Ray tracing and extensions

Informàtica gràfica
Curs 2018-19

Gonzalo Besuievsky
IMAE - UdG

Classic Ray Tracing

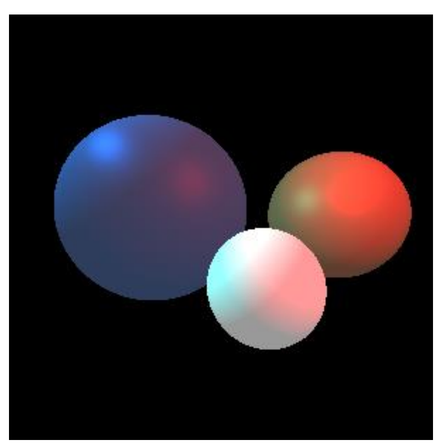
- Introduced in 1980 by Turner Whitted

Contents

- Ray casting review
- Recursive ray tracing
- Acceleration Techniques
- Extended ray tracing techniques

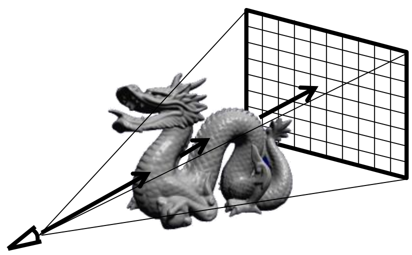
Ray casting

- Extend the visibility algorithm
- Local illumination
- Image quality similar to rendering pipeline, but:
  - Higher quality shapes
  - Higher quality specular illumination
  - Slower



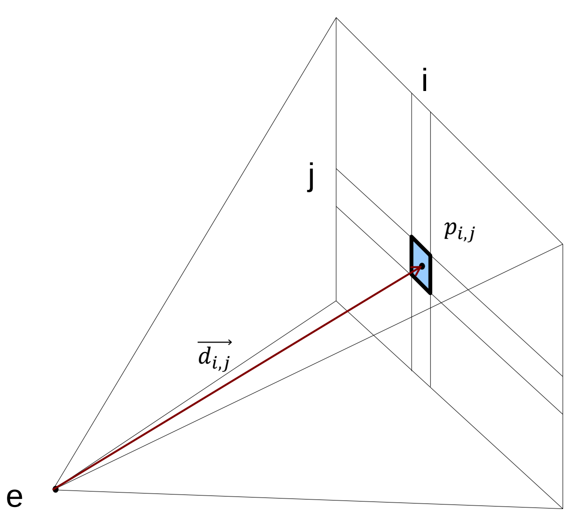
Pixel oriented

- Intersection pixel-model



- For a scene with N primitives:
  - Each pixel → 1 ray → N intersection
  - We get the closest one
  - Then compute illumination

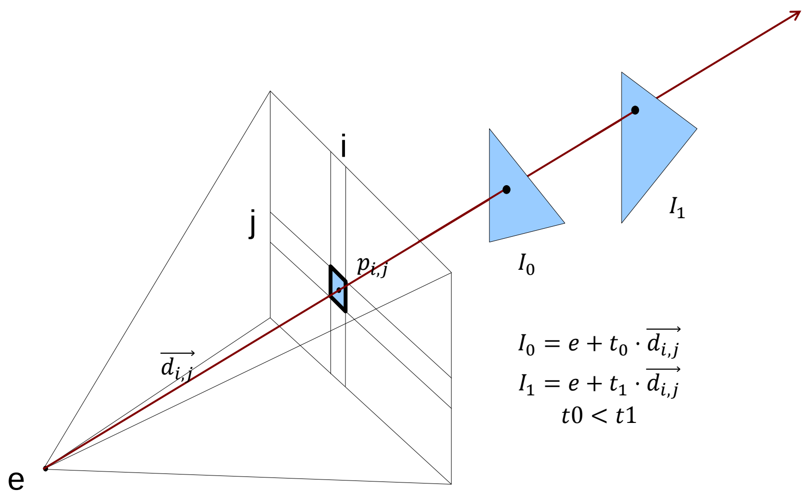
Ray representation



$$p = e + t \cdot \overrightarrow{d_{i,j}}$$

$$\overrightarrow{d_{i,j}} = \frac{p_{i,j} - e}{\| p_{i,j} - e \|}$$

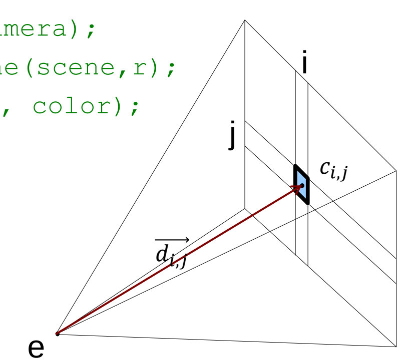
Intersections



$$I_0 = e + t_0 \cdot \overrightarrow{d_{i,j}}$$
$$I_1 = e + t_1 \cdot \overrightarrow{d_{i,j}}$$
$$t0 < t1$$

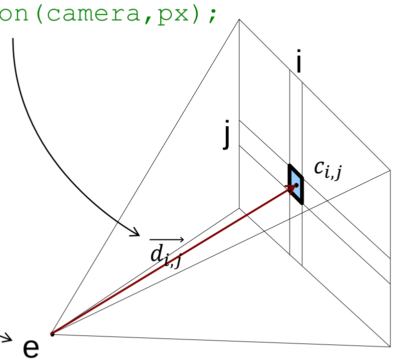
Algorithm

```
action RayCasting(scene, camera)
  for each Pixel px in camera do
     r=defineRay(e,px,camera);
     color=intersectScene(scene,r);
     setPixel(px.i, px.j, color);
  end for
end action
```



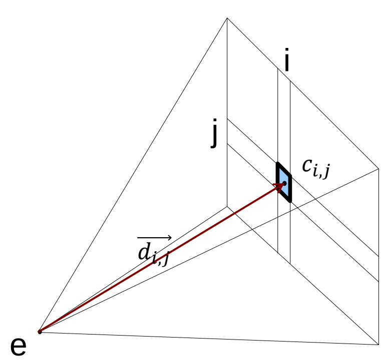
Define ray

```
function defineRay(e,px,camera):Ray
  var r:Ray;
  r.o=e;
  r.d=computeRayDirection(camera,px);
  return r;
end function
```



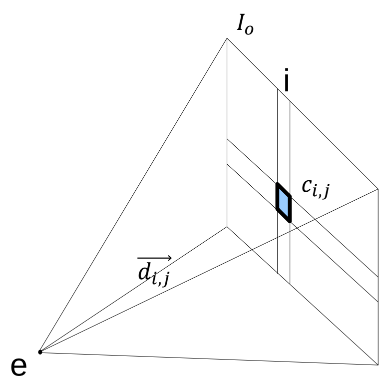
Compute pixel view direction

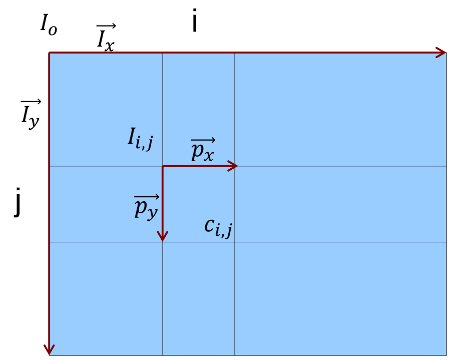
```
function computeViewDirection(camera, px): vector
```



?

Compute pixel view direction

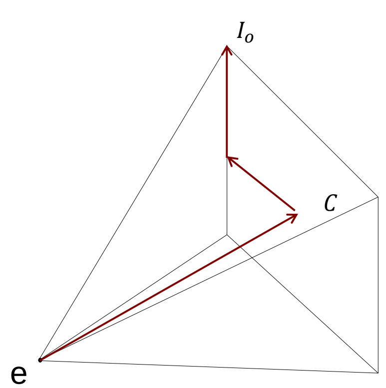




$$I_{i,j} = I_0 + i \cdot \overrightarrow{p_x} + j \cdot \overrightarrow{p_y}$$

$$c_{i,j} = I_{i,j} + 0.5 \cdot \overrightarrow{p_x} + 0.5 \cdot \overrightarrow{p_y} \Rightarrow I_0 + (i + 0.5) \cdot \overrightarrow{p_x} + (j + 0.5) \cdot \overrightarrow{p_y}$$

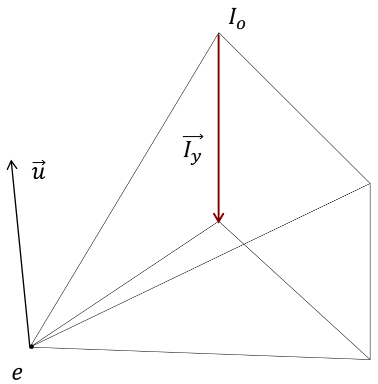
Compute pixel view direction

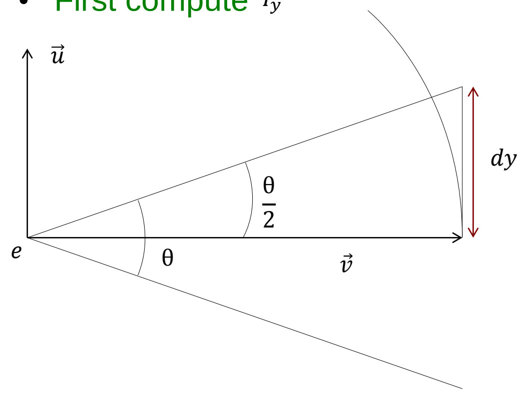


- How do we compute I_o ?
- Camera data:
  - View point: e
  - View direction: $\vec{v}, \| v \| = 1$
  - Up vector: $\vec{u}, \| u \| = 1$
  - Right vector: $\vec{r}, \| r \| = 1$
  - Field of view, aspect ratio

Compute pixel view direction

- First compute $\overrightarrow{I_y}$

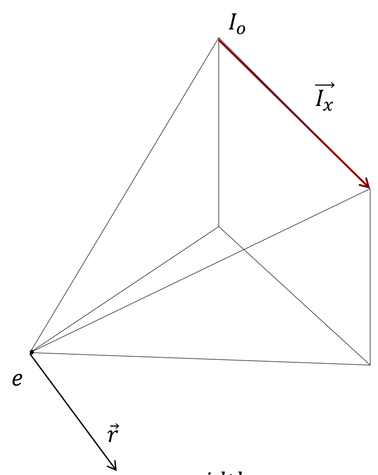


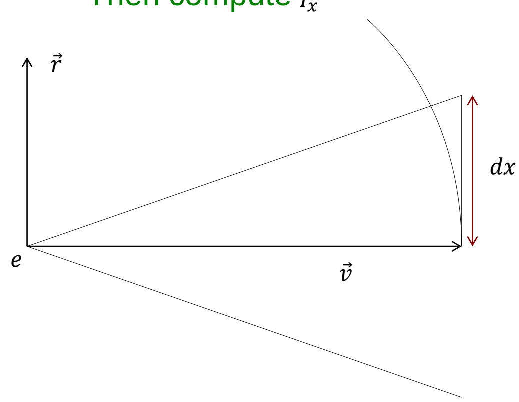


$$dy = \tan\left(\frac{\theta}{2}\right) \Rightarrow \overrightarrow{I_y} = -2 \cdot \tan\left(\frac{\theta}{2}\right) \cdot \vec{u}$$

Compute pixel view direction

- Then compute $\overrightarrow{I_x}$

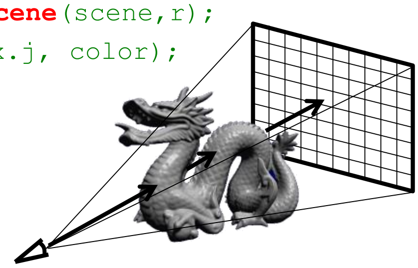




$$ar = \frac{width}{height} \Rightarrow dx = ar \cdot dy = ar \cdot \tan\left(\frac{\theta}{2}\right) \Rightarrow \overrightarrow{I_x} = 2 \cdot ar \cdot \tan\left(\frac{\theta}{2}\right) \cdot \vec{r}$$

Scene intersection

```
action RayCasting(scene, camera)
  for each Pixel px in camera do
    r=defineRay(e,px,camera);
    color=intersectScene(scene,r);
    setPixel(px.i, px.j, color);
  end for
end action
```



Algorithm

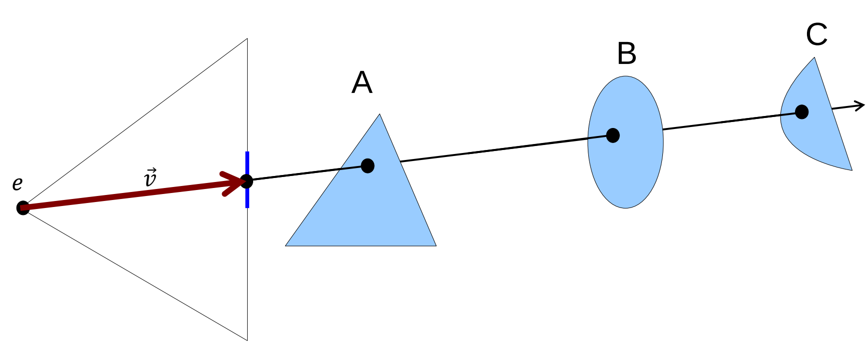
```
function intersectScene(scene,r): Color
  hit=computeFirstHit(scene,r);
  if interaction(hit)
    return computeColor(scene, hit);
  end if
  return BACKGROUND_COLOR;
end function
```

hit stores all the information about the intersection: point, normal, surface id

Algorithm

```
function computeFirstHit(scene,r): Hit
  Hit h;
  for each Primitive p in scene
   Hit h2 = p.intersect(r);
      if h2.t < h.t
          H = h2
      end if
  end for
  return h
end function
```

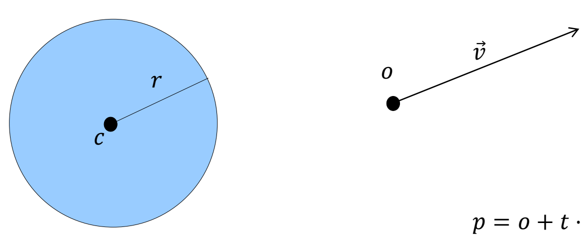
Example



Types of primitives

- Any that can be intersected with a ray:
  - Any polygon
  - Cone
  - Sphere
  - Cilindre
  - Splines
  - NURBS
  - Subdivision surfaces
  - ...

Ray-Sphere intersection



$$\| p - c \| = r$$

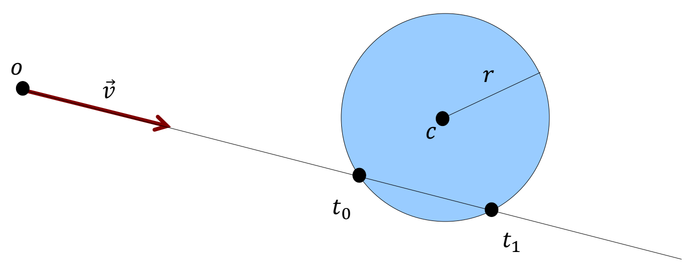
$$p = o + t \cdot \vec{v}$$
$$\| \vec{v} \| = 1$$

$$\| o + t \cdot \vec{v} - c \| = r \Rightarrow \| o + t \cdot \vec{v} - c \|^2 = r^2$$
$$\Rightarrow (o_x + t \cdot v_x - c_x)^2 + \left(o_y + t \cdot v_y - c_y\right)^2 + (o_z + t \cdot v_z - c_z)^2 = r^2$$
$$\Rightarrow t^2 \cdot \left(v_x^2 + v_y^2 + v_z^2\right)$$
$$+t \cdot \left(2 \cdot (o_x - c_x) \cdot v_x + 2 \cdot \left(o_y - c_y\right) \cdot v_y + 2 \cdot (o_z - c_z) \cdot v_z\right)$$
$$+(o_x - c_x)^2 + \left(o_y - c_y\right)^2 + (o_z - c_z)^2$$

Ray-Sphere intersection

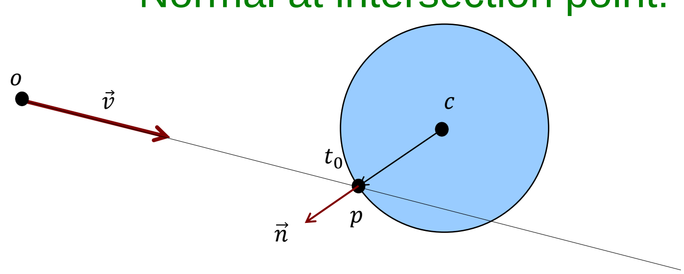
- Second degree equation:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \Rightarrow \begin{cases} a = 1 \\ b = 2(o-c)\cdot\vec{v} \\ c = (o-c)\cdot(o-c) - r^2 \end{cases}$$



Ray-Sphere intersection

- Normal at intersection point:



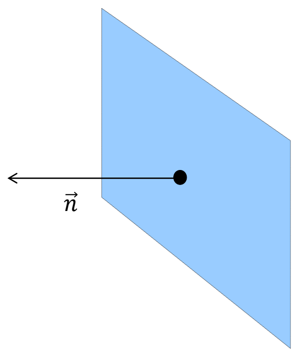
$$p = o + t_0 \cdot \vec{v}$$
$$\vec{n} = \frac{p-c}{\| p-c \|} \Rightarrow \frac{p-c}{r}$$

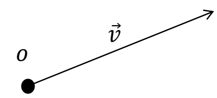
Ray-plane intersection

$$Ax + By + Cz + D = 0$$
$$\vec{n} = (A, B, C)$$

$$\vec{n} \cdot P + D = 0$$

Però també:
$$\vec{n} \cdot (P - P0) = 0$$
On P0 és un punt del pla



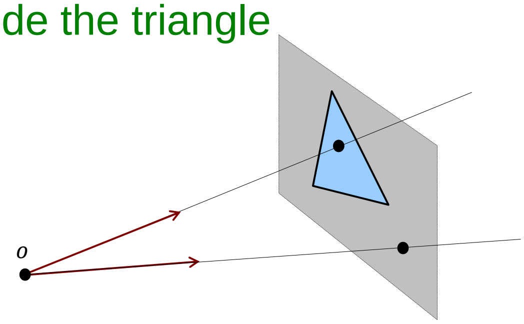


$$p = o + t \cdot \vec{v}$$
$$\| \vec{v} \| = 1$$

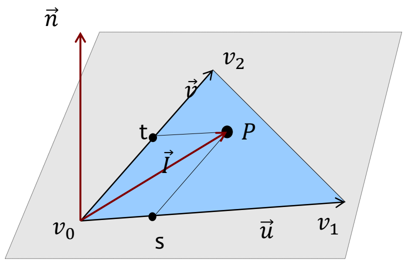
$$Ax + By + Cz + D = 0 \Rightarrow \vec{n} \cdot P + D = 0 \Rightarrow \vec{n} \cdot (o + t \cdot \vec{v}) + D = 0$$
$$\Rightarrow t = \frac{-D - \vec{n} \cdot o}{\vec{n} \cdot \vec{v}}$$

Ray-triangle intersection

- First compute intersection with triangle plane
- Then check if intersection point is inside the triangle



Ray-triangle intersection



$$\vec{u} = v_1 - v_0$$
$$\vec{v} = v_2 - v_0$$
$$P = v_0 + \mathrm{s} \cdot \vec{u} + \mathrm{t} \cdot \vec{v}$$

Work with v_0 as origin:

$$\vec{I} = P - v_0 \Rightarrow \vec{I} = \mathrm{s} \cdot \vec{u} + \mathrm{t} \cdot \vec{v}$$

$$\vec{I} = \mathrm{s} \cdot \vec{u} + \mathrm{t} \cdot \vec{v} \Rightarrow \begin{cases} I_x = su_x + tv_x \\ I_y = su_y + tv_y \\ I_z = su_z + tv_z \end{cases}$$

$$s = \frac{(\vec{u} \cdot \vec{v})(\vec{I} \cdot \vec{v}) - (\vec{v} \cdot \vec{v})(\vec{I} \cdot \vec{u})}{(\vec{u} \cdot \vec{v})^2 - (\vec{u} \cdot \vec{u})(\vec{v} \cdot \vec{v})}$$
$$t = \frac{(\vec{u} \cdot \vec{v})(\vec{I} \cdot \vec{u}) - (\vec{u} \cdot \vec{u})(\vec{I} \cdot \vec{v})}{(\vec{u} \cdot \vec{v})^2 - (\vec{u} \cdot \vec{u})(\vec{v} \cdot \vec{v})}$$

Intersection if 0 <= s+t <= 1

Ray-triangle intersection precomputations

$$s = \frac{(\vec{u} \cdot \vec{v})(\vec{I} \cdot \vec{v}) - (\vec{v} \cdot \vec{v})(\vec{I} \cdot \vec{u})}{(\vec{u} \cdot \vec{v})^2 - (\vec{u} \cdot \vec{u})(\vec{v} \cdot \vec{v})}$$
$$t = \frac{(\vec{u} \cdot \vec{v})(\vec{I} \cdot \vec{u}) - (\vec{u} \cdot \vec{u})(\vec{I} \cdot \vec{v})}{(\vec{u} \cdot \vec{v})^2 - (\vec{u} \cdot \vec{u})(\vec{v} \cdot \vec{v})}$$

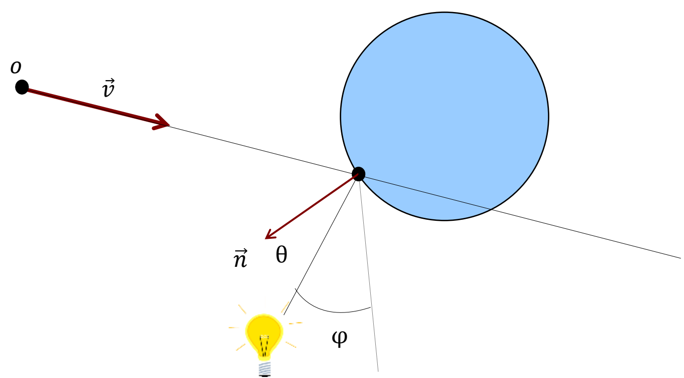
$$s = \frac{UV(\vec{I} \cdot \vec{v}) - VV(\vec{I} \cdot \vec{u})}{UV2 - UU \cdot VV} \qquad \Rightarrow \qquad s = \frac{UV(\vec{I} \cdot \vec{v}) - VV(\vec{I} \cdot \vec{u})}{D}$$
$$t = \frac{UV(\vec{I} \cdot \vec{u}) - UU(\vec{I} \cdot \vec{v})}{UV2 - UU \cdot VV} \qquad \qquad t = \frac{UV(\vec{I} \cdot \vec{u}) - UU(\vec{I} \cdot \vec{v})}{D}$$

Only four floats precomputed: UV, VV, UU, D

Local illumination

- Same formula than for OpenGL:

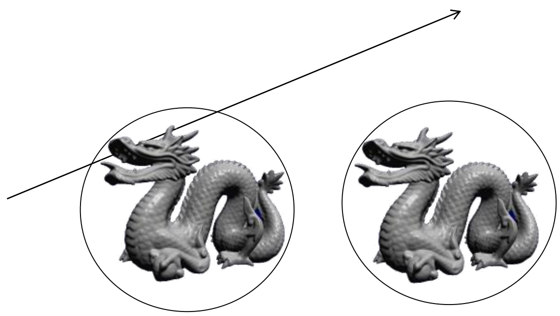
$$I_R = A_R \cdot kd_R + \sum_{i=0..N_f} \left(I_R^i \cdot kd_R \cdot \cos(\theta_i) + I_R^i \cdot ks_R \cdot \cos^n(\varphi_i) \right)$$



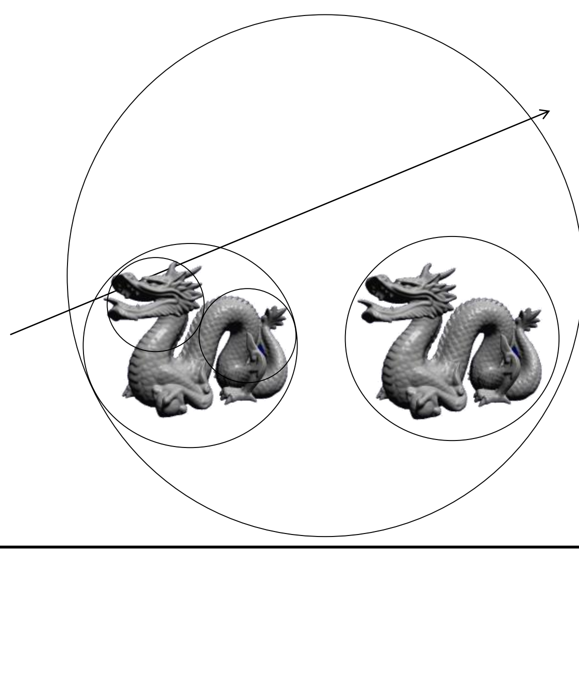
Accelerations

- Bounding volumes
- Uniform grids
- Octrees

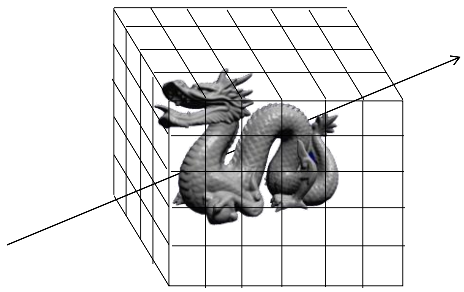
Bounding volumes



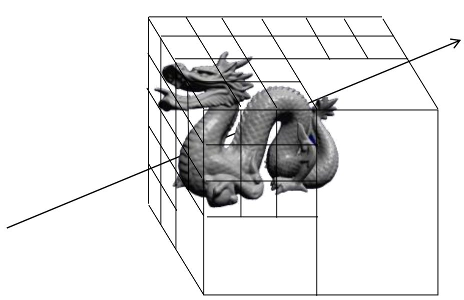
Bounding volumes



Uniform grids

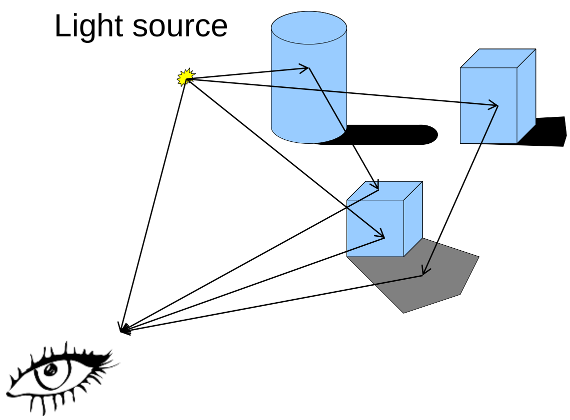


Octrees



Beyond local illumination

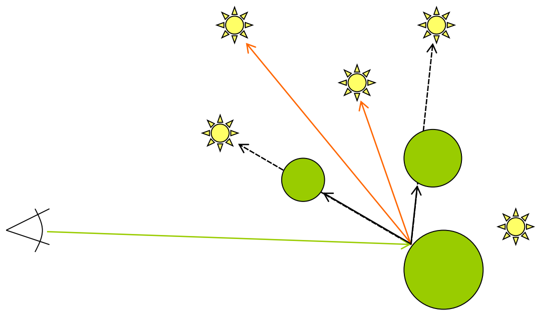
- Light bounces on surfaces



Material optical properties

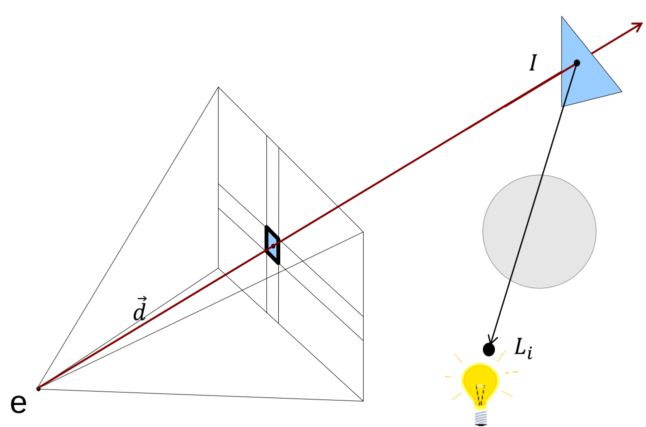
- Diffuse
- Specular
- Transparent

Shadows



$$I_R = A_R \cdot kd_R + \sum_{i=0..N_f} \mathbf{V_i} \cdot \left(\left(I_R^i \cdot kd_R \cdot \cos(\theta_i) + I_R^i \cdot ks_R \cdot \cos^n(\varphi_i) \right) \right)$$

Shadow ray



$$I + t \cdot \overrightarrow{d_{Li}}, d_{Li} = \frac{L_i - I}{\| L_i - I \|}$$

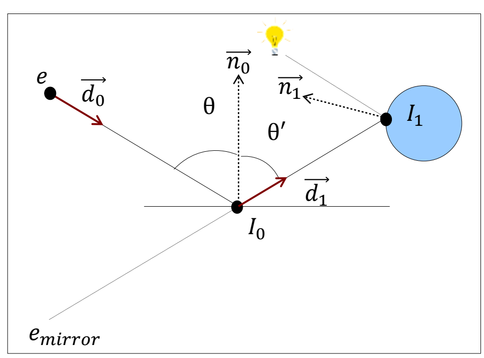
$$V_i = \begin{cases} 0 \; if \; \exists t \Rightarrow t \in (0, \| L_i - I \|) \\ 1 \quad otherwise \end{cases}$$

Cost of shadow computation

- For an 1000x1000 image with 10 light sources:
  - Ray casting:
    - Pixels rays: 1000x1000
    - Total: 1.000.000
  - Ray Tracing:
    - Pixels rays: 1000x1000
    - For each pixel: 10 shadow rays → 1000x1000x10
    - Total: 10.000.000
    - Upper bound:
      - Not all pixels intersect with an object
      - Lights back facing intersection point do not need shadow ray

Specular (mirror) reflections

- Recursivity:



First *ray:* $e + t \cdot \overrightarrow{d_0}$

First intersection point: $I_0, \overrightarrow{n_0}$

I_0 *belongs to a mirror surface* $\Rightarrow$ *recursive ray tracing*

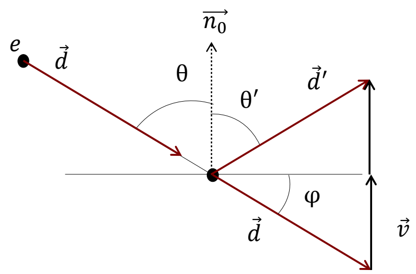
Perfect reflection direction: $\overrightarrow{d_1}$

Second ray: $I_0 + t \cdot \overrightarrow{d_1}$

Second intersection point: $I_1, \overrightarrow{n_1}$

I_1 *is not a mirror* $\Rightarrow$ *compute illumination and return*

Perfect reflection direction



$$\left.\begin{matrix}\vec{v} = \sin(\varphi) \cdot \vec{n} \\ \vec{d}' = \vec{d} + 2 \cdot \vec{v}\end{matrix}\right\} \Rightarrow \varphi = \frac{\pi}{2} - \theta \Rightarrow \sin(\varphi) = \cos(\theta) = -\vec{d} \cdot \vec{n}$$

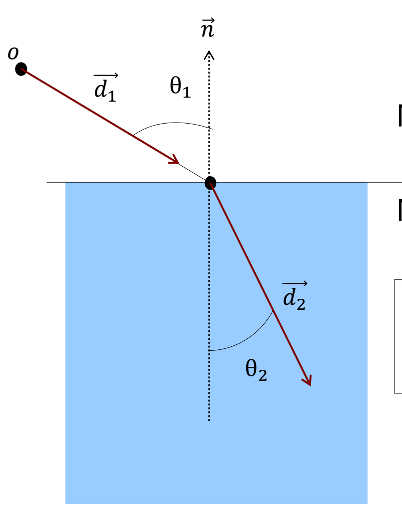
$$\boxed{\vec{d}' = \vec{d} - 2\vec{n}(\vec{d} \cdot \vec{n})}$$

Refraction

- Wikipedia: “Refraction is the change in direction of a wave due to a change in its speed”
- This happens when one wave passes from one medium to another:
  - Air – cristal
  - Air – water
  - ...

Refraction

Snell's law

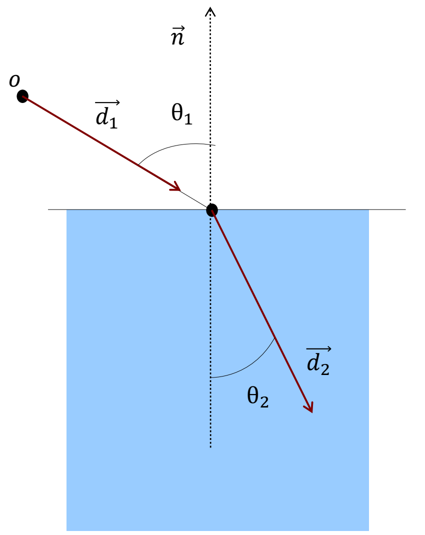


Medium 1: refraction index n1

Medium 2: refraction index n2

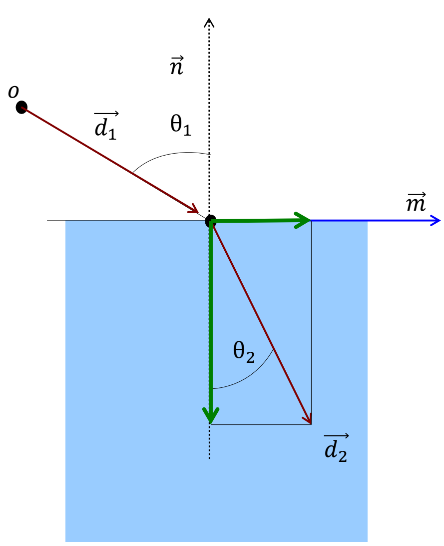
Snell's *Law*: $\dfrac{\sin(\theta_1)}{\sin(\theta_2)} = \dfrac{n_2}{n_1}$

Refraction direction



We *know:* $o, \overrightarrow{d_1}, \vec{n}, n_1, n_2$

Refraction direction

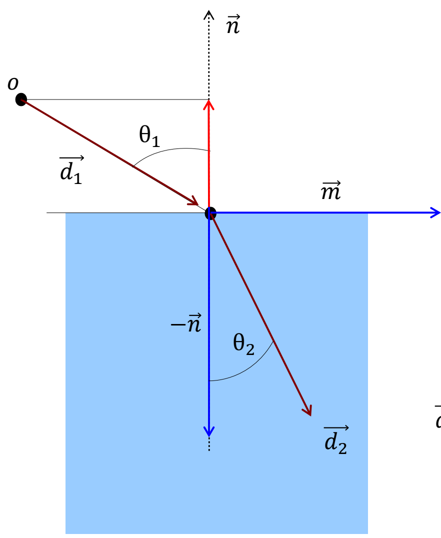


We *can express* $\overrightarrow{d_2}$ *as:*

$$\overrightarrow{d_2} = \sin(\theta_2)\vec{m} - \cos(\theta_2)\vec{n}$$

where: $\| \vec{m} \| = 1$

Refraction direction



We *can express* $\vec{m}$ *as:*

$$\vec{m} = \frac{\vec{d_1} + \vec{n} \cdot \cos(\theta_1)}{\sin(\theta_1)}$$

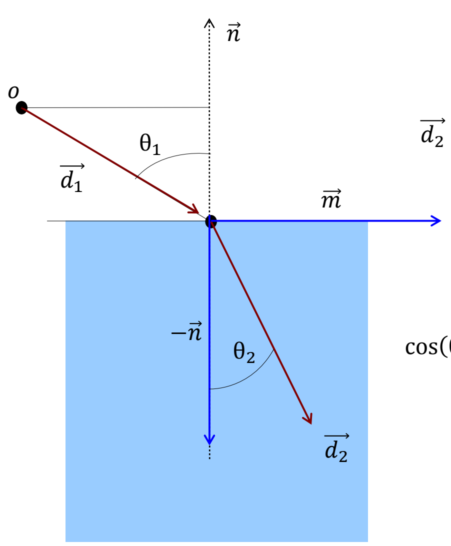
Substituting *in:*

$$\vec{d_2} = \sin(\theta_2) \cdot \vec{m} - \cos(\theta_2) \cdot \vec{n}$$

we get:

$$\vec{d_2} = \frac{\sin(\theta_2)}{\sin(\theta_1)}\left(\vec{d_1} + \vec{n} \cdot \cos(\theta_1)\right) - \cos(\theta_2) \cdot \vec{n}$$

Refraction direction



Using *Snell's law:* $\frac{\sin(\theta_2)}{\sin(\theta_1)} = \frac{n_1}{n_2}$

$$\vec{d_2} = \frac{n_1}{n_2}\left(\vec{d_1} + \vec{n} \cdot \cos(\theta_1)\right) - \cos(\theta_2) \cdot \vec{n}$$

And *then:*

$$\cos(\theta_2) = \sqrt{1 - \sin(\theta_2)^2} = \sqrt{1 - \sin(\theta_1)^2 \cdot \frac{n_1^2}{n_2^2}}$$

Illumination in ray tracing

- More components than ray casting
  - Diffuse (same as ray casting, with shadow)
  - Specular (same as ray casting, with shadows)
  - Mirror reflection (recursive)
  - Refraction (recursive)

Illumination in ray tracing

$$
\begin{aligned}
I \quad = \quad & A_R \cdot kd_R \\
& + \sum_{i=0..N_f} V_i \cdot \left(\left(I_R^i \cdot kd_R \cdot \cos(\theta_i) + I_R^i \cdot ks_R \cdot \cos^n(\varphi_i) \right) \right) \\
& + K_s \cdot I_{mirror} \\
& + K_r \cdot I_{refraction}
\end{aligned}
$$

$I_{mirror}, I_{refraction}$ are computed recursively

Where to stop ?!

Algorithm

```
action RayTraceScene(scene, camera)
  for each Pixel px in camera do
     r=defineRay(e, px, camera);
     color=RayTrace(scene,r);
     setPixel(px.i, px.j, color);
  end for
end action
```

Algorithm (no recursivity)

```
action RayTrace(scene, r)
  hit = intersectScene(scene, r)
  if !interaction(hit) do
    return (0,0,0)
  end if
  I = (0,0,0)
  I += ambient(hit, scene)
  for each Light l in scene do
    if lightVisible(l, hit, scene) do
      I += diffuseComponent(hit, l)
      I += specularComponent(hit, l)
    end if
  end for
  return I
end action
```

Recursive algorithm

```
action RayTrace(scene, r)
  // Compute ambient, diffuse and specular components
  // as before...
  d1 = computeReflectionDiretion(r, hit)
  Ray r1(hit.Point(), d1)
  I += hit.KM() * RayTrace(scene, r1)
  d2 = computeRefractionDiretion(r, hit)
  Ray r2(hit.Point(), d2)
  I += hit.KR() * RayTrace(scene, r2)
  return I
end action
```

Ray tree depth

- When to stop recursivity ?
- Two main choices:
  - Maximum fixed tree depth
  - Minimum contribution to recurse

Algorithm (max depth)

```
action RayTrace(scene, r, depth)
  // Compute ambient, diffuse and specular components
  // as before...
  if (depth < MAX_DEPTH) do
    d1 = computeReflectionDiretion(r, hit)
    Ray r1(hit.Point(), d1)
    I += hit.KM() * RayTrace(scene, r1, depth+1)
    d2 = computeRefractionDiretion(r, hit)
    Ray r2(hit.Point(), d2)
    I += hit.KR() * RayTrace(scene, r2, depth+1)
  end if
  return I
end action
```

Algorithm (min contribution)

```
action RayTrace(scene, r, c)
  // Compute ambient, diffuse and specular components
  // as before...
  if (c > MIN_CONTRIB) do
    d1 = computeReflectionDiretion(r, hit)
    Ray r1(hit.Point(), d1)
    I += hit.KM() * RayTrace(scene, r1, c * hit.KM())
    d2 = computeRefractionDiretion(r, hit)
    Ray r2(hit.Point(), d2)
    I += hit.KR() * RayTrace(scene, r2, c * hit.KR())
  end if
  return I
end action
```

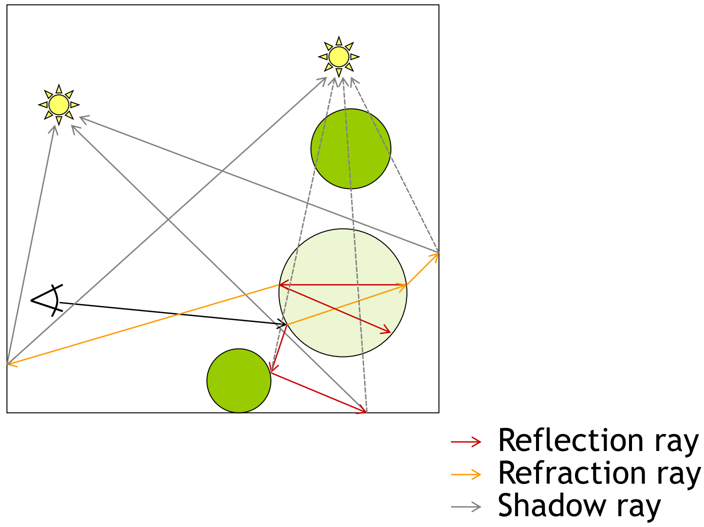
Algorithm (combined)

```
action RayTrace(scene, r, c, d)
  // Compute ambient, diffuse and specular components
  // as before...
  if (d < MAX_DEPTH and c > MIN_CONTRIB) do
    d1 = computeReflectionDiretion(r, hit)
    Ray r1(hit.Point(), d1)
    I += hit.KM() * RayTrace(scene, r1, c * hit.KM(), d+1)
    d2 = computeRefractionDiretion(r, hit)
    Ray r2(hit.Point(), d2)
    I += hit.KR() * RayTrace(scene, r2, c * hit.KR(), d+1)
  end if
  return I
end action
```

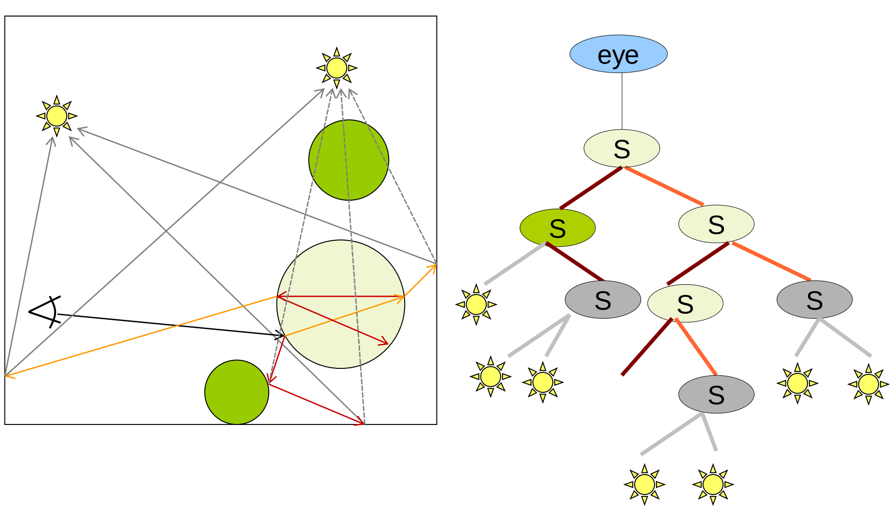
Ray tree

- Ray tracing produces a tree of rays
- Each intersection is a new node
- Each node can have two types of children:
  - Non-recursive: shadow rays
  - Recursive: reflection and/or refraction rays

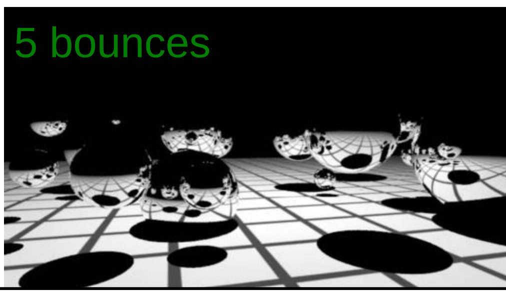
Ray tree

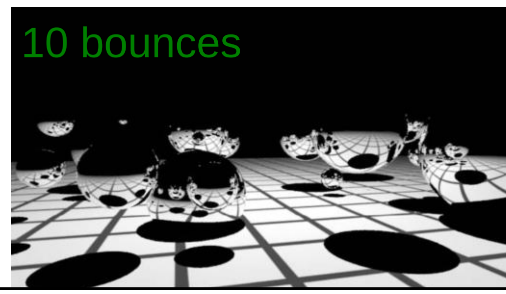


Ray tree

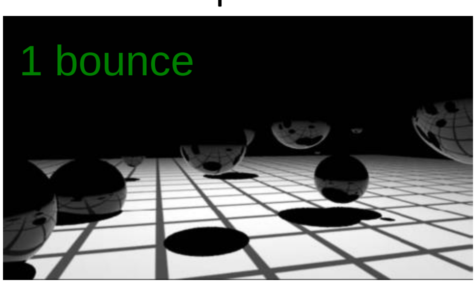


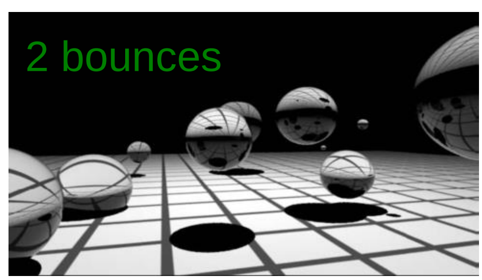
Example: Reflection

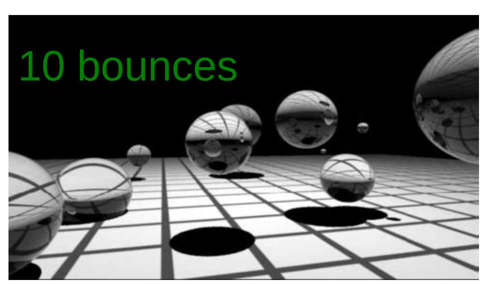




Example Refraction



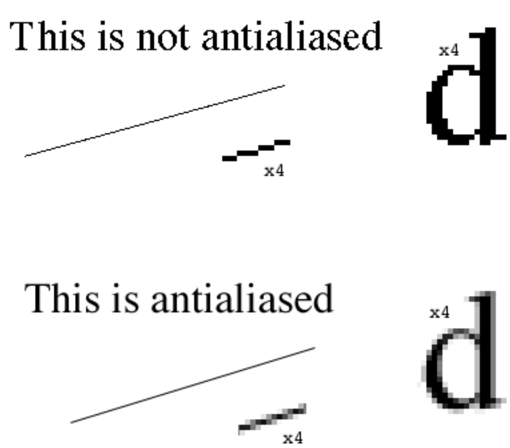




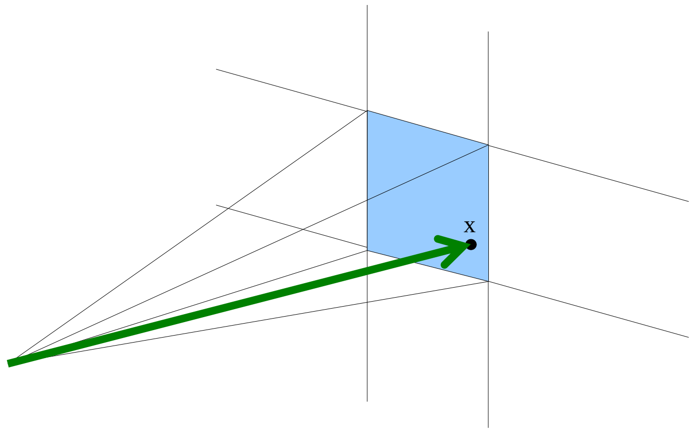
Example

Reflection + Refraction (10 bounces)

Antialiasing



Antialiasing

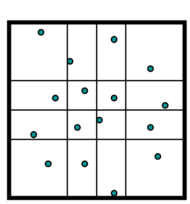


$$\int_S RT(x)dx$$

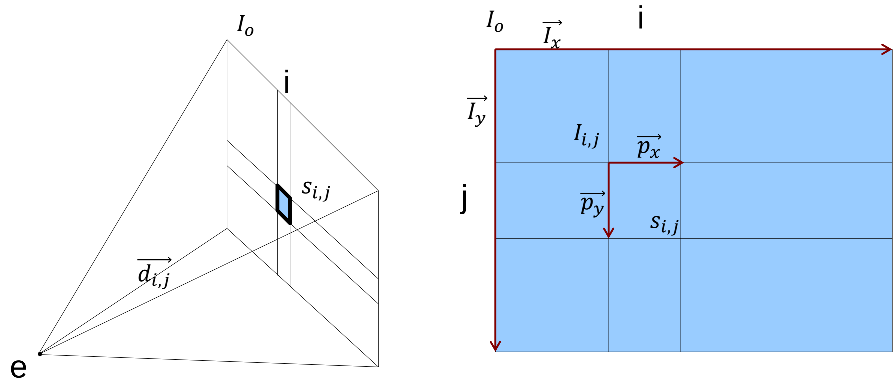
S = pixel area

Antialiasing

- Trace several random rays through each pixel
- Directions determined using *jittering*
- Can easily implement area-weighted jittered Gaussian distribution



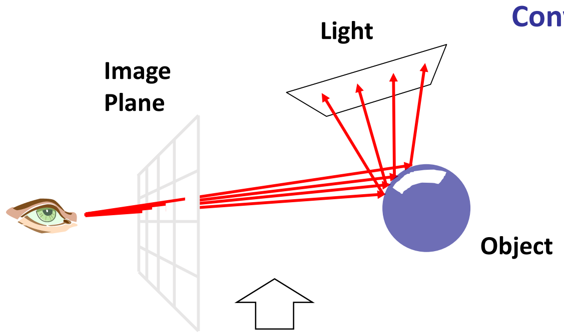
Generate pixel samples



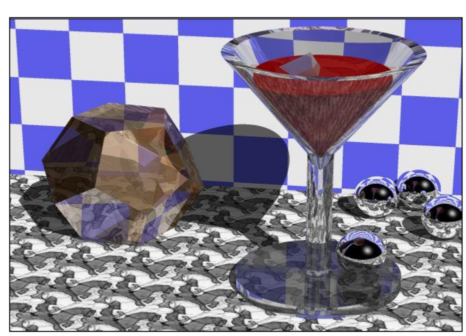
$$s_{i,j} = I_{i,j} + u \cdot \overrightarrow{p_x} + v \cdot \overrightarrow{p_y} \Rightarrow I_0 + (i+u) \cdot \overrightarrow{p_x} + (j+u) \cdot \overrightarrow{p_y}$$

$$\mathit{where{:}}\, u \in [0,1]\, \mathit{and}\, v \in [0,1]$$

Conventional Ray Tracing



- Infinitesimally thin rays
- Perfect sharp multiple reflections and shadows
- Not in reality!
- Surfaces are not perfectly smooth!



Distributed Ray Tracing

Cook-Porter-Carpenter (1984)

Apply distribution-based sampling to many parts of the ray-tracing algorithm
Rays can also be stochastically distributed in *object space* to simulate

Diffuse reflection

- Perturb directions reflection/transmission, with distribution based on angle from ideal ray

Depth of field

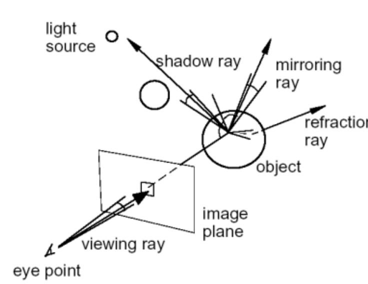
- Perturb eye position on lens

Soft shadows shadow

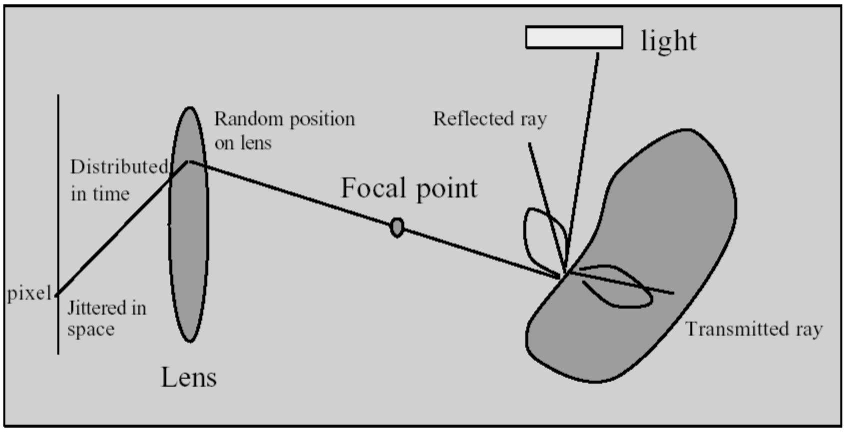
- Perturb illumination rays across area light

Motion blur

- Perturb eye ray samples in time

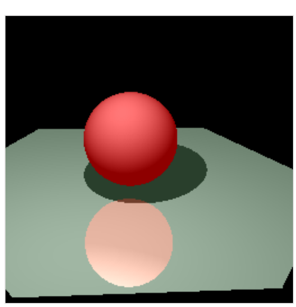


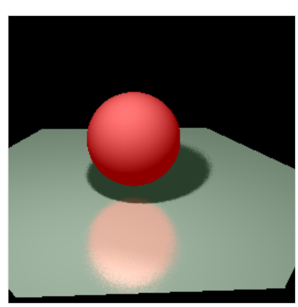
Distributed Ray Tracing



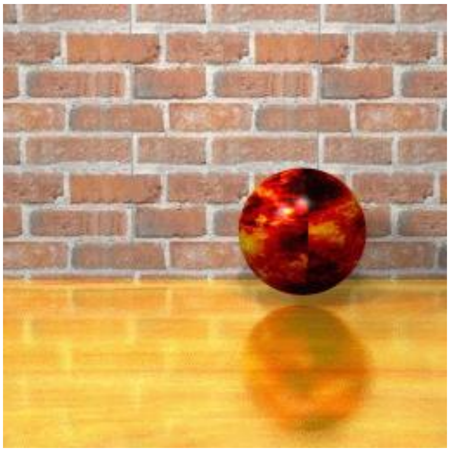
DRT: Diffuse reflection

- Blurry reflections and refractions are produced by randomly perturbing the reflection and refraction rays from their "true" directions.

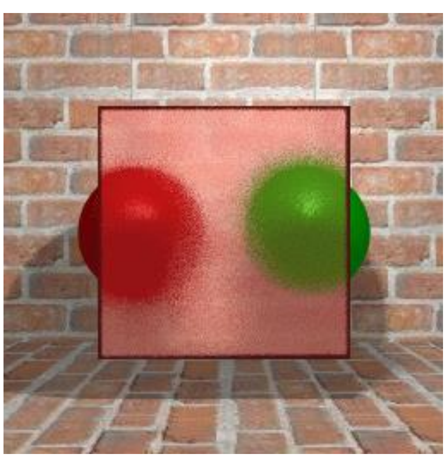


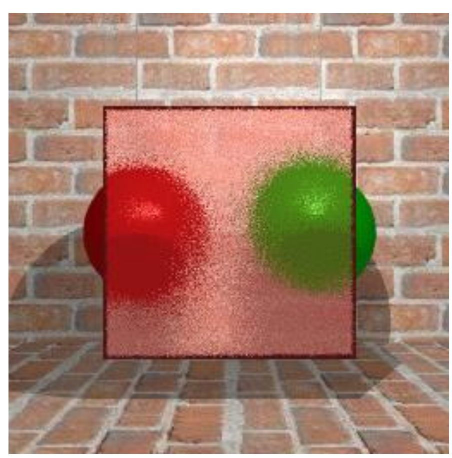


Reflection

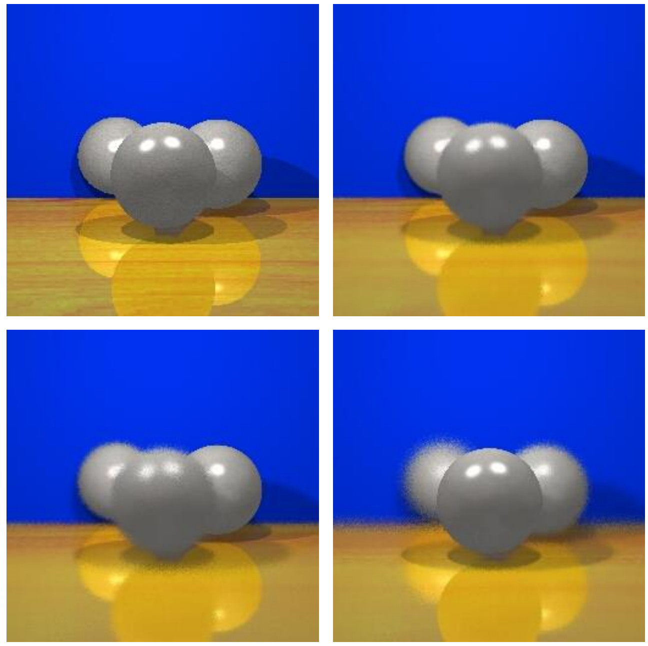


Transparency



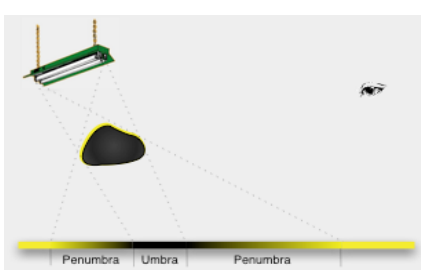


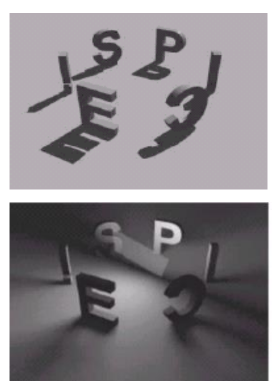
Depth of Field



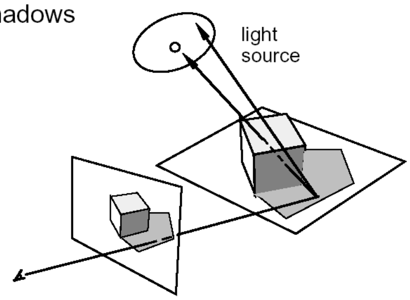
Area Lights

•Traditional point light sources are unrealistic:
-Hard shadows
-Sharp highlights
•Real lights have some shape (area), which produces:
-Soft shadows
-Soft lighting on objects
-Gives shape to highlights

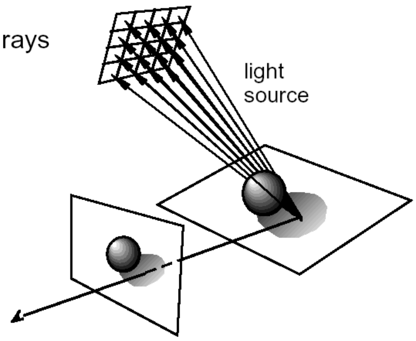


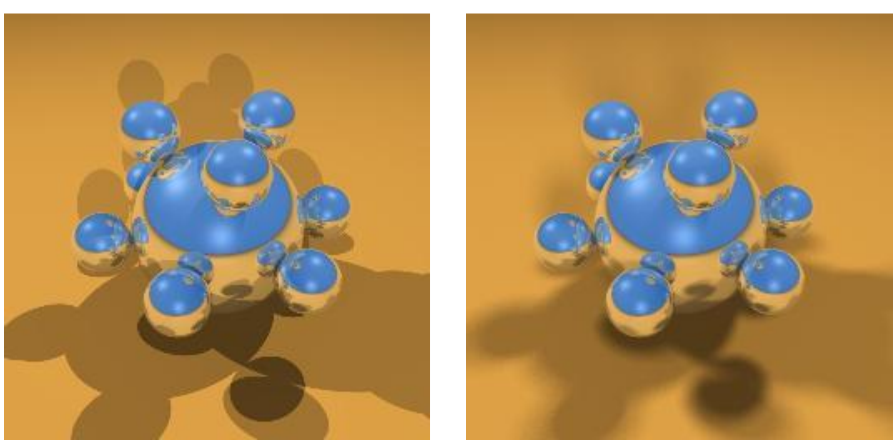


Soft shadows



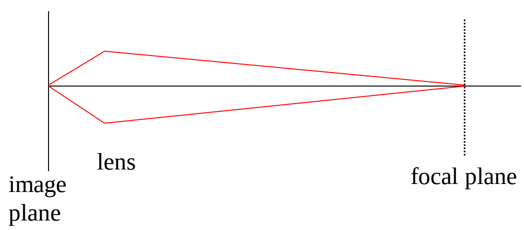
Distributing shadow rays is comparable to sampling an area light source with multiple rays





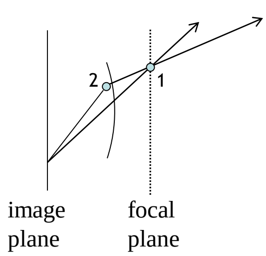
Depth of Field

- With a camera lens, only objects at the *focal distance* are sharp
  - *depth of field* refers to the zone of acceptable sharpness
  - In CG, "depth of field" refers to rendering including lens focus/blurring effect
- Amount of blurring depends on the aperture (how wide open the shutter is)
  - With a pinhole camera, there's no blurring
  - With a wider aperture, blurring increases (a point maps to a "circle of confusion")



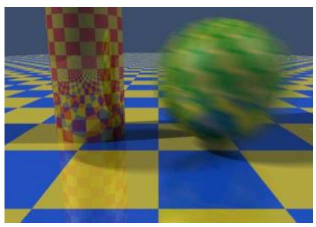
Depth of Field

- Distribute rays across the aperture
  - Can trace them through a real lens model or something simpler
  - For an object at the focal distance, whatever path the rays take all will reach the same spot on the object
  - For an object outside the depth of field, the different rays will hit different spots on the object
    - Combining the rays will yield a blur
- Start with normal eye ray and find intersection with focal plane
- Choose jittered point on lens and trace line from lens point to focal point



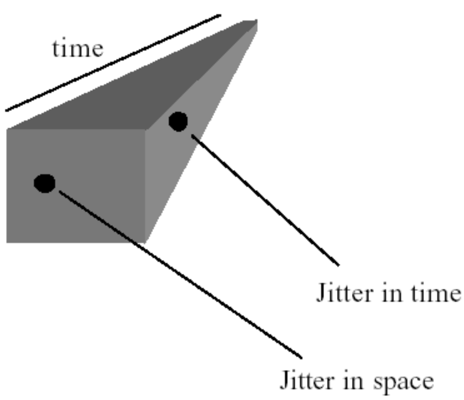
Motion Blur

- Assume we know the motion of our objects as a function of time
- Distribute rays in time
  - Give each ray a time value
    - E.g., jittered time distribution during the “shutter open”
  - For intersection test, use the object’s position at the ray’s time
  - Combine ray colors
    - If the object is moving, the result is motion blur



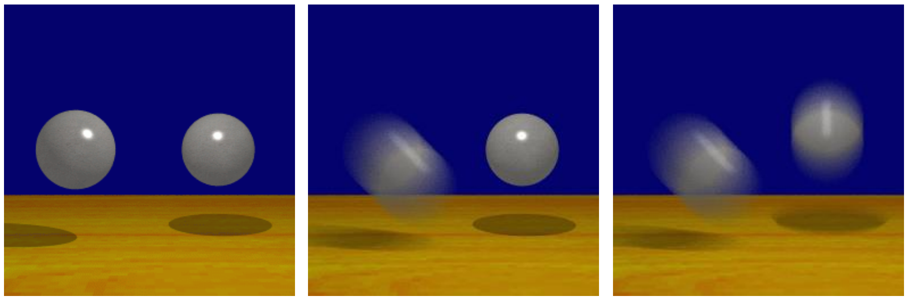
First CG image with motion blur, from [Cook84]

Temporal Jittering Sampling



7	11	3	14
4	15	13	9
16	1	8	12
6	10	5	2

Motion Blur



Remarks

- Ray Tracing
  - Good quality rendering
  - Not for interactive applications
  - “easy” to implement
- Next: Global Illumination