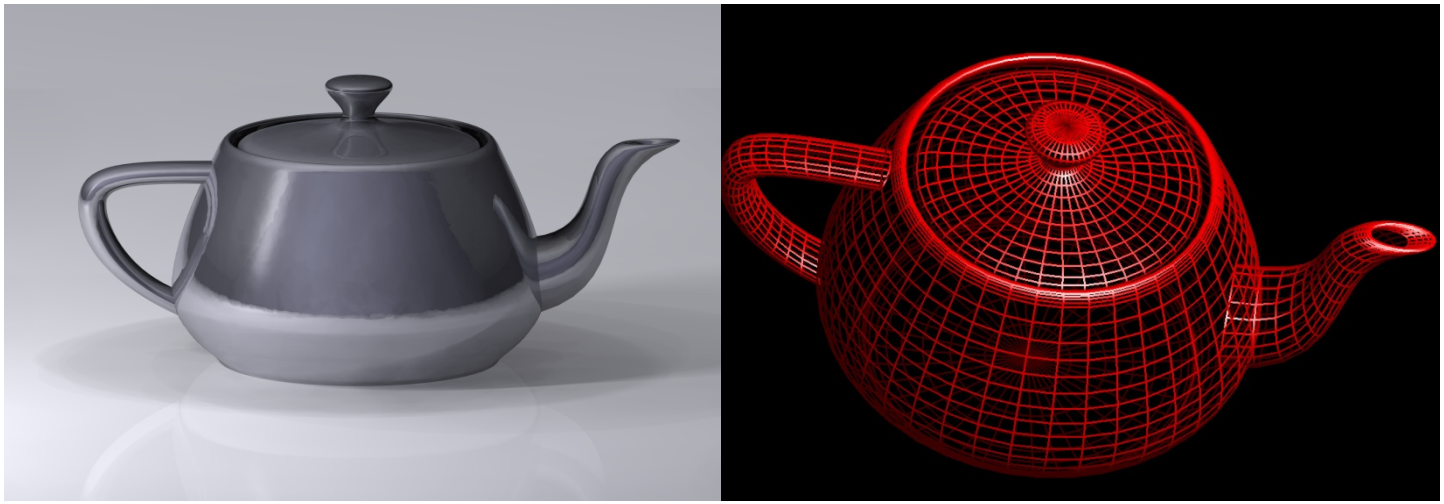


Foundations of Computer Graphics

Saurabh Ray

Today's plan.

More spatial data structures.

Distribution Ray Tracing.

Spatial Data Structures

Goal: to quickly figure out which objects a ray intersects.

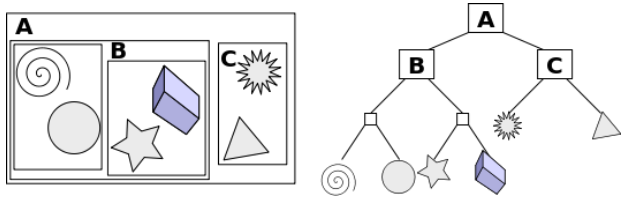
Object Partitioning Schemes

- *divide objects into disjoint groups*
- *these groups may overlap in space*

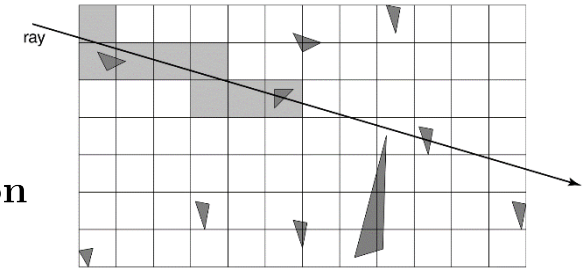
Space Partitioning Schemes

- *space is partitioned*
- *an object may intersect multiple regions*

Hierarchical Bounding Boxes



Uniform Spatial Subdivision

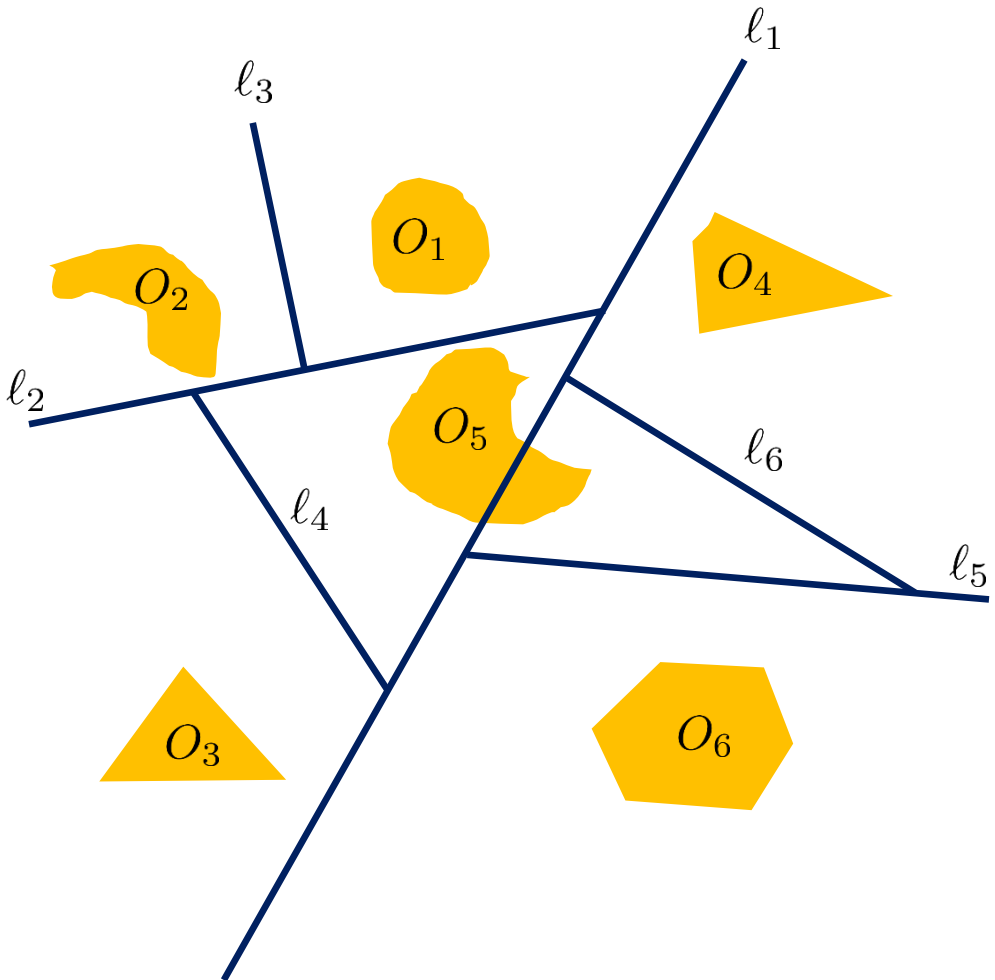


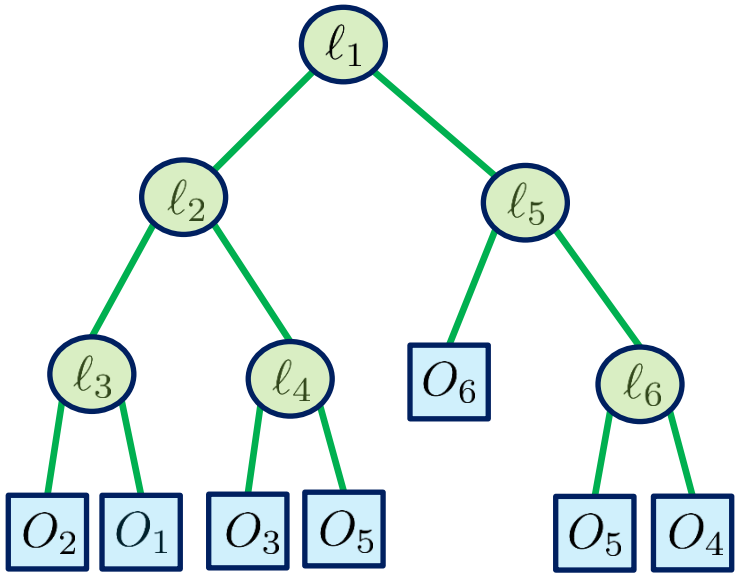
Spatial data structures can also be used for collision detection.

Binary Space Partition

Input: n disjoint objects. **Goal:** build a data structure for fast ray shooting.

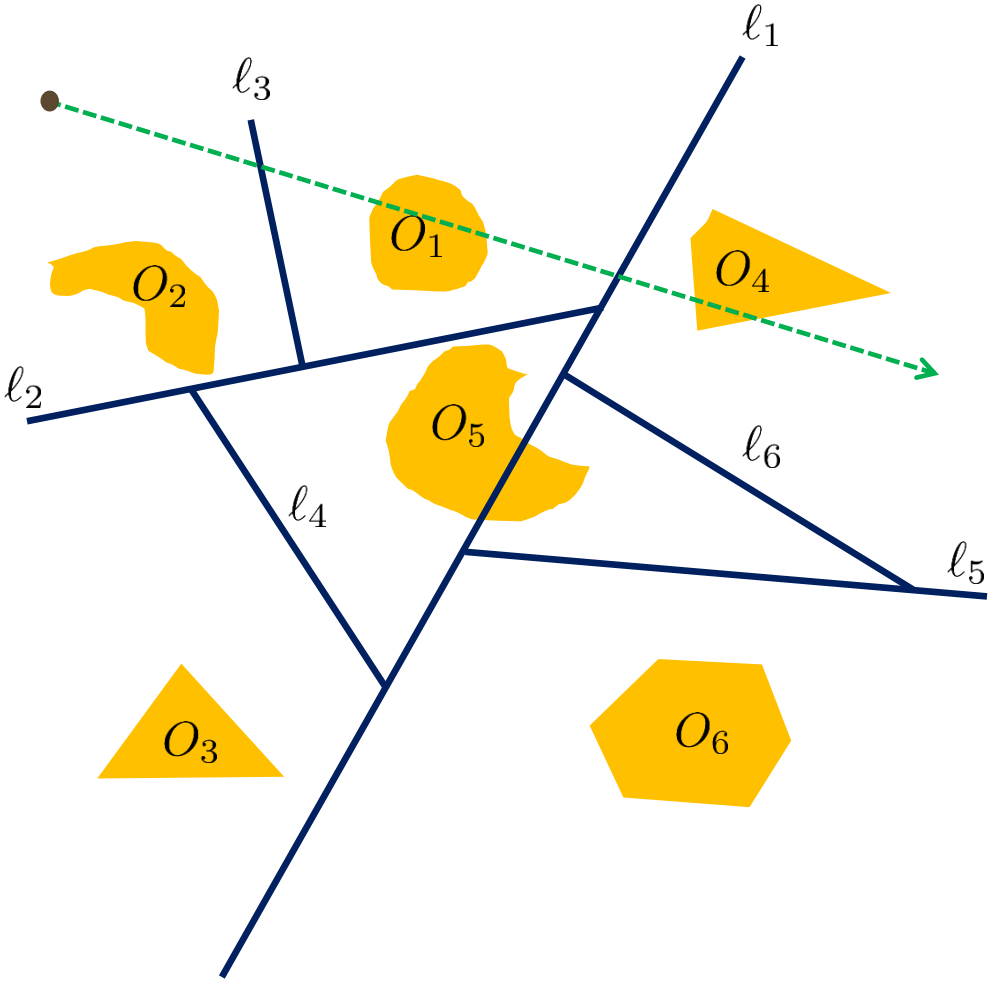
Idea: *recursively subdivide the space using a plane (a line in 2D)*
some objects may be cut by the splitting planes (or lines)

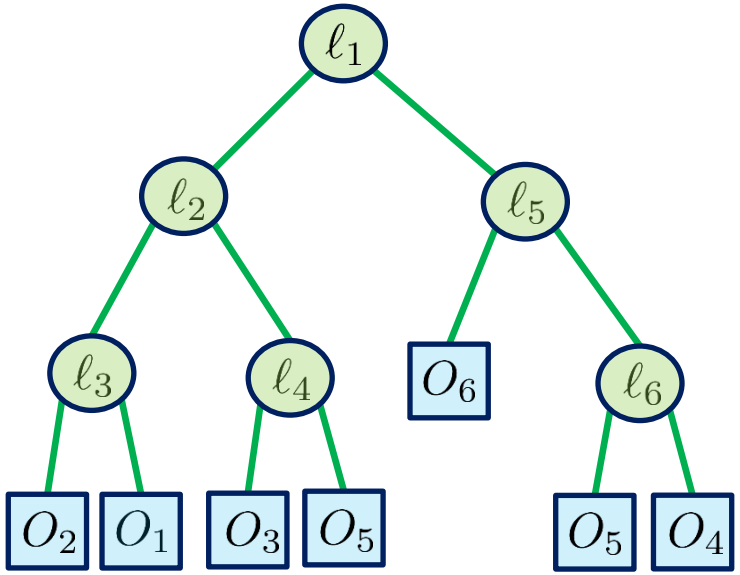




Note: O_5 *appears in two places.*

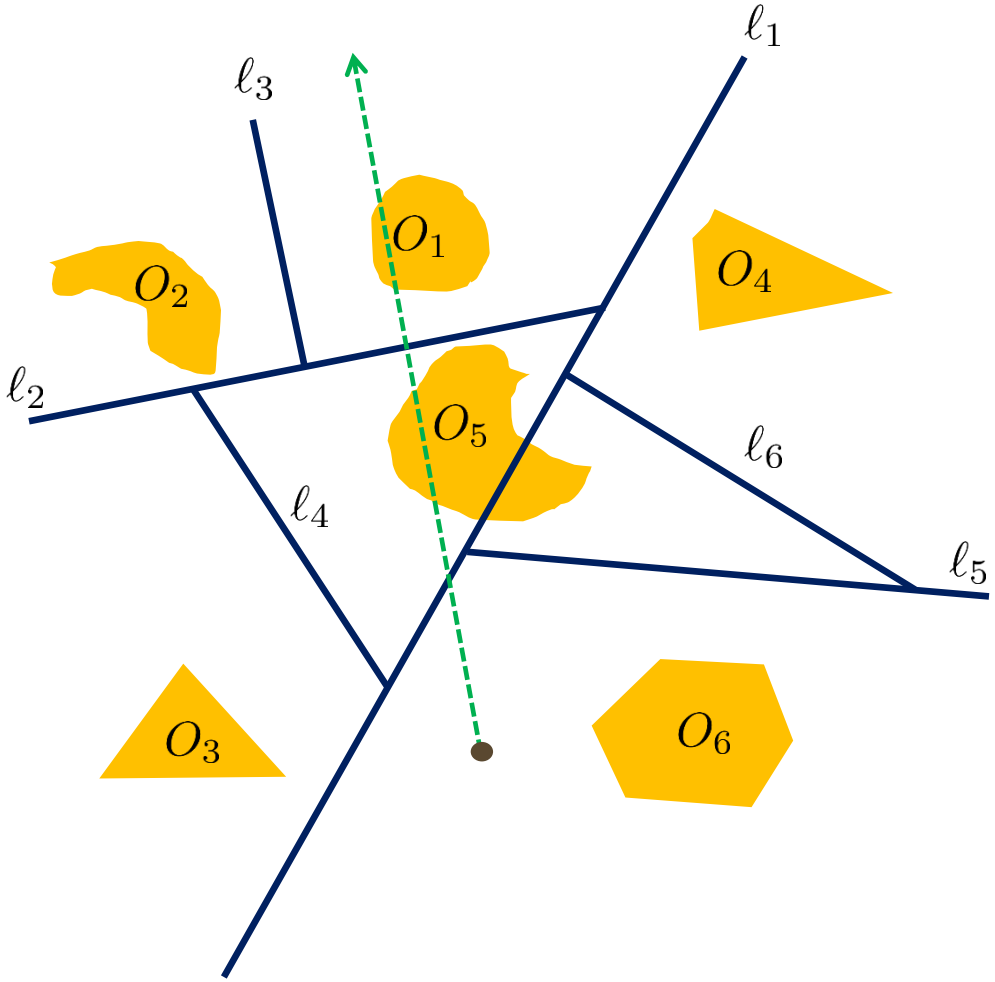
Spatial Data Structures

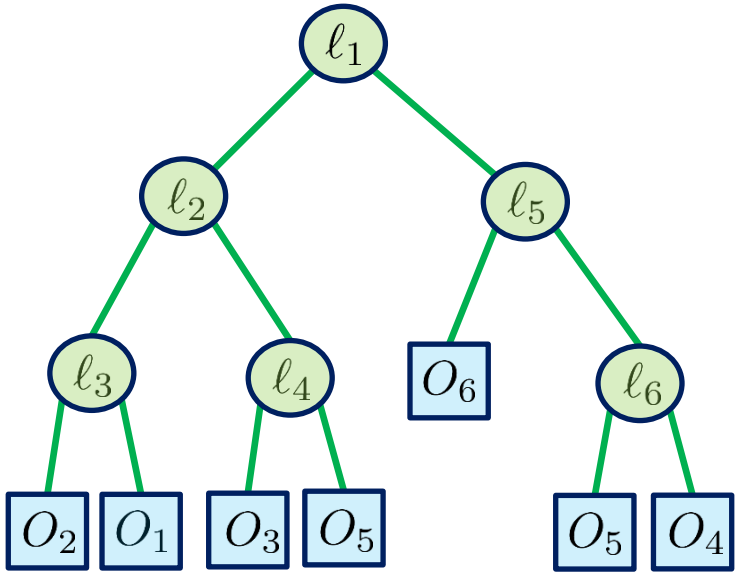




Path followed in the tree: ℓ_1 ℓ_2 ℓ_3 O_2 ℓ_3 O_1

Spatial Data Structures



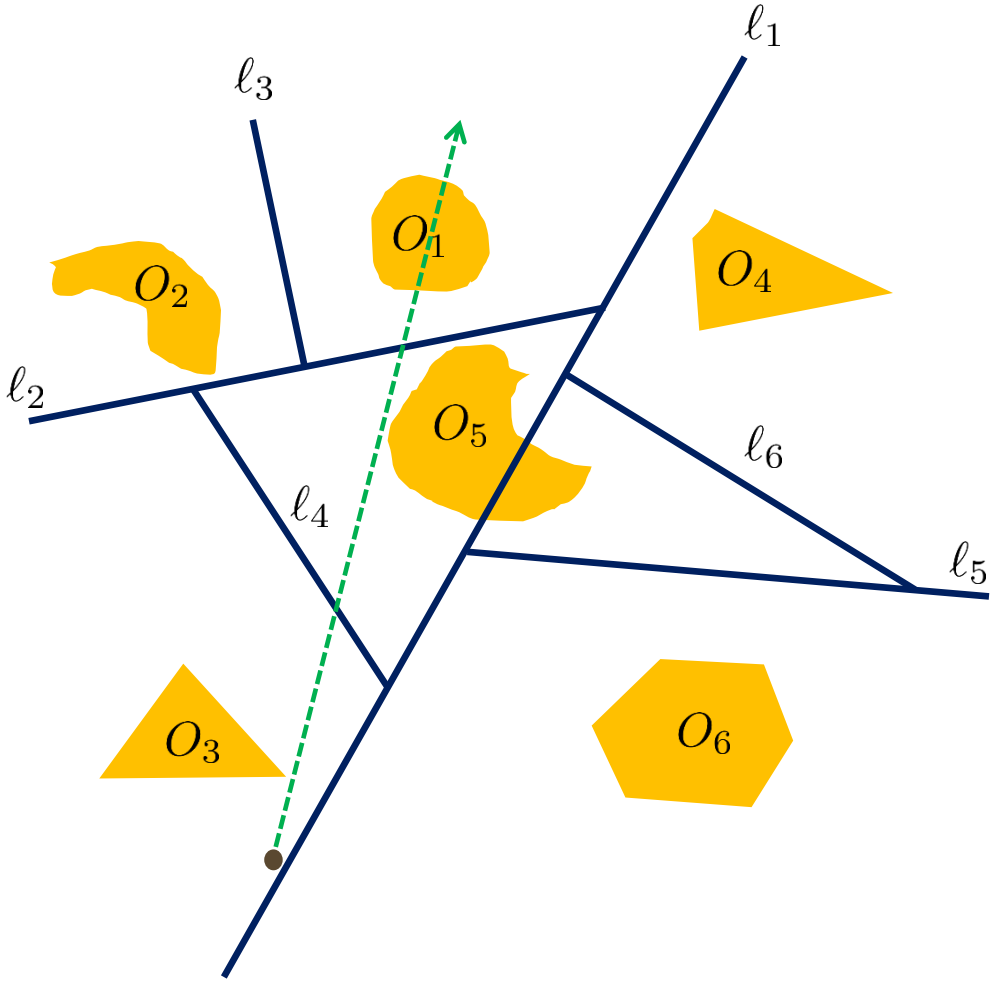


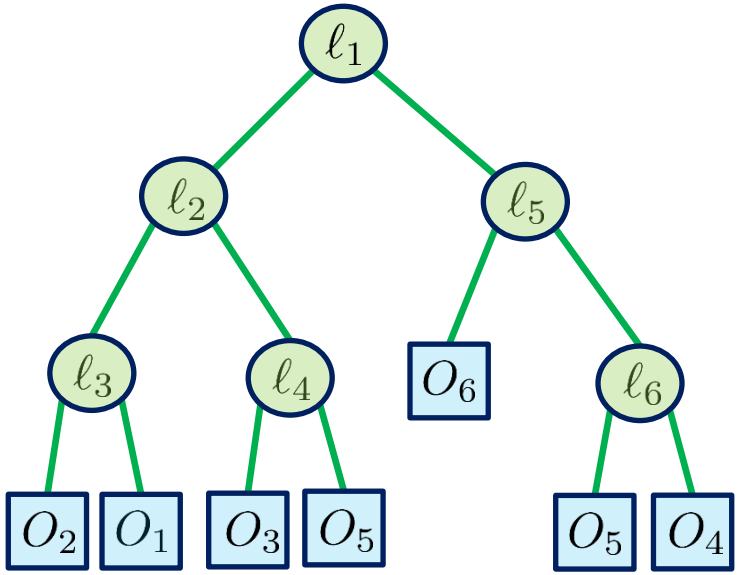
Path followed in the tree:

ℓ_1 ℓ_5 O_6 ℓ_5 ℓ_6 O_5 O_4 ℓ_6 ℓ_5 ℓ_1 ℓ_2 ℓ_4 O_5

no intersection with the part of O_5 in the left subtree of ℓ_6!

Spatial Data Structures





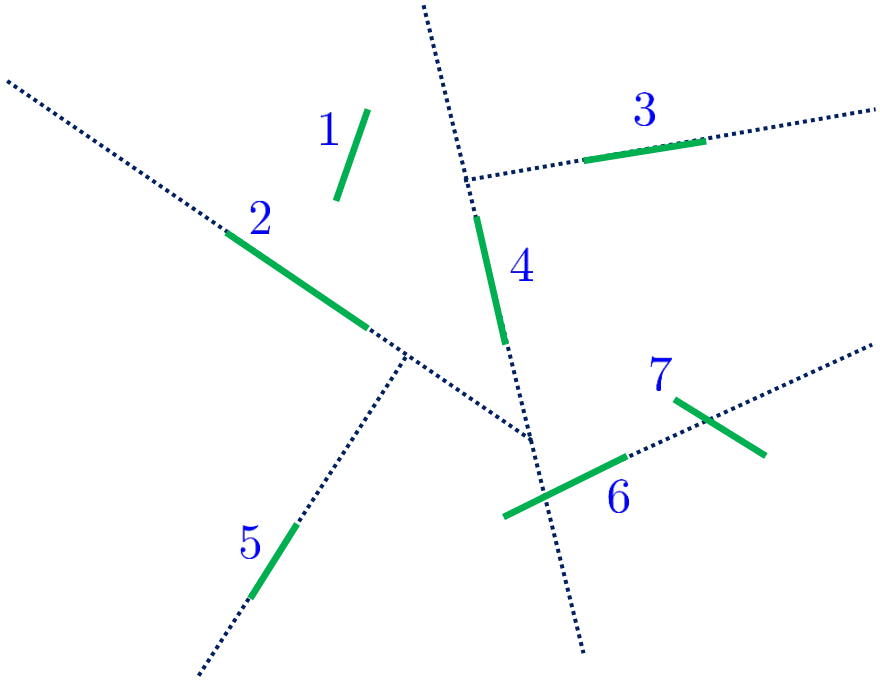
Path followed in the tree: ℓ_1 ℓ_2 ℓ_4 O_3 l_4 O_5 ℓ_4 ℓ_2 ℓ_3 O_2 ℓ_3 O_1

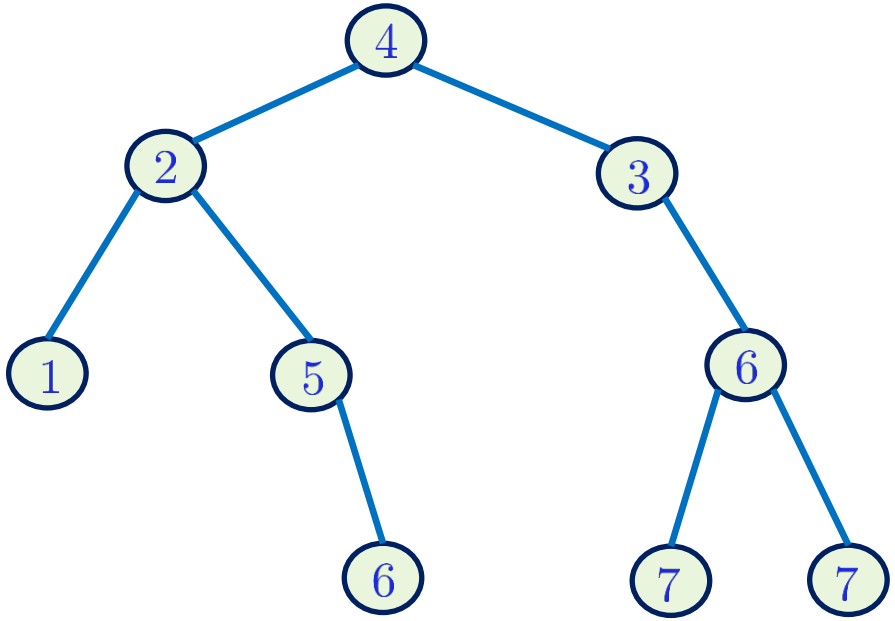
Spatial Data Structures

In the previous example, all objects correspond to leaves in the tree.

If the objects were a set of line segments in the plane, we could have used lines containing one of the segments as a splitting line.

In that case some of the objects correspond to internal nodes.





Ray shooting is done in exactly the same way as before, except that when move from one side of a node to another, we check intersection with the object at the node too.

Spatial Data Structures

For any line (plane) we define the positive side and the negative side.

In 2D, a line ℓ has an equation of the form $ax + by = 0$.

We define ℓ^+ as $\{(x, y) : ax + by > 0\}$, and ℓ^- as $\{(x, y) : ax + by < 0\}$.

In 3D, a plane h has equation $ax + by + cz = 0$.

We define h^+ as $\{(x, y, z) : ax + by + cz > 0\}$, and h^- as $\{(x, y, z) : ax + by + cz < 0\}$.

Algorithm 2DBSP(S)
Input. A set $S = \{s_1, s_2, \ldots, s_n\}$ of segments.
Output. A BSP tree for S.

1. **if** card(S) $\leqslant 1$
2. **then** Create a tree $\mathcal{T}$ consisting of a single leaf node, where the set S is stored explicitly.
3. **return** $\mathcal{T}$
4. **else** (∗ Use $\ell(s_1)$ as the splitting line. ∗)
5. $S^+ \leftarrow \{s \cap \ell(s_1)^+ : s \in S\}$; $\mathcal{T}^+ \leftarrow$2DBSP(S^+)
6. $S^- \leftarrow \{s \cap \ell(s_1)^- : s \in S\}$; $\mathcal{T}^- \leftarrow$2DBSP(S^-)
7. Create a BSP tree $\mathcal{T}$ with root node v, left subtree $\mathcal{T}^-$, right subtree $\mathcal{T}^+$, and with $S(v) = \{s \in S : s \subset \ell(s_1)\}$.
8. **return** $\mathcal{T}$

This algorithm always blindly chooses the line containing the first segment as the splitting line.

Is this a good idea? No! This line may be cutting too many segments.

Alternative: pick a segment that cuts the fewest number of segments.

Is this a good idea? No! All other segments may be on the same side of the line.

What else can you do? 

Algorithm 2DRANDOMBSP(S)
1. Generate a random permutation $S' = s_1, \ldots, s_n$ of the set S.
2. $\mathcal{T} \leftarrow$ 2DBSP(S')
3. **return** $\mathcal{T}$

Theorem: The expected number of fragments generated by this algorithm is $O(n \log n)$.

Theorem: A BSP tree of size $O(n \log n)$ can be compute in expected time $O(n \log^2 n)$.

Spatial Data Structures

In three dimensions, instead of segments we have triangles.

Algorithm 3DBSP(S)
Input. A set $S = \{t_1, t_2, \ldots, t_n\}$ of triangles in $\mathbb{R}^3$.
Output. A BSP tree for S.

1. **if** card$(S) \leqslant 1$
2. **then** Create a tree $\mathcal{T}$ consisting of a single leaf node, where the set S is stored explicitly.
3. **return** $\mathcal{T}$
4. **else** ($*$ Use $h(t_1)$ as the splitting plane. $*$)
5. $S^+ \leftarrow \{t \cap h(t_1)^+ : t \in S\}$; $\mathcal{T}^+ \leftarrow$3DBSP(S^+)
6. $S^- \leftarrow \{t \cap h(t_1)^- : t \in S\}$; $\mathcal{T}^- \leftarrow$3DBSP(S^-)
7. Create a BSP tree $\mathcal{T}$ with root node v, left subtree $\mathcal{T}^-$, right subtree $\mathcal{T}^+$, and with $S(v) = \{t \in S : t \subset h(t_1)\}$.
8. **return** $\mathcal{T}$

We can again use a random permutation of the triangles.

We don't know how to analyze the randomized algorithm.

Fortunately, a slightly modified variant can be analyzed!

Theorem: Given a set of disjoint triangles in $\mathbb{R}^3$, a BSP tree of size $O(n^2)$ exists. This bound cannot be improved!

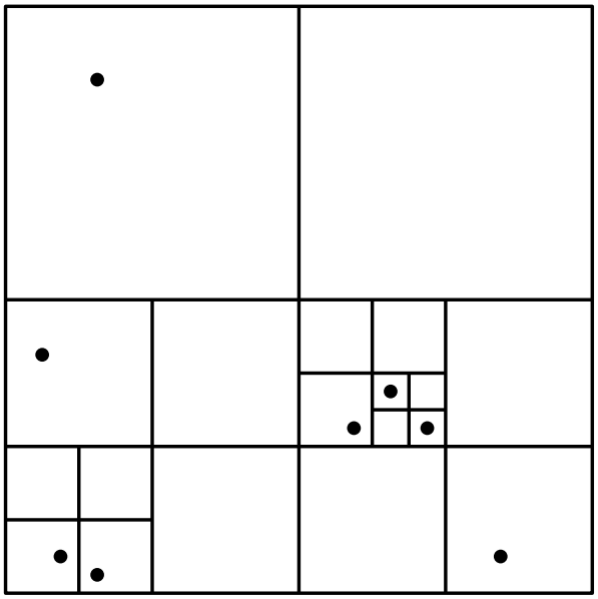
Spatial Data Structures

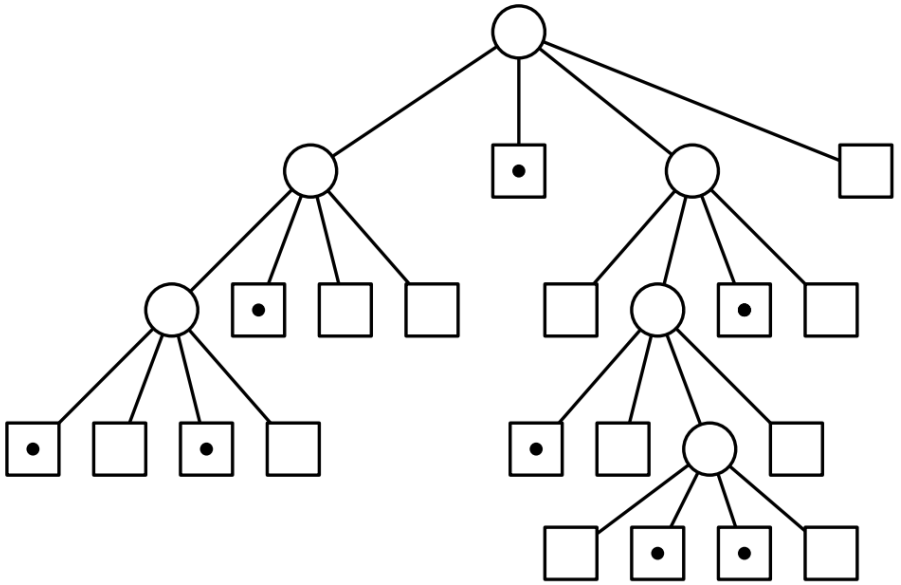
$O(n^2)$ is too bad when we have hundred thousand triangles.

Good news: this worst case behavior does not happen in practice.

If the objects are "relatively well separated" in a sense that can be made precise, the size of the BSP tree is provably small (almost linear).

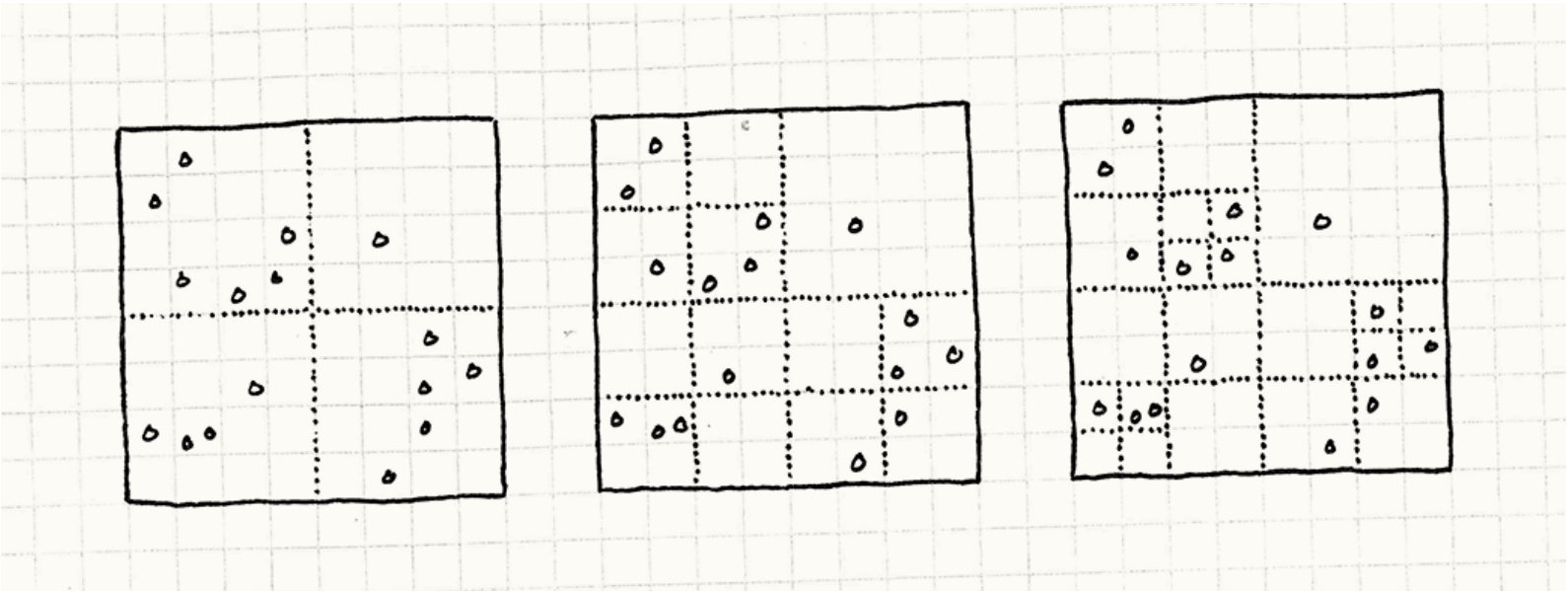
Quadtree

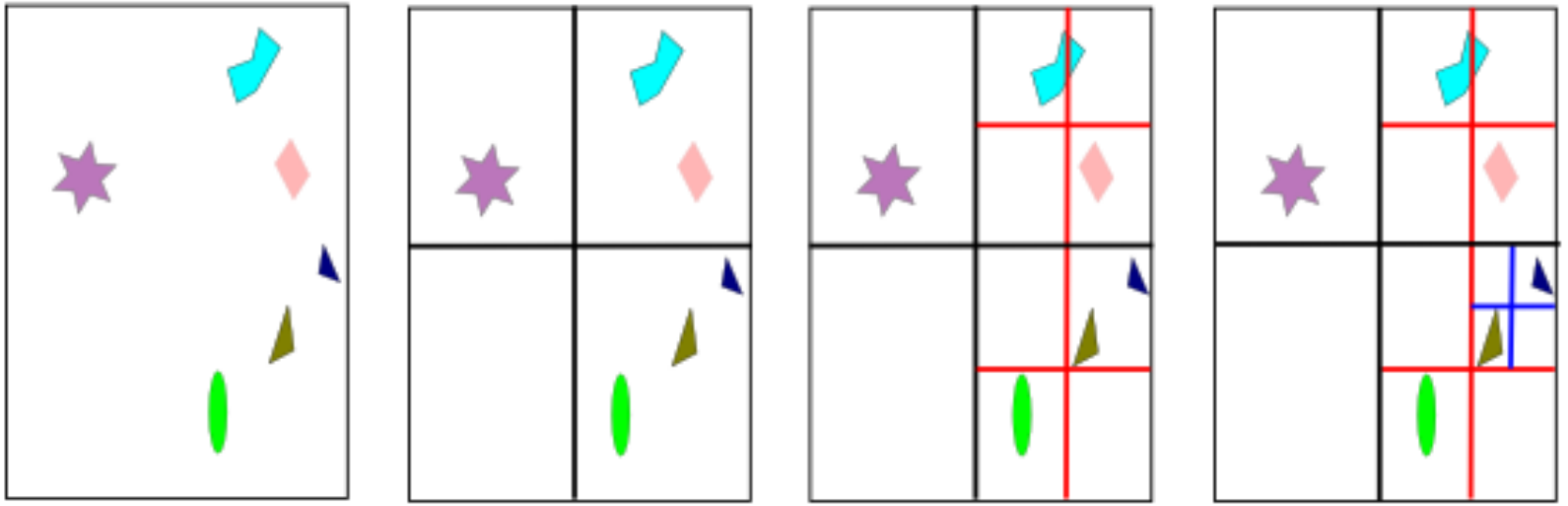




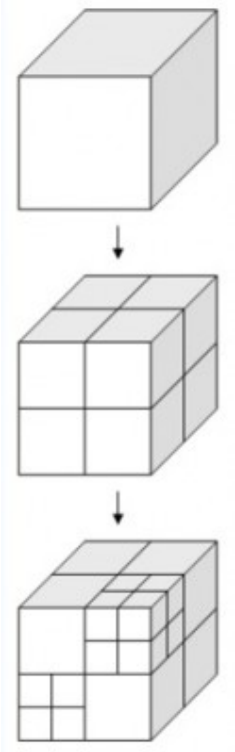
Divide the bounding box into four equal pieces. Subdivide each box that contains more than a constant number of objects.

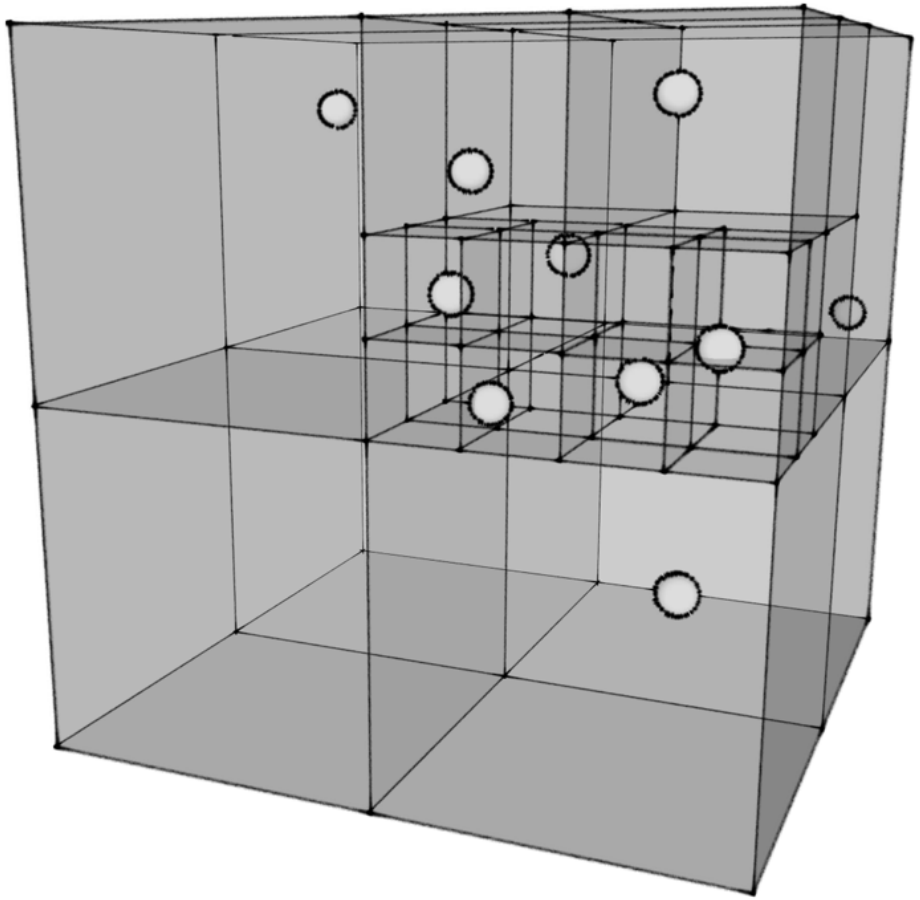
Spatial Data Structures



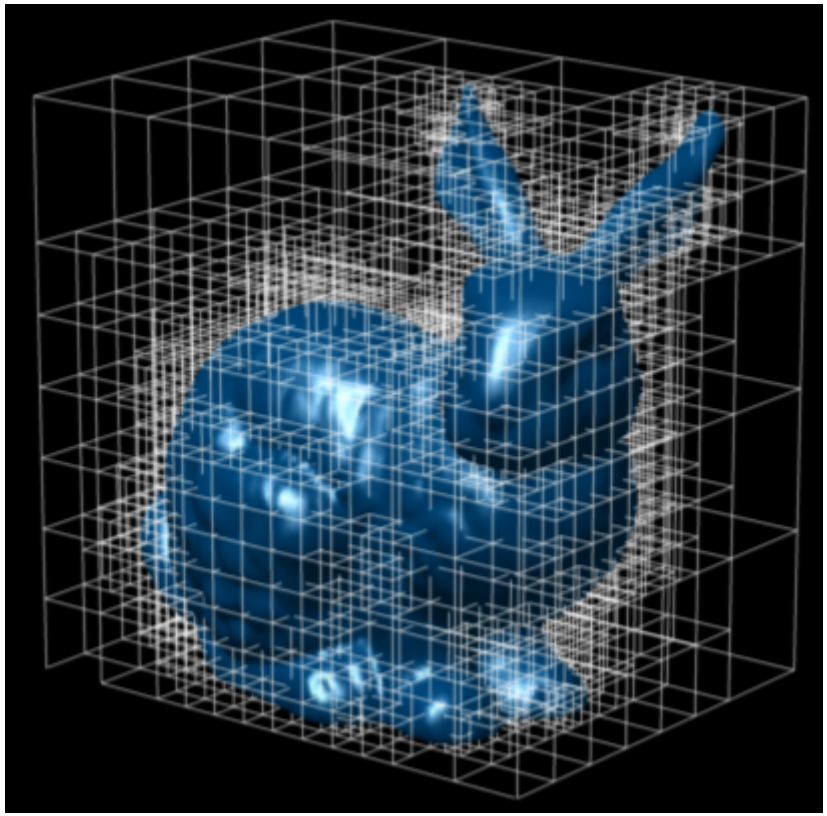


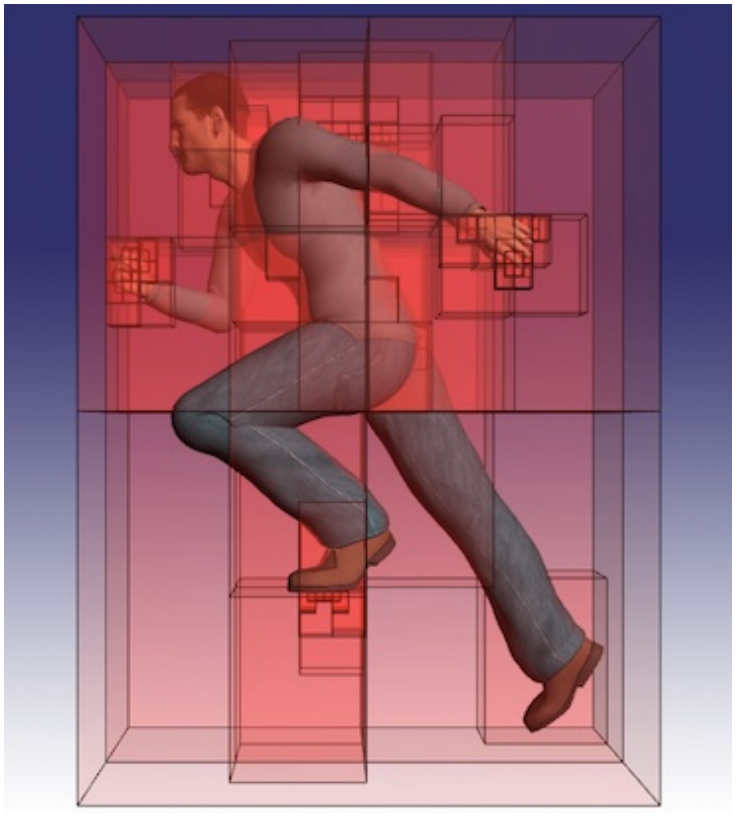
Octree





Spatial Data Structures



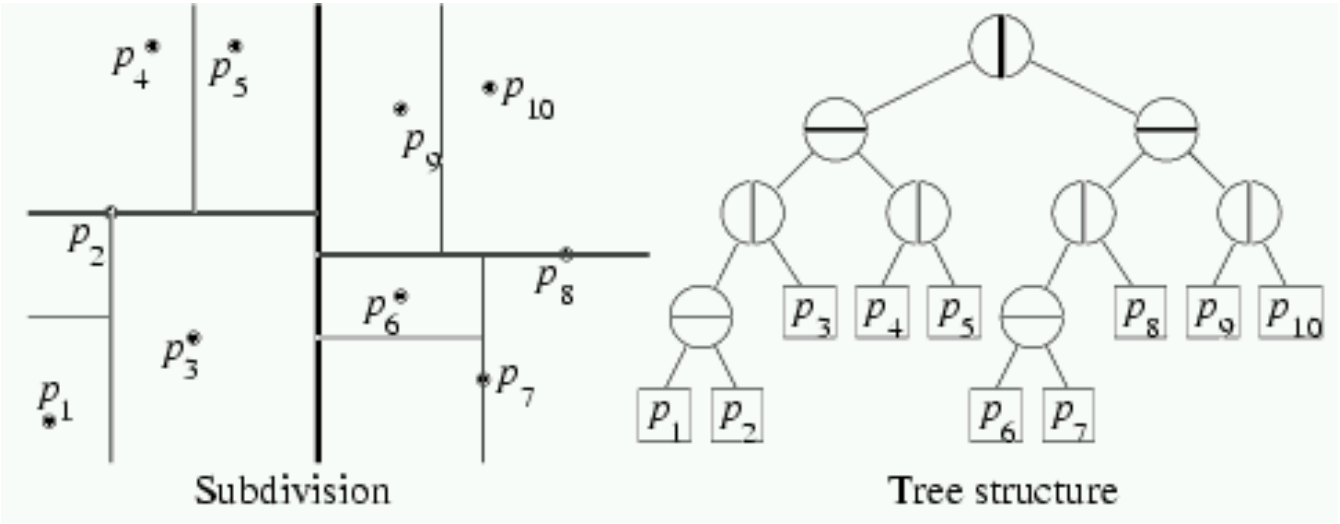


Advantages over uniform spatial subdivision?

Faster and requires less space.

Spatial Data Structures

kd trees (k dimensional trees) (*also known as bintree)*

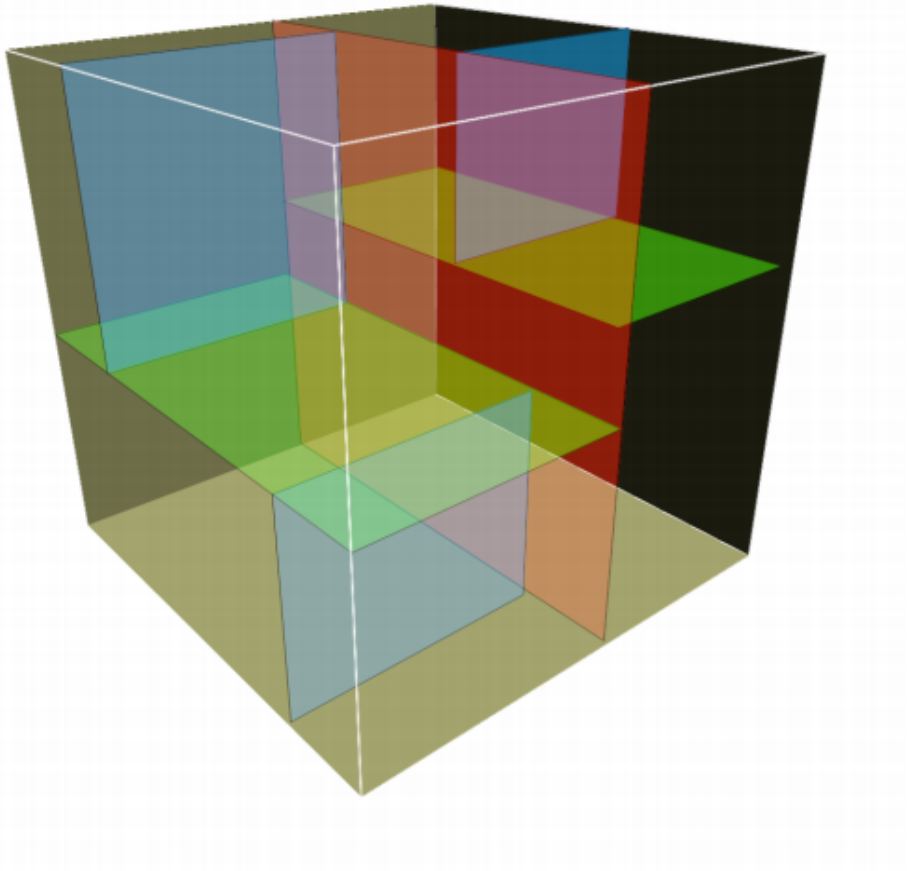


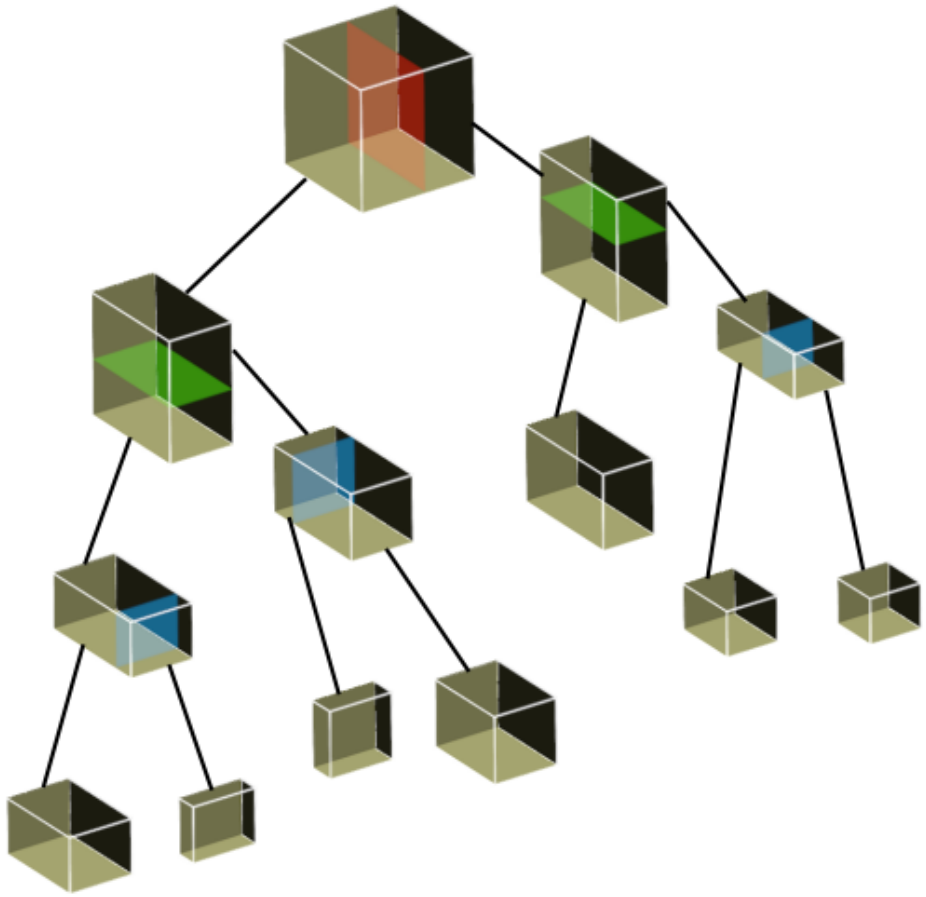
Each node corresponds to a plane perpendicular to one of the axes.

Various criteria could be used for finding the cutting plane.

- *direction along which the magnitude is maximum*
- *direction along which the median split has maximum gap*

Spatial Data Structures





Constructing a good kd Tree

Typically constructed top-down

Start with an object soup and recursively subdivide.

Surface Area Heuristic (SAH)

We aim to minimizes a certain cost function.

Cost of leaf node N: $C_N = \sum T_i$

T_i: *cost of intersection test for an object in the leaf node N*

Cost of interior node N: $C_N = C_{TS} + p_L \times C_L + p_R \times C_R$

C_{TS}: *cost of interior node traversal (intersection test)*

C_L: *cost of traversing left child*

C_R: *cost of traversing right child*

p_L: *probability that the left child is hit given that the parent node is hit*

p_R: *probability that the right child is hit given that the parent node is hit*

How do we estimate p_L and p_R?

$$p_L = \frac{\text{surface area of the left child}}{\text{surface area of parent}} \qquad p_R = \frac{\text{surface area of the right child}}{\text{surface area of parent}}$$

By "surface area of the left child" we mean the total surface area of all the objects in the left child.

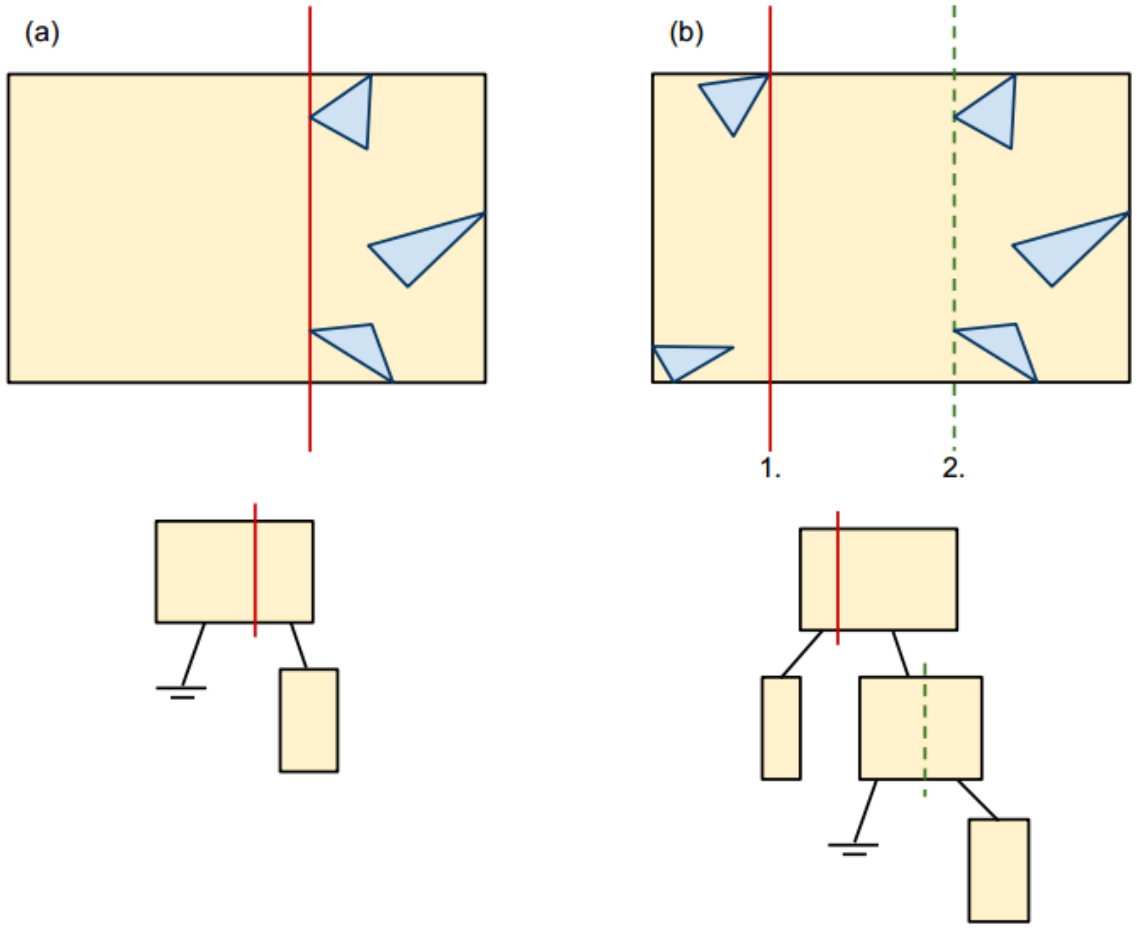
Justification: for a convex object the probability that a "random" ray intersects it is roughly proportional to its surface area.

SAH seems to be a good trade-off between simplicity and accuracy and performs very well in practice.

For offline constructions, all splitting positions can be evaluated.

For real-time constructions, only a few are used.

Another heuristic : Cut off empty spaces early



Distribution Ray Tracing

Problems with simple ray tracing:

- Jagged (uneven) edges
- Hard Shadows (too sharp)
- No motion blur
- Everything is in focus
- Perfectly shiny surfaces
- Perfectly clear glass

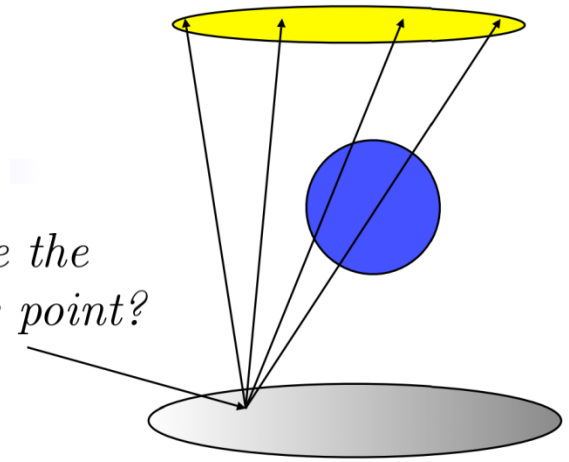
Distribution Ray Tracing:
(a.k.a Distributed Ray Tracing)

Instead of a single ray, use a distribution of rays.

Distribution Ray Tracing

Shadows

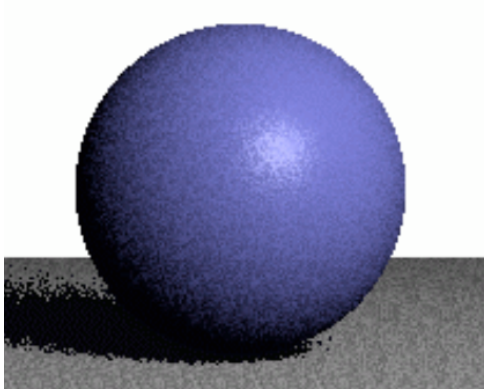
Point light sources are unrealistic.
We typically have area light sources.

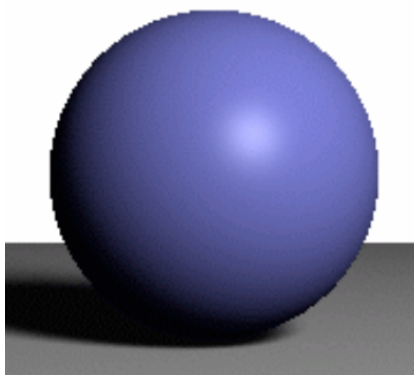


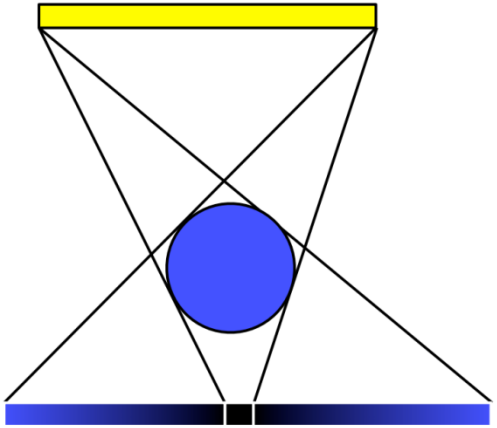
Cast many shadow rays from the surface to different points on the light source.

$$\text{Intensity} \propto \frac{\#\text{ hits}}{\#\text{ rays}}$$

We need to jitter the points to avoid aliasing.


On shadow ray


Many shadow rays

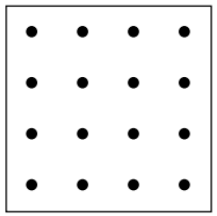


Distribution Ray Tracing

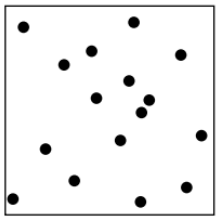
Antialiasing

Supersampling - multiple rays per pixel

regular sampling



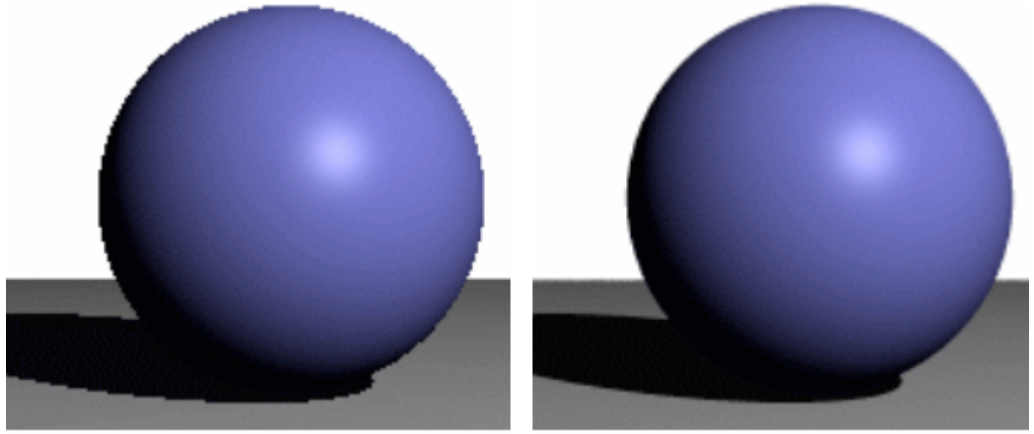
random sampling



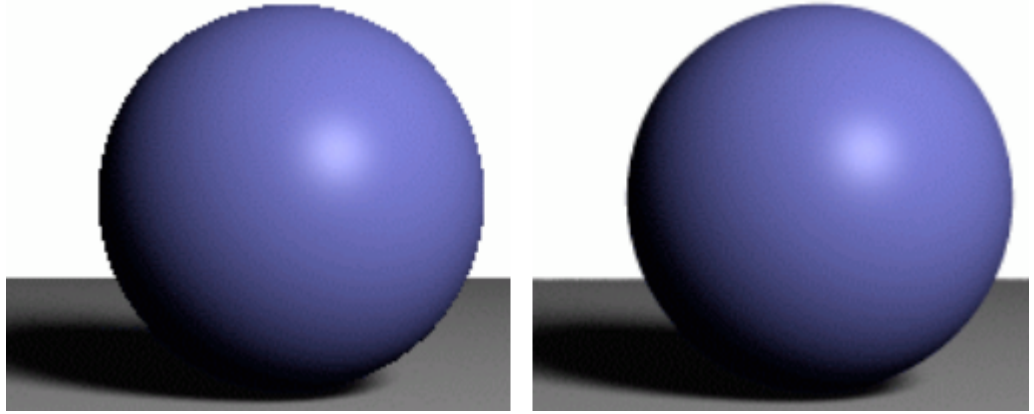
Single Ray

Multiple Rays

Point light



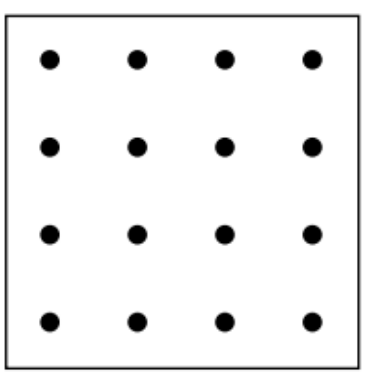
Area light



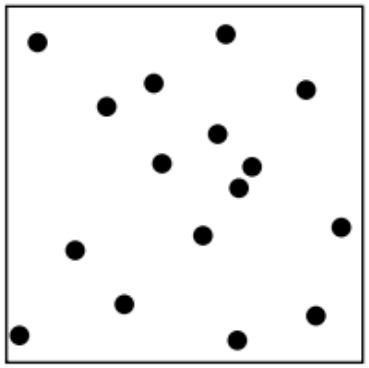
Distribution Ray Tracing

Antialiasing

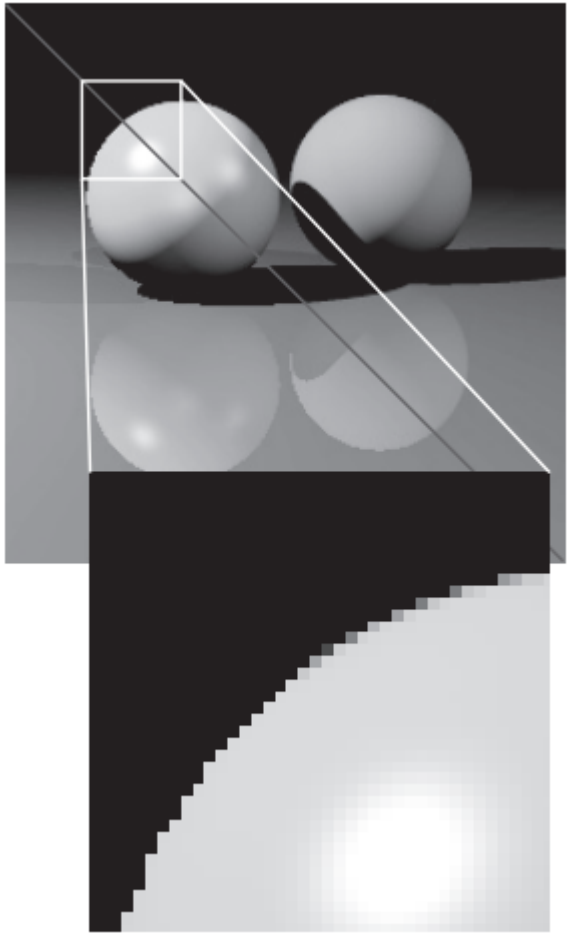
Supersampling - multiple rays per pixel



regular sampling



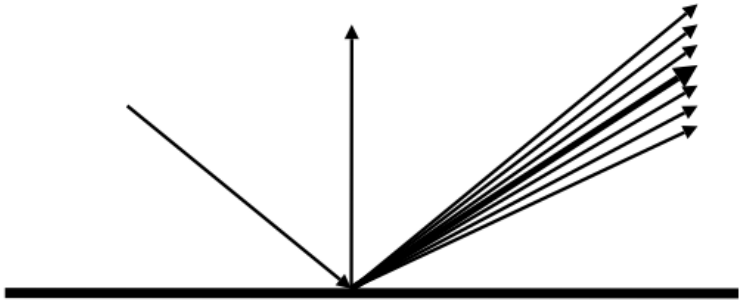
random sampling



Regular sampling can create regular artifacts.

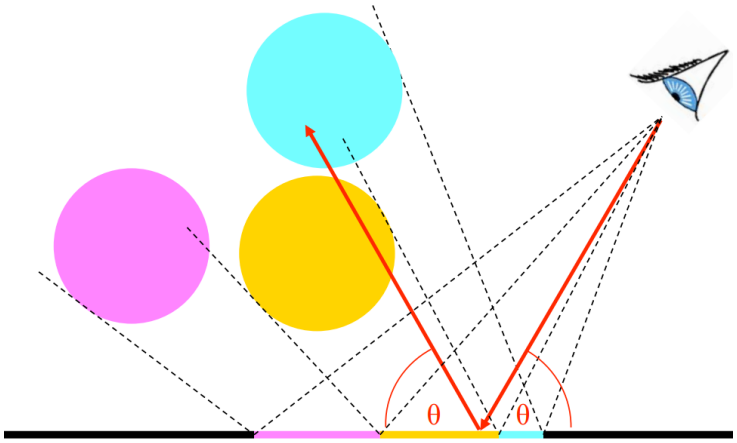
Distribution Ray Tracing

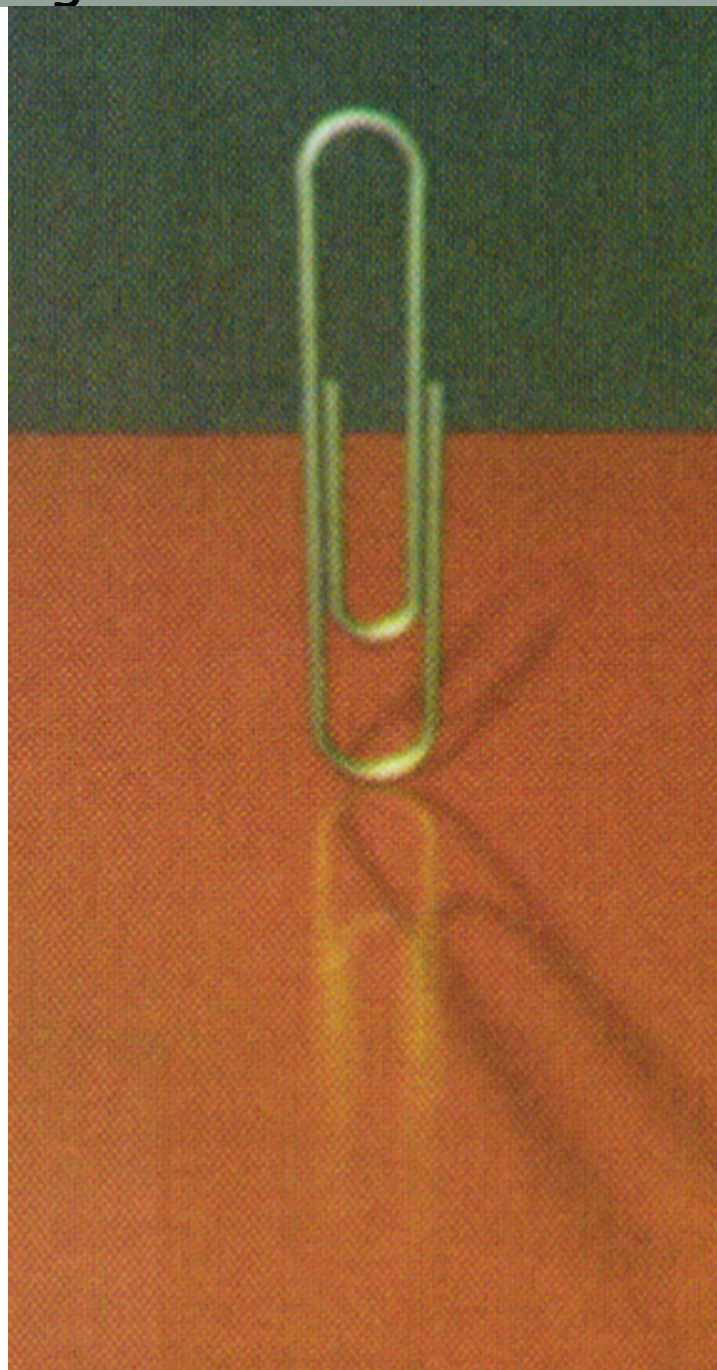
Reflections



Due to unevenness in surface (microfacets) the reflected rays go in many directions.

Farther objects are more blurry than nearby objects.

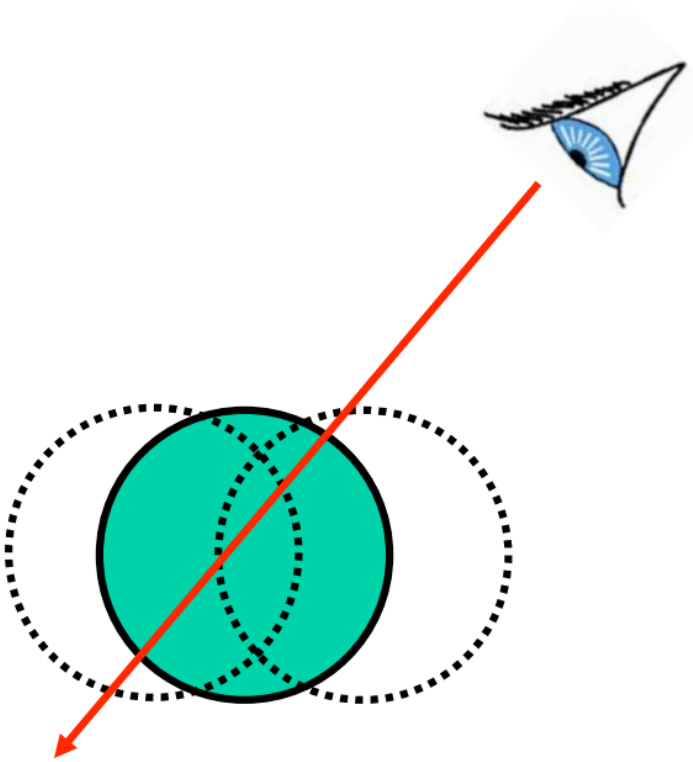




Motion Blur

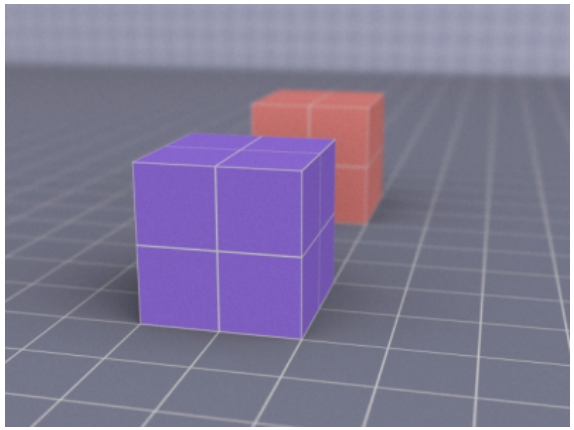
Cast multiple rays at different times.

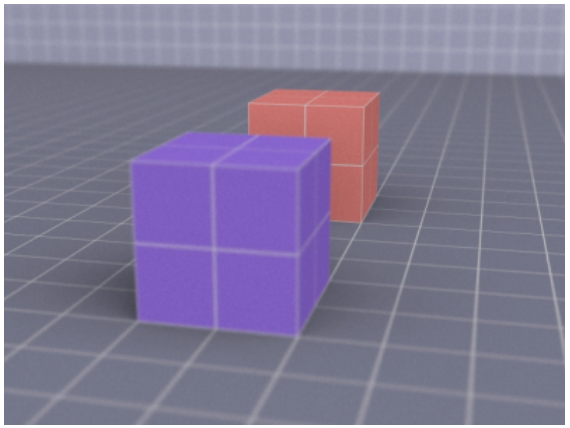
Take the average.

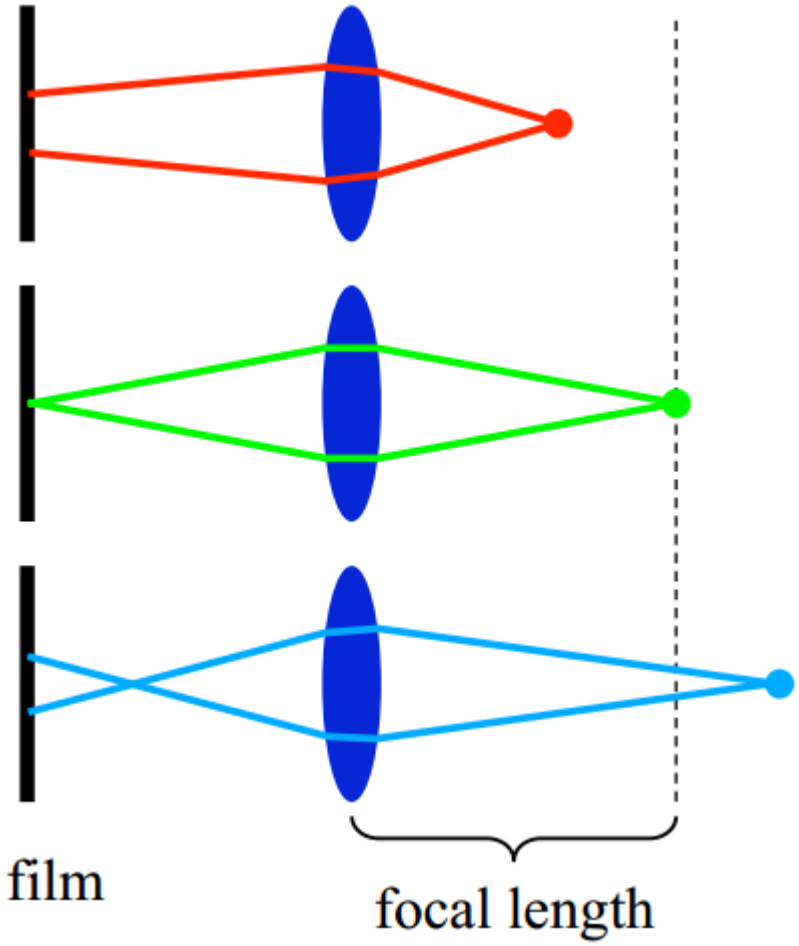


Distribution Ray Tracing

Depth of Field



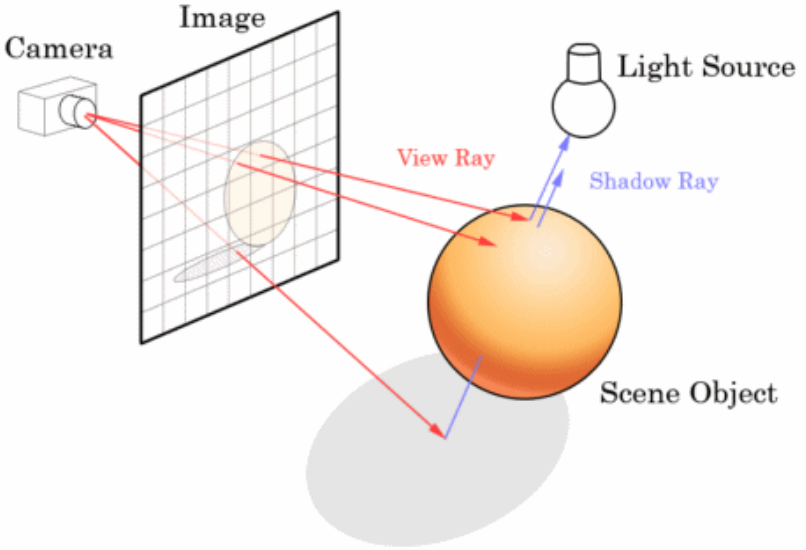




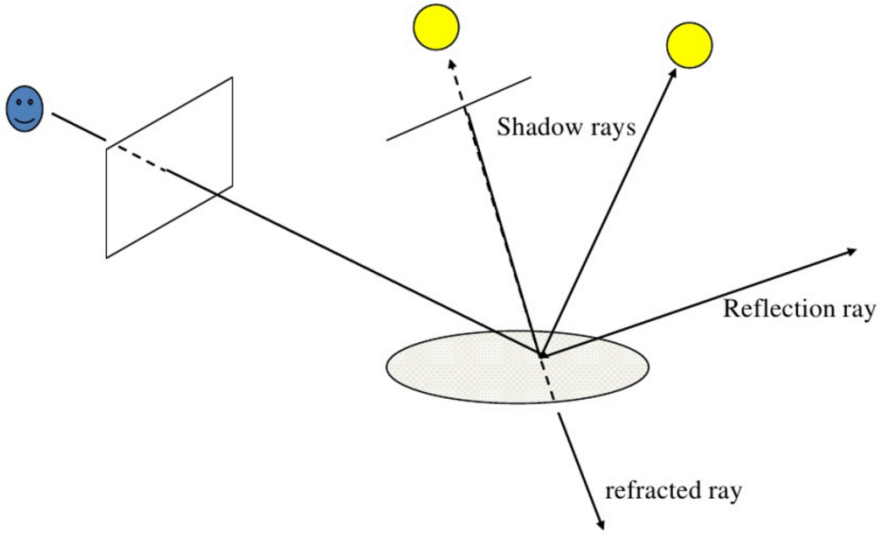
Multiple rays are used for every pixel.

Sample points on the lens and trace through them.

Ray Tracing



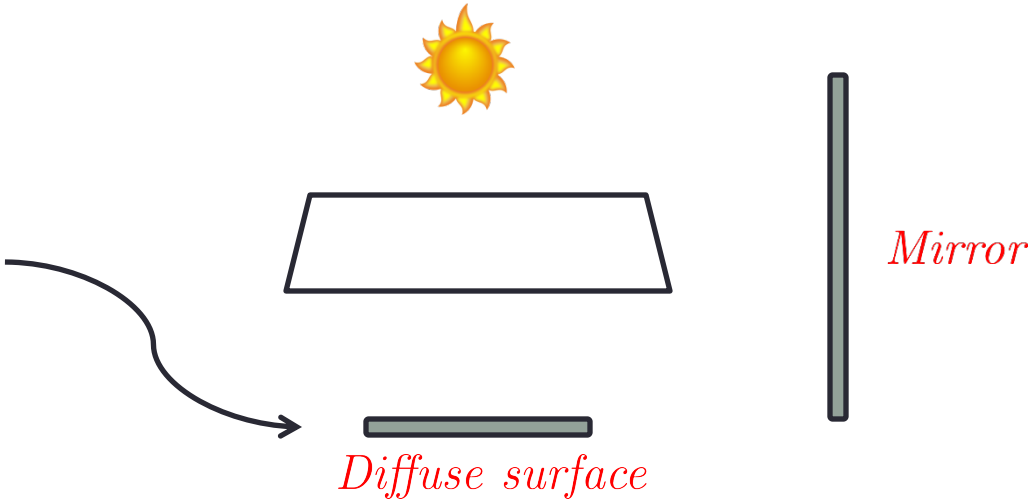
Ray Casting



Recursive Ray Tracing

We don't trace the scattered rays: too many rays. *This leads to inaccuracies.*

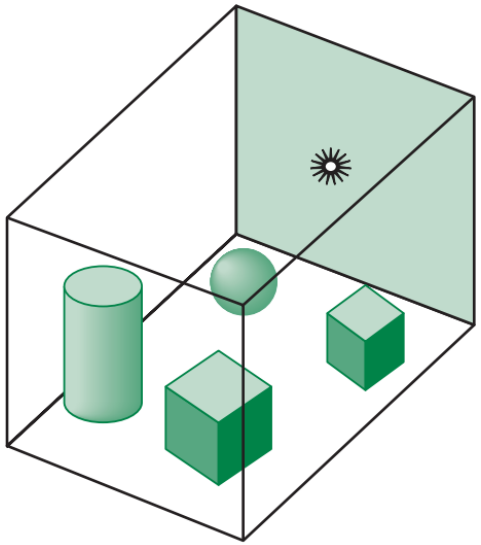
This surface gets reflected light from the mirror but our algorithm doesn't see it.



Rendering Equation

A surface can absorb, emit, reflect or refract light.

(*light sources emit light*)



If we look at a closed environment then light bounces around but stays in the environment.

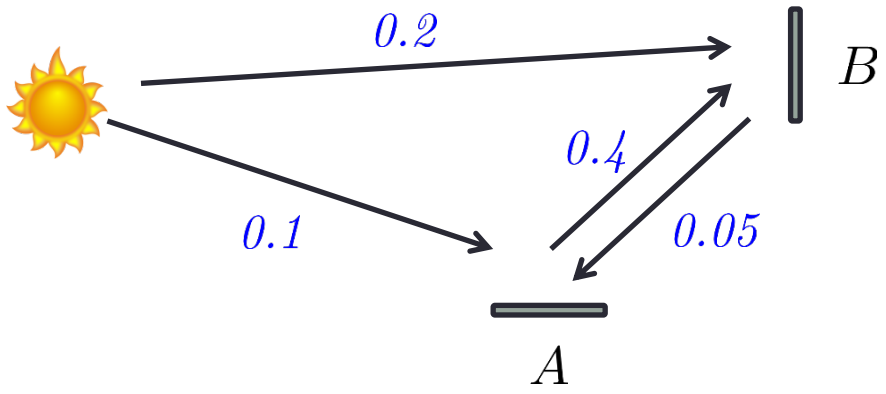
So, the total energy is conserved.

We want to calculate how much energy arrives per unit time at each of the points in the steady state.

We know the relations between the intensities at different points and from this we can calculate the intensities in the steady state.

Rendering Equation

Simplified Example: Consider two patches of surfaces A and B.



Consider the following hypothetical situation:

The source emits one unit of energy per unit time.

20% of the light emitted by the source falls on B

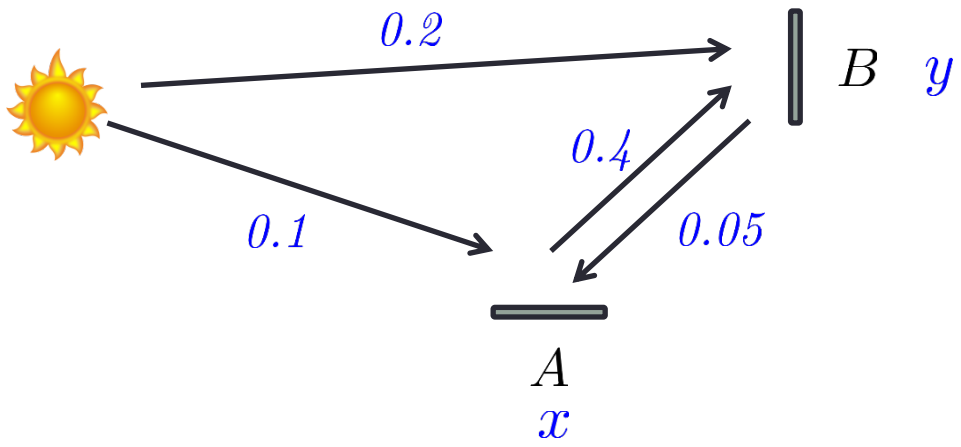
10% of the light emitted by the source falls on A

40% of the light received at A is reflected towards B

05% of the light received at B is reflected towards A

What is the amount of light energy received at A and B per unit time?

Rendering Equation



Let x and y be the amount of light energy received per unit time at A and B respectively.

Then,

$$x = 0.1 + 0.05y$$

$$y = 0.2 + 0.4x$$

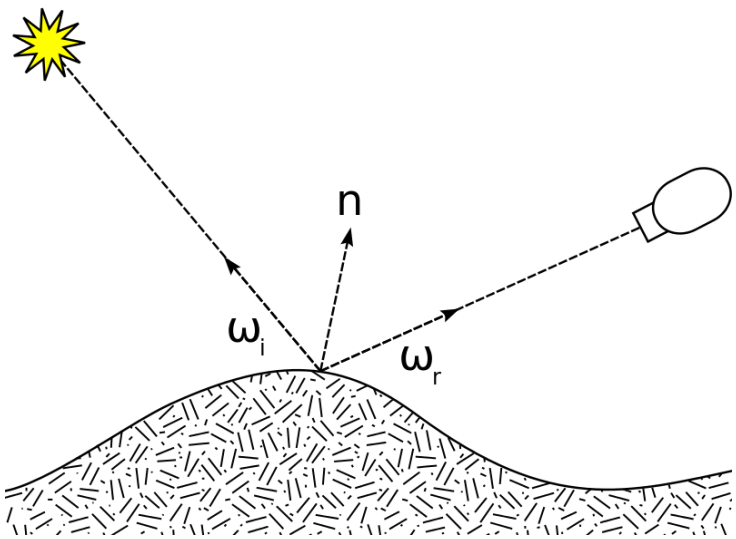
$$\Longrightarrow \quad x \approx 0.112, \quad y \approx 0.245$$

Rendering Equation

Consider infinitesimally small patches and write equations for every patch.

We need to know what fraction of light received at a point goes in each direction.

How light is scattered by the material at a given point is given by the **bidirectional reflection distribution function** (BRDF).

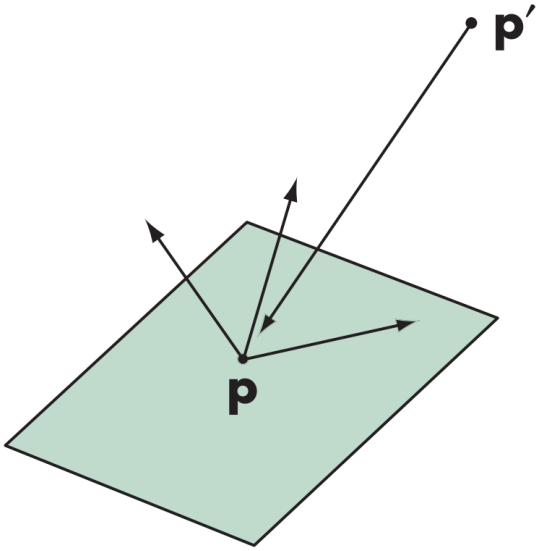


The BRDF tells us what fraction of the incoming light coming from direction ω_i is reflected along direction ω_r.

The directions ω_i and ω_r can be specified by two angles each (spherical coordinates).

This function may change from point to point on the surface.

Rendering Equation



Consider two points p and p'.

Let $i(p, p')$ denote the amount of light received by p from p'.

$i(p, p')$ depends on the amount of light emitted by p' towards p, if p' is a source.

It also depends on the amount of light p' received from another point p'' and what fraction of that was reflected towards p.

$\rho(p, p', p'') =$ fraction of light received by p' from p'' that is reflected towards p. *(given by BRDF)*

$\varepsilon(p, p') =$ amount of light emitted by p' towards p

$$i(\mathbf{p}, \mathbf{p}') = \nu(\mathbf{p}, \mathbf{p}') \left(\epsilon(\mathbf{p}, \mathbf{p}') + \int \rho(\mathbf{p}, \mathbf{p}', \mathbf{p}'') i(\mathbf{p}', \mathbf{p}'') \, d\mathbf{p}'' \right)$$

$\nu(p, p') = 0$ is p and p' cannot see each other and $1/r^2$ otherwise, $r = dist(p, p')$.

Rendering Equation

The rendering equation cannot be analytically solved in general.

Various techniques are used to approximate the solution numerically.

Photon Mapping:

Follow photons from the light source until they are absorbed.

When a photon strikes a surface, the BRDF gives a probability distribution on the reflected directions the photon can take.

The photon takes a random direction according to this distribution.

A similar calculation is done for refraction.

There is also some probability of the photon being absorbed by the surface.

A large number of photons are traced to figure out the amount of light received at every point in the scene.

Various optimizations are used to make this feasible.

Rendering Equation

Radiosity: Assumes that all surfaces are perfectly diffuse reflectors.

The BRDF is very simple.

Consider a scene with n patches numbered 1 to n.

a_i: *area of patch* i.

b_i: *intensity (energy/unit time) per unit area leaving patch* i.

total intensity leaving patch $i = b_i a_i$.

e_i: *intensity of light emitted per unit area by patch* i.

ρ_i: *reflectivity of patch* i

f_{ij}: *fraction of intensity leaving patch* i *that reaches patch* j.

known as **form factor**, *depends on the orientation and mutual visibility.*

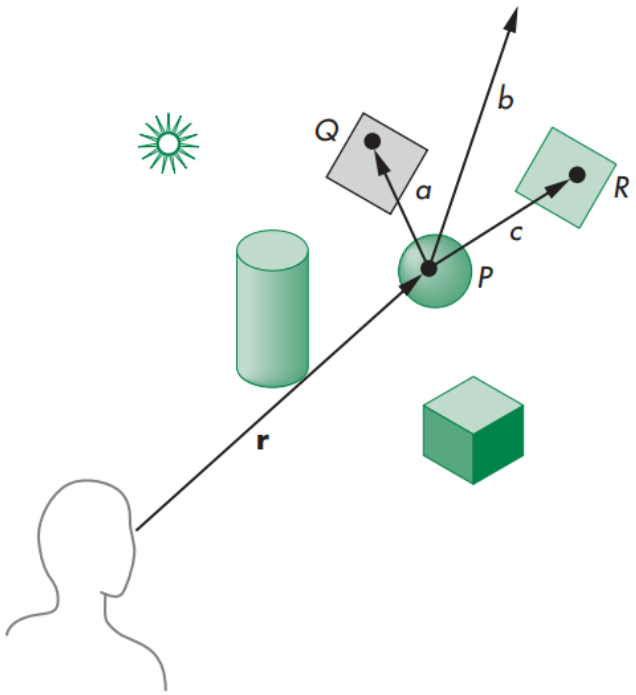
Then, $$b_i a_i = e_i a_i + \rho_i \sum_{j=1}^{n} f_{ji} b_j a_i \implies \boxed{b_i = e_i + \rho_i \sum_{j=1}^{n} f_{ji} b_j}$$

Radiosity Equation

Rendering Equation

Path Tracing:

Same as ray tracing except when we reflect, we use the BRDF to pick a random direction.



Continue until we hit a source or it is clear that we are heading to infinity.

Need to use extended light sources, otherwise 0 probability of hitting one.

Keep track of absorption losses at every surface the ray strikes.