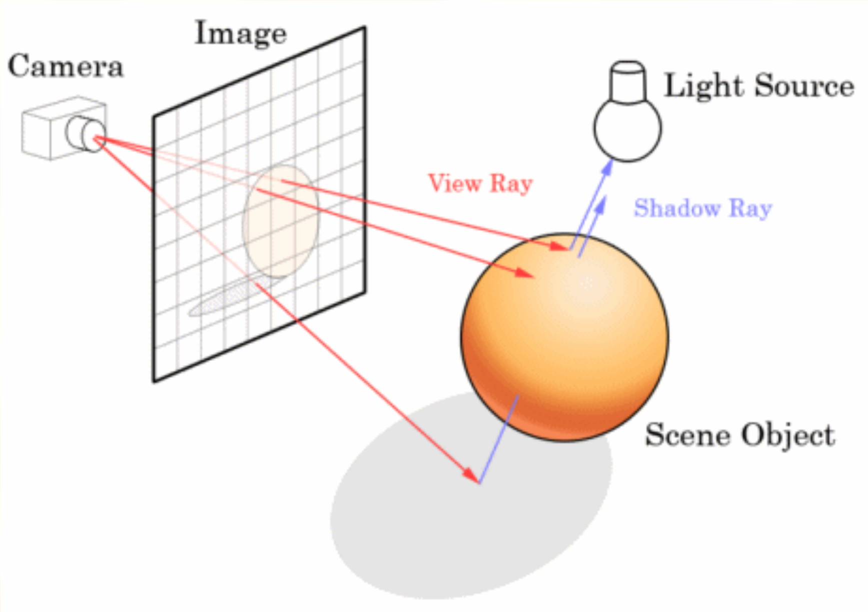
What is a GPU?
(and why you should care)

Donald S. Fussell
Department of Computer Science
The University of Texas at Austin

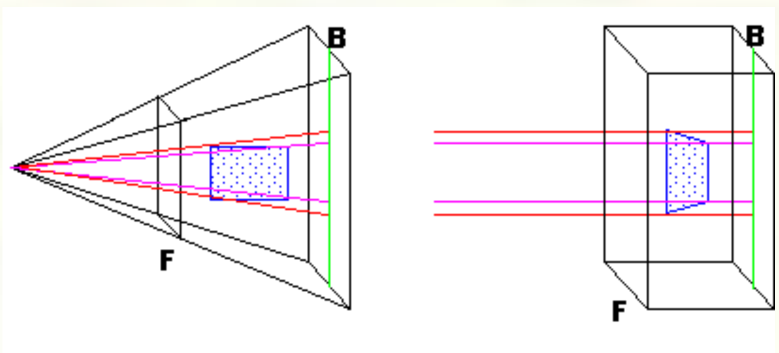
What is Rendering?

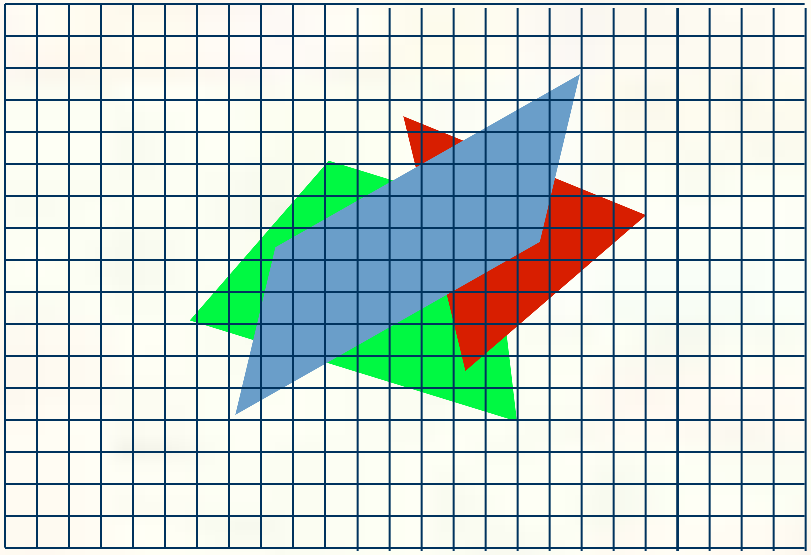
- Determining the color to be assigned to each pixel in the image by simulating the transport of light in a synthetic scene.



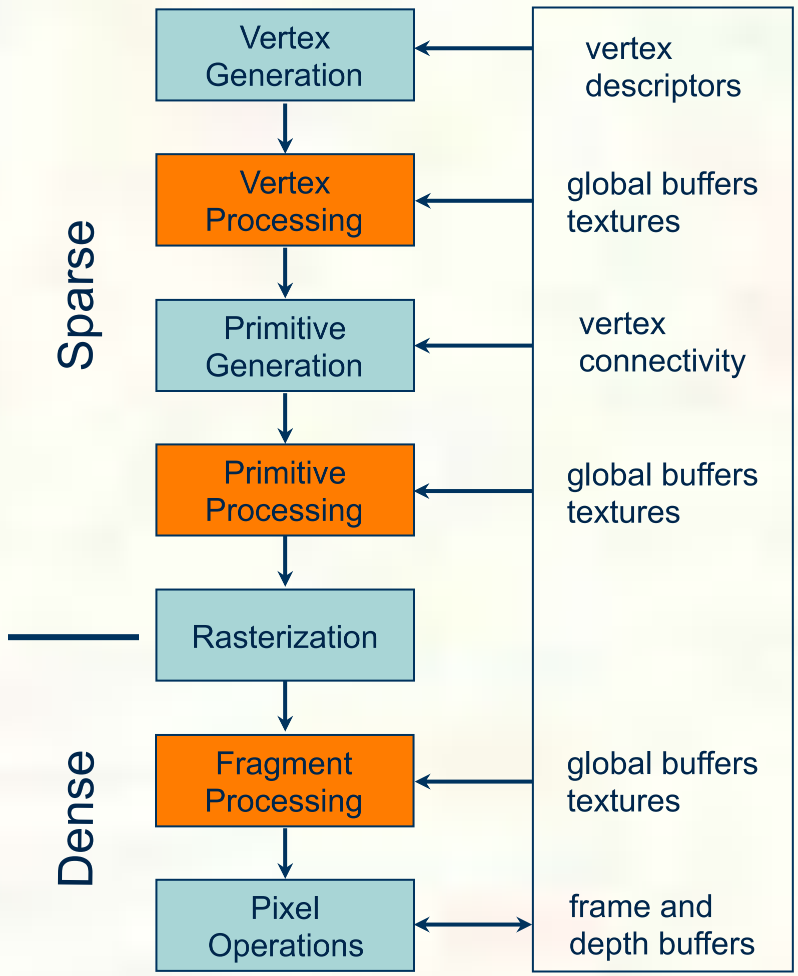
The Key Efficiency Trick

- Transform into perspective space, densely sample, and produce a large number of independent SIMD computations for shading





The Rendering Pipeline



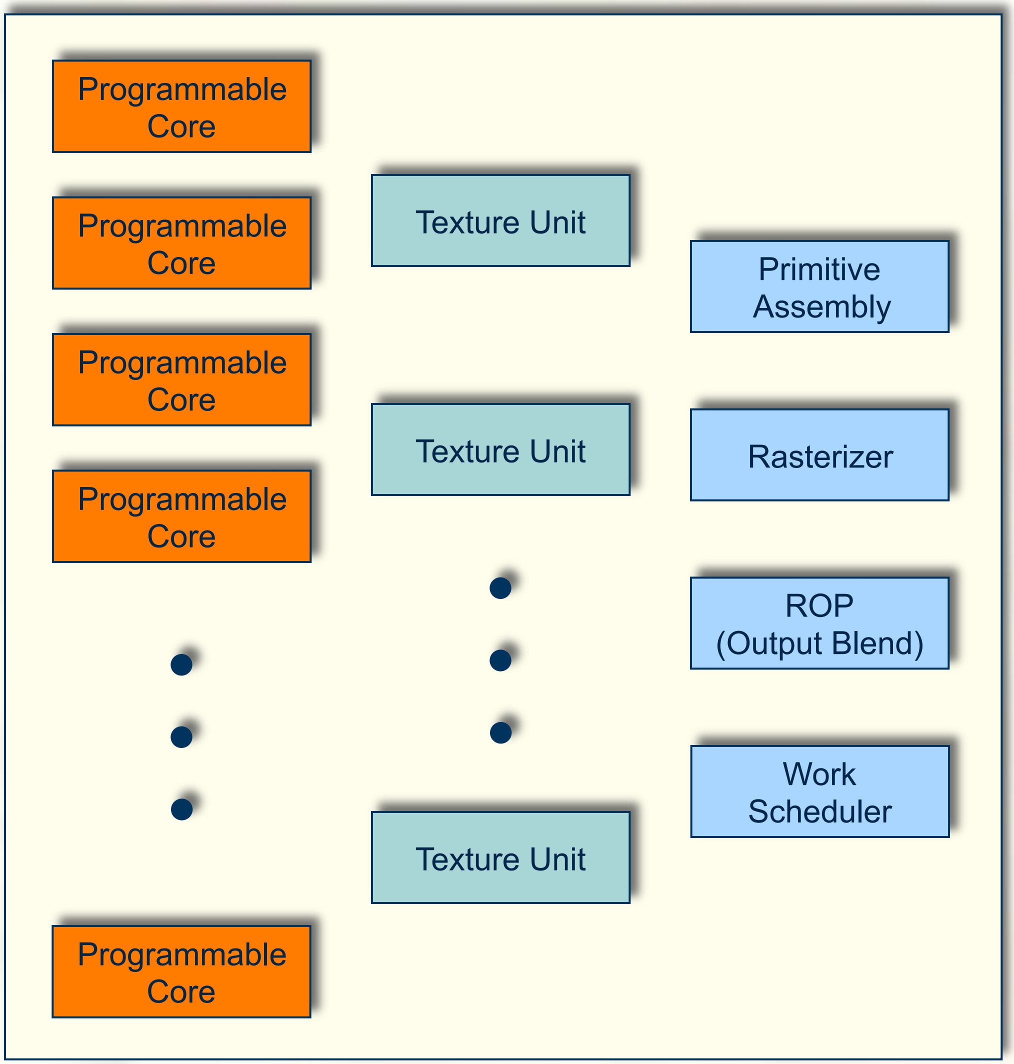
- Green - fixed function
- Orange - programmable

Evolution:

- Once all fixed function
- Then separate programmable stages
- Now homogeneous parallel system for programmable parts, software pipeline

- For coarse polygonal models about 80% of the workload is in the shading (fragment processing)

Modern GPU Characteristics



- Homogeneous programmable cores for all of the programmable stages
- Relatively few special purpose texture units
- Even fewer other types of fixed function units.

- Fixed function for non-SIMD operations
- Task parallel at the pipeline level

Shading a Fragment

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv) {
        float3 kd;
        kd = myTex.Sample(mySamp, uv);
        kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
        return float4(kd, 1.0);
}
```

compile

```
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
```

- Simple Lambertian shading of texture-mapped fragment.
- Sequential code
- Performed in parallel on a large number of independent fragments
- How many is “large number”? At least 10s of thousands per frame

Work per Fragment

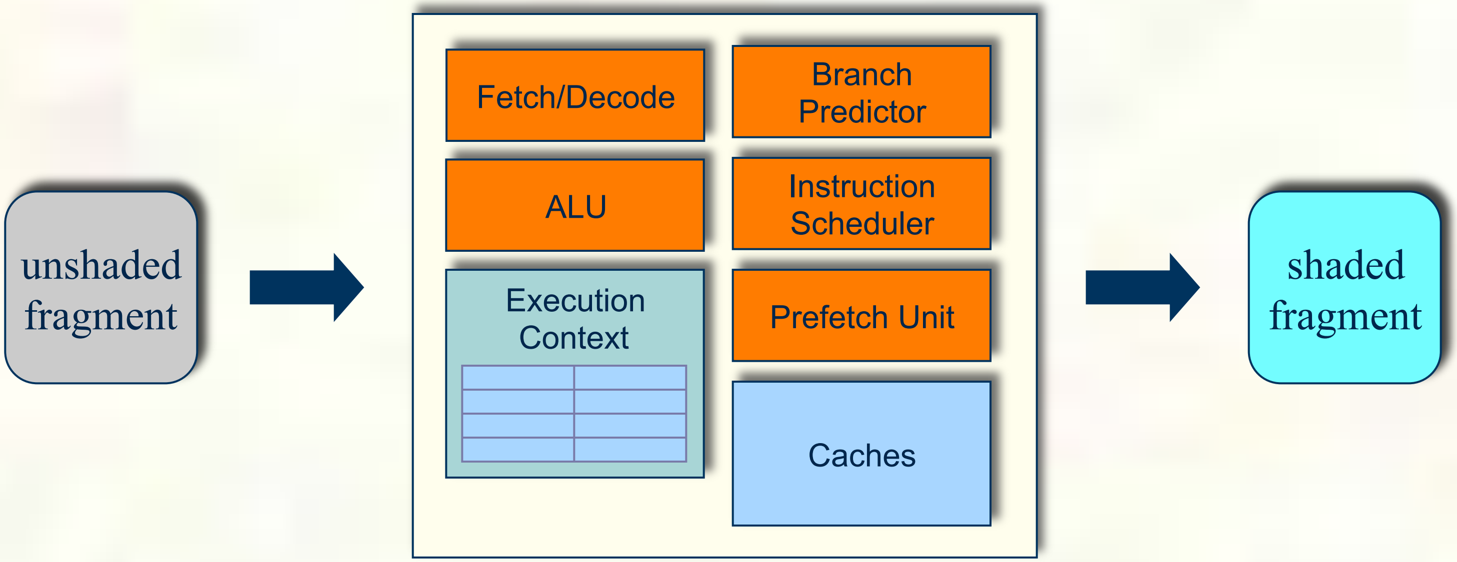
unshaded fragment →

```
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
```

→ shaded fragment

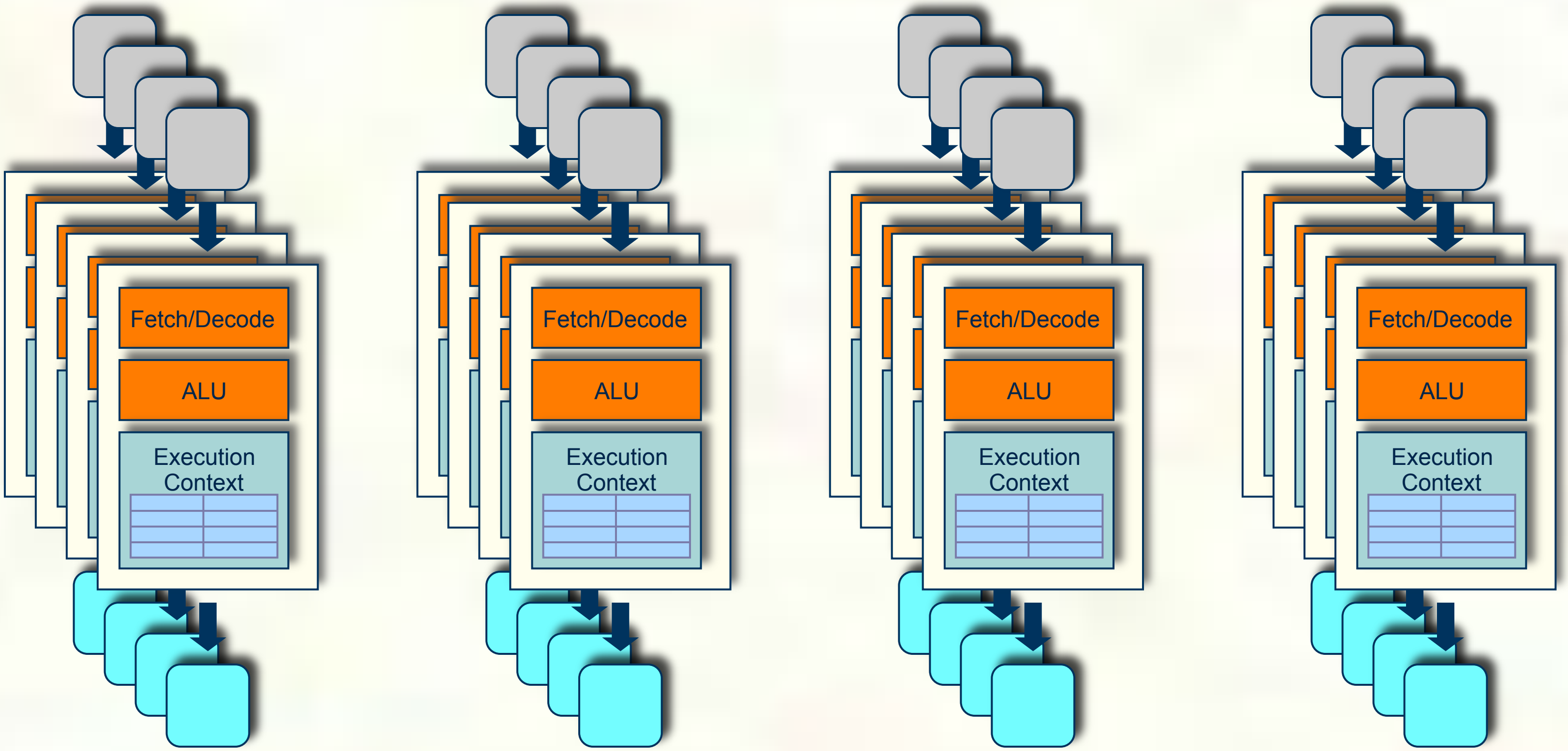
- Do a a couple hundred thousand of these @ 60 Hz or so
- How?
- Since we have independent threads to execute, use multiple cores
- What kind of cores?

The CPU Way



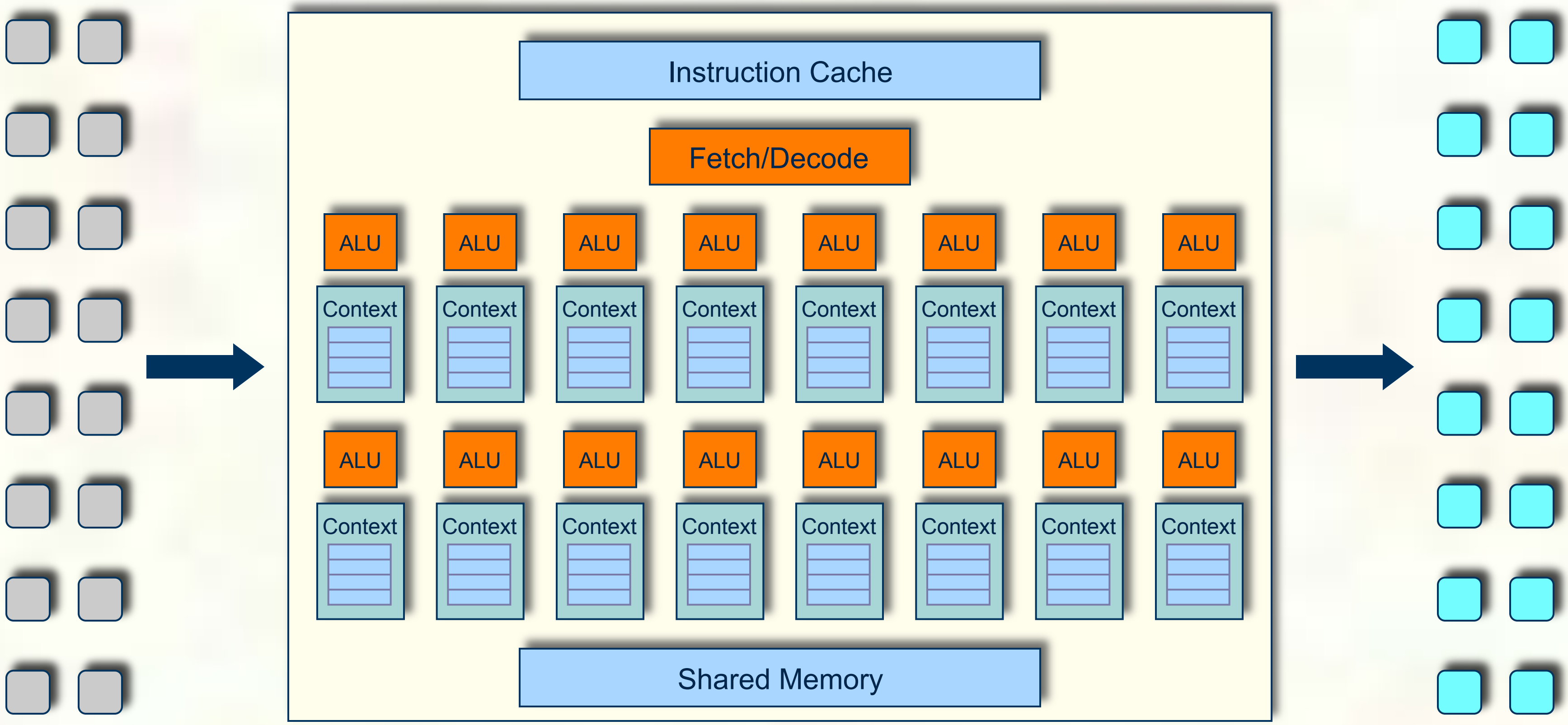
- Big, complex, but fast on a single thread
- However, if fragment shader time << frame time, we don't really care how fast the shader thread executes, we care how many of them we can do by the deadline.

Simplify and Parallelize



- Don't use a few CPU style cores, use simpler ones and more of them.

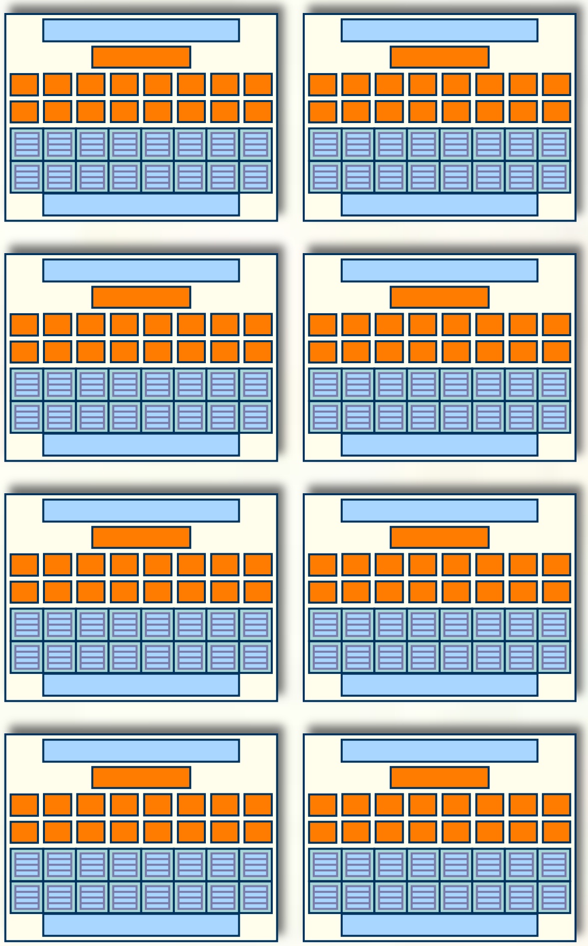
Shared Instructions



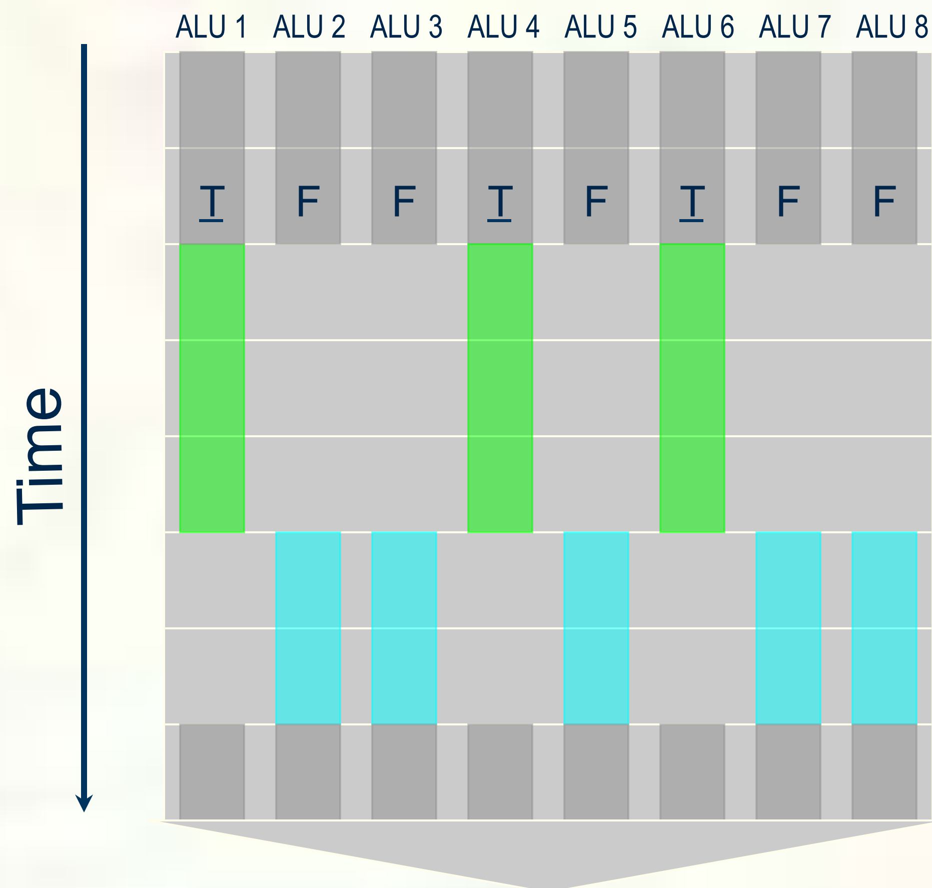
- Since we're basically doing the same thing to each fragment (or in other parts of the pipeline to vertices, primitives, etc.) in parallel, they should be able to share a single instruction stream.
- Thus SIMD - Amortize instruction handling over multiple ALUs

But What about the Other Processing?

- A graphics pipeline does more than shading. We have other places where we do different things in parallel, like transforming vertices for example. So we will need to be executing more than 1 program in the system.
- If we replicate these SIMD processors, we now have the ability to do different SIMD computations in parallel in different parts of the machine.
- In this example, we can have 128 threads in parallel, but only 8 different programs simultaneously running



What about Branches?

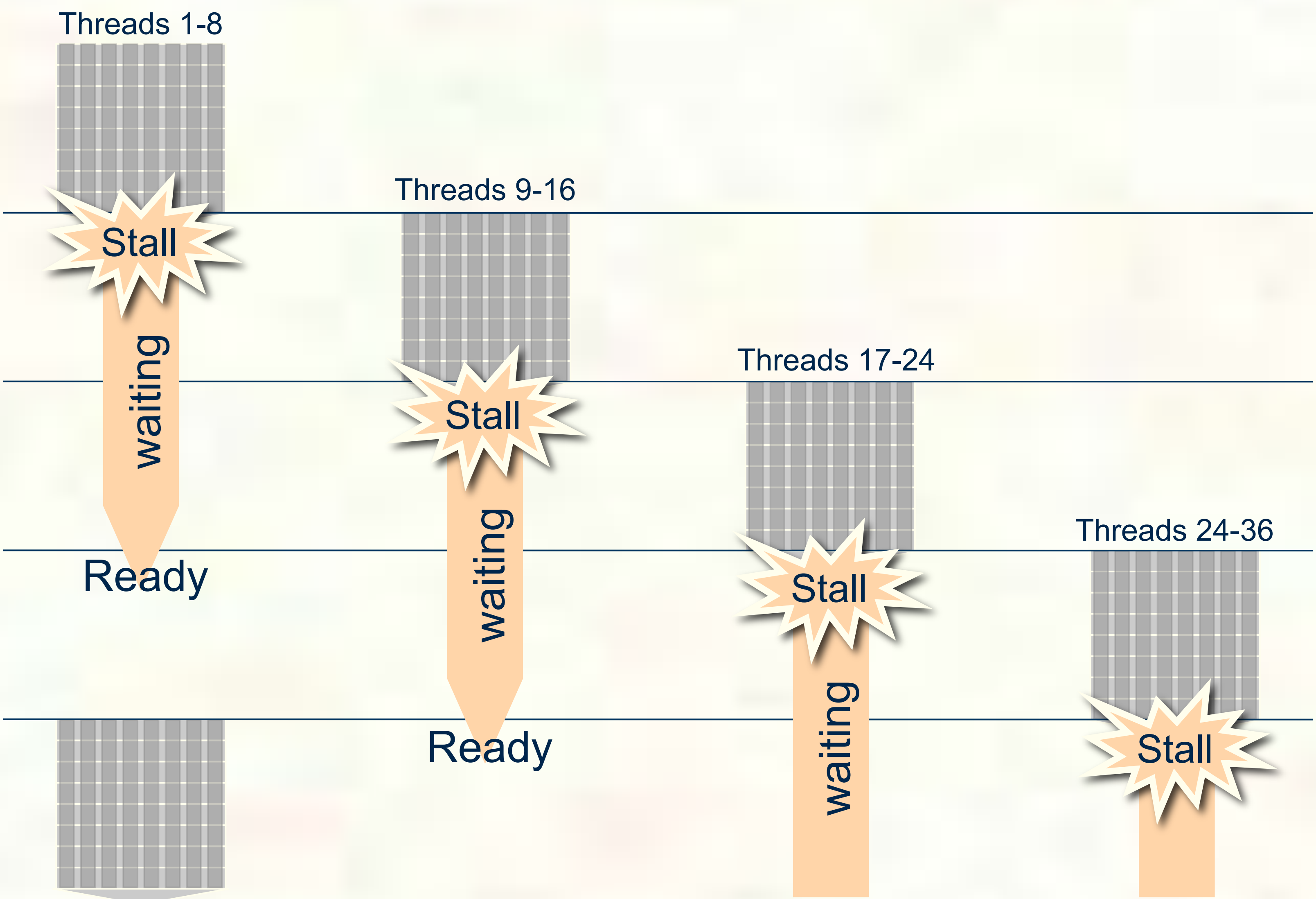



```
<unconditional shader code>
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<unconditional shader code>
```

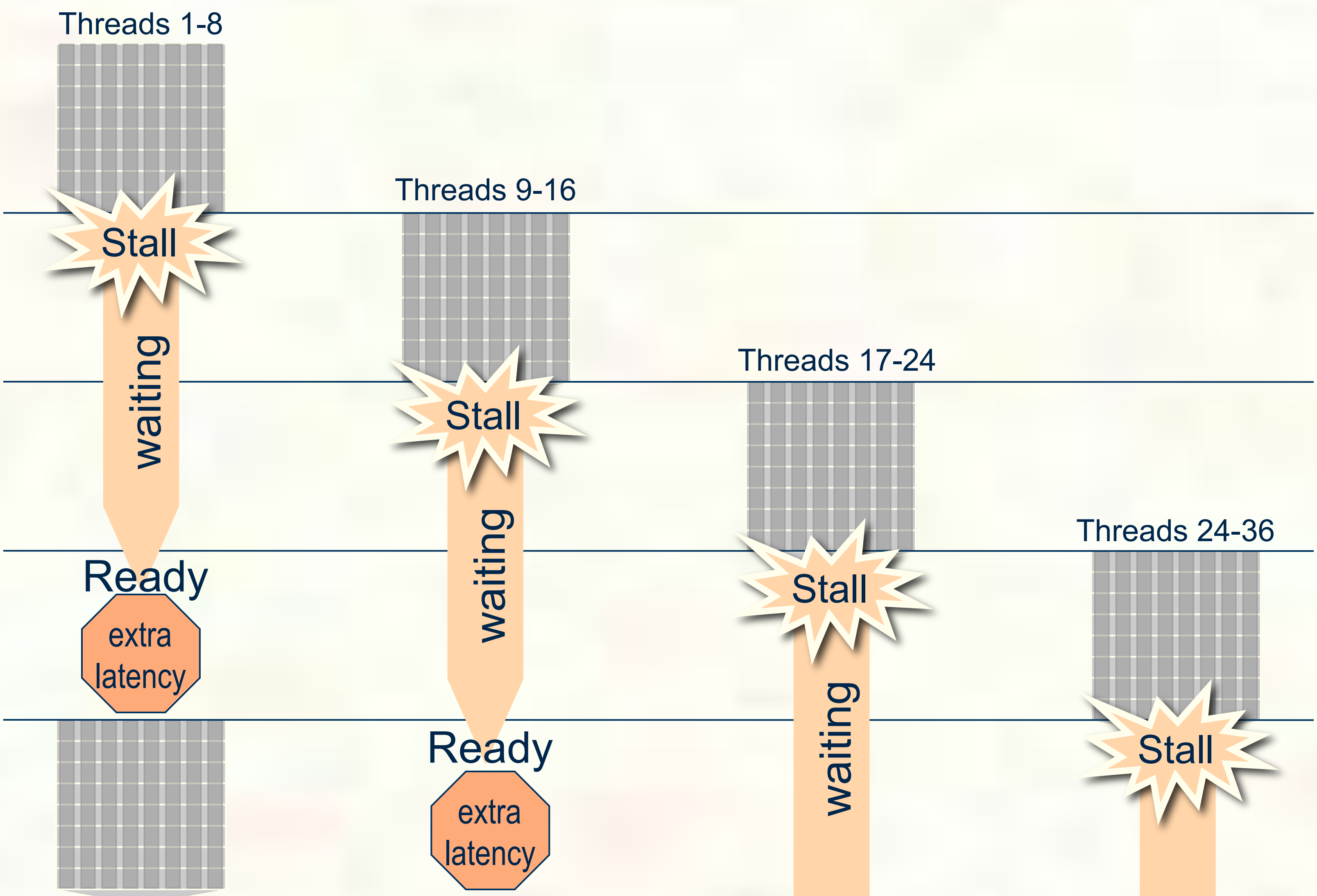
Efficiency - Dealing with Stalls

- A thread is stalled when its next instruction to be executed must await a result from a previous instruction.
  - Pipeline dependencies
  - Memory latency
- The complex CPU hardware omitted from these machines was effective at dealing with stalls.
- What will we do instead?
- Since we expect to have lots more threads than processors, we can interleave their execution to keep the hardware busy when a thread stalls.
- Multithreading!

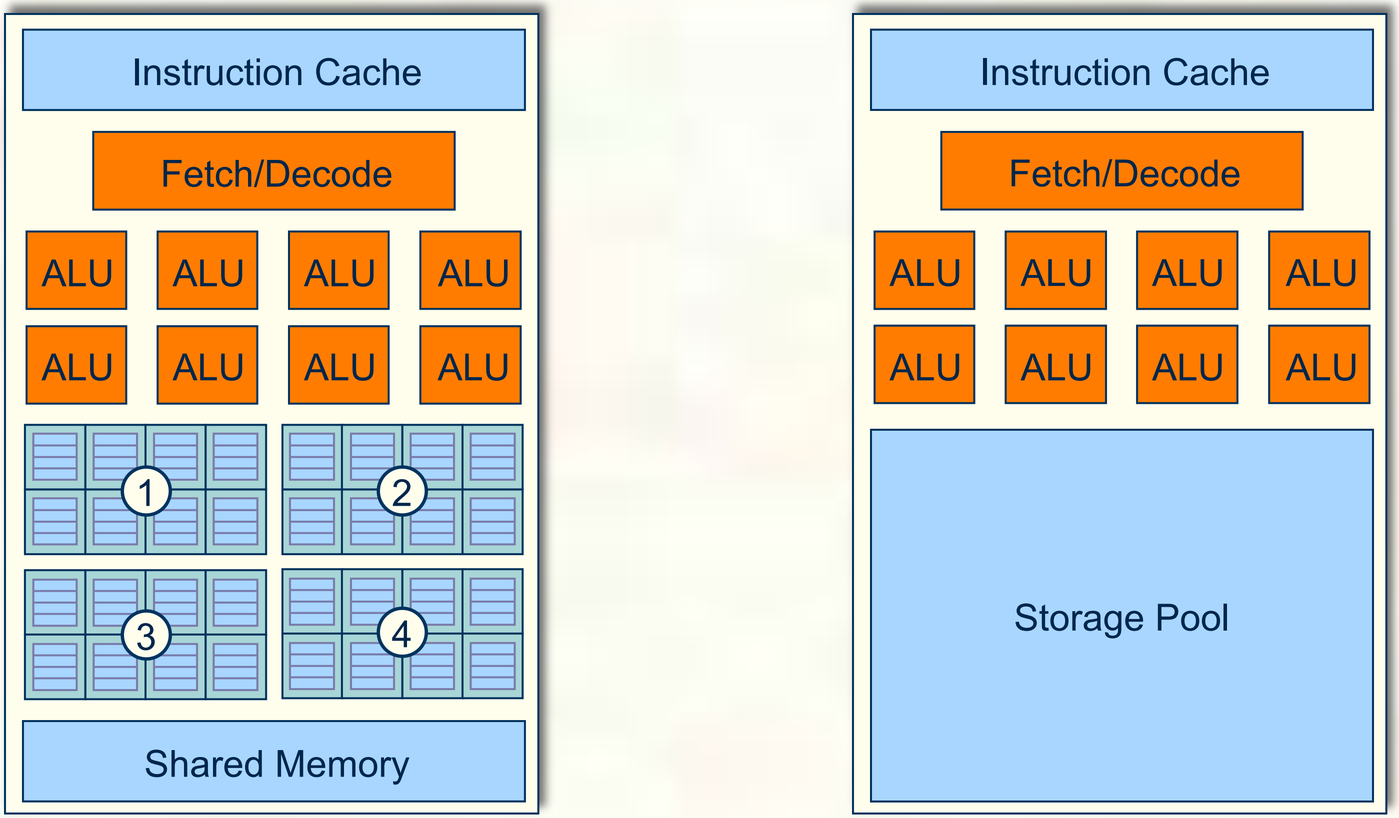
Multithreading



Multithreading

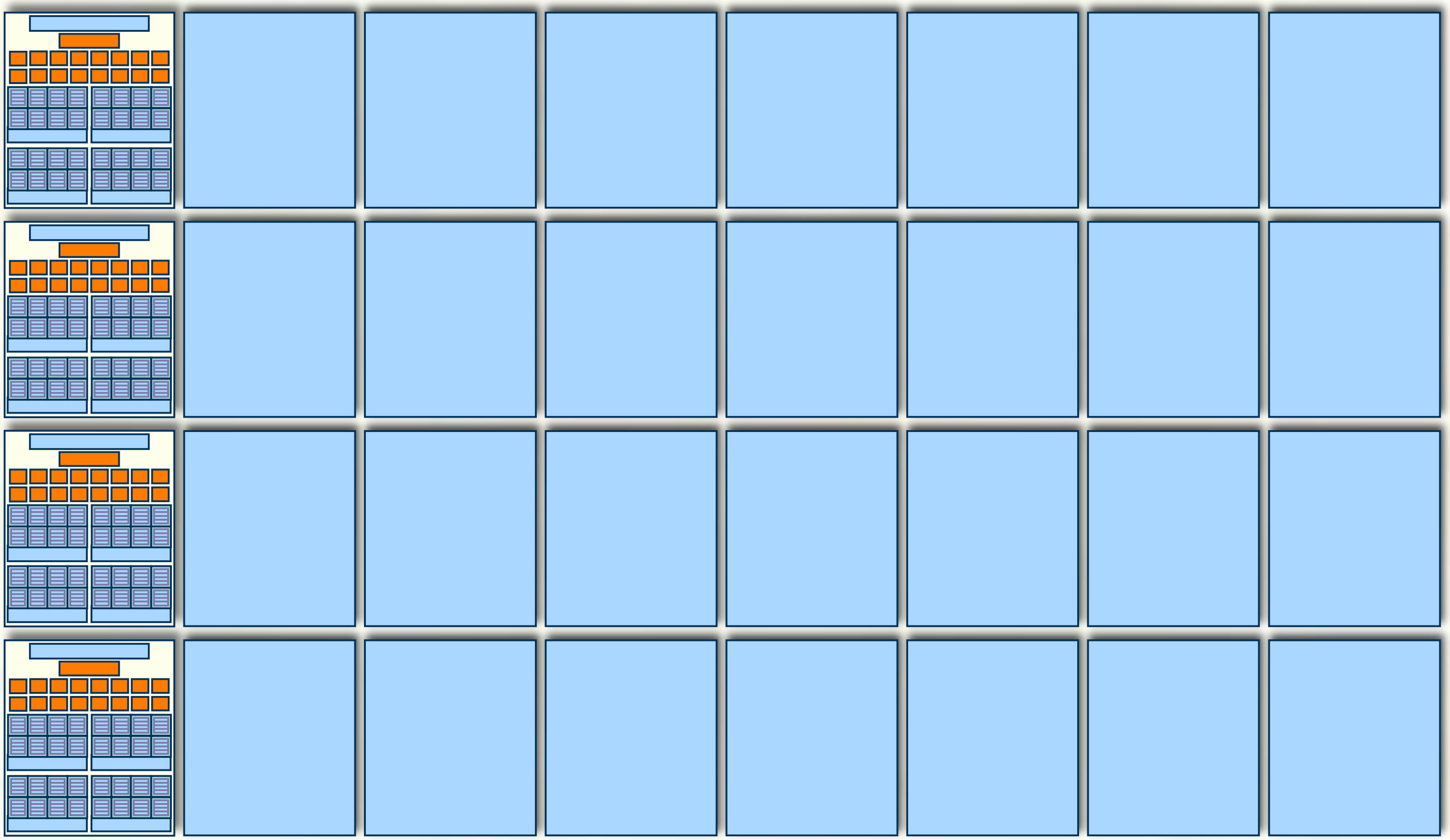


Costs of Multithreading



- Adds latency to individual threads in order to minimize time to complete all threads.
- Requires extra context storage. More contexts can mask more latency.

Example System



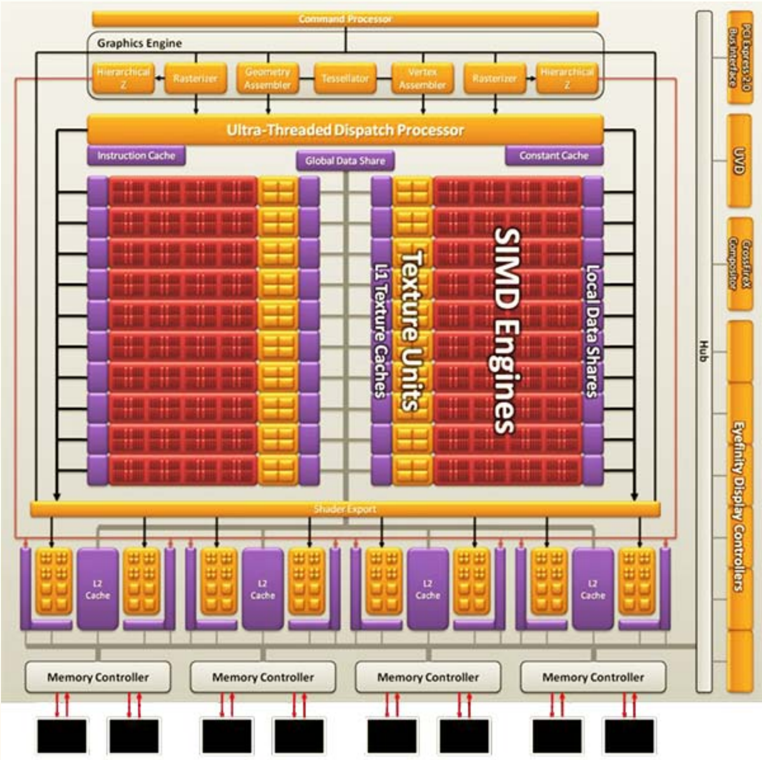
32 cores x 16 ALUs/core = 512 (madd) ALUs @ 1 GHz = 1 Teraflop

Real Example - NVIDIA GeForce GTX 285

- 30 Cores
- 8 SIMD Functional Units per Core
- Each FU has 1 multiplier and 1 madder
- Peak 720 floating point ops per clock
- 2 level multithreading
  - Fine-grained: 4 threads interleaved into pipelined FUs
  - Thus up to 32 threads concurrently executing (called a "WARP")
  - Coarse-grained: Up to 32 WARPS interleaved per core to mask latency to memory

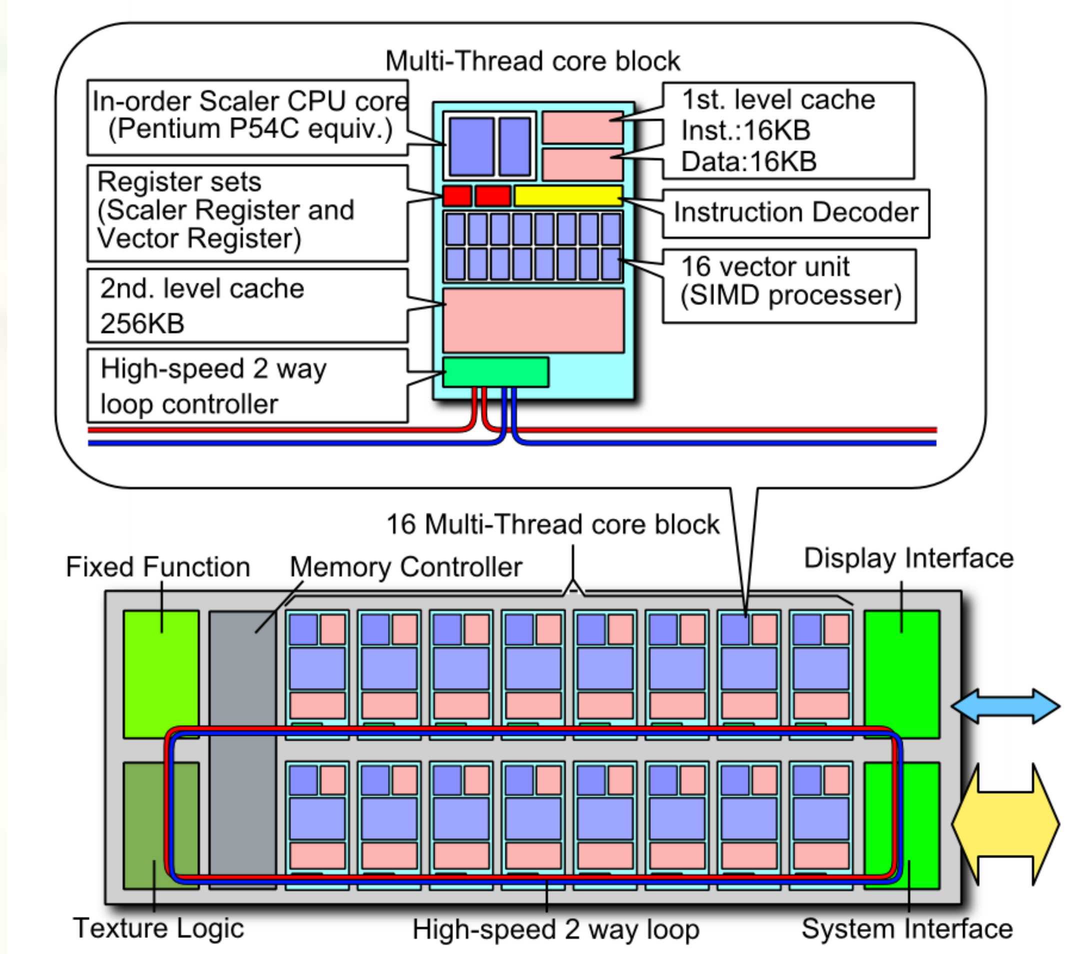
Real Example - AMD Radeon HD 4890

- 10 Cores
- 16 SIMD Functional Units per Core
- 5 madders per FU
- Peak 1600 floating point ops per clock
- 2 level multithreading
  - Fine-grained: 4 threads interleaved into pipelined FUs
  - Up to 64 concurrent threads (not called a “WARP”)
  - Coarse-grained: groups of 64 threads interleaved to mask memory latency



"Real" Example - Intel Larrabee

- Some number of cores
- Explicit 16-wide vector ISA (16-wide madder unit)
- Peak 32*n* floating point operations per clock for *n* cores
- Each core interleaves 4 x86 instruction streams
- Additional interleaving under software control



Memory architecture

- CPU style
  - Multiple levels of cache on chip
  - Takes advantage of temporal and spatial locality to reduce demand on remote slow DRAM
  - Provides local high bandwidth to cores on chip
  - 25GB/sec to main memory

GPU-style memory architecture

- Local execution contexts (64kB) and a similar amount of local memory
- Read-only texture cache
- Traditionally no cache hierarchy (but see NVIDIA Fermi and Larrabee)
- Much higher bandwidth to main memory - 150 GB/sec

Bandwidth is critical for throughput

- So GPU memory system is designed for throughput
  - Wide Bus (150 GB/sec)
  - Likewise high bandwidth DRAM organization (GDDR3-5)
  - Careful scheduling of memory requests to make efficient use of available bandwidth

Graphics applications and GPUs

- If an NVIDIA GTX 285 has a 1.5 GHz clock (for the arithmetic units) and 720 floating point ops per clock, we have 1080 Gflops peak compute
- If we have 150 GB/sec memory bandwidth, then at peak efficiency our application has to be doing at least 6 flops per byte transferred
- For AMD Radeon HD 4890 at 1 GHz, the arithmetic intensity needs to be about 10 rather than 6
- Many graphics workloads do this much math, but not all of them

Rendering applications

- Transforms
  - 4 element matrix vector multiply - matrix locally resident for many vertices
  - Fetch 3 32-bit coordinates per vertex - 12 bytes
  - Perform 4 multiplications and 4 additions per coordinate
  - That's 12 madds and 12 bytes fetched, a ratio of 1 madd per byte
  - Or, for wide SIMD, it's 4 madds and 12 bytes for a .33 ratio
  - Fortunately, this is a small part of the workload
  - Also fortunately, this has a regular memory access pattern, so can be prefetched, etc.
- DRAM bandwidth is the limiting factor for most application designers!!

Trends

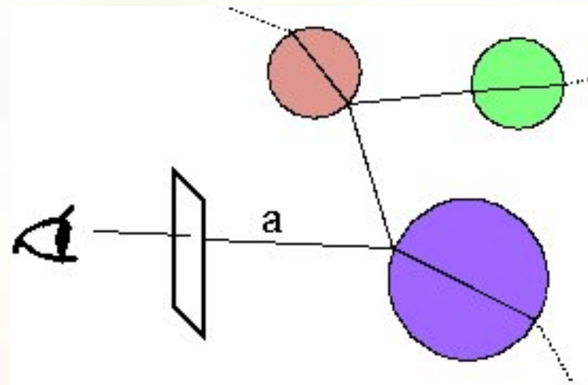
- Higher rendering quality
  - Micropolygons a la Pixar
  - Ray tracing and irregular computations
  - Both put more pressure on system, irregular computation, lower arithmetic intensity (1 sample per fragment)
- Games - PC and Console
  - Games aren't just renderers - they have various types of physics simulations, character animation, AI, networking, sound, etc. All has to work against real-time deadlines.
  - So, games overall are a throughput application, but multiple tasks, each multithreded
  - Shouldn't most of this leverage the high performance part of the system - the GPU?
  - So, more heterogeneous apps sharing GPU resources.

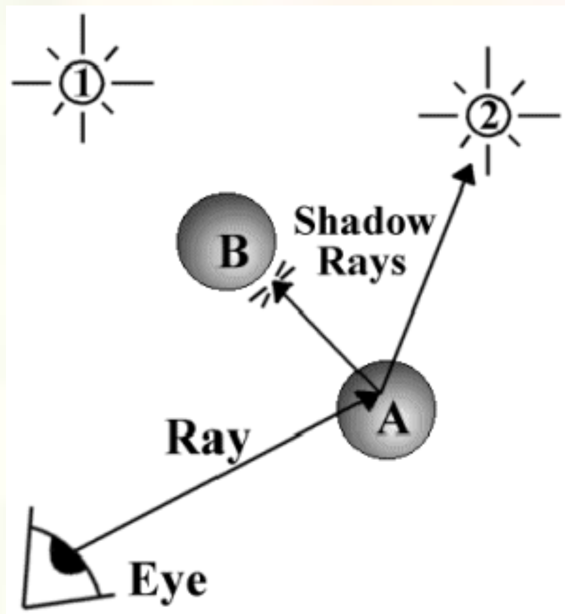
Trends

- Flexibility
  - Larrabee has less hardware control than NVIDIA/AMD
    - Scheduling flexibility makes programming more difficult, but ameliorates issues with builtin schedulers
- Local cache hierarchy
  - Larrabee has a traditional cache hierarchy
  - Fermi has more local memory that can be configured as either cache or local memory or both
- Software vs. hardware control?
  - Software scheduling?
  - Software rasterizing?
- Continuing pressure on memory bandwidth
  - Radeon HD 5870 has twice the peak computation rate of the HD 4890 (2.7 Tflops) and still 150 GB/sec memory bandwidth

Ray tracing

- Most flexible technique for global illumination
- Primary (and shadow) rays regular (common origin)
- Other secondary rays are a real challenge

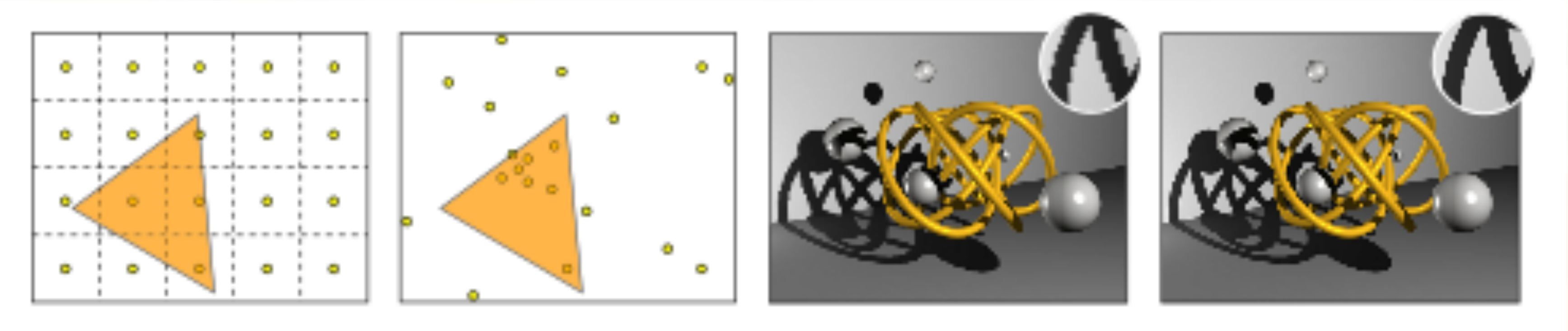




Ray tracing

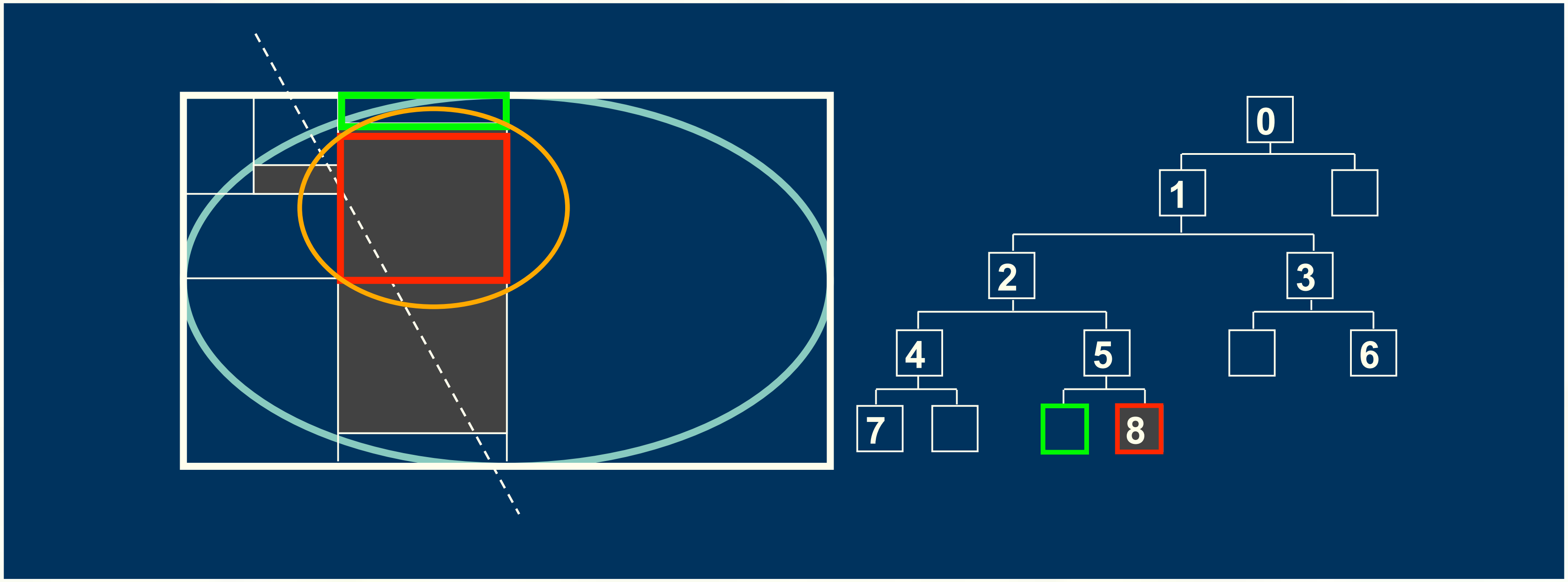
- Lots of light bounces (specular here, actually easier than diffuse)
- Shadows can be done well

Shadows and irregular sampling



- Ray tracing does this naturally
- Rasterization can be modified to do it, but need data structures that aren't just uniform grids

Data structures



- Hierarchical data structures (e.g. k-d tree)
- Must be built and traversed
- For ray tracing, scaling rasterization, irregular z-buffer

Ray Tracing

- Ray Tracing 1
  - Basic algorithm
  - Overview of pbrt
  - Ray-surface intersection (triangles, …)
- Ray Tracing 2
  - Brute force: $|I| \times |O|$
  - Acceleration data structures

Ray Tracing Acceleration Techniques

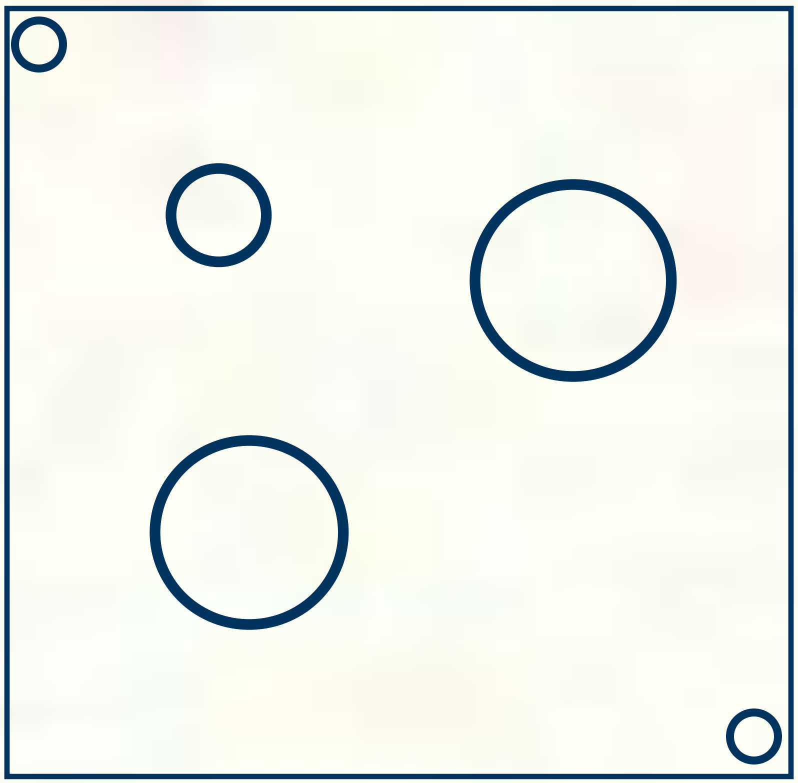
Approaches

- **Faster Intersection**
  - N
    - Uniform grids
    - Spatial hierarchies
      - k-d, oct-tree, bsp
      - hierarchical grids
    - Hierarchical bounding volumes (HBV)
  - 1
    - Tighter bounds
    - Faster intersector
- **Fewer Rays**
  - Early ray termination
  - Adaptive sampling
- **Generalized Rays**
  - Beam tracing
  - Cone tracing
  - Pencil tracing

Primitives

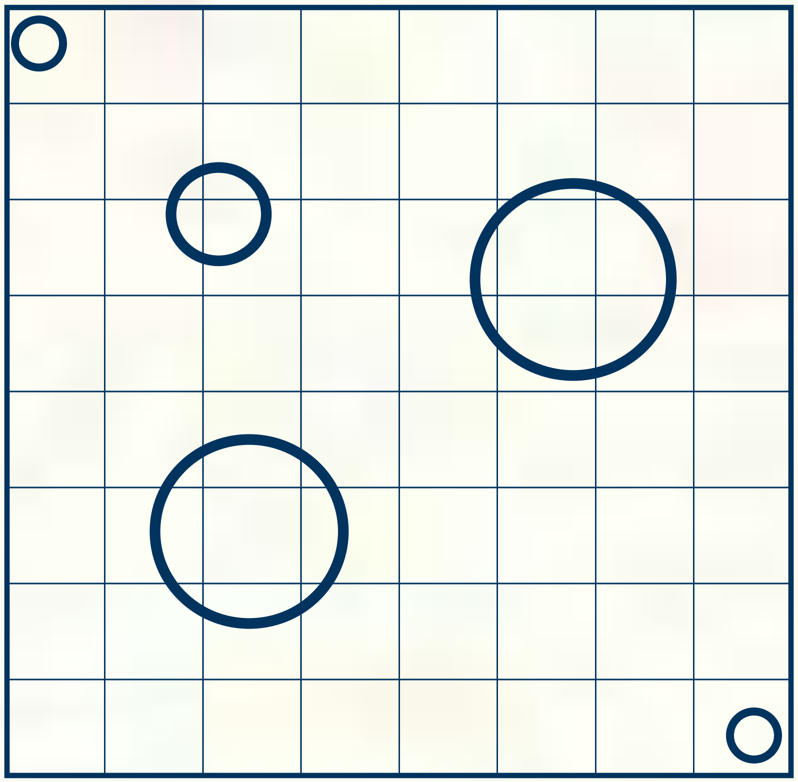
- pbrt primitive base class
  - Shape
  - Material and emission (area light)
- Primitives
  - Basic geometric primitive
  - Primitive instance
    - Transformation and pointer to basic primitive
  - Aggregate (collection)
    - Treat collections just like basic primitives
    - Incorporate acceleration structures into collections
    - May nest accelerators of different types
    - Types: grid.cpp and kdtree.cpp

Uniform Grids



- Preprocess scene
  - Find bounding box

Uniform Grids

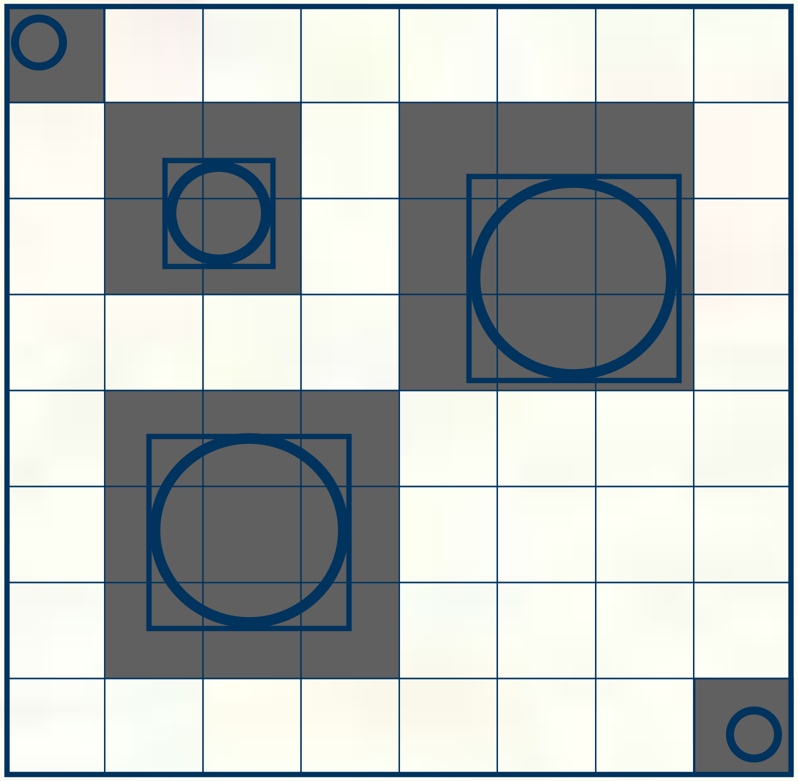


- Preprocess scene
  - Find bounding box
  - Determine resolution

$$n_v = n_x n_y n_z \propto n_o$$

$$\max(n_x, n_y, n_z) = d\sqrt[3]{n_o}$$

Uniform Grids

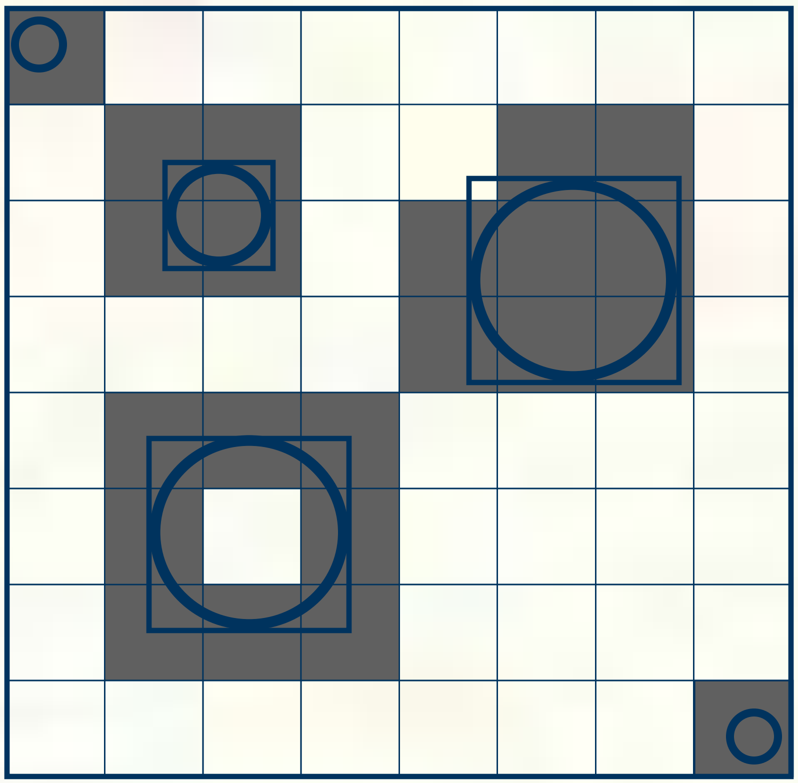


- Preprocess scene
  - Find bounding box
  - Determine resolution

  - Place object in cell, if object overlaps cell

$$\max(n_x, n_y, n_z) = d\sqrt[3]{n_o}$$

Uniform Grids

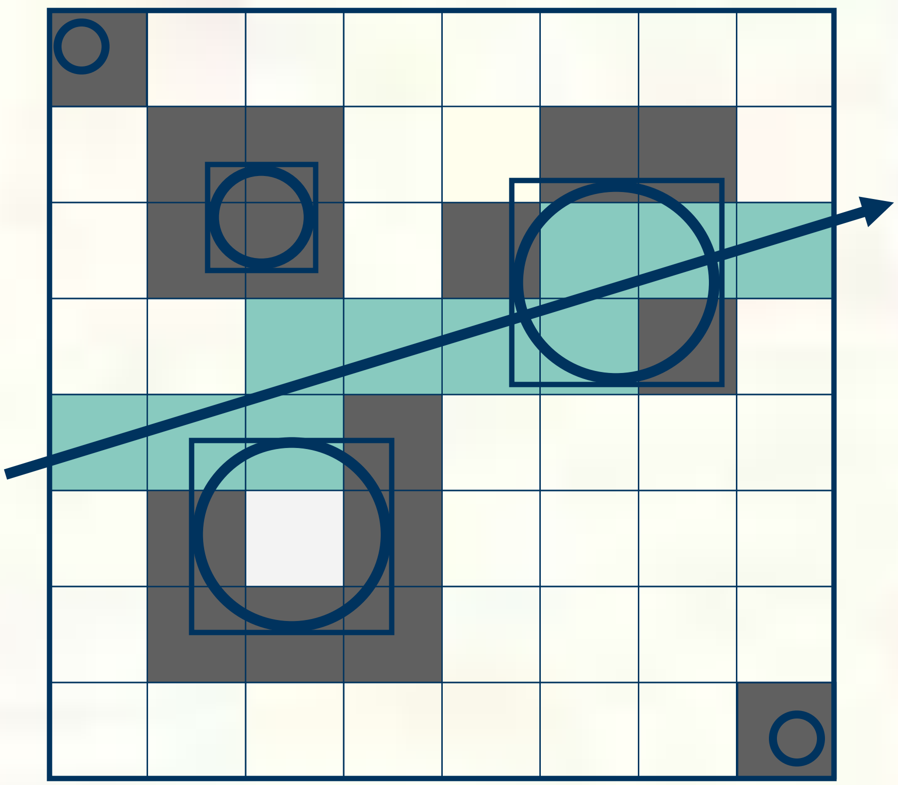


- Preprocess scene
  - Find bounding box
  - Determine resolution

$$\max(n_x, n_y, n_z) = d\sqrt[3]{n_o}$$

  - Place object in cell, if object overlaps cell
  - Check that object intersects cell

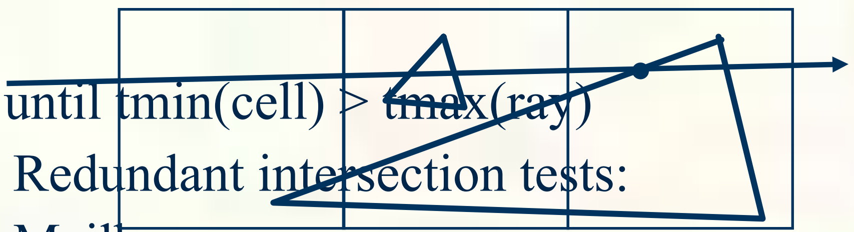
Uniform Grids



- Preprocess scene
- Traverse grid
  - 3D line – 3D-DDA
  - 6-connected line
- Section 4.3

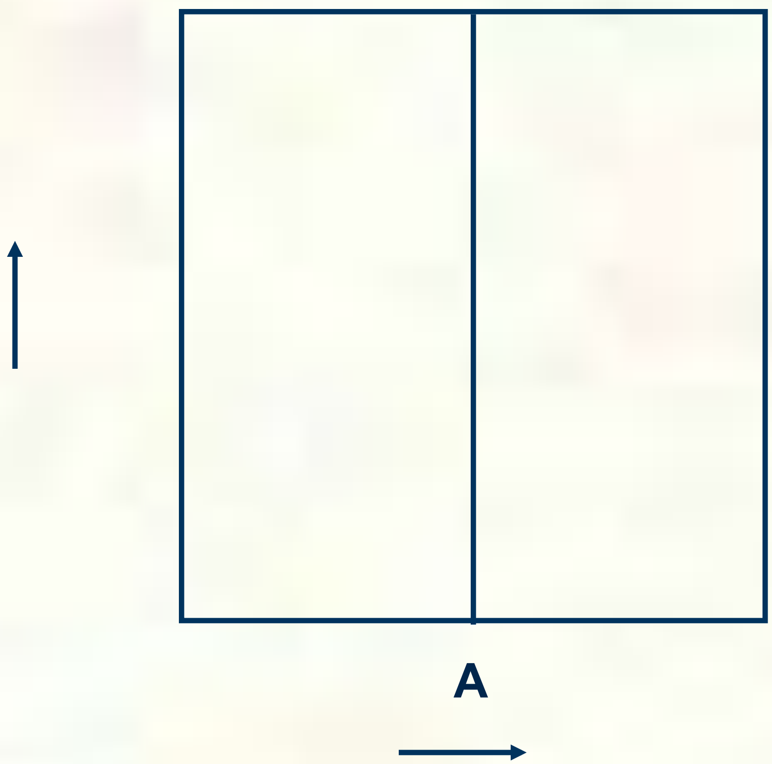
Caveat: Overlap

- *Optimize for objects that overlap multiple cells*



- Traverse until tmin(cell) > tmax(ray)
- Problem: Redundant intersection tests:
- Solution: Mailboxes
  - Assign each ray an increasing number
  - Primitive intersection cache (mailbox)
    - Store last ray number tested in mailbox
    - Only intersect if ray number is greater

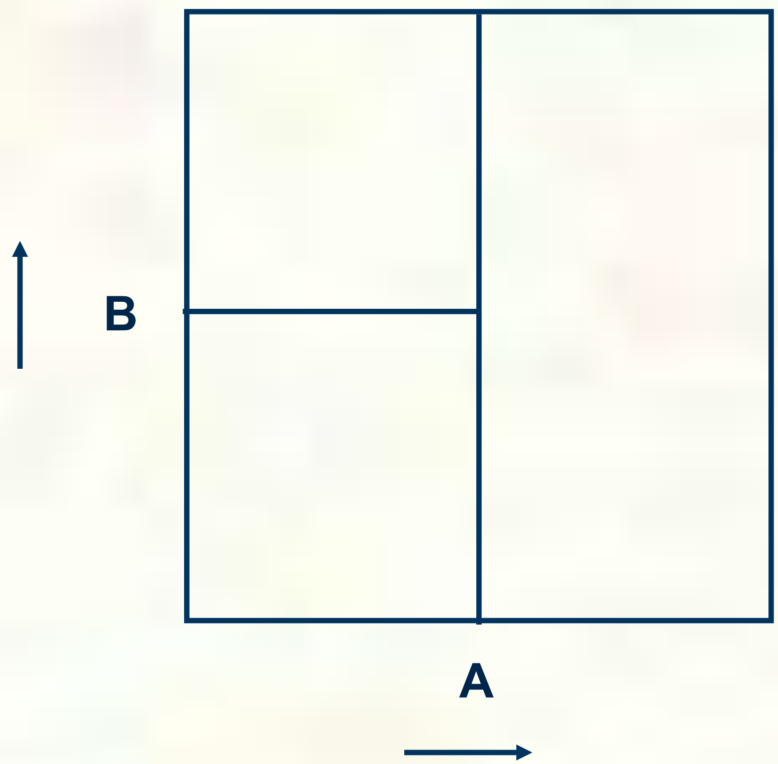
Spatial Hierarchies

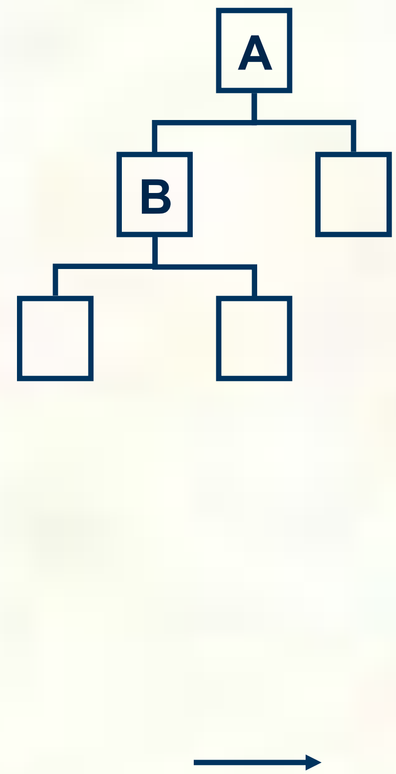


Letters correspond to planes (A)

Point Location by recursive search

Spatial Hierarchies

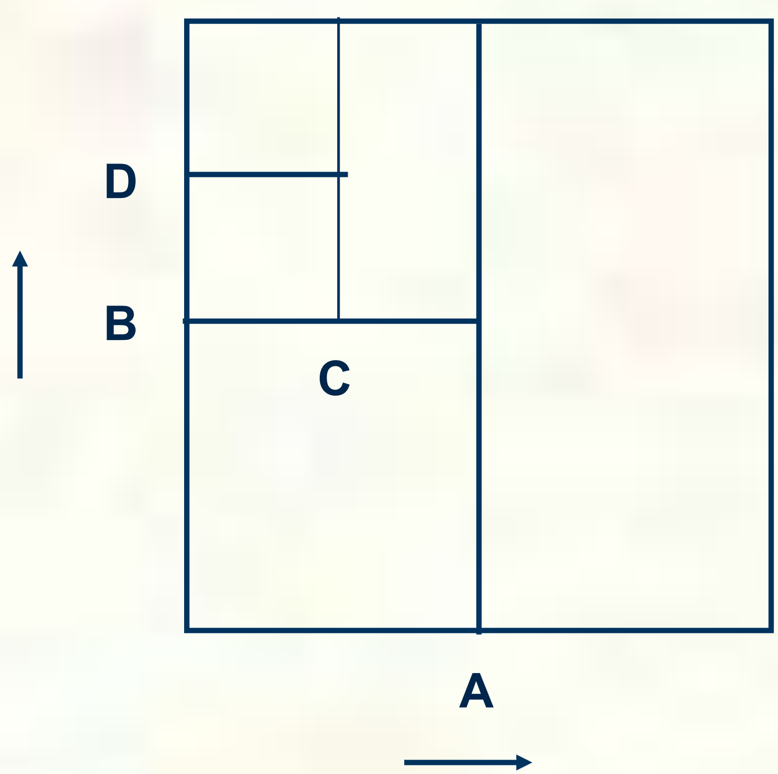


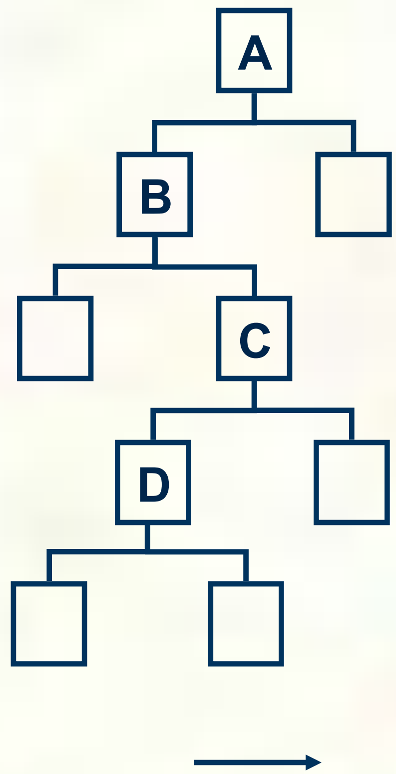


Letters correspond to planes (A, B)

Point Location by recursive search

Spatial Hierarchies

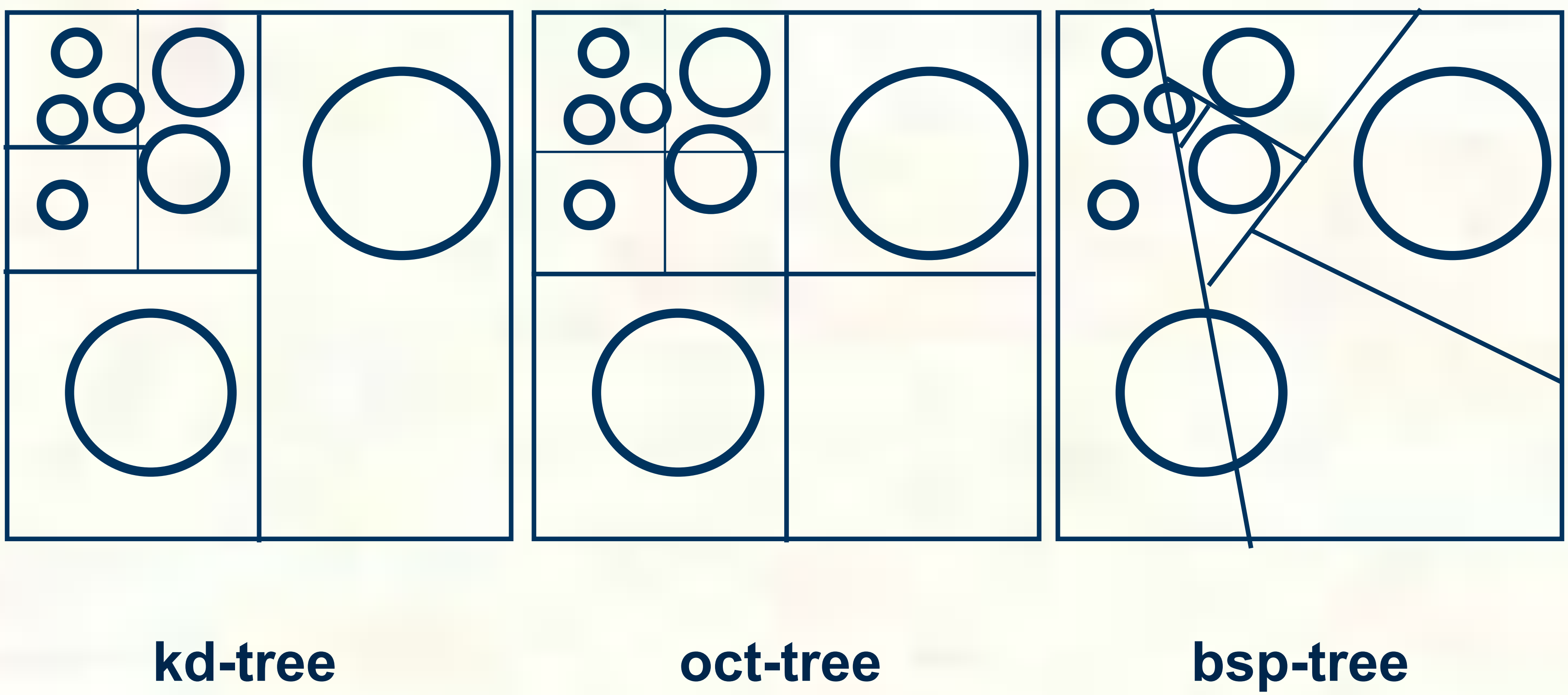




Letters correspond to planes (A, B, C, D)

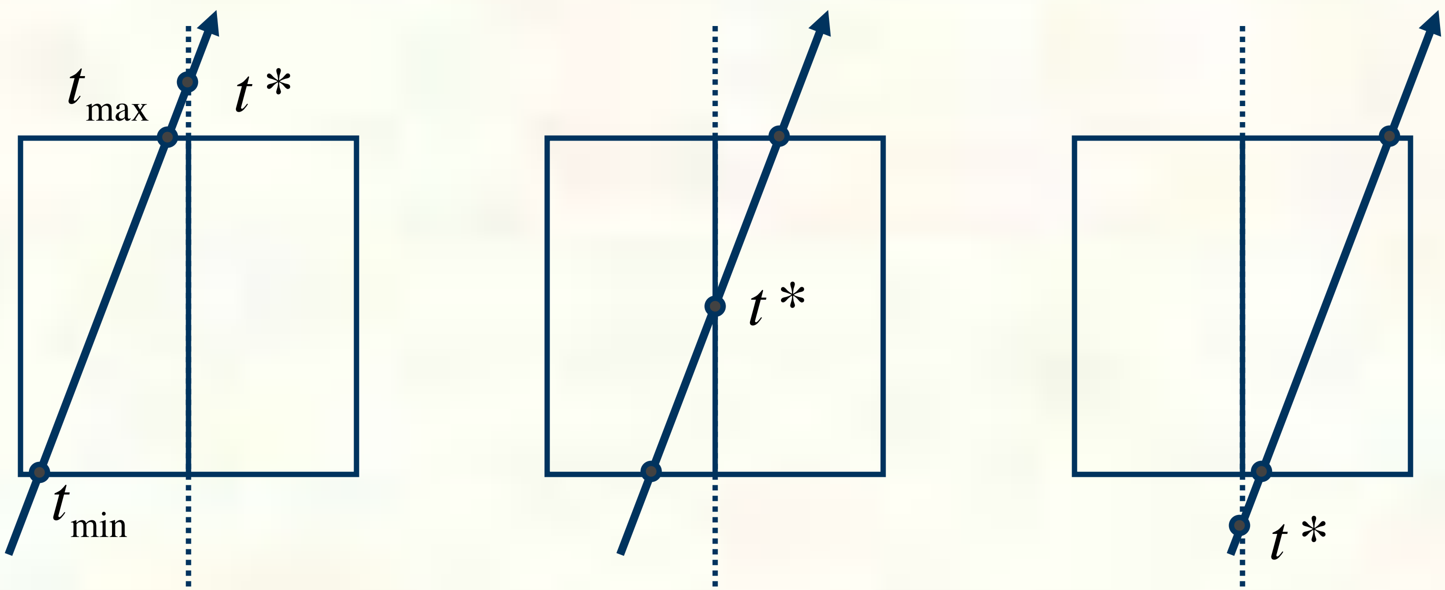
Point Location by recursive search

Variations



Ray Traversal Algorithms

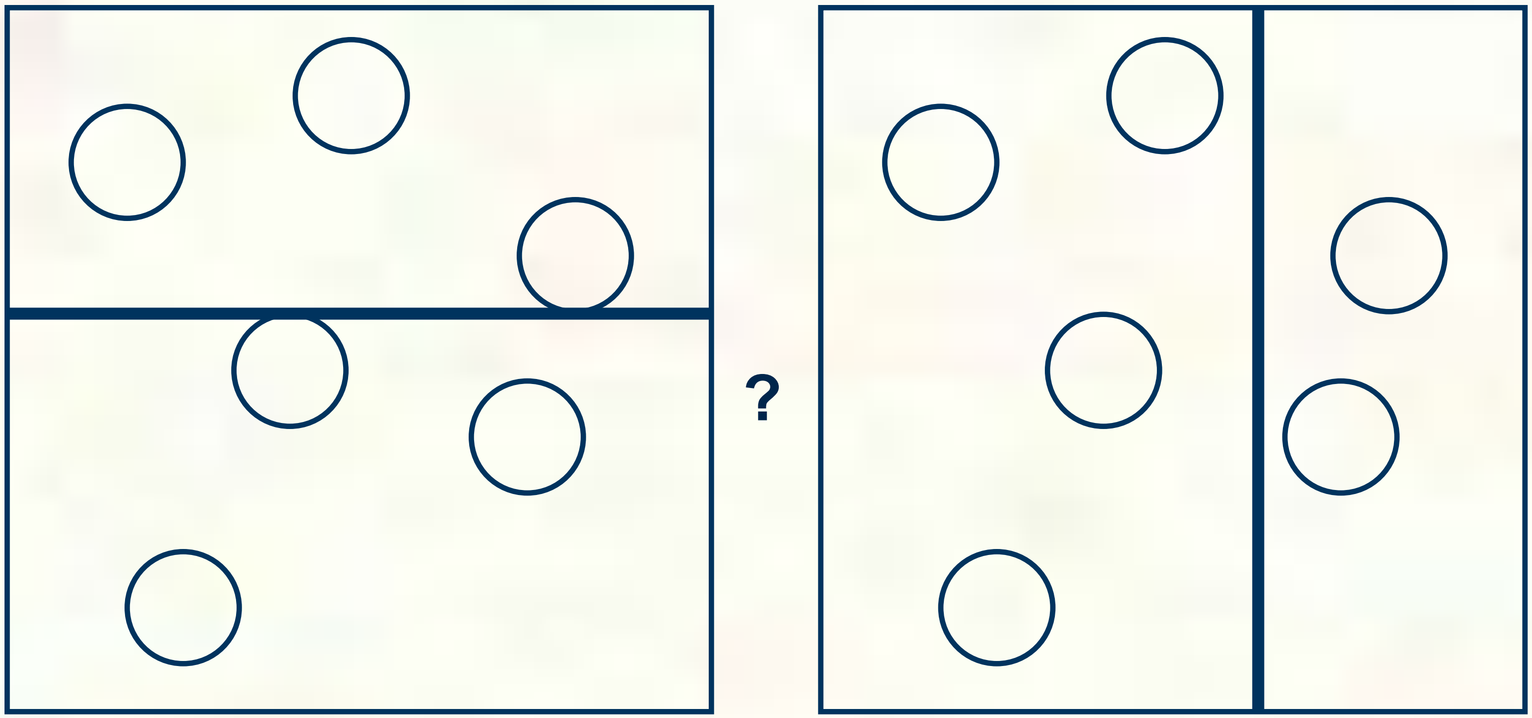
- Recursive inorder traversal
- [Kaplan, Arvo, Jansen]



$t_{max} < t^*$ $\quad$ $t_{min} < t^* < t_{max}$ $\quad$ $t^* < t_{min}$

```
Intersect(L,tmin,tmax)  Intersect(L,tmin,t*)  Intersect(R,tmin,tmax)
                        Intersect(R,t*,tmax)
```

Build Hierarchy Top-Down

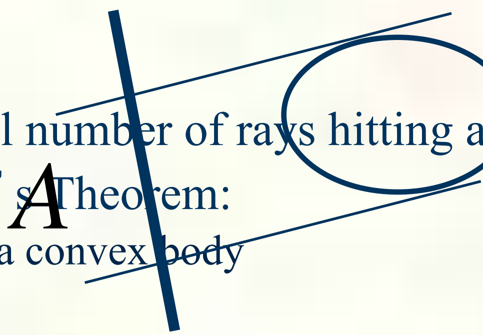


Choose splitting plane

- **Midpoint**
- **Median cut**
- **Surface area heuristic**

Surface Area and Rays

- Number of rays in a given direction that hit an
- object is proportional to its projected area



- The total number of rays hitting an object is $4\pi \bar{A}$
- Crofton's Theorem:
  - For a convex body

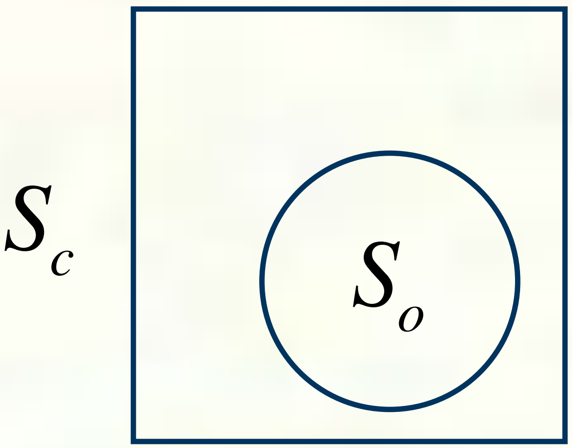
$$\bar{A} = \frac{S}{4}$$

- For example: sphere

$$S = 4\pi r^2 \quad \bar{A} = A = \pi r^2$$

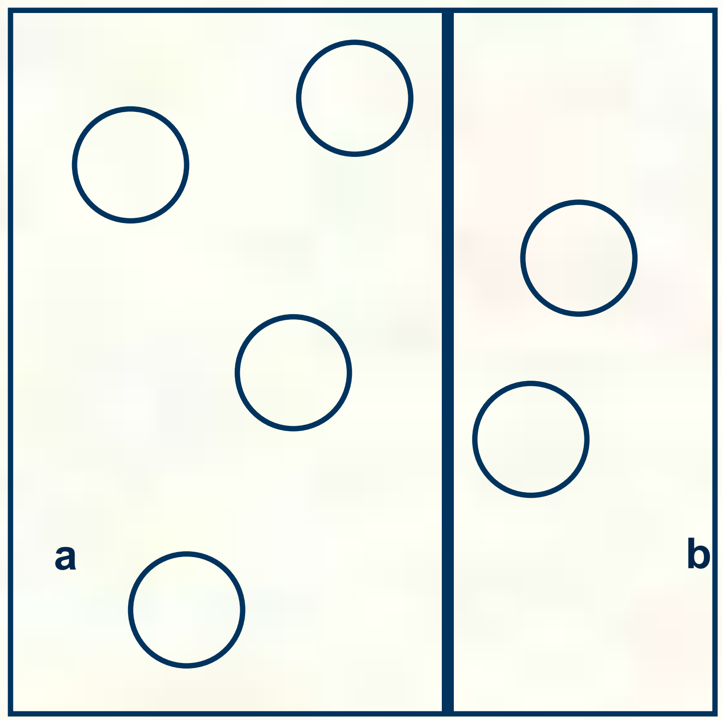
Surface Area and Rays

- The probability of a ray hitting a convex shape
- that is completely inside a convex cell equals



$$\Pr[r \cap S_o \mid r \cap S_c] = \frac{S_o}{S_c}$$

Surface Area Heuristic



Intersection time

$$t_i$$

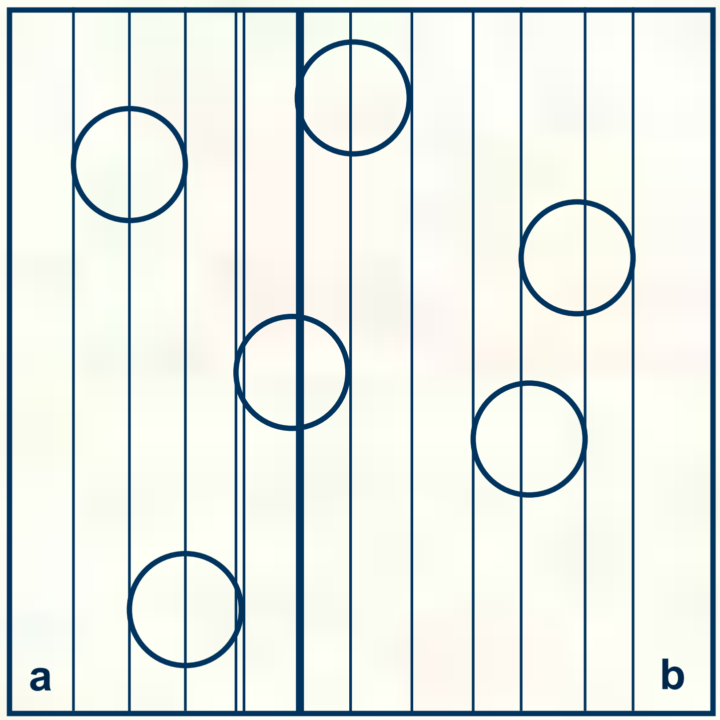
Traversal time

$$t_t$$

$$t_i = 80t_t$$

$$C = t_t + p_a N_a t_i + p_b N_b t_i$$

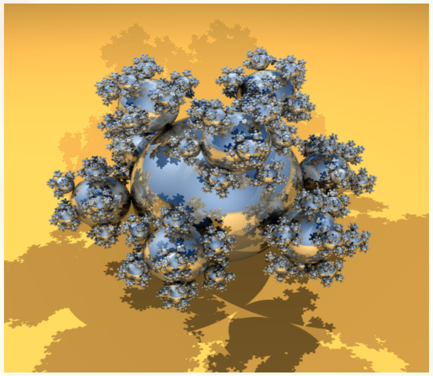
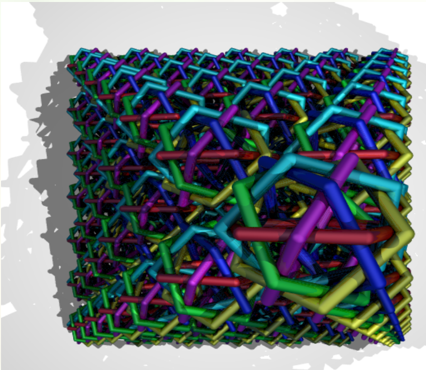
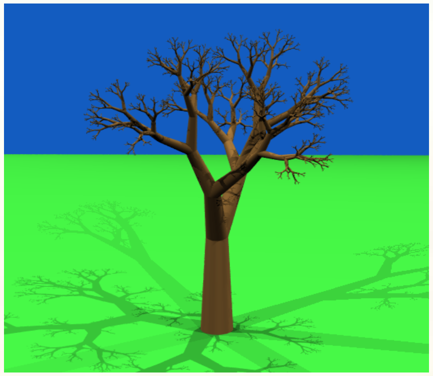
Surface Area Heuristic



2n splits

$$p_a = \frac{S_a}{S} \qquad p_b = \frac{S_b}{S}$$

Comparison

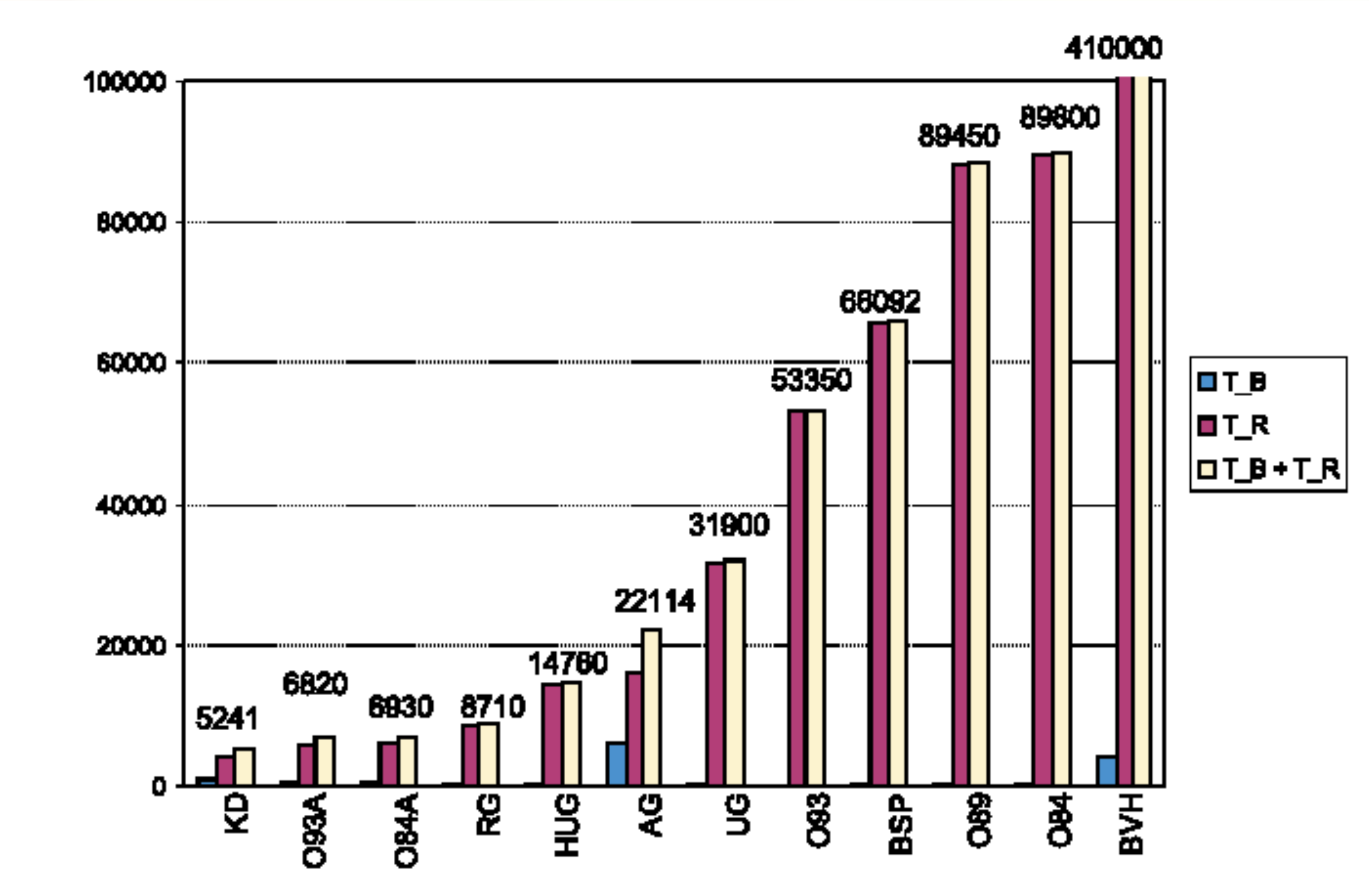
  

Time		Spheres	Rings	Tree
Uniform Grid	d=1	244	129	1517
	d=20	38	83	781
Hierarchical Grid		34	116	34

V. Havran, Best Efficiency Scheme Project

`http://sgi.felk.cvut.cz/BES/`

Comparison



100000
80000
60000
40000
20000
0
5241
6820
6930
8710
14760
22114
31900
53350
66092
89450
89800
410000
KD
O93A
O84A
RG
HUG
AG
UG
O83
BSP
O89
O84
BVH
T_B
T_R
T_B + T_R

Univ. Saarland RTRT Engine

- Ray-casts per second = FPS @ 1K × 1K

RT&Shading Scene	SSE no shd.	SSE simple shd.	No SSE simple shd.
ERW6 (static)	7.1	2.3	1.37
ERW6 (dynamic)	4.8	1.97	1.06
Conf (static)	4.55	1.93	1.2
Conf (dynamic)	2.94	1.6	0.82
Soda Hall	4.12	1.8	1.055

- Pentium-IV 2.5GHz laptop
- Kd-tree with surface-area heuristic [Havran]

- Wald et al. 2003 [`http://www.mpi-sb.mpg.de/~wald/`]

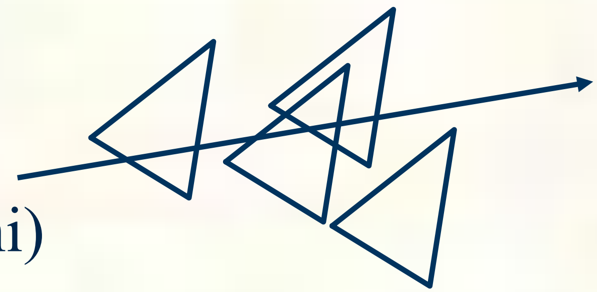
Interactive Ray Tracing

- Highly optimized software ray tracers
  - Use vector instructions; Cache optimized
  - Clusters and shared memory MPs
- Ray tracing hardware
  - AR250/350 ray tracing processor
    www.art-render.com
  - SaarCOR
- Ray tracing on programmable GPUs

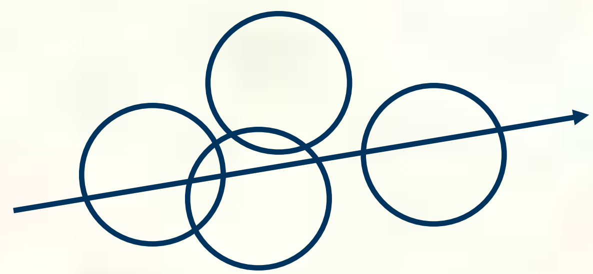
Theoretical Nugget 1

- Computational geometry of ray shooting

- 1. Triangles (Pellegrini)
  - Time: $O(\log n)$
  - Space: $O(n^{5+\varepsilon})$



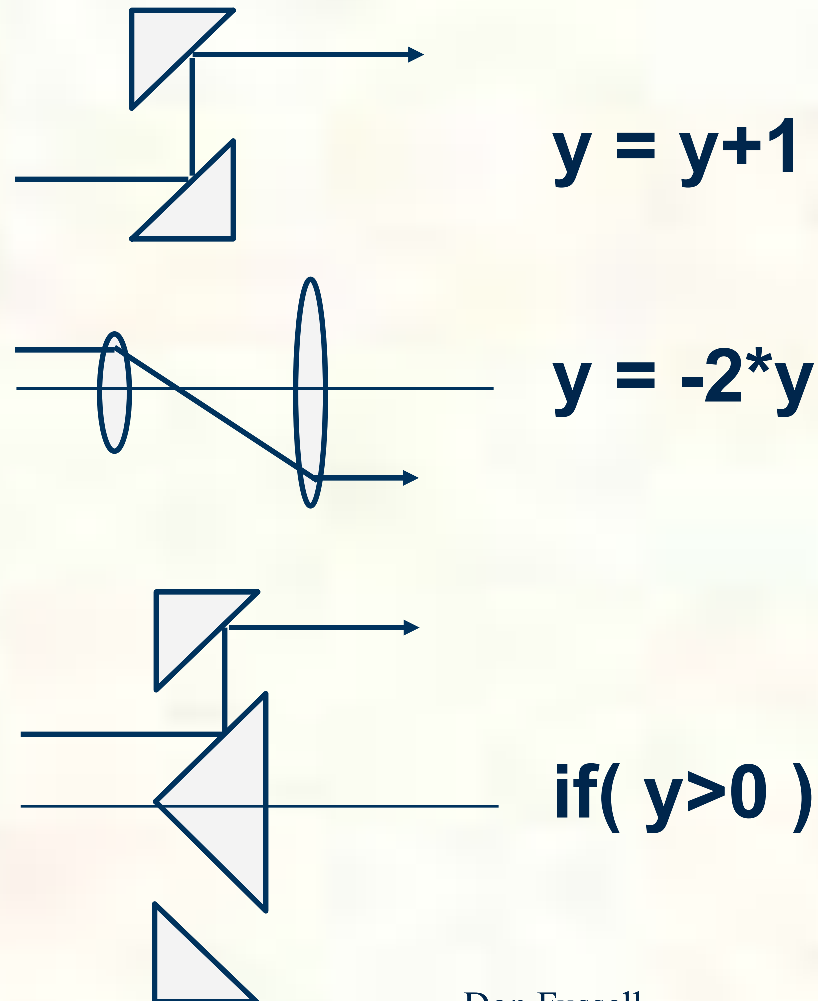
- 2. Sphere (Guibas and Pellegrini)
  - Time: $O(\log^2 n)$
  - Space: $O(n^{5+\varepsilon})$



Theoretical Nugget 2

- Optical computer = Turing machine
- Reif, Tygar, Yoshida

- Determining if a ray
- starting at y0 arrives
- at yn is undecidable

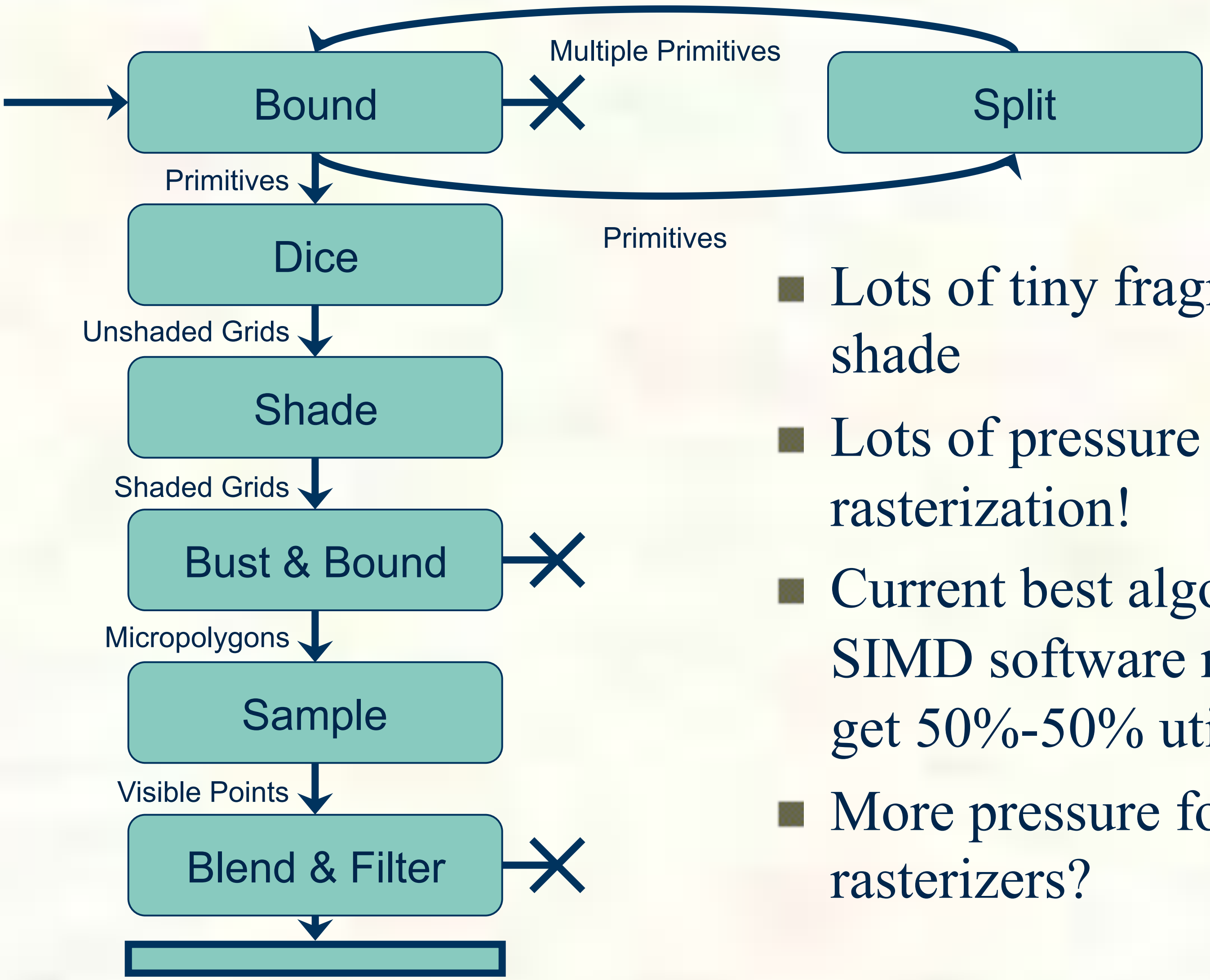


Ray tracing and rasterization

- For nice regular primary and shadow rays
  - Ray tracing:
```
for each ray {
    for each object {
        is there an intersection?
    }
}
```
  - Graphics pipeline:
```
for each object {
    for each ray {
        is there an intersection?
    }
}
```
- Just a loop transform
- Trick - Make it regular - do it in perspective space
- Regular doesn't have to mean regular samples, just easy search!
- Now can be done in real time for primary and shadows
- Faster on CPUs than GPUs

Micropolygons

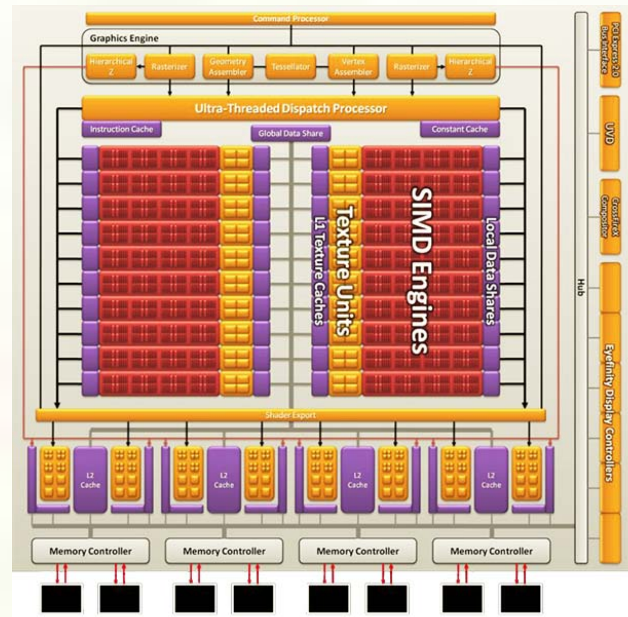
Micropolygons



- Lots of tiny fragments to shade
- Lots of pressure on rasterization!
- Current best algorithms for SIMD software rasterizers get 50%-50% utilization
- More pressure for hardware rasterizers?

Trends

- More cores integrated onto common substrate
  - With DRAM?
- Will the cores be homogeneous or heterogeneous?
  - Some CPU style latency-oriented cores?
  - Some GPU style throughput-oriented cores?
  - Only CPU style?
    - Fewer, more area devoted to on-chip memory
  - Only GPU style?
    - More cores, more compute, more pressure on memory bandwidth
- How are we going to program any of this stuff?



Summary

- High performance GPUs have some of the characteristics of the macrochip design and need some of the same parts capabilities.
- But these are commodity products. Can the optical interconnect and high-bandwidth DRAMs be commodity components?
- Are there other graphics applications, such as perhaps render farms for animation companies, that would be better suited? Could this help solve the big production problem of managing data more effectively?

Summary

- Wide SIMD is here to stay.
- But we had to make some basic quality tradeoffs to make things like this
- So it's not enough, irregular computations growing in importance
- DRAM bandwidth!
- Parallel programming!
- Can we rely less on streaming techniques, regular access patterns, etc.?
- Lower the arithmetic intensity (flops/byte)

Acknowledgment

Portions of this talk adapted from Kayvon Fatahalian's excellent Siggraph GPU tutorial

Thanks Kayvon!