CS4620/5620: Lecture 37

Ray Tracing

Announcements

- Review session
  - Tuesday 7-9, Phillips 101
- Posted notes on slerp and perspective-correct texturing
- Prelim on Thu in B17 at 7:30pm

Basic ray tracing

- Basic ray tracer: one sample for everything
  - one ray per pixel
  - one shadow ray for every point light
  - one reflection ray per intersection
    - one refraction ray (if necessary) per intersection
- Many advanced methods build on the basic ray tracing paradigm

Soft shadows

Creating soft shadows

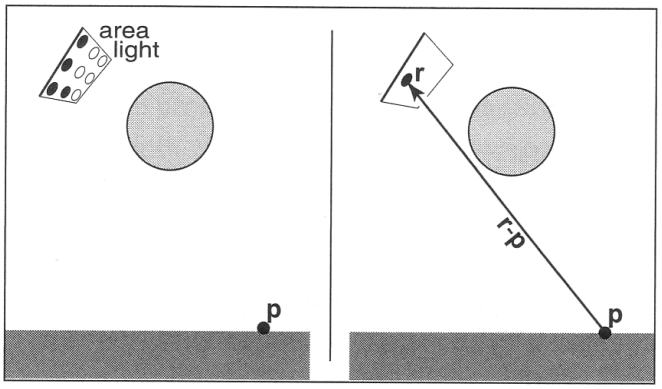
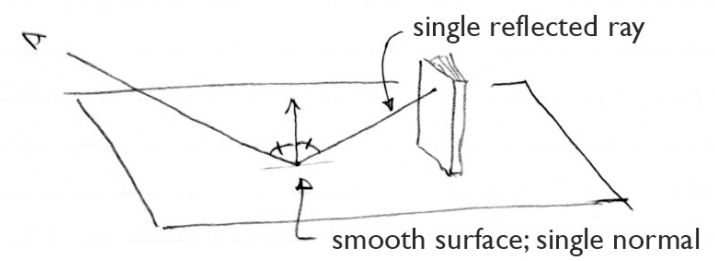


Figure 13.13. Left: an area light can be approximated by some number of point lights; four of the nine points are visible to **p** so it is in the penumbra. Right: a random point on the light is chosen for the shadow ray, and it has some chance of hitting the light or not.

Glossy reflection

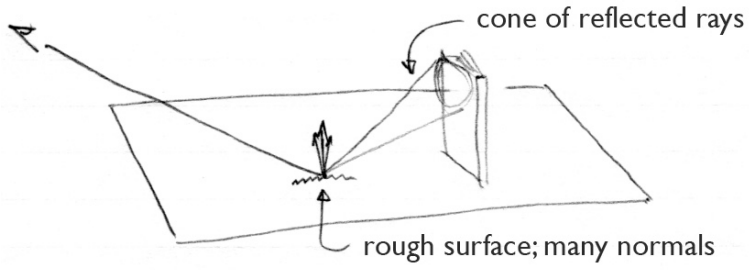
[Lafortune et al. 97]

Cause of glossy reflection



smooth surfaces produce sharp reflections

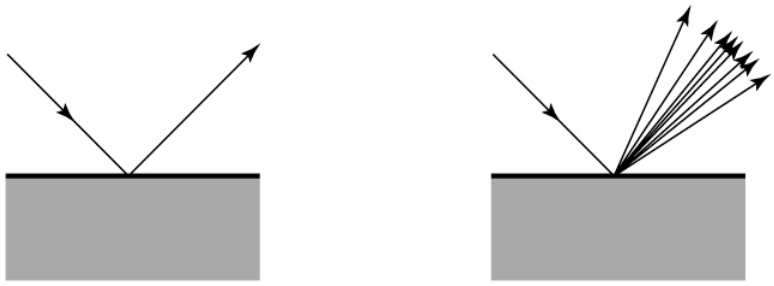
Cause of glossy reflection



rough surfaces produce soft (glossy) reflections

Creating glossy reflections

- Jitter the reflected rays
  - Not exactly in mirror direction; add a random offset
  - Can work out math to match Phong exactly
  - Can do this by jittering the normal if you want



Creating glossy reflections

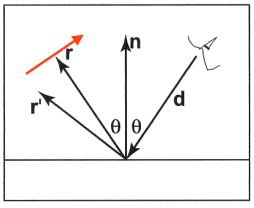


Figure 13.18. The reflection ray is perturbed to a random vector $\mathbf{r}'$.

To choose $\mathbf{r}'$, we again sample a random square. This square is perpendicular to $\mathbf{r}$ and has width a which controls the degree of blur. We can set up the square's orientation by creating an orthonormal basis with $\mathbf{w} = \mathbf{r}$ using the techniques in Section 2.4.6. Then, we create a random point in the 2D square with side length a centered at the origin. If we have 2D sample points $(\xi, \xi') \in [0, 1]^2$, then the analogous point on the desired square is

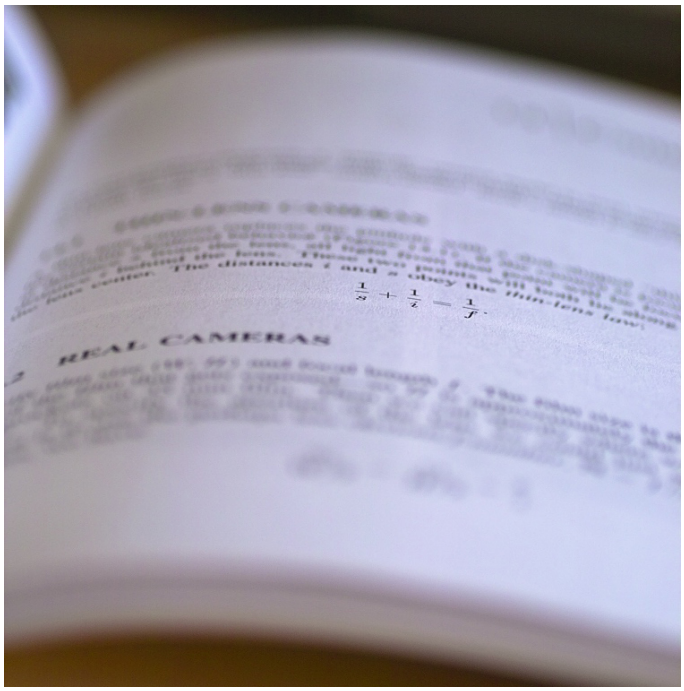
$$u = -\frac{a}{2} + \xi a,$$
$$v = -\frac{a}{2} + \xi' a.$$

Because the square over which we will perturb is parallel to both the $\mathbf{u}$ and $\mathbf{v}$ vectors, the ray $\mathbf{r}'$ is just

$$\mathbf{r}' = \mathbf{r} + u\mathbf{u} + v\mathbf{v}.$$

Note that $\mathbf{r}'$ is not necessarily a unit vector and should be normalized if your code requires that for ray directions.

Depth of field



Cause of focusing effects

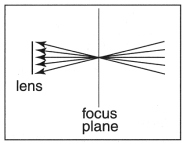
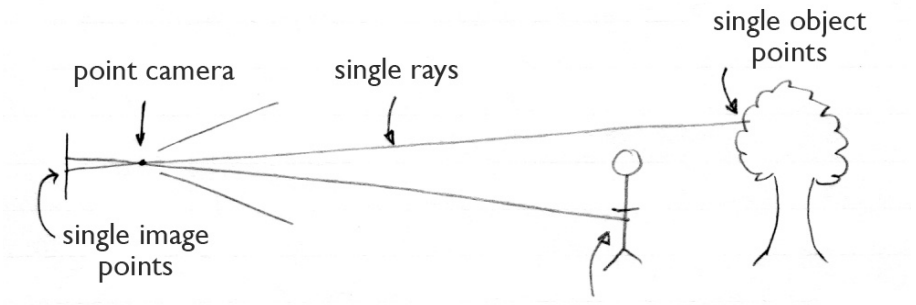
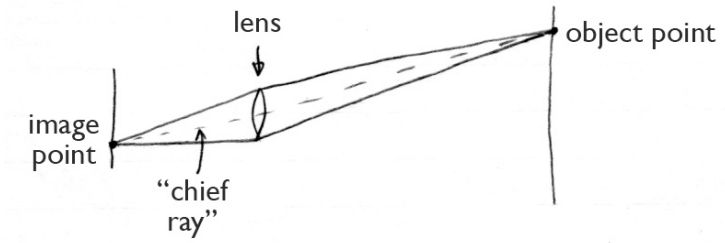


Figure 13.15. The lens averages over a cone of directions that hit the pixel location being sampled.



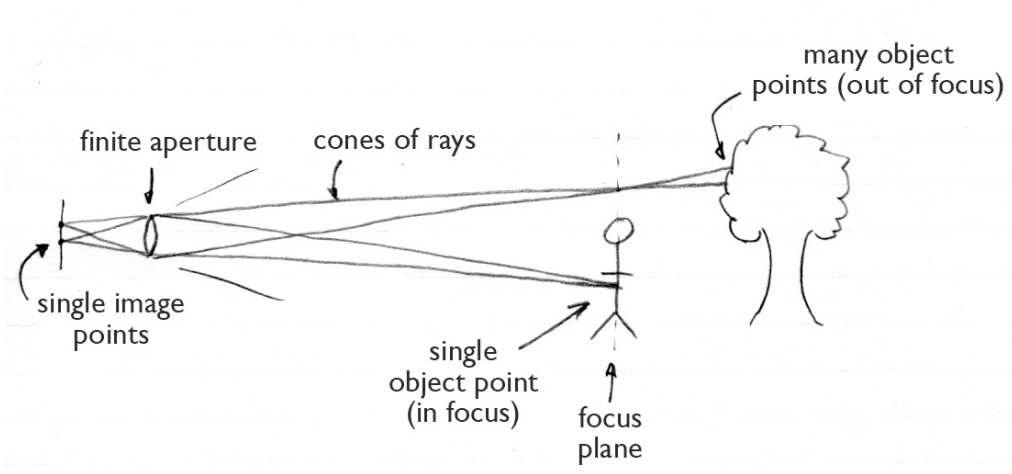
point aperture produces always-sharp focus

Cause of focusing effects



what lenses do (roughly)

Cause of focusing effects



finite aperture produces limited depth of field

Depth of field

- Make eye rays start at random points on aperture
  - always going toward a point on the focus plane

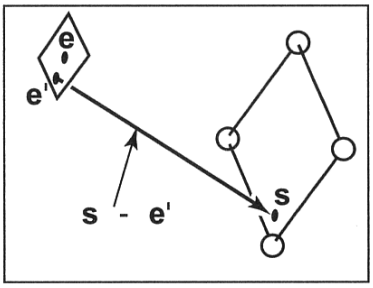
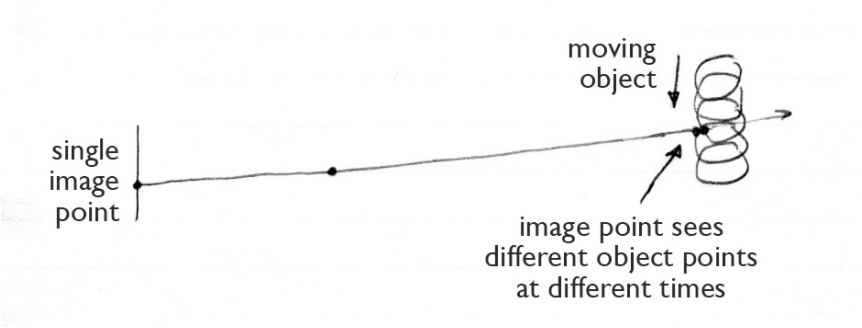


Figure 13.17. To create depth-of-field effects, the eye is randomly selected from a square region.

Motion blur

[Cook, Porter, Carpenter 1984]

Cause of motion blur



Motion blur

- Caused by finite shutter times
- Introduce time as a variable throughout the system
  - object are hit by rays according to their position at a given time
- Then generate rays with times distributed over shutter interval

$$T = T_0 + \xi(T_1 - T_0)$$

Generating a full ray tracer

- A complicated question in general
- Basic idea: start with random points in a square
- Monte Carlo methods—see 600-level graphics courses

Roulette

Polygons:	151,752
Light Points:	23,000
Gather Points:	306
Gather/Light Pairs:	7,047,430
Cut Size:	174 (0.002%)

Tableau

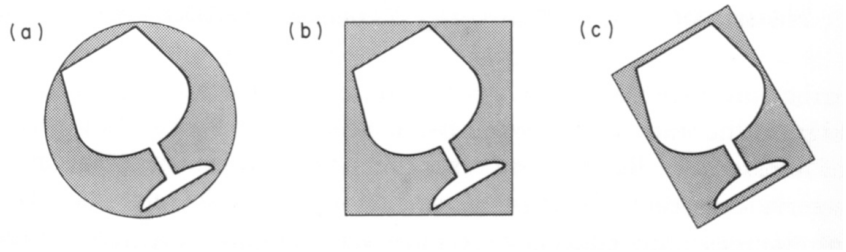
Polygons:	630,843
Light Points:	13,000
Gather Points:	180
Gather/Light Pairs:	234,000
Cut Size:	447 (0.2%)

How to make ray tracing fast?

- Ray tracing is typically slow
  - Ray tracers spend most of their time in ray-surface intersection methods
- Ways to improve speed
  - Make intersection methods more efficient
    - Yes, good idea. But only gets you so far
  - Call intersection methods fewer times
    - Intersecting every ray with every object is wasteful
    - Basic strategy: efficiently find big chunks of geometry that definitely do not intersect a ray

Bounding volumes

- Quick way to avoid intersections: bound object with a simple volume
  - Object is fully contained in the volume
  - If it doesn't hit the volume, it doesn't hit the object
  - So test bvol first, then test object if it hits



[Glassner 89, Fig 4.5]

Implementing bounding volume

- Just add new Surface subclass, "BoundedSurface"
  - Contains a bounding volume and a reference to a surface
  - Intersection method:
    - Intersect with bvol, return false for miss
    - Return surface.intersect(ray)
  - Like transformations, common to merge with group
  - This change is transparent to the renderer (only it might run faster)
- Note that all Surfaces will need to be able to supply bounding volumes for themselves

Bounding volumes

- Cost: more for hits and near misses, less for far misses
- Worth doing? It depends:
  - Cost of bvol intersection test should be small
    - Therefore use simple shapes (spheres, boxes, …)
  - Cost of object intersect test should be large
    - Bvols most useful for complex objects
  - Tightness of fit should be good
    - Loose fit leads to extra object intersections
    - Tradeoff between tightness and bvol intersection cost

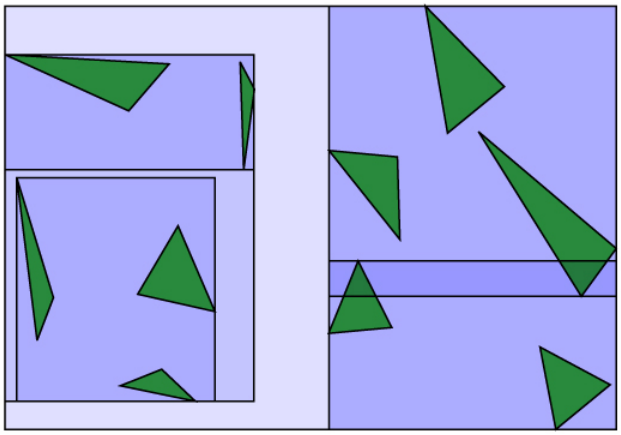
If it's worth doing, it's worth doing hierarchically!

- Bvols around objects may help
- Bvols around groups of objects will help
- Bvols around parts of complex objects will help
- Leads to the idea of using bounding volumes all the way from the whole scene down to groups of a few objects

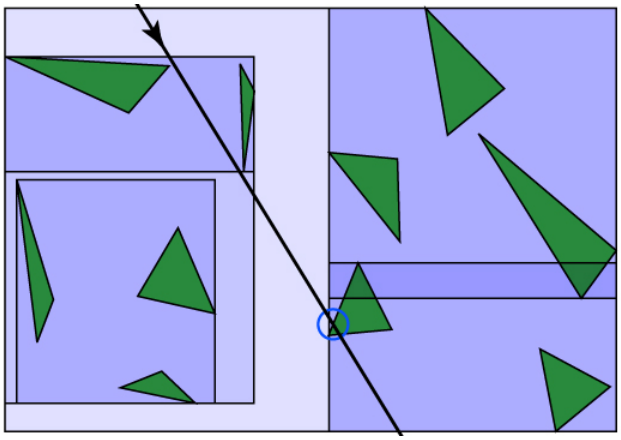
Implementing a bvol hierarchy

- A BoundedSurface can contain a list of Surfaces
- Some of those Surfaces might be more BoundedSurfaces
- Voilà! A bounding volume hierarchy
  - And it's all still transparent to the renderer

BVH construction example



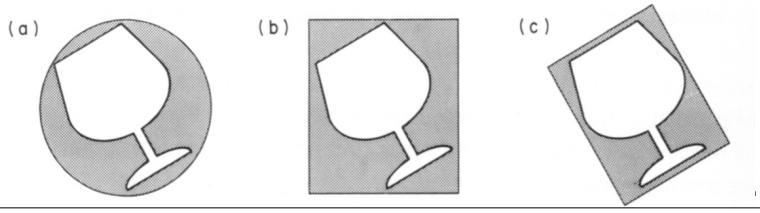
BVH ray-tracing example



- Trace rays with root node
- If intersection, trace rays with ALL children
- (If no intersection, eliminate tests with all children)

Choice of bounding volumes

- Spheres -- easy to intersect, not always so tight
- Axis-aligned bounding boxes (AABBs) -- easy to intersect, often tighter (esp. for axis-aligned models)
- Oriented bounding boxes (OBBs) -- easy to intersect (but cost of transformation), tighter for arbitrary objects
- Computing the bvols
  - For primitives -- generally pretty easy
  - For groups -- not so easy for OBBs (to do well)
  - For transformed surfaces -- not so easy for spheres



Axis aligned bounding boxes

- Probably easiest to implement
- Computing for primitives
  - Cube: duh!
  - Sphere, cylinder, etc.: pretty obvious
  - Groups or meshes: min/max of component parts
- How to intersect them
  - Treat them as an intersection of slabs (see Shirley)

Building a hierarchy

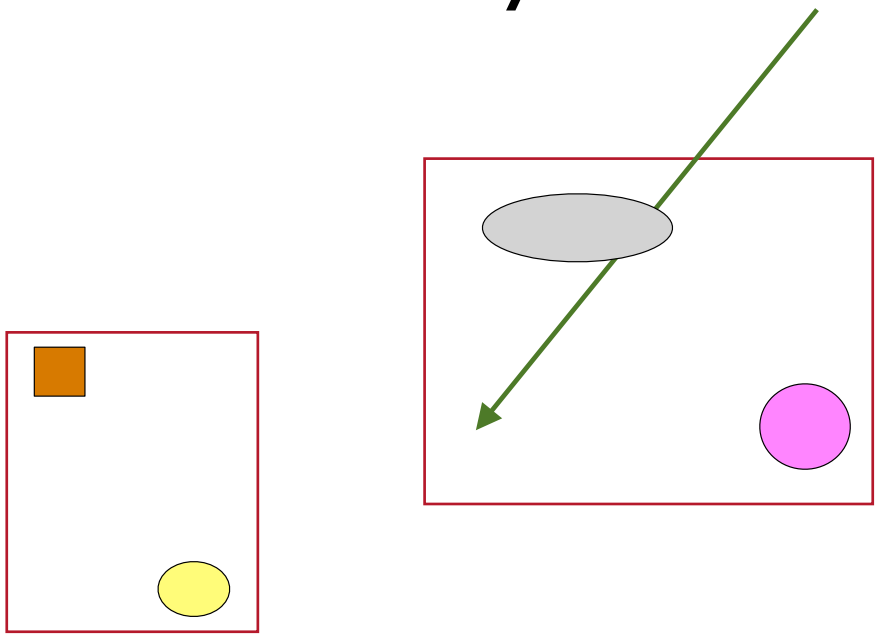
- Can do it top down or bottom up

- Top down
  - Make bbox for whole scene, then split into parts
    - Recurse on parts
    - Stop when there are just a few objects in your box
      - Or if you are too deep (say max depth = 24)

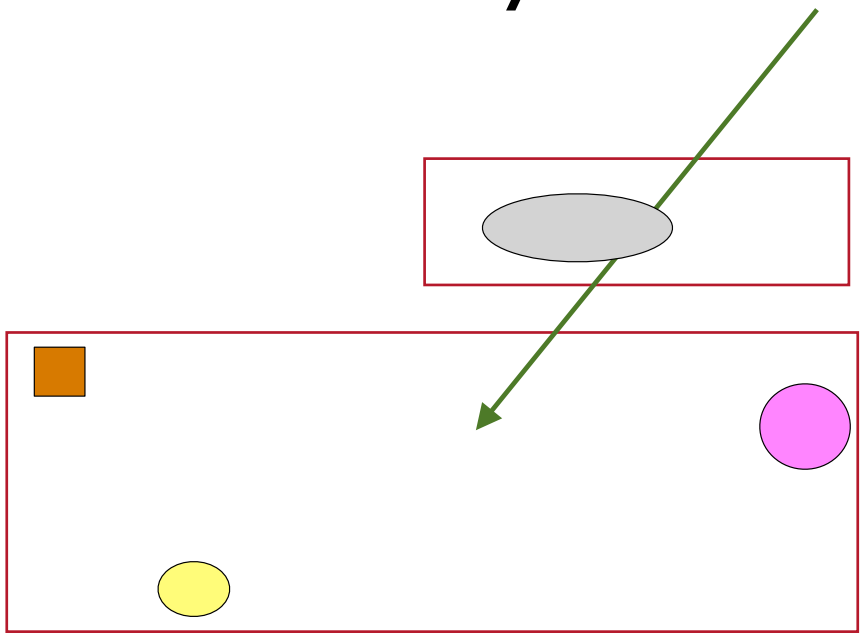
Building a hierarchy

- How to partition?
  - Practical: partition along axis
    - Center partition
      - Simple
      - Unbalanced tree
    - Median partition
      - More expensive
      - More balanced tree
- Objects that cross the median partition
  - Pick one of the sides to put the object on
  - Expand the bbox to cover that object

Construction Possibility 1



Construction Possibility 2

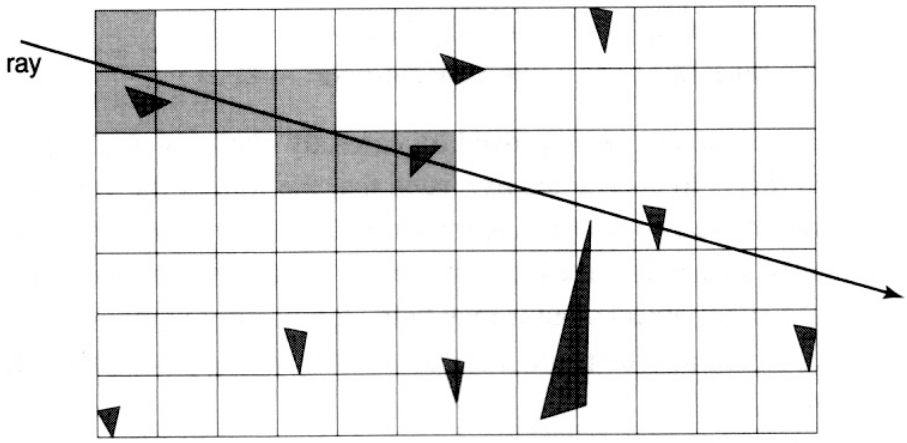


Hierarchical Data Structures

- From O(N) to O(log N)
  - Cluster objects hierarchically
  - Single intersection might eliminate cluster
- Bounding volume hierarchy
- Space subdivision
  - Octree
  - Kd-tree
  - Uniform

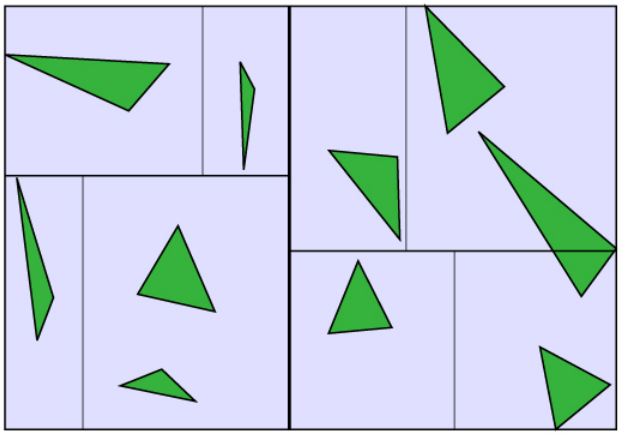
Regular space subdivision

- An entirely different approach: uniform grid of cells



Non-regular space subdivision

- *k*-d Tree
  - subdivides space, like grid
  - adaptive, like BVH



Implementing acceleration structures

- Conceptually simple to build acceleration structure into scene structure
- Better engineering decision to separate them

