Ray Tracing 2

Shading

Last Time

Quick reminder how to transform the image plane into canonical representation

---

Now in 3D

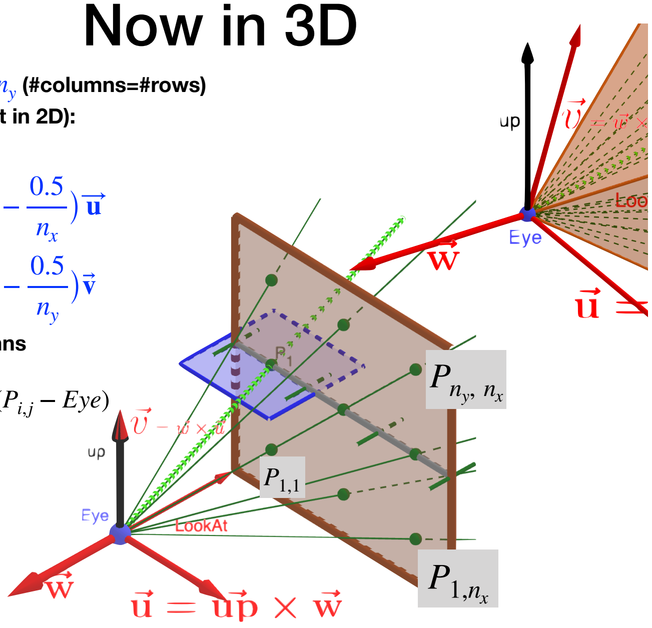
Assume first $n_x = n_y$ **(#columns=#rows)**

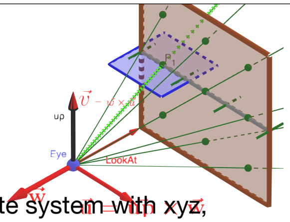
Witness points (first in 2D):

$$P_{i,j} = Eye - D\vec{\mathbf{w}} + \left(2\frac{j}{n_x} - 1 - \frac{0.5}{n_x}\right)\vec{\mathbf{u}} + \left(2\frac{i}{n_y} - 1 - \frac{0.5}{n_y}\right)\vec{\mathbf{v}}$$

$i, j = 1,2,\dots$**#columns**

Ray r: $r = Eye + t(P_{i,j} - Eye)$





Assume first $n_x = n_y$ (#columns=#rows)

Witness points (first in 2D):

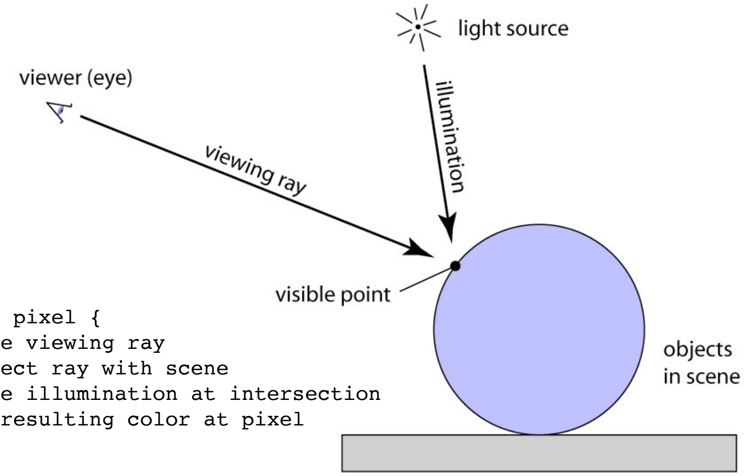
If you prefer to aligned the cameras' coordinate system with xyz,

Use the matrix $M = Rotation_{\vec{u},\vec{v}.\vec{w}} \cdot Trans(-Eye) \cdot p$

$$Rotation_{\vec{u},\vec{v}.\vec{w}} = \begin{bmatrix} - & \vec{u} & - \\ - & \vec{v} & - \\ - & \vec{w} & - \end{bmatrix}$$

https://www.geogebra.org/m/jdqhhwku

Ray Tracing Algorithm

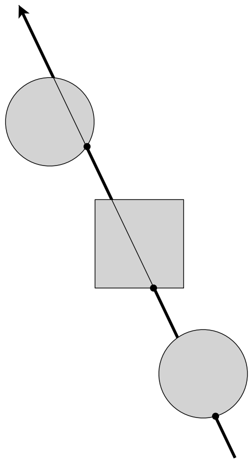


```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Intersecting Objects

```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Intersection with Many Types of Shapes



- In a given scene, we also need to track which shape had the nearest hit point along the ray.
- This is easy to do by augmenting our interface to track a range of possible values for t, $[t_{\min}, t_{\max}]$:

```
intersect(eye, dir, t_min, t_max);
```

- After each intersection, we can then update the range

Intersection with Many Types of Shapes

```
for each pixel p in Image {
  let [eye, dir] = camera.compute_ray(p);
  let hit_surf = undefined;   let hit_rec = undefined;
  let t_min = 0;   let hit_t = Infinity;

  scene.surfaces.forEach( function(surf) {
    let intersect_rec = surf.intersect(eye, dir, t_min, hit_t);
    if (intersect_rec.hit) {
      hit_surf = surf;
      hit_t = intersect_rec.t;
      hit_rec = intersect_rec;
    }
  });

  //Compute a color c
  image.update(p, c);
}
```

```
for each pixel of the output image {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

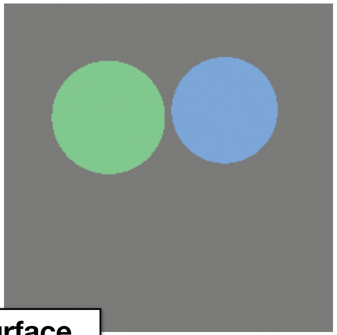
Illumination

```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Our images so far

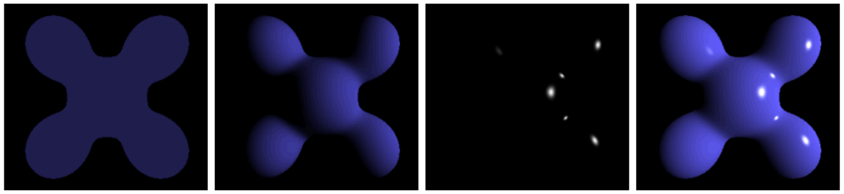
- With only eye-ray generation and scene intersection

```
for each pixel p in Image {
  let hit_surf = undefined;
  ...

  scene.surfaces.forEach( function(surf) {
    if (surf.intersect(eye, dir, ...)) {
      hit_surf = surf;
      ...
    }
  });

  c = hit_surf.ambient;
  Image.update(p, c);
}
```



Each surface storing a single ambient color

Today: shading

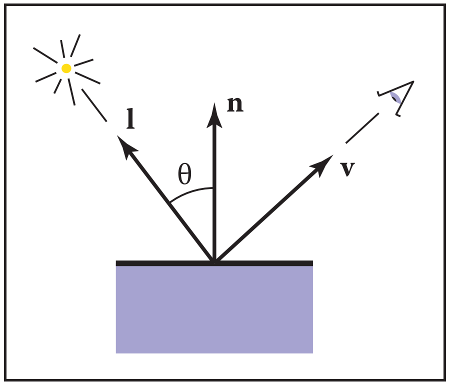


From this (ambient shading) + Diffuse Shading + Specular Shading ⇒ this

https://en.wikipedia.org/wiki/Phong_shading

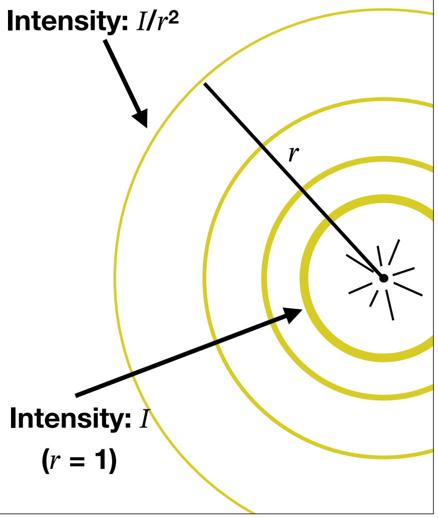
Shading

- Goal: Compute light reflected toward camera
- Inputs:
    - eye direction
    - light direction (for each of many lights)
    - surface normal
    - surface parameters (color, shininess, …)



Light Sources

- There are many types of possible ways to model light, but for now we'll focus on **point lights**
- Point lights are defined by a position $\mathbf{p}$ that irradiates equally in all directions
- Technically, illumination from real point sources falls off relative to distance squared, but **<u>we will ignore this for now.</u>**



Intensity: I/r^2

r

Intensity: I

$(r = 1)$

Shading Models

Just to be sure:

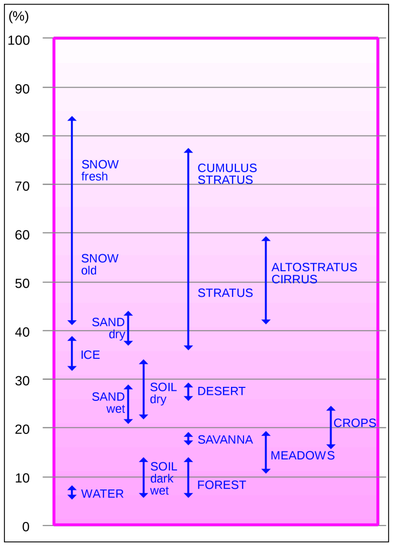
Shading ≠ Shadows

- **Shadows** are casted by occluding sources of light.
- **Shading** of a surface - changing of intensity of the **reflected** light due to surface properties ad geometry, and its locations in 3D with respect to locations of viewer and light source.

We will cover Diffuse shading and Specular Shading. We will study a trick that is easy to program, and ``looks" like physical diffuse shading.

Ambient coefficient ≠ Albedo coefficient

- Albedo coefficient - percentage of white light reflected by the object
- White light -might contains all visible frequencies, not only RGB.
- No attention to color.

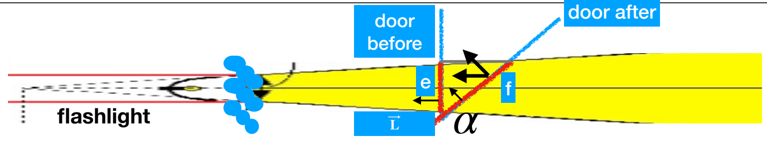


Ambient ``shading” and Albedo

- Ambient light - has no particular direction.
- Every material has 3 coefficients($\mathbf{k_d . r}$, $\mathbf{k_d . g}$, , $\mathbf{k_d . b}$).
- $\mathbf{k_d . b}$ specifies the percentage of **blue** light that the surface reflects (obviously, as blue light).
- The location of viewer and the location of the light-source are irrelevant.
- If a sphere has Ambient coefficient $(k_d . r,\ k_d . g,\ k_d . b) = (0.1,\ 0.9,\ 0.9)$ it looks very dim in Red light, but bright in Blue or Green light.
- If illuminated by while light, then the sphere color is cyan.
- When describing a scene to (say) OpenGL, WebGL, processing.org etc, we could specify for every light source how much intensity it emits (in RGB).
- In reality, there is no ambient light.
- In OpenGL, we could specify 3 sets of coefficients (for ambient, for diffuse, and for specular. We can also specify the scene ambient RGB.
- E.g. specifying the ambient light in the scene as (0.3, 0.1, 0.9), and a sphere with k_d=(0, 0, 0.5), will be seen with $RGB = (0,\ 0,\ 0.45)$

Lambertian (Diffuse) Shading

- Consider a door illuminated by a flashlight (see below).
- Lets think about the intensity reflected from the door as the door rotates.
- I denotes the intensity. Think about I as #photons$/inch^2$
- Let e be a portion of the door with area 1_{in^2}. The number of photons falling on e is I.
- Now open the door (without moving e). Let f be the area of the shadow that e casts on the door. The area of f is $1_{in^2}/\cos\alpha$ (where α is the angle of the door)
- The same amount of photos that are passing via e are falling on a large area

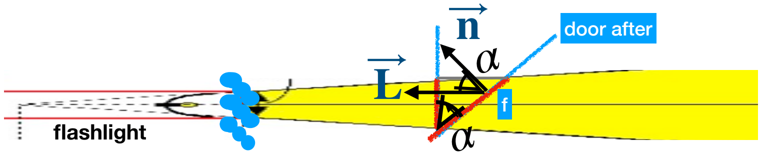


Intensity of the light on f = #photons falling on 1_{in^2}

The number of photons on e and on f is the same, but the area

increases to $1_{in^2}/\cos\alpha$, so intensity now is$I/f = I/\frac{1}{\cos\alpha} = I\cos\alpha$

Lambertian (Diffuse) Shading



Intensity of the light on f = #photons falling on 1_{in^2}

The number of photons on e and on f is the same, but the area

increases to $1_{in^2}/\cos\alpha$, so intensity now is $I/f = I/\frac{1}{\cos\alpha} = I\cos\alpha$

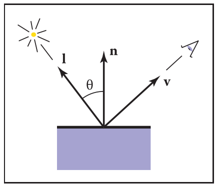
Let $\vec{\mathbf{L}}$ be a unit vector from f toward the light source, and let $\vec{\mathbf{n}}$ be the normal to the door.

$\cos\alpha = \vec{\mathbf{L}} \cdot \vec{\mathbf{n}}$

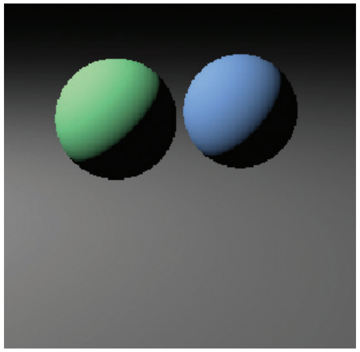
The intensity of light reflected from f is intensity of light hitting f times k_d

Conclusion: To create diffuse shading, render f with RGB= $k_d\ I\ \vec{L} \cdot \vec{n}$

Lambertian (Diffuse) Shading



- Simple model: amount of energy from a light source depends on the direction at which the light ray hits the surface
- Results in shading that is *view independent*



$$L_d = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

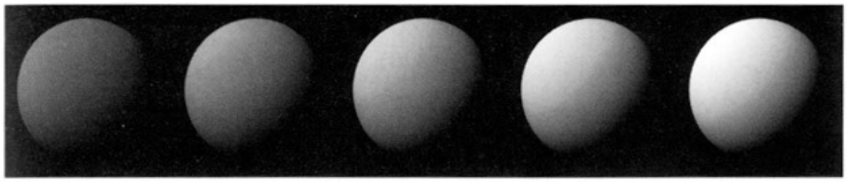
diffuse coefficient

intensity/color of light

cos θ

Lambertian Shading

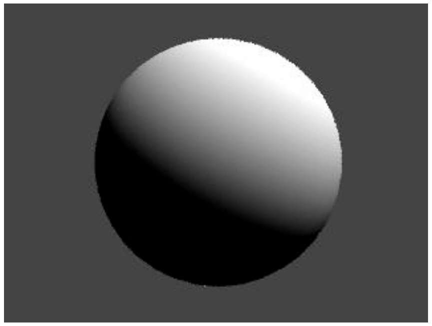
- k_d is a property of the surface itself (3 constants - one per each color channel)
- Produces matte appearance of varying intensities



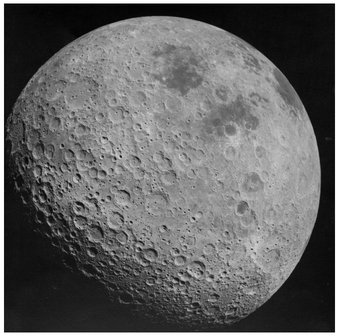
$k_d \longrightarrow$

The moon paradox

- why don't we see this gradual shading when looking at the moon ?

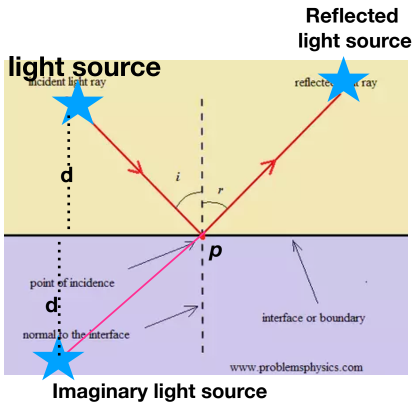


VS



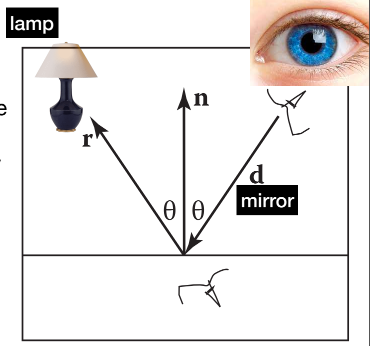
Toward Specular Shading: Perfect Mirror

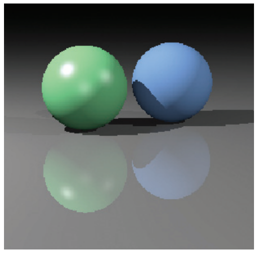
- Many real surfaces show some degree of shininess that produce **specular** reflections
- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading)
- Idea: produce reflection when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal



Mirrors - perfect reflections

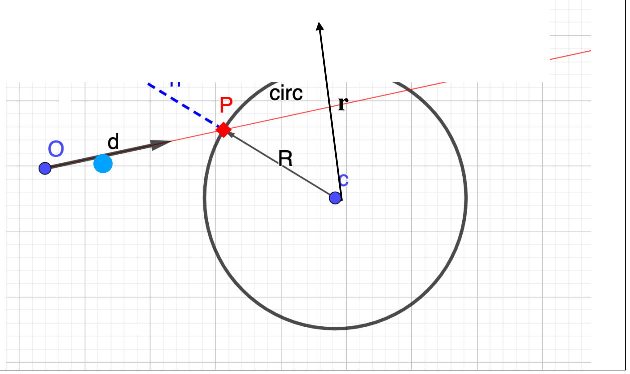
- Before talking about specular reflection, lets see how to render a scene that contains mirror.
- Ray tracing: For each pixel on the image plane, trace a ray d from the eye via this pixel, till hits an object. If this object is a mirror, we need to continue this ray in the deflected direction r.
- How could find find r **?**
- **Claim**: $\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$, **n** is a unit vector orthogonal to the mirror.
- **Proof**
  - Assume wlog that **n=(0,1)** (vertical upward).
  - Look at the components: **d**=(d.x,d.y), **r**=(r.x, r.y)
  - $\mathbf{r}$ and $\mathbf{d}$ have the same x-value, but opposite y-value:
    - r.x=d.x and
    - r.y= -d.y = r.y+ (-2r.y) = r.y -2 (n · r)
  - $(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$=(0, r.y).





Application: mirror sphere

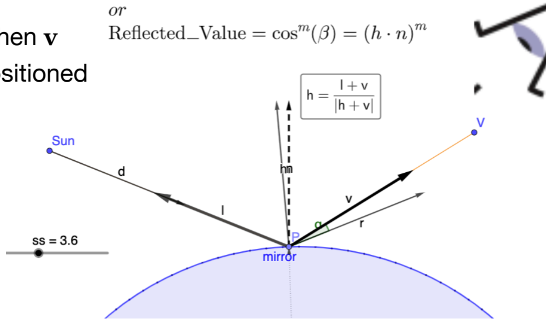
- A ray d that hits the sphere B. We find the intersection point P, find the normal to B at P, $\mathbf{n} = \dfrac{P-c}{|P-c|}$,
- **and bounced in the direction** $\mathbf{r} = \mathbf{d} - 2(\mathbf{n}\cdot\mathbf{d})\mathbf{d}$



Blinn-Phong (Specular) Shading

- Many real surfaces show some degree of shininess that produce specular reflections
- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading)
- Idea: produce reflection when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal

$$\text{Reflected\_Value} = \cos^m(\alpha) = (h\cdot n)^m.$$
or
$$\text{Reflected\_Value} = \cos^m(\beta) = (h\cdot n)^m$$



Blinn-Phong (Specular) Shading

- For any two unit vectors $\vec{v}, \vec{l}$, the vector $\mathbf{v} + \mathbf{l}$ is a bisector of the angle between these vectors.
- Normalize $\mathbf{v} + \mathbf{l}$

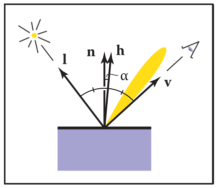
$$\mathbf{h} = (\mathbf{v} + \mathbf{l}) \,/\, \|\mathbf{v} + \mathbf{l}\|$$

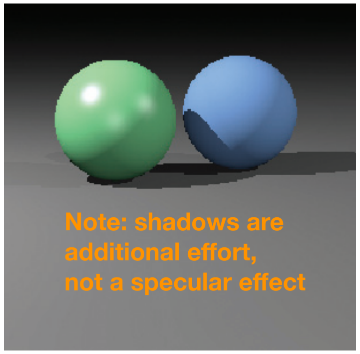
- In a perfect mirror, the 100% of the reflection occurs at the surface point where h is the normal n
- Diffuse reflection. Reflect large value for points where h is ``almost” n
- Phong heuristic:

$$L_s = k_s I \max(0, \quad (\mathbf{n}\cdot\mathbf{h})^{p.} \;)$$

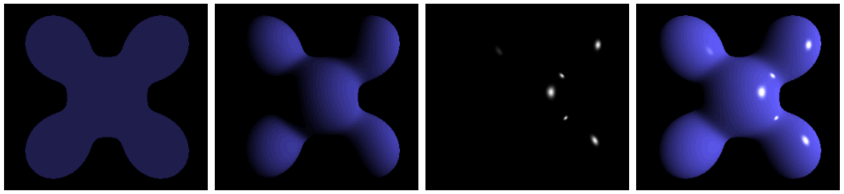
specular coefficient

Phong exponent




Note: shadows are additional effort, not a specular effect

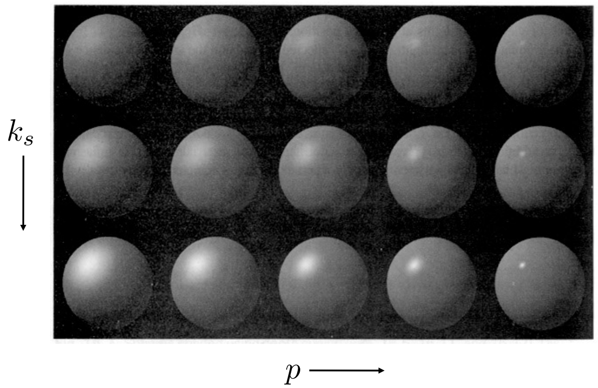
Blinn-Phong Decomposed


Ambient + **Diffuse** + **Specular** = **Phong Reflection**

https://en.wikipedia.org/wiki/Phong_shading

Blinn-Phong Shading

- Increasing p narrows the lobe
- This is kind of a hack, but it does look good



[Foley et al.]

Putting it all together

- Usually include ambient, diffuse, and specular in one model

$$L = L_a + L_d + L_s$$

$$L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

- And, the final result accumulates for all lights in the scene

$$L = k_a I_a + \Sigma_i \left(k_d I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p \right)$$

- Be careful of overflowing! You may need to clamp colors, especially if there are many lights.

Simple Ray Tracer

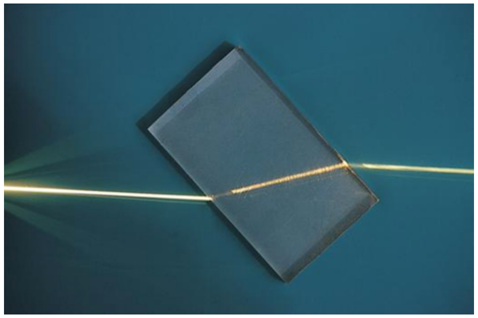
```
function ray_cast(eye, dir, near, far) {
  let hit_surf = undefined;   let hit_rec = undefined;
  let t_min = 0;   let hit_t = Infinity;
  let color = background;    //default background color

  scene.surfaces.forEach( function(surf) {
    let intersect_rec = surf.hit(eye, dir, t_min, hit_t);
    if (intersect_rec.hit) {
      hit_surf = surf;
      hit_t = intersect_rec.t;
      hit_rec = intersect_rec;
    }
  });

  if (hit_surf !== undefined) {
    color = hit_surf.kA * Ia;
    scene.lights.forEach( function(light) {
      //compute li, hi
      color = color + hit_surf.kD*Ii*max(0,n · li) + hit_surf.kS*Ii*max(0,n · hi)p;
    });
  }

  return color;
}
```

```
for each pixel p in Image {
  let [eye, dir] = camera.compute_ray(p);
  let c = ray_cast(eye, dir, 0, Infinity);
  image.update(p, c);
}
```

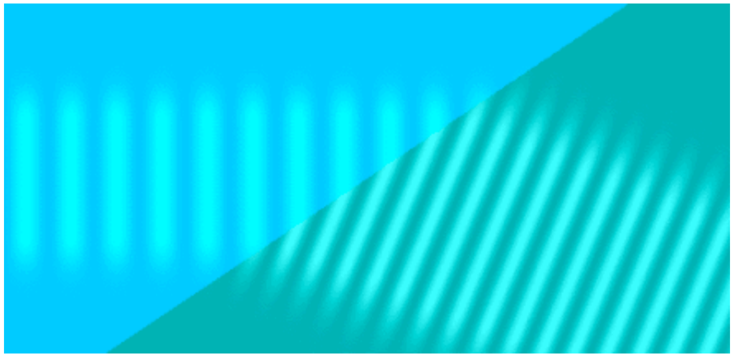
Refraction and Snell Law

- When light passes from one **medium** to another, (say air→ glass or glass →air, its direction might change.
- This happens when the speed of light in the two mediums are different

Credit: Wikipedia



Following the wavefront



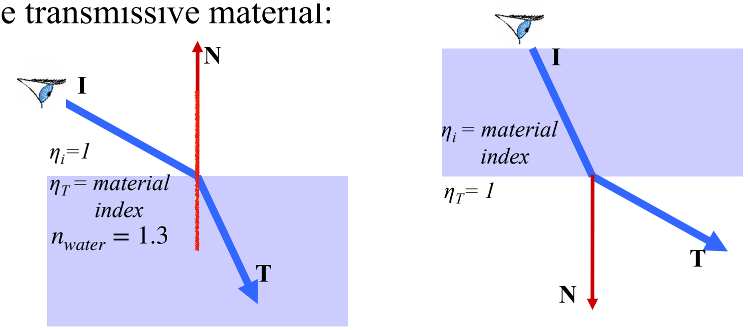
- Credit: Wikipedia

For the wavefronts to stay connected at the boundary the wave must change direction.

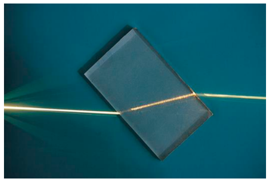
Refraction and Snell Law

- When ray of light traverses from one medium (e.g. from air to water) it might bend. This is called **refraction.**

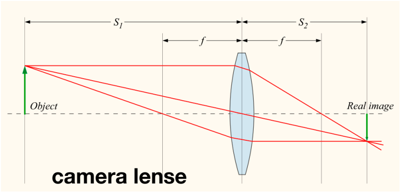
the transmissive material:



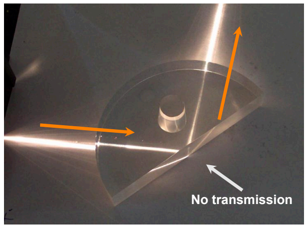
Examples of refractions



credit: wikipedia



camera lense



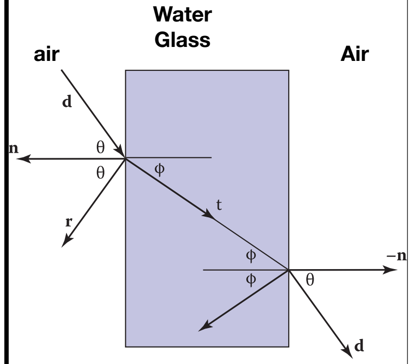
Fiber optics

Refraction and Snell's Law

- Governs the angle at which a refracted ray bends when traversing from air to glass, water etc.
- Computation based on ***refraction index*** (*confusingly denoted* n_t) of the mediums. The mediums here are air and glass.
- Typical air has refraction indexed

n_{air} =	1
n_{glass} =	1.5
n_{water} =	1.3
$n_{fiber\ optics}$ =	1.46

- Snell law: $n_t \sin\theta = n \sin\phi$



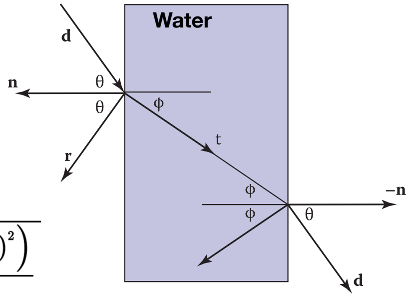
$$\phi = \arcsin(\frac{1}{1.3}\sin\theta)$$

Snell’s Law and vector calculus

- Working with cosine’s are easier because we can use dot products
- Can derive the vector for the refraction direction **t** as

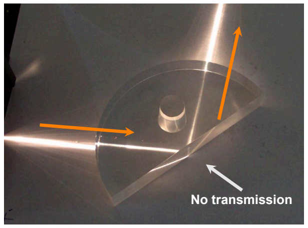
$$\mathbf{t} = \frac{n\left(\mathbf{d}+\mathbf{n}\cos\theta)\right)}{n_t} - \mathbf{n}\cos\phi$$

- $$= \frac{n\left(\mathbf{d}-\mathbf{n}\left(\mathbf{d}\cdot\mathbf{n}\right)\right)}{n_t} - \mathbf{n}\sqrt{1 - \frac{n^2\left(1-\left(\mathbf{d}\cdot\mathbf{n}\right)^2\right)}{n_t^2}}$$



Careful:
don’t confuse **n** (a normal vector) with (1.3n_t for water) and with n (=1 for air)

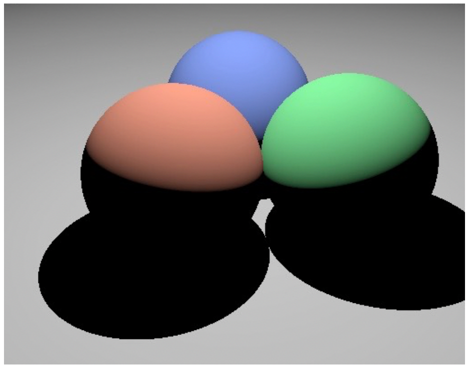
Total Internal Reflection



Recursive Ray Tracing

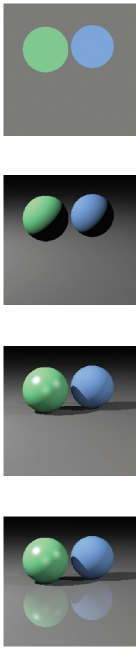
Shadows

- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting $t_{\min}$



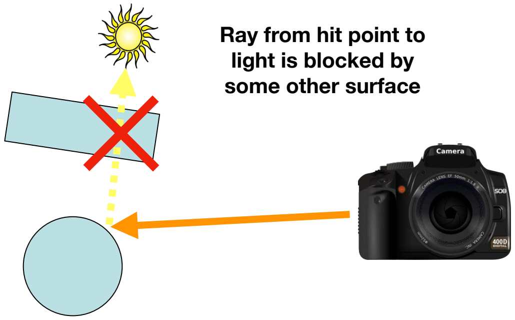
Recursive Ray Tracer

```
Color ray_cast(Ray ray, SurfaceList scene, float near, float far) {
  ...
  //initialize color;  compute hit_surf, hit_position;
  ...

  if (hit_surf is valid) {
    color = hit_surf.kA * Ia;

  }

  return color;
}
```



Shadows

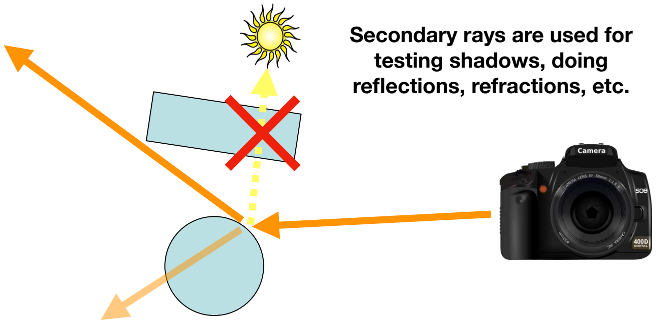
- Surface should only be illuminated if nothing blocks the light from hitting the surface
- This can be easily checked by intersecting a new ray with the scene!



Ray from hit point to light is blocked by some other surface

Ray Casting vs Ray Tracing

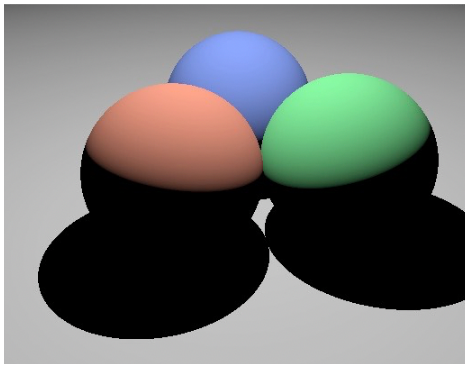
- Ray casting: tracing rays from eyes only
- Ray tracing: tracing secondary rays



Secondary rays are used for testing shadows, doing reflections, refractions, etc.

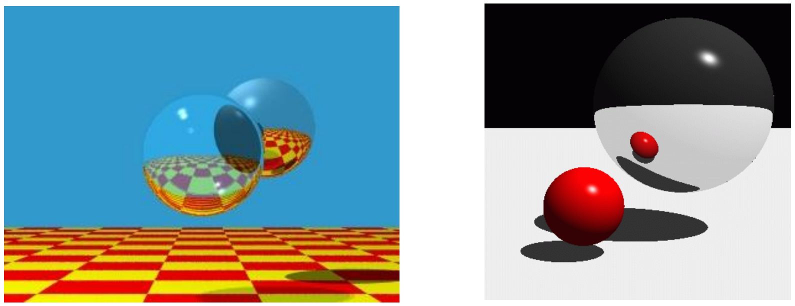
(hard) Shadows

- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting t_{min}

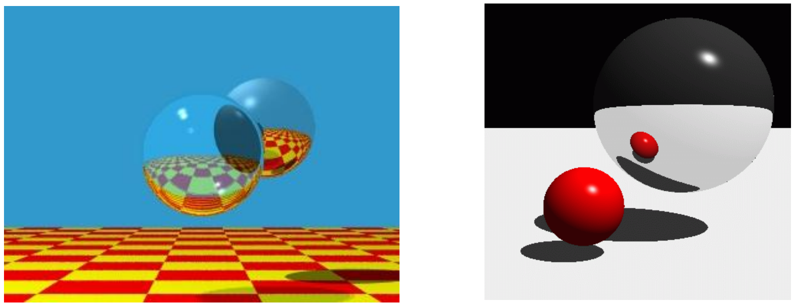


Distribution Ray Tracing

Reality Check: Do These Pictures Look Real?



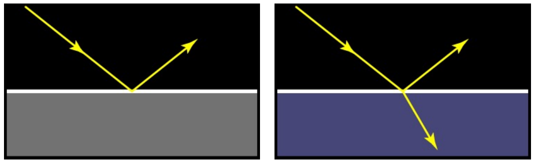
What's Wrong?



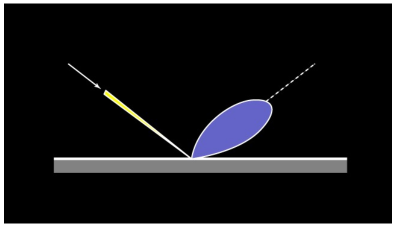
- No surface is a perfect mirror because no surface is perfectly smooth

---

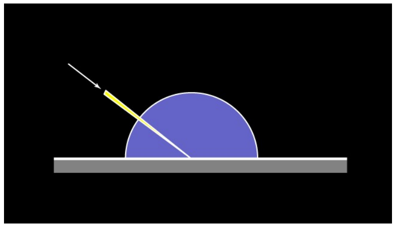
What have we modeled?



ideal specular (mirror)

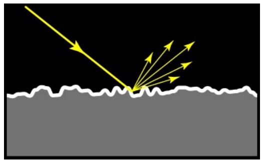


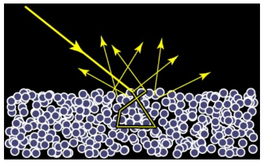
glossy specular



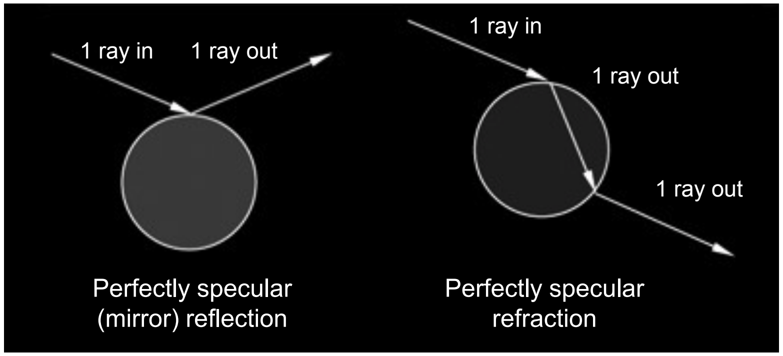
Lambertian

Most Surfaces have Microgeometry





Ideal Reflection/Refraction



1 ray in

1 ray out

Perfectly specular (mirror) reflection

1 ray in

1 ray out

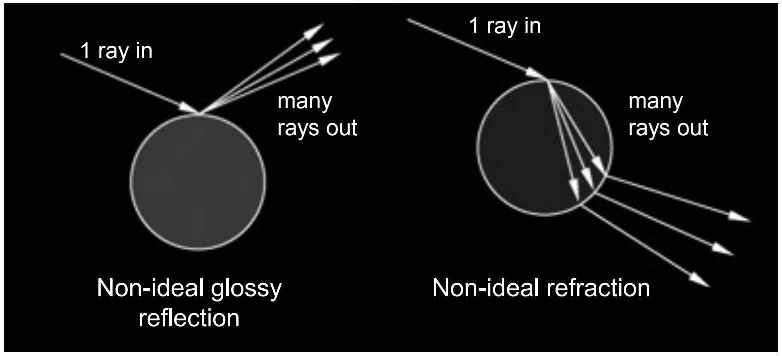
1 ray out

Perfectly specular refraction

Adapted from blender.org

Non-Ideal Reflection/Refraction

- Can approximate the microgeometry



1 ray in

many rays out

Non-ideal glossy reflection

1 ray in

many rays out

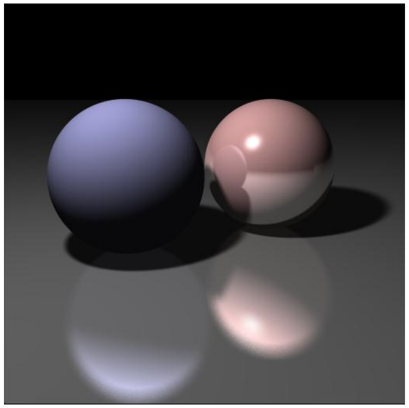
Non-ideal refraction

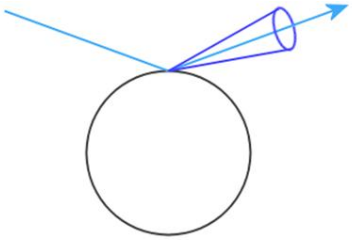
Adapted from blender.org

Glossy (as opposed to mirror) reflection

Glossy (as opposed to perfect) refraction

Approach: Distribution Glossy Reflection by Randomly Sampling Rays

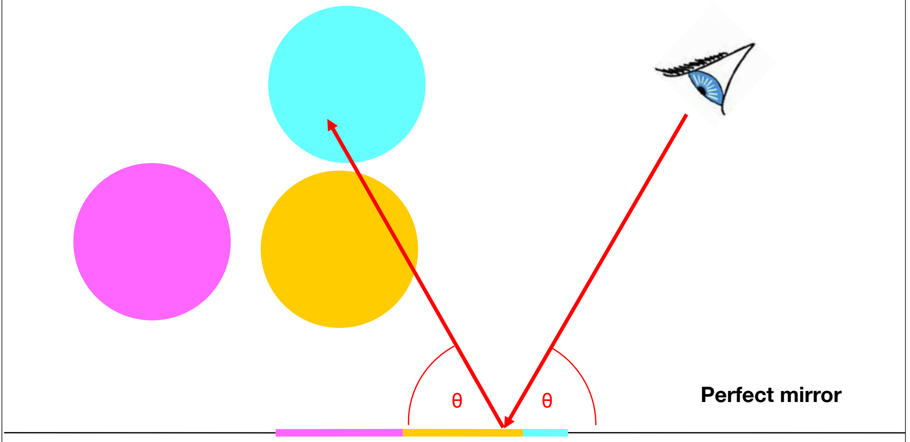




https://graphics.stanford.edu/wikis/cs148-11-fall/RaytracingResults
http://www.baylee-online.net/Projects/Raytracing/Algorithms/Glossy-Reflection-Transmission

---

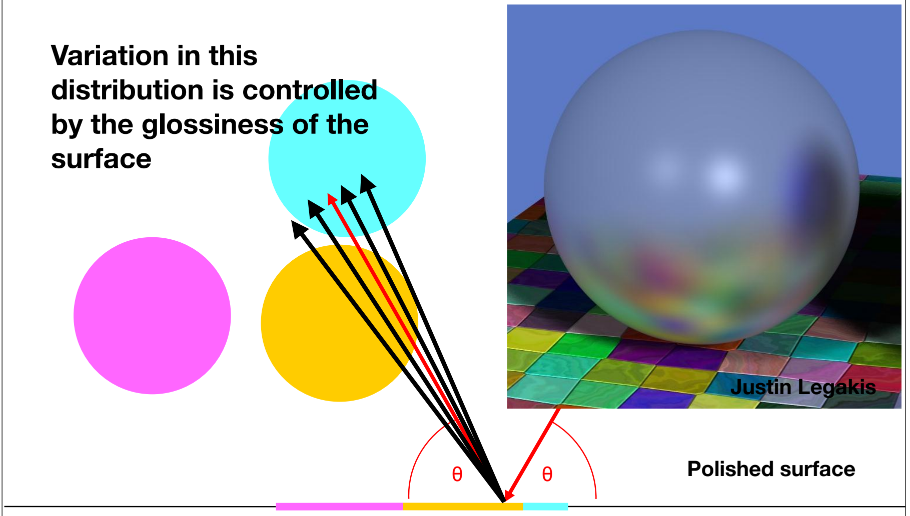
Ideal Reflection: One Ray Per Bounce



Perfect mirror

Glossy Reflection: Compute Many Rays per Bounce and Average

Variation in this distribution is controlled by the glossiness of the surface

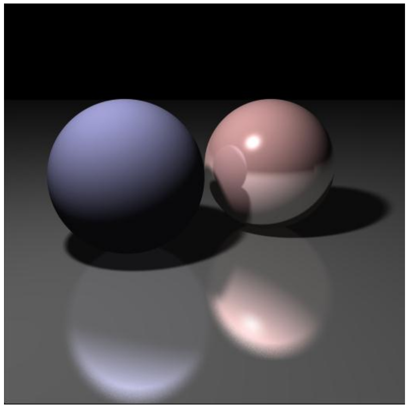


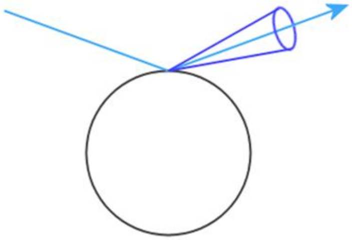
Justin Legakis

Polished surface

Other Uses of Distribution Ray Tracing

Approach: Distribution Glossy Reflection by Randomly Sampling Rays

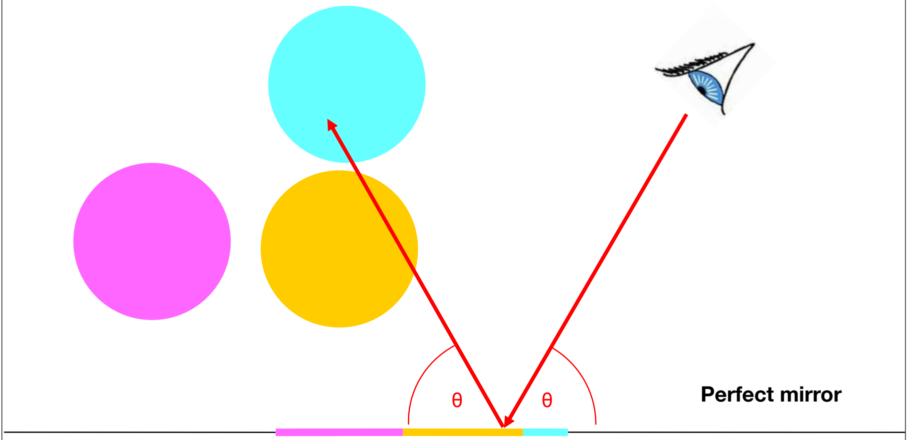




https://graphics.stanford.edu/wikis/cs148-11-fall/RaytracingResults
http://www.baylee-online.net/Projects/Raytracing/Algorithms/Glossy-Reflection-Transmission

---

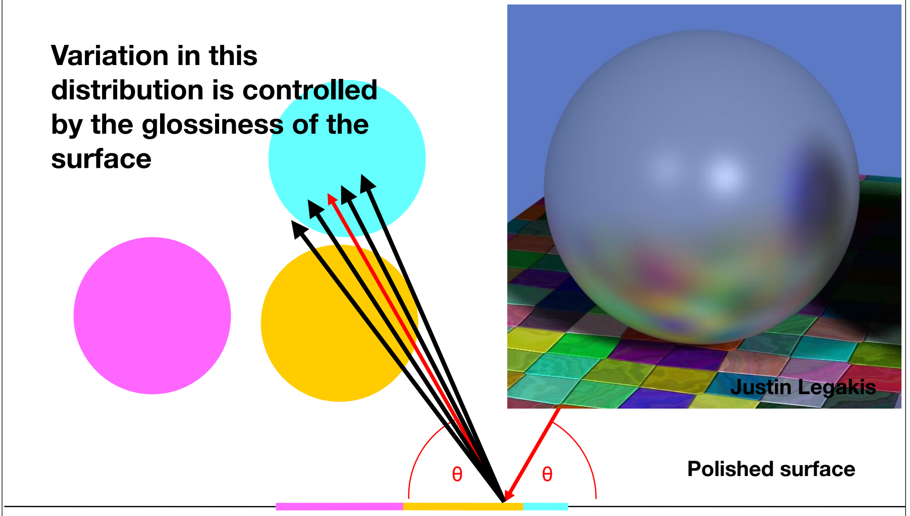
Ideal Reflection: One Ray Per Bounce



Perfect mirror

Glossy Reflection: Compute Many Rays per Bounce and Average

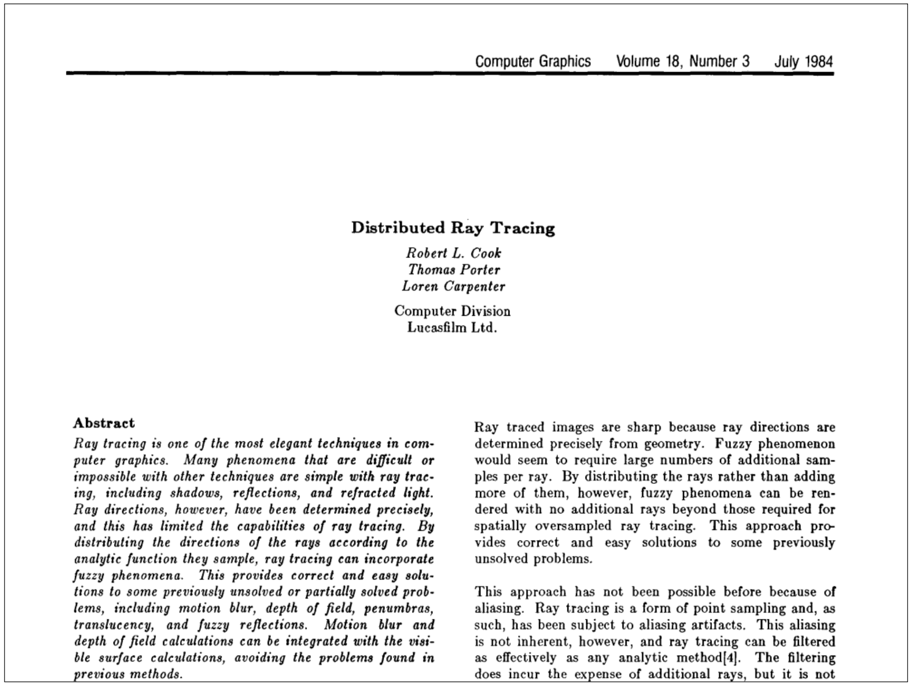
Variation in this distribution is controlled by the glossiness of the surface



Justin Legakis

Polished surface

Other Uses of Distribution Ray Tracing



Computer Graphics Volume 18, Number 3 July 1984

Distributed Ray Tracing

Robert L. Cook
Thomas Porter
Loren Carpenter

Computer Division
Lucasfilm Ltd.

Abstract

Ray tracing is one of the most elegant techniques in computer graphics. Many phenomena that are difficult or impossible with other techniques are simple with ray tracing, including shadows, reflections, and refracted light. Ray directions, however, have been determined precisely, and this has limited the capabilities of ray tracing. By distributing the directions of the rays according to the analytic function they sample, ray tracing can incorporate fuzzy phenomena. This provides correct and easy solutions to some previously unsolved or partially solved problems, including motion blur, depth of field, penumbras, translucency, and fuzzy reflections. Motion blur and depth of field calculations can be integrated with the visible surface calculations, avoiding the problems found in previous methods.

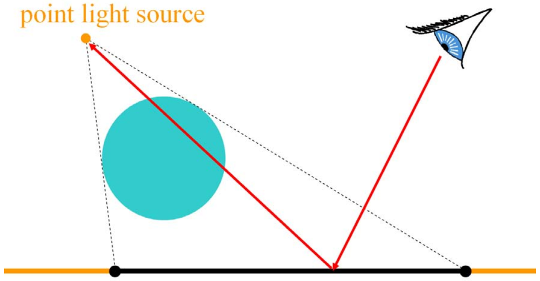
Ray traced images are sharp because ray directions are determined precisely from geometry. Fuzzy phenomenon would seem to require large numbers of additional samples per ray. By distributing the rays rather than adding more of them, however, fuzzy phenomena can be rendered with no additional rays beyond those required for spatially oversampled ray tracing. This approach provides correct and easy solutions to some previously unsolved problems.

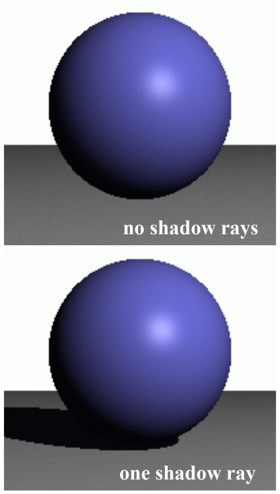
This approach has not been possible before because of aliasing. Ray tracing is a form of point sampling and, as such, has been subject to aliasing artifacts. This aliasing is not inherent, however, and ray tracing can be filtered as effectively as any analytic method[4]. The filtering does incur the expense of additional rays, but it is not

---

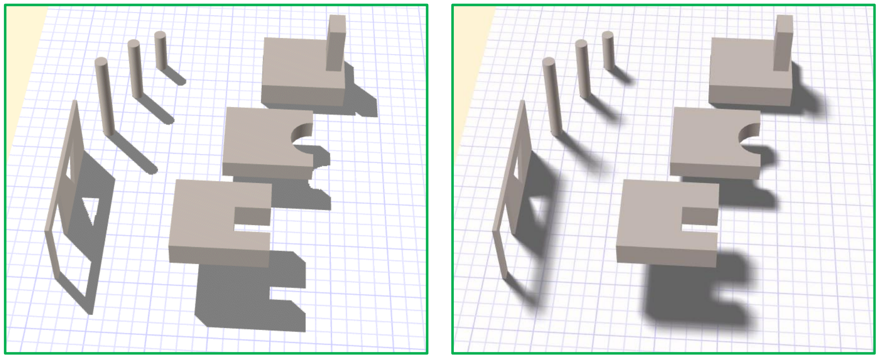
Problem: Hard Shadows

- One shadow ray per intersection per point light source





Soft Shadows



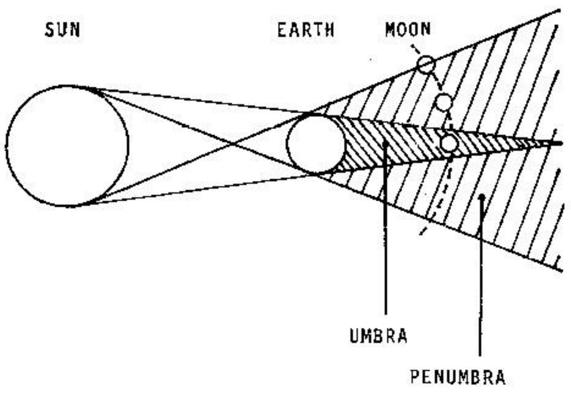
Hard shadows

Soft shadows

http://erich.realtimerendering.com/shadow_comparison.html

Soft Shadows

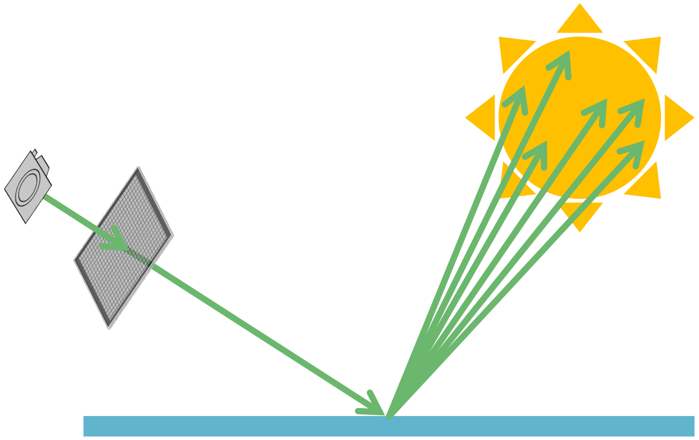
What Causes Soft Shadows



http://user.online.be/felixverbelen/lunecl.jpg

Lights aren't all point sources

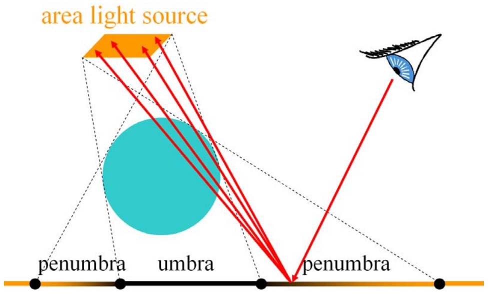
Distribution Soft Shadows

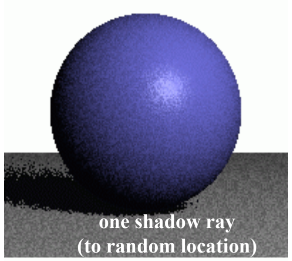


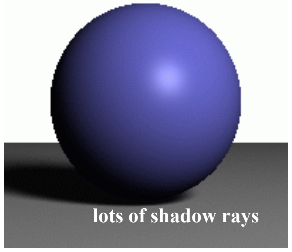
Randomly sample light rays

Computing Soft Shadows

- Model light sources as spanning an area
- Sample random positions on area light source and average rays







Problem: Aliasing
Drawing a black line on a white board

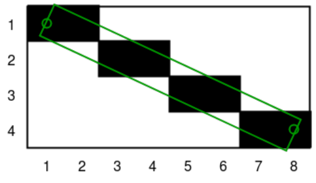
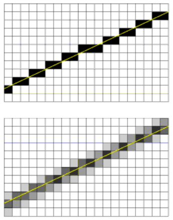


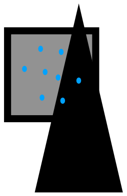
Fig. B: y=f(x) approximation



Some pixels need to be rendered as gray, with gray level=

$$\frac{\textbf{Area of black region in pixel}}{\textbf{Area of pixel}}$$

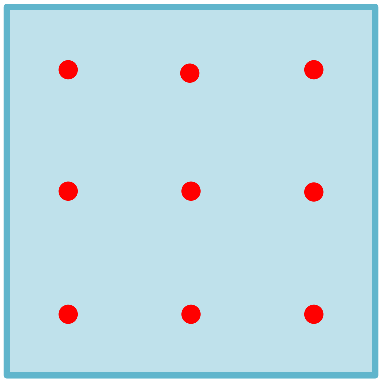
Pixel:



- Problem: Hard to calculate how much of the pixel is covered
- Solution: Random sample points in the pixel.
- Calculate what is the percentage of the point of each color

---

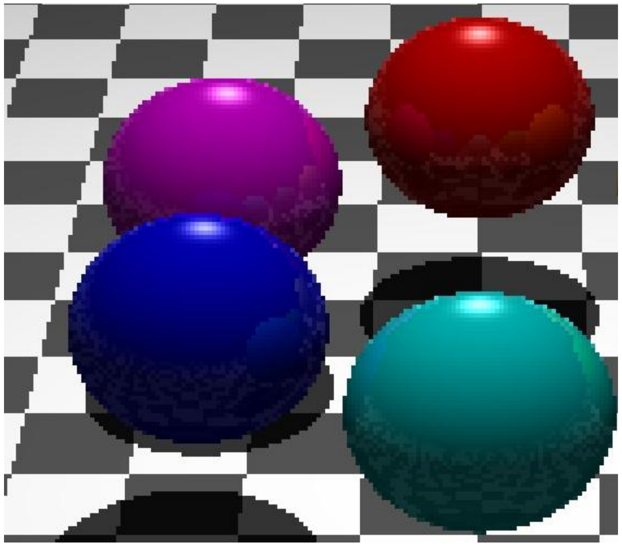
Distribution Antialiasing w/ Regular Sampling



http://upload.wikimedia.org/wikipedia/commons/f/fb/Moire_pattern_of_bricks_small.jpg

Multiple rays per pixel

Problem: Aliasing

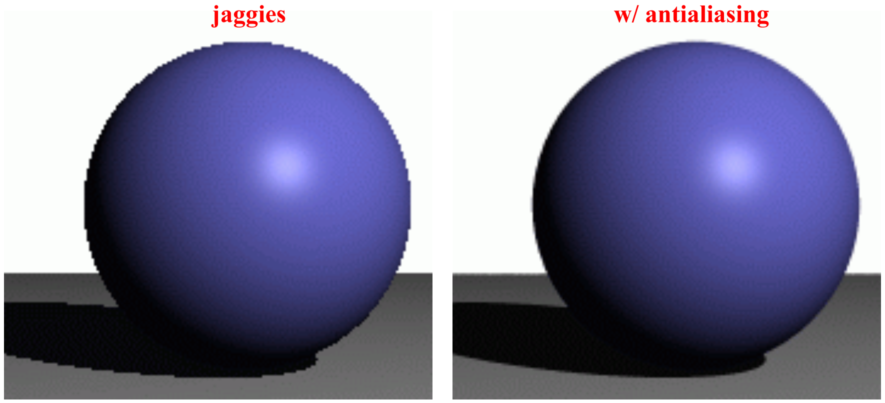


http://www.hackification.com/2008/08/31/experiments-in-ray-tracing-part-8-anti-alias

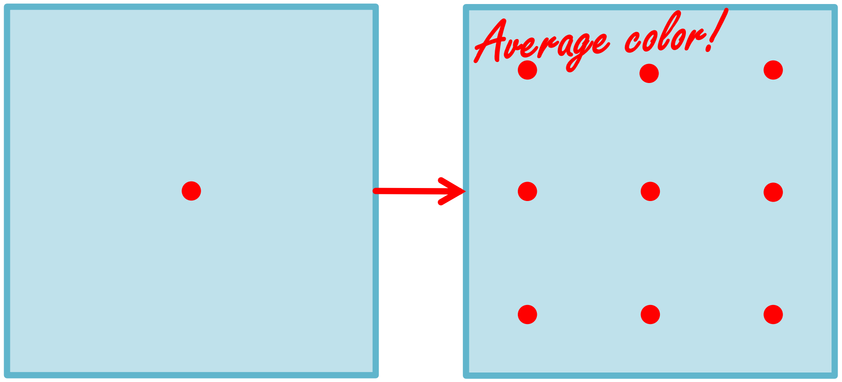
---

Antialiasing w/ Supersampling

- Cast multiple rays per pixel, average result

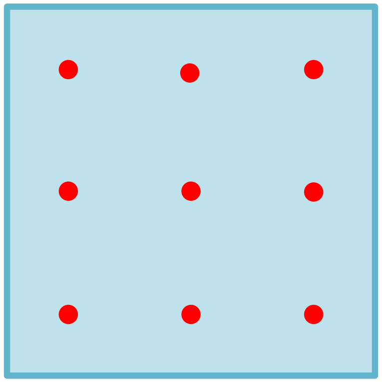


Distribution Antialiasing



Multiple rays per pixel

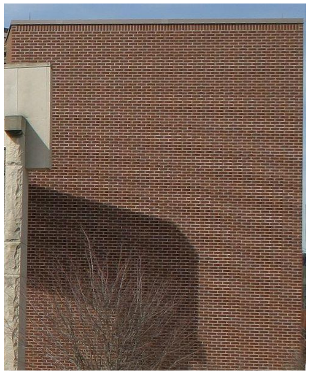
Distribution Antialiasing w/ Regular Sampling

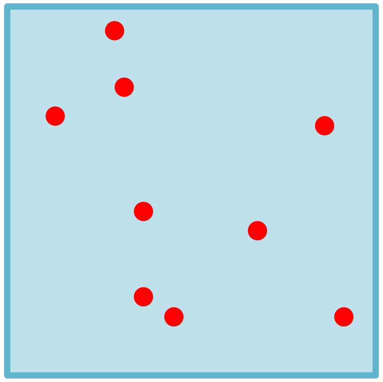


http://upload.wikimedia.org/wikipedia/commons/f/fb/Moire_pattern_of_bricks_small.jpg

Multiple rays per pixel

Distribution Antialiasing w/ Random Sampling

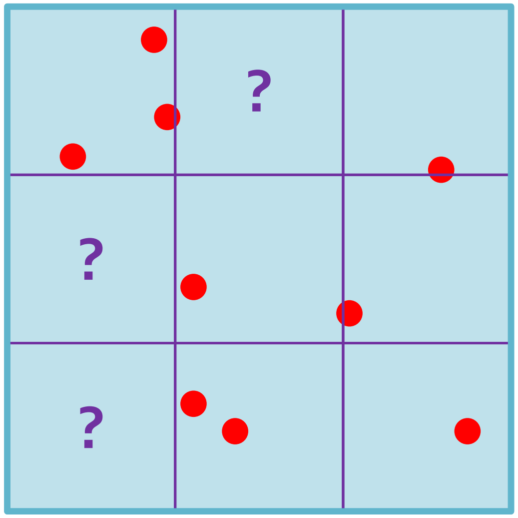




http://en.wikipedia.org/wiki/File:Moire_pattern_of_bricks.jpg

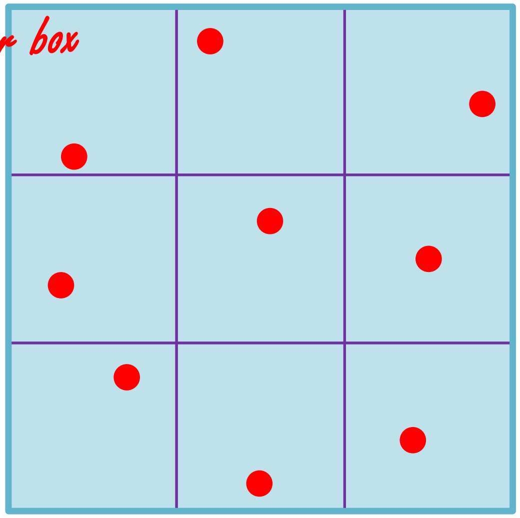
Remove Moiré patterns

Random Sampling Could Miss Regions Without Enough Sampling



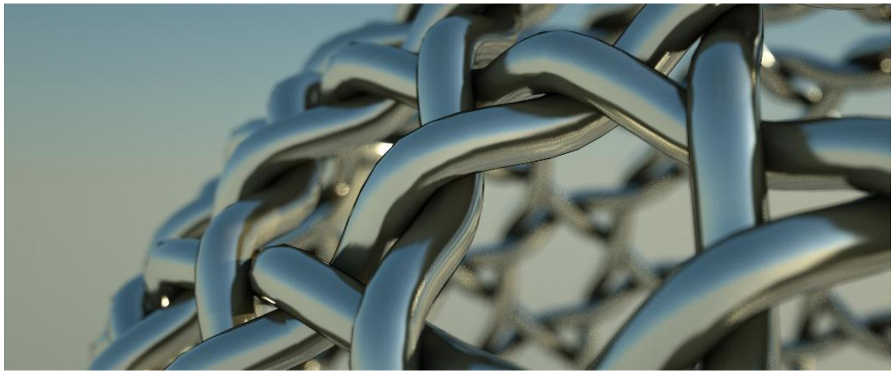
Stratified (Jittered) Sampling

One ray per box



Problem: Focus
Real Lenses Have Depth of Field

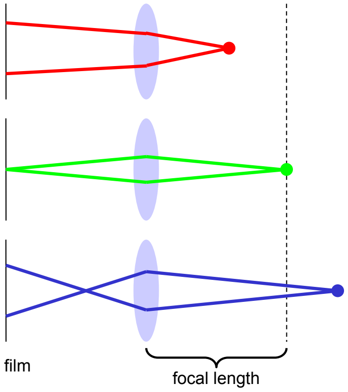
Problem: Focus
Real Lenses Have Depth of Field

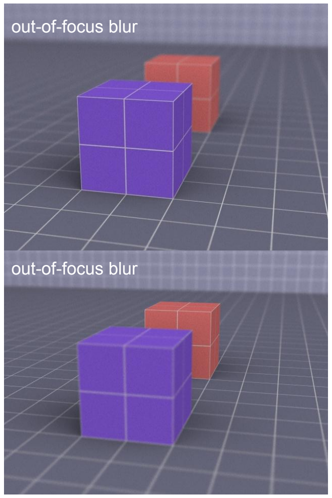


http://liam887.files.wordpress.com/2010/08/weaver.jpg

Depth of Field

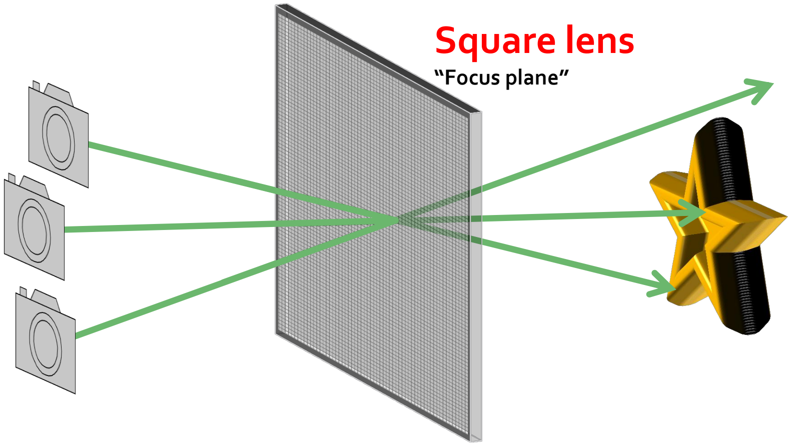
- Multiple rays per pixel, sample lens aperture





Justin Legakis

Distribution Depth of Field



Square lens

"Focus plane"

Randomly sample eye positions

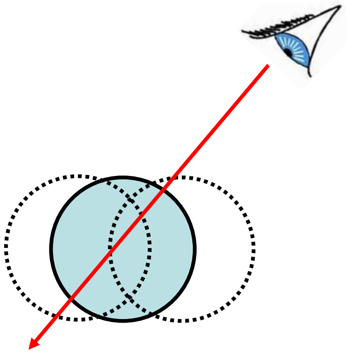
Problem: Exposure Time
Real Sensors Take Time to Acquire

Problem: Exposure Time
Real Sensors Take Time to Acquire

http://www.matkovic.com/anto/3dl-test-balls-01.jpg

Motion Blur

- Sample objects temporally over a time interval



Rob Cook