More Ray Tracing

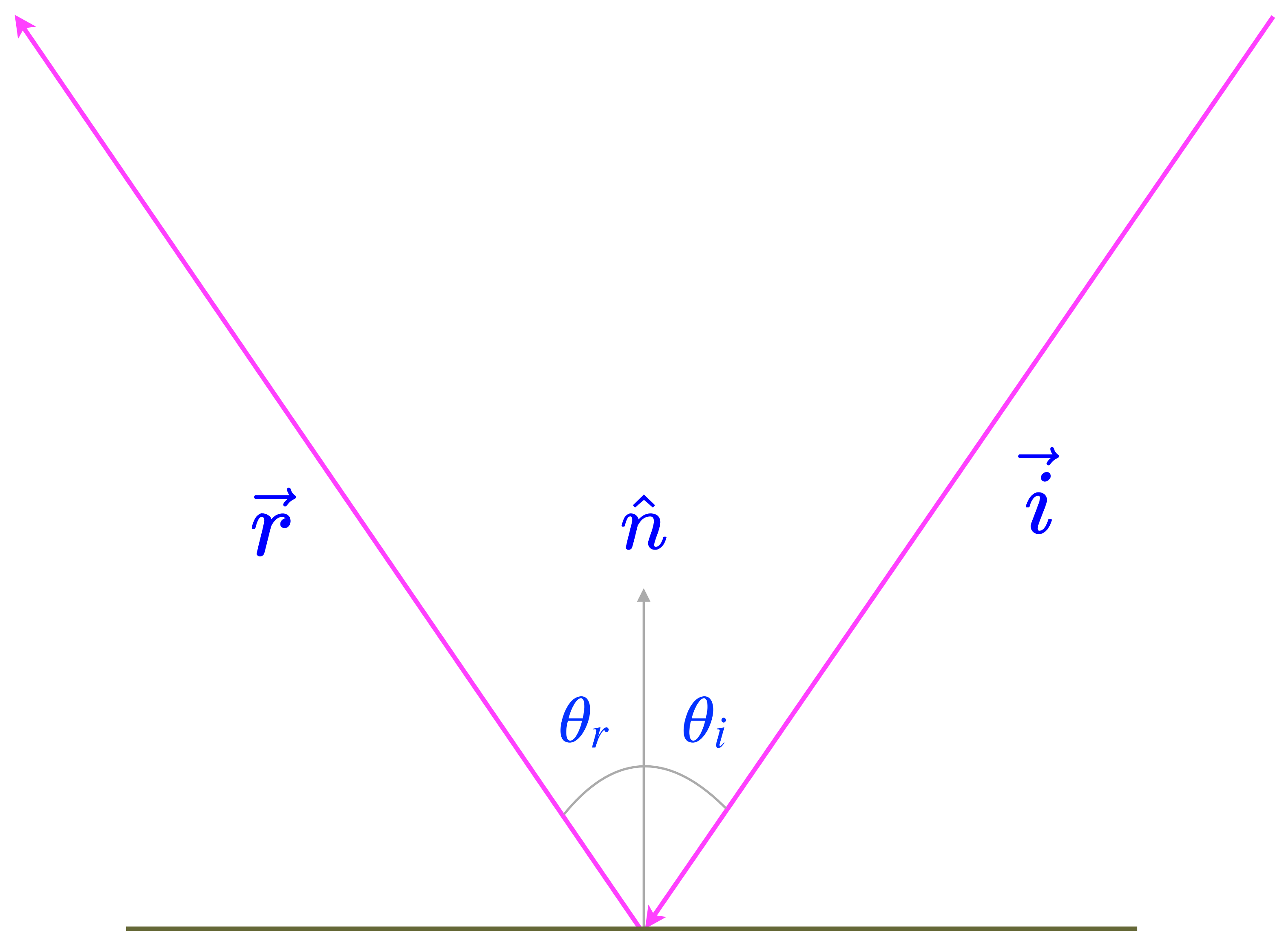
CS 385 - Class 28
5 May 2022

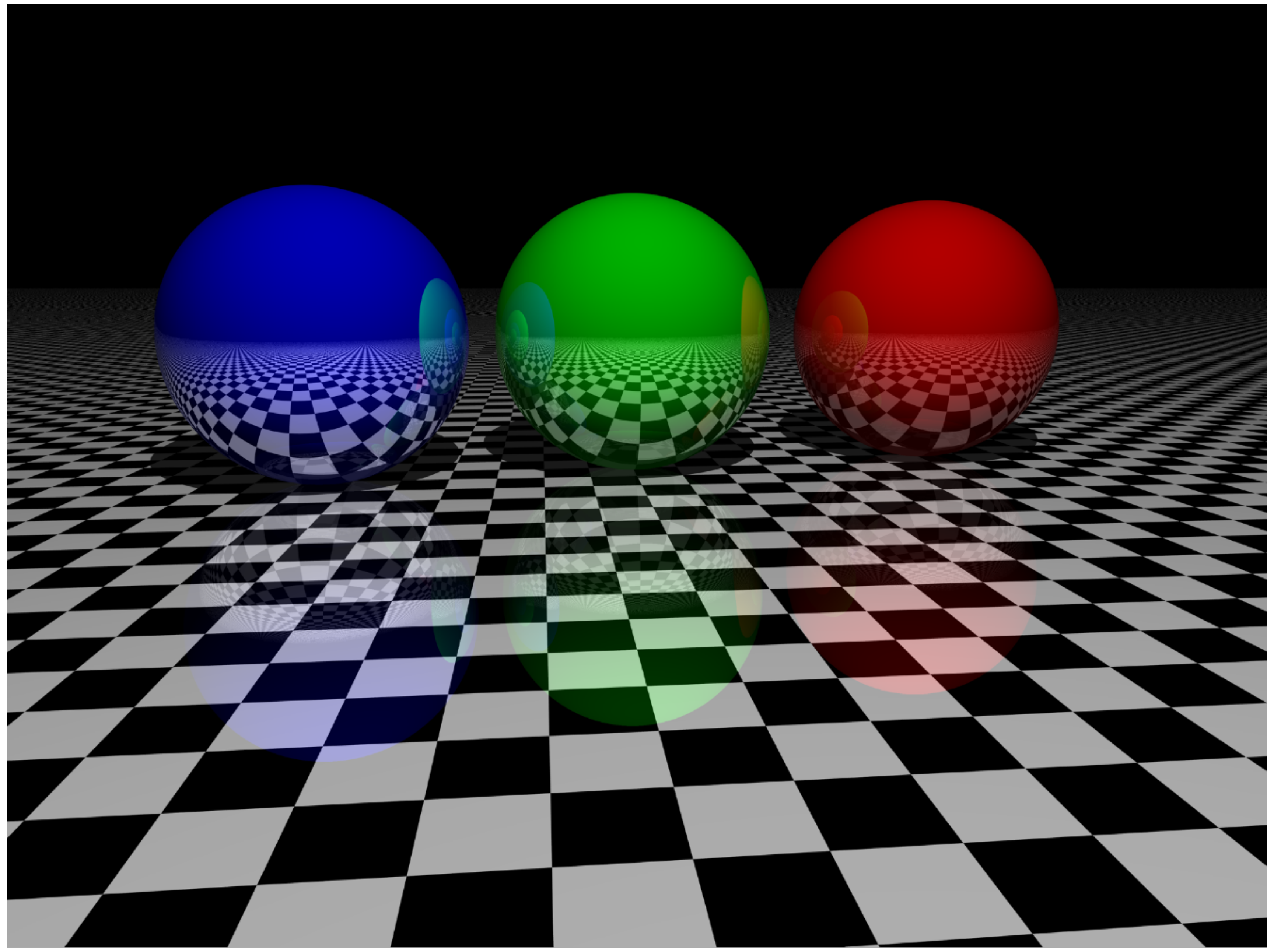
Reflections, Refractions, and Shadows

What Happens when We Hit Something?

- If it's reflective, we need to *reflect* the vector around the *surface normal*
- Start a new ray:
  - intersection point becomes new $\vec{r}_0$
  - reflected, normalized incident vector becomes new ray direction $\hat{r}_d$

$$\vec{r} = \vec{i} - 2(\vec{i} \cdot \hat{n})\hat{n}$$





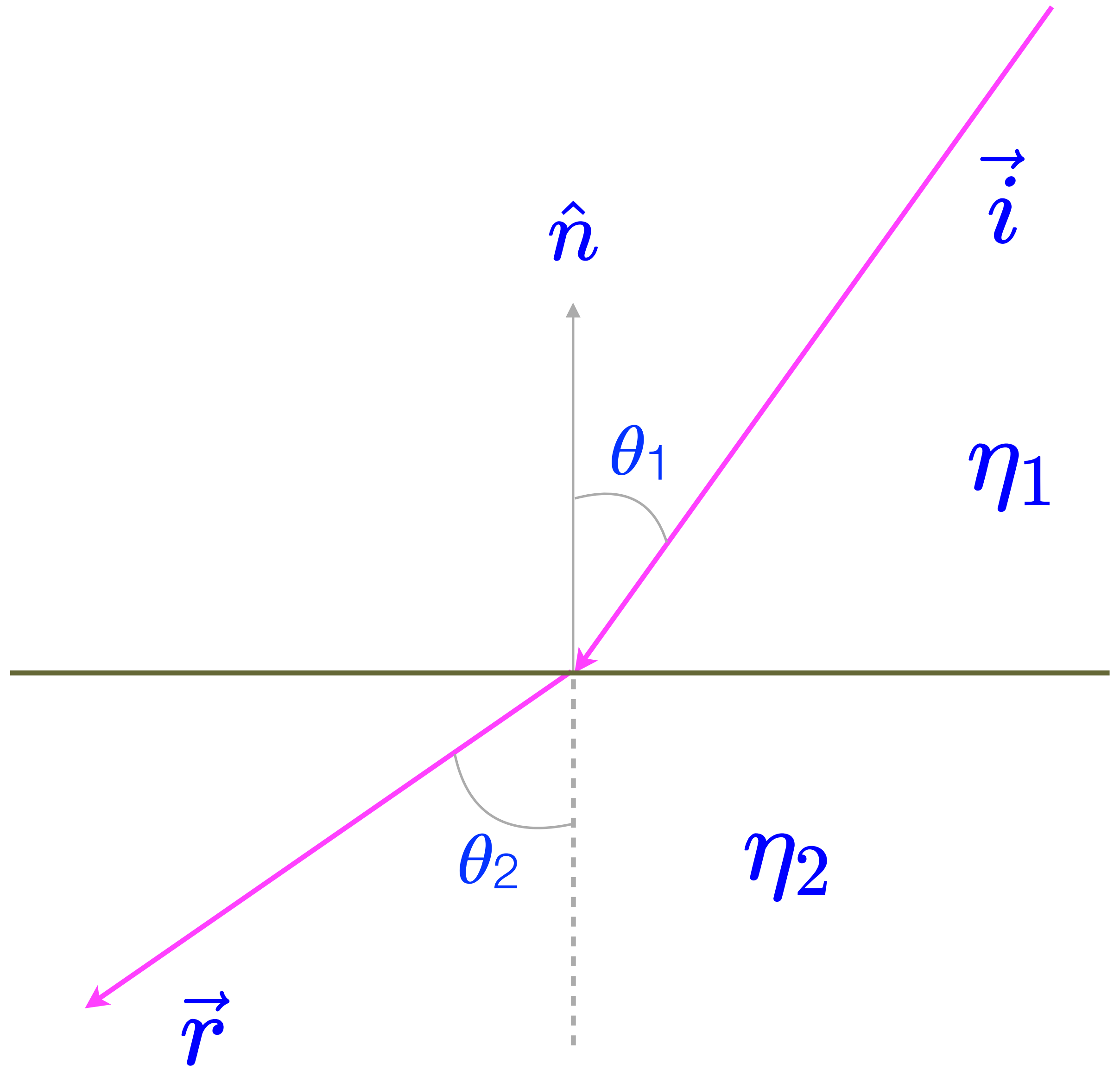
What Happens when We Hit Something?

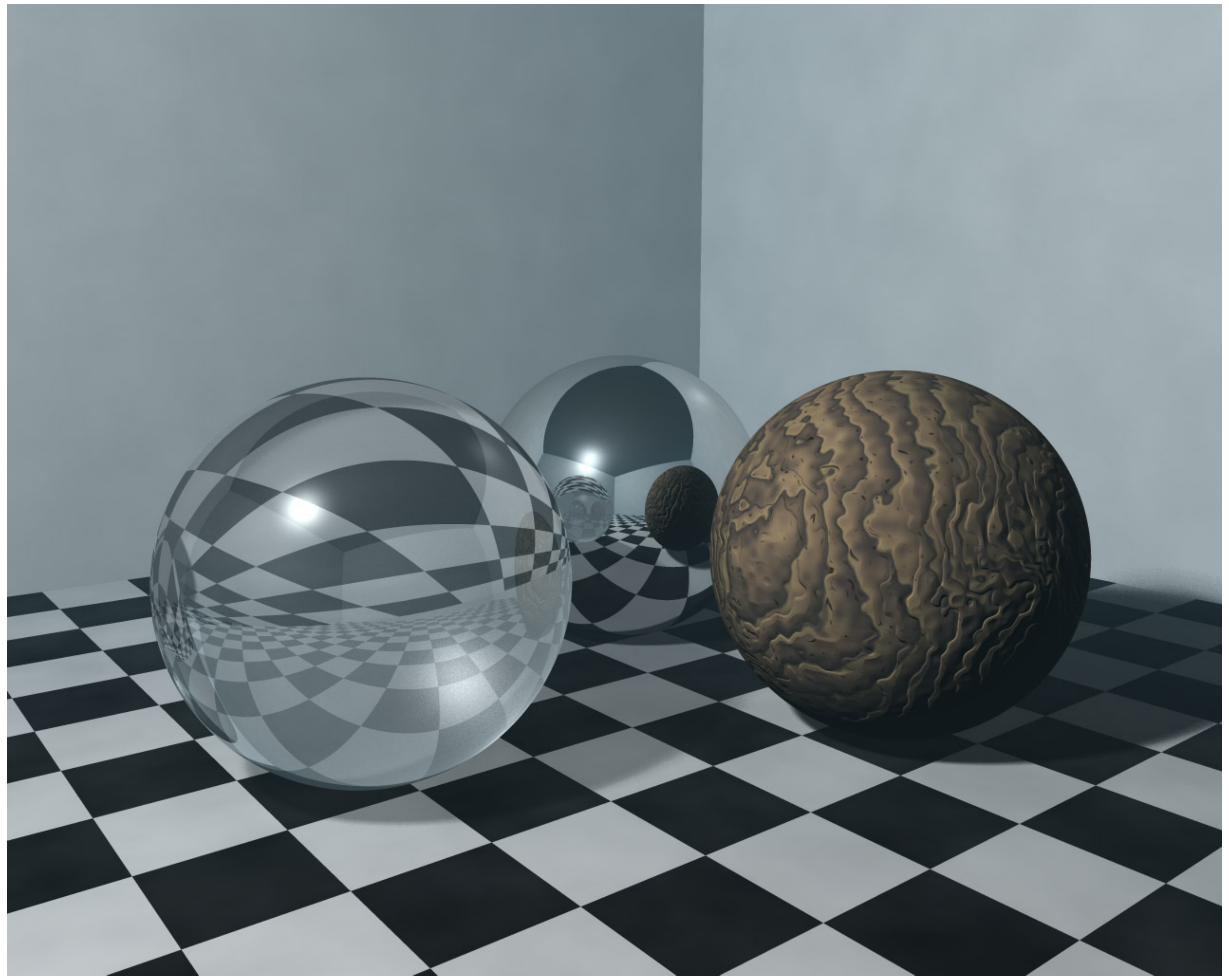
- However, if it transmits light, it's *refractive*
- Similar math, same operation
  - employ *Snell's law*

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$$

- Start a new ray:
  - intersection point becomes new $\vec{r}_0$
  - refracted, normalized incident vector becomes new ray direction $\hat{r}_d$

$$\vec{r} = \left(\frac{\eta_1}{\eta_2}\right)\vec{i} + \left(\frac{\eta_1}{\eta_2}\cos\theta_1 - \cos\theta_2\right)\hat{n}$$





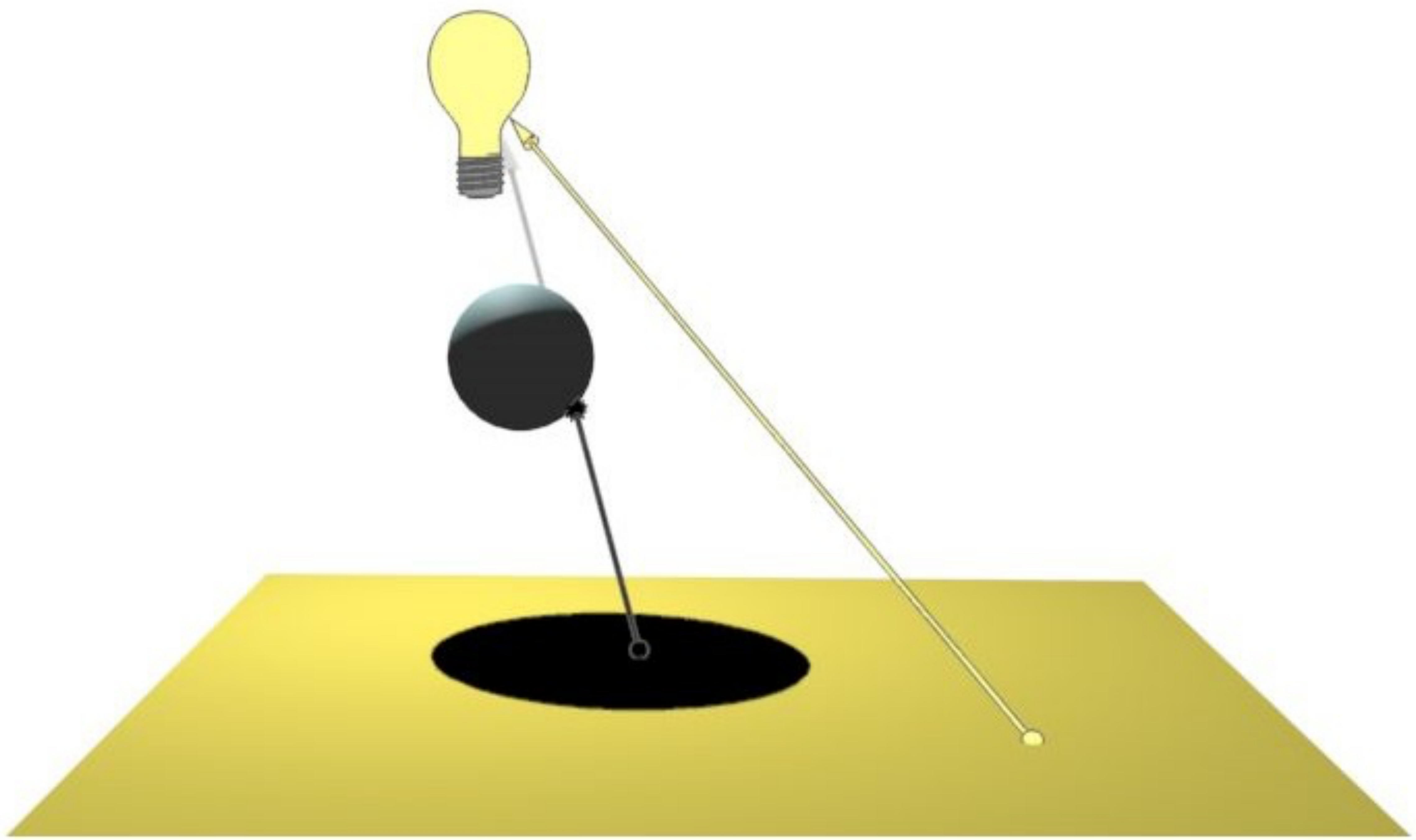
When Do We Stop Bouncing Around?

- There are three possible outcomes for a ray:

ray action	reaction
hits a light source	return the lights color, and unwind hit stack
goes off to ∞ (you know it doesn't hit anything else)	return no light (e.g., black or background color) and unwind stack
infinite reflections between two perfect reflectors	cry! (and then rewrite your renderer to have a fixed number of iterations)

Shadows are “free” in Ray Tracing

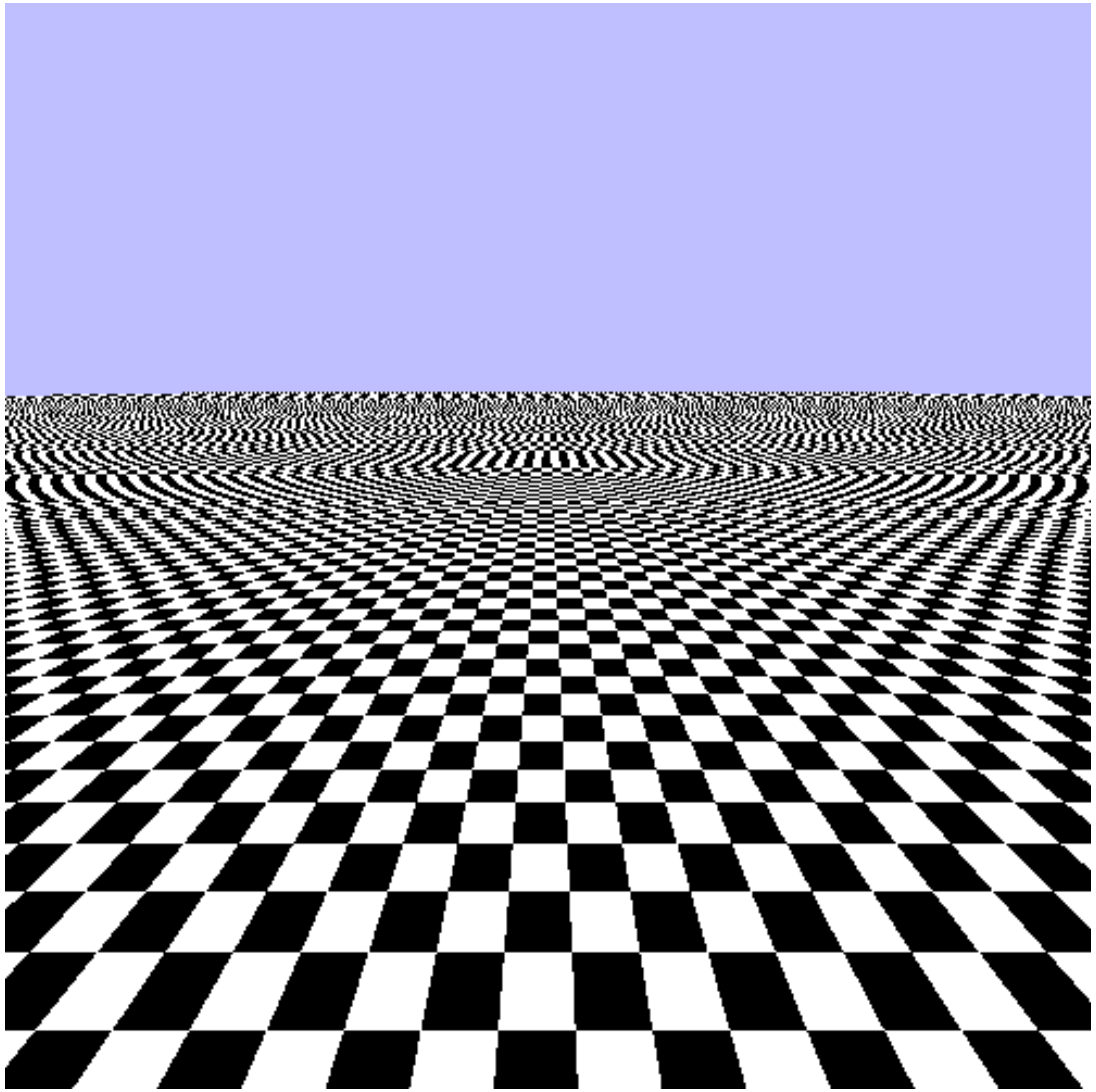
- If a ray hits another object before hitting a light, it’s in shadow



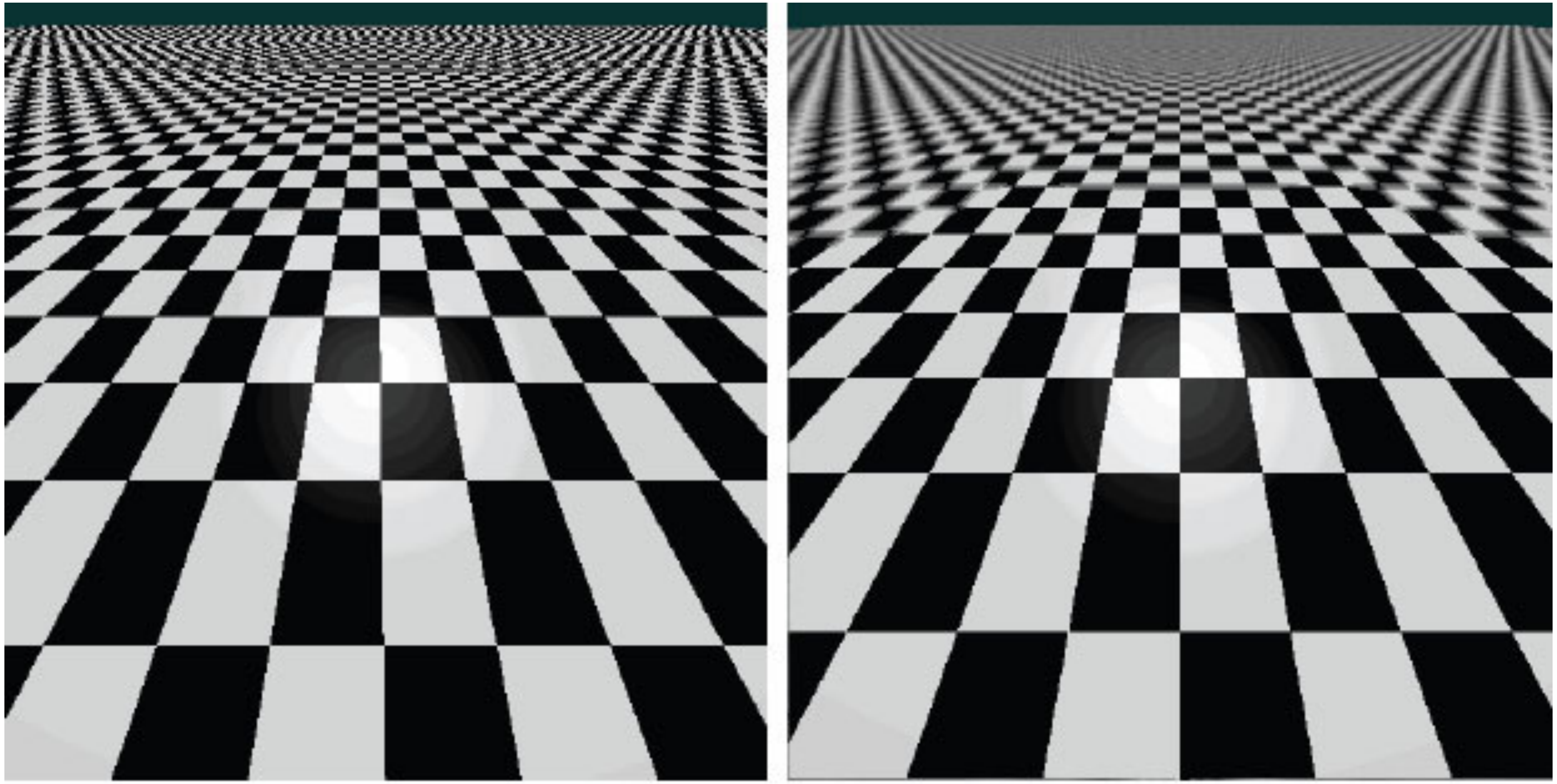
Aliasing & Anti-aliasing

Enter the Jaggies

- Aliasing occurs when you try to change colors too fast across pixels
  - "a high-frequency color change"
- For the pixels at the horizon, the colors of the checkerboard are changing multiple times *per pixel*
- In order to avoid aliasing, you need to *sample at twice the Nyquist Limit*



Increasing the sampling rate

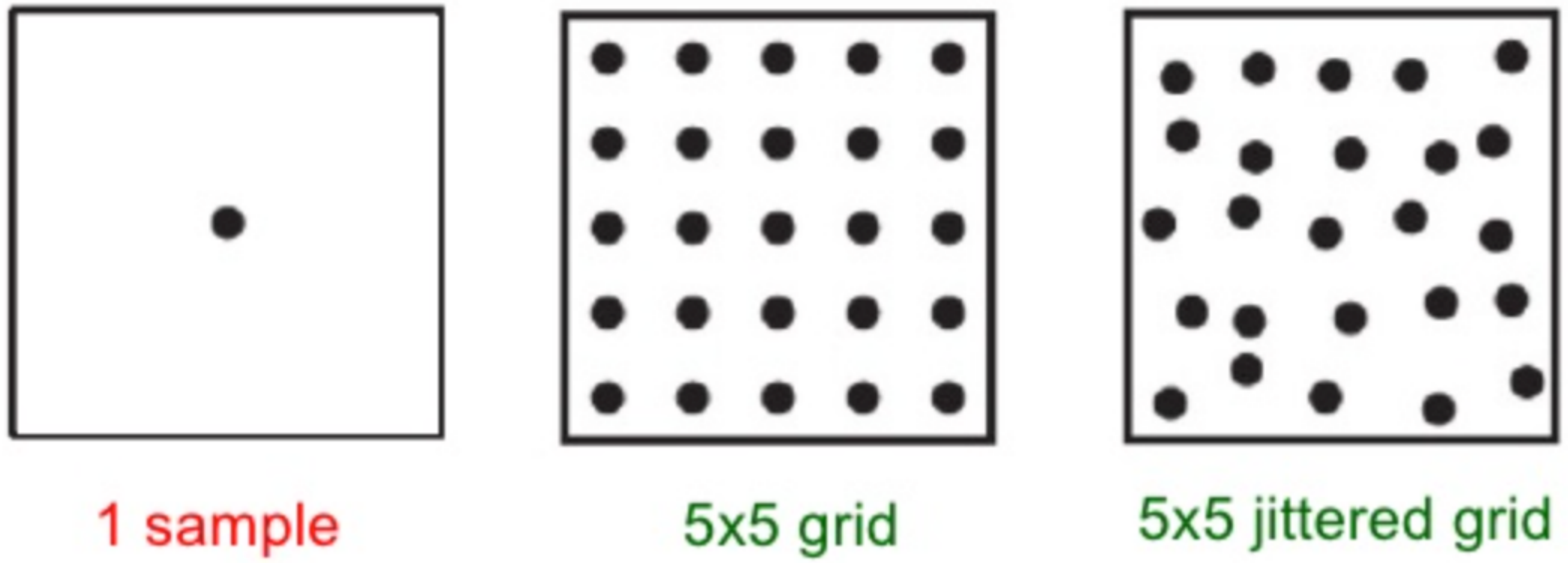


aliasing effects

anti-aliasing by over-sampling

Multi-sampling

- Trace multiple rays per pixel
- *Uniform sampling* across a pixel
  - simple, but may not resolve the problem
- *Jittered sampling*
  - randomly select sub-pixel position
  - better, but samples may tend to group
- *Poisson Disk*
  - again, random sub-pixel selection
  - guarantee that no two samples are too close



Scene Management

Ray Tracing is really a Database Search Problem

- Sure, at the end of it there's an image, but there's a lot of work that goes into each pixel
- Recall the algorithm:

```
foreach ( pixel in the frame ) {
  initialize( ray );
  foreach ( primitive in the scene ) {
    intersect( ray, primitive );
  }
}
```

- and that's not quite the whole story
  - `intersect()` can have a lot of recursions

Performance Considerations

- Consider:
  - 3840 ×2160 image (your standard 4K image)
  - 5 rays / pixel
  - 1000 objects (with a modest 100 triangles/object)
  - max of only three levels of recursion
- That’s 12,441,600,000,000 (12.4×10^{15} or 12.4 *peta*) intersection tests per frame
  - at 1,000,000 intersections/second that’s 3,456 hours (144 days) per frame
    - for a 1.5 hour movie @ 24 frames/second, that's 51,129.9 years
      - Pixar's probably not going to wait that long

Scene Management

- Testing each and every triangle is likely very wasteful
- Can we somehow segment the objects to easily skip objects we know can't been intersected?
  - yeah, we can probably do better than the naive approach
  - in fact, there are multiple ways to approach this problem

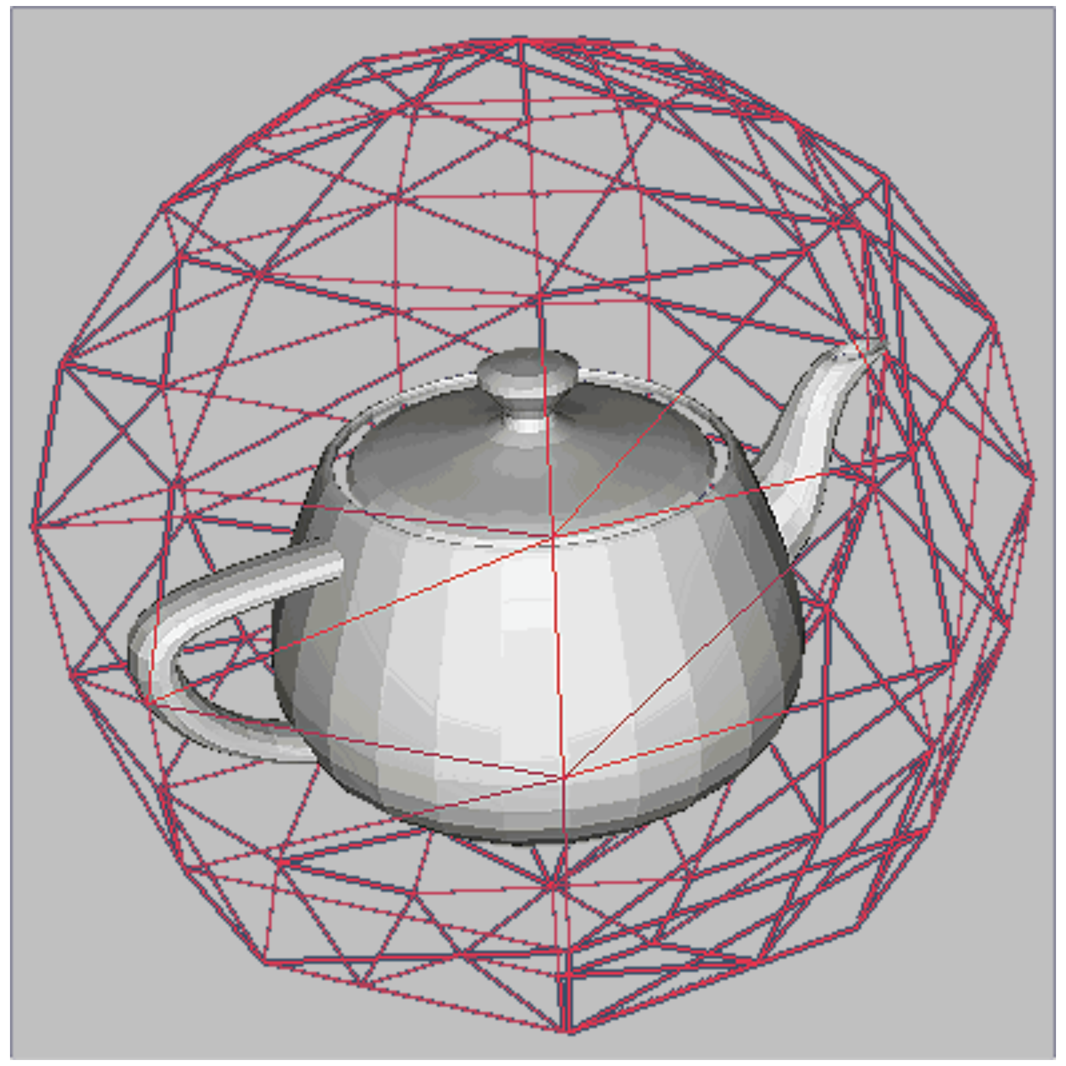
Bounding Spheres

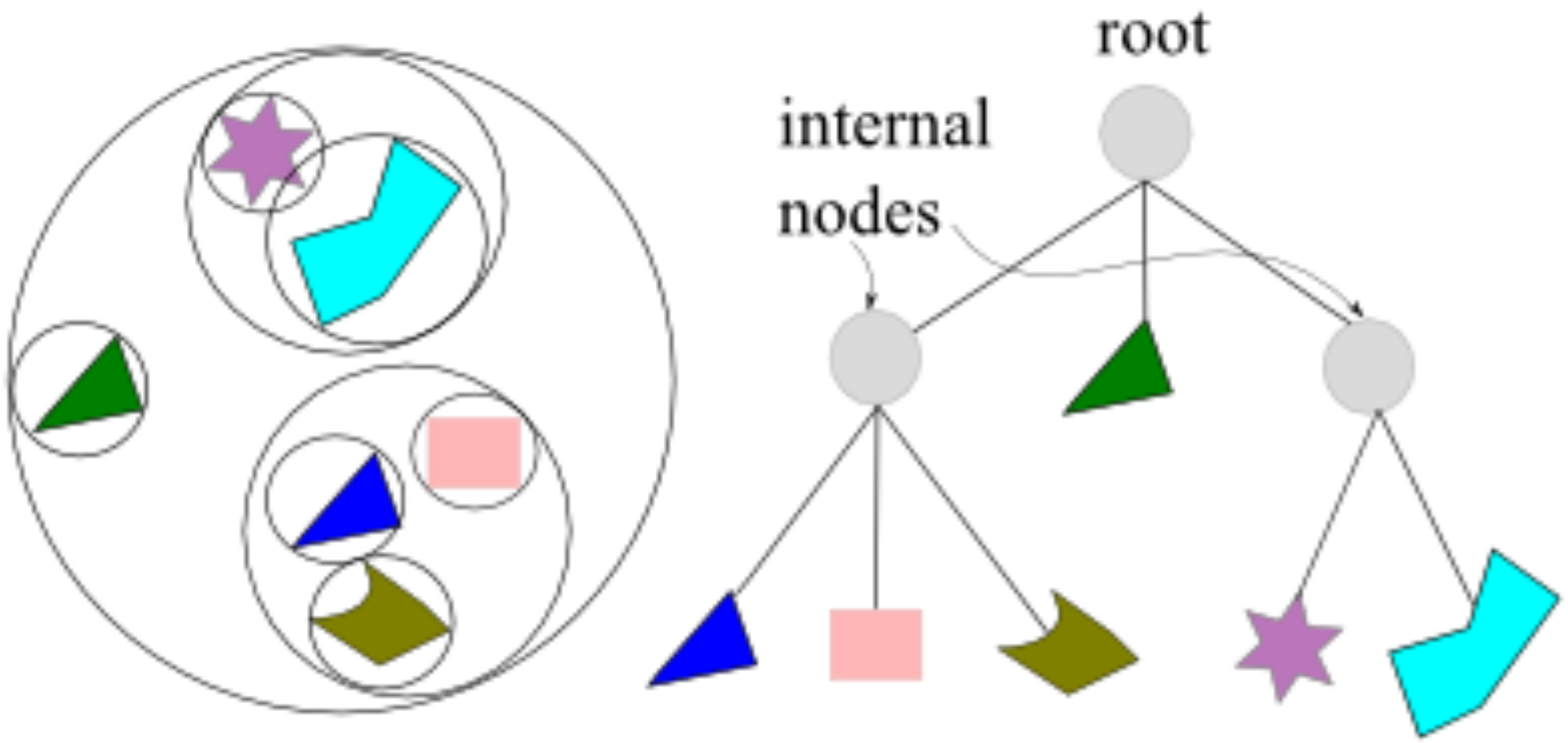
How about Using Simple Bounding Shapes?

- For scene management, we like simple:
  - Spheres
    - easy to intersect
    - most things kind of approximate a sphere
      - sure, there's some wasted space in the volume, but the object can be in any orientation
- We can make a *bounding volume hierarchy* (BVH)
  - we'll use spheres for our first attempt, but we'll see other versions soon

Bounding Sphere Hierarchy

- Enclose each object in a tightly-bounding sphere
  - center of sphere coincident with centroid of all the object's vertices
  - diameter of the sphere is maximum separation between any two vertices
- Create a hierarchy based a user-defined criteria, e.g.,
  - locally group objects based on their location
  - logically group them based on some search criteria
    - group all of the players into a higher-level bounding sphere





Creating a Bounding Sphere Hierarchy

- For each "object" (which may just be a connected set of primitives)
  - **find the maximum separation between any two vertices**
  - set the bounding sphere's
    - center as the midpoint of the separation line
    - radius to half the distance between the two maximal points
- Next group (either logically or spatially) sets of spheres into a larger sphere
  - here we'll find the maximum separation between any two spheres in the set
  - define the diameter of the bounding sphere as the separation of the two sphere's centers, plus the radius of each of the spheres
  - define the bounding sphere's center as the midpoint of that line

```
foreach ( object ) {
  vec3 object.vertices = { ... };

  vec3 min = object.vertices[0];
  vec3 max = object.vertices[0];

  for ( var i = 1; i < vertices.length(); ++i ) {
    var v = object.vertices[i];
    if ( v.x < min.x ) min.x = v.x;
    if ( v.x > max.x ) max.x = v.x;
    if ( v.y < min.y ) min.y = v.y;
    if ( v.y > max.y ) max.y = v.y;
    if ( v.z < min.z ) min.z = v.z;
    if ( v.z > max.z ) max.z = v.z;
  }

  sphere[j].center = 0.5 * (min + max);
  sphere[j].radius = 0.5 * length( max - min );
}
```

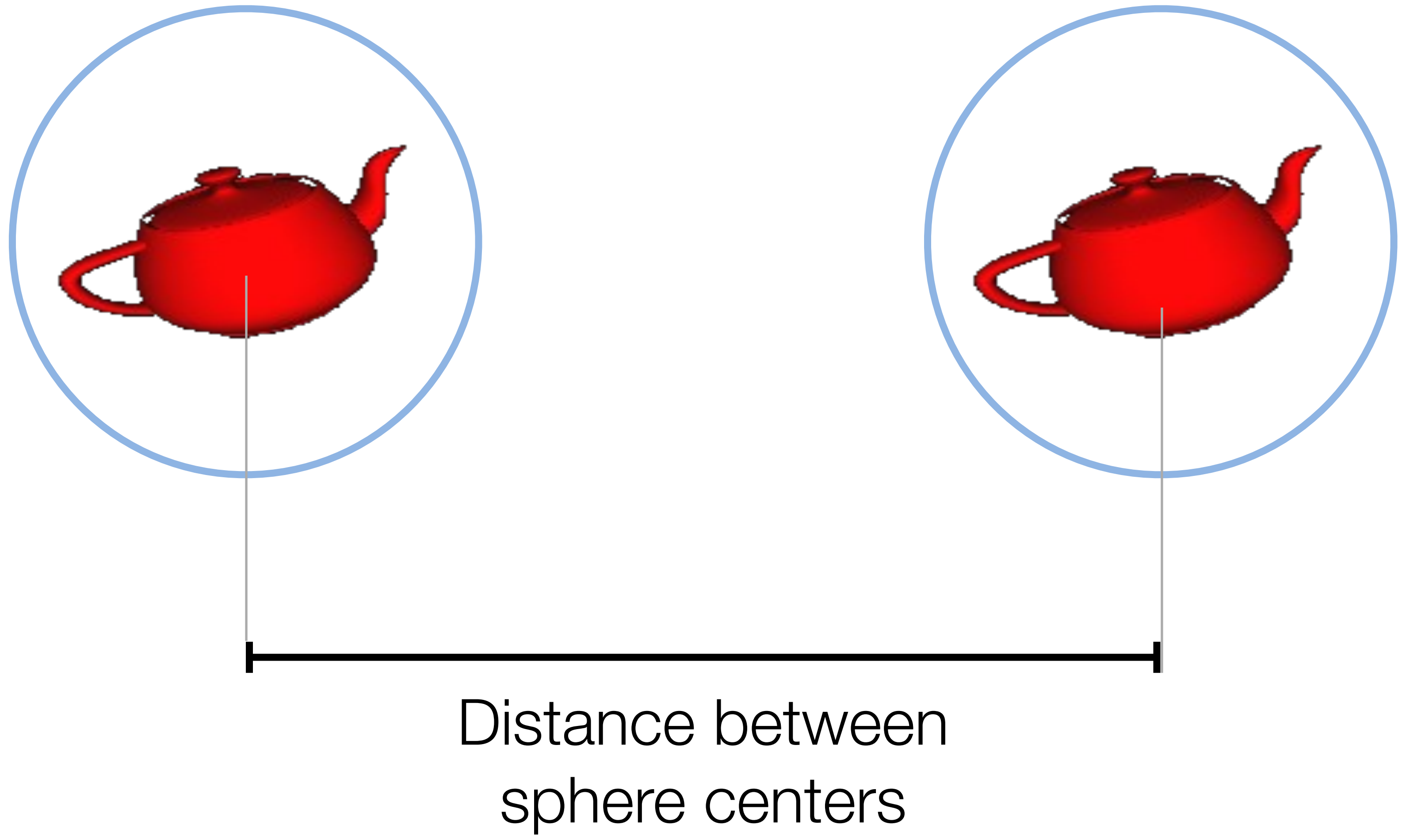
Creating a Bounding Sphere Hierarchy

- For each "object" (which may just be a connected set of primitives)
  - find the maximum separation between any two vertices
  - **set the bounding sphere's**
    - **center as the midpoint of the separation line**
    - **radius to half the distance between the two maximal points**
- Next group (either logically or spatially) sets of spheres into a larger sphere
  - here we'll find the maximum separation between any two spheres in the set
  - define the diameter of the bounding sphere as the separation of the two sphere's centers, plus the radius of each of the spheres
  - define the bounding sphere's center as the midpoint of that line

```
foreach ( object ) {
  vec3 object.vertices = { ... };

  vec3 min = object.vertices[0];
  vec3 max = object.vertices[0];

  for ( var i = 1; i < vertices.length(); ++i ) {
    var v = object.vertices[i];
    if ( v.x < min.x ) min.x = v.x;
    if ( v.x > max.x ) max.x = v.x;
    if ( v.y < min.y ) min.y = v.y;
    if ( v.y > max.y ) max.y = v.y;
    if ( v.z < min.z ) min.z = v.z;
    if ( v.z > max.z ) max.z = v.z;
  }

  sphere[j].center = 0.5 * (min + max);
  sphere[j].radius = 0.5 * length( max - min );
}
```

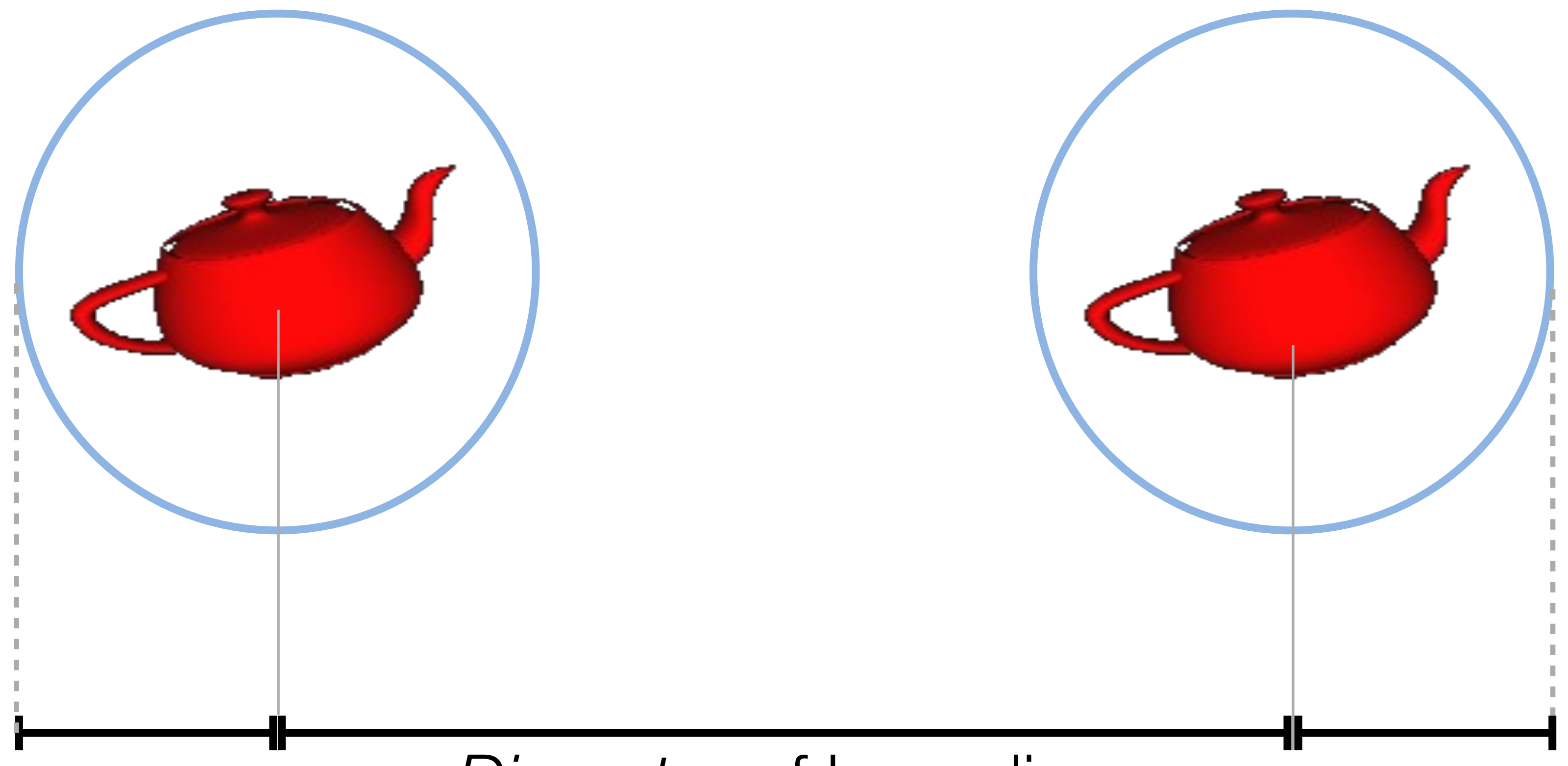
Creating a Bounding Sphere Hierarchy

- For each "object" (which may just be a connected set of primitives)
  - find the maximum separation between any two vertices
  - set the bounding sphere's
    - center as the midpoint of the separation line
    - radius to half the distance between the two maximal points
- Next group (either logically or spatially) sets of spheres into a larger sphere
  - **here we'll find the maximum separation between any two spheres in the set**
  - define the diameter of the bounding sphere as the separation of the two sphere's centers, plus the radius of each of the spheres
  - define the bounding sphere's center as the midpoint of that line



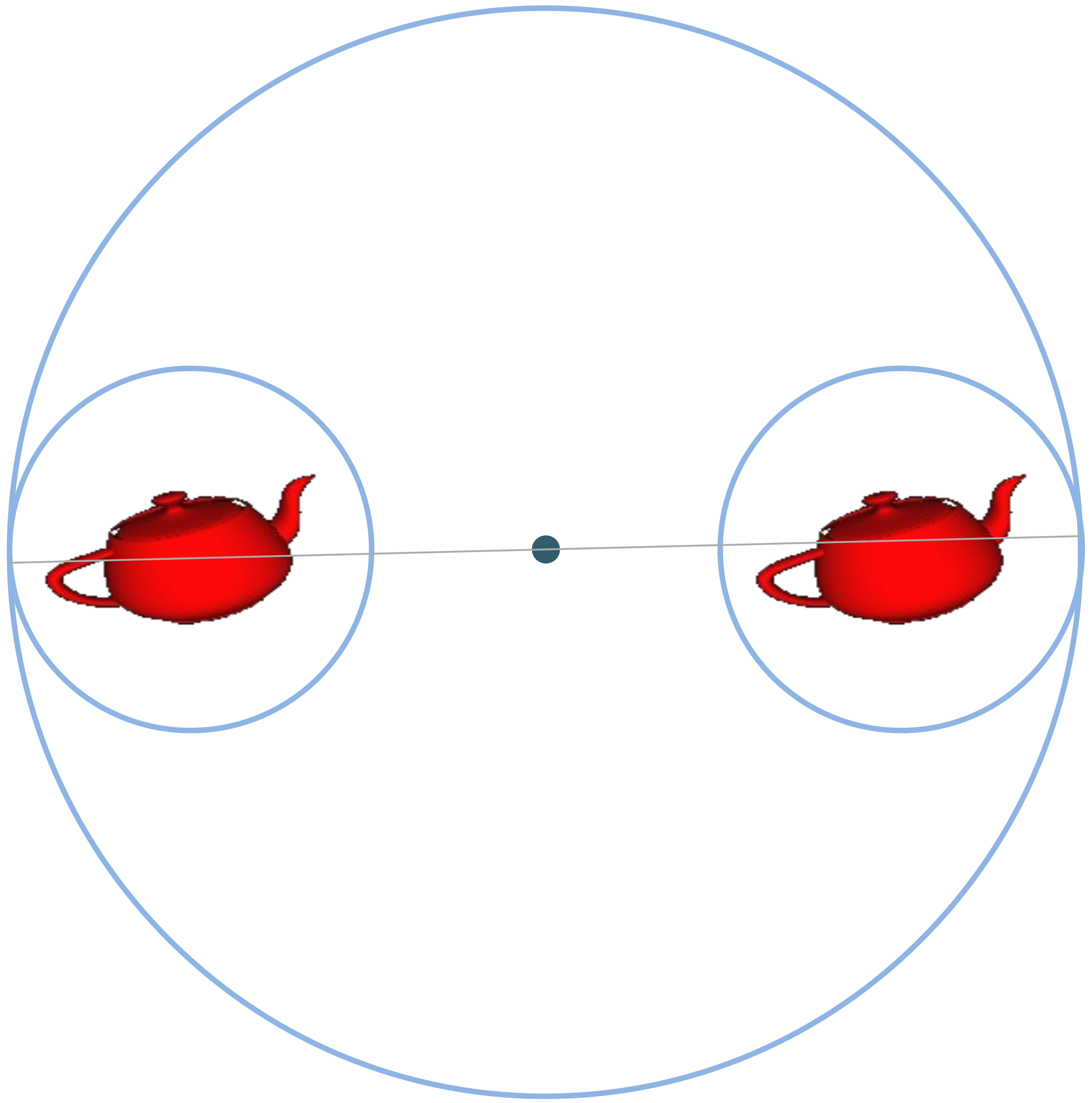
Creating a Bounding Sphere Hierarchy

- For each "object" (which may just be a connected set of primitives)
  - find the maximum separation between any two vertices
  - set the bounding sphere's
    - center as the midpoint of the separation line
    - radius to half the distance between the two maximal points
- Next group (either logically or spatially) sets of spheres into a larger sphere
  - here we'll find the maximum separation between any two spheres in the set
  - **define the diameter of the bounding sphere as the separation of the two sphere's centers, plus the radius of each of the spheres**
  - define the bounding sphere's center as the midpoint of that line



Creating a Bounding Sphere Hierarchy

- For each "object" (which may just be a connected set of primitives)
  - find the maximum separation between any two vertices
  - set the bounding sphere's
    - center as the midpoint of the separation line
    - radius to half the distance between the two maximal points
- Next group (either logically or spatially) sets of spheres into a larger sphere
  - here we'll find the maximum separation between any two spheres in the set
  - define the diameter of the bounding sphere as the separation of the two sphere's centers, plus the radius of each of the spheres
  - **define the bounding sphere's center as the midpoint of that line**



Bounding Boxes

Types of Bounding Boxes

Axis-aligned

- simply find the maximal extents in each coordinate direction
- (or in English, find the min & max values in x, y, and z)
- Simple
  - easy representation and intersections
  - however, may not be the tightest fit
    - think about a thin cylinder going from corner-to-corner

Object-aligned

- find the maximum extent along the *primary axis*
    - other dimensions perpendicular to that vector
  - better object *fitting*
  - worse intersections
    - intersecting two arbitrarily orientated boxes yields a *polytope*
      - good for trivia contests, not so much for graphics

Bounding Boxes (cont'd)

- Digression — *3D Ellipsoids*
    - they're the cat's meow
    - better for tight bounds, and can handle arbitrary orientations
    - however, intersecting two arbitrarily-oriented 3D ellipsoids is an unsolved problem (like smart people haven't figured this out yet)
        - there's probably a cash prize for solving the problem ... go crazy!
    - So, this ain't gonna work like we hoped

What do We Do?

- While all of these approaches are tenable, perhaps we can do better
- The real problem is knowing when to intersect some geometry to a ray
- This leads to thinking about ways to isolate geometry in more useful ways
  - *space partitioning* is the current thinking on how to do this

Space Partition Methods

Segregating Space

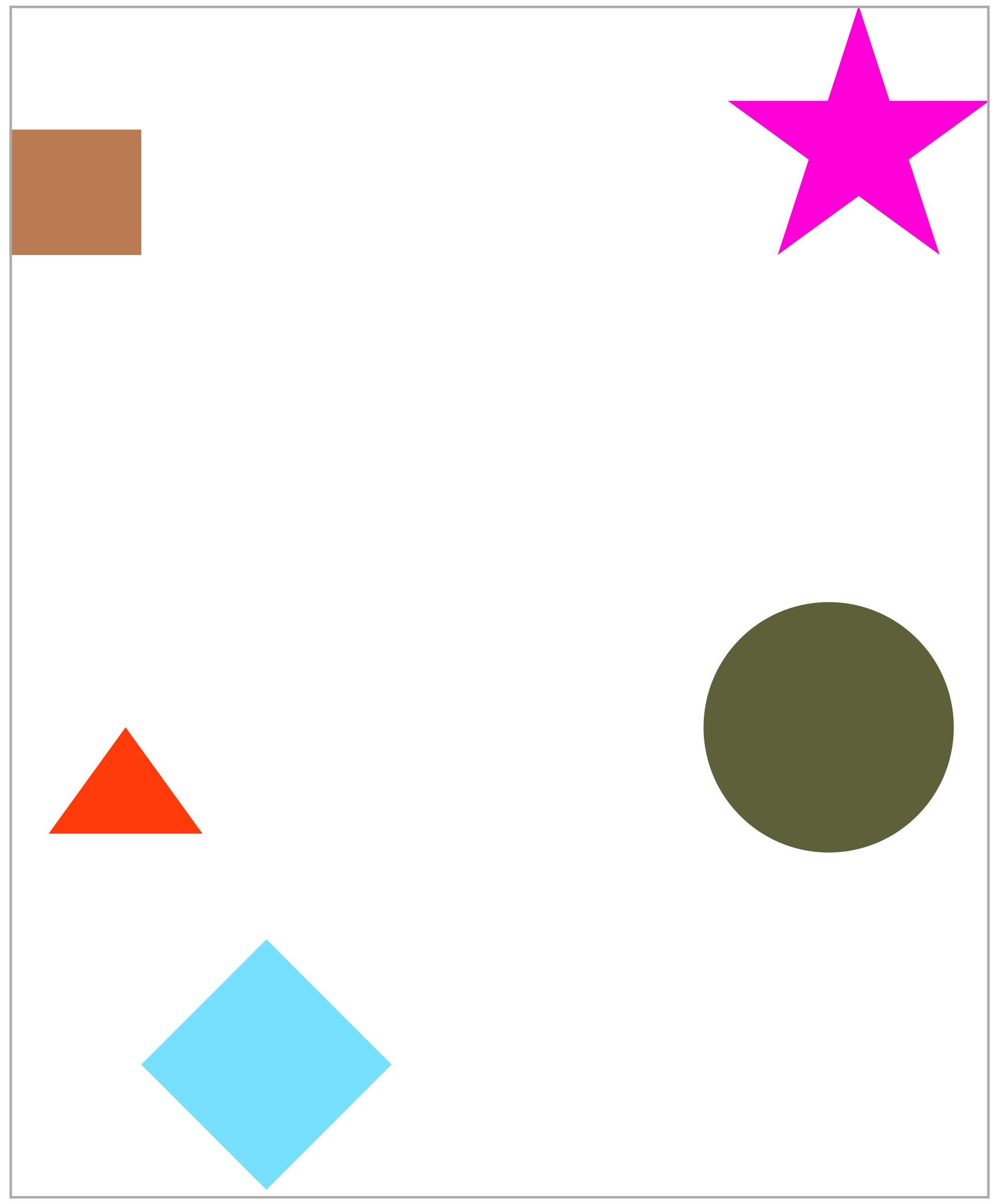
- We can use another approach and divide space into regions
- All of these methods rely on breaking a region into subspaces using a *half-plane* (which is just a plane that separates to spaces)
- This is when the point-normal form of a plane equation comes in handy
- We'll look at two methods:
  - uniformly partition a volume
  - partition a volume based on its occupancy

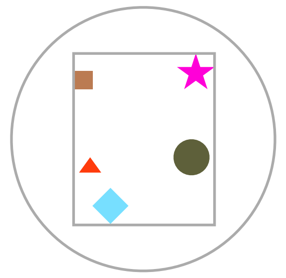
Binary Space Partition

- Split space down the middle
- works in any dimension:
  - for 2D, we create a *quadtree*
  - in 3D, we create an *octree*

Quadtrees

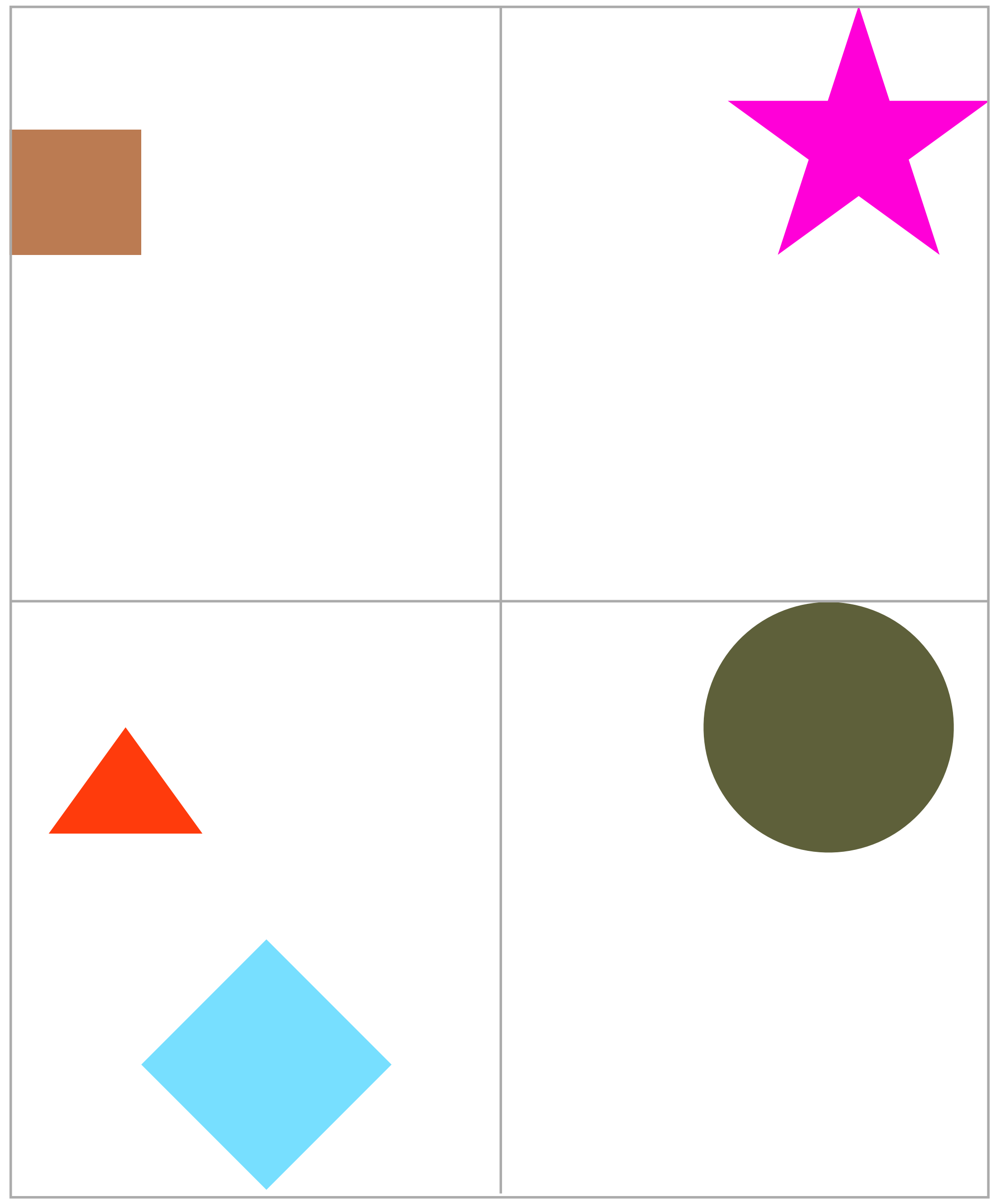
- Assume we know the extents of all our objects in 2D
- Divide space evenly in a dimension
- Recurse until we have the granularity we feel is appropriate

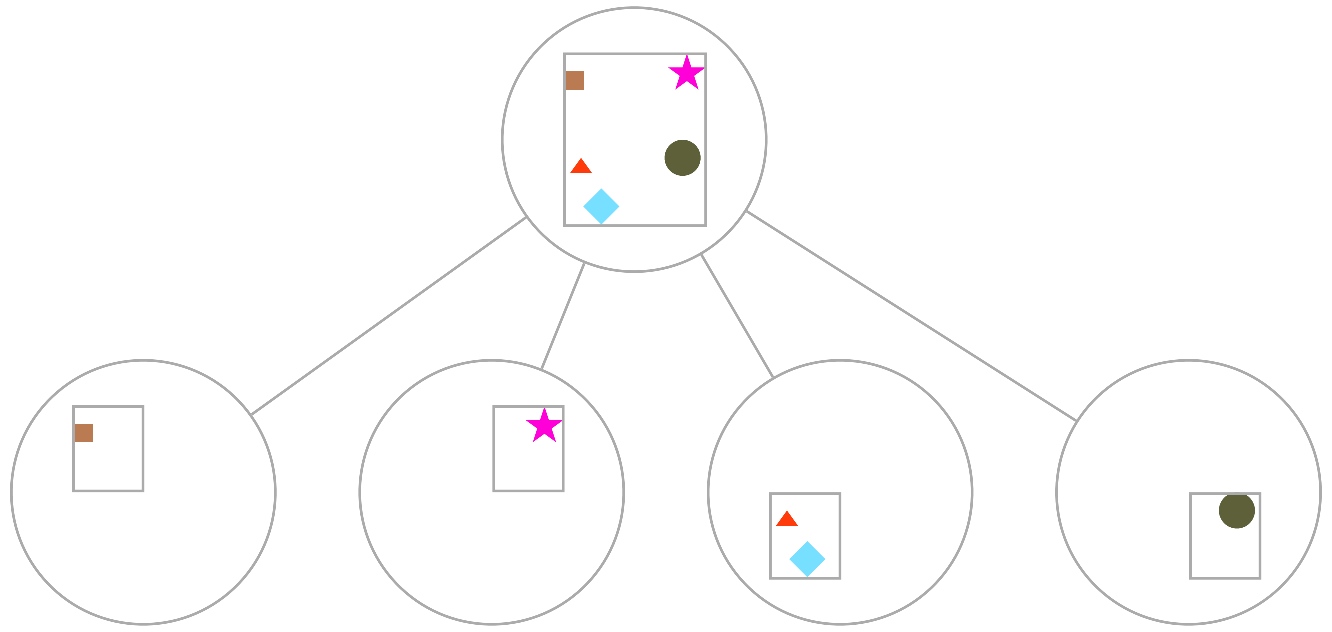




Quadtrees

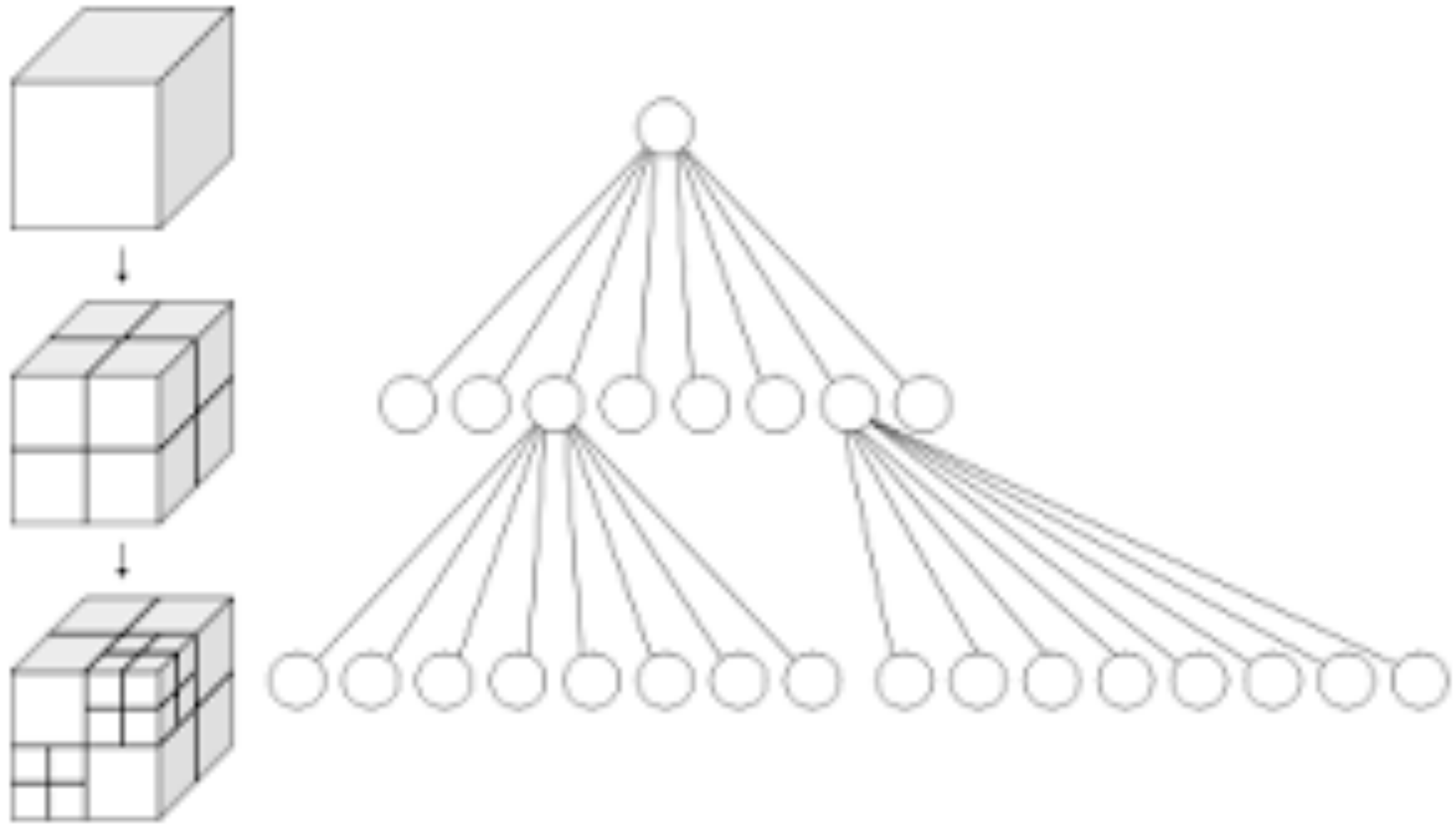
- Assume we know the extents of all our objects in 2D
- Divide space evenly in a dimension
- Recurse until we have the granularity we feel is appropriate





Expanding into three dimensions: Octrees

- Find the maximally extending bounding box for a scene
- Partition it into eights
  - halve each side
- Build a tree of the contents, until a suitably sized object (e.g., a triangle) is only resident of a leaf node



kD-Trees

- Split space using planes
- Attempt to create a balanced tree by subdividing space with approximately equal numbers of objects in each partition
- Another recursive traversal system