SIGGRAPH 2012

The **39th** International **Conference** and **Exhibition** on **Computer Graphics** and **Interactive Techniques**

Virtual Texturing in Software and Hardware

Monday, 6 August, 2:00pm - 3:30pm

Juraj Obert
Advanced Micro Devices

J.M.P. van Waveren
id Software

Graham Sellers
Advanced Micro Devices

SIGGRAPH2012

Agenda

SIGGRAPH2012

- Introduction to Virtual Texturing
- Software Virtual Textures (Megatexture) in RAGE
- Partially Resident Textures (PRTs)
- OpenGL sparse texture extension
- Demo (RAGE running PRTs)
- Conclusion & Discussion

Introduction to Virtual Texturing

Juraj Obert
Advanced Micro Devices

SIGGRAPH2012

Virtual Texturing

SIGGRAPH2012

- Non-virtual textures
  - One (or multiple) physical textures per game object
  - Game needs to bind them all before a draw call

Virtual Texturing

SIGGRAPH2012

- Virtual textures
  - One massive virtual texture that contains data for the entire world
  - Only one texture needs to be bound at any given time
- Problem
  - The texture cannot possibly fit into video memory
  - E.g., some RAGE virtual textures are 128K x 128K texels (64 GB)

Virtual Texturing

- Paging
  - Making only a part of the virtual texture resident in GPU memory
  - Tile (page) granularity
  - **Working set** – the set of texture tiles resident in GPU memory
    - Represented as another **physical texture** in GPU memory
    - Orders of magnitude smaller than the virtual texture (needs to fit in GPU memory)
    - Application decides based on FOV, map location, view direction, etc.

Virtual Texturing

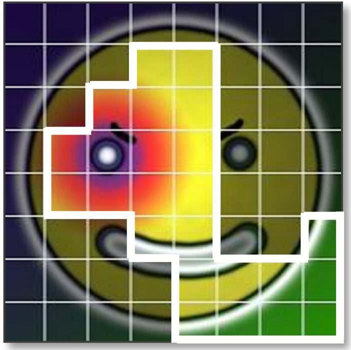
- Paging
  - Virtual texture subdivided into **tiles (pages)**

Virtual texture

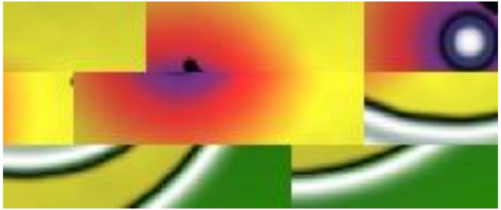
Images courtesy of Sean Barrett

Virtual Texturing

- Paging
  - Tiles uploaded into the physical texture



Virtual texture

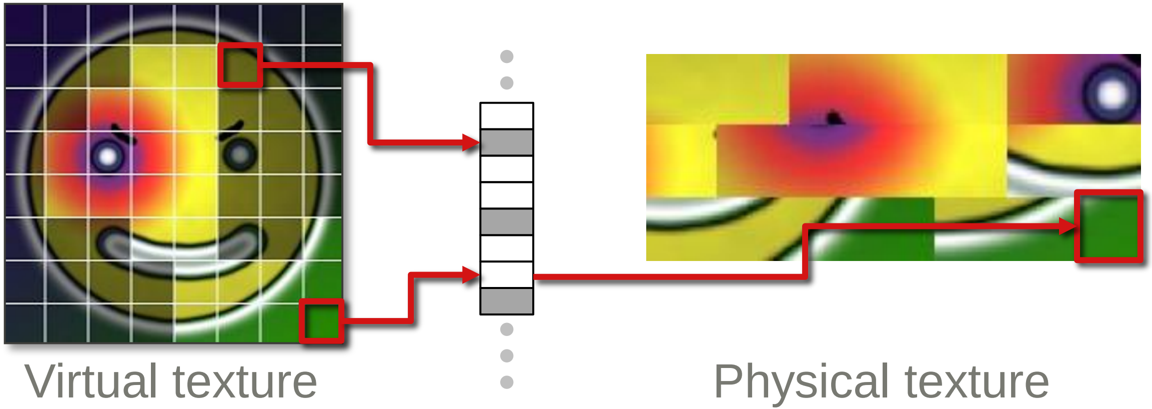


Physical texture

Images courtesy of Sean Barrett

Virtual Texturing

- Virtual texture coordinates are mapped to physical texture coordinates through a **page table texture**



Images courtesy of Sean Barrett

Partially Resident Textures – Shader

SIGGRAPH2012

- SVT texture lookup

```
uniform sampler2D samplerPageTable;                 // page table texture
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
}
```

Virtual Texturing

- Software virtual textures
  - Powerful tool to handle massive datasets
  - Simple in theory, but hard to implement efficiently

Software Virtual Textures in RAGE

J.M.P. van Waveren
id Software

SIGGRAPH2012

Virtual Textures in Software

- Motivation
- Address Translation
- Texture Filtering

RAGE

RAGE

Unique Texture Detail

SIGGRAPH2012

- Desire for unique detail at a distance and up close.
- Texture mapping efficiently adds surface detail to geometric primitives.
- Tiling, blending and decals are forms of manual texture compression.
- Tiling looks bad at a distance.
- Bilinear magnification looks bad up close.
- Hunger for truly unique detail results in huge texture data set.

Key Observations

SIGGRAPH2012

- Massive amount of texture data and only so much physical memory.
- GPU compression formats designed for rendering performance.
- Texture data can be stored highly compressed on secondary storage.
- Lossy compression is perfectly fine for many use cases.
- Only small subset of texture data needed at any time.
- Temporarily fall back to slightly blurrier texture data without stalling execution (trade quality vs. performance).

Virtual and Physical Texture Data

- Massive amount of texture data in a virtual address space.
  - Possibly highly compressed in non-renderable format.
- Smaller resident subset in a physical address space.
  - Possibly compressed in a GPU renderable format.
- Translate virtual texture addresses to physical addresses.
  - Various address translation schemes can be applied.

Address Translation Options

SIGGRAPH2012

- Per Model.
  - LOD system where each geometry LOD has its own texture LOD.
  - Make a different texture resident for each LOD.
- Per Vertex.
  - Modify the geometry texture coordinates at run-time.
- Per Fragment.
  - Translate the texture address per fragment (or per texture lookup).
  - Unwrap all UV islands onto one very large texture.
  - Divide this large texture into pages that are made resident as needed.
  - Virtual texture pages map to physical texture pages.
  - Use address translation to map virtual addresses to physical ones.
- Per Point Sample.
  - Filtering in software is rather expensive. Need hardware support!

ClipMap

- Back in the day required hardware support.
- Can easily be implemented on programmable graphics hardware.
- Texture sub-square resident around single focus point on texture.
- Single region of interest significantly simplifies the address translation.
- No page table needed!
- Limited to environments with natural spatial correlation between texture data and geometry.

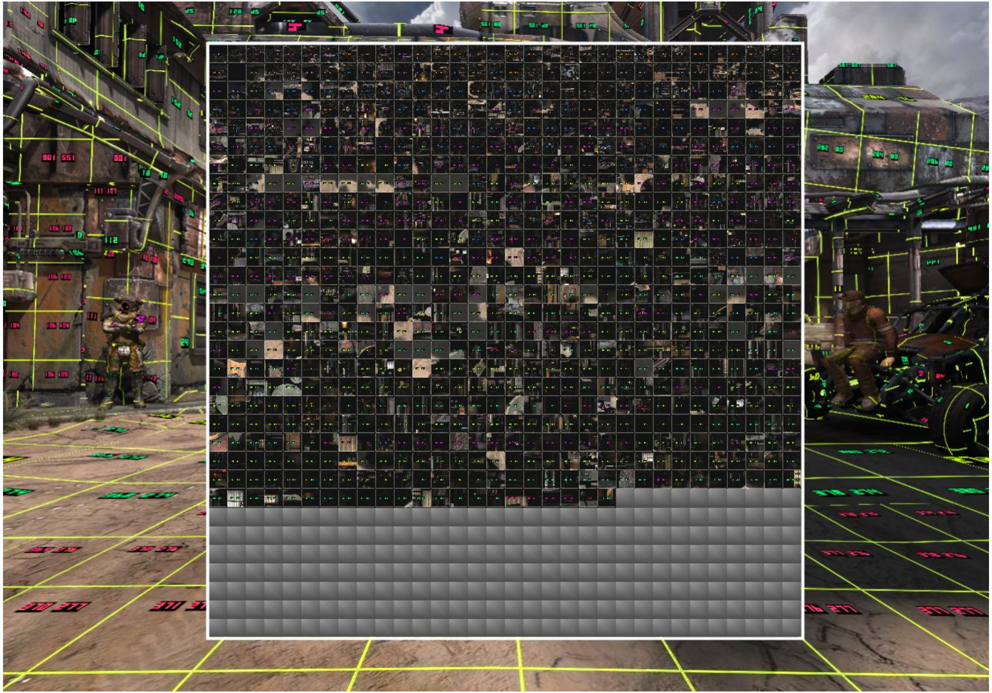
Flexible Address Translation

- Not all environments have a natural correlation between the geometry and texture data.
- Need more flexible texture management and address translation.
- Need to map arbitrary virtual texture pages to physical memory.

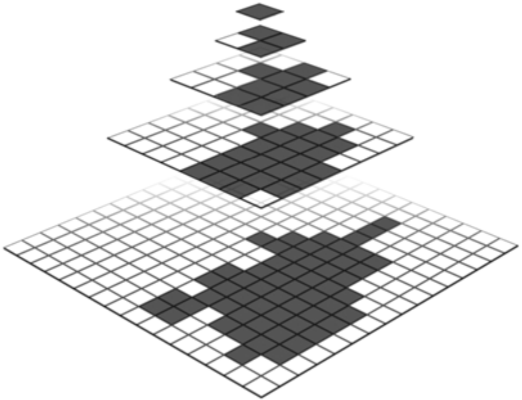
Virtual Textures in RAGE

Virtual Textures in RAGE

Virtual Textures in RAGE

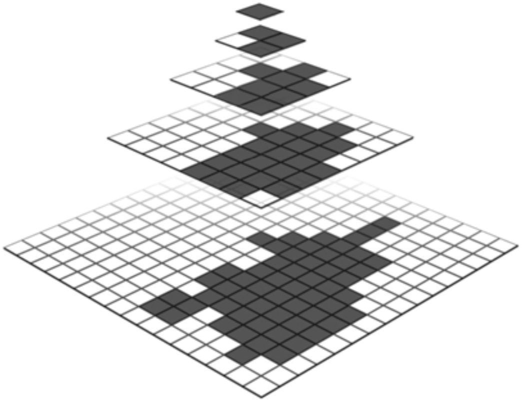


Flexible Address Translation

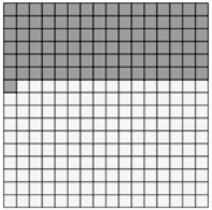


Virtual Texture Pyramid with Sparse Page Residency

Flexible Address Translation

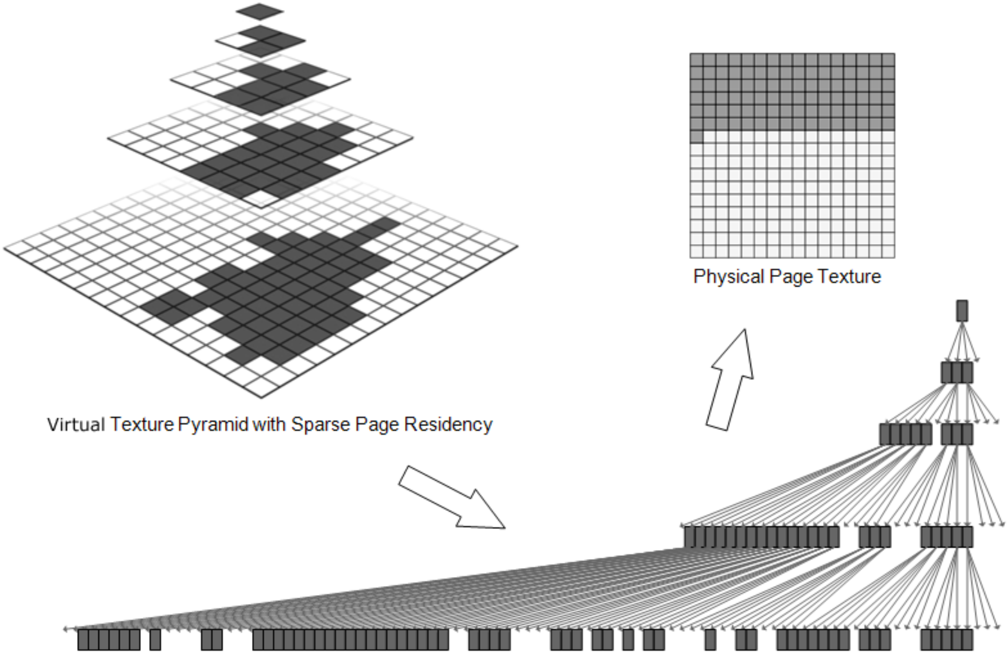


Virtual Texture Pyramid with Sparse Page Residency



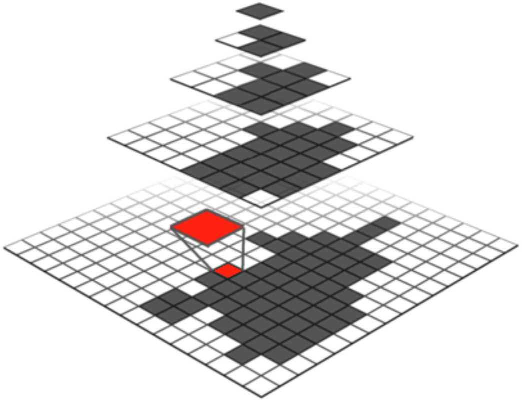
Physical Page Texture

Flexible Address Translation

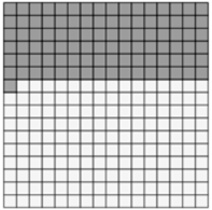


Quad-tree of Sparse Texture Pyramid

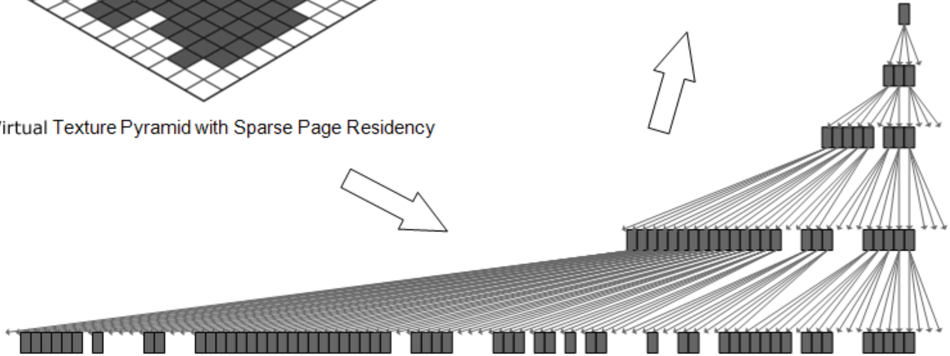
Flexible Address Translation



Virtual Texture Pyramid with Sparse Page Residency

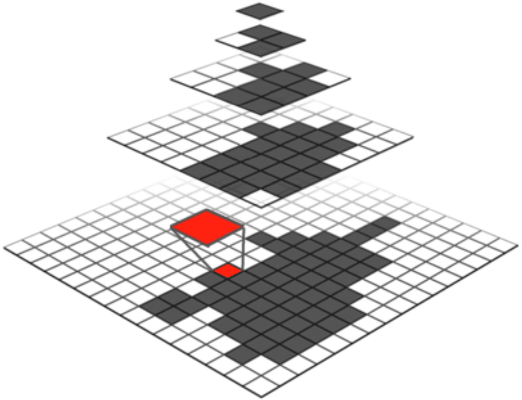


Physical Page Texture

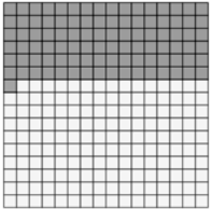


Quad-tree of Sparse Texture Pyramid

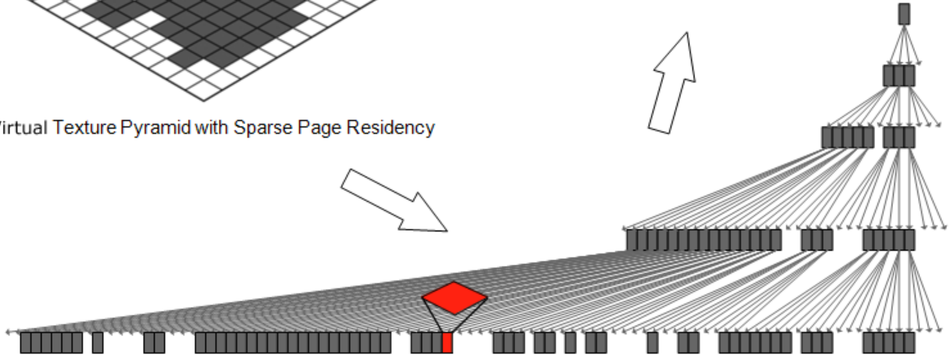
Flexible Address Translation



Virtual Texture Pyramid with Sparse Page Residency

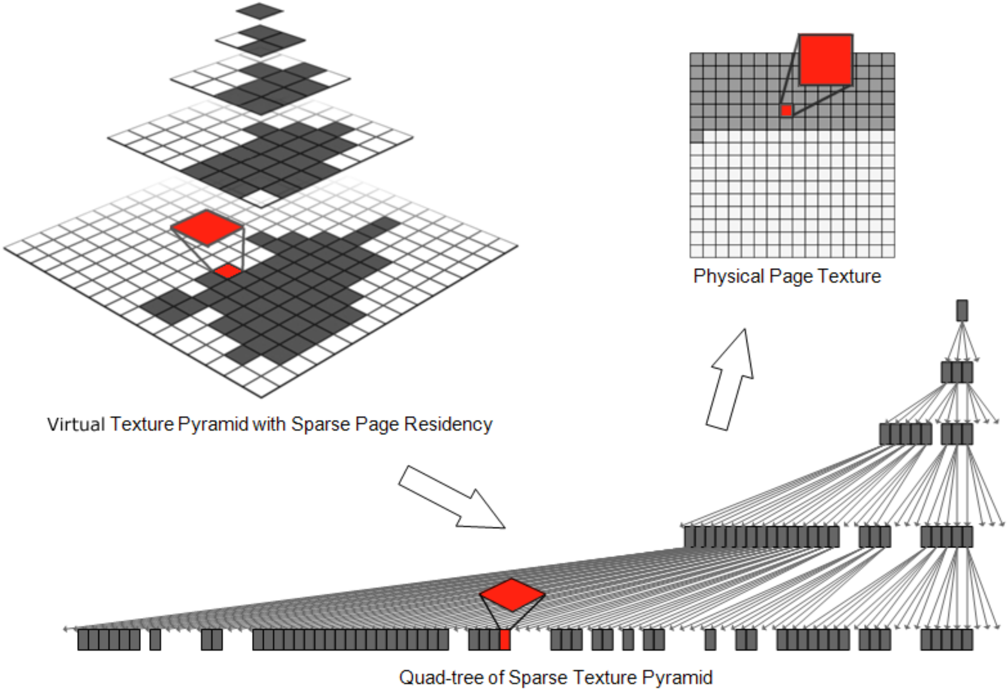


Physical Page Texture

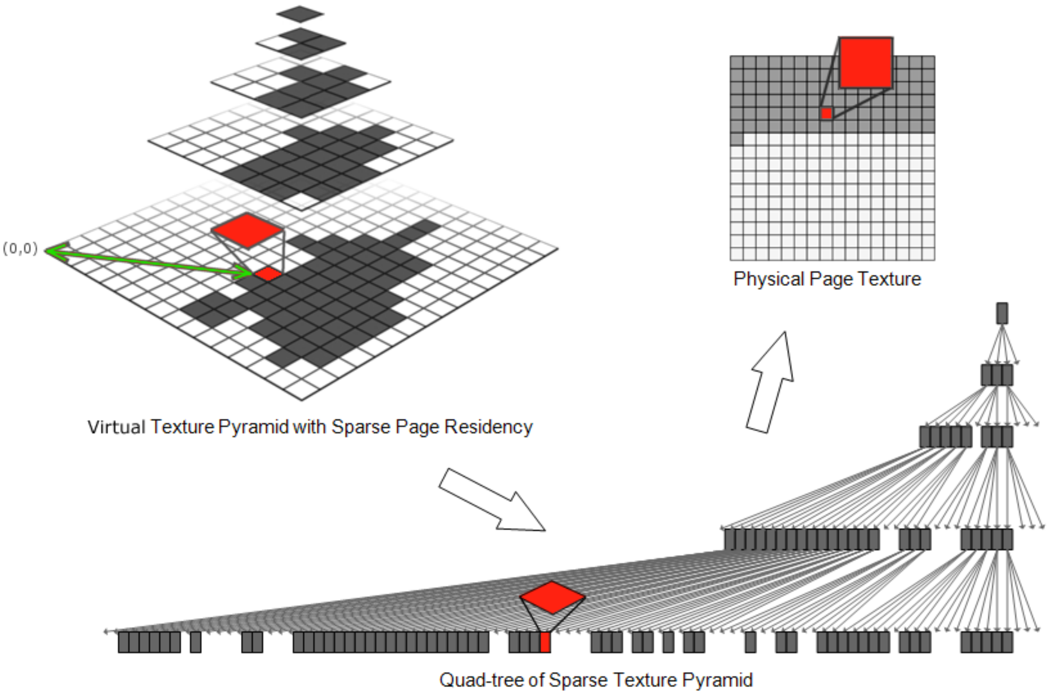


Quad-tree of Sparse Texture Pyramid

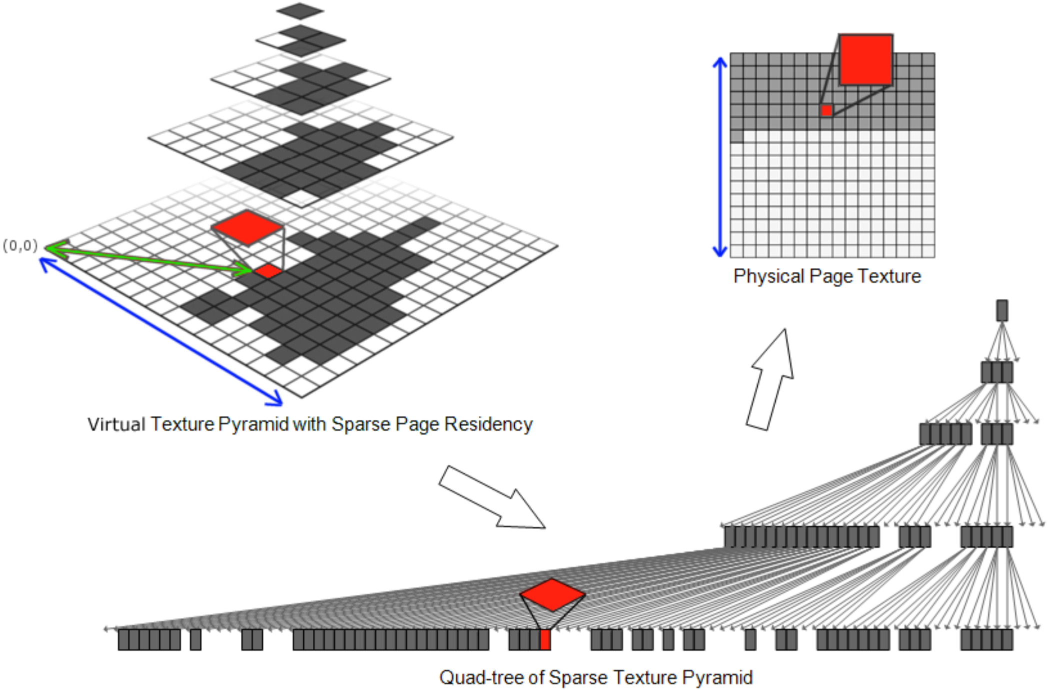
Flexible Address Translation



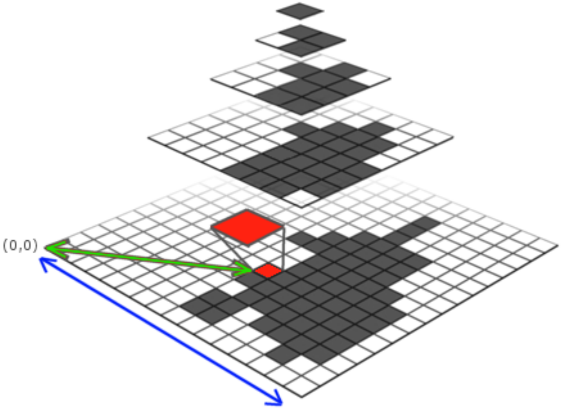
Flexible Address Translation



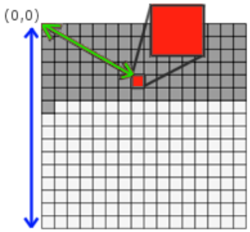
Flexible Address Translation



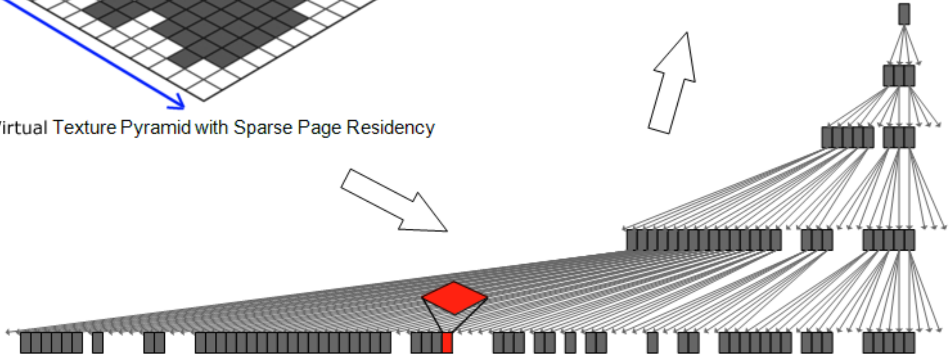
Flexible Address Translation



Virtual Texture Pyramid with Sparse Page Residency

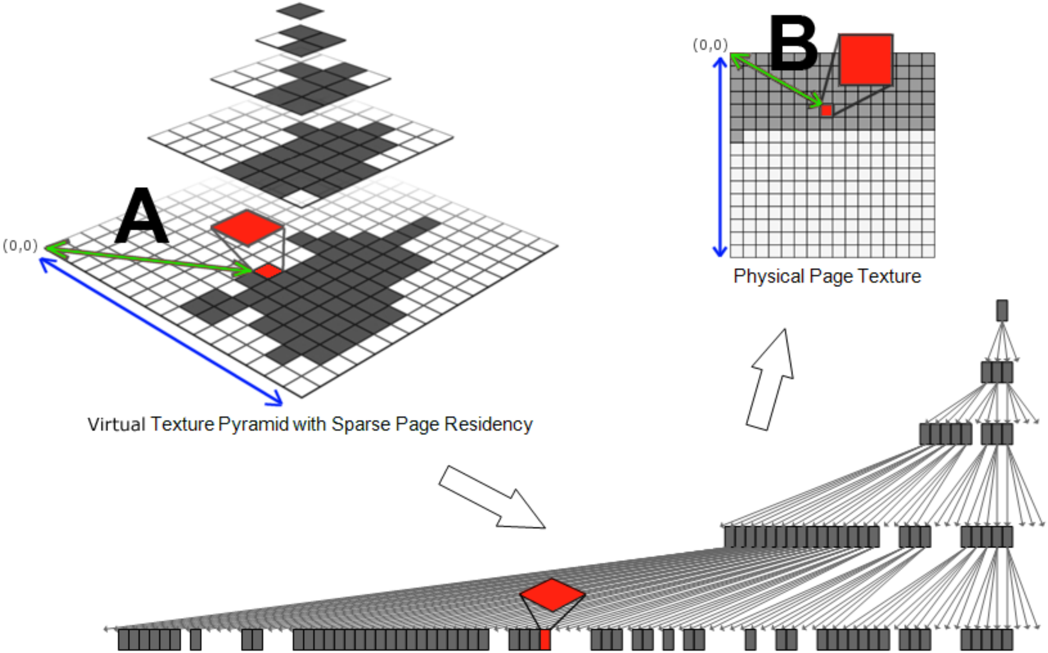


Physical Page Texture



Quad-tree of Sparse Texture Pyramid

Flexible Address Translation

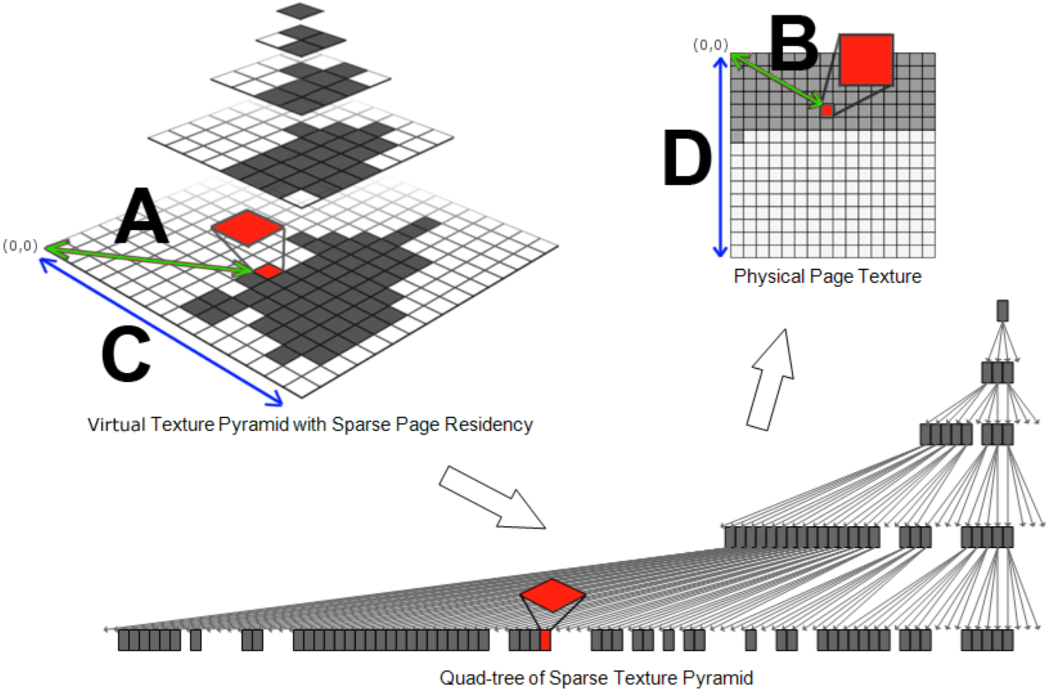


Virtual Texture Pyramid with Sparse Page Residency

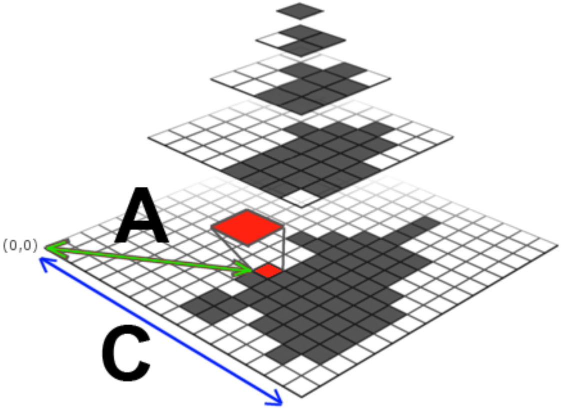
Physical Page Texture

Quad-tree of Sparse Texture Pyramid

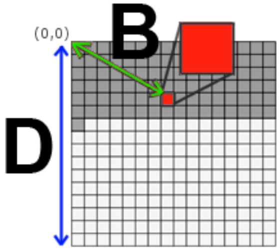
Flexible Address Translation



Flexible Address Translation



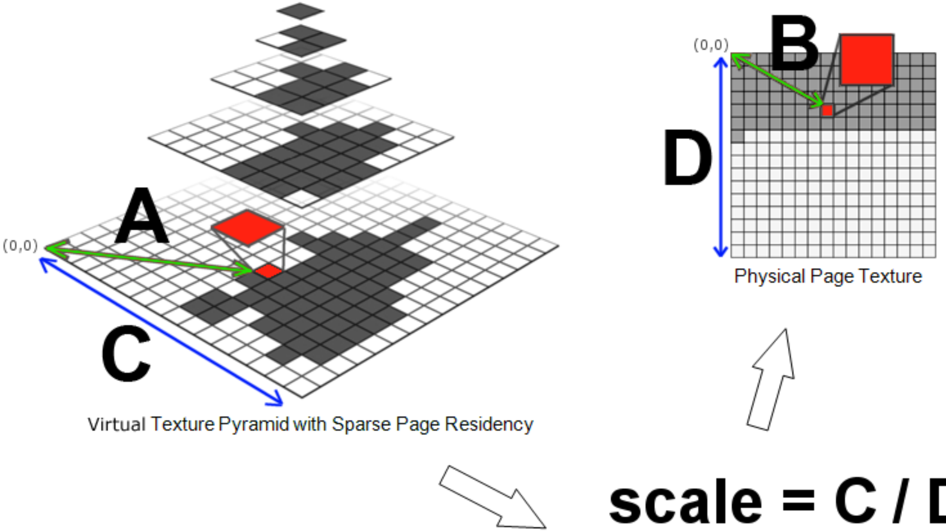
Virtual Texture Pyramid with Sparse Page Residency



Physical Page Texture

physical = (virtual - A) x (C / D) + B

Flexible Address Translation

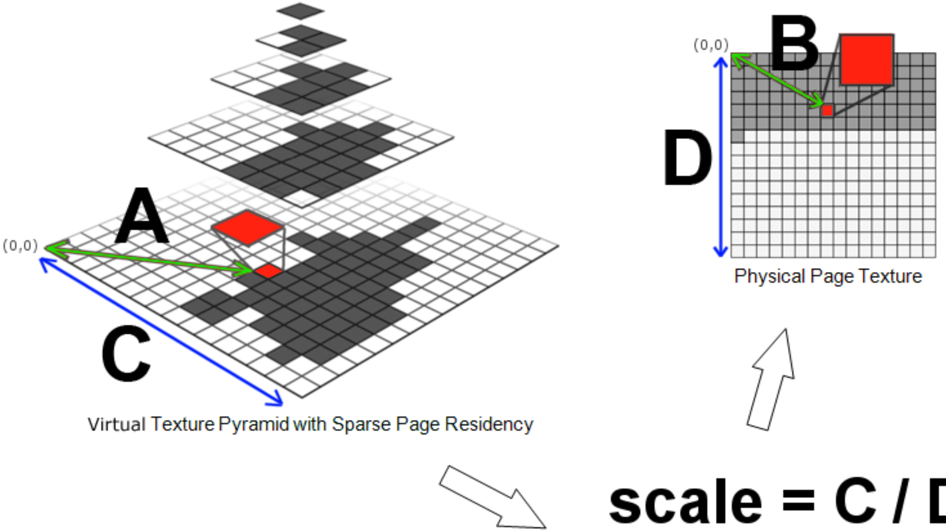


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

scale = C / D

Flexible Address Translation



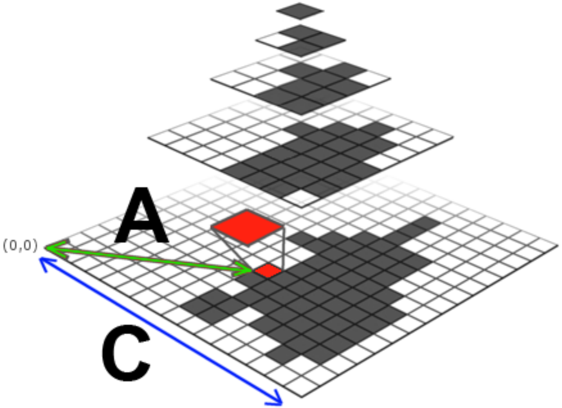
Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

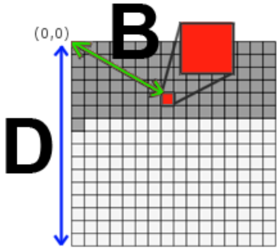
scale = C / D

bias = B – A x scale

Flexible Address Translation



Virtual Texture Pyramid with Sparse Page Residency



Physical Page Texture

scale = C / D

bias = B – A x scale

physical = virtual x scale + bias

Page Tables

- Scale is ratio between virtual mip level size and physical texture size.
- The bias is offset to physical page minus scaled offset to virtual page.
- One scale value if virtual and physical textures are square.
- Two scale values if using non-square virtual or physical texture.
- Two bias values to map virtual pages to arbitrary physical pages.

Page Tables

- Quad-tree
  - Minimal memory footprint.
  - Quad-tree updates are cheap.
  - Dependent lookup for each level accessed.
- Hash table
  - Small memory footprint.
  - Hash table updates are relatively cheap.
  - Need multiple lookups when the desired page is not resident.
- Page table texture
  - Allows texture hardware to be used to directly find the scale & bias for a virtual address.
  - Larger memory footprint because it effectively stores the full quad-tree whether pages are resident or not.
  - Texels for pages that are not resident point to the nearest coarser resident page.
  - May need to update large squares of texels when a page is mapped or unmapped.

Page Table Textures

SIGGRAPH2012

- Store complete quad-tree as a mip-mapped texture.
  - Store full FP32x4 with scale and bias.
  - Encode scale and bias into UINT16x4.
- Use a page table plus mapping texture to store the scale and bias.
  - Store 8:8 page table texture with 1 texel per virtual page.
  - Store FP32x4 mapping texture with 1 texel per physical page.
- Calculate the scale and bias in a fragment program.
  - Store physical page coordinates and base-two logarithm of mip-level width in pages.
  - 8:8:8:8 = X:8 + Y:8 + W:16
  - 5:6:5 = X:5 + W:6 + Y:5
  - Pre DX10 hardware has different conversions from 8-bits to FP32.

Texture Filtering

SIGGRAPH2012

- Bilinear filtering without borders
  - Adjacent virtual pages are not necessarily adjacent in the physical texture.
  - Clamp at border causes objectionable seams at mip level transitions.
- Bilinear filtering with borders
  - Need at least a 1 texel border.
- Trilinear filtering with borders.
  - Mip mapped physical texture.
  - Two address translations.
- Anisotropic filtering with borders.
  - 4-texel border (max aniso = 4)
  - Explicit derivatives + TXD (texgrad)
  - Using implicit derivatives work surprisingly well!

Anisotropic Texture Filtering

- Page table is point sampled.
- Page table lookup unaware of anisotropic lookup that follows.
- May end up with a page that is too coarse.
- Not enough texture detail for the anisotropic texture filter.
- Bias the page table lookup based on the anisotropic footprint.

Anisotropic Page Table Bias

```
float minAnisoBias = -2;                           // - log2( maxAniso = 4 )

float2 dx = ddx( virtualTexCoords.xy );
float2 dy = ddy( virtualTexCoords.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 );
```

Software Virtual Texture Issues

- Memory cost
  - Page table textures can take up a fair amount of memory.
- Performance cost
  - Dependent texture lookup(s) for address translation.
- Texture filtering.
  - Various trade-offs.
  - High quality filtering is still costly.

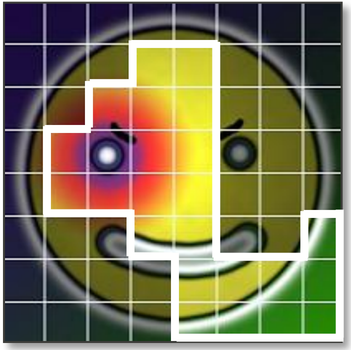
Partially Resident Textures

Juraj Obert
Advanced Micro Devices

SIGGRAPH2012

Partially Resident Textures

- Memory requirements determined by the number of resident tiles, not texture dimensions



RGBA8, 1024x1024, 64 tiles

	Non-PRT	PRT
Memory	4096 kB	1536 kB

Partially Resident Textures

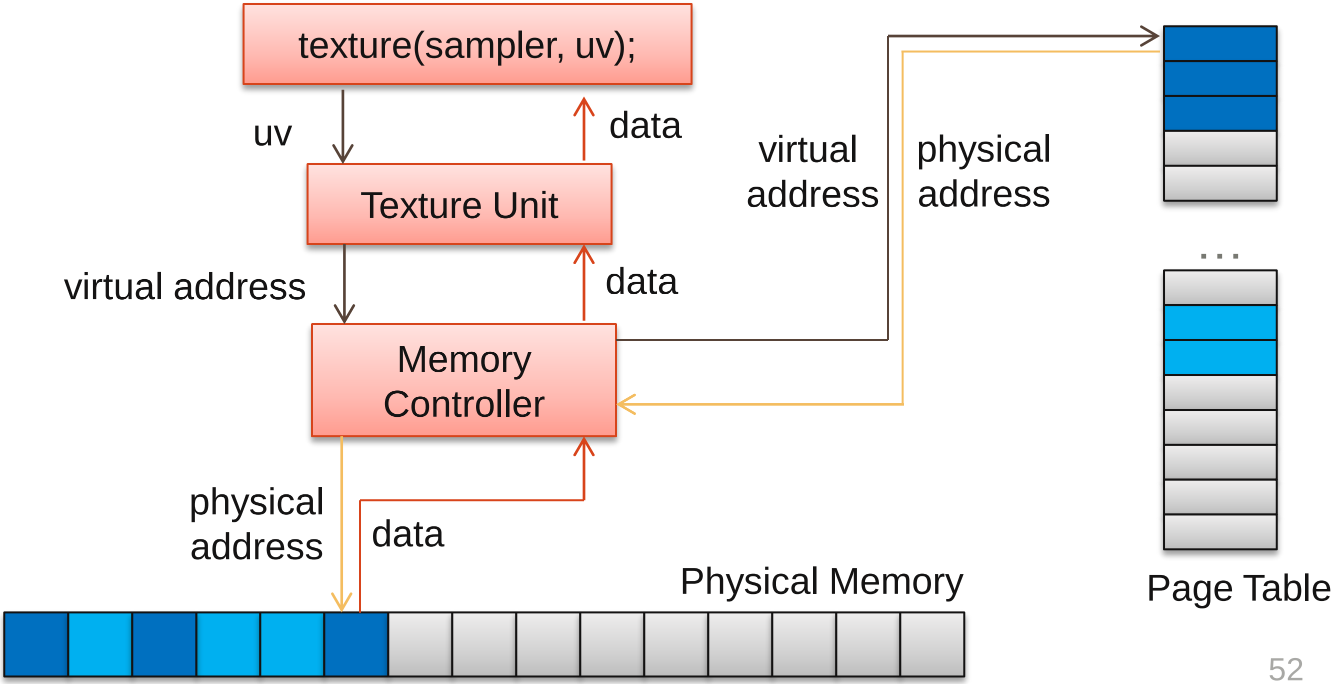
- PRTs rely on 3 core components:
  - Hardware virtual memory subsystem (HW VM)
  - Shader core feedback
  - SW driver stack

Partially Resident Textures – HW VM

SIGGRAPH2012

- Hardware virtual memory
  - Latest generation GPUs use virtual addresses
  - Page table in the on-board GPU memory
  - Address translation entirely in hardware

Partially Resident Textures – HW VM



Partially Resident Textures – HW VM

- Texture Unit
  - UV to virtual address translation
  - Hardware filtering
  - Caching
- Memory Controller
  - Virtual to physical address translation
  - Page table
  - Caching

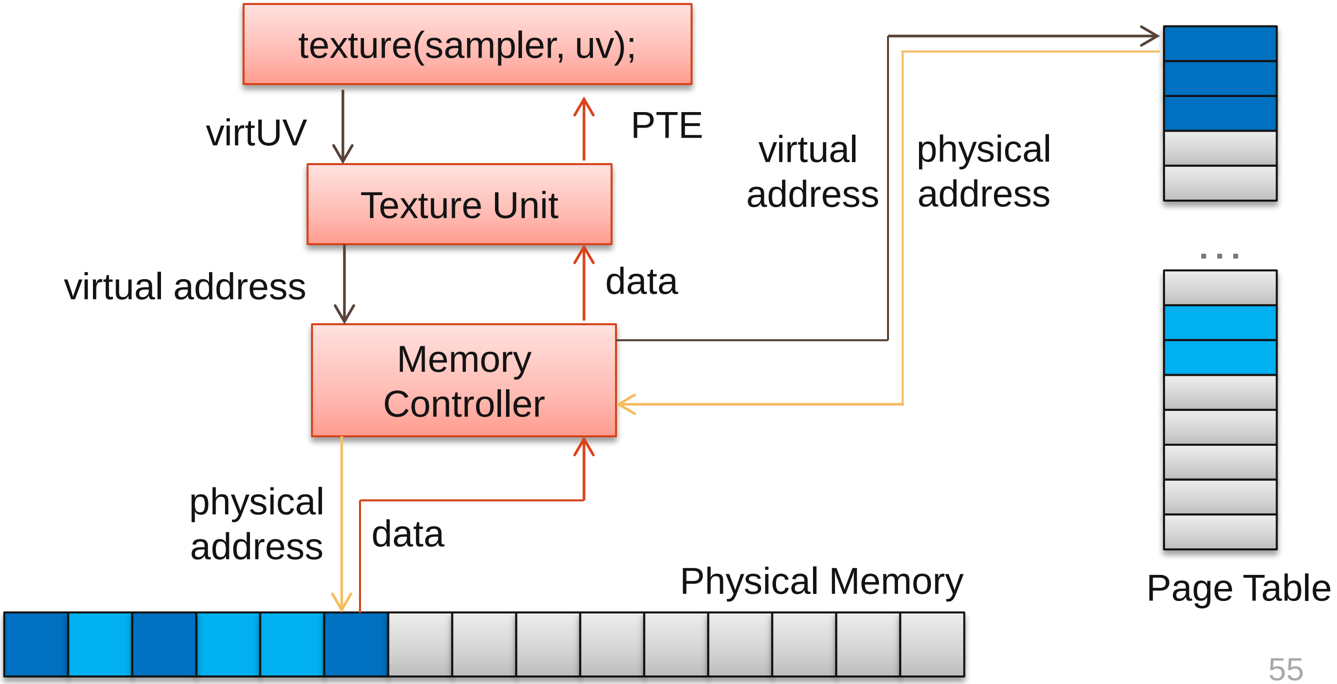
Partially Resident Textures – Shader

SIGGRAPH2012

- SVT texture fetch

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




}
```

Partially Resident Textures – HW VM



Partially Resident Textures – Shader

- SVT texture fetch

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


}
```

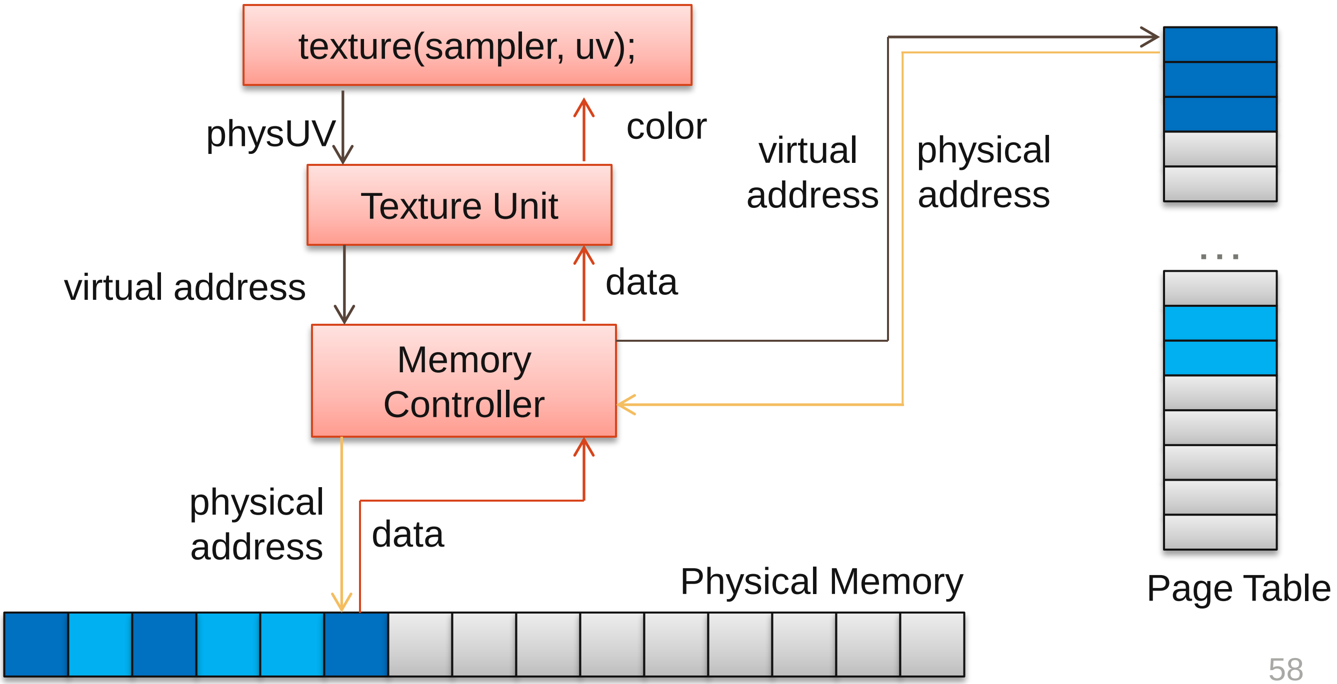
Partially Resident Textures – Shader

SIGGRAPH2012

- SVT texture fetch

```
uniform sampler2D samplerPageTable;                             // page table texture
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
}
```

Partially Resident Textures – HW VM



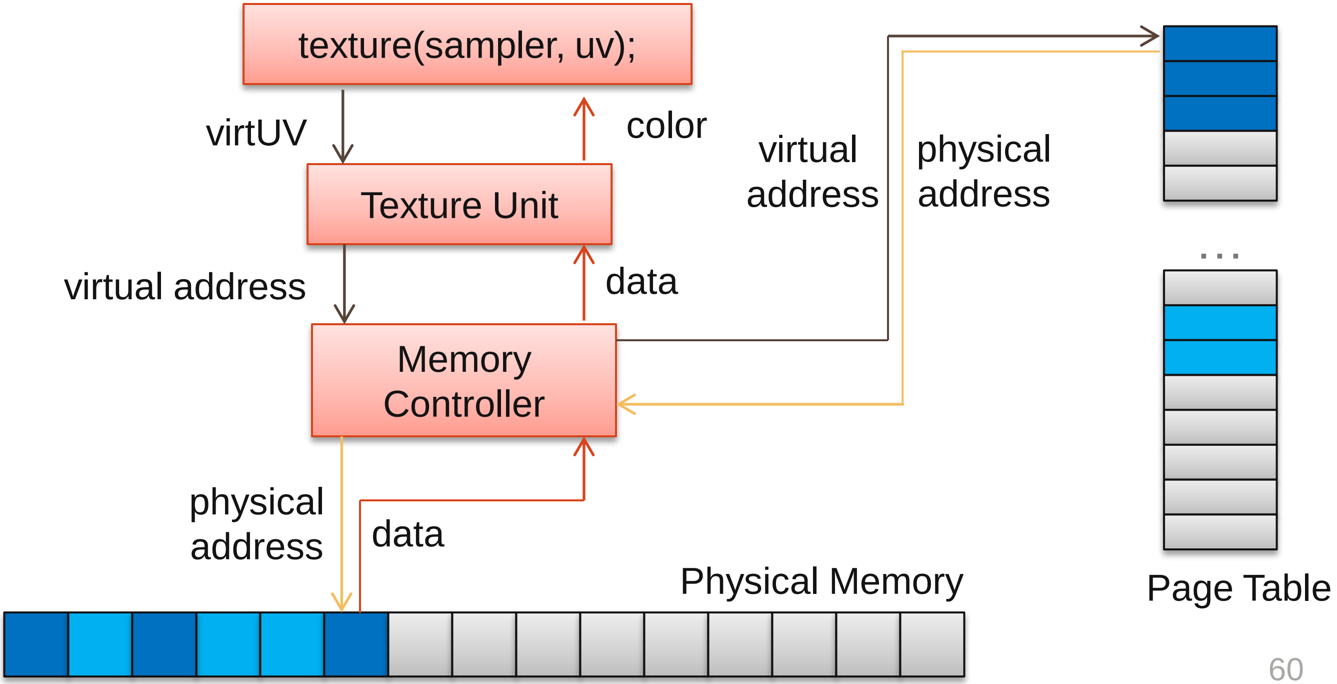
Partially Resident Textures – Shader

- PRT texture fetch

```
uniform sampler2D samplerPRT;                    // partially-resident texture

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



void main()
{


    color = vec4(0.0);

    texture(samplerPRT, virtUV.xy, color);       // 3
}
```

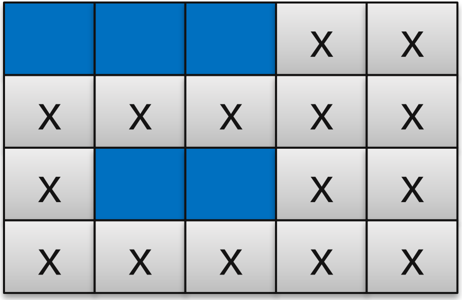
Partially Resident Textures – HW VM

SIGGRAPH2012

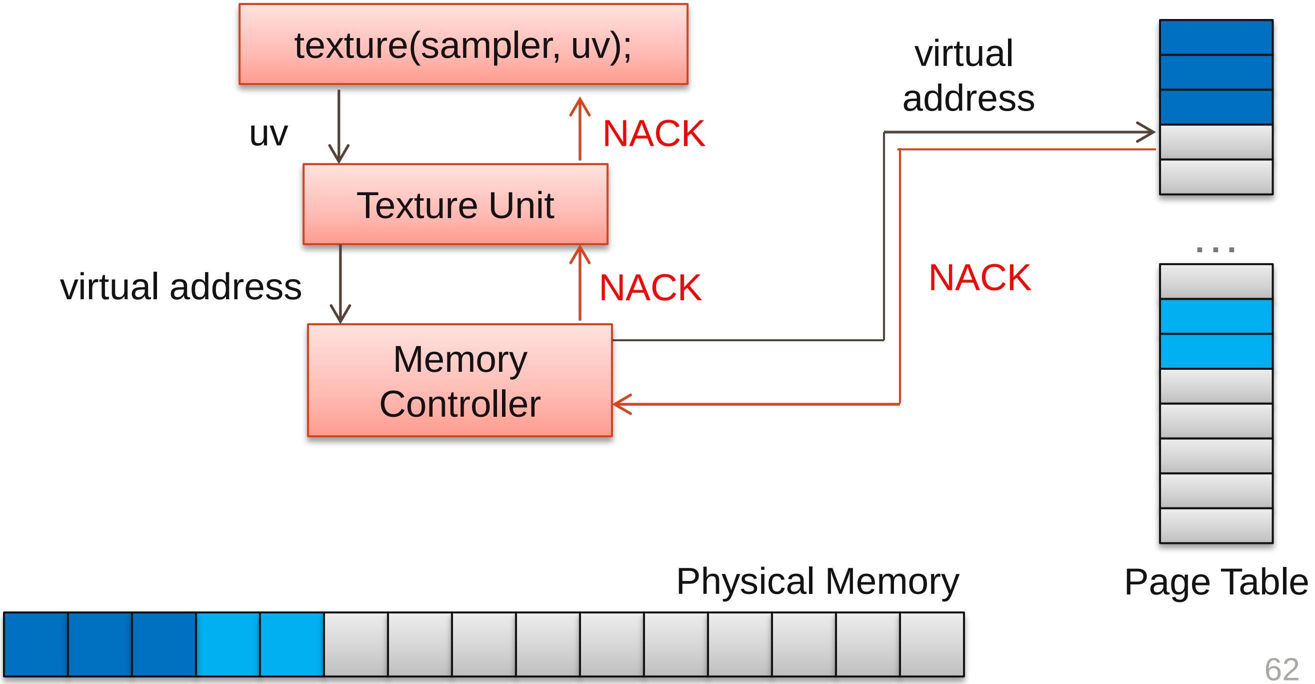


Partially Resident Textures – Shader

- Virtual address space
  - Segmented into 64 kB tiles (pages)
  - Each tile can be either mapped (resident) or unmapped (non-resident)
  - Mapping/unmapping controller by the application/driver



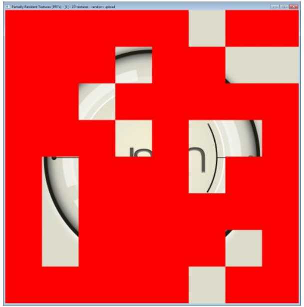
Partially Resident Textures – NACK



Partially Resident Textures – Shader

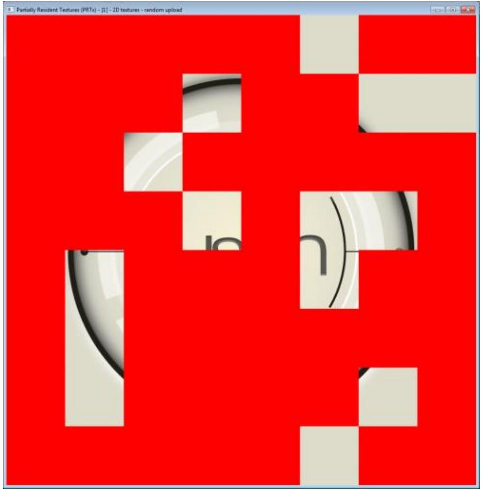
- NACKs in shaders

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

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

    if (code == 0)
    {
        // data resident
        gl_FragColor = vec4(outColor.rgb, 1.0);
    }
    else
    {
        // NACK
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
    }
}
```



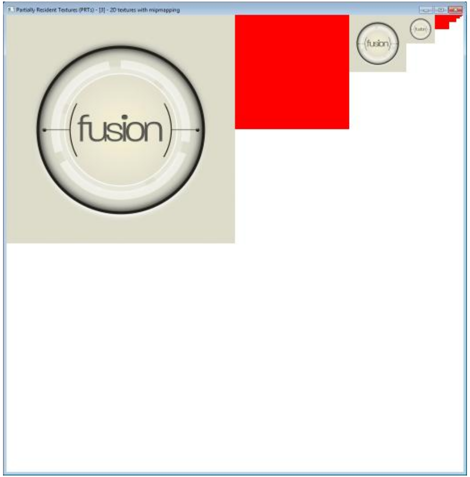
Partially Resident Textures – Shader

- What can be sparse?
  - Any tile-aligned sub-rectangle of a texture mipmap level



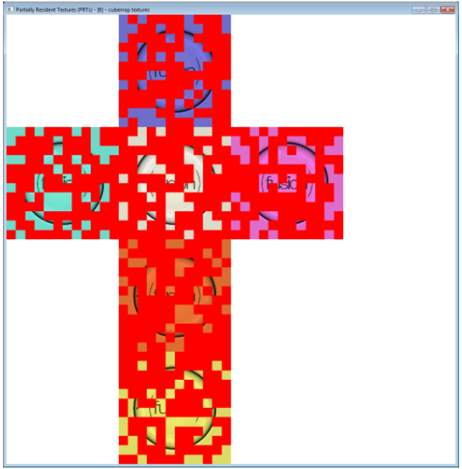
Partially Resident Textures – Shader

- What can be sparse?
  - An entire mipmap level



Partially Resident Textures – Shader

- What can be sparse?
  - Any part of any cubemap face (anyone ever used the bottom cubemap face for anything useful?)



Partially Resident Textures – Shader

- What can be sparse?
  - And any combination of everything just mentioned

- One limitation
  - Everything needs to be tile-aligned

Partially Resident Textures – Driver

- Driver SW stack functionality
  - Create/destroy partially resident resources
  - Map/unmap individual tiles
  - Back virtual allocations by physical memory

Partially Resident Textures – Driver

SIGGRAPH2012

- Backing storage
  - A set of physical allocations managed by the driver
  - The goal is to find balance between the number of resources and unused physical memory
  - Each application has different requirements

Partially Resident Textures – Driver

0	1	2	3	X	8	9	10	11	X
16	17	18	19	X	20	21	22	23	X
12	13	14	15	X	28	29	30	31	X
X	X	X	X	X	X	X	X	X	X
4	5	6	7	X	X	X	X	X	X
X	X	X	X	X	X	X	24	25	X
X	X	X	X	X	X	X	26	27	X
X	X	X	X	X	X	X	X	X	X

PRT

Chunk 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Chunk 2	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Partially Resident Textures – Driver

0	1	2	3	x	8	9	10	11	x
16	17	18	19	x	20	21	x	x	x
12	13	14	15	x	28	29	x	x	x
x	x	x	x	x	x	x	x	x	x
4	5	6	7	x	x	x	x	x	x
x	x	x	x	x	x	x	24	25	x
x	x	x	x	x	x	x	26	27	x
x	x	x	x	x	x	x	x	x	x

PRT

Chunk 1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Chunk 2	16	17	18	19	20	21	x	x	24	25	26	27	28	29	x	x

Partially Resident Textures

SIGGRAPH2012

- Summary

	SVTs	PRTs
Address translation	Shader code	HW page table

Partially Resident Textures

- Summary

	SVTs	PRTs
Address translation	Shader code	HW page table
Filtering	HW + shader code	HW only

Partially Resident Textures

- Summary

	SVTs	PRTs
Address translation	Shader code	HW page table
Filtering	HW + shader code	HW only
# of texture fetches	2, dependent	1

Partially Resident Textures

- Summary

	SVTs	PRTs
Address translation	Shader code	HW page table
Filtering	HW + shader code	HW only
# of texture fetches	2, dependent	1
Supported formats	The ones implemented	All supported by HW

Partially Resident Textures

- Summary

	SVTs	PRTs
Address translation	Shader code	HW page table
Filtering	HW + shader code	HW only
# of texture fetches	2, dependent	1
Supported formats	The ones implemented	All supported by HW
Supported texture types	The ones implemented	All supported by HW

Sparse Textures in OpenGL

Graham Sellers
Advanced Micro Devices

SIGGRAPH 2012

OpenGL Extension

SIGGRAPH2012

- GL_AMD_sparse_texture
- Major design goals:
  - Minimally invasive to the OpenGL API
  - Easy to retrofit into existing application
  - Plays well with non-sparse textures
  - Easy fallback path

OpenGL Extension

SIGGRAPH2012

- Most of the same code will work in the absence of the extension
- Two parts to the extension
  - Update to the API – 1 function, a hand full of tokens
  - Update to the shading language

Example Using Existing OpenGL API

- Use of immutable texture storage

```
GLuint tex;

glGenTextures(1, &tex);
glBindTexture(GL_TEXTURE_2D, tex);
glTexStorage2D(GL_TEXTURE_2D, 10, GL_RGBA8, 1024, 1024);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 1024, 1024,
                GL_RGBA, GL_UNSIGNED_BYTE, data);
```

- Existing OpenGL immutable storage API
  - Declare storage, specify image data

Example Using New Extension

SIGGRAPH2012

- Use of sparse texture storage

```
GLuint tex;

glGenTextures(1, &tex);
glBindTexture(GL_TEXTURE_2D, tex);
glTexStorageSparseAMD(GL_TEXTURE_2D, GL_RGBA, 1024, 1024, 1,
                    1, GL_TEXTURE_STORAGE_SPARSE_BIT_AMD);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 1024, 1024,
             GL_RGBA, GL_UNSIGNED_BYTE, data);
```

- glTexStorageSparseAMD is the one new function in the extension

Control Residency With Existing API

- Previous example used glTexSubImage2D
  - Upload sub-region of the texture
  - Physical pages allocated on demand by the OpenGL driver
  - Unused pages remain free

Control Residency With Existing API

- Allocate disjoint chunks

```
glTexStorageSparseAMD(GL_TEXTURE_2D, GL_RGBA, 1024, 1024, 1,
                      10, GL_TEXTURE_STORAGE_SPARSE_BIT_AMD);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 256, 256,
                GL_RGBA, GL_UNSIGNED_BYTE, data1);
glTexSubImage2D(GL_TEXTURE_2D, 0, 768, 768, 256, 256,
                GL_RGBA, GL_UNSIGNED_BYTE, data2);
```

  - Enough storage for two 256x256 regions allocated

Control Residency With Existing API

- Pass NULL to glTexSubImage2D

```
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 256, 256,
                GL_RGBA, GL_UNSIGNED_BYTE, NULL);
```

  – Makes pages non-resident
  – Driver returns physical pages to the pool

Determining Page Sizes

- Sparse Textures rely on VM subsystem
  - Pages are 64 kilobytes in size on Southern Islands
  - Size of a page in texels depends on texture format

BPP	Texels
128	4096
64	8192
32	16384
16	32768
8	65636

BPP	Tile Width	Tile Height
128	64	64
BC2/3/5/6H/7	256	256
64	128	64
BC1/4	512	256
32	128	128
16	256	128
8	256	256

Getting Page Sizes from OpenGL

- Reuse existing API: glGetInternalFormativ

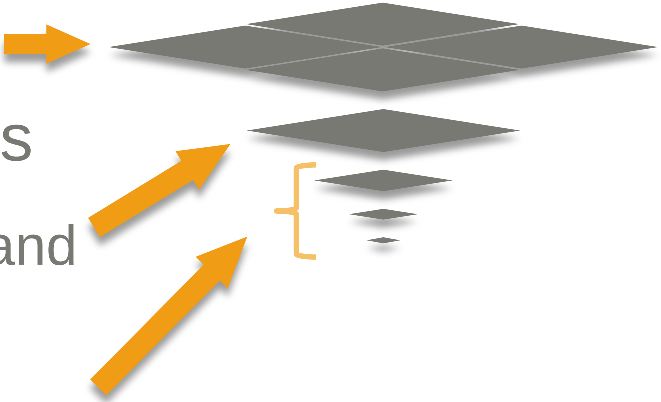
```
GLint page_size_x;

glGetInternalFormativ(GL_TEXTURE_2D, GL_RGBA8,
                      GL_VIRTUAL_PAGE_SIZE_{X,Y,Z}_AMD,
                      sizeof(GLint), &page_size_{x,y,z});
```

  - Given a target and format, returns the page size

- It is not necessary to create a texture to get this information

Mipmaps

- Each LOD requires a different number of pages
  - Each LOD requires fewer and fewer pages
  - Eventually, one LOD does not fill a page
  - Now what?



Low Resolution Mipmaps

- Eventually, we make all LODs resident
  - Use glGetInternalFormativ to retrieve the lowest sparse level for a given target/format

```
GLint min_sparse_level;

glGetInternalFormativ(GL_TEXTURE_2D, GL_RGBA16F,
                      GL_MIN_SPARSE_LEVEL_AMD,
                      1, &min_sparse_level);
```

  - All levels below this reside in the same page and share residency

LOD Warnings

SIGGRAPH2012

- A per-texture low water mark is included
  - Set this to lowest LOD that’s fully resident
  - When this is hit, the shader is signaled
  - Returned data is still valid
  - Start streaming the next mip
- Exposed using the glTexParameter API

Setting the LOD Watermark

SIGGRAPH2012

- Exposed using the glTexParameter API

```
glTexParameteri(GL_TEXTURE_2D, GL_MIN_WARNING_LOD_AMD, 4);
```

  – Here, an LOD warning will be returned to the shader if hardware attempts to access LOD 4 or lower

- More on residency returns later...

Rendering to a Sparse Texture

- Render to a texture using an FBO

```
GLuint prt, fbo;

glGenTextures(1, &prt);
glBindTexture(GL_TEXTURE_2D, prt);
glTexStorageSparseAMD(GL_TEXTURE_2D, GL_RGBA, 1024, 1024,
                      1, 1, GL_TEXTURE_STORAGE_SPARSE_BIT_AMD);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 1024, 1024, GL_RGBA, GL_UNSIGNED_BYTE, data);
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, prt, 0);
```

  - Writes to unmapped regions are silently dropped

Reading From a Sparse Texture

- Read data to memory using existing APIs
  - Call glGetTexImage to read entire content

```
glGetTexImage(GL_TEXTURE_2D, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

  - Bind to FBO, use glReadPixels or glBlitFramebuffer

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, prt, 0);
glReadPixels(0, 0, 1024, 1024, GL_RGBA, GL_UNSIGNED_BYTE, data);
glBlitFramebuffer(0, 0, 1024, 1024, 0, 0, 128, 128, GL_COLOR_BUFFER_BIT, GL_LINEAR);
```

Restrictions (or Lack Thereof)

- Sparse textures have some restrictions:
  - Dimensions of the base level must be integer multiples of the page size
    - This means... no sparse textures below this size
  - No buffer textures or “TBOs”
  - No depth or stencil textures, nor MSAA textures

Managing Failure

- Virtual address space is extremely large
  - It will run out eventually, but it'll take a while
  - It's still possible to run out of physical memory
  - glTexSubImage2D etc., may fail
  - Draw calls may fail

Managing Failure

- Physical memory is a limited resource
  - Feel free to create a 4k x 4k x 4k volume
  - Don't try to make it all resident at once!
- There are no sparse read-backs
  - glGetTexImage could read gigabytes of data
  - This will fail

Sparse Textures in Shaders

- Texture type in GLSL is the ‘sampler’
- Several types of samplers exist...
  - sampler2D, sampler3D, samplerCUBE, sampler2DArray, etc.
- We didn’t add any new sampler types
  - Sparse and normal textures use the same types

Sparse Textures in Shaders

- Read textures using ‘texture’
  - Built-in function, with several overloads

```
gvec4 texture(gsampler1D sampler, float P [, float bias]);
gvec4 texture(gsampler2D sampler, vec2 P [, float bias]);
gvec4 texture(gsampler2DArray sampler, vec3 P [, float bias]);
gvec4 textureLod(gsampler2D sampler, vec2 P, float lod);
gvec4 textureProj(gsampler2D sampler, vec4 P [, float bias]);
gvec4 textureOffset(gsampler2D sampler, vec2 P, ivec2 offset [, float bias]);
// ... etc.
```

  - We didn’t add any new overloads

Extending GLSL

- Adding new function overloads is difficult
  - Need to return a status code and a texel
  - Need user-specified defaults with conditional move like functionality
  - Optional parameters in existing overloads made this very difficult

Extending GLSL

- Added new built-in functions
  - Return both a status code and texel data:

```
int sparseTexture(gsampler2D sampler, vec2 P, inout gvec4 texel [, float bias]);
int sparseTextureLod(gsampler2D sampler, vec2 P, float lod, inout gvec4 texel);
// ... etc.
```

  - Most existing texture functions have a sparseTexture equivalent
  - Non-sparse textures work with new functions

Extending GLSL | sparseTexture

SIGGRAPH2012

- sparseTexture returns two pieces of data:

```
int sparseTexture(gsampler2D sampler, vec2 P, inout gvec4 texel [, float bias]);
```

  – Texel data via inout parameter
  – Residency status code

Extending GLSL | sparseTexture

- Texel data returned in inout parameter
  - If texel fetch fails, old data remains in variable
  - Think of it as a CMOV type operation
- Return code is hardware-dependent
  - More built-in functions for decoding status codes

Extending GLSL | sparseTexture

- No direct support for ‘default value’
  - But this can be emulated easily:

```
vec4 texel = vec4(1.0, 0.0, 0.7, 1.0); // Default value

sparseTexture(s, texCoord, texel);

// On success, texel contains texture data. On failure, it has the shader-supplied
// default value in it (pinkish magenta here).
```

Extending GLSL | sparseTexture

- Original functions work on sparse textures

```
vec4 texel = texture(s, texCoord);
```

  – Return value for unmapped regions undefined
  – Useful when residency is predetermined

GLSL | Residency Information

- Residency information returned to shader

```
vec4 texel = vec4(1.0, 0.0, 0.7, 1.0); // Default value
int code;

code = sparseTexture(s, texCoord, texel);
```

- Code is interpreted by additional functions

```
bool sparseTexelResident(int code);
bool sparseTexelMinLodWarning(int code);
int sparseTexelLodWarningFetch(int code);
```

GLSL | Residency Information

SIGGRAPH2012

- Was texel resident?

```
bool sparseTexelResident(int code);
```

  - Returns true if data is valid, false otherwise

GLSL | Residency Information

SIGGRAPH2012

- Was texel resident?

```
bool sparseTexelResident(int code);
```

  - Texel miss is generated if any required sample is not resident, including:
    - Texels required for bilinear or trilinear sampling
    - Missing mip maps, anisotropic filter taps, etc.

GLSL | Low-Water Mark

SIGGRAPH2012

- Did I hit the low-water mark?

```
bool sparseTexelMinLodWarning(int code);
```

  - Occurs when generating a texel requires data from an LOD lower than the low-water mark specified by the application
  - This can be a signal to the application to start streaming more mip levels

GLSL | Low-Water Mark

- What LOD caused the warning?

```
int sparseTexelLodWarningFetch(int code);
```

  – sparseTexelLodWarningFetch returns 0 if the warning was not hit

Sparse Textures – Use Cases

SIGGRAPH2012

- Drop-in replacement for traditional SVT
  - Almost... maximum texture size hasn't grown
- Extremely large texture arrays
  - Only populate a sub-set of the slices
  - Can eliminate texture binds in some applications

Sparse Textures – Use Cases

SIGGRAPH2012

- Large volume textures
  - Voxels, medical applications
  - Use maximum step size as ‘default’ value
- Variable size texture arrays
  - Create a large array texture
  - Populate different mip levels in each slice

Future Work

SIGGRAPH2012

- Planning further extension(s)
  - Application-controlled physical pool
  - Map the same page multiple times
  - Partially resident buffers
    - Streaming geometry
    - Lazy allocation for fragment lists

Demo (RAGE running PRTs)

J.M.P. van Waveren
id Software

Graham Sellers
Advanced Micro Devices

SIGGRAPH2012

PRTs in RAGE

RAGE with PRTs (Image courtesy of id Software)

Discussion

Conclusion

- Partially Resident Textures
  - Hardware implementation of virtual texturing
    - Hardware virtual memory subsystem
    - Shader core feedback
  - OpenGL extension available

- Developer feedback very important

Backup

Partially Resident Textures

SIGGRAPH2012

- Paging
  - The process of making resources resident in GPU-visible memory (for simplicity, assume on-board memory)
  - Handled by the DirectX Graphics Kernel subsystem and the kernel-mode device driver
  - Regular, non-PRT, resources (textures, buffers) paged in/out with resource granularity