
# Texture and Depth Compression

LUGG
Lund University Graphics Group

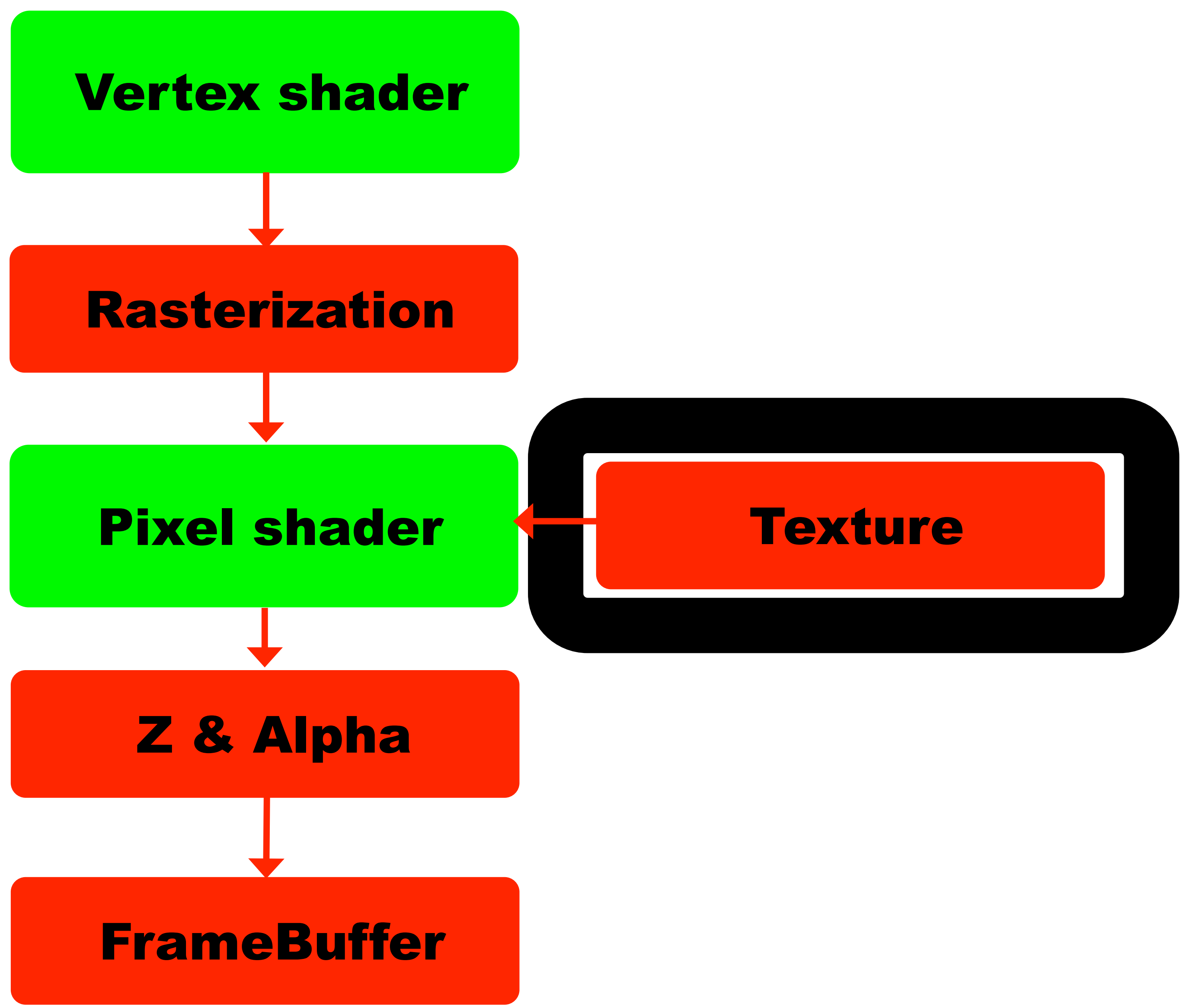
Michael Doggett
Department of Computer Science
Lund University

2023

# Project

- 3D graphics project
  - Implement 3D graphics algorithm(s)
  - C++/OpenGL(Lab2)/3D engine
  - Demo, Game
  - **Proposal - Long paragraph by next Monday (Nov 27)**
- More in the next lecture

# Today's stage of the Graphics Pipeline

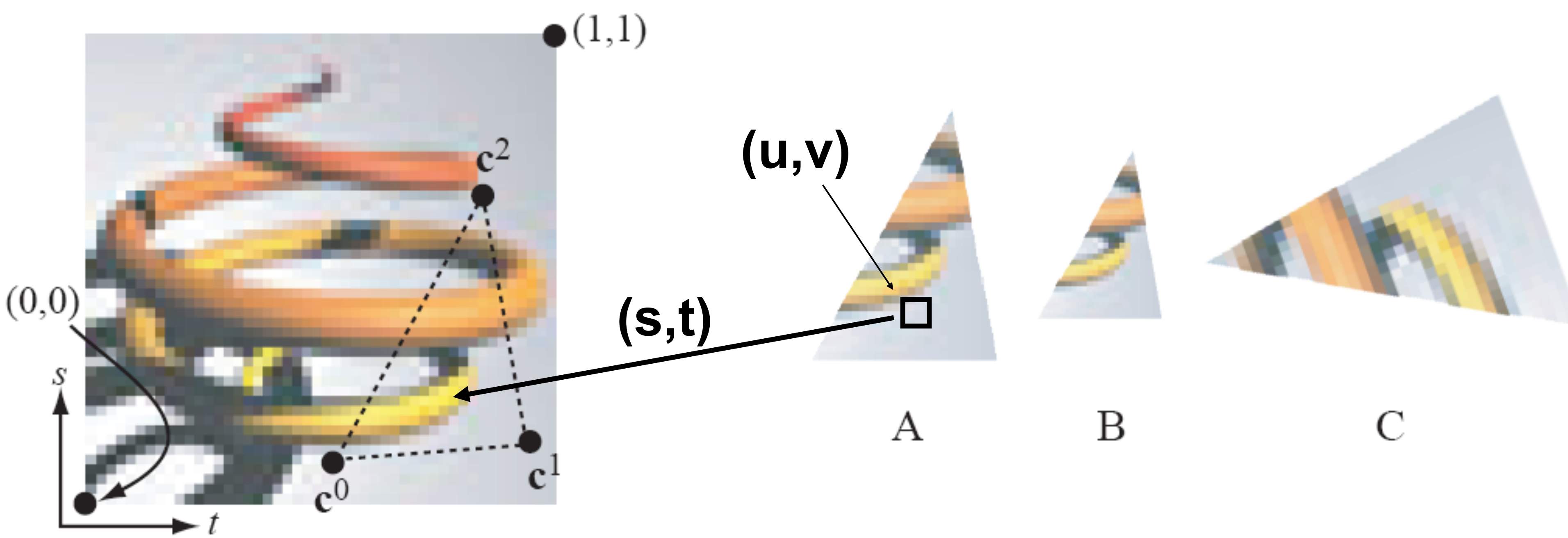


# Texturing – the tiny details


**Image from ”lpics”-paper by Pellacini et al. SIGGRAPH 2005**
**PIXAR Animation Studios**

- Surprisingly simple technique
  - –Extremely powerful, especially with programmable shaders
  - –Simplest form: ”glue” images onto surfaces (or lines, or points)

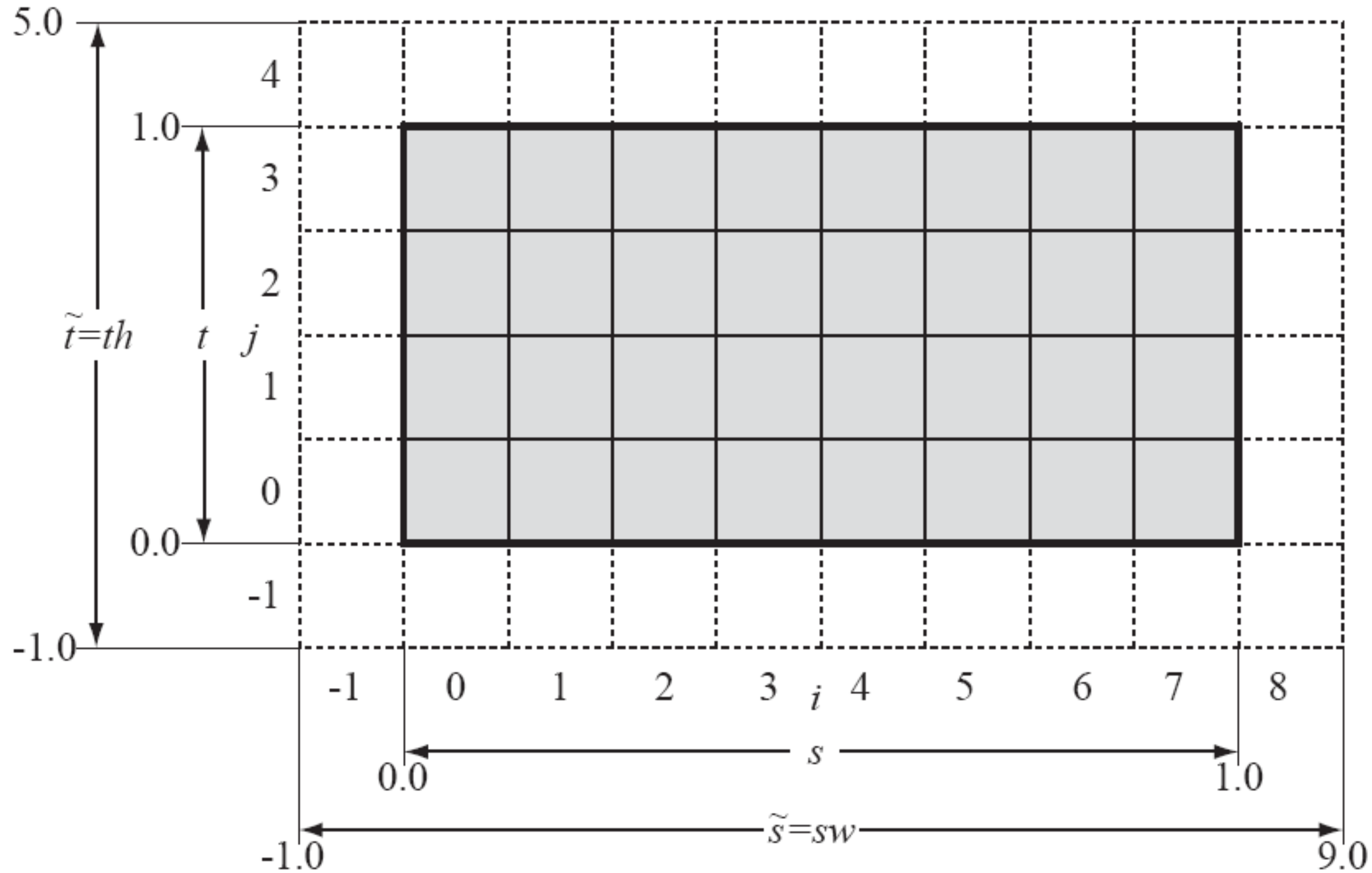
# Texture space, $(s,t)$




- Texture resolution, often $2^a$ x $2^b$ texels
- The $\mathbf{c}^k$ are *texture coordinates*, and belong to a triangle's vertices
- When rasterizing a triangle, we get (u,v) interpolation parameters for each pixel (x,y):
  - Thus the texture coords at (x,y) are:

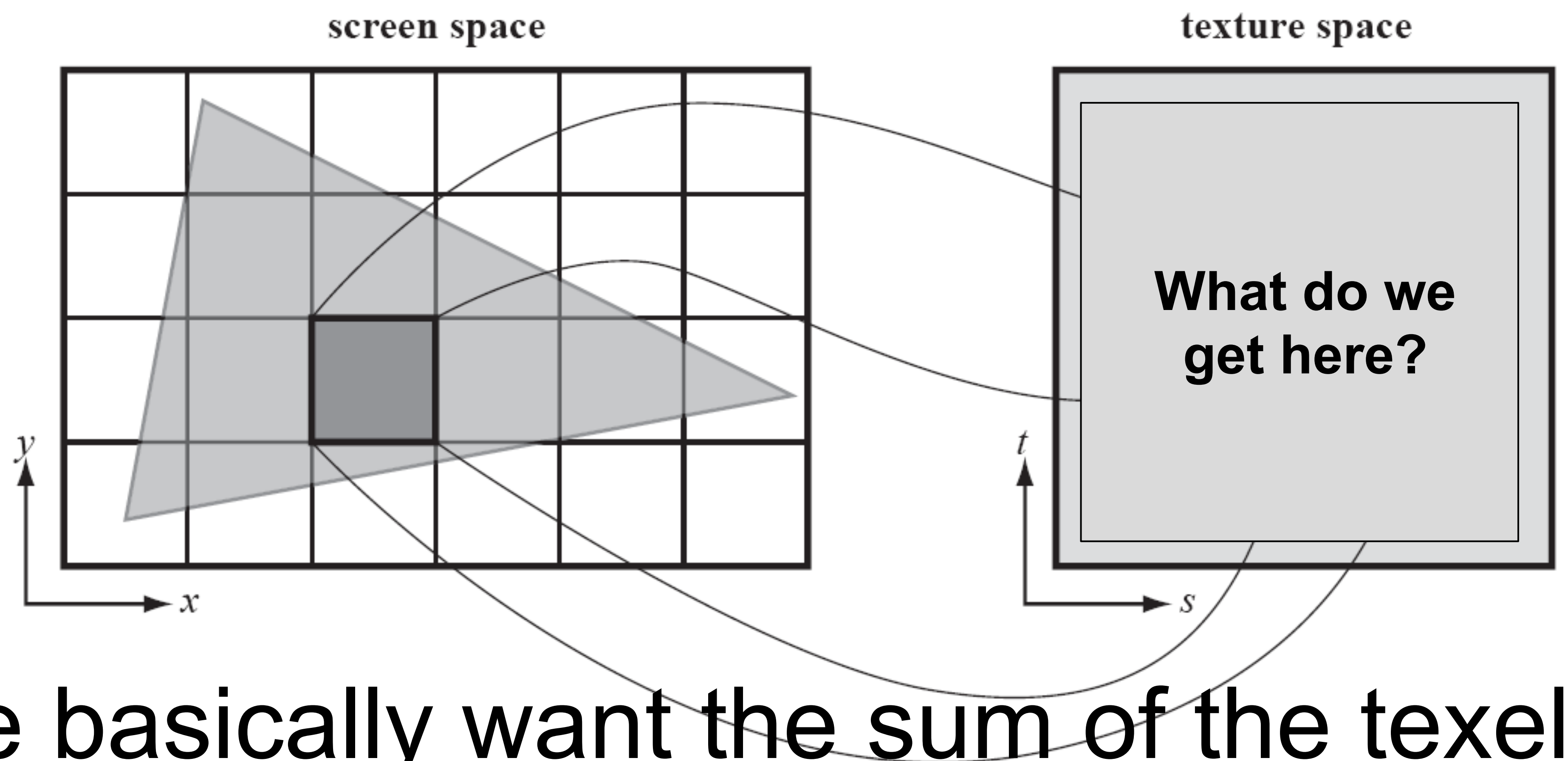
$$(s,t) = (1-u-v)\mathbf{c}^0 + u\mathbf{c}^1 + v\mathbf{c}^2$$

# A texture image + coord systems



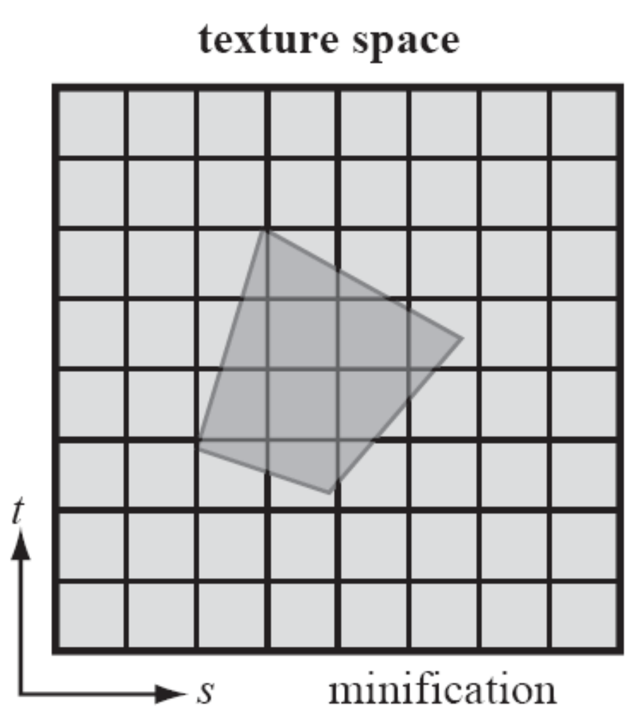

- An $w$x$h$=8x4 texture.
  – $(s,t)$ are independent of texture resolution
  – $(sw,th)$ depend on the resolution, and are used to access texels…
- Each pixel in a Texture is called a "**Texel**"

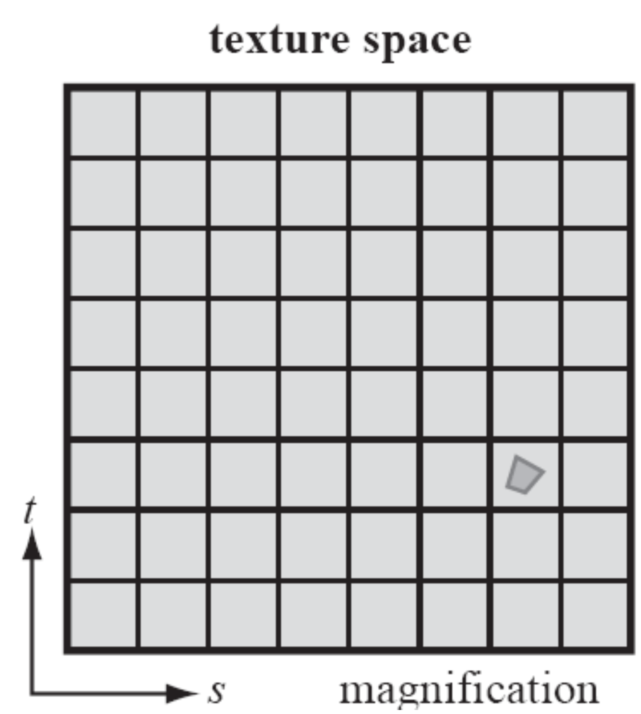
# Texture filtering




- We basically want the sum of the texels in the footprint (dark gray) to the right

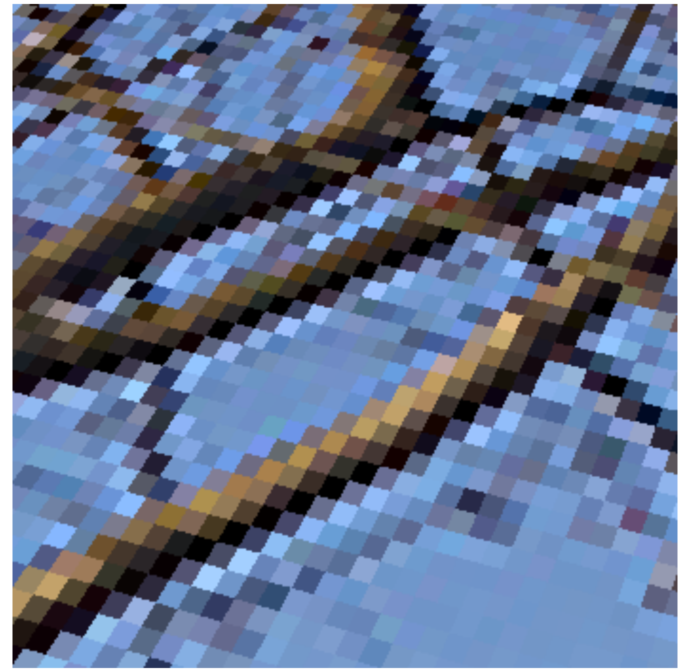
MINIFICATION



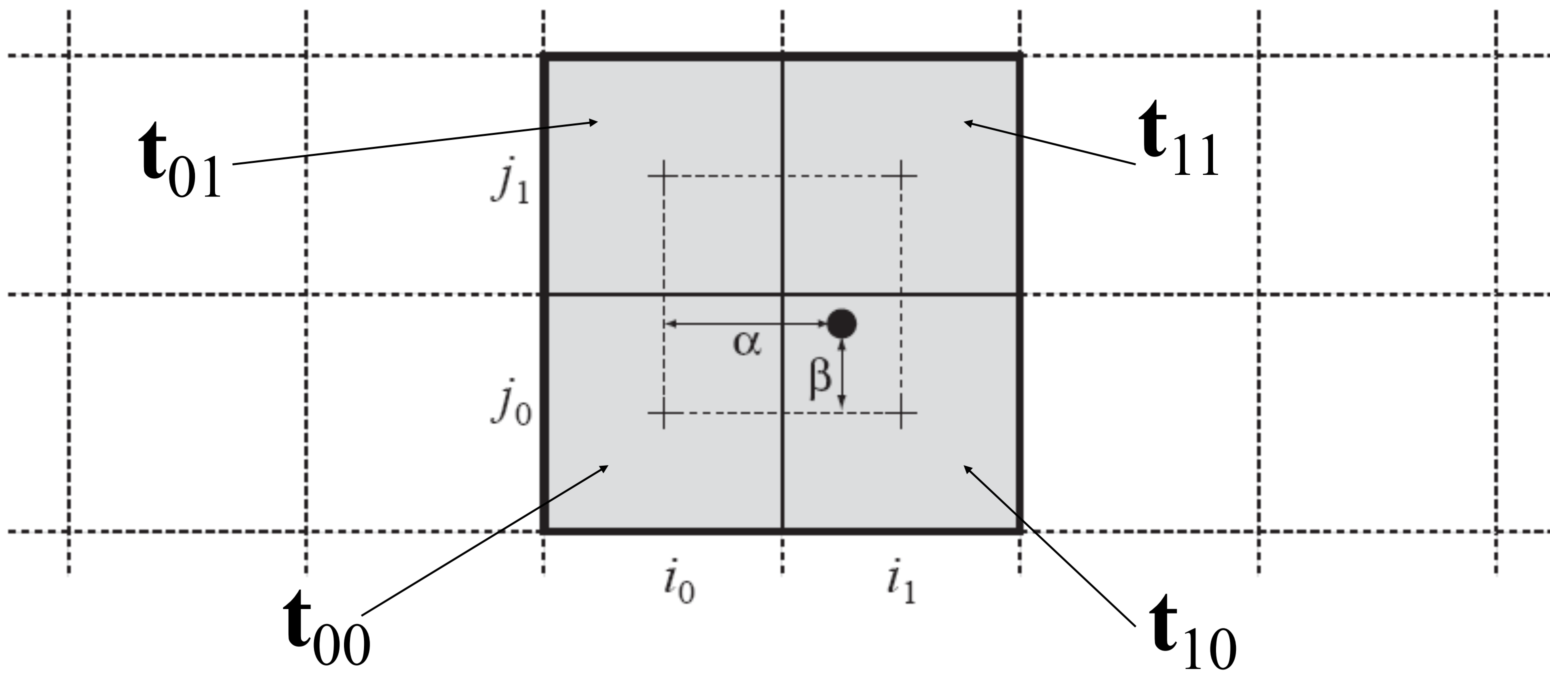

MAGNIFICATION

# Texture magnification (1)



- Middle: **nearest neighbor** – just pick nearest texel
- Right: **bilinear** filtering: use the four closest texels, and weight them according to actual sampling point

# Texture magnification (2)




- Bilinear filtering is simply, linear filtering in x:

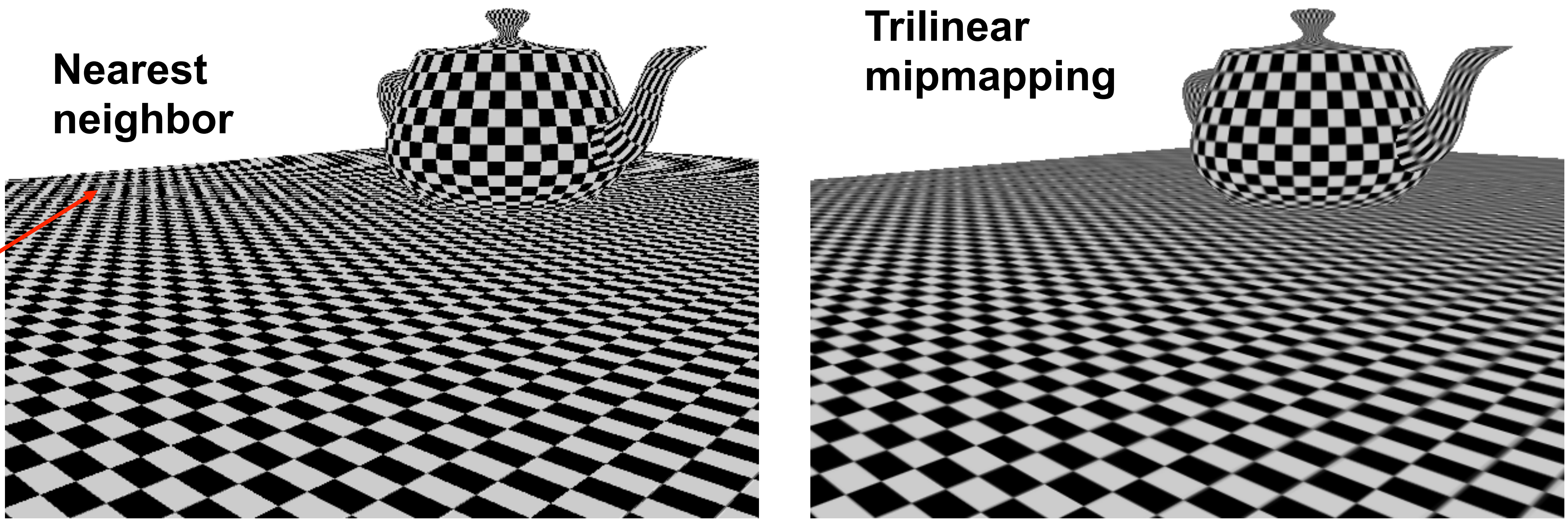
$$\begin{aligned}\mathbf{a} &= (1-\alpha)\mathbf{t}_{00} + \alpha\mathbf{t}_{10} \\ \mathbf{b} &= (1-\alpha)\mathbf{t}_{01} + \alpha\mathbf{t}_{11}\end{aligned}$$

- Followed by linear filtering in y:

$$\mathbf{f} = (1-\beta)\mathbf{a} + \beta\mathbf{b}$$

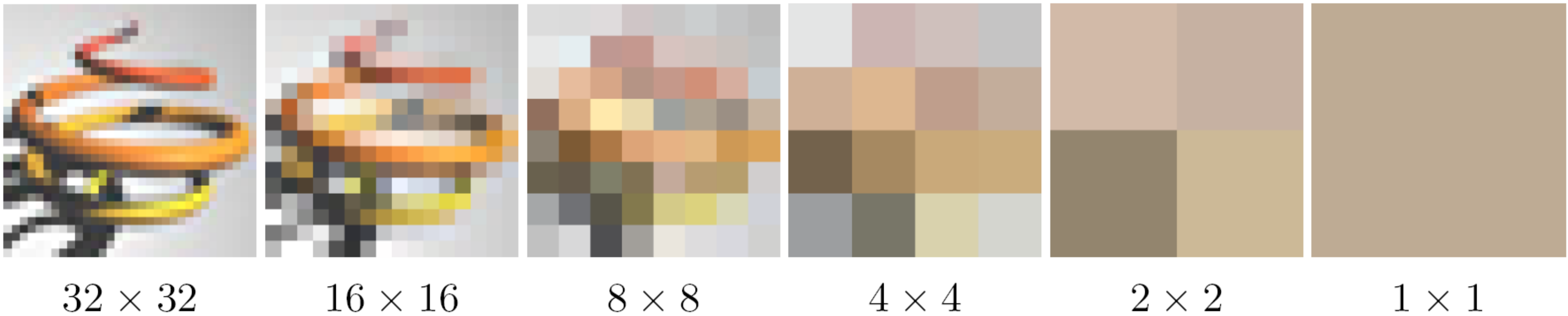
# Texture minification

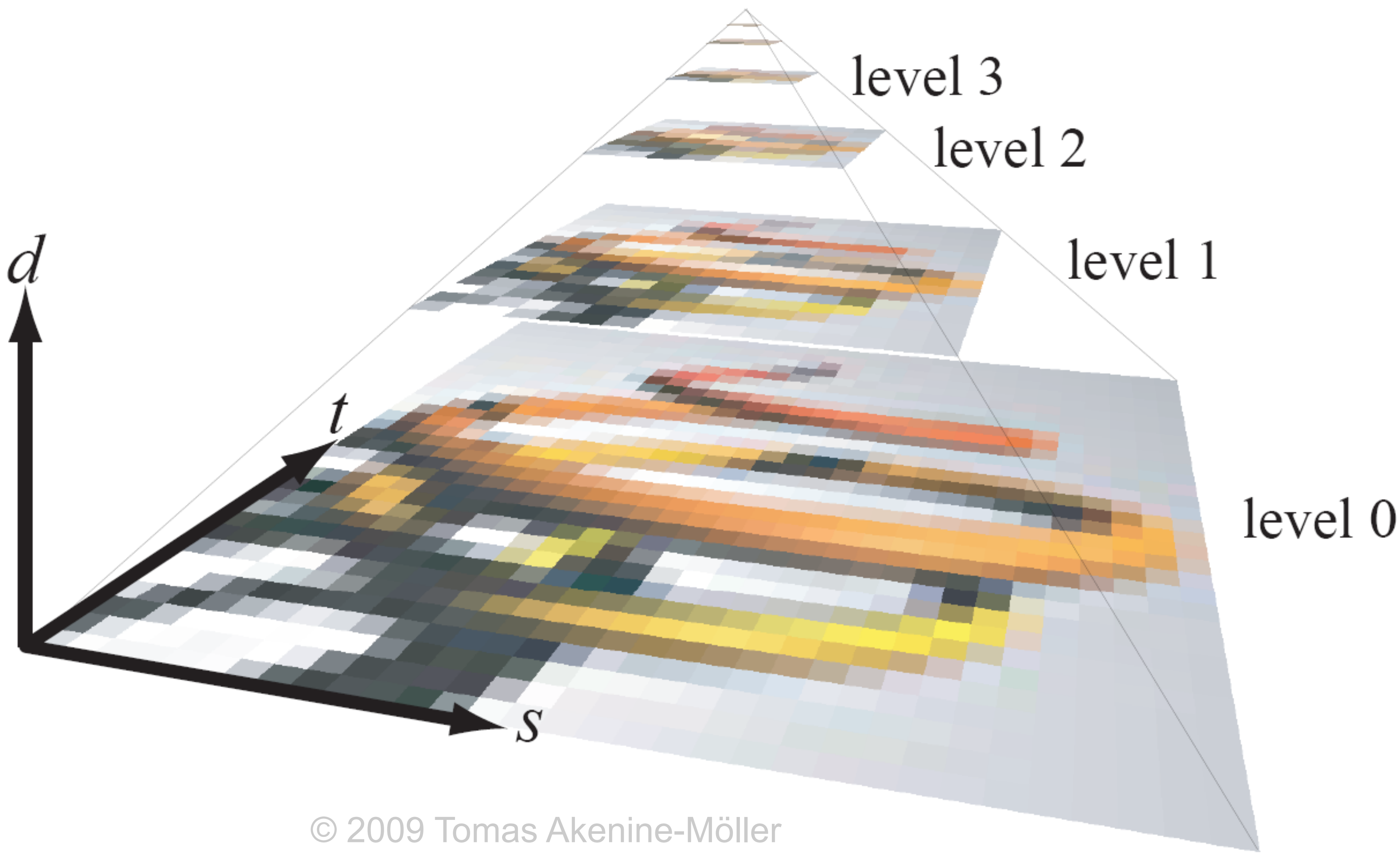
- If nearest neighbor or bilinear filtering is used, then serious flickering will result
  - –Extremely annoying




**For a pixel here, there is a 50% chance of getting a black texel**

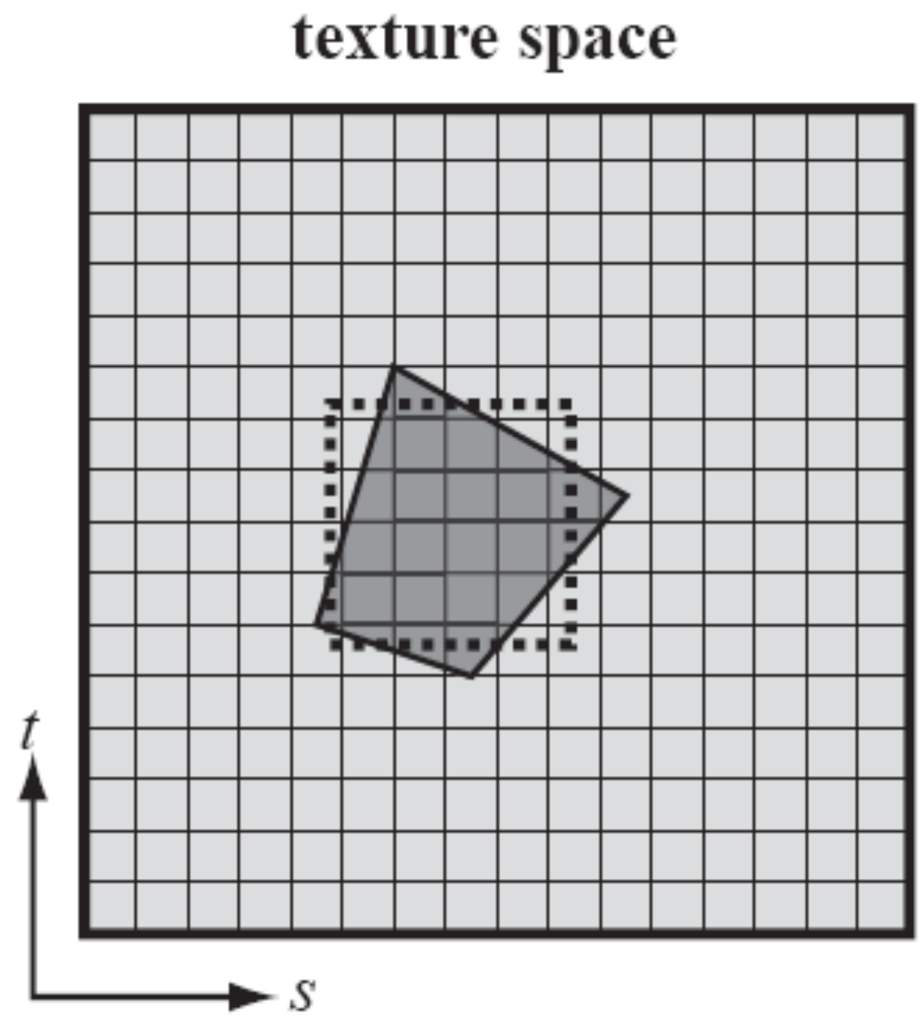
# Texture minification: mipmapping



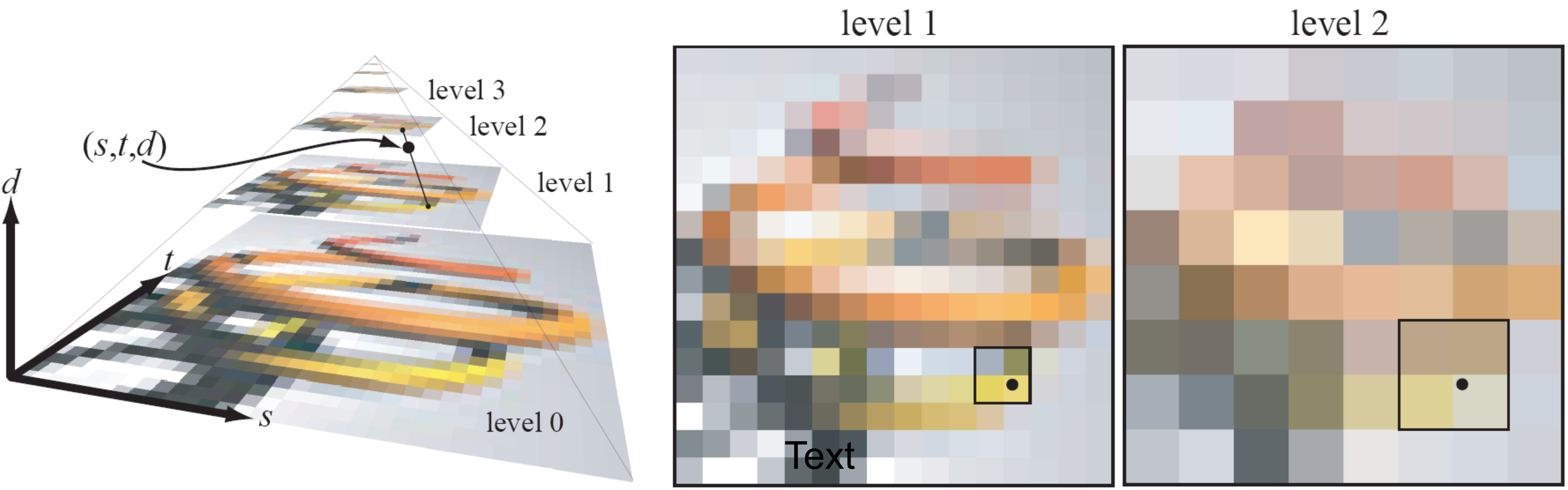

**An image pyramid of low-pass filtered images**

# Trilinear Mipmapping (1)




- Basic idea:
  - –**Approximate** (dark gray footprint) with square
  - –Then we can use texels in mipmap pyramid

# Trilinear mipmapping (2)




- Compute $d$ (Level of Detail, LOD) (see Chapter 5), and then use two closest mipmap levels
  - In example above, level 1 & 2
- Bilinear filtering in each level, and then linear blend between these colors → trilinear interpolation
- Nice bonus: makes for much better texture cache usage

# Representation of textures in memory

- Normally, a 4x4 texture is stored as:
  - $RGBA_0$, $RGBA_1$, $RGBA_2$, ... $RGBA_{15}$
- What if, we traverse in the vertical direction?
  - E.g., accessing 1,5,9,13
  - Quite bad if we read, say, 4 texels into the cache at a time

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

- Are better texel orderings possible?
- With representation to the right, only two blocks are read into the cache

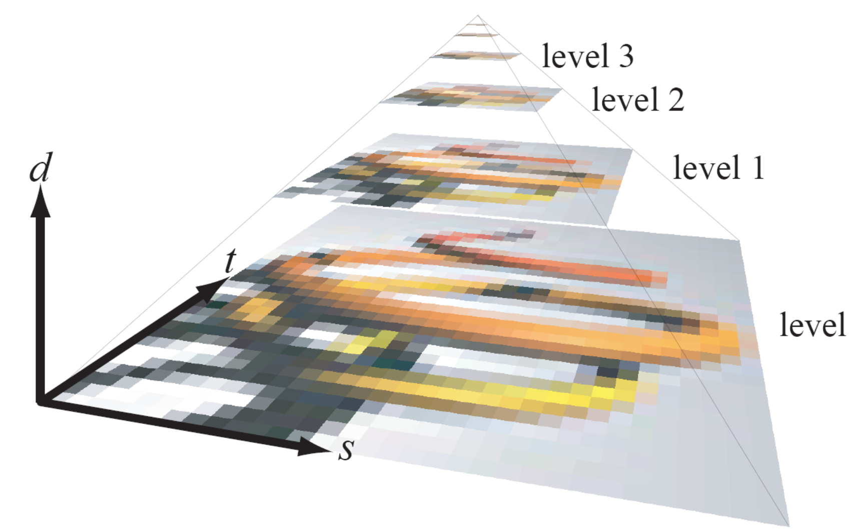
| | | | |
|---|---|---|---|
| 0 | 1 | 4 | 5 |
| 2 | 3 | 6 | 7 |
| 8 | 9 | 12 | 13 |
| 10 | 11 | 14 | 15 |

- This representation will (on average) get the same performance regardless of traversal direction!!!

# Representation of textures in memory

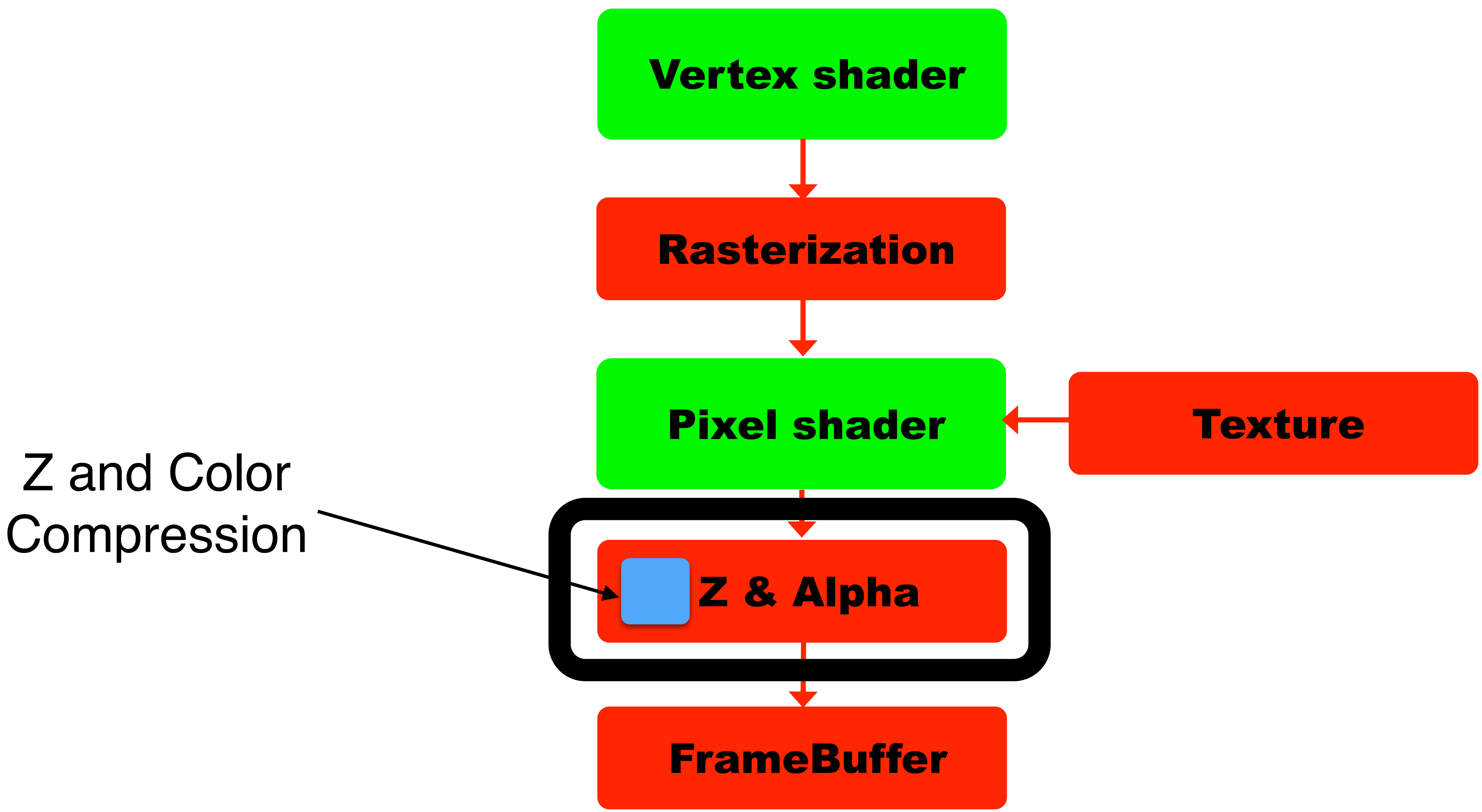
| | | | |
|---|---|---|---|
| 0 | 1 | 4 | 5 |
| 2 | 3 | 6 | 7 |
| 8 | 9 | 12 | 13 |
| 10 | 11 | 14 | 15 |

- This is called a ”blocked” or ”tiled” representation - “z-order”
- It is a 4D structure: first find 2x2 block, then texel in block




- In general, we have an $n$x$n$ block...
  - $n$ is power of 2
- Good representation for texture caches

# Graphics Pipeline
# Z compression



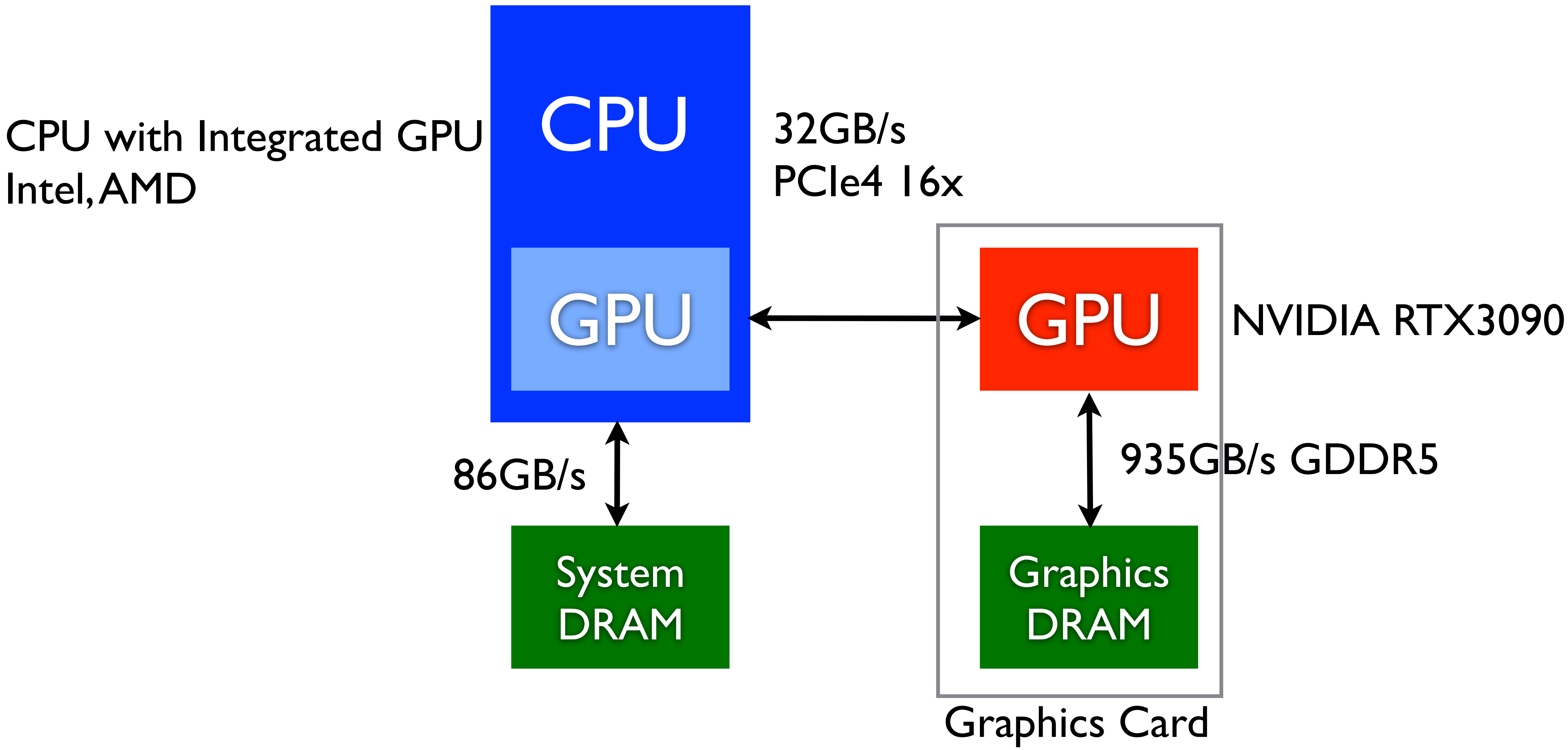
# Z & Alpha Performance

- Recall Memory Bandwidth has a big impact on 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

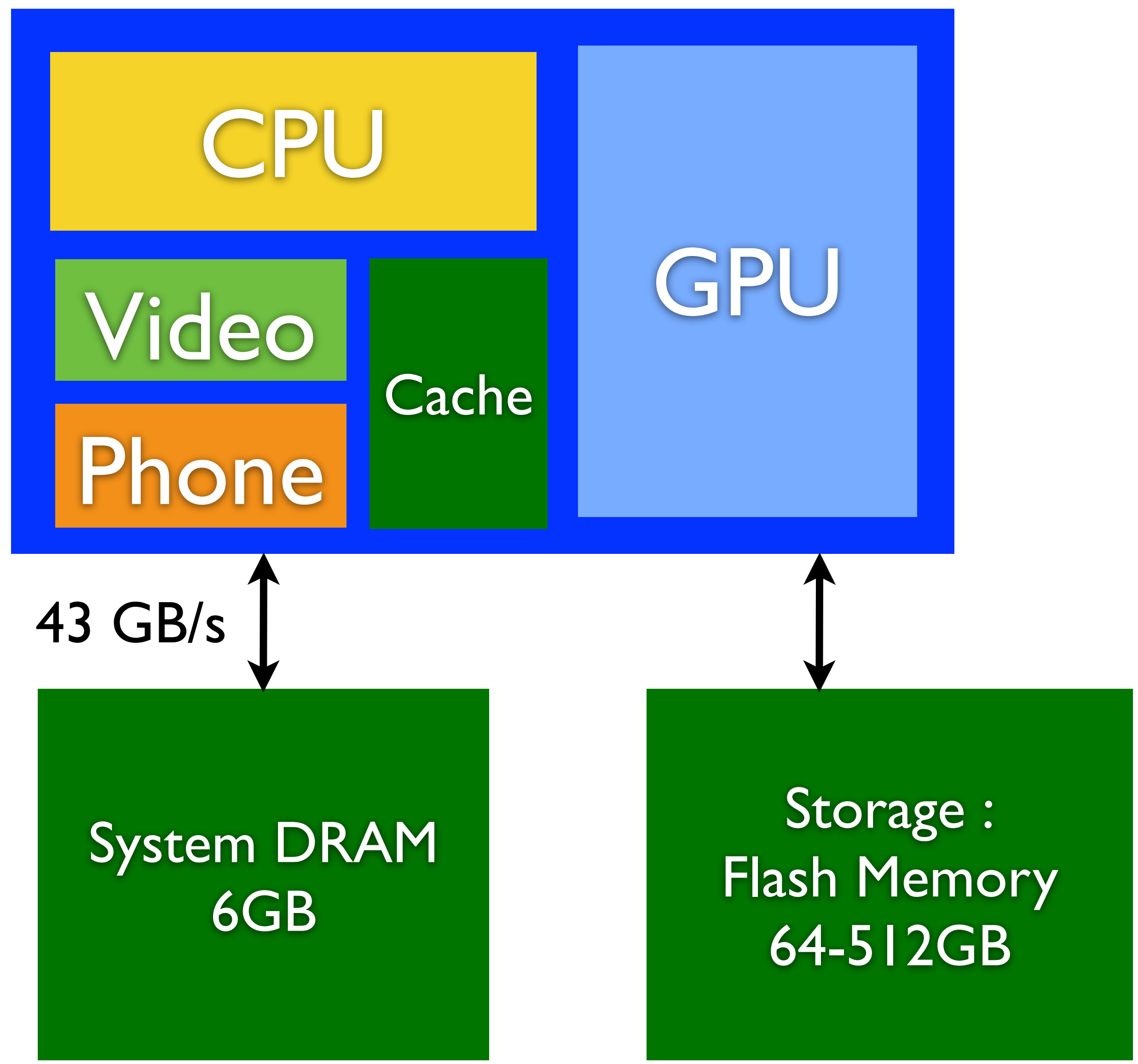
# DRAM overview

- 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 clock
  - QDR, then GDDR (Graphics DDR) versions 2, 3, 4, 5, 6, 6X
- HBM - High Bandwidth Memory
  - 3D-stacked DRAM

# PC memory architecture



# Mobile Memory Architecture




Based on 2020 iPhone12

# Why use depth buffering for visibility determination?

> hardware 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.

**From "A Characterisation of Ten Hidden-Surface Algorithms",**
**Ivan Sutherland, Robert Sproul, and Robert Schumacker (ACM Computing Surverys, March 1974)**

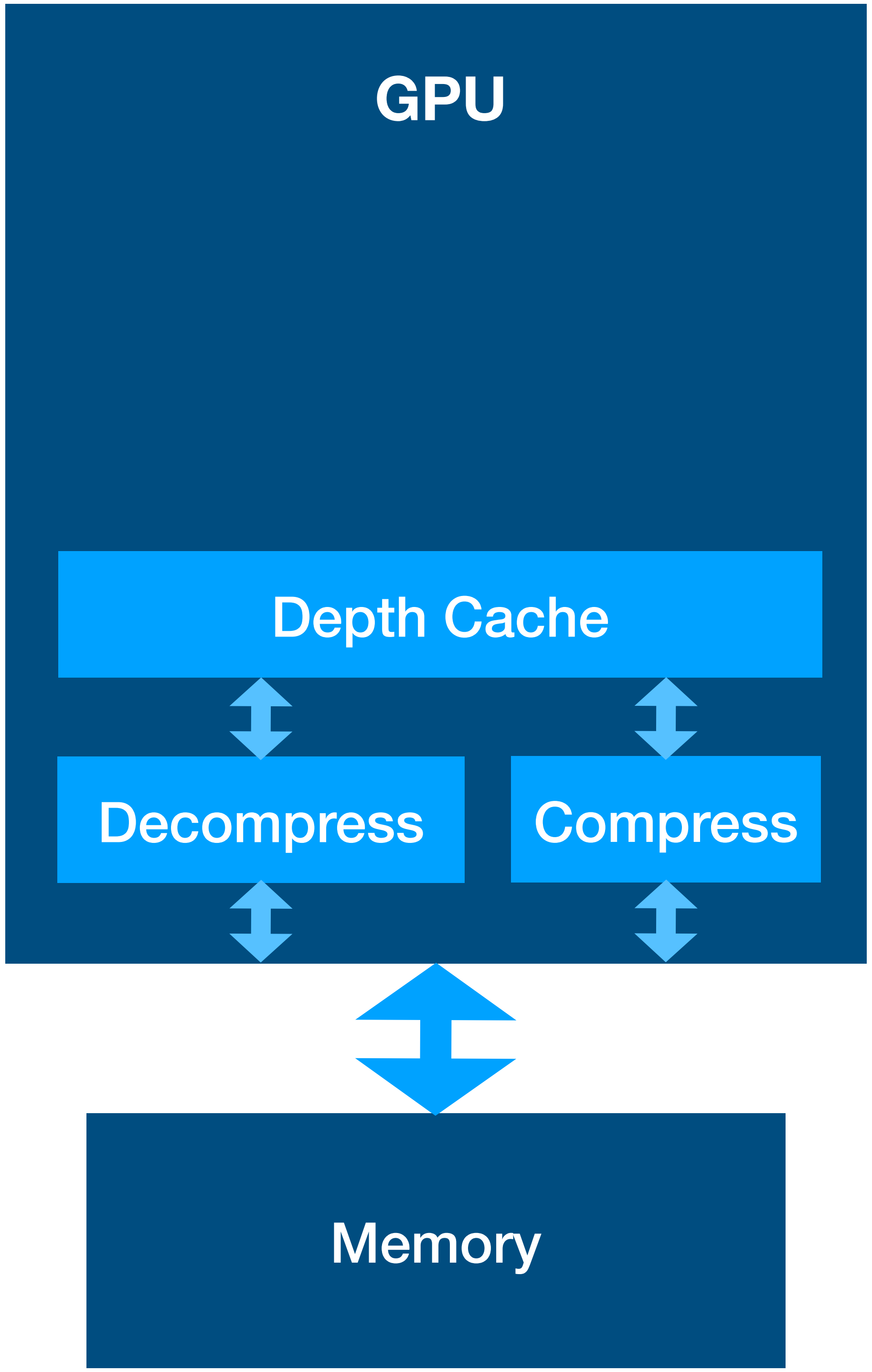
- **The "brute-force approach" is depth buffering**
- **Other methods considered used polygon sorting**
- **Depth-buffering was very expensive comparatively,**
  - **but won when DRAM became cheap**

# Depth Buffer Bandwidth

- A major component of memory bandwidth
- Hierarchical Z reduced the cost
- Compression will reduce that cost further

# Depth Buffer Compression

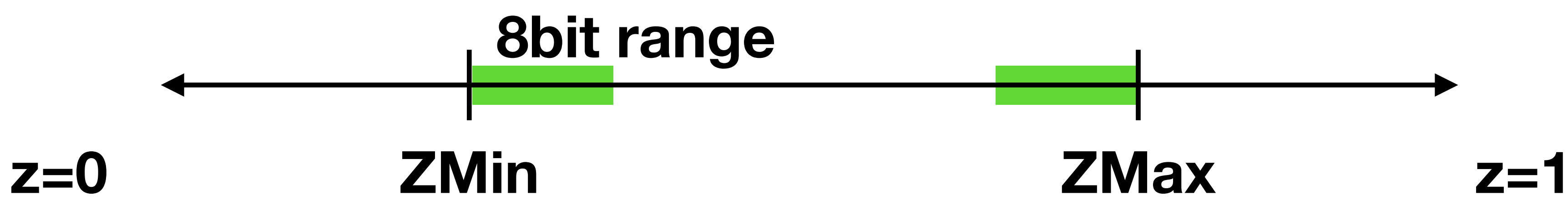
- Custom hardware to compress/decompress data
    - Compress and write, Read and decompress
- Depth Buffer stored in graphics memory compressed
- Depth on-chip Cache, can be compressed
    - If not changed, no need to write back
- Reduces bandwidth, not necessarily storage
    - Must always have memory for decompressed mode



# Depth Buffer Compression

- Very little public information
- Highly coherent set of values
  - Depth is linear in screen space
  - Could be all the same triangle
- Pixel tiles, e.g. 8x8

# Depth Offset Compression




- Use ZMin and ZMax values
    - Store Z as offset from these
    - Can use limited number of bits as offsets
        - Offset must be within limited range of ZMin or ZMax
- Example storage
    - Use 8 bits for each offset, reduced from 24 bits for Z
- Can use use MSBs for ZMin/ZMax, and LSBs for offsets
    - Allows concatenation, so don’t need adders/subtractors

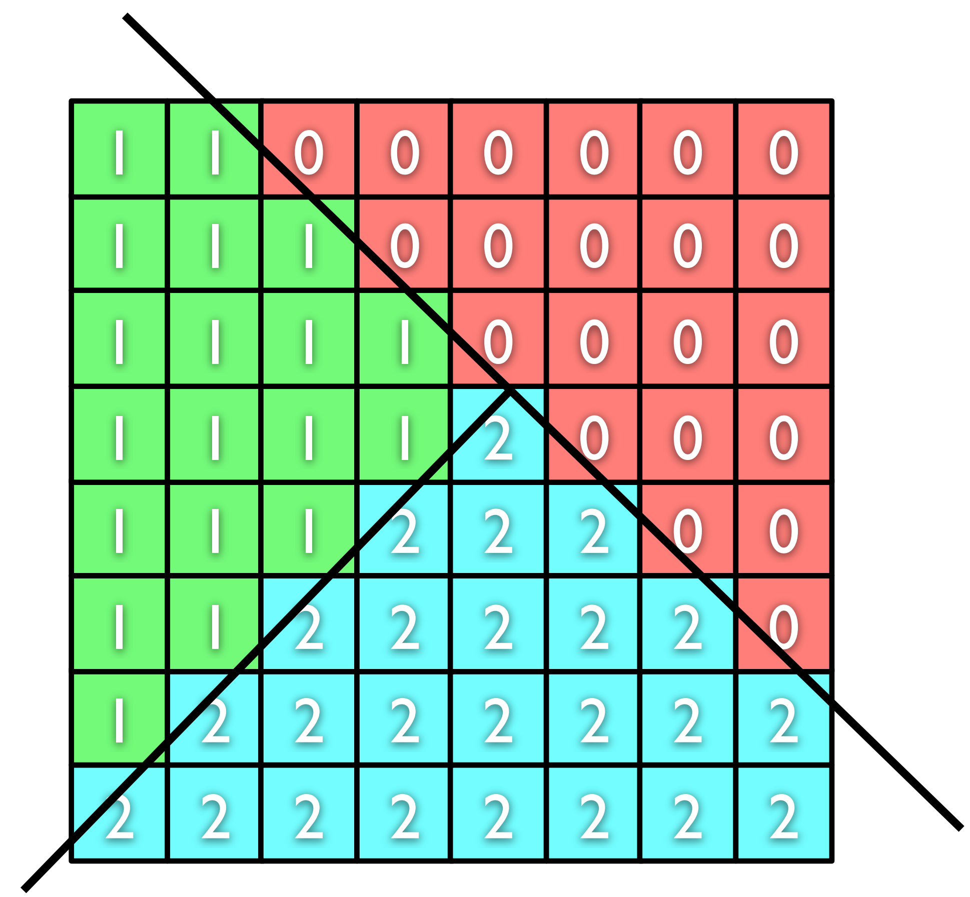
# Plane Equation Compression

- 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 (dz/dy) 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 : $Z_c$, x slope, y slope
- Plane 1 : $Z_c$, x slope, y slope
- Plane 2 : $Z_c$, x slope, y slope

- Plane Equations
  - $Z_c$ is 3 bytes, slopes are 2 bytes
  - 3 x (3 + 2 + 2)Bytes = 21Bytes
- Indexes
  - 64 x 2bits = 16Bytes
- Compressed
  - 37Bytes
- Uncompressed
  - 64 x 3Bytes = 192Bytes
- Compression ratio
  - 19%



# Next ...

- Tuesday and Wednesday
  - Assignment 2 marking in Pluto
- Next lecture
  - Antialiasing
  - Texture Compression
  - Start **project**