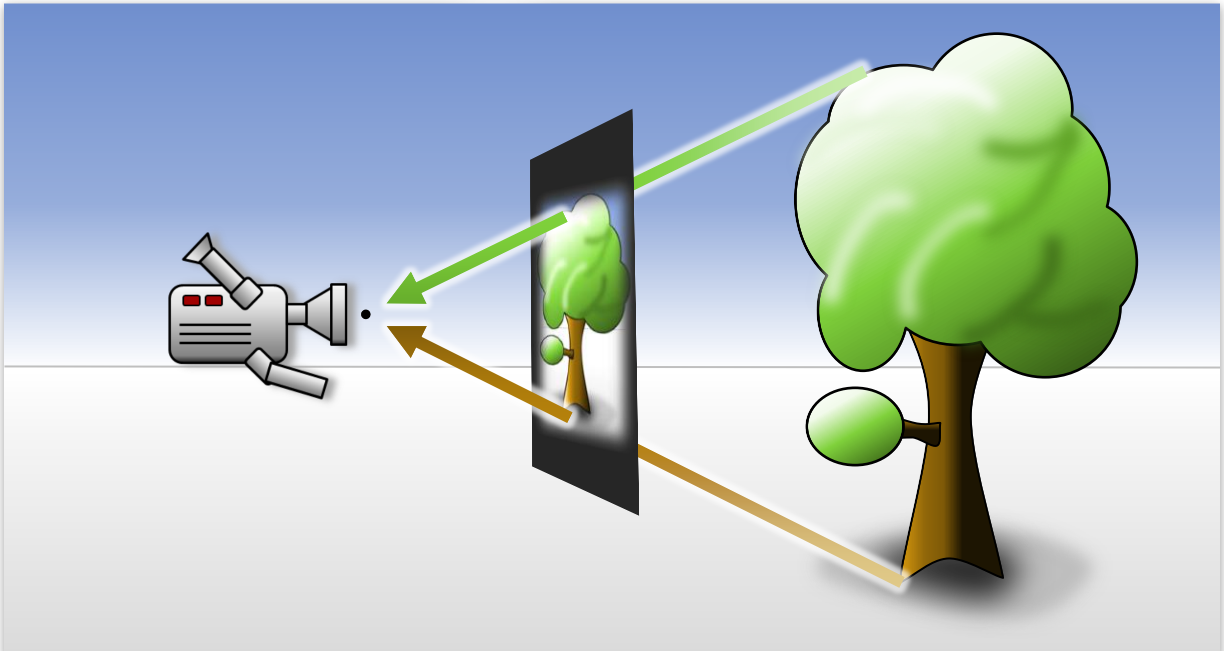
Prof. Dr. Tino Weinkauf

Introduction to Visualization and Computer Graphics

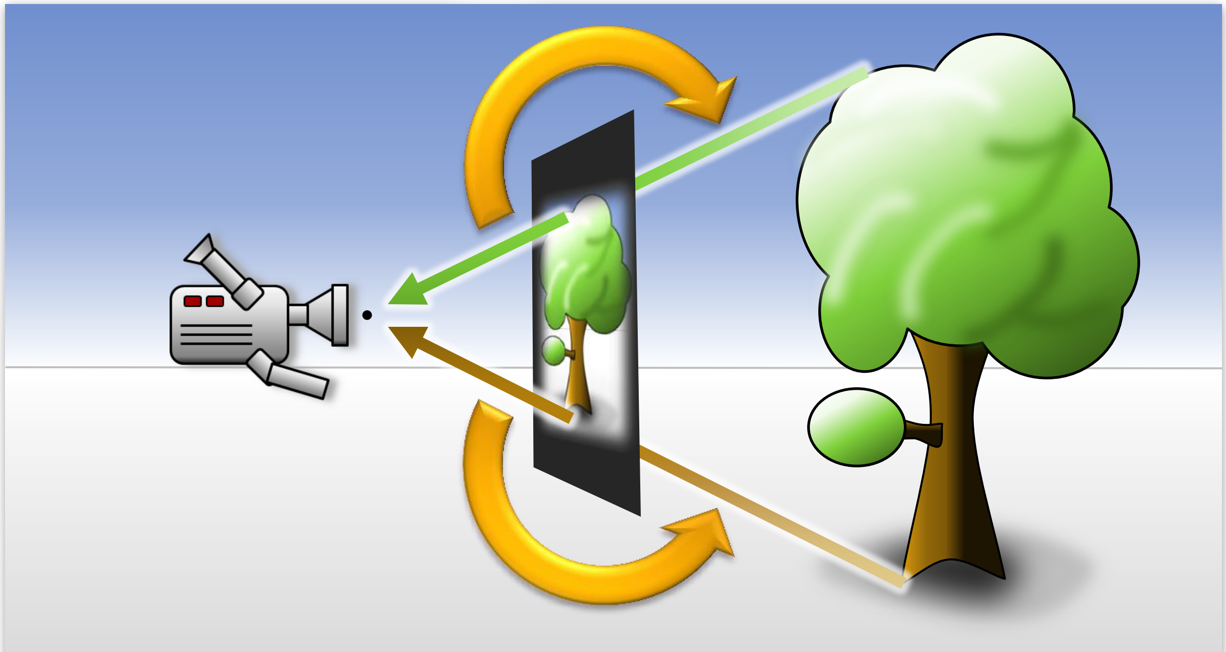
Raytracing

Basic Raytracing

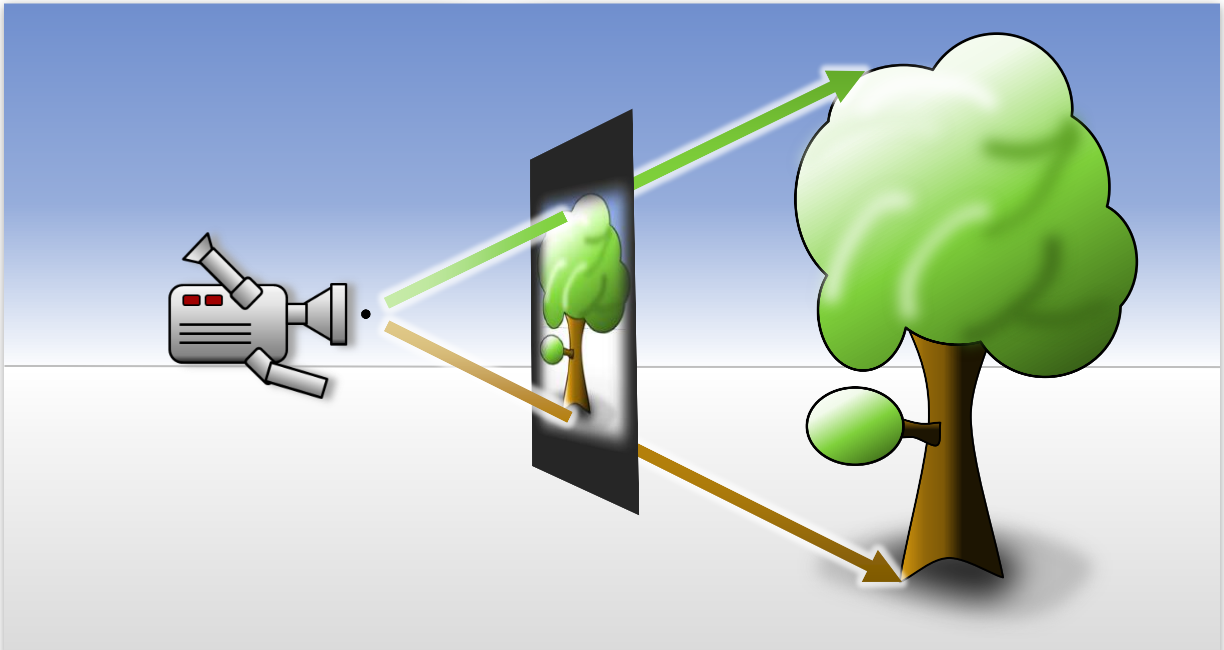
Central Projection



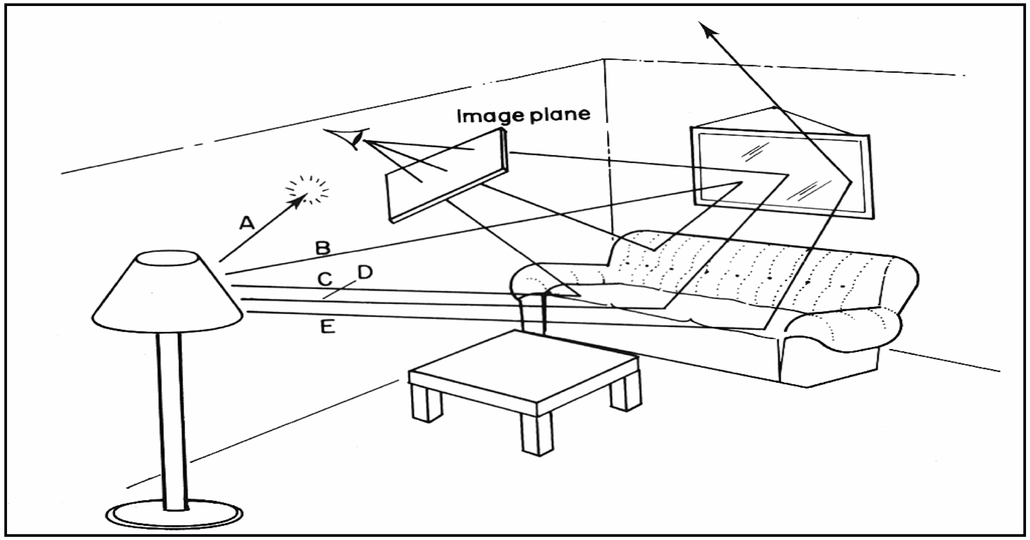
Central Projection



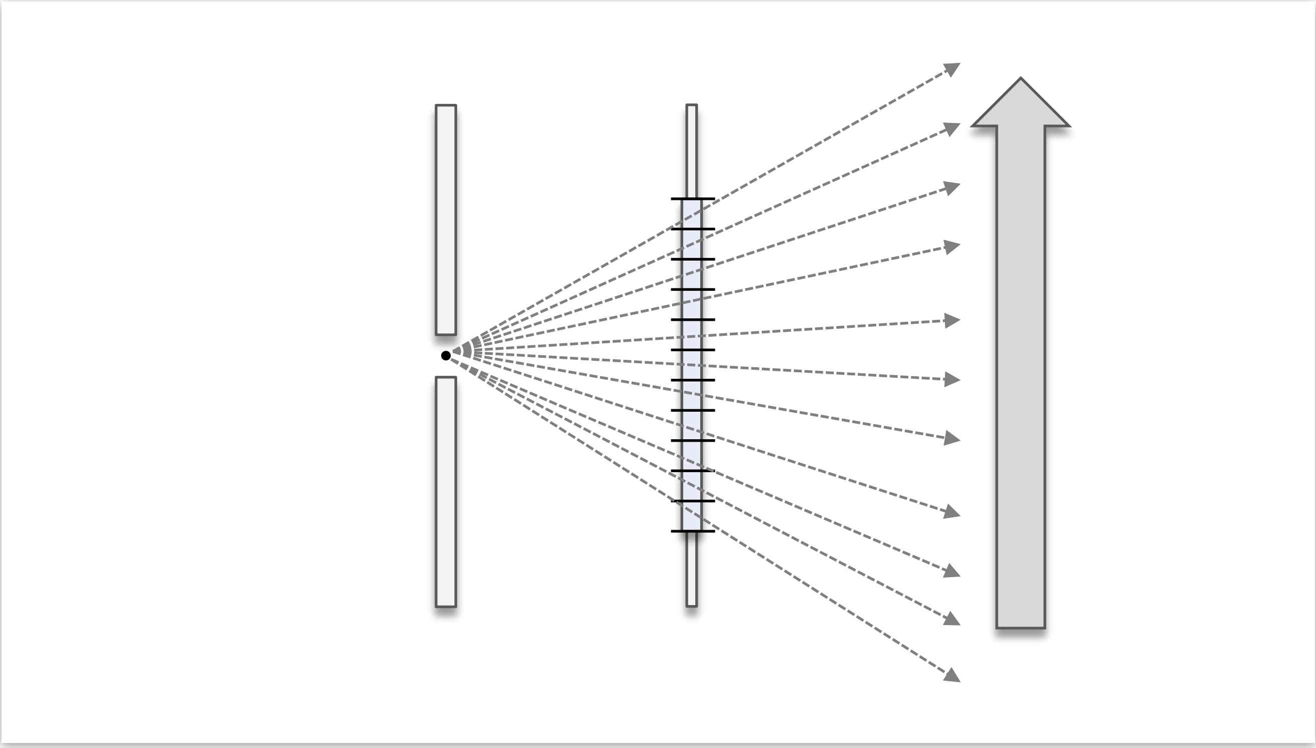
Ray Tracing



Some light rays (like A and E) never reach the image plane at all. Others follow simple (like C) or complicated routes. In general, it would be impractical to analyze all possible light paths because most of them do not have any practical influence on the generated image (do not intersect the image plane). In ray tracing, light rays are processed in inverse direction. It is assumed that all rays start at the eye of an observer (the viewing point), pass through the image plane toward objects in a scene, and finally reach the light sources. This approach reduces the number of traced light rays.



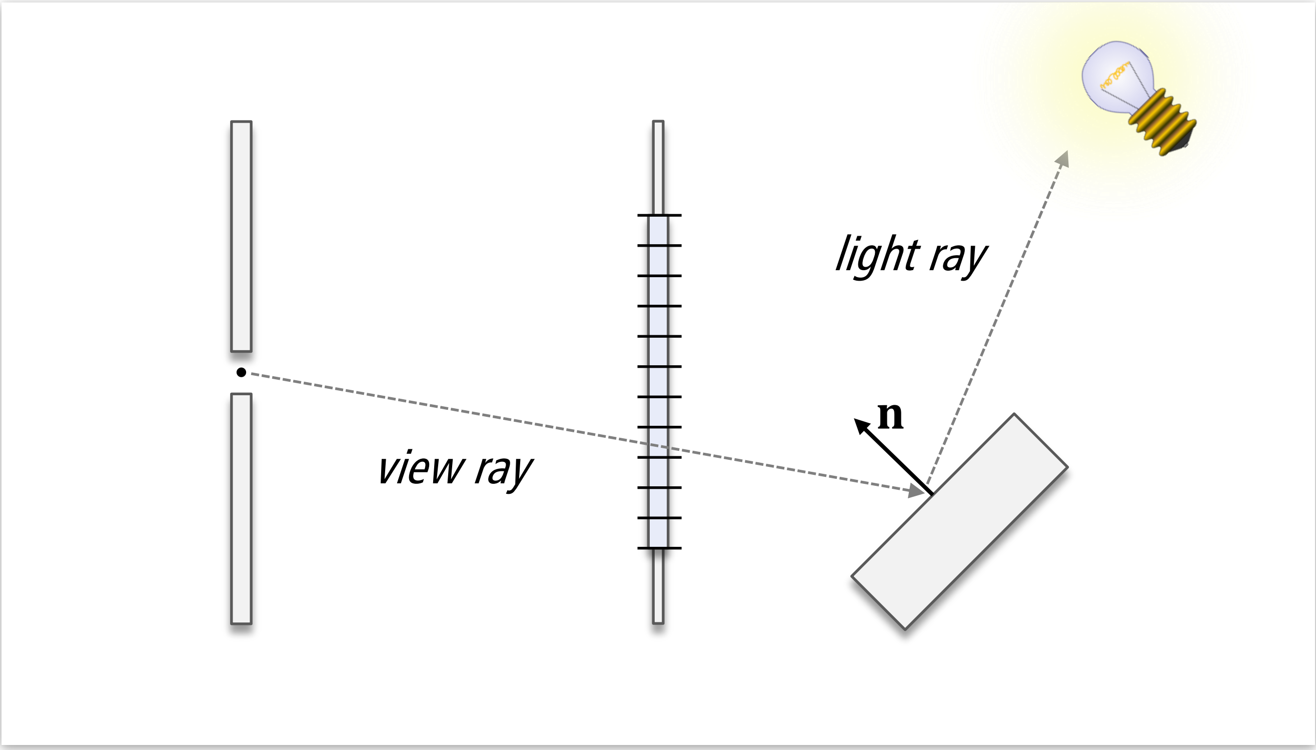
Ray Tracing



Primary Rays

- Rays through each pixel

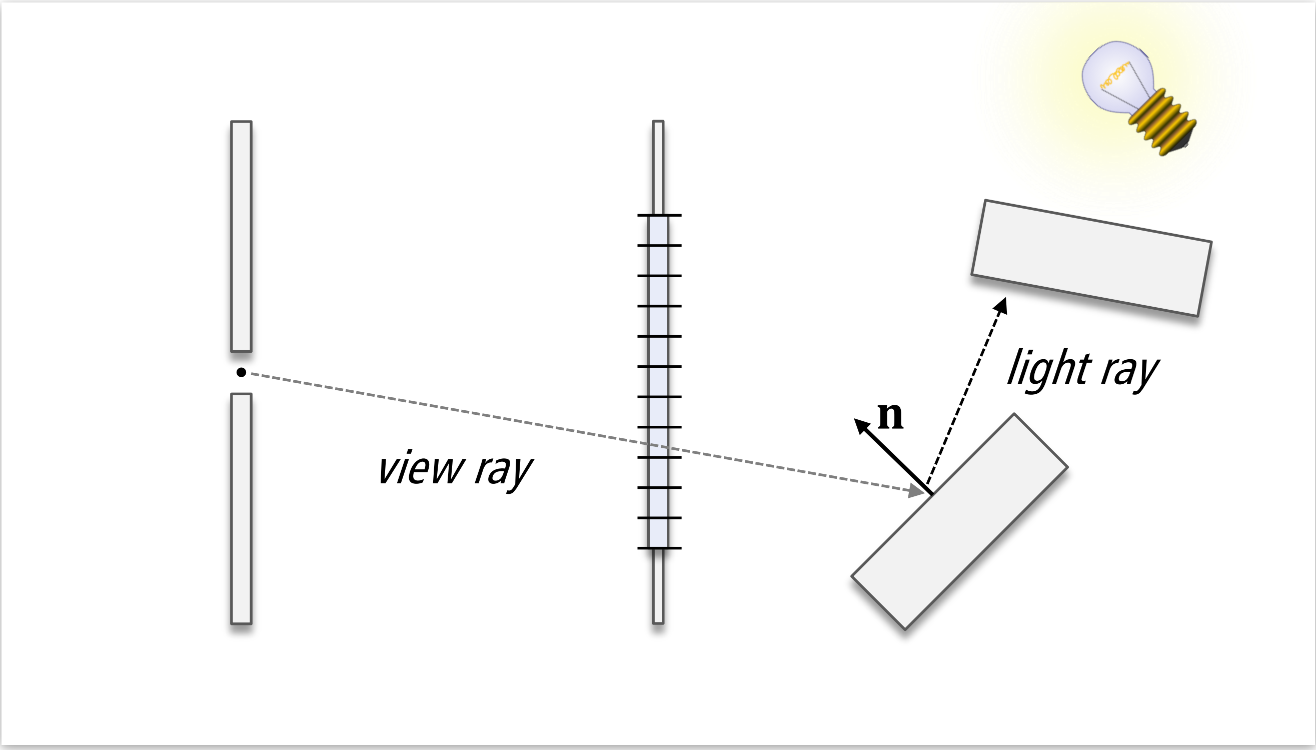
Local Illumination



Primary Rays

- Rays through each pixel
- (Basic trigonometry)

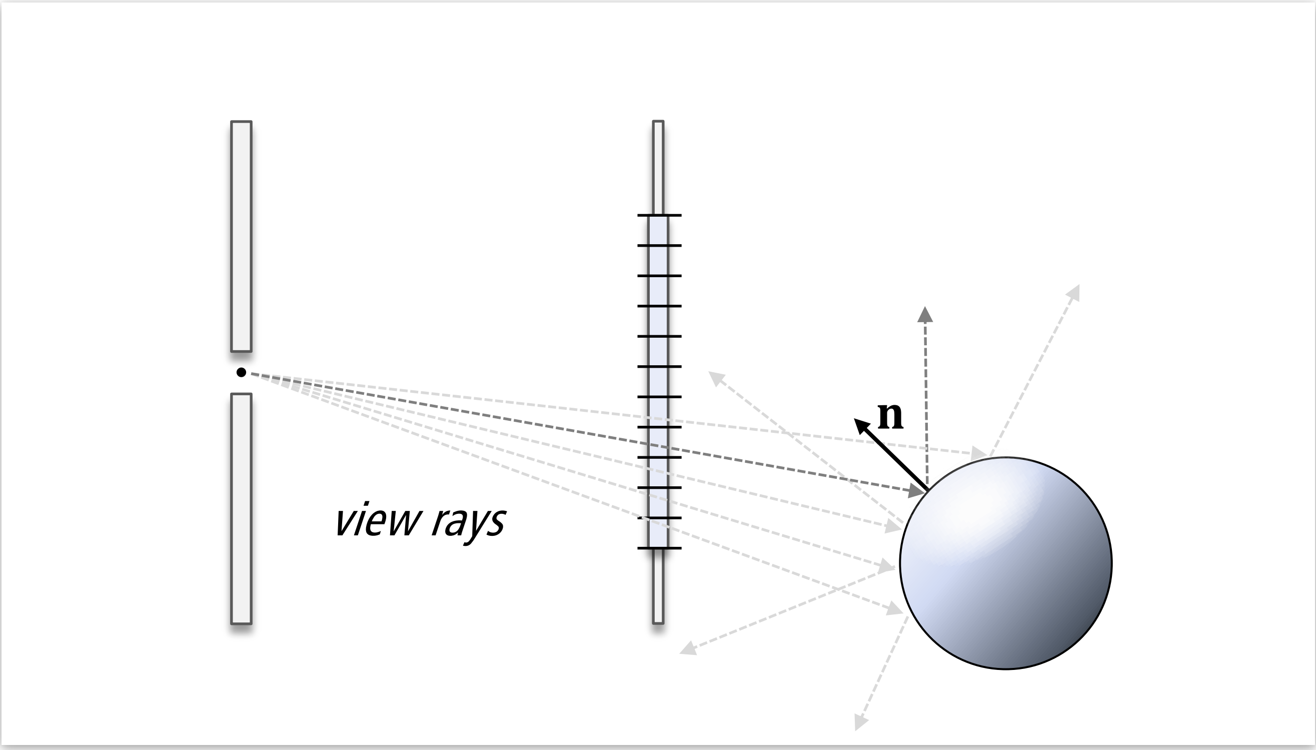
Shadows



Shadow rays

- Blocked by occluders (hard shadows)

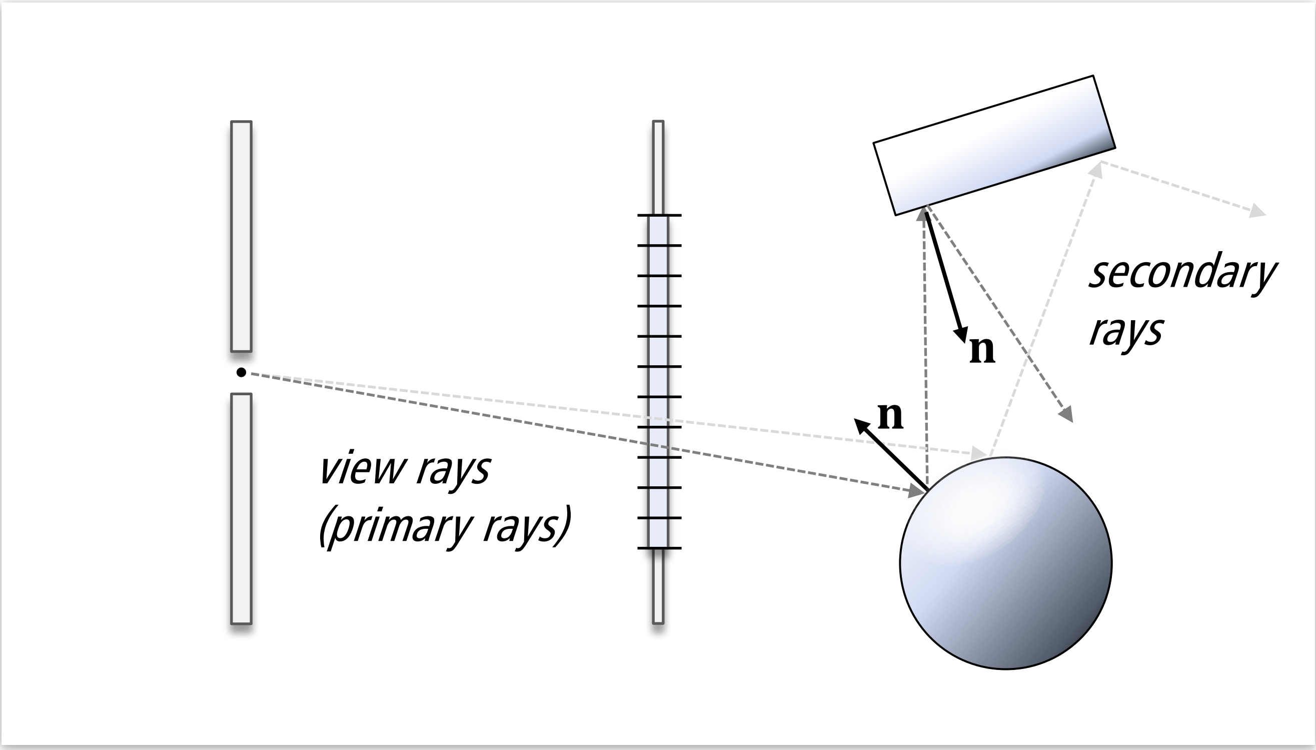
Reflection



Reflection

- Reflect ray across normal at intersection point
- (Basic linear algebra)

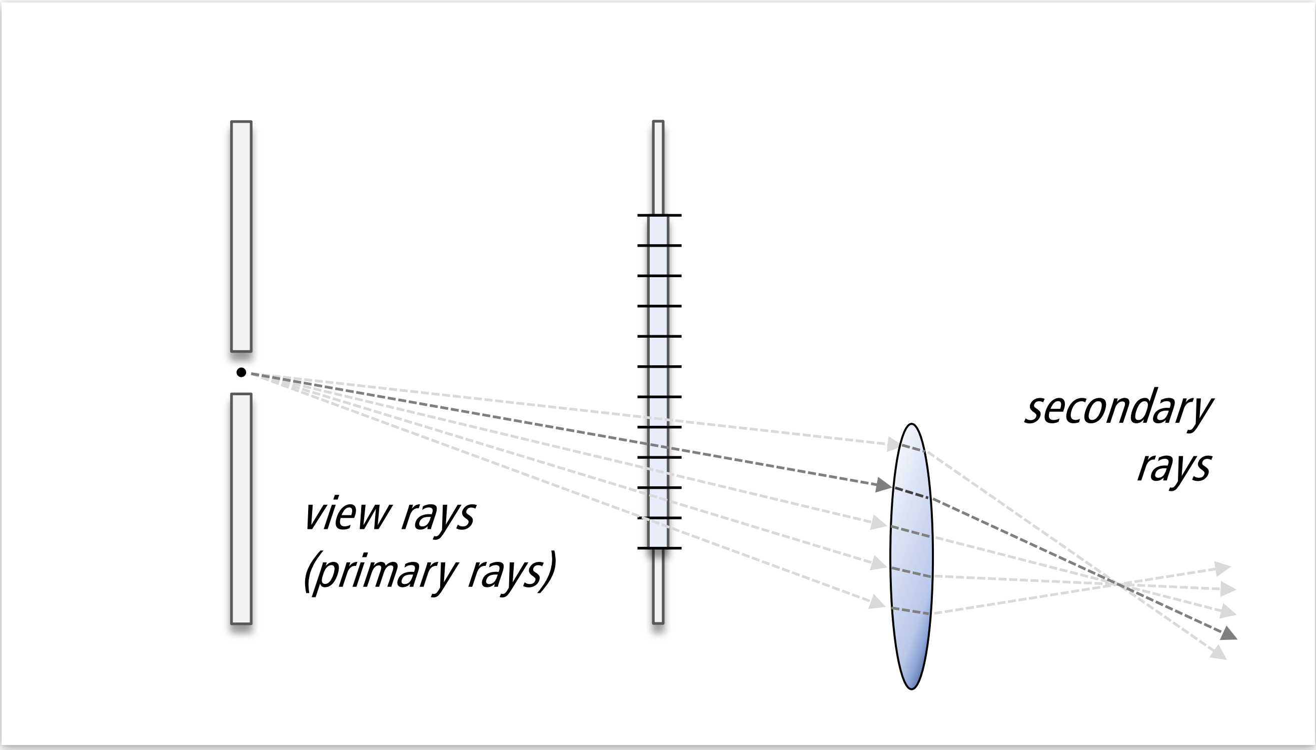
Multiple Reflections: Recursion



Multiple Reflections

- Call algorithm recursively for secondary rays
- (Terminate after *n* levels, for safety)

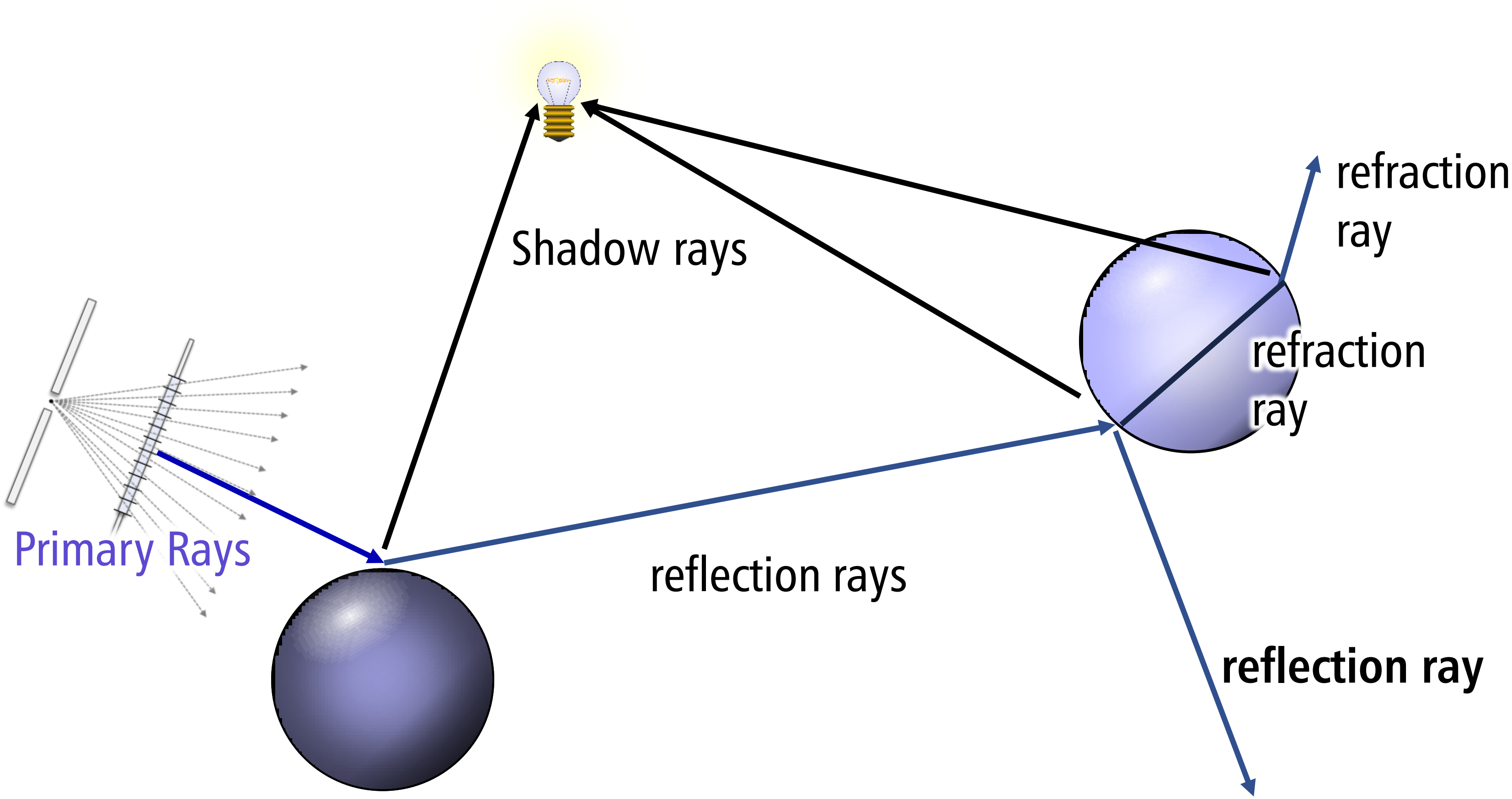
Refraction



Refraction

- Same story
- New rays: Snellius' law

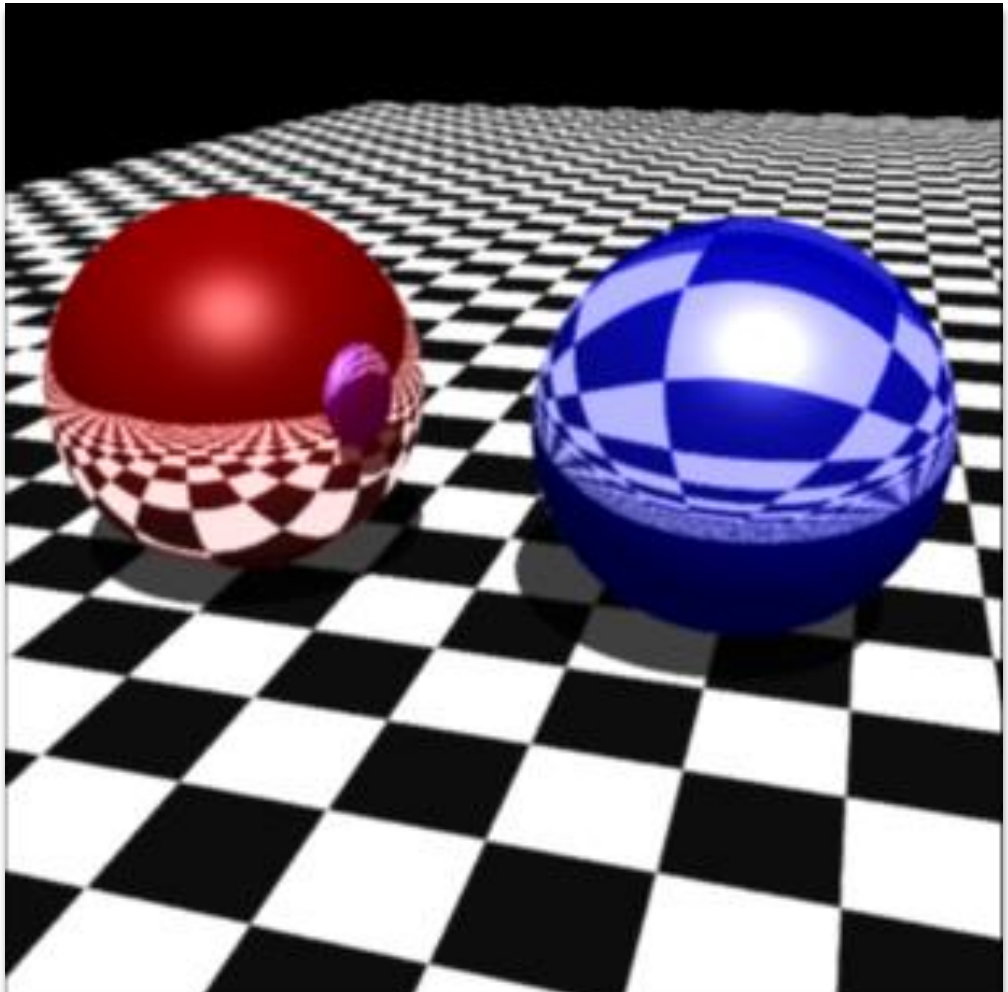
Recursive Raytracing



Worst-case complexity

- $\mathcal{O}(n \cdot m \cdot 2^{r})$
- n = Triangles, m = Pixels, r = maximum recursion depth

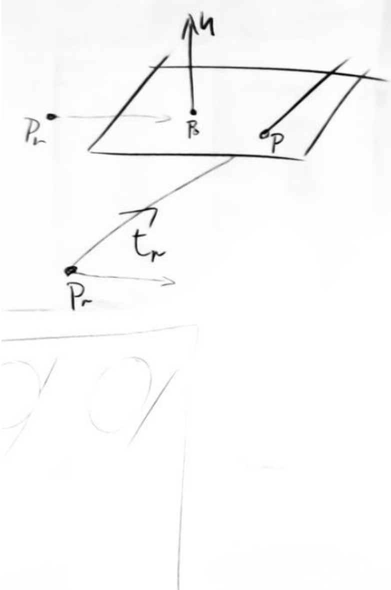
Raytracing in a Nutshell



Intersection Tests

- Ray-Plane
- Ray-Triangle

Ray-Plane Intersection



Plane:

$$(p - p_0)\cdot n = 0$$

Ray:

$$p = p_r + \lambda t_r \quad , \lambda \geq 0$$

$$\lambda < \lambda_{max} \Rightarrow \text{practical}$$

Ray-Plane Intersection

$$(p_r + \lambda t_r - p_0)\cdot n = 0$$

$$\lambda t_r \cdot n + (p_r - p_0)\cdot n = 0$$

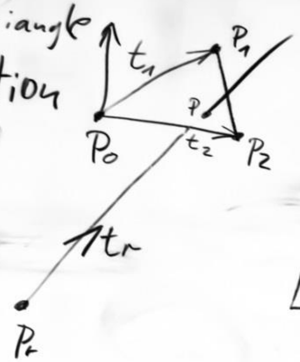
$$\lambda = \frac{(p_0 - p_r)\cdot n}{t_r \cdot n}$$

$$\boxed{t_r \cdot n \neq 0}$$

fabs( ) > ε -> practice

Ray-Triangle Intersection

Ray-Triangle Intersection



$$t_1 = P_1 - P_0$$

$$t_2 = P_2 - P_0$$

$$\boxed{n = t_1 \times t_2}$$

↓

Ray-Plane-Intersection

↓

$P \in$ Triangle?

Possible Plane Equation:

$$P = P_0 + \lambda_1 t_1 + \lambda_2 t_2$$

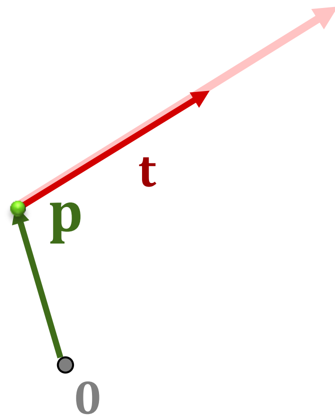
$\lambda_1, \lambda_2 \in \mathbb{R} \Rightarrow$ plane

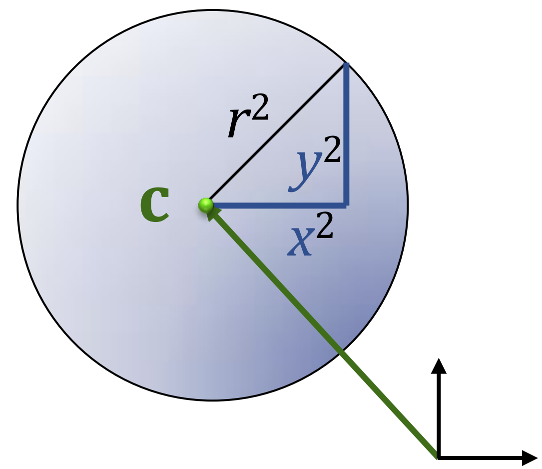
$\lambda_1, \lambda_2 \geq 0$

$\lambda_1, \lambda_2 \leq 1$ within triangle

$\lambda_1 + \lambda_2 \leq 1$

Ray-Sphere Intersection





Parametric line equation:

$$\mathbf{x}(\lambda) = \mathbf{p}_r + \lambda \mathbf{t}_r$$
$$\lambda \geq 0$$

Sphere (Implicit!)

$$\langle \mathbf{x} - \mathbf{c}, \mathbf{x} - \mathbf{c} \rangle = r^2$$

$$\langle \mathbf{x}(\lambda) - \mathbf{c}, \mathbf{x}(\lambda) - \mathbf{c} \rangle - r^2 = 0$$
$$\langle \mathbf{p}_r + \lambda \mathbf{t}_r - \mathbf{c}, \mathbf{p}_r + \lambda \mathbf{t}_r - \mathbf{c} \rangle - r^2 = 0$$

Derivation

Solving the equation:

$$\langle \mathbf{x}(\lambda) - \mathbf{c}, \mathbf{x}(\lambda) - \mathbf{c} \rangle - r^2 = 0$$

$$(\mathbf{x}(\lambda) - \mathbf{c})^2 - r^2 = 0$$

$$(\mathbf{p}_r + \lambda \mathbf{t}_r - \mathbf{c})^2 - r^2 = 0$$

$$\left(\lambda \mathbf{t}_r + (\mathbf{p}_r - \mathbf{c})\right)^2 - r^2 = 0$$

Result: 1D Quadratic equation in λ

$$\lambda^2 \mathbf{t}_r^2 + \lambda 2(\mathbf{t}_r \cdot (\mathbf{p}_r - \mathbf{c})) + (\mathbf{p}_r - \mathbf{c})^2 - r^2 = 0$$

Ray-Sphere Intersection (unit sphere)

For the unit sphere: center at origin, radius=1

$$\lambda^2 \mathbf{t}_r^2 + \lambda 2(\mathbf{t}_r \cdot \mathbf{p}_r) + \mathbf{p}_r^2 - 1 = 0$$

Spatial Data Structures

Range Queries

Spatial Data Structures

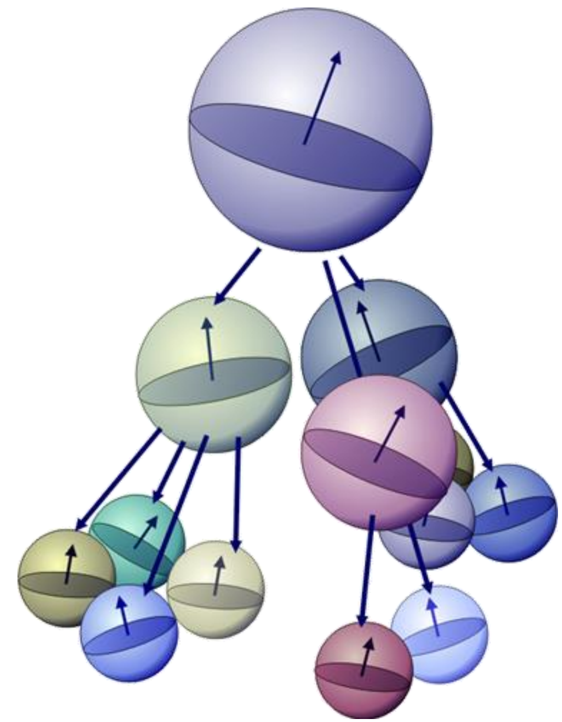
Range Queries

- Common problems
  - Raytracing
  - Select object by mouse click
  - Collision detection
- This should work on large models
  - Scale to billions of primitives
  - Asymptotic complexity

Spatial Data Structures

Basic Idea: Hierarchical decomposition

- If number objects too large:
  - Form spatially coherent groups
  - For each group:
    - Simple bounding volume
    - Apply recursively



Result

- We obtain a tree of bounding volumes
- "Bounding volume hierarchy"

Bounding Volumes

Axis-Aligned Bounding Box

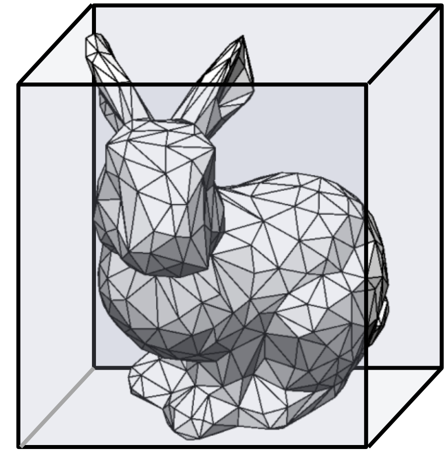
- Store minimum x,y,z-coord

  and

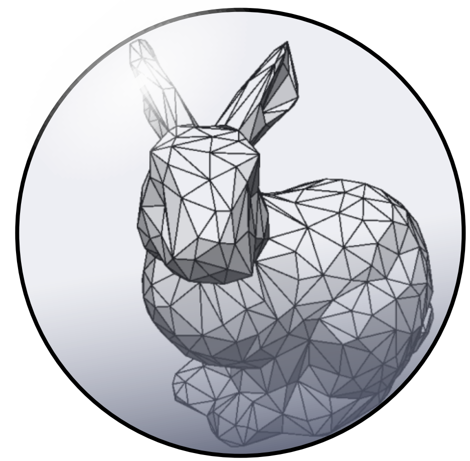
- maximum x,y,z-coord

Bounding Sphere

- Store radius, center
- Such that all geometry is contained



axis-aligned bounding box

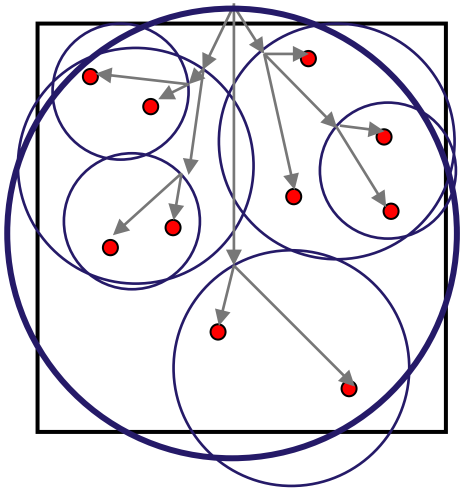


bounding sphere

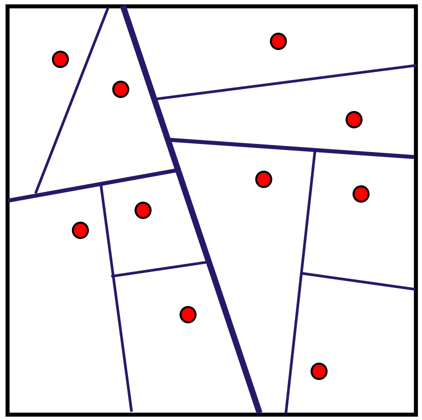
Variants

Variants:

- Bounding volume hierarchy
  - General definition
  - Any bounding volumes
  - Image: spheres



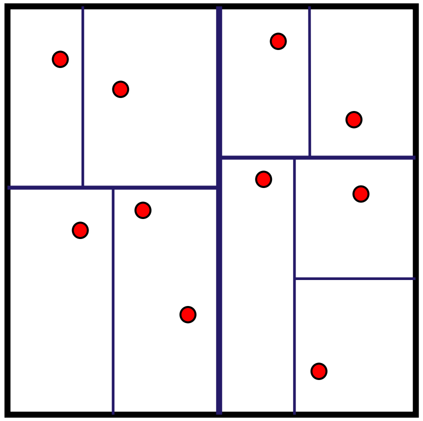
- BSP-tree
  - Split planes (half-spaces)
  - “Binary space partition tree”
  - Arbitrary planes



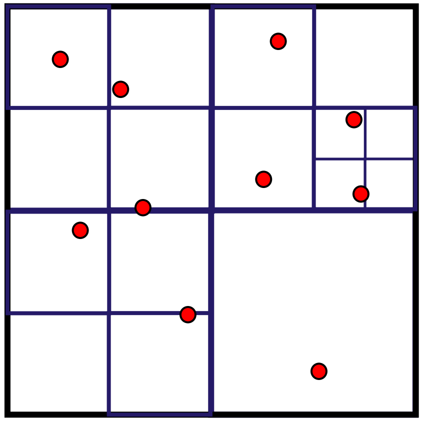
Variants

Variants

- Axis aligned BSP tree / kD-tree
  - Axis-parallel splitting planes
  - Special case: kD-tree
    - Alternating splitting dimensions
    - Median cut:
      split at median coordinate



- Quadtrees / Octrees
  - Divide into 4/8 cubes
  - Special case of the above
    (no binary tree though)



Extended Objects

Extended objects (other than points)

- Extended objects:
  - Triangles
  - Polygons
  - etc...
- Division of space might intersect with object
- Three solutions
  - Split objects (expensive, uncommon)
  - Overlapping nodes (common)
  - Storage multiple times (also common)

Splitting Objects

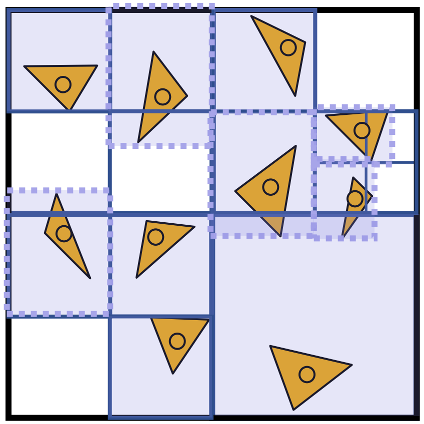
First solution: *splitting*

- **Example:** Triangles in BSP tree
  - Split at plane
  - Aim at few splits
- (Rather) easy to see:
  - General BSP tree needs still $\mathcal{O}(n^2)$ fragments (worst case, n triangles; practice: $\approx \mathcal{O}(n \log n)$)
  - Lower bound for kD trees, octrees, etc...
- Splitting usually too expensive
  - Used in early low-polygon 3D engines (BSP-visibility)

Overlapping Regions

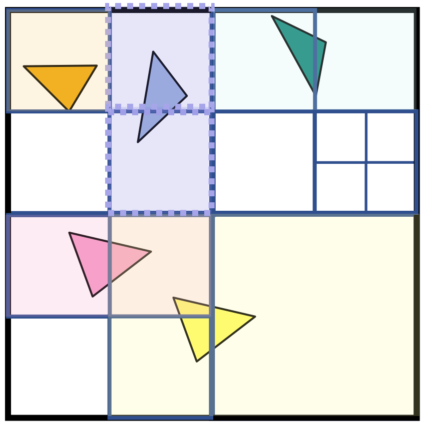
Second Solution: *overlap*

- Permit overlapping bounding volumes
- E.g., second bounding box (octree)
- Possible strategy:
  - Up to 10% oversize (in each direction)
  - No fit into leaf nodes: use an inner node
- Overlap reduces efficiency
  - Multi-coverage of volume
  - 10% in each direction means $1.2^3 \approx 1.7\times$
  - Effect on algorithms might vary



Overlapping Regions

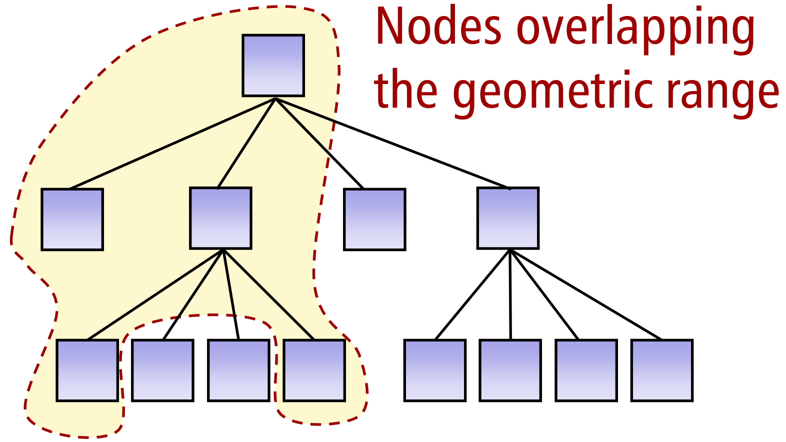
Third Solution: *store multiple times*



- Store primitive multiple times
- Disadvantages
  - Reduced efficiency
  - Additional memory
- Advantages
  - Regular structures
  - No additional bounding boxes
- Common for raytracing

Range Queries

Range Query Algorithm

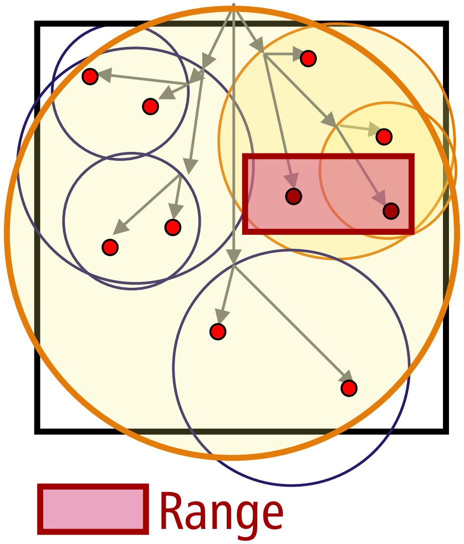


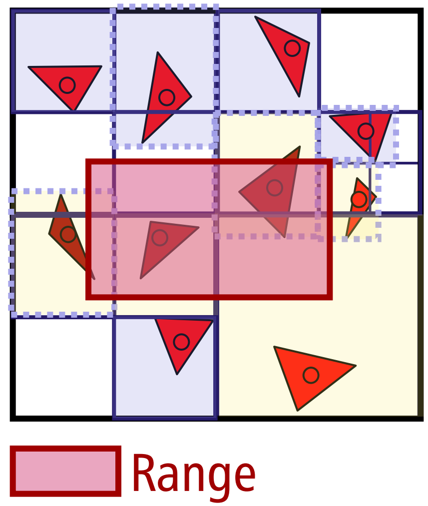
Start at root node: Then, recursively

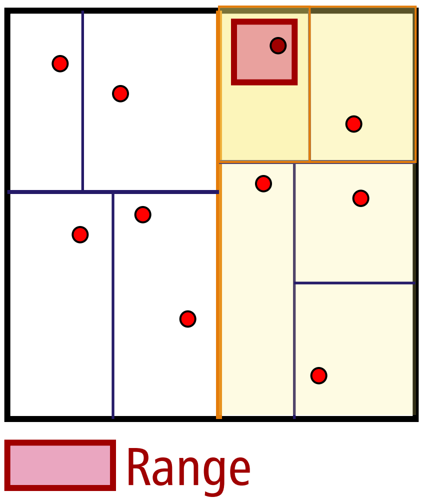
- If *range* overlaps *bounding box*
  - Test node primitives
    - Report if within range
  - Call recursively for child nodes
- If *range* does not overlap *bounding box*
  - End recursion

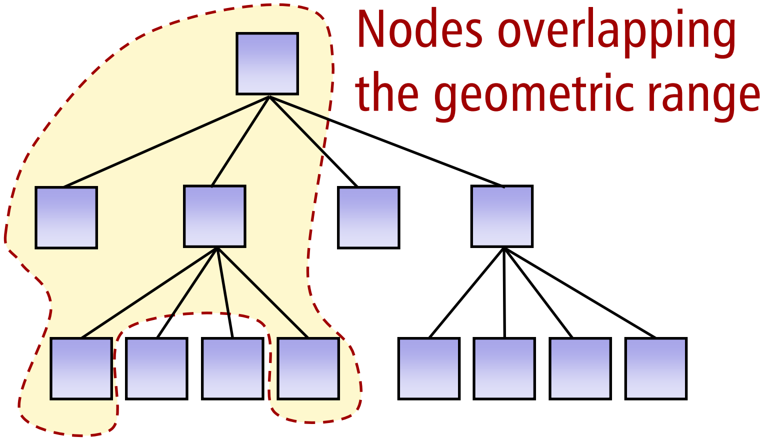
algorithm works for all hierarchy types

Examples

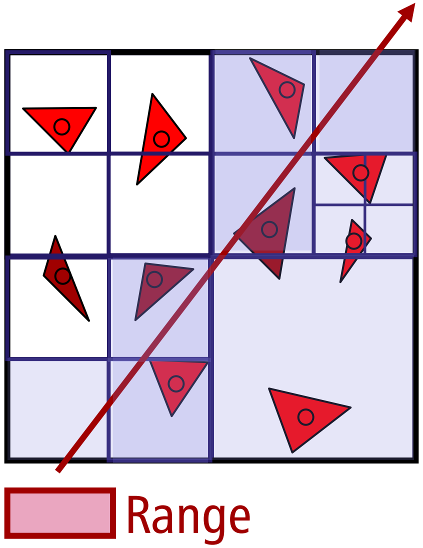








Raytracing



Raytracing: special case

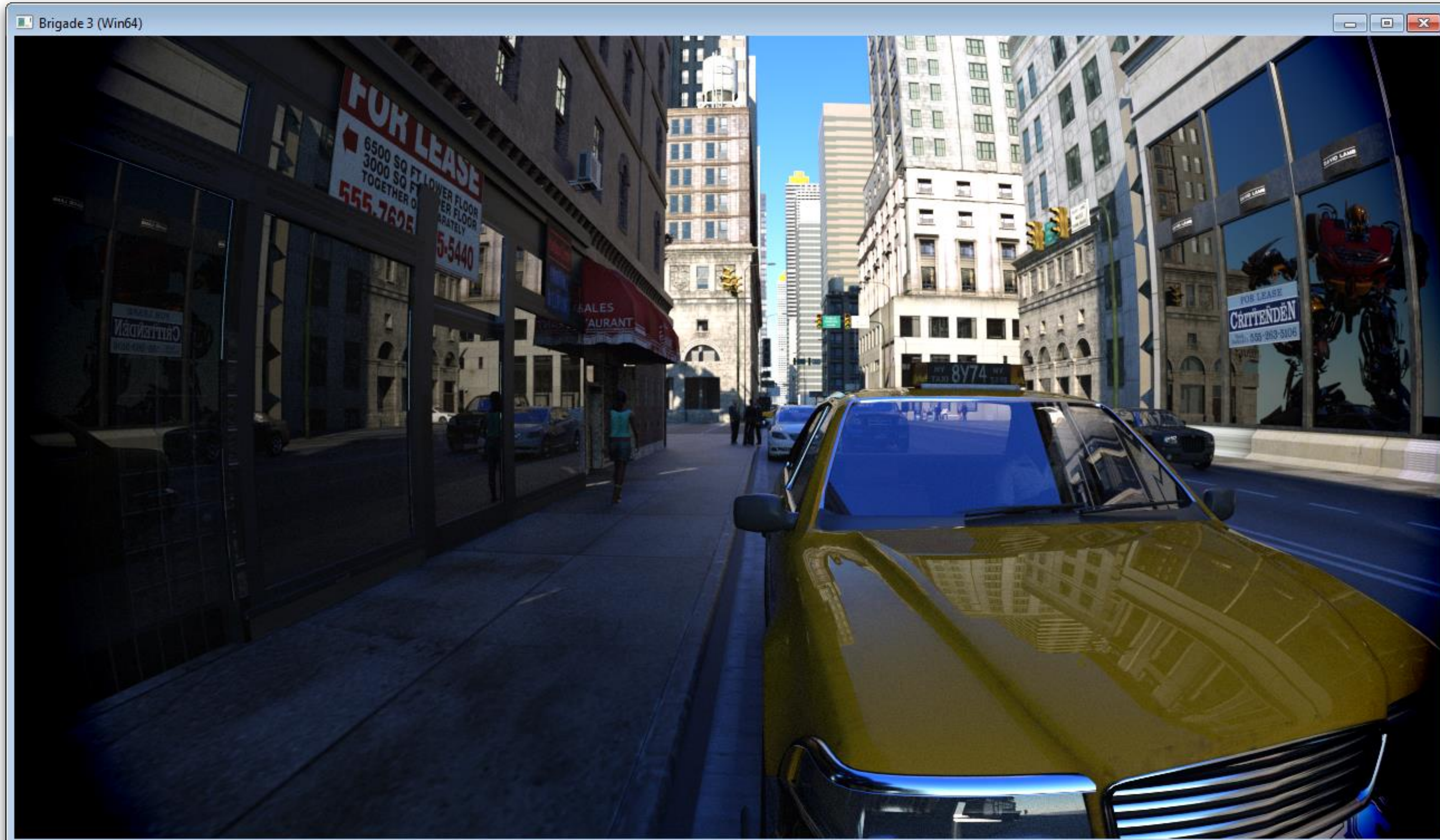
- Ray is the range
- Early ray termination
  - Sorted recursion (child closer to the camera: first)
  - Stop after hit

In Practice

Significant Speedup

- Simple implementation
  - Axis-aligned BSP tree
  - Single-core C++
  - 1.000.000 triangle scene
  - ~500.000 triangle-ray intersections per second
- If you work harder...
  - Optimized software ~15M
  - GPU implementations up to 100M
  - Optimized versions:
    Performance also depends on ray coherence

Brigade Renderer



0.5-1 sec/frame **on 2 × GeFoce GTX Titan** – http://raytracey.blogspot.nl/

Brigade Renderer

realtime (youtube compression artifacts!) – Samuel Lapere / Youtube