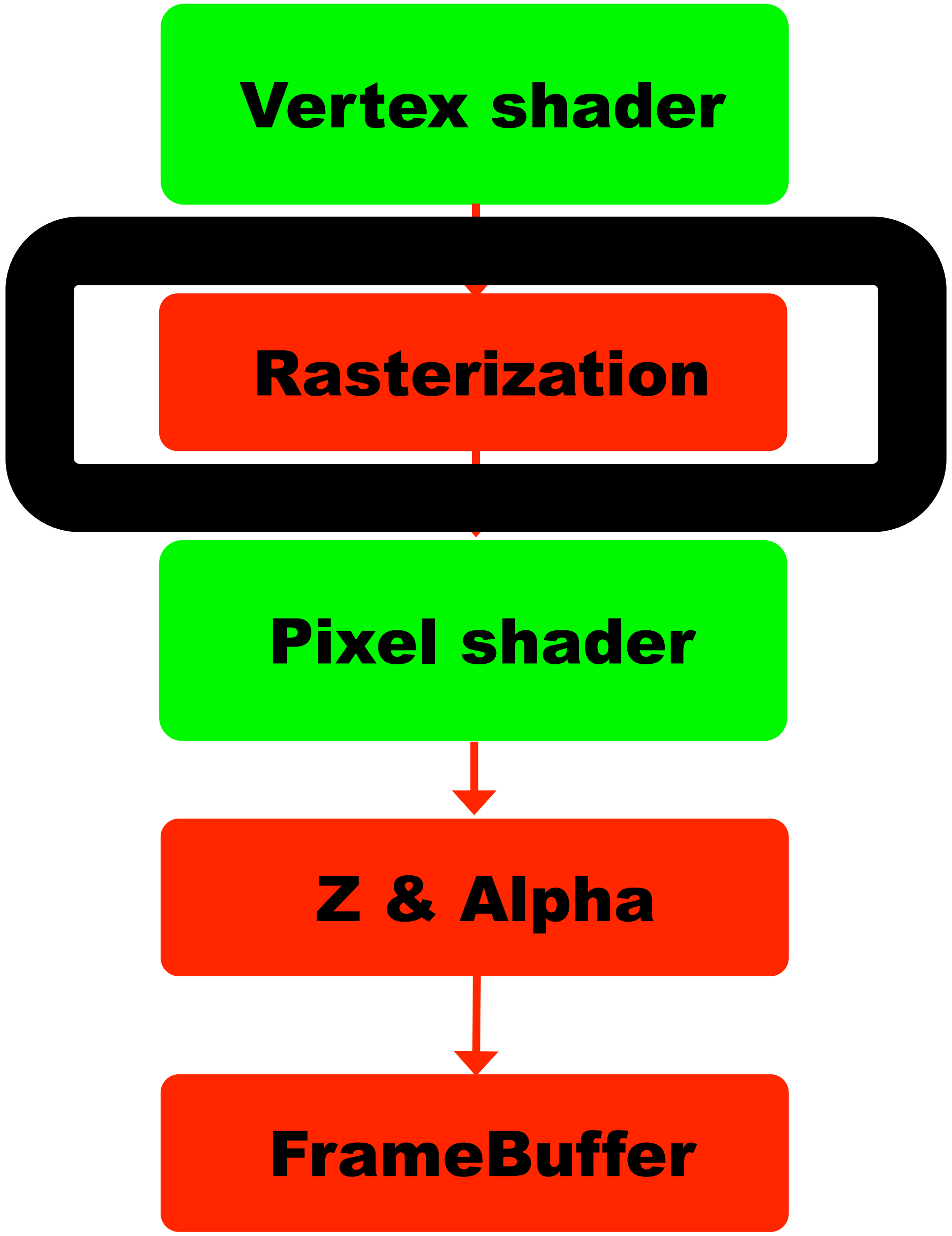
Fixed point math, texturing and texture caching

LUGG
Lund University Graphics Group

Michael Doggett
Department of Computer Science
Lund university

Last week's stage of the Graphics Pipeline



Vertex shader

Rasterization

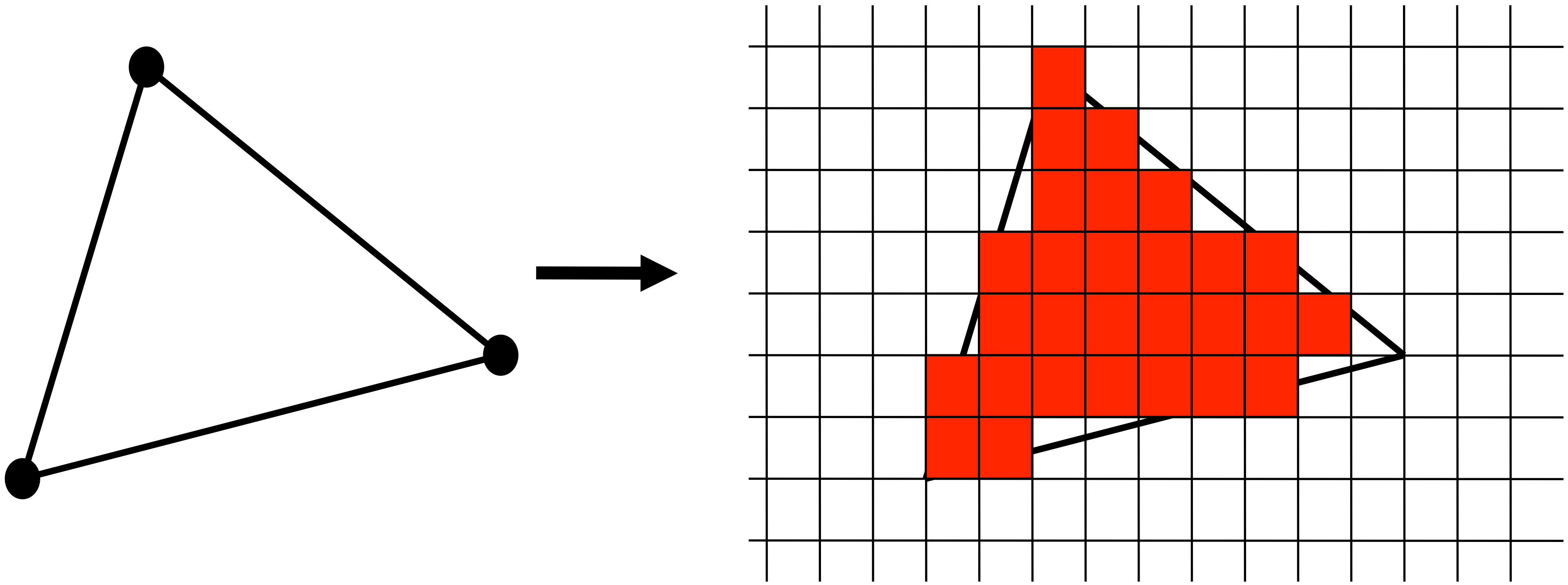
Pixel shader

Z & Alpha

FrameBuffer

Last week

Rasterization



Edge functions

Vertex positioning

Traversal

Interpolation

But first, Assignment 1!

- C++ programming, but very localized in functions where you should add code
  - C++ should be no problem (if it is, then ask on the forum)
- Only uses simple OpenGL
  - should work on any GPU
    - But requires Windows

Assignment 1

- Two small parts in this assignment:
  - Find three bad things in small scenes
    - Fix the code so that correct behaviour is obtained
  - Use a texture cache
    - Should be able to reduce texture bandwidth to 10-15%

Overview

- **Theory:**
  - Fixed-point math (Appendix A – online)
  - Texturing (Chapter 5 – online)
  - Texture caching (see assigned papers)
    - Caches (Section 5.5 in notes)
  - For assignment 1, it will help to read chapters 2 and 3 as well (online)

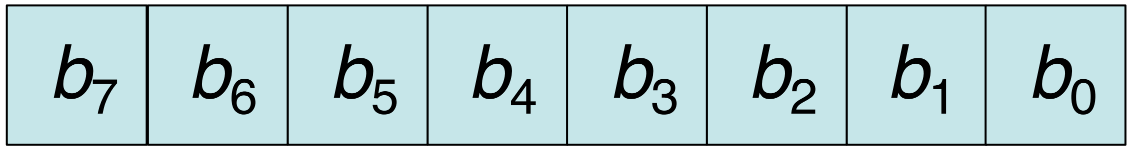
- **Practice:**
  - More about the rasterizer framework for assignment 1
  - More about the actual assignment

Fixed-point math

- Not floating point...
- Good to know!
- Essential for hardware design
- Can be used for performance optimizations

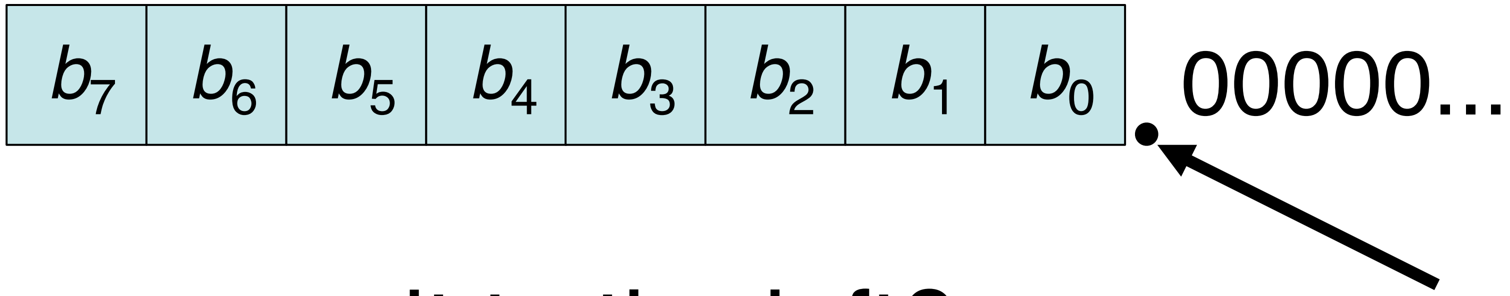
Integer vs fixed-point

- An 8-bit integer:



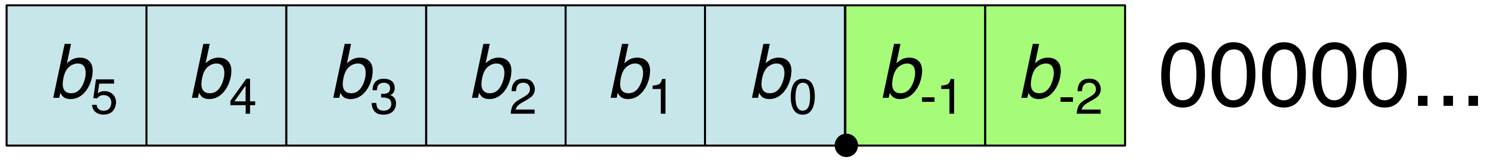
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

- b_i is ”worth” 2^i as usual
  - –But where is the decimal point?



b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	00000...

- What if we move it to the left?

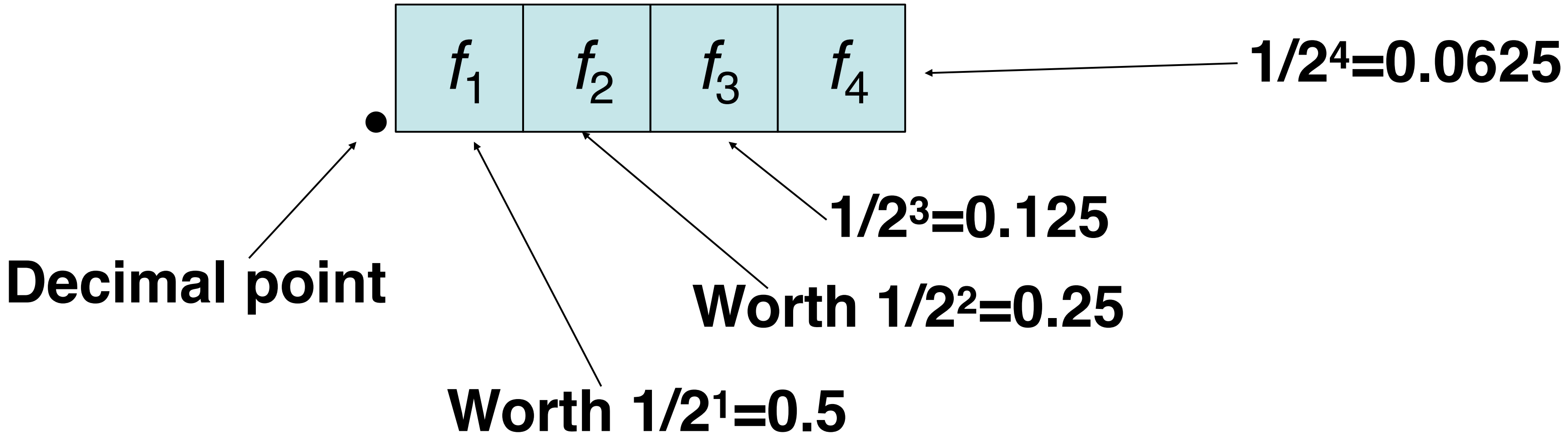


b_5	b_4	b_3	b_2	b_1	b_0	b_{-1}	b_{-2}	00000...

- b_i is still ”worth” 2^i: $b_{-1}=0.5$, $b_{-2}=0.25$, ...

What is fixed?

- The decimal point...
- A fixed-point number has a representation of [*i*.*f*] bits
  - – *i* bits for the integer part (with sign, or without)
    - We assume that two's-complement is used, i.e., integer math can be used
  - – *f* bits for the fractional part
- Look at the fractional bits...



Resolution

- f fractional bits → resolution is 2^{-f}
- Examples:

f	Resolution	Resolution
1	1/2	0.5
2	1/4	0.25
3	1/8	0.125
4	1/16	0.0625
5	1/32	0.03125
6	1/64	0.015625
7	1/128	0.0078125
8	1/256	0.00390625
12	1/4,096	0.000244140625
16	1/65,536	0.0000152587890625
24	1/16,777,216	0.000000059604644775390625
32	1/4,294,967,296	0.00000000023283064365386962890625

How to maintain the best accuracy?

- The number of bits needed for exact accuracy is increased after each mathematical operation (e.g., addition)
  - Overflow
- We focus on
  - Addition/subtraction
  - Multiplication
- Reason: needed for part of assignment 1

Conversion: to fixed and back again

- We have floating point number, *a*, and want fixed-point, [*i.f*]
- To fixed: $\texttt{round}(a \times 2^f)$
- Nice thing: we now have an integer, and so can use integer addition, mult etc (but see next slides on that)
- Rounding is implemented:

$$\texttt{round}(a \times 2^f) = \texttt{int}(\lfloor a \times 2^f + 0.5 \rfloor)$$

- If we have fixed-point number, *b*, we get a float as:

$$\texttt{float}(b \times 2^{-f})$$

- x 2^f and x 2^{-f} are implemented as left and right shifts (fast!)

Very simple example:

- We have float b=0.25
- And want to represent it in fixed-point with 3 fractional bits, i.e., f=3
- round(0.25*2^3)=2
- Thus 2 is the fixed-point representation of 0.25 with three fractional bits
- Can look at the 8 bits of the integer:
  - 0000.010 (= 2 if you disregard the decimal point)

© 2009 Tomas Akenine-Möller

Addition precision

$$[i.f] \pm [i.f] \ = [i + 1.f]$$

- Why? Imagine the worse case:
  - –Both numbers hold their maximum number:
    - Eg 111.11_b +111.11_b = 1111.10_b
    - Result grows by one bit in integer part!
- Number of bits becomes:

$$[i_1.f_1] \pm [i_2.f_2] = [\max(i_1, i_2) + 1.\max(f_1, f_2)]$$

Note ”+1”

Multiplication precision

- More complex. Can be seen as many adds!
  - So intuitively, should need more bits to store

$$[i.f] \times [i.f] = [2i.2f]$$

- Note, that if you want to maintain exact accuracy, we need to move the ”fixed-point”
  - Need twice as many fractional bits!
- In general:

$$[i_1.f_1] \times [i_2.f_2] = [i_1 + i_2.f_1 + f_2]$$

- See appendix A for an explanation
  - Basically, a mult is a series of additions of shifted numbers

Fixed-point in practice

- In C++ code, you deal with these as **`int`**’s
  - 32 bit signed numbers (but you need not use all of the bits)
- However, you need to prepare the calculations so that bits are not lost
- For edge functions it is of utmost importance to maintain exact values
  - (after you have rounded off floating-point screen space coordinates to sub-pixel fixed-point coords)
- Example: **`a`** and **`b`** are [8.2]. If you write:
  - **`c=a*b;        // then c is [16.4]`**
  - **`d=c>>2;       // d is now 16.2 format`**
  - **`              // (but we’ve lost 2 LSB`**
  - **`              // fractional bits)`**

Fixed-point example

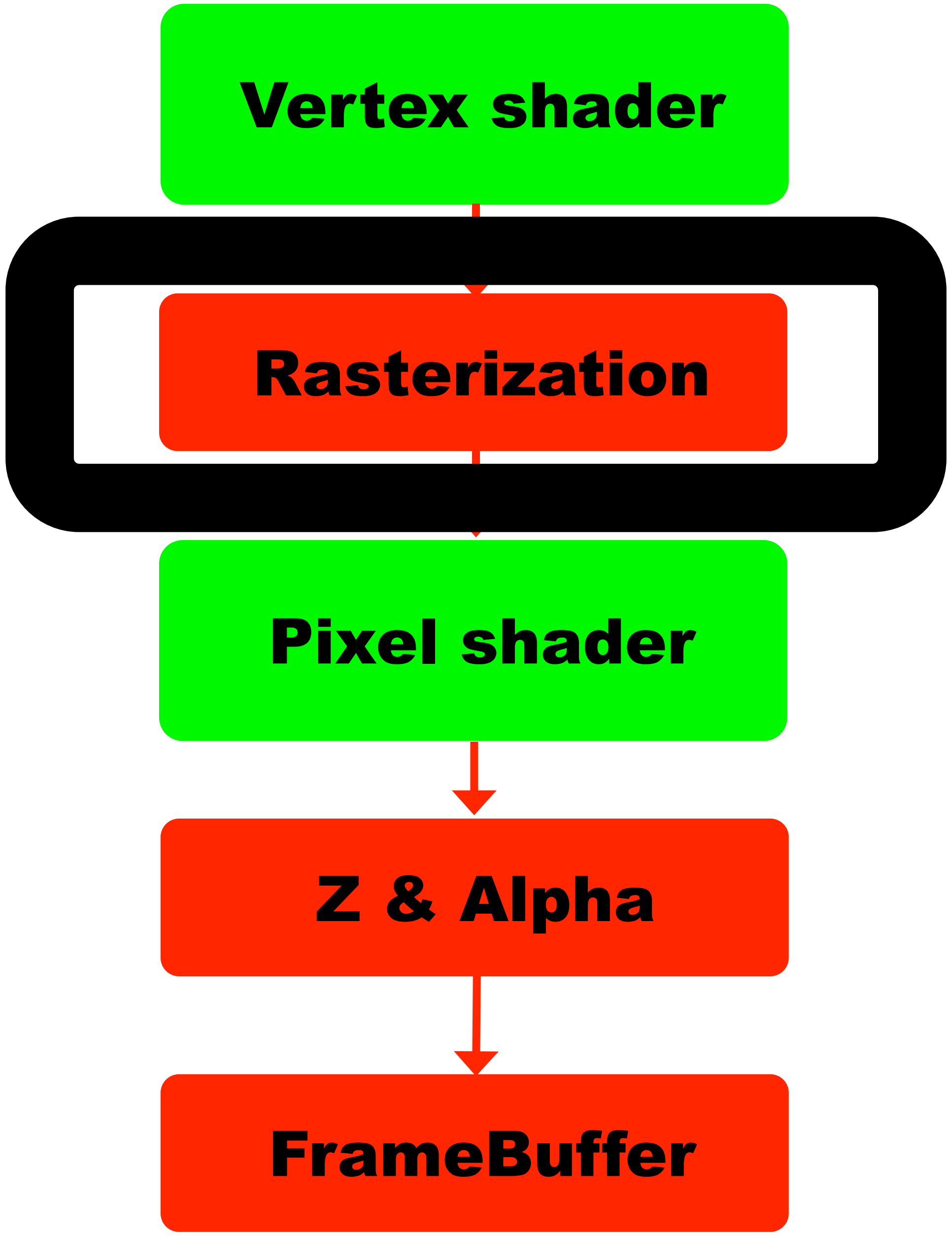
- `float a=2.75f;`
- `int ai=int(a*(1<<2)+0.5); // [2.2]`
- `// should use floatToFixed()`

- `float b=2.5f;`
- `int bi=int(b*(1<<1)+0.5); //[2.1]`
- `// how to compute ai+bi?`
- `int ci=ai+(bi<<1);   // [3.2] bits`

End of fixed-point...

- In software frame work, a function **`int floatToFixed(fracBits, float_number)`** is used.
- When you do a matrix/vector multiply
  - You often do [16.16]*[16.16] ~=[32.16], or worse
- Remember
  - Full accuracy needed for edge-functions
- Read appendix A and chapter on Edge Funcs again
  - Available on course website

Last week's stage of the Graphics Pipeline



Vertex shader

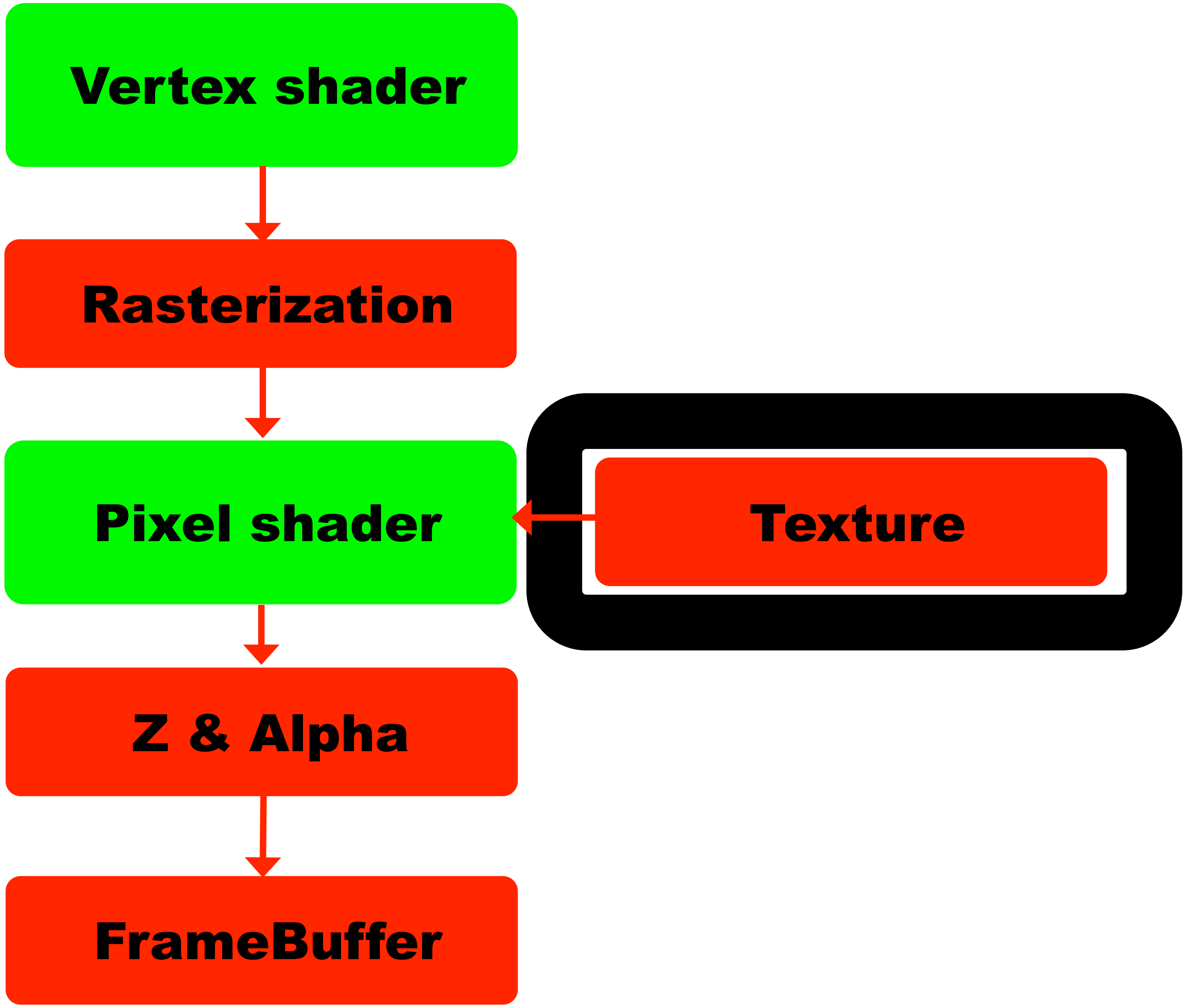
Rasterization

Pixel shader

Z & Alpha

FrameBuffer

Today's stage of the Graphics Pipeline



Vertex shader

Rasterization

Pixel shader

Texture

Z & Alpha

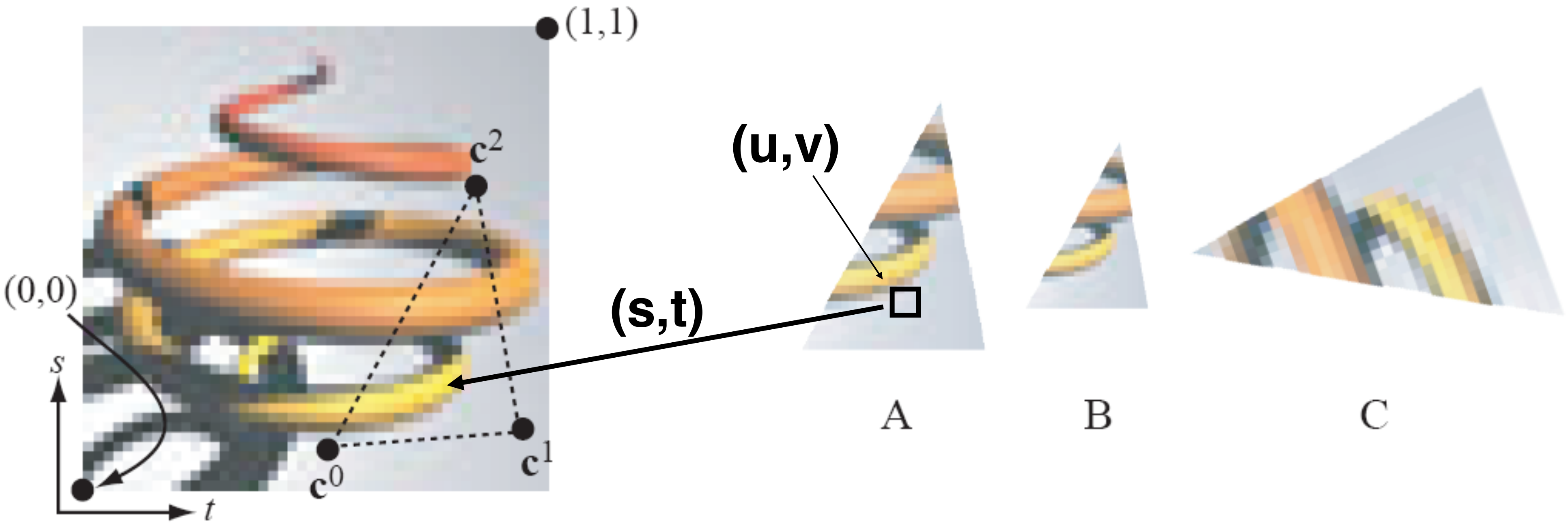
FrameBuffer

Texturing – the tiny details

Image from ”Ipics”-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*)

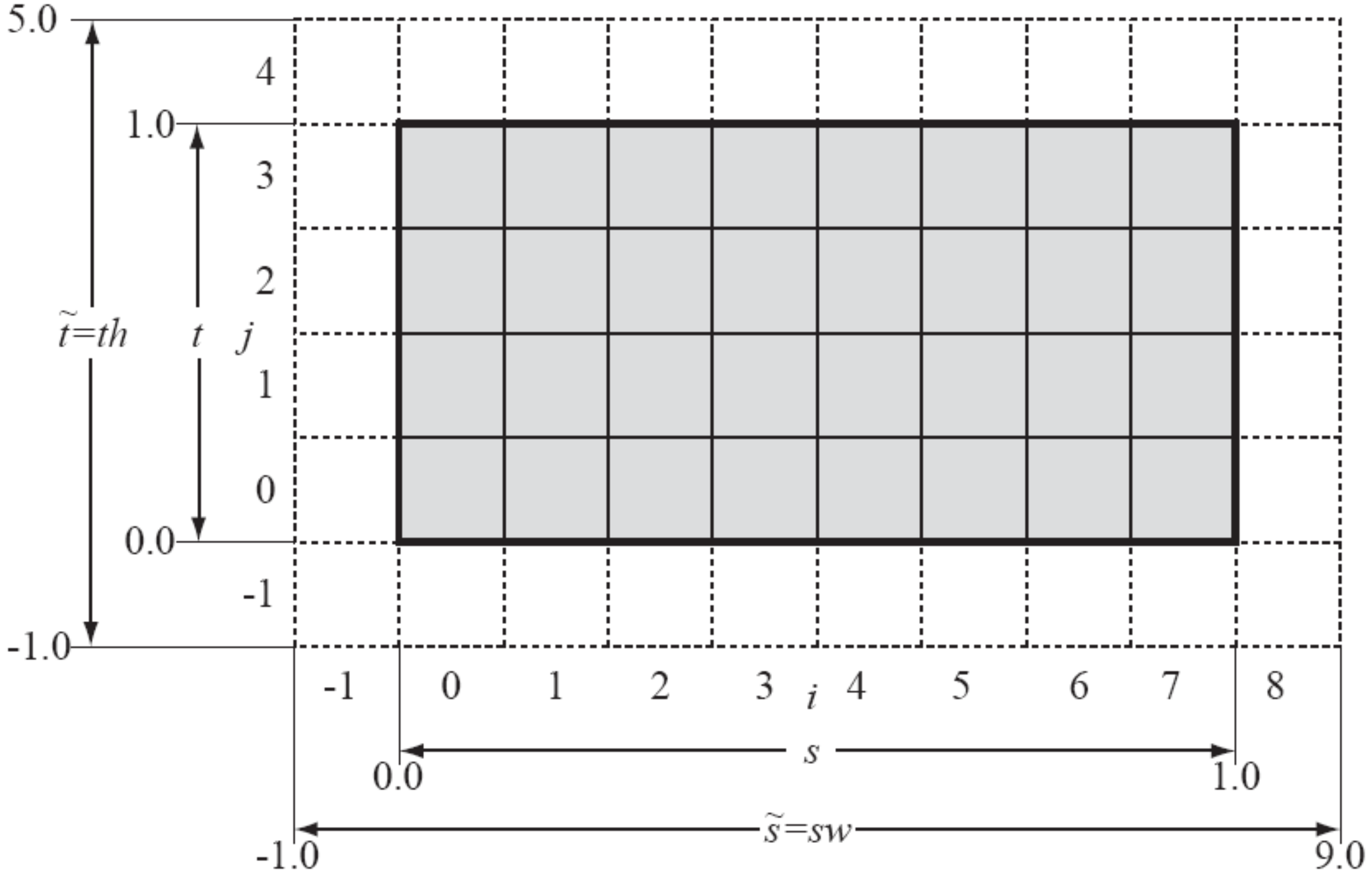


- Texture resolution, often $2^a \times 2^b$ texels
- The $\mathbf{c}^k$ are *texture coordinates*, and belong to a triangle's vertices
- When rasterizing a triangle, we get (u,v) interpolation parameters for each pixel (x,y):
  - Thus the texture coords at (x,y) are:

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

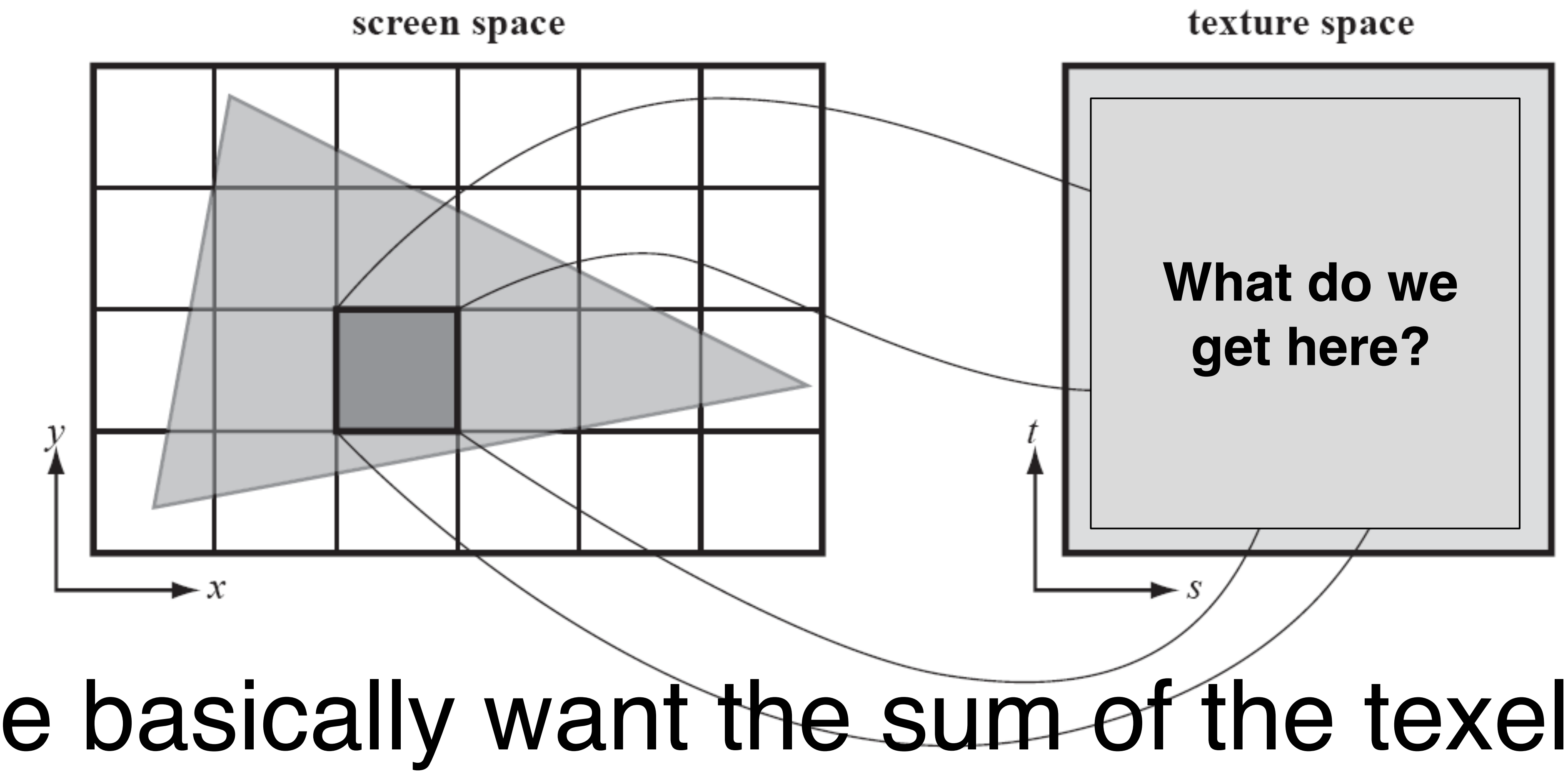
© 2009 Tomas Akenine-Möller

A texture image + coord systems



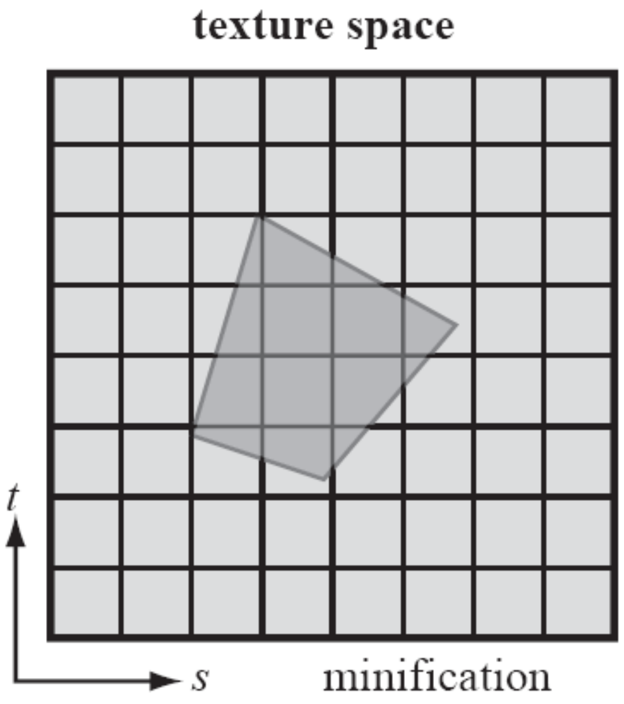
- An *w*x*h*=8x4 texture.
  - (*s*,*t*) are independent of texture resolution
  - (*sw*,*th*) depend on the resolution, and are used to access texels…
- Each pixel in a Texture is called a “**Texel**”

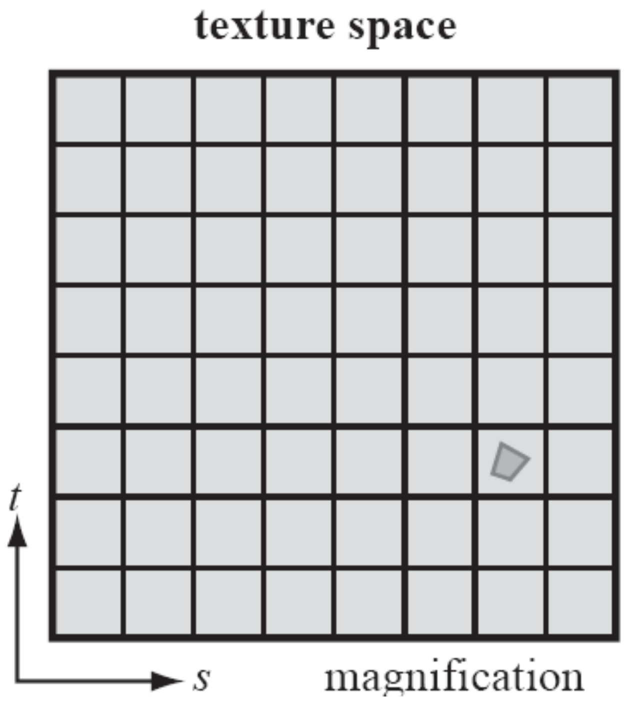
Texture filtering



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

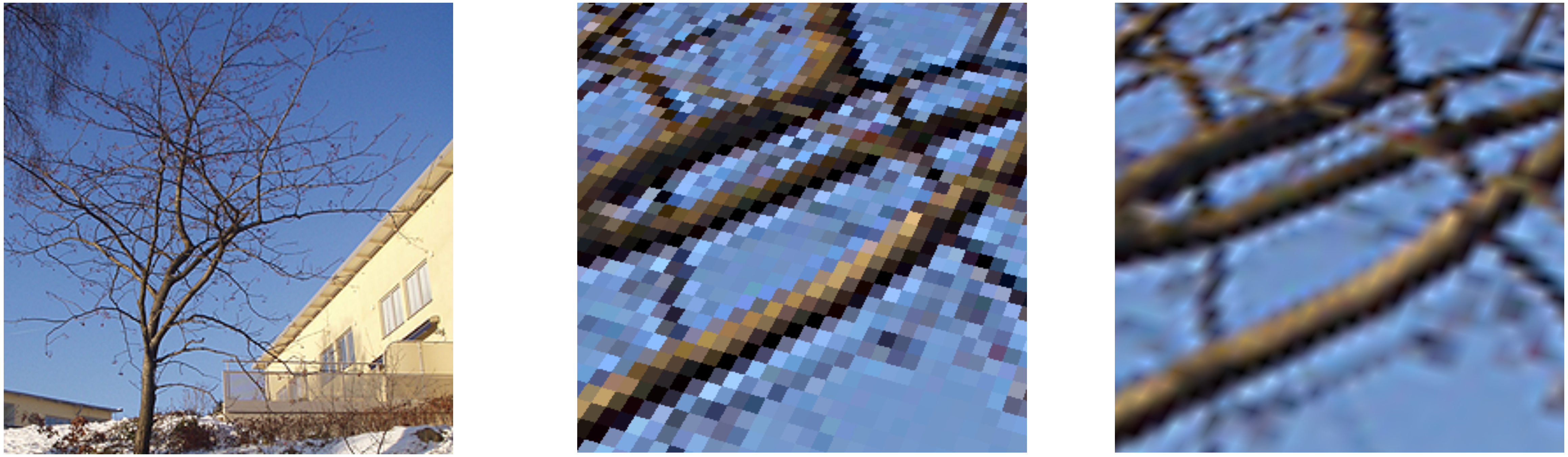
MINIFICATION





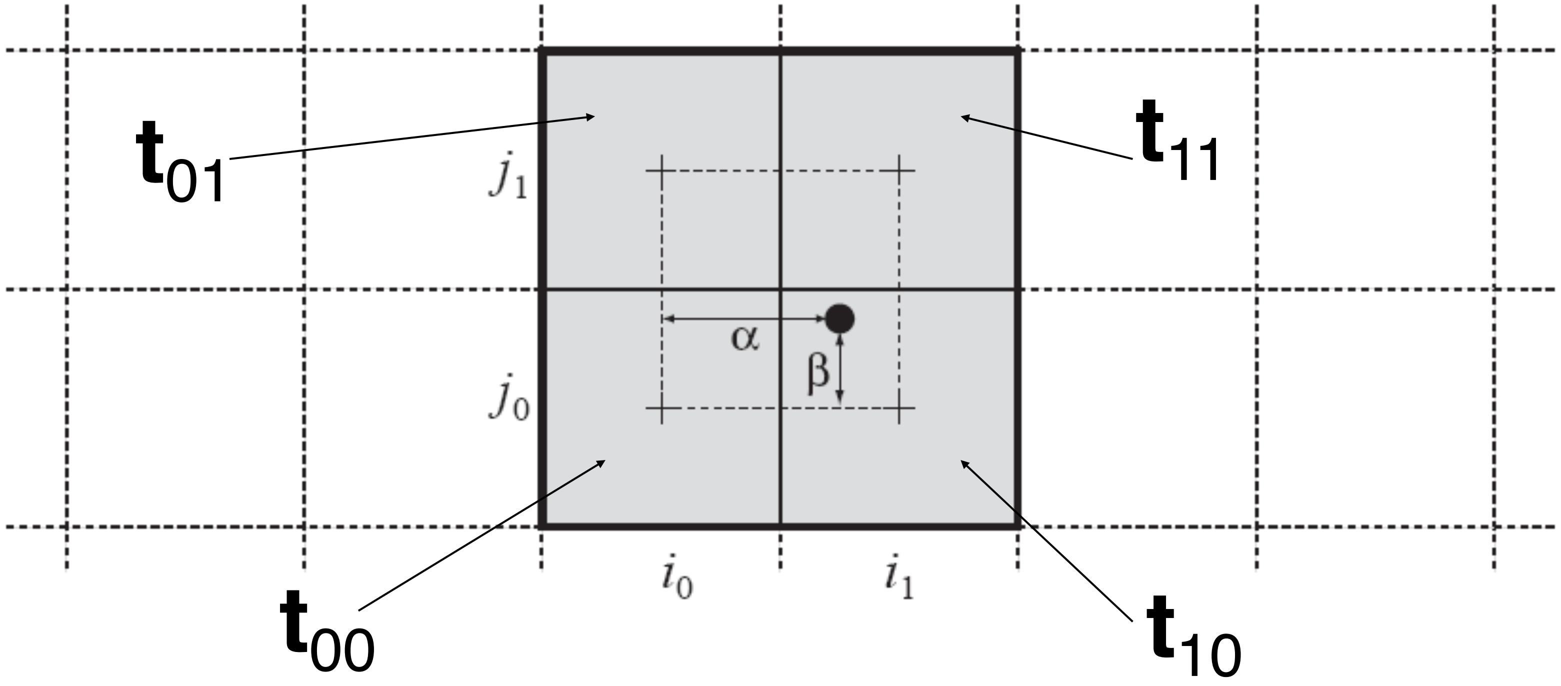
MAGNIFICATION

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:

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

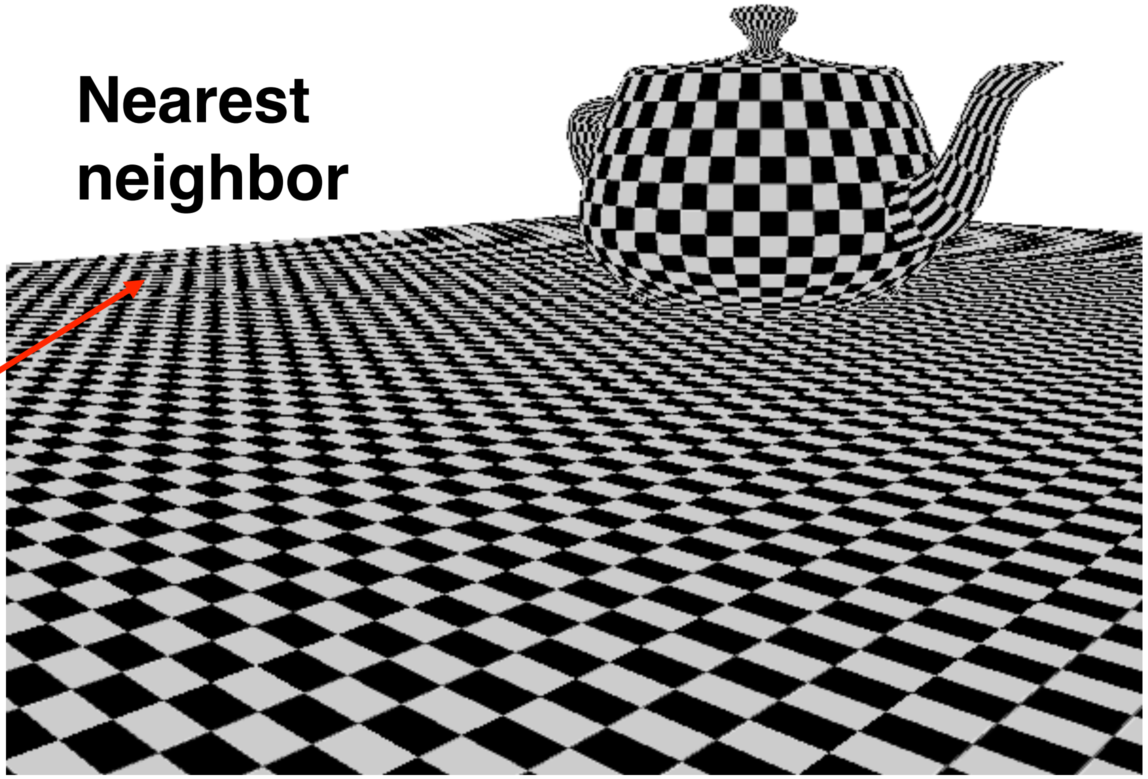
- Followed by linear filtering in y:

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

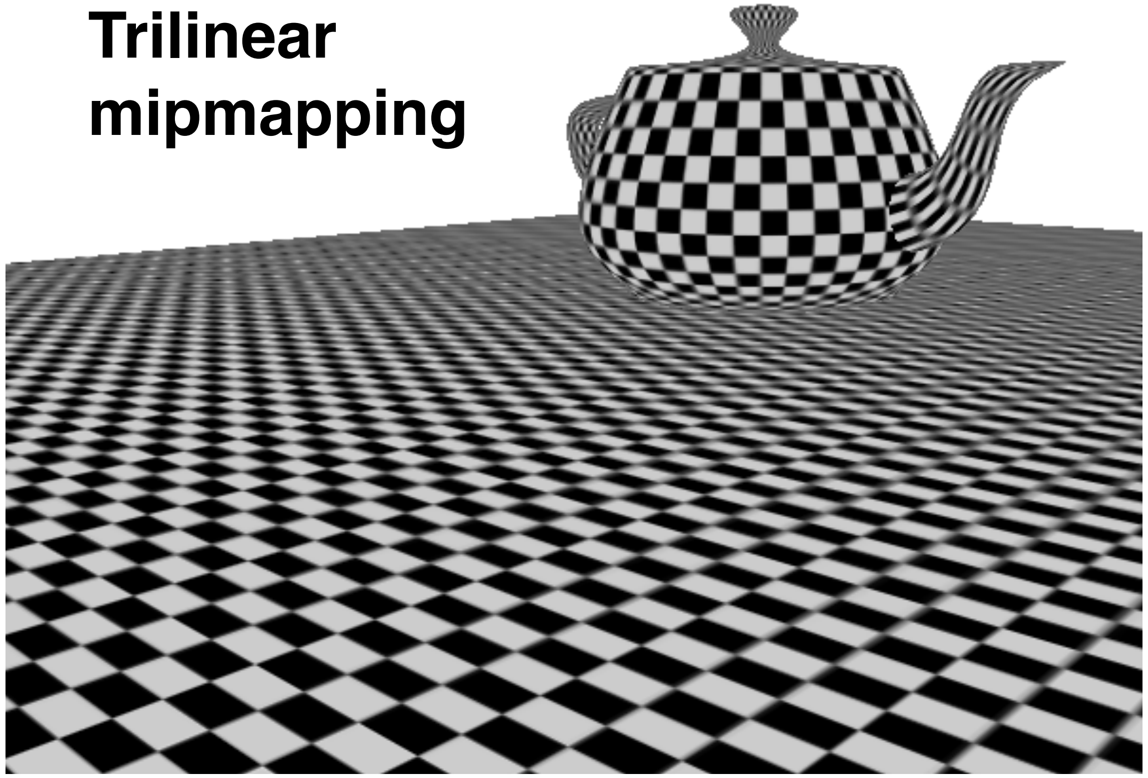
Texture minification

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

Nearest neighbor

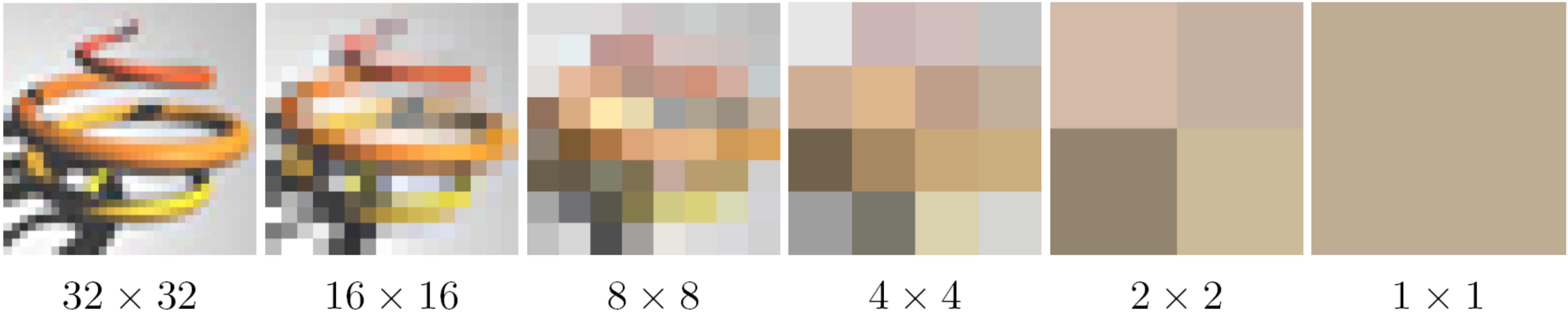


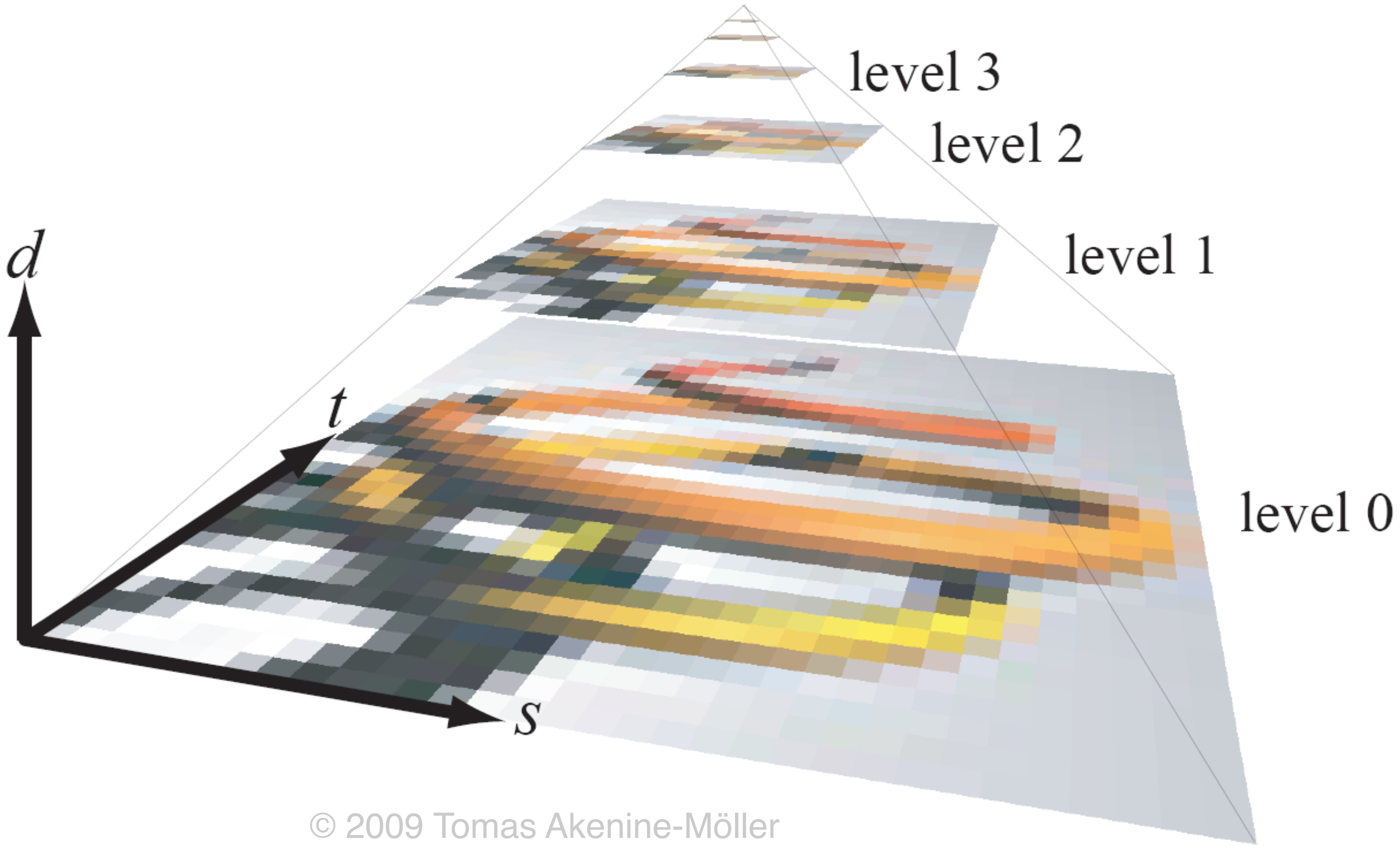
Trilinear mipmapping



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

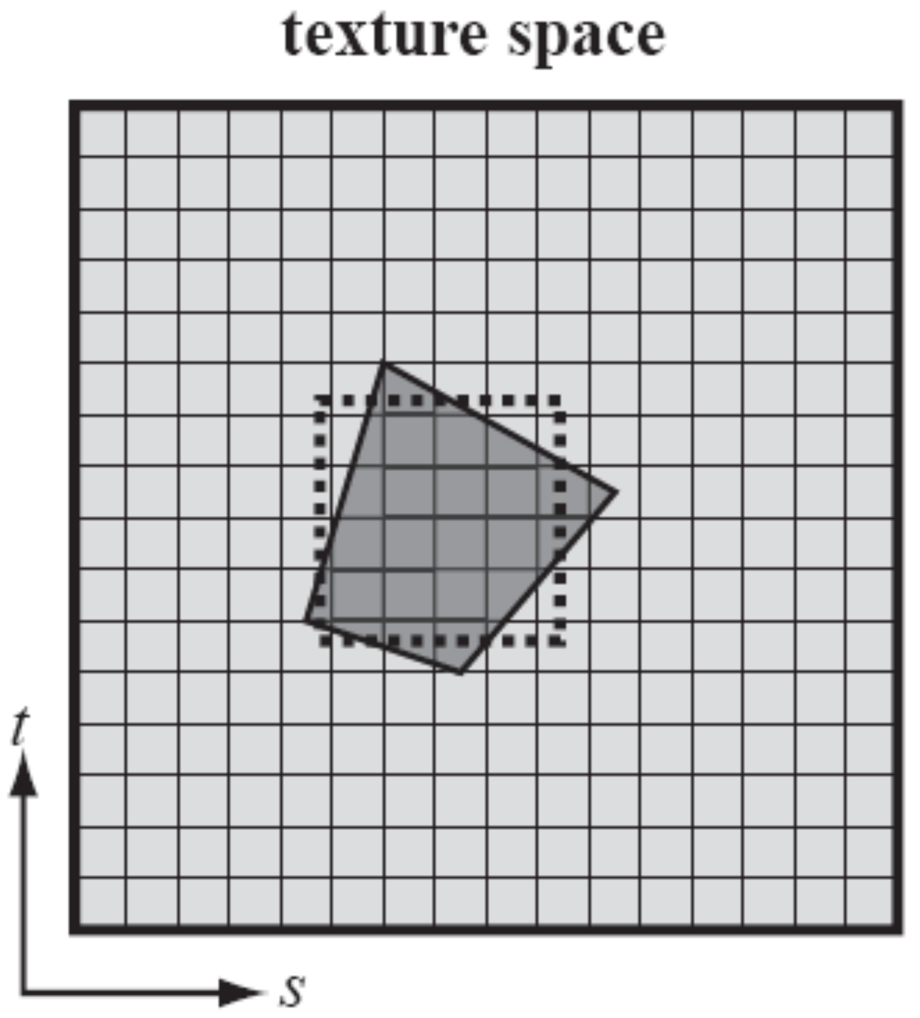
Texture minification: mipmapping





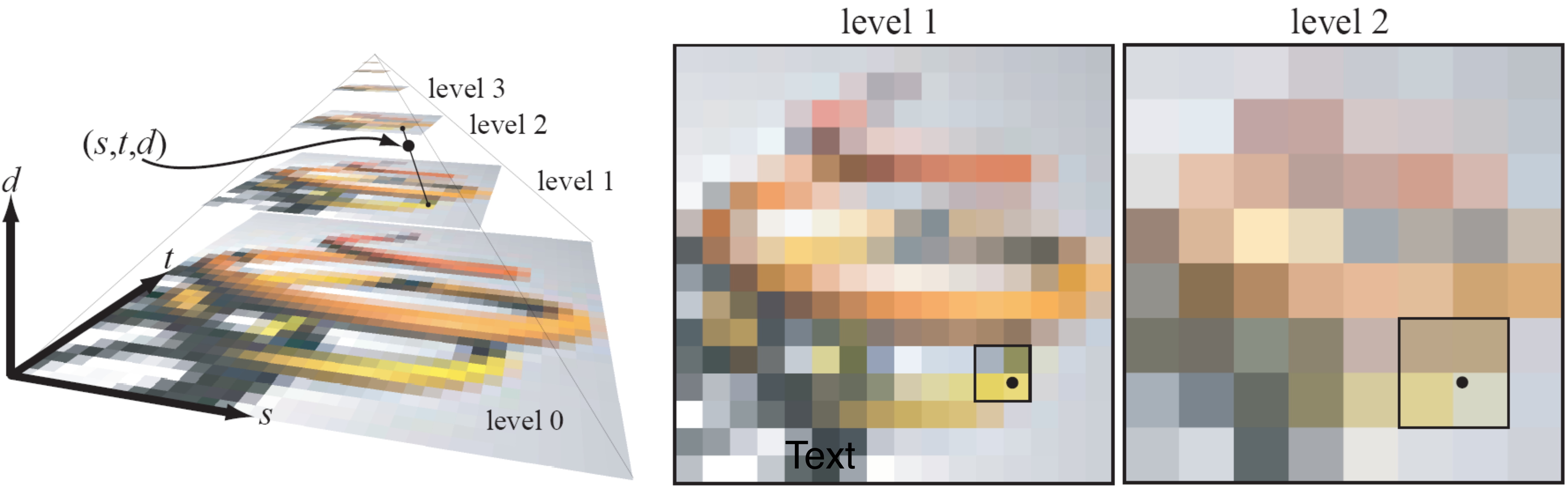
An image pyramid of low-pass filtered images

Trilinear Mipmapping (1)



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

Trilinear mipmapping (2)



- Compute *d* (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

demo1.exe

Texture
Caching

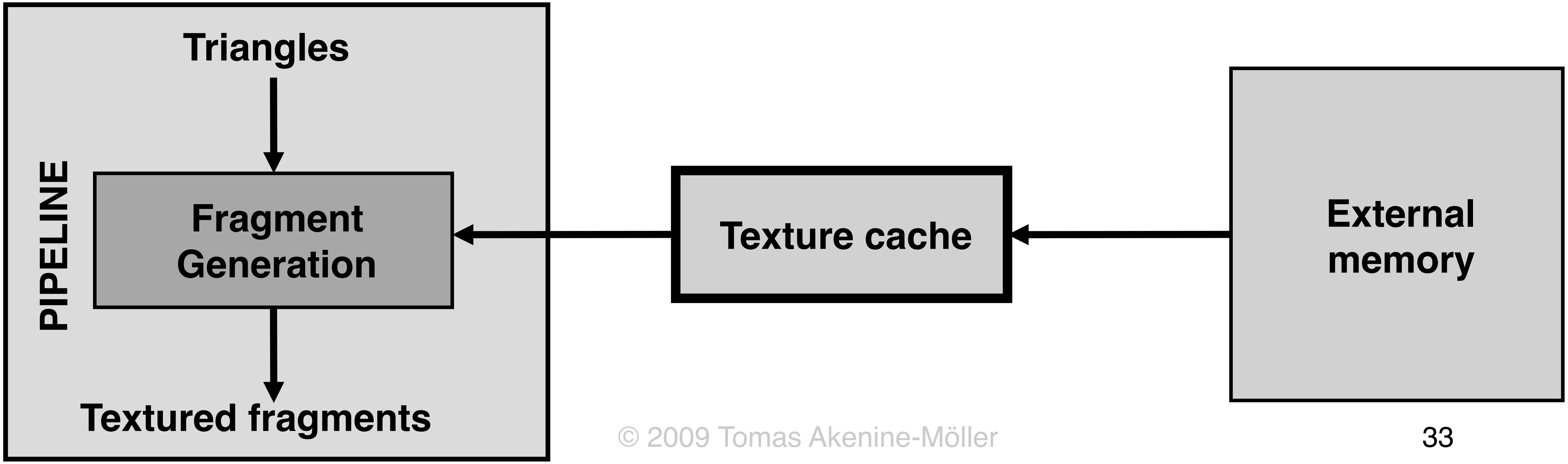
Texture caching

- Without a cache, we can get ridiculously expensive texturing...
- Basic idea is: just use a cache for recently accessed texels
  - Since we access coherently, hit rate should be quite high!
  - In hardware, a cache can be:
    - A small SRAM memory, or
    - A set of flipflops
    - We assume that an access in the cache is for ”free”

- In the assignment, texture filtering (eg mipmapping) is done for you.
  - You should experiment with caching parameters!

Assumptions: memory architecture

- Accesses to external memory are expensive
  - –Both in time and from energy perspective
  - –Bursting (i.e., send a sequence of continuous words) is often (much) cheaper
    - E.g., fetching 8x 32-bit words (32 bytes) in a sequence is much faster than fetching 8x 32-bit words that are in random places...



© 2009 Tomas Akenine-Möller

Texture cache readings

- A nice introduction :
  - My “Texture Caches” paper from IEEE Micro 2012
- Also :
  - ”The Design and Analysis of a Cache Architecture for Texture Mapping”, by Hakura and Gupta, in ISCA 97.
  - ”Prefetching in a Texture Cache Architecture”, by Igehy et al, in Graphics Hardware 1998.
    - Note that these are old papers, and cache sizes etc don’t apply to modern systems...
    - The general results still apply though

- GPU Example: NEON architecture (1998)
  - Built by Digital Equipment Corporation (bought by Compaq (bought by HP))
  - Has 256 bytes of cache, fully associative
  - Split into 8 different small caches
    - So 8 texels can be fetched every clock cycle
  - Cache line size is 32 bits
    - This is very small. The optimal size depends on what type of external memory you have

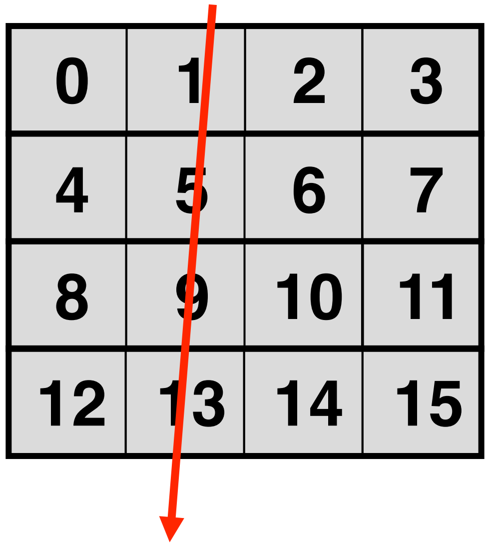
- More about GPU memory architecture in a later lecture

How to get good efficiency

- Three important things [Hakura & Gupta]:
  - How texels in texture are ordered in memory
  - Rasterization algorithm
  - Cache parameters
    - Associativity
      - Number of cache lines = sets X ways
      - n - way associate cache : means n blocks(lines) in each set
    - Cache line size
    - Cache size

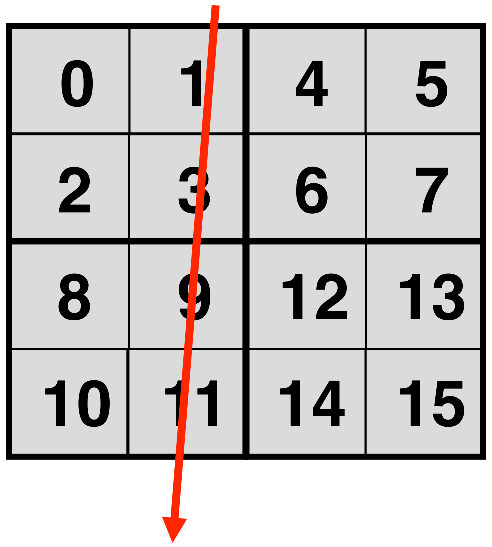
Representation of textures in mem

- 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...

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

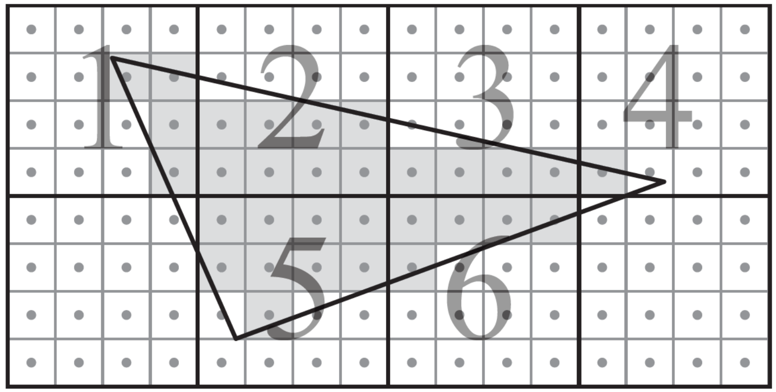
- In general, we have an *n*x*n* block...
  - *n* is power of 2
- Mipmap levels can thrash at exactly the same location in a direct mapped cache
- Solution:
  - Use a fully associative cache
  - Hakura & Gupta shows that a 2-way associative cache gives similar results
  - Or simpler, ”bake” the mipmap level into the computation of the ”cache key” (tag)

Texture cache recommendations

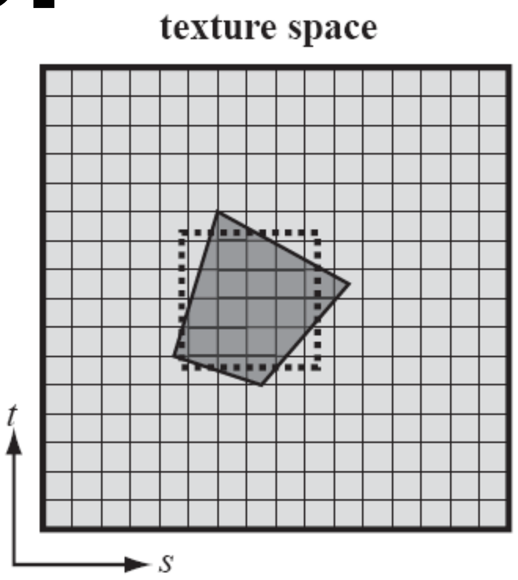
- Tile (block) size in texture should be equal to cache line size
- Can even extend to 6D addressing
  - Another level, where each block is the size of the entire cache...
    - Further minimizes conflict misses
  - Also, Igehy et al use two separate direct-mapped caches:
    - One for odd mipmap levels, and one for even
    - Is enough to get good results
    - Again, one direct-mapped cache would work if the cache key (tag) take mipmap level into account (but having two caches gives more bandwith from the caches)

Traversal algorithm

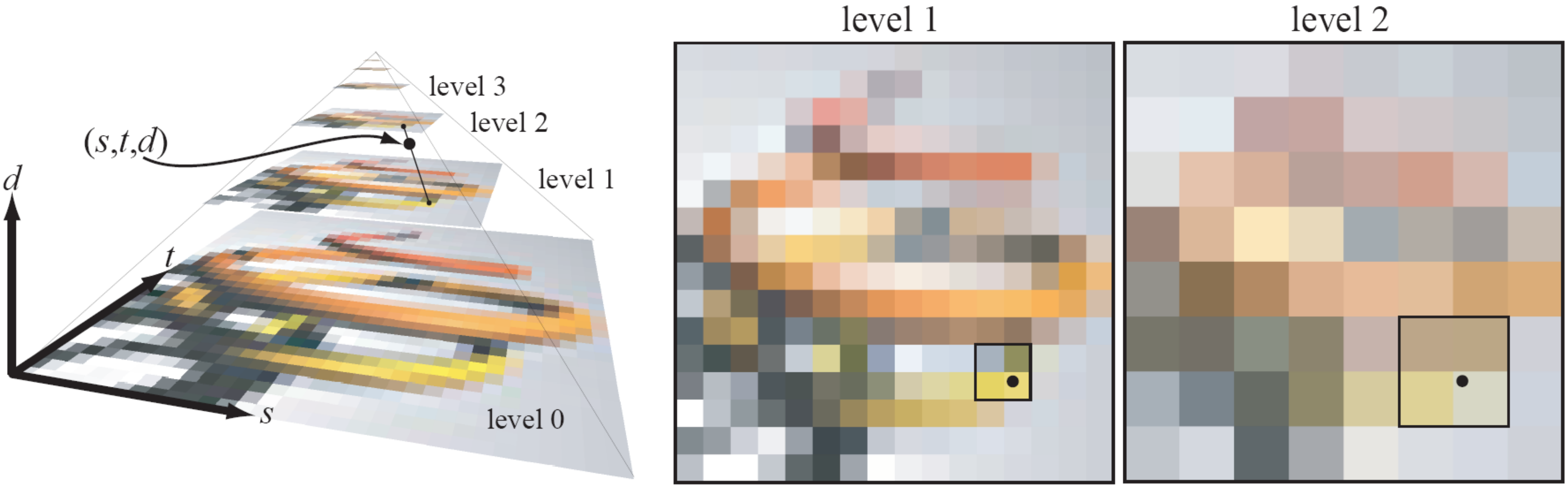
- Traversal algorithm affects the order in which texels are accessed →
  - Also influences texture caching...
- With scanline-based traversal, we do not get any positive effects for pixels below current scanline
  - This is assuming a small cache
  - Positive effects should be possible, due to bilinear filtering (used in mipmapping and magnification)
- Tiled traversal performs better!
  - Especially for large triangles



Why is mipmapping good for texture caching?



- We choose mipmap levels to access where footprint becomes ~1 texel



- Therefore, traversal moves slowly in texture space → many cache hits!
- Better than nearest neighbor (minification)

Back to the assignment... The coding framework (1)

- Implements a subset of OpenGL
  - (mostly focused on the rasterizer)
- Designed so
  - that is, it is built around units that exist in real hardware
- Programmability
  - We have fragment shaders as well
  - Though, focus is not on using them right now...

The coding framework (2)

- Uses Microsoft Visual Studio 2008
  - But upgrades to work with 2010/12/13/15
- Nice feature for this assignment:
  - Press the R key, and you can toggle rasterizer
  - You can switch from
    - our software rasterizer
    - to the OpenGL hardware rasterizer

Actual assignment (1)

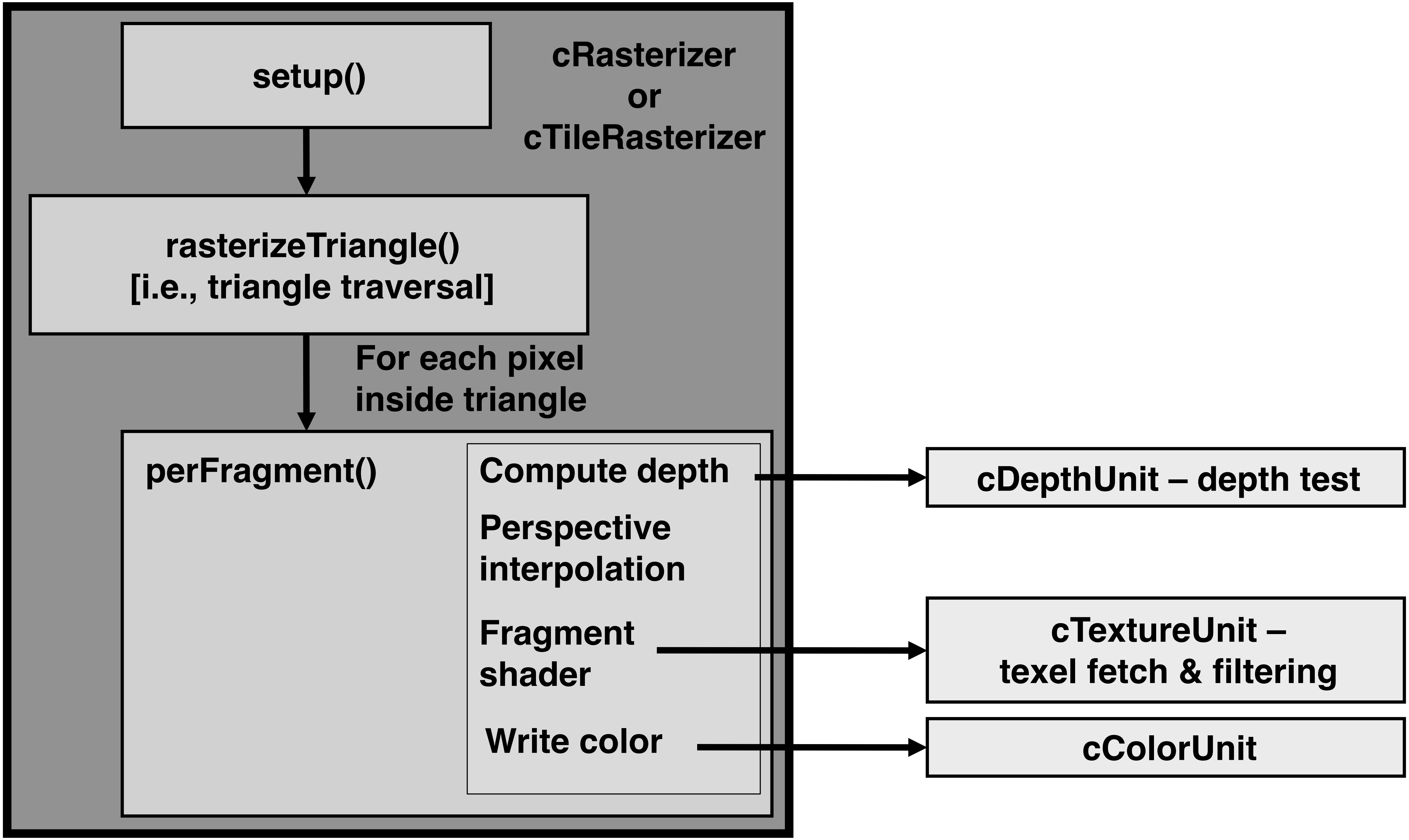
- Two tasks..
- **Task 1**:
  - –Switch between the software rasterizer and hardware OpenGL rasterizer (press ’R’)
  - –Use this to detect the ”artifacts”
    - Three artifacts: need to be corrected so that results are ”very near” identical to hardware OpenGL
    - How could I know how to correct the artifacts?
    - Read the literature that we recommend!
  - –Everything is very localized in the source:
    - Change in **`cRasterizer.*`** + **`cEdgeFunc.*`**

Actual assignment (2)

- **Task 1**: Fix pixel errors.

- **Task 2**:
  - Time to reduce texture bandwidth
  - In **`glstate.cpp`**, add a texture cache...
  - Should be able to reduce texture bandwidth to at most 10-15%...
  - You need to experiment quite a bit to get this kind of performance...

More about the software framework



setup()

cRasterizer
or
cTileRasterizer

rasterizeTriangle()
[i.e., triangle traversal]

For each pixel
inside triangle

perFragment()

Compute depth

Perspective
interpolation

Fragment
shader

Write color

cDepthUnit – depth test

cTextureUnit –
texel fetch & filtering

cColorUnit

Next

- Don’t forget to read the literature!
  - Text has full background to the slides
  - Very valuable for assignments too
- Labs
  - Find a partner (ask on the forum)
  - Sign up
- Check the web page for info http://cs.lth.se/edan35
- Ask questions on the forum
- Next Lecture :
  - Shader programming