

Ray Tracing II

Improving Raytracing Speed

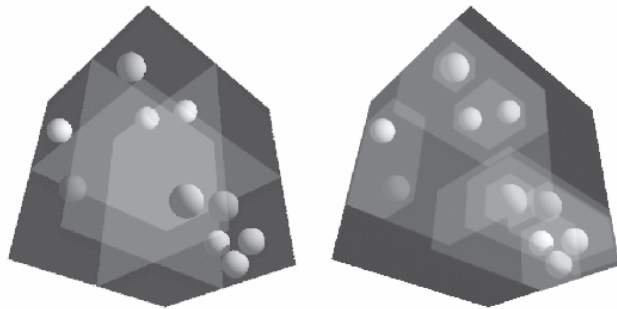
Raytracing Computational Complexity

- ray-scene intersection is expensive
 - improve by low-level optimizations
 - important but does not provide scalability
 - improve by reducing computational complexity

Improving Computational Complexity

- ray-scene intersection is an $O(n)$ algorithm, where n is the number of objects
 - loop over each object, check intersection
 - note that it is fundamentally a search algorithm
- use divide and conquer to turn it to $O(\log n)$
 - similar principles of search algorithms
 - divide objects into sets
 - discard sets of objects quickly
 - test inside set only when necessary
 - not as easy since data structures become complicated
- which data structures to use?

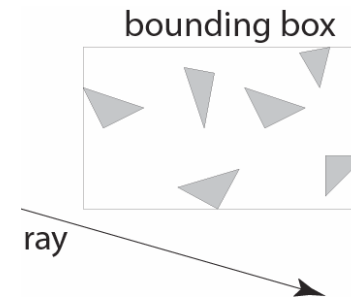
Acceleration Structures Examples



[Shirley – original from Dematle]

Bounding Volumes

- wrap groups of objects in bounding volumes
 - all points on the objects are inside the bounding vol.
 - if ray does not hit volume, than it does hit the object
 - if ray hits the volume, test the object



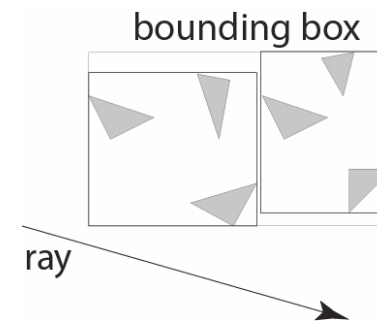
[Shirley]

Bounding Volumes

- worth if volume hit-testing cheaper than object's
 - e.g. polygon mesh inside a bounding sphere
- need to know if intersection exists, not where
 - speed up by removing computation
- use only simple primitives
 - sphere, axis-aligned boxes (AABB), oriented boxes (OBB)
 - choose based on tightness of fit vs. intersection speed
 - increasing tightness: spheres, AABBs, OBBs
 - increasing speed: OBBs, AABBs, spheres

Axis-Aligned Bounding Boxes

- most commonly used
- intersection with multiple infinite slabs



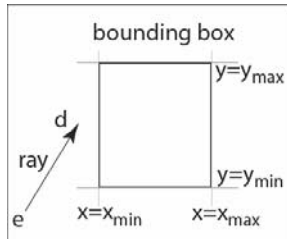
[Shirley]

Axis-Aligned Boxes in 2D

$$\mathbf{P} = \mathbf{E} + t\mathbf{I} \quad \text{point on a ray}$$

$$P_x \in [x_m, x_M] \wedge P_y \in [y_m, y_M] \quad \text{point in the box}$$

assume $I_x > 0 \wedge I_y > 0$
 compute t for each plane



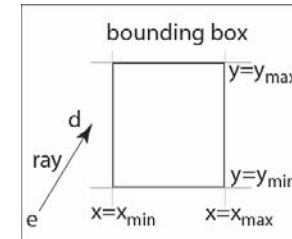
[Shirley]

Axis-Aligned Boxes in 2D

$$t_{xm} = (x_m - E_x) / I_x \quad t_{ym} = (y_m - E_y) / I_y$$

$$t_{xM} = (x_M - E_x) / I_x \quad t_{yM} = (y_M - E_y) / I_y$$

intersect if $t_{xm} \leq t_{yM} \wedge t_{ym} \leq t_{xM}$
 since $t \in [t_{xm}, t_{xM}] \wedge t \in [t_{ym}, t_{yM}]$



[Shirley]

Axis-Aligned Boxes

extend to other quadrants by redefining intersections

for $I_x < 0$ define

$$t_{xm} = (x_M - E_x) / I_x$$

$$t_{xM} = (x_m - E_x) / I_x$$

for $I_y < 0$ define

$$t_{ym} = (y_M - E_y) / I_y$$

$$t_{yM} = (y_m - E_y) / I_y$$

avoid division by zero as in Shirley 10.9

extend to 3D by including two new planes

Hierarchical Bounding Volumes

- group bounding volumes hierarchically
 - this is what gives us the *expected* $O(\log n)$
 - not provable, strongly depends on input data
- volumes bound all surfaces inside them
- not a perfect split: siblings volumes can overlap
 - hierarchical intersection testing
 - if parent does not intersect, then no intersection
 - else intersect all children and pick the closest intersection (if any)

Creating Bounding Hierarchies

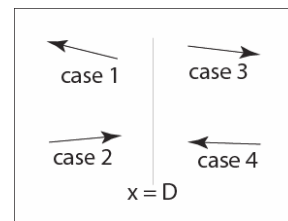
- effective space partitioning is necessary
 - following transform hierarchy is not always efficient
- basic greedy algorithm for binary trees
 - pick a direction, say along x axis
 - split objects into two groups, and bound them
 - continue splitting each group
 - change axis at each iteration: x, y, z, x, y, z, \dots
- goal 1: balanced number of children in subtrees
 - split for same number of primitive in each group
- goal 2: equal space for each subtree
 - split the parent bounding box in the middle spatially

Binary Space Partitioning (BSP) Trees

- conceptually similar to bounding hier. creation
- node defined by splitting plane
 - often pick a plane orthogonal to x , then y , then z , ...
- children contain objects in one of two half spaces
 - objects intersecting plane are in each child

Intersecting BSP trees

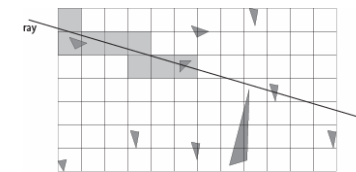
- starts at the top and determine which of
- case 1: test only Left
- case 2: test Left; if no intersections test Right
- case 3: test Right; if no intersections test Left
- case 4: test only Right



[Shirley]

Regular Space Subdivision

- non hierarchical acceleration structure
- uniformly split scene in a volume grid
 - same object can be in multiple grid cell
 - but a point can only be in one cell at a time
- intersect by walking the grid incrementally



[Shirley]

Hierarchical Regular Space Subdivision

- at each cell of a regular grid, store a regular grid
 - normally 2-3 levels deep
- $O(kn)$ behavior, with $k \ll 1$
 - expect to skip a constant fraction of all objects
 - very fast incremental walk
 - easy to update
 - not efficient for scenes with big holes in them

Acceleration Structures

- which one is the winner?
 - depends on type of input
 - and if we need to update for animation
- start with a hierarchical grid

Software Engineering Considerations

- acceleration structs are subclasses of Surface
 - high-level raytracing code should not know about them
 - can switch data structures
 - can combine data structures
 - hierarchical grids as grids of grids
 - hierarchical bounding volumes as volumes of surfaces
 - hierarchical boxes with a grid per box in the leaves
- regular grid: stores Surface[n][n][n]
- bounding volume: stores Surface[n]
 - often binary for simplicity
- BSP node: stores Surface[2]

Texture Mapping for Raytracing

Texture Mapping Considerations

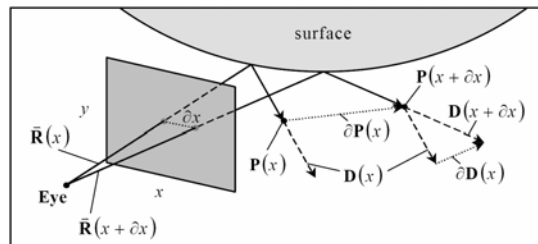
- especially efficient in raytracing
 - since ray-object intersection is very expensive
 - trivial support for texture and bump mapping
 - unclear how to efficiently do displacement maps
 - makes us lose raytracing nice properties
- mapping function: object-intersection code returns uv parameters

Determining Mapping Function

- sphere: given intersection point P , return angles from spherical coordinates
- triangle: change intersection code to use barycentric coordinates
 - Shirley, Ch. 10.3.2

Filtering Textures

- compute average of texture subtended by pixel
 - in general quite hard for reflection/refraction
- ray differentials
 - new and elegant discovery: Igehy, SIGGRAPH 1999
 - propagate differentials together with rays



[Igehy, 1999]

Filtering Texture Comparison

no filtering

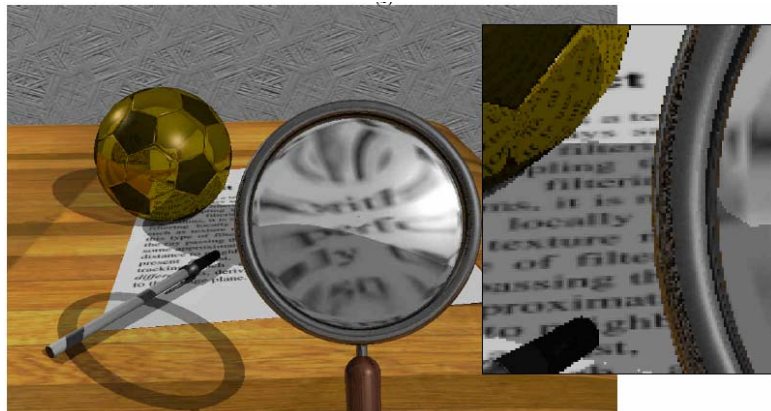


bad for view rays

[Igehy, 1999]

Filtering Texture Comparison

mip-map filtering based on distance



[Igehy, 1999]

ok for view ray, bad for reflection/refractions

Filtering Texture Comparison

mip-map filtering based on ray differentials



[Igehy, 1999]

ok in most cases, just a bit of overblur

Filtering Texture Comparison

accurate filtering based on ray differentials



[Igehy, 1999]

works great