Computer Graphics
- Ray-Tracing II -

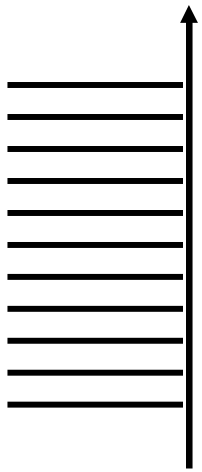
Philipp Slusallek

Overview

- **Last lecture**
  - Ray tracing I
    - Basic ray tracing
    - What is possible?
    - Recursive ray tracing algorithm
    - Intersection computations
- **Today**
  - History of intersection algorithms
  - Advanced acceleration structures
    - Theoretical Background
    - Hierarchical Grids, kd-Trees, Octrees
    - Bounding Volume Hierarchies
  - Dynamic changes to scenes
  - Ray bundles
- **Next lecture**
  - Realtime ray tracing

Theoretical Background

- **Unstructured data results in (at least) linear complexity**
  - Every primitive could be the first one intersected
  - Must test each one separately
  - Coherence does not help
- **Reduced complexity only through pre-sorted data**
  - Spatial sorting of primitives (indexing like for data base)
    - Allows for efficient search strategies
  - Hierarchy leads to O(log n) search complexity
    - But building the hierarchy is still O(n log n)
  - Trade-off between run-time and building-time
    - In particular for dynamic scenes
  - Worst case scene is still linear !!
- **It is a general problem in graphics**
  - Spatial indices for ray tracing
  - Spatial indices for occlusion- and frustum-culling
  - Sorting for transparency



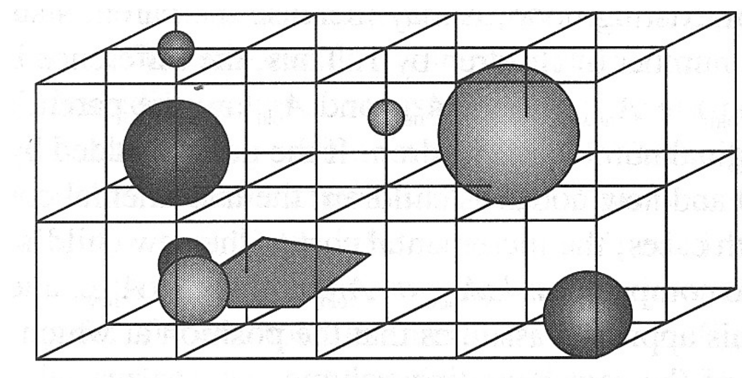
Worst case RT scene: Ray barely misses every primitive

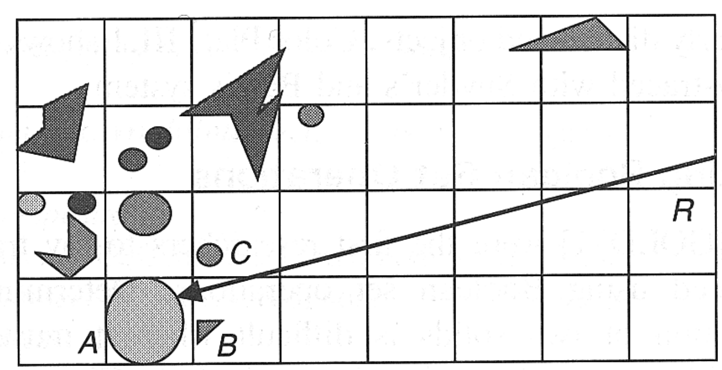
Ray Tracing Acceleration

- **Intersect ray with all objects**
  - Way too expensive
- **Faster intersection algorithms**
  - Little effect (but efficient algorithms are still necessary)
- **Less intersection computations**
  - Space partitioning (often hierarchical)
    - Grid, hierarchies of grids
    - Octree
    - Binary space partition (BSP) or kd-tree
    - Bounding volume hierarchy (BVH)
  - Directional partitioning (not very useful)
  - 5D partitioning (space and direction, once a big hype)
    - Close to pre-compute visibility for all points and all directions
- **Tracing of continuous bundles of rays**
  - Exploits coherence of neighboring rays, amortize cost among them
    - Cone tracing, beam tracing, ...

Grid

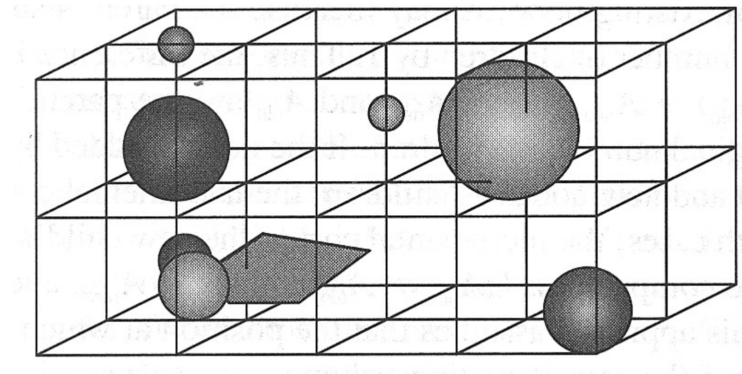
- **Grid**
  - Partitioning with equal, fixed sized „voxels“
- **Building a grid structure**
  - Partition the bounding box (bb)
  - Resolution: often $\sqrt[3]{n}$
  - Inserting objects
    - Trivial: insert into all voxels overlapping objects bounding box
    - Easily optimized
- **Traversal**
  - Iterate through all voxels in order as pierced by the ray
  - Compute intersection with objects in each voxel
  - Stop if intersection found in current voxel

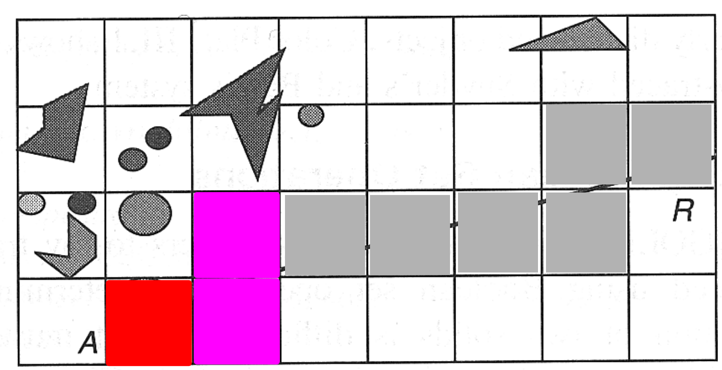




Grid

- **Grid**
  - Partitioning with equal, fixed sized „voxels“
- **Building a grid structure**
  - Partition the bounding box (bb)
  - Resolution: often $\sqrt[3]{n}$
  - Inserting objects
    - Trivial: insert into all voxels overlapping objects bounding box
    - Easily optimized
- **Traversal**
  - Iterate through all voxels in order as pierced by the ray
  - Compute intersection with objects in each voxel
  - Stop if intersection found in current voxel



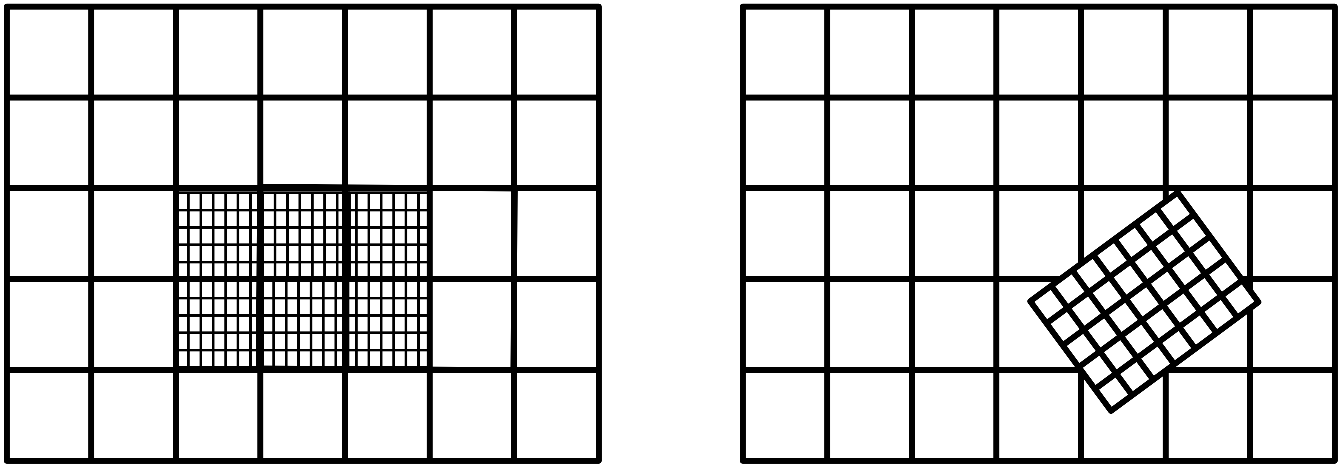


Grid: Issues

- **Grid traversal**
  - Requires enumeration of voxel along ray
    → 3D-DDA, modified Bresenham (later)
  - Simple and hardware-friendly
- **Grid resolution**
  - Strongly scene dependent
  - Cannot adapt to local density of objects
    - Problem: „Teapot in a stadium“
  - Possible solution: grids within grids ➔ hierarchical grids
- **Objects spanning multiple voxels**
  - Store only references to objects
  - Use mailboxing to avoid multiple intersection computations
    - Store object in small per-ray cache (e.g. with hashing)
    - Do not intersect again if found in cache
  - Original mailbox stores ray-id with each triangle
    - Simple, but likely to destroy CPU caches

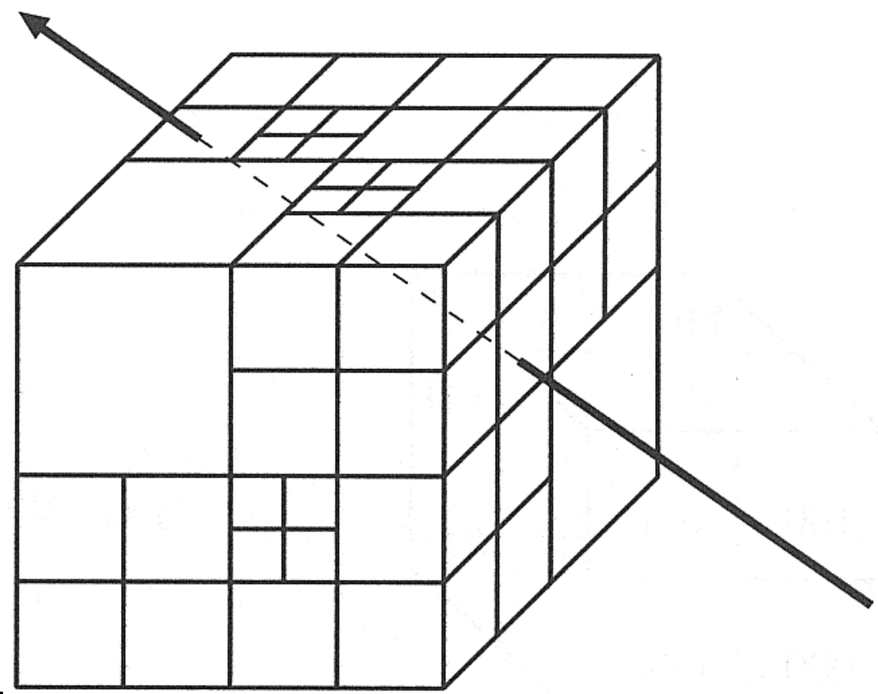
Hierarchical Grids

- **Simple building algorithm**
  - Coarse grid for entire scene
  - Recursively create grids in high-density voxels
  - Problem: What is the right resolution for each level?
- **Advanced algorithm**
  - Place cluster of objects in separate grids
  - Insert these grids into parent grid
  - Problem: What are good clusters?



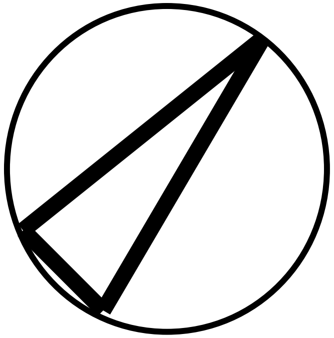
Octree

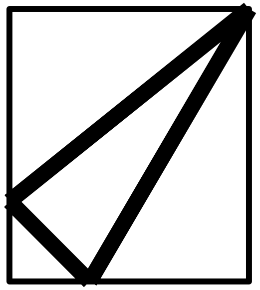
- **Hierarchical space partitioning**
  - Start with bounding box of entire scene
  - Recursively subdivide voxels into 8 equal sub-voxels
  - Subdivision criteria:
    - Number of remaining primitives and maximum depth
  - Result in adaptive subdivision
    - Allows for large traversal steps in empty regions
- **Problems**
  - Pretty complex traversal algorithms
  - Slow to refine complex regions
- **Traversal algorithms**
  - HERO, SMART, ...
  - Or use kd-tree algorithm …

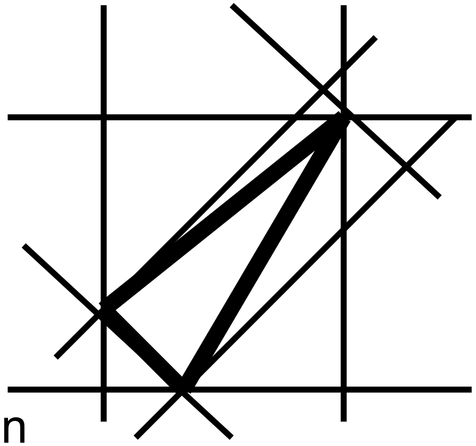


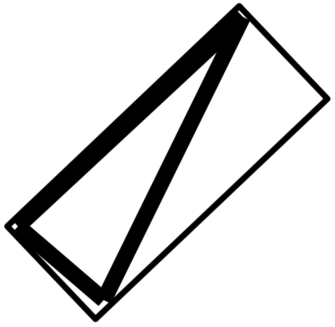
Bounding Volumes (BV)

- **Observation**
  - Bound geometry with BV
  - Only compute intersection if ray hits BV
- **Sphere**
  - Very fast intersection computation
  - Often inefficient because too large
- **Axis-aligned box**
  - Very simple intersection computation (min-max)
  - Sometimes too large
- **Non-axis-aligned box**
  - A.k.a. „oriented bounding box (OBB)"
  - Often better fit
  - Fairly complex computation
- **Slabs**
  - Pairs of half spaces
  - Fixed number of orientations
    - Addition of coordinates w/ negation
  - Fairly fast computation



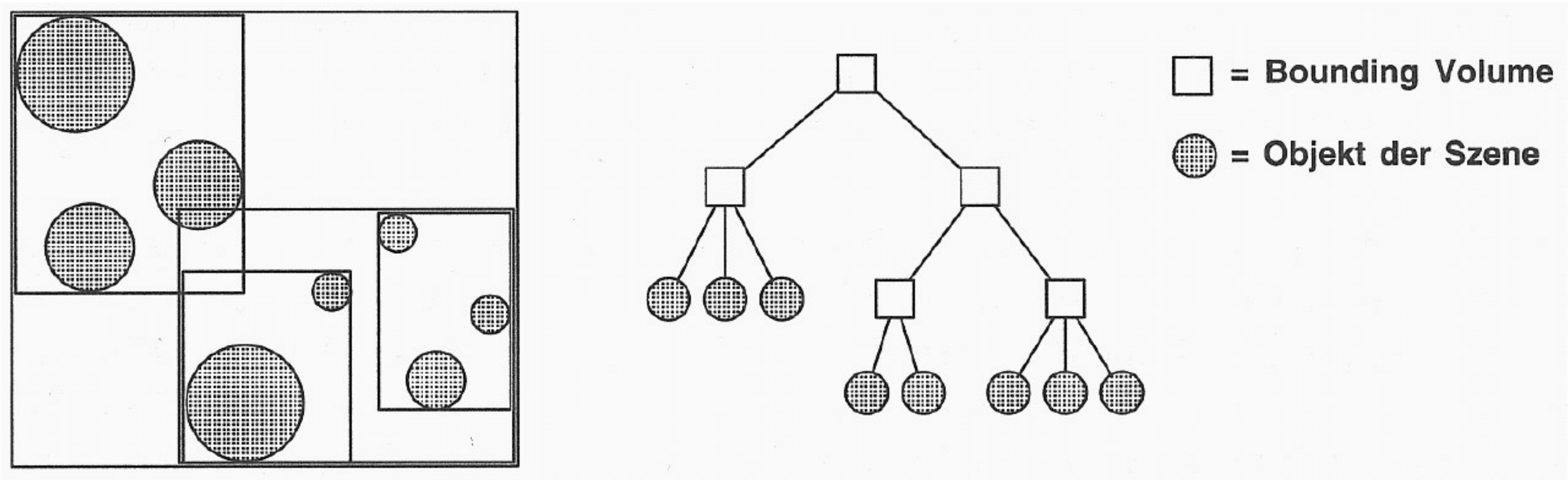






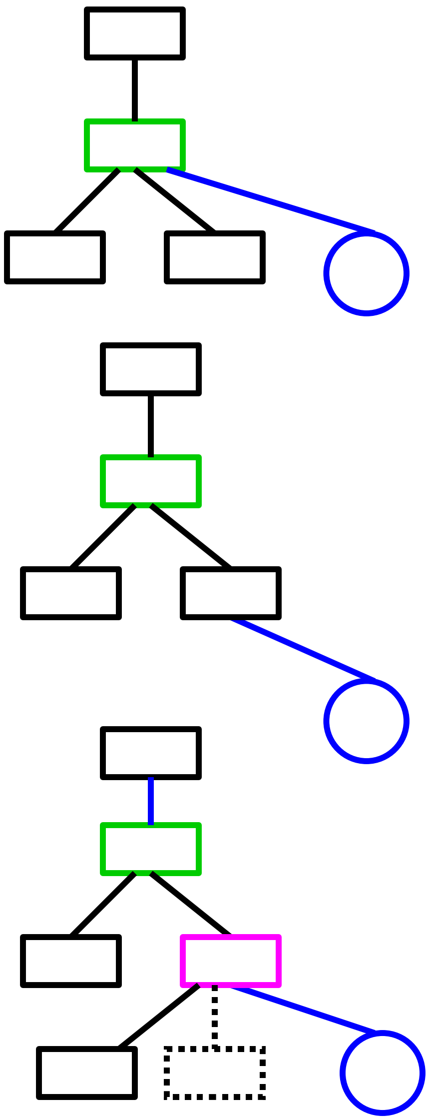
Bounding Volume Hierarchies

- **Idea:**
  - Organize bounding volumes hierarchically into new BVs
- **Advantages:**
  - Very good adaptivity
  - Efficient traversal O(log N)
  - Often used in ray tracing systems
- **Problems**
  - How to arrange BVs?



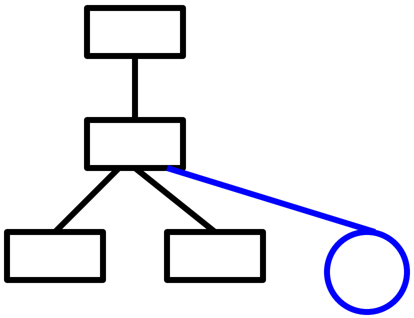
Bounding Volume Hierarchy

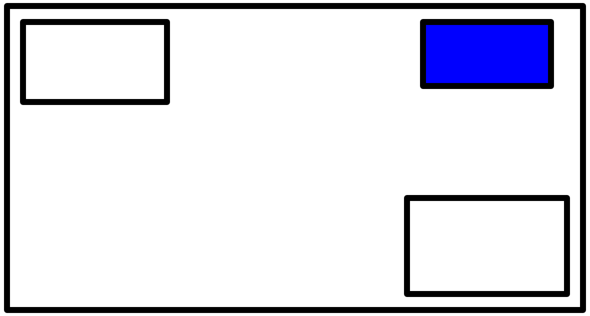
- **Possible building strategy**
  - Manual
  - Given by input structure (e.g. CAD system)
  - Incremental insertion (top-down)
- **Incremental recursive insertion**
  - Algorithm from Goldsmith/Salmon´87
  - Cost function:
    - Surface of object / BVs
    - Cost for intersection with children
    - Local decisions only (otherwise NP-hard)
  - Evaluate cost function for three cases
    - Insert as child in current BV
    - Propagate to some child and recurse
    - Create new BV as child and merge new object with other old children



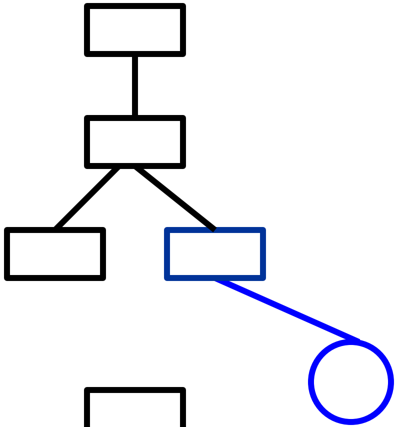
Bounding Volume Hierarchy

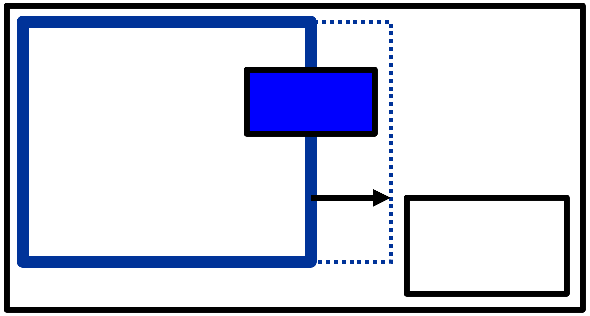
Case 1



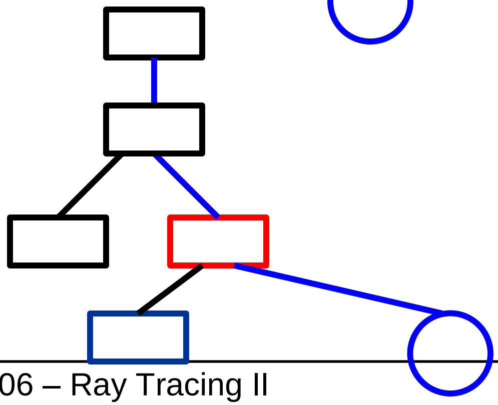


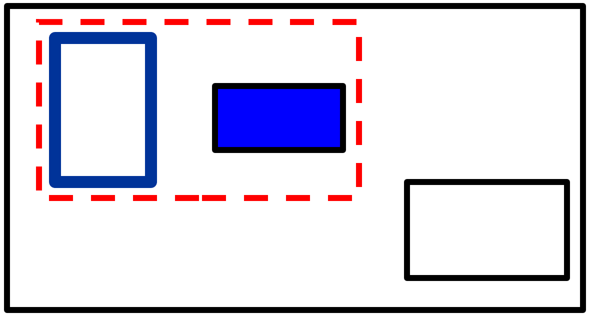
Case 2





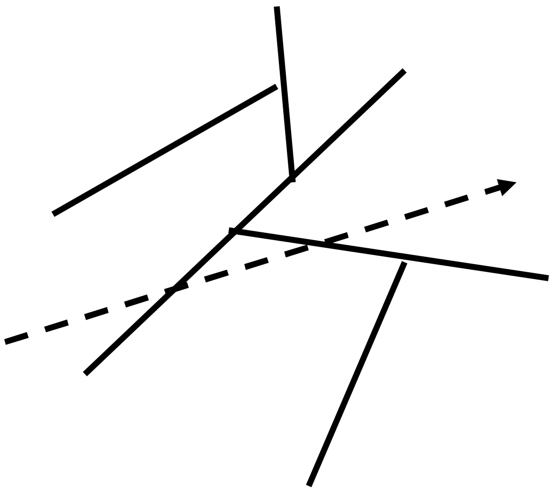
Case 3





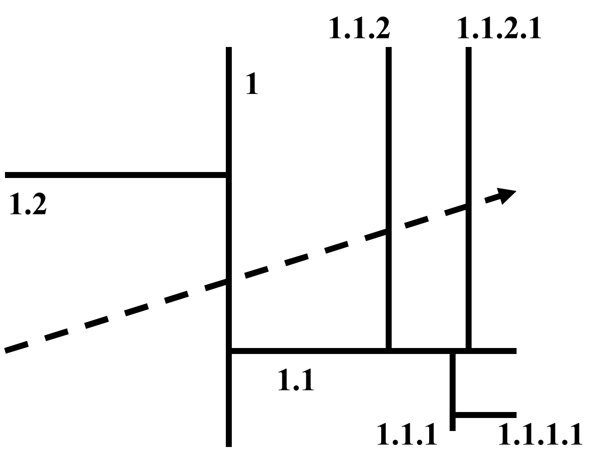
BSP- and Kd-Trees

- **Recursive space partitioning with half-spaces**
- **Binary Space Partition (BSP):**
  - Recursively split space into halves
  - Splitting with half-spaces in arbitrary position
    - Often defined by existing polygons
  - Often used for visibility in games (→ Doom)
    - Traverse binary tree from front to back



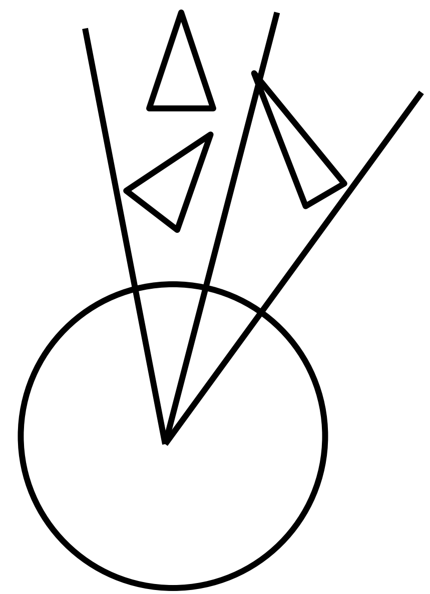
- **Kd-Tree**
  - Special case of BSP
    - Splitting with axis-aligned half-spaces
  - Defined recursively through nodes with
    - Axis-flag
    - Split location (1D)
    - Child pointer(s)

  - See separate slides for details

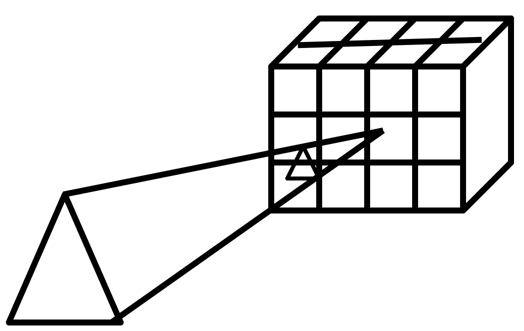


Directional Partitioning

- **Applications**
  - Useful only for rays that start from a single point
    - Camera
    - Point light sources
  - Preprocessing of visibility
  - Requires scan conversion of geometry
    - For each object locate where it is visible
    - Expensive and linear in # of objects
- **Generally not used for primary rays**

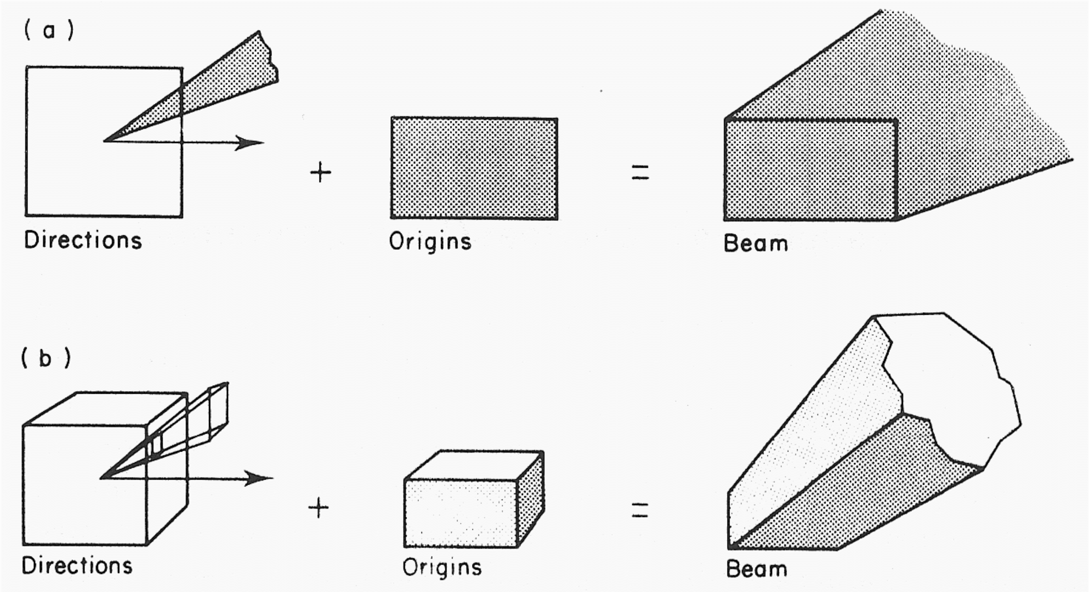


- **Variation: Light buffer**
  - Lazy and conservative evaluation
  - Store occluder that was found in directional structure
  - Test entry first for next shadow test



Ray Classification

- **Partitioning of space and direction [Arvo & Kirk´87]**
  - Roughly pre-computes visibility for the entire scene
    - What is visible from each point in each direction?
  - Very costly preprocessing, cheap traversal
    - Improper trade-off between preprocessing and run-time
  - Memory hungry, even with lazy evaluation
  - Seldom used in practice



Dynamic Scenes

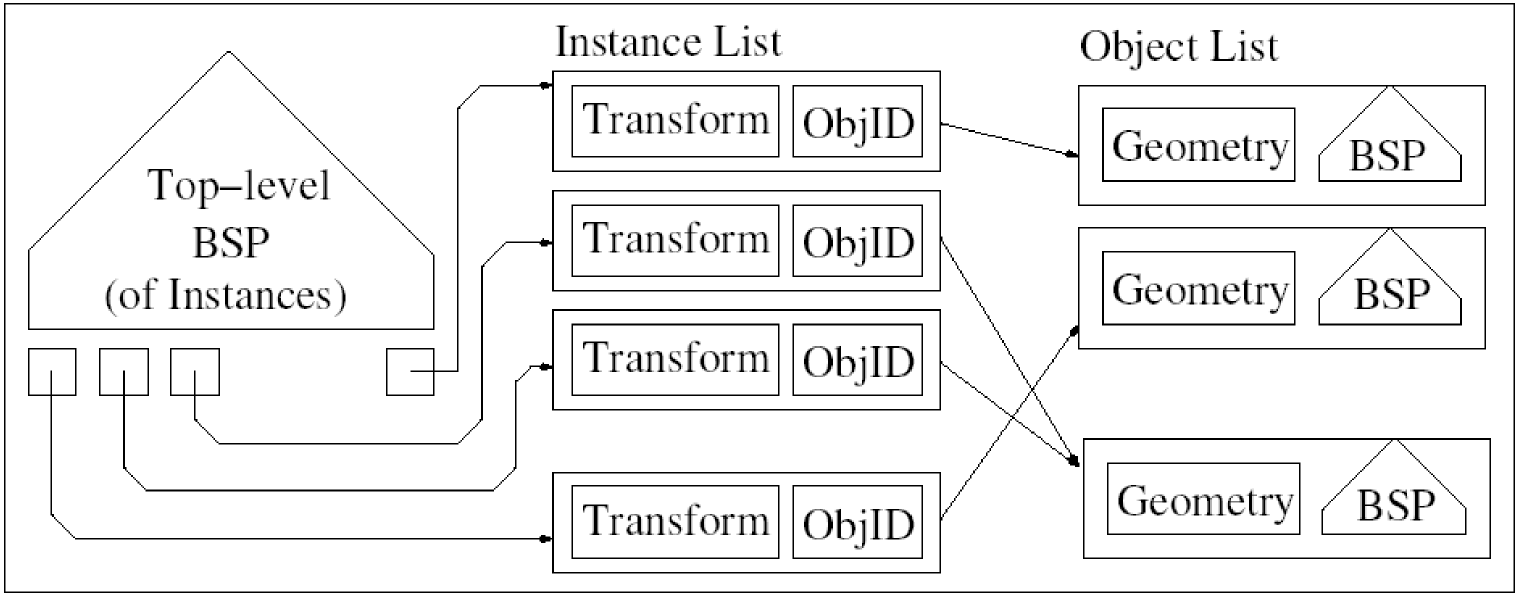
- **Changes to spatial indices**
  - In interactive context
- **Very little research despite general usefulness**
  - Efficient dynamic data structures
    - From computational geometry (i.e. kinetic data structures)
    - Not realtime
  - Animation with predefined motion [Glassner’88, Gröller’91, …]
  - Exclude dynamic primitives [Parker’99]
  - Constant time rebuild [Reinhard’00]
  - Divide and conquer [Lext’00]
- **Different Types of Motion**
  - Hierarchical: Affine transformations for groups of primitives
  - Unstructured: Arbitrary movements of primitives

Divide & Conquer Approach

- **Observation**
  - 80/20 rule: Very often a simple approach is sufficient
  - Building hierarchical index structures requires O(n logn)
    - Divide and conquer reduces complexity
- **Categorize primitives into independent groups/objects**
  - Static parts of a scene (often large parts of a scene)
  - Structured motion (affine transformations)
  - Anything else
- **Select suitable approach for each group**
  - Do nothing
  - Transform rays instead of primitives
  - Only update index structure for relevant groups

Divide & Conquer Approach

- **Two-level index structure**
  - Find relevant objects
  - Transform ray (efficient SSE code)
  - Find primitives in object
  - Same kd-tree traversal algorithms in both cases
- **Results in some run-time overhead**



Implementation

- **KD-tree building algorithms**
  - Static & structured motion
    - Build once with sophisticated and slow algorithm [Havran'01]
    - Optimize for traversal (as low as 1.5 intersection per ray)
  - Unstructured Motion
    - Will be used for single or few frames
    - Balance construction and traversal time
      - Allow more primitives in deeper nodes
  - Top-Level:
    - Significantly more efficient than for primitives
    - Possible splitting planes for kd-tree are already given

Implementation

- **Index Structure Updates**
  - Static: Done
  - Structured Motion
    - Update transformation
    - Schedule update of top-level index
  - Unstructured Motion
    - Rebuild local index and bounding box
    - Schedule top-level update, iff bounding box changed
    - Could be optimized with top-level *hierarchy*
      - Not yet necessary

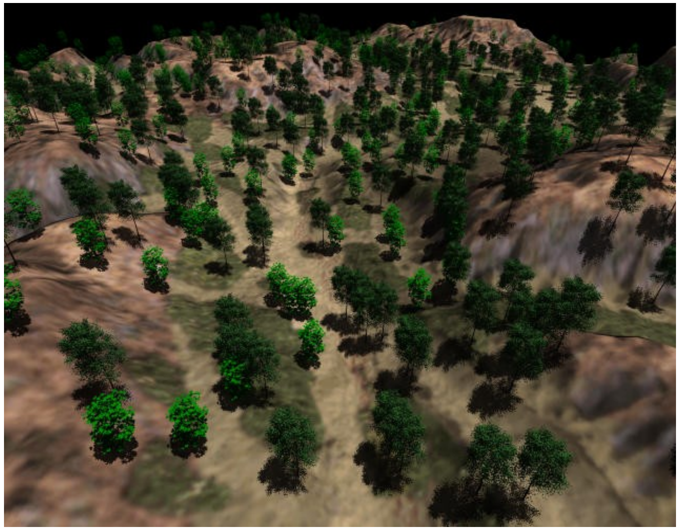
Results

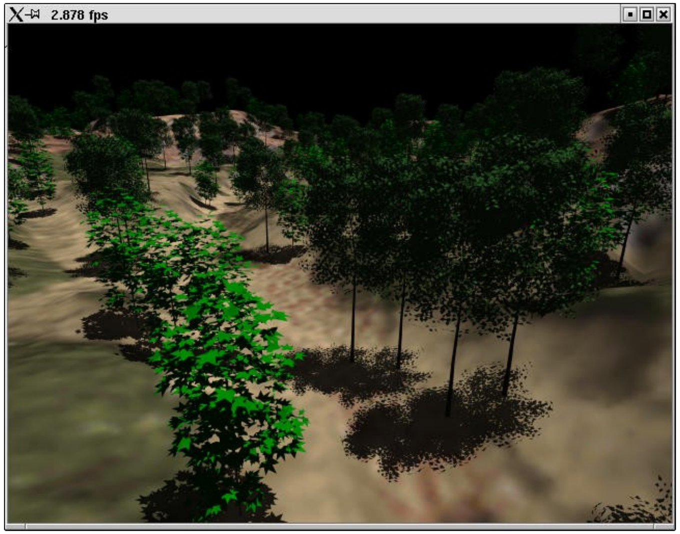
- **BART Kitchen**
  - 110,000 triangles in 5 objects, 6 lights with shadows
  - Little structured motion
  - 3.8 Mrays/frame resulting in 0.9 Mrays per second and CPU
  - Performance (fps)

Shading \ CPUs	2	4	8	16	32
OpenGL-like	3.2	6.4	12.8	25.6	> 26
Ray Tracing	0.47	0.94	1.88	3.77	7.55

Results

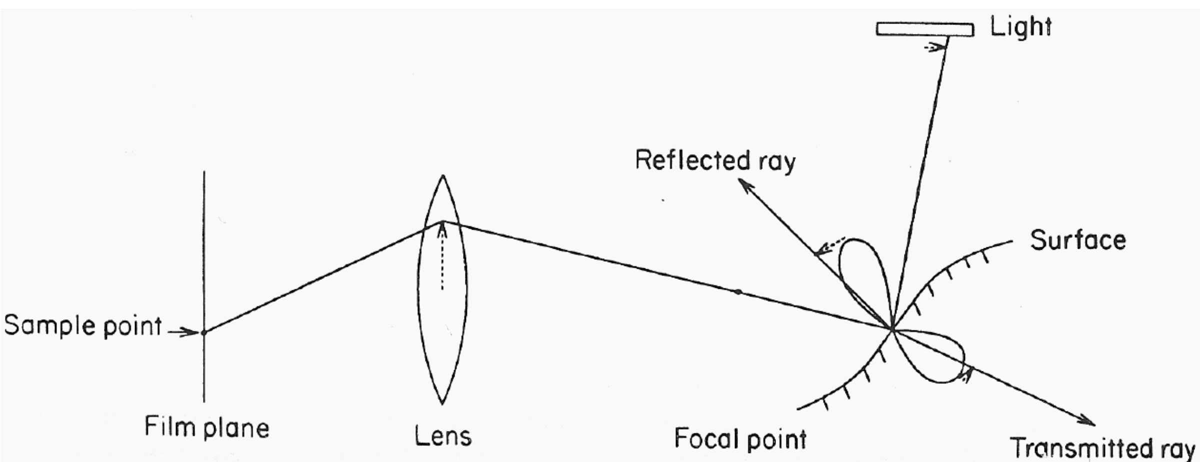
- **Outdoor Terrain**
  - 661 objects, total of 10 Mtris
  - Single point light source
  - Accurate shadows between leaves
  - Interactive translation of all trees
  - Performance
    - Update for top-level kd-tree: 4ms

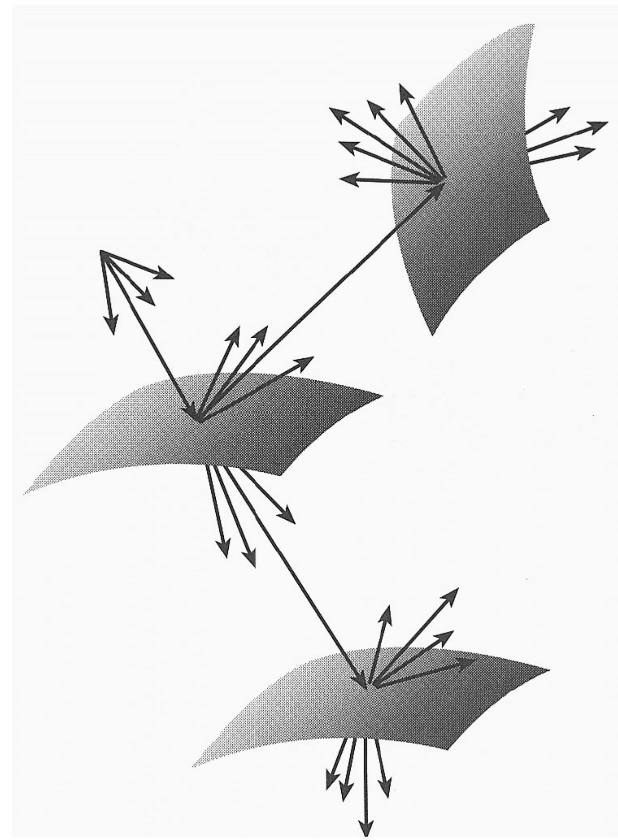




Distribution Ray Tracing

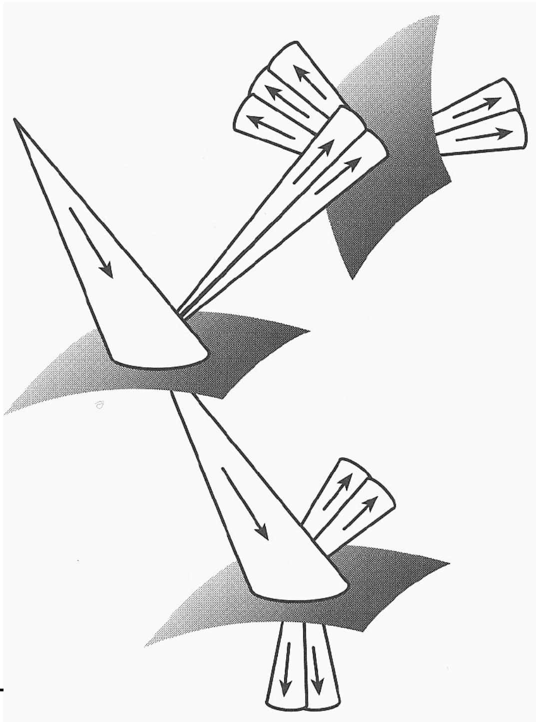
- **Formerly called Distributed Ray Tracing [Cook`84]**
- **Stochastic Sampling of**
  - Pixel: Antialiasing
  - Lens: Depth-of-field
  - BRDF: Glossy reflections
  - Lights: Smooth shadows from area light sources
  - Time: Motion blur



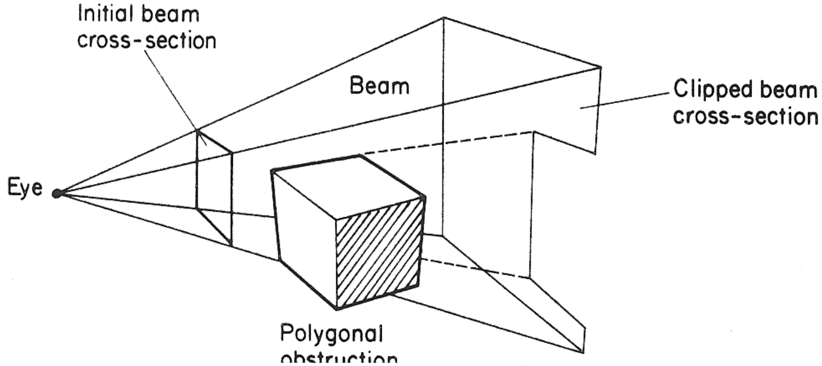


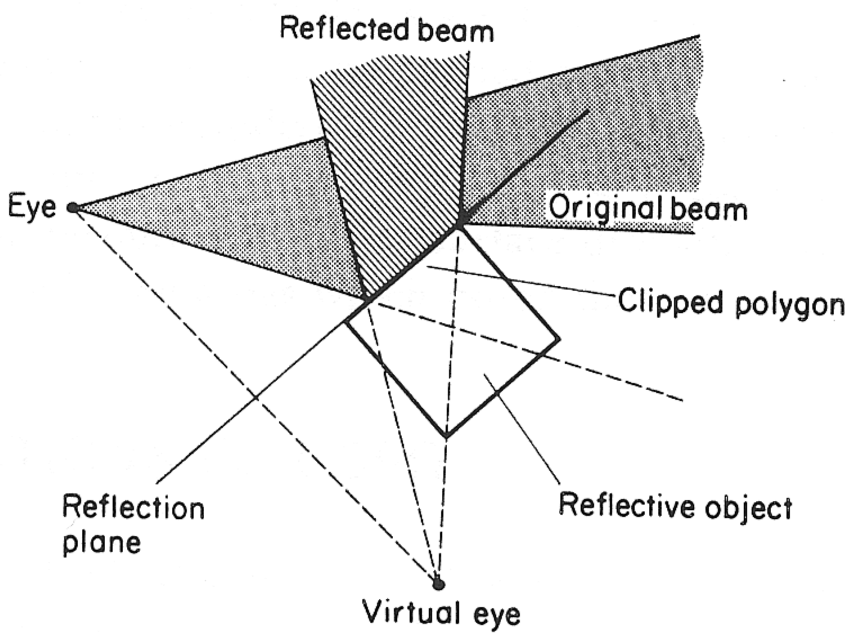
Beam und Cone Tracing

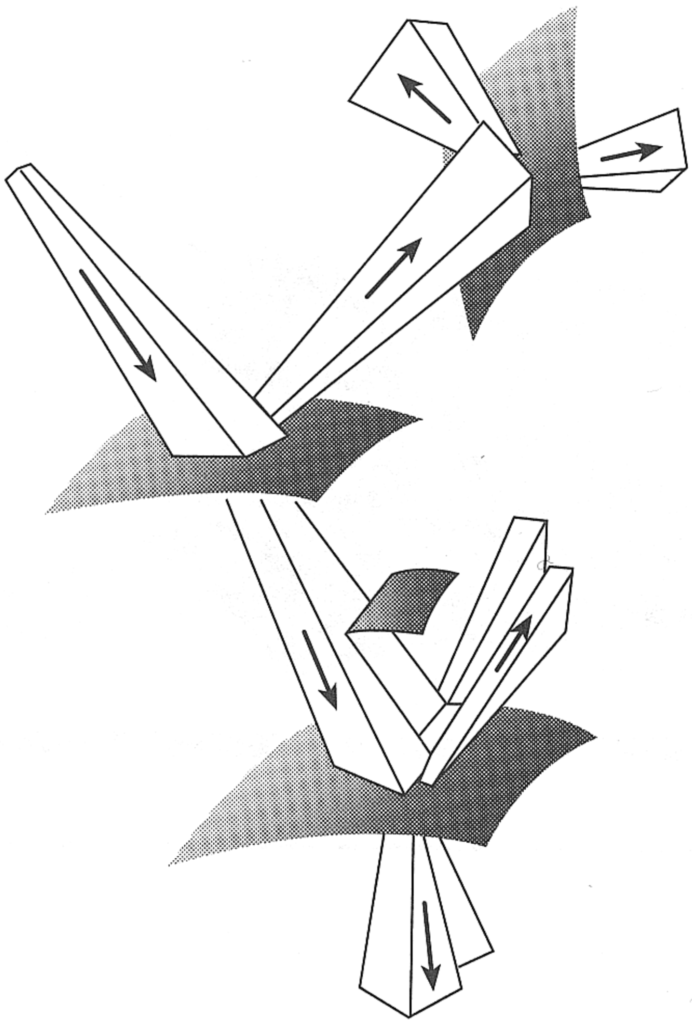
- **General idea:**
  - Trace continuous bundles of rays
- **Cone Tracing:**
  - Approximate collection of ray with cone(s)
  - Subdivide into smaller cones if necessary
- **Beam Tracing:**
  - Exactly represent a ray bundle with pyramid
  - Create new beams at intersections (polygons)
- **Problems:**
  - Clipping of beams?
  - Good approximations?
  - How to compute intersections?
- **Not really practical !!**



Beam Tracing







Packet Tracing

- **Approach**
  - Combine many similar ray (e.g. primary or shadow rays)
  - Trace them together in SIMD fashion
    - All rays perform the same traversal operations
    - All rays intersect the same geometry
  - Exposes coherence between rays
    - All rays touch similar spatial indices
    - Loaded data can be reused (in registers & cache)
    - More computation per recursion step → better optimization
  - Overhead
    - Rays will perform unnecessary operations
    - Overhead low for coherent and small set of rays (e.g. up to 4x4 rays)

Wrap Up

- **Acceleration Structures / Spatial Indices**
  - Necessary for sub-linear scalability (in scene size)
  - Hierarchies achieve O(log n)
  - Kd-trees offer
    - Simple building and traversal algorithms
    - Good performance for almost all scenes
  - BVH are also very popular
  - Dynamic changes to scenes
    - Require (partial) rebuilding of index
    - More research required
- **Handling Ray Bundles**
  - Cone- and beam tracing are not very practical
  - Packet tracing combines advantages with practical implementation