Computer Graphics Lecture 07: Ray Tracing

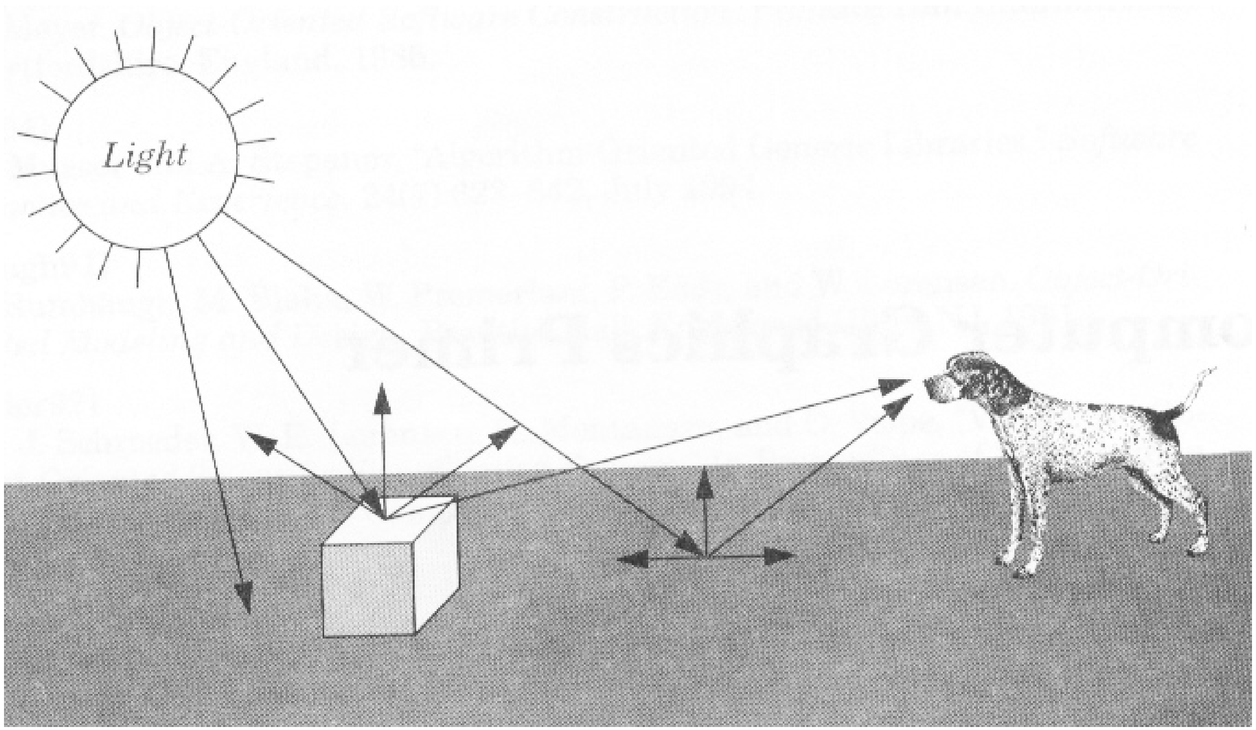
DR. ELVIS S. LIU

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

SPRING 2018

Rendering

- A process of generating an image from a 2D or 3D model, by means of computer programs

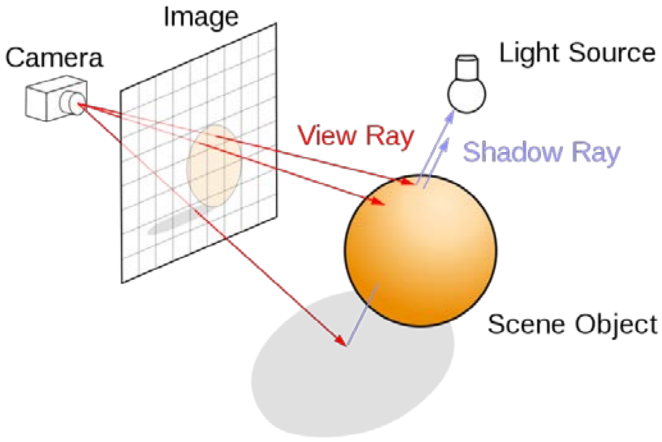


Rendering (cont.)

- "Rendering" refers to the entire process that produces color values for pixels, given a 3D representation of the scene
- Pixels correspond to rays; need to figure out the visible scene point along each ray
    - Called "hidden surface problem" in older texts
    - "Visibility" is a more modern term
    - Also, we assume (for now) a single ray per pixel
- Major algorithms: Ray casting

Rendering Methods

- Forward tracing
  - Consider every photon emitted by light source
  - Inefficient and extremely slow
  - Large number of light rays
- Backward tracing (ray tracing)
  - Trace hypothetical photo backwards from eye in particular direction
  - Slow
  - Number of light rays equals to the screen resolution
- Surface rendering
  - What light shines on surface?
  - How does material interact with light?
  - What part of result is visible to eye?

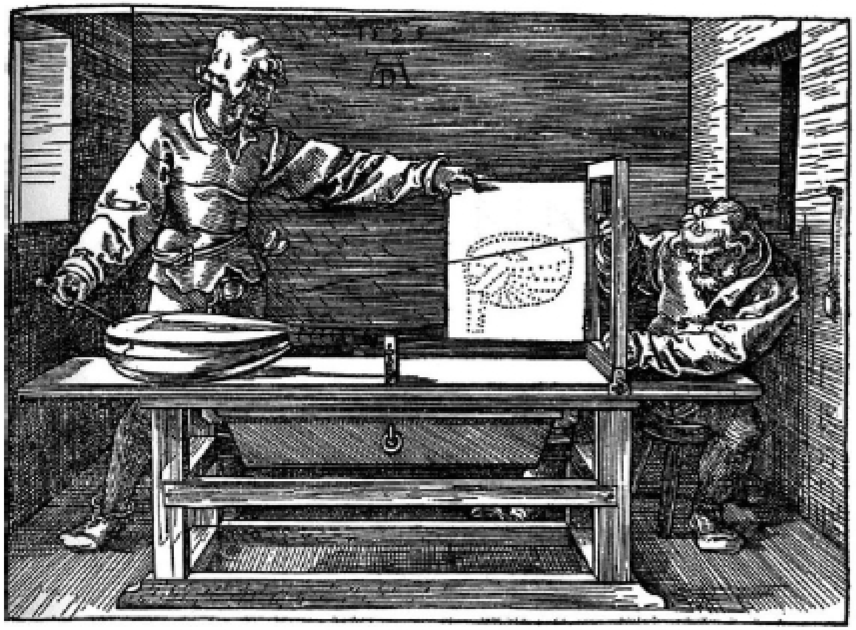


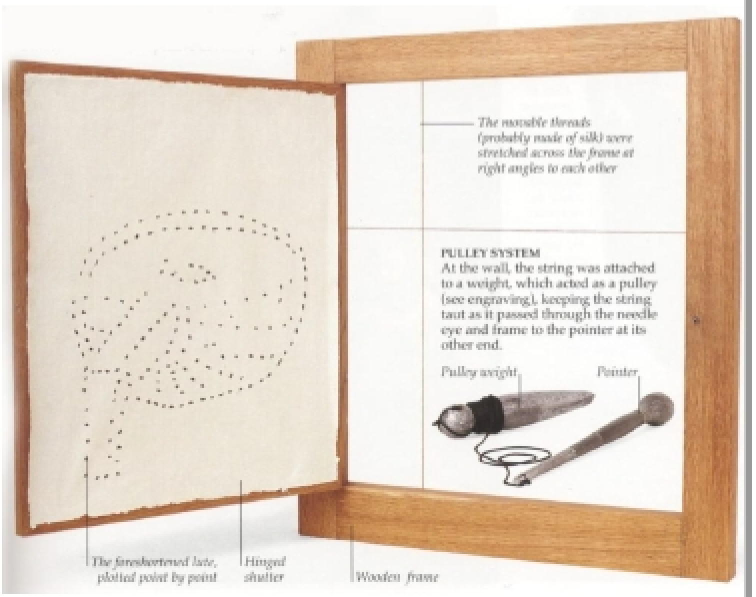
Origins of Ray Tracing

LECTURE 07: RAY TRACING

Dürer’s Ray Casting Machine

- Generalising from Albrecht Dürer’s wood cut (16th Century) showing perspective projection
- Durer: Record string intersection from center of projection (eye), to nearest object as points on a 2D plane



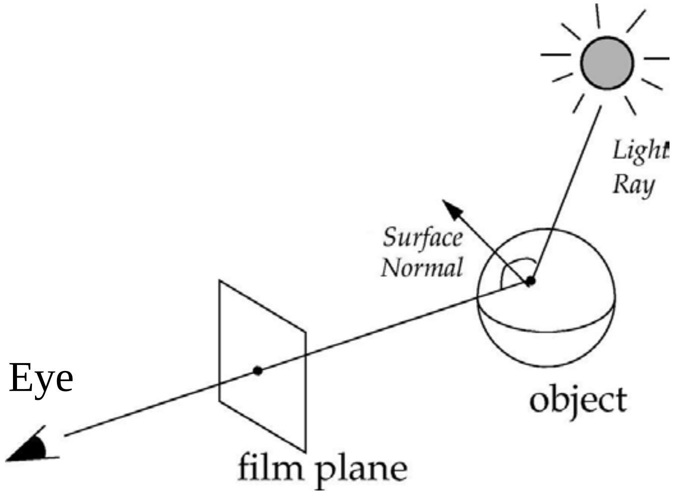


Dürer's Ray Casting Machine (cont.)

- 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
  - Avoids 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
- Generalising 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

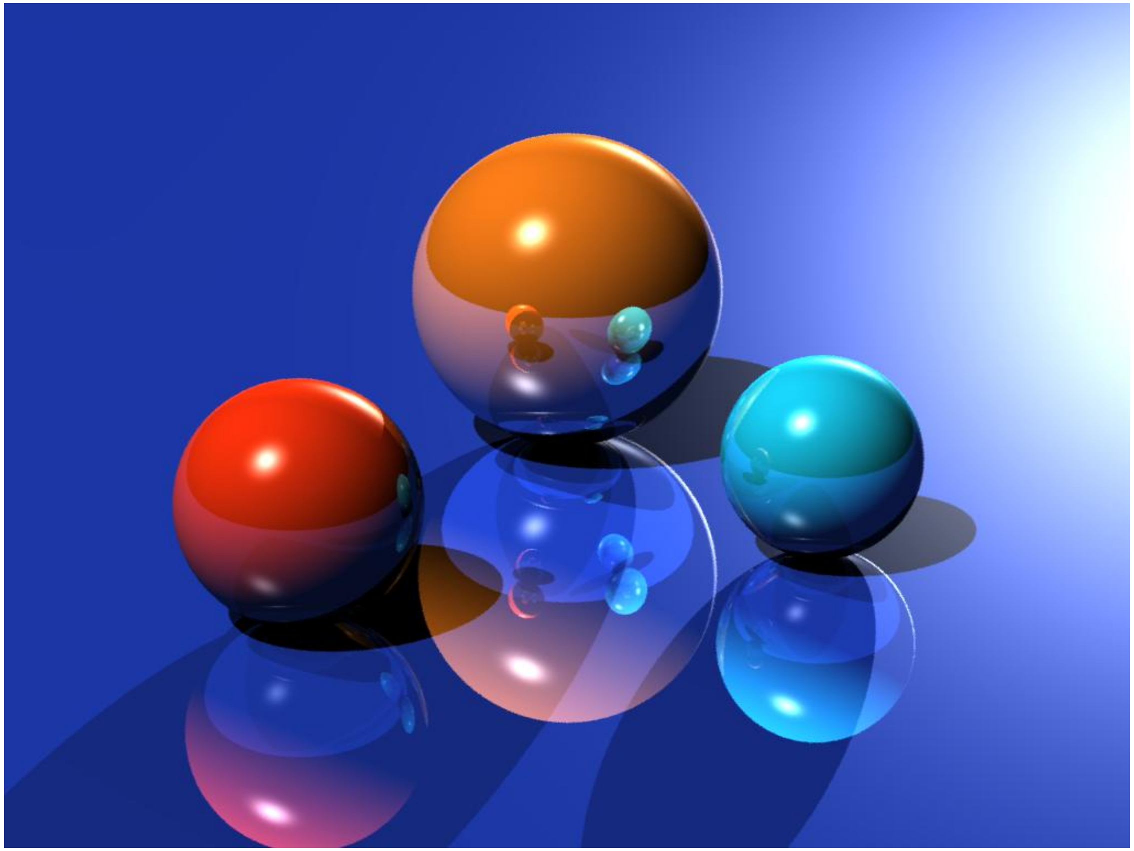
Ray Tracing Fundamentals (cont.)

- Calculate lighting (i.e., colour)
  - 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; RRT (Recursive Ray Tracing) only uses specular reflection rays because they are easy to compute and specular reflections are typically brighter than average
  - Sum of contributions determines color of sample point
  - No diffuse reflection rays → RRT is limited approximation to global illumination
- Finesse need for shading rule – ray-trace for lighting equation evaluation at each sample point

Ray Casting vs. Ray Tracing

- Ray Casting: eye rays only
- Ray Tracing: consider also secondary rays which are used for testing shadows, doing reflections, refractions, etc.

Ray Tracing Examples



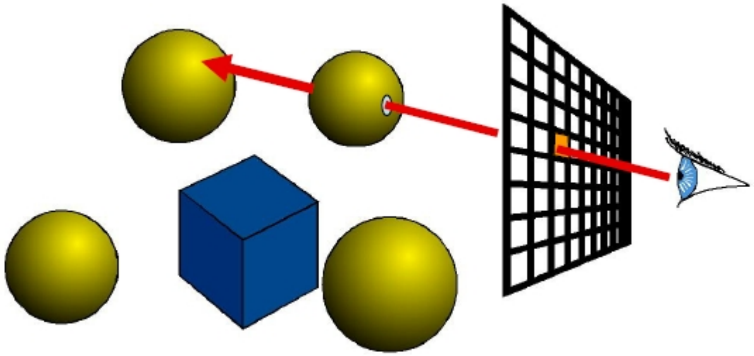
Ray Tracing Examples (cont.)

Ray Generation

LECTURE 07: RAY TRACING

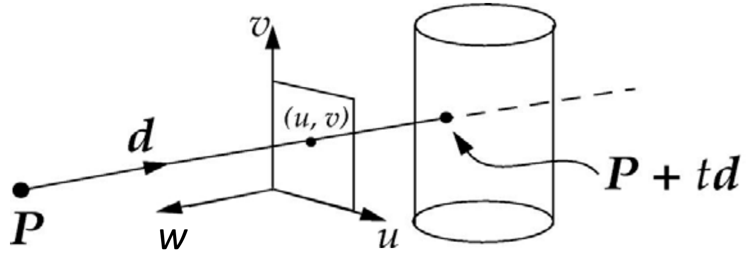
Simple Ray Casting

- For every pixel
  - Construct a ray from the eye
  - For every object in the scene
    - Find intersection with the ray
    - Keep if closest



Ray Origin

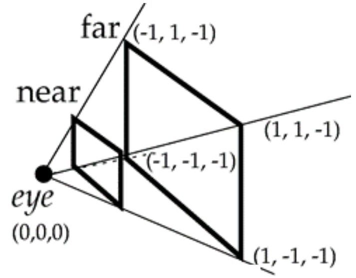
- Consider the geometry of the problem in normalised world space with canonical perspective frustum (i.e., do not apply perspective transformation)
- Start a ray from “eye point” **P**
- Shoot ray in some direction **d** from **P** toward a point in film plane (a rectangle in the **u-v** plane in the 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



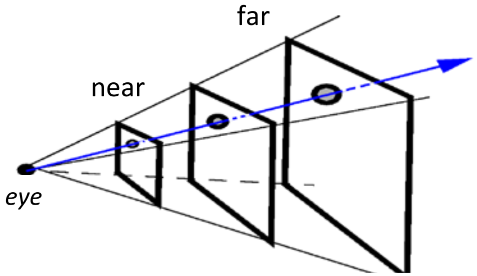
- “Eye point” **P** is center of projection in perspective view volume (view frustum)

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. Then we can get the direction vector by subtracting the COP from the 3D screen space point
- Note that ray generated will be intersecting objects in normalised world space coordinate system before the perspective transformation



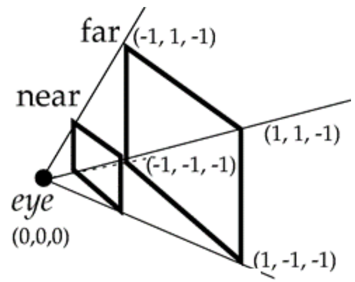
canonical frustum in normalized world coordinates
Any plane $z = k$, $-1 <= k < 0$ can be the film plane



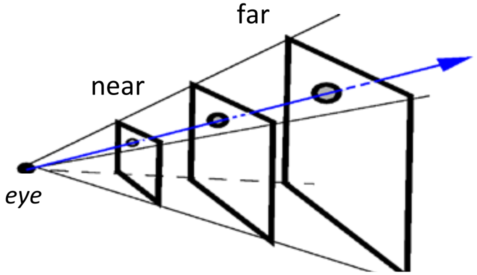
Normalised world space

Ray Direction (cont.)

- Any plane orthogonal to look vector is a convenient film plane (plane z = k in canonical frustum)
- Choose a film plane and then create a function that maps screen integer (u, v) space points onto it to floats
    - 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



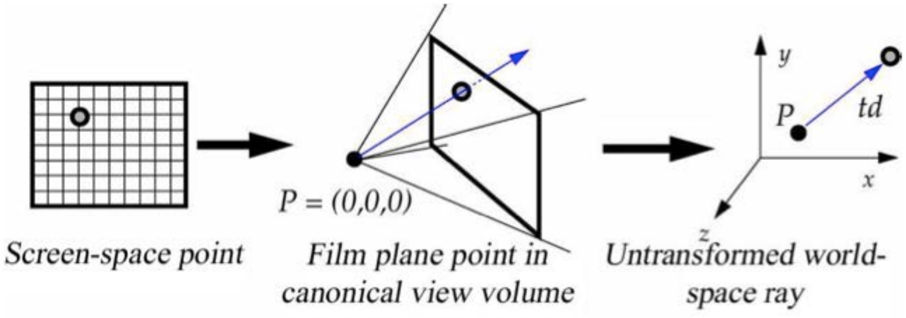
canonical frustum in normalized world coordinates
Any plane $z = k$, $-1 <= k < 0$ can be the film plane



Normalised world space

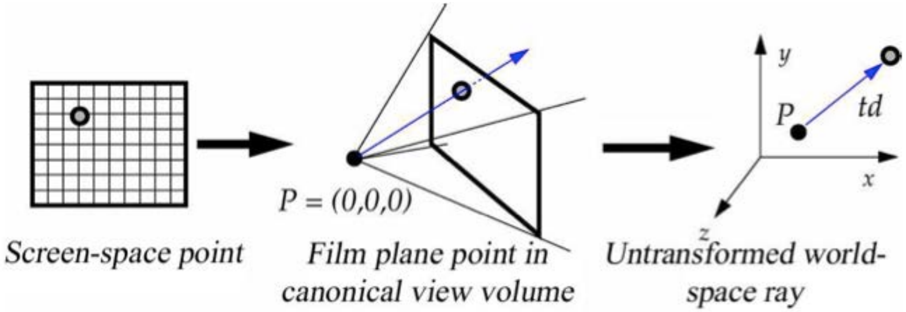
Ray Direction (cont.)

- Once have a 3D point on the film plane, need to transform to pre-normalisation world space where objects are
  - Why? illumination model prefers intersection point to be in world-space (less work than normalising lights)
  - 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-normalisation world space



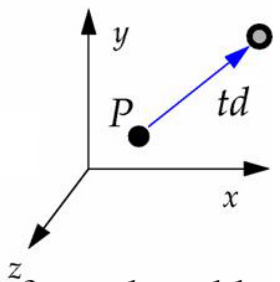
Ray Direction (cont.)

- Normalising 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 normalising transformation: Viewing Transformation
    - Note: not same as viewing transform you use in OpenGL (e.g., ModelView matrix)



Summary

- Start the ray at center of projection ("eye point")
- Map 2D integer screen-space point onto 3D film plane in normalised 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 (sub)pixel) to an untransformed world-space point
  - need to undo normalising 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

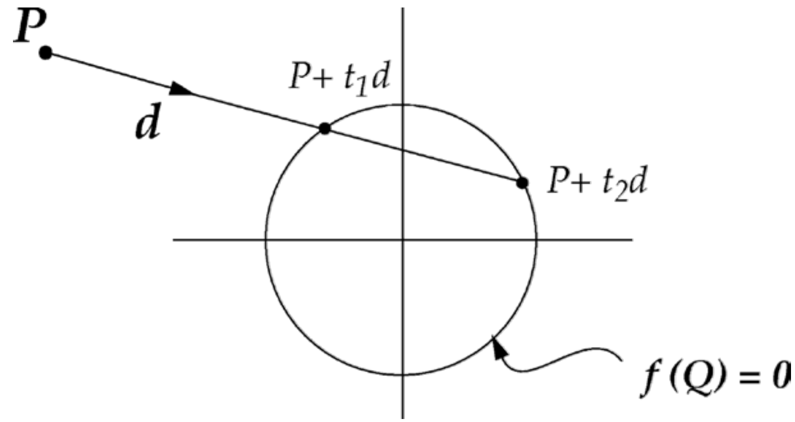
LECTURE 07: RAY TRACING

Implicit Objects

- If an object is defined implicitly by a function f , where f(Q) = 0 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) = 0
- A sphere of radius R in 3-space:
  - $f(x,y,z) = x^2 + y^2 + z^2 – R^2$

Implicit Objects (cont.)

- At what points (if any) does the ray intersect an object?
- Points on a ray have form **P** + t**d** where t is any non-negative real number
- A surface point Q lying on an object has the property that f(Q) = 0
- Combining, we want to know "For which values of t is f(**P** + t**d**) = 0 ?"
- We are solving a system of simultaneous equations in x, y (in 2D) or x, y, z (in 3D)



Explicit vs. Implicit

- Ray equation is explicit $P(t) = R_o + t \times R_d$
  - Parametric
  - Generates points
  - Hard to verify that a point is on the ray
- Plane equation is implicit $H(P) = n \cdot P + D = 0$
  - Solution of an equation
  - Does not generate points
  - Verifies that a point is on the plane
- What about implicit plane and explicit ray?

An Explicit Example – 2D Ray-circle Intersection

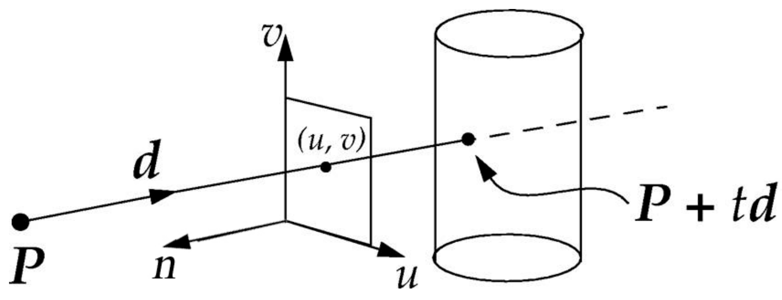
- Consider the eye-point $\boldsymbol{P}$ = (-3, 1), the direction vector $\boldsymbol{d}$ = (0.8, -0.6) and the unit circle: $f(x,y) = x^2 + y^2 - R^2$
- A typical point of the ray is: $Q = \boldsymbol{P} + t\boldsymbol{d} = (-3,1) + t(0.8, -0.6) = (-3+0.8t, 1-0.6t)$
- Plugging this into the equation of the circle: $f(Q) = f(-3+0.8t, 1-0.6t) = (-3+0.8t)^2 + (1-0.6t)^2 - 1$
- Expanding, we get: $9 - 4.8t + 0.64t^2 + 1 - 1.2t + .36t^2 - 1$
- Setting this equal to zero, we get: $t^2 - 6t + 9 = 0$

An Explicit Example – 2D Ray-circle Intersection (cont.)

- Using the quadratic formula: $$roots = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
- We get: $$t = \frac{6 \pm \sqrt{36 - 36}}{2}, \qquad t = 3, 3$$
- 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 – 2D Ray-circle Intersection (cont.)

- Generalising:
  - We can take an arbitrary implicit surface with equation f(Q) = 0, A ray **P** + t**d**, and plug the latter into the former:
    - f (**P** + t**d**) = 0
- The result, after some algebra, is an equation with t as unknown
- We then solve for t, analytically or numerically



Cylindrical Objects – Multiple Conditions

- For cylindrical objects, the implicit equation
  - $f(x, y, z) = x^2 + z^2 - 1 = 0$
  - In 3D-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

Cylinder caps	top:	$x^2 + z^2 - 1 \leq 0,\ y = 1$
	bottom:	$x^2 + z^2 - 1 \leq 0,\ y = -1$

Cylinder Pseudocode

- Solve in a case-by-case approach

Ray_inter_finite_cylinder(P,d):

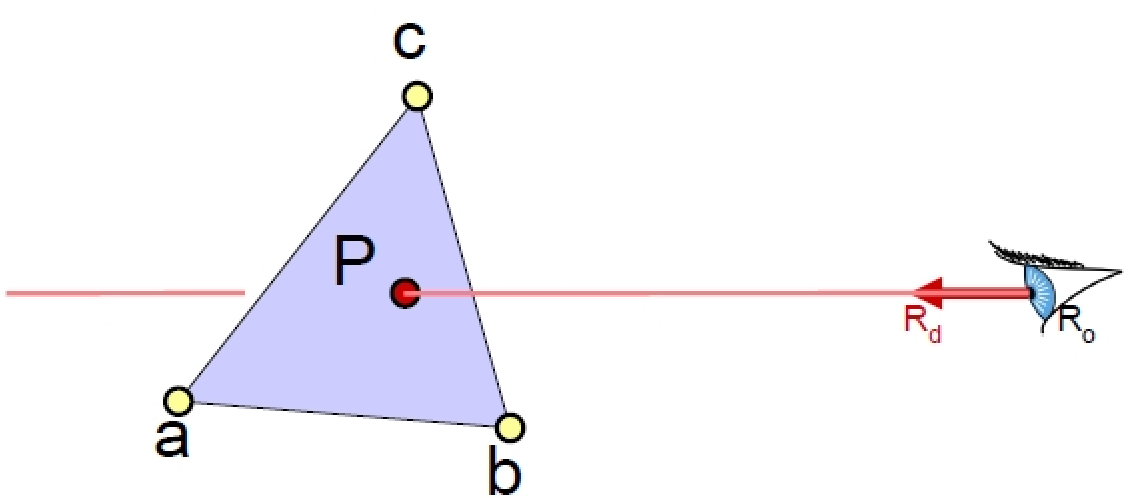
```
t1,t2 = ray_inter_infinite_cylinder(P,d)      // Check for intersection with infinite cylinder
compute P + t1*d, P + t2*d
if y > 1 or y < -1 for t1 or t2: toss it        // If intersection, is it between cylinder caps?

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
```

Implicit Surface Strategy Summary

- Substitute ray (P + td) 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 hierarchy to raytrace!
  - "blobby objects", which are implicit surfaces defined by sums of implicit equations (F(x,y,z)=0)

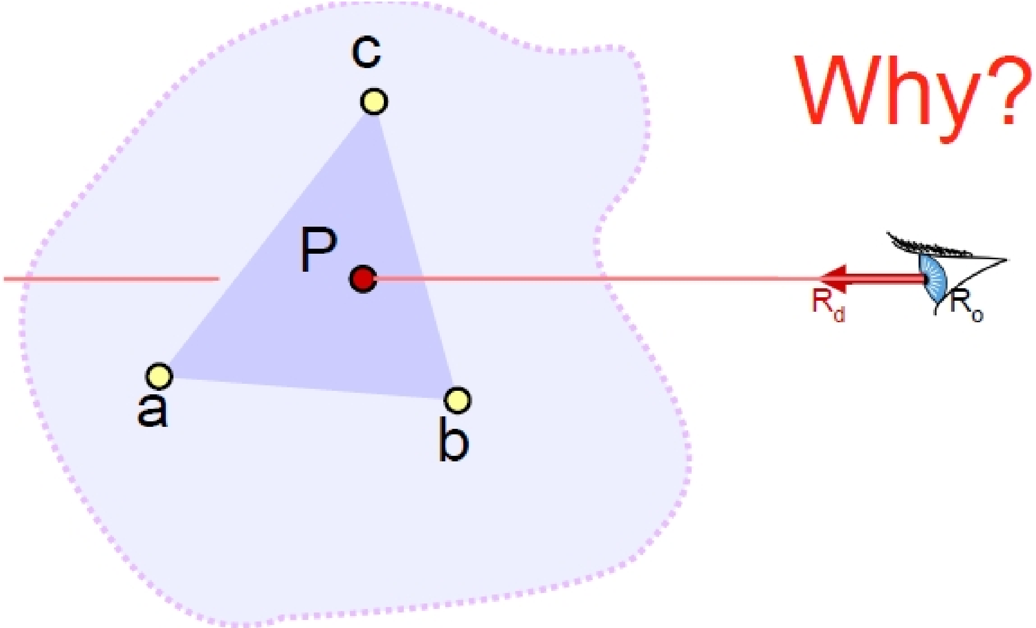
Ray-Triangle Intersection

- Motivation: 3D objects are usually represented by triangle meshes
- Use ray-plane intersection followed by in-triangle test
- Or try to be smarter
  - Use barycentric coordinates



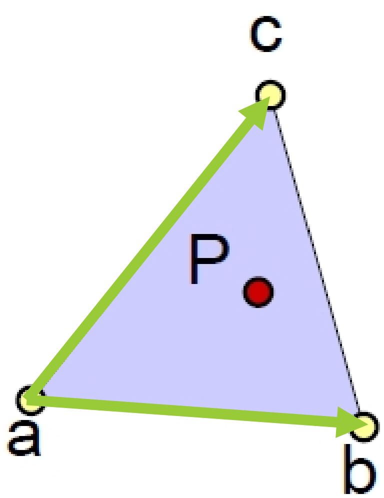
Barycentric Definition of a Plane

- A (non-degenerate) triangle (a,b,c) defines a plane
- Any point P on this plane can be written as
  - P(α,β,γ,) = α **a** + β **b** + γ **c**,
  - with α + β + γ = 1



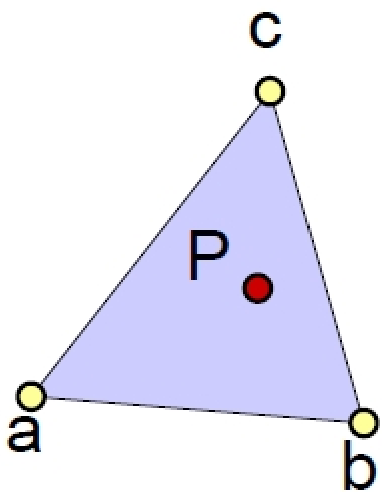
Barycentric Coordinates

- Since $\alpha + \beta + \gamma = 1$, we can write
  - $\alpha = 1 - \beta - \gamma$
- $\mathbf{P}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$
- $\mathbf{P}(\beta, \gamma) = (1-\beta-\gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\,\mathbf{c}$
  - $= \mathbf{a} + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a})$



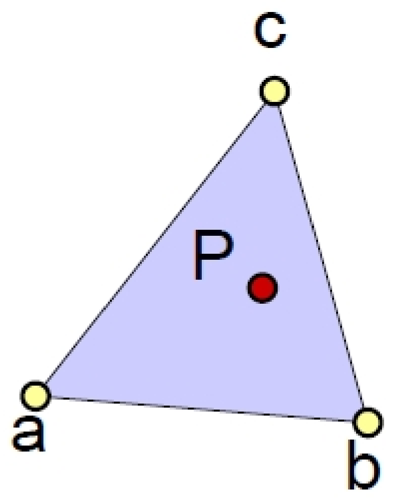
Barycentric Definition of a Plane

- **P**(α, β, γ) = α**a** + β**b** + γ**c**
  - With α + β + γ =1
- Is it explicit or implicit?
- **P** is the **barycenter**, the single point upon which the triangle would balance if weights of size α, β, & γ are placed on points **a**, **b** & **c**



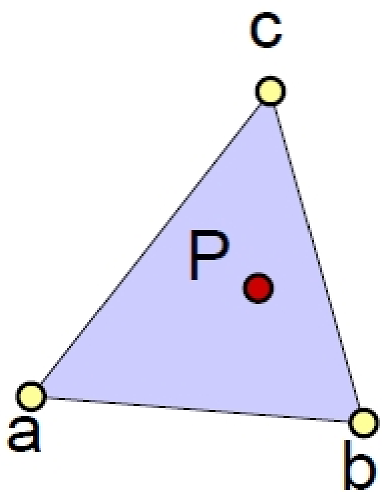
Barycentric Definition of a Plane (cont.)

- $\mathbf{P}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$
  - With $\alpha + \beta + \gamma = 1$ parameterises the entire plane
- If we require in addition that $\alpha, \beta, \gamma >= 0$, we get just the triangle
  - Note that with $\alpha + \beta + \gamma = 1$ this implies $0 <= \alpha <= 1$ & $0 <= \beta <= 1$ & $0 <= \gamma <= 1$
  - Verify:
    - $\alpha = 0 \Rightarrow$ **P** lies on line **b**-**c**
    - $\alpha, \beta = 0 \Rightarrow \mathbf{P} = \mathbf{c}$
    - etc.



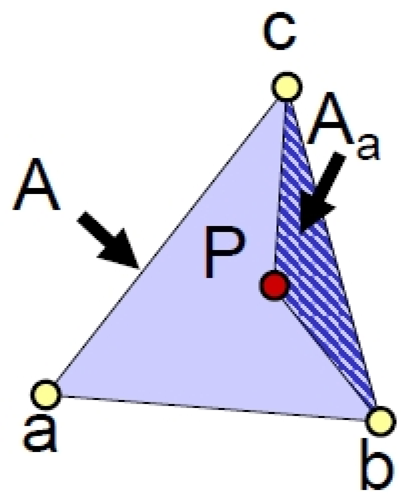
Barycentric Definition of a Plane (cont.)

- **P**(α, β, γ) = α**a** + β**b** + γ**c**
  - With α + β + γ =1 parameterises the entire plane
- Condition to be barycentric coordinates:
  - α + β + γ =1
- Condition to be inside the triangle
  - α, β,γ >= 0



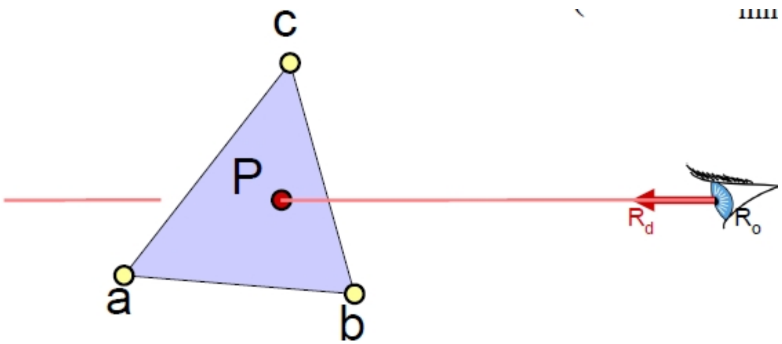
How to Compute α, β, γ?

- Ratio of opposite sub-triangle area to total area
  - $\alpha = A_a/A$
  - $\beta = A_b/A$
  - $\gamma = A_c/A$
- Use signed areas for points outside the triangle



Intersection with Barycentric Triangle

- Again, set ray equation equal to barycentric equation
- $\mathbf{P}(t) = \mathbf{P}(\beta, \gamma)$
- $\mathbf{R_o} + t\,\mathbf{R_d} = a + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a})$
- • Intersection if
  - $\beta + \gamma <= 1$, and
  - $\beta >= 0$, and
  - $\gamma >= 0$



Intersection with Barycentric Triangle (cont.)

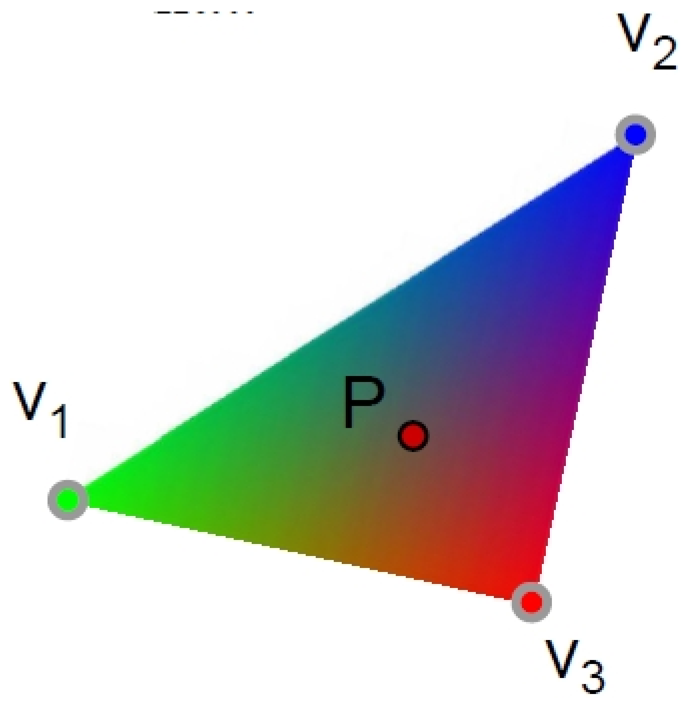
- $\mathbf{R_o} + t * \mathbf{R_d} = a + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a})$
  - $R_{ox} + tR_{dx} = a_x + \beta(b_x-a_x) + \gamma(c_x-a_x)$
  - $R_{oy} + tR_{dy} = a_y + \beta(b_y-a_y) + \gamma(c_y-a_y)$
  - $R_{oz} + tR_{dz} = a_z + \beta(b_z-a_z) + \gamma(c_z-a_z)$
- Regroup and write in matrix form Ax=b
- $\begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$

Barycentric Intersection Advantages

- Efficient
- Stores no plane equation
- Get the barycentric coordinates for free
- Useful for interpolation, texture mapping

Barycentric Interpolation

- Values v_1, v_2, v_3 defined at a, b, c
  - Colors, normal, texture coordinates, etc.
- **P**(α, β, γ) = α **a** + β **b** + γ **c** is the point...
- $v(\alpha, \beta, \gamma) = \alpha\, v_1 + \beta\, v_2 + \gamma\, v_3$ is the barycentric interpolation of v_1,v_2,v_3 at point **P**
  - Sanity check: $v(1,0,0) = v_1$, etc.
- i.e, once you know α, β, γ you can interpolate values using the same weights.
  - Convenient!



Summary of Ray Casting (put it all together)

```
P = eyePt

for each sample of image:
   Compute d
   for each object:
        Take world space intersection point P+td
        Convert to object space,   +t
        Use implicit formula to calculate t for object
        Store object space intersection for later normal calculation

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

   Compute object space normal at object space intersection
   Transform object space normal to world space

   Use normal and intersection point (both in world space) for
   lighting computation
```

Shadows

LECTURE 07: RAY TRACING

Equation of Shadows

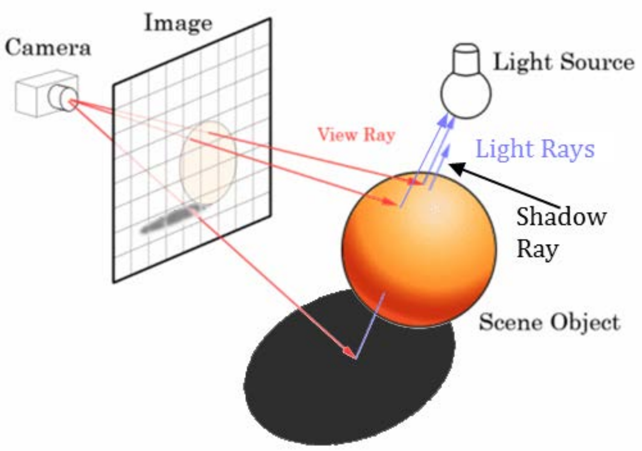
- Each light in the scene contributes to the color and intensity of a surface element…

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

- If and only if light source reaches the object
  - could be occluded/obstructed by other objects in scene
  - could be self-occluding

Algorithm of Shadows

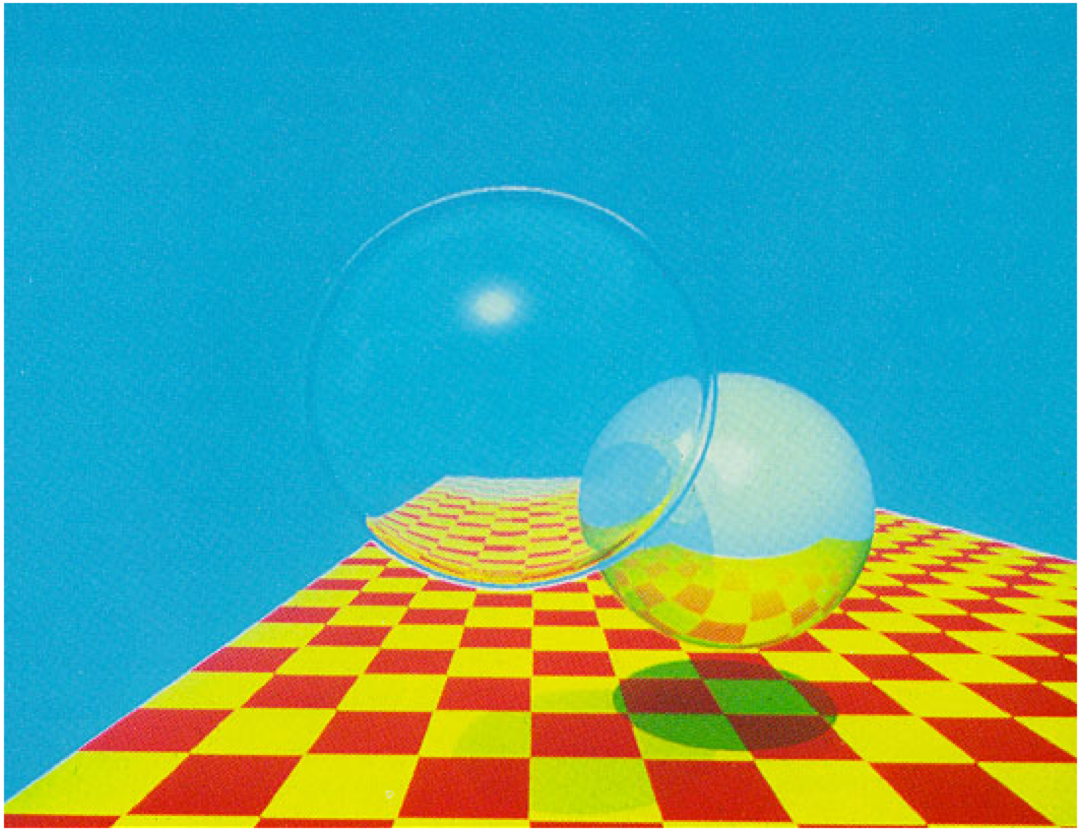
- Construct a ray from the surface intersection to each light
- Check if light is first object intersected
  - if first object intersected is the light, count light's full contribution
  - if first object intersected is not the light, do not count (ignore) light's contribution
  - this method generates 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 $\Longrightarrow$ need recursive ray tracing



Recursive Ray Tracing

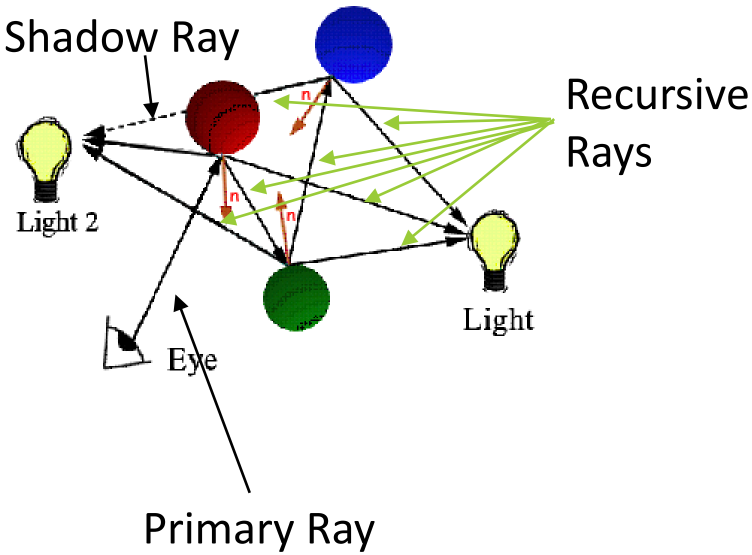
LECTURE 07: RAY TRACING

Recursive Ray Tracing



Simulating global lighting effects

- 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)



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

Lighting Equation for Recursive Ray Tracing

- 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; LP 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
  - n is the normal vector at the object surface

Lighting Equation for Recursive Ray Tracing (cont.)

- New lighting equation (Phong lighting + specular reflection + transmission):

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

  - l is the vector to the light
  - r is the reflected light vector
  - 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)

Some Old Videos

- https://www.youtube.com/watch?v=sg8YwRNA5j8

Some Old Videos (cont.)

- https://www.youtube.com/watch?v=b_UqzLBFz4Y

Transparent Surfaces

LECTURE 07: RAY TRACING

Non-Refractive Transparency

- For a partially transparent polygon

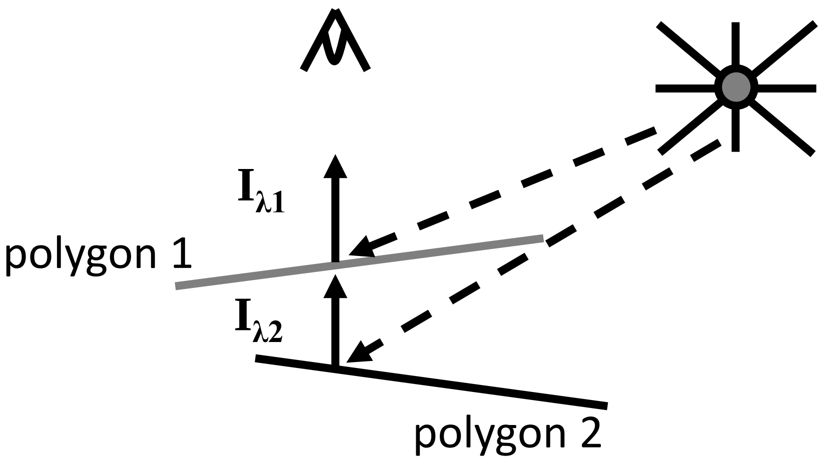
$$I_\lambda = (1 - k_{t1}) I_{\lambda 1} + k_{t1} I_{\lambda 2}$$

$$k_{t1} = \text{transmittance of polygon } 1$$

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

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

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



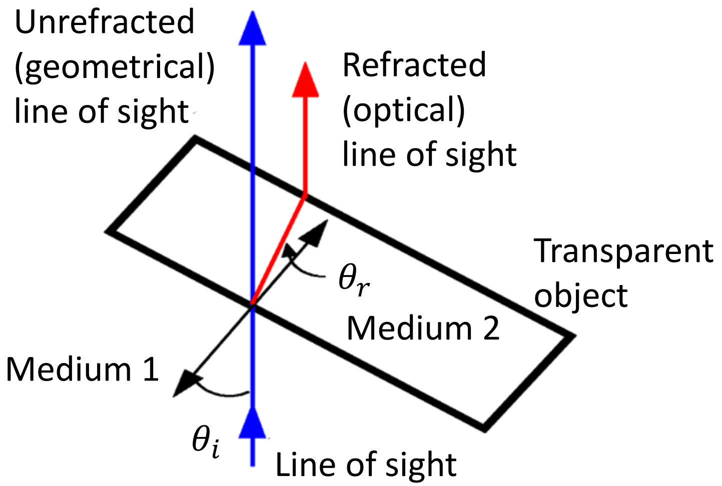
Refractive Transparency

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

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

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

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

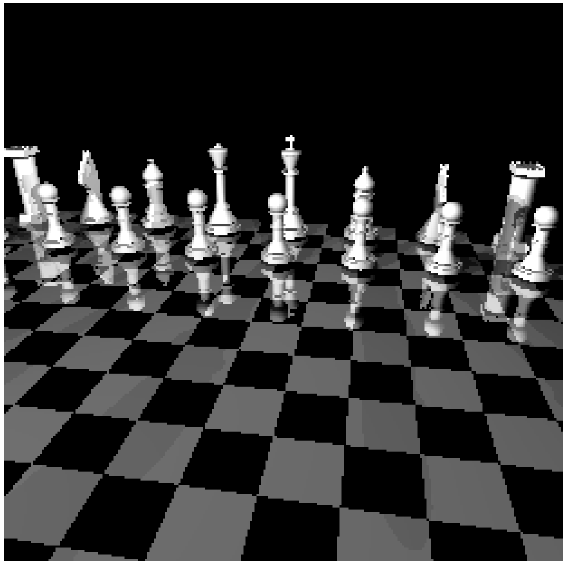


Sampling

LECTURE 07: RAY TRACING

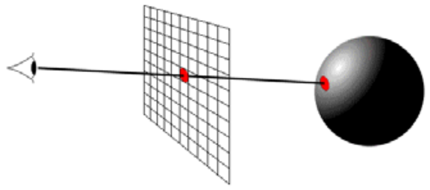
Choosing Samples

- This example samples once per pixel. This generates images similar to this:
- We have a clear case of the jaggies
- Can we do better?

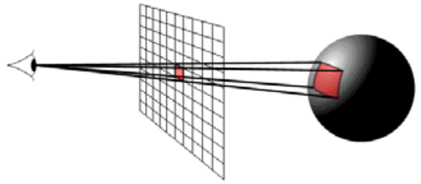


Choosing Samples (cont.)

- 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



vs.



Supersampling Example

WITH SUPERSAMPLING

WITHOUT SUPERSAMPING