CS 655 - Advanced Computer Graphics

Ray Tracing Part I - The Basics

Ray Tracing

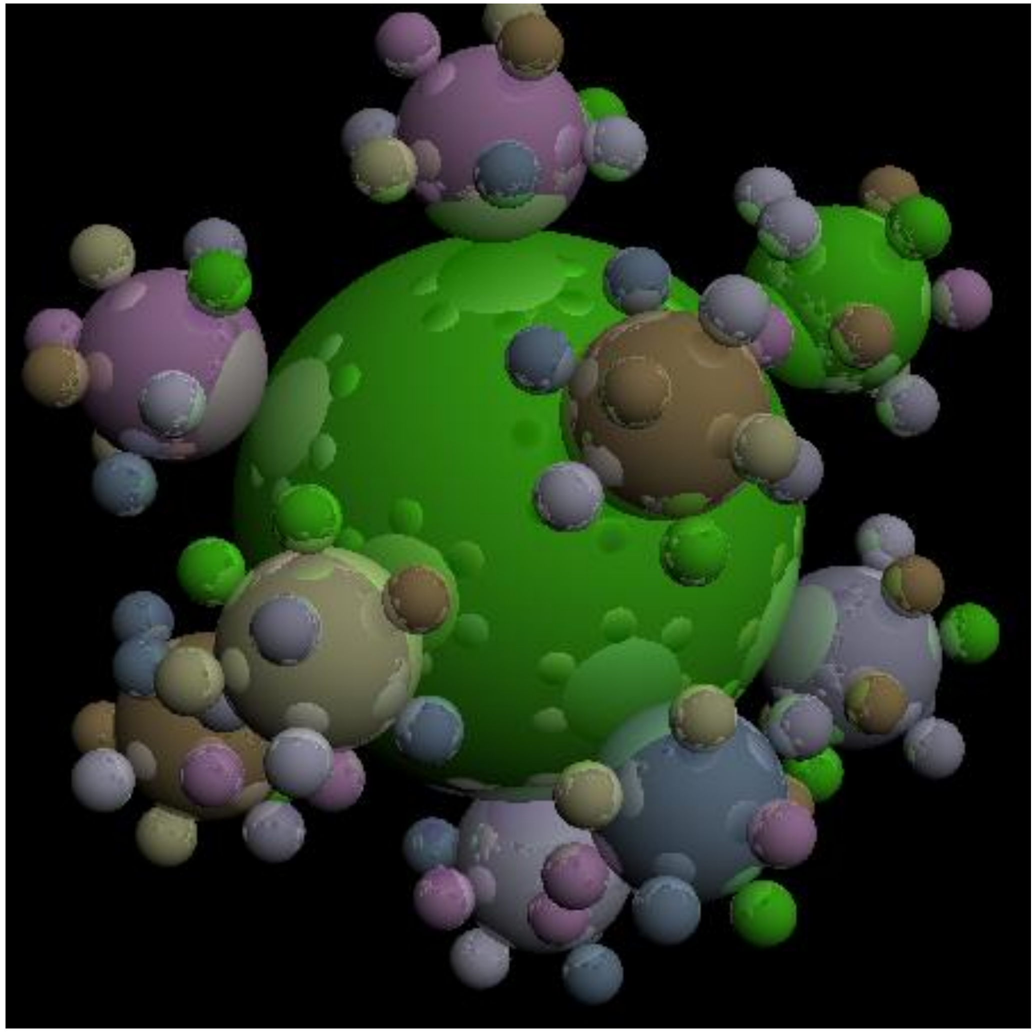
What is ray tracing?

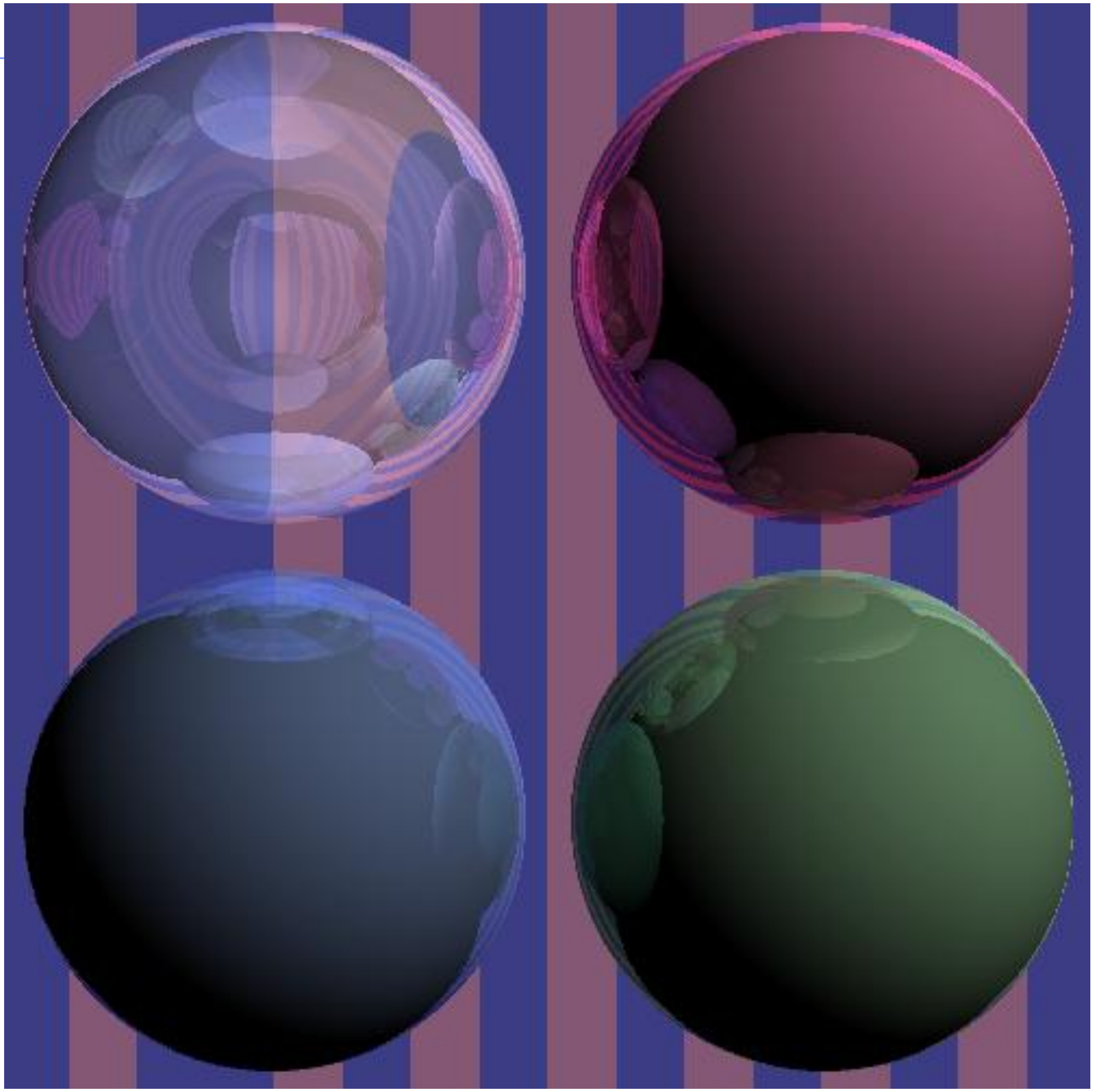
- Follow (trace) the path of a ray of light and model how it interacts with the scene
- When a ray intersects an object, send off secondary rays (reflection, shadow, transmission) and determine how they interact with the scene
- Basic algorithm allows for:
  - Hidden surface removal
  - Multiple light sources
  - Hard shadows
  - Reflections
  - Transparent refractions
- Extensions can achieve:
  - Soft shadows
  - Blurred reflections (glossiness)
  - Translucent refractions
  - Motion blur
  - Depth of field (finite apertures)
  - and more

Ray Tracing

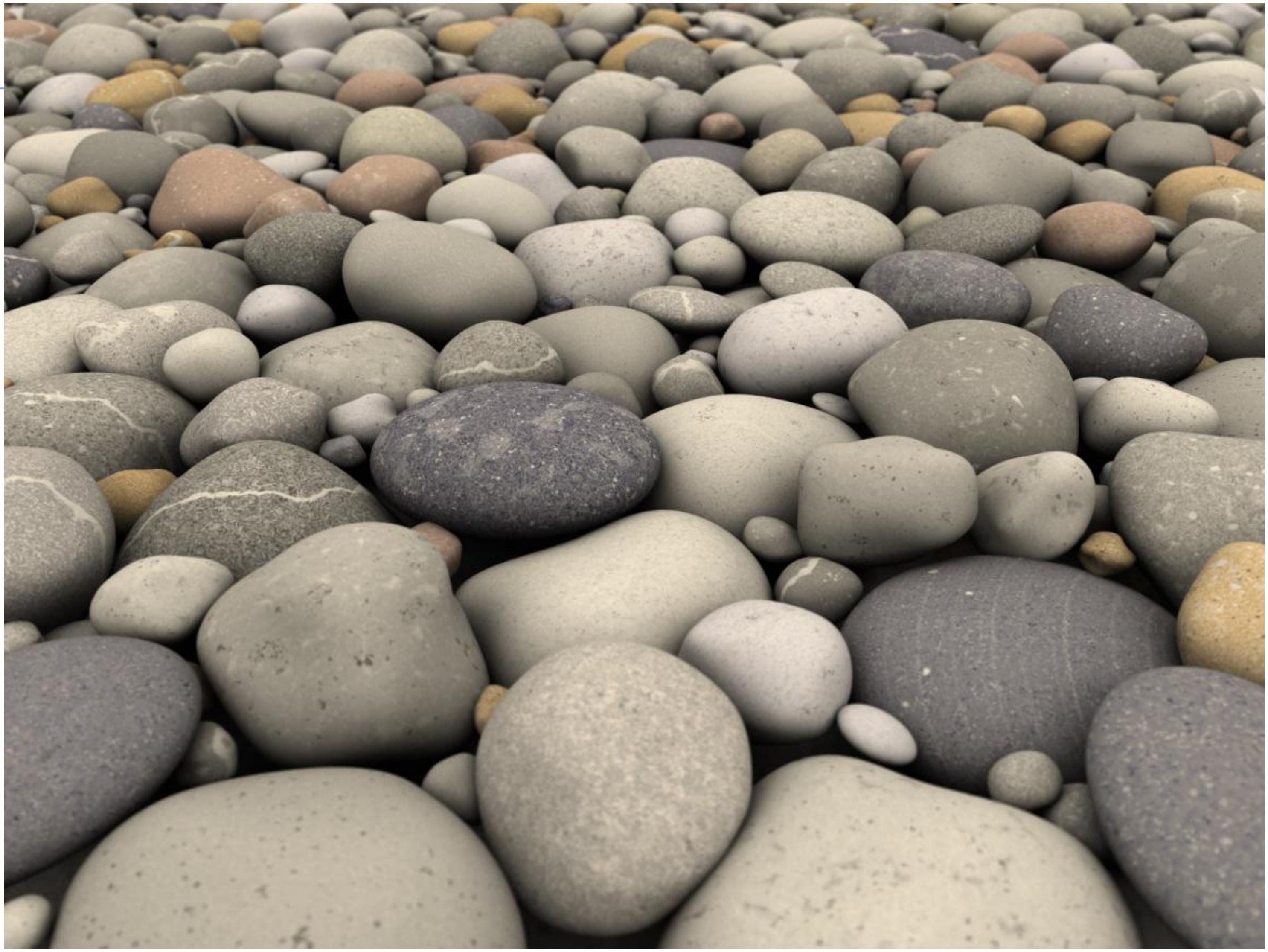
- Produces Highly realistic scenes
- Strengths:
  - Specular reflections
  - Transparency
- Weaknesses:
  - Color bleeding (diffuse reflections)
  - Time consuming
- References:
  - "An Improved Illumination Model for Shaded Display," Turner Whitted, CACM, June 1980.
  - "Distributed Ray Tracing," Cook, Porter, and Carpenter, Computer Graphics, July 1984, pp. 137-145.

Ray traced images



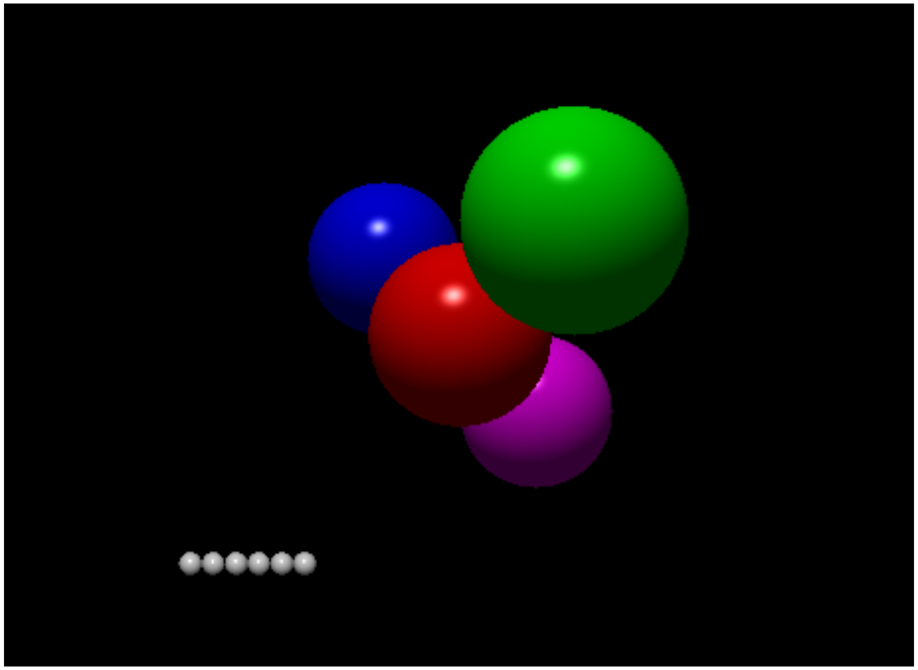



PXL-506
Pov-Tea



"Pebbles" by ***Jonathan Hunt*** *(2008)* 4.5 days to render on an Athlon 5600+

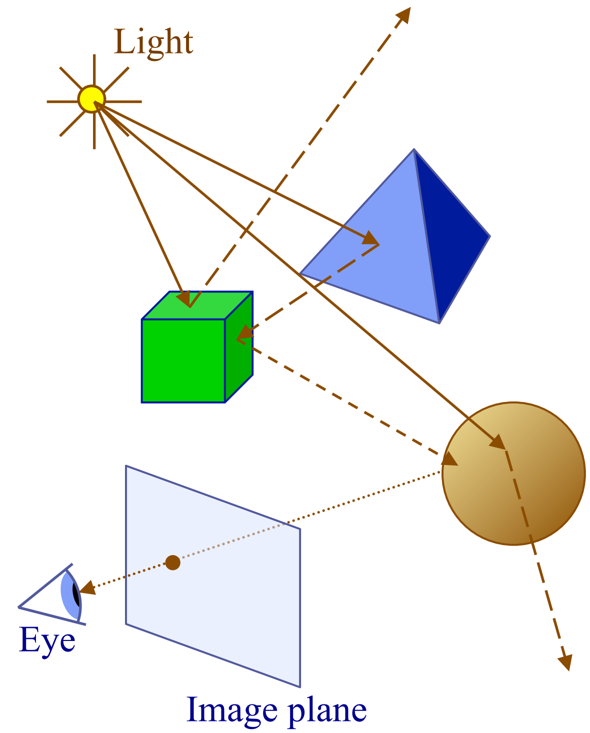
Asbjørn Heid implemented a realistic metal material to render this very realistic buddha model. (from PBRT Gallery)



22.5 fps on a PS3 using 7 cells in 6/2007 by Eric Rollins

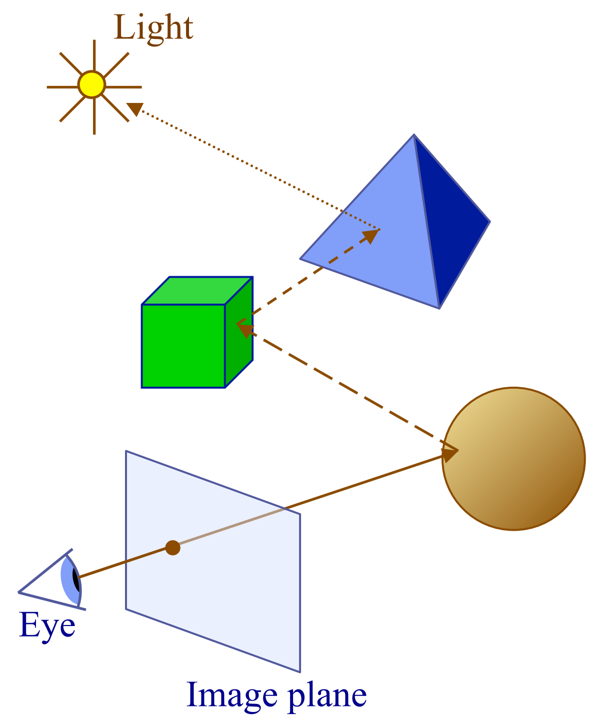
Ray Tracing

- "Backward" ray tracing:
  - Traces the ray *forward* (in time) from the light source through potentially many scene interactions
  - Physically based
  - Global illumination model:
    - Color bleeding
    - Caustics
    - Etc.
  - Problem: most rays will never even get close to the eye
  - Very inefficient since it computes many rays that are never seen



Ray Tracing

- "Forward" ray tracing:
  - Traces the ray *backward* (in time) from the eye, through a point on the screen
  - Not physically based
  - Doesn't properly model:
    - Color bleeding
    - Caustics
    - Other changes in light intensity and color due to refractions and non-specular reflections
  - More efficient: computes only visible rays (since we start at eye)
  - Generally, "ray tracing" refers to *forward* ray tracing



Ray Tracing

- Ray tracing is an image-precision algorithm: Visibility determined on a per-pixel basis
  - Trace one (or more) rays per pixel
  - Compute closest object (triangle, sphere, etc.) for each ray
- Produces realistic results
- Computationally expensive

1024×1024, 16 rays/pixel
~ 10 hours on a 99 MHz HP workstation

Ray Tracing

- Ray tracing is an image-precision algorithm: Visibility determined on a per-pixel basis
  - Trace one (or more) rays per pixel
  - Compute closest object (triangle, sphere, etc.) for each ray
- Produces realistic results
- Computationally expensive

4.5 days on Athlon 5600+

Minimal Ray Tracer

- A basic (minimal) ray tracer is simple to implement:
  - The code can even fit on a 3×5 card (code courtesy of Paul Heckbert with a small change to output as a PPM file):

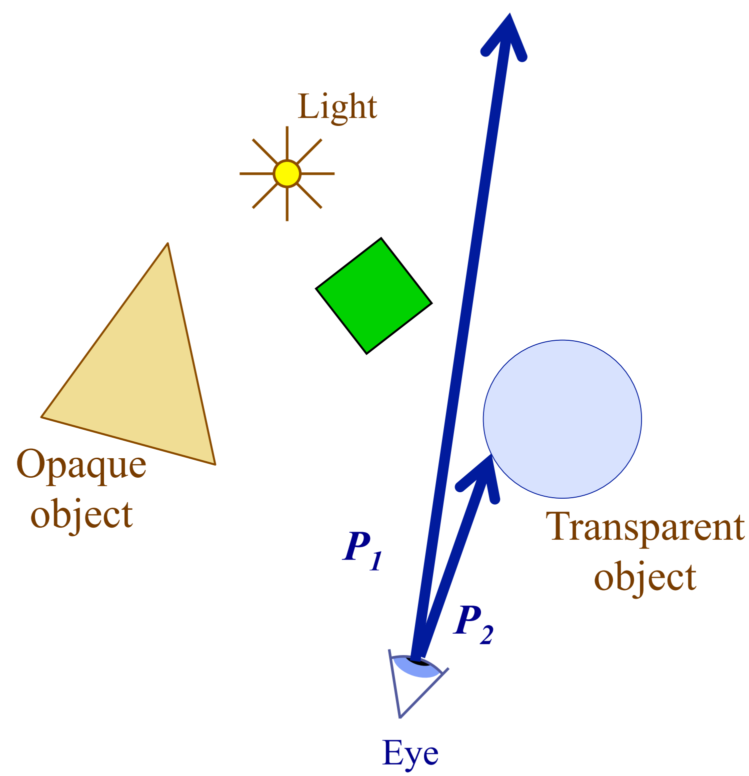
```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,
.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
1.,5.,0.,0.,0.,.5,1.5,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A
,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;{B.x+=a*
A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}vec vunit(A)vec A;{return vcomb(1./sqrt(
vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s=
sph+5;while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s
->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u:
tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return
amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen
)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d= -d;l=sph+5;while(l-->sph)if((e=l
->kl*vdot(N,U=vunit(vcomb(-1.,P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e
,l->color,color);U=s->color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta*
eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt
(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,
color,vcomb(s->kl,U,black))));}main(){puts("P3\n32 32\n255");while(yx<32*32)
U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,
trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}/*minray!*/
```

Minimal Ray Tracer

- This code implements:
  - Multiple spheres (with different properties)
  - Multiple levels of recursion:
    - Reflections
  - Transparency:
    - Refraction
  - One point light source:
    - Hard shadows
  - Hidden surface removal
  - Phong illumination model
  - It even has a comment

Ray Tracing: Types of Rays

- Primary rays:
  - Sent from the eye, through the image plane, and into the scene
  - May or may not intersect an object in the scene:
    - No intersection → set pixel color to background color (P_2)
    - Intersects object → send out secondary rays and compute lighting model (P_1)



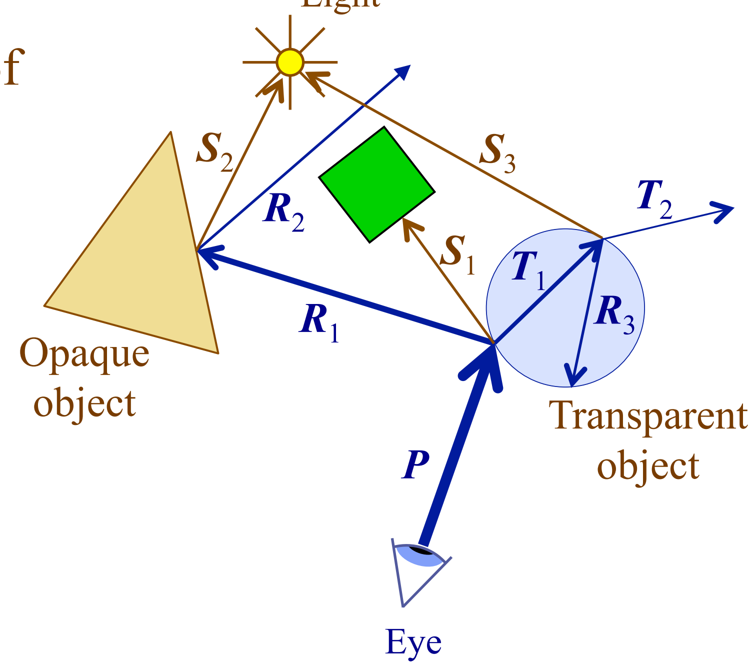
Ray Tracing: Types of Rays

- Secondary Rays:
  - Sent from the point at which the ray intersects an object
- Multiple types:

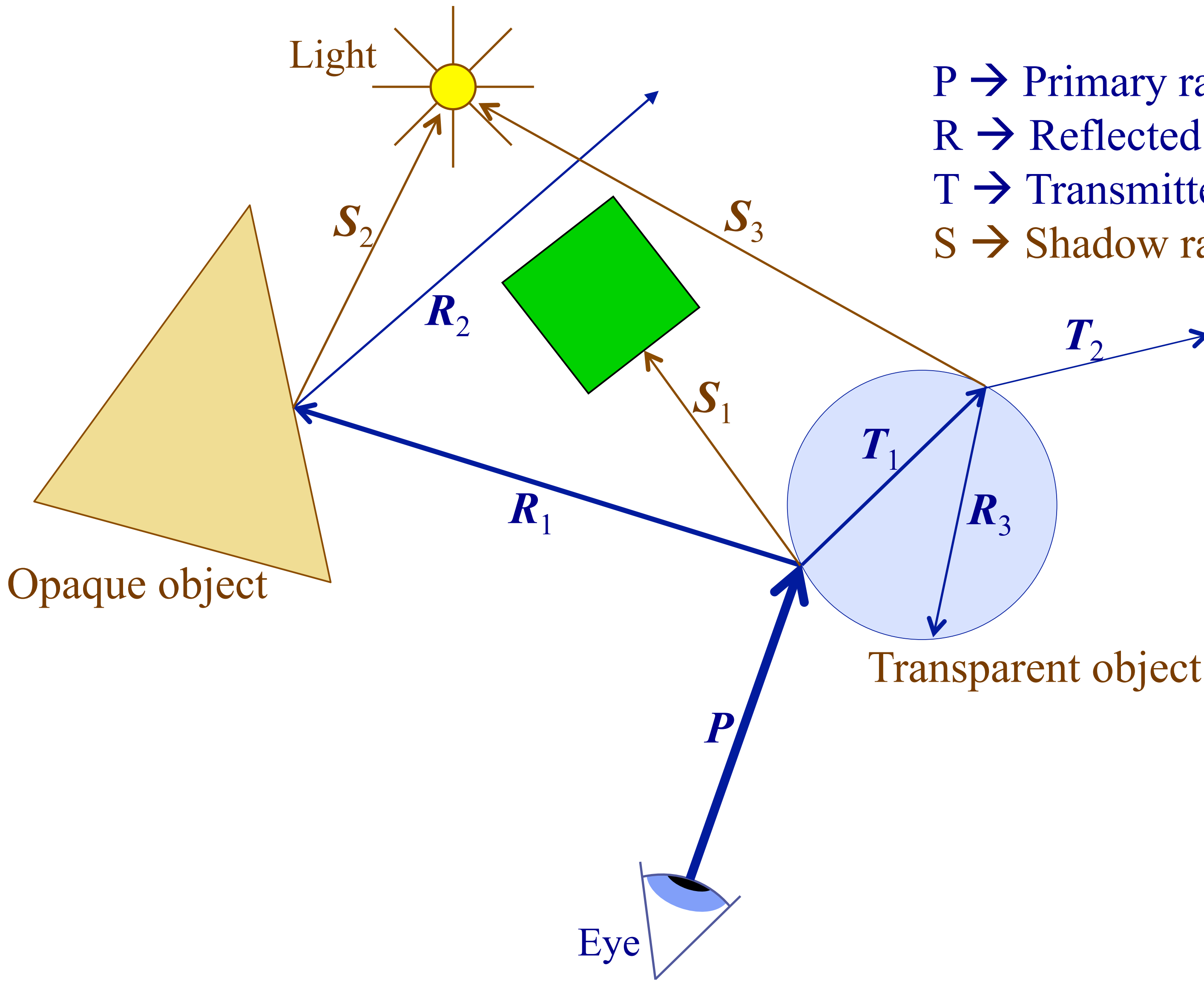
Transmission (T): sent in the direction of refraction

Reflection (R): sent in the direction of reflection, and used in the Phong illumination model

Shadow (S): sent toward a light source to determine if point is in shadow or not.



Ray Tracing: Types of Rays

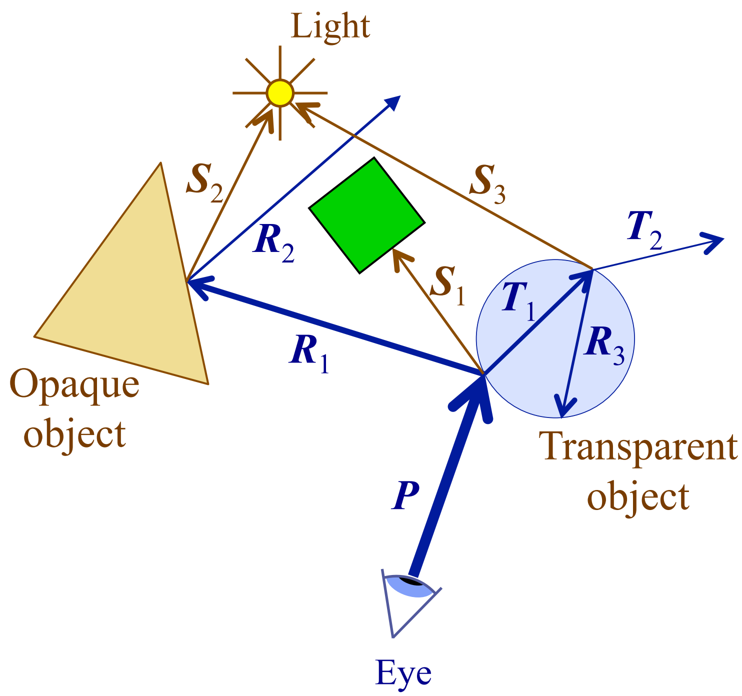


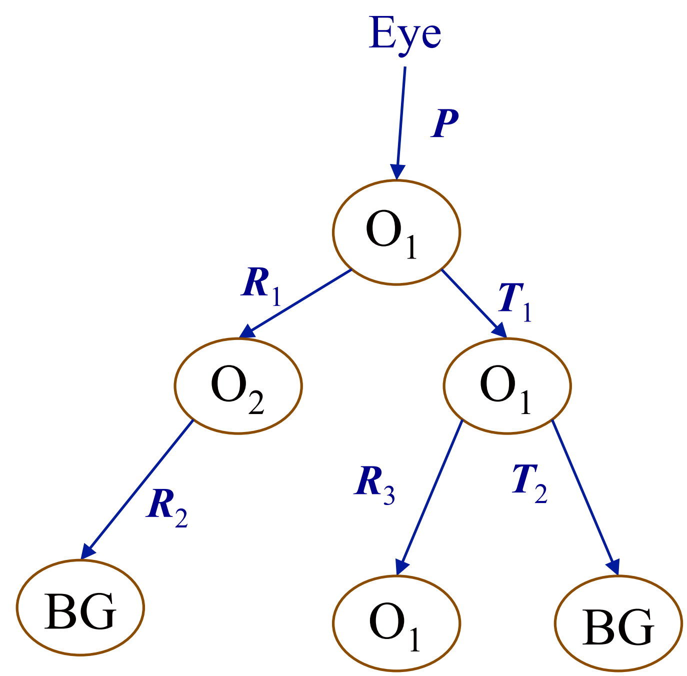
P → Primary rays
R → Reflected rays
T → Transmitted rays
S → Shadow rays

Ray Tracing: Ray Tree

- Each intersection may spawn secondary rays:
  - Rays form a ray tree
  - Nodes → Intersection points
  - Edges → Reflected/transmitted ray
- Rays are recursively spawned until:
  - Ray does not intersect any object
  - Tree reaches a maximum depth
  - Light reaches some minimum value
- Shadow rays are sent from every intersection point (to determine if point is in shadow), but they do not recursively spawn secondary rays

Ray Tracing: Ray Tree Example





Ray tree is evaluated from bottom up:

- Depth-first traversal
- Each node's color is calculated as a function of its children's colors

Basic Ray Tracing Algorithm

- Generate one ray for each pixel
- For each ray:
  - Determine the nearest object intersected by the ray
  - Compute intensity information for the intersection point using the illumination model
  - Calculate and trace reflection ray (if surface is reflective)
  - Calculate and trace transmission ray (if surface is transparent)
  - Calculate and trace shadow ray
  - Combine results of the intensity computation, reflection ray intensity, transmission ray intensity, and shadow ray information
  - If the ray misses all objects, set the pixel color to the background color

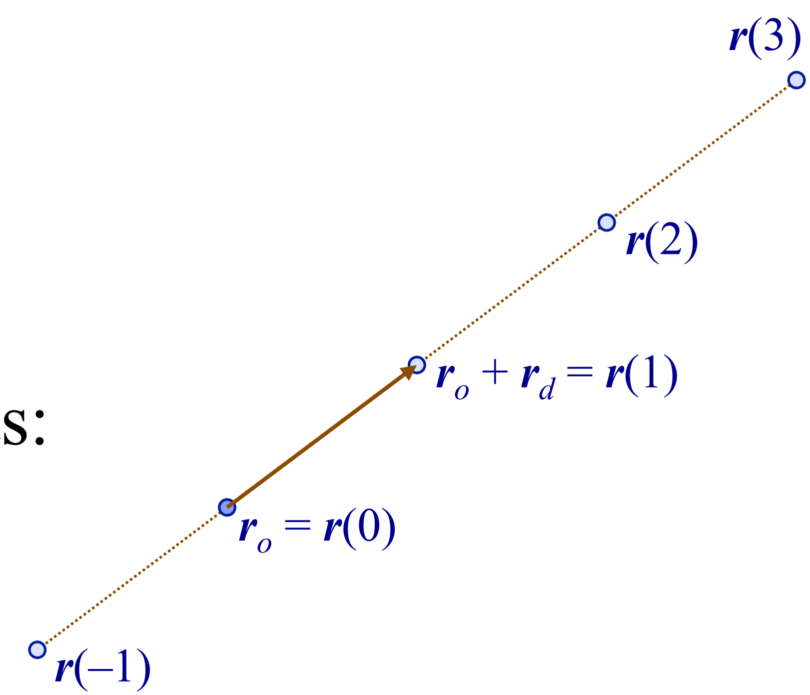
Tracing Rays

- Basic (non-recursive) ray tracing algorithm:
  1. Send a ray from the eye through the screen
  2. Determine which object that ray first intersects
  3. Compute pixel color
- Most (approx. 75%) of the time in step 2:
  - Simple method:
    - Compare every ray against every object and remember the closest object hit by each ray
  - Very time consuming:
    - Several optimizations possible

Ray Representation

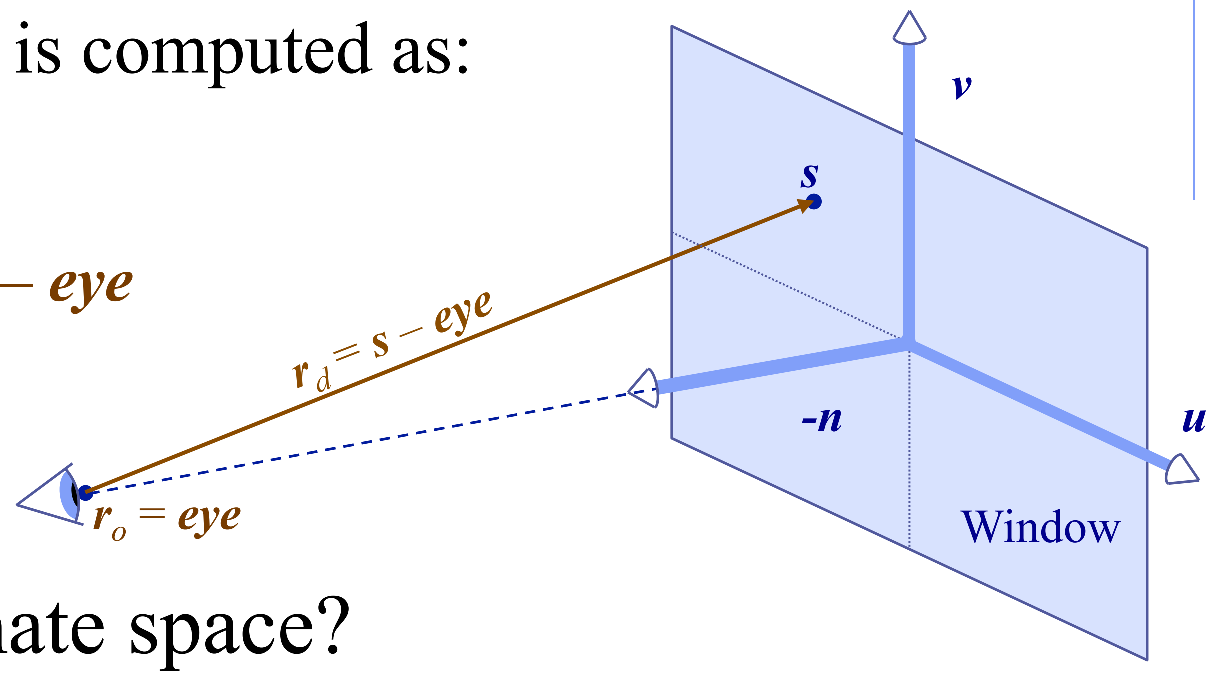
- A ray can be represented explicitly (in parametric form) as an origin (point) and a direction (vector):
  - Origin: $\boldsymbol{r}_o = \begin{bmatrix} x_o \\ y_o \\ z_o \end{bmatrix}$
  - Direction: $\boldsymbol{r}_d = \begin{bmatrix} x_d \\ y_d \\ z_d \end{bmatrix}$
- The ray consists of all points:

$$\boldsymbol{r}(t) = \boldsymbol{r}_o + \boldsymbol{r}_d t$$



Viewing Ray

- The primary ray (or viewing ray) for a point ***s*** on the view plane (i.e., screen) is computed as:
  - Origin: $\boldsymbol{r}_o = \boldsymbol{eye}$
  - Direction: $\boldsymbol{r}_d = \boldsymbol{s} - \boldsymbol{eye}$



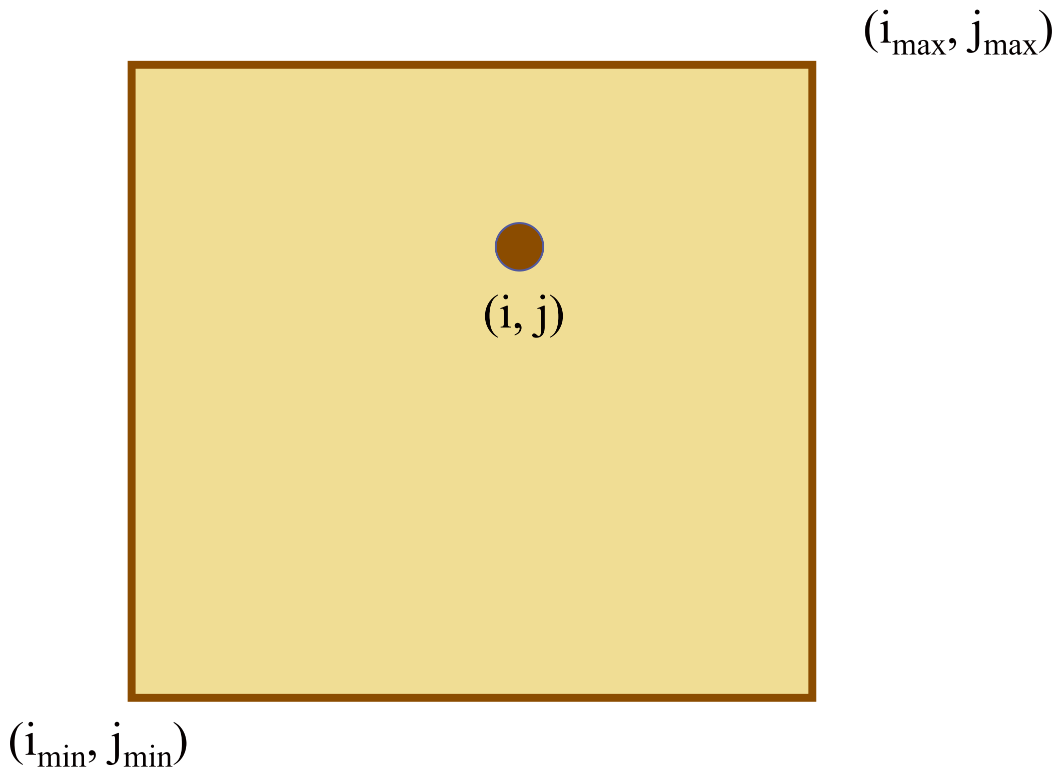
- Which coordinate space?
  - Want to define rays in terms world-space coordinates (x, y, z)
  - Eye is already in specified in terms of (x, y, z) position
  - Screen point ***s*** is easiest to define in terms of where it is on the window in viewing-space coordinates (u, v, n)

Viewing Ray: Screen Point

- Given:
  - Our scene in world-coordinates
  - A camera (eye) position in world-coordinates (x, y, z)
  - A pixel (i, j) in the viewport

- We need to:
  - Compute the point on the view plane window that corresponds to the (i, j) point in the viewport
  - Transform that point into world-coordinates

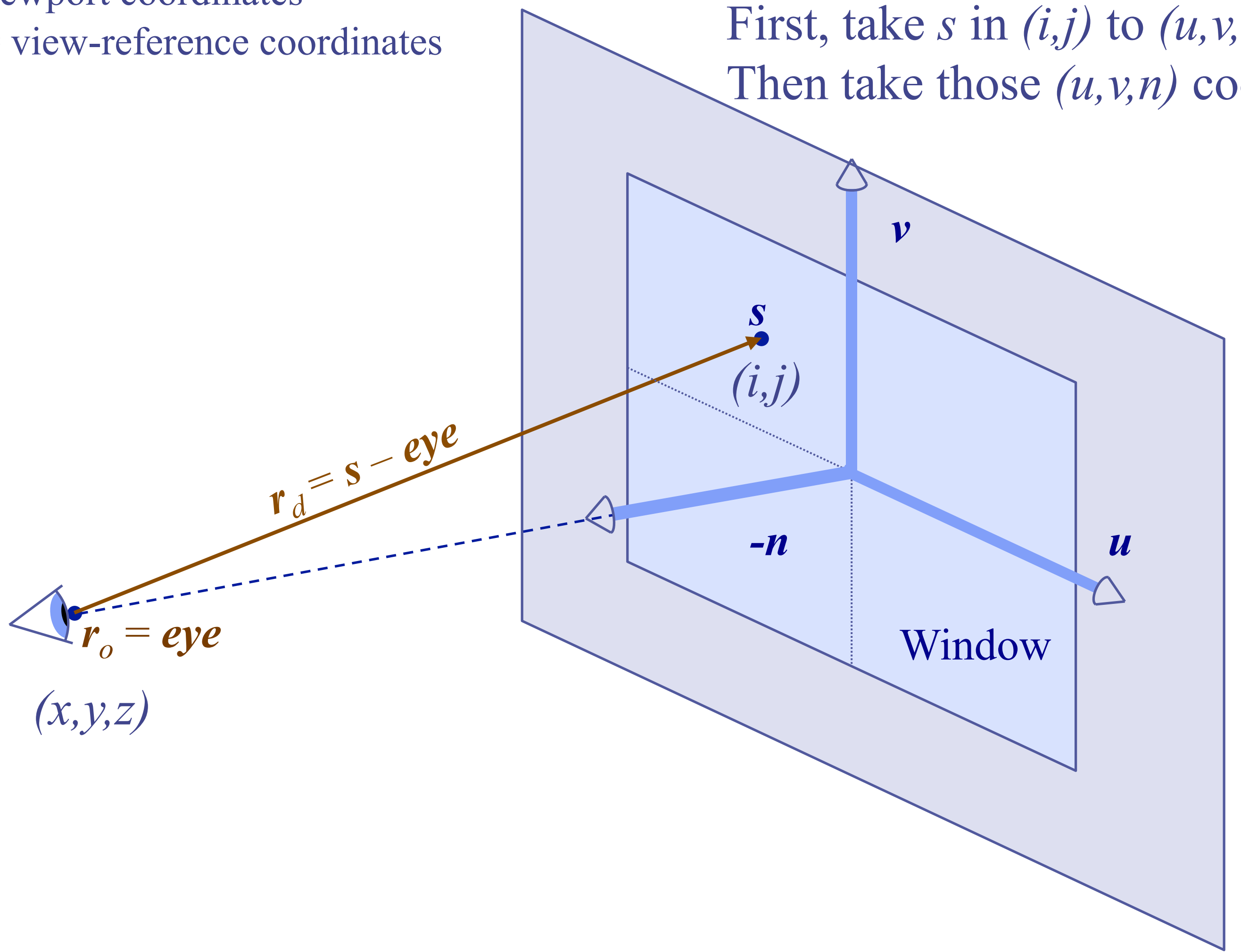
Viewport



Viewing Ray (in pictures)

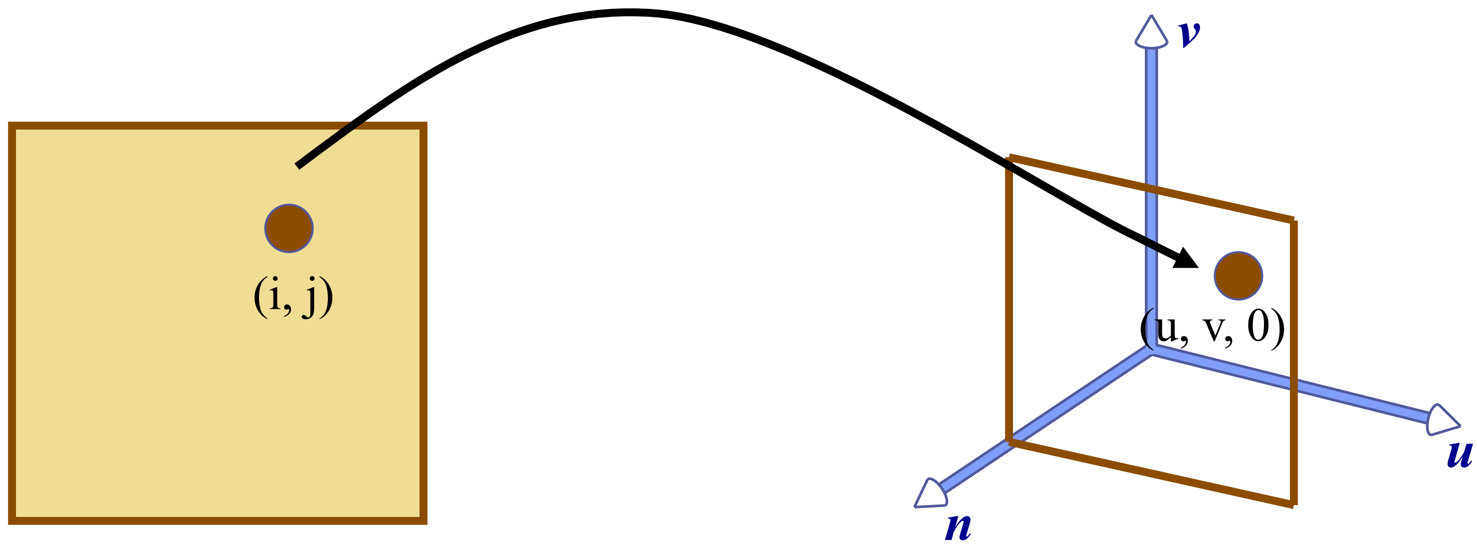
By Convention:
(x,y,z) = world coordinates
(i,j) = viewport coordinates
(u,v,n) = view-reference coordinates

Have *s* as an *(i,j)* (inside a dbl for loop)
Need *s* in *(x,y,z)* world coordinates
First, take *s* in *(i,j)* to *(u,v,n)* coordinates.
Then take those *(u,v,n)* coordinates to *(x,y,z)*



Computing Window Point

- Step 1: Reverse the Window-to-Viewport transformation



Viewport

View Reference coordinates

Viewport-Window transform

- Window-viewport:

$$i = (u - u_{\min}) \bullet \left(\frac{i_{\max} - i_{\min}}{u_{\max} - u_{\min}} \right) + i_{\min}$$

$$j = (v - v_{\min}) \bullet \left(\frac{j_{\max} - j_{\min}}{v_{\max} - v_{\min}} \right) + j_{\min}$$

- Inverse transform (viewport-window)

$$u = (i - i_{\min}) \bullet \left(\frac{u_{\max} - u_{\min}}{i_{\max} - i_{\min}} \right) + u_{\min}$$

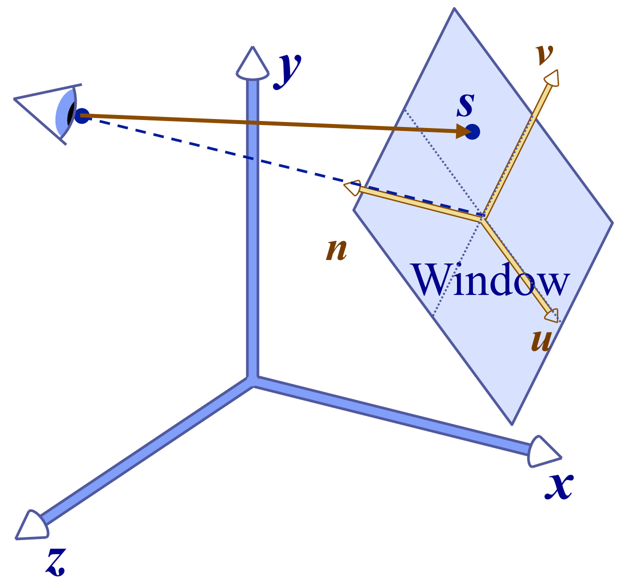
$$v = (j - j_{\min}) \bullet \left(\frac{v_{\max} - v_{\min}}{j_{\max} - j_{\min}} \right) + v_{\min}$$

$$n = 0$$

View-reference to World transform

- Given the screen point in terms of viewing-space coordinates (u, v, n), transform to world-space (x, y, z):
  - The viewing transform takes a point from world space to view space:

$$\mathbf{M}_v = \mathbf{RT} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -LookAt_x \\ 0 & 1 & 0 & -LookAt_y \\ 0 & 0 & 1 & -LookAt_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



  - We want to reverse this:

$$\mathbf{s} = \mathbf{M}_v^{-1} \begin{bmatrix} u_s \\ v_s \\ n_s \\ 1 \end{bmatrix} = \mathbf{T}^{-1}\mathbf{R}^{-1} \begin{bmatrix} u_s \\ v_s \\ n_s \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & v_x & n_x & LookAt_x \\ u_y & v_y & n_y & LookAt_y \\ u_z & v_z & n_z & LookAt_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_s \\ v_s \\ n_s \\ 1 \end{bmatrix}$$

or

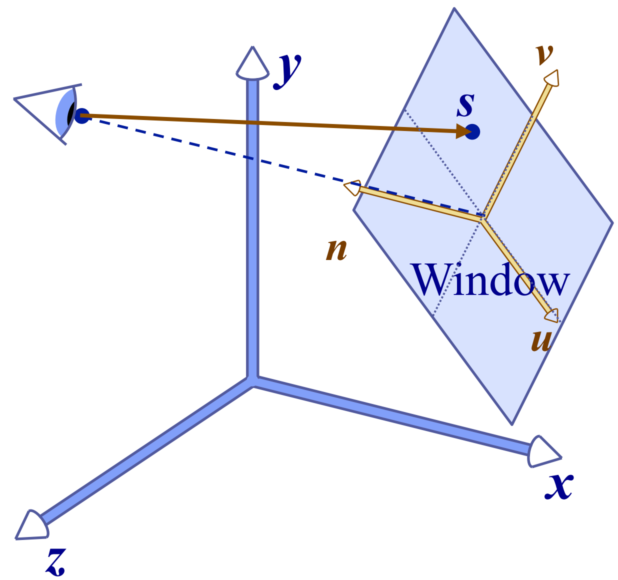
$$\boldsymbol{s} = \boldsymbol{LookAt} + u_s\boldsymbol{u} + v_s\boldsymbol{v} + n_s\boldsymbol{n}$$

Mv = world to view (given *x,y,z* return *u,v,n*)

$$\mathbf{M}_v = \mathbf{R}\mathbf{T} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -LookAt_x \\ 0 & 1 & 0 & -LookAt_y \\ 0 & 0 & 1 & -LookAt_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translate lookAt point to origin

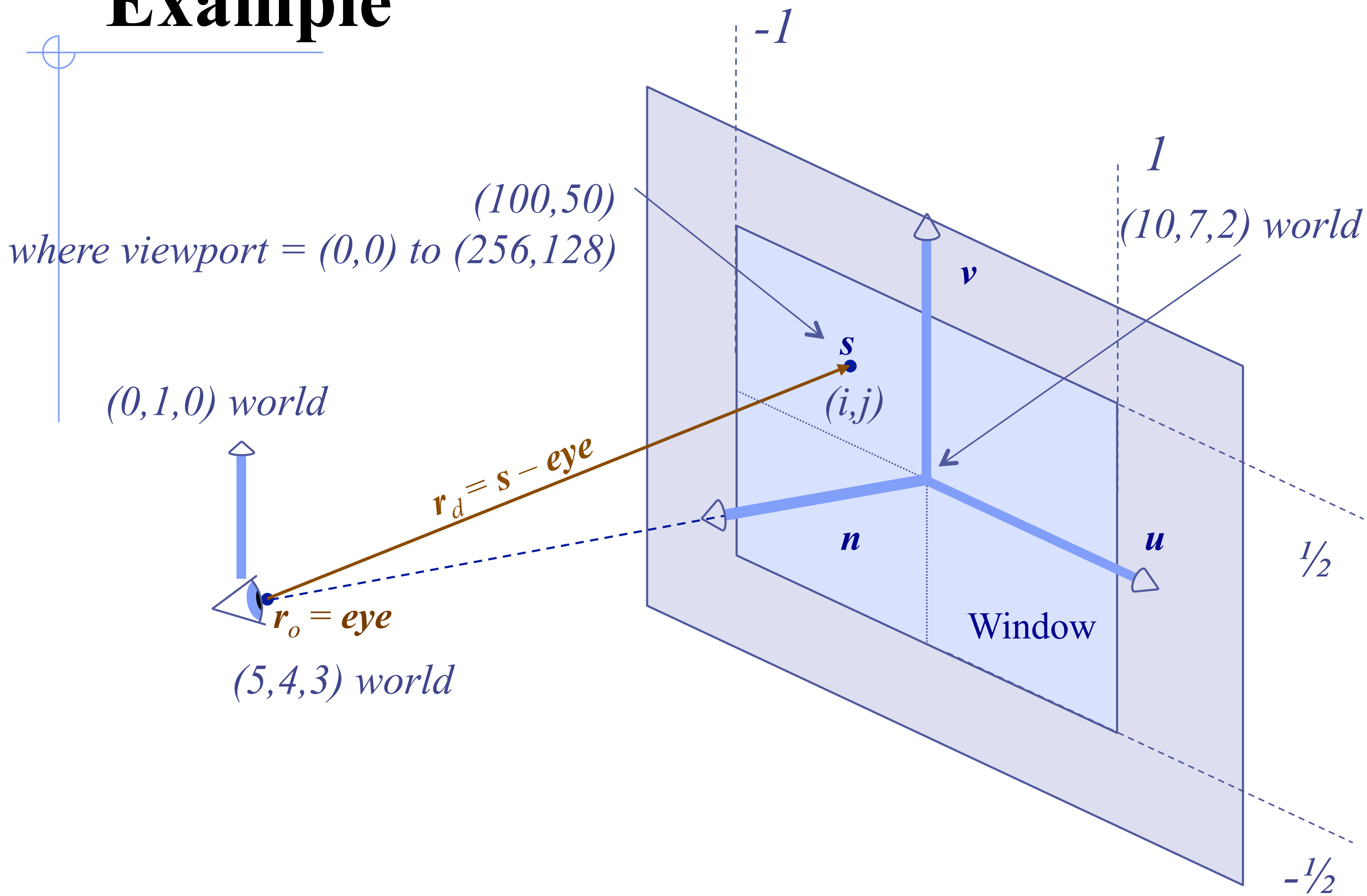
“Rotate” u,v,n axes to line up with x,y,z axes.



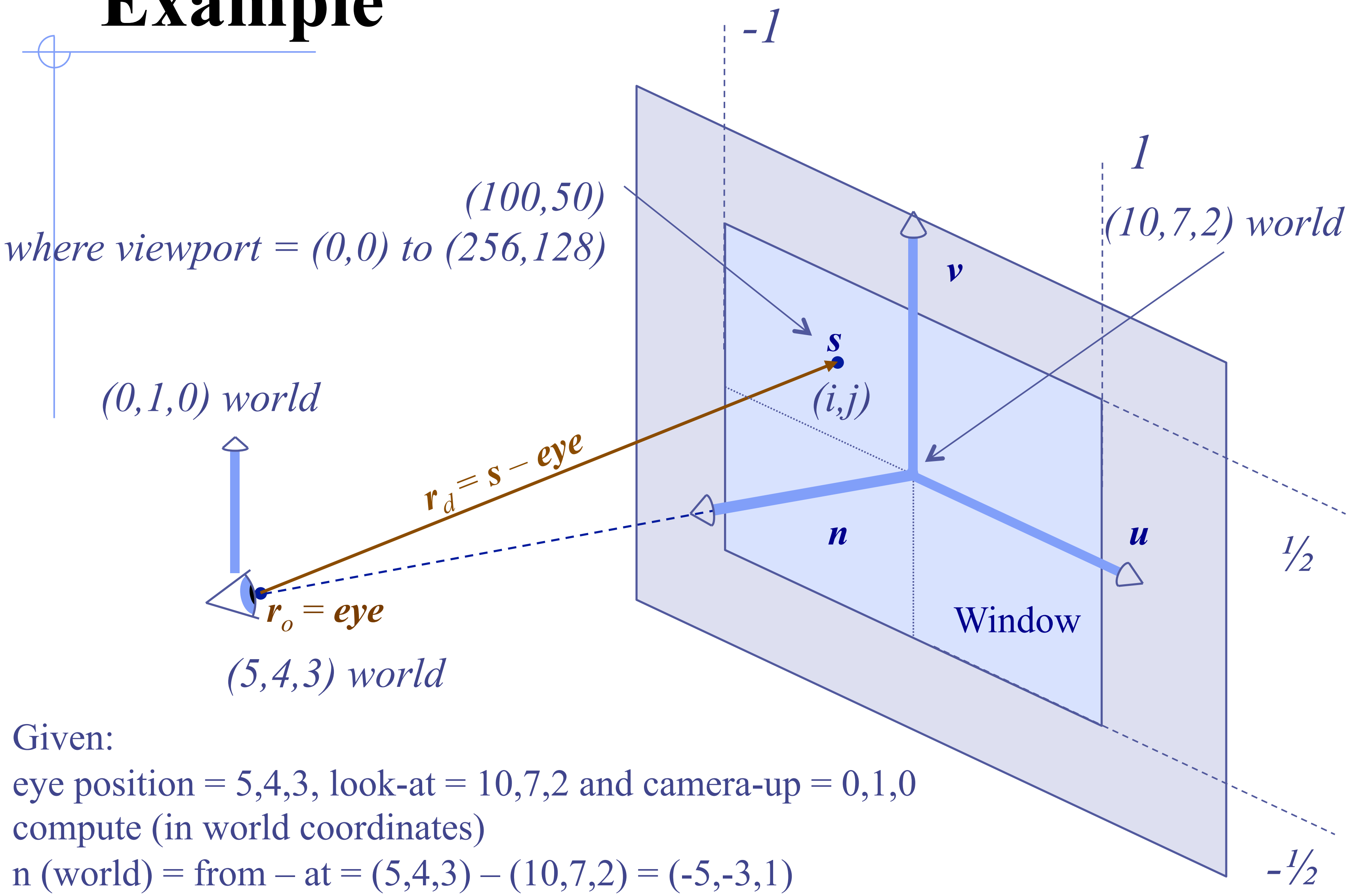
Mv-1 = view to world (given *u,v,n* return *x,y,z*)

$$\mathbf{M}_v^{-1} = \mathbf{T}^{-1}\mathbf{R}^{-1} = \begin{bmatrix} u_x & v_x & n_x & LookAt_x \\ u_y & v_y & n_y & LookAt_y \\ u_z & v_z & n_z & LookAt_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example


-1
1
(100,50)
where viewport = (0,0) to (256,128)
(10,7,2) world
v
s
(i,j)
(0,1,0) world
$r_d = s - eye$
n
u
½
Window
$r_o = eye$
(5,4,3) world
-½

Example



Given:

eye position = 5,4,3, look-at = 10,7,2 and camera-up = 0,1,0

compute (in world coordinates)

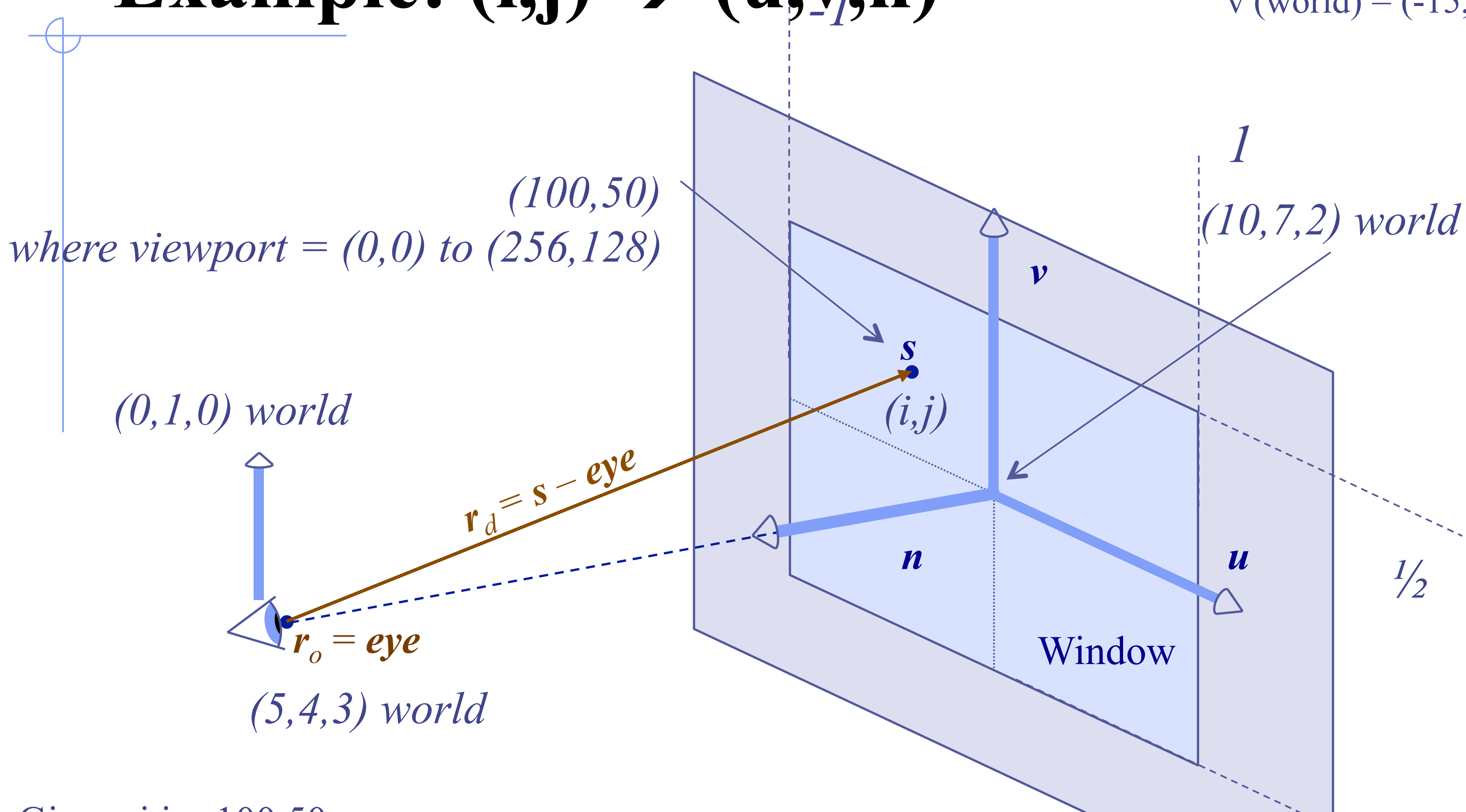
n (world) = from – at = (5,4,3) – (10,7,2) = (-5,-3,1)

u (world) = up x n = (0,1,0) x (-5,-3,1) = (1,0,5)

v (world) = n x u = (-5,-3,1) x (1,0,5) = (-15,26,3)

Example: (i,j) → (u,v,n)

n (world) = (-5,-3,1)
u (world) = (1,0,5)
v (world) = (-15,26,3)



$n = 0$

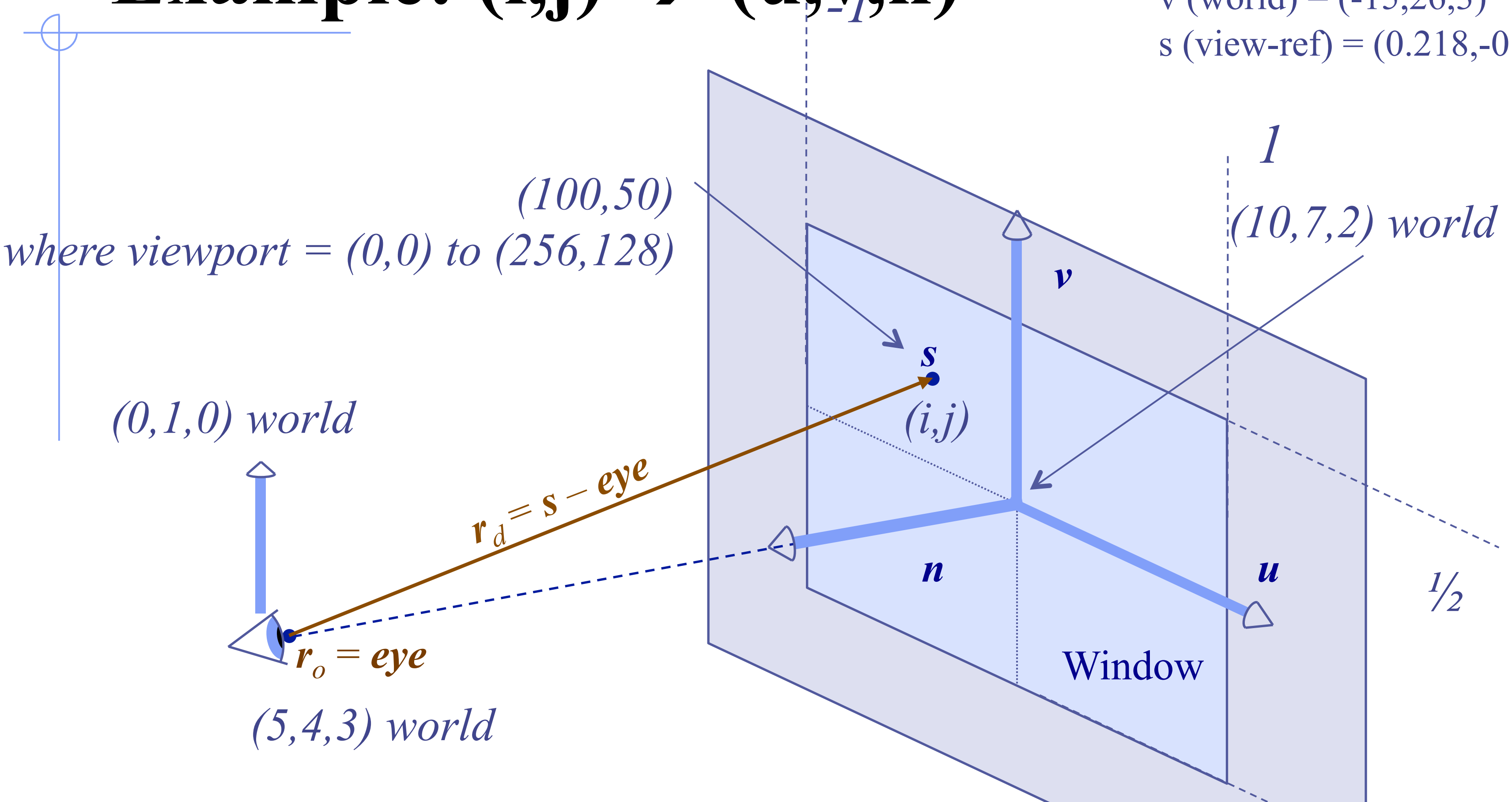
$$u = (i - i_{\min}) \bullet \left(\frac{u_{\max} - u_{\min}}{i_{\max} - i_{\min}} \right) + u_{\min}$$

$$v = (j - j_{\min}) \bullet \left(\frac{v_{\max} - v_{\min}}{j_{\max} - j_{\min}} \right) + v_{\min}$$

Given: i,j = 100,50
u = (100 – 256)x(1-(-1))/(256-0) -1 = 0.21875
v = (50 – 128) x (1/2 – (-1/2))/(128-0) – ½ = -0.152
n = 0

Example: (i,j) → (u,v,n)

n (world) = (-5,-3,1)
u (world) = (1,0,5)
v (world) = (-15,26,3)
s (view-ref) = (0.218,-0.152,0)



Compute s in world coordinates.

s(world) = M_v-1 s(view-ref)

$$\begin{bmatrix} 1 & -15 & -5 & 10 \\ 0 & 26 & -3 & 7 \\ 5 & 3 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.218 \\ -0.152 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 10.11 \\ 6.86 \\ 2.19 \\ 1 \end{bmatrix}$$