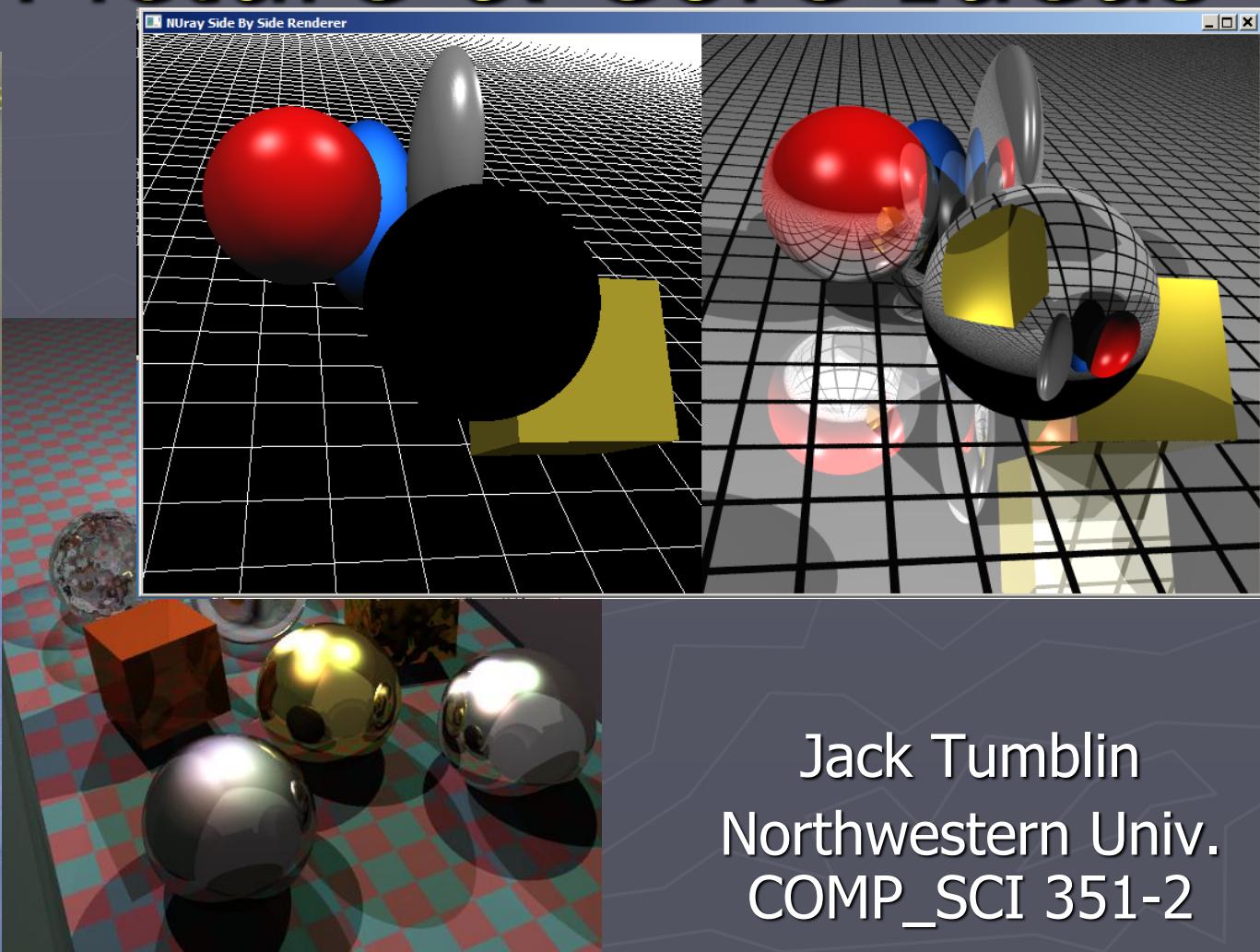


Ray Tracing A: The Big Picture & Core Ideas



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

Ray-Tracing Is Fundamental

► Rich, fun, largely informal history:

- Greeks, 10th century Arabic reasoning...
- Turner Whitted's 1980 [renegade project at Bell Labs](#);
- 1987 SIGGRAPH '[Jello](#)' Paper ; ([or here](#))
- Ray-Tracing News,
[http://www.realtimerendering.com/resources/RTNews/html/](#)
- Back side of Eric Haine's business card:
[https://www.gabrielgambetta.com/tiny-raytracer.html](#)

► Yet a rigorous model of light transport

- For optical design: [https://support.zemax.com/hc/en-us/articles/1500005488361-Understanding-paraxial-ray-tracing](#)
- (caution ... neglects diffraction/wave optics effects!!)
- Movies & Games -- see 'the PBRT book': [http://www.pbrt.org/](#)
- [Monte-Carlo: 'gold standard'](#) for simulation & realism...

Real Photo? or Synthetic?

Ray tracing + Glob. Illum. Yields Accurate Simulations



Loft in New York - Gab 2009 - Blender/Yafaray

How can we compute this? Trace 3D 'rays' to estimate where light travels in scene...

Key Idea 1: Time-Reversed Photons

**WATCH THESE
(try 2X speed)**

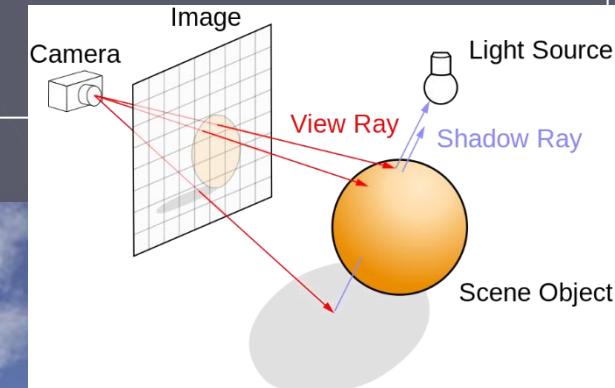
Pixar/Khan: <https://www.khanacademy.org/computing/pixar/rendering/rendering1/a/start-here-rendering>

Haines/nVIDIA: <https://news.developer.nvidia.com/ray-tracing-essentials-part-1-basics-of-ray-tracing/>

BOOK: <http://www.realtimerendering.com/raytracinggems/>

DEMO: <https://youtu.be/KYekhnLHGms>

TWIST: <https://youtu.be/KYekhnLHGms?t=255>



"Forward" Photon Streams

eye
pixel
lens
air/blob
blob/air
air/cloud
cloud/air
sun

Time-Reversed 'Rays' for Ray-Tracing

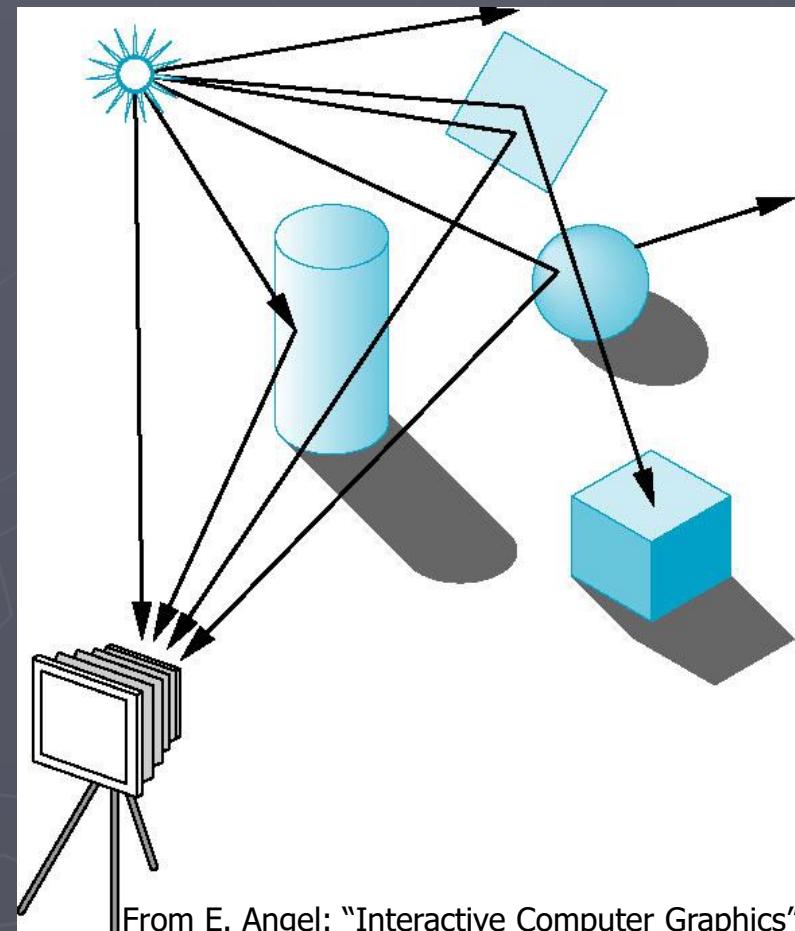
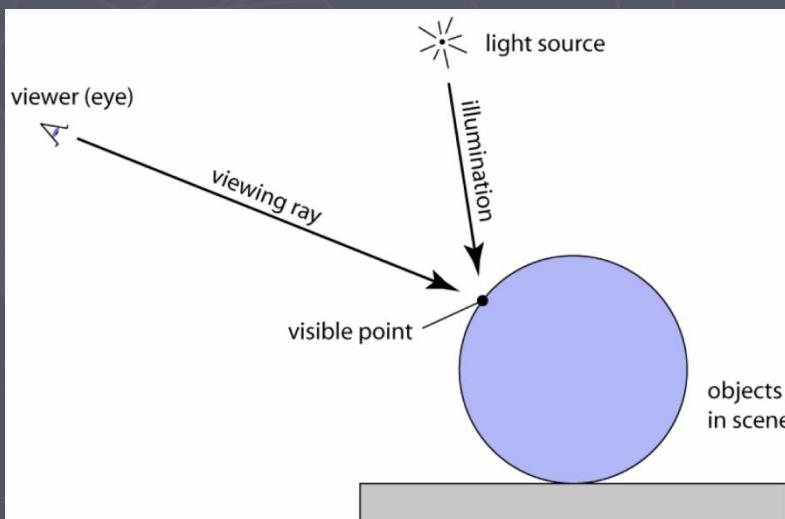


Key Idea 1: Time-Reversed Photons

- ▶ 'ray' defined as 3D line parameterized by 't':
$$\text{ray}(t) = (\text{origin_point}) + (\text{direction_vector}) \cdot t$$



- ▶ Let 'rays' simulate paths of photons through scene, but trace *BACKWARDS*:

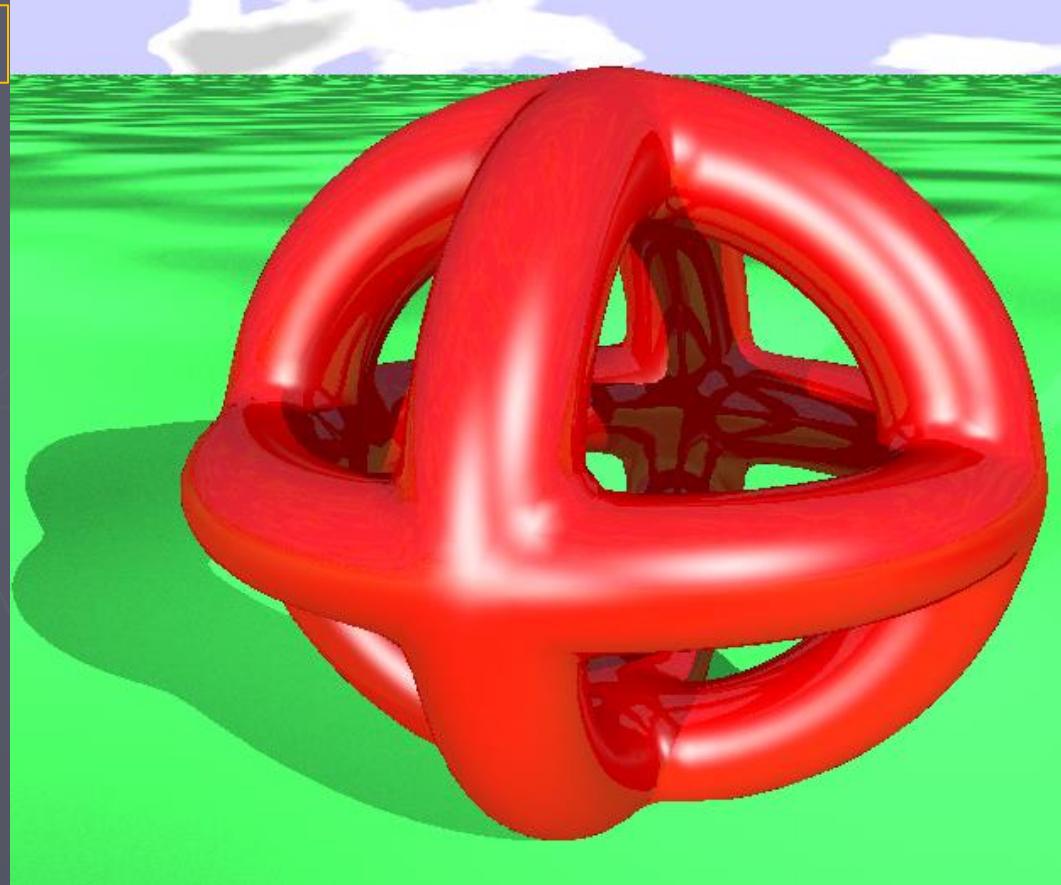


From E. Angel: "Interactive Computer Graphics"

Key Idea 2: Implicit Surfaces

Wikipedia: POV ray-tracer

3D Surfaces
Defined by
Scalar Function:



Surface == constant distance to nearest point on 3 circles

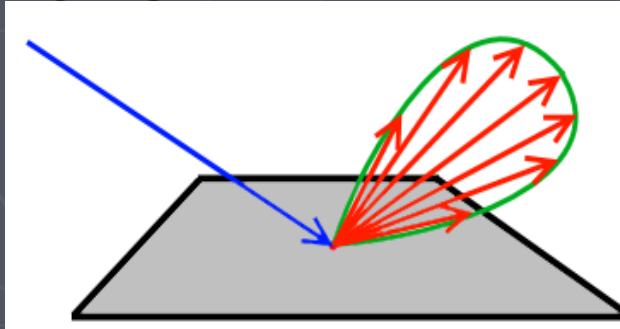
Another EXAMPLE: 'distance to sphere' function

$$D(x,y,z) = x^2 + y^2 + z^2 - 5^2$$

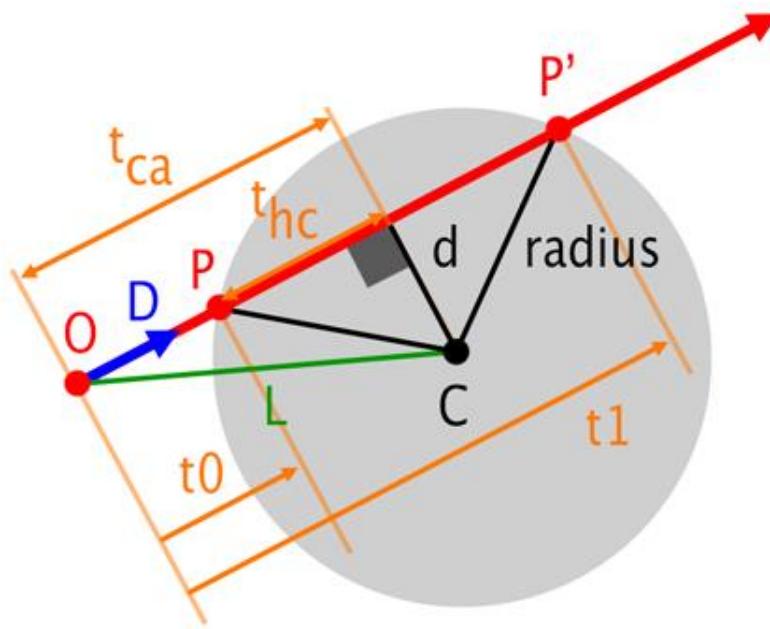
Key Idea 3: Ray-Surface Intersections

Ray Tracing ==

- ▶ Make ray(s) for each image pixel;
- ▶ Insert **ray equation** into **surface equation**: solve for ray-length t_{hit} at the 'hit-point'
- ▶ Compute surface color at hit-point.
 - HOW? more rays, recursively...)

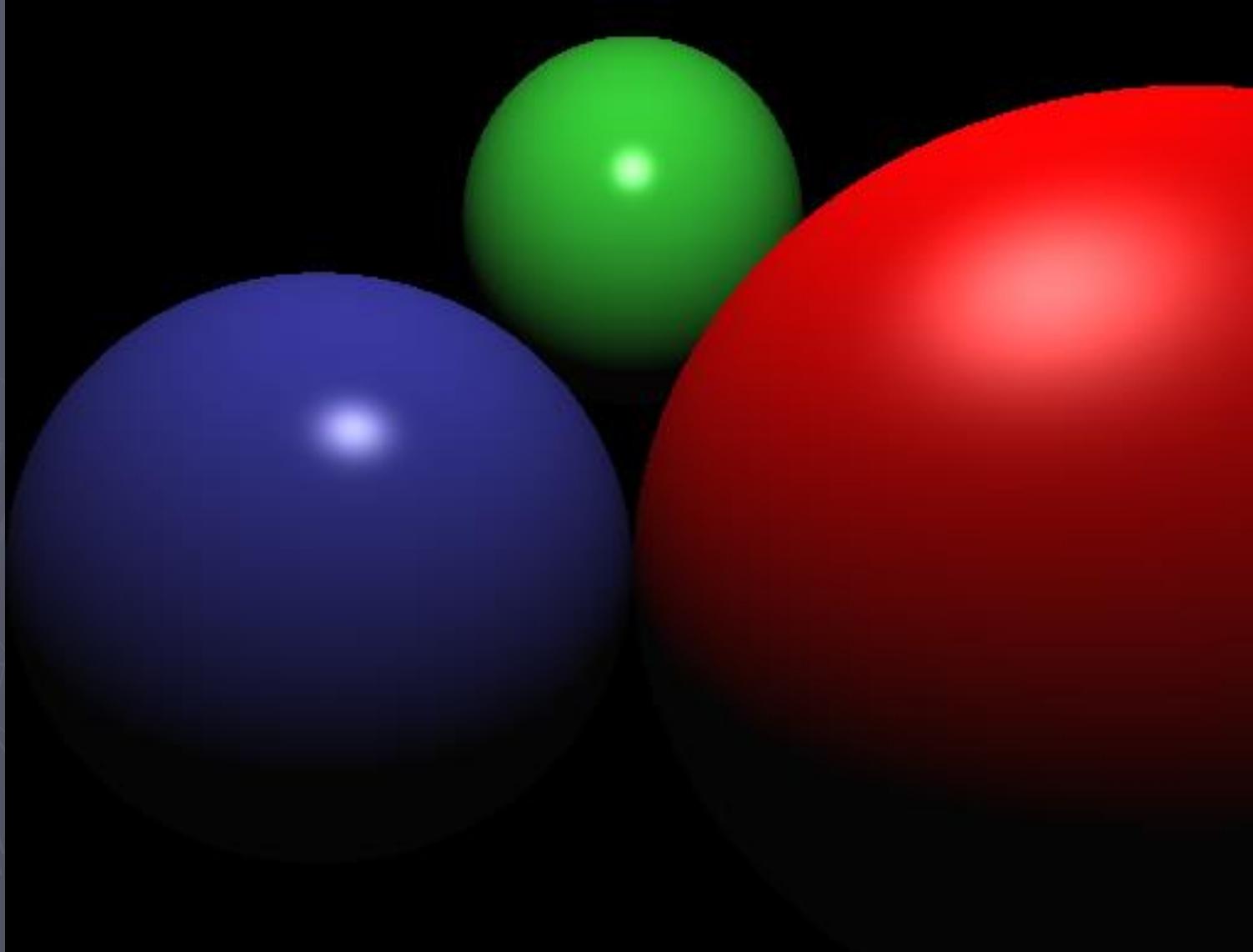


© www.scratchapixel.com



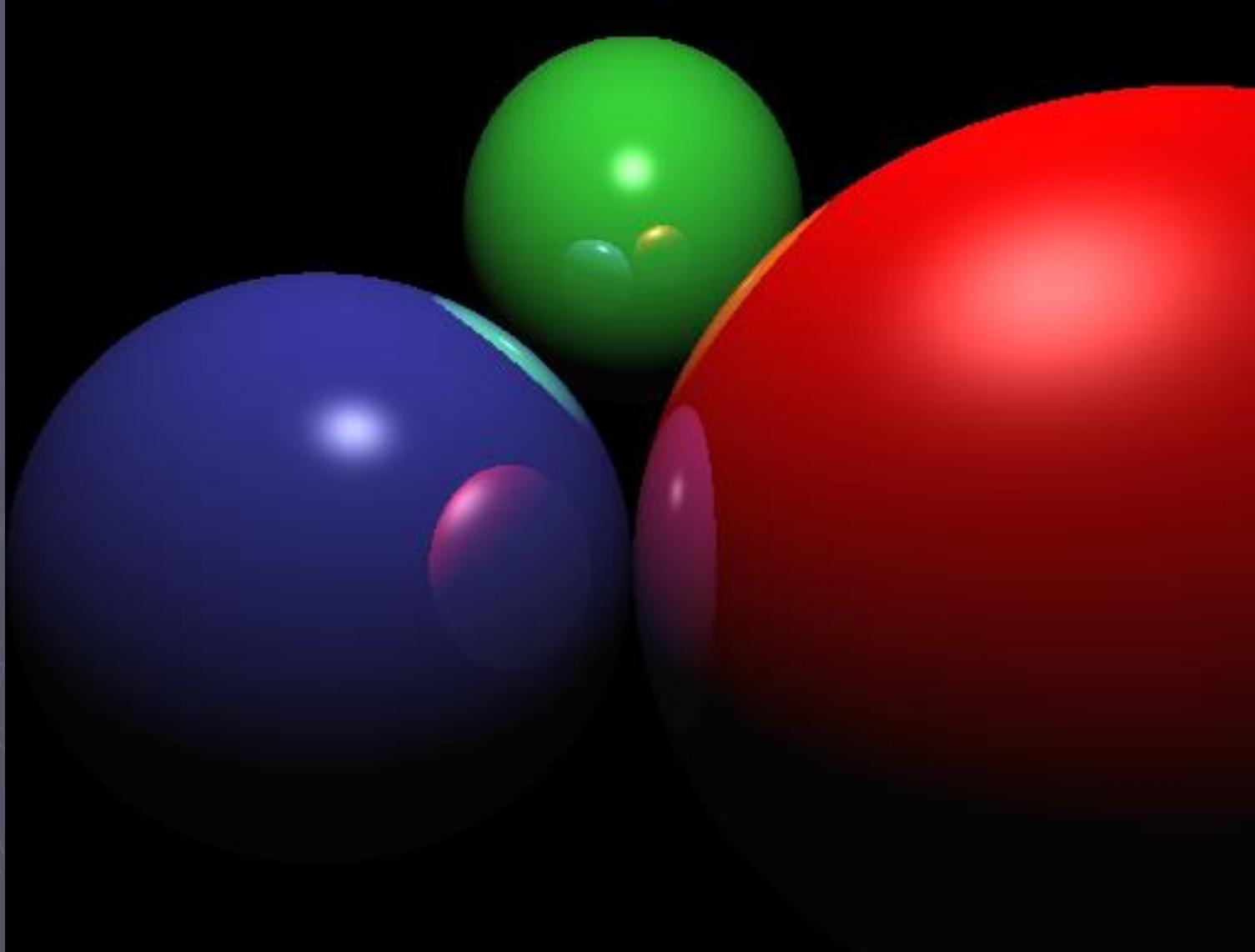
Key Idea 4: Recursive Reflection (0)

(Images From Univ. Delaware)



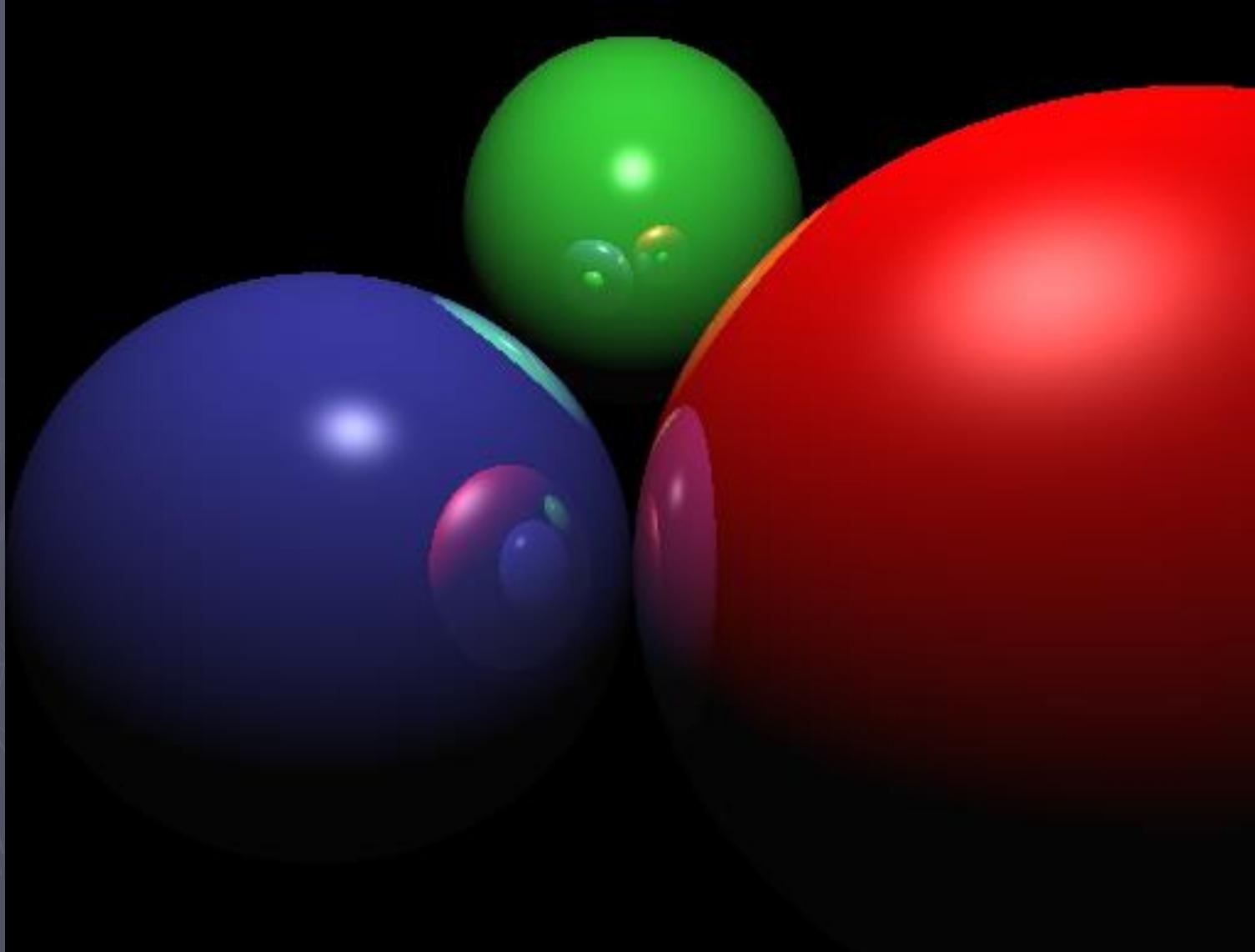
Key Idea 4: Recursive Reflection (1)

(Images From Univ. Delaware)

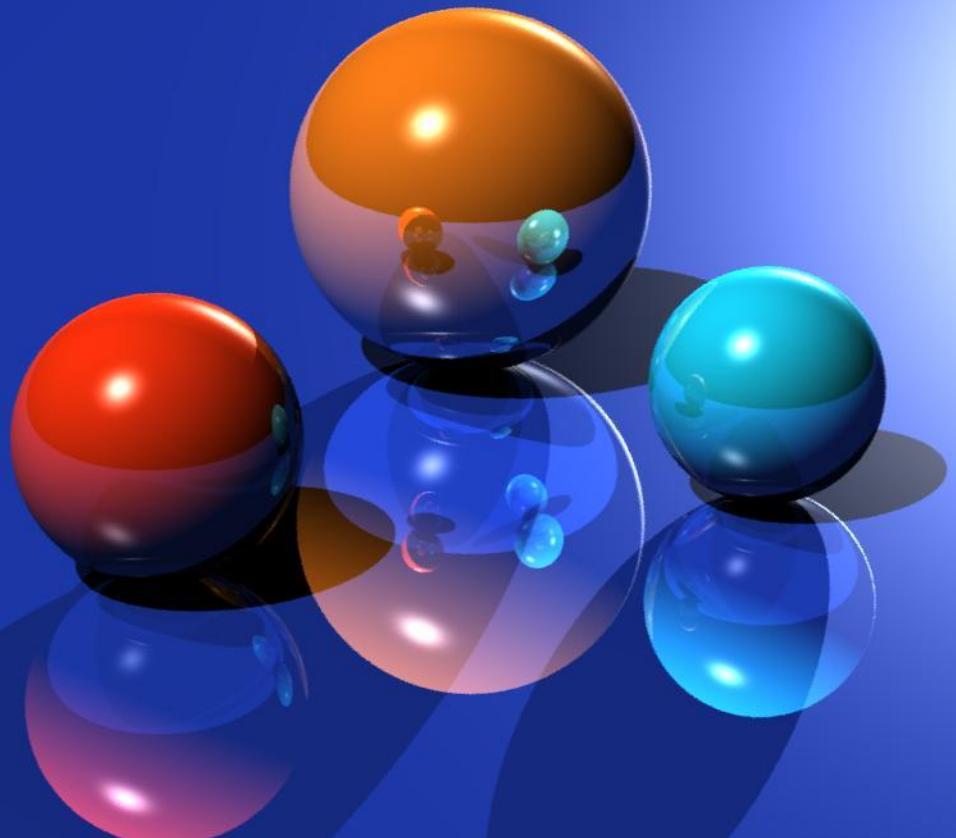


Key Idea 4: Recursive Reflection (2)

(Images From Univ. Delaware)



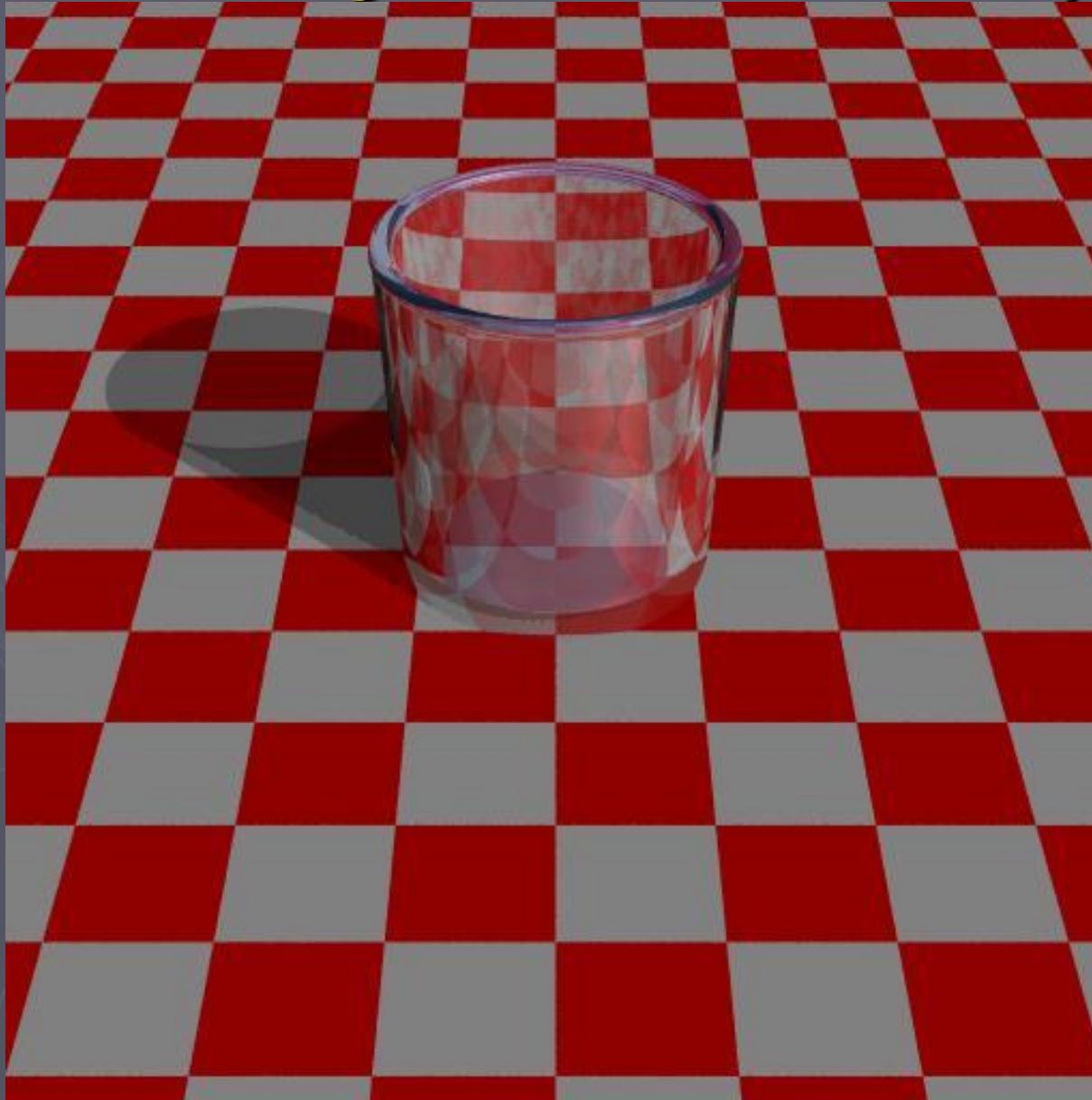
Project B: You will do this...



Cool circa 1990s,

- Fewer Lighting hacks
- Better organized code
- Still synthetic looking

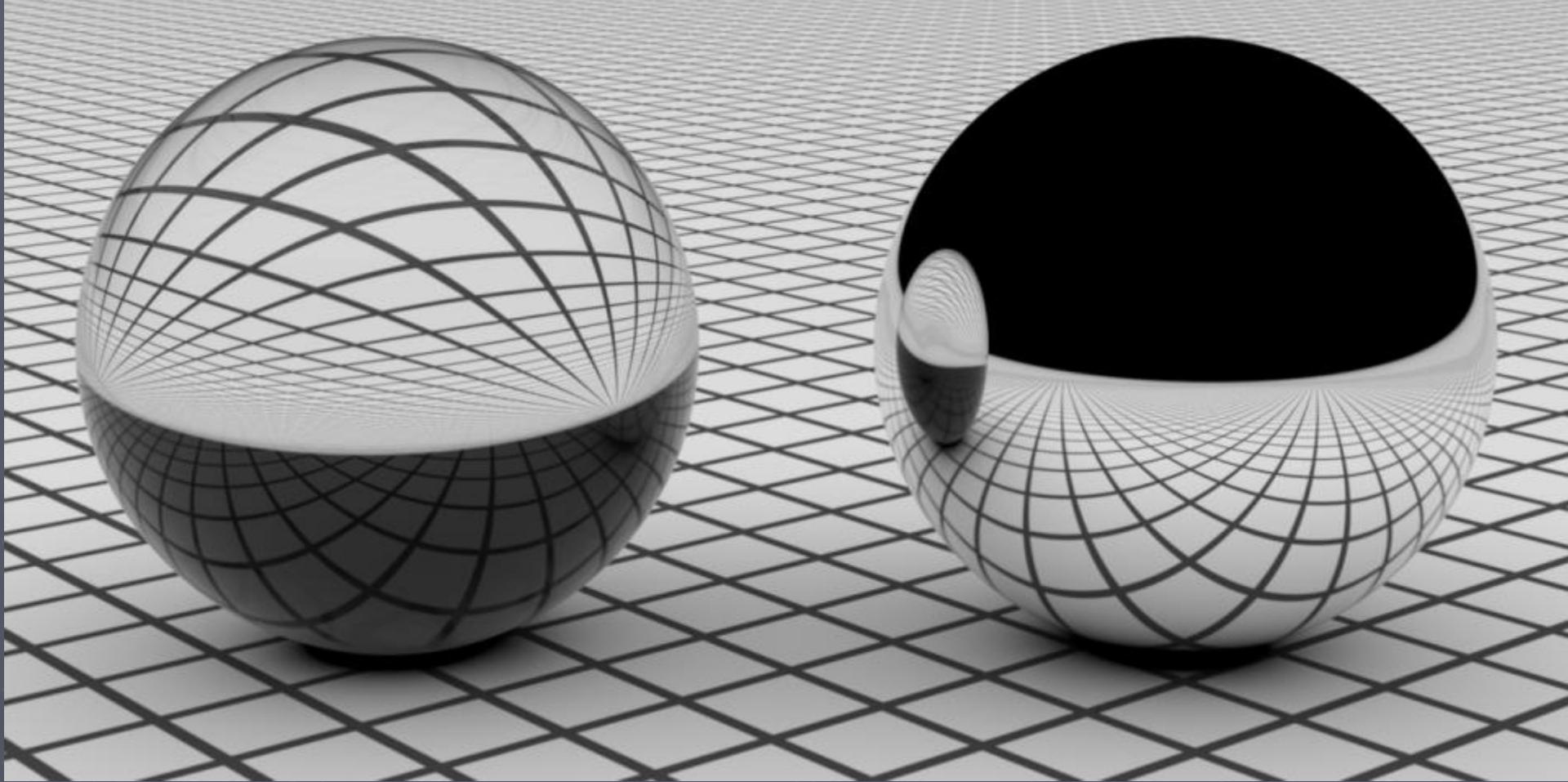
Project B: And maybe this:



Cool circa late 1990s,

- Procedurally-defined shapes
- Better Materials models
- Better organized code
- Still synthetic looking

How? step-by-step, build this:



- Std. circa early 2000s,
- jittered super-sampling for Antialiasing (low-noise)
- Extended 'soft' lights
- Code extensible to global illumination
- More natural-looking shadows
- FROM 'the PBRT book': <http://www.pbrt.org/>

In A System Extendable to this:



- Wavelength-dependent refraction (rainbows from diamonds)
- FROM 'the PBRT book': <http://www.pbrt.org/>

GREAT book! (free!)

And Even This: (More Ambitious)

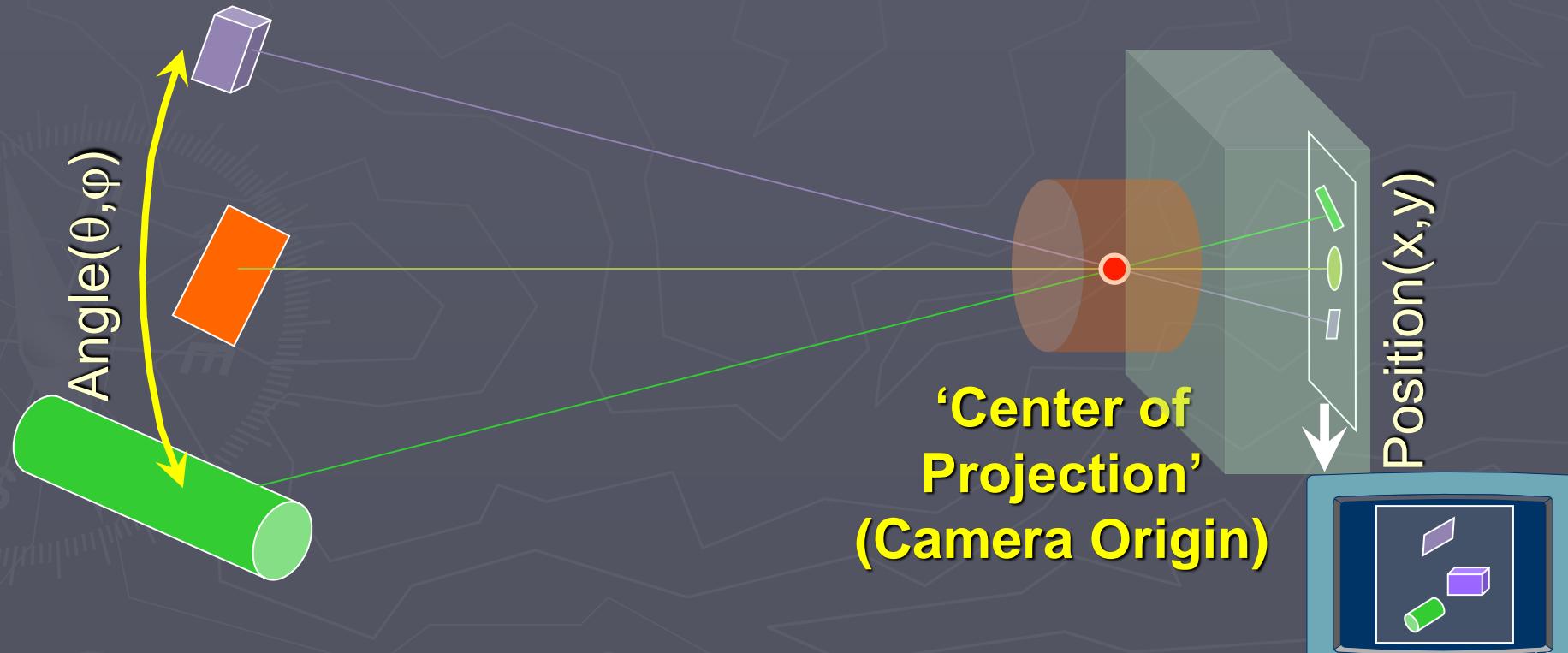


Many more in
the PBRT gallery:

<https://pbrt.org/gallery.html>

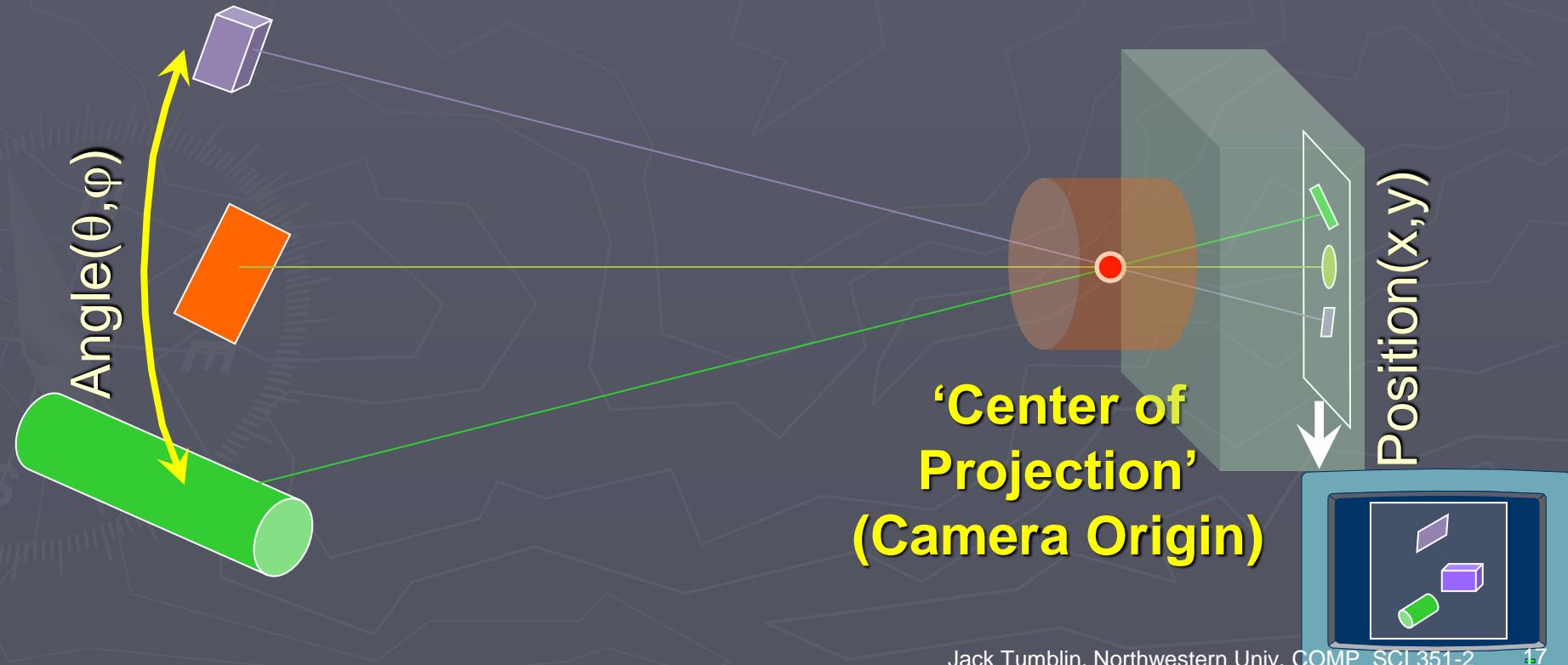
BASICS (1): What defines a 'camera' ?

- ▶ **Center of Projection:**
 - all camera rays pass through it:
 - used as the 'origin' of the camera coordinate system
- ▶ **Angle in 3D Scene \leftrightarrow position on image plane**



BASICS (2): How much light in a 'Ray'?

- ▶ Center of Projection:
 - all camera rays pass through it:
 - used as the 'origin' of the camera coordinate system
- ▶ Angle in 3D Scene \leftrightarrow position on image plane



BASICS (2): How much light is in a Ray?

- Center of Projection:
 - all camera rays pass through it
 - used as the 'origin'
- Angle in view:

Hmmmm....
Not so easy;

- A ray is **DOUBLY INFINITESIMAL**
(Area, Direction)
- With a **Maddening Measurement Quirk**
(cosine falloff)

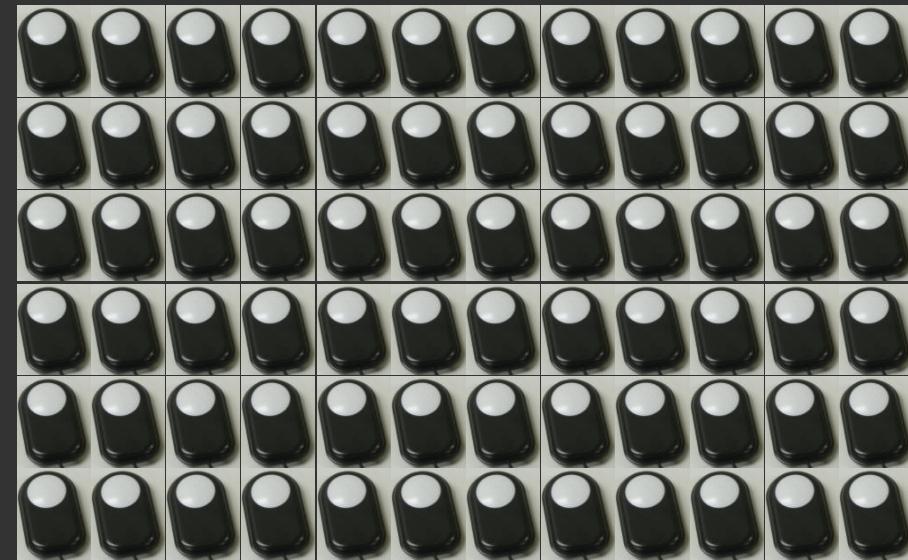
Here's why...

'Center of
Projection'
(Camera Origin)

Position(x, y)



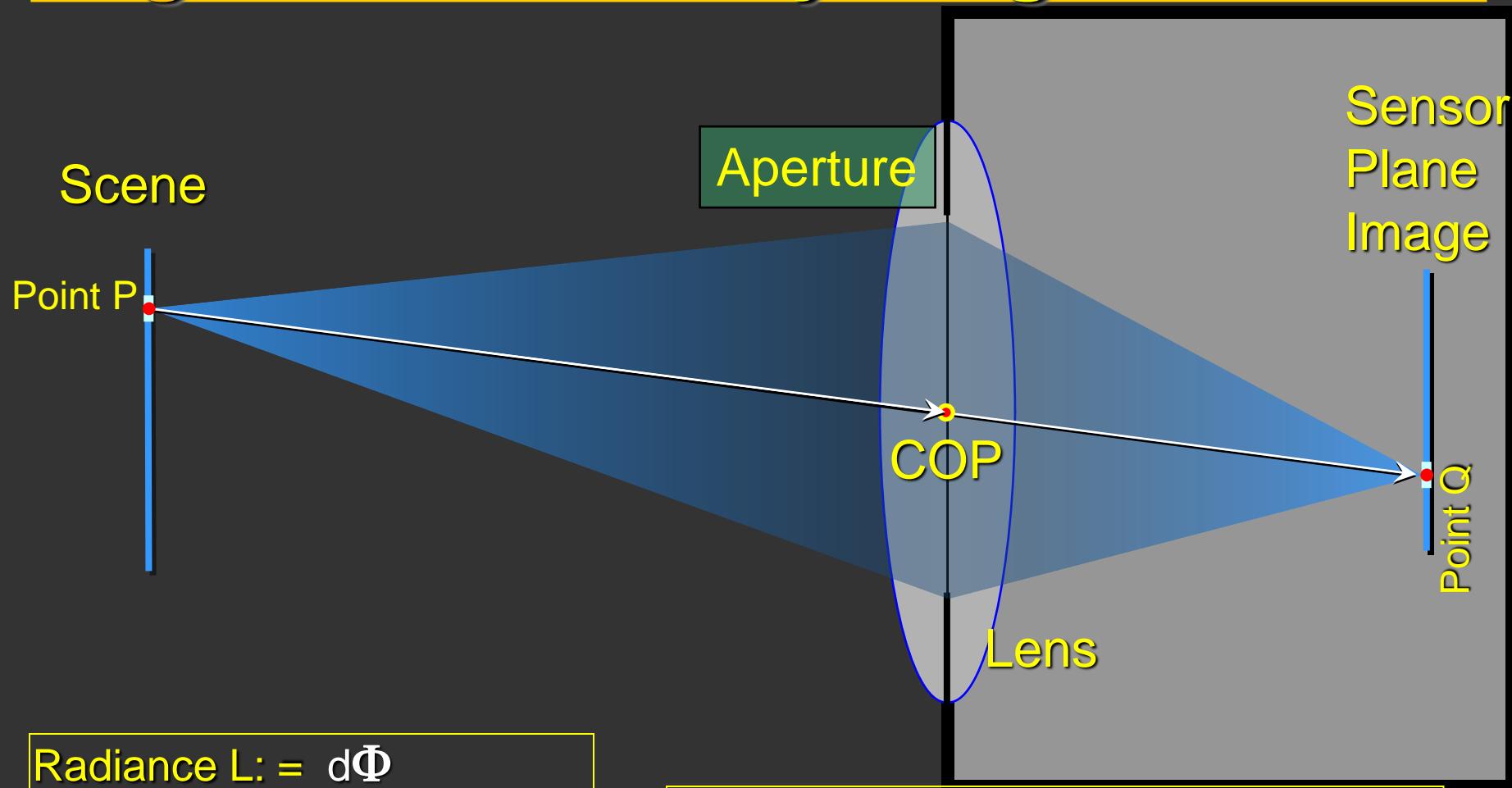
Camera Sensor: Array of Light Meters



Ideal Sensor Goals:

- *Instantaneous* Irradiance (E watts/m²) measure
- *Unlimited* resolution;
- *Unlimited* sensitivity,
- *Unlimited* Dynamic Range
- *Zero noise, perfect* accuracy

But a ‘ray’ transports ‘Radiance’: Integrate ‘cone’ of rays to get Irradiance



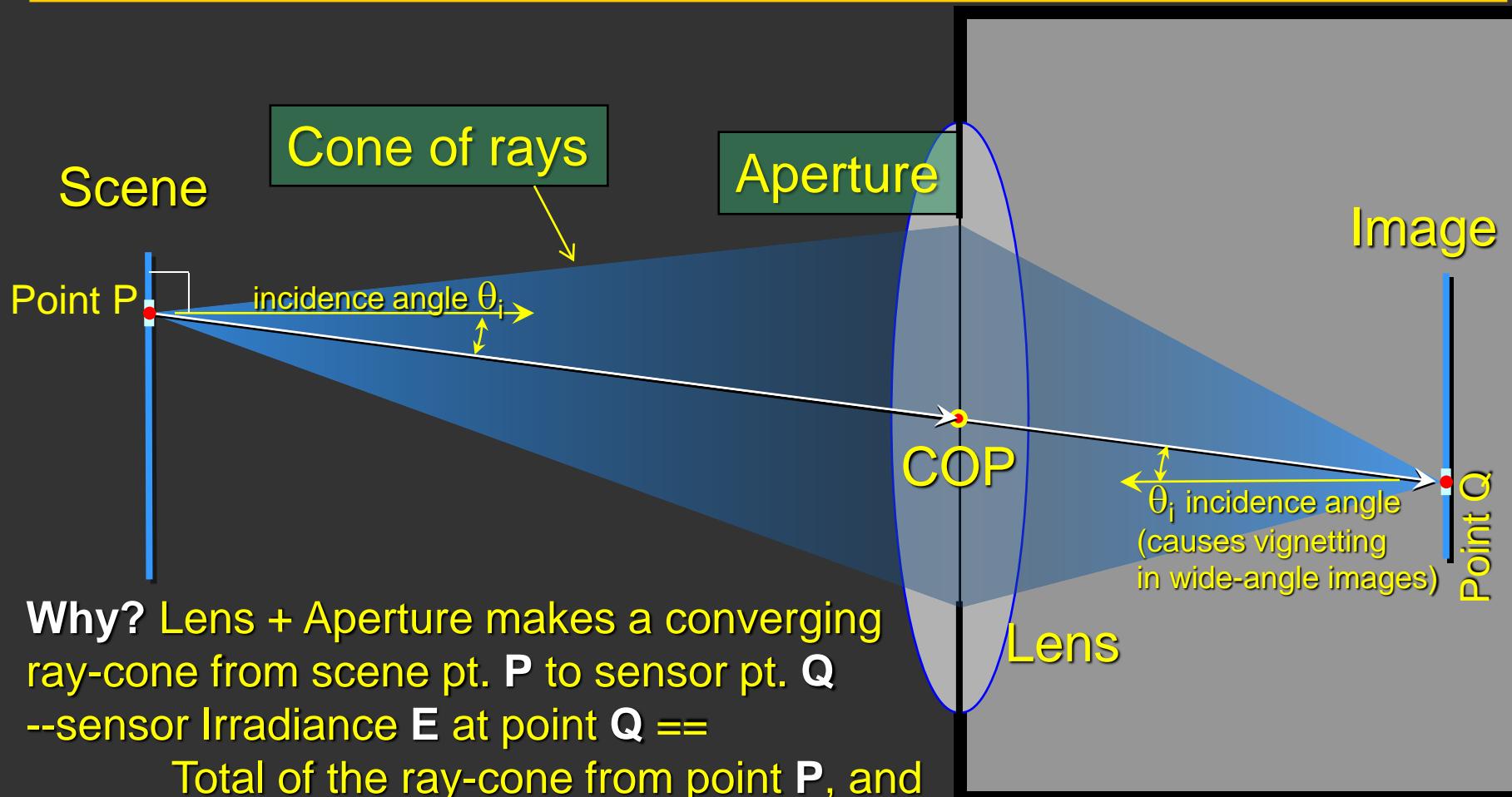
$$\text{Radiance } L := \frac{d\Phi}{dA d\omega \cos \theta_i}$$

$L = \text{watts} / (\text{meter}^2 \text{ steradian})$

$L = (\text{w/m}^2) / \text{sr} = (\text{w/sr})/\text{m}^2$...versus ... E or $I = \text{watts/meter}^2$

$$\text{Irradiance } E := \iint_0^{cone} \text{Radiance}(\theta, \varphi) d\theta d\varphi$$

Pixel Values? approx. proportional to Radiance



Why? Lens + Aperture makes a converging ray-cone from scene pt. **P** to sensor pt. **Q**

--sensor Irradiance **E** at point **Q** ==

Total of the ray-cone from point **P**, and

--if **P** is diffuse, radiance is constant for all rays in the cone, and

--if we shrink the aperture to a 'pinhole', ray-cone shrinks to a single ray.

Radiance **L**: $d \text{ (watts / meter}^2) / \text{steradian} = (\text{w/m}^2) / \text{sr}$

--THUS--

use Radiance **L**

to measure the amount of light
in each ray of your ray-tracer.

(Really? That's it? That's 'how much light in a ray'?

Yes—that's it; that's all.)

? Then why is Ray-Tracing so often unitless ?

(ANSWER: Lenses + Fear & Confusion)

Pixel Intensity? R+G+B? or what?

► Measurement conundrum: MKS units?

- How can we measure the light in just one ray?
(doubly-infinitesimal: watts per unit area per unit solid-angle)
- How do we convert 'ray strength' to RGB values?

► A) WebGL's 'display-units-only' practice: $(0.0 \leq R, G, B \leq 1.0)$ for **EVERYTHING?!**

- Light Src: ambient,diffuse,specular RGB
- Materials: ambient,diffuse,specular,emissive RGB
- Textures: ambient,diffuse,specular,emissive RGB
- Displays: pixel RGB = Light*Materials & Texture

► B) Ray Trace with RGB 8-bit light? Oh please **NO!**

MKS for Accurate 'Visual Strength'

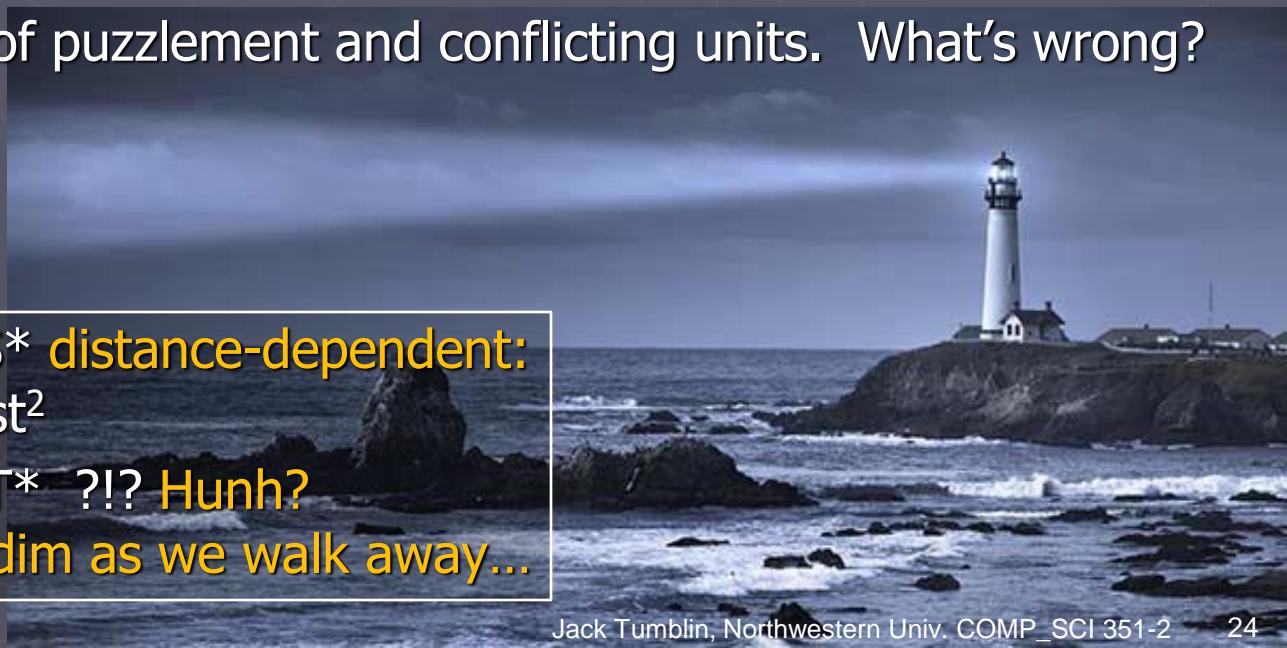
Well, what 'visual strengths' do our eyes see?

- *NOT* additive! (TRY IT: turn on one light, then two, then three. Equal 'Brightness' steps?) (? If perception was linear, then why do night-time stars vanish in daylight?)
- light's 'visual strength' $\approx \log(\text{power})$: I see *ratios* of light!
- great for 'lightness' (perceived reflectance: %-age!), but
- quite misleading for 'brightness' (perceived intensity)

Measured 'Light strength' ? !A Nasty Conundrum!

- Long (boring) history of puzzlement and conflicting units. What's wrong?

- And it is *SOMETIMES* distance-dependent:
$$\text{power} = P_{\max}/\text{dist}^2$$
- But *SOMETIMES NOT* ?!? Hunh?
Room walls don't get dim as we walk away...

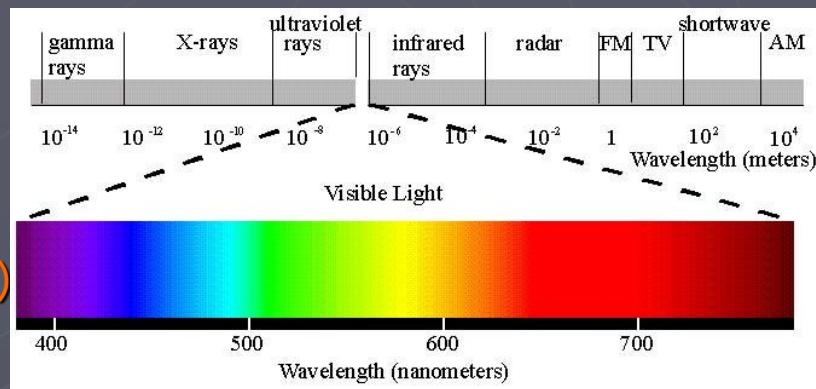
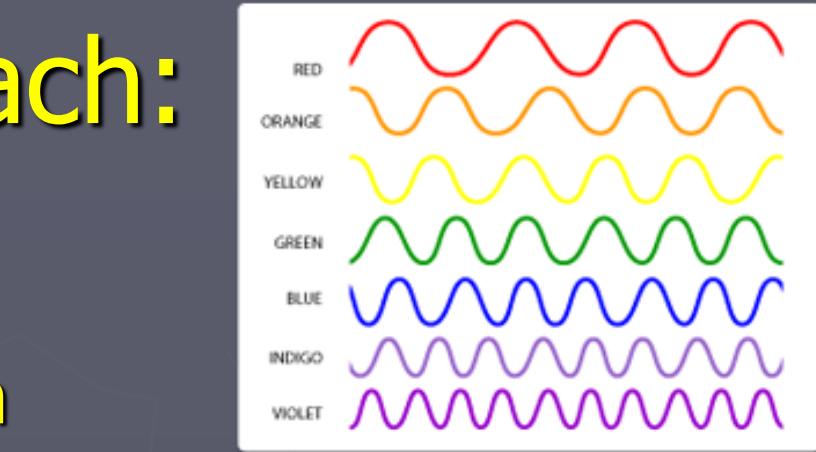


A principled approach:

(without any calculus!)

What, exactly, is light?

- Narrow-band EM radiation
- Fast, line-like Propagation Of Energy -- model as
- “rays” == photon-paths
- Wave/Particle Duality
(that we ignore, mostly. Would anything look *wrong* if rendered without diffraction effects?)
- 3-D rays from a 4-D set



**Wait, Wait, Wait – what's that again?
– set of all 3D rays forms a FOUR-D set?
(a brief Digression)**

Review: How many Rays in a 3-D Scene?

**4-D set of infinitesimally distinct rays:
uncountably infinite.**

(Levoy et al. SIGG'96)

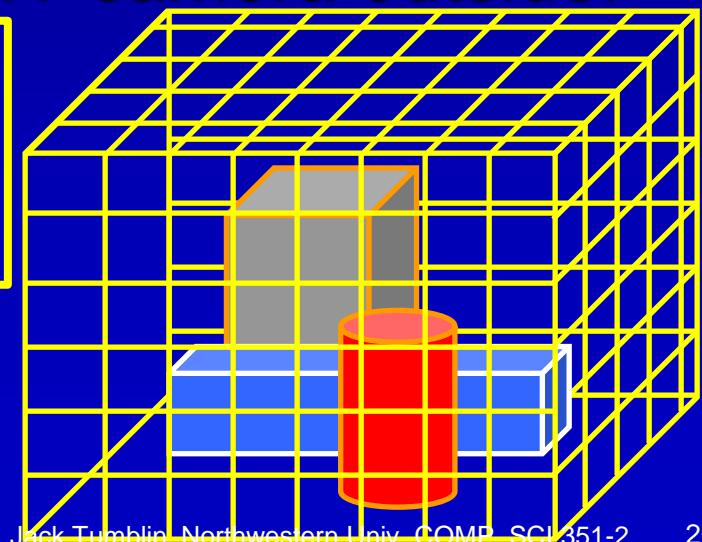
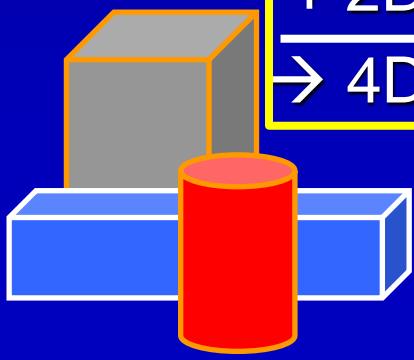
(Gortler et al. '96)

Imagine:

- Convex Enclosure of a 3D scene
- Inward-facing ray camera at every surface point
- Pick the rays you need for ANY camera outside.

2D surface of cameras,
+ 2D ray set for each camera,
 \rightarrow 4D set of rays.

4D view rays
+4D illumination rays
8D scene reflectance

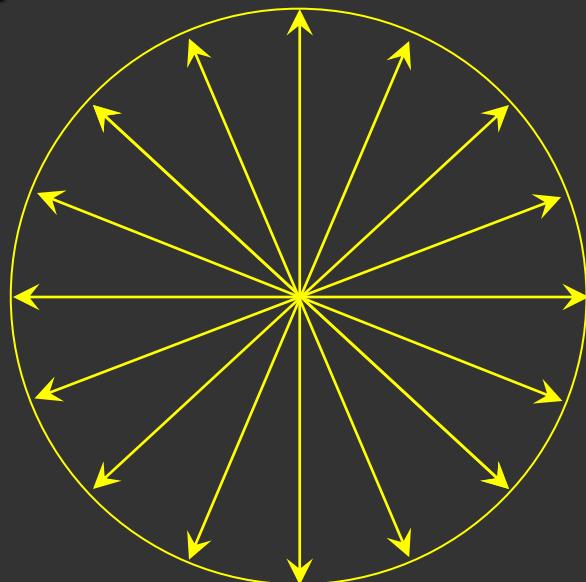


Point-Light Source inside a Sphere

- Imagine a point-like Lamp that emits **W** watts of light flux,
- Enclose the lamp in sphere of radius **r** :

A) How much ‘Light’ arrives at surface?

B) How much ‘Light’ leaves the surface?



Radiant Flux $\Phi == W$ watts
Sphere surface area: $A = 4\pi r^2$

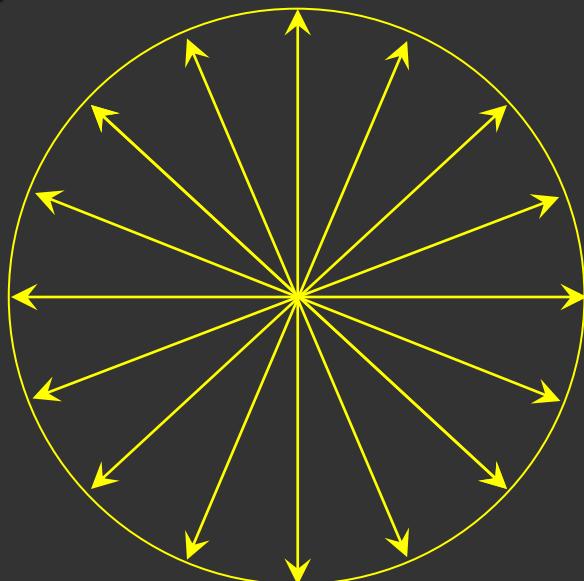
Point-Light Source inside a Sphere

- Imagine a point-like Lamp that emits **W** watts of light flux,
- Enclose the lamp in sphere of radius **r** :

Material: Perfect Glass?
Perfect Black?
Perfect White?
Perfect Mirror?

A) How much ‘Light’ arrives at surface?

B) How much ‘Light’ leaves the surface?



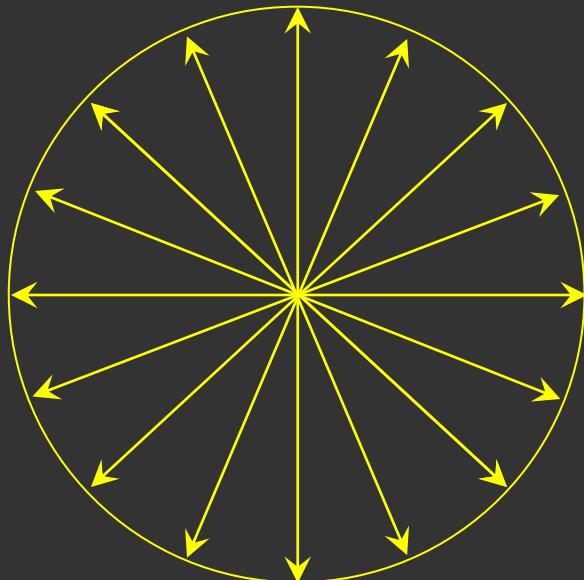
Two ways to ‘leave’ the sphere surface:
TRANSMISSION or b) by **REFLECTION**

ONE RAY from the light point pierces each sphere point

Radiant Flux $\Phi = W$ watts
Sphere surface area: $A = 4\pi r^2$

Flux Φ and Irradiance E on a Sphere

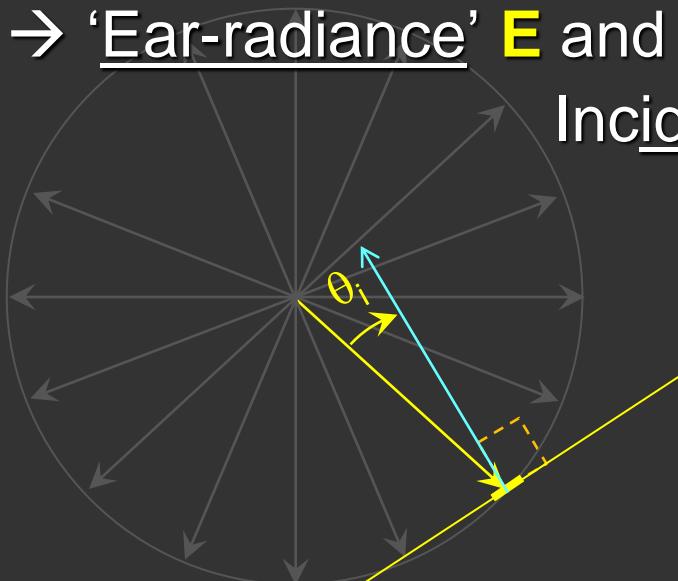
- Imagine a point-like Lamp that emits **W** watts of light flux,
- Enclose that lamp in sphere of radius **r**, where all the light
- A) Enters ^{imaginary} **surface** with →‘Ear-radiance’ == **E**
- B) Leaves ^{imaginary} **surface** with →‘Radiant Intensity’ == **I**
MKS or ‘SI’ units: (watts/meter²) for BOTH E and I



Radiant Flux Φ == W watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A =$
 $W / 4\pi r^2$ watts/meter²

Flux Φ and Irradiance E on a Plane

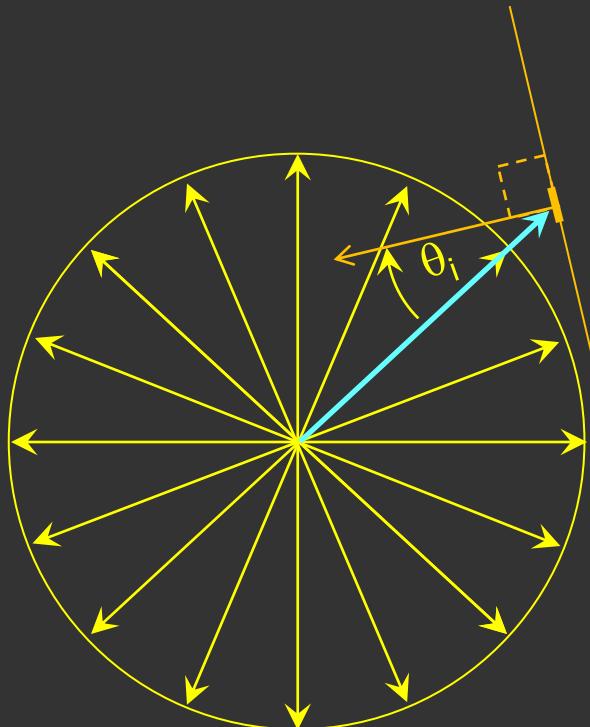
- Return to the point-like Lamp that emits **W** watts of light flux,
- Enclose the lamp in sphere of radius **r**, then
- Replace a small (infinitesimal) patch of the sphere with a tiny planar patch, and TILT that patch by angle θ_i
- This spreads the same incident light over a larger area:
→ ‘Ear-radiance’ **E** and ‘Radiant Intensity’ **I** also depend on
Incident Angle θ_i , aka ‘cosine falloff’ rule:



Radiant Flux $\Phi = W$ watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A = I =$
 $W \cdot \cos(\theta_i) / 4\pi r^2$ watts/meter²

How much light does a ray carry?

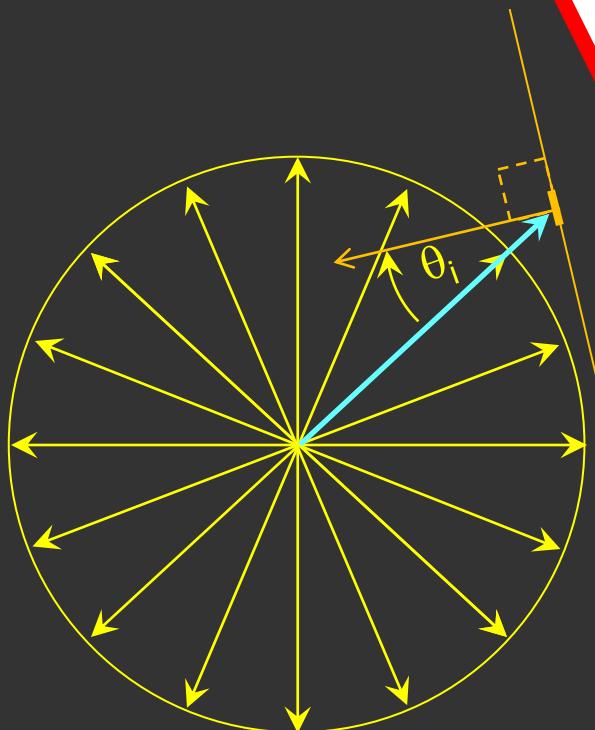
- ? Does Irradiance E (== radiant intensity I) measure the ray-strength for a point-source light?
- ? Should ray-strength measures include cosine-falloff? Distance from light source?



Radiant Flux $\Phi = W$ watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A = I =$
 $W \cdot \cos(\theta_i) / 4\pi r^2$ watts/meter²

How much light does a ray carry?

- ? Does Irradiance E (== radiant intensity) measure the ray-strength for a point light?
- ? Should ray-strength model cosine-falloff?



Cosine Falloff for Ray !NO!

Surface arrival-angle θ_i
isn't part of the ray –
it's set by the surface patch!

Distance Falloff for Ray?
Yes for Lights, No for Cams?!?!

→ 'camera pixel approximates ray'
Eye-to-wall distance has no effect...
THUS

So !NO! – ray has no distance falloff

watts

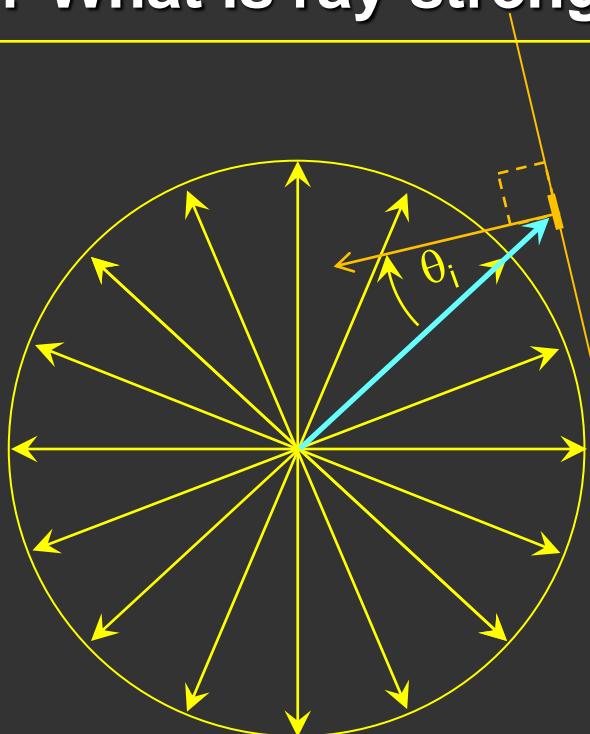
Surface area: $A = 4\pi r^2$

Irradiance $E = \Phi/A = I =$

$W \cdot \cos(\theta_i) / 4\pi r^2$ watts/meter²

How much light does a ray carry?

- ? Does Irradiance E (== radiant intensity I) measure the ray-strength for a point-source light?
- cosine-falloff? **NO!** Distance falloff? **NO!**).
- ? What is ray-strength for a *larger lamp*? (not a point?)



Radiant Flux $\Phi = W$ watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A = I =$
 $W \cdot \cos(\theta_i) / 4\pi r^2$ watts/meter²

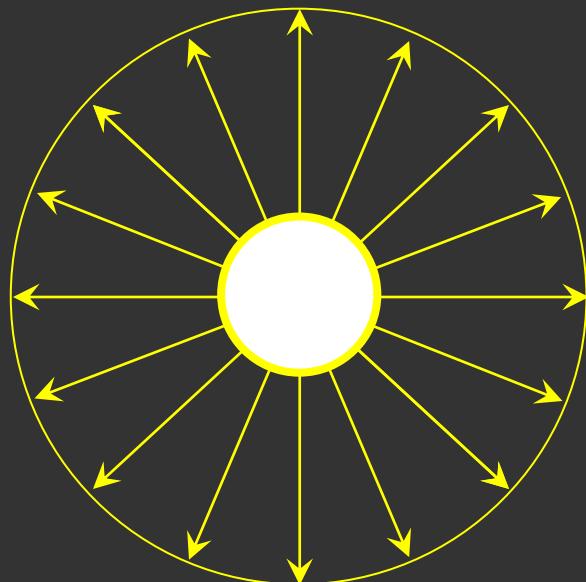
Effect of Ray's Arrival Direction(s)?

Imagine a **spherical** Lamp that emits **W** watts of light flux,

Enclose the lamp in an **imaginary(invisible)** sphere of radius **r** :

- ? Does the sphere still receive uniform irradiance **E**?

???



Radiant Flux $\Phi = W$ watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A =$
??? **watts/meter²**

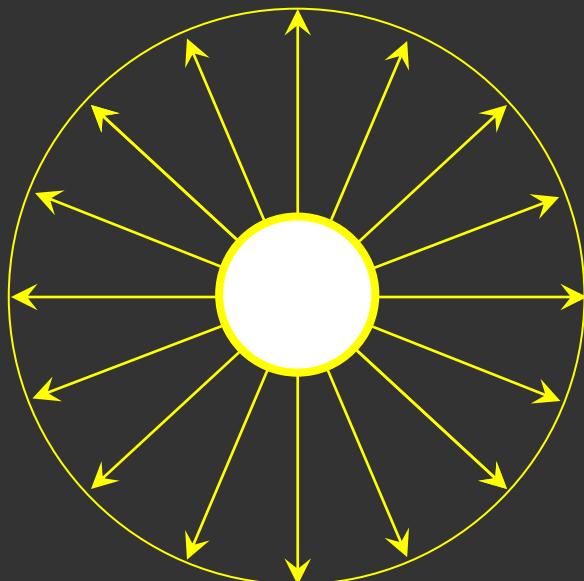
Effect of Ray's Arrival Direction(s)?

Imagine a **spherical** Lamp that emits **W** watts of light flux,

Enclose the lamp in an **imaginary(invisible)** sphere of radius **r** :

- ? Does the sphere still receive uniform irradiance **E**?

YES: same emitted power **W** hits the same area $4\pi r^2$



Radiant Flux $\Phi == W$ watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A =$
 $W / 4\pi r^2$ watts/meter²

SO: How much light does a ray carry?

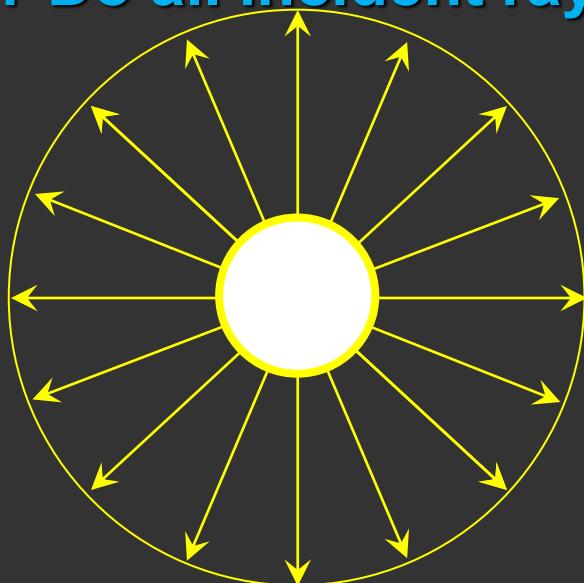
Imagine a **spherical** Lamp that emits **W** watts of light flux,

Enclose the lamp in an **imaginary(invisible)** sphere of radius **r** :

- ? Does the sphere still receive uniform irradiance **E**?

YES: same emitted power **W** hits the same area $4\pi r^2$

- ? Do all incident rays still have uniform ‘strength’ ?



Radiant Flux $\Phi = W$ watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A =$
 $W / 4\pi r^2$ watts/meter²

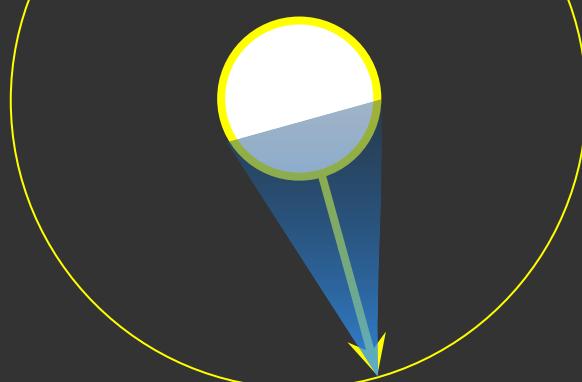
SO: How much light does a ray carry?

Imagine a **spherical** Lamp that emits **W** watts of light flux,

Enclose the lamp in an **imaginary(invisible)** sphere of radius **r** :

- ? Does the sphere still receive uniform irradiance **E**?
YES: same emitted power **W** hits the same area $4\pi r^2$
- ? Do all incident rays still have uniform ‘strength’ ?

****UNKNOWN!**:** no longer just 1 ray from 1 direction --
each point gets a cone-shaped bundle of rays



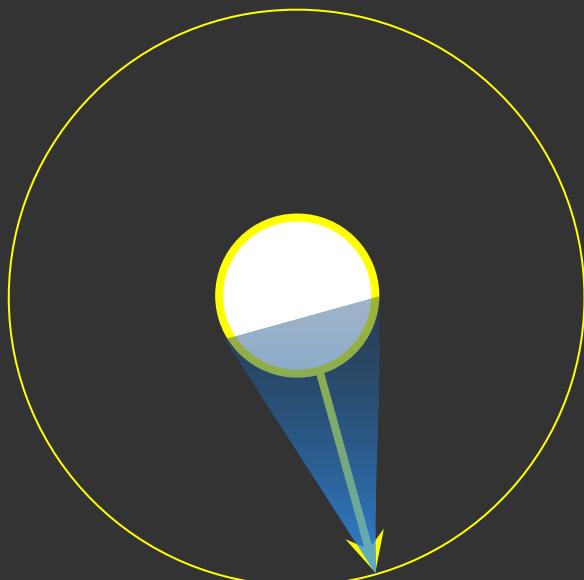
Radiant Flux $\Phi = W$ watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A =$
 $W / 4\pi r^2$ watts/meter²

SO: How much light does a ray carry?

!AHA! A 3D BUNDLE of rays carries total irradiance E !!

THUS ‘Ray Strength’ must *distribute* or *divide* the watts-per-unit-area (irradiance E) across the cone of rays, a **3D span-of-directions...**

? Wait, wait, wait... How can we measure a ‘3D span-of-directions’?



Radiant Flux $\Phi == W$ watts
Sphere surface area: $A = 4\pi r^2$
Irradiance $E = \Phi/A =$
 $W / 4\pi r^2$ watts/meter²

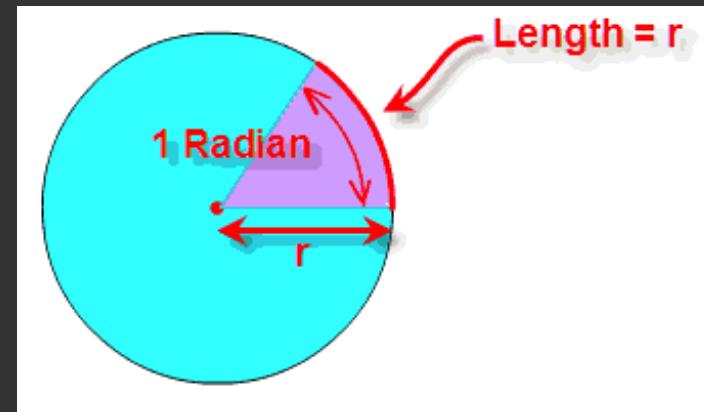
How to Measure Angles: in 2D...

Measure a **span-of-directions** in **2D**? \equiv “Angle”

- Construct a unit circle ($r=1$) around 2D origin:

$$x^2 + y^2 = 1^2$$

DETAILS: Every **point** on circle specifies
one unique **direction** from origin; ALSO
Every **direction** from origin specifies
one unique **point** on the circle.
(formally: a **bijection**: **one-to-one** && **onto**)

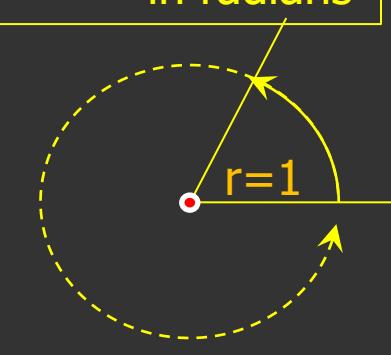


- ‘Angle’ measures a 1D span of directions by their 1D span of **distance** on the unit **circle**

distance == angle
in radians

- entire distance on unit-radius circle: $2\pi * \text{radius}$
entire span of all angles:

2π radians



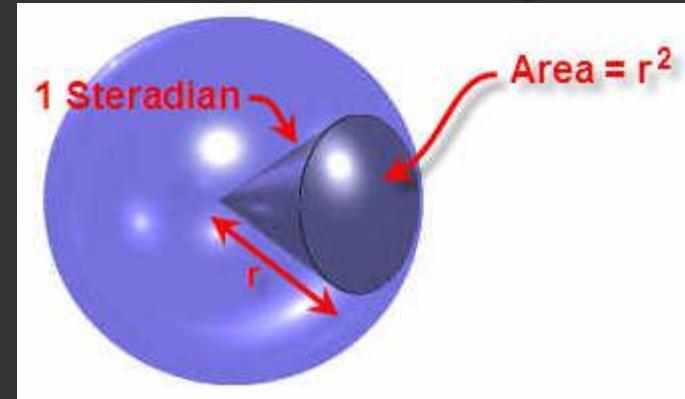
How to Measure Angles: in 3D

Measure a span-of-directions in 3D? \equiv “Solid Angle”

- Unit sphere ($r = 1$) around 3D origin:

$$x^2 + y^2 + z^2 = 1^2$$

DETAILS: Every point on sphere specifies
one unique direction from origin; ALSO
Every direction from origin specifies
one unique sphere point.
(formally: a **bijection**: one-to-one && onto)



area == solid angle
in steradians

- ‘Solid Angle’ measures a 2D span of directions by their 2D span of surface area on the unit sphere
- entire surface area on unit sphere: **4π area in radius² units**
entire span of all directions: **4π steradians**
see: <http://www.mathsisfun.com/geometry/steradian.html>
and: <http://apps.usd.edu/coglab/schieber/trb2000/sld021.htm>

SO: How much light does a ray carry?

'Ray Strength'

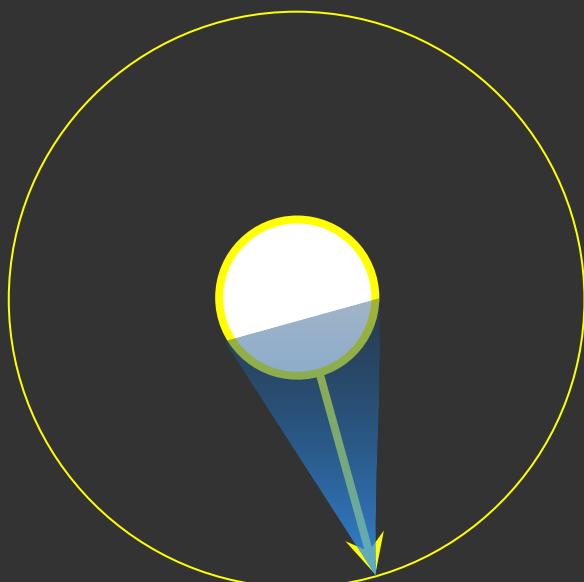
or 'light – power – from – one – point – in – one – direction'

RADIANCE **L** measures

doubly-differential power density:

$$L = E / \text{steradian} \text{ (solid angle)}$$

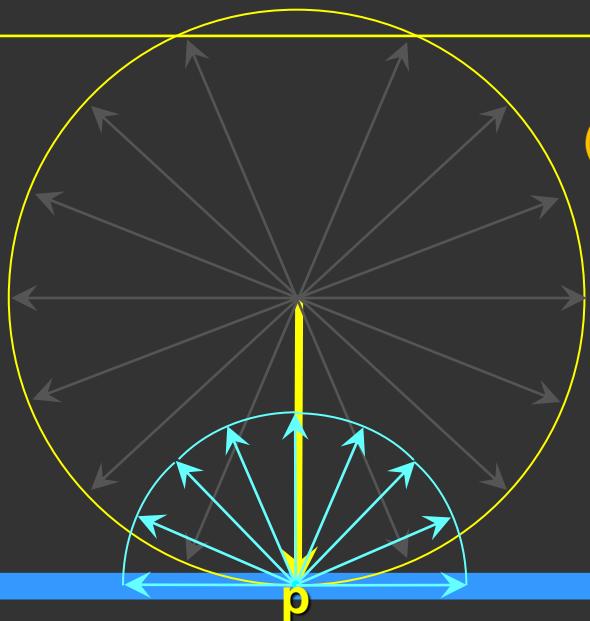
$$L = (\text{Watts} / \text{meter}^2) / \text{steradian}$$



"Ah, good! Got it, makes sense,
no problem, we're done, right?"

The NIGHT-LIGHT PUZZLE: How much light does it emit (or reflect) ?

- Pretty Blue Night-Light (an ‘electroluminescent plate’ lamp) looks uniform from all directions, lit or unlit
- Pick a **single point ‘p’** on the surface. How much light does that one infinitesimal point emit? (or if unlit: how much light does it reflect?)
- Does each POINT emit the same ‘light strength’ in all directions?



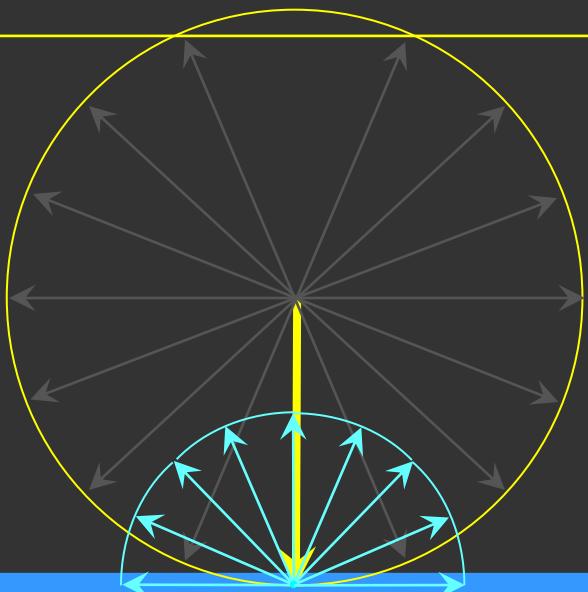
(Hmm. Yes, it does *LOOK* uniform in all directions.
Let's say 'yes: point emits same light in all directions'
OK, then suppose we tilt the lamp by 60° ($\cos 60^\circ=0.5$)
We still see all the same surface points,
but at this angle they're closer together by 2X.
Won't the surface look 2X brighter? Why not?

EL Night-light surface ↘

The NIGHT-LIGHT PUZZLE: How much light does it emit? (or reflect) ?

- Pretty Blue Night-light
has uniform light emission
- Pick a surface point.
How much light does it emit?
(or if unlit, how much light does it reflect?)
- Does each POINT emit the same ‘light strength’
in all directions?

!!!BZZT!!! INVALID QUESTION!
“ray strength”
is not clearly defined
by area and light directions alone!!!

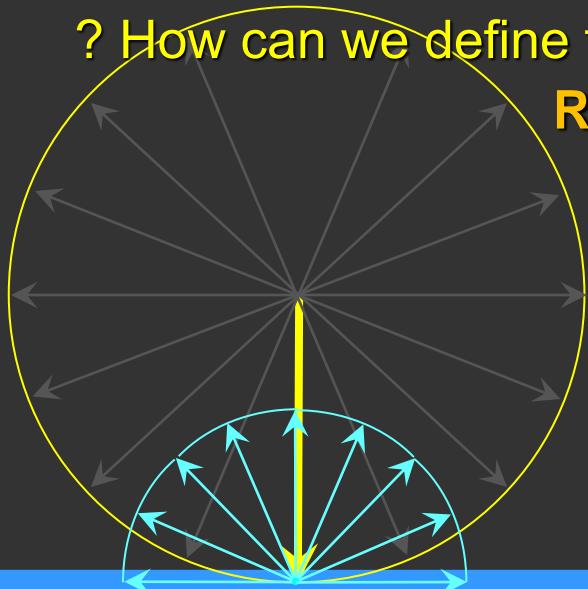


(Hmm. Yes, it does *LOOK* uniform in all directions.
Let’s say ‘yes:point emits same light in all directions’.
OK, then suppose we tilt the lamp by 60° ($\cos 60^\circ=0.5$)
We still see all the same surface points,
but at this angle they’re closer together by 2X.
Won’t the surface look 2X brighter? Why not?)

EL Night-light surface ↗

The NIGHT-LIGHT PUZZLE: How much light does it emit (or reflect)?

- !?! WHUUTT ?!? ?Must we define a ‘directional’ strength?
 - Surface point has zero area: emits/receives *infinitesimal* power, AND watts emitted = $\text{Area} * E = (0^+ \text{ meter}^2) * (\text{W} \cdot \cos(\theta) / 4\pi r^2)$ (Watts/meter²)
 - The infinitesimal power that leaves this point
!! spreads out in all directions !!
 - ? *What fraction* of that infinitesimal light power did your eye capture?
? How can we define the ‘light strength’ in just **one** direction?!?



RECALL: Irradiance E has a ‘cosine falloff’ term.

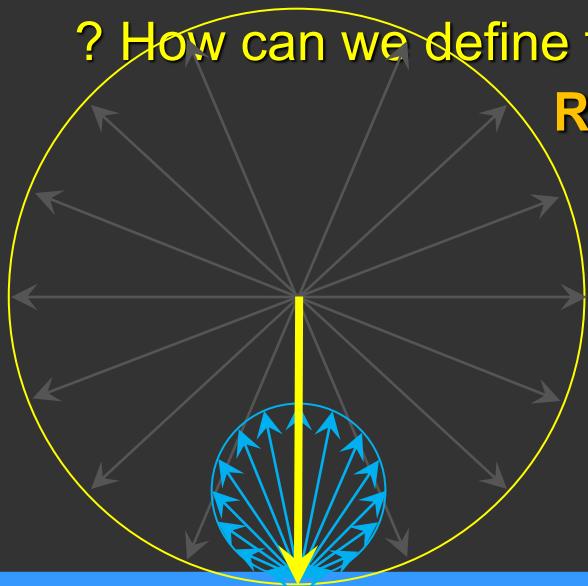
Wouldn’t the ‘light strength’ of one ray
also have a **cosine falloff** ?
or would it remain **constant** with angle?

Shown here:

CONSTANT or **COSINE FALLOFF**
EL Night-light surface

The NIGHT-LIGHT PUZZLE: How much light does it emit (or reflect)?

- **!!!BZZT!!!** INVALID QUESTION! define ‘directional’ strength
 - Surface point has zero area: emits/receives *infinitesimal* power, AND watts emitted = $\text{Area} * E = (0^+ \text{ meter}^2) * (\text{W} \cdot \cos(0) / 4\pi r^2) (\text{Watts/meter}^2)$
 - The infinitesimal power that leaves this point
!! spreads out in all directions !!
 - ? *What fraction* of that infinitesimal light power did your eye capture?
? How can we define the ‘light strength’ in just **one** direction?!?



RECALL: Irradiance E has a ‘cosine falloff’ term.

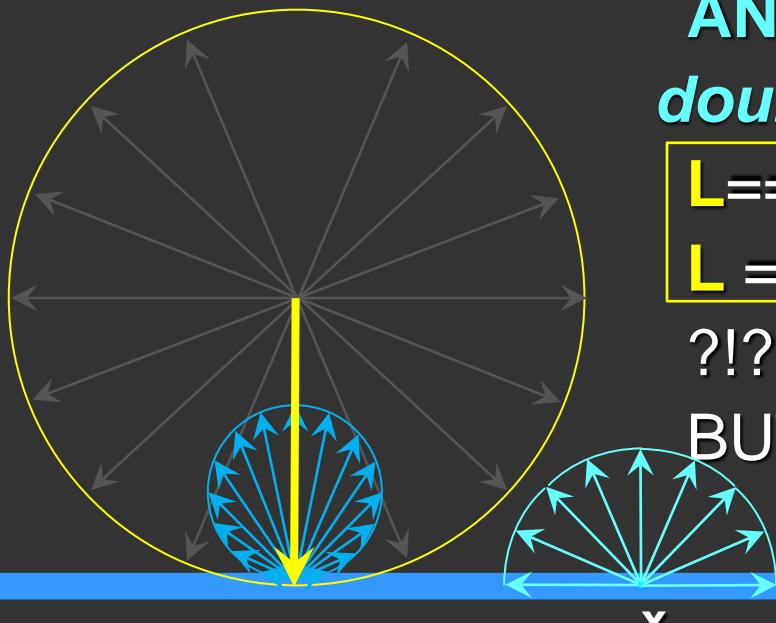
Wouldn’t the ‘light strength’ of one ray
also have a **cosine falloff** ?
or would it remain **constant** with angle?

Shown here:

CONSTANT or **COSINE FALLOFF**
EL Night-light surface

The NIGHT-LIGHT PUZZLE: How much light does it emit (or reflect)?

- Useful Thought Experiment:
 - All surface points receive(or emit) the same total irradiance **E** (or exitance **I**)
 - My eyes report the same surface ‘brightness’ viewed from any/all directions
 - That ‘brightness’ **doesn’t** follow the cosine-falloff rules of irradiance **E** (or **I**).
 - How can I describe ray light-strength to keep this fixed ‘observed brightness’?



ANSWER: **RADIANCE L** measures **doubly**-differential power density:

$$L == E / \text{steradian} \text{ (solid angle)}$$

$$L == (\text{Watts} / \text{meter}^2) / \text{steradian}$$

?!? does **E** have a ‘cosine falloff’ ?!?

BUT what we see from nightlight
stays constant – ?!?WHY!?

END

END



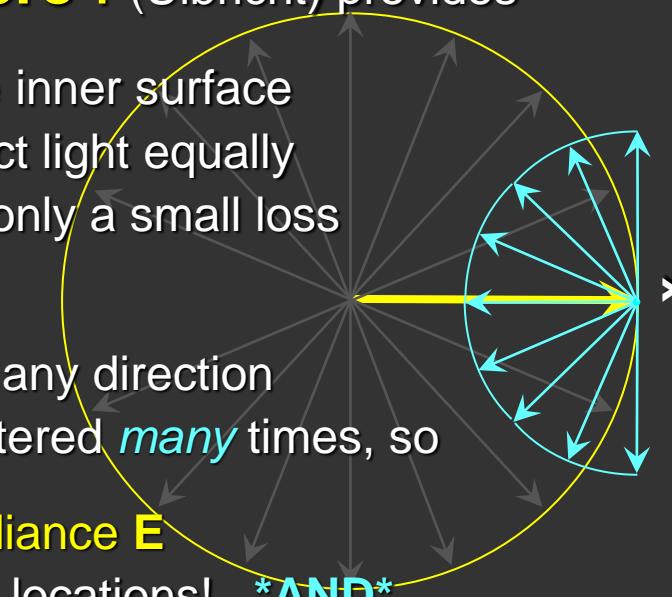
Inter-reflection – How much Light?

- **No ‘Perfect’ materials:** even the best-made surfaces reflect, absorb, transmit and scatter some fractions of incident light: → They all re-direct light energy (in several ways)

- **‘Integrating Sphere’:** (Ulbricht) provides

- uniform diffuse-white inner surface (Lambertian) to reflect light equally in all directions with only a small loss
- Any light entering from any location, in any direction gets reflected & scattered *many* times, so
- (nearly) **uniform irradiance E** result at all surface locations! ***AND***
- (nearly) **uniform radiance R** in all directions

http://en.wikipedia.org/wiki/Integrating_sphere



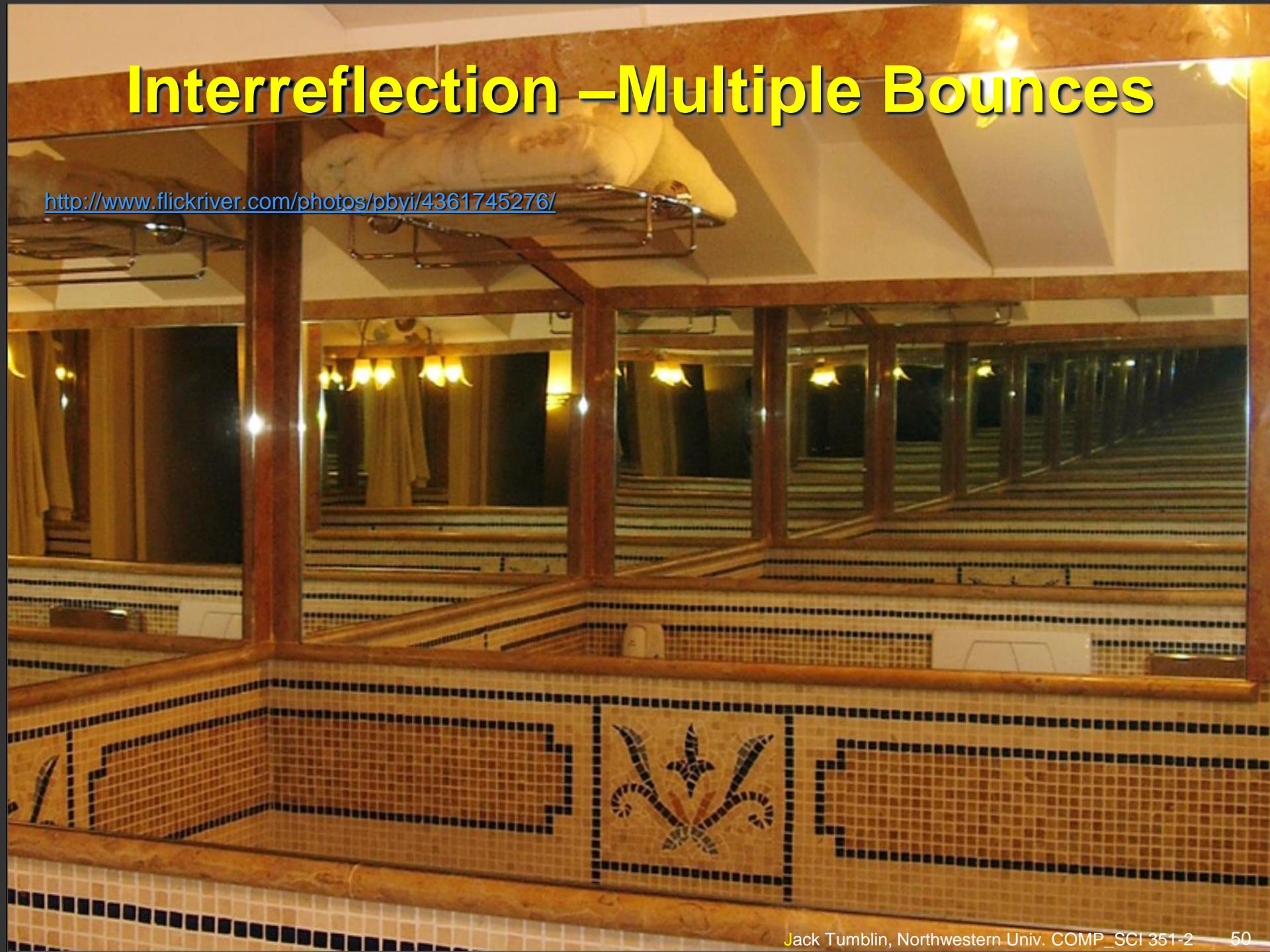
Sphere Materials & Reflectance?

Complete this Chart:

Material	Irradiance E (watts/m ²)	Radiant Intensity (or 'radiant exitance') I (watts/m ²)
• Perfect Glass	$W / 4\pi r^2$	$W / 4\pi r^2$
• Perfect Black	$W / 4\pi r^2$	0
• <i>Perfect Mirror</i>	?	?
• <i>Perfect White</i>	?	?
• Polished Gold	?	?
• Clean Paper	?	?

Interreflection –Multiple Bounces

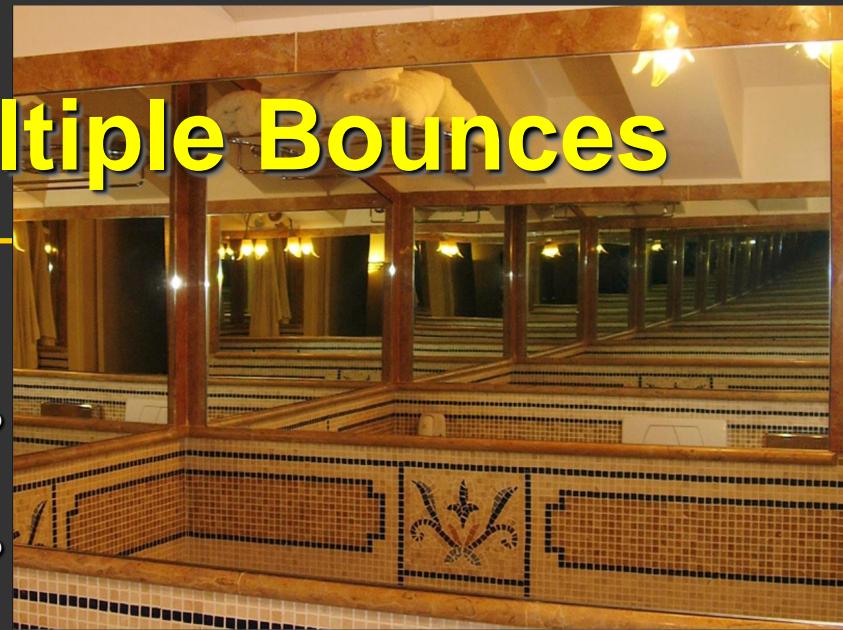
<http://www.flickrriver.com/photos/pbyi/4361745276/>



Interreflection –Multiple Bounces

Parallel Mirrors:

- ?HIGHEST reflectance mirrors you find?
how can we measure it?
how many bounces to lose 98% of light?
(humans can't see contrasts < ~1-2%)
- Typical bathroom mirror reflectance?
Estimate **R** ($0 \leq R \leq 1$) for these mirrors: <http://www.flickr.com/photos/pbyi/4361745276/>
https://www.flickr.com/photos/ogre_snap/4520226770/



CHALLENGE:

If the 1st reflection of the white box has pixel value of 200, and the 12th reflection has pixel value of 25, what's the reflectance of the mirror?

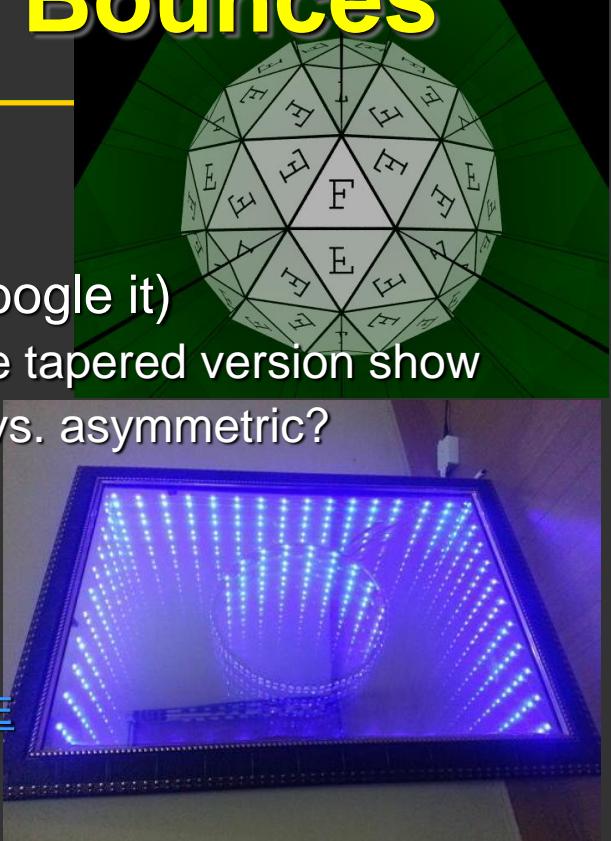
Interreflection –Multiple Bounces

Other Cyclic Reflection Uses:

- **Kaleidoscope?** Seek out those by **Ken Perlin** (google it)
2 sides? 3 sides? 4? more? mirror tube? What does the tapered version show you? What's result of equal-angles between all mirrors vs. asymmetric?

- **'Infinity Mirrors'?** (google it)
Why does tunnel 'curve'? can you 'twist' the tunnel?
Spiral it? measure the speed of light?

<https://youtu.be/VTOKZkaVX4?t=16s> <https://youtu.be/sAPGw0SD1DE>

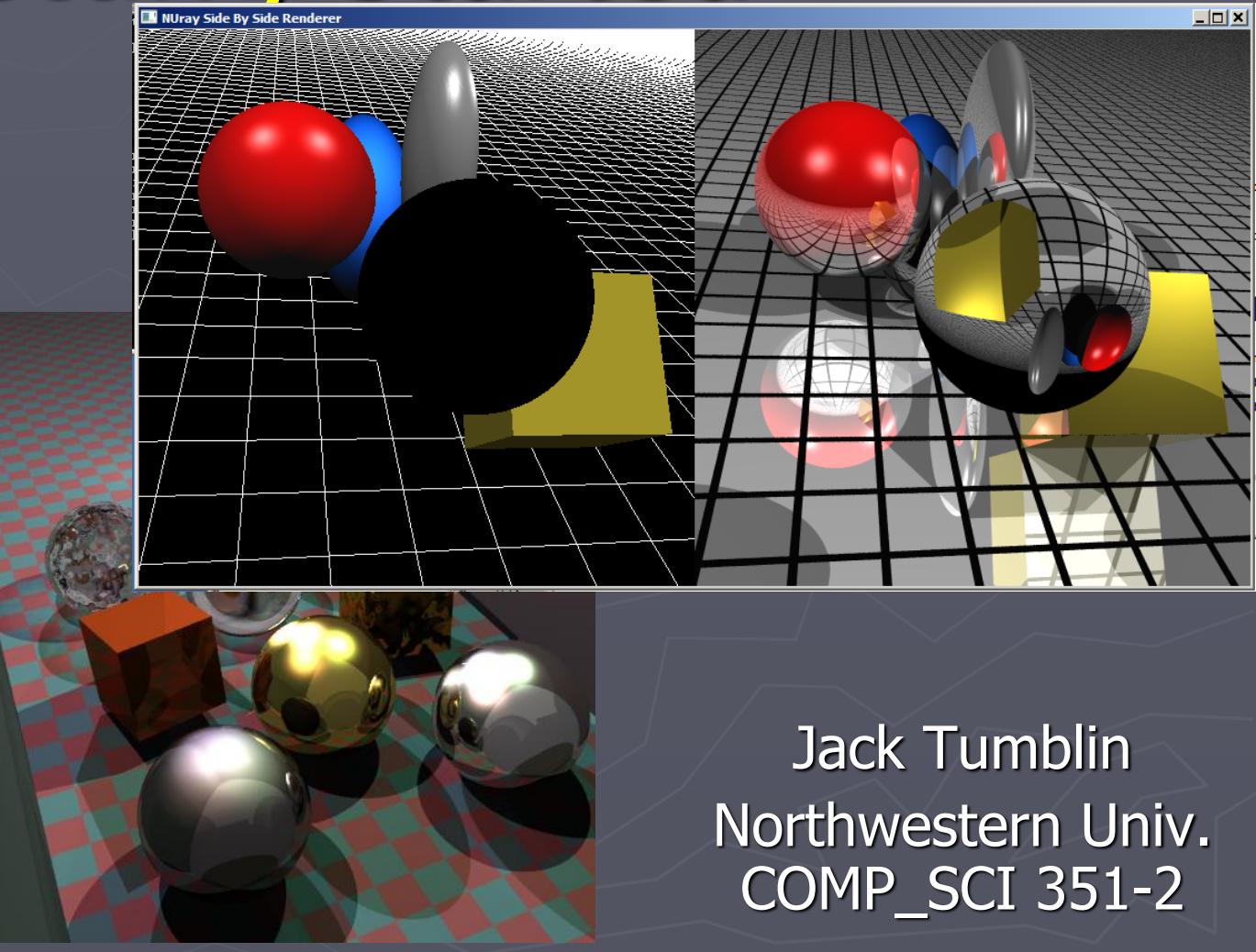


- **Interferometers?**
Michelson; Fabry-Perot; Fizeau; Mach-Zehnder; ...
<https://youtu.be/aDzMB27XVxQ?t=1m6s>

- **Schlieren Photography?**
Why do you think Schlieren photos are mostly forgotten?
What more-recent methods solve the same problem more easily?
https://youtu.be/mLp_rSBztel?t=31s How it works: <https://www.youtube.com/watch?v=OBe2YdJdRYo>



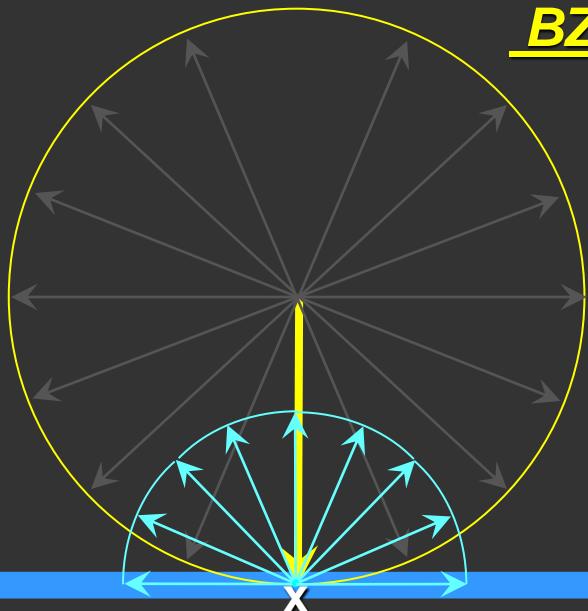
Ray Tracing B: Getting Started



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

The NIGHT-LIGHT PUZZLE: How much light does a ray carry?

- Pretty Blue Night-Light (an ‘electroluminescent plate’ lamp) has uniform appearance from all directions, lit or unlit
- Pick a **single point ‘x’** on the surface.
How much light power does that one infinitesimal point emit in each direction?
- It LOOKS uniform – **IS IT** uniform?



BZZT! -- STOP!

First define ‘light strength in each direction’!

RADIANCE L measures
doubly-differential power density:

$L == E / \text{steradian}$ (solid angle)

$L == (\text{Watts} / \text{meter}^2) / \text{steradian}$

EL Night-light surface ↘

The NIGHT-LIGHT PUZZLE: Radiance L has ? NONSENSICAL UNITS ?

RADIANCE L measures *doubly*-differential power density:

$$\boxed{\begin{aligned} L &== E / \text{steradian (solid angle)} \\ L &== (\text{Watts} / \text{meter}^2) / \text{steradian} \end{aligned}}$$

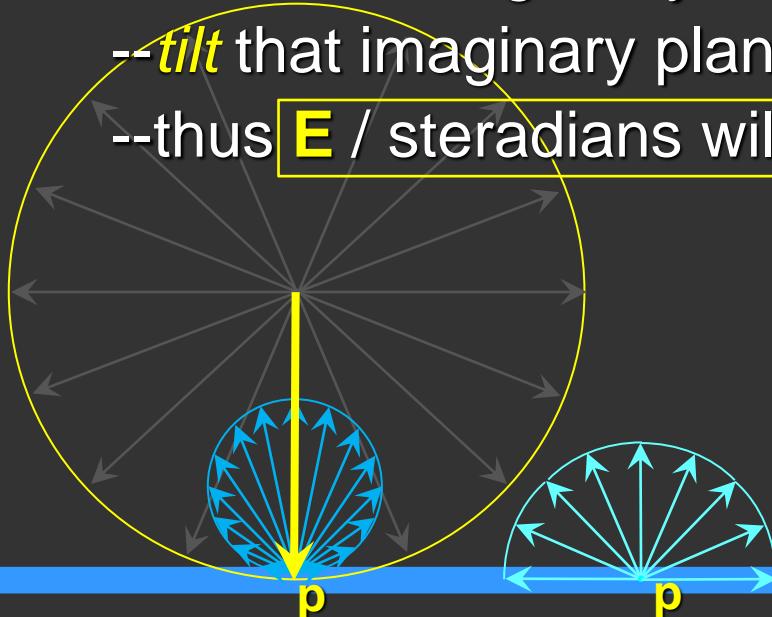
"?!? But that *includes* the cosine falloff in the E term?!?!"

.YES. "Huh? but, but, but, **THIS CAN'T be true!**"

--measure a light ray through an imaginary plane

--*tilt* that imaginary plane; it *changes* the irradiance E

--thus **E / steradians** will change when the light did not!?



.YES. (!!)

"?! What sets that plane ?!"

AHA! Our night-light surface!! ↗

Night Light Puzzle SOLUTION: WEIRDNESS IN RADIANCE UNITS L

- Radiance L is **NOT** defined by its units alone: **watts/m²·sr**;
- Its **WEIRD!** to compute Radiance L, you must also *divide by cos(θ_i)* to *remove* incidence-angle dependence:

formally:

$$\begin{aligned} \mathbf{L} &= \frac{\partial^2 \Phi}{\partial A \partial \Omega (\cos \theta_i)} == \mathbf{E} / (\text{solid angle} * \cos(\theta_i)) \\ &= \text{Watts} / (\text{meter}^2 * \text{steradian} * \cos(\theta_i)) \\ &\quad \text{where } \theta_i \text{ is the incidence angle at} \\ &\quad \text{the surface that defines the measured area } A \end{aligned}$$

- Radiance L at point x measures angular rate of E (or I), but
 - Radiance does not depend on surface orientation!
 - Radiance measured in one ray, in any direction, to/from any point, into any incident surface, or out of any emitting surface (e.g. blue night-light) *is constant and finite-valued (and NOT infinitesimal!)*

Night Light Puzzle SOLUTION: WEIRDNESS IN RADIANCE UNITS L

- Radiance L is NOT defined by its units alone: **watts/m²·sr**;
- Its **WEIRD!** to compute **Radiance L**, you must also divide by $\cos(\theta_i)$ to *remove* incidence angle dependence:

formally:

$$\begin{aligned} \mathbf{L} &= \frac{\partial^2 \Phi}{\partial A \partial \Omega (\cos \theta_i)} == \mathbf{E} / (\text{solid angle} * \boxed{\cos(\theta_i)}) \\ &= \text{Watts} / (\text{meter}^2 * \text{steradian} * \boxed{\cos(\theta_i)}) \\ &\quad \text{where } \theta_i \text{ is the incidence angle at} \\ &\quad \text{the surface that defines the measured area A} \end{aligned}$$

- Radiance **L** at point x measures angular rate of **E** (or **I**), but
 - Radiance does not depend on surface orientation!
 - Radiance measured in one ray, in any direction, to/from any point, into any incident surface, or out of any emitting surface (e.g. blue night-light) *is constant and finite-valued (and NOT infinitesimal!)*

Night Light Puzzle Answer: WEIRDNESS IN RADIANCE UNITS L

- Radiance L is **NOT** defined by its units alone: **watts/m²·sr**;
- Its **WEIRD!** to compute **Radiance L**, you must also *divide* by $\cos(\theta_i)$ to *remove* the incident-angle dependence:

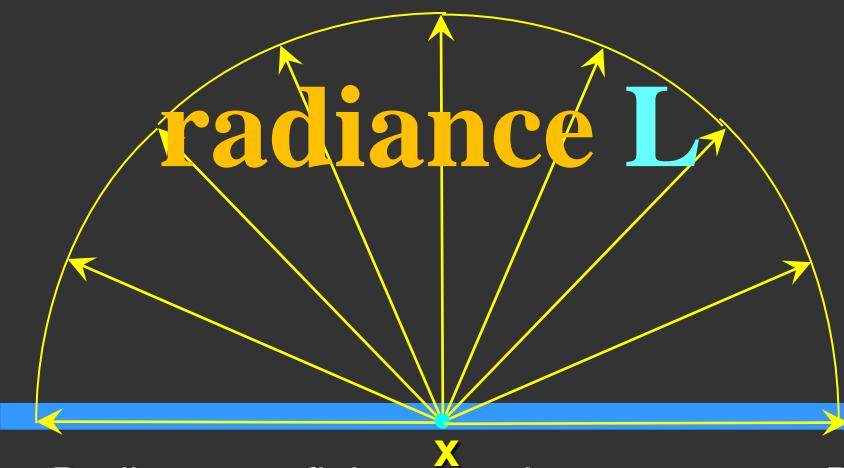
formally:

$$\begin{aligned} \mathbf{L} &= \frac{\partial^2 \Phi}{\partial A \partial \Omega (\cos \theta_i)} == \mathbf{E} / (\text{solid angle} * \boxed{\cos(\theta_i)}) \\ &= \text{Watts} / (\text{meter}^2 * \text{steradian} * \boxed{\cos(\theta_i)}) \\ &\quad \text{where } \theta_i \text{ is the incidence angle at} \\ &\quad \text{the surface that defines the measured area A} \end{aligned}$$

- Radiance **L** at point **x** measures angular rate of **E** (or **I**), yet
 - **Radiance** does not depend on surface orientation !!
 - **Radiance** measured in one ray, in any direction, to/from any point, into any incident surface, or out of any emitting surface (e.g. blue night-light) **is constant** and **finite-valued** (*and NOT infinitesimal!*)

Night Light Puzzle SOLUTION: WEIRDNESS IN RADIANCE UNITS L

- Eyes & Cameras give us estimates of Radiance **L**,
not the irradiance **E**, and *not* the radiant exitance **I**
- EXAMPLE: Night-Light Measurements
 - Total flux emitted at point x is infinitesimal, because its area is 0+, but
 - Radiant Exitance **I** (watts/m²) at point **x** is finite, but has $\cos \theta_i$ falloff
 - Radiance **L** at point **x** is finite, constant in all directions,
and follows the ‘brightness’ our eyes see



Radiance **L**: finite valued, constant.



Radiant Intensity (**I** or **E**): finite, angle-dependent

So What's Radiance?

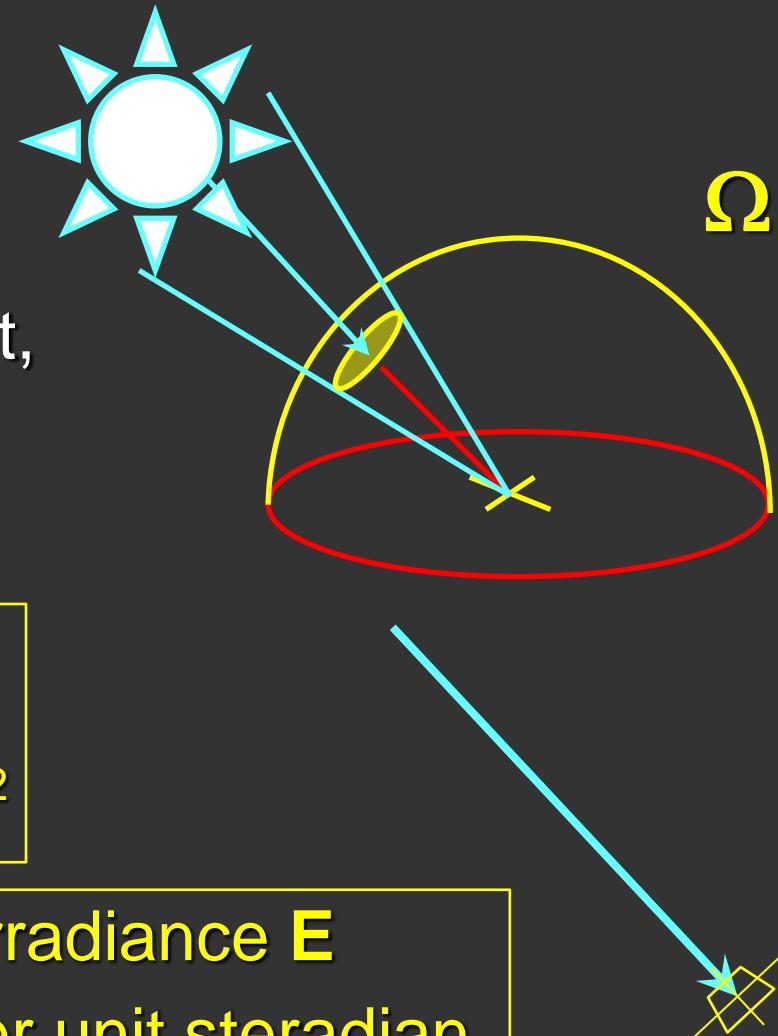
TL;DR:

- Radiance L :

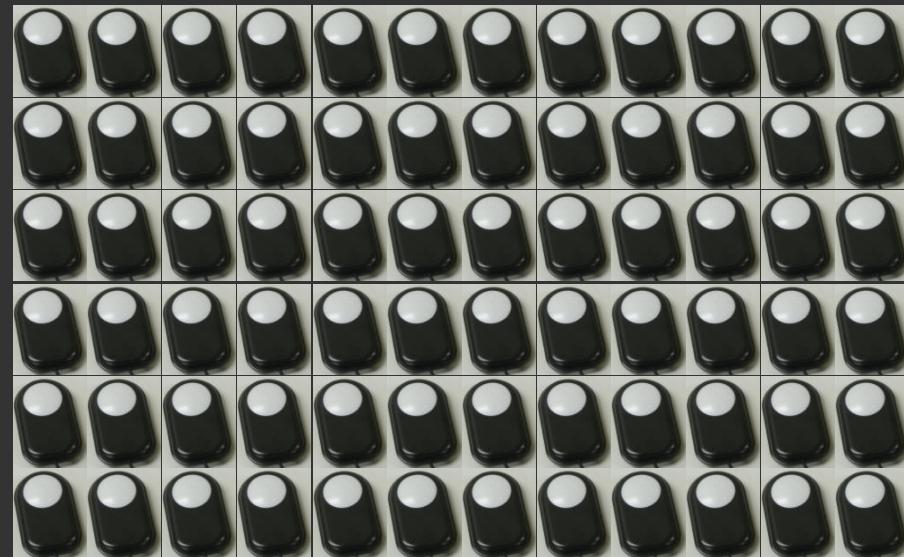
- ‘ray-strength’ in MKS,
- camera-pixels approximate it,
- Doubly-differential:
it measures TWO rates...

Watts per Steradian 
per **PERPENDICULAR** meter²

e.g. the **PERPENDICULAR** Irradiance E
per unit steradian



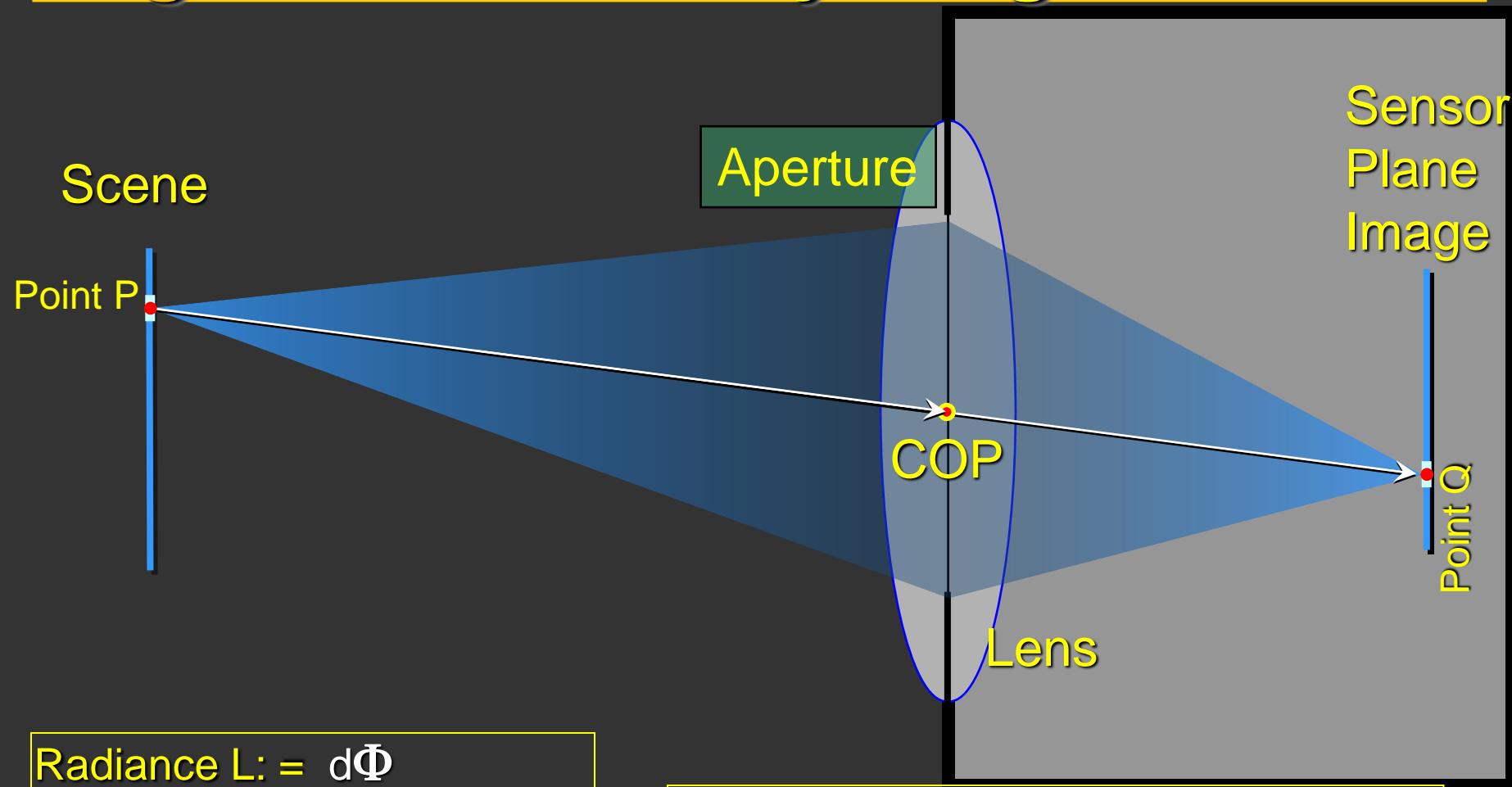
Camera Sensor: Array of Light Meters



Ideal Sensor Goals:

- *Instantaneous* Irradiance (E) measurement
- *Unlimited* resolution;
- *Unlimited* sensitivity,
- *Unlimited* Dynamic Range
- *Zero* noise visible

Individual Rays transport Radiance: Integrate ‘cone’ of rays to get Irradiance



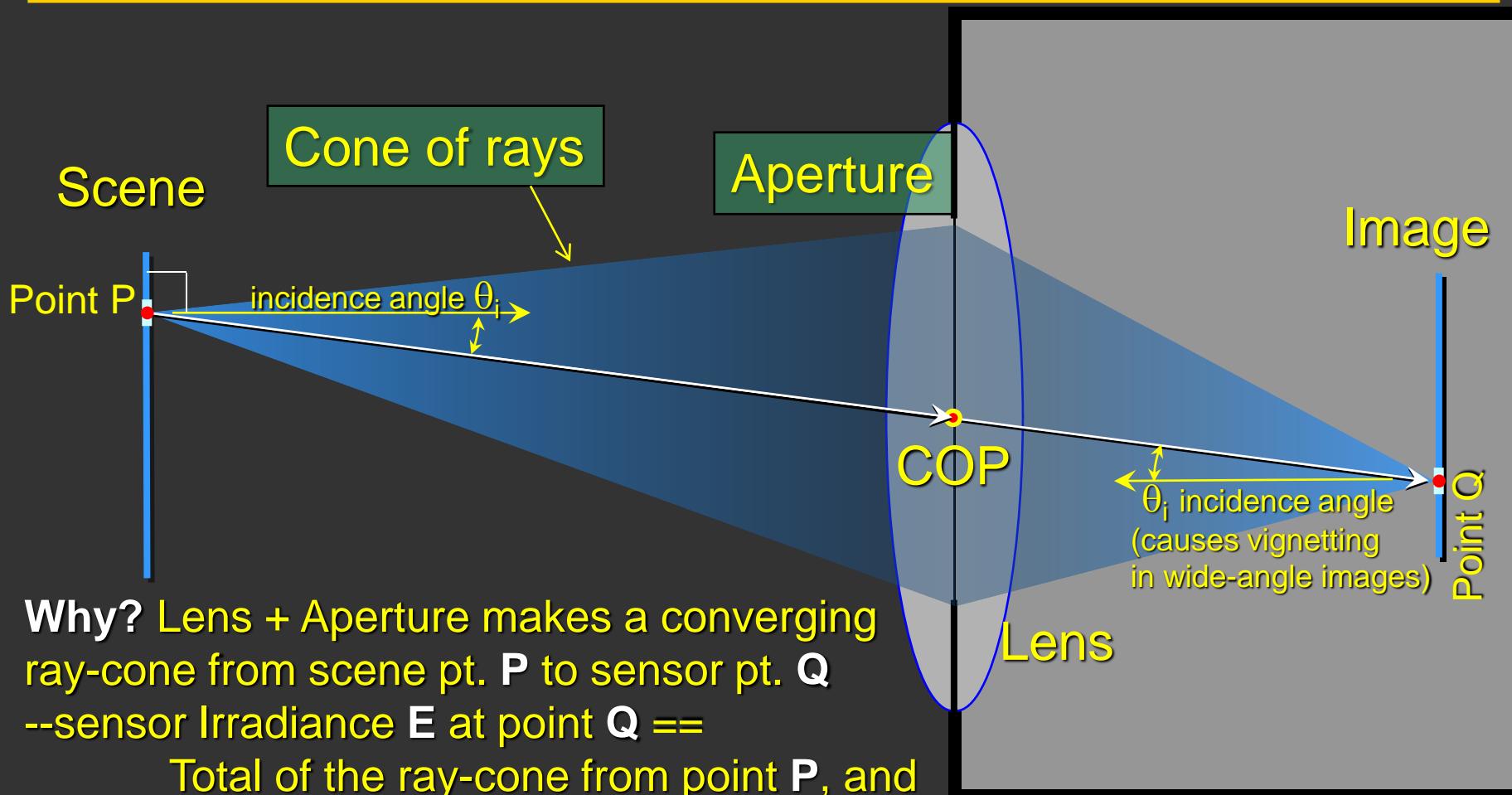
$$\text{Radiance } L := \frac{d\Phi}{dA d\omega \cos \theta_i}$$

$L = \text{watts} / (\text{meter}^2 \text{ steradian})$

$L = (\text{w/m}^2) / \text{sr} = (\text{w/sr})/\text{m}^2$...versus ... E or $I = \text{watts/meter}^2$

$$\text{Irradiance } E := \iint_0^{cone} \text{Radiance}(\theta, \varphi) d\theta d\varphi$$

Pixel Values are approx. proportional to Radiance



Why? Lens + Aperture makes a converging ray-cone from scene pt. **P** to sensor pt. **Q**

--sensor Irradiance **E** at point **Q** ==

Total of the ray-cone from point **P**, and

--if **P** is diffuse, radiance is constant for all rays in the cone, and

--if we shrink the aperture to a 'pinhole', ray-cone shrinks to a single ray.

Radiance **L**: d (watts / \perp meter 2) / steradian = (w/ \perp m 2) / sr

THUS

use Radiance **L**

to measure the amount of light
in each ray of your ray-tracer.

(Really? That's it? That's 'how much light in a ray'? Yes—that's it, that's all!)

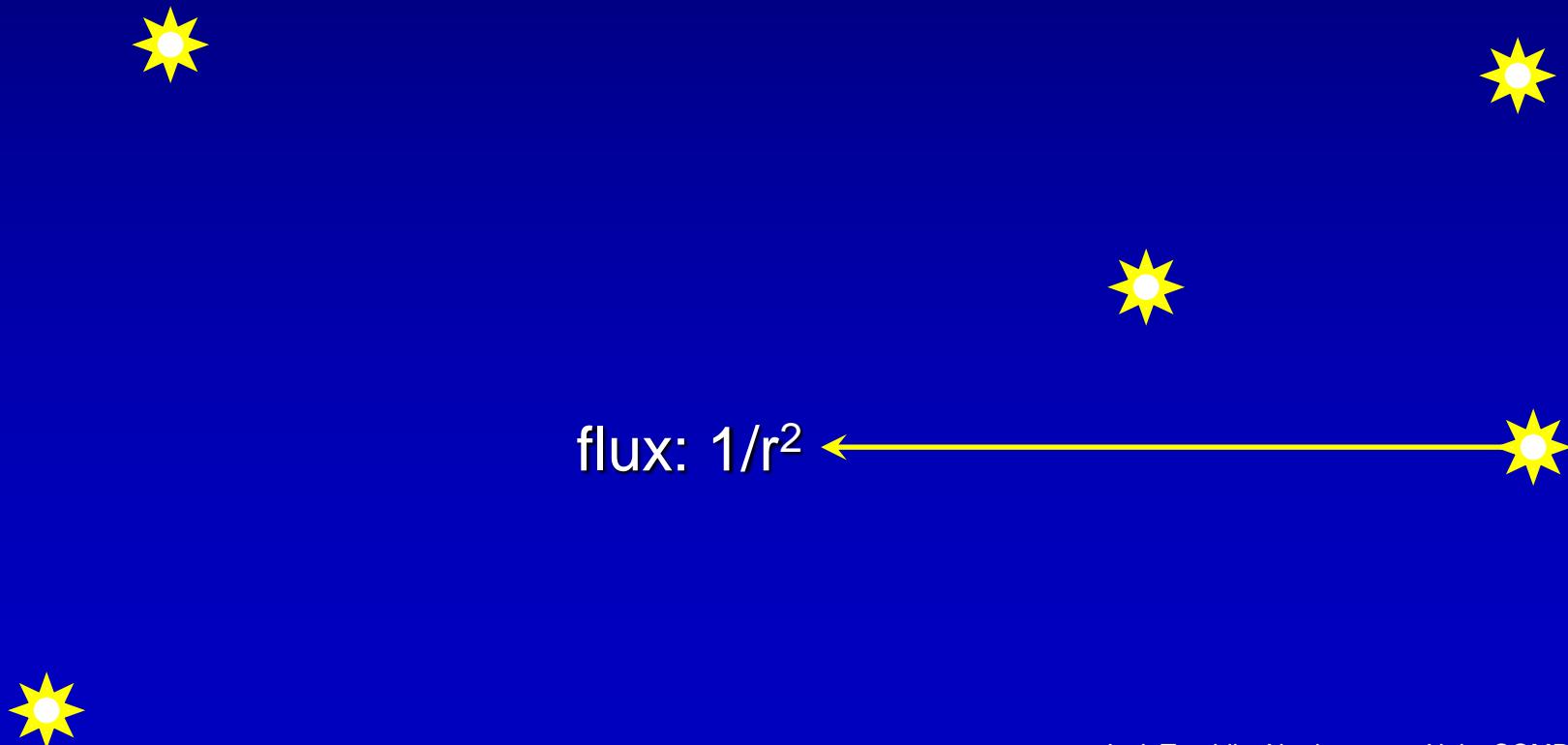
Any other practical uses for 'radiance'?

YES! MANY! ... for example:

Lighting Invariants

A SECOND (closely related) PUZZLE:

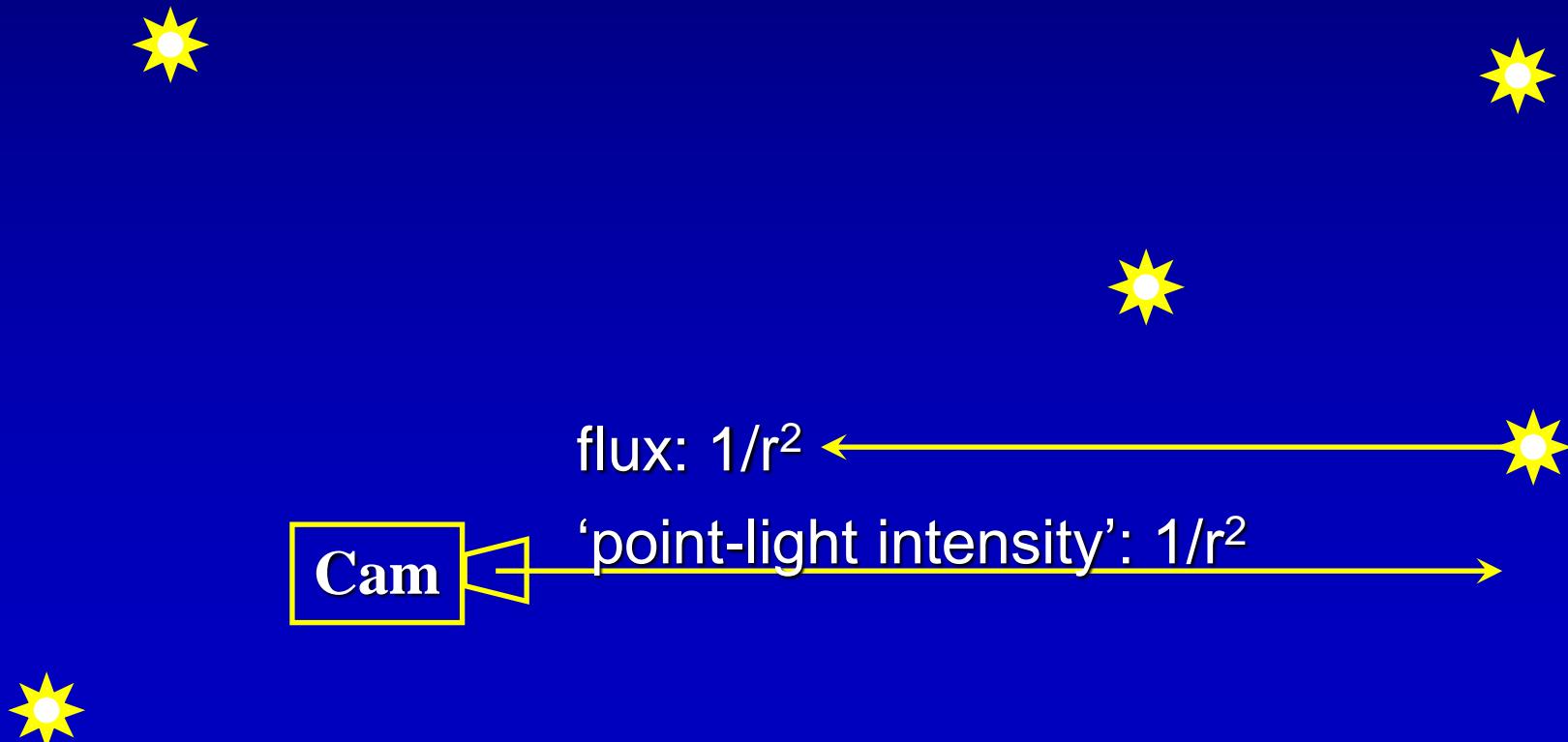
- We know point source flux drops with distance: $1/r^2$



Lighting Invariants

A SECOND (closely related) PUZZLE:

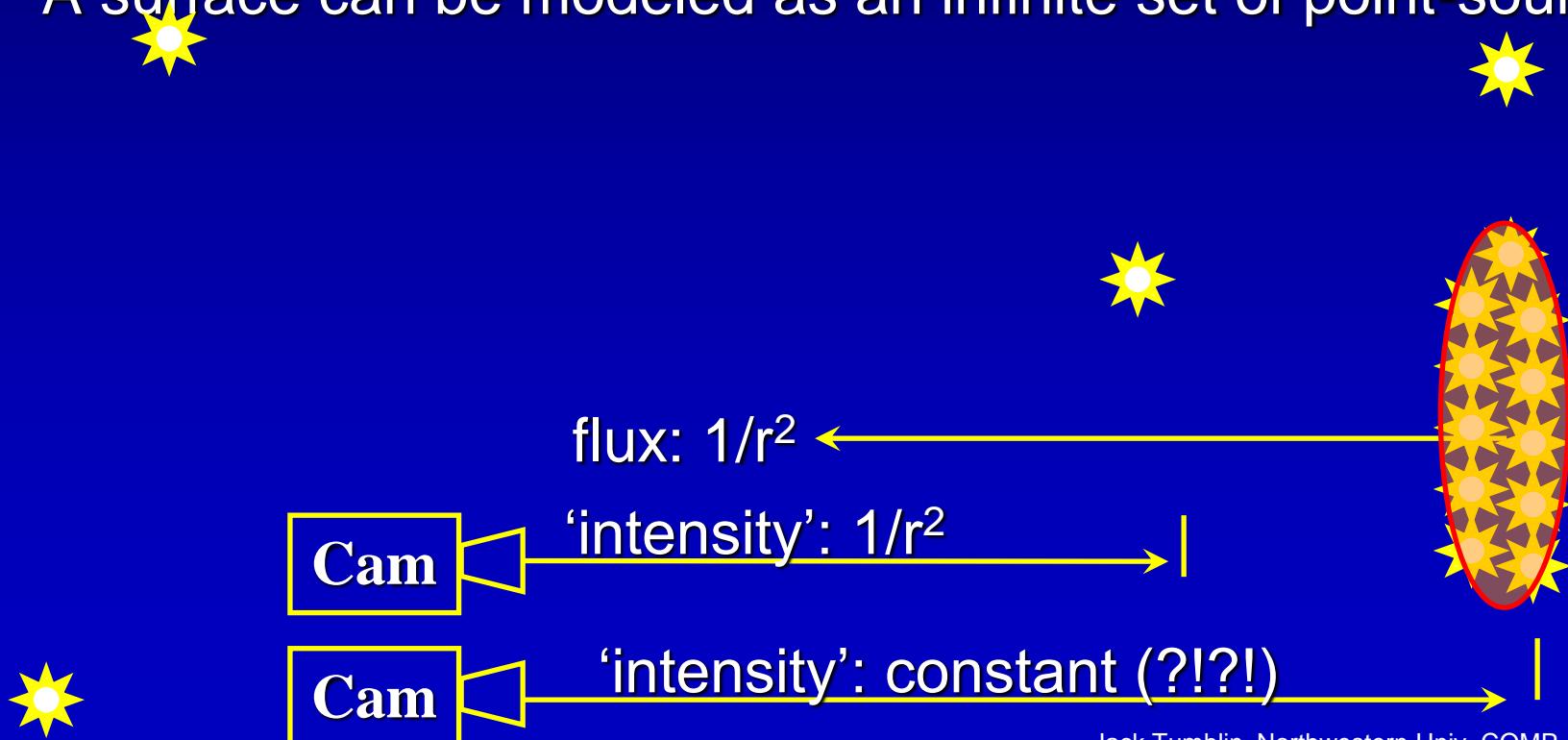
- We know point source flux drops with distance: $1/r^2$ *AND*
- Camera image of point source gets DIMMER by $1/r^2$



Lighting Invariants

A SECOND (closely related) PUZZLE:

- We know point source flux drops with distance: $1/r^2$ *AND*
- Camera image of point source gets DIMMER by $1/r^2$ *AND*
- A surface can be modeled as an infinite set of point-sources...



Lighting Invariants

A SECOND (closely related) PUZZLE:

- We know point source flux drops with distance: $1/r^2$ *AND*
- Camera image of point source gets DIMMER by $1/r^2$ *AND*
- A surface can be modeled as an infinite set of point-sources...

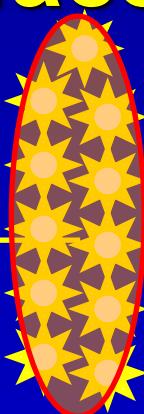


***SO*... Why doesn't the surface
get dimmer? ***

flux: $1/r^2$

Cam

'intensity': $1/r^2$



Cam

'intensity': constant (?!?!)

Lighting Invariants

A SECOND (closely related) PUZZLE:

- We know point source flux drops with distance: $1/r^2$ *AND*
- Camera image of point source gets DIMMER by $1/r^2$ *AND*
- A surface can be modeled as an infinite set of point-sources...

*So what's wrong here?

Why doesn't the surface
get dimmer?

flux: $1/r^2$

Cam

'intensity': $1/r^2$

Cam

'intensity': constant (?!?!)

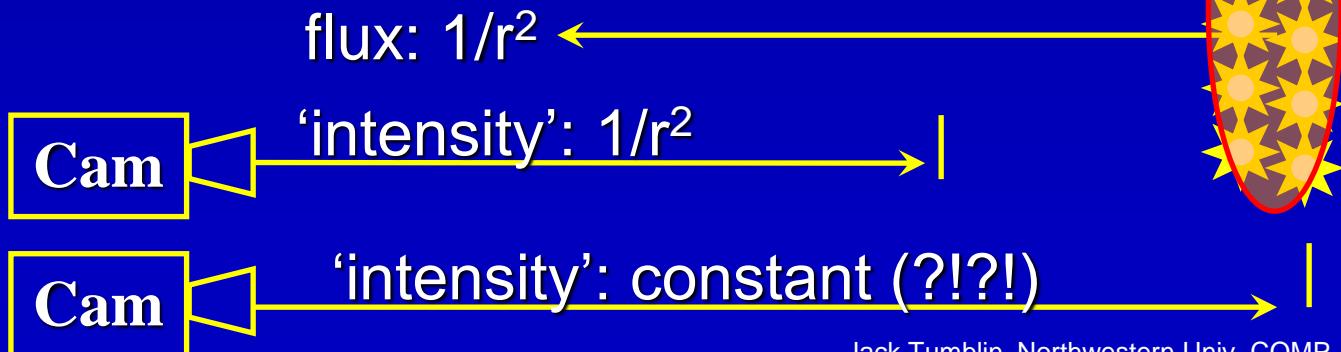


Lighting Invariants

PUZZLE: Why aren't far-away surfaces dimmer?

- We know point source flux drops with distance: $1/r^2$ *AND*
- Camera image of point source gets DIMMER by $1/r^2$ *AND*
- A surface can be modeled as an infinite set of point-sources...

HINT: HOW MANY *point sources* viewed per pixel?



Lighting Invariants

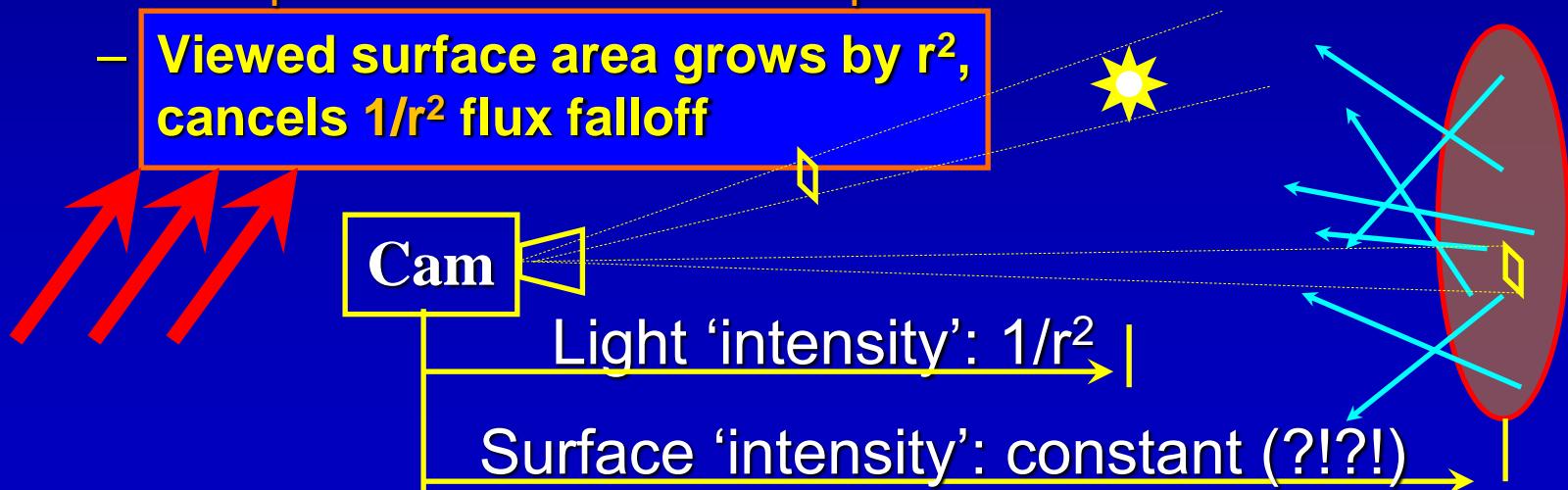
PUZZLE: Why aren't far-away surfaces dimmer?

Because camera pixels (& eyes) estimate **Radiance**, not flux!

- pixel value \approx flux * $\cos(\theta_i)$ / sr
- ‘good lens’ design: $\cos(\theta_i)$ term vanishes. Vignetting=residual error.
- Pixel’s size in steradians (sr) \approx fixed:

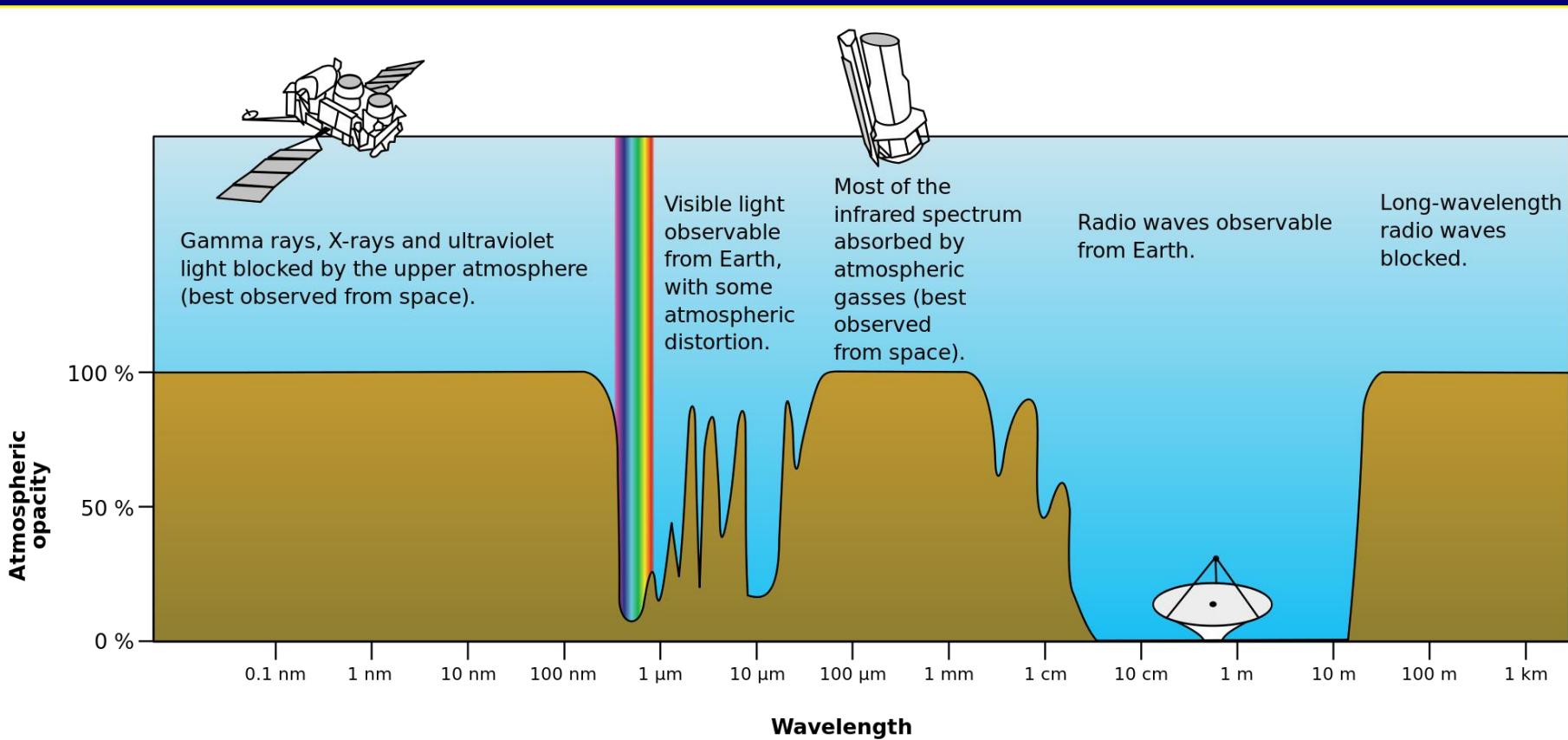
- One point source fits into one pixel: $1/r^2$ BUT L includes AREA;

Viewed surface area grows by r^2 ,
cancels $1/r^2$ flux falloff



Wavelength Dependence: Radiance R → Luminance L

- **Radiance R:** light ‘strength’ in a ray at **ALL** wavelengths.
- Earth’s atmosphere absorbs many of these wavelengths
? How do I LIMIT ‘radiance’ to visible-light only?

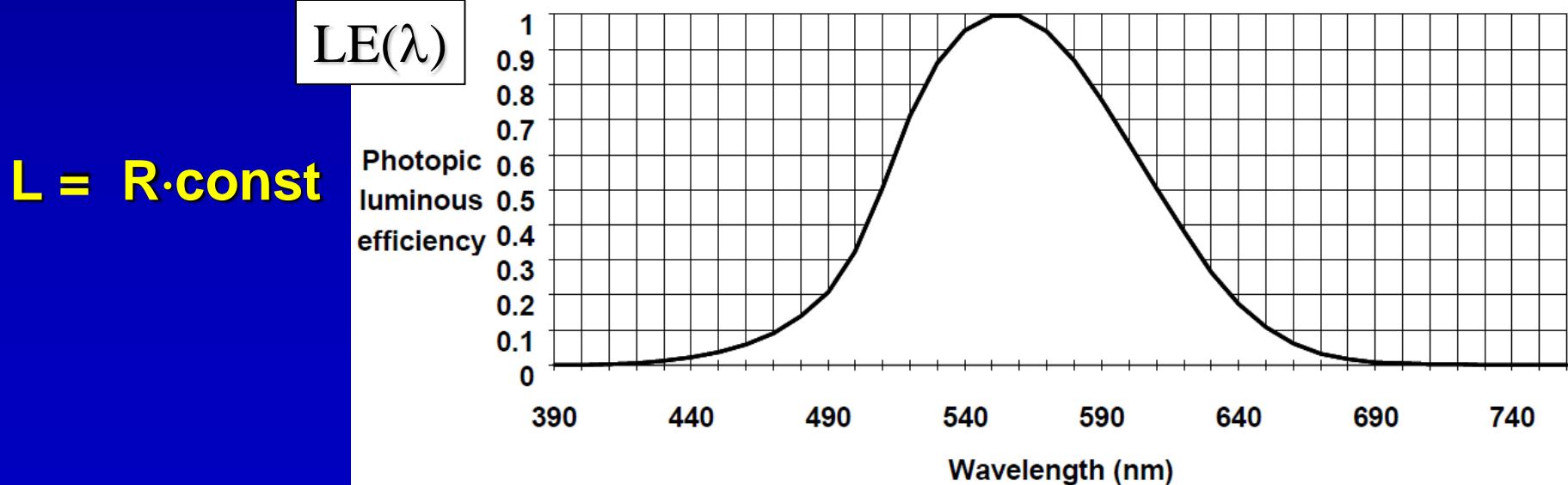


Wavelength Dependence: Radiance R → Luminance L

- **Radiance R:** sum of light ‘strength’ at **ALL** wavelengths.
(a ‘radiometric’ measure)
- **Luminance L:** weighted for **VISIBLE** wavelengths only.
(a ‘photometric’ measure)
- Conversion:

$$L = \int R(\lambda) LE(\lambda) d\lambda$$

Fixed spectrum? Simple to Convert -- a scale factor!



Wavelength Dependence: Radiance R → RGB Values

- **Radiance R:** light ‘strength’ at **ALL** wavelengths.
(radiometric measurement)
- **R,G,B:** a fixed spectrum for each given camera & display
→→ tied to radiance R by fixed scale factors

THUS: use **floating-point R,G,B values** for everything:
ray-tracer lights, materials, results images.

- Radiance **R** – to – RGB conversions? Weighted Spectra
Luminance **L** – to – RGB conversions? Weighted Spectra
(How? Scale $L(\lambda)$ by $R(\lambda)$, $G(\lambda)$, $B(\lambda)$ & integrate)

OPTIONAL READING: FS Hill Chap. 12 (canvas) and/or

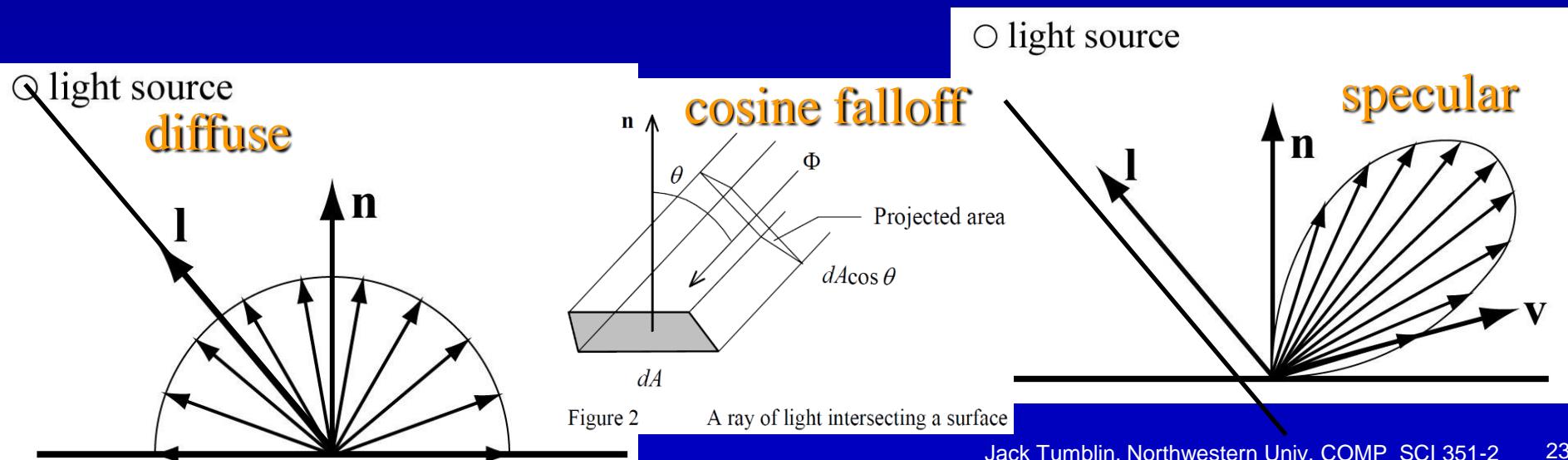
(Slides) <http://www.cs.cornell.edu/courses/cs465/2005fa/lectures/23a-murdoch.pdf>

(Reference book) <https://www.wiley.com/en-us/The+Reproduction+of+Colour%2C+6th+Edition-p-9780470024256>

Reflection: Phong Model

SAME as before (from COMP_SCI 351-1)

- Emissive (glow: light made the the surface)
- Ambient (the lighting inside shadows) +
- Diffuse (irradiance * albedo) +
- Specular (mirror-like highlights)

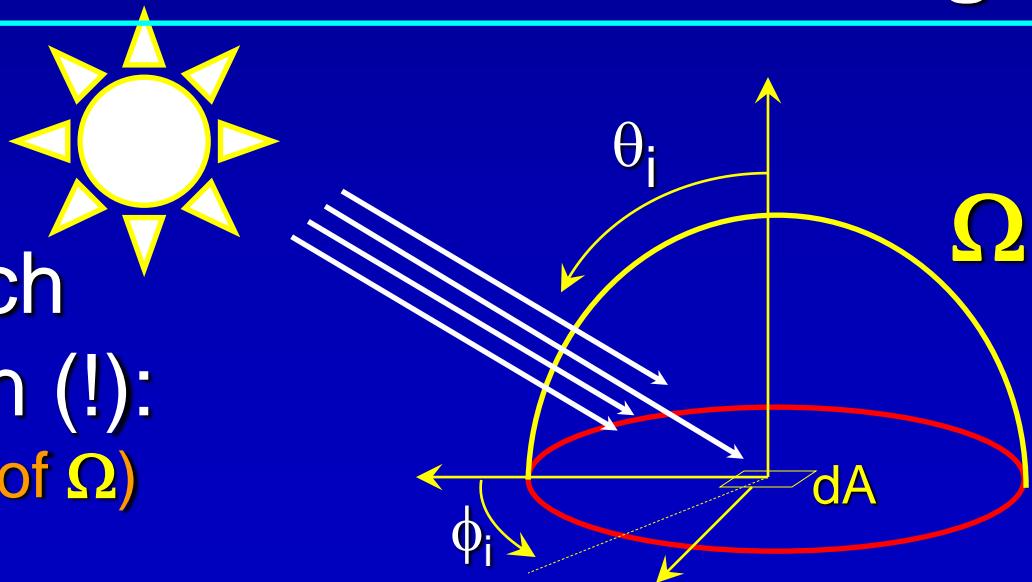


Point-wise Light Reflection

- Given:
 - Infinitesimal surface patch dA ,
 - illuminated by irradiance amount E
 - from *just one* direction (θ_i, ϕ_i)

How should we measure the returned light?

- Ans: by emitted RADIANCE measured for each outgoing direction (!):
(measured on surface of Ω)



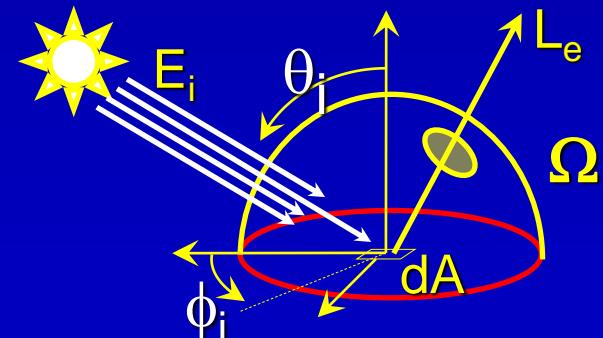
Point-wise Light Reflection: BRDF

Bidirectional Reflectance Distribution Function

$$F_r(\theta_i, \phi_i, \theta_e, \phi_e) = L_e(\theta_e, \phi_e) / E_i(\theta_i, \phi_i)$$

- Still a ratio of (outgoing/incoming) light, but
- BRDF: Ratio of outgoing RADIANCE in one direction: $L_e(\theta_e, \phi_e)$ that results from incoming IRRADIANCE in one direction: $E_i(\theta_i, \phi_i)$
- Units are tricky:

$$\text{BRDF} = F_r = L_e / E_i$$



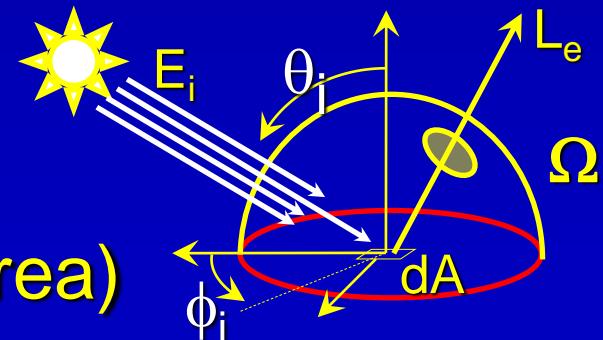
Point-wise Light Reflection: BRDF

Bidirectional Reflectance Distribution Function

$$F_r(\theta_i, \phi_i, \theta_e, \phi_e) = L_e(\theta_e, \phi_e) / E_i(\theta_i, \phi_i)$$

- Still a ratio (outgoing/incoming) light, but
- BRDF: Ratio of outgoing RADIANCE in one direction: $L_e(\theta_e, \phi_e)$ that results from incoming IRRADIANCE in one direction: $E_i(\theta_i, \phi_i)$
- **Units are tricky, as before:**

$$\text{BRDF} = F_r = L_e / E_i = \\ (\text{Watts}/(\cos(\theta_i) * \text{area}/\text{sr})) / (\text{Watts}/\text{area})$$



Point-wise Light Reflection: BRDF

Bidirectional Reflectance Distribution Function

$$F_r(\theta_i, \phi_i, \theta_e, \phi_e) = L_e(\theta_e, \phi_e) / E_i(\theta_i, \phi_i)$$

- Still a ratio (outgoing/incoming) light, but
- BRDF: Ratio of outgoing RADIANCE in one direction: $L_e(\theta_e, \phi_e)$ that results from incoming IRRADIANCE in one direction: $E_i(\theta_i, \phi_i)$
- **Units are tricky:** “reflectance per unit steradian”

$$\text{BRDF} = F_r = L_e / E_i = (\text{Watts/area/sr}) / (\text{Watts/area}) = 1/\text{sr}$$

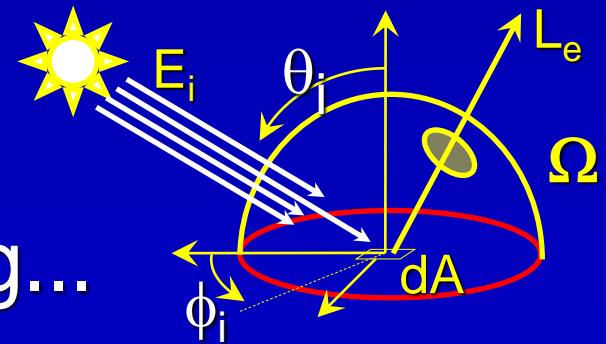
(and don't forget the built-in $1/\cos(\theta_i)$ in L_e)

Point-wise Light Reflection: BRDF

Bidirectional Reflectance Distribution Function

$$F_r(\theta_i, \phi_i, \theta_e, \phi_e) = L_e(\theta_e, \phi_e) / E_i(\theta_i, \phi_i), \text{ and (1/sr) units}$$

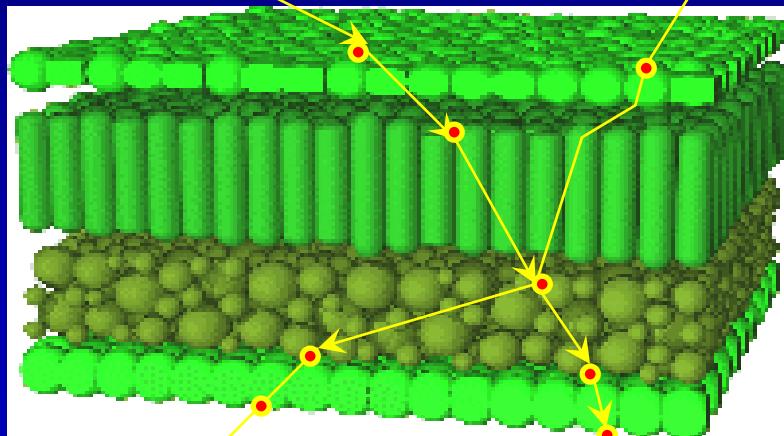
- ‘**Bidirectional**’ because value is SAME if we swap the in, out directions: $(\theta_e, \phi_e) \leftrightarrow (\theta_i, \phi_i)$
Important Property! aka ‘Helmholtz Reciprocity’
- Truly opaque surfaces: microscopic surface shape explains ‘BRDF’ well;
- Still only an approximation:
ignores subsurface scattering...



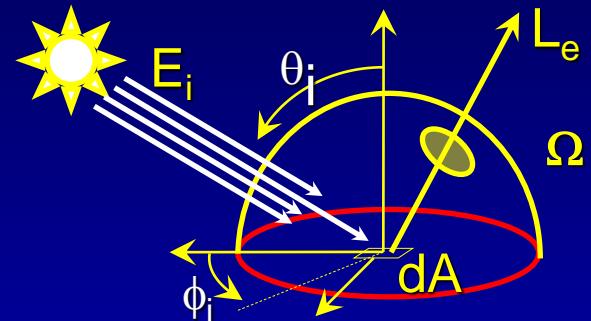
Scattering Difficulties:

single-point BRDFs aren't enough for all surfaces:

Example:
Leaf Structure



Depends on total area
of cell wall interfaces



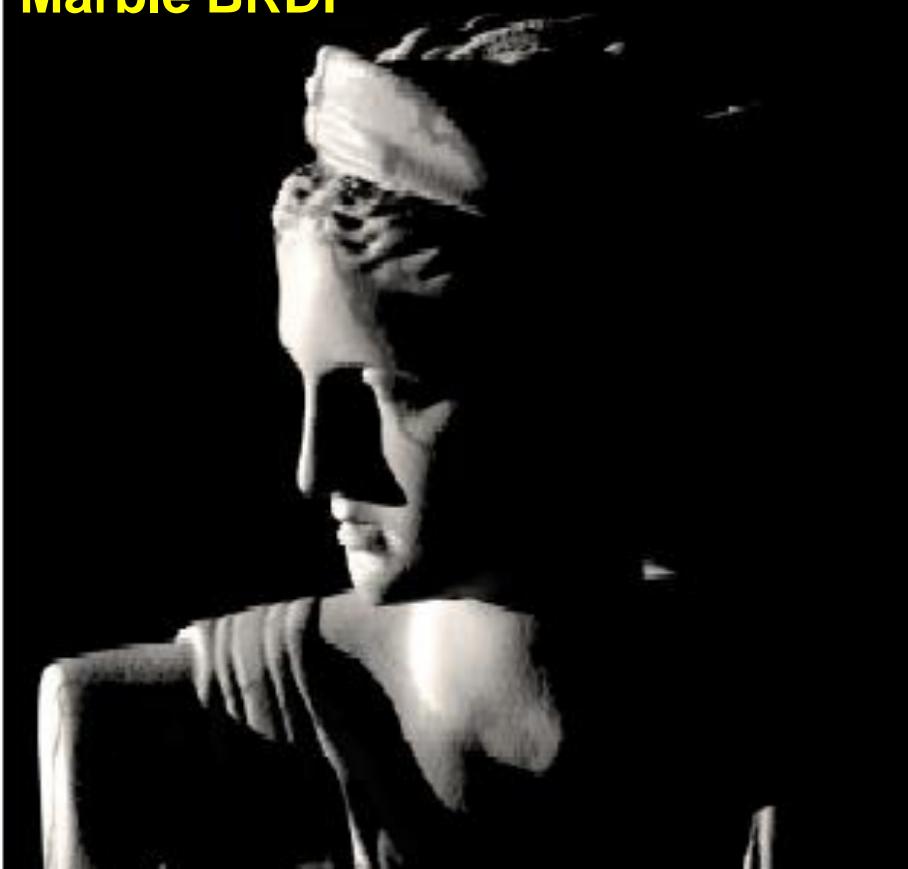
Angles Depend on
refractive index,
scattering, cell wall
structures, etc.

Subsurface Scattering Models

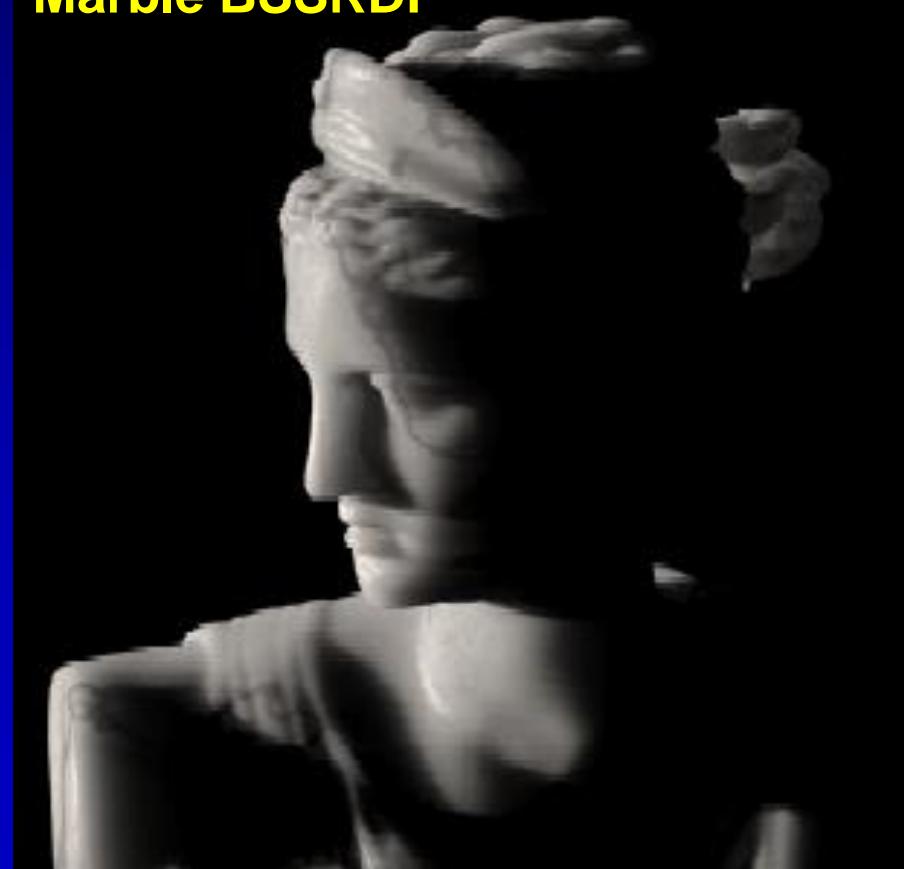
Classical: Kubelka-Monk (1930s, for paint; many proprietary variants),
CG approach: Hanrahan & Krueger(1990s)

Still State-of-the-art: ‘dipole model’ (2001, Jensen)

Marble BRDF



Marble BSSRDF



Subsurface Scattering Models

Classical: Kubelka-Monk (1930s, for paint; many proprietary variants),
CG approach: Hanrahan & Krueger(1990s)

More Recent: ‘dipole model’ (2001, Jensen)

Skin BRDF (measured)



Skin BSSRDF (approximated)



BSSRDF Model

Approximates scattering result as embedded point sources below a BRDF surface:

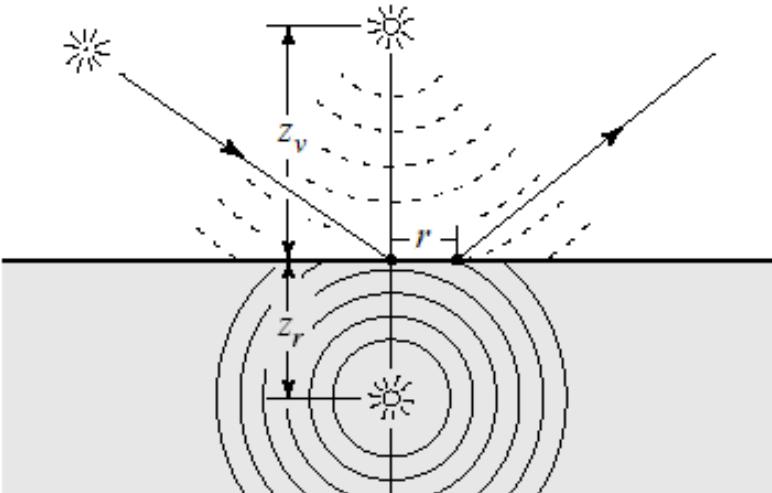


Figure 3: An incoming ray is transformed into a dipole source for the diffusion approximation.

Pixar Renderman Documentation::

https://renderman.pixar.com/resources/RenderMan_20/subsurface.html



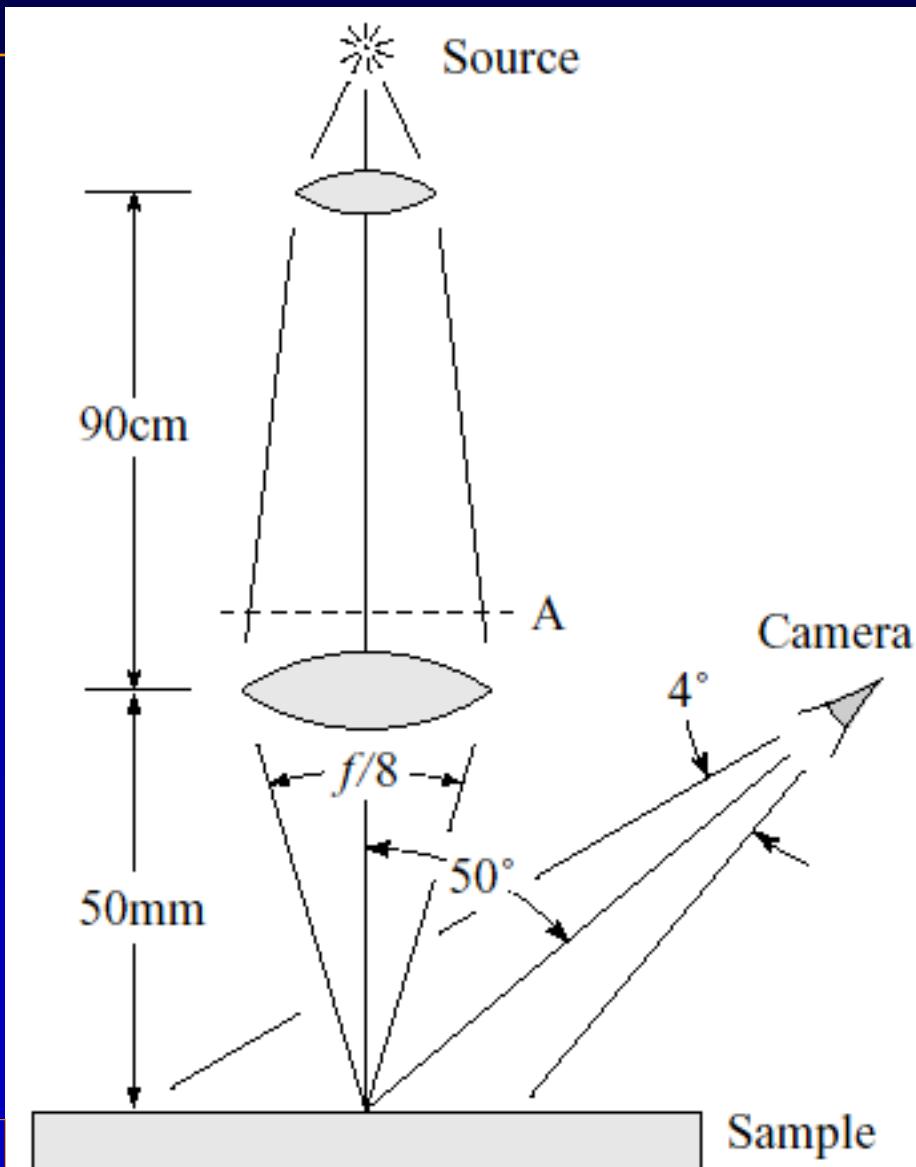
BSSRDF: “A Practical Model for Subsurface Light Transport”

Henrik Wann Jensen, Steve Marschner, Marc Levoy, Pat Hanrahan,
SIGGRAPH’01 (<http://graphics.ucsd.edu/~henrik/images/subsurf.html>)

BSSRDF Model

- Embedded point sources below a BRDF surface
- Ray-based, tested, Physically-Measurable Model, tested/verified
- Now widely used and/or approximated in movie sfx and animation (Shrek's skin, etc)

Wann Jensen et al., 2001



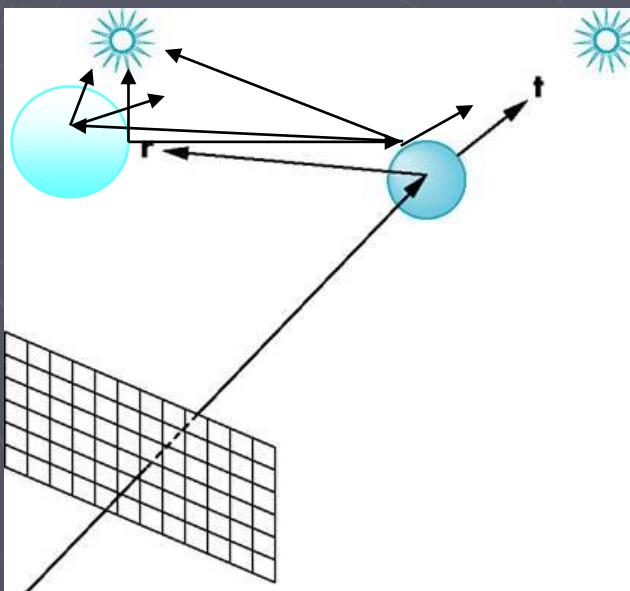
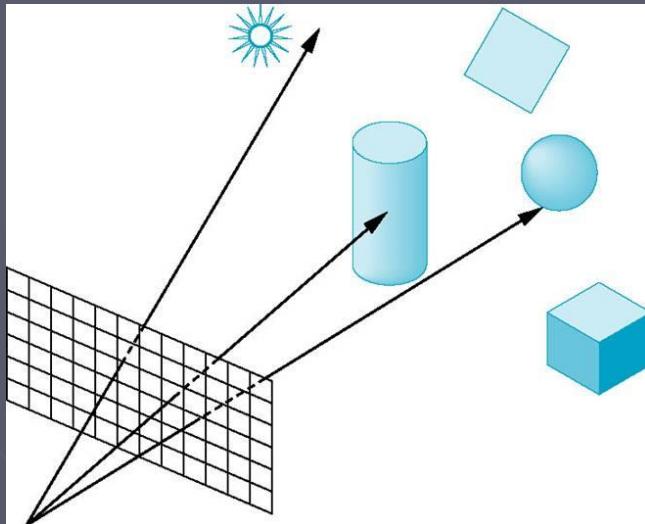
STOP !! Enough, Already !!

- ▶ Now I know 'radiance' measures light in rays,
- ▶ & Now I know to use floats for RGB values
- ▶ & Now I know 'reflectance' isn't so simple, yet ray-strength ratios describe it precisely.

But I **still** don't know how to make a picture!

?! How do I make pictures from rays !?

Ray-Tracing Overview:



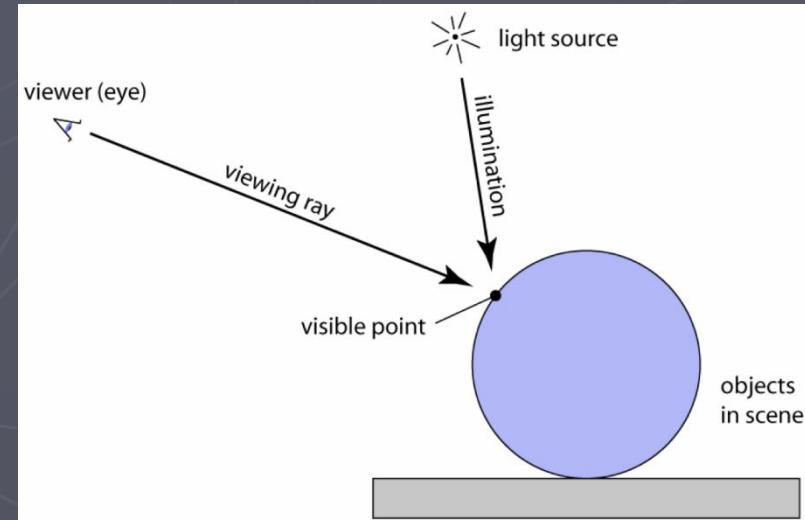
- **Ray Casting** first:
 - 'cast' rays of light from the camera to the scene, and
 - copy colors to pixels
- Then **Recursive Ray Tracing**:
 - Extension to Ray Casting:
 - Recursively cast rays from the points of intersection for shadows, reflections, transparency, etc.
 - Combine the light amounts

What's A 'Ray'?

- ▶ Simple! A 3D line parameterized by 't':
$$\text{ray}(t) = (\text{origin_point}) + (\text{direction_vector}) \cdot t$$



- ▶ Trace 'eye' ray outwards thru camera pixel to 'hit-point'
- ▶ Trace 'shadow' ray from hit-point to light source



From E. Angel: "Interactive Computer Graphics"

Concise Ray-Tracer Outline

FS Hill "Computer Graphics Using OpenGL" 2nd Edition, Chap 14, pg 736, Fig 14.4:

define the objects and light sources in the scene

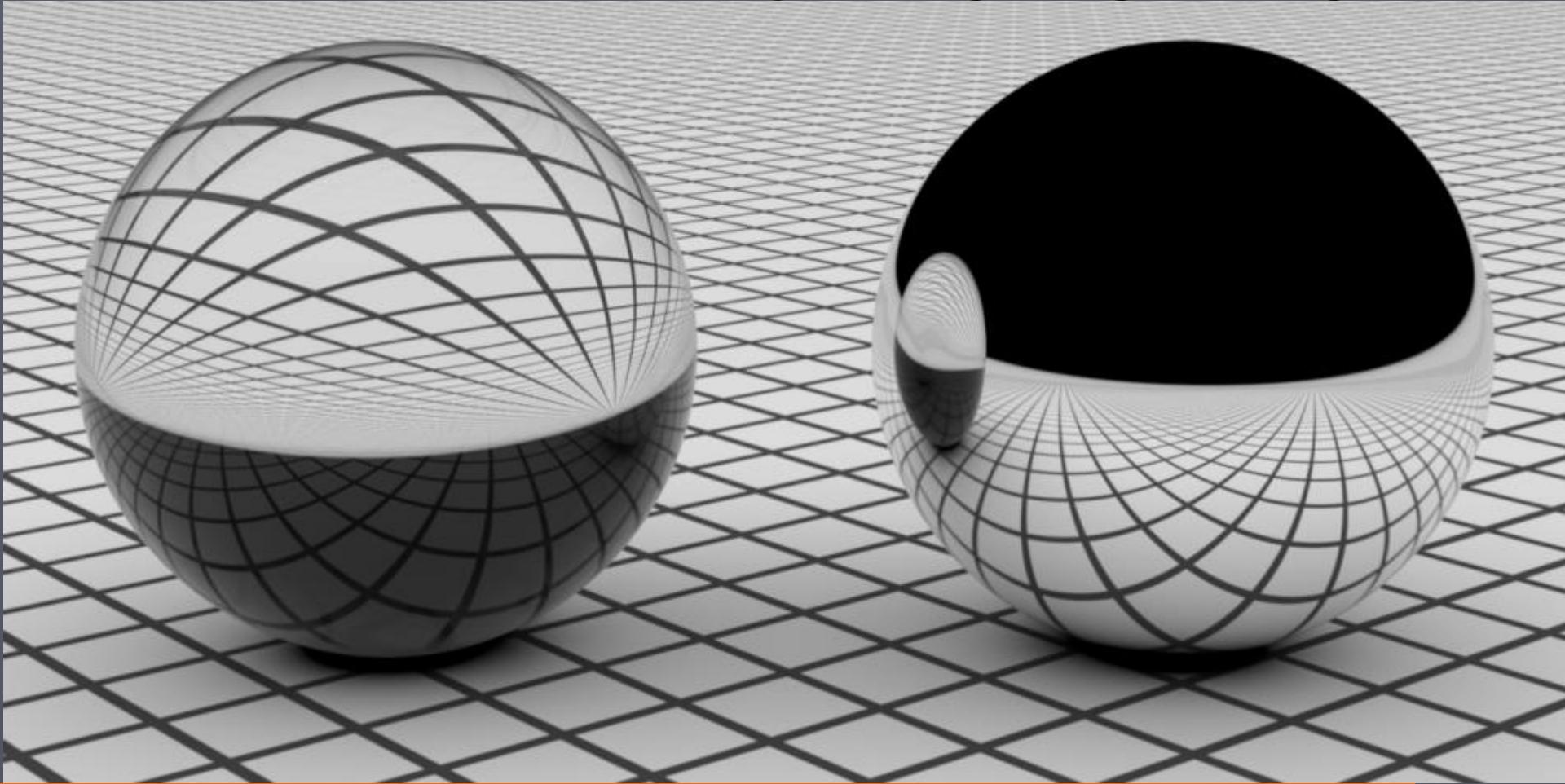
set up the camera

```
for(int r = 0; r < nRows; r++)
    for(int c = 0; c < nCols; c++)
    {
```

- 1. Build the rc-th ray*
- 2. Find all intersections of the rc-th ray with objects in the scene*
- 3. Identify the intersection that lies closest to, and in front of, the eye*
- 4. Compute the "hit point" where the ray hits this object, and the normal vector at that point*
- 5. Find the color of the light returning to the eye along the ray from the point of intersection*
- 6. Place the color in the rc-th pixel.*

```
}
```

Let's Build this, Step-by-Step:



Matt Pharr, Greg Humphries: Physically-Based Rendering Techniques (**PBRT**)
highly recommended beautiful in-depth book <https://www.pbrt.org/gallery.html>

Week 1 Goal: Camera + grid

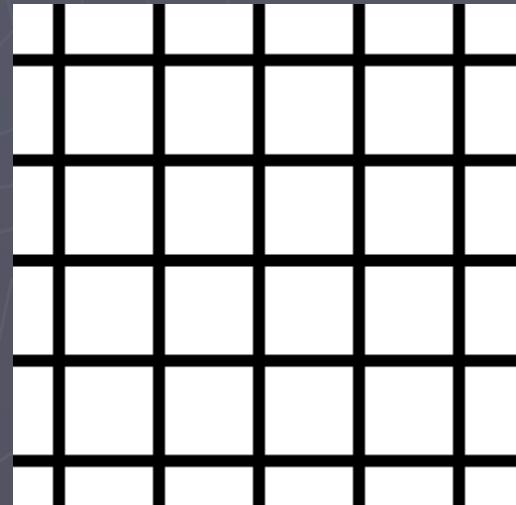
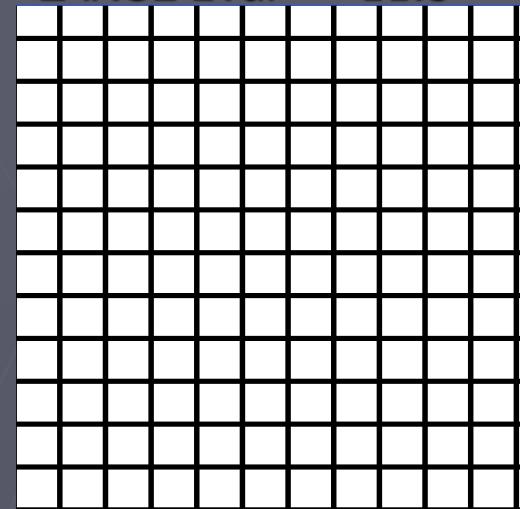
Fixed Camera, with Perspective, at 3D Origin
looks down on a ground-plane grid in (x,y)

Trace just ONE object -- a 'line-grid'
at $z = -5$

Allow just ONE interactive user adjustment:
move line-grid closer or farther away

You know how to do this in WebGL:
How do we do it in ray-tracing?

LARGE zVal = -11.5



Small zVal = -4.5

Our Big Plan:

Write code for:

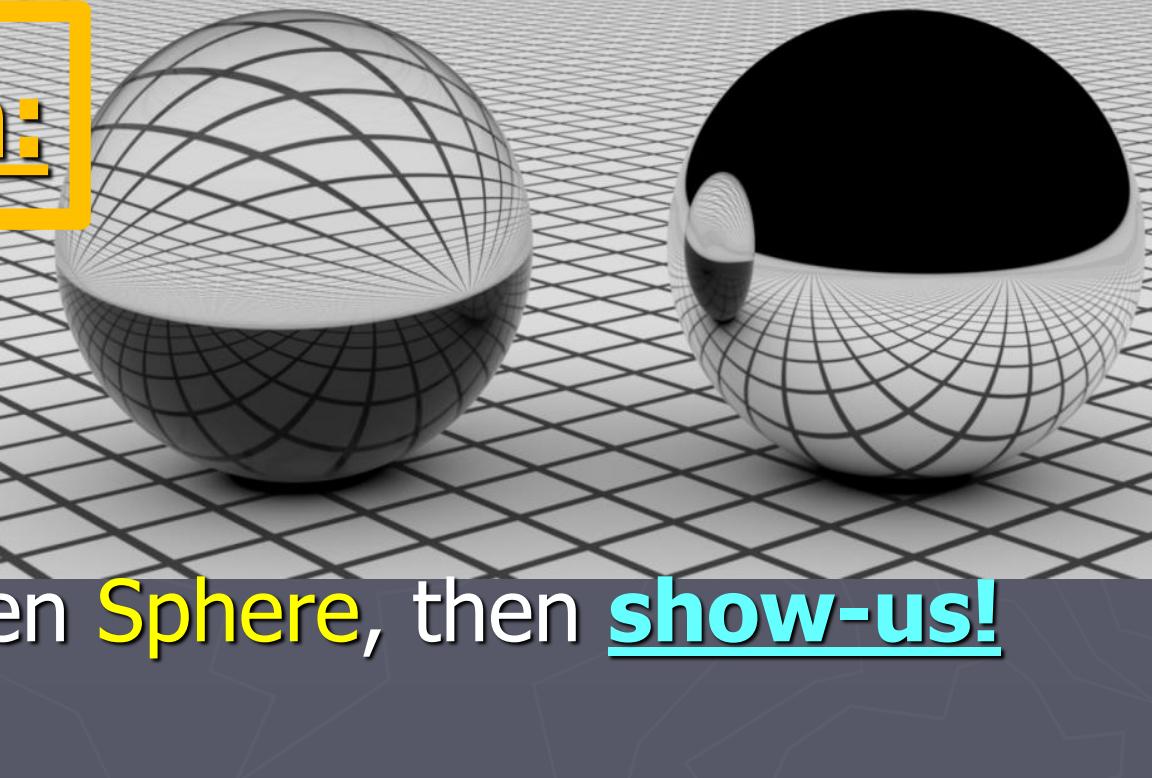
- ▶ Camera, then
- ▶ Gnd Plane, then
- ▶ Antialiasing, then Sphere, then **show-us!**
- ▶ Shadow, then
- ▶ Reflection, then
- ▶ Transparency, then

DEMO DAY

Further Step-by-Step Refinements:

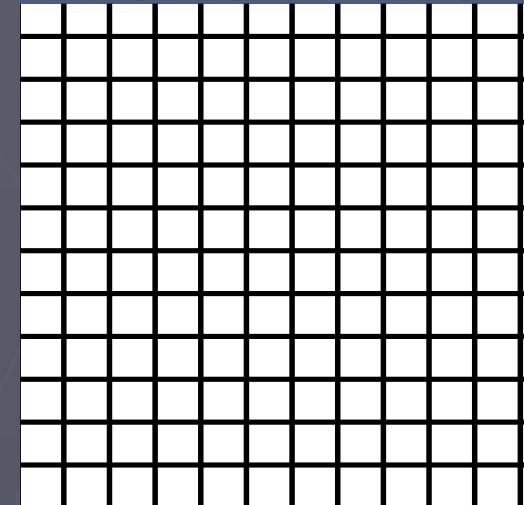
Lighting → Materials → Texture → Envir. Maps...

***** MAKE PICTURES AT EVERY STEP *****



Week 1 Goal: Raytrace a LineGrid with adjustable zVal

LARGE zGrid = -11.5



'CRay' object:

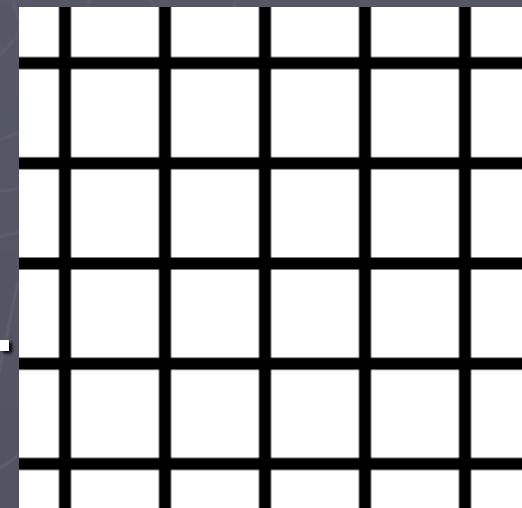
- one ray: orig point, dir vec (x,y,z,w)

'CCamera' object:

- a simple camera, fixed at world origin
- generates 'eye' rays into scene

'CGeom' object:

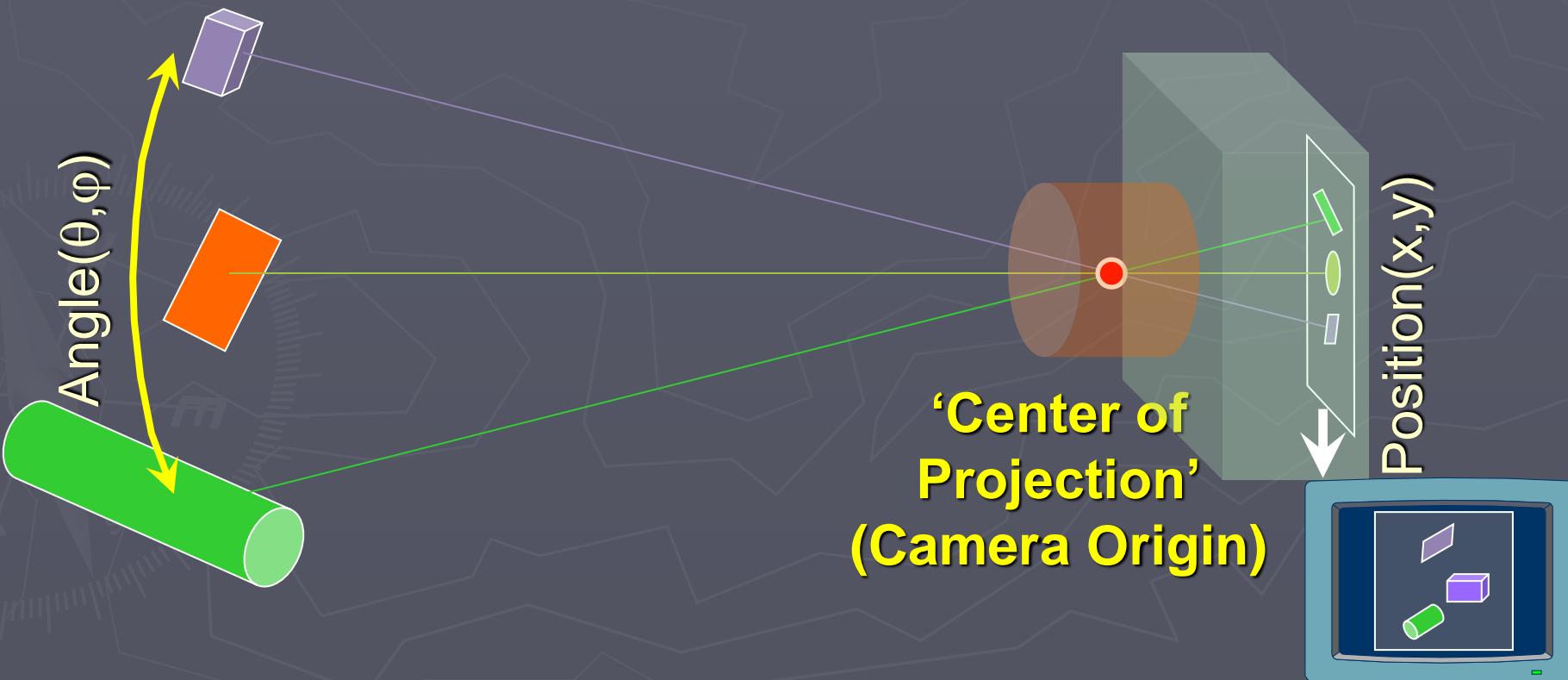
- a simple x,y grid-plane at zGrid
- Make zGrid user-adjustable...
- Contains `traceGrid(inRay);` funcn.
- to find ray/gridplane intersection
 - ▶ Return `lineColor` if we hit a line
 - ▶ Return `gapColor` if we hit a gap



Small zGrid = -4.5

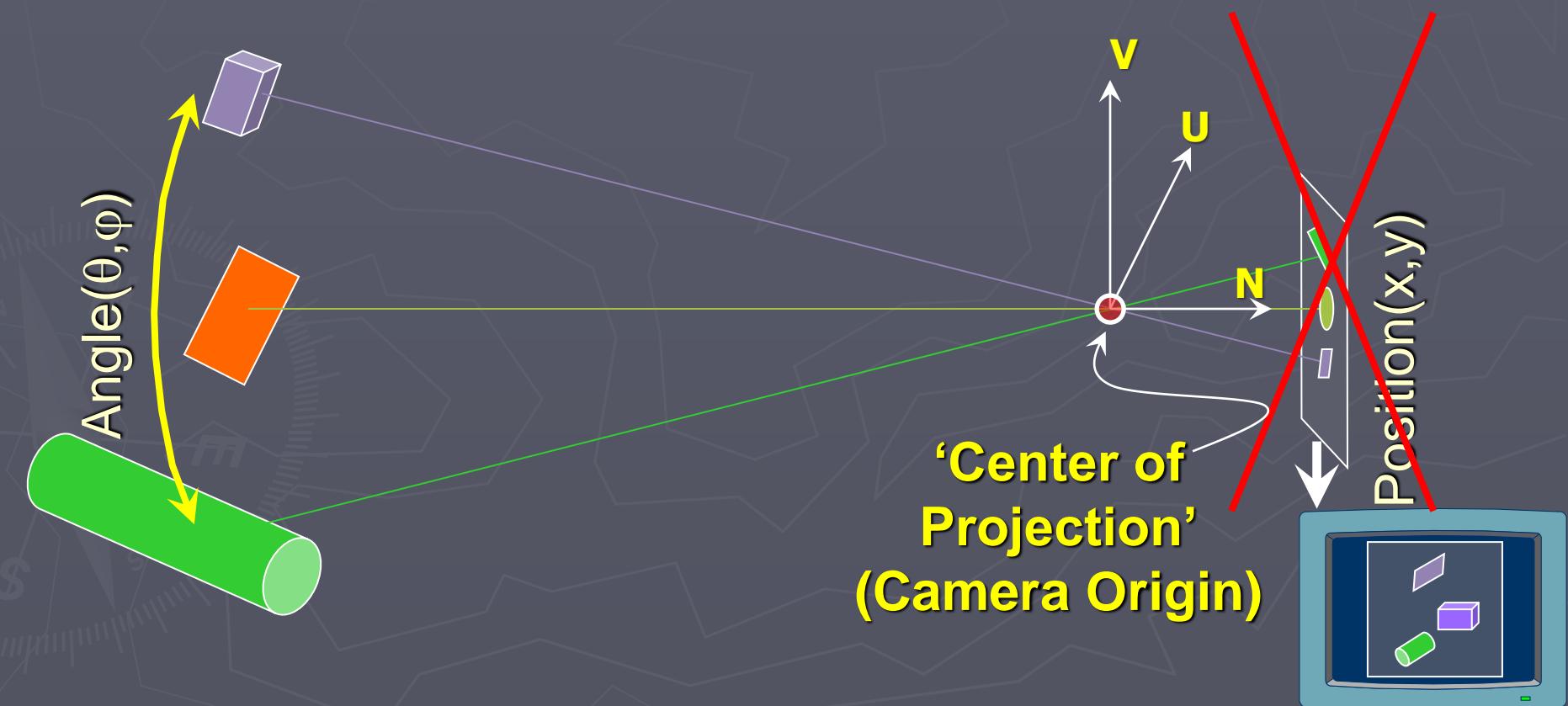
What defines a 'camera' ?

- ▶ Center of Projection + 'image plane' (splits the universe)
 - all camera rays pass through plane:
 - COP is 'origin' of the camera coordinate system, & 'origin' of rays.
- ▶ Angle in 3D Scene \leftrightarrow position on image plane



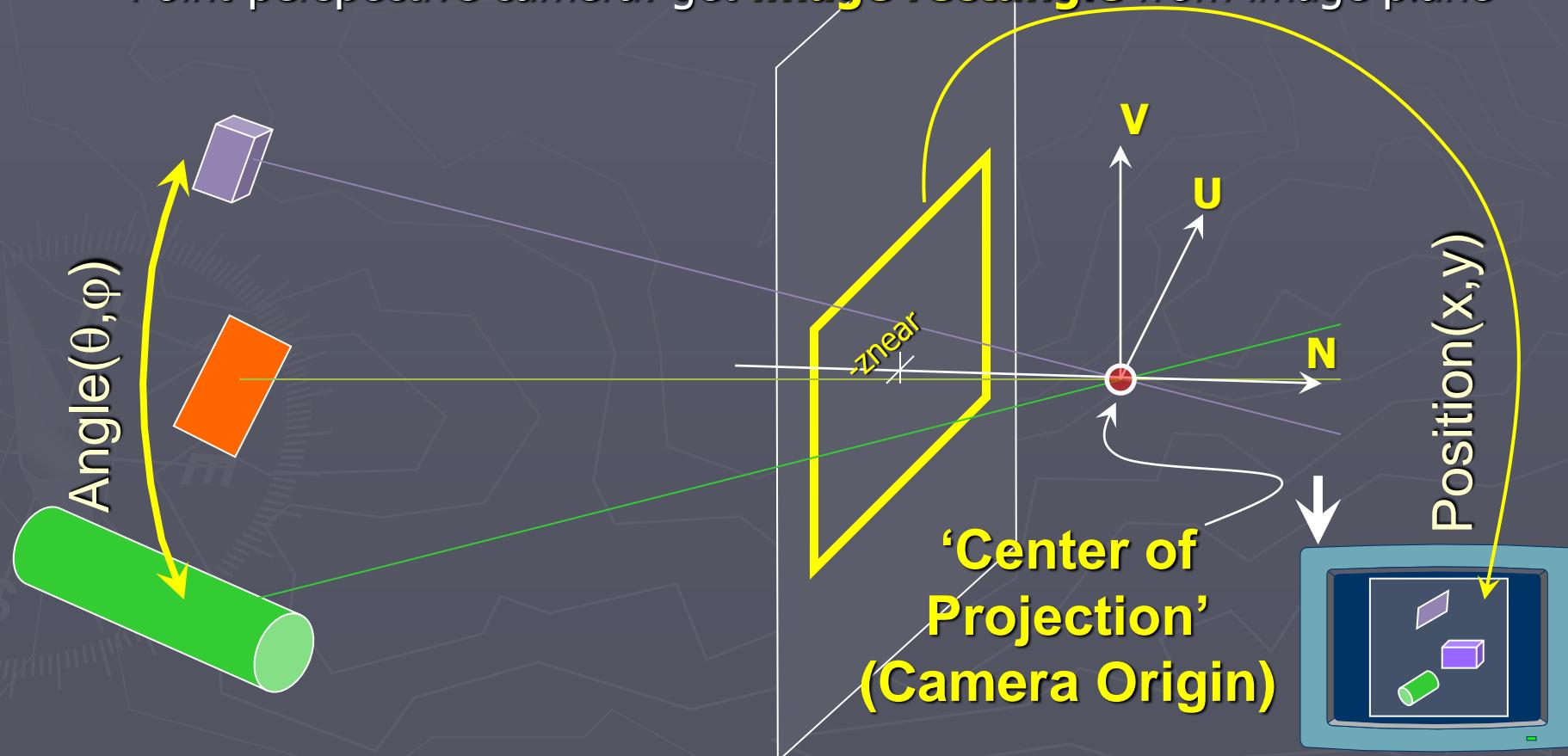
What defines a 'ray-tracing Camera' ?

- Center of Projection: == Origin of camera coords **U,V,N**
always use right-handed coord. systems, just like WebGL,
THUS we always look down the **-z** direction == **-N** axis



1) Copy WebGL 'frustum()' Camera:

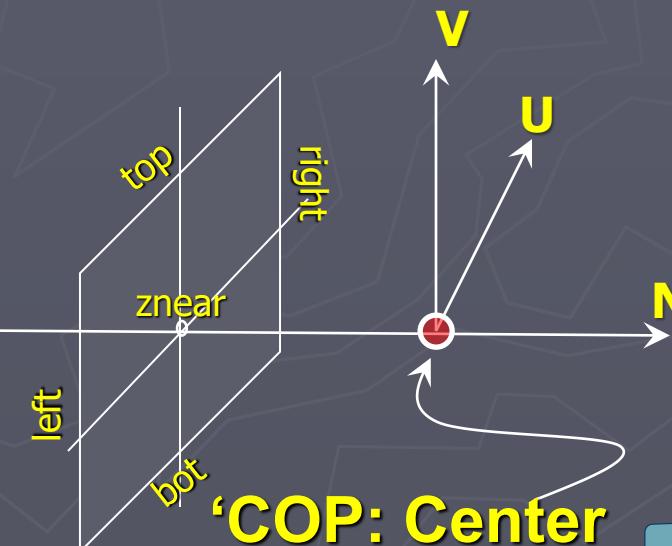
- ▶ **Image plane** for any camera will 'cut the universe in half'
 - simple u,v plane at $n = -z_{near}$, (same as WebGL)
 - rays from COP copy scene colors onto image plane
 - Point-perspective camera: get **image rectangle** from image plane



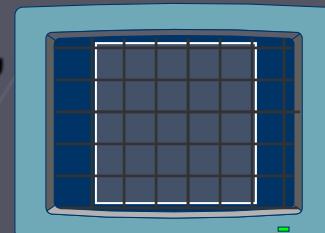
1) Copy WebGL 'frustum()' Camera:

- ▶ World-Space 'camera coordinate axes' U,V,N
- ▶ Camera Image Rectangle:
(HINT: mimic our WebGL
`setFrustum()` by
writing a ray-camera version)

U coords: $\text{left} \leq u < \text{right}$
V coords: $\text{bot} \leq v < \text{top}$
N coords: $n = -z_{\text{near}}$



'COP: Center of Projection'
COP == Camera Origin
COP == Ray Origin



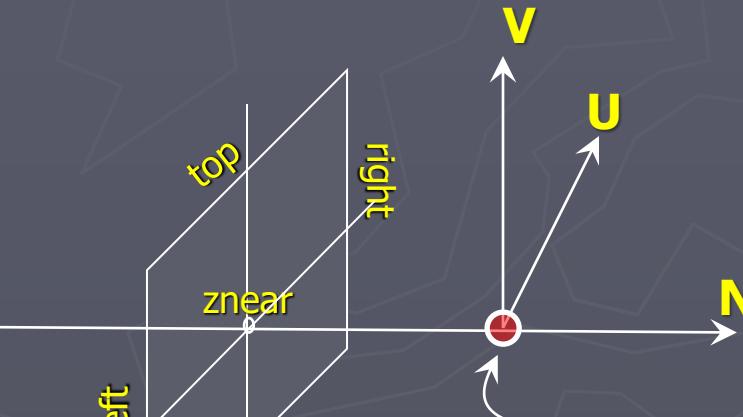
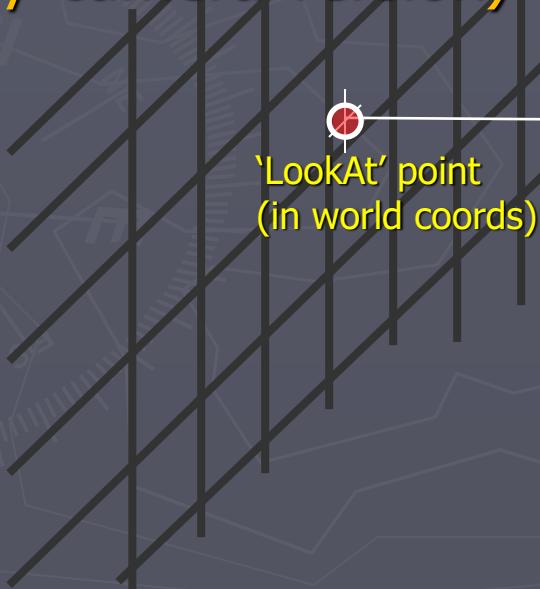
2) Copy WebGL Camera Aiming:

Specify eye rays from camera in ****WORLD**** coordinates!

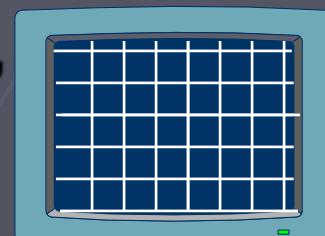
A) Find **camera origin (COP)** and the **U, V, N** vectors

- **COP** point == camera origin == **View_Reference_Point** or '**VRP**'
- **N_vector** == (**COP** - '**lookAt**'_point), normalized
- **U_vector** == (**ViewUpVector** *CROSS* **N_vector**), normalized
- **V_vector** == (**N_vector** *CROSS* **U_vector**), normalized

B) HINT: mimic WebGL's
setLookAt() by writing
a ray-camera version)



'COP: Center of Projection'
COP == Camera Origin
COP == Ray Origin



How Does Camera make an 'Eye' Ray?

C) Find image rectangle's lower-left corner in world coord system:

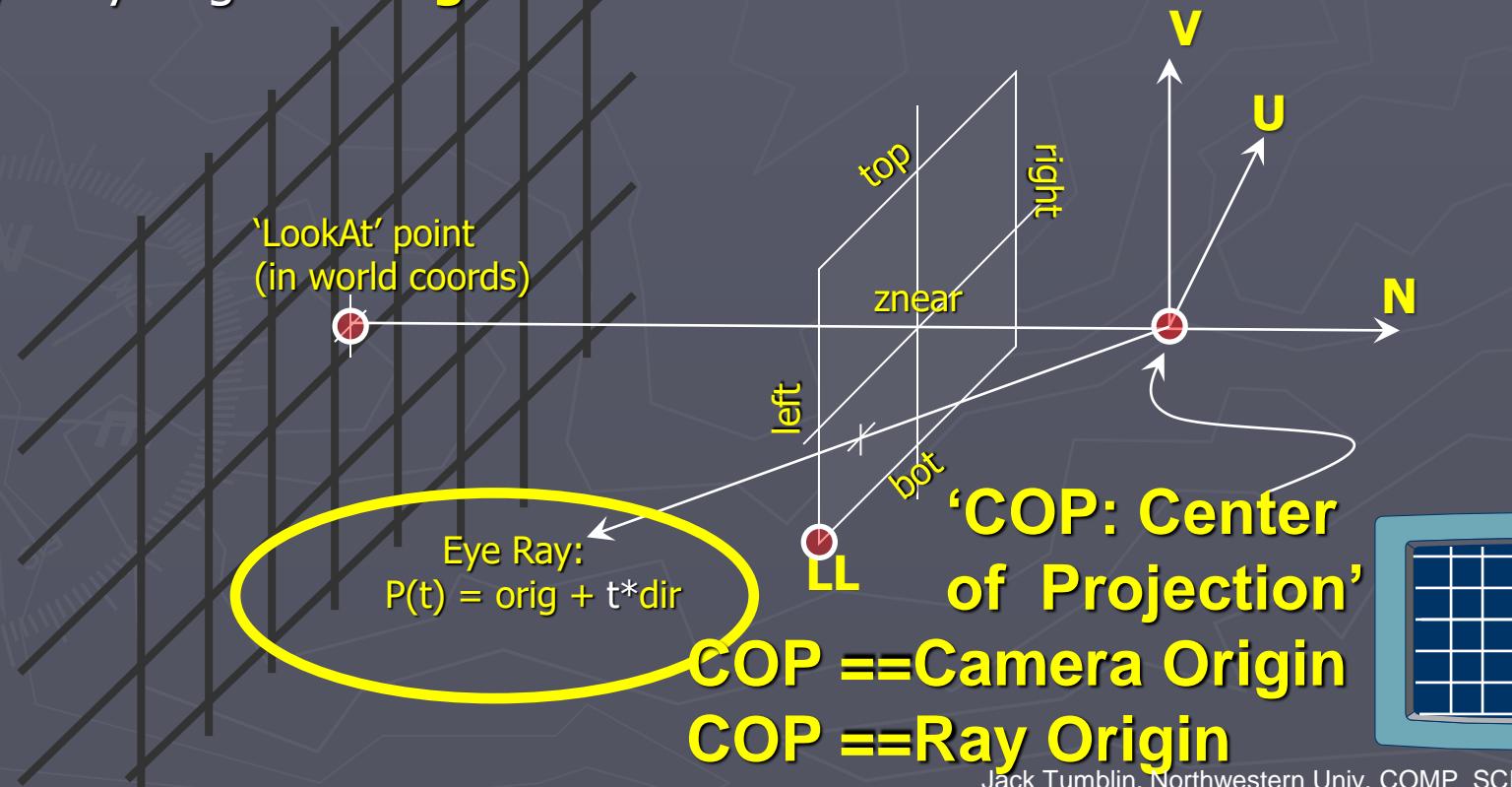
$$\mathbf{LL} = \mathbf{COP_point} + \mathbf{U}^*\text{left} + \mathbf{V}^*\text{bot} + \mathbf{N}^*(-\text{znear})$$

D) Find the pixel 'tile' width and height for image with **xmax,ymax** pixels

$$dx = (\text{right-left})/\text{xmax}; \quad dy = (\text{top-bot})/\text{ymax}$$

E) Pixel x,y tile-center: $\mathbf{P}(x,y) = \mathbf{LL} + \mathbf{U}^*dx*(x+0.5) + \mathbf{V}^*dy*(y+0.5)$

F) Eye-ray direction: $\mathbf{dir} = \mathbf{P}(x,y) - \mathbf{COP}$ (all given in world coords!):
Eye-ray origin: $\mathbf{orig} = \mathbf{COP}$



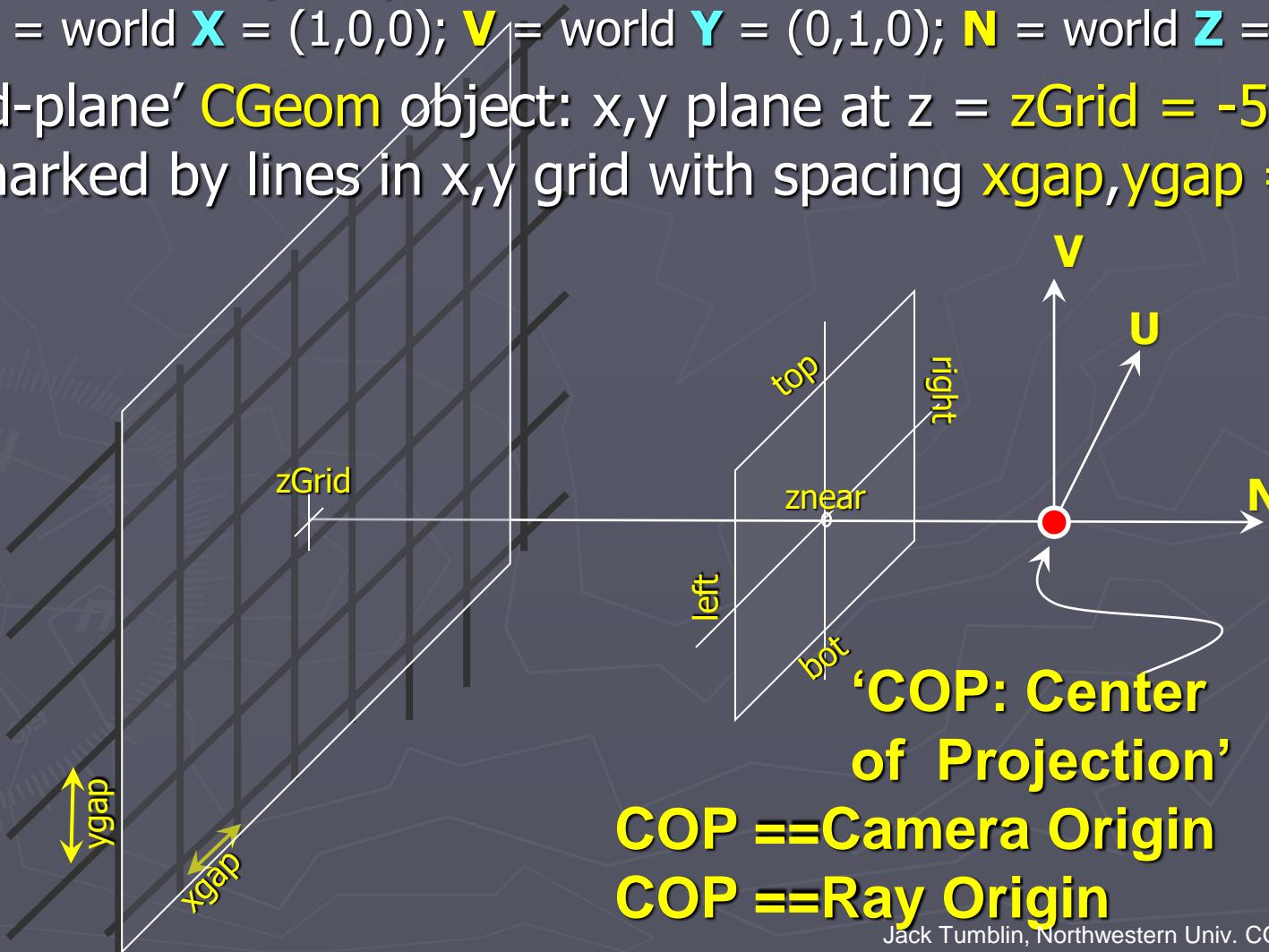
THEN: Trace a grid-plane object (1)

Keep our first camera & scene simple:

COP = world $(0,0,0)$;

U = world **X** = $(1,0,0)$; **V** = world **Y** = $(0,1,0)$; **N** = world **Z** = $(0,0,1)$;

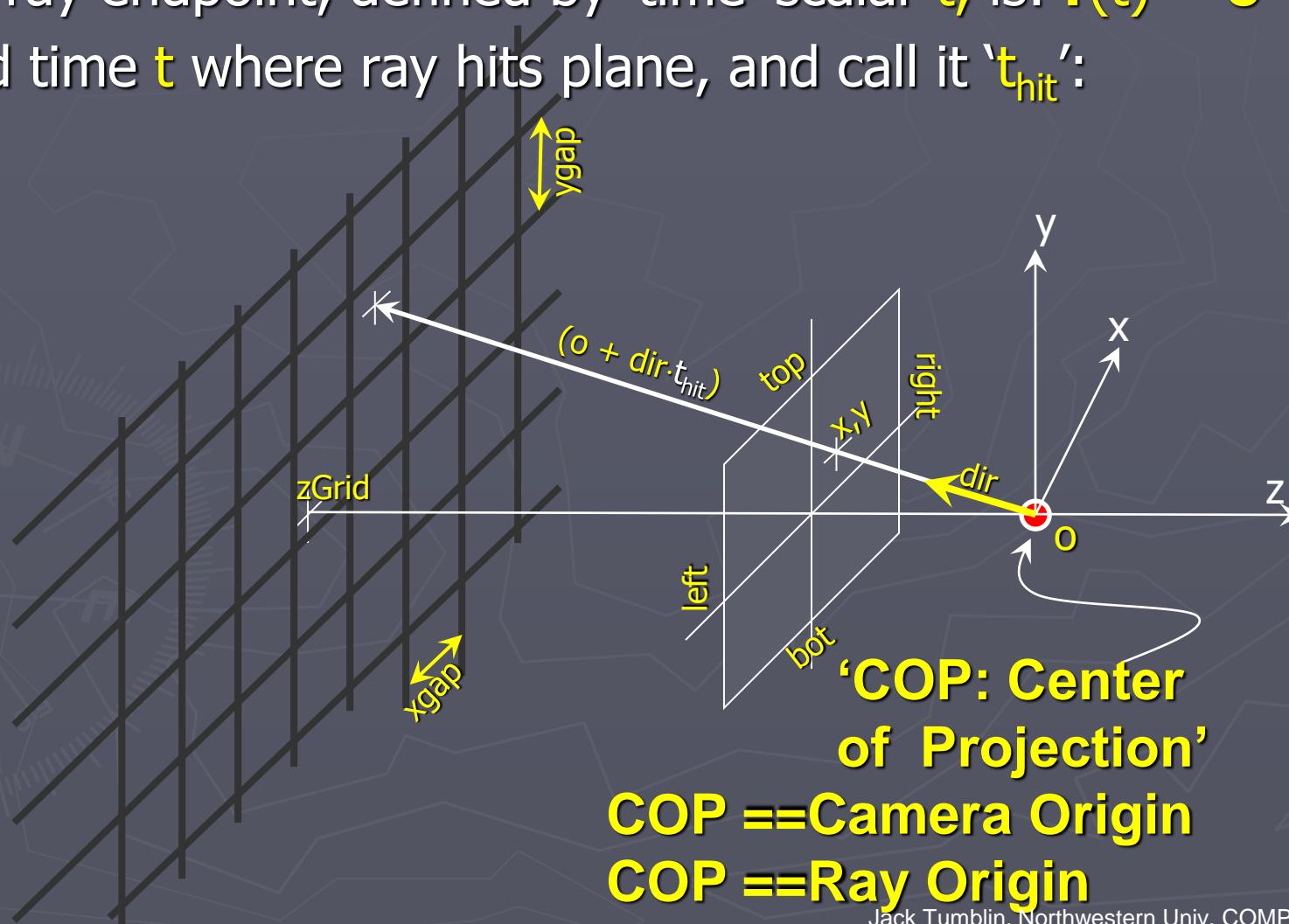
'Grid-plane' CGeom object: x,y plane at $z = z\text{Grid} = -5$;
marked by lines in x,y grid with spacing **xgap,ygap** = $(1,1)$



'COP: Center of Projection'
COP == Camera Origin
COP == Ray Origin

THEN: Trace a grid-plane object (2)

- Grid-plane implicit fcn: $f(x,y,z) = z - z_{\text{Grid}}$; surface at $f(x,y,z)=0$
- 3D ray endpoint, defined by 'time' scalar t , is: $\mathbf{r}(t) = \mathbf{o} + t \cdot \mathbf{dir}$
- find time t where ray hits plane, and call it ' t_{hit} ':



THEN: Trace a grid-plane object (3)

--Solve for $t=t_{\text{hit}}$ where $f(r(t_{\text{hit}})) = 0$: $f(x,y,z) == (z) - z_{\text{Grid}}$

$$f(r(t_{\text{hit}})) = 0 = (o_z + \text{dir}_z \cdot t_{\text{hit}}) - z_{\text{Grid}}, \text{ thus}$$

$$t_{\text{hit}} = (z_{\text{Grid}} - o_z) / \text{dir}_z \text{ and}$$

$$(x,y,z)_{\text{hit}} = r(t_{\text{hit}}), \text{ thus}$$

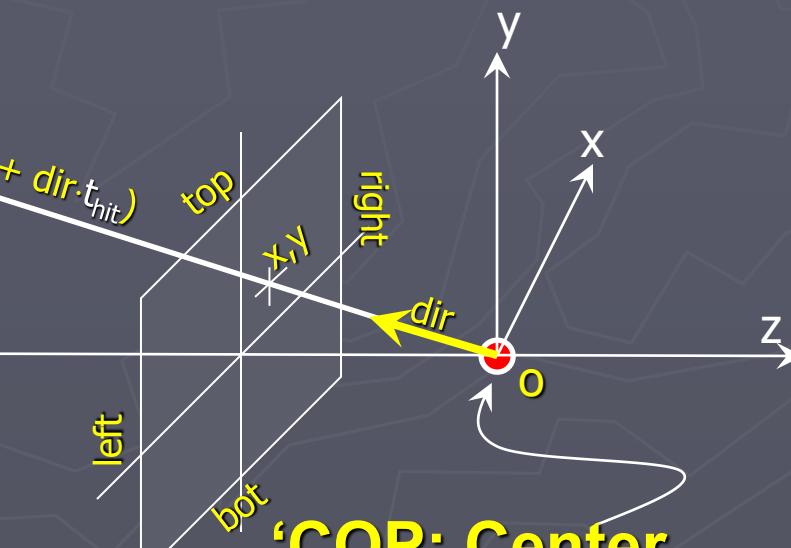
$$x_{\text{hit}} = o_x + \text{dir}_x \cdot t_{\text{hit}}$$

$$y_{\text{hit}} = o_y + \text{dir}_y \cdot t_{\text{hit}}$$

$$z_{\text{hit}} = o_z + \text{dir}_z \cdot t_{\text{hit}}$$



ray origin == $\mathbf{o} == (0,0,0)$



'COP: Center
of Projection'

COP == Camera Origin

COP == Ray Origin

(4) From 3D hit point, Find Color

What's the color on the grid-plane
at world-space position (x,y,z)?

- ▶ z component doesn't matter in lineGrid plane
- ▶ find fractional part of $(x/xgap)$; call it 'xfrac'
find fractional part of $(y/ygap)$; call it 'yfrac'
- ▶ if $(xfrac < lineWidth \text{ OR } yfrac < lineWidth)$
then use 'line' color
else use 'gap' color

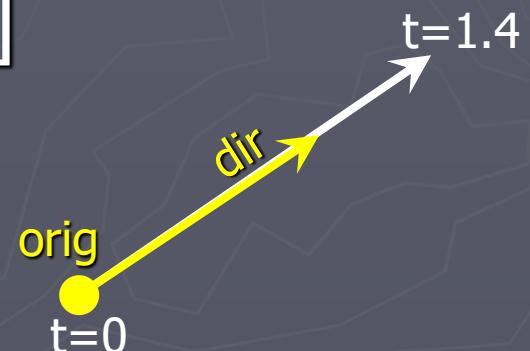
Week 1: CRay Object holds a 'ray'

- ▶ Simple as Possible! No need to normalize yet
- ▶ Describes **ONLY** ray origin & ray direction
- ▶ Parameter 't' : defines x,y,z position on ray:
$$\text{ray}(t)[4] = \text{orig}[4] + t \cdot \text{dir}[4]$$

(YES, use `vec4`, not `vec3`, for all rays!
Ensures we can apply all 3D transforms...)

CRay

```
vec4 orig;           // start point
vec4 dir;            // positive direction vector
```



YES, I strongly advise `glMatrix.js` library for ray-tracing;
Use it to replace 'cuon-matrix-quat03.js' (slow & lacks too many fcns)

Week 1: 'Basic' CCamera Object

- ▶ Describes camera geometry
- ▶ Camera creates each 'eye' ray for each pixel in our image;
- ▶ SIMPLE AS POSSIBLE for now:

COP fixed at the origin (0,0,0);

(U,V,N) axes == world (X,Y,Z) axes

CCamera xmax,ymax == frame-buffer size (256x256)

```
var iLeft, iRight, iBot, iTop, zNear; // View frustum
```

FUNCTIONS:

makeCam();

// +/-45 degree square cam at origin.
// (later: add webGL-like camera fcns:
// rayFrustum(), rayPerspective(),
// rayOrtho, etc

eyeRay = makeEyeRay (xpos, ypos); // ray from origin towards xpos,ypos
// on camera's image plane

Week 1: `lineGrid' CGeom Object

- ▶ At first, describes just one simple kind of shape:
a 2-color 'grid-plane' object
- ▶ **Boundless surface**, marked by parallel x,y lines:
 - **xgap,ygap** spacing from line-center to line-center,
 - Lines with width of **xgap*linewidth, ygap*linewidth**

CGeom

```
var zGrid;                      // where gridplane crosses z axis
var xgap,ygap;                  // line spacing on gridplane in x,y directions
var linewidth;                  // fraction of xgap,ygap filled by the line color

lineColr = vec4.fromValues(0.2,0.5,0.2,1.0); // dark green RGBA (A=opacity)
gapColr = vec4.fromValues(0.9,0.9,0.9,1.0); // off-white between lines
skyColr = vec4.fromValues(0.3,0.9,0.9,1.0) // light cyan sky.

FUNCTIONS:
traceGrid(inRay);              // find color at location where inRay hits us
```

Week 1 'Basic' Ray Tracer

JUST TWO GLOBALS:

CImgBuf g_myPic;

CScene g_myScene;

// (or a better idea; put myPic into myScene)
// image-buffer object: holds ray-traced image.
// as floating-point RGB pixels.
// 3D scene to render (ray-trace and webGL preview)

CScene

CCamera

rayCam;

// One ray-tracing camera object;
// creates eye rays, makes image.

CGeom

item[];

// Collection of 3D shapes:
// start with just one grid-plane object

var skyColr = vec4.fromValues(0.3,0.9,0.9,1.0)

// background 'sky' color (ray hit nothing!)

var blankColr = vec4.fromValues(0.5,0.5,0.5,1.0);

// neutral gray; initial value for recursive rays

FUNCTIONS:

setImgBuf(g_myPic); // Set ray-tracer to fill 'myPic' output image, and

makeRayTracedImage(); // Do it!

Big Strategy for Ray-Tracing

Simpler is always better, and usually faster!

- ▶ 1) Create a simple ray-trace camera at origin
- ▶ 2) Create, trace a simple ground-plane grid.

----- **AFTER THIS WORKS** -----

- ▶ 1) Turn the camera: can you see the horizon?
- ▶ 2) Define new shapes, but always at the origin:
(e.g. disc, sphere, cone, cube, torus, plane,...)
- ▶ 3) As with WebGL, use transformations to move,
stretch, squash, and twist them
- ▶ 4) Surprise: DON'T transform any shape or ;
transform the ray! (see Week 2)

Objects for a full Ray Tracer

Write 'dual use' stub classes/prototypes: we will use them for BOTH webGL AND for ray-tracing:

CScene:	master class for webGL + ray-tracing
CCamera:	defines 1 camera frustum, position, & aiming
CGeom:	describes 1 shape we see in the scene
CMatl:	describes 1 material (includes textures)
CLight;	describes 1 light source (point, area, etc)
CImgBuf:	holds 1 Image Buffer for ray-traced result (a Float32Array ; avoids quantizing errors)

Ray-tracing-only objects and functions:

CRay	describes 1 ray: origin, direction, usage, etc
CHit	describes 1 ray/object intersection
CHitList[]	Array or List CHit objects found for one ray
<u>Trace(ray);</u>	// Find ALL ray/shape intersections (CHits)
<u>FindShade();</u>	// Find ray's color at 1 hit-point (Recursive!)

'EVERYTHING' Ray Tracer

JUST TWO GLOBALS:

```
CImgBuf myPic;           // image buffer: holds one ray-traced image.  
CScene myScene;          // one 3D scene to render (ray-trace & webGL preview)
```

CScene

CCamera

// One ray-tracing camera object:
// creates eye rays to trace an image.

GeomList → CGeom | CGeom | CGeom | ... | CGeom // Shape COLLECTION

MatlList → CMatl | CMatl | CMatl | ... | CMatl // Materials COLLECTION

LampList → CLight | CLight | CLight | ... | CLight // Lights COLLECTION

'EVERYTHING' Ray Tracer

JUST TWO GLOBALS:

```
CImgBuf myPic;  
CScene myScene;
```

```
// (think about it: should myPic be member of myScene?)  
// image buffer: holds ray-traced image.  
// 3D scene to render (ray-trace and openGL preview)
```

CScene

```
CCamera rayCam; // Ray-tracing camera object: eye-ray source  
  
var depthMax; // recursion depth limit  
  
var itemCount; // how many shapes we have  
var CGeomList = []; // COLLECTION: array of shapes  
  
int matlCount; // how many materials we have  
var CMatList = []; // COLLECTION: array of materials  
  
var lampCount; // how many lights we have  
var CLampList = []; // COLLECTION: array of light sources
```

Strategies for Simplicity

► Key Idea for *Versatile* Ray Tracing:

- 1) Always describe shapes centered at the origin,
- 2) Use WebGL-like transformations to move, stretch, squash, and twist them
- 3) Ray-Trace uses “backwards” transformations:

► DON'T transform the shape;

transform the ray! ;

- WebGL: 'modelView'
Ray-tracing: 'ViewModel'

SOUNDS nice—lets us keep and use our same
'tree of transformations' etc. **but how?**

Goals for Week 1 is:

- ▶ 1) Ray-trace a 2-colored grid-plane
 - Your own Camera, Geometry, Ray, & Hit classes
 - Solve for t_{hit} to find where ray hits grid-plane
 - Do everything simply, in world space:
 - ▶ Camera at origin aimed at -z; grid-plane at zGrid=-5,
 - ▶ From computed $(x_{\text{hit}}, y_{\text{hit}})$, find ray color for x,y grid
 - ▶ LOW resolution – just 256 x 256 please!
- ▶ 2) Rotate the camera 90° to see horizon:
 - Create your own `CCamera.setLookAt()` function to position & aim the camera in world coords.
 - Investigate – why does the horizon look so bad?!?!

Next Week: Transformational Treachery

Ray Trace: DON'T transform the shape;
transform the ray!

- ▶ OK: HOW? (Week 1):
 - Start with ray-camera at origin: simple!
 - Start with grid-plane at $z=z_{\text{Grid}}$; well OK;
- ▶ THEN add in some TRANSFORMS:
 - webGL: (fwd) 'push object out from camera'
in MODELVIEW: `gl.Translate(0,0,-5)`
 - rayTrace: (inverse) 'pull ray into object coords'
in CGeom: `worldRay2model ← trans(0,0,+5)`

END



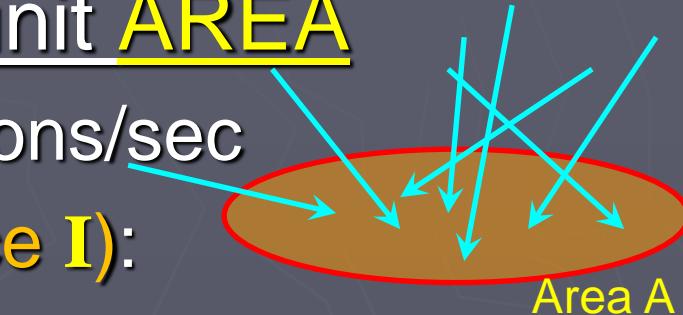
END

RECALL: How do we MEASURE the light in a Ray?

First, define Light Power per unit AREA

- ▶ Flux **W** = power, Watts, # photons/sec
- ▶ Irradiance **E** (or Radiant Exitance **I**):
flux arriving (or leaving) per unit area,
(regardless of direction or wavelength)

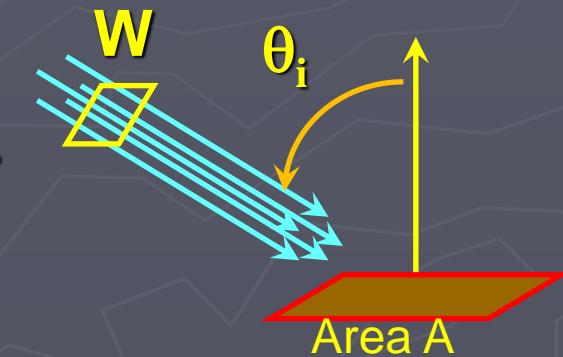
$$E = I = \text{Watts/area} = dW / dA$$



Second, apply directional effects:

Irradiance **E** falls by incident angle θ_i :

$$E = \cos(\theta_i) W/A;$$

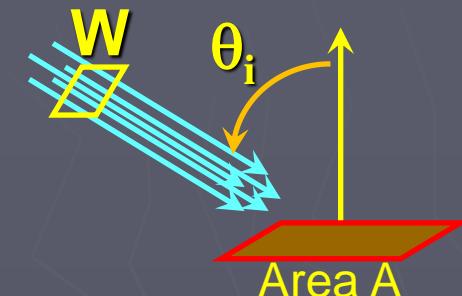


RECALL: How do we MEASURE the light in a Ray?

- Irradiance **E** falls by incident angle θ_i :

$$E = \cos(\theta_i) W/A;$$

(units: watts/meter²)

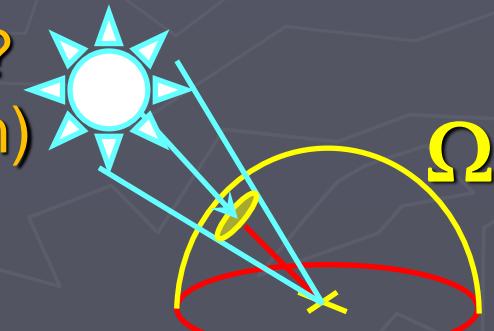


- Total Light-Power from **ALL** directions?

- weight the flux (in watts) by its direction $\cos(\theta_i)$
(remember, Radiance **L** removes that dependence!)
- integrate incoming flux over entire hemisphere Ω :
- (Units for this ‘incoming flux’ we measure?
it’s the ‘radianc intensity’ in watts/steradian)

Wait Wait Wait

?!?! What’s a ‘**steradian**’ again?



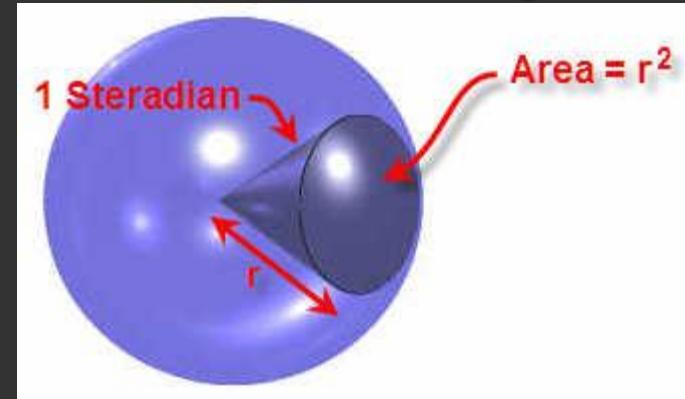
How to Measure a ‘Cone’ of Angles

Measure a span-of-directions in 3D? \equiv “Solid Angle”

- Unit sphere ($r=1$) around 3D origin:

$$x^2 + y^2 + z^2 = 1^2$$

DETAILS: Every point on sphere specifies
one unique direction from origin; ALSO
Every direction from point specifies
one unique sphere point.
(formally: a bijection: one-to-one && onto)



area ==solid angle
in steradians

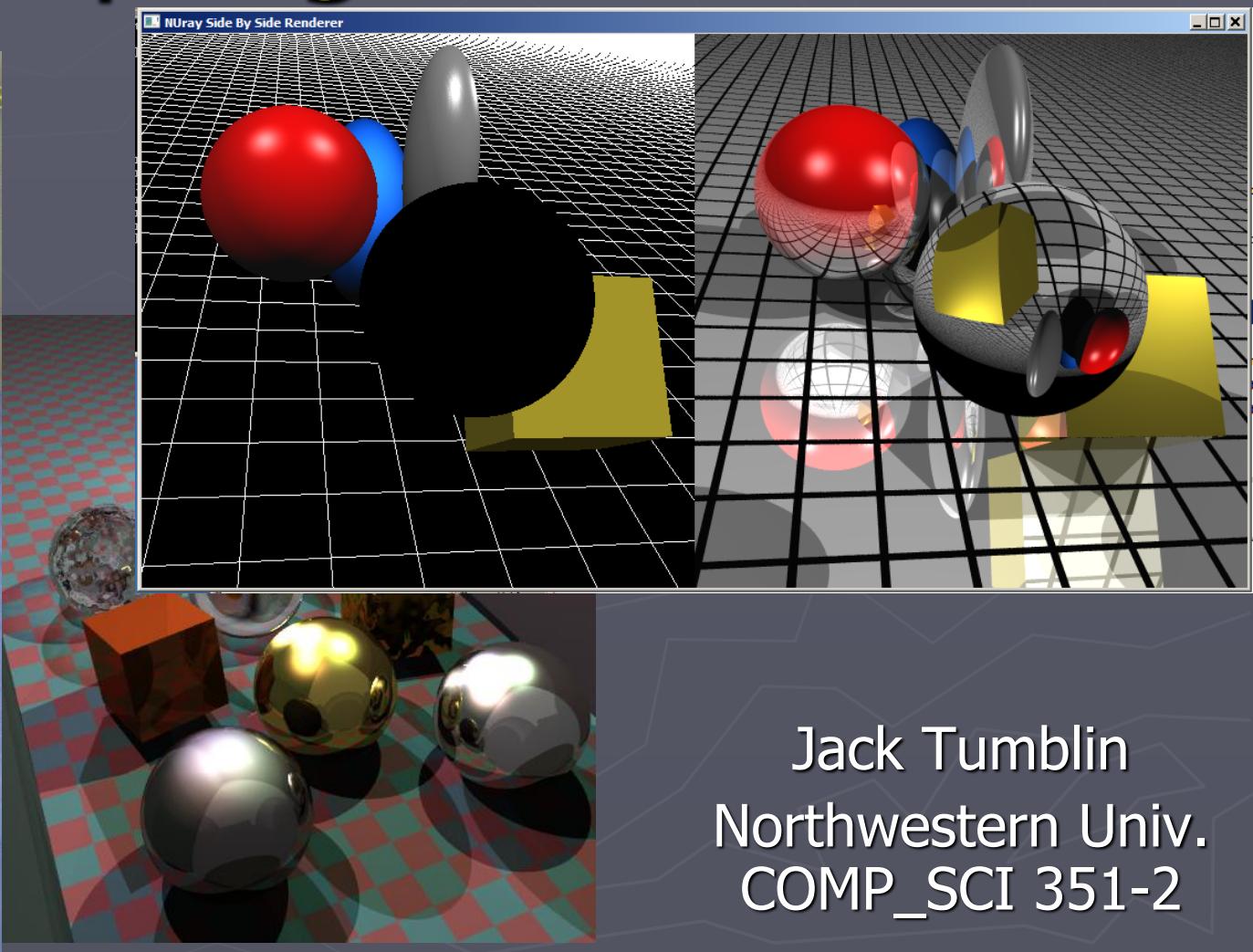
- ‘Solid Angle’ measures a 2D span of directions by their 2D span of surface area on the unit sphere

- entire sphere surface area: 4π (in units of radius²)
entire span of all directions: 4π steradians

see: <http://www.mathsisfun.com/geometry/steradian.html>

and: <http://apps.usd.edu/coglab/schieber/trb2000/sld021.htm>

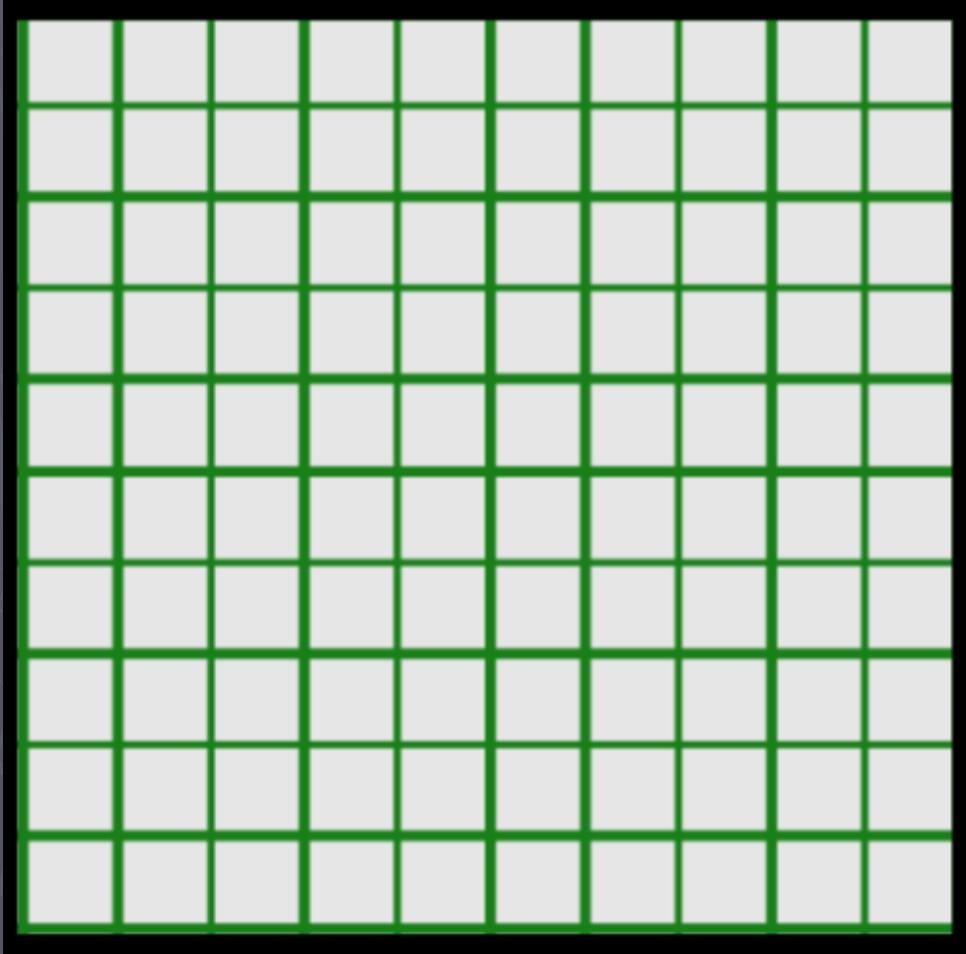
Ray Tracing C: Ray Sampling & Transforms



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

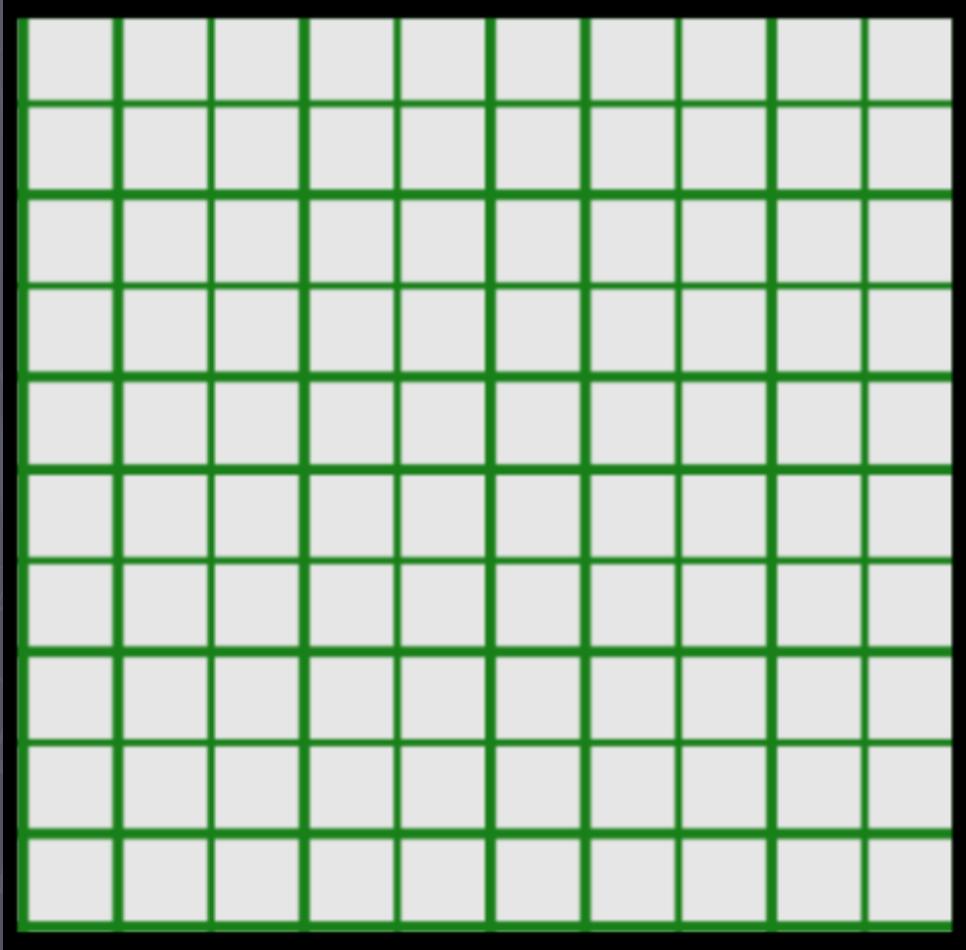
Can you Turn your Camera Yet?

- ▶ Your first ray-traced camera aims at the lineGrid object for pictures that look like a grid:



Can you Turn your Camera Yet?

- ▶ Your first ray-traced camera aims at the lineGrid object for pictures that look like a grid:



SURPRISE!

- This starter code:
[Week1LineGrid_READY](#)
does this for you!
- RUN the code, press 't' key.
- Look at code here:
 - [Week01_traceSupplement](#)
at -- line 98...
at -- line 200...

Recall: Make 'Eye' Ray in World Coords

C) Find image rectangle's lower-left corner in world coord system:

$$\mathbf{LL} = \mathbf{COP_point} + \mathbf{U}*left + \mathbf{V}*bot + \mathbf{N}*(-znear)$$

D) Find the pixel 'tile' width and height for image with **xmax,ymax** pixels

$$dx = (\text{right-left})/\text{xmax}; \quad dy = (\text{top-bot})/\text{ymax}$$

E) Pixel x,y tile-center: $\mathbf{P}(x,y) = \mathbf{LL} + \mathbf{U}*dx*(x+0.5) + \mathbf{V}*dy*(y+0.5)$

F) Eye-ray direction: $\mathbf{dir} = \mathbf{P}(x,y) - \mathbf{COP}$ (all given in world coords!):
Eye-ray origin: **orig** = **COP**



NEXT: Turn Camera 90° to see Horizon

G) BEFORE: camera axes in world coords were:

U == world **X** = (1,0,0); **COP** = world (0,0,0)

V == world **Y** = (0,1,0);

N == world **Z** = (0,0,1);

Now turn camera 90° on it's **U** axis...

D) AFTER: camera axes in world coords are:

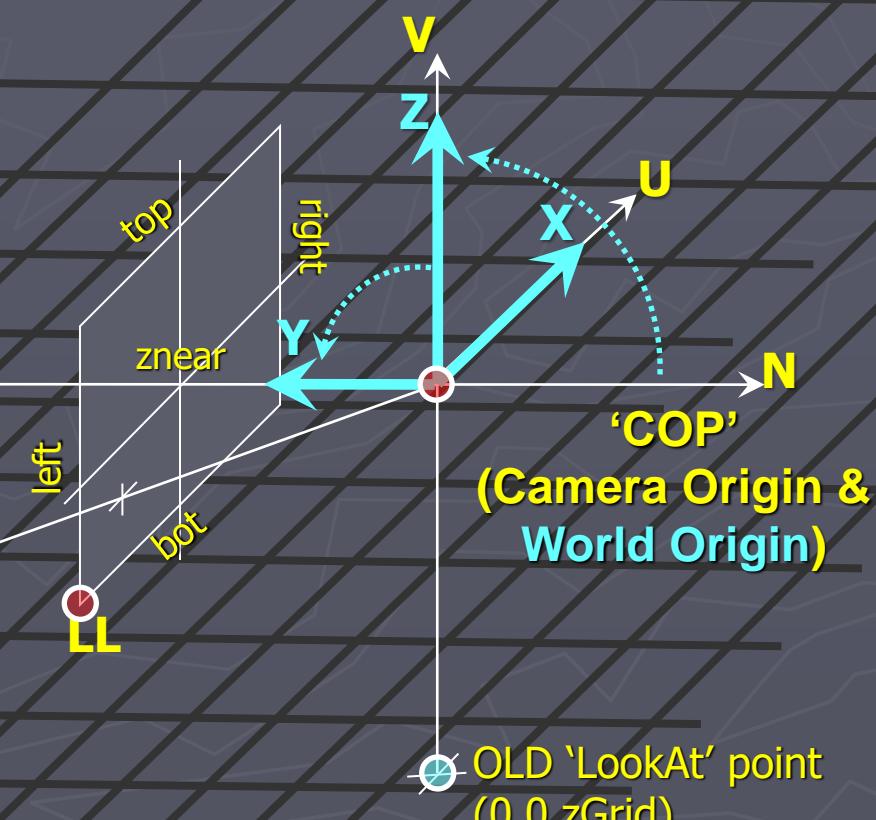
U == world **X** = (1, 0, 0);

V == world **Z** = (0, 0, 1);

N == world **-Y** = (0,-1, 0);

NEW
'LookAt' point
(0, zGrid, 0)

Eye Ray:
 $P(t) = \text{orig} + t * \text{dir}$



SURPRISE! Horizon Looks Bad!

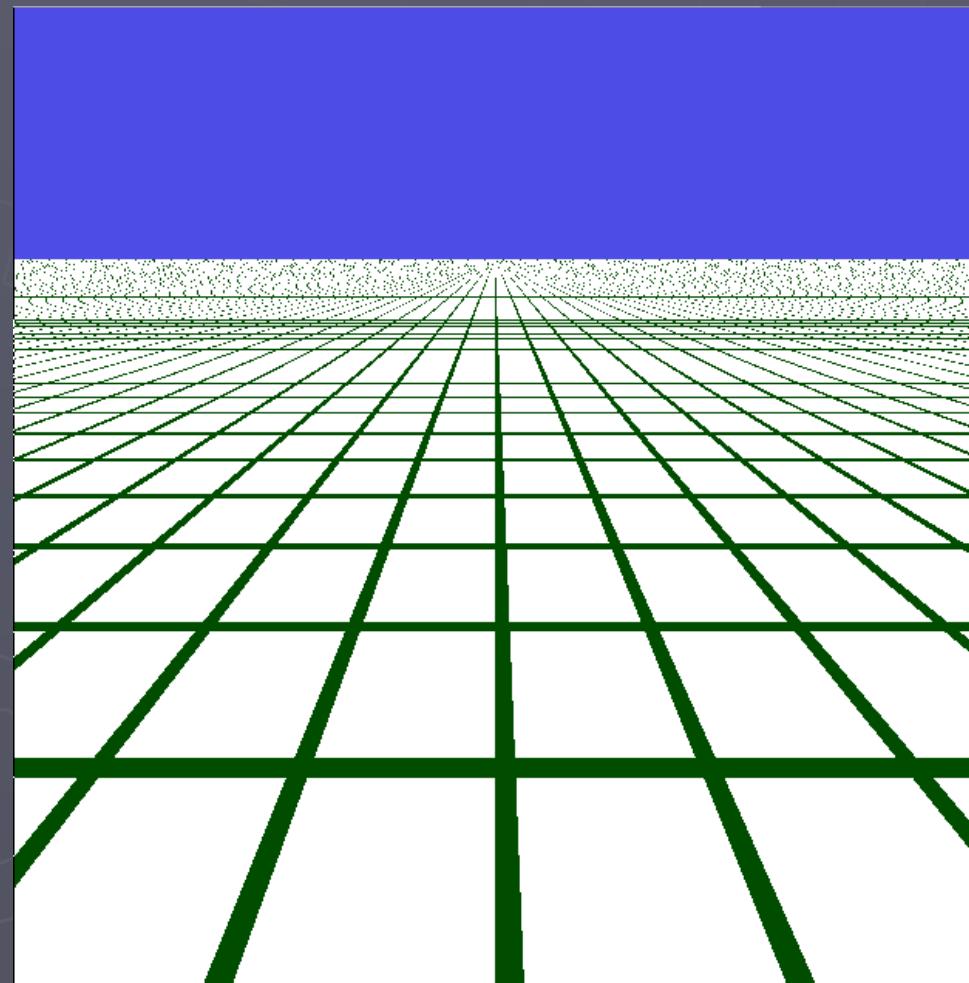
Do you know why?

Look closely...

What's the approx. spacing
between the ray hit-points
measured on ground plane?

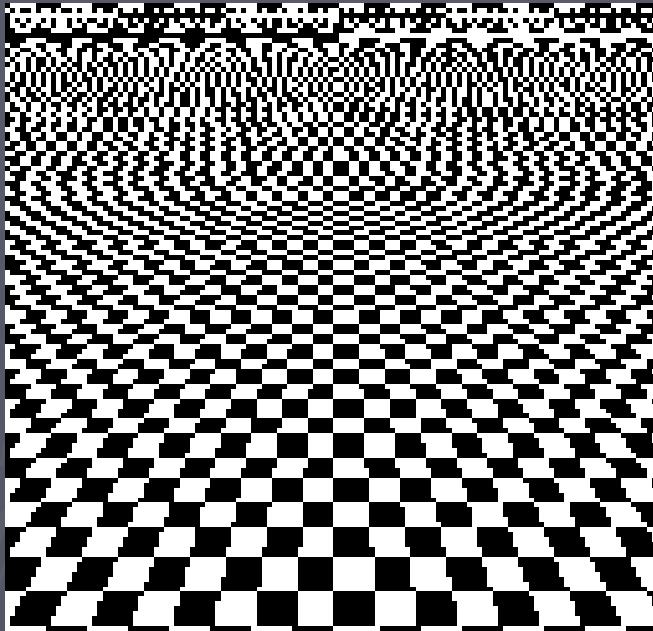
- Near the camera?
- Near the horizon?

Compare:
what's the approx. spacing
between gridPlane lines?

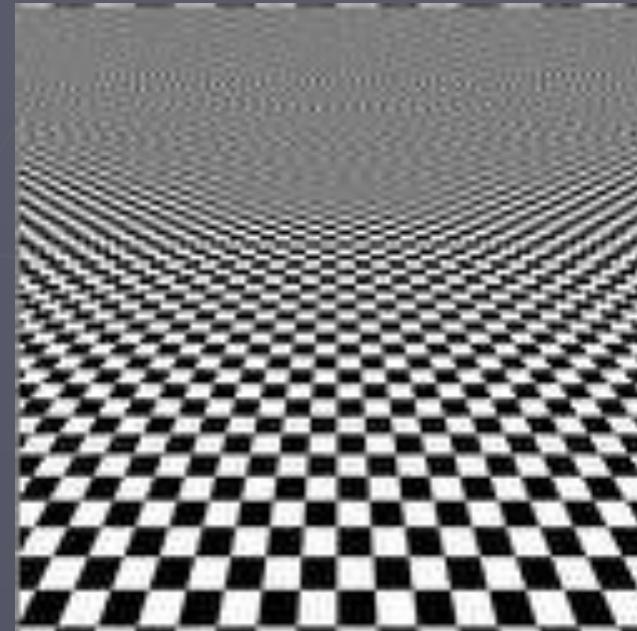


You're Seeing Aliasing Problems!

- ▶ Coarse periodic point-sampling misses fine features!

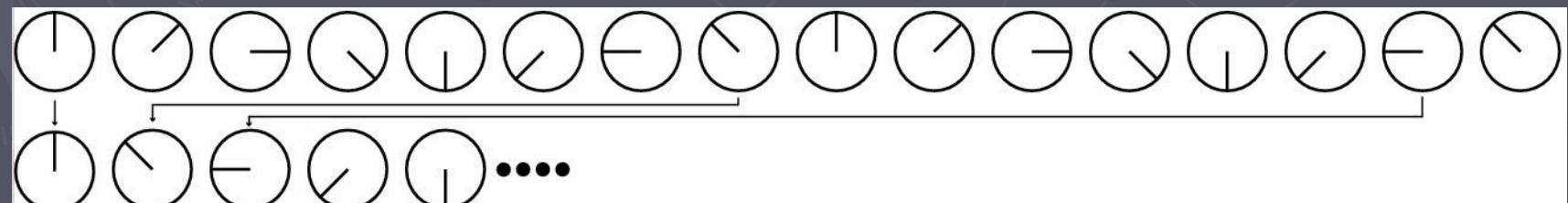


Aliased



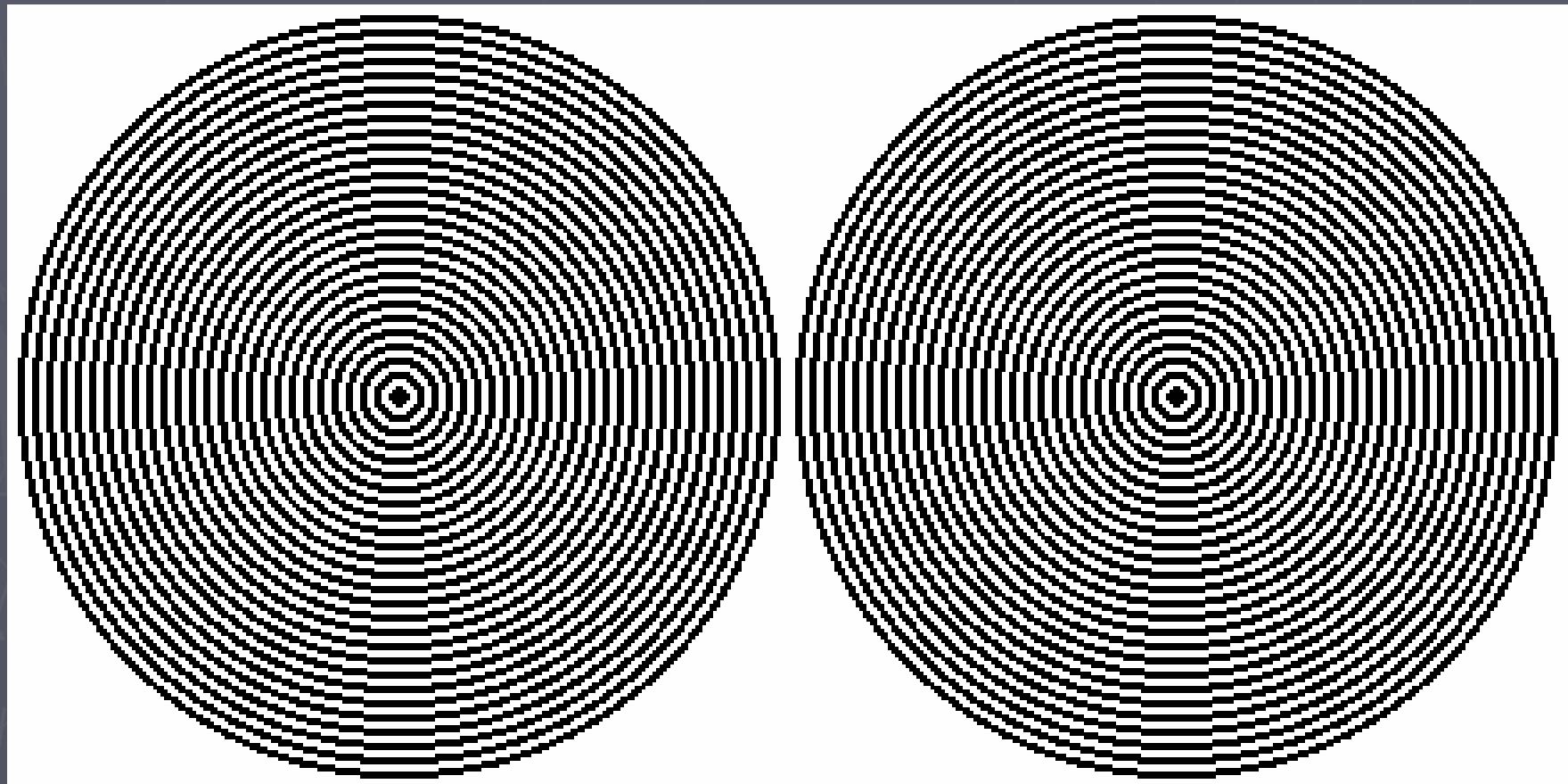
Better: Anti-aliased

- ▶ Temporal aliasing: 'Backward-spinning Car Wheels':



Related: Moire' Patterns

- When Periodic Sampling ALMOST Matches Periodic features, we see big ghostly shapes:



Aliasing Examples

(from slide by Marcus Magnor)

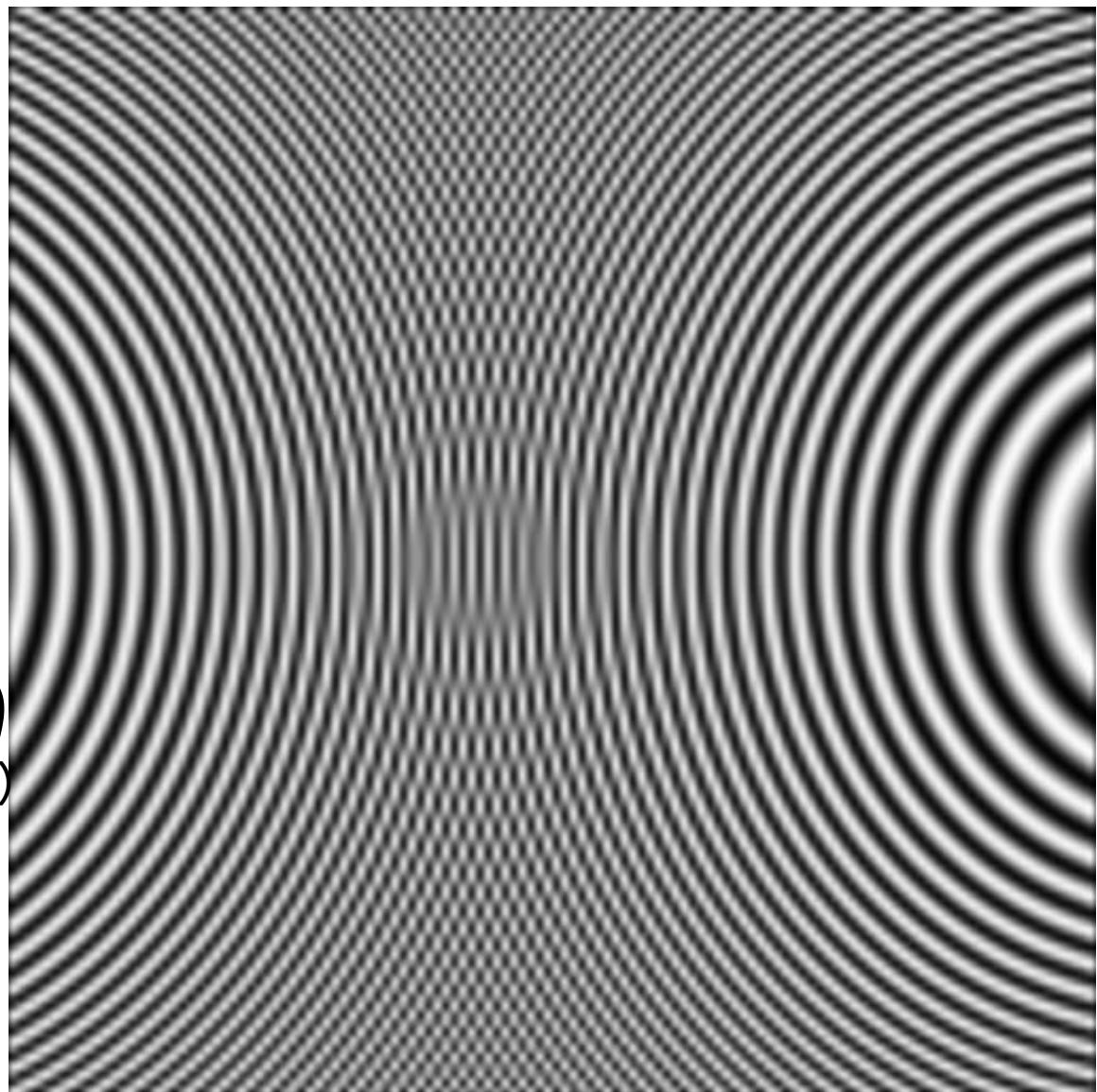
'Zone Plate' Pattern:

$$I(x,y) =$$

$$0.5 + \sin(2\pi f(x^2 + y^2))$$

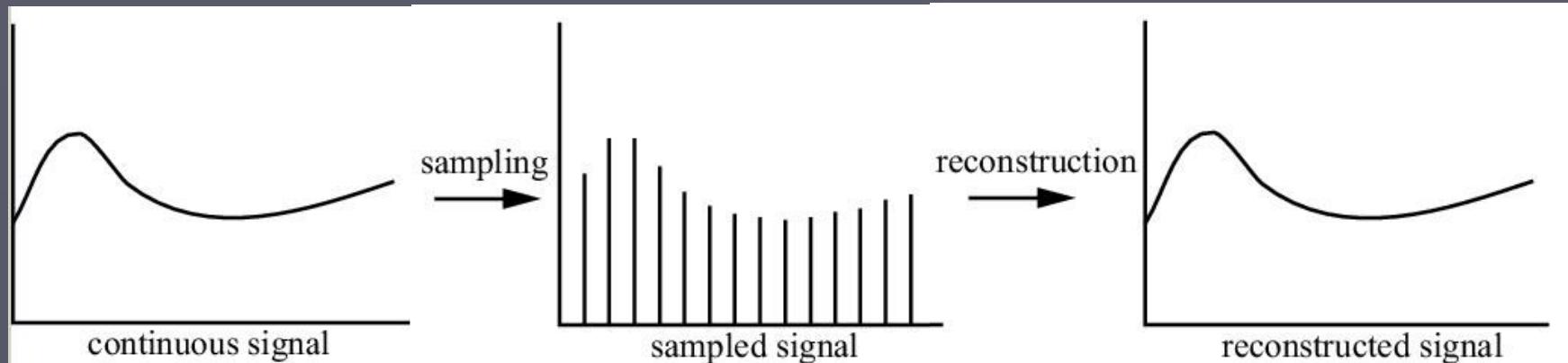
(origin is
about here)

only ONE sinusoid;
NOT repeated!
→ALIASING←
changes high freq. to low...



Aliasing: What Causes It?

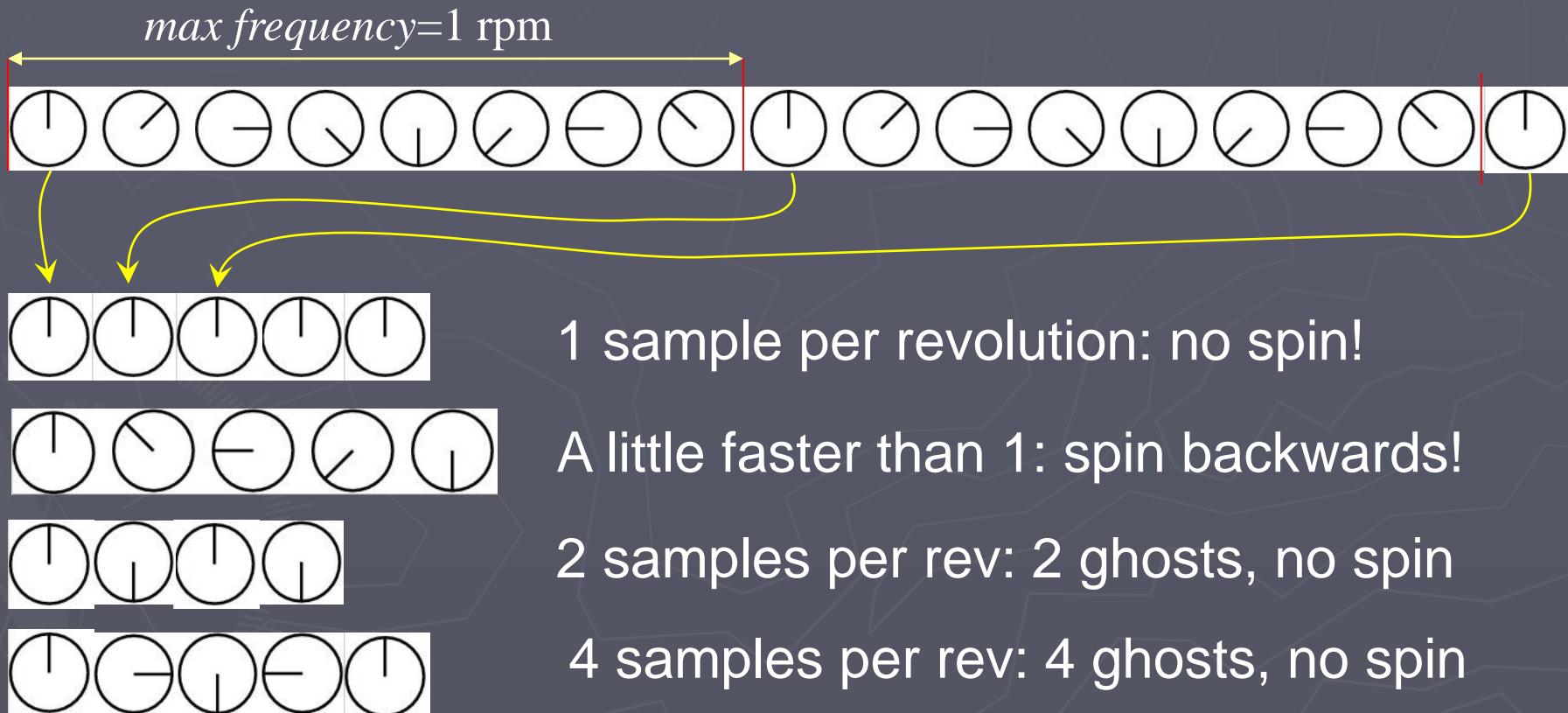
- ▶ Digital Signals: sampling & reconstruction



- ▶ **Nyquist Theorem:**
to capture *all* signal details, sampling freq.
must be $>2X$ max signal frequency
- ▶ **Trouble!** No sampling rate is enough for
sharp triangles: infinitely sharp edges!

Sampling: One-Spoke Car Wheel

► Nyquist says: set sample rate $> 2X$ max freq

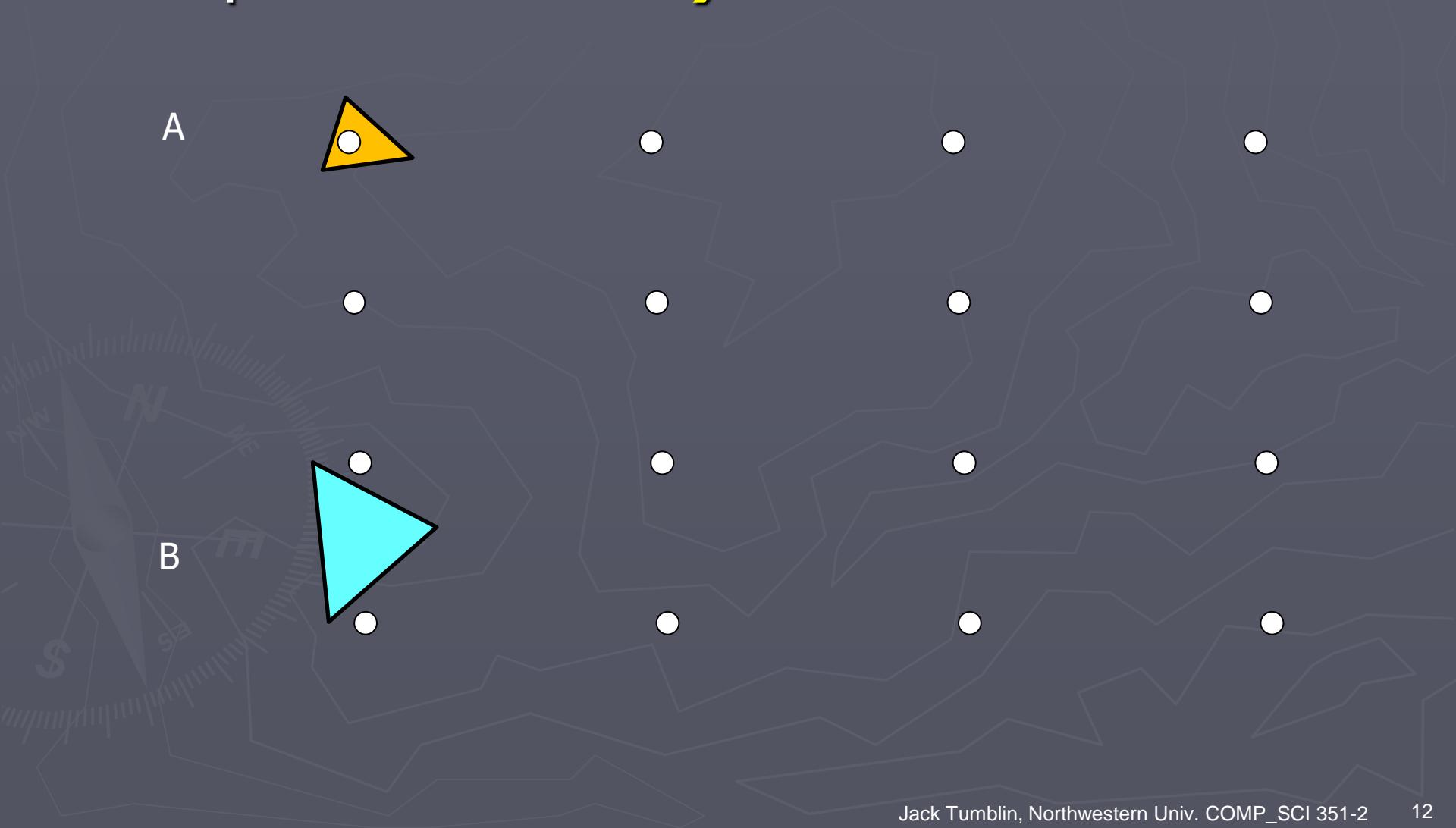


from: Tomas Akenine-Möller, 2003

A little SLOWER than 1 sample/rev?
What happens?

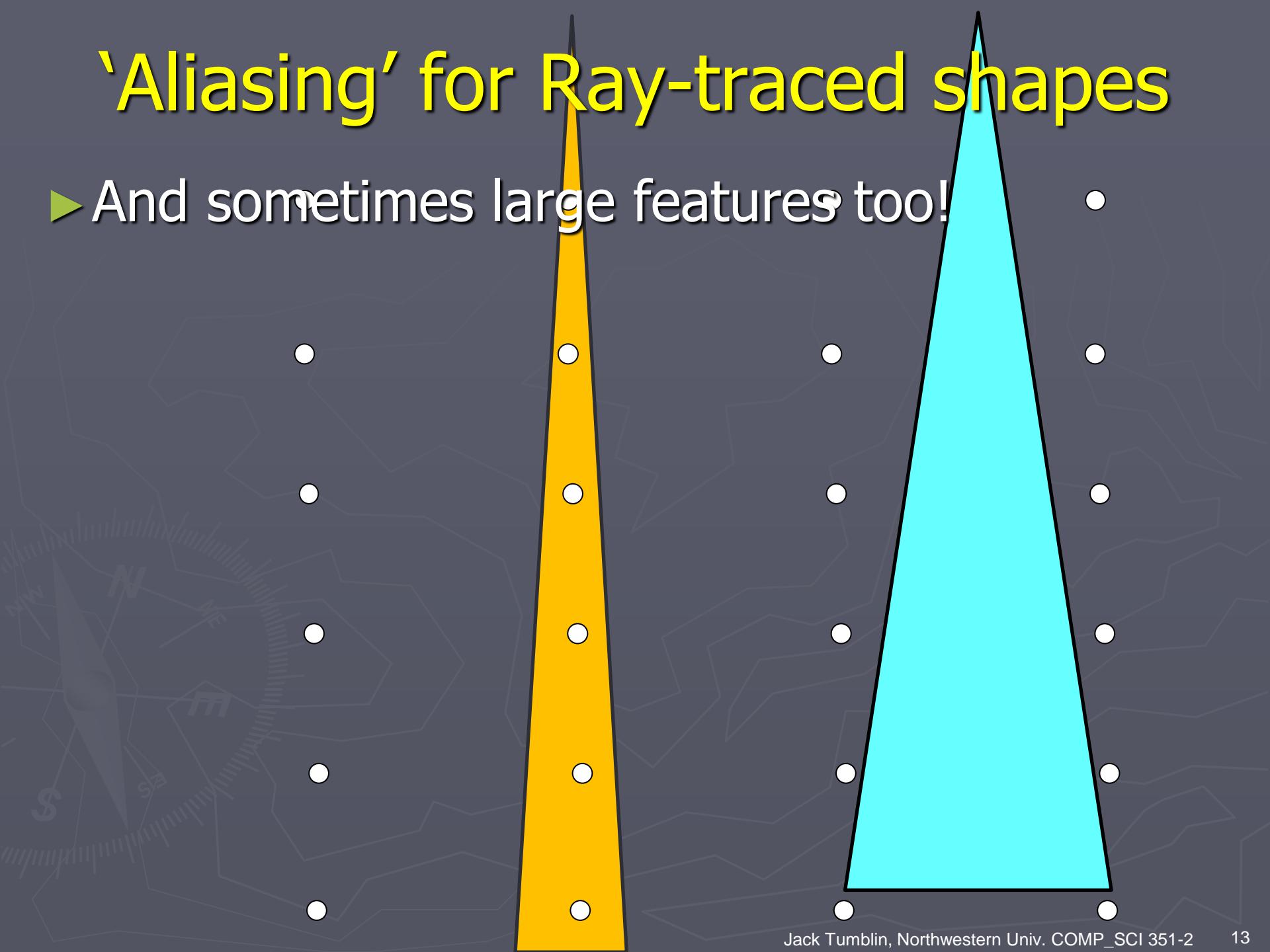
'Aliasing' for Ray-Traced Shapes

- samples can *entirely miss* small features



'Aliasing' for Ray-traced shapes

- And sometimes large features too!



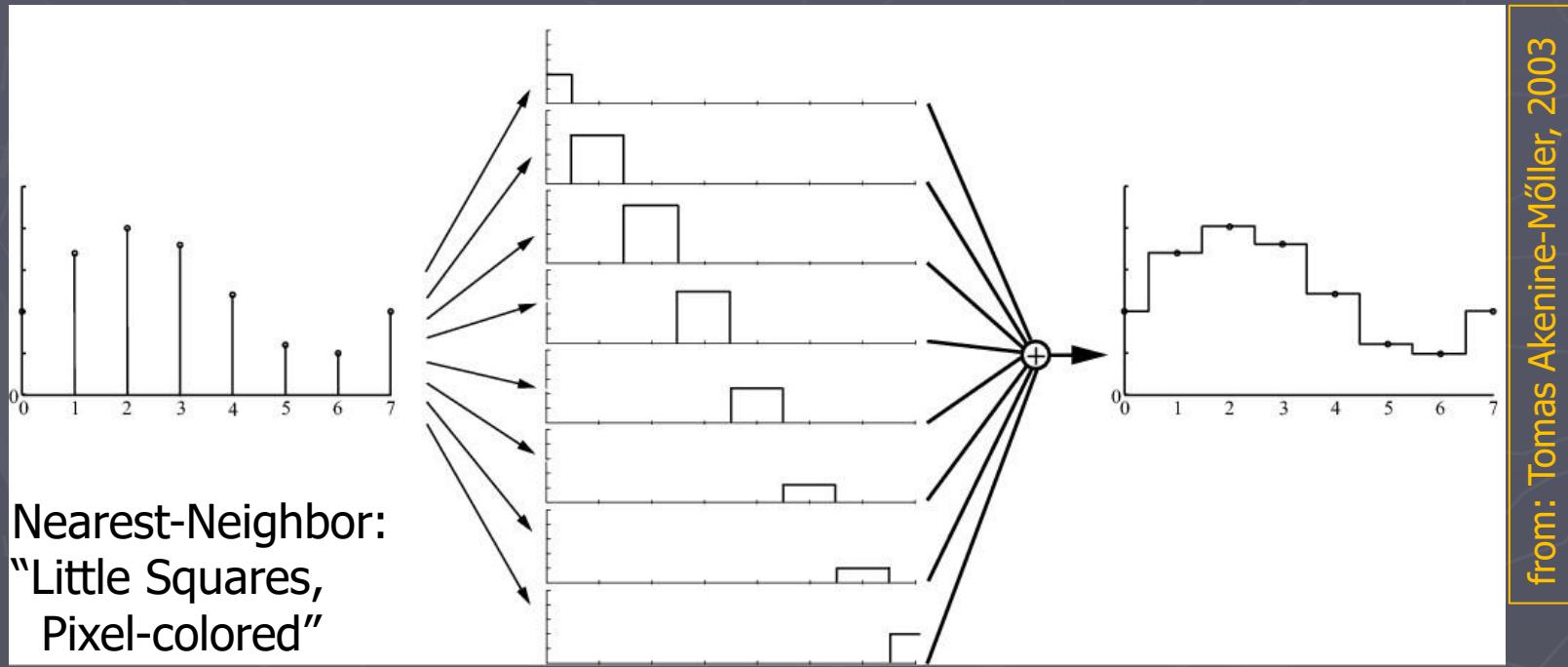
The Frosted-Glass Analogy

(on chalkboard)

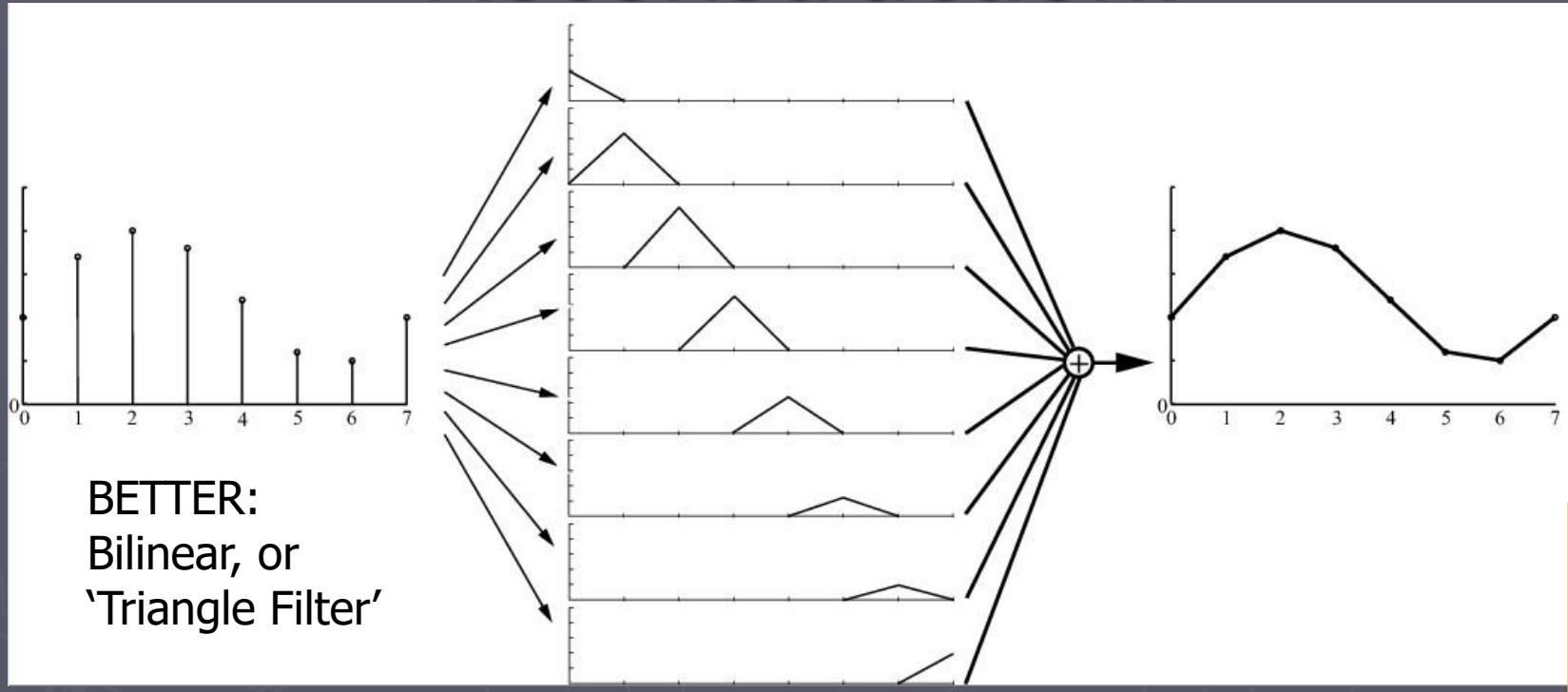
- ▶ **Key idea:** Pixels == infinitesimal points!
(Alvy Ray Smith: 'A pixel is NOT a little square!!' (1996))
Key idea: Each pixel should summarize image content within **a local neighborhood**
- ▶ Display pixel grid \approx grid of point-like lights covered by patterned or frosted glass
- ▶ Overlapped 'glow' from lights form image; frosted glass \approx reconstruction filter
- ▶ Anti-Aliasing Problem: what 'filter' best summarizes scene around the pixel location?

Reconstruction:

- ▶ Pixel Value == value at a single point that *summarizes* a small image *neighborhood*;
- ▶ How do we re-create an image from pixels?
- ▶ 'Point-light Pixels + Frosted-glass' analogy



Reconstruction:



32x32
texture



Nearest neighbor Filter

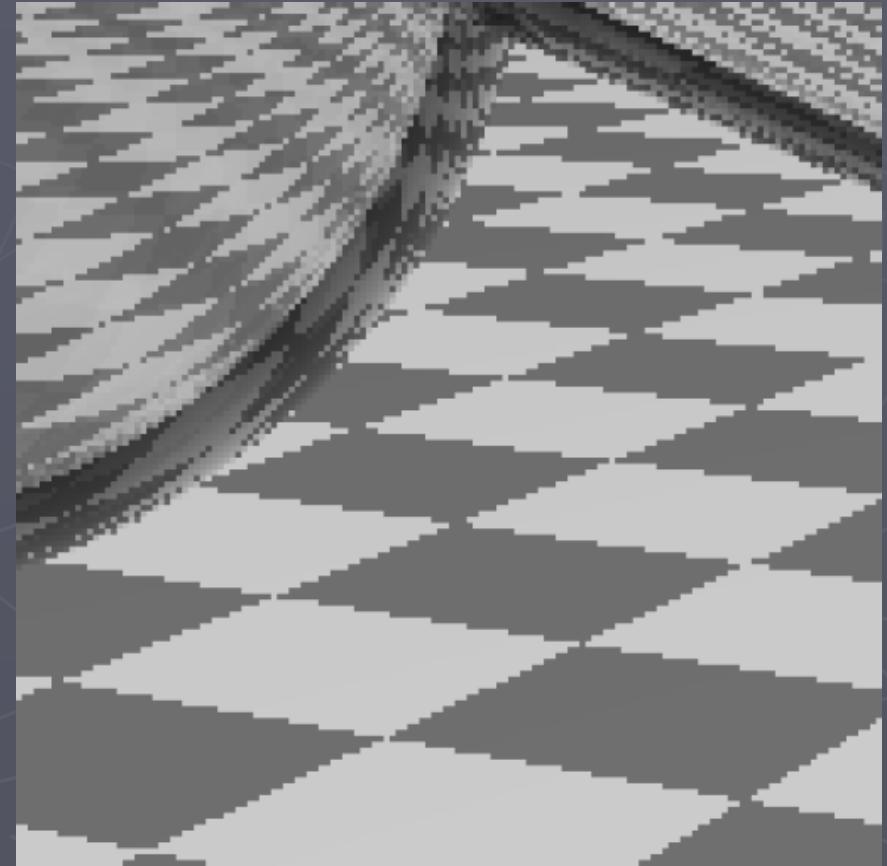
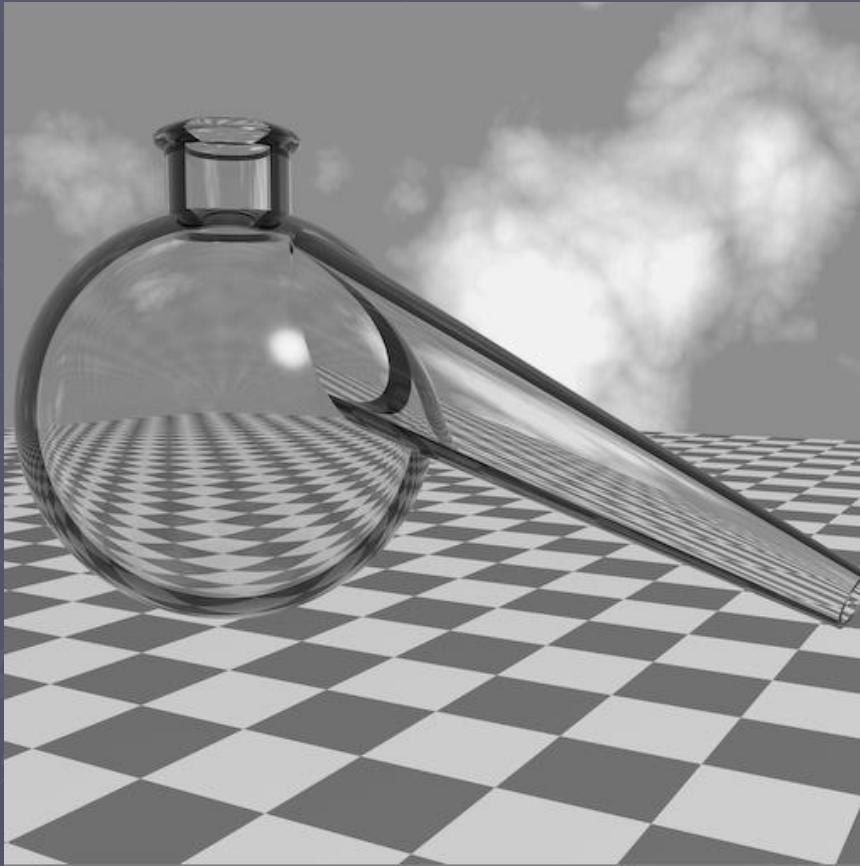


Bilinear Filter

from: Tomas Akenine-Möller, 2003

Very Early Ray-Traced Image

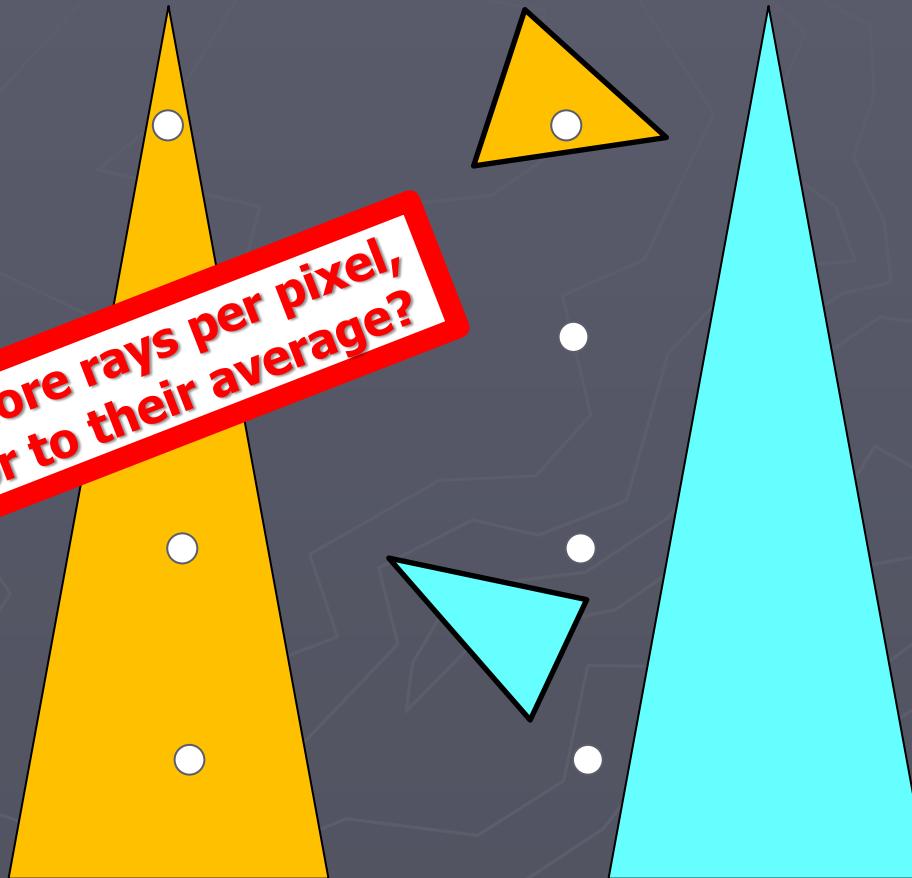
- ▶ Close-up – jagged pixels, or ‘jaggies’
- ▶ ‘Pixels’ shown as tiny little fixed-color squares:



Aliasing for Ray-Traced Shapes

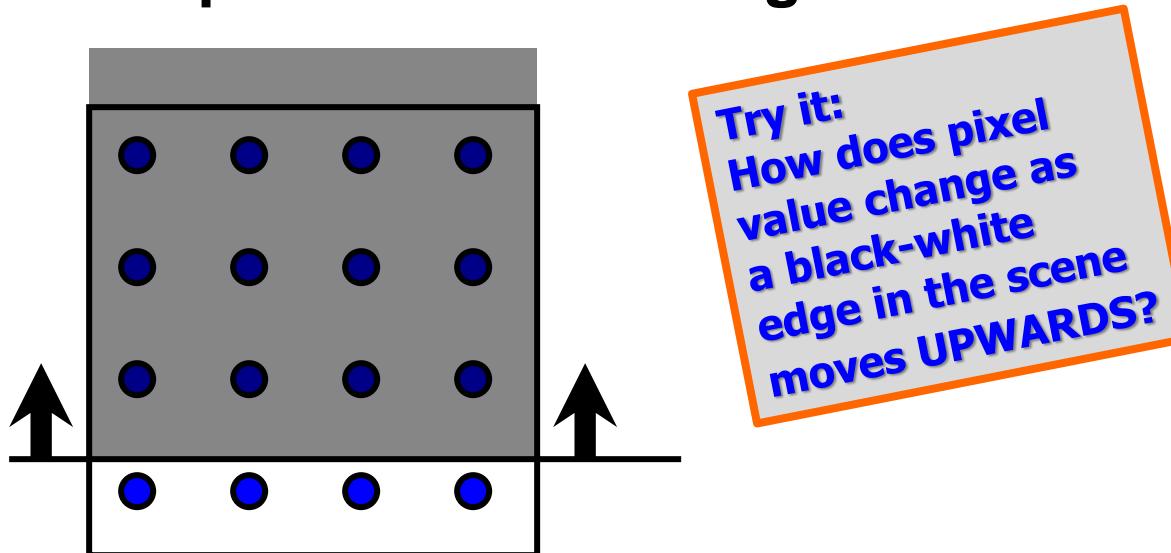
- samples can entirely miss vital features

What if we sent more rays per pixel,
and set pixel color to their average?



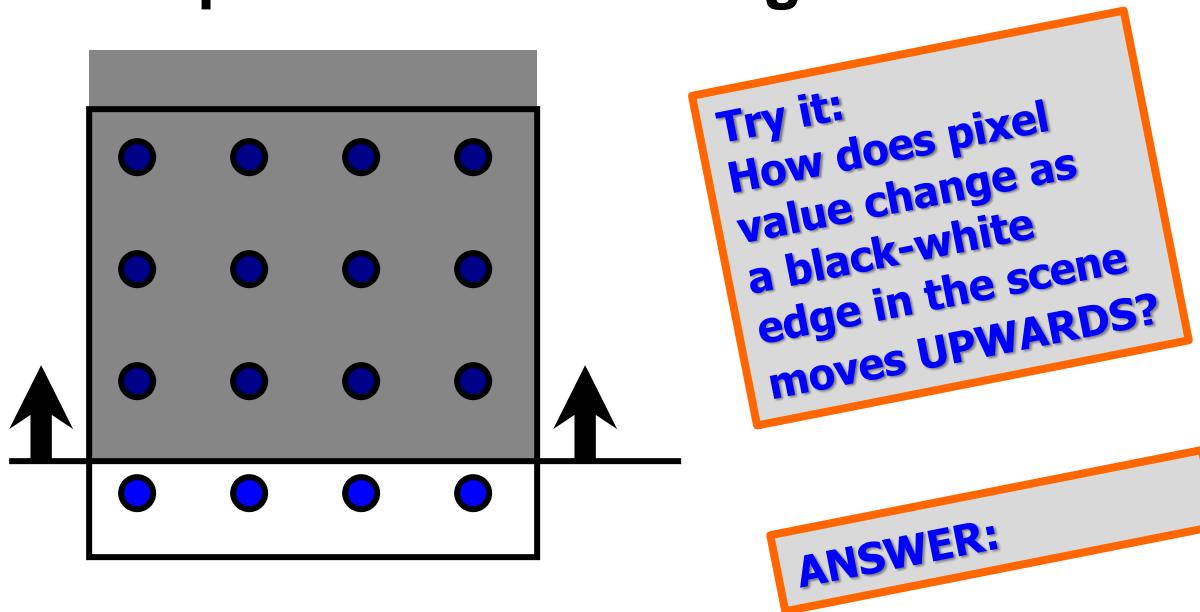
Obvious Solution: Super-sampling

- **RECALL:** one pixel summarizes image content in a small local neighborhood (e.g. the square tile below)
- Suppose we shoot **MULTIPLE** rays thru that square and set pixel to their average color?



Super-sampling in Practice

- **RECALL:** one pixel summarizes image content in a small local neighborhood (e.g. the square below)
- Suppose we shoot **MULTIPLE** rays thru that square and set pixel to their average color?

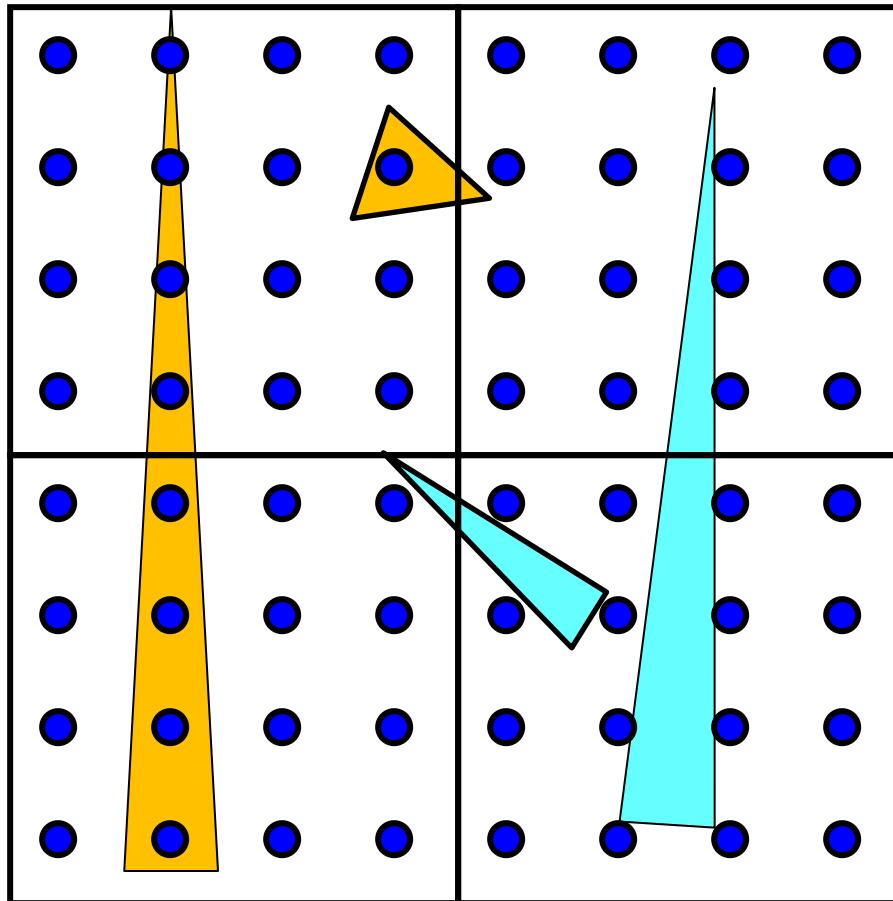


Ray-traced pixel value as black-white edge moves:

$0 \rightarrow 4/16 \rightarrow 8/16 \rightarrow 12/16 \rightarrow 16/16$

Super-sampling in Practice

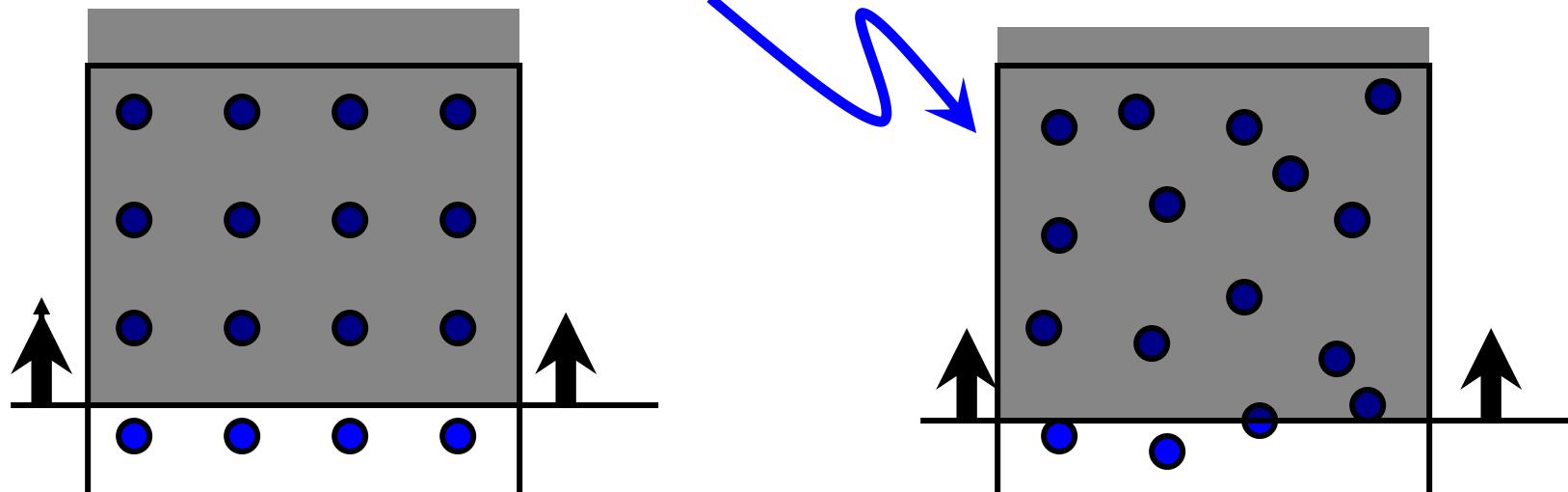
- **Problems with uniform, periodic supersampling**
 - Expensive: 4-fold to 16-fold effort
 - Non adaptive: Same effort everywhere
 - **Still Too uniform!** # of intensity levels \leq # of super-samples



STILL misses tiny features!
NEVER guarantees the correctly-averaged pixel value!
?How could we reduce that error 'most of the time'?

Super-sampling in Practice

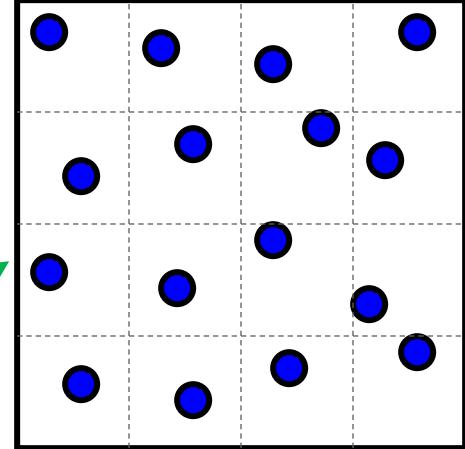
- **Problems with regular supersampling**
 - Expensive: 4-fold to 16-fold effort
 - Non adaptive: Same effort everywhere
 - Too regular: Reduced number of intensity levels
- Should we compute area covered? (Nope-please avoid analytical computation of pixel coverage! that's UGLY!)
- Stochastic supersampling. Implementation? How?
- Let's try uniform statistical sampling patterns instead:



$0 \rightarrow 4/16 \rightarrow 8/16 \rightarrow 12/16 \rightarrow 16/16$

Better, but noisy

4x4 Jittered Super-Sampling

- Cut up entire screen area into square ‘neighborhood’ tiles, as before:
 - Sub-divide each tile into 4x4 ‘sub-tiles’
(one pixel: one tile: 16 sub-tiles)
 - Shoot 16 rays per pixel:
one ray through each sub-tile
 - ‘super-sampling’ == shoot one ray thru each sub-tile’s center-point
 - ‘jittered super-sampling == shoot one ray each sub-tile at a new, randomly-chosen location within the sub-tile, with all sub-tile locations equally probable
- (HINT: see `Math.random()`: http://www.w3schools.com/jsref/jsref_random.asp)
-
- Pixel color == **average** of all 16 ray-colors
 - **4x4 jittered: popular de-facto standard for anti-aliasing:**
~best quality/cost trade-off (e.g. Pixar movies’ final renderings)
 - Try it! 2x2? 3x3? 4x4? 5x5? **How much better-looking?**

Comparison

(slide by Marcus Magnor)

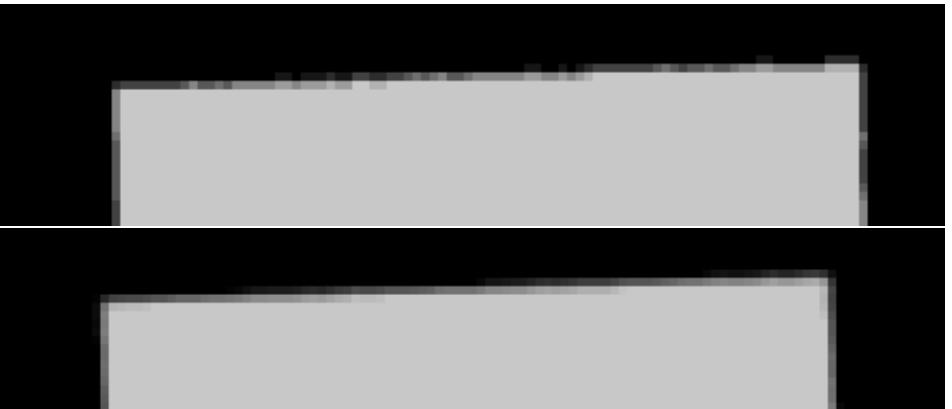
Regular Supersampling, 1x1



Regular Supersampling, 7x7

Jittered Supersampling, 3x3

Jittered Supersampling, 7x7



Beyond Week 1 Goals:

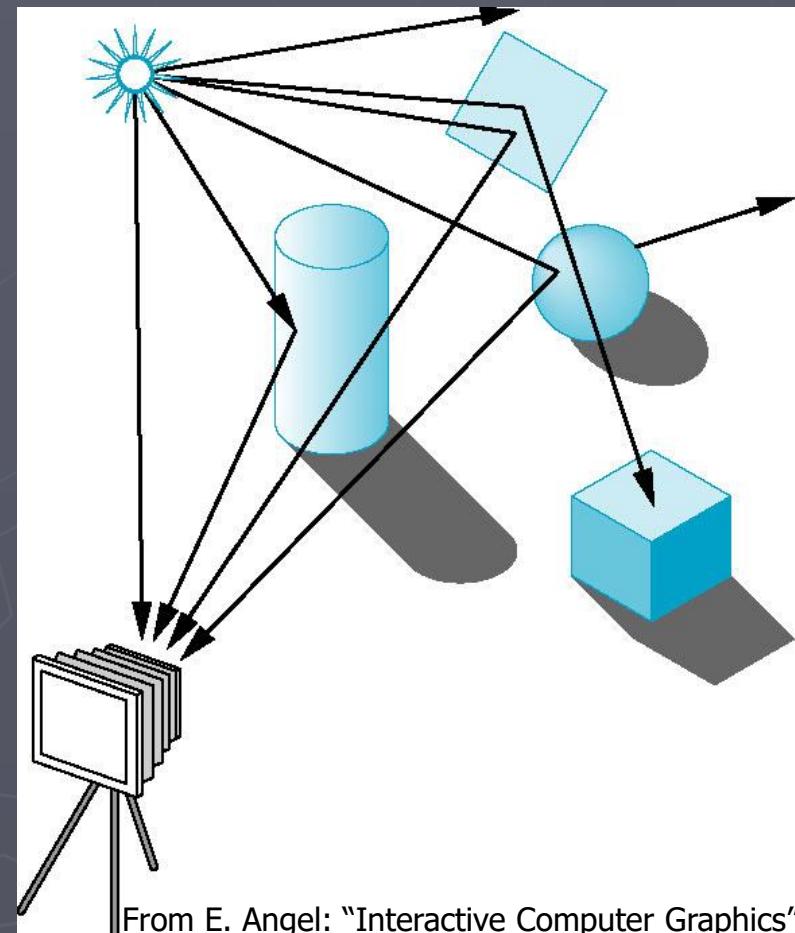
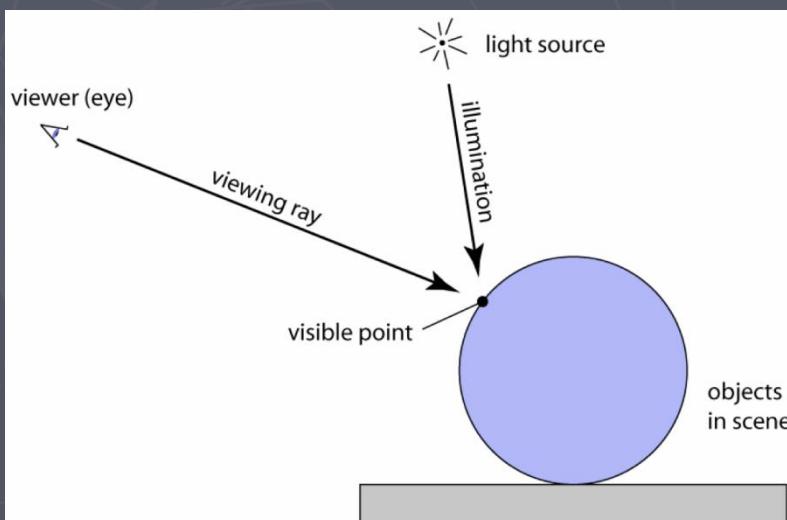
- ▶ Week 1) Ray-trace a 2-colored grid-plane
 - Your own Camera, Geometry, Ray, & Hit classes
 - Solve for t_{hit} to find where ray hits grid-plane
 - Use fixed transform already given:
 - ▶ Camera at origin; grid-plane at $z = z_{\text{val}}$, x-rotated 45°
 - ▶ From computed $(x_{\text{hit}}, y_{\text{hit}})$, find ray color for x,y grid
- ▶ Now make BETTER PICTURES:
***Anti*-Aliasing by Jittered Supersampling**
 - User-Controlled (on/off; more/less; jitter/fixed)
 - Min. Req'd: switch between **none** (1 sample/pix) and **4x4 jittered supersampling**

Return to our Ray Tracer:

- ▶ 'ray' defined as 3D line parameterized by 't':
$$\text{ray}(t) = (\text{origin_point}) + (\text{direction_vector}) \cdot t$$



- ▶ Let 'rays' simulate paths of photons through scene, but trace *BACKWARDS*:



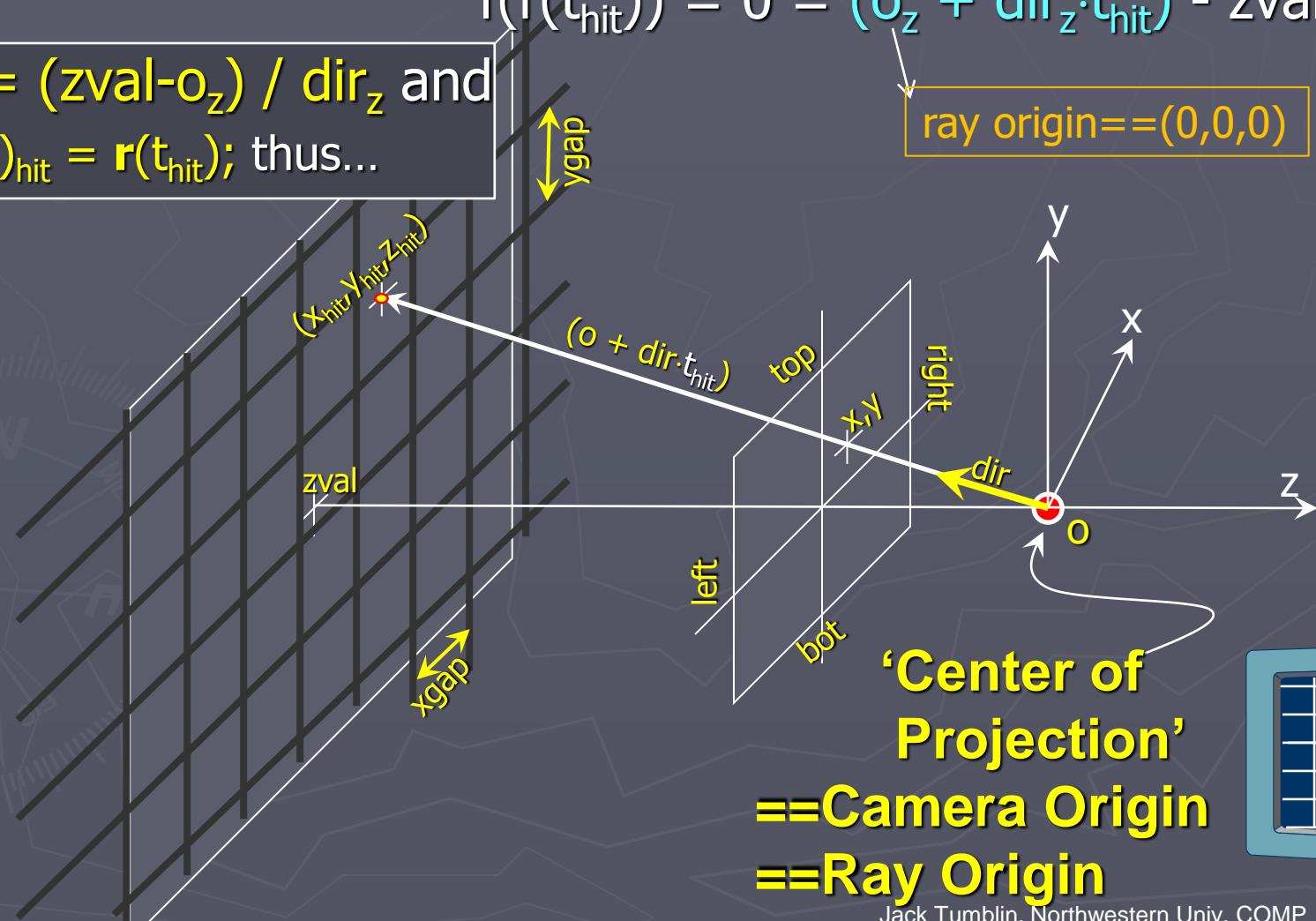
From E. Angel: "Interactive Computer Graphics"

Trace a grid-plane object: 1) Find t_{hit}

--Solve for $t=t_{hit}$ where $f(r(t_{hit})) = 0$: $f(x,y,z) == (z) - zval$

$$f(r(t_{hit})) = 0 = (o_z + \text{dir}_z \cdot t_{hit}) - zval, \text{ thus}$$

$t_{hit} = (zval - o_z) / \text{dir}_z$ and
 $(x,y,z)_{hit} = r(t_{hit})$; thus...



2) From t_{hit} find 3D 'hit' point

--Solve for $t=t_{hit}$ where $f(r(t_{hit})) = 0$: $f(x,y,z) == (z) - zval$

$$f(r(t_{hit})) = 0 = (o_z + \text{dir}_z \cdot t_{hit}) - zval, \text{ thus}$$

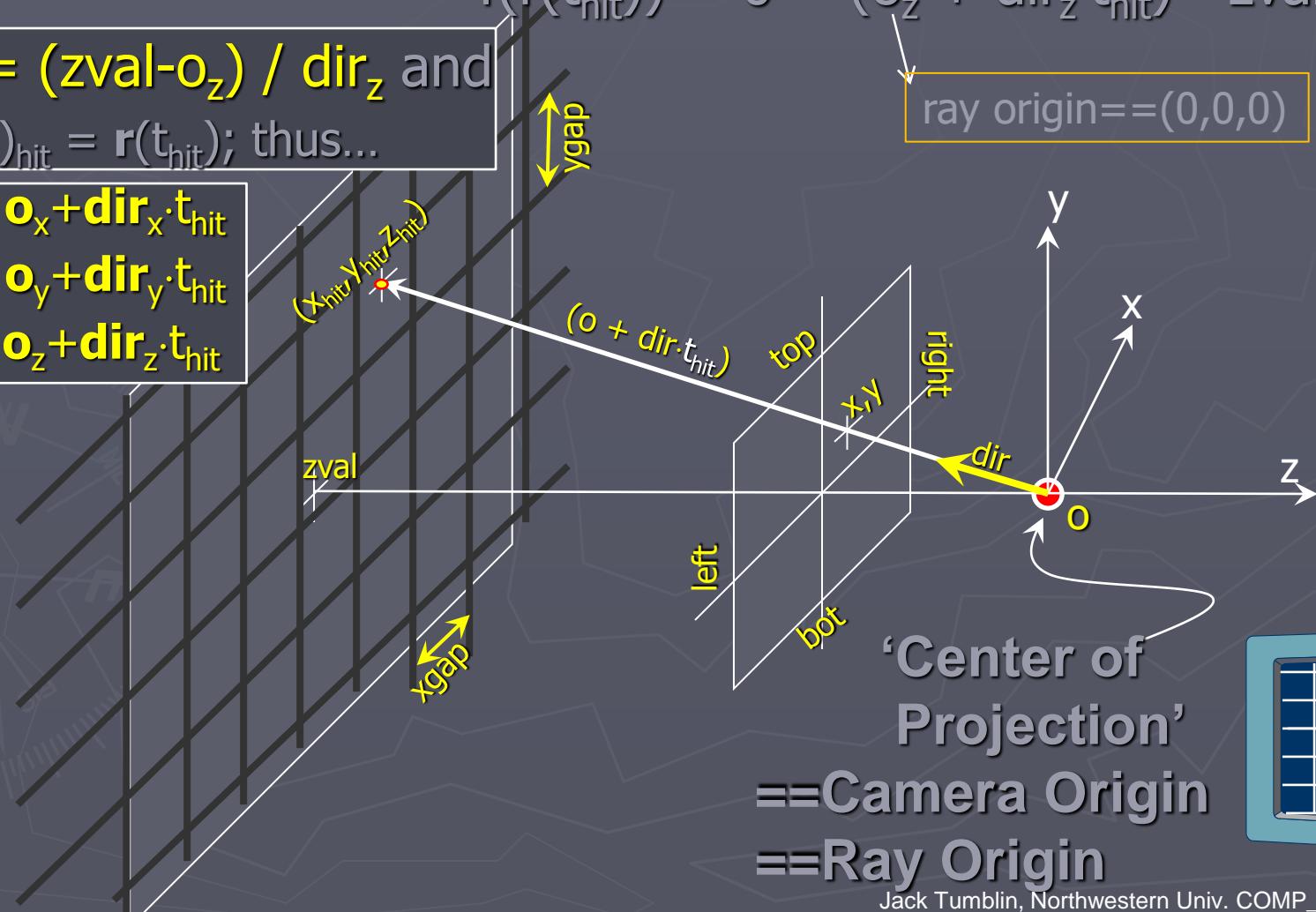
$$t_{hit} = (zval - o_z) / \text{dir}_z \text{ and}$$

$(x,y,z)_{hit} = r(t_{hit})$; thus...

$$x_{hit} = o_x + \text{dir}_x \cdot t_{hit}$$

$$y_{hit} = o_y + \text{dir}_y \cdot t_{hit}$$

$$z_{hit} = o_z + \text{dir}_z \cdot t_{hit}$$



3) From 3D hit point, Find Color

What's the color on the grid-plane
at world-space position (x,y,z)?

- ▶ z component doesn't matter in lineGrid plane
- ▶ find fractional part of $(x/xgap)$; call it 'xfrac'
find fractional part of $(y/ygap)$; call it 'yfrac'
- ▶ if $(xfrac < lineWidth \text{ OR } yfrac < lineWidth)$
then use 'line' color
else use 'gap' color

World-space colors!? ... DANGER!

- ▶ ! YIKES! If we create/define colors and patterns only in 3D world space, then they 'slide around' & 'shift' as the objects move, rotate, and re-scale!
That's not good. What can we do?
- ▶ Need to *describe* models in 'model' space, but need to *ray-trace* them in 'world' space.

Uh-Oh... – do we have to 'transform' all our models, all their colors, and all their materials and all other features to world space?

NO --- here's a MUCH, MUCH better idea...

Ray-Tracing: A Fundamental Change

- ▶ Vertex/Polygon Renderers (WebGL) transform vertex coord values from **model → world** coords (by applying **model** matrix to verts)
- ▶ Ray-Tracing Renderers will instead transform eye ray values (**orig + dir*t**) from **world → model** coords somehow ...
- ▶ THUS: ray-tracing needs to build & use the *inverse* of WebGL's **model** matrix!

RECALL:
ray-tracing camera
creates eye rays
in **world** coords

Transformational Treachery

- ▶ Ray Tracer: DON'T transform the shape;
transform the ray!
- ▶ Week 1: NO TRANSFORMS:
 - Put ray-camera at world origin: simple!
 - Define grid-plane at $z=zVal$;
- ▶ Week 2: TRANSFORMS everywhere:
 - WebGL: (fwd) 'push object out from camera'
 $\text{modelView} \leftarrow \text{Translate}(0,0,-5)$
 - rayTrace: (inverse) 'pull ray into object coords'
in CGeom: $\text{worldRay2model} \leftarrow \text{Translate}(0,0,+5)$

Recall WebGL's Transformations

- ▶ For jointed objects,
we traversed a 'tree of transformations'
 - If we start here:
`mvMatrix.setIdentity(); // wipe its contents: [I]`
 - Each sequential transformation WebGL call:
`mvMatrix.translate(); // apply [T] matrix`
`mvMatrix.rotate(); // apply [R] matrix`
`mvMatrix.scale3d(); // apply [S] matrix`
 - ***PRE-MULTIPLIES*** mvMatrix contents:
[I][T][R][S]
 - Creates new 'drawing axes' from old, ***OR*** (equivalently)...

WebGL vs Ray Transform Matrices

- Each Sequential WebGL transformation call:
`mvMatrix.translate(); // apply [T] matrix`
`mvMatrix.rotate(); // apply [R] matrix`
`mvMatrix.scale(); // apply [S] matrix`
- *PRE-MULTIPLIES* mvMatrix contents

$$v_{\text{new}} = [I][T][R][S]v_{\text{old}}$$

- Changes vertex values (the x,y,z numbers):
'model coord. numbers' $\rightarrow \rightarrow$ 'world coord numbers'

► Ray Tracer Goal: change the rays:

'model coord numbers' $\leftarrow \leftarrow$ 'world coord numbers'

$$\text{ray}_{\text{new}} = [S^{-1}] [R^{-1}] [T^{-1}] \text{ray}_{\text{old}}$$

WebGL vs Ray Transform Matrices

WebGL Goal: change vertex coord values:

- 'model coordinates' $\leftarrow\leftarrow$ 'world coordinates'
- $v_{\text{new}} = [I][T][R][S]v_{\text{old}}$
- careful! Will *PRE-MULTIPLY* by xform matrices!

Ray Tracer Goal: change ray coord values:

- 'model coordinates' $\rightarrow\rightarrow$ 'world coordinates'
- $\text{ray}_{\text{new}} = [S^{-1}] [R^{-1}] [T^{-1}] [I] \text{ray}_{\text{old}}$
- careful! (unlike WebGL) our ray-tracer will
POST-MULTIPLY by these INVERSE matrices!

Ray-Tracer's Transformations

- ▶ **Ray Tracer Goal:** change the rays:
 - 'model coordinates' $\leftarrow\leftarrow$ 'world coordinates'

$$\text{ray}_{\text{new}} = [\mathbf{S}^{-1}] [\mathbf{R}^{-1}] [\mathbf{T}^{-1}] \text{ray}_{\text{old}}$$
- ▶ Method: mimic WebGL/textbook API; same args
 - `CGeom.raySetIdentity();` `CGeom.rayIdentity();`
 - `CGeom.raySetTranslate();` `CGeom.rayTranslate();`
 - `CGeom.raySetRotate();` `CGeom.rayRotate();`
 - `CGeom.raySetScale();` `CGeom.rayScale()`
 - Where's the matrix? Mat4 object: `CGeom.world2model`
- ▶ *BUT* (unlike WebGL) we must
POST-multiply by INVERSE functions!

Ray-Tracer's Transformations

- ▶ *BUT* (unlike WebGL) xform fcns must POST-multiply by INVERSE matrices.
- ▶ Sounds difficult, but it's quite easy:
 - **CGeom.rayIdentity()**
identity matrix is its own inverse: NO CHANGE
 - **CGeom.rayTranslate()**
inverse of $\text{translate}(x,y,z)$ == $\text{translate}(-x,-y,-z)$
 - **CGeom.rayRotate()**
inverse of $\text{rotate}(\theta)$ == $\text{rotate}(-\theta)$
 - **CGeom.rayScale()**
inverse of $\text{scale}(sx,sy,sz)$ ==
 $\text{scale}(1/sx, 1/sy, 1/sz)$

Goals for Week 1 was:

- ▶ 1) Ray-trace a 2-colored grid-plane
 - Your own Camera, Geometry, Ray, & Hit classes
 - Solve for t_{hit} to find where ray hits grid-plane
 - Do everything simply, in world space:
 - ▶ Camera at origin; grid-plane at $z = zval$, x-rotated 45°
 - ▶ From computed (x_{hit}, y_{hit}) , find ray color for x,y grid
- ▶ 2) Rotate the camera 90° to see horizon:
 - Create your own `CCamera.setLookAt()` function to position & aim the camera in world coords.
 - Investigate – why does the horizon look bad?!?!

Goals for Week 2:

- ▶ ***Anti*-Aliasing by Jittered Supersampling**
 - User-Controlled (on/off; more/less; jitter/fixed)
 - Min. Req'd: switch between **none** (1 sample/pix) and **4x4 jittered supersampling**
- ▶ **Add Ray-tracing Transform Functions**
 - Build on anti-aliased grid-drawing program
 - Write, test CGeom member fcns:
`rayLoadIdentity()`, `rayTranslate()`,
`rayRotate()`, `rayScale()`. Test `rayLookAt()` too!
- ▶ **Mouse/Keybd-Adjusted WebGL transform**
 - mouse/keybd control of translate, rotate, scale
 - Perfectly matched WebGL & Ray-Traced result

Goals for Week 3:

- ▶ 1) Ray-Trace Your First Shapes: disk, sphere,
 - Move camera away from origin to $z = +5$
 - Make unit sphere, at origin, in model coords
 - Color? Try $R, G, B = x_{\text{hit}}, y_{\text{hit}}, z_{\text{hit}}$ in model coords
(→the color helps your debugging!)
- ▶ 2) Make your WebGL previews: disk, sphere,
 - 'basicShapes.js' can match ray-traced sphere...
- ▶ 3) Disk, Sphere Shadows on your grid-plane.
 - 'Light src' at $(0,0,3)$ in grid-plane model coords
 - Shadow detected? Scale grid-plane color by 0.3

Week 4 (Final) Goals

- ▶ 1) Make World-Space Surface Normals →
WebGL-like Phong Lighting
 - Transform normals? → See lecture notes ...
 - CMatl objects: ambient, diffuse, specular, & shinyness members
 - Move lights with Mouse, Keyboard? More transforms!
- ▶ 2) Recursion: Show multiple reflections among objects;
 - 1 or more Grid-plane objects, 2 or more Spheres (try 3: )
 - ORGANIZE: 
- ▶ 3) At least one '3D Solid Texture' or 'Procedural Material'
 - CRayHit: add member(s) for model-space hit-point
 - CMatl: & selectable model-space color fcns: R(x,y,z) G(x,y,z) B(x,y,z)
 - Functions for Concentric Shells, Line→Grid→Cage, Checkerboards & other 'easy' textures? see FS Hill, Chap 14.8, pp. 770-773

Further Goals:

The web is FULL of good ideas, and...

See:

-- Assignment Sheet
-- Grading Sheet

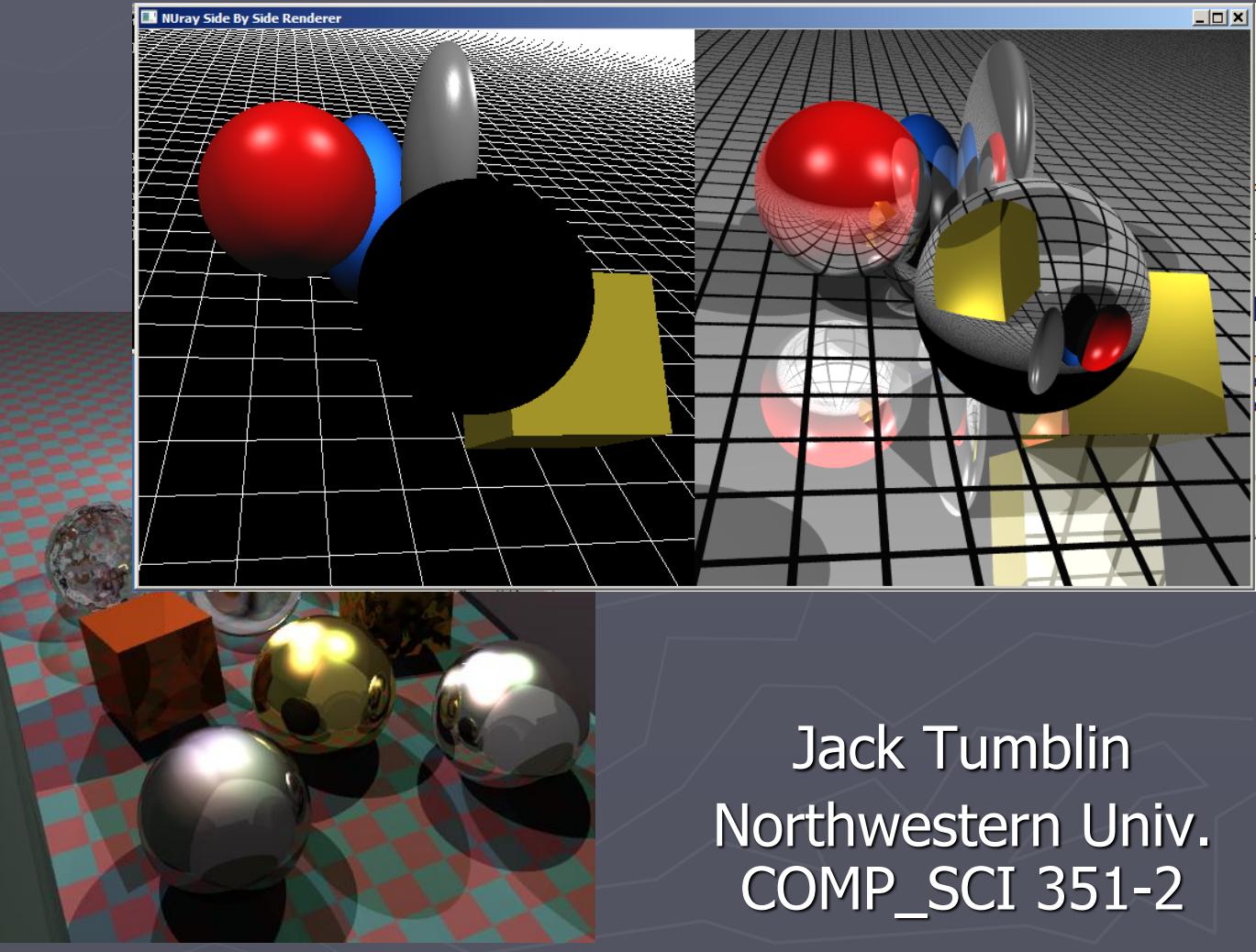
END



END

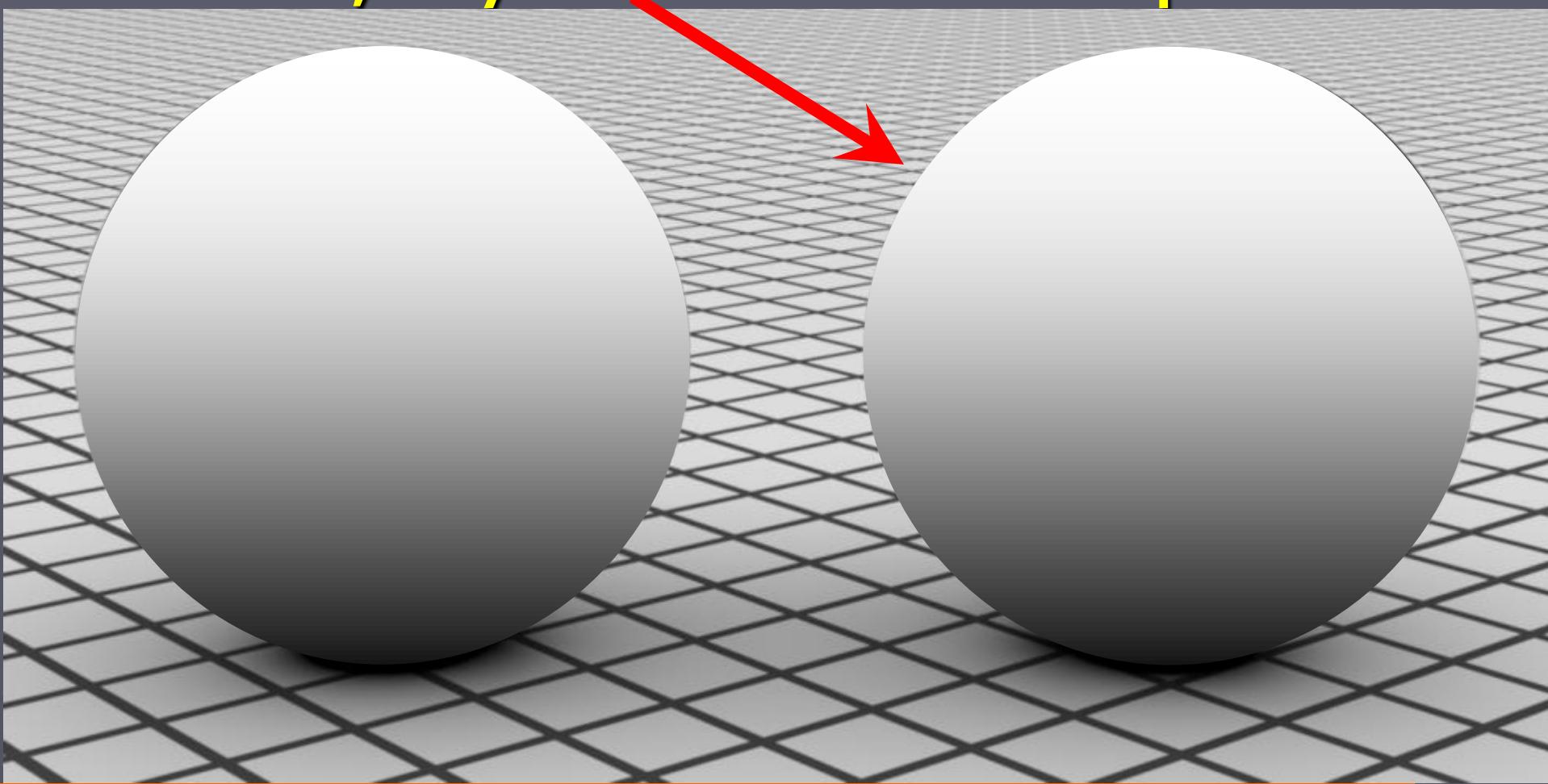


Ray Tracing D: Spheres, Shadows & Shading



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

Next, try this : a basic sphere



Matt Pharr, Greg Humphries: Physically Based Rendering Techniques (**PBRT**)
highly recommended beautiful in-depth book: <https://www.pbrt.org/gallery.html>

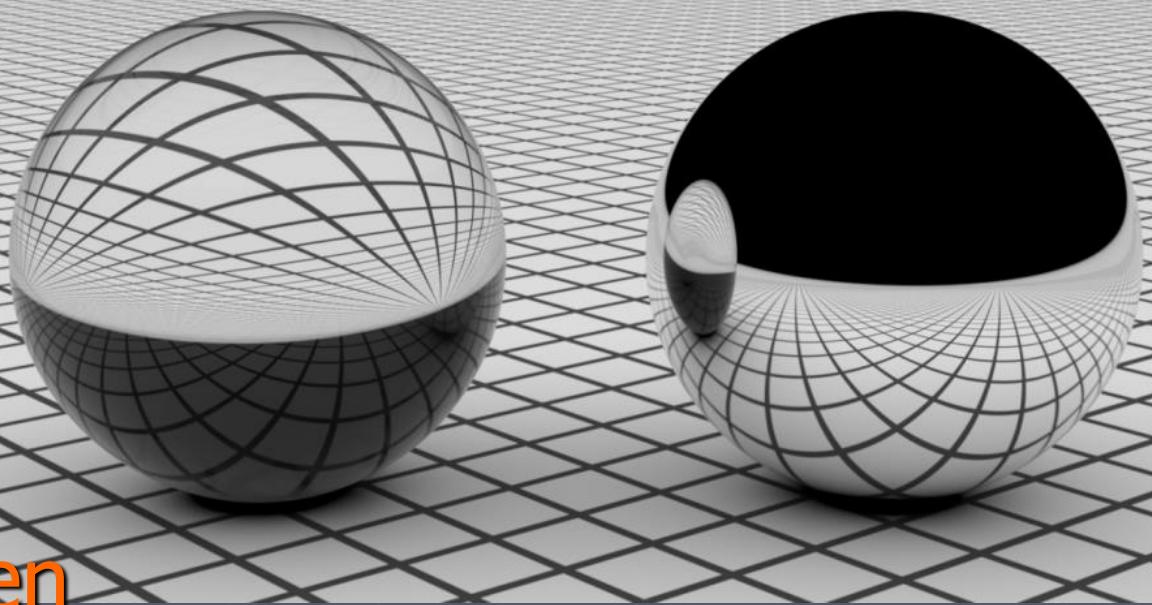
Our Plan:

Write code for:

- ▶ Camera, then
- ▶ Plane, then
- ▶ Antialiasing, then
- ▶ Transform, then
- ▶ Basic Disk and Sphere, then Shadow, then
- ▶ Phong Lighting,
- ▶ Reflection, then Transparency, then

Further Step-by-Step Refinements:

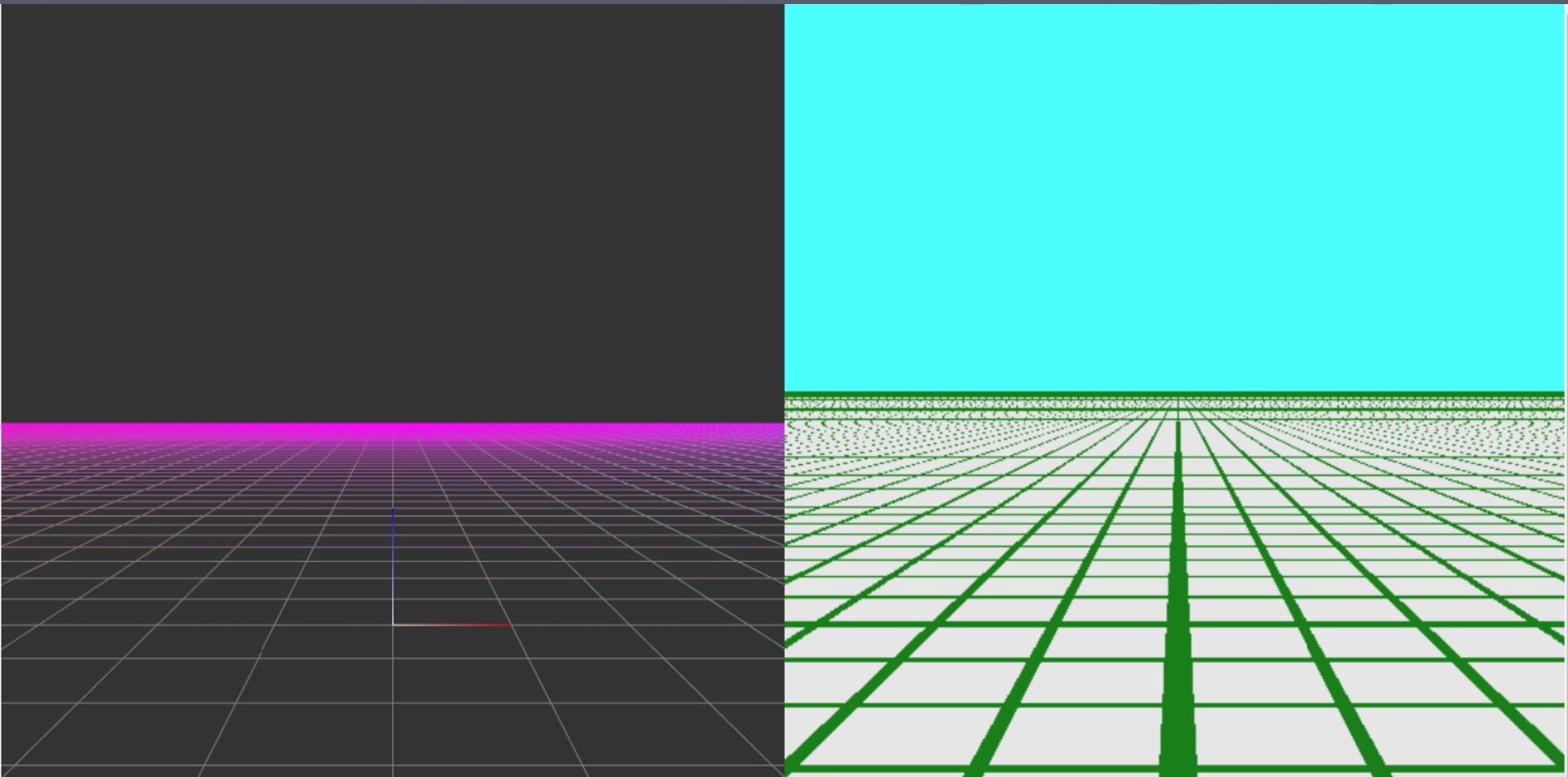
Lighting → Materials → Texture → CSG, etc. ...



*** **MAKE PICTURES AT EVERY STEP** ***

What we have: Gnd Plane

What's the **simplest 1st shape** we can add to it?



RECALL: Ray Tracing Implicit Surfaces

Shape specified by one scalar function $F(x,y,z)$:

$F() < 0$ for all (x,y,z) points **INSIDE**,

$F() > 0$ for all (x,y,z) points **OUTSIDE**, and

$F() = 0$ for all (x,y,z) points **ON** the surface.

Implicit functions for

arbitrary **plane**, distance d from origin:

$$F(x, y, z) = ax + by + cz + d = \mathbf{n} \cdot \mathbf{x} + d$$

unit-radius **sphere** at origin:

$$F(x, y, z) = x^2 + y^2 + z^2 - 1 = \mathbf{x} \cdot \mathbf{x} - 1$$

unit-radius **cylinder** along z axis:

$$F(x, y, z) = x^2 + y^2 - 1 \quad 0 < z < 1$$

Make your own! **HINT:** Search “Jules Bloomenthal”,
“Superquadrics”, “Implicit Surfaces”, “Metaballs”, etc ...

Ray Tracing Implicit Surfaces

Implicit Surface == set of all points where

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

Ray tracing == $\text{ray}(t) = \text{orig} + t \cdot \text{dir}$
or just $\mathbf{r}(t) = \mathbf{o} + t \cdot \mathbf{d}$

Here's a method for Ray-Object Intersections:
Plug $\mathbf{r}(t)$ into implicit fcn and solve for t_{hit} :

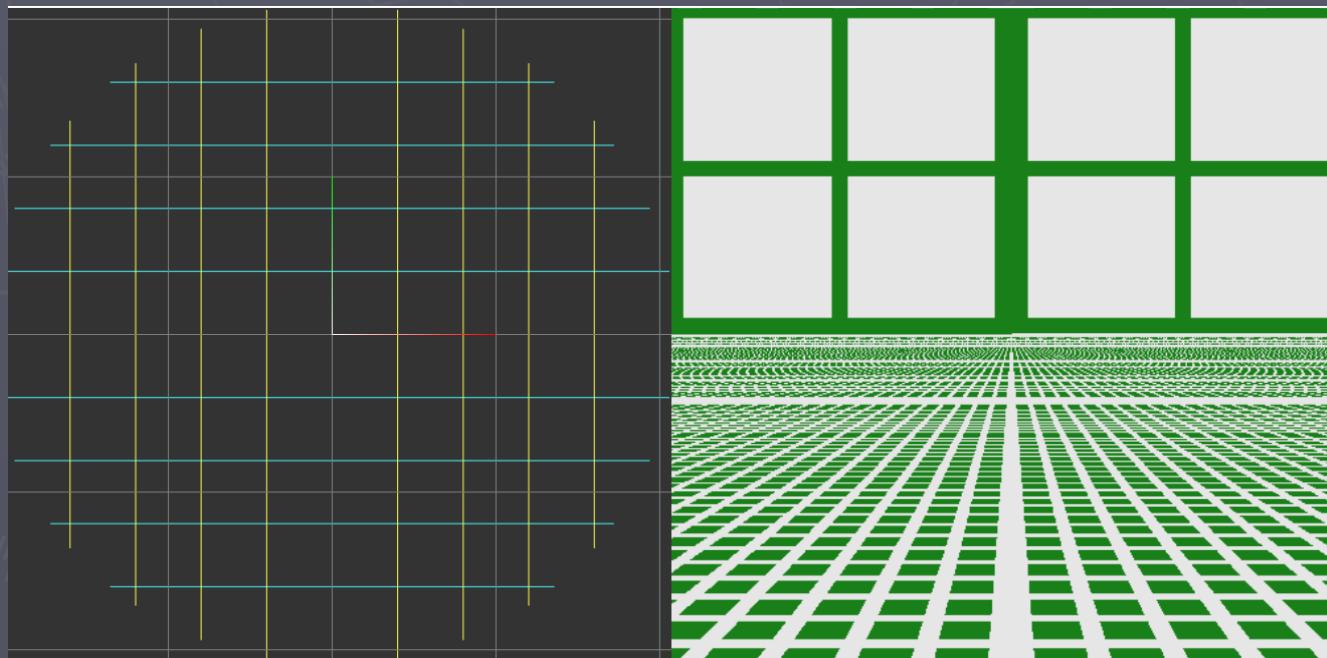
$$f(r_x(t_{\text{hit}}), r_y(t_{\text{hit}}), r_z(t_{\text{hit}})) = 0$$

Simplest 1st Shape? Try **WALL:**

Spheres? Ugh. Tricky, error prone, and
we can't see their orientation on-screen.

***AHA! As ground-plane tracing *already works*, let's try...

WALL: Ray-trace a 2nd ground-plane
that we ROTATED by 90°

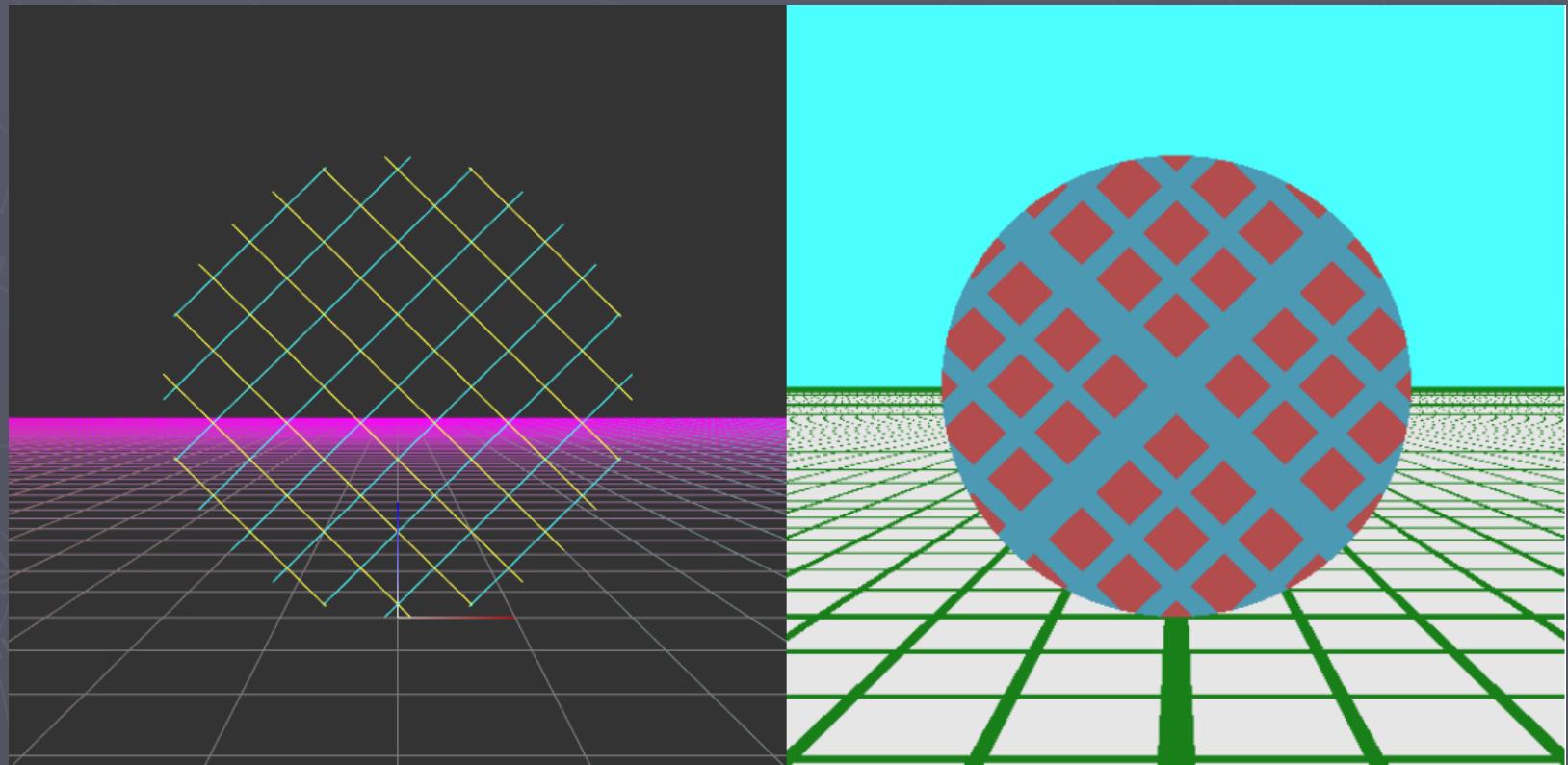


Hmm...
Awkward;
Confusing...

Simplest 1st Shape? Try **DISK**:

AHA! Let's limit the size of the wall...

- **DISK:** same as wall, but limit ray 'hits' to
 $x^2+y^2 < \text{CGeom.diskRadius2}$

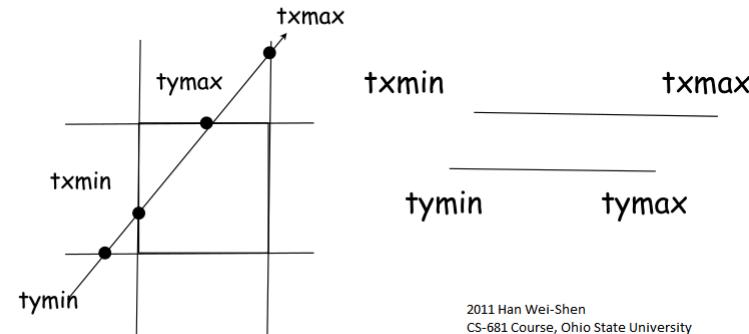


Try a CUBE: How?

► Simple-as-possible shape; NO parameters!

- Centered at origin, and
- Corners at $(\pm 1, \pm 1, \pm 1)$
(resembles CVV!)
- Use Transforms to set position, size, rotation...
- GIVEN this 2D ray/box test →
EXTEND to 3D!

Assuming the ray hits the box boundary lines at intervals $[txmin, txmax]$, $[tymin, tymax]$, the ray hits the box if and only if the intersection of the two intervals is not empty



2011 Han Wei-Shen
CS-681 Course, Ohio State University

► Cube is the intersection of 6 planes (or 3 slabs)

- How did we Ray-trace the ground-plane?
- What conditionals do we need to add?
- How can we avoid divide-by-zero errors?
- How can we make code efficient? (fewer costly conditionals)

Try a SPHERE: How? (3 answers)

- ▶ Naïve ray tracers skip transforms (bad idea):
 - More params: center **c**, squared radius **r²**, etc.
 - For some very clear (but naïve) summaries:
<http://www.lighthouse3d.com/tutorials/mathematics/ray-sphere-intersection/>
<https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/ray-tracing-practical-example>
- ▶ Our Ray tracer: sphere has **radius==1** centered at origin in model space
 - Parameters? **NONE!**
 - transform ray: **worldRay2model** matrix
 - Find **t** values (if any) where ray pierces sphere
(? How will you indicate 'no hit' in CHit class?)

Sphere I: (Naïve) Quadratic Solver

- ▶ Ray $\mathbf{x}(t)$ leaves origin point \mathbf{o} in direction \mathbf{d} ; at time t ($t > 0$) ray ends at $\mathbf{x}(t) = \mathbf{o} + t * \mathbf{d}$.
- ▶ At what time t_{hit} does ray hit a sphere centered at point \mathbf{c} with radius r ?
- ▶ Implicit Sphere: every point where $F(\mathbf{x})=0$
$$F(\mathbf{x}) = (\mathbf{x}-\mathbf{c}) \cdot (\mathbf{x}-\mathbf{c}) - r^2 = 0$$
(squared distance from center c matches r^2)
- ▶ Substitute ray $\mathbf{x}(t)$, simplify:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) = r^2$$

Sphere I: (Naïve) Quadratic Solver

- ▶ Expand this: $(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) = r^2$
- ▶ Put in quadratic form: $At^2 + Bt + C = 0$, and solve:

$$A = \mathbf{d} \cdot \mathbf{d},$$

$$B = 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d},$$

$$C = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2$$

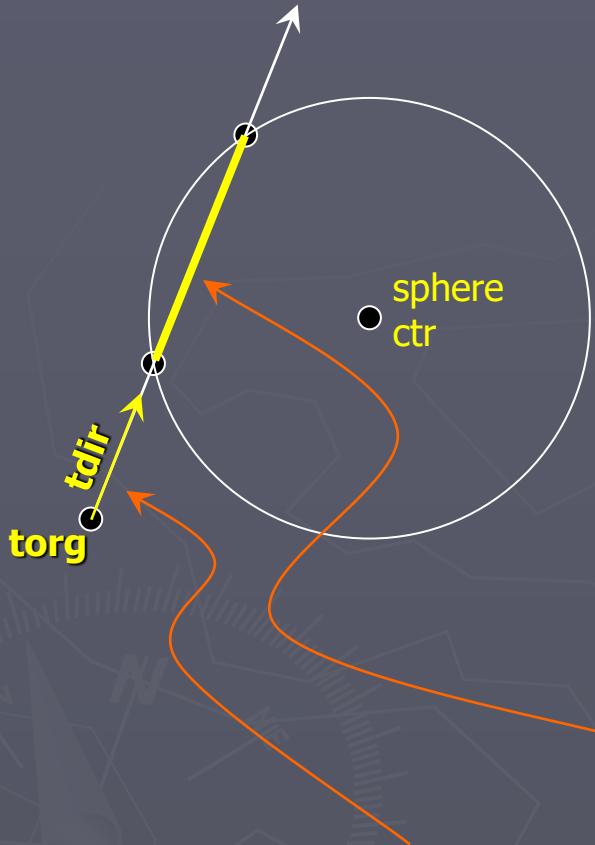
$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

- ▶ Standard Quadratic Solution:
if discriminant $D = (B^2 - 4AC) < 0$, $\rightarrow \rightarrow$ *NO* hits! --!MISS!
else ($D \geq 0$), so hit points are $t = (-B \pm \sqrt{D})/2A$

- ▶ **SLOW!** – there are better, MUCH faster methods!

NEXT: a very fast, popular, more ‘geometric’ method:
it also tells you if your ray starts within the sphere;
(we will need that!)

Sphere II: Half-Chord Solver



HARDER, BETTER, FASTER (, STRONGER ... Daft Punk)

'Half-Chord' or '2-triangle' method:

Key Ideas (1):

- ▶ Any 3D line will either:
 - MISS the sphere, or
 - PIERCE the sphere at 2 points
- ▶ 'Chord' == line segment inside

Careful! Use **model space** ray:

torg == *transformed* origin pt.

tdir == *transformed* direction vector

**FASTER! CLEVER!
LESS AMBIGUOUS!**
(but complex; less obvious)

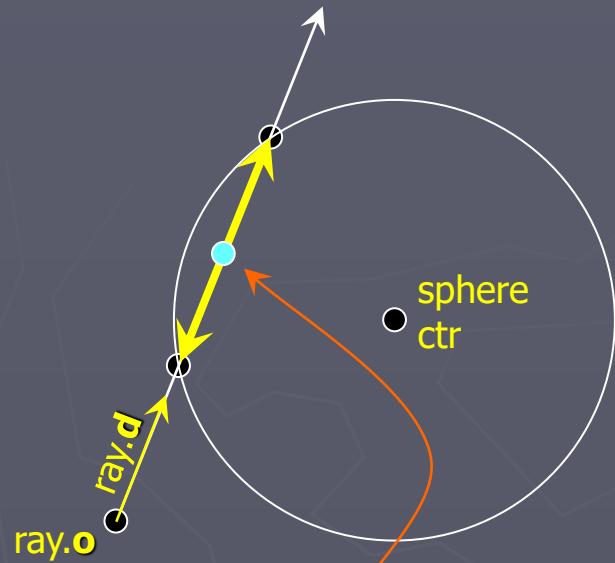
Sphere II: Half-Chord Solver

HARDER, BETTER, FASTER (, STRONGER ... Daft Punk)

'Half-Chord' or '2-triangle' method:

Key Ideas (2):

- ▶ All 3D lines either
 - MISS the sphere, or
 - PIERCE the sphere at 2 points
- ▶ 'Chord' == line segment inside
- ▶ Find Chord mid-point, then move
- ▶ +/- 'Half-Chord' length along ray



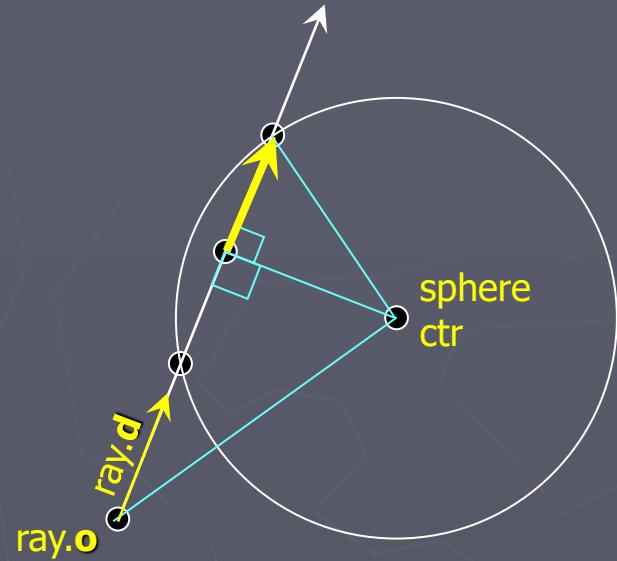
Sphere II: Half-Chord Solver

FASTER, SIMPLER, BETTER:
'Half-Chord' or '2-triangle' method:

Key Ideas (2):

- ▶ All 3D lines either
 - MISS the sphere, or
 - PIERCE the sphere at 2 points
- ▶ 'Chord' == line segment inside
- ▶ Find Chord mid-point, then move
- ▶ +/- 'Half-Chord' length along ray

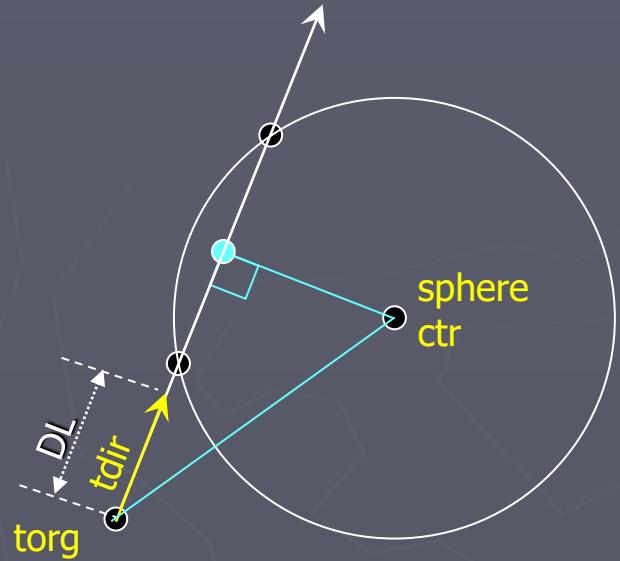
How? Use Pythagorean Theorem
on 2 right triangles ...



$$a^2 + b^2 = c^2$$

Sphere II: Half-Chord Solver

Step 1: Define Transformed Ray

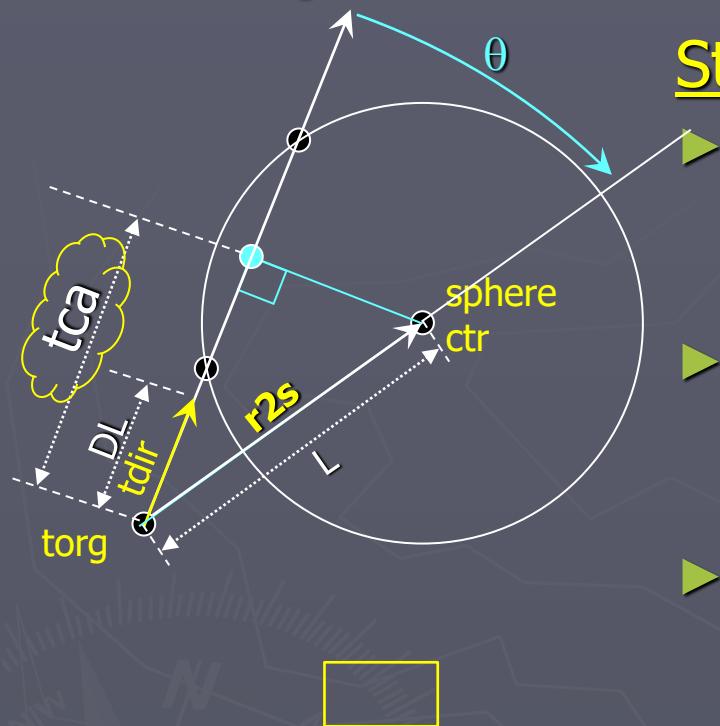


- ▶ Transform world-space ray to model-space ray:
 $torg = [\text{worldRay2Model}] [\text{worldRay.origin}]$
 $tdir = [\text{worldRay2Model}] [\text{worldRay.dir}]$
- ▶ **No need** to normalize $tdir$! Instead,

Use scalar DL to denote length of $tdir$ vector, and we'll write DL^2 as $DL2$.
CAREFUL! don't compute DL yet!

- ▶ Recall: point on ray at ray-time t is:
 $\text{ray}(t) = torg + t*tdir$

Sphere II: Half-Chord Solver



Careful!

Ray.o is a point: w=1;
r2s is a vector: w=0;

Step 2: Test 1st Triangle

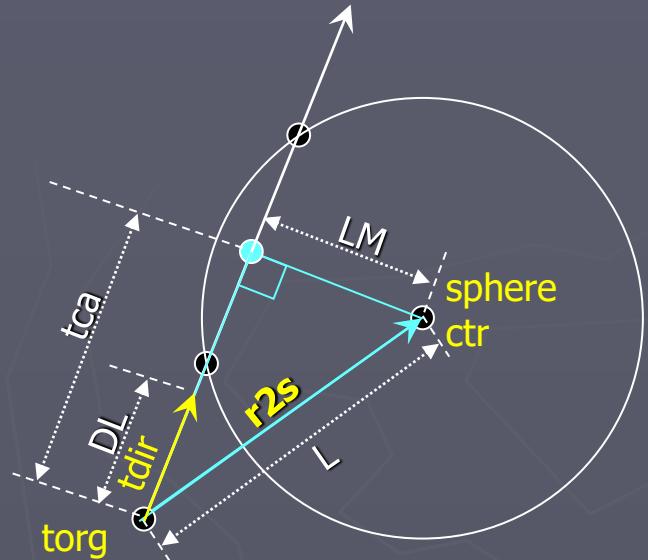
- ▶ Make $r2s$: ray-to-sphere-ctr vector
$$r2s = \text{ctr} - \text{torg} = 0 - \text{torg}$$

- ▶ Find L² (scalar): ($r2s$ length)²
$$L^2 = L^2 = (r2s \cdot r2s);$$
- ▶ Find Mid-point distance: dot-product
$$\begin{aligned} tdir \cdot r2s &= \|tdir\| * (\|r2s\| * \cos \theta) \\ &= DL * \left(\frac{tca}{DL} \right) \\ tdir \cdot r2s &= tcaS, \text{ the } \underline{\text{scaled}} \text{ tca.} \end{aligned}$$

IF ($L^2 > \text{radius}^2$) THEN:

- ▶ ray begins OUTSIDE the sphere. good!
- ▶ IF ($tcaS < 0$) THEN *** MISS! ***
sphere is **behind** the ray origin !

Sphere II: Half-Chord Solver



$$\text{ray}(t) = torg + t * tdir$$

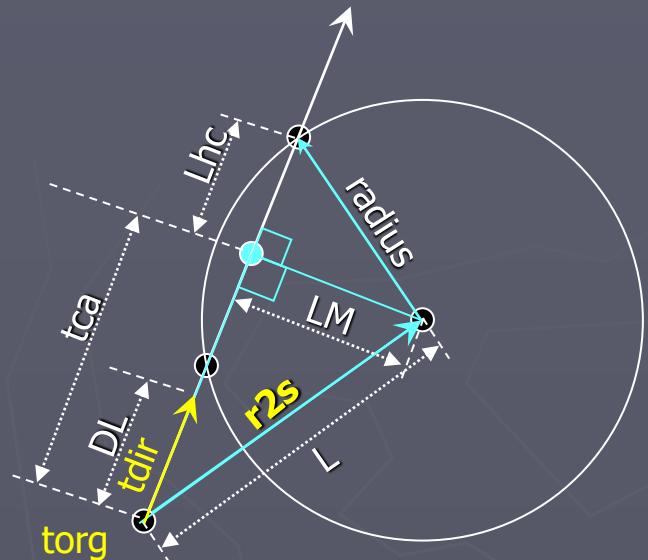
$$a^2 + b^2 = c^2$$



Step 3: Measure 1st Triangle (cont'd)

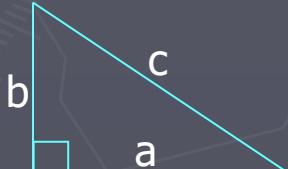
- ▶ IF ($L^2 > \text{radius}^2$) THEN:
 - ray begins OUTSIDE the sphere
 - IF ($tcaS < 0$) THEN ***!MISS!***
sphere is behind the ray origin!
- ▶ Find DL^2 : = tdir vector length²
$$DL^2 = \boxed{DL^2 = (\text{tdir} \cdot \text{tdir})}$$
- ▶ Find tca^2 (*without* any scaling):
 $(tcaS)^2 = (tca \cdot DL)^2 = tca^2 \cdot DL^2$, so
$$tca^2 = \boxed{tcaS \cdot tcaS / DL^2 = tca2}$$
- ▶ Find LM^2 By Pythagorean Theorem:
 $LM^2 + tca^2 = L^2$, so
 $LM^2 = L^2 - tca^2$, or $\boxed{LM^2 = L^2 - tca2}$

Sphere II: Half-Chord Solver



$$\text{ray}(t) = \text{torg} + t * \text{tdir}$$

$$a^2 + b^2 = c^2$$



Step 4: Measure 2nd Triangle

- ▶ **IF($LM^2 > \text{radius}^2$) → ***MISS!*****
(shortest distance² from ray to sphere center is $> \text{radius}^2$)
- ▶ **IF (is shadow ray) → ***STOP! *****
(the ray hit the sphere;
but we don't care where! Stop!)
- ▶ **Find half-chord length² $L2hc$ by Pythagorean Theorem:**
 $\text{Lhc}^2 + \text{LM}^2 = \text{radius}^2$, so
 $\text{Lhc}^2 = \text{radius}^2 - \text{LM}^2$

Sphere II: Half-Chord Solver

Step 5: Measure ray using 2nd Triangle

Find ray-time t at ray/sphere intersection points:

Ray-length at ANY ray-time t: $RL = t * DL$

Ray-length at chord ends: $RL = tca \pm Lhc$
combine, solve: $t = (tca \pm Lhc) / DL$

Recall these earlier results:

$tcaS = tca * DL$: rearrange to get:

$tca = tcaS / DL$, where

$DL = \sqrt{DL^2}$ and recall that

$Lhc = \sqrt{Lhc^2}$.

Now substitute, and divide again by DL:

$$\begin{aligned} t &= (tca \pm Lhc) / DL \\ &= ((tcaS / DL) \pm \sqrt{Lhc^2}) / DL \\ &= (tcaS / DL^2) \pm (\sqrt{Lhc^2} / \sqrt{DL^2}) \end{aligned}$$

IF ($L^2 > radius^2$) THEN

ray must pierce sphere at **two** hit-points:

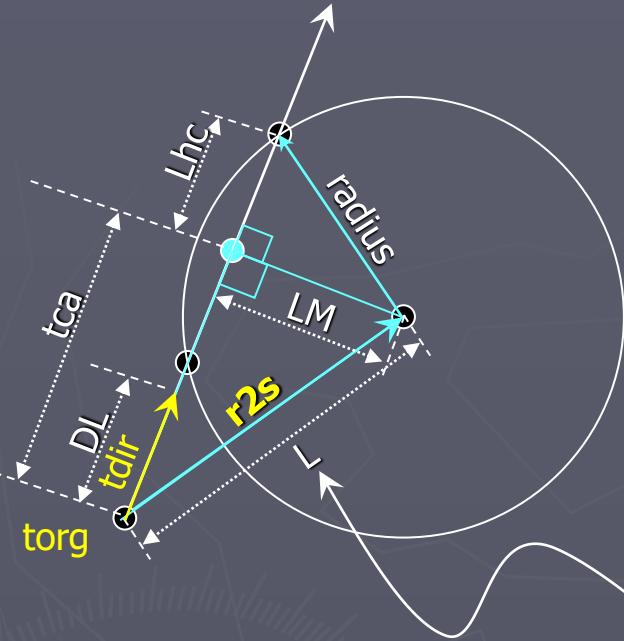
$$t_0, t_1 = tcaS / DL^2 \pm \sqrt{Lhc^2 / DL^2}$$

ELSE ray began INSIDE the sphere;

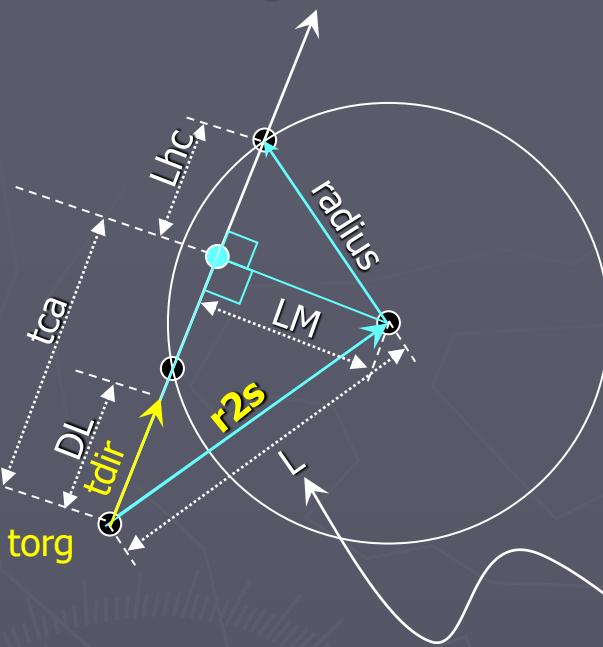
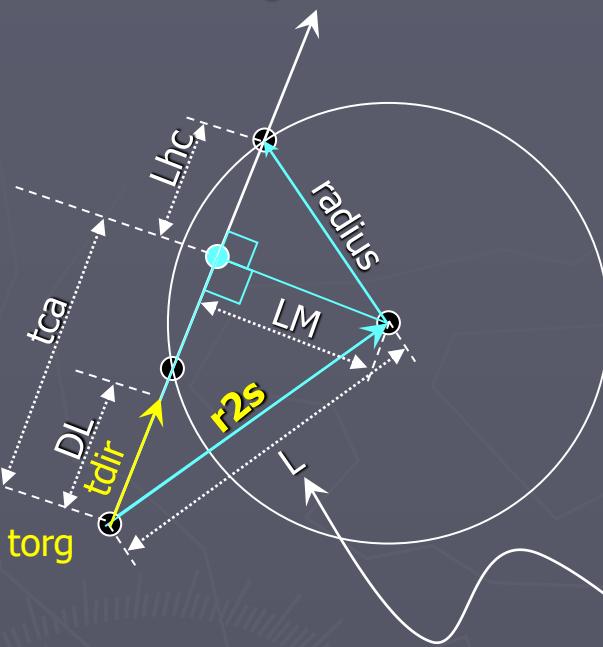
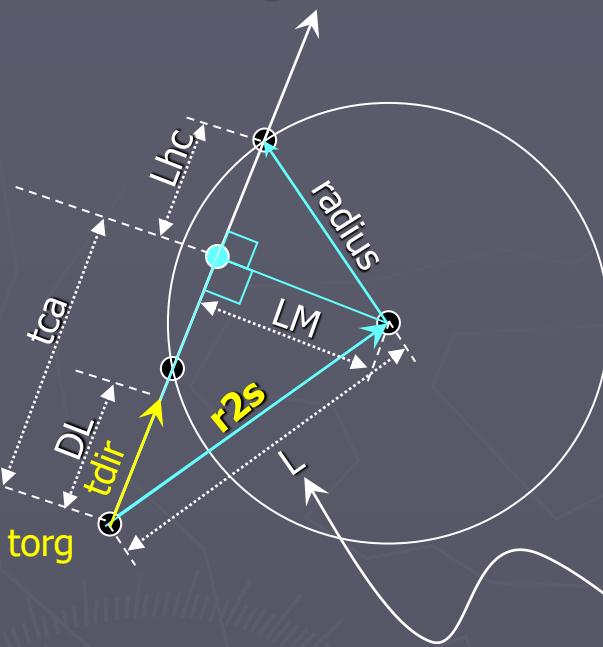
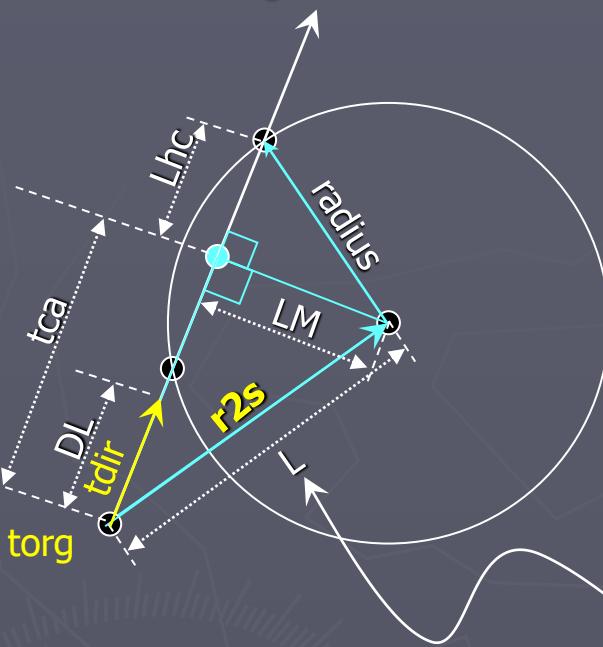
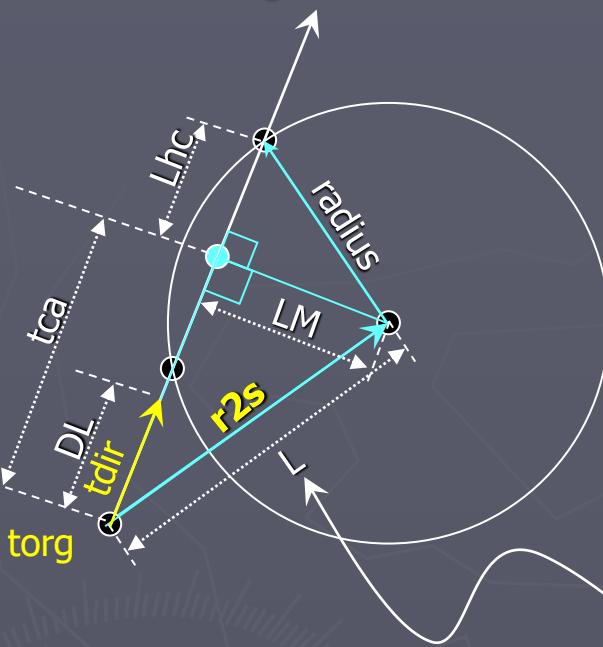
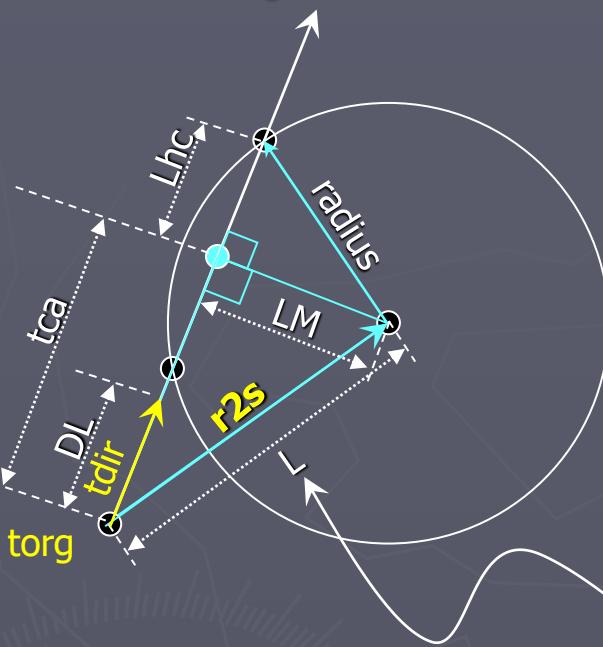
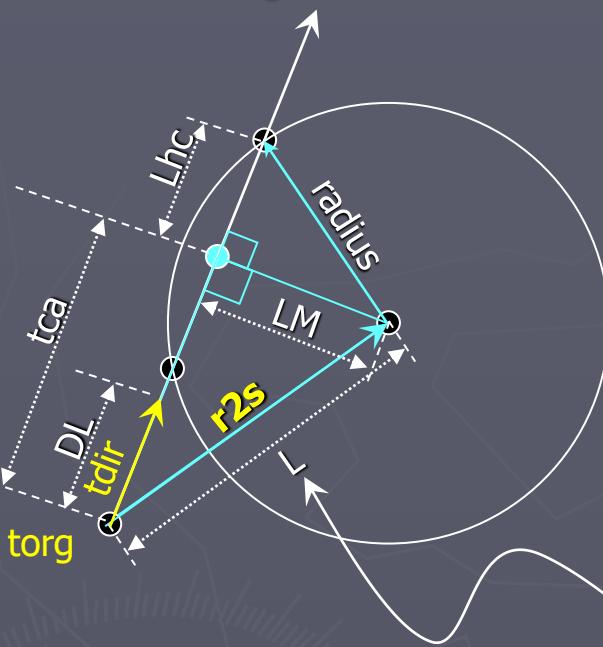
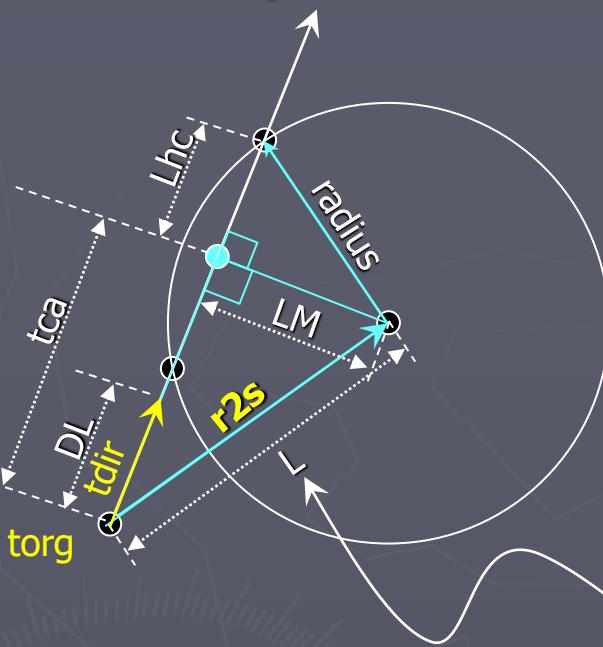
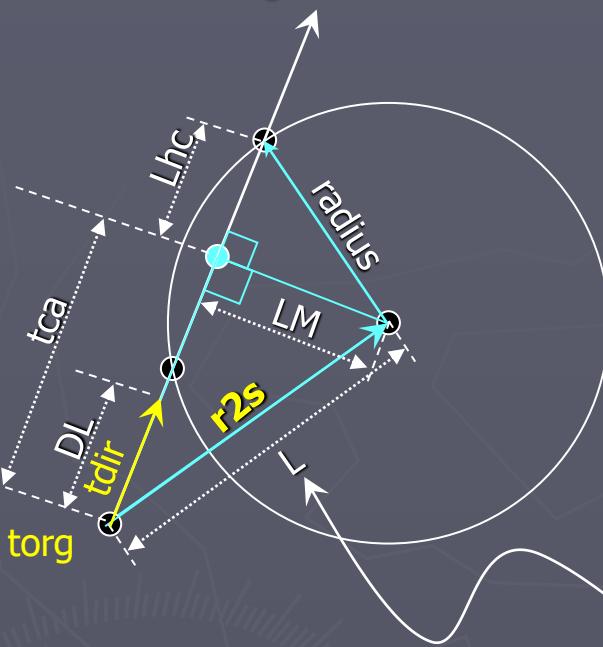
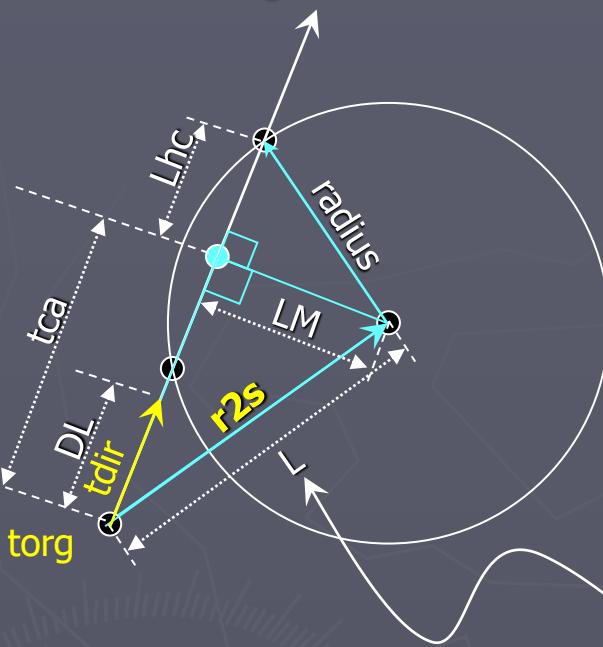
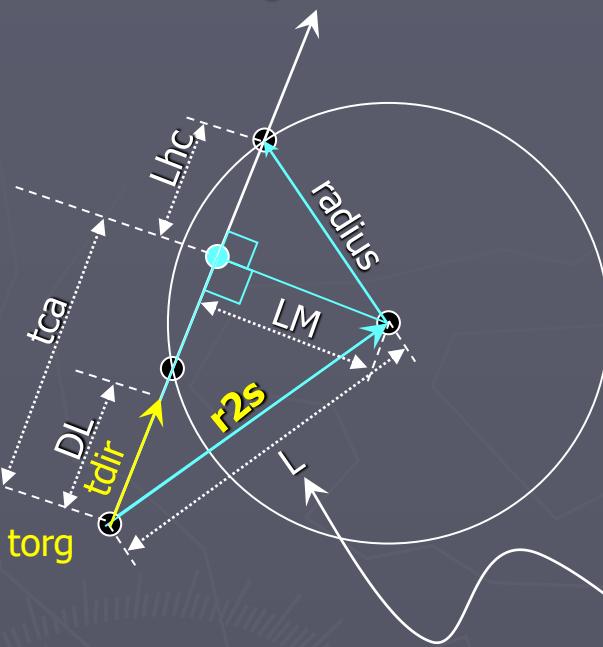
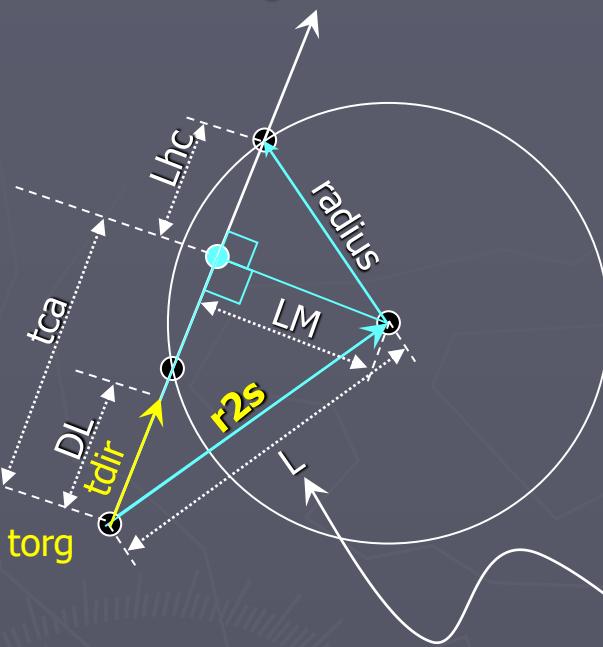
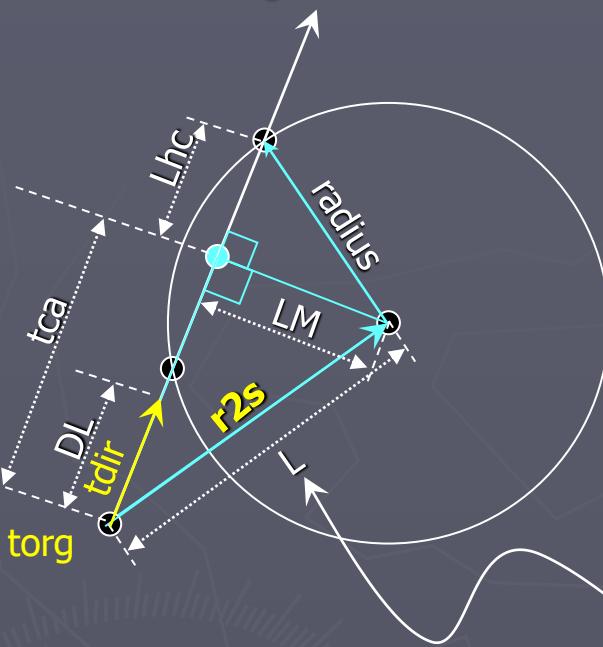
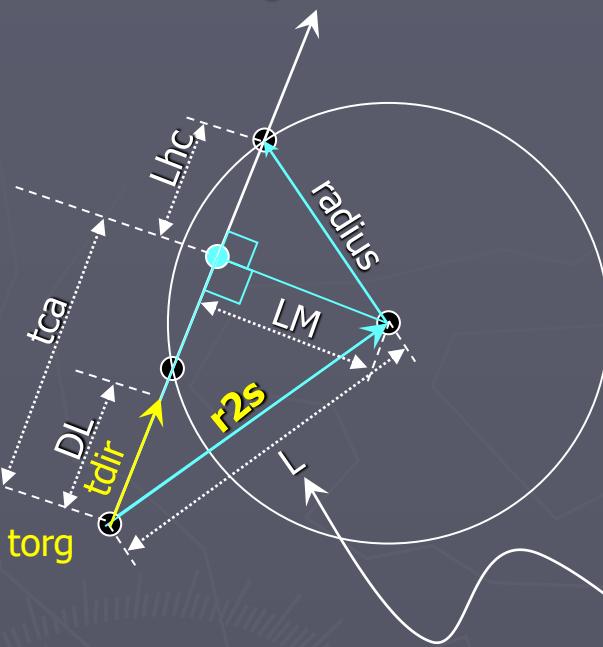
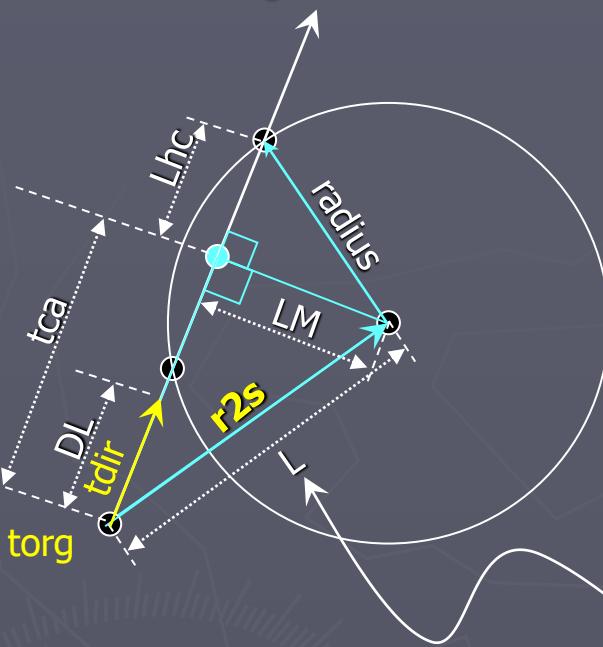
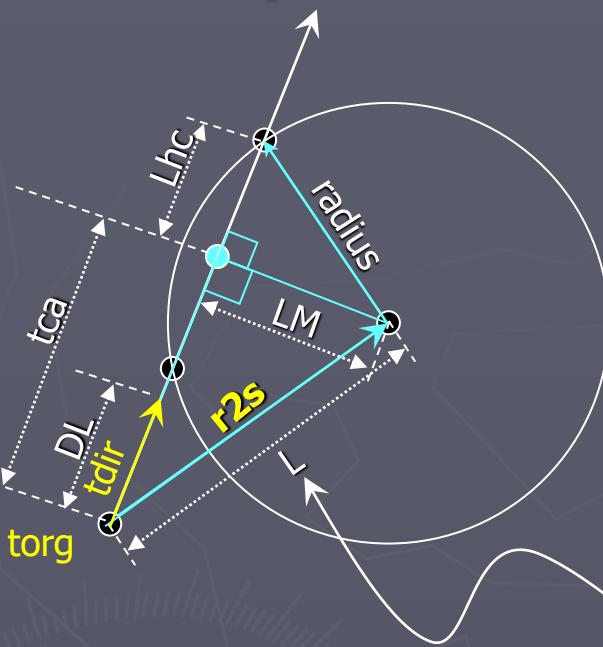
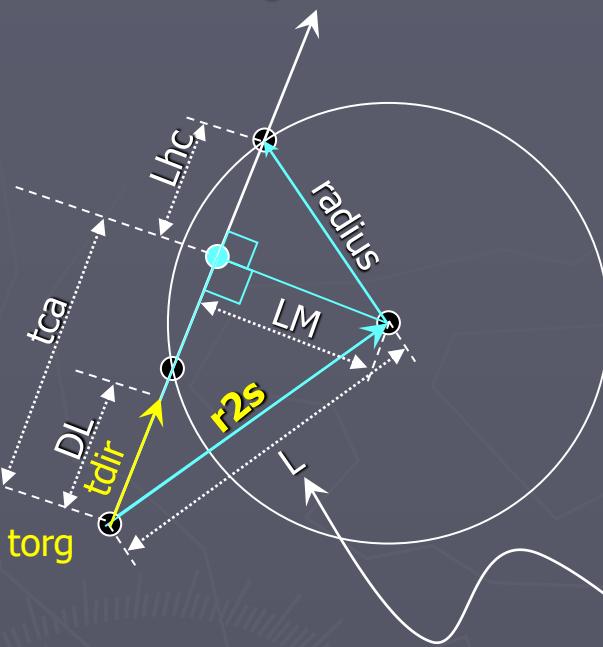
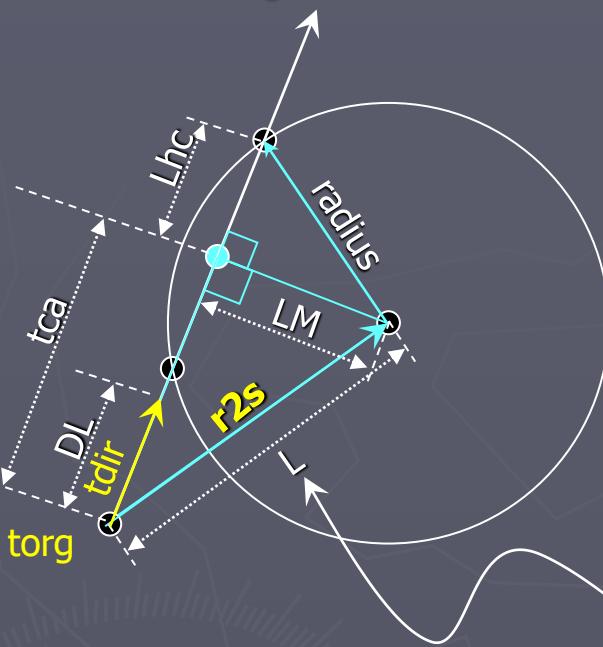
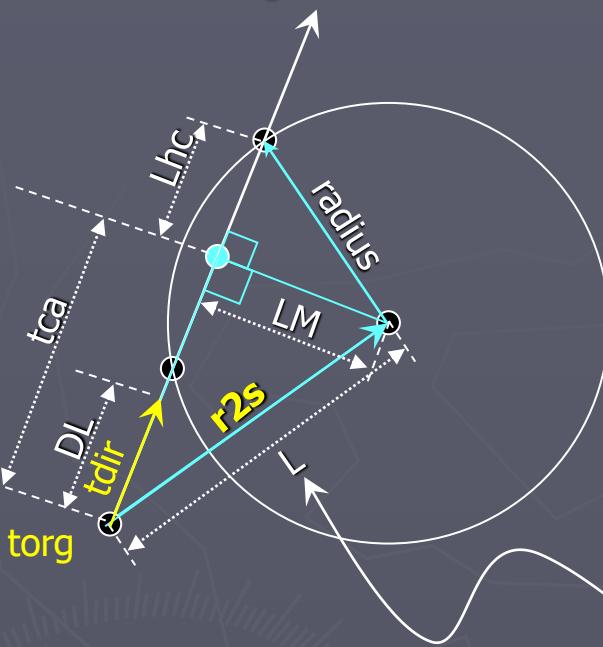
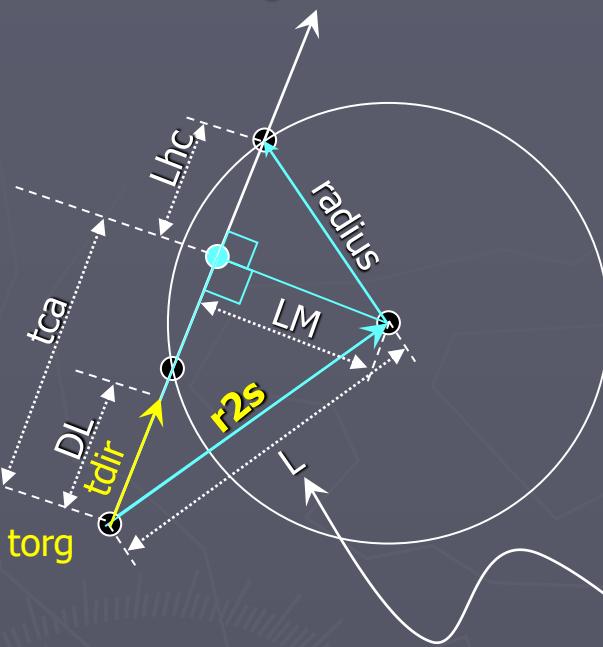
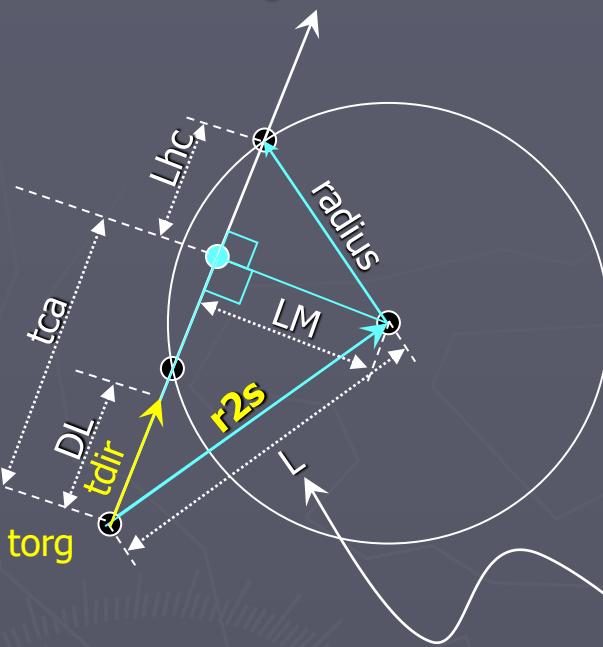
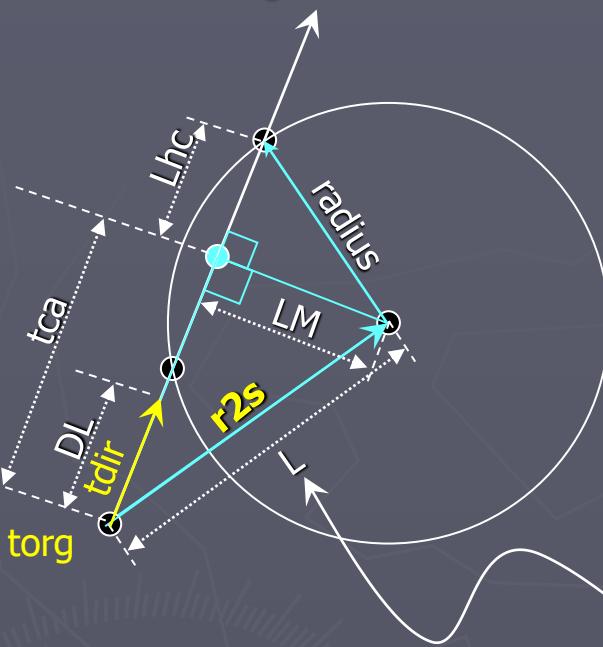
$$t_0 = tcaS / DL^2 + \sqrt{Lhc^2 / DL^2}$$



$$a^2 + b^2 = c^2$$



$$\text{ray}(t) = \text{torg} + t * \text{tdir}$$



Sphere II: Half-Chord Solver

Summary:

Half-Chord or '2-triangles' method:

1) Make transformed ray:

$\text{torg} = [\text{worldRay2model}][\text{worldRay.origin}]$
 $\text{tdir} = [\text{worldRay2Model}][\text{worldRay.dir}]$

2) Find vector: $\text{r2s} = \text{Sphere.ctr} - \text{torg}$;

Find scalar: $L2 = (\text{r2s} \cdot \text{r2s})$;

Find tca Scaled : $\text{tcaS} = \text{tdir} \cdot \text{r2s}$

IF ($L2 > \text{radius}^2$) then:

Ray origin is OUTSIDE the sphere, so:

► IF($\text{tcaS} < 0$) **MISS!**: sphere behind ray origin

3) Find scalar: $\text{DL2} = (\text{tdir} \cdot \text{tdir})$

Find scalar: $\text{tca2} = (\text{tcaS} * \text{tcaS}) / \text{DL2}$

Find scalar: $\text{LM2} = L2 - \text{tca2}$

4) IF($\text{LM2} > \text{radius}^2$) *** **MISS!** ***

(shortest distance² from ray to sphere center is $> \text{radius}^2$)

IF (shadow ray), *STOP*: we'll hit somewhere.

Find scalar: $\text{L2hc} = \text{radius}^2 - \text{LM2}$;

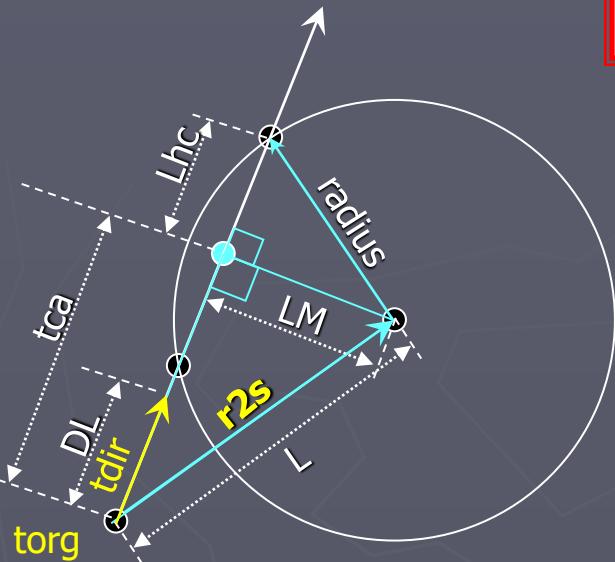
5) IF ($L2 > \text{radius}^2$): ray origin OUTSIDE sphere

▪ THEN Find ray-time t for 2 hit-points:

$t0, t1 = (\text{tcaS}/\text{DL2}) +/- \sqrt{(\text{L2hc}/\text{DL2})}$

▪ ELSE: ray origin INSIDE sphere: get t for 1 hit-point

$t0 = (\text{tcaS}/\text{DL2}) + \sqrt{(\text{L2hc}/\text{DL2})}$



$$\text{ray}(t) = \text{torg} + t * \text{tdir}$$

$$a^2 + b^2 = c^2$$



Sphere II: Half-Chord Solver (My verbal explanation)

Several names used for this method: look for 'Half-Chord' or '2-triangle' method

1) First, find the vector r_{2s} that stretches from ray.orig to the sphere's center.

Easy for us, because we put our sphere at model-space origin $(0,0,0,1)$;

$$R_{2s} = (0,0,0,1) - \text{ray.orig.}$$

2) What's the squared length of r_{2s} ? Call it L_2 .

If it's $L_2 < \text{sphere radius}^2 = 1$,

then our ray MUST begin INSIDE the sphere, a special case we'll address later.

3) If $L_2 > \text{sphere radius}^2 = 1$,

then our ray begins OUTSIDE the sphere.

We will extend the ray both ways ($t < 0, t > 0$) to form a line

that either misses the sphere entirely, or pierces it at TWO points, not one.

4) Define the 'chord' as the line-segment inside the sphere between those two points.

The chord mid-point is special; it is the point on the ray closest to the sphere's center.

5) Use dot-product to find the distance along the ray to the chord's midpoint, where $t=tca$.
But there's a trick here: we don't REALLY need tca , we only need $tca^2 \dots$

SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Half Chord solver: fast, but complex; can hide bugs...
Is there an elegantly simple (but slower) solution?

YES! A simple, fairly-obvious, but *iterative* method!

YES! Generalizes too!
works on ANY signed-distance functions,
not just spheres! (e.g. Bloomenthal ‘skeletons’)

See (video): <https://youtu.be/PGtv-dBi2wE>

See (illustrated text):

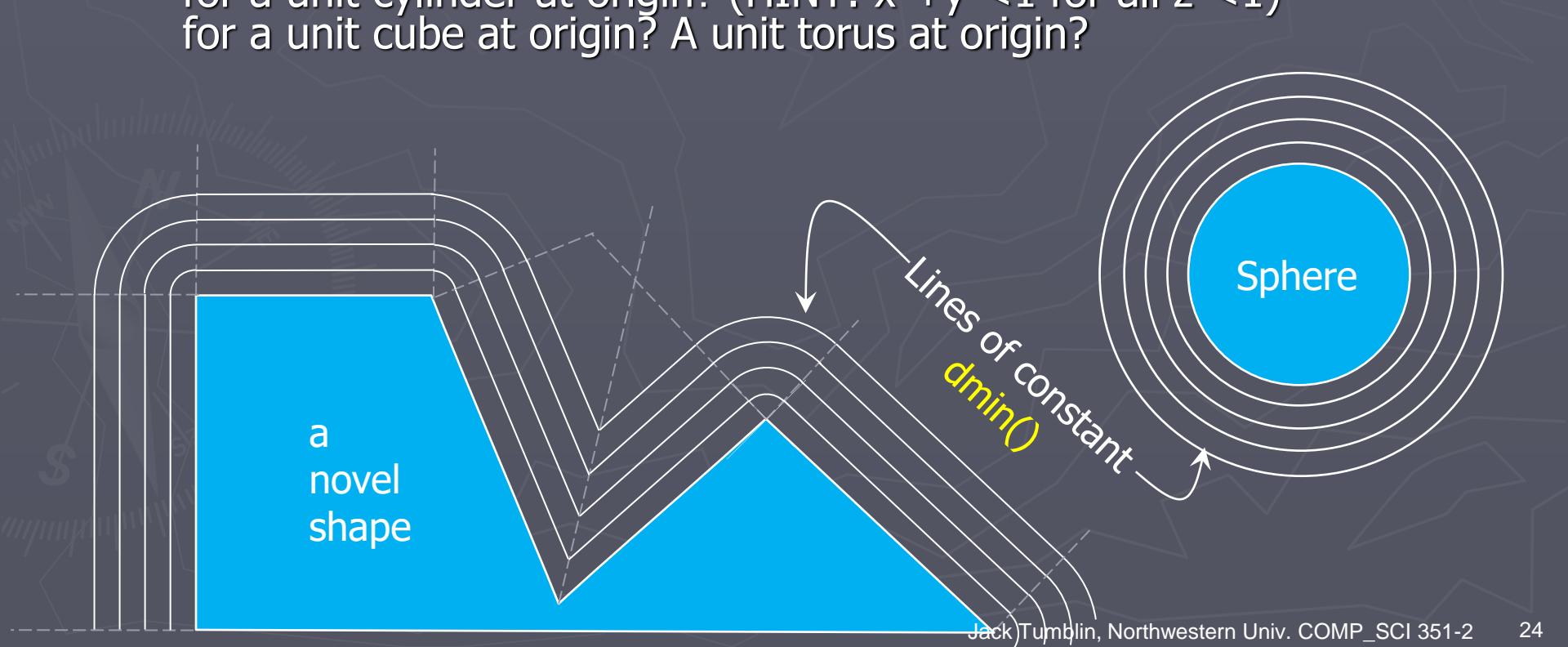
<http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/>

SPHERE III: 'Ray Marching', aka 'Sphere Tracing'

Key Idea 1:

'Make a 'close-ness' function $dmin(x,y,z)$ that computes the smallest distance from point (x,y,z) to a ray-traced shape:

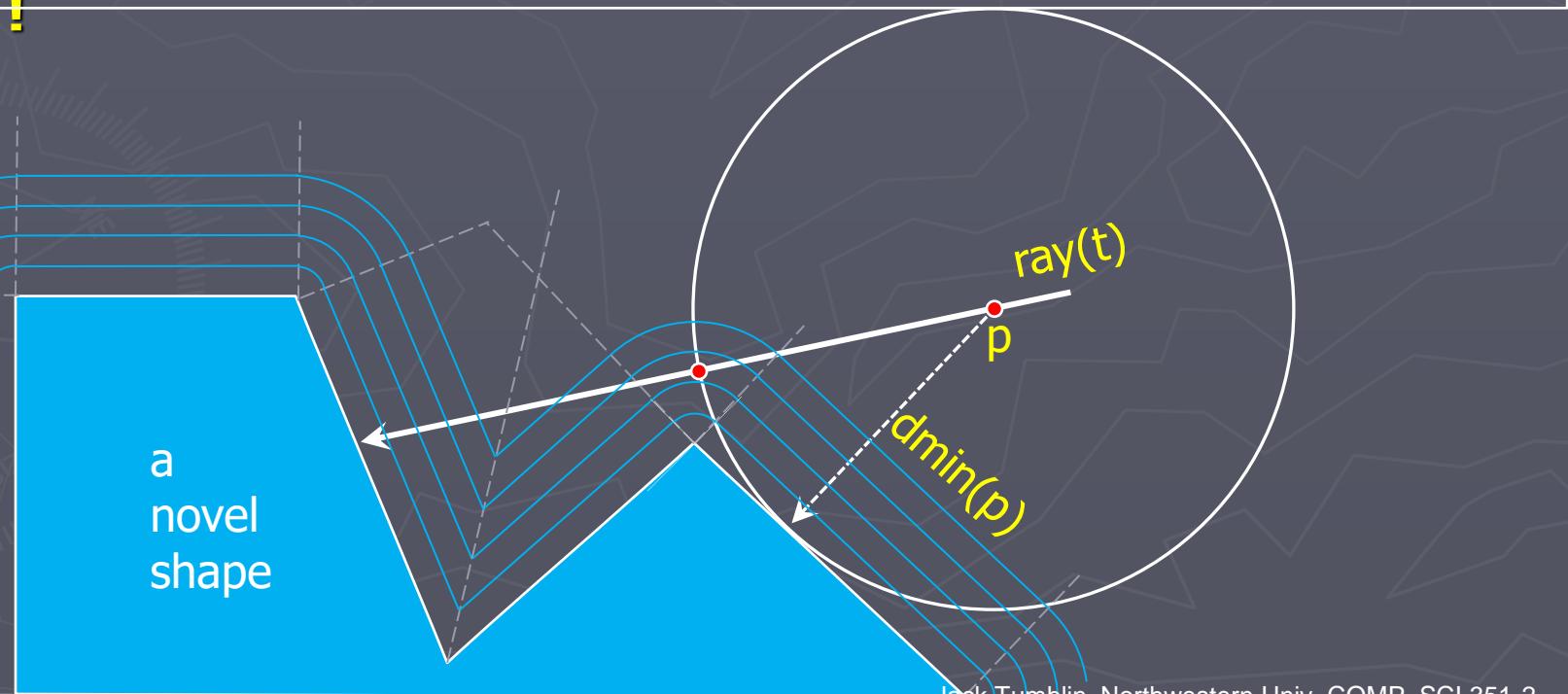
- EASY for a unit sphere at origin: $dmin(x,y,z) = \sqrt{x^2+y^2+z^2} - 1$;
- **CHALLENGE:** how would you compute $dmin()$ for other shapes?
for a unit cylinder at origin? (HINT: $x^2+y^2<1$ for all $z^2<1$)
for a unit cube at origin? A unit torus at origin?



SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Key Idea 2:

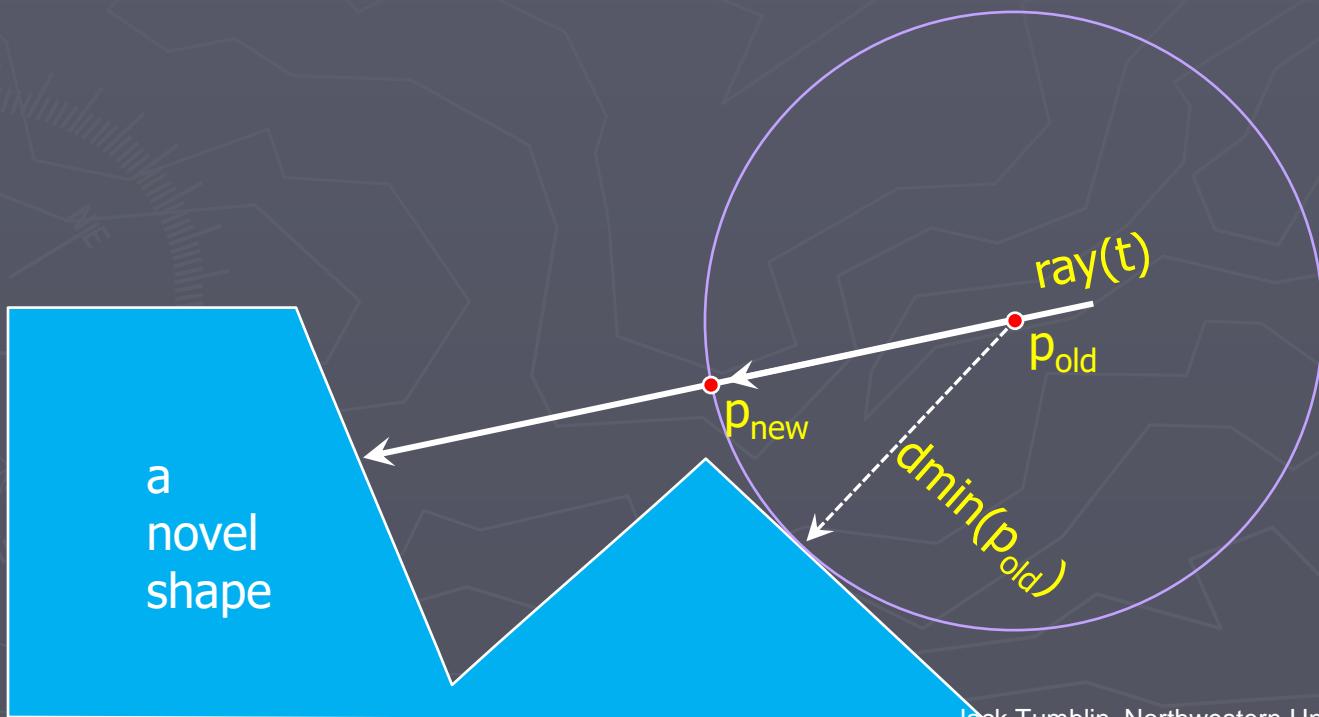
- ▶ Imagine a point p located on a ray origin: $\text{ray}(0) = p$
- ▶ Imagine a sphere centered at point p with radius $d_{\min}(p)$:
 - That sphere can’t contain any point on our surface. None!
(if it did, $d_{\min}(p)$ would be the distance to that surface point)
 - **THUS** point p can safely “march” forward along the ray by $d_{\min}(p)$
!



SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Key Idea 3:

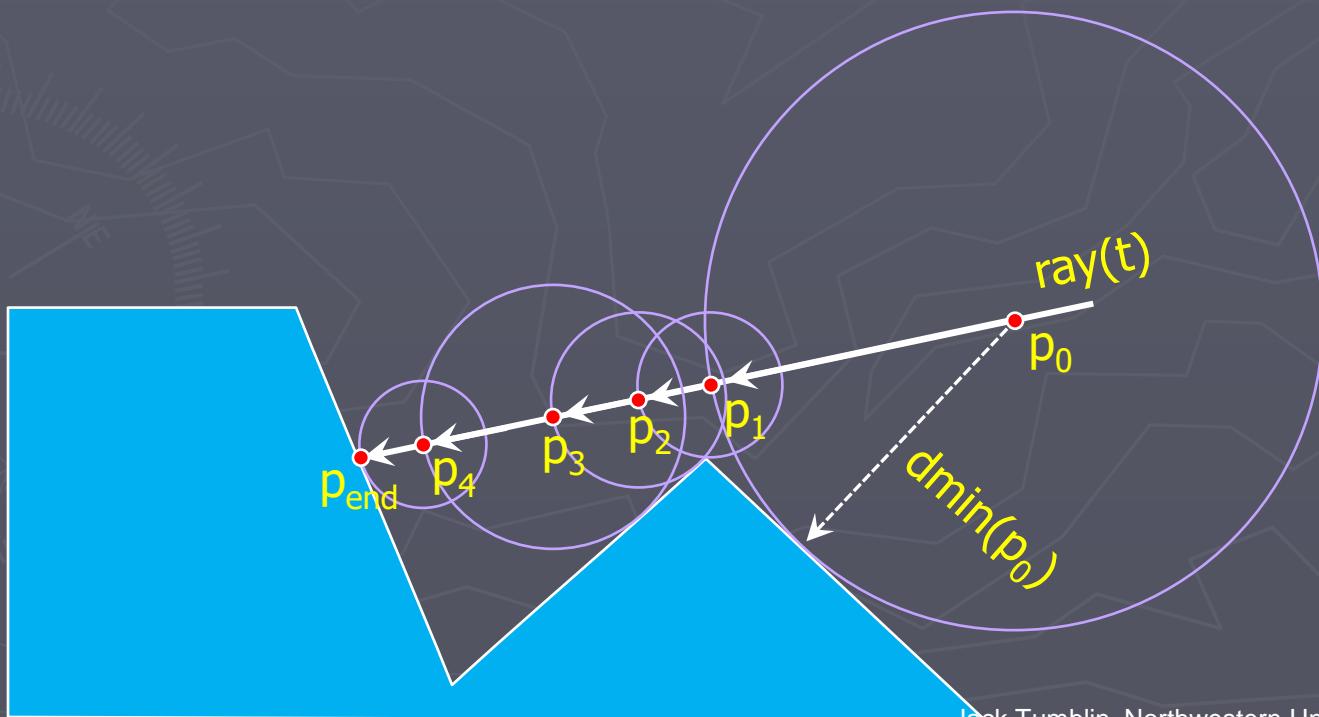
- ▶ Move(or “march”) p along ray to: $p_{\text{new}} = p_{\text{old}} + d_{\text{min}}(p_{\text{old}})$
- ▶ Keep “marching” until ...



SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Key Ideas 4 ,5:

- ▶ Move (or “march”) p_n along the ray: $p_{n+1} = p_n + dmin(p_n)$
- ▶ Keep “marching” until p_n arrives at surface, or misses it
e.g. until $dmin(p_n) \approx 0$ or $dmin(p_n) \approx \infty$ (“too big”)

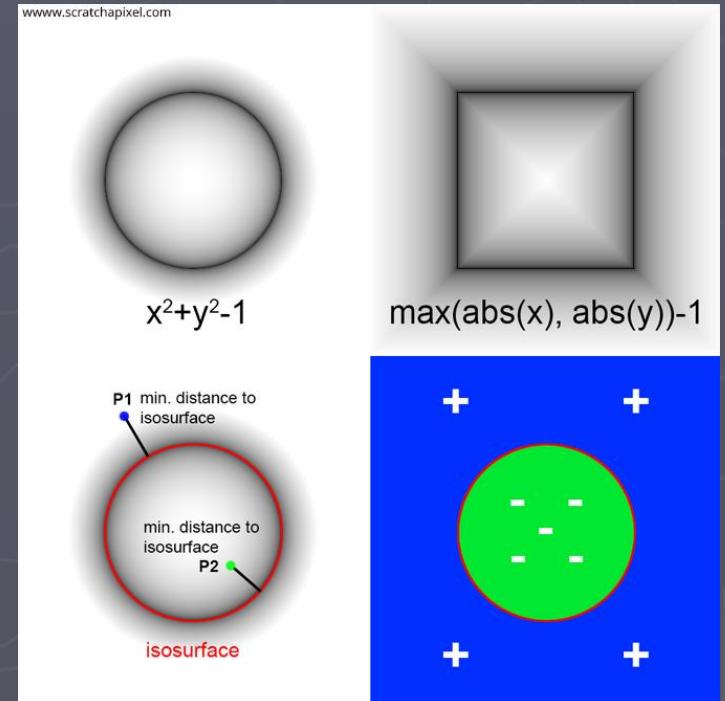


SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Surprise! Ray-Marching works for *ANY* shape for which you can define a precise **dmin()** !

Try web-searches (also search image,video) on:

- Signed distance fields
- Jules Bloomenthal images
‘Unchained Geometry’
- Sphere Tracing
- Ray Marching
- Hypertextures



Other Interesting Implicit Surfaces

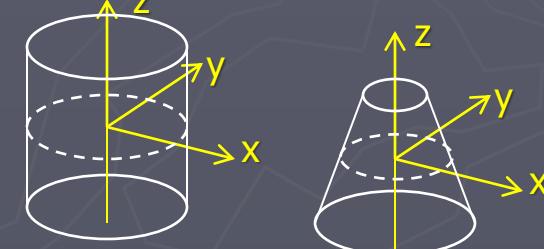
- ▶ Unit Cube (+/-1) at origin:

$$f(x,y,z) = \begin{cases} \text{if}(x^2 > \text{both } y^2 \text{ & } z^2): x^2 - 1 \\ \text{if}(y^2 > \text{both } x^2 \text{ & } z^2): y^2 - 1 \\ \text{if}(z^2 > \text{both } x^2 \text{ & } y^2): z^2 - 1 \end{cases}$$

- ▶ Unit-radius Cylinder at origin:

- ▶ Generalized cylinder:

make top disc's radius adjustable: set to zero to create a cone



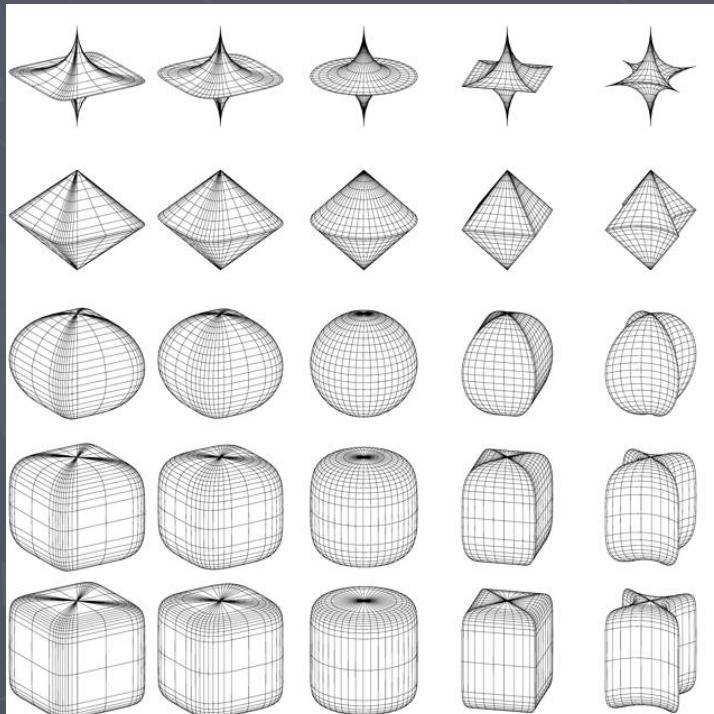
- ▶ Torus? Blinn's 'blobbies'? Meta-Balls?
Quadrics? SuperQuadrics?
See Lengyel reading, FS Hill reading, see
Google... (ray – <shape> intersection test)

SuperQuadratics

3D spheres raised to higher powers:

$$f(x,y,z) = x^r + y^s + z^t - 1$$

for any integer r,s,t



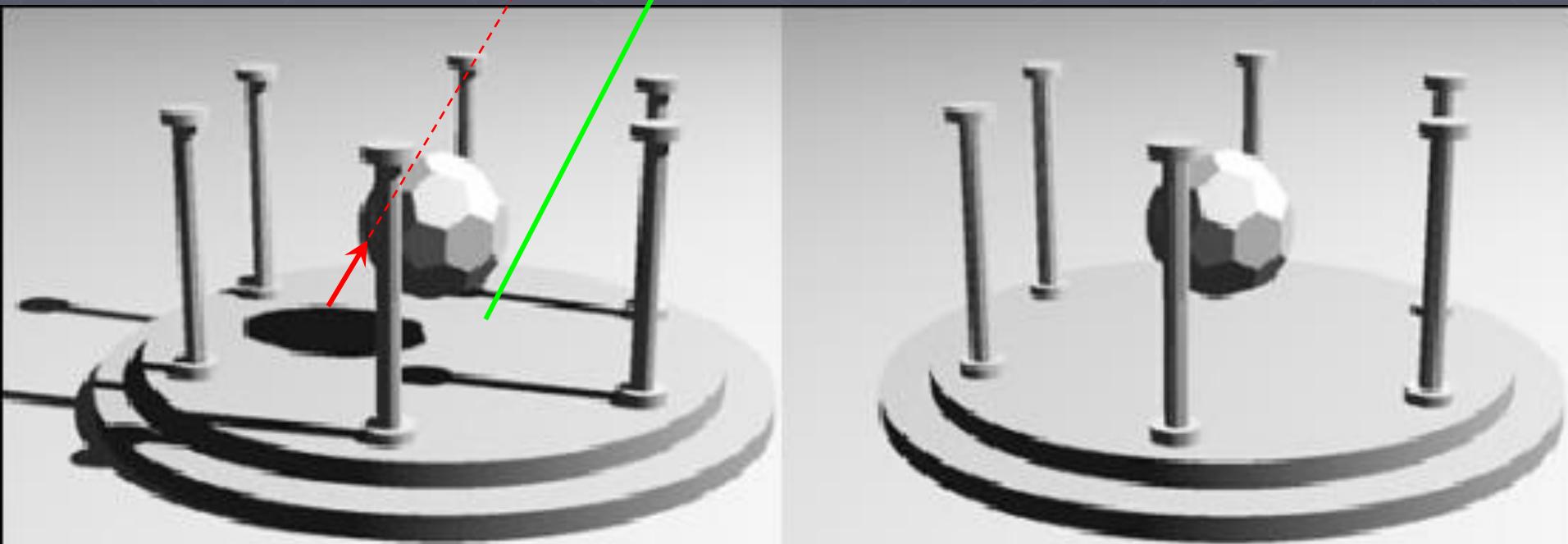
Light your First Sphere? soon...

Phong Basics:

- ▶ First, Make a fixed-color sphere ('diffuse' color) K_d
- ▶ Next, Find Surface normal N . Set color: $\text{clamp}(n_x, n_y, n_z)$
- ▶ Next, Finds N dot-product with vector L
(L == vector towards light from hit-pt.)
- ▶ Compute diffuse-only screen color: $K_d N \cdot L$
- ▶ Easy yet POWERFUL COLORING ALTERNATIVES:
'procedural' textures
'solid' textures $r(x,y,z)$, $g(x,y,z)$, $b(x,y,z)$
3D checkerboards, 3D ramps, sinusoids, etc.
Perlin noise, turbulence functions....

Do Shadows First -- Very Easy!

- ▶ Important height cues for disjoint objects
- ▶ Simple in ray-tracing, hard in WebGL
- ▶ How? Trace a 'shadow' ray from hit-point to lamp position:
IF this 'shadow' ray is blocked by ANY CGeom object,
THEN no light from that lamp reaches that hit-point



Code Organizing 1: More Shapes

- ▶ CScene Class:
make **COLLECTION** of CGeom objects
(Simplest? Use `CGeom item[MAX_SHAPES];`)
- ▶ CGeom class: expand to describe *any* shape:
-- Ideally, make CGeom-derived prototypes
-- OR -- Too exotic? Use this simple (but ugly) way:

- Add data member: `shapeType`; values from
 - `const GEOM_NONE = 0; // no shape; empty/invisible object`
 - `const GEOM_GNDPLANE = 1; // xy plane with lines`
 - `const GEOM_DISK = 2; // flat 2-sided zero-thickness disk`
 - `const GEOM_SPHERE = 3; // unit sphere @ origin`
 - `const GEOM_CUBE = 4; // unit cube @ origin`
 - `const GEOM_CYLINDER = 5; // cone or cylinder`

- Include *all* data members needed for all shapes:
ignore those not needed for current '`shapeType`'

Code Organizing 2: Tracing Refactored

Multiple CGeom objects?

Light sources?

Shadow rays?

Reflection rays?

YIKES!

Suddenly your `CScene.makeRayTracedImage();`
can get HUGE, gets MESSY, gets COMPLICATED!

*** Refactor:

divide collections of complex tasks
among several smaller simpler functions ***

Code Organizing 2: Tracing Refactored

- ▶ **GOAL:** create function(s) to trace one *given* ray, and to find a color at the ray endpoint:
 - What's the **input**? → One 'eye' ray through one CCamera pixel-area;
 - What's the **output**? → The ray's on-screen color (for **CImgBuf** image)
 - Be sure to choose functions suitable for recursion!
- ▶ **STRATEGY (recommended):**
 - 1) **CScene.traceRay(r)** : trace one given ray **r** thru the entire **CScene** contents. Find all its hits, and where.
 - 2) **CScene.findShade()**
From that ray and its collisions, find color. As needed, trace shadow rays, reflection rays, transparency rays; apply lighting, materials and textures to find ray color(s).
 - 3) **CRay prototype** already describes rays; now create a **CHit prototype** to hold one ray/scene collision point

CHit: Holds Ray 'Collision' Pt

Create **CHit** object prototype to:

- ▶ Hold the **traceRay()** results: (for **findShade()**)
 - 't' value; traced ray length (to object intersection point)
 - 'hitPoint'; ray/object intersection point in world space
 - 'hitGeom'; (reference to) CGeom object pierced by ray
- ▶ Describe surface conditions at ray-piercing point:
 - Hold index (or indices) for material(s) used there;
 - Hold all geometry needed for shading: view vector, surface normal, shadow rays, transparency rays, etc.
- ▶ Acts as input and output for **findShade()**

-- CHit objects keep recursive tracing simple --

Try This: Split up the big fcn `CScene.makeRayTracedImage()`

How should it delegate tracing & color-finding?

- ▶ Currently it's a **Great Big Loop**; it makes all pixels in the image buffer (global var: `CImageBuf g_myPic`)
 - Calls `CCamera.makeEyeRay()` for each pixel to make all of that pixel's 'eye' rays (`CScene.rayNow`) (No change)
 - Calls `CScene.traceRay(inRay, myHit)` to test **all** `CGeom` objects in our `CScene` for ray-intersection. Modifies `CHit` object '`myHit`' to describe 'hit point' nearest the camera (holds **what** we hit & **where** we hit it).
 - Calls `CScene.findShade(myHit);` to assess screen color at the end of the ray. It completes the `CHit` object, gets the final color for the eye ray. (and may use recursion to do it! ...)

REVISED CScene.makeRayTracedImage()

- ▶ How should it delegate tracing & color-finding?
- ▶ a Great Big Loop; makes all pixels in the image buffer
(global var: CImageBuf myPic)
 - Calls CCamera.makeEyeRay() for each pixel
to make all of that pixel's 'eye' rays (CScene.rayNow)

Trace the eye Ray, find its color : USE THIS PAIR RECURSIVELY

- Calls **CScene.traceRay(inRay, myHit)**
to test ***all*** CGeom objects in our CScene for ray-intersection.
Modifies CHit object '**myHit**' to describe 'hit point' nearest the camera
(holds ***what*** we hit & ***where*** we hit it).
- Calls **CScene.findShade(myHit);**
to assess screen color at the end of the ray.
It completes the CHit object, gets the final color for the eye ray.
(and may use recursion to do it! ...)

CScene.findShade()

(may grow and need further refactoring later)

Find the screen-color from the nearest 'hit-point' for this ray:
compute OpenGL-like shading (Phong light-sources and materials),
but improve it with shadows, reflections, and transparency.

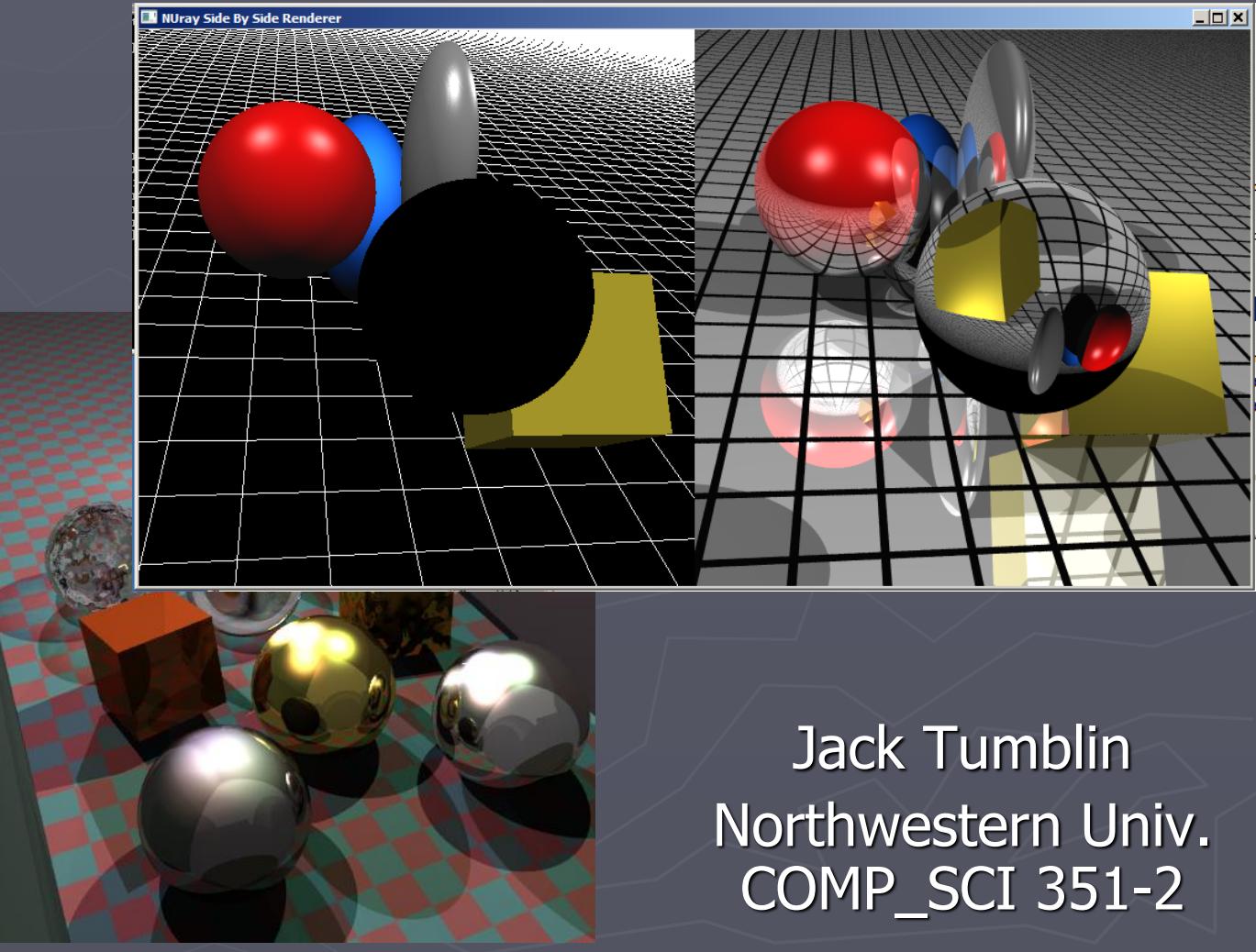
- ▶ Call on the CGeom object we hit:
 - Find/retrieve its materials properties at the hit-point (K_ambient, K_diffuse, K_specular, K_emissive, K_shininess)
 - Find/retrieve the shading vectors at the hit-point: surface normal, view vector, light-direction vectors, reflection vectors, etc.
- ▶ Test the hit-point's lighting: what light sources illuminate this surface point?
 - Create a shadow ray from hit-point to each light source
 - call CScene.traceRay() to find what each shadow ray hits, if anything.
Nothing? no shadow at hit-point; enable this light
- ▶ Find the color of a reflected ray:
 - Create a reflection ray at the hit-point (use surface normal and viewN vector)
 - Call CScene.traceRay() to find **what** and **where** the reflected ray hits (if anything)
 - Call CScene.findShade() to **find the color** at the reflection-ray's endpoint.

END

END

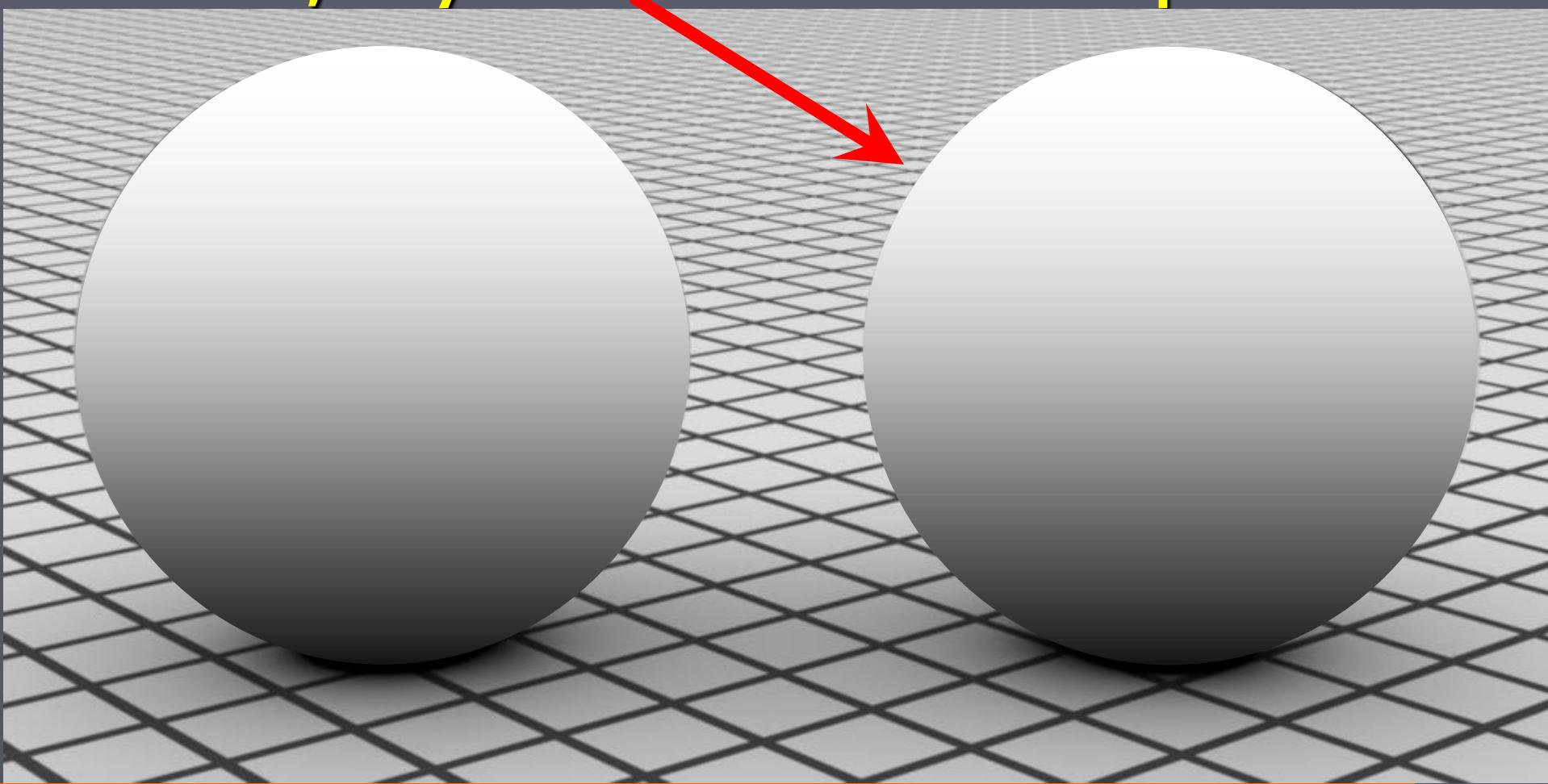


Ray Tracing D: Spheres, Shadows & Shading



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

Next, try this? : a basic sphere?



Matt Pharr, Greg Humphries: Physically Based Rendering Techniques (**PBRT**)
highly recommended beautiful in-depth book: <https://www.pbrt.org/gallery.html>

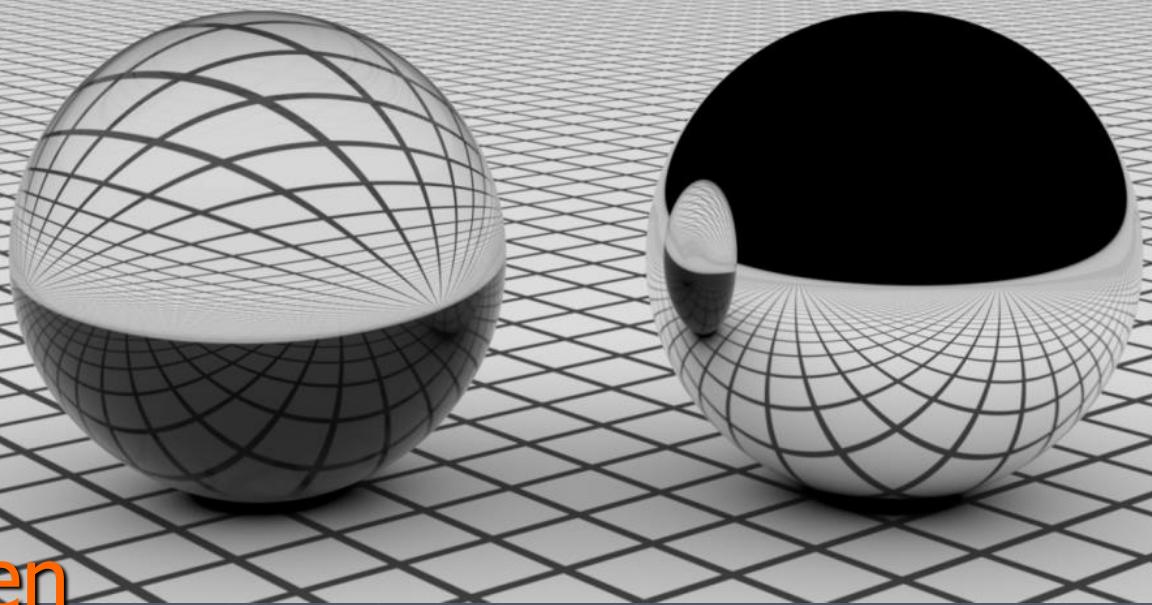
Our Plan:

Write code for:

- ▶ Camera, then
- ▶ Plane, then
- ▶ Antialiasing, then
- ▶ Transform, then
- ▶ Basic Disk and Sphere, then Shadow, then
- ▶ Phong Lighting,
- ▶ Reflection, then Transparency, then

Further Step-by-Step Refinements:

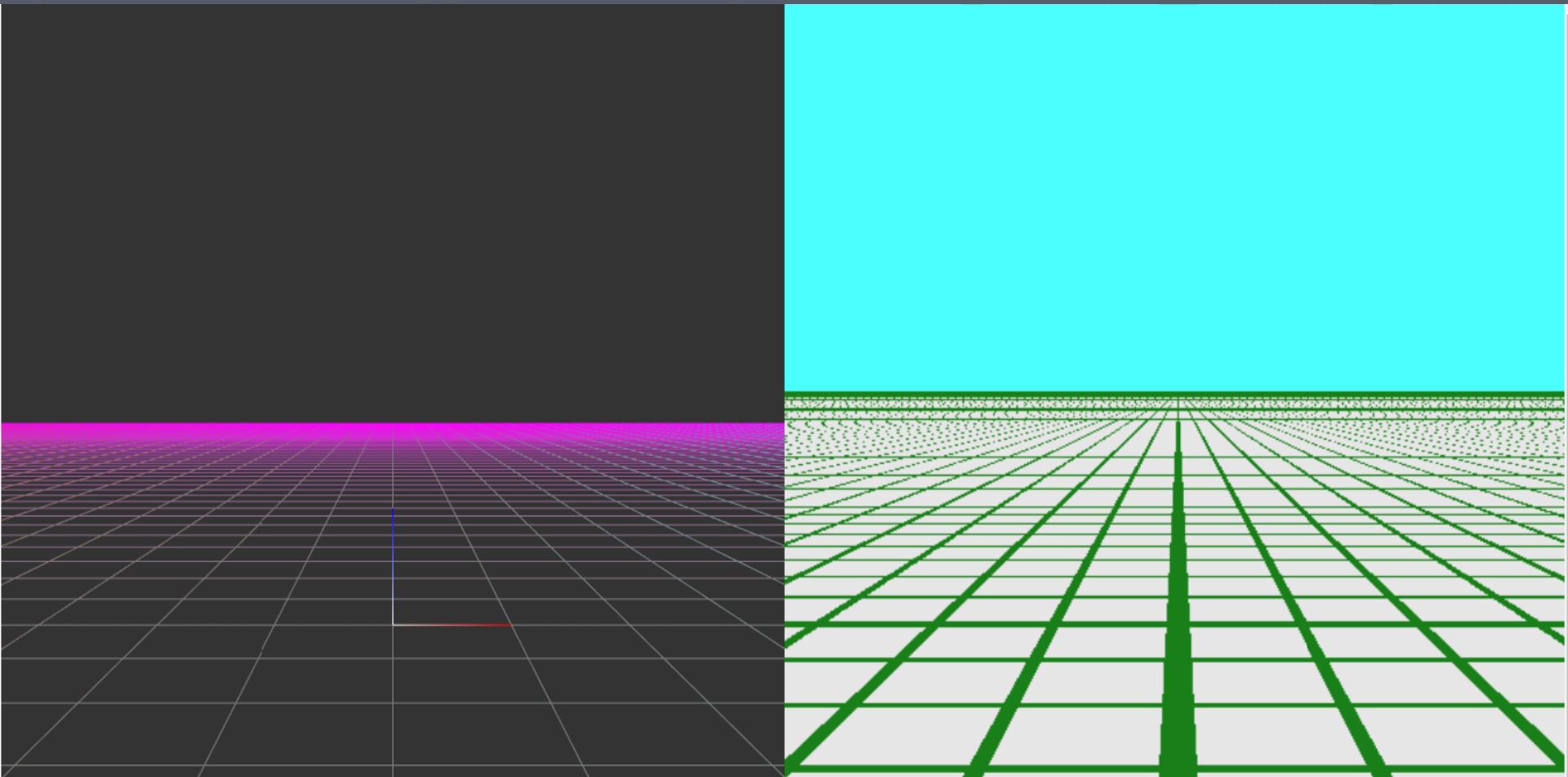
Lighting → Materials → Texture → CSG, etc. ...



*** **MAKE PICTURES AT EVERY STEP** ***

What we have: Gnd Plane

What's the **simplest 1st shape** we can add to it?



RECALL: Ray Tracing Implicit Surfaces

Shape specified by one scalar function $F(x,y,z)$:

$F() < 0$ for all (x,y,z) points **INSIDE**,

$F() > 0$ for all (x,y,z) points **OUTSIDE**, and

$F() = 0$ for all (x,y,z) points **ON** the surface.

Implicit functions for

arbitrary **plane**, distance d from origin:

$$F(x, y, z) = ax + by + cz + d = \mathbf{n} \cdot \mathbf{x} + d$$

unit-radius **sphere** at origin:

$$F(x, y, z) = x^2 + y^2 + z^2 - 1 = \mathbf{x} \cdot \mathbf{x} - 1$$

unit-radius **cylinder** along z axis:

$$F(x, y, z) = x^2 + y^2 - 1 \quad 0 < z < 1$$

Make your own! **HINT:** Search “Jules Bloomenthal”,
“Superquadrics”, “Implicit Surfaces”, “Metaballs”, etc ...

Ray Tracing Implicit Surfaces

Implicit Surface == set of all points where

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

Ray tracing == $\text{ray}(t) = \text{orig} + t \cdot \text{dir}$
or just $\mathbf{r}(t) = \mathbf{o} + t \cdot \mathbf{d}$

Here's a method for Ray-Object Intersections:
Plug $r(t)$ into implicit fcn and solve for t_{hit} :

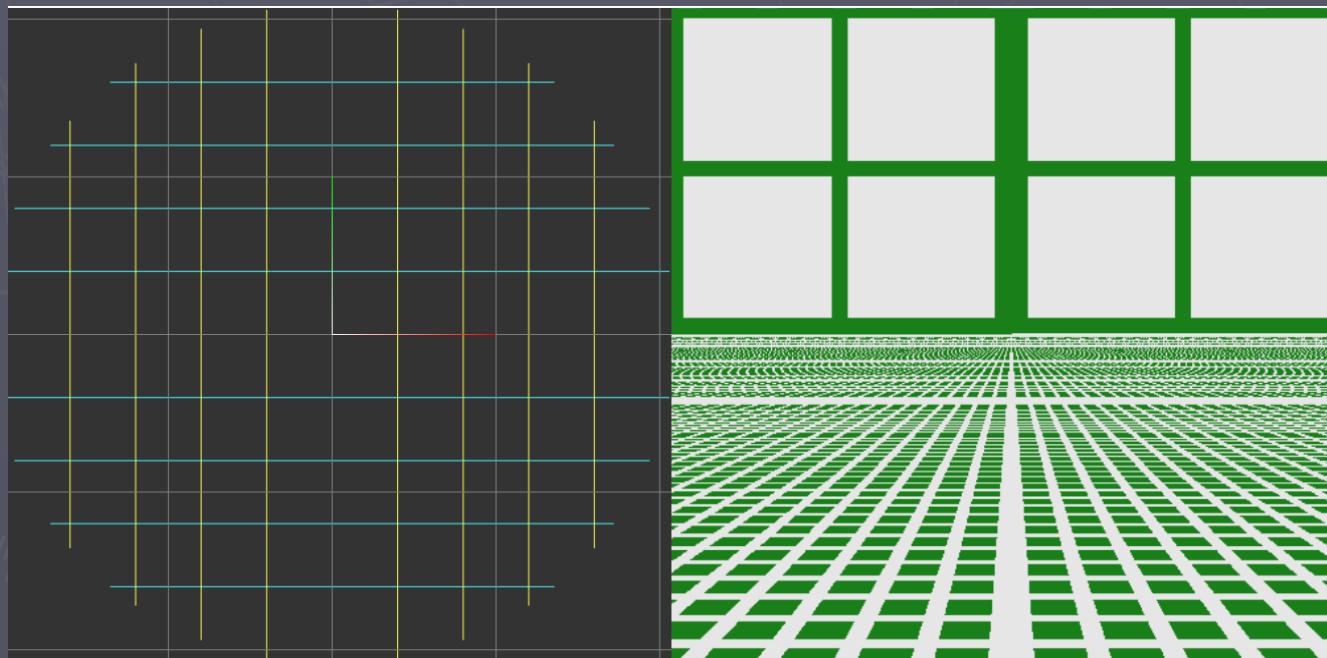
$$f(r_x(t_{\text{hit}}), r_y(t_{\text{hit}}), r_z(t_{\text{hit}})) = 0$$

Simplest 1st Shape? Try **WALL:**

Spheres? Ugh. Tricky, error prone, and
we can't see their orientation on-screen.

***AHA! As ground-plane tracing *already works*, let's try...

WALL: Ray-trace a 2nd ground-plane
that we ROTATED by 90°

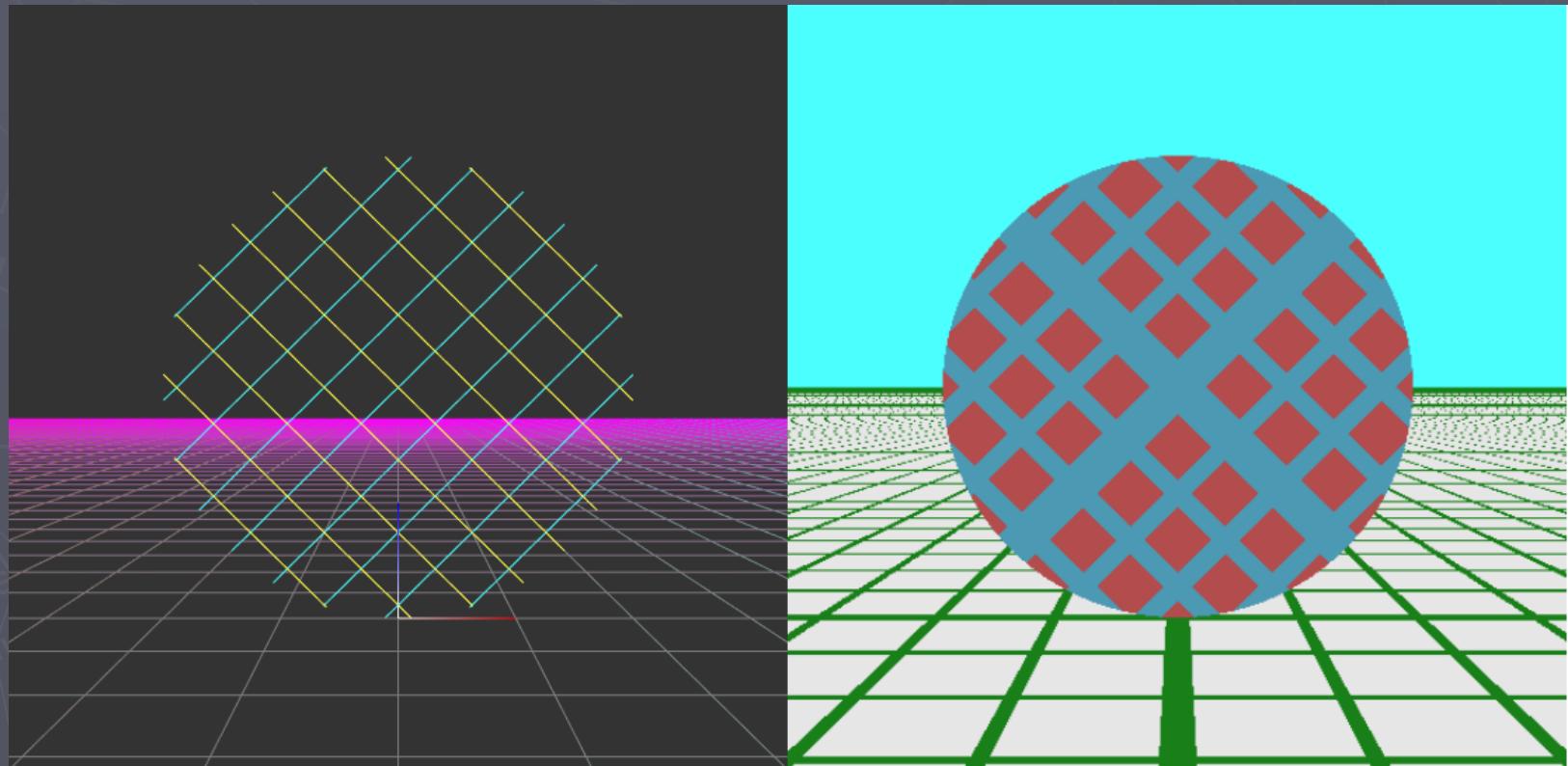


Hmm...
Awkward;
Confusing...

Simplest 1st Shape? Try **DISK**:

AHA! Let's limit the size of the wall...

- **DISK:** same as wall, but limit ray 'hits' to
 $x^2+y^2 < \text{CGeom.diskRadius2}$

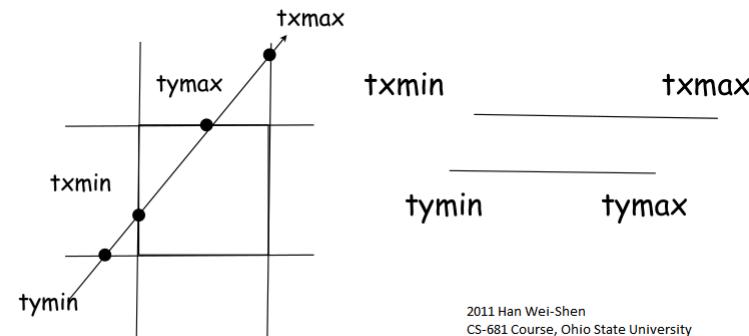


Try a CUBE: How?

► Simple-as-possible shape; NO parameters!

- Centered at origin, and
- Corners at $(\pm 1, \pm 1, \pm 1)$
(resembles CVV!)
- Use Transforms to set position, size, rotation...
- GIVEN this 2D ray/box test →
EXTEND to 3D!

Assuming the ray hits the box boundary lines at intervals $[txmin, txmax]$, $[tymin, tymax]$, the ray hits the box if and only if the intersection of the two intervals is not empty



2011 Han Wei-Shen
CS-681 Course, Ohio State University

► Cube is the intersection of 6 planes (or 3 slabs)

- How did we Ray-trace the ground-plane?
- What conditionals do we need to add?
- How can we avoid divide-by-zero errors?
- How can we make code efficient? (fewer costly conditionals)

Try a SPHERE: How? (3 answers)

- ▶ Naïve ray tracers skip transforms (bad idea):
 - More params: center **c**, squared radius **r²**, etc.
 - For some very clear (but naïve) summaries:
<http://www.lighthouse3d.com/tutorials/mathematics/ray-sphere-intersection/>
<https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/ray-tracing-practical-example>
- ▶ Our Ray tracer: sphere has **radius==1** centered at origin in model space
 - Parameters? **!NONE!**
 - transform ray: **worldRay2model** matrix
 - Find **t** values (if any) where ray pierces sphere
(? How will you indicate 'no hit' in CHit class?)

Sphere I: (Naïve) Quadratic Solver

- ▶ Ray $\mathbf{x}(t)$ leaves origin point \mathbf{o} in direction \mathbf{d} ; at time t ($t>0$) ray ends at $\mathbf{x}(t) = \mathbf{o} + t*\mathbf{d}$.
- ▶ ? At what time t_{hit} does ray hit a sphere centered at point \mathbf{c} with radius r ?
- ▶ Implicit Sphere: every point where $F(\mathbf{x})=0$
$$F(\mathbf{x}) = (\mathbf{x}-\mathbf{c}) \cdot (\mathbf{x}-\mathbf{c}) - r^2 = 0$$
(squared distance from center c matches r^2)
- ▶ Substitute ray $\mathbf{x}(t)$, simplify:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) = r^2$$

Sphere I: (Naïve) Quadratic Solver

- ▶ Expand this: $(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) = r^2$
- ▶ Put in quadratic form: $At^2 + Bt + C = 0$, and solve:

$$A = \mathbf{d} \cdot \mathbf{d},$$

$$B = 2(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d},$$

$$C = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2$$

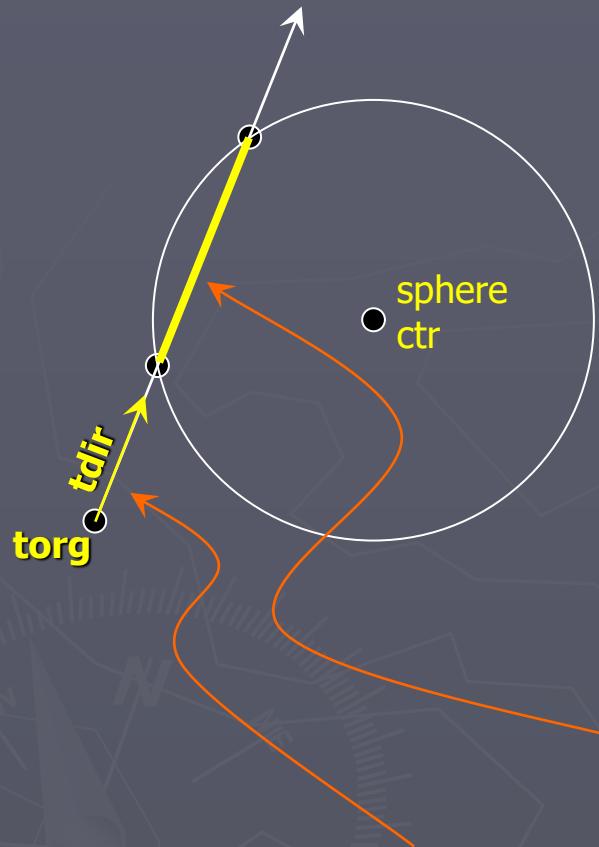
$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

- ▶ Standard Quadratic Solution:
if discriminant $D = (B^2 - 4AC) < 0$, $\rightarrow \rightarrow$ *NO* hits! --!MISS!
else ($D \geq 0$), so hit points are $t = (-B \pm \sqrt{D})/2A$

- ▶ **SLOW!** – there are better, MUCH faster methods!

NEXT: a very fast, popular, more ‘geometric’ method:
it also tells you if your ray starts within the sphere;
(we will need that!)

Sphere II: Half-Chord Solver



HARDER, BETTER, FASTER (, STRONGER... Daft Punk (2001))

'Half-Chord' or '2-triangle' method:

Key Ideas (1):

- ▶ Any 3D line will either:
 - MISS the sphere, or
 - PIERCE the sphere at 2 points
- ▶ 'Chord' == line segment inside

Careful! Use **model space** ray:

`torg` == *transformed origin pt.*

`tdir` == *transformed direction vector*

**FASTER! CLEVER!
LESS AMBIGUOUS!
(but complex; less obvious)**

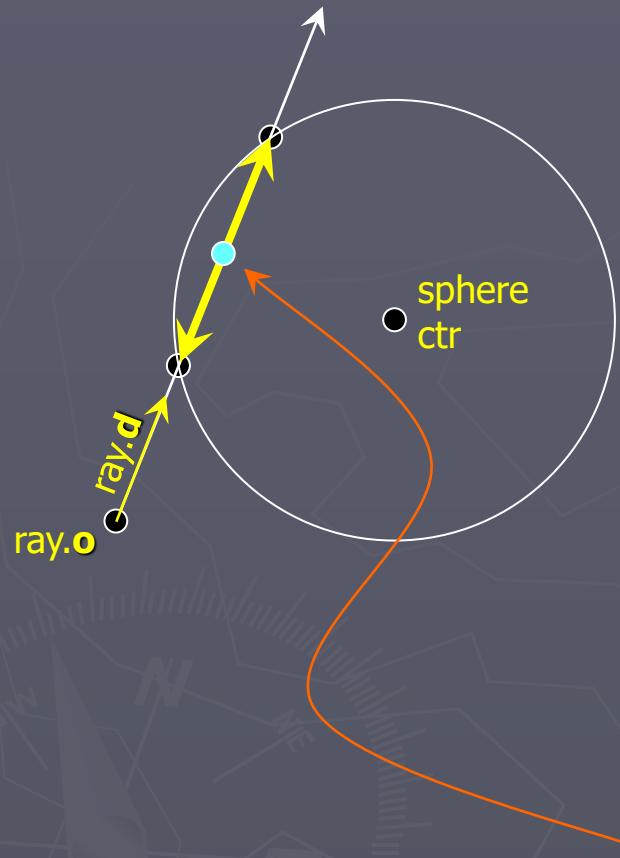
Sphere II: Half-Chord Solver

HARDER, BETTER, FASTER (, STRONGER ... Daft Punk)

'Half-Chord' or '2-triangle' method:

Key Ideas (2):

- ▶ All 3D lines either
 - MISS the sphere, or
 - PIERCE the sphere at 2 points
- ▶ 'Chord' == line segment inside
- ▶ Find Chord mid-point, then move
- ▶ +/- 'Half-Chord' length along ray



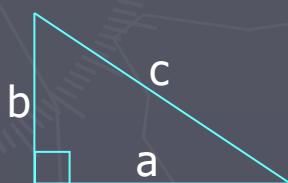
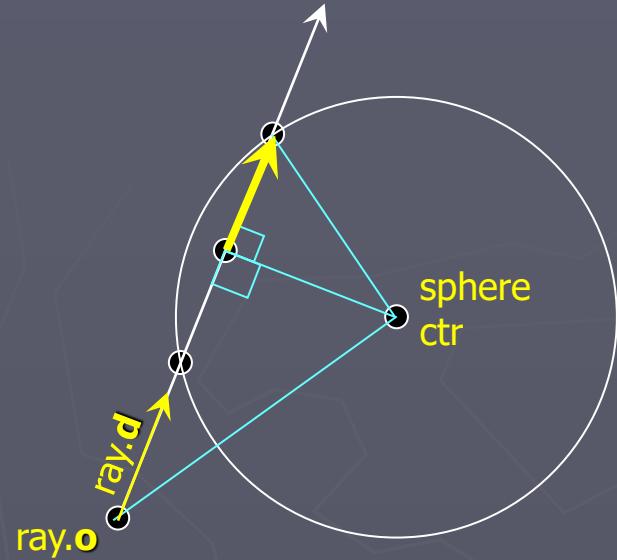
Sphere II: Half-Chord Solver

FASTER, SIMPLER, BETTER:
'Half-Chord' or '2-triangle' method:

Key Ideas (2):

- ▶ All 3D lines either
 - MISS the sphere, or
 - PIERCE the sphere at 2 points
- ▶ 'Chord' == line segment inside
- ▶ Find Chord mid-point, then move
- ▶ +/- 'Half-Chord' length along ray

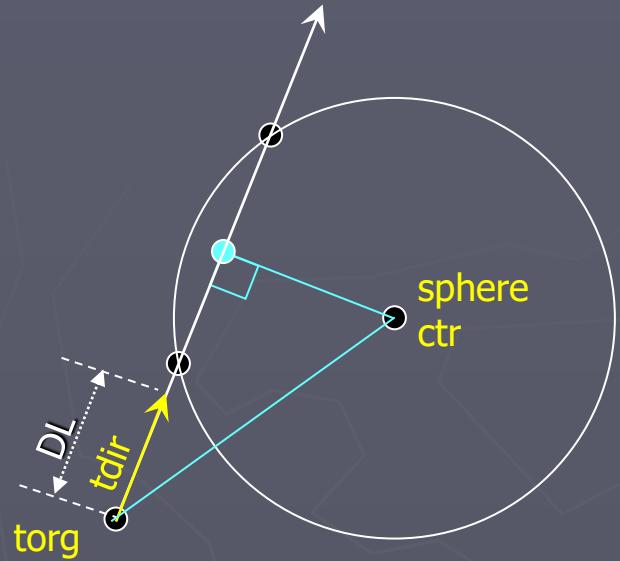
How? Use Pythagorean Theorem
on 2 right triangles, like this...



$$a^2 + b^2 = c^2$$

Sphere II: Half-Chord Solver (1)

Step 1: Define Transformed Ray

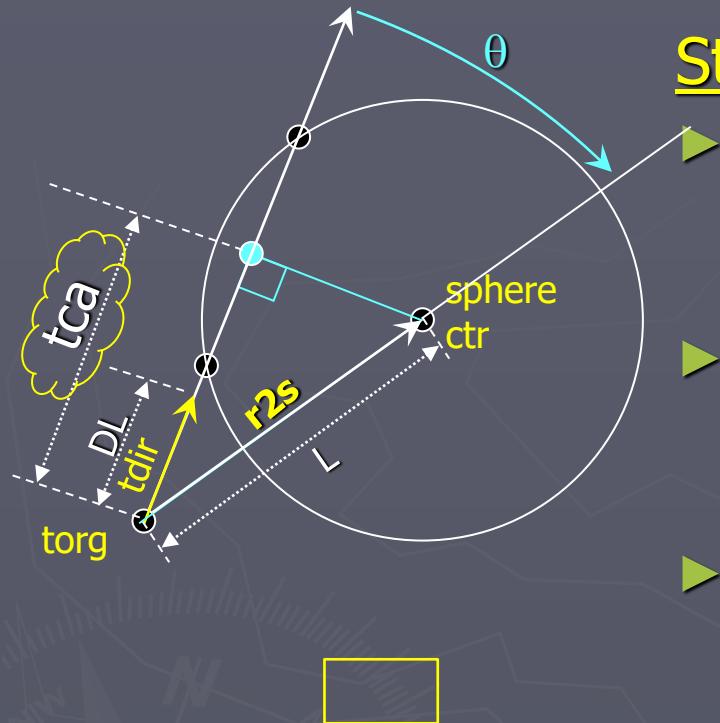


- ▶ Transform world-space ray to model-space ray:
 $torg = [\text{worldRay2Model}] [\text{worldRay.origin}]$
 $tdir = [\text{worldRay2Model}] [\text{worldRay.dir}]$
No need to normalize $tdir$! Instead,

Use scalar DL to denote length of $tdir$ vector, and we'll write DL^2 as $DL2$.
CAREFUL! don't compute DL yet!

- ▶ Recall: point on ray at ray-time t is:
 $\text{ray}(t) = torg + t*tdir$

Sphere II: Half-Chord Solver (2)



Careful!

Ray.o is a point: w=1;
r2s is a vector: w=0;

Step 2: Test 1st Triangle

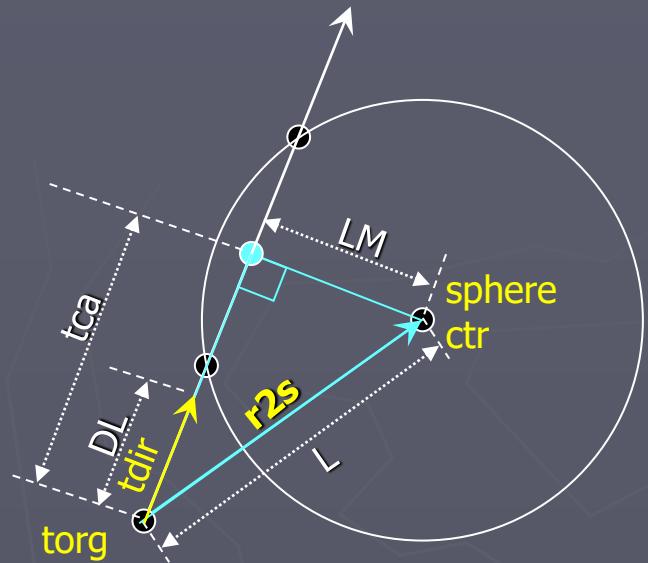
- ▶ Make $r2s$: ray-to-sphere-ctr vector
$$r2s = \text{ctr} - \text{torg} = 0 - \text{torg}$$

- ▶ Find L₂ (scalar): ($r2s$ length)²
$$L^2 = L^2 = (r2s \cdot r2s);$$

- ▶ IF ($L^2 > \text{radius}^2$) THEN:
 - ray begins OUTSIDE the sphere. good!
 - IF ($tcaS < 0$) THEN: !MISS! All Done!
sphere entirely behind the ray origin !
(No? maybe we hit the sphere – let's find where...)

- ▶ Find Mid-pt distance, by dot-product:
$$\begin{aligned} tdir \cdot r2s &= \|tdir\| * (\|r2s\| * \cos \theta) \\ &= DL * \left(\frac{tca}{\|r2s\|} \right) \\ tdir \cdot r2s &= tcaS, \text{ the } \underline{\text{scaled}} \text{ tca.} \end{aligned}$$

Sphere II: Half-Chord Solver (3)



$$\text{ray}(t) = \text{torg} + t * \text{tdir}$$

$$a^2 + b^2 = c^2$$

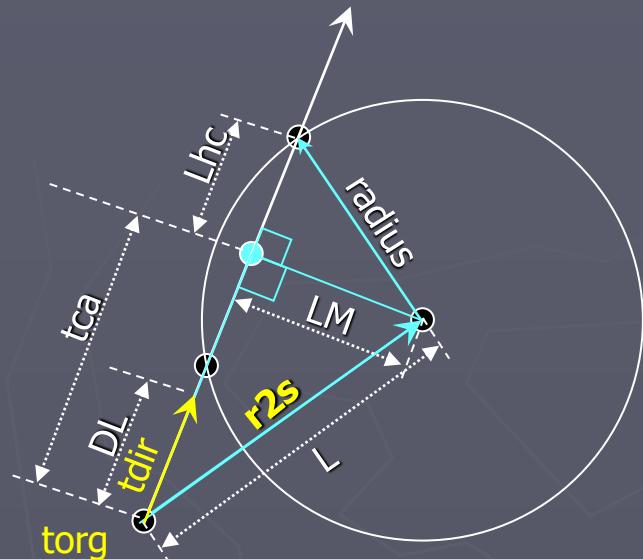


Step 3: Measure 1st Triangle (cont'd)

- ▶ Find Mid-pt distance, by dot-product:
$$\begin{aligned}\text{tdir} \cdot \text{r2s} &= ||\text{tdir}|| * (||\text{r2s}|| * \cos \theta) \\ &= \text{DL} * (\frac{\text{tca}}{\text{tcaS}}) \end{aligned}$$

$$\text{tdir} \cdot \text{r2s} = \text{tcaS}$$
, the *scaled* tca.
- ▶ Find DL2: = tdir vector length²
$$\text{DL}^2 = \boxed{\text{DL2} = (\text{tdir} \cdot \text{tdir})}$$
- ▶ Find tca² (*without* any scaling):
$$(\text{tcaS})^2 = (\text{tca} \cdot \text{DL})^2 = \text{tca}^2 \cdot \text{DL}^2$$
, so
$$\text{tca}^2 = \boxed{\text{tcaS} \cdot \text{tcaS} / \text{DL2} = \text{tca2}}$$
- ▶ Find LM² By Pythagorean Theorem:
$$\text{LM}^2 + \text{tca}^2 = \text{L}^2$$
, so
$$\text{LM}^2 = \text{L}^2 - \text{tca}^2$$
, or
$$\boxed{\text{LM2} = \text{L2} - \text{tca2}}$$

Sphere II: Half-Chord Solver (4)



$$\text{ray}(t) = \text{torg} + t * \text{tdir}$$

$$a^2 + b^2 = c^2$$



Step 4: Measure 2nd Triangle

- ▶ **IF($LM^2 > \text{radius}^2$) → ***MISS!*****
(shortest distance² from ray to sphere center is > radius^2)
- ▶ **IF (is_shadow_ray) → ** STOP! ****
(the ray hit the sphere;
but we don't care where! Stop!)
- ▶ **Find half-chord length² $L2hc$ by Pythagorean Theorem:**
 $Lhc^2 + LM^2 = \text{radius}^2$, so
 $Lhc^2 = \text{radius}^2 - LM^2$

Sphere II: Half-Chord Solver (5)

Step 5: Measure ray using 2nd Triangle

Find ray-time t at ray/sphere intersection points:

Ray-length at ANY ray-time t: $RL = t * DL$

Ray-length at chord ends: $RL = tca \pm Lhc$
combine, solve: $t = (tca \pm Lhc) / DL$

Recall these earlier results:

$tcaS = tca * DL$: rearrange to get:

$tca = tcaS / DL$, where

$DL = \sqrt{DL^2}$ and recall that

$Lhc = \sqrt{Lhc^2}$.

Now substitute, and divide again by DL:

$$\begin{aligned} t &= (tca \pm Lhc) / DL \\ &= ((tcaS / DL) \pm \sqrt{Lhc^2}) / DL \\ &= (tcaS / DL^2) \pm (\sqrt{Lhc^2} / \sqrt{DL^2}) \end{aligned}$$

IF ($L^2 > radius^2$) THEN

ray must pierce sphere at **two** hit-points:

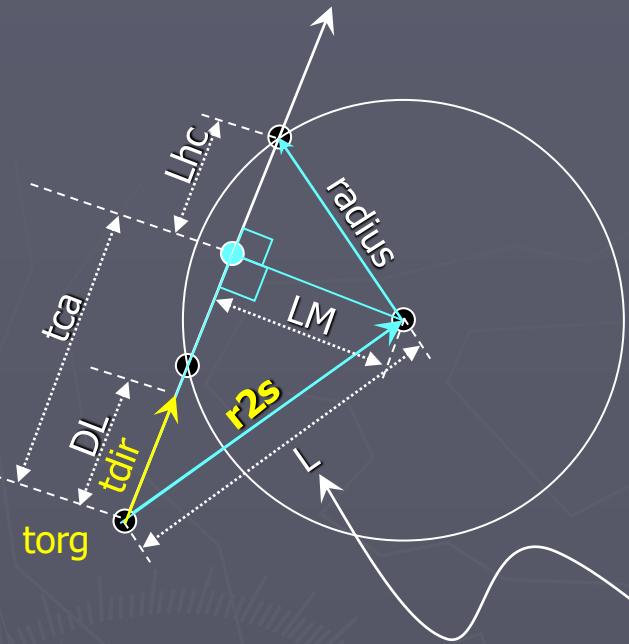
$$t_0, t_1 = tcaS / DL^2 \pm \sqrt{Lhc^2 / DL^2}$$

ELSE ray began INSIDE the sphere;

$$t_0 = tcaS / DL^2 + \sqrt{Lhc^2 / DL^2}$$



$$a^2 + b^2 = c^2$$



Sphere II: Half-Chord Solver (6)

Summary:

Half-Chord or '2-triangles' method:

1) Make transformed ray:

$\text{torg} = [\text{worldRay2model}][\text{worldRay.origin}]$
 $\text{tdir} = [\text{worldRay2Model}][\text{worldRay.dir}]$

2) Find vector: $\text{r2s} = \text{Sphere.ctr} - \text{torg}$;

Find scalar: $L2 = (\text{r2s} \cdot \text{r2s})$;

Find tca Scaled : $\text{tcaS} = \text{tdir} \cdot \text{r2s}$

IF ($L2 > \text{radius}^2$) then:

Ray origin is OUTSIDE the sphere, so:

► IF($\text{tcaS} < 0$) **MISS!**: sphere behind ray origin

3) Find scalar: $\text{DL2} = (\text{tdir} \cdot \text{tdir})$

Find scalar: $\text{tca2} = (\text{tcaS} * \text{tcaS}) / \text{DL2}$

Find scalar: $\text{LM2} = L2 - \text{tca2}$

4) IF($\text{LM2} > \text{radius}^2$) *** **MISS!** ***

(shortest distance² from ray to sphere center is $> \text{radius}^2$)

IF (shadow ray), *STOP*: we'll hit somewhere.

Find scalar: $\text{L2hc} = \text{radius}^2 - \text{LM2}$;

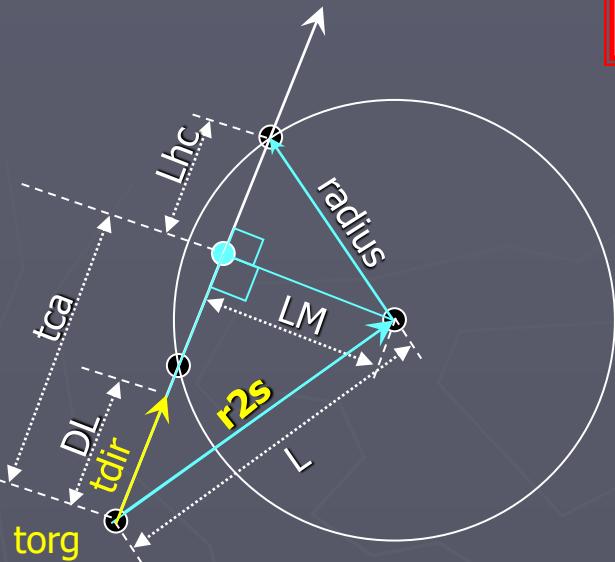
5) IF ($L2 > \text{radius}^2$): ray origin OUTSIDE sphere

▪ THEN Find ray-time t for 2 hit-points:

$t0, t1 = (\text{tcaS}/\text{DL2}) +/- \sqrt{(\text{L2hc}/\text{DL2})}$

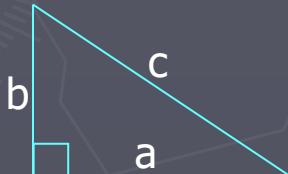
▪ ELSE: ray origin INSIDE sphere: get t for 1 hit-point

$t0 = (\text{tcaS}/\text{DL2}) + \sqrt{(\text{L2hc}/\text{DL2})}$



$$\text{ray}(t) = \text{torg} + t * \text{tdir}$$

$$a^2 + b^2 = c^2$$



Sphere II: Half-Chord Solver (My verbal explanation)

Several names used for this method: look for 'Half-Chord' or '2-triangle' method

1) First, find the vector r_{2s} that stretches from ray.orig to the sphere's center.

Easy for us, because we put our sphere at model-space origin $(0,0,0,1)$;

$$R_{2s} = (0,0,0,1) - \text{ray.orig.}$$

2) What's the squared length of r_{2s} ? Call it L_2 .

If it's $L_2 < \text{sphere radius}^2 = 1$,

then our ray MUST begin INSIDE the sphere, a special case we'll address later.

3) If $L_2 > \text{sphere radius}^2 = 1$,

then our ray begins OUTSIDE the sphere.

We will extend the ray both ways ($t < 0, t > 0$) to form a line

that either misses the sphere entirely, or pierces it at TWO points, not one.

4) Define the 'chord' as the line-segment inside the sphere between those two points.

The chord mid-point is special; it is the point on the ray closest to the sphere's center.

5) Use dot-product to find the distance along the ray to the chord's midpoint, where $t=tca$.
But there's a trick here: we don't REALLY need tca , we only need $tca^2 \dots$

SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Half Chord solver: **fast, but complex**; can hide bugs...
Is there an elegantly simple (but slower) solution?

YES! A simple, fairly-obvious, but *iterative* method!

YES! Generalizes too!
works on ANY signed-distance functions,
not just spheres! (e.g. Bloomenthal ‘skeletons’)

See (video): <https://youtu.be/PGtv-dBi2wE?t=219>

See (illustrated text):

<http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/>

SPHERE III: 'Ray Marching', aka 'Sphere Tracing'

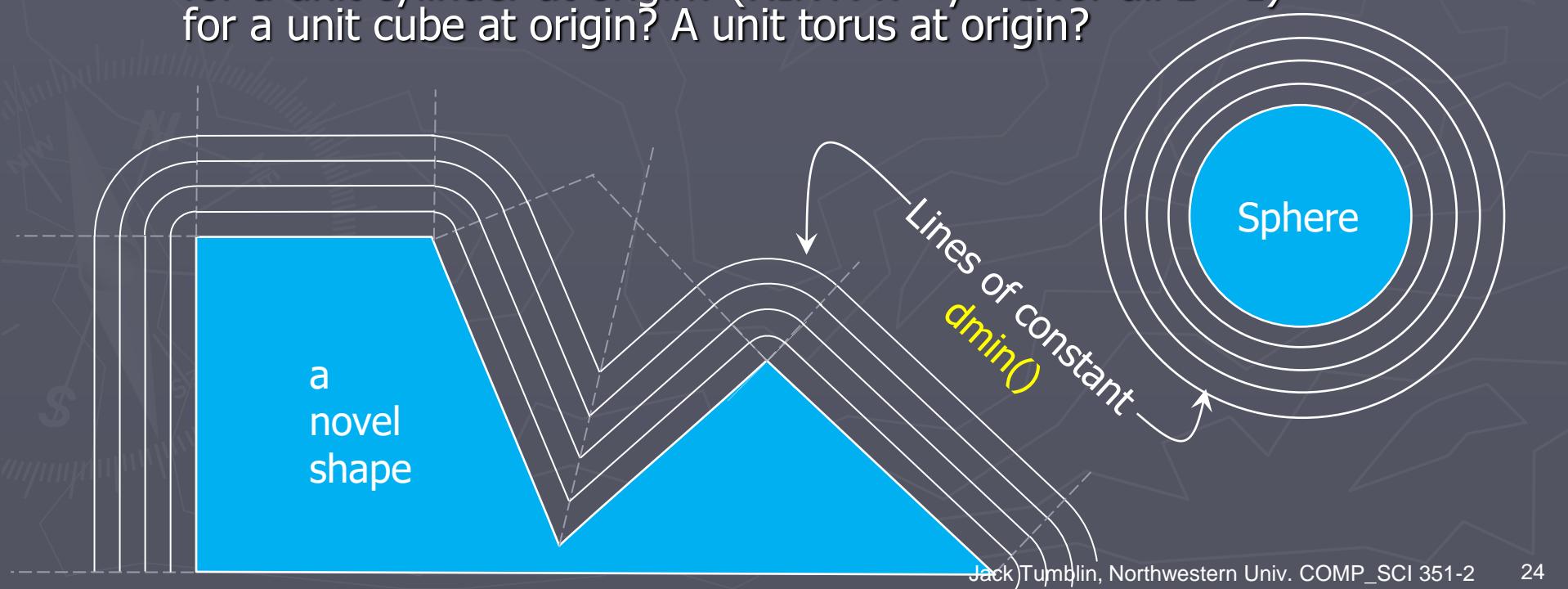
Key Idea 1:

'Make a 'close-ness' function $dmin(x,y,z)$

aka 'signed distance function' that gives us the

smallest distance from point (x,y,z) to our 3D shape:

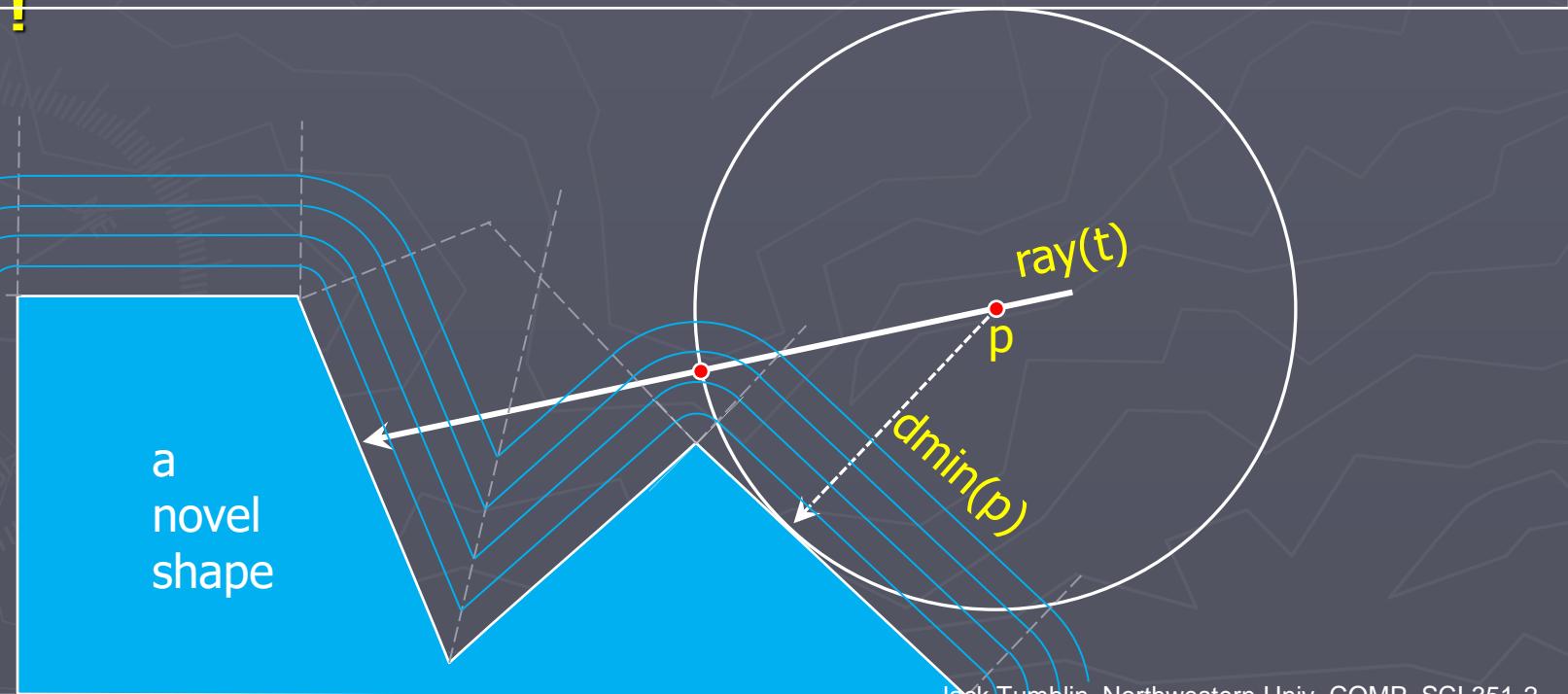
- EASY for a unit sphere at origin: $dmin(x,y,z) = \sqrt{x^2+y^2+z^2} - 1$;
- **CHALLENGE:** how would you compute $dmin()$ for other shapes?
for a unit cylinder at origin? (HINT: $x^2+y^2<1$ for all $z^2<1$)
for a unit cube at origin? A unit torus at origin?



SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Key Idea 2:

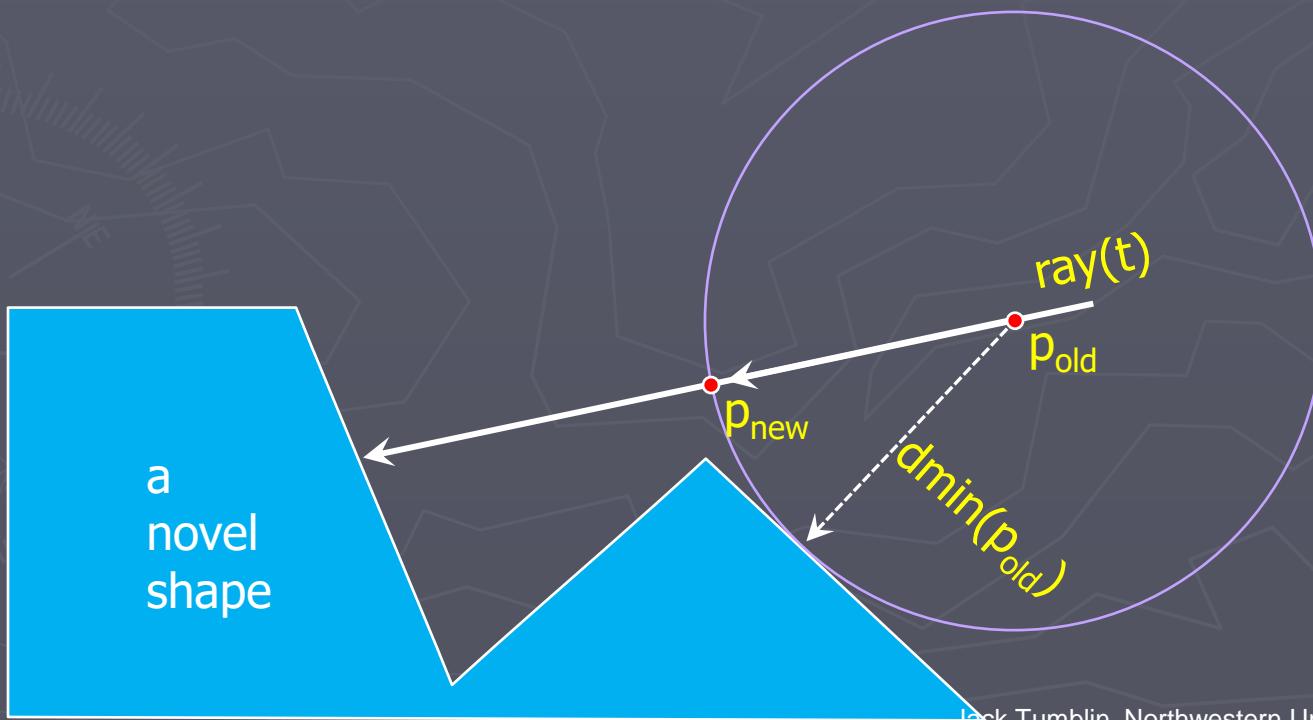
- ▶ Imagine a point p located on a ray origin: $\text{ray}(0) = p$
- ▶ Imagine a sphere centered at point p with radius $d_{\min}(p)$:
 - That sphere can’t contain any point on our surface. None!
(if it did, $d_{\min}(p)$ would be the distance to that surface point)
 - **THUS** point p can safely “march” forward along the ray by $d_{\min}(p)$
!



SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Key Idea 3:

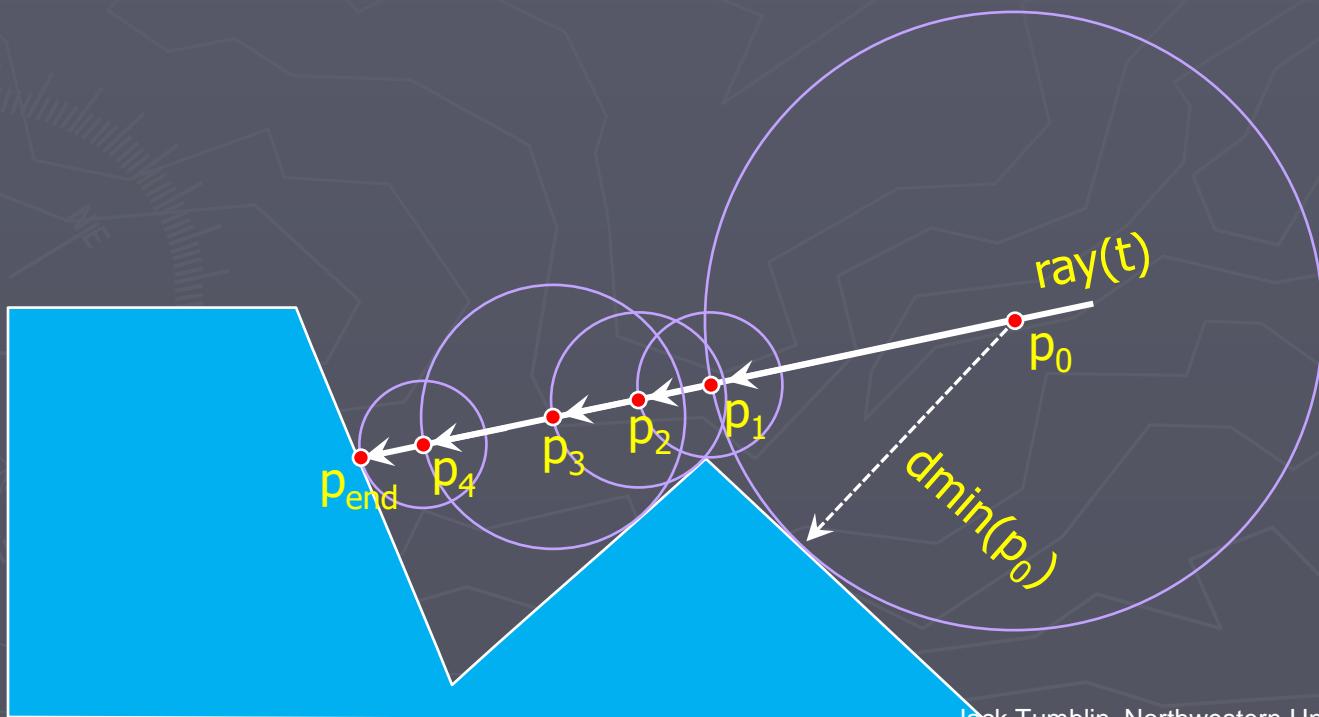
- ▶ Move(or “march”) p along ray by d_{min} :
$$p_{\text{new}} = p_{\text{old}} + d_{min}(p_{\text{old}})$$
- ▶ Repeat until we hit something! Or miss it entirely!



SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Key Ideas 4 ,5:

- ▶ Move (or “march”) p_n along the ray: $p_{n+1} = p_n + dmin(p_n)$
- ▶ Keep “marching” until p_n arrives at surface, or misses it
e.g. until $dmin(p_n) \approx 0$ or $dmin(p_n) \approx \infty$ (“too big”)

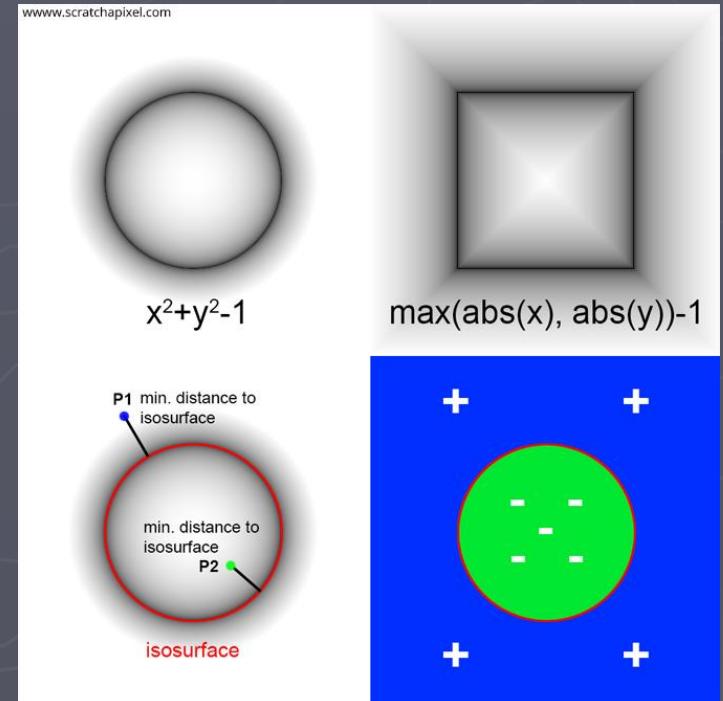


SPHERE III: ‘Ray Marching’, aka ‘Sphere Tracing’

Surprise! Ray-Marching works for *ANY* shape for which you can define a precise **dmin()** !

Try web-searches (also search image,video) on:

- Signed distance fields
- Jules Bloomenthal images
‘Unchained Geometry’
- Sphere Tracing
- Ray Marching
- Hypertextures



Other Interesting Implicit Surfaces

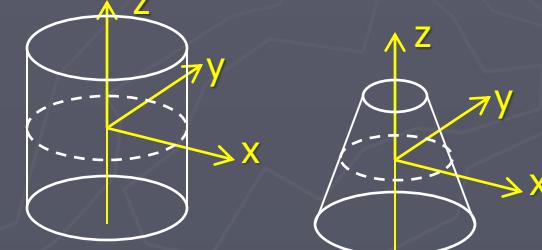
- ▶ Unit Cube (+/-1) at origin:

$$f(x,y,z) = \begin{cases} \text{if}(x^2 > \text{both } y^2 \text{ & } z^2): x^2 - 1 \\ \text{if}(y^2 > \text{both } x^2 \text{ & } z^2): y^2 - 1 \\ \text{if}(z^2 > \text{both } x^2 \text{ & } y^2): z^2 - 1 \end{cases}$$

- ▶ Unit-radius Cylinder at origin:

- ▶ Generalized cylinder:

make top disc's radius adjustable: set to zero to create a cone



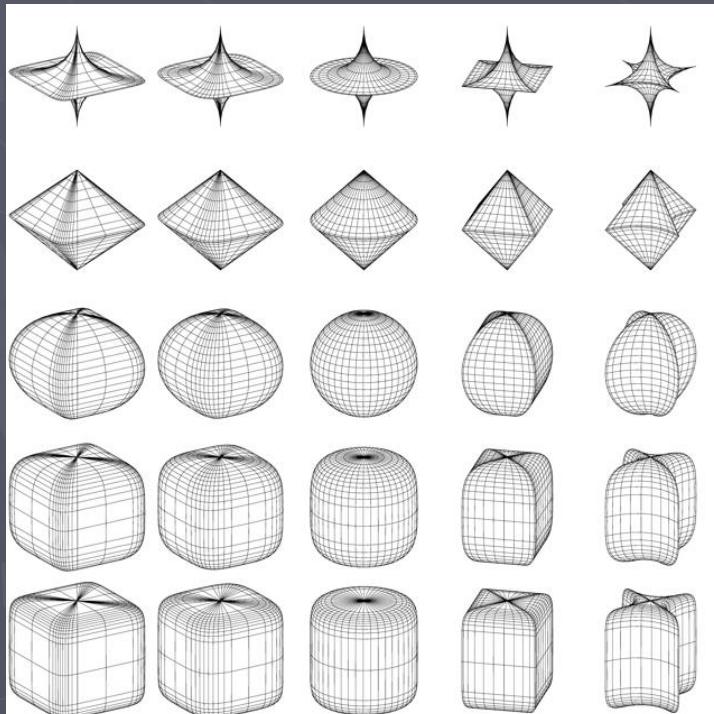
- ▶ Torus? Blinn's 'blobbies'? Meta-Balls?
Quadrics? SuperQuadrics?
See Lengyel reading, FS Hill reading, see
web-search: ray <shapename> intersection test

SuperQuadratics

3D spheres raised to higher powers:

$$f(x,y,z) = x^r + y^s + z^t - 1$$

for any integer r,s,t



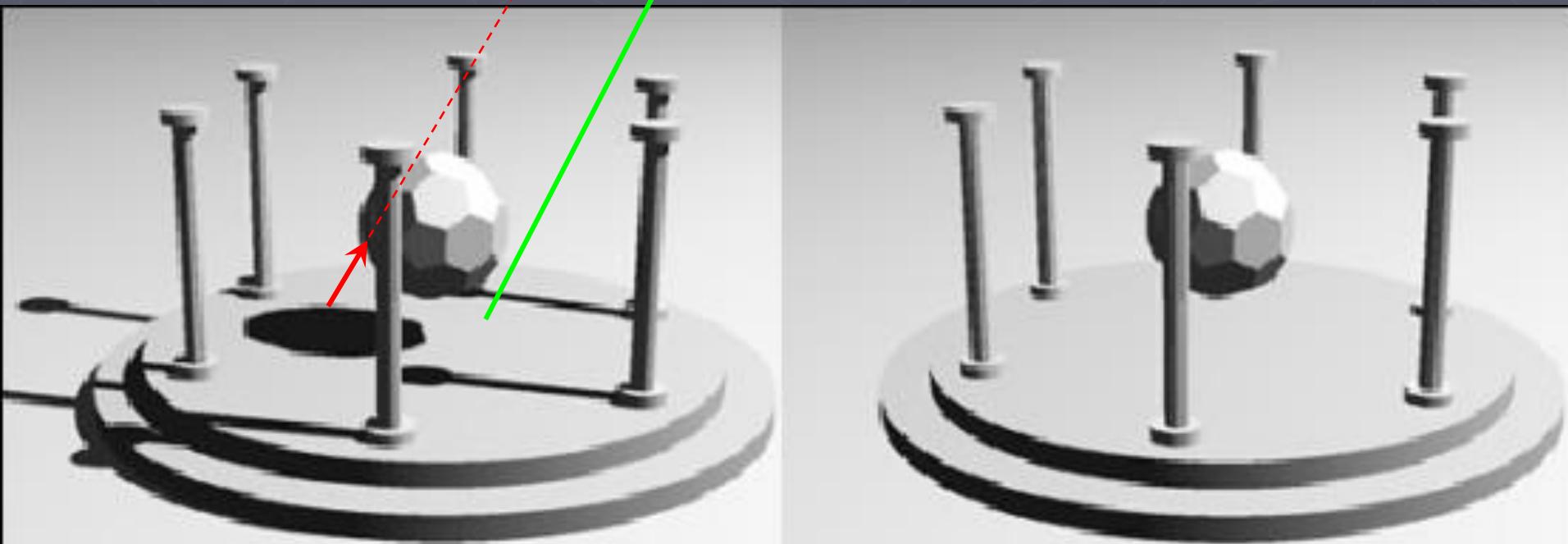
Light your First Sphere? soon...

Phong Basics:

- ▶ First, Make a fixed-color sphere ('diffuse' color) K_d
- ▶ Next, Find Surface normal N . Set color: $\text{clamp}(n_x, n_y, n_z)$
- ▶ Next, Finds N dot-product with vector L
(L == vector towards light from hit-pt.)
- ▶ Compute diffuse-only screen color: $K_d N \cdot L$
- ▶ Easy yet POWERFUL COLORING ALTERNATIVES:
 - 'procedural' textures
 - 'solid' textures $r(x,y,z)$, $g(x,y,z)$, $b(x,y,z)$
 - 3D checkerboards, 3D ramps, sinusoids, etc.
 - Perlin noise, turbulence functions....

Do Shadows First -- Very Easy!

- ▶ Important height cues for disjoint objects
- ▶ Simple in ray-tracing, hard in WebGL
- ▶ How? Trace a 'shadow' ray from hit-point to lamp position:
IF this 'shadow' ray is blocked by ANY CGeom object,
THEN no light from that lamp reaches that hit-point



Code Organizing 1: More Shapes

- ▶ **CScene** Class:
make **COLLECTION** of CGeom objects
(Simplest? Use `CGeom item[MAX_SHAPES];`)
- ▶ **CGeom** class: expand to describe *any* shape:
-- Ideally, make CGeom-derived classes / prototypes
-- OR -- Too exotic? Use this simple (but ugly) way:

- Add data member: `shapeType`; values from

```
const GEOM_NONE      =0; // no shape; empty/invisible object
const GEOM_GNDPLANE =1; // xy plane with lines
const GEOM_DISK       =2; // flat 2-sided zero-thickness disk
const GEOM_SPHERE     =3; // unit sphere @ origin
const GEOM_CUBE       =4; // unit cube @ origin
const GEOM_CYLINDER   =5; // cone or cylinder
```

- Include *all* data members needed for all shapes:
ignore those not needed for current '`shapeType`'

Code Organizing 2: Tracing Refactored

Multiple CGeom objects?

Light sources?

Shadow rays?

Reflection rays?

YIKES!

Suddenly your `CScene.makeRayTracedImage();`
gets HUGE, gets MESSY, gets COMPLICATED!

*** Refactor:

divide collections of complex tasks
among several smaller simpler functions ***

Code Organizing 2: Tracing Refactored

- ▶ **GOAL:** create function(s) to trace one *given* ray, and to find a color at the ray endpoint:
 - What's the **input**? → One 'eye' ray through one CCamera pixel-area;
 - What's the **output**? → The ray's on-screen color (for **CImgBuf** image)
 - Be sure to choose functions suitable for recursion!
- ▶ **STRATEGY (recommended):**
 - 1) **CScene.traceRay(r)** : trace one given ray **r** thru the entire **CScene** contents. Find all the hits, and where.
 - 2) **CScene.findShade()**
From that ray and all its hits, find color. As needed, trace shadow rays, reflection rays, transparency rays; apply lighting, materials and textures to find ray color(s).
 - 3) **CRay prototype** already describes rays; now create a **CHit prototype** to hold one ray/scene collision point

CHit: Holds Ray 'Collision' Pt

Create **CHit** object prototype to:

- ▶ Hold the **traceRay()** results: (for **findShade()**)
 - 't' value; traced ray length (to object intersection point)
 - 'hitPoint'; ray/object intersection point **in world space**
 - 'hitGeom'; (reference to) CGeom object pierced by ray
- ▶ Describe **surface conditions** at ray-piercing point:
 - Hold index (or indices) for material(s) used there;
 - Hold all geometry needed for shading: view vector, surface normal, shadow rays, transparency rays, etc.
- ▶ Acts as input and output for **findShade()**

-- CHit objects keep recursive tracing simple --

Try This: Split up the big fcn CScene.makeRayTracedImage()

How should it delegate tracing & color-finding?

- ▶ **Currently** it's a **Great Big Loop**; it makes all pixels in the image buffer (global var: CImageBuf g_myPic)
 - Calls **CCamera.makeEyeRay()** for each pixel to make all of that pixel's 'eye' rays (**CScene.rayNow**) (No change)
 - Calls **CScene.traceRay(inRay, myHit)** to test **all** CGeom objects in our CScene for ray-intersection. Modifies CHit object '**myHit**' to describe 'hit point' nearest the camera (holds **what** we hit & **where** we hit it).
 - Calls **CScene.findShade(myHit);** to assess screen color at the end of the ray. It completes the CHit object, gets the final color for the eye ray. (and may use recursion to do it! ...)

REVISED CScene.makeRayTracedImage()

- ▶ **How should it** delegate tracing & color-finding?
- ▶ a Great Big Loop; makes all pixels in the image buffer (global var: CImageBuf myPic)

- Calls CCamera.makeEyeRay() for each pixel to make all of that pixel's 'eye' rays (CScene.rayNow)

Trace the eye Ray, find its color : USE THIS PAIR RECURSIVELY

- Calls **CScene.traceRay(inRay, myHit)** to test **all** CGeom objects in our CScene for ray-intersection. Modifies CHit object '**myHit**' to describe 'hit point' nearest the camera (holds **what** we hit & **where** we hit it).
- Calls **CScene.findShade(myHit);** to assess screen color at the end of the ray. It completes the CHit object, gets the final color for the eye ray. (and may use recursion to do it! ...)

CScene.findShade()

(may grow and need further refactoring later)

Find the screen-color from the nearest 'hit-point' for this ray:
compute OpenGL-like shading (Phong light-sources and materials),
but improve it with shadows, reflections, and transparency.

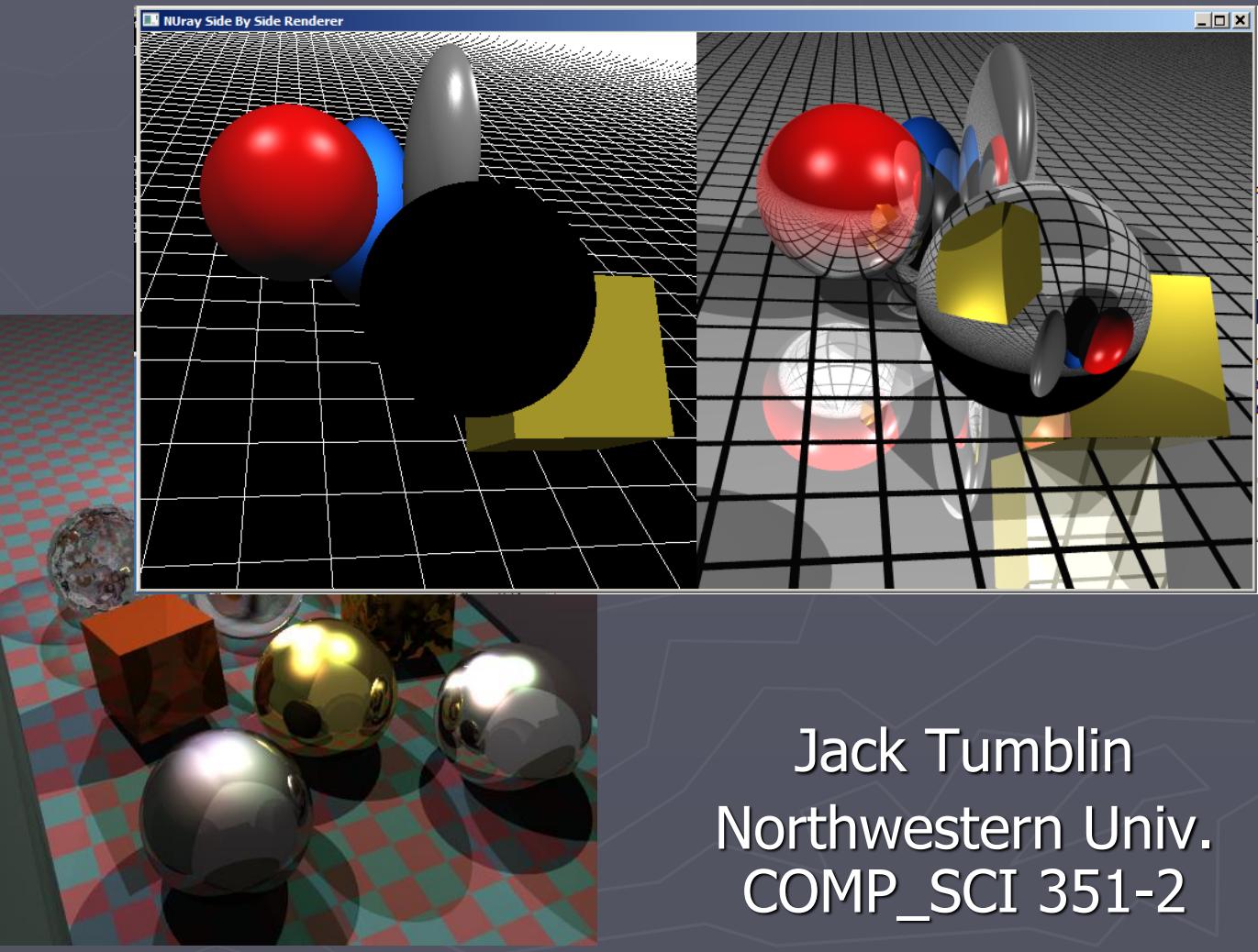
- ▶ Refer to the CGeom object we hit:
 - Find/retrieve its materials properties at the hit-point (K_ambient, K_diffuse, K_specular, K_emissive, K_shininess)
 - Find/retrieve the shading vectors at the hit-point: surface normal, view vector, light-direction vectors, reflection vectors, etc.
- ▶ Test the hit-point's lighting: what light sources illuminate this surface point?
 - Create a shadow ray from hit-point to each light source
 - call CScene.traceRay() to find what each shadow ray hits, if anything. Nothing? no shadow at hit-point; enable this light
- ▶ Find the color of a reflected ray:
 - Create a reflection ray at the hit-point (use surface normal **N** and view **V** vector)
 - Call CScene.traceRay() to find **what** and **where** the reflected ray hits (if anything)
 - Call CScene.findShade() to **find the color** at the reflection-ray's endpoint.

END

END

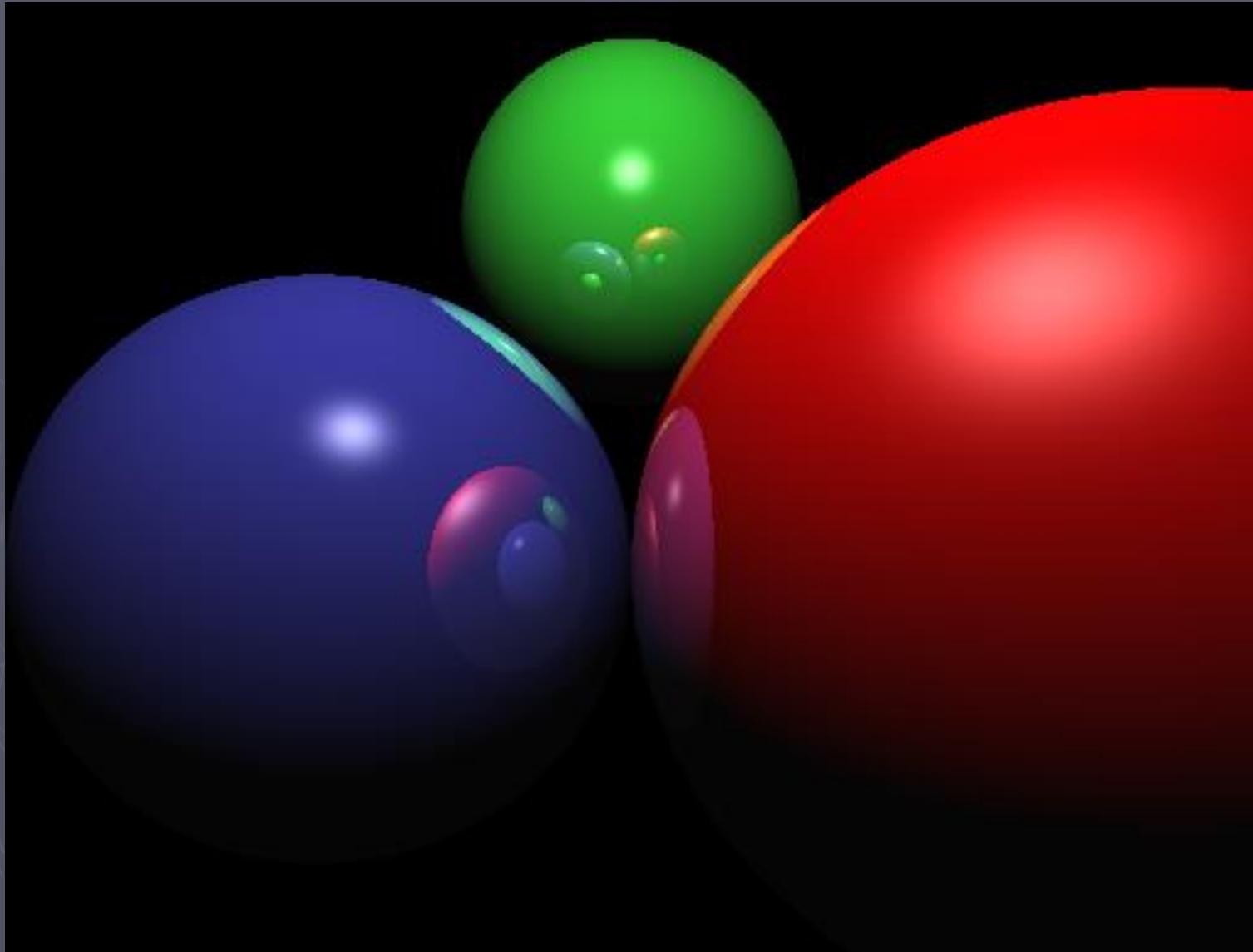


Ray Tracing E: Normals, Lighting, & Mirrors



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

Next!: Phong Lighting + Reflection



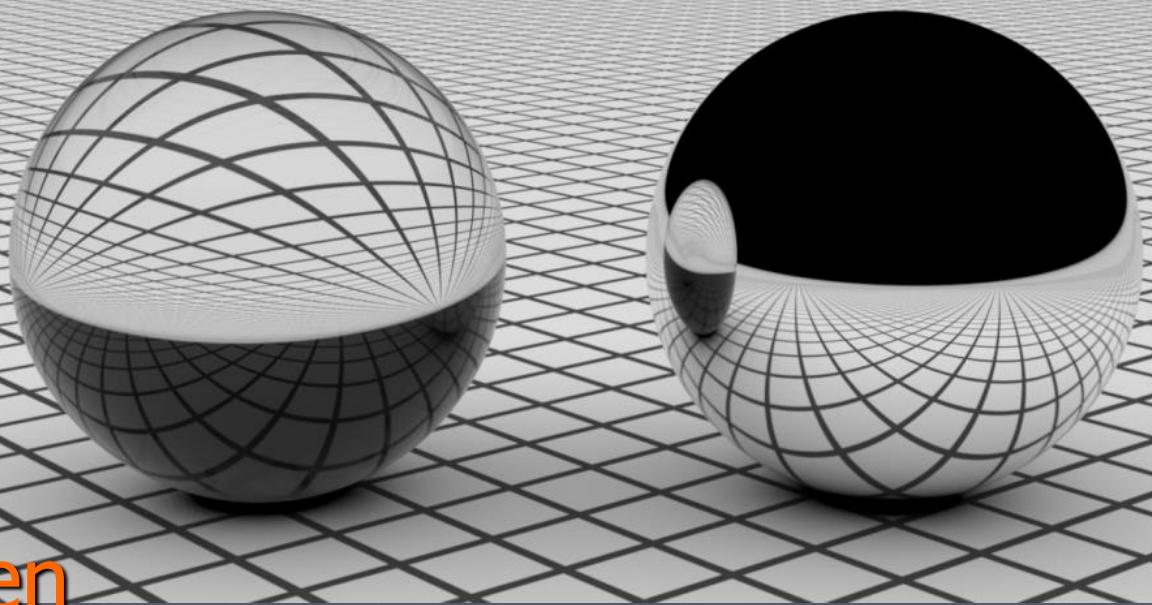
Our Plan:

Write code for:

- ▶ Camera, then
- ▶ Plane, then
- ▶ Antialiasing, then
- ▶ Transform, then
- ▶ Basic Disk and Sphere, then Shadow, then
- ▶ Phong Lighting,
- ▶ Reflection, then Transparency, then

Further Step-by-Step Refinements:

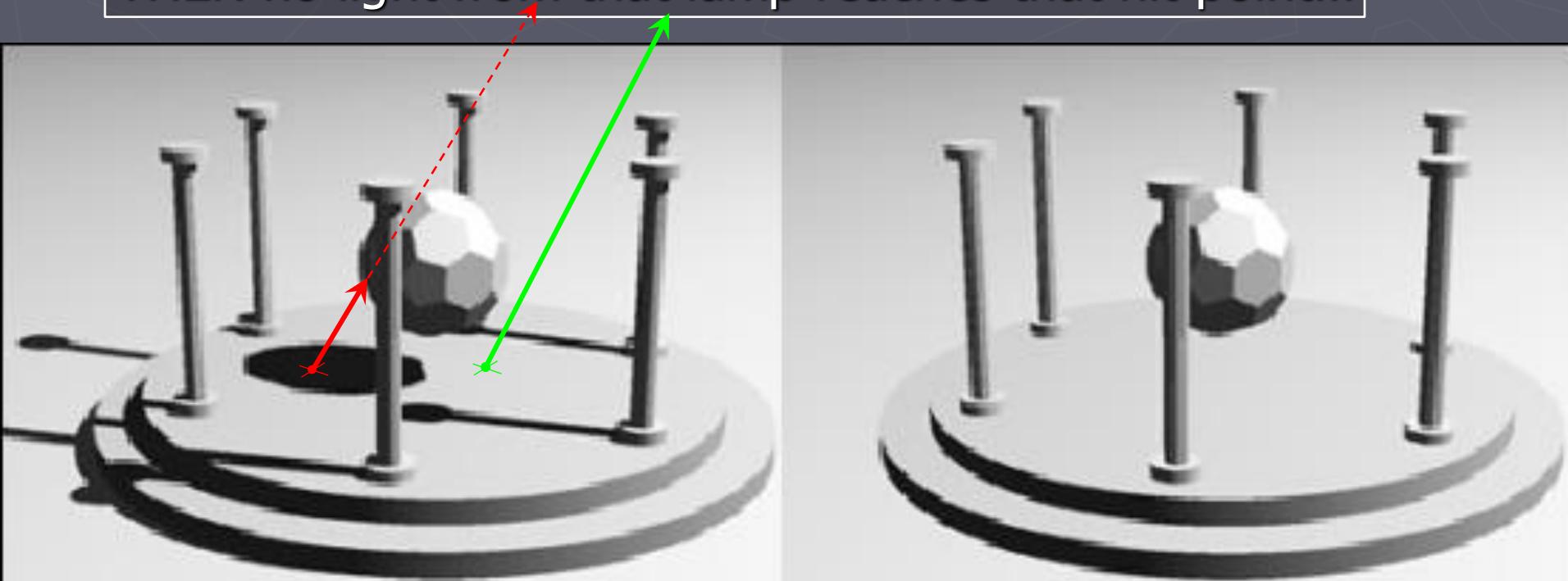
Lighting → Materials → Texture → CSG, etc. ...



*** **MAKE PICTURES AT EVERY STEP** ***

Shadows Matter, Yet Very Easy!

- ▶ Important height cues for disjoint objects
- ▶ Simple in ray-tracing, hard in WebGL (skip it for preview)
- ▶ How? Trace 'shadow' ray from hit-point to lamp position(s):
IF this 'shadow' ray is blocked by ANY CGeom object,
THEN no light from that lamp reaches that hit-point...



Confused? Lost in the starter-code?

- ▶ **STOP! Read!** In Reading Materials
(Canvas→Home page→ RayTracing module→ Weeks1-2 Reading, Weeks3-4 Reading),
find the PDFs for
Week1-4_FS_HillComputerGraphicsChap14...
- ▶ **READ CAREFULLY:** Their ray-tracer matches my notes and ray-tracer quite well, & explains all ray-tracer principles in detail.

Concise Ray-Tracer Outline

- FS Hill "Computer Graphics Using OpenGL"
2nd Edition, Chap 14, pg 736, Fig 14.4:

define the objects and light sources in the scene

set up the camera

```
for(int r = 0; r < nRows; r++)
    for(int c = 0; c < nCols; c++)
    {
```

- 1. Build the rc-th ray*
- 2. Find all intersections of the rc-th ray with objects in the scene*
- 3. Identify the intersection that lies closest to, and in front of, the eye*
- 4. Compute the "hit point" where the ray hits this object, and the normal vector at that point*
- 5. Find the color of the light returning to the eye along the ray from the point of intersection*
- 6. Place the color in the rc-th pixel.*

```
}
```

Nearly Identical Organization

Our Starter Code:

- ▶ Classes: CScene, CRay, CCamera
- ▶ Global Vars: myScene (contains rayCam), myPic (image buffer)
- ▶ CScene.
 makeRayTracedImage()
- ▶ CScene data member:
 CRay eyeRay;
- ▶ CScene.findShade()
 - Calls CScene.traceRay()
- ▶ Arrays of CHit objects
(I skip the containing class)

FS HILL book:

- ↔ Classes: Scene, Ray, Camera
- ↔ Global Vars: theScene, theCamera
?! (has no image buffer!)
- ↔ Camera::Raytrace()
- ▶ Camera data member:
 Ray theRay;
- ↔ Scene::shade()
 - Calls Scene::FindFirstHit()
- ↔ Intersection class:
 holds array of 'HitInfo' objects

Review: how to color 'Eye' Rays

1. Our CScene.**makeRayTracedImage()** function steps through each pixel in the CImgBuf image.
2. For each pixel, **CCamera** object makes an **eye ray** that finds the pixel's color.
3. Trace one eye ray:
find **what** the ray hits, and **where**;

```
myScene.traceRay(eyeRay, hitList);  
// myScene: our CScene object that holds everything  
// hitList: holds array of CHit objects
```

traceRay() fcn. builds an array of CHit object(s)

- FS Hill: "save ALL the ray intersections in a list of 'HitInfo' objects held in the Intersection class (see pg 745), then find, copy, and use the nearest HitInfo object."

4. Write **ONE FUNCTION** to find ray color ...

Shading: Do it (almost) all in World-Space

5. How? Call the **CScene** fcn: **findShade(HitList)** to:

- a) select & complete the suitable (nearest) **CHit** object.

When finished, that **CHit** object will hold all this:

(see `JT_Tracer0-Scene.js` and FS Hill, pg 746, **HitInfo** class)

- **t0**; (floating point 'hit time' for traced ray)
- **hitGeom**; (which CGeom object we hit)
(HINT: can use an integer array **index** for `CScene.GeomList` array)
- **isEntering**; (entering or leaving object? boolean)
- (hit point in WORLD space) = **orig + t0*dir**
- (hit point in MODEL space) = **torig + t0*tdir**
- (surface normal in *WORLD* space) found like this:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2Model}]^T \mathbf{N}_{\text{model}}$$

- b) **THEN:** use Phong lighting(and more) to find Ray's final color!

Shading: Do it (almost) all in World-Space

5. How? Call the CScene fcn: `findShade(HitList)` to:

a) find & complete the nearest CHit object.

When finished, that CHit object will holds
(see FS Hill, pg 746, HitInfo class)

- (floating point 'hit time' for traced ray) `t0`;
- (the CGeom object we hit)
(you can use an integer array index for CScene.GeomList array)
- (entering or leaving object? boolean) `isEntering`;
- (hit point in WORLD space) = `orig + t0*dir`
- (hit point in MODEL space) = `torig + t0*tdir`
- (surface normal in *WORLD* space) found like this:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2Model}]^T \mathbf{N}_{\text{model}}$$

b) **THEN:** use Phong lighting(and more) to find Ray's final color!

Shading: Do it (almost) all in World-Space

5. Call ~~[r,g,b] = findshading()~~

!? uhh-- WHUT ?!

How did we figure out
the WORLD-space surface normal N_{world}

from a MODEL-space normal N_{model} !?

& Why is THIS
the solution?

(point in MODEL space) $t_{\text{orig}} + t_0 * t_{\text{dir}}$

(surface normal in WORLD space)

$$N_{\text{world}} = [\text{worldRay2Model}]^T N_{\text{model}}$$

AND (the Ray's final color!)

Surface **Normals** in World Space

- ▶ **RECALL:** We defined all CGeom objects each in their own local 'model' coord systems
 - CGeom shapes are *ALWAYS* centered at origin &
 - *ALWAYS* fixed, unit-sized (e.g. sphere radius=1)
- ▶ How do World-coord-system rays trace it? How do we set the object's position? Size? Orientation?
 - Easy: put it all in the [`worldRay2Model`] matrix; (transforms rays from world- to model-space)

▶ **BIG PROBLEM!**

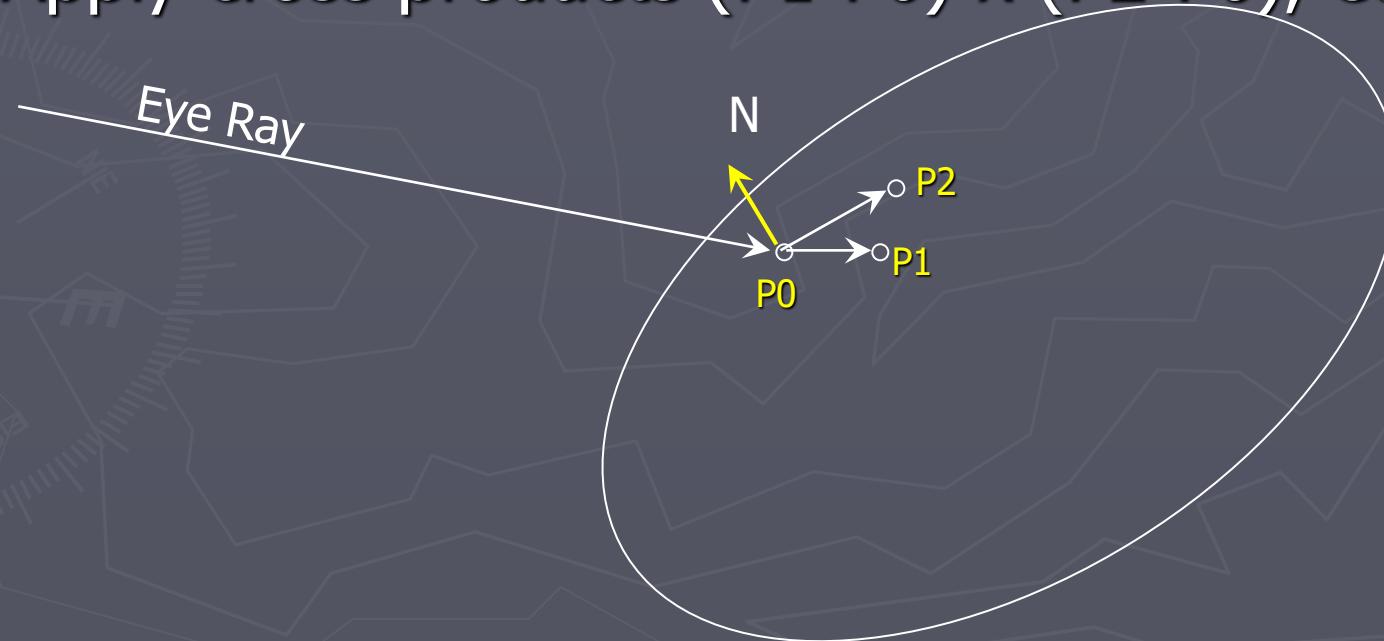
- we can **FIND** CGeom surface normals **N** easily in 'model' coord system, but
- But we **NEED** them in the 'world' coord system!

Normals? NAÏVE (bad!) SOLUTION:

► Find it by CONSTRUCTION:

Find and transform some model-space points **backwards** to world coords to find normal:

- Find 3 nearby non-collinear points on surface,
- Apply cross products $(P_1 - P_0) \times (P_2 - P_0)$, etc...



Normals? NAÏVE (bad!) SOLUTION:

► Find it by CONSTRUCTION:

Find and transform some model-space points **backwards** to world coords to find normal:

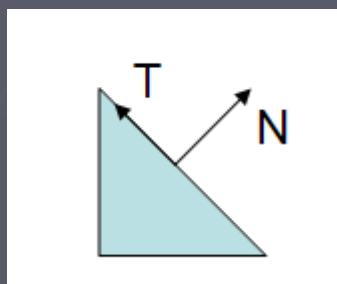
- Find 3 nearby non-collinear points on surface,
- Apply cross products $(P1-P0) \times (P2-P0)$, etc...

!!!UGLY!!!

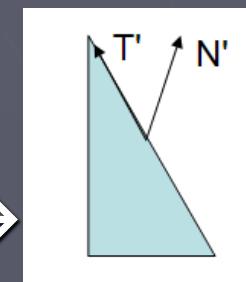
- REQUIRES 2 extra model-space points,
- REQUIRES CGeom to hold the *inverse* of the `worldRay2model` matrix:
?!? `Mat4 modelRay2world` also?!?! NOO!!!!

A Better Way to find Normals (1)

- RECALL: scale transform operations can *distort* the surface-normal vectors:



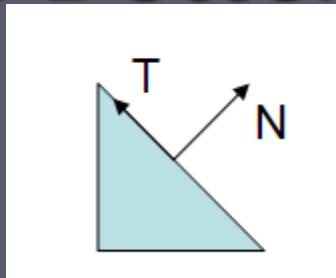
→ scale x coords by 0.5 →



(from: <http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/>)
(detailed analysis? See http://www.songho.ca/opengl/gl_normaltransform.html)

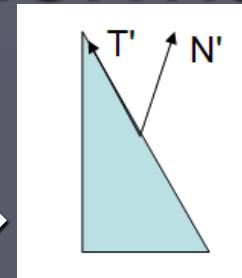
- THUS as WebGL transforms **vertices** by matrix **M**
we *also* transform **normals** by matrix **M^{-T}**
- This **inverse-transpose matrix M^{-T}** keeps normals perpendicular to surface for any directional scaling.
- How do we apply this to ray-tracing? **BACKWARDS!**

A Better Way to find Normals (2)



surface-normal vectors:

→ scale x coords by 0.5 →



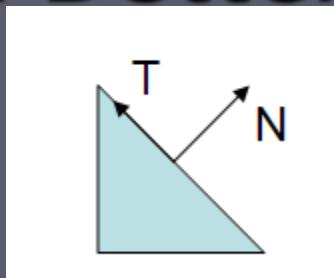
RECALL: in WebGL we transform **vertices** by matrix **M**
and we transform **normals** by matrix **M^{-1}**

- ▶ Ray tracing transforms **rays** from world \rightarrow model coords by the *inverse* of WebGL's vertex-transform matrix: use **M^{-1}**
(In my ray tracer, the **M^{-1}** matrix is [worldRay2model])
- ▶ To transform **rays** to world \leftarrow model coords, just use **M**,
and in my ray tracer: **(M) = ([worldRay2model] $^{-1}$)**
- ▶ To transform **normals** to world \leftarrow model coords, Use **M^{-T}** .

How? In my ray tracer: **$M^{-T} = ([worldRay2model]^{-1})^{-T}$**
since **$M^{-T} = ((M)^{-1})^T = ([worldRay2model]^{-1})^{-1})^T$**

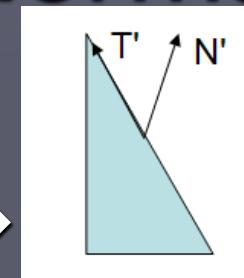
Look!

A Better Way to find Normals (3)



surface-normal vectors:

→ scale x coords by 0.5 →



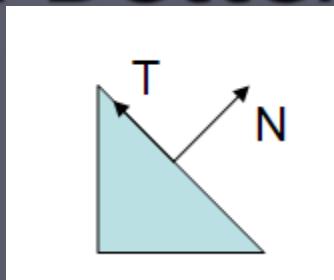
RECALL: in WebGL we transform **vertices** by matrix **M**
and we transform **normals** by matrix **M^{-T}**

- ▶ Ray tracing transforms rays from world \rightarrow model coords by the *inverse* of WebGL's vertex-transform matrix: use M^{-1} (In my ray tracer, the M^{-1} matrix is [worldRay2model])
- ▶ To transform rays to world \leftarrow model coords, just use M , and in my ray tracer: $(M) = ([worldRay2model]^{-1})$
- ▶ To transform **normals** from world \leftarrow model coords, Use **M^{-T}** .

How? In my ray tracer: $M^{-T} = ([worldRay2model]^{-1})^{-T}$

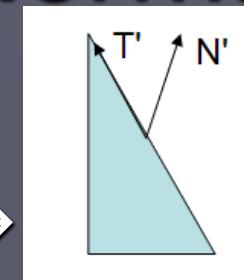
since $M^{-T} = ((M)^{-1})^T = ([worldRay2model]^{-1})^{-1})^T$
that double-inverse cancels, and ...

A Better Way to find Normals (4)



surface-normal vectors:

→ scale x coords by 0.5 →



RECALL: in WebGL we transform **vertices** by matrix **M**
and we transform **normals** by matrix **M^{-1}**

- ▶ Ray tracing transforms rays from world \rightarrow model coords by the *inverse* of WebGL's vertex-transform matrix: use **M^{-1}** (In my ray tracer, the **M^{-1}** matrix is `[worldRay2model]`)
- ▶ To transform rays to world \leftarrow model coords, just use **M**, and in my ray tracer: **$(M) = ([worldRay2model]^{-1})^{-1}$**
- ▶ To transform **normals** from world \leftarrow model coords, Use **M^{-1}** .

How? In my ray tracer: $M^{-1} = ([worldRay2model]^{-1})^{-1}$

since $M^{-1} = ((M)^{-1})^T = ([worldRay2model]^{-1})^{-1}T$

where **$M^{-1} = ([worldRay2model]^T)$**

A Better Way to find Normals (5)

HOW TO DO THAT in your code:

To transform ray **model**-space normal vector $\mathbf{N}_{\text{model}}$
to ray **world**-space normal vector $\mathbf{N}_{\text{world}}$

Just apply *TRANSPOSE* of the **worldRay2model** matrix:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2model}]^T \mathbf{N}_{\text{model}};$$

(COOL! fast and easy!
now we can do ALL our lighting calcs in world-space!)

A Better Way to find Normals (6)

HOW TO DO THAT in your code:

To transform ray **model**-space normal vector $\mathbf{N}_{\text{model}}$
to ray **world**-space normal vector $\mathbf{N}_{\text{world}}$

Just apply *TRANSPOSE* of the **worldRay2model** matrix:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2model}]^T \mathbf{N}_{\text{model}}$$

?Using **vec4** and **mat4**? (4x4 Matrix)?

!CAUTION! -- ! UGLY SURPRISE !

- ▶ **worldRay2model** 4th column has no effect on normal vector $\mathbf{N}_{\text{model}}$
- ▶ But in $[\text{worldRay2model}]^T$ the 4th row would **disastrously corrupt** $\mathbf{N}_{\text{model}}$!

$$\begin{bmatrix} a & b & c & tx \\ d & e & f & ty \\ g & h & i & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ tx & ty & tz & 1 \end{bmatrix}$$

ANSWER: mat3, or set bottom row to [0 0 0 1]!

A Better Way to find Normals (7)

HOW TO DO THAT in your code:

To transform ray **model**-space normal vector $\mathbf{N}_{\text{model}}$
to ray **world**-space normal vector $\mathbf{N}_{\text{world}}$

Just apply *TRANSPOSE* of the worldRay2model matrix:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2model}]^T \mathbf{N}_{\text{model}}$$

?Using **vec4** and **mat4**? (4x4 Matrix)?

!CAUTION! -- ! UGLY SURPRISE !

- ▶ worldRay2model 4th column has no effect on normal vector $\mathbf{N}_{\text{model}}$
- ▶ But in $[\text{worldRay2model}]^T$ the 4th row would disastrously corrupt $\mathbf{N}_{\text{model}}!$

$$\begin{bmatrix} a & b & c & tx \\ d & e & f & ty \\ g & h & i & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ tx & ty & tz & 1 \end{bmatrix}$$

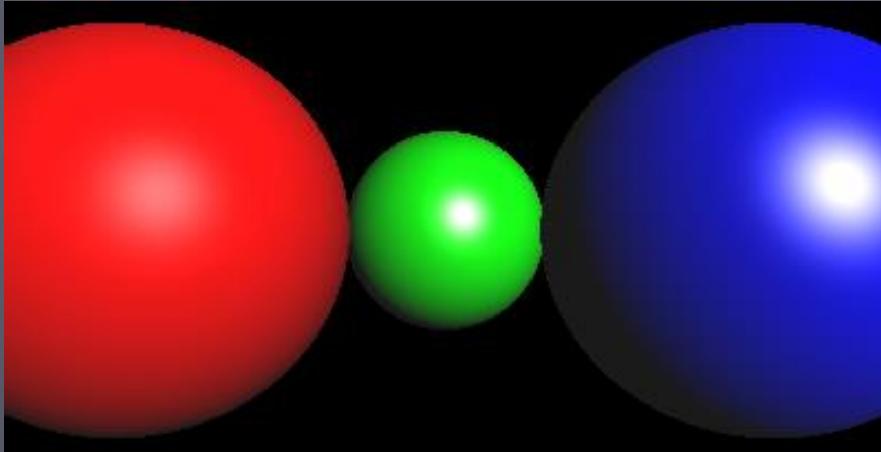
ANSWER: mat3, or set bottom row to [0 0 0 1]!

Week 4 Goals

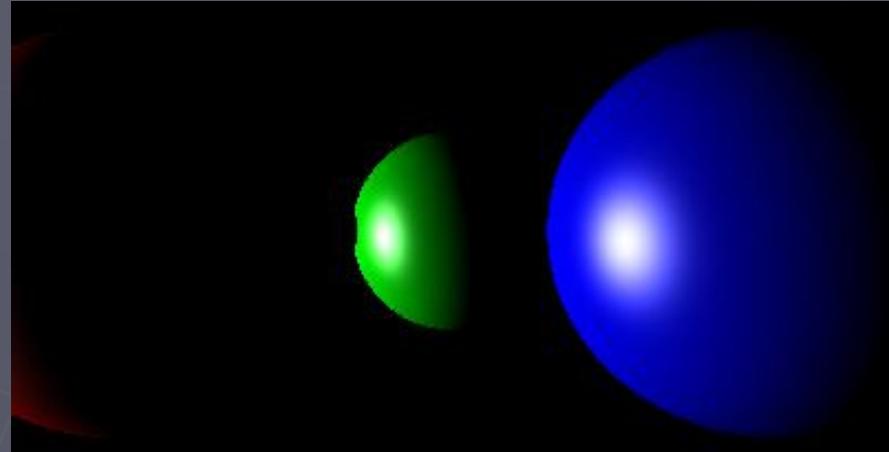
- ▶ 1) Make World-Space Surface Normals → OpenGL-like Phong Lighting
 - Transform normals? → See CS 351-1 Reading...
 - CMatl objects: ambient, diffuse, specular, & shinyness members
 - Move lights with Mouse, Keyboard? More GUIbox member fcns!
- ▶ 2) Recursion: Show multiple reflections among objects;
 - 1 or more Grid-plane objects, 2 or more Spheres (try 3: )
 - ORGANIZE:  traceRay() calls findShade(), calls shade()
 - Extra Credit: transparency/refraction too!
- ▶ 3) At least one 'Solid' (or 'Procedural') Texture
 - CHit: add member(s) for model-space hit-point, use it in
 - CMatl: model-space 3D color fcns: R(x,y,z) G(x,y,z) B(x,y,z)
 - Functions for Concentric Shells, Line→Grid→Cage, Checkerboards & other 'easy' textures? see FS Hill, Chap 14.8, pp. 770-773

Ray Casting: *without* shadows

Right-side light:



Left-side light:

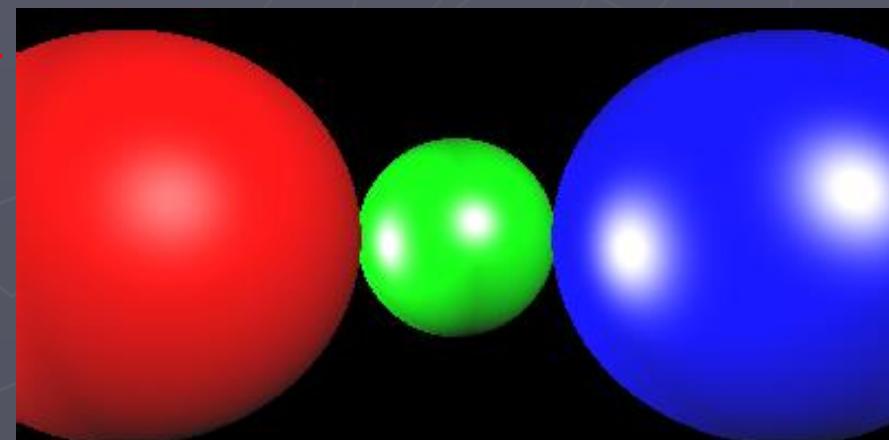


Combined lights:



ADD ***all*** their diffuse
& specular terms

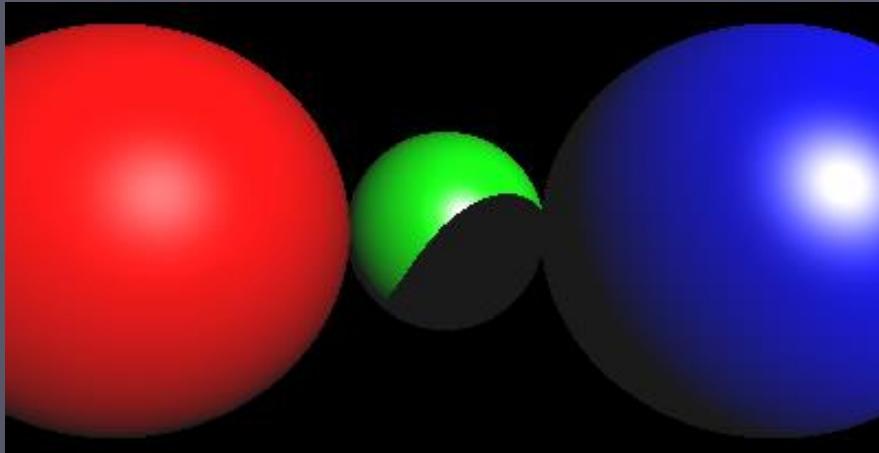
BUT use ***only one*** of the
ambient terms. (do you know why?)



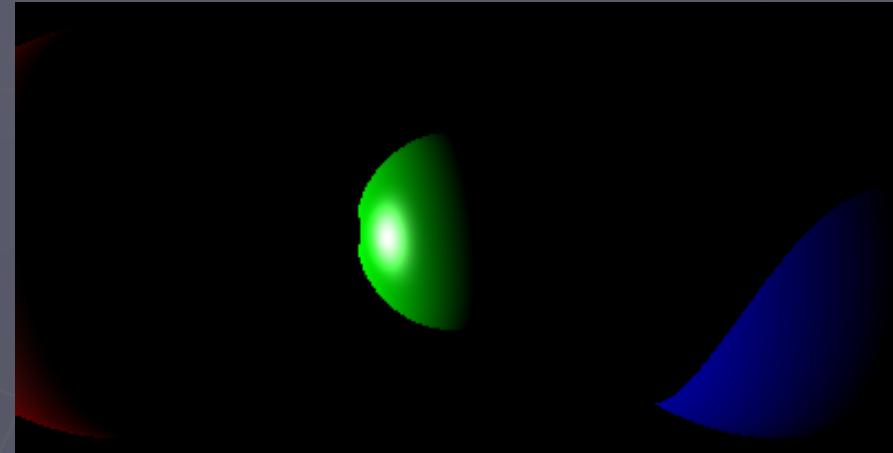
(Images From Univ. Delaware)

Ray Casting: *with* shadows

Right-side light



Left-side light



Look very closely at
how these shadows combine.

SURPRISE!

**shadows *DON'T BLOCK*
ambient light!** (do you know why?)



(Images From Univ. Delaware)

Ray Color? Start with Phong Lighting

Who finds ray color? which function?

I Recommend: `cScene` member function

`CScene.findShade(myHitList)`

IMPORTANT: Compute and Use
ALL vectors for Phong Lighting
in the **world** coord system:

V: View Vector (to eye; negative of eye ray: `-dir`)

L: 'Lamp' or Shadow vector (one to each light)

N: Surface Normal vector (in World-Space)

Later: **Mir**: mirror-reflected ray

Tr : transmitted ray, for transparency

Phong Lighting Step 1: Find Scene Vectors L,N,V

To find **Ray Color**
at hit-point **P (start)**:

1) Find all 3D scene vectors first:

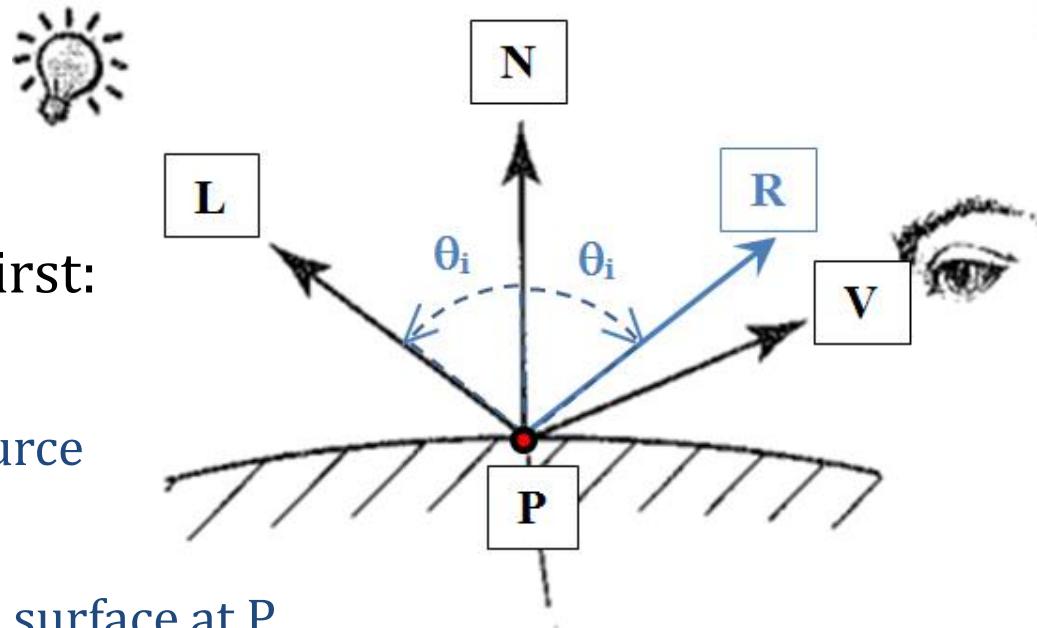
a) Light Vector **L**:
unit vector towards light source

b) Normal Vector **N**:
unit vector perpendicular to surface at P

c) View Vector **V**:
unit vector towards camera eye-point (easy! it's **-eyeRay.dir**)

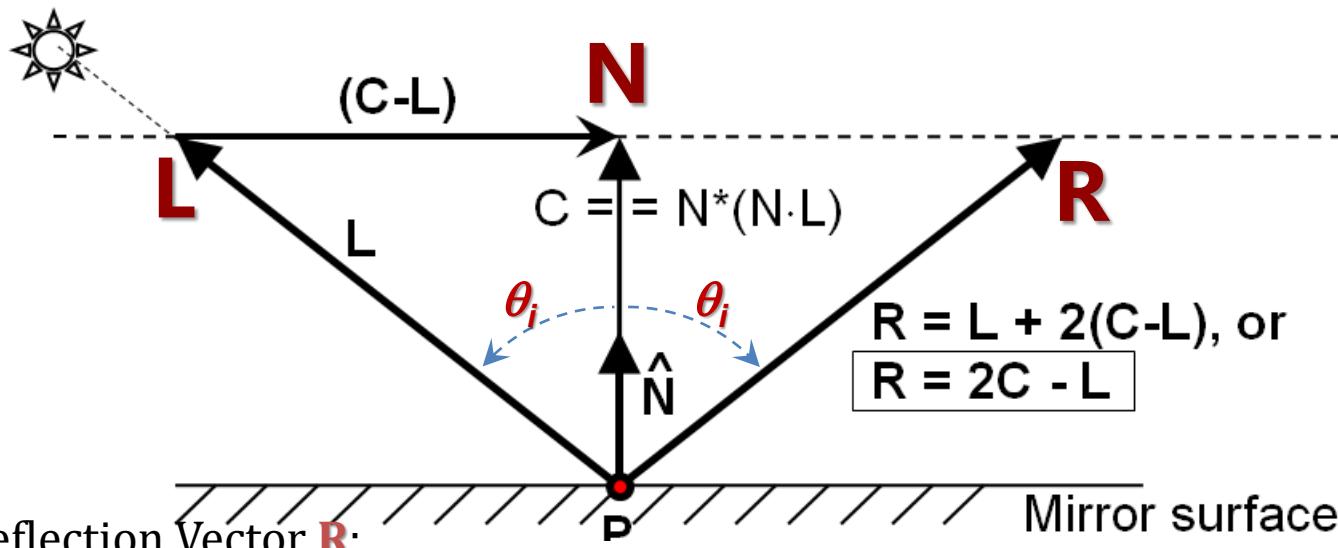
REMEMBER –all are vectors world-space coords!

Onwards to step 2: how do we find the Reflected-light Vector **R**?



Phong Lighting Step 2: Find reflection Vector R

To find **Ray Color**
at hit-point **P** (*cont'd*):



2) COMPUTE the Light Reflection Vector **R**:

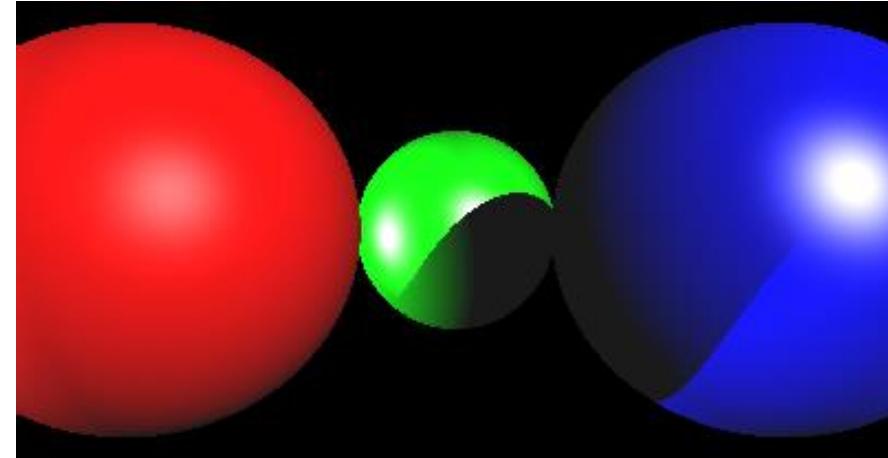
- Given unit light vector **L**, find lengthened surface normal **C**
$$\mathbf{C} = \mathbf{N} (\mathbf{L} \cdot \mathbf{N})$$
 (NOTE! does not require normalized **N** !)
- In diagram, if we add vector $2^*(C-L)$ to L vector we get R vector. Simplify:
$$\mathbf{R} = 2\mathbf{C} - \mathbf{L}$$
- Result:** reflected vector **R** GLSL-ES → See built-in '`reflect()`' function
(If **N** is a unit-length vector, then **R** vector length matches **L** vector length)

Phong Lighting Step 3: **Gather Light** **& Material Data**

To find **Ray Color**
at hit-point **P** (*cont'd*):

3) For each light source, gather:

- ▶ RGB triplet for **Ambient** Illumination
- ▶ RGB triplet for **Diffuse** Illumination
- ▶ RGB triplet for **Specular** Illumination



For each surface material, gather:

- ▶ RGB triplet for **Ambient** Reflectance
- ▶ RGB triplet for **Diffuse** Reflectance
- ▶ RGB triplet for **Specular** Reflectance
- ▶ RGB triplet for **Emissive** term(often zero)
- ▶ Scalar 'shininess' or 'specular exponent' term

$$\mathbf{I_a} \quad 0 \leq I_{ar}, I_{ag}, I_{ab} \leq 1$$

$$\mathbf{I_d} \quad 0 \leq I_{dr}, I_{dg}, I_{db} \leq 1$$

$$\mathbf{I_s} \quad 0 \leq I_{sr}, I_{sg}, I_{sb} \leq 1$$

$$\mathbf{K_a} \quad 0 \leq K_{ar}, K_{ag}, K_{ab} \leq 1$$

$$\mathbf{K_d} \quad 0 \leq K_{dr}, K_{dg}, K_{db} \leq 1$$

$$\mathbf{K_s} \quad 0 \leq K_{sr}, K_{sg}, K_{sb} \leq 1$$

$$\mathbf{K_e} \quad 0 \leq K_{er}, K_{eg}, K_{eb} \leq 1$$

$$\mathbf{Se} \quad 1 \leq S_e \leq \sim 100$$

Phong Lighting Step 4: Sum of All Light Amounts

To find **Ray Color**
at hit-point **P** (*cont'd*):

sum of each kind of light at **P**:

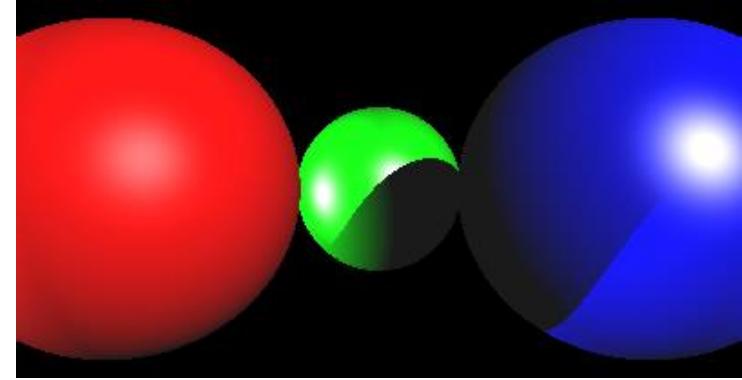
Phong Lighting = Emissive + Ambient + (Diffuse + Specular)*notShadow
SUMMED for all light sources,
BUT in shadows, both (Diffuse + Specular) drop to zero.

4) For the i-th light source, find:

(ignore shadows in your WebGL preview)

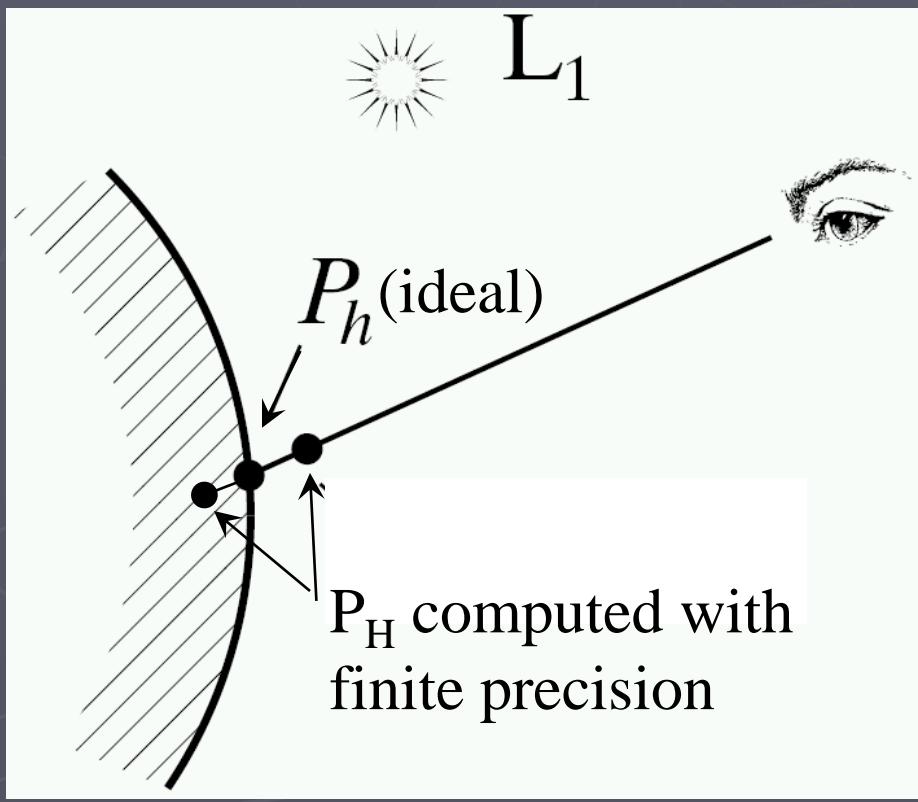
```
RGB= Ke + // 'emissive' material; it glows!  
      Ia*Ka; // ambient light * ambient reflectance  
if(!shadow) {  
    RGB += Id*Kd*Att*max(0,(N·L)) + // diffuse light * diffuse reflectance  
          Is*Ks*Att*(max(0,R·V))Se, // specular light * specular reflectance
```

- ▶ Distance Attenuation scalar: $0 \leq \text{Att} \leq 1$
 - ▶ Fast, OK-looking default value: $\text{Att} = 1.0$
 - ▶ Physically correct value: $\text{Att}(\mathbf{d}) = 1/(\text{distance to light})^2$ (too dark too fast!)
 - ▶ Faster, Nice-looking 'Hack': $\text{Att}(\mathbf{d}) = 1/\text{distance to light}$
 - ▶ OpenGL compromise: $\text{Att}(\mathbf{d}) = \min(1, 1/(c_1 + c_2 \cdot d + c_3 \cdot d^2))$
- ▶ 'Shinyness' or 'specular exponent' $1 \leq \text{Se} \leq \sim 100$ (large for sharp, small highlights)



UGLY BUG! Ray-Intersection Precision

- ▶ Finite precision: even **double** math results have tiny quantization errors ($\sim \text{result} * 10^{-17}$)
- ▶ At ray length '**t**', tiny errors misplace the computed hit-point **P_h**
- ▶ on, above, or below the surface. Uh oh.



UGLY BUG! Ray-Intersection Precision

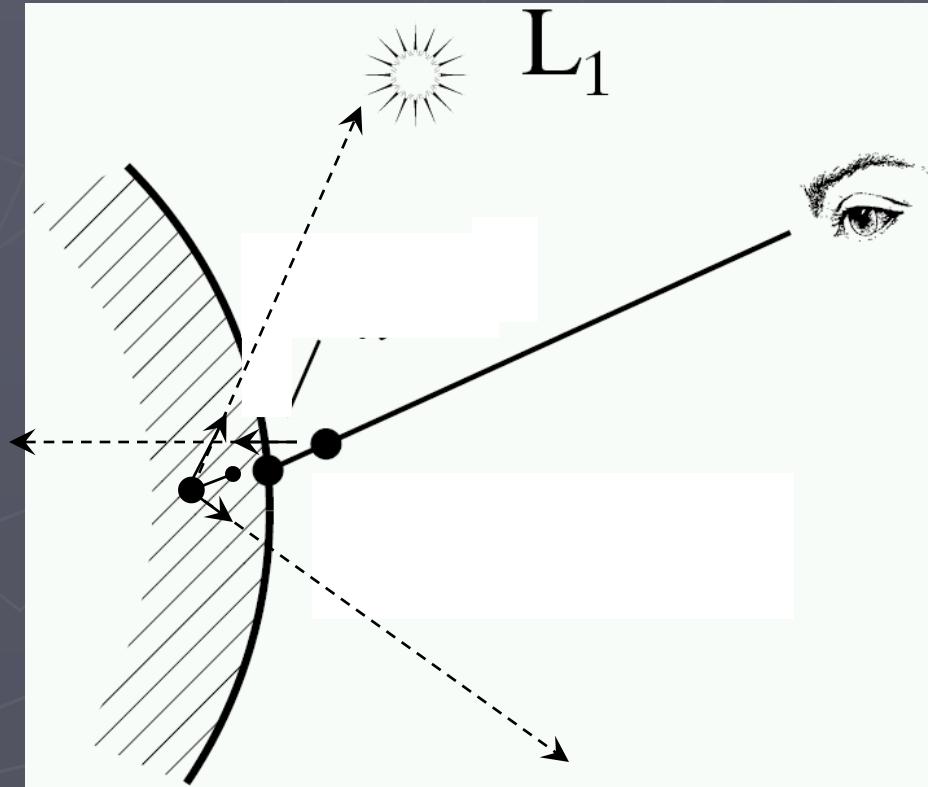
PROBLEMS!

Is Hit-point P_H *ON* or *BELOW* surface?

- All Shadow rays **fail!**
- Reflection ray **fails!**
- (surface blocks them)

► Hit-point ***ABOVE***?

- Transparency rays **fail!**
- (they need to begin *INSIDE* the surface)



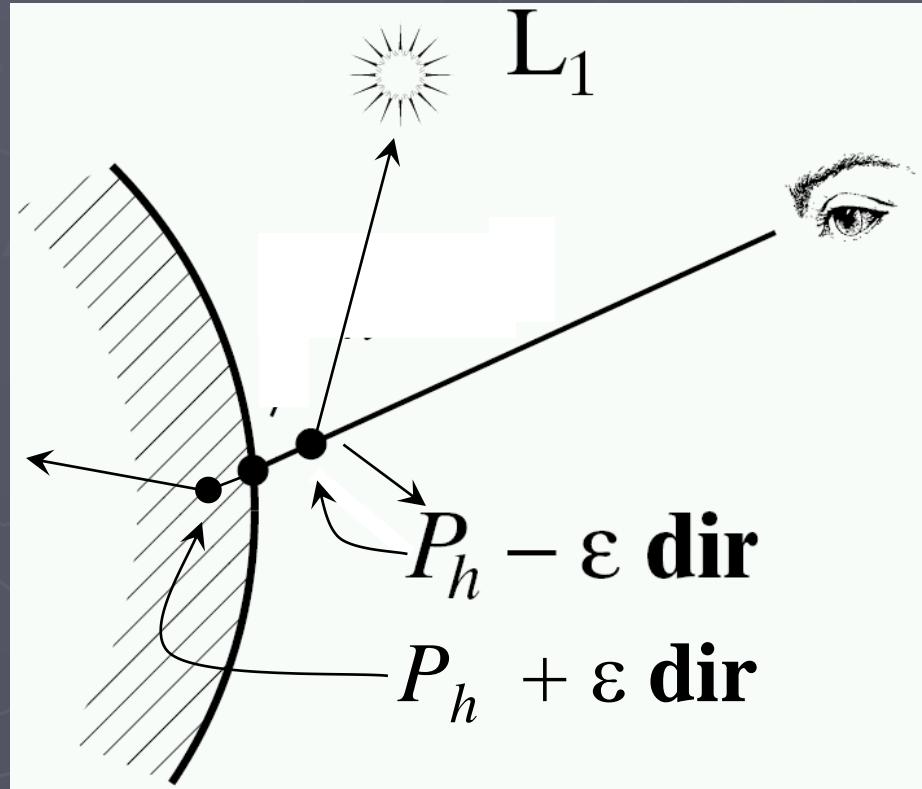
UGLY BUG! Ray-Intersection Precision

► SOLUTION:

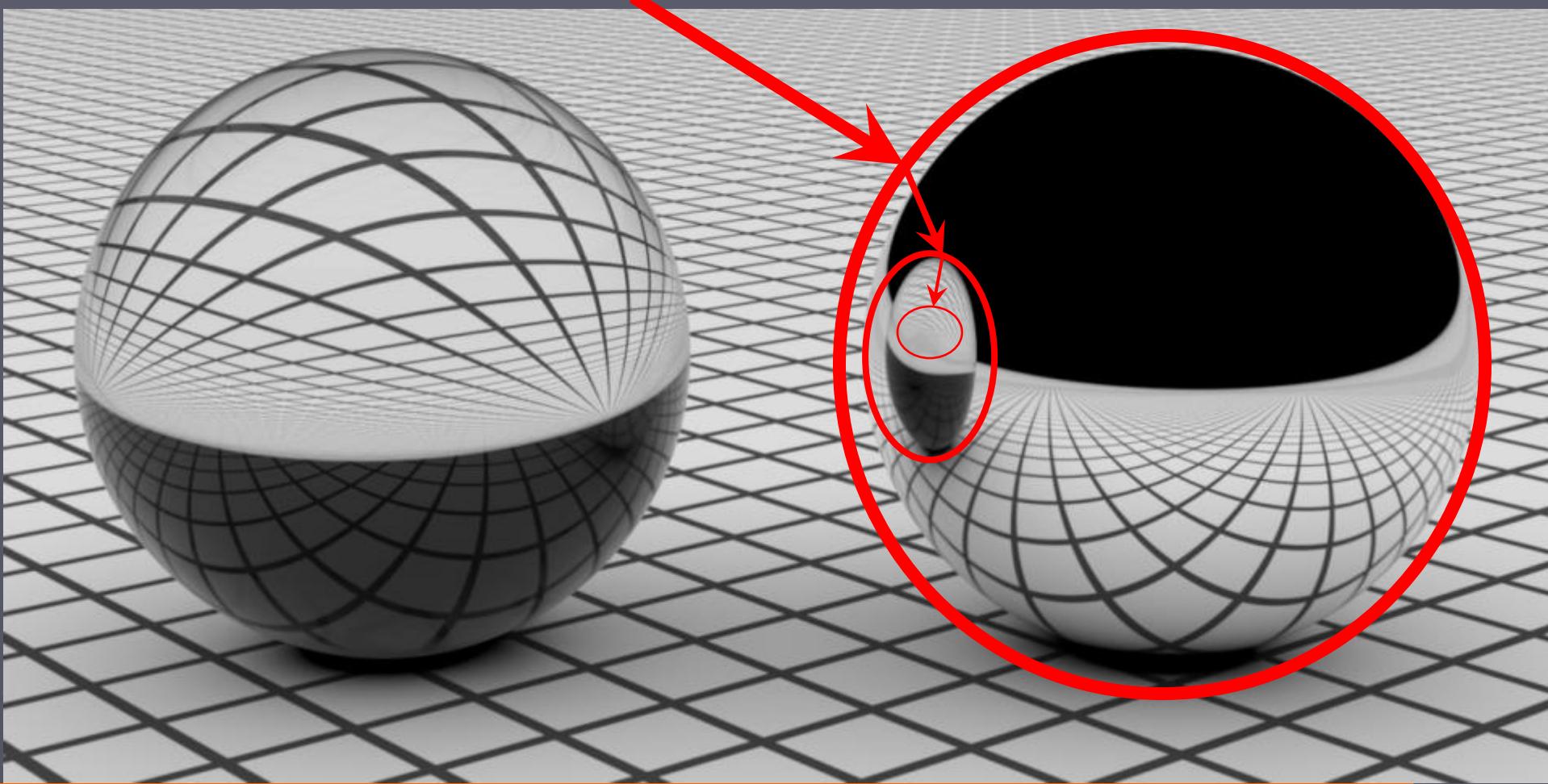
displace hit-point by tiny amount ε

(try $\varepsilon = 10^{-14}$):

- Shadow, reflection rays:
start at $(P_h - \varepsilon \text{ dir})$
- Transparency rays:
start at $(P_h + \varepsilon \text{ dir})$



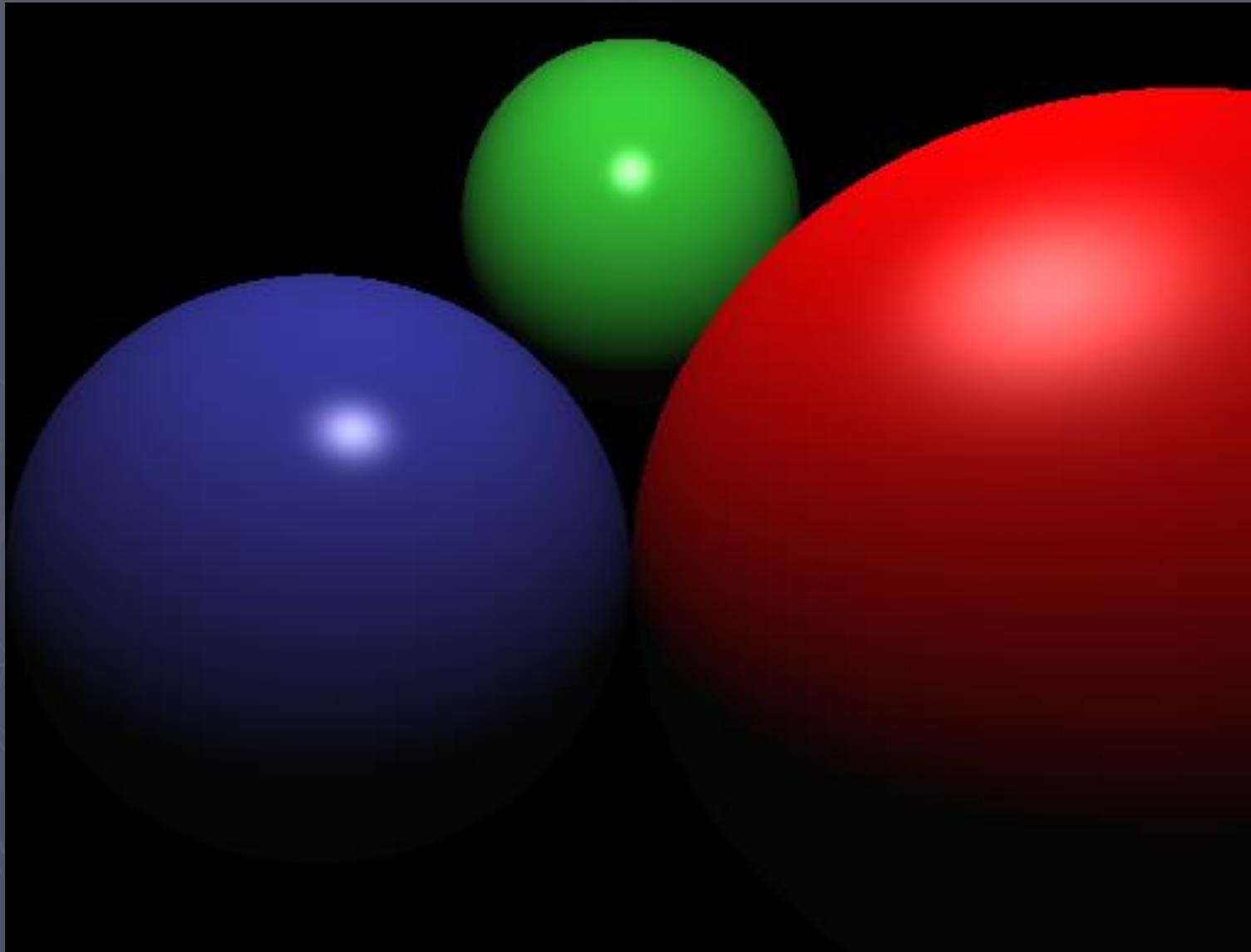
Next!: Recursive 'mirror' Reflection



Matt Pharr, Greg Humphries: Physically Based Rendering Techniques (PBRT)
highly recommended beautiful in-depth book <https://www.pbrt.org/gallery.html>

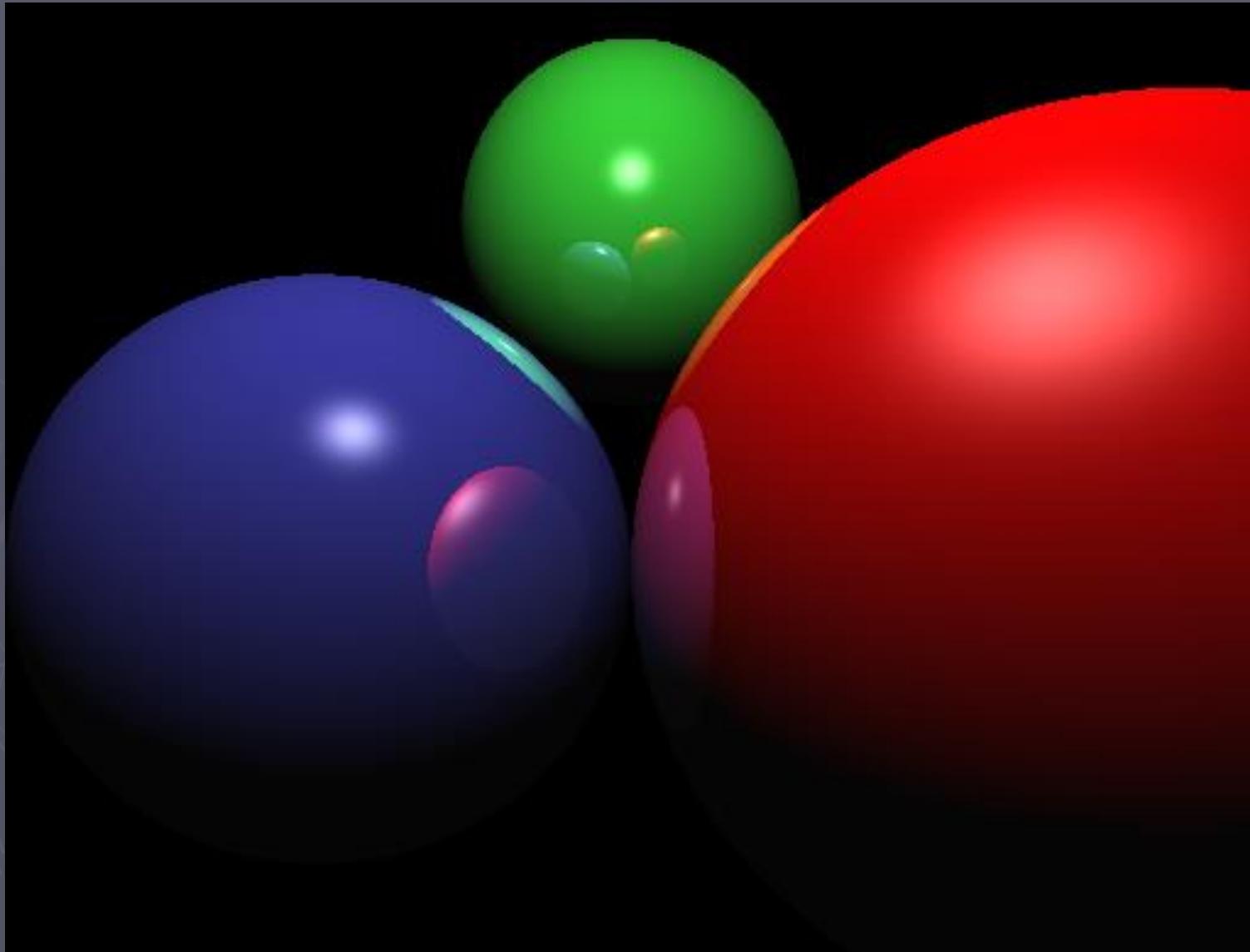
Ray Tracing: Reflection depth=0

(Images From Univ. Delaware)



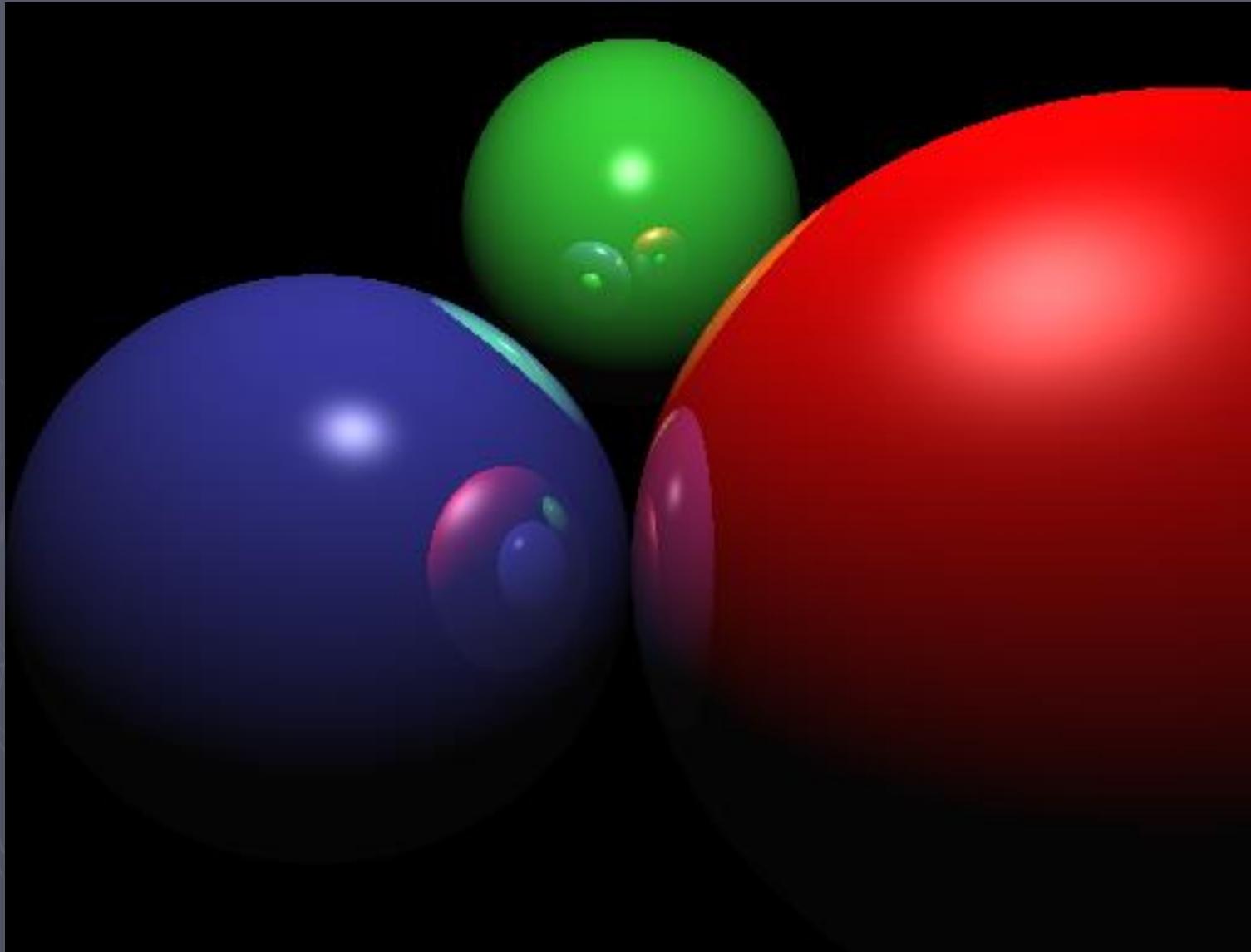
Ray Tracing: Reflection depth=1

(Images From Univ. Delaware)



Ray Tracing: Reflection depth=2

(Images From Univ. Delaware)



How to Organize Your Ray-Tracer for **RECURSION** without pain & agony

How should `CScene.makeRayTracedImage()` split up tracing & color-finding?

- ▶ a **Great Big Loop**; makes all pixels in the image buffer
(global var: `CImageBuf g_myPic`)
 - Calls `CCamera.makeEyeRay()` for each pixel; this method makes all that pixel's 'eye' ray(s) as needed (`CScene.rayNow`)

To trace the eye Ray and find its color?

SPLIT UP THE TASK!

- Calls `CScene.traceRay(rayNow)` to test ~~all CGeom objects~~ in our `CScene` for ray-intersection.
Returns `CHit` object 'eyeHit'
that holds *what* we hit & *where* we hit it.
- Calls `CScene.findShade(eyeHit)`;
to assess screen color at the end of the ray.
It completes the `CHit` object, gets the final color for the eye ray.
(and recursion may start here)

How to Organize Your Ray-Tracer for RECURSION without pain & agony

How should `CScene.makeRayTracedImage()` split up tracing & color-finding?

- ▶ a **Great Big Loop**; makes all pixels in the image buffer
(global var: `CImageBuf g_myPic`)
 - Calls `CCamera.makeEyeRay()` for each pixel to make all of that pixel's 'eye' rays (`CScene.rayNow`)

USE THIS FUNCTION PAIR RECURSIVELY:

- Calls `CScene.traceRay(rayNow)` to test *all* `CGeom` objects in our `CScene` for ray-intersection.
Returns `CHit` object 'eyeHit' (or better yet: a LIST of hit points) that holds **what** we hit & **where** we hit it.
- Calls `CScene.findShade(eyeHit)` to assess screen color at the end of the ray.
It completes the `CHit` object, gets the final color for the eye ray.
(and recursion may start here)

CScene.traceRay(rayNow)

(may grow and need further refactoring later)

- ▶ Trace this ray to find *all* its intersections with *all* CGeom objects in our **CScene**.
- ▶ Returns **CHit** object 'eyeHit' that holds *what* we hit & *where* we hit it
- ▶ LATER:
 - extend **traceRay()** to return **CHitList**: an array of **CHit** objects that describe ALL ray/object intersections (not just the nearest one!)
 - create a '**findHit()**' function that selects the **CHit** object for use by '**findShade()**' function; implements CSG.

CScene.findShade()

(may grow and need further refactoring later)

Find the screen-color at the end of this ray:

compute OpenGL-like shading (Phong light-sources and materials),
but improve it with shadows, reflections, and transparency.

- ▶ Examine the CGeom object we hit:
 - At the hit-point: find/retrieve the materials properties (K_ambient, K_diffuse, K_specular, K_emissive, K_shininess)
 - At the hit-point: find/retrieve the vectors we need for shading: surface normal, view vector, light-direction vectors, reflection vectors, etc.
- ▶ Compute Shadows at hit-point:
 - Create a shadow ray from hit-point to each light source
 - call **CScene.traceRay()** to find what each shadow ray hits, if anything.
Nothing? no shadow at hit-point; enable this light
- ▶ Find the color of reflected ray(s):
 - Create a reflection ray at the hit-point (use surface normal and view ray)
 - Call **CScene.traceRay()** to find **what** and **where** the reflected ray hits (if anything)
 - Call **CScene.findShade()** to **find the color** at the reflection-ray's endpoint.

Week 4 Goals: Just one last thing...

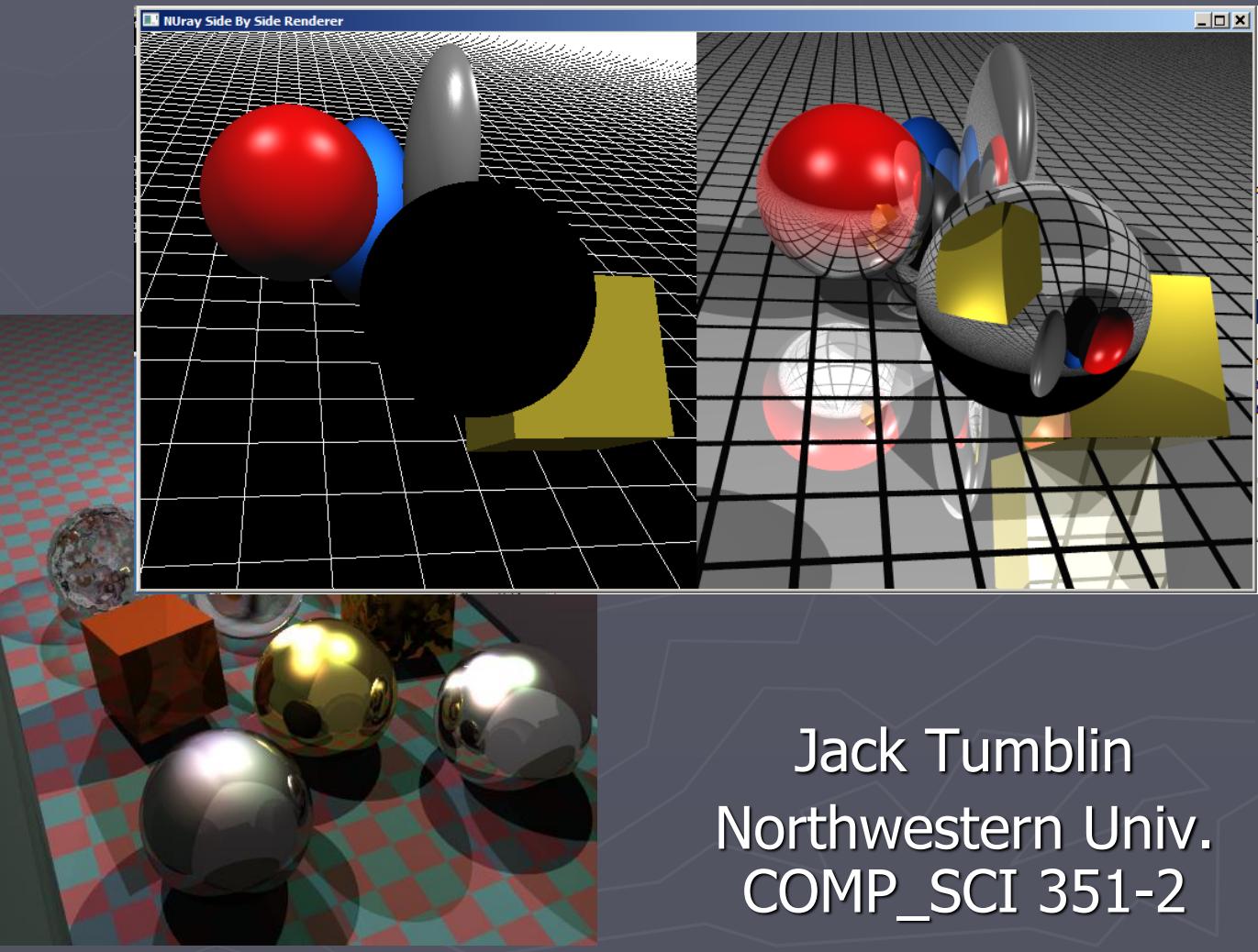
- ▶ 1) Make World-Space Surface Normals → OpenGL-like Phong Lighting
 - Transform normals? → READING: F.S.Hill, Section 14.7, pg 760,1,2;
 - CMatl objects: ambient, diffuse, specular, & shinyness members
 - Move lights with Mouse, Keyboard? More CTransRot objects!
- ▶ 2) Recursion: Show multiple reflections among objects;
 - 1 or more Grid-plane objects, 2 or more Spheres (try 3:  - ORGANIZE: 
 - Extra Credit: transparency/refraction too!
- ▶ 3) At least one 'Solid' (or 'Procedural') Texture
 - CHit: add member(s) for **model-space** hit-point, use it in
 - CMatl: model-space 3D color fcns: $R(x,y,z)$ $G(x,y,z)$ $B(x,y,z)$
 - Functions for Concentric Shells, Line→Grid→Cage, Checkerboards & other 'easy' textures? see FS Hill, Chap 14.8, pp. 770-773

END

END

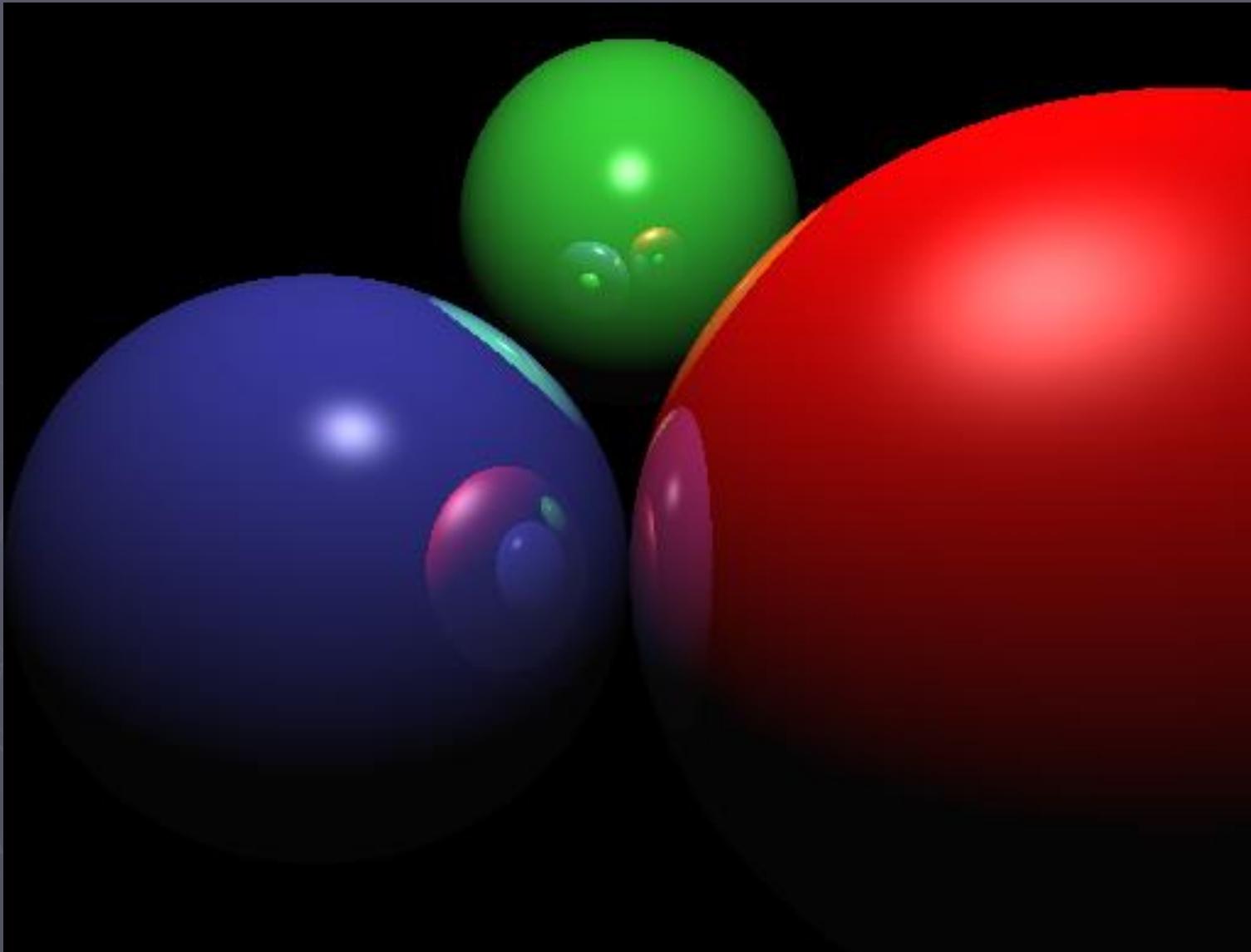


Ray Tracing E: Normals, Lighting, & Mirrors



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

Next!: Phong Lighting + Reflection



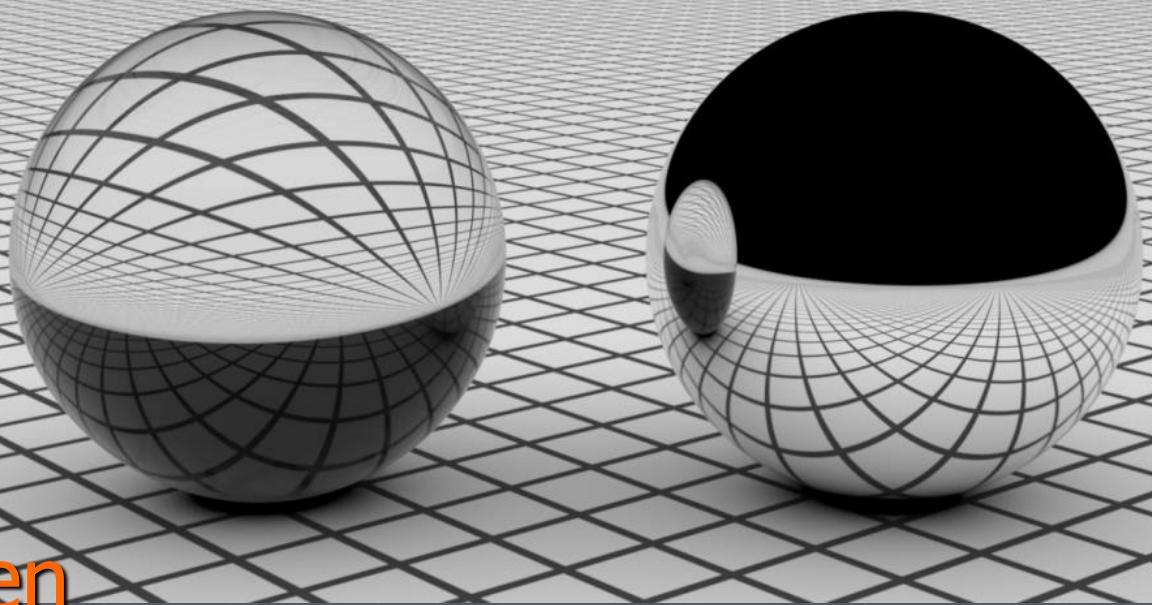
Our Plan:

Write code for:

- ▶ Camera, then
- ▶ Plane, then
- ▶ Antialiasing, then
- ▶ Transform, then
- ▶ Basic Disk and Sphere, then Shadow, then
- ▶ Phong Lighting,
- ▶ Reflection, then Transparency, then

Further Step-by-Step Refinements:

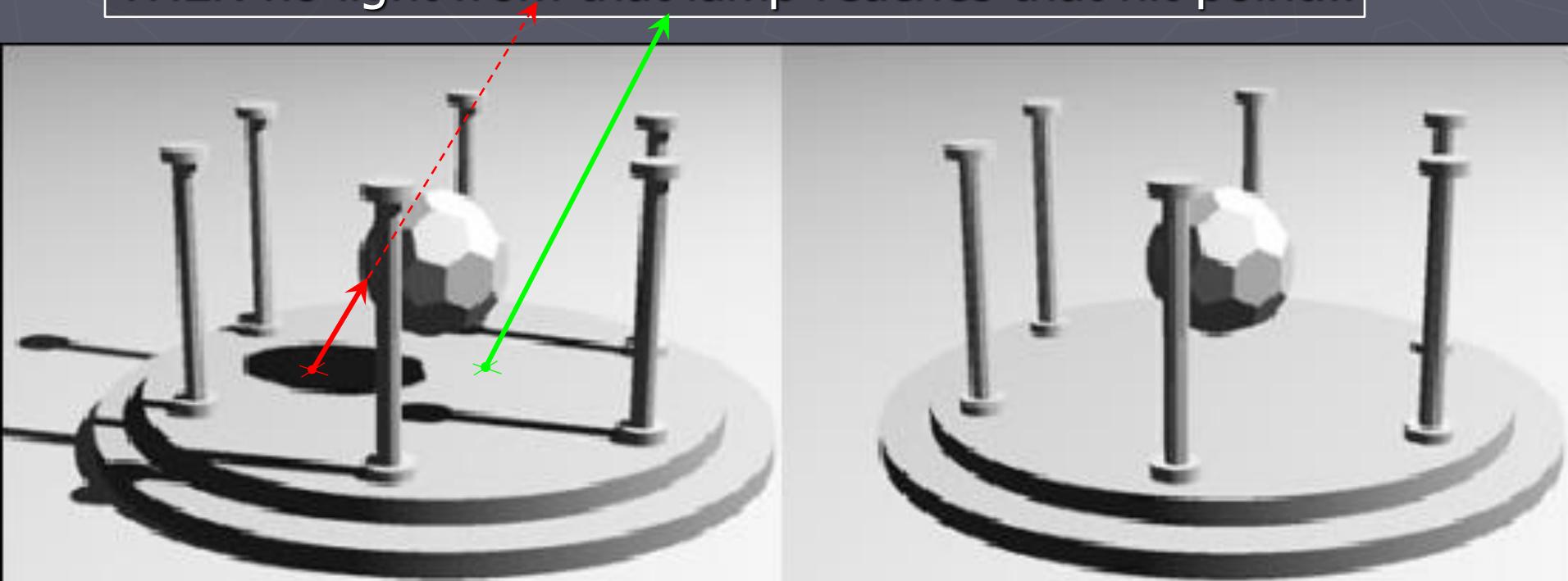
Lighting → Materials → Texture → CSG, etc. ...



*** **MAKE PICTURES AT EVERY STEP** ***

Shadows Matter, Yet Very Easy!

- ▶ Important height cues for disjoint objects
- ▶ Simple in ray-tracing, hard in WebGL (skip it for preview)
- ▶ How? Trace 'shadow' ray from hit-point to lamp position(s):
IF this 'shadow' ray is blocked by ANY CGeom object,
THEN no light from that lamp reaches that hit-point...



Confused? Lost in the starter-code?

- ▶ **STOP! Read!** In Reading Materials
(Canvas→Home page→ RayTracing module→ Weeks1-2 Reading, Weeks3-4 Reading),
find the PDFs for
Week1-4_FS_HillComputerGraphicsChap14...
- ▶ **READ CAREFULLY:** Their ray-tracer matches my notes and ray-tracer quite well, & explains all ray-tracer principles in detail.

Concise Ray-Tracer Outline

- FS Hill "Computer Graphics Using OpenGL"
2nd Edition, Chap 14, pg 736, Fig 14.4:

define the objects and light sources in the scene

set up the camera

```
for(int r = 0; r < nRows; r++)
    for(int c = 0; c < nCols; c++)
    {
```

- 1. Build the rc-th ray*
- 2. Find all intersections of the rc-th ray with objects in the scene*
- 3. Identify the intersection that lies closest to, and in front of, the eye*
- 4. Compute the "hit point" where the ray hits this object, and the normal vector at that point*
- 5. Find the color of the light returning to the eye along the ray from the point of intersection*
- 6. Place the color in the rc-th pixel.*

```
}
```

Nearly Identical Organization

Our Starter Code:

- ▶ Classes: CScene, CRay, CCamera
- ▶ Global Vars: myScene (contains rayCam), myPic (image buffer)
- ▶ CScene.
 makeRayTracedImage()
- ▶ CScene data member:
 CRay eyeRay;
- ▶ CScene.findShade()
 - Calls CScene.traceRay()
- ▶ Arrays of CHit objects
(I skip the containing class)

FS HILL book:

- ↔ Classes: Scene, Ray, Camera
- ↔ Global Vars: theScene, theCamera
?! (has no image buffer!)
- ↔ Camera::Raytrace()
- ▶ Camera data member:
 Ray theRay;
- ↔ Scene::shade()
 - Calls Scene::FindFirstHit()
- ↔ Intersection class:
 holds array of 'HitInfo' objects

Review: how to color 'Eye' Rays

1. Our CScene.**makeRayTracedImage()** function steps through each pixel in the CImgBuf image.
2. For each pixel, **CCamera** object makes an **eye ray** that finds the pixel's color.
3. Trace one eye ray:
find **what** the ray hits, and **where**;

```
myScene.traceRay(eyeRay, hitList);  
// myScene: our CScene object that holds everything  
// hitList: holds array of CHit objects
```

traceRay() fcn. builds an **array** of CHit object(s)

- **FS Hill:** "save ALL the ray intersections in a **list** of 'HitInfo' objects held in the Intersection class (see pg 745), then find, copy, and use the nearest HitInfo object."

4. Write **JUST ONE FUNCTION** to find ray color ...

Shading: Do it (almost) all in World-Space

5. How? Use the **CScene** fcn: **findShade(HitList)** to:

- a) select & complete the most-suitable **CHit** object.

When completed, that **CHit** object will hold all this:

(see **JT_Tracer0-Scene.js** and FS Hill, pg 746, **HitInfo** class)

- **t0**; (floating point 'hit time' for traced ray)
- **hitGeom**; (tells us which **CGeom** object was hit)
(HINT: can use an integer array index for **CScene.GeomList** array)
- **isEntering**; (entering or leaving object? Boolean)
- (hit point in **WORLD** space) = **orig + t0*dir**
- (hit point in **MODEL** space) = **torig + t0*tdir**
- (surface normal in **WORLD** space) found like this:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2Model}]^T \mathbf{N}_{\text{model}}$$

- b) **THEN:** use Phong lighting(and more) to find the ray's color!

Shading: Do it (almost) all in World-Space

5. How? Call the CScene fcn: `findShade(HitList)` to:

a) find & complete the nearest CHit object.

When finished, that CHit object will holds
(see FS Hill, pg 746, HitInfo class)

- (floating point 'hit time' for traced ray) `t0`;
- (the CGeom object we hit)
(you can use an integer array index for CScene.GeomList array)
- (entering or leaving object? boolean) `isEntering`;
- (hit point in WORLD space) = `orig + t0*dir`
- (hit point in MODEL space) = `torig + t0*tdir`
- (surface normal in *WORLD* space) found like this:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2Model}]^T \mathbf{N}_{\text{model}}$$

b) **THEN:** use Phong lighting(and more) to find the ray's color!

Shading: Do it (almost) all in World-Space

5. Call ~~[r,g,b] = findshading()~~

!? uhh-- WHUT ?!

How did we figure out
the WORLD-space surface normal N_{world}

from a MODEL-space normal N_{model} !?

& Why is * *THIS* *
the solution?

(point in MODEL space) $t_{\text{orig}} + t_0 \cdot t_{\text{dir}}$

(surface normal in *WORLD* space)

$$N_{\text{world}} = [\text{worldRay2Model}]^T N_{\text{model}}$$

AND (the Ray's final color!)

Surface **Normals** in World Space

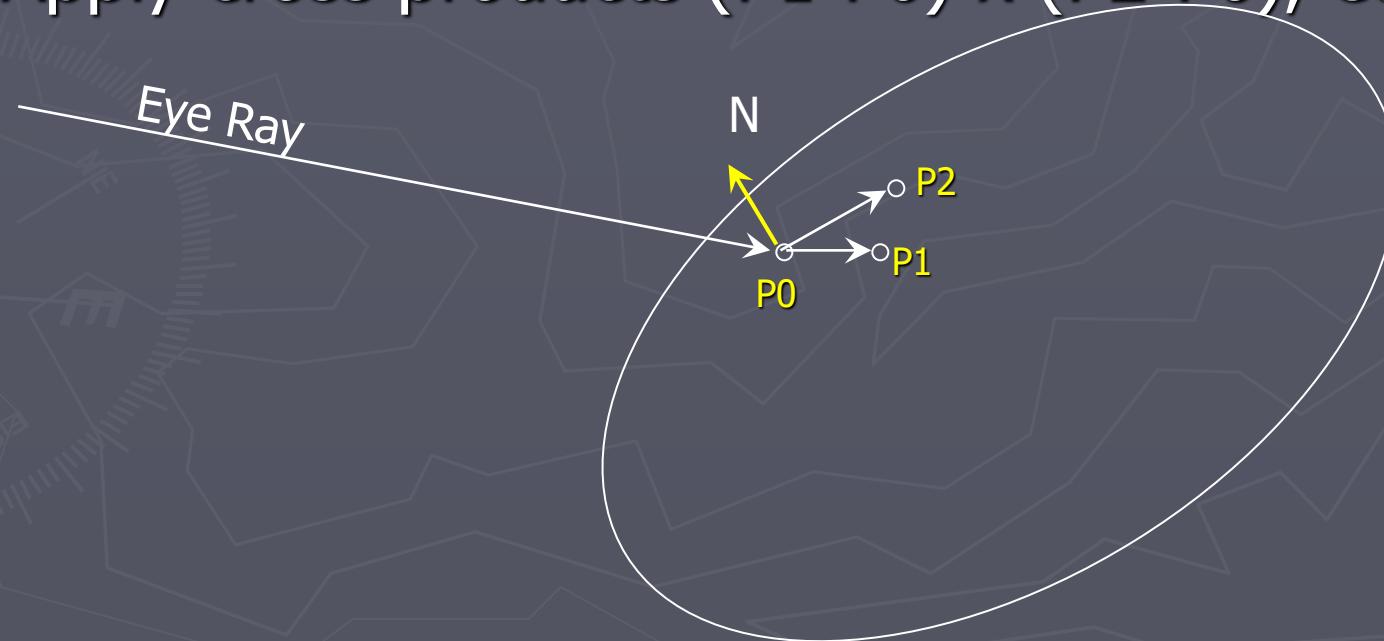
- ▶ **RECALL:** We defined all CGeom objects each in their own local 'model' coord systems
 - CGeom shapes are *ALWAYS* centered at origin &
 - *ALWAYS* fixed, unit-sized (e.g. sphere radius=1)
- ▶ ?How do **world**-coord-system rays trace in **model**?
How do we set object's positions? Sizes? Angles?
 - Easy: put it all in the [**worldRay2Model**] matrix;
(transforms rays from world- to model-space)
- ▶ **OK, but causes a new PROBLEM!**
 - we can **FIND** CGeom surface normals **N** easily in 'model' coord system, but
 - But we **NEED** them in the 'world' coord system!

Normals? NAÏVE (bad!) SOLUTION:

► Find it by CONSTRUCTION:

Find and transform some model-space points **backwards** to world coords to find normal:

- Find 3 nearby non-collinear points on surface,
- Apply cross products $(P_1 - P_0) \times (P_2 - P_0)$, etc...



Normals? NAÏVE (bad!) SOLUTION:

► Find it by CONSTRUCTION:

Find and transform some model-space points **backwards** to world coords to find normal:

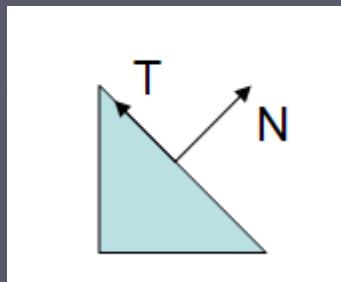
- Find 3 nearby non-collinear points on surface,
- Apply cross products $(P1-P0) \times (P2-P0)$, etc...

!!!UGLY!!!

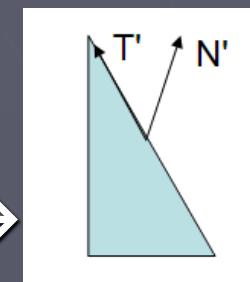
- REQUIRES 2 extra model-space points,
- REQUIRES CGeom to hold the *inverse* of the `worldRay2model` matrix:
?!? `Mat4 modelRay2world` also?!?! NOO!!!!

A Better Way to find Normals (1)

- RECALL: scale transform operations can *distort* the surface-normal vectors:



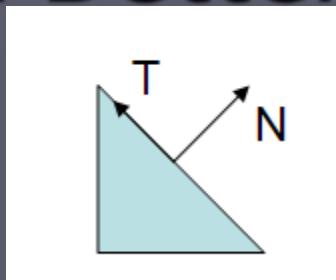
→ scale x coords by 0.5 →



(from: <http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/>)
(detailed analysis? See http://www.songho.ca/opengl/gl_normaltransform.html)

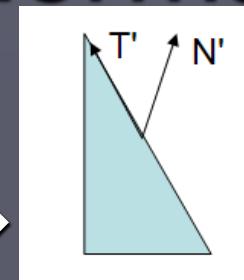
- THUS as WebGL transforms **vertices** by matrix **M**
we *also* transform **normals** by matrix **M^{-T}**
- This **inverse-transpose matrix M^{-T}** keeps normals perpendicular to surface for any directional scaling.
- How do we apply this to ray-tracing? **BACKWARDS!**

A Better Way to find Normals (2)



surface-normal vectors:

→ scale x coords by 0.5 →



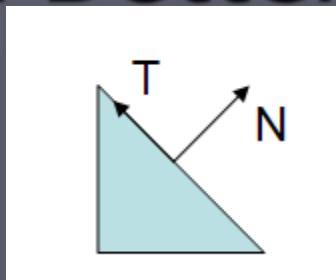
RECALL: in WebGL we transform **vertices** by matrix **M**
and we transform **normals** by matrix **M^{-T}**

- ▶ Ray tracing transforms **rays** from world \rightarrow model coords by the *inverse* of WebGL's vertex-transform matrix: uses **M^{-1}**
(In my ray tracer, the **M^{-1}** matrix is [worldRay2model])
- ▶ To transform **rays** to world \leftarrow model coords, just use **M**,
and in my ray tracer: **$(M) = ([worldRay2model]^{-1})$**
- ▶ To transform **normals** to world \leftarrow model coords, Use **M^{-T}** .

How? In my ray tracer:

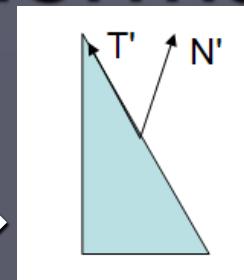
$$M^{-T} = ((M)^{-1})^T = ([worldRay2model]^{-1})^{-1})^T$$

A Better Way to find Normals (2)



surface-normal vectors:

→ scale x coords by 0.5 →



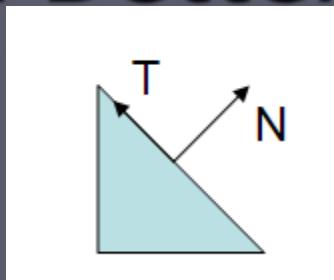
RECALL: in WebGL we transform **vertices** by matrix **M**
and we transform **normals** by matrix **M^{-1}**

- ▶ Ray tracing transforms **rays** from world \rightarrow model coords by the *inverse* of WebGL's vertex-transform matrix: uses **M^{-1}**
(In my ray tracer, the **M^{-1}** matrix is [worldRay2model])
- ▶ To transform **rays** to world \leftarrow model coords, just use **M**,
and in my ray tracer: **$(M) = ([worldRay2model]^{-1})$**
- ▶ To transform **normals** to world \leftarrow model coords, Use **M^{-T}** .

How? In my ray tracer:

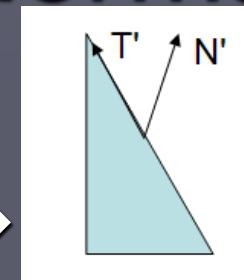
$$M^{-T} = ((M)^{-1})^T = ([worldRay2model]^{-1})^{-1})^T$$

A Better Way to find Normals (3)



surface-normal vectors:

→ scale x coords by 0.5 →



RECALL: in WebGL we transform **vertices** by matrix **M**
and we transform **normals** by matrix **M^{-T}**

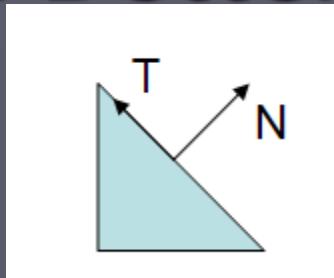
- ▶ Ray tracing transforms rays from world \rightarrow model coords by the *inverse* of WebGL's vertex-transform matrix: uses M^{-1} (In my ray tracer, the M^{-1} matrix is [worldRay2model])
- ▶ To transform rays to world \leftarrow model coords, just use **M**, and in my ray tracer: $(M) = ([worldRay2model]^{-1})$
- ▶ To transform **normals** to world \leftarrow model coords, Use **M^{-T}** .

How? In my ray tracer: $M^{-T} = ([worldRay2model]^{-1})^{-T}$

$$M^{-T} = ((M)^{-1})^T = ([worldRay2model]^{-1})^{-1}^T$$

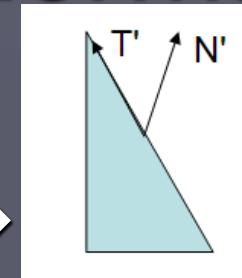
Look!

A Better Way to find Normals (4)



surface-normal vectors:

→ scale x coords by 0.5 →



RECALL: in WebGL we transform **vertices** by matrix **M**
and we transform **normals** by matrix **M^{-T}**

- ▶ Ray tracing transforms rays from world \rightarrow model coords by the *inverse* of WebGL's vertex-transform matrix: use M^{-1} (In my ray tracer, the M^{-1} matrix is [worldRay2model])
- ▶ To transform rays to world \leftarrow model coords, just use M , and in my ray tracer: $(M) = ([worldRay2model]^{-1})$
- ▶ To transform **normals** to world \leftarrow model coords, Use **M^{-T}** .

How? In my ray tracer: $M^{-T} = ([worldRay2model]^{-1})^{-T}$

$$M^{-T} = ((M)^{-1})^T = ([worldRay2model]^{-1})^{-1})^T$$

where **$M^{-T} = ([worldRay2model])^T$**

A Better Way to find Normals (5)

HOW TO DO THAT in your code:

To transform ray **model**-space normal vector $\mathbf{N}_{\text{model}}$
to ray **world**-space normal vector $\mathbf{N}_{\text{world}}$

Just apply *TRANSPOSE* of the **worldRay2model** matrix:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2model}]^T \mathbf{N}_{\text{model}};$$

(COOL! fast and easy!
now we can do ALL our lighting calcs in world-space!)

A Better Way to find Normals (6)

HOW TO DO THAT in your code:

To transform ray **model**-space normal vector $\mathbf{N}_{\text{model}}$
to ray **world**-space normal vector $\mathbf{N}_{\text{world}}$

Just apply *TRANSPOSE* of the **worldRay2model** matrix:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2model}]^T \mathbf{N}_{\text{model}};$$

?Using **vec4** and **mat4**? (4x4 Matrix)?

!CAUTION! -- ! UGLY SURPRISE !

- ▶ **worldRay2model**: the 4th column has no effect on normal vector $\mathbf{N}_{\text{model}}$
- ▶ But in $[\text{worldRay2model}]^T$ the 4th row would **disastrously corrupt** $\mathbf{N}_{\text{model}}$!

$$\begin{bmatrix} a & b & c & tx \\ d & e & f & ty \\ g & h & i & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ tx & ty & tz & 1 \end{bmatrix}$$

A Better Way to find Normals (7)

HOW TO DO THAT in your code:

To transform ray **model**-space normal vector $\mathbf{N}_{\text{model}}$
to ray **world**-space normal vector $\mathbf{N}_{\text{world}}$

Just apply *TRANSPOSE* of the worldRay2model matrix:

$$\mathbf{N}_{\text{world}} = [\text{worldRay2model}]^T \mathbf{N}_{\text{model}};$$

?Using **vec4** and **mat4**? (4x4 Matrix)?

!CAUTION! -- ! UGLY SURPRISE !

- ▶ worldRay2model: the 4th column has no effect on normal vector $\mathbf{N}_{\text{model}}$
- ▶ But in $[\text{worldRay2model}]^T$ the 4th row would disastrously corrupt $\mathbf{N}_{\text{model}}!$

$$\begin{bmatrix} a & b & c & tx \\ d & e & f & ty \\ g & h & i & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ tx & ty & tz & 1 \end{bmatrix}$$

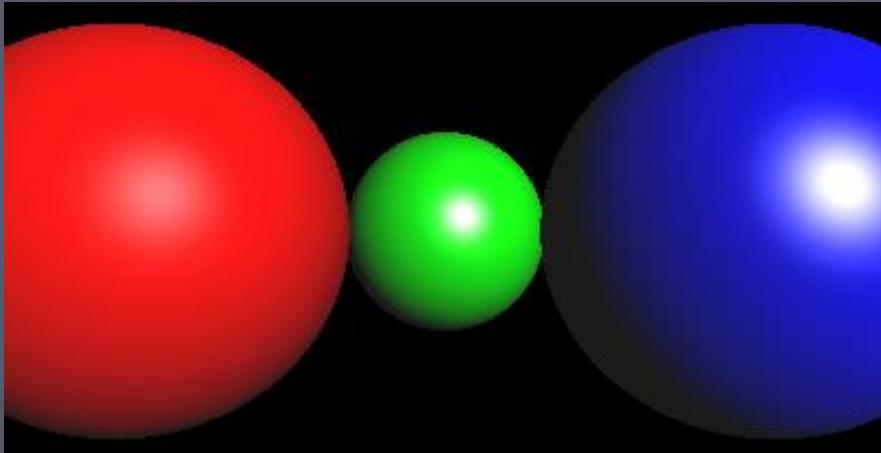
ANSWER: mat3, or set bottom row to [0 0 0 1]!

Week 4 Goals

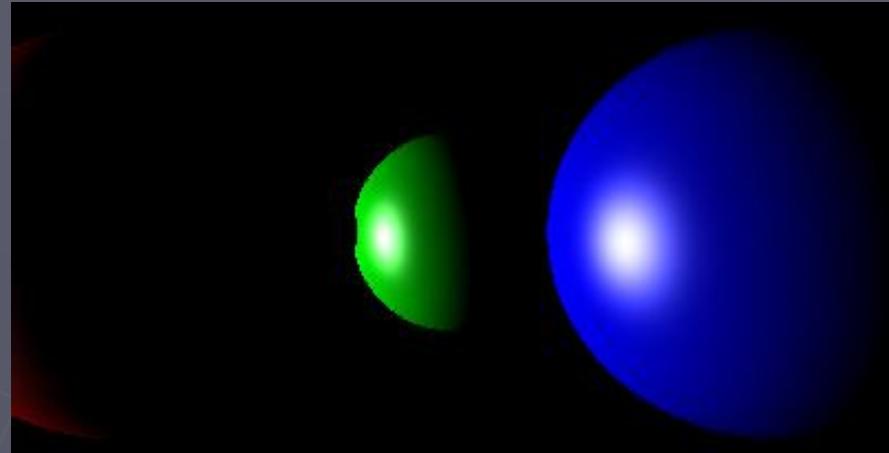
- ▶ 1) Make World-Space Surface Normals → OpenGL-like Phong Lighting
 - Transform normals? → See CS 351-1 Reading...
 - CMatl objects: ambient, diffuse, specular, & shinyness members
 - Move lights with Mouse, Keyboard? More GUIbox member fcns!
- ▶ 2) Recursion: Show multiple reflections among objects;
 - 1 or more Grid-plane objects, 2 or more Spheres (try 3: )
 - ORGANIZE: 
 - Extra Credit: transparency/refraction too!
- ▶ 3) At least one 'Solid' (or 'Procedural') Texture
 - CHit: add member(s) for model-space hit-point, use it in
 - CMatl: model-space 3D color fcns: $R(x,y,z)$ $G(x,y,z)$ $B(x,y,z)$
 - Functions for Concentric Shells, Line→Grid→Cage, Checkerboards & other 'easy' textures? see FS Hill, Chap 14.8, pp. 770-773

Ray Casting: *without* shadows

Right-side light:



Left-side light:

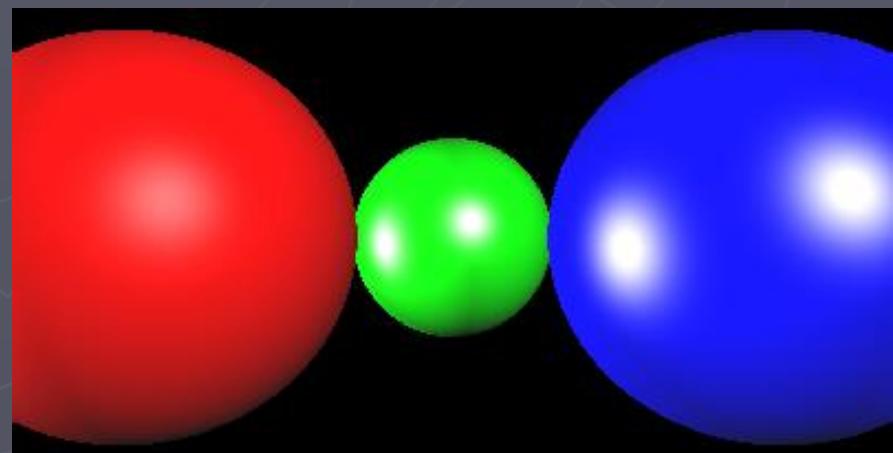


Combined lights:



ADD ***all*** their diffuse
& specular terms

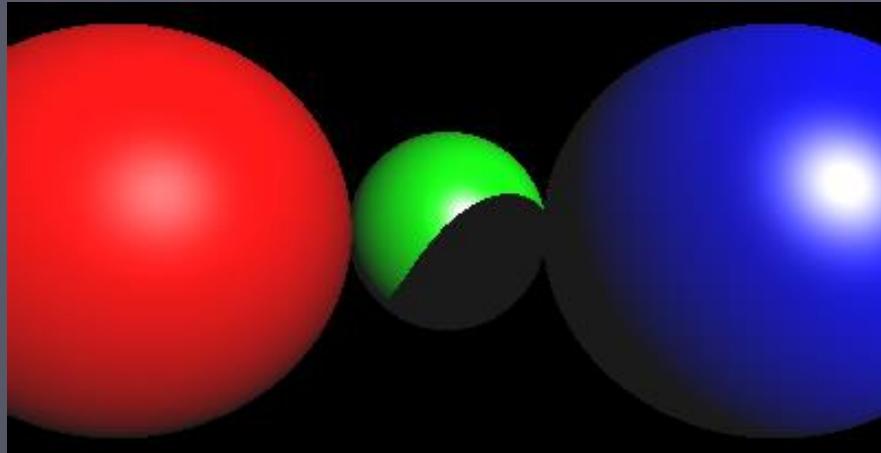
BUT use ***only one*** of the
ambient terms. (do you know why?)



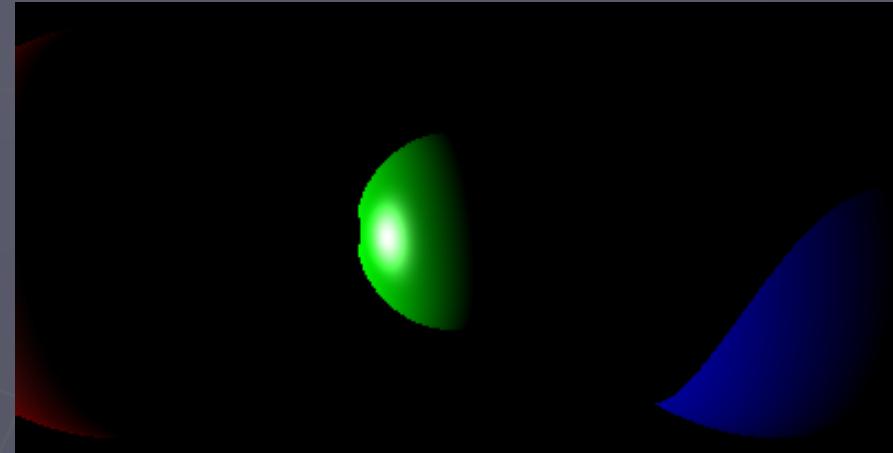
(Images From Univ. Delaware)

Ray Casting: *with* shadows

Right-side light



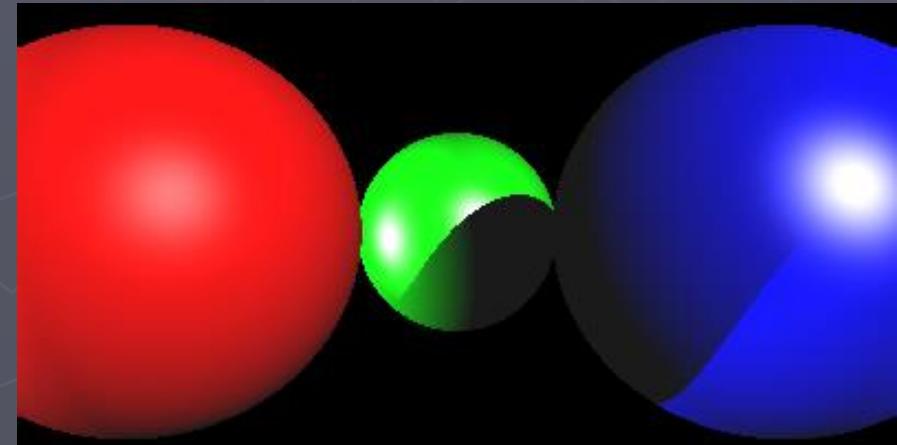
Left-side light



Look very closely at
how these shadows combine.

SURPRISE!

**shadows *DON'T BLOCK*
ambient light!** (do you know why?)



(Images From Univ. Delaware)

Ray Color? Start with Phong Lighting

Who finds ray color? which function?

I Recommend: `cScene` member function

`CScene.findShade(myHitList)`

IMPORTANT: Compute and Use
ALL vectors for Phong Lighting
in the **world** coord system:

V: View Vector (to eye; negative of eye ray: `-dir`)

L: 'Lamp' or Shadow vector (one to each light)

N: Surface Normal vector (in World-Space)

Later: **Mir:** mirror-reflected ray

Tr : transmitted ray, for transparency

Phong Lighting Step 1: Find Scene Vectors L,N,V

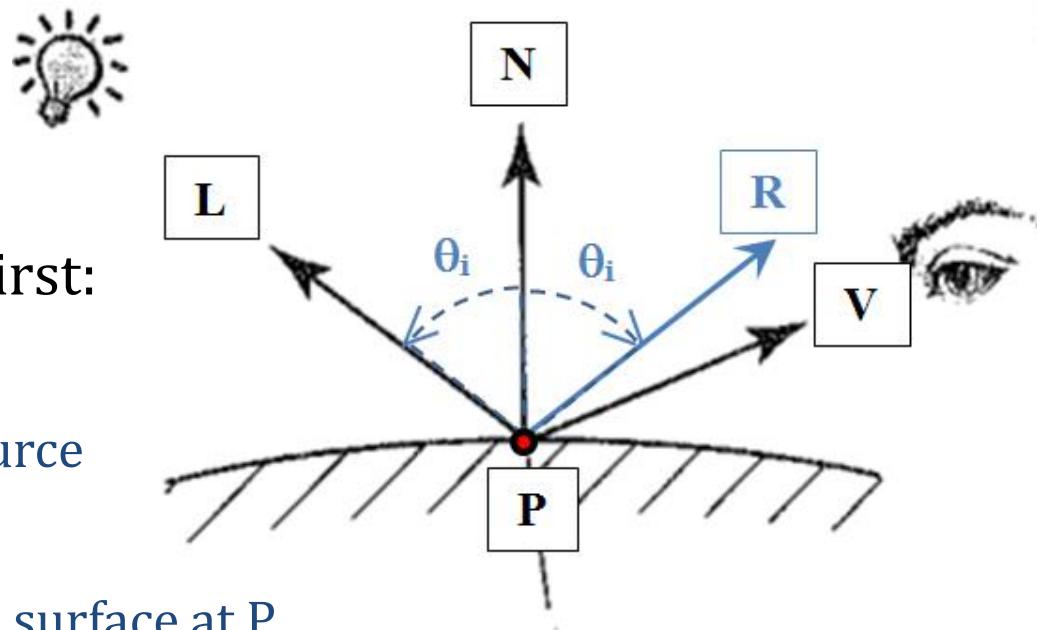
To find **Ray Color**
at hit-point **P (start)**:

1) Find all 3D scene vectors first:

a) Light Vector **L**:
unit vector towards light source

b) Normal Vector **N**:
unit vector perpendicular to surface at P

c) View Vector **V**:
unit vector towards camera eye-point (easy! it's **-eyeRay.dir**)

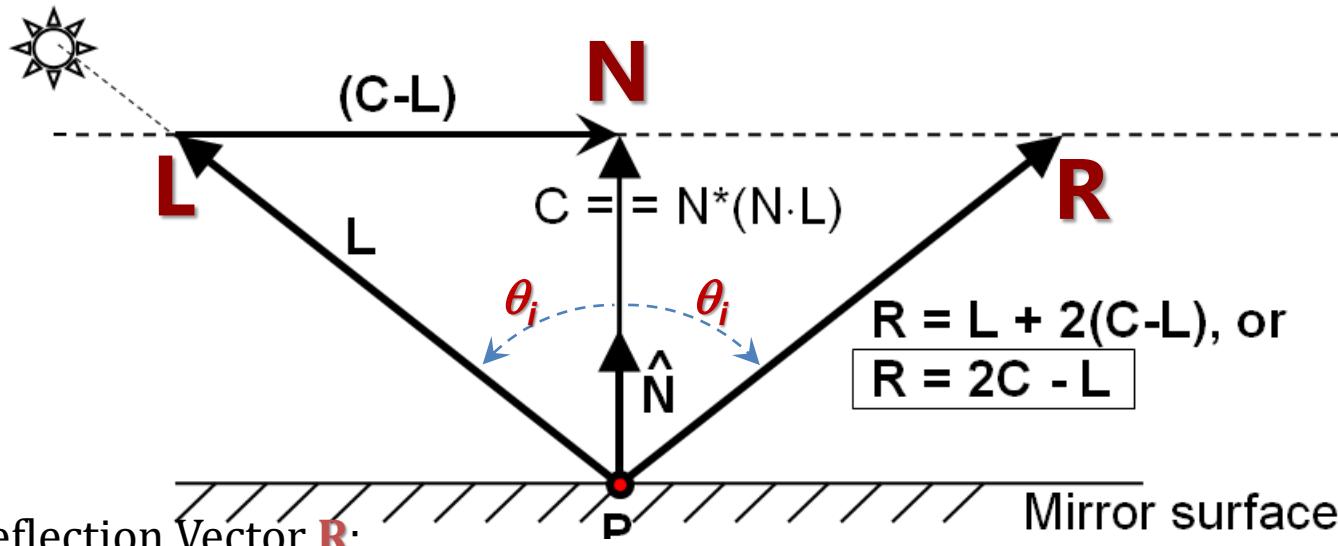


REMEMBER –all are vectors world-space coords!

Onwards to step 2: how do we find the Reflected-light Vector **R**?

Phong Lighting Step 2: Find reflection Vector R

To find **Ray Color**
at hit-point **P** (*cont'd*):



2) COMPUTE the Light Reflection Vector **R**:

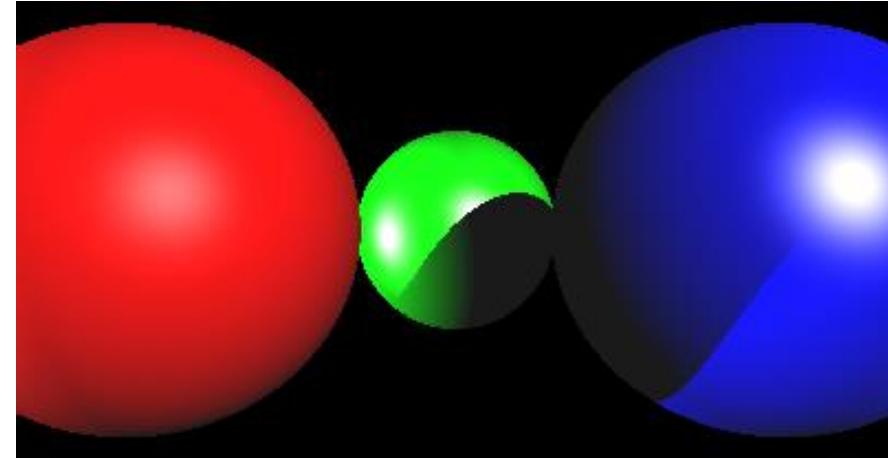
- Given unit light vector \mathbf{L} , find lengthened surface normal \mathbf{C}
$$\mathbf{C} = \mathbf{N} (\mathbf{L} \cdot \mathbf{N})$$
 (NOTE! does not require normalized \mathbf{N} !)
- In diagram, if we add vector $2^*(\mathbf{C}-\mathbf{L})$ to \mathbf{L} vector we get \mathbf{R} vector. Simplify:
$$\mathbf{R} = 2\mathbf{C} - \mathbf{L}$$
- Result:** reflected vector **R** GLSL-ES → See built-in '`reflect()`' function
(If \mathbf{N} is a unit-length vector, then \mathbf{R} vector length matches \mathbf{L} vector length)

Phong Lighting Step 3: **Gather Light** **& Material Data**

To find **Ray Color**
at hit-point **P** (*cont'd*):

3) For each light source, gather:

- ▶ RGB triplet for **Ambient** Illumination
- ▶ RGB triplet for **Diffuse** Illumination
- ▶ RGB triplet for **Specular** Illumination



For each surface material, gather:

- ▶ RGB triplet for **Ambient** Reflectance
- ▶ RGB triplet for **Diffuse** Reflectance
- ▶ RGB triplet for **Specular** Reflectance
- ▶ RGB triplet for **Emissive** term(often zero)
- ▶ Scalar 'shininess' or 'specular exponent' term

$$\mathbf{I_a} \quad 0 \leq I_{ar}, I_{ag}, I_{ab} \leq 1$$

$$\mathbf{I_d} \quad 0 \leq I_{dr}, I_{dg}, I_{db} \leq 1$$

$$\mathbf{I_s} \quad 0 \leq I_{sr}, I_{sg}, I_{sb} \leq 1$$

$$\mathbf{K_a} \quad 0 \leq K_{ar}, K_{ag}, K_{ab} \leq 1$$

$$\mathbf{K_d} \quad 0 \leq K_{dr}, K_{dg}, K_{db} \leq 1$$

$$\mathbf{K_s} \quad 0 \leq K_{sr}, K_{sg}, K_{sb} \leq 1$$

$$\mathbf{K_e} \quad 0 \leq K_{er}, K_{eg}, K_{eb} \leq 1$$

$$\mathbf{Se} \quad 1 \leq S_e \leq \sim 100$$

Phong Lighting Step 4: Sum of All Light Amounts

To find **Ray Color**
at hit-point **P** (*cont'd*):

sum of each kind of light at **P**:

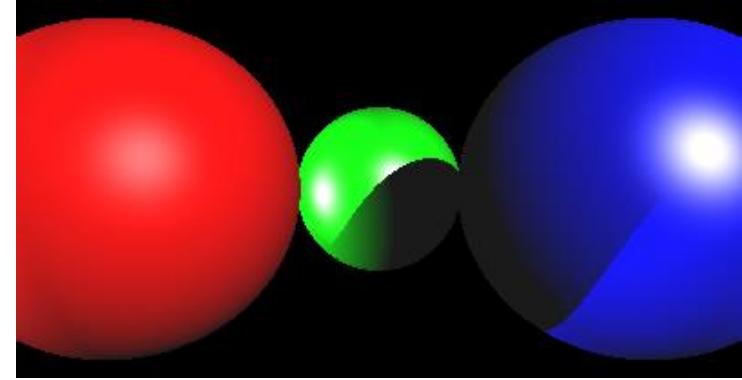
Phong Lighting = Emissive + Ambient + (Diffuse + Specular)*notShadow
SUMMED for all light sources,
BUT in shadows, both (Diffuse + Specular) drop to zero.

4) For the i-th light source, find:

(ignore shadows in your WebGL preview)

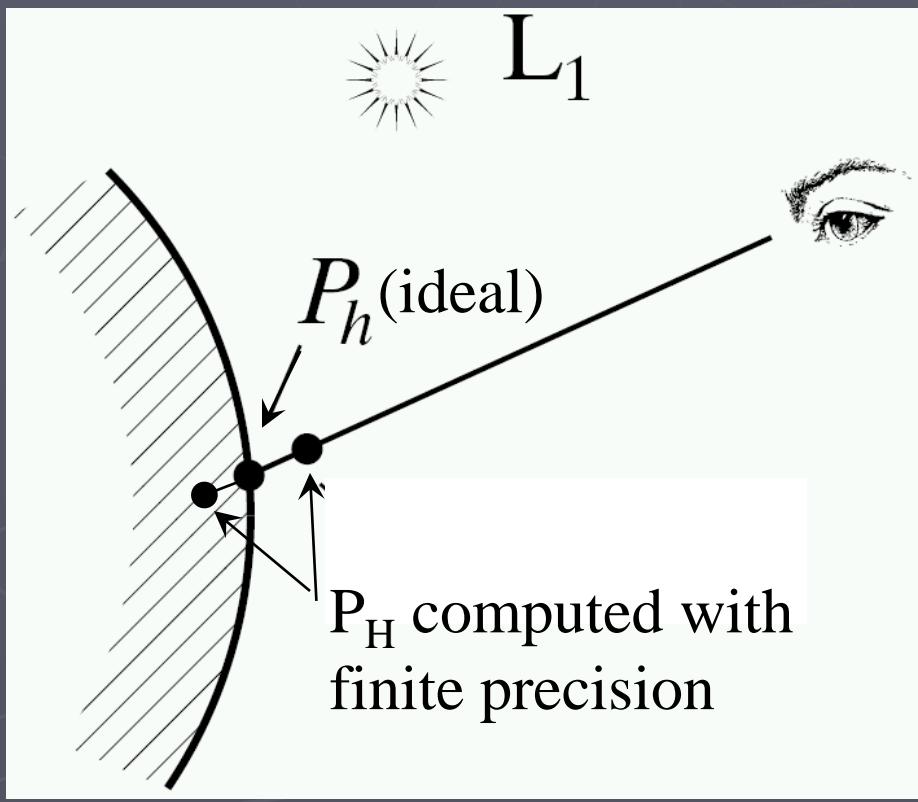
```
RGB= Ke + // 'emissive' material; it glows!  
      Ia*Ka; // ambient light * ambient reflectance  
if(!shadow) {  
    RGB += Id*Kd*Att*max(0,(N·L)) + // diffuse light * diffuse reflectance  
          Is*Ks*Att*(max(0,R·V))Se, // specular light * specular reflectance
```

- ▶ Distance Attenuation scalar: $0 \leq \text{Att} \leq 1$
 - ▶ Fast, OK-looking default value: $\text{Att} = 1.0$
 - ▶ Physically correct value: $\text{Att}(\mathbf{d}) = 1/(\text{distance to light})^2$ (too dark too fast!)
 - ▶ Faster, Nice-looking 'Hack': $\text{Att}(\mathbf{d}) = 1/\text{distance to light}$
 - ▶ OpenGL compromise: $\text{Att}(\mathbf{d}) = \min(1, 1/(c_1 + c_2 \cdot d + c_3 \cdot d^2))$
- ▶ 'Shinyness' or 'specular exponent' $1 \leq \text{Se} \leq \sim 100$ (large for sharp, small highlights)



UGLY BUG! Ray-Intersection Precision

- ▶ Finite precision: even **double** math results have tiny quantization errors ($\sim \text{result} * 10^{-17}$)
- ▶ At ray length '**t**', tiny errors misplace the computed hit-point **P_h**
- ▶ on, above, or below the surface. Uh oh.



UGLY BUG! Ray-Intersection Precision

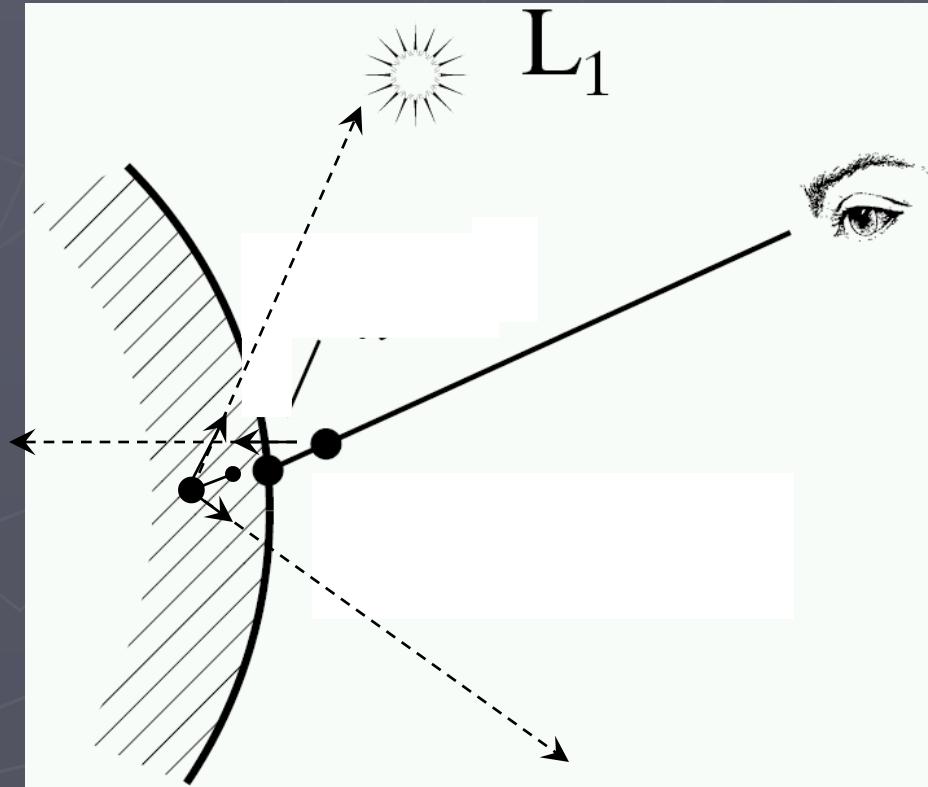
Random Speckled 'Acne' -- PROBLEMS!

Is Hit-point P_H *ON* or *BELOW* surface?

- All Shadow rays **fail!**
- Reflection ray **fails!**
- (surface blocks them)

► Hit-point ***ABOVE***?

- Transparency rays **fail!**
- (they need to begin *INSIDE* the surface)



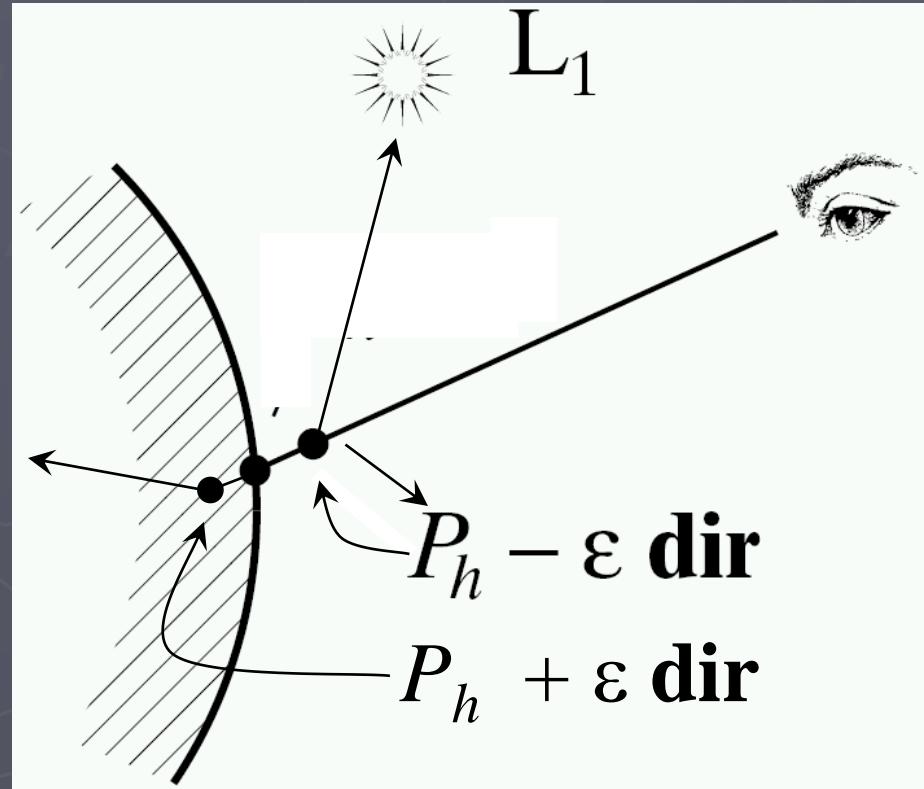
UGLY BUG! Ray-Intersection Precision

► SOLUTION:

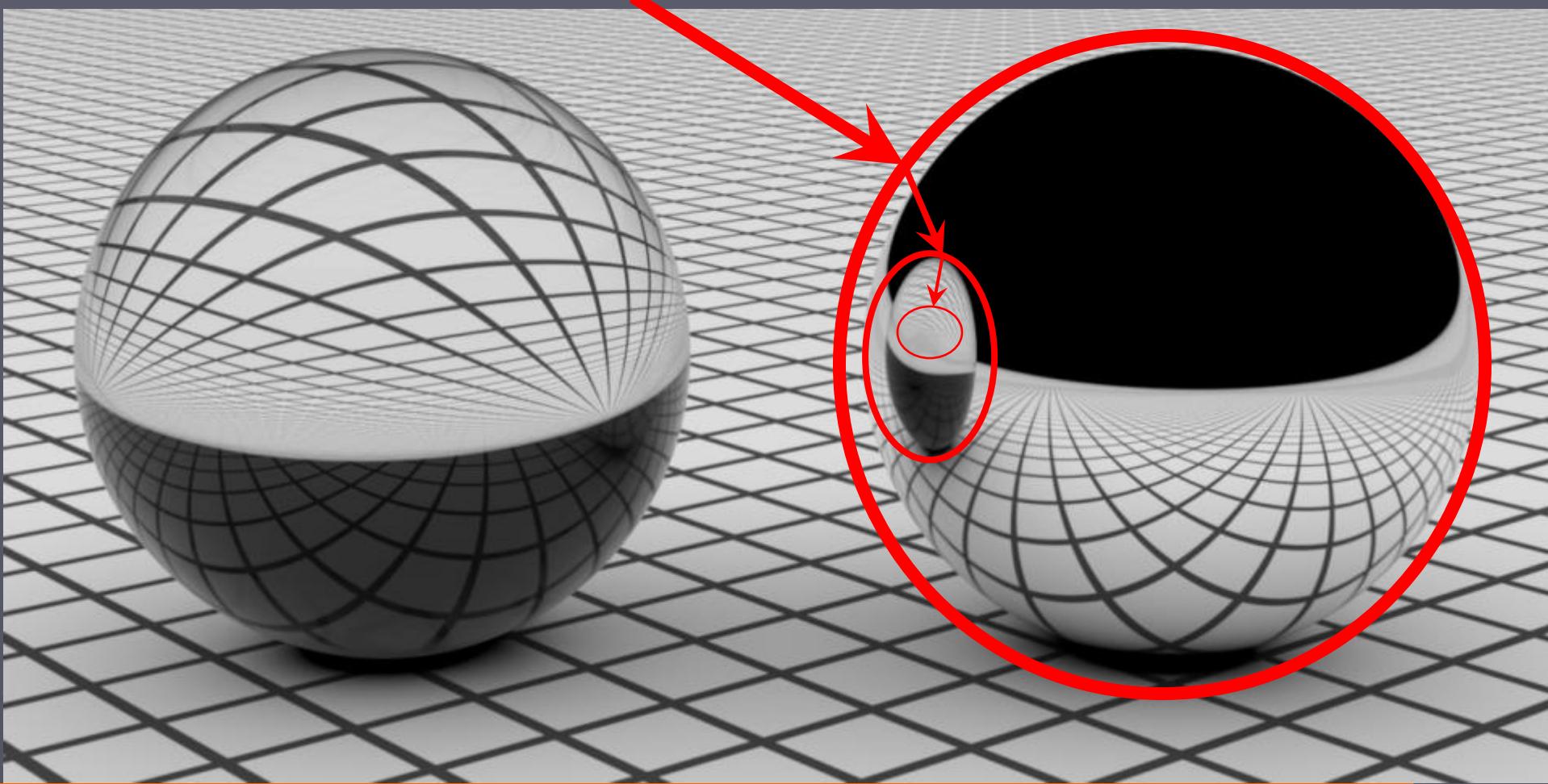
displace hit-point by tiny amount ε

(try $\varepsilon = 10^{-14}$):

- Shadow, reflection rays:
start at $(P_h - \varepsilon \text{ dir})$
- Transparency rays:
start at $(P_h + \varepsilon \text{ dir})$



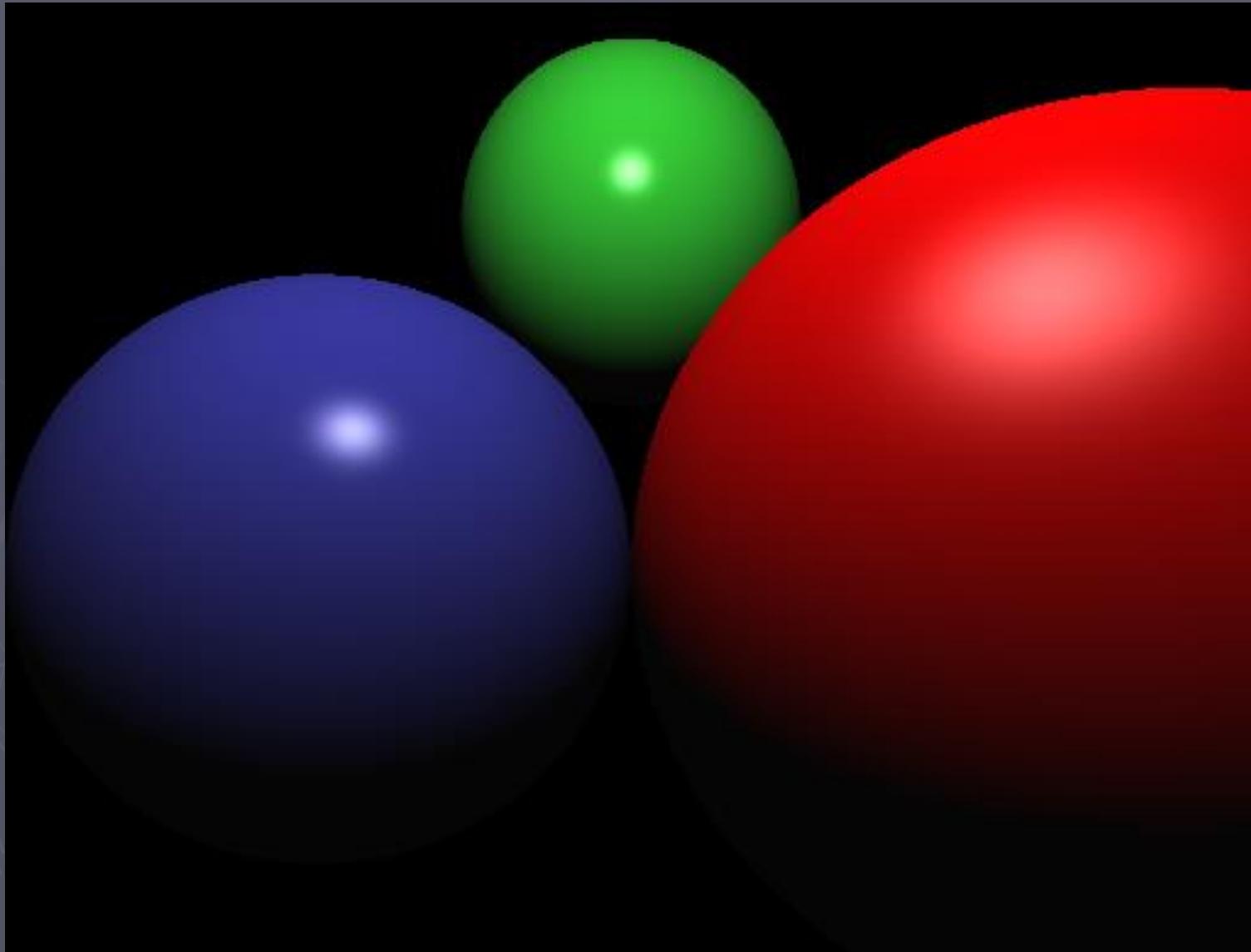
Next!: Recursive 'mirror' Reflection



Matt Pharr, Greg Humphries: Physically Based Rendering Techniques (PBRT)
highly recommended beautiful in-depth book <https://www.pbrt.org/gallery.html>

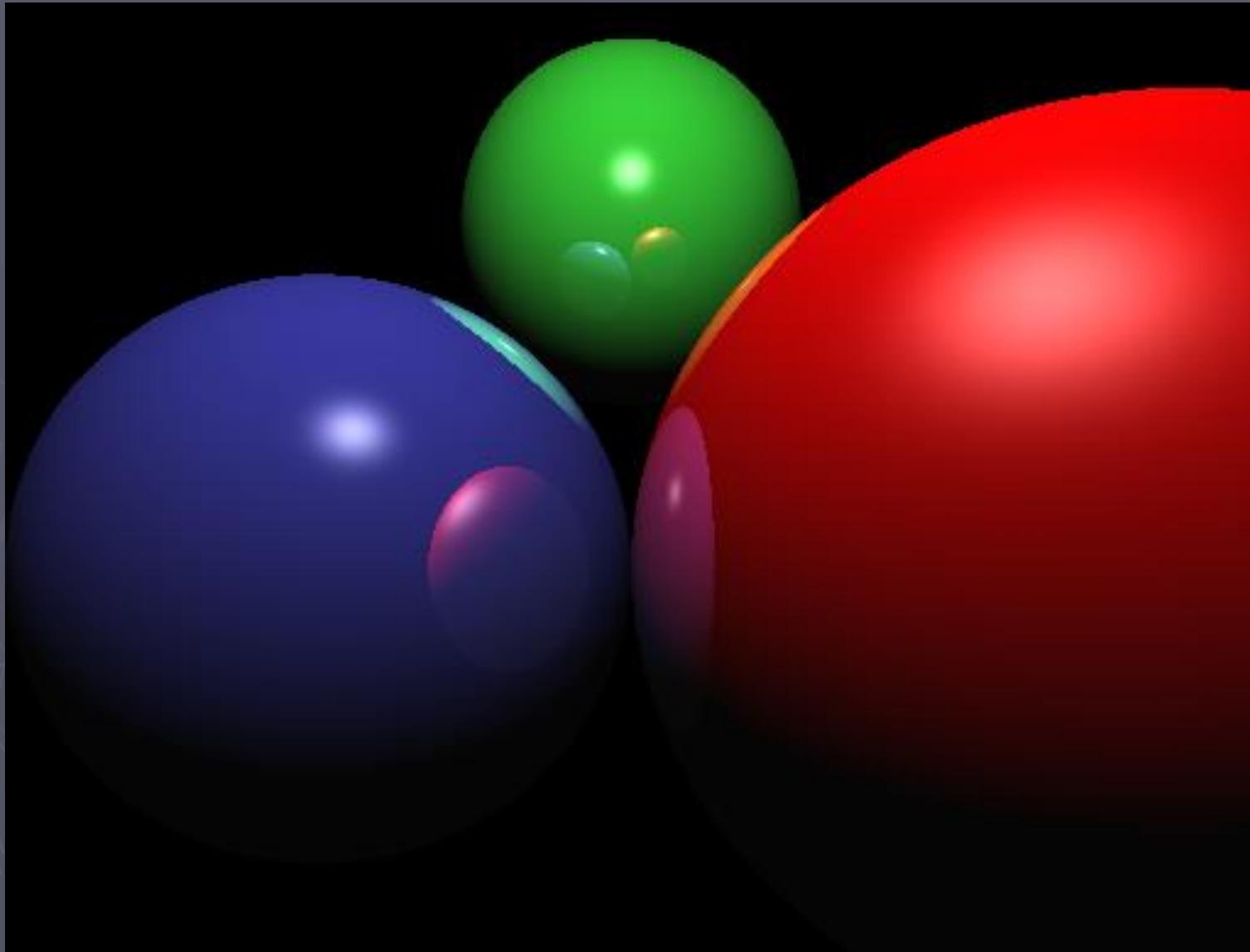
Ray Tracing: Reflection depth=0

(Images From Univ. Delaware)



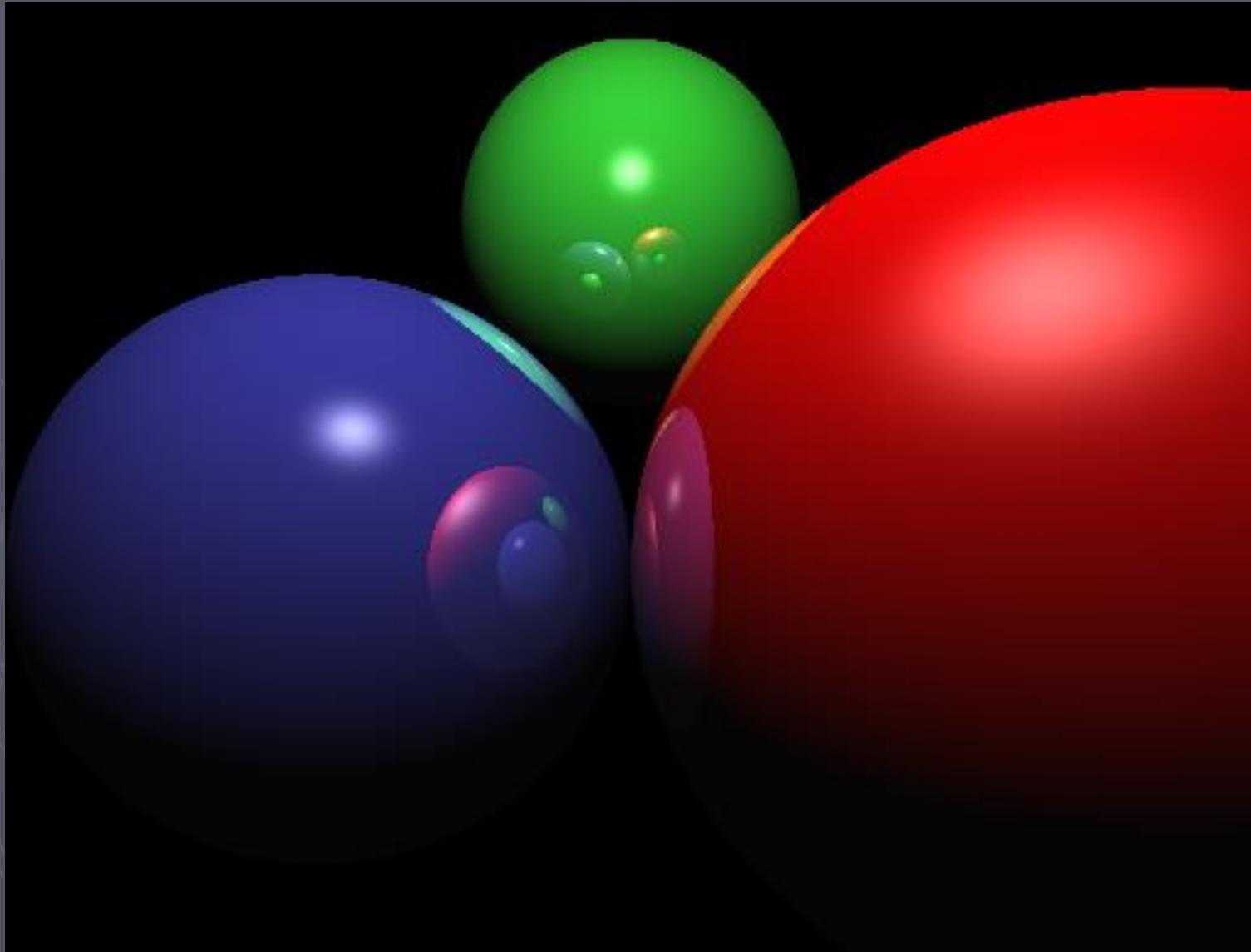
Ray Tracing: Reflection depth=1

(Images From Univ. Delaware)



Ray Tracing: Reflection depth=2

(Images From Univ. Delaware)



Ray-Tracer RECURSION without the agonizing pain...

How should `CScene.makeRayTracedImage()` split up tracing & color-finding?

- ▶ a **Great Big Loop**; makes all pixels in the image buffer (global var: `CImageBuf g_myPic`)
 - Calls `CCamera.makeEyeRay()` for each pixel; this method makes all that pixel's 'eye' ray(s) as needed (`CScene.rayNow`)

To trace the eye Ray and find its color?

SPLIT UP THE TASK!

- Calls `CScene.traceRay(rayNow)` to test ~~all CGeom objects~~ in our `CScene` for ray-intersection.
Returns `CHit` object 'eyeHit'
that holds *what* we hit & *where* we hit it.
- Calls `CScene.findShade(eyeHit)`;
to assess screen color at the end of the ray.
It completes the `CHit` object, gets the final color for the eye ray.
(and recursion may start here)

Ray-Tracer RECURSION without the agonizing pain...

How should `CScene.makeRayTracedImage()` split up tracing & color-finding?

- ▶ a **Great Big Loop**; makes all pixels in the image buffer
(global var: `CImageBuf g_myPic`)
 - Calls `CCamera.makeEyeRay()` for each pixel to make all of that pixel's 'eye' rays (`CScene.rayNow`)

USE THIS FUNCTION PAIR RECURSIVELY:

- Calls `CScene.traceRay(rayNow)` to test *all* `CGeom` objects in our `CScene` for ray-intersection.
Returns `CHit` object 'eyeHit' (or better yet: a LIST of hit points) that holds **what** we hit & **where** we hit it.
- Calls `CScene.findShade(eyeHit)` to assess screen color at the end of the ray.
It completes the `CHit` object, gets the final color for the eye ray.
(and recursion may start here; it may call `traceRay()` ...)

CScene.traceRay(rayNow)

(may grow and need further refactoring later)

- ▶ Trace this ray to find *all* its intersections with *all* CGeom objects in our **CScene**.
- ▶ Returns **CHit** object 'eyeHit' that holds *what* we hit & *where* we hit it
- ▶ LATER:
 - extend **traceRay()** to return **CHitList**: an array of **CHit** objects that describe ALL ray/object intersections (not just the nearest one!)
 - create a '**findHit()**' function that selects the **CHit** object for use by '**findShade()**' function; implements CSG.

CScene.findShade()

(may grow and need further refactoring later)

Find the screen-color at the end of this ray:

compute OpenGL-like shading (Phong light-sources and materials),
but improve it with shadows, reflections, and transparency.

- ▶ Examine the CGeom object we hit:
 - At the hit-point: find/retrieve the materials properties (K_ambient, K_diffuse, K_specular, K_emissive, K_shininess)
 - At the hit-point: find/retrieve the vectors we need for shading: surface normal, view vector, light-direction vectors, reflection vectors, etc.
- ▶ Compute Shadows at hit-point:
 - Create a shadow ray from hit-point to each light source
 - call **CScene.traceRay()** to find what each shadow ray hits, if anything.
Nothing? no shadow at hit-point; enable this light
- ▶ Find the color of reflected ray(s):
 - Create a reflection ray at the hit-point (use surface normal and view ray)
 - Call **CScene.traceRay()** to find **what** and **where** the reflected ray hits (if anything)
 - Call **CScene.findShade()** to **find the color** at the reflection-ray's endpoint.

Week 4 Goals: Just one last thing...

- ▶ 1) Make World-Space Surface Normals → OpenGL-like Phong Lighting
 - Transform normals? → READING: F.S.Hill, Section 14.7, pg 760,1,2;
 - CMatl objects: ambient, diffuse, specular, & shinyness members
 - Move lights with Mouse, Keyboard? More CTransRot objects!
- ▶ 2) Recursion: Show multiple reflections among objects;
 - 1 or more Grid-plane objects, 2 or more Spheres (try 3:  - ORGANIZE:  traceRay() calls findShade(), calls findShade()
 - Extra Credit: transparency/refraction too!
- ▶ 3) At least one 'Solid' (or 'Procedural') Texture
 - CHit: add member(s) for **model-space** hit-point, use it in
 - CMatl: model-space 3D color fcns: $R(x,y,z)$ $G(x,y,z)$ $B(x,y,z)$
 - Functions for Concentric Shells, Line→Grid→Cage, Checkerboards & other 'easy' textures? see FS Hill, Chap 14.8, pp. 770-773

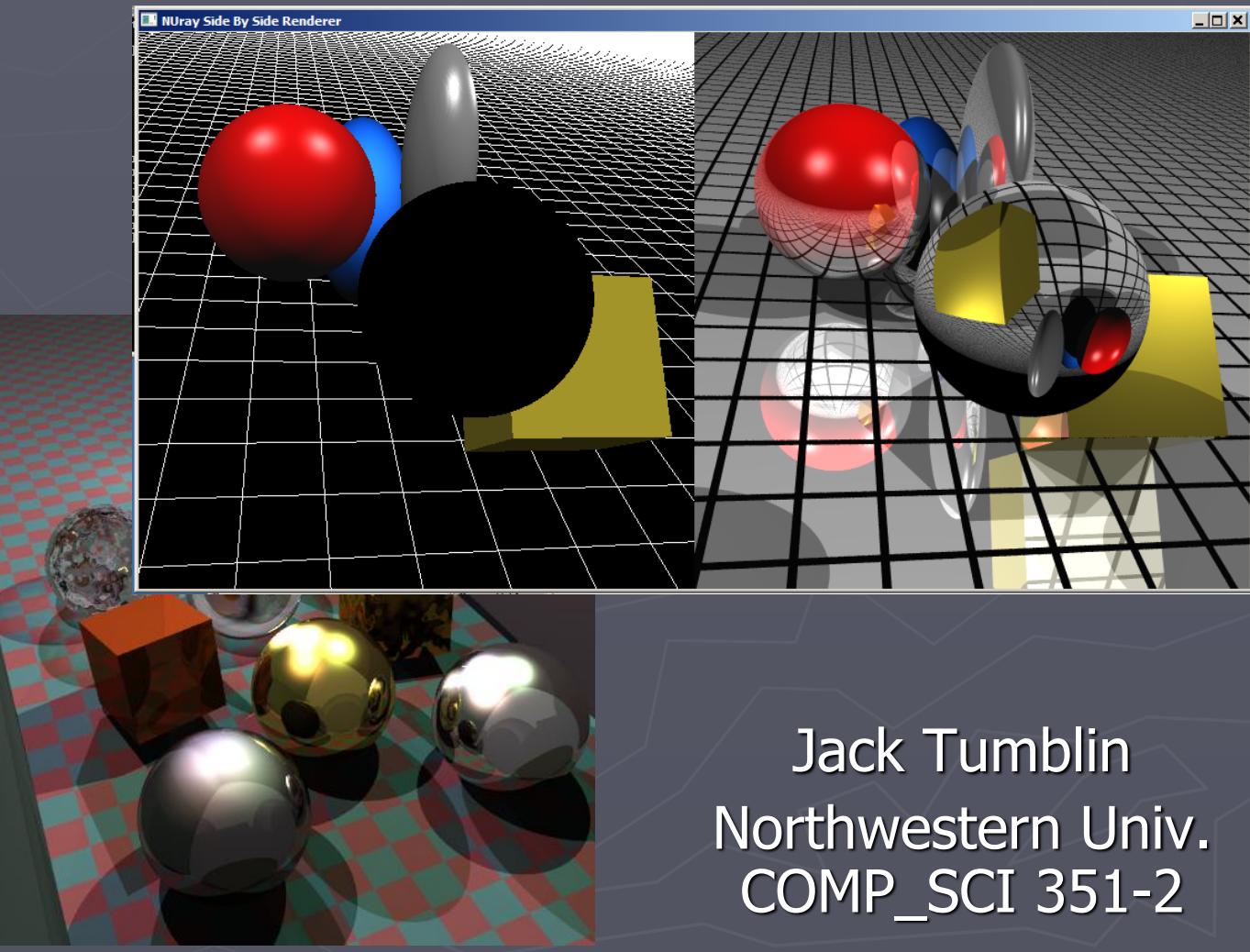
END



END

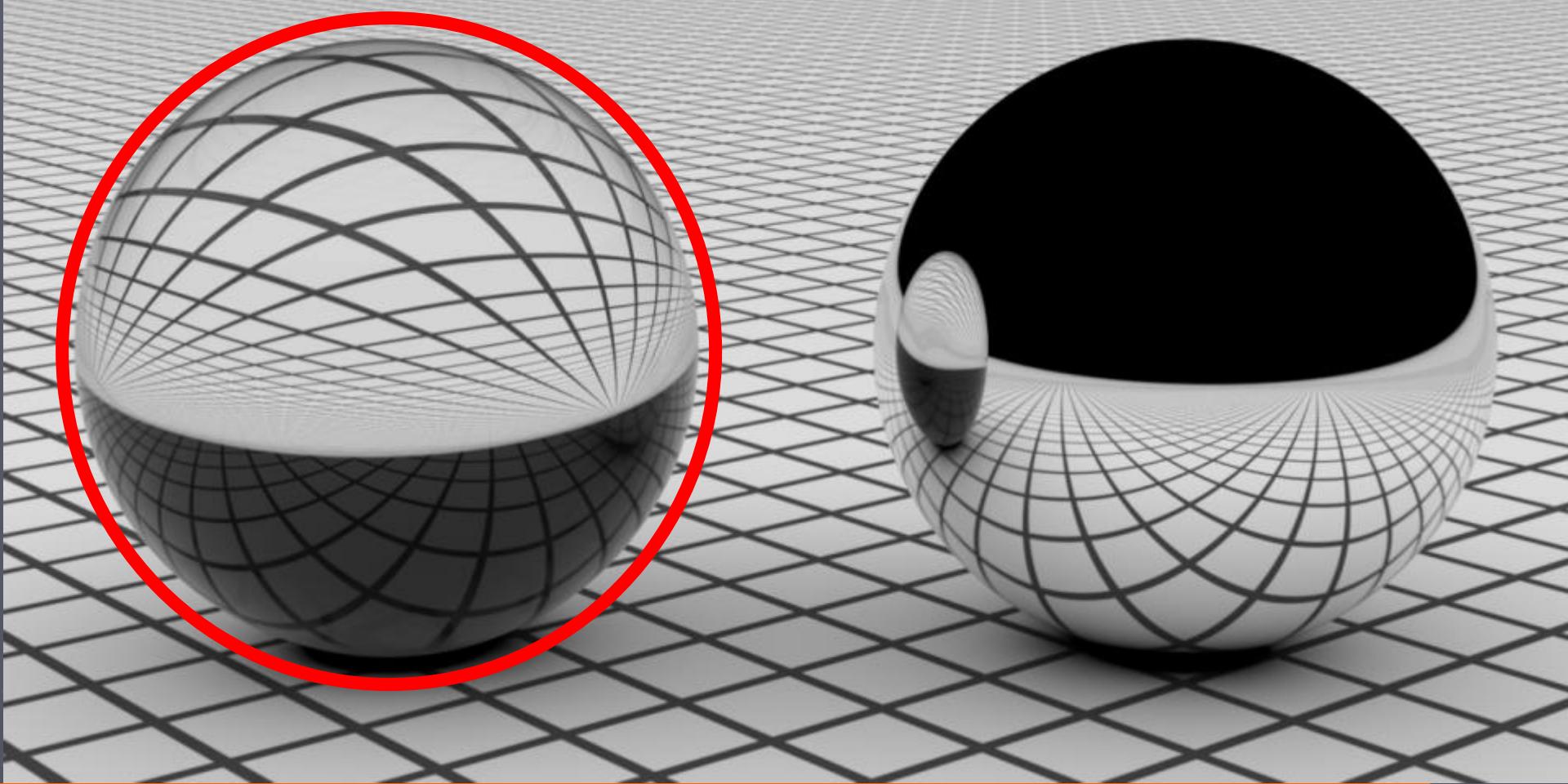


Ray Tracing F: Refraction & Procedural Materials



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

Next!: Transparency and Refraction



Matt Pharr, Greg Humphries: Physically Based Rendering Techniques (**PBRT**)
highly recommended beautiful in-depth book <https://www.pbrt.org/gallery.html>

Transparent Objects: Refraction

- ▶ Light rays **bend** as they cross surface boundary into a different refractive index. (**dir** is reversible)
- ▶ **Snell's Law:**

$$c_1 \sin \theta_1 = c_2 \sin \theta_2$$

- ▶ θ_1, θ_2 : Incidence angles
- ▶ c_1, c_2 : Refractive Index

Refractive Index **c**:

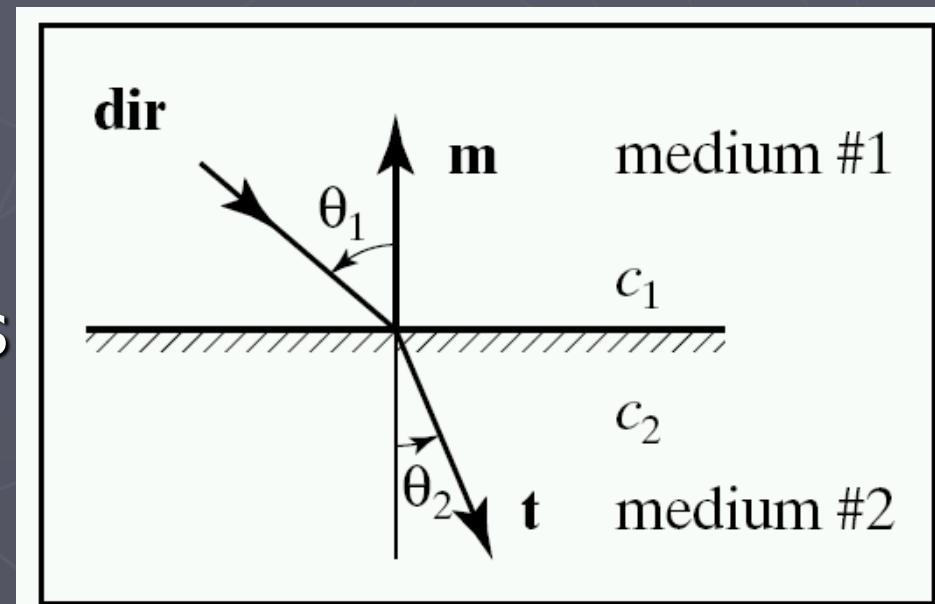
Vacuum: == 1.0 (by definition)

Air, 25° C @1 atm: 1.000293

Water 1.33

Flint Glass 1.6

Silicon 4.0



F.S. Hill, "Computer Graphics using OpenGL" 2nd Edition,
Fig 14.66, Pg 795.

Want More? See Wikipedia & the sources it cites...

Glass & Other Transparent Materials

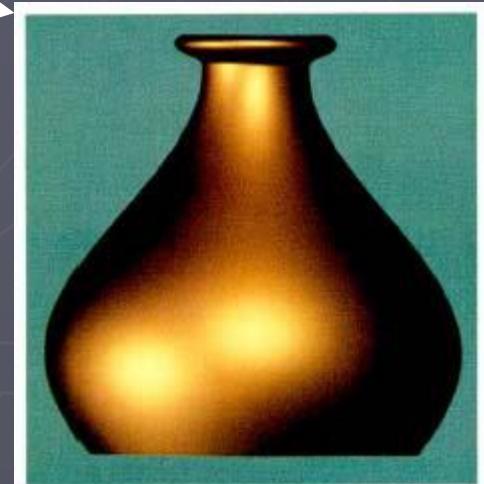
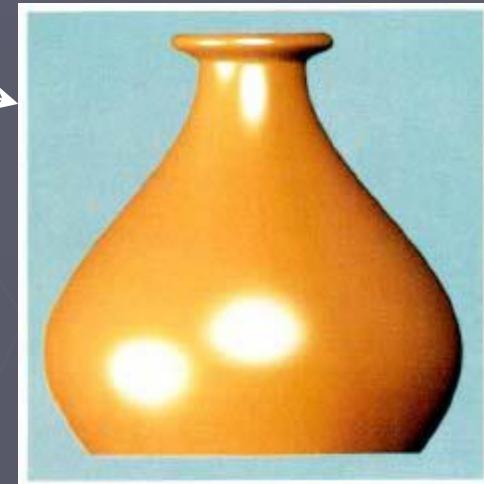
- ▶ Color at glass surface consists of:
 - Phong Shading(or better): $I_a \cdot K_a + I_d \cdot K_d(N \cdot L) + I_s \dots$ etc.
 - PLUS mirror-reflected light,
 - PLUS transmitted light.
- ▶ Even the **clearest** solid material has nonzero ambient, diffuse, specular, and mirror contribs:
 - But *weak* ambient & diffuse terms (usually near zero),
 - *Strong*, very narrow specular lobe (high shininess),
 - *Moderate* mirror-like reflection ray.
- ▶ Augment `findShade()` function:
trace a 'transparency' ray from hit-point.
- ▶ New ray **ADDS** more light to hit-point surface color

Week 4 Goals

- ▶ 1) Make World-Space Surface Normals → OpenGL-like Phong Lighting
 - Transform normals? → READING: F.S.Hill, Section 14.7, pg 760,1,2;
 - CMatl objects: ambient, diffuse, specular, & shinyness members
 - Move lights with Mouse, Keyboard? More CTransRot objects!
- ▶ 2) Recursion: Show multiple reflections among objects;
 - 1 or more Grid-plane objects, 2 or more Spheres (try 3: )
 - ORGANIZE 
 - Extra Credit: transparency/refraction too!
- ▶ 3) At least one 'Solid' (or 'Procedural') Texture
 - CHit: add member(s) for model-space hit-point, use it in
 - CMatl: model-space 3D color fcns: $R(x,y,z)$ $G(x,y,z)$ $B(x,y,z)$
 - Functions for Concentric Shells, Line→Grid→Cage, Checkerboards & other 'easy' textures? see FS Hill, Chap 14.8, pp. 770-773
 - Advanced? <http://www.upvector.com/?section=Tutorials&subsection=Intro%20to%20Procedural%20Textures>

We can do better than Phong!

- ▶ The very first ACM TOG paper (1981):
Cook-Torrance Shading Model
 - Reflectance from statistical micro-facets
 - Greatly improved metals depictions
- ▶ **Many** others even more accurate,
compared to physical measurements of
BRDF, BTF,... See: **Torrance-Sparrow, He,**
LaFortune, etc.
- ▶ Many good refinements can
now approach perfection:
Combine global illumination
with accurate materials
models and measured data:
BRDF, BTF, BSSRDF, etc



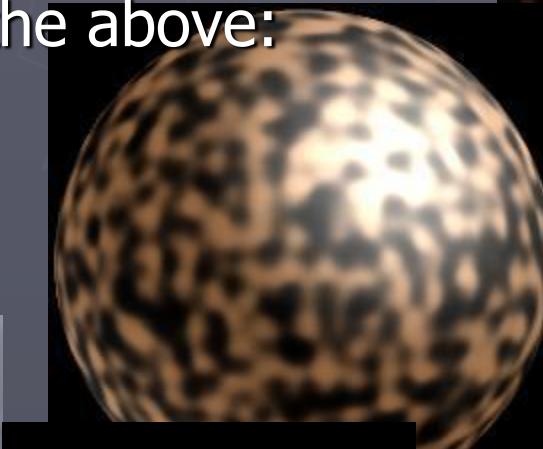
Procedural Materials



- ▶ **AMAZINGLY VERSATILE!**
- ▶ **Idea:** define each material parameter as a scalar function of 3D position (model space): *e.g.* color, or diffuse reflectance or normal or specular from (x,y,z) :
 $K_d_{red}(x,y,z)$ $K_d_{grn}(x,y,z)$ $K_d_{blu}(x,y,z)$
- ▶ Evaluate these matl-parameter functions at the ray/object hit-point, & apply material described.

Some Good Examples: Try it!

- ▶ 3D 'xyz line-grid' (extend your 2D plane-grid)
- ▶ 3D Sine-Wave series; hyperbolic functions
- ▶ 3D Concentric Shells/ radial functions
- ▶ 3D Concentric Cylinders (wood-like texture)
- ▶ Randomly Displace any of the above:
See 'Perlin Noise'
See "Turbulence Fcns"
See "Hypertexture"
See "Fractal Textures", etc.



How To Build A 3D Checkerboard

► IDEA:

- Add up integer-only parts of model-space x,y,z.
- Odd result? → Material A,
Even result? → Material B.

Some pseudo-code:

```
result =int(x) + int(y) + int(z);  
if(result%2) { myMatl = MatlA; }  
else { myMatl = MatlB; };
```

► Try it, and ...

How To Build A 3D Checkerboard

► IDEA:

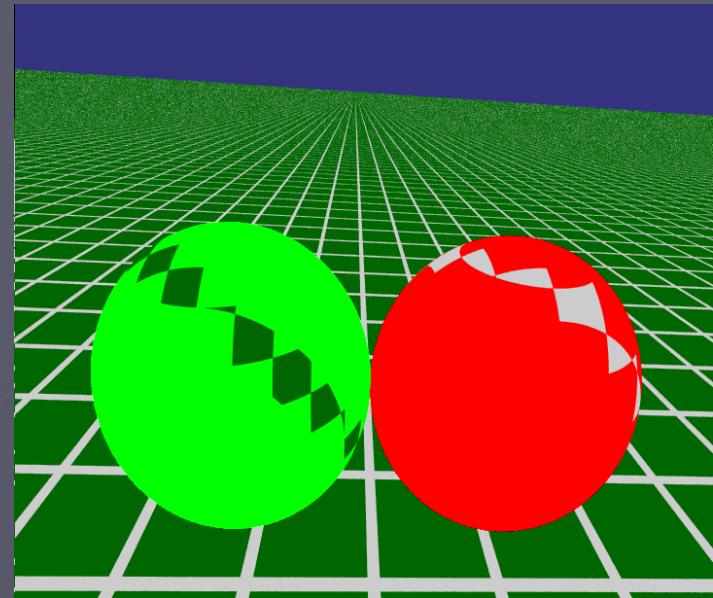
- Add up the integer-only parts of x, y, z
- Odd result? \rightarrow Material A
Even result? \rightarrow Material B

► IMPLEMENTATION:

```
result = Math.floor(x/xgap)
        + Math.floor(y/ygap)
        + Math.floor(z/zgap);

if(result%2) myMat = Mat1A;
else myMat = Mat1B;
```

► WEIRD RESULTS! ?!?! Why ?!?!



How To Build A 3D Checkerboard

A Bit Tricky Because:

- ▶ JavaScript `Math.floor()` function is *signed*:
 $\text{Math.floor}(1.5)$ value is 1.0
 $\text{Math.floor}(0.5)$ value is 0.0
 $\text{Math.floor}(-0.5)$ value is -1.0
- ▶ JavaScript modulus (`%`) is *signed*:
 $x \% 2$ can yield negative numbers!
 $(-3) \% 2$ value is -1
 $(3) \% 2$ value is +1

How To Build A 3D Checkerboard

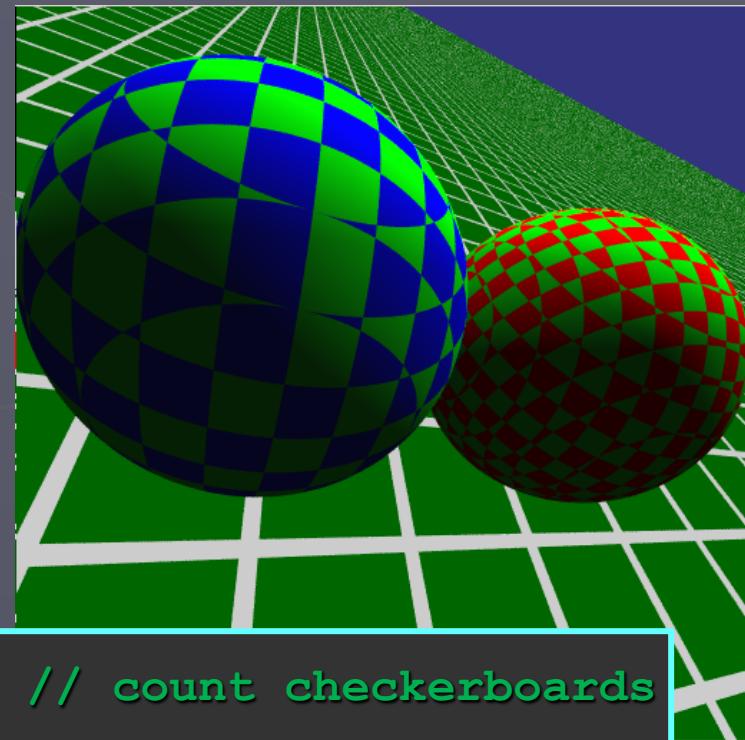
► SOLUTION:

- Easy -- Absolute value!
- odd result? → Material A
even result? → Material B

Given hit-point (x, y, z) :

```
tot = Math.floor(x/xgap) + // count checkerboards
      Math.floor(y/ygap) +
      Math.floor(z/zgap));

if(tot<0)y = -y;           // absolute value,
if(y >0.5) myMatl = Mat1A; // discard tiny errors
else myMatl = Mat1B;
```



OTHER 3D Textures?

► Randomized materials:

- Randomized ambient, diffuse, specular, mirror reflectances: save min/max values, save 'seed' value for **srand()** to ensure repeatability.
- **Perlin Noise**: better looking, more controllable, but more complex (google it)
 - 'Hypertexture' Methods

► Function-Generator Methods:

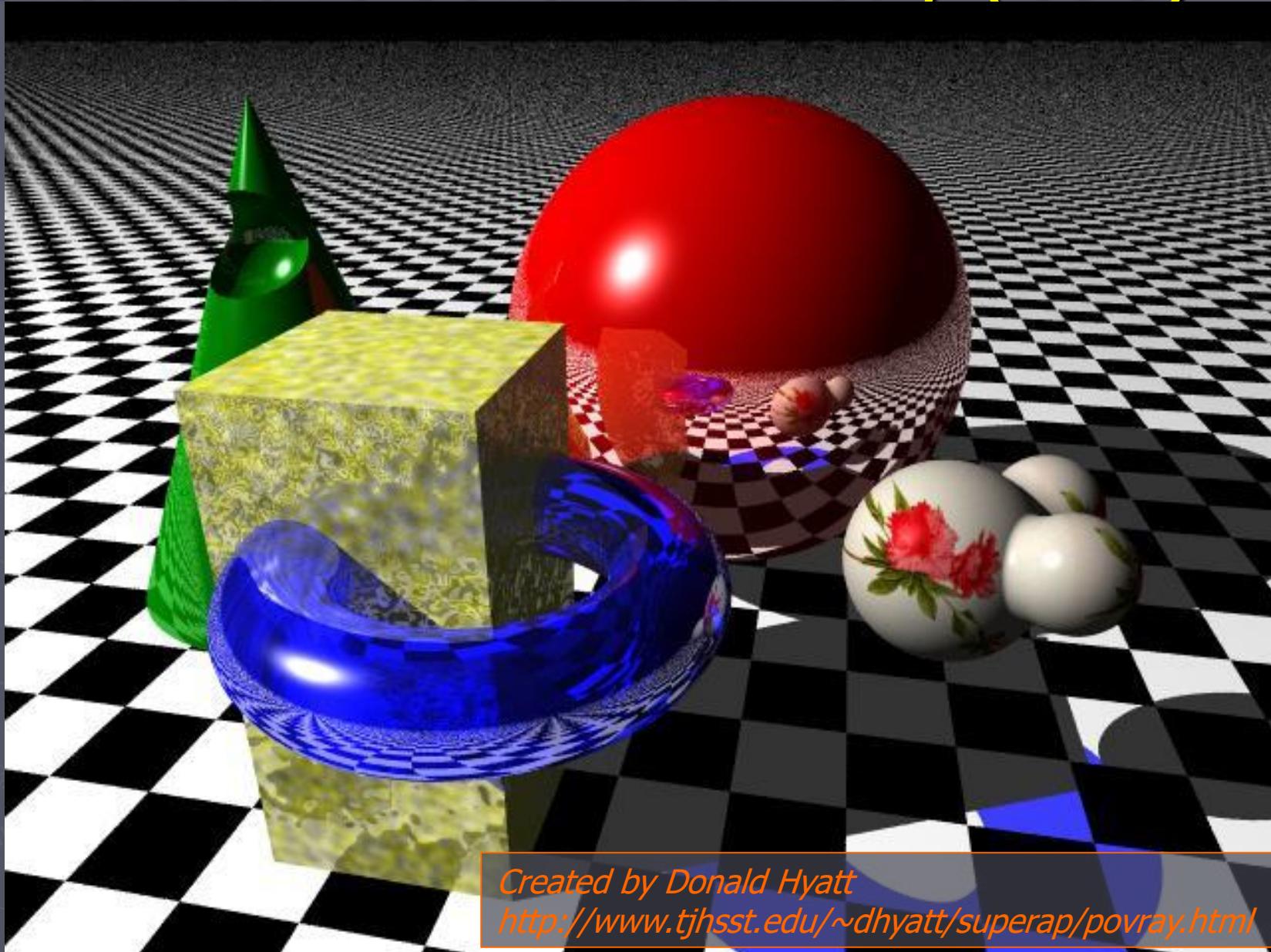
- Cylindrical sine-waves → Wood
- 'Turbulence' Functions (Perlin) for wavyness...

Constructive Solid Geometry (CSG)

- ▶ Your ray tracer already builds **CHitList**: use it as a 'Priority-Queue' of all ray hit-points;
- ▶ Be sure to collect ALL the ray hitpoints, (even those behind the camera!)
- ▶ Apply Boolean Logic to decide where the ray hits the 'constructed' surface:
 - Union (+)
 - Difference (-)
 - Intersection (*)

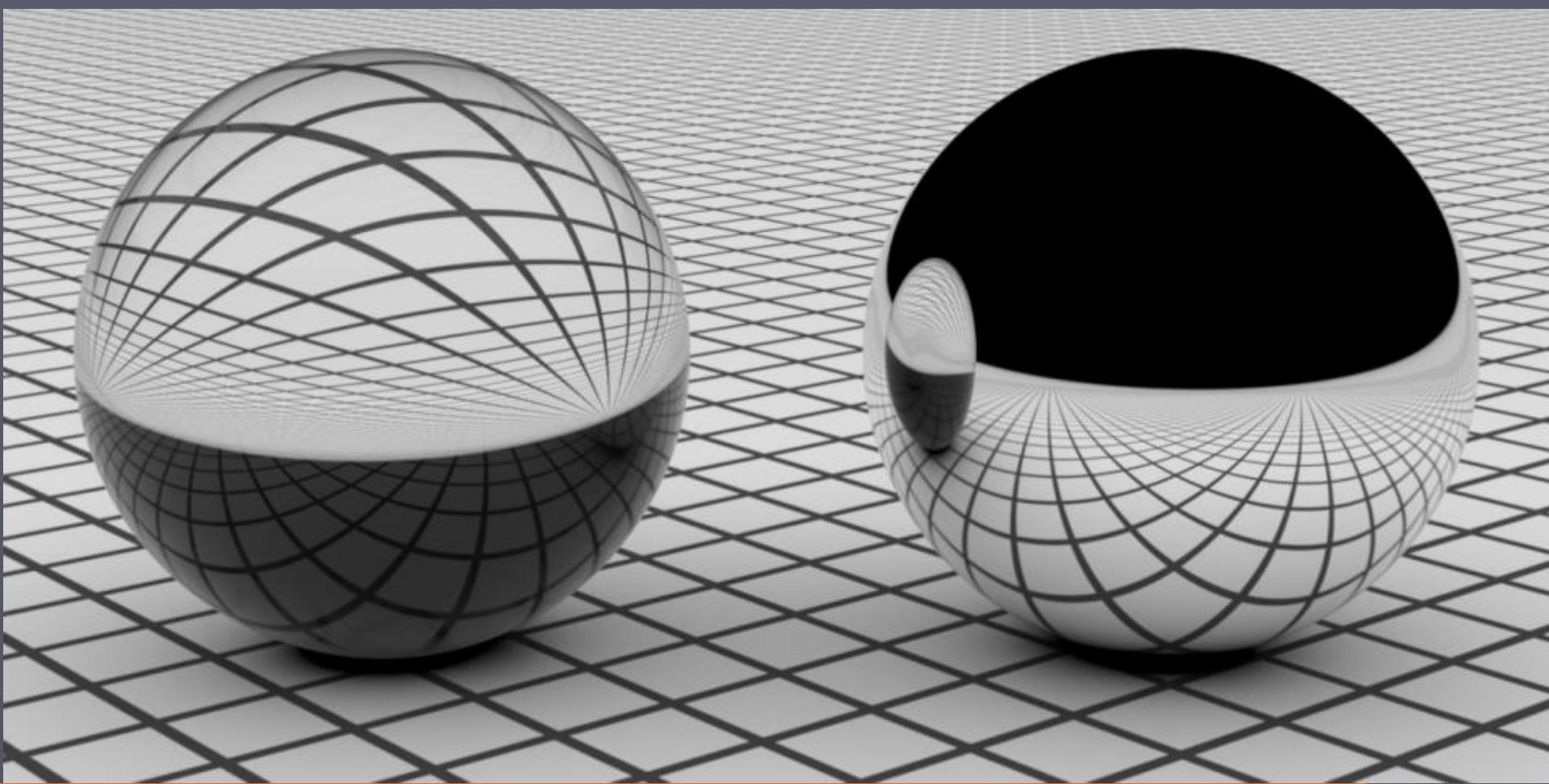
Constructive Solid Geometry (CSG)

► Ex



Created by Donald Hyatt
<http://www.tjhsst.edu/~dhyatt/superap/povray.html>

Reminder: DEMO DAY Thurs Mar 11



Matt Pharr, Greg Humphries: Physically Based Rendering Techniques (**PBRT**)
highly recommended beautiful in-depth book <http://www.pbrt.org/gallery.php>

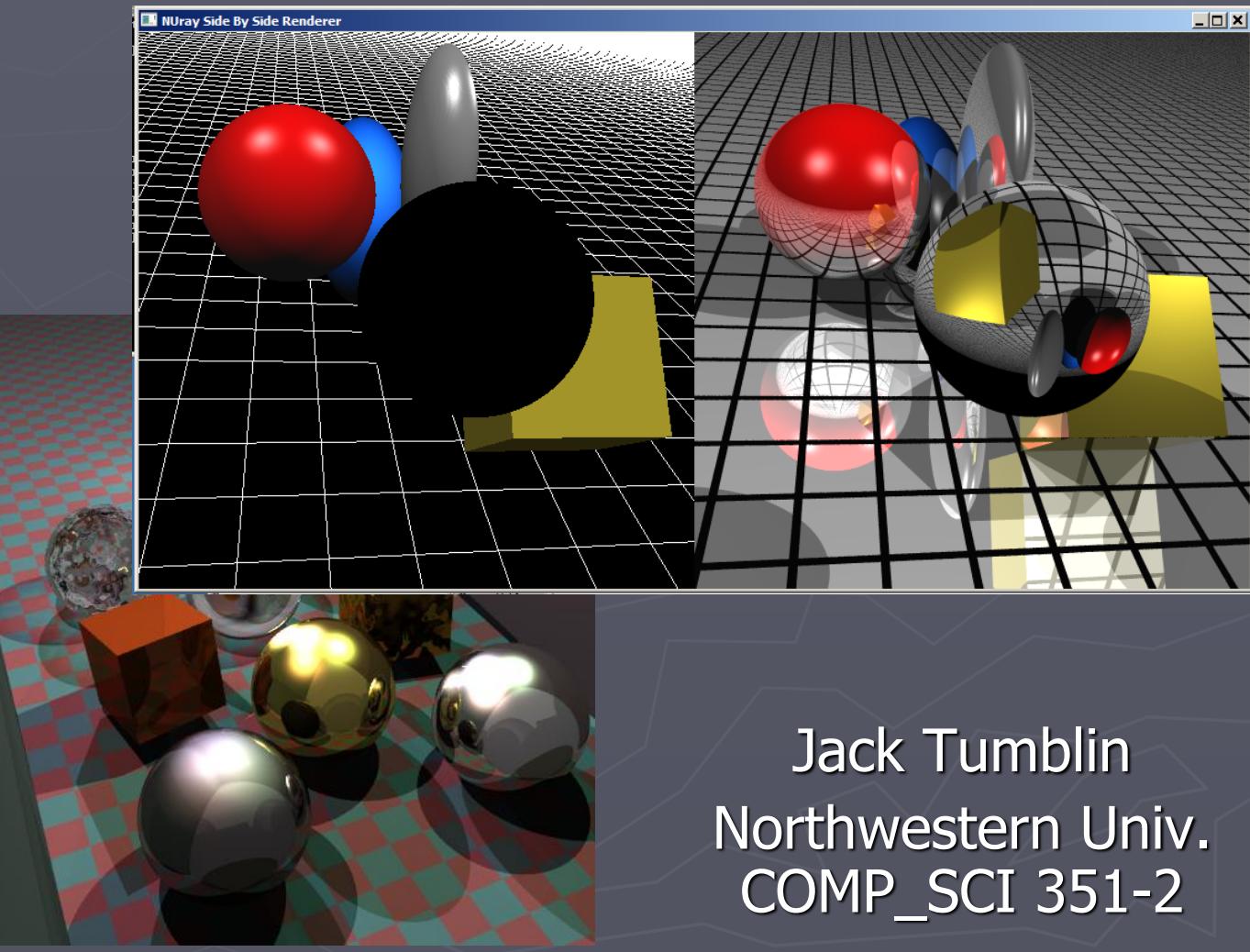
END



END

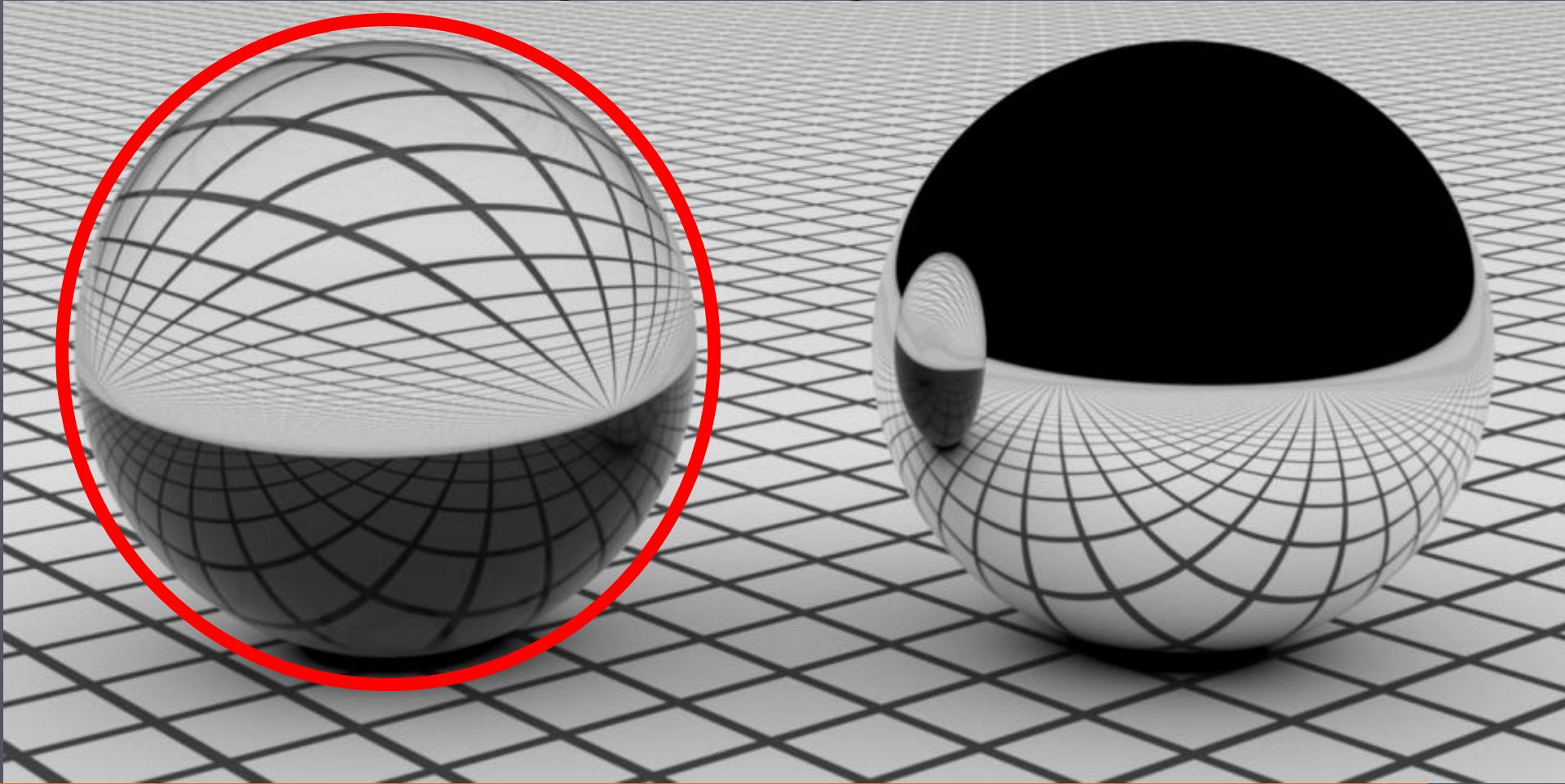


Ray Tracing F: Refraction & Procedural Materials



Jack Tumblin
Northwestern Univ.
COMP_SCI 351-2

END!: Transparency and Refraction



Matt Pharr, Greg Humphries: Physically Based Rendering Techniques (**PBRT**)
highly recommended beautiful in-depth book <https://www.pbrt.org/gallery.html>

Transparent Objects: Refraction

- ▶ Light rays **bend** as they cross surface boundary into a different refractive index. (**dir** is reversible)
- ▶ **Snell's Law:**

$$c_1 \sin \theta_1 = c_2 \sin \theta_2$$

- ▶ θ_1, θ_2 : Incidence angles
- ▶ c_1, c_2 : Refractive Index

Refractive Index **c**:

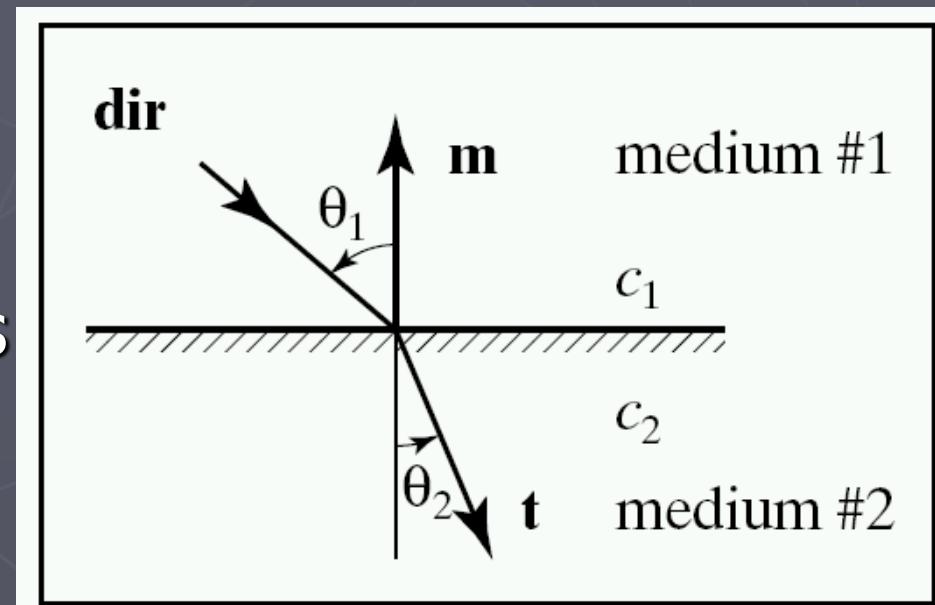
Vacuum: == 1.0 (by definition)

Air, 25° C @1 atm: 1.000293

Water 1.33

Flint Glass 1.6

Silicon 4.0



F.S. Hill, "Computer Graphics using OpenGL" 2nd Edition,
Fig 14.66, Pg 795.

Want More? See Wikipedia & the sources it cites...

Glass & Other Transparent Materials

- ▶ Color at glass surface consists of:
 - Phong Shading(or better): $I_a \cdot K_a + I_d \cdot K_d(N \cdot L) + I_s \dots$ etc.
 - PLUS mirror-reflected light,
 - PLUS transmitted light.
- ▶ Even the **clearest** solid material has nonzero ambient, diffuse, specular, and mirror contribs:
 - But: *weak* ambient & diffuse terms (usually near zero),
 - *Strong*, very narrow specular lobe (high shininess),
 - *Mild* mirror-like reflection ray (due to index change).

HOW?

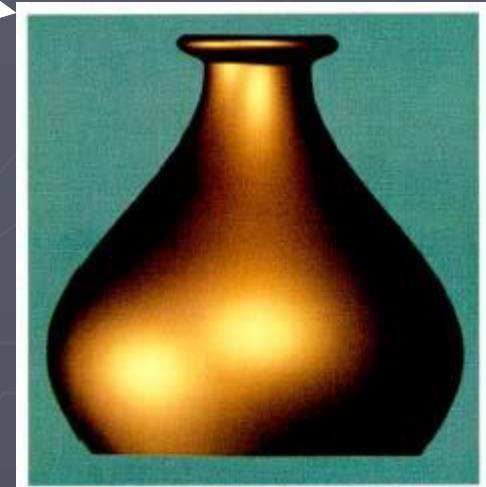
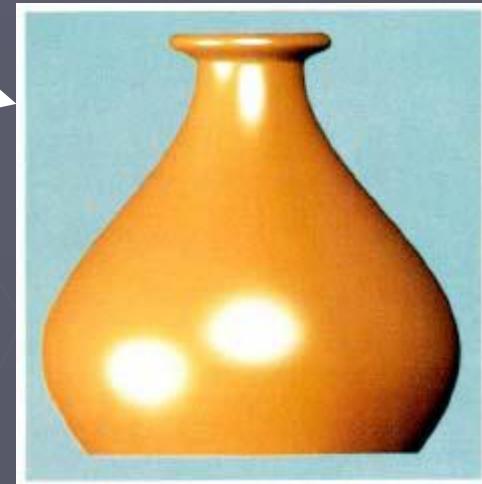
- ▶ Augment `findShade()` function:
trace a 'transparency' ray from hit-point.
- ▶ This ray **ADDS** more light to hit-point surface color

Week 4 Goals

- ▶ 1) Make World-Space Surface Normals → OpenGL-like Phong Lighting
 - Transform normals? → READING: F.S.Hill, Section 14.7, pg 760,1,2;
 - CMatl objects: ambient, diffuse, specular, & shinyness members
 - Move lights with Mouse, Keyboard? More CTransRot objects!
- ▶ 2) Recursion: Show multiple reflections among objects;
 - 1 or more Grid-plane objects, 2 or more Spheres (try 3: )
 - ORGANIZE 
 - Extra Credit: transparency/refraction too!
- ▶ 3) At least one 'Solid' (or 'Procedural') Texture
 - CHit: add member(s) for model-space hit-point, use it in
 - CMatl: model-space 3D color fcns: $R(x,y,z)$ $G(x,y,z)$ $B(x,y,z)$
 - Functions for Concentric Shells, Line→Grid→Cage, Checkerboards & other 'easy' textures? see FS Hill, Chap 14.8, pp. 770-773
 - Advanced? <http://www.upvector.com/?section=Tutorials&subsection=Intro%20to%20Procedural%20Textures>

We can do better than Phong!

- ▶ The very first ACM TOG paper (1981):
Cook-Torrance Shading Model
 - Reflectance from statistical micro-facets
 - Greatly improved metals depictions
- ▶ **Many** others even more accurate,
compared to physical measurements of
BRDF, BTF,... See: **Torrance-Sparrow, He,**
LaFortune, etc.
- ▶ With more refinements, can
now approach perfection:
Approx. global illumination
with accurate materials
models and measured data:
BRDF, BTF, BSSRDF, etc



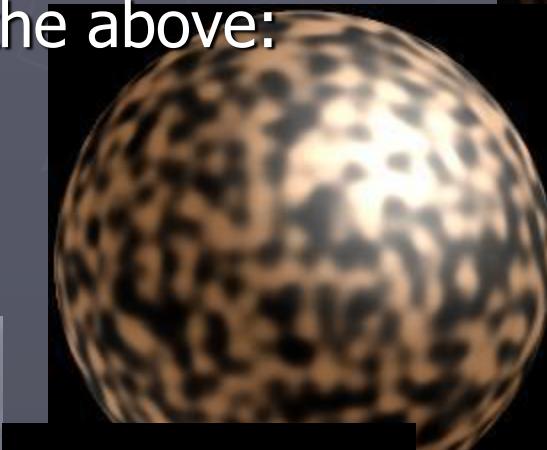
Procedural Materials



- ▶ **AMAZINGLY VERSATILE!**
- ▶ **Idea:** define each material parameter as a scalar function of model-space (x,y,z): *e.g.*
 $Kd_red(x,y,z)$; $Kd_grn(x,y,z)$; $Kd_blu(x,y,z)$;
how about transparency? Refractive index?
- ▶ Evaluate these matl-parameter functions at the ray/object hit-point, & apply material described.

Some Good Examples: Try it!

- ▶ 3D 'xyz line-grid' (extend your 2D plane-grid)
- ▶ 3D Sine-Wave series; hyperbolic functions
- ▶ 3D Concentric Shells/ radial functions
- ▶ 3D Concentric Cylinders (wood-like texture)
- ▶ Randomly Displace any of the above:
See 'Perlin Noise'
See "Turbulence Fcns"
See "Hypertexture"
See "Fractal Textures", etc.



How To Build A 3D Checkerboard

► IDEA:

- Add up integer-only parts of model-space x,y,z.
- Odd result? → Material A,
Even result? → Material B.

Some pseudo-code (BAD!) :

```
result =int(x) + int(y) + int(z);  
if(result%2) { myMatl = MatlA; }  
else { myMatl = MatlB; };
```

► Try it, and ...

How To Build A 3D Checkerboard

- ▶ Wait, Wait -- Let's make it adjustable too!

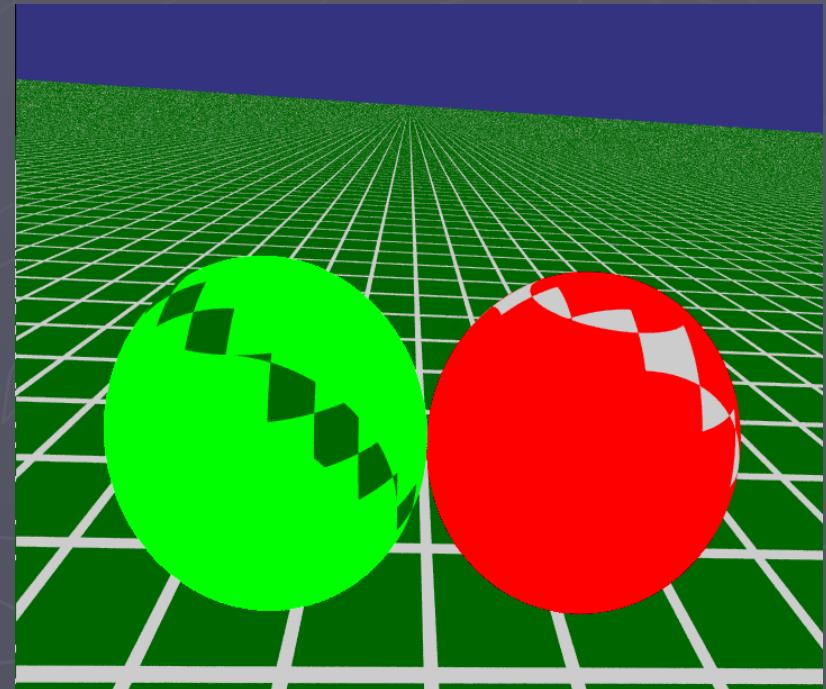
```
result = Math.floor(x/xgap) // Adjustable!
        + Math.floor(y/ygap)
        + Math.floor(z/zgap);

if(result%2)myMat = Mat1A;
else myMat1 = Mat1B;
```

- ▶ WEIRD RESULTS! ?!?! Why ?!?!
(Ans: 2 flaws with negative #s)

- ▶ !CAREFUL!

What happens when
any, some, or all of X, Y, or Z
are <0? Is 'result' ever <0?



How To Build A 3D Checkerboard

A Bit Tricky Because:

- ▶ JavaScript `Math.floor()` function is *signed*:
 $\text{Math.floor}(1.5)$ value is 1.0
 $\text{Math.floor}(0.5)$ value is 0.0
 $\text{Math.floor}(-0.5)$ value is -1.0
- ▶ JavaScript modulus (%) is *signed*:
 $x \% 2$ can yield negative numbers!
 $(-3) \% 2$ value is -1
 $(3) \% 2$ value is +1

How To Build A 3D Checkerboard

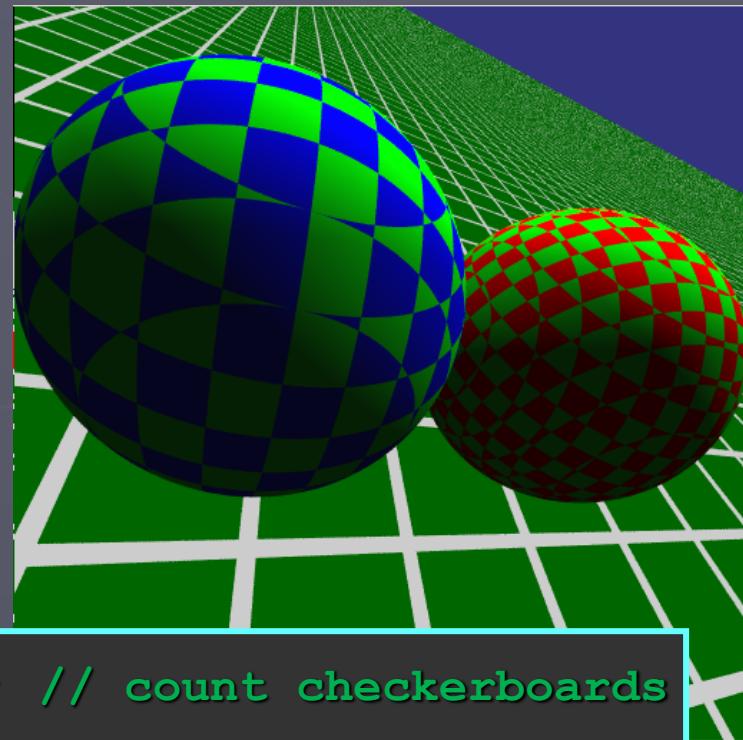
► SOLUTION:

- Easy -- Absolute value!
- odd result? → Material A
even result? → Material B

Given hit-point (x,y,z) :

```
tot = Math.floor(x/xgap) + // count checkerboards
      Math.floor(y/ygap) +
      Math.floor(z/zgap));

if(tot<0) y = -y;           // absolute value,
if(y >0.5) myMatl = Mat1A; // discard tiny errors
else myMatl = Mat1B;
```



OTHER 3D Textures?

► Randomized materials:

- Randomized ambient, diffuse, specular, mirror reflectances: save min/max values, save 'seed' value for **srand()** to ensure repeatability.
- **Perlin Noise**: better looking, more controllable, but more complex (try a [web search](#) ...)
 - 'Hypertexture' Methods (see [this pdf](#))

► Function-Generator Methods:

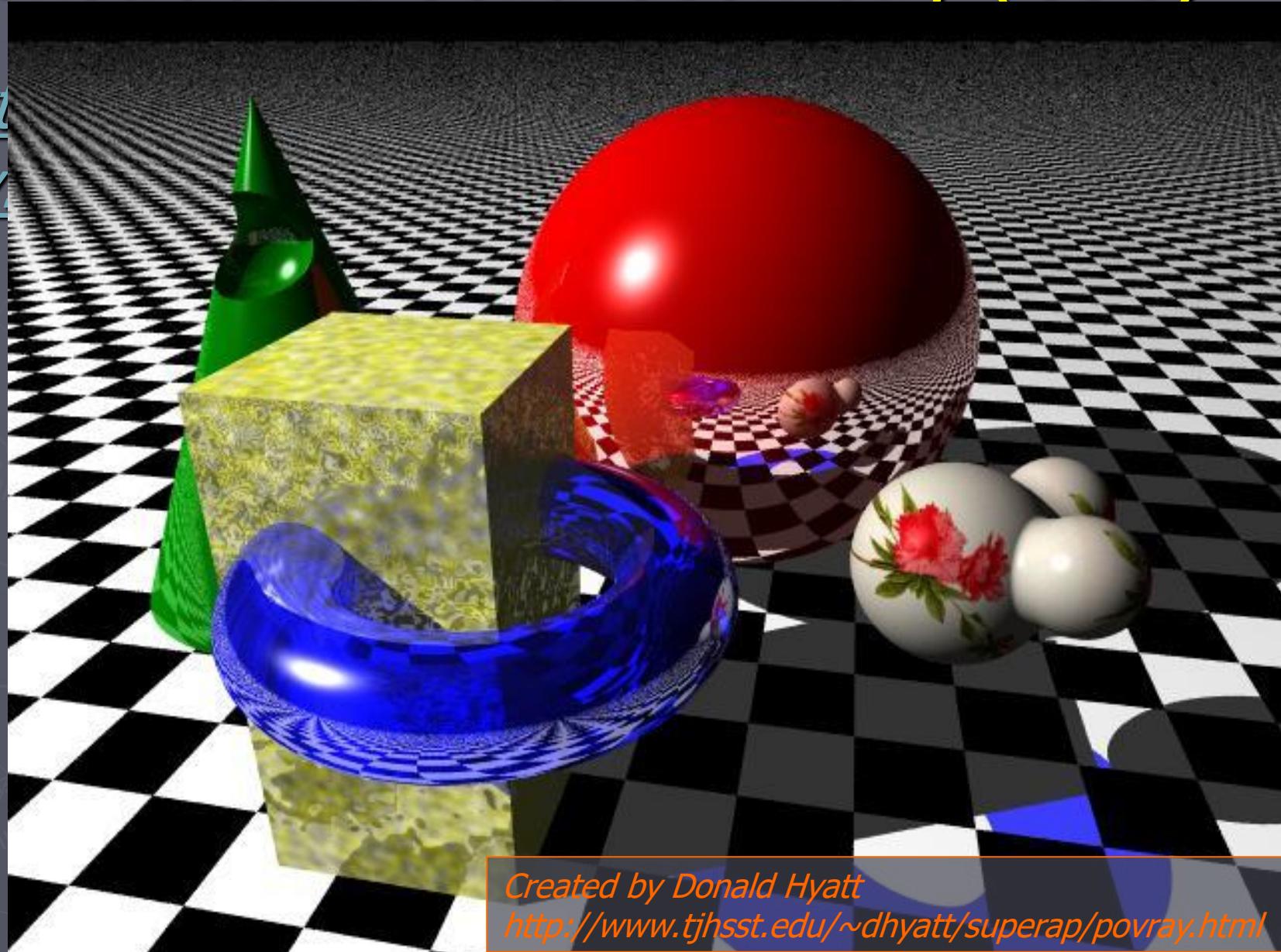
- Cylindrical sine-waves → Wood
- 'Turbulence' Functions (Perlin) [for wavyness...](#)

Constructive Solid Geometry (CSG)

- ▶ Your ray tracer already builds **CHitList**: use it as a 'Priority-Queue' of all ray hit-points;
- ▶ Be sure to collect ALL the ray hitpoints, (even those behind the camera!)
- ▶ *THEN* you can do this ... → → →

Constructive Solid Geometry (CSG)

► <http://www.tjhsst.edu/~dhyatt/superap/povray.html>

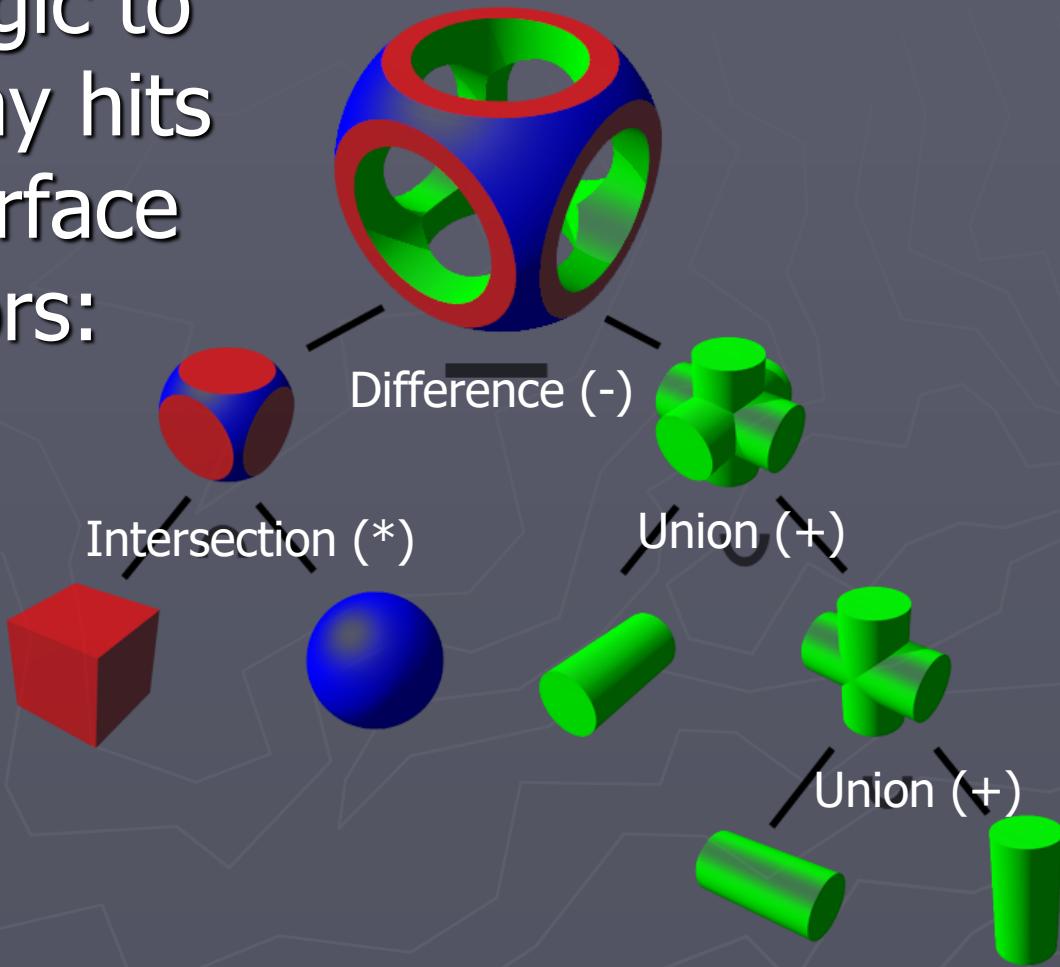


Created by Donald Hyatt
<http://www.tjhsst.edu/~dhyatt/superap/povray.html>

Constructive Solid Geometry (CSG)

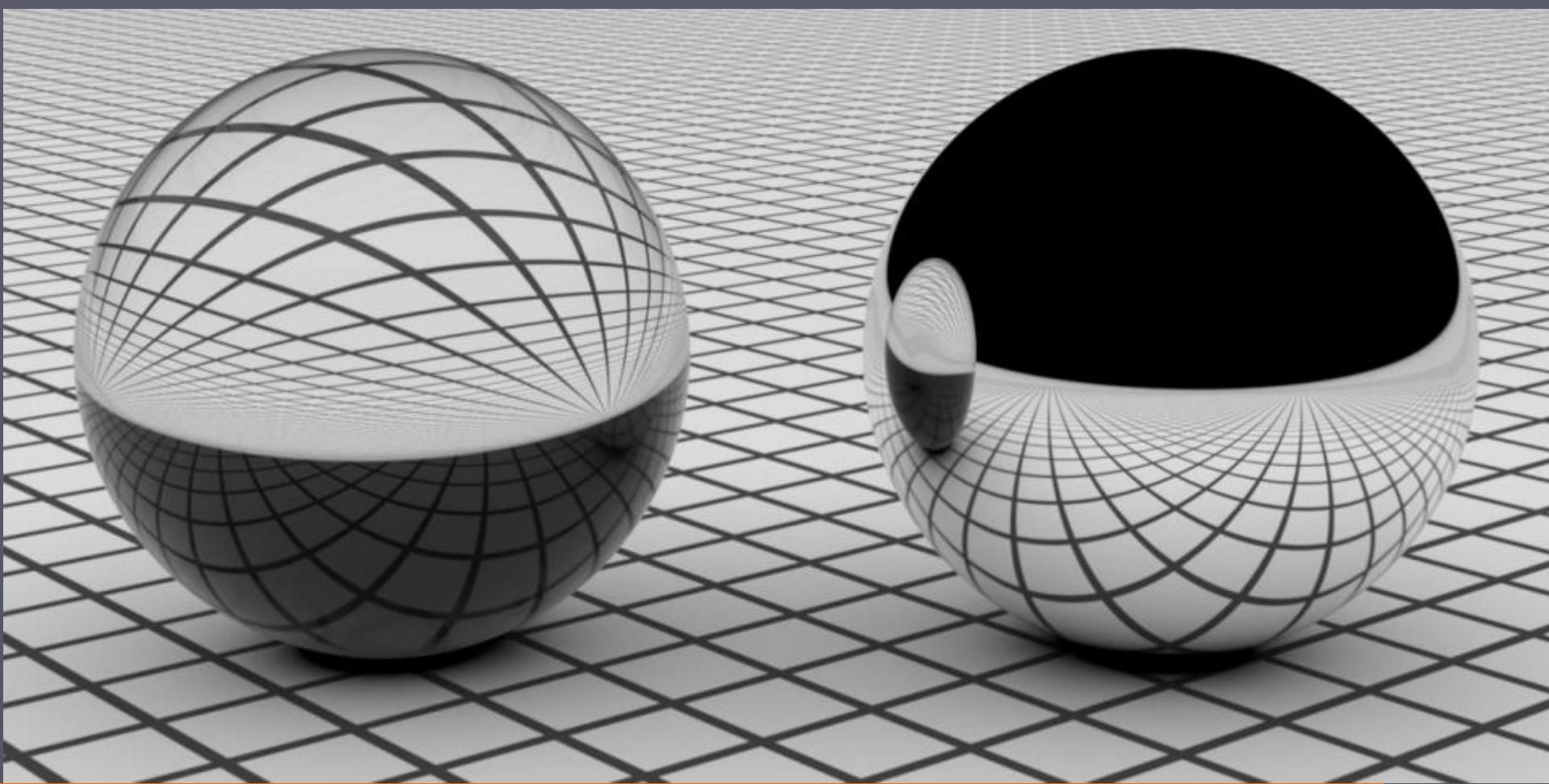
- ▶ Apply Boolean Logic to decide where a ray hits a 'constructed' surface built by 3 operators:

- Union (+)
- Difference (-)
- Intersection (*)



https://en.wikipedia.org/wiki/Constructive_solid_geometry

Reminder: DEMO DAY Thurs Mar 10



Matt Pharr, Greg Humphries: Physically Based Rendering Techniques (**PBRT**)
highly recommended beautiful in-depth book <http://www.pbrt.org/gallery.php>

END

END

