Performance Analysis and Culling Algorithms

Michael Doggett

Department of Computer Science

Lund University

Assignment 2

- Sign up for Pluto labs on the web page

Mobile graphics problems...

- Mobile devices use batteries (doh!)
- Memory accesses cost **a lot** of energy
- Strategy for making a good architecture:
  - Attempt to reduce memory accesses!
  - (preferably) without loss in image quality!
- Memory accesses cost in terms of power
  - → use too many accesses, and battery dies too soon!
- But other implications as well...

Remains a challenge? Power!

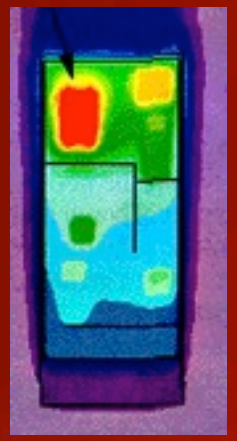
- Battery improvement doesn't follow Moore's law
  - Only 5-10% per year
- Gene's law
  - "power consumption of integrated circuits decreases exponentially" over time and because of that the whole system built around chips will get smaller, and batteries will last longer
  - Since 1994, the power required to run an IC has declined 10x every two years
  - But the performance of two years ago is not enough
    - Pump up the speed
    - Use up the power savings

N-GAGE NOKIA

Slide above is courtesy of Kari Pulli at Nokia

Remains a challenge? Thermal mgt!

- But ridiculously good batteries still won't be the miracle cure
  - The devices are small
  - Generated power must get out
  - No room for fans
- Thermal management must be considered early in the design
  - Hot spot would fry electronics
    - Or at least inconvenience the user…
  - The heat **must** be conducted through the case walls, and finally removed to the ambient



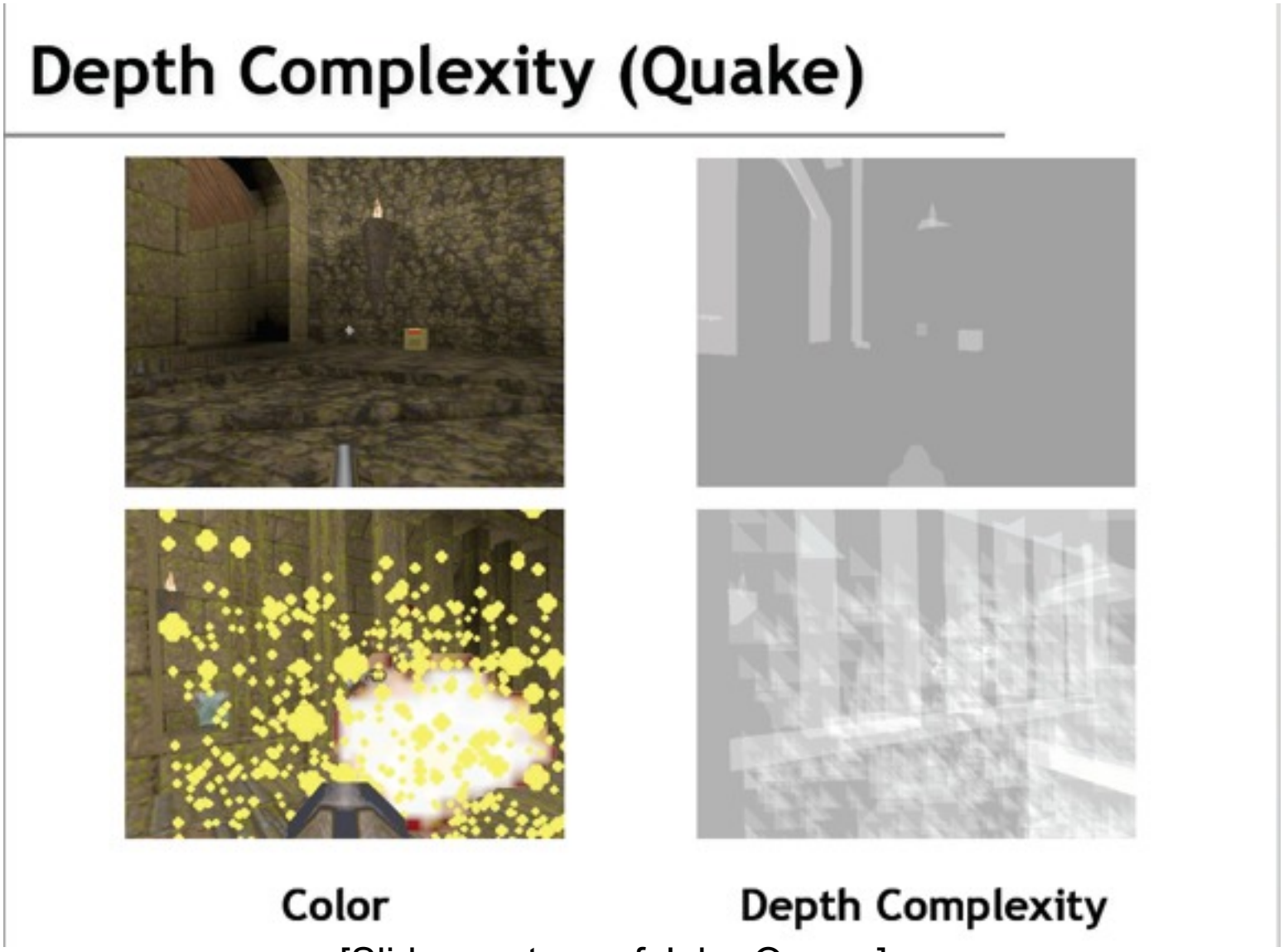
www.coolingzone.com

Slide above is courtesy of Kari Pulli at Nokia

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?



[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

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

- 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: 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% missrate

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...

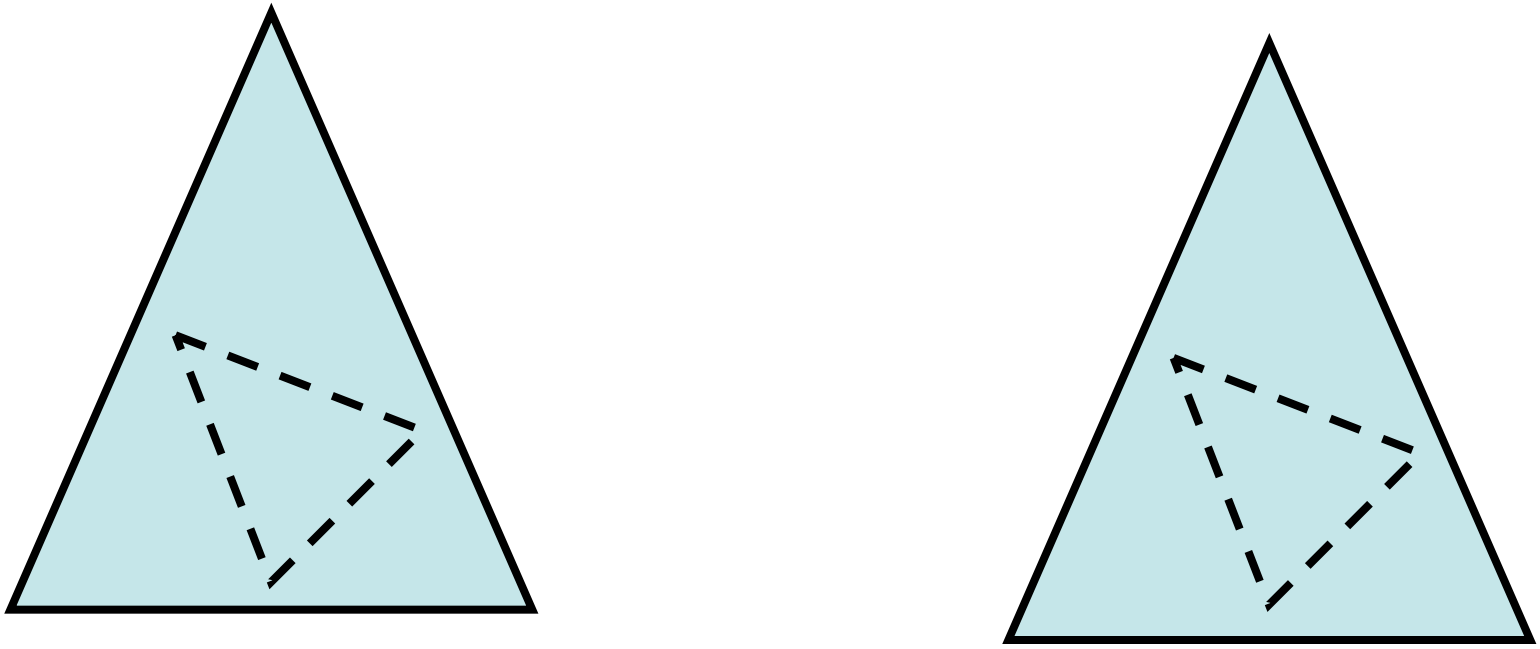
What else needs to be improved?

- $b = 4*3 + 2*(3 + 3 + 0.2*32) = 37$ 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: render-to-texture, e.g.
- 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?

Zmax vs Zmin



- Left: small triangle is behind big triangle
- Right: small triangle is in front of big triangle

Zmax-culling (1)

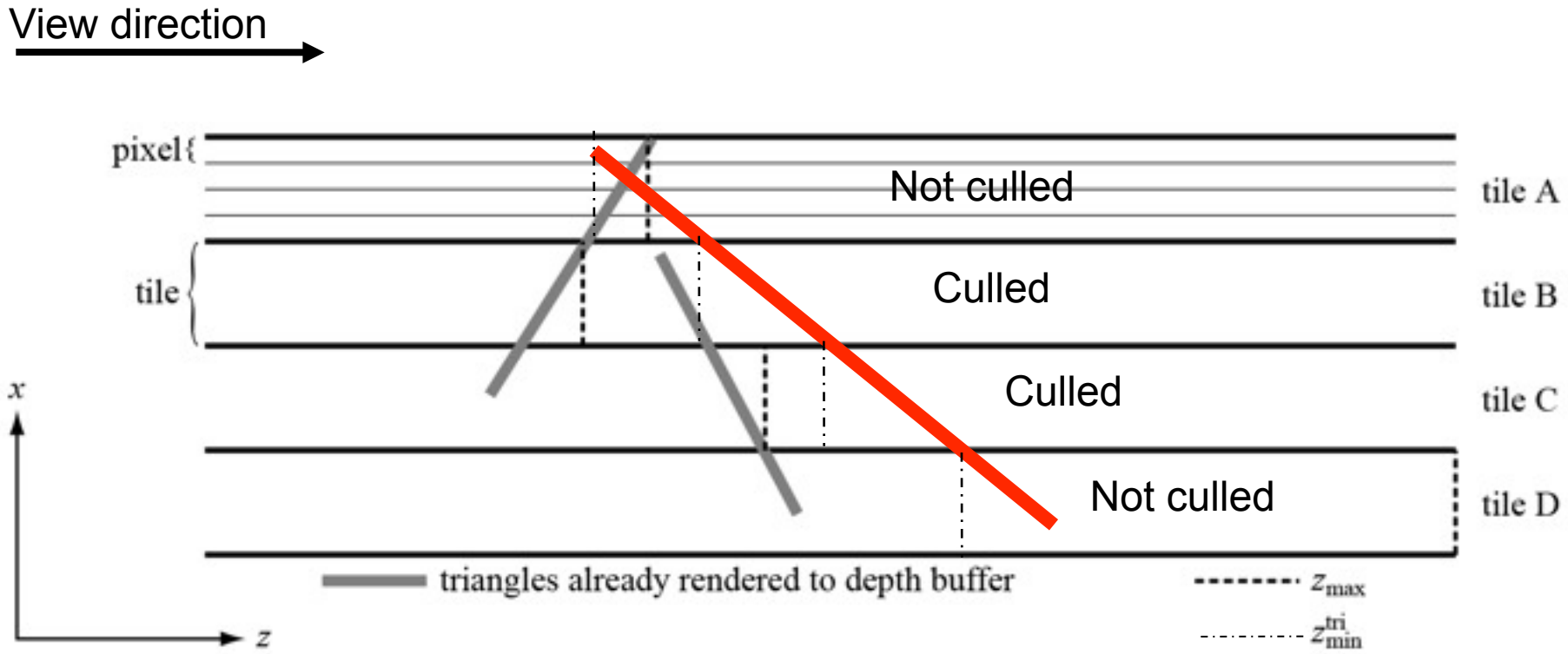
- What about a fragment that fails the depth test (if test is **`less_or_equal`**)?
  - i.e., the fragment is occluded (not visible)
- Ideally, we do not want to process them at all!

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

- We know that $o <= d$, 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

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

Zmax-culling example



- Now render red triangle

Zmax-culling (2)

- A tile is wxh pixels, and its depths:

$$d(i,j),\ i \in [0, w-1] \text{ and } j \in [0, h-1]$$

- The key is to, per tile, maintain:

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

- 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”: $z^{\text{tri}}_{\min}$
  - If the following is true, we know that triangle is occluded by zmax in tile, and can avoid depth reads: $z^{\text{tri}}_{\min} > z_{\max}$

Zmax-culling (3)

- Great, how compute $z_{\min}^{\text{tri}}$?
- 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

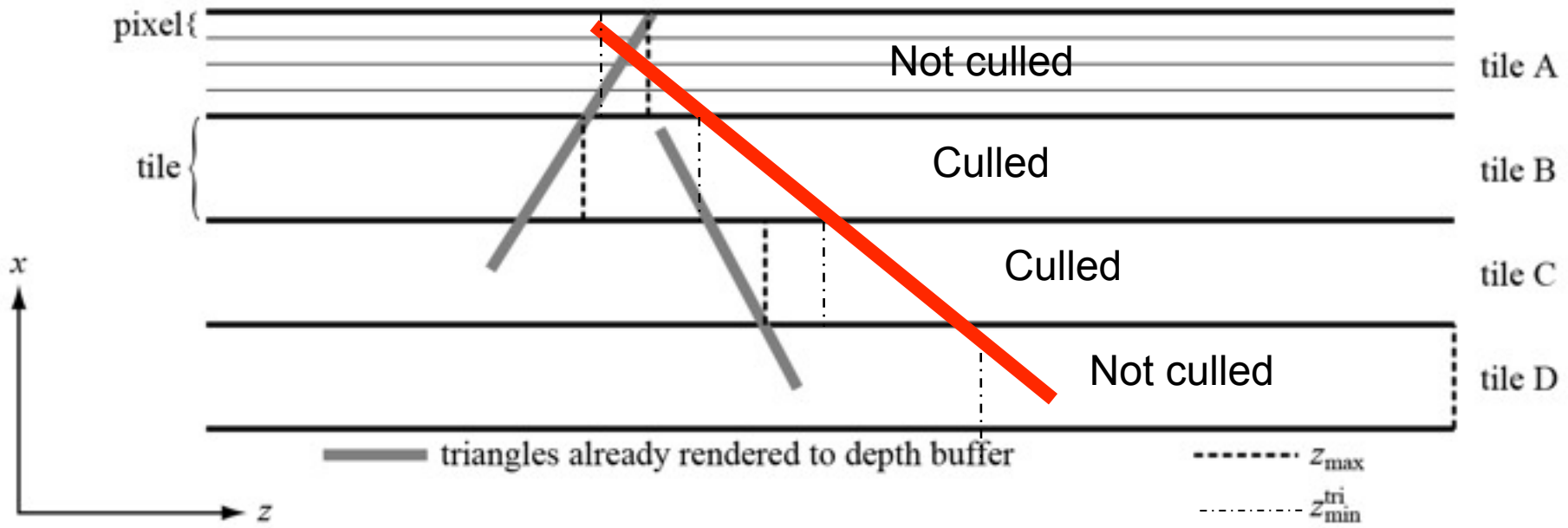
Zmax-culling (4)

- How make conservative estimate of $z_{\min}^{\text{tri}}$?
- 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)

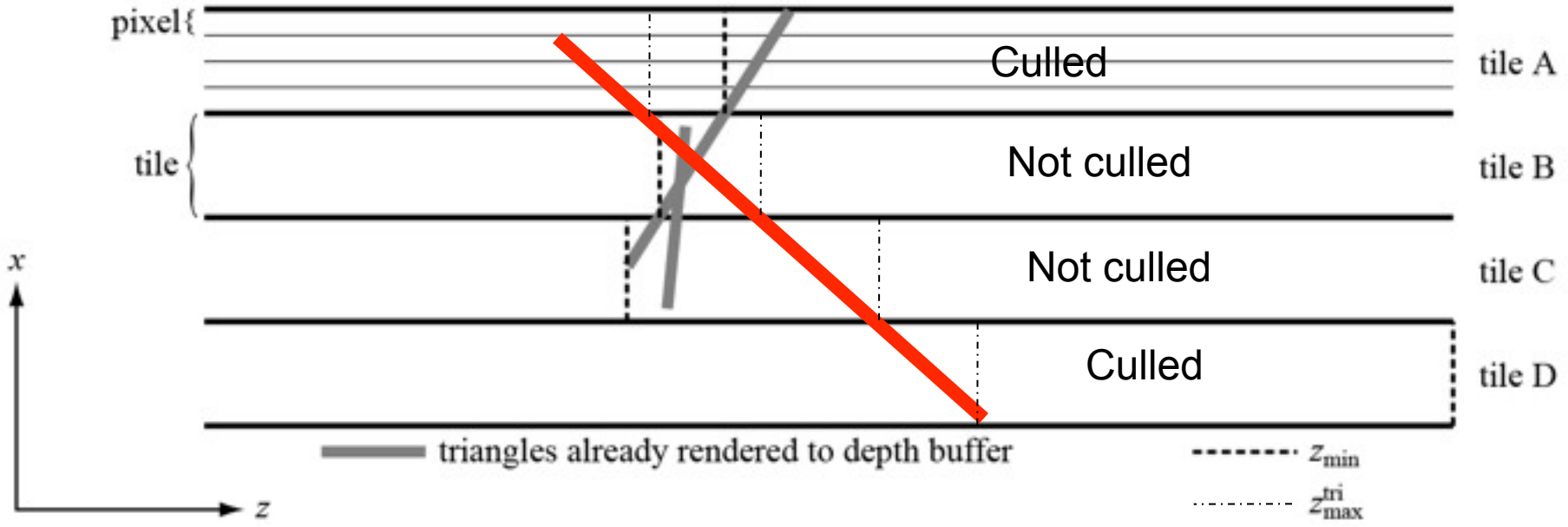
- Where should we store each tile's zmax-value?
  - Remember: we want to reduce memory bandwidth
  - On-chip memory, or if that takes too much memory, access the zmax's through a cache
- How should we update the zmax?
  - Can only become smaller
  - Only way: read all depths in tile, compute maximum... ☹
  - Could be expensive, but works well with depth buffer compression (next lecture)
- Some study has shown that depth reads are reduced by 10-32%...

Zmax-culling example (same example again)



- Now render red triangle

Zmin-culling example



- Red triangle is currently being rendered

Zmin-culling (1)

- Similar to Zmax-culling, but instead we store, per tile: $z_{\min} = \min_{ij}[d(i,j)]$
- When tiled traversal algorithm arrives at tile overlapping the triangle, we compute the ”maximum of the triangle’s z in that tile”: $z_{\max}^{\text{tri}}$
- This value can be computed using equivalent techniques for Zmax-culling...
- We can avoid depth reads, when the following is true: $z_{\max}^{\text{tri}} < z_{\min}$

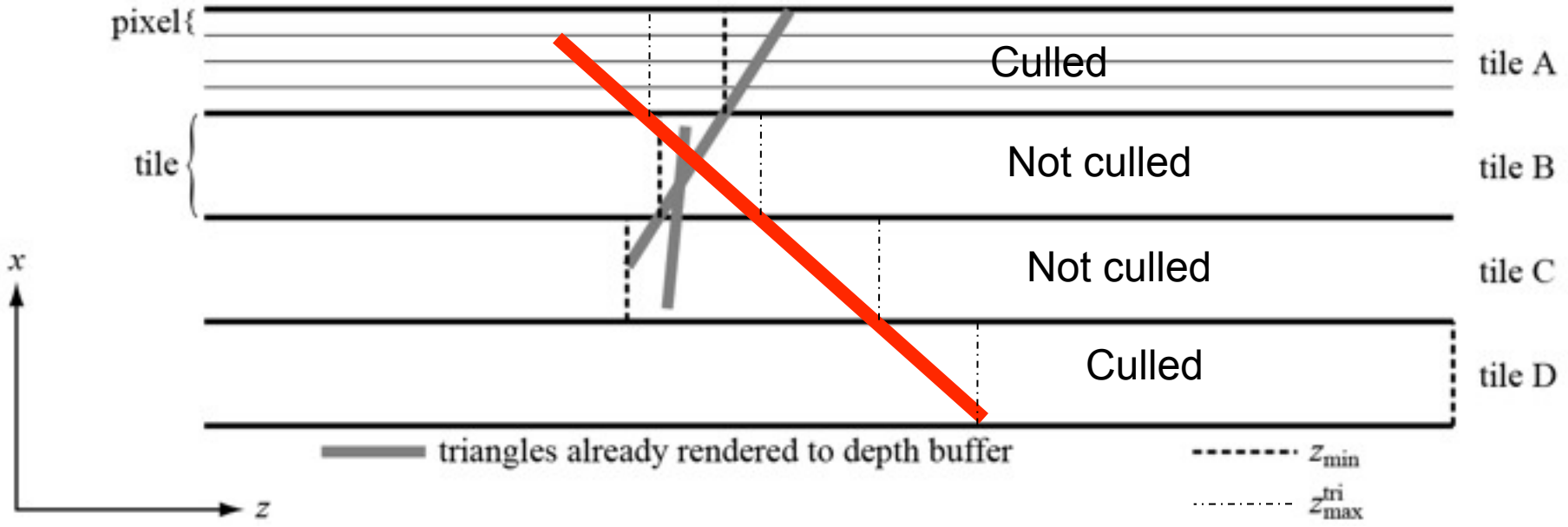
Zmin-culling (2)

Cull when:

$$z_{\max}^{\text{tri}} < z_{\min}$$

- 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 onchip or in cache (as Zmax)
- Updating zmin?
  - Simple: as soon as a depth, d, has been written that is smaller than $z_{\min}$, we update $z_{\min}=d$
  - i.e., no need to read all depth values in tile (as in Zmax)
  - Thus much much more feasible for mobiles!

Zmin-culling example again



- Red triangle is currently being rendered

Zmin-culling results

- Algorithm developed for mobile devices
- Applications for mobile devices were used for benchmarking

- d=0.65 d=2.5 d=1.5
- Reduction in depth reads:
- 84% 69% 49%


Why did 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, 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
- 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!

On Thursday...

- Lecture will deal with:
  - Real-time depth buffer compression
- Colors can also be compressed, but very little public info is known about this...
  - Patent issues ☹

The end