Real-Time Buffer Compression

Michael Doggett
Department of Computer Science
Lund university
Project

- 3D graphics project
- Implement 3D graphics algorithm(s)
- C++/OpenGL(Lab2)/iOS/android/3D engine
- Demo, Game
- **Proposal - Long paragraph by next Thursday**
- More in the next lecture
Stages we have looked at so far

- Vertex shader
- Rasterization
- Pixel shader
- Z & Alpha
- FrameBuffer
- Texture
Today’s stages of the Graphics Pipeline

- **Vertex shader**
- **Rasterization**
- **Pixel shader**
- **Texture**
- **Z & Alpha**
- **Framebuffer**

**Z and Color Compression**
Z & Alpha Performance

• Recall Memory Bandwidth determines performance
• Both units connect directly to memory
• Computation power of GPUs and CPUs increasing more rapidly than memory bandwidth
• Compression reduces the data before we transfer it
DRAM overview

• Dynamic Random Access Memory
  • Must have power and be refreshed to maintain it
• Discrete GPU memory - Fast and reasonably priced
• Many different types: SDRAM, VRAM, SGRAM, etc.
• Multiple improvements of data transfer
  • DDR - sends data on both low-to-high and high-to-low
  • QDR, then GDDR (Graphics DDR) versions 2, 3, 4, 5
• HBM - High Bandwidth Memory
  • 3D-stacked DRAM
PC memory architecture

Integrated CPU and GPU
Intel Sandy/Ivy Bridge/
Haswell/Skylake
AMD Fusion

12.8GB/s
System DRAM

8GB/s
PCIe2 16x

NVIDIA GTX980
224GB/s GDDR5

CPU
GPU

GPU
System DRAM
Graphics DRAM

© 2012 Michael Doggett
Mobile Memory Architecture

Based on 2014 iPhone5S

Back to Graphics Hardware algorithms
Why depth buffer?

Thus the only variation of interest here is Newell et al, an order of magnitude less "costly" and the brute-force approach which is already ridiculously expensive.


[Slide courtesy of John Owens]

The "brute-force approach" is depth buffering (aka Z-buffering): It won over sorting-polygon-methods because memory became ridiculously inexpensive...
Depth buffer bandwidth

• Still could be quite expensive!
• Zmin/Zmax-culling helped (previous lecture)

• Real-Time Buffer Compression can help reduce
  – Depth buffer bandwidth
  – Color buffer bandwidth
  – Other buffers...
Real-Time Buffer Compression

• Techniques that are or *may be* used in GPUs...

• Basic idea:
  – Lots of coherency (correlation) between pixels
  – Use that to compress buffer info
  – Send compressed buffer info over the bus
  – Special hardware handles compression and decompression on-the-fly
  – Must be lossless!!
General Compression System

Compression System

Control block

Tile table

Compression unit

Decompression unit

Cache

cached block 0

cached block 1

cached block 2

cached block 3

cached block 4
...
cached block n-1

Buffer in external memory

Tile data 0

Tile data 1

Tile data 2
...
Tile data \( t_w \)
...
Tile data \( t_{w+1} \)
...
Tile data \( t_w - 1 \)

© 2008 Tomas Akenine-Möller
Compression System

• Works on a tile basis
  – Eg 8x8 pixels at a time

• Cache is important!
  – Do not want to decompress tile for every fragment that needs access to values in that tile

• Tile table store ”per-tile info”:
  – E.g., which compression mode is used
  – Example: 00 is uncompressed, 01 is compressed at 25%, 10 is at 50%, 11 is cleared
  – Always needs one uncompressed mode as a fallback
Example

• Read request → ctrl block
• Checks cache
  – If there, deliver immediately
  – If not
    • Evict one tile from cache by attempting to compress info, and sending resulting representation, update tile table for that tile
    • Check tile table for requested tile, and
    • Read appropriate amount of bytes
    • Decompress (or send cleared info without reading, or in case of data being uncompressed, no decompression needed)
• Done
Dirty bit

• Each tile in cache has one bit for this
• When new info has been written to a tile in cache, set dirty bit=1
• When a tile in the cache needs to be evicted, check dirty bit
  – If =0, information in external memory is up to date → no need to write back!
  – If =1, attempt to compress, and send to external memory
• Saves a lot when no updates
  – Example: particle systems – do not write depth!
Depth buffer compression

• Hard to get accurate information about this
• Looking at patents we can extract some ideas
• Three techniques:
  – Depth offset compression
  – DPCM compression
  – Layered plane equation compression
Depth buffer compression

- Simplest buffer to compress
  - Highly coherent info (big triangles WRT tile size)
  - Depth is linear in screen space
- Depth cache and depth compression helps Zmax update for Zmax-culling
- Depths, $d(i,j)$ per tile,
  - $i$ is in $[0, w-1]$, $j$ is in $[0, h-1]$
  - Min depth value is $00...00_b$ (all zeroes, e.g. 24 bits)
  - Max depth value is $11...11_b$ (all ones)
    - i.e., we can use integer math
Depth offset compression (1)

• Identify a set of reference values, $r_k$,
  – and compress each depth as an offset with respect to one reference value
• Easiest to only use two reference values
  – Use $Z_{\text{min}}$ and $Z_{\text{max}}$ of tile!
  – Rationale: we have two layers
    • One with depths close to $Z_{\text{min}}$ and
    • one with depths close to $Z_{\text{max}}$
Visually, this means...

- Can encode if all z-values are in the gray regions
Depth offset compression (2)

• Use an offset range of $t=2^p$
• Can use offset, $o(i,j)$, per pixel as:

\[
o(i, j) = \begin{cases} 
  d(i, j) - z_{\text{min}}, & \text{if } d(i, j) - z_{\text{min}} < t, \\
  z_{\text{max}} - d(i, j), & \text{if } z_{\text{max}} - d(i, j) < t.
\end{cases}
\]

• If at least one pixel, $(i,j)$, cannot fulfil the above, the tile cannot be compressed!
• Info to store (if compression possible):
  – Zmin and Zmax
  – Plus $wxh$ $p$-bit values
Depth offset compression (3)

• Example with following assumptions:
  – 8x8 pixels per tile
  – $t=2^8$ means 8 bits per offset, $o$
  – 24 bits depth $\rightarrow$ Zmin and Zmax has 24 bits each

• Storage (uncompressed: 8x8x3=192 bytes):
  – 1 bit per pixel to indicate whether offset to Zmin or Zmax $\rightarrow$
    8x8x1 bits= 8 bytes
  – Offsets: 8x8x8 bits= 64 bytes
  – Zmin & Zmax: 6 bytes (might be on-chip though)
  – Total: 8+64+6=78 bytes $\rightarrow$ 100*78/192=41% compression
Less expensive implementation

- Only possible to compress if all depths in a tile are in gray regions
- There are some extensions to this that makes the hardware simpler!
- Make the offset computation inexpensive!
  - Currently costs an adder in HW
Inexpensive offsets...

• Instead of storing exactly Zmin and Zmax, store only \( m \) most significant bits (MSBs)
  –Call these truncated values, \( u_{\text{min}} \) and \( u_{\text{max}} \)

• Offset is now simply the \( k-m \) least significant bits of depth (no add needed)
Disadvantage of cheap offsets, and a solution

- Only values in dark gray area can be coded → loss of compressibility!
- Simple solution: use one more bit per pixel → four reference values:
  - $u_{\text{min}}$, $u_{\text{min}} + 1$, $u_{\text{max}} - 1$ and $u_{\text{max}}$
Decompression hardware

- Very simple
Compression ratio with inexpensive variant

- Slightly worse → 44% instead of 41%
- But, range of offsets is larger!
  - Best case: range is twice as large
  - Worst case: range is only one depth value larger
  - Average case: range is about 50% larger!
- So more tiles can be compressed, but still costs more
DPCM Compression

• DPCM = differential pulse code modulation
• Basic idea: we usually have linearly varying values in tile
  – Second derivative of linear function is zero!
  – However, we have discretized function, so need discretized "second derivatives"
DPCM: Focus on one column of depths

- For linear functions, the $\Delta s$’s will be close to 0
- Reconstruction is simple (next slide)
DPCM reconstruction

• From definition, we get: 
  \[ d_i = d_{i-1} + s_i, \quad i \geq 1 \]

• Only \( s_1 \) is known, but:
  \[ s_i = s_{i-1} + \Delta s_i, \quad i \geq 2 \]

\[
\begin{align*}
d_0, & \quad \text{already known} \\
d_1 &= d_0 + s_1, & \quad s_1 \text{ already known} \\
d_2 &= d_1 + s_2, & \quad \text{where } s_2 = s_1 + \Delta s_2 \\
d_3 &= d_2 + s_3, & \quad \text{where } s_3 = s_2 + \Delta s_3 \\
\ldots & \quad \text{and so on}
\end{align*}
\]
Process each column independently

- Not ideal: still many d’s and s’s
- Process first two rows similarly

© 2008 Tomas Akenine-Möller
DPCM: How to store this?

- One depth, d, two slopes, s, and 61 $\Delta s$
- The $\Delta s$ are small, in $[-1,+1]$ inside triangle
- Use two extra bits per pixel:
  - 00: add 0
  - 01: add +1
  - 10: add -1
  - 11: use as escape code to handle extraordinary cases...
- Best case compression (no escapes at all):
  - 24 bits + 25 +25 +8x8x2 ≈ 25 bytes (13% ratio)
  - If a single triangle covers entire tile
    - Do not need the 11-escape case then though...
DPCM: common case

• Single column:
  – Depths: 1, 2, 3, 4, 8, 10, 12, 14
  – Slopes: 1, 1, 1, 4, 2, 2, 2
  – Diff slopes: 0, 0, 3, -2, 0, 0

• Two escape codes needed per column to change from one plane eq (tri) to another
  – Becomes expensive! 40% compression ratio

• Solution: encode from the top & down and from bottom & up
  – Store also where transition happens
  – Gives about 20% compression ratio!
  – Might be possible to use fewer bits per slope
  – Can only handle two plane equations per tile
  – Still does not use escape
Plane Equation Compression

• Each triangle can be represented as a plane
• For every triangle in a tile store the triangle’s plane equation
  • Store one depth in center of tile, and an x-slope (dz/dx), and y-slope (dy/dz) across the tile
• For every pixel in the tile store an index to find the matching plane equation
• Works great for multisample!
• Random access
  • only decompress necessary pixels
• More info
  • [VanHook07] US Patent 7,242,400
Plane Equation Compression

- Plane 0: Z, x slope, y slope
- Plane 1: Z, x slope, y slope
- Plane 2: Z, x slope, y slope

- Plane Equations
  - $3 \times (3 + 2 + 2)$Bytes = 21Bytes

- Indexes
  - $64 \times 2$bits = 16Bytes

- Compressed
  - 37Bytes

- Uncompressed
  - $64 \times 3$Bytes = 192Bytes

- Compression ratio
  - 19%

© 2010 Michael Doggett
Color Buffer Compression

• Could use offset compression for R, G, and B separately (perhaps)
• Could use JPG’s non-lossy algorithms
• Can do simple color compression for multi-sample anti-aliasing
• Can compress clear color
• Is generally very difficult due to restrictions
  – Cannot be lossy
  – Must decode very fast for alpha blending
Conclusion

• Compression reduces bandwidth further
  – Several options for depth
    • For more details read Notes chapter 7
  – Harder for color
  – Needs cache
  – Needs fallback for non-compressed mode
Next ...

- Today and Friday
  - Assignment 2 marking in Uranus
- Next lecture
  - Antialiasing
  - Texture Compression
  - Start project
- Next week:
  - GPU Architecture
  - Graphics Architecture and OpenCL