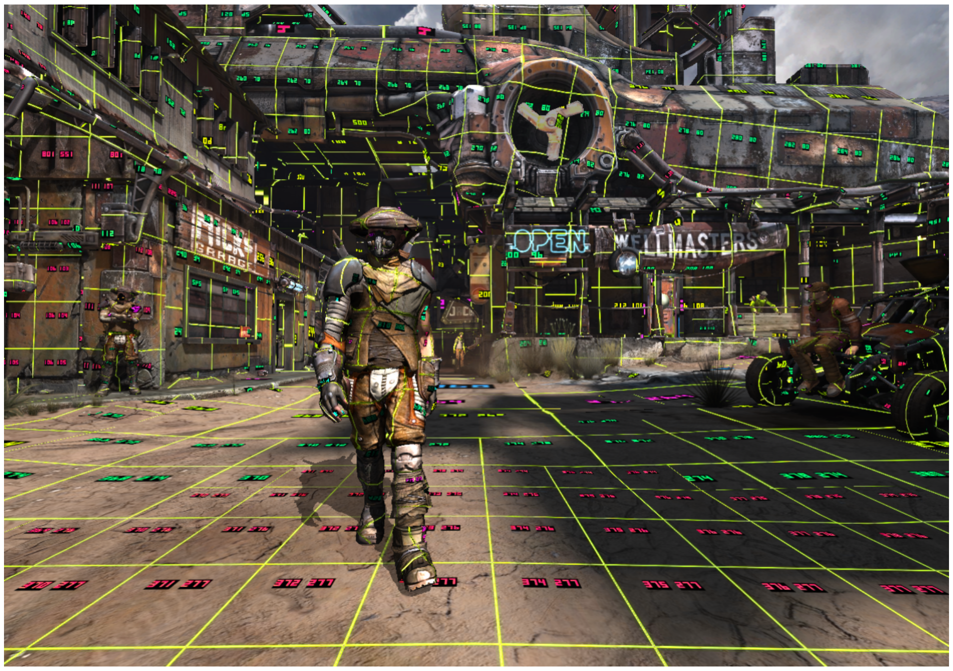


Figure 2.1: Virtual texture pages in RAGE.

Virtual textures not only provide for a reduction of memory requirements but also improved rendering performance through a reduction in both graphics driver and GPU 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 and viable solutions are presented. These include solutions for address translation, texture filtering, compression, caching, and streaming.

2.1 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 128×128. 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 4096×4096 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.

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 pool of physical pages. Figure 2.2 shows an overview of the virtual to physical address translation and the calculation of the scale and bias.

Instead of using a quad-tree data structure, the virtual to physical translation can be implemented in various different ways. These different virtual to physical translations all implement a trade between:

1. Cost of virtual to physical translation during rendering
2. Page table memory requirements
3. Cost of page table updates

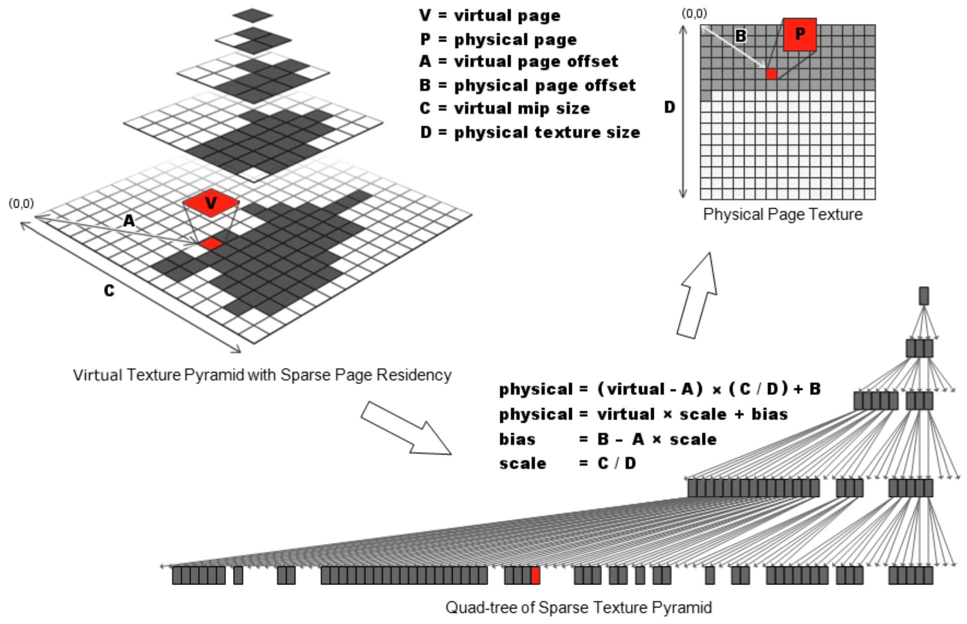

V = virtual page
P = physical page
A = virtual page offset
B = physical page offset
C = virtual mip size
D = physical texture size
(0,0)
A
V
C
(0,0)
B
P
D
Virtual Texture Pyramid with Sparse Page Residency
Physical Page Texture
physical = (virtual - A) × (C / D) + B
physical = virtual × scale + bias
bias = B - A × scale
scale = C / D
Quad-tree of Sparse Texture Pyramid

Figure 2.2: Virtual-to-physical address translation in RAGE.

For instance, 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.

A straightforward approach to implementing the virtual to physical translation is looking up the scale and bias in a mip-mapped FP32x4 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. Even though this results in one of the simplest implementations the page table tends to be rather large (21.33 MB for a virtual texture with 1024×1024 virtual pages) and FP32x4 texture lookups may be costly on some hardware.

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. This approach saves memory by storing the floating-point scale and bias in a much smaller (typically 32×32) texture with one texel per physical page, while allowing access to this texture at the cost of a fully resident much larger (typically 1024×1024) 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.

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 applying the 'frac()' function. The fraction is scaled into the correct range before being added to the physical page offset. The physical page coordinates and the mip level can be stored in an 8-bit per component RGBA texture. It is also possible to store the physical page coordinates in the 5-bit components, and the mip level in the 6 bit component of a 5:6:5: RGB texture. However, this limits the size of the physical page texture and the number of mip levels of the virtual texture. On DirectX9 class hardware there are also various differences in the way these texture components are made available as floating-point values in a fragment program which significantly complicates the calculation of the virtual to physical translation.

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 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.

2.2 Texture Filtering

One of the unfortunate complexities of software virtual textures is that the texture unit, being unaware of the actual texture pages, cannot filter across page boundaries. Instead of using the filter hardware, it is too costly to implement texture filtering completely in a fragment program.

In order to support hardware bi-linear filtering, each physical texture page must have a border of texels around it. Implementations of software virtual textures 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 create and upload this mip level for every physical page that is updated. 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.

Hardware accelerated anisotropic filtering can be supported if the page border is wider than 1 texel. For instance, a 4-texel border is used around each physical page. The 4-texel border maps well to the 4x4 block size of a DXT compressed physical page texture and allows for reasonable quality anisotropic filtering with the maximum anisotropy set to 4.

Ideally, the virtual texture coordinate is used to compute the anisotropic footprint and TXD (tex2Dgrad()) is used for fetching into physical pages. 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. No additional data needs to be stored when the virtual to physical mapping is calculated in the fragment.

Calculating and scaling the derivatives adds fragment program complexity and on most hardware TXD 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 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 arbitrarily large with an arbitrary sign, the anisotropic footprint is still bounded to a single physical texture page because the maximum anisotropy is equal to the border width. The erroneous footprints at page boundaries are a reasonable performance vs. quality trade-off on most hardware. The quality is surprisingly good and the erroneous footprints are only noticeable under significant magnification.

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 is point-sampled using a regular texture lookup unaware of the anisotropic texture fetch that follows. As such the anisotropic texture fetch typically ends up sampling a physical texture page from a mip level that is too coarse to provide useful texture detail. To provide additional texture detail for an 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 bias of the page table texture lookup can be dynamically adjusted based on the anisotropy. The calculation of the bias is shown below.

```
const float minAnisoBias = -2;    // -log2( maxAniso )

float2 dx = ddx( virtCoords.xy );
float2 dy = ddy( virtCoords.xy );

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 anisoBias = max( minLod - maxLod, minAnisoBias );
```

Obviously calculating the LOD bias adds 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 dynamic LOD bias 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 is often 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 minimal shimmering or aliasing appears on surfaces orthogonal to the view direction.

2.3 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 needs to be rendered to a separate buffer to store the virtual page coordinates (x,y), desired mip level, and virtual texture ID (to allow multiple virtual textures). This information is then used to pull in the texture pages 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).

Only the texture coordinates and not the actual texture data are used in the feedback rendering pass which means that alpha tested surfaces are considered completely opaque. To properly pull in texture data that is visible through an alpha tested surface any such 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 the 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, but this turns out not to be a problem in practice.

A similar approach can be used for alpha tested surfaces where every other pixel of the feedback buffer covered by an alpha tested surfaces 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 alpha tested surfaces stacked on top of each other. Using more complex patterns and a different pattern per alpha tested surface can alleviate this problem. This is similar to rendering with screen-door transparency or alpha-to-coverage.

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 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.

Chapter 3

Hardware Virtual Texturing — Partially Resident Textures

Partially Resident Textures provide direct hardware support for the majority of all tasks present in the virtual texturing pipeline. Application developers are no longer required to deal with managing of the page table, address translation and/or figuring out which texture types need to be supported. The responsibility of managing the virtual nature of a texture is moved toward the hardware and the driver.

Partially Resident Textures are supported in all AMD Radeon HD 7xxx GPUs. The functionality is exposed to application developers via the AMD_sparse_texture OpenGL extension and an upcoming DX extension (exact details are not known at the moment).

PRT support in hardware relies on 3 core components:

- HW Virtual Memory subsystem
- Page Residency information propagation
- Driver stack support for efficient mapping/unmapping

3.1 Virtual Memory

Memory addresses used to fetch texture data on Radeon 7xxx GPUs are virtual. When a shader attempts to fetch a texel from a texture using UV coordinates, a dedicated GPU block first computes the virtual memory address of the texel (or a block of texels if filtering is used). The address computation depends on the texture type, format, UV coordinate values, desired mipmap level, offset, internal texture tiling, etc. The virtual memory address is then fed to the virtual memory subsystem. The VM subsystem performs the virtual-to-physical address translation and then initiates a read operation from the physical memory. When the read completes, the texel data is returned to the shader.

The virtual-to-physical address translation inside the VM subsystem leverages a dedicated hardware page table. This is in stark contrast to software virtual texturing techniques in which the page table is just another texture managed by the application. A dedicated hardware page table provides several benefits when compared to a software one.

Unified format When dealing with software (texture) page tables, the application must decide on its size, format, etc. None of this is required with hardware page tables as they have a unified format and support all texture types, formats and sizes. The advantage of this is that applications can use different formats for different purposes, without having to rewrite their address encoding schemes.

The only downside of the unified format is that texture tiles might have different dimensions based on what type/format they are using. In the current hardware, the page size is fixed to 64kB, which means that a 32-bit RGBA8 texture will have tile dimensions of 128 $\times$ 128 texels. PRT tile dimensions for uncompressed 2D textures are listed in Table 3.1.

Texture BPP	PRT Tile Width	PRT Tile Height
128	64	64
64	128	64
32	128	128
16	256	128
8	256	256

Figure 3.1: PRT tile dimensions for uncompressed 2D textures.

Filtering Hardware page tables provide support for all texture filtering modes. With software page tables, certain filtering modes (e.g. trilinear or anisotropic filtering) are very difficult to implement in a robust fashion. As discussed in the previous chapter, this is because the physical texture coordinates are not contiguous across page boundaries. The hardware is not aware of any page boundaries and therefore cannot filter across pages. On the other hand, PRT-enabled hardware supports filtering across page boundaries without issues.

Two-level structure Software page tables used in virtual texturing are traditionally only one-level (in the virtual memory terminology, they only contain the PTEs). This impacts their sizes and does not allow for any kind of compression. Hardware page tables can be two-level (they contain both PDEs and PTEs), which decreases their memory footprint if the virtual address space is only sparsely populated.

Address translation performance Sampling from a virtual texture with software page tables amounts to a texture fetch using virtual texture coordinates followed by another texture fetch using physical texture coordinates. Both of these require roundtrips between the shader and memory, which can incur significant performance penalties should cache trashing occur. With hardware page tables, the address translation happens as a part of the texture fetch that uses virtual texture coordinates directly, diminishing the bandwidth requirements by 50% in the general case.

Simplified programming When dealing with software page tables, the application attempting to map/unmap a page needs to take care of the page table updates. Hardware page tables are programmed by the SW driver stack when the client application commits/clears individual texture tiles. The application is only required to specify which parts of the texture should be resident for the upcoming commands.

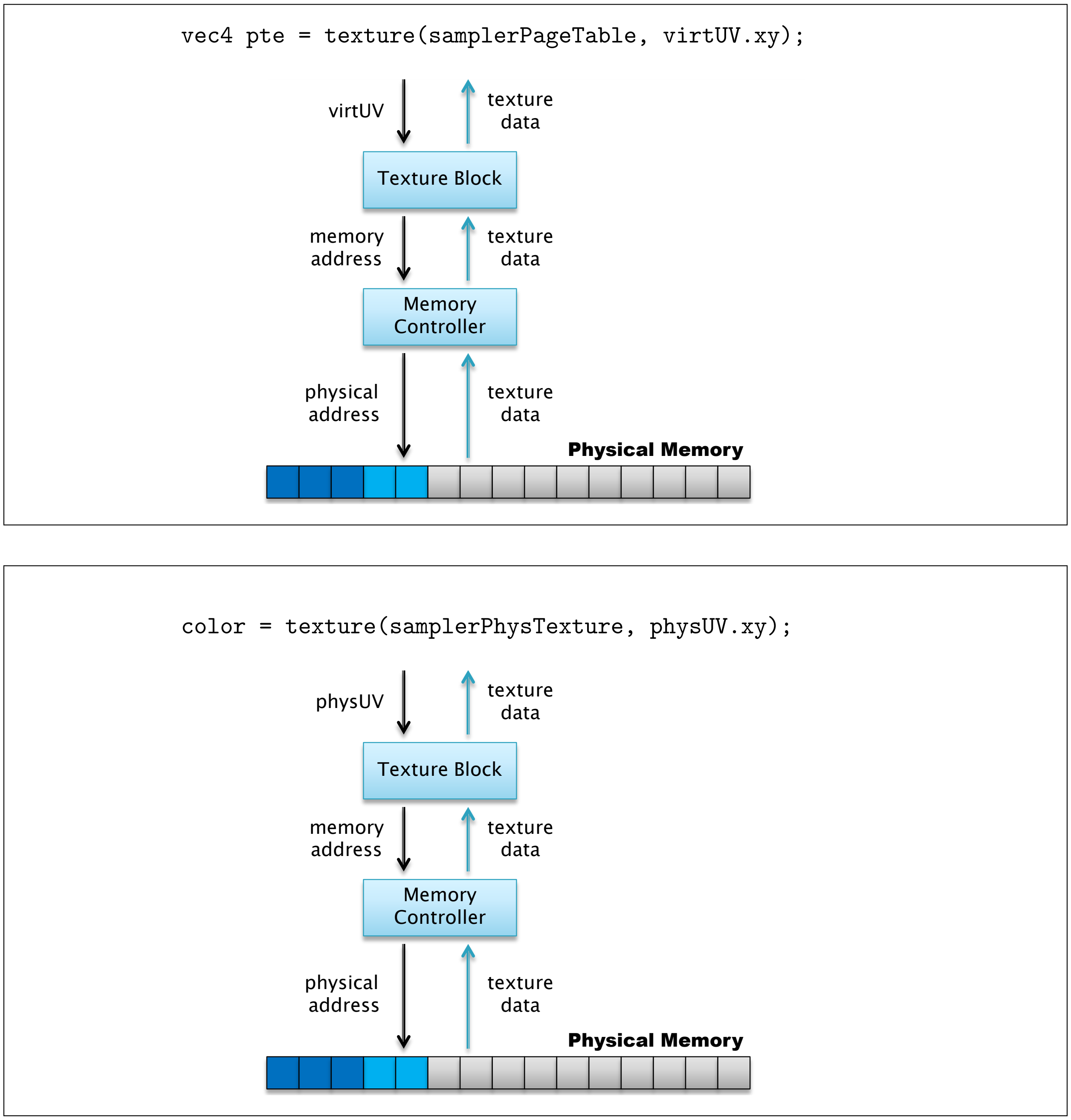
Caching efficiency Hardware page tables can take advantage of special HW caches to speed up lookups and the virtual-to-physical address translation. This is not possible with software page tables, as they go down the traditional texture fetch hardware path.

Consider the following fragment shader that performs sampling using a software (texture) page table:

```
uniform sampler2D samplerPageTable;          // page table
uniform sampler2D samplerPhysTexture;        // physical texture

in vec4 virtUV;                              // virtual texture coordinates
out vec4 color;                              // output color

vec2 getPhysUV(vec4 pte);                    // translation function

void main()
{
    vec4 pte = texture(samplerPageTable, virtUV.xy);        // 1
    vec2 physUV = getPhysUV(pte);                           // 2
    color = texture(samplerPhysTexture, physUV.xy);         // 3
}
```

Figure 3.2 illustrates what happens inside the hardware during software-based virtual-to-physical address translation operation. In the first texture fetch invocation (line 1), the virtual texture coordinates are used to look up the PTE (page table entry). The PTE is an application-specific data structure stored in the page table texture. In the next step (line 2), the PTE is converted to physical texture coordinates, again by an application-specific function that deals with the encoding scheme, filtering, formats, etc. Finally (line 3), the physical texture coordinates are used to fetch the texture data.

From the hardware's point of view, both texture fetches (lines 1 and 3) are pretty much identical except that they are accessing different textures. One important detail to notice is that both texture fetches are dependent, i.e. the second one cannot be launched before the first one completes. Dependent texture fetches are generally not a good approach when high-performance is desirable.



```
vec4 pte = texture(samplerPageTable, virtUV.xy);
```

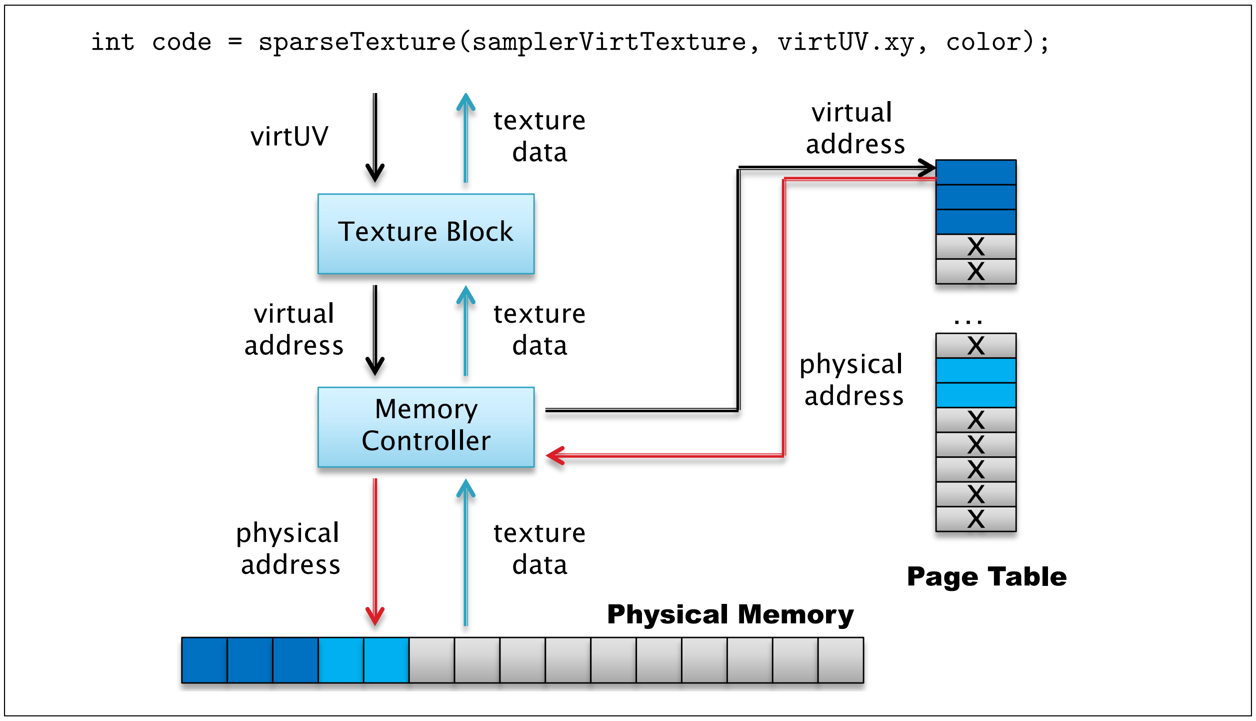
```
color = texture(samplerPhysTexture, physUV.xy);
```

Figure 3.2: Virtual-to-physical address translation using a software (texture) page table and two dependent texture fetches.

Now consider a different fragment shader that takes advantage of a PRT-enabled texture fetch (the sparseTexture() texture sampling instruction is the new instruction introduced in the AMD_sparse_texture OpenGL extension — details in Chapter 4):

```
    uniform sampler2D samplerVirtTexture;

    in vec4 virtUV;                              // virtual texture coordinates
    out vec4 color;                              // output color

    void main()
    {
        // sparse texture fetch
        int code = sparseTexture(samplerVirtTexture, virtUV.xy, color);
    }
```

The shader no longer uses two dependent texture fetches, but instead, the virtual-to-physical address translation is performed directly in hardware based on the virtual texture coordinates passed to the sampling function. The hardware function is depicted in Figure 3.3.



```
int code = sparseTexture(samplerVirtTexture, virtUV.xy, color);
```

Figure 3.3: Virtual-to-physical address translation using a hardware page table.

3.2 Page/Tile Residency Information

The previous section introduced the concept of hardware page tables and discussed their advantages in the context of fetching data from pages that we know are resident in GPU memory. However, a completely different class of algorithms can be based on the idea of determining page residency information at runtime.

Page fault is a virtual memory event that occurs when the client attempts to translate a virtual address that does not have an entry in the page table (i.e. the page is not mapped to any physical address). In virtual texturing systems, it is very convenient when a shader is able to determine page residency status in an efficient manner.

Querying page residency status from software page tables is straightforward — the shader performs a texture fetch from the page table texture (as in Figure 3.2) and then tests the resulting address for validity (an application specific value can be stored in the page table to indicate unmapped access). The cost of the texture fetch is equal to the cost of any other texture fetch.

With PRTs, the hardware directly supports propagating of the page residency information from the page table to the shader core. In other words, if the shader attempts to read from an unmapped virtual address, the hardware will report failure without having to perform a read from the texture memory. The return code is referred to as a *NACK* in the rest of this document. The sequence of events is illustrated in Figure 3.4.

```
int code = sparseTexture(samplerVirtTexture, virtUV.xy, color);
```

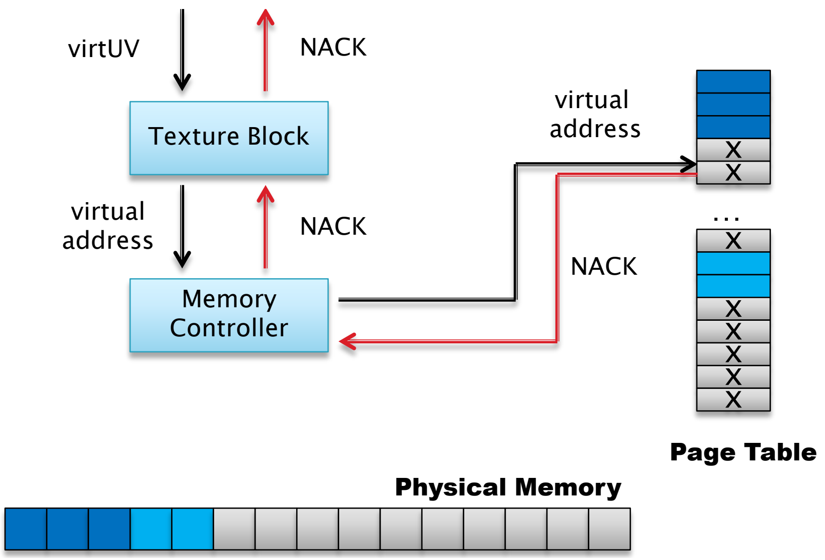


Figure 3.4: NACK propagation.

Chapter 4

PRT Use Cases & Future Development

In the first part of this chapter, we cover some use cases of PRTs in the real world and demonstrate techniques that may be implemented using the PRT feature. Use cases are enumerated below. In a second part of this chapter, we address some current limitations of the approach implemented in AMD's hardware, and some thoughts on future directions.

4.1 Very Large Texture Arrays

First, we discuss the use of very large texture arrays as an application managed cache of textures that may be used to virtually eliminate texture binds in a real-time application. Under this scheme, one, very large texture array is allocated for each class of texture (say, diffuse albedo, specular coefficients, normal maps, etc.). These array textures are bound and left bound for the lifetime of the application. Each material in a scene is assigned a slice of the array. On current hardware, we are able to support more than 8,000 slices in a single texture array, allowing more than 8,000 unique materials to be represented in a single array. Of course, a moderate sized array texture (of the order of 2K $\times$ 2K texels) with 8,000 slices consumes more than 10s of gigabytes of address space and so it is impossible to ensure that all of the texture data is resident at all times.

However, assuming that the live data set for a single rendering call can be made resident (i.e., it fits in GPU memory), several advantages arise from using texture arrays with sparse textures. The first of these is that the layout of textures in memory is consistent between materials. All materials have access to their diffuse albedo, specular coefficient and normal map textures if they have them. For those materials that do not have some of those components, then those slices of the array may be left non-resident without consuming precious physical memory — only virtual memory is reserved. This simplifies shader development as it allows texture layout to be declared boiler-plate style.

The second, and perhaps more important aspect to this approach is that the texture array can be considered an application-controlled cache. When a material is about to be rendered, the application must ensure that the relevant slices of the appropriate texture arrays are present in GPU memory. However, there is no need to bind new textures as all of the texture data is actually part of the same set of array textures. As materials are rendered, new slices of the array are uploaded to the GPU as needed and then left resident. If the same slice is needed again, then it is already resident and no texture upload or rebind operation is necessary. If, during texture upload, an out-of-memory error is detected, slices that are no longer needed may be discarded and a new attempt to make pages resident made. If, during the rendering of a single frame, all textures needed fit into GPU memory, then nothing is discarded, everything remains resident and no paging is necessary on the next frame. Thus, the subsequent frame may be rendered with no texture binds at all.

Once the need for binding textures between draw commands is eliminated, several common optimizations found in modern realtime graphics engines become redundant. For example, engines often sort or bin geometry in order to reduce state changes. As a change in texture is no longer considered a state change, this sorting becomes less important. As another example, large, complex models consisting of surfaces with many materials are often broken into several smaller parts for rendering. As all of the texture data for these parts can now be made resident simultaneously, this can be avoided by simply attaching a per-chunk material ID to what would previously have been separate drawing commands. Other graphics features such as instancing become more applicable here.

4.2 Incomplete Mip-map Chains

A second technique that becomes possible with sparse textures is the use of incomplete mip-map chains. These may be used for procedurally generated textures or streaming texture data from networks, optical drives or other slow media. Under such circumstances, a minimum level of detail is made resident before scene rendering begins. This level can be chosen by the developer, but due to artifacts of the PRT implementation, is likely to be a minimum of 64KB per texture. This data may, perhaps, be kept closer to the engine in the form of a decompressed base-level texture set, or a set of texture data that is downloaded first. During rendering, a record is made of which textures are actually necessary during scene traversal. This could be done on the CPU based on some simple CPU-based rendering, through GPU assisted techniques such as occlusion queries, or entirely on the GPU by writing texture access data into images in GPU memory. The application then periodically examines the list of live textures and brings them into GPU memory on demand.

On the shader side, an attempt is made to fetch the textures that are required to render the scene. If the necessary textures are not resident in GPU memory, a signal is returned to the shader to indicate so, and the shader begins traversing the mip-map pyramid until a resident texel is found. Because the application made all of the lowest resolution mip-map levels resident during initialization, it is guaranteed that some reasonable texture data is found during this pyramid walk. Over the next few frames, texture data becomes resident – either by loading it from the slow resource, or by generating it on the fly using the CPU or even the GPU itself. Non-resident textures are displayed as blurry, downsampled versions of their higher resolution counterparts at first, and over the course of one or more frames, become sharper. Because no physical address space is required for non-resident texture data, the largest resolution layer of the mip-map pyramid need not even exist if it is known a-priori that it will never be accessed by the texture. The same algorithm follows, though; traverse the mip-map pyramid, starting from the desired LoD until a resident texel is found.

4.3 Truly Sparse Textures

Truly sparse textures are another excellent use case for PRT. For example, consider a traditional texture atlas. In general, tools must find a balance between tightly packing atlas components in order to conserve empty space, and leaving enough space between those components to avoid bleeding during the generation of the mip-map chain. With PRT, this is not as necessary. Large regions of unused space may be left empty between components of a texture atlas. Any 64KB chunk of texture can be ignored as it will not be allocated in physical storage. Large, irregular shapes may be created in the atlas without worrying about filling the voids in the convex or even hollow outlines. Sparsity is even more relevant in 3D and volumetric data-sets. A 3D scan of a large volume can often consume many gigabytes of storage, but contain large homogeneous regions and even voids. By using a PRT to store these types of texture, larger volumes that would previously have been impossible to render without complex shader driven page tables may be simply treated as large contiguous textures. Those regions that are completely empty may be left entirely un-allocated. For those regions where lower frequency or even single-valued data is acceptable, the very lowest level of the 3D mip-map pyramid may be used. Use in ray-marching or slice-based rendering algorithms of these apparently complete data sets is then trivial.

4.4 Current Limitations and Thoughts on the Future

The PRT feature we are shipping in hardware is certainly very powerful, but does not address all the wants or needs of the current SVT community. In particular, the maximum texture size has not changed - it is 16K $\times$ 16K $\times$ 8K texels. The limit lies in the precision of the representation of texture coordinates with enough sub-texel resolution for artifact-free linear sampling. To some degree, this may be easy to lift, but we are seeing requests from developers to go as high as 1M $\times$ 1M or more in a single texture. This presents significant architectural challenges and may or may not be feasible in the near term.

It is also easy to see that with large textures and high precision texel formats, we start to exhaust even the virtual address space of the GPU. The largest possible texture is 16K $\times$ 16K $\times$ 8K $\times$ 16 bytes per texel. This amounts to 32 terabytes of linear address space. This far exceeds the addressable space available to the GPU, irrespective or residency. Furthermore, as it is backed by the virtual memory subsystem, page table entries need to be allocated for those pages referenced by sparse textures. The approximate overhead of the page tables for a virtual allocation on current-generation hardware is 0.02% of the virtual allocation size. This does not seem like much and for traditional uses of virtual memory, it is not. However, when we consider ideas such as allocation of a single texture which consumes a terabyte of virtual address space, this overhead is 20GB — much larger than will fit into the GPU's physical memory. To address this, we need to consider approaches such as non-resident page tables and page table compression.

There are several use cases for PRT that seem reasonable but that come with subtle complexities that prevent their clean implementation. One such complexity is in the use of PRTs as renderable surfaces. Currently, we support rendering to PRTs as color surfaces. Writes to un-mapped regions of the surface are simply dropped. However, supporting PRTs as depth or stencil buffers becomes complex. For example, what is the expected behavior of performing depth or stencil testing against a non-resident portion of the depth or stencil buffer? Also, supporting rendering to MSAA surfaces is not well supported. Because of the way compression works for multisampled surfaces, it is possible for a single pixel in a color surface to be both resident and non-resident simultaneously, depending on how many edges cut that pixel. For this reason, we do not expose depth, stencil or MSAA surfaces as renderable on current generation hardware.

The operating system is another component in the virtual memory subsystem which must be considered. Under our current architecture, a single virtual allocation may be backed by multiple physical allocations. Our driver stack is responsible for virtual address space allocations whereas the operating system is responsible for the allocation of physical address space. The driver informs the operating system how much physical memory is available and the operating system creates allocations from these pools. During rendering, the operating system can ask the driver to page physical allocations in and out of the GPU memory. The driver does this using DMA and updates the page tables to keep GPU virtual addresses pointing at the right place. During rendering, the driver tells the operating system which allocations are referenced by the application at any given point in the submission stream and the operating system responds by issuing paging requests to make sure they are resident. When there is a 1-to-1 (or even a many-to-1) correspondence between virtual and physical allocations, this works well. However, when a large texture is slowly made resident over time, the list of physical allocations referenced by a single large virtual allocation can become very long. This presents some performance challenges that real-world use will likely show us in the near term and will need to be addressed.

Appendix A

Code Samples

To query PRT tile dimensions for a give texture format/type:

```
GLint sizeX = 0;
GLint sizeY = 0;
GLint sizeZ = 0;

glGetInternalformativ(GL_TEXTURE_2D, GL_RGBA8, GL_VIRTUAL_PAGE_SIZE_X_AMD, 1, &sizeX);
glGetInternalformativ(GL_TEXTURE_2D, GL_RGBA8, GL_VIRTUAL_PAGE_SIZE_Y_AMD, 1, &sizeY);
glGetInternalformativ(GL_TEXTURE_2D, GL_RGBA8, GL_VIRTUAL_PAGE_SIZE_Z_AMD, 1, &sizeZ);
```

To create a 2D partially resident texture with 5×10 tiles:

```
GLuint prtTexture = 0;
glGenTextures(1, &prtTexture);
glBindMultiTexture(GL_TEXTURE0, GL_TEXTURE_2D, prtTexture);
glTexStorageSparseAMD(GL_TEXTURE_2D, GL_RGBA8, sizeX * 5, sizeY * 10, 1, 0,
                      GL_TEXTURE_STORAGE_SPARSE_BIT_AMD);
```

To map a 2×1 PRT region at offset $(0, 0)$ in mipmap level 0:

```
glBindMultiTexture(GL_TEXTURE0, GL_TEXTURE_2D, prtTexture);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, sizeX * 2, sizeY * 1,
                GL_RGBA, GL_UNSIGNED_BYTE, data);
```

To unmap the same PRT region:

```
glBindMultiTexture(GL_TEXTURE0, GL_TEXTURE_2D, prtTexture);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0,  sizeX * 2, sizeY * 1,
                GL_RGBA, GL_UNSIGNED_BYTE, NULL);
```

To check for tile residency inside a fragment shader:

```
uniform sampler2D sampler;

in vec4 colorVert;
in vec4 texCoordVert;
out vec4 fragmentColor;

void main()
{
    vec4 outColor = vec4(1.0, 1.0, 1.0, 1.0);

    int code = sparseTexture(sampler, texCoordVert.xy, outColor);

    if (sparseTexelResident(code))
    {
        // data present
        fragmentColor = vec4(outColor.rgb, 1.0);
    }
    else
    {
        // NACK
        fragmentColor = vec4(1.0, 0.0, 0.0, 1.0);
    }
}
```