Raytracing

Adam Mally
CIS 561 Spring 2017
University of Pennsylvania

What is raytracing?

- Computer graphics technique used to simulate the way light bounces off surfaces
- Render fairly realistic-looking images without having to fully simulate the infinitely-many photons light sources give off
- Trace light paths backwards, from camera to light source(s)

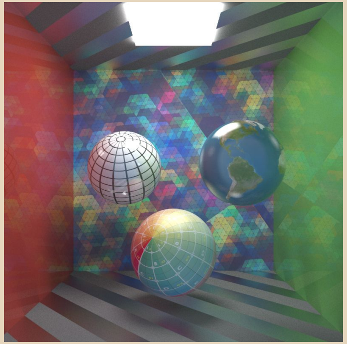


Image source:
http://www.yiningkarlli.com/

Basic Structure of a Raytracer

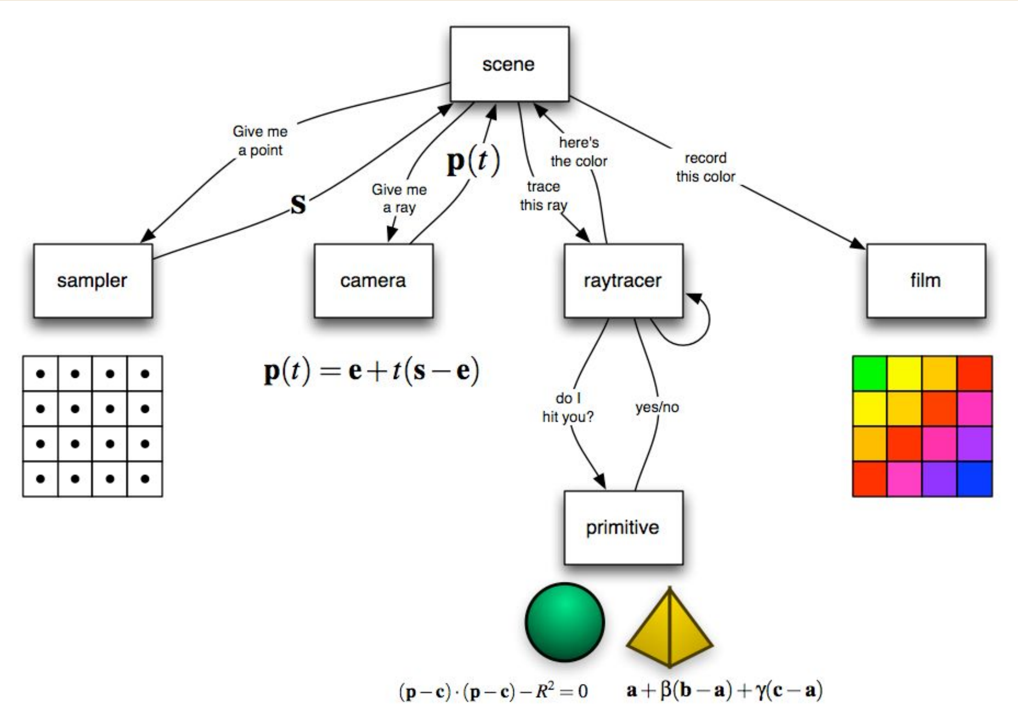


Image source: UC Berkeley CS184

Today’s Topics

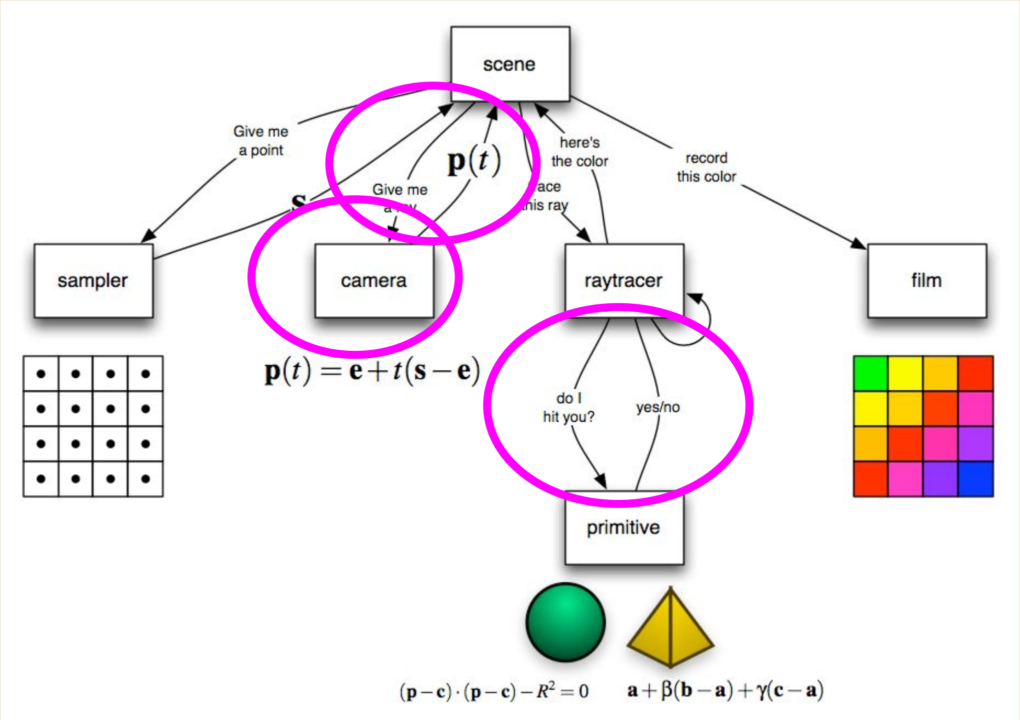


Image source: UC Berkeley CS184

Discussion: Camera Transformations

- What does it mean to transform a point from world space to camera space?
- What is the range of values for (Sx, Sy, Sz, Sw) and (Px, Py, Pz, Pw)?
  - When Cz == near_clip, what is Pz?
  - When Cz == far_clip, what is Pz?
- How can we use this sequence of camera transformations to cast a ray that corresponds to a particular pixel in the screen?

Pixel Space, P = (Px, Py, Pz, Pw)

Px = ((Sx + 1)/2) * Width ↑ ↓ Sx = (Px/Width) * 2 - 1
Py = ((1 - Sy)/2) * Height ↑ ↓ Sy = 1 - (Py/Height) * 2

Screen Space, P = (Sx, Sy, Sz, Sw)

P /= Uw ↑ ↓ P *= Uw

Unhomogenized Screen Space, P = (Ux, Uy, Uz, Uw = Cz)

Proj_Mat * P ↑ ↓ Proj_Mat^{-1} * P

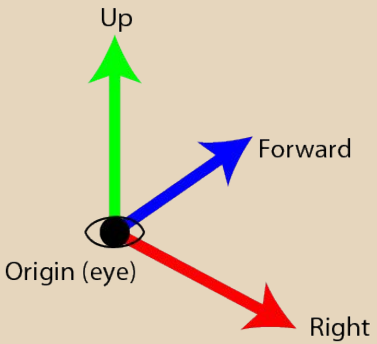
Camera Space, P = (Cx, Cy, Cz, Cw)

View_Mat * P ↑ ↓ View_Mat^{-1} * P

World Space, P = (Wx, Wy, Wz, Ww)

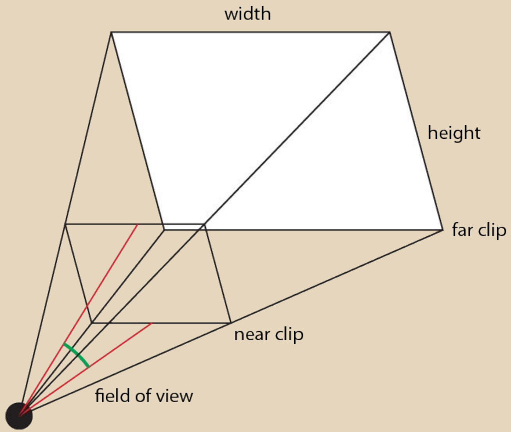
Review: View matrix parameters

- A space coordinate system is defined by two features: an origin and a set of axes
- Camera origin: simply the position the camera inhabits in world space
- Forward direction (F): Also known as the "look vector". A direction in world coordinates that represents the direction in which the camera is looking. Represents the local Z-axis
- Local right (R): A direction in world coordinates that represents the direction that is "rightward" in the camera's local coordinates. Represents local X-axis
  - Perpendicular to the look direction
- Local up (U): A direction in world coordinates that represents the direction that is "upward" in the camera's local coordinates. Represents local Y-axis
  - Perpendicular to both local right and look direction



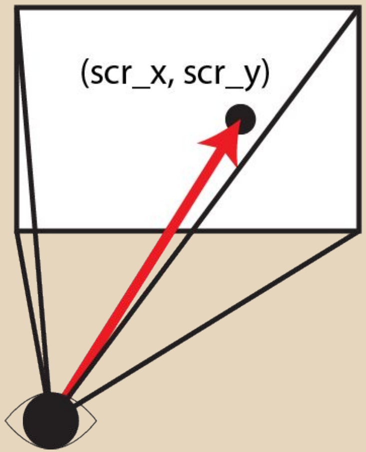
Review: Perspective Frustum

- We can visualize the volume in which objects are visible to our camera
- Geometry outside the pyramid is not visible
- Shaped by several components:
    - Field of view
    - Aspect ratio: screen width / screen height
    - Near clip and far clip planes



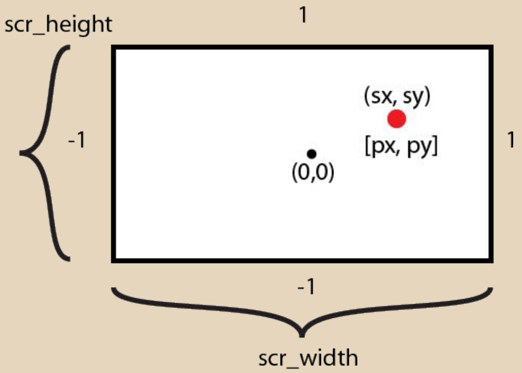
What is raycasting?

- Creating a line that passes through the viewing frustum and travels from the eye to some endpoint on a slice of the frustum (e.g. the far clip plane)
- The line’s endpoint is determined by the pixel on our screen from which we want to raycast



Normalized device coordinates

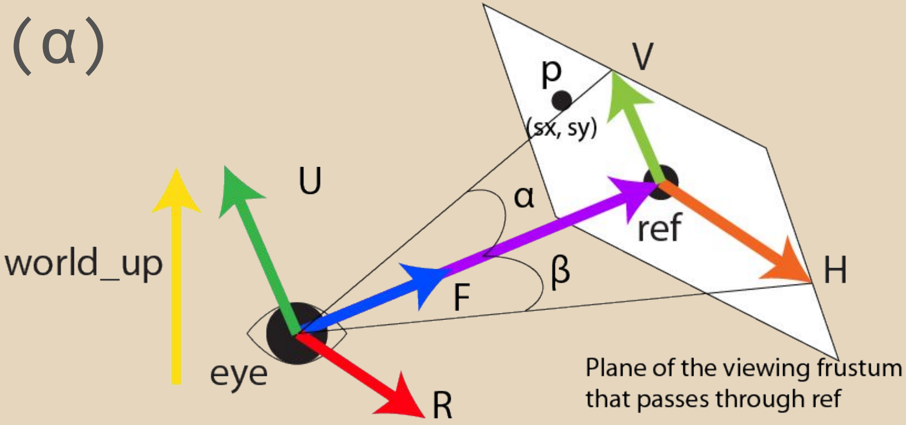
- Recall that your screen ranges from -1 to 1 on both the X and Y axes
- We can convert to NDC from any given pixel



- sx = (2 * px/scr_width) – 1
- sy = 1 – (2 * py/scr_height)
- px and py are the given pixel’s x and y coordinates

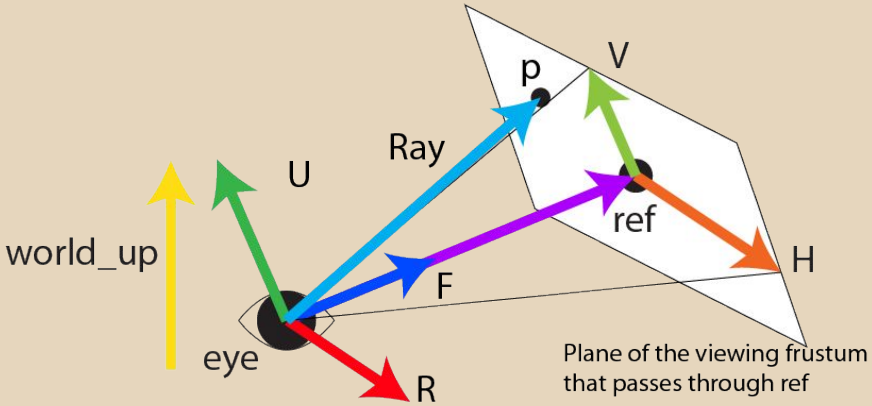
Screen point to world point

- ref = eye + t * F
    - t = any float > 0
- len = |ref - eye|
- V = U*len*tan(α)
- H = R*len*aspect*tan(α)
- α = FOVY/2
- p = ref + sx*H + sy*V
    - sx, sy are in NDC



Getting a ray from the world point

- ray_origin = eye
- ray_direction = normalize(p - eye)
- Arbitrary point on ray = eye + t*ray_direction



Faster method for computing a ray

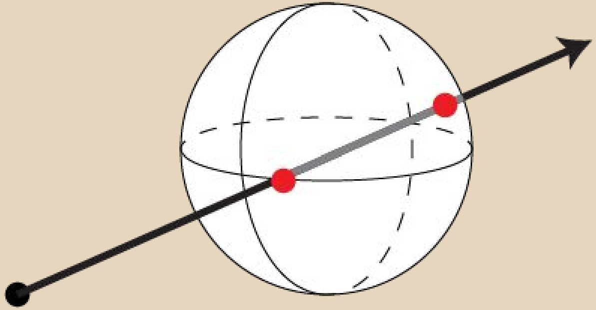
- `P = ViewMat`$^{-1}$ `* ProjMat`$^{-1}$ `* ((px, py, 1, 1) * farClip)`
  - `px, py` are coordinates in NDC
  - `ray_origin = eye`
  - `ray_direction = normalize(P - eye)`
- Pixel Z coord is technically arbitrary as long as it's in range (0, 1]
  - We use a value of 1 because we know where the far clip plane is, so un-homogenizing the vector is easy.
- What spatial transformation does this accomplish?
- Only need to compute the inverse of the view-projection matrix once per camera change

What do we do with rays?

- Find their intersections with geometry in the scene
- Compute “fragment data” for these intersections
- Use this data to shade the pixels that correspond to each ray
- How do we find these intersections?

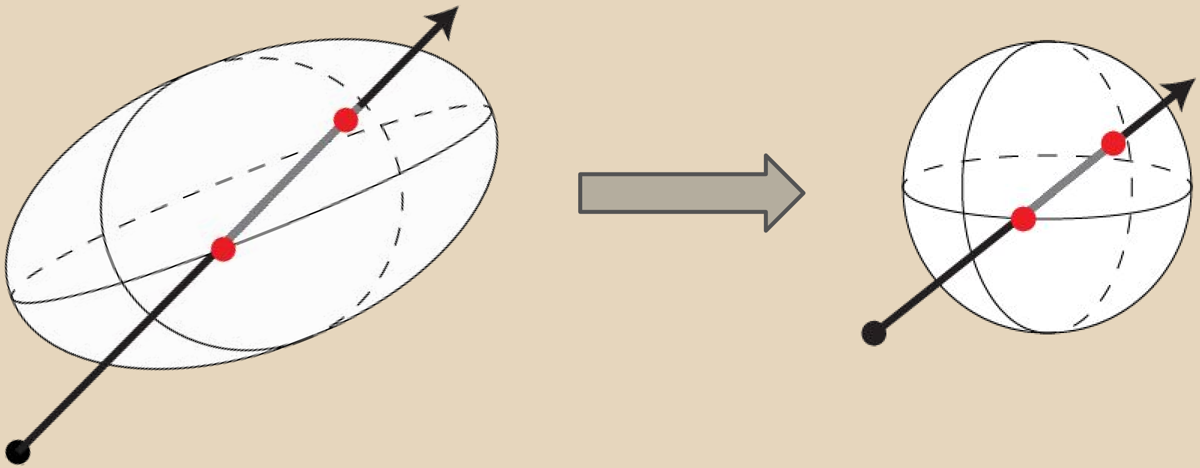
Ray-polygon intersection

- Most common intersection test is ray-triangle
- We'll also cover ray-sphere and ray-cube
  - All three are commonly used in basic raytracer testing
- Generally, want to test for intersection against the **untransformed** geometry



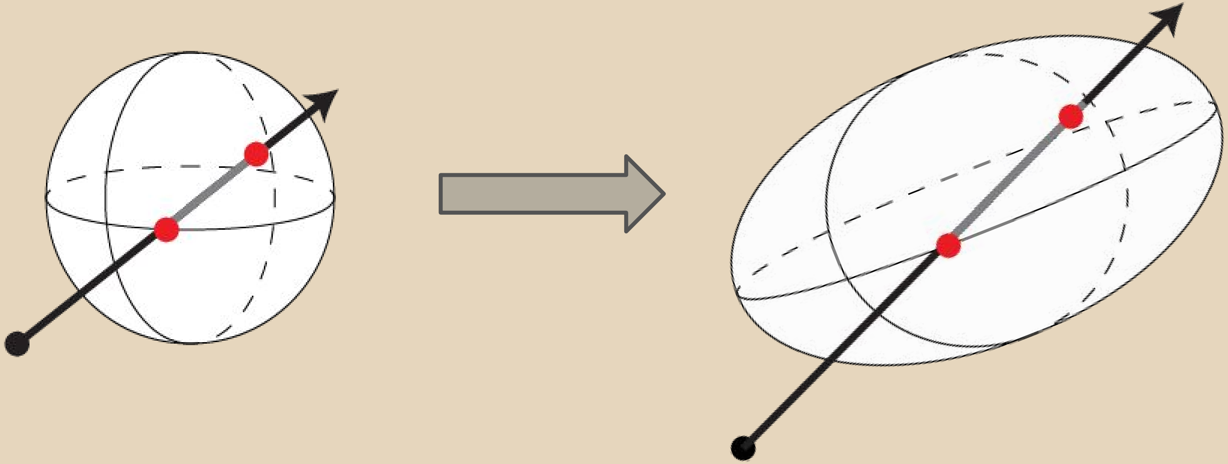
Frames of reference

- Before you try to test a ray against primitive geometry, you must first transform the ray so from its perspective, the geometry in question **is** primitive
- Simply transform the ray's direction and origin by the *inverse* of the geometry's model matrix



Frames of reference

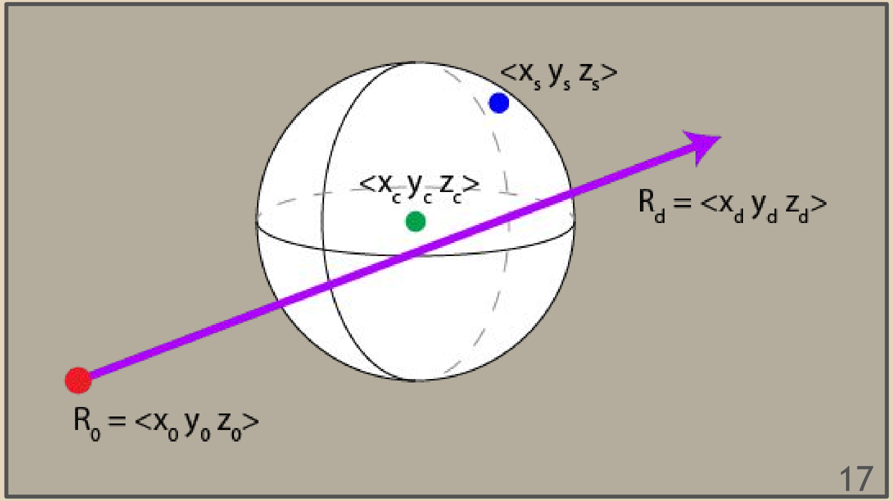
- Similarly, make sure you transform the results of your intersection test back into world space (e.g. the point of intersection, the surface normal at the intersection, etc.)



Ray-sphere intersection

- Sphere defined as $(x_s - x_c)^2 + (y_s - y_c)^2 + (z_s - z_c)^2 = r_s^{\ 2}$
  - Sphere center = $\langle x_c\ y_c\ z_c \rangle$
  - All points on the sphere surface = $\langle x_s\ y_s\ z_s \rangle$
  - r_s is the sphere’s radius

- Ray defined as: $R_0 + t * R_d$
  - $R_0 = \langle x_0\ y_0\ z_0 \rangle$
  - $R_d = \langle x_d\ y_d\ z_d \rangle$
  - t is a parameterization of R_d (i.e. a float)



Ray-sphere intersection

- Substitute $<x_s y_s z_s>$ for the ray equation:

$(x_0 + t*x_d - x_c)^2 + (y_0 + t*y_d - y_c)^2 + (z_0 + t*z_d - z_c)^2 = r_s^2$

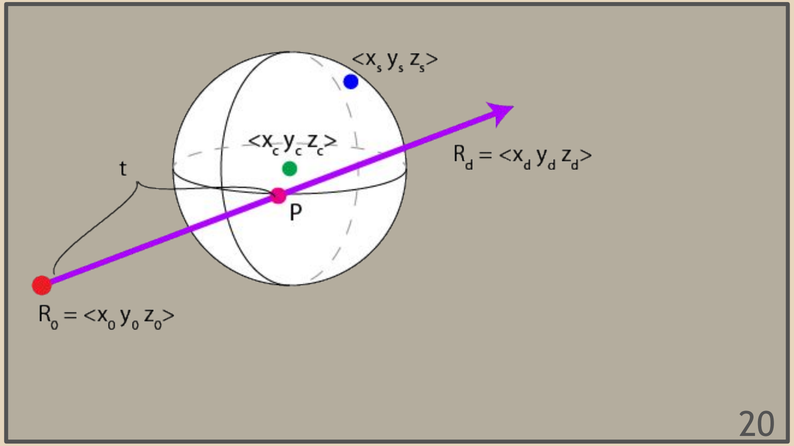
- Can also be written as:
- $At^2 + Bt + C = 0$
  - $A = x_d^2 + y_d^2 + z_d^2$
  - $B = 2(x_d(x_0 - x_c) + y_d(y_0 - y_c) + z_d(z_0 - z_c))$
  - $C = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r_s^2$
- Note that we now have a quadratic equation
  - We can solve for t using the quadratic formula!

Ray-sphere intersection

- t_0 , t_1 = **$(-B \pm \sqrt{(B^2-4AC)})/(2A)$**
  - t_0 is for the - case and t_1 is for the + case
- Remember: if the discriminant is negative, then there is no real root and therefore no intersection
  - Discriminant = B^2-4AC
- If t_0 is positive, then we're done. If not, then compute t_1.

Ray-sphere intersection

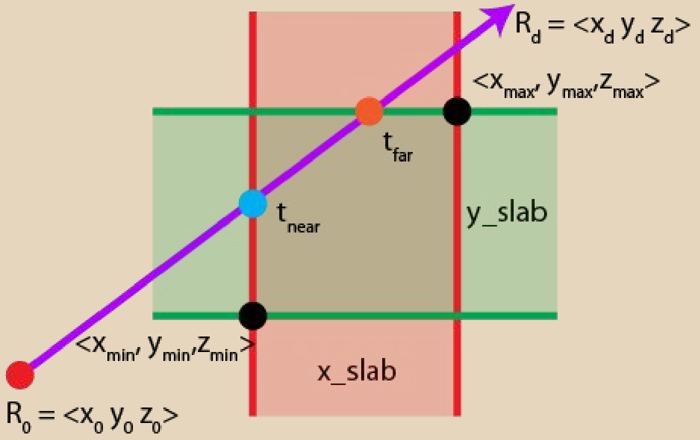
- Once we have t, we can plug it into our ray equation to find the closest point of intersection on our sphere.
  - If all we care about is whether or not we hit the sphere, we can just check:
  - near_clip < t < far_clip
- $P = R_0 + t * R_d$



Ray-cube intersection

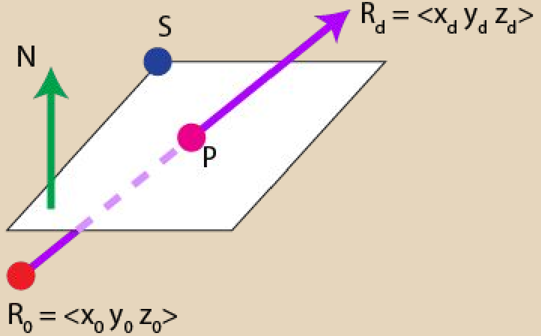
- Begin by storing t_{near} = -infinity and t_{far} = infinity
- For each pair of planes associated with the X, Y, and Z axes (the example uses the X "slab"):
  - If x_d is 0, then the ray is parallel to the X slab, so
    - If $x_0 < x_{min}$ or $x_0 > x_{max}$ then we miss
  - $t_0 = (x_{min} - x_0)/x_d$
  - $t_1 = (x_{max} - x_0)/x_d$
  - If $t_0 > t_1$ then swap them
  - If $t_0 > t_{near}$ then $t_{near} = t_0$
  - If $t_1 < t_{far}$ then $t_{far} = t_1$
- Repeat for Y and Z
- If $t_{near} > t_{far}$ then we miss the box

We can't see the z_slab because it's coming directly out of the screen



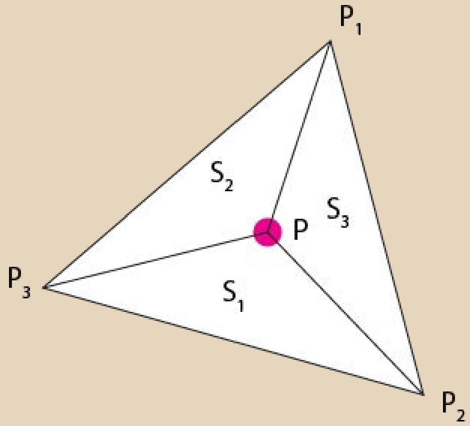
Ray-plane intersection

- Plane defined as: dot(N,(P-S)) = 0
  - N is the plane's normal
  - S is some point on the plane
  - P is the point of intersection
- Ray defined as: R_0 + t * R_d
- Substitute P for ray:
  - dot(N,(R_0 + t * R_d - S)) = 0
- Solve for t:
  - t = dot(N,(S - R_0)) / dot(N,R_d)



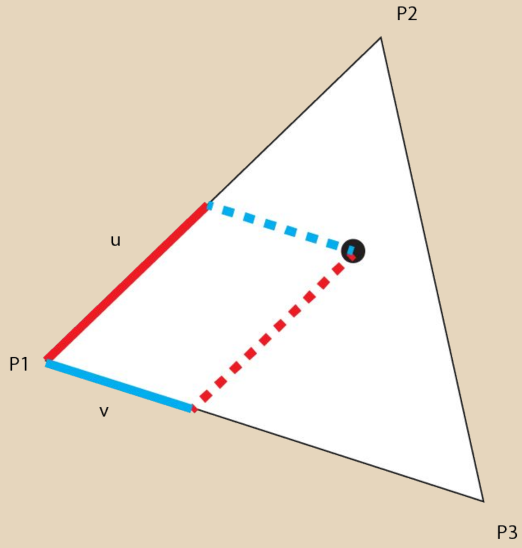
Point-in-triangle

- Use **barycentric coordinates** to test if P is within the bounds of a triangle
  - The **barycenter** of a triangle is its center of mass, often given unequal weighting to its vertices
- $S = \text{area}(P_1, P_2, P_3)$
- $S_1 = \text{area}(P, P_2, P_3)/S$
- $S_2 = \text{area}(P, P_3, P_1)/S$
- $S_3 = \text{area}(P, P_1, P_2)/S$
- Therefore, $P = S_1P_1 + S_2P_2 + S_3P_3$
- So, P is within the triangle if:
  - $0 \le S_1 \le 1$
  - $0 \le S_2 \le 1$
  - $0 \le S_3 \le 1$
  - $S_1 + S_2 + S_3 = 1$



Better triangle test

- Use only two weights to parameterize the triangle
- Point in triangle = $(1 - u - v)P_1 + uP_2 + vP_3$
  - $u >= 0, v >= 0, (u + v) <= 1$
- Point on ray = $R_0 + t * R_d$
- Substitute ray equation in:
- $R_0 + t * R_d = (1 - u - v)P_1 + uP_2 + vP_3$
- Reformulate:
  - $[-R_d, P_2 - P_1, P_3 - P_1] \begin{vmatrix} t \\ u \\ v \end{vmatrix} = R_o - P_1$



“Fast, Minimum Storage Ray/Triangle Intersection” Moller and Trumbore
https://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

Better triangle test

- $[-R_d, P_2 - P_1, P_3 - P_1]\begin{vmatrix} t \\ u \\ v \end{vmatrix} = R_o - P_1$

- Change to a format that can be solved with Cramer's Rule:
  - $P_2 - P_1 = E_1,\ P_3 - P_1 = E_2,\ R_o - P_1 = T$
  - $\begin{vmatrix} t \\ u \\ v \end{vmatrix} = \dfrac{1}{|-R_d \quad E_1 \quad E_2|} \begin{vmatrix} |T & E_1 & E_2| \\ |-D & T & E_2| \\ |-D & E_1 & T| \end{vmatrix}$

"Fast, Minimum Storage Ray/Triangle Intersection" Moller and Trumbore

https://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

Better triangle test

- $$\begin{vmatrix} t \\ u \\ v \end{vmatrix} = \frac{1}{\begin{vmatrix} -R_d & E_1 & E_2 \end{vmatrix}} \begin{vmatrix} \begin{vmatrix} T & E_1 & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & T & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & E_1 & T \end{vmatrix} \end{vmatrix}$$

- The length of a vector <A,B,C> can also be expressed as $-(A \times C)\cdot B = -(C \times B)\cdot A$

$$\begin{vmatrix} t \\ u \\ v \end{vmatrix} = \frac{1}{(R_d \times E_2)\cdot E_1} \begin{vmatrix} (T\times E_1)\cdot E_2 \\ (D\times E_2)\cdot T \\ (D\times E_1)\cdot D \end{vmatrix}$$

“Fast, Minimum Storage Ray/Triangle Intersection” Moller and Trumbore
https://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

Square intersection

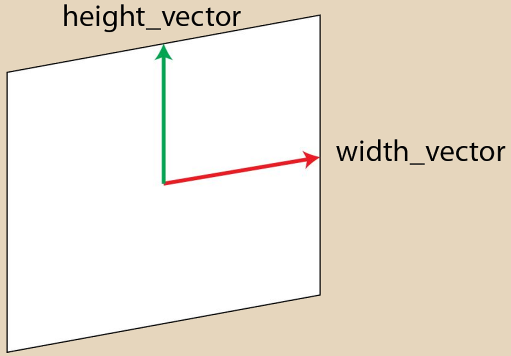
- If we have a planar square in a scene, what methods could one use to find its intersection with a ray?

Square intersection

- Find the point of intersection with the infinite plane the square lies in
- Test to see if this point lies within the bounds of the square
- Transform ray into square’s local space and just test that $-0.5 < P.x < 0.5$ and $-0.5 < P.y < 0.5$
  - This assumes the unit square is aligned with the XY plane, is centered at the origin, and has a side length of 1.

Square intersection

- If you know the vectors that make up the “width” and “height” of the square, you don’t even need to transform the ray into the square’s local space
  - If the following two statements are true, then the point-in-plane lies within the square
  - abs(dot(P - square_center, width_vector)) < length(width_vector)^2
  - abs(dot(P - square_center, height_vector)) < length(height_vector)^2
  - Why?



Finding local normals

- When we’ve found an intersection in local object space, we usually want to find the local normal at that point
  - Normals are essential for most shading computations
- How can we find the object-space normal of a sphere?

Finding local normals

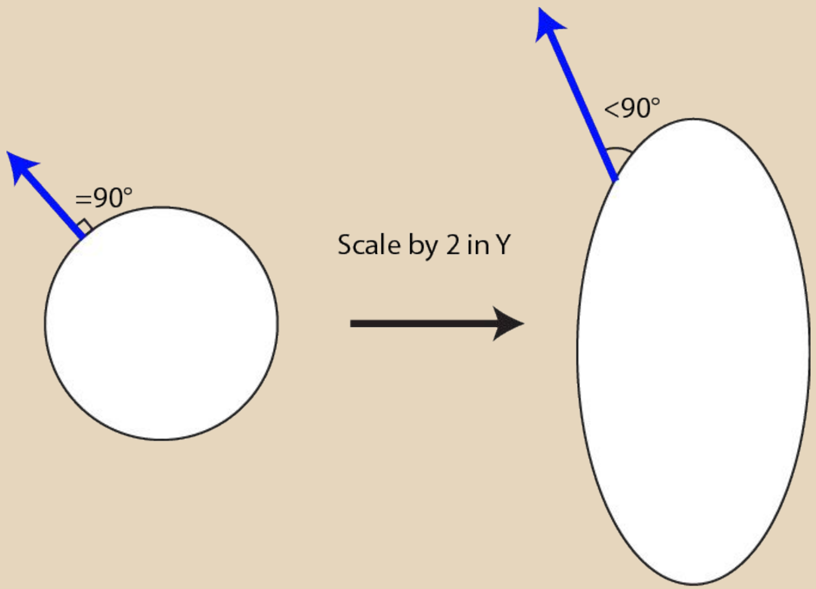
- When we’ve found an intersection in local object space, we usually want to find the local normal at that point
  - Normals are essential for most shading computations
- How can we find the object-space normal of a sphere?
  - Normalize the point of intersection (assuming W is 0 beforehand)
- What about cubes?

Finding local normals

- When we've found an intersection in local object space, we usually want to find the local normal at that point
  - Normals are essential for most shading computations
- How can we find the object-space normal of a sphere?
  - Normalize the point of intersection (assuming W is 0 beforehand)
- What about cubes?
  - Need to find which coordinate has the greatest magnitude
  - This determines which of the three major axes the normal lies along
  - Set the positivity of the normal by multiplying it with the **sign** of the largest-magnitude coordinate
  - Example: intersection is <-0.5, 0.45, -0.3>, so normal is <-1, 0, 0>

Surface normals

- Properly transforming surface normals from object space into world space is not as simple as multiplying them by the model matrix
    - Doing so skews them slightly, making them no longer normal (orthogonal) to the surface



Surface normals

- Of translate, rotate, and scale operations, only scale incorrectly transforms normals
- We want to **invert** the scale that is applied to the normal while keeping rotation the same
  - Translation has no effect on surface normal, so we ignore it entirely
- If we just invert the model matrix, both **rotation** and **scale** are inverted
- If we **invert and transpose** the model matrix, only **scale** is inverted
  - Inverting a rotation matrix $R(\theta)$ is equivalent to making a matrix $R(-\theta)$
  - Transposing a rotation matrix $R(\theta)$ is also equivalent to making a matrix $R(-\theta)$
  - Combining the inverse and transpose double-inverses the rotation, ultimately leaving it as just $R(\theta)$

Surface normals

- Given a point on a surface, there exists a surface normal n and some tangent vector t
  - $\mathrm{Dot}(n, t) = n^T t = 0$
- When the object is transformed by a model matrix M, $Mt = t'$
  - The transformed normal n' must remain orthogonal to t', so we multiply n by some matrix S to get n'
  - $0 = (n')^T t'$
  - $0 = (Sn)^T Mt$
  - $0 = (n)^T S^T Mt = n^T t$
  - The above identity implies that $S^T M = I$, which in turn implies that $S^T = M^{-1}$, therefore $S = (M^{-1})^T$
- In summary, multiply the local-space surface normal by the *inverse transpose* of the model matrix to correctly bring it into world space
- Matrix **must not include** translation components before being inverted or it will transform the normals improperly
- This same proof can be found on page 94 of PBRT vol. 3

Basic Structure of a Raytracer

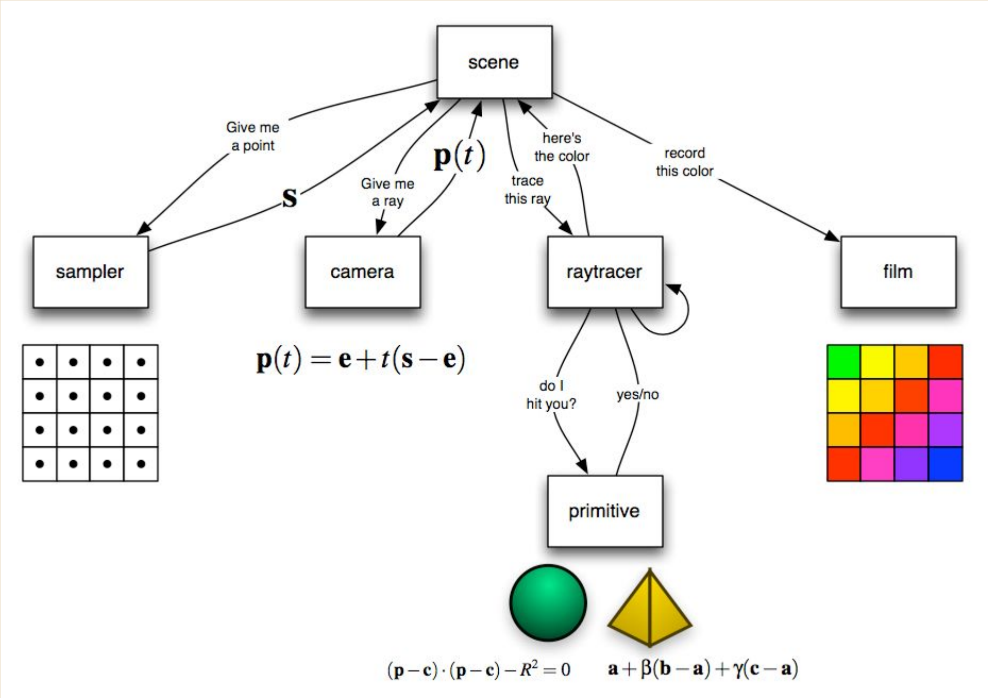
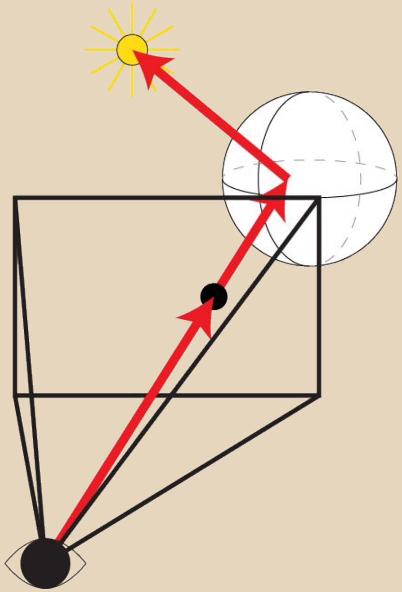


Image source: UC Berkeley CS184

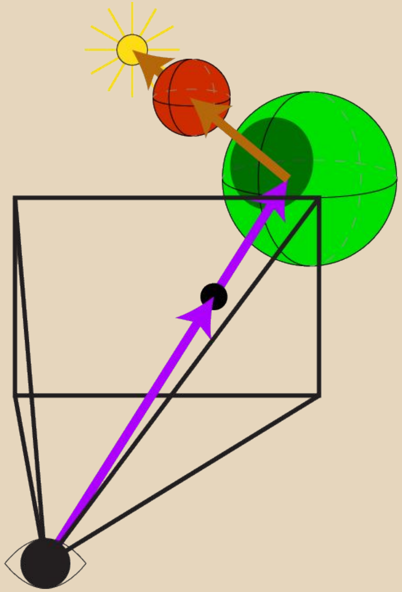
Backward Raytracing

- In the real world, photons are emitted by light sources and bounce off surfaces
- A small fraction of reflected photons reach our eyes, meaning the majority of emitted photons have no bearing on what we see
- To avoid computing the paths of all these extra photons, we trace the paths of photons in reverse
  - From each pixel of our camera screen into the scene, and back to the light source



Basic Raytracing

- The most basic raytracing algorithm involves two main ray types:
  camera rays and **light-feeler rays**
- Camera rays are emitted through each pixel of the view screen and are tested against all geometry in the scene
- When a camera ray hits a scene object, a light-feeler ray is cast from the point of intersection to each light source in the scene
  - If a light-feeler ray reaches its light source then it contributes a portion of light to the overall color of the camera ray that created it
  - If a light-feeler ray is obstructed by an object in the scene, then it contributes pure black to the overall camera ray color



Light-feeler issue: floating point error

- When computing the point of intersection with a surface, floating point error often results in the POI being slightly inside the object intersected
- When casting the light-feeler ray, it will intersect the object for which we are trying to compute the lighting
- This causes an effect commonly known as “shadow acne”

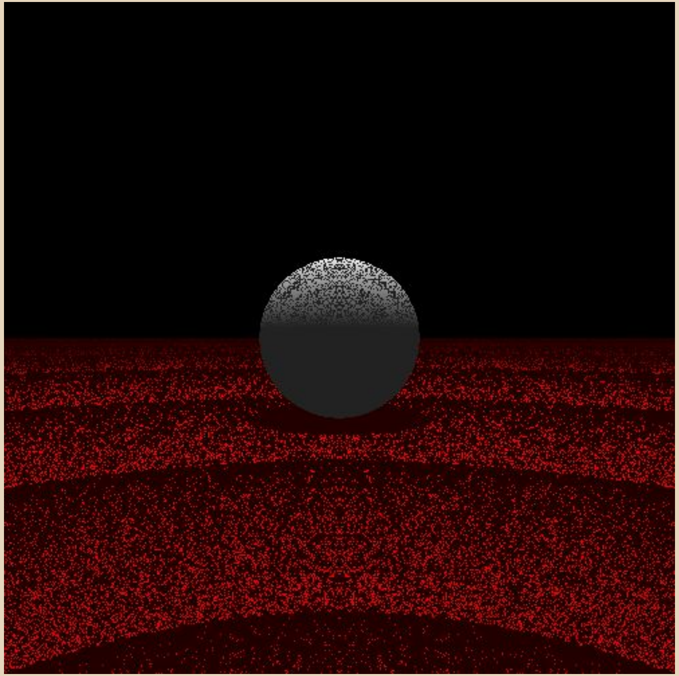


Image source: http://joedoliner.com/wp-content/uploads/2011/01/surface_acne_sphere_plane.png

Shadow acne: what to do?

- Ignore intersection with last object hit
  - Problem: prevents self-shadowing
- Use a minimum *t* value for the shadow test
  - Problem: A "good" *t* value depends on the scale of the scene and the distance the intersection is from the camera
- Use *doubles* instead of *floats* for extra precision
  - Problem: Increases memory needed to store the scene
- Offset the computed intersection by moving it along the surface normal
  - Should only move a *very* small amount (e.g. a factor above the smallest possible increment of a floating point, $\sim 10^{-6}$)
- PBRT chapter 3.9 discusses this at length

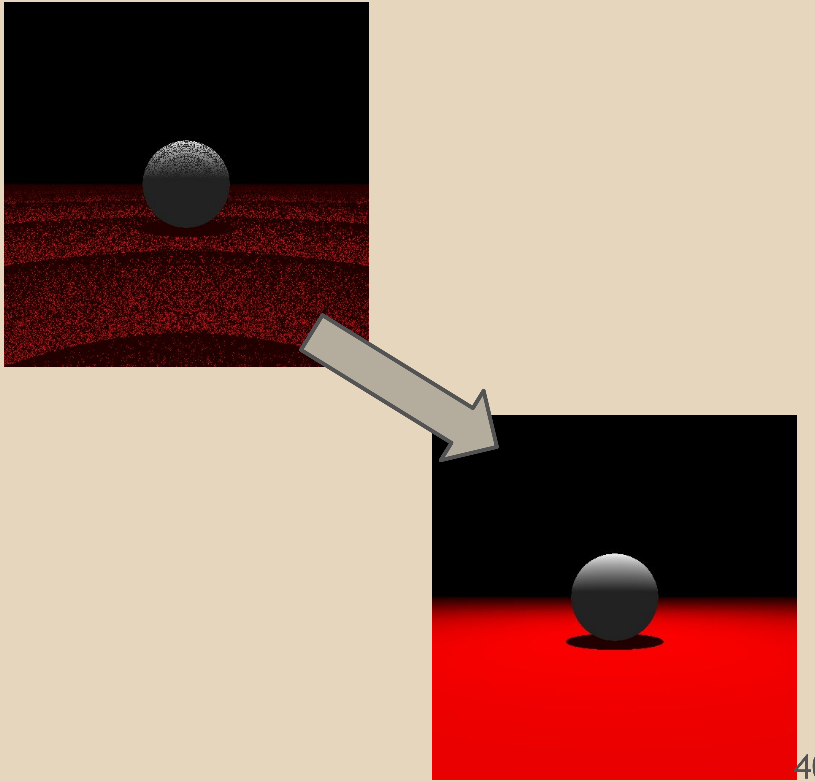
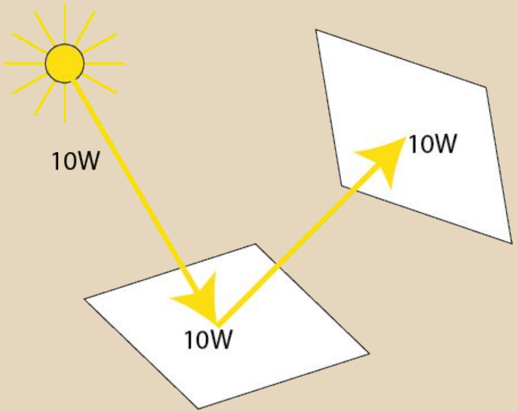


Image source: http://joedoliner.com/wp-content/uploads/2011/01/sphere-plane.png

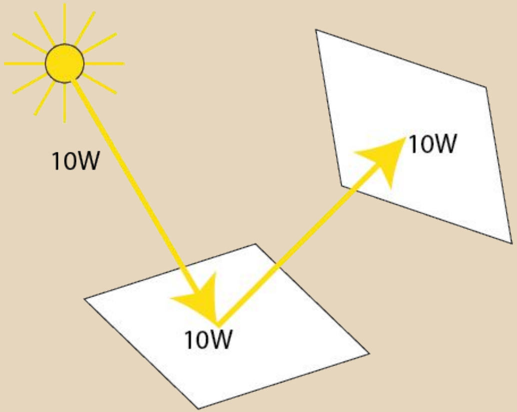
Radiometry

- There are several important properties of light that make up the foundation of raytracing (excluding certain phenomena such as black holes)
  - A light ray travels in a perfectly straight line from one point to another
  - Light rays do not interfere with one another if they intersect
  - Given two points in space that can directly see one another, the amount of light emitted from point A towards point B is the same amount of light seen at point B (i.e. the light is invariant)



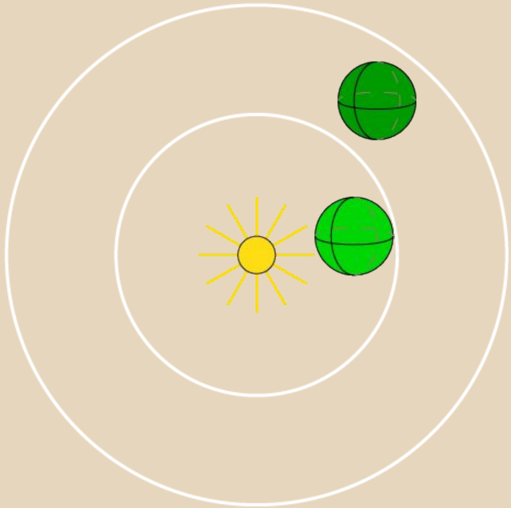
Radiometry

- Within radiometry, there is a system of units and measures for illumination
- Take an optics (geometric) approach to light
- Again, treat light as if it travels in perfectly straight lines
  - Excellent approximation when the light’s wavelength is much smaller than the objects with which the light interacts
  - Cannot model diffraction, interference, etc.

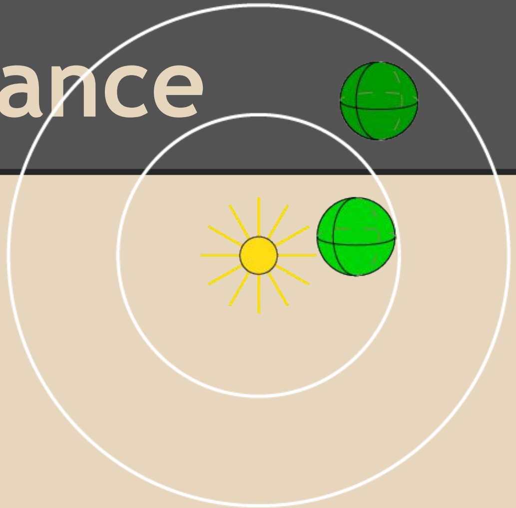


Physics of lighting: Flux

- We want to simulate the gradual shading that objects exhibit in real life
- By using various shading models (Lambert, Phong, Gouraud, etc.) we can approximate the energy reflected at a surface point
- A light's intensity reduces as it travels away from its source
    - **Radiant flux** is the amount of energy passing through a region of space per unit of time, measured in Joules/sec (aka Watts) and commonly denoted as Φ
    - Both spheres surrounding the point light have the same amount of total **flux**, but any one local area of the larger sphere has less flux than a local area on the smaller sphere
        - Hence objects further from the light being more weakly illuminated



Physics of lighting: Irradiance



- We can represent the amount of **flux** arriving at a surface as **irradiance** (*E*), and the amount of **flux** leaving a surface as **radiant exitance** (*M*)
- Their unit of measurement is Watts/meter^2
- The **irradiance equation** for our point light emission sphere is:
  $\boldsymbol{E = \Phi/(4\pi r^2)}$
  - This is just flux/surface area
- The amount of energy (illumination) from this light falls off proportionally to the squared distance from the light since it becomes diffused over an increasingly larger area

Physics of lighting: Lambert's law

- We can use the irradiance equation to help understand the origin of **Lambert's law**
  - Lambert's law: the amount of light arriving at a surface is proportional to the cosine of the angle between the light direction and surface normal
  - $\boldsymbol{E = \Phi \cos(\theta)/A_L}$
- As θ increases, the area of the lit surface increases, meaning the flux is distributed across a larger surface area, causing the irradiance to decrease for any one point on the lit surface
- Review: `dot(A, B) = |A||B|cos(θ)`
  - θ = angle between A and B
- Can re-write Lambert's equation as $\boldsymbol{E = \Phi\,\mathrm{dot}(N, L)/A_L}$
  - Assuming N and L are normalized, of course
- The amount of light reaching a single point on a surface from a point light ($A_L = 0$) can be represented simply as ***`clamp(dot(N, L), 0, 1)`***