Global Illumination I
Whitted-Style Ray Tracing

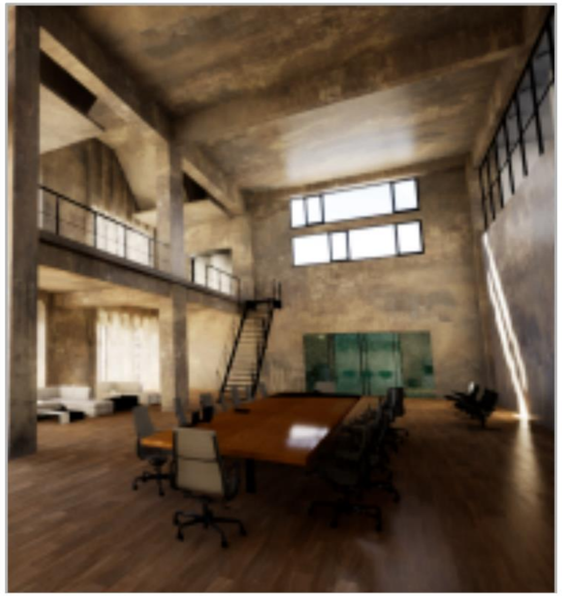
Baoquan Chen

Why Ray Tracing?

- Rasterization couldn’t handle **global effects** well
  - (Soft) shadows
  - Light bounces more than once

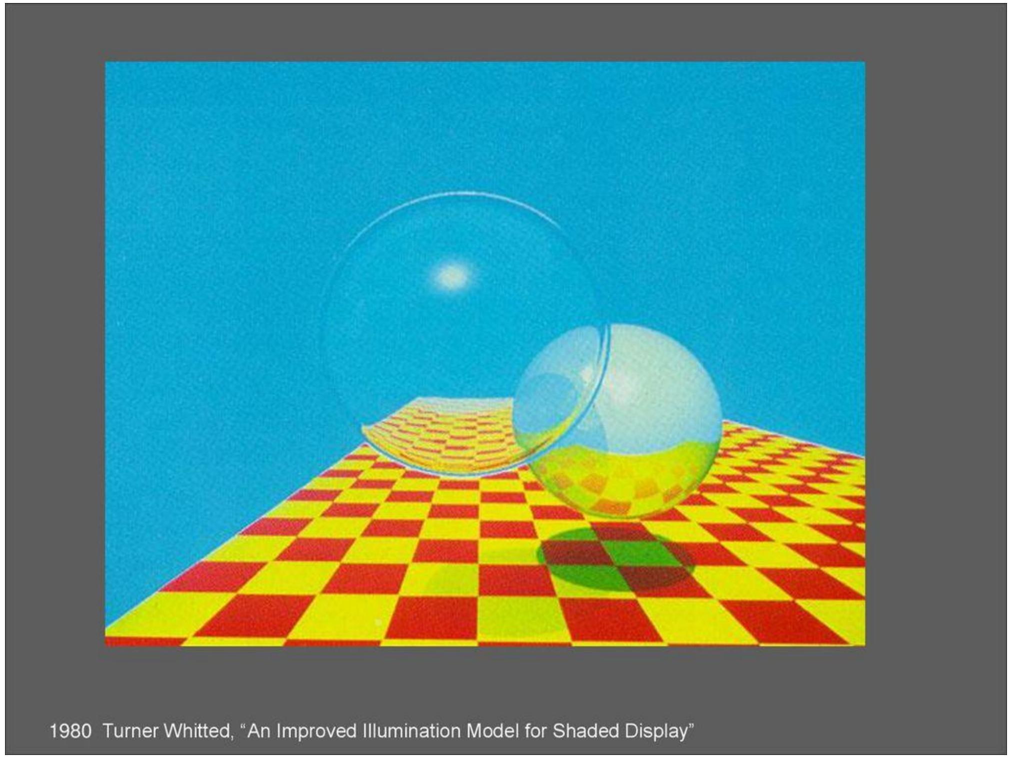
Soft shadows

Glossy reflection



Indirect illumination

Ray Tracing by Turner Whitted



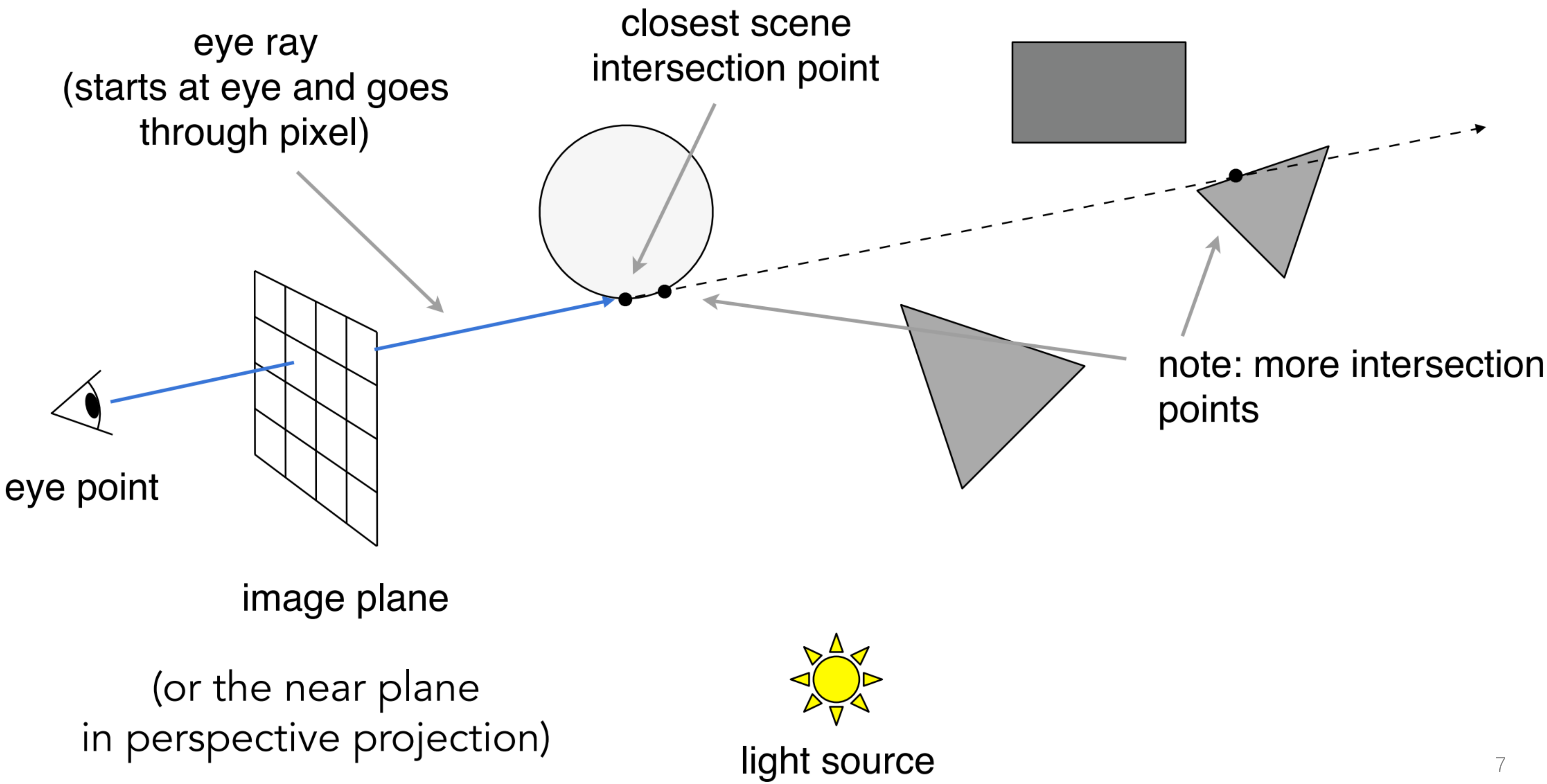
The first scene through ray tracing

Turner Whitted (right)

From Rasterization to Ray Tracing

- Simple shading (typified by OpenGL, z-buffering, and Phong illumination model) assumes:
  - direct illumination (light leaves source, bounces at most once, enters eye)
  - no shadows (except using shadow buffer)
  - opaque surfaces
  - point light sources (otherwise integration for area lights)
  - sometimes fog
- (Whitted-style) ray tracing relaxes that, simulating:
  - specular reflection
  - shadows
  - transparent surfaces (transmission with refraction)
  - sometimes indirect illumination (a.k.a. global illumination)
  - sometimes area light sources
  - sometimes fog

Let's start from: Ray Casting



Ray Casting

- A very flexible visibility algorithm
  - loop y, loop x
    - shoot ray from eye point through pixel (x,y) into scene
    - intersect with all surfaces, find first one the ray hits
    - shade that surface point to compute pixel (x,y)'s color

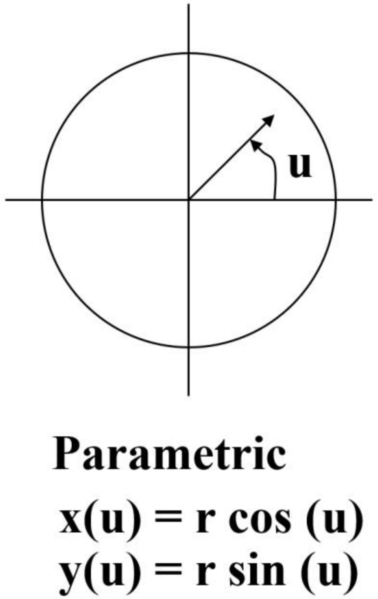
```
Raycast()                          // generate a picture
    for each pixel x,y
        color(pixel) = Trace(ray_through_pixel(x,y))

Trace(ray)                         // fire a ray, return RGB radiance
                                   // of light traveling backward along it
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color

Closest_intersection(ray)
    for each surface in scene
        calc_intersection(ray, surface)
    return the closest point of intersection to viewer
    (also return other info about that point, e.g., surface
  normal, material properties, etc.)

Shade(point, ray)                 // return radiance of light leaving
                                  // point in opposite of ray direction
    calculate surface normal vector
    use Phong illumination formula (or something similar)
    to calculate contributions of each light source
```

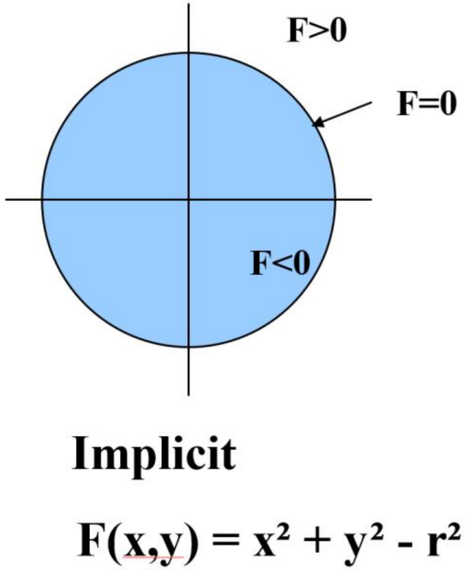
Ray Casting

- This can be easily generalized to give recursive ray tracing, that will be discussed later
- Can handle translucency (which rasterization cannot!)
- calc_intersection (ray, surface) is the most important operation
  - compute not only coordinates, but also geometric or appearance attributes at the intersection point

Ray-Surface Intersections

- How to represent a ray?
  - A ray is $\boldsymbol{p} + t\boldsymbol{d}$: $\boldsymbol{p}$ is ray origin, $\boldsymbol{d}$ the direction
  - $t = 0$ at origin of ray, $t > 0$ in positive direction of ray
  - typically assume $||\boldsymbol{d}|| = 1$
  - $\boldsymbol{p}$ and $\boldsymbol{d}$ are typically computed in world space
- Recap: how to represent a surface?
  - Implicit functions: $f(\boldsymbol{x}) = 0$
  - Parametric functions: $\boldsymbol{x} = \boldsymbol{g}(u, v)$





Ray-Surface Intersections

- Compute Intersections:
  - Substitute ray equation for x
  - Find roots
  - Implicit: $f(\boldsymbol{p} + t\boldsymbol{d}) = 0$
    - one equation in one unknown – univariate root finding
  - Parametric: $\boldsymbol{p} + t\boldsymbol{d} - \boldsymbol{g}(u, v) = 0$
    - three equations in three unknowns (t,u,v) – multivariate root finding
  - For univariate polynomials, use closed form solution; otherwise, use numerical root finder

Ray-Sphere Intersection

- Ray-sphere intersection is an easy case
- A sphere's implicit function is: $x^2 + y^2 + z^2 - r^2 = 0$ if sphere at origin
- The ray equation is: $x = p_x + td_x$
  $y = p_y + td_y$
  $z = p_z + td_z$
- Substitution gives: $(p_x + td_x)^2 + \left(p_y + td_y\right)^2 + \left(p_y + td_y\right)^2 - r^2 = 0$
- A quadratic equation in t.
- Quadratic formula has two roots: $t = (-B \pm \sqrt{B^2 - 4C})/2$
  - which correspond to the two intersection points
  - negative discriminant means ray misses sphere
- Solve the standard way: $A = d_x^2 + d_y^2 + d_z^2 = 1$ (unit vector)
  $B = 2\left(p_x d_x + p_y d_y + p_z d_z\right)$
  $C = p_x^2 + p_y^2 + p_z^2 - r^2$

Ray-Polygon Intersection

- Assuming we have a planar polygon
  - first, find intersection point of ray with plane
  - then check if that point is inside the polygon
- Latter step is a point-in-polygon test in 3-D:
  - inputs: a point x in 3-D and the vertices of a polygon in 3D
  - output: INSIDE or OUTSIDE
  - problem can be reduced to point-in-polygon test in 2-D (**how?**)
- Point-in-polygon test in 2-D:
  - easiest for triangles
  - easy for convex n-gons
  - harder for concave polygons
  - most common approach: subdivide all polygons into triangles
  - for optimization tips, see article by Haines in the book **Graphics Gems IV**

Ray-Plane Intersection

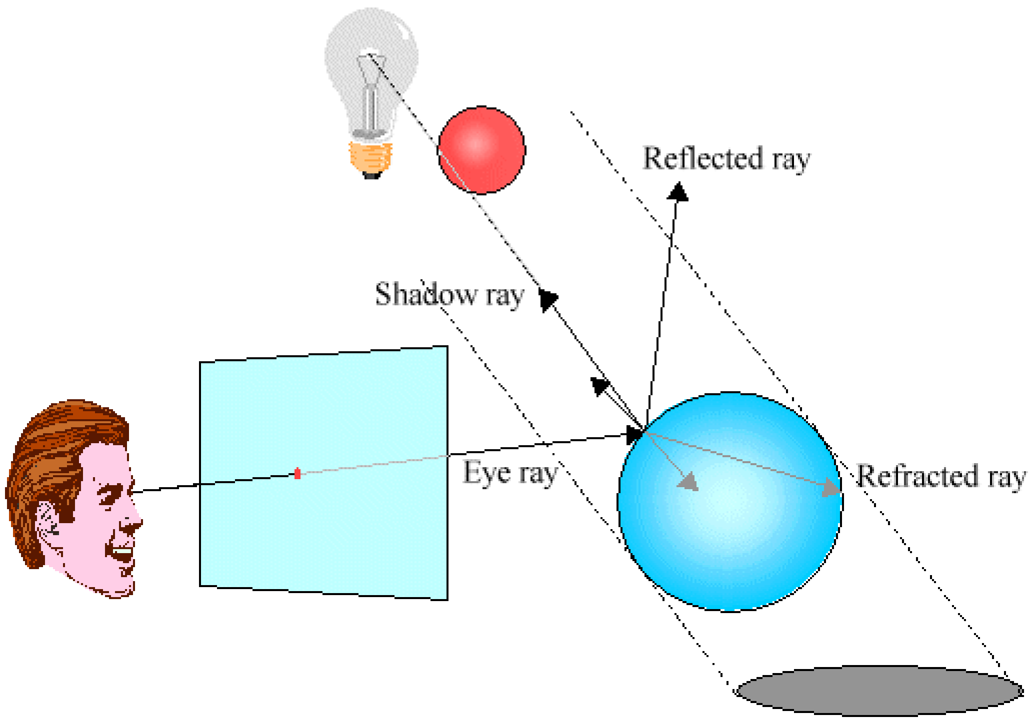
- Plane: $(\boldsymbol{x} - \boldsymbol{q}) \cdot \boldsymbol{n} = 0$
  - where $\boldsymbol{q}$ is reference point on plane, $\boldsymbol{n}$ is plane normal. (some might assume $\|\boldsymbol{n}\| = 1$; we won't)
  - $\boldsymbol{x}$ is point on plane
  - if what you're given is vertices of a polygon
    - compute n with cross product of two (non-parallel) edges
    - use one of the vertices for $\boldsymbol{q}$
  - rewrite plane equation as $\boldsymbol{x} \cdot \boldsymbol{n} + D = 0$
    - equivalent to the familiar formula $Ax + By + Cz + D = 0$
      - where $(A, B, C) = \boldsymbol{n}$, $D = -\boldsymbol{q} \cdot \boldsymbol{n}$
    - fewer values to store
- Steps:
  - substitute ray formula $(\boldsymbol{p} + t\boldsymbol{d})$ into plane eqn, yielding 1 equation in 1 unknown (t).
  - solution: $t = -\frac{\boldsymbol{p} \cdot \boldsymbol{n} + D}{\boldsymbol{d} \cdot \boldsymbol{n}}$
    - note: if $\boldsymbol{d} \cdot \boldsymbol{n} = 0$ then ray and plane are parallel - REJECT
    - note: if $t < 0$ then intersection with plane is behind ray origin - REJECT
  - compute t, plug it into ray equation to compute point $\boldsymbol{x}$ on plane

Projecting A Polygon from 3D to 2D

- Point-in-polygon testing is simpler and faster if we do it in 2D
  - The simplest projections to compute are to the xy, yz, or zx planes
  - If the polygon has plane equation $Ax + By + Cz + D = 0$, then
    - $|A|$ is proportional to projection of polygon in yz plane
    - $|B|$ is proportional to projection of polygon in zx plane
    - $|C|$ is proportional to projection of polygon in xy plane
    - Example: the plane $z = 3$ has $(A, B, C, D) = (0,0,1,-3)$, so $|C|$ is the largest and xy projection is best. We should do point-in-polygon testing using x and y coords.
  - In other words, project into the plane for which the perpendicular component of the normal vector $\boldsymbol{n}$ is largest
- Optimization:
  - We should optimize the inner loop (ray-triangle intersection testing) as much as possible
  - We can determine which plane to project to, for each triangle, as a preprocess
- Point-in-polygon testing in 2D is still an expensive operation (**how to reduce?**)
- Point-in-rectangle is a special case

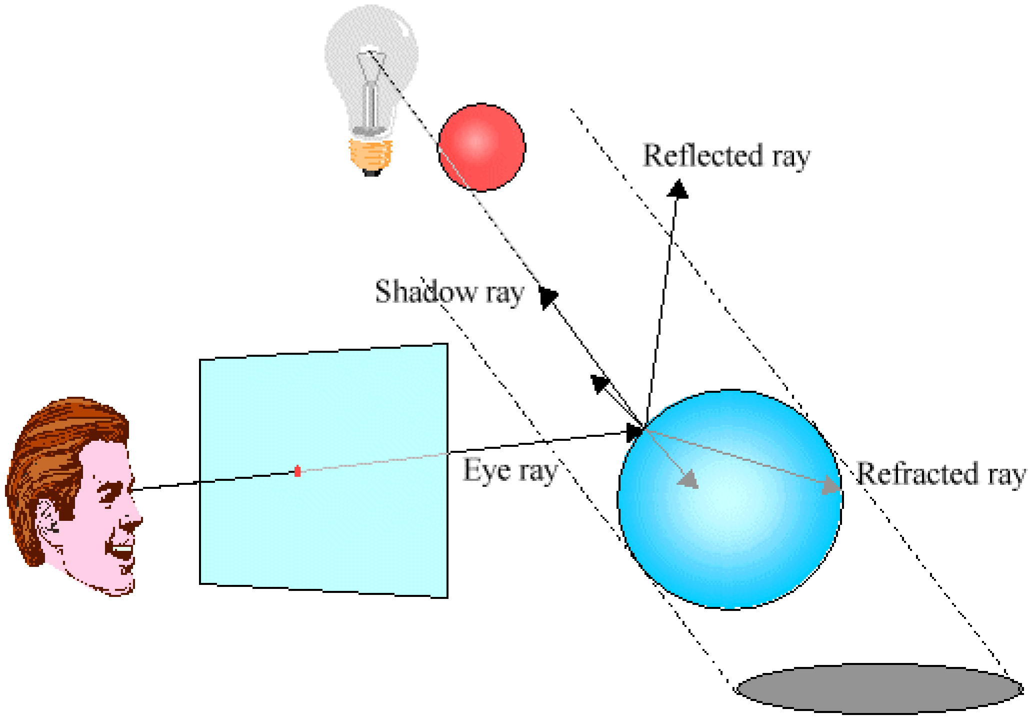
Now Ray Tracing: Ray Types

- We'll distinguish four ray types:
  - Eye rays: originating at the eye
  - Shadow rays: from surface point toward light source
  - Reflection rays: from surface point in mirror direction
  - Transmission rays: from surface point in refracted direction



Ray Tracing Algorithm

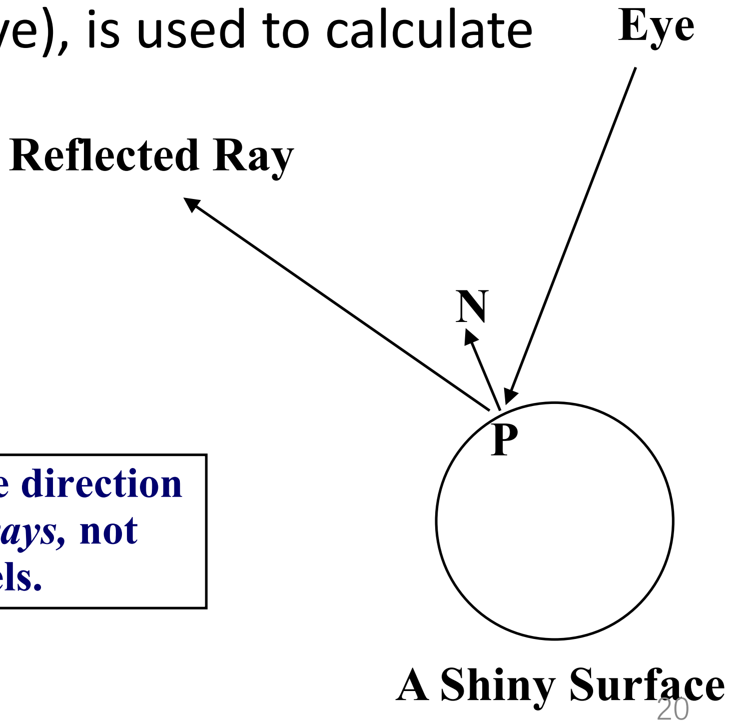
- send ray from eye through each pixel (eye ray)
- compute point of closest intersection with a scene surface
- shade that point by computing shadow rays
- **spawn reflected and refracted rays, repeat**



Specular Reflection Rays

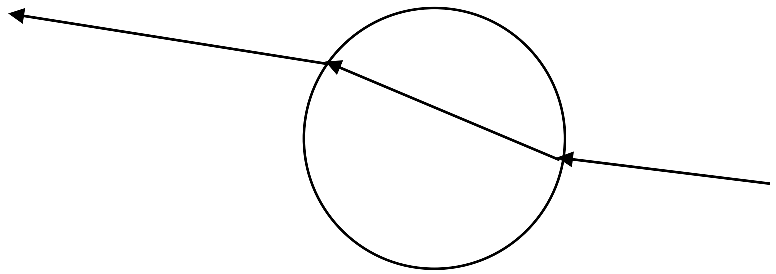
- An eye ray hits a shiny surface
  - We know the direction from which a specular reflection would come, based on the surface normal
  - Fire a ray in this reflected direction
  - The reflected ray is treated just like an eye ray: it hits surfaces and spawns new rays
  - Light flows in the direction opposite to the rays (towards the eye), is used to calculate shading
  - It’s easy to calculate the reflected ray direction

Note: arrowheads show the direction in which we're *tracing the rays*, not the direction the light travels.



Specular Transmission Rays

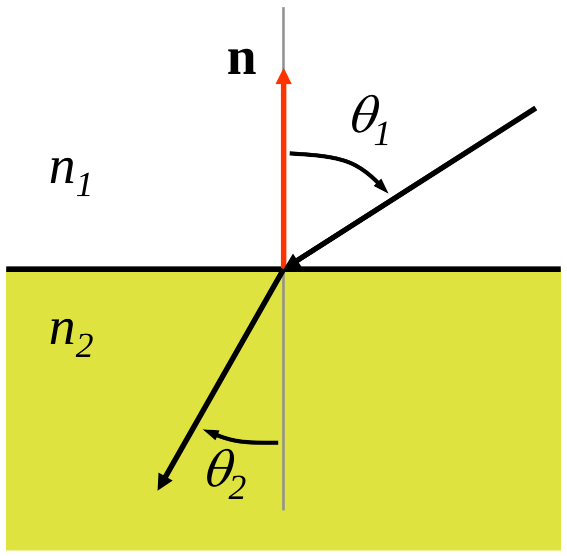
- To add transparency:
  - Add a term for light that's coming from within the object
  - These rays are refracted (bent) when passing through a boundary between two media with different refractive indices
  - When a ray hits a transparent surface fire a *transmission ray* into the object at the proper refracted angle
  - If the ray passes through the other side of the object then it bends again (the other way)



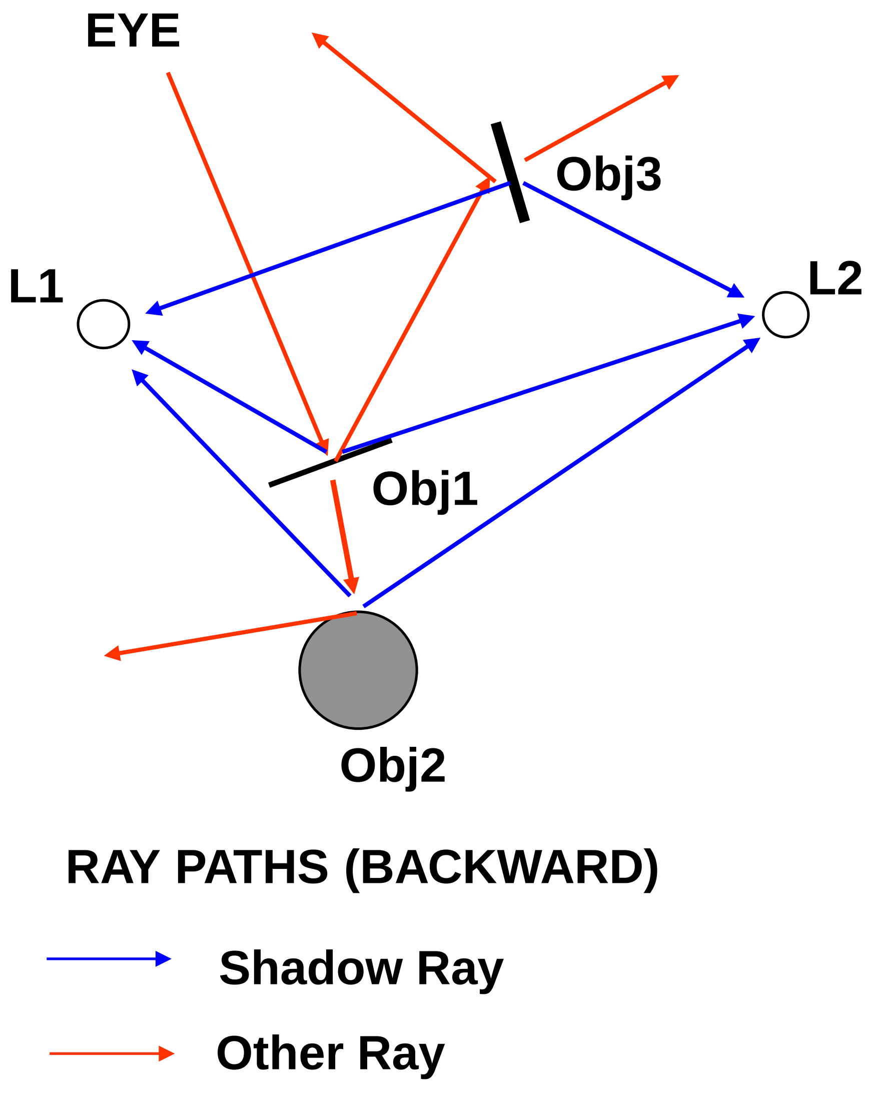
Refraction

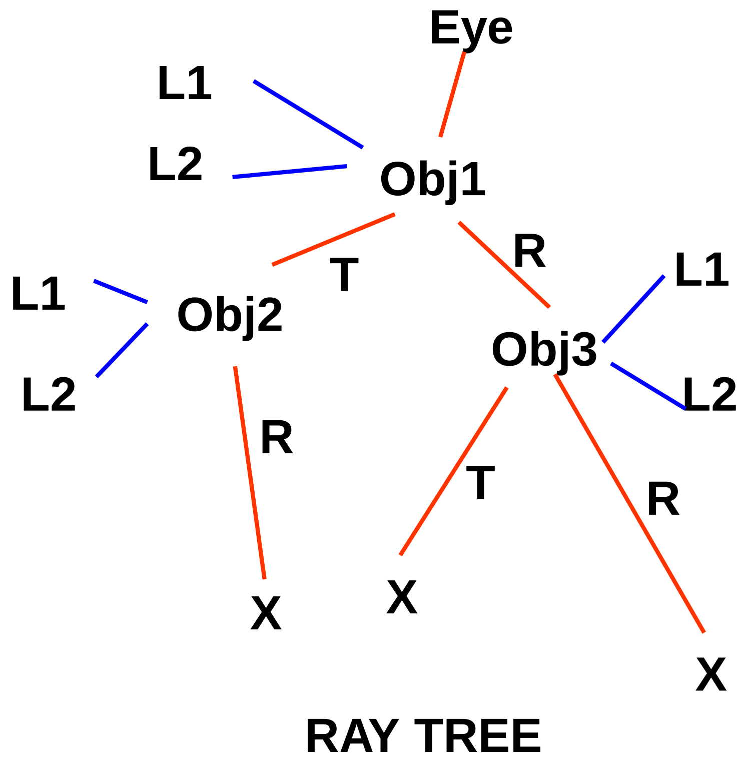
- Refraction:
  - The bending of light due to its different velocities through different materials
  - rays bend toward the normal when going from sparser to denser materials (e.g. air to water), away from normal in opposite case
- Refractive index:
  - Light travels at speed c/n in a material of refractive index n
  - c is the speed of light in a vacuum
  - c varies with wavelength, hence rainbows and prisms
  - Use Snell's law $n_1 \sin \theta_1 = n_2 \sin \theta_2$ to derive refracted ray direction
    - note: ray dir. can be computed without trig functions (only sqrts)

MATERIAL	INDEX OF REFRACTION
air/vacuum	1
water	1.33
glass	about 1.5
diamond	2.4

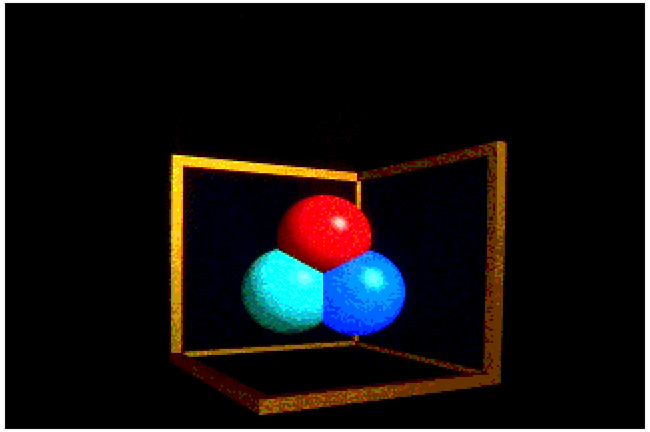


Ray Hierarchy

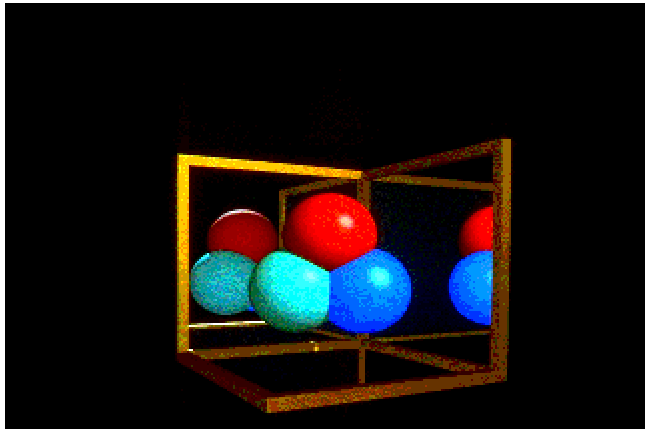




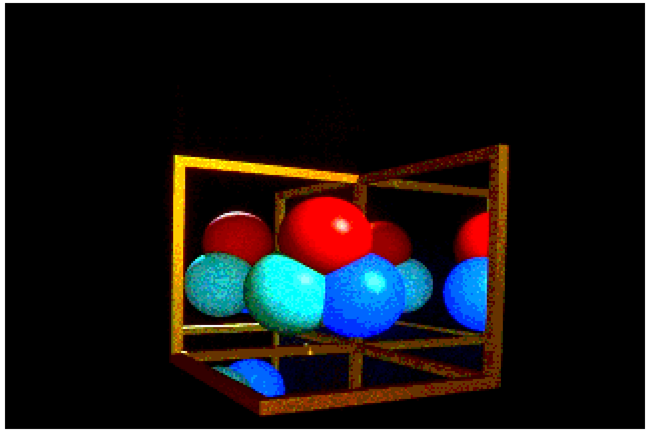
Ray Casting vs. Ray Tracing



Ray Casting -- 1 bounce



Ray Tracing -- 2 bounce



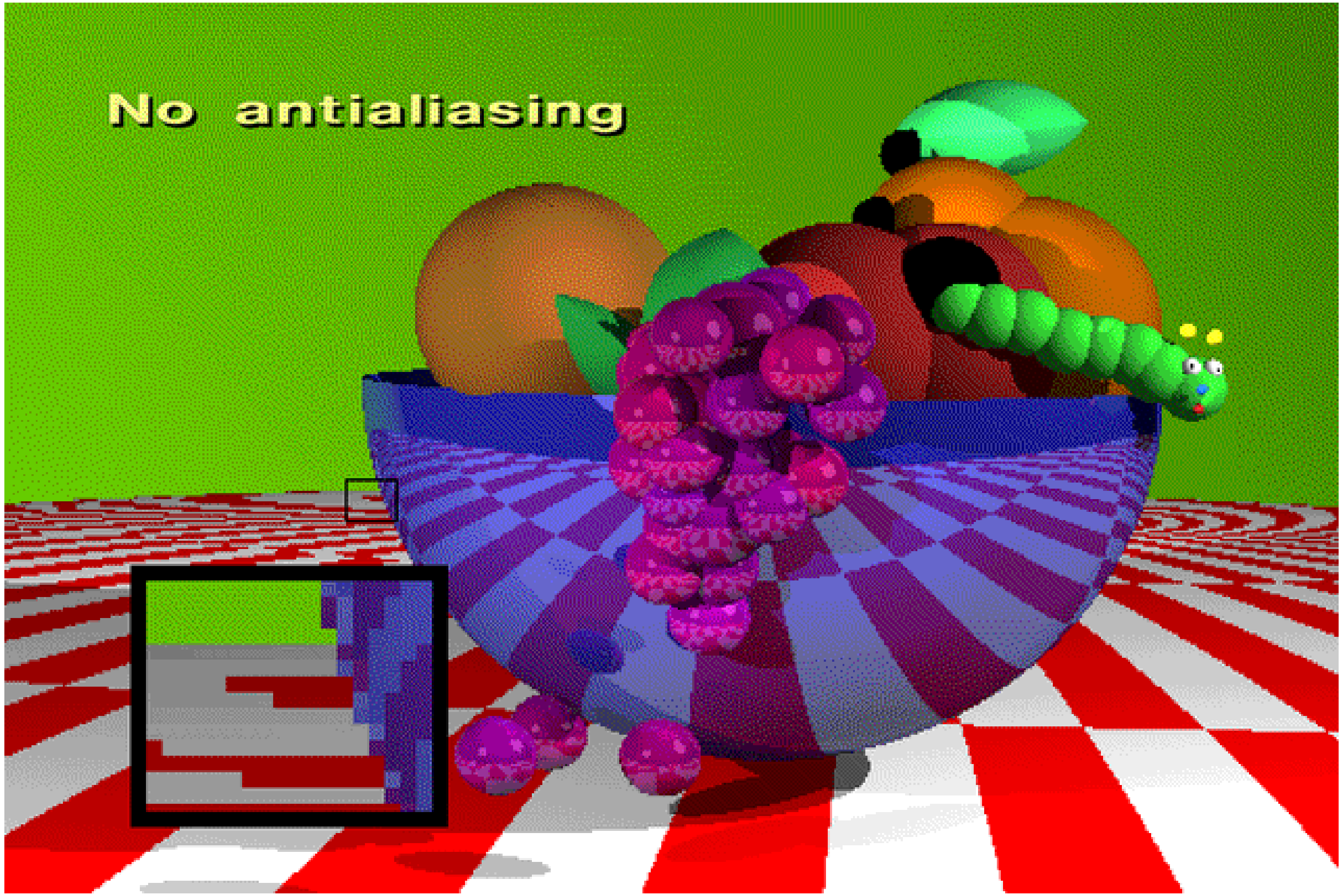
Ray Tracing -- 3 bounce

From a Ray Caster to a Ray Tracer

```
Trace(ray)                  // fire a ray, return RGB radiance
                                   // of light traveling backward along it
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color

Shade(point, ray)            /* return radiance along ray */
    radiance = black;        /* initialize color vector */
    for each light source
        shadow_ray = calc_shadow_ray(point,light)
        if !in_shadow(shadow_ray,light)
            radiance += phong_illumination(point,ray,light)
    if material is specularly reflective
        radiance += spec_reflectance *
            Trace(reflected_ray(point,ray)))
    if material is transmissive
        radiance += spec_transmittance *
            Trace(refracted_ray(point,ray)))
    return radiance
```

Problem with Simple Ray Tracing

One problem:



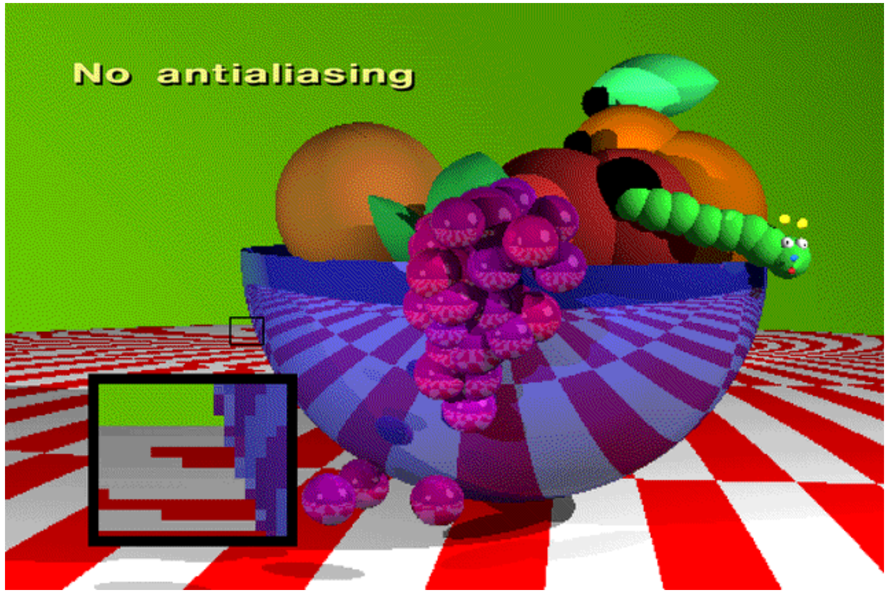
Any other problems?

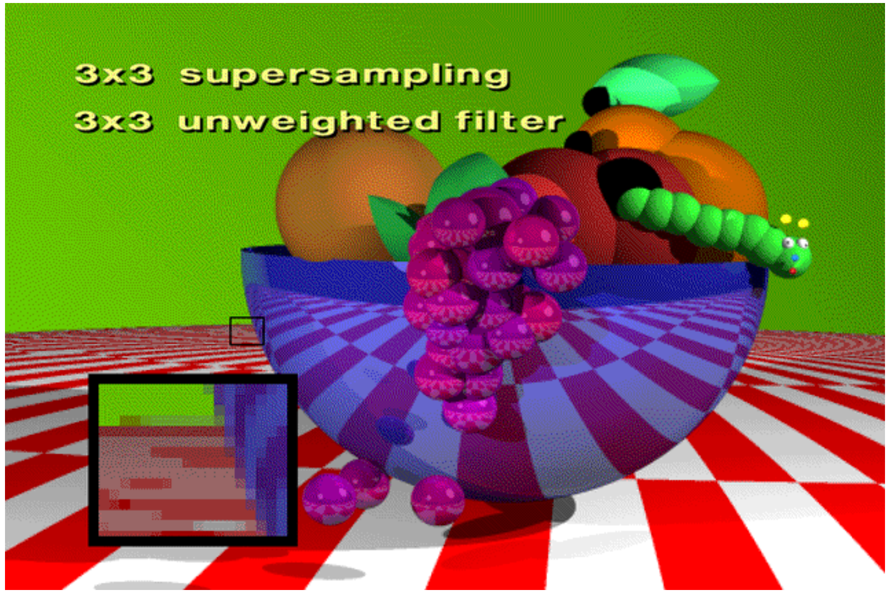
Aliasing

- Ray tracing shoots one ray per pixel
- But a pixel represents an area; one ray samples only one point within the area; an area consists *infinite* number of points
  - These points may not all have the same color
  - This leads to *aliasing*
    - jaggies
    - moiré patterns
- How do we fix this problem?
  - Recall antialiasing we talked earlier

Antialiasing: Supersampling

- We talked about two antialiasing methods
  - Supersampling
  - Pre-filtering (e.g., MIP-mapping for texture mapping)
- Here we use supersampling
  - Fire more than one ray for each pixel (e.g., a 3x3 grid of rays)
  - Average the results using a filter (or some kind of filter)





- What if pre-filtering?

Antialiasing: Adaptive Supersampling

- Supersampling can be done **adaptively**
  - divide pixel into 2x2 grid, trace 5 rays (4 at corners, 1 at center)
  - if the colors are similar then just use their average
  - otherwise recursively subdivide each cell of grid
  - keep going until each 2x2 grid is close to uniform or limit is reached
  - filter the result

- Behavior of adaptive supersampling
  - Areas with fairly constant appearance are sparsely sampled
  - Areas with lots of variability are heavily sampled

Antialiasing: Stochastic Adaptive Supersampling

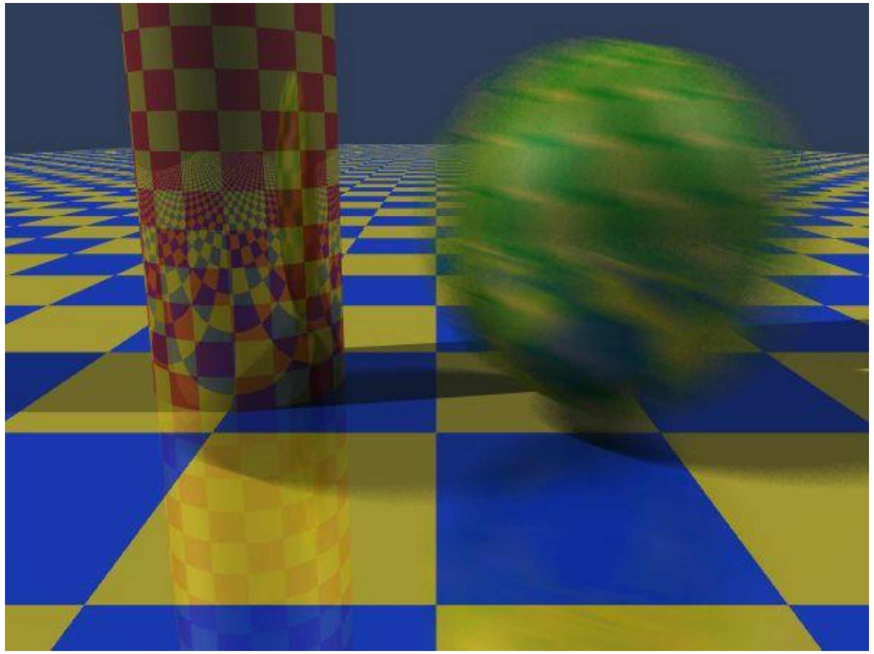
- Issues
  - even with massive supersampling visible aliasing is possible when the sampling grid interacts with regular structures that may be almost aligned with sampling grids
  - noticeable beating, moiré patterns, etc… are possible
- Solution: adaptive supersampling can be done ***stochastically***
  - instead of a regular grid, subsample randomly (or pseudo)
  - aliasing is replaced by less visually annoying noise
  - adaptively sample statistically
  - keep taking samples until the color estimates converge
- How?
  - jittering: perturb a regular grid
  - Jitter pattern can be pre-generated (designed)
  - Consider blue noise!

Temporal Aliasing

- Aliasing happens in time as well as space
  - the sampling rate is the frame rate, 30Hz for NTSC video, 24Hz for film
  - fast moving objects move large distances between frames
  - if we point-sample time, objects have a jerky look
- Real media (film and video) automatically do temporal anti-aliasing
  - photographic film integrates over the exposure time
  - video cameras have persistence (memory)
  - this shows up as *motion blur* in the photographs
- To avoid temporal aliasing we need to filter in time too
  - so compute frames at 120Hz (should it be fixed?) and average them together (with appropriate weights)?
  - fast-moving objects become blurred streaks

Motion Blur

- Apply stochastic sampling to time as well as space
- Assign a time as well as an image position to each ray
- The result is still-frame motion blur and smooth animation
- This is an example of **distribution ray tracing**



Motion Blur: a classic example

- From Foley et. al. Plate III.16
- Rendered using distribution ray tracing at 4096x3550 pixels, 16 samples per pixel.
- Note motion-blurred reflections and shadows with penumbrae cast by extended light sources.

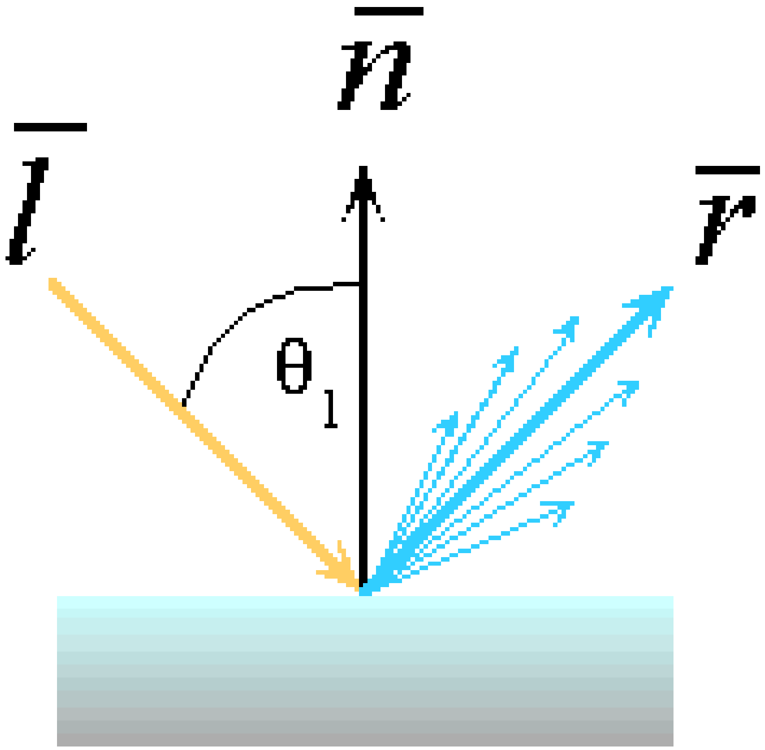
Distribution Ray Tracing

- We've done
  - distribute rays throughout a pixel to get spatial antialiasing
  - distribute rays in time to get temporal antialiasing (motion blur)
- We can
  - distribute rays in reflected ray direction to simulate gloss
  - distribute rays across area light source to simulate penumbras (soft shadows)
  - distribute rays throughout lens area to simulate depth of field
  - distribute rays across hemisphere to simulate diffuse interreflection (radiosity)

- a.k.a. "**distributed ray tracing**" or stochastic ray tracing
- powerful idea! (but can get slow)

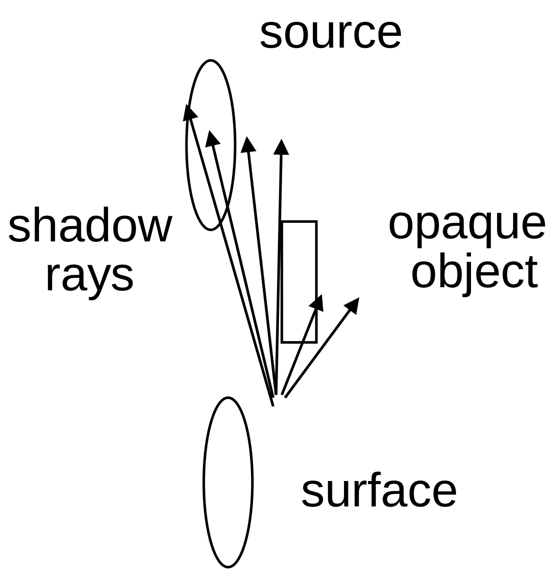
Gloss and Highlights

- Simple ray tracing spawns only one reflected ray
- But Phong illumination models a cone of rays
  - Produces fuzzy highlights
  - Change fuzziness (cone width) by varying the shininess parameter
- The solution is to spawn a cluster of rays
- Again, *stochastic sampling* can be used
  - Stochastically sample rays within the cone
  - Sampling probability drops off sharply away from the specular angle
  - Highlights can be soft, blurred reflections of other objects

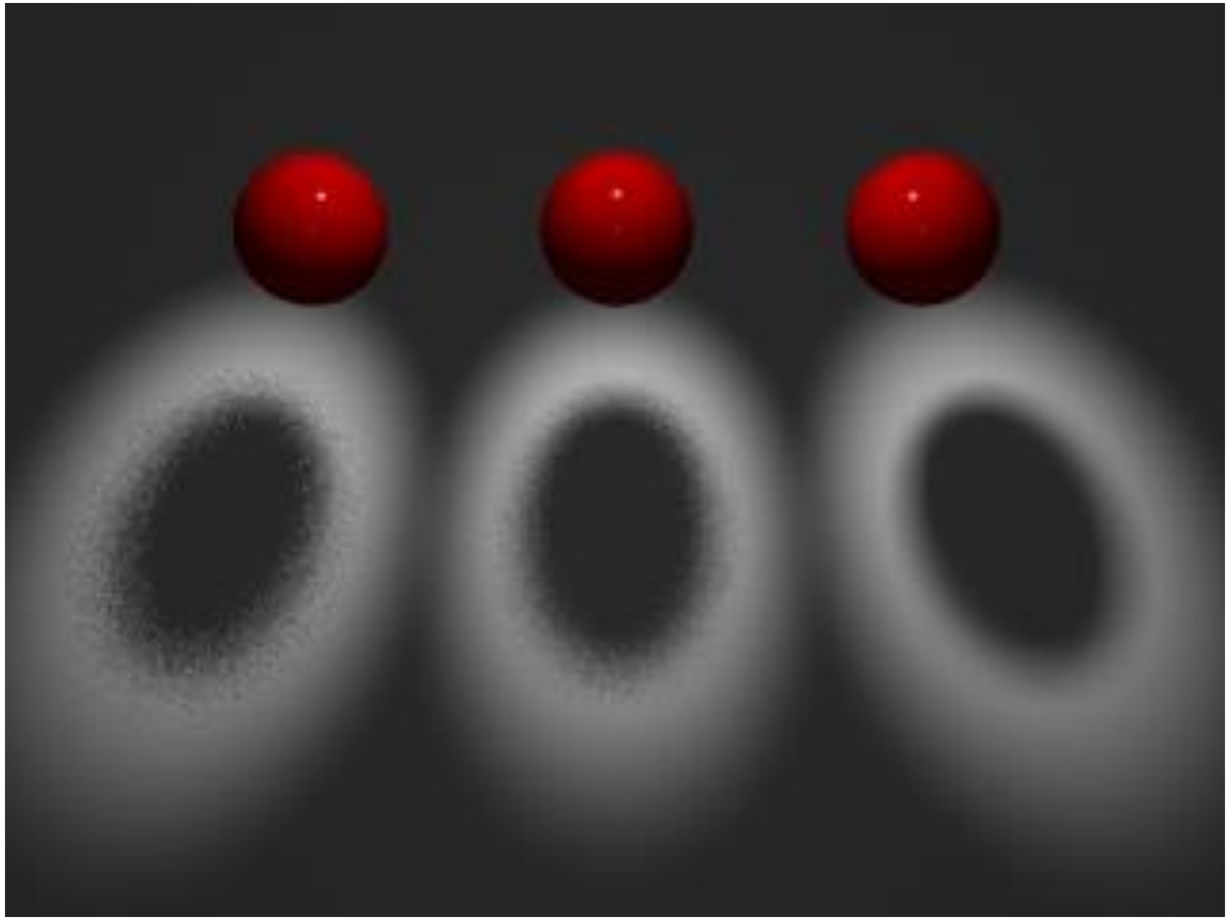


Soft Shadows

- Point light sources produce sharp shadow edges
  - the point is either shadowed or not
  - only one ray is required
- With an extended light source the surface point may be partially visible to it (*partial eclipse*)
  - only part of the light from the sources reaches the point
  - the shadow edges are softer
  - the transition region is the *penumbra*
- Distribution ray tracing can simulate this:
  - fire shadow rays from random points on the source
  - weight them by the brightness
  - the resulting shading depends on the fraction of the obstructed shadow rays

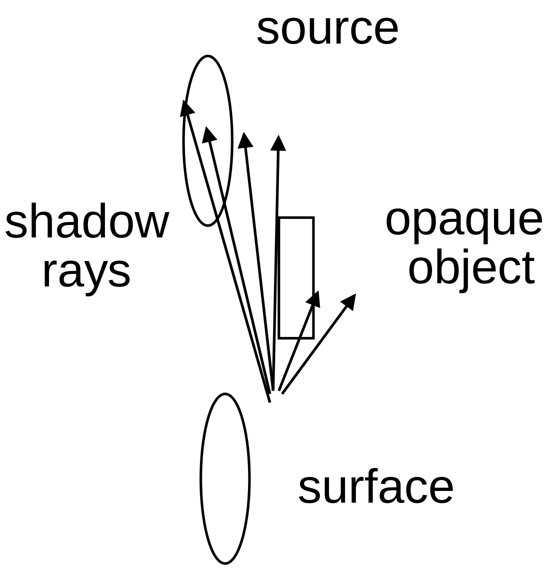


Soft Shadows



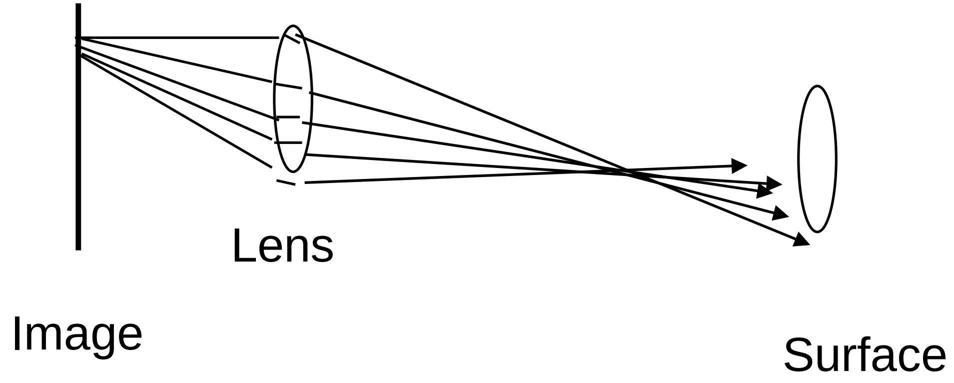
fewer rays, more noise

more rays, less noise



Depth of Field

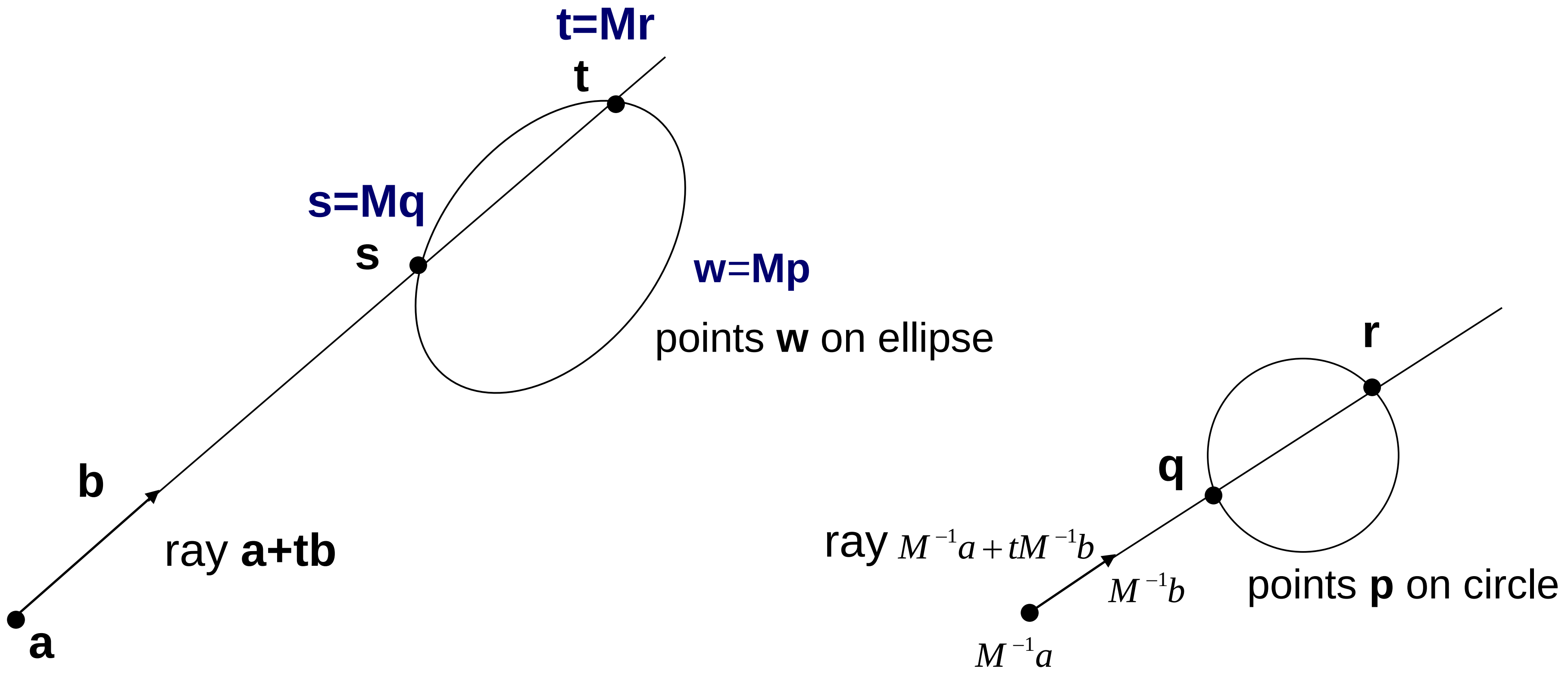
- The pinhole camera model only approximates real optics
  - real cameras have lenses with focal lengths
  - only one plane is truly in focus
  - points away from the focus project as disks
  - the further away from the focus the larger the disk
- the range of distance that appear in focus is the *depth of field*
- simulate this using stochastic sampling through different parts of the lens



Instancing

- The basic idea of instancing is that an object is distorted by a transformation matrix before the object is displayed. For example, in 2D an arbitrary ellipse is an instance of a circle because we can store a unit circle and the composite transformation matrix that transforms the circle to the ellipse. Thus the explicit construction of the ellipse is left as a future procedure operation at render time.
- With the concept of instancing, in ray tracing we can choose what space to do ray-object intersection in. If we have a ray a+tb (a: eye point; b: ray vector; t: parameter) we want to intersect with the transformed object, we can instead intersect an inversely-transformed ray (still a ray!) with the untransformed object. That means, computing a ray and an ellipse intersection can be converted to a problem of computing ray-circle intersection instead.
- Pay attention to normal transformation for correct shading: if the normal at the intersection point of the base object is n, compute its correct normal in the transformed space.

Instancing


t=Mr
t
s=Mq
s
w=Mp
points **w** on ellipse
b
ray **a+tb**
a
r
q
ray $M^{-1}a + tM^{-1}b$
$M^{-1}b$
points **p** on circle
$M^{-1}a$

Speeding Up Ray Tracing

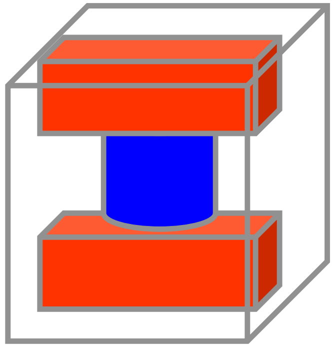
- Trace fewer rays
  - most relevant in recursive ray tracing
- Do fewer ray-surface intersection tests
  - subsequent hits on the same object often hit the same polygon.
  - shadow object caching
    - When a shadow ray hits an object, remember that object and check it first against the next shadow ray heading toward that light.
    - If it hits, you know that shadow applies.
- Speed up each ray-surface intersection test
  - optimize ray-triangle, ray-sphere intersection code
  - compile with optimizer

Spatial Data Structures

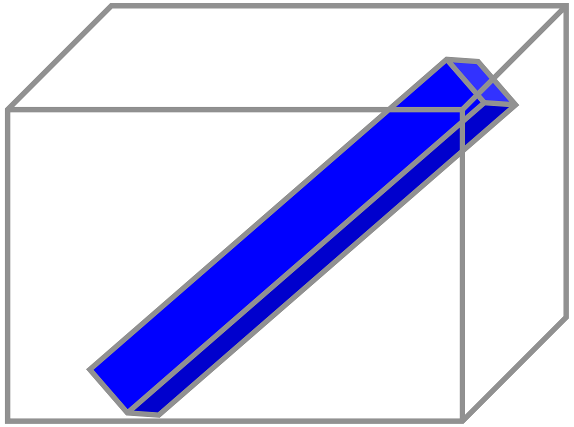
- Data structures for efficiently storing geometric information
- They are useful for
  - Collision detection (will the spaceships collide?)
  - Location queries (which is the nearest post office?)
  - Chemical simulations (which protein will this drug molecule interact with?)
  - Rendering (is this aircraft carrier on-screen?), and more
- Good data structures can give speed up ray tracing by 10x, 100x, or more
- We'll look at
  - Hierarchical bounding volumes
  - Grids
  - Octrees
  - BSP trees

Bounding Volumes

- Simple notion: wrap things that are hard to check for ray intersection in things that are easy to check.
  - Example: wrap a complicated polygonal mesh in a box
  - Ray can’t hit the real object unless it hits the box
  - Adds some overhead, but generally pays for itself.
- Most common bounding volume types: sphere and box
  - box can be axis-aligned (good and bad), or not
- You want a snug fit!



Good!



Bad!

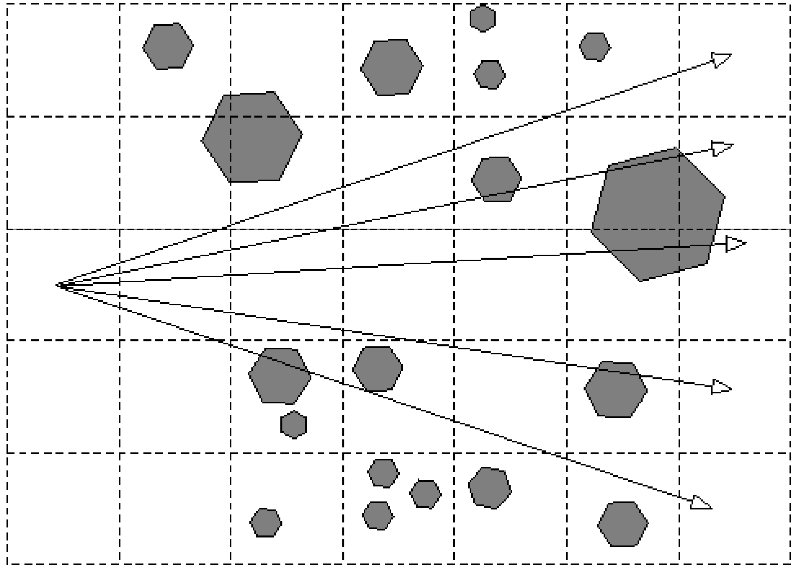
Hierarchical Bounding Volumes (HBV's)

- Tree data structure:
  - List of bounding volumes (BV's), e.g. spheres, boxes
  - Each BV can contain a list of sub-volumes
  - E.g., Human figure:
    - torso bounding-box (BB) contains arm BB, which contains finger BB, etc.
- Intersection testing: recursively descend tree

```
intersect(BV)
    if ray misses BV, return MISS
    closest = infinity
    for each subvolume stored in BV
        if ray intersects subvolume, and closer than closest
            update closest
    return closest
```

- Works well if you use good (appropriate) bounding volumes
- If your BVs are objects, you can have multiple classes and pick the best for each enclosed object!

Grids



- Data structure: a 3-D array of cells (voxels) that tile space
  - Each cell points to list of all surfaces intersecting that cell
- Intersection testing:
  - Start tracing at cell where ray begins
  - Step from cell to cell, searching for the first intersection point
  - At each cell, test for intersection with all surfaces pointed to by that cell
  - If there is an intersection, return the closest one

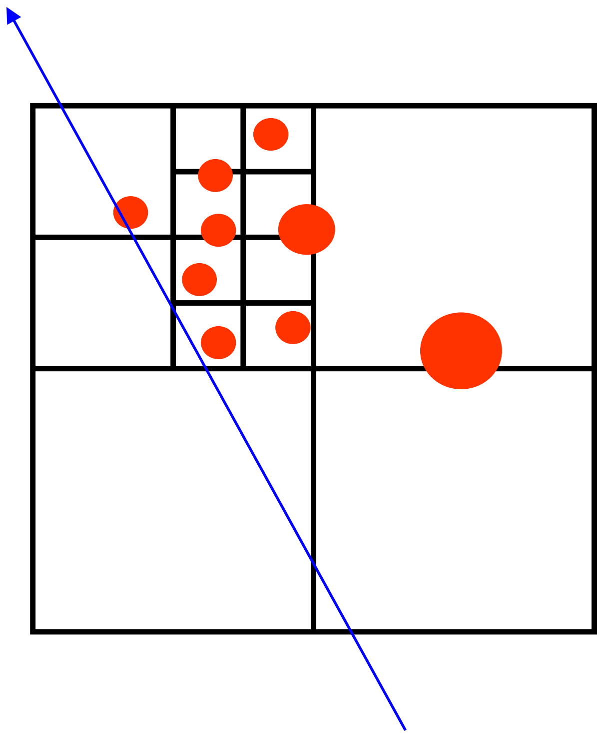
More on Grids

- Be Careful! The fact that a ray passes through a cell and hits an object doesn't mean the ray hit that object in *that* cell
- Optimization: cache intersection point and ray id with respect to each object

- Grids are a poor choice when the world is nonhomogeneous
  - e.g. a teapot in a stadium: many polygons clustered in a small space
- How many cells to use?
  - too few $\Rightarrow$ many objects per cell $\Rightarrow$ slow
  - too many $\Rightarrow$ many empty cells to step through $\Rightarrow$ slow
- Grids work well when you can arrange that each cell lists a few (ten, say) objects
- Better strategy for some scenes: *nested grids*

Octrees

- Quadtree is the 2-D generalization of binary tree
  - node (cell) is a square
  - recursively split into four equal sub-squares
  - stop when leaves get "simple enough"



Octrees

- Octree is the 3-D generalization of quadtree
  - node (cell) is a cube, recursively split into eight equal sub-cubes
  - for ray tracing:
    - stop splitting when the number of objects intersecting the cell gets “small enough” or the tree depth exceeds a limit
    - internal nodes store pointers to children, leaves store list of surfaces
  - more expensive to traverse than a grid
  - but an octree adapts to nonhomogeneous, clumpy scenes better

```
trace(cell, ray) {                    // returns object hit or NONE
    if cell is leaf, return closest (objects_in_cell(cell))
    for child cells pierced by ray, in order          // 1 to 4 of these
        obj = trace(child, ray)
        if obj!=NONE return obj
    return NONE
}
```

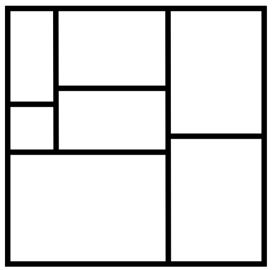
Which Data Structure is Best for Ray Tracing?

- Grids are easy to implement, but they’re memory hogs (and slow) for nonhomogeneous scenes, i.e. most scenes
- Octrees are pretty good, but not as fast as grids for some scenes
- Nested grids seem to be the fastest on static scenes

- If scene is dynamic, the cost of regenerating or updating the data structure may become an issue
- In such cases, hierarchical bounding volumes may be best
- Hierarchical bounding volumes easy to implement if your model is naturally hierarchical (e.g. human), otherwise not

k-d Trees

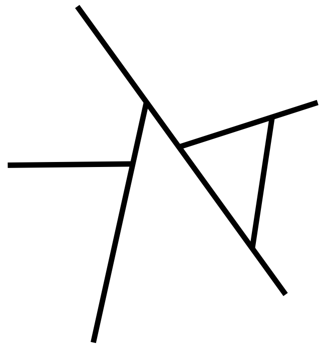
- Relax the rules for quadtrees and octrees:



- first variant: *k-dimensional (k-d) tree*
  - don’t always split at midpoint
  - split only one dimension at a time (i.e. *x* or *y* or *z*)
  - useful for clustering and choosing colormaps for color image quantization

BSP Trees

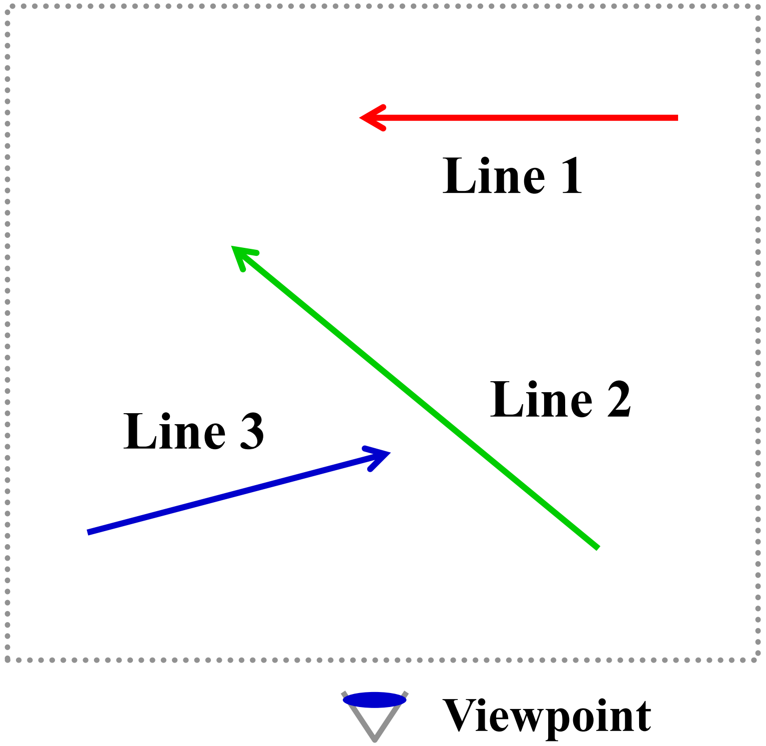
- Relax the rules for quadtrees and octrees:

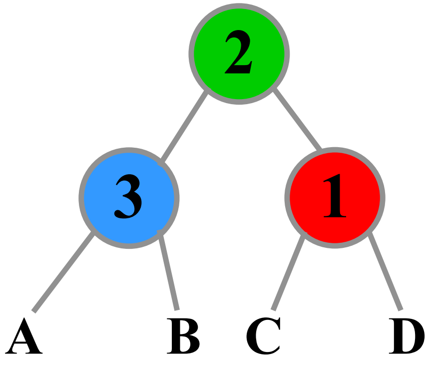


- second variant: *binary space partitioning (BSP) tree*
  - permit splits with any line
  - in general, split k dimensional space with k-1 dimensional hyperplane
    - 2-D space split with lines (most of our examples)
    - 3-D space split with planes
    - each node corresponds to a (potentially unbounded) convex polyhedron
  - useful for Painter's algorithm

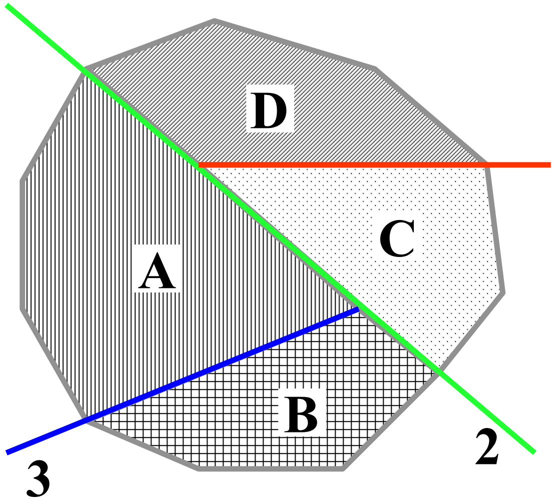
Building a BSP Tree

- Let's look at simple example with 3 line segments
- Arrowheads are to show left and right sides of lines.
- Using line 1 or 2 as root is easy.





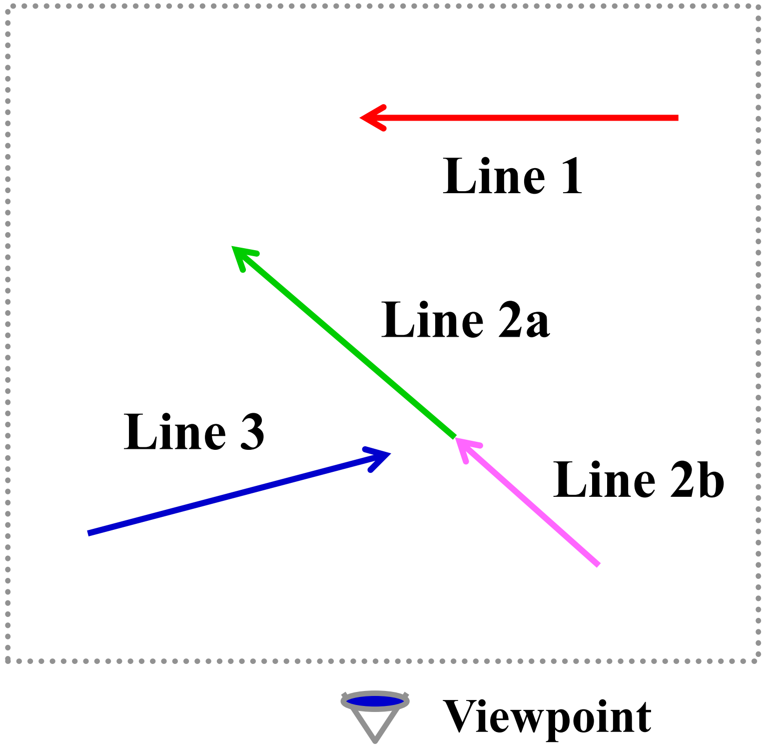
a BSP tree using 2 as root

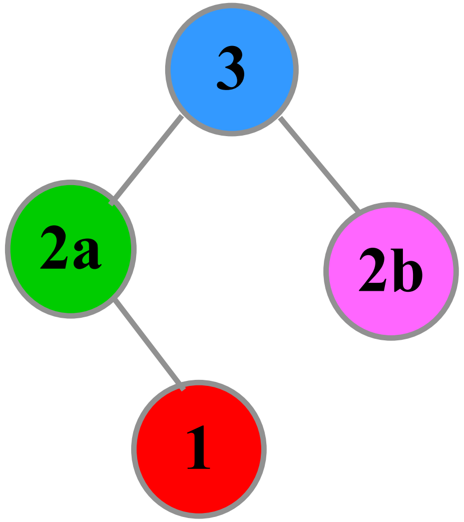


the subdivision of space it implies

Building the Tree 2

- Using line 3 for the root requires a split





Uses for Binary Space Partitioning (BSP) Trees

- Painter's algorithm rendering
  - good for
    - static 3-D scenes with moving viewpoint (flight simulators)
    - architectural scenes with a small number of polygons (DOOM, an old game)
    - if you don't have z-buffer hardware
- Ray tracing

- History:
  - BSP trees first used by Naylor, Fuchs, et al. for Painter's algorithm ~1980
  - theoreticians scoffed at their worst-case performance
  - considered unpromising
  - revived by John Carmack, author of Quake, and the PC game community
    - out of necessity: no z-buffer hardware for PC's at the time

Painter's Algorithm with BSP trees

- Build the tree
  - Involves splitting some polygons
  - Slow, but done only once for static scene

- Correct traversal lets you draw in back-to-front or front-to-back order for any viewpoint
  - Order is view-dependent
  - Precompute tree once
  - Do the "sort" on the fly

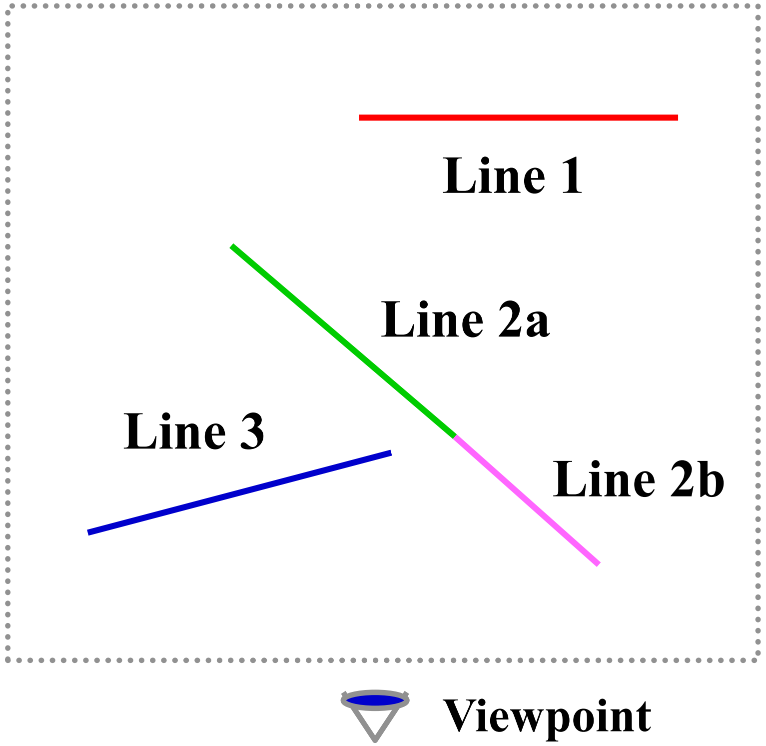
Drawing a BSP Tree

- Each polygon has a set of coefficients:
  Ax + By + Cz + D
- Plug the coordinates of the viewpoint in and see:
  >0 : front side
  <0 : back facing
  =0 : on plane of polygon
- Back-to-front draw: inorder traversal, do farther child first
- Front-to-back draw: inorder traversal, do near child first

```
front_to_back(tree, viewpt) {
   if (tree == null)  return;
   if (positive_side_of(root(tree), viewpt)) {
      front_to_back(positive_branch(tree, viewpt);
      display_polygon(root(tree));
      front_to_back(negative_branch(tree, viewpt);
   }
   else { …draw negative branch first…}
}
```

Drawing Back to Front

- Use Painter's Algorithm for hidden surface removal



Steps:

–*Draw objects on far side of line 3*

»*Draw objects on far side of line 2a*

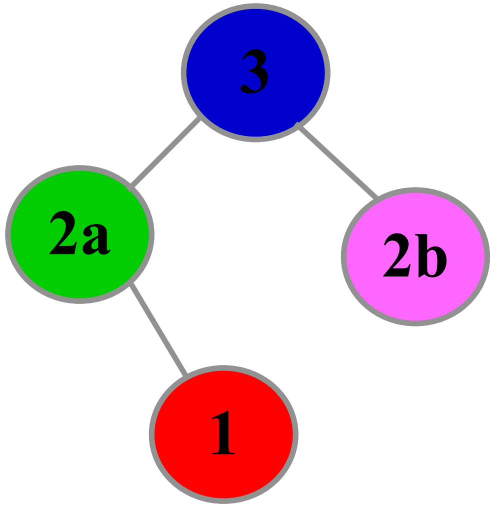
–Draw line 1

»Draw line 2a

–Draw line 3

–*Draw objects on near side of line 3*

»Draw line 2b



Further Speedups

- Do back-face culling with same sign test
- Draw front to back, and…
  - Keep track of partially filled spans
  - Only render parts that fall into spans that are still open
  - Quit when the image is filled
- Clip the BSP tree against the portions of space that you can see!
  - Called *portals*
  - Initial view volume is entire viewing frustum
  - When you look through a doorway, intersect current volume with “beam” defined by doorway
  - Skip a BSP node if it doesn’t intersect the current view volume
  - Much faster than clipping every polygon

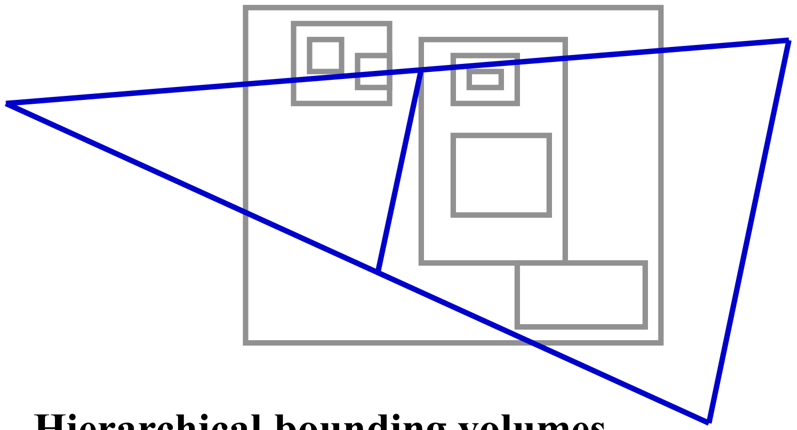
Clipping BSP Trees

- Suppose you have all n polygons in a BSP tree, and it's time to clip them for rendering.

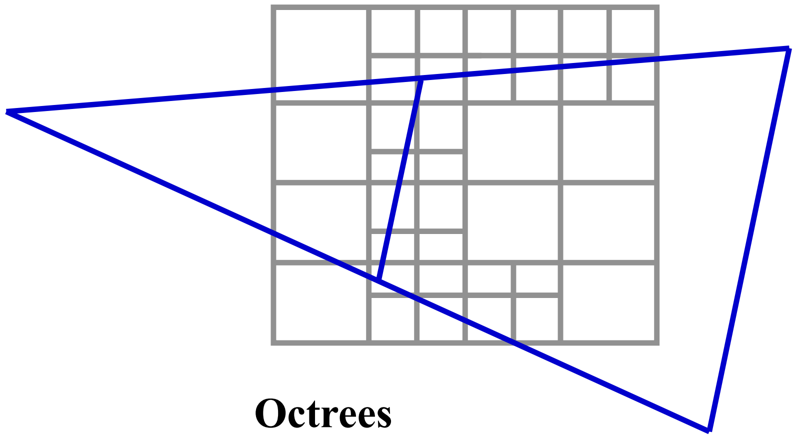
- Clip the tree to the view frustum!
  - This is an intersection operation between the tree of polygons and a BSP tree representing the frustum
  - An O(log n) operation, while clipping all n polygons is O(n)

Clipping Using Spatial Data Structures

- The data structures we used to accelerate ray tracing will work here too!
- In each case, the goal is to accept or reject whole sets of polygons.
- The O(n) task becomes O(log n)
- Scene must be (mostly) fixed, to amortize cost of building the data structure
  - terrain fly-through
  - gaming
- Off-screen stuff can swap out!



Hierarchical bounding volumes



Octrees

More about rendering

- Micro-surfaces (e.g., different roughness)
- Special materials (e.g., different paints or coatings)
- Volumetric material (e.g., skin, with sub-surface reflection)
- Special material + special geometry (e.g., hair)