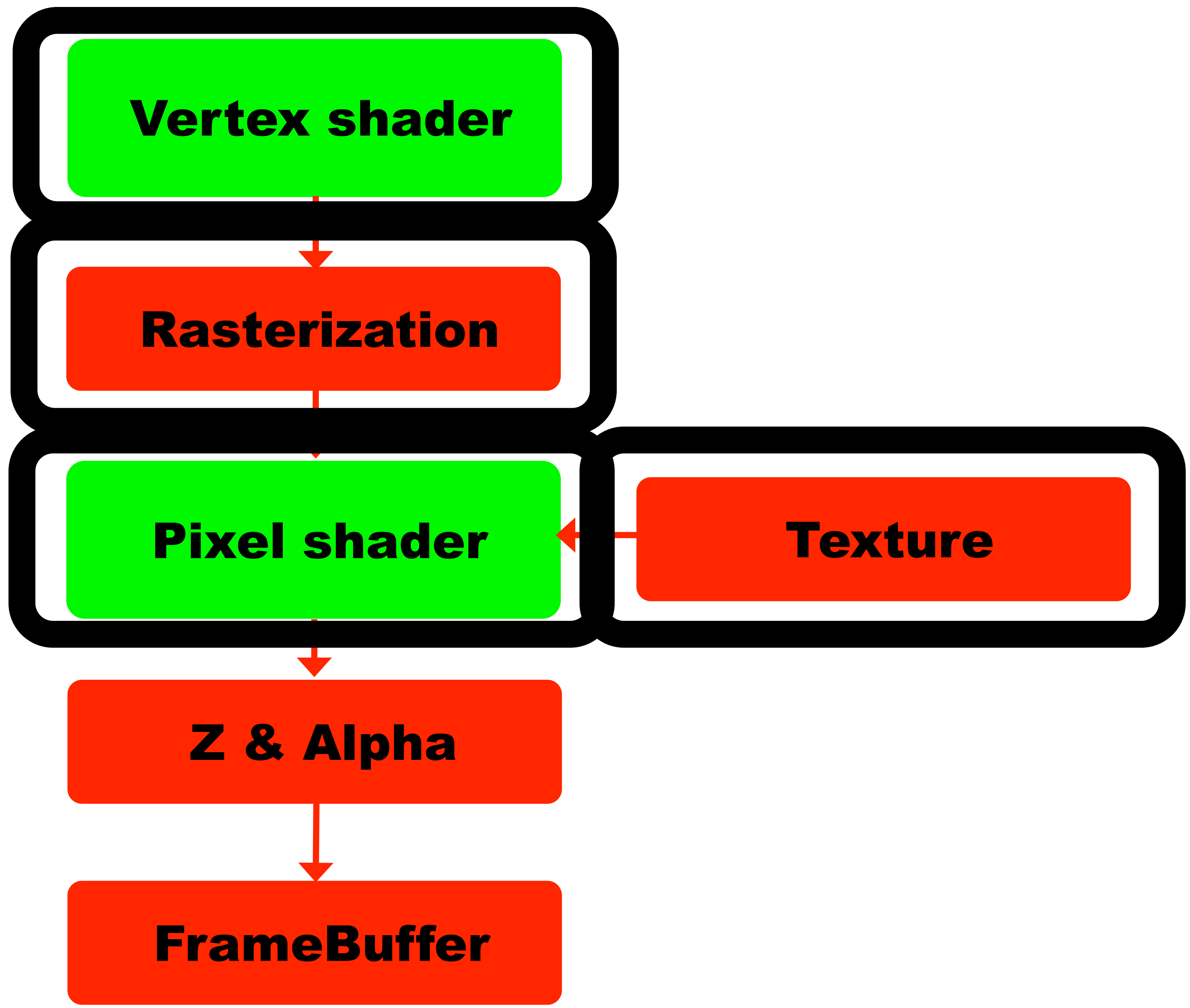
Performance Analysis and Culling Algorithms

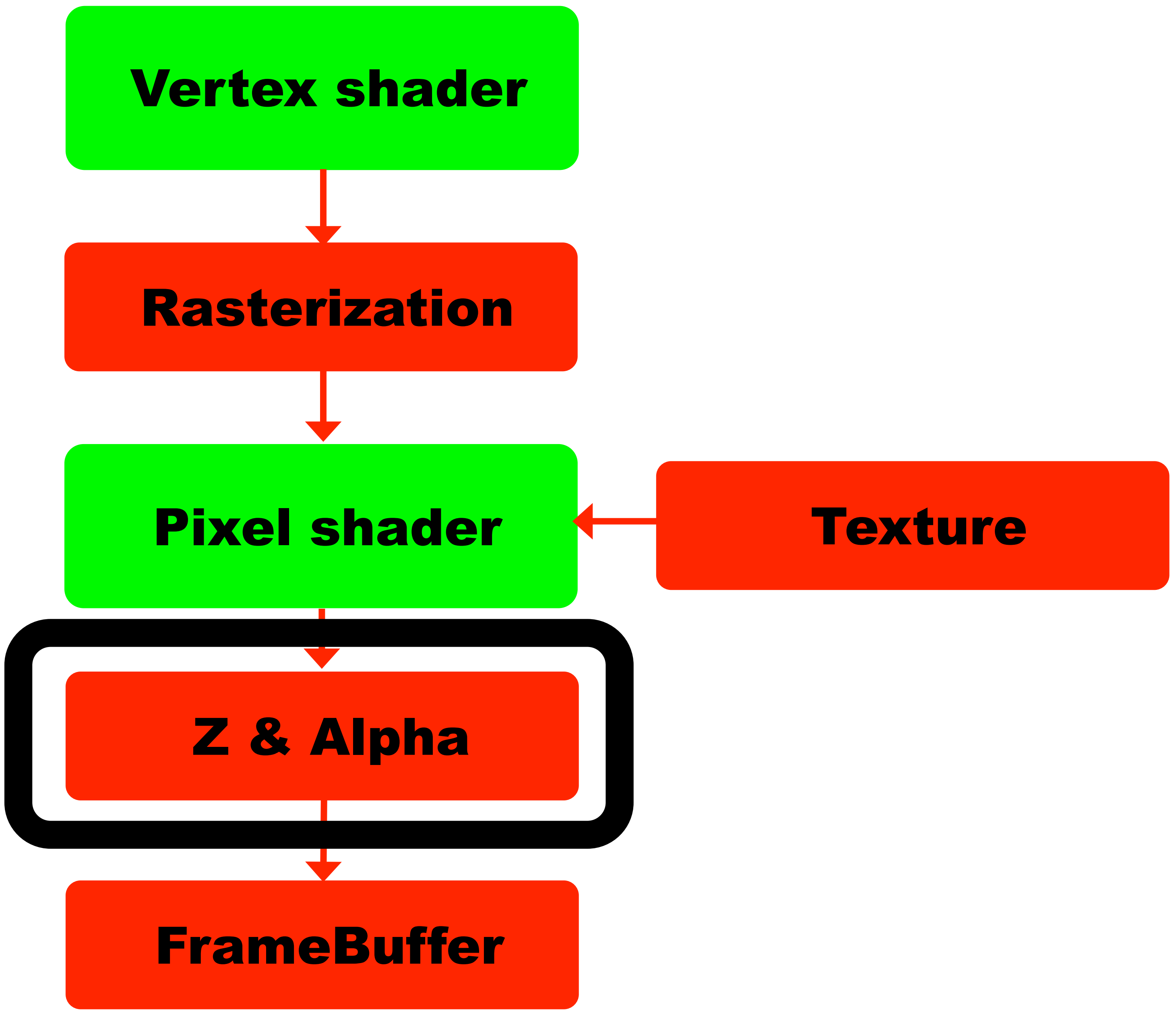
LUGG
Lund University Graphics Group

Michael Doggett
Department of Computer Science
Lund university

Stages we have looked at so far

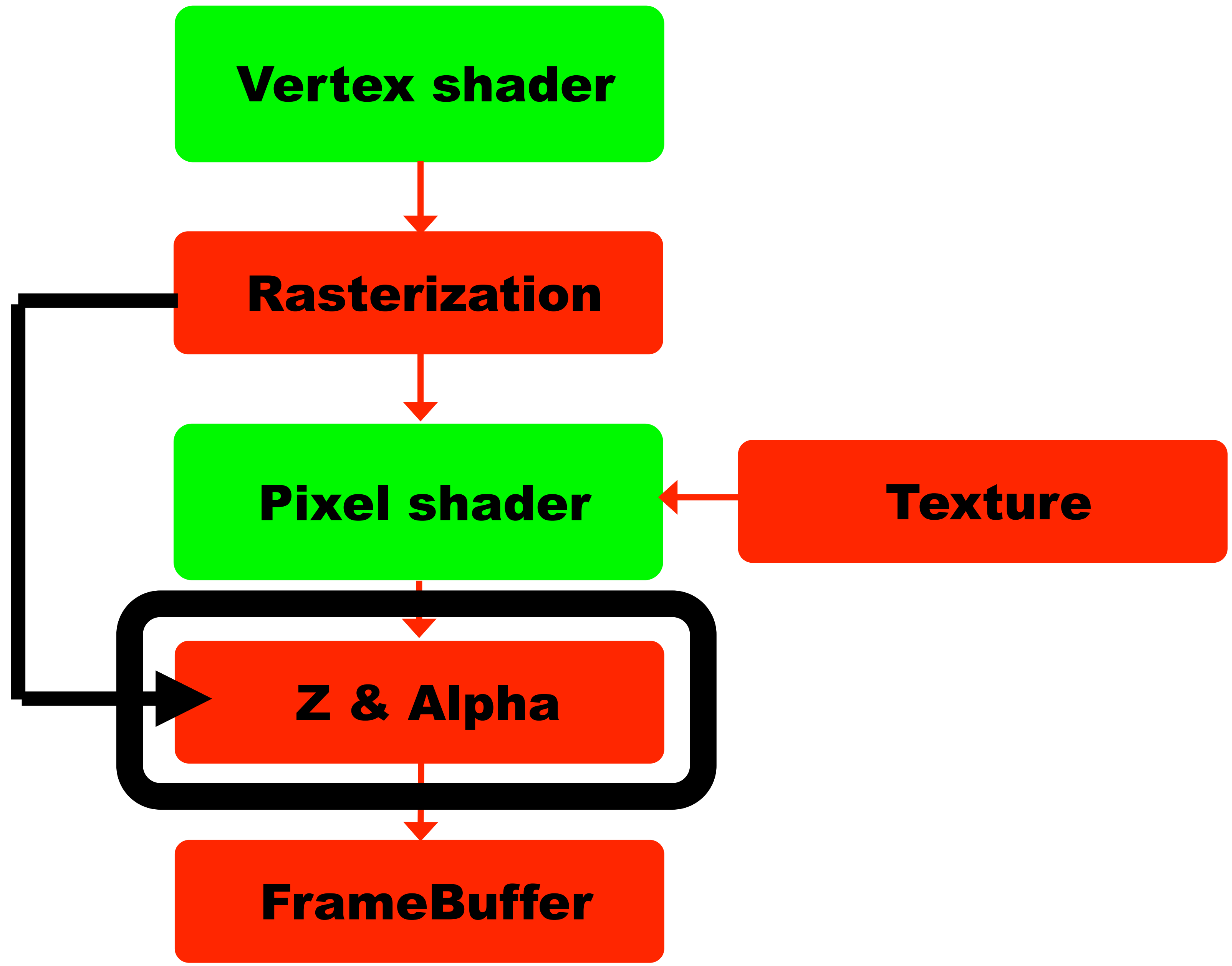


Today's stages of the Graphics Pipeline



Today's stages of the Graphics Pipeline

Rasterization uses Z hardware to determine visibility before Pixel Shader

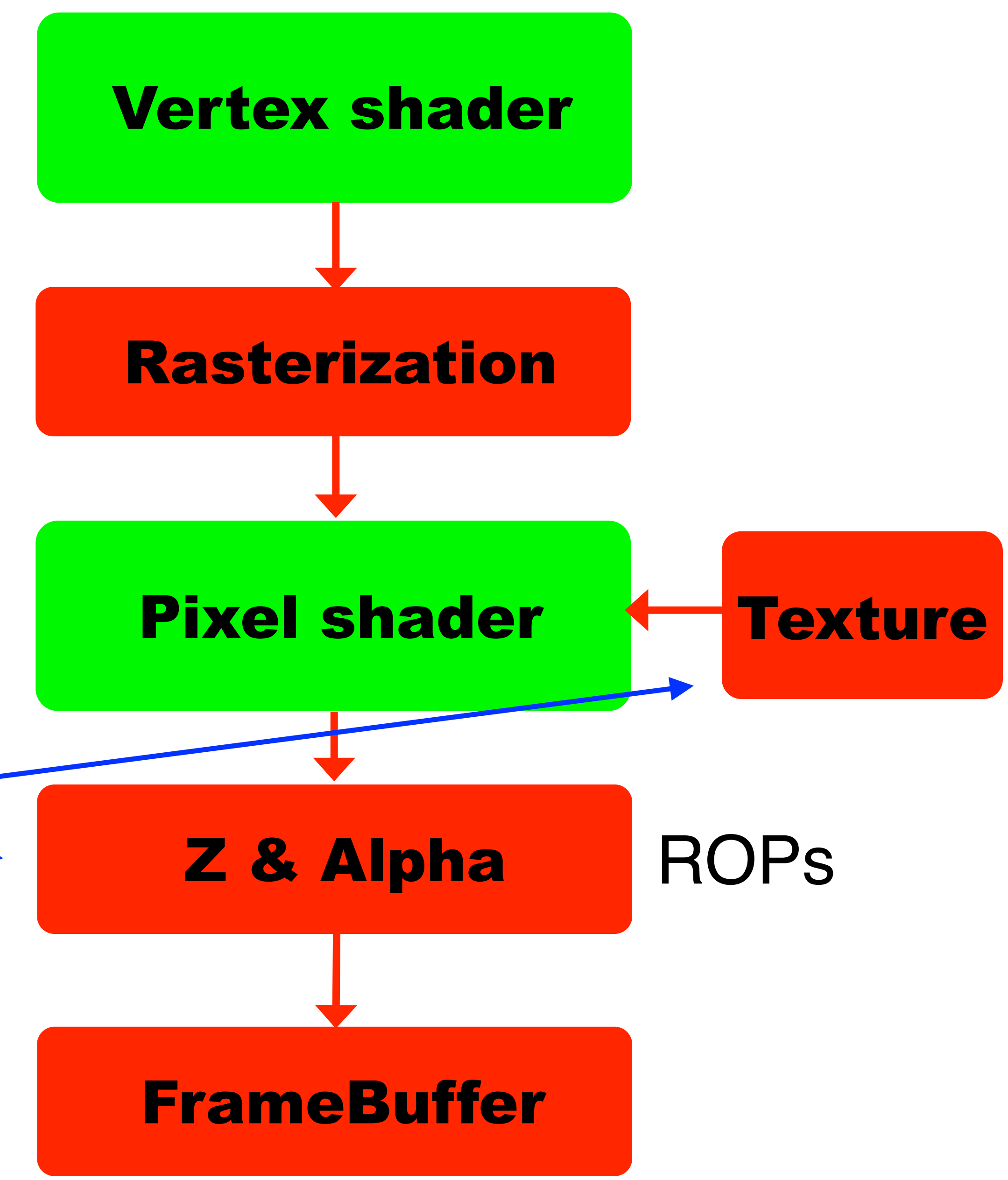


Overview

- Aspects of GPU Performance
- Rasterization Equation
- Hierarchical Z Culling

GPU Performance

- GPU compute
  - TFLOPS - Tera (10^{12}) floating-point operations per second
  - total FLOPS = cores x clock x FLOPs/cycle
    - FLOP is a 32bit FP add or multiply
- Memory Bandwidth
- Graphics Hardware
  - Number of units
  - Algorithms - Compression



GPU example
GeForce GTX 980 Ti
(2014)

- GPU Compute
    - 2816 cores x 1075 MHz clock x 2 FLOPS/cycle
    - 5632 single precision GFLOPS (5.6 TFLOPS)
- Memory BW : 336GB/s
- Graphics Hardware :
    - 176 Texture units and 96 Render output units (ROPs)

GPU Performance

- Hardware specifications
  - Clock speed, memory size and speed, number of processing units
- Code
  - Algorithm complexity
  - Parallel performance
    - Amdahl's law - parallelisation is only as effective as how much it parallelises
  - Data locality
    - Data needs to be close to computation unit
    - Data movement is expensive in time and energy

"It's the Memory, Stupid!"

Richard Sites, Microprocessor Report 1996

- Memory bandwidth creates an upper limit for Graphics
- GPU graphics performance has increased ~16x in last 10 years ('13-'22)
- GPU compute performance increased ~1000x from 2000 to 2010
  - 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)
    - Nvidia Tesla V100 (GV100) (2017) 15 TeraFLOPS (32bit FP)
    - Nvidia Ampere (RTX3090) (2020) 35 TeraFLOPS (32bit FP)
    - Nvidia Ada (RTX4090) (2022) 82 TeraFLOPS (32bit FP)
- Memory operations use power
  - Power is limited
  - Especially true for Mobile devices
  - Thermal management is also a problem

Performance Optimization

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

GPU Performance measured using Actual Games

- Games
  - Ars Technica 3080 review uses
    - MS Flight Simulator, AC Odyssey, Far Cry 5, RDR2, GTA V, Hitman 2, Control, Minecraft RTX, Wolf Youngblood, Shadow of the Tomb Raider
- Synthetic benchmarks
  - Triangles/second

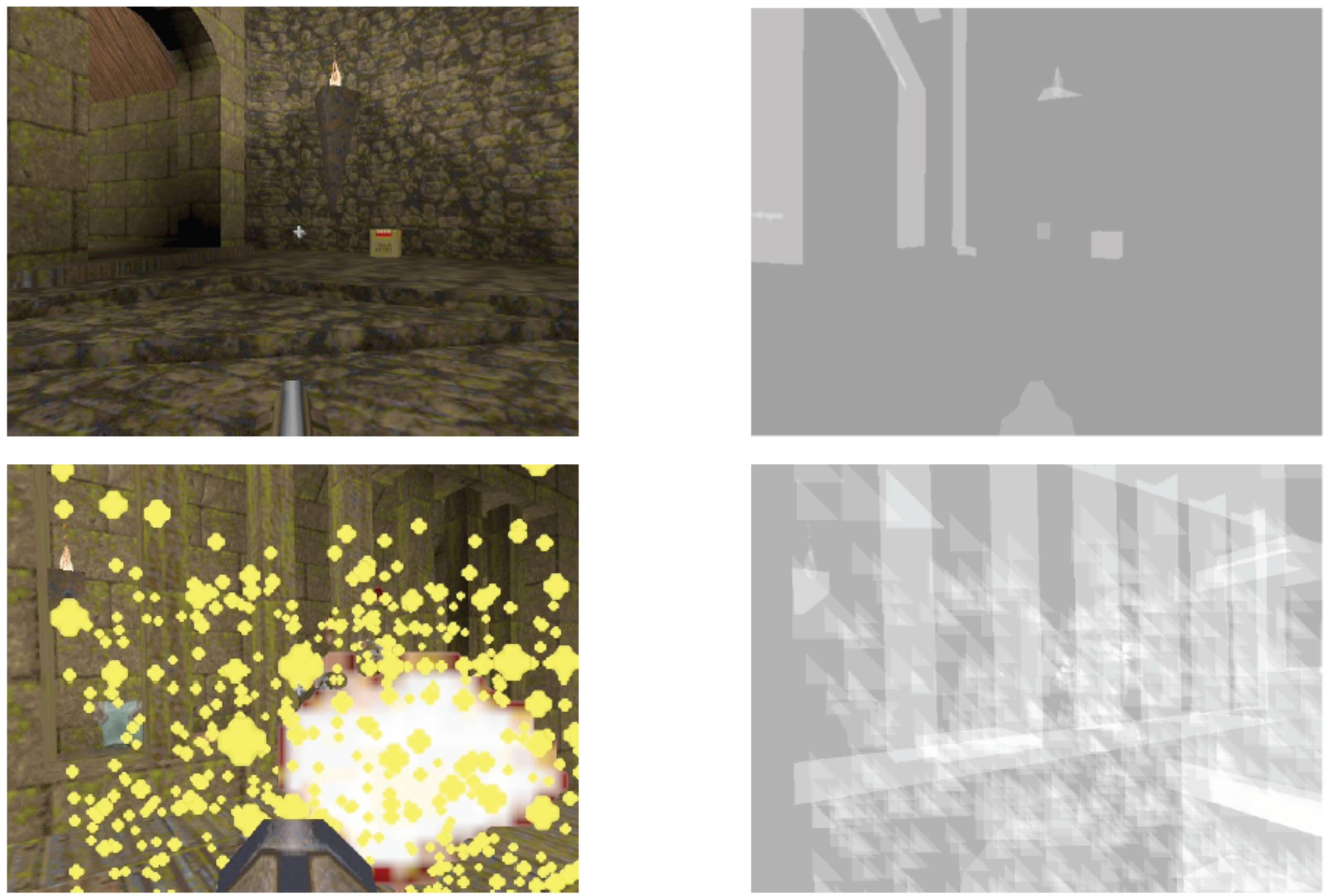
Minecraft RTX

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, $o<=d$
- **Statistical** model of overdraw, o:

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

Example:
$d=4$ gives
$o=2$ (approx)

- 1: first triangle is always written
- ½: second triangle has 50% of being in front of previous triangle
- 1/3: third triangle has a 33% chance of being in front of previous two triangles, and so on.

Theoretical performance analysis of rasterizer (3)

- T_r is texture read
  - –32 bits per texel, trilinear mipmapping needs 8 texels → 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

~~$b = d \times (Z_r + Z_w + C_w + T_r)$~~ $\quad$ $b = d \times Z_r + o \times (Z_w + C_w + T_r)$

- Recall, d=4 → o=2 (approx)
  - Significant difference (assume 3 bytes per color and depth):
    - b=4*3 + 2*(3 + 3 + 32) = 88 bytes per pixel
    - b=4*(3 + 3 + 3 + 32) = 164 bytes per pixel (old formula)

Note: Sometimes (Late Z), the texture lookup is before the 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

Rasterization equation

$$\begin{aligned} b &= d \times Z_r + o \times (Z_w + C_w + m \times T_r) \\ &= \underbrace{d \times Z_r + o \times Z_w}_{\text{depth buffer, } B_d} + \underbrace{o \times C_w}_{\text{color buffer, } B_c} + \underbrace{o \times m \times T_r}_{\text{texture read, } B_t} \\ &= B_d + B_c + B_t \end{aligned}$$

- 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...

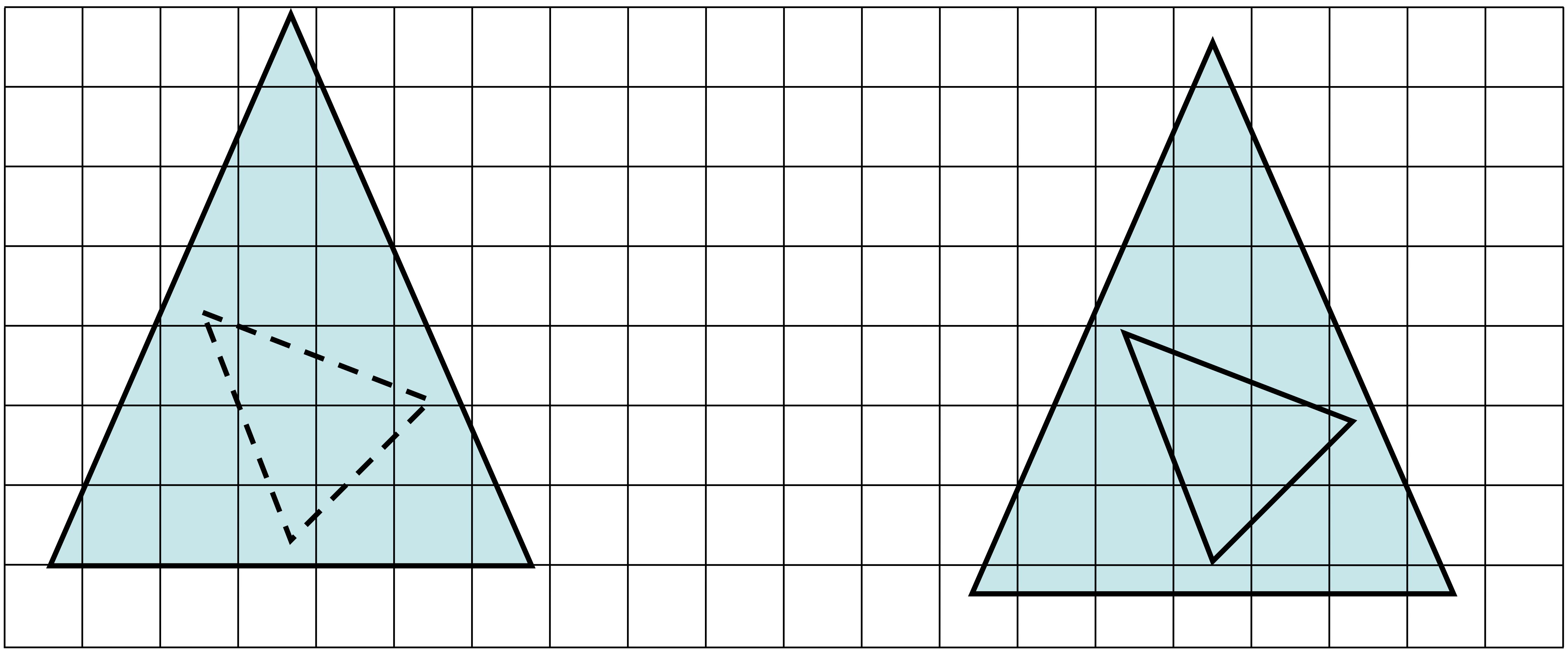
What else needs to be improved?

- b=4*3 + 2*(3 + 3 + 0.2*32) = 37 bytes per pixel
- Texture bandwidth (2*0.2*32=12.8 bytes): ok
  - Can be reduced further with compression:
    - At 4 bits per texel: 2*0.2*8*4/8=1.6 bytes...
    - Does not work always though: e.g. render-to-texture
- Color buffer (2*3=6 bytes): ok, not bad
- Depth buffer (4*3 + 2*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?

Hierarchical Z
Zmax vs Zmin



- Left: small triangle is behind big triangle
- Right: small triangle is in front of big triangle
- Use screen tiles to cull parts of triangle

Zmax-culling (1)

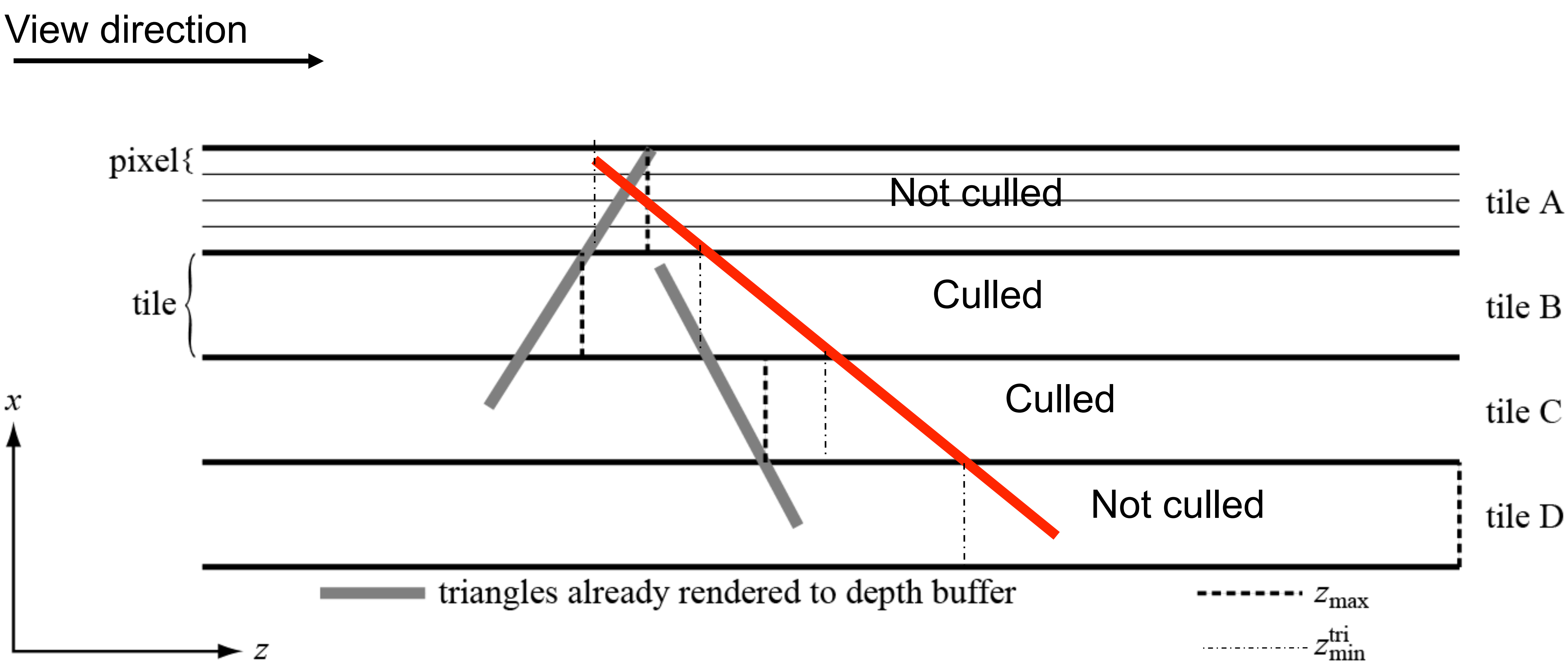
- What about a fragment that fails the depth test (if test is `less_or_equal`)?
  - i.e., the fragment is occluded (not visible)
- Ideally, we do not want to process them at all!

$$B_d = \underbrace{d \times Z_r}_{\text{reads}} + \underbrace{o \times Z_w}_{\text{writes}},$$

- We know that $d>=o$, 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

- **Cull when Z_tri_min > Z_tile_max**

Zmax culling

- Each tile is *w x h* pixels in size, with a Z (depth) at each pixel
- Store maximum of tile's Z values (Z_tile_max)
    - Together all Z_tile_max values look like a low resolution Z buffer
- When rasteriser performs tiled based traversal, at each tile
    - Compute smallest Z value from triangle in current tile (Z_tri_min)
    - Check if (Z_tri_min > Z_tile_max)
        - If true, cull tile, avoid Z reads

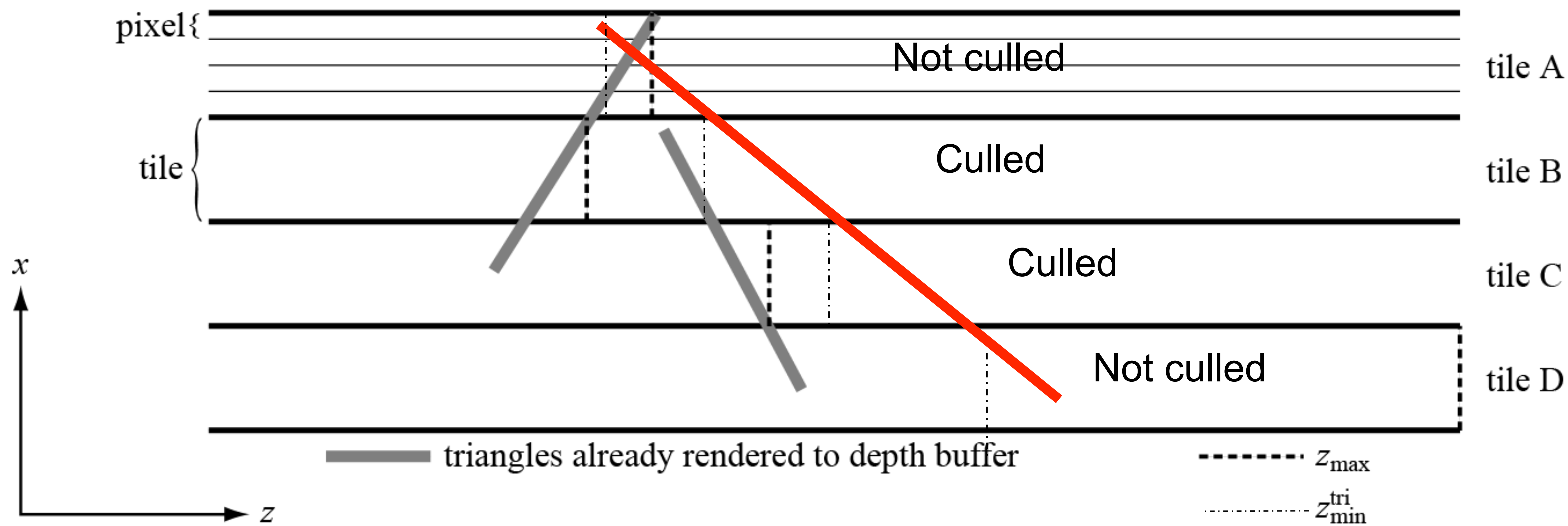
How to compute minimum Z value in Tile?

- Approximate values will work
  - Must use conservative testing
  - Computed Zmin must be less than actual Zmin
- Many ways to compute triangles Zmin value
  1. Find minimum triangle vertex
     - Ideal if triangle is inside tile
     - Bad if triangle is large, and much bigger than tile
  2. Find minimum tile corner values
     - Ideal if triangle covers the whole tile
     - Bad if triangle is small, and worse if triangle is parallel to view direction
  3. Find minimum of triangle clipped to tile
     - Expensive computation
  4. Take maximum of 1 & 2

Tile Zmax storage and update

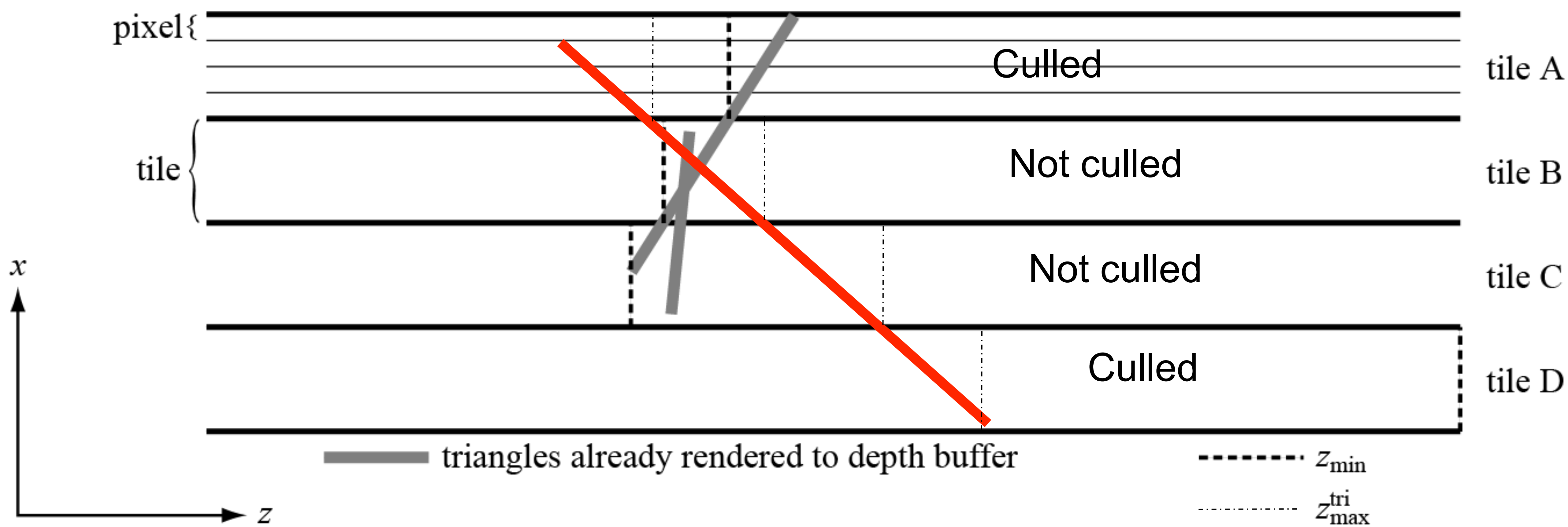
- Store Tile Zmax values in on-chip cache
  - Fast and avoids adding memory bandwidth
  - If too big for on-chip memory, a cache is a good option
- Zmax update
  - Only gets smaller
  - Must check all Z values in tile, find maximum
  - Z compression helps reduce cost of update

Zmax-culling example (same example again)



- Now render red triangle
- Zmax culling saves **Read** pixel bandwidth
  - **Cull when Z_tri_min > Z_tile_max**

Zmin-culling example



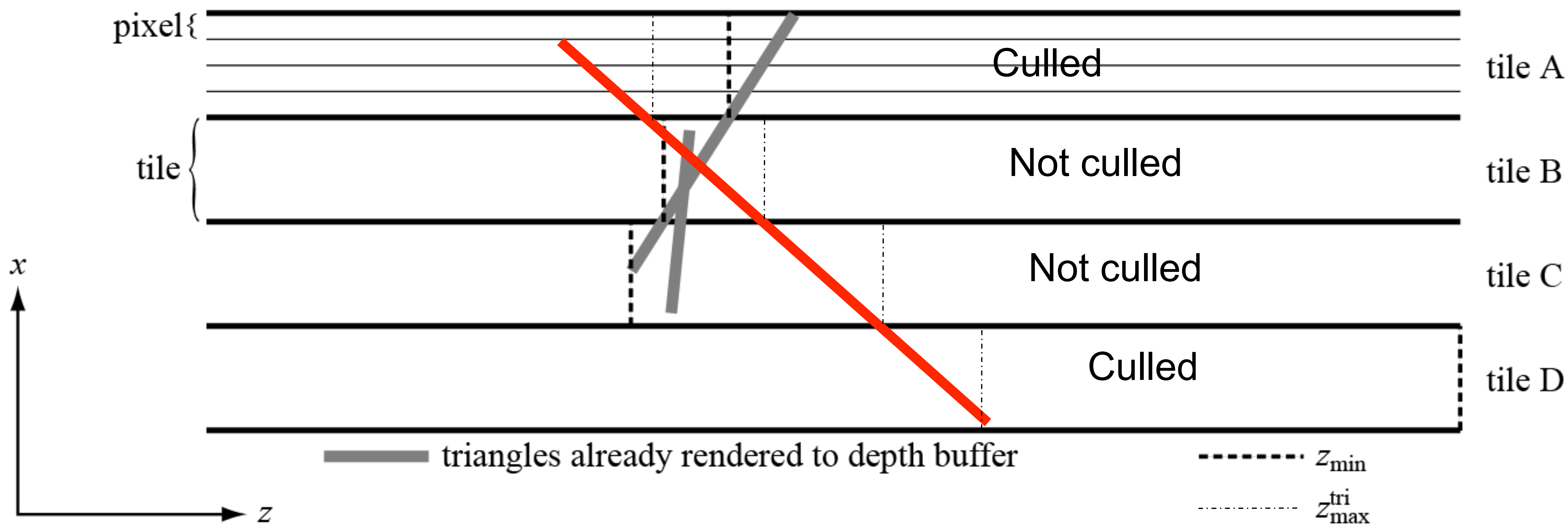
- Red triangle is currently being rendered

• Cull when Z_tri_max < Z_tile_min

Zmin culling

- When rasteriser performs tiled based traversal, at each tile
  - Compute largest Z value from triangle in current tile (Z_tri_max)
    - Use same approach as for Z_tri_min
  - Check if (Z_tri_max < Z_tile_min)
    - If true, all pixels pass, avoid Z reads
      - All pixels are in front of everything in the current tile
  - Store Z_tile_min in on-chip cache (same as Z_tile_max)
  - Z_tile_min update
    - If any Z is < Z_tile_min, update
    - Much easier than Z_tile_max

Zmin-culling example again



- Red triangle is currently being rendered

- **Cull when Z_tri_max < Z_tile_min**

Can Zmin work better than Zmax?

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

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

These fragments fail the depth test
i.e., ”occluded fragments”

Zmax-culling can potentially
avoid these reads

These fragments pass the depth test
i.e., ”visible fragments”

Zmin-culling can potentially
avoid these **reads**

- d-o fragments for Zmax, o for Zmin-culling
- There are more fragments for Zmax when:

$$d - o > o \quad \Leftrightarrow \quad d > 2o$$

$$d - o > o \quad \Leftrightarrow \quad d > 2o$$

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. `glBeginQuery(GL_ANY_SAMPLES_PASSED, 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 ...

- Next week:
  - Buffer compression and Antialiasing
  - Lab 2 Deferred Shading
- Think about project!