Performance Analysis and Culling Algorithms

Michael Doggett
Department of Computer Science
Lund university
Stages we have looked at so far:

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

- Vertex shader
- Rasterization
- Pixel shader
- Texture
- Z & Alpha
- FrameBuffer
What determines performance?
“It’s the Memory, Stupid!”

Richard Sites, Microprocessor Report 1996

- GPU compute performance has increased
- GPU graphics performance has increased ~1000x in last 10 years
  - From
    - Radeon7500 (2001) 1.84 GigaOPS (16?bit fixed point)
    - Radeon5870 (2009) 2.72 TeraFLOPS (32bit floating point)
      - Radeon R9 290X (2013) 5.6 TeraFLOPS (32bit FP)
      - Nvidia GeForce 980 (2014) 4.6 TeraFLOPS (32bit FP)
  - Memory operations use power
    - Power is now limited
    - Especially true for Mobile devices
    - Thermal management is also a problem
GPU Performance

- **GPU compute**
  - G/TFLOPS - Giga/Tera floating-point operations per second
  - FLOPS = cores x clock x FLOPs/cycle

- **Memory Bandwidth**

- **Graphics Hardware**
  - Number of units
  - Algorithms - Compression
GPU example
GeForce GTX 980 Ti

- GPU Compute
  - 2816 cores x 1075 MHz clock x 2FLOPS/cycle
  - 5632 single precision GFLOPS
- Memory BW : 336GB/s
- Graphics Hardware :
  - 176 Texture, 96 Render output units
Performance Optimization

• Reduce load on a particular unit
  • if performance increases, that is the bottleneck
  • disable textures, alpha blending
  • replace shaders with single computation
also Benchmarking

- Games
- Tech Report uses
  - Beyond3D, Project Cars, The Witcher 3, GTA V, Far Cry 4, Alien: Isolation, Civilization: Beyond Earth, Battlefield 4, Crysis 3
- Synthetic benchmarks
- Triangles/second
Theoretical performance analysis of rasterizer (1)

- Some simple, useful formulae
- Useful tools when you should buy someone’s hardware...
  - Or investigate whether it is worth trying out particular algorithm
- New term: depth complexity
  - Measured per pixel
  - The number of triangles that overlap with a pixel (even though each triangle need not write to the pixel)
  - However, often say that a scene has an average depth complexity of, e.g., $d=4$
What is depth complexity?

Depth Complexity (Quake)

Color

Depth Complexity

[Slide courtesy of John Owens]
Theoretical performance analysis of rasterizer (2)

• New term: overdraw
  – Measured per pixel as well
  – How many times we write to a pixel
  – Less than or equal to depth complexity, \( o \leq d \)
• Statistical model of overdraw, \( o \):

\[
o(d) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{d}.
\]

• 1: first triangle is always written
• \( \frac{1}{2} \): second triangle has 50% of being in front of previous triangle
• \( \frac{1}{3} \): third triangle has a 33% chance of being in front of previous two triangles, and so on.

Example: 
\( d=4 \) gives \( o=2 \) (approx)
Theoretical performance analysis of rasterizer (3)

- \( T_r \) is texture read
  - 32 bits per texel, trilinear mipmapping needs 8 texels \( \rightarrow \) 32 bytes per access
- \( Z_r \) and \( Z_w \) are depth (Z) read and writes
  - 16, 24, or 32 bits
- \( C_r \) and \( C_w \) are color read and writes
  - 16, 24, or 32 bits
- Good formula for bandwidth, \( b \), per pixel:
  \[
  b = d \times (Z_r + Z_w + C_w + T_r)
  \]
  Not good!... Upper bound, though.
Theoretical performance analysis of rasterizer (4)

• Need to take overdraw into account...
  – Fragments that do not pass the depth test, do not need to: access texture, write depth, write color

\[ b = d \times (Z_r + Z_w + C_w + T_r) \]

\[ b = d \times Z_r + o \times (Z_w + C_w + T_r) \]

• Recall, \( d=4 \rightarrow o=2 \) (approx)
  – Significant difference (assume 3 bytes per color and depth):
    • \( b=4 \times 3 + 2 \times (3 + 3 + 32) = 88 \) bytes per pixel
    • \( b=4 \times (3 + 3 + 3 + 32) = 164 \) bytes per pixel (old formula)

Note: some architectures, do the texture lookup before depth test!
Theoretical performance analysis of rasterizer (5)

• Need to take texture cache into account too
  – With miss rate of, $m$, e.g., $m=0.2$ for 20% miss rate

  $$b = d \times Z_r + o \times (Z_w + C_w + m \times T_r)$$
  $$= d \times Z_r + o \times Z_w + o \times C_w + o \times m \times T_r$$
  = depth buffer, $B_d$  color buffer, $B_c$  texture read, $B_t$
  $$= B_d + B_c + B_t$$

• Significant difference again:
  – Miss rate $m=0.2$:
    • $b=4*3 + 2*(3 + 3 + 0.2*32) = 37$ bytes per pixel
    • $b=4*3 + 2*(3 + 3 + 32) = 88$ bytes per pixel
    • $b=4*(3 + 3 + 3 + 32) = 164$ bytes per pixel

Note: can have many more texture accesses per fragment though...
What else needs to be improved?

- $b = 4 \times 3 + 2 \times (3 + 3 + 0.2 \times 32) = 37$ bytes per pixel
- Texture bandwidth ($2 \times 0.2 \times 32 = 12.8$ bytes): ok
  - Can be reduced further with compression:
    - At 4 bits per texel: $2 \times 0.2 \times 8 \times 4/8 = 1.6$ bytes...
    - Does not work always though: e.g. render-to-texture
- Color buffer ($2 \times 3 = 6$ bytes): ok, not bad
- Depth buffer ($4 \times 3 + 2 \times 3 = 18$ bytes)
  - The worst bandwidth consumer at this point
    - Reads are worse than writes...
  - This lecture: reduce depth bandwidth using culling algorithms
  - Next lecture: compression of buffers
Culling and compression algorithms

• So far, we have seen texture caching and texture compression as good ways of reducing usage of texture bandwidth

• What else can be done?
  – Culling:
    • Zmax-culling and Zmin-culling
    • Object culling
  – Compression:
    • Depth buffer compression
    • Color buffer compression?
Zmax vs Zmin

- Left: small triangle is behind big triangle
- Right: small triangle is in front of big triangle
- Use tiles to cull parts of triangle
Zmax-culling (1)

• What about a fragment that fails the depth test (if test is `less_or_equal`)?
  – i.e., the fragment is occluded (not visible)
• Ideally, we do not want to process them at all!

$$B_d = d \times Z_r + o \times Z_w,$$

reads \hspace{1cm} writes

• We know that $o \leq d$, so reads consume more than writes
• Zmax-culling:
  – Very simple technique
  – Culls occluded fragments on a tile basis (tiled traversal is a must!)
  – Works without user intervention, i.e., fully automatic

AMD and NVIDIA has some form of Zmax-culling in their hardware
Zmax-culling example

Now render red triangle
Zmax-culling (2)

• A tile is $w \times h$ pixels, and its depths:
  
  \[ d(i, j), \quad i \in [0, w - 1] \text{ and } j \in [0, h - 1] \]

• The key is to, per tile, maintain:

  \[ z_{\text{max}} = \max_{ij} [d(i, j)] \]

• Can be seen as a low-res Z-buffer!

• When tiled traversal algorithm arrives at a new tile that overlaps triangle, do the following per-tile computations:
  
  – Compute "smallest z-value on triangle": \( z_{\text{min}}^{\text{tri}} \)

  – If the following is true, we know that triangle is occluded by zmax in tile, and can avoid depth reads:

    \[ z_{\text{min}}^{\text{tri}} > z_{\text{max}} \]
Zmax-culling (3)

• Great, how to compute $z_{\text{tri}}^{\text{min}}$?
• Many different ways, but key insight is that it need not be the exact value, as long as we err on the right side:
  – Compute it in a **conservative** manner!
  – In this case, it means that as long as we compute a value that is **smaller** than the exact value, the algorithm will not generate incorrect results. Only performance will be slightly worse
Zmax-culling (4)

• How to make a conservative estimate of $z_{\text{min}}^{\text{tri}}$?

• What is smaller than the ”smallest z-value of the triangle” in that particular tile?

• Different techniques include:
  1. Minimum of vertices’ depth values
  2. Evaluate depth at corners of tile, then pick minimum of these
     • Optimal, if triangle cover entire tile!
     • Bad, when, e.g., all vertices are inside a tile!
  3. Always optimal: clip triangle against tile, evaluate depth at all vertices of clipped triangle
     • Not feasible though!
  4. Hybrid: take maximum of result of 1 & 2
Zmax-culling (5)

• Where should we store each tile’s zmax-value?
  – Remember: we want to reduce memory bandwidth
  – On-chip memory, or if that takes too much memory, access the zmax’s through a cache

• How should we update the zmax?
  – Can only become smaller
  – Only way: read all depths in tile, compute maximum...
  – Could be expensive, but works well with depth buffer compression (next lecture)
• Now render red triangle

• Zmax culling saves **Read** pixel bandwidth
Zmin-culling example

- Red triangle is currently being rendered
Zmin-culling (1)

• Similar to Zmax-culling, but instead we store, per tile:

\[ z_{\text{min}} = \min_{ij} [d(i,j)] \]

• When tiled traversal algorithm arrives at tile overlapping the triangle, we compute the ”maximum of the triangle’s z in that tile”:

\[ z_{\text{tri}}^{\text{max}} \]

• This value can be computed using equivalent techniques for Zmax-culling...

• We can avoid depth reads, when the following is true: 

\[ z_{\text{tri}}^{\text{max}} < z_{\text{min}} \]
Zmin-culling (2)

• This means that the triangle currently being rendered is definitely in front of the contents in depth buffer
• Which means that the depth test will pass, and thus reading the depth can be avoided
• Zmin stored on-chip or in cache (as Zmax)
• Updating zmin?
  – **Simple**: as soon as a depth, \( d \), has been written that is smaller than \( z_{\text{min}} \), we update \( z_{\text{min}} = d \)
  – i.e., no need to read all depth values in tile (as in Zmax)
Zmin-culling example again

- Red triangle is currently being rendered
Zmin-culling results

• Applications for mobile devices were used for benchmarking

- $d=0.65$
- $d=2.5$
- $d=1.5$

• Reduction in depth reads:
  - 84%
  - 69%
  - 49%
How can Zmin work better than Zmax?

- Back to the equations, depth buffer bandwidth, $B_d$:

$$B_d = d \times Z_r + o \times Z_w = \underbrace{(d - o) \times Z_r}_{\text{fragments that only read}} + \underbrace{o \times Z_r + o \times Z_w}_{\text{fragments that read and write}}$$

- $d-o$ fragments for Zmax, $o$ for Zmin-culling

- There are more fragments for Zmax when:

$$d - o > o \iff d > 2o$$
Zmin vs Zmax

• For \( d=4 \) we get \( o=2 \) (approx), and hence we will get:
  – more fragments for Zmax when \( d>4 \), and
  – more fragments for Zmin when \( d<4 \)

• Start rendering of a scene:
  – Depth complexity is zero for all tiles
  – Render triangles, and depth complexity starts to build up. Zmin-culling works immediately here
  – When depth complexity is \( >4 \), Zmax-culling starts to work better than Zmin-culling
Zmin & Zmax

• Both algorithms can only get rid of depth reads!
  – [Or for architectures which always do texturing before per-pixel depth reads (Late Z), you get rid of texturing and pixel shader executions as well]

• Both should be implemented for best performance, however, for low depth complexity Zmin will pay off the most

• Zmin is also simpler to implement

• Normally, depth is 16, 24, or 32 bits per pixel
  – A conservative value for Zmin and Zmax works well:
    • 8 bits might be enough
    • Trade-off though...
Object Culling

• Can cull an entire object at a time
  – Can save bandwidth from CPU to GPU, vertex processing, and fragment processing!
• Needs user intervention, i.e., not automatic
• User can issue an "occlusion query":
  – Render a set of triangles, count the fragments that passes the depth test
  – i.e. GL_ARB_occlusion_query
• Common use: render bounding box of complex object (character, e.g.)
  – If no fragments passes, then entire BBOX is hidden
  – Means: entire object is hidden too
  – I.e, do not render object!
Next ...

- Tomorrow
  - Assignment 1 marking in Uranus

- Thursday
  - Shadows and deferred shading
  - Download and start on assignment 2
  - Start thinking about project

- Monday next week:
  - Real-time depth buffer compression