Lecture 22: Ray Tracing

Yuhao Zhu

http://yuhaozhu.com
yzhu@rochester.edu

CSC 259/459, Fall 2025
Computer Imaging & Graphics

The Roadmap

Theoretical Preliminaries

Human Visual Systems

Display and Camera

Modeling and Rendering

3D Modeling

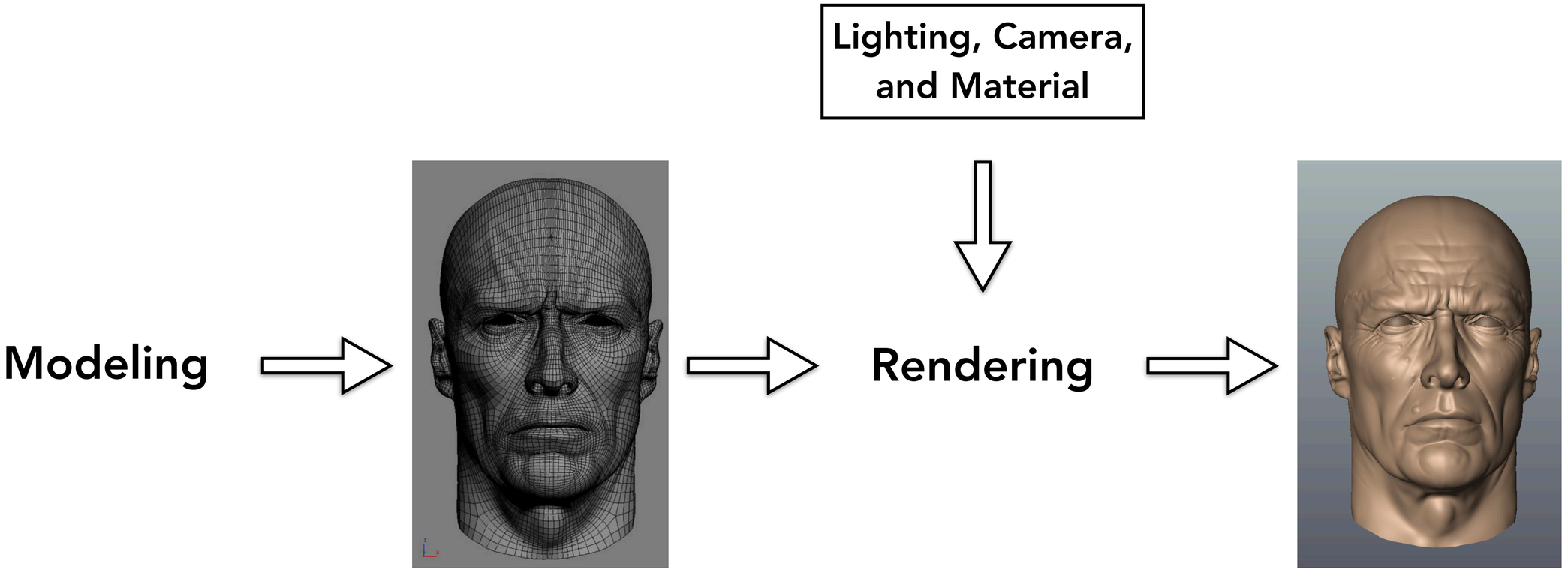
Rasterization and GPU

Ray Tracing

Shading

Rendering in AR/VR

Graphics



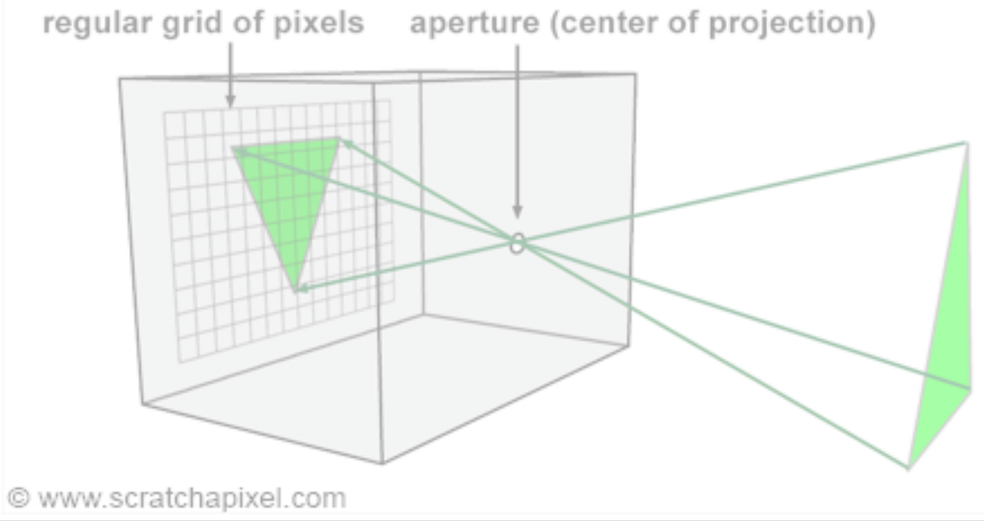
https://docs.blender.org/manual/en/dev/render/introduction.html http://www.cgarena.com/freestuff/tutorials/max/thomas_highway/sergeant.html

Visibility Problem

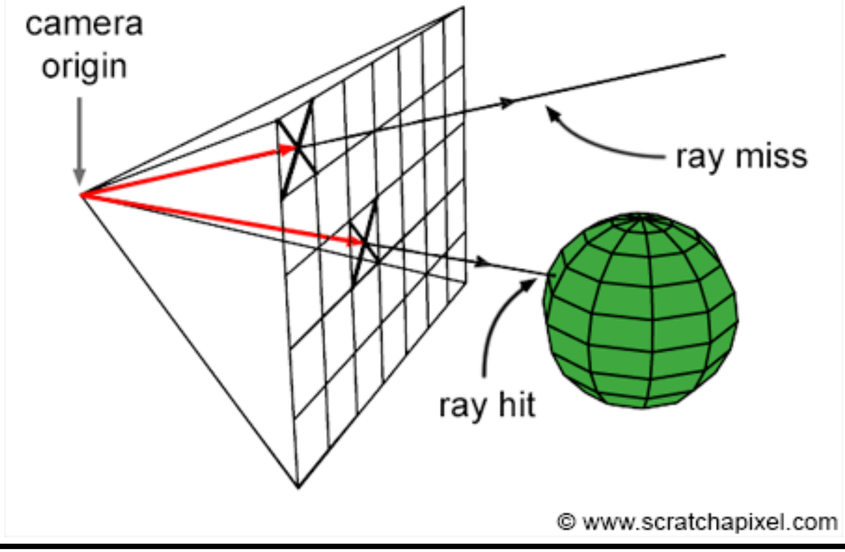
Two fundamental classes of visibility algorithms

- Object-centric (Rasterization)
- Image-centric (Ray tracing)

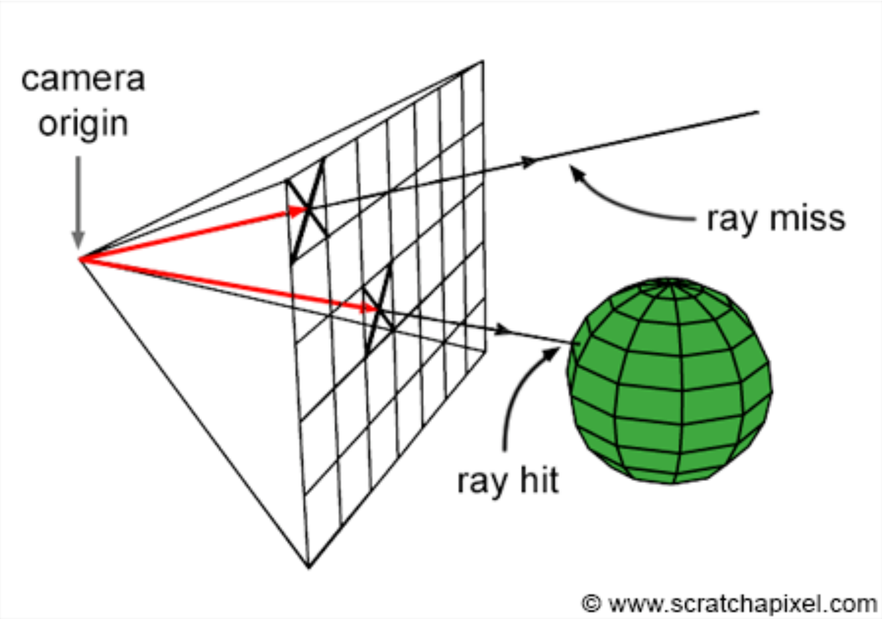
Given a point P [x, y, z], what's the corresponding pixel coordinates [u, v] on the camera sensor?



Given a pixel [u, v] on the sensor, what's the associated point in the scene?

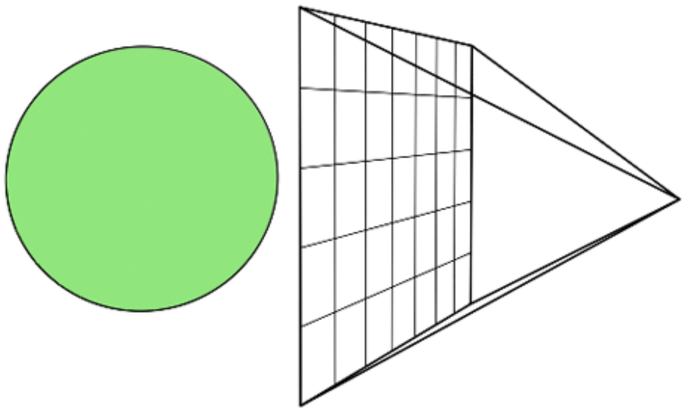


Basic Idea of Ray Tracing




```
foreach pixel in image
  ray = buildRay(camera, pixel)
  if (P = intersect(ray, mesh))
    pixel.color = shade(P)
  else
    pixel.color = backgroundColor
```

Basic Idea of Ray Tracing



```
foreach pixel in image
  ray = buildRay(camera, pixel)
  if (P = intersect(ray, mesh))
    pixel.color = shade(P)
  else
    pixel.color = backgroundColor
```

Can sample multiple rays per pixel for anti-aliasing

Generating Rays

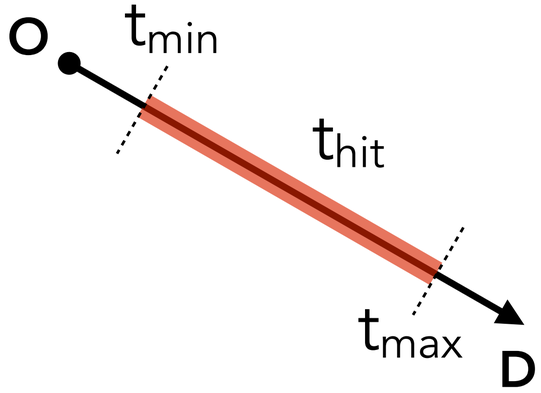
What Defines a Ray?

A ray is defined by its original and the directional vector.

We usually define ray as a segment, with a min and a max.

- Its purpose will become clear later, but briefly it allows us to reduce computation in visibility test.

Ray: **O** + t**D**, t_{min} <= t <= t_{max}

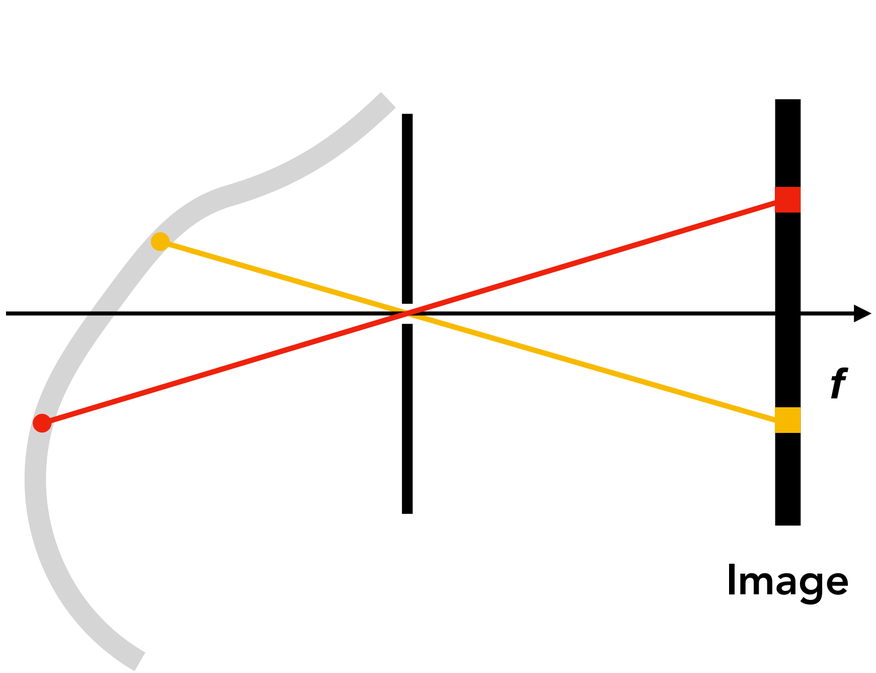


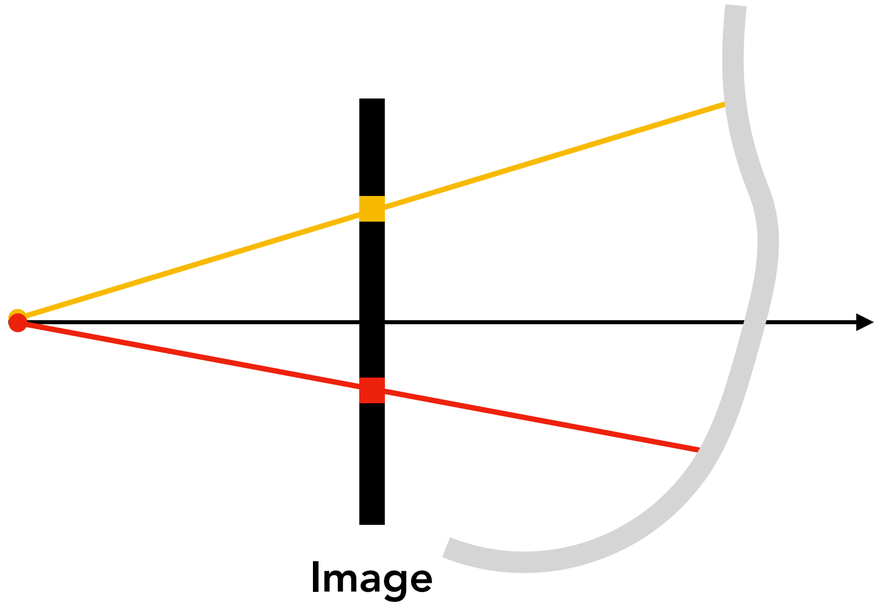

```
class Ray {
  …
  Vec3f O;
  Vec3f D;
  float tmin;
  float tmax;
};
```

Generating a Ray in a Pinhole Camera

Very intuitive: connect a pixel and the pinhole to form a ray.

Remember in actual implementations the canvas is before the pinhole.

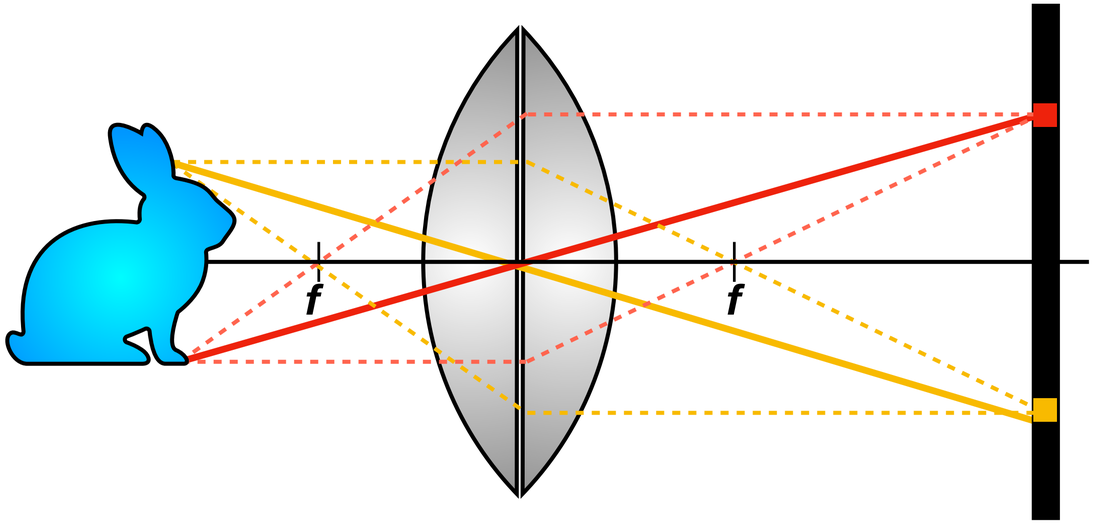




Generating a Ray Under an Ideal Thin Lens

Many rays incident on one pixel. So for each pixel we need to sample the lens multiple times to trace **multiple rays**.

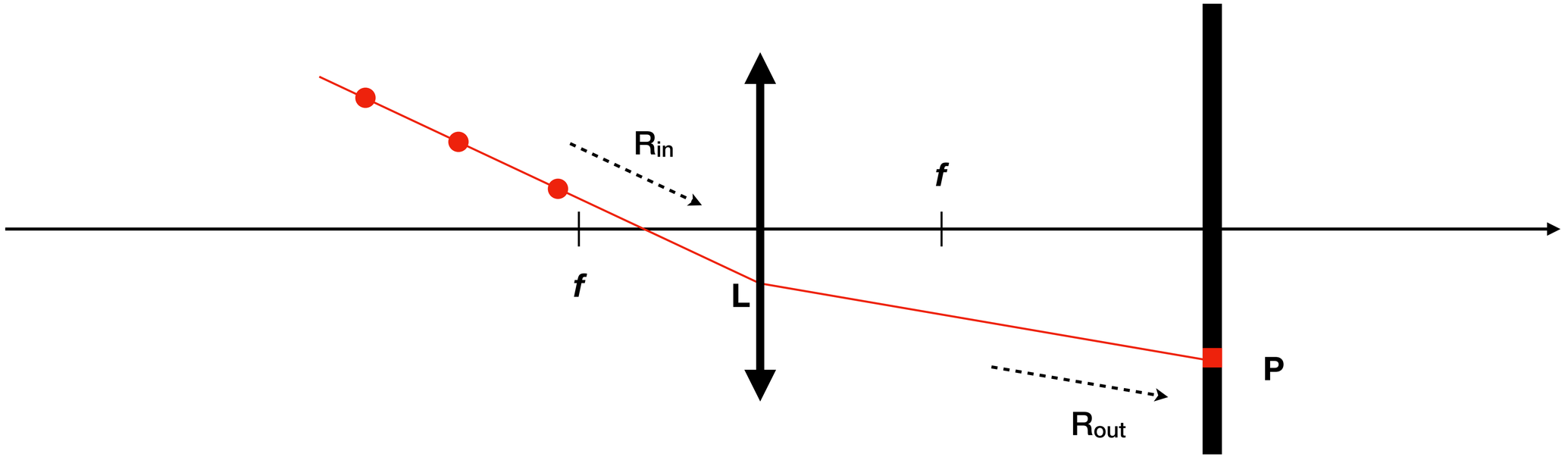
- Why trace many rays? Because the pixel color depends on all incident rays.
- This is not something achievable using only the perspective matrix.



Generating a Ray Under an Ideal Thin Lens

Goal: given an arbitrary R_{out} how do we find R_{in}?

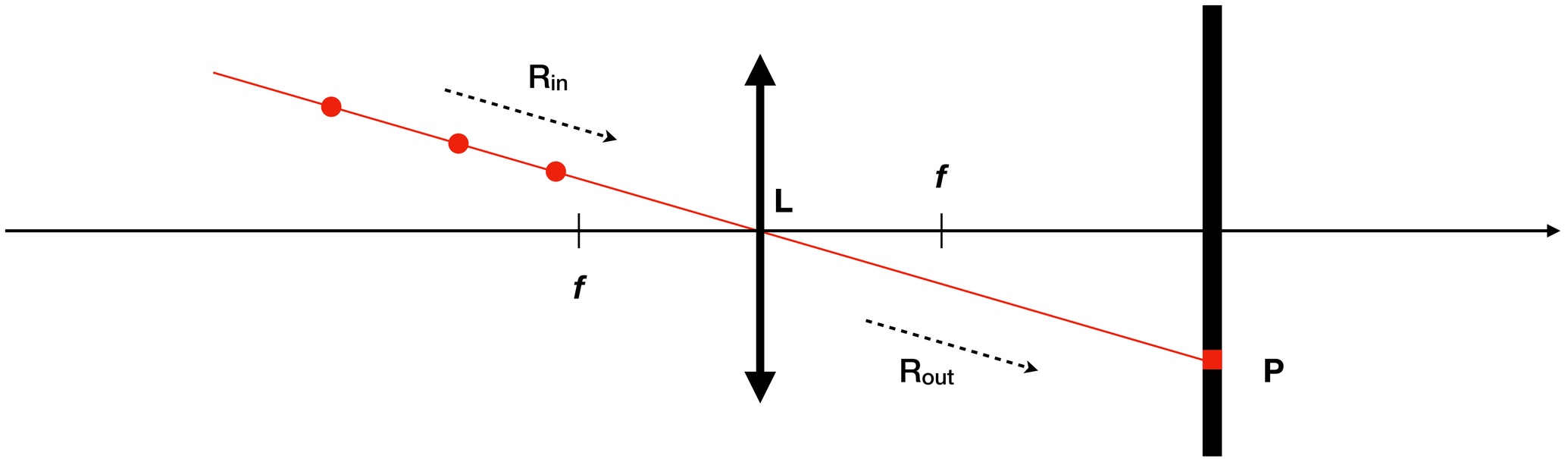
- There is a unique ray, R_{out}, between a pixel **P** and a sample **L** on the lens
- There is a unique ray, R_{in}, going into the lens that generates R_{out}
- The closest point on R_{in} before the lens will hit **P**



Generating a Ray Under an Ideal Thin Lens

Use geometrical optics principles to determine R_{in} for a given R_{out}

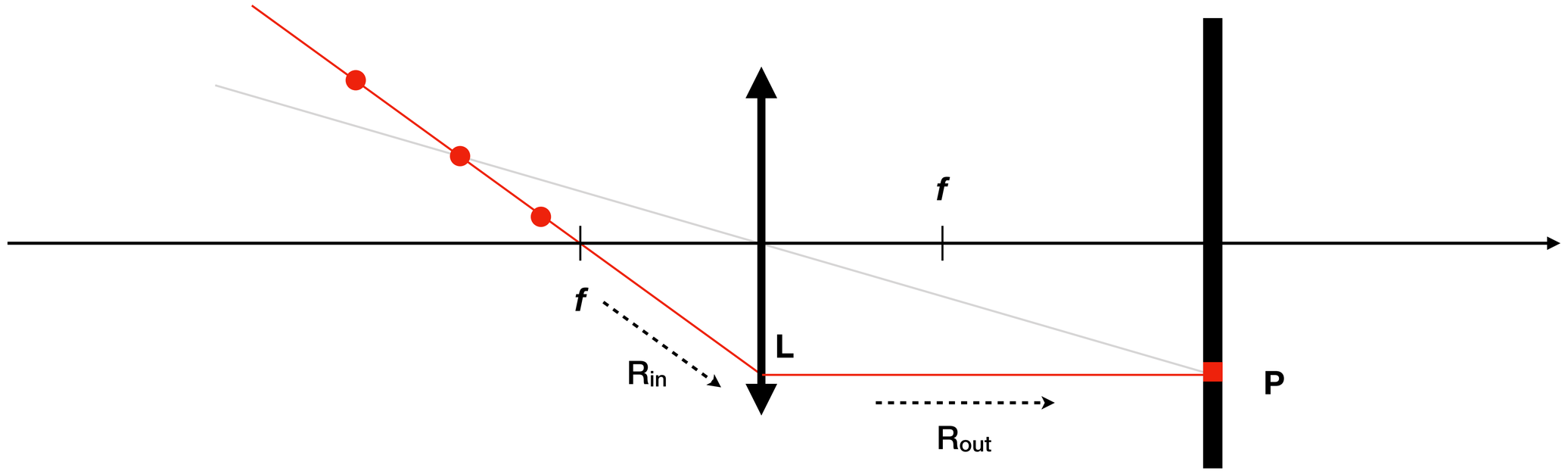
- Rays go through the lens center (chief ray) don't change their directions



Generating a Ray Under an Ideal Thin Lens

Use geometrical optics principles to determine R_{in} for a given R_{out}

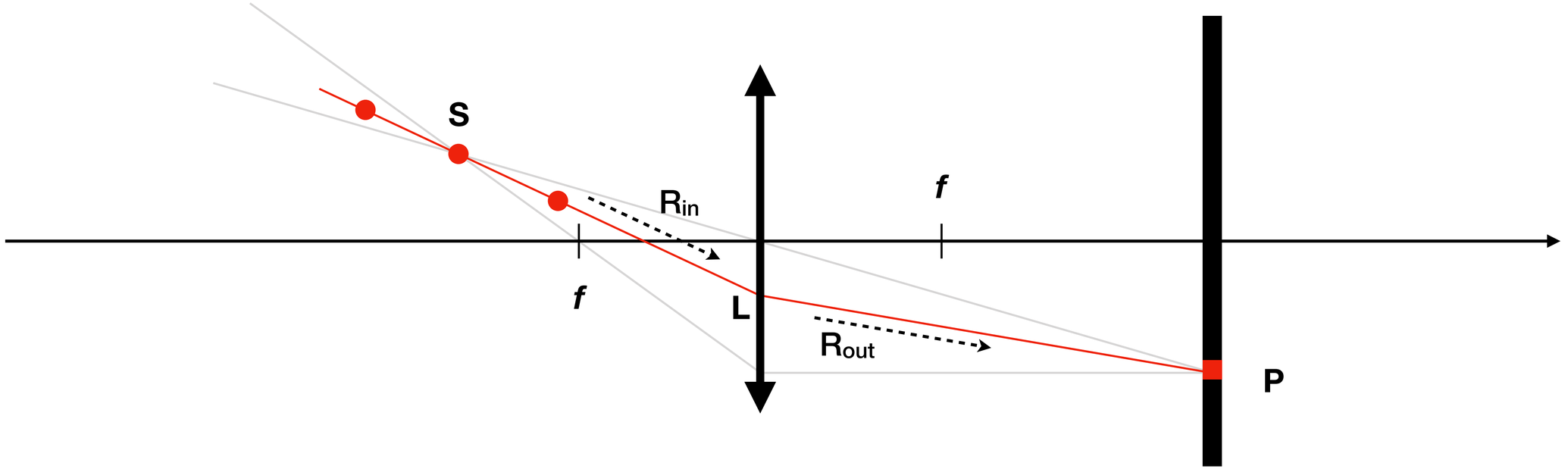
- Rays go through the lens center (chief ray) don't change their directions
- Rays parallel to the optical axis (parallel ray) pass through the lens focus



Generating a Ray Under an Ideal Thin Lens

Steps:

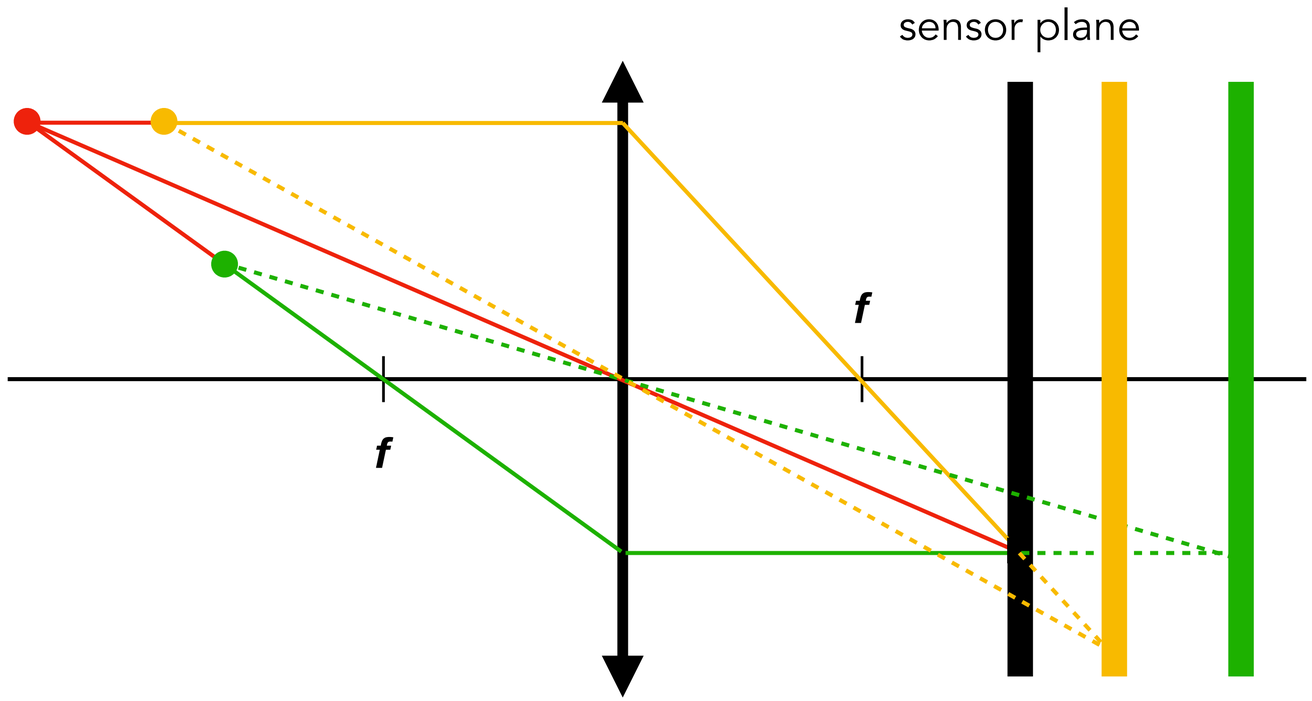
- 1. use chief and parallel rays to find the intersection point **S** in the scene
- 2. find **L** on the lens from R_{out}
- 3. R_{in} is the ray between **S** and **L**



Can Ray Tracing Capture DOF?

What's described before doesn't rely on whether the closest hit is actually in-focus on the pixel.

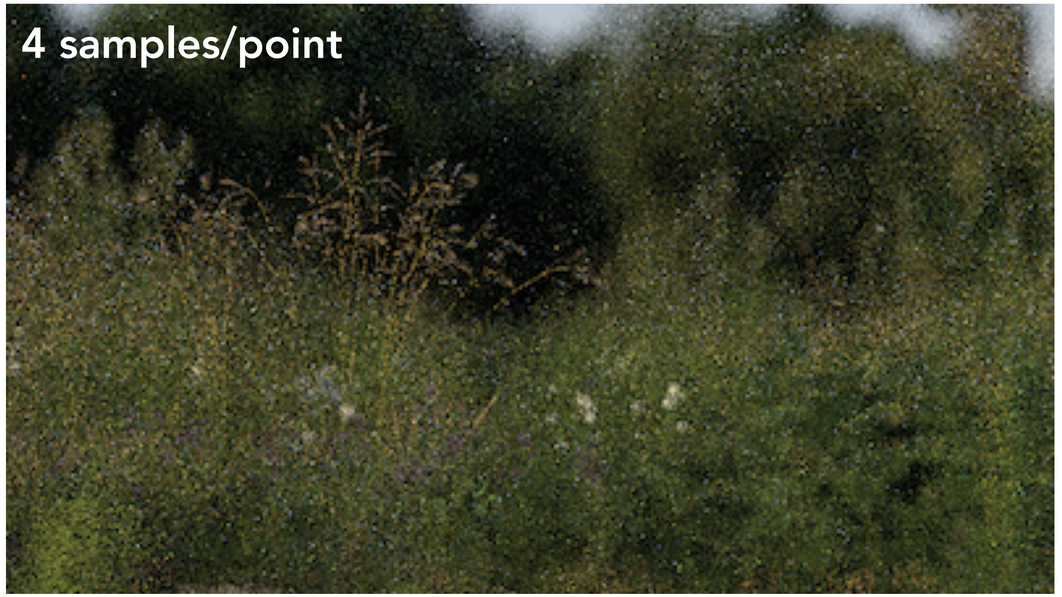
- So it can inherent trace scene points that are out-of-focus, i.e., simulate depth of field.



Lens Sampling

Each sensor plane point receives infinitely many rays, so we need to sample many rays for a point to reduce noise (more in shading lecture).

- This is orthogonal to sampling multiple points per pixel.



* the artifacts from low sampling rate here is not aliasing; it's due to high variance in Monte Carlo integration.

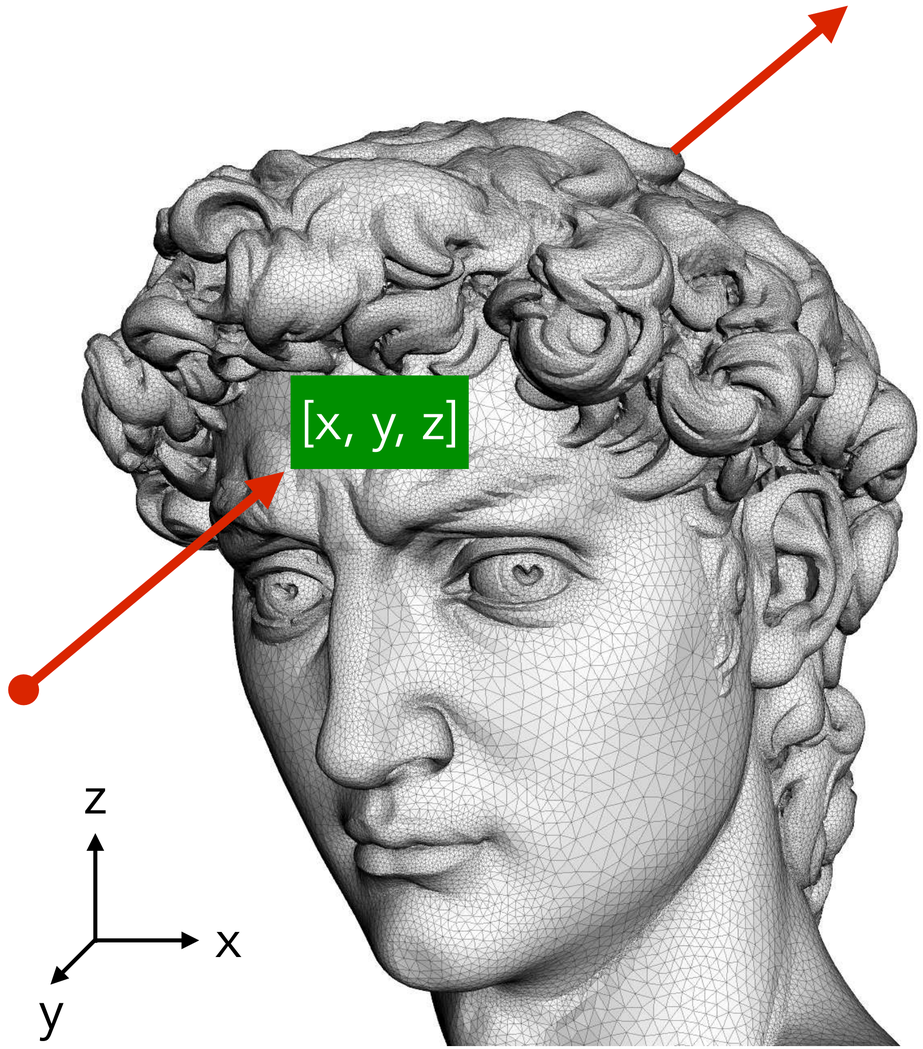
Ray-Scene Intersection

Ray-Scene Intersection

Goal: calculate the [x, y, z] coordinates of the closest hit between the ray and the mesh.

Why closest hit?

- Preserve visibility (like the z-buffer in rasterization)



Valette, et al. [TVCG'08]

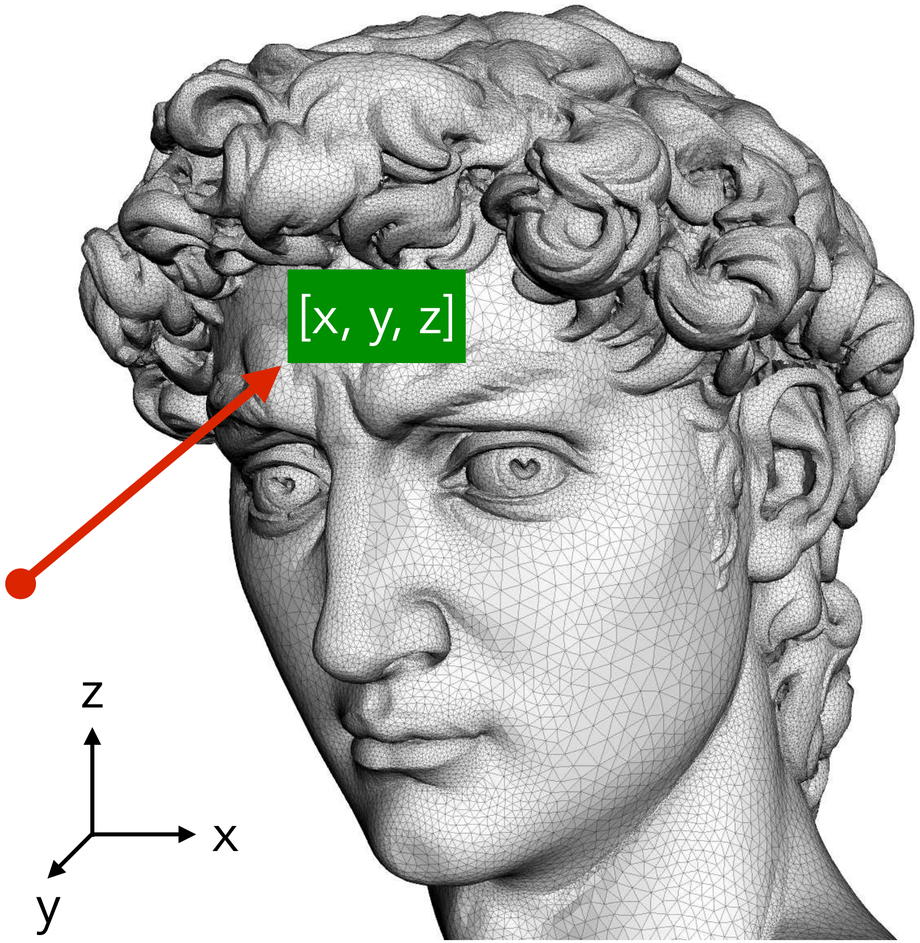
The Simplest Algorithm

Brute-force approach:

- iterate all triangles
- test intersection for each triangle
- return the closest hit, if any

Key task:

- Ray-triangle intersection test and calculate the coordinates of the hit point, if any.
- General plan:
  - Identify the intersection point between the ray and the plane in which the triangle lies
  - Test whether the intersection point is inside the triangle



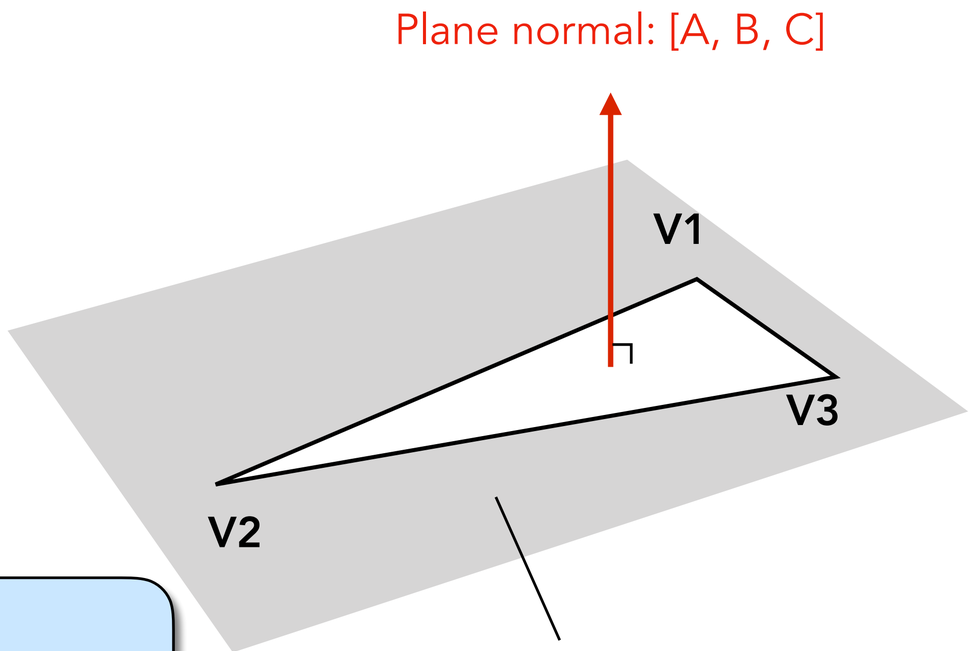
Valette, et al. [TVCG'08]

What Defines a Triangle?

A plane with three vertices.

- The vertices are guaranteed to be co-planar.

The plane that the triangles are in can be expressed as an implicit equation and can be calculated from the vertices.



$$\begin{cases} A(V1_x - V2_x) + B(V1_y - V2_y) + C(V1_z - V2_z) = 0 \\ A(V1_x - V3_x) + B(V1_y - V3_y) + C(V1_z - V3_z) = 0 \\ A \times V1_x + B \times V1_y + C \times V1_z = X \end{cases} \Rightarrow A : B : C : X$$

Plane: **A**x + **B**y + **C**z = **X**

Ray-Triangle Intersection

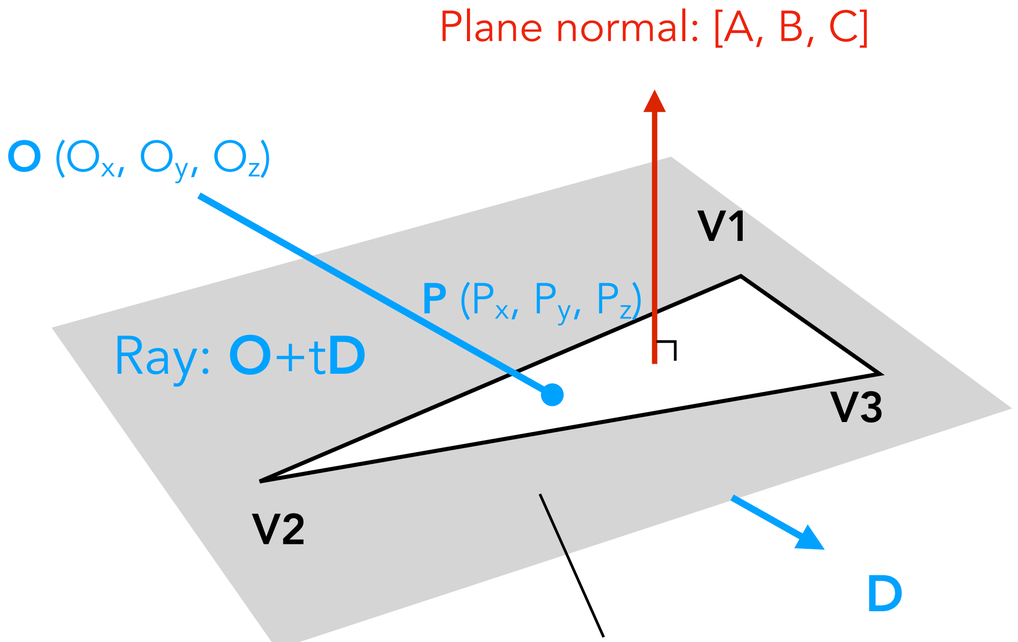
$$P_x = O_x + D_x \times t$$

$$P_y = O_y + D_y \times t$$

$$P_z = O_z + D_z \times t$$

$$A \times P_x + B \times P_y + C \times P_z = 1$$

$$t = \frac{1 - (A \times O_x + B \times O_y + C \times O_z)}{A \times D_x + B \times D_y + C \times D_z}$$

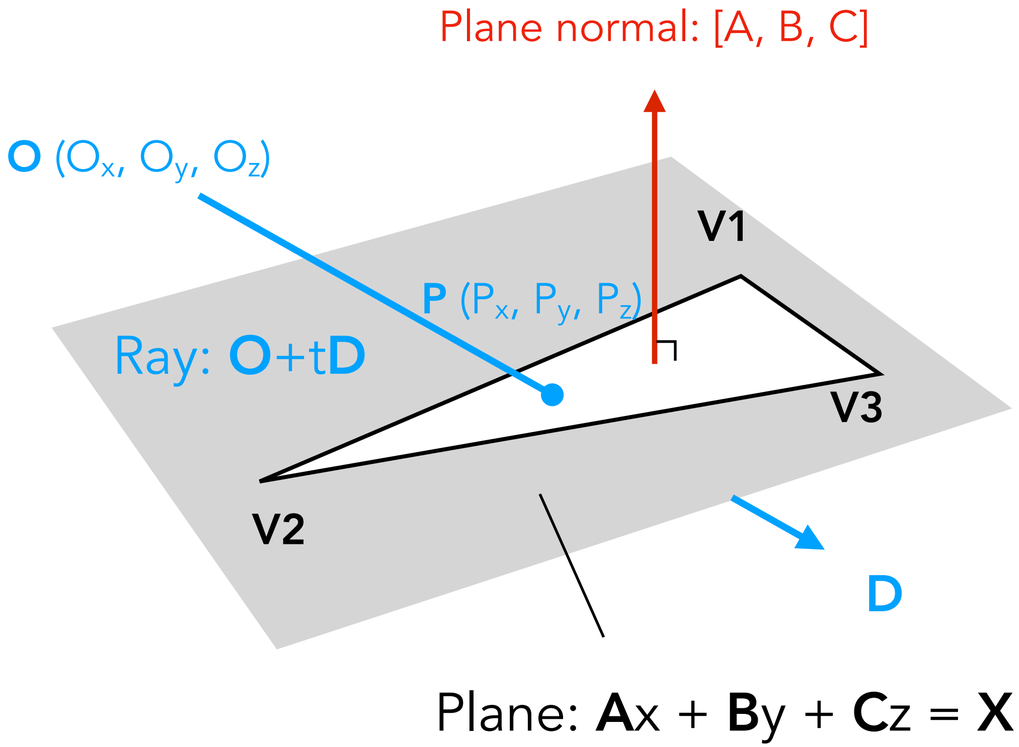


Plane: **A**x + **B**y + **C**z = **X**

Three Caveats

$$t = \frac{1 - (A \times O_x + B \times O_y + C \times O_z)}{A \times D_x + B \times D_y + C \times D_z}$$

1. The denominator is 0 if the normal is perpendicular to the direction of the ray (i.e., ray is parallel to the plane).
   - Need a special test for whether the ray is parallel with the plane (before division).

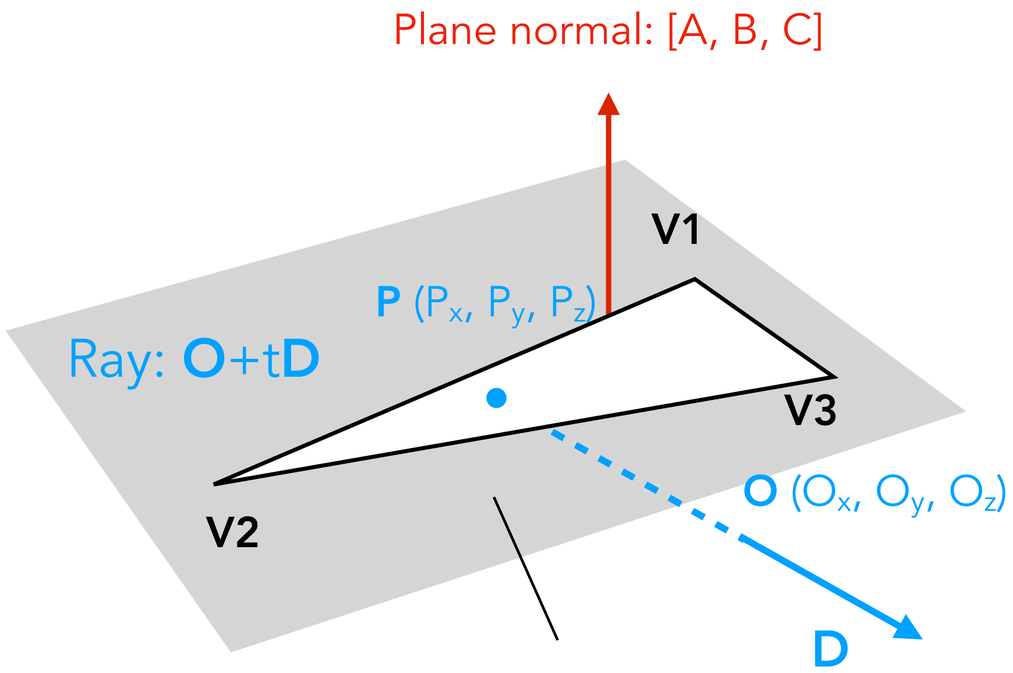


Plane: **A**x + **B**y + **C**z = **X**

Three Caveats

2. A ray doesn't intersect with a plane if the triangle plane is behind the origin of the ray

- i.e., t is negative.



Plane: **A**x + **B**y + **C**z = **X**

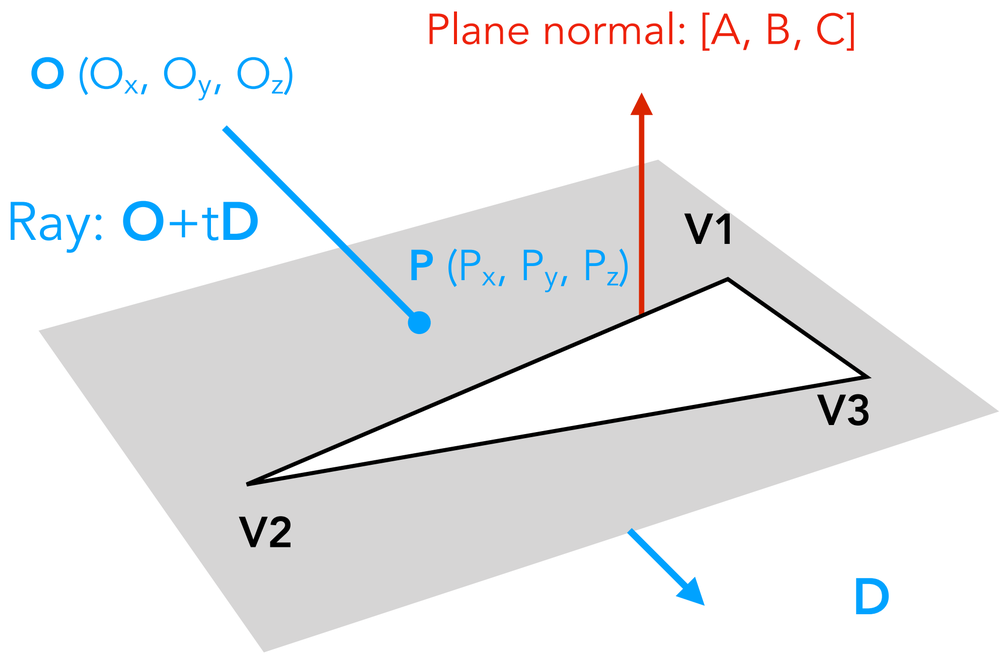
Three Caveats

2. A ray doesn't intersect with a plane if the triangle plane is behind the origin of the ray

- i.e., t is negative.

3. Even if a real intersection point is found, the intersection point could be outside the triangle.

- Use barycentric coordinates to test whether a point is outside of a triangle.



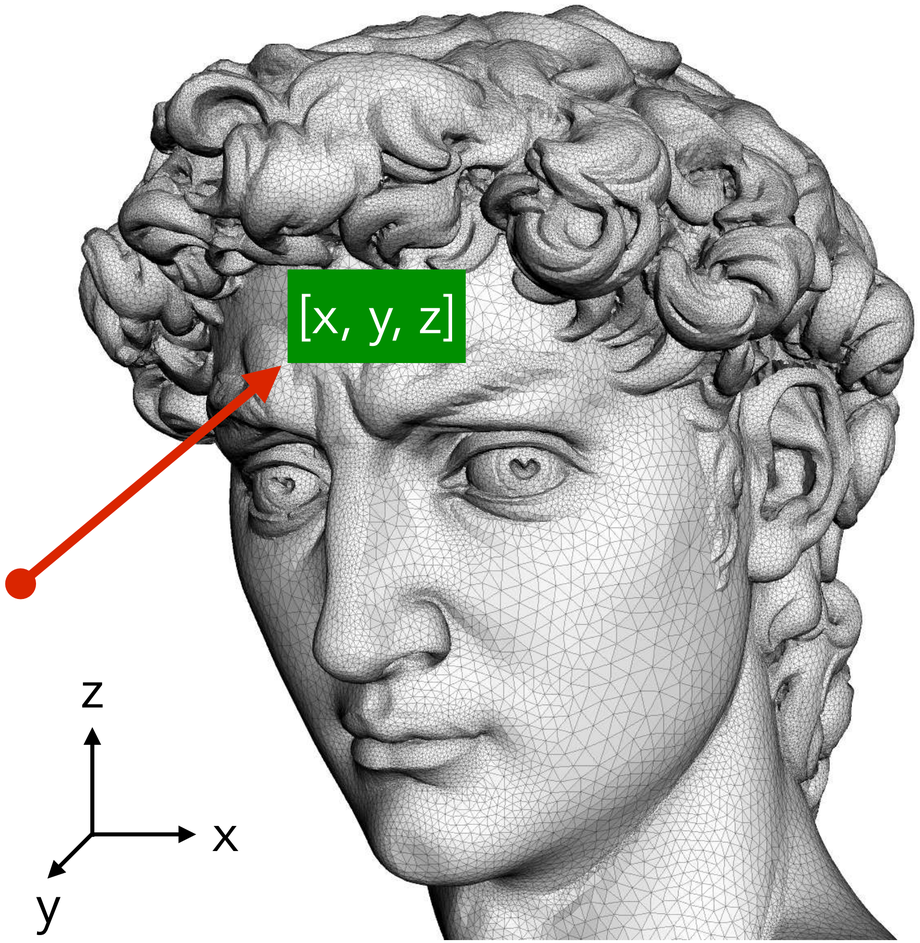
Brute-Force Approach is Extremely Inefficient

Brute-force approach:

- iterate all triangles
- test intersection for each triangle
- return the closest hit, if any

Time complexity:

- O(# of rays x # of triangles)



Valette, et al. [TVCG'08]

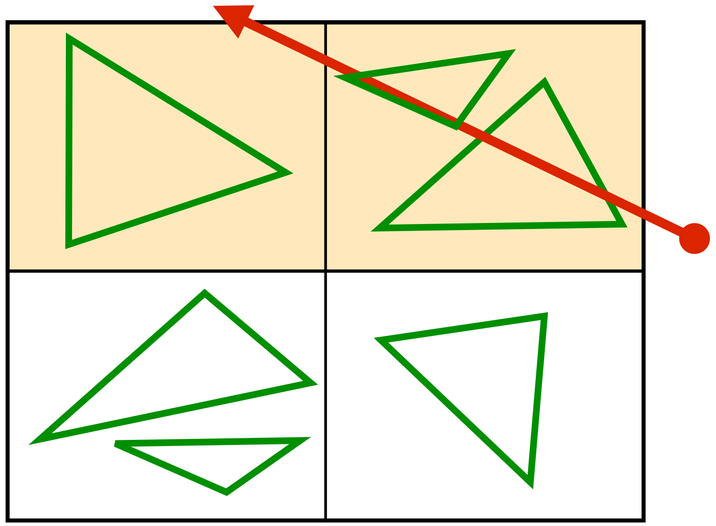
Accelerating Ray-Scene Intersection

Speeding Up Ray-Triangle Intersection Test

Prune the search space.

Only search part of the scene that does intersect the ray.

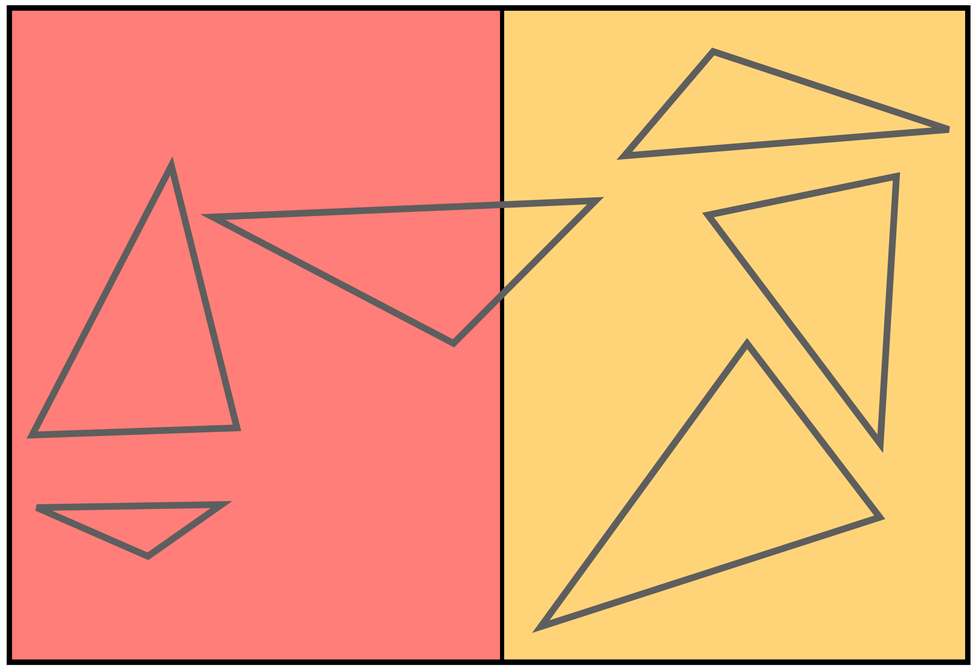
Key: how to partition the space?



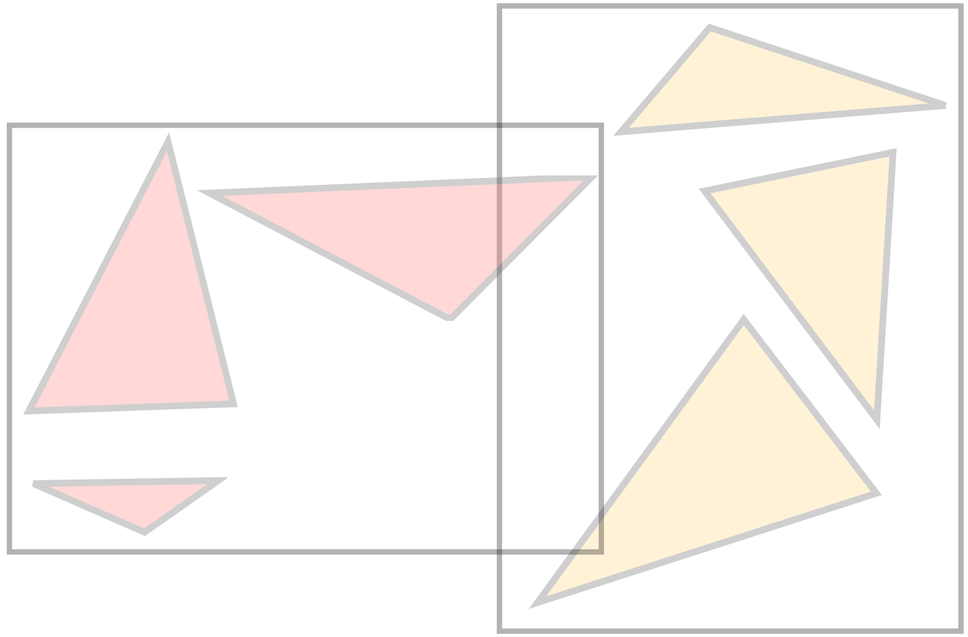
```
intersect(space, ray) {
  if ray doesn't intersect space boundary:
    return
  else
    foreach subspace in space
      if (subspace != empty)
        intersect(subspace, ray)
}
```

Space vs. Object Partitioning

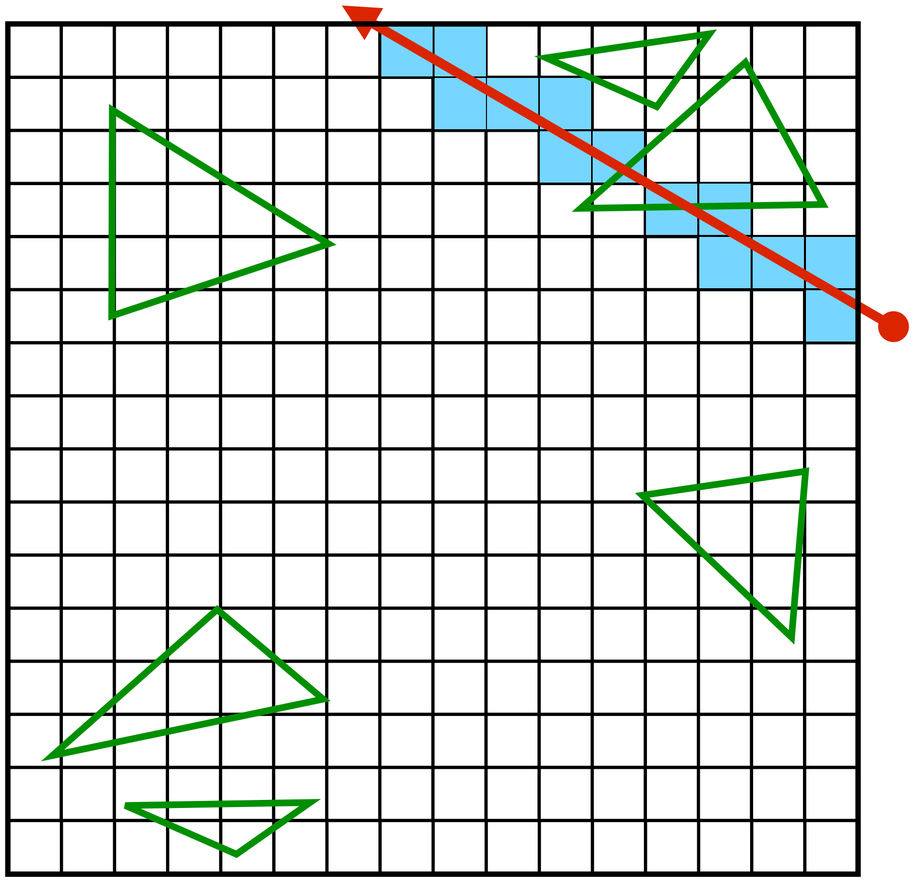
Space partitioning: One object could be in different partitions



Object partitioning: different partitions could overlap in space



Uniform Grid



Find the bounding box of the scene

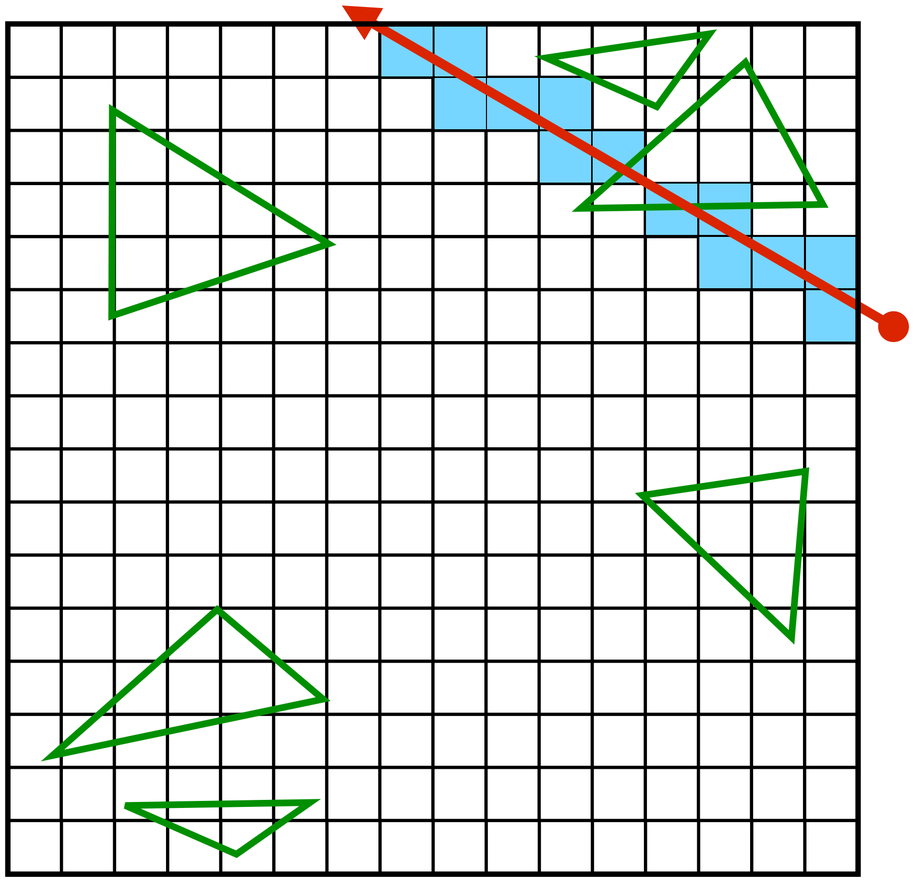
Generate a uniform grid

Find intersecting cells

For each intersecting cell:

- Iterate over all the containing triangles
- Get the closet intersection within the cell
- Update the global closet intersection

Grid Resolution



Too few cells:

- Little speedup

Too many cells:

- Many empty cells to check and to store

A useful heuristics:

- The number of cells should be proportional to the number of triangles
- #cell in each dimension = n^{1/3}

When Uniform Grid Works

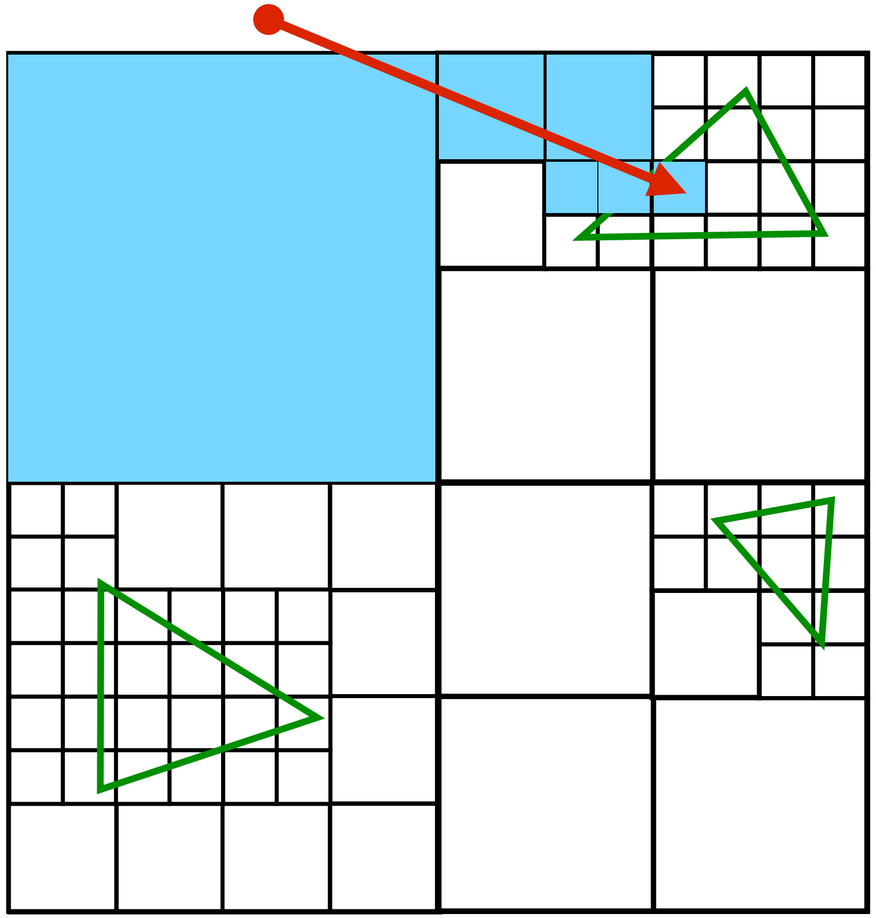
Small objects roughly uniformly distributed in space

When Uniform Grid is Inefficient

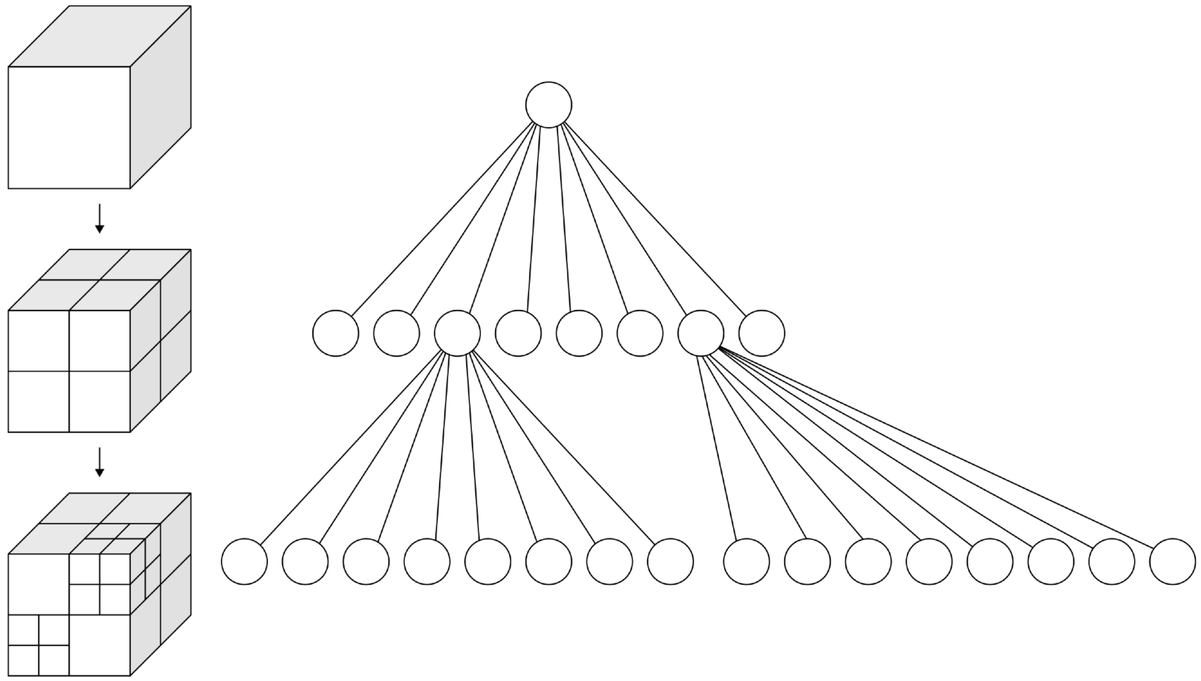
Objects sparsely distributed in space (“teapot in a stadium”)

Non-Uniform (Adaptive) Grid

Quadtree (2D)

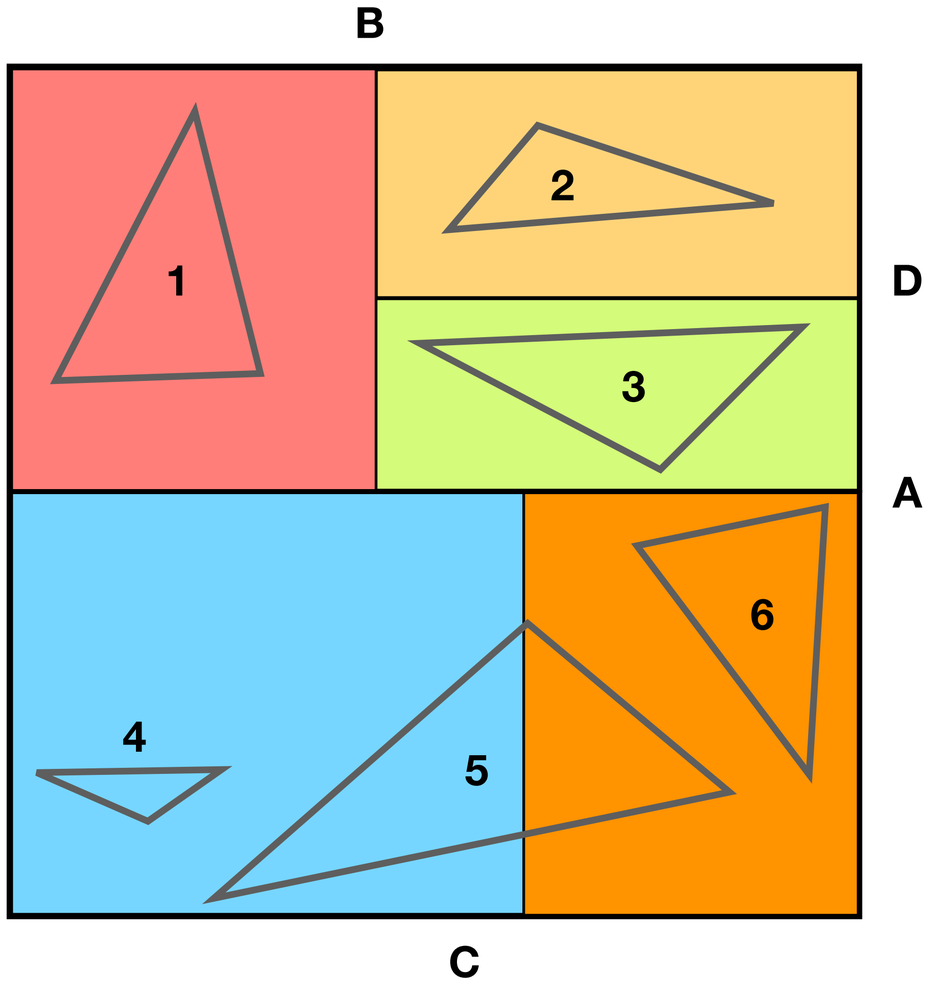


Octree (3D)



https://en.wikipedia.org/wiki/Octree

Building K-D Tree



Recursively using axis-aligned planes to split the space

Stop when certain terminating conditions are met

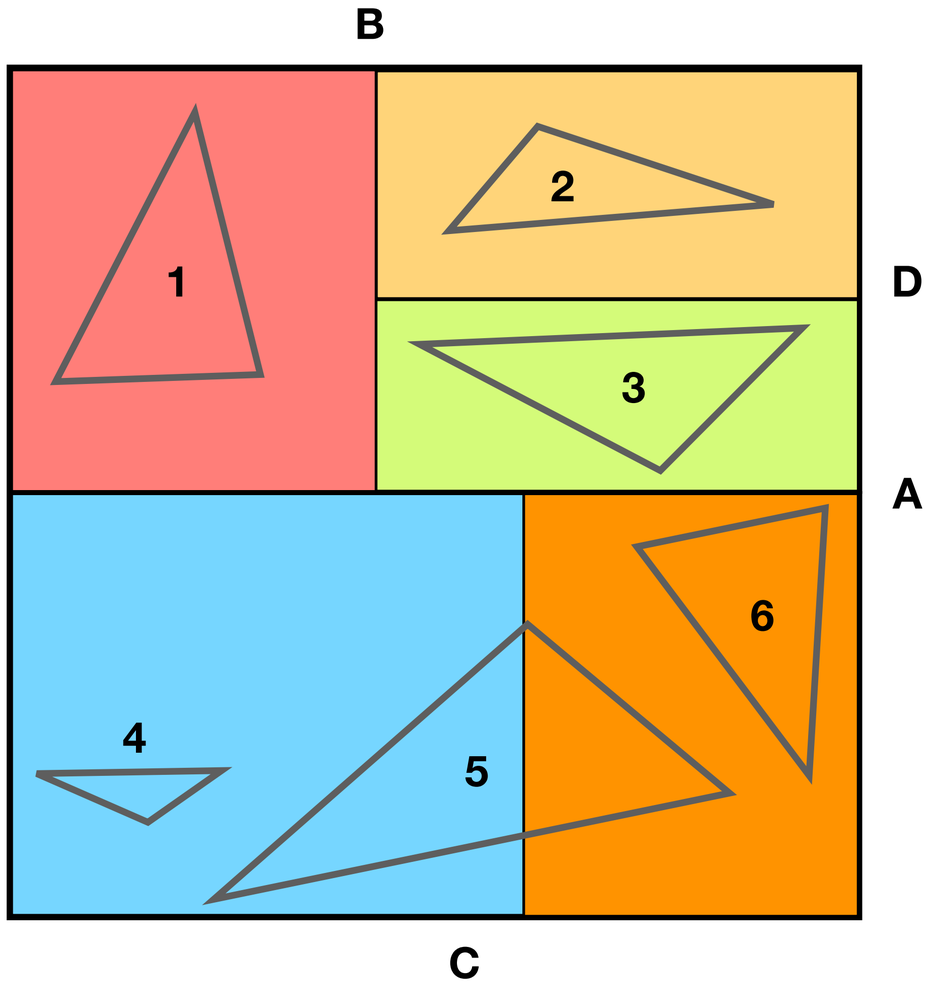
- # of objects in a cell < threshold
- Max tree depth met

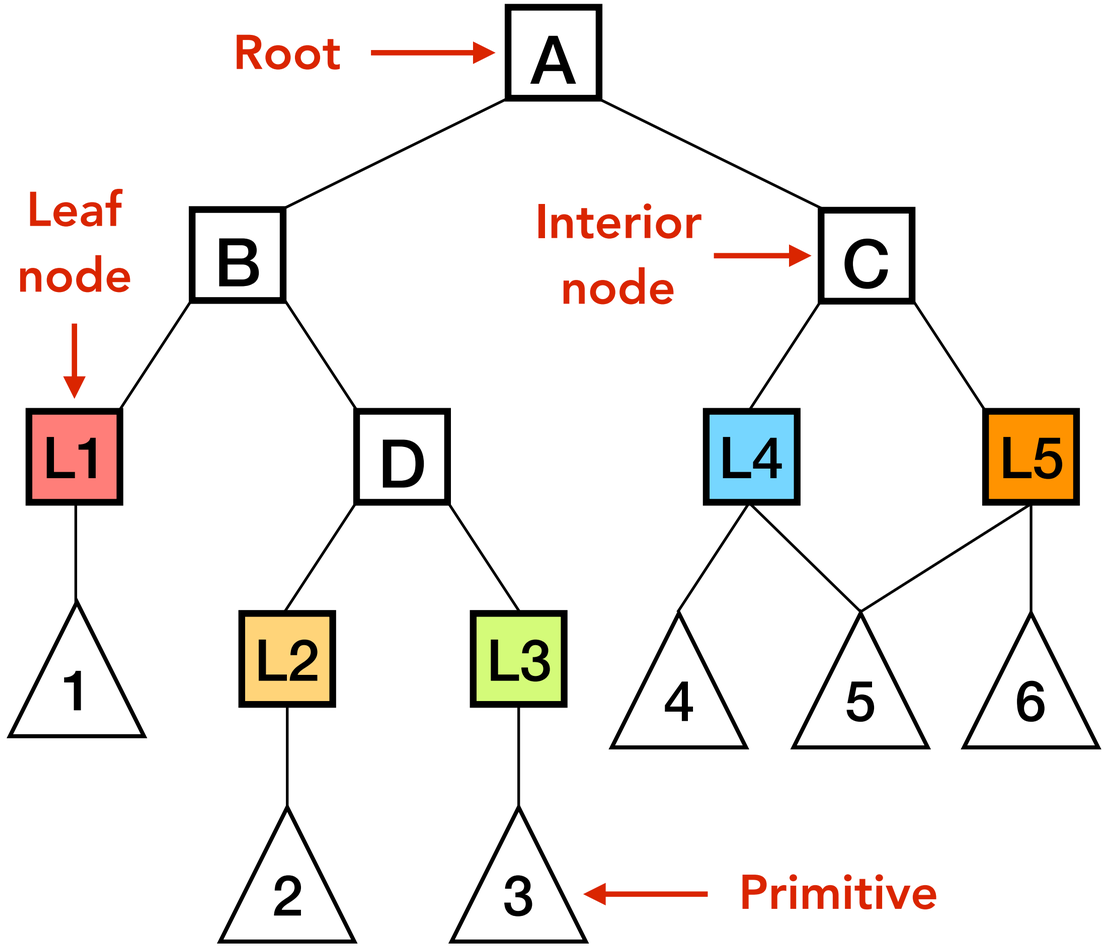
Organize the splits using a tree

Find the closest hit by traversing the tree

http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m13/kd.html

K-D Tree

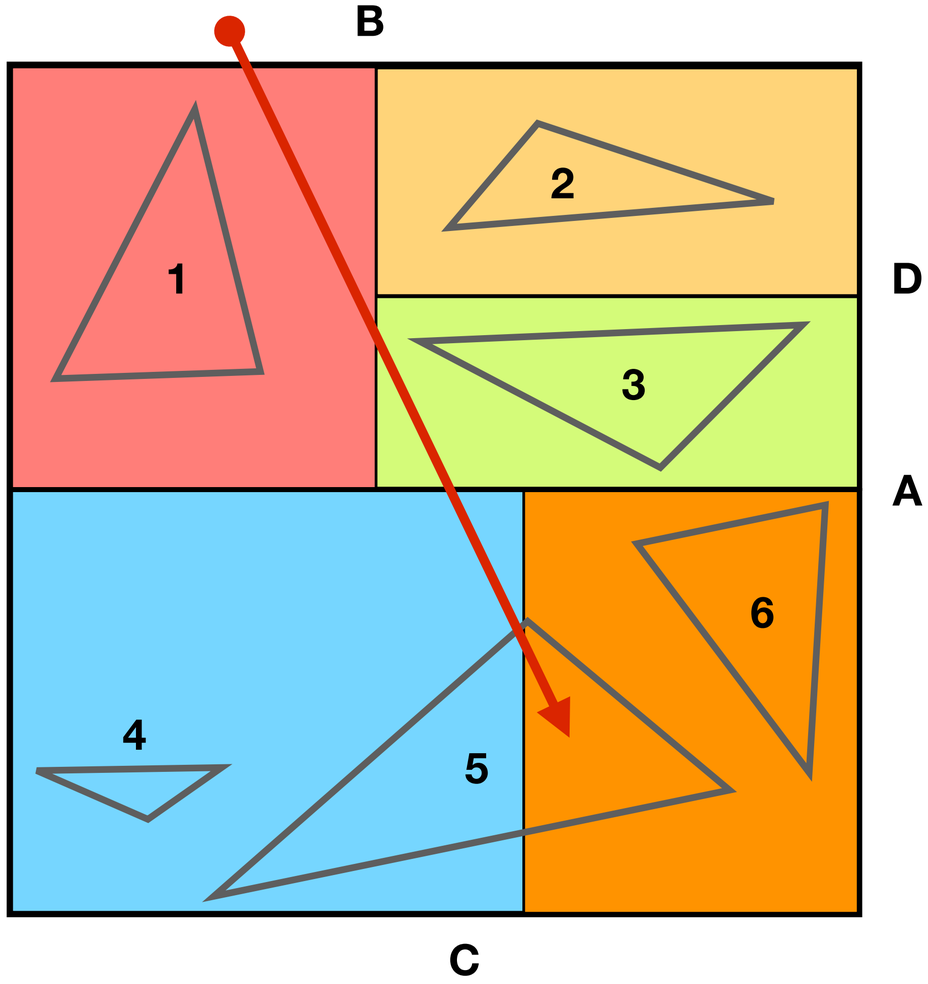

B
1
2
3
D
A
6
4
5
C

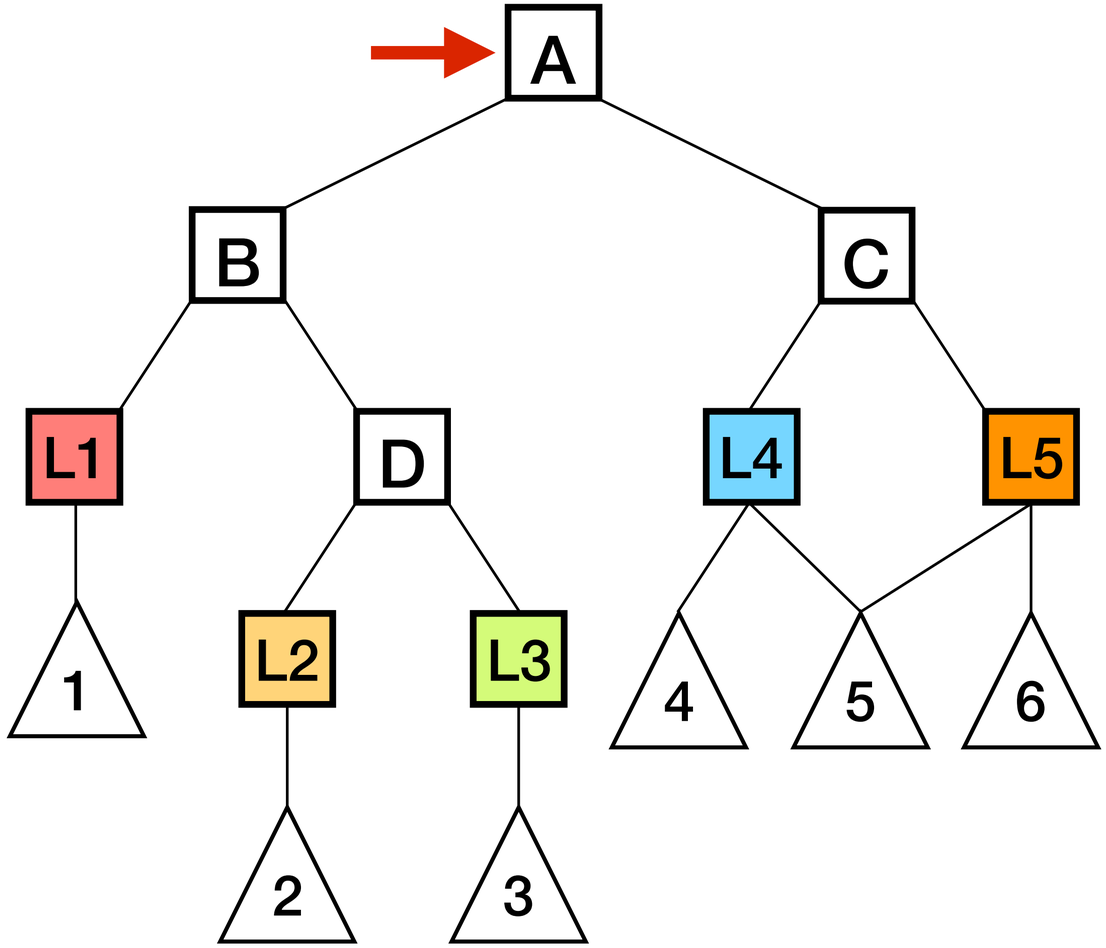

Root
A
Leaf
node
B
Interior
node
C
L1
D
L4
L5
1
L2
L3
4
5
6
2
3
Primitive

Traversing K-D Tree

Current [A]

Stack

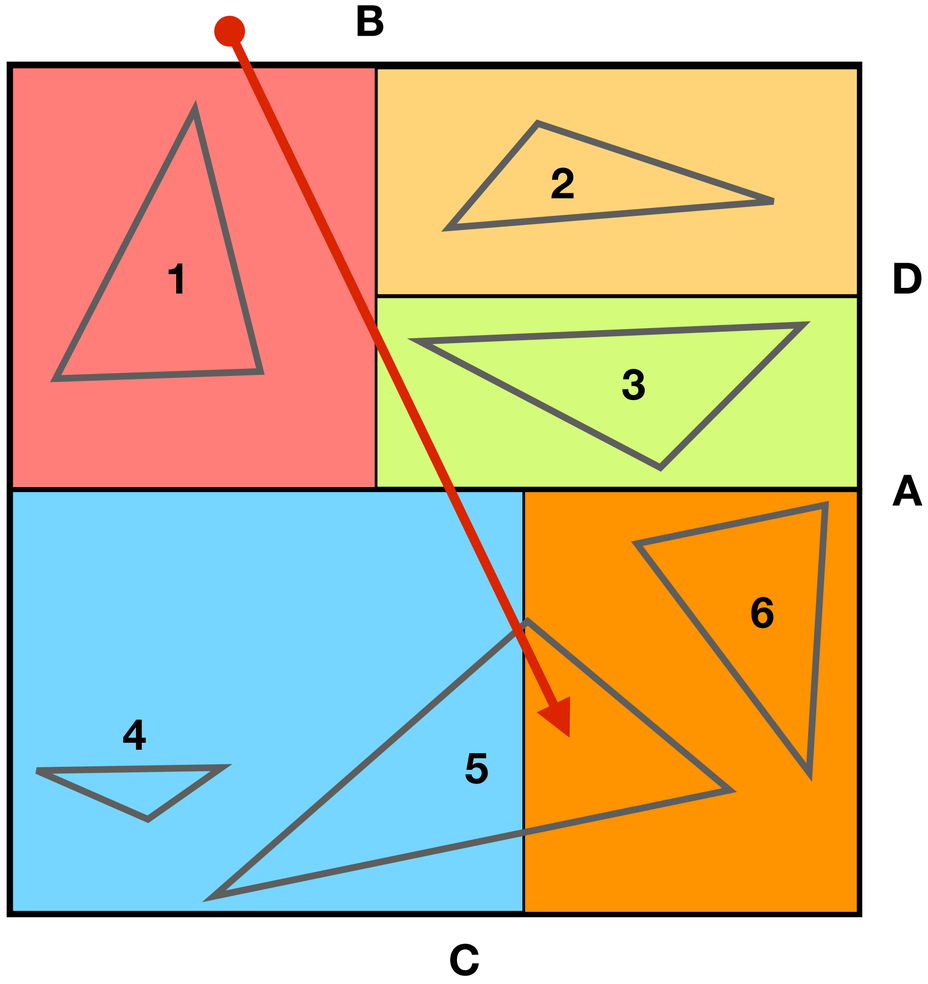


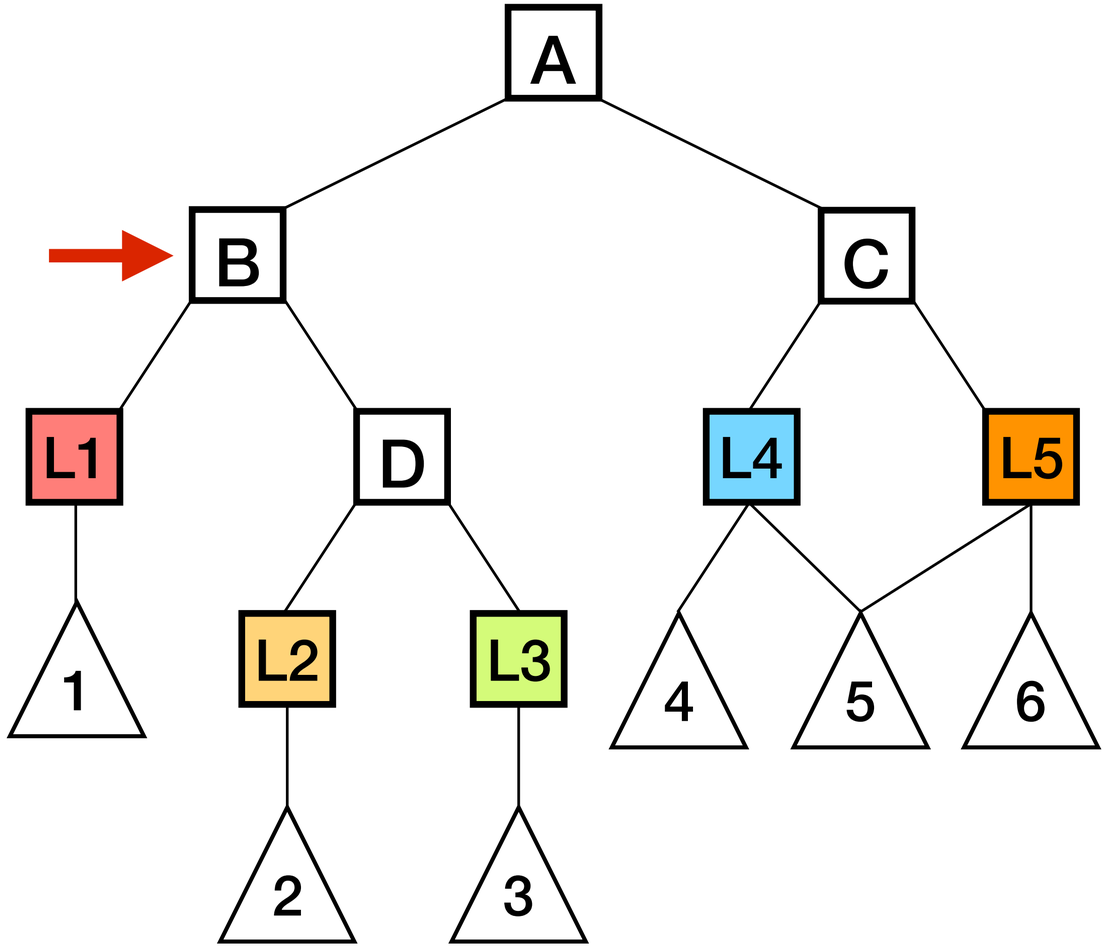


Traversing K-D Tree

Current B

Stack C

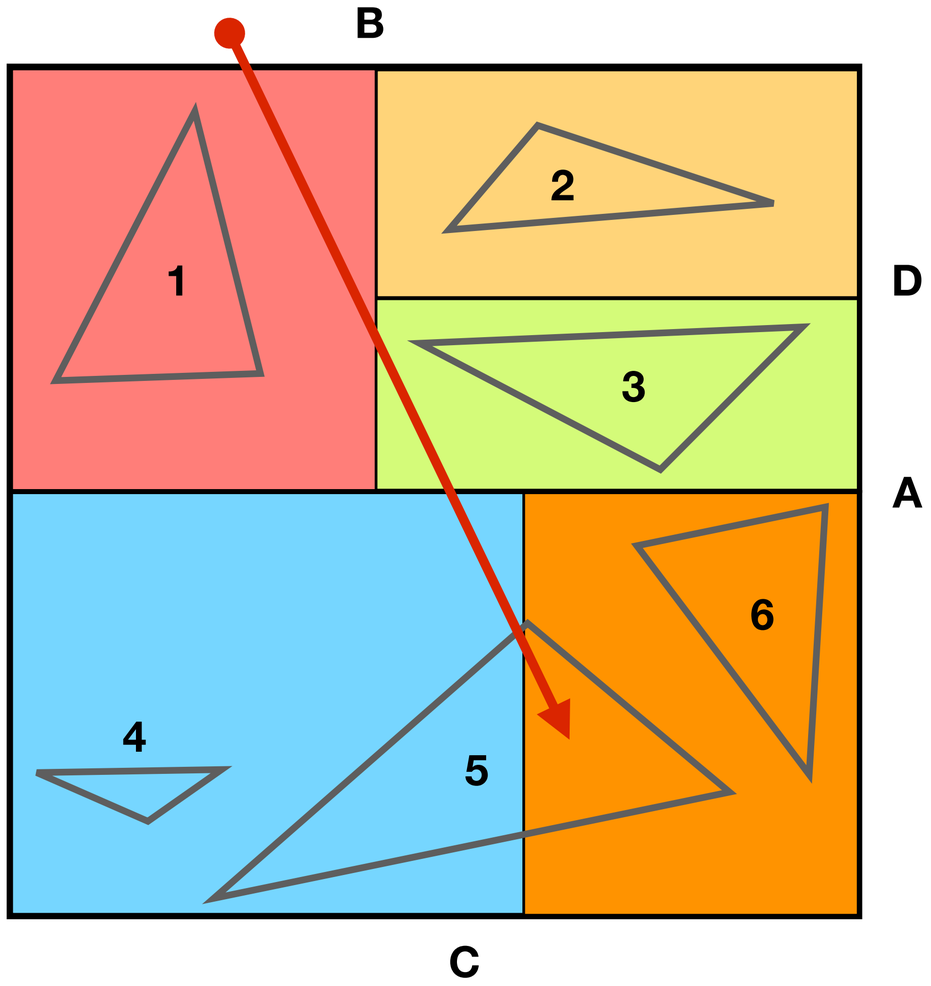


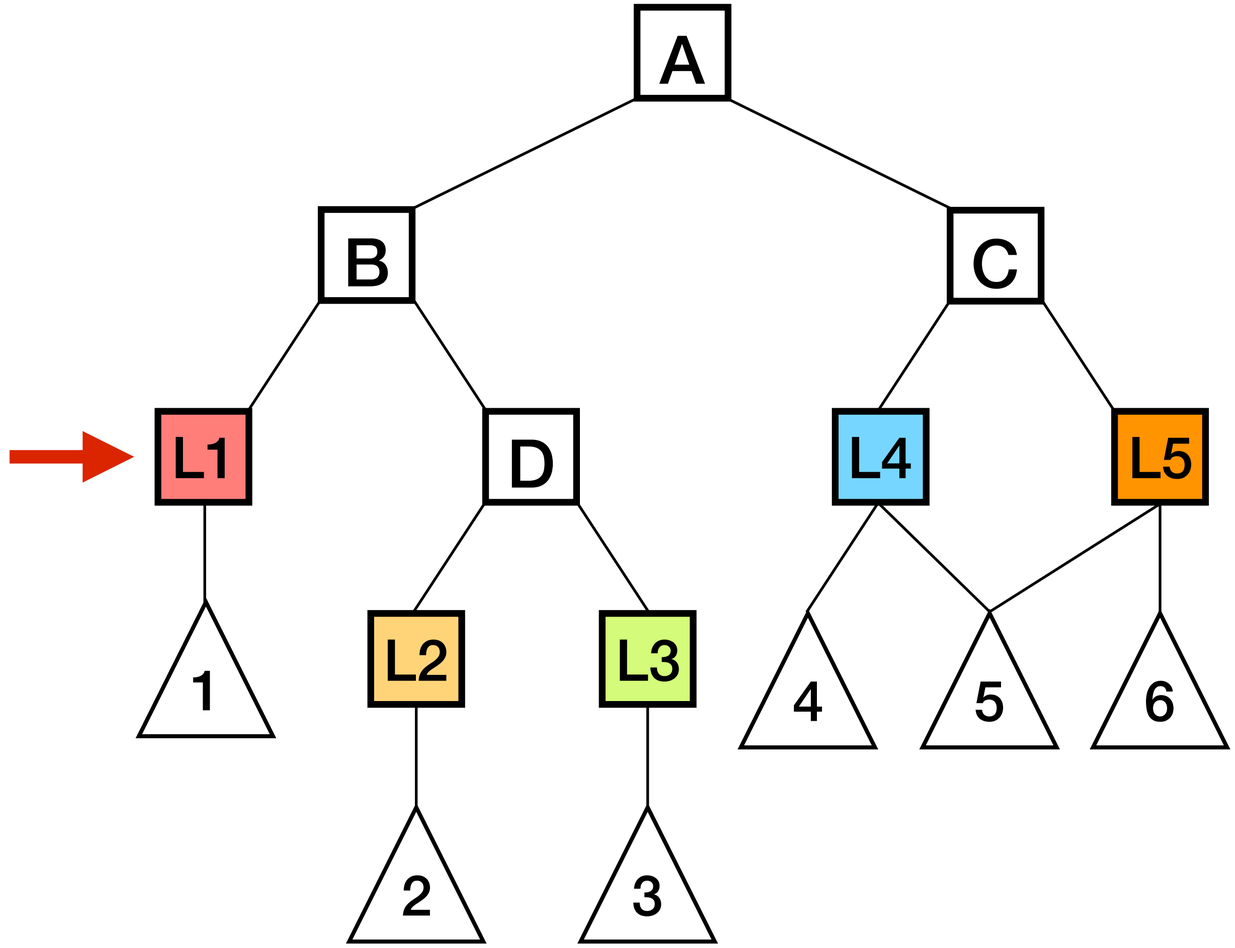


Traversing K-D Tree

Current L1

Stack C D

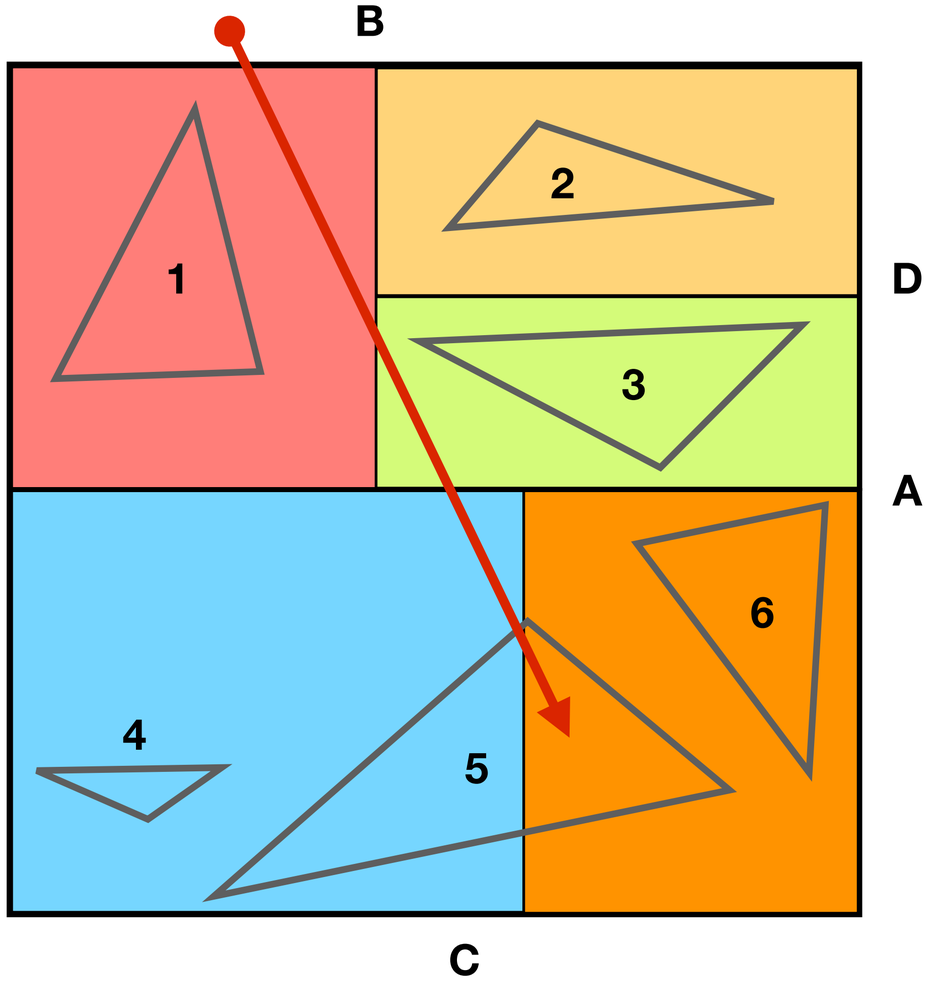


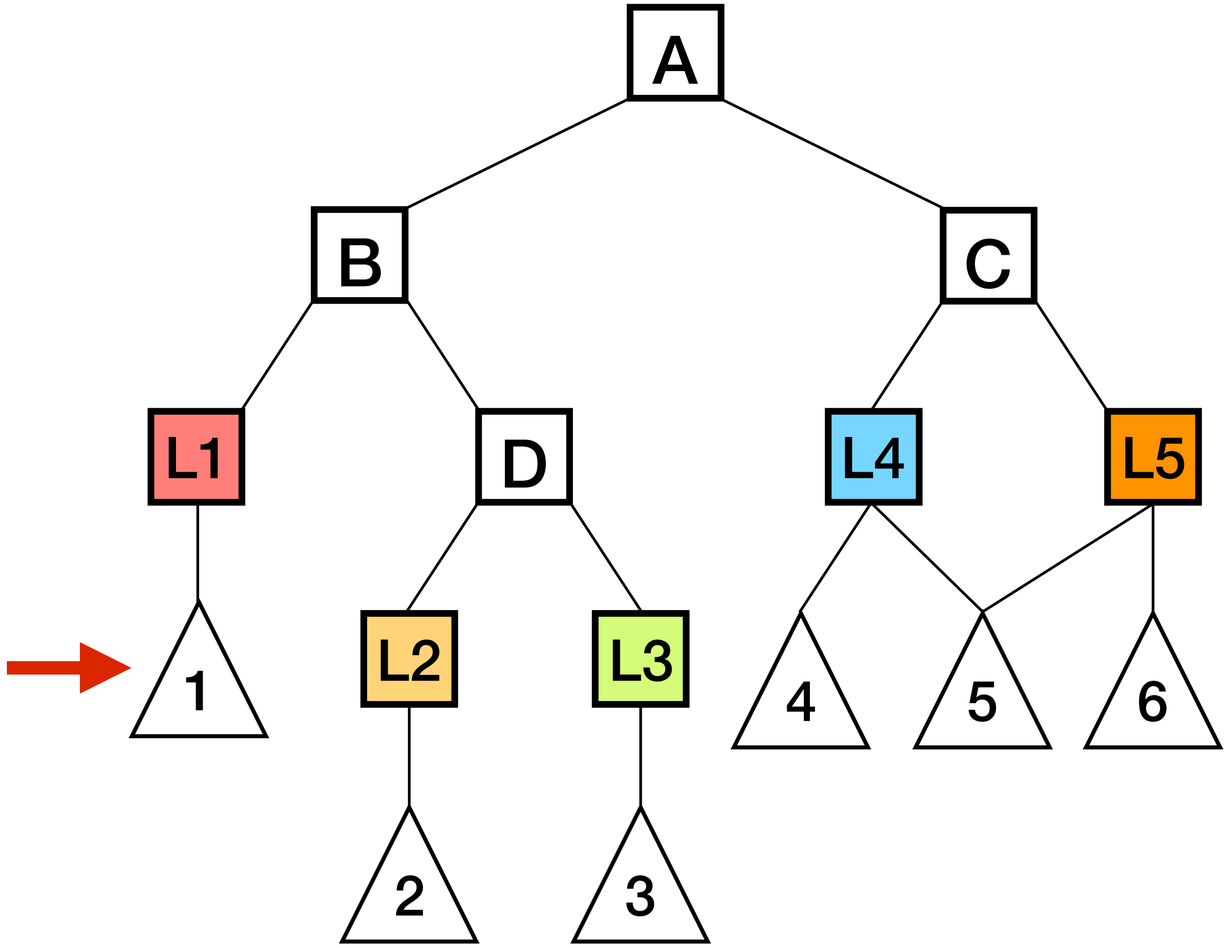


Traversing K-D Tree

Current 

Stack C D

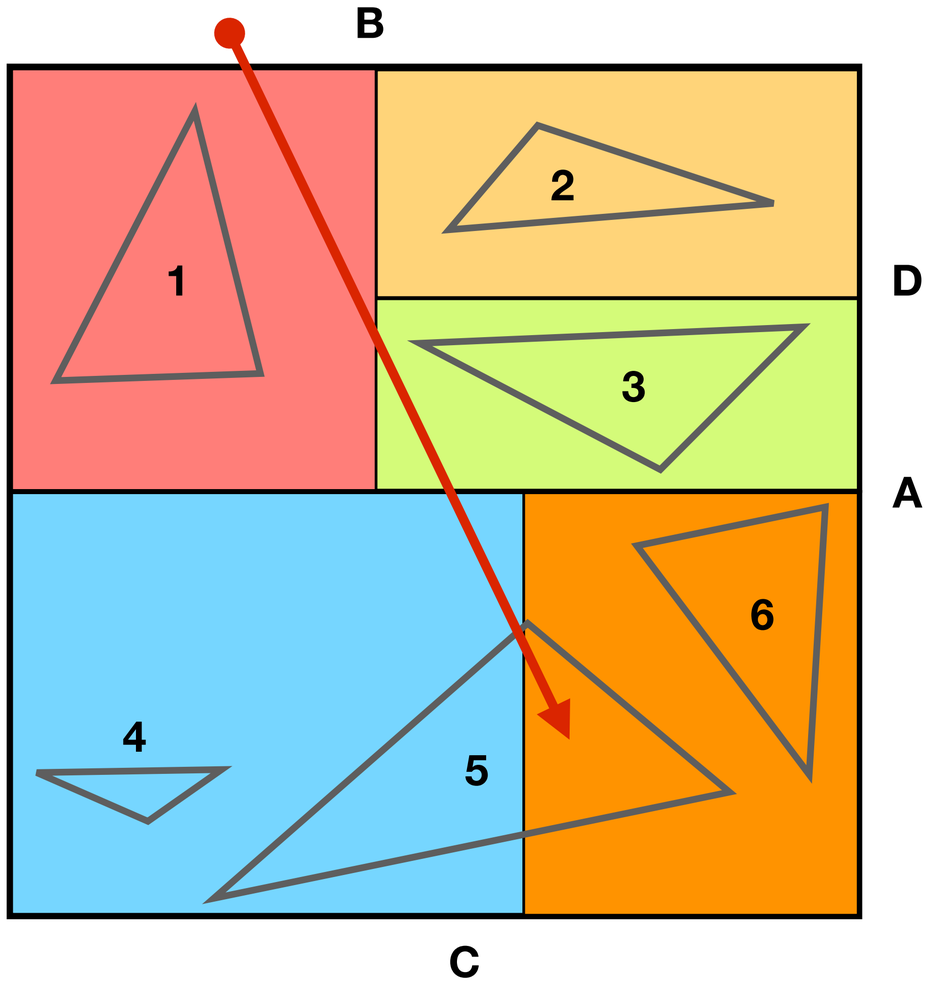


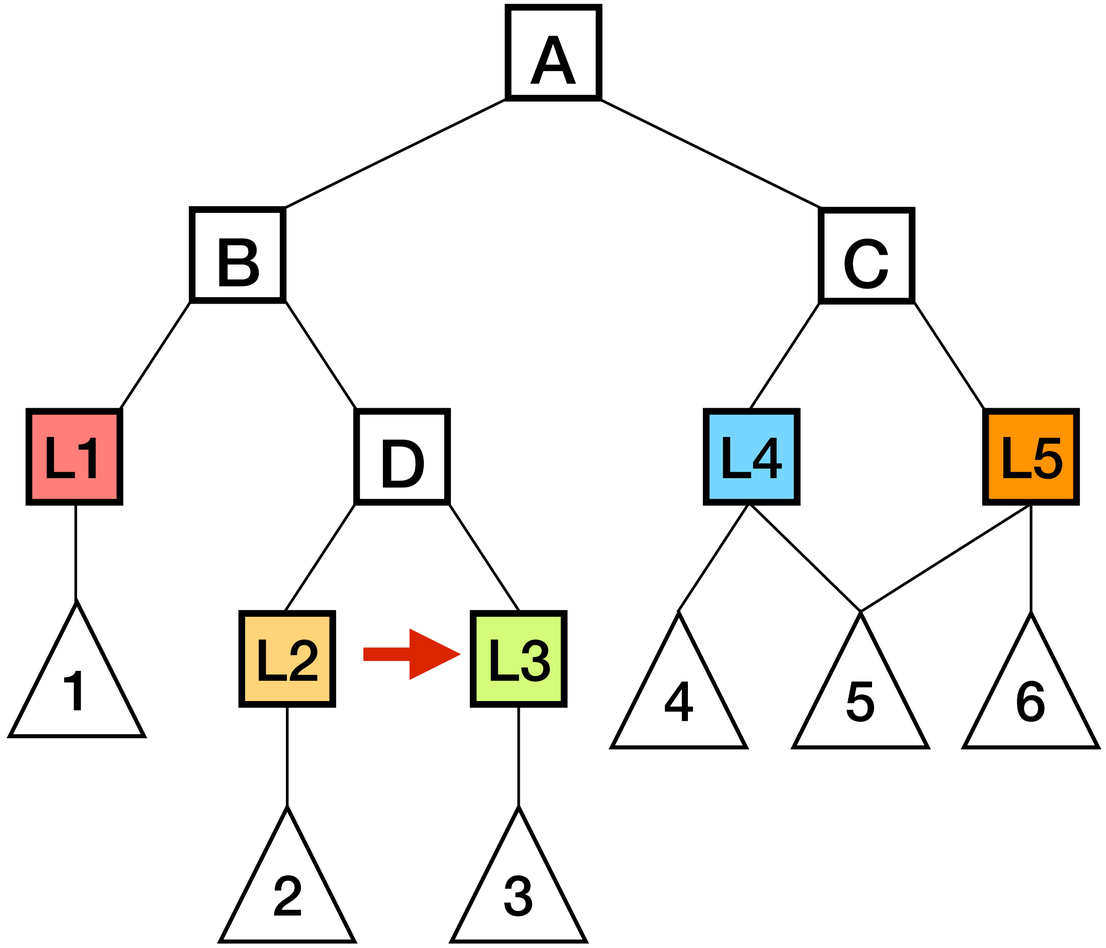


Traversing K-D Tree

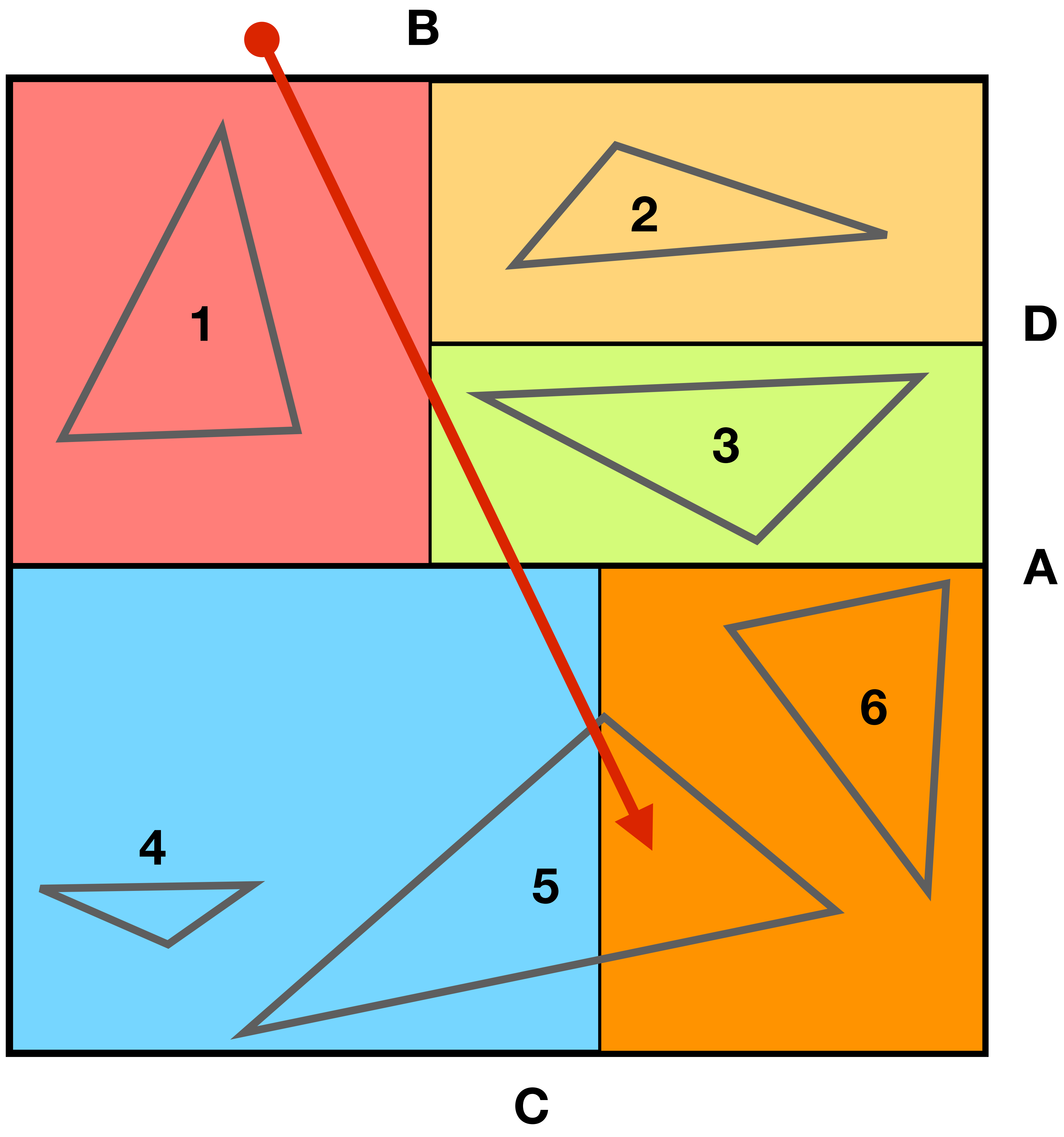
Current L3

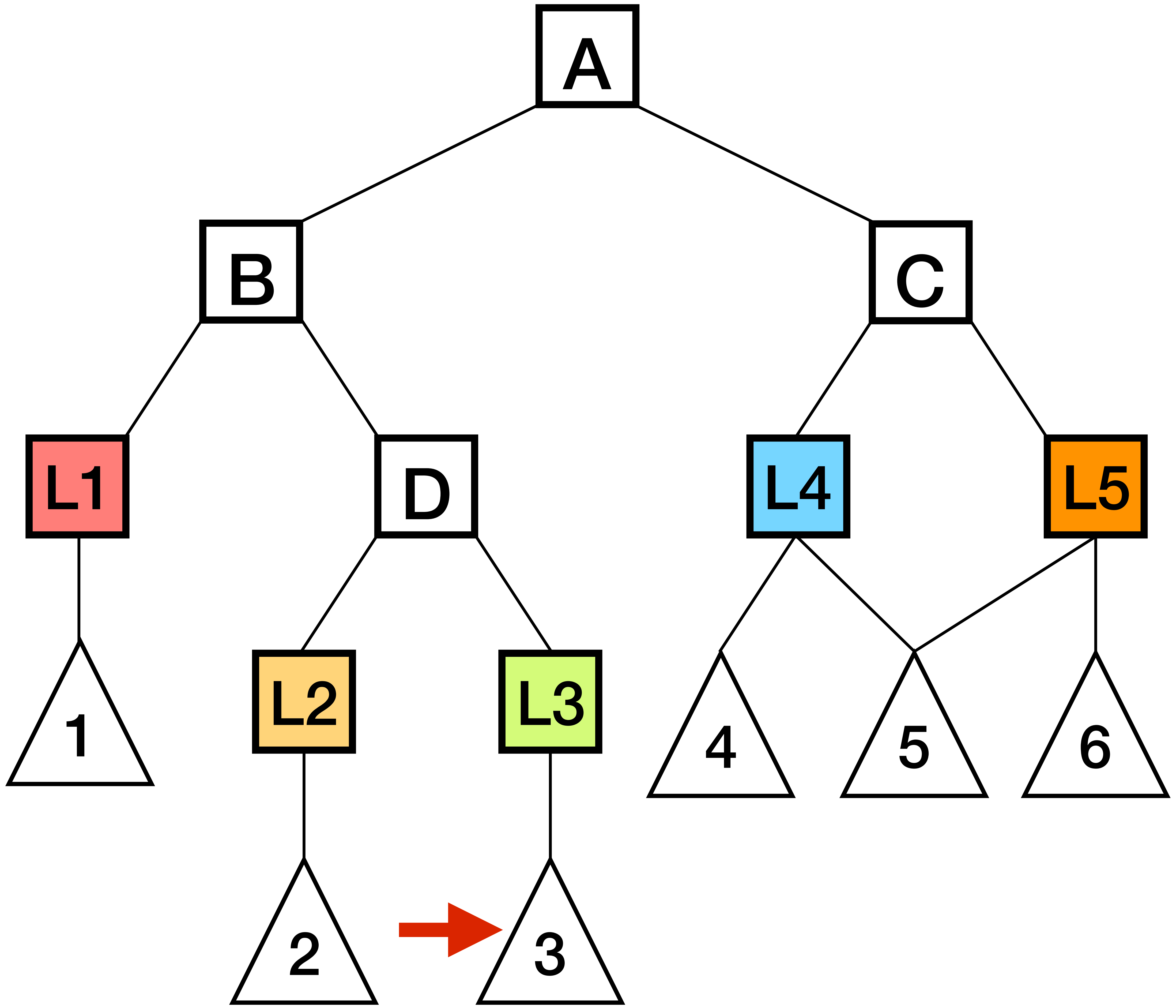
Stack C





Traversing K-D Tree



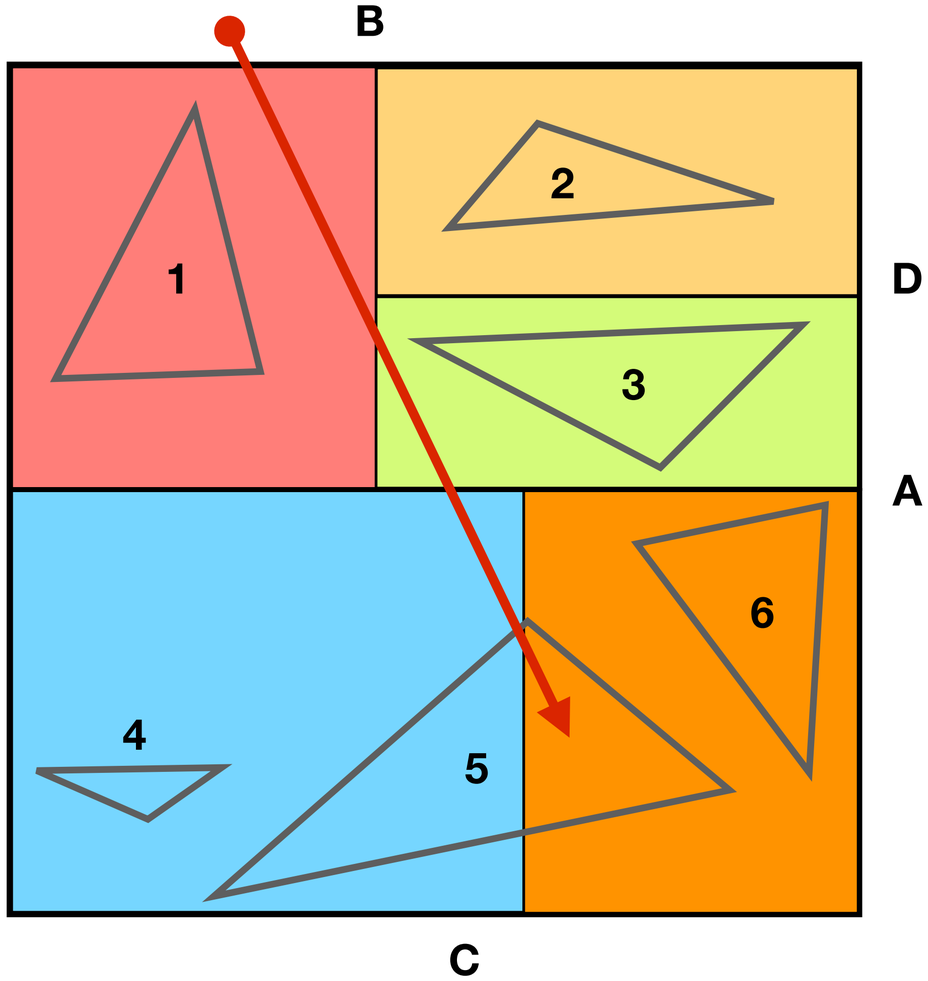


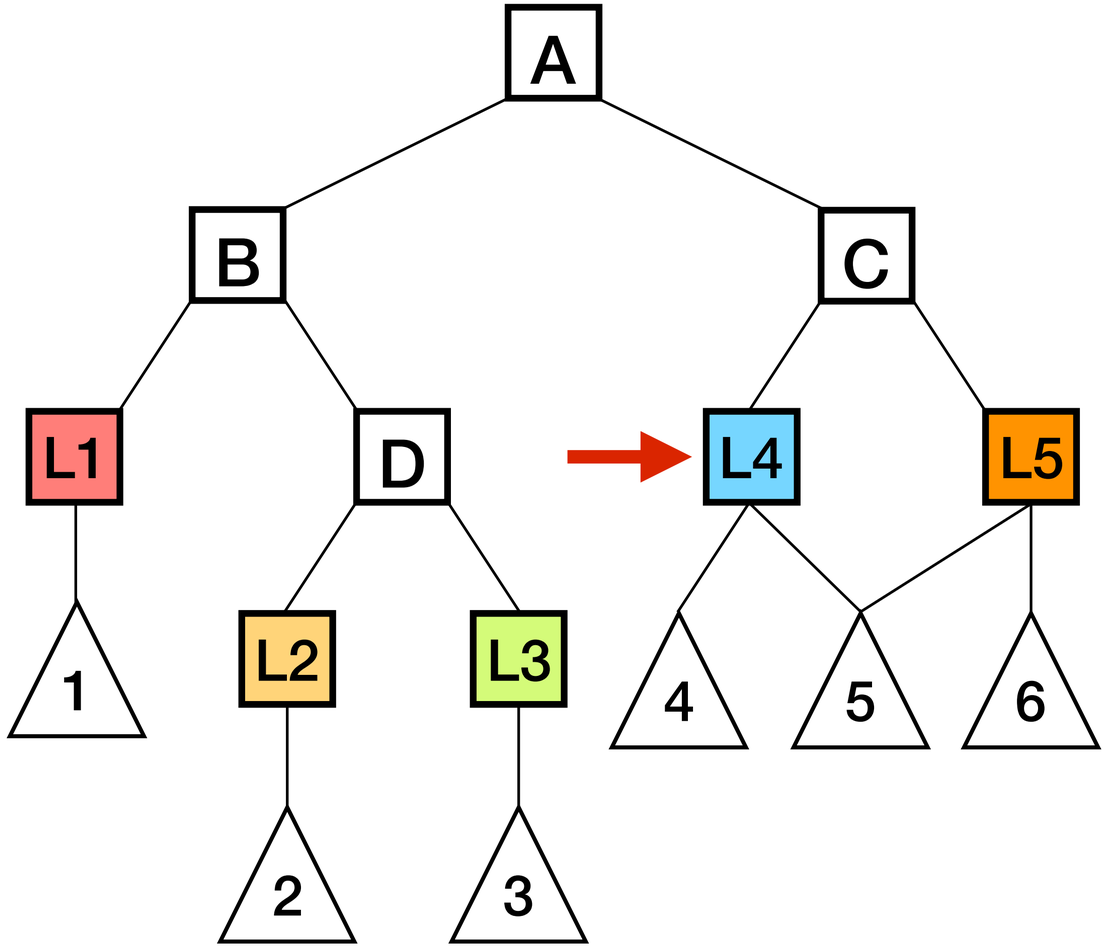
http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m13/kd.html

Traversing K-D Tree

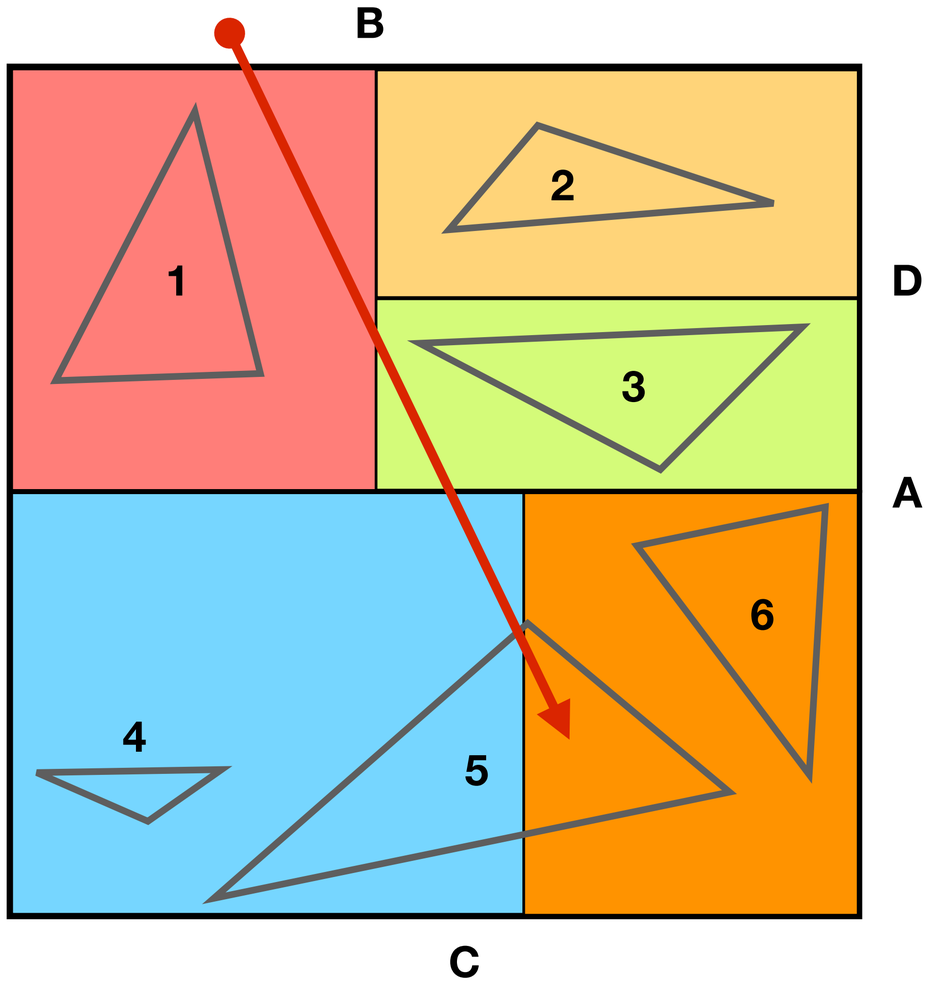
Current L4

Stack L5

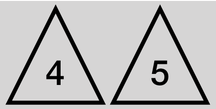




Traversing K-D Tree

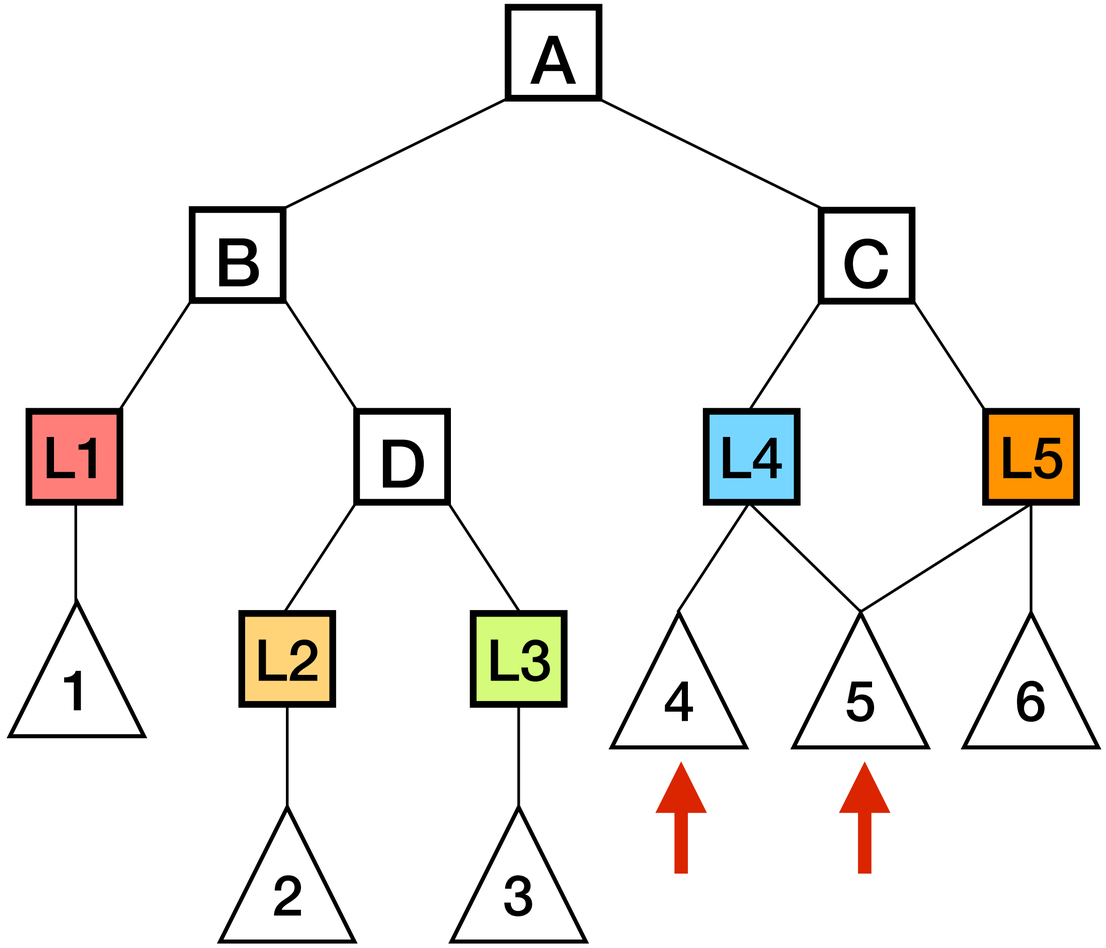


Current



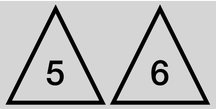
Stack

Result



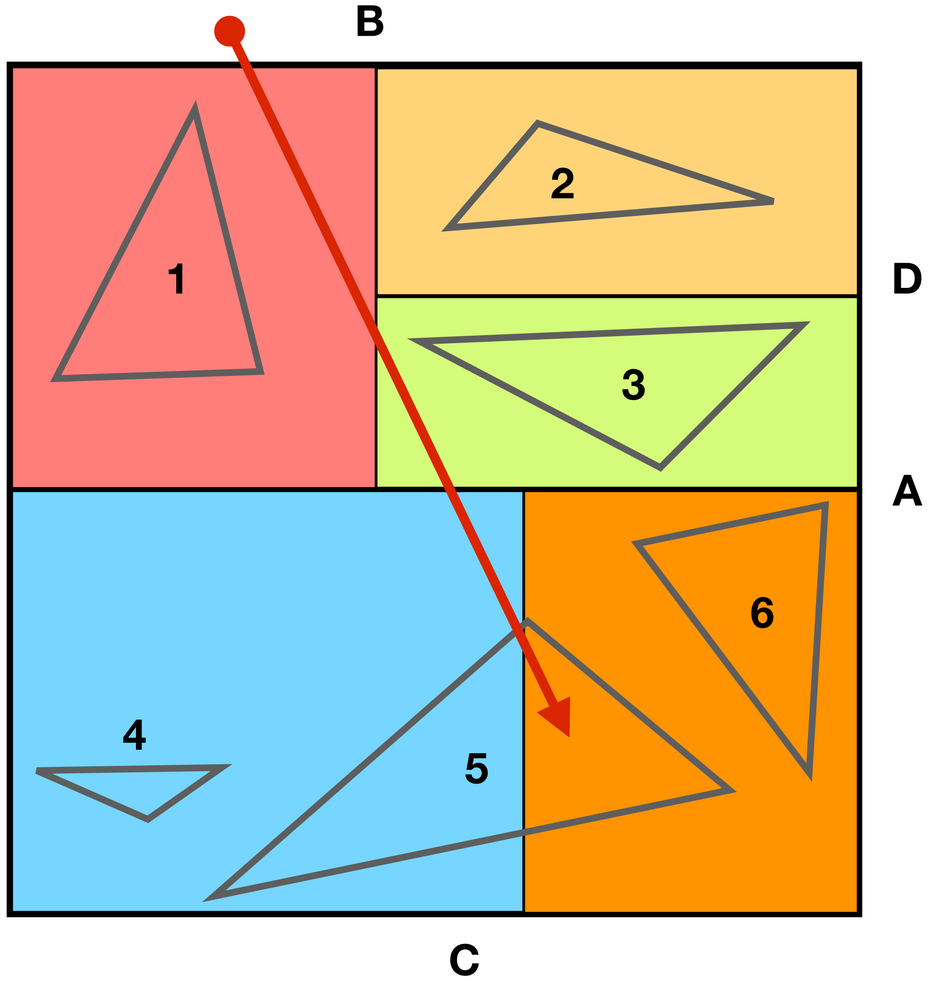
Traversing K-D Tree

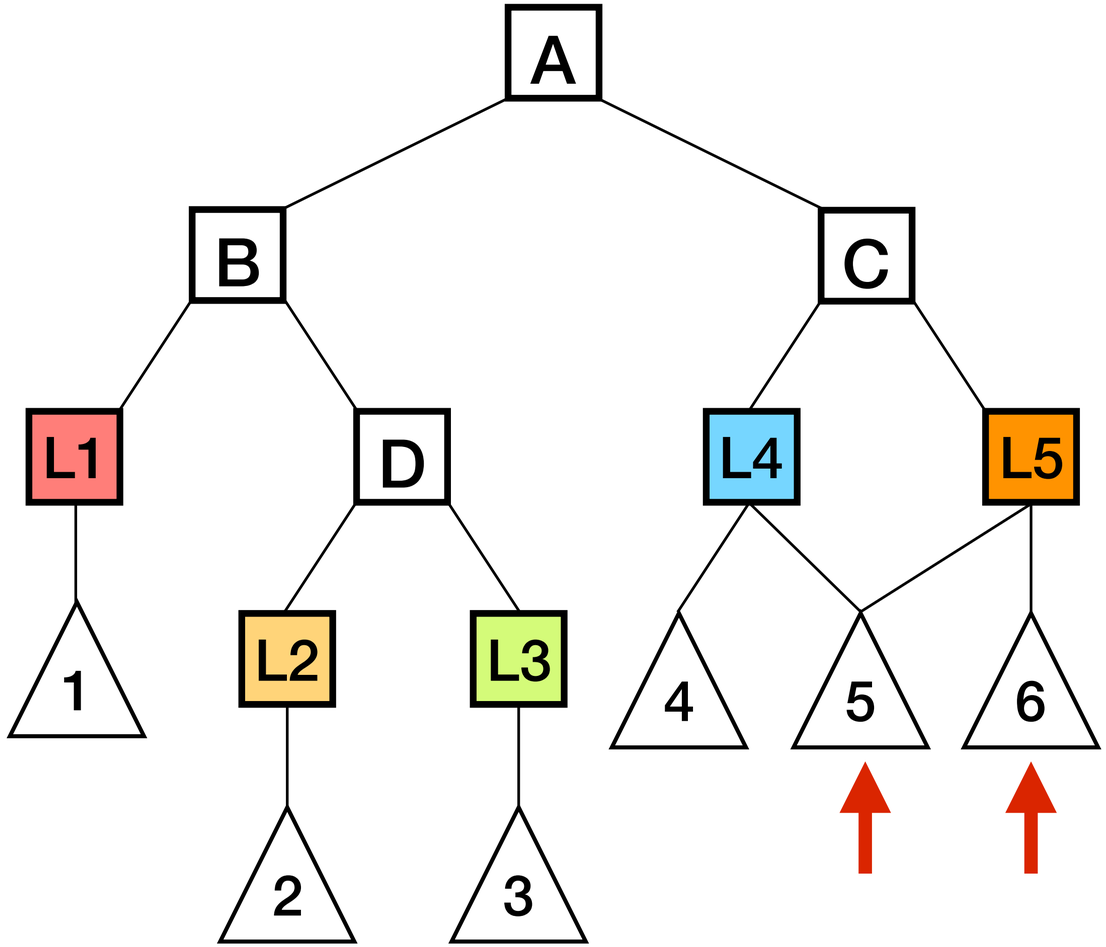
Current



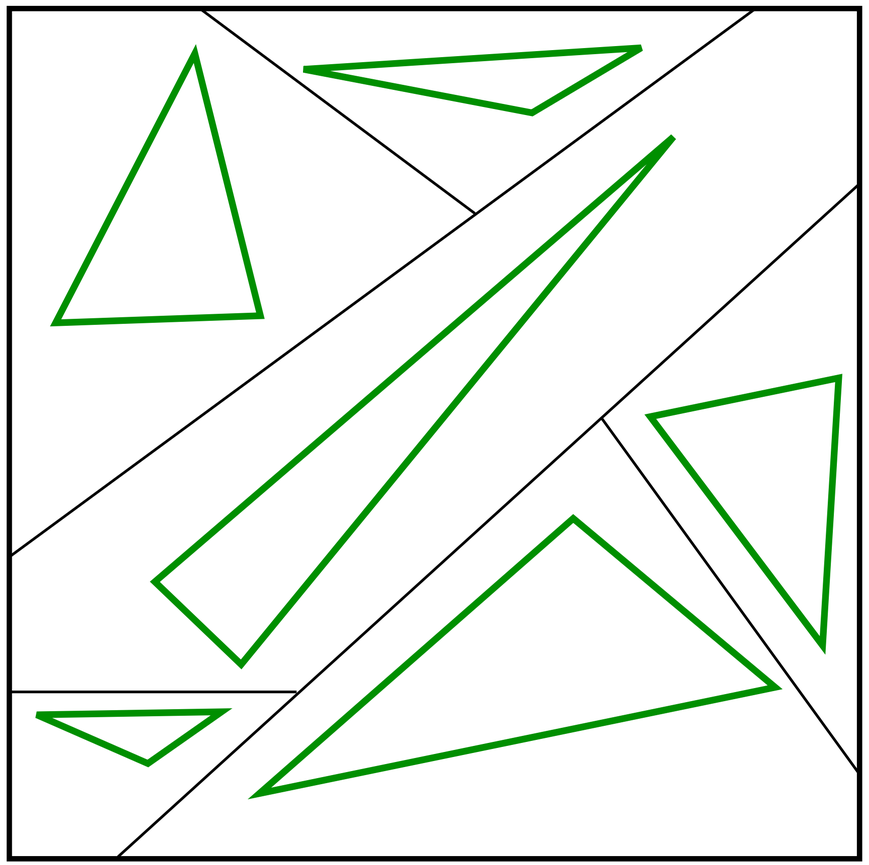
Stack

Result





Binary Space Partitioning Tree



K-D tree is a special case of binary space partitioning (BSP) tree, which recursively split the space with planes (3D) or lines (2D)

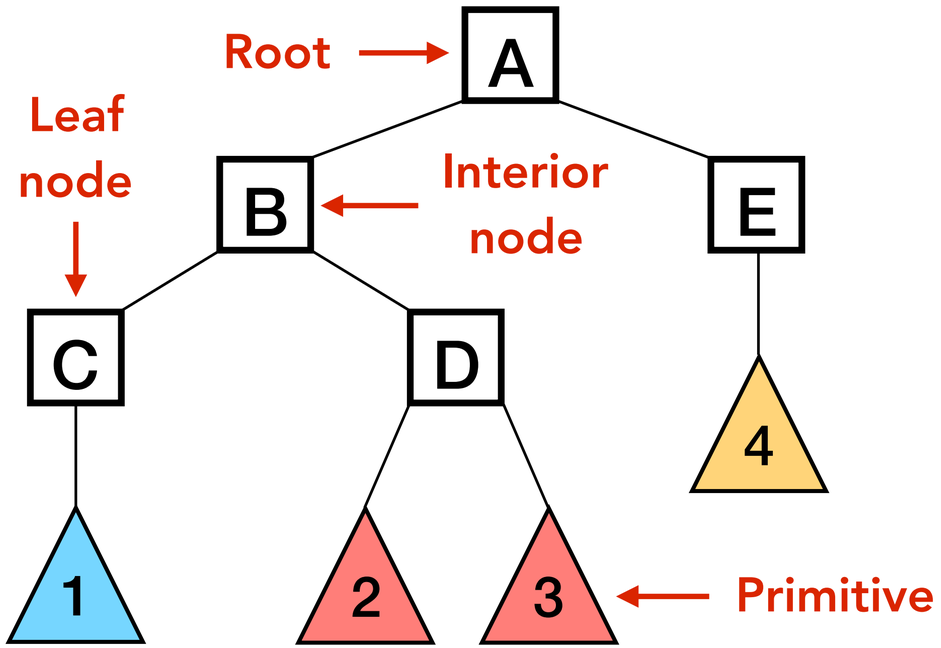
- Arbitrary split planes here

Useful when objects are large and non-axis-aligned, in which case K-D tree will split objects into different partitions

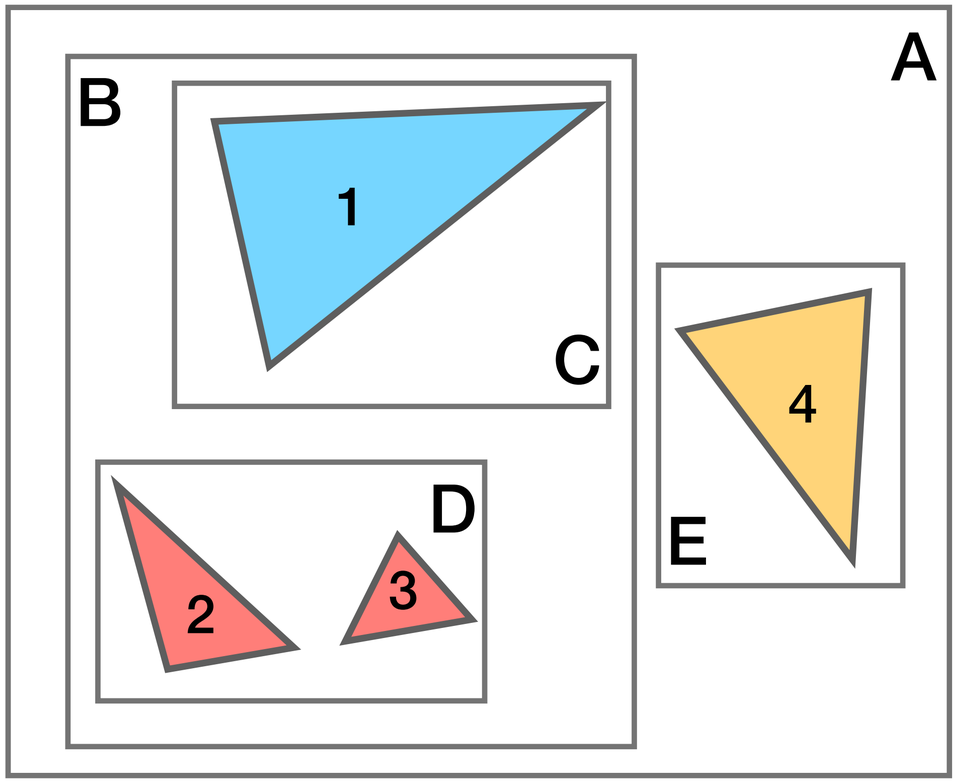
- Good reference: Ray Tracing with the BSP Tree [Ize, Wald, Parker, 2008]

Bounding Volume Hierarchy (Object Partition)

BVH Tree

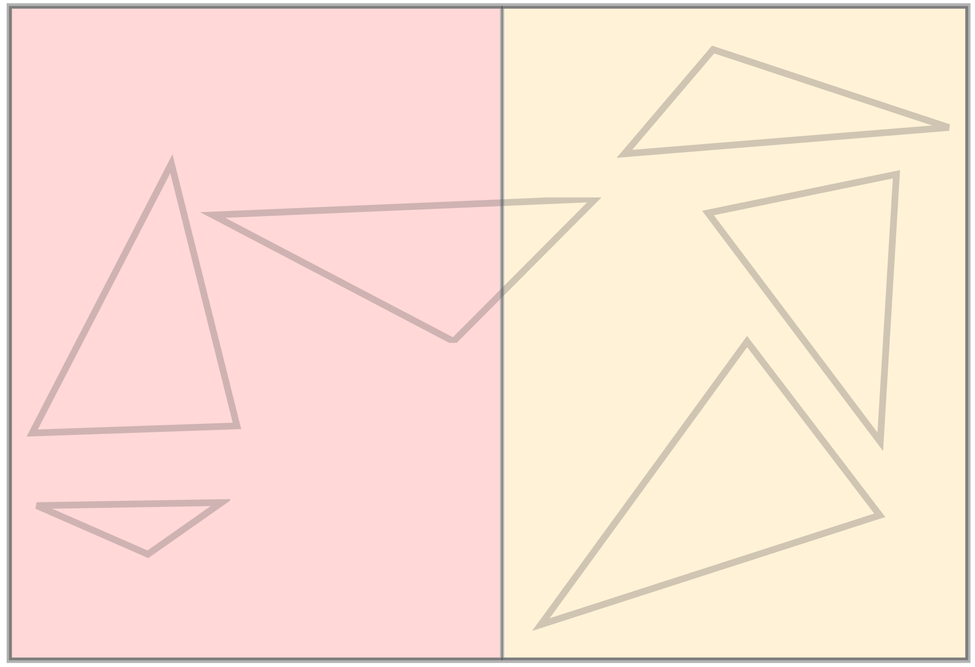


Scene

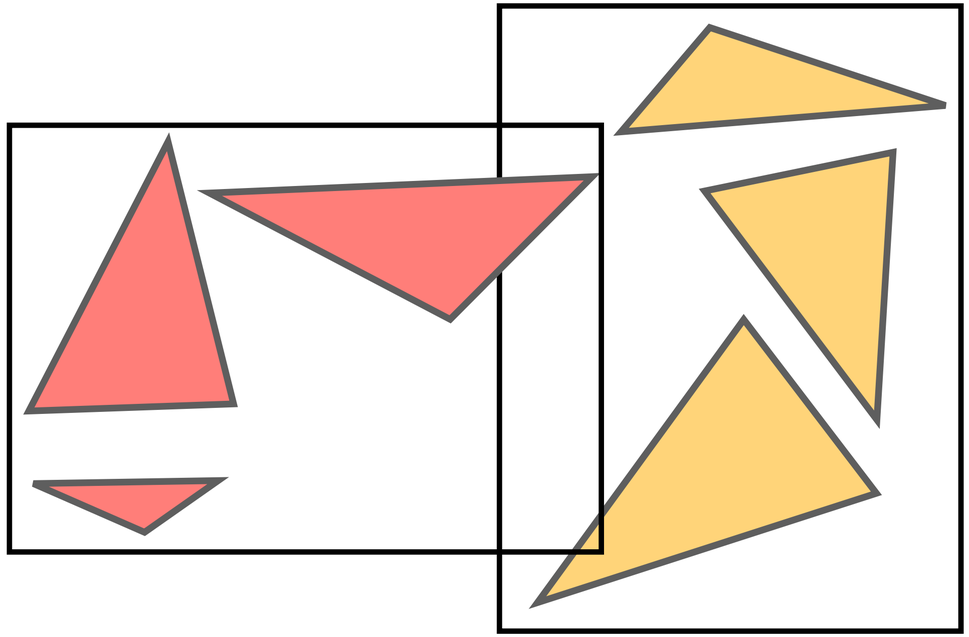


Object vs. Space Partitioning

Space partitioning: One object could be in different partitions

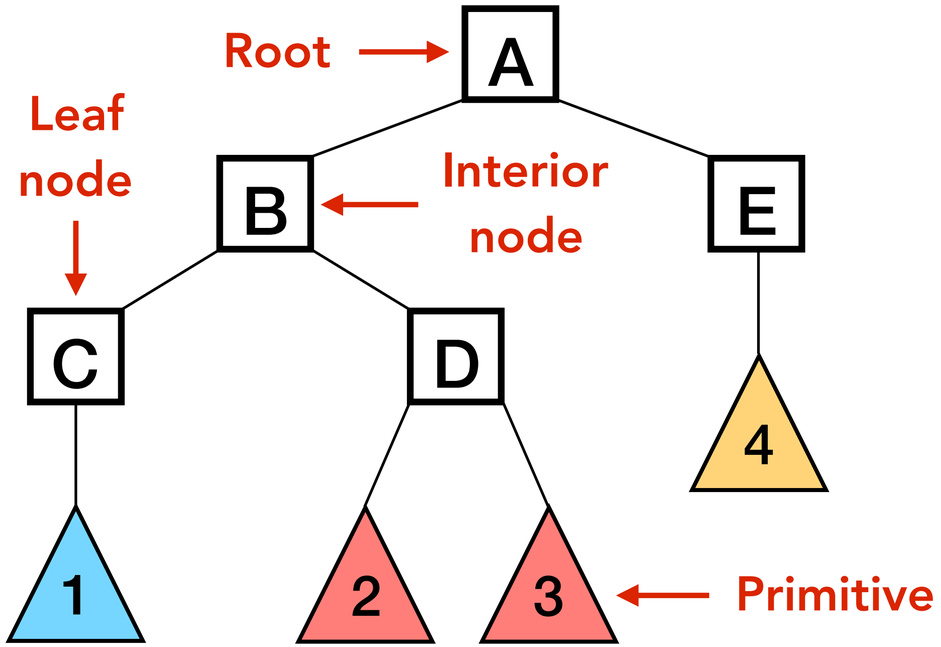


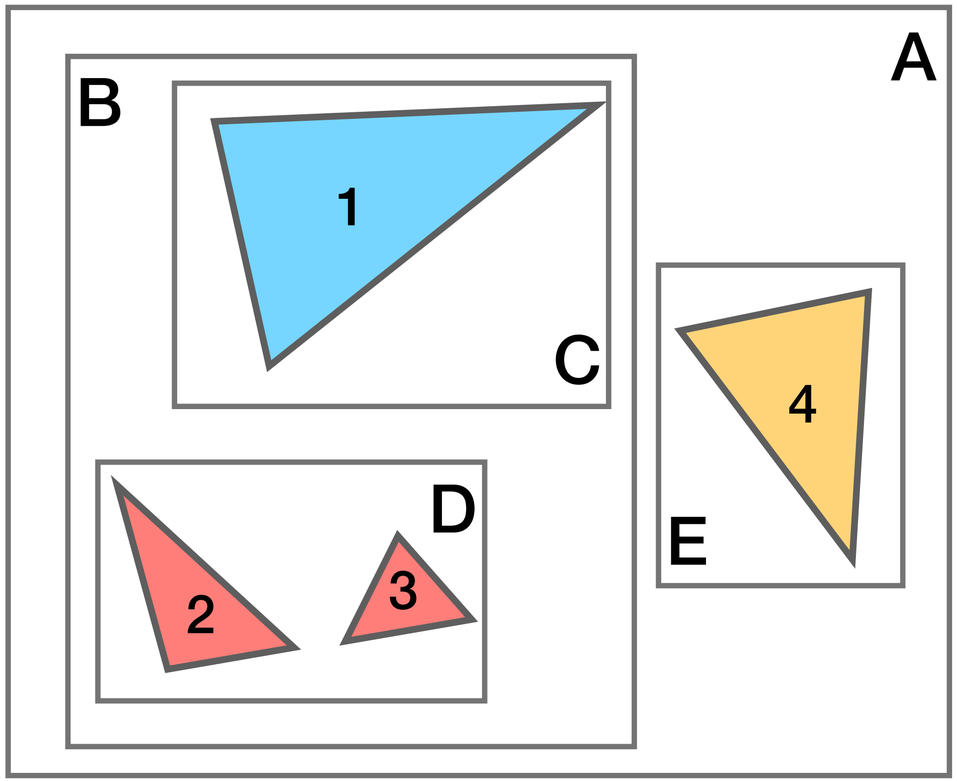
Object partitioning: different partitions could overlap in space



Bounding Volume Hierarchy (Object Partition)

- A, B, C, D, E are the bounding volumes, which are Axis-Aligned Bounding Boxes (AABBs) here. Other (irregular) bounding volumes are possible.



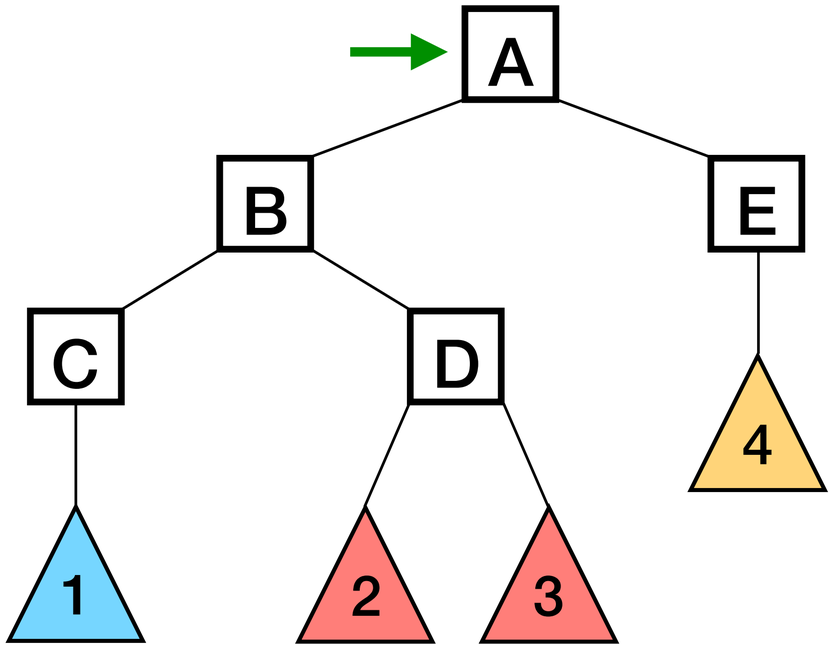


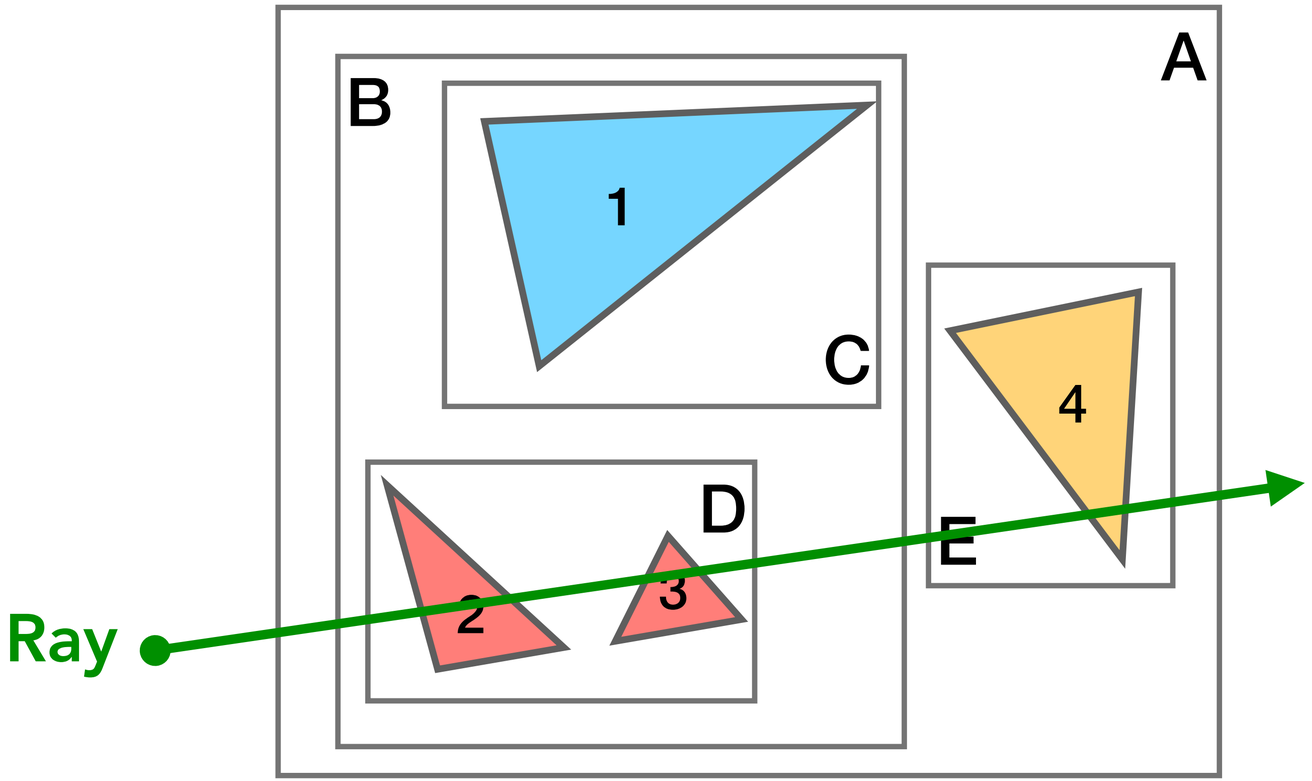
Intersection Test Using BVH

ClosestHit = NA

Current A Ray-AABB Intersection Test

Stack



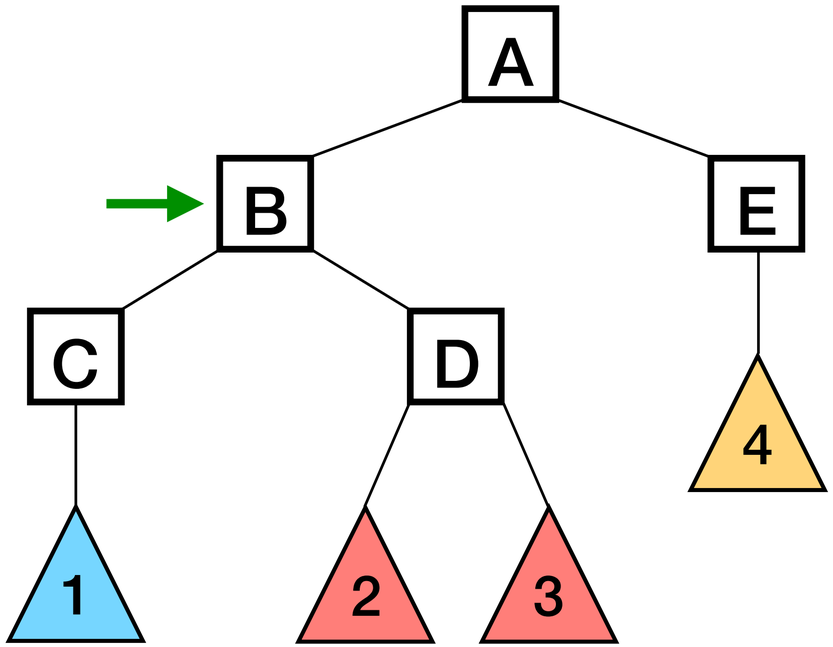


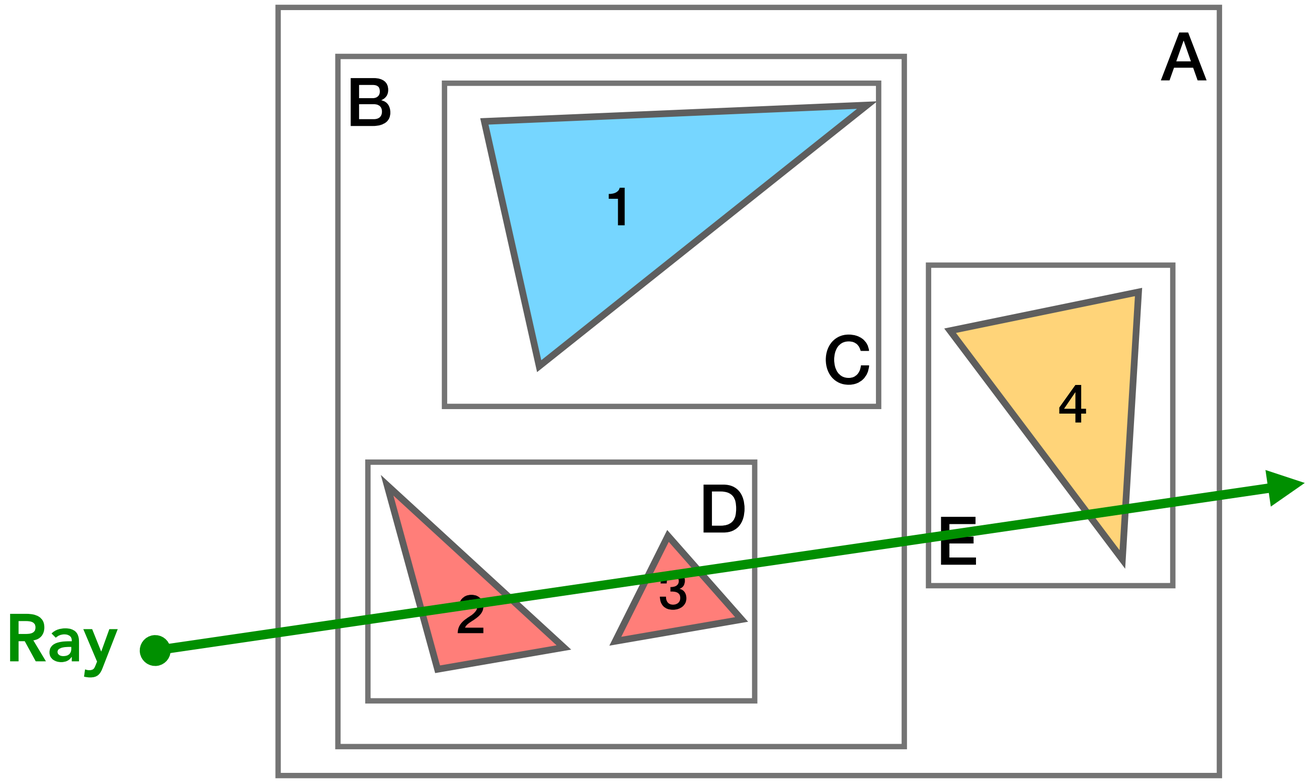
Intersection Test Using BVH

ClosestHit = NA

Current B Ray-AABB Intersection Test

Stack E



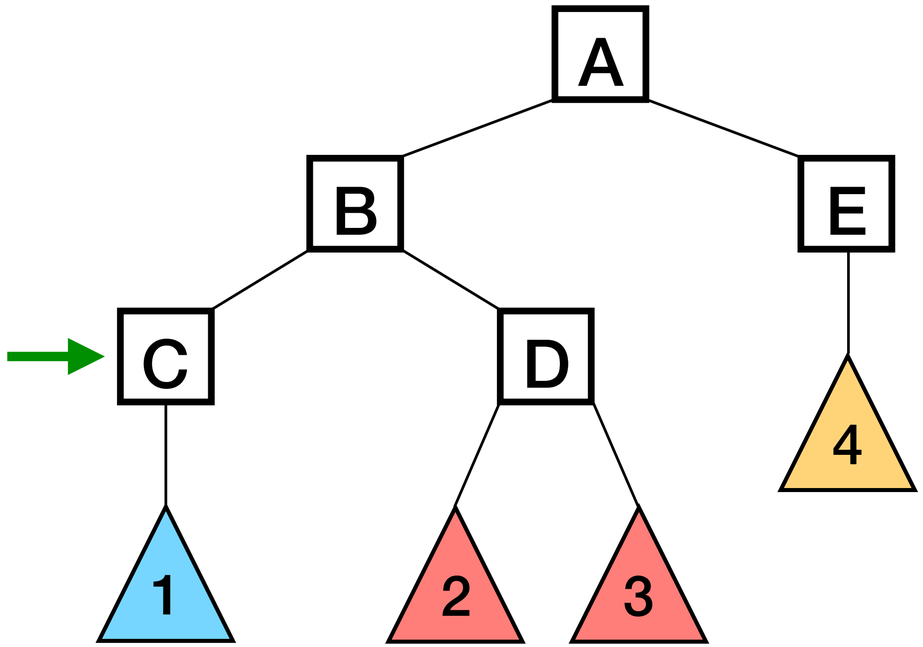


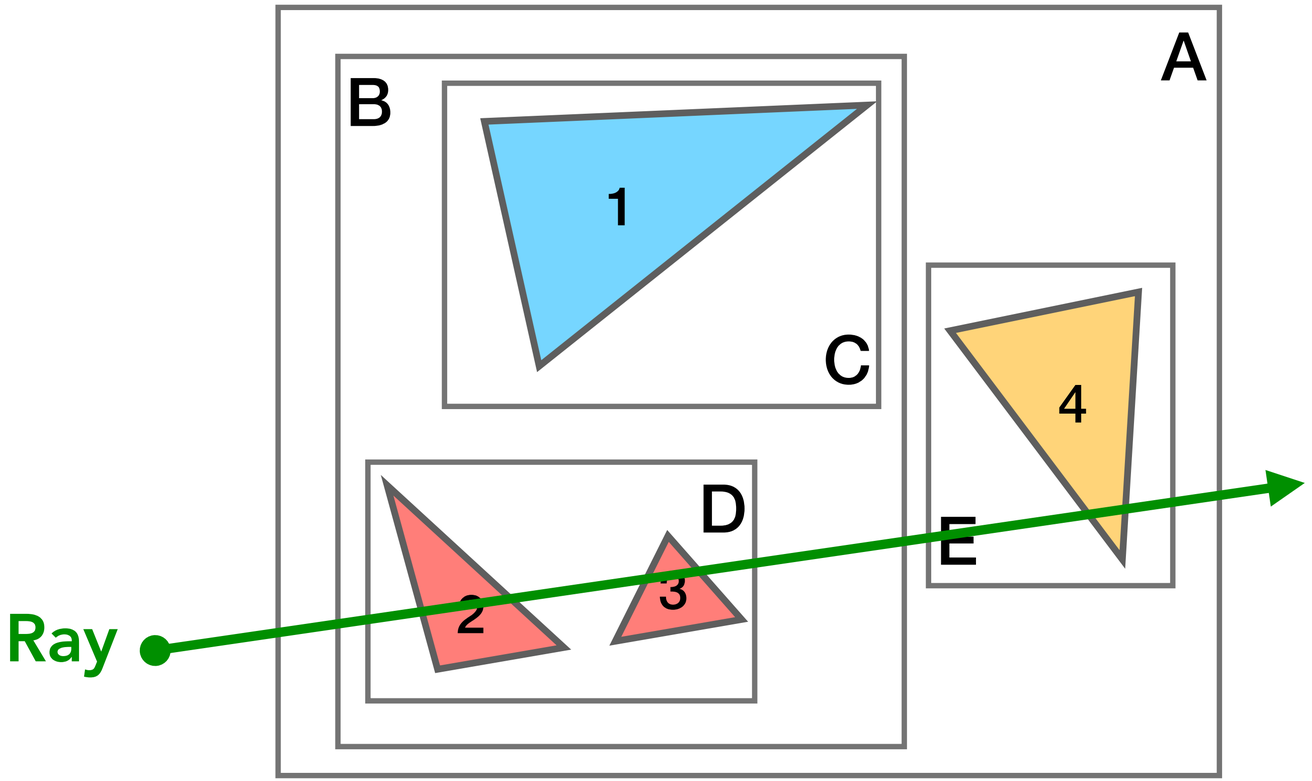
Intersection Test Using BVH

ClosestHit = NA

Current C Ray-AABB Intersection Test

Stack E D



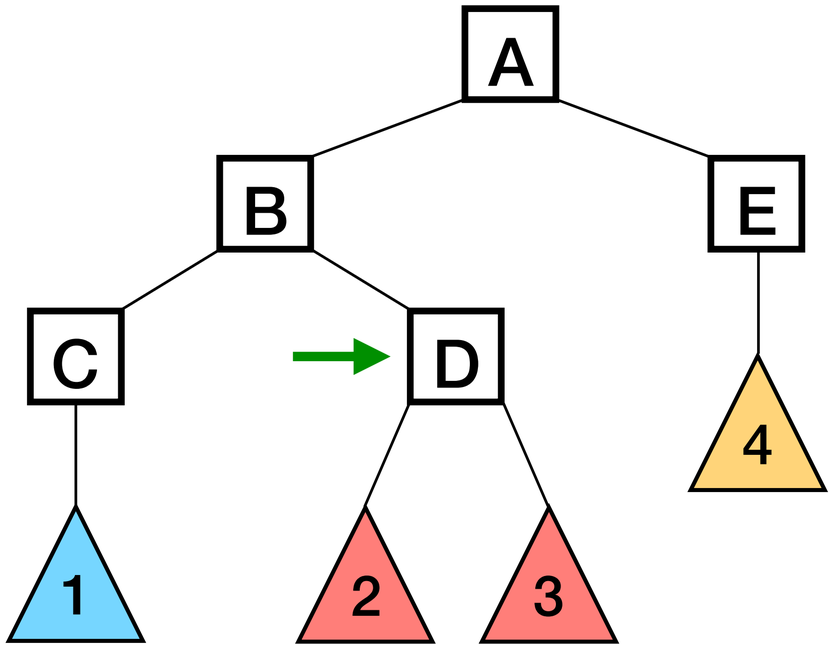


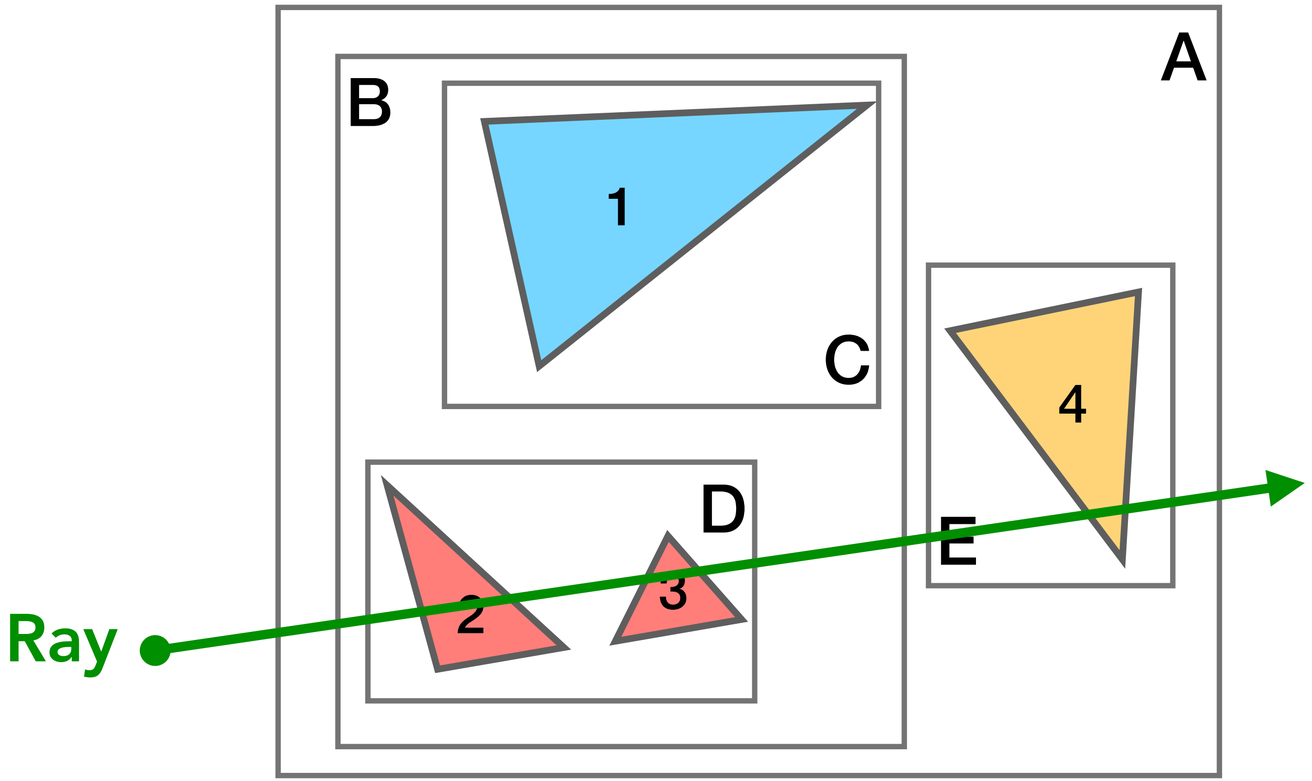
Intersection Test Using BVH

ClosestHit = NA

Current D Ray-AABB Intersection Test

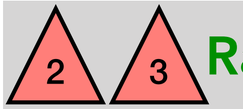
Stack E



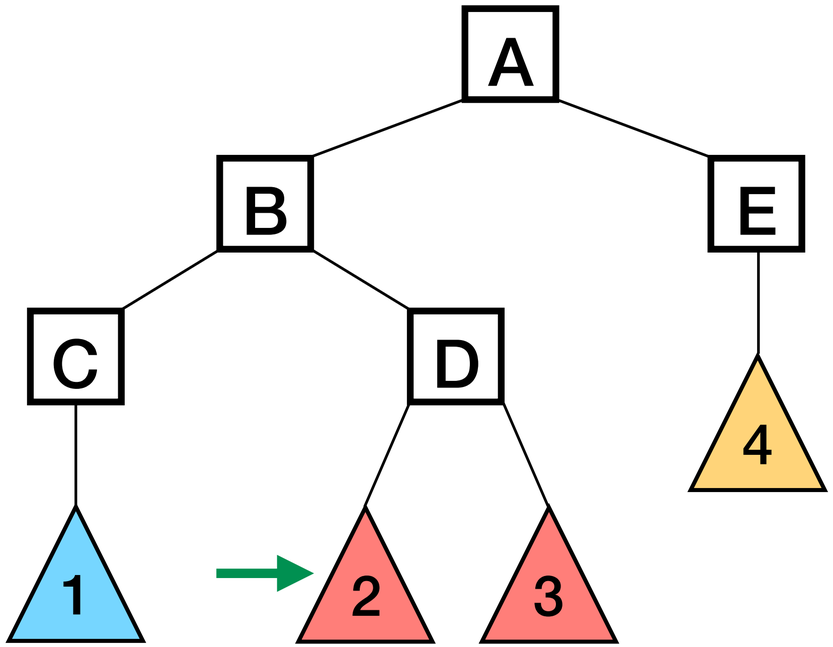


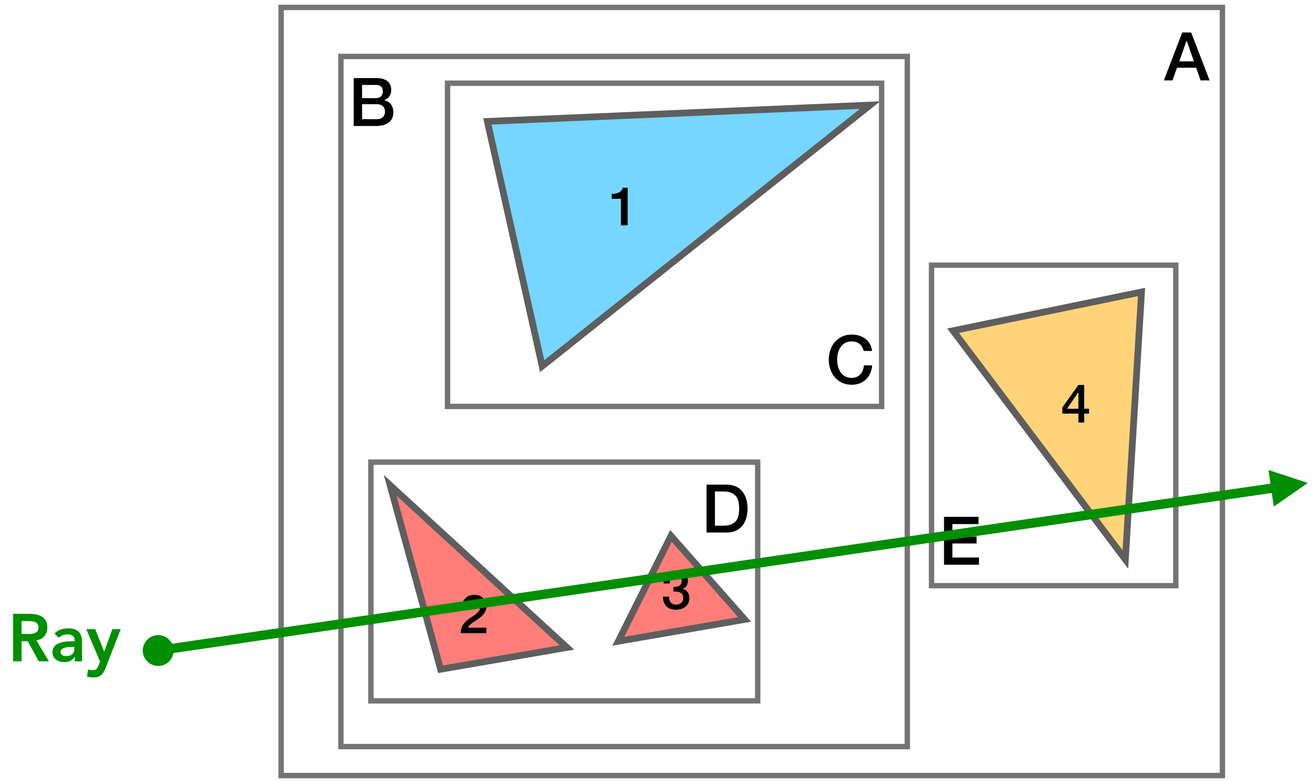
Intersection Test Using BVH

ClosestHit = NA

Current  **Ray-Triangle Intersection Test**

Stack E





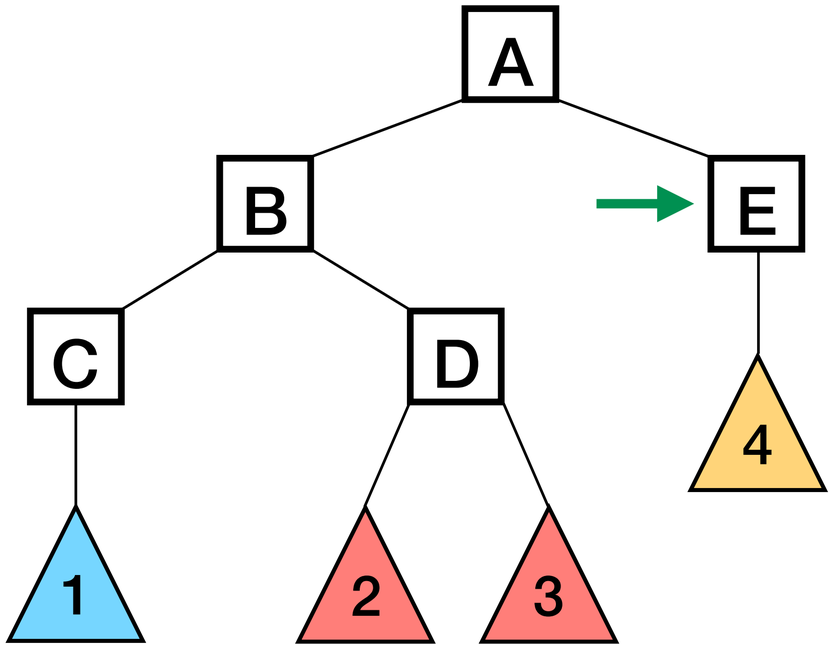
Intersection Test Using BVH

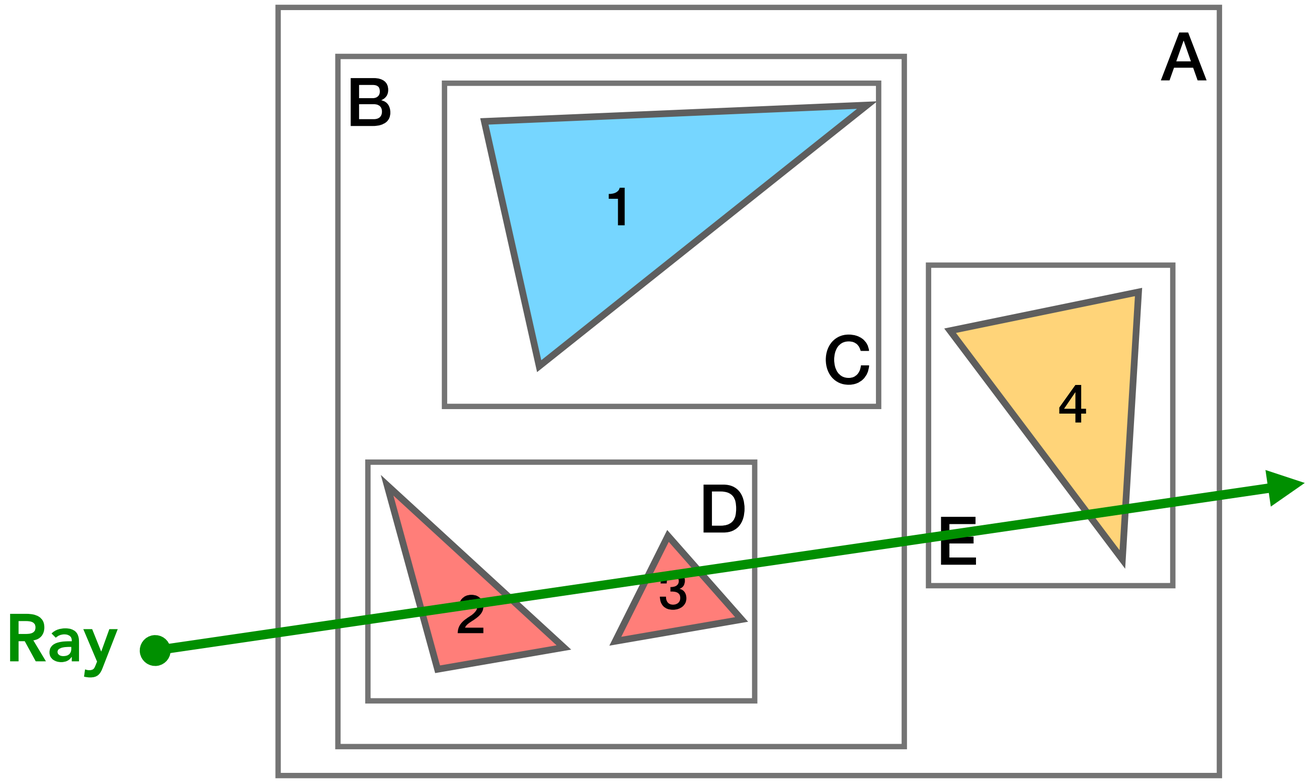
ClosestHit = 2

Current E Ray-AABB Intersection Test

Distance to E > Distance to 2; Stop!

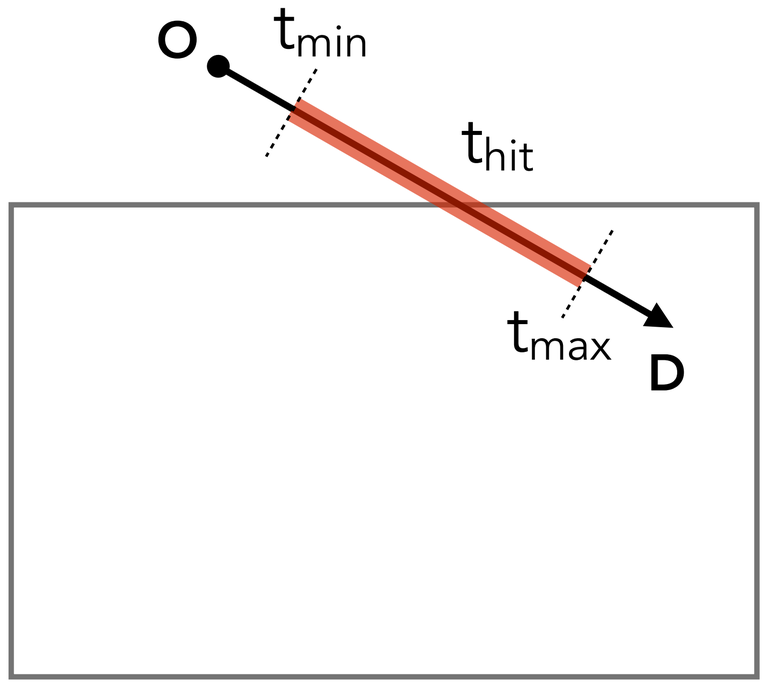
Stack





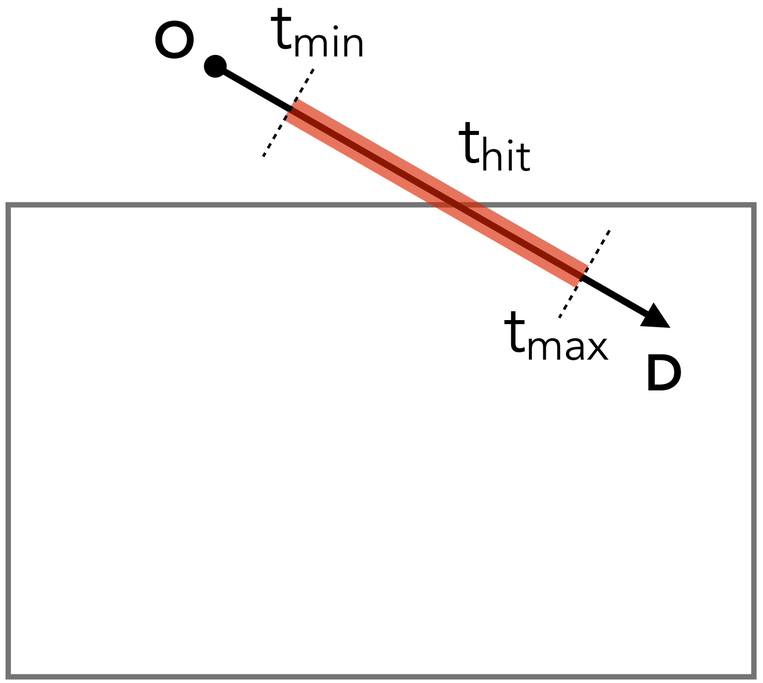
Ray-AABB Intersection

Ray: **O** + t**D**, t_{min} <= t <= t_{max}



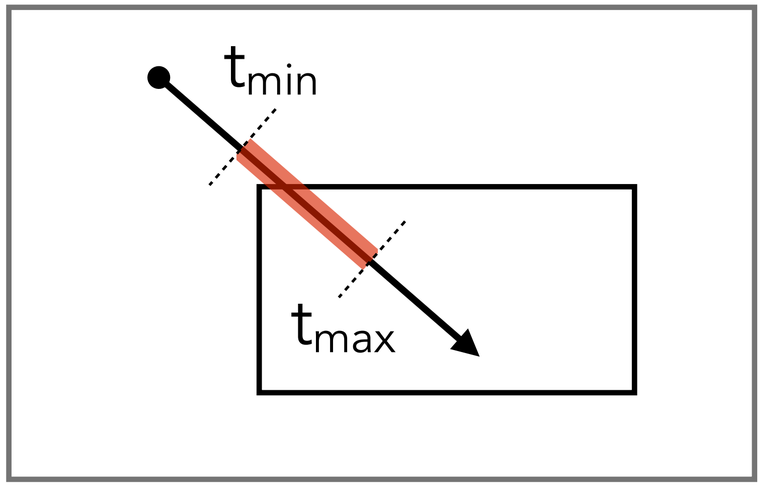
A Subtle but Critical Case

Ray: **O** + t**D**, t_{min} <= t <= t_{max}



Should this be counted as a hit?

Yes; any ray segment that originates from within an AABB must be treated as intersecting.



Various Trade-offs Worth Considering

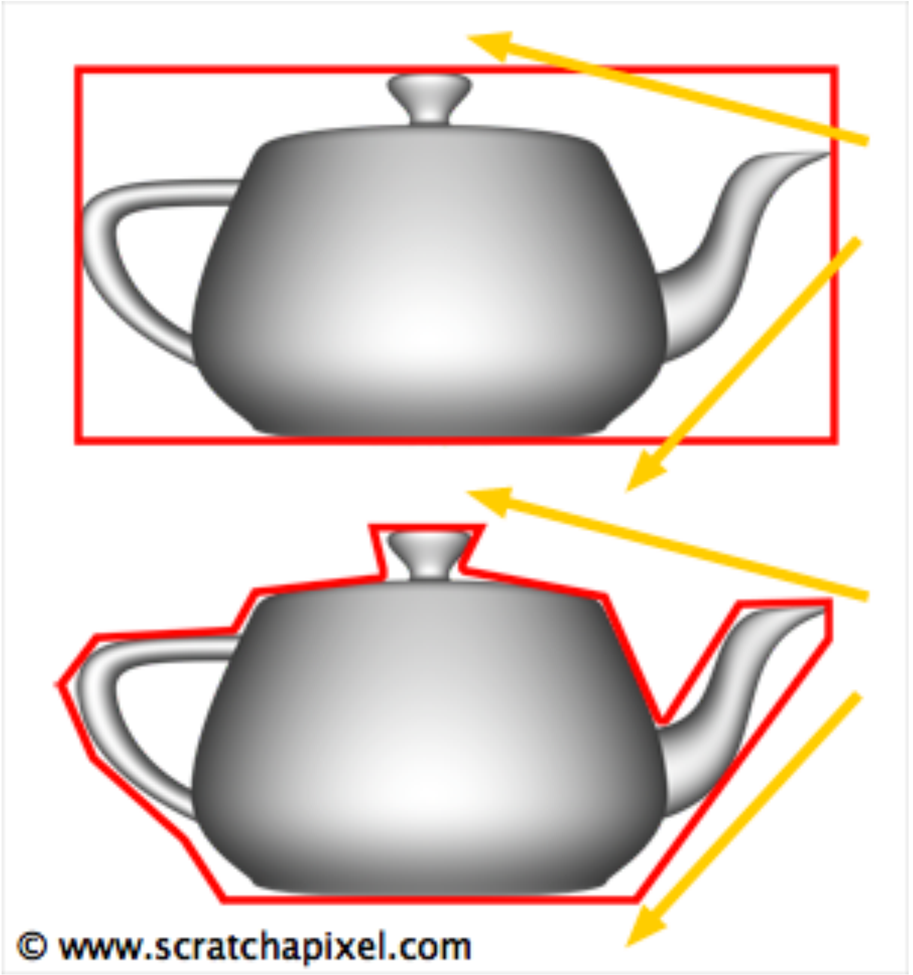
Time to build the tree vs. time to search.

- Incrementally update a tree (e.g., scene slowing changing in an animation)?
- Can we built the tree offline?

Shape of the bounding volume.

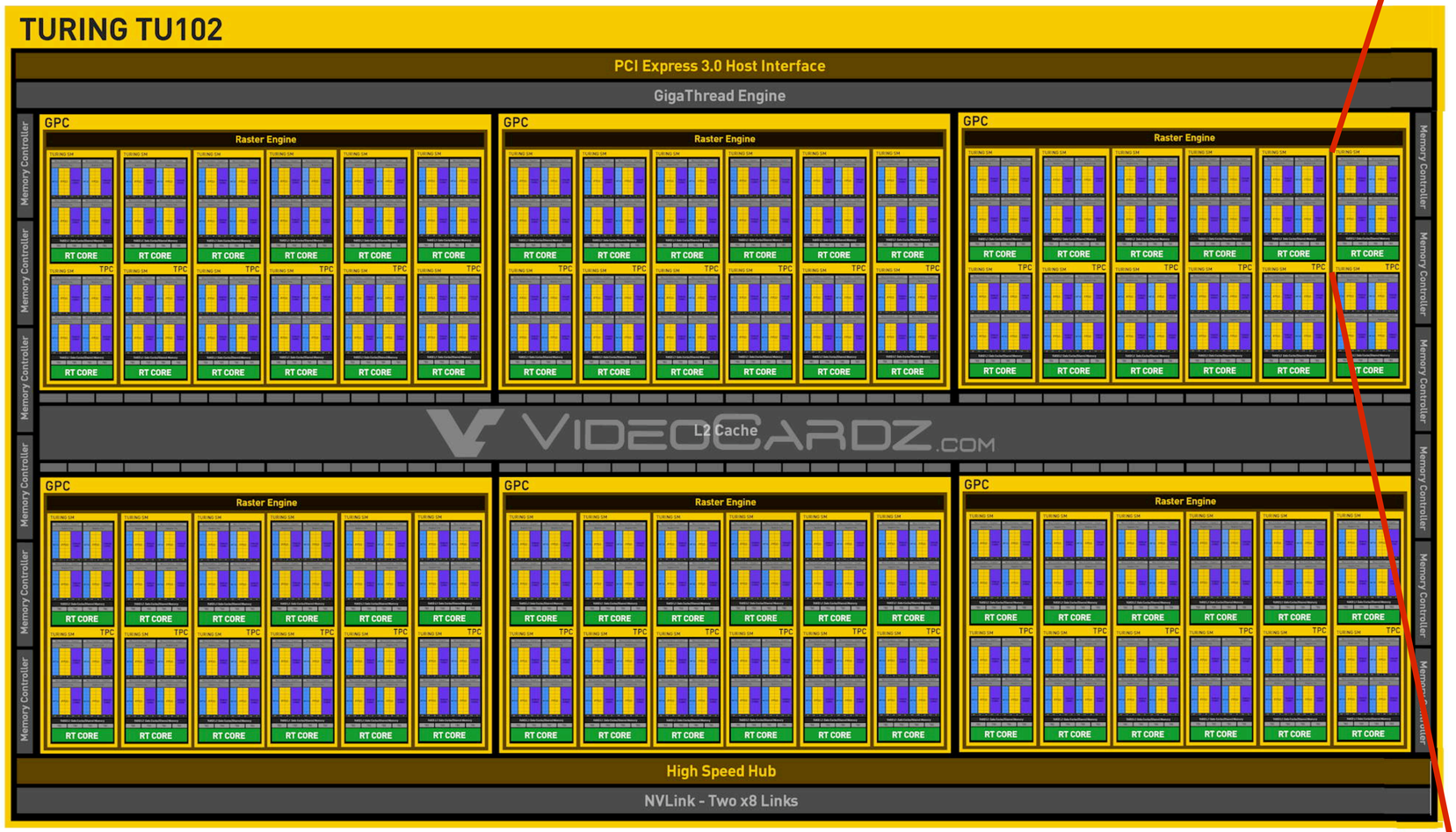
- Tight bounding volumes provide more precise intersect test, but are costly to build and to search.

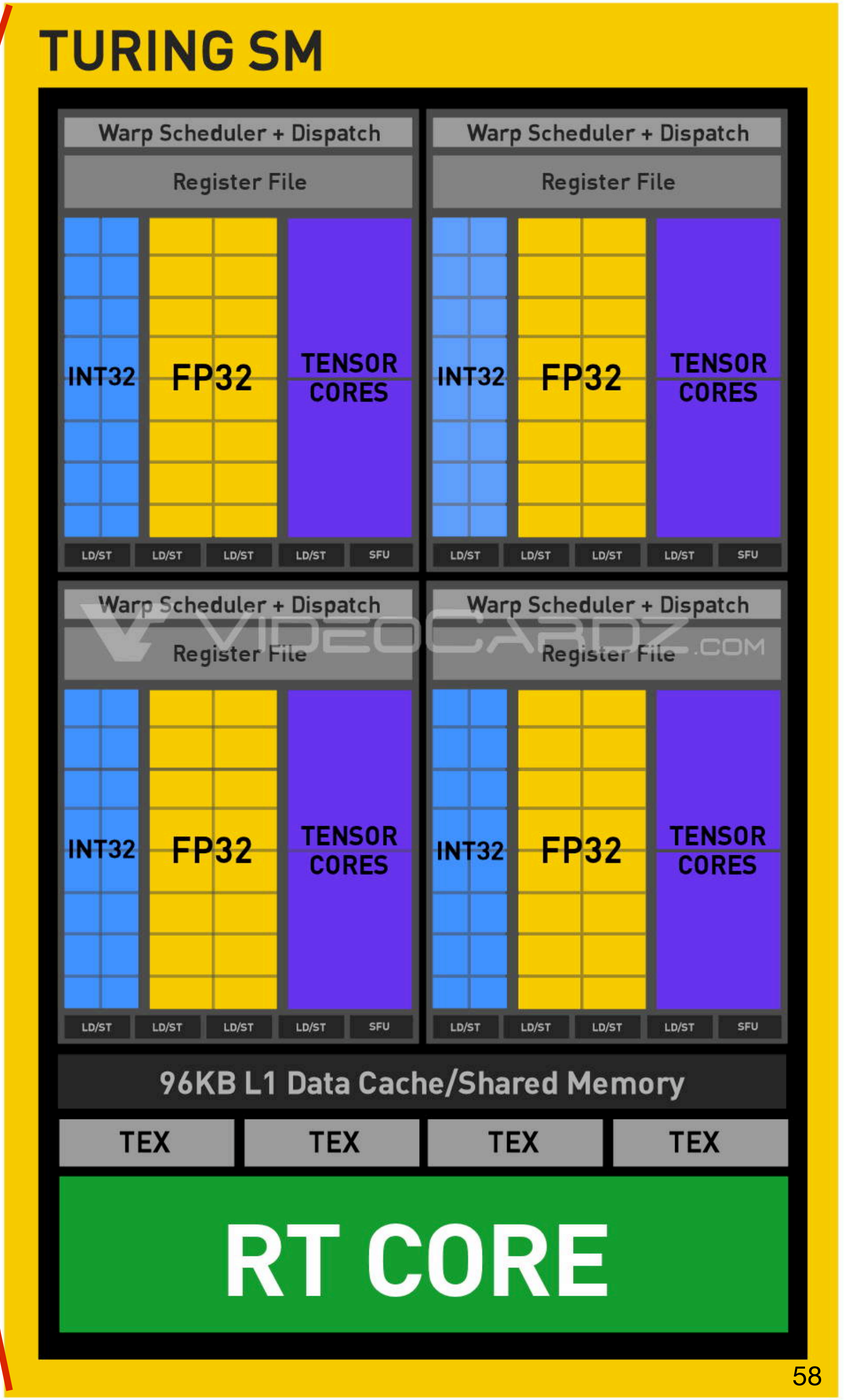
Tree structures take memory.



https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part1

An SM in Turing GPU



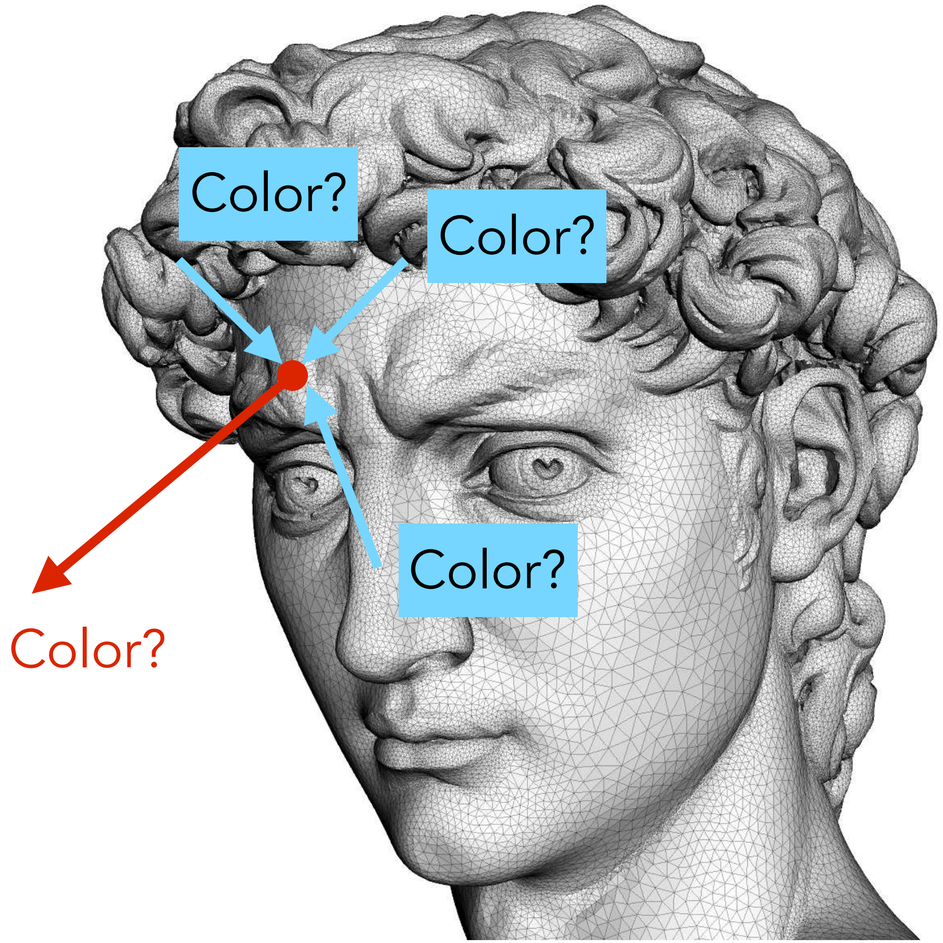


https://wccftech.com/nvidia-turing-gpu-architecture-geforce-rtx-graphics-cards-detailed/

Recursive Ray Tracing

Why Recursive Ray Tracing?

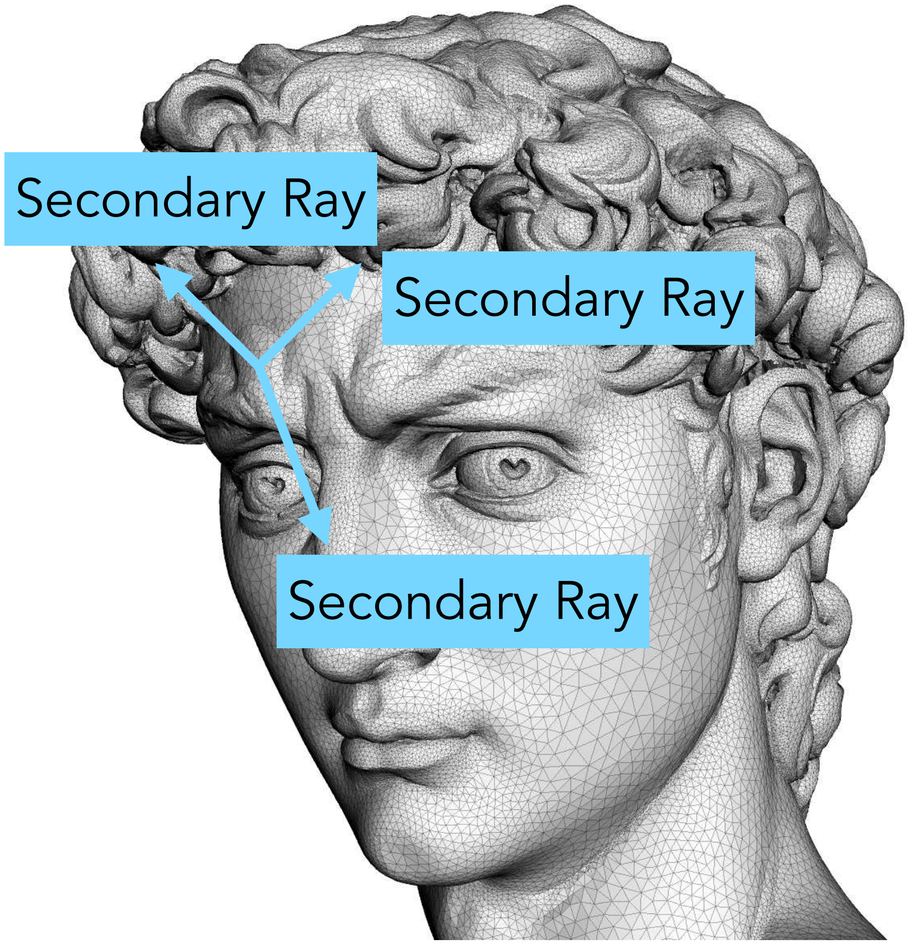
- To implement realistic shading.
- The color of an exiting ray depends on the colors of all incident rays.
  - color here really means radiance.
  - also depends on the surface material (diffuse vs. specular vs. …); later.
- How do we know the color of an incident ray? Cast more rays!



Valette, et al. [TVCG'08]

Why Recursive Ray Tracing?

- To implement realistic shading.
- The color of an exiting ray depends on the colors of all incident rays.
  - color here really means radiance.
  - also depends on the surface material (diffuse vs. specular vs. …); later.
- How do we know the color of an incident ray? Cast more rays!



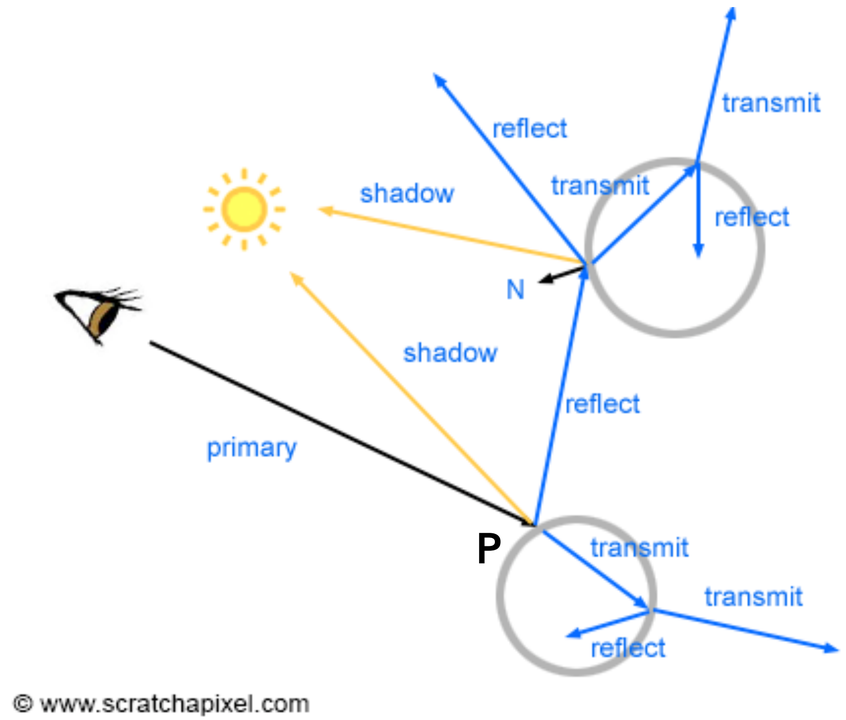
Valette, et al. [TVCG'08]

Simple Whitted-Style Recursive Ray Tracing

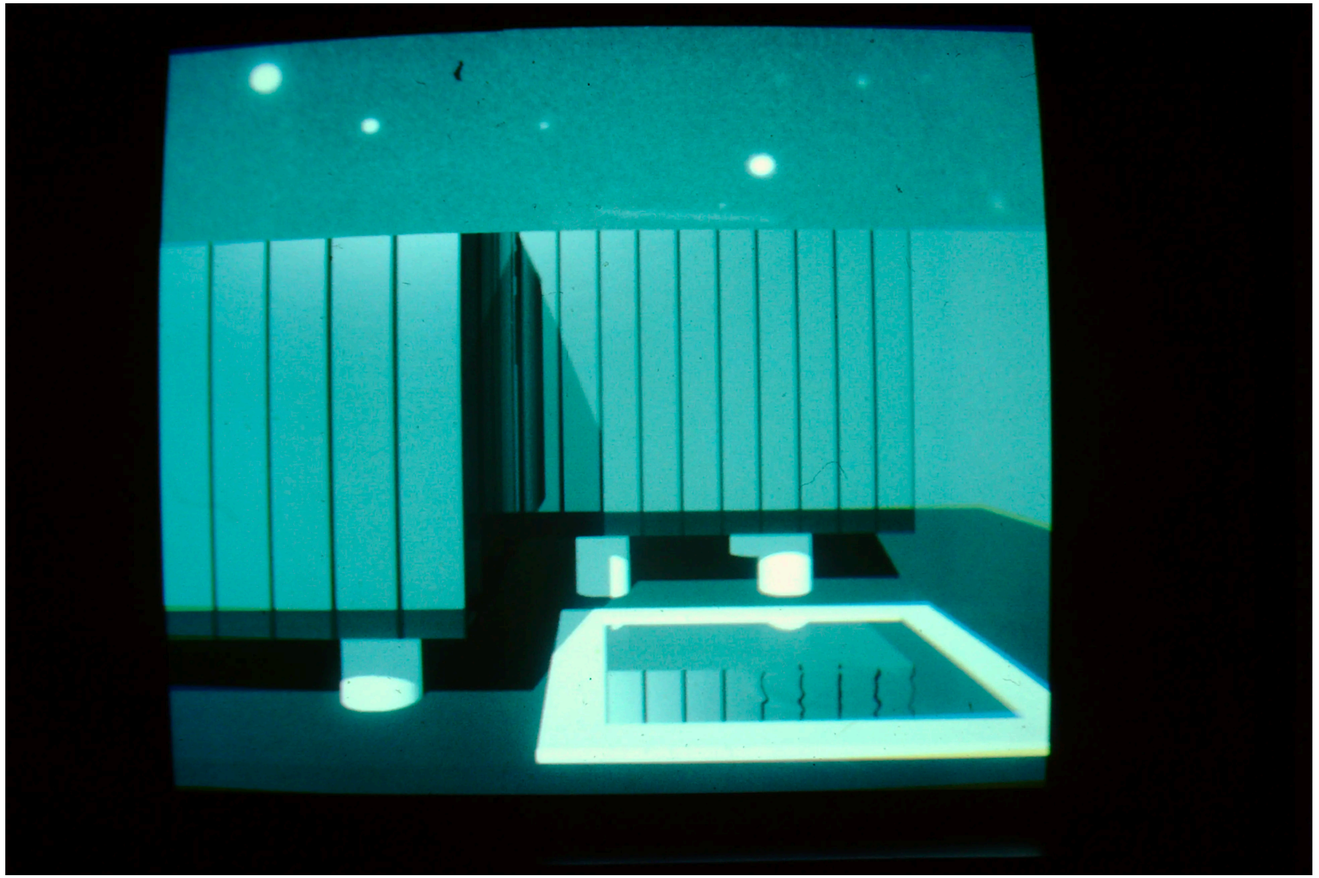
A simplification of Whitted-style ray tracing, assuming purely transparent surface.

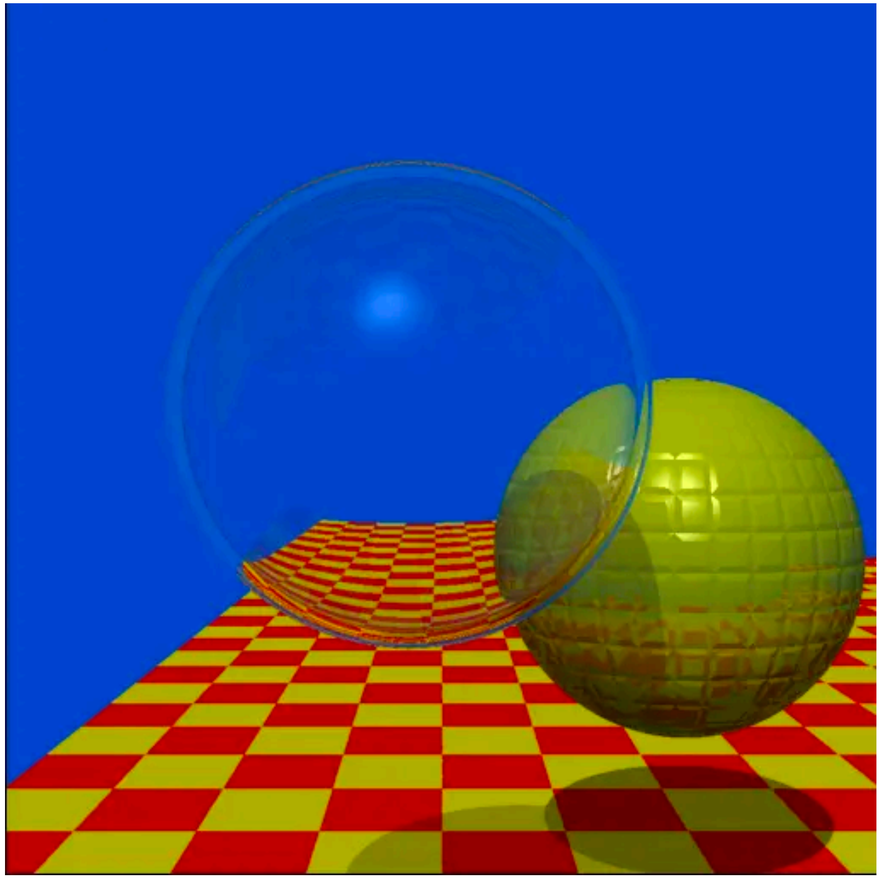
```
castRay(ray, mesh) {
  if (P = nearestIntersect(ray, mesh))
    reflectRay = buildReflectRay(P)
    refractRay = buildRefractRay(P)
    reflectColor = castRay(reflectRay, mesh))
    refractColor = castRay(refractRay, mesh))

    float kr
    fresnel(dir, N, hitObject->ior, kr)
    P.color = reflectionColor * kr +
              refractionColor * (1 - kr)
  else P.color = backgroundColor
}
```



Simple Whitted-Style Recursive Ray Tracing





https://blogs.nvidia.com/blog/2018/08/01/ray-tracing-global-illumination-turner-whitted/

Things to Remember

Ray tracing makes it easy (conceptually) to implement realistic shading.

Compared to rasterization, ray tracing is much more time consuming, dominated by ray-scene intersection test, which is exacerbated by the need for recursive ray tracing.

We can accelerate the testing using acceleration structures that prune the search space. BVH is the most common acceleration structure.

Modern GPUs, while traditionally optimized for rasterization, now have hardware support for ray tracing (e.g., BVH traversal, ray-AABB/triangle intersection test).