UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 4370 Fall 2023

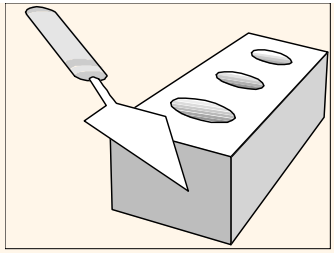
Interactive Computer Graphics

M & W 5:30 to 7:00 PM

Prof. Victoria Hilford

PLEASE TURN your webcam ON

NO CHATTING during LECTURE



COSC 4370

5:30 to 7

PLEASE

LOG IN

CANVAS

Please close all other windows.

11.15.2023 (W 5:30 to 7) (25)		**Lecture 13** **(Advanced Rendering)**
11.20.2023 (M 5:30 to 7) (26)		**EXAM 4 REVIEW**
11.27.2023 (M 5:30 to 7) (27)		**PROJECT 4**
11.29.2023 (W 5:30 to 7) (28)		**EXAM 4**
12.11.2023 (M 5:30 to 7)		**FINAL EXAM**

COSC 4370 – Computer Graphics

Lecture 13

Advanced Rendering
(NOT pipeline rendering)
Chapter 13

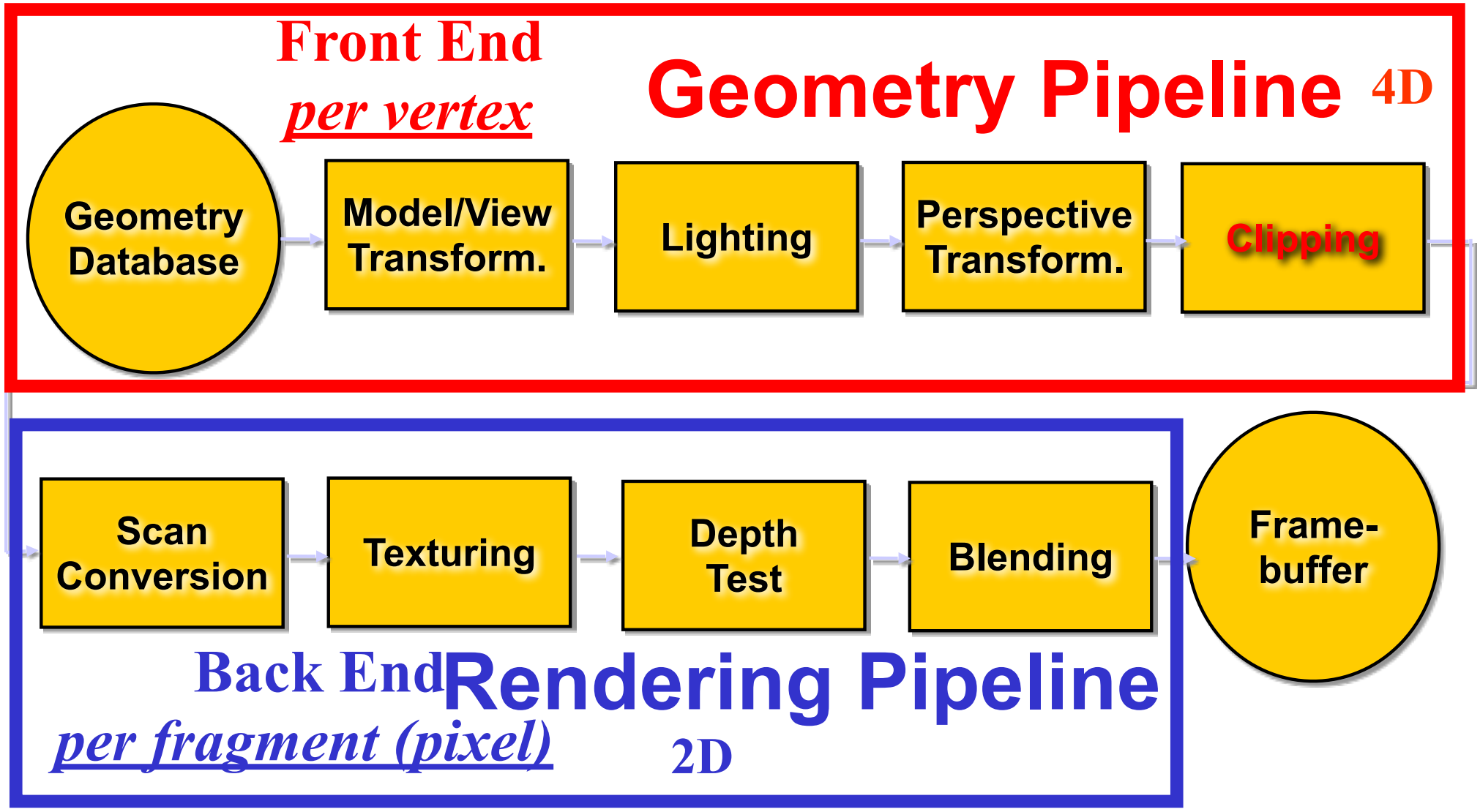
Advanced Rendering

- In this final chapter, we consider a **variety of alternative approaches to the** pipeline rendering **strategy supported by most hardware and software for interactive applications**. We have multiple motivations for introducing these other approaches. **We want to be able to incorporate effects, such as global lighting, that are not possible to render in real time.** We also want to produce high-quality images whose **resolution is beyond that of standard computer displays**. For example, a single frame of a computer-generated movie may contain over five million pixels and take hours to **render**.

- First, we explore ray tracers. We then look at the rendering equation that provides a physical basis for **global rendering**. Although we cannot solve the rendering equation in general, we will look at a special solution for a particular shading model that yields the **radiosity approach to rendering**.

The Graphics Pipeline

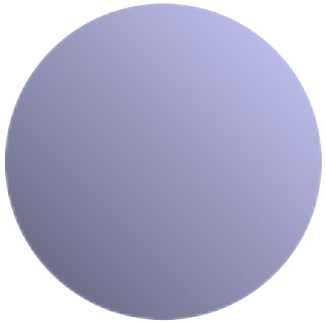
pipeline rendering - local lighting model



Why we need shading

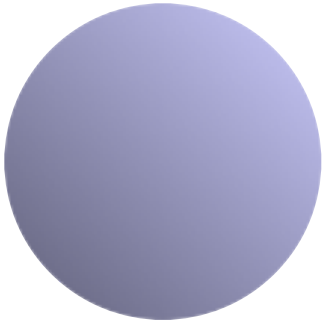
- Suppose we build a **model of a sphere** using **many polygons** and color it with `glColor`. We get something like

- **But we want**



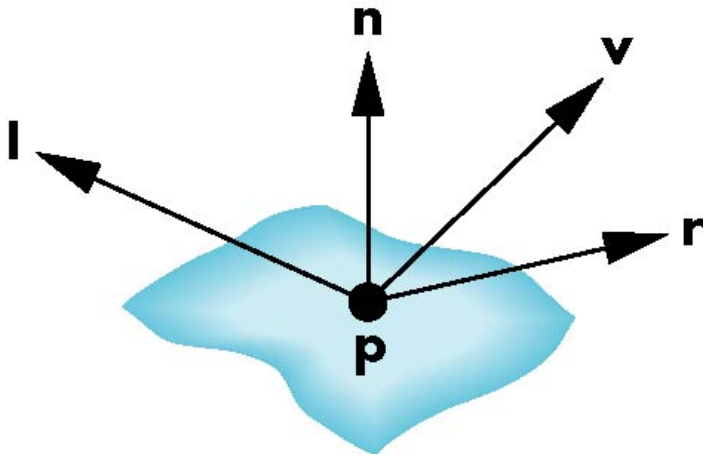
Shading

- Why does the **image** of a **real sphere looks like**



- **Light-material interactions** cause **each Point to have a different color or shade**

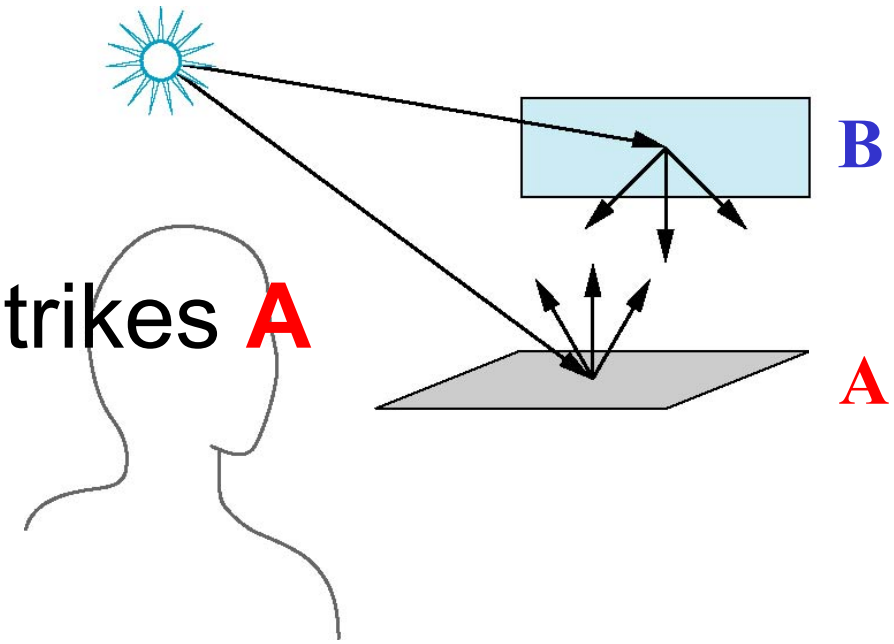
- Need to consider
  - **Light source(s) l**
  - **Material properties r**
  - **Location of viewer v**
  - **Surface orientation P → n**



Scattering

- **Light** strikes **A**
  - Some scattered
  - Some absorbed
- Some of scattered light strikes **B**
  - Some scattered
  - Some absorbed
- Some of this scattered light strikes **A**

and so on



Rendering Equation

The **infinite** scattering and absorption of light can be described by the ***Rendering Equation***

(**Bidirectional Reflection Distribution Function (BRDF**))

- **Cannot be solved in general**
- Ray tracing is a special case for **perfectly reflecting surfaces**

Rendering Equation is **Global** and includes

- **Shadows**
- **Multiple scattering from object to object**

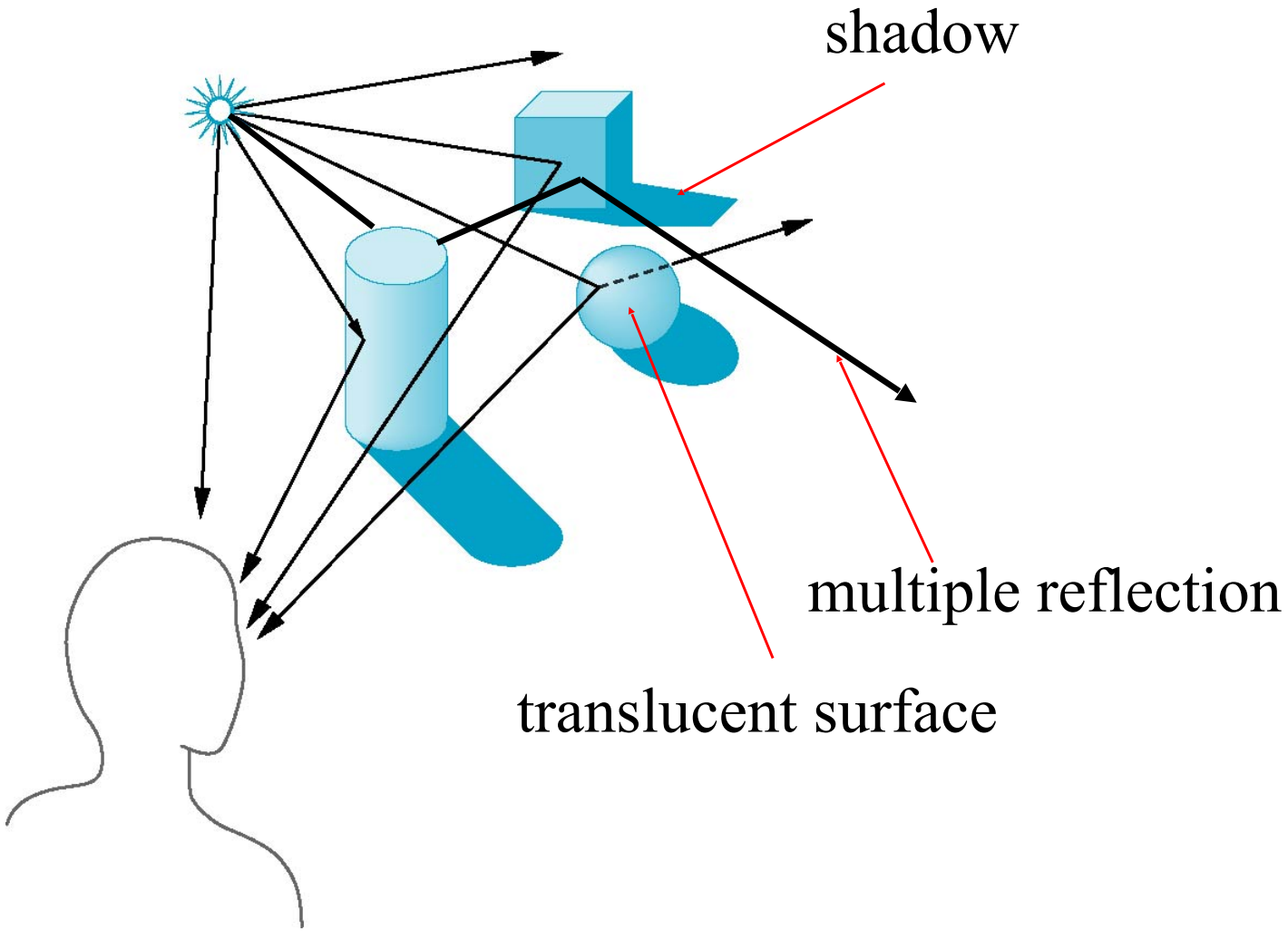
$$I_{out}(\Phi_{out}) = E(\Phi_{out}) + \int_{2\pi} R_{bd}(\Phi_{out}, \Phi_{in}) I_{in}(\Phi_{in}) \cos\theta \, d\omega$$

emission

bidirectional reflection coefficient

angle between normal and Φin

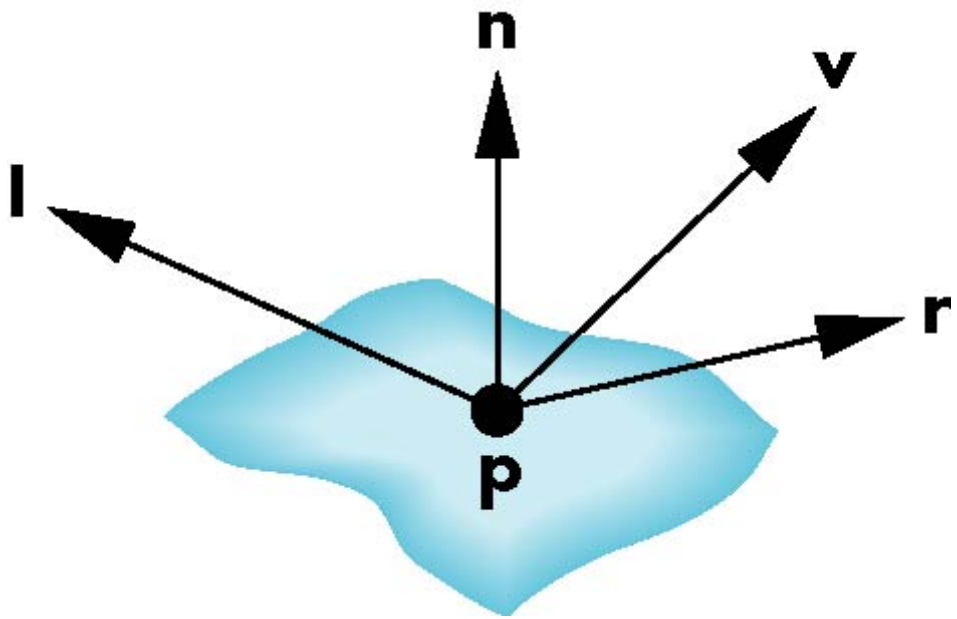
Global Effects
Ray Tracers



Phong Model

pipeline rendering - local lighting model

- **A simple Model that can be computed rapidly**
- Has three components
  - **Ambient**
  - **Diffuse**
  - **Specular**
- Uses four vectors
  - **To source l**
  - **To viewer v**
  - **Normal n**
  - **Perfect reflector r**



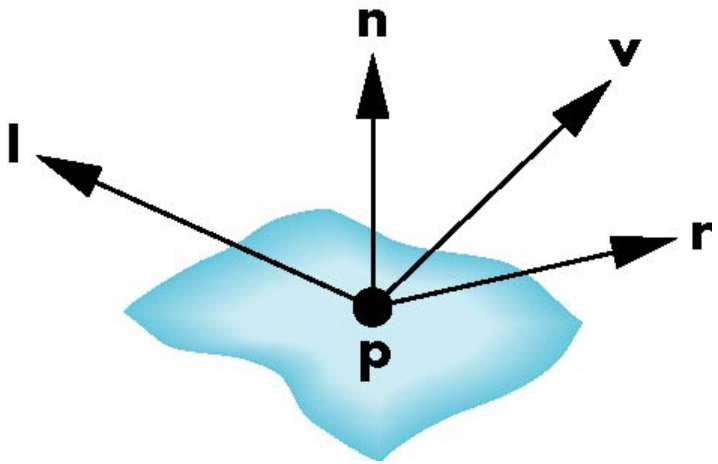
http://www.cs.princeton.edu/~min/cs426/classes/light.html

Adding up the Components

For **each light source** and **each color component**, the **Phong model** can be written (without the distance terms d) as

$$I = k_d I_d \; \mathbf{l} \bullet \mathbf{n} \; + k_s I_s (\mathbf{v} \bullet \mathbf{r})^{\alpha} + k_a I_a$$

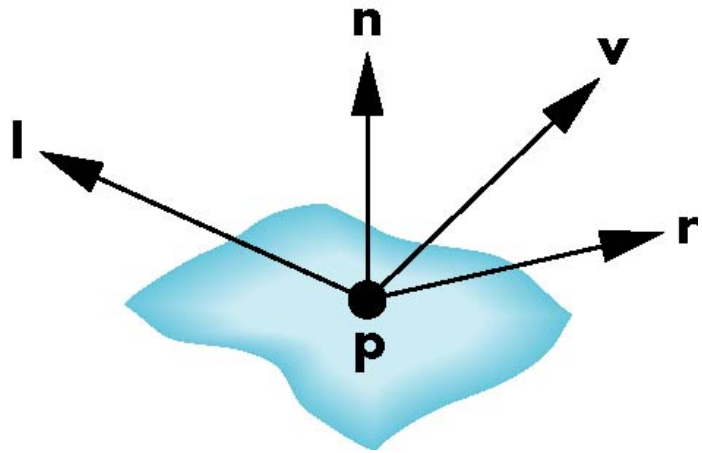
For each color component (RGB) we add contributions from all sources



Modified Phong - Blinn Model

pipeline rendering - local lighting model

The **specular term** in the **Phong Model** is problematic because it requires the calculation of a **new reflection vector r** and **view vector v** for **each vertex**

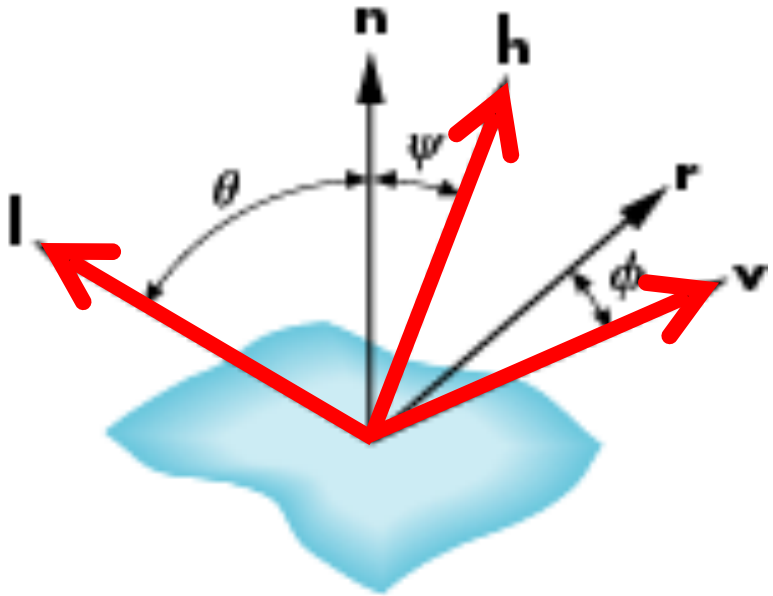


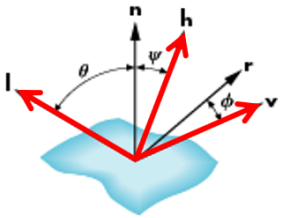
Blinn suggested an approximation using the **halfway vector** that is more efficient

The Halfway Vector h

- **h** is **normalized vector** halfway between **l** and **v**

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





Using the halfway vector h

- Replace $(\mathbf{v} \bullet \mathbf{r})^{\alpha}$ by $(\mathbf{n} \bullet \mathbf{h})^{\beta}$
- β is chosen to match shininess

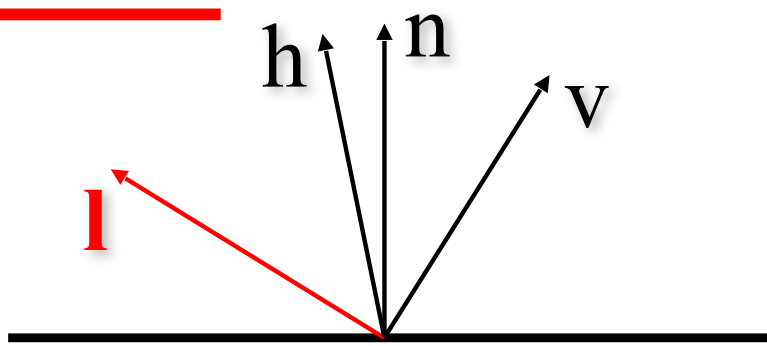
- Note that halfway angle is half of angle between **v** and **r** if vectors are **coplanar**

- Resulting model is known as the **modified Phong or Blinn lighting model**

  Specified in **OpenGL** standard

Reflection Equations

- **Blinn** improvement



- **full Phong lighting model**

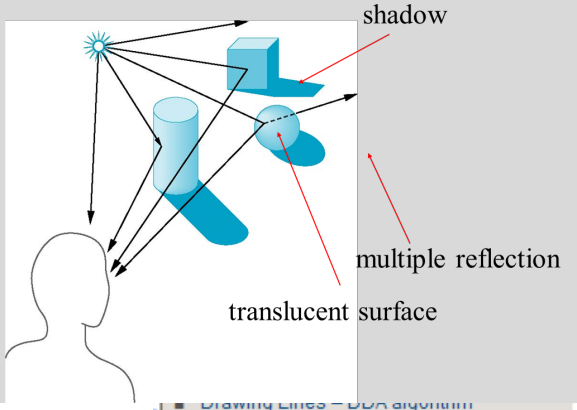
  combine ambient, diffuse, specular components

Local vs Global Rendering

Correct shading requires a Global calculation involving all Objects and light sources

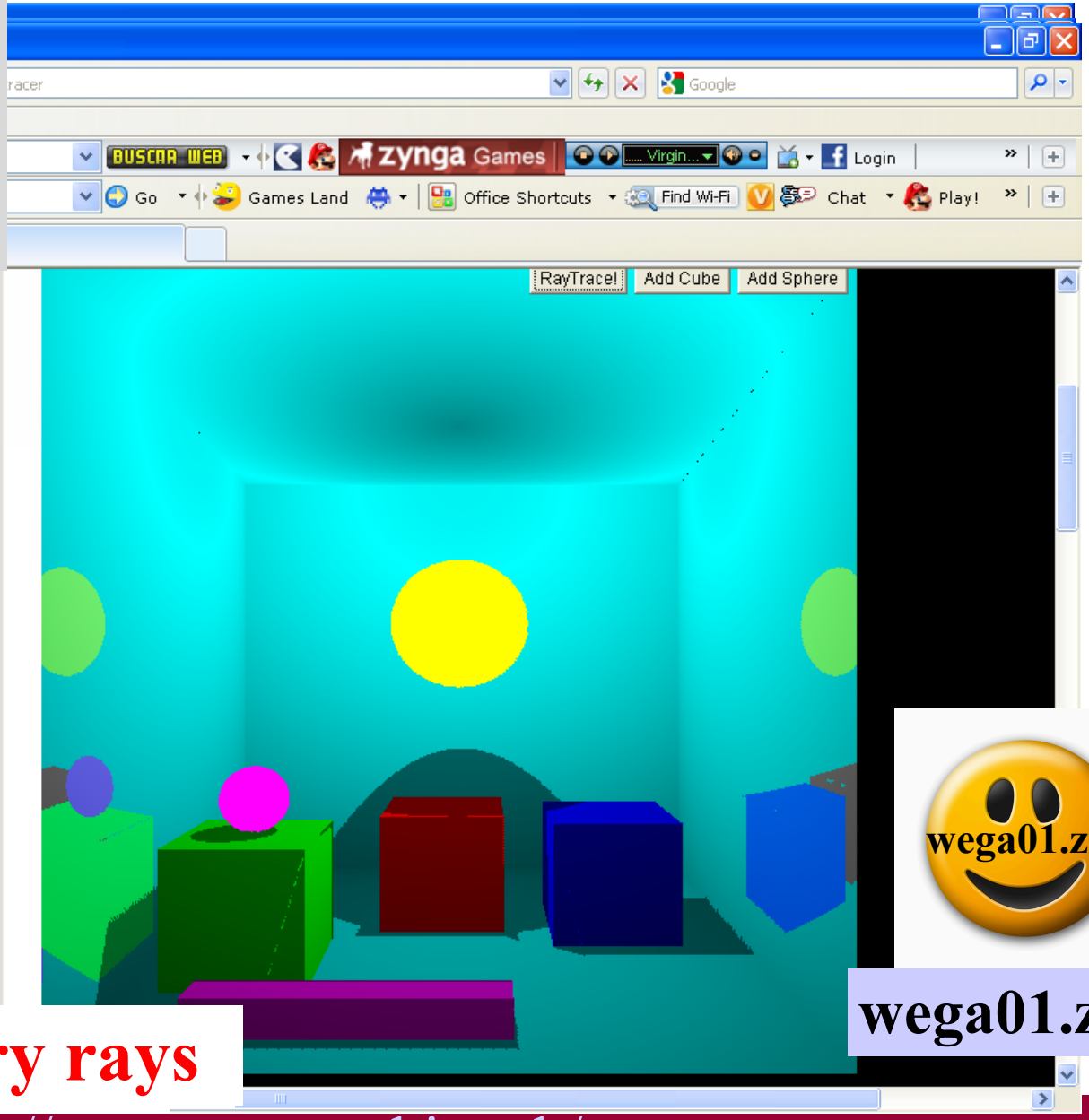
Incompatible with OpenGL pipeline model which shades each polygon independently (Local rendering)

Global Effects
Ray Tracers



Ray Tracing

- Drawing Lines – DDA algorithm 2
- Drawing Lines – The Bresenham algorithm
- Drawing Lines – The Bresenham algorithm 2
- Bresenham algorithm for a circle
- Bresenham algorithm for a circle 2
- Draw circle with method of transformation of rotation
- Draw line define point on circle
- Draw a circle with input raster value
- Drawing Ellipses – The Bresenham Algorithm
- Drawing Ellipse – The Bresenham Algorithm 2
- 4-connectivity 8-connectivity
- Simple wave recursive algorithms of filling areas
- FloodFill algorithm
- FloodFill algorithm 2
- Boundary fill algorithm
- Boundary fill algorithm 2
- Scan-line algorithm
- Fill polygon
- Clipping a point in a window
- Scan line polygon clipping



wega01.zip

Calculates tertiary rays

http://www.netgraphics.sk/ray-tracer

Lecture 13 Ray Tracer - NetBeans IDE 6.9.1

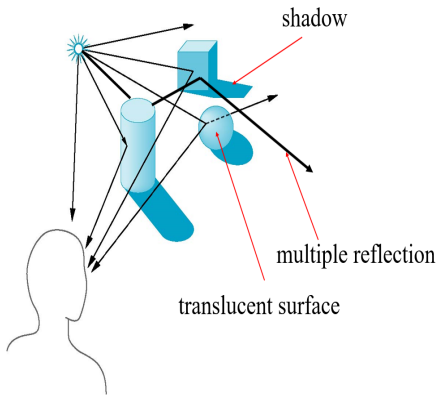
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

<default config>

Search (Ctrl+I)

Pr... Files Services

- Lecture 13 Ray Tracer
  - Source Packages
    - rtracer
      - Cube.java
      - Light.java
      - Line.java
      - MirrorRect.java
      - Rectangle.java
      - Sphere.java
      - Vector.java
      - rt.java
  - Test Packages
  - Libraries
  - Test Libraries



rt.java

```
13  public class rt extends Applet implements MouseListener, MouseMotionListener,
14          AdjustmentListener, ActionListener {
15
16          private static final long serialVersionUID = 1L;
17          static MainFrame mf;
18          public static double [] nul = {0,0,0};
19
20          public Cube      [] cubes      = new Cube      [4];
21          public Rectangle [] rectangles = new Rectangle[5];
22          public Sphere    [] spheres    = new Sphere    [2];
23
24          public Line cam;
25          public Light L1;
26
27          PixelMap [][] im;
28
29          Object obj, selObj;
30
31          Scrollbar [][] cubeScrolls   = new Scrollbar[4][3];
32          Scrollbar [][] sphereScrolls = new Scrollbar[3][3];
33          Label     [][] Label         = new Label    [4][4];
34          Button traceIt, addCube, addSphere, removeIt;
35
36          public Graphics g;
37          /**
38           * kontrola double bodu, ci je v imaginarnom svete vektorov
39           * @param point
40           * @return
41           */
42          public static boolean isOnWorld(double point) {
43                  if(point < 0 || point > Line.max) return false;
44                  return true;
```

Lecture 13 Ray Tracer (run) running... 1 | 1 INS

Lecture 13 Ray Tracer - NetBeans IDE 6.9.1

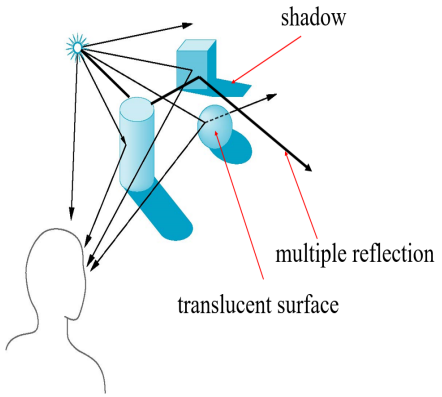
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

<default config>

Search (Ctrl+I)

Pr... Files Services

- Lecture 13 Ray Tracer
  - Source Packages
    - rtracer
      - Cube.java
      - Light.java
      - Line.java
      - MirrorRect.java
      - Rectangle.java
      - Sphere.java
      - Vector.java
      - rt.java
  - Test Packages
  - Libraries
  - Test Libraries



rt.java

```
235             */
236             public void drawWorld() {
237                     Line v;
238                     int _max = (int)(Line.max);
239                     g.setColor(new Color(0, 0, 255));
240                     for(int i = 0; i <= _max; i++) {
241                             //bottom side
242                             v = new Line(0,    0,    i, _max, 0, 0   );
243                             g.drawLine(v.X1, v.Y1, v.X2, v.Y2);
244                             v = new Line(i,    0,    0, 0,    0, _max);
245                             g.drawLine(v.X1, v.Y1, v.X2, v.Y2);
246                             //top side
247                             v = new Line(0,    _max, i, _max, 0, 0   );
248                             g.drawLine(v.X1, v.Y1, v.X2, v.Y2);
249                             v = new Line(i,    _max, 0, 0,    0, _max);
250                             g.drawLine(v.X1, v.Y1, v.X2, v.Y2);
251                             //left side
252                             v = new Line(0,    0,    i, 0, _max, 0   );
253                             g.drawLine(v.X1, v.Y1, v.X2, v.Y2);
254                             v = new Line(0,    i,    0, 0, 0,    _max);
255                             g.drawLine(v.X1, v.Y1, v.X2, v.Y2);
256                             //right side
257                             v = new Line(_max, 0,    i, 0, _max, 0   );
258                             g.drawLine(v.X1, v.Y1, v.X2, v.Y2);
259                             v = new Line(_max, i,    0, 0, 0,    _max);
260                             g.drawLine(v.X1, v.Y1, v.X2, v.Y2);
261                     }
                }
263             /**
264              * @param minY
265              * @param maxY
266              * @param minX
```

Lecture 13 Ray Tracer (run) running... 262 | 10 INS

Lecture 13 Ray Tracer - NetBeans IDE 6.9.1

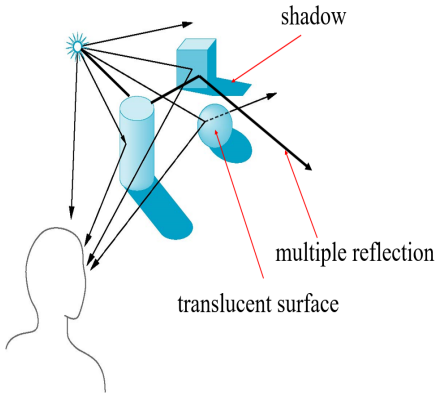
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

<default config>

Search (Ctrl+I)

Pr... Files Services

- Lecture 13 Ray Tracer
  - Source Packages
    - rtracer
      - Cube.java
      - Light.java
      - Line.java
      - MirrorRect.java
      - Rectangle.java
      - Sphere.java
      - Vector.java
      - rt.java
  - Test Packages
  - Libraries
  - Test Libraries



rt.java

```
487                                             new Rectangle[cubes.length][3];
488                         cubes[i].setVisibleSides(((MirrorRect)r).refl);
489                 ((MirrorRect)r).reflSides[i]   = cubes[i].visibleSides;
490                                 }
491                         }
492                         cubes[i].visibleSides = vs;
493                 }
494         }
495
496         /**
497          * inicializacne vytvorenie svetla, sveta, kocik a gul
498          */
          public void init() {
500             double [] tmp = {Line.max/2, Line.max/2, -Line.max/2};
501             double [] tmp2 = {Line.max/2, Line.max, Line.max/2};
502             cam = new Line(tmp, null);
503
504             L1 = new Light(new Line(tmp2, null));
505
506             generateRects();
507             generateCubes();
508             generateSpheres();
509
510             addMouseListener( this );
511             addMouseMotionListener( this );
          }
513         /**
514          * vykreslenie bud nakresu sveta, alebo jeho vizualu akolany rtrace
515          */
          public void paint( Graphics g ) {
517             this.g = g;
518             g.setColor(new Color(0, 0, 0));
```

Camera

Light

Objects

Lecture 13 Ray Tracer - NetBeans IDE 6.9.1

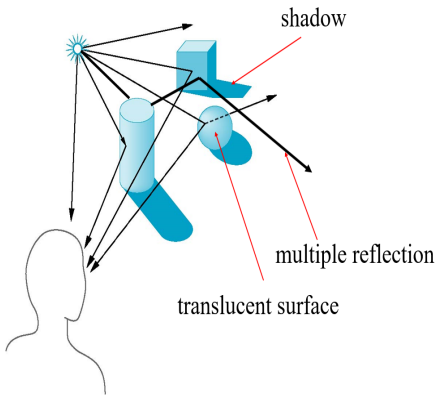
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

<default config>

Search (Ctrl+I)

Pr... Files Services

- Lecture 13 Ray Tracer
  - Source Packages
    - rtracer
      - Cube.java
      - Light.java
      - Line.java
      - MirrorRect.java
      - Rectangle.java
      - Sphere.java
      - Vector.java
      - rt.java
  - Test Packages
  - Libraries
  - Test Libraries



rt.java

```
557         public void rtrace() {
558                 this.g = getGraphics();
559                 im = new PixelMap[(int)Line.wLength+1][(int)Line.wLength+1];
560                 double [] tmp = {0, 0, 0};
561                 g.setColor(new Color(0, 0, 0));
562                 g.fillRect(0, 0, 850, 650);
563                 for(double k = 0.1; k >= 0.033; k-=0.033) {
564                         for(tmp[0] = 0; tmp[0] < Line.max; tmp[0]+=k) {
565                                 for(tmp[1] = 0; tmp[1] < Line.max; tmp[1]+=k) {
566                                         fireRay(cam.pointTo(tmp));
567                                 }
568                         }
569                 }
570         }
571         /**
572          * samotne pokusy luca o dopad na niektoru z viditelnych stran kociek,
573          * do oblasti gule alebo do oblasti stvorcov ohranicujucich svet
574          * @param ray
575          */
576         public void fireRay(Line ray) {
577                 for(Cube c: cubes)
578                         for(Rectangle r: c.visibleSides)
579                                 fireRayToRect(ray, r, c);
580
581                 for(Sphere r: spheres)
582                         fireRayToSphere(ray, r);
583
584                 for(Rectangle r: rectangles)
585                         fireRayToRect(ray, r, r);
586
587                 return;
            }
```

ray to Cube

ray to Spher

ray to Rectan

Lecture 13 Ray Tracer - NetBeans IDE 6.9.1

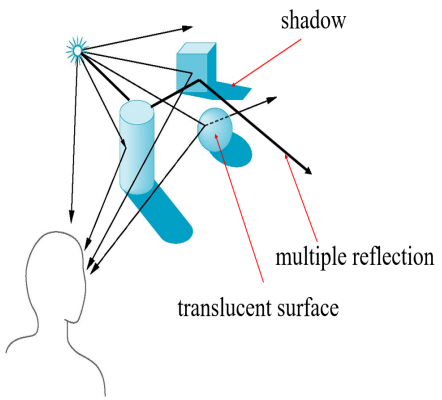
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

<default config>

Search (Ctrl+I)

Pr... Files Services

- Lecture 13 Ray Tracer
  - Source Packages
    - rtracer
      - Cube.java
      - Light.java
      - Line.java
      - MirrorRect.java
      - Rectangle.java
      - Sphere.java
      - Vector.java
      - rt.java
  - Test Packages
  - Libraries
  - Test Libraries



rt.java

```
641                 g.drawLine(ray.X2, ray.Y2, ray.X2, ray.Y2);
642         }
643         /**
644          * test luca ray vzhladom na gulu r
645          * @param ray
646          * @param r
647          */
648         public void fireRayToSphere(Line ray, Sphere r) {
649                 double [] t = r.getIntersectionPoint(ray); //existuje
650                 if(t == null) return;
651                 ray = ray.pointTo(t);
652                 if(im[ray.Y2][ray.X2] == null) im[ray.Y2][ray.X2] =
653                         new PixelMap(ray.X2, ray.Y2, ray.getSize());
654                 else if(im[ray.Y2][ray.X2].depth < ray.getSize()) return;
655                 im[ray.Y2][ray.X2].depth = ray.getSize();
656                 im[ray.Y2][ray.X2].setColor(r.c1RGB[0], r.c1RGB[1], r.c1RGB[2]);
657                 im[ray.Y2][ray.X2].o = r;
658                 g.setColor(new Color(r.c1RGB[0], r.c1RGB[1], r.c1RGB[2]));
659                 g.drawLine(ray.X2, ray.Y2, ray.X2, ray.Y2);
660         }
661
662         public static void main(String[] args) {
663                 mf = new MainFrame(new rt(), 850, 590);
664         }
665
666         /**
667          * aplikacie zmien pri posune nejakeho scrollbaru nastaveni objektu
668          * (funkcie snad pochopitelne z nazvov metod)
669          */
          @Override
671         public void adjustmentValueChanged(AdjustmentEvent e) {
672                 int minY = 0;
```

ray to Cube

Lecture 13 Ray Tracer - NetBeans IDE 6.9.1

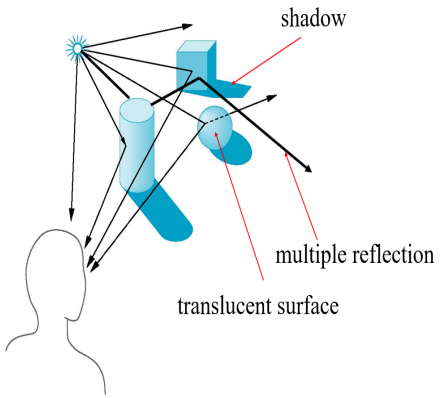
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

<default config>

Search (Ctrl+I)

Pr... Files Services

- Lecture 13 Ray Tracer
  - Source Packages
    - rtracer
      - Cube.java
      - Light.java
      - Line.java
      - MirrorRect.java
      - Rectangle.java
      - Sphere.java
      - Vector.java
      - rt.java
  - Test Packages
  - Libraries
  - Test Libraries



rt.java | Rectangle.java

```
595         public void fireRayToRect(Line ray, Rectangle r, Object o) {
596             double [] point = r.getIntersectionPoint(ray); //surad n
597
598             if(!isOnWorld(point[0]) || !isOnWorld(point[1]) ||
599                     !isOnWorld(point[2])) return; // ak je mimo svetc
600             ray = ray.pointTo(point); // samotne predlzenie vodu
601
602             if(!isOnWorld(ray.X2) || !isOnWorld(ray.Y2)) return; // il
603             if(!Vector.isPointOnPolygon(ray.Y2, ray.X2, r.Y, r.X)) return;
604             if(r.c1RGB[0] == 0 && r.c1RGB[1] == 0 && r.c1RGB[2] == 0) return;
605
606             if(im[ray.Y2][ray.X2] == null) im[ray.Y2][ray.X2] =
607                     new PixelMap(ray.X2, ray.Y2, ray.getSize());
608             else if(im[ray.Y2][ray.X2].depth < ray.getSize()) return; //
609
610             im[ray.Y2][ray.X2].depth = ray.getSize(); // doplnenie novej
611
612             double [] E = Vector.norm(ray.v);
613             double S = Vector.scalarProduct(E, r.N) * (-2);
614             Line u = new Line(ray.getP2X(), ray.getP2Y(), ray.getP2Z(),
615                     (S*r.N[0])+E[0], (S*r.N[1])+E[1], (S*r.N[2])+E[2]);
616             // podla nejakeho vzorca najdeneho googlom zo storca
617
618             L1.lightRay = L1.lightRay.pointTo(ray.getP2()); // vna
619             int size = (int)Math.abs((L1.lightRay.getSize()-20)*6);
620             // kedze sa vzdialenost od lampy pohybovala vzhladom k s
621             // takouto divnou rovnicou clovek najde pre mna optimalial
622             int [] nC = {r.c1RGB[0] - size, r.c1RGB[1] - size,
623             r.c1RGB[2] - size};
624             int [] okColor = null;
625             if(r instanceof MirrorRect)
                okColor = ((MirrorRect)r).controlReflection(u, spheres, nC);
```

ray to Rec

nor

refle

Lecture 13 Ray Tracer - NetBeans IDE 6.9.1

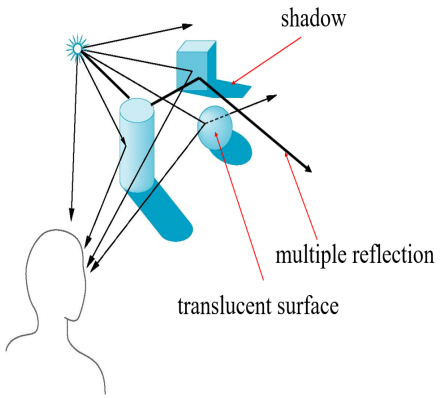
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

<default config>

Search (Ctrl+I)

Pr... Files Services

- Lecture 13 Ray Tracer
  - Source Packages
    - rtracer
      - Cube.java
      - Light.java
      - Line.java
      - MirrorRect.java
      - Rectangle.java
      - Sphere.java
      - Vector.java
      - rt.java
  - Test Packages
  - Libraries
  - Test Libraries



rt.java | Rectangle.java

```
125          */
126         public double getDistanceFrom(double  x, double y, double z) {
127                 return (N[0]*(x-midPoint[0]) + N[1]*(y-midPoint[1]) +
128                         N[2]*(z-midPoint[2])) / Math.sqrt(N[0]*N[0] +
129                         N[1]*N[1] + N[2]*N[2]);
130         }
131         public double getDistanceFrom(double [] w) {
132                 return getDistanceFrom(w[0], w[1], w[2]);
133         }
134         public double getDistanceFrom(Line w) {
135                 return getDistanceFrom(w.X, w.Y, w.Z);
136         }
137         /**
138          * vypocet suradnic bodu po predlzeni do roviny efinovanej stvorcom
139          * @param w
140          * @return
141          */
142         public double [] getIntersectionPoint(Line w) {
143                 double t = (N[0]*(midPoint[0]-w.X) + N[1]*(midPoint[1]-w.Y) +inters
144                         N[2]*(midPoint[2]-w.Z)) / (N[0]*w.v.X + N[1]*w.v.Y +
145                         N[2]*w.v.Z);
146                 double [] res = {w.X + t*w.v.X, w.Y + t*w.v.Y, w.Z + t*w.v.Z};
147                 return res;
148         }
149 }
150
```

Introduction

pipeline rendering - local lighting model

- OpenGL is based on a **pipeline model** in which primitives are **rendered** one at time
  - No shadows (except by tricks or **multiple renderings**)
  - No multiple reflections

- **Global** approaches
  - Rendering equation
  - **Ray tracing**
  - **Radiosity**

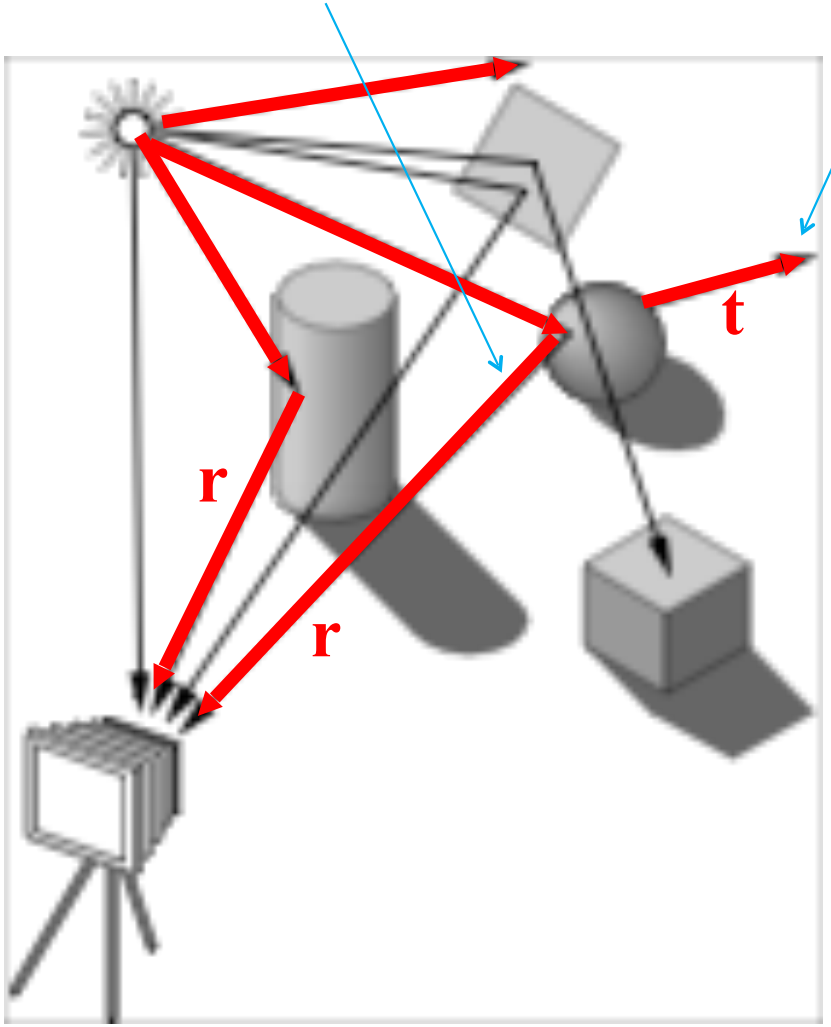
Introduction

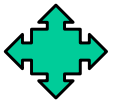
- OpenGL is based on a pipeline model in which primitives are rendered one at time
  - No shadows (except by tricks or multiple renderings)
  - No multiple reflections

- **Global** approaches
  - Rendering equation
  - **Ray tracing**
  - **Radiosity**

Ray Tracing

- Follow rays of light from a point source
- Can account for *reflection* and *transmission*





Computation

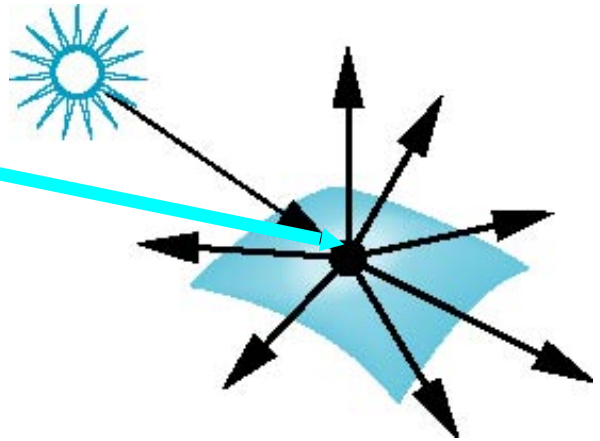
Should be able to **handle all physical interactions**

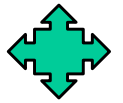
Ray tracing paradigm **is not computational**

Most rays do not affect what we see

Scattering produces many (infinite) **additional rays**

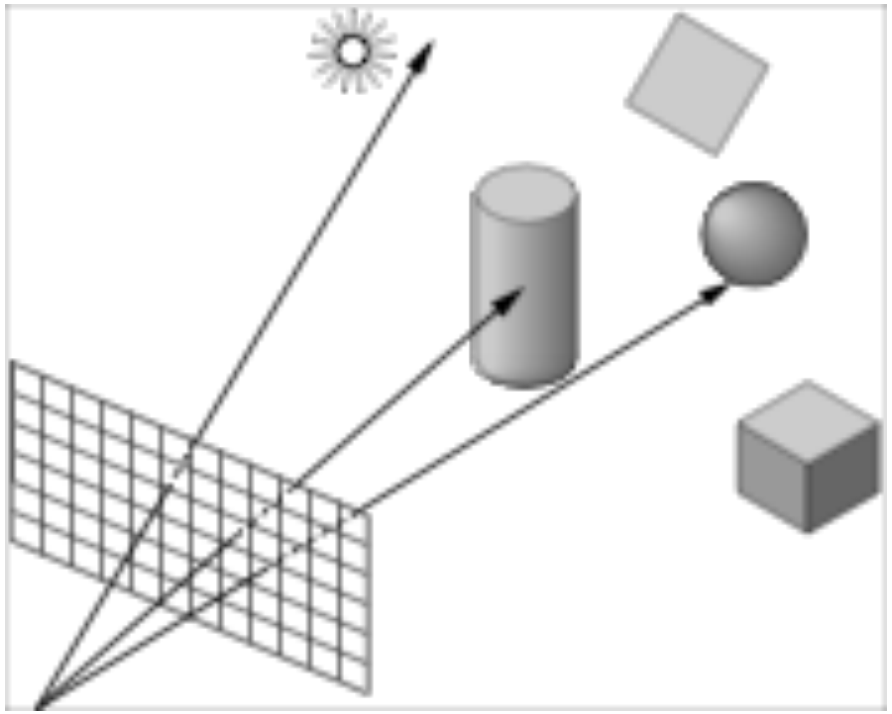
Alternative: **ray casting**



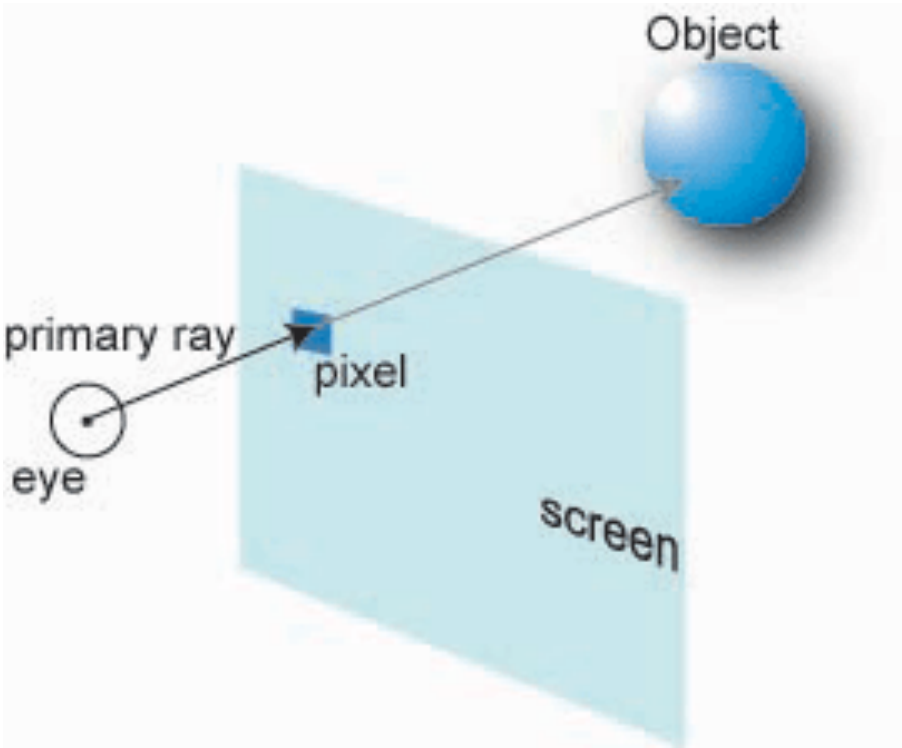


Ray Casting

- Only rays that reach the eye COP matter
- Reverse direction and cast rays
- Need at least one ray per pixel



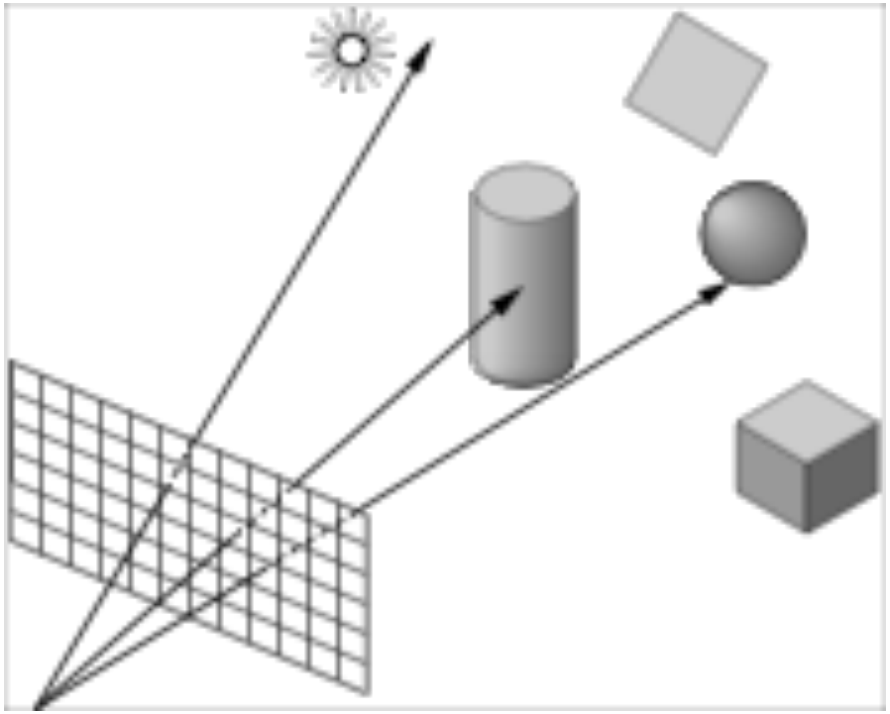
COP

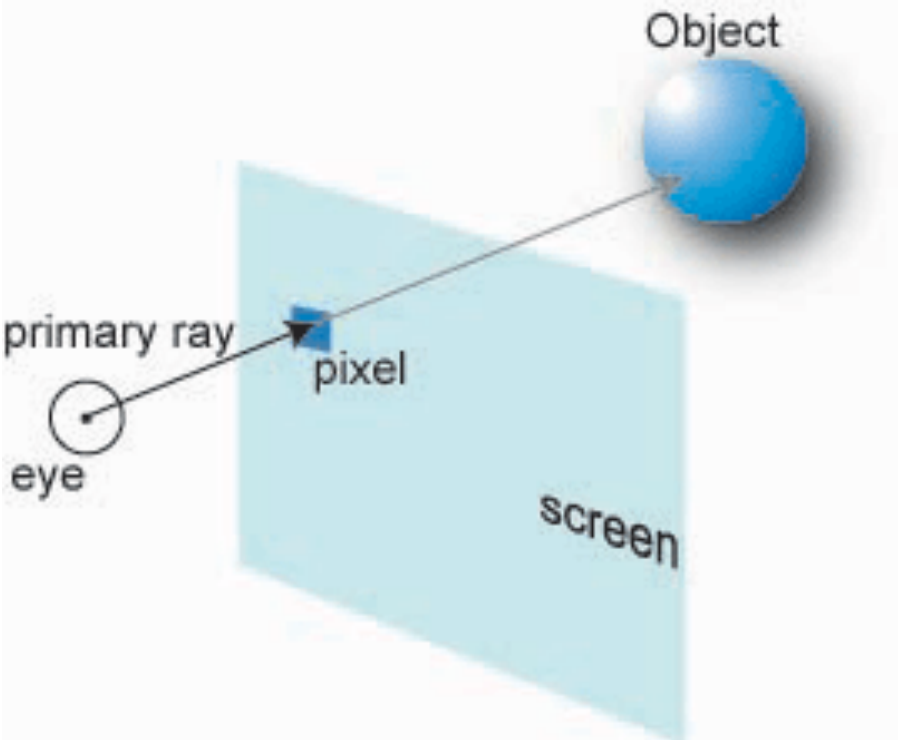


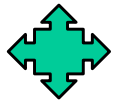
Ray Casting

Note that the process that we have described thus far requires all the same steps as we use in our pipeline renderer: object modeling, projection, and visible-surface determination.

However, the order in which the calculations are carried out is different. The pipeline renderer works on a vertex-by-vertex basis; the ray caster works on a pixel-by-pixel basis.





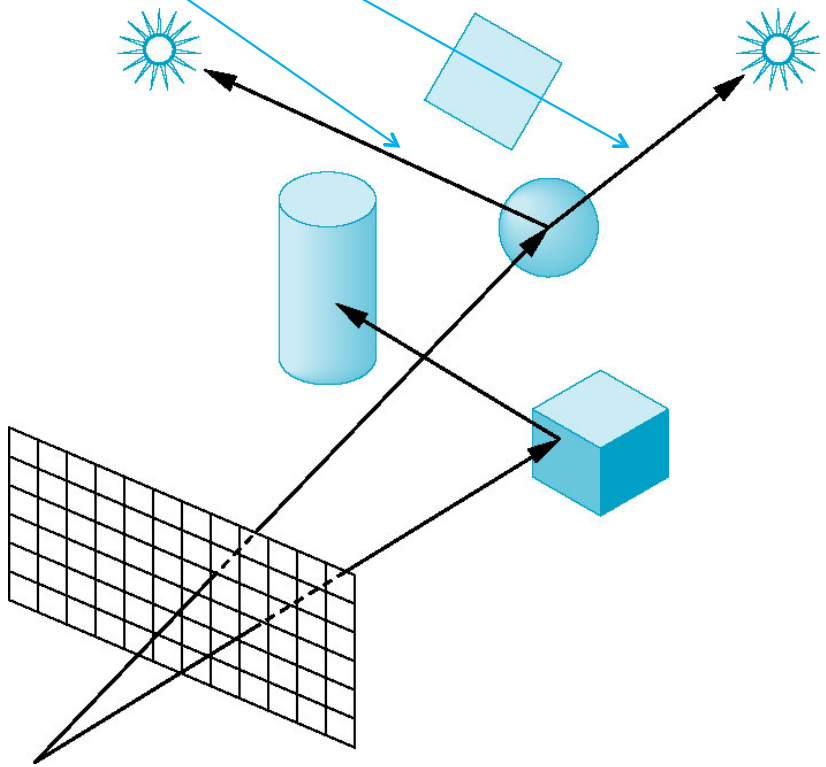


Ray Casting

Compute shadow, or feeler, rays **from the** Point **on the surface** to each **source.**

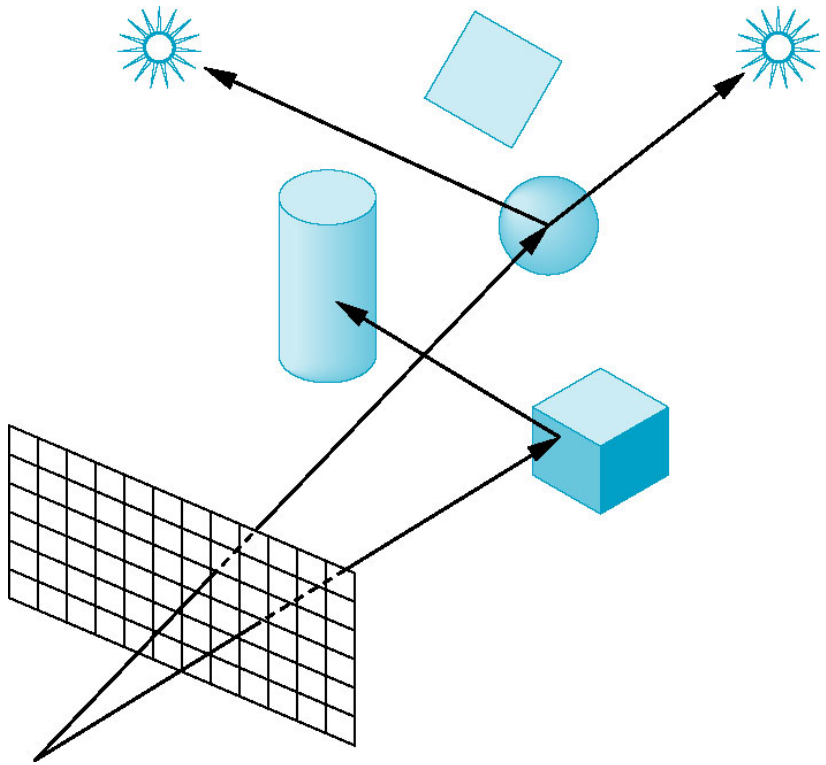
If a **shadow ray intersects** a **surface** before it meets the **source**, the light is blocked from reaching the Point under consideration, and this Point is in shadow, at least from this **source**.

No lighting calculation needs to be done for sources that are blocked from a Point on the **surface**.



Ray Casting

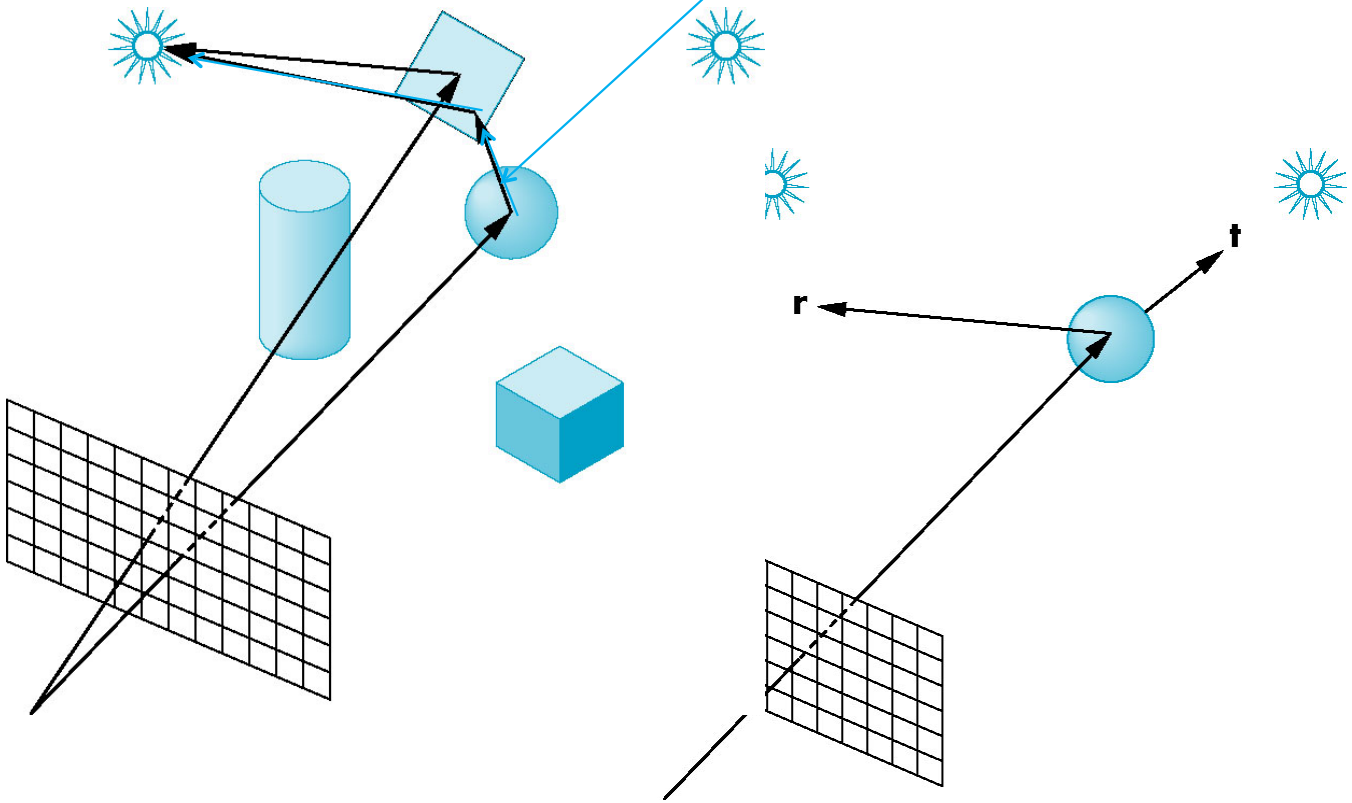
If all surfaces are opaque and we do not consider light scattered from surface to surface, we have an image that has shadows added to what we have already done without ray tracing. The price we pay is the cost of doing a type of hidden-surface calculation for each Point of intersection between a cast ray and a surface.



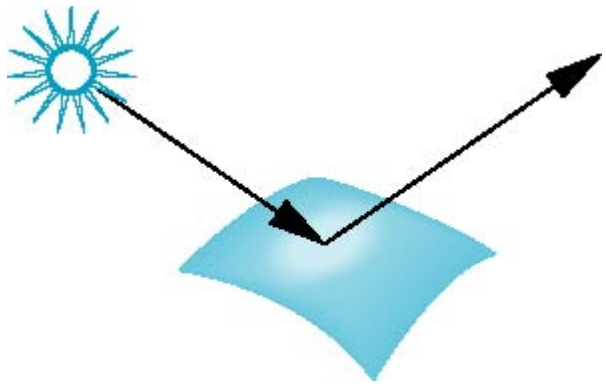
Ray Casting

Suppose that some of our surfaces are highly reflective.

We can follow the shadow ray as it bounces from surface to surface, until it either goes off to infinity or intersects a source. Such calculations are usually done recursively and take into account any absorption of light at surfaces.



Specular reflections



smooth surface

Ray Casting

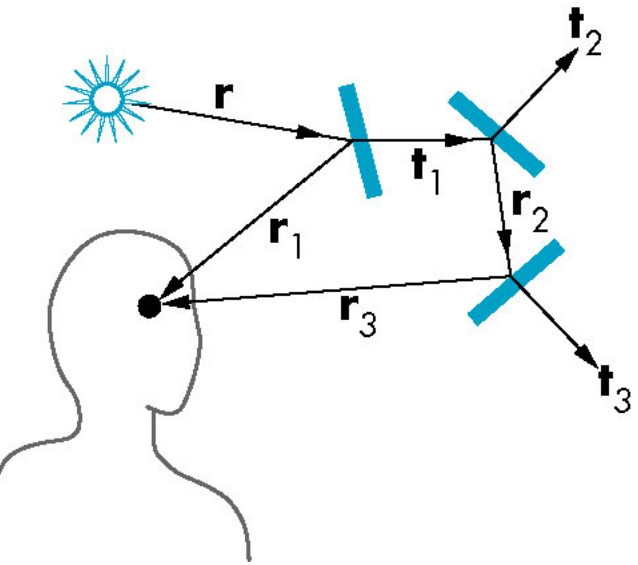
From the perspective of the **cast ray**, if a light **source** is visible at the **intersection** Point, then we need to do three tasks:

First, we must compute the contribution from the light source at the Point, using our **standard reflection model**.

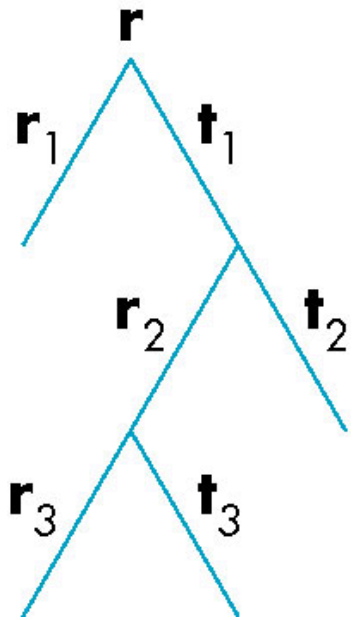
Second, we must cast a ray in the direction of a **perfect reflection**.

Third, we must cast a ray in the direction of the **transmitted ray**.

These two cast rays are treated just like the original cast ray; that is, they may intersect other surfaces, they can end at a source, or they can go off to infinity. At each surface that these rays intersect, additional rays may be generated by reflection and transmission of light.



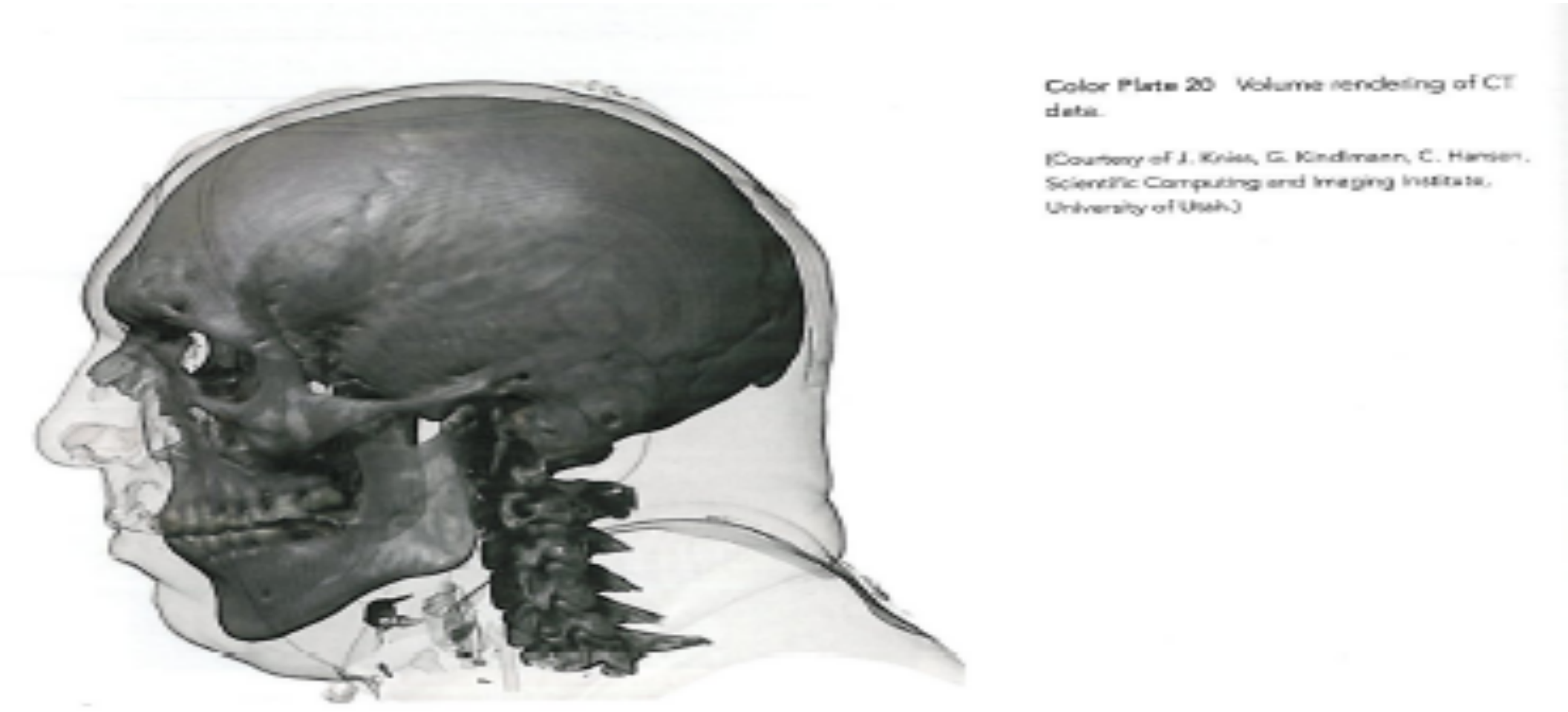
Ray Tree



Ray Casting

Ray Casting of three dimensional data sets, **shadow rays** were not followed.

By making the **skin transparent** and the **muscle opaque**, however, the **Ray Casting** paradigm was able to create an image of only the latter.

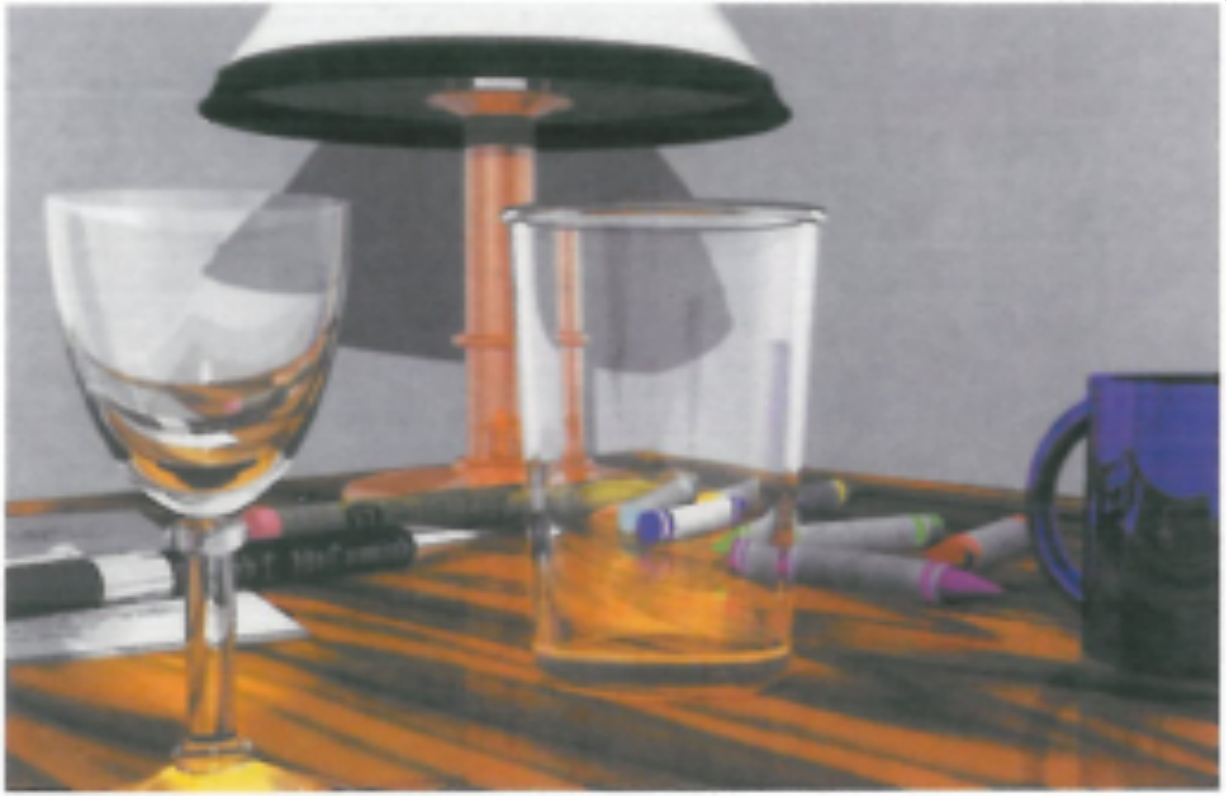


Color Plate 20 Volume rendering of CT data.

(Courtesy of J. Kniss, G. Kindlmann, C. Hansen, Scientific Computing and Imaging Institute, University of Utah.)

Ray Casting

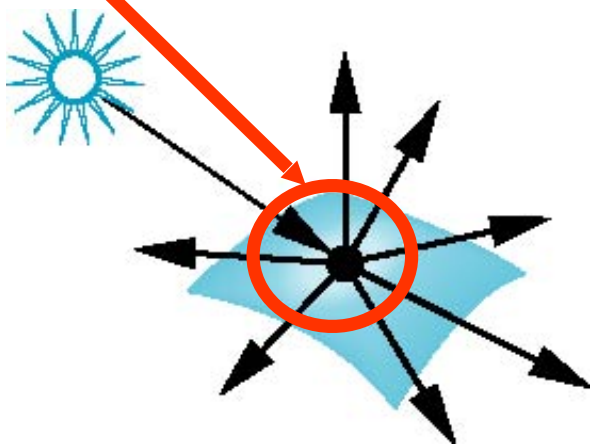
Although our **Ray Caster** uses the **Blinn-Phong model** to include a **diffuse term** at the **Point of intersection between a ray and a surface**, the light that is **scattered diffusely at this Point is ignored**. If we were to attempt to follow such light, we would **have so many rays** to deal with that the **Ray Caster** might never complete execution.



Color Plate 23 Rendering using ray tracer.

(Courtesy of Patrick McCormick, University of New Mexico and Los Alamos National Laboratory)

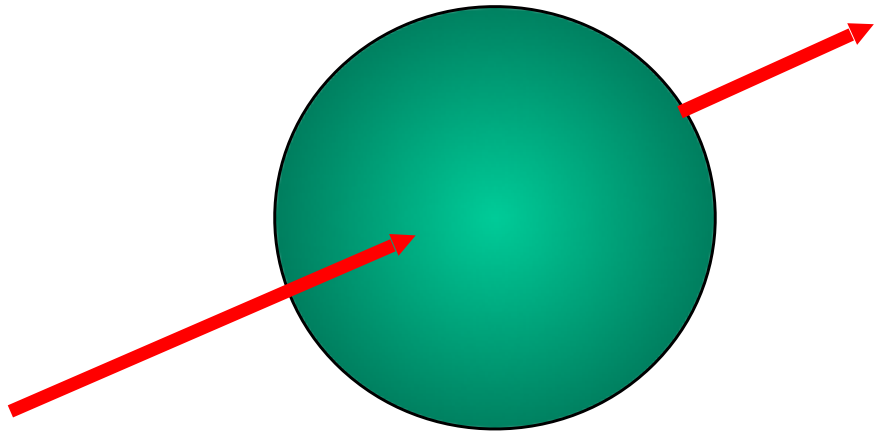
Diffuse reflections



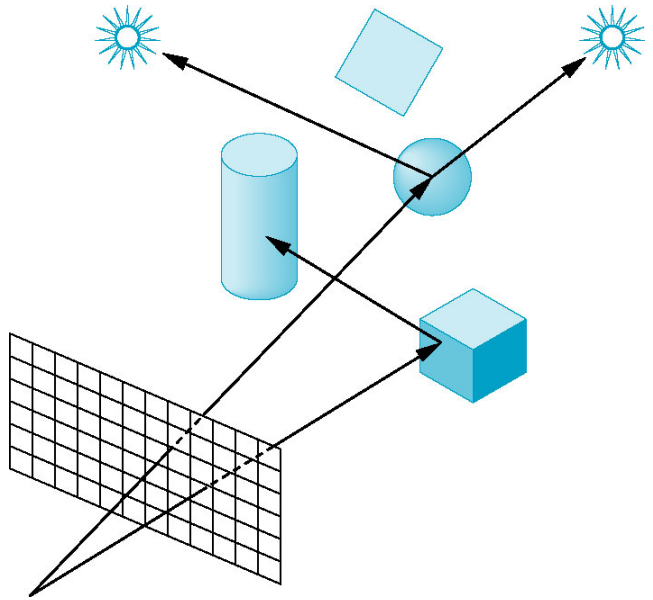
rough surface

Ray Casting a Sphere

- **Ray** is parametric with parameter **t**
- **Sphere** is **quadric**
- Resulting equation is a **scalar quadratic equation in t** which gives entry and exit Points of **ray** (or no solution if **ray** misses)



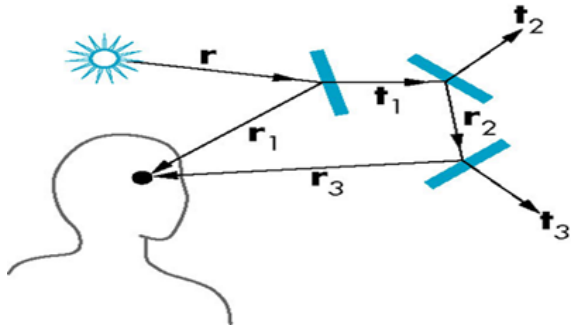
Building a Ray Caster



- **Best expressed recursively**
- Can remove recursion later
- **Image based approach**

  For each ray …….

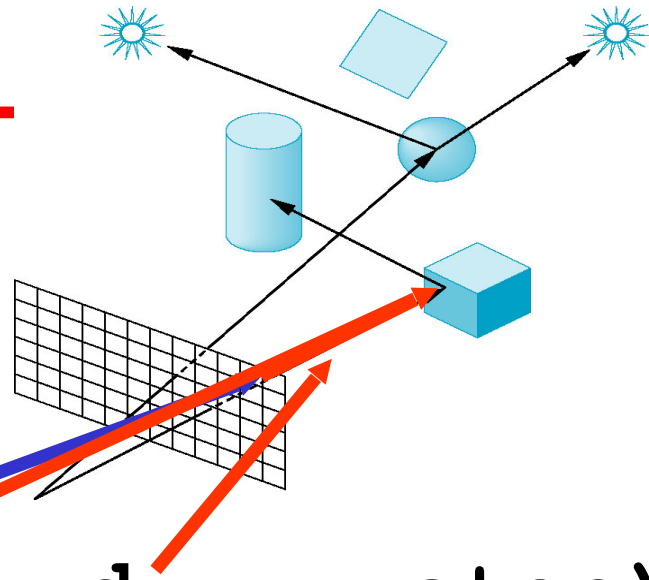
  - **Find intersection with closest surface**
    - –Need whole **object** database available
    - –Complexity of calculation limits **object** types
  - **Compute lighting at surface**
  - Trace **reflected** and **transmitted rays**



When to stop

- Some light will be absorbed **at each intersection**
  - **Track amount left**
- **Ignore rays** that go off to infinity
  - **Put large sphere around problem**
- **Count steps**

Recursive Ray Caster

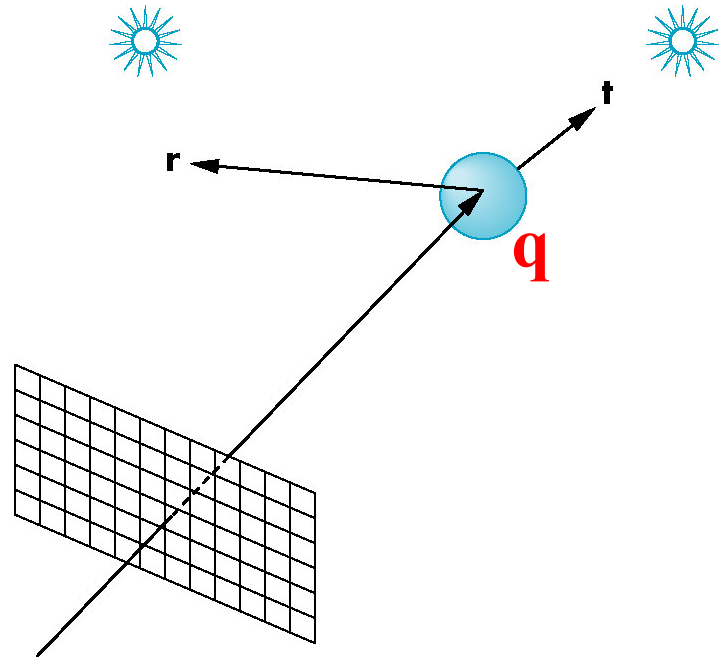


```
color c = trace(point p, vector d, int step)
{
  color local, reflected, transmitted;
  point q;     // intersection point
  normal n;
  if(step > max)
 return(background_color);
```

Recursive Ray Caster

```
q = intersect(p, d, status);
if(status==light_source)
 return(light_source_color);
if(status==no_intersection)
 return(background_color);

n = normal(q);
r = reflect(q, n);
t = transmit(q,n);
```

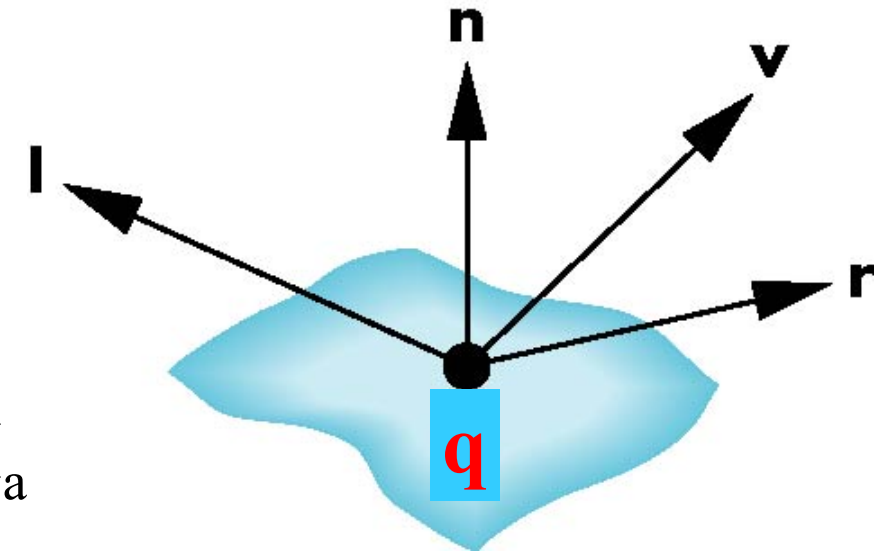


Recursive Ray Caster

```
local = phong(q, n, r);
reflected   = trace(q, r, step+1);
transmitted = trace(q, t, step+1);

return(local+ reflected+ transmitted);

}
```



$$I = k_d I_d \; \mathbf{l} \bullet \mathbf{n} + k_s I_s (\mathbf{v} \bullet \mathbf{r})^{\alpha} + k_a I_a$$

Computing Intersections

- Planes
- Implicit Objects
  - Quadrics
- Polyhedra
- Parametric Surfaces

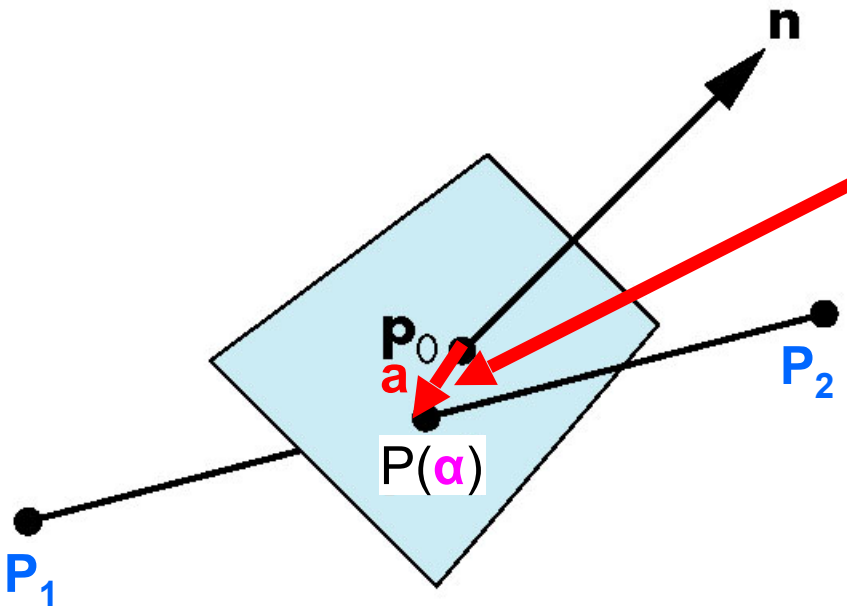
Plane-Line Intersections

Line segment between P_1 and P_2 parametric **α form**

$$P(\alpha) = P_1 + \alpha\,(P_2 - P_1)$$

Plane equation **n ● a** = 0 **n ● (P(α) – P_0)** = 0

What is **α for the point of intersection?**



Plane-Line Intersections

$P(\alpha) = P_1 + \alpha (P_2 - P_1)$

$n \bullet (P(\alpha) - P_0) = 0$

What is **α for the point of intersection?**

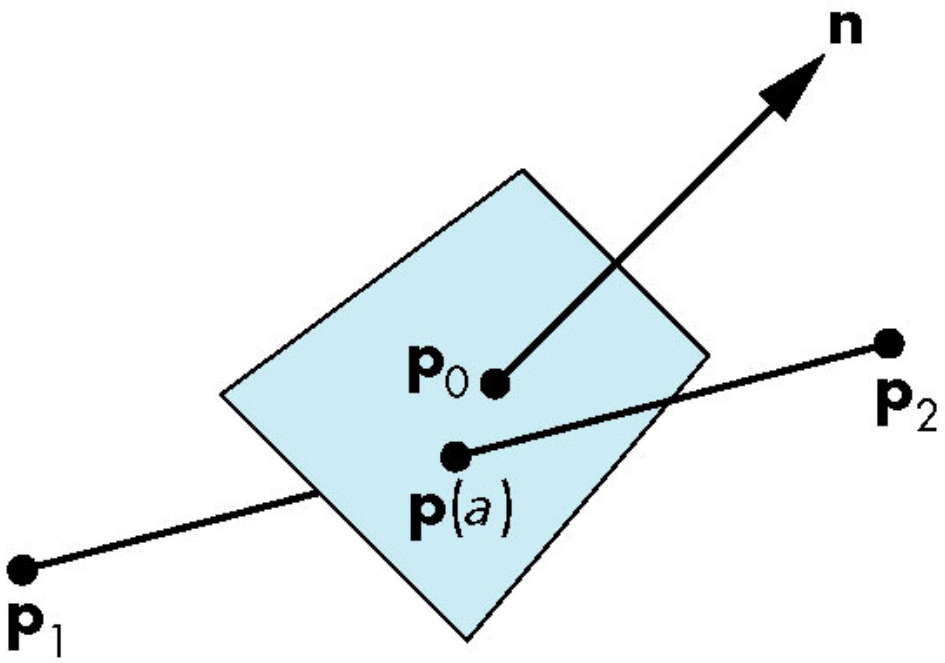
$n \bullet (P_1 + \alpha (P_2 - P_1) - P_0) = 0$

$n \bullet ((P_1 - P_0) + \alpha (P_2 - P_1)) = 0$

$n \bullet (P_1 - P_0) + \alpha\, n \bullet (P_2 - P_1)) = 0$

$n \bullet (P_0 - P_1) = \alpha\, n \bullet (P_2 - P_1))$ α

Plane-Line Intersections

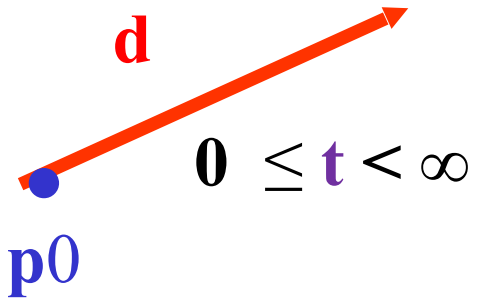


α

Implicit Surfaces

Ray from $\mathbf{p}_0$ in direction $\mathbf{d}$

$$\mathbf{p}(\mathbf{t}) = \mathbf{p}_0 + \mathbf{t}\,\mathbf{d}$$



General **implicit surface**

$$f(\mathbf{p}) = 0$$

Solve scalar equation for $\mathbf{t}$

$$f(\mathbf{p}(\mathbf{t})) = 0$$

General case requires numerical methods

Implicit Sphere

$$(\mathbf{p} - \mathbf{p}_c) \bullet (\mathbf{p} - \mathbf{p}_c) - r^2 = 0$$

Implicit Sphere

$$\mathbf{p}(\mathbf{t}) = \mathbf{p}_0 + \mathbf{t}\,\mathbf{d}$$

Ray equation

$$(\mathbf{p}_0 + \mathbf{t}\,\mathbf{d} - \mathbf{p}_c) \bullet (\mathbf{p}_0 + \mathbf{t}\,\mathbf{d} - \mathbf{p}_c) - r^2 = 0$$

$$(\mathbf{t}\,\mathbf{d} + \mathbf{p}_0 - \mathbf{p}_c) \bullet (\mathbf{t}\,\mathbf{d} + \mathbf{p}_0 - \mathbf{p}_c) - r^2 = 0$$

$$\mathbf{d} \bullet \mathbf{d}\,\mathbf{t}^2 + 2(\mathbf{p}_0 - \mathbf{p}_c)\mathbf{d}\,\mathbf{t} + (\mathbf{p}_0 - \mathbf{p}_c)^2 - r^2 = 0$$

Can get t

Planes

$\mathbf{p} \cdot \mathbf{n} + c = 0$ **Implicit** Plane

$\mathbf{p}(t) = \mathbf{p}_0 + t\,\mathbf{d}$ **Ray** equation

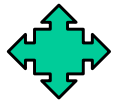
$t = -(\mathbf{p}_0 \cdot \mathbf{n} + c)/\,\mathbf{d} \cdot \mathbf{n}$

Can get t

Introduction

- OpenGL is based on a pipeline model in which primitives are rendered one at time
  - No shadows (except by tricks or multiple renderings)
  - No multiple reflections

- **Global** approaches
  - Rendering equation
  - **Ray tracing**
  - **Radiosity**



Radiosity

$$I_{out}(\Phi_{out}) = E(\Phi_{out}) + \int_{2\pi} R_{bd}(\Phi_{out}, \Phi_{in}) I_{in}(\Phi_{in}) \cos\theta \, d\omega$$

emission

bidirectional reflection coefficient

angle between normal and Φin

Terminology

- Energy ~ light (incident, transmitted)

  Must be conserved

- Energy flux = luminous flux = power = energy/unit time

  Measured in **lumens**

  Depends on wavelength so we can integrate over spectrum using **luminous efficiency curve** of sensor

- Energy density (Φ) = energy flux/unit area

$$I_{out}(\Phi_{out}) = E(\Phi_{out}) + \int_{2\pi} R_{bd}(\Phi_{out}, \Phi_{in}) I_{in}(\Phi_{in}) \cos\theta \, d\omega$$

emission

angle between normal and Φin

bidirectional reflection coefficient

Terminology

Intensity I ~ brightness

Brightness is perceptual

= flux/area-solid angle ω = power/unit projected area per solid angle

Measured in **candela**

$$\Phi = \int \int I \, dA \, d\omega$$

$$I_{out}(\Phi_{out}) = E(\Phi_{out}) + \int_{2\pi} R_{bd}(\Phi_{out}, \Phi_{in}) I_{in}(\Phi_{in}) \cos\theta \, d\omega$$

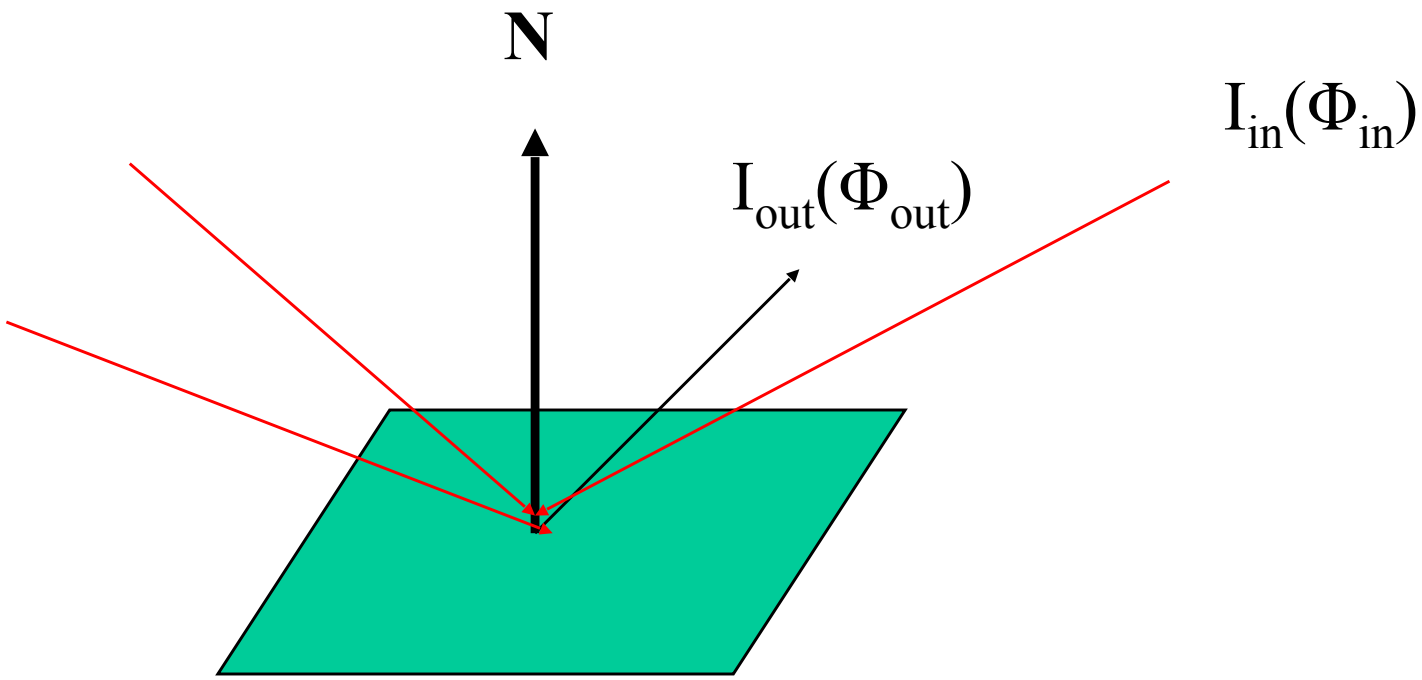
emission

bidirectional reflection coefficient

angle between normal and Φin

Rendering Equation (Kajiya)

- Consider a Point on a surface



$$I_{out}(\Phi_{out}) = E(\Phi_{out}) + \int_{2\pi} R_{bd}(\Phi_{out}, \Phi_{in}) I_{in}(\Phi_{in}) \cos\theta \, d\omega$$

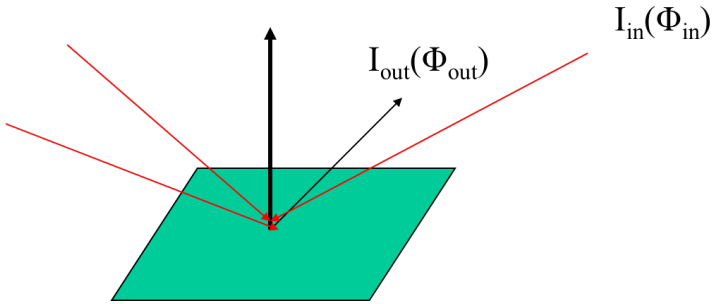
emission

angle between normal and Φin

bidirectional reflection coefficient

Rendering Equation

- **Outgoing light** $I_{out}(\Phi_{out})$ is from two sources
  - **Emission**
  - **Reflection of incoming light**



- Must integrate over **all incoming light**
  - Integrate over hemisphere

- Must account for foreshortening of **incoming light**

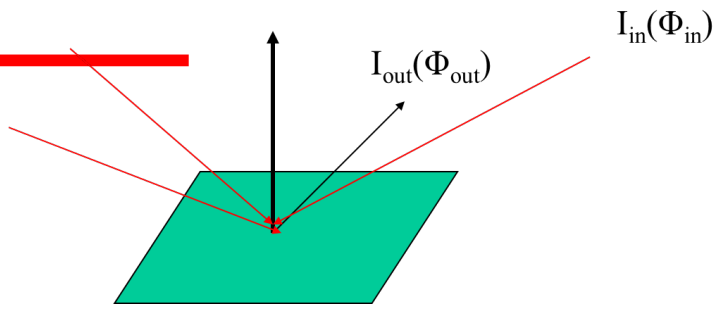
$$I_{out}(\Phi_{out}) = E(\Phi_{out}) + \int_{2\pi} R_{bd}(\Phi_{out}, \Phi_{in}) I_{in}(\Phi_{in}) \cos\theta \, d\omega$$

emission

angle between normal and Φin

bidirectional reflection coefficient

Rendering Equation



$$I_{out}(\Phi_{out}) = E(\Phi_{out}) + \int_{2\pi} R_{bd}(\Phi_{out}, \Phi_{in}) I_{in}(\Phi_{in}) \cos\theta \, d\omega$$

emission

angle between normal and Φin

bidirectional reflection coefficient

Note that angle is really two angles in 3D and wavelength is fixed

Rendering Equation

Rendering equation is an energy balance

Energy in = energy out

Integrate over hemisphere

Fredholm integral equation

Cannot be solved analytically in general

Various approximations of R_{bd} give **standard rendering models**

Should also add an **occlusion term** in front of right side to account for **other objects** blocking light from reaching **surface**

Another version

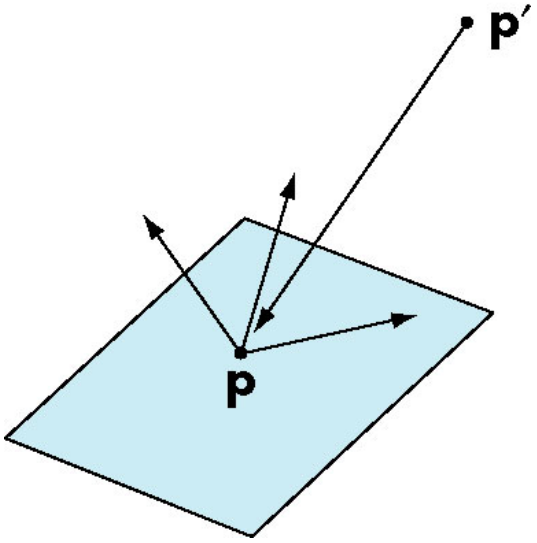
Consider light at a point **p** arriving from **p'**

$$i(\mathbf{p}, \mathbf{p'}) = \upsilon(\mathbf{p}, \mathbf{p'})(\varepsilon(\mathbf{p},\mathbf{p'}) + \int \rho(\mathbf{p}, \mathbf{p'}, \mathbf{p''})i(\mathbf{p'}, \mathbf{p''})d\mathbf{p''}$$

occlusion = 0 or $1/d^2$

emission from **p'** to **p**

light reflected at **p'** from all points **p''** towards **p**

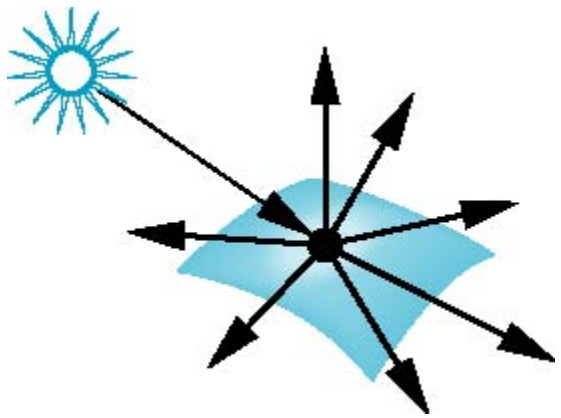


Radiosity

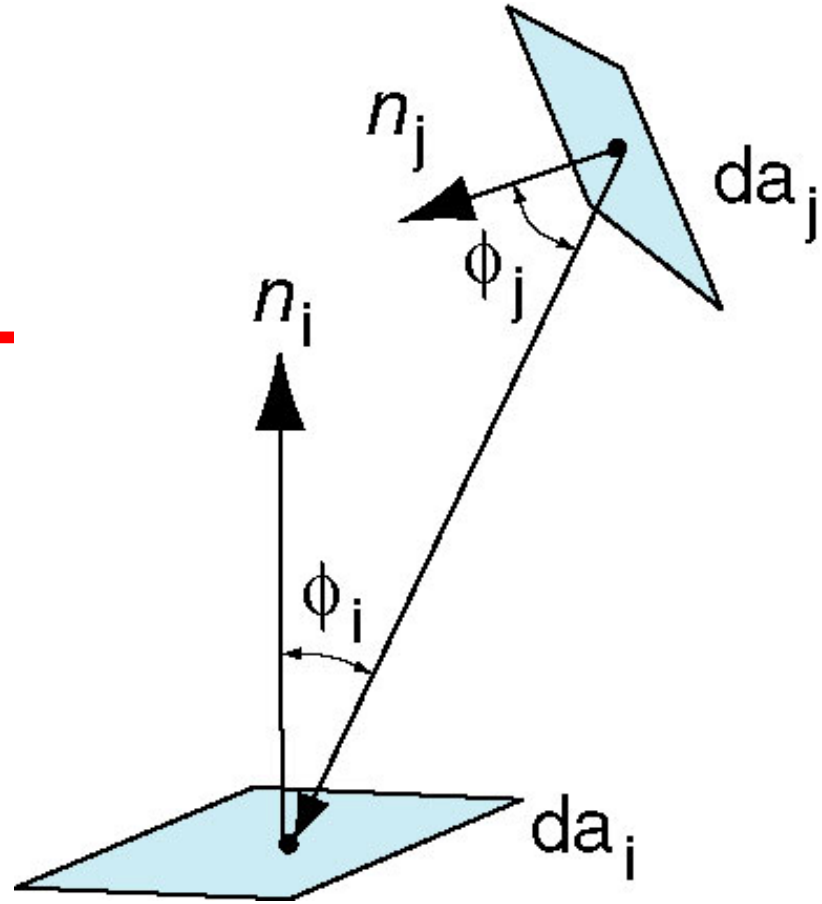
Consider **objects to be broken up into flat patches** (wh
may correspond to the **polygons in the model**)

Assume that **patches** are **perfectly diffuse reflectors**

Radiosity = flux = energy/unit area/ unit time leaving
patch



Notation



n patches numbered 1 to n

b_i = radiosity of patch i

a_i = **area patch i**

total intensity leaving patch **i** = $b_i a_i$

$e_i a_i$ = **emitted intensity** from patch **i**

ρ_i = **reflectivity of patch i**

f_{ij} = **form factor** = **fraction of energy** leaving patch **i** that reaches patch **j**

Radiosity Equation

$e_i a_i$ = **emitted intensity** from patch **i**

b_i = radiosity of patch i

ρ_i = **reflectivity of patch i**

energy balance

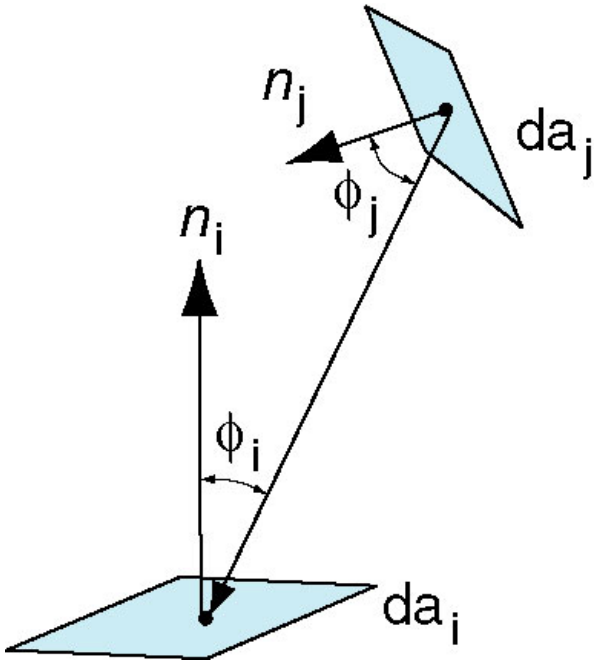
$$b_i a_i = e_i a_i + \rho_i \sum f_{ji} b_j a_j$$

reciprocity

$$f_{ij} a_i = f_{ji} a_j$$

radiosity equation

$$b_i = e_i + \rho_i \sum f_{ij} b_j$$



Matrix Form

b_i = radiosity of patch i

$$\mathbf{b} = [b_i]$$

$$\mathbf{e} = [e_i]$$

$$\mathbf{R} = [r_{ij}] \qquad r_{ij} = \rho_i \text{ if } i \neq j \qquad r_{ii} = 0$$

$$\mathbf{F} = [f_{ij}]$$

f_{ij} **form factor** = the **fraction of energy** leaving patch **i** that reaches patch **j**

Matrix Form

b_i = radiosity of patch i

$$\mathbf{b} = \mathbf{e} - \mathbf{RFb}$$

formal solution

$$\mathbf{b} = [\mathbf{I}-\mathbf{RF}]^{-1}\mathbf{e}$$

Not useful since number of patches n is usually very large
Alternative: use observation that F is sparse

We will consider determination of form factors later

Solving the Radiosity Equation

b_i = radiosity of patch i

For **sparse matrices**, **iterative methods** usually require only **O**(**n**) operations per iteration

Jacobi's method

$$\mathbf{b}^{k+1} = \mathbf{e} - \mathbf{RF}\mathbf{b}^{k}$$

Gauss-Seidel: use immediate updates

Series Approximation

b_i = radiosity of patch i

$$1/(1-x) = 1 + x + x^2 + \ldots\ldots$$

$$[\mathbf{I-RF}]^{-1} = \mathbf{I} + \mathbf{RF} + (\mathbf{RF})^2 + \ldots$$

$$\mathbf{b} = [\mathbf{I-RF}]^{-1}\mathbf{e} = \mathbf{e} + \mathbf{RFe} + (\mathbf{RF})^2\mathbf{e} + \ldots$$

Carrying out Radiosity

In practice, **radiosity rendering** has three major steps.

$$b = [I-RF]^{-1}e = e + RFe + (RF)^2e$$

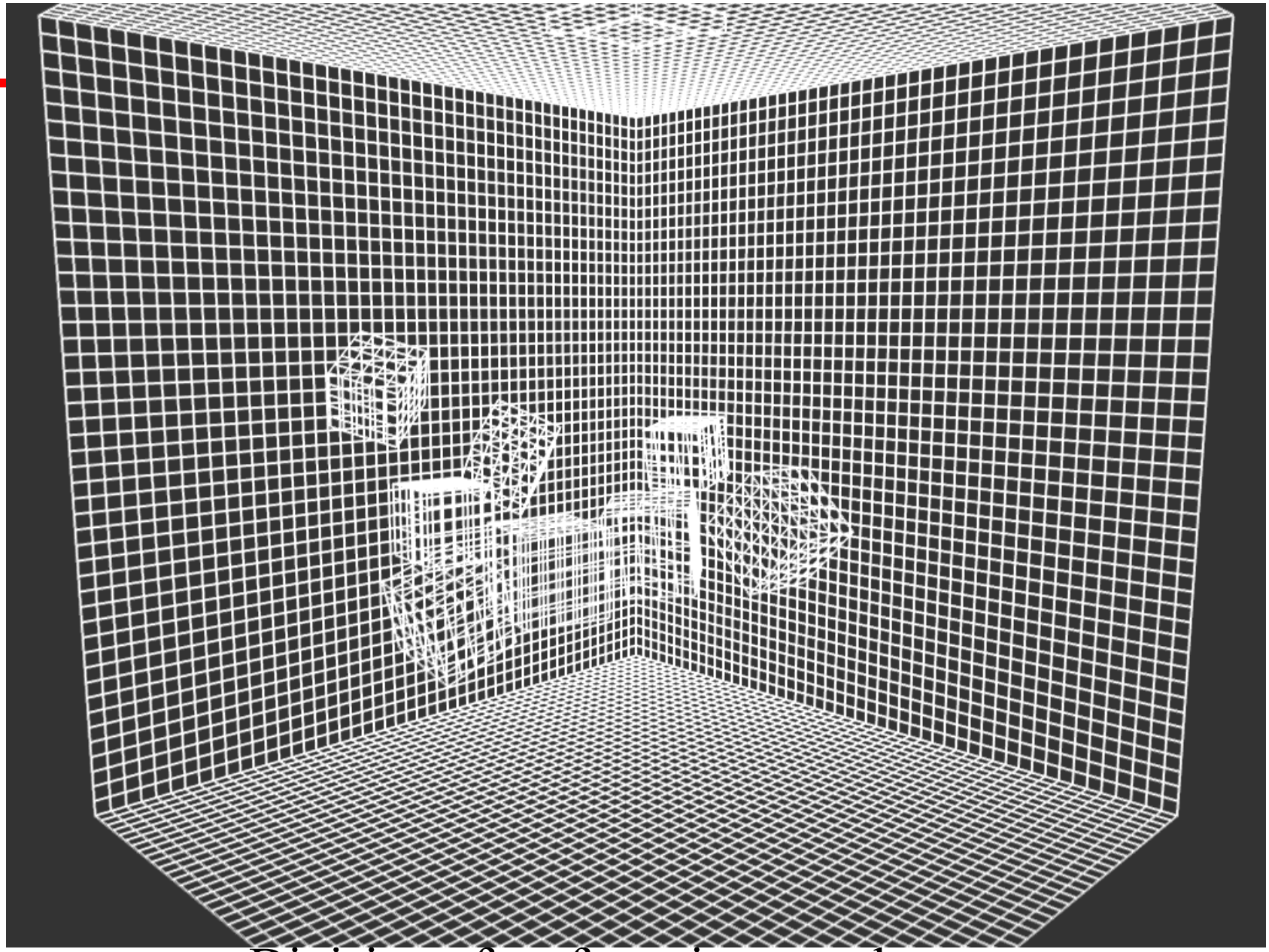
First, **we divide the scene into a mesh of patches**. This step requires some skill, because more patches require the calculation of more form factors. However, it is the division of surfaces into patches that allows radiosity to yield images with subtle diffuse interactions. Often the creation of the initial mesh can be done interactively, allowing the placement of more patches in regions such as corners between surfaces where we expect to see diffuse diffuse interactions. Another approach is based on the observation that the radiosity of a large surface is equal to the area- weighted sum of the radiosities of any subdivision of it. Hence, we can start with a fairly rough mesh and refine it later (progressive radiosity).

Once we have a mesh, we can compute the **form factors**, **F** the most computationally intense part of the process.

Once we have mesh and the form factors **F**, we can **solve the radiosity equation**. We form the emission array **e** using the values for the light sources in the scene and assign colors to the surfaces, forming **R**. Now we can solve the radiosity equation to determine **b**. The components of **b** act as the new colors of the patches.

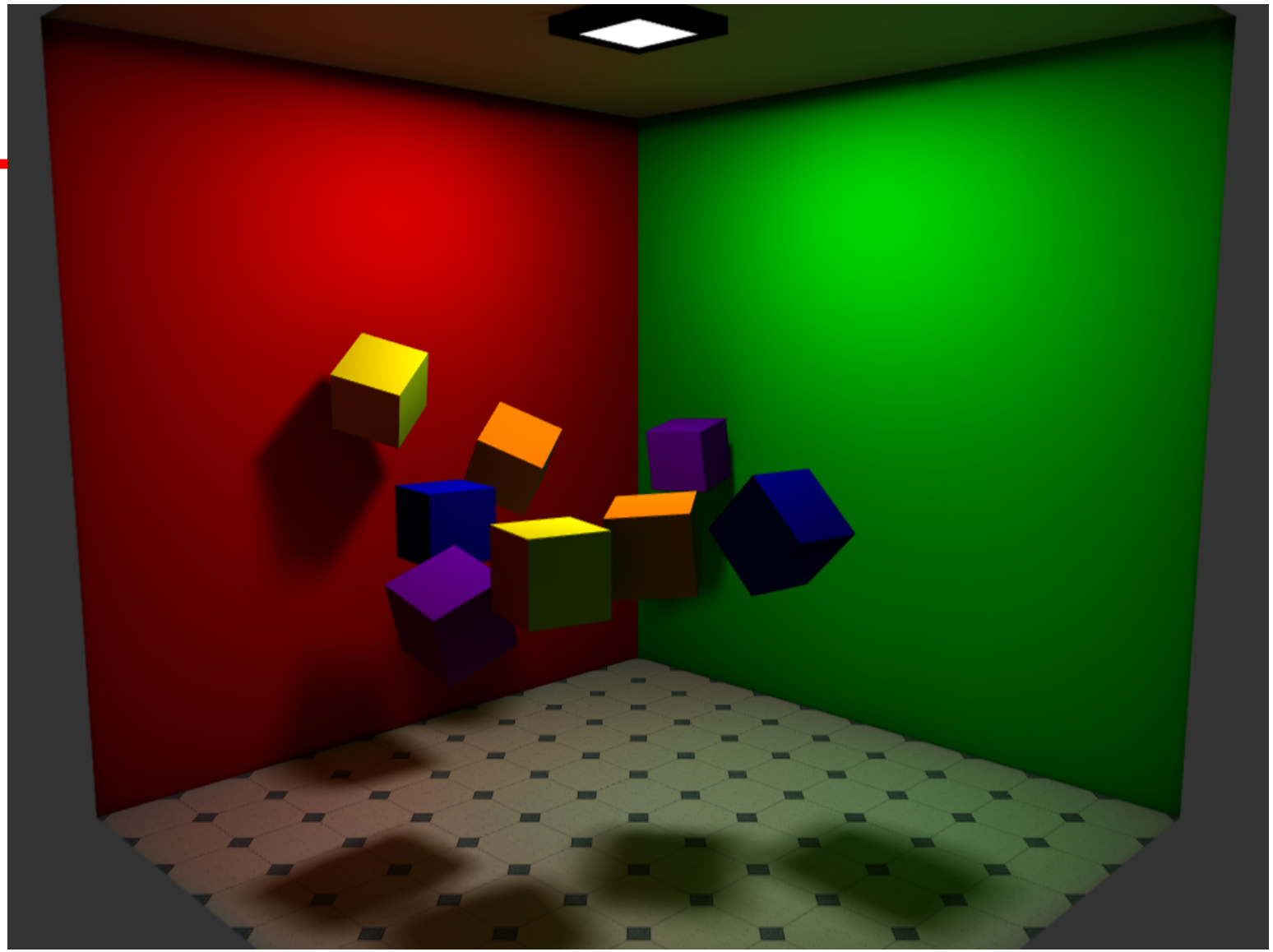
We can now place the viewer in the scene and render with a conventional renderer.

Patches



Division of surfaces into patches

Rendered Image



It started with the initial mesh in previous slide, which was then altered with a **particle system to achieve a better set of patches**. It shows the strength of **radiosity** for rendering interiors that are composed of **diffuse reflectors**.

NEXT.

11.20.2023 (M 5:30 to 7) (26)		EXAM 4 REVIEW
11.27.2023 (M 5:30 to 7) (27)		PROJECT 4
11.29.2023 (W 5:30 to 7) (28)		EXAM 4
12.11.2023 (M 5:30 to 7)		FINAL EXAM

At 6:45 PM.

End Class 25

VH, Download Attendance Report
Rename it:
11.15.2023 Attendance Report FINAL

VH, upload Lecture 13 to CANVAS.