Real-Time 3D Graphics Architecture

William R. Mark – University of Texas at Austin

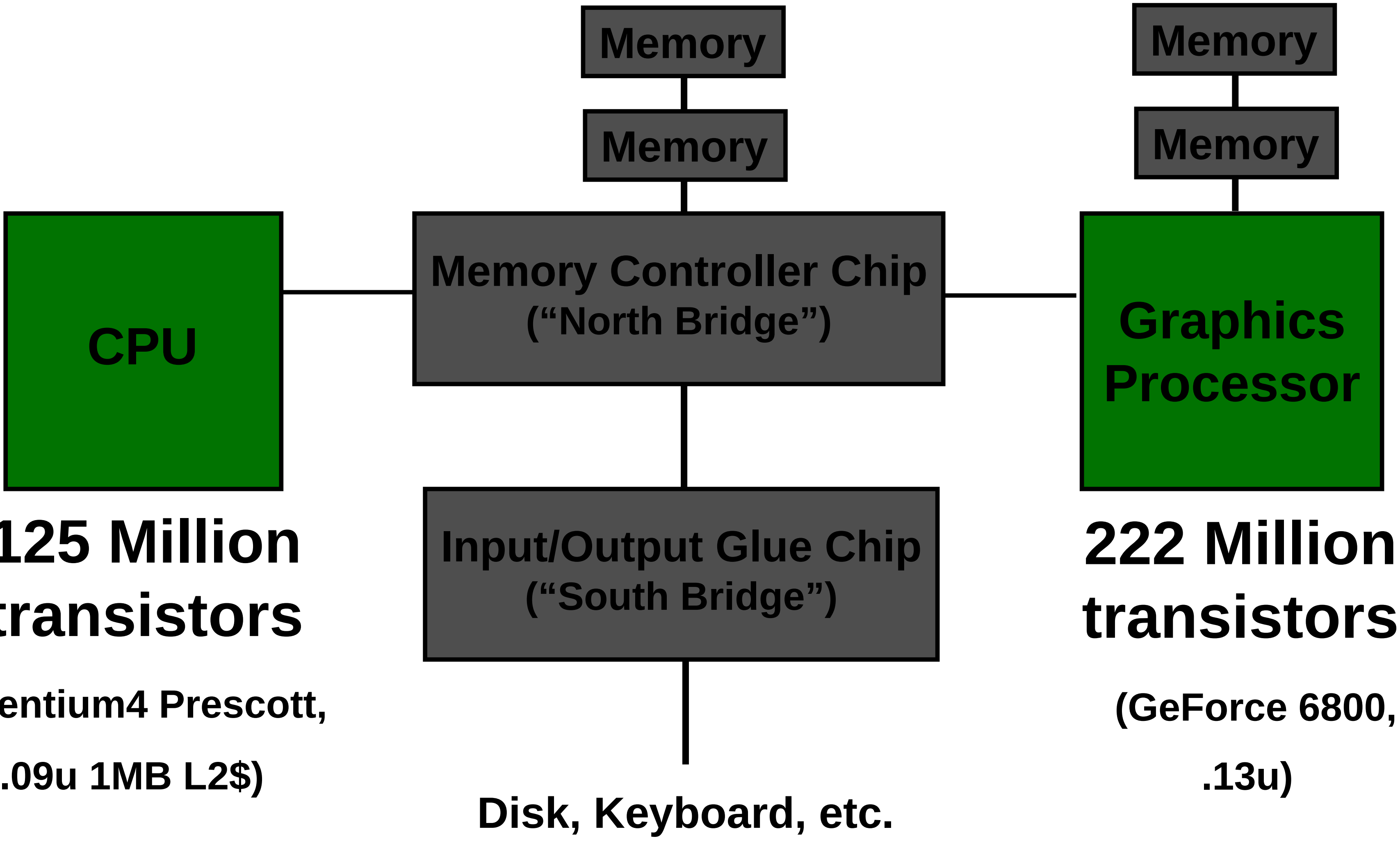
Henry Moreton – NVIDIA

Micro-37 Conference Tutorial
Dec. 4, 2004

Speaker backgrounds

- William Mark – Univ. of Texas at Austin
  - Research area: Real-time 3D graphics systems
  - Formerly at NVIDIA; led design of Cg language
- Henry Moreton - NVIDIA
  - Senior architect in NVIDIA architecture group
  - Formerly at SGI

Dedicated graphics chip in modern PCs



125 Million transistors

(Pentium4 Prescott, .09u 1MB L2$)

222 Million transistors

(GeForce 6800, .13u)

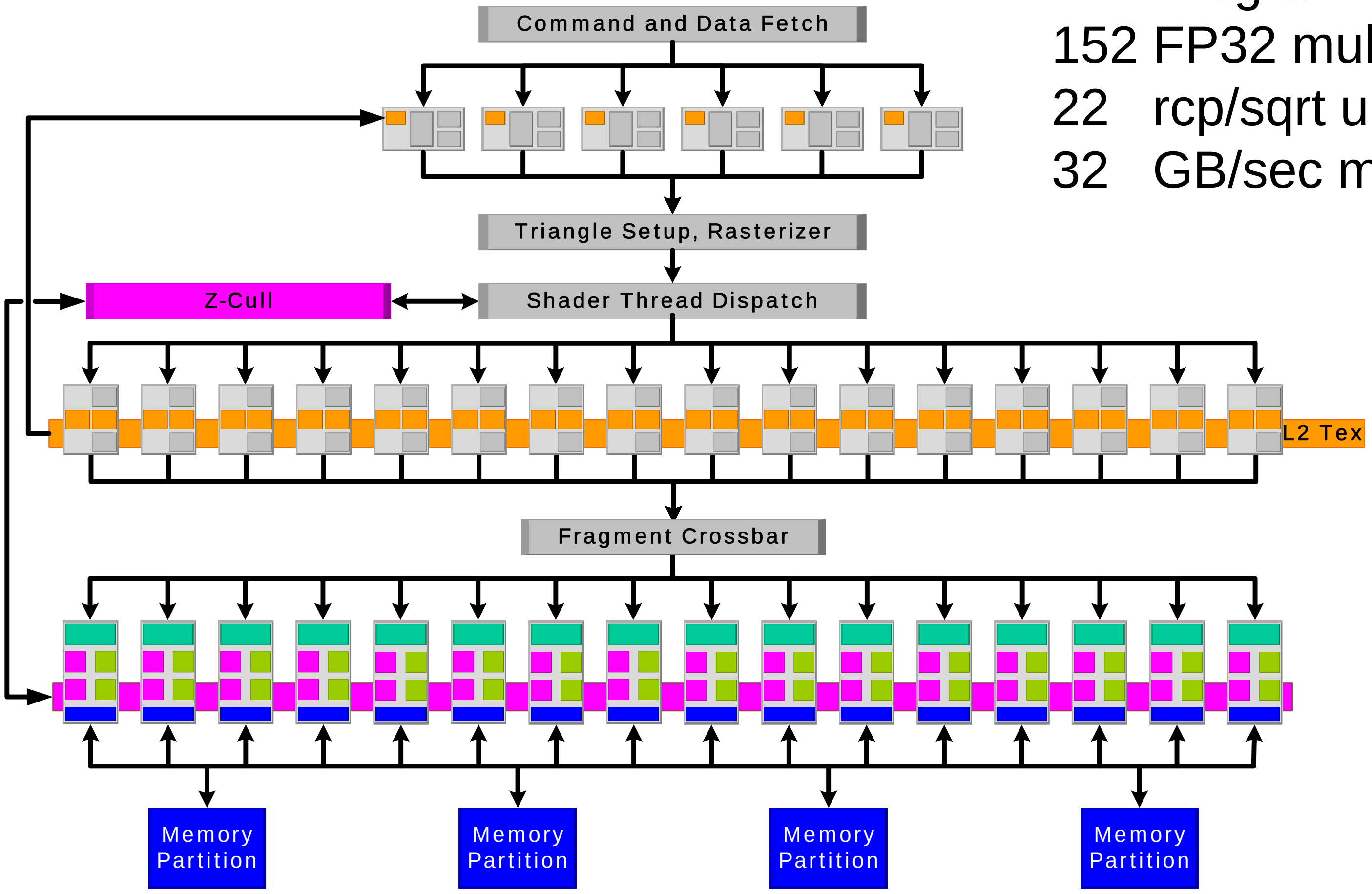
CPU vs. Graphics Peak Performance

	Pentium 4 1.06 GHz FSB	**ATI Radeon X800**
Clock rate	3.8 GHz	0.5 GHz
Peak GFLOPS	15.2	63.7 (fragment unit)
Memory BW	8.4 GB/sec	32 GB/sec

GFLOPS source: Fatahalian et al, GH2004

Highly parallel, single chip architecture

GeForce 6800



22	Programmable Cores
152	FP32 mult/add units
22	rcp/sqrt units
32	GB/sec memory BW

Task is computationally intensive

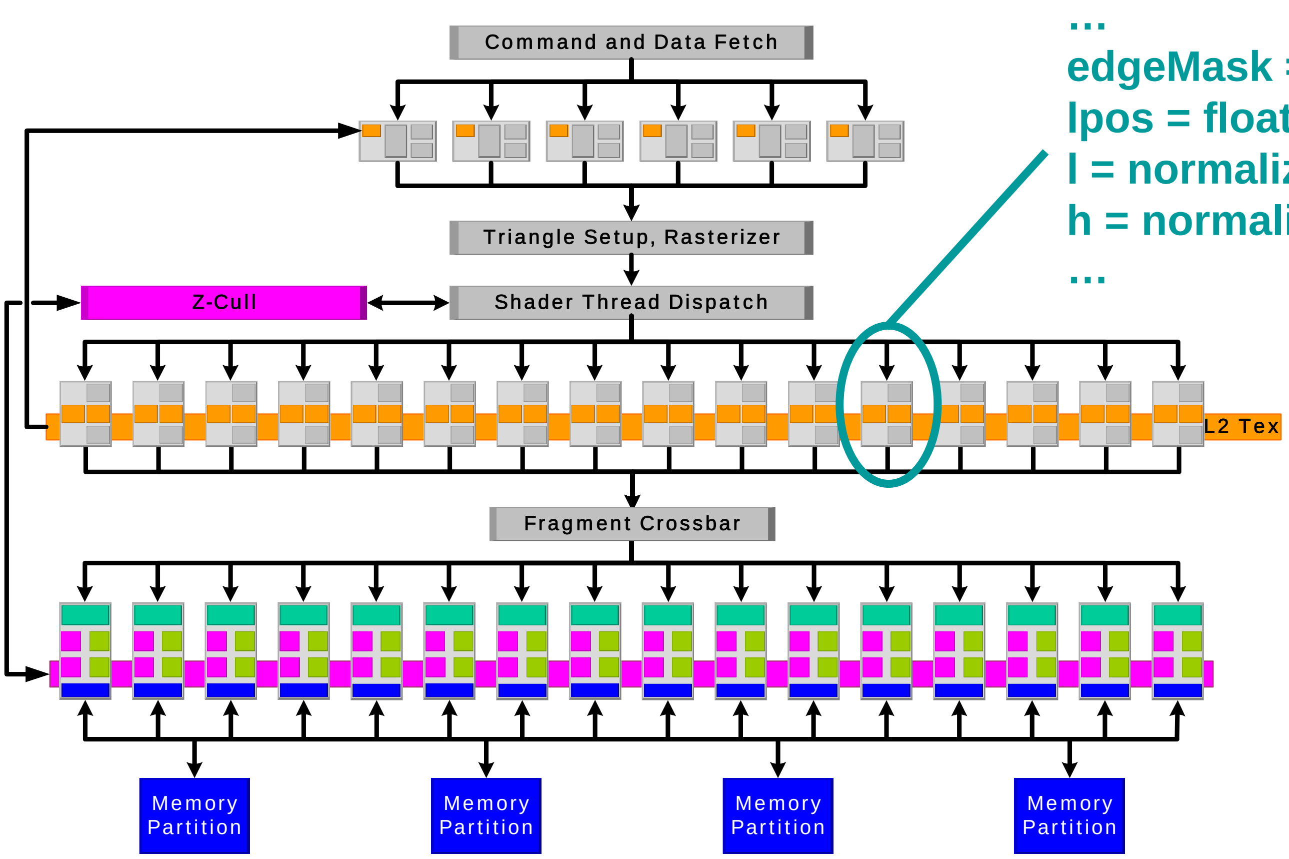
1 million pixels
@ 60 frames/sec:

**60 million
pixels/sec**.

Lots of work
for each pixel.

Half Life 2
Valve Software
Nov. 2004

HW is programmable (for some units)



```
…
edgeMask = (dot(e, n) >  0.4) ? 1 : 0;
lpos = float3(3,3,3);
l = normalize(lpos - In.TEX7.xyz);
h = normalize(l + e);
…
```

“Mainstream” architects can learn from GPUs

- Parallelism is becoming more important
  - Peak performance vs. FLOPS/$$$ or FLOPS/Watt
- GPUs are first highly-parallel processors in PCs
  - And now they’re programmable
- Graphics is major driver of PC performance
  - Large market
  - Performance is not yet “good enough”
  - Innovative and talented software developers
    - Willing to experiment
    - Willing to endure (some) pain to get performance

Tutorial outline

- Fundamentals of 3D graphics [Bill]
- Z-buffer graphics pipeline [Bill]
- NV40 case study [Henry]
- More details of modern architectures [Bill]
- Discussion and future trends [Bill]

Please interrupt at any time to ask questions.

Fundamentals of 3D Graphics

Motivation for learning graphics fundamentals

- Q: I'm an architect. I do hardware, not algorithms. Can't we just skip ahead to the case study?

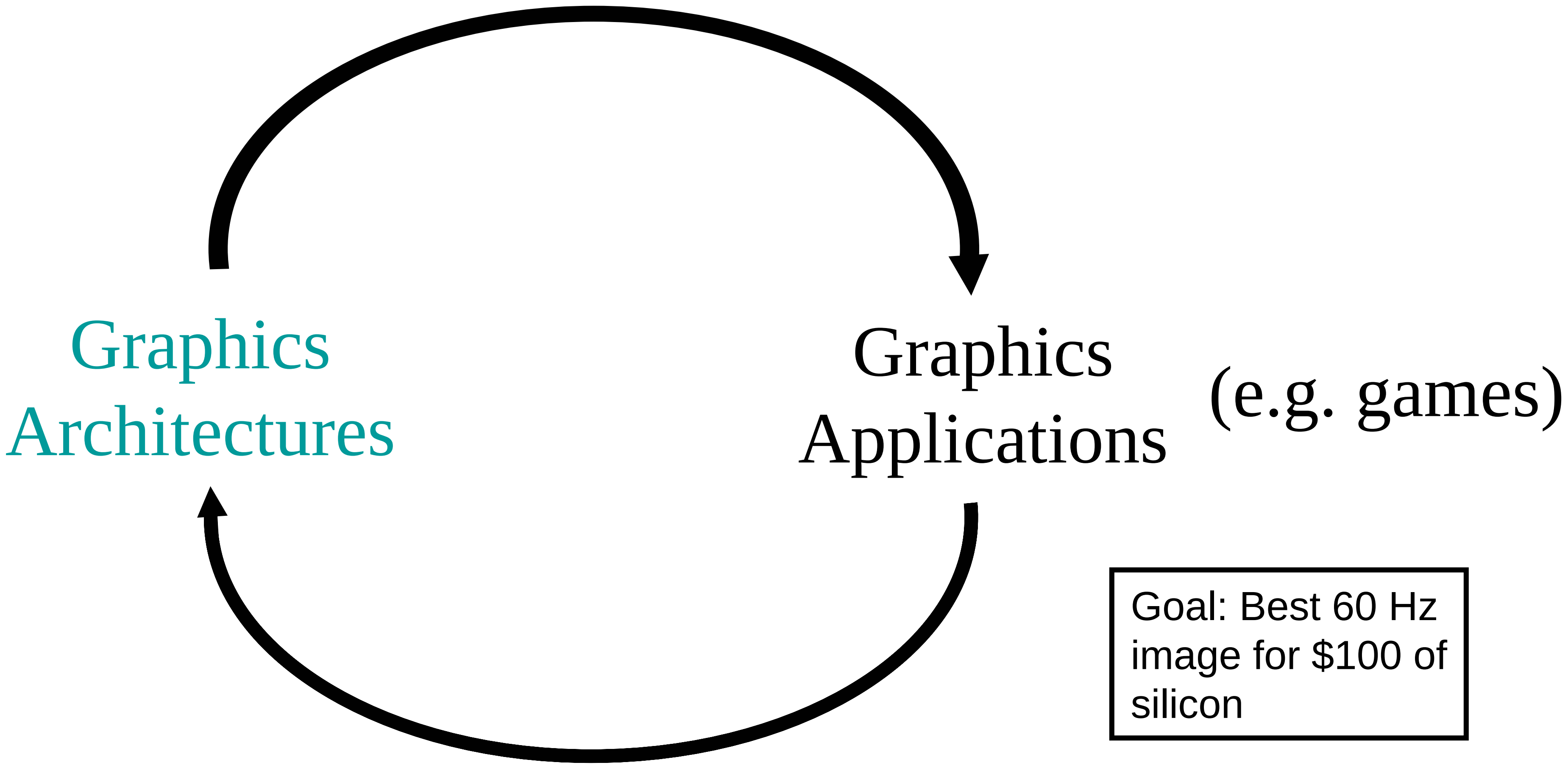
  A: Not really. You can't understand 3D graphics architectures without understanding 3D graphics algorithms.

- Q: Could I design my new Acme FlexiGPU architecture by optimizing for current graphics applications/traces/benchmarks?

  A: No, not if you want your architecture to be relevant when it's done.

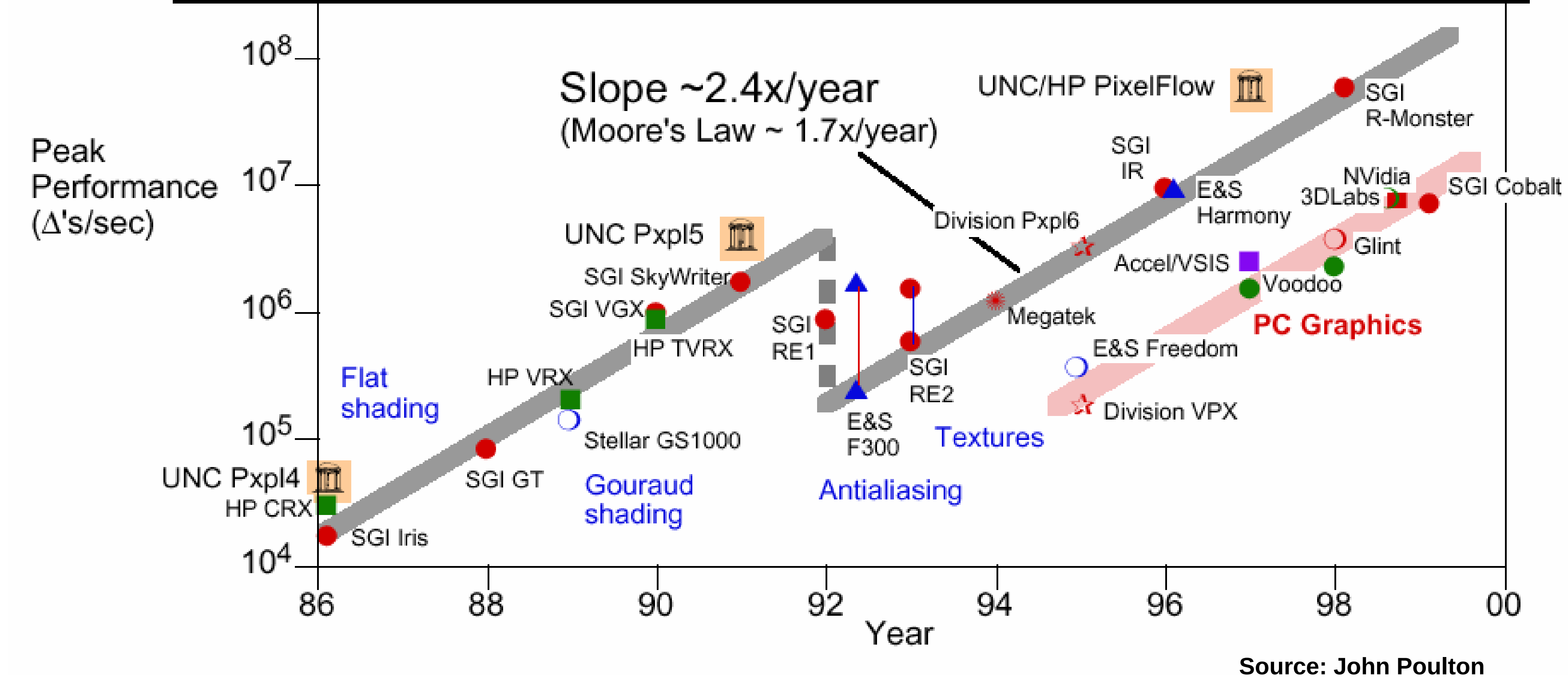
Graphics applications and HW co-evolve

Architecture strongly influences applications



Goal: Best 60 Hz image for $100 of silicon

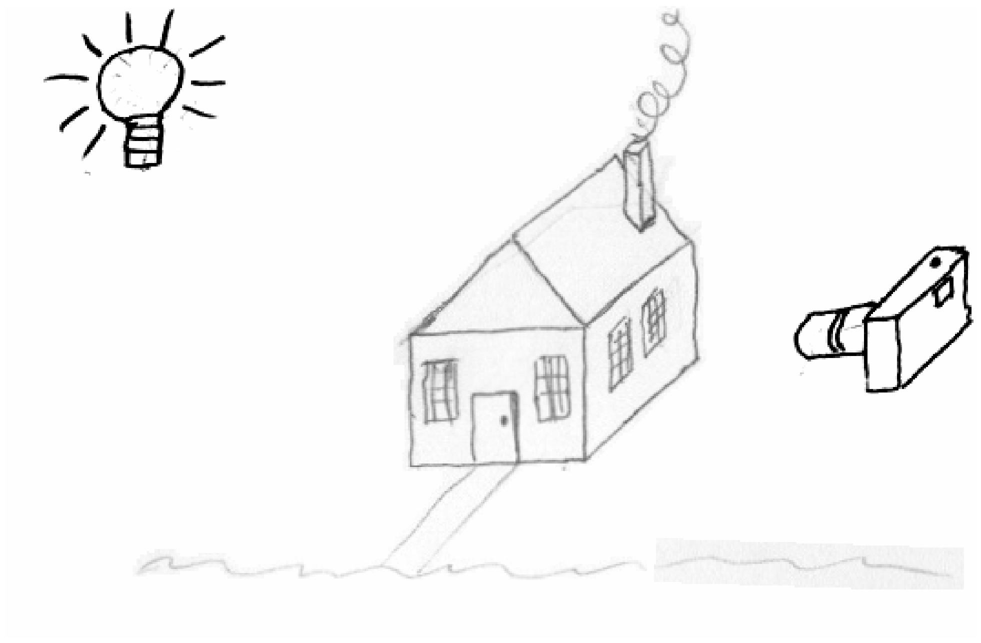
A previous change



Source: John Poulton

The rendering problem

- Given:
  - 3D world
    (objects and materials)
  - Light locations
  - A viewpoint



- Compute:
  - 2D image seen from the viewpoint

The computational questions in rendering

- Where are moving objects located right now?
- What objects are visible…
  - From the viewpoint?
  - From the lights? (shadowed vs. non-shadowed)
  - From other (reflective) surfaces?
- How do objects reflect light?
  - What color? What intensity?
  - How does reflectivity vary with direction?
- How much light reaches…
  - Various surfaces in the scene?
  - The viewpoint, from various directions?

The rendering equation

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{M^2} f_r(x,\omega_i(x-x') \to \omega_o) G(x,x') L_o(x',\omega'_o(x-x'))\, dA'$$

Integrate over All surfaces

Geometry term

$$G(x,x') = \frac{\cos\theta_i \cos\theta'_o}{\|x-x'\|^2} V(x,x')$$

$$V(x,x') = \begin{cases} 1 & \text{visible} \\ 0 & \text{not visible} \end{cases}$$

Visibility term

Source: Pat Hanrahan

All practical real-time systems make major approximations to this equation in order to solve it.

Surface geometry and properties

Geometry is modeled using triangle meshes

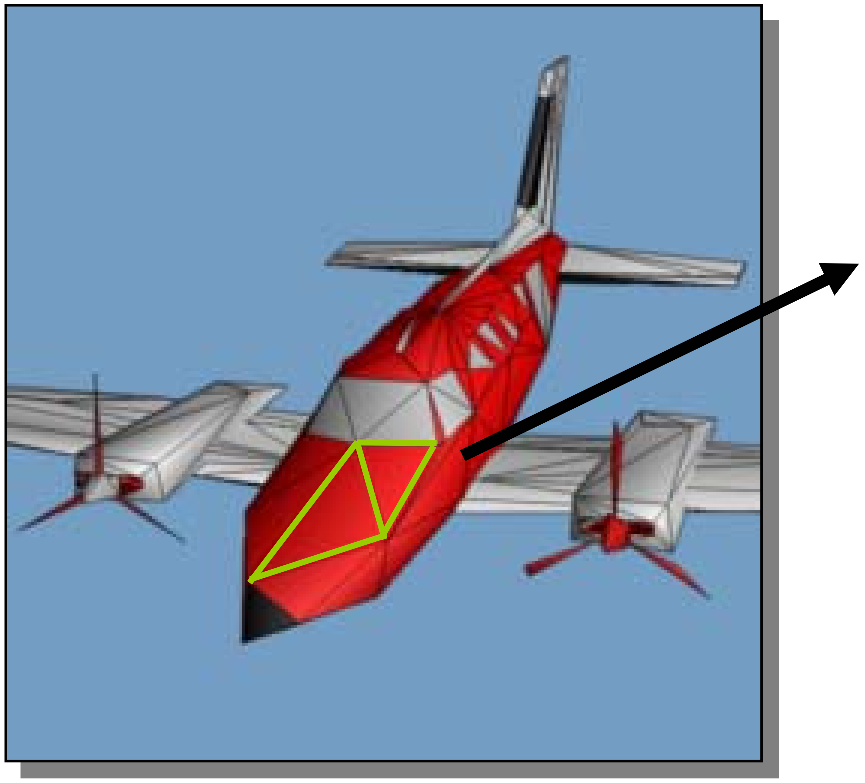
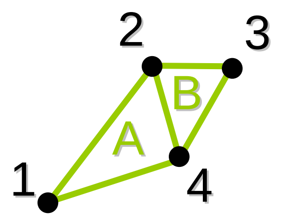


Image: Hughes Hoppe, Microsoft Research



Vertex Array
(x1, y1, z1)
(x2, y2, z2)
…

Index Array
V1, V2, V4 – represents Triangle A
V4, V2, V3 – represents Triangle B

Vertices are only stored once.
Triangles point to their vertices.

Store a surface-normal vector at each vertex

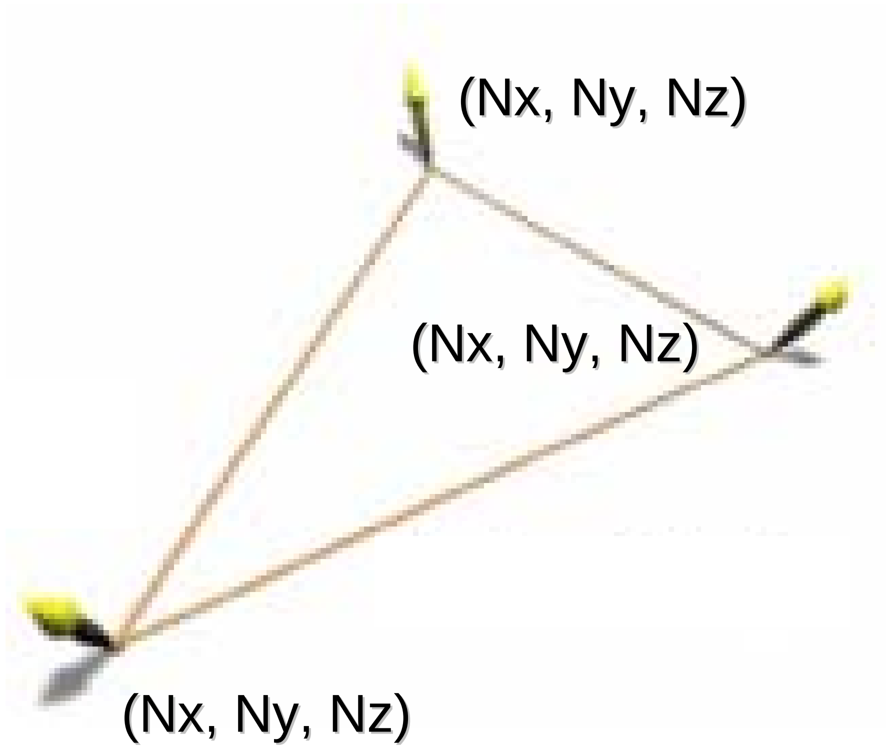


Image: ATI Technologies

- Allows computation of "true" normal vector at any point on triangle
  - Triangle is just an approx to curved surface
- Normal vectors needed to compute surface/light interaction (shading).

Triangles sometimes subdivided by GPU

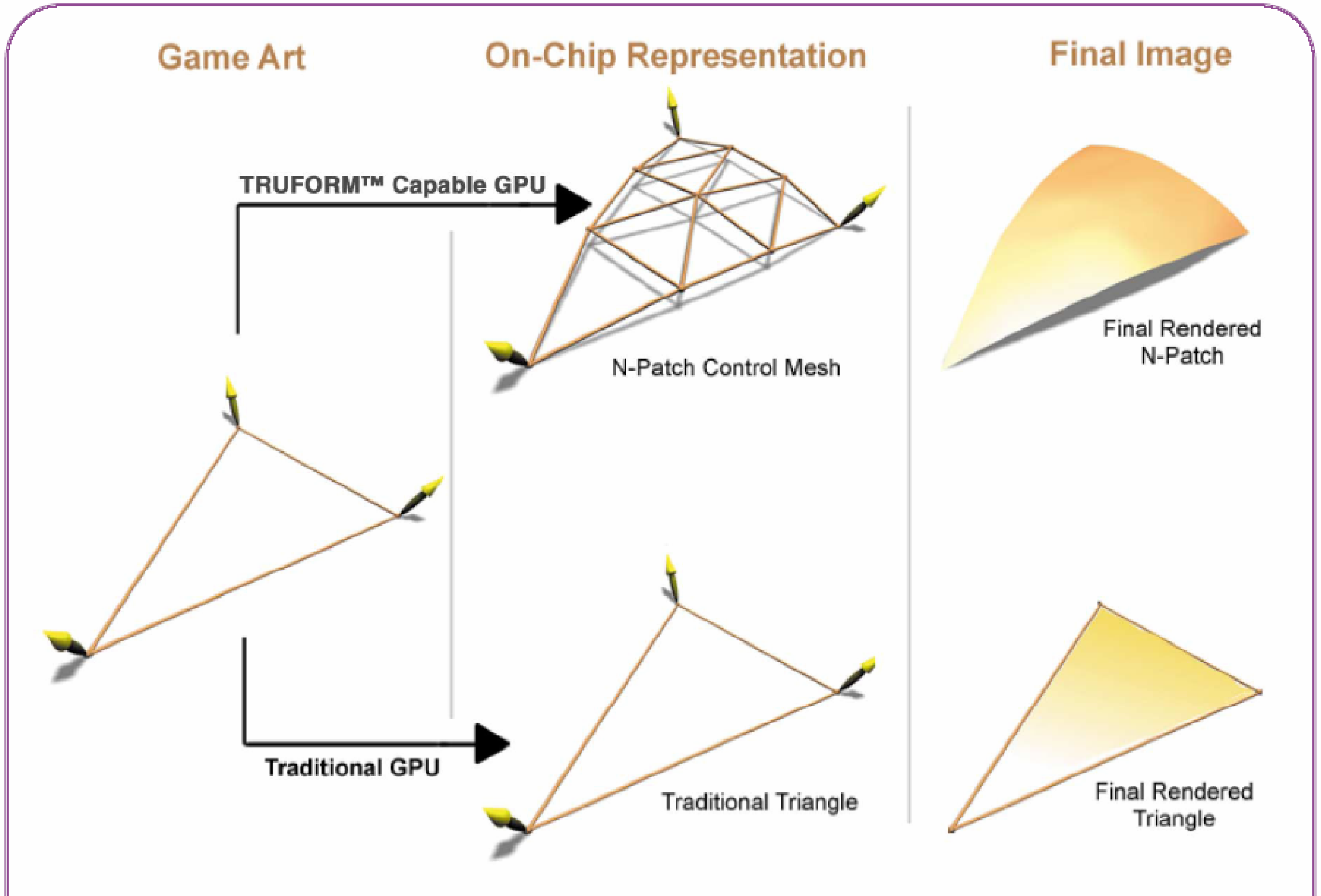


Figure 5: *N-Patches. Note how the input data consists of just three vertices and three vertex normals, and how the game art is compatible with any graphics processor.*

Image: ATI Technologies

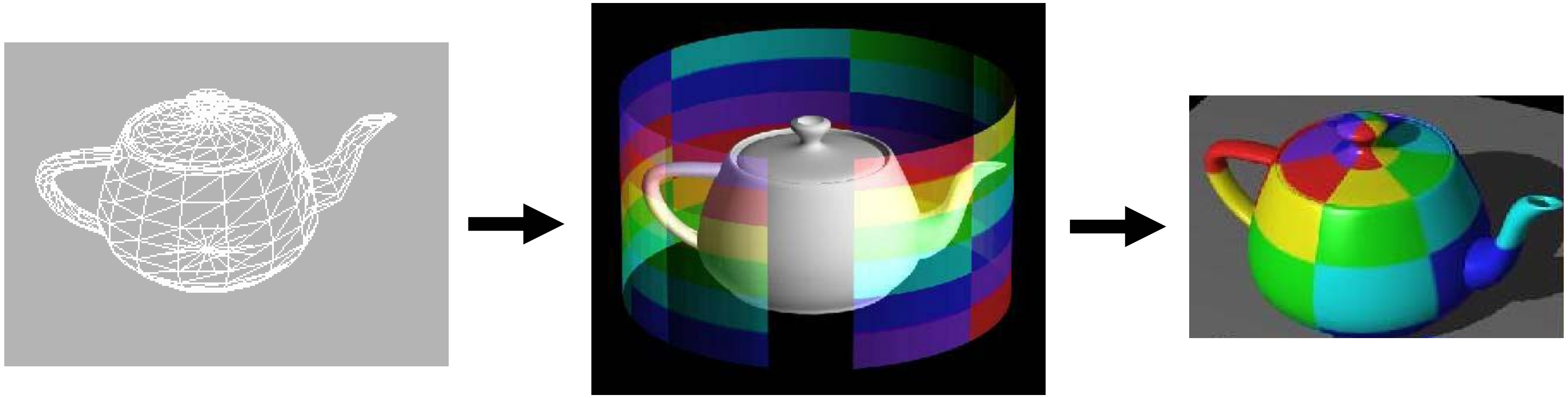
GPU tessellation will become more common in next few years.

- Reduces CPU->GPU Bandwidth

- Tesselation can be adaptive

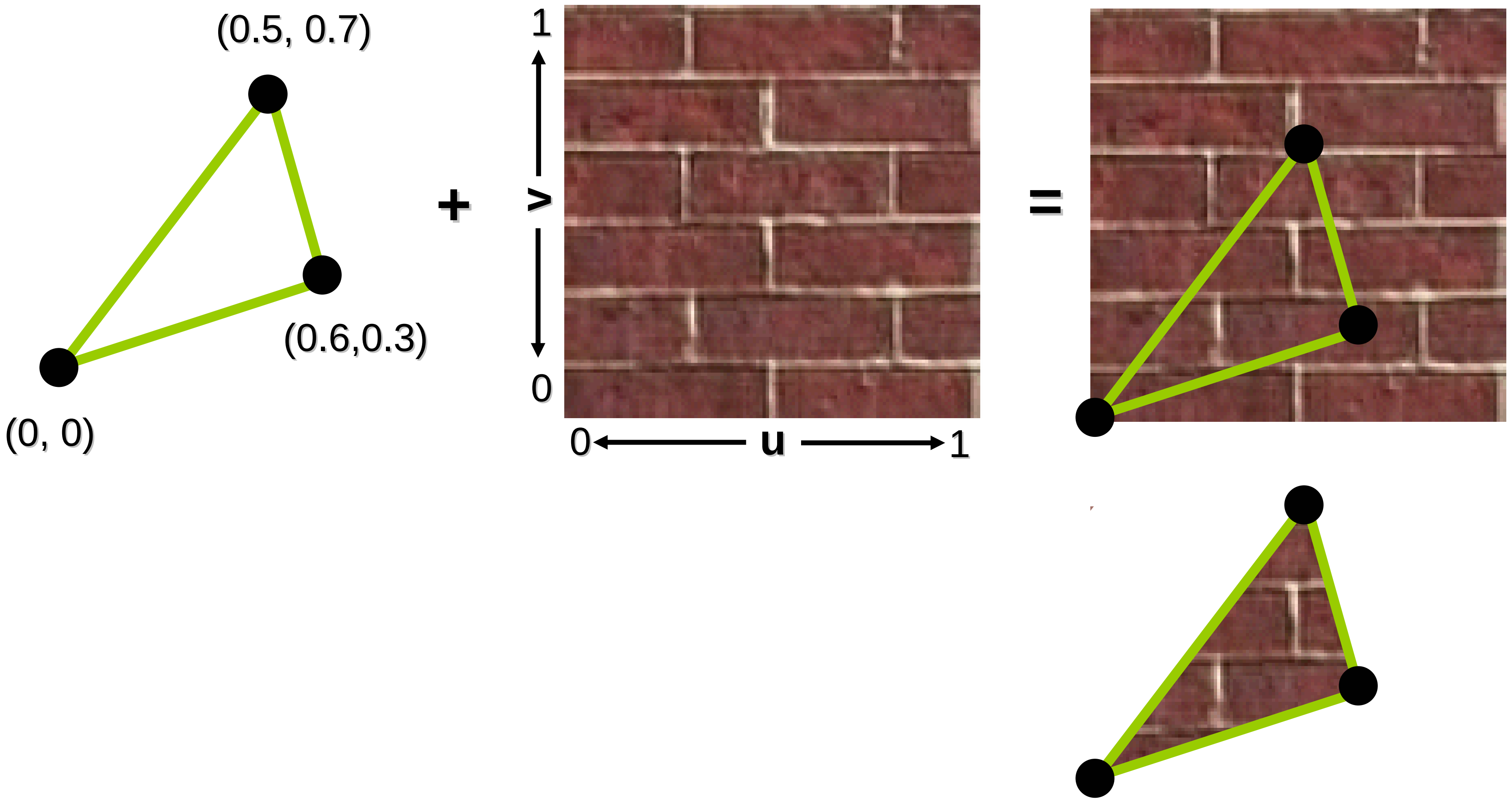
Texture mapping – apply image to geometry

Example #1 – Use simple formula for mapping image to geometry



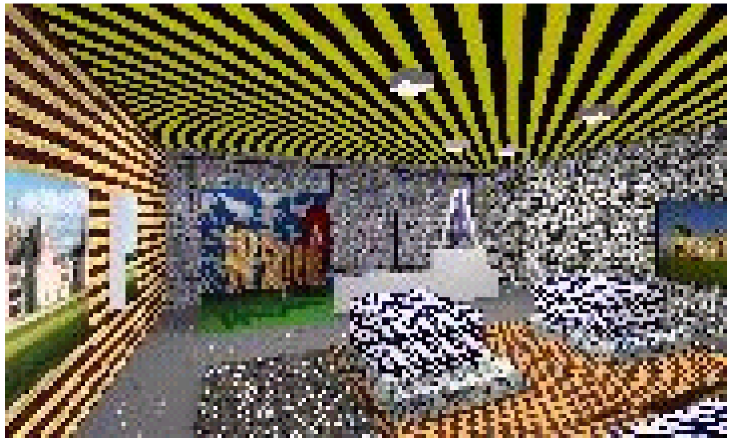
Texture mapping using texture coordinates

Example #2 – Use **texture coordinates** to map image to geometry

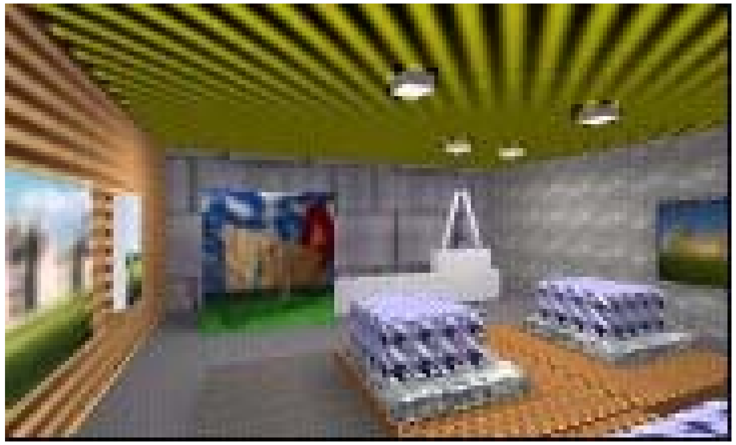


Texture aliasing problem

Simple texture sampling



Better: use MIP-mapping
(a form of filtered sampling)



MIP-maps

- For a texture of 2^n x 2^n texels, pre-compute n-1 textures, each at ½ the resolution of previous:

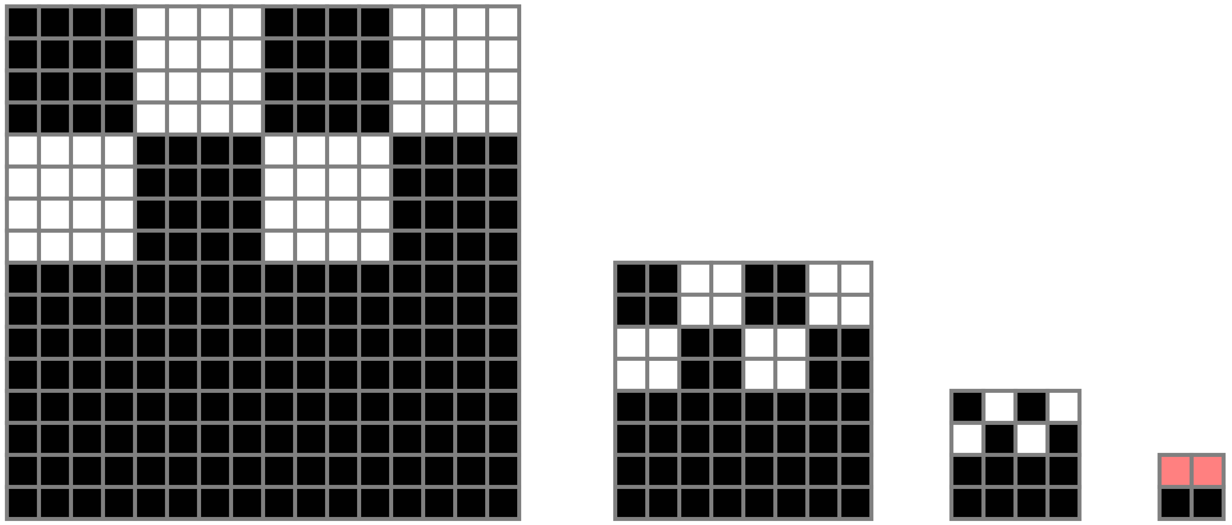

Original Texture *Lower Resolution Versions*

Figure: David Luebke

- This multiresolution texture is called a *MIP-map*

At runtime, for each fragment:

For now, think of fragment as “potential pixel”

- Interpolate texture coordinates from vertices
- Compute needed MIP-map level
  - Use “derivatives” of texture coordinates
  - Actual implementations group fragments into groups of 4, and compute discrete differences.
- Retrieve eight texture samples
  - 4 from “bigger” image, 4 from “smaller” image
  - Small cache captures local reuse; but BW intensive
- Linearly interpolate the eight samples

Anisotropic texture filtering

- Uses more than 8 samples when surface is angled with respect to viewpoint
- Gradually replacing simple MIP-maping as standard texture algorithm
- Architectural implications:
  - Uses more memory BW
  - Uses more computation
  - Variable-iteration loop at each fragment – texturing is no longer “SIMD”.

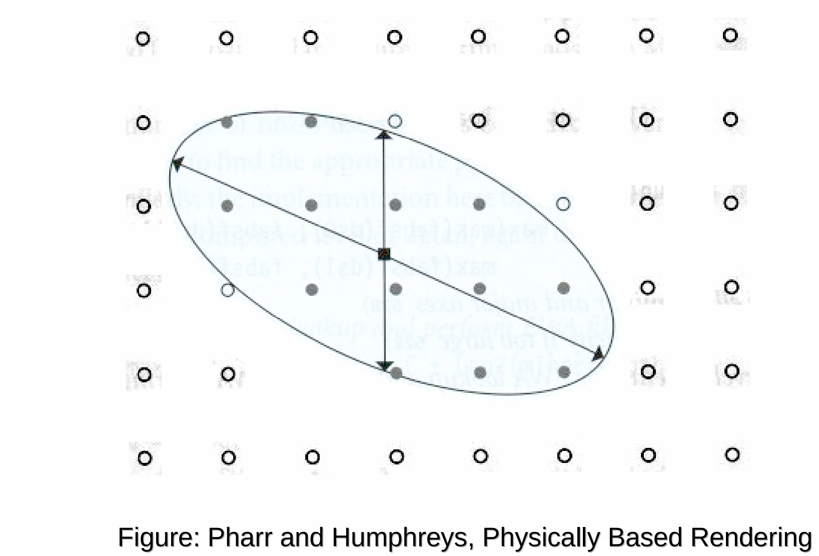
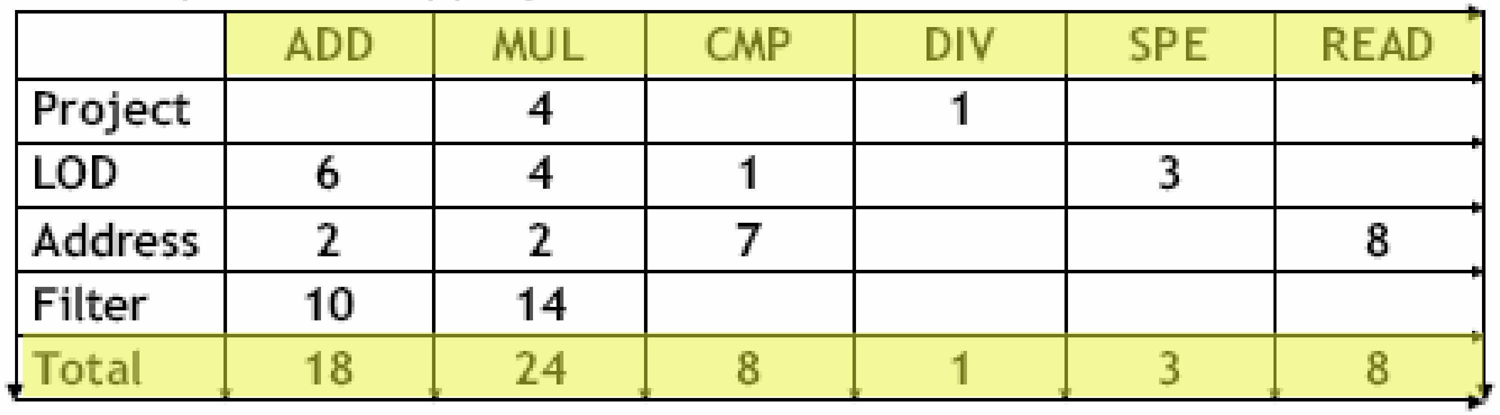


Figure: Pharr and Humphreys, Physically Based Rendering

Per-fragment texturing cost in Z-buffer system

For simple MIP-mapping:

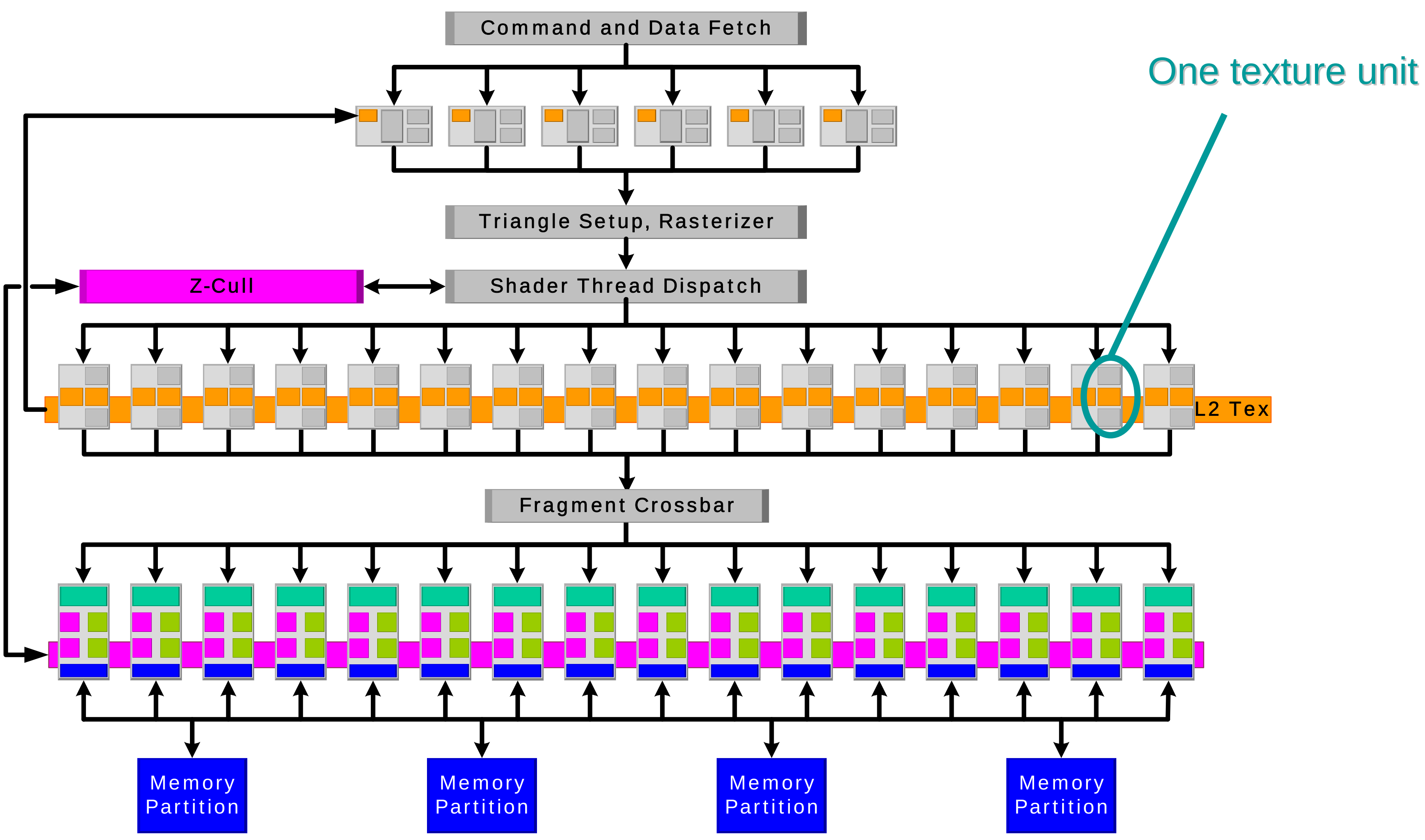


	ADD	MUL	CMP	DIV	SPE	READ
Project		4		1		
LOD	6	4	1		3	
Address	2	2	7			8
Filter	10	14				
Total	18	24	8	1	3	8

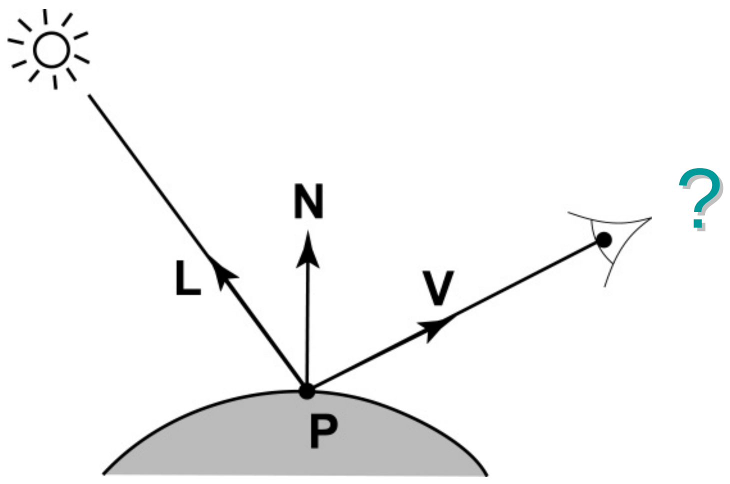
Source: Pat Hanrahan and Kurt Akeley

- Very computationally intensive
  - 2004 GPU's can sustain 8 mipmapped texture lookups **per cycle**
- 16-bit arithmetic is used for filtering computations
- Still implemented primarily with dedicated HW
- Memory reads are tricky – more on this later

Where texture HW lives in Z-buffer pipeline



Surface/light interaction (shading)



P = point on surface
N = surface-normal vector

- Compute light intensity/color leaving surface in a particular direction.
- Given:
  - Incoming light intensity (often expressed using light locations)
  - Surface position
  - Surface normal vector
  - Surface properties

Shading combines two computations

- What are the surface properties at this point?
  - e.g. look up surface color in a texture map
- How does incoming light interact with surface?
  - e.g. what is intensity of reflected light from Light #2?

Shading dominates rendering cost

- ~50% of die area devoted to texturing/shading
  - And this fraction is increasing
  - May eventually exceed 90%, following batch rendering

A simple shading computation

- Phong lighting model at every fragment:

$$I = k_e + k_a I_a + k_d I_\ell (\mathbf{N}\cdot\mathbf{L})_+ + k_s I_\ell (\mathbf{V}\cdot\mathbf{R})_+^{n_s}$$

Also, must first interpolate N and re-normalize it.

Operations:
- 1/x
- 1/sqrt(x)
- x^y
- clamp
- lots of add, multiply

All of this at ~1 billion fragments/sec

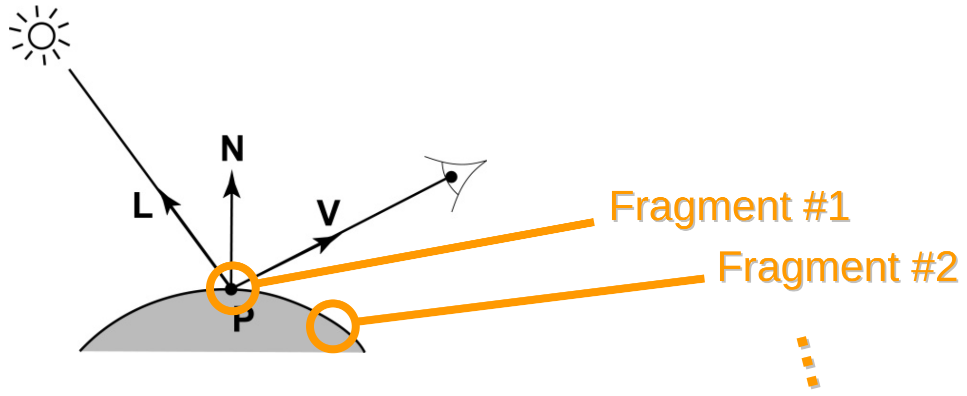
Variety in materials → programmable shaders

- Real world has infinite variety of materials
  - Need programmable shaders to describe them
- Example fragment program in Cg/HLSL:

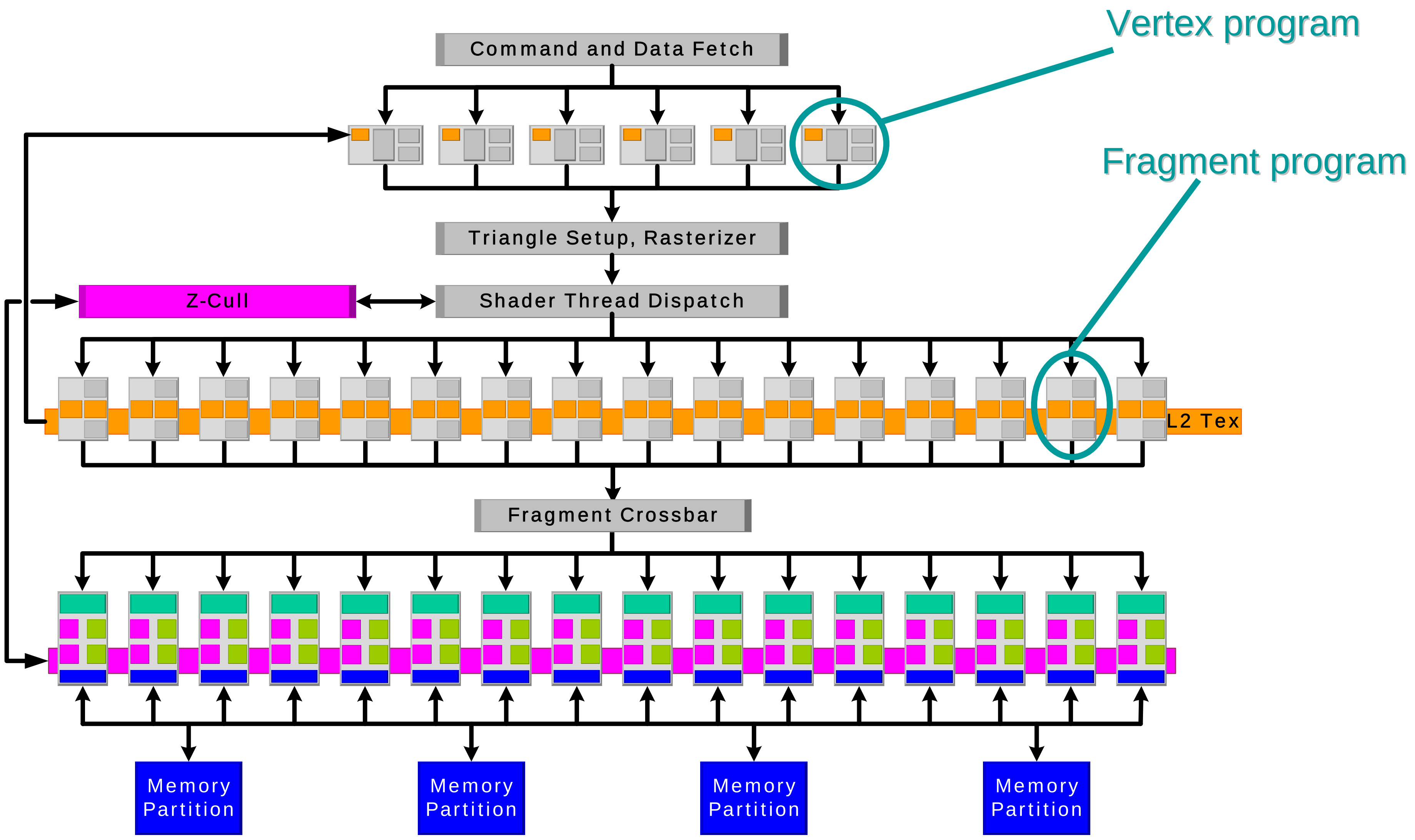
```
void normalmapped(float2 normalMapTexCoord : TEXCOORD0,
                  …
              out float4 color : COLOR,
                  uniform float ambient,
                  …)
{
  float3 normalTex, …;
  normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
  …
  diffuse = saturate(dot(normal, normLightDir);
  …
  color = Kd * (ambient + diffuse ) +
          Ks * pow(specular, specularExponent;
}
```

Programming model makes parallelism easy

- Program is re-run for every fragment (or vertex)
- Perfect parallelism:
  - Program cannot communicate with other fragments
  - No persistent state (each fragment is independent)
  - In most respects, a “stream programming” model
    - Each fragment gets one input record and one output record
    - Fragment program = “stream kernel” or “filter”



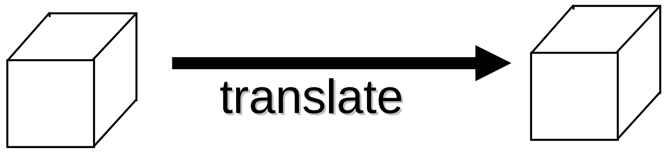
Where shader programs execute



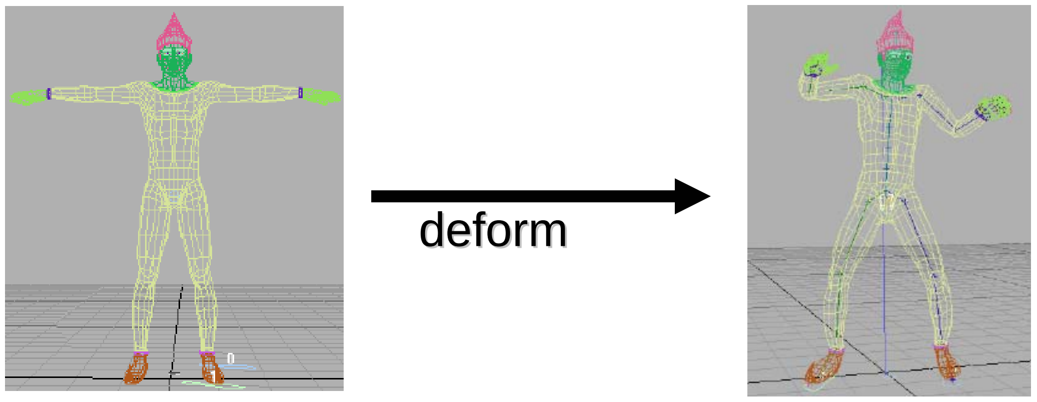
Animation:
Objects that move and deform

Two main questions:

- How do we describe object movement?



- How do we describe object deformation?



Translation and rotation – the “math” way

Translation
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\text{new}} = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\text{old}}$$

Translation vector

Rotation
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\text{new}} = \begin{bmatrix} R_{xx} & R_{xy} & R_{xz} \\ R_{yx} & R_{yy} & R_{yz} \\ R_{zx} & R_{zy} & R_{zz} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\text{old}}$$

3x3 rotation matrix

Translation and rotation – The "graphics" way

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{\text{new}} = \begin{bmatrix} R_{xx} & R_{xy} & R_{xz} & T_x \\ R_{yx} & R_{yy} & R_{yz} & T_y \\ R_{zx} & R_{zy} & R_{zz} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{\text{old}}$$

Implicit division

4x4 rotation/translation/etc. matrix

This is why GPUs (and SSE2) support 4-vector short-SIMD operations.

Dot product and MAC are critical…

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\text{final}} = \frac{1}{w} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{\text{new}}$$

Division is eventually performed explicitly.
But HW should just support reciprocal.

Scene graph describes entire scene

- Scene graph is a data structure:
  - Includes objects and 4x4 xforms.
  - Managed by CPU.
  - Updated as objects move.
- CPU traverses scene graph every frame:
  - Feeds 4x4 xforms, shaders to GPU
  - Feeds triangles to GPU
  - May skip parts of scene known to be hidden from view.
- Games are often CPU-limited!

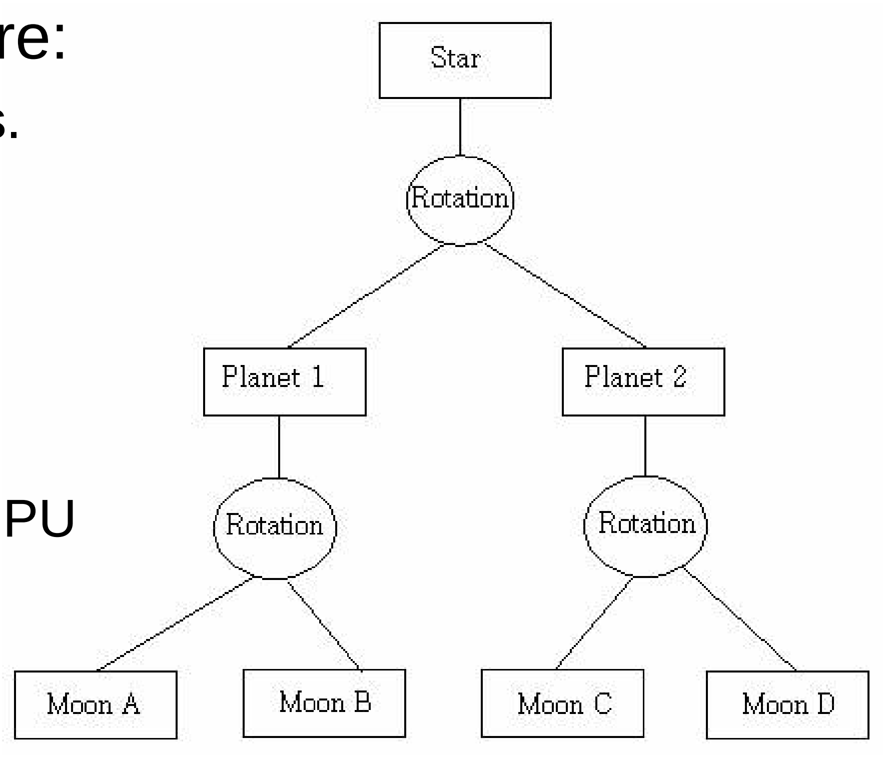


Figure: gamedev.net

“Skinning” – blending multiple xform matrices

- Used for non-rigid objects, like humans
- One xform matrix for each major bone
  - E.g. upper arm, lower arm
- Vertices near bone boundaries use two or more matrices.
  - E.g. near elbow, shoulder.

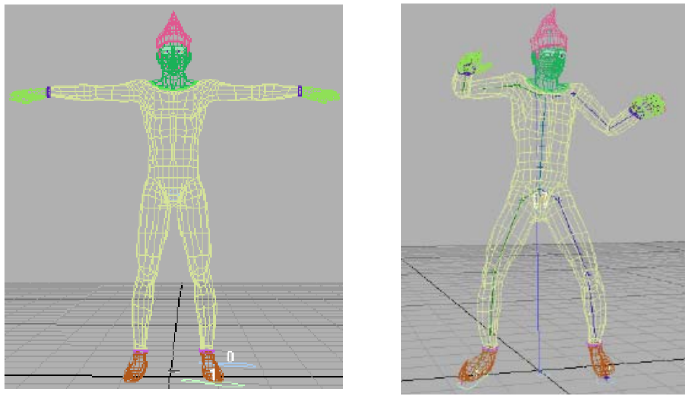


Figure: nvidia.com

Skinning needs variable-iteration loop

```
for (int i = 0; i < numBones; i++)
  {
     // transform the offset by bone i
     transformedPosition = transformedPosition + curWeight.x *
                              vec4((boneMatrices[int(curIndex.x)] *
                              position).xyz, 1.0);

     // transform normal by bone i
     transformedNormal = transformedNormal + curWeight.x *
                              (mat3(boneMatrices[int(curIndex.x)]) * normal).xyz;

     curIndex = curIndex.yzwx;
     curWeight = curWeight.yzwx;
  }

  gl_Position = gl_ModelViewProjectionMatrix * transformedPosition;
```

Visibility and Z-buffers

Hidden surface problem

- At each pixel,
  which surface is closest to the viewpoint?
  - We want the pixel to be the color of that surface point
- Many different algorithms for this task:
  - "Z-buffer" is used in all current GPU's
  - But augmented with preprocessing on CPU
  - Other solutions used in past (and future?)

- More general version of this problem:
  - Which surface, if any, comes between points A and B?
  - Needed for shadows, global illumination, …

The Z-buffer algorithm

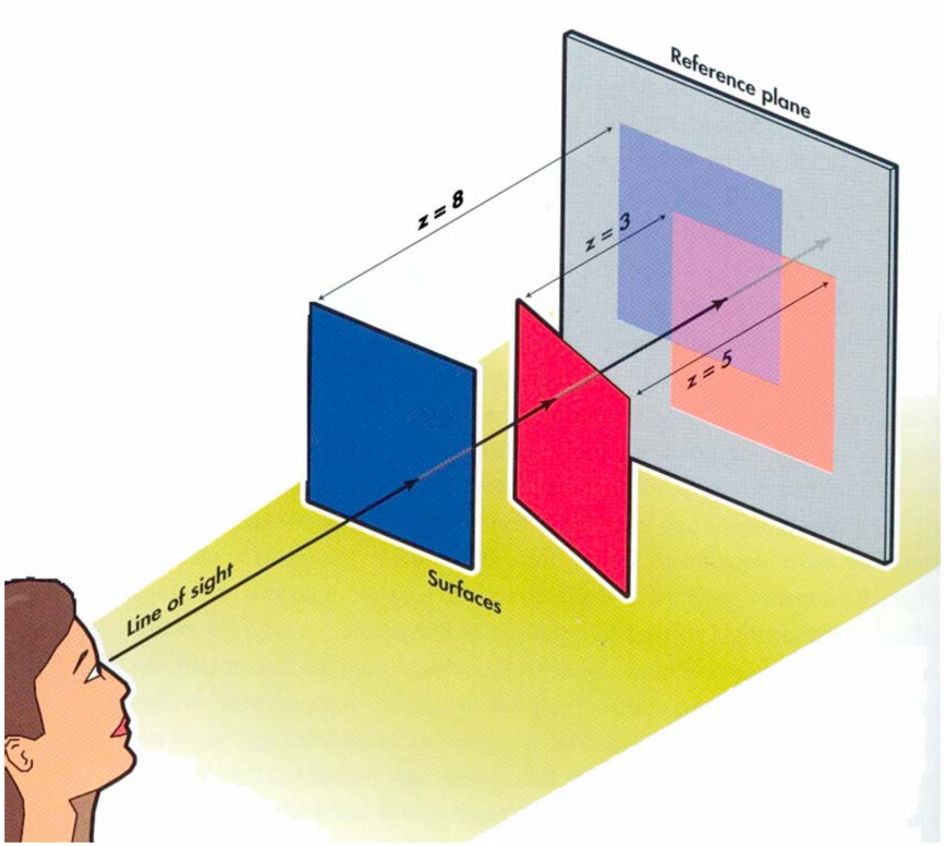
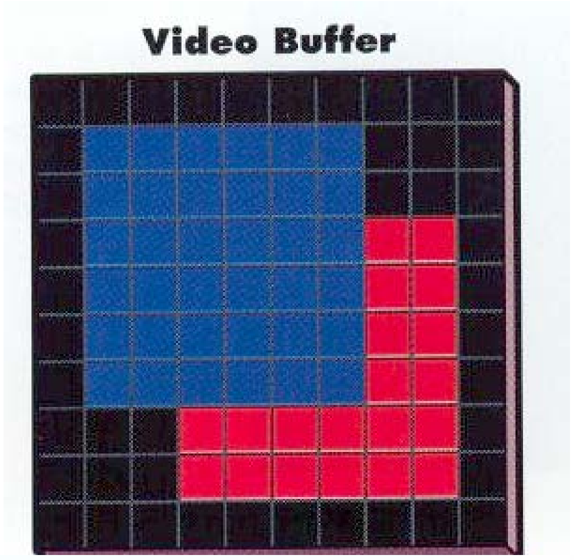
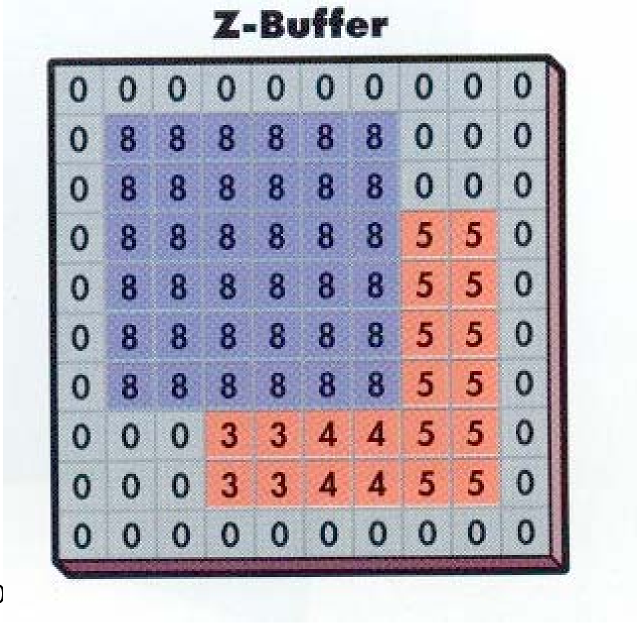


Figure: Prosise, How Computer Graphics Work




Z-Buffer

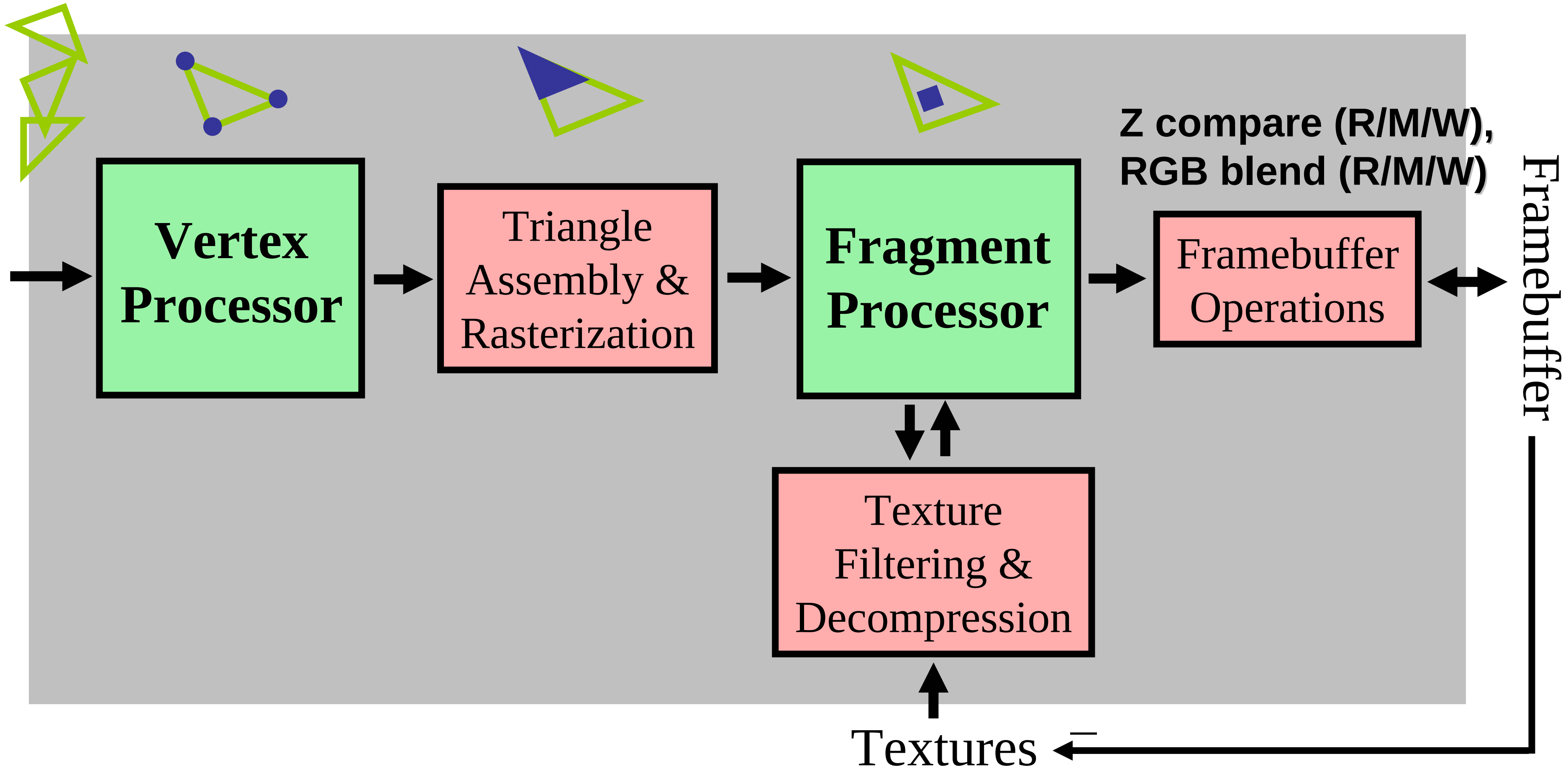
0	0	0	0	0	0	0	0	0	0
0	8	8	8	8	8	8	0	0	0
0	8	8	8	8	8	8	0	0	0
0	8	8	8	8	8	8	5	5	0
0	8	8	8	8	8	8	5	5	0
0	8	8	8	8	8	8	5	5	0
0	8	8	8	8	8	8	5	5	0
0	0	0	3	3	4	4	5	5	0
0	0	0	3	3	4	4	5	5	0
0	0	0	0	0	0	0	0	0	0

Z-buffer algorithm uses “brute force”

```
for each pixel (i,j) do
    Z-buffer [i,j] ← FAR
    Framebuffer[i,j] ← <background color>
end for
for each polygon A do
    for each pixel in A do
        Compute depth z and shade s of A at (i,j)
        if z > Z-buffer [i,j] then
            Z-buffer [i,j] ←    z
            Framebuffer[i,j] ←    s
        end if
    end for
end for
```

- Touches each polygon exactly once.
- Application can choose polygon order.
- But: Nearly-random accesses to frambuffer.

Modern Z-buffer graphics pipeline



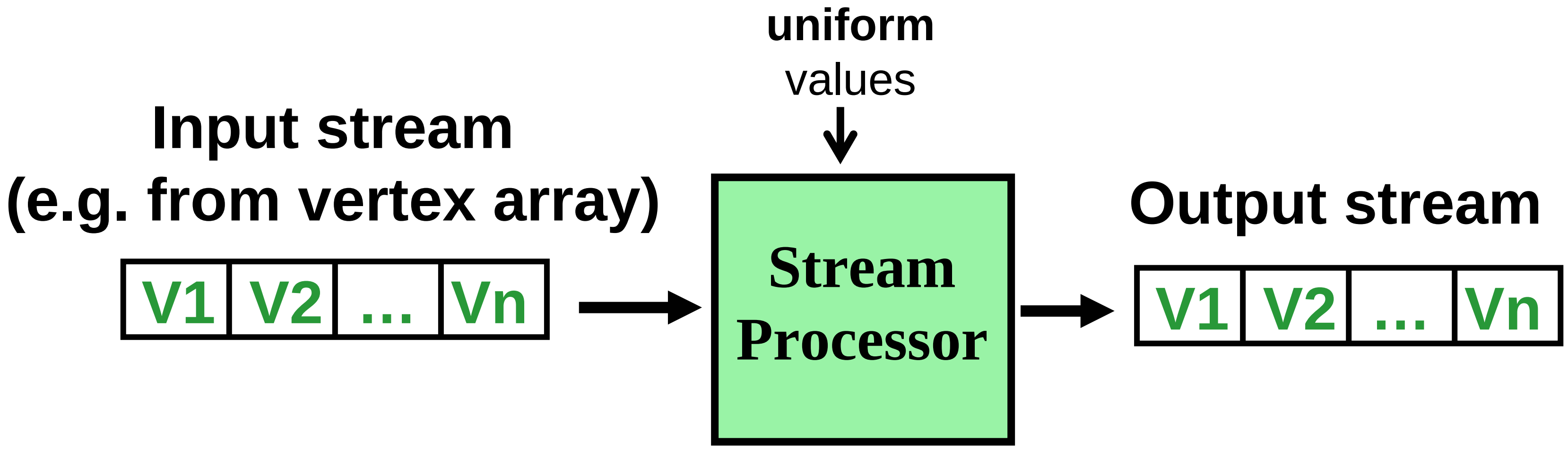
🟩 = Programmable

🟥 = Not programmable – hardwired algorithms

Why is graphics hardware fast?

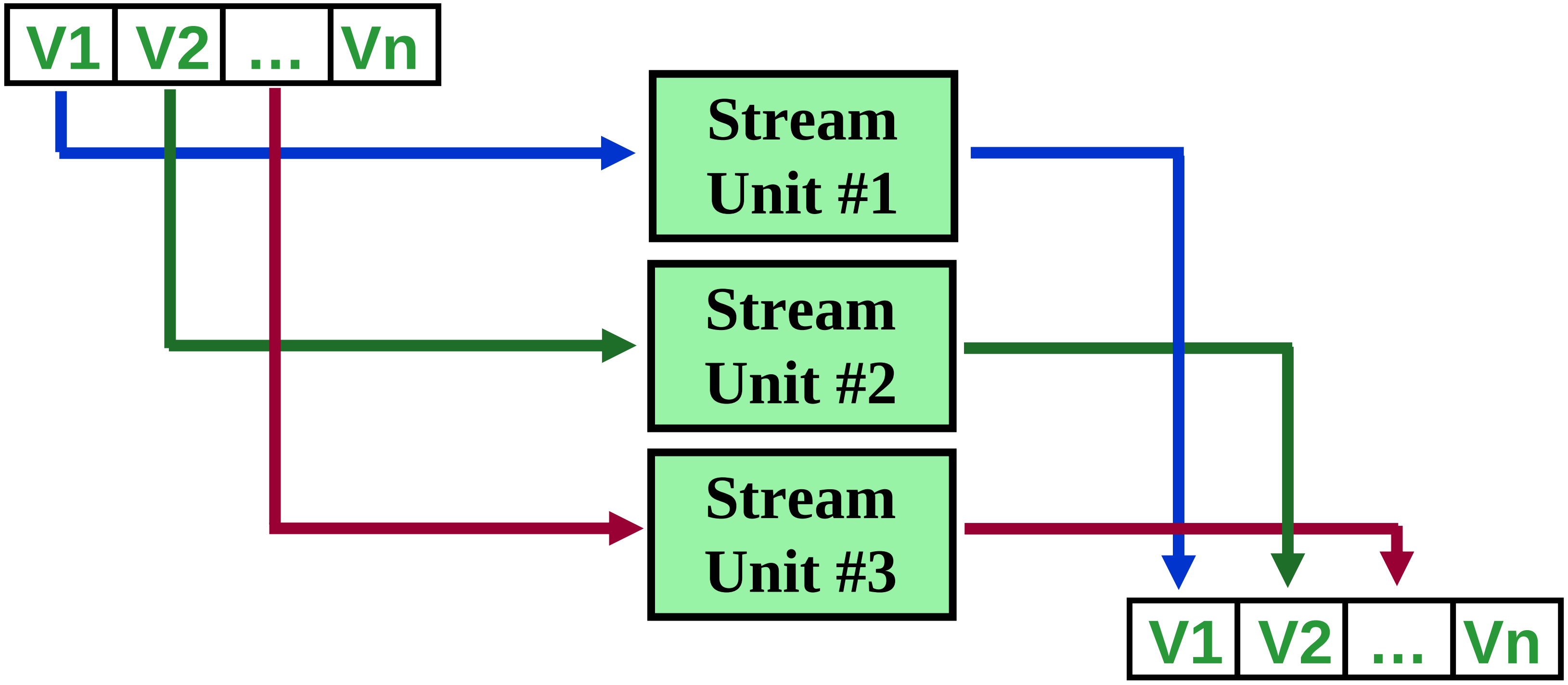
- Specialization
  - Gradually becoming less important (esp. for FLOPS)
  - But still matters a lot (esp. for memory subsystems)
- Parallelization
  - Rapidly becoming more important
  - Two kinds:
    - Task parallelism – pipeline of operations
    - Data parallelism

Programmable units in GPUs process data streams



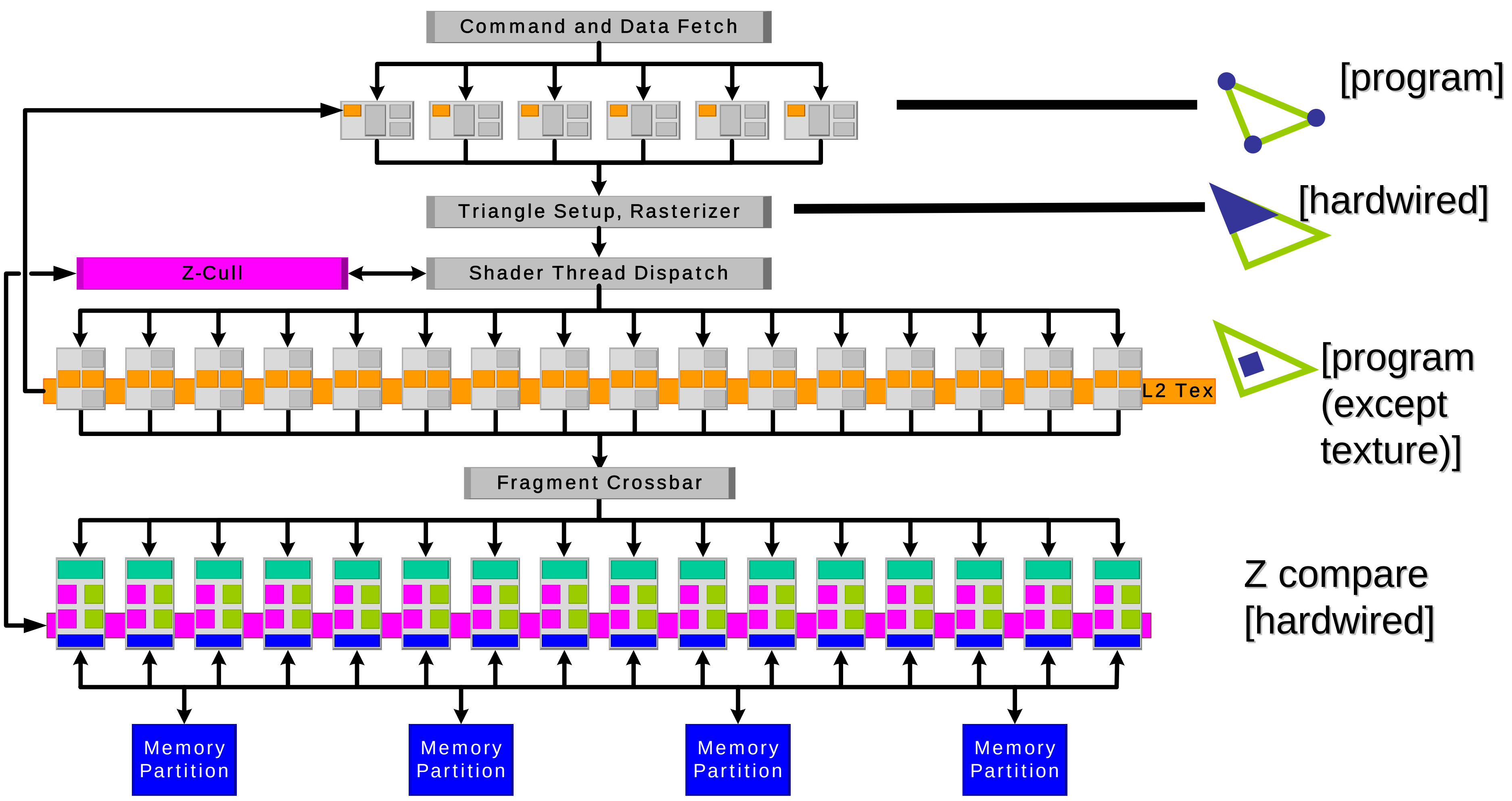
- **The programmable unit executes a computational kernel (i.e. vertex program) for each input element (i.e. vertex)**
- **Streams consist of ordered elements**
- **Fragment processor can read from texture memory**

Stream model supports data parallelism



- Communication between elements is prohibited

Lots of data parallelism – at *most* stages



A bane of parallelism: Order matters!

- Current APIs require in-order triangle completion
  - Observable if Z compare is a tie
  - Or when old triangle is blended with new one
- Requirement reduces to:
  - In-order completion of fragments from *different* triangles that map to the *same* pixel.
  - Non-overlapping fragments can (and do) complete out of order.
- Places major constraints on possible architectures
- Serialization points exist at command processor and rasterizer.

But why not eliminate ordering requirement?

- Several answers:
  - Ordering is useful, particularly for blending
  - Ordering requirement defines results precisely
    - Without it, could have race conditions
    - Different hardware (or even clock rates) could produce different results from same program!
    - Rapid HW evolution requires precise programming model
- This is a fundamental issue in parallel architecture
  - Could the GPU solution be generalized?
  - Not clear if current GPU approach can be sustained as GPU programming models become more flexible

Multisampling – decouple visibility & shading

- Goal: reduce “jaggies” at object edges

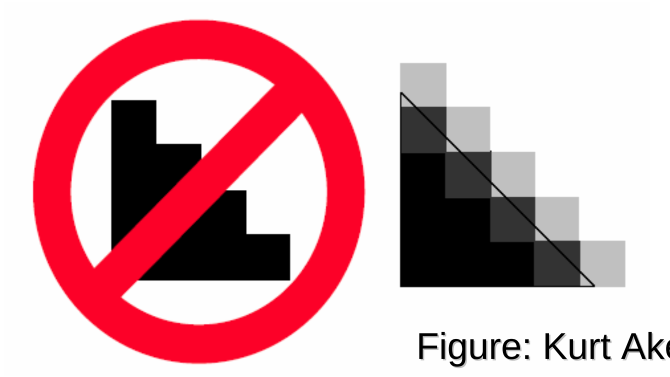
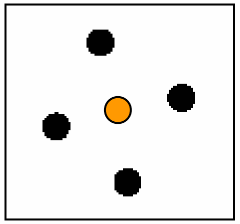


Figure: Kurt Akeley

- Solution: Compare Z values at several points within pixel
  - Color only computed once per pixel (per triangle)
  - Color and Z are stored 4 times (but possibly compressed!)

One pixel:

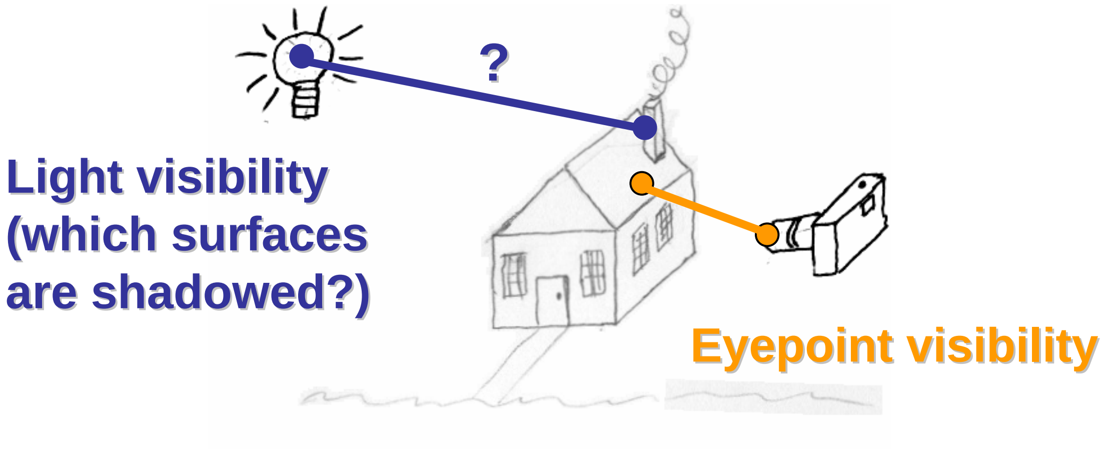


● Z sample locations

● Color sample location

More difficult visibility problems
(not just from eyepoint)

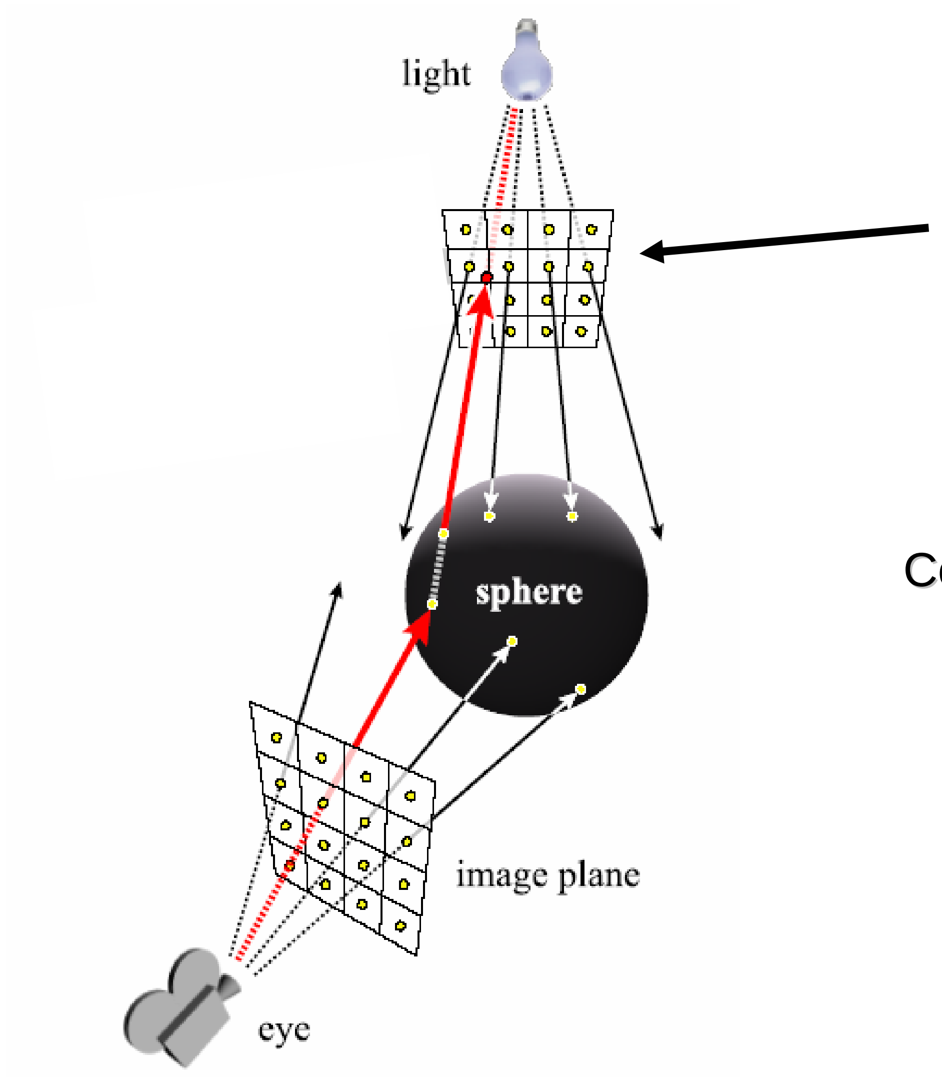
Shadows: Visibility from light



Any surface that is not visible from the light is in shadow.

Simplest case: Treat light as a single point (“hard shadows”)

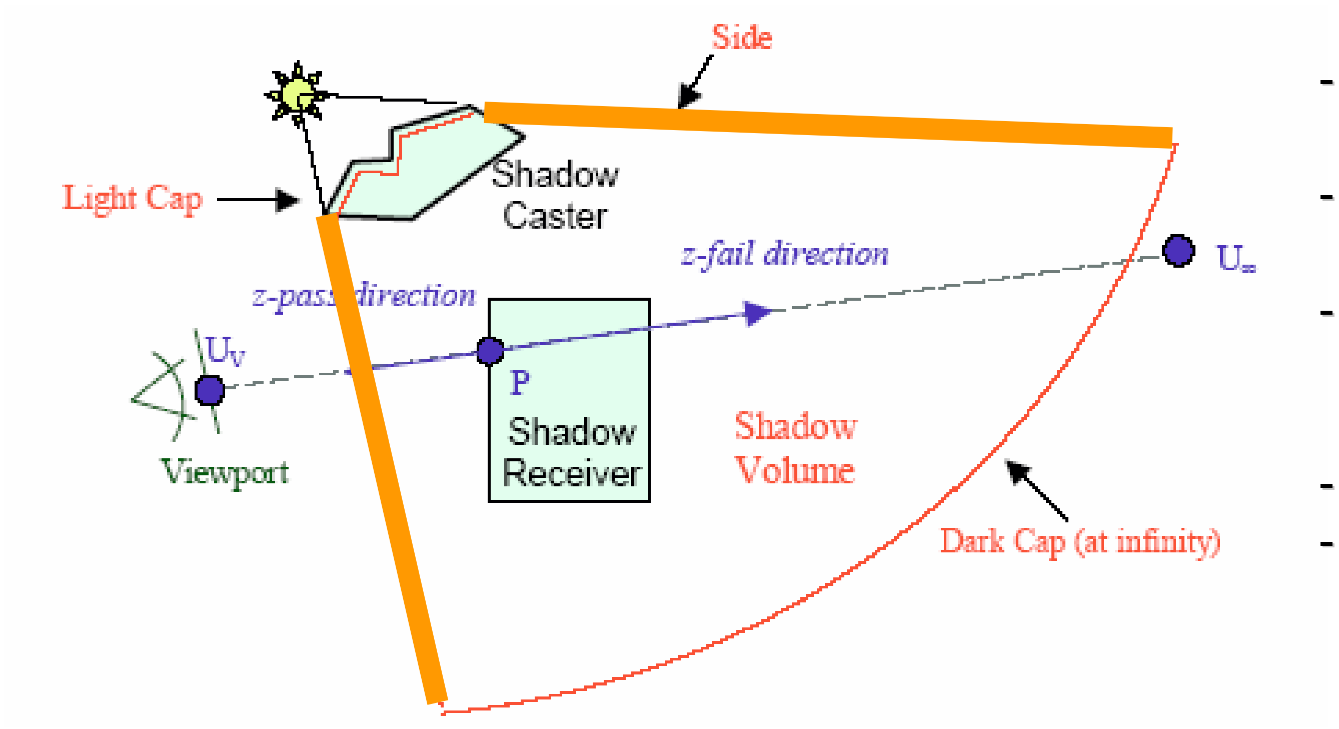
Solution #1 for hard shadows: Shadow maps



Shadow map =
Auxiliary Z-buffer using
light as viewpoint

Comments:
- Conventional shadow maps prone to sampling artifacts.
- Current games do not use shadow maps
- Enhanced variants likely to fix these problems over next few years.

Solution #2 for hard shadows: Shadow volumes



- Uses "stencil buffer" (counter at each pixel)
- Renders many large polygons.
- Demands high stencil buffer bandwidth

- Used in DOOM 3 game
- But algorithm likely to be phased out in near future.

Source: McGuire *et al.*: Fast, Practical, and Robust Shadows, 2003

Shadow algorithms are in flux

- Realistic shadows are new in games (~1 year)
- Current approaches are inadequate
  - Issues with performance and/or image quality
- Wide variety of new techniques under development
- Likely to be major driver of new HW capabilities
- Interaction between algorithms and hardware is especially complex

My opinion:

- Representative of future challenges for GPU's.

Data Types

Data type (precision) summary

	Old	**New**
Fragment processor	fixed10-12	float32
Framebuffer color, blend unit	fixed8	fixed8, float16
Textures, texture filter	fixed8	fixed8, float16
Vertex processor (positions)	float32	float32
Rasterizer	Various float	Various float

Increasing precision driven by:

- Programmable shading -- [fragment processor]
- High-dynamic-range rendering and framebuffers -- [texture, framebuffer, blend]
- Global illumination (mostly for future) -- [fragment processor, framebuffer, textures]

Henry Moreton – GPU case study

[35 Missing Slides]

We are unfortunately unable to put Henry Moreton's portion of the presentation (35 slides in total) online.
Please contact Henry directly for any questions about this matter.
His email is <hmoreton-at-nvidia-dot-com>.

Bill Mark - More architectural details

Massive multithreading can hide memory latency

(a.k.a. “Never stall on a cache miss”)

- Consider texture mapping:

```
for each fragment {
    compute_texture_addresses();
    texels = memory_read(texaddress1, 2, 3, 4, 5, 6, 7, 8);
    compute_color(texels);
}
```

- Each fragment is a thread
- Context switch on texture fetch
  - Must hide memory latency – cache miss rate > 10%
- Need 100+ threads per fragment processor!
  - Fortunately, thread context is small (<< 100 bytes, typ.)

Texture prefetching architecture

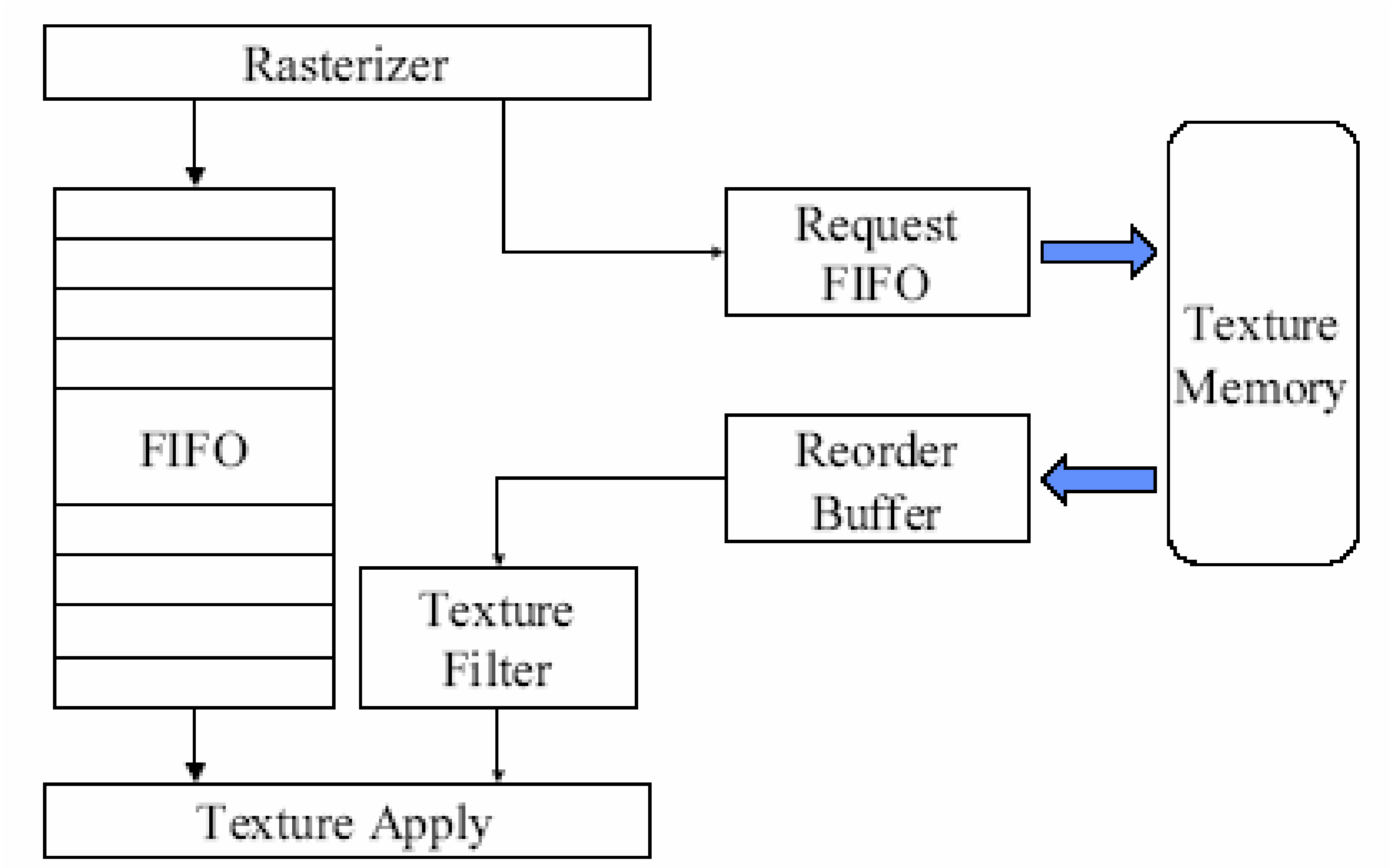


Figure: Pat Hanrahan and Kurt Akeley.
Reference: Ighey *et al*, Prefetching in a Texture Cache Architecture

Prefetching with a texture cache

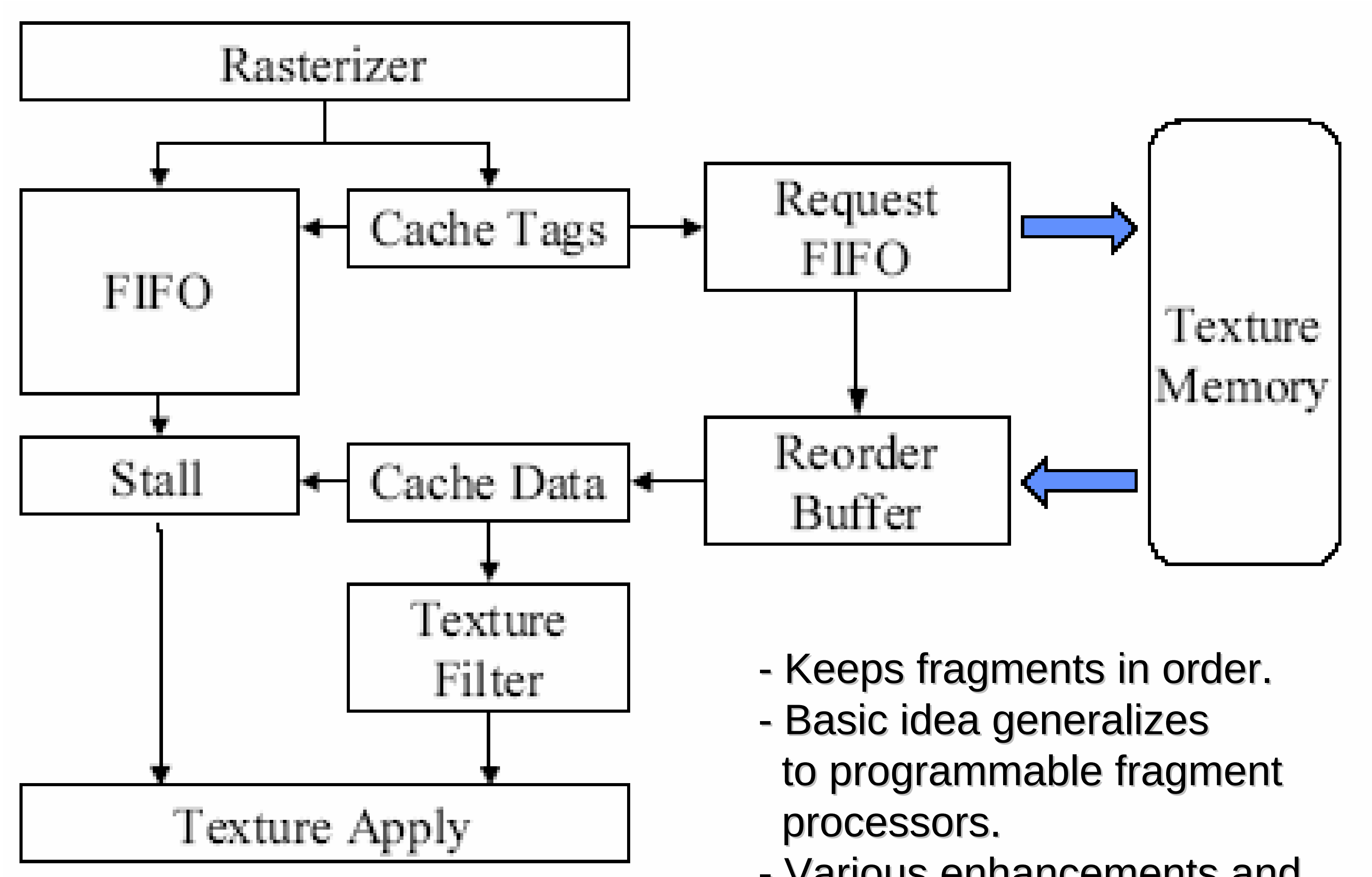


Figure: Pat Hanrahan and Kurt Akeley.

- Keeps fragments in order.
- Basic idea generalizes to programmable fragment processors.
- Various enhancements and modifications can be made.

Multithreaded prefetch also used for framebuffer

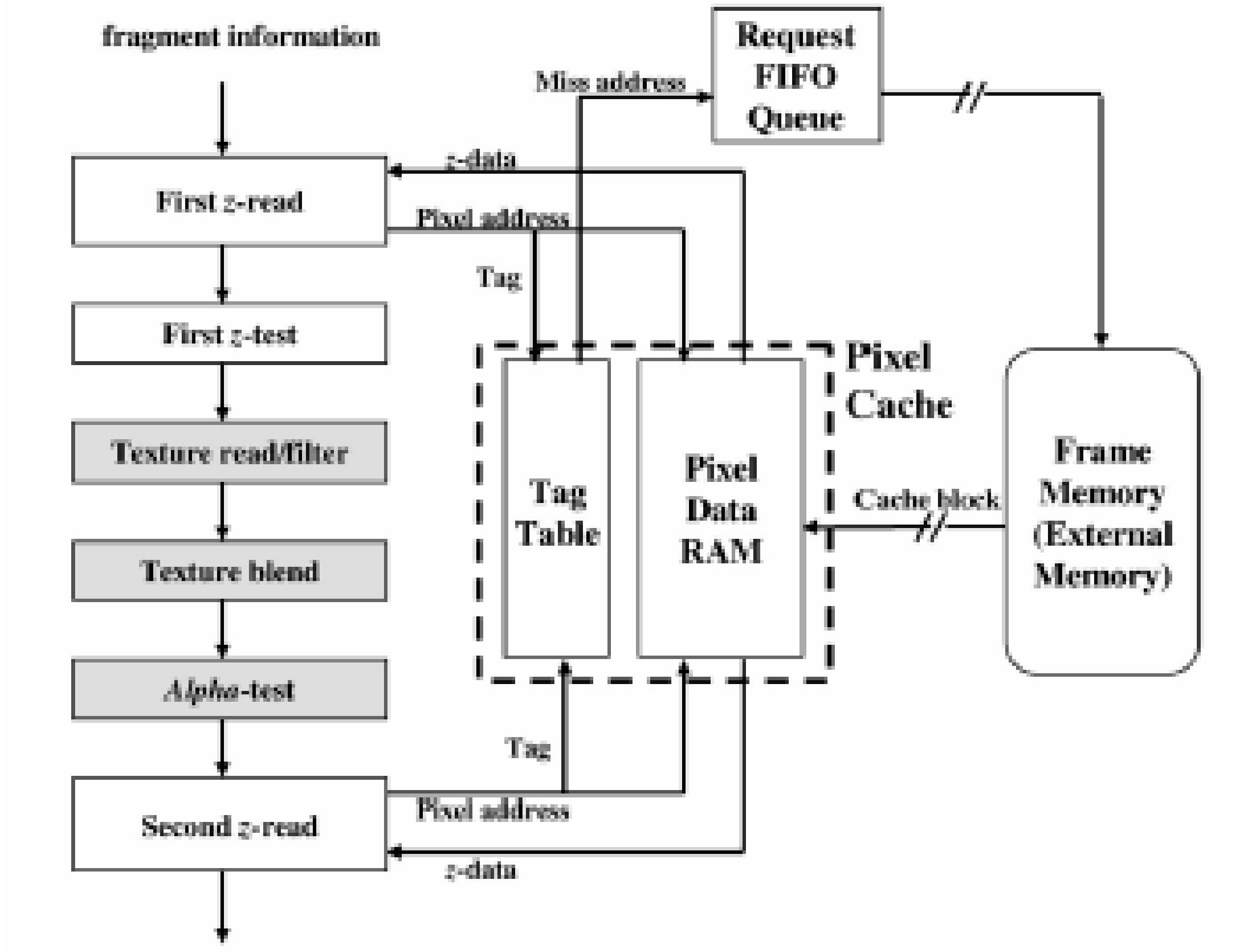


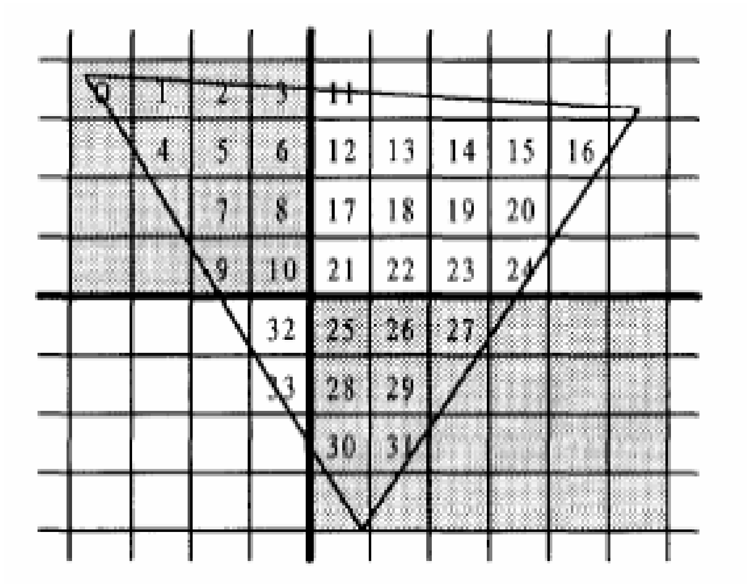
Figure: W. Park et al,
An Effective Pixel Rasterization Pipeline Architecture for 3D Rendering Processors, 2003
Also, see NVIDIA and ATI patents, e.g. US #6,734,861, filed Oct 2000.

Framebuffer has R/W hazards

- Semantics say:
  - Preserve ordering
  - Atomic R/M/W for Z compare
- In practice:
  - Semantics only matter for two fragments at same pixel
  - Detect this and special case it
    - Conceptually, 1 million locks (one for each pixel)
    - Hash instead!
- All of this is hardwired
  - Needs very high throughput
  - One of the big “tar pits” for general purpose hardware

Maximize cache hit rates with 2D tiling

- Framebuffer and textures organized into tiles
  - Allows capture of 2D spatial coherence by caches
- Rasterizer generates fragments in tile order
- All of this is hardwired



Reference:
McCormack et al,
Neon: a single-chip 3D workstation graphics accelerator,
1998

Other important optimizations

- Early fragment kill
  - Perform Z and/or stencil test before shading, texturing
  - Be careful, since semantically it occurs afterward
- Hierarchical (low-res) Z/stencil buffers
  - Keep low-res buffers on-chip
  - Improves performance of early-discard tests
  - Annoying interactions with other features
    - E.g. Turn this stuff off if fragment shader can modify Z

Miscellaneous

Yield tricks

- Top-of-the-line HW has 16 fragment units
  - But it’s quite hard to find these parts
- Almost-top HW has 12 fragment units
  - Much easier to find these parts
- Why might that be?

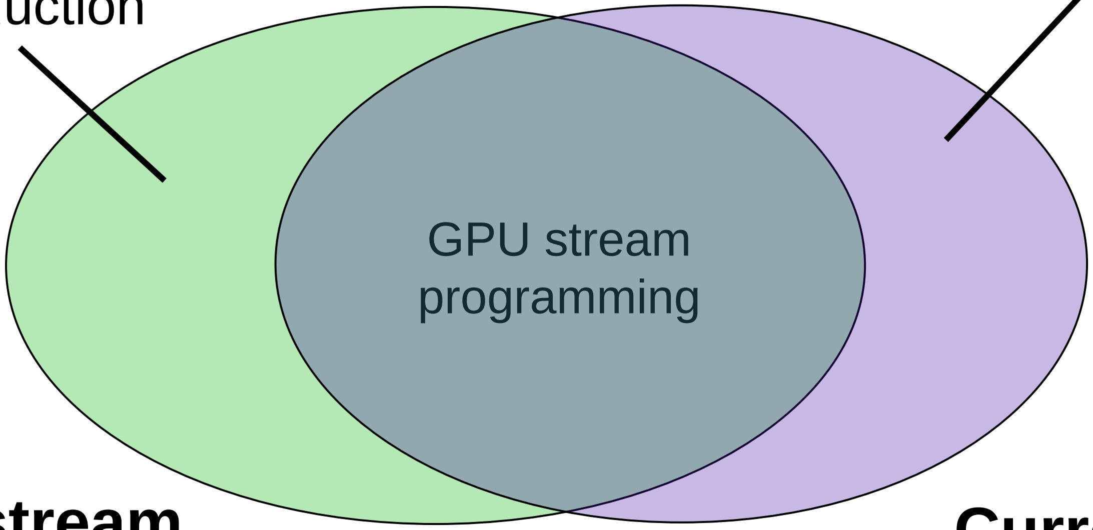
General purpose computation on GPUs

- Use fragment processors as stream processors
- Specialized languages for this purpose
  - Brook for GPU’s [Buck et al., 2003]
  - Sh [McCool et al, 2004]
- Applications include:
  - Image processing
  - Some BLAS routines (single-precision only)
  - Ray casting

Full stream proc $\not\equiv$ GPU

- Scatter to memory
- Conditional kernel outputs
- Efficient reduction
- etc.

- Tagged caches
- R/M/W blend,Z
- etc.



Full stream programming
(e.g. Imagine processor)

Current GPU's
(efficient Z-buffer rendering, with programmable shading)

Fortunately, many of the more general stream programming features that today's GPU's lack could be added with minimal impact on cost and rendering performance

Historical trends

Year	NVIDIA Product	Mtri/ sec	Mfrag/ sec (*)	BW GB/sec	Clk MHz	Trnst cnt (M)	Proc (um)
1998	Riva ZX	3	100	1.6	100	4	.35
1999	Riva TNT2	9	350	3.2	175	9	.22
2000	GeForce2 GTS	25	664	5.3	166	25	.18
2001	GeForce3	30	800	7.4	200	57	.18
2002	GeForce4 Ti 4600	60	1200	10.4	300	63	.15
2003	GeForce FX	167	2000	16.0	500	121	.13
2004	GeForce 6800 Ultra	170	6800	35.2	425	222	.13

* Fragment fill rate for 1 texture.

Source: Mark Kilgard, NVIDIA

- Yearly growth rates well above CPU rate of ~1.5
  - While adding substantial new functionality!
- But growth rates for BW & die area probably unsustainable

Recap

Why is graphics hardware fast?

- Specialization
  - Serial bottlenecks such as rasterization
  - Memory access, caching, compression, addressing
  - Ordering of parallel memory writes
  - Shepherding of parallelism, data flows, communication
  - Smart work avoidance: early Z tests, etc.
  - Texture filtering
- Parallelism
  - Multithreaded vertex processor
  - Multithreaded fragment/texture processor
  - “Multithreaded” ROP unit (Z test, etc)

Advantages of Z-buffer algorithm

- Reasonable computational cost
- Each polygon touched just once
  - Application can feed polygons in any order.
  - Works well for moving objects.
- Producer-consumer locality within HW pipeline
- Good spatial locality of memory accesses
  - Texture
  - Framebuffer
- Most parts of algorithm easily parallelized

The “tar pits” for conventional architects

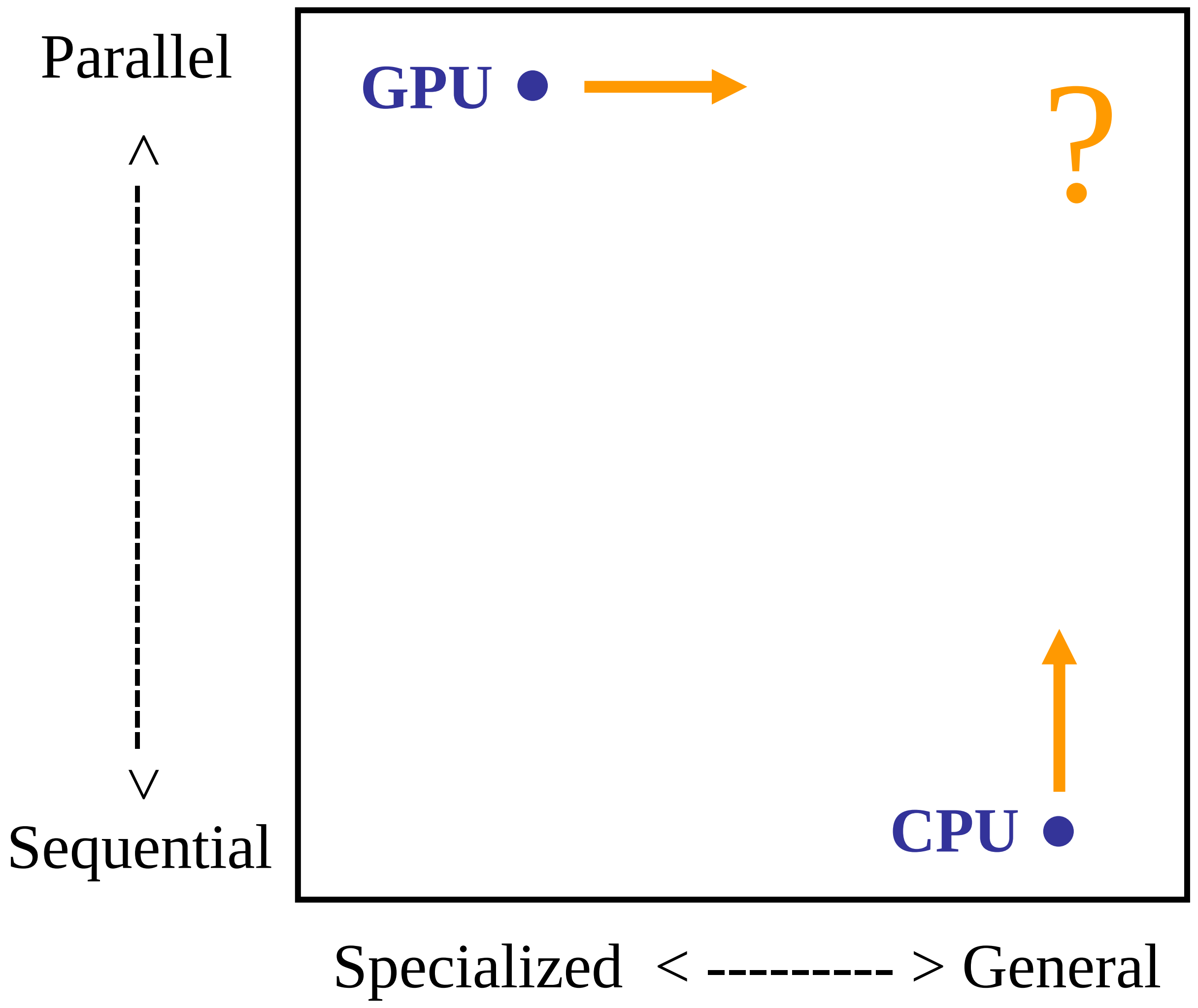
- GPUs must optimize for throughput, not latency
  - We see this trend emerging elsewhere, too.
- 3D graphics computations != signal processing
  - Surprisingly irregular and complicated
  - Especially with optimizations like compression
- Functionality changes rapidly
  - Big mistake to design for old benchmarks
  - Challenging for academics to keep up
- Specialized HW still critical to Z-buffer performance
- Architects must understand the application
  - Perhaps generally true for parallel systems?

The Future

Near term (next two years)

- Additional pipeline stages become programmable
  - E.g. Geometry subdivision/tesselation
- Additional flexibility in data flow, communication
  - Easier to implement innovative graphics algorithms
  - Easier to use GPU as “general purpose” parallel processor.
- First real successes for using GPU as “general purpose” processor
  - But limitations of stream programming model will also become apparent.

Longer term



Forces driving long-term evolution

- Desire to accelerate other computations
  - Collision detection and response, AI, etc.
  - Scene management
- Desire for more realistic images
  - Better shadows, indirect illumination, antialiasing, etc.
  - Z buffer has trouble with needed visibility computations
  - Possibilities include:
    - Enhancements to Z buffer
    - Raycasting visibility algorithms
- Work smarter, not harder
  - Trend away from brute-force, one-size-fits-all algorithms

Long-term predictions

- Graphics algorithms continue to evolve rapidly
  - End of Z-buffer as we know it
- Graphics is major driver of single-chip parallelism
  - Return to “software rendering”
  - Two parallel programming models: Streams and CSP
- One chip combines “CPU” and “GPU”
  - Fine grained throughput-optimized cores
  - Coarse grained latency-optimized cores
  - Specialized HW for certain tasks
  - Who makes it?
  - What are details of its architecture?

Game consoles as innovation platform

- Clean slate design
  - Minimal need for backward compatibility
- One company controls entire system design
  - Graphics processor
  - CPU
  - APIs and programming languages
  - Operating system
  - Application software
- But economics still discourage radical designs

Open research questions

- How should real-time graphics algorithms and architectures co-evolve?
  - What new/enhanced algorithms? What HW?
- Specialized vs. General HW?
  - What is the right balance?
  - Is semi-specialized HW useful? (e.g. R/M/W)
- What programming model for parallel units?
  - Stream, CSP, both, other?
- What granularity of parallel units?
  - Lots of little ones vs. a few big ones vs. hybrids
- Can HW for graphics also accelerate other apps?

The End

Questions?