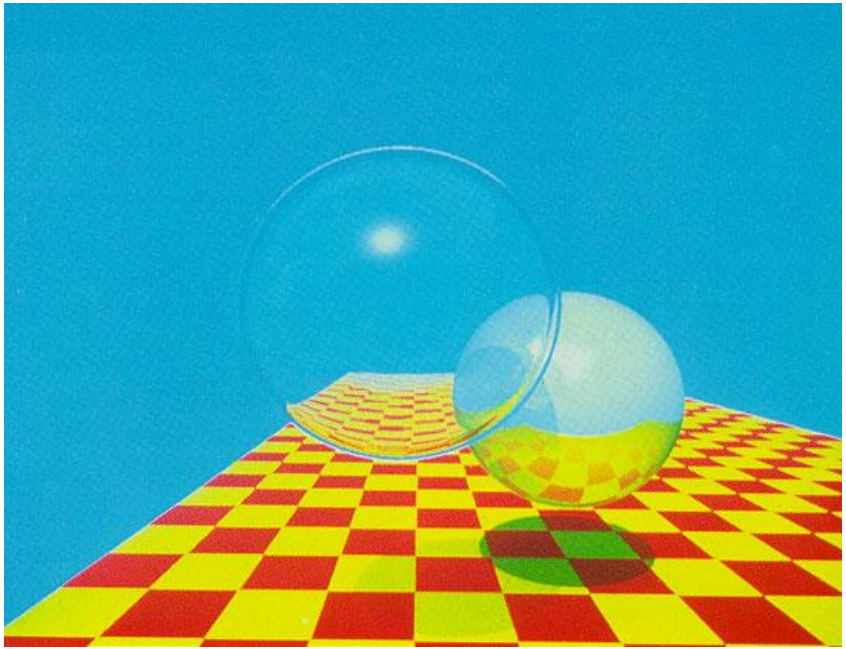
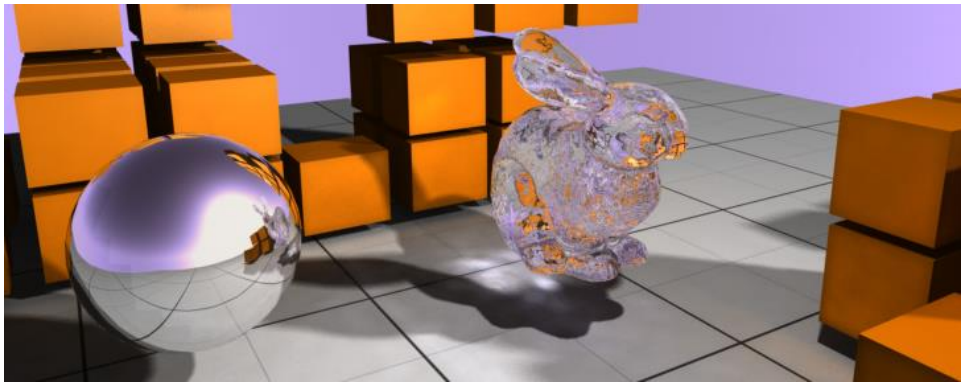
Ray Tracing

Rendering and Techniques





Images: (Left) http://cg.alexandra.dk/?p=278, (Right) Turner Whitted , "An improved illumination model for shaded display," Bell Laboratories 1980

Polygonal Rendering

- OpenGL with one-polygon-at-a-time rendering uses simple Phong lighting and shading models – not physically-based
- No global illumination for inter-object reflection (besides ambient hack)
- VSD/HSR done via h/w z-buffer - fast, but leads to z-fighting and no longer considered “photo-realistic”
- Later take another look at recursive “rendering equation” which models physics of light-object interaction, including inter-object reflection – good approximations exist but are still hugely compute-intensive
- Ray-tracing is the simplest approximation and is feasible in real-time for modest scenes

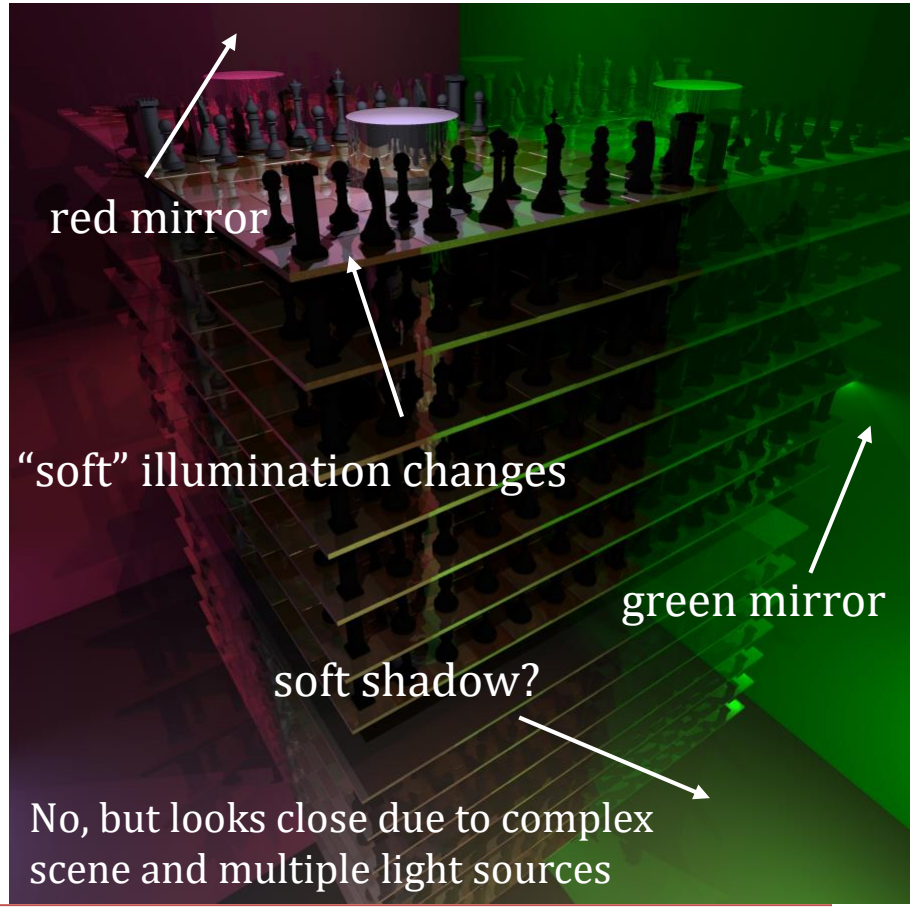
Rendering

Rendered with NVIDIA Iray, a photorealistic rendering solution which adds physically accurate global illumination on top of ray tracing using a combination of physically-based rendering techniques

Introduction

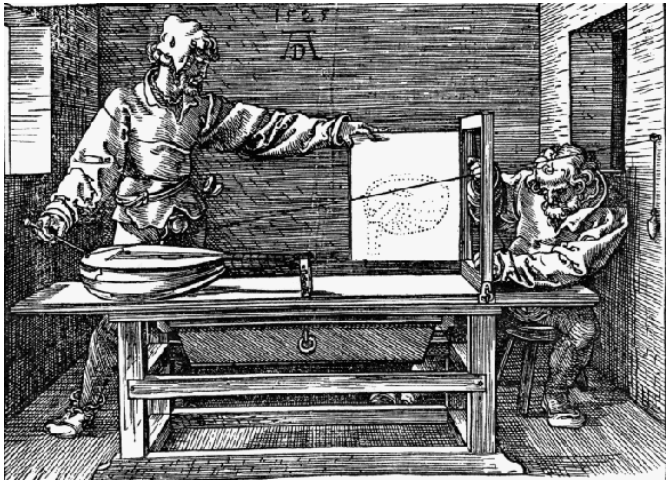
What "effects" do you see?

Rendered in a matter of seconds with Travis Fischer's '09 ray tracer



Origins of Ray Tracing

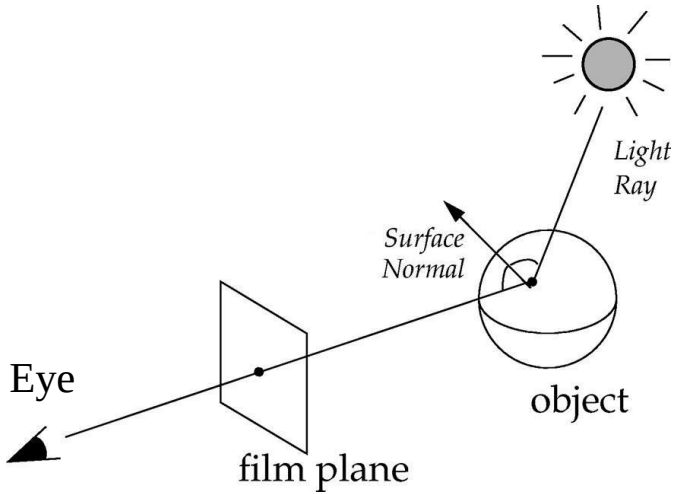
- Generalizing from Durer's wood cut showing perspective projection



- Durer: Record string intersection from center of projection(eye) to nearest object as points on a 2D plane
- Points created are perspective projection of 3D object onto 2D plane – our pixels
- Can think of first starting with sample points on objects and then drawing ray OR starting with ray thru pixel center (or sample points within supersampled pixel)

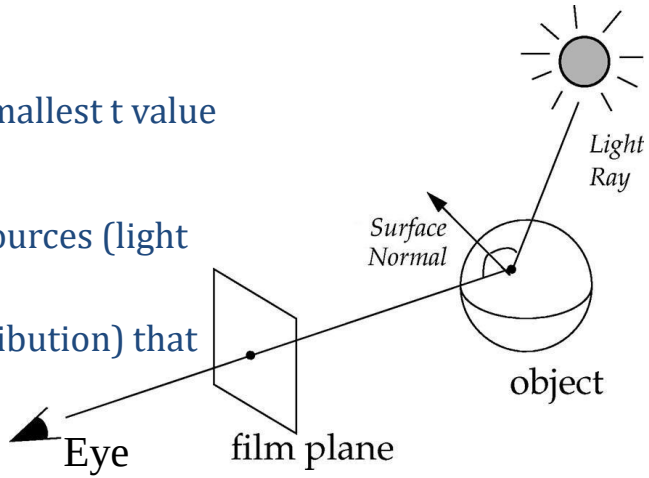
What is a Raytracer?

- A finite back-mapping of rays from camera (eye) through each sample (pixel or subpixel) to objects in scene, to avoid forward solution of having to sample from an infinite number of rays from light sources, not knowing which will be important for PoV
- Each pixel represents either:
  - a ray intersection with an object/light in scene
  - no intersection
- A ray traced scene is a “virtual photo” comprised of many samples on film plane
- Generalizing from one ray, millions of rays are shot from eye, one through each point on film plane



Ray Tracing Fundamentals

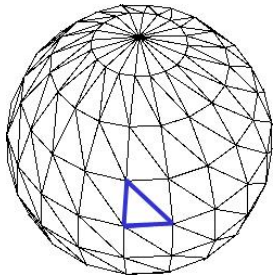
- Generate primary ray
  - shoot rays from eye through sample points on film plane
  - sample point is typically center of a pixel, but alternatively supersample pixels (recall supersampling from Image Processing IV)
- Ray-object intersection
  - find first object in scene that ray intersects with (if any)
  - solves VSD/HSR problem – use parametric line equation for ray, so smallest t value
- Calculate lighting (i.e., color)
  - use illumination model to determine direct contribution from light sources (light rays)
  - reflective objects recursively generate secondary rays (indirect contribution) that also contribute to color; RT tracing only uses specular reflection rays
  - Sum of contributions determines color of sample point
  - No diffuse reflection rays => RT is limited approximation to global illumination



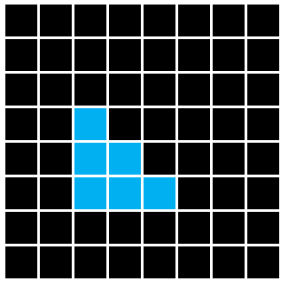
Ray Tracing vs. Triangle Scan Conversion (1/2)

- How is ray tracing different from polygon scan conversion? Shapes and Sceneview both use scan conversion to render objects in a scene and have same pseudocode:

```
for each object in scene:
    for each triangle in object:
        pass vertex geometry, camera matrices, and
        lights to the shader program,
        which renders each triangle (using z-buffer)
        into framebuffer; use simple or complex
            illumination model
```



(triangle rendered to screen)

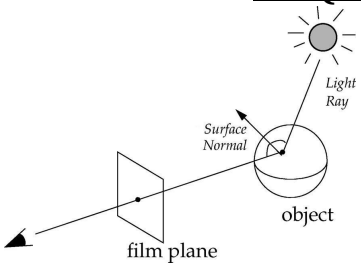


Ray Tracing vs. Triangle Scan Conversion (2/2)

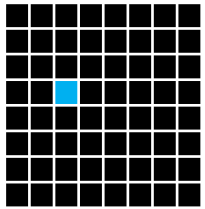
- Ray tracing uses the following pseudocode:

```
for each sample in film plane:
    determine closest object in scene hit by
        a ray going through that sample from eye point
    set color based on calculation of simple/complex
                illumination model for intersected object
```

- Note the distinction: polygonal scan conversion iterates over all **VERTICES** whereas ray tracing iterates over **2D (sub)PIXELS** and calculates intersections in a 3D scene



(ray intersection rendered to screen)

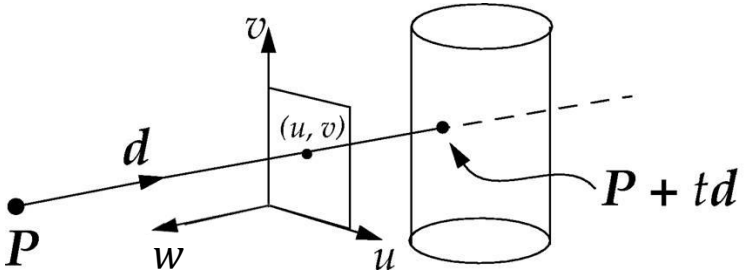


- For polygonal scan conversion must mesh curved objects while with ray tracing we can use an implicit equation directly (if it can be determined) – see Slide 14

Generate Primary Ray (1/4)

Ray Origin

- Let's look at geometry of the problem in normalized world space with canonical perspective frustum (i.e., do not apply perspective transformation)
- Start a ray from "eye point": ***P***
- Shoot ray in some direction ***d*** from ***P*** toward a point in film plane (a rectangle in the ***u-v*** plane in the camera's ***uvw*** space) whose color we want to know
- Points along ray have form ***P*** + t***d*** where
  - ***P*** is ray's base point: camera's eye
  - ***d*** is unit vector direction of ray
  - t is a non-negative real number

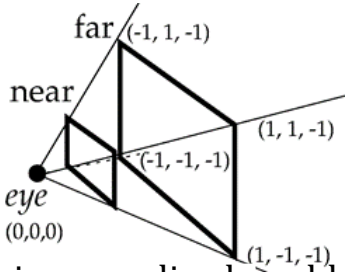


- "Eye point" is center of projection in perspective view volume (view frustum)
- Don't use de-perspectivizing (unhinging) step in order to avoid dealing with inverse of non-affine perspective transformation later on – stay tuned

Generate Primary Ray (2/4)

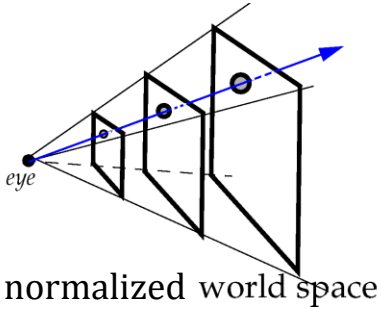
Ray Direction

- Start with 2D screen-space sample point ((sub)pixel)
- To create a ray from eye point through film plane, 2D screen-space point must be converted into a 3D point on film plane
- Note that ray generated will be intersecting objects in normalized world space coordinate system BEFORE the perspective transformation
- Any plane orthogonal to look vector is a convenient film plane (plane z = k in canonical frustum)



canonical frustum in normalized world coordinates

Any plane $z = k$, $-1 <= k < 0$ can be the film plane



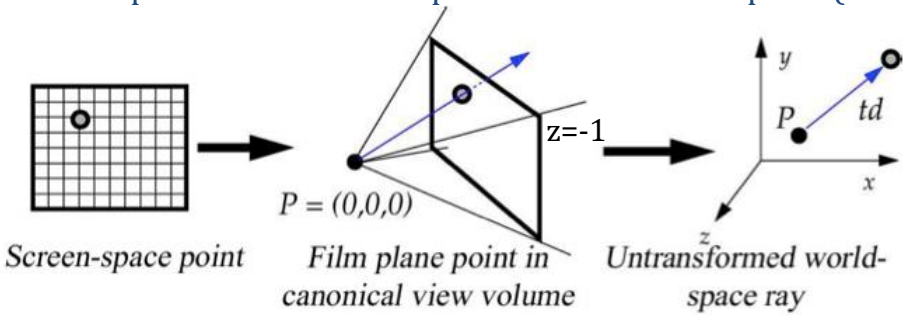
normalized world space

- Choose a film plane and then create a function that maps screen space points onto it
  - what's a good plane to use? Try the far clipping plane ☺ (z = - 1)
  - to convert, scale integer screen-space coordinates into floats between -1 and 1 for x and y, z = -1

Generate Primary Ray (3/4)

Ray Direction (continued)

- Once have a 3D point on the film plane, need to transform to pre-normalization world space where objects are
  - make a direction vector from eye point P (at center of projection) to 3D point on film plane
  - need this vector to be in world-space in order to intersect with original object in pre-normalization world space
  - because illumination model prefers intersection point to be in world-space (less work than normalizing lights)

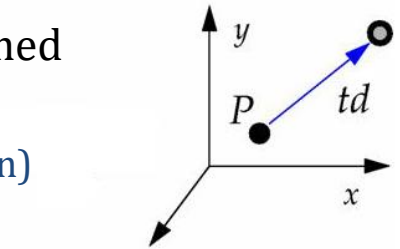


- Normalizing transformation maps world-space points to points in the canonical view volume
  - translate to the origin; orient so *Look* points down –Z, *Up* along Y; scale x and y to adjust viewing angles to 45˚, scale z: [-1, 0]; x, y: [-1, 1]
- How do we transform a point from the canonical view volume back to untransformed world space?
  - apply the inverse of the normalizing transformation: **Viewing Transformation**
    - Note: not same as viewing transform you use in OpenGL (e.g., ModelView matrix)

Generate Primary Ray (4/4)

Summary

- Start the ray at center of projection ("eye point")
- Map 2D integer screen-space point onto 3D film plane in normalized frustum
  - scale x, y to fit between -1 and 1
  - set z to -1 so points lie on the far clip plane
- Transform 3D film plane point (mapped pixel) to an untransformed world-space point
  - need to undo normalizing transformation (i.e., viewing transformation)
- Construct the direction vector
  - a point minus a point is a vector
  - direction = (world-space point (mapped pixel)) – (eye point (in untransformed world space))



Untransformed world-space ray

Ray-Object Intersection (1/5)

Implicit objects

- If an object is defined implicitly by a function *f* where *f(Q)* = IFF *Q* is a point on the surface of the object, then ray-object intersections are relatively easy to compute
  - many objects can be defined implicitly
  - implicit functions provide potentially infinite resolution
  - tessellating these objects is more difficult than using the implicit functions directly
- For example, a circle of radius *R* is an implicit object in a plane, with equation:

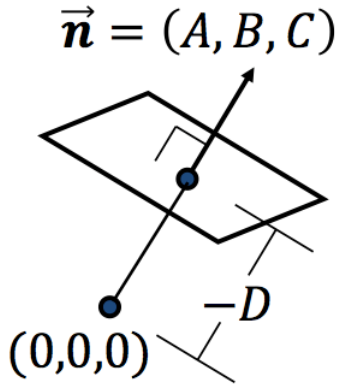
$$f(x,y) = x^2 + y^2 - R^2$$

  - *point (x,y) is on the circle when f(x,y) =*
- An infinite plane is defined by the function:

$$f(x,y,z) = Ax + By + Cz + D$$

- A sphere of radius *R* in 3-space:

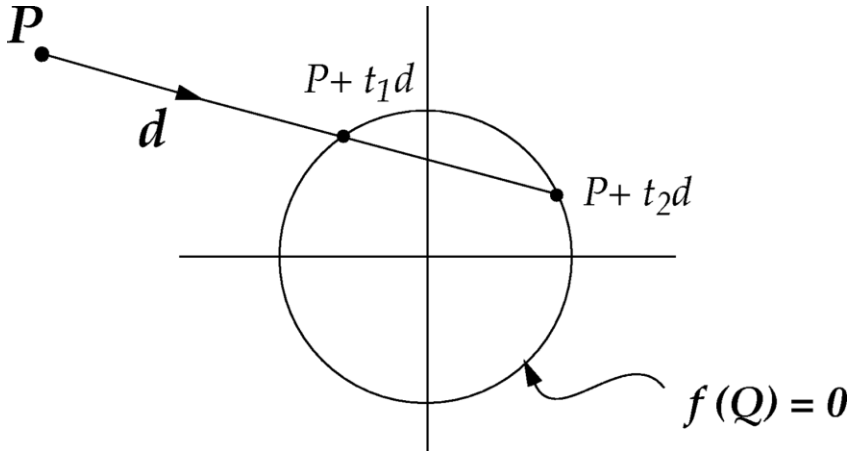
$$f(x,y,z) = x^2 + y^2 + z^2 - R^2$$



Ray-Object Intersection (2/5)

Implicit objects (continued)

- At what points (if any) does the ray intersect an object?
- Points on a ray have form $\boldsymbol{P} + t\boldsymbol{d}$ where t is any non-negative real number
- A surface point Q lying on an object has the property that $f(Q) =$
- Combining, we want to know "For which values of t is $f(\boldsymbol{P} + t\boldsymbol{d}) =$?"



- We are solving a system of simultaneous equations in x, y (in 2D) or x, y, z (in 3D)

An Explicit Example (1/3)

2D ray-circle intersection example

- Consider the eye-point $\boldsymbol{P}$ = (-3, 1), the direction vector $\boldsymbol{d}$ = (.8, -.6) and the unit circle: $\boldsymbol{f(x,y) = x^2 + y^2 - R^2}$
- A typical point of the ray is: $Q = \boldsymbol{P} + t\boldsymbol{d} = (-3,1) + t(.8,-.6) = (-3 + .8t, 1 - .6t)$
- Plugging this into the equation of the circle:
  $f(Q) = f(-3 + .8t, 1 - .6t) = (-3+.8t)^2 + (1-.6t)^2 - 1$
- Expanding, we get: $9 - 4.8t + .64t^2 + 1 - 1.2t + .36t^2 - 1$
- Setting this equal to zero, we get: $t^2 - 6t + 9 = 0$

An Explicit Example (2/3)

2D ray-circle intersection example (cont)

- Using the quadratic formula: $roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- We get: $t = \frac{6 \pm \sqrt{36 - 36}}{2}, \quad t = 3, 3$
- Because we have a root of multiplicity 2, ray intersects circle at only one point (i.e., it's tangent to the circle)
- Use discriminant D = $b^2 - 4ac$ to quickly determine if true intersection:
  - if $D < 0$, imaginary roots; no intersection
  - if $D = 0$, double root; ray is tangent
  - if $D > 0$, two real roots; ray intersects circle at two points
- Smallest non-negative real t represents the intersection nearest to eye-point

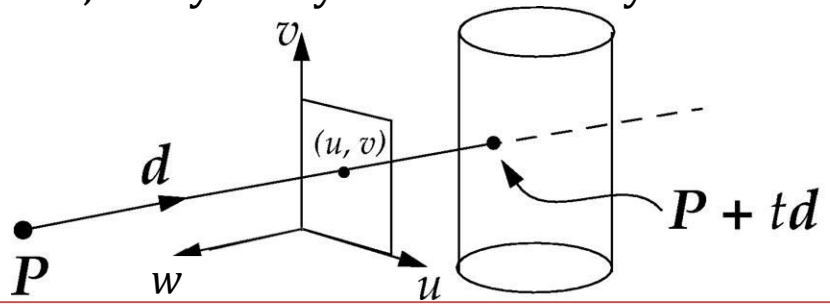
An Explicit Example (3/3)

2D ray-circle intersection example (continued)

- Generalizing:
  - we can take an arbitrary implicit surface with equation $f(Q) = 0$, a ray $\boldsymbol{P} + t\boldsymbol{d}$, and plug the latter into the former:

$$f(\boldsymbol{P} + t\boldsymbol{d}) = 0$$

- The result, after some algebra, is an equation with t as unknown
- We then solve for t, analytically or numerically



Ray-Object Intersection (3/5)

Implicit objects (continued) – multiple conditions

- For cylindrical objects, the implicit equation

$$f(x,y,z) = x^2 + z^2 - 1 = 0$$

in 3-space defines an infinite cylinder of unit radius running along the y-axis

- Usually, it's more useful to work with finite objects
  - e.g. a unit cylinder truncated with the limits:

cylinder body: $x^2 + z^2 - 1 = 0, -1 \leq y \leq 1$

- But how do we define cylinder "caps" as implicit equations?
- The caps are the insides of the cylinder at the cylinder's y extrema (or rather, a circle)
  - cylinder caps
    - top: $x^2 + z^2 - 1 \leq 0,\ y = 1$
    - bottom: $x^2 + z^2 - 1 \leq 0,\ y = -1$

Ray-Object Intersection (4/5)

Implicit objects (continued) – cylinder pseudocode

- Solve in a case-by-case approach

Ray_inter_finite_cylinder(P,d):

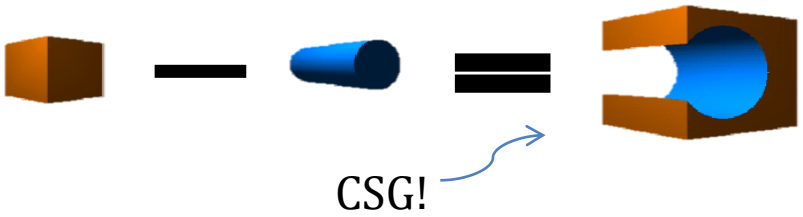
```
t1,t2 = ray_inter_infinite_cylinder(P,d)      // Check for intersection with infinite cylinder
compute P + t1*d, P + t2*d
if y > 1 or y < -1 for t1 or t2: toss it      // If intersection, is it between cylinder caps?

t3 = ray_inter_plane(plane y = 1)             // Check for an intersection with the top cap
Compute P + t3*d
if x^2 + z^2 > 1: toss out t3                 // If it intersects, is it within cap circle?

t4 = ray_inter_plane(plane y = -1)            // Check intersection with bottom cap
Compute P + t4*d
if x^2 + z^2 > 1: toss out t4                 // If it intersects, is it within cap circle?
```

Of all the remaining t's (t1 – t4), select the smallest non-negative one.
If none remain, ray does not intersect cylinder

Ray-Object Intersection (5/5)

Implicit surface strategy summary

- Substitute ray (***P*** + *t****d***) into implicit surface equations and solve for *t*
  - smallest non-negative *t*-value is from the closest surface you see from eye point
- For complicated objects (not defined by a single equation), write out a set of equalities and inequalities and then code individual surfaces as cases…
- Latter approach can be generalized cleverly to handle all sorts of complex combinations of objects
  - constructive Solid Geometry (CSG), where objects are stored as a hierarchy of primitives and 3-D set operations (union, intersection, difference) – don't have to evaluate the CSG to raytrace!
  - "blobby objects", which are implicit surfaces defined by sums of implicit equations (F(x,y,z)=0)



World Space Intersection

- To compute using illumination model, objects easiest to intersect in world space since normalizing lights with geometry is too hard and normals need to be handled. Thus need an analytical description of each object in world space
- Example: unit sphere translated to (3, 4, 5) after it was scaled by 2 in the x-direction has equation

$$f(x, y, z) = \frac{(x-3)^2}{2^2} + (y-4)^2 + (z-5)^2 = 0.5^2$$

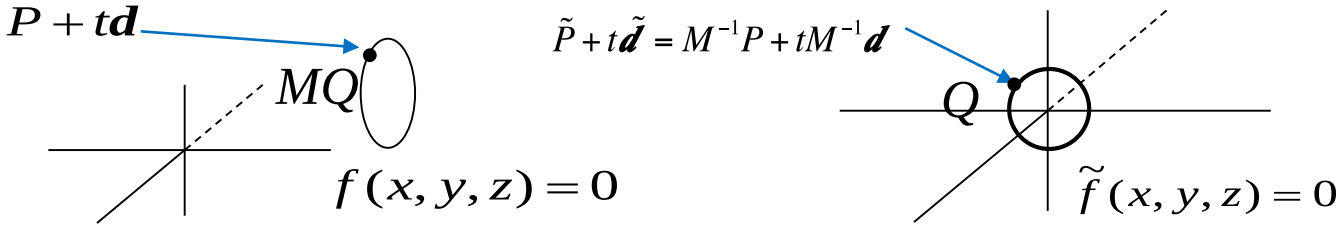
- Can take ray $\boldsymbol{P}+t\boldsymbol{d}$ and plug it into the equation, solving for t

$$f(\boldsymbol{P} + t\boldsymbol{d}) = 0$$

- But intersecting with an object arbitrarily transformed by modeling transforms is difficult
  - intersecting with an untransformed shape in object's original object coordinate system is much easier
  - can take advantage of transformations to change intersection problem from world space (arbitrarily transformed shape) to object space (untransformed shape)

Object Space Intersection

To get *t*, transform ray into object space and solve for intersection there



- Let the world-space intersection point be defined as *MQ*, where *Q* is a point in object space:

$$P + td = MQ$$
$$M^{-1}(P + td) = Q$$
$$M^{-1}P + tM^{-1}d = Q$$
$$\text{Let } \tilde{P} = M^{-1}P, \tilde{d} = M^{-1}d$$

- If is the equation of the untransformed object, we just have to solve $f(\tilde{P} + t\tilde{d}) = 0$
  - note: $\tilde{d}$ is probably **not** a unit vector
  - the parameter *t* along this vector and its world space counterpart always have the same value.
  - normalizing $\tilde{d}$ would alter this relationship. **Do NOT normalize $\tilde{d}$**

World Space vs. Object Space

- To compute a world-space intersection, we have to transform the implicit equation of a canonical object defined in object space - *often difficult*
- To compute intersections in object space, we only need to apply a matrix (M^{-1}) to $\boldsymbol{P}$ and $\boldsymbol{d}$ - *much simpler, but* does M^{-1} always exist?
  - M was composed from two parts: the cumulative modeling transformation that positions the object in world-space, and the camera's normalizing transformation (not including the perspective transformation!)
  - modeling transformations are comprised of translations, rotations, and scales (all invertible)
  - normalizing transformation consists of translations, rotations and scales (also invertible); but the perspective transformation (which includes the homogenization divide) is not invertible! (This is why we used canonical frustum directly rather than de-perspectivizing/unhinging it)
- When you're done, you get a t-value
- This t can be used in two ways:
  - $\boldsymbol{P} + t\boldsymbol{d}$ is the world-space location of the intersection between ray and transformed object
  - $\widetilde{\boldsymbol{P}} + t\widetilde{\boldsymbol{d}}$ is the corresponding point on untransformed object (in object space)

Normal Vectors at Intersection Points (1/4)

Normal vector to implicit surfaces

- For illumination (diffuse and specular), need normal at point of intersection in world space
- Instead, start by solving for point of intersection in object's own space and compute the normal there. Then transform this object space normal to world space

- If a surface bounds a solid whose interior is given by

$$f(x, y, z) < 0$$

- Then we can find a normal vector at point (x, y, z) via *gradient* at that point:

$$\mathrm{n} = \nabla f(x, y, z)$$

- Recall that the gradient is a vector with three components, the partial derivatives:

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}(x, y, z), \frac{\partial f}{\partial y}(x, y, z), \frac{\partial f}{\partial z}(x, y, z) \right)$$

Normal Vectors at Intersection Points (2/4)

- For the sphere, the equation is:

$$f(x,y,z) = x^2 + y^2 + z^2 - 1$$

- Partial derivatives are

$$\frac{\partial f}{\partial x}(x, y, z) = 2x$$

$$\frac{\partial f}{\partial y}(x, y, z) = 2y$$

$$\frac{\partial f}{\partial z}(x, y, z) = 2z$$

- So gradient is

$$\boldsymbol{n} = \nabla f(x, y, z) = (2x, 2y, 2z)$$

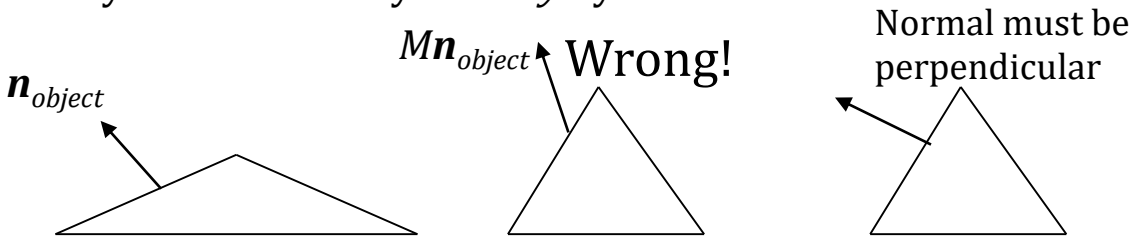
- Remember to normalize ***n*** before using in dot products!
- In some degenerate cases, gradient may be zero and this method fails…nearby gradient is cheap hack

Normal Vectors at Intersection Points (3/4)

Transforming back to world space

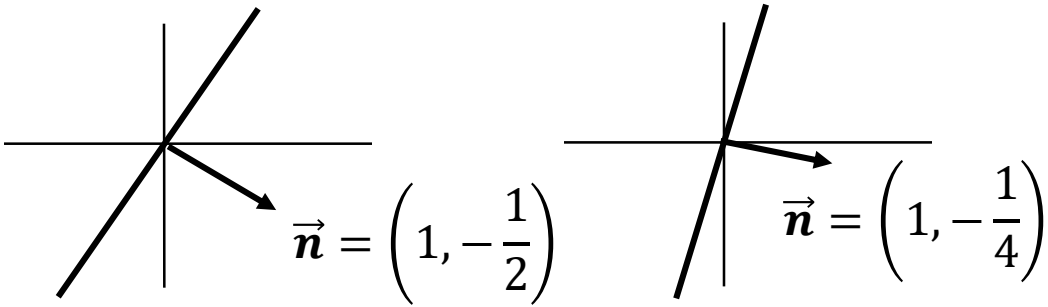
- We now have an object-space normal vector
- We need a world-space normal vector for the illumination equation
- To transform an object to world coordinates, we just multiplied its vertices by *M*, the object's CTM
- Can we do the same for the normal vector?
  - answer: NO $\boldsymbol{n}_{world} \neq M\boldsymbol{n}_{object}$

- Example: say *M* scales in *x* by .5 and *y* by 2



Normal Vectors at Intersection Points (4/4)

- Why doesn't multiplying normal by *M* work?
- For translation and rotation, which are rigid body transformations, it actually does work fine
- Scaling, however, distorts normal in exactly opposite sense of scale applied to surface
  - scaling y by a factor of 2 causes the normal to scale by .5:



- We'll see this algebraically in the next slides

Transforming Normals (1/4)

Object-space to world-space

- As an example, let's look at polygonal case
- Let's compute relationship between object-space normal $\boldsymbol{n}_{obj}$ to polygon H and world-space normal $\boldsymbol{n}_{world}$ to transformed version of H, called MH
- For any vector $\boldsymbol{v}$ in world space that lies in the polygon (e.g., one of its edge vectors), normal is perpendicular to $\boldsymbol{v}$:

$$\boldsymbol{n}_{world} \bullet \boldsymbol{v}_{world} = 0$$

- But $\boldsymbol{v}_{world}$ is just a transformed version of some vector in object space, $\boldsymbol{v}_{obj}$. So we could write:

$$\boldsymbol{n}_{world} \bullet M\boldsymbol{v}_{obj} = 0$$

- Recall that since vectors have no position, they are unaffected by translations (thus they have $w = 0$)
  - so to make things easier, we only need to consider:

M_3 = upper left 3 x 3 of M (rotation/scale component)

$$\mathbf{n}_{world} \bullet M_3 \mathbf{v}_{obj} = 0$$

Transforming Normals (2/4)

Object-space to world-space (continued)

- So we want a vector $\boldsymbol{n}_{world}$ such that for any $\boldsymbol{v}_{obj}$ in the plane of the polygon:

$$\mathrm{n}_{world} \bullet M_3 \mathrm{v}_{obj} = 0$$

- We will show on next slide that this equation can be rewritten as:

$$M_3^{\,t} \boldsymbol{n}_{world} \bullet \boldsymbol{v}_{obj} = 0$$

- We also already have:

$$\boldsymbol{n}_{obj} \bullet \boldsymbol{v}_{obj} = 0$$

- Therefore:

$$M_3^{\,t} \boldsymbol{n}_{world} = \boldsymbol{n}_{obj}$$

- Left-multiplying by $(M_3^t)^{-1}$,

$$\boldsymbol{n}_{world} = \left(M_3^{\,t}\right)^{-1} \boldsymbol{n}_{obj}$$

Transforming Normals (3/4)

Object-space to world-space (continued)

- So how did we rewrite this:

$$\boldsymbol{n}_{world} \bullet M_3 \boldsymbol{v}_{obj} = 0$$

- As this:

$$(M_3{}^t \boldsymbol{n}_{world}) \bullet \boldsymbol{v}_{obj} = 0$$

- Recall that if we think of vector as n x 1 matrices, then switching notation:

$$\boldsymbol{a} \bullet \boldsymbol{b} = \boldsymbol{a}^t \boldsymbol{b}$$

- Rewriting our original formula, yielding:

$$\boldsymbol{n}^t{}_{world} M_3 \boldsymbol{v}_{obj} = 0$$

- Writing $M = M^{tt}$, we get:

$$\boldsymbol{n}^t{}_{world} M_3{}^{tt} \boldsymbol{v}_{obj} = 0$$

- Recalling that $(AB)^t = B^t A^t$, we can write:

$$(M_3{}^t \underline{\boldsymbol{n}_{world}})^t \boldsymbol{v}_{obj} = 0$$

- Switching back to dot product notation, our result:

$$(M_3{}^t \boldsymbol{n}_{world}) \bullet \boldsymbol{v}_{obj} = 0$$

Transforming Normals (4/4)

Applying inverse-transpose of M to normals

- So we ended up with: $\boldsymbol{n}_{world} = (M_3{}^t)^{-1}\boldsymbol{n}_{obj}$
- "Invert" and "transpose" can be swapped, to get our final form: $\boldsymbol{n}_{world} = (M_3{}^{-1})^t\boldsymbol{n}_{obj}$
- Why do we do this? It's easier!
  - instead of inverting composite matrix, accumulate composite of inverses which are easy to take for each individual transformation
- A hand-waving interpretation of $(M_3{}^{-1})^t$
  - M_3 is composition of rotations and scales, R and S (why no translates?). Therefore

$$((RS\ldots)^{-1})^t = (\ldots S^{-1}R^{-1})^t = ((R^{-1})^t(S^{-1})^t\ldots)$$

  - so we're applying transformed (inverted, transposed) versions of each individual matrix in original order
  - for rotation matrix, transformed version equal to original rotation, i.e., normal rotates with object
    - $(R^{-1})^t = R$; inverse reverses rotation, and transpose reverses it back
  - for scale matrix, inverse inverts scale, while transpose does nothing:
    - $(S(x,y,z)^{-1})^t = S(x,y,z)^{-1} = S(1/x,1/y,1/z)$

Summary - putting it all together

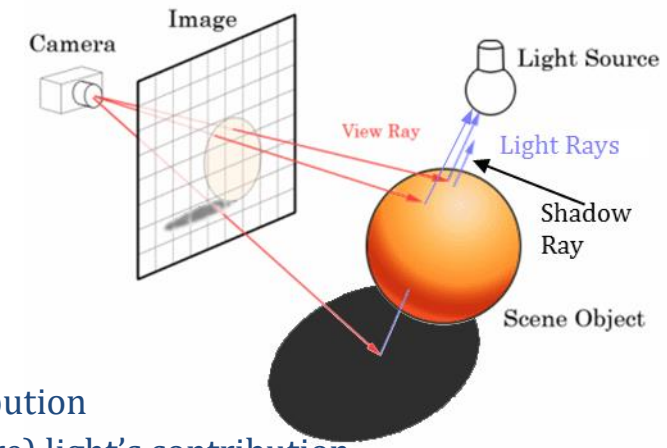
Simple, non-recursive raytracer

```
P = eyePt

for each sample of image:
    Compute d
    for each object:
         Intersect ray P+td with object

    Select object with smallest non-negative t-value (visible object)

    For this object, find object space intersection point

    Compute normal at that point
    Transform normal to world space

    Use world space normal for lighting computations
```

Shadows

- Each light in the scene contributes to the color and intensity of a surface element…

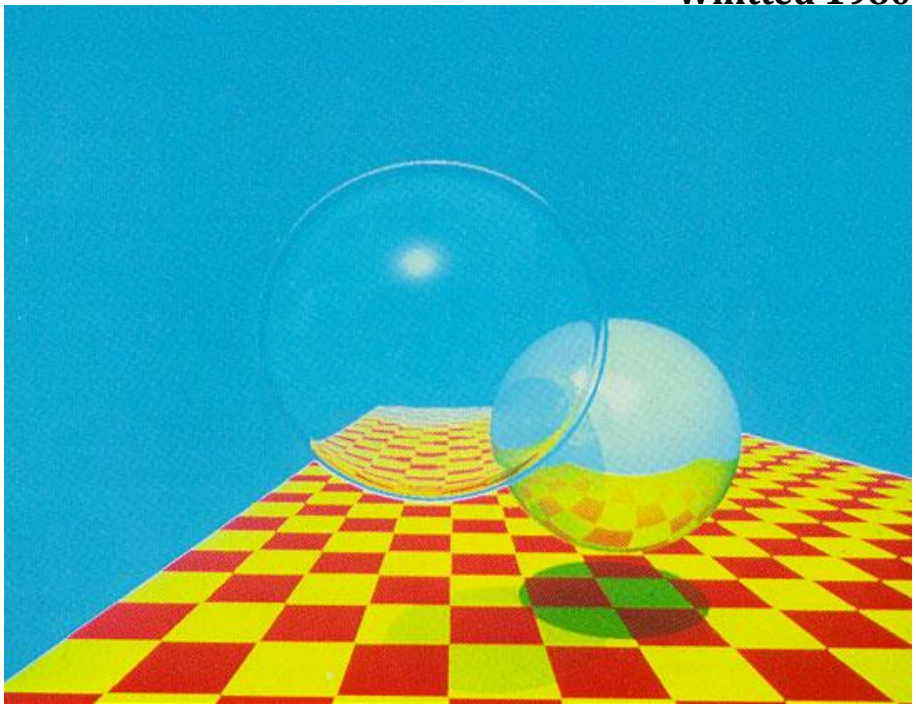
$$objectIntensity_\lambda = ambient + \sum_{light=1}^{numLights} attenuation \cdot lightIntensity_\lambda \cdot [diffuse + specular]$$



- **If and only if** light source reaches the object!
  - could be occluded/obstructed by other objects in scene
  - could be self-occluding
- Construct a ray from the surface intersection to each light
- Check if light is first object intersected
  - if first object intersected **is** the light, **count** light's full contribution
  - if first object intersected **is not** the light, **do not count** (ignore) light's contribution
  - this method generates hard shadows; soft shadows are harder to compute (must sample)
- What about transparent or specular (reflective) objects? Such lighting contributions are the beginning of global illumination => need recursive ray tracing

Recursive Ray Tracing Example

Whitted 1980

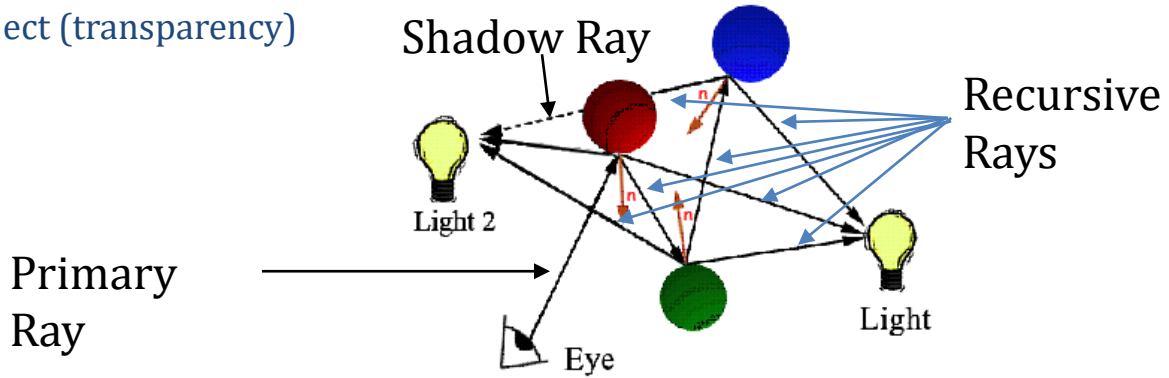


- Ray traced image with recursive ray tracing: transparency and refractions

Recursive Ray Tracing (1/4)

Simulating global lighting effects (Whitted, 1980)

- By recursively casting new rays into the scene, we can look for more information
- Start from point of intersection with object
- We'd like to send rays in all directions , but that's too hard/computationally taxing
- Instead, just send rays in directions likely to contribute most:
  - toward lights (blockers to lights create shadows for those lights)
  - specular bounce off other objects to capture specular inter-object reflections
  - use ambient hack to capture diffuse inter-object reflection
  - refractive rays through object (transparency)



Recursive Ray Tracing (2/4)

- Trace "secondary" rays at intersections:
  - **light**: trace a ray to each light source. If light source is blocked by an opaque object, it does not contribute to lighting
  - **specular reflection**: trace reflection ray (i.e., about normal vector at surface intersection)
  - **refractive transmission/transparency**: trace refraction ray (following Snell's law)
  - recursively spawn new light, reflection, and refraction rays at each intersection until contribution negligible or some max recursion depth is reached
- Limitations
  - recursive inter-object reflection is strictly specular
  - diffuse inter-object reflection is handled by other techniques
- Oldies-but-goodies
  - *Ray Tracing: A Silent Movie*
  - *Quest: A Long Ray's Journey into Light*

Recursive Ray Tracing (3/4)

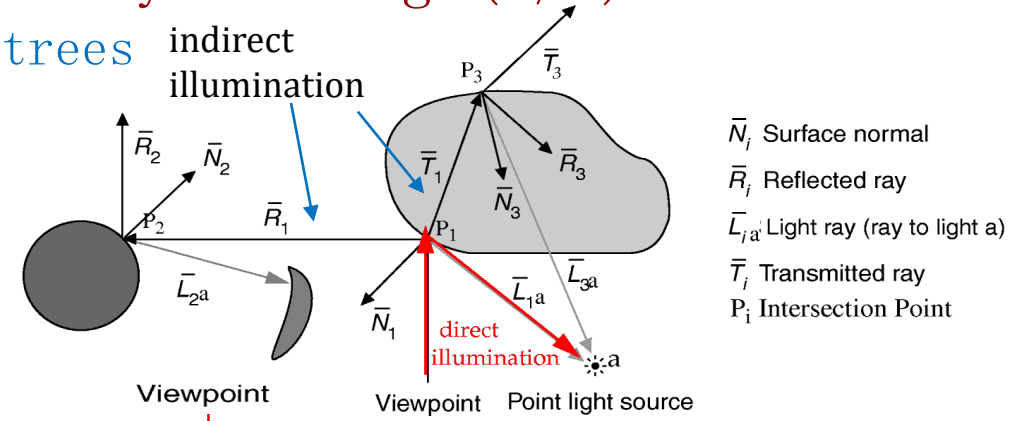
- Your new lighting equation (Phong lighting + specular reflection + transmission):

$$I_\lambda = \underbrace{L_{a\lambda} k_a O_{a\lambda}}_{ambient} + \sum_{lights} f_{att} L_{p\lambda} [\underbrace{k_d O_{d\lambda} (\vec{n} \bullet \vec{l})}_{diffuse} + \underbrace{k_s O_{s\lambda} (\vec{r} \bullet \vec{v})^n}_{specular}] + \underbrace{\underbrace{k_s O_{s\lambda} I_{r\lambda}}_{reflected} + \underbrace{k_t O_{t\lambda} I_{t\lambda}}_{transmitted}}_{recursive\ rays}$$

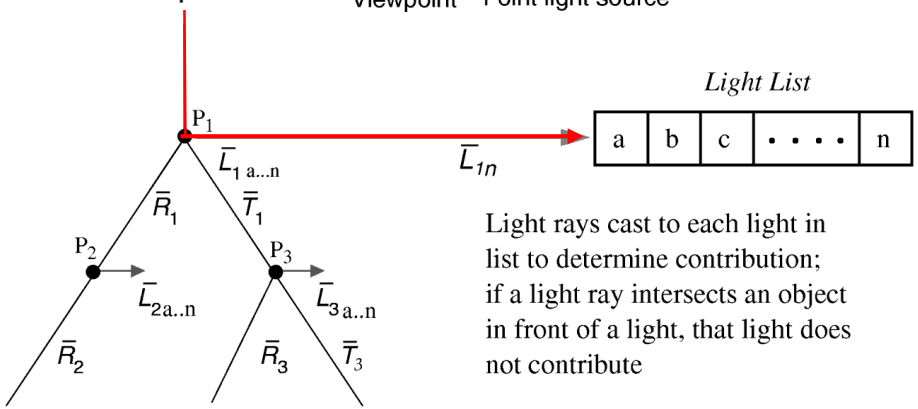
  - I is the total color at a given point (lighting + specular reflection + transmission, λ subscript for each r,g,b)
    - Its presence in the transmitted and reflected terms implies recursion
  - L is the light intensity; L_P is the intensity of a point light source
  - k is the attenuation coefficient for the object material (ambient, diffuse, specular, etc.)
  - O is the object color
  - f_{att} is the attenuation function for distance
  - $\boldsymbol{n}$ is the normal vector at the object surface
  - $\boldsymbol{l}$ is the vector to the light
  - $\boldsymbol{r}$ is the reflected light vector
  - $\boldsymbol{v}$ is the vector from the eye point (view vector)
  - n is the specular exponent
  - note: intensity from recursive rays calculated with the same lighting equation at the intersection point
  - light sources contribute specular and diffuse lighting
- Note: single rays of light do not attenuate with distance; purpose of f_{att} is to simulate diminishing intensity per unit area as function of distance for point lights (typically an inverse quadratic polynomial)

Recursive Ray Tracing (4/4)

Light-ray trees



per ray:



Light rays cast to each light in list to determine contribution; if a light ray intersects an object in front of a light, that light does not contribute

Transparent Surfaces (1/2)

Non-refractive transparency

- For a partially transparent polygon

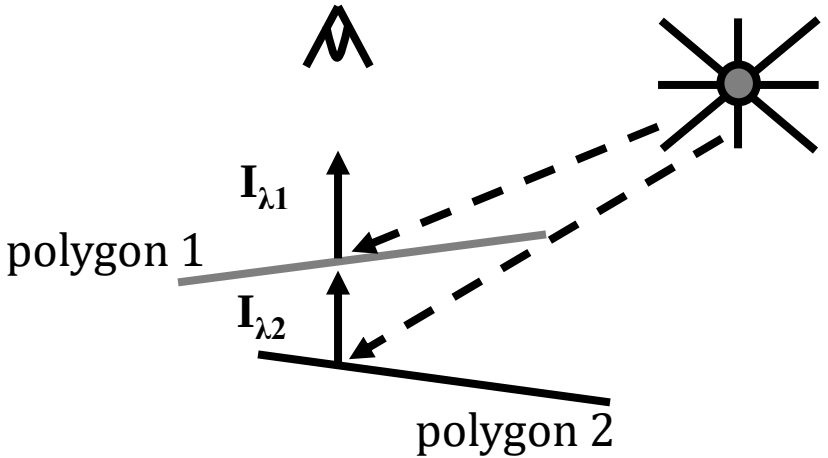
$$I_\lambda = (1-k_{t1})I_{\lambda 1} + k_{t1}I_{\lambda 2}$$

k_{t1} = transmittance of polygon 1

$(0 = \text{opaque}; 1 = \text{transparent})$

$I_{\lambda 1}$ = intensity calculated for polygon 1

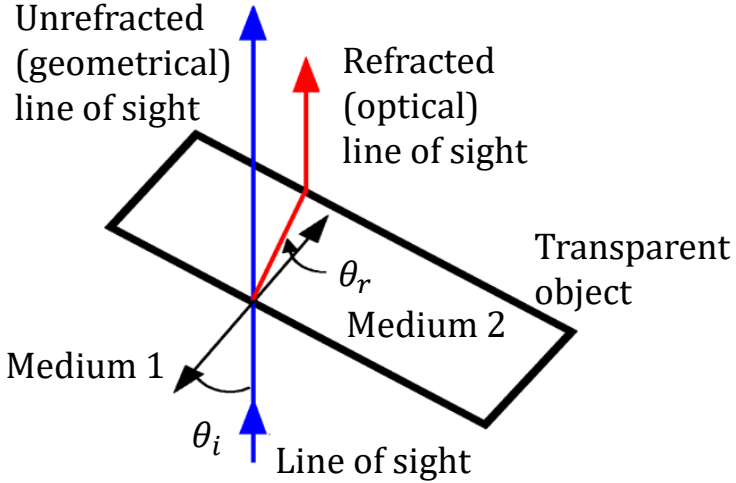
$I_{\lambda 2}$ = intensity calculated for polygon 2



Transparent Surfaces (2/2)

Refractive transparency

- We model the way light bends at interfaces with Snell's Law



$$\sin\theta_r = \frac{\sin\theta_i \eta_{i\lambda}}{\eta_{r\lambda}}$$

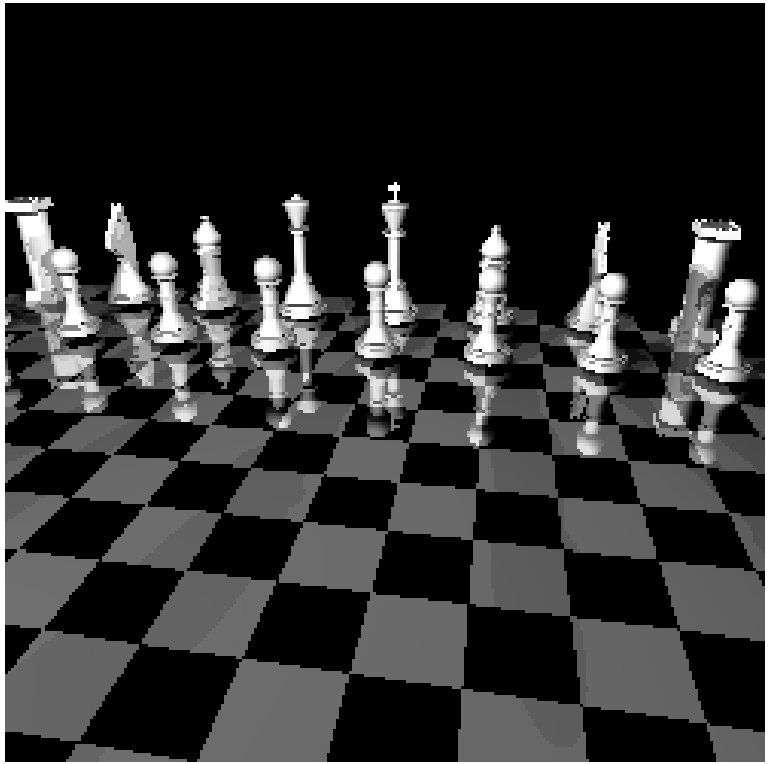
$\eta_{i\lambda}$ = index of refraction of medium 1

$\eta_{r\lambda}$ = index of refraction of medium 2

Choosing Samples (1/2)

Sampling

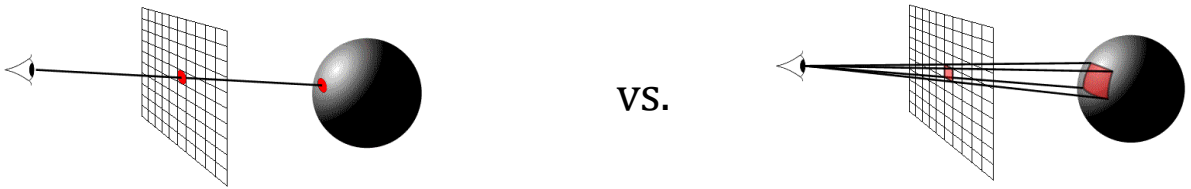
- In both the examples and in your assignments we sample once per pixel. This generates images similar to this:



- We have a clear case of the jaggies
- Can we do better?

Choosing Samples (2/2)

- In the simplest case, sample points are chosen at pixel centers
- For better results, *supersamples* can be chosen *(called supersampling)*
  - e.g., at corners of pixel as well as at center
- Even better techniques do *adaptive sampling*: increase sample density in areas of rapid change (in geometry or lighting)
- With *stochastic sampling*, samples are taken probabilistically (recall Image Processing IV slides)
  - Actually converges on "correct" answer faster than regularly spaced sampling
- For fast results, we can *subsample*: fewer samples than pixels
  - take as many samples as time permits
  - *beam tracing*: track a bundle of neighboring rays together
- How do we convert samples to pixels? Filter to get weighted average of all the samples per pixel!



Instead of sampling one point, sample within a region to create a better approximation

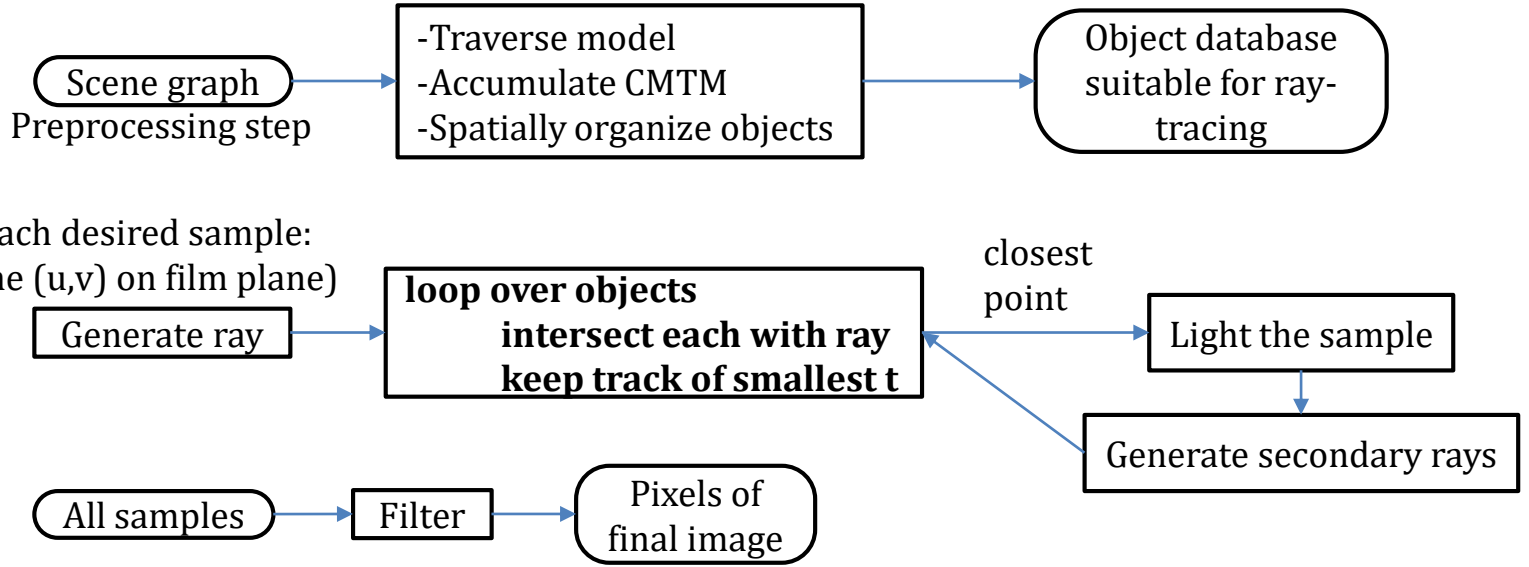
Supersampling example

With Supersampling

Without Supersamping

Ray Tracing Pipeline

- Raytracer produces visible samples from model
  - samples convolved with filter to form pixel image
- Additional pre-processing
  - pre-process database to speed up per-sample calculations
  - For example: organize by spatial partitioning via bins and/or bounding boxes (k-d trees, octrees, etc. - upcoming)



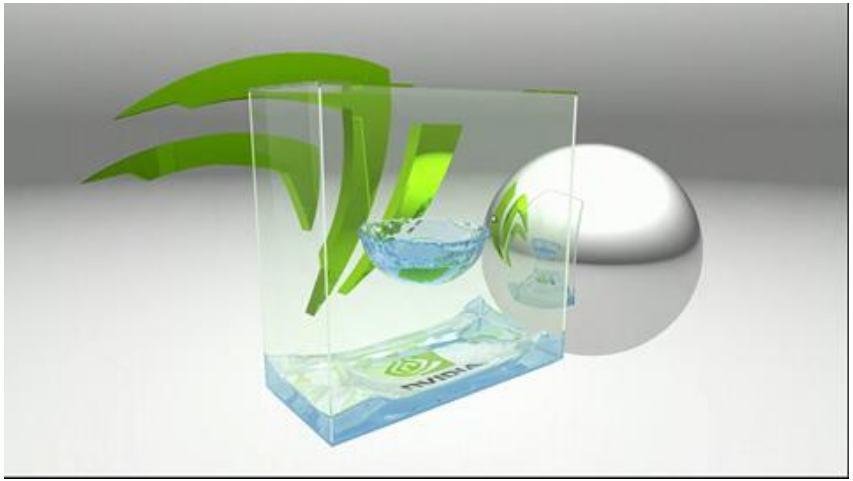
Real-time Ray Tracing (1/2)

- Traditionally, ray tracing was computationally impossible to do in real-time
  - "embarrassing" parallelism due to independence of each ray, so one CPU or core/pixel
  - hard to make hardware optimized for ray tracing:
    - large amount of floating point calculations
    - complex control flow structure
    - complex memory access for scene data
- One solution: software-based, highly optimized raytracer using cluster with multiple CPUs
  - Prior to ubiquitous GPU-based methods, ray tracing on CPU clusters to take advantage of parallelism
  - hard to have widespread adoption because of size and cost
  - Can speed up with more cores per CPU
- Large CPU render still dominant for non-real time CGI for movies and animations
  - Weta Digital (Planet of the Apes), ILM (Jurassic World), etc.

OpenRT rendering of five maple trees and 28,000 sunflowers (35,000 triangles) on 48 CPUs

Real-time Ray Tracing (2/2)

- Modern solution: Use GPUs to speed up ray tracing
  - NVIDIA Kepler architecture capable of real time ray tracing
    - Demo at GTC 2012: http://youtu.be/h5mRRElXy-w?t=2m5s



Scene ray traced in real time using NVIDIA Kepler

GPU Ray Tracing

2015 NVIDIA Iray Demo

- NVIDIA Optix framework built on top of CUDA
  - CUDA is a parallel computing platform for nVidia GPUs
  - Allows C/C++ and Fortran code to be compiled and run on nVidia GPUs
  - Optix is a programmable ray tracing framework that allows developers to quickly build ray tracing applications
- You can run the demos and download the SDKs yourself
  - http://developer.nvidia.com/optix
  - http://developer.nvidia.com/cuda

GPU Ray Tracing

- GLSL shaders (like the ones you've been writing in lab) can be used to implement a ray tracer
- OpenCL is similar to CUDA and can be used to run general purpose programs on a GPU, including a ray tracer
  - http://www.khronos.org/opencl/
- Weta Digital (Iron Man, Fantastic Four, Hunger Games) moving to GPU clusters for better efficiency

POV-Ray: Pretty Pictures

Free Advanced Raytracer

- Full-featured raytracer available online: povray.org
- Obligatory pretty pictures (see hof.povray.org):

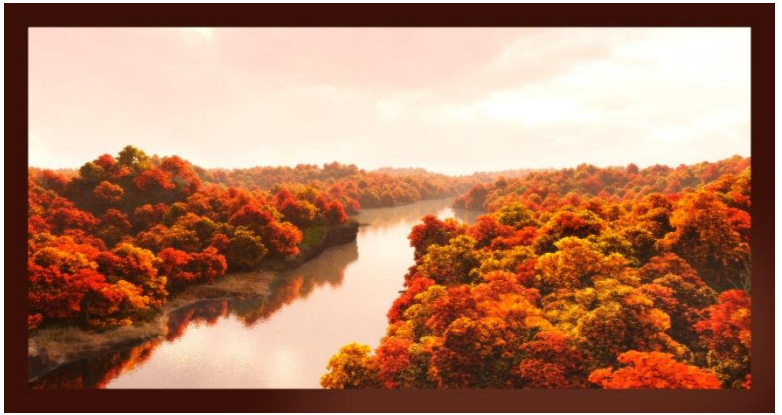


Image credits can be found on hof.povray.org