

# Ray Tracing II

CS 4620 Lecture 16

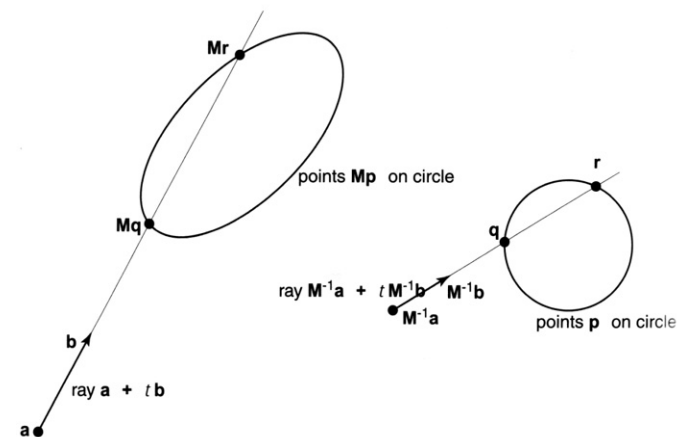
## Topics

- Transformations in ray tracing
  - Transforming objects
  - Transformation hierarchies
- Ray tracing acceleration structures
  - Bounding volumes
  - Bounding volume hierarchies
  - Uniform spatial subdivision
  - Adaptive spatial subdivision

## Transforming objects

- In modeling, we've seen the usefulness of transformations
  - How to do the same in RT?
- Take spheres as an example: want to support transformed spheres
  - Need a new Surface subclass
- Option 1: transform sphere into world coordinates
  - Write code to intersect arbitrary ellipsoids
- Option 2: transform ray into sphere's coordinates
  - Then just use existing sphere intersection routine

## Intersecting transformed objects



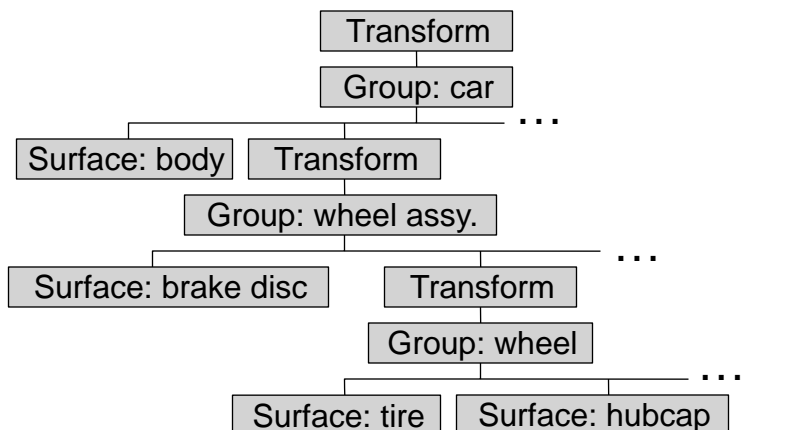
## Implementing RT transforms

- Create wrapper object “TransformedSurface”
  - Has a transform  $T$  and a reference to a surface  $S$
  - To intersect:
    - Transform ray to local coords (by inverse of  $T$ )
    - Call `surface.intersect`
    - Transform hit data back to global coords (by  $T$ )
      - Intersection point
      - Surface normal
      - Any other relevant data (maybe none)

## Groups, transforms, hierarchies

- Often it's useful to transform several objects at once
  - Add “SurfaceGroup” as a subclass of Surface
    - Has a list of surfaces
    - Returns closest intersection
      - Opportunity to move ray intersection code here to avoid duplication
- With TransformedSurface and SurfaceGroup you can put transforms below transforms
  - Voilà! A transformation hierarchy.

## A transformation hierarchy

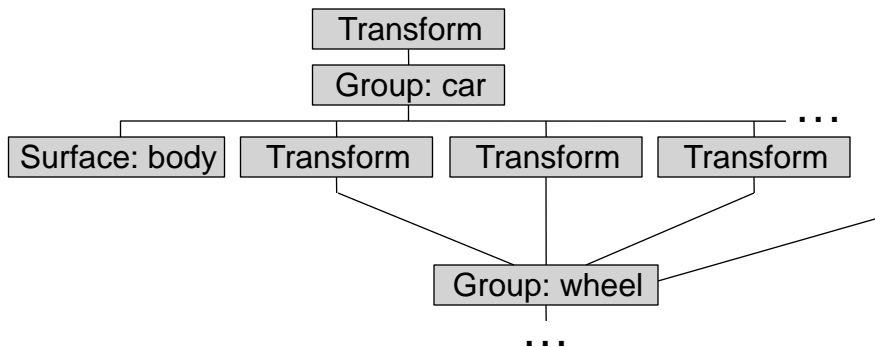


- Common optimization: merge transforms with groups
  - This is how we did it in the modeler assignment

## Instancing

- Anything worth doing is worth doing  $n$  times
- If we can transform objects, why not transform them several ways?
  - Many models have repeated subassemblies
    - Mechanical parts (wheels of car)
    - Multiple objects (chairs in classroom, ...)
  - Nothing stops you from creating two TransformedSurface objects that reference the same Surface
    - Allowing this makes the transformation tree into a DAG
      - (directed acyclic graph)
    - Mostly this is transparent to the renderer

## Hierarchy with instancing



## Hierarchies and performance

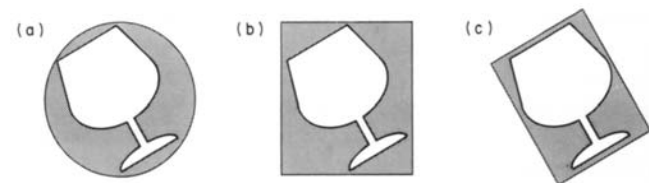
- Transforming rays is expensive
  - minimize tree depth: flatten on input
    - push all transformations toward leaves
    - triangle meshes may do best to stay as group
      - transform ray once, intersect with mesh
  - internal group nodes still required for instancing
    - can't push two transforms down to same child!

## Ray tracing acceleration

- Ray tracing is slow. This is bad!
  - 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



## 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

## 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

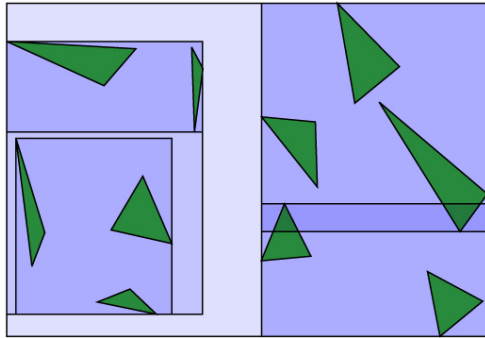
## If it's worth doing, it's worth doing hierarchically!

- 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

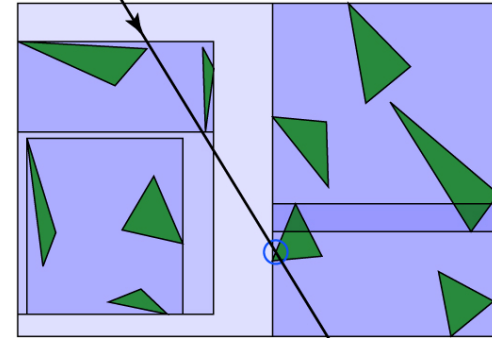
## Implementing a bvol hierarchy

- 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



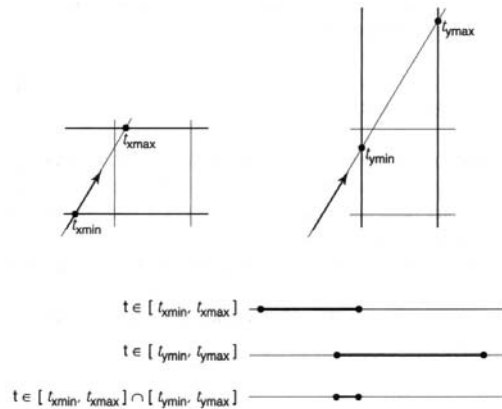
## Choice of bounding volumes

- 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
- AABBs for transformed surface
  - Easy to do conservatively: bbox of the 8 corners of the bbox of the untransformed surface
- How to intersect them
  - Treat them as an intersection of slabs (see Shirley)

## Intersecting boxes



## Building a hierarchy

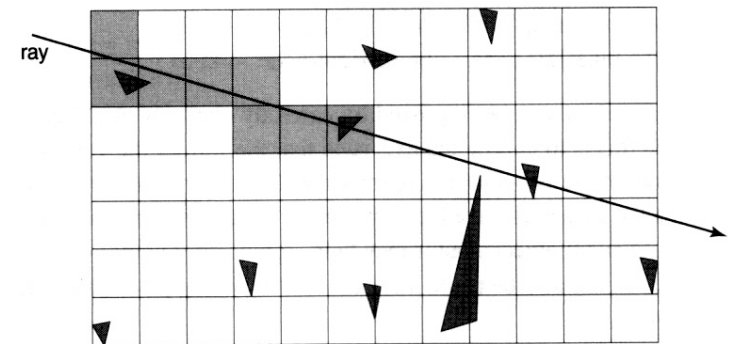
- Usually do it top-down
- Make bbox for whole scene, then split into (maybe 2) parts
  - Recurse on parts
  - Stop when there are just a few objects in your box

## Building a hierarchy

- How to partition?
  - Ideal: clusters
  - Practical: partition along axis
    - Median partition
      - More expensive
      - More balanced tree
    - Center partition
      - Less expensive, simpler
      - Unbalanced tree, but that may actually be better

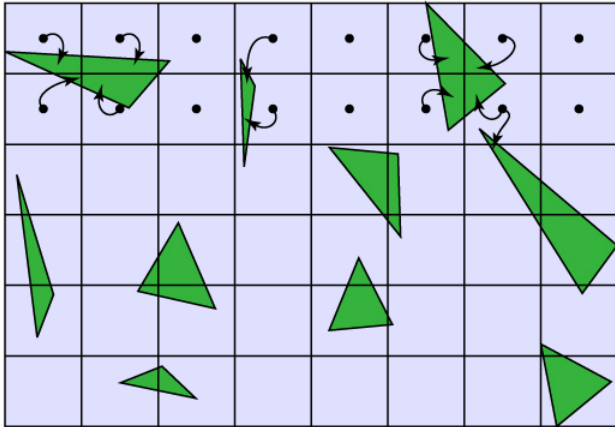
## Regular space subdivision

- An entirely different approach: uniform grid of cells

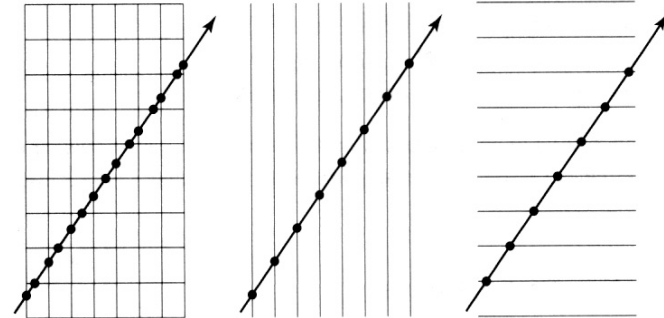


## Regular grid example

- Grid divides space, not objects

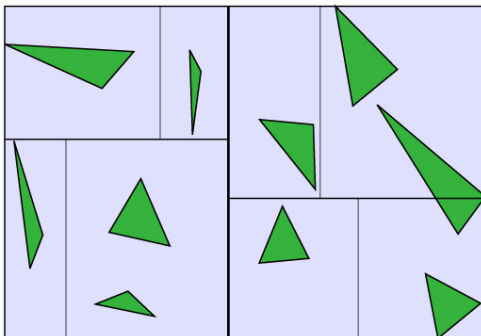


## Traversing a regular grid



## 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