01418585 Rendering and Shading Techniques
Lecture 02

pramook@gmail.com

เวกเตอร์สามมิติ

- ลำดับของจำนวนจริงสามตัว

$$(x, y, z)$$

- สัญลักษณ์: ตัวอักษรตัวพิมพ์เล็กหนา

$$\mathbf{u}, \mathbf{v}, \mathbf{w}$$

- เซตของเวกเตอร์สามมิติ

$$\mathbb{R}^3 = \{(x, y, z) : x, y, z \in \mathbb{R}\}$$

- เวกเตอร์พิเศษ

$$\mathbf{x} = (1, 0, 0),\ \mathbf{y} = (0, 1, 0),\ \mathbf{z} = (0, 0, 1),\ \mathbf{0} = (0, 0, 0)$$

Linear Combination

- ให้ $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ เป็นเวกเตอร์ใดๆ

  และ $c_1, c_2, \ldots, c_n$ คือจำนวนจริงใดๆ

  เราเรียก

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_n\mathbf{v}_n$$

  ว่า linear combination ของ $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$

Linear Combination (ต่อ)

- เวกเตอร์สามมิติทุกเวกเตอร์เป็น linear combination ของ $\mathbf{x}$, $\mathbf{y}$, และ $\mathbf{z}$

$$(x, y, z) = x(1, 0, 0) + y(0, 1, 0) + z(0, 0, 1) = x\mathbf{x} + y\mathbf{y} + z\mathbf{z}$$

Span

- ถ้า $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ เป็นเวกเตอร์ใดๆ แล้ว

  เราเรียกเซต

  $$\{c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n : c_1, c_2, \ldots, c_n \in \mathbb{R}\}$$

  ว่า span ของ $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$

Span (ต่อ)

- Span ของ $\mathbf{x}$, $\mathbf{y}$, และ $\mathbf{z}$ มีค่าเท่ากับ $\mathbb{R}^3$
- Span ของ $\mathbf{x}$ มีค่าเท่ากับแกน x
- Span ของ $\mathbf{y}$ มีค่าเท่ากับแกน y
- Span ของ $\mathbf{z}$ มีค่าเท่ากับแกน z
- Span ของ $\mathbf{x}$ และ $\mathbf{y}$ มีค่าเท่ากับระนาบ xy
- Span ของ $\mathbf{x}$ และ $\mathbf{z}$ มีค่าเท่ากับระนาบ xz
- Span ของ $\mathbf{y}$ และ $\mathbf{z}$ มีค่าเท่ากับระนาบ yz

Linear Dependence

- เรากล่าวว่า $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$
  เป็นกลุ่มของเวกเตอร์ที่ linearly dependent
  ถ้ามีสเกลาร์ $c_1, c_2, \ldots, c_n$ ที่มีตัวใดตัวหนึ่งมีค่าไม่เท่ากับ 0
  ที่ทำให้

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_n\mathbf{v}_n = \mathbf{0}$$

Linear Independence

- ตรงข้ามกับ linear dependence
- เรากล่าวว่า $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$

  เป็นกลุ่มของเวกเตอร์ที่ linearly independent

  ถ้าค่า $c_1, c_2, \ldots, c_n$ ที่ทำให้

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_n\mathbf{v}_n = \mathbf{0}$$

  คือ $c_1 = c_2 = \cdots = c_n = 0$ เท่านั้น

Linear Independence (ต่อ)

- **x**, **y**, และ **z** **---** linearly independent
- (1,2,0) และ (2,4,0) --- linearly dependent
  เพราะ 2*(1,2,0) – (2,4,0) = (0,0,0)
- (1,0,1), (2,3,0), (2,1.5,1) --- linearly dependent
  เพราะ (1,0,1) + 0.5*(2,3,0) – (2,1.5,1) = (0,0,0)
- (0,0,0) --- linearly dependent
  เพราะ c(0,0,0) = (0,0,0) สำหรับค่า c ใดๆ ที่ไม่เท่ากับ 0

Basis

- ถ้า span ของ **u, v,** และ **w**

  มีค่าเท่ากับเซตของเวกเตอร์สามมิติทั้งหมด

  เราเรียก **u, v,** และ **w** ว่าเป็น basis ของปริภูมิเวกเตอร์สามมิติ

Basis (ต่อ)

- **x**, **y**, และ **z** เป็น basis ของปริภูมิเวกเตอร์สามมิติ
- (1,1,0), (0,1,1), และ (1,0,1) ก็เป็น basis
- แต่ (1,0,1), (2,3,0), (2,1.5,1) ไม่ใช่ เพราะมันไม่ linearly independent

ผลคูณเวกเตอร์ (ต่อ)

- สมบัติต่างๆ

$$\mathbf{u} \times \mathbf{v} = -\mathbf{v} \times \mathbf{u}$$

$$\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = \mathbf{u} \times \mathbf{v} + \mathbf{u} \times \mathbf{w}$$

$$\mathbf{u} \times (r\mathbf{v}) = (r\mathbf{u}) \times \mathbf{v} = r(\mathbf{u} \times \mathbf{v})$$

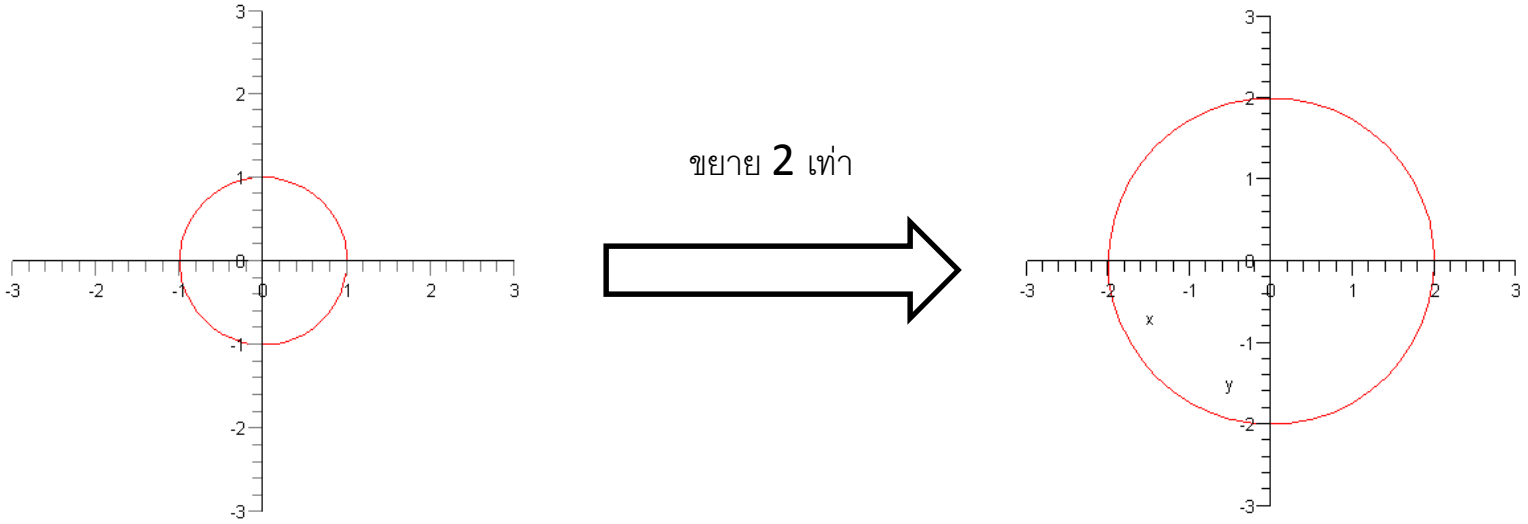
การแปลง

การแปลง (Transformations)

- ตัวอย่าง
  - “เลื่อนไปทางซ้าย 1 หน่วย”
  - “หมุนรอบแกน y 90 องศา”
  - “ขยายขนาดตามแกน z 2 เท่า”
- เอาไปใช้ที่ไหน?
  - Modeling
  - Animation
  - Rendering Pipeline

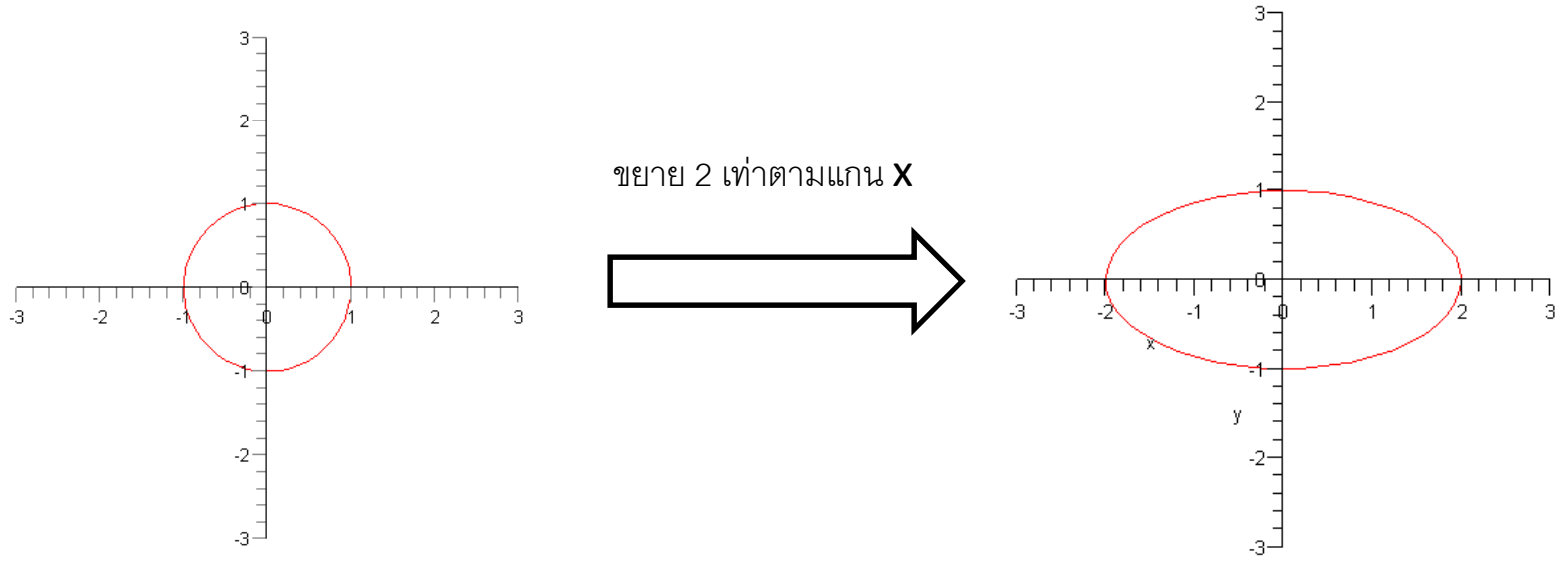
ตัวอย่าง

- สมมติเรารู้วิธีสร้างวงกลมรัศมี 1 หน่วย จุดศูนย์กลางอยู่ที่ (0,0)
- อยากได้วงกลมรัศมี 2 หน่วย จุดศูนย์กลางอยู่ที่เดิม



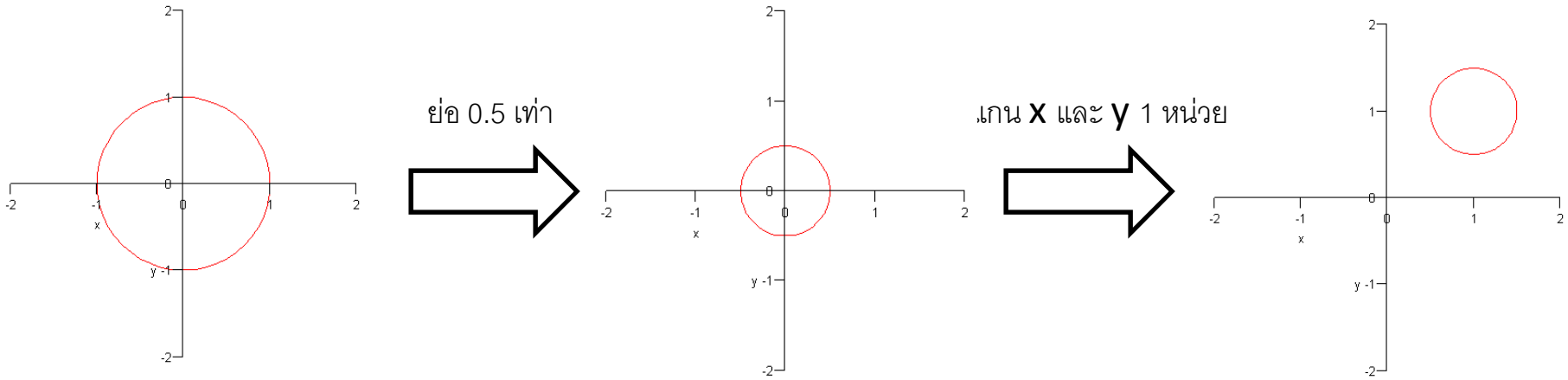
ตัวอย่าง

- อยากได้วงรีแกนเอกยาว 2 หน่วย แกนโทยาว 1 หน่วย



ตัวอย่าง

- อยากได้วงกลมรัศมี 0.5 หน่วย จุดศูนย์กลางอยู่ที่จุด (1,1)



การแปลงในสามมิติ

ในปริภูมิสามมิติ

- พิกัดในสามมิติแทนด้วยลำดับ (x,y,z)
- หรือด้วย (x,y,z,w) ถ้าอยู่ในรูป homogeneous coordinate
- homogeneous coordinate (x,y,z,w) หมายถึงพิกัด $(x/w,\ y/w,\ z/w)$ ในปริภูมิสามมิติ

ในปริภูมิสามมิติ (ต่อ)

- พิกัดในสามมิติสามารถเขียนได้อีกแบบหนึ่งในรูป matrix

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- มีความหมายเหมือนกับ homogeneous coordinate $(x,y,z,1)$

การแปลงในปริภูมิสามมิติ

- การแปลงแอฟไฟน์สามชนิดที่เรียนผ่านมา
  - การเลื่อนแกนขนาน (translation)
  - การย่อขยาย (scaling)
  - การหมุน (rotation)

  สามารถแทนได้ด้วย matrix 4 คูณ 4

การเลื่อนแกนขนาน

- สัญลักษณ์ $T_{a,b,c}$
- ส่งพิกัด (x,y,z) ไปยังพิกัด (x+a, y+b, z+c)
- มี matrix เป็น

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

การย่อขยายขนาด

- สัญลักษณ์ $S_{a,b,c}$
- ส่งพิกัด (x,y,z) ไปยังพิกัด (ax, by, cz)
- นี่เป็นการย่อขยายรอบพิกัด (0,0,0) เนื่องจากพิกัด (0,0,0) ไม่เปลี่ยนแปลง
- มี matrix เป็น

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

การหมุน

- เวลาหมุนจะต้องบอกสองอย่าง
  - **แกน**ที่จะใช้หมุน
  - มุมที่จะใช้หมุน
- เวลาระบุแกนเราจะระบุด้วยเวกเตอร์ (a,b,c)
- **แกน**คือเส้นตรงที่เกิดจากจุดทั้งหมดที่อยู่ในรูป (at, bt, ct) เมื่อ t เป็นจำนวนจริงใดๆ
- แกนจะผ่านจุด (0,0,0) เสมอ
- เวลาทำการหมุน จุดที่อยู่บนแกนจะไม่เคลื่อนที่
- มุมที่จะใช้หมุนส่วนใหญ่จะใช้สัญลักษณ์ θ

การหมุนรอบแกน Z

- แกน Z คือเซตของพิกัดต่างๆ ที่อยู่ในรูป (0, 0, t)
- สามารถระบุได้ด้วยเวกเตอร์ (0,0,1)
- สัญลักษณ์ $R_{\theta,0,0,1}$
- ส่งพิกัด $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ ไปยังพิกัด $\begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \\ 1 \end{bmatrix}$

การหมุนรอบแกน Z (ต่อ)

- มี matrix เป็น

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

การหมุนรอบแกน X

- แกน X คือเซตของพิกัดต่างๆ ที่อยู่ในรูป (t, 0, 0)
- สามารถระบุได้ด้วยเวกเตอร์ (1,0,0)
- สัญลักษณ์ $R_{\theta,1,0,0}$
- ส่งพิกัด $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ ไปยังพิกัด $\begin{bmatrix} x \\ y\cos\theta - z\sin\theta \\ y\sin\theta + z\cos\theta \\ 1 \end{bmatrix}$

การหมุนรอบแกน **X** (ต่อ)

- มี matrix เป็น

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

การหมุนรอบแกน y

- แกน z คือเซตของพิกัดต่างๆ ที่อยู่ในรูป (0, t, 0)
- สามารถระบุได้ด้วยเวกเตอร์ (0,1,0)
- สัญลักษณ์ $R_{\theta,0,1,0}$
- ส่งพิกัด $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ ไปยังพิกัด $\begin{bmatrix} z\sin\theta + x\cos\theta \\ y \\ z\cos\theta - x\sin\theta \\ 1 \end{bmatrix}$

การหมุนรอบแกน y (ต่อ)

- มี matrix เป็น

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

การหมุนรอบแกนใดๆ

- สัญลักษณ์ $R_{\theta,a,b,c}$
- มี matrix เป็น

$$\begin{bmatrix} a^2(1-C)+C & ab(1-C)+cS & ac(1-C)+bS & 0 \\ ba(1-C)+cS & b^2(1-C)+C & bc(1-C)+aS & 0 \\ ca(1-C)+bS & cb(1-C)+aS & c^2(1-C)+C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

เมื่อ $C = \cos\theta$ และ $S = \sin\theta$

การแปลง affine

- การแปลง affine คือการแปลงที่สามารถเขียนอยู่ในรูป matrix

$$\begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ระบบพิกัด

- ระบบพิกัดเป็นตัวกำหนดว่าพิกัดใดแทนจุดใด
- พิกัดและจุด?
  - พิกัดคือลำดับของเลขสามตัว: **(x,y,z)**
  - จุดคือจุดที่เราเห็นด้วยตา
- ระบบพิกัดในสามมิติมีส่วนประกอบอยู่สามส่วน
  - จุดออริจิน **o**: จุดนี้จะแทนด้วยพิกัด (0,0,0) ในระบบพิกัด
  - เวกเตอร์สามตัว **i, j,** และ **k** สำหรับกำหนดทิศทางแกน **x, y,** และ **z** ตามลำดับ

ระบบพิกัด (ต่อ)

- พิกัด **(x,y,z)** ในระบบพิกัดนี้จึงหมายถึงจุด

$$\mathbf{o} + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

กล่าวคือมันคือจุดที่อยู่ห่างจากจุด **o**
ไปตามแนวเวกเตอร์ **i** เป็นระยะ **x** เท่าของความยาวเวกเตอร์ **I**
ไปตามแนวเวกเตอร์ **j** เป็นระยะ **y** เท่าของความยาวเวกเตอร์ **j**
ไปตามแนวเวกเตอร์ **k** เป็นระยะ **z** เท่าของความยาวเวกเตอร์ **k**

ระบบพิกัด

- เขียนได้อีกแบบหนึ่งว่าพิกัด (x,y,z) หมายถึงจุด

$$\begin{bmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{o}\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$$

ระบบพิกัดกับการแปลง

- พิจารณาการแปลง affine

$$M = \begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ระบบพิกัดกับการแปลง (ต่อ)

- มันส่งพิกัด $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ ไปยังพิกัด $\begin{bmatrix} ax + dy + iz + l \\ bx + ey + jz + m \\ cx + fy + kz + n \\ 1 \end{bmatrix}$

ระบบพิกัดกับการแปลง (ต่อ)

- พูดได้อีกอย่างหนึ่งคือ M ส่งจุด

$$\begin{bmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{o}\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$$

ไปยังจุด

$$\begin{bmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{o}\end{bmatrix}\begin{bmatrix}ax+dy+iz+l\\ bx+ey+jz+m\\ cx+fy+kz+n\\ 1\end{bmatrix}=\begin{bmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{o}\end{bmatrix}\begin{bmatrix}a & d & i & l\\ b & e & j & m\\ c & f & k & n\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$$

ระบบพิกัดกับการแปลง (ต่อ)

- แต่เราอาจมองได้อีกว่า

$$\begin{bmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k} & \mathbf{o}\end{bmatrix}\begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

มีค่าเท่ากับ

$$\begin{bmatrix} a\mathbf{i}+b\mathbf{j}+c\mathbf{k} & d\mathbf{i}+e\mathbf{j}+f\mathbf{k} & i\mathbf{i}+j\mathbf{j}+k\mathbf{k} & o+l\mathbf{i}+m\mathbf{j}+n\mathbf{k}\end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

ระบบพิกัดกับการแปลง (ต่อ)

- ดังนั้นการแปลง M จึงสามารถมองได้ว่าเป็นการเปลี่ยนระบบพิกัดจากระบบพิกัดที่
  - มี **o** เป็นจุดออริจิน
  - มี **i** เป็นตัวกำหนดทิศทางแกน **x**
  - มี **j** เป็นตัวกำหนดทิศทางแกน **y**
  - มี **k** เป็นตัวกำหนดทิศทางแกน **z**

  เป็นระบบพิกัดที่
  - มี **o+li+mj+nk** เป็นจุดออริจิน
  - มี **ai+bj+ck** เป็นตัวกำหนดทิศทางแกน **x**
  - มี **di+ej+fk** เป็นตัวกำหนดทิศทางแกน **y**
  - มี **ii+jj+kk** เป็นตัวกำหนดทิศทางแกน **z**

ระบบพิกัดกับการแปลง (ต่อ)

- หรือกล่าวได้อีกอย่างหนึ่งคือ
  - จุดออริจินใหม่คือจุดที่มีพิกัด (l,m,n) ในระบบพิกัดเดิม
  - เวกเตอร์แกน x ใหม่ คือเวกเตอร์ (a,b,c) ในระบบพิกัดเดิม
  - เวกเตอร์แกน y ใหม่ คือเวกเตอร์ (d,e,f) ในระบบพิกัดเดิม
  - เวกเตอร์แกน z ใหม่ คือเวกเตอร์ (i,j,k) ในระบบพิกัดเดิม

ระบบพิกัดกับการแปลง (ต่อ)

$$\begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

แกน x ใหม่

ระบบพิกัดกับการแปลง (ต่อ)

$$\begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

แกน y ใหม่

ระบบพิกัดกับการแปลง (ต่อ)

$$\begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

แกน z ใหม่

ระบบพิกัดกับการแปลง (ต่อ)

$$\begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

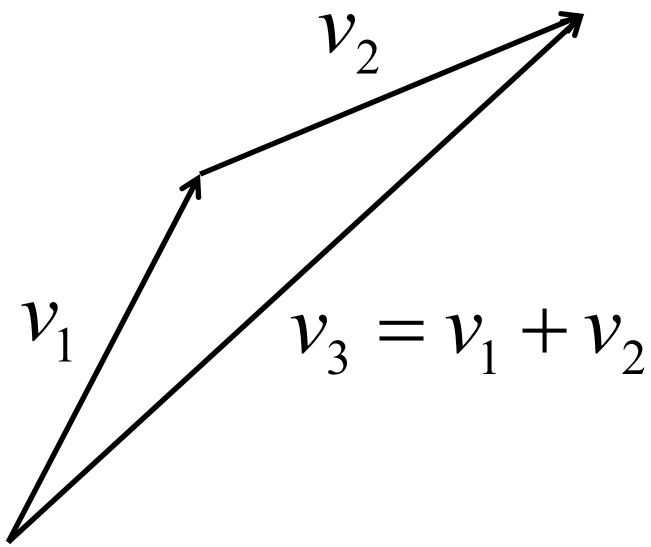
จุด origin ใหม่

Homogeneous Coordinate กับเวกเตอร์

- Homogeneous coodinate สามารถใช้แทนได้ทั้ง**จุด**และ**เวกเตอร์**
- ถ้า w ใน (x,y,z,w) เป็น 1 แสดงว่ามันแทนจุด
  - ถ้ามันไม่ใช่ 1 ให้เอา w ไปหาทุกตัวเพื่อทำให้มันเป็น 1 เสีย
- ถ้า w ใน (x,y,z,w) เป็น 0 แสดงว่ามันแทนเวกเตอร์ (ทิศทาง)

จุดกับเวกเตอร์

- จุด คือ “ตำแหน่ง”
- เวกเตอร์ คือ “ทิศทาง”
- คุณเอาเวกเตอร์สองเวกเตอร์มาบวกกันได้

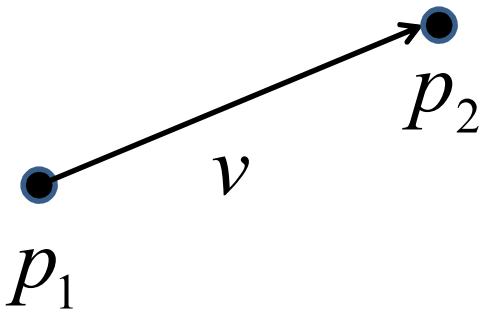


จุดกับเวกเตอร์ (ต่อ)

- แต่คุณเอาจุดสองจุดมาบวกกันไม่ได้

$$p_1 + p_2 = ???$$

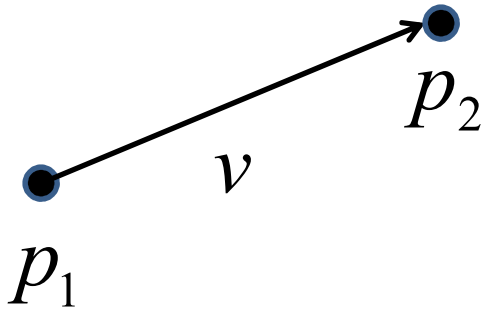
- แต่เอาจุดมาบวกกันเวกเตอร์ได้ จะได้จุดอีกจุดหนึ่ง



$$p_1 + v = p_2$$

จุดกับเวกเตอร์ (ต่อ)

- ในทำนองเดียวกัน คุณสามารถหาผลต่างของจุดได้ ซึ่งจะได้ผลลัพธ์ออกมาเป็นเวกเตอร์



$$p_1 + v = p_2$$

- ยกตัวอย่างเช่น $\begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix} - \begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 3 \\ 0 \end{bmatrix}$

จุดกับเวกเตอร์ (ต่อ)

- การแปลง **affine** มีผลต่อจุดและเวกเตอร์ต่างกัน

$$\begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} (ax+dy+iz)+l \\ (bx+ey+jz)+m \\ (cx+fy+kz)+n \\ 1 \end{bmatrix}$$

แต่

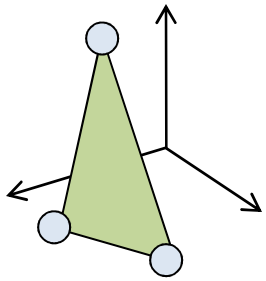
$$\begin{bmatrix} a & d & i & l \\ b & e & j & m \\ c & f & k & n \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} ax+dy+iz \\ bx+ey+jz \\ cx+fy+kz \\ 0 \end{bmatrix}$$

จุดกับเวกเตอร์ (ต่อ)

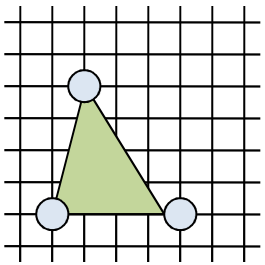
- ให้
  - M เป็นการแปลง affine
  - p เป็นจุด
  - ให้ v เป็นเวกเตอร์
- ได้ว่า
  - Mp เป็นจุด
  - Mv เป็นเวกเตอร์
- ในการแปลงจุดจะมีการเลื่อนแกนขนานติดมาด้วย
- แต่ในการแปลงเวกเตอร์ จะไม่มีการเลื่อนแกนขนานติดมาด้วย

TRANSFORMATIONS IN RAY TRACING

OpenGL Vertex Transformation (ต่อ)



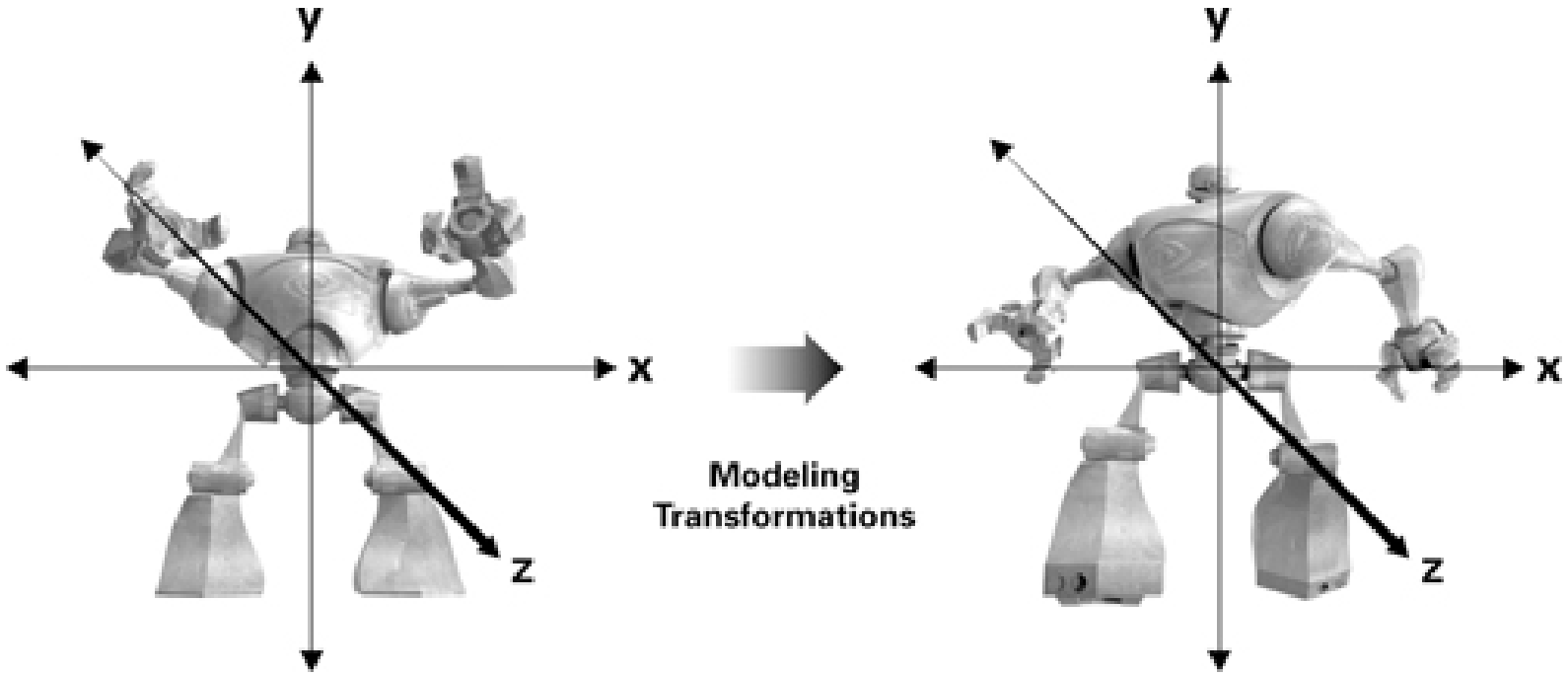
$$\begin{bmatrix} x_w \\ y_w \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \text{View} \\ \text{Transform} \end{bmatrix} \begin{bmatrix} \text{Model} \\ \text{Transform} \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$



Modeling Transform

- Object space คือระบบพิกัดที่ศิลปินทำการขึ้นโมเดลมาให้
- World space คือระบบพิกัดกลางของฉากที่โมเดลหลายๆ โมเดลมาอยู่ร่วมกัน
- Modeling transform ทำหน้าที่เปลี่ยน vertex จากที่อยู่ใน object space มาอยู่ใน world space
- ในขณะเดียวกันมันอาจจะเปลี่ยนแปลงหน้าตาหรือท่าทางของโมเดลได้ด้วย

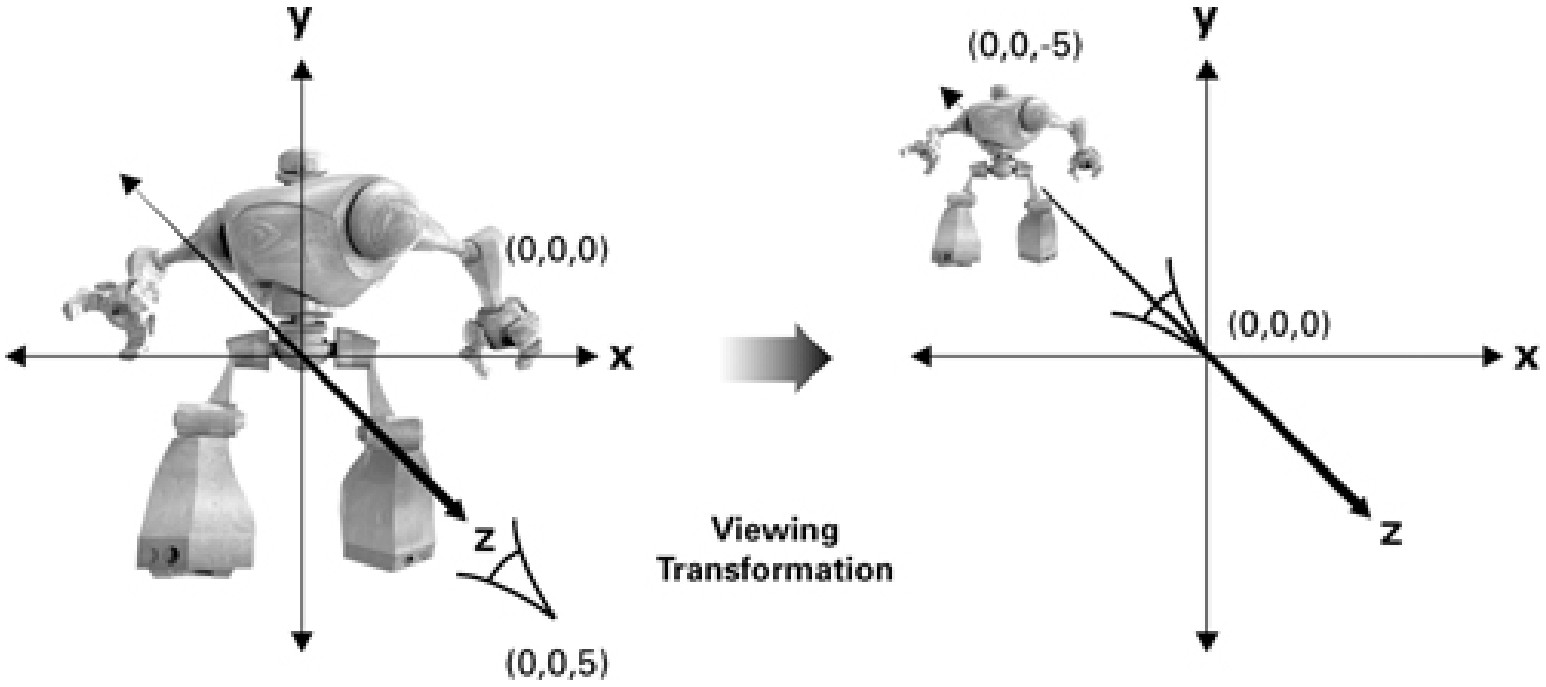
Modeling Transform (ต่อ)



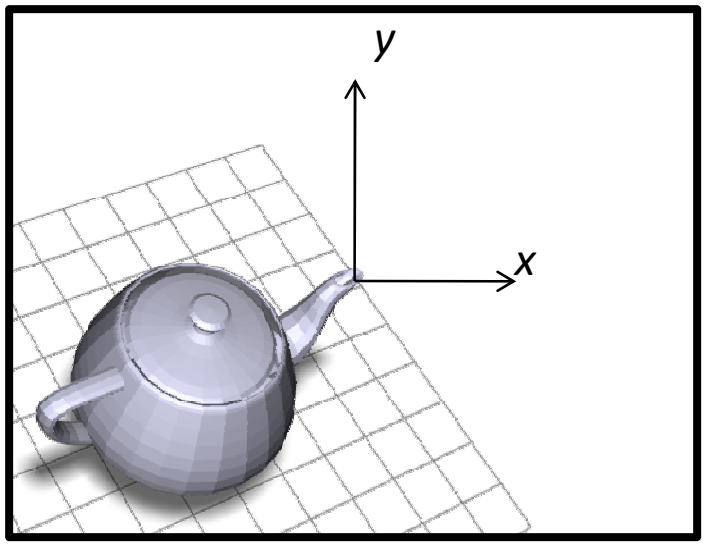
View Transform

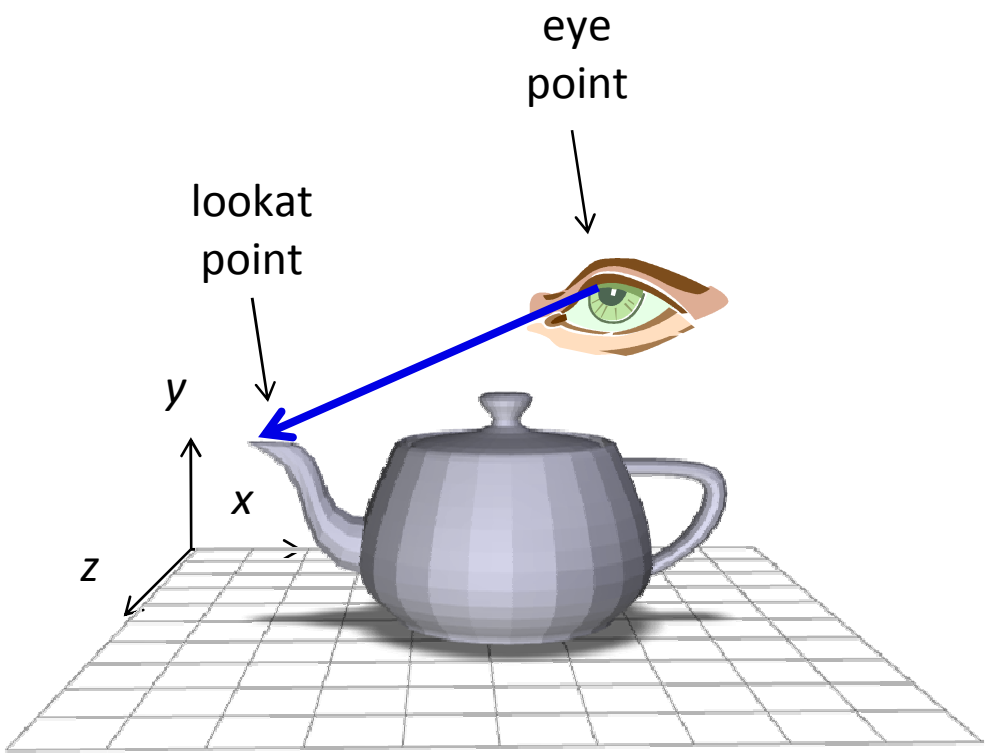
- View transform ใช้ในการเซตมุมกล้อง
- Eye space คือระบบพิกัดที่
  - ตาเราอยู่ที่จุด (0,0,0)
  - เรามองไปในทิศทางแกน z ทางลบ (ในทิศทางของเวกเตอร์ **–k**)
  - ทิศทางแกน y คือ “ด้านบน”
- View transform เปลี่ยน vertex ที่อยู่ใน world space มาอยู่ใน eye space

View Transform (ต่อ)



View Transform (ต่อ)

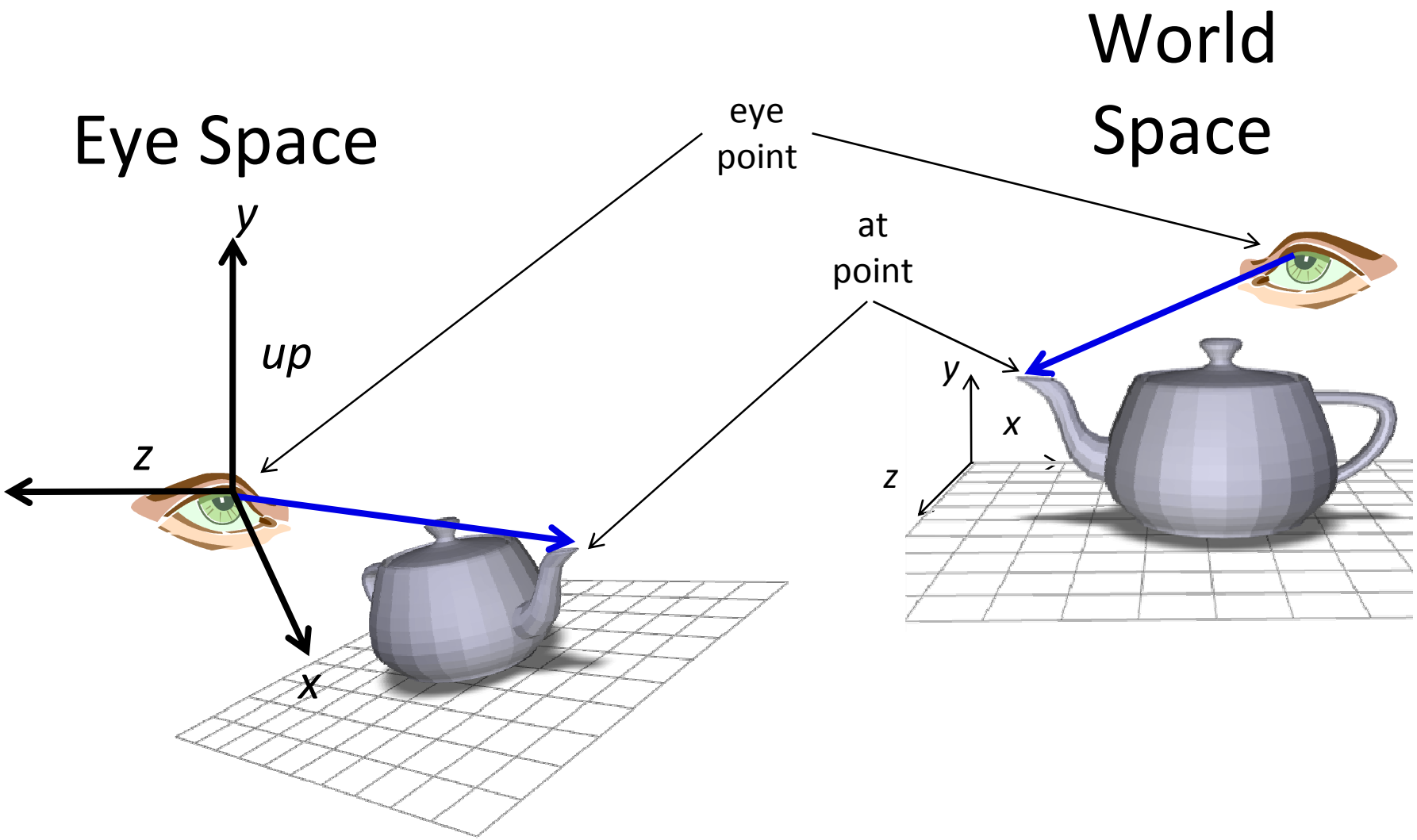




LookAt Transform

- การเซตมุมกล้องอย่างง่ายแบบหนึ่ง
- บอก
  - eye = ตำแหน่งของตา
  - at = ตำแหน่งที่ตามอง
  - up = ทิศทางด้านบน

การเปลี่ยนระบบพิกัดของ LookAt Transform



การคำนวณ LookAt Transform

- เวกเตอร์จาก **at** ไปยัง **eye** จะกลายเป็นแกน **z**
- เวกเตอร์ **up** จะกลายเป็นแกน **y** แต่ไม่ใช่ซะทีเดียว
- เราจะต้องทำให้เวกเตอร์ที่เป็นแกน **y** ตั้งฉากกับแกน **z**
- ทำอย่างไร?
  - คำนวณ **up** $\times$ **(eye-at)**
  - เวกเตอร์นี้ตั้งฉากกับ **up** (แกน **y**) และ **eye-at** (แกน **z**)
  - ฉะนั้นให้มันเป็นแกน **x**
  - แล้วคำนวณ **(eye-at)** $\times$ **x** แล้วให้มันเป็นแกน **y** แทน

การคำนวณ LookAt Transform

- กล่าวคือเราทำการคำนวณ

$$\mathbf{z} = \text{normalize}(\mathbf{at} - \mathbf{eye})$$
$$\mathbf{x} = \text{normalize}(\mathbf{up}) \times \mathbf{z}$$
$$\mathbf{y} = \mathbf{z} \times \mathbf{x}$$
$$M = \begin{bmatrix} | & | & | & | \\ \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{eye} \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- เมตริกซ์ M จะทำการเปลี่ยนระบบพิกัดจาก eye space เป็น world space

การคำนวณ LookAt Transform

- ใน API เช่น OpenGL และ DirectX โปรแกรมจะทำการคำนวณ M^{-1}แทน M

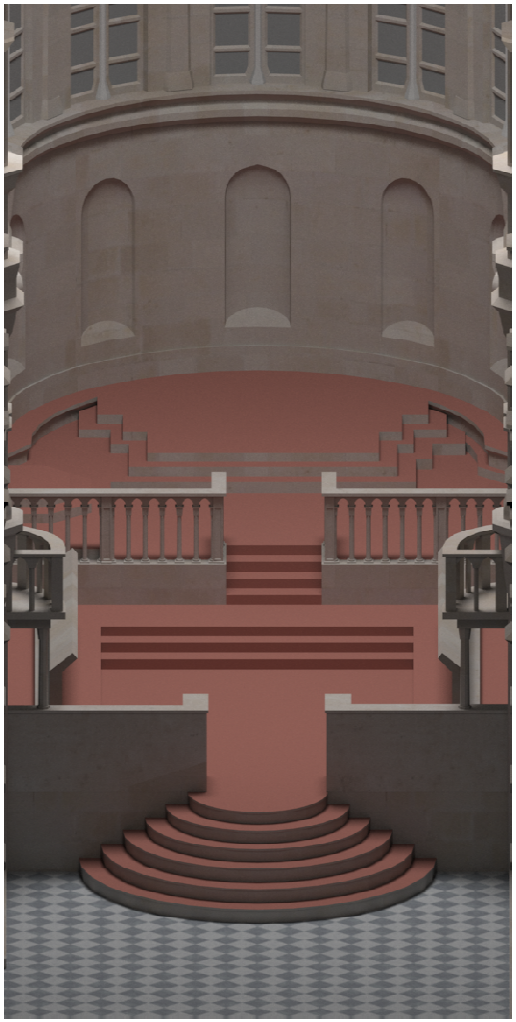
PERSPECTIVE PROJECTION

Perspective Projection

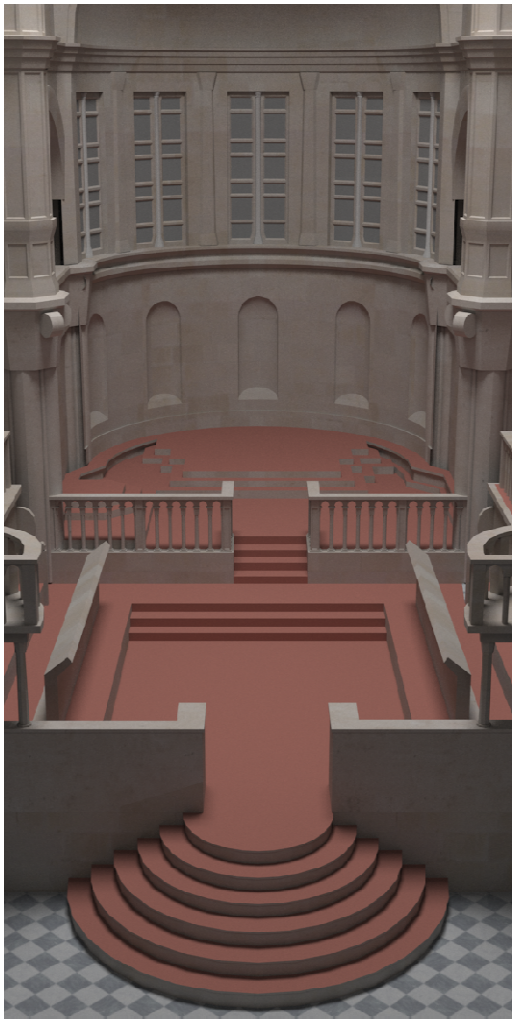
- ปริมาตรของบริเวณที่เห็นเป็น frustum (ปีระมิดยอดตัด)
- มี foreshortening กล่าวคือ อะไรที่อยู่ใกล้จะเห็นใหญ่กว่า
- หลังจากฉายแล้ว เส้นขนานอาจจะไม่ขนานกันเหมือนเดิม
- ให้ความเป็นสามมิติ เพราะเหมือนกับที่ตาคนทำงาน ทำให้เหมือนเข้าไปอยู่ในฉากจริงๆ
- ใช้กับโปรแกรมทางความบันเทิง

Perspective Projection (ต่อ)

Perspective Projection (cont.)



orthographic

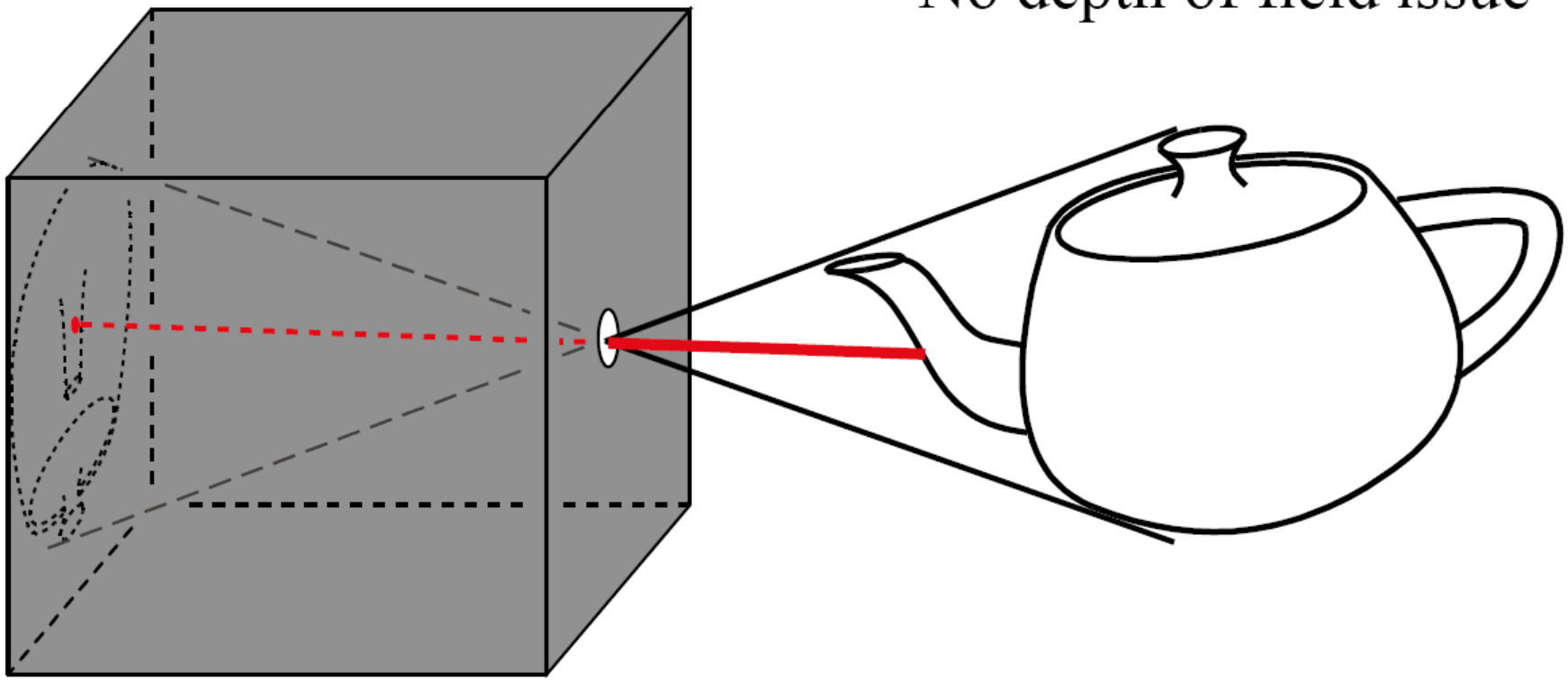


perspective

Pinhole camera

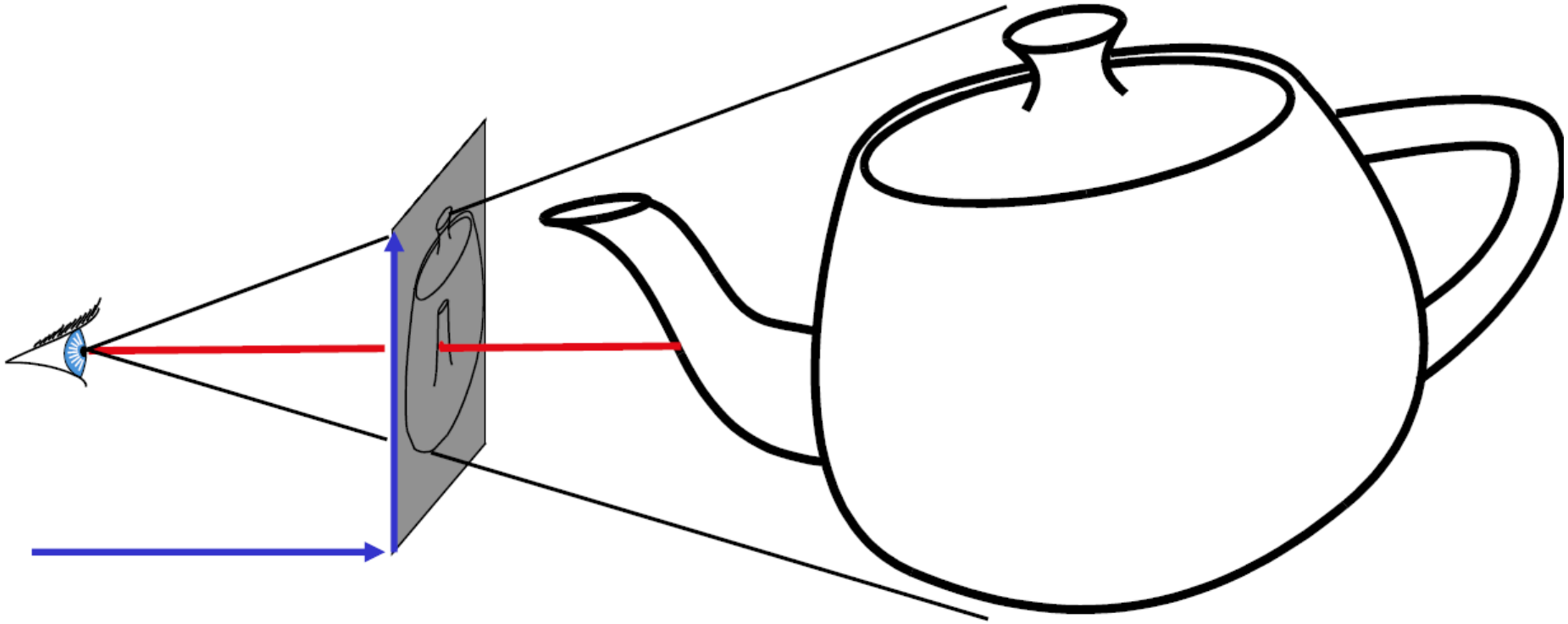
- Box with a tiny hole
- Inverted image
- Similar triangles

- Perfect image if hole infinitely small
- Pure geometric optics
- No depth of field issue



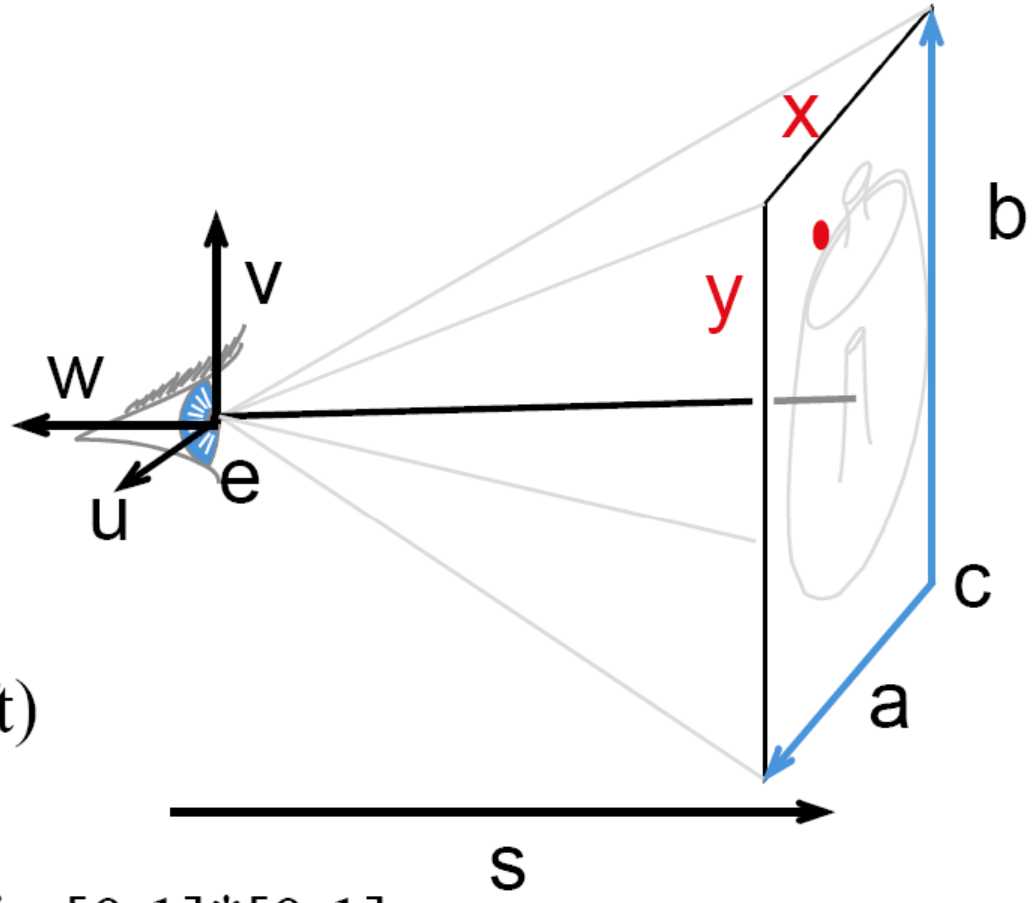
Simplified pinhole camera

- Eye-image pyramid (frustum)
- Note that the distance/size of image are arbitrary



Camera description

- Eye point e
- Orthobasis u, v, w
- Image distance s
- Image rectangle (u0, v0, u1, v1)



- Deduce c (lower left)
- Deduce a and b
- Screen coordinates in [0,1]*[0,1]
- A point is then c + x a +y b

Code

```
class Camera
{
public:
    Camera(const Xform &world2camera);
    virtual ~Camera();
    virtual Ray gen_ray(float sx, float sy) const = 0;
public:
    Xform world2camera;
};
```

Code

```
class PerspectiveCamera : public Camera
{
public:
    PerspectiveCamera(const Xform &_light2camera,
        float _fovy, float _aspect, float _hither, float _yon);
    virtual ~PerspectiveCamera();

    virtual Ray gen_ray(float sx, float sy) const;

public:
    float fovy;
    float aspect;
    float hither;
    float yon;
};
```

Code

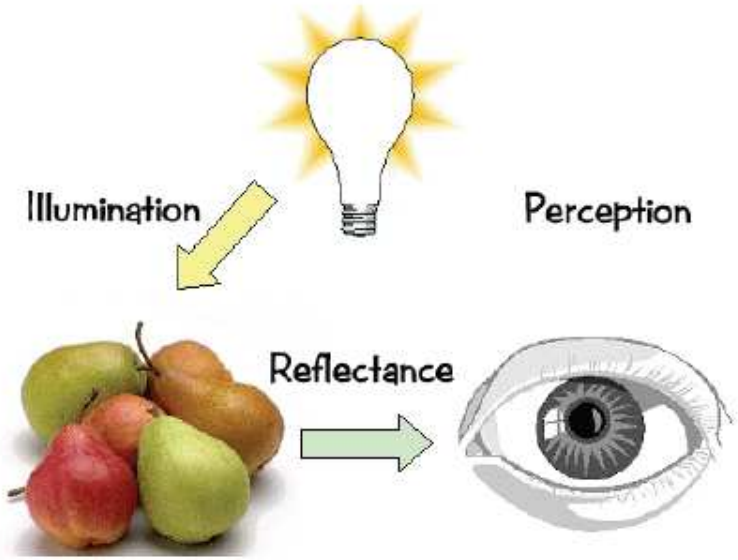
```
Ray PerspectiveCamera::gen_ray(float sx, float sy) const
{
    float scale_y = (float)tan(fovy / 2.0f);
    float scale_x = aspect * scale_y;
    Float3 d = Float3(scale_x * sx, scale_y * sy, -1);

    d = normalize(world2camera.mi.multiply_vector(d));
    Float3 o =
  world2camera.mi.multiply_point(Float3(0,0,0));

    return Ray(o, d, hither, yon);
}
```

Illumination & Reflectance

February 6, 2007

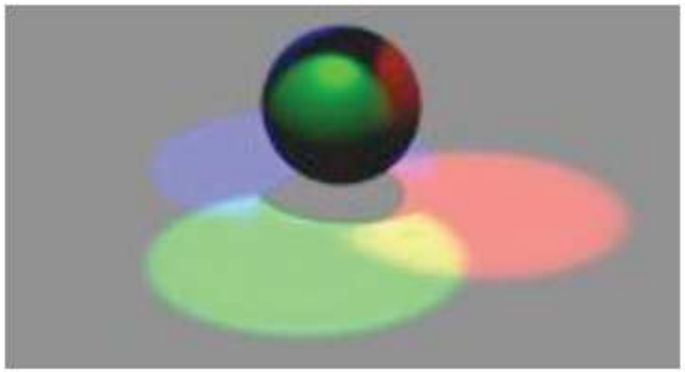
Making Pictures in the Real World

- Light leaves a light source and travels in a straight line through space.
- Light *strikes*, reflects from, and possibly passes through objects.
- Light ultimately arrives at a viewer (the eye or a camera).



Two Components of Illumination

- Light sources with:
  - Emittance spectrum (color)
  - Geometry (position and direction)
  - Directional attenuation (falloff)
- Surface properties with:
  - Reflectance spectrum (color)
  - Geometry (position, orientation, and micro-structure)
  - Absorption

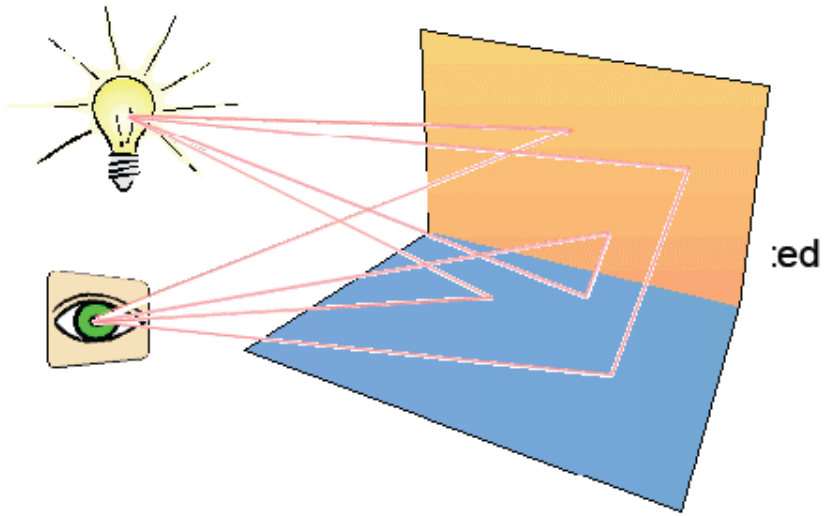


Computer Graphics Jargon

- **Illumination:** the transport of energy from light sources between points via direct and indirect paths
- **Lighting**: the process of computing the light intensity reflected from a specific 3-D point
- **Shading**: the process of assigning a color to a pixel based on the illumination in the scene

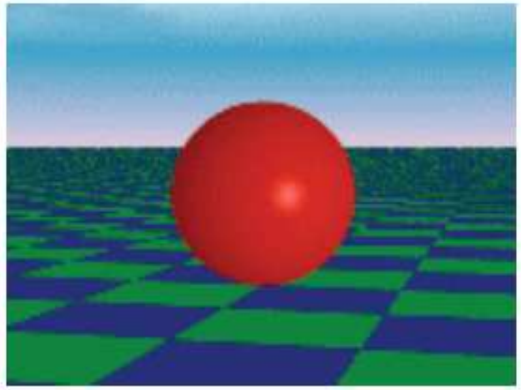
Direct and Global Illumination

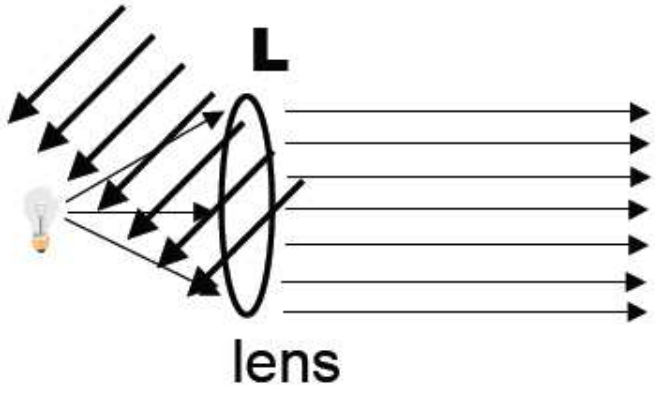
- Direct illumination: A surface point receives light directly from all light sources in the scene
  - Computed by the local illumination model
  - Determine which light sources are visible
- Global illumination: A surface point receives light after the light rays interact with other objects in the scene



Directional Light Sources

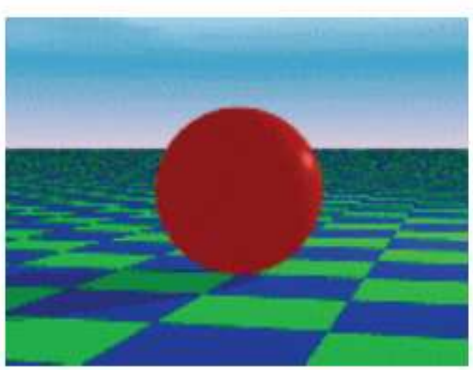
- All of the rays from a directional light source have a common direction
- The direction is a constant at every point in the scene
- It is as if the light source was infinitely far away from the surface that it is illuminating
- Examples?

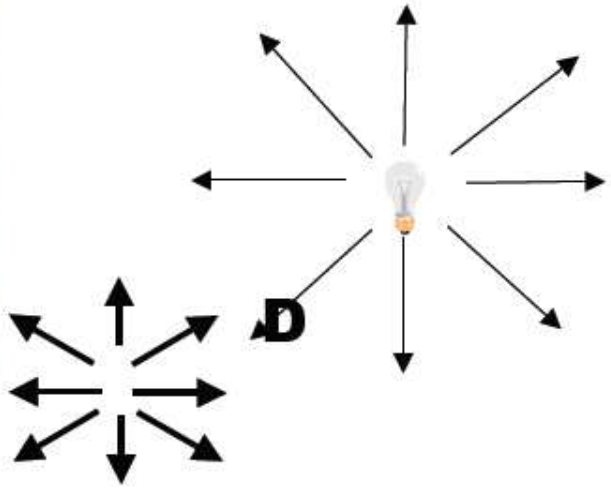




Point Light Sources

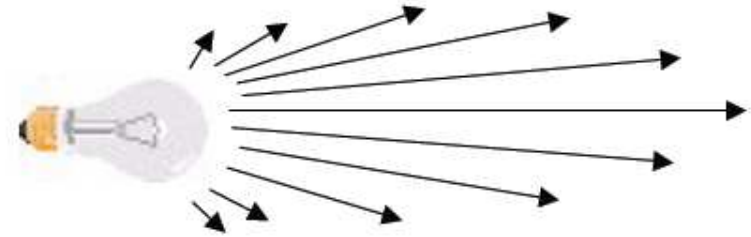
- The rays emitted from a point light radially diverge from the source
- Direction to the light changes at each point
- Examples?



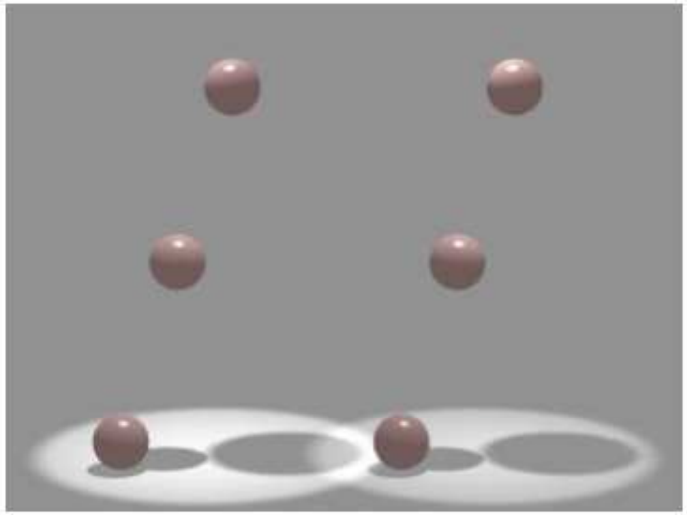


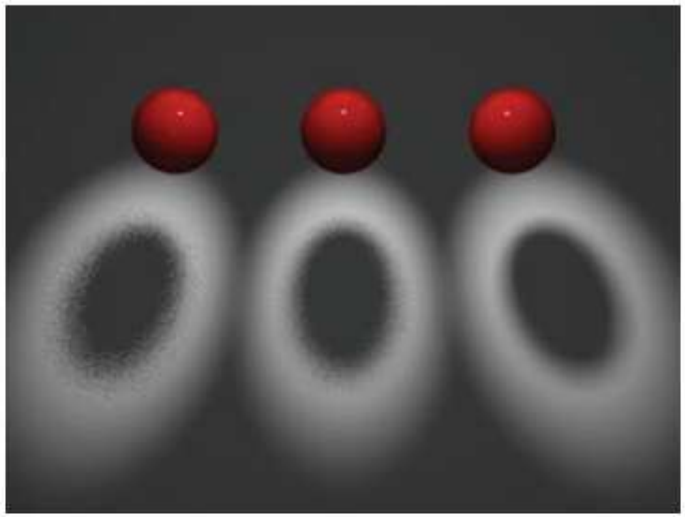
Other Light Sources

- Spotlights



- Area light sources
  - Light source occupies a 2D area (polygon)
  - Generates *soft* shadows.
- Extended light sources
  - Spherical light source
  - Generates *soft* shadows





Linearity of Light

=

+

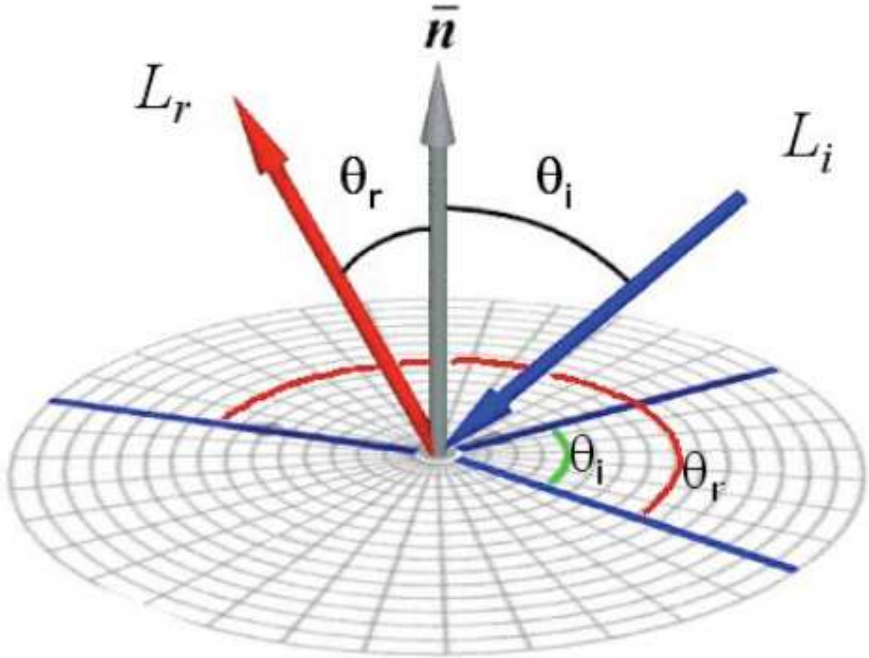
+

Paul Haeberli, Grafica Obscura

Linearity of Light

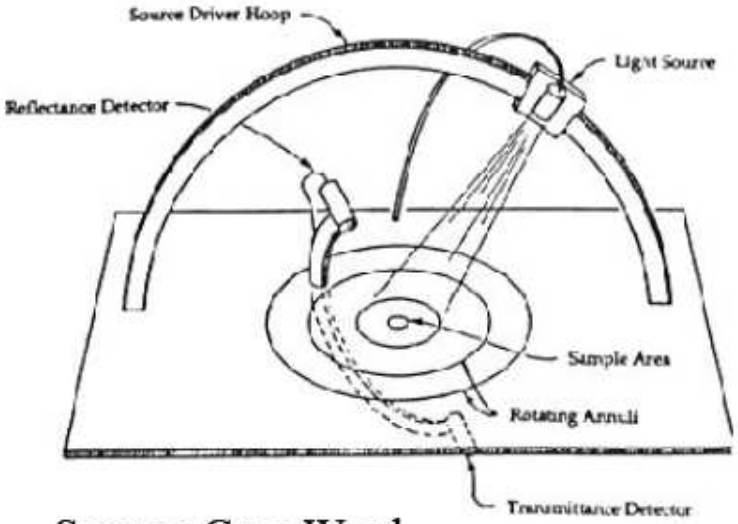
Reflectance Models

- Bi-Directional Reflection Distribution Function (BRDF)
- Difficult to measure in practice
- Often modeled analytically



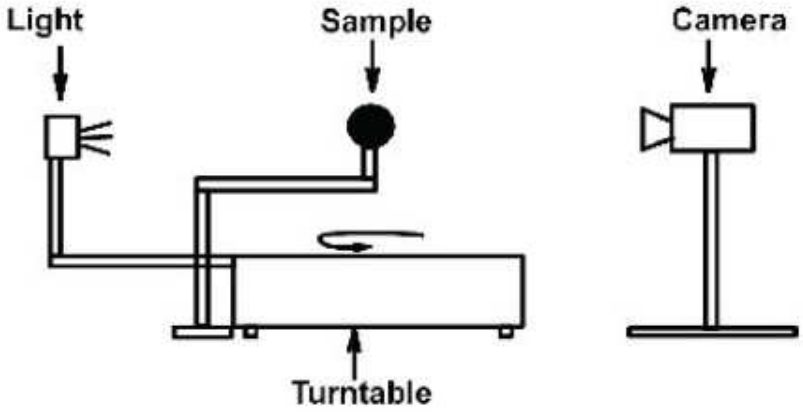
How do we obtain BRDFs?

- Gonio-reflectometer



Source: Greg Ward

- Steve Marschner / MERL

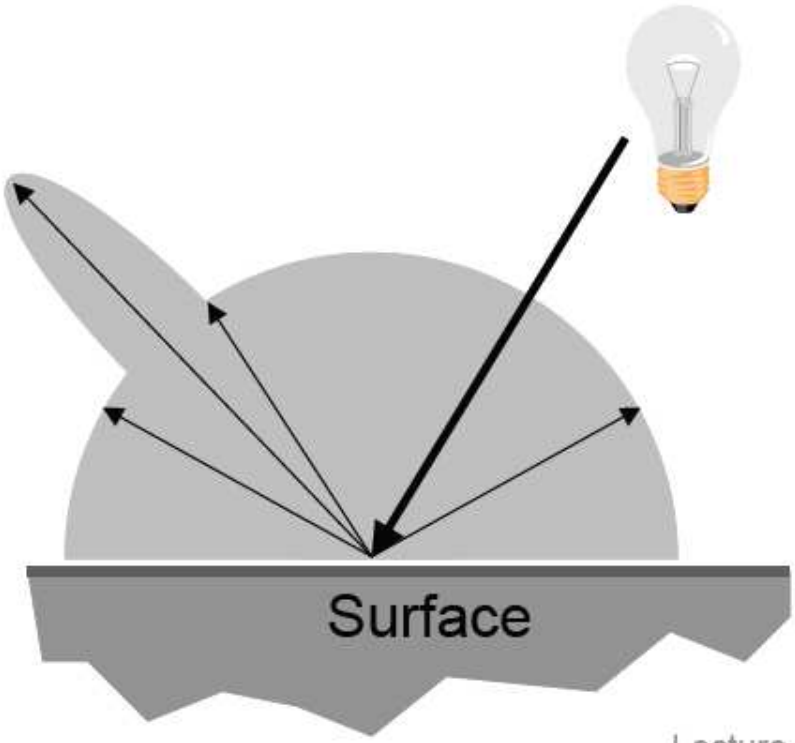


BRDF Models

- Physically-based models
  - Based on laws on physics
  - Derived from measurements
- Phenomenological (empirical) models
  - “Ad hoc” formulas that work
  - Easier to control by artist
- Data-driven models [Matusik et al. 2003]
  - Lot of data – require data reduction techniques
  - Very realistic, but harder to edit & manipulate

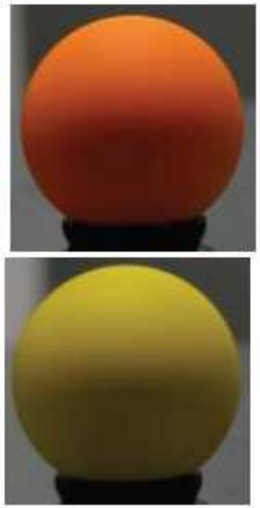
OpenGL Reflectance Model

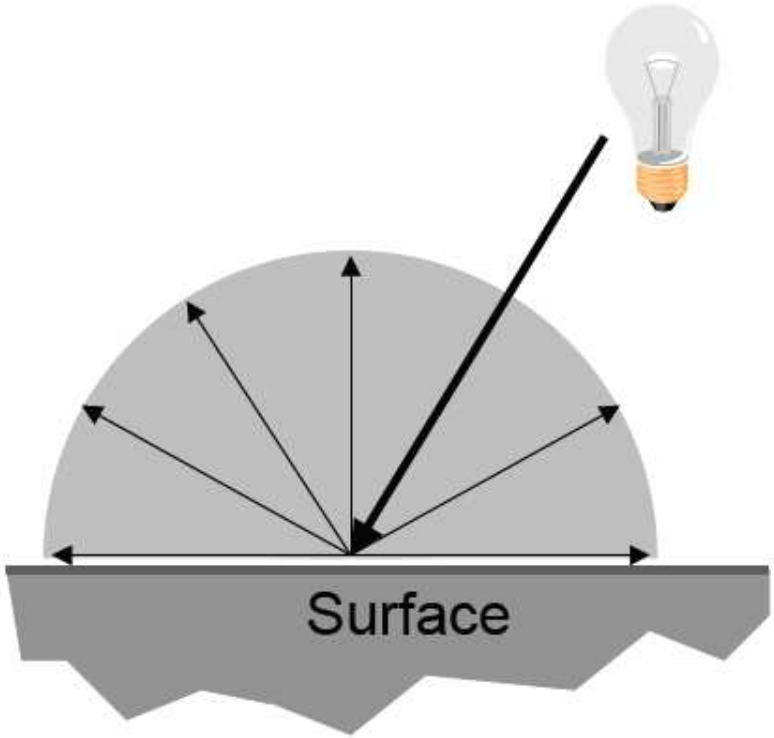
- Simple analytic model proposed by T. Phong:
  - diffuse reflection +
  - specular reflection +
  - “ambient”



Ideal Diffuse Reflectance

- Surface reflects light equally in all directions
- Why? Examples?





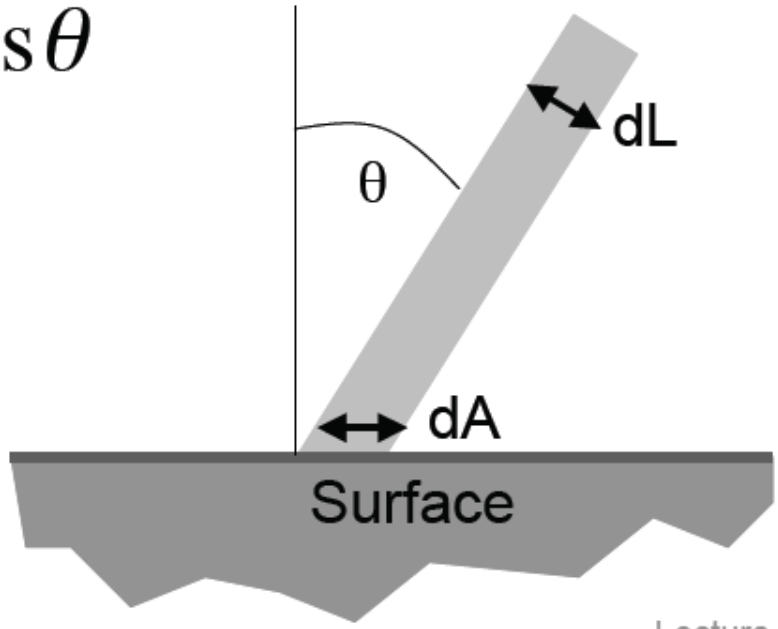
Lambert's Cosine Law

- Diffuse reflectance scales with cosine of angle

  cos θ is maximum when θ = 0°

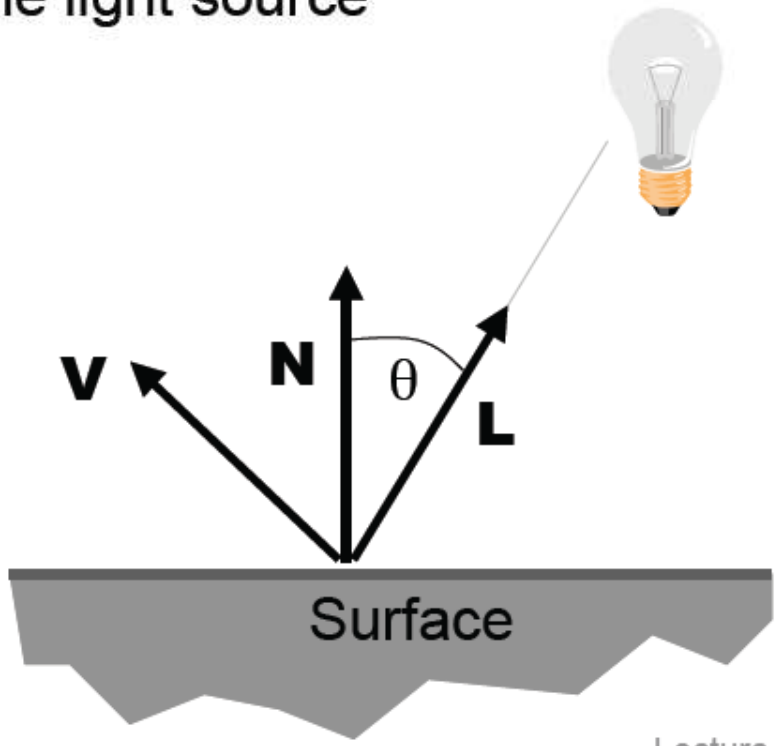
  cos θ is zero when θ = 90°

$$dL = dA\cos\theta$$



Geometric Ingredients

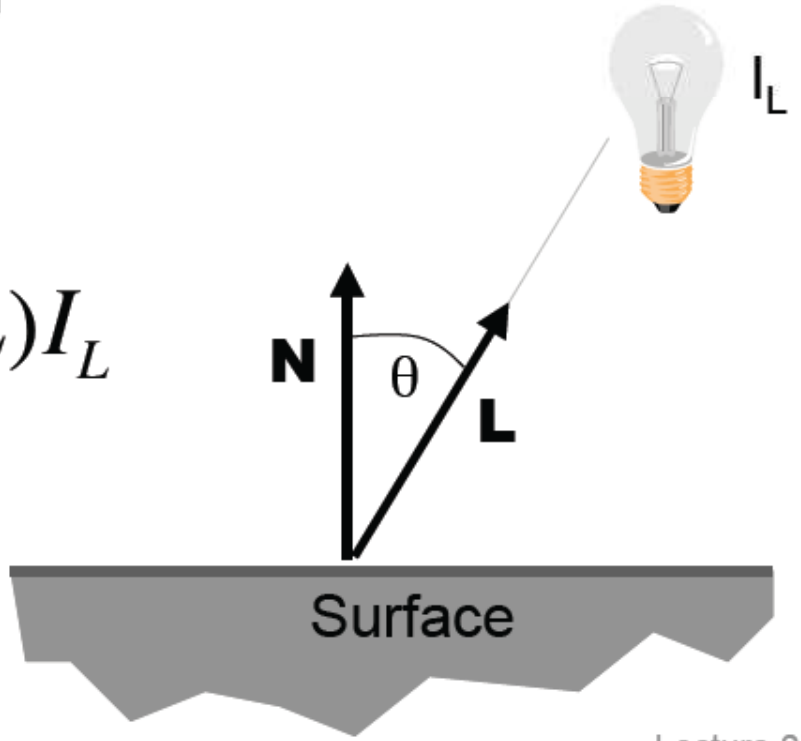
- P: Point on the surface
- **N**: Normal vector at the surface point
- **V**: Vector from P to the camera / eye
- **L**: Vector from P to the light source



Ideal Diffuse Reflectance

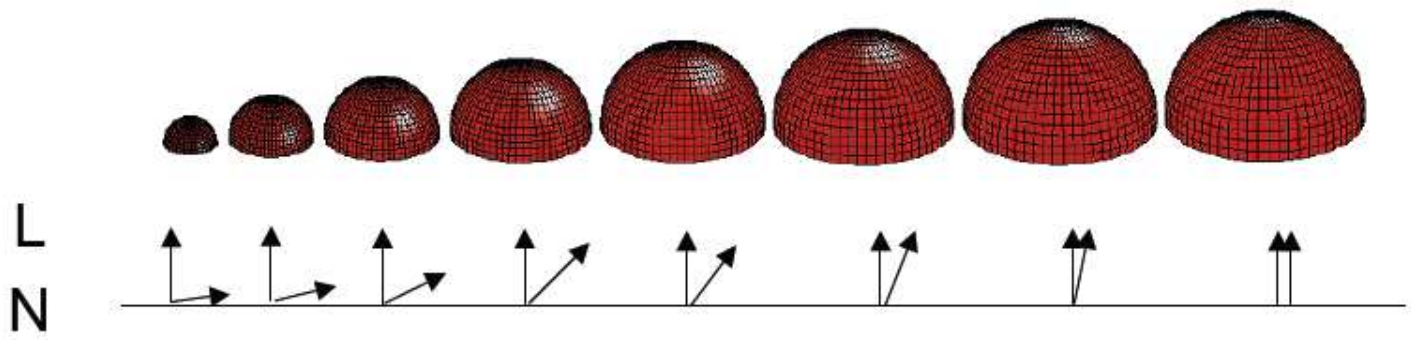
- Lambertian reflection model
  - I_L: The incoming light intensity
  - k_d: The diffuse reflection coefficient
  - N: Surface normal
  - L: Light direction

$$I_d = k_d (N \bullet L) I_L$$

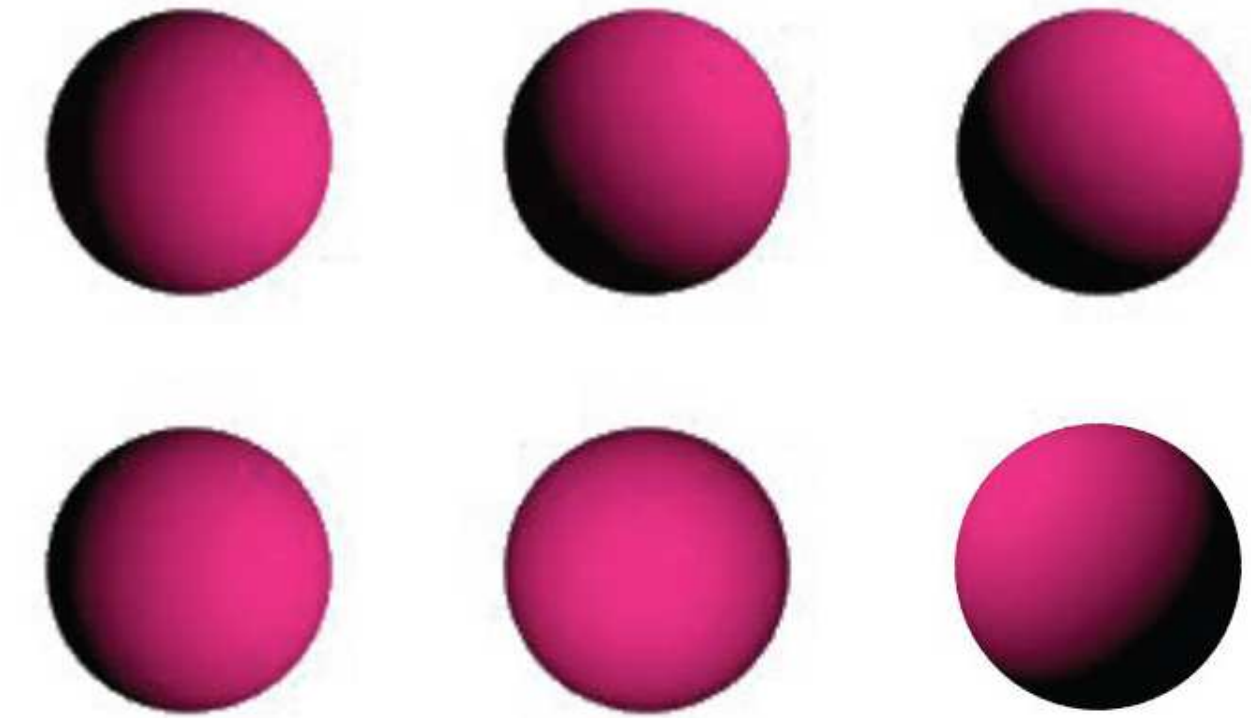


Ideal Diffuse Reflectance

- If **N** and **L** are facing away from each other, **N** • **L** becomes negative.
- Using max((**N** • **L**),0) makes sure that the result is zero.
  - From now on, we mean max() when we write •
- Do not forget to normalize your vectors for the dot product!

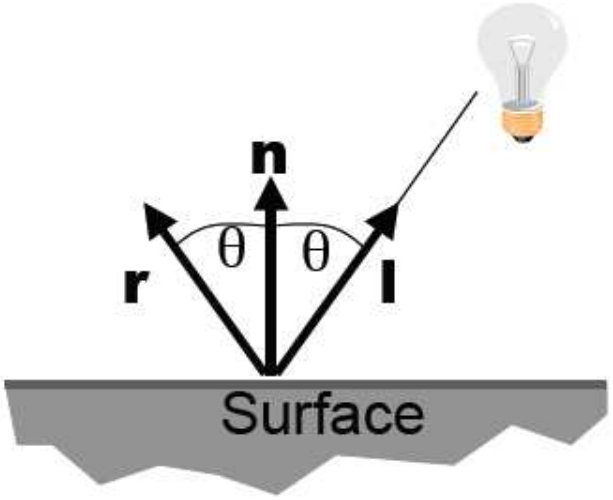


Diffuse Reflectance Example



Ideal Specular Reflectance

- Surface reflects light only at mirror angle
- Reflection is strongest near mirror angle
- Why? Examples?



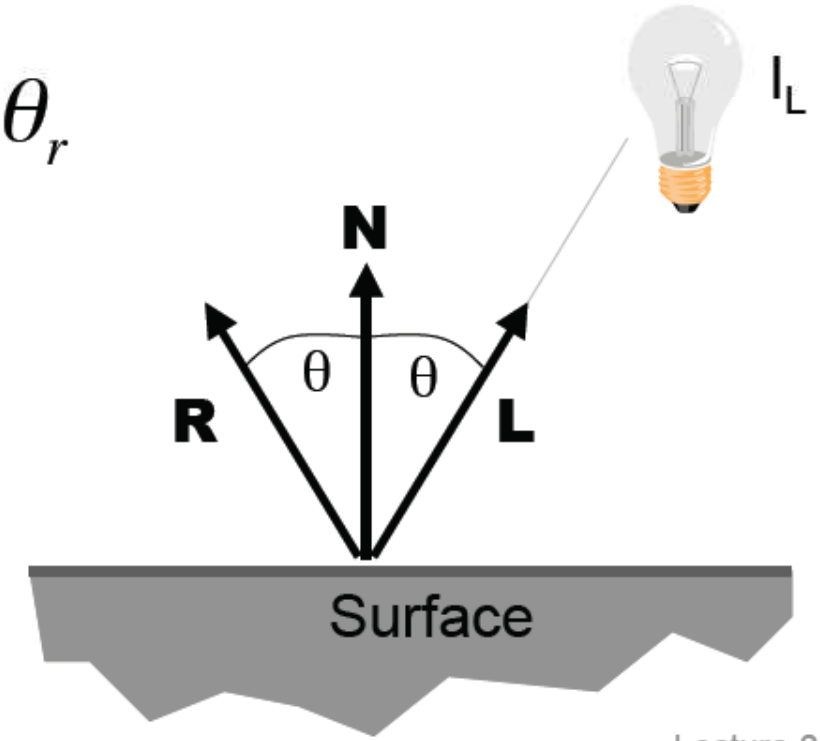
Ideal Specular Reflectance

- Special case of Snell's Law
  - The incoming ray, the surface normal, and the reflected ray all lie in a common plane

$$n_l \sin\theta_l = n_r \sin\theta_r$$

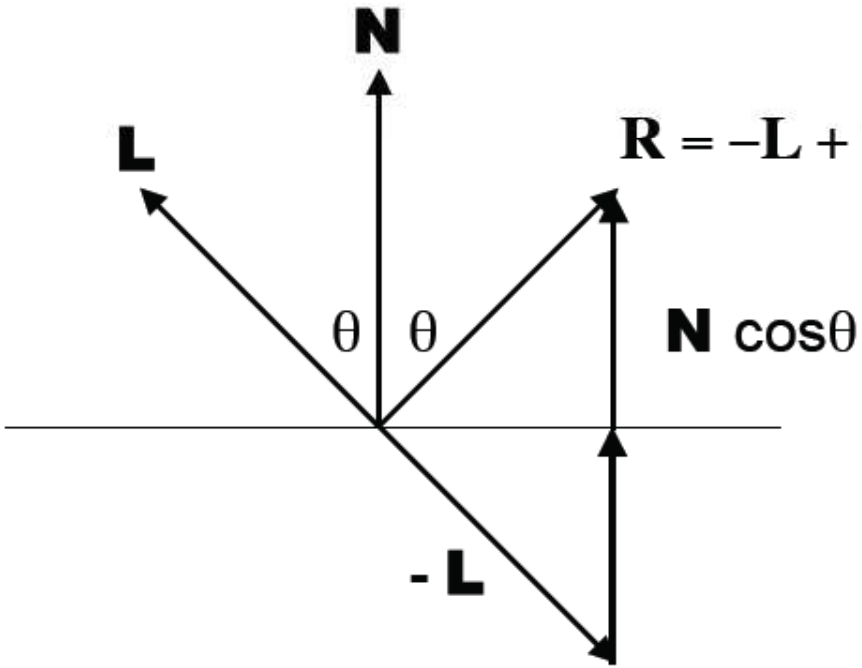
$$n_l = n_r$$

$$\theta_l = \theta_r$$



Reflection Vector R

- The vector R can be computed from the incident ray direction L and the surface normal N
- Note that all vectors have unit length



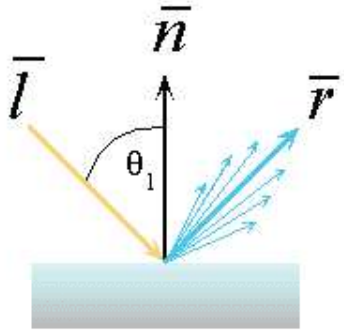
$$\mathbf{R} = -\mathbf{L} + 2\mathbf{N}cos\theta = -\mathbf{L} + 2(\mathbf{L} \bullet \mathbf{N})\mathbf{N}$$

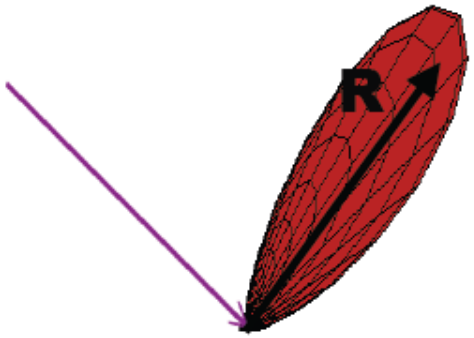
Non-ideal Reflectors

- Real materials tend to deviate significantly from ideal mirror reflectors

Non-ideal Reflectors

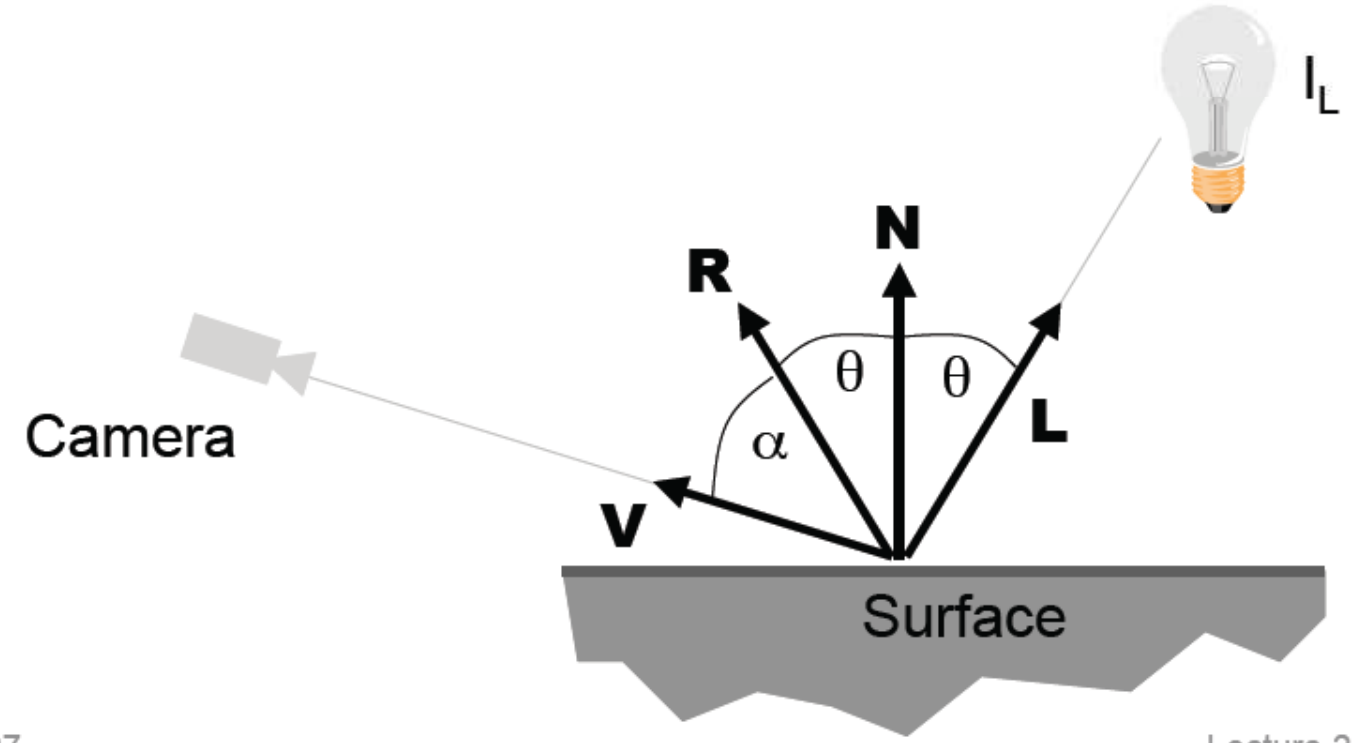
- Real materials deviate significantly from ideal reflectors
- Introduce an empirical model that is consistent with our experience
- The amount of reflected light is greatest in the direction of the perfect mirror reflection **R**
- The reflected light forms a “beam” pattern around this mirror direction





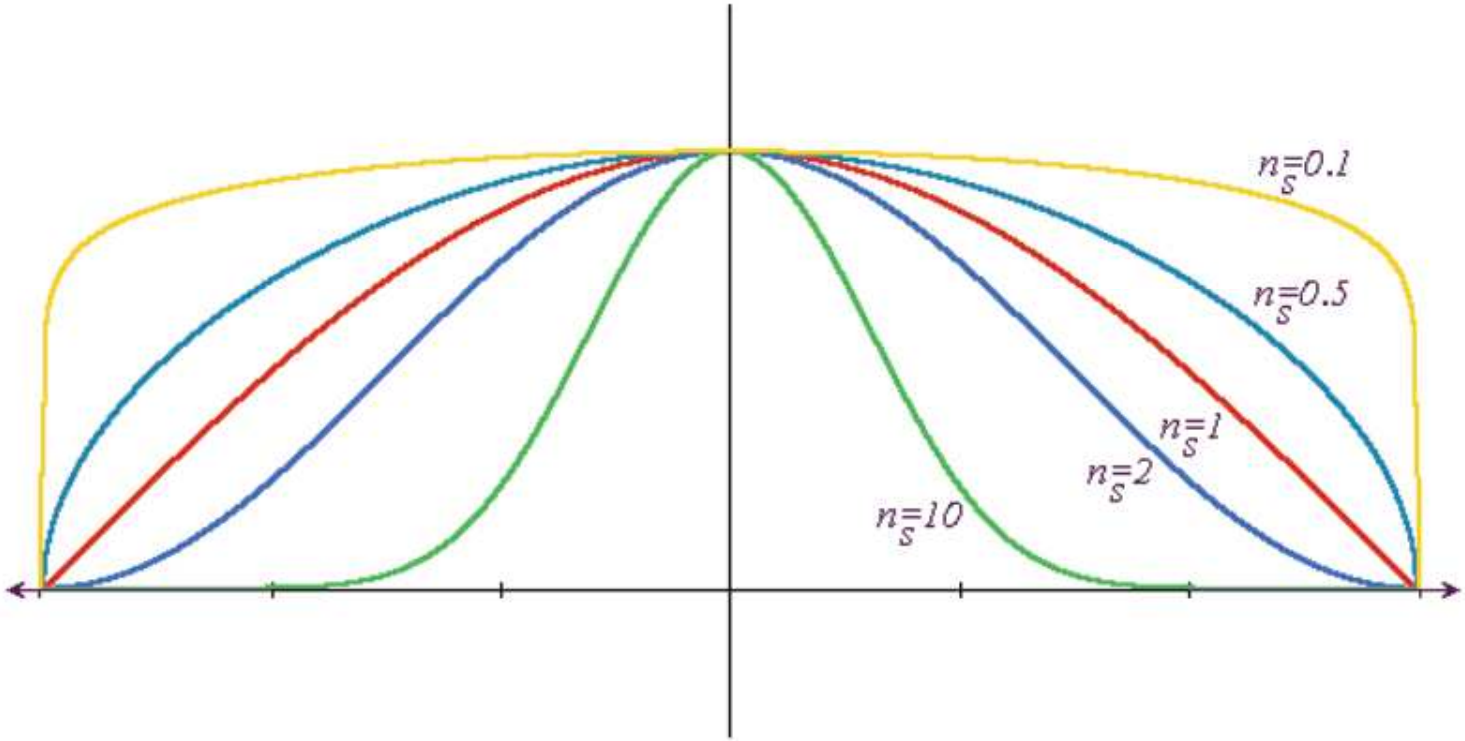
Phong Specular Reflection

$$I_s = k_s (V \bullet R)^n I_L$$



Effect of n

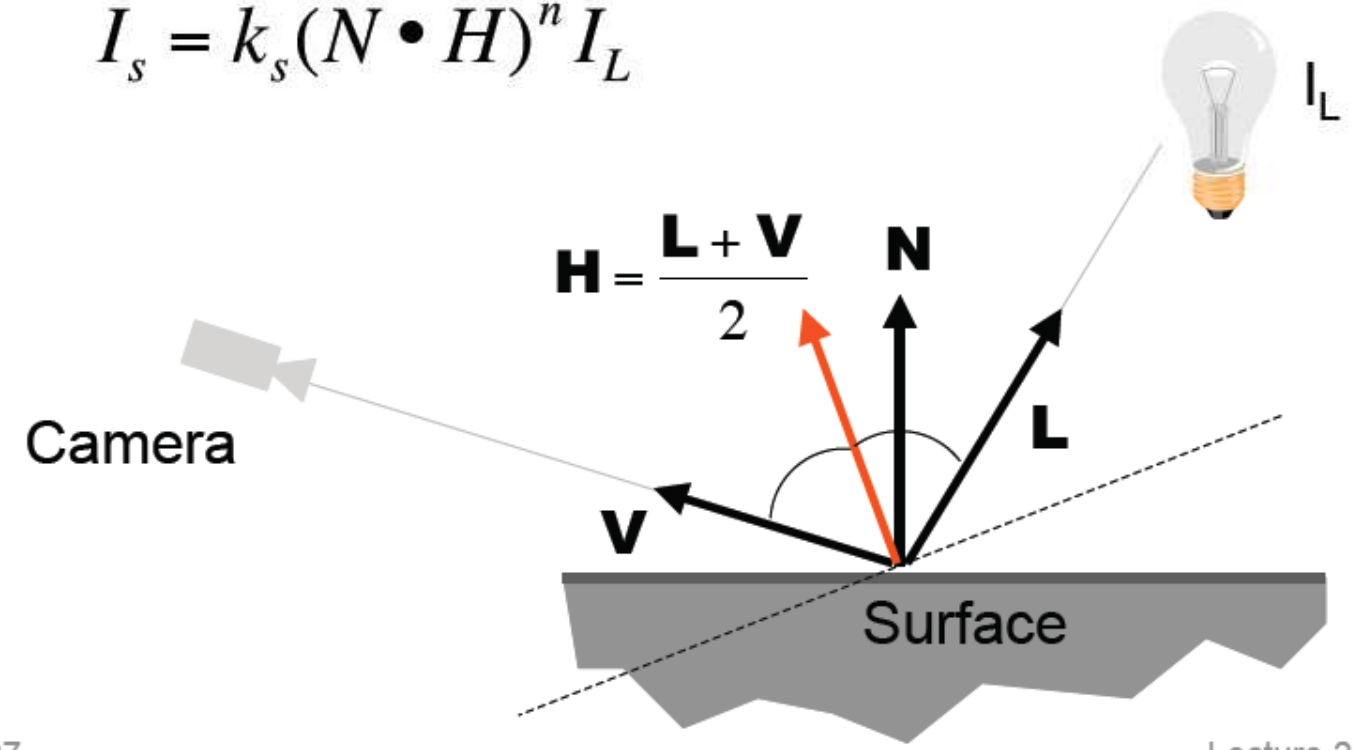
- The cosine lobe gets more narrow with increasing n.



Blinn & Torrance Variation

- Uses the halfway vector H between L and V
- H is the normal to the (imaginary) surface that maximally reflects light in the V direction

$$I_s = k_s(N \bullet H)^n I_L$$



Blinn & Torrance Variation

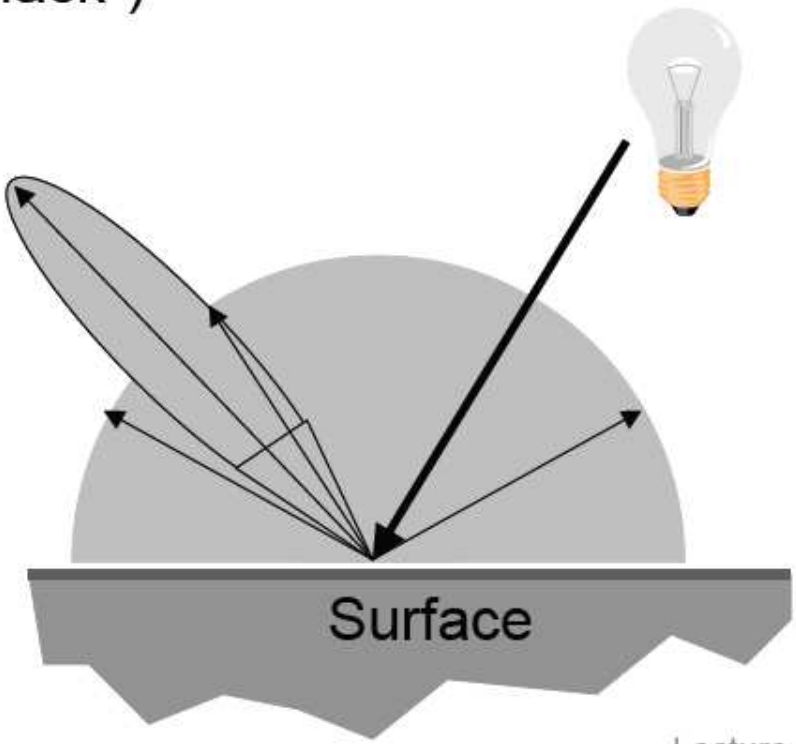
- No need to compute reflection vector **R** at every point
- I_s is a function only of **N**, if:
  - the viewer is very far away and **V** does not change for all points on the object (e.g., orthographic projection)
  - **L** does not change for all points on the object (e.g., directional lights)

The Phong Illumination Model

- Sum of three components:

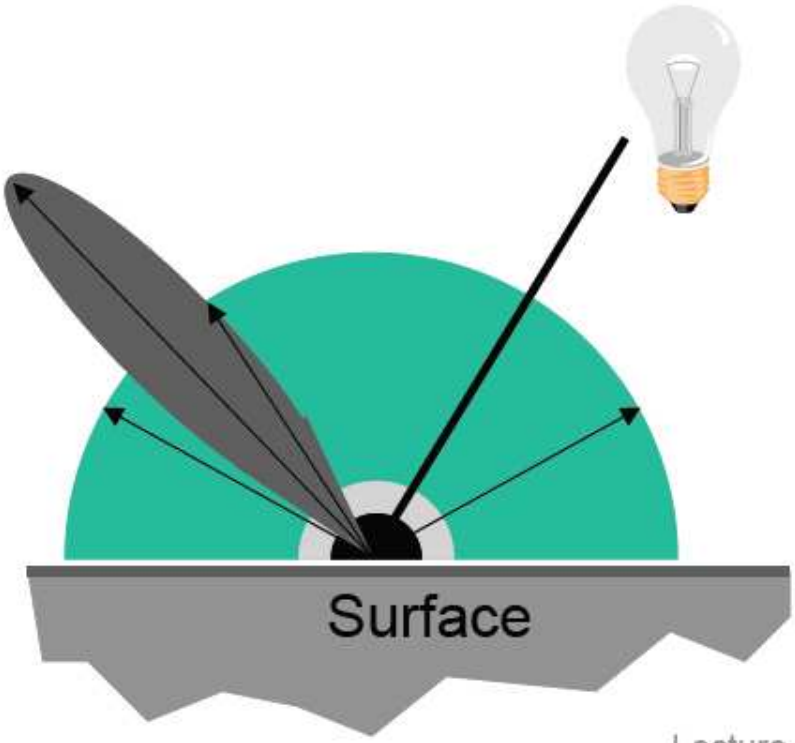
  diffuse reflection + specular reflection + "ambient"

- Ambient represents the reflection of all indirect illumination (aka a "hack")



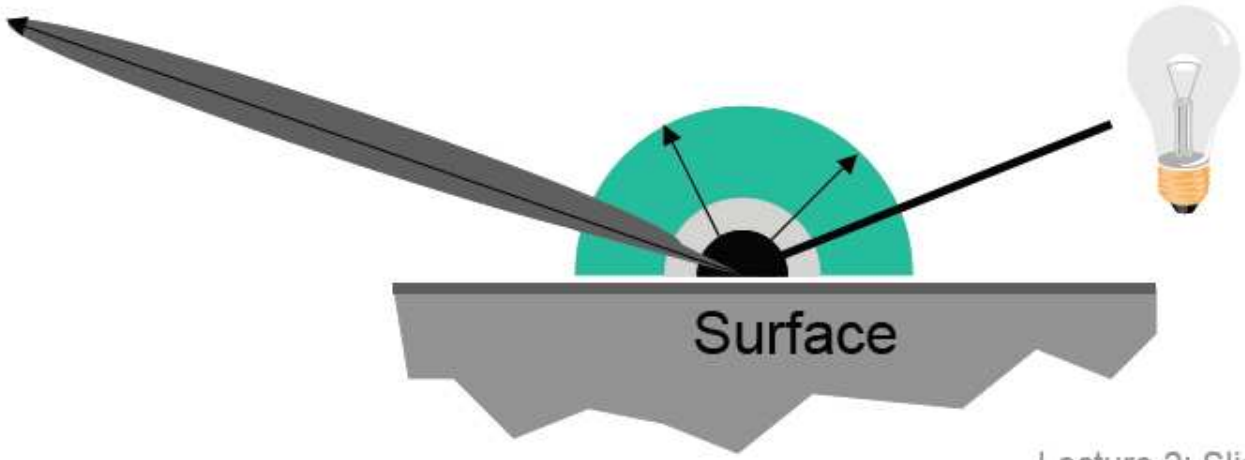
Phong Reflectance Model

- Simple analytical model:
  - diffuse reflection +
  - specular reflection +
  - "ambient"



Phong Reflectance Model

- Simple analytical model:
  - diffuse reflection +
  - specular reflection +
  - "ambient"

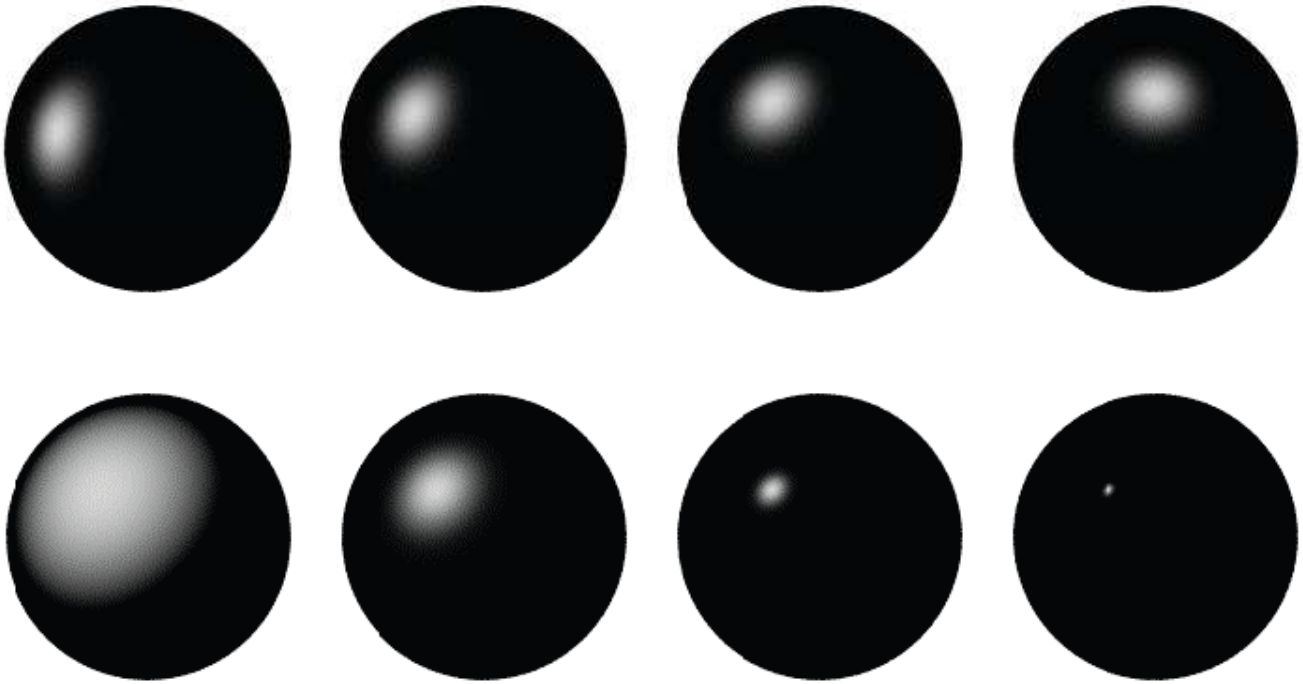


Phong Reflectance Model

Phong	$\rho_{ambient}$	$\rho_{diffuse}$	$\rho_{specular}$	ρ_{total}
$\phi_i = 60^\circ$				
$\phi_i = 25^\circ$				
$\phi_i = 0^\circ$				

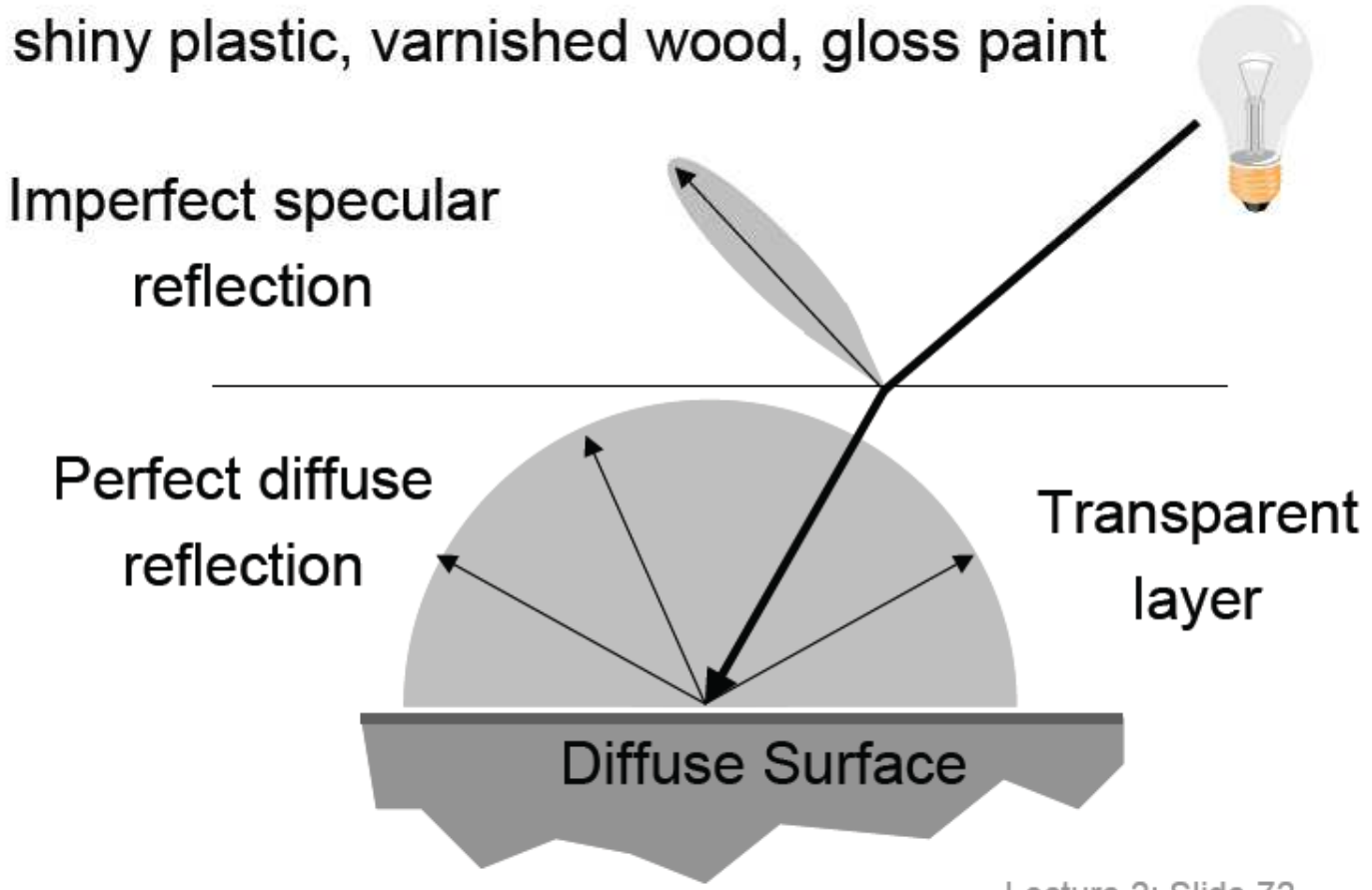
Phong Examples

- The direction of the light source and the n are varied



The Plastic Look

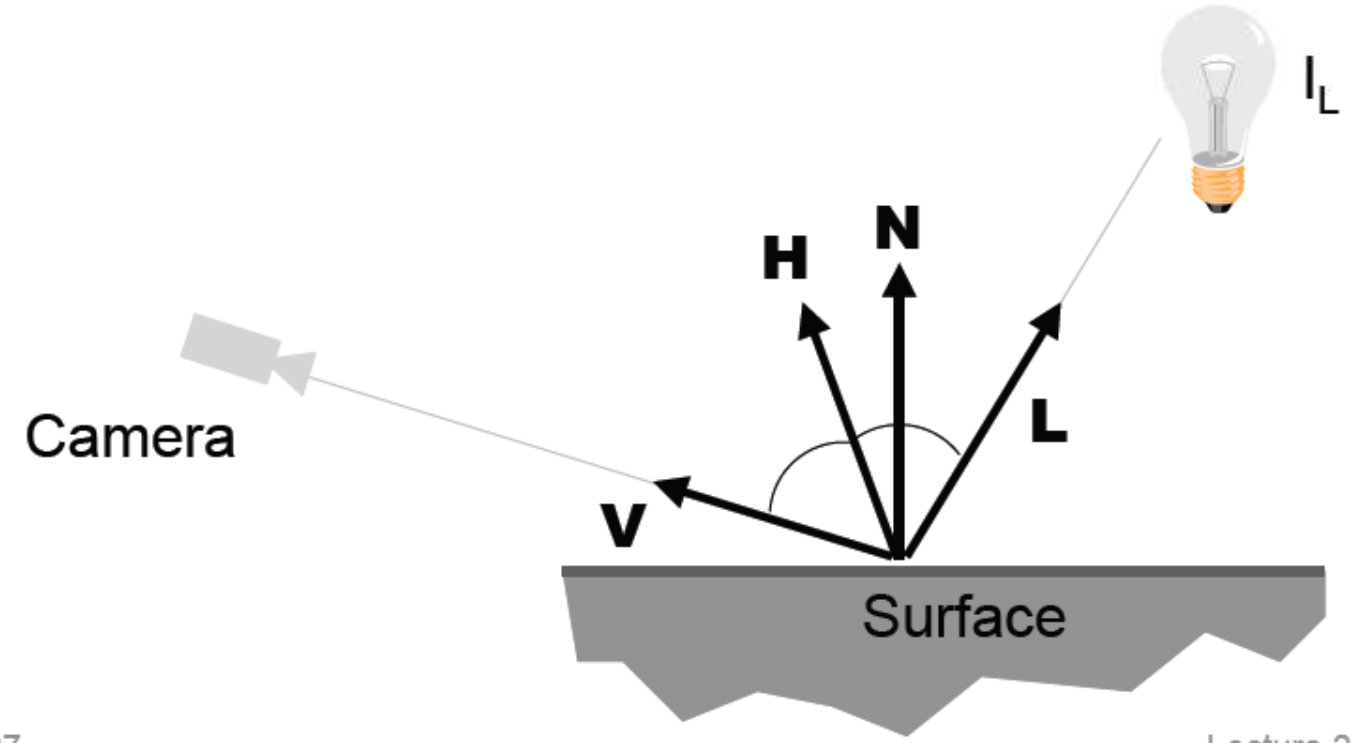
- The Phong illumination model is an approximation of a surface with a specular and a diffuse layer
  - E.g., shiny plastic, varnished wood, gloss paint



Phong Reflectance Model

- Single light source:

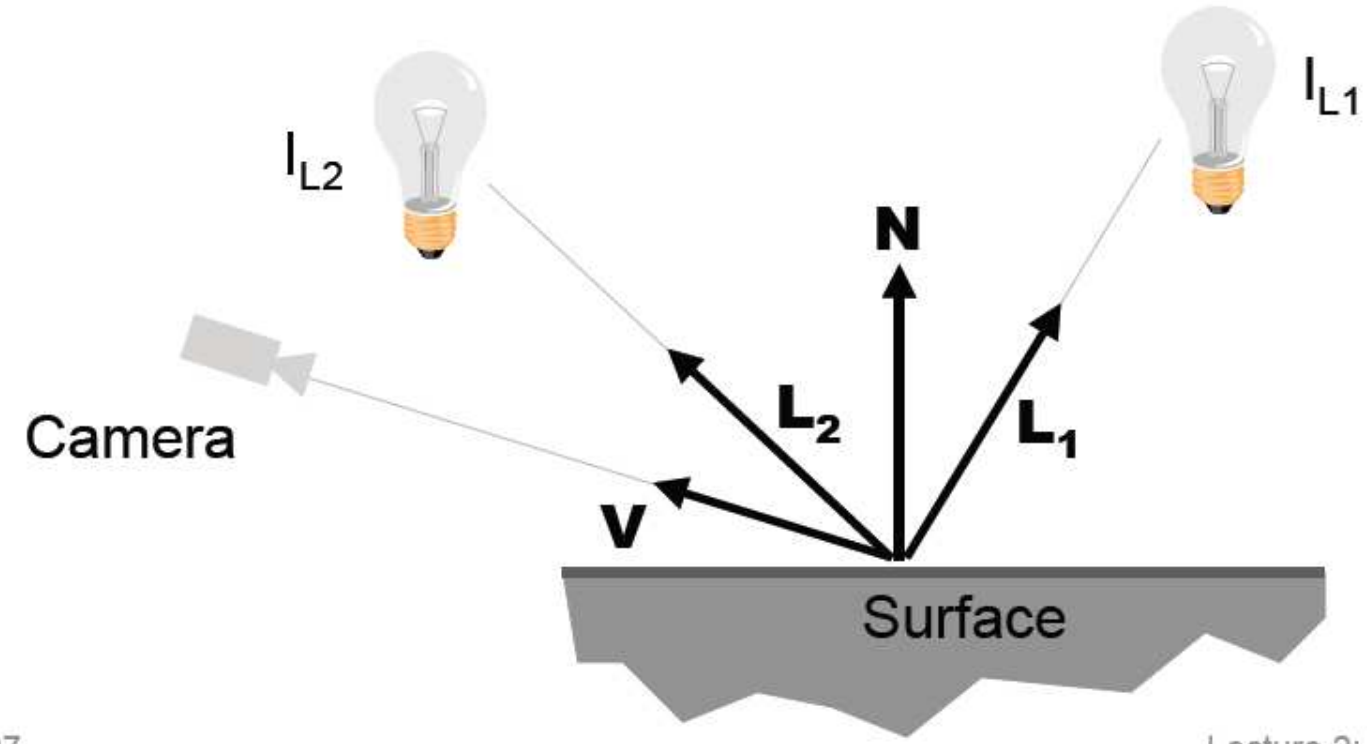
$$I = k_a I_a + k_d (N \bullet L) I_L + k_s (N \bullet H)^n I_L$$



Phong Reflectance Model

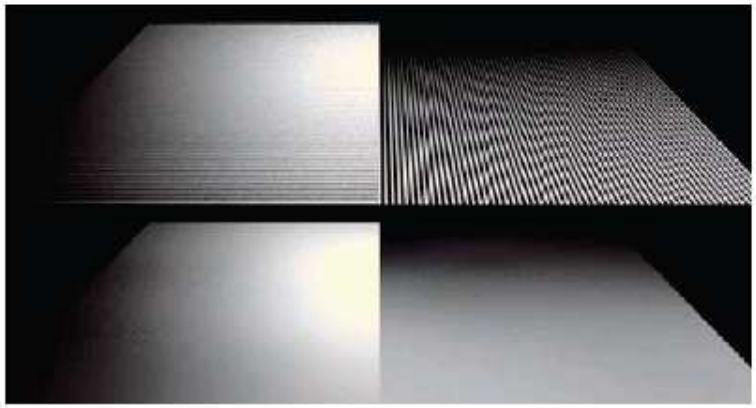
- Multiple light sources:

$$I = k_a I_a + \sum_i \left(k_d (N \bullet L_i) I_i + k_s (N \bullet H_i)^n I_i \right)$$



Anisotropic BRDFs

- Surfaces with strongly oriented micro-elements
- Examples:
  - Brushed metals,
  - Hair, fur, cloth, velvet



Source: Westin et.al 92

Ray Tracing

Courtesy of James Arvo and David Kirk. Used with permission.

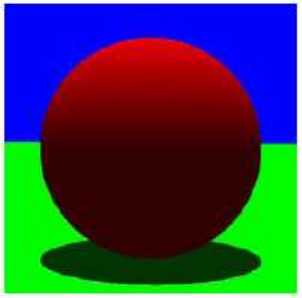
MIT EECS 6.837

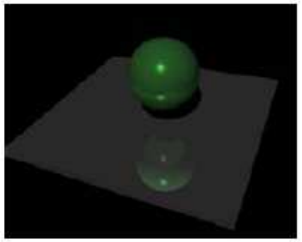
Frédo Durand and Barb Cutler

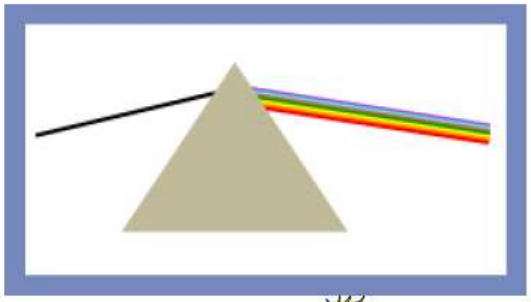
Some slides courtesy of Leonard McMillan

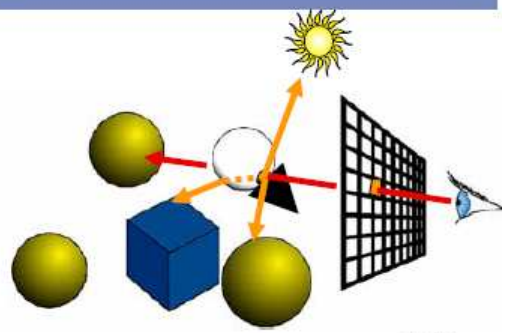
Overview of today

- Shadows
- Reflection
- Refraction
- Recursive Ray Tracing





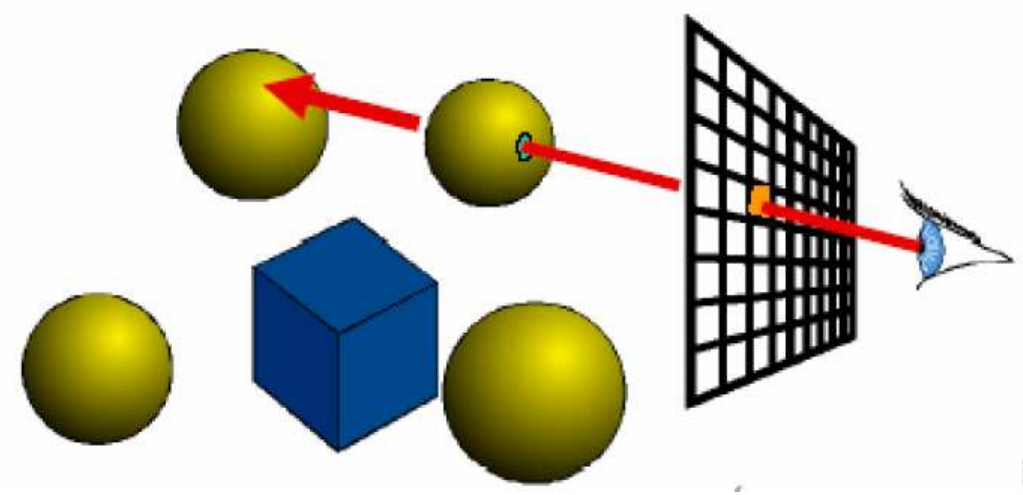




Ray Casting (a.k.a. Ray Shooting)

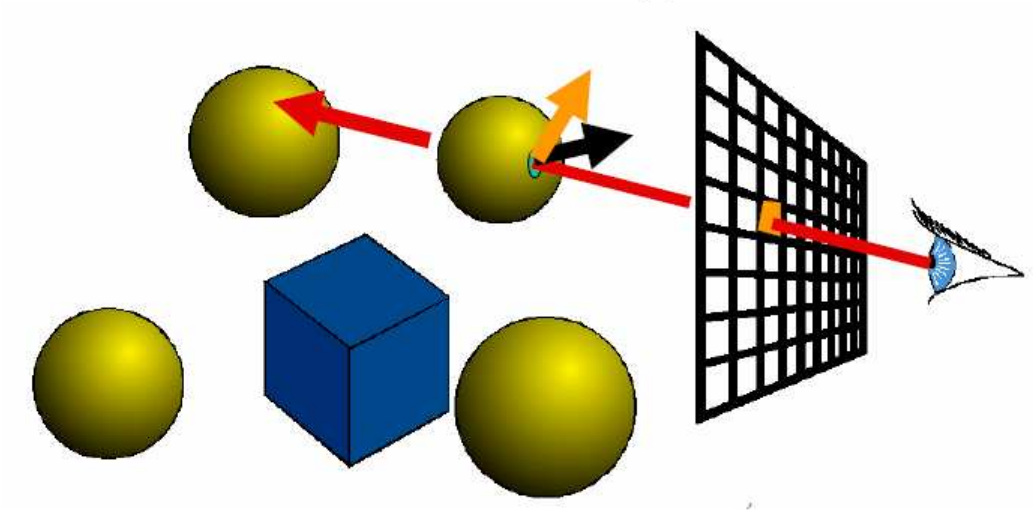
```
For every pixel (x,y)
   Construct a ray from the eye
   color[x,y]=castRay(ray)
```

- Complexity?
  - O(n * m)
  - n: number of objects, m: number of pixels



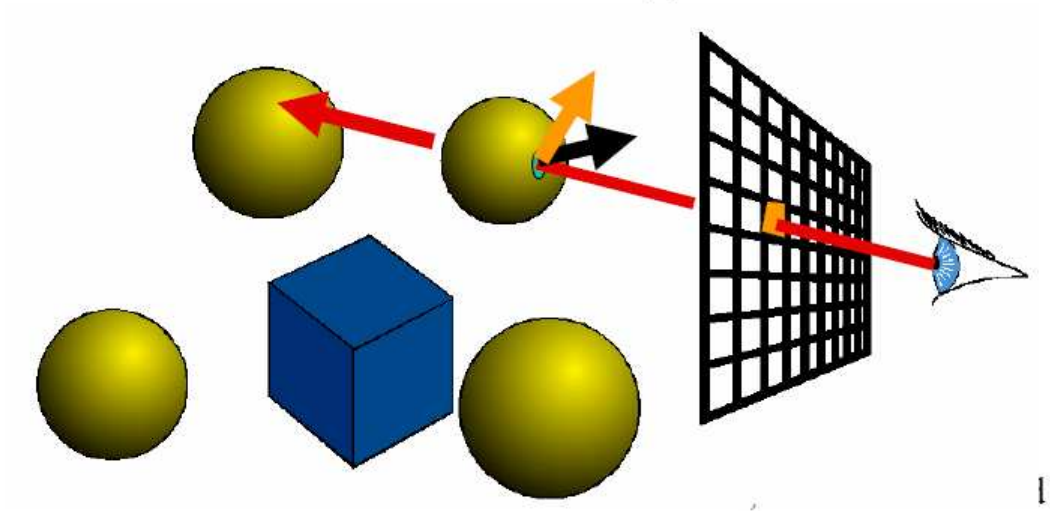
Ray Casting with diffuse shading

```
Color castRay(ray)
   Hit hit();
   For every object ob
       ob->intersect(ray, hit, tmin);
    Color col=ambient*hit->getColor();
    For every light L
       col=col+hit->getColorL()*L->getColor*
         L->getDir()->Dot3( hit->getNormal() );
    Return col;
```



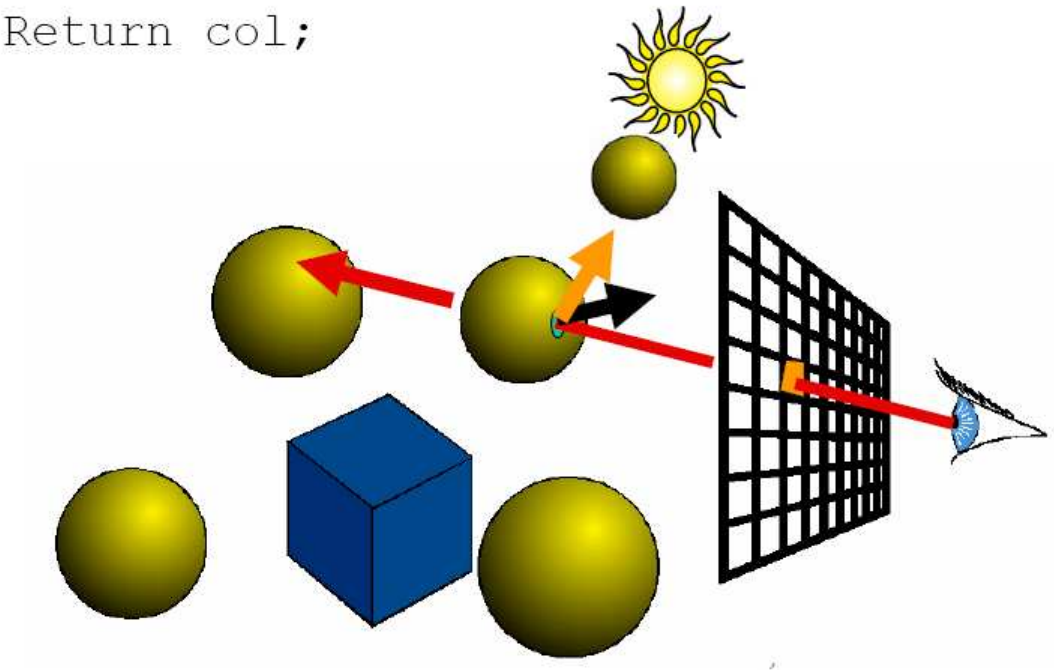
Encapsulating shading

```
Color castRay(ray)
   Hit hit();
   For every object ob
       ob->intersect(ray, hit, tmin);
    Color col=ambient*hit->getMaterial()->getDiffuse();
    For every light L
       col=col+hit->getMaterial()->shade
         (ray, hit, L->getDir(), L->getColor());
    Return col;
```



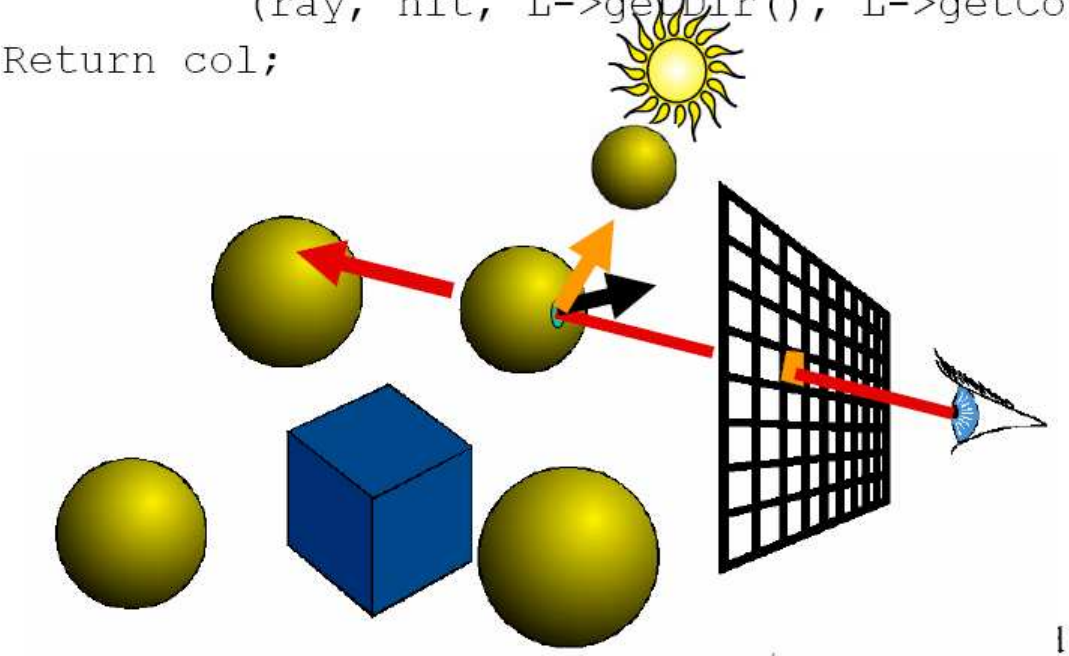
How can we add shadows?

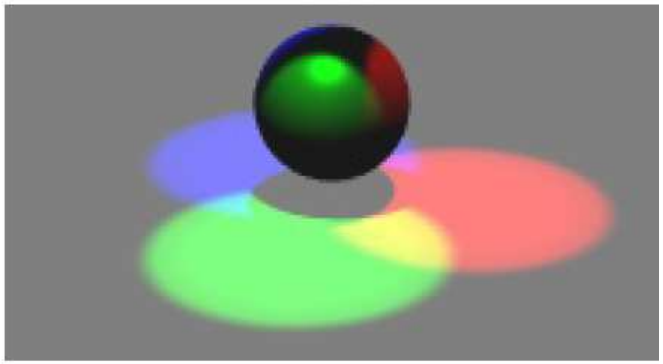
```
Color castRay(ray)
   Hit hit();
   For every object ob
       ob->intersect(ray, hit, tmin);
    Color col=ambient*hit->getMaterial()->getDiffuse();
    For every light L
       col=col+hit->getMaterial()->shade
         (ray, hit, L->getDir(), L->getColor());
    Return col;
```



Shadows

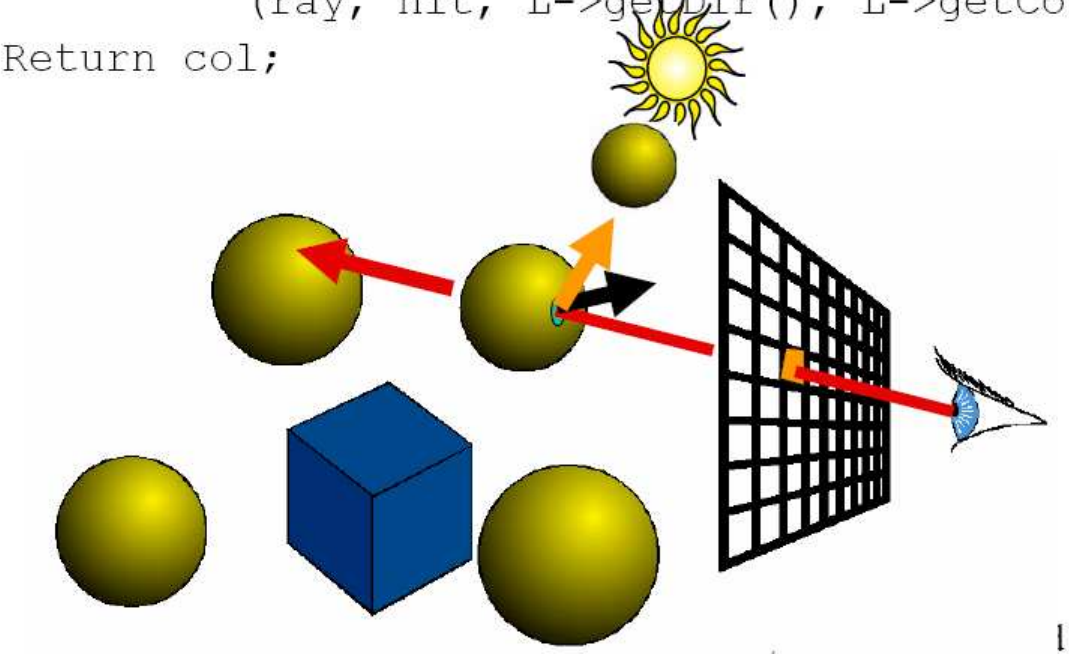
```
Color castRay(ray)
     Hit hit();
     For every object ob
            ob->intersect(ray, hit, tmin);
       Color col=ambient*hit->getMaterial()->getDiffuse();
      For every light L
          Ray ray2(hitPoint, L->getDir()); Hit hit2(L->getDist(),,)
          For every object ob
              ob->intersect(ray2, hit2, 0);
          If (hit->getT> L->getDist())
              col=col+hit->getMaterial()->shade
                (ray, hit, L->getDir(), L->getColor());
      Return col;
```

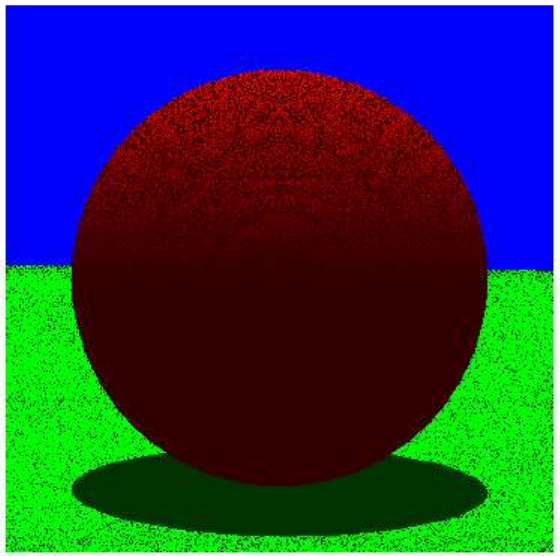




Shadows – problem?

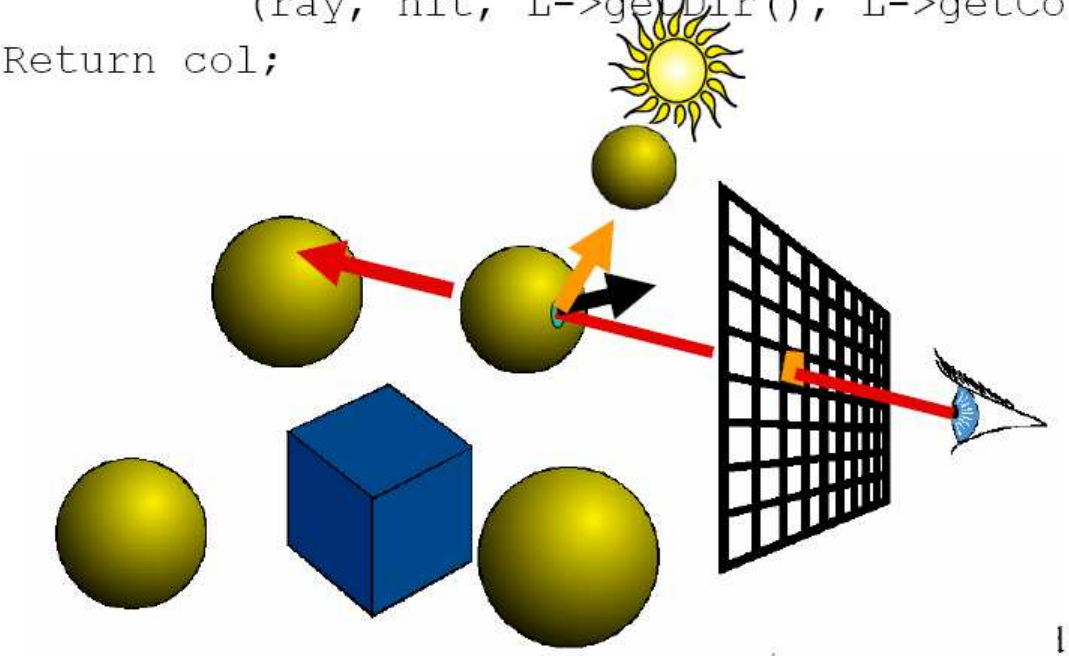
```
Color castRay(ray)
     Hit hit();
     For every object ob
            ob->intersect(ray, hit, tmin);
       Color col=ambient*hit->getMaterial()->getDiffuse();
      For every light L
          Ray ray2(hitPoint, L->getDir()); Hit hit2(L->getDist(),,)
          For every object ob
              ob->intersect(ray2, hit2, 0);
          If (hit->getT> L->getDist())
              col=col+hit->getMaterial()->shade
                (ray, hit, L->getDir(), L->getColor());
      Return col;
```

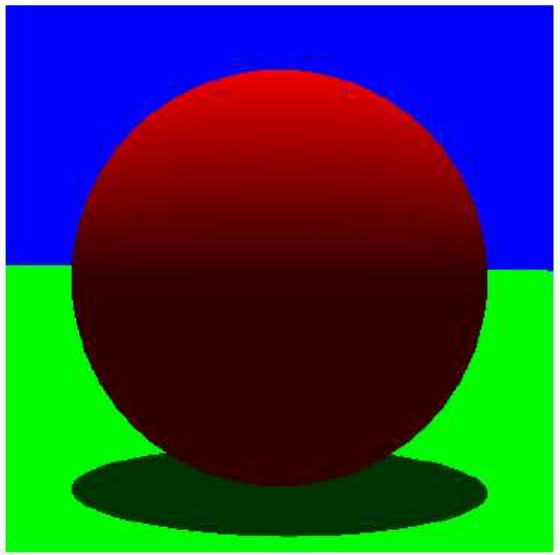




Avoiding self shadowing

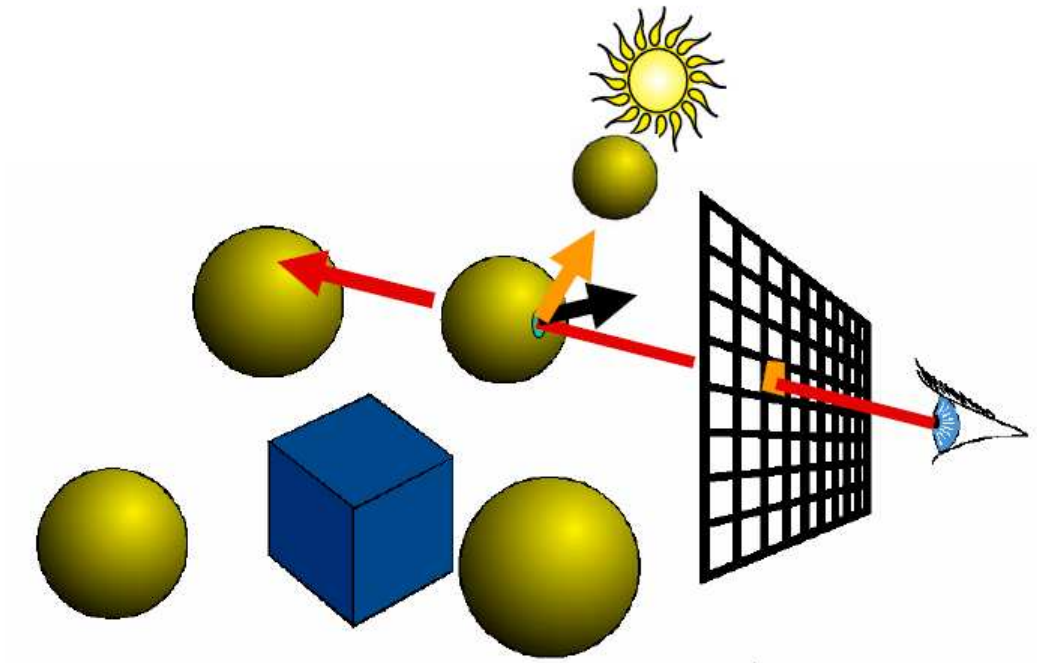
```
Color castRay(ray)
    Hit hit();
    For every object ob
          ob->intersect(ray, hit, tmin);
     Color col=ambient*hit->getMaterial()->getDiffuse();
     For every light L
         Ray ray2(hitPoint, L->getDir()); Hit hit2(L->getDist(),,)
         For every object ob
             ob->intersect(ray2, hit2, epsilon);
         If (hit->getT> L->getDist())
             col=col+hit->getMaterial()->shade
               (ray, hit, L->getDir(), L->getColor());
     Return col;
```





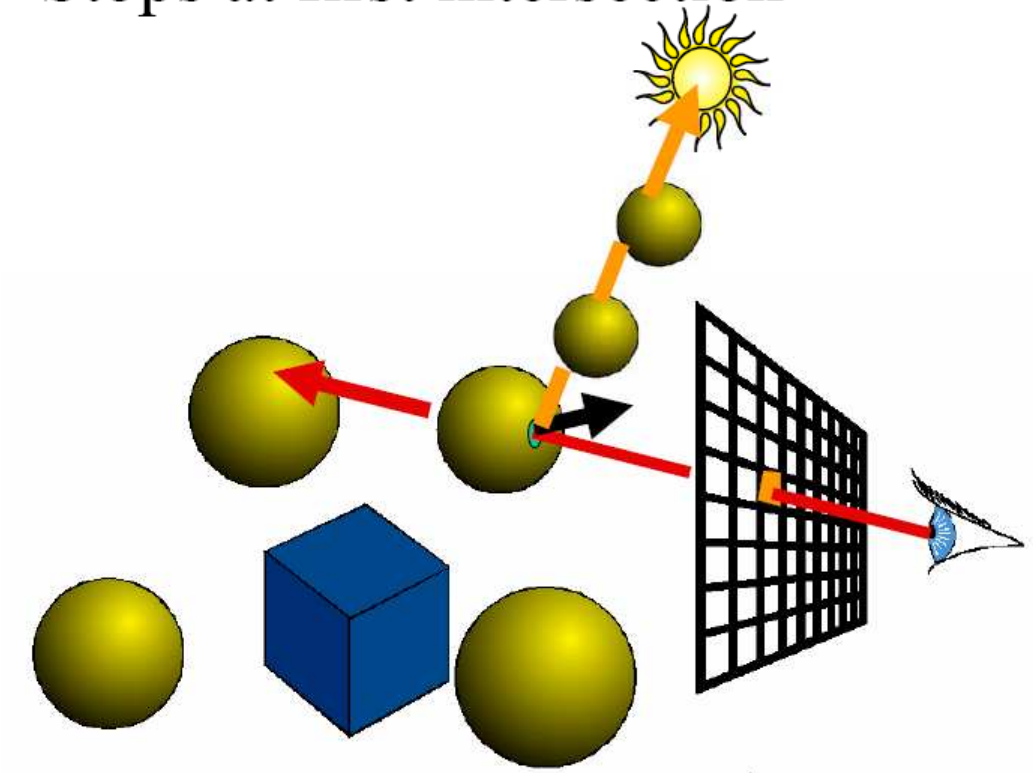
Shadow optimization

- Shadow rays are special
- How can we accelerate our code?



Shadow optimization

- We only want to know whether there is an intersection, not which one is closest
- Special routine `Object3D::intersectShadowRay()`
  - Stops at first intersection

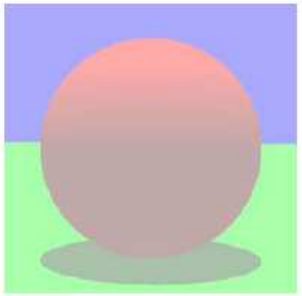


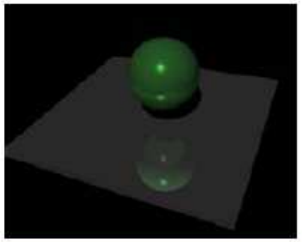
Shadow ray casting history

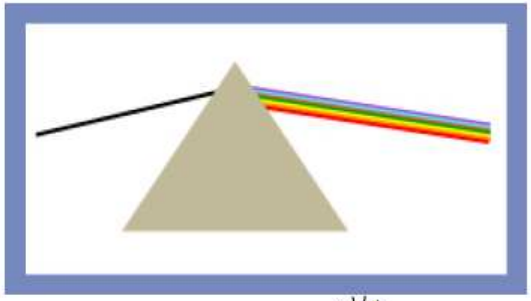
- Due to Appel [1968]
- First shadow method in graphics
- Not really used until the 80s

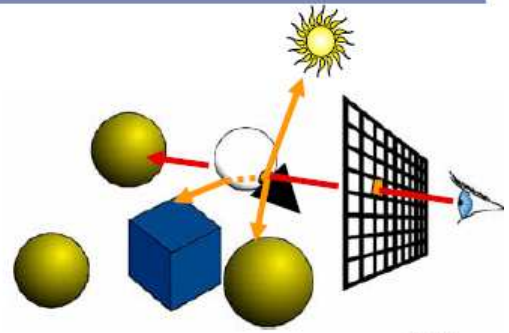
Overview of today

- Shadows
- Reflection
- Refraction
- Recursive Ray Tracing



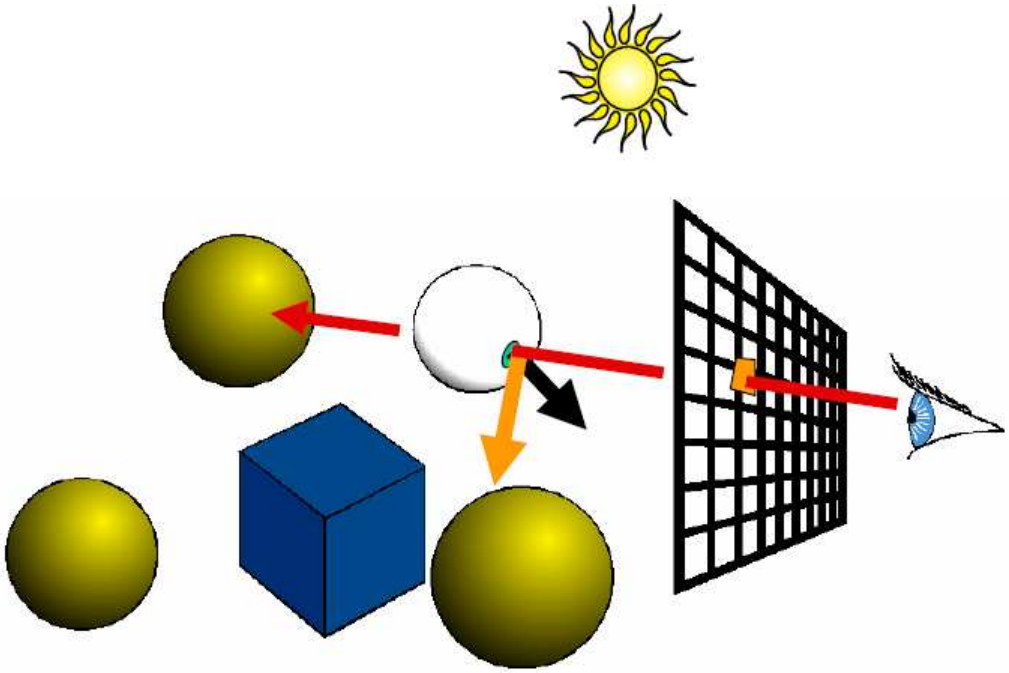


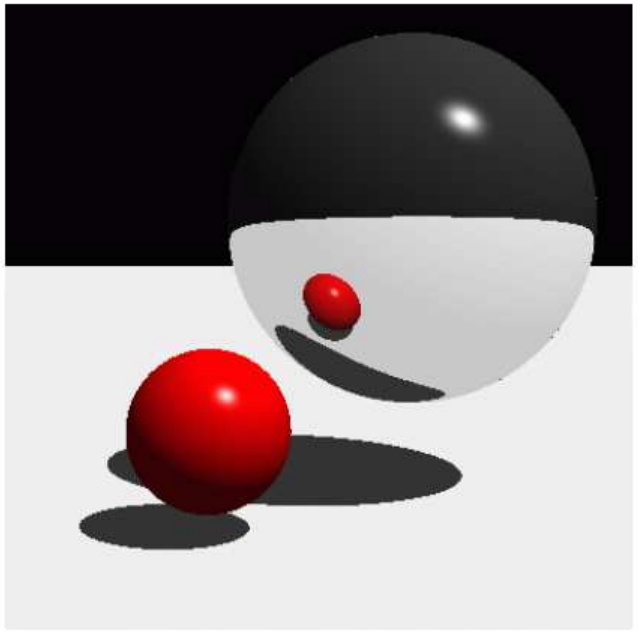




Mirror Reflection

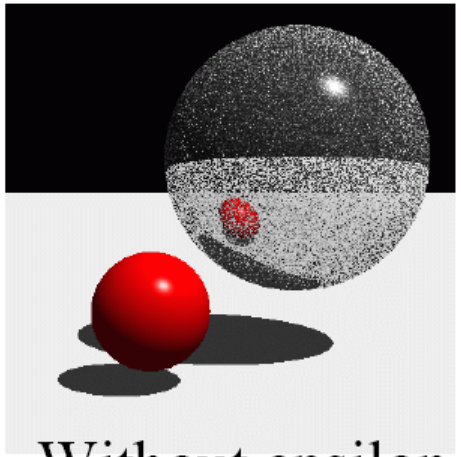
- Compute mirror contribution
- Cast ray
  - In direction symmetric wrt normal
- Multiply by reflection coefficient (color)



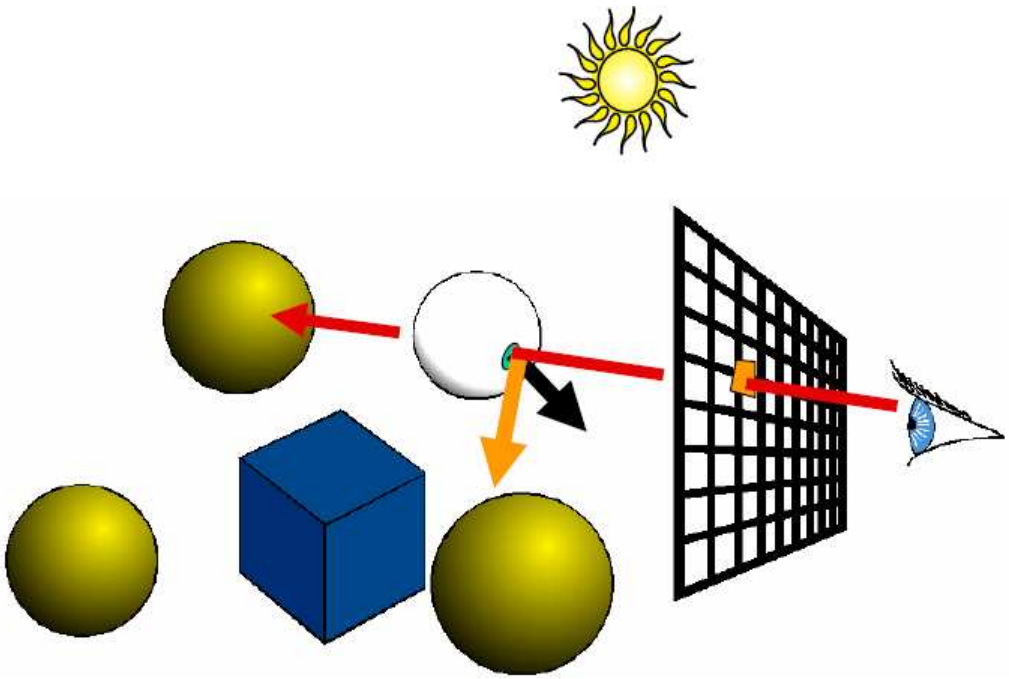


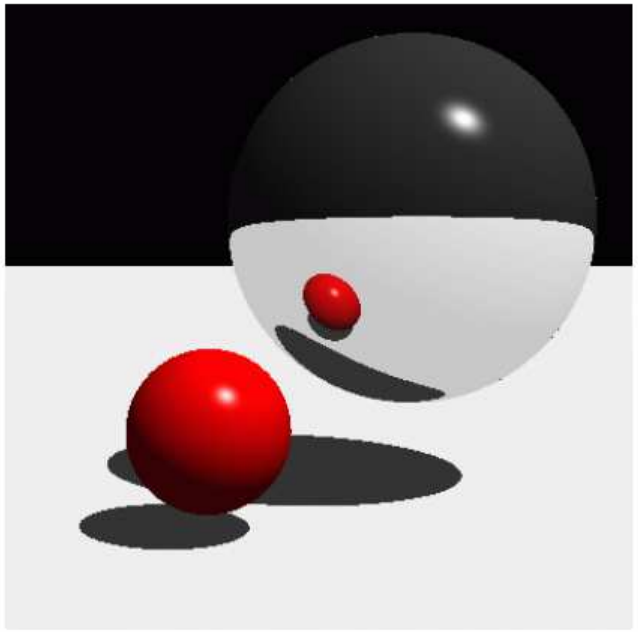
Mirror Reflection

- Cast ray
  - In direction symmetric wrt normal
- Don't forget to add epsilon to the ray



Without epsilon



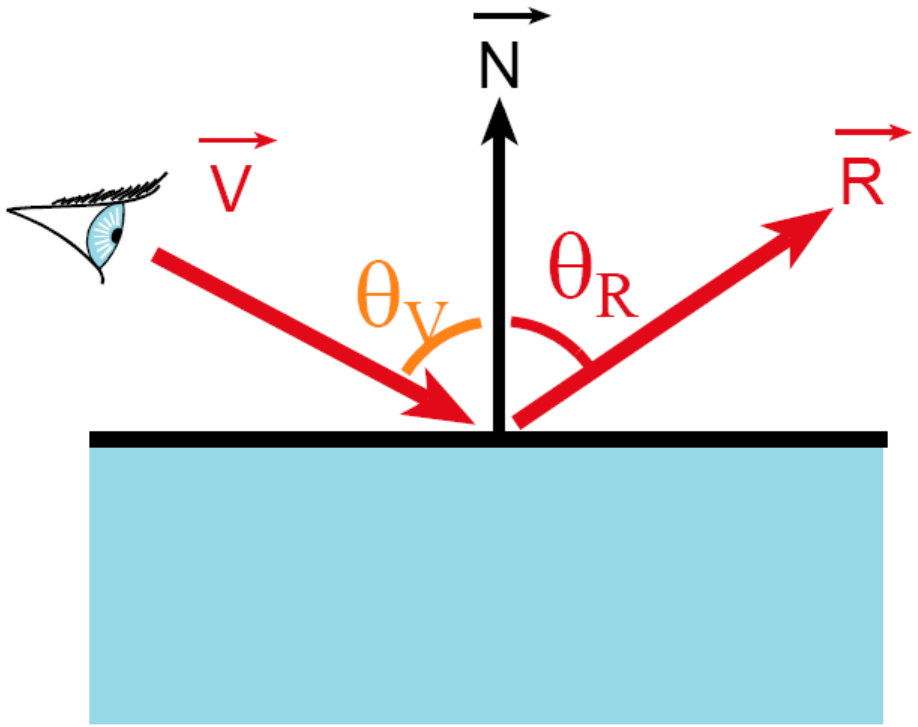


With epsilon

and Durand

Reflection

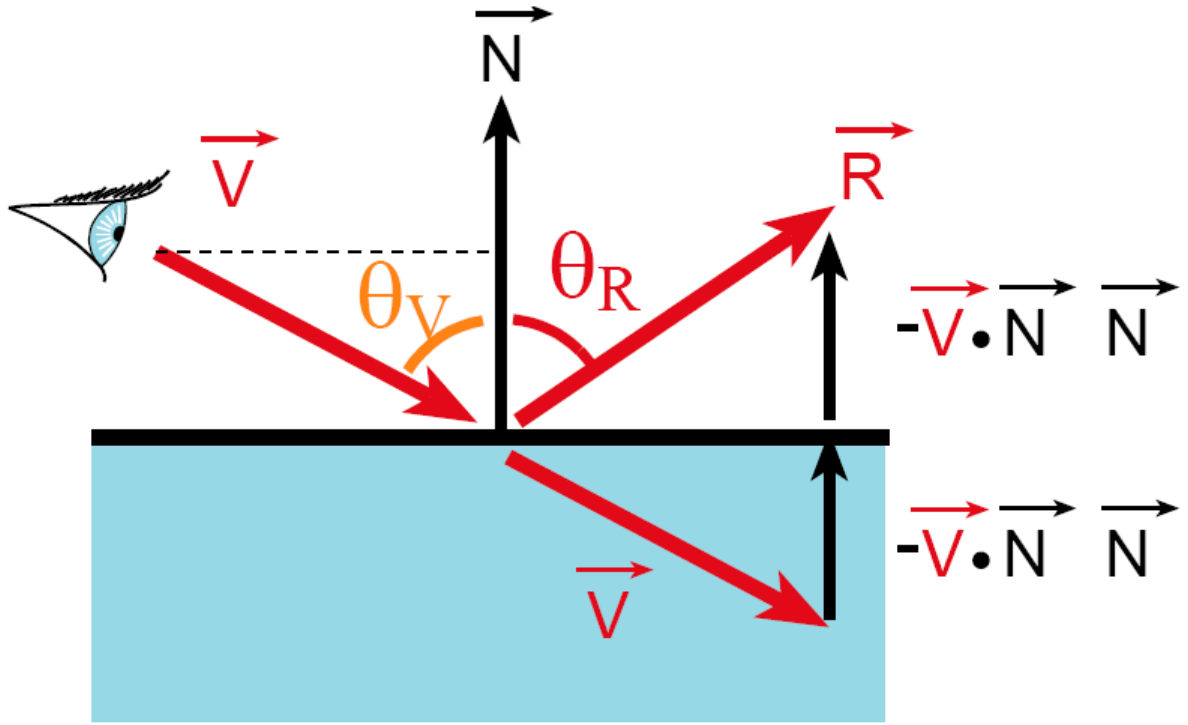
- Reflection angle = view angle



Reflection

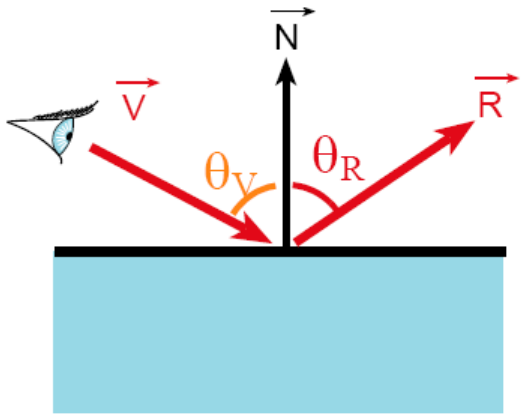
- Reflection angle = view angle

$$\vec{R} = \vec{V} - 2\left(\vec{V} \bullet \vec{N}\right)\vec{N}$$



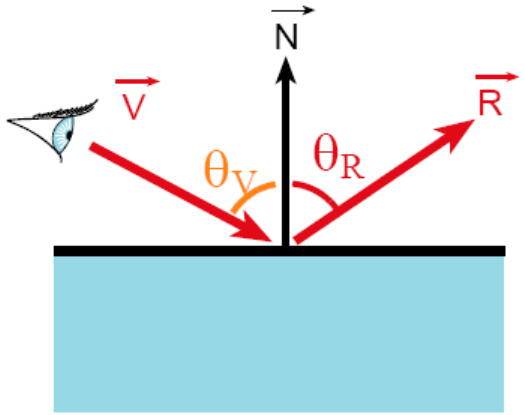
Amount of Reflection

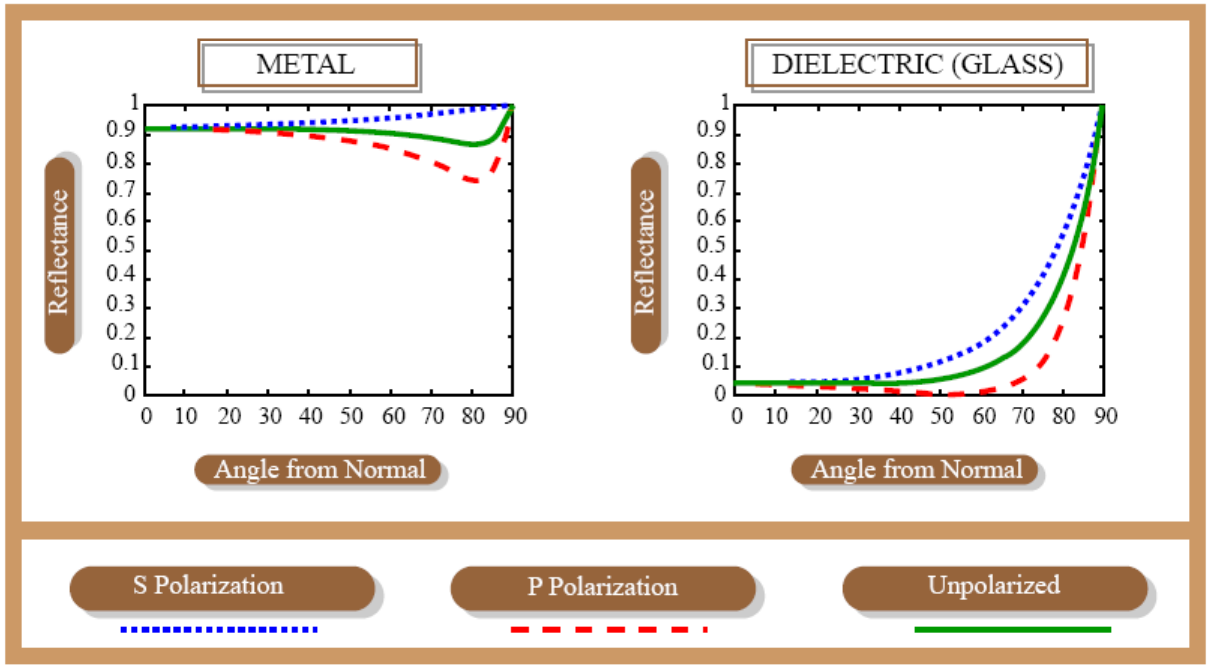
- Traditional (hacky) ray tracing
  - Constant coefficient `reflectionColor`
  - Component per component multiplication



Amount of Reflection

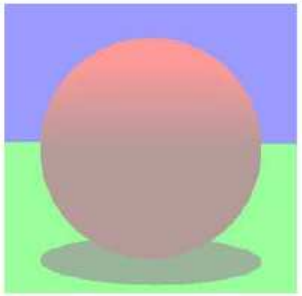
- More realistic:
  - Fresnel reflection term
  - More reflection at grazing angle
  - Schlick's approximation:
    $R(\theta)=R_0+(1-R_0)(1-\cos\theta)^5$

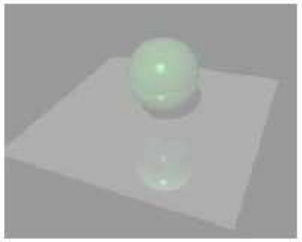


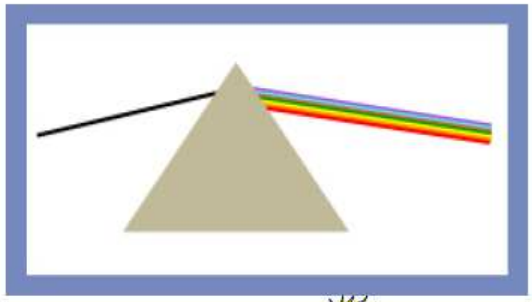


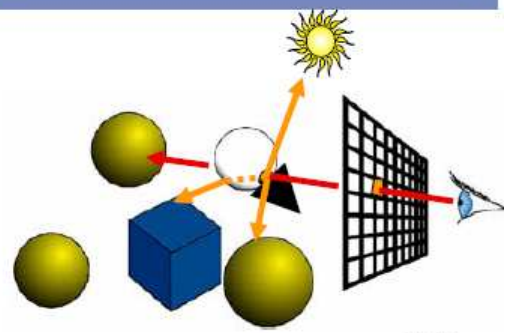
Overview of today

- Shadows
- Reflection
- Refraction
- Recursive Ray Tracing



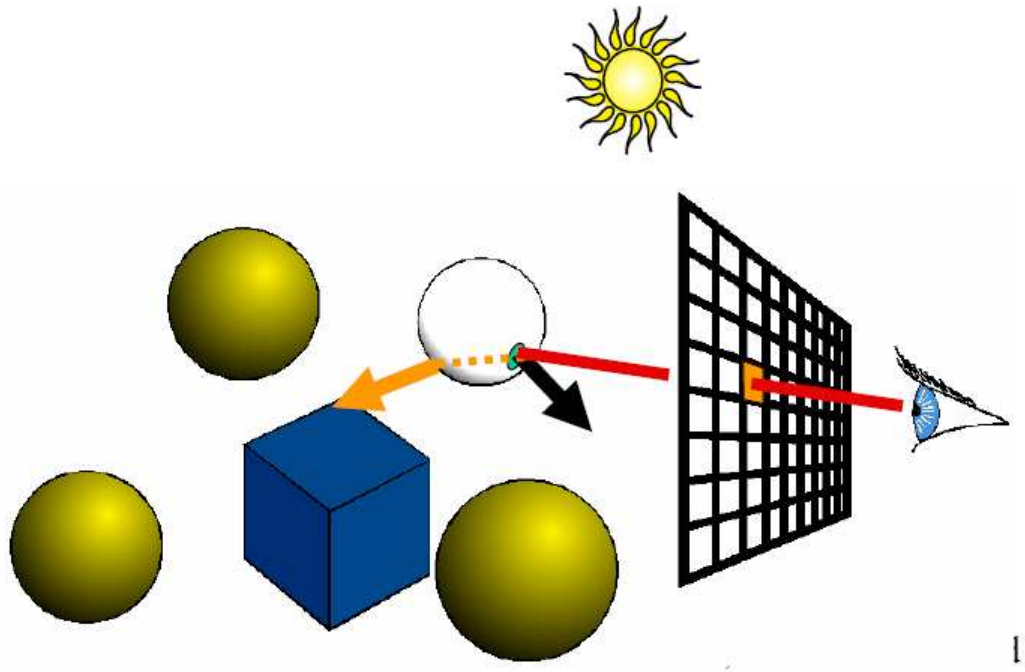






Transparency

- Compute transmitted contribution
- Cast ray
  - In refracted direction
- Multiply by transparency coefficient (color)



Qualitative refraction

- From "Color and Light in Nature" by Lynch and Livingston

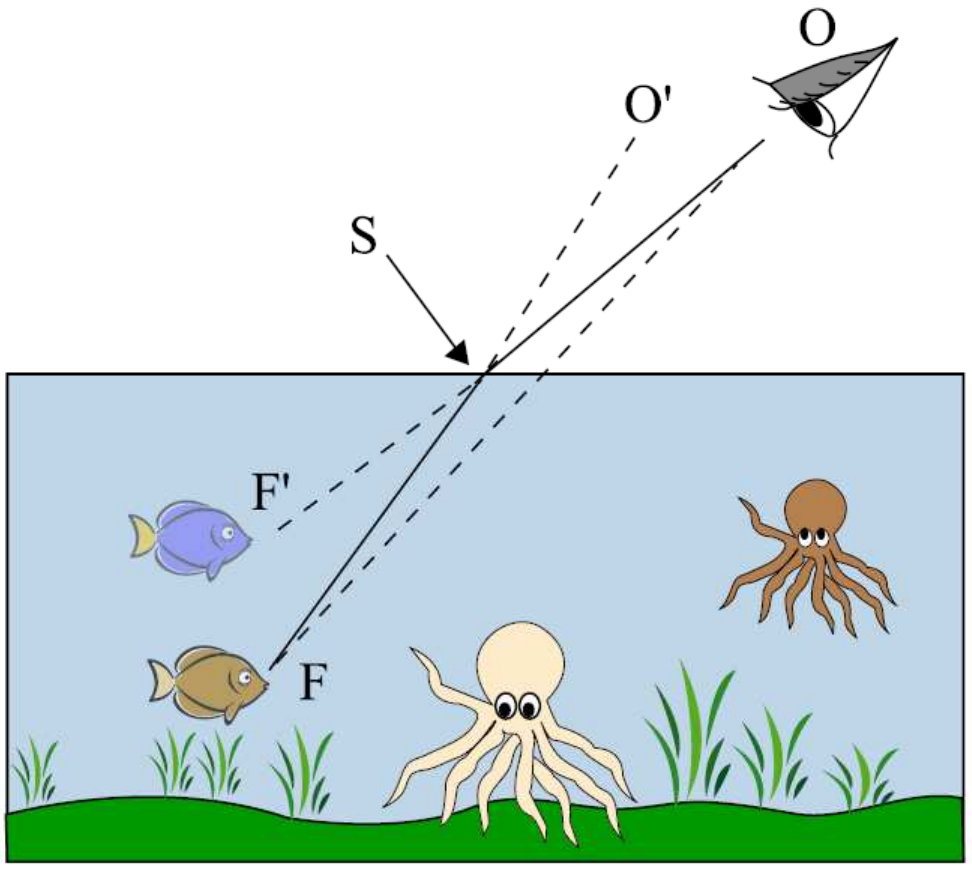
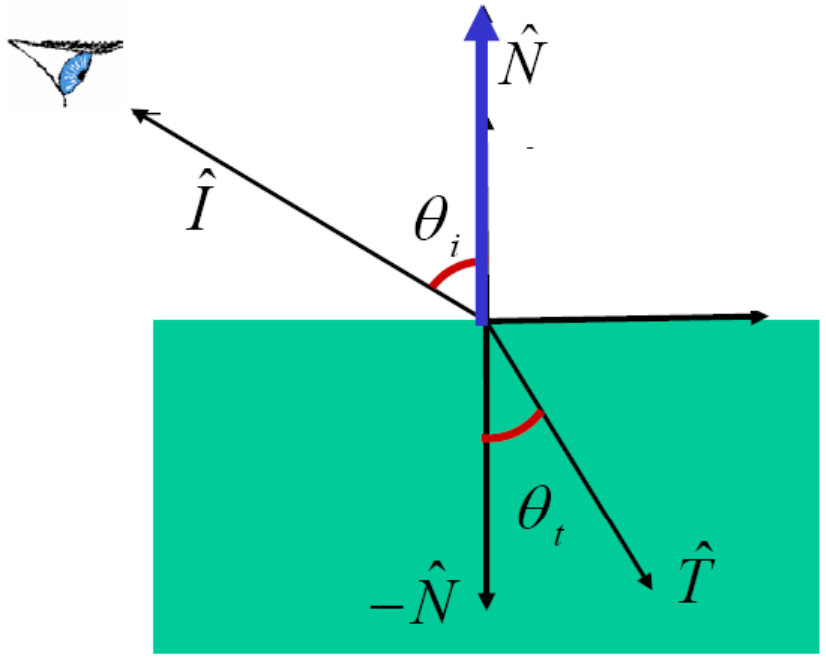


Image adapted from:
Lynch, David K. and William Livingston. *Color and Light in Nature*. Cambridge University Press. June 2001. ISBN: 0-521-77504-3.

Refraction

Snell-Descartes Law

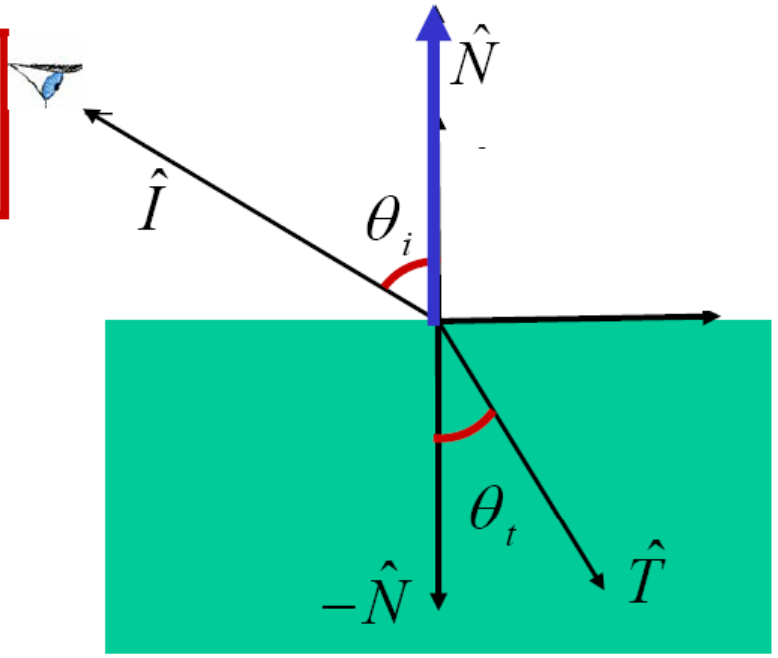


Note that I is the negative of the incoming ray

Refraction

Snell-Descartes Law

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_t}{\eta_i} = \eta_r$$



Note that I is the negative of the incoming ray

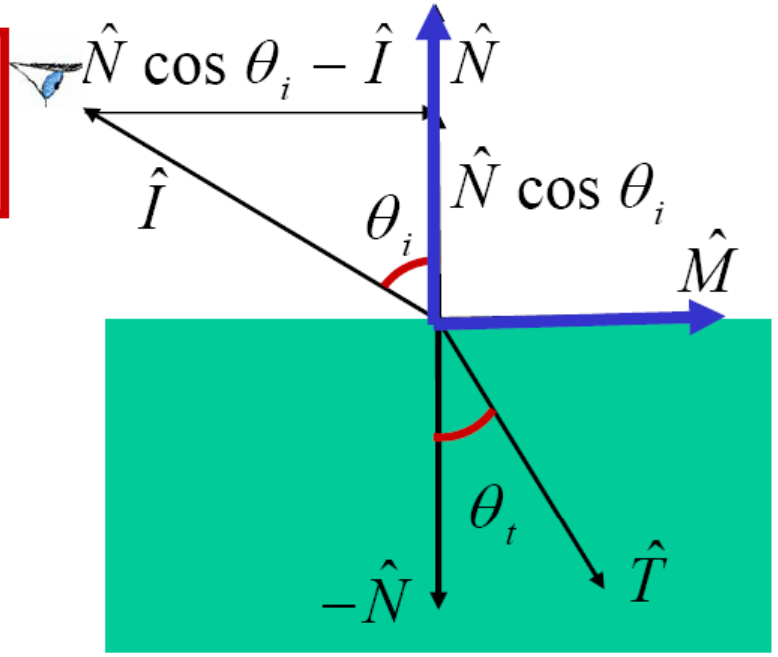
Refraction

Snell-Descartes Law

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_t}{\eta_i} = \eta_r$$

$$\hat{T} = \sin\theta_t \hat{M} - \cos\theta_t \hat{N}$$

$$\hat{M} = \frac{(\hat{N}\cos\theta_i - \hat{I})}{\sin\theta_i}$$



Note that I is the negative of the incoming ray

Refraction

Snell-Descartes Law

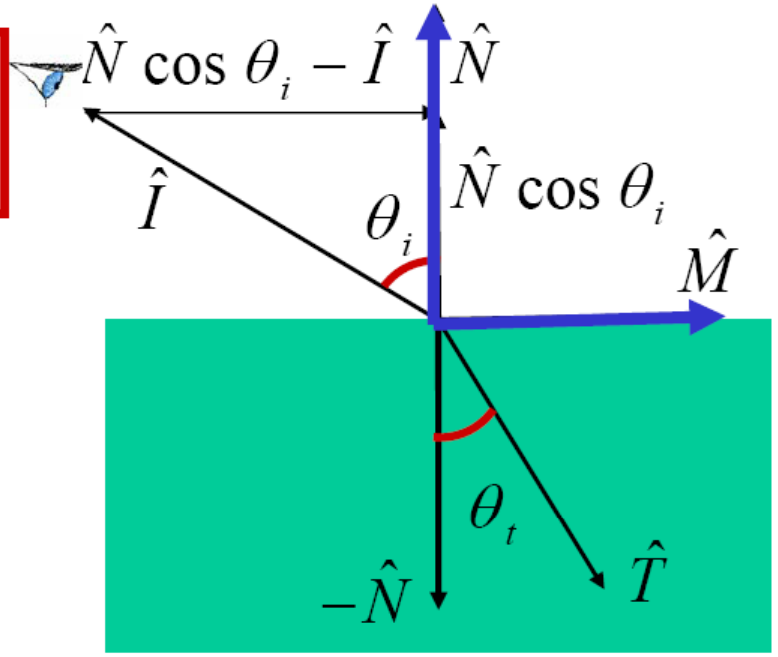
$$\frac{\sin\theta_t}{\sin\theta_i} = \frac{\eta_i}{\eta t} = \eta_r$$

$$\hat{T} = \sin\theta_t \hat{M} - \cos\theta_t \hat{N}$$

$$\hat{M} = \frac{(\hat{N}\cos\theta_i - \hat{I})}{\sin\theta_i}$$

$$\hat{T} = \frac{\sin\theta_t}{\sin\theta_i}(\hat{N}\cos\theta_i - \hat{I}) - \cos\theta_t \hat{N}$$

$$\hat{T} = (\eta_r \cos\theta_i - \cos\theta_t)\hat{N} - \eta_r \hat{I}$$

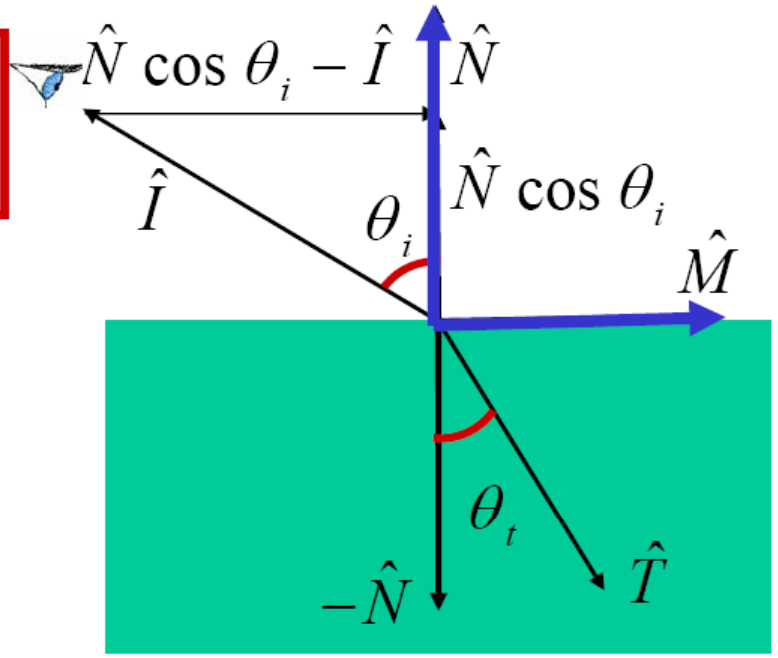


Note that I is the negative of the incoming ray

Refraction

Snell-Descartes Law

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_t}{\eta_i} = \eta_r$$



Note that I is the negative of the incoming ray

$$\hat{T} = \sin\theta_t \hat{M} - \cos\theta_t \hat{N}$$

$$\hat{M} = \frac{(\hat{N}\cos\theta_i - \hat{I})}{\sin\theta_i}$$

$$\hat{T} = \frac{\sin\theta_t}{\sin\theta_i}(\hat{N}\cos\theta_i - \hat{I}) - \cos\theta_t \hat{N}$$

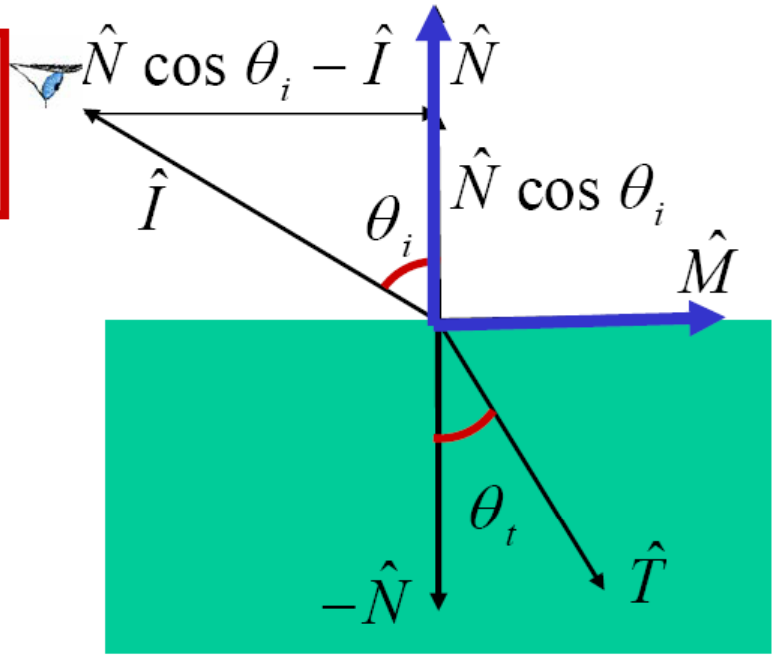
$$\hat{T} = (\eta_r \cos\theta_i - \cos\theta_t)\hat{N} - \eta_r \hat{I}$$

$$\cos\theta_i = \hat{N}\cdot\hat{I}$$

$$\cos\theta_t = \sqrt{1-\sin^2\theta_t} = \sqrt{1-\eta_r^2\sin^2\theta_i} = \sqrt{1-\eta_r^2(1-(\hat{N}\cdot\hat{I})^2)}$$

Refraction

Snell-Descartes Law $\frac{\sin\theta_t}{\sin\theta_i} = \frac{\eta_i}{\eta_t} = \eta_r$



Note that I is the negative of the incoming ray

$$\hat{T} = \sin\theta_t \hat{M} - \cos\theta_t \hat{N}$$

$$\hat{M} = \frac{(\hat{N}\cos\theta_i - \hat{I})}{\sin\theta_i}$$

$$\hat{T} = \frac{\sin\theta_t}{\sin\theta_i}(\hat{N}\cos\theta_i - \hat{I}) - \cos\theta_t \hat{N}$$

$$\hat{T} = (\eta_r \cos\theta_i - \cos\theta_t)\hat{N} - \eta_r \hat{I}$$

$$\cos\theta_i = \hat{N}\cdot\hat{I}$$

$$\cos\theta_t = \sqrt{1-\sin^2\theta_t} = \sqrt{1-\eta_r^2\sin^2\theta_i} = \sqrt{1-\eta_r^2(1-(\hat{N}\cdot\hat{I})^2)}$$

$$\hat{T} = \left(\eta_r(\hat{N}\cdot\hat{I}) - \sqrt{1-\eta_r^2(1-(\hat{N}\cdot\hat{I})^2)}\right)\hat{N} - \eta_r\hat{I}$$

Don't forget to normalize

Total internal reflection when the square root is imaginary

Total internal reflection

- From "Color and Light in Nature" by Lynch and Livingstone

THE OPTICAL MANHOLE. LIGHT FROM THE HORIZON (ANGLE OF INCIDENCE = 90^{o}) IS REFRACTED DOWNWARD AT AN ANGLE OF 48.6^{o}. THIS COMPRESSES THE SKY INTO A CIRCLE WITH A DIAMETER OF 97.2^{o} INSTEAD OF ITS USUAL 180^{o}.

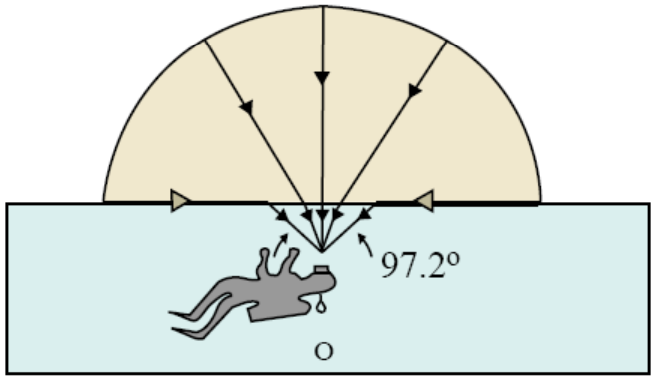
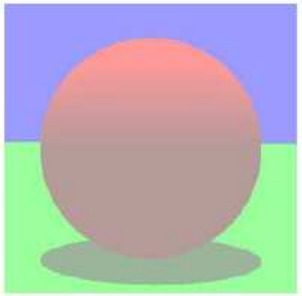
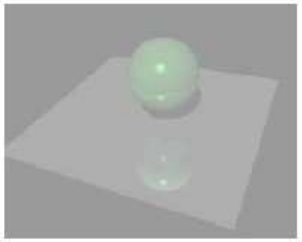


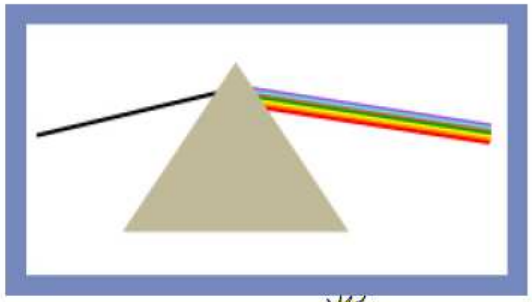
Image adapted from:
Lynch, David K. and William Livingston. *Color and Light in Nature*. Cambridge University Press. June 2001. ISBN: 0-521-77504-3.

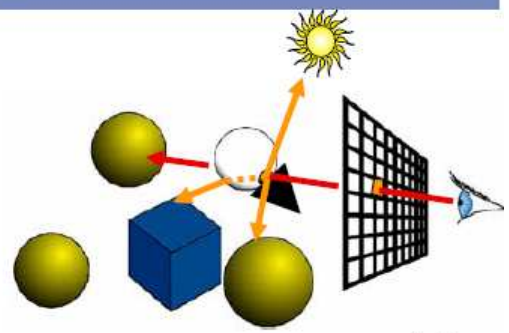
Overview of today

- Shadows
- Reflection
- Refraction
- Recursive Ray Tracing

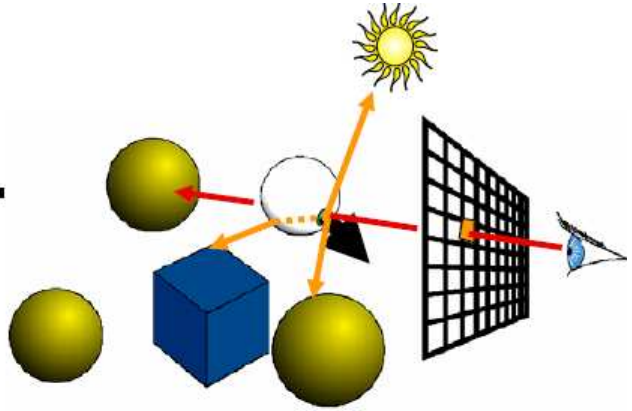






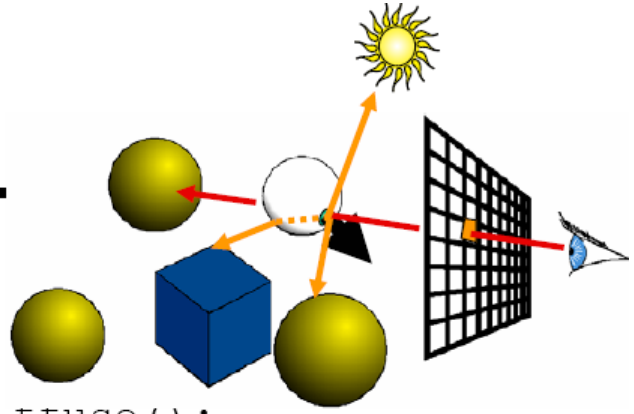


Recap: Ray Tracing



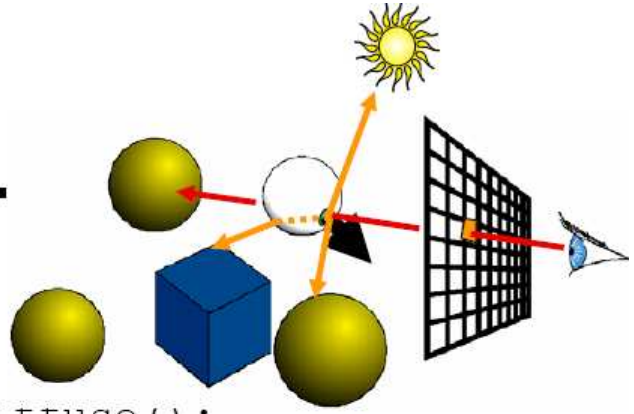
```
traceRay
   Intersect all objects
   Ambient shading
   For every light
      Shadow ray
      shading
   If mirror
      Trace reflected ray
   If transparent
      Trace transmitted ray
```

Recap: Ray Tracing



```
Color traceRay(ray)
   For every object ob
        ob->intersect(ray, hit, tmin);
    Color col=ambient*hit->getMaterial()->getDirruse();
    For every light L
        If ( not castShadowRay( hit->getPoint(), L->getDir())
            col=col+hit->getMaterial()->shade
              (ray, hit, L->getDir(), L->getColor());
    If (hit->getMaterial()->isMirror())
        Ray rayMirror (hit->getPoint(),
          getMirrorDir(ray->getDirection(), hit->getNormal());
        Col=col+hit->getMaterial->getMirrorColor()
          *traceRay(rayMirror, hit2);
    If (hit->getMaterial()->isTransparent()
        Ray rayTransmitted(hit->getPoint(),
          getRefracDir(ray, hit->getNormal(), curentRefractionIndex,
          hit->Material->getRefractionIndex());
        Col=col+hit->getMaterial->getTransmittedColor()
          *traceRay(rayTransmitted, hit3);
    Return col;
```

Does it end?



```
Color traceRay(ray)
   For every object ob
        ob->intersect(ray, hit, tmin);
    Color col=ambient*hit->getMaterial()->getDirruse();
    For every light L
        If ( not castShadowRay( hit->getPoint(), L->getDir())
            col=col+hit->getMaterial()->shade
              (ray, hit, L->getDir(), L->getColor());
    If (hit->getMaterial()->isMirror())
        Ray rayMirror (hit->getPoint(),
          getMirrorDir(ray->getDirection(), hit->getNormal());
        Col=col+hit->getMaterial->getMirrorColor()
          *traceRay(rayMirror, hit2);
    If (hit->getMaterial()->isTransparent()
        Ray rayTransmitted(hit->getPoint(),
          getRefracDir(ray, hit->getNormal(), curentRefractionIndex,
          hit->Material->getRefractionIndex());
        Col=col+hit->getMaterial->getTransmittedColor()
          *traceRay(rayTransmitted, hit3);
    Return col;
```

Avoiding infinite recursion

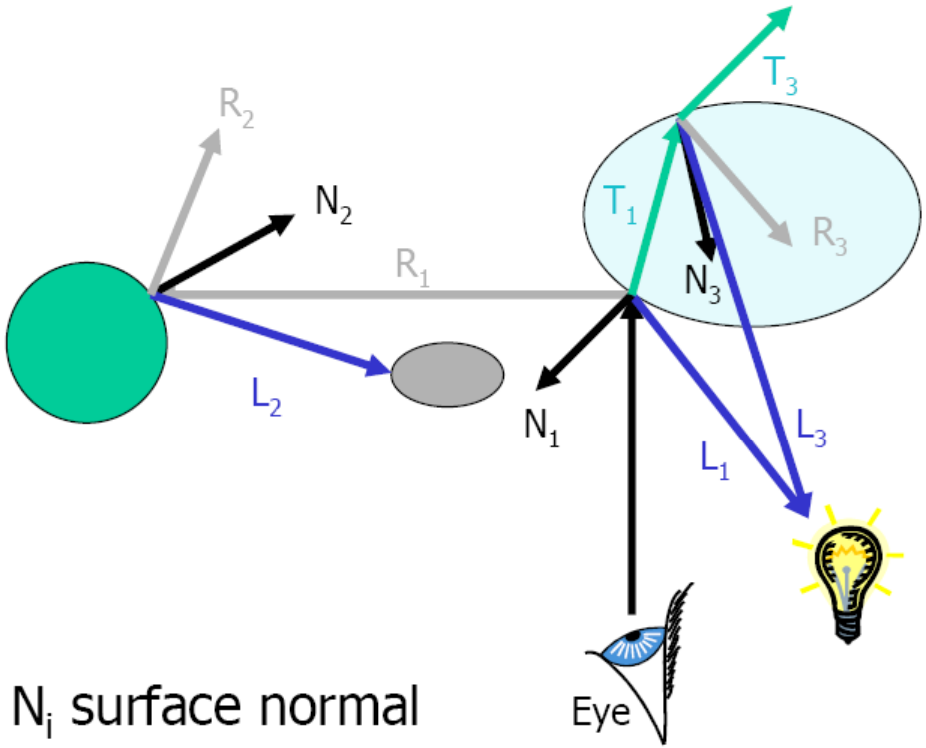
Stopping criteria:

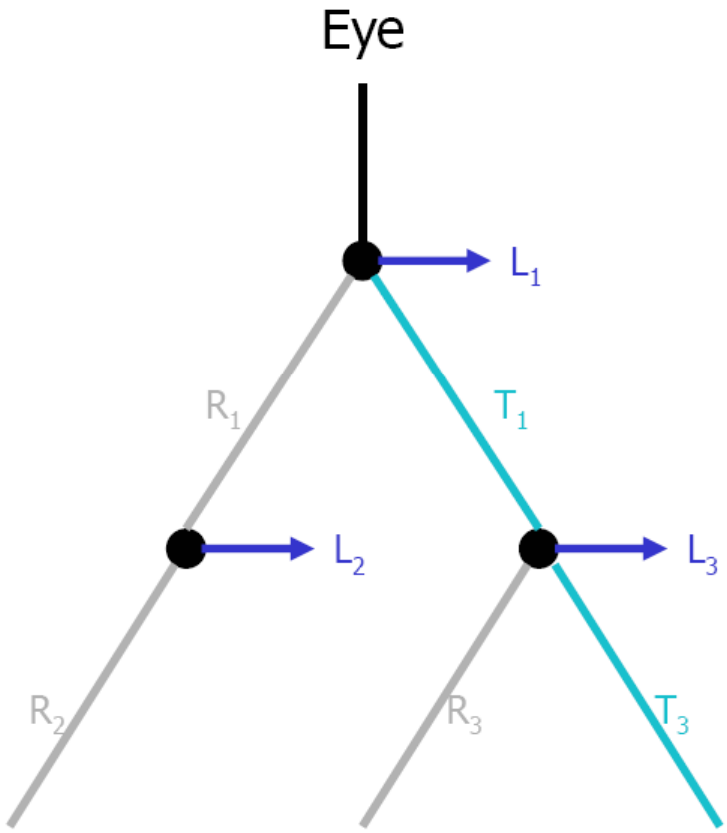
- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if transparency/transmitted attenuation becomes too small

Usually do both

```
Color traceRay(ray)
     For every object ob
            ob->intersect(ray, hit, tmin);
       Color col=ambient*hit->getMaterial()->getDiffuse();
       For every light L
            If ( not castShadowRay( hit->getPoint(), L->getDir())
                 col=col+hit->getMaterial()->shade
                     (ray, hit, L->getDir(), L->getColor());
       If (hit->getMaterial()->isMirror())
            Ray rayMirror (hit->getPoint(),
                getMirrorDir(ray->getDirection(), hit->getNormal());
            Col=col+hit->getMaterial->getMirrorColor()
                *traceRay(rayMirror);
       If (hit->getMaterial()->isTransparent()
            Ray rayTransmitted(hit->getPoint(),
                getRefracDir(ray, hit->getNormal(),
                curentRefractionIndex, hit->Material-
                >getRefractionIndex());
            Col=col+hit->getMaterial->getTransmittedColor()
                *traceRay(rayTransmitted);
       Return col;
```

The Ray Tree





N_i surface normal

R_i reflected ray

L_i shadow ray

T_i transmitted (refracted) ray

Ray Tracing History

- Ray Casting: Appel, 1968
- CSG and quadrics: Goldstein & Nagel 1971
- Recursive ray tracing: Whitted, 1980

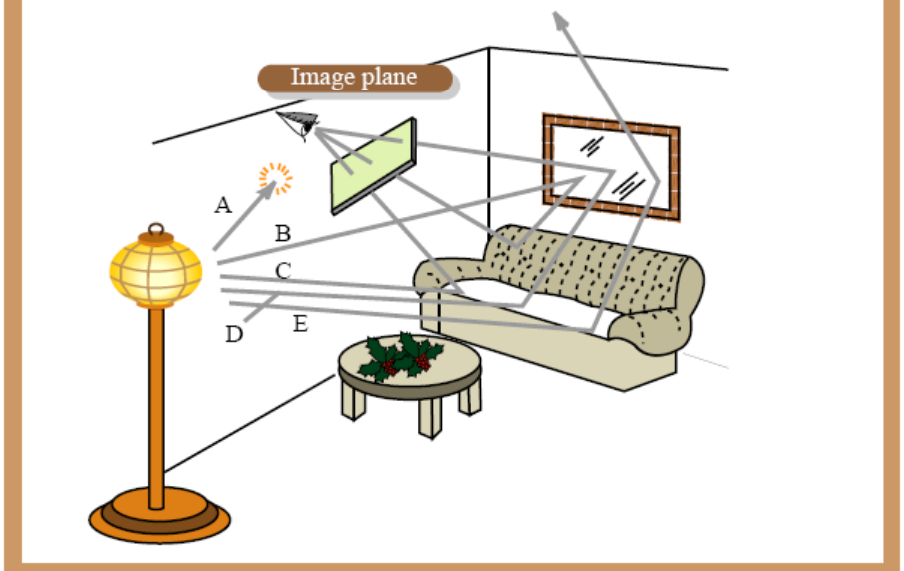
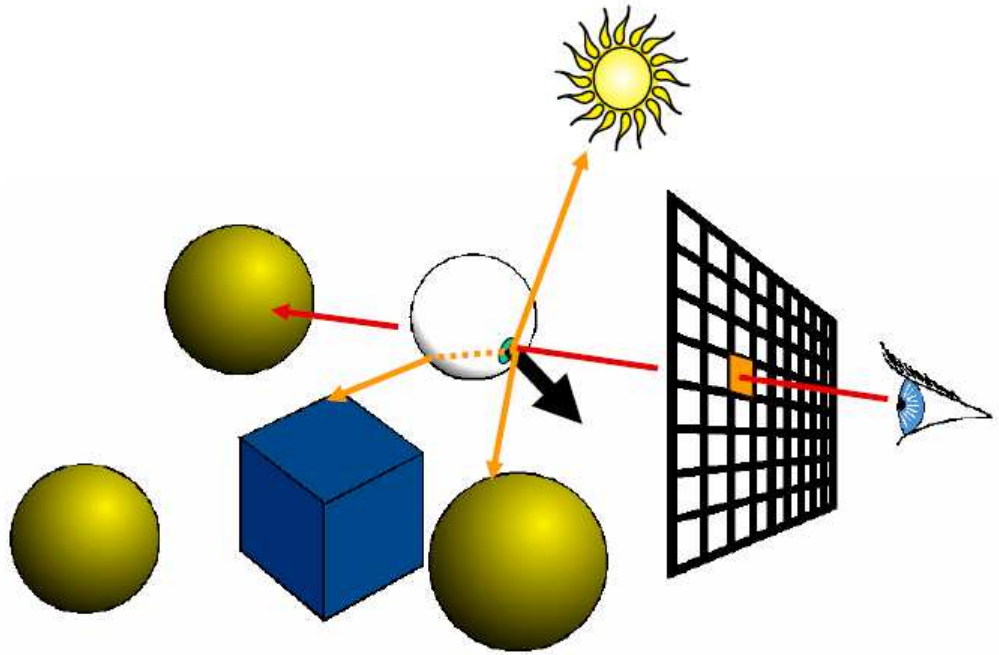


Image adapted from:
Appel, A. "Some Techniques for Shading Machine Renderings of Solids." *Proceedings of the Spring Joint Computer Conference*. 1968, pp. 37-45.

Image removed due to copyright considerations.

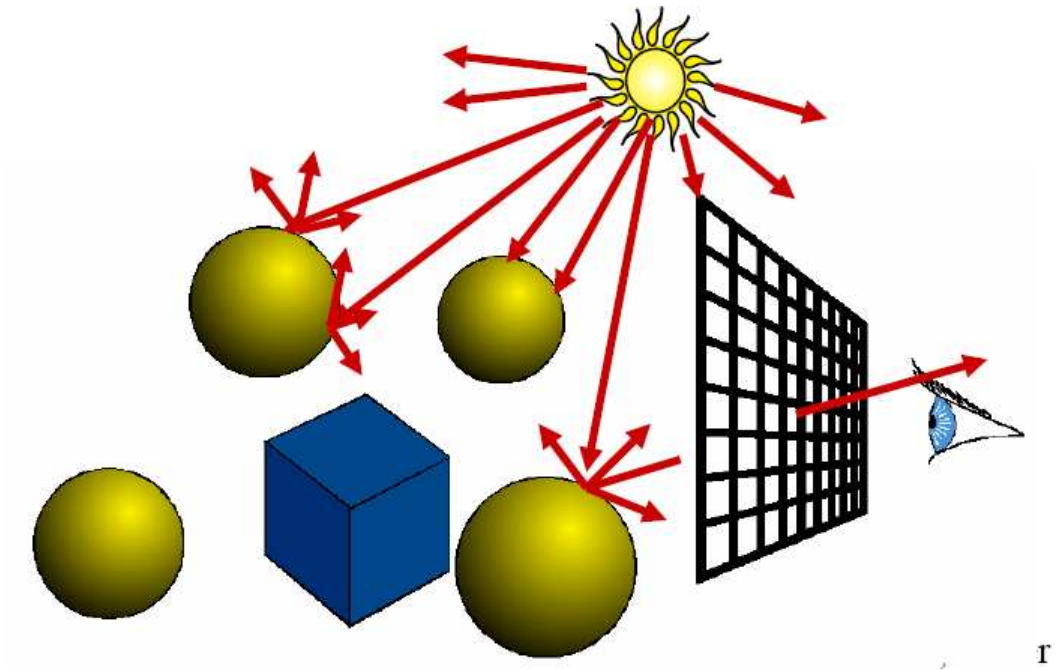
Does Ray Tracing simulate physics?

- Photons go from the light to the eye, not the other way
- What we do is backward ray tracing



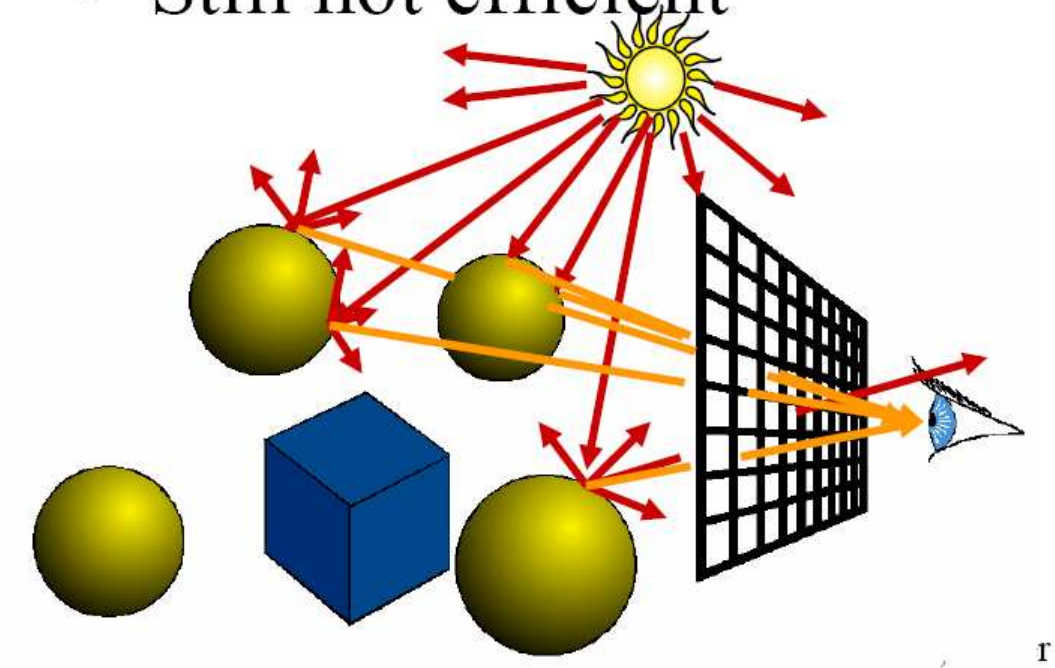
Forward ray tracing

- Start from the light source
- But low probability to reach the eye
  - What can we do about it?



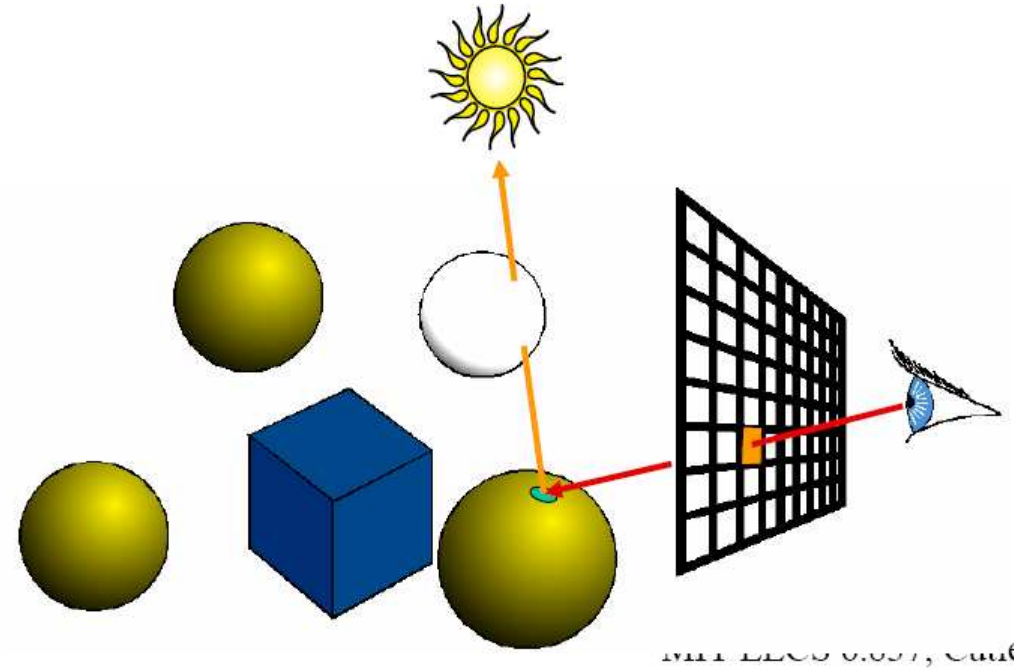
Forward ray tracing

- Start from the light source
- But low probability to reach the eye
  - What can we do about it?
  - Always send a ray to the eye
- Still not efficient



Does Ray Tracing simulate physics?

- Ray Tracing is full of dirty tricks
- e.g. shadows of transparent objects
  - Dirtiest: opaque
  - Still dirty: multiply by transparency color
    - But then no refraction



Ray Casting II

Courtesy of James Arvo and David Kirk. Used with permission.

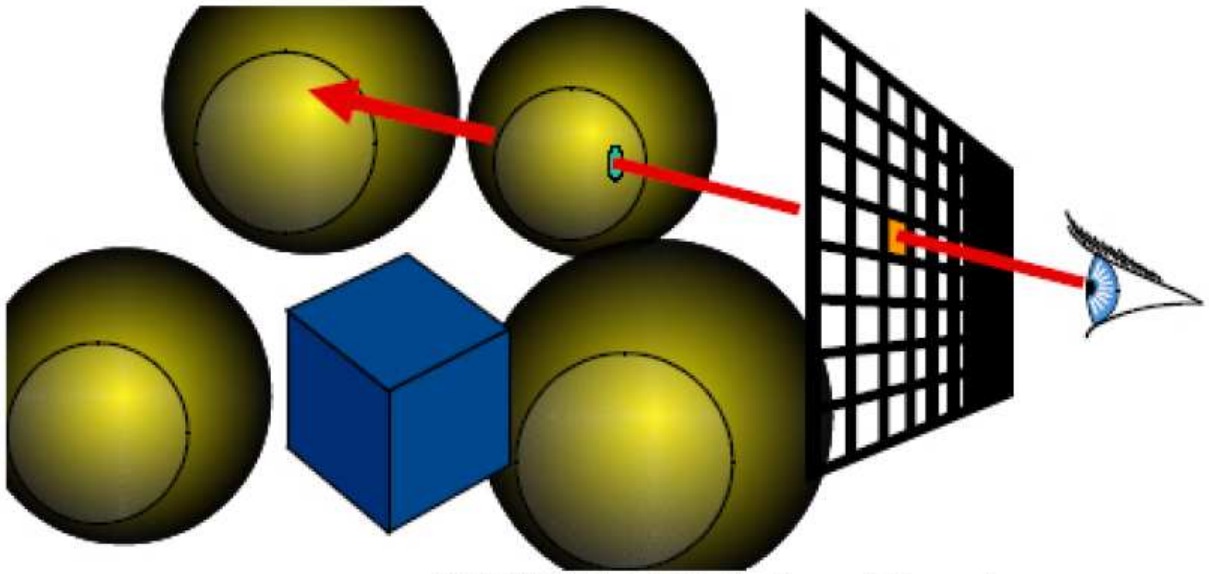
MIT EECS 6.837

Frédo Durand and Barb Cutler

Some slides courtesy of Leonard McMillan

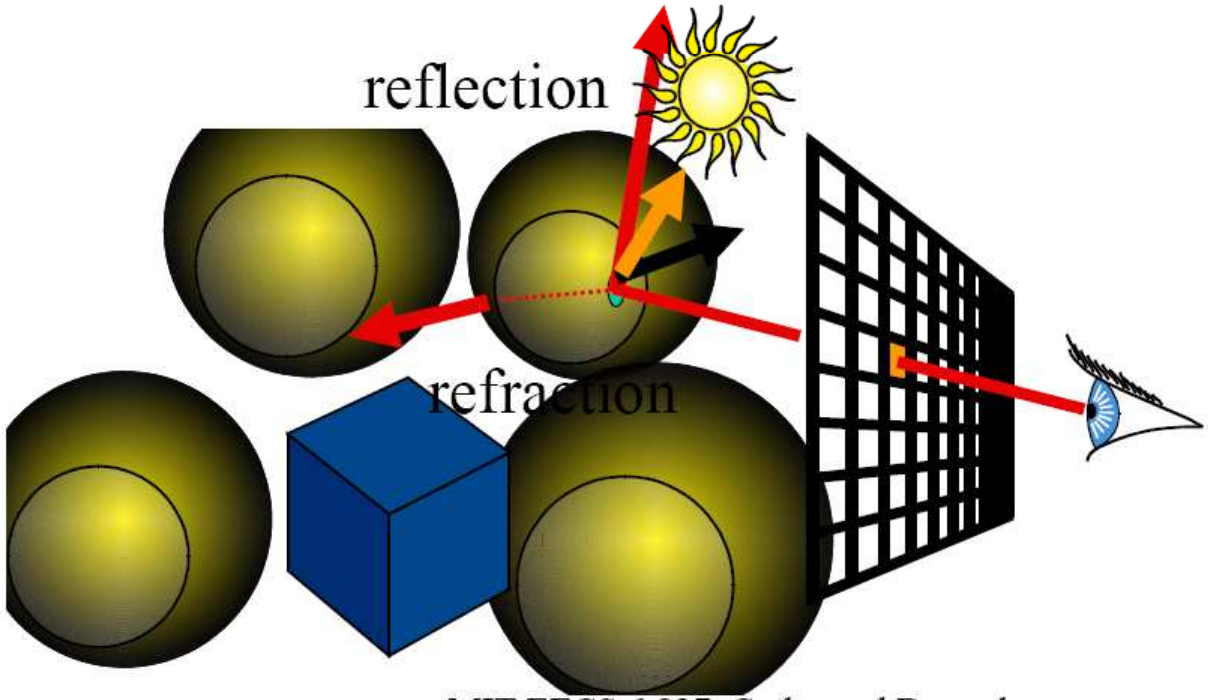
Ray Casting

```
For every pixel
   Construct a ray from the eye
   For every object in the scene
       Find intersection with the ray
       Keep if closest
```



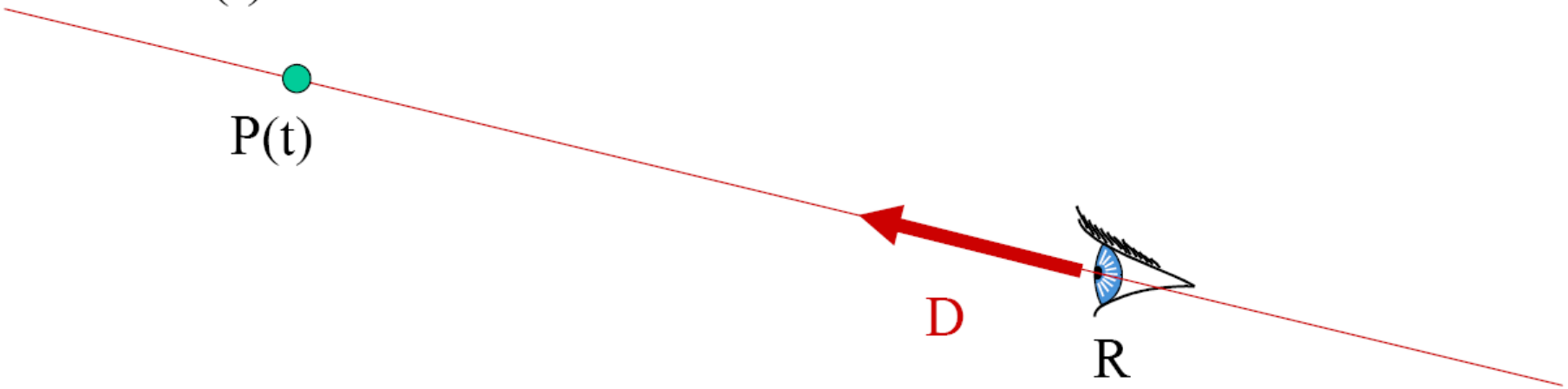
Ray Tracing

- Secondary rays (shadows, reflection, refraction)
- In a couple of weeks



Ray representation

- Two vectors:
  - Origin
  - Direction (normalized is better)
- Parametric line
  - $P(t) = R + t * D$

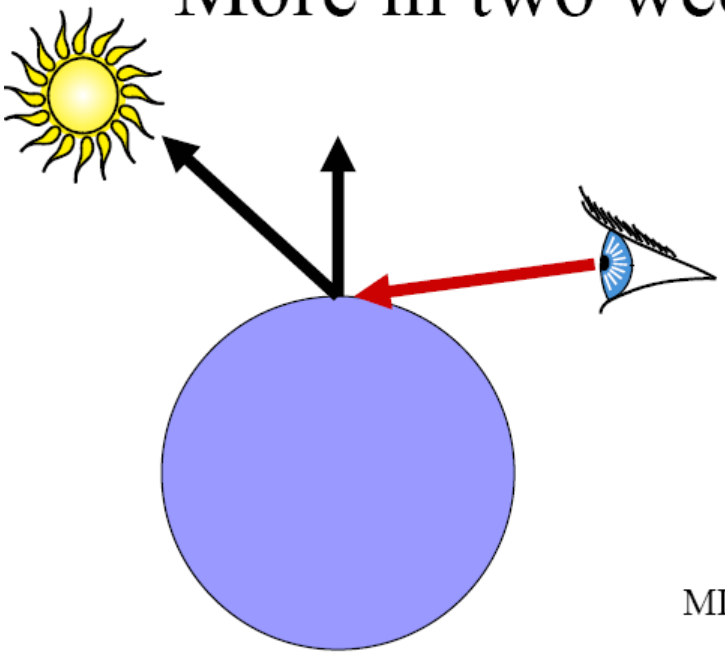


Explicit vs. implicit

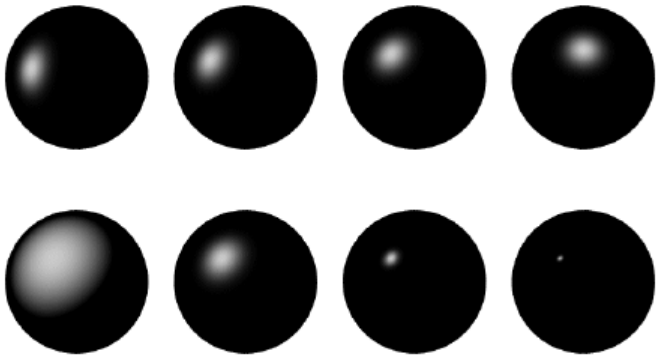
- Implicit
  - Solution of an equation
  - Does not tell us how to generate a point on the plane
  - Tells us how to check that a point is on the plane
- Explicit
  - Parametric
  - How to generate points
  - Harder to verify that a point is on the ray

A note on shading

- Normal direction, direction to light
- Diffuse component: dot product
- Specular component for shiny materials
  - Depends on viewpoint
- More in two weeks



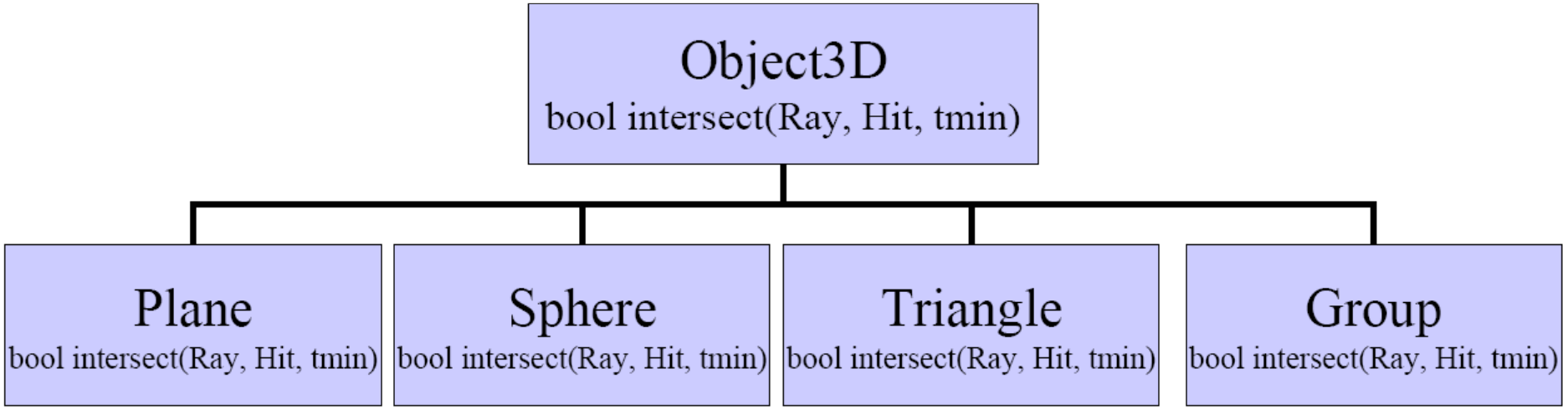
Diffuse sphere



shiny spheres

Object-oriented design

- We want to be able to add primitives easily
  - Inheritance and virtual methods
- Even the scene is derived from Object3D!



Ray

```
class Ray {

public:

  // CONSTRUCTOR & DESTRUCTOR
  Ray () {}
  Ray (const Vec3f &dir, const Vec3f &orig) {
    direction = dir;
    origin = orig; }
  Ray (const Ray& r) {*this=r;}

  // ACCESSORS
  const Vec3f& getOrigin() const { return origin; }
  const Vec3f& getDirection() const { return direction; }
  Vec3f pointAtParameter(float t) const {
    return origin+direction*t; }

private:

  // REPRESENTATION
  Vec3f direction;
  Vec3f origin;

};
```

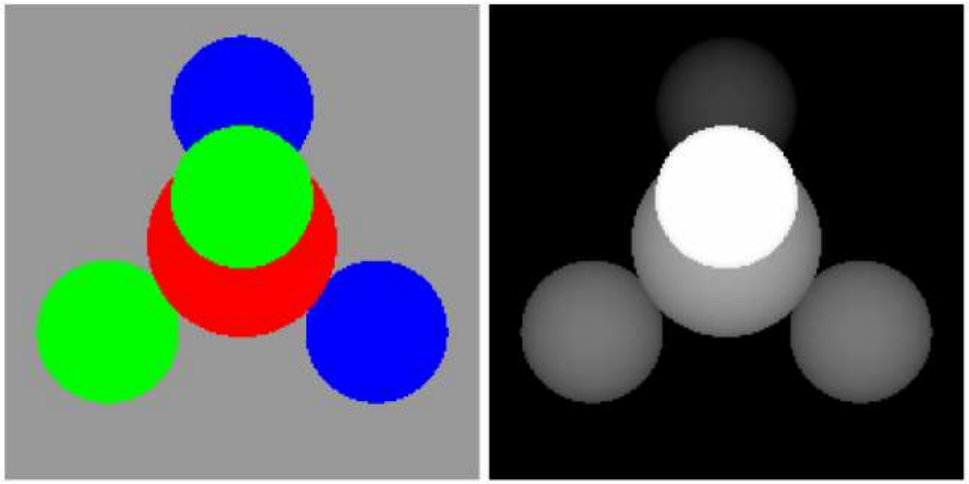
Hit

- Store intersection point & various information

```
class Hit {
public:
    // CONSTRUCTOR & DESTRUCTOR
    Hit(float _t, Vec3f c) { t = _t; color = c; }
    ~Hit() {}
    // ACCESSORS
    float getT() const { return t; }
    Vec3f getColor() const { return color; }
    // MODIFIER
    void set(float _t, Vec3f c) { t = _t; color = c; }
private:
    // REPRESENTATION
    float t;
    Vec3f color;
    //Material *material;
    //Vec3f normal;
};
```

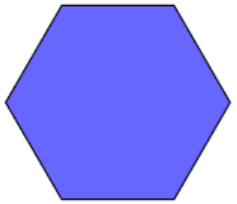
Tasks

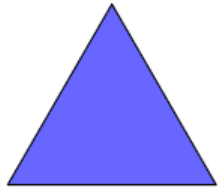
- Abstract Object3D
- Sphere and intersection
- Group class
- Abstract camera and derive Orthographic
- Main function



Overview of today

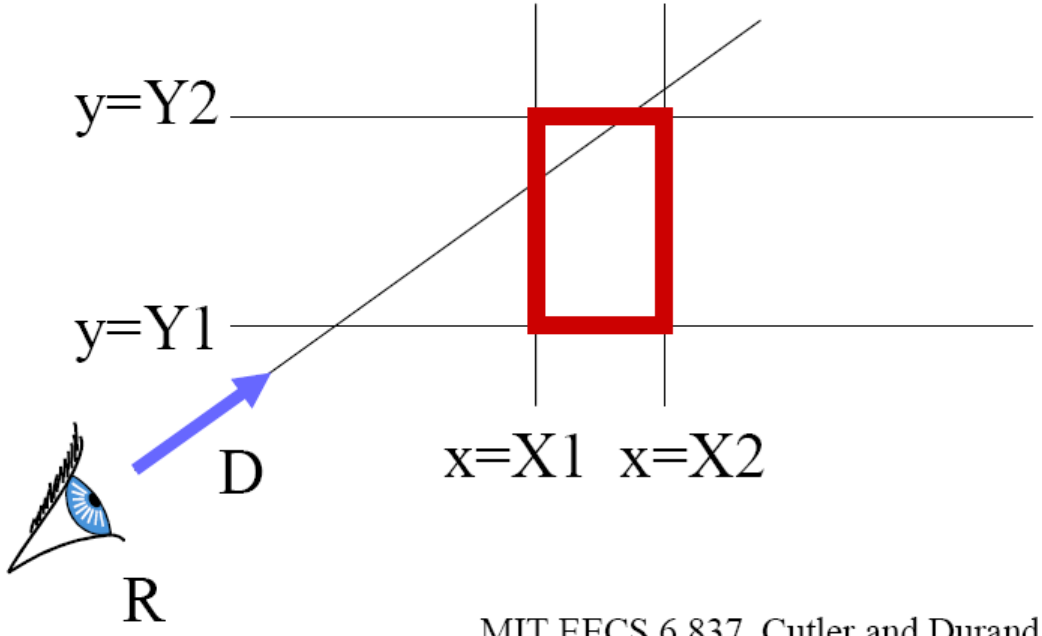
- Ray-box intersection
- Ray-polygon intersection
- Ray-triangle intersection





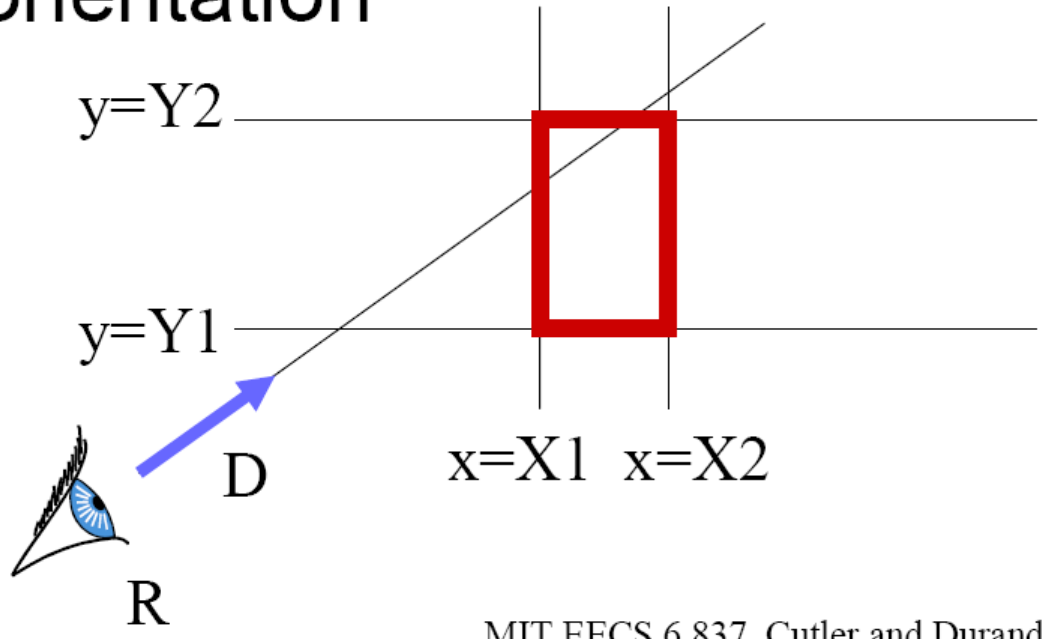
Ray-Parallelepiped Intersection

- Axis-aligned
- From (X1, Y1, Z1) to (X2, Y2, Z2)
- Ray P(t)=R+Dt



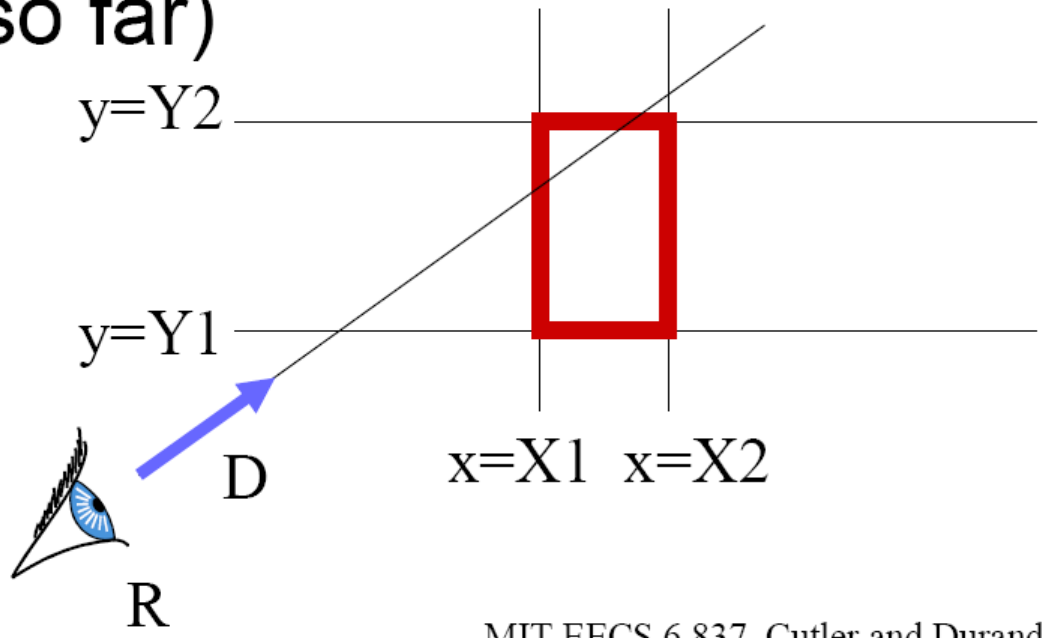
Naïve ray-box Intersection

- Use 6 plane equations
- Compute all 6 intersection
- Check that points are inside box Ax+by+Cz+D<0 with proper normal orientation



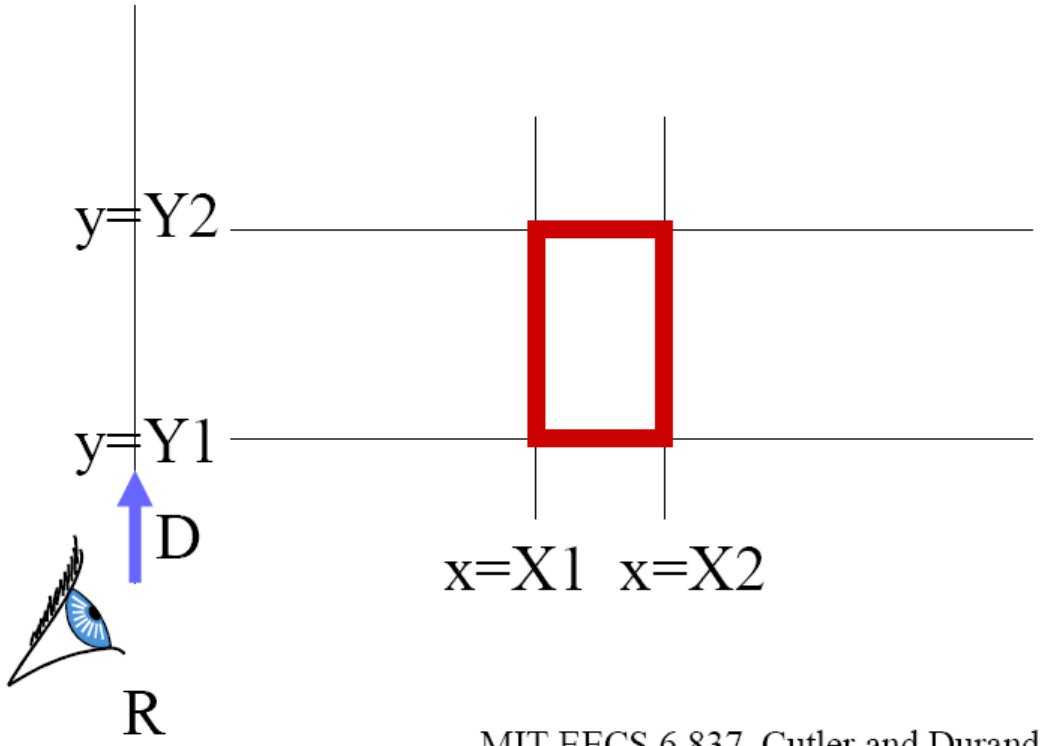
Factoring out computation

- Pairs of planes have the same normal
- Normals have only one non-0 component
- Do computations one dimension at a time
- Maintain tnear and tfar (closest and farthest so far)



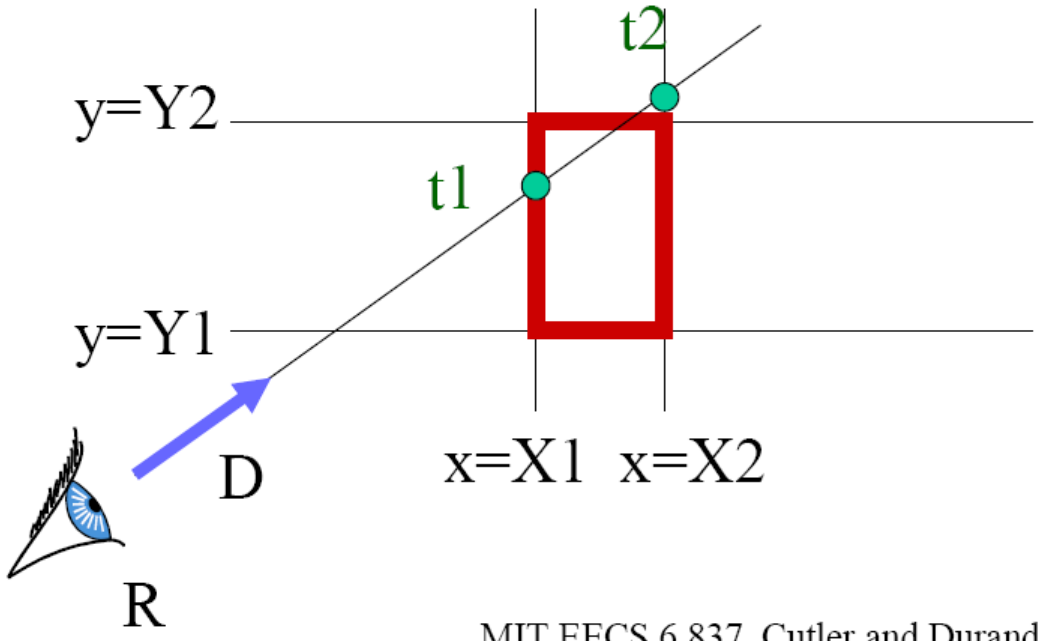
Test if parallel

- If Dx=0, then ray is parallel
  - If Rx<X1 or Rx>x2 return false



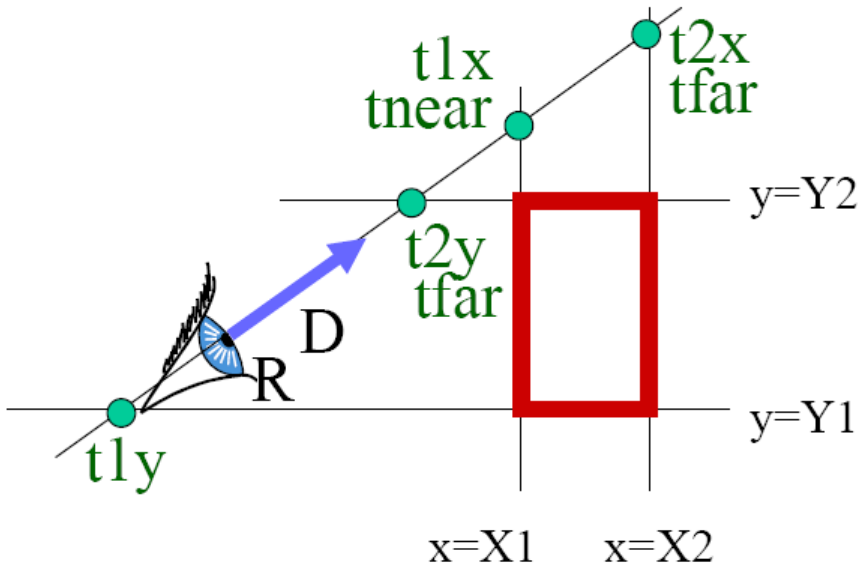
If not parallel

- Calculate intersection distance t1 and t2
  - t1=(X1-Rx)/Dx
  - t2=(X2-Rx)/Dx



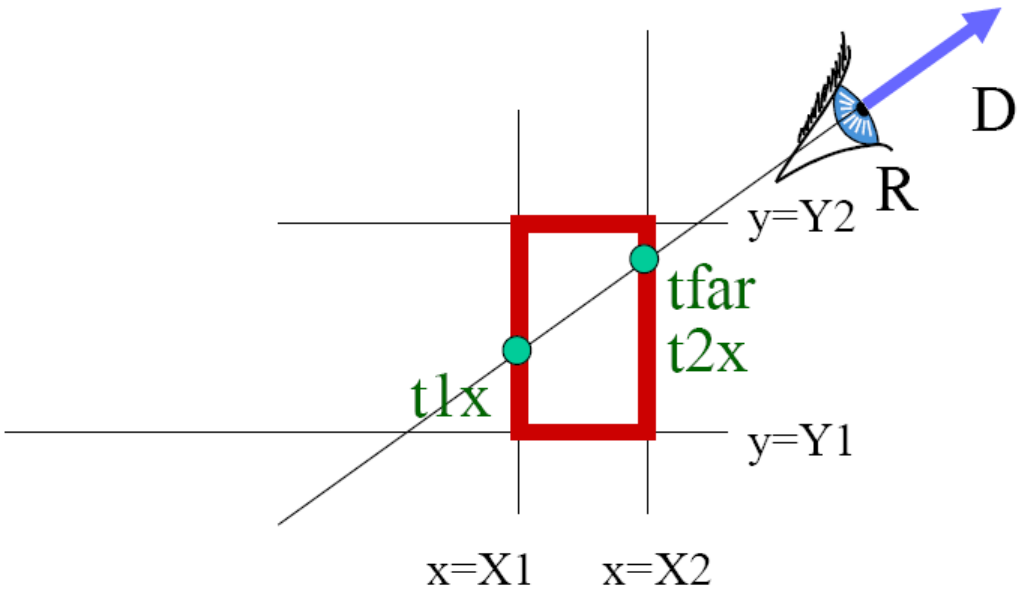
Test 1

- Maintain tnear and tfar
  - If t1>t2, swap
  - if t1>tnear, tnear =t1 if t2<tfar, tfar=t2
- If tnear>tfar, box is missed



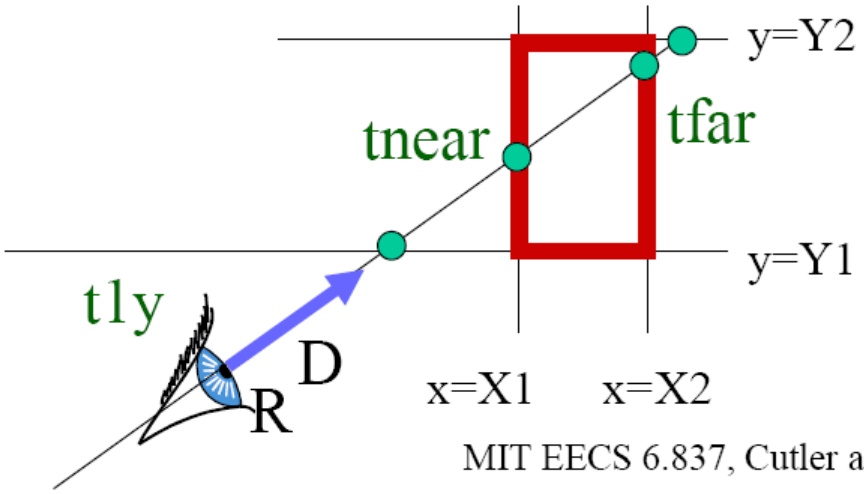
Test 2

- If tfar<0, box is behind



Algorithm recap

- Do for all 3 axis
  - Calculate intersection distance t1 and t2
  - Maintain tnear and tfar
  - If tnear>tfar, box is missed
  - If tfar<0, box is behind
- If box survived tests, report intersection at tnear

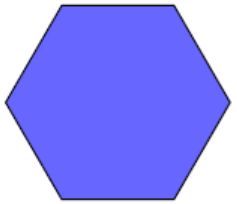


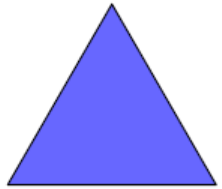
Efficiency issues

- Do for all 3 axis
  - Calculate intersection distance t1 and t2
  - Maintain tnear and tfar
  - If tnear>tfar, box is missed
  - If tfar<0, box is behind
- If box survived tests, report intersection at tnear
- 1/Dx, 1/Dy and 1/Dz can be precomputed and shared for many boxes
- Unroll the loop
  - Loops are costly (because of termination if)
  - Avoids the tnear tfar for X dimension

Overview of today

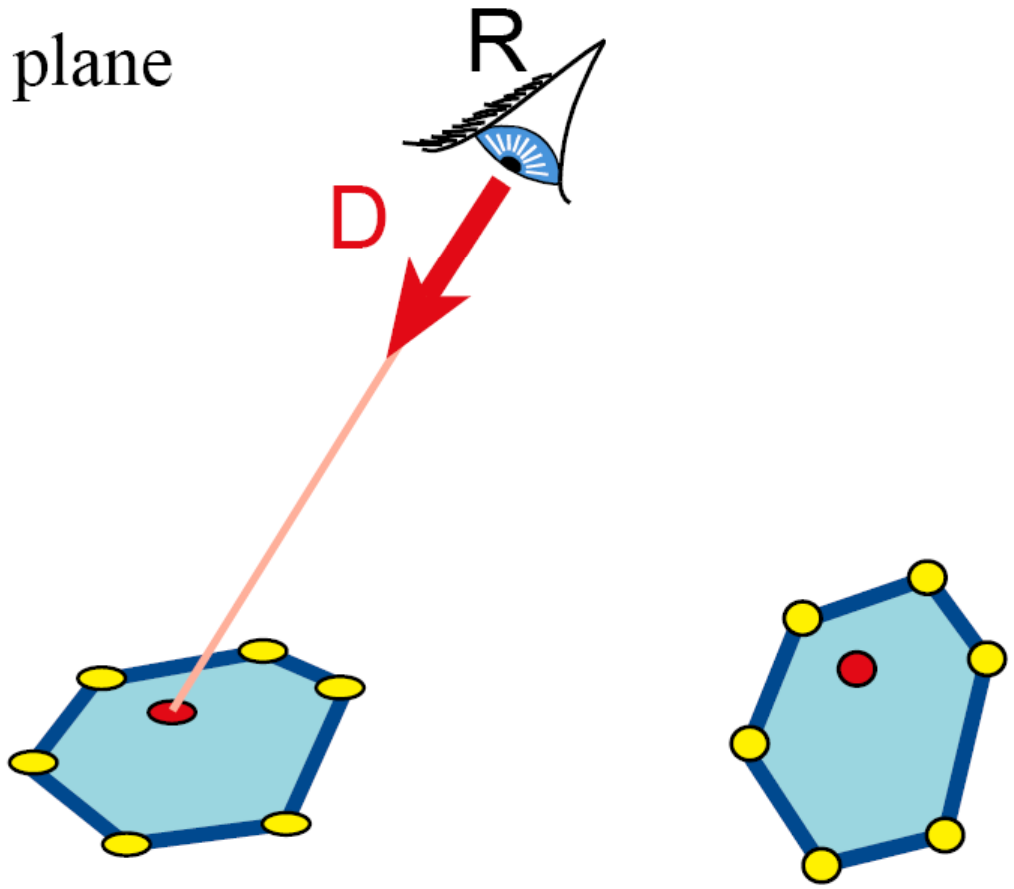
- Ray-box intersection
- Ray-polygon intersection
- Ray-triangle intersection





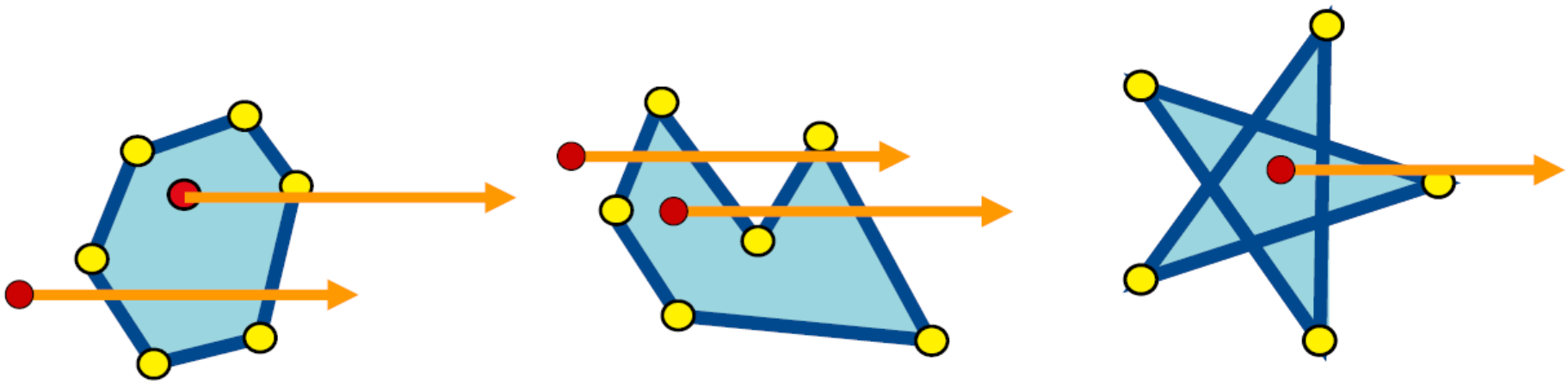
Ray-polygon intersection

- Ray-plane intersection
- Test if intersection is in the polygon
  - Solve in the 2D plane



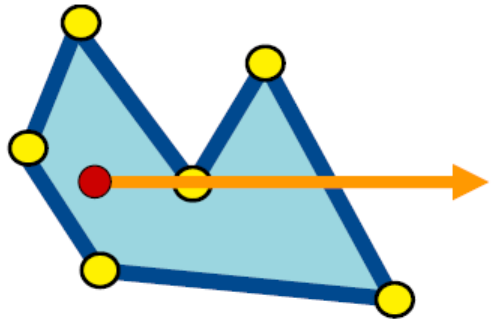
Point inside/outside polygon

- Ray intersection definition:
  - Cast a ray in any direction
    - (axis-aligned is smarter)
  - Count intersection
  - If odd number, point is inside
- Works for concave and star-shaped



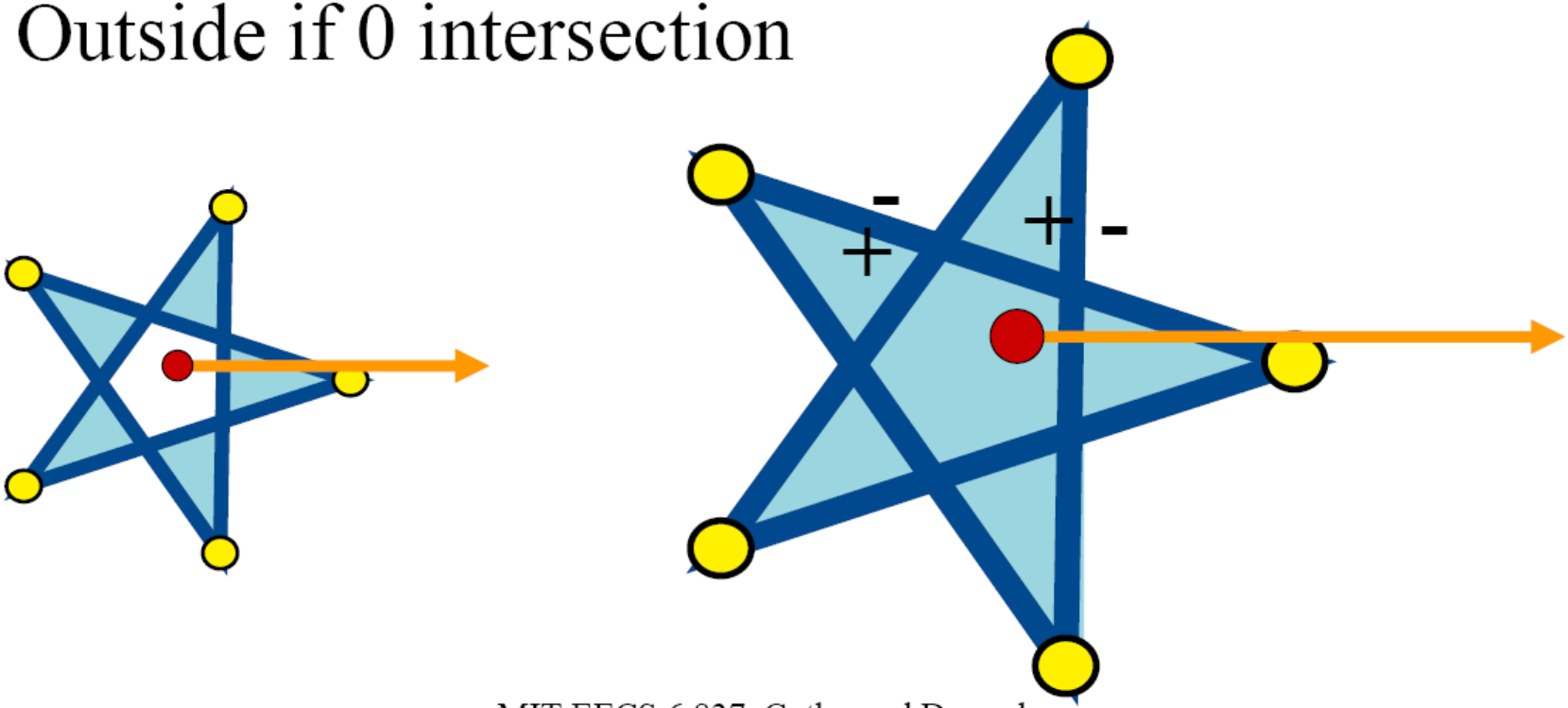
Precision issue

- What if we intersect a vertex?
  - We might wrongly count an intersection for each adjacent edge
- Decide that the vertex is always above the ray



Winding number

- To solve problem with star pentagon
- Oriented edges
- Signed number of intersection
- Outside if 0 intersection

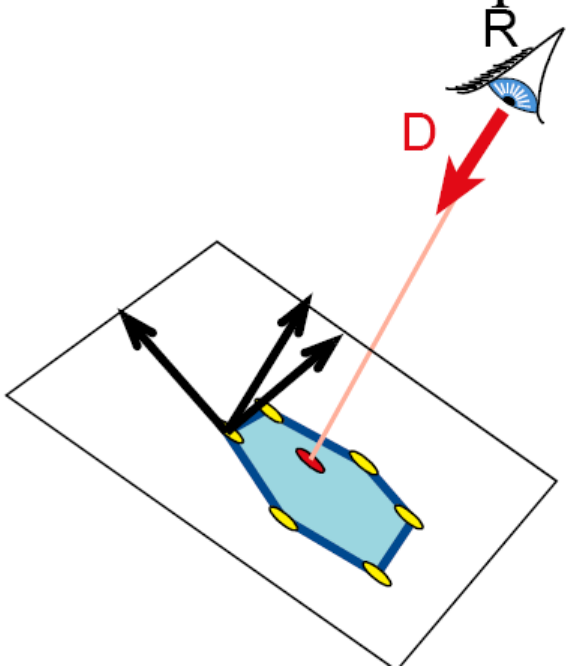


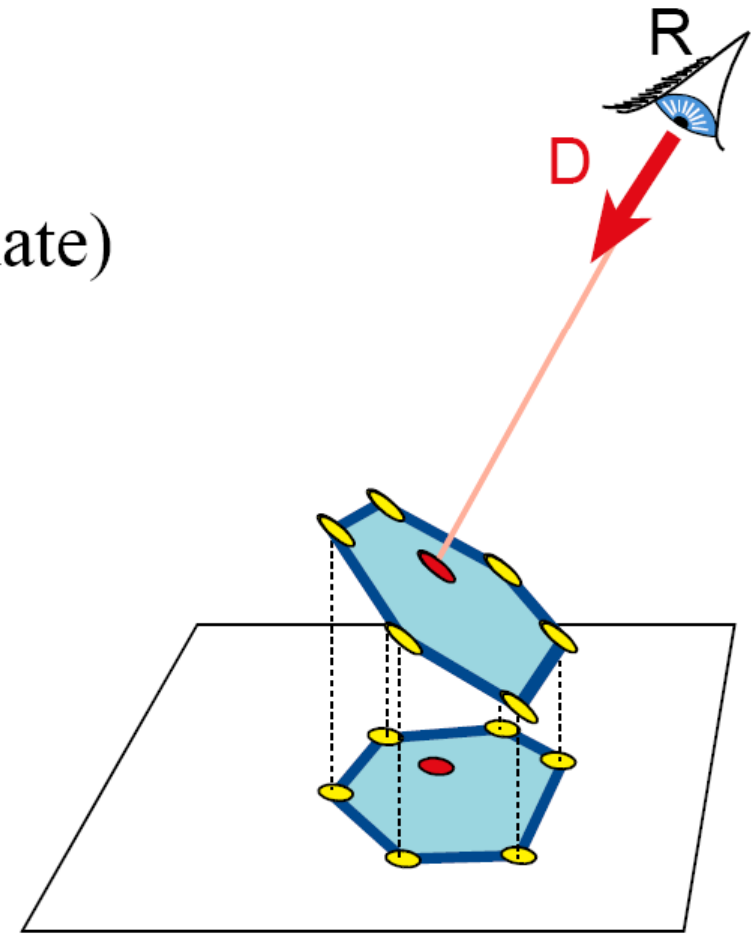
Alternative definitions

- Sum of the signed angles from point to vertices
  - 360 if inside, 0 if outside
- Sum of the signed areas of point-edge triangles
  - Area of polygon if inside, 0 if outside

How do we project into 2D?

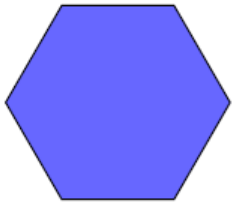
- Along normal
  - Costly
- Along axis
  - Smarter (just drop 1 coordinate)
  - Beware of parallel plane

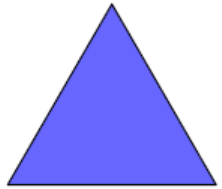




Overview of today

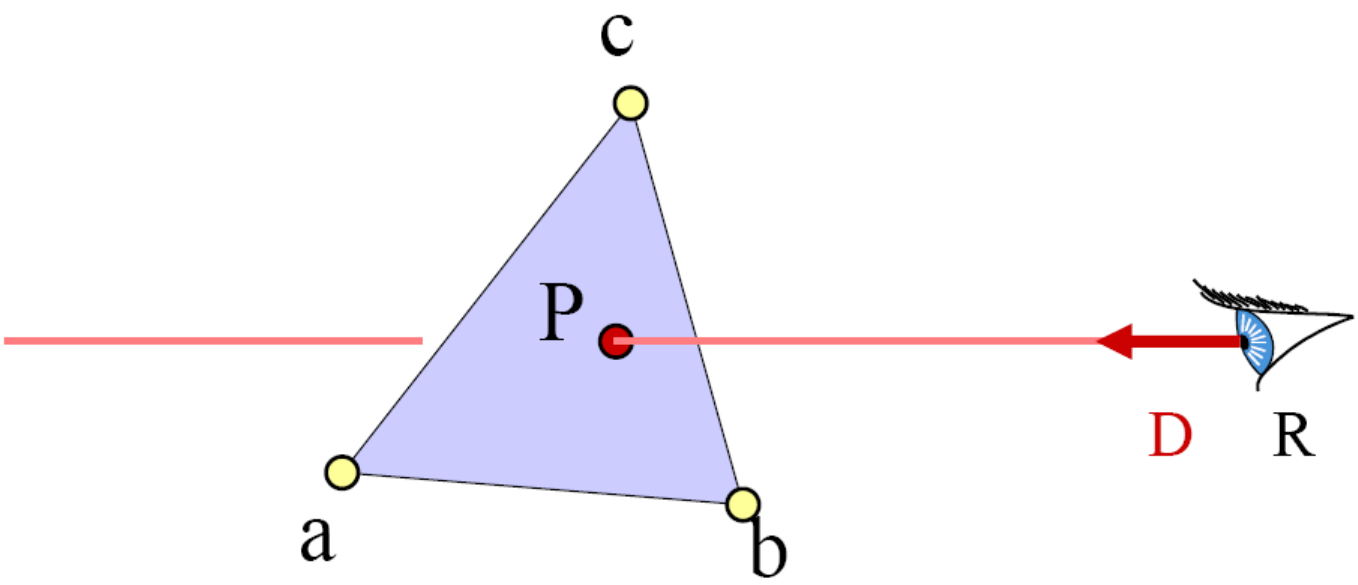
- Ray-box intersection
- Ray-polygon intersection
- Ray-triangle intersection





Ray triangle intersection

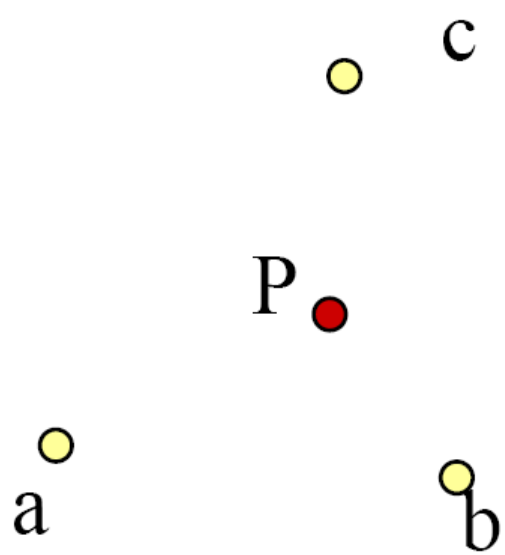
- Use ray-polygon
- Or try to be smarter
  - Use barycentric coordinates



Barycentric definition of a plane

- $P(\alpha, \beta, \gamma)=\alpha a+\beta b+\gamma c$ with $\alpha + \beta + \gamma = 1$
- Is it explicit or implicit?

[Möbius, 1827]



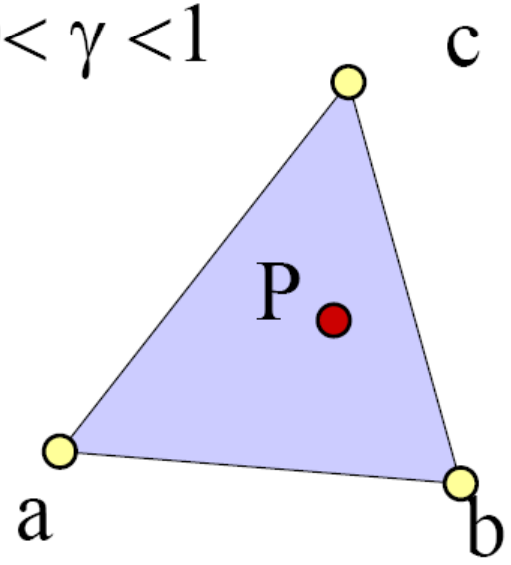
Barycentric definition of a triangle

- $P(\alpha, \beta, \gamma)=\alpha a+\beta b+\gamma c$
  with $\alpha + \beta + \gamma = 1$

$0< \alpha <1$

$0< \beta <1$

$0< \gamma <1$

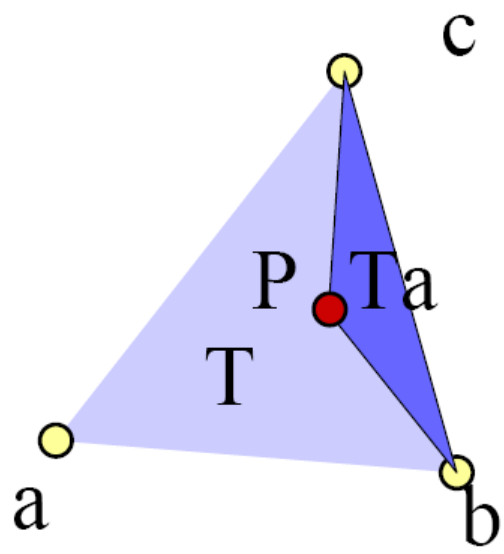


Given P, how can we compute α, β, γ ?

- Compute the areas of the opposite subtriangle
  - Ratio with complete area

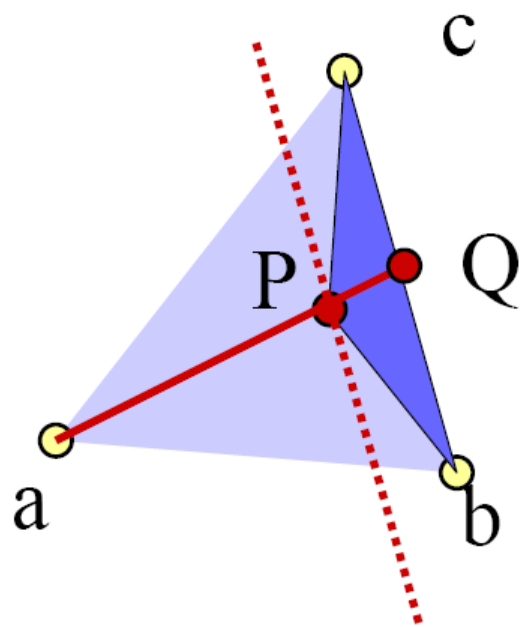
$$\alpha = A_a/A, \qquad \beta = A_b/A \qquad \gamma = A_c/A$$

Use signed areas for points outside the triangle



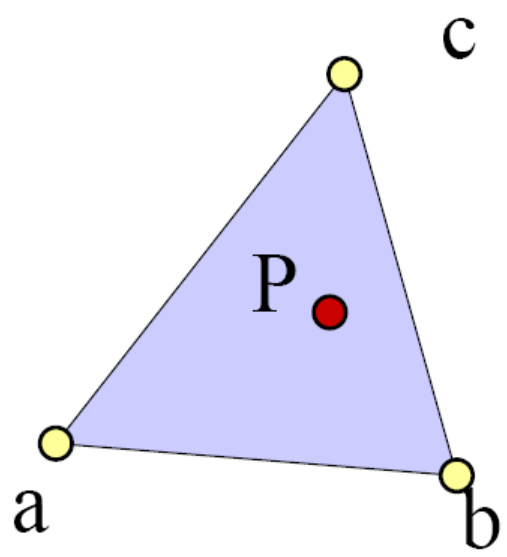
Intuition behind area formula

- P is barycenter of a and Q
- A is the interpolation coefficient on aQ
- All points on line parallel to bc have the same α
- All such Ta triangles have same height/area



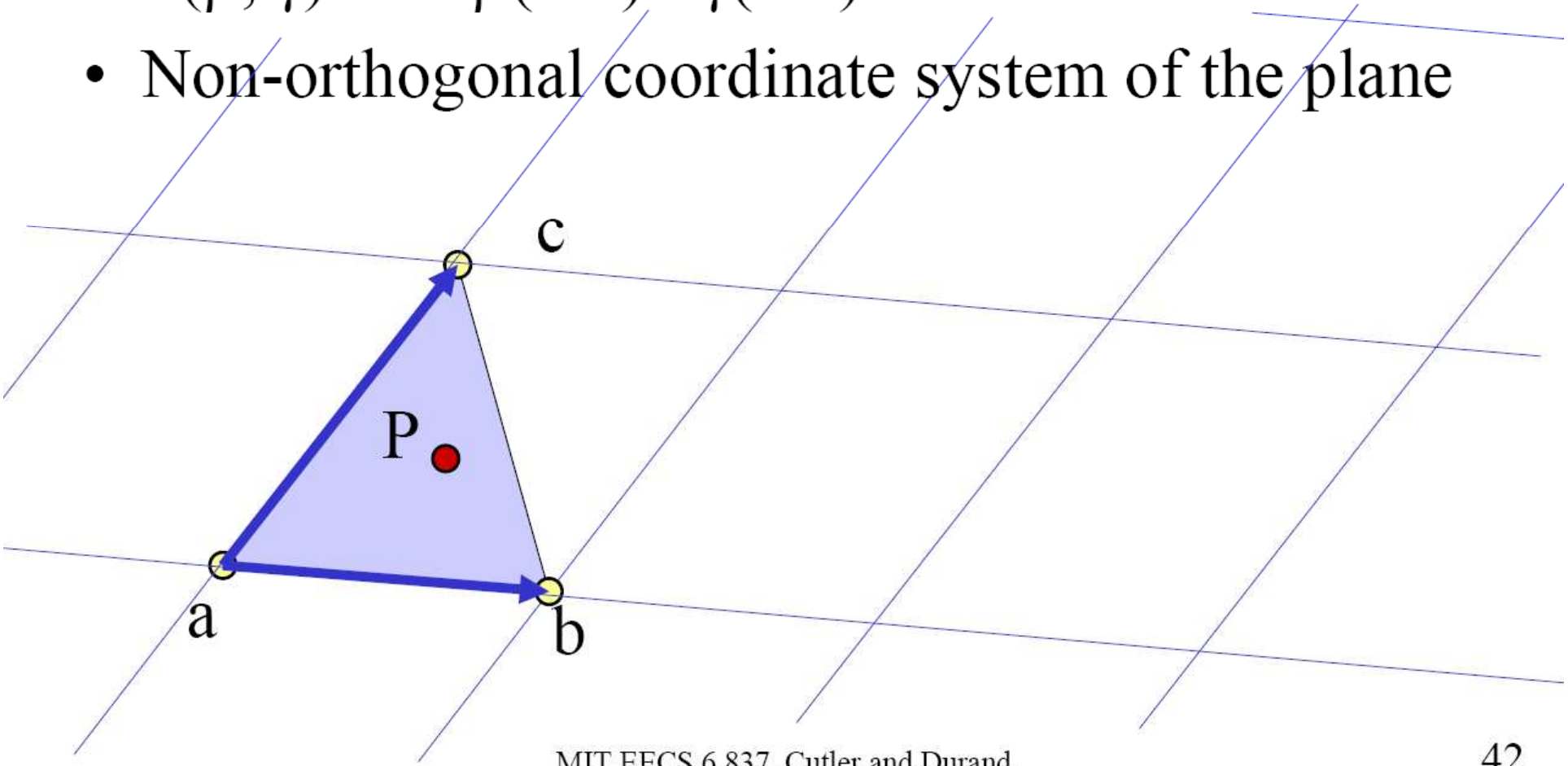
Simplify

- Since $\alpha + \beta + \gamma = 1$
  we can write $\alpha = 1 - \beta - \gamma$
- $P(\beta, \gamma) = (1-\beta-\gamma)\, a + \beta b + \gamma c$



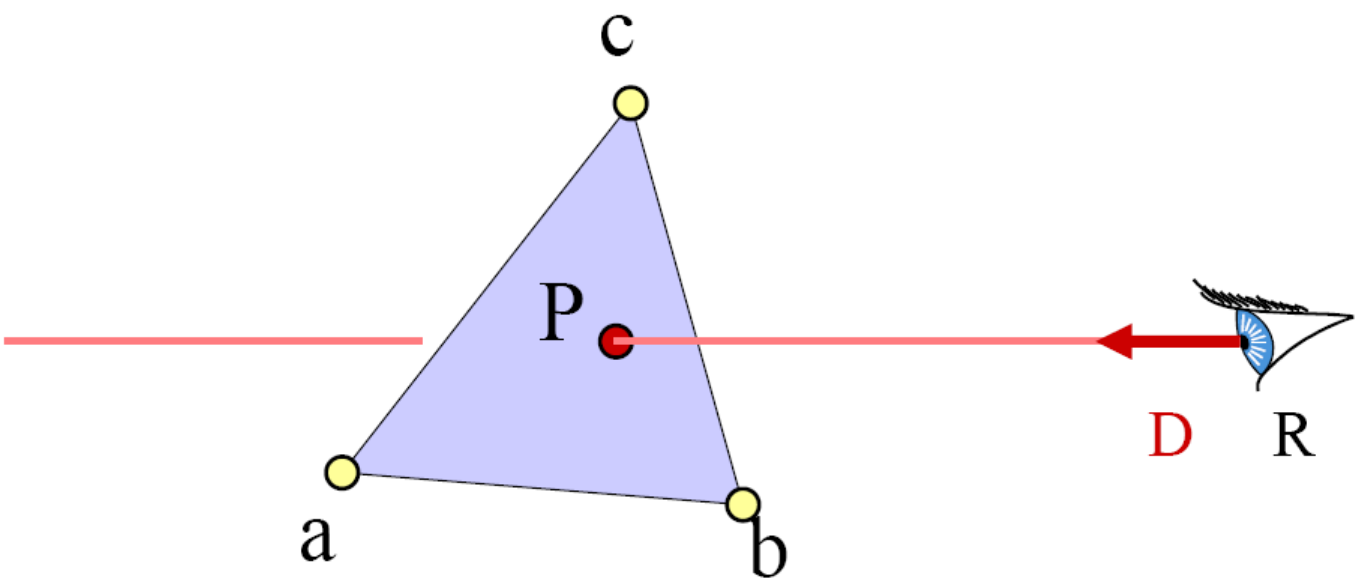
Simplify

- $P(\beta, \gamma)=(1-\beta-\gamma)\, a + \beta b + \gamma c$
- $P(\beta, \gamma)=a + \beta(b-a) + \gamma(c-a)$
- Non-orthogonal coordinate system of the plane



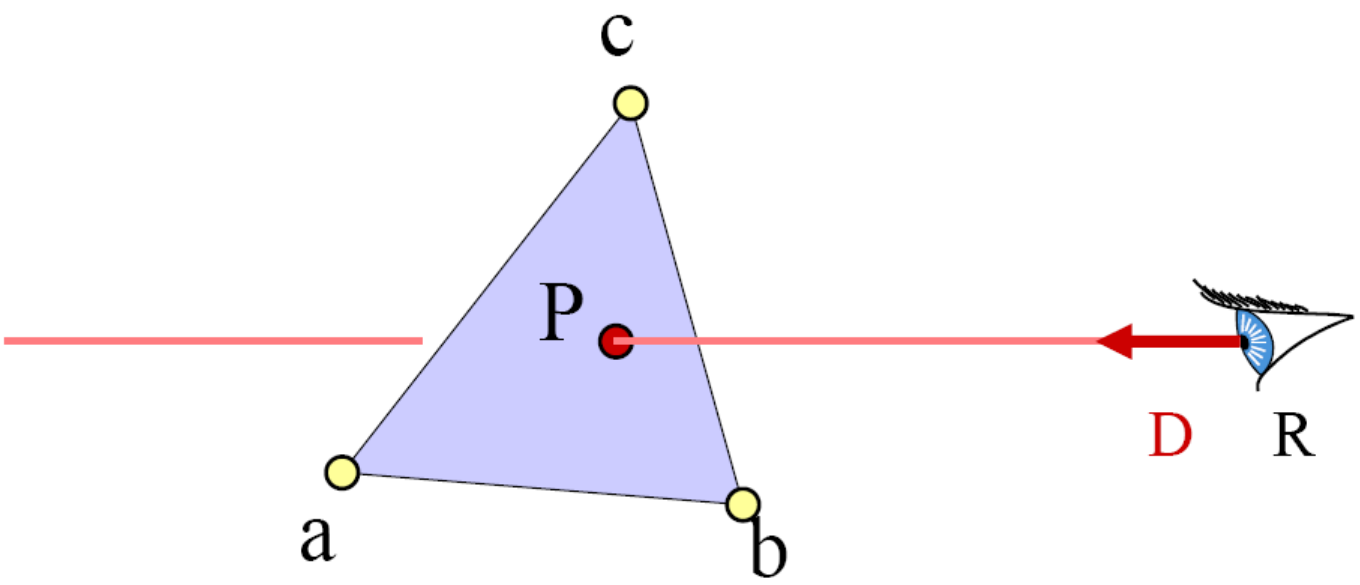
How do we use it for intersection?

- Insert ray equation into barycentric expression of triangle
- $P(t)= a+\beta\,(b-a)+\gamma\,(c-a)$
- Intersection if $\beta+\gamma<1$; $0<\beta$ and $0<\gamma$



Intersection

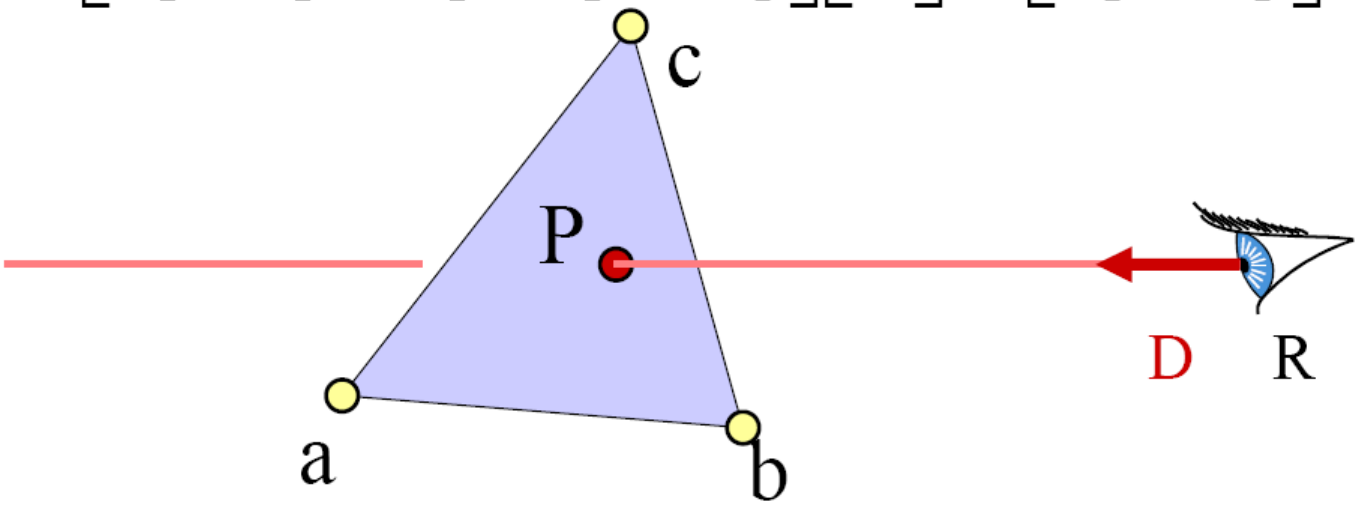
- $R_x+tD_x= a_x+\beta\ (b_x-a_x)+\gamma\ (c_x-a_x)$
- $R_y+tD_y= a_y+\beta\ (b_y-a_y)+\gamma\ (c_y-a_y)$
- $R_z+tD_z= a_z+\beta\ (b_z-a_z)+\gamma\ (c_z-a_z)$



Matrix form

- $R_x+tD_x= a_x+\beta (b_x-a_x)+\gamma (c_x-a_x)$
- $R_y+tD_y= a_y+\beta (b_y-a_y)+\gamma (c_y-a_y)$
- $R_z+tD_z= a_z+\beta (b_z-a_z)+\gamma (c_z-a_z)$

$$\begin{bmatrix} a_x-b_x & a_x-c_x & D_x \\ a_y-b_y & a_y-c_y & D_y \\ a_z-b_z & a_z-c_z & D_z \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x-R_x \\ a_y-R_y \\ a_z-R_z \end{bmatrix}$$

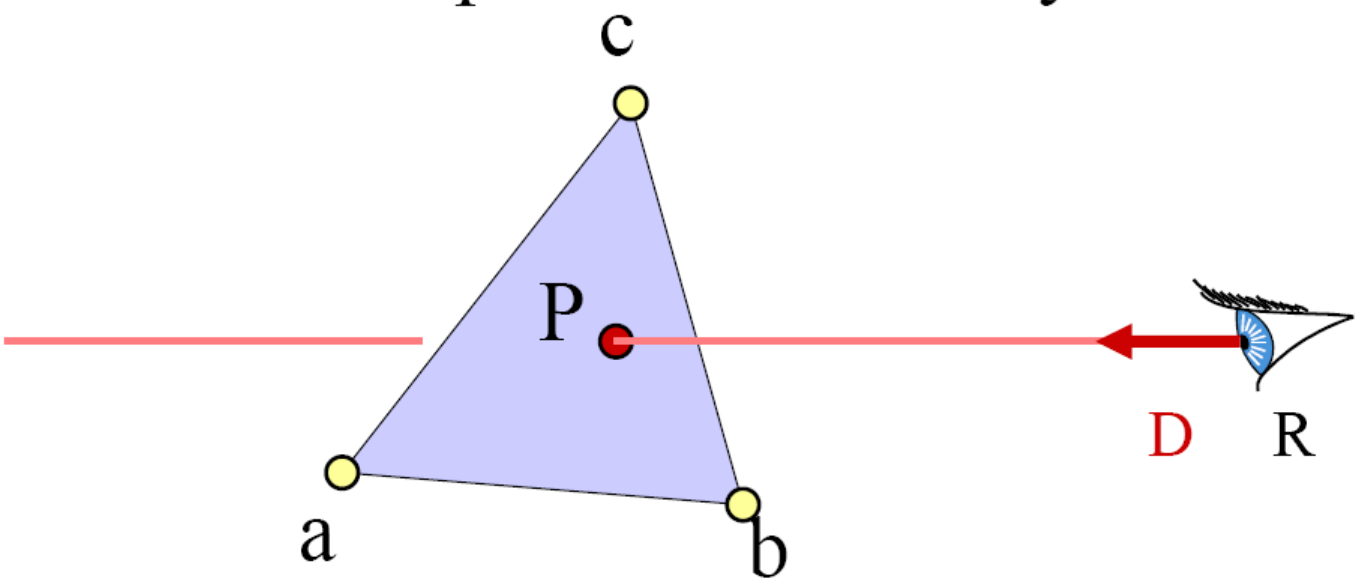


Cramer's rule

- | | denotes the determinant

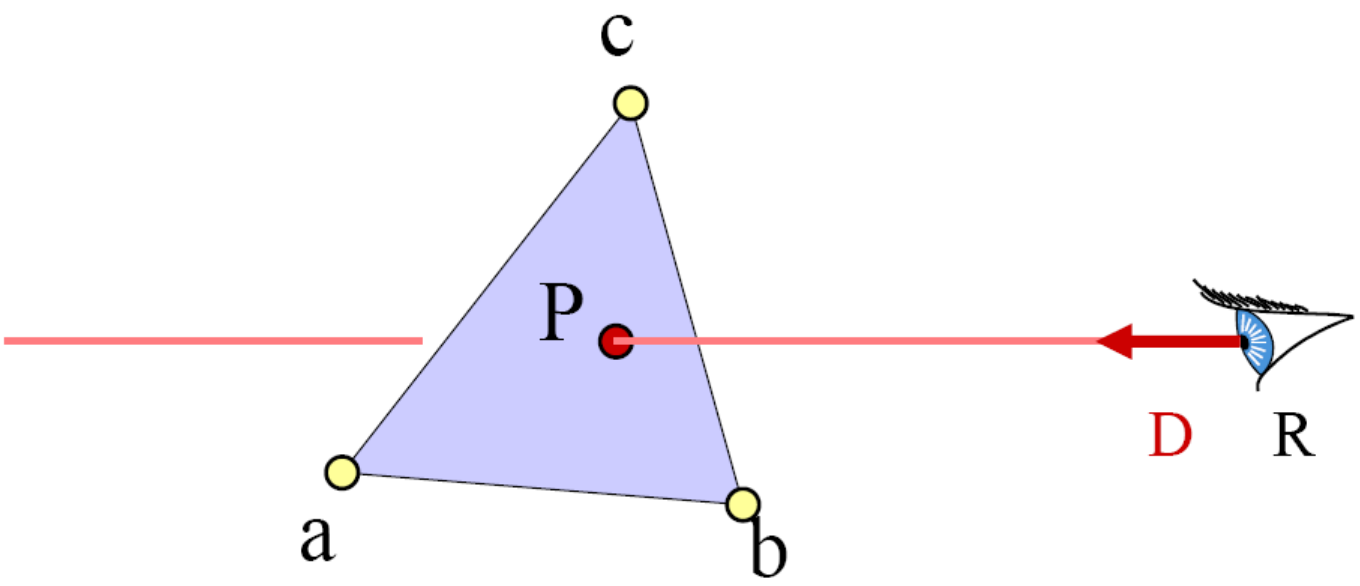
$$\beta = \frac{\begin{vmatrix} a_x - R_x & a_x - c_x & D_x \\ a_y - R_y & a_y - c_y & D_y \\ a_z - R_z & a_z - c_z & D_z \end{vmatrix}}{|A|} \qquad \gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - R_x & D_x \\ a_y - b_y & a_y - R_y & D_y \\ a_z - b_z & a_z - R_z & D_z \end{vmatrix}}{|A|} \qquad t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_x \\ a_y - b_y & a_y - c_y & a_y - R_y \\ a_z - b_z & a_z - c_z & a_z - R_z \end{vmatrix}}{|A|}$$

- Can be copied mechanically in the code



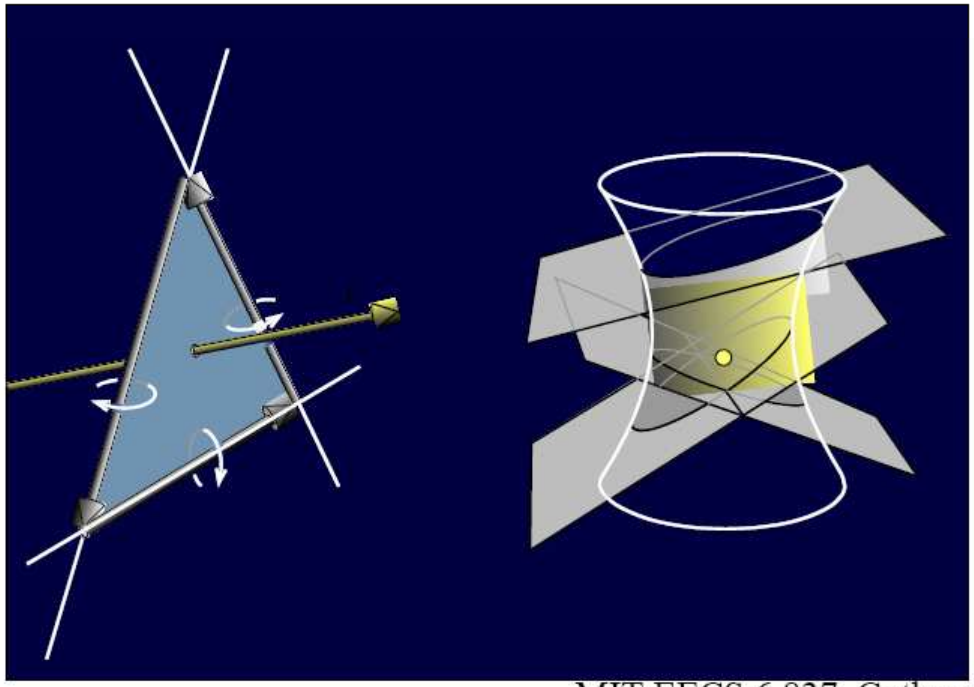
Advantage

- Efficient
- Store no plane equation
- Get the barycentric coordinates for free
  - Useful for interpolation, texture mapping



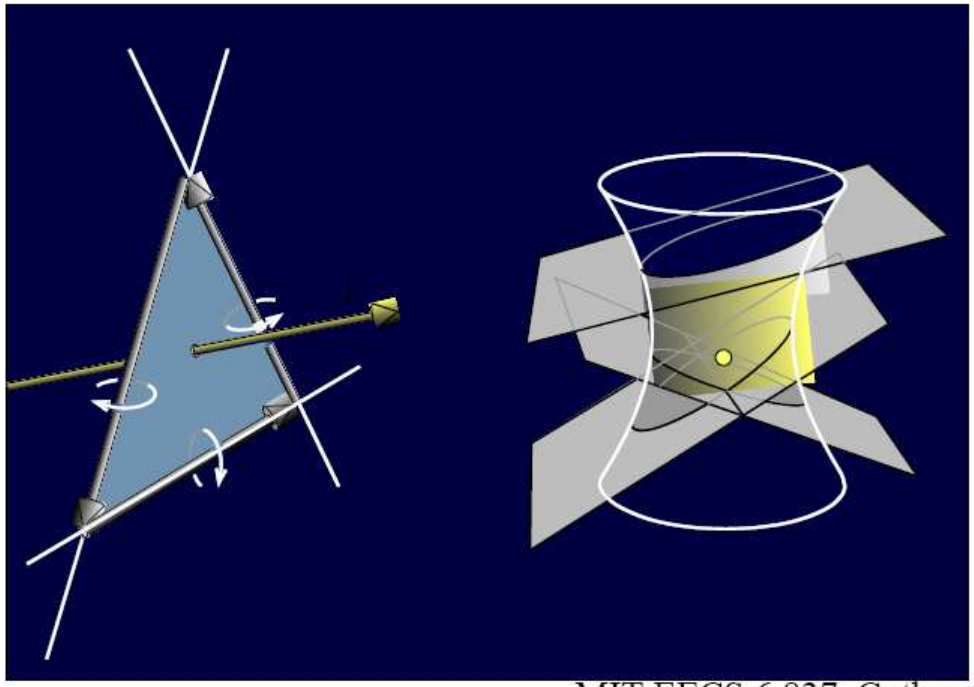
Plucker computation

- Plucker space:
  6 or 5 dimensional space describing 3D lines
- A line is a point in Plucker space



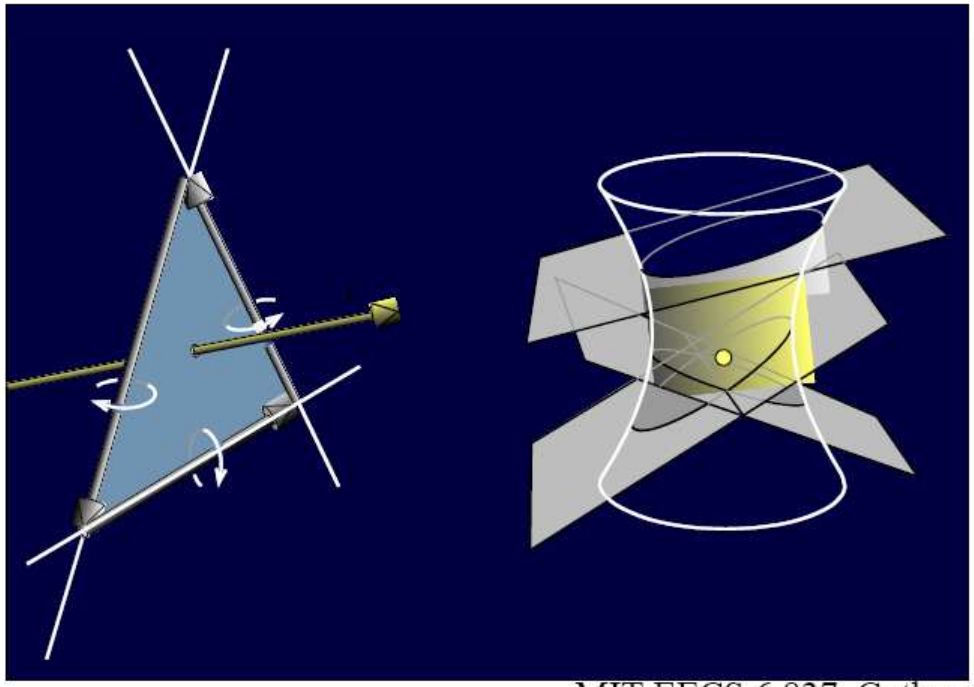
Plucker computation

- The rays intersecting a line are a hyperplane
- A triangle defines 3 hyperplanes
- The polytope defined by the hyperplanes is the set of rays that intersect the triangle



Plucker computation

- The rays intersecting a line are a hyperplane
- A triangle defines 3 hyperplanes
- The polytope defined by the hyperplanes is the set of rays that intersect the triangle



- Ray-triangle intersection becomes a polytope inclusion
- Couple of additional issues