GPU TECHNOLOGY CONFERENCE

Using Virtual Texturing to Handle Massive Texture Data

San Jose Convention Center - Room A1 | Tuesday, September, 21st, 14:00 - 14:50

J.M.P. Van Waveren – id Software

Evan Hart – NVIDIA

PRESENTED BY NVIDIA.

How we describe our environment ?

- Polygonal boundary representations
  - convenient / compressed description of the material world
- Tiling / repeating / blending textures
  - primitive forms of texture compression ?

Today

- Polygonal boundary representations
  - convenient / compressed description of the material world
- ~~Tiling / repeating / blending textures~~
  - ~~primitive forms of texture compression ?~~

Tonight ?

- ~~Polygonal boundary representations~~
  - ~~convenient / compressed description of the material world~~
- ~~Tiling / repeating / blending textures~~
  - ~~primitive forms of texture compression ?~~

Unique texture detail

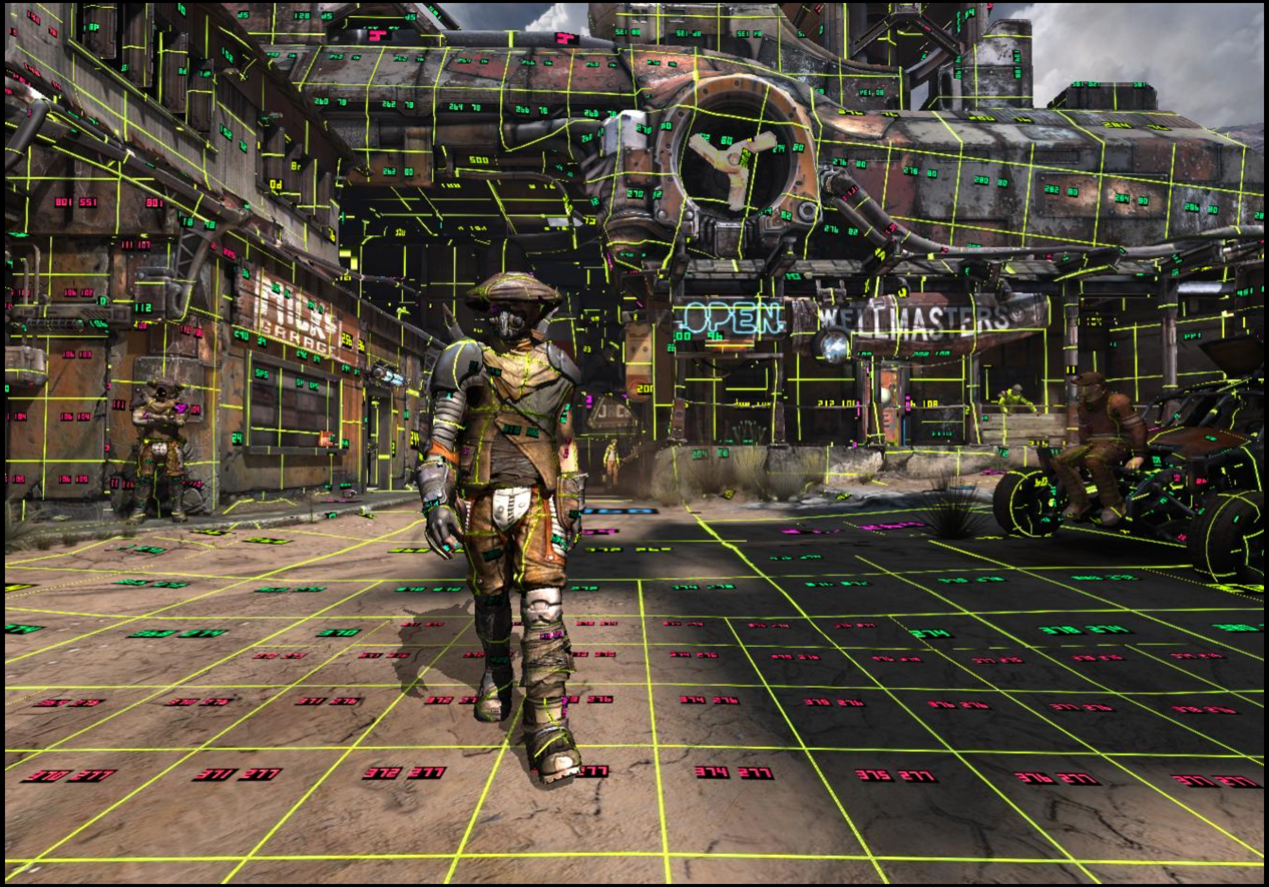
Very large textures

Virtual Texture vs. Virtual Memory

- fall back to blurrier data without stalling execution
- lossy compression is perfectly acceptable

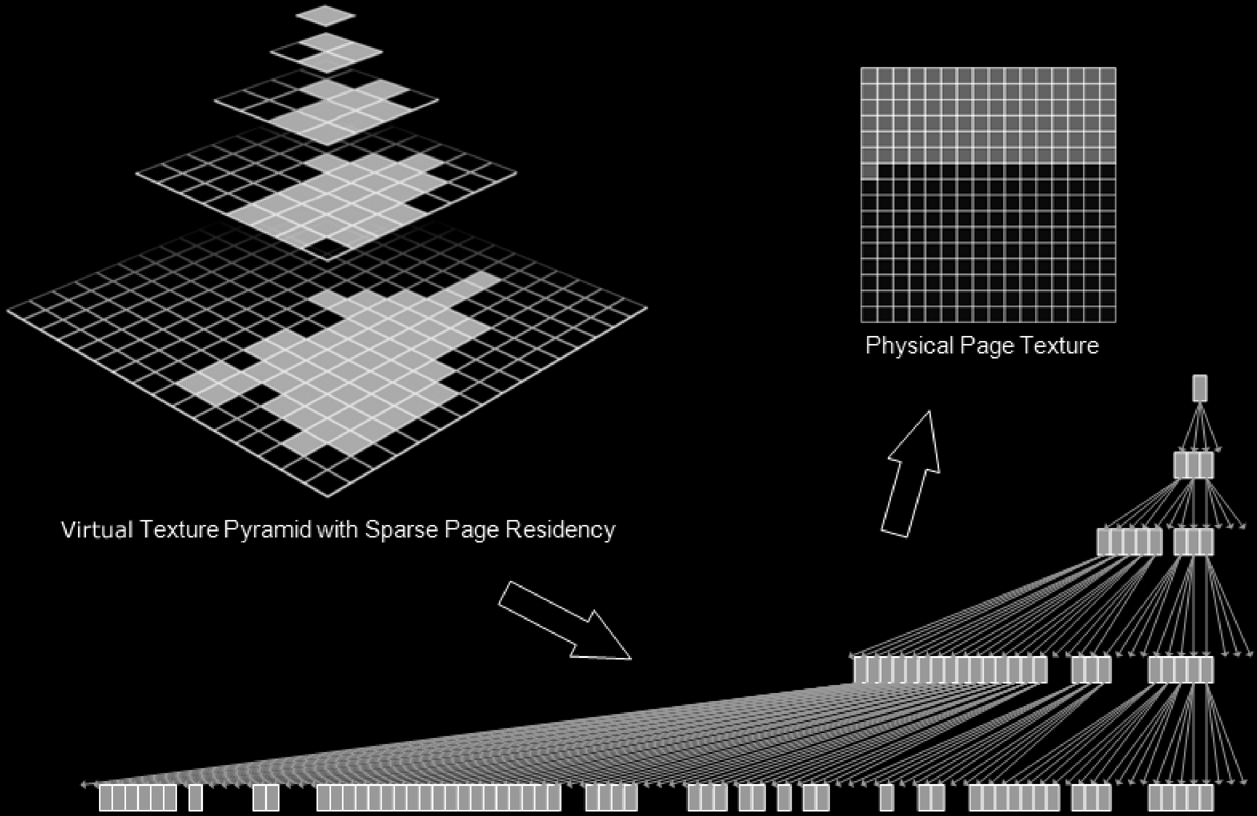
Universally applied virtual textures

Virtual textures with virtual pages



Physical texture with physical pages

Virtual to Physical Translation

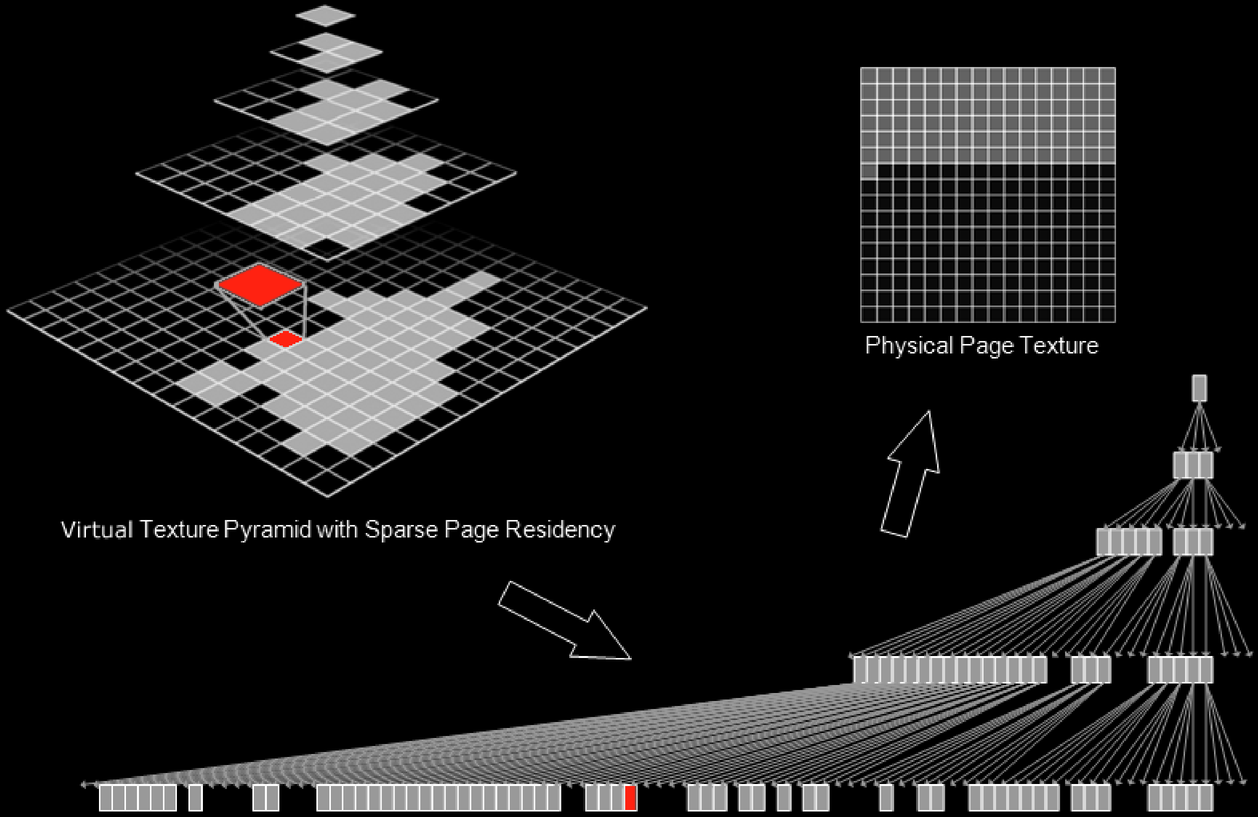


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

Quad-tree of Sparse Texture Pyramid

Virtual to Physical Translation

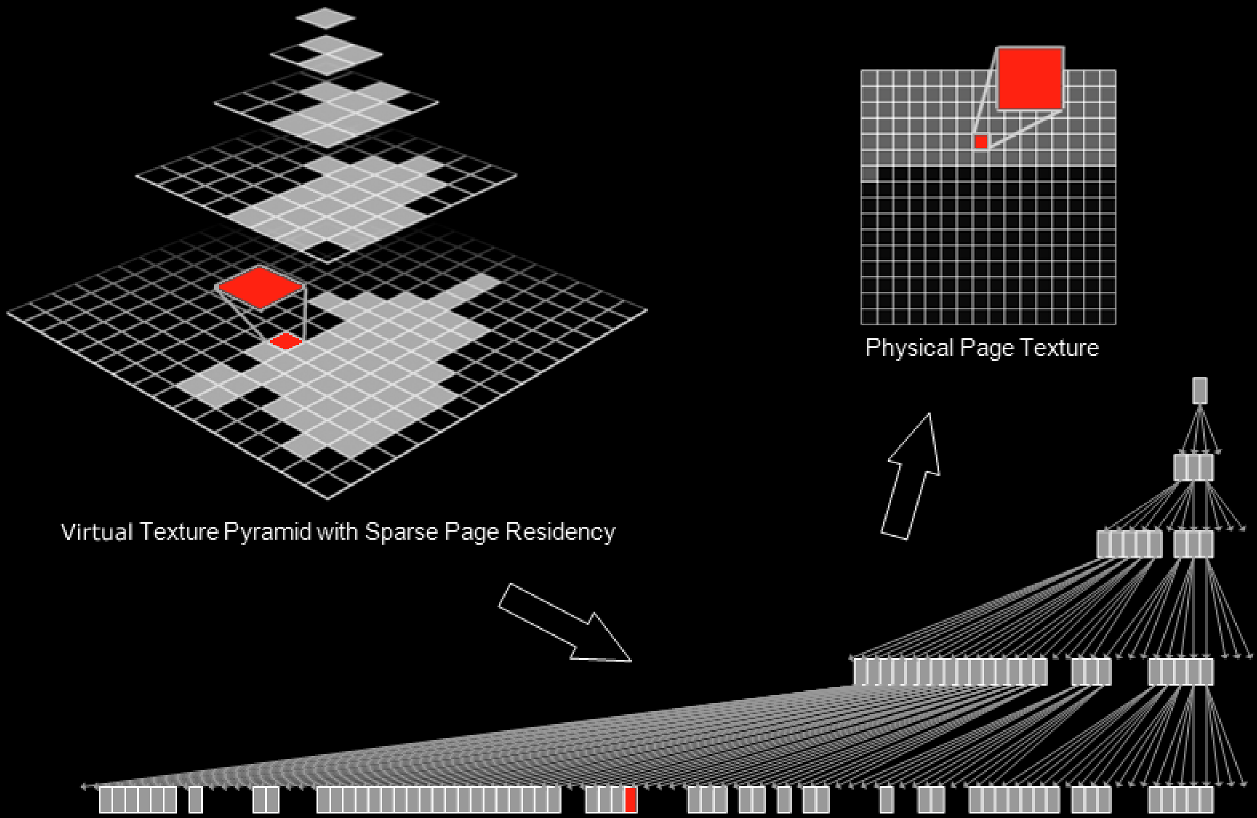


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

Quad-tree of Sparse Texture Pyramid

Virtual to Physical Translation

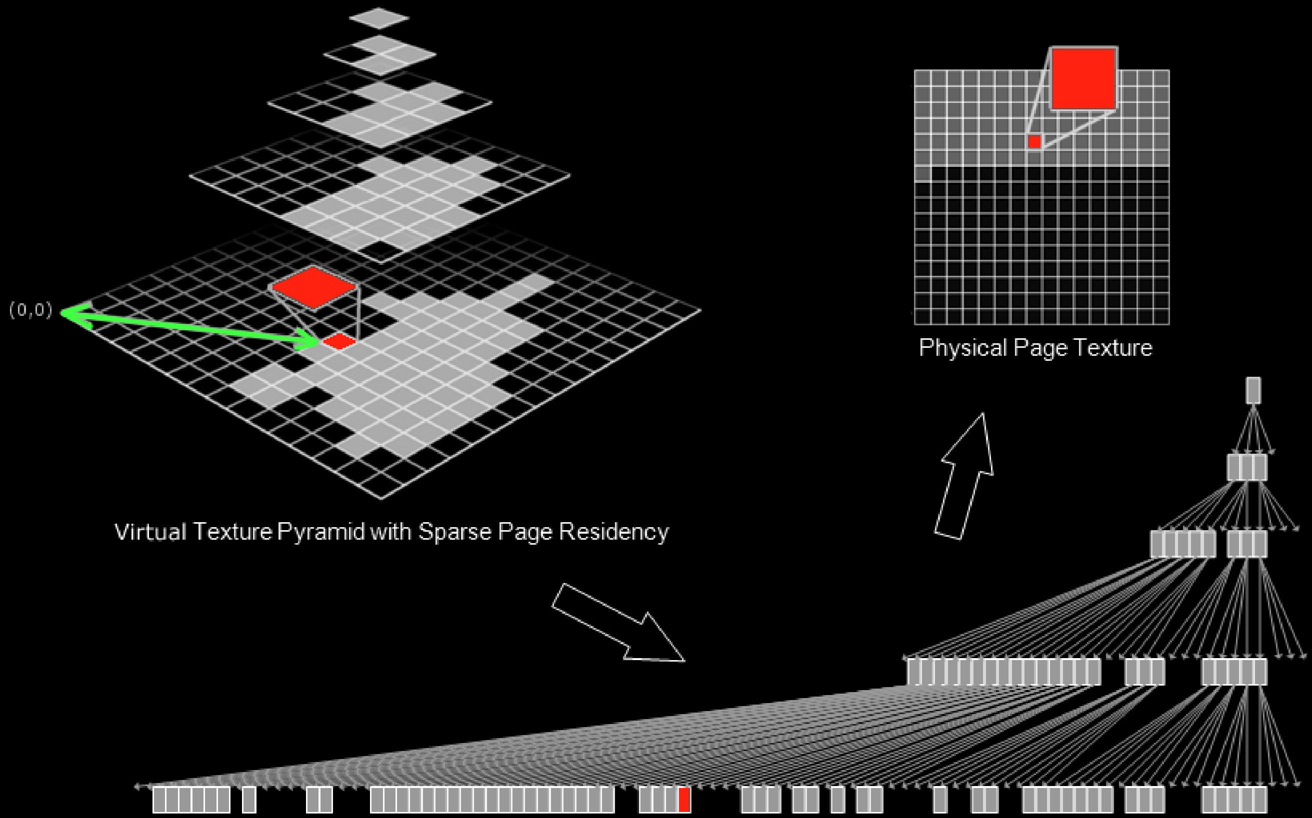


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

Quad-tree of Sparse Texture Pyramid

Virtual to Physical Translation

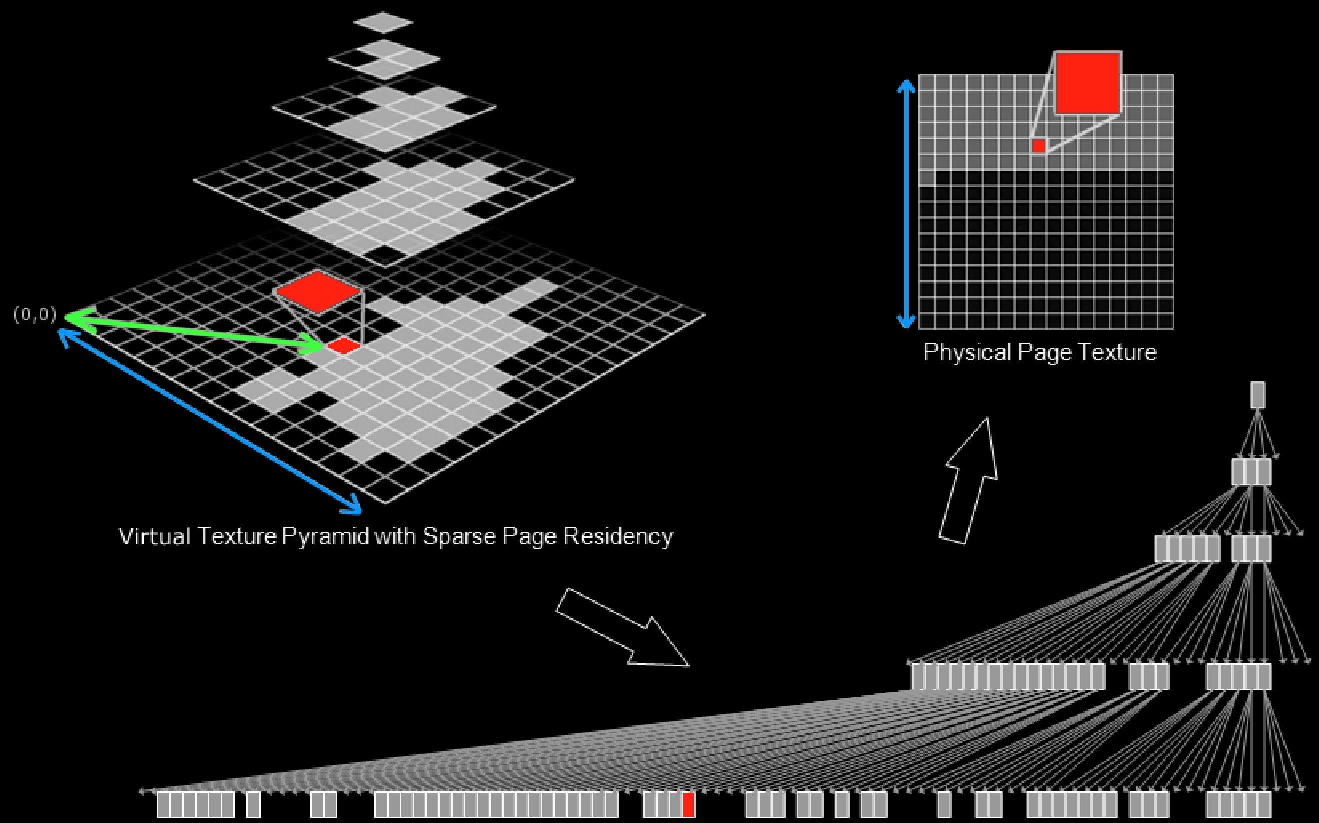


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

Quad-tree of Sparse Texture Pyramid

Virtual to Physical Translation

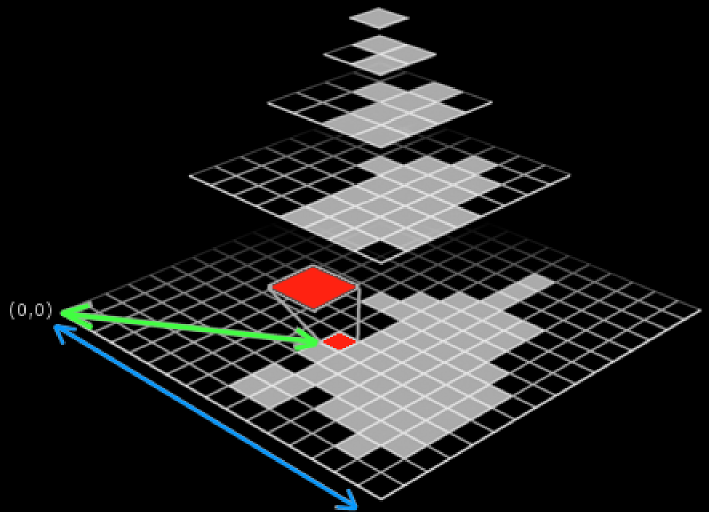


Virtual Texture Pyramid with Sparse Page Residency

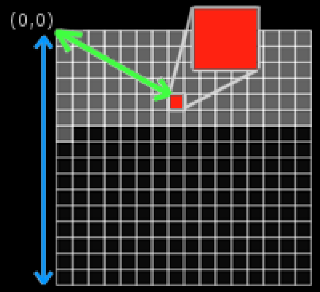
Physical Page Texture

Quad-tree of Sparse Texture Pyramid

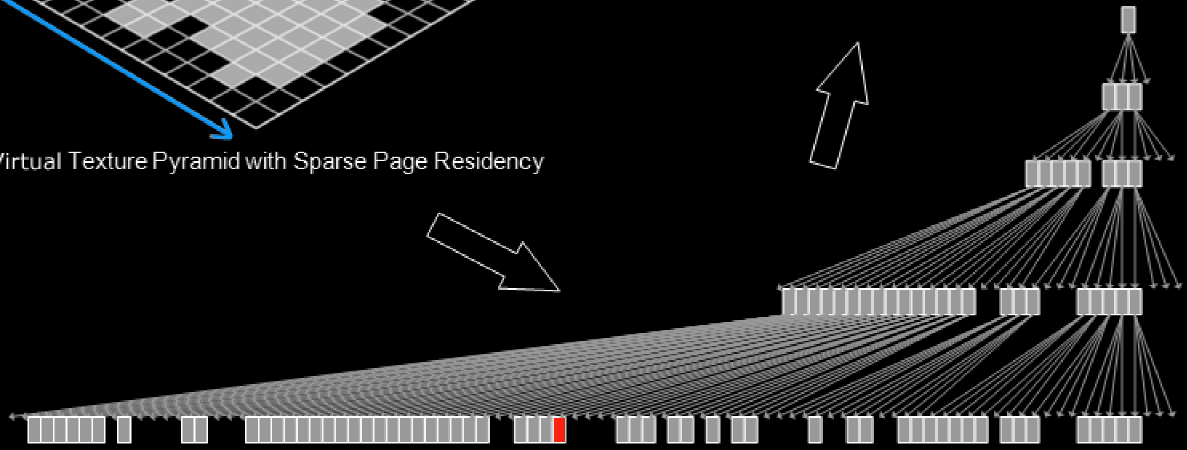
Virtual to Physical Translation



Virtual Texture Pyramid with Sparse Page Residency

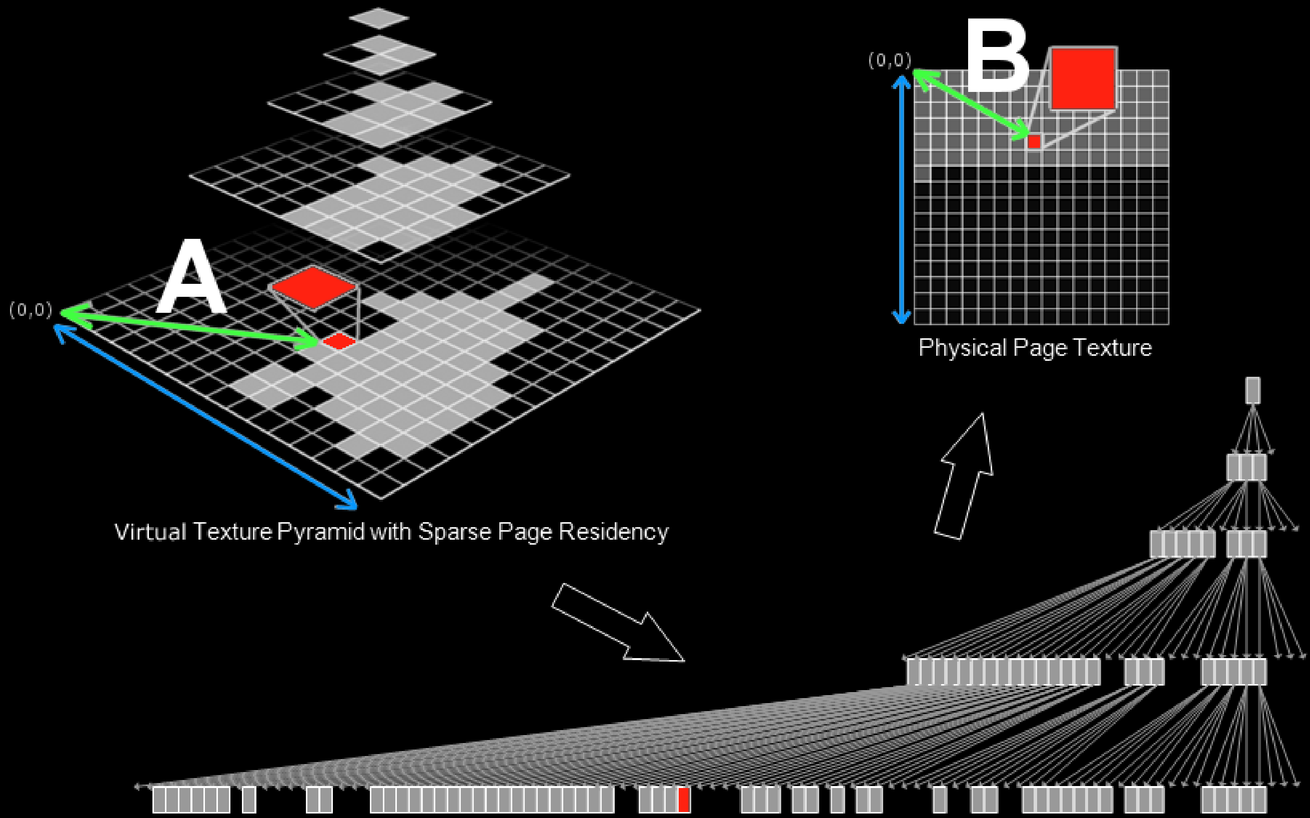


Physical Page Texture



Quad-tree of Sparse Texture Pyramid

Virtual to Physical Translation

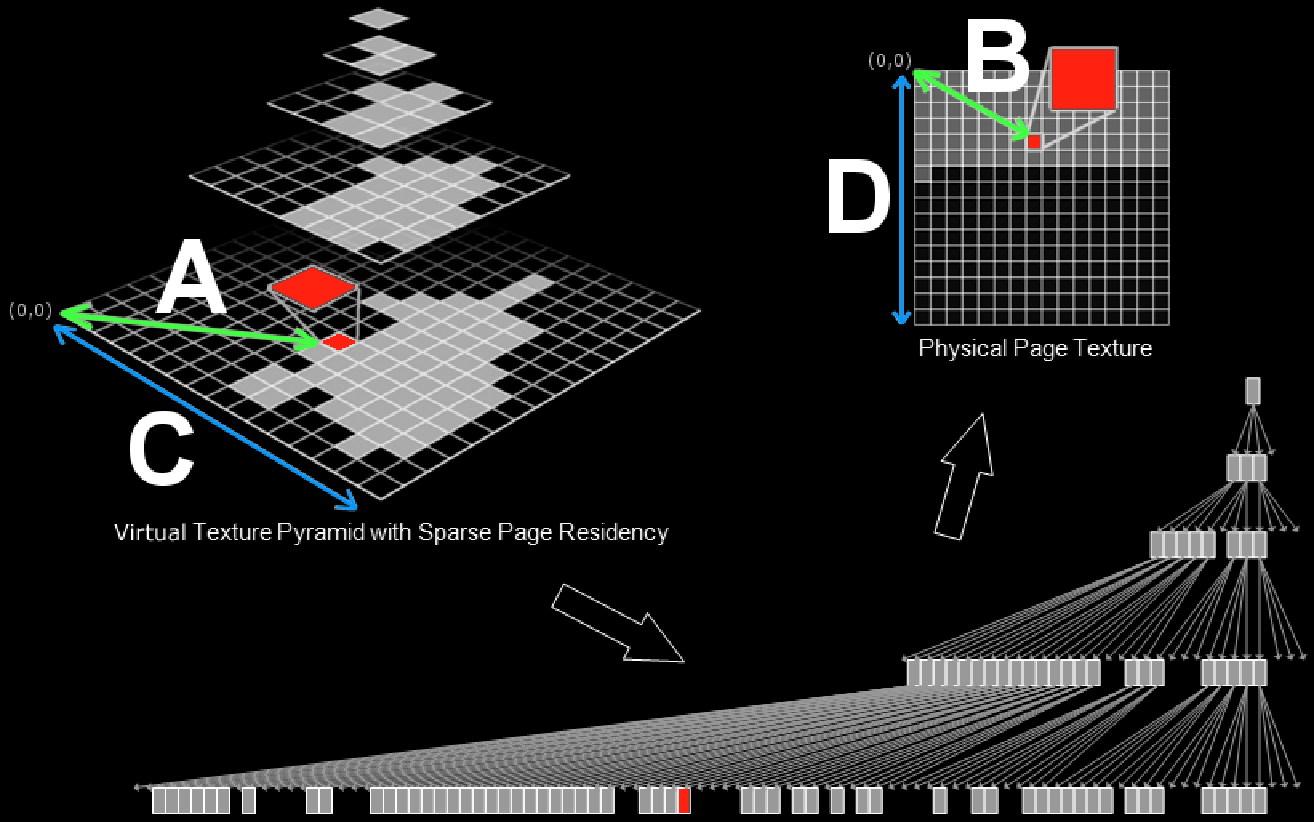


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

Quad-tree of Sparse Texture Pyramid

Virtual to Physical Translation

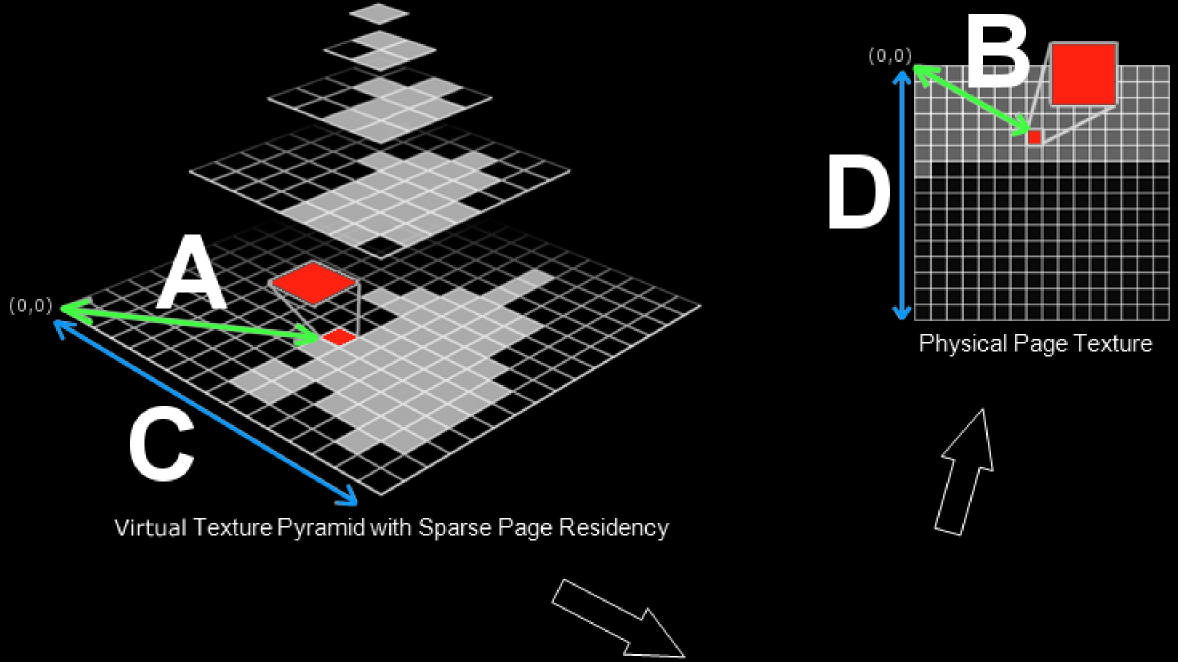


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

Quad-tree of Sparse Texture Pyramid

Virtual to Physical Translation

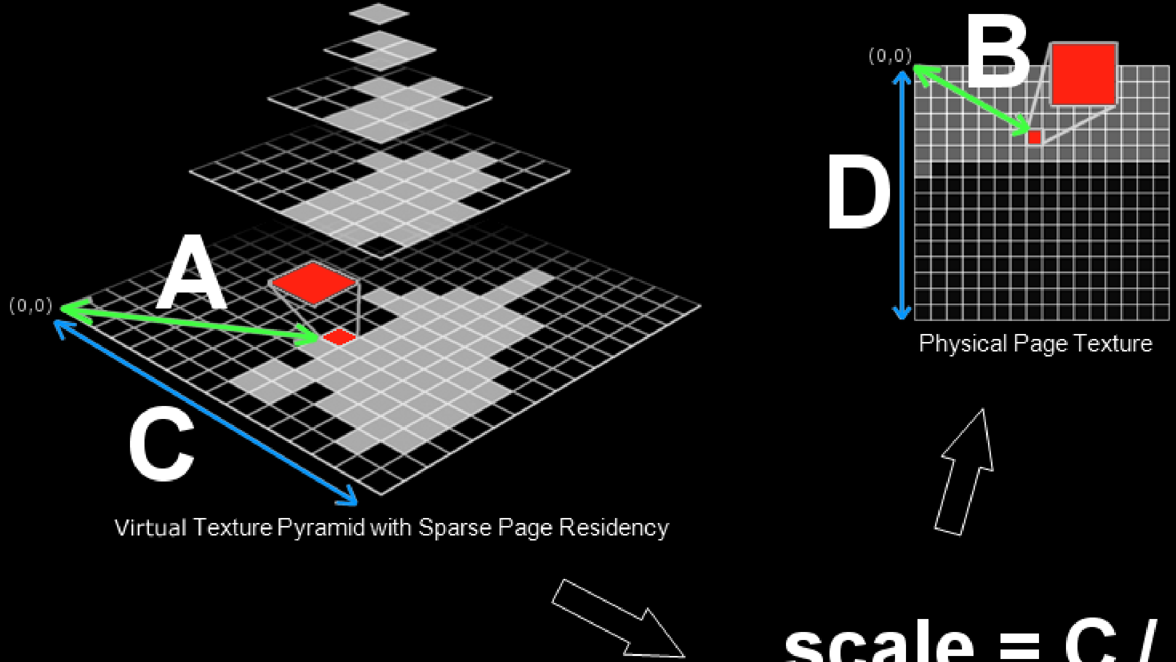


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

$$physical = (virtual - A) \times (C / D) + B$$

Virtual to Physical Translation

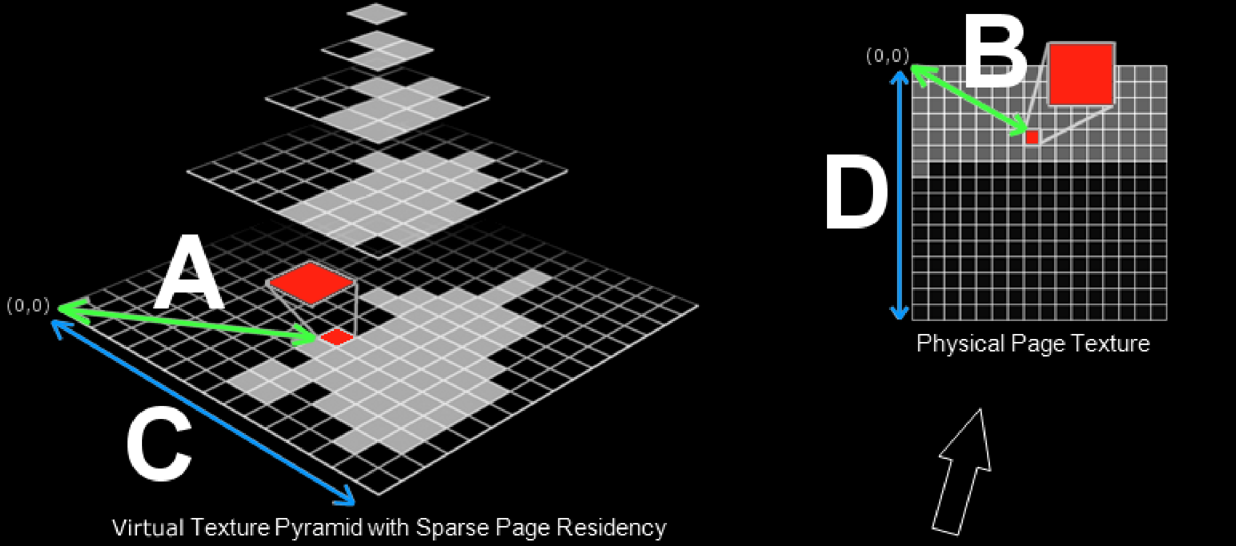


Virtual Texture Pyramid with Sparse Page Residency

Physical Page Texture

scale = C / D

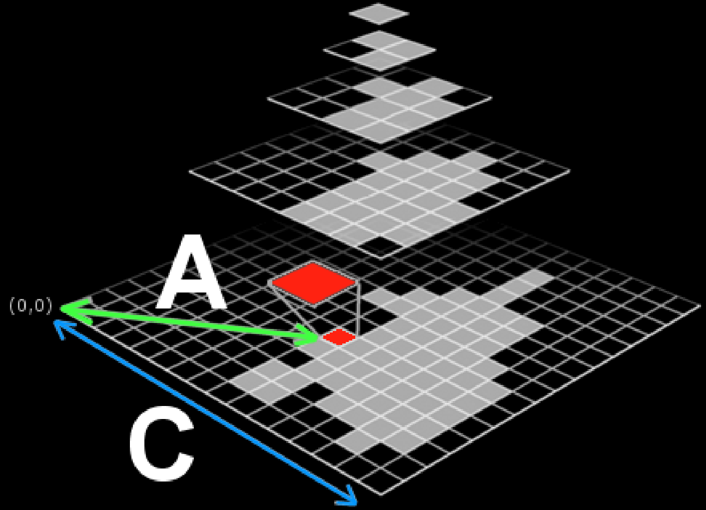
Virtual to Physical Translation



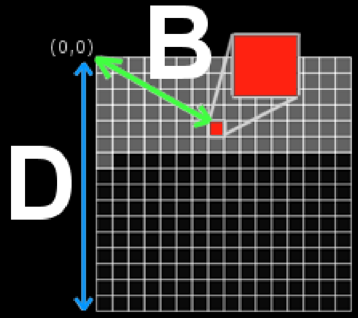
scale = C / D

bias = A – B x scale

Virtual to Physical Translation



Virtual Texture Pyramid with Sparse Page Residency



Physical Page Texture

scale = C / D

bias = A – B x scale

physical = virtual x scale + bias

Optimized virtual to physical translations

- Store complete quad-tree as a mip-mapped texture
  - FP32x4
- Use a mapping texture to store the scale and bias
  - 8:8 + FP32x4
- Calculate the scale and bias in a fragment program
  - 8:8:8:8
  - 5:6:5

Texture Filtering

- Bilinear filtering without borders
- Bilinear filtering with borders
- Trilinear filtering (mip mapped vs. two translations)
- Anisotropic filtering
  - 4-texel border (max aniso= 4)
  - explicit derivatives + TXD (texgrad)
  - implicit derivatives works surprisingly well

Which pages need to be resident?

- Feedback rendering
  - separate rendering pass
  - or combined with depth pass
  - factor 10 smaller is ok
- Feedback analysis
  - run as parallel job on CPU
  - run on the GPU
  - ~ .5 msec on CPU for 80 x 60

How to store huge textures?

- diffuse + specular + normal + alpha + power = 10 channels
  - 128k x 128k x 3 x 8-bit RGBA = 256 GigaBytes
  - DXT compressed (1 x DXT1 + 2 x DXT5) = 53 GigaBytes

- use brute force scene visibility to throw away data
  - down to 20 – 50 GigaBytes uncompressed
  - 4 – 10 GigaBytes DXT compressed

Need variable bit rate compression!

- DCT-based compression
  - 300 – 800 MB
- HD-Photo compression
  - 170 – 450 MB

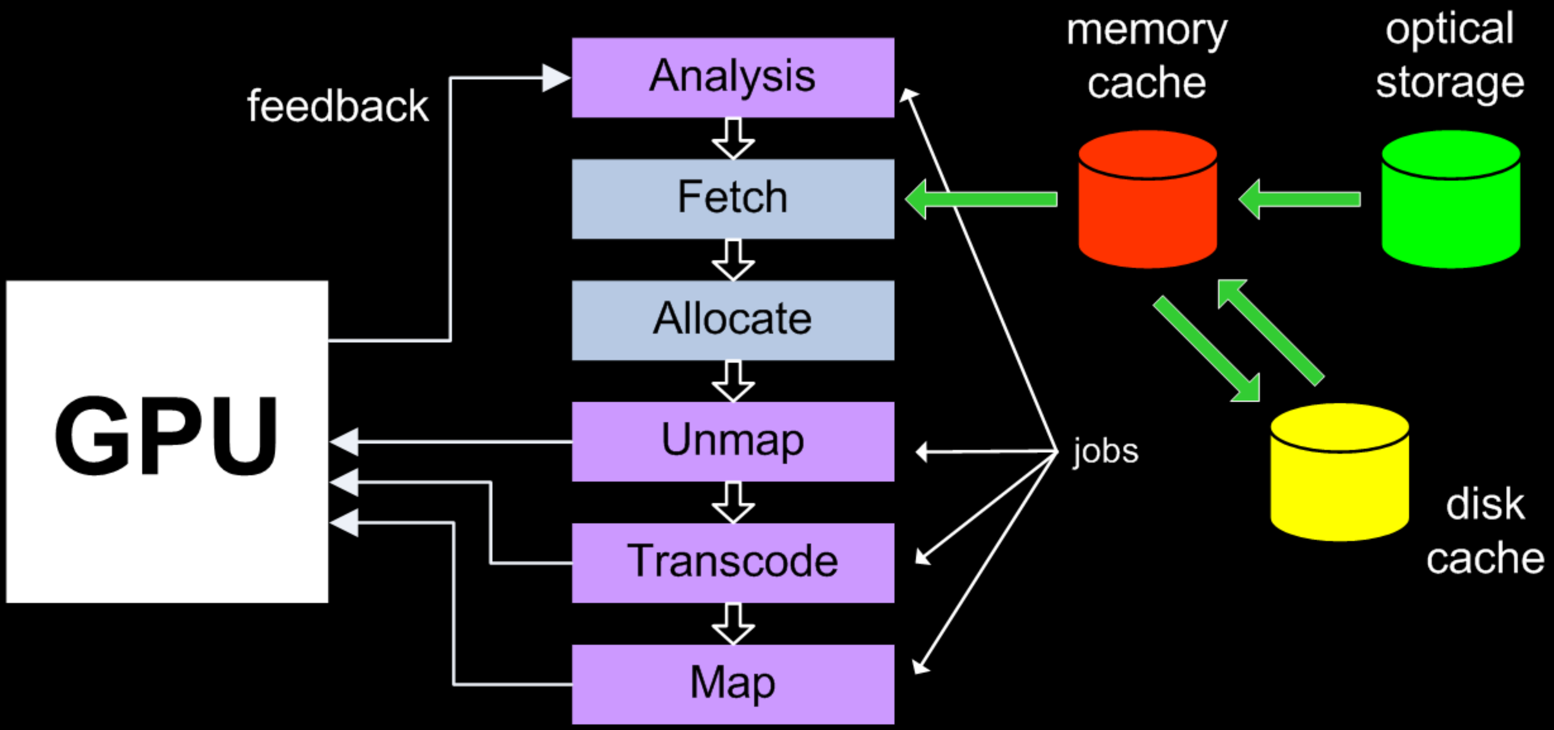
What does this look like per page?

- 128 x 128 texels per page
  - 120 x 120 payload + 4 texel border on all sides
  - 192 kB uncompressed
  - 40 kB DXT compressed
  - 1 – 6 kB DCT-based or HD-Photo compressed

Can’t render from variable bit rate

- Transcode DCT-based or HD-Photo to DXT
  - Significant computational load
  - 1 to 2 milliseconds per page on a single CPU core

Pipeline overview



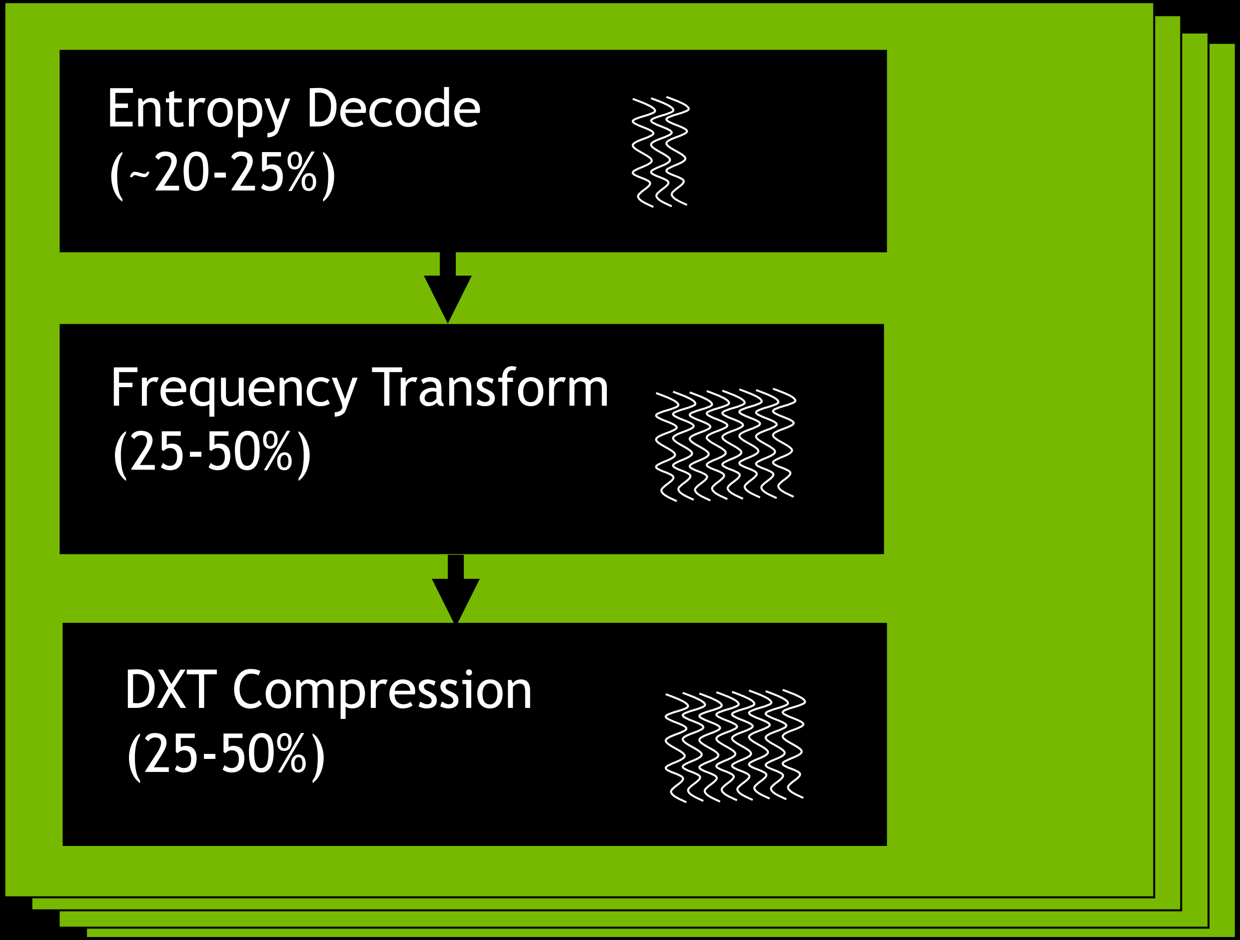
GPU Transcoding Motivation

- Transcode rate tied to quality / performance
  - Drop frames - Image is lower detail
  - Wait for results – frame rate degrades
- Densely occluded environment may desire in excess of 46 MTex/s
- DCT-based transcoding can exceed 20 ms per frame
- HD-Photo transcoding can exceed 50 ms per frame

Transcoding Analysis

- Several jobs (pages) per frame
- Jobs occur in several stages
  - Entropy decode
  - Dequantization
  - Frequency transform
  - Color space transformation
  - DXT compression

Transcoding Pipeline



Entropy Decode
(~20-25%)

Frequency Transform
(25-50%)

DXT Compression
(25-50%)

Transcoding Breakdown

- Entropy Decode
  - 20-25% CPU time
- Dequantization + Frequency transform
  - 25-50% CPU time
- Color transform + DXT compression
  - 25-50% CPU time

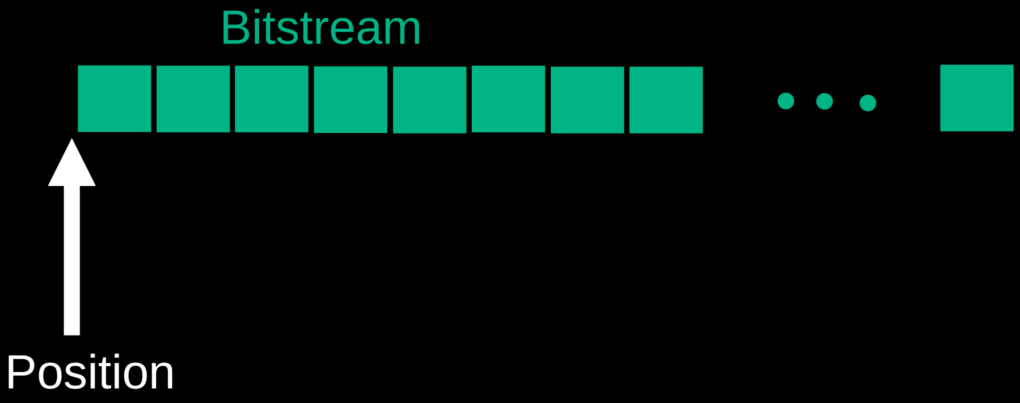
Transcoding Parallelism

- Entropy Decode
  - Semi-parallel, dozens to hundreds
- Dequantization + Frequency transform
  - Extremely parallel, hundreds to thousands
- Color transform + DXT compression
  - Extremely parallel, hundreds to thousands

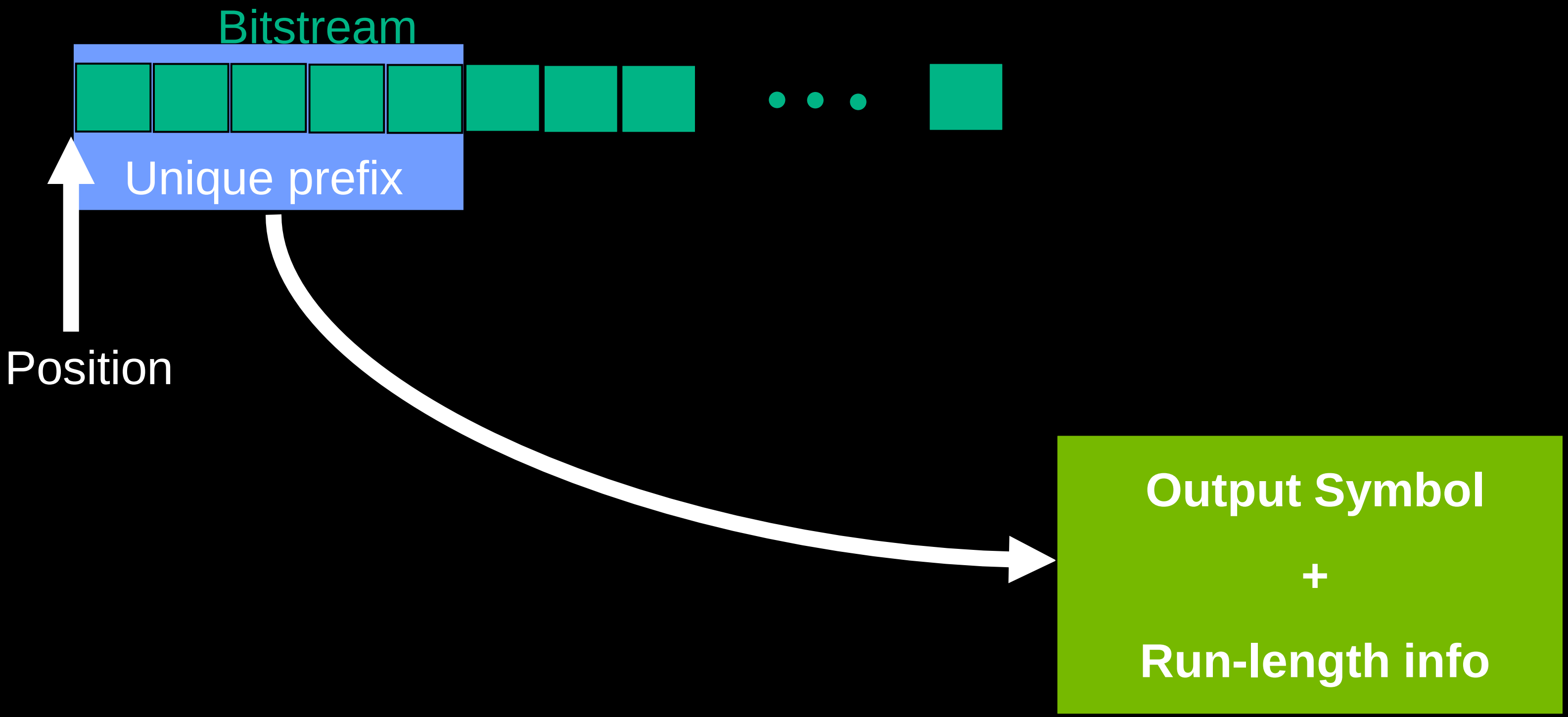
Entropy Decode

- Huffman based coding schemes
  - Variable bit-width symbol
  - Run-length encoding
- Serial dependencies in bit stream
- Substantial amount of branching

Huffman Decode Basics



Huffman Decode Basics


Bitstream

Unique prefix

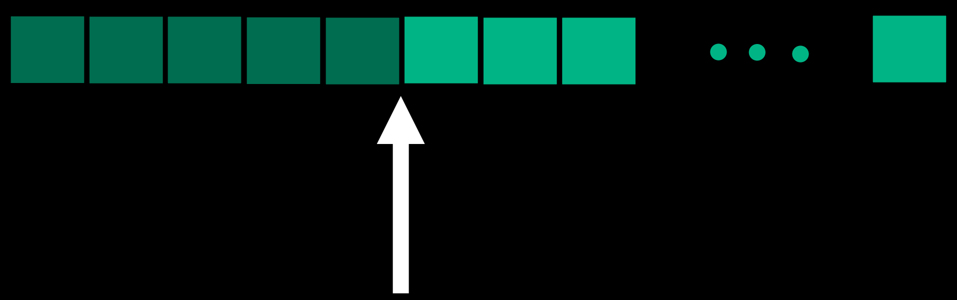
Position

Output Symbol

+

Run-length info

Huffman Decode Basics



Bitstream

• • •

Position

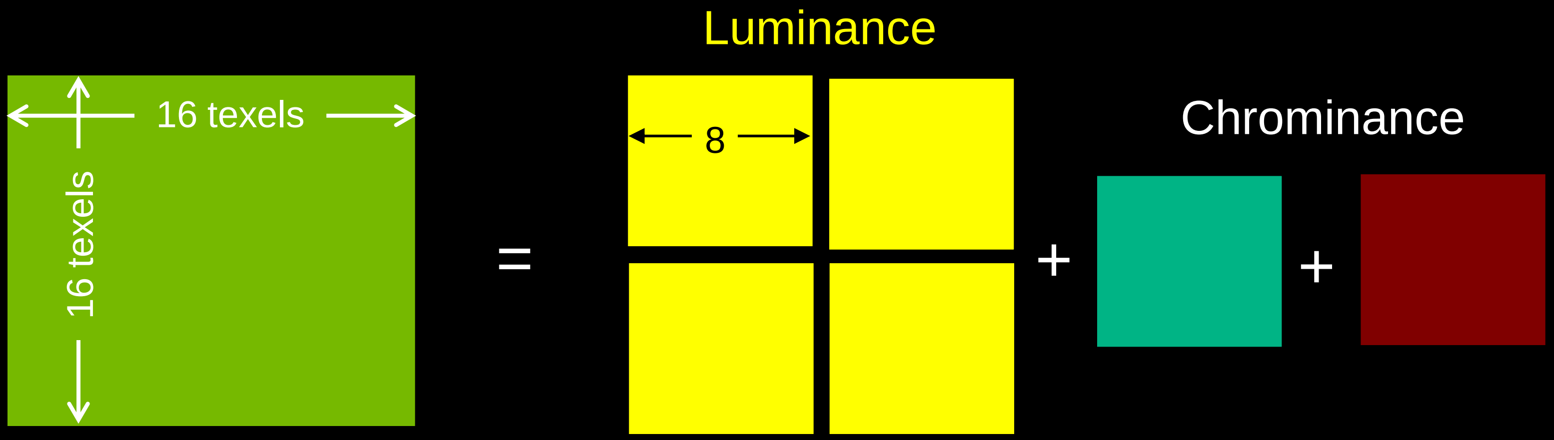
Huffman GPU Processing

- Long serial dependencies limit parallelism
- Relatively branchy (divergence)
- Relatively few threads
- Can perform reasonably with very many small streams
  - Not the case here
- CPU offers better efficiency today

Frequency Transform

- Block-based transform from frequency domain
- iDCT of macro blocks
  - Inherently parallel at the block level
  - Uses NVPP derived iDCT kernel to batch several blocks into a single CTA
  - Shared memory allows CTA to efficiently transition from vertical to horizontal phase

Macroblock



- Image broken into macro blocks
  - 16x16 for DCT with color encoded as 4:2:0
  - Blocks are 8x8

CUDA iDCT

- 2D iDCT is separable
  - 8x8 block has two stages 8-way parallel
  - Too little parallelism for a single CTA
- Luminance and Chrominance blocks may require different quantization
- Group 16 blocks into a single CTA
  - Store blocks in shared memory to enable fast redistribution between vertical and horizontal phase

iDCT Workload

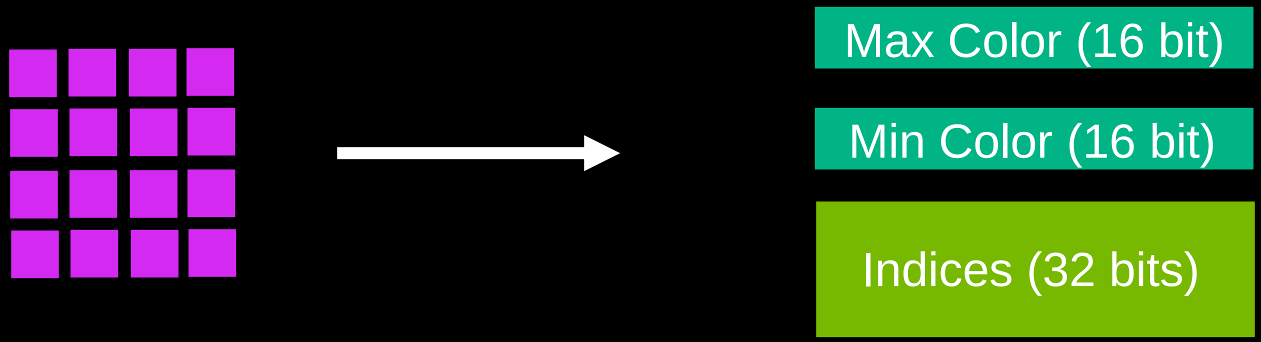
- 64 Macroblocks per 128x128 RGB page
- 6 Blocks per macroblock (4 lum. + 2 chroma)
- 8 Threads per block
- 3072 Threads per RGB page
  - Fills roughly 1/5th of the GTX 480

DXT Compression

- DXT relies on 4x4 blocks
  - 1024 blocks in one 128x128 image
- Thread per block works well
  - There is finer parallelism, but communication can be too much
  - Careful packing of data useful to prevent register bloat

DXT Blocks

- 4x4 texel block
- Min color, Max color, and 16 2-bit indices



Max Color (16 bit)

Min Color (16 bit)

Indices (32 bits)

CUDA DXT Compression

- All operations performed on integer colors
  - Matches CPU reference implementation
  - Allows packing of 4 colors into a 32-bit word
    - 4x better register utilization
- CTA is aligned to Macroblock boundaries
  - Allows fetch of 4:2:0 data to shared memory for efficient memory utilization
- Presently 32x32 texel region

Putting it Together

- CPU Entropy Decode needs to work on large blocks
  - Dozens of tasks per frame
- GPU kernels desire larger sets
  - All pages as a single kernel launch is best for utilization
  - Parameters, like quantization level and final format, can vary per page
- Must get data to the GPU efficiently

Solution CPU-side

- CPU task handles entropy decode directly to locked system memory
- CPU task generates tasklets for the GPU
  - Small job headers describing the offset and parameters for a single CTA task

Solution GPU-Side

- Tasks broken into two natural kernels
  - Frequency transform
  - DXT compression
- Kernels read one header per CTA to guide work
  - Offset to input / result
  - Quantization table to use
  - Compression format (diffuse or normal/specular)

One more thing

- CPU -> GPU bandwidth can be an issue
  - Solution 1
    - Stage copies to happen in parallel with computation
    - Forces an extra frame of latency
  - Solution 2
    - Utilize zero copy and have frequency transform read from CPU
    - Allows further bandwidth optimization

Split Entropy Decode

- Huffman coding for DCT typically truncates the coefficient matrix
- CPU decode can prepend a length and pack multiple matrices together
- GPU fetches a block of data, and uses matrix lengths to decode run-length packing
- Can easily save 50% of bandwidth

Run Length Decode

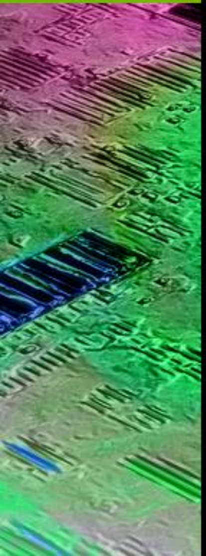
- Fetch data from system mem into shared mem
- Read first element as length
- If threadIdx < 64 and threadIdx < length copy
- Advance pointer
- Refill shared memory if below low water mark
- Repeat for all blocks

Results

- CPU performance increase
  - from 20+ ms (Core i7)
  - down to ~4 ms (Core i7)
- GPU costs
  - < 3ms (GTS 450)
- Better image quality and/or better frame rate
  - Particularly on moderate (2-4 core CPUs)

Conclusions

- Virtual Texturing offers a good method for handling large datasets
- Virtual texturing can benefit from GPU offload
- GPU can provide a 4x improvement resulting in better image quality



Thanks

id Software

NVIDIA Devtech