Computer Graphics I

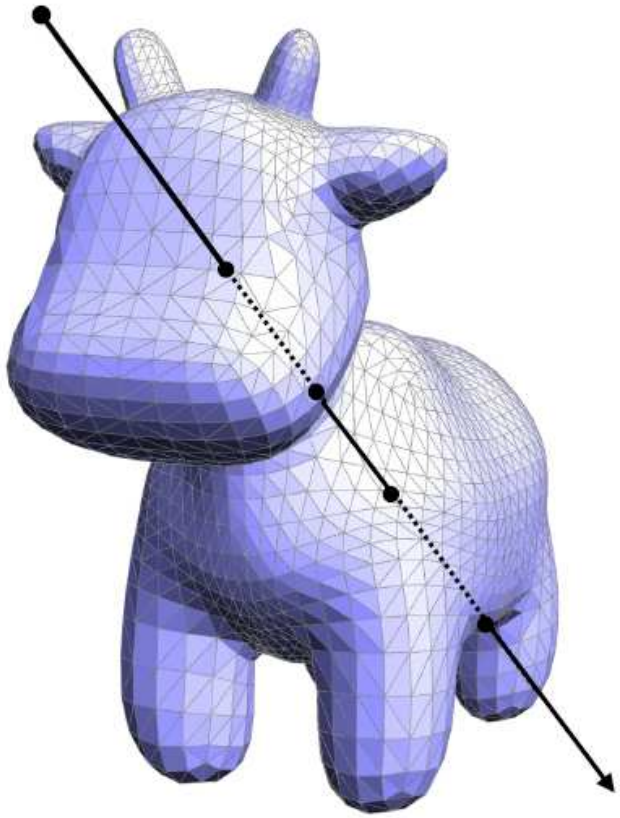
Lecture 10: Efficient ray-geometry intersection

Xiaopei LIU

School of Information Science and Technology
ShanghaiTech University

Ray-mesh intersection

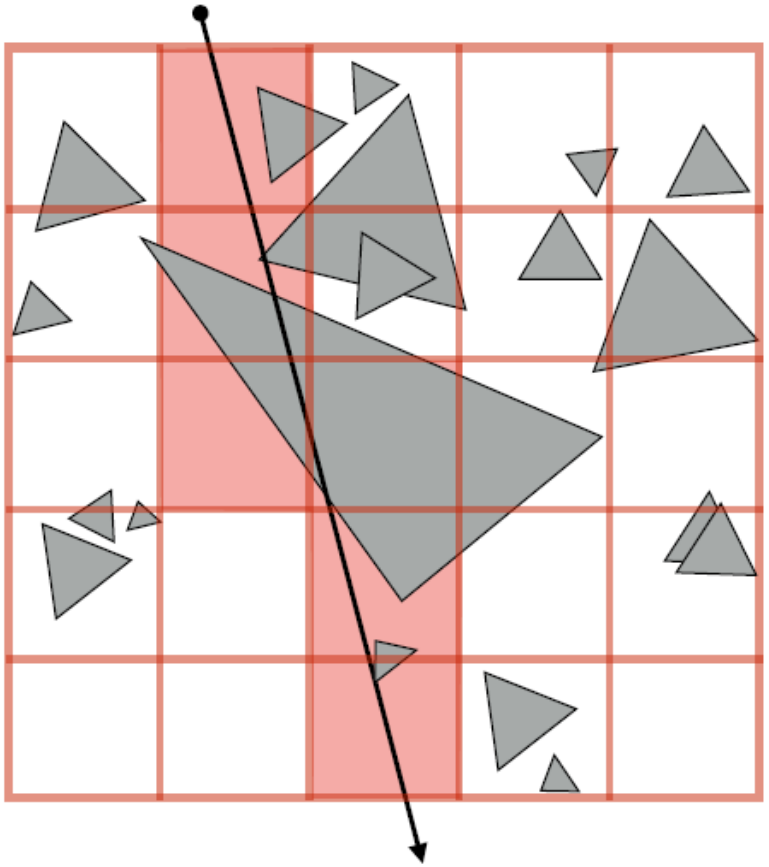
- **How to intersect a mesh?**
  - Intersect its triangles
  - Search the triangles the ray hits
  - Obtain intersection point and normal



1. Grid acceleration

Uniform grid

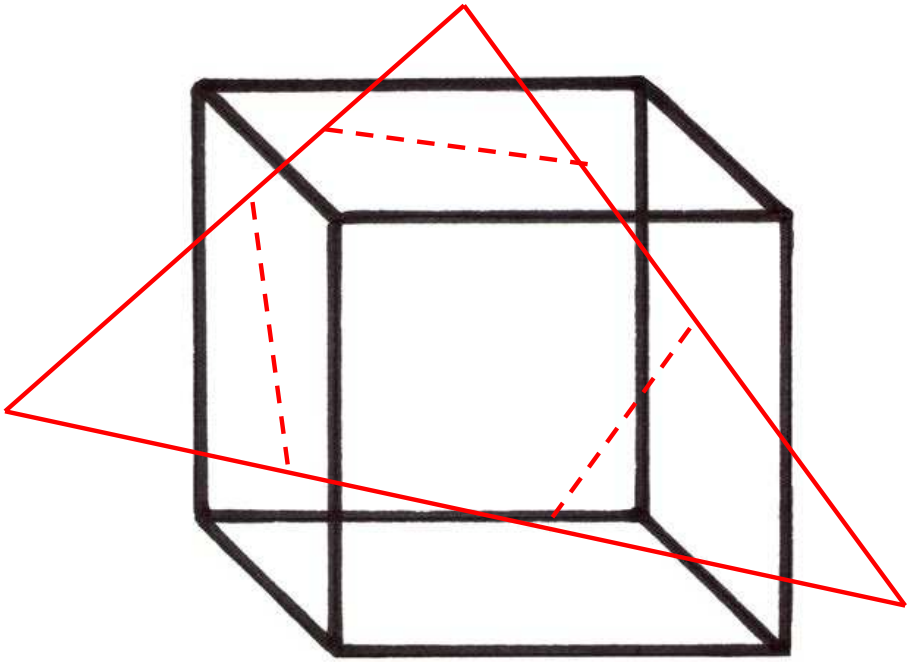
- **How to search the intersected triangles?**
  - Grid acceleration structure
    - For each cell, we record 2D triangles included
    - Check whether any triangle locates inside the cell



- **Partition space into equal sized volumes ("voxels")**
- **Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)**
- **Walk ray through volume in order**
  - **Very efficient implementation possible (think: 3D line rasterization)**
  - **Only consider intersection with primitives in voxels the ray intersects**

Triangle-box overlap testing

- **Test each box face independently**
  - Compute the intersection line (AABB is very easy)
  - Identify whether the line is inside the rectangle with Cohen-Sutherland line clipping algorithm
  - If any face has intersection line inside, the triangle overlaps with the box



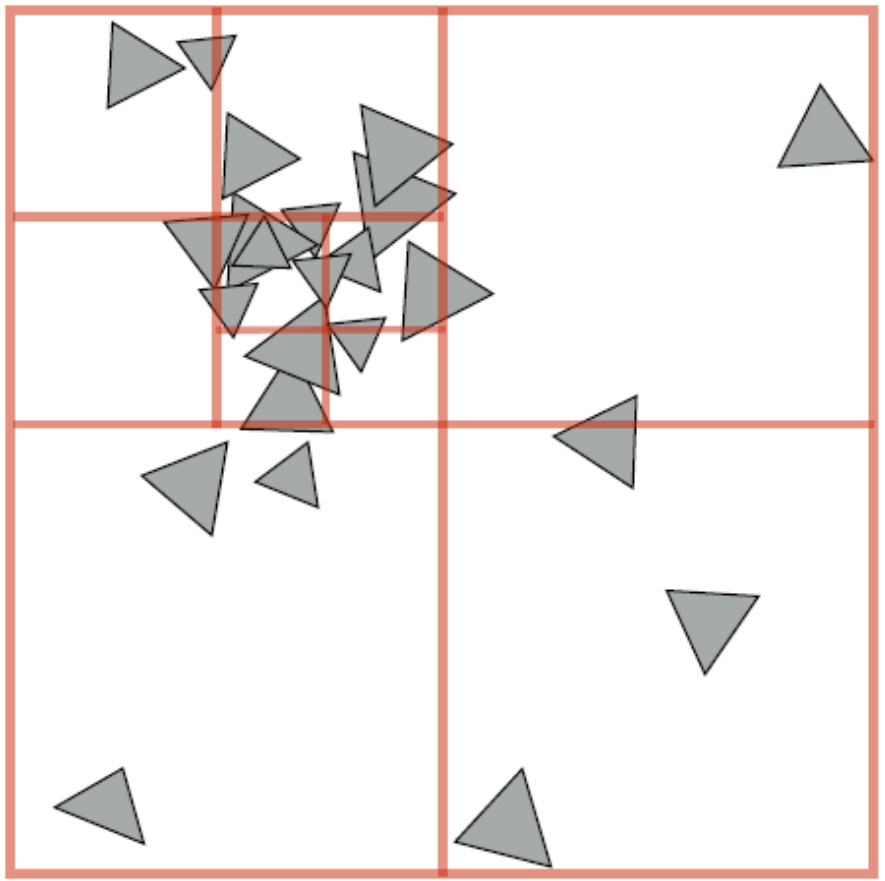
Adaptive grid

- **Quad-tree / octree**

  Like uniform grid: easy to build (don't have to choose partition planes)

  Has greater ability to adapt to location of scene geometry than uniform grid.

  But lower intersection performance than K-D tree (only limited ability to adapt)



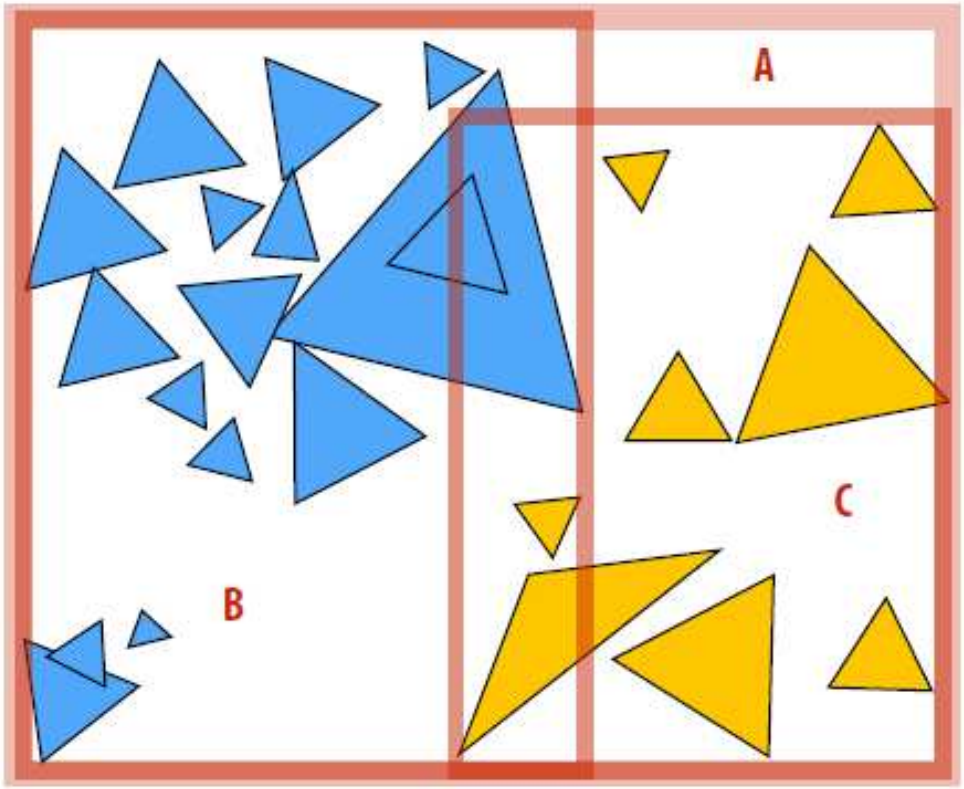
Quad-tree: nodes have 4 children (partitions 2D space)

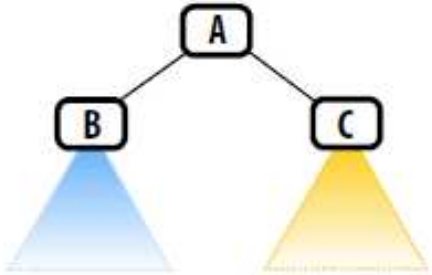
Octree: nodes have 8 children (partitions 3D space)

2. Bounding volume hierarchies

Bounding volume hierarchies

- BVH partitions each node's primitives into disjoints sets
  - Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)



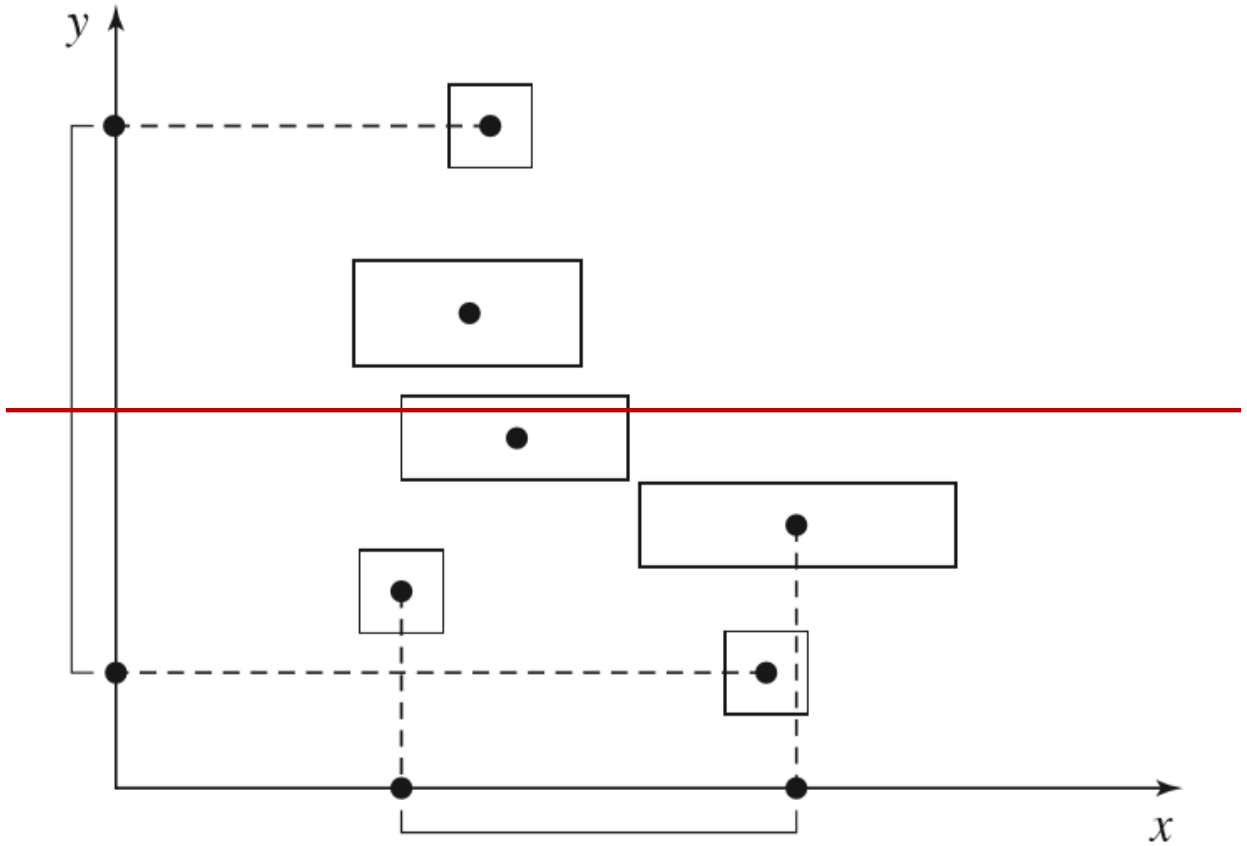


Bounding volume hierarchies

- **Division (partitioning) criteria**
  - **Basic goal**
    - Select a partition of primitives that doesn't have too much overlap of the bounding boxes
      - If there is substantial overlap, it will more frequently be necessary to traverse both children and subtrees
    - Simple rule:
      - Choose one of the three axes with greatest variation of bounding box centroids
      - Use the mid-point of the centroids along that axis to subdivide

Bounding volume hierarchies

- **Partition based on greatest variation of bounding box centroids**



Bounding volume hierarchies

- **Surface area heuristic (SAH)**
  - **The SAH model estimates the costs of ray-intersection tests**
    - The time on traversing nodes
    - The time on ray-primitive intersection tests
    - Algorithm to minimize total costs
    - A greedy algorithm is used to minimize the cost for each single node

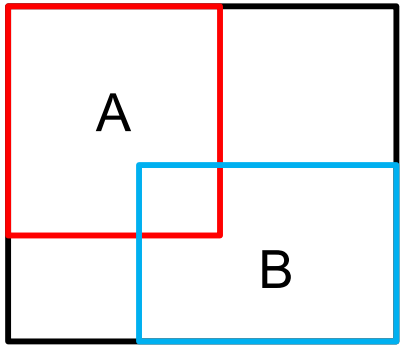
Bounding volume hierarchies

- **Surface area heuristic (SAH)**
  - **The idea behind the SAH cost model is straightforward**
    - Leaf-node test
      - We could just create a leaf node for the current region and geometry
      - Any ray that passes through this region will be tested against all of the overlapping primitives
      - This will incur a cost of:

$$\sum_{i=1}^{N} t_{\text{isect}}(i)$$

Bounding volume hierarchies

- **Surface area heuristic (SAH)**
  - **The idea behind the SAH cost model is straightforward**
    - Splitting test
      - The other option is to split the region
      - In that case, rays will incur the cost:



$$c(A, B) = t_{\text{trav}} + p_A \sum_{i=1}^{N_A} t_{\text{isect}}(a_i) + p_B \sum_{i=1}^{N_B} t_{\text{isect}}(b_i)$$

where t_{trav} is the time to traverse the interior node and determine the child the ray passes through

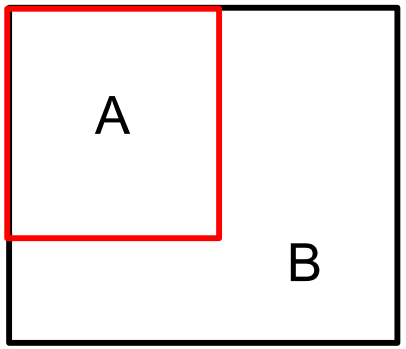
p_A and p_B are the probabilities that the ray passes through each of the child nodes

Bounding volume hierarchies

- **Surface area heuristic (SAH)**
  - **The idea behind the SAH cost model is straightforward**
    - How to partition primitives affects the values of the two probabilities
    - How to compute the probabilities p_A *and* p_B?
      - Geometric probability
      - For a convex volume *A* contained in another convex volume *B*, the conditional probability that a random ray passing through *B* will also pass through *A* is:

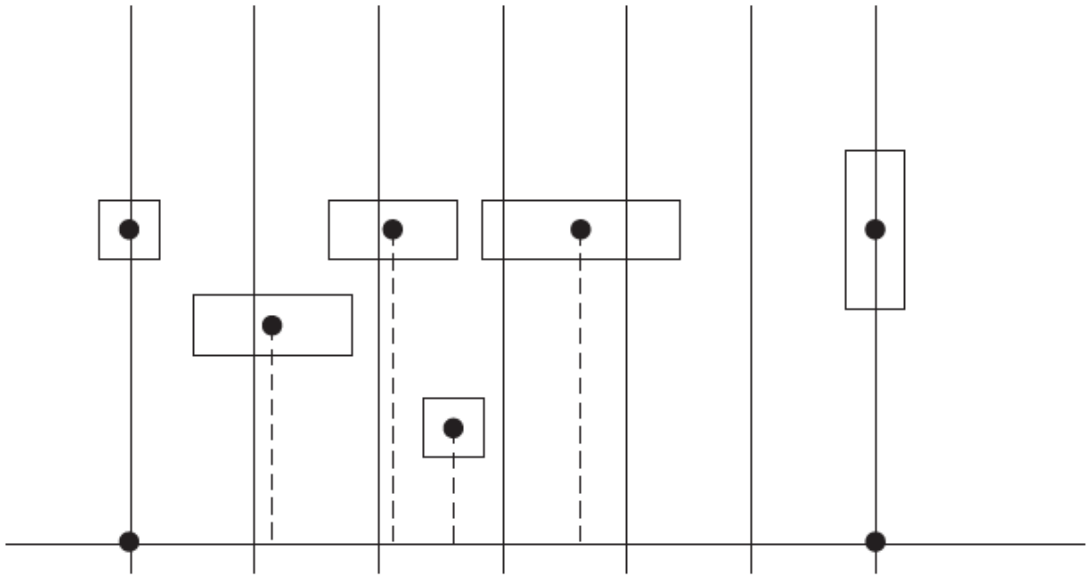
$$p(A|B) = \frac{s_A}{s_B}$$

s_A and s_B are surface areas of primitives



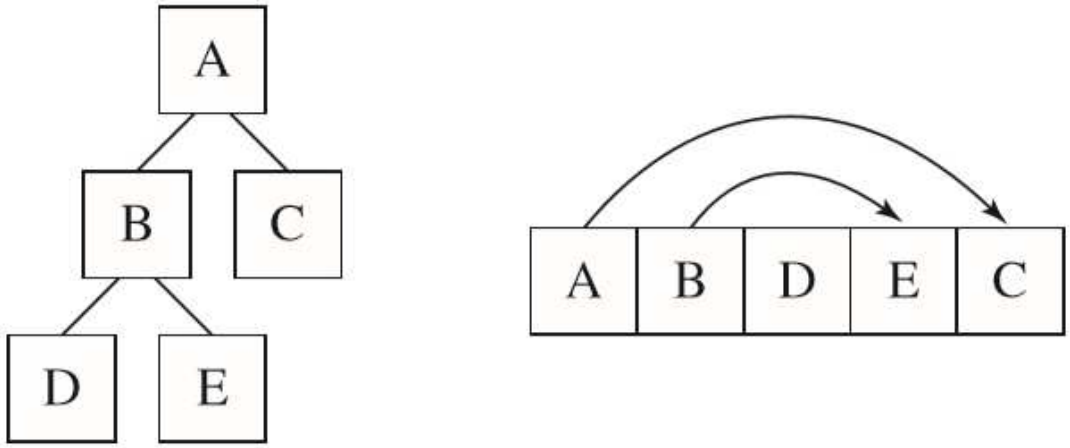
Bounding volume hierarchies

- **Partition based on SAH**
  - Choose a partition of the primitives along the chosen axis that gives a minimal SAH cost estimate
  - Consider a number of candidate partitions
  - It creates the most efficient trees for rendering



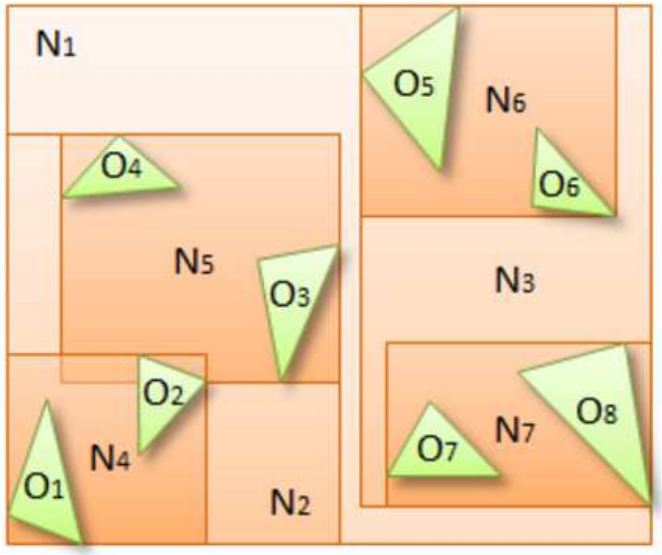
Bounding volume hierarchies

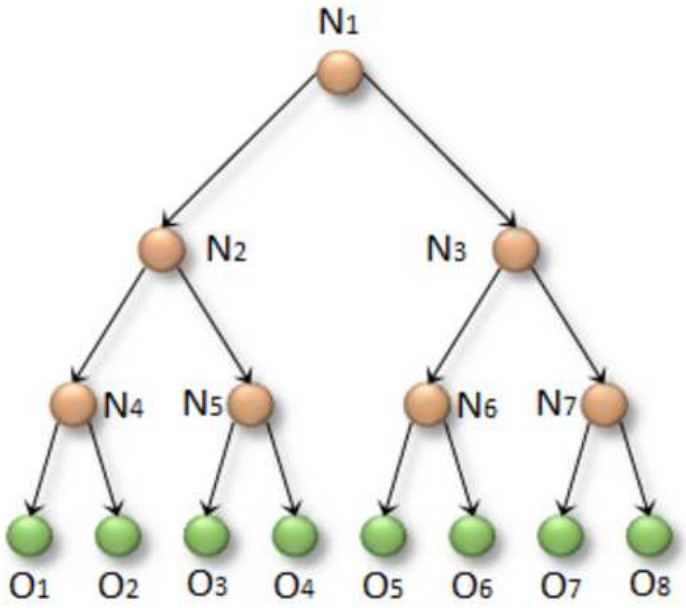
- **Compact BVH for traversal**
  - The final BVH is stored in a linear array in memory
  - Doing so improves cache, memory, and thus overall system performance
  - The nodes of the original tree are laid out in depth-first order



Bounding volume hierarchies

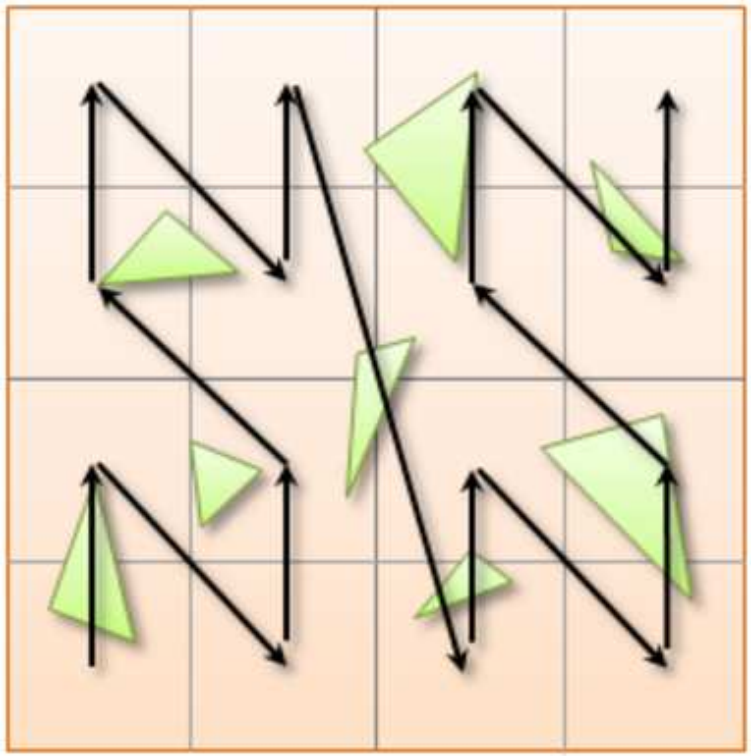
- **BVH based on Morton codes**
  - **Motivations**
    - Nearby spatial primitives should be stored as nearby tree nodes
    - Improving cache hit rate and thus overall performance





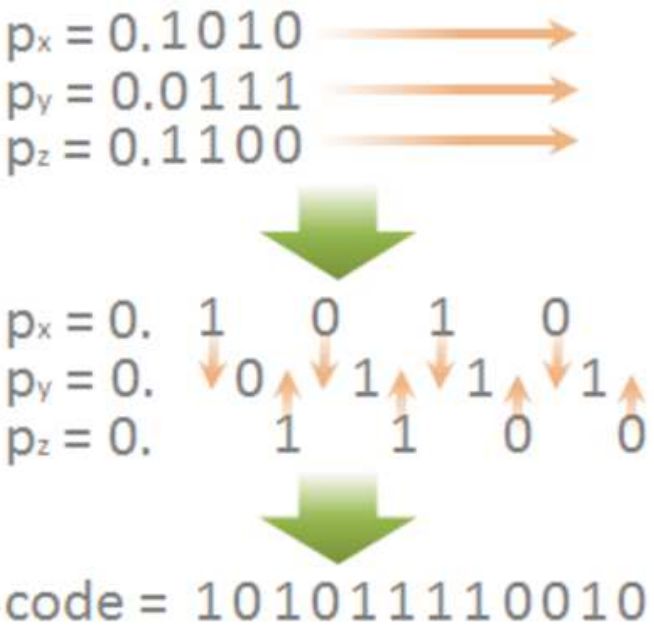
Bounding volume hierarchies

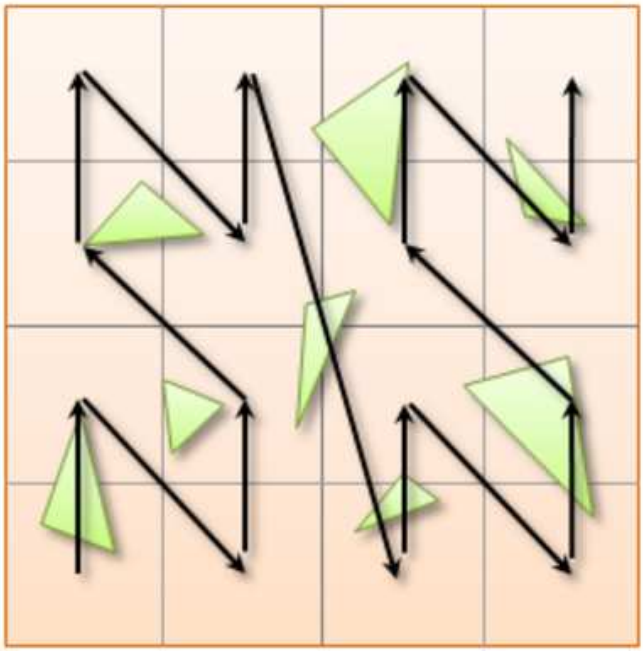
- **Space-filling object ordering**
  - A reasonable choice is to sort primitives along a space-filling curve
  - We use the Z-order curve for simplicity



Bounding volume hierarchies

- **Morton codes for z-ordering**
  - Sort the object according to the Morton code
    - Parallel radix sort is preferred



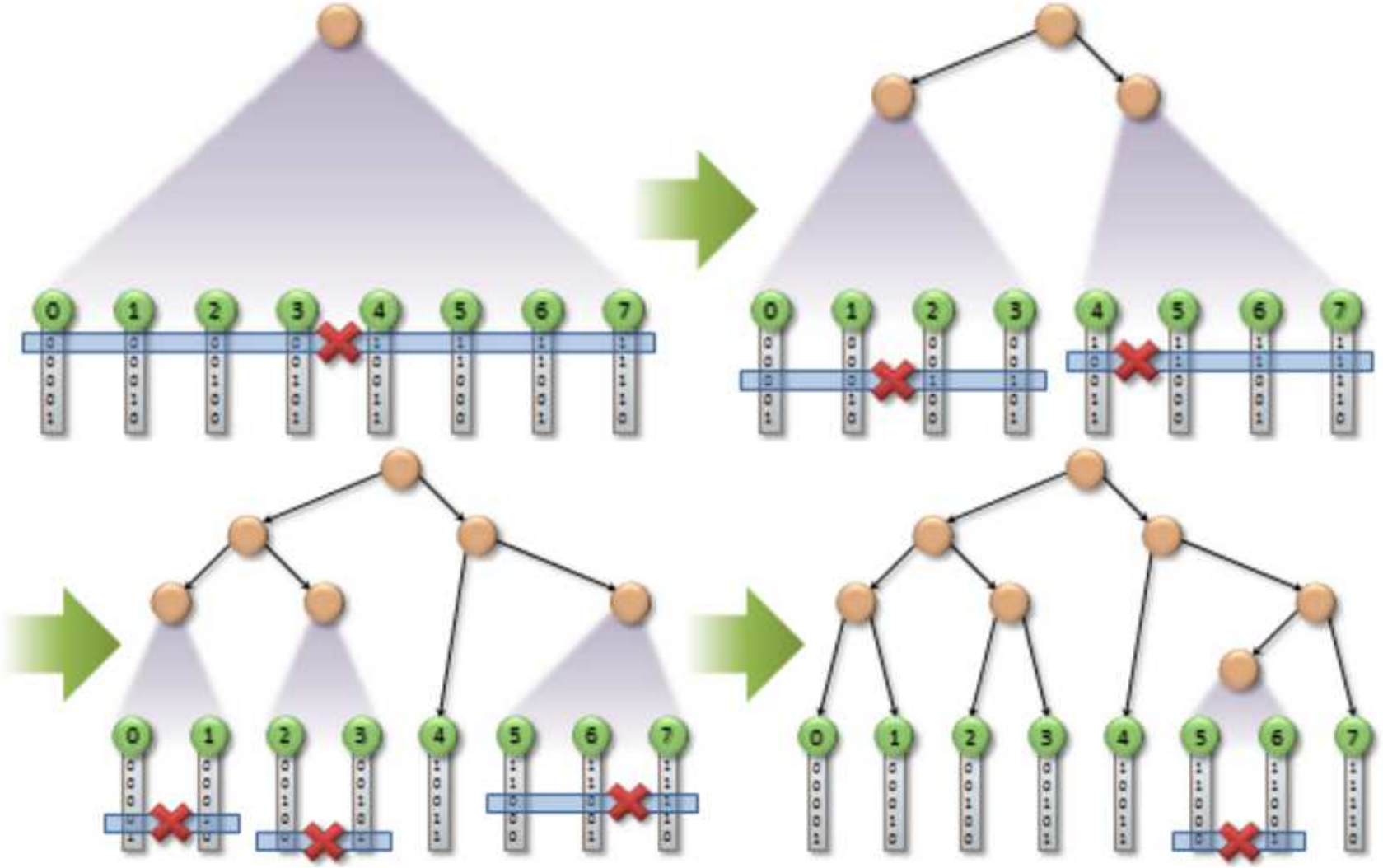


Bounding volume hierarchies

- **Top-down hierarchy generation**
  - We start with a range that covers all objects (first=0, last=N-1), and determine an appropriate position to split the range into two (split=γ)
  - The split position is determined by examining the bit changes (a bit change indicates a larger difference in spatial relation)
  - The process is recursively applied to sub-trees

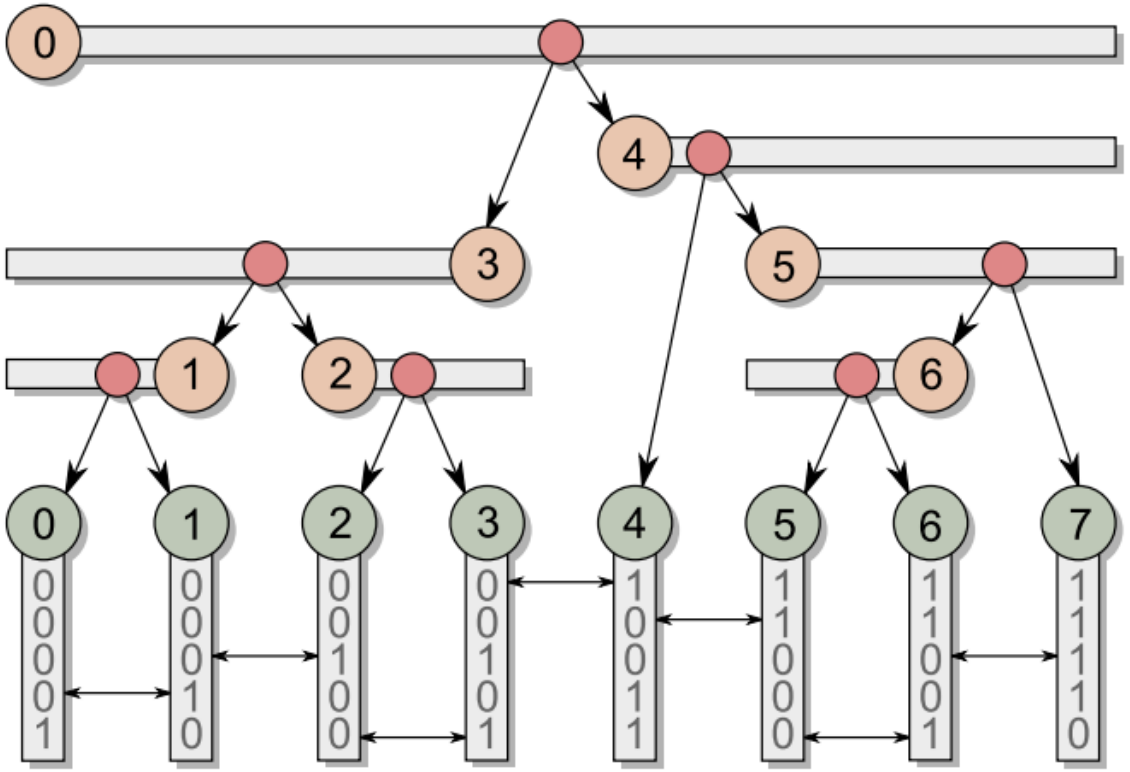
Bounding volume hierarchies

- **Top-down hierarchy generation**
  - Illustration



Bounding volume hierarchies

- **Any possibility of parallel BVH construction?**
  - Tero Karras, NVIDIA Research, Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees, High Performance Graphics (2012)



3. Binary space partitioning

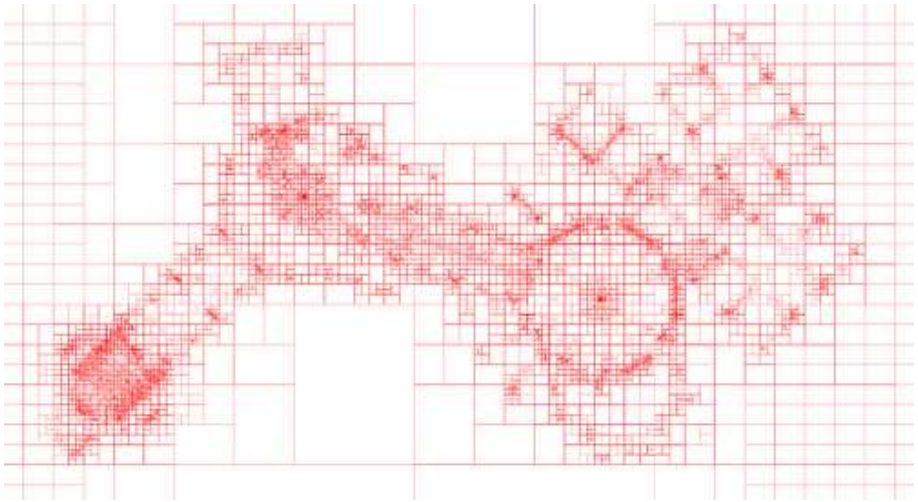
Binary space partitioning

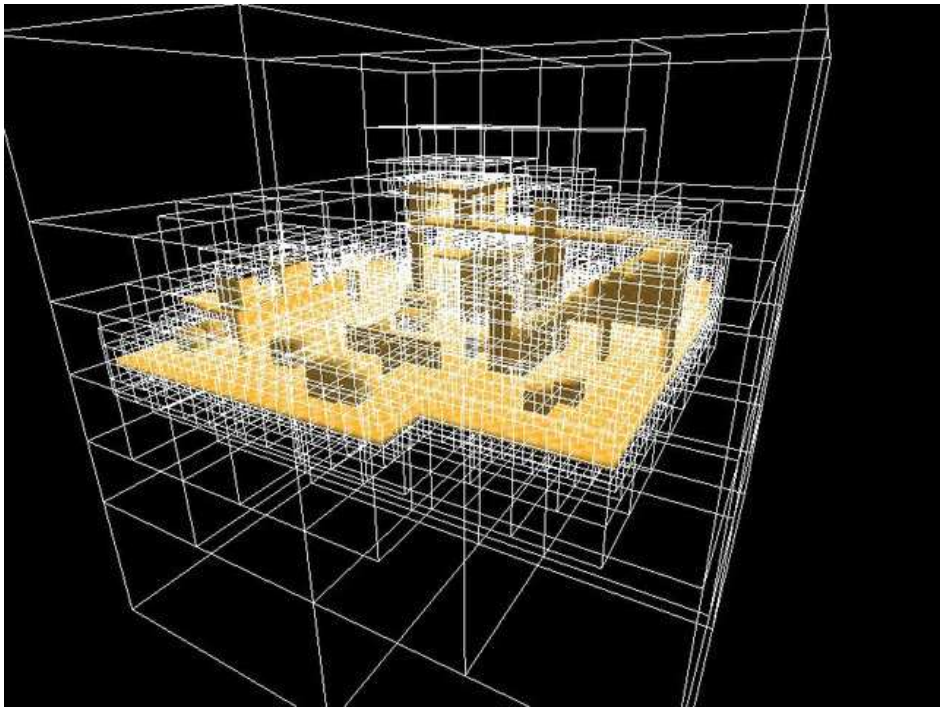
- **Binary space partitioning (BSP) tree**
  - Adaptively subdivide space into irregularly sized regions
  - Compared to grid, much more effective for storing irregularly distributed collections of geometry

- **General construction principle**
  - Step 1: BSP tree starts with a bounding box that encompasses the entire scene
  - Step 2: If the number of primitives in the box is greater than some threshold, split the box in half
  - Step 3: Continue splitting until each leaf region contains sufficiently small number of primitives

Binary space partitioning

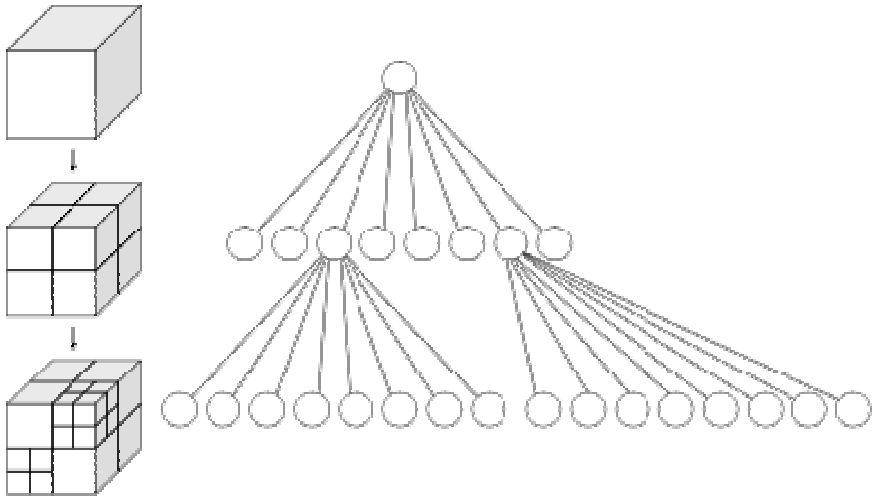
- **Two variations of BSP trees**
  - K-D tree: divide the box into two sub-boxes for each space subdivision
  - Not necessarily equal subdivision along each dimension

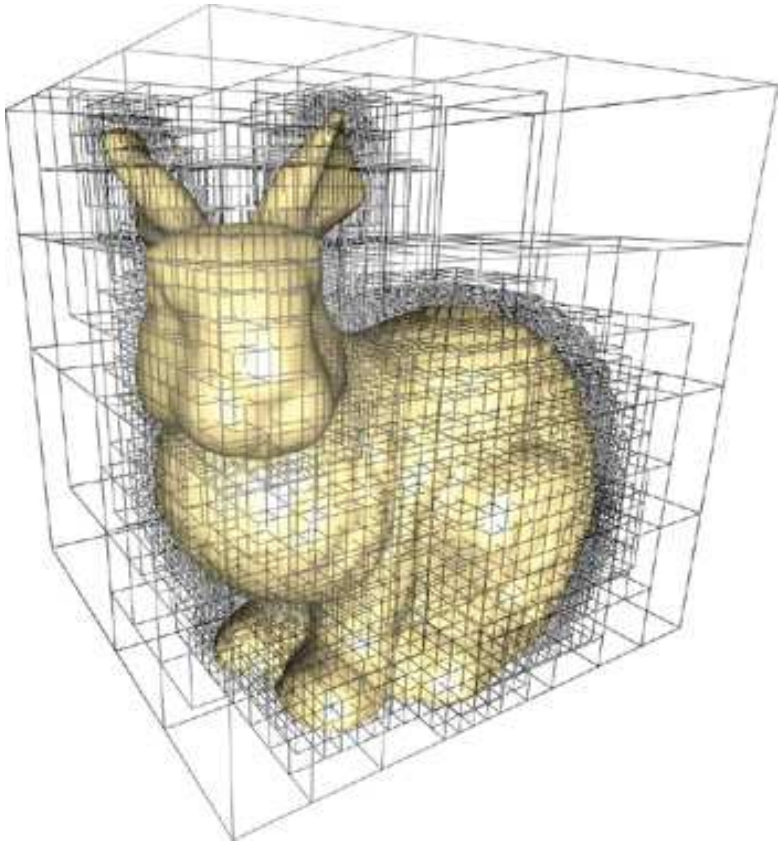




Binary space partitioning

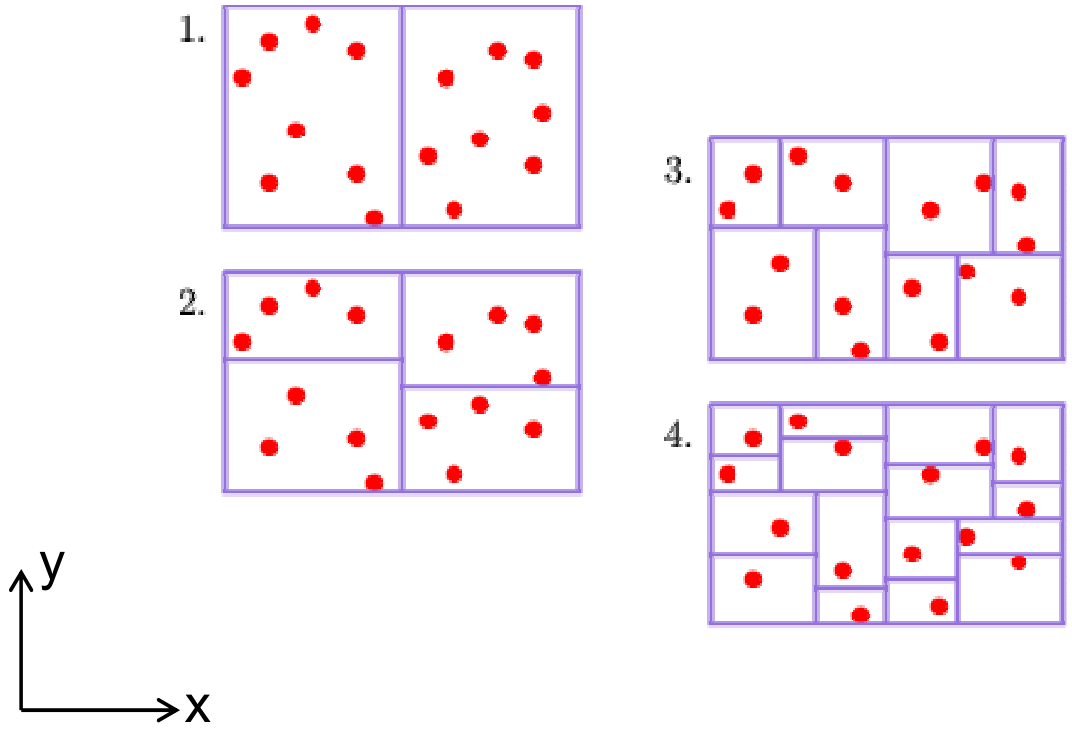
- **Two variations of BSP trees**
  - Octree: divide the box into eight sub-boxed for each space subdivision;
  - Equal subdivision along each dimension





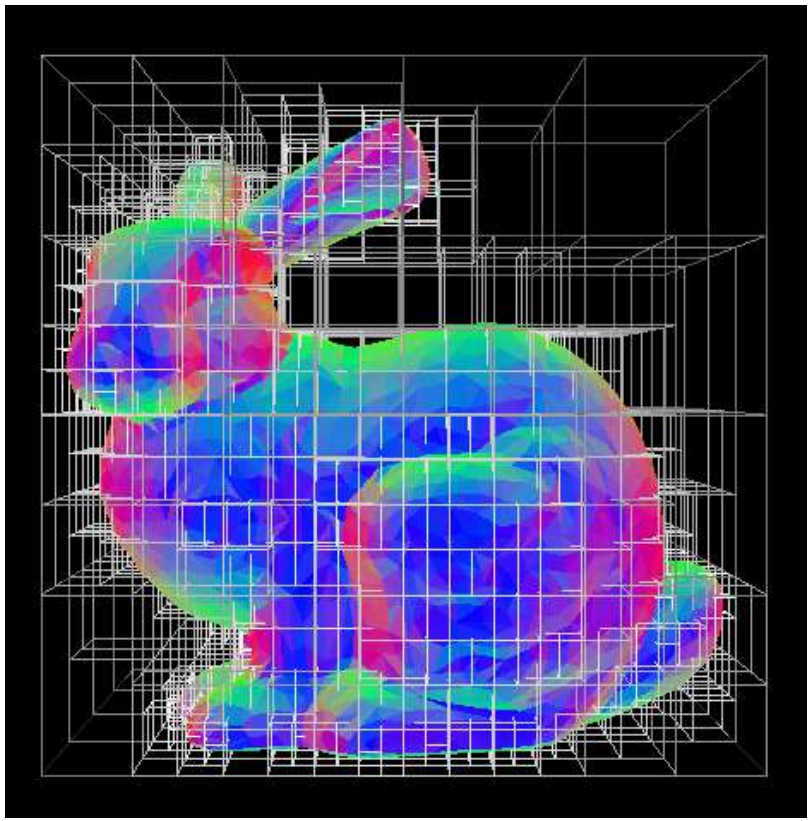
K-D tree construction

- **Primitive representation**
  - Each primitive is represented by its centroid
- **Coordinate cycling**



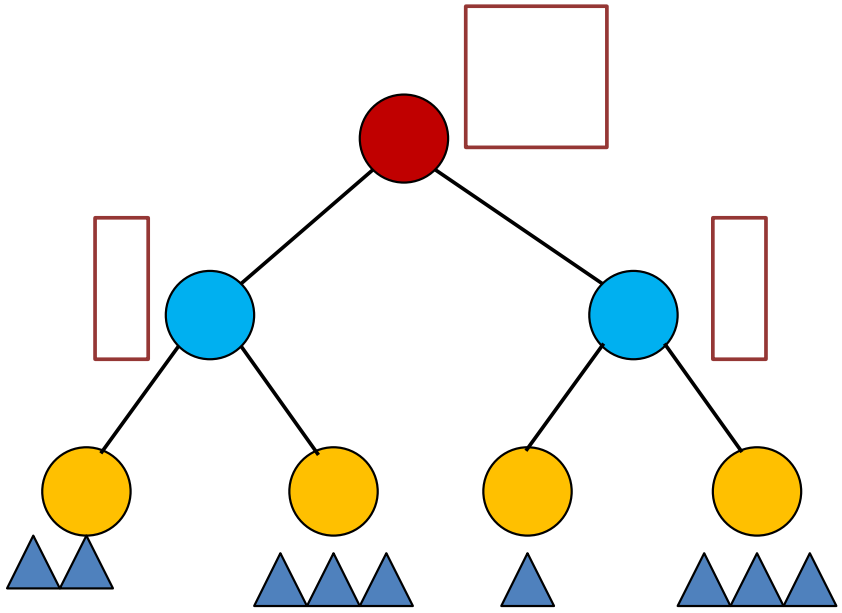
K-D tree construction

- **Different partition rules**
  - Mid-point of bounding box along one axis
  - Median of centroids along one axis
  - SAH
  - etc.



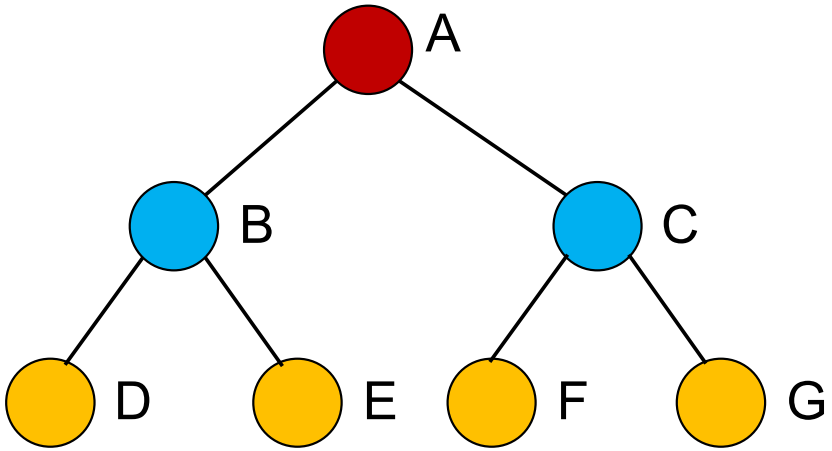
K-D tree construction

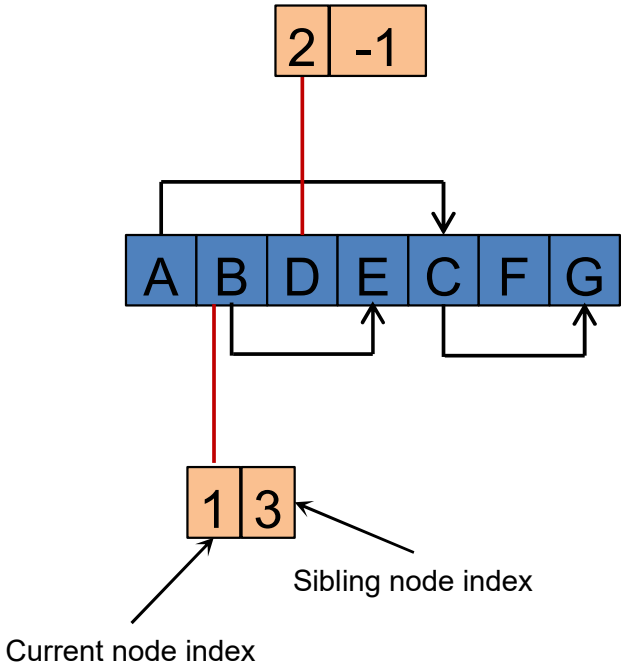
- **Tree structure**
  - **For internal node:**
    - Store bounding box
  - **For leaf node:**
    - Store primitives that overlap with the bounding box



K-D tree construction

- **Storage structure**
  - Depth-first order



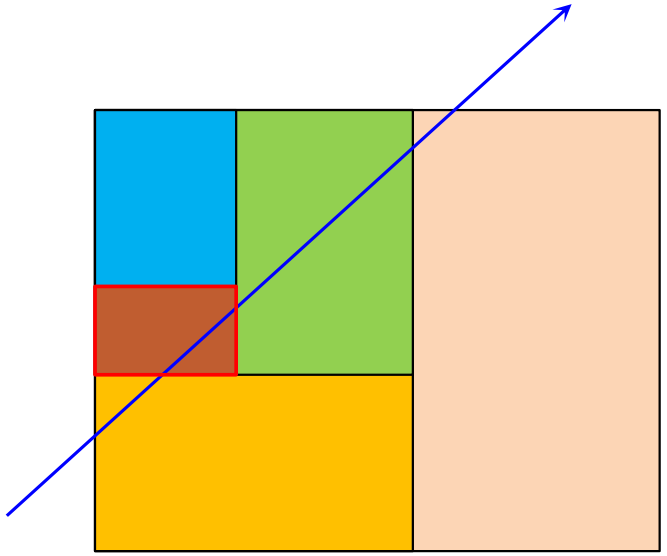


K-D tree construction

- **Real-time K-D tree construction on the GPU?**
  - Please refer to
    Real-Time KD-Tree Construction on Graphics Hardware, Kun Zhou, Qiming Hou, Rui Wang, Baining Guo, SIGGRAPH Asia 2008

Ray traversal

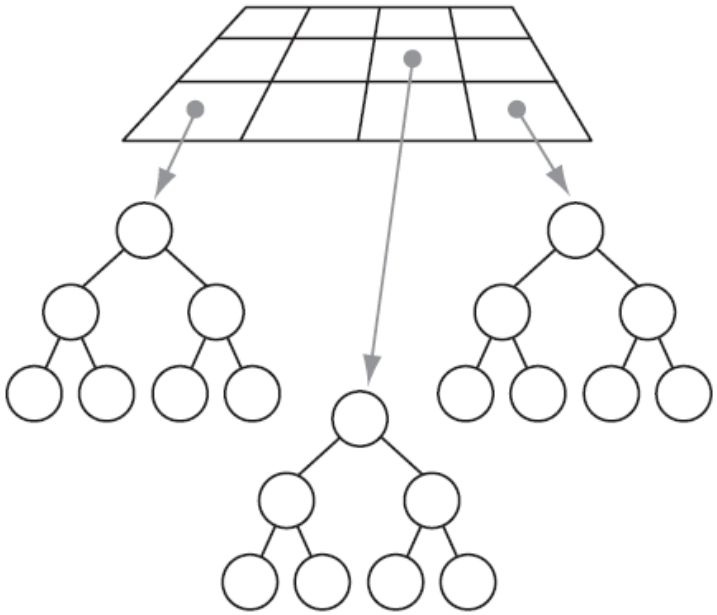
- **For internal node**
  - Perform ray-box intersection test
  - Only detect whether ray-box intersect
  - Sort during the traversal (the nearest intersected box must be visited first)



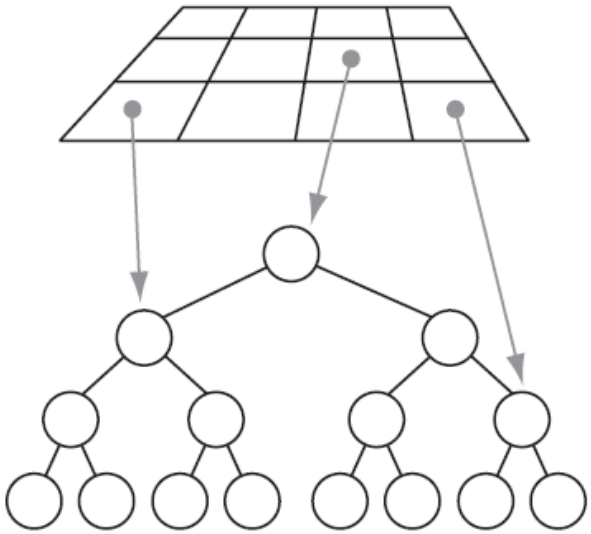
- **For leaf node**
  - Test primitives (triangles) one by one

Hybrid Spatial Partitioning

- **Hybrid schemes**
  - A grid of trees, each grid cell containing a separate tree or part of a tree



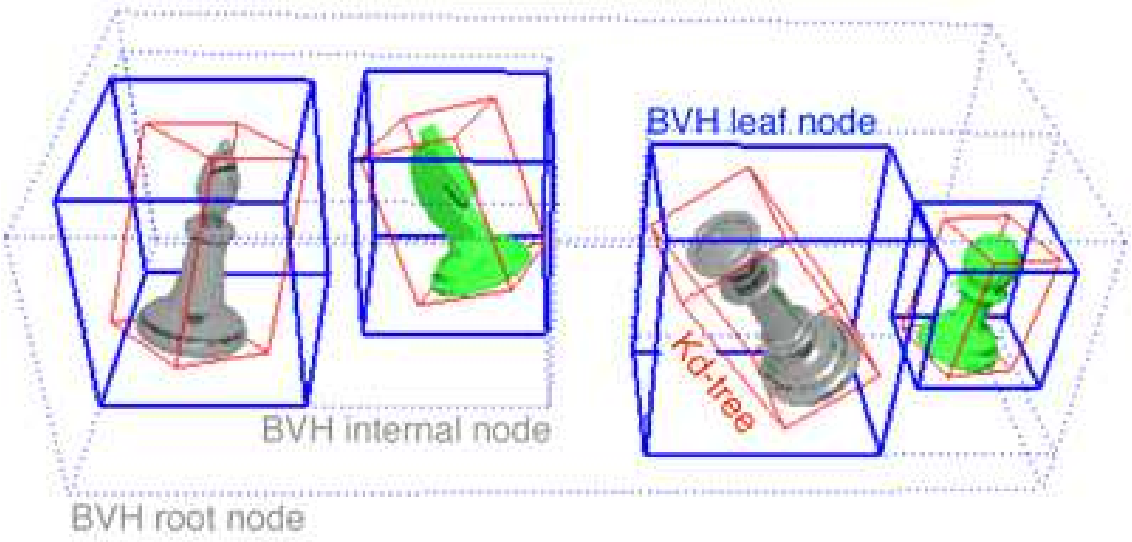
(a)



(b)

Scene organization

- **How an entire rendered scene is organized?**
  - Two levels of tree construction
    - First level: object level: BVH tree
    - Second level: mesh level, K-D/Octree tree



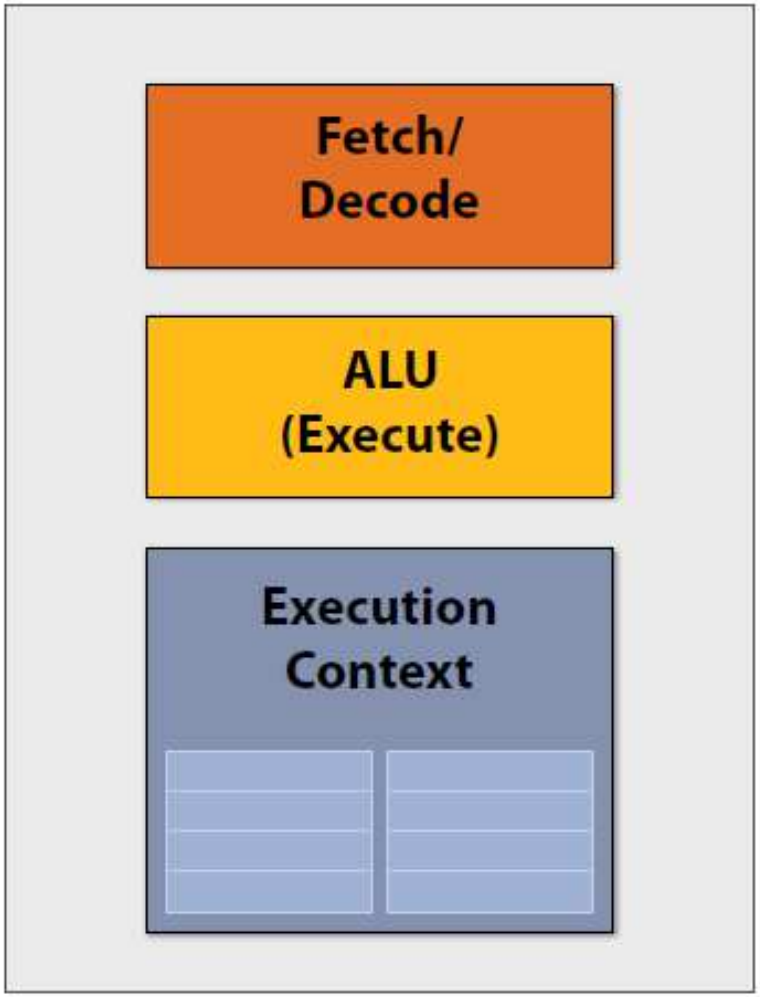
4. Parallel ray-geometry intersection

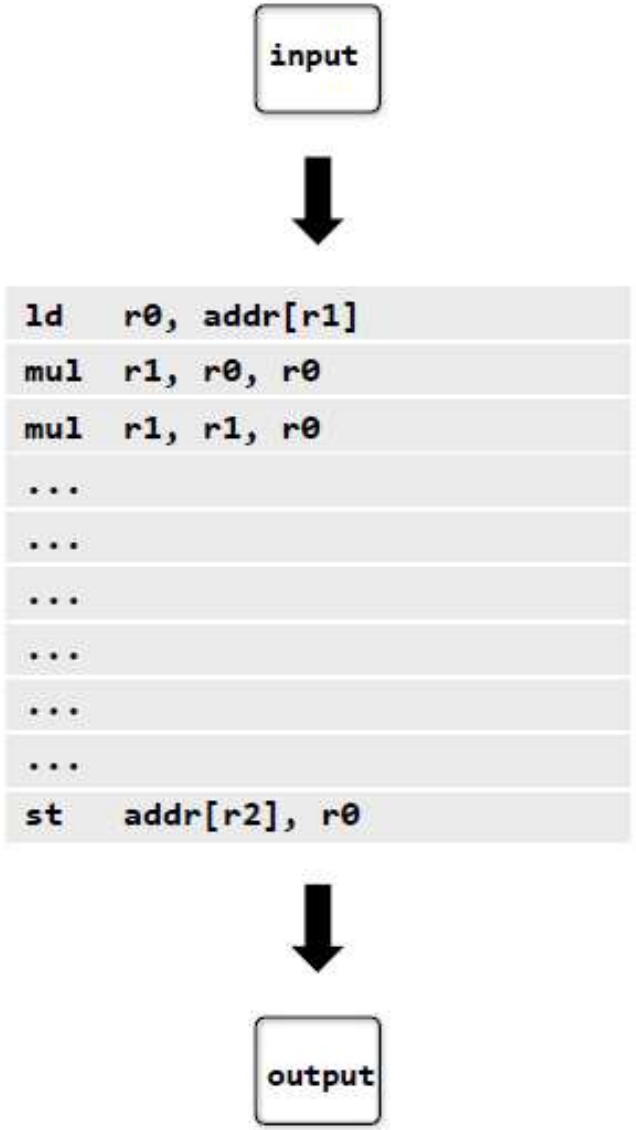
Parallel ray tracing

- **Imagine I give you a 16-core CPU, how would you parallelize your ray tracer to render this picture?**

Image credit: NVIDIA (this ray traced image can be rendered at interactive rates on modern GPUs)

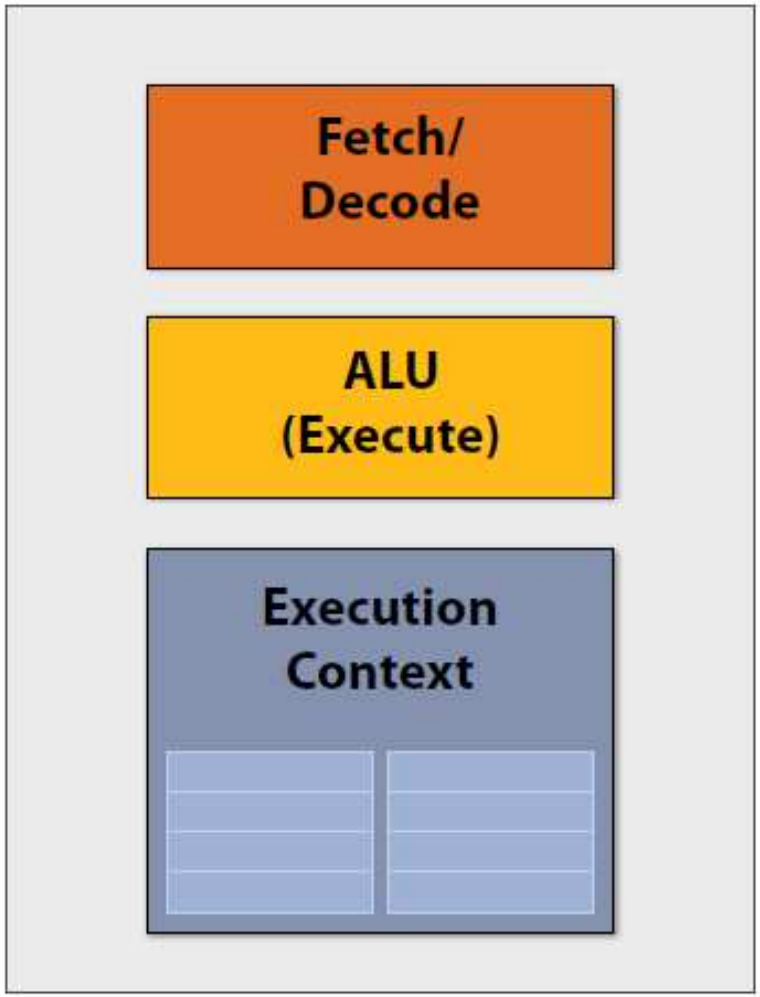
What does a processor do?

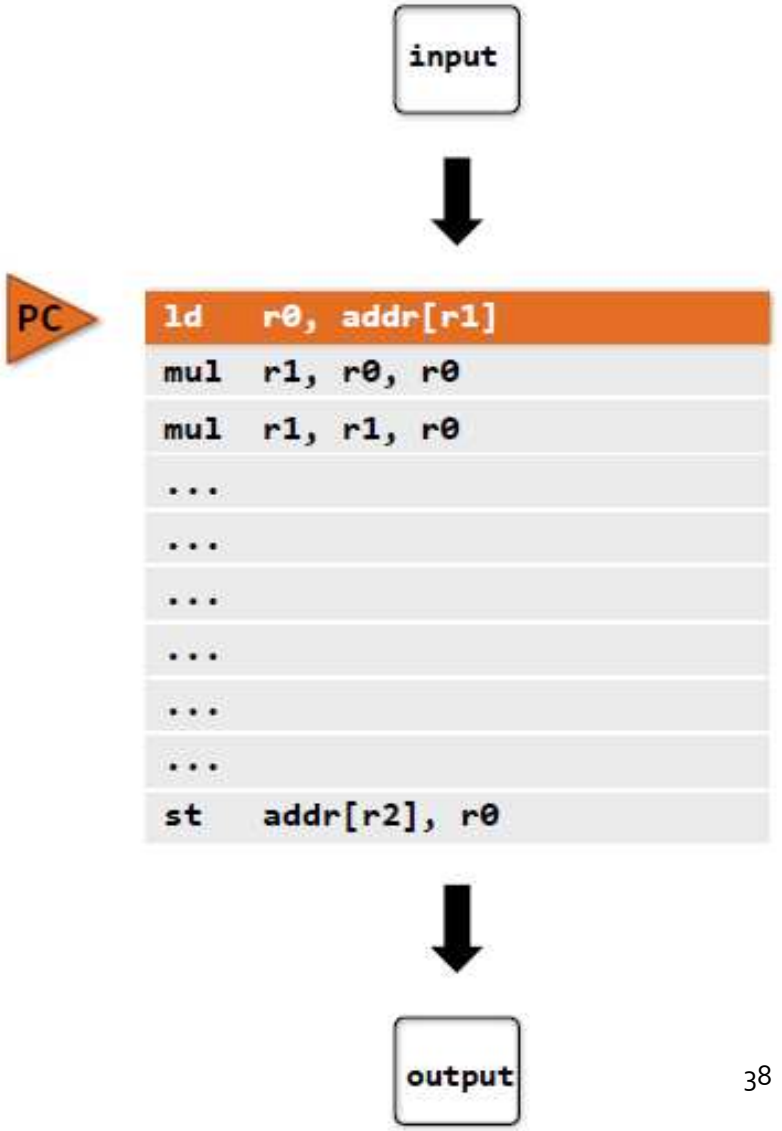




A processor executes an instruction stream

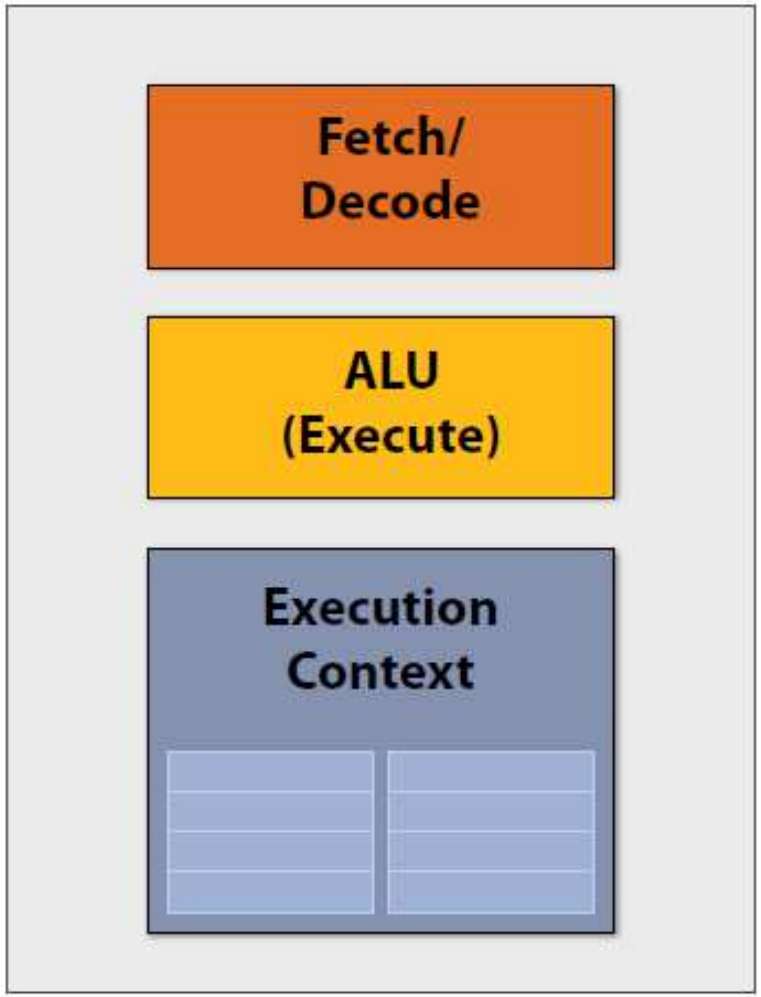
My very simple processor: executes one instruction per clock

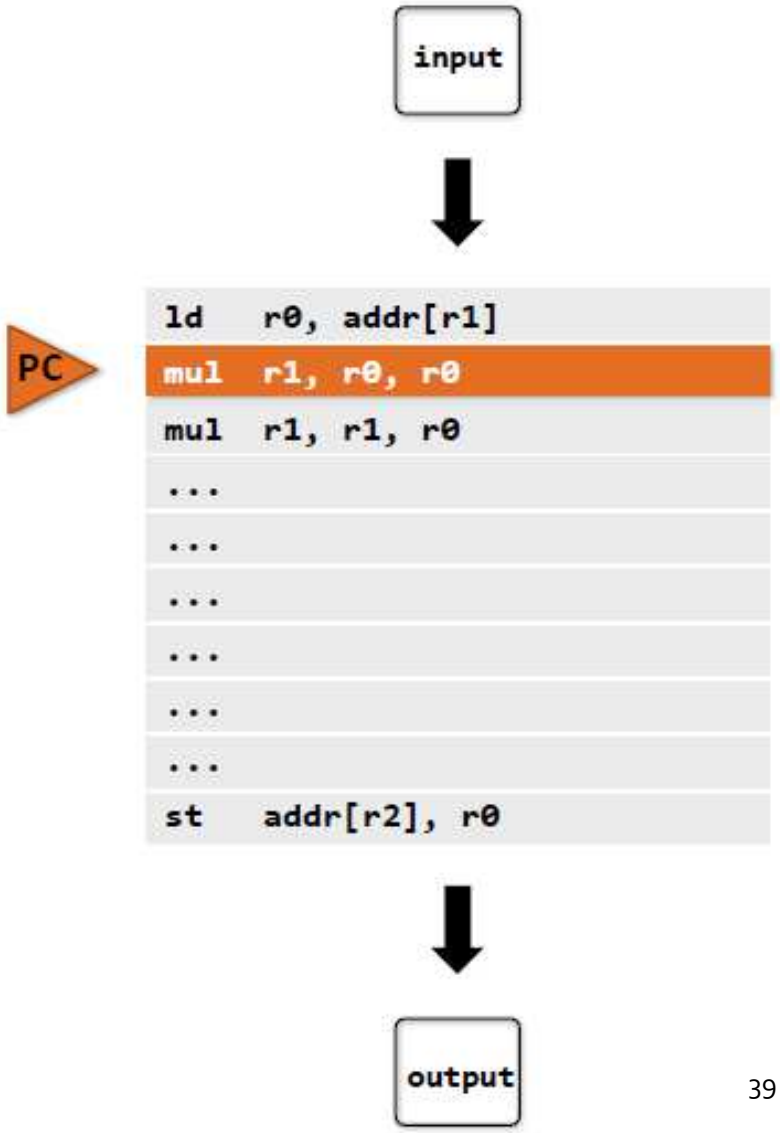




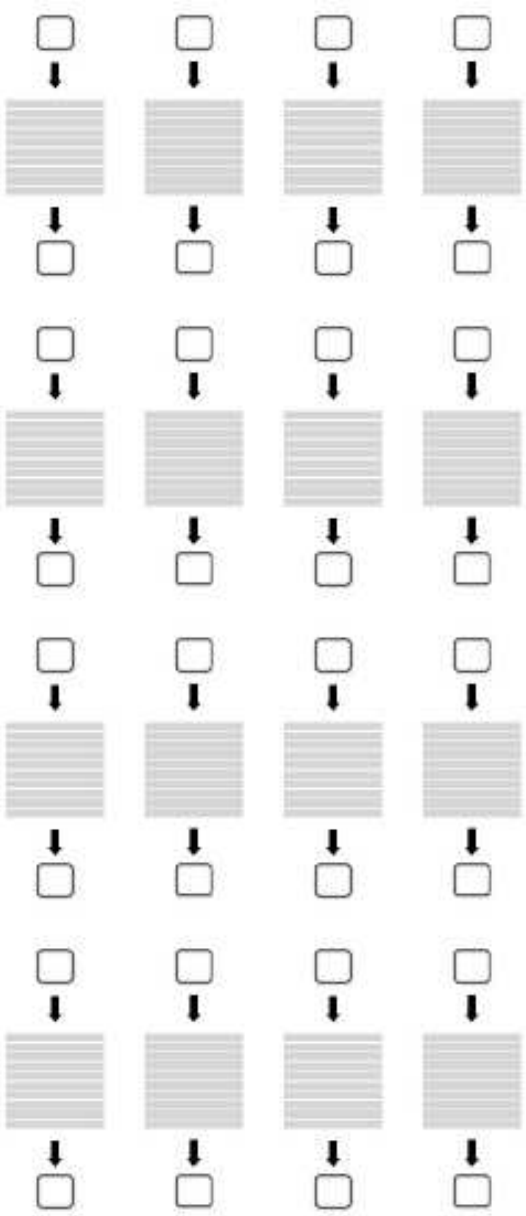
Execute program

My very simple processor: executes one instruction per clock

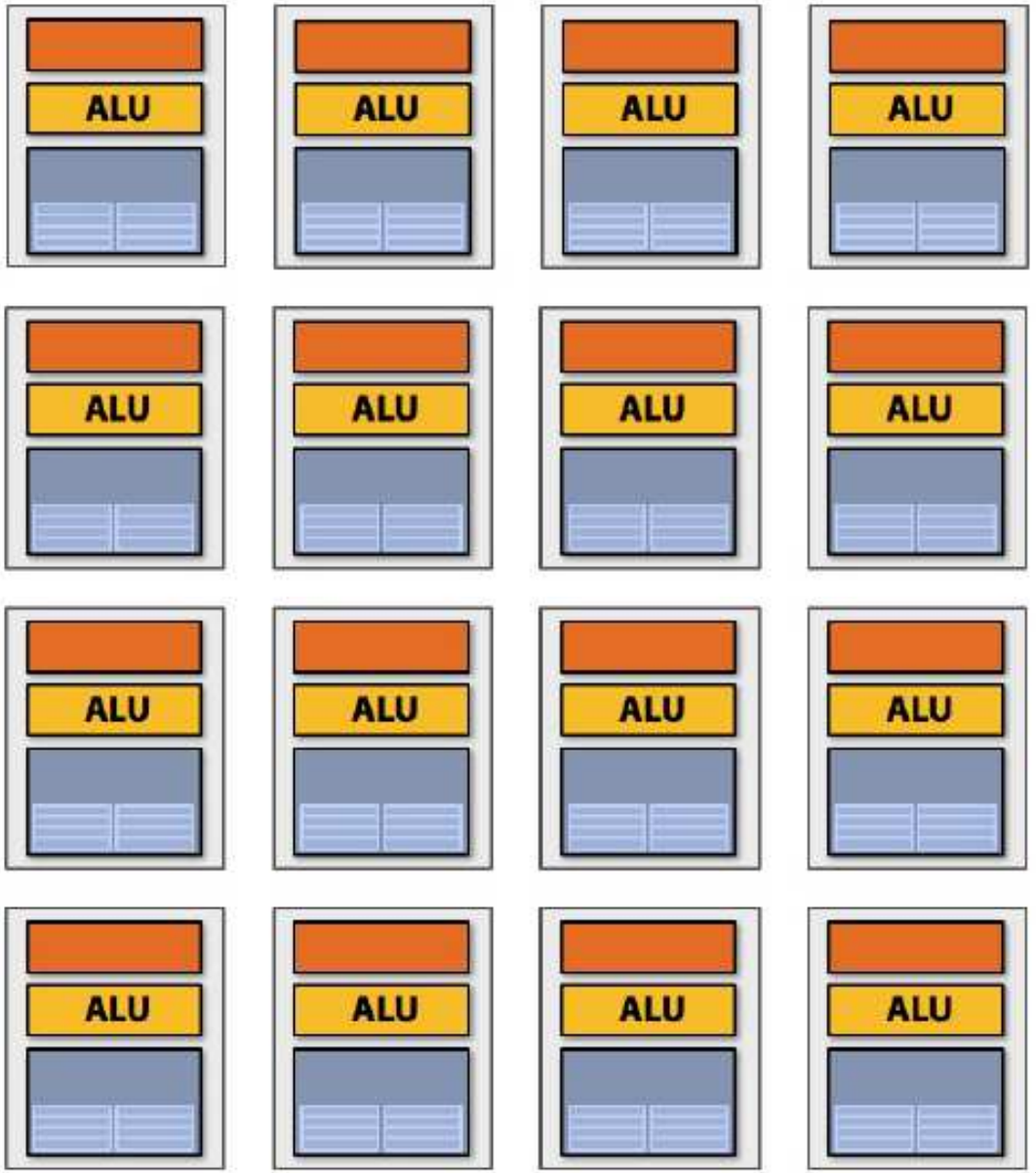




Sixteen cores: process 16 tasks in parallel



Sixteen tasks processed at once

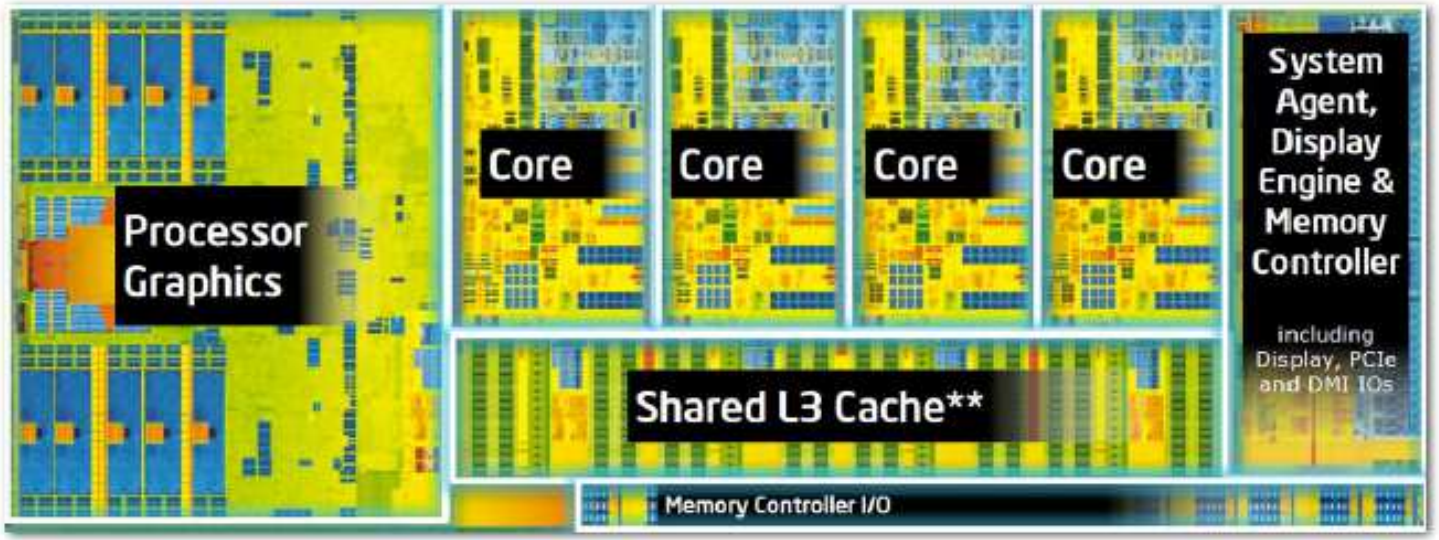


Sixteen cores

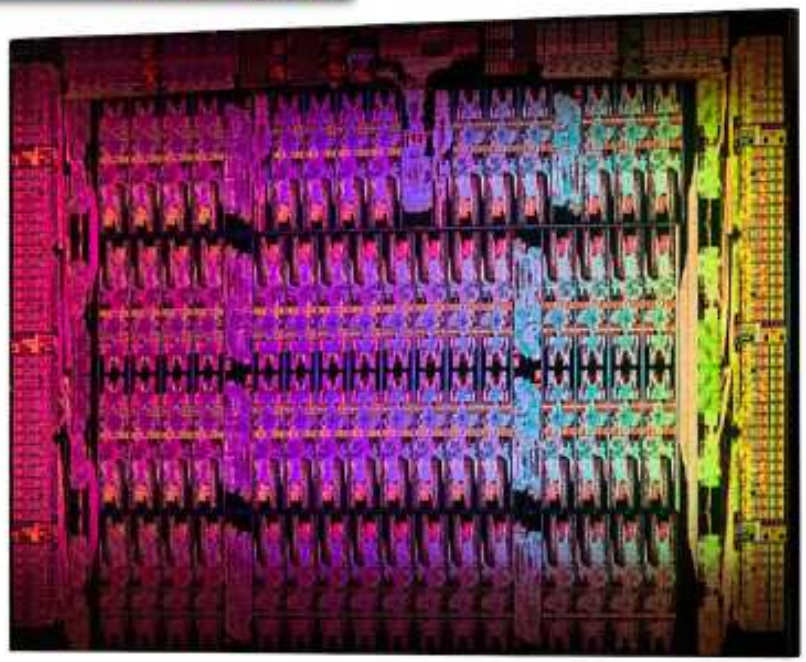
An efficient ray tracer implementation must use all the cores on a modern processor (this is quite easy)

Multi-core processors

Intel Core i7 (Haswell): quad-core CPU



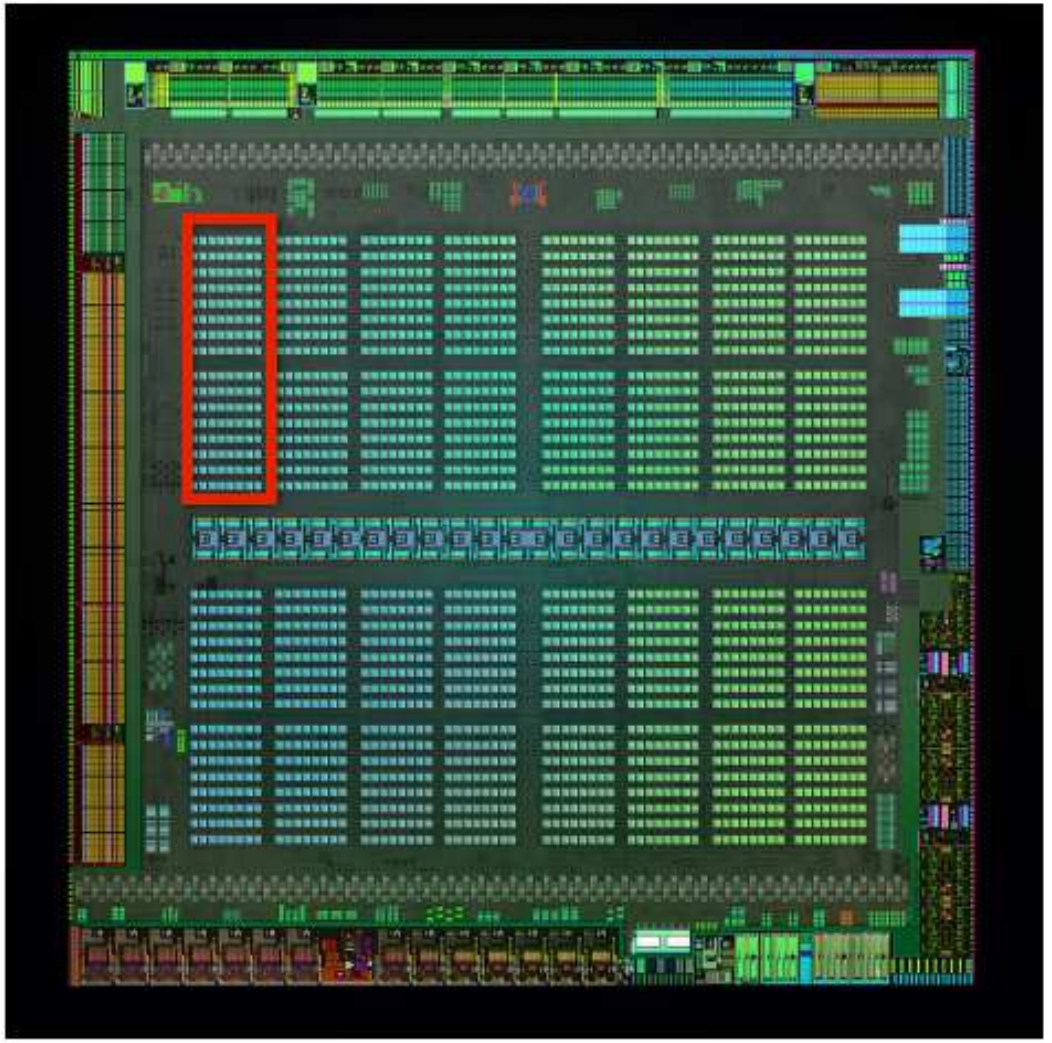
Intel Xeon Phi: 60 core CPU



Multi-core processors

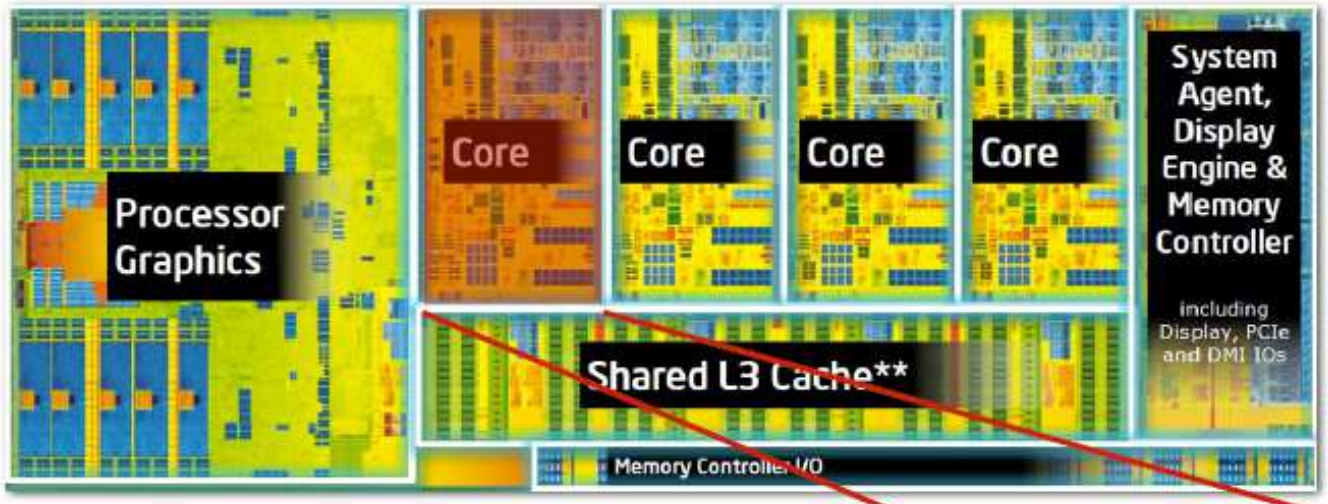
NVIDIA

GeForce GTX 980 (Maxwell) GPU



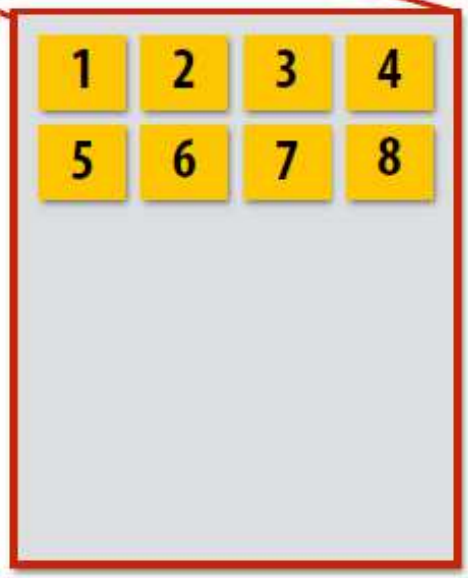
SIMD processing

Single instruction, multiple data



Each core can execute the same instruction simultaneously on multiple pieces of data:

e.g., add vector A to vector B (length 8)
32-bit addition performed in parallel for each vector element.

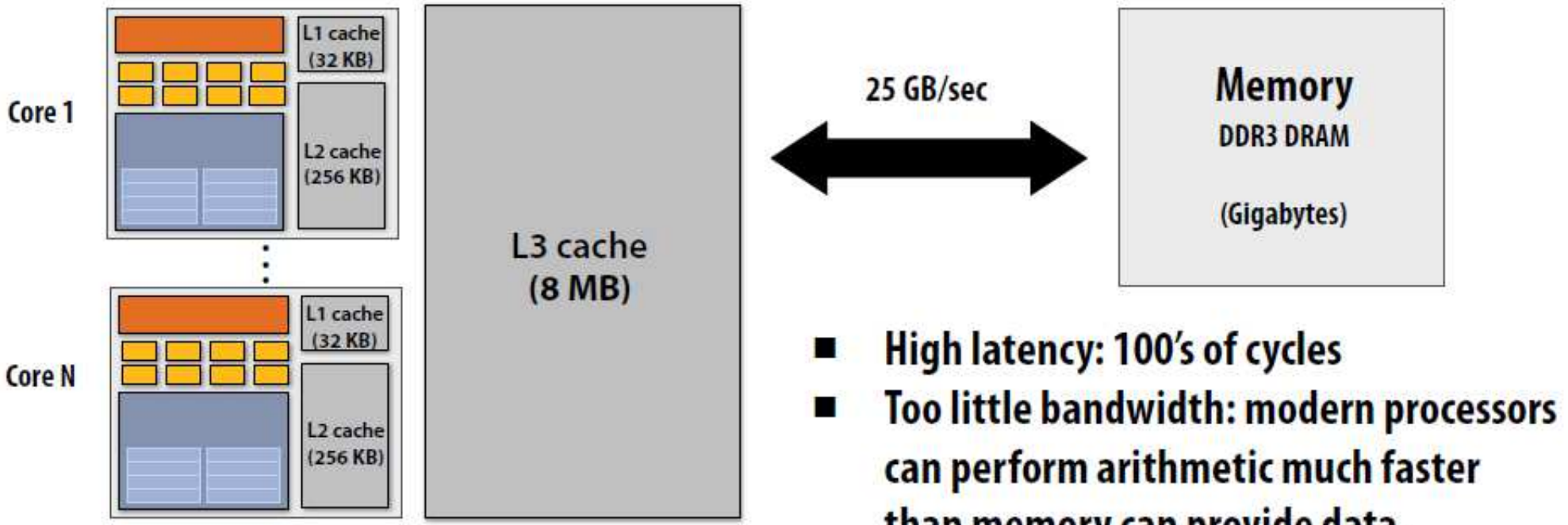


An efficient ray tracer implementation must also utilize the SIMD execution capabilities of modern processors

CPUs: up to a factor of 8

GPUs: up to a factor of 32

Accessing memory has high cost



- High latency: 100's of cycles
- Too little bandwidth: modern processors can perform arithmetic much faster than memory can provide data.

Product	Peak flops (fp)	DRAM Bandwidth (compute/BW ratio)
NVIDIA Tesla K40 (2014, high-end discrete GPU)	4.4 TF (fp32)	288 GB/sec (8.75 flops/byte)
NVIDIA Tegra X1 (2015, high-end mobile GPU)	512 GF (fp32) 1 TF (fp16)	25.6 GB/sec (10 flops/byte)
Imagination PowerVR GX6450 (2014, iPhone 6)	115.2 GF (fp32)	12.4 GB/sec (9.2 flops/byte)
Imagination PowerVR GT7900 (2015, high-end mobile GPU)	384 GF (fp32) 768 GF (fp16)	TBD since processor is not in shipping SoCs
Intel Integrated HD 350 GPU (2014, integrated desktop GPU)	384 GF (fp32)	34.1 GB/sec (11.2 flops/byte)
Xbox 360 (2005)	240 GF (fp32)	32.1 GB/sec (7.5 flops/byte)

An efficient ray tracer implementation must be careful to reduce memory access costs as much as possible.

Rules of the game

- Many individual processor cores
  - Run tasks in parallel
- SIMD instruction capability
  - Single instruction carried out on multiple elements of an array in parallel (8-wide on modern GPUs, 16-wide on Xeon Phi, 8-to-32-wide on modern GPUs)
- Accessing memory is expensive
  - Processor must wait for data to arrive
  - Role of CPU caches is to reduce wait time (want good locality)

High-throughput ray tracing

- **Want work-efficient algorithms (do less)**
  - High-quality acceleration structures (minimize ray-box, ray-primitive tests)
  - Smart traversal algorithms (early termination, etc.)

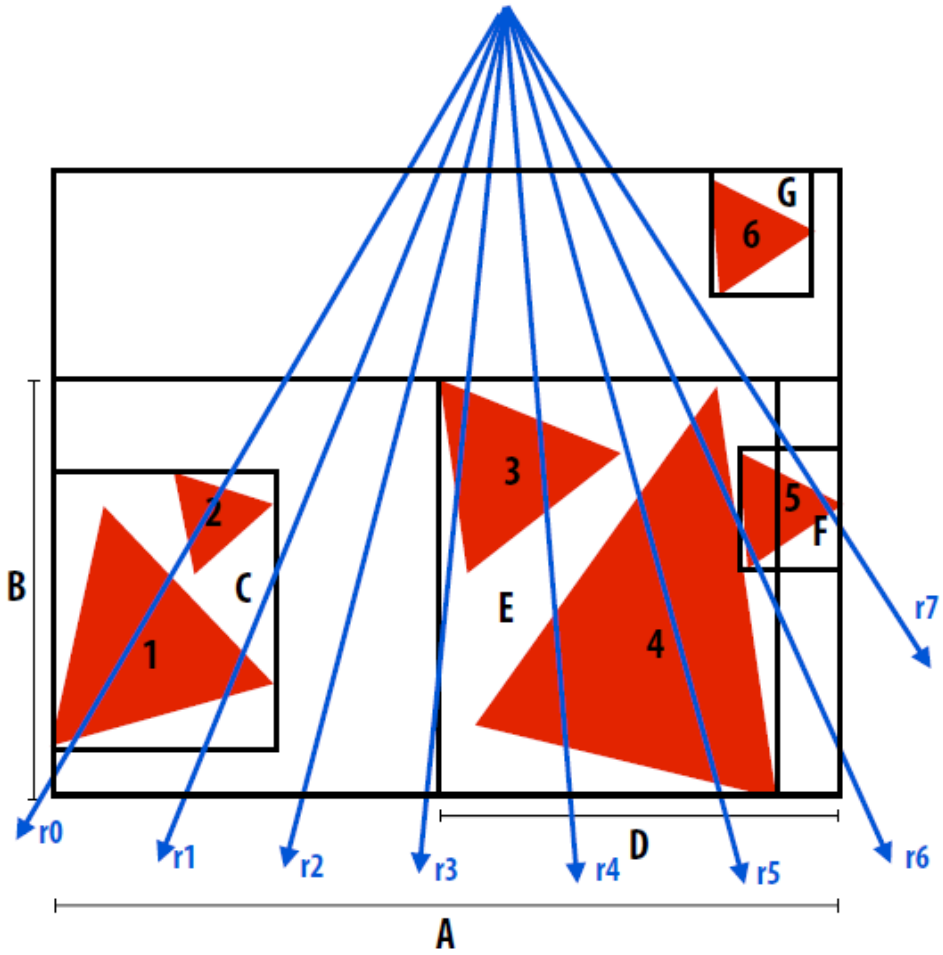
  Discussed in earlier lecture

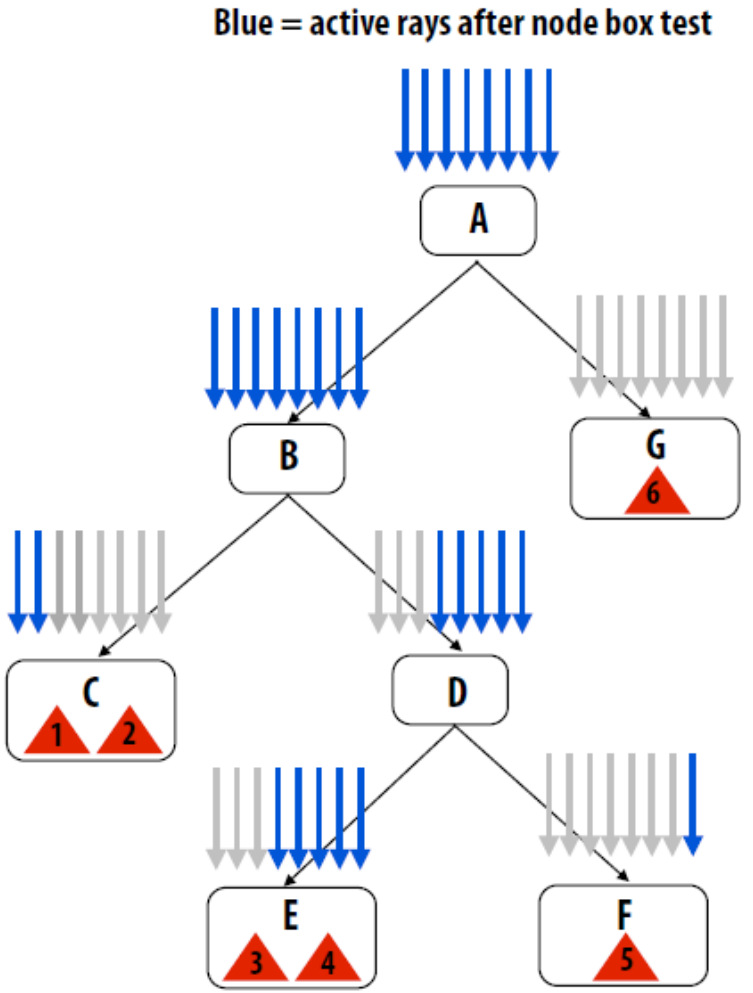
- **Implementations for existing parallel hardware (CPUs/GPUs):**
  - High multi-core, SIMD execution efficiency
  - Help from fixed-function processing?

- **Bandwidth-efficient implementations:**
  - How to minimize bandwidth requirements?

Parallelize across rays

- **Ray packet tracing**

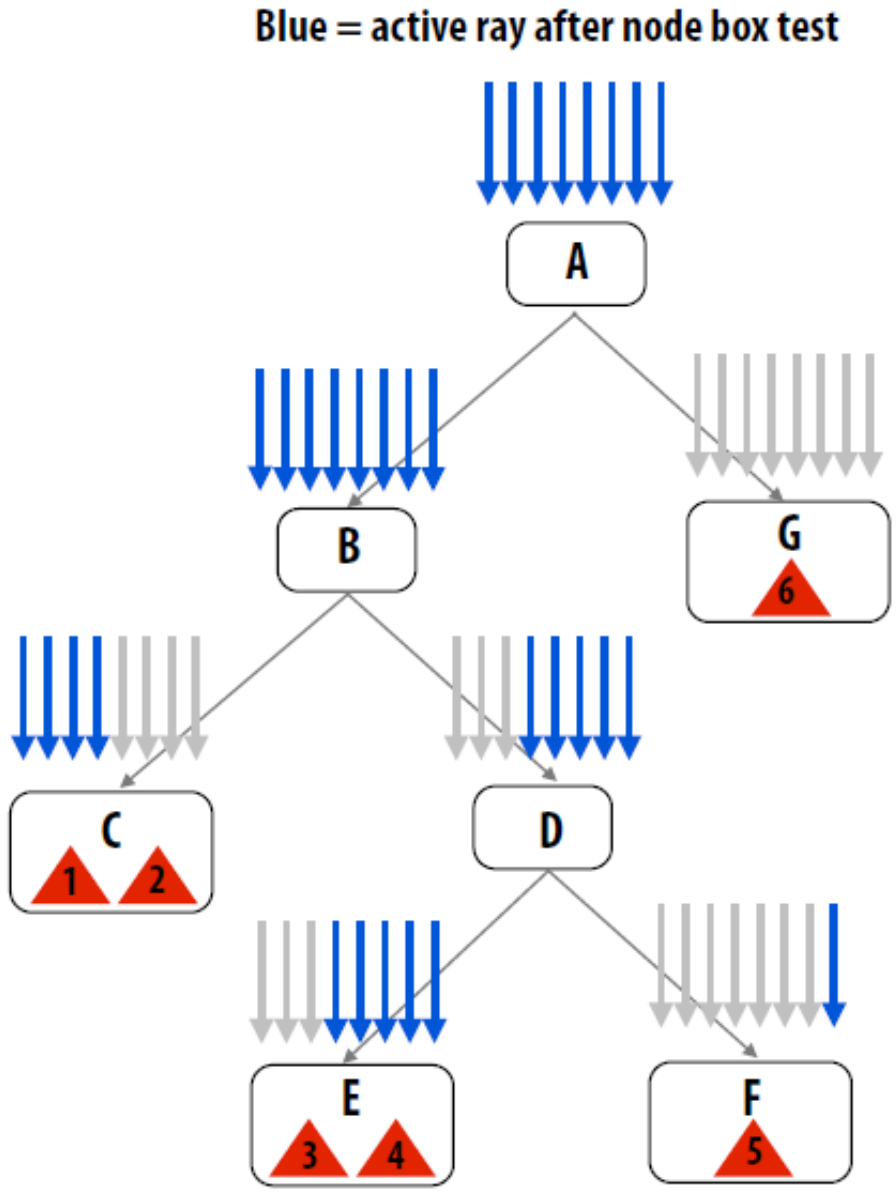




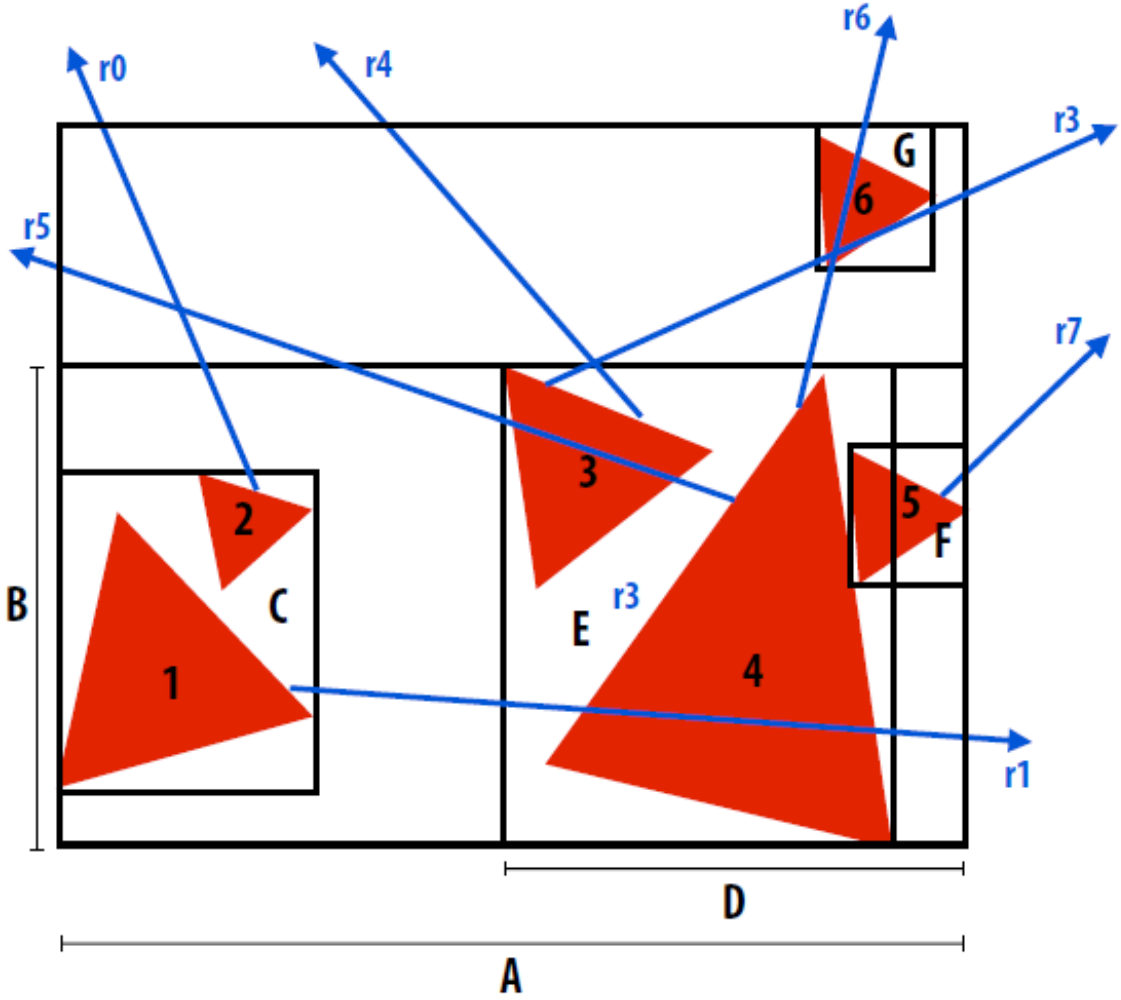
Note: r6 does not pass node F box test due to closest-so-far check, and thus does not visit F

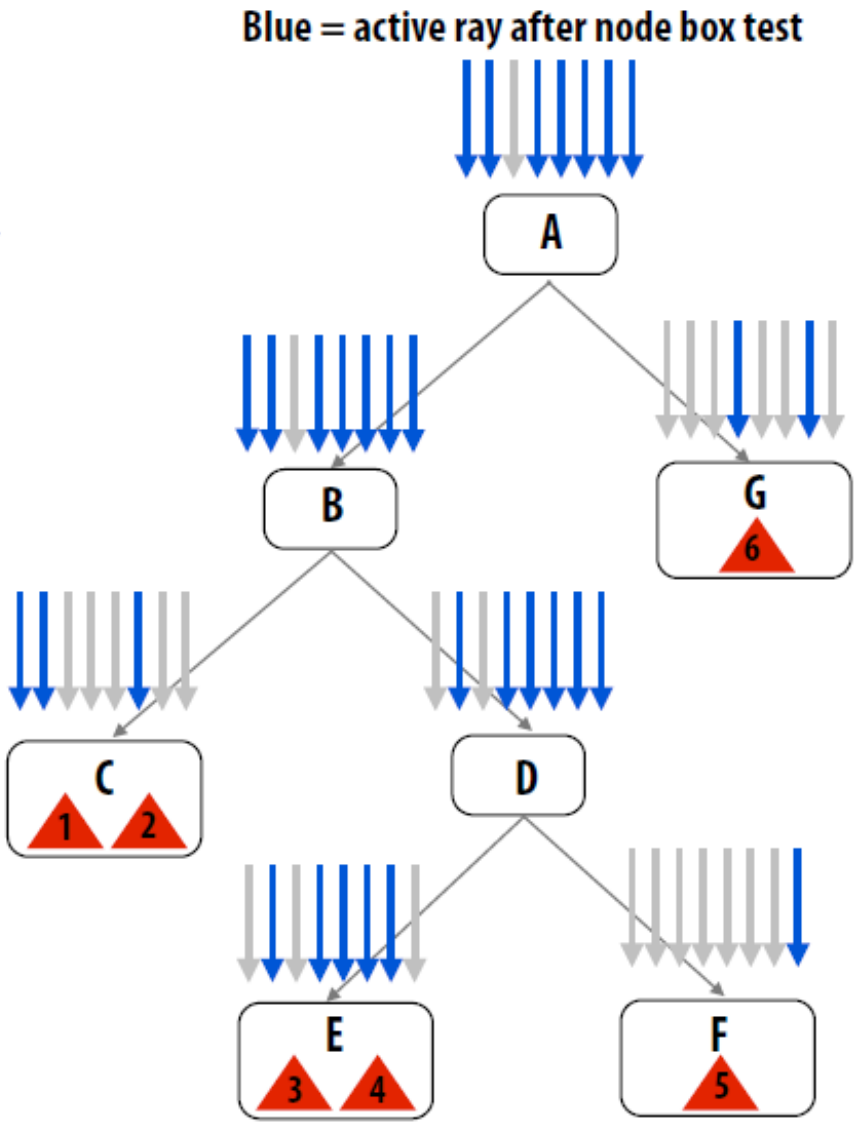
Disadvantages of packets

- If any ray must visit a node, it drags all rays in the packet along with it)
- Loss of efficiency: node traversal, intersection, etc. amortized over less than a packet's worth of rays
- Not all SIMD lanes doing useful work



Ray packet tracing: incoherent rays



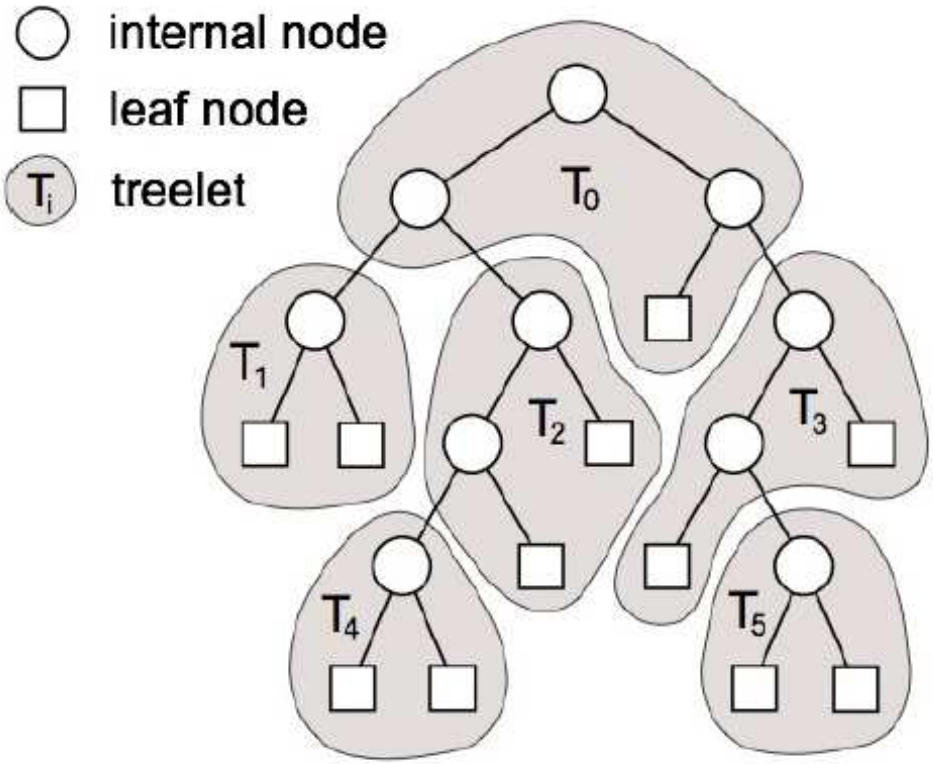


When rays are incoherent, benefit of packets can decrease significantly. This example: packet visits all tree nodes. (So all eight rays visit all tree nodes! No culling benefit!)

Global ray reordering

[Pharr 1997, Navratil 07, Alia 10]

Idea: dynamically batch up rays that must traverse the same part of the scene. Process these rays together to increase locality in BVH access



Partition BVH into treelets
(treelets sized for L1 or L2 cache)

1. When ray (or packet) enters treelet, add rays to treelet queue

2. When treelet queue is sufficiently large, intersect enqueued rays with treelet (amortize treelet load over all enqueued rays)

Buffering overhead to global ray reordering: must store per-ray "stack" (need not be entire call stack, but must contain traversal history) for many rays.

Per-treelet ray queues sized to fit in caches (or in dedicated ray buffer SRAM)

[Pharr 1997, Navratil 07, Alia 10]

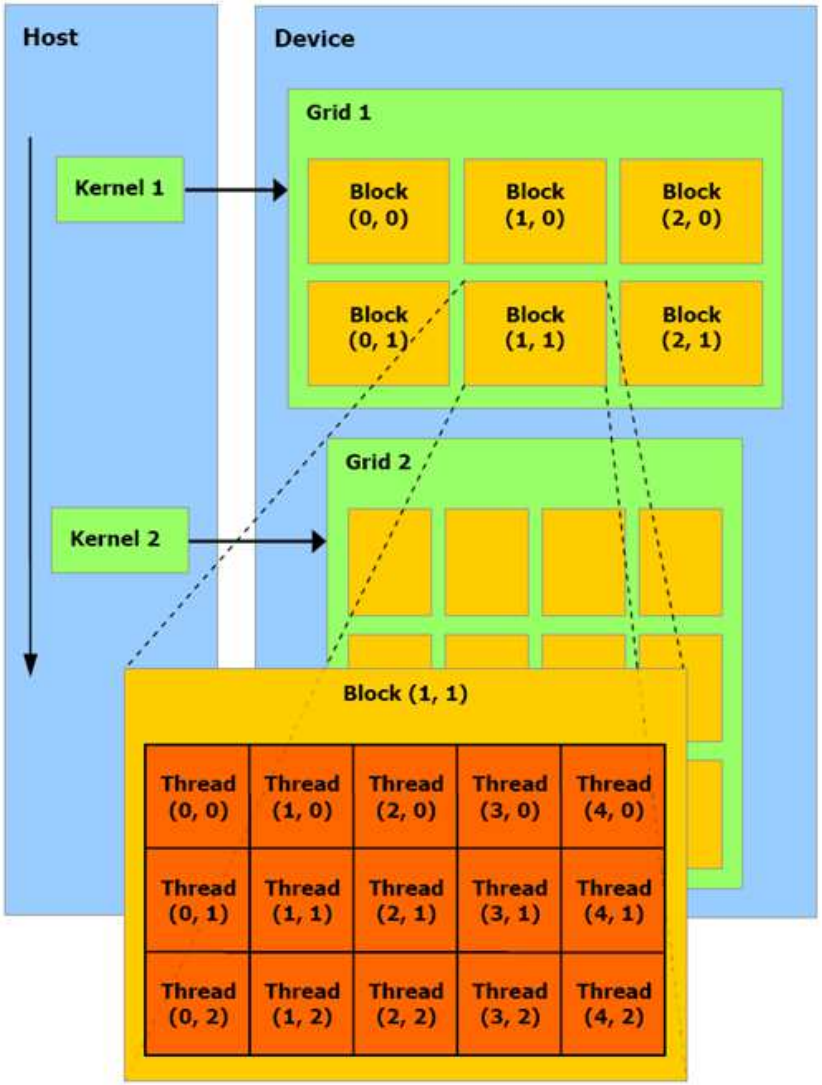
5. CUDA implementation

What is CUDA?

- A parallel computing platform inside NVIDIA GPU
- An application programming interface (API) model
- A software environment that allows developers to use C/C++ as a high-level programming language

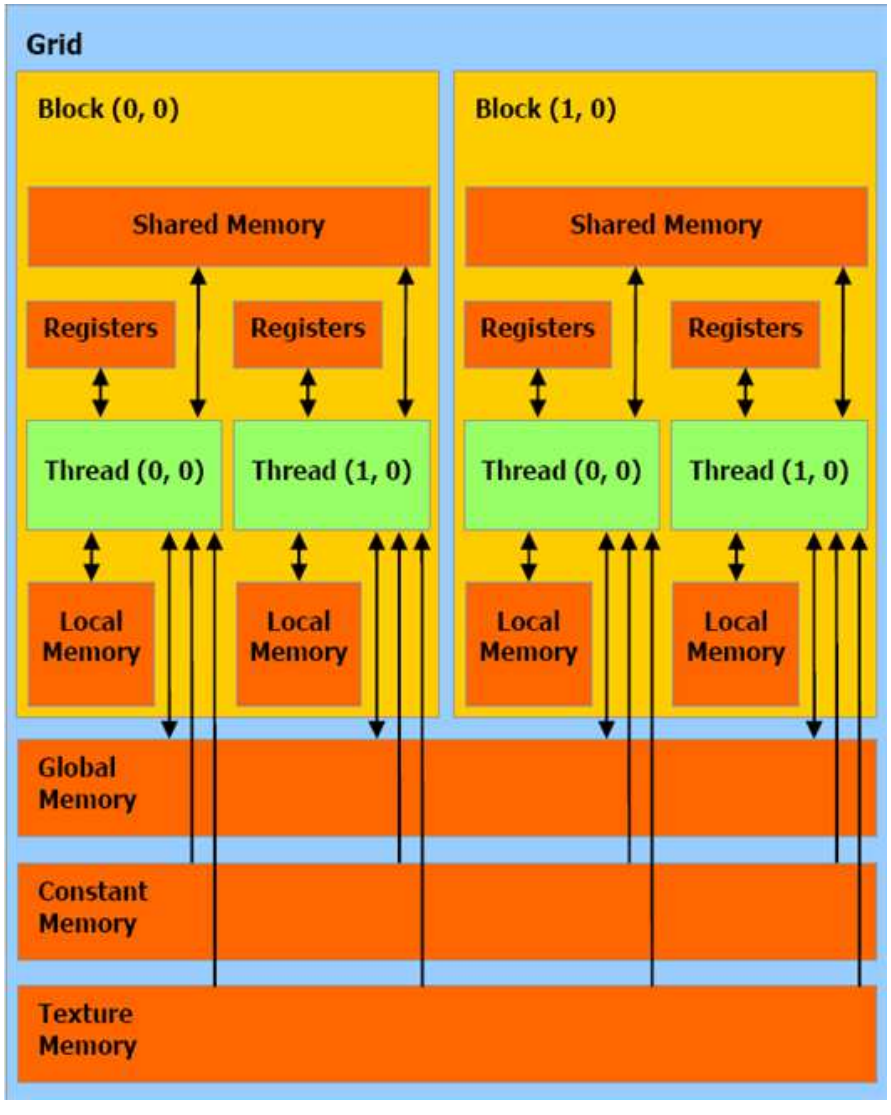
CUDA basics

- **Structure of CUDA program**



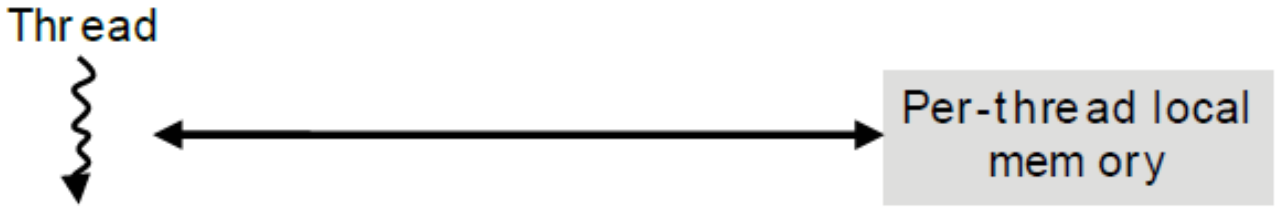
CUDA memory

- **CUDA memory structure**

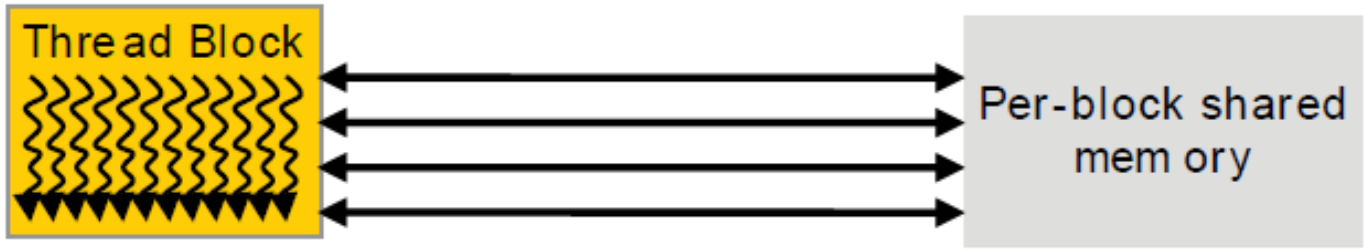


Memory hierarchy

- **Single thread**

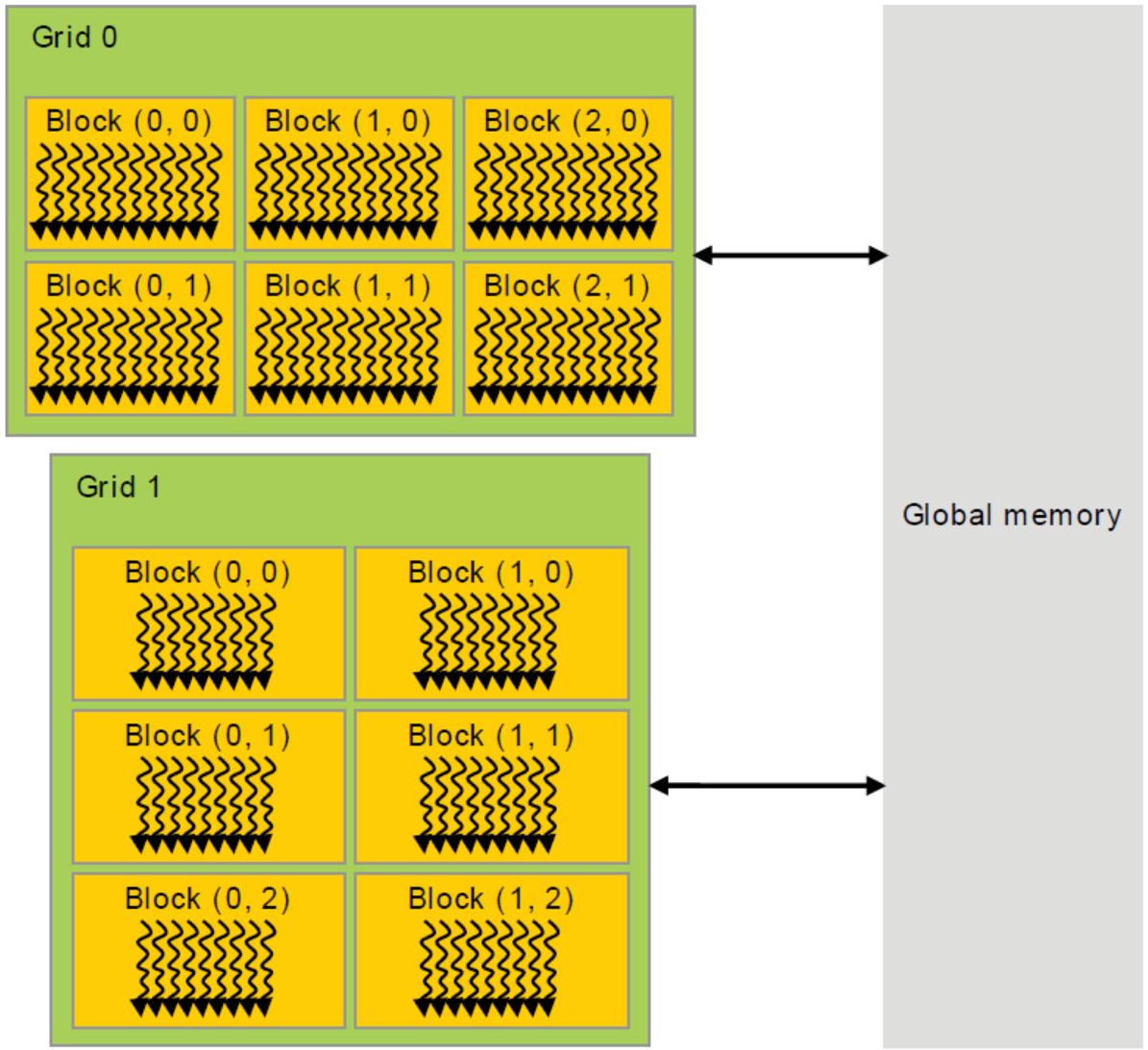


- **Single block**



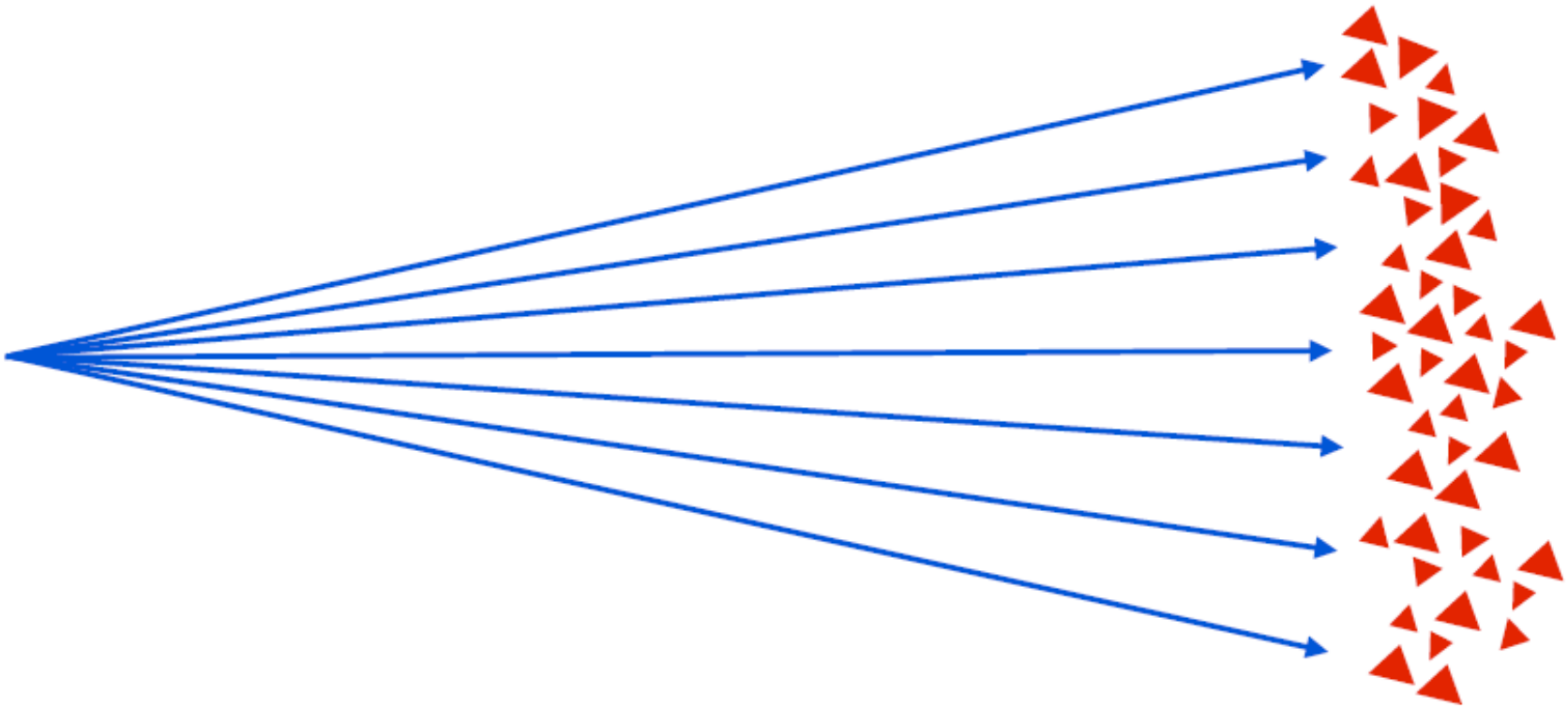
Memory hierarchy

- **Multiple grids/blocks**



CUDA implementation for ray tracing

- **CUDA ray tracing**
  - Tree stored in a linear array in global memory
  - Shooting rays in parallel and perform calculations in each CUDA kernel

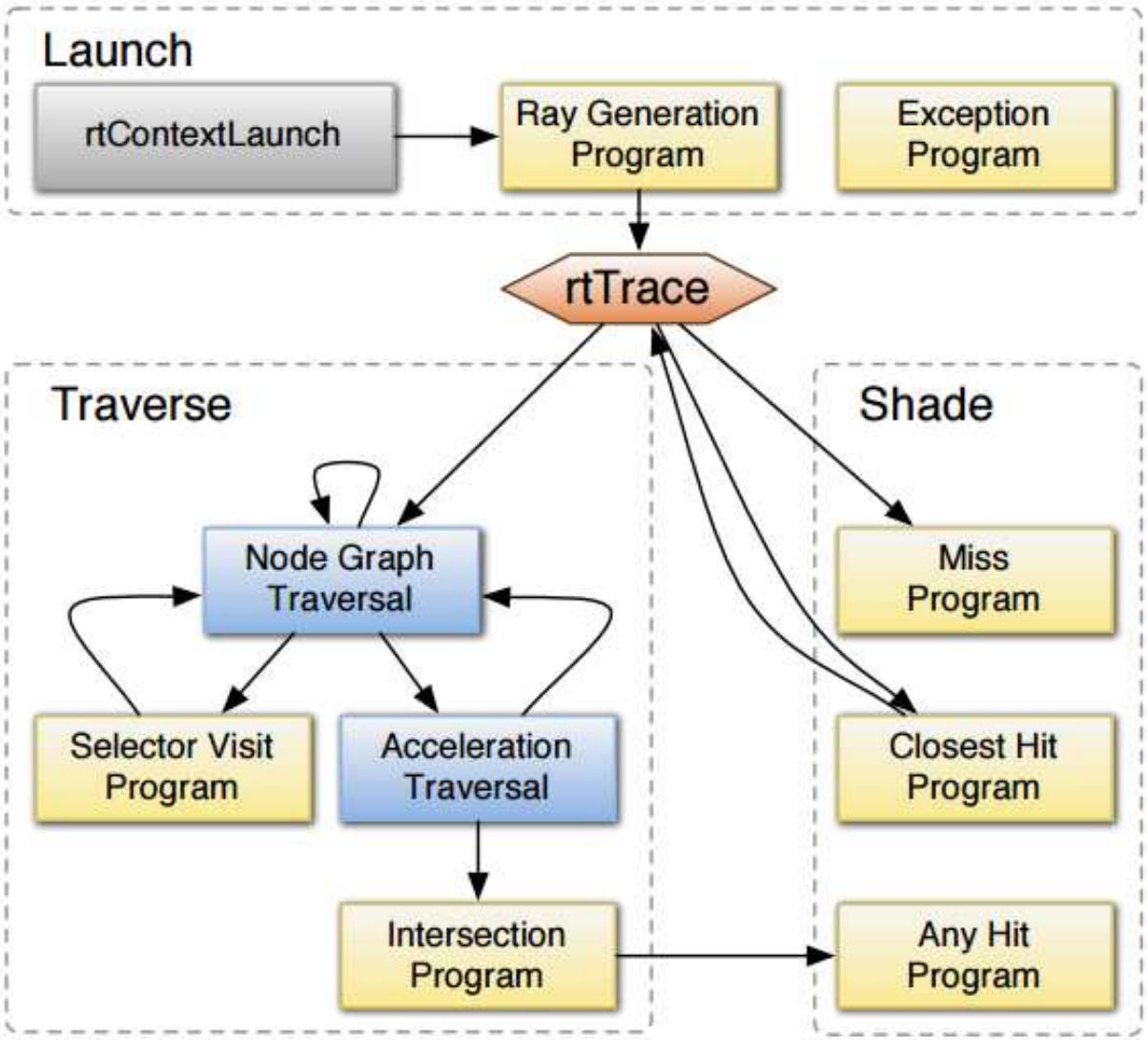


OptiX: NVidia parallel ray-tracing engine

- **The NVIDIA® OptiX™ ray tracing engine is a programmable system designed for NVIDIA GPUs**

OptiX: NVidia parallel ray-tracing engine

- **OptiX call structure**



Next lecture: Numerical integration