

Computer Graphics

Dr. Norman I. Badler
Professor, Computer & Information Science
Director, SIG Center for Computer Graphics

University of Pennsylvania

CIS 460 / CIS 560

Fall 2013

Part B

© Norman I. Badler

(except where otherwise noted)

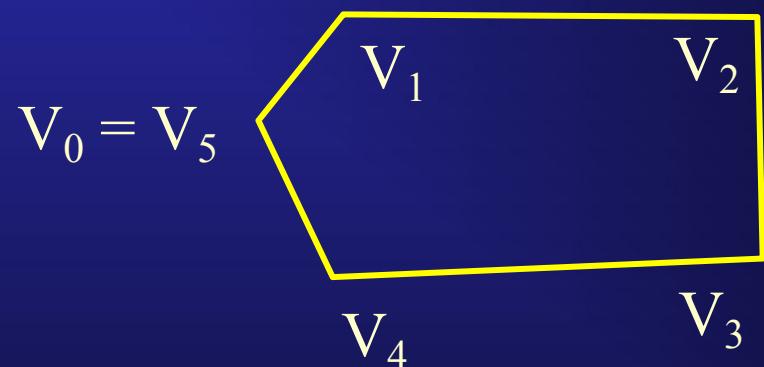
Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Properties of Points, Lines and Polygons in the Plane (Leads into the general topic of “Computational Geometry”)

- Intersecting line segments?
- Degenerate cases
- Am I a polygon?
- Turning angle?
- Convex?
- Point on line?
- Point in half-plane?
- Point-in-polygon?
- Bounding boxes
- Polygon clipping
- Line-polygon intersection (needed for scanline algorithms)
- Polygon-polygon boolean operations: union, intersection and difference

Polygon Data Structure



$$\text{Polygon} = (V_0 \ V_1 \ V_2 \ V_3 \ V_4 \ V_5)$$

That is, an ordered list of vertices (x_i, y_i) or (x_i, y_i, z_i) that implicitly define the edges of the polygon.

Polygon = Simple Closed “Curve”

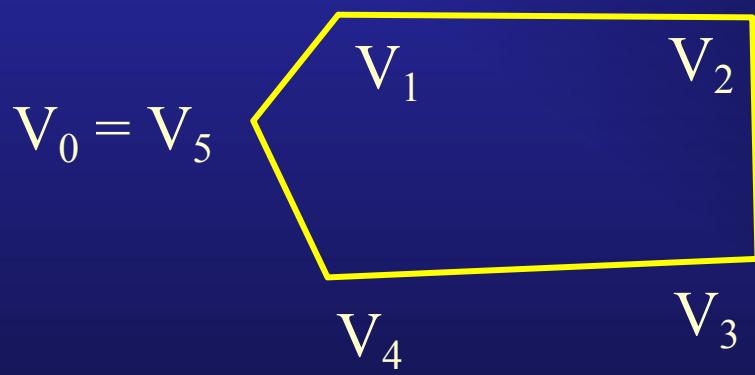
This is the mathematical definition of a polygon. We need to realize this computationally:

- **Curve:** For us, edges are straight line segments (a polyline).
- **Closed:** “First” vertex = “last” vertex.
- **Simple:** Polyline does not self-intersect.

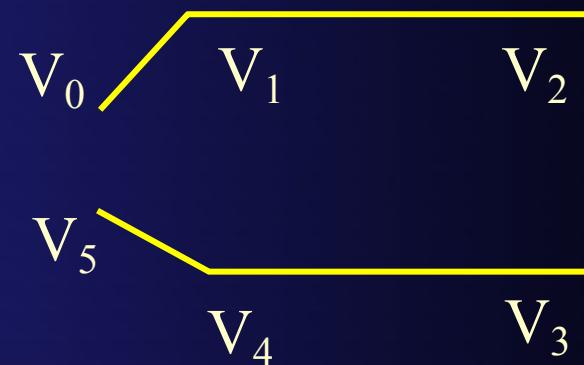
Let's see how to check for these properties to better understand the polygon representation.

Closed

- Polygon $P = (V_0, V_1, \dots, V_{n-1}, V_n)$ where $V_n = V_0$

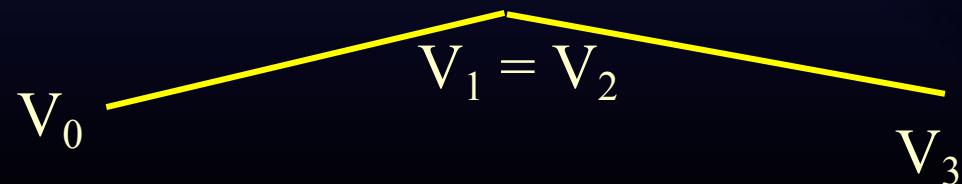


Closed



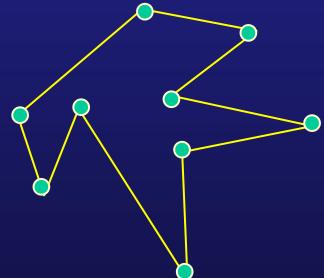
Not closed (polyline)

- Assume all vertices are distinct, that is: $V_i \neq V_{i+1} \ \forall i = 1, \dots, n$.
- Check for and prevent, e.g.:

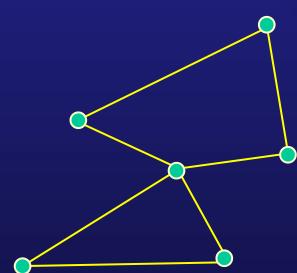


Simple

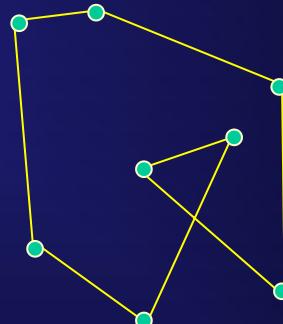
- Simple means the polyline does not intersect itself.
- Several cases are possible:



Simple



Not simple:
intersection
point appears
twice in
vertex list



Not simple:
two edges
actually
intersect



Not simple:
two edges
are coincident

Checking for Intersections

- A polyline $P = [V_0 \ V_1 \ V_2 \ \dots \ V_{n-1}]$ ($n \geq 3$) that is closed ($V_0 = V_{n-1}$) is really a polygon if P does not *self-intersect*.
- P intersects itself if

(1) $V_i = V_j$ for some $i \neq j$ ($0 \leq i, j \leq n-2$)

OR

(2) $\exists i, j$ such that $V_i \ V_{i+1} \cap V_j \ V_{j+1} \neq \emptyset$ for $i \neq j$ ($0 \leq i, j \leq n-2$)
unless they share a vertex $V_{i+1} = V_j$

- Therefore must test if two line segments intersect...

Representing a Line Segment

- Can't use slope intercept form $y = mx + b$
 - lines may be vertical ($m = \infty$)
 - this form does not represent the finite line segment
- Use linear *parametric* form:

$$V = (x', y') = V_1 (1 - t) + V_2 t \quad t \in [0, 1]$$



- that is:

$$x' = x_1 (1 - t) + x_2 t \quad \text{and} \quad y' = y_1 (1 - t) + y_2 t$$



Equivalent Parametric Form

$$\begin{aligned} V = (x', y') &= V_1(1 - t) + V_2 t \\ &= V_1 - V_1 t + V_2 t \\ &= V_1 + (V_2 - V_1)t \end{aligned}$$

This uses only one vector multiplication.

Works for 3D lines as well:

$$V = (x', y', z')$$

Linear Interpolation (LERP)

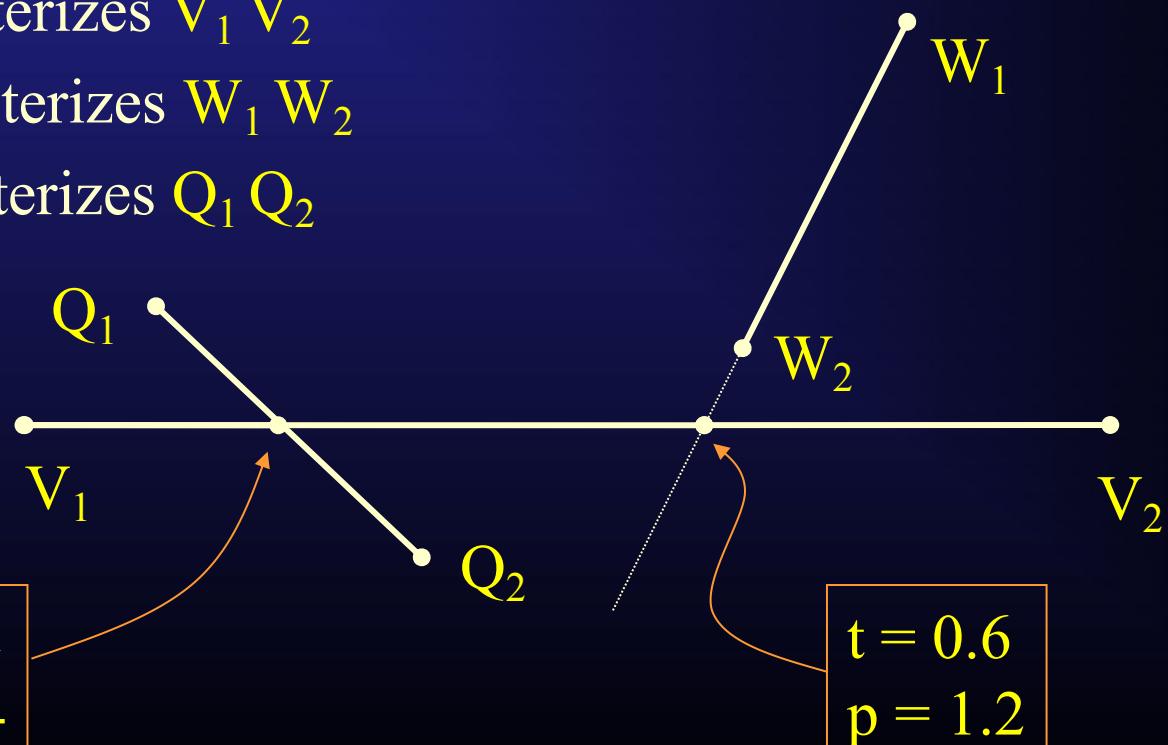
- This parametric form is also called Linear Interpolation (LERP).
- It will be very useful for both geometry and animation.
- For example, we can step any transformation by linearly interpolating its parameters:

```
range = param_end - param_start;  
for (i = 0; n; i++){ // 0, 1, 2, ..., n; n+1 steps  
    Transformation(param_start + (i/n) * range)  
}
```

- E.g., check how this works for Transformation = a rotation by 90° and n=4 steps.

Intersecting Two Line Segments in Parametric Form

- Two line segments intersect if and only if there is a point on both with parametric representation such that BOTH parameters lie in the range $[0, 1]$:
- t parameterizes $V_1 V_2$
- p parameterizes $W_1 W_2$
- r parameterizes $Q_1 Q_2$



Intersection Calculation

V_1, V_2 ($V_1 \neq V_2$) and W_1, W_2 ($W_1 \neq W_2$) intersect if a point V lies on both for t, p such that $t \in [0,1]$ and $p \in [0,1]$

$$V = V_1 + (V_2 - V_1)t = W_1 + (W_2 - W_1)p$$

rewriting this:

$$0 = V_1 + (V_2 - V_1)t - W_1 - (W_2 - W_1)p$$

$$W_1 - V_1 = (V_2 - V_1)t - (W_2 - W_1)p$$

$$W_1 - V_1 = (V_2 - V_1)t + (W_1 - W_2)p$$

which can be represented as the matrix:

$$\begin{bmatrix} W_{1x} - V_{1x} \\ W_{1y} - V_{1y} \end{bmatrix} = \begin{bmatrix} V_{2x} - V_{1x} & W_{1x} - W_{2x} \\ V_{2y} - V_{1y} & W_{1y} - W_{2y} \end{bmatrix} \begin{bmatrix} t \\ p \end{bmatrix}$$

Solving the Matrix Equation

- Rename the entries to simplify the math.
- All the entries except t and p are known from the line segment vertices and hence are constants.

$$\begin{bmatrix} E \\ F \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} t \\ p \end{bmatrix}$$

- Solve for t :

$$\begin{bmatrix} ED \\ -FB \end{bmatrix} = \begin{bmatrix} AD & BD \\ -CB & -BD \end{bmatrix} \begin{bmatrix} t \\ p \end{bmatrix} \text{ multiply by } D$$

$$\begin{bmatrix} ED \\ ED - FB \end{bmatrix} = \begin{bmatrix} AD & BD \\ AD - CB & 0 \end{bmatrix} \begin{bmatrix} t \\ p \end{bmatrix} \text{ add row 1 to row 2}$$

Solve for t and p

$$t = \frac{ED - FB}{AD - CB} \quad \text{and hence:} \quad p = \frac{AF - CE}{AD - CB}$$

- Provided that $AD - CB \neq 0$
- Thus intersection is a point on both segments if and only if $t \in [0,1]$ and $p \in [0,1]$: its coordinates are $V = V_1 + (V_2 - V_1)t$
- Actually, we use $t \in [0,1)$ and $p \in [0,1)$ so that adjacent line segments will not produce an intersection at their common vertex:



- But we still can't ignore the possibility that the denominator is 0. What does this mean?

Degenerate Cases

- If $AD - CB = 0$ then the line segments are parallel.
- But there are several possibilities:

- parallel



- collinear



- overlapped



- superimposed



Solving the Degenerate Cases

- If the line segments belong to the same line, then the p and t numerators

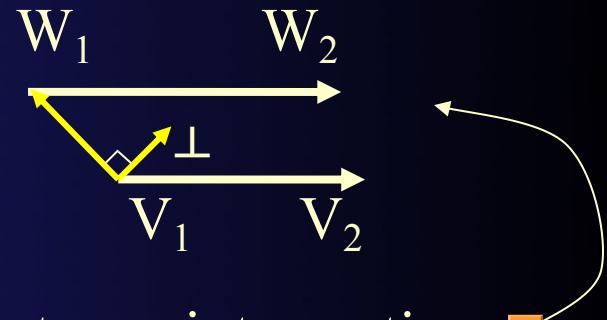
$$ED - FB = AF - CE = 0$$

- This is a consequence of (e.g.)*

$$\perp (\mathbf{W}_1 - \mathbf{V}_1) \cdot (\mathbf{V}_2 - \mathbf{V}_1) = 0$$

- So if this is false, the lines are parallel but non-intersecting. ■
- And if this is true, then this reduces to a 1D problem: the segments intersect if and only if the segments

$[0, \|\mathbf{V}_1\mathbf{V}_2\|]$ and $[\|\mathbf{V}_1\mathbf{W}_1\|, \|\mathbf{V}_1\mathbf{W}_2\|]$ intersect.



* Recall definition of *dot product* of two vectors: $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$

$$p \cdot q = p_x q_x + p_y q_y + p_z q_z$$

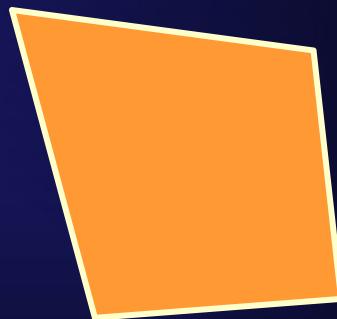
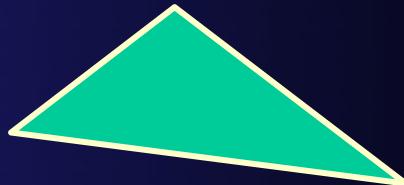
$p \cdot q = 0$ iff p and q are *orthogonal (perpendicular)*.

Simple Polygons, Conclusion

- We can now determine if a polygon is simple by checking every pair of edges for intersections.
- This yields an $O(n^2)$ algorithm.
- Actually, we can do better with a fancier data structure and a technique called **plane sweep**. See the discussion of the Watkins visible surface scan-line algorithm in these notes.

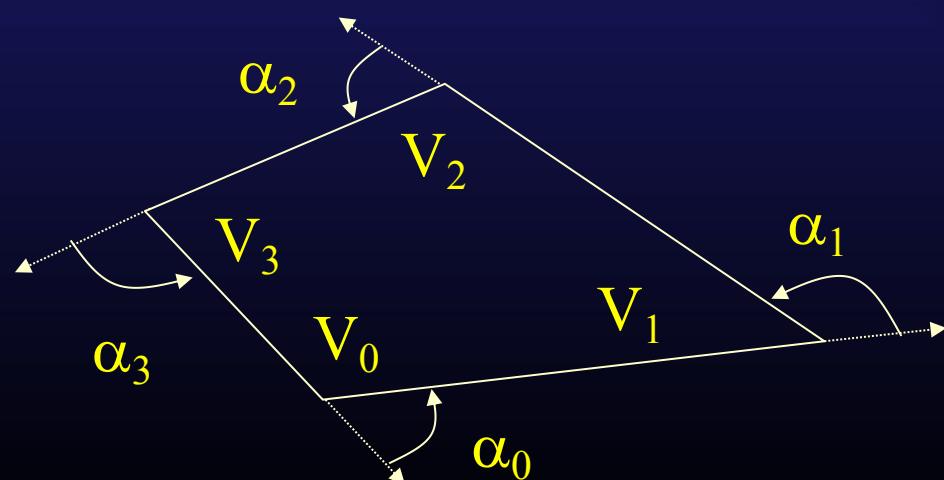
Most Common Polygons

- Tri(angle)s
 - Always planar
 - May be degenerate (slivers)
 - Always convex
- Quad(rilateral)s
 - Is likely non-planar
 - May be non-convex
 - Often split into triangles



Determining Edge-Edge Turning Angle

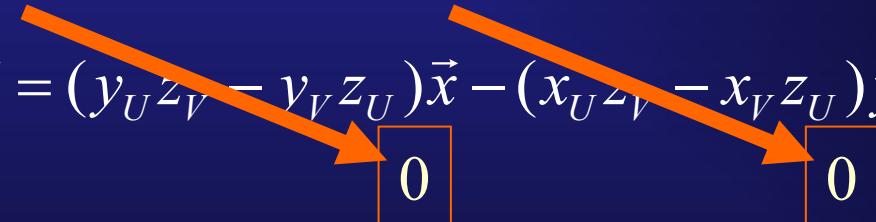
- Convex polygons are sometimes required by certain algorithms.
- (And OpenGL prefers convex polygons.)
- How to check for convexity?
- Polygon is convex if we always turn in the same sense or direction as we traverse the edges:



That is,
convex iff
 $\alpha_i \geq 0 \forall i$
OR
 $\alpha_i \leq 0 \forall i$
 $i=0, \dots, n-1$

To Determine Convexity

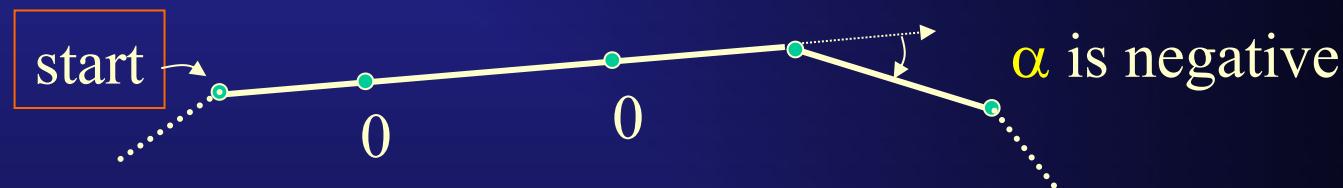
- Use vector cross product to get normal (perpendicular) and turning direction between successive polygon edges.
- Since our polygon can be assumed (without loss of generality for now) to lie in the x-y plane, the cross product simplifies greatly:

$$W = (y_U z_V - y_V z_U) \vec{x} - (x_U z_V - x_V z_U) \vec{y} + (x_U y_V - x_V y_U) \vec{z}$$


- This makes sense, since the perpendicular vector for a 2D polygon in the x-y plane must point straight out of the plane and thus must have zero x and y components.
- All we need is the *sign* of the boxed term for successive pairs of polygon edges to establish the angle α direction: positive or negative.

Convexity Algorithm

- Check cross product z magnitude of successive polygon edge pairs until a non-zero value is found.

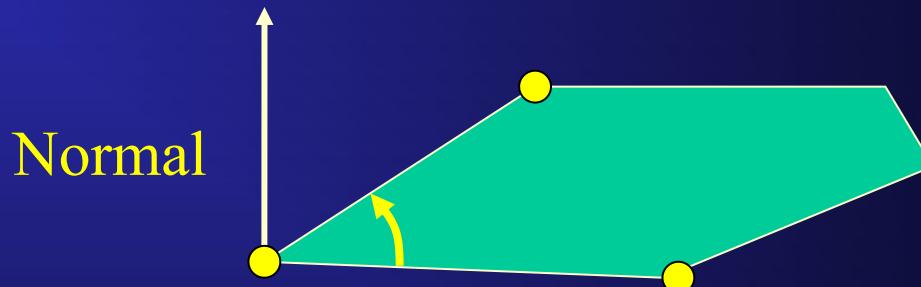


- Continue around the polygon. If each turn has the same sense as turn from above (or 0), then polygon is convex.
- Clearly $O(n)$.

Computing the Polygon Normal from its Plane Equation

- Given plane equation $Ax + By + Cz + D = 0$
- The normal = $(a, b, c) = (A, B, C)/\|(A, B, C)\|$
- i.e., the coefficients are interpreted as a vector (A, B, C) and then normalized to length 1.
- Simple reality check: e.g., x-y plane is $z=0$, so $A=B=0$ and $C=1$, so normal is $(0,0,1)$!

Computing the Normal from a Polygon Face (Don't need to find the plane equation first!)



Normal = cross product of 2 edges formed by 3 non-collinear vertices; or to avoid special cases and numerical error (Newell)
(m = number of vertices in polygon face):

$$j = (\text{if } i = m - 1 \text{ then } 0 \text{ else } i + 1)$$

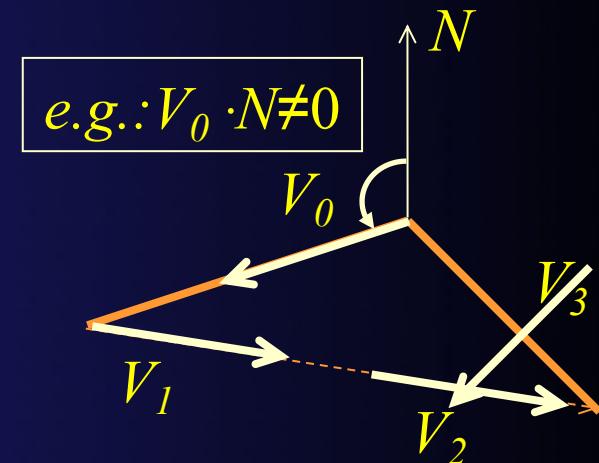
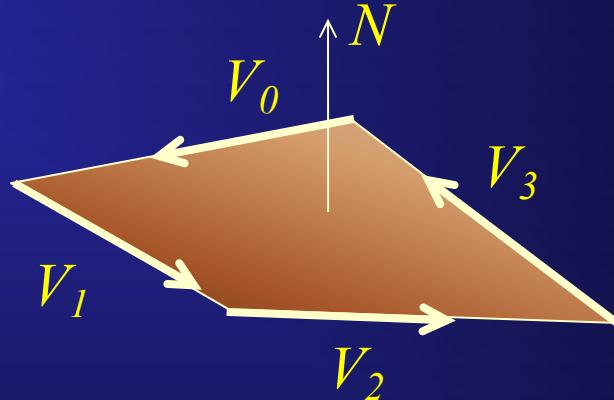
$$\text{Normal } (a, b, c) = \begin{cases} a = \sum_{i=0}^{m-1} (Y_i - Y_j)(Z_i + Z_j) \\ b = \sum_{i=0}^{m-1} (Z_i - Z_j)(X_i + X_j) \\ c = \sum_{i=0}^{m-1} (X_i - X_j)(Y_i + Y_j) \end{cases}$$

Planarity

- Any polygon $P = [P_i], i = [0..k-1]$ with more than $k = 3$ (distinct) 3D vertices is apt to be non-planar due to numerical error in the finite representation of floating point numbers.
(That's why triangles are graphically attractive.)
- So how to tell if a polygon with $k \geq 4$ vertices is planar?
- Choose an epsilon ε . (ε can be a constant such as $1.0e-4$, but it would be better to make it relative to the size of the space in which the 3D coordinates are found)
- Compute $m = \text{MAX}(V_i \cdot N)$ for $i \in [0..k-1]$ where V_i is the unit length vector from P_i to P_{i+1} , and N is the computed unit length normal at the polygon centroid (e.g., the formula that uses all vertices of the polygon in the normal vector estimate).
- P is “planar” if $m < \varepsilon$.

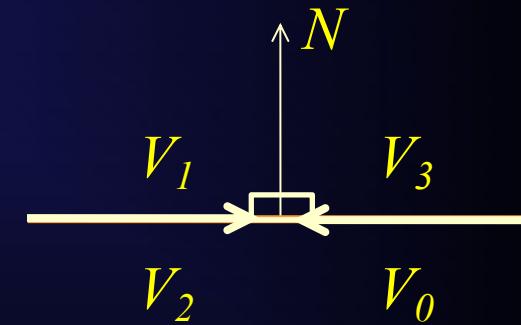
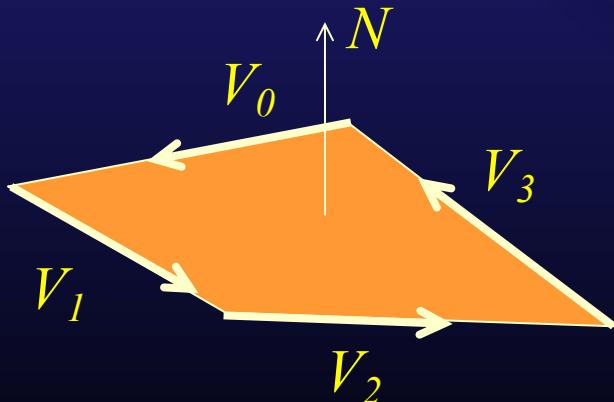
Examples

- Non-planar



edge-on view, e.g.

- Planar

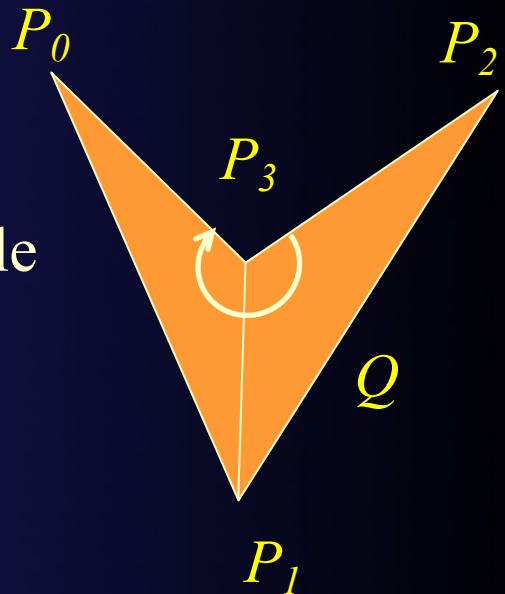


all edges $V_i \cdot N = 0$

Convexity

- A triangle is obviously convex.
- Quad Q is convex or else concave like this:

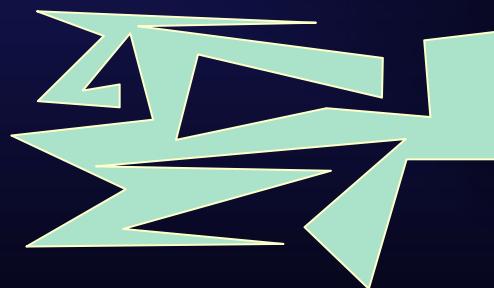
P_3 is a **reflex** or **concave** vertex: its internal angle is $>180^\circ$.



- We already saw how to test for this easily using the cross product.
- Convince yourself that a *quad* cannot have more than one reflex vertex!
- If Q has a reflex vertex (P_i) then split into two triangles from P_i to $P_{(i+2) \bmod 4}$.

What about Polygons with more than 4 Vertices?

- For general polygons with k vertices, this can become a difficult problem if one aims for a *minimal* (fewest convex pieces) decomposition.
- We'll look at one method that always works and is quite efficient but does not give a minimal decomposition. It does guarantee the convex pieces are quads or triangles, though! (That's great for OpenGL or GPUs.)



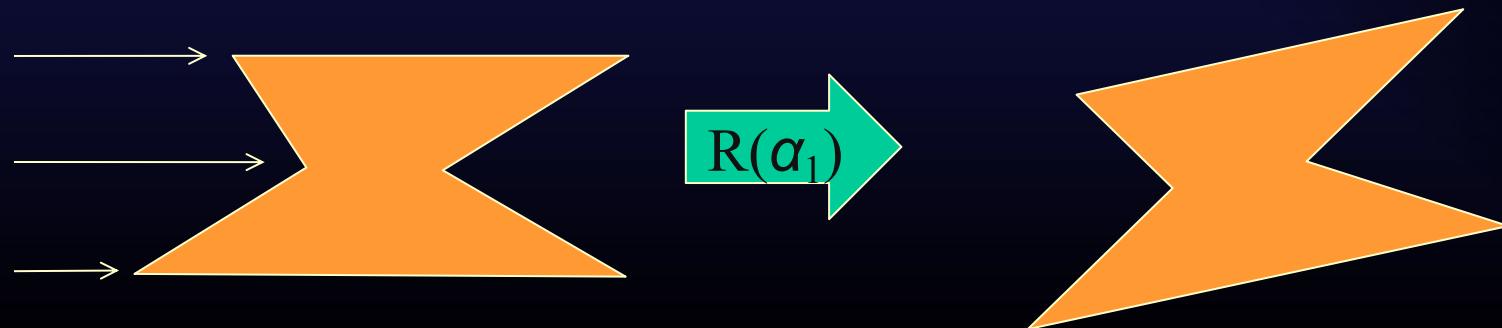
Convex Decomposition into Trapezoids

- Assume the polygon P is in the XY plane and oriented with a counterclockwise vertex ordering. (We can always project a 3D polygon into a canonical XY, XZ, or YZ plane depending on the largest normal component of the original polygon plane Z, Y, or X, respectively).
- Sort the vertices of P by increasing Y coordinate.
- → minor issue to take care of here... (next page)
- When we have the proper sort, find the bounding box B of P and increase its size by any small positive scale factor relative to its centroid so that every vertex is wholly contained in the “bounding*” box B^* .



Decomposition into Trapezoids (Issue)

- It's easier for us if no two vertices have exactly the same Y coordinate. Since we're using floats, this is actually unlikely, but we don't really want to create extremely slender "sliver" trapezoids or triangles that occur only because of floating point numerical precision.
- A randomization approach is likely to work: choose a random rotation angle α_1 and rotate the vertices of P around vertex P_0 . The chances that we will create an alignment of two other Y coordinates is unlikely but conceivable: if we're unlucky, choose a different random angle α_2 and try again (and repeat as needed).

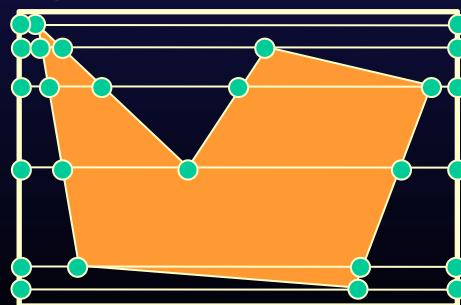


Find all Y intercepts in ascending order

Set $P_{next} = P_i$ with the smallest Y coordinate.

If P_{next} is the vertex with the largest Y, we're done, otherwise:

- Find the intersections of the line parallel to the X axis through P_{next} with all other edges of P and the X_{min} and X_{max} sides of the bounding* box B^* . (There are $k-2$ such edges in P since we don't use the two edges incident to this vertex.)
- (→There is a more efficient way to do this; it's not crucial now.)
- Sort these intersections by increasing X coordinate.
- Link these sorted X intersection lists in ascending Y coordinate order (smallest to largest).



Now loop through this data structure from smallest Y coordinate upwards

Start at the point $P(x_{left}, y_{low})$ with the smallest Y coordinate y_{low} and smallest X coordinate x_{left} ;

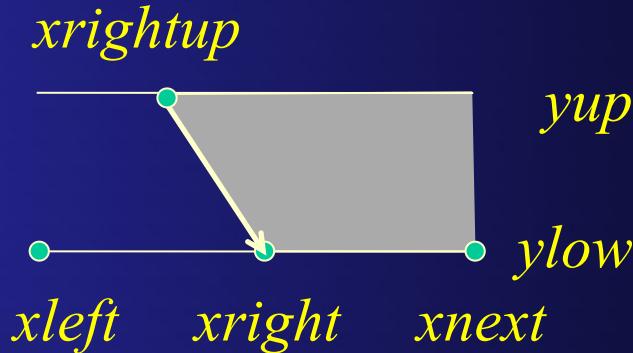
```
// next moves to the next higher Y intercept line //
while next(next((ylow)) != nil do // process line with Y=  $y_{low}$ 
     $y_{up}$  = next(ylow);
     $x_{left}$  = first (leftmost) intersection point in X list for  $y_{low}$ ;
    TrapList = nil; // TrapList holds 3 or 4 vertices of new
                      polygon //
    while next(next(xleft)) != nil do // X traverse loop //
         $x_{right}$  = next(xleft);  $x_{next}$  = next(xright);
        case  $x_{right}$ : // leaving out the code for testing the cases //
```

There are 6 cases to consider; Degenerate ones eliminated by the constraint to unique Y coordinates!

1. P has a local left edge at $xright$ on the y_{low} line.
2. P has a local right edge at $xright$ on the y_{low} line.
3. y_{low} line contains a local minimum of interior of P at $xright$.
4. y_{low} line contains a local minimum of exterior of P at $xright$.
5. y_{low} line contains a local maximum of interior of P at $xright$.
6. y_{low} line contains a local maximum of exterior of P at $xright$.

Case 1:

P has a local left edge at $xright$ on the y_{low} line



while $xnext \neq \text{nil}$ do

$xrightup$ = next point at yup on incoming edge to $xright$;

$TrapList$ = push($TrapList$, $xrightup$);

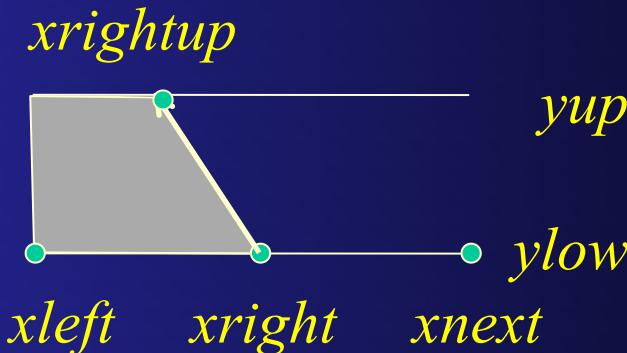
$TrapList$ = push($TrapList$, $xright$);

$xleft$ = $xright$;

end while; // continue to next X point along line y_{low} //

Case 2:

P has a local right edge at $xright$ on the y_{low} line



while $xnnext \neq \text{nil}$ do

$xrightup$ = next point at yup on outgoing edge from $xright$;

$TrapList$ = push($TrapList$, $xright$);

$TrapList$ = push($TrapList$, $xrightup$);

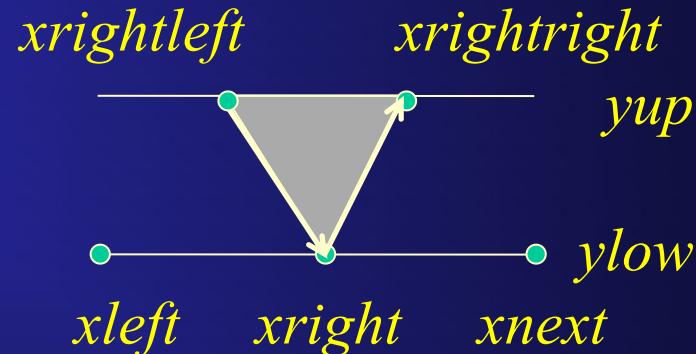
 output($TrapList$);

$TrapList$ = nil;

$xleft = xright$;

end while; // continue to next X point along line y_{low} //

Case 3:
*y*_{low} line contains a local minimum of interior of *P* at *x*_{right}



while *x*_{next} != nil do

*x*_{rightright} = next point at *y*_{up} on outgoing edge from *x*_{right};

*x*_{rightleft} = next point at *y*_{up} on incoming edge to *x*_{right};

 TrapList = push(*TrapList*, *x*_{right});

 TrapList = push(*TrapList*, *x*_{rightright});

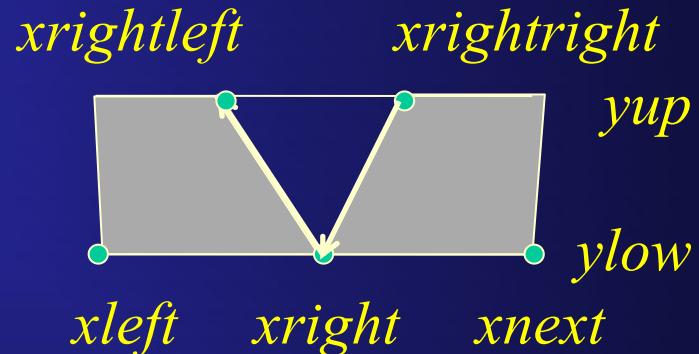
 TrapList = push(*TrapList*, *x*_{rightleft});

 output *TrapList*; *TrapList* = nil;

*x*_{left} = *x*_{right};

end while; // continue to next X point along line *y*_{low} //

Case 4:
y_{low} line contains a local minimum of exterior of P at *x_{right}*



while *x_{next}* != nil do

x_{rightright} = next point at *y_{up}* on incoming edge to *x_{right}*;

x_{rightleft} = next point at *y_{up}* on outgoing edge from *x_{right}*;

TrapList = push(*TrapList*, *x_{right}*);

TrapList = push(*TrapList*, *x_{rightleft}*);

 output *TrapList*; *TrapList* = nil;

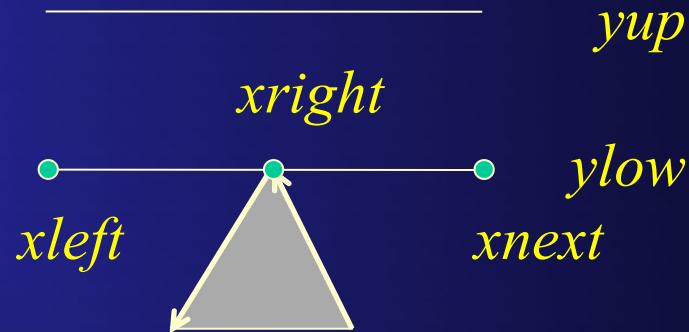
x_{left} = *x_{right}*; *x_{right}* = *x_{next}*; *x_{next}* = next(*x_{next}*);

TrapList = push(*TrapList*, *x_{rightright}*);

TrapList = push(*TrapList*, *x_{right}*);

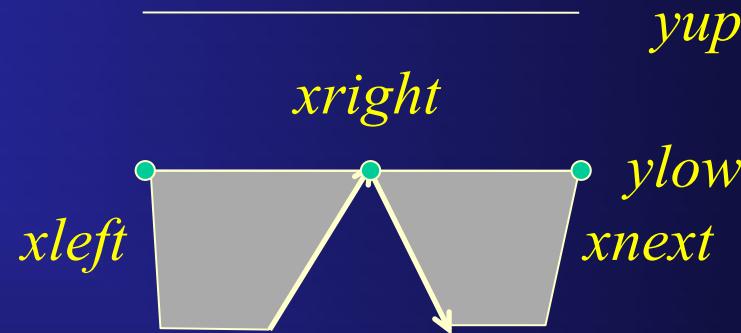
end while; // continue to next X point along line *y_{low}* //

Case 5:
*y*_{low} line contains a local maximum of interior of P at x _{right}



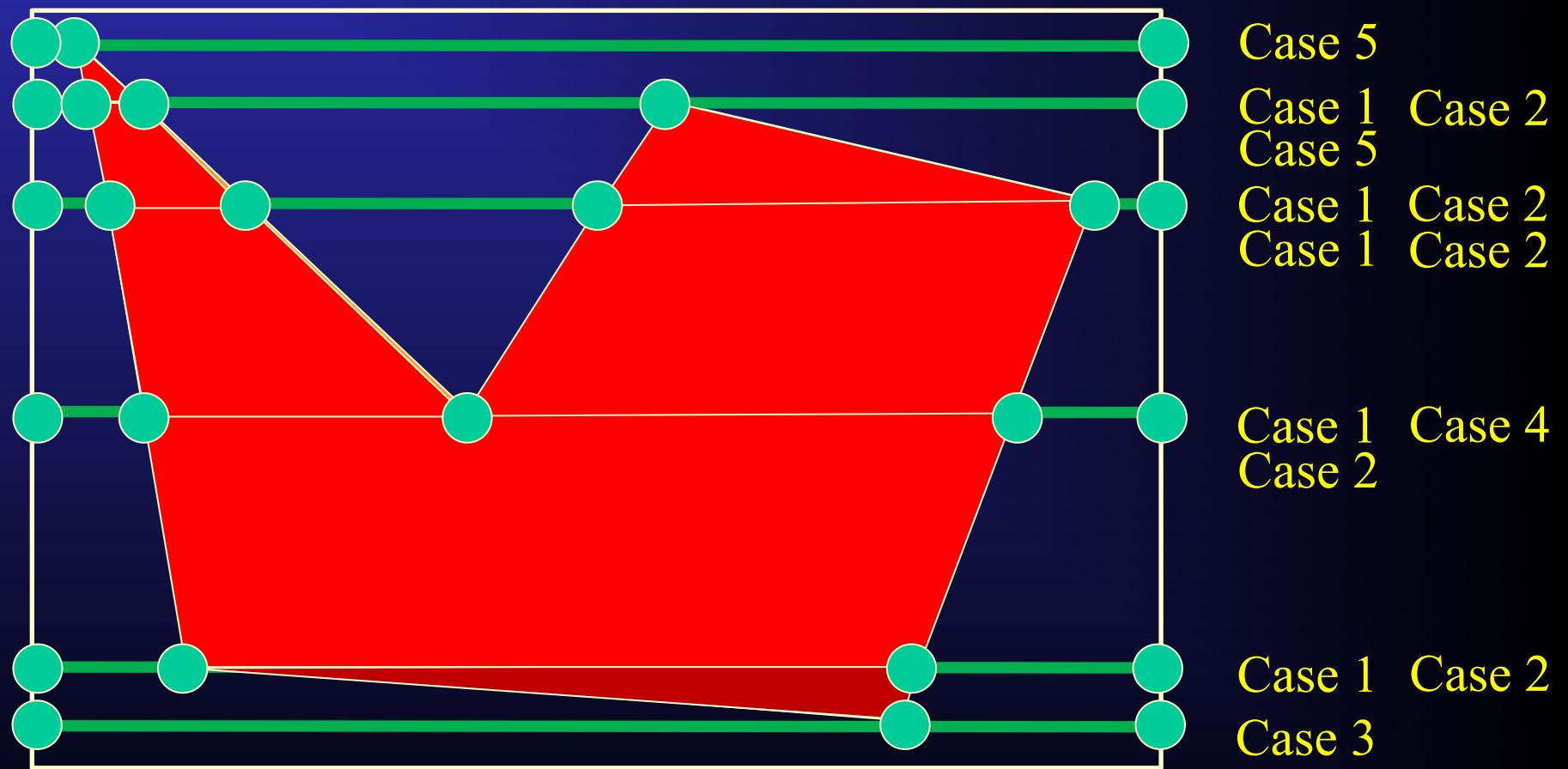
// continue to next X point along line y_{low} //

Case 6:
y_{low} line contains a local maximum of exterior of P at x_{right}

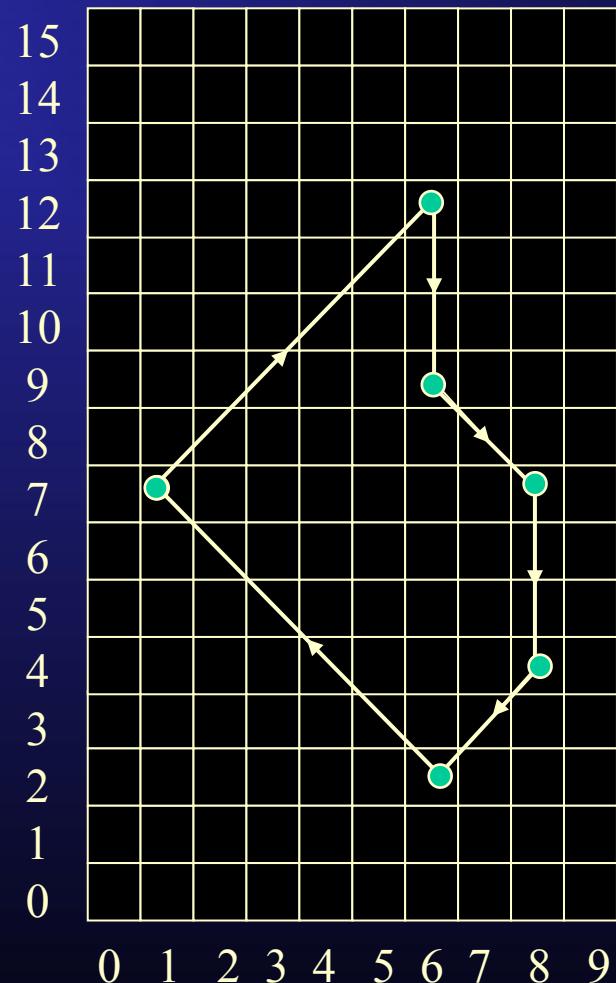


// continue to next X point along line y_{low} //

Example



Polygon Rasterization

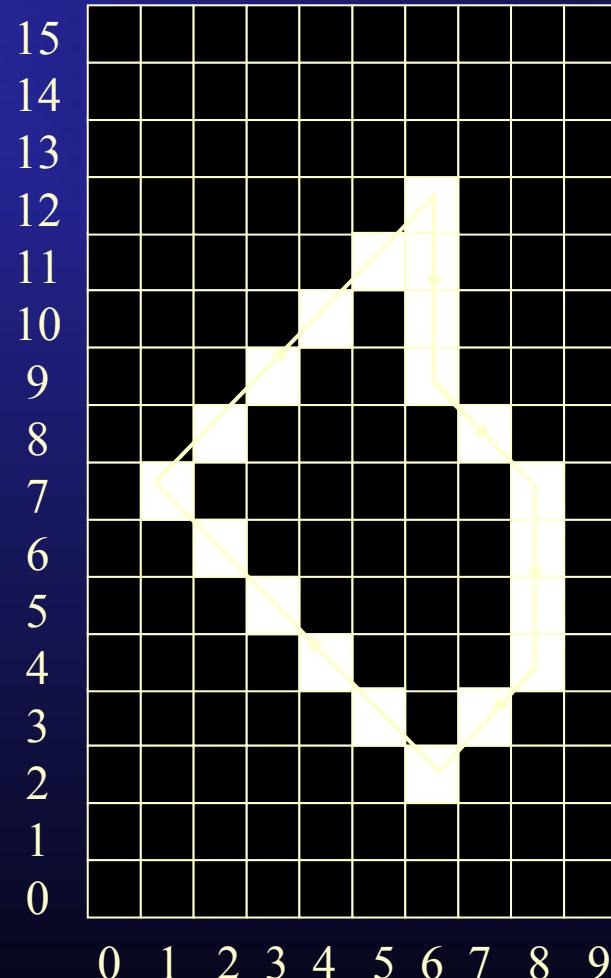


Polygon input vertices:

(1,7)
(6,12)
(6,9)
(8,7)
(8,4)
(6,2)
(1,7)

Notice that edges are given in order

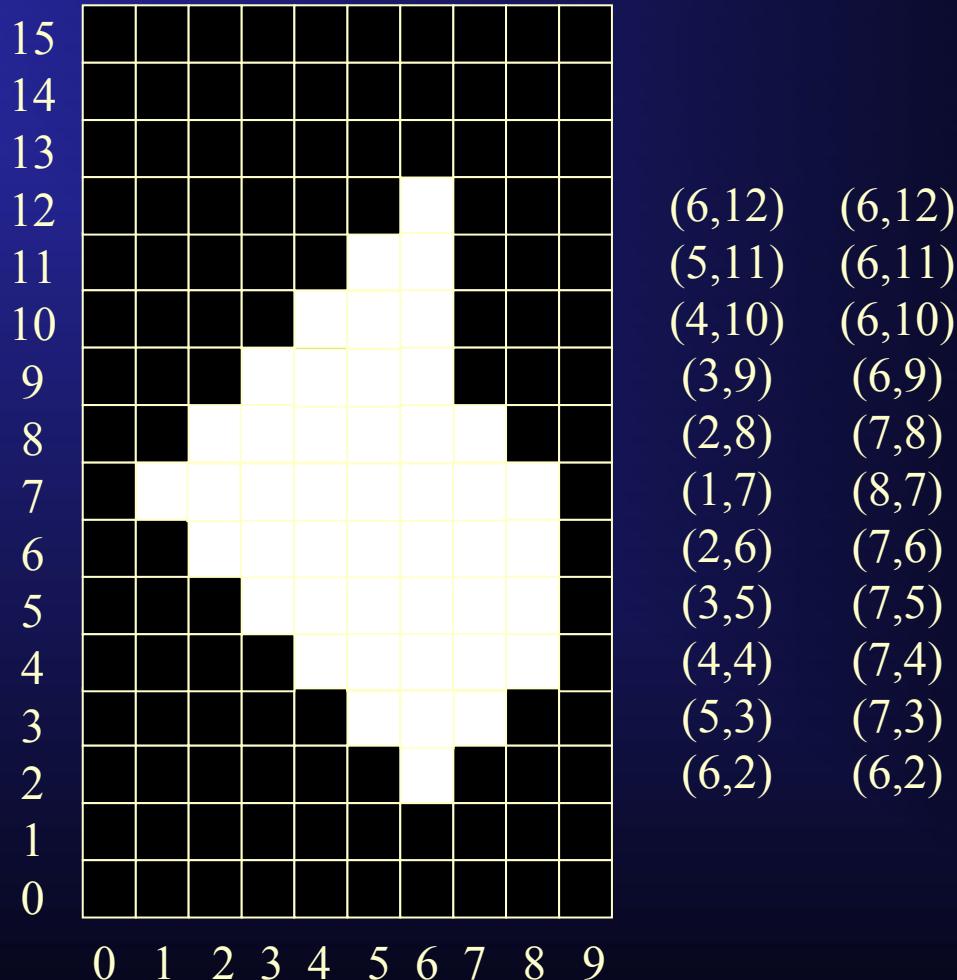
Polygon Rasterization



(6,12)	(6,12)
(5,11)	(6,11)
(4,10)	(6,10)
(3,9)	(6,9)
(2,8)	(7,8)
(1,7)	(8,7)
(2,6)	(7,6)
(3,5)	(7,5)
(4,4)	(7,4)
(5,3)	(7,3)
(6,2)	(6,2)

Compute edge intersections with scan lines
(round to nearest integer by using DDA-type algorithm)
Sort by y coordinate, then by x coordinate

Polygon Rasterization



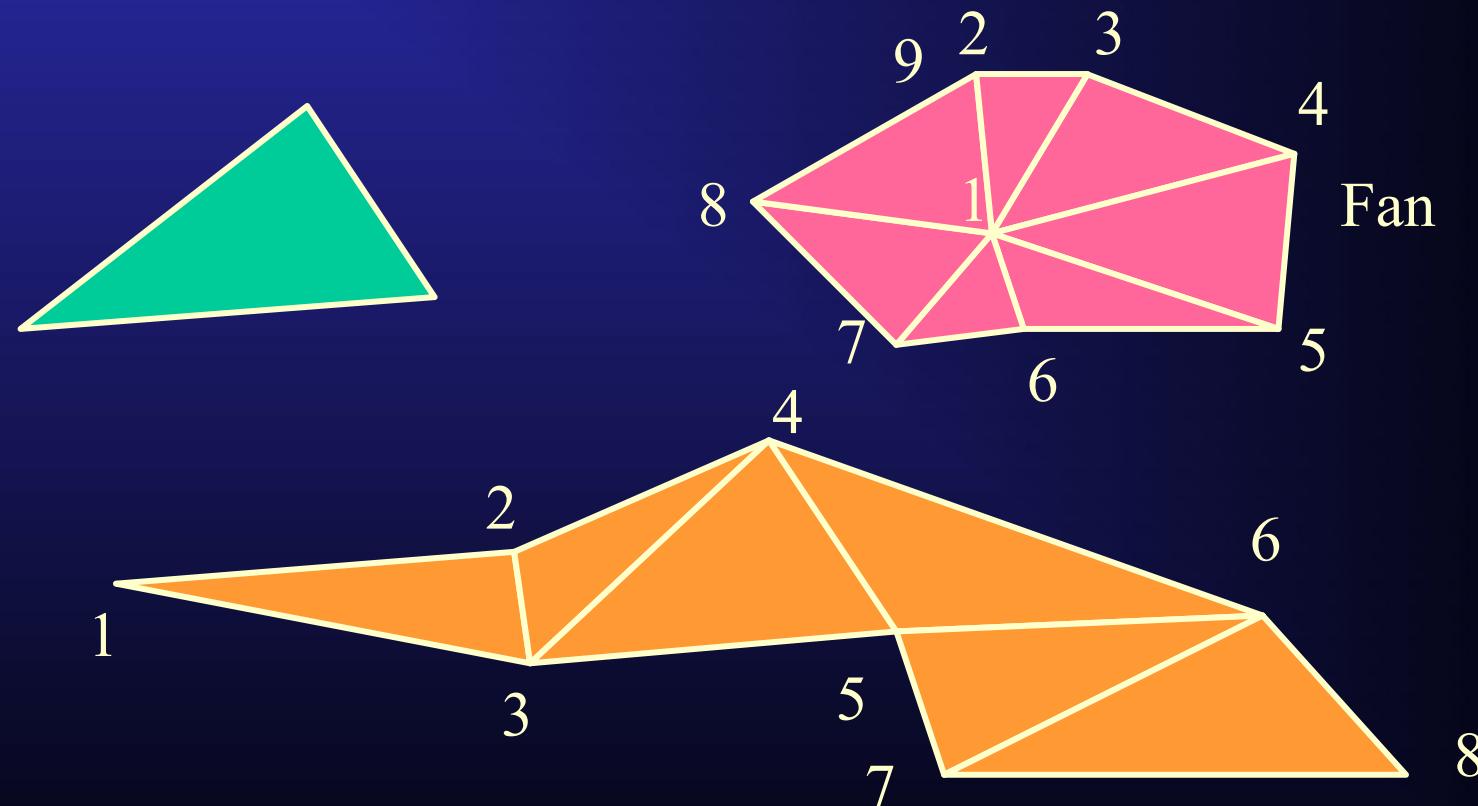
Notice that the sort leaves intersection points in y coordinate pairs:
so just fill in pixels between each pair of intersections!

Triangles are Nice

- No really nasty degenerate cases for triangle rasterization.
- Split Quads.
- Rasterizer on GPU creates per-pixel fragments.
- But lots of triangles means lots of vertex data: worst case $3n$ vertices for n triangles.
- We can do better...

Discrete Triangles, Fans, and Strips

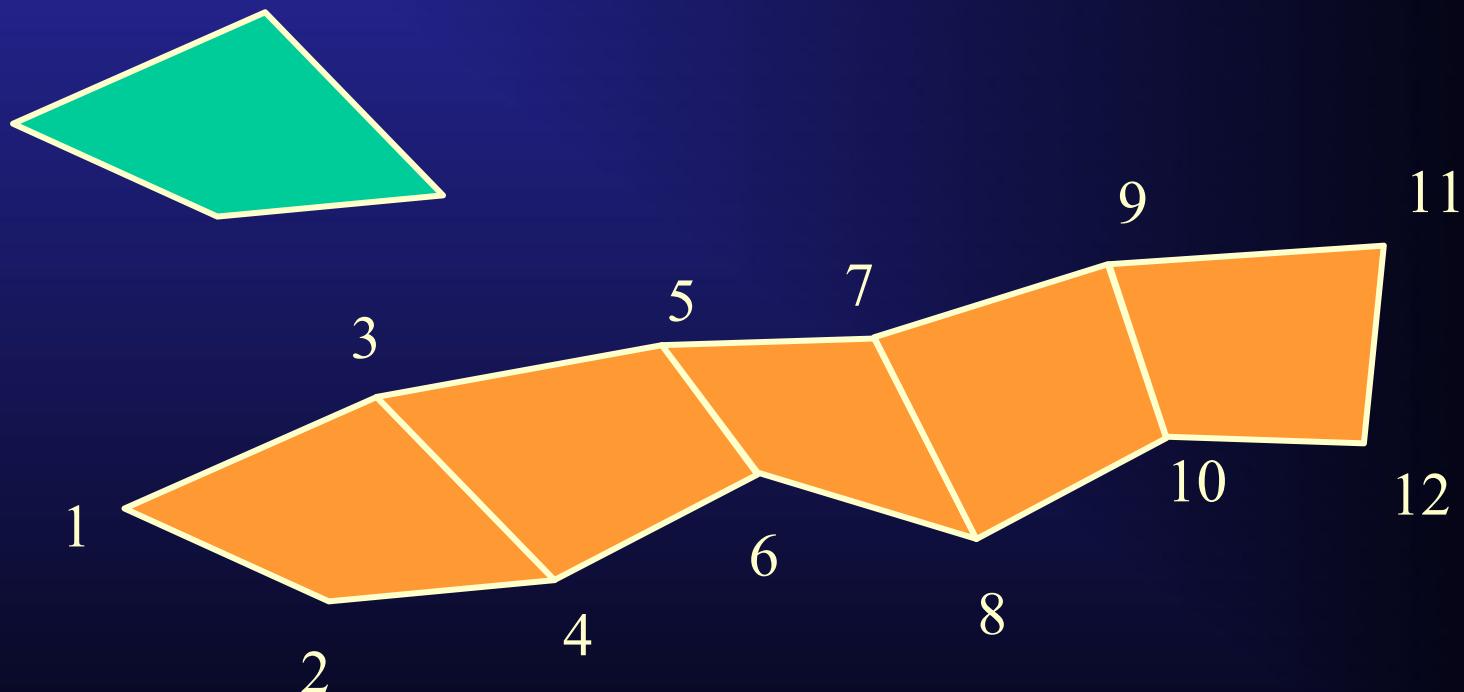
- Used to minimize data sent to graphics processor...



Strip: each new vertex builds two edges off previous two vertices:
 n triangles require only $n+2$ vertices, not $3n$.

Discrete Quads and Quad Strips

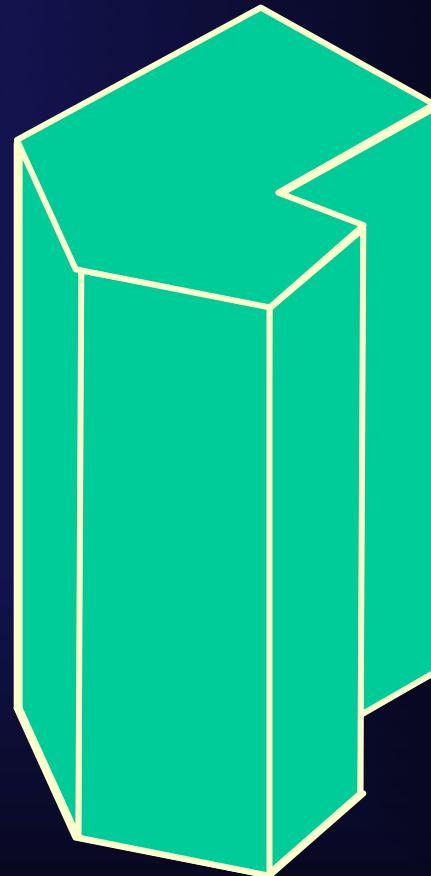
- Used to minimize data sent to GPU.



Each new vertex pair builds three edges off previous edge:
 n quads require only $2n+2$ vertices, not $4n$.

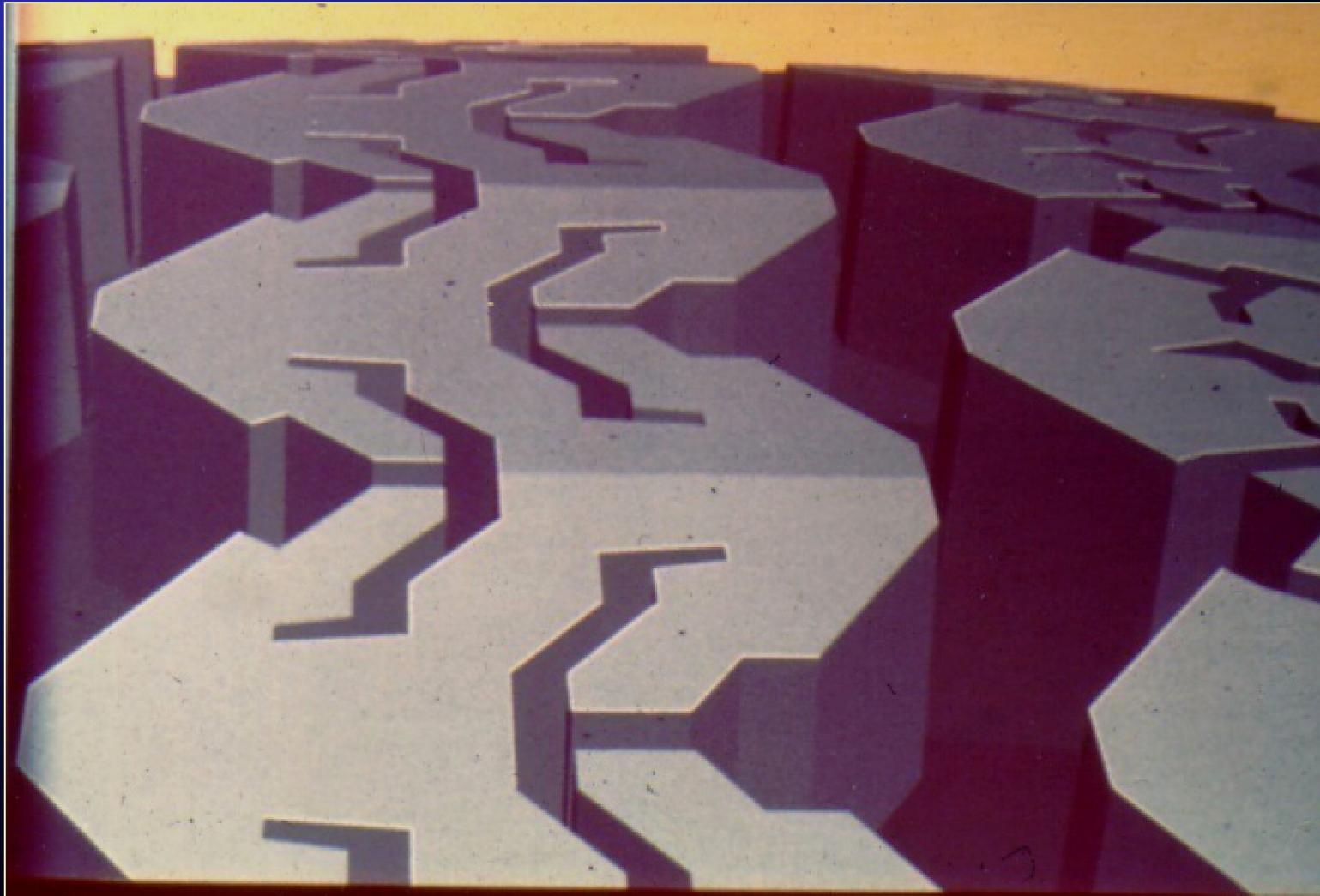
Polygon Object Generation: Extrusion

- Extrusion along a line segment as a 3D prism.



SketchUp

Tire Tread (or is it?)

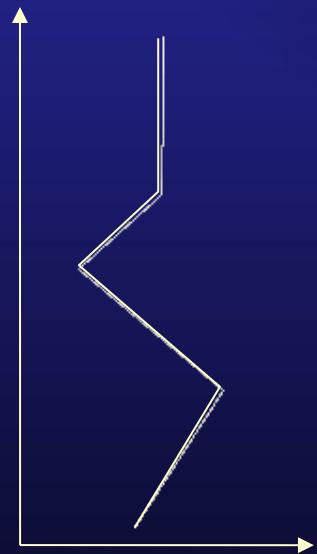


© 1982 DIGITAL EFFECTS

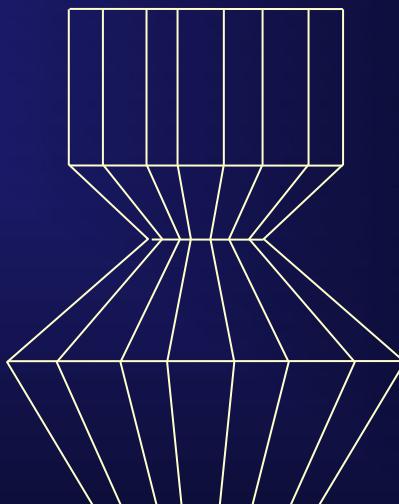
648

Polygon Object Generation: Surface of Rotation

- Surface of rotation (around line axis).



Profile curve



Rotated at discrete angular intervals, making polygons

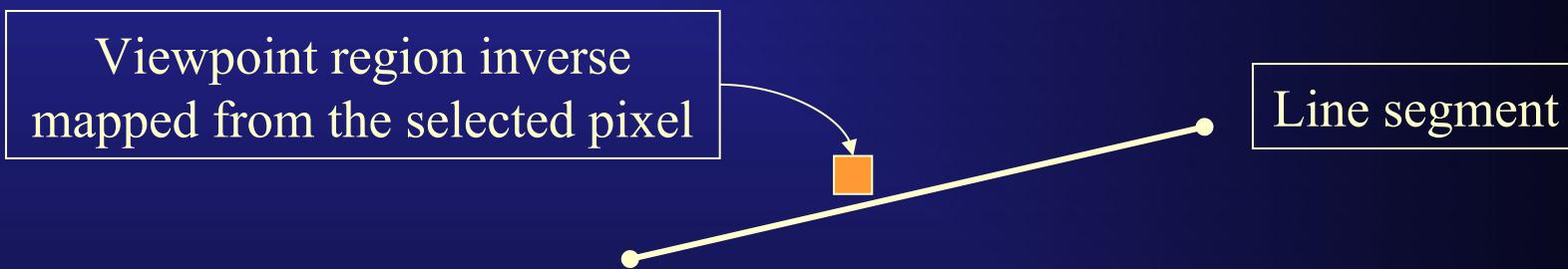
Point on Line?

- In interactive applications we may need to check whether some given point is actually within or on a given line segment.
- This is complicated by the discrepancy between the finite *integer* coordinate system of the graphics display (e.g., its resolution in pixels) and the *floating point* representation of numbers representing coordinates.
- We saw this inverse viewing transformation earlier.

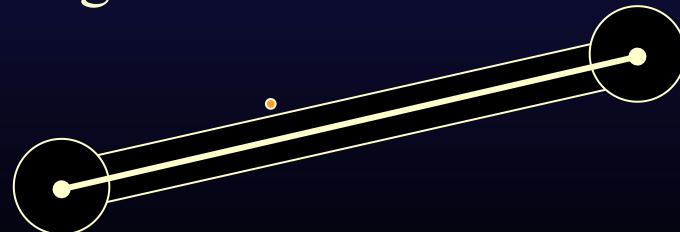


Back to Point on Line...

- So just checking if a point is *mathematically* on a line segment is not going to work:



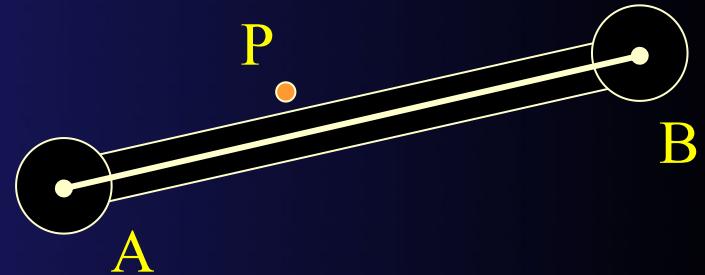
- So what we do is put a “gravity field” around the line segment and test it against the *point* at the center of the inverse mapped region.



Point on Line Algorithm

- Line A to B; test point P.
- Need two parameters:
 - Radius at an endpoint: *rad*
 - Width along line segment: *width*
- *Rad* and *width* need not be the same, and should be chosen based on the window to viewport size ratio, but we'll just make them constants for now, with *rad* > *width*.
- Algorithm:

If (P is inside disk of radius *rad* at A) ||
(P is inside disk of radius *rad* at B) ||
(P is inside rectangle of width *width* centered along
line segment AB)
then inside=true else inside=false



The Inside Checks

- Inside disk: $\|\mathbf{P} - \mathbf{A}\| < \text{rad}$ (and likewise $\|\mathbf{P} - \mathbf{B}\| < \text{rad}$)
- What about point in rectangle? This rectangle may not be aligned with the coordinate axes. The key is to find the distance between the point \mathbf{P} and the line segment \mathbf{AB} represented parametrically by solving for t :

$$(\mathbf{P} - \mathbf{AB}(t)) \bullet \mathbf{AB} = 0$$

- This equation says that the closest distance to line segment \mathbf{AB} is where the vector from \mathbf{P} to the parametric point on \mathbf{AB} is perpendicular to \mathbf{AB} , as long as $t \in [0,1]$.

Solving the Distance Equation

$(P - AB(t)) \bullet AB = 0$; rewriting by definition : $(P - (Bt + A(1-t))) \bullet AB = 0$
separating into x and y terms and taking the dot product :

$$(P_x - (B_x t + A_x(1-t)))(B_x - A_x) \\ + (P_y - (B_y t + A_y(1-t)))(B_y - A_y) = 0$$

now multiply everything out and collect terms to solve for t :

$$t = \frac{(A_x - P_x)(A_x - B_x) + (A_y - P_y)(A_y - B_y)}{(A_x - B_x)^2 + (A_y - B_y)^2}$$

Check that this makes sense :

if $A = B$ then the denominator is 0 (AB is not a line). ✓

if $P = A$ then numerator is 0 so $t = 0$. ✓

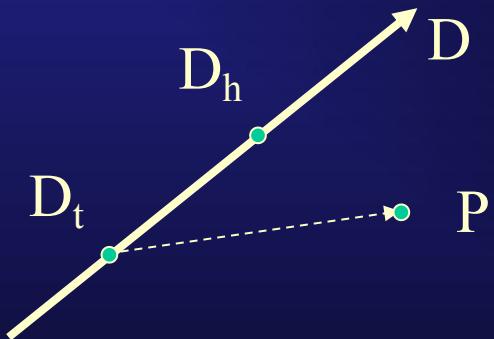
if $P = B$ then numerator = denominator, so $t = 1$. ✓

So compute t , check $t \in [0,1]$, and if true, then substitute

t in parametric form for AB , and finally compute the distance
between P and $AB(t)$ by the usual distance formula.

Point in Half-Plane?

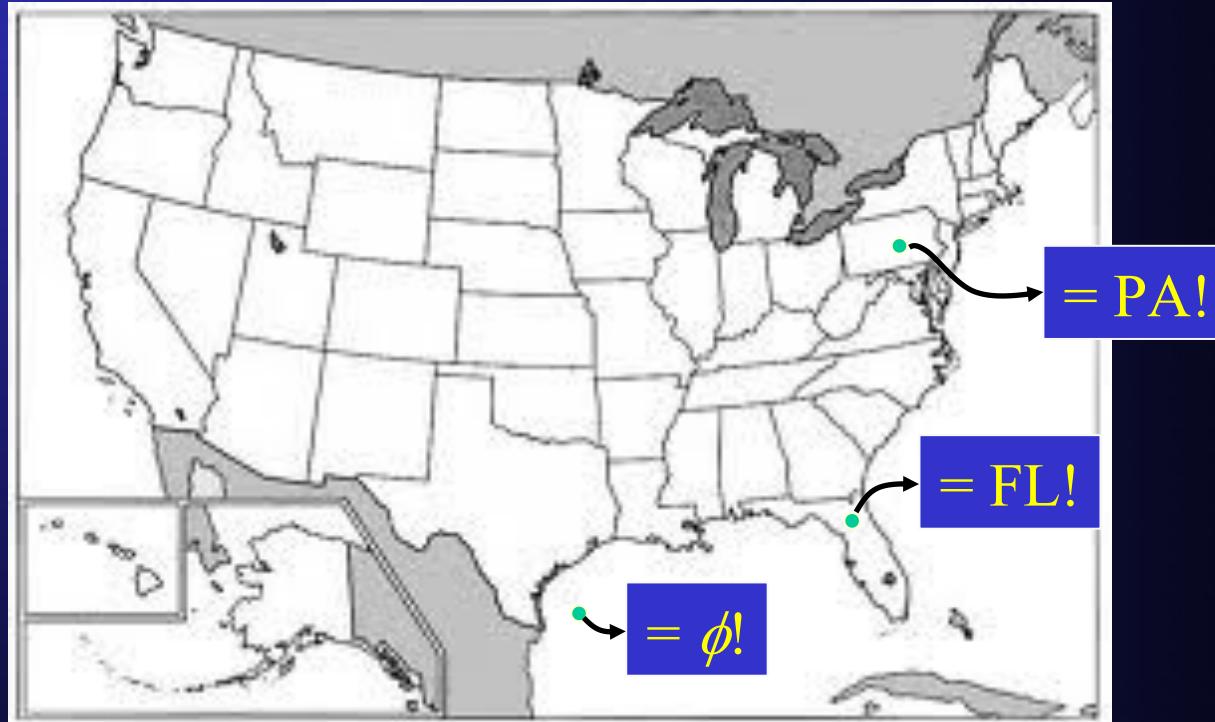
- Let D be a directed line in the 2D x - y plane and P a point in the x - y plane.
- It makes sense to ask which half-plane the point P lies in relative to the directed line D : to its left or right.



- Compute easily via cross product and right-hand rule:

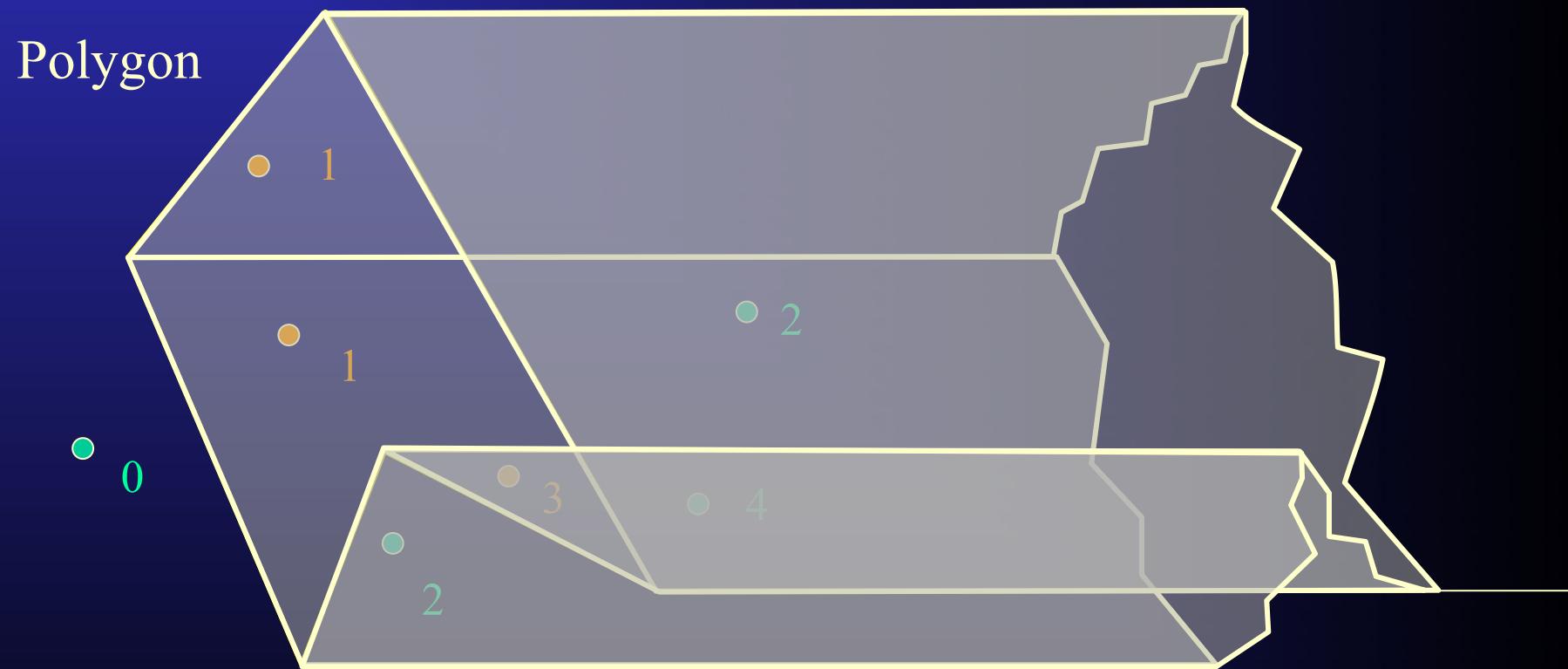
$$((P - D_t) \times (D_h - D_t))_z = \begin{cases} > 0 & \text{then } P \text{ is to right of } D \\ = 0 & \text{then } P \text{ is on } D \\ < 0 & \text{then } P \text{ is to left of } D \end{cases}$$

Point in Polygonal Shape?



- For example, we'd like to select a state by clicking *inside* it, but there's “nothing” there to pick.
- Need to test for point inside polygon in order to select a particular state from the map.

Determining if a Point is Inside a Polygon via the Projection Method



Count number of “strips” point is inside.

Even \Rightarrow outside

Odd \Rightarrow inside

$O(n)$ for n sides.

Point Inside Polygon Algorithm

Adapted from Shimrat: Collected Algorithms of ACM, #112, Aug. 1962, p.434

Input: polygon and test point

Output: PointInPoly → true if point is inside polygon, otherwise false

Note: this algorithm is possibly ambiguous if the point is on or almost on an edge of the polygon. Accordingly, it is best to test if the point lies on any polygon edge first before running the inside test, if the results need to be accurate.

Also, if the polygon contains horizontal line segments and the point lies on such a horizontal segment, the answer may vary.

These two cases are fixable, but are left as an exercise.

```

bool PointInPoly(const gMatrix& polygon, const gVector& point){
    bool inside = false;
    gVector p1;
    gVector p2;
    for (i = 0; i<polygon.cols()-1; i++){ // iterate through each edge
        p1 = polygon.getCol(i); //get first edge endpoint from polygon
        p2 = polygon.getCol(i+1); // get second edge endpoint from polygon
        if (p1[1] > p2[1]) swap(p1, p2); //want p1 to p2 to point in +y direction
        if (point[1] > p1[1]){ //above lower edge endpoint
            if (point[1] <= p2[1]){ //below upper edge endpoint
                if (p1[1] != p2[1]){ //if edge is not horizontal do half-plane check
                    // z of crossproduct(point-p1, p2-p1) > 0) means we're to the right
                    if ((point[0] - p1[0]) * (p2[1] - p1[1])
                        - ((p2[0] - p1[0]) * (point[1] - p1[1]))) > 0) inside = !inside
                    // if true, point is to right of edge (and thus in the “strip”)
                }
            }
        }
    }
    return inside;
}

```

Computing the Axis-Aligned Bounding Box for Polygon Vertices

Let AABB be the 2x2 matrix: $\text{AABB} = \begin{bmatrix} x_{min} & x_{max} \\ y_{min} & y_{max} \end{bmatrix}$

To construct the AABB values:

```
bool SetBoundingBox(const gMatrix& V, const gMatrix& AABB){  
    int i;  
    AABB[0][0] = AABB[0][1] = V[0][0];  
    AABB[1][0] = AABB[1][1] = V[1][0];  
    for (i = 1; i = V.cols()-1; i++) {  
        if (AABB[0][0] > V[0][i]) AABB[0][0] = V[0][i];  
        if (AABB[0][1] < V[0][i]) AABB[0][1] = V[0][i];  
        if (AABB[1][0] > V[1][i]) AABB[1][0] = V[1][i];  
        if (AABB[1][1] < V[1][i]) AABB[1][1] = V[1][i]}  
}
```

Combining Algorithms for Greater Efficiency

- If point is outside polygon bounding box, then point is outside polygon; Otherwise do PointInPoly test.



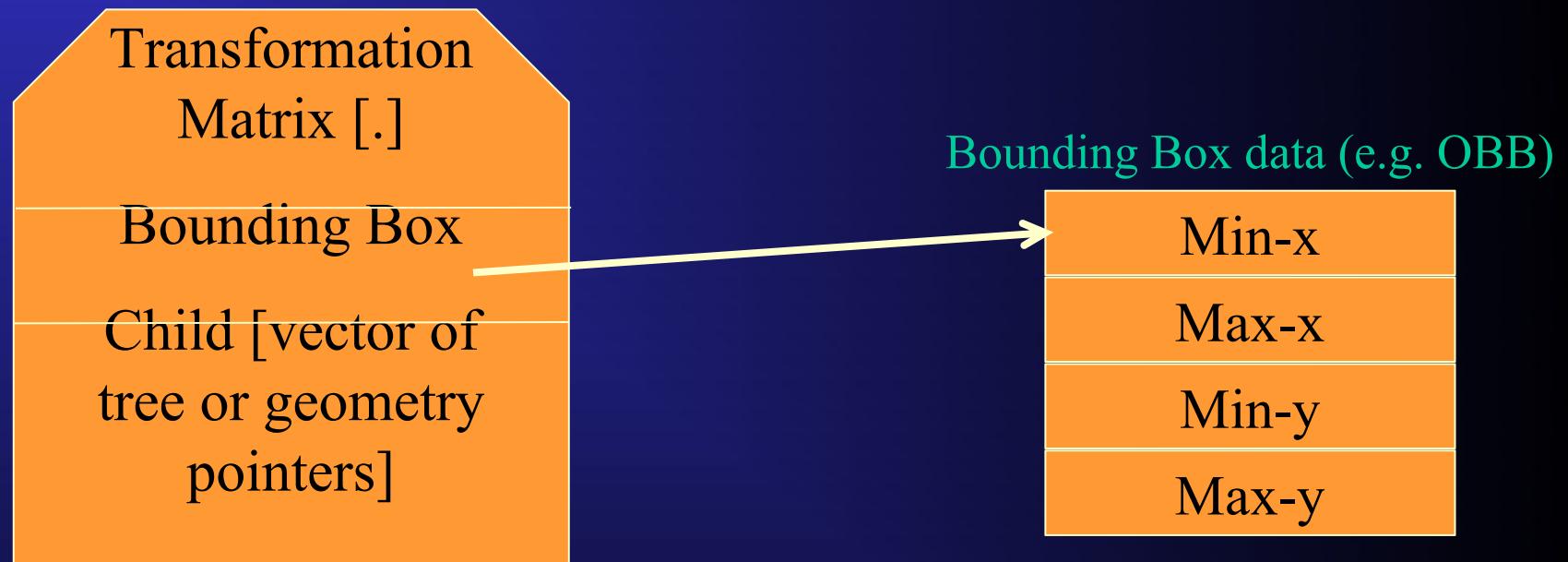
- More elaborate combination algorithms are possible.

Bounding Boxes Used for Test Efficiency

```
bool PointInPolyWithBoundingBoxCull(const gMatrix&
polygon, const gMatrix& BoundingBox, const gVector&
point ){
    if ((point[0] <= BoundingBox(xmin)) ||
        (point[0] >= BoundingBox(xmax)) ||
        (point[1] <= BoundingBox(ymin)) ||
        (point[1] >= BoundingBox(ymax)))
        return false
    else return PointInPoly(polygon, point)
}
```

Note that this is not a very good implementation in the sense that we have only *one* Bounding Box array; in general we'll need to associate a bounding box with a polygon or object model. A bounding box array can be stored at each node of the scene graph.

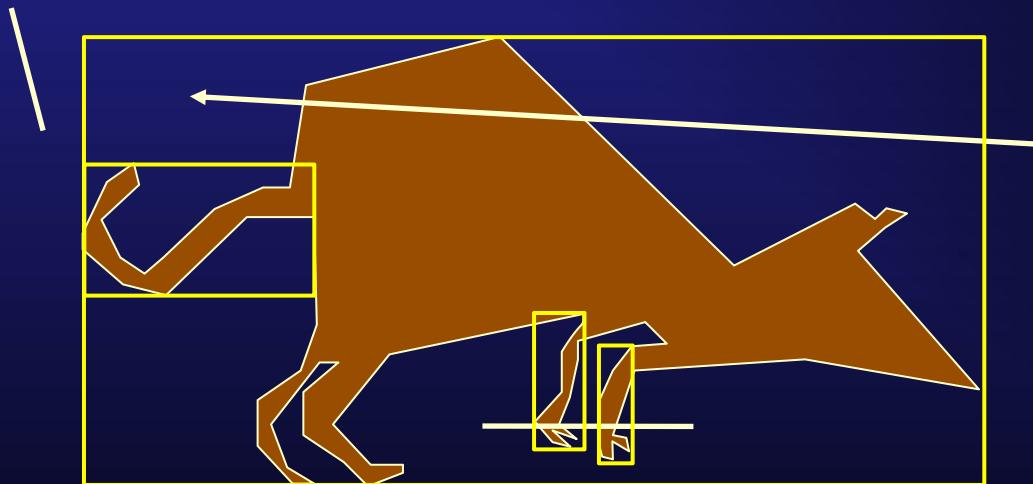
Extend Scene Graph Node with Bounding Boxes



- Alternative bounding box representations may be stored.
- Bounding box must be re-evaluated after transformation is applied to child geometry.
- Can use “dirty-bit” to avoid recomputing all lower BBs if a middle scene graph transformation is changed but intersection tests are not yet needed.

Line – Polygon Intersection

- For each polygon edge, compute intersection of line with edge and test if intersection is “real”.
- If polygon is “complicated”, can use (hierarchical) bounding boxes to advantage in “busy” (edge-rich) regions of its boundary.



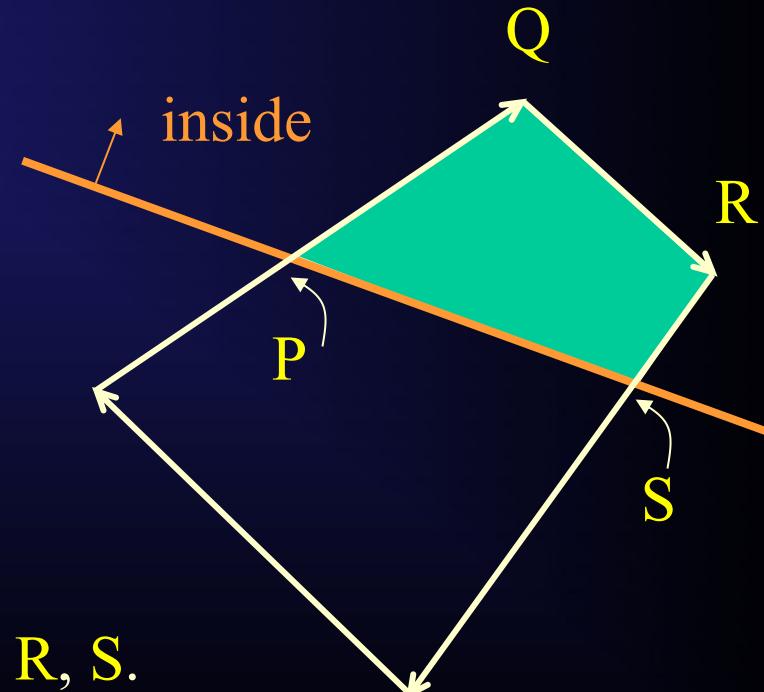
- Note that if the line is *directed* (a *ray*, for example), we may need to find the *first* intersection with the polygon. Think “projectile”.

Polygon Clipping

- *Clipping* means removing parts of an graphical object that lie outside a visible region.
- We already saw clipping a line against a rectangular region, such as the screen, a rectangle, or a rectangular viewpoint (“window” on the display screen).
- Now let’s clip a polygon against a line (creating a possibly new polygon).
- Then we’ll clip a polygon against several lines.

Clipping a Polygon to a Half-Plane

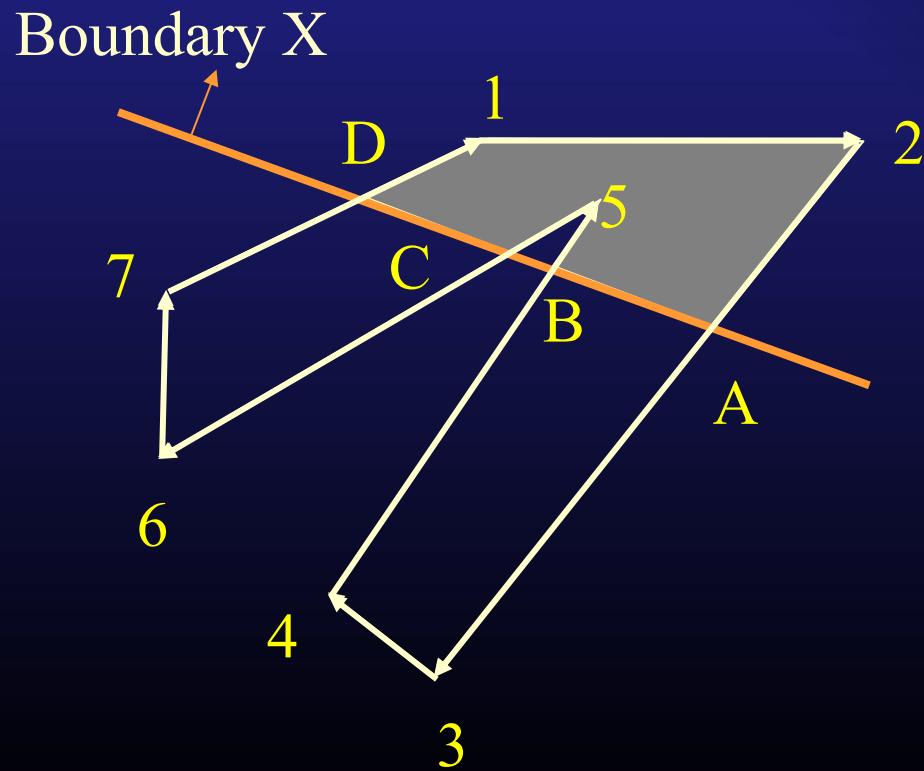
- Consider each polygon edge in turn [$O(n)$].
- ‘Output’ means add point to new polygon vertex list.
- Maintaining correct vertex order is crucial.
- Four cases (this is a simple Finite State Machine):
 - ENTER: Output P, Q
 - STAY IN: Output R
 - LEAVE: Output S
 - STAY OUT: (no output)



Therefore clipped polygon is P, Q, R, S.

Polygon Clipping Example

- Works for more complex shapes.

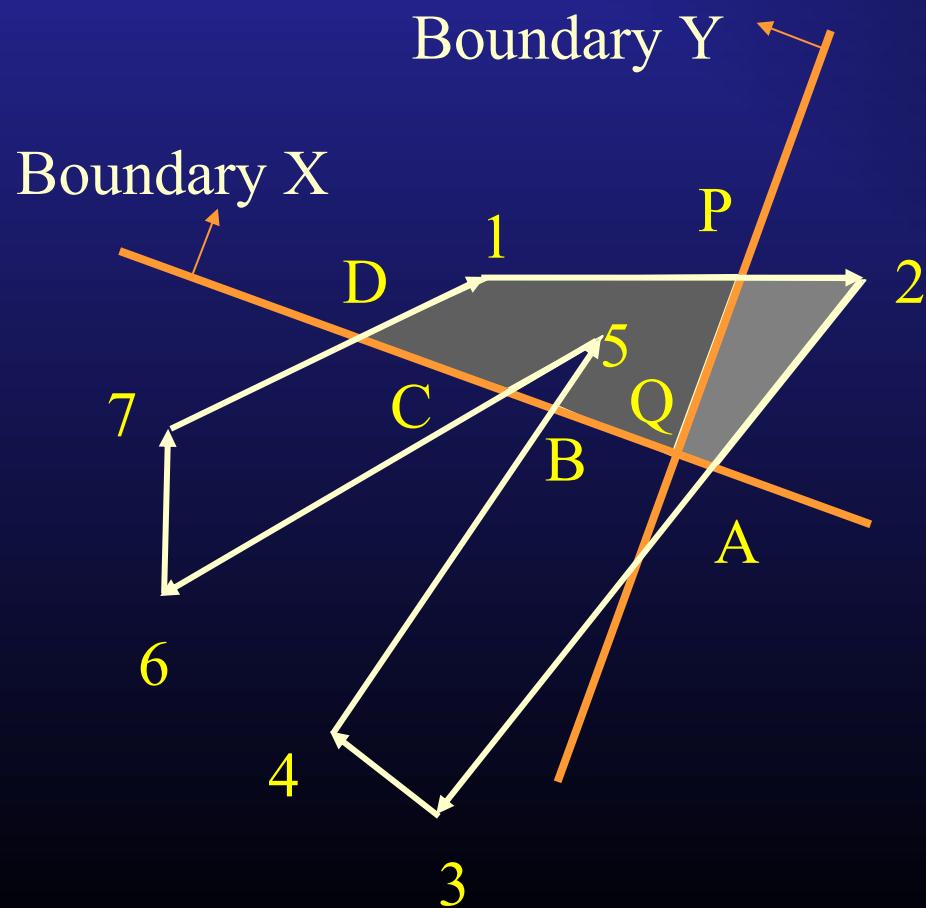


Input	Case	Output
1	start	-
2	stay in	2
3	leave	A
4	stay out	-
5	enter	B, 5
6	leave	C
7	stay out	-
1	enter	D, 1

2 A B 5 C D 1

Clipping a Polygon Against Multiple Half-Planes

- Can clip [resulting] polygon against each boundary in turn.



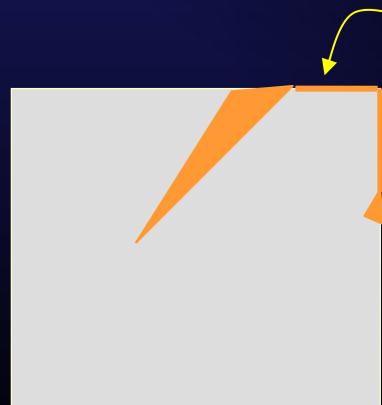
2 A B 5 C D 1

Input	Case	Output
2	start	-
A	stay out	-
B	enter	Q, B
5	stay in	5
C	stay in	C
D	stay in	D
1	stay in	1
2	leave	P

Q B 5 C D 1 P

In Practice...

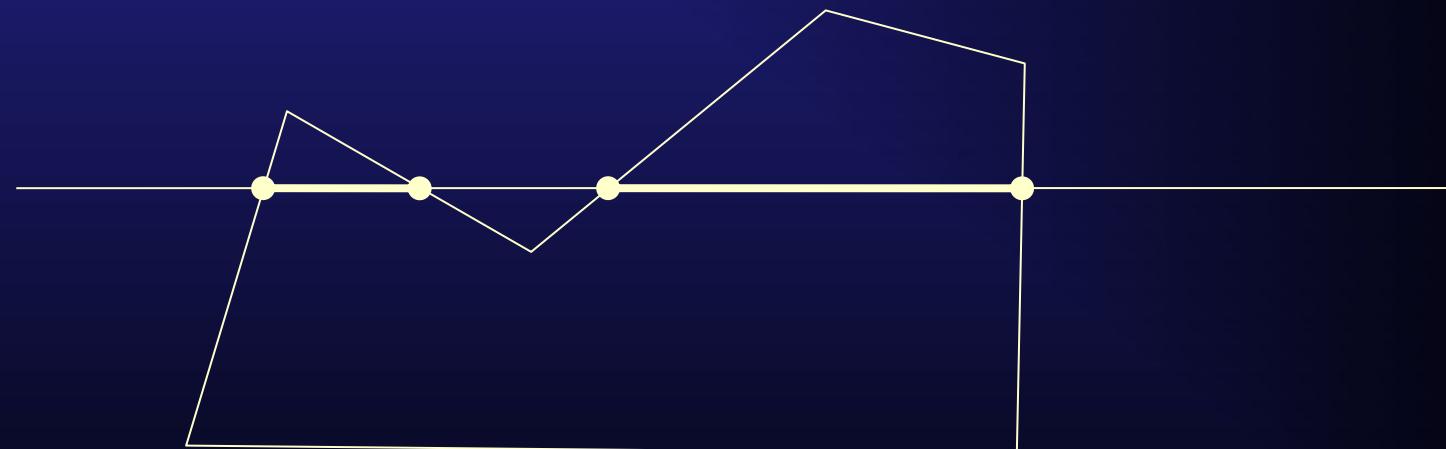
- For a polygon with N vertices, the polygon vertices are processed about $6 \times N$ times in 3D clipping.
- Clever code processes each vertex against all the clipping planes, so polygon vertices are traversed only once.
- Multiple components can occur -- and are OK, but...
- Degenerate results can occur at “turning points” (corners) and be nasty unless considerable care is taken:



0-thickness
polygon --
these edges
should not
appear.

2D Line - Polygon Intersection

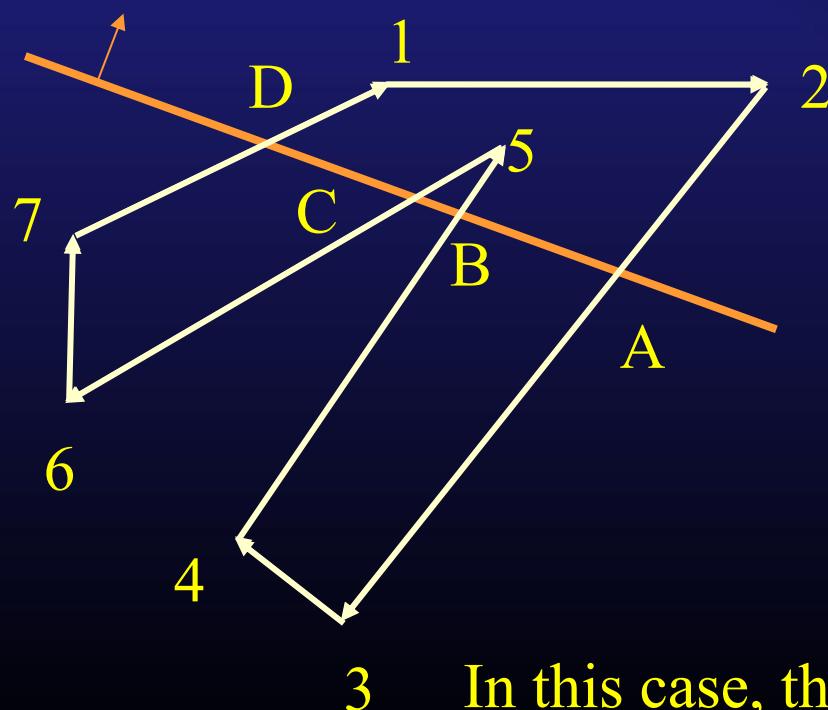
- An important algorithm is intersecting a line with a polygon to find what, if any, finite segments of the line lie inside the polygon.



- Note that we can apply the polygon clipping algorithm using this line and just see what NEW segments are added...

Finding the Intersecting Line Segments

- Simplest case: Assume that the line does not pass through any vertex of the polygon. In that case we just keep the segments that have both endpoints that are NEW:



Input	Case	Output
1	start	-
2	stay in	2
3	leave	A
4	stay out	-
5	enter	B, 5
6	leave	C
7	stay out	-
1	enter	D, 1

In this case, that's line segments AB and CD.

But the Situation is a Bit More Complicated

- The assumption that the line does not pass through a polygon vertex is much too strong. Either we have to remove this restriction or add code to handle it.
- Approach 1: (Hack) Since vertex coordinates are Floats, it is actually rather unlikely that a line will exactly hit a vertex. But if it does, then add a tiny number (a small displacement) to the vertex so that it will not intersect any more. This is ugly and dangerous, since it means we're playing with the shape of the polygon – something we shouldn't do without the user's (or application's) permission. ☹
- Approach 2: We saw that decomposition into trapezoids for rasterization was possible; in the current case we can't move the polygon, so we have to handle it!

First Observation

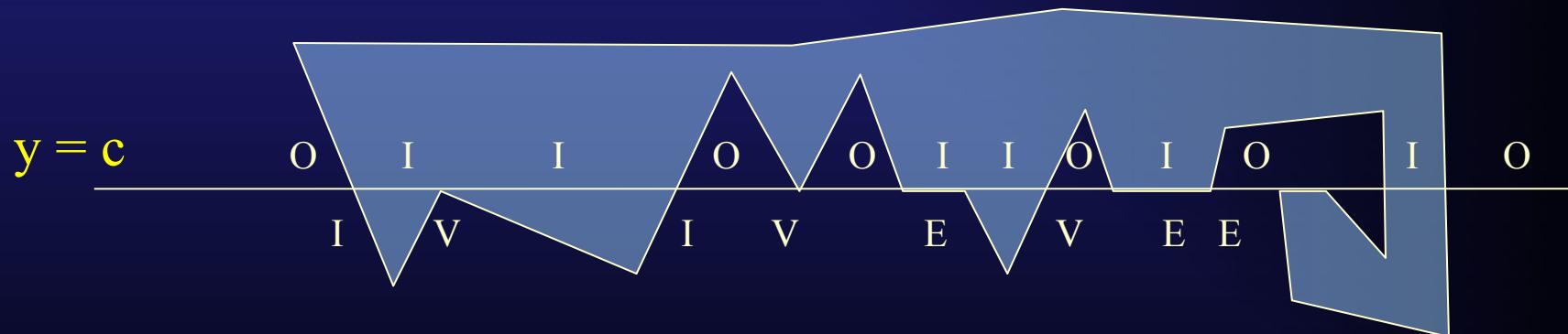
- Without loss of generality, let's assume the polygon is in the XY plane and the line intersecting the polygon is horizontal, i.e. $y = c$ for some constant c .
- The line intersects the polygon in alternating outside (O) – inside (I) – outside (O) ... segments:



- So “just” find intersections and read segments off in pairs.
- Nice idea, but life is full of degeneracies; e.g., if we move the line down a bit...

The Better Approach

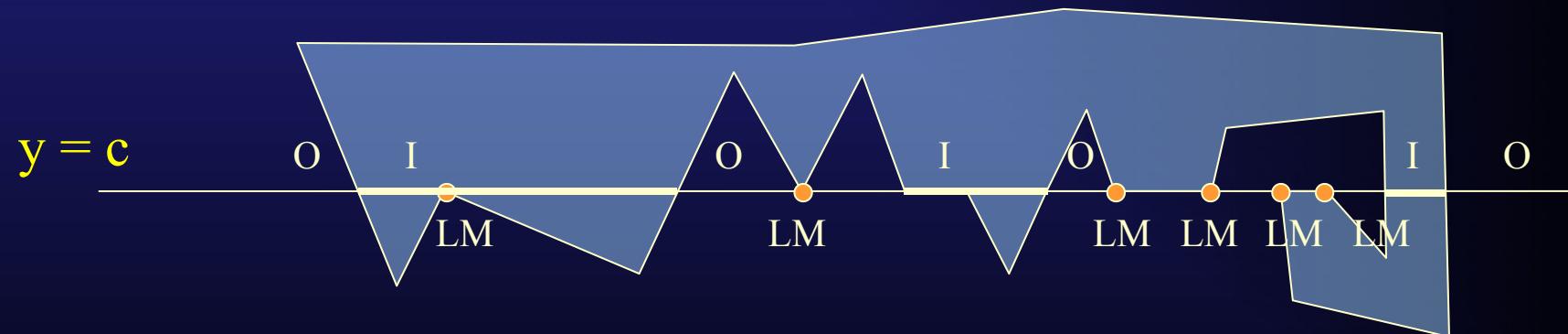
- Now we're getting all sorts of intersections that do not alternate.
- Intersect inside-edge (I); Intersect vertex (V); Intersect edge (E).



- To make sense of this, we need to use *context*.

Segments are Inside or Outside Depending on Local Maxima or Minima

- Check whether the intersection occurs at a LOCAL MINIMUM or LOCAL MAXIMUM **y** value (LM):
 - The “Enter/Exit” Finite State Machine (FSM) needs to keep track of whether successive polygon edges actually cross the line or not.

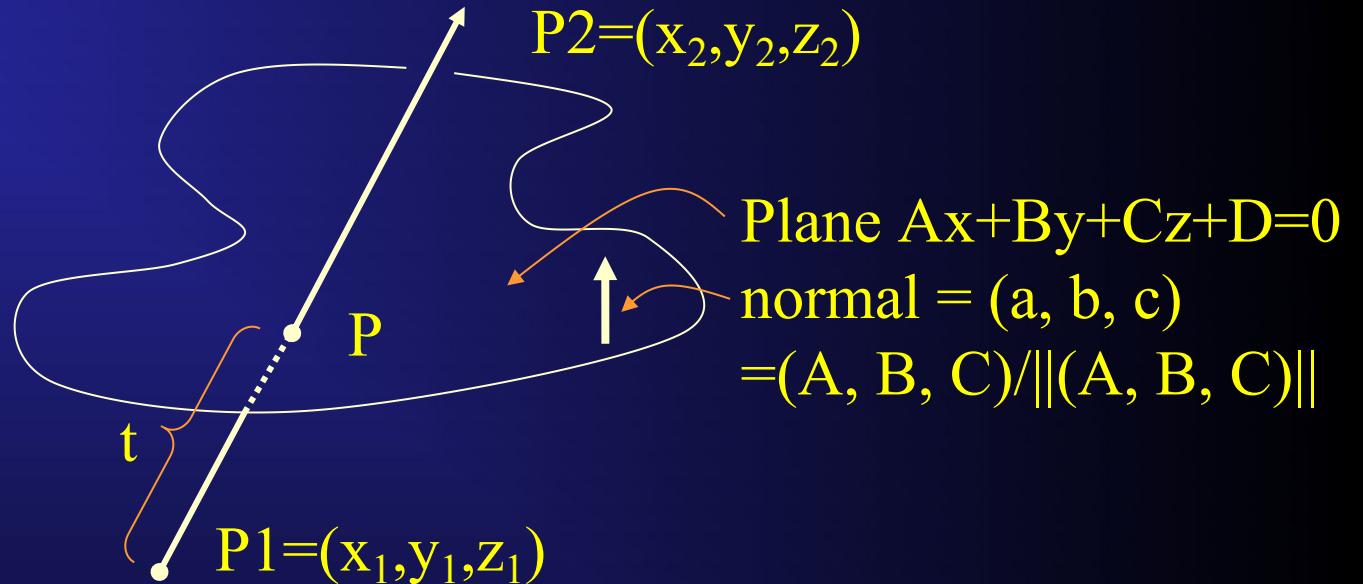


- Add more states to the FSM to handle these cases.

Geometry Intersection Algorithms

- Line (ray) with plane intersection
- Line (ray) with triangle intersection
- Line (ray) with polygon intersection
- Line (ray) with sphere intersection
- Line (ray) with cylinder intersection

Intersection of Ray with Arbitrary Plane



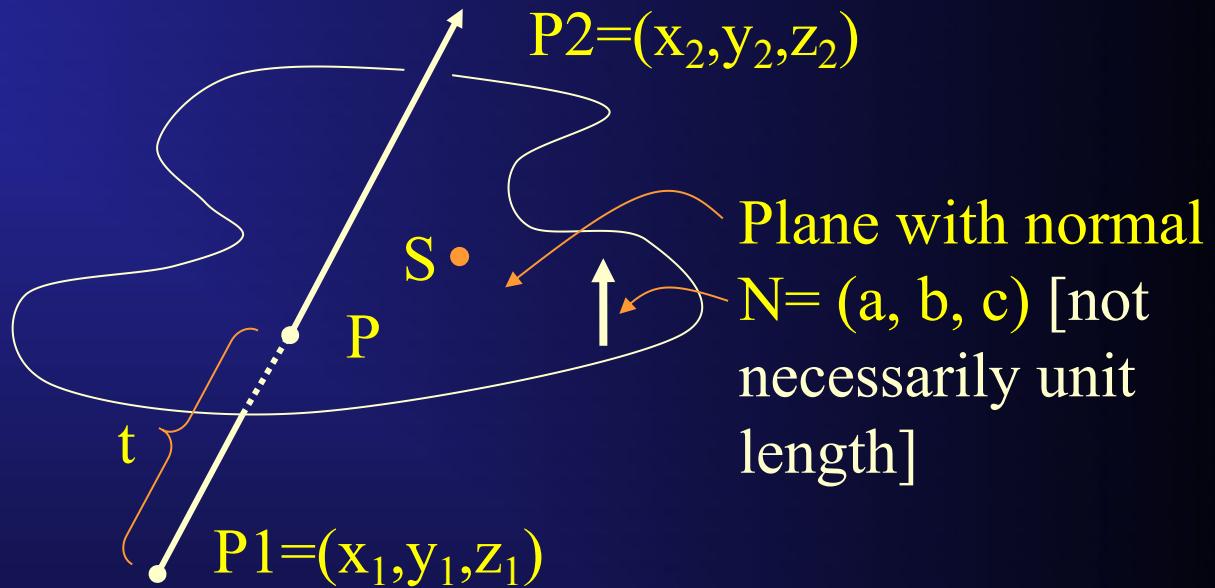
$$P = P_1 + t(P_2 - P_1) \quad \text{from parametric form: want } P, \text{ thus need } t:$$

$$A(x_1 + t(x_2 - x_1)) + B(y_1 + t(y_2 - y_1)) + C(z_1 + t(z_2 - z_1)) + D = 0$$

Solving:

$$t = (Ax_1 + By_1 + Cz_1 + D) / (A(x_1 - x_2) + B(y_1 - y_2) + C(z_1 - z_2))$$

Alternate Form: Intersection of Ray with Plane



$N \cdot (P - S) = 0$ is the plane equation

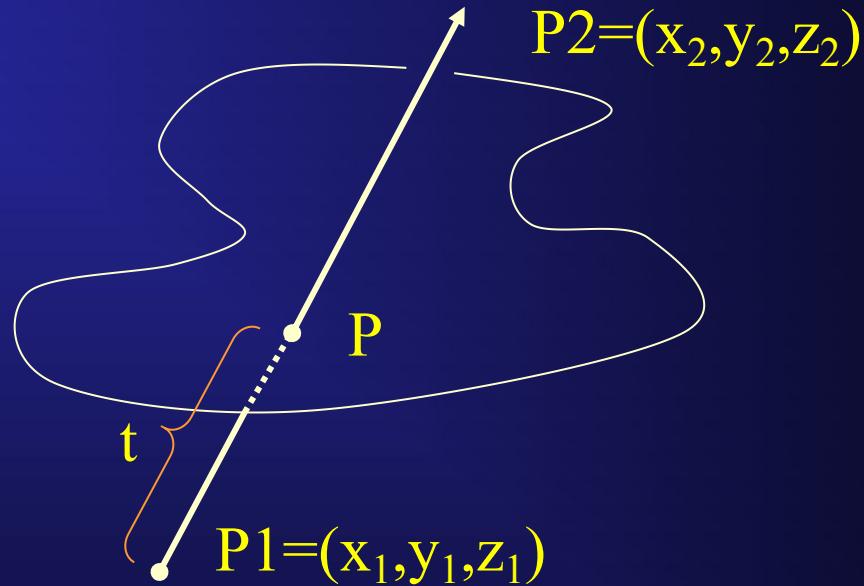
(for any points S (e.g., known, perhaps a polygon vertex defining the plane) and P (unknown) on the plane).

$P = P_1 + t(P_2 - P_1)$ is the line equation (P unknown).

Substituting: $N \cdot (P_1 + t(P_2 - P_1) - S) = 0$

And solving: $t = N \cdot (S - P_1) / N \cdot (P_2 - P_1)$

Intersection of Ray with Arbitrary Plane



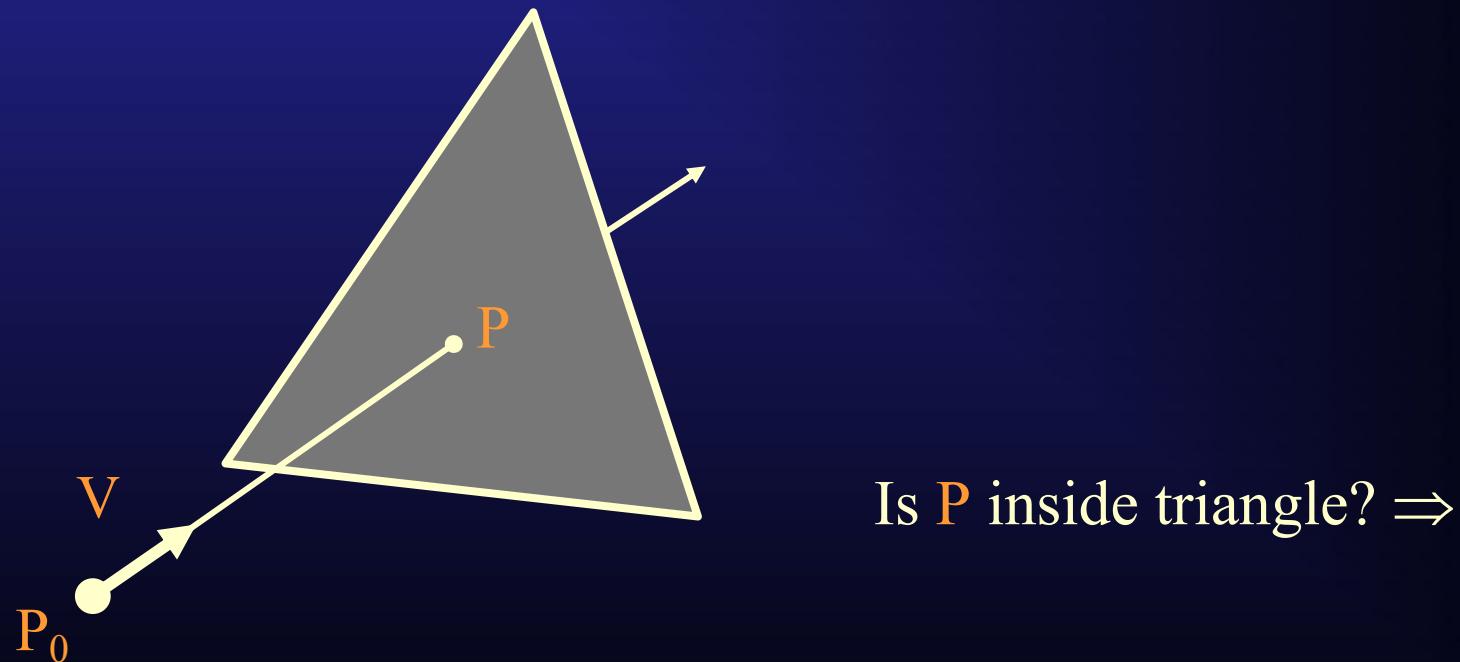
If intersection with ray is required, check $0 \leq t$.

If intersection with segment is required, check $0 \leq t \leq 1$.

In either case, if the denominator is 0, then the line $P_2 - P_1$ is parallel to the plane and there are either no solutions or an infinite number (line in plane).

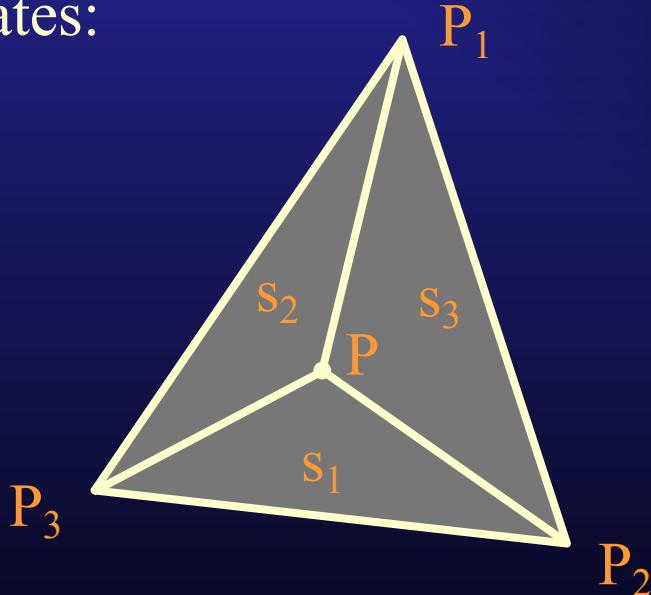
Ray-Triangle Intersection

- Intersect ray $P_0 + tV$ with plane of triangle (yielding point P).
- Check if $t \geq 0$ and P is inside triangle.



Ray-Triangle Intersection Inside Test

- Several possible methods to check P in $\Delta P_1P_2P_3$
- (Normalized) **Barycentric** coordinates:



$$s = \text{area}(\Delta P_1P_2P_3) \quad [*]$$
$$s_1 = \text{area}(\Delta PP_2P_3) / s$$
$$s_2 = \text{area}(\Delta PP_3P_1) / s$$
$$s_3 = \text{area}(\Delta PP_1P_2) / s$$

$$P = s_1P_1 + s_2P_2 + s_3P_3$$

Then P is inside if

$$0 \leq s_1 \leq 1$$

$$0 \leq s_2 \leq 1$$

$$0 \leq s_3 \leq 1$$

$$s_1 + s_2 + s_3 = 1$$

* See next slide

Area of Triangle: via Determinants

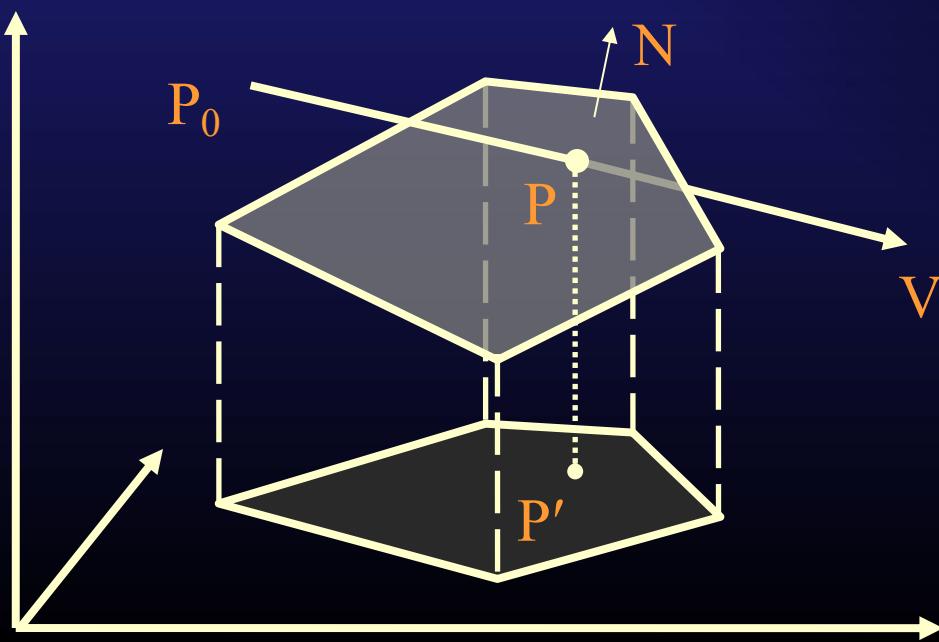
$$\Delta area = \frac{1}{2} \sqrt{\left| \begin{vmatrix} P_1^y & P_1^z & 1 \\ P_2^y & P_2^z & 1 \\ P_3^y & P_3^z & 1 \end{vmatrix} \right|^2 + \left| \begin{vmatrix} P_1^z & P_1^x & 1 \\ P_2^z & P_2^x & 1 \\ P_3^z & P_3^x & 1 \end{vmatrix} \right|^2 + \left| \begin{vmatrix} P_1^x & P_1^y & 1 \\ P_2^x & P_2^y & 1 \\ P_3^x & P_3^y & 1 \end{vmatrix} \right|^2}$$

- See http://en.wikipedia.org/wiki/Rule_of_Sarrus
- E.g.:

$$\left| \begin{vmatrix} P_1^y & P_1^z & 1 \\ P_2^y & P_2^z & 1 \\ P_3^y & P_3^z & 1 \end{vmatrix} \right| = (P_1^y P_2^z - P_2^y P_1^z) + (P_1^z P_3^y - P_3^z P_1^y) + (P_2^z P_3^y - P_3^z P_2^y)$$

Ray-Polygon Intersection

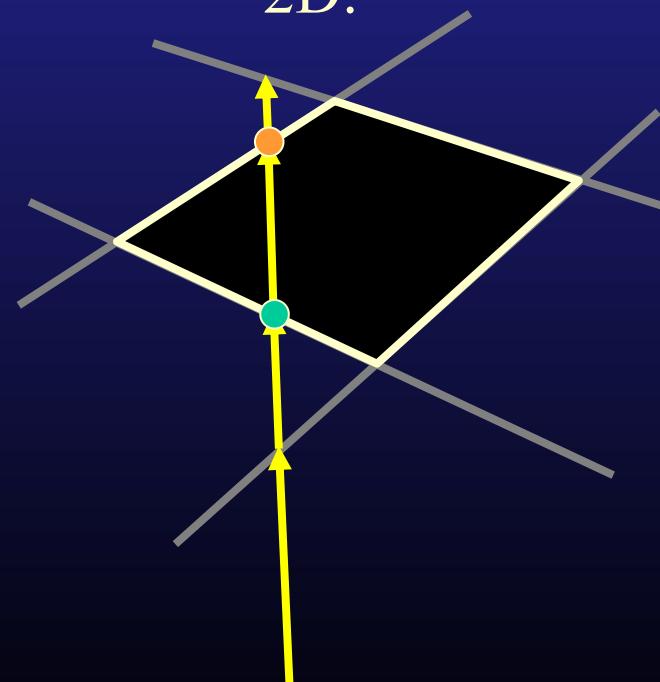
- Intersect ray $P_0 + tV$ with polygon's plane of support (yielding point P); plane normal is $N=(N_x, N_y, N_z)$.
- Check if P is inside 3D polygon by projecting polygon and P onto xy , xz , or yz plane (depending on which is largest, respectively, $|N_z|$, $|N_y|$, or $|N_x|$) and doing 2D point P' in polygon test.
- (Don't really *project*; just ignore the extra coordinate.)



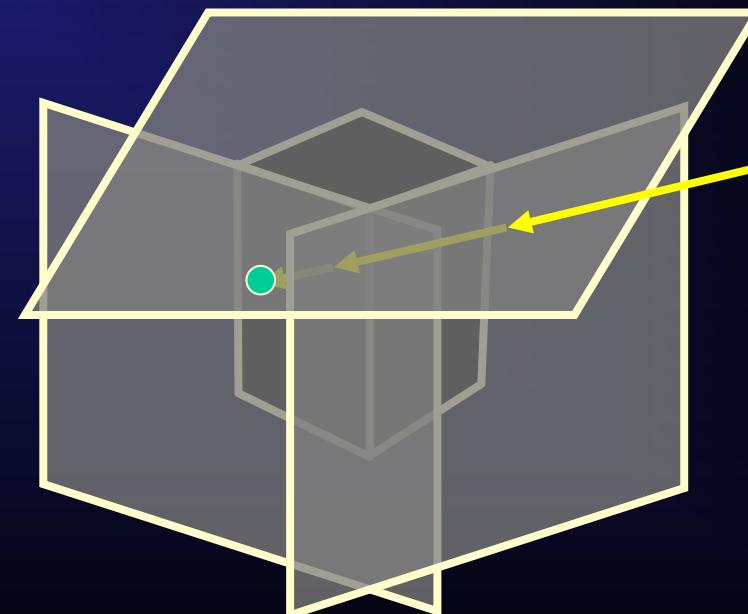
Ray-Convex Polyhedron Intersection

- Intersect ray with planes of **convex** polyhedral volume (box, tetrahedron, prism, etc.)
- Ray enters at **furthest** intersection with **front-facing** surfaces.
- Ray exits at **closest** of **rear-facing** surfaces.

2D:

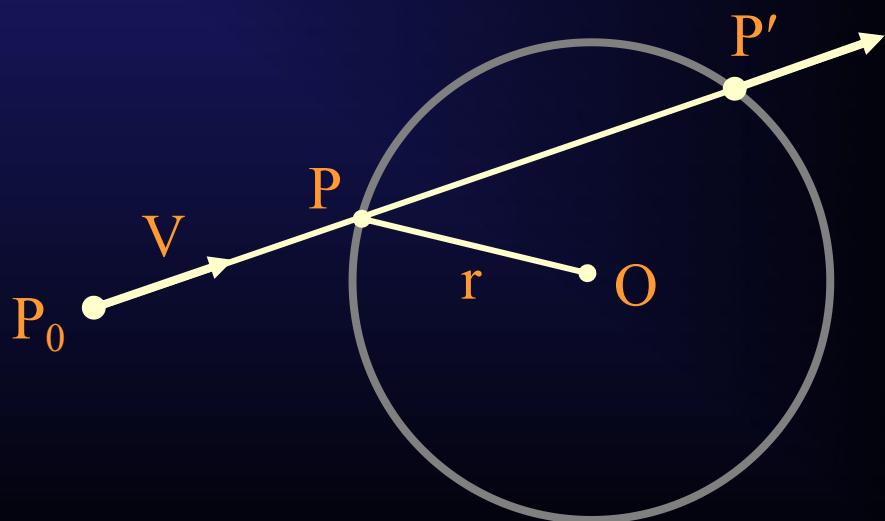
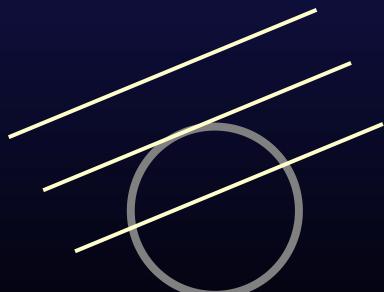


3D:



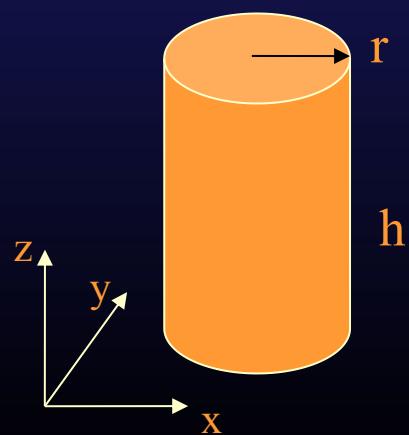
Ray-Sphere Intersection (Algebraic Method)

- Intersect ray $P = P_0 + tV$ with sphere $|P - O|^2 - r^2 = 0$.
- Substitute ray form into sphere equation: $|P_0 + tV - O|^2 - r^2 = 0$.
- Solve quadratic equation $at^2 + bt + c = 0$ where
 $a = 1; \quad b = 2V \cdot (P_0 - O); \quad c = |P_0 - O|^2 - r^2$
(expand $|P_0 + tV - O|^2 = |tV + P_0 - O|^2 = |tV + (P_0 - O)|^2$ and
recall that $V \cdot V = 1$)
$$t = (-b \pm \sqrt{b^2 - 4ac}) / 2a$$
- 0, 1, or 2 solutions!



Ray-Cylinder Intersection

- Three separate surfaces to consider:
 - 2 planar end caps, e.g.:
$$|P_x - O_x|^2 + |P_y - O_y|^2 - r^2 \leq 0 \text{ and } P_z = 0 \text{ or } P_z = h$$
 - Cylinder body: $|P_x - O_x|^2 + |P_y - O_y|^2 - r^2 = 0$ and $0 \leq P_z \leq h$
- We've seen how to do the planar ends.
- Cylinder is similar to sphere except have 2D circle and a height.
- Ray intersects cylinder if and only if it intersects planar end cap inside circle or cylinder wall in between end caps.



(Watch for ray in surface cases!)

Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Image Synthesis

SHADING

REFLECTIONS

TRANSLUCENCY

BUMPS

SHADOWS

HIGHLIGHTS

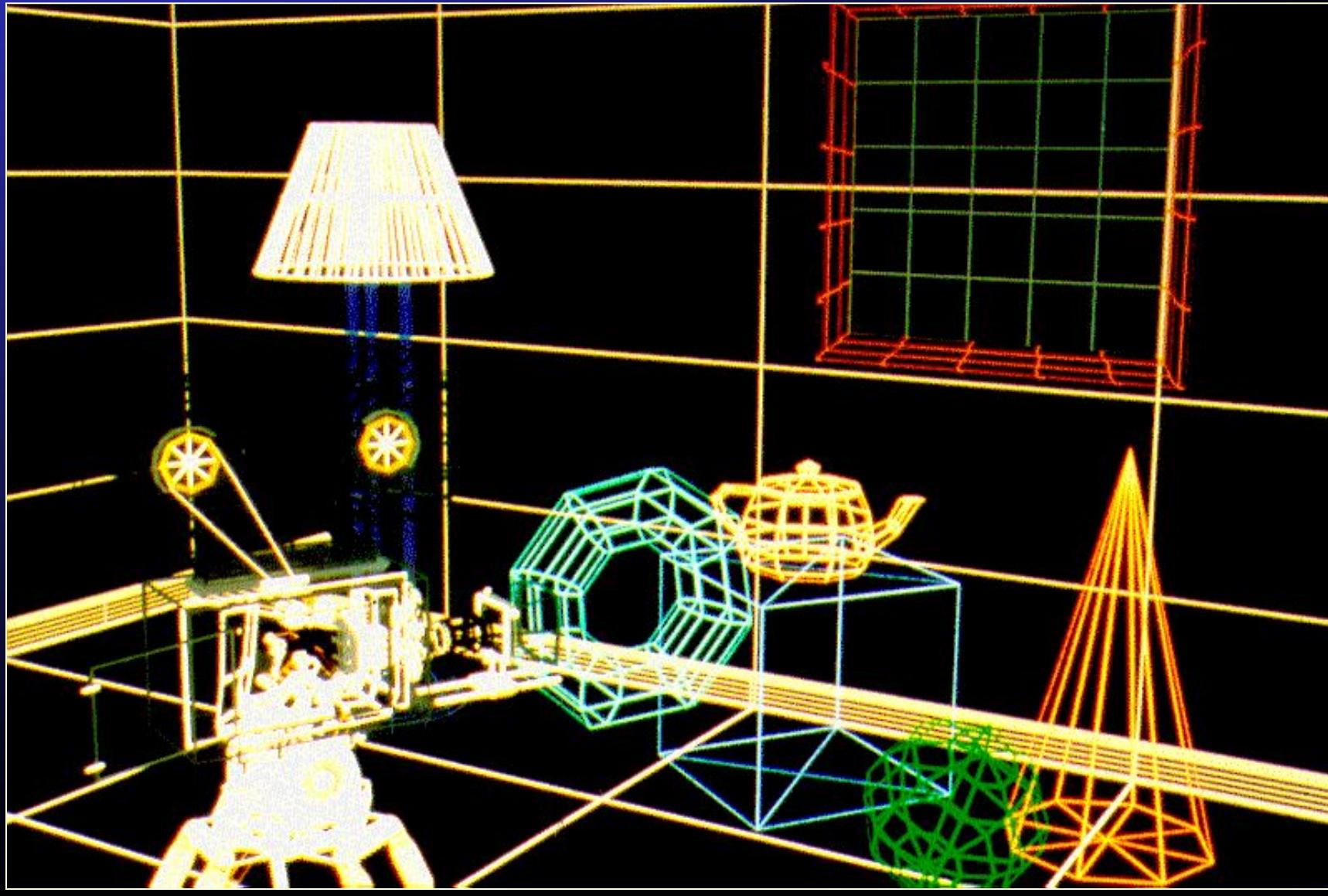
TEXTURE

REFRACTIONS

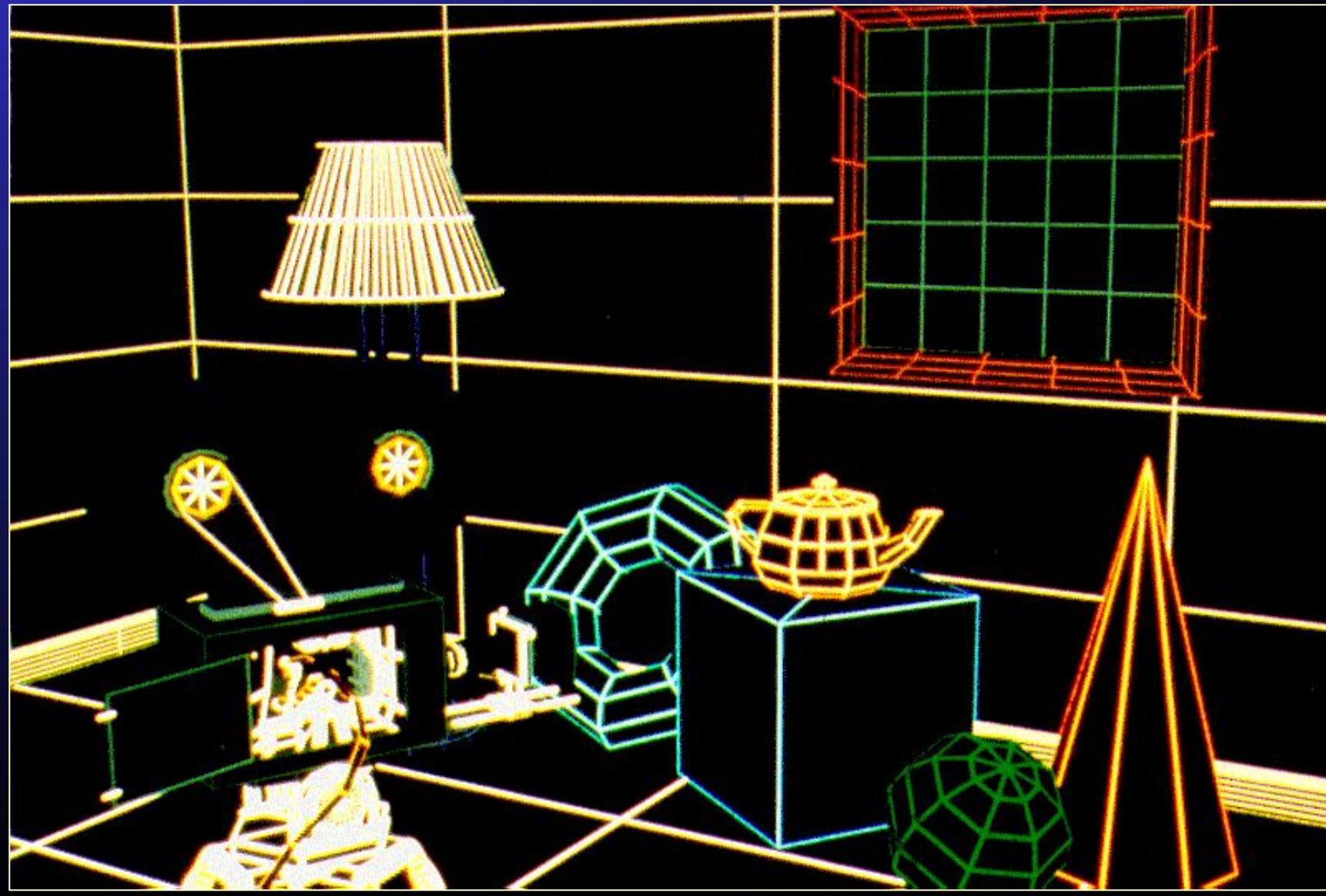
Illumination and Shading Models

- CG is the study of **geometry** and **light** and their **interactions**.
- Render depth and lighting effects to provide 3D perceptions.
- Improve realism -- to make models and scenes appear more like the “real world”.
- Originated in trying to give polygonal models the *appearance* (*illusion*) of smooth curvature.
- Numerous shading models
 - Simple; local illumination
 - Physics-based; global illumination
 - Specific techniques for particular effects
 - Non-photorealistic rendering (NPR) techniques (pen and ink, brushes, etching)

Wireframe: Color but no Substance



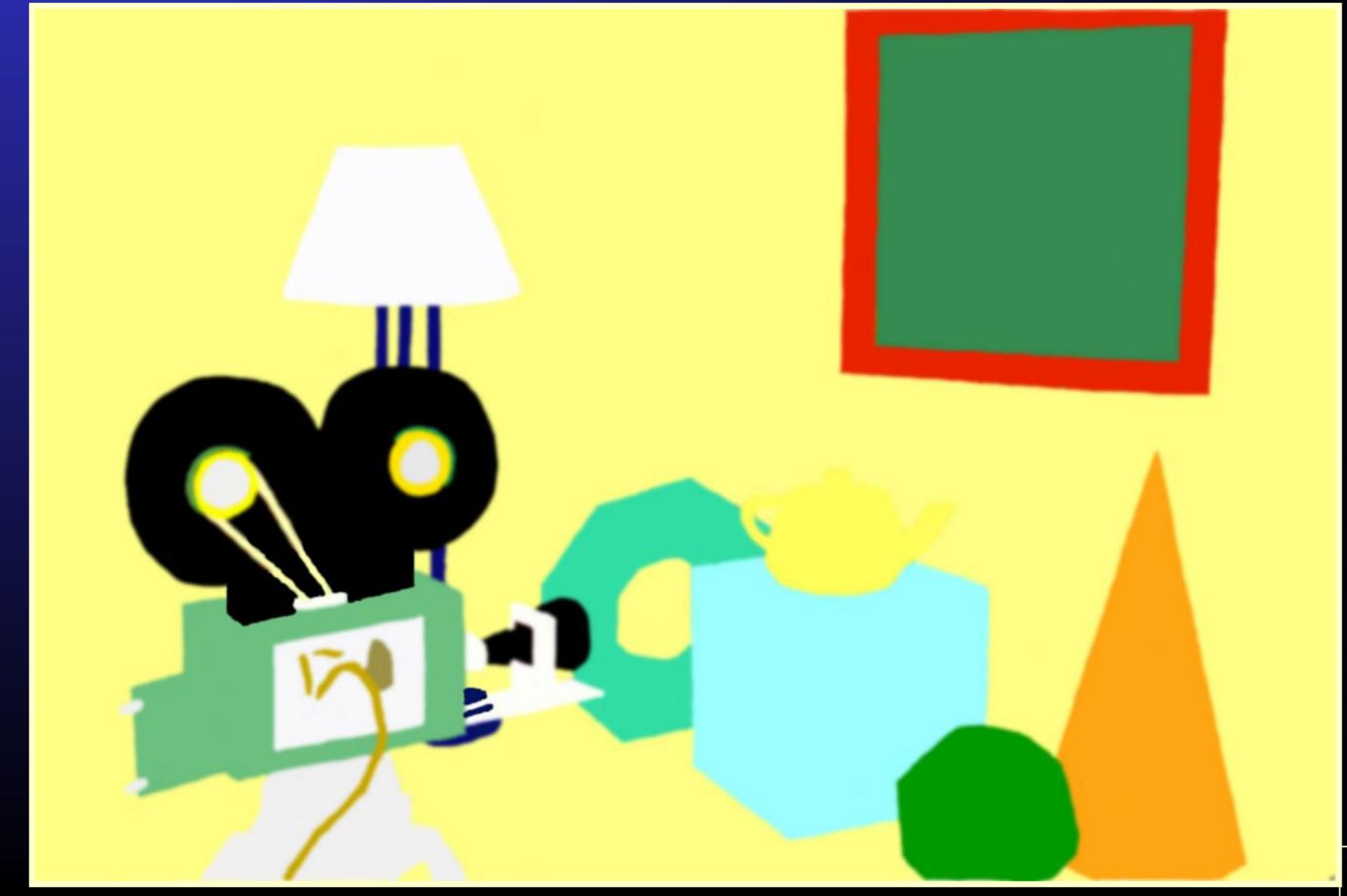
With Visible Lines Only: Substance but no Surfaces



Trivial (Flat) Shading

- Assign polygon or triangle color (R, G, B) to each polygon or triangle vertex uniformly (e.g., red triangle means each vertex is assigned color (1,0,0)).

Object Surface Color \neq Visible Shading;
That's why the Surface Normal is Needed!



Shading Computation

- Given direction V from a point P to eye point E and (point) light source L . Assume all vectors are normalized to unit length.
- Know surface geometry, know surface normal, or can otherwise compute the surface normal N at P .
- Know/specify desired color (or properties) of surface containing P .
- Compute *reflected* color at P as a function of L, N, V and other attributes of the surface and lighting, based on:
 - Measured (empirical) reflectance properties
 - Phenomenological models: most simple CG reflection models
 - Simulation: microgeometry, particles, threads
 - Physical optics: radiance, wavelength
 - Geometric optics: derived from shape and/or surface statistics
- We'll start with a LOCAL illumination model, and return to more physically correct models later.

Reflection Categories (General)

a) Diffuse

- Light reflected equally in all directions

b) Glossy specular

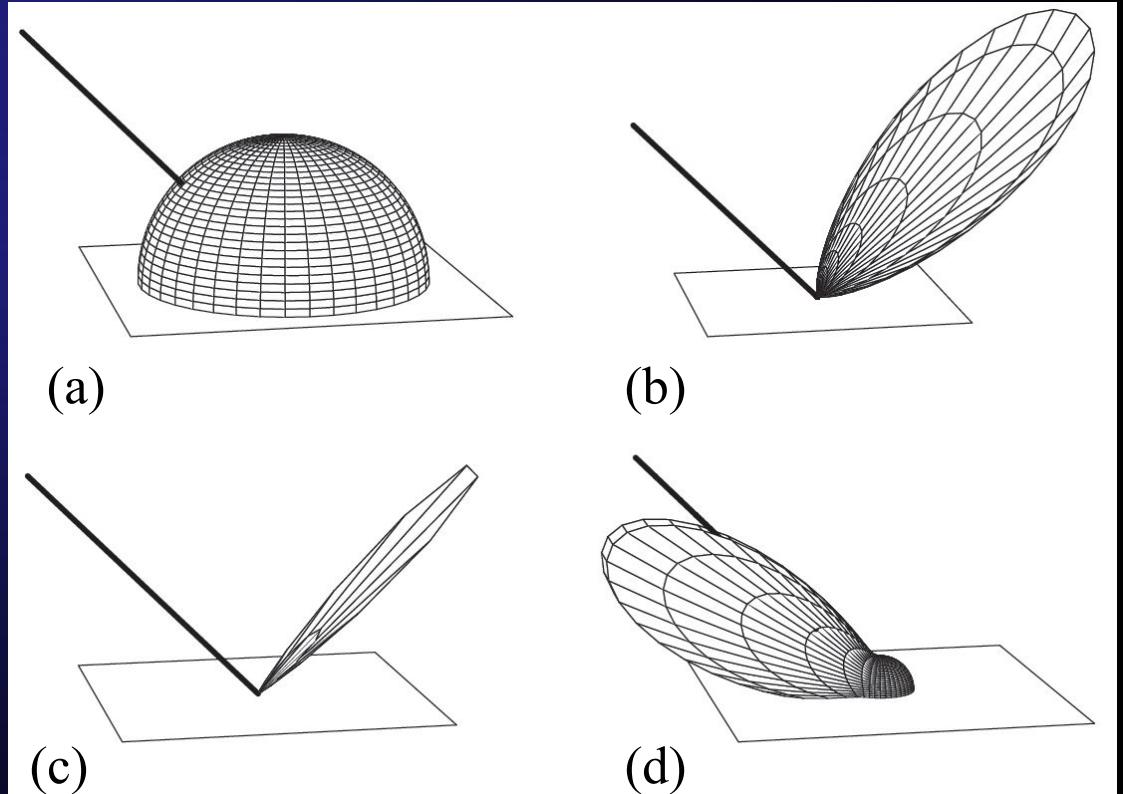
- Light reflected generally in a preferred direction

c) Perfect specular

- Single outgoing direction, like a mirror

d) Retro-reflective

- Primarily back along incident direction

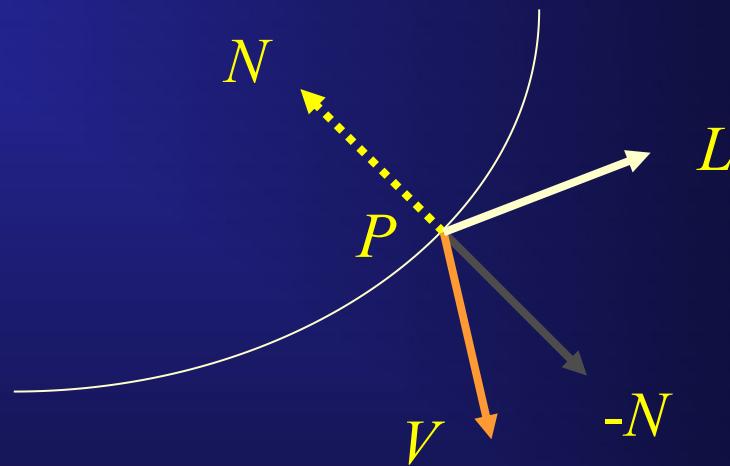


Shade Computation

Models:

- Diffuse or Lambert
- Vertex color interpolation and Gouraud
- Perfect specular
- Phong: Glossy specular
- Recursive ray tracing
- Microfacet
- Empirical

There are Really Two Surface Normals at Point P



Want outward-facing normal :

If $((N \cdot V) < 0)$ and $((N \cdot L) < 0)$ then $N \leftarrow -N$

If $(N \cdot L) < 0$ then P is not illuminated by L as seen from V .

Light Reflection from a Surface at a Point (Simple, local model)

$$I_{\text{reflected}} = k_a I_a + k_d I_d + k_s I_s$$

$$\text{for } k_a + k_d + k_s = 1$$

to account for all reflected light.

This means the surface shade $I_{\text{reflected}} = (r, g, b)_{\text{reflected}}$ is a function of
 I_a *ambient* light
 I_d *diffuse reflection*
 I_s *specular reflection*
themselves expressed in (r, g, b) form.

We need to establish working definitions for these terms.

The Three Reflectance Terms

- Diffuse: $I_d = I_i(L \cdot N)$

Where I_i is the intensity of the incident (incoming) light.

- Specular: $I_s = I_i(R \cdot V)^n$

- Ambient: I_a = “omnidirectional” light factor

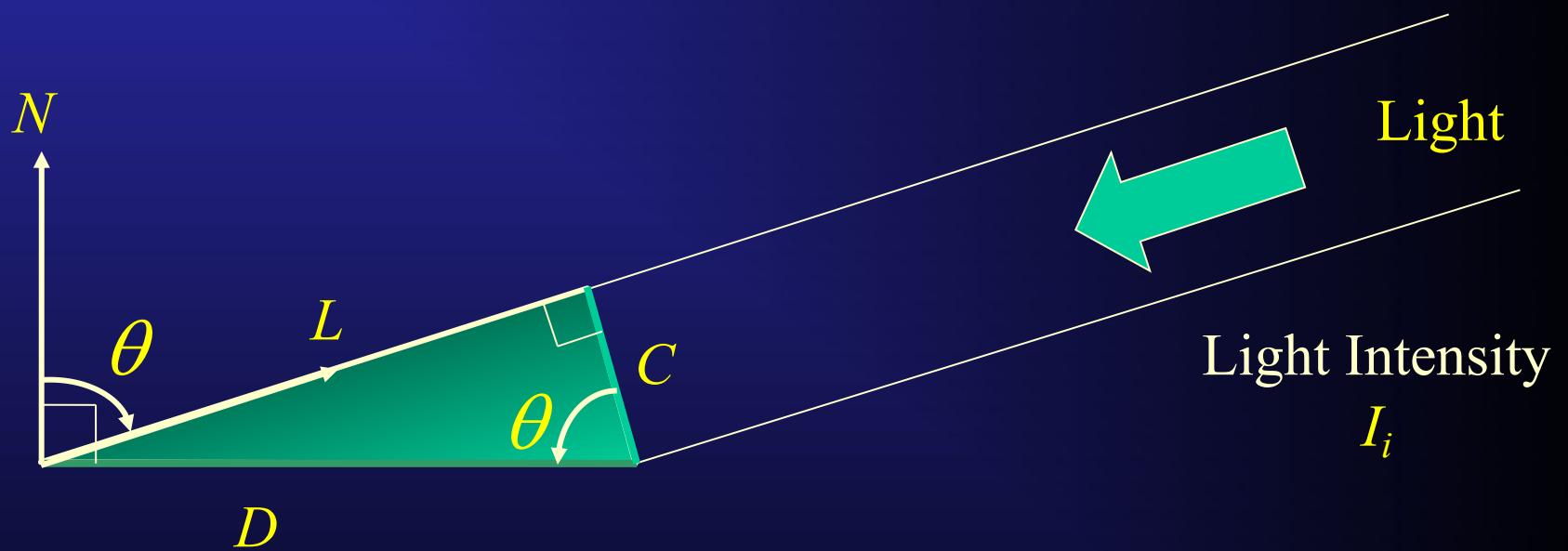
Let’s start by motivating the diffuse term. (We’ll see how to actually get rid of the ambient term later.)

Shade Computation

Models:

- DIFFUSE or LAMBERT
- Vertex color interpolation and Gouraud
- Perfect specular
- Phong: Glossy specular
- Recursive ray tracing
- Microfacet
- Empirical

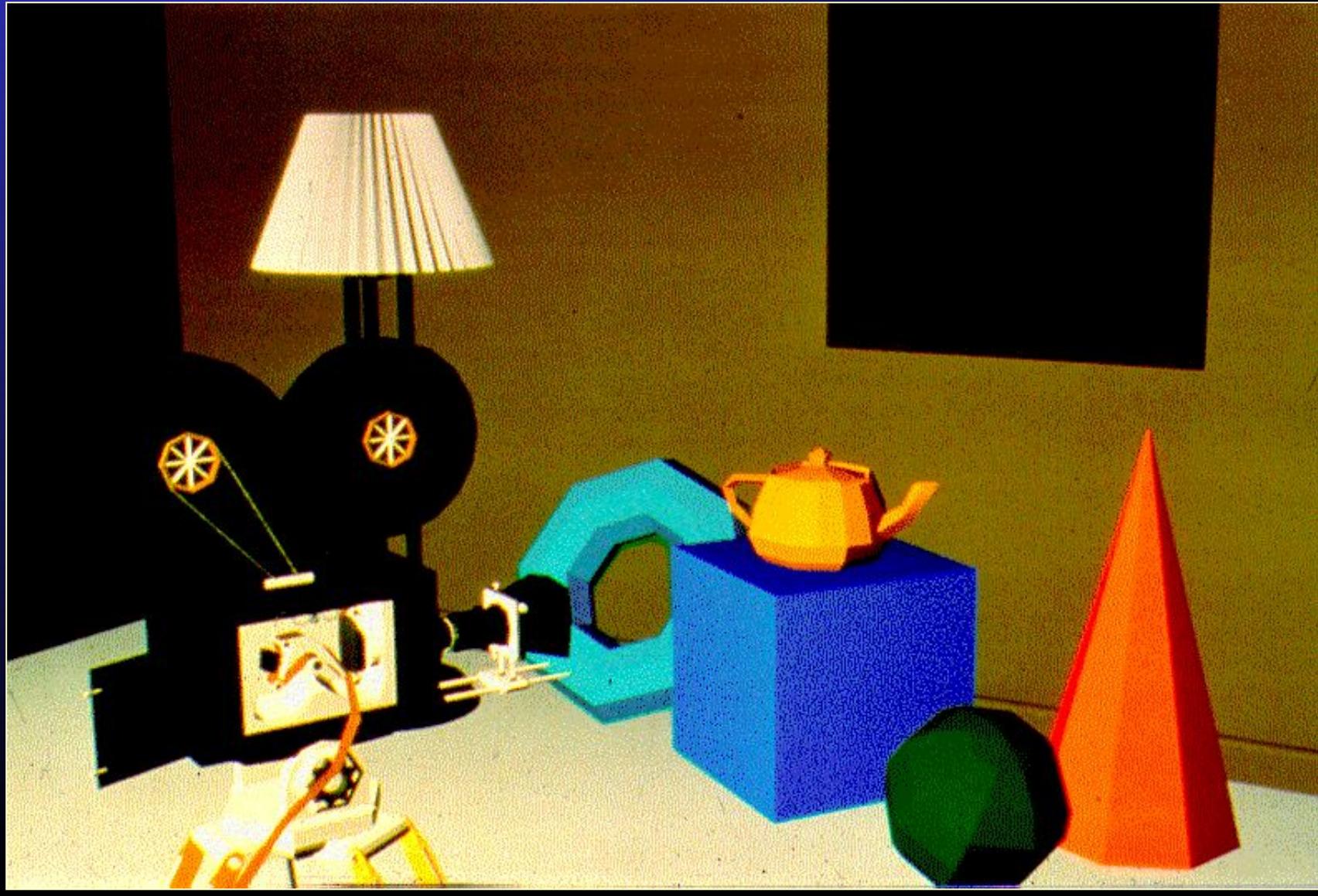
Diffuse Shading: Lambert's Cosine Law of Illumination



$$I_d = I_i \frac{C}{D} = I_i \cos(\theta) = I_i(L \cdot N) \text{ or } 0 \text{ if } L \cdot N < 0$$

Note that reflected light I_d is independent of observer (V) direction!

Diffuse Shading on Polygon Surfaces



Shade Computation

Models:

- Diffuse or Lambert
- VERTEX COLOR INTERPOLATION and GOURAUD
- Perfect specular
- Phong: Glossy specular
- Recursive ray tracing
- Microfacet
- Empirical

Diffuse Computation and Interpolation for Triangles

- There are multiple ways of assigning a color to a triangle vertex; this diffuse method is just one way.
- If light source is a point infinitely far away, then the diffuse shading for each vertex of a triangle will be the same (it only depends on the constant normal N and light direction L).
- If the point light is located at a finite distance, then the direction L will vary by vertex. In this case we can compute the diffuse term for each vertex of the triangle and interpolate shades on the interior of the triangle.
- Method commonly used for simple vertex shading.

Triangle Vertex Color Interpolation

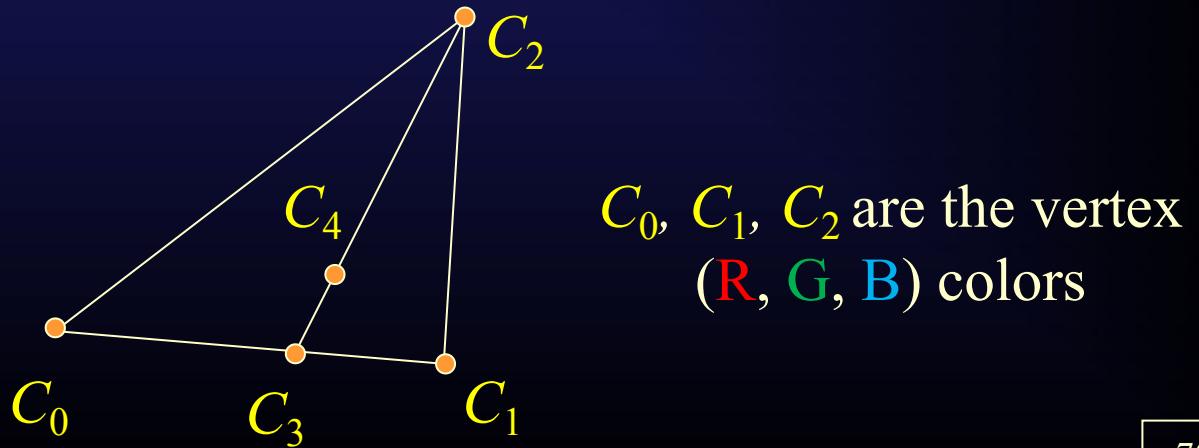
- Linear interpolate with parameter α each of the (R, G, B) color components along an edge C_{01} :

$$C_{01}(\alpha) = (1 - \alpha)C_0 + \alpha C_1$$

- As α goes from 0 to 1, compute color C_3 . ■
- Then linearly interpolate with parameter β the color from the other vertex C_2 to C_3 , yielding color at any interior point C_4 : ■

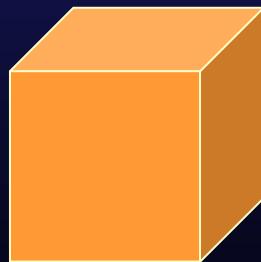
$$C_{32}(\beta) = (1 - \beta)C_3 + \beta C_2$$

- (Order of edges used does not matter.)

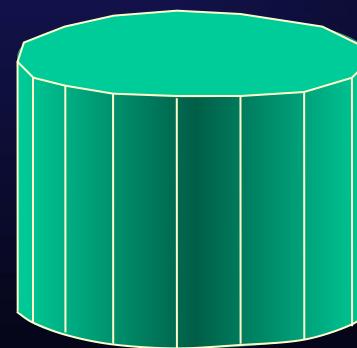


Gouraud Shading to Fake Surface Curvature on Polygon Models

- Fake curvature due to neighboring polygons
 - Often make edge type dependent on dihedral angle between polygon faces:
 - If around 180° plus/minus a threshold (say 15°), shade smoothly.
 - Otherwise treat as sharp (defined) edge.



“hard” edges

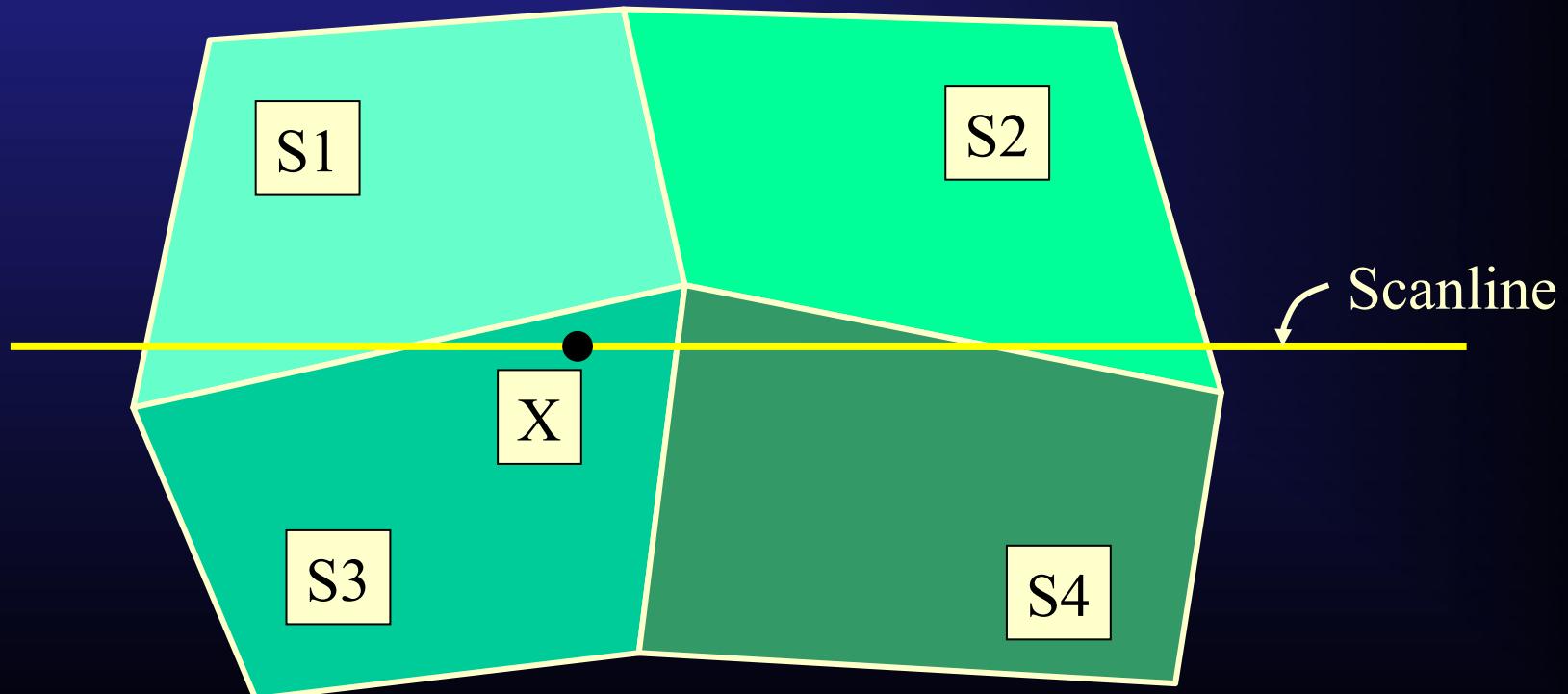


“soft” edges

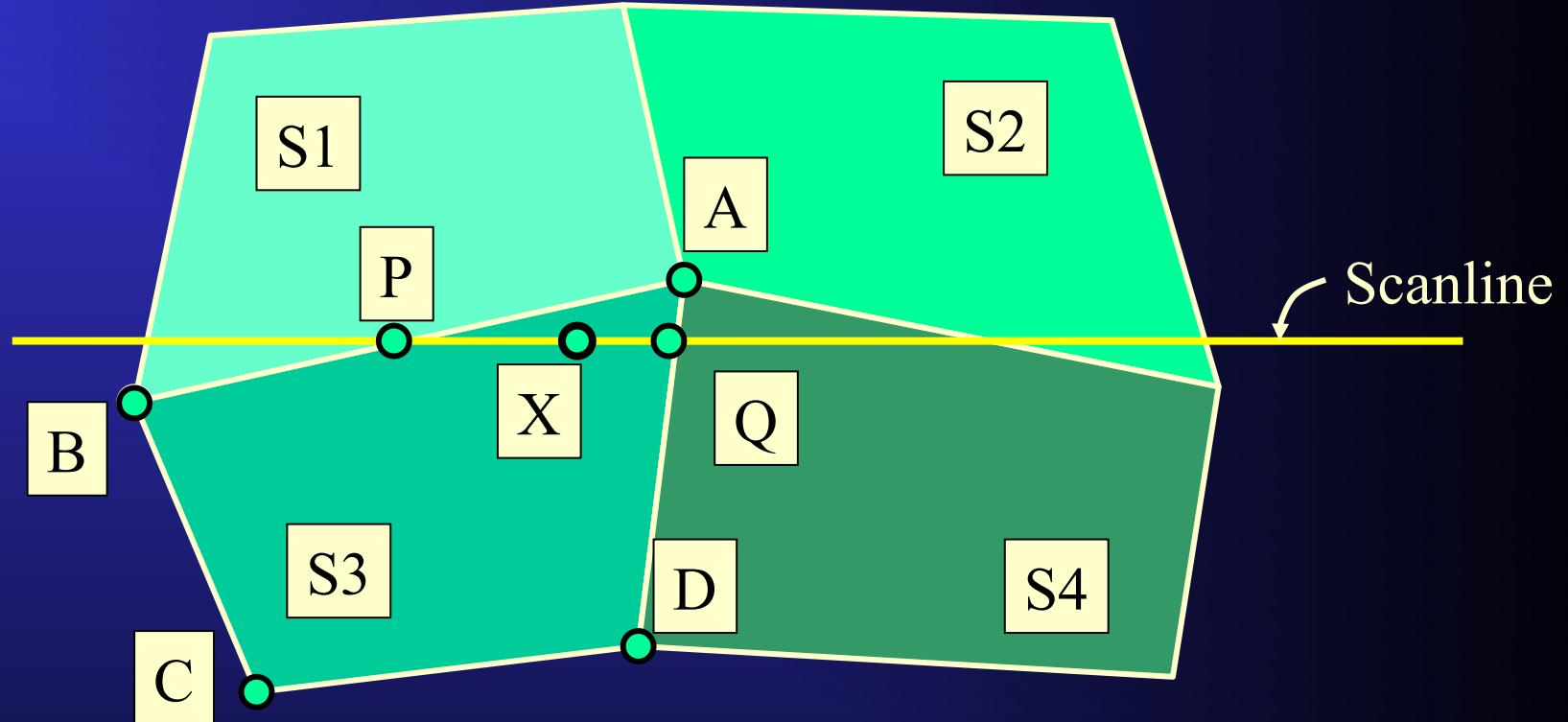
Gouraud Shading

For faked smooth shading:

First compute color shades of polygons (e.g., by local illumination model), then interpolate shades.



What is the Gouraud shade at X?

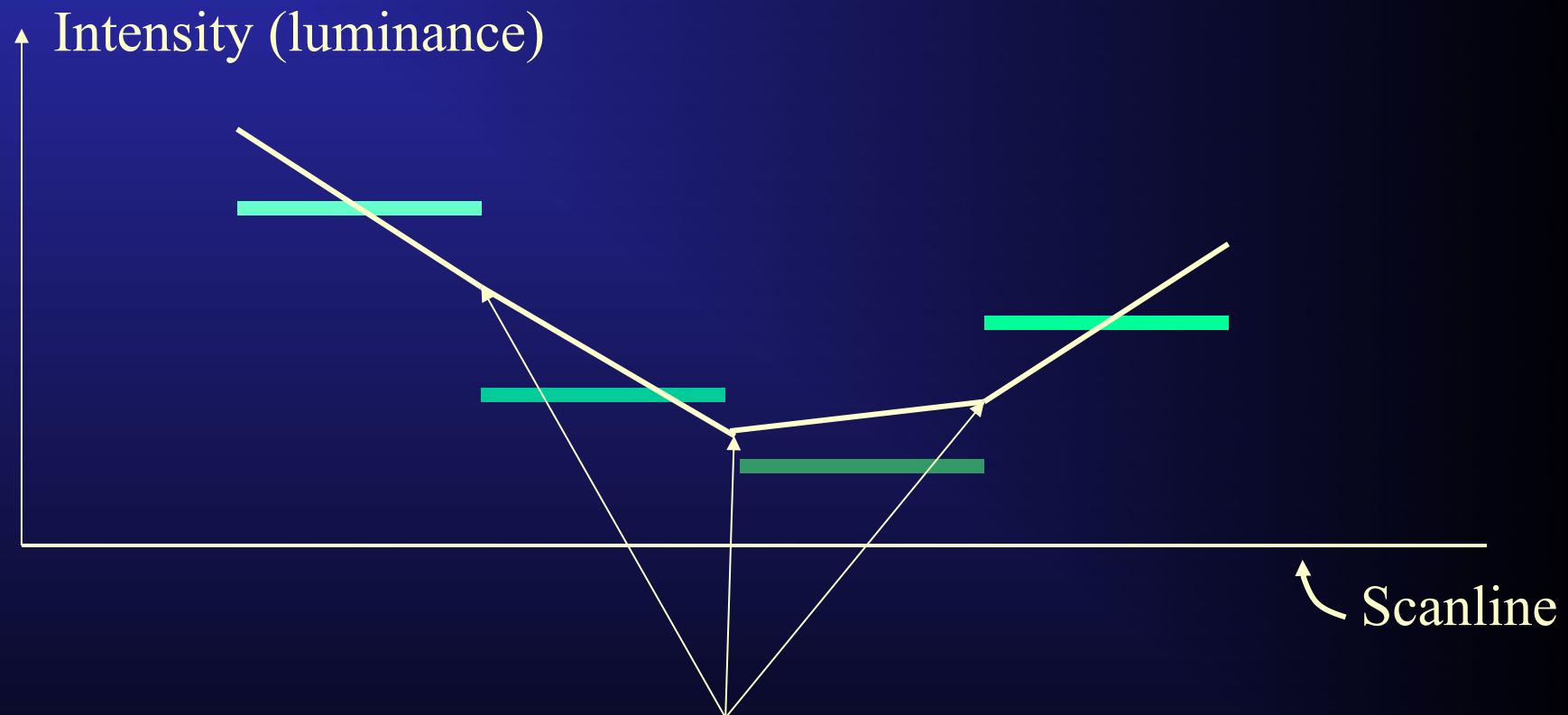


- 1. Vertex-vertex interpolation:
 - ➡ $\text{Shade(A)} = 1/4 [\text{color(S1)} + \text{color(S2)} + \text{color(S3)} + \text{color(S4)}]$
 - ➡ Similar computations for Shades at B, C, and D.
 - ➡ $\text{Shade(P)} = \text{weighted average of Shade(A) and Shade(B)}$
 - ➡ $\text{Shade(Q)} = \text{weighted average of Shade(A) and Shade(D)}$
- 2. Scanline interpolation:
 - ➡ $\text{Shade(X)} = \text{weighted average of Shade(P) and Shade(Q)}$

The colors before interpolation along the scanline

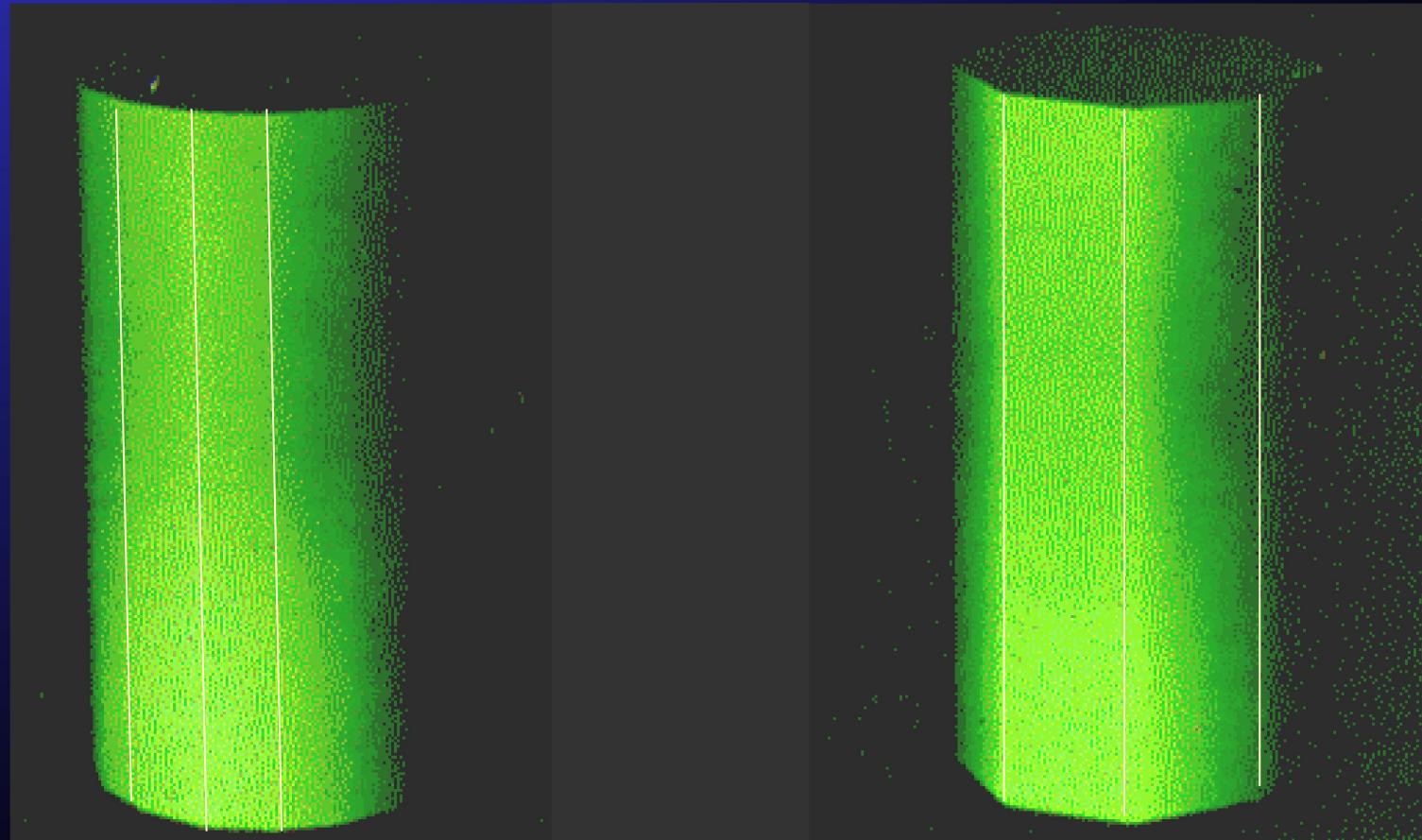


After linear interpolation along the scanline

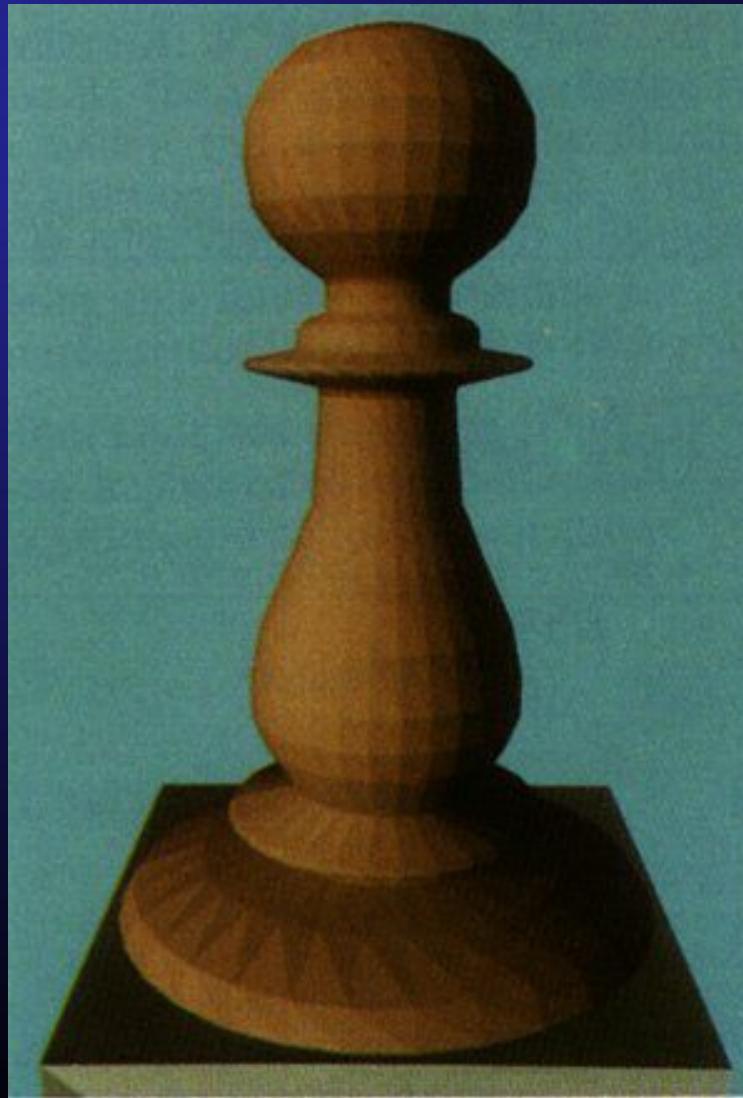


Because we interpolate linearly, we get smooth ramps BUT
DISCONTINUOUS FIRST DERIVATIVES at every edge:
creating MACH Bands!

Mach bands on cylinders with polygon faces



Mach Bands (Bishop, SIGGRAPH '86)



Shade Computation

Models:

- Diffuse or Lambert
- Vertex color interpolation and Gouraud
- Perfect specular
- Phong: Glossy specular
- Recursive ray tracing
- Microfacet
- Empirical

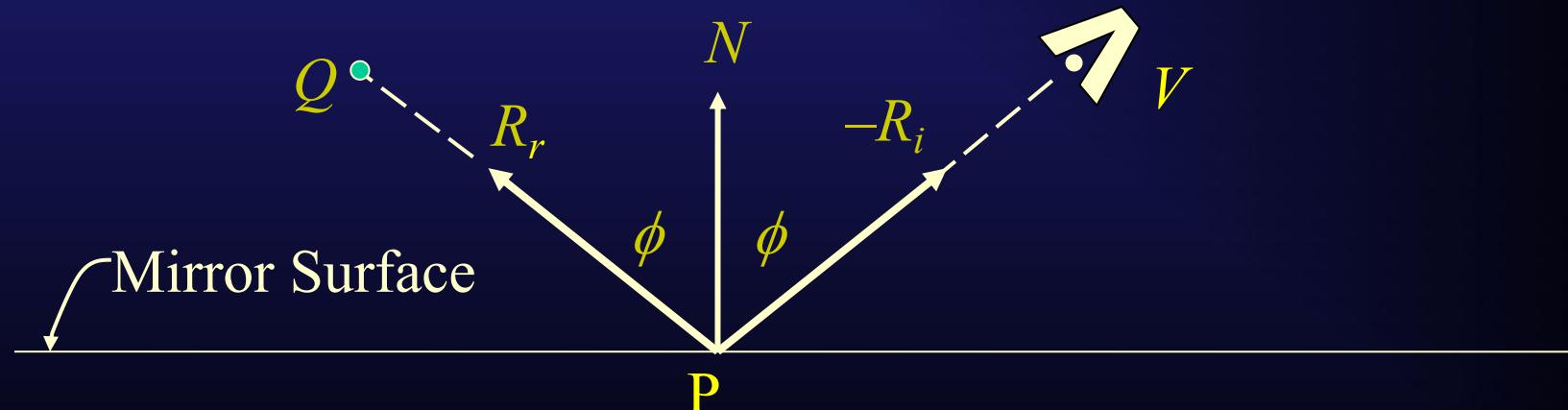
Perfect Specular (Mirror) Reflection

R_i = Incident ray direction (unit vector) from eye (V) to some surface point P .

R_r = Reflected ray direction (about P).

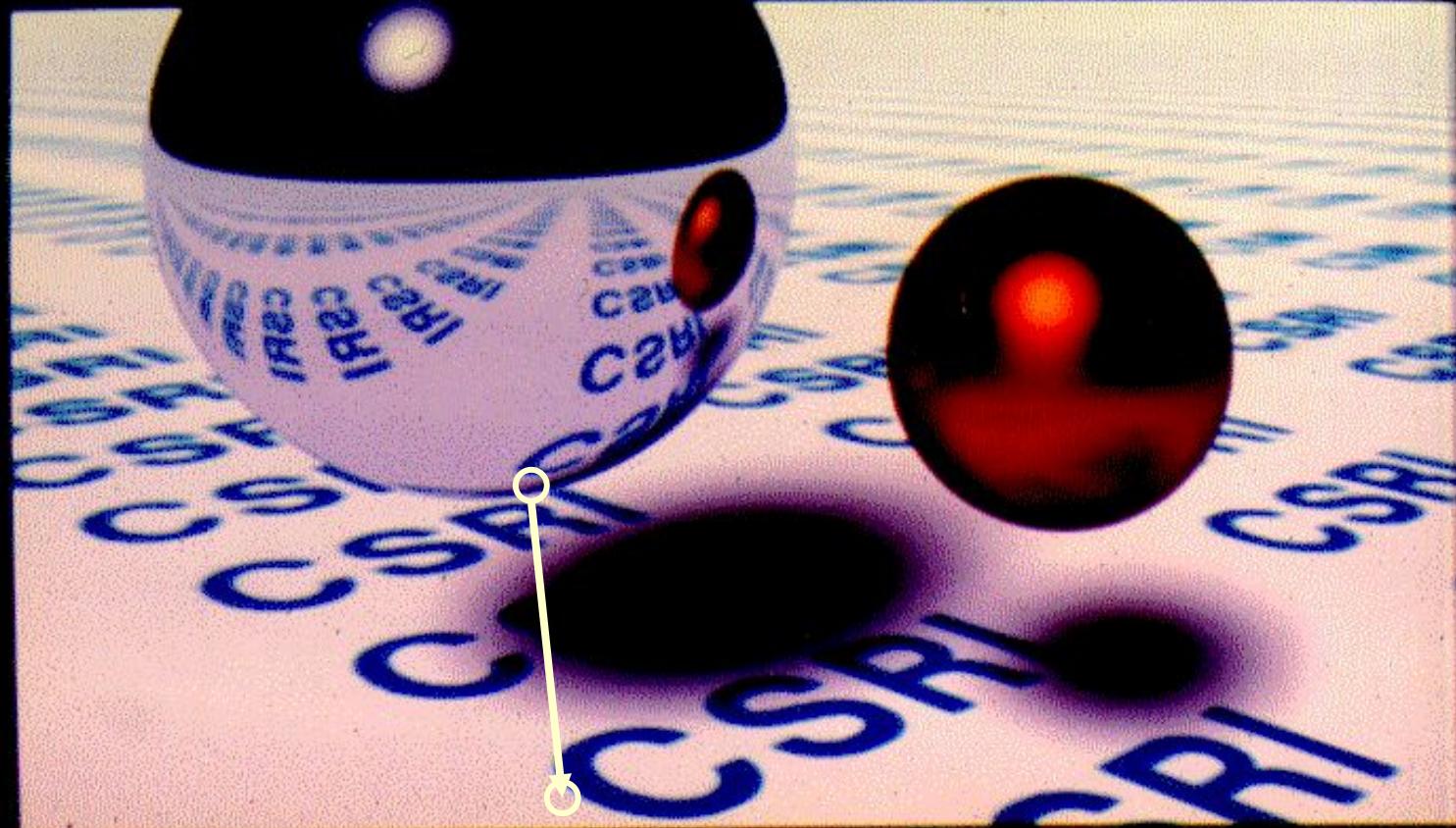
$$R_r = R_i - 2 N (R_i \bullet N)$$

Therefore if $V = -R_i$ then eye sees Q else eye does not.



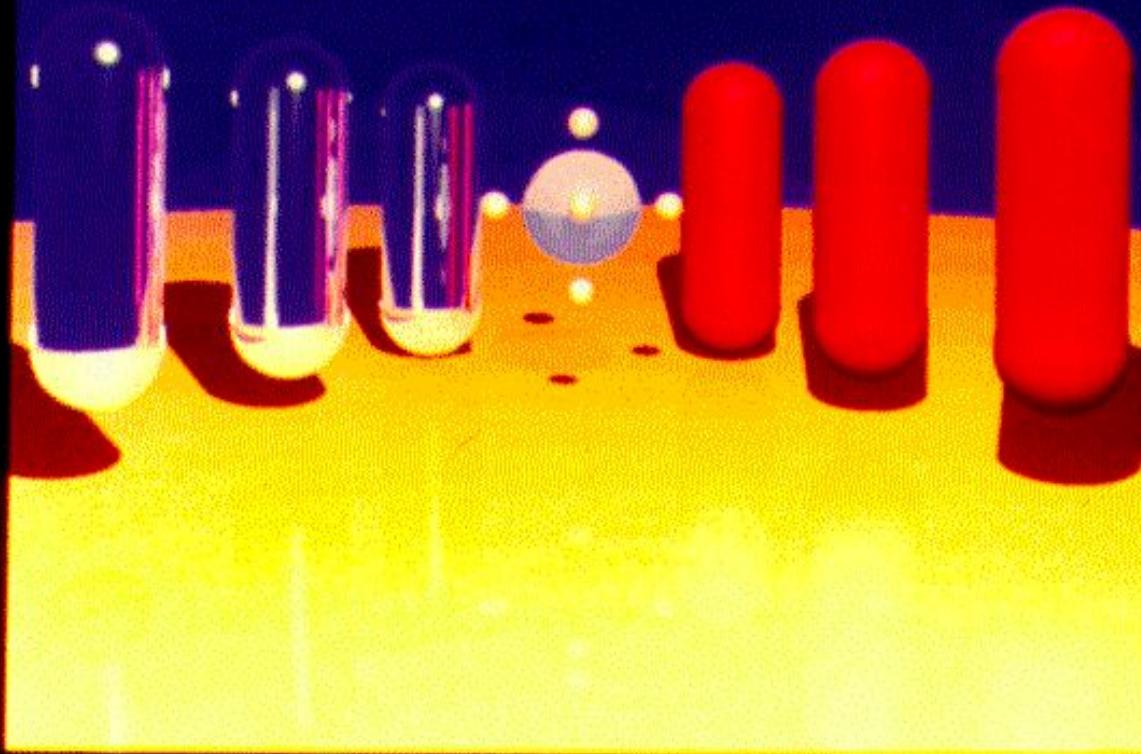
[Check: If $R_i \parallel N$ (i.e., $-R_i = N$) then $R_r = N$; if $-R_i \perp N$ then $R_r = R_i$]

Mirror Reflection Example



© 1985 JOHN AMANATIDES—UNIV. OF TORONTO

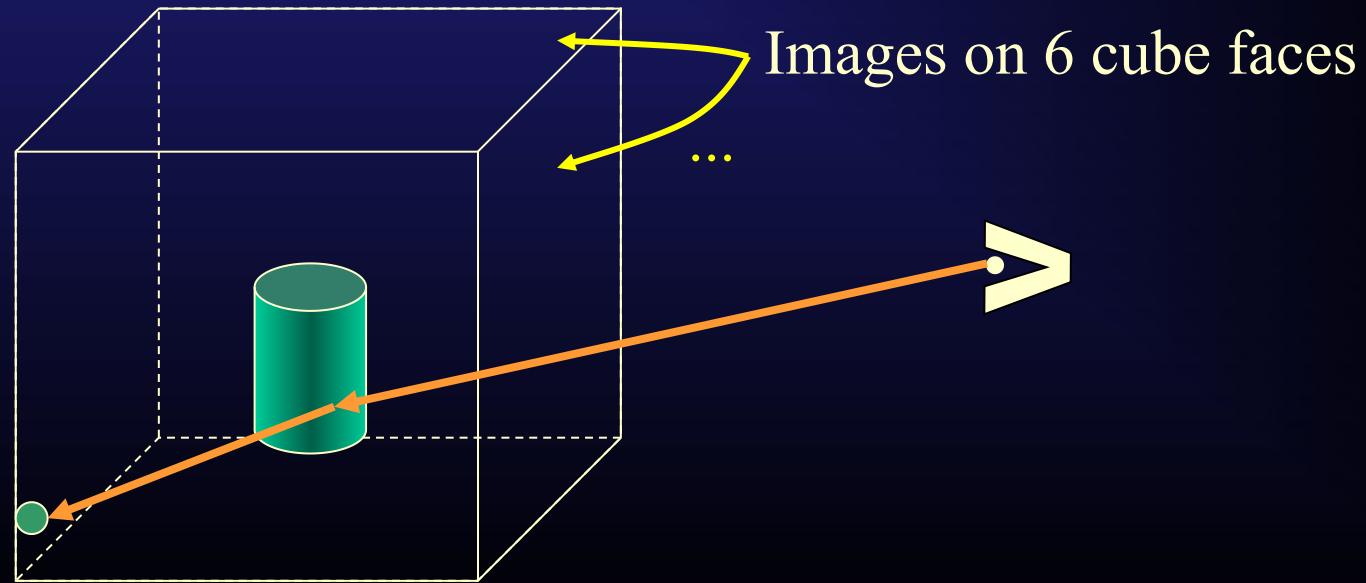
Mirror vs. Diffuse Reflection Example



DAVIS, J.—PURDUE

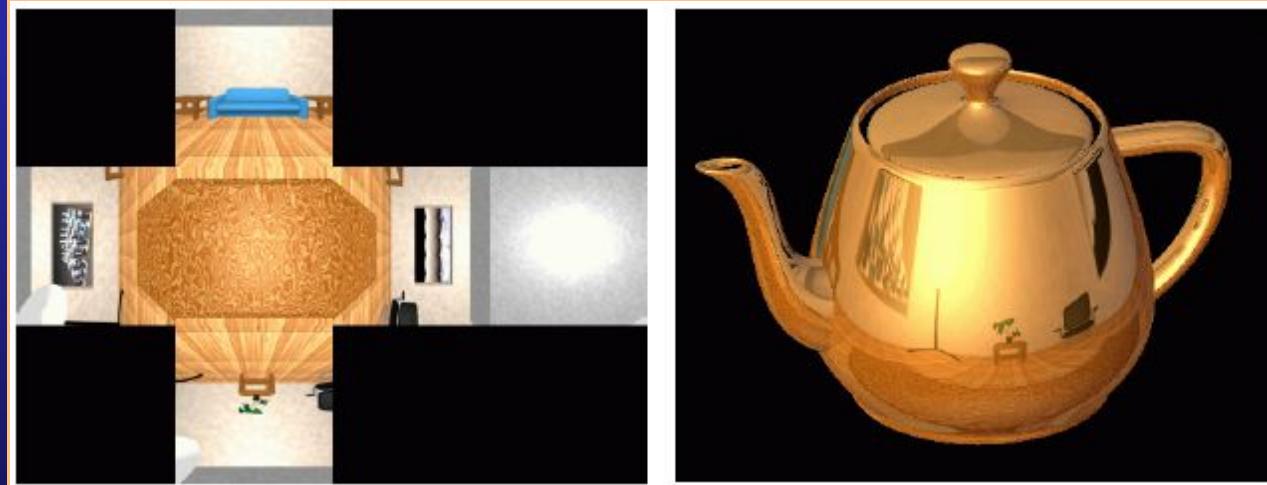
Application of Mirror Reflection: The Background or World Map

- Place 3D scene inside cube or sphere of *background* images: **world or environment map**.
- Use mirror reflected ray to hit world map to find reflected color.



Use Environment Maps Rendered from Virtual Models or Constructed from Real Scenes

Cube



Sphere



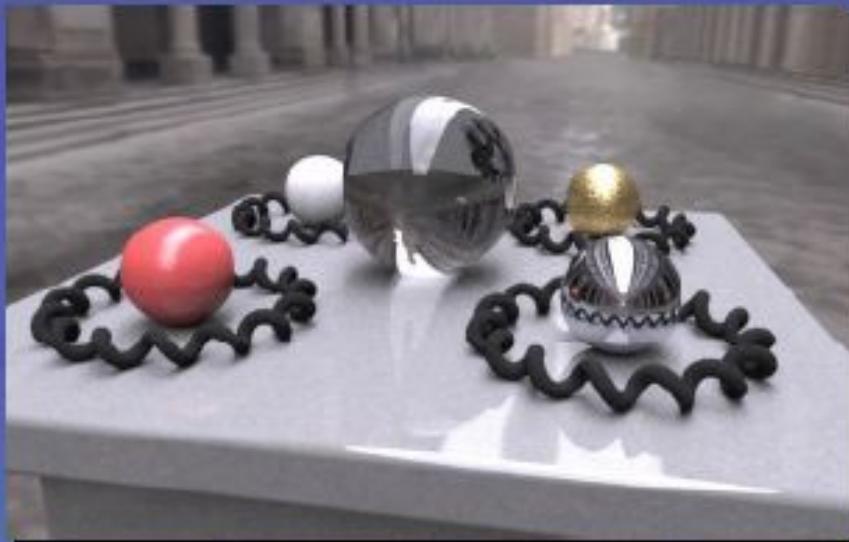
Environment Map

- Contains view of the world (light) from the object of interest's approximate center.
- Images stored on environment space surfaces (sphere or cube).
- Introduces distortions, so better for small object in a large room (or outdoor) space; but usually don't notice.
- Object doesn't reflect itself.

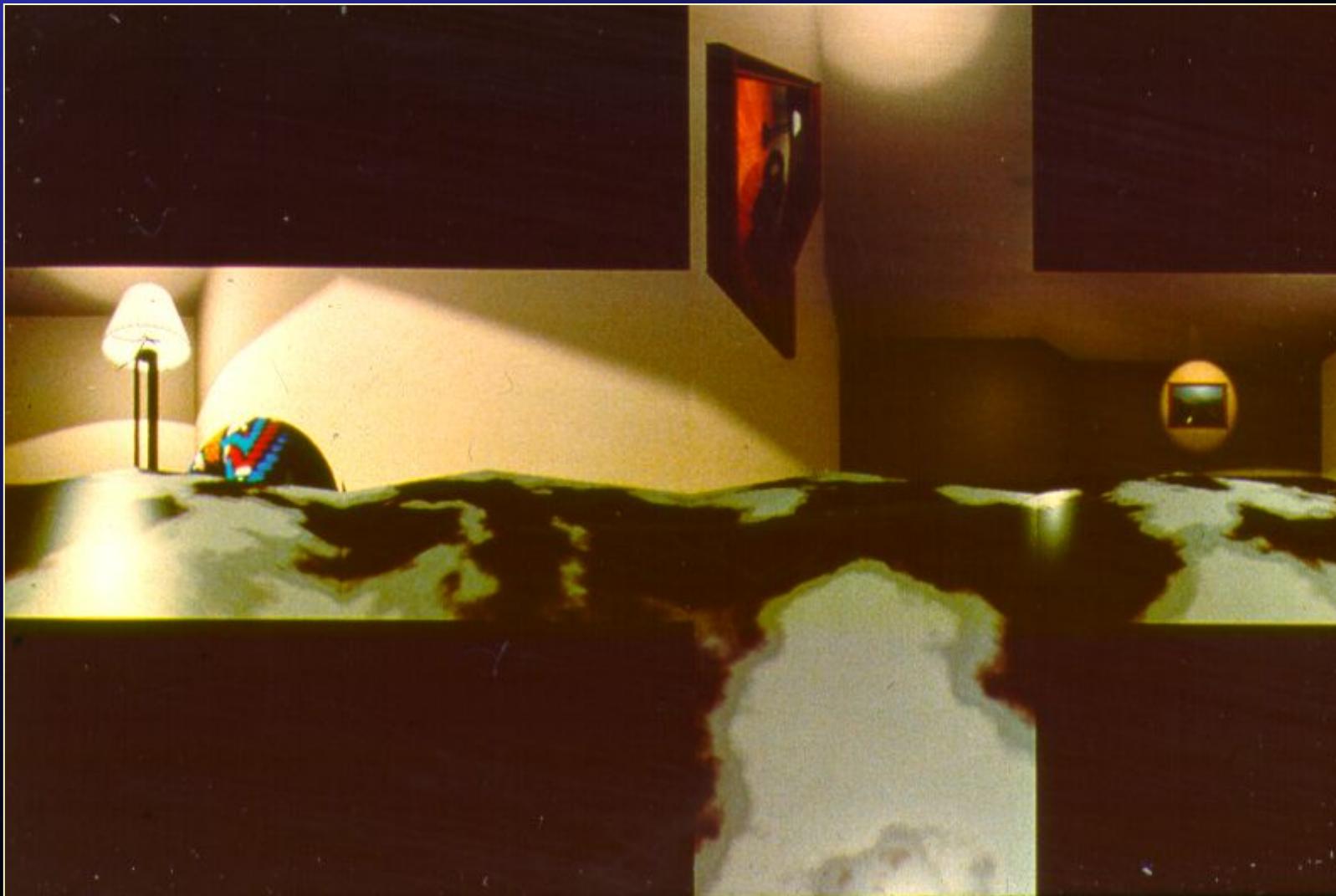
Can Illuminate Objects with Measurements of
Actual Lighting (Details later)



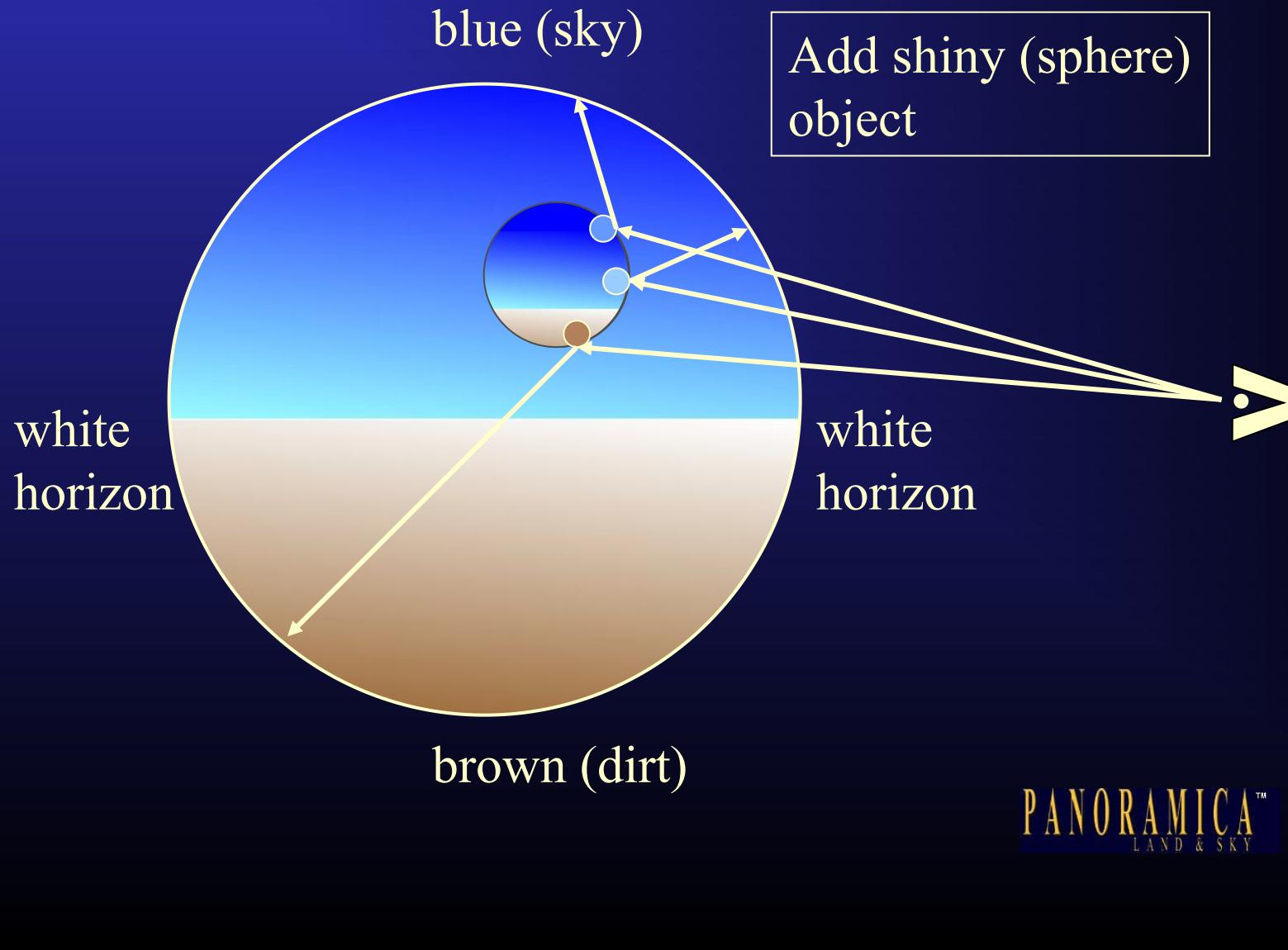
Debevec, IEEE CG&A 2002



World Map for the Room Scene (6 cube surfaces)

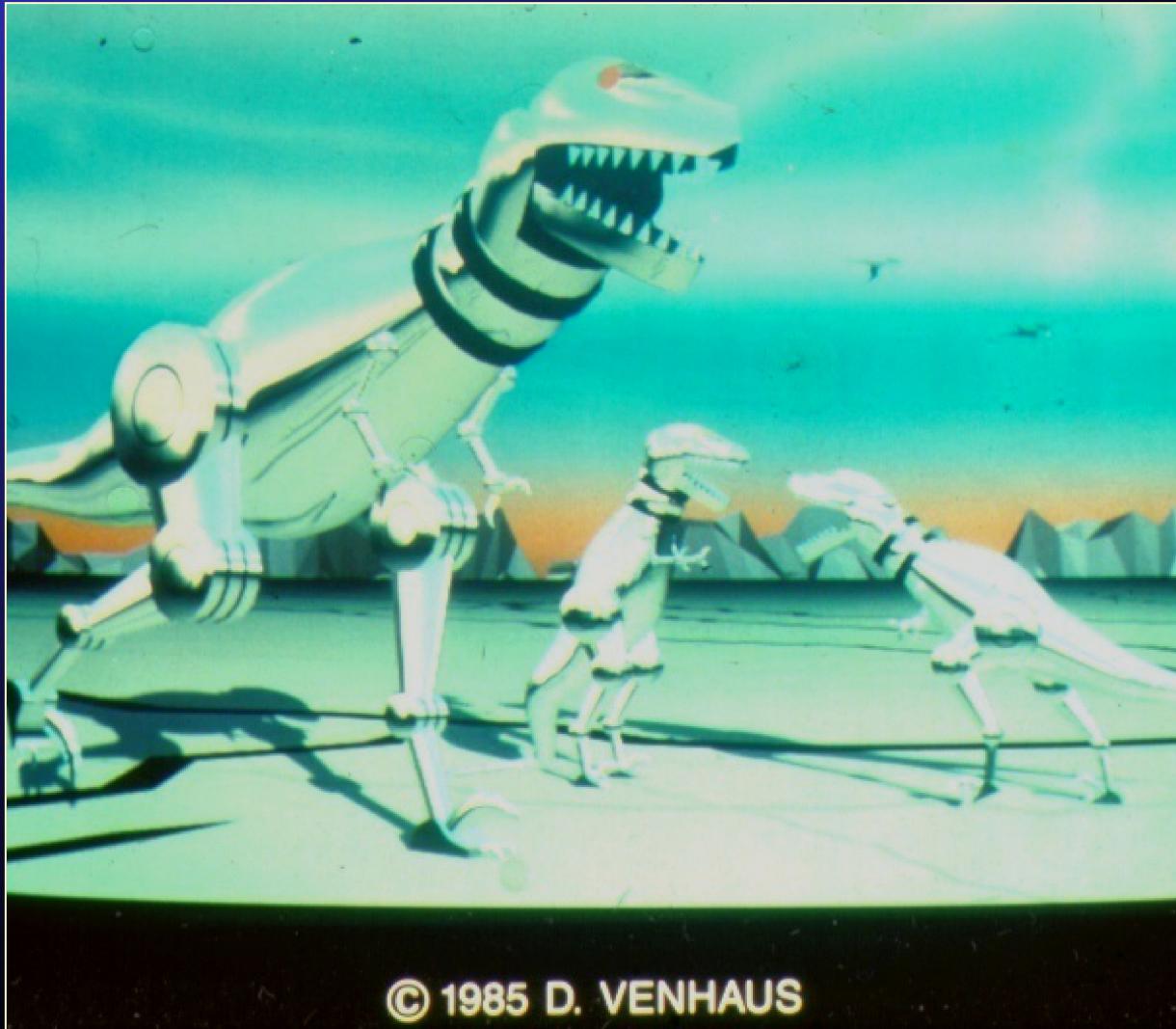


World Map as a Spherical Background Map



PANORAMICA™
LAND & SKY

Chromosaurus



© 1985 D. VENHAUS

Environment Mapping vs. Ray Tracing



Ren Ng

725

“Flight of the Navigator”



Shade Computation

Models:

- Diffuse or Lambert
- Vertex color interpolation and Gouraud
- Perfect specular
- Phong: Glossy specular
- Recursive ray tracing
- Microfacet
- Empirical

Recall the Three Reflectance Terms

- Diffuse: $I_d = I_i(L \cdot N)$

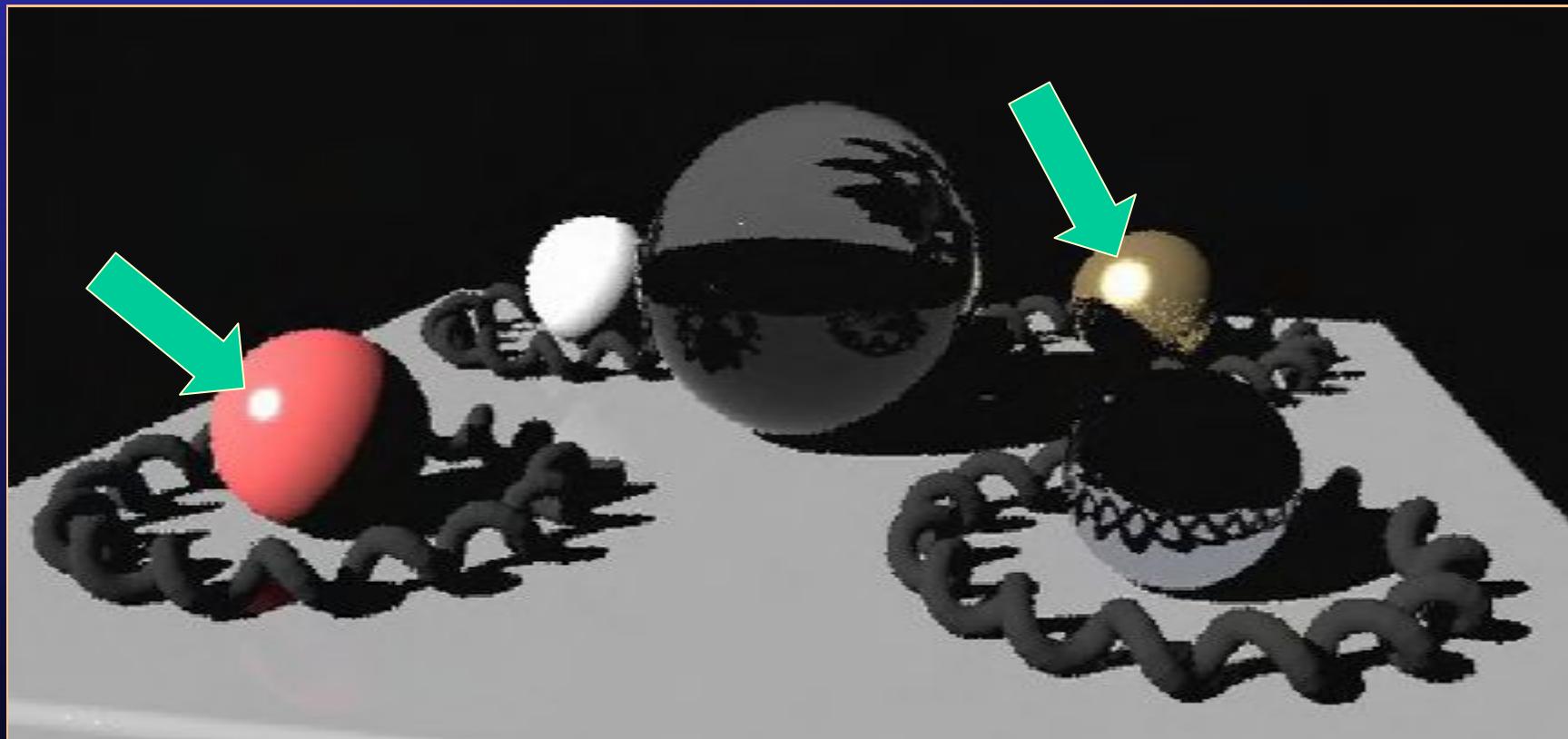
Where I_i is the intensity of the incident (incoming) light.

- Specular: $I_s = I_i(R \cdot V)^n$

- Ambient: I_a = “omnidirectional” light factor

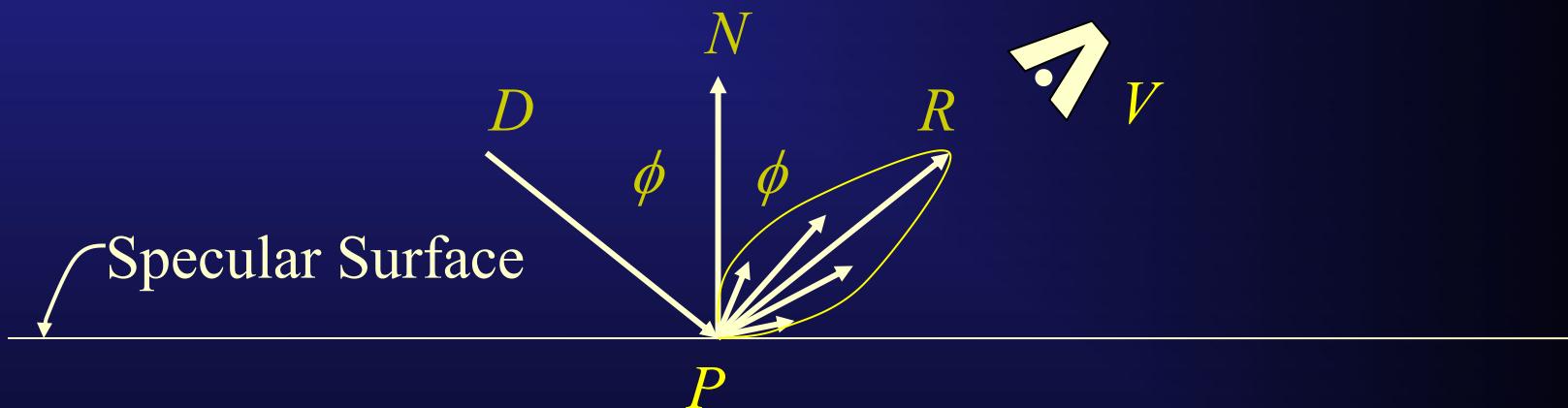
Now we can motivate the specular term.

Note Specular “Shiny” Spots from Point Light Source



Specular Highlights

- Intermediate between diffuse and mirror.



If $V = R$ then eye sees a **highlight** at surface point P .
There is a bright region about P whose size depends on
the amount of **specular** reflection.

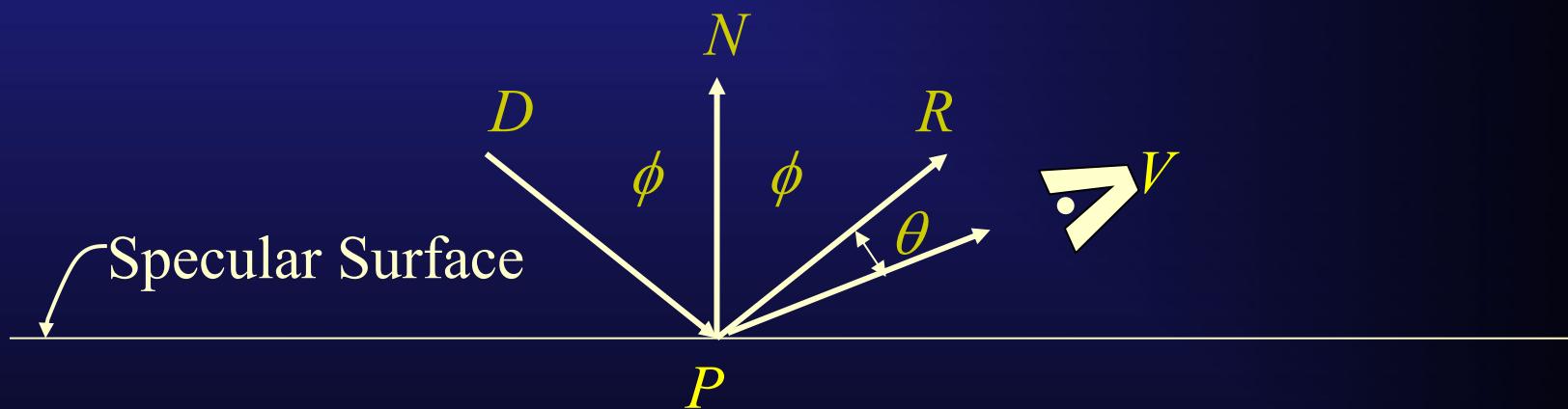
Approximating Specularity with Cosine Powers

- Blinn and Phong estimated the specularity of a surface as

$$(R \cdot V)^n = \cos^n \theta$$

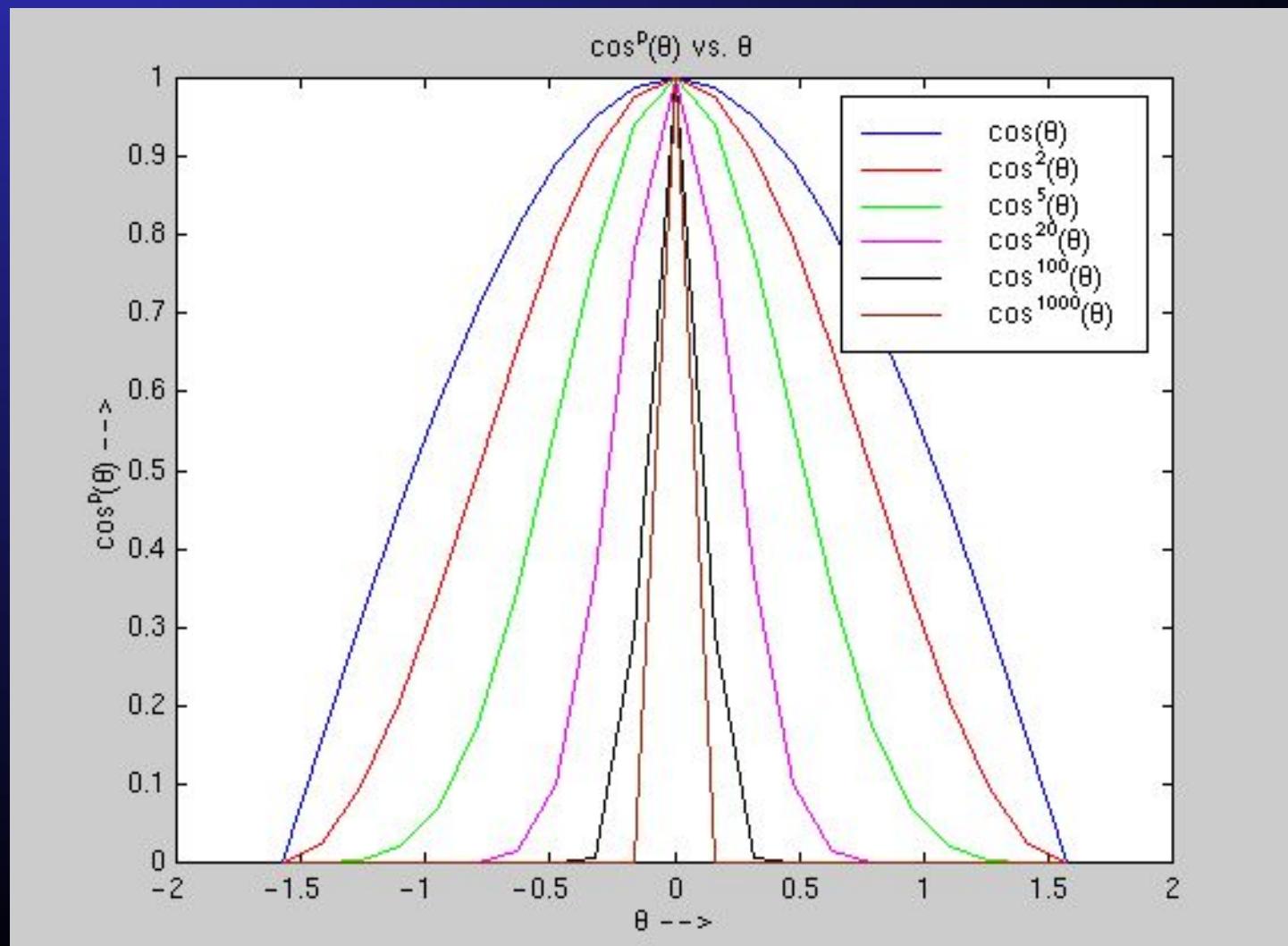
where n approximates the amount of mirror reflectance.

- $n = 1 \Rightarrow$ diffuse; n large (> 100) \Rightarrow glossy (shiny)



- As n approaches infinity, the cosine curve converges to the impulse function which is zero everywhere except at $\theta = 0$ where it is 1:

$\cos^n \theta$



Light Reflection from a Surface at a Point

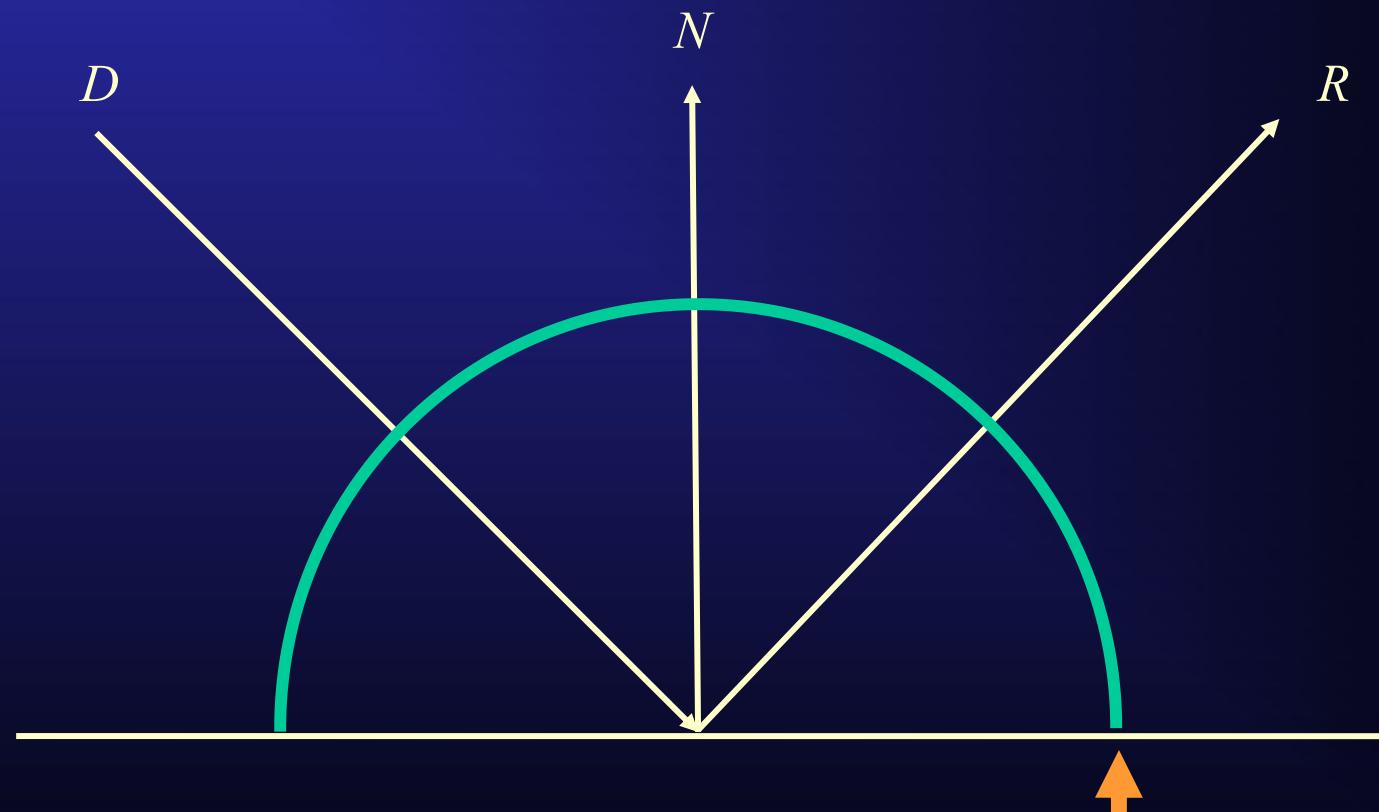
So we have examined two terms of this formula so far

$$\begin{aligned} I_{\text{reflected}} &= k_a I_a + \boxed{k_d I_d + k_s I_s} \\ &= k_a I_a + k_d I_i (L \cdot N) + k_s I_i (R \cdot V)^n \\ &= k_a I_a + I_i (k_d (L \cdot N) + k_s (R \cdot V)^n) \end{aligned}$$

Here's what this looks like diagrammatically:

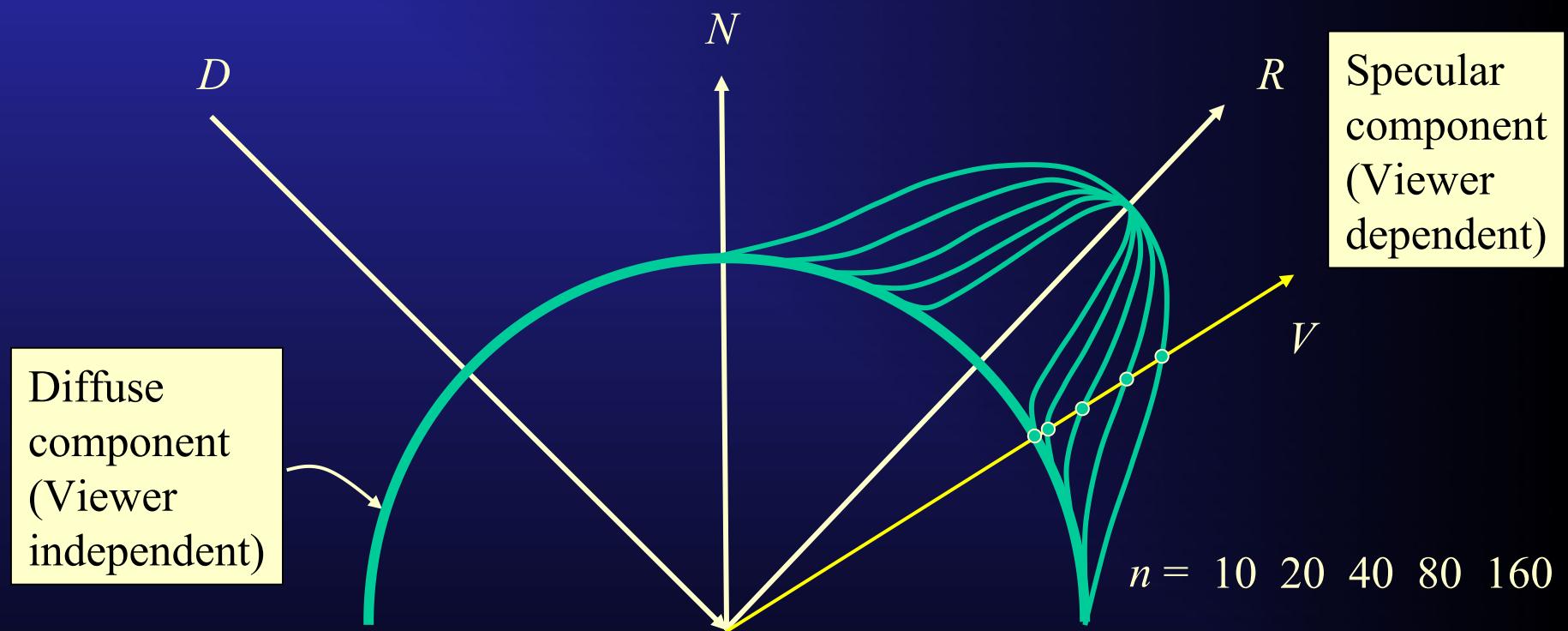
Diffuse Component (Viewer Independent)

High incidence angle (ϕ small, $\cos \phi \sim 1$) \Rightarrow large diffuse reflection
Low incidence angle (ϕ large, $\cos \phi \sim 0$) \Rightarrow small diffuse reflection



Magnitude of diffuse component

Specular “Bump” Adds More Reflected Light on Top of Diffuse Component



Adding Color into the Light Reflection Formula

$$\bar{C}_{\text{reflected}} = k_a \bar{C}_{\text{ambient}} \bar{C}_{\text{surface}} + \bar{C}_{\text{incident}} \left(k_d \bar{C}_{\text{surface}} (L \cdot N) + k_s (R \cdot V)^n \right)$$

For a colored surface s
(i.e., its “intrinsic” color):

$$\bar{C}_{\text{surface}} = (s_r, s_g, s_b)$$

The specular highlight takes
the color of the light source l :

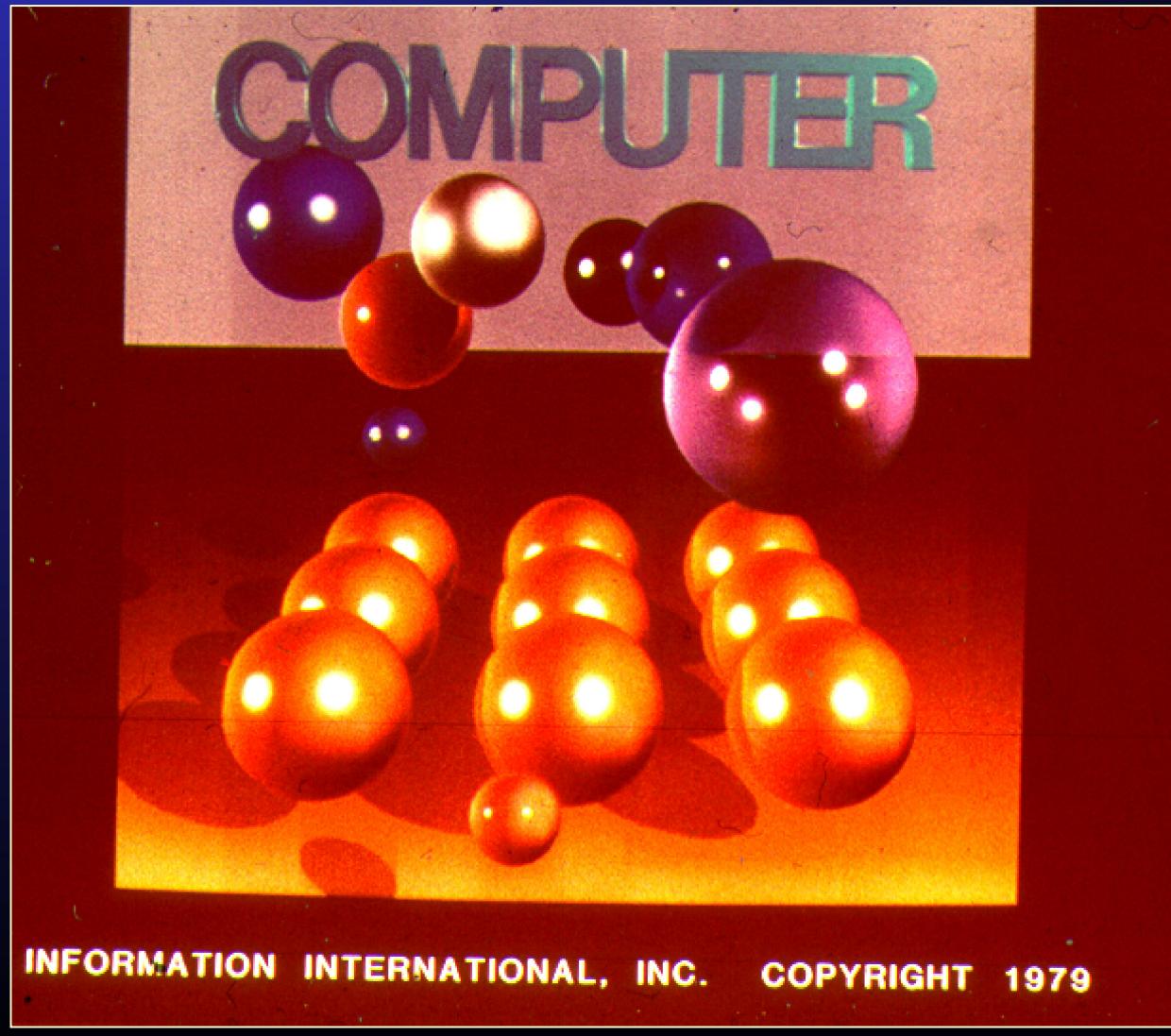
$$\bar{C}_{\text{incident}} = (l_r, l_g, l_b)$$

And the ambient light a may
also be colored:

$$\bar{C}_{\text{ambient}} = (a_r, a_g, a_b)$$

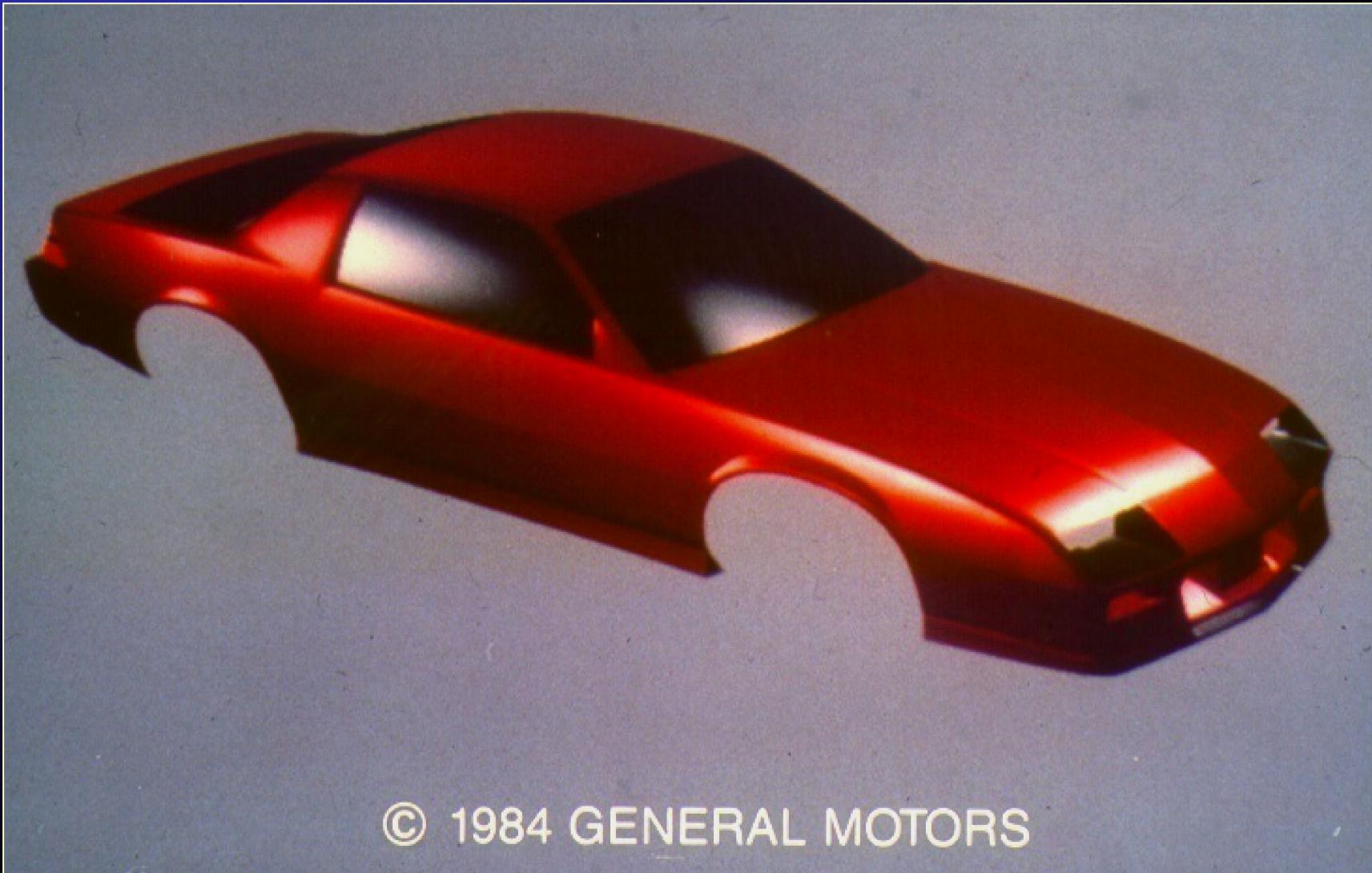
(Recall that colors are just
multiplied component-wise)

Phong Specular Spheres



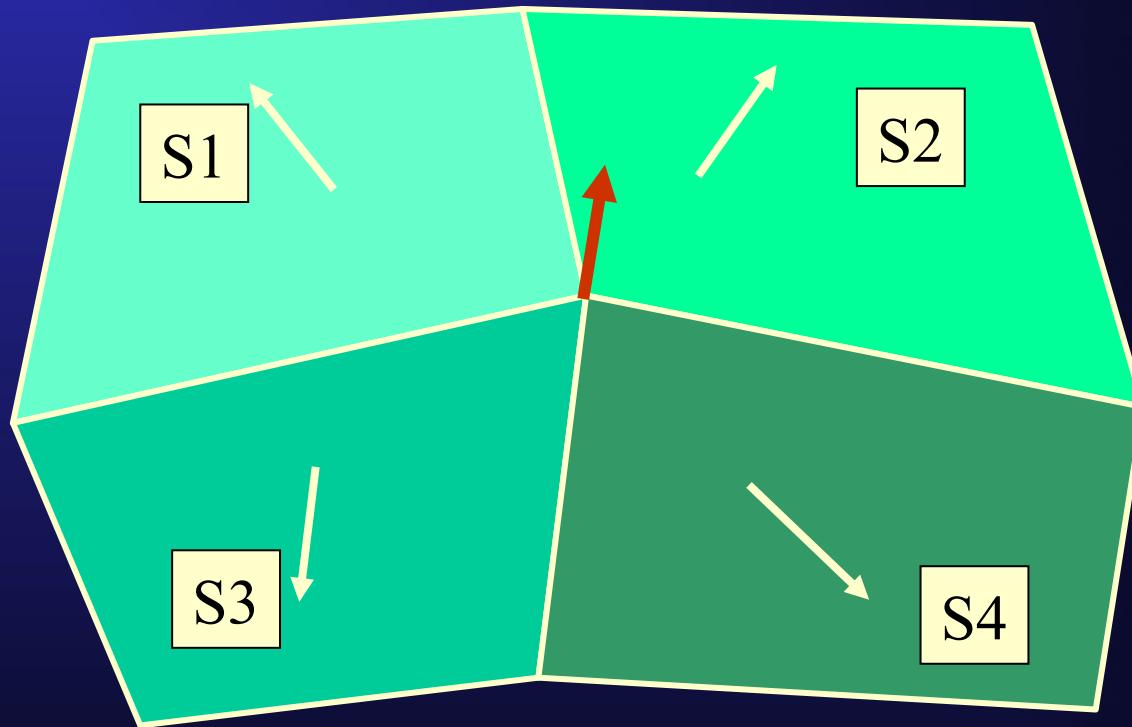
INFORMATION INTERNATIONAL, INC. COPYRIGHT 1979

Note that Specular Highlights take Color of Light Source -
Not Color of Surface



© 1984 GENERAL MOTORS

Phong's Great Idea: Use Vertex Normals and then Interpolate Them



↑ = **Vertex Normal** = average of neighboring face normals.
The average may be unweighted or weighted by adjacent face angle or area. (Or the normal can just be given any nonzero value!)

Interpolating Normals

- We must be sure to end up with a unit length normal vector.
- For the general weighted average of normals N_a and N_b :

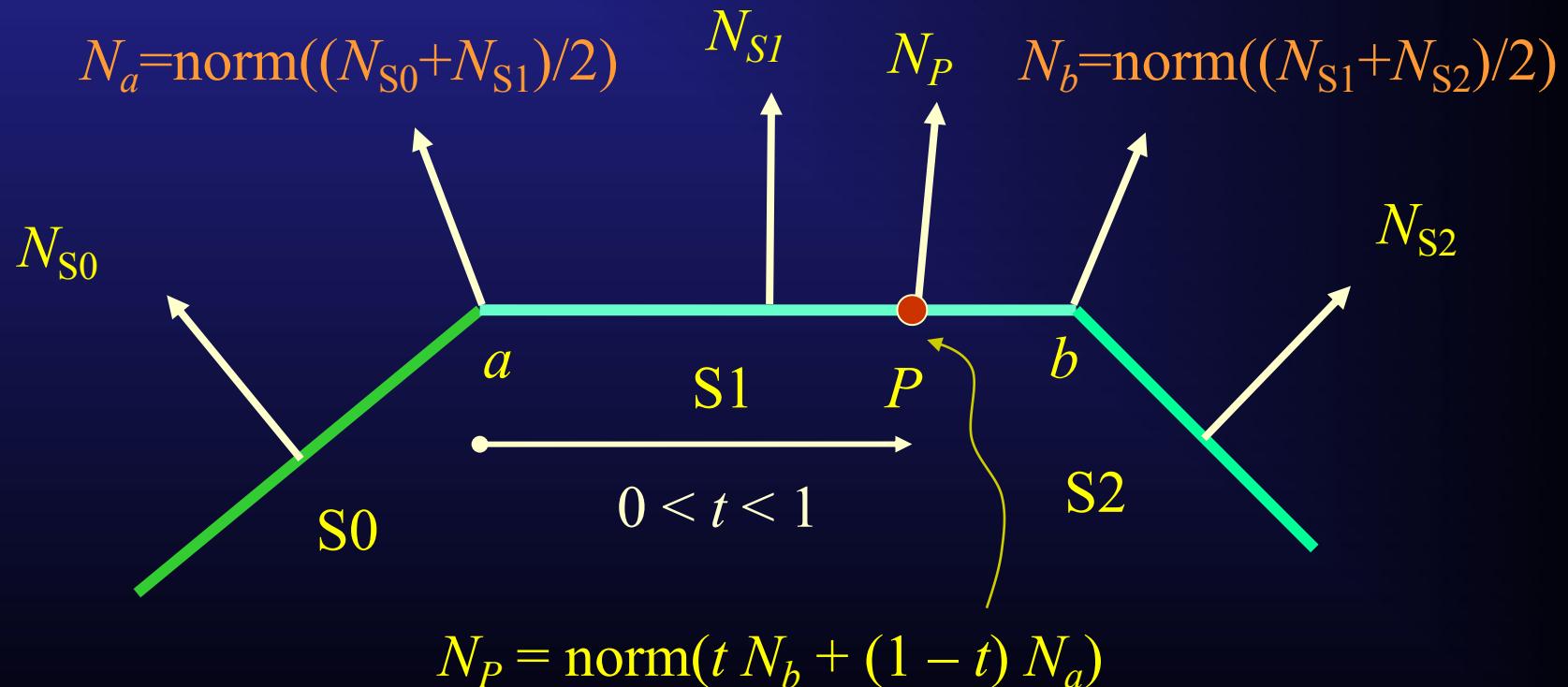
$$D = t N_b + (1 - t) N_a$$

$$N_{\text{interpolated}} = D / \|D\| = \text{norm}(D)$$

provided $D \neq (0,0,0)$

PHONG Shading

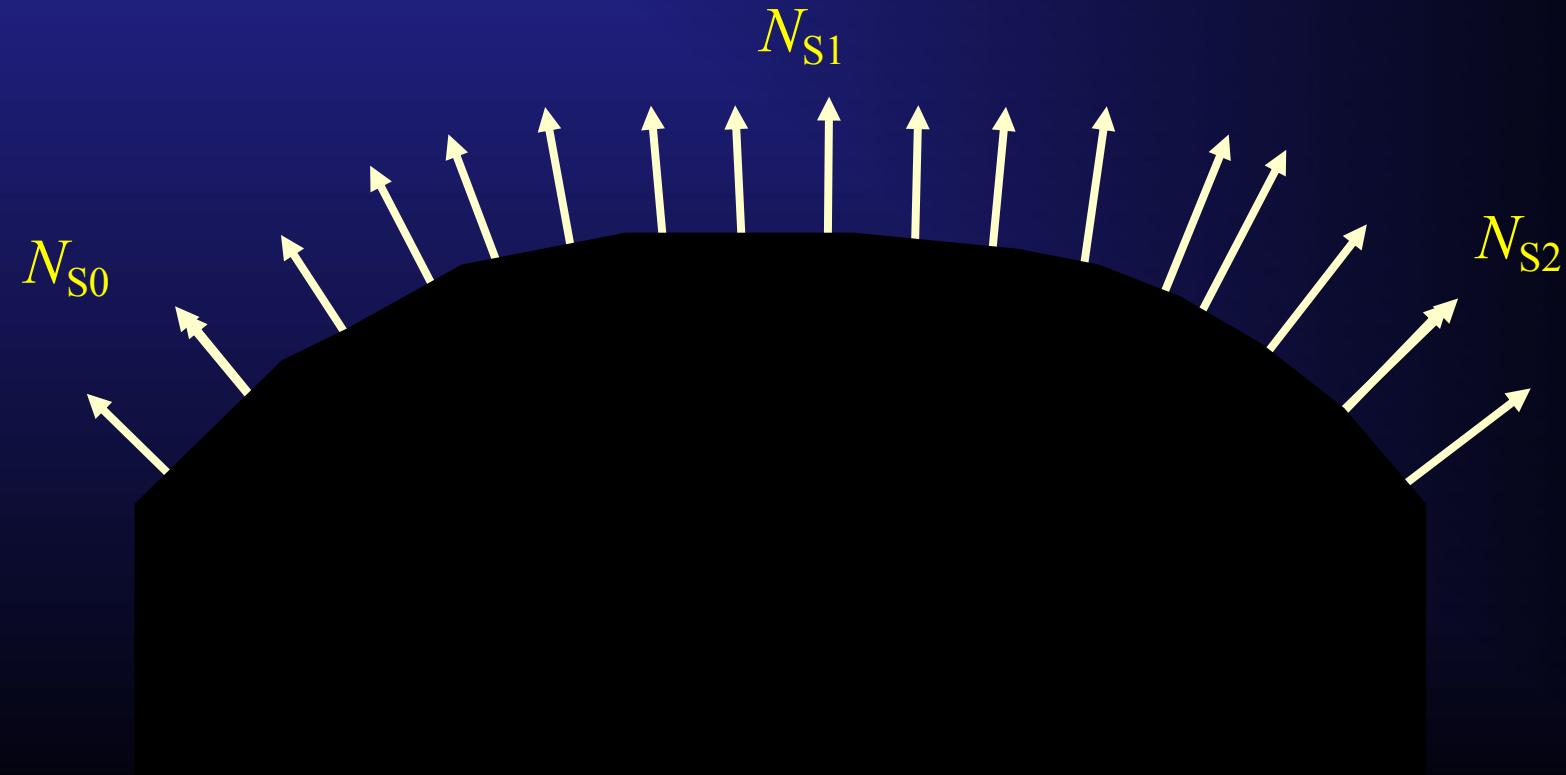
First compute vertex normals at a and b , then interpolated normals, e.g. at P , then finally compute shade from N_p .



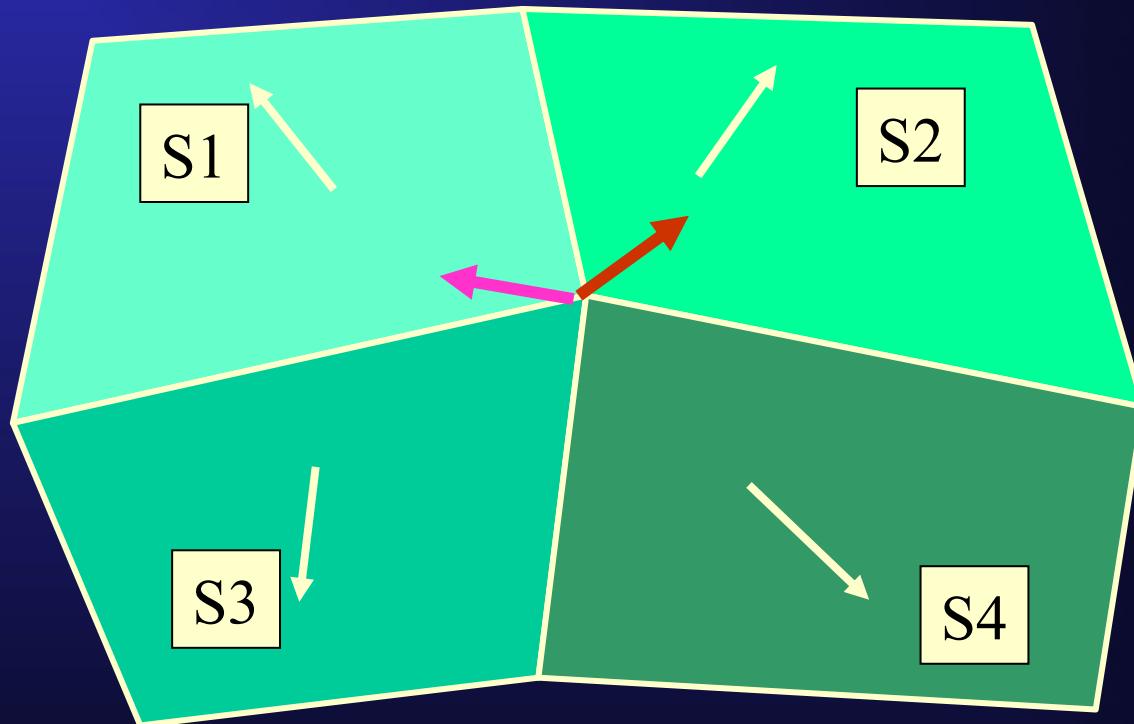
PHONG Shading

Do this everywhere on scanline that a normal is needed:

Gives the **illusion** of a smoothly rounded object surface!



Interpolating Normals: Preserving a hard edge



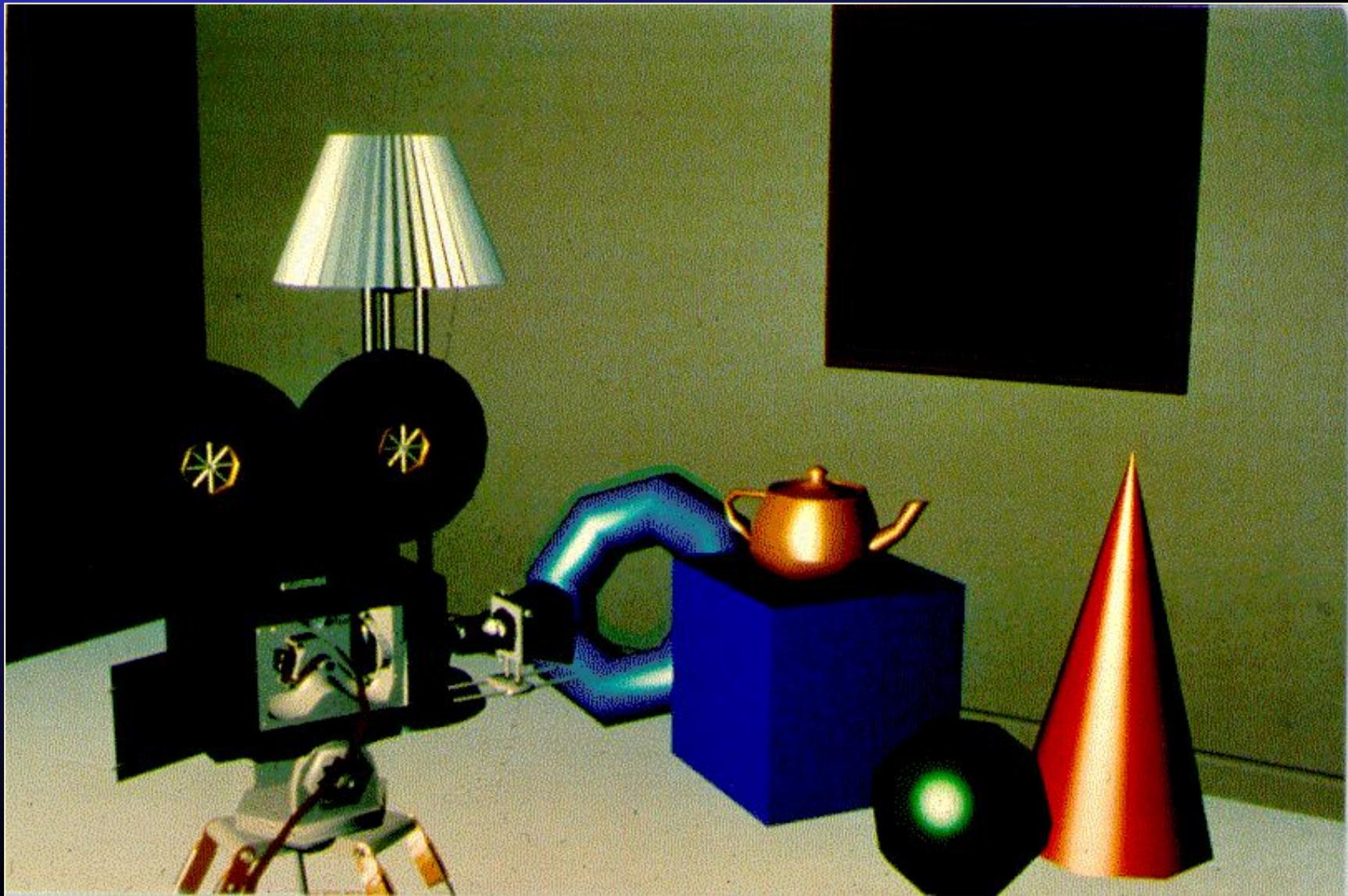
Vertex normals stored per polygon vertex!

Take two separate averages, e.g.:

the left set: ← and the right set: →

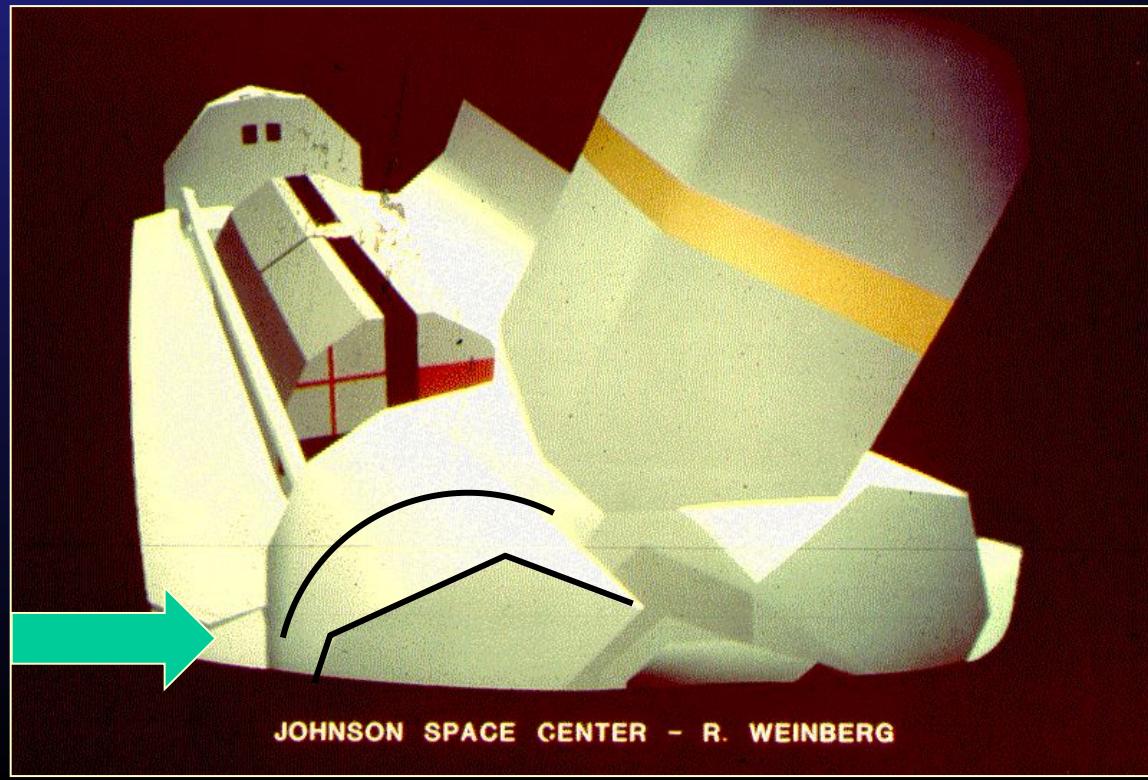
S1 - S3 edge will be smoothed; S2 - S4 edge will be smoothed.

Phong Shaded Polygons (Note silhouette edges!)



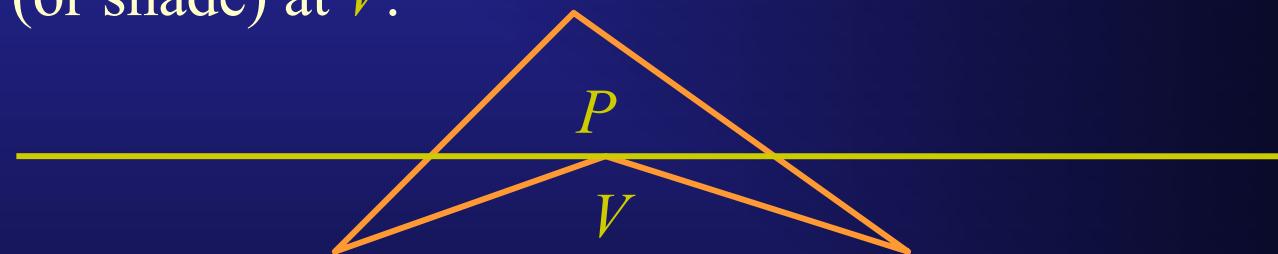
Polygon Edge Problem

- Neither Phong nor Gouraud shading can possibly change the underlying polygon model.
- Therefore silhouette edges appear linear even if shading is smooth!

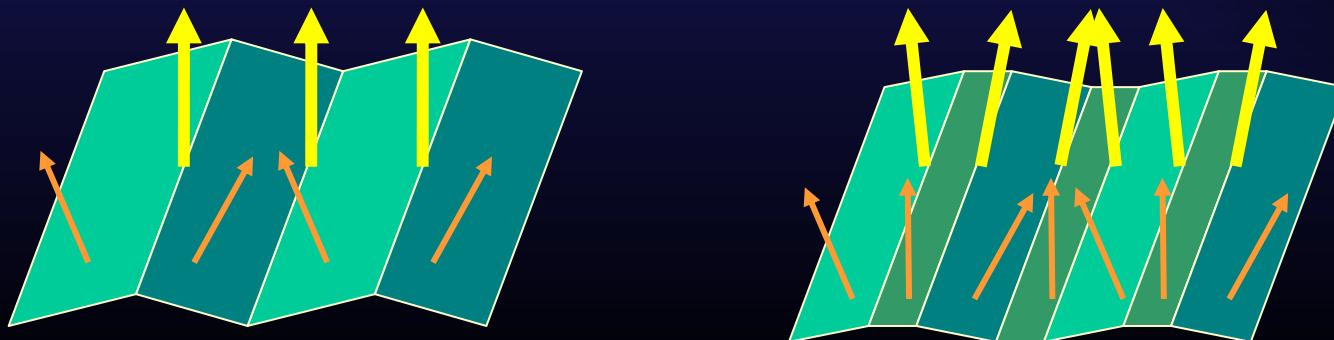


Problems with Normal Vector Interpolation

- Non-convex faces can result in non-unique interpolants -- Normal (or shade) at P may have no relationship to normal (or shade) at V :



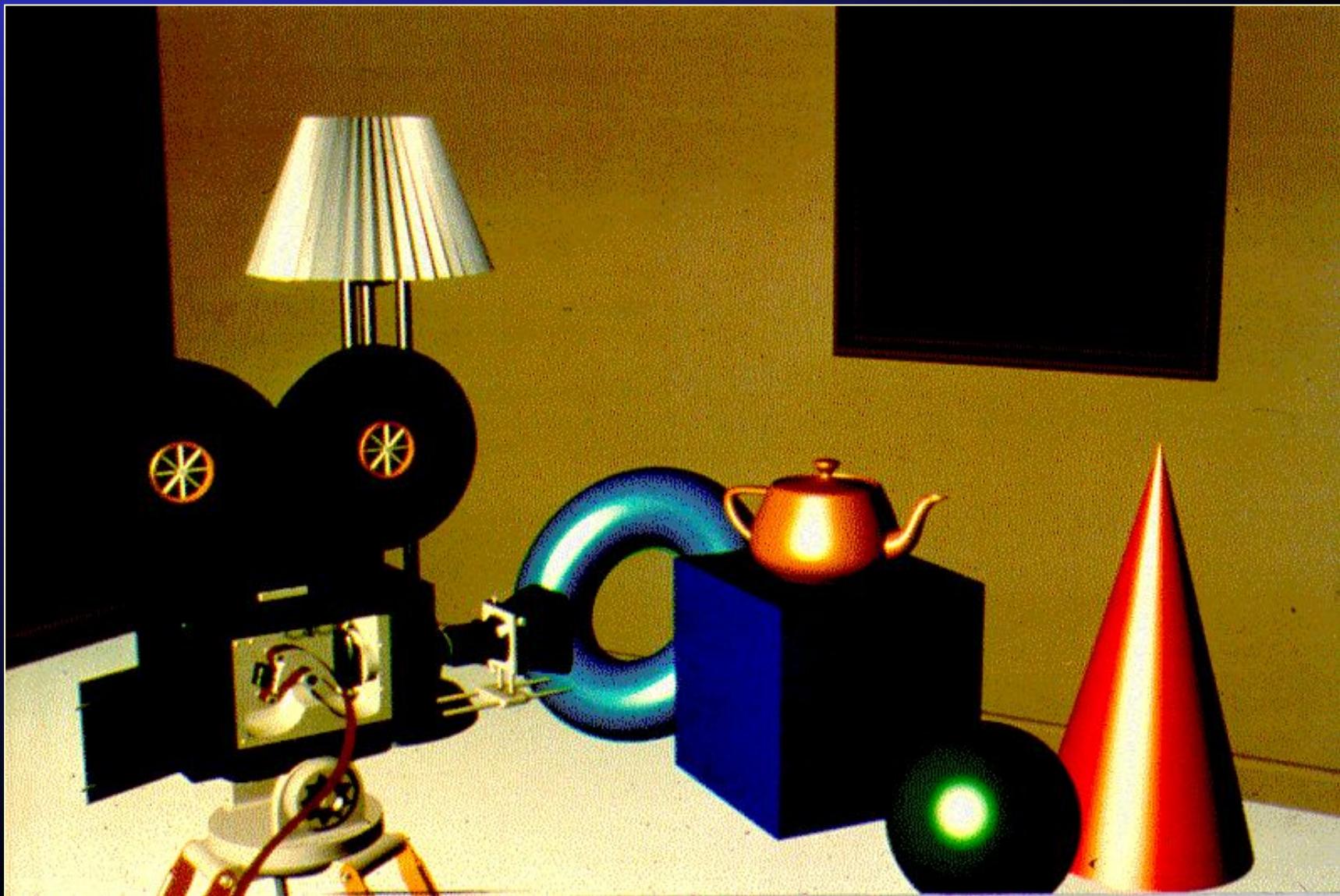
- Interpolation may mask regular shape changes -- so may have to change underlying polygon model:



To Get Smooth Edges as Well as Surfaces

- To get a smooth appearance throughout the model, we need to have a **smooth model**. A curved surface representation will do.
- For example, if appropriate scene polygons are interpreted as a **control mesh** for curved patches, the results are much better...

Polygons Interpreted as Curved Surface Control Mesh



Multiple Light Sources, Curved Surfaces, and Phong Shading



The LOCAL Illumination Formula

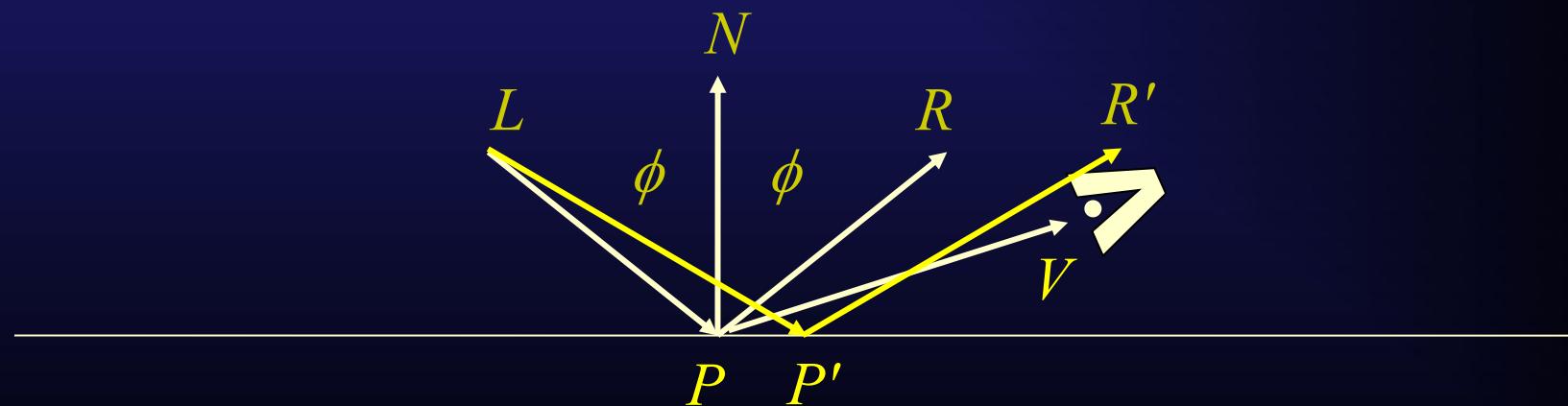
$$\begin{aligned}\bar{C}_{\text{reflected}} &= \\ k_{\text{ambient}} \bar{C}_{\text{ambient}} \bar{C}_{\text{surface}} &+ \\ \sum_{j=1}^m \delta_{L_j} \bar{C}_{L_j} \left[k_{\text{diffuse}} C_{\text{surface}} (L_j \cdot N) + k_{\text{specular}} (R_j \cdot V)^n \right]\end{aligned}$$

where $\delta_{L_j} = 1$ if light L_j is visible, otherwise $= 0$.

Color (r, g, b) reflected at the visible point =
ambient light + (for each light source that is not occluded)
light color \times [diffuse component \times surface color +
specular reflection component]

Evaluating the Local Illumination Formula

- This model is **LOCAL**: it depends only on the normal N , the eye position V and surface reflectance properties (color, glossiness) at a point P .
- Also, note that L_j and R_j may vary over a surface, especially if L_j is close to the surface: to avoid this, often lights are placed infinitely far away so that their *direction* L_j is constant. (Good game hack.)



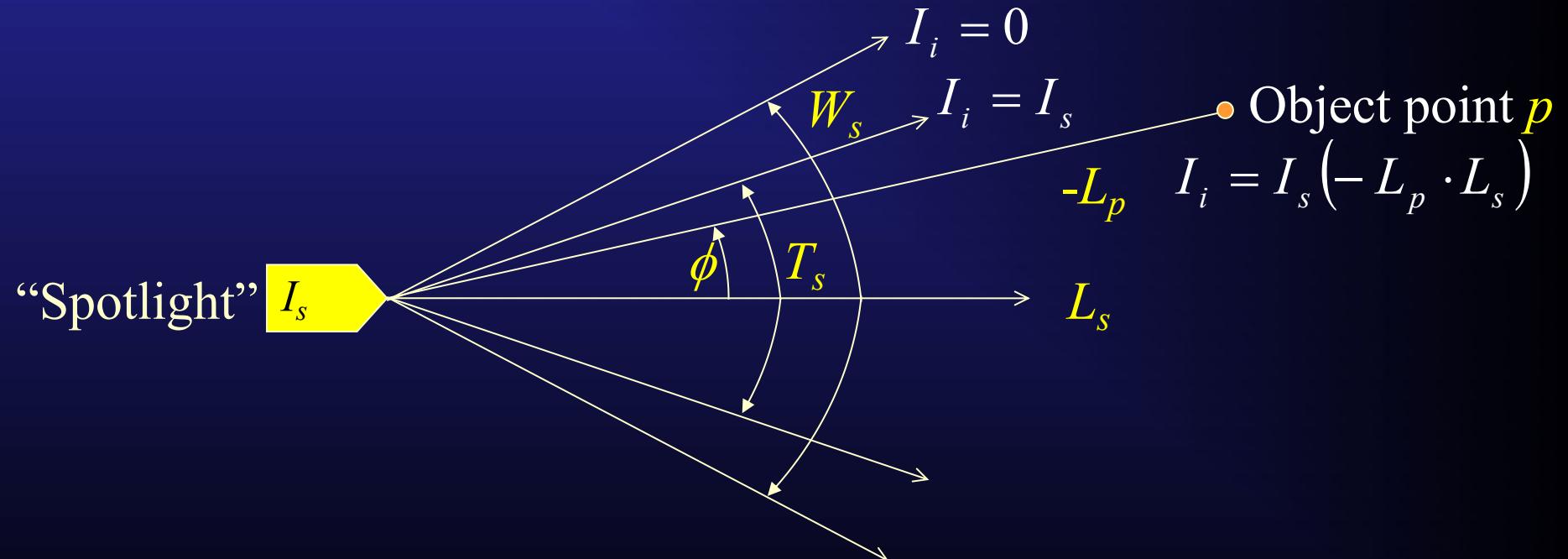
- Finally, note that N will vary over a curved surface (at every P).

Light Source Types

- Points (saw this already)
- Infinite area (environment) maps (saw this already, too, but only for mirrored surface reflections)
- Spotlights
- Texture projection
- Goniophotometric diagrams
- Area lights

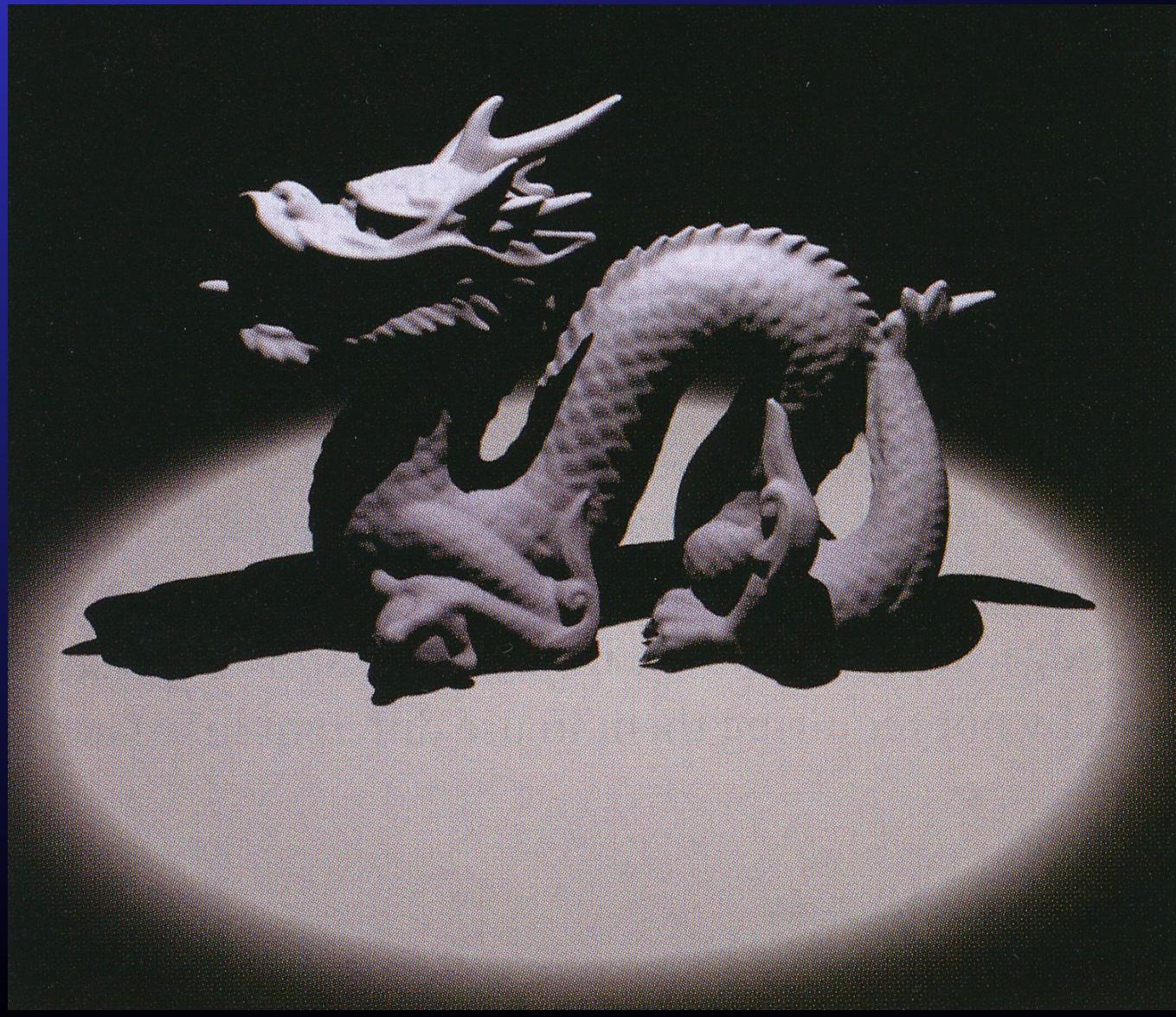
Light Intensity Spotlight Model

- Use direction L_s , fall-off threshold T_s and light half width W_s
- Light intensity I_s constant for $\phi < T_s$; 0 for $\phi > W_s$; and linearly interpolated between T_s and W_s



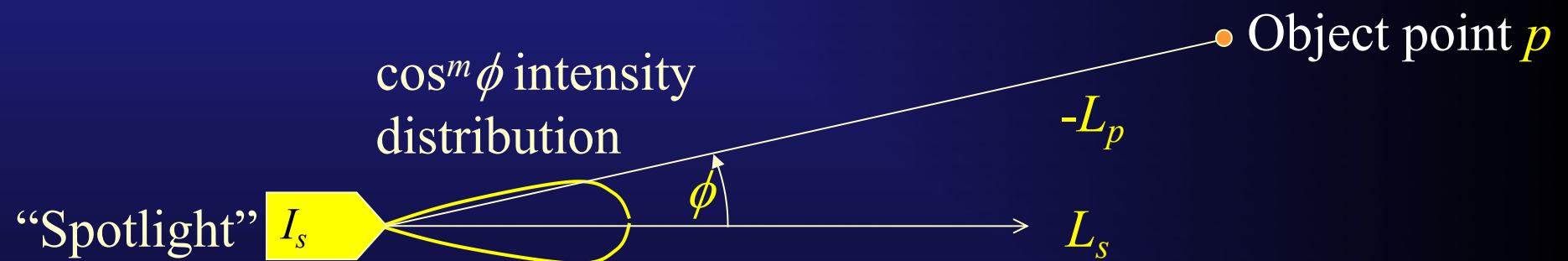
This implies the light vector L_p is no longer constant for all points p .

Spotlight with Soft Edge Fall-Off



“Flashlight” Light Intensity Spotlight Model

- Or adapt the Phong model, so that light intensity = $I_s \cos^m \phi$



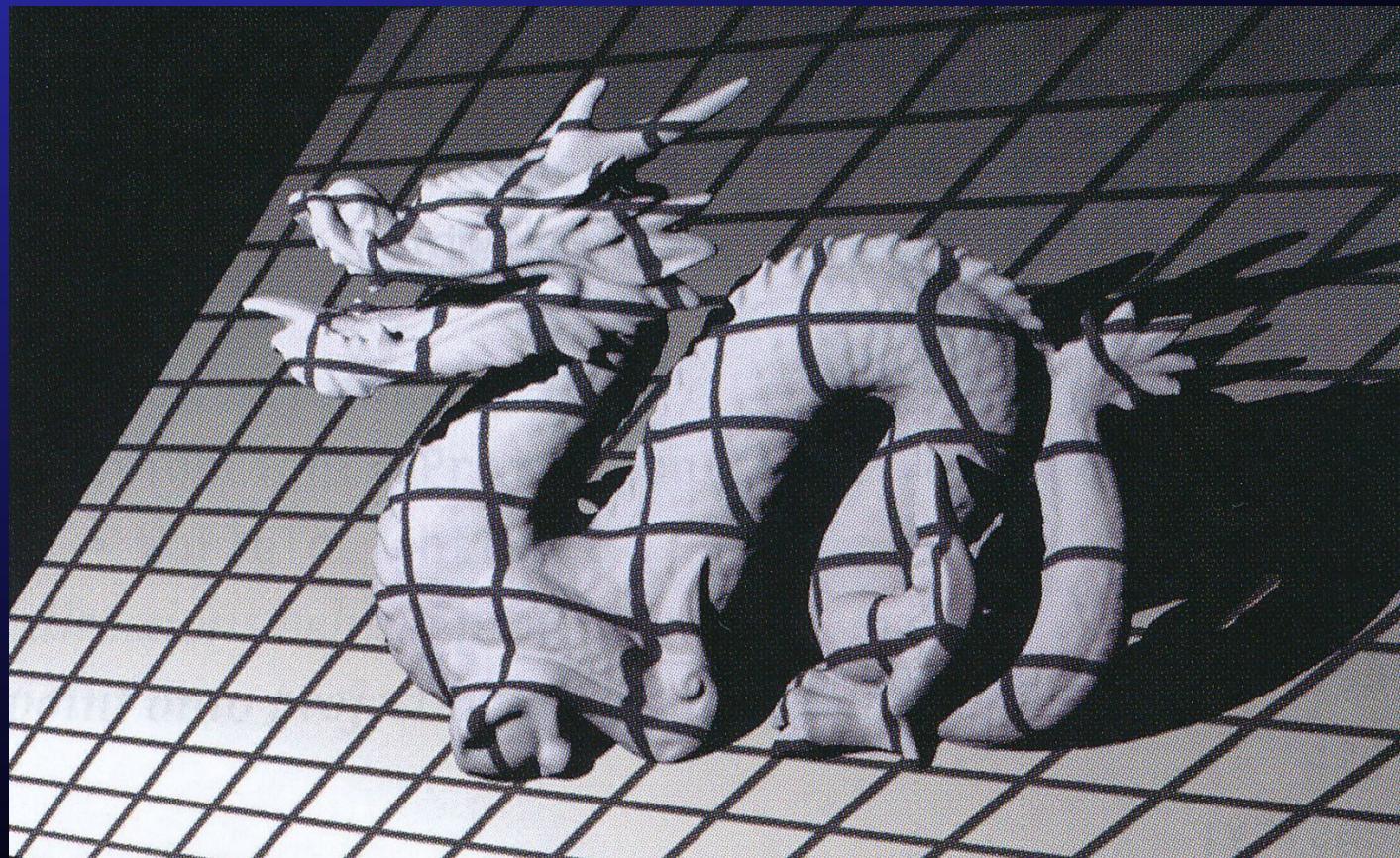
$$\text{Hence: } I_i = I_s (-L_p \cdot L_s)^m$$

Again, the light vector L_p varies over p .

Texture Projection

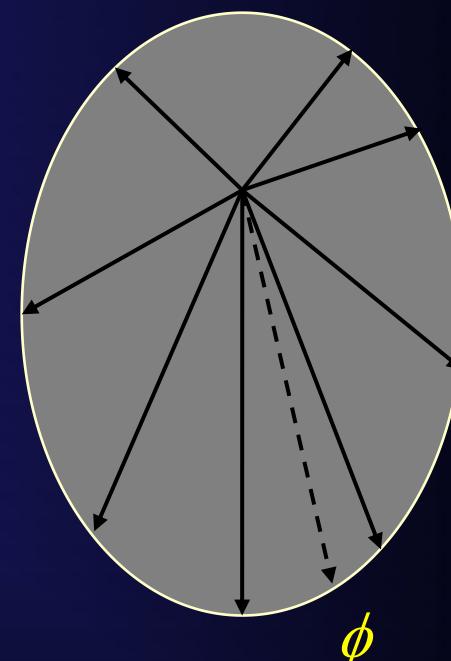
- Use simple ray-tracing projection to cast rays from a point light or a spotlight into a 2D map:
 - The ray passes through a 2D pixel (or computed function) that is either 0 or 1;
 - If 1 then the ray continues to illuminate a surface; if 0 it does not.
- Watch for boundaries and 2D map passing behind the light source.
- Can also use arbitrary image and cast suitable colored rays into scene, just like a real slide projector.

Projected Texture (Note Sharp Shadows)

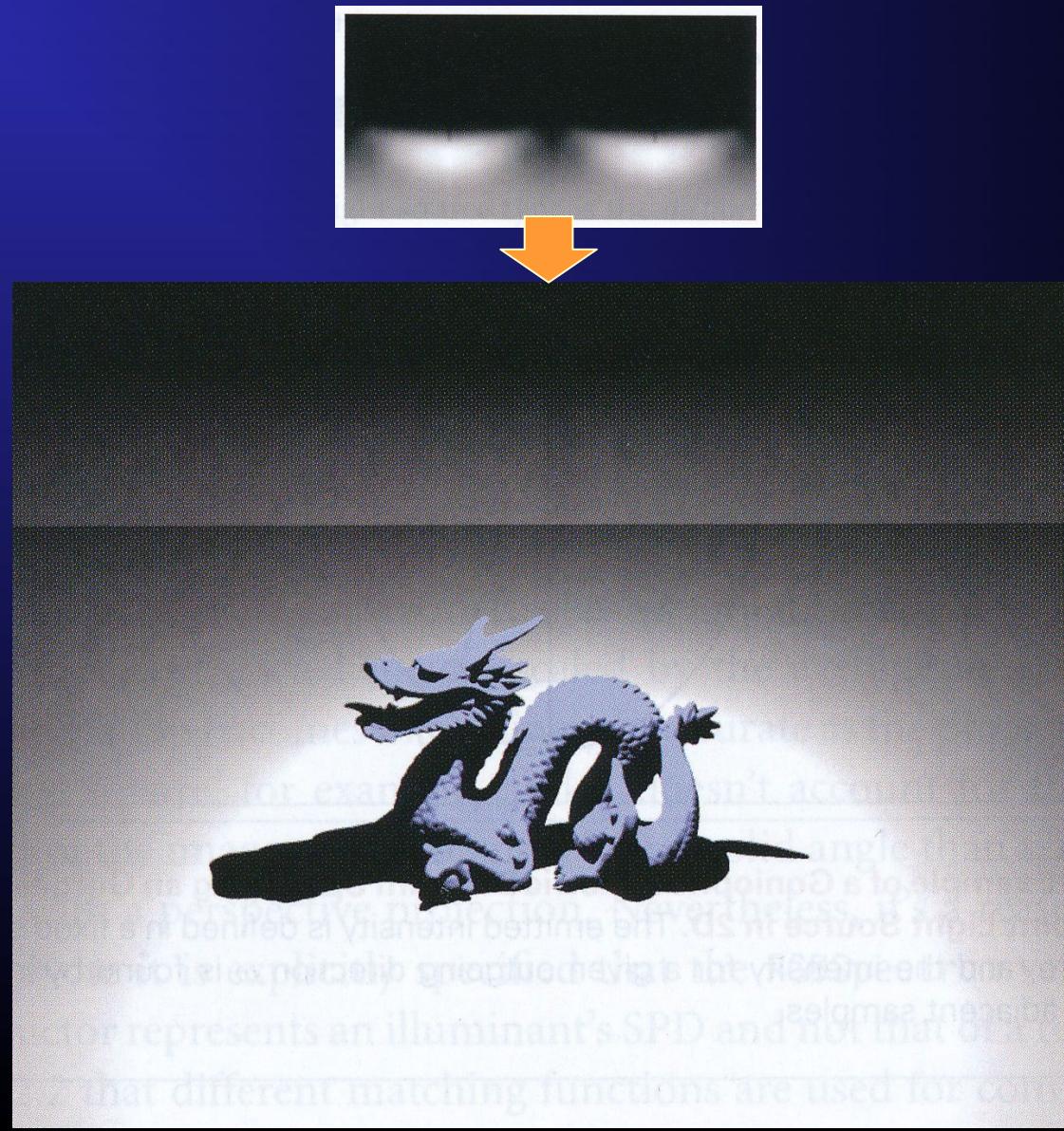


Goniophotometric Diagrams

- Goniophotometric diagrams are a general way of defining a light distribution in a spherical pattern from a point source.
- E.g., (see the 2D example \Rightarrow) sample values given in various directions.
- Light intensity in a specific direction ϕ is computed by angular interpolation or by a given function or from a spherical image (texture \downarrow).



Sample Goniophotometric Texture and a Resulting Image (Plus a Point Light)

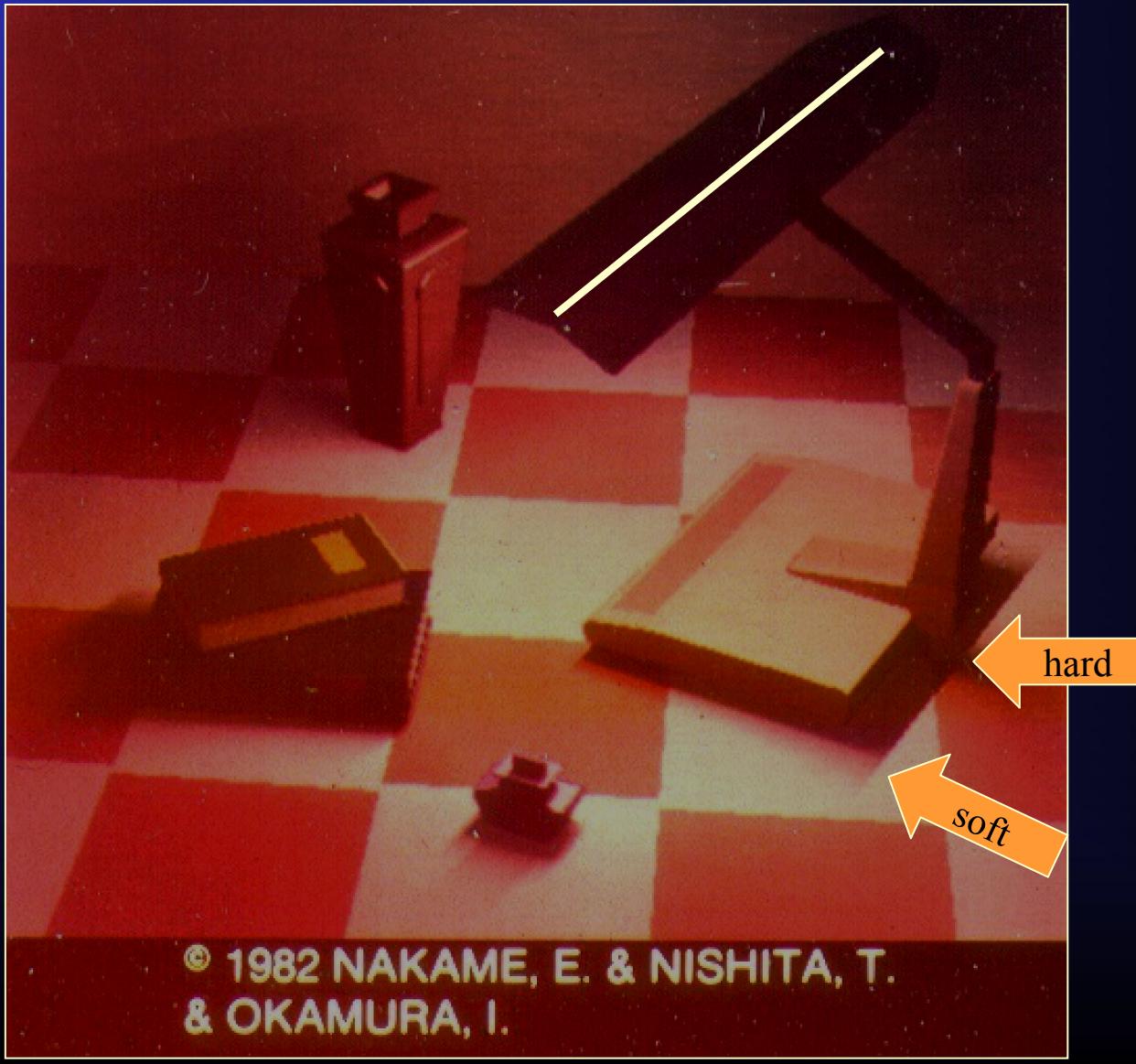


Light Fixtures

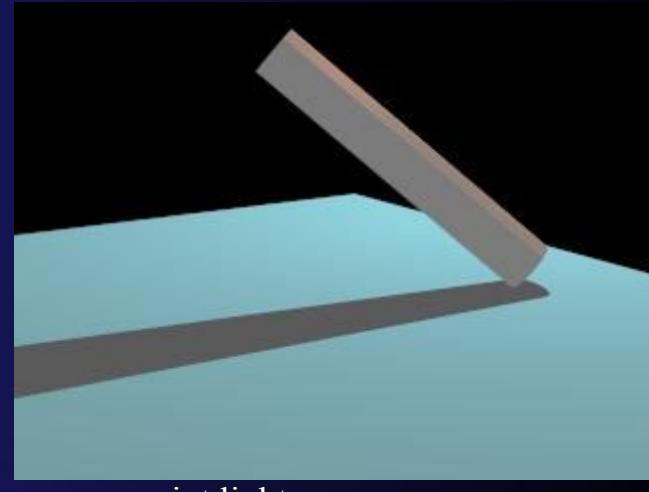
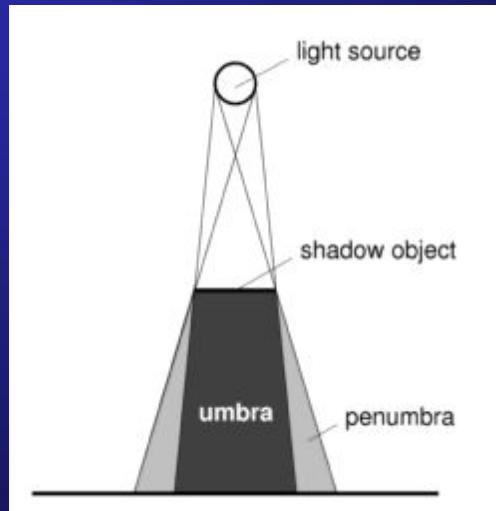


© 1987 LIGHTING TECHNOLOGIES

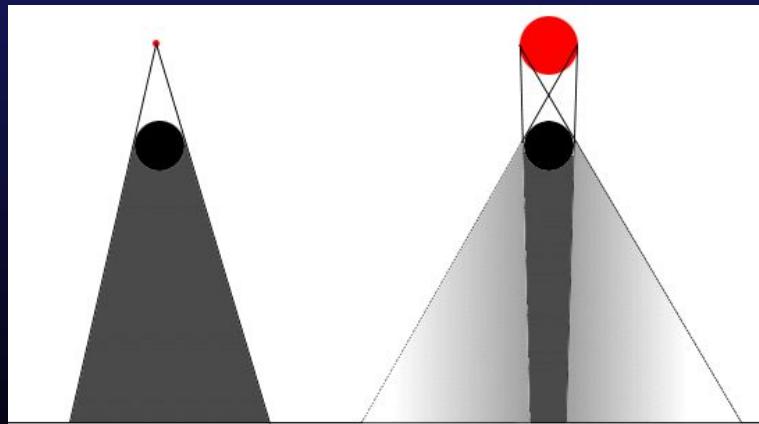
First Soft Shadows



Area Lights

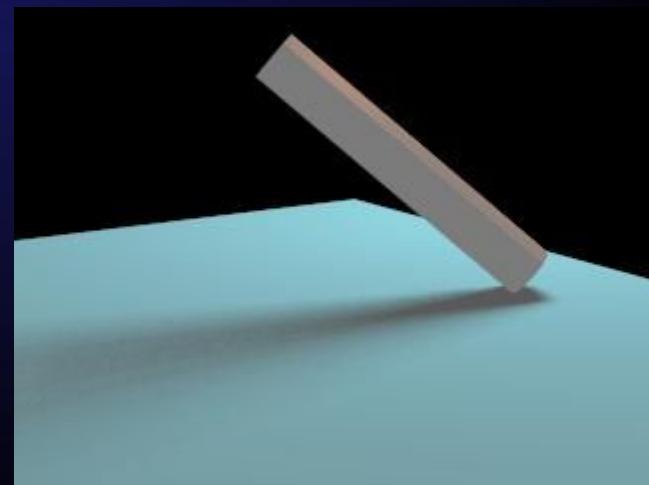


point light source



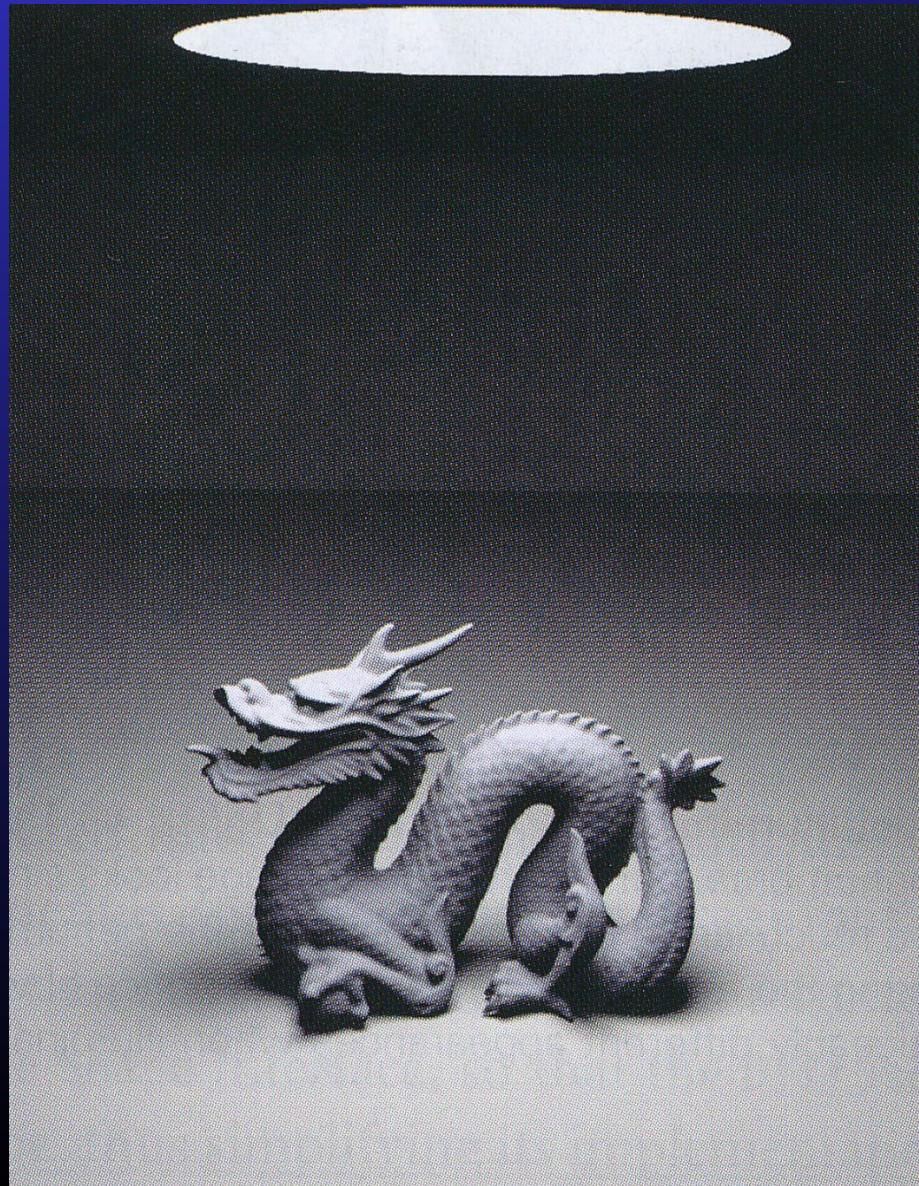
Point Source, sharp shadow

Area source, fuzzy shadow

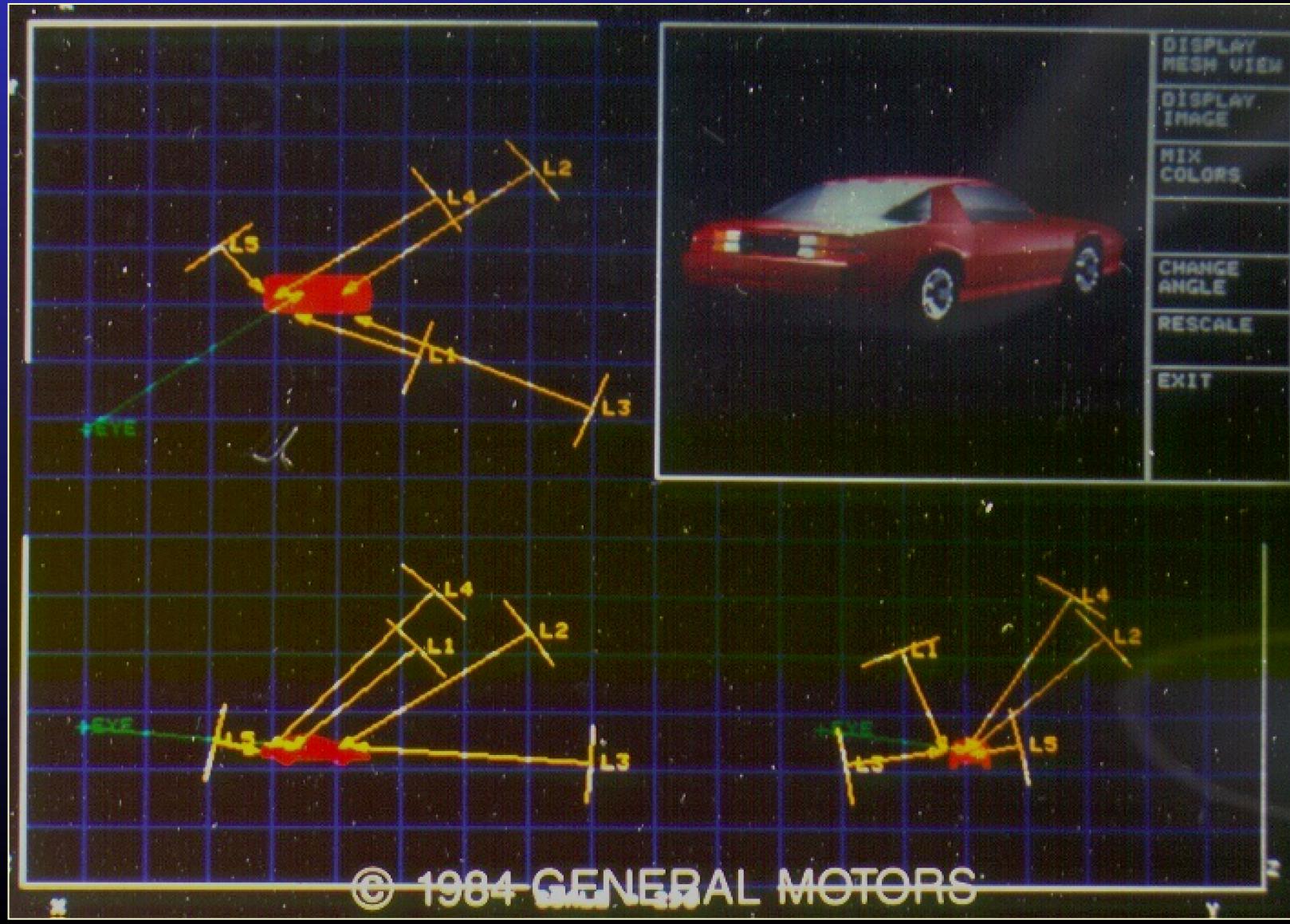


Area light source

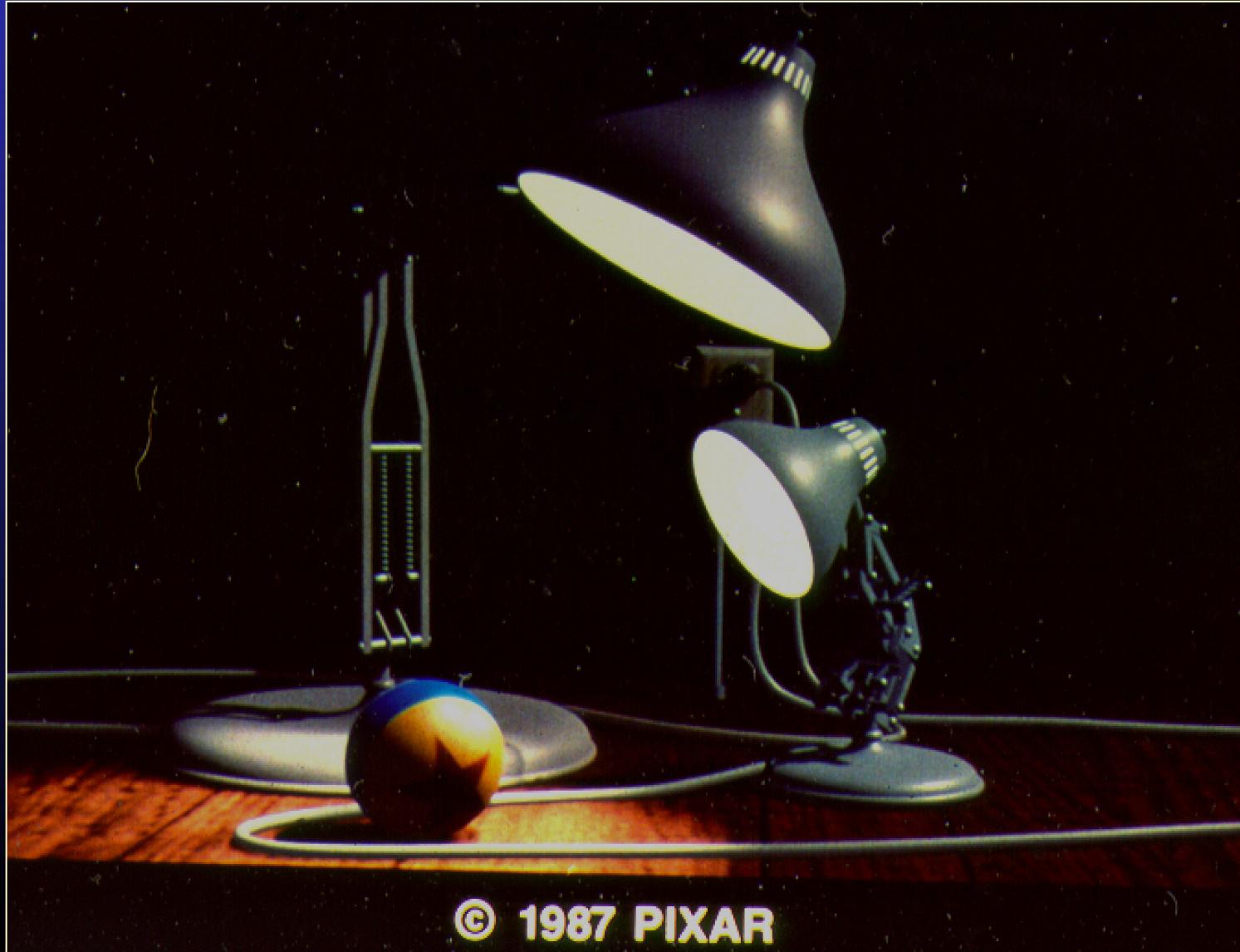
Disk Area Light (from above) with Soft Shadows (Large Disk – Small Disk)



Lighting Design: Synthetic Lights Enhance Surface Shapes



Lights as Actors (Actually in the Scene) (Why is the background so dark?)



© 1987 PIXAR

765

Shade Computation

Models:

- Diffuse or Lambert
- Vertex color interpolation and Gouraud
- Perfect specular
- Phong: Glossy specular
- RECURSIVE RAY TRACING
- Microfacet
- Empirical

Rendering Images of 3D Models Having Various Shading Attributes: Ray Tracing with “Local Illumination”

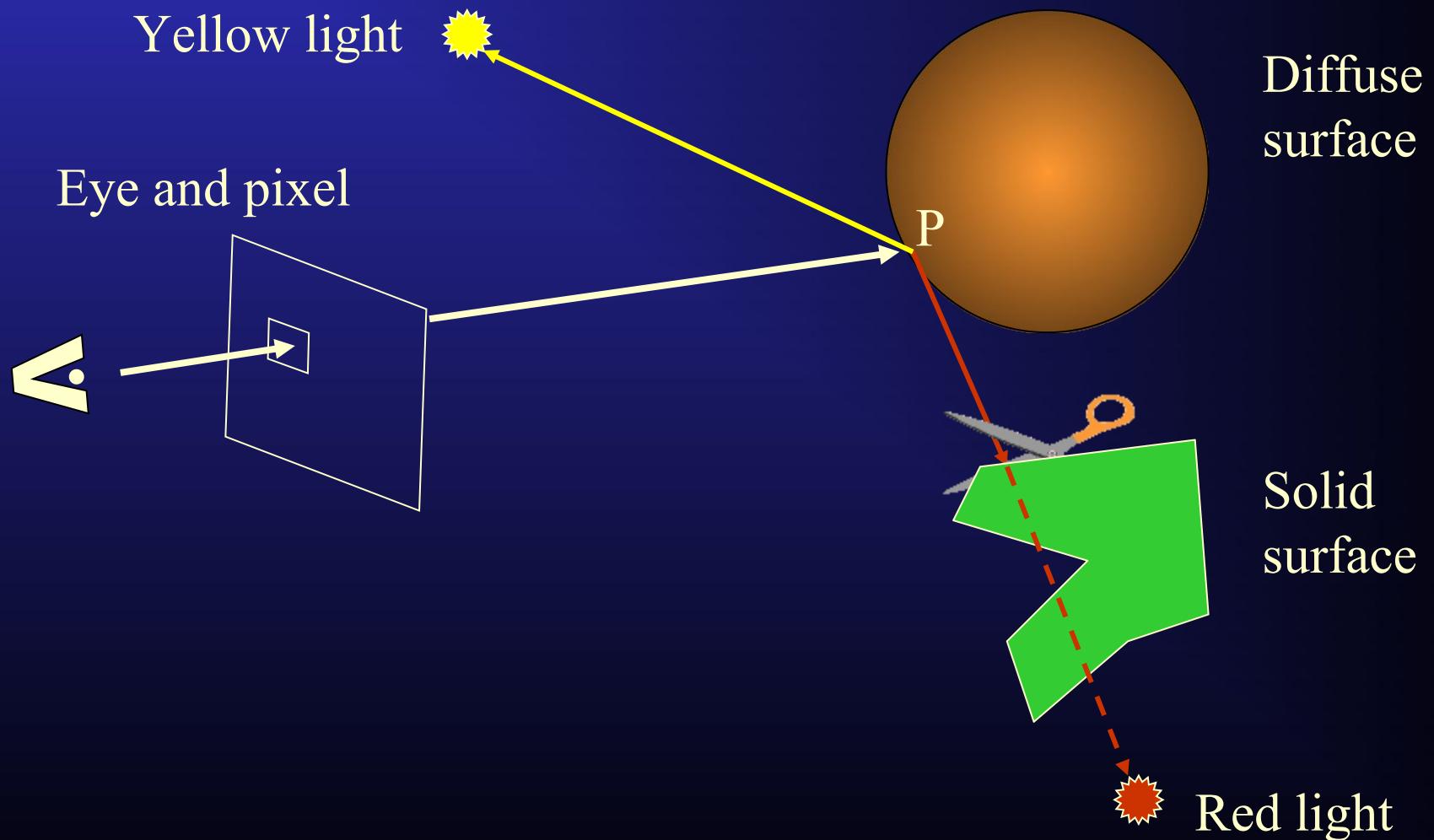


WHITTED, T.—BELL LABS

Overview of Ray Tracing

- Core algorithm is ray casting; just do it recursively.
- Start at eye point.
- Pass ray through pixel (center or corner).
- Trace rays in direction **opposite** of light flow.
- Recursively spawn rays at reflective and refractive surfaces.
- Trace shadow rays to light sources.
- Stop at diffuse surface, or when maximum number of recursive steps exceeded.
- Color at visible point is established by recursively combining the shades computed at the end of each ray -- which depends of course, on what, if anything, it hits.

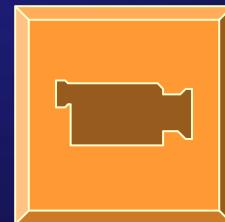
Ray Tracing Diagram (Diffuse Surface)



P is visible; illuminated by yellow light; but not by red light

Ray Tracing Applet: Incorporating Reflective and Translucent Surfaces (2D for Simplicity)

http://www.siggraph.org/education/materials/HyperGraph/raytrace/rt_java/raytrace.html



Colors Computed While Unwinding the Recursion

For each ray-traced pixel:

- Know surface that was hit (if any);
- Know which point lights are visible from shadow rays;
- Apply local illumination model at hit surface point.

Get shadows “for free”.

Feeds the LOCAL Illumination Formula

$$\begin{aligned}\bar{C}_{\text{reflected}} &= \\ k_{\text{ambient}} \bar{C}_{\text{ambient}} \bar{C}_{\text{surface}} &+ \\ \sum_{j=1}^m \delta_{L_j} \bar{C}_{L_j} \left[k_{\text{diffuse}} C_{\text{surface}} (L_j \cdot N) + k_{\text{specular}} (R_j \cdot V)^n \right]\end{aligned}$$

where $\delta_{L_j} = 1$ if light L_j is visible, otherwise $= 0$.

Color (r, g, b) reflected at the visible point =
ambient light + (for each light source ray that is not occluded) light
color \times [diffuse component \times surface color +
specular reflection component]

“Typical” Ray Traced Image



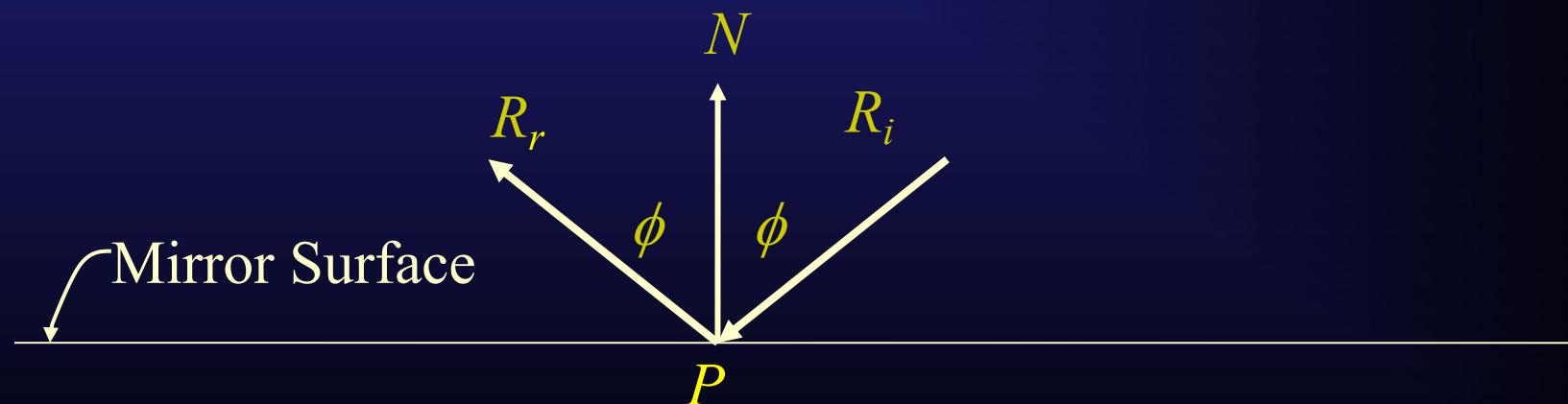
© 1985 J. ARVO/M. SCIULLI — APOLLO COMPUTER INC.

Mirror Reflection

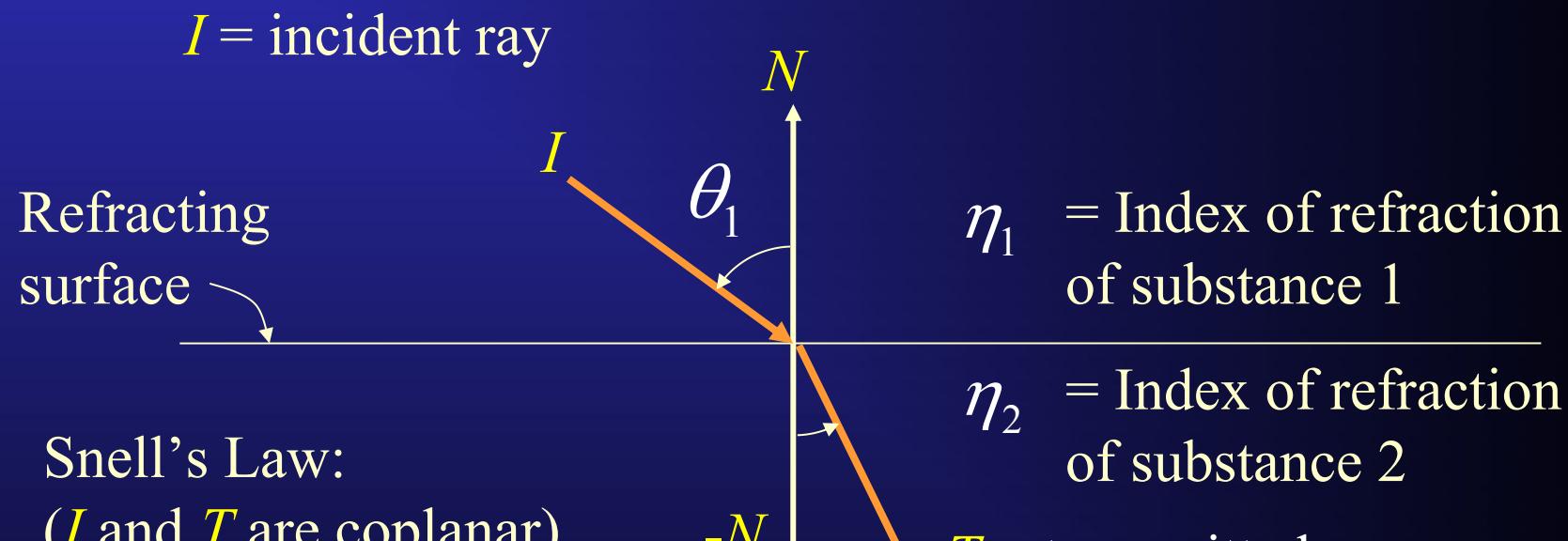
R_i = Incident ray direction (unit vector) to some surface point P.

R_r = Reflected ray direction (about P).

$$R_r = R_i - 2 N (R_i \bullet N)$$



Refracted Ray Geometry: Index of Refraction η and Snell's Law



η_1 = Index of refraction
of substance 1

η_2 = Index of refraction
of substance 2

T = transmitted ray

$\eta =$
1.0 vacuum
1.00029 air at sea level
1.31 ice
1.333 water 20°
1.52 crown glass
1.66 flint glass
2.42 diamond

An index of refraction that varies by wavelength causes dispersion (e.g., a prism breaks white light into colors).

Computing the Refracted Ray

- One can derive the vector T :

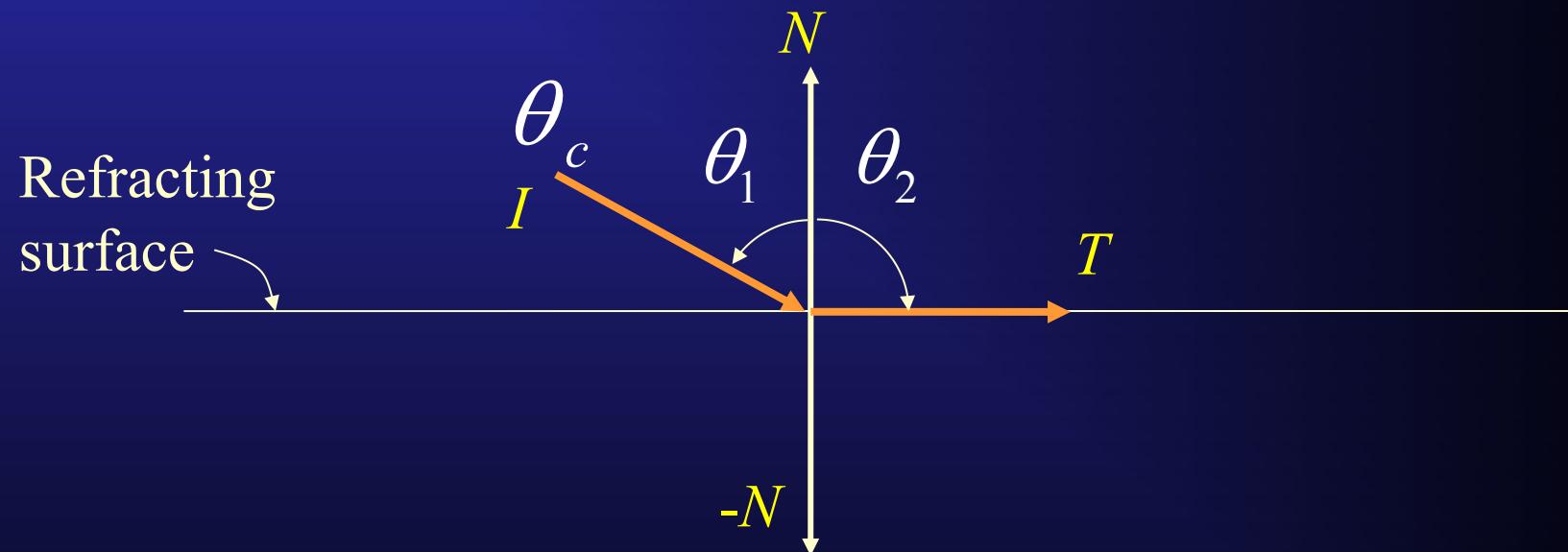
$$T = \left(-\eta_{12}(N \cdot I) - \sqrt{1 - \eta_{12}^2(1 - (N \cdot I)^2)} \right) N + \eta_{12}I$$

where $\eta_{12} = \eta_1 / \eta_2$

- Notice that the $\sqrt{}$ may become imaginary; in this case there is no refraction at all.
- This is called the **critical angle** θ_c .

Refraction and the Critical Angle

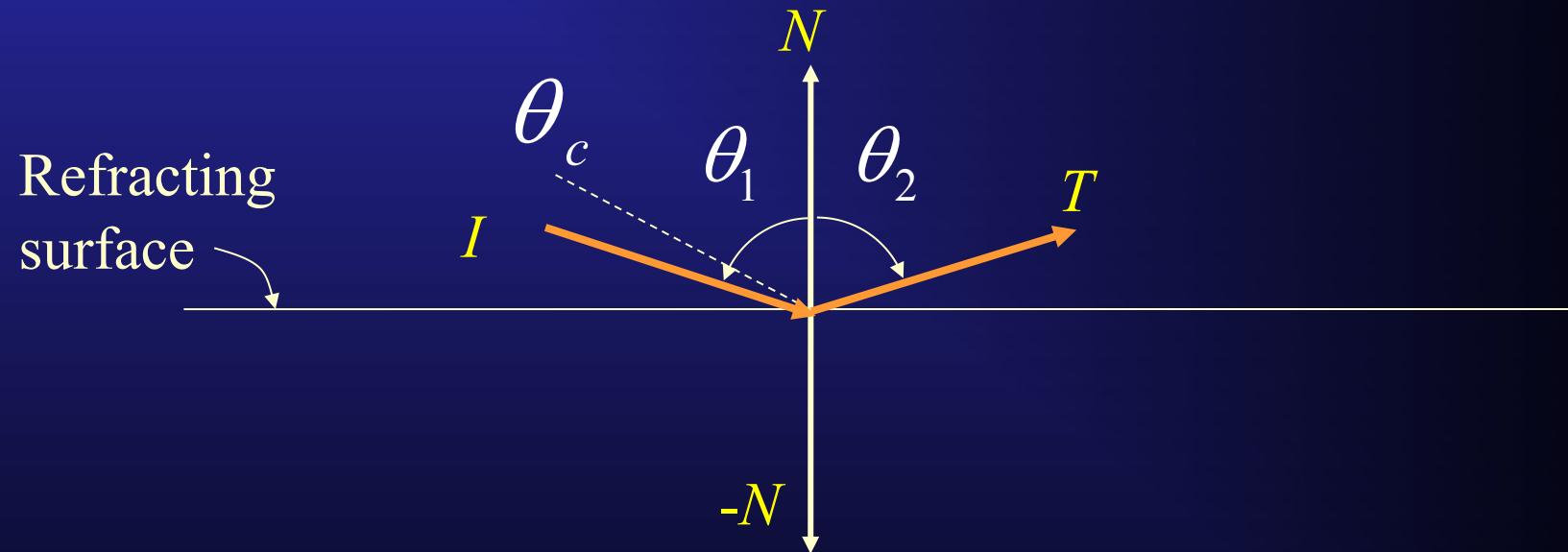
- No refraction at the critical angle θ_c



if $\theta_1 = \theta_c$ then $\theta_2 = \pi/2$

Refraction and the Critical Angle

- Internal (mirror) reflection beyond the critical angle θ_c



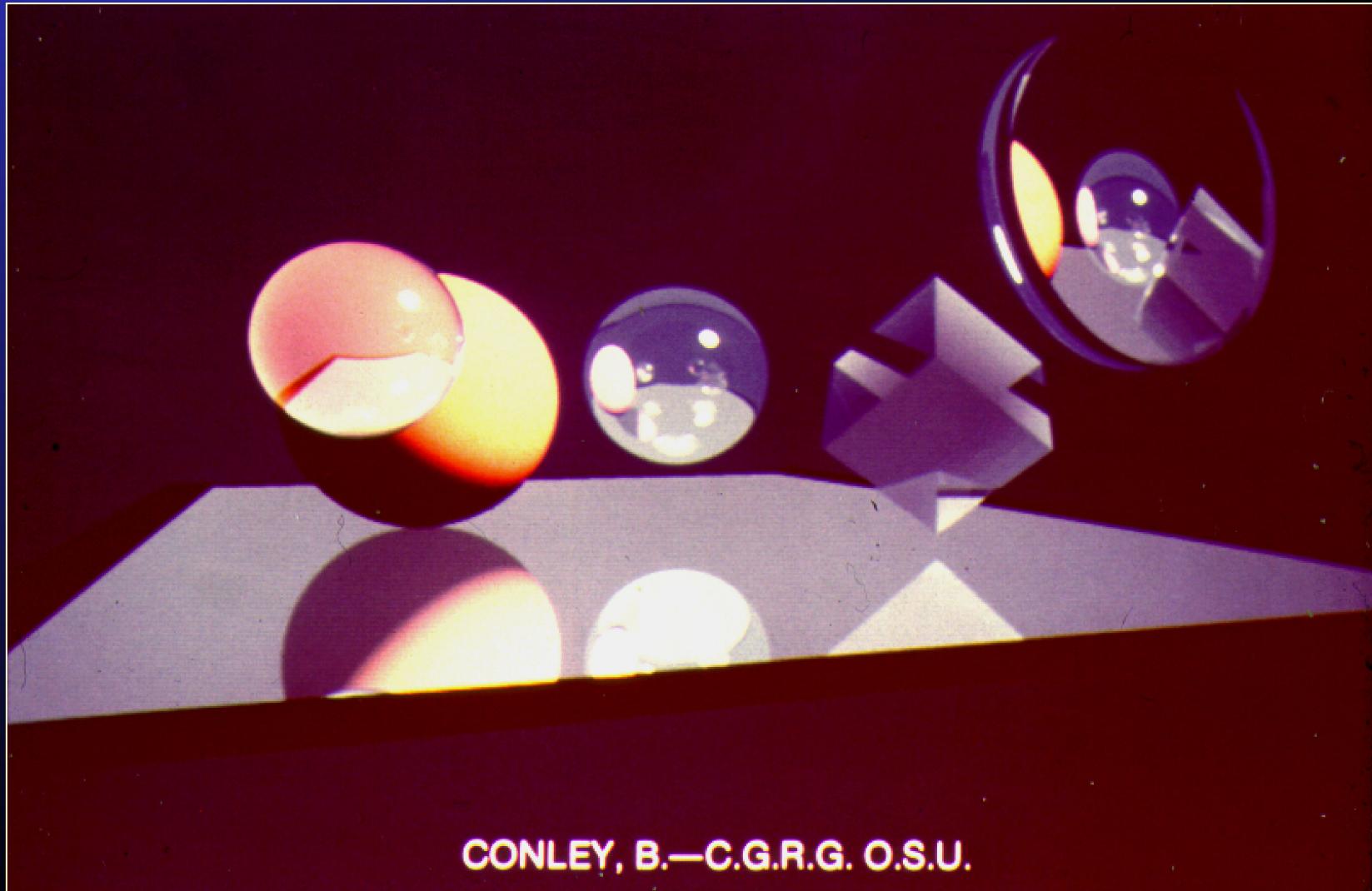
if $\theta_1 > \theta_c$ then $\theta_2 = \theta_1$

Complex Images from Reflection and Refraction



© 1984 PATRICIA SEARCH
RENSSELAER POLYTECHNIC INST.

Ray Tracing with Several Ray Effects



CONLEY, B.—C.G.R.G. O.S.U.

Recursive Ray Trace Pseudo-Code: Some Definitions

rgb LtSource [m];	<i>{color of light source (m of them)}</i>
rgb BackgroundColor;	<i>{background color}</i>
rgb ambient;	<i>{ambient light color}</i>
int MaxDepth;	<i>{maximum recursion depth of ray tree}</i>
rgb color;	<i>{returned color}</i>
ray R, D, P, ...;	<i>{rays}</i>
vector N;	<i>{surface normal}</i>
vector L[m];	<i>{list of m light source locations}</i>
object objects [n];	<i>{list of n objects in scene}</i>
rgb C[n];	<i>{color for each object}</i>
float Kd [n];	<i>{diffuse reflectivity factor for each object}</i>
float Ks [n];	<i>{specular reflectivity factor for each object}</i>
float Kn [n];	<i>{specular Phong exponent for each object}</i>
float Kt [n];	<i>{transmittance for each object}</i>
float Ka [n];	<i>{ambient factor for each object}</i>
float Kr [n];	<i>{index of refraction for each object}</i>

The Rays are Generated in the Outer Loops – (Refer to “Simple Viewing Geometry”)

for pixel(x,y) column x = 0 to width-1

 for pixel(x,y) row y = 0 to height-1

 begin

 P = M + (2*x/(width-1) - 1)*H + (2*y/(height-1) - 1)*V;

 D = (P - E) / |P - E|;

 TraceRay(E, D, 0, color);

 pixel(x,y) = color

 end

Ray Tracing Algorithm

```
procedure TraceRay(start, direction: vectors;
                    depth: integer; var color: rgb);
ray ReflectedDirection, RefractedDirection, TransmittedDirection;
rgb spec, ReflectedColor, refr, RefractedColor, trans, TransmittedColor;

begin
  if (depth > MaxDepth) then
    color = BackgroundColor
    return;
  {intersect ray with all objects and find intersection point (if any) on
  object j that is closest to start of ray; else return nil}
  IntersectionPoint = RayIntersect(objects, j);
  if j == nil
    then if (ray is parallel to any of the m light directions, say k)
        then color = LtSource[k]
        else color = BackgroundColor
    else {have first intersection with object j}
```

```
begin
  (compute object j normal N at IntersectionPoint)
  if (Ks[j] > 0) then {non-zero specular reflectivity}
    {Compute direction of reflected ray}
    TraceRay(IntersectionPoint, ReflectedDirection,
              depth+1, ReflectedColor);
    spec = Ks[j] * ReflectedColor
  else spec = 0;
  if (Kt[j] > 0) then {non-zero transmittance}
    {Compute direction of refracted ray}
    TraceRay(IntersectionPoint, RefractedDirection,
              depth+1, RefractedColor);
    refr = Kt[j] * RefractedColor
  else refr = 0;
```

```

{do shadow rays}
if inside(ray, object[j])
then color =(0,0,0)
else begin {simple local reflectance model}‡
color = Ka[j] * ambient * C[j] + refr;
for i =1 to m
  if ShadowRayUnblocked(IntersectionPoint, L[i]))
    then color += LtSource[i] * (Kd[j] * C[j] * (N · L[i])
      + spec*(ReflectedRay(L[i], N,
                            IntersectionPoint) · E)^Kn[j]))
end
end
end

```

boolean ShadowRayUnblocked(P1,P2): computes whether the ray from P1 to P2 is blocked by any object.

ray ReflectedRay(P1, PN,PO): returns the reflection of ray P1 about normal PN at point PO.

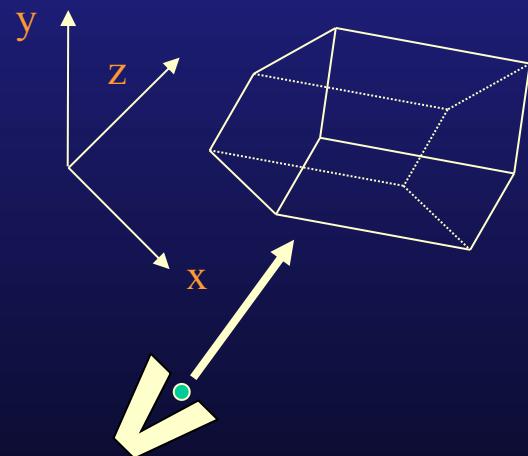
‡ Uses the Local Illumination model formula

Ray-Object Intersection Efficiency

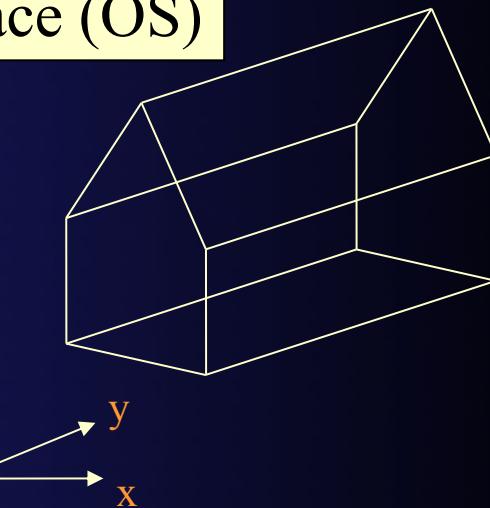
- Ray in world space coordinates: $R = (R_{\text{start_ws}}, R_{\text{direction_ws}})$.
- Polygon mesh object $P = \{P_j\}$ has numerous 3D vertices (j is large).
- World coordinates of P are CP for some composite modeling transformation C from the scene graph.
- Computing world coordinates of mesh object in the scene graph is very expensive and unnecessary!
- *Transform ray into mesh object definition space, find intersection (if any) and then transform any intersections back into world coordinates.*
- Works for other modeling types, too, such as surfaces, quadrics, volumes, and implicit functions: that means the intersection test can be done in a standard geometric position, which makes the math much nicer!

Intuition: Transform 3D Model from Object Space (OS)
to World Space (WS) [O(object vertices)]

Object definition space (OS)



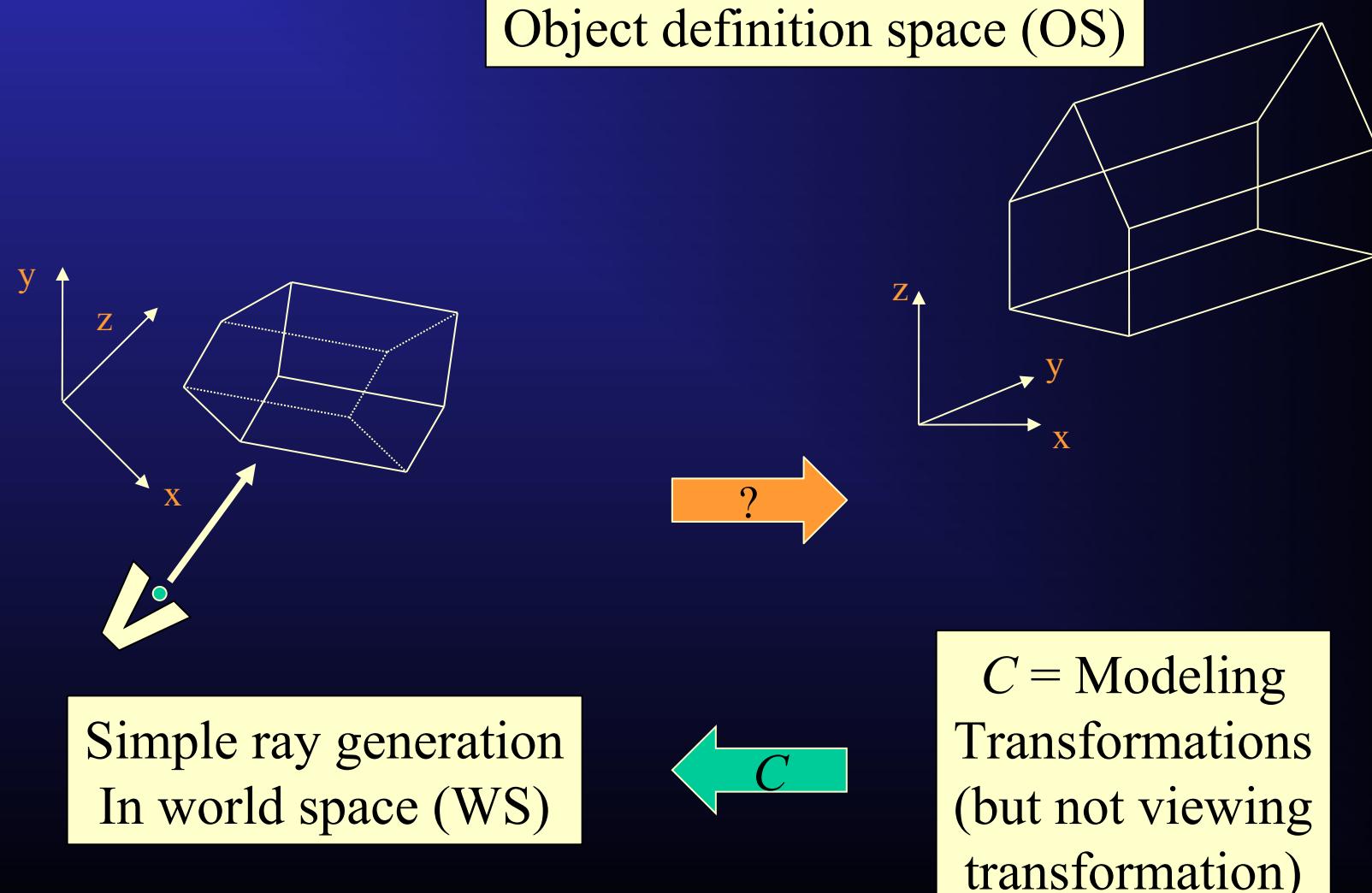
Simple ray generation
In world space (WS)



C

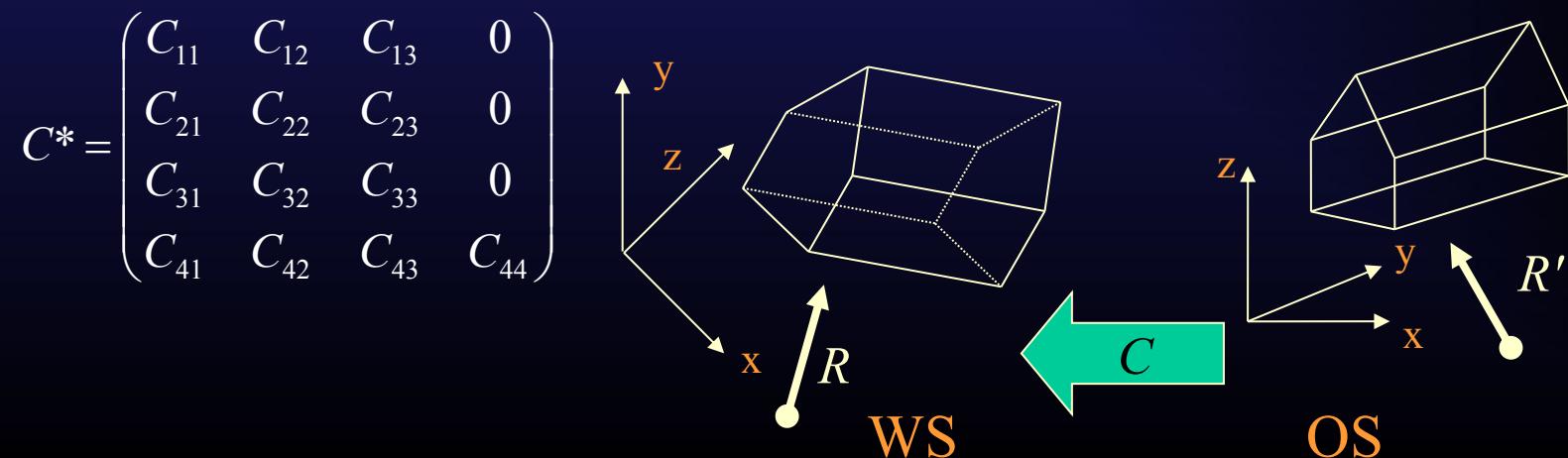
C = Modeling
Transformations
(but not viewing
transformation)

More Efficient to Transform Ray into Object Space! [O(2)]



Transforming Ray Back into Object Coordinates

- Let C be the local object definition space (OS) to world coordinate space (WS) modeling transformation: $\{\text{Model}\}_{\text{ws}} = C \{\text{Model}\}_{\text{os}}$
- Transform the WS ray's origin $R_{\text{start ws}}$ and direction $R_{\text{direction ws}}$ to OS.
- Origin is just $R'_{\text{start os}} = C^{-1}R_{\text{start ws}}$
- How about $R'_{\text{direction os}}$? Cannot just do $R'_{\text{direction os}} = C^{-1}R_{\text{direction ws}}$.
- Endpoints of ray have equal translations, so transform both ends and subtract result. This is equivalent to just ignoring the translation component of C : $R'_{\text{direction os}} = (C^*)^{-1}R_{\text{direction ws}}$



What About Mapping the Ray-Surface Intersection Back into the World Space?

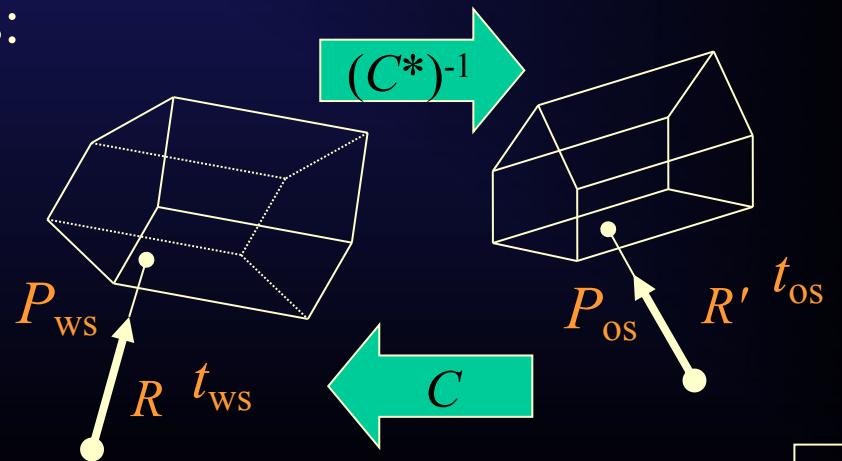
- If we do the ray intersection in OS we get an intersection parameter t_{os} ; we need to have t_{ws} so we can compute the surface color, etc.
- If we use normalized directions for rays then after

$$R'_{\text{direction_os}} = (C^*)^{-1} R_{\text{direction_ws}}$$

$\| R'_{\text{direction_os}} \|$ may not be 1 (e.g., if C has scale factors $\neq 1$).

- So $t_{ws} = t_{os} / \| R'_{\text{direction_os}} \|$
- The intersection point in WS is:

$$P_{ws} = R_{\text{start_ws}} + t_{ws} * R_{\text{direction_ws}}$$



Reality Check

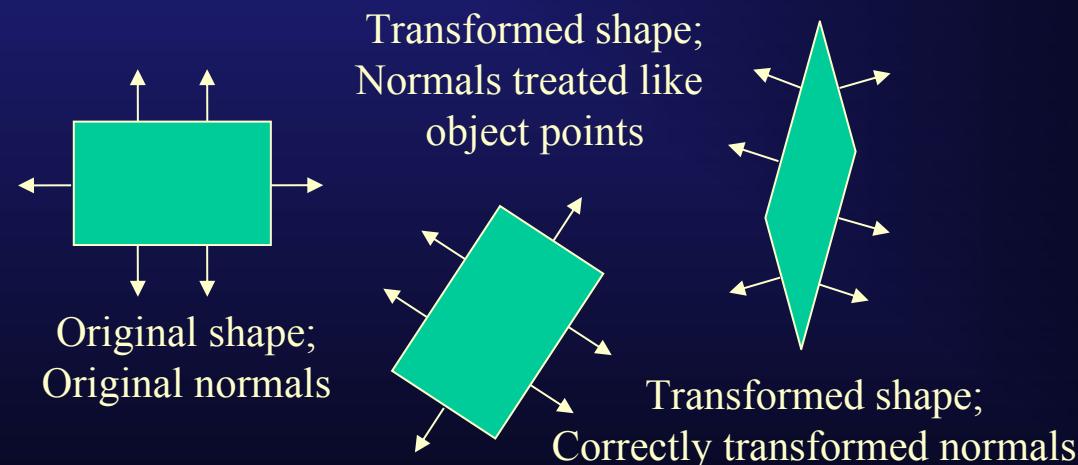
1. Take an OS unit cube centered at the origin that is uniformly scaled by 2 into WS: its surfaces are in the planes $x = \pm 2$, $y = \pm 2$, and $z = \pm 2$.
2. Suppose a unit length WS ray with starting point $(4, 0, 0)$ points straight at the $x = 2$ face of the cube. Trivially, we would expect $t_{ws} = 2$, because the cube's face is hit at $(2, 0, 0)$.
3. Now transform the ray into OS: this entails scaling it by 0.5 ($=1/2$ = inverse of the cube's WS modeling transformation). The transformed ray starts at $(2, 0, 0)$ now, but is only *half* the length. But to reuse any ray-object intersection code, we re-normalize the ray to *unit* length.
4. The OS cube is centered at the origin, and its face is at $(1, 0, 0)$. $t_{os} = 1$ because the ray starting point and cube face are only 1 unit apart.
5. But the ray WAS scaled from WS to OS therefore $\|R_{os}\| = 0.5$.
6. So then $t_{ws} = 1/0.5 = 2$, the right answer as per (#2).

Transform the Normal Back into the World Space

- We still need the transformed normal from the object space intersection:

$$N_{ws} \neq C N_{os}$$

- So can't just use C^{-1} !



- Key is to transform the TANGENT plane...

Transform the Normal Back into the World Space

Pick any vector V_{os} in the tangent plane: $V_{ws} = C V_{os}$

$N_{os} \perp V_{os}$ so:

$$0 = N_{os}^T V_{os}$$

$$0 = N_{os}^T (C^{-1} C) V_{os}$$

$$0 = (N_{os}^T C^{-1}) (C V_{os})$$

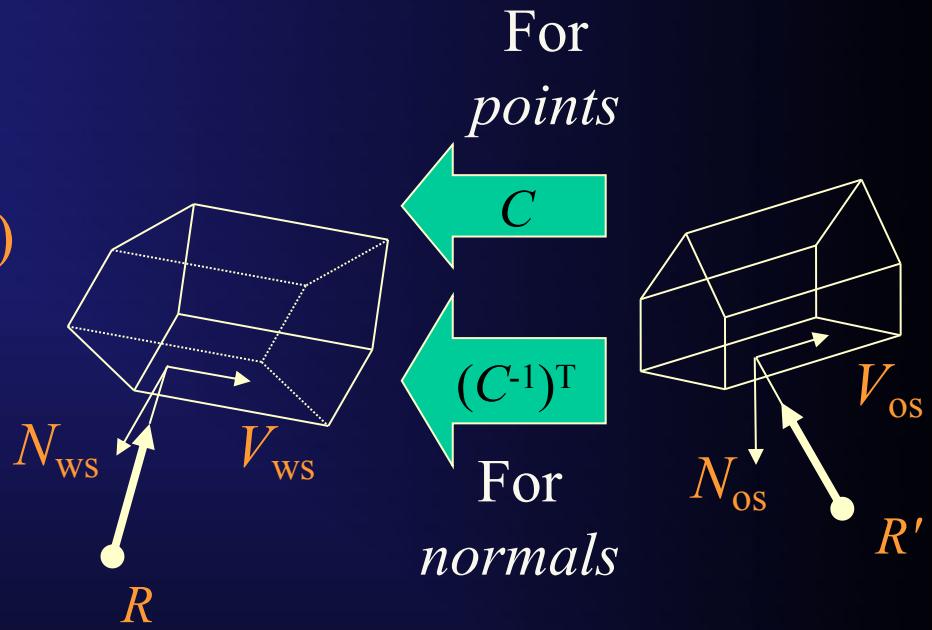
$$0 = (N_{os}^T C^{-1}) V_{ws}$$

Since $N_{ws} \perp V_{ws}$

$$N_{ws}^T = N_{os}^T C^{-1}$$

$$(N_{ws}^T)^T = (N_{os}^T C^{-1})^T$$

$$N_{ws} = (C^{-1})^T N_{os}$$



Ray Tracing Efficiency Considerations

$$COST = X \cdot Y \cdot 2^a (m + 1) (2^n - 1) P$$

- X = number of pixels horizontally
- Y = number of pixels vertically
- a = anti-aliasing super-sampling factor
- m = number of (point) light sources
- n = tree (recursion) depth
- P = number of primitives tested for ray intersections

For example, if $X=Y=1000$, $a=0$, $m=1$, $n=10$, $P=1000$ then

$COST \sim 2,046,000,000,000$ ray-primitive intersection tests!

Clearly there must be a better way!

Ray Tracing Speed Ups

- Extents (bounding boxes)
 - Spatial hierarchies (e.g., finding implicit function surfaces)
 - Spatial partitioning (PVC, BSP)
 - Item buffers
 - Reflection maps
 - Adaptive tree-depth control
 - Ray classification
 - ...
- In practice, ray tracing is not done in real-time (unless reflections are limited to one bounce, like environment maps)
 - There are good commercial software ray tracers.
 - There is a board (ART's RenderDrive) to do limited types of ray tracing.
 - Ray tracing shaders can be cost effective for certain movie special effects.

Ray Tracing Summary

- Ray-tracing provides “hard-edge” effects easily, including specular reflections, refraction, translucency and shadows from multiple light sources.
- Global illumination gives better diffuse lighting. We’ll do this later.

Shade Computation

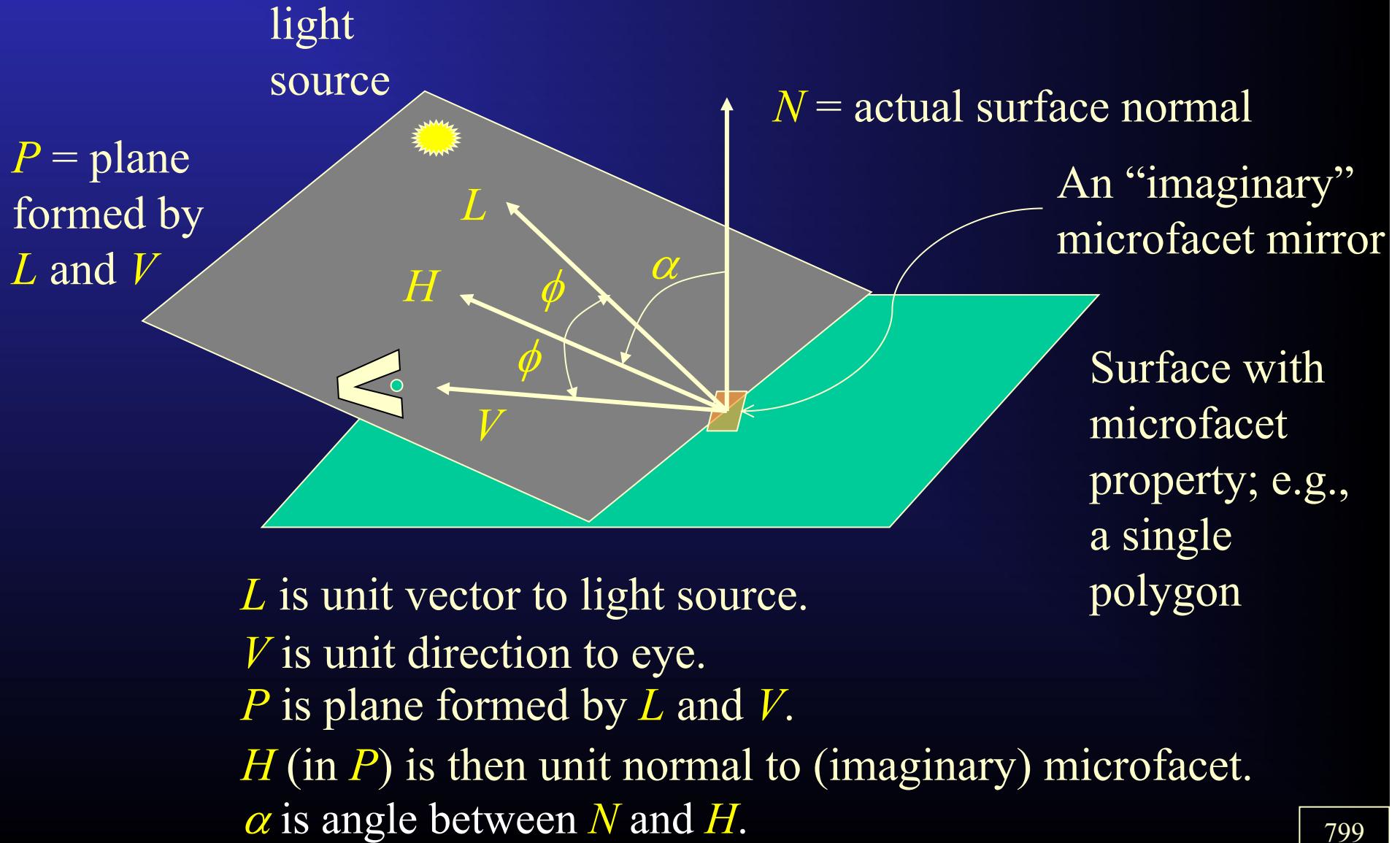
Models:

- Diffuse or Lambert
- Vertex color interpolation and Gouraud
- Perfect specular
- Phong: Glossy specular
- Recursive ray tracing
- MICROFACET
- Empirical

Microfacets: Model Surface Reflectance Properties Without Using “Tiny Polygons”

- How would you create a 3D model of a very rough yet reflective surface, e.g., sandpaper?
- Surface is a **statistical** collection of randomly-oriented “microfacets”; these microfacets are **not explicitly modeled** as polygons, but act like miniature surfaces with their own reflectance properties (e.g., mirror or diffuse).
- The specular component arises from microfacets that are oriented in the direction H : the “half-vector” bisecting the pair of rays from a point to the light and the eye:
- So if the eye is looking at a point on the surface, we compute from a probability distribution whether we “see” the light reflected in a microfacet with normal H .

Microfacet Geometry for a Surface



Specular Reflectance Models for Microfacets

$$R_s = \frac{D G F}{(N \cdot V)}$$

D = proportion of microfacets oriented in direction H ; $\alpha = \cos^{-1}(N \cdot H)$:

$D(\alpha)$ = microfacet distribution: ($0 \leq D(\alpha) \leq 1$): (0 rough; 1 smooth)

G = fraction of light not shadowed or masked ($0 \leq G \leq 1$) :

deep grooves (more attenuation) -- no grooves (flat)

F = Fresnel reflection ($0 \leq F(\phi, \eta, \lambda) \leq 1$) : where

η = index of refraction of surface

λ = wavelength of incident light

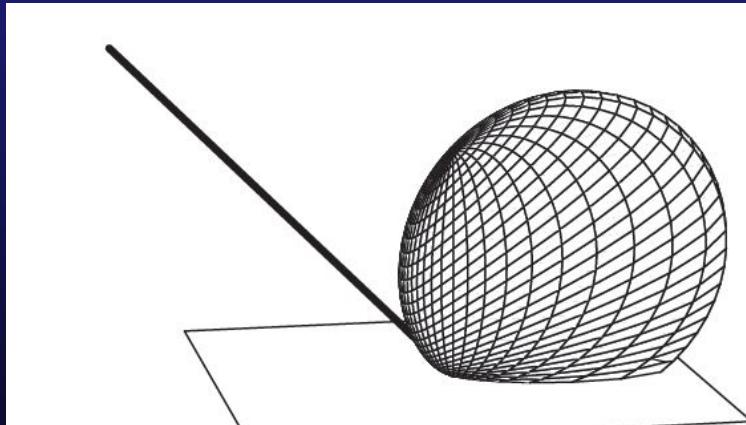
(some light may be absorbed by surface)

Specular Reflectance Function (D)

- Blinn-Phong: $D(\alpha) = \frac{n+2}{2\pi} \cos^n \alpha$

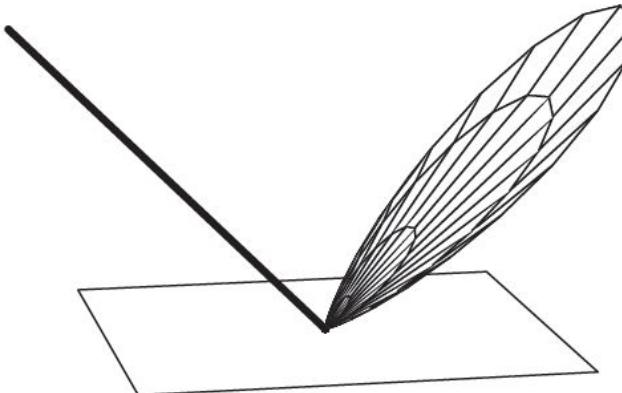
n = 4.0

(rough)



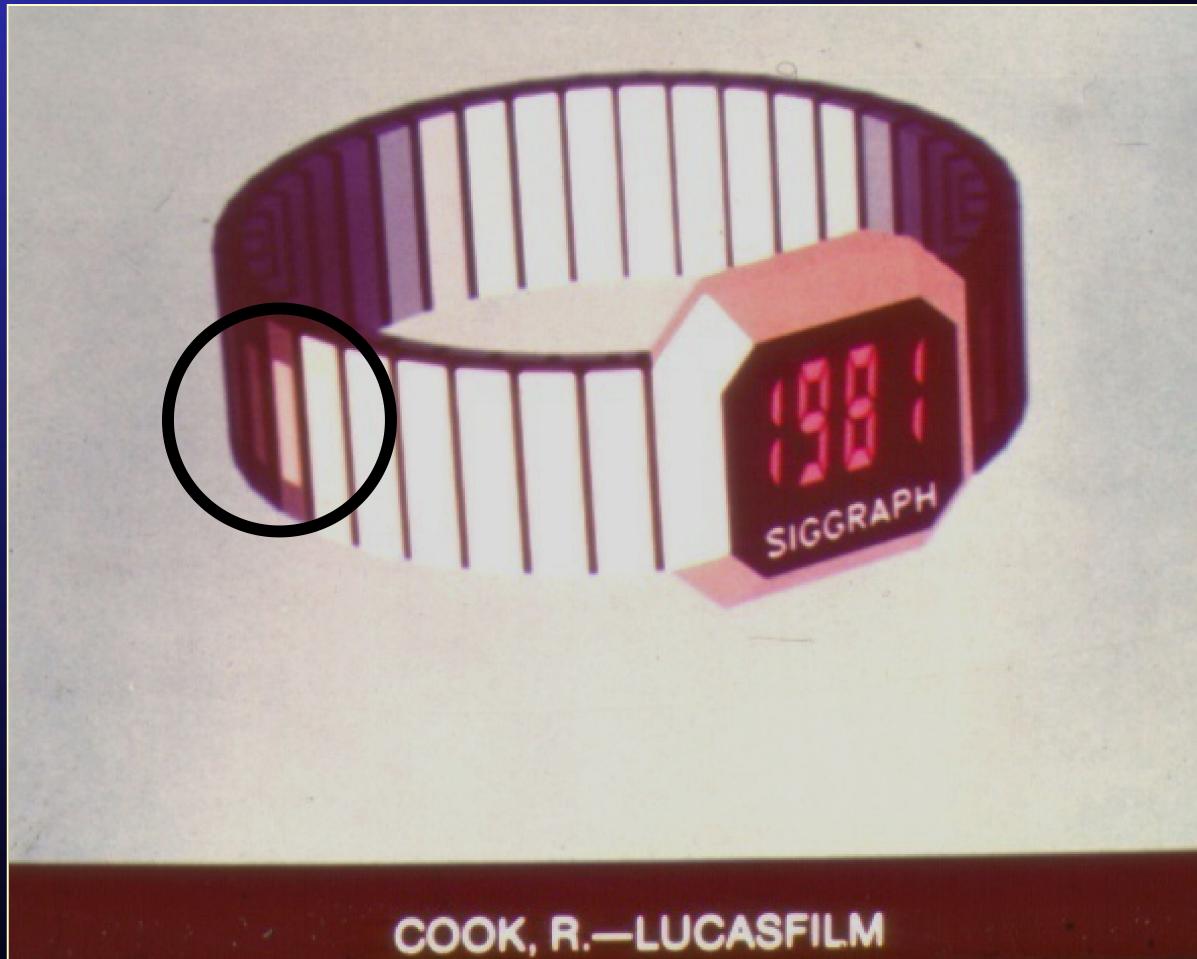
(a)

n = 20.0 (smooth)



(b)

Surface Roughness

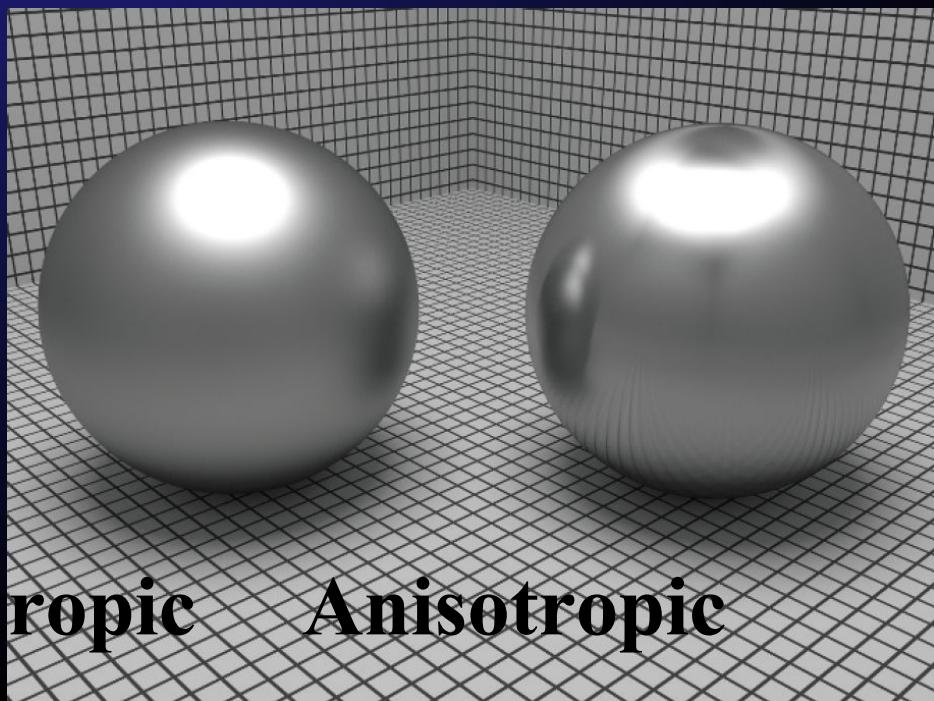
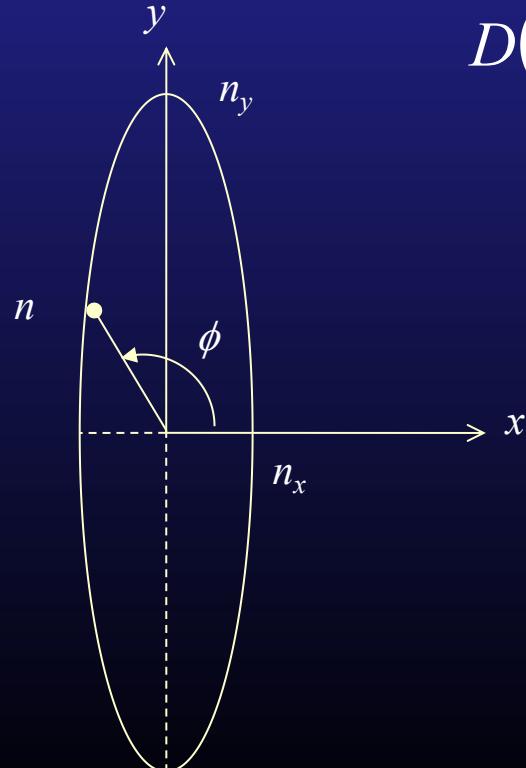


COOK, R.—LUCASFILM

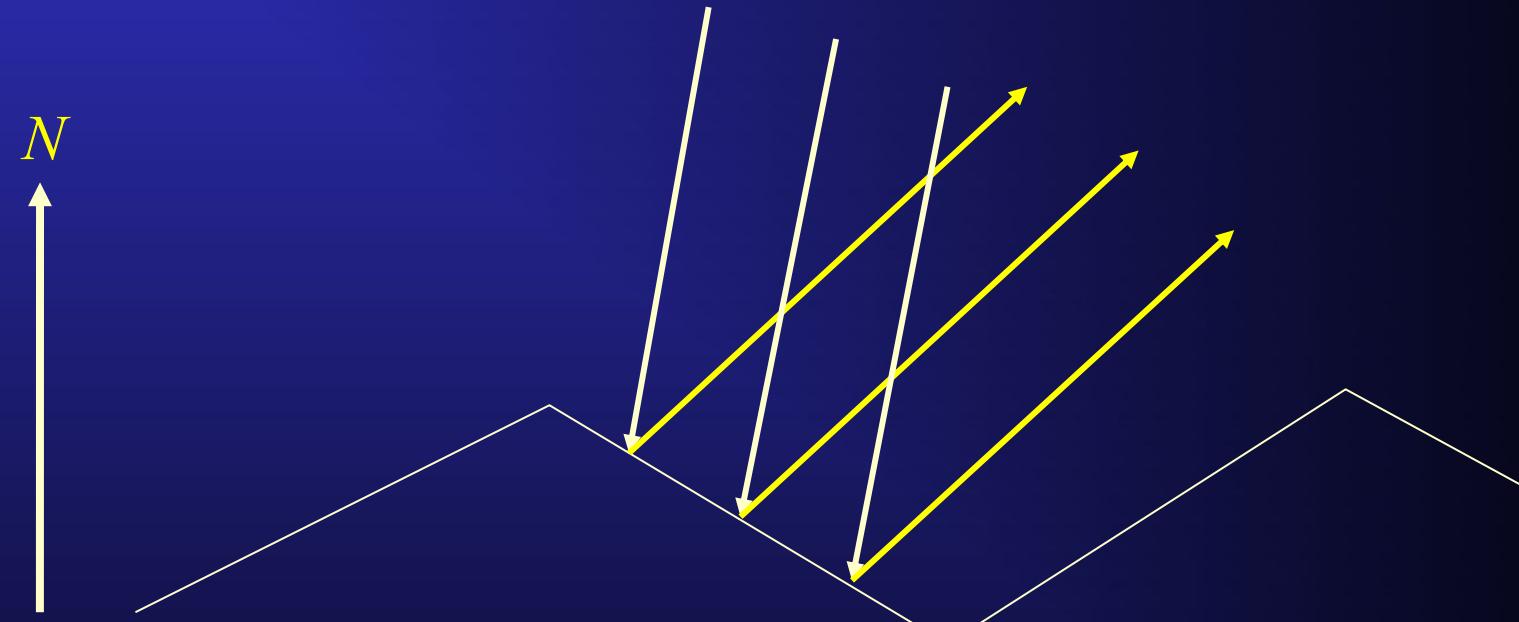
Anisotropic Variance

- Ashikhmin and Shirley's model: Anisotropic microfacet distribution replaces isotropic parameter n by two parameters n_x and n_y which are the Blinn-Phong exponents (magnitudes) of long/short axis of an ellipse:

$$D(\alpha) = \frac{\sqrt{(n_x + 2)(n_y + 2)}}{2\pi} \cos^{n_x \cos^2 \phi + n_y \sin^2 \phi} (\alpha)$$



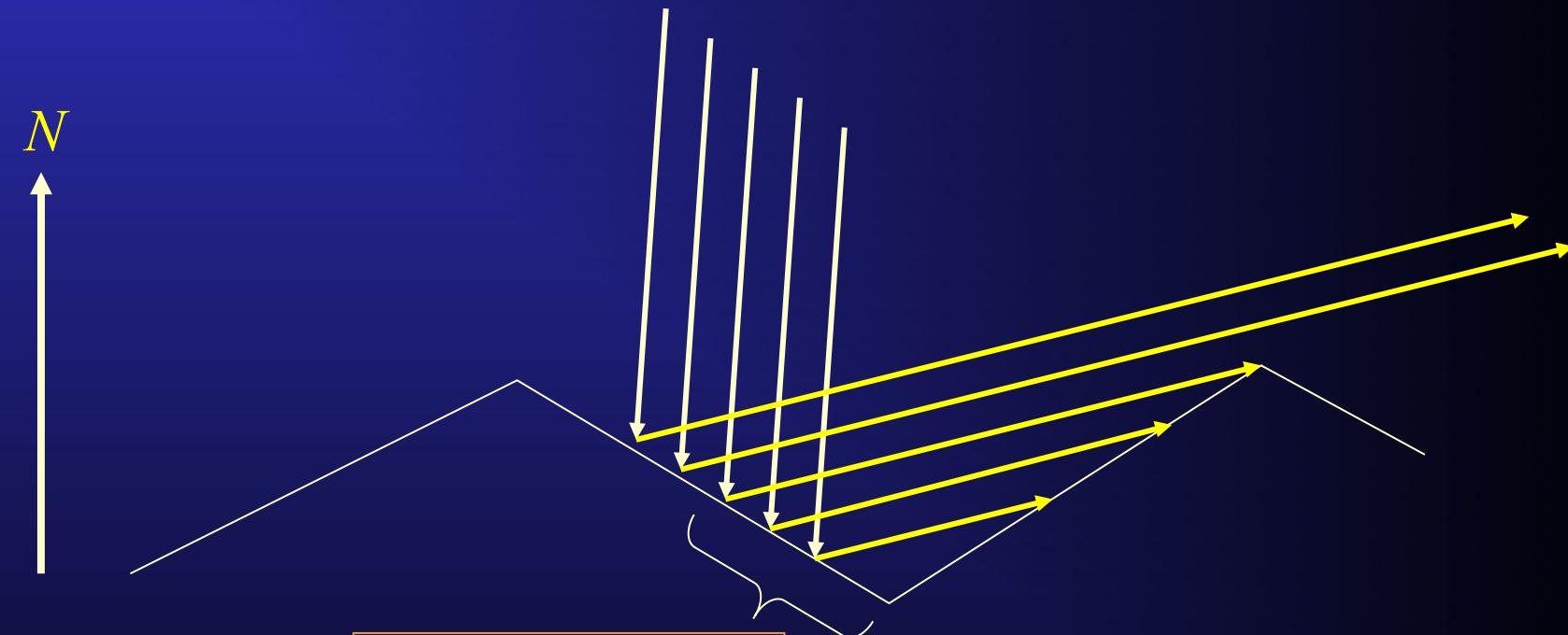
Microfacet Shadowing and Masking (G)



Actual
surface
normal

No interference with microfacets:
 $G_1 = 1$

Microfacet Masking

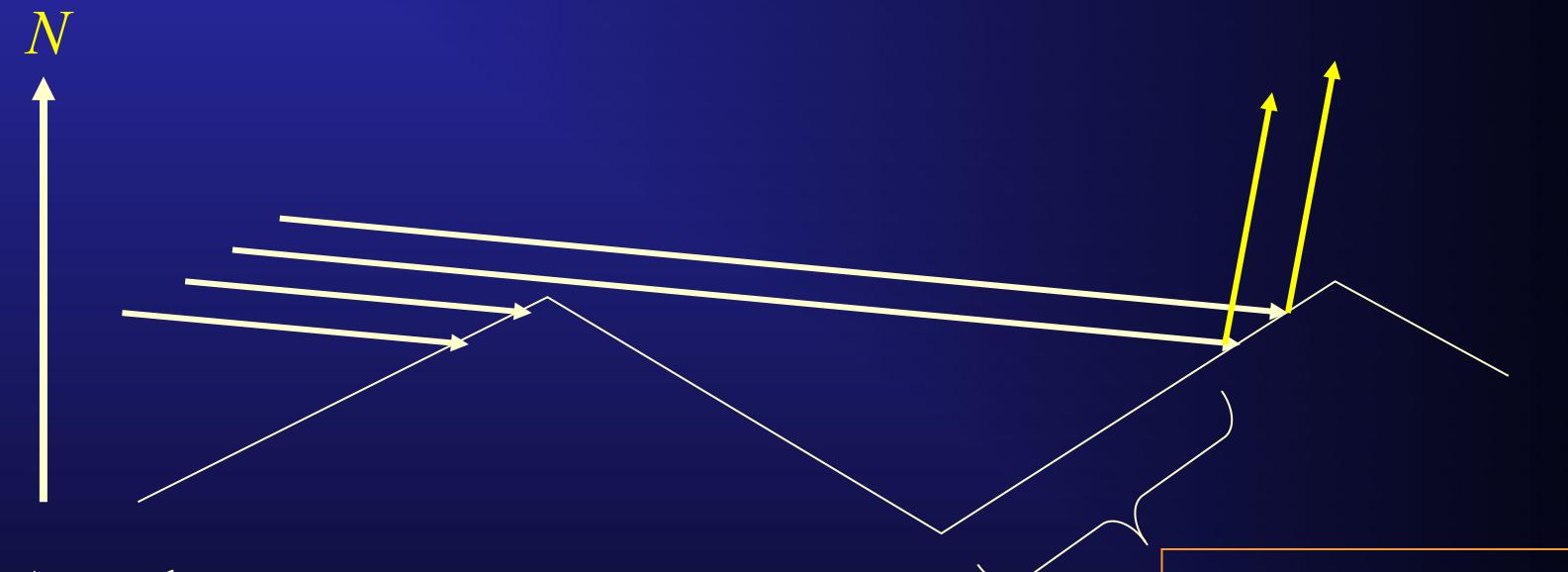


Actual
surface
normal

No light escapes

Some reflected light is blocked:
 $G_2 = 2(N \cdot H)(N \cdot V)/(V \cdot H)$

Microfacet Shadowing



Actual
surface
normal

Some incident light is blocked:
 $G_3 = 2(N \cdot H)(N \cdot L)/(V \cdot H)$

Summary: $G = \min(G_1, G_2, G_3)$

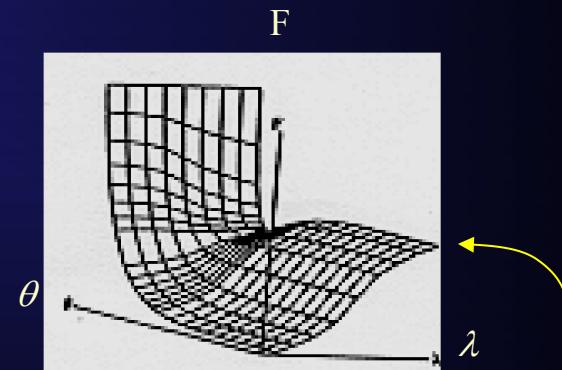
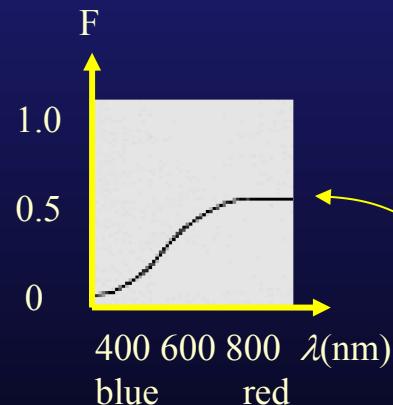
Shade Computation

Models:

- Diffuse or Lambert
- Vertex color interpolation and Gouraud
- Perfect specular
- Phong: Glossy specular
- Recursive ray tracing
- Microfacet
- EMPIRICAL

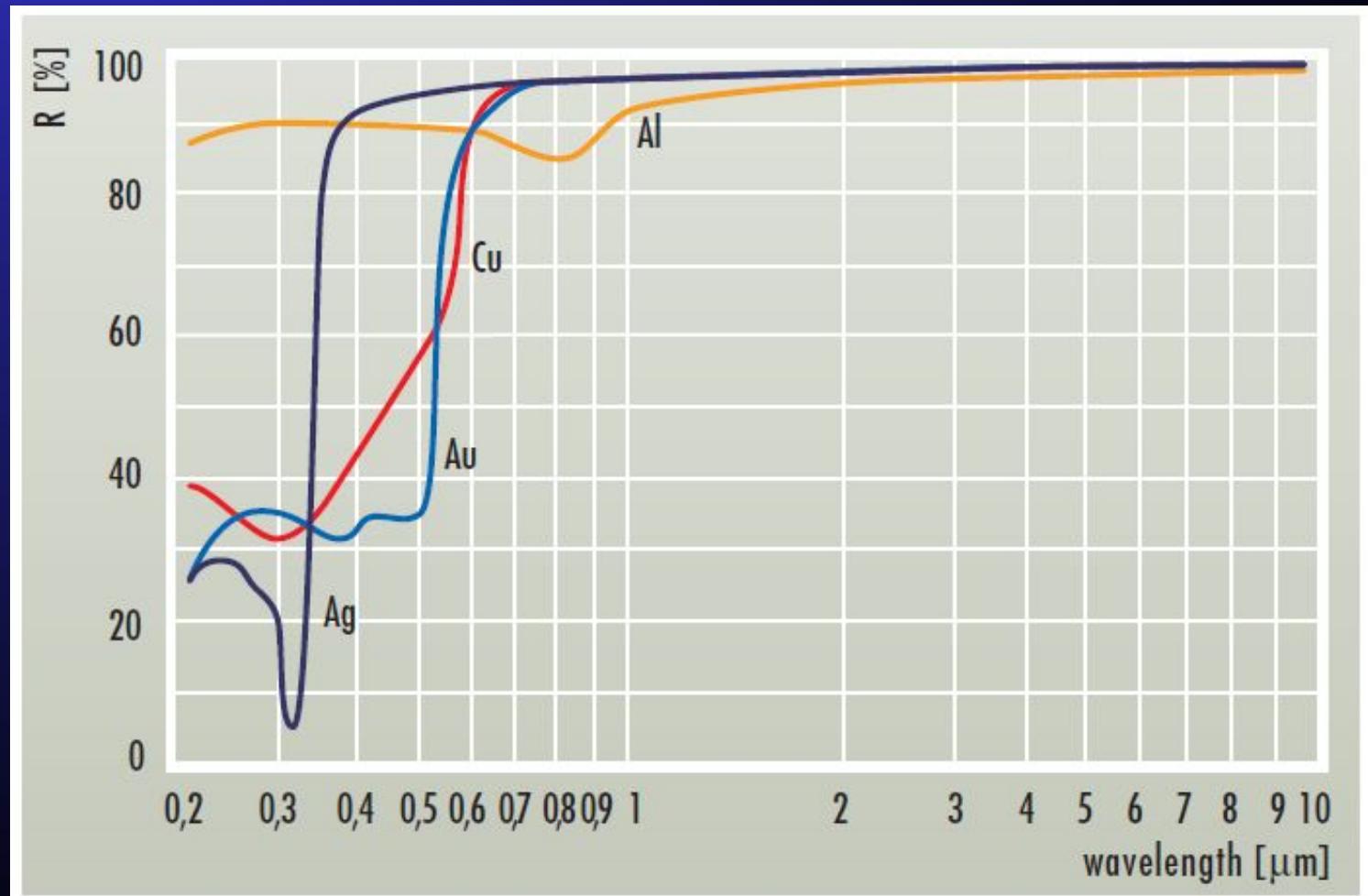
Reflectance Depends on Wavelength of Light (F)

- Fresnel term F is a function of surface index of refraction and wavelength of incident light.
- Computed by multiple passes for several wavelength samples (or RGB colors), then summed.



Reflectance of bronze at normal incidence ($\theta = 0$)
and as a function of incidence angle θ and wavelength λ .

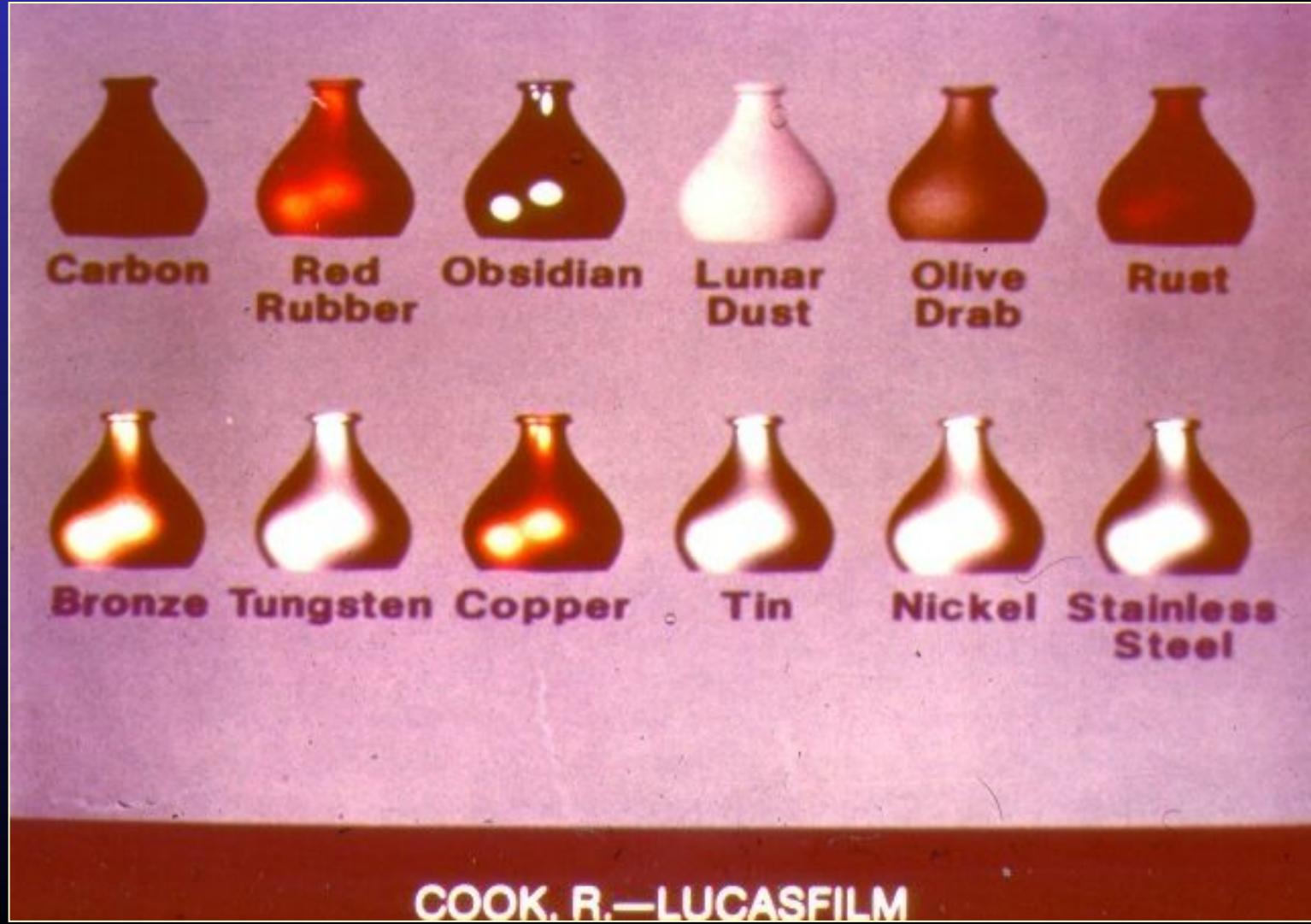
Example Metal Reflectance Functions by Wavelength



Red

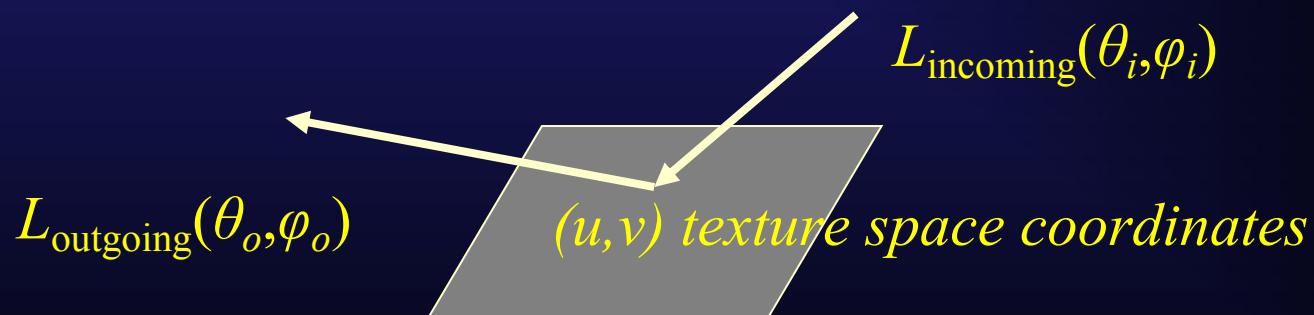
Blue

Varying Material Properties (Cook's Jars)



General Material Reflectance Models

- We need to know desired surface reflectance properties at P .
- That's what distinguishes one material from another – its BRDF:
 - Bidirectional Reflectance Distribution Function
 - General BRDF is a function of incoming and outgoing light directions (in spherical coordinates), surface color λ (wavelength), and observed position (u,v) on the surface:



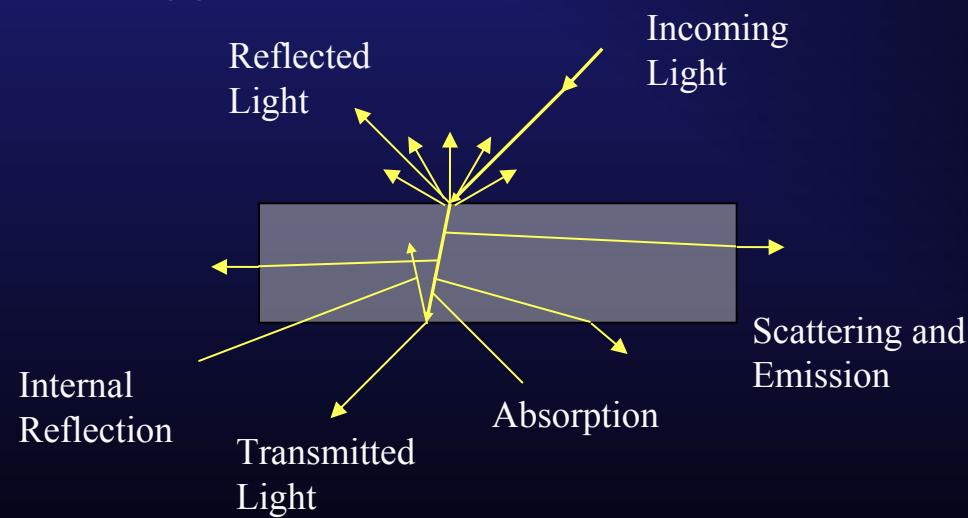
- Often use simple approximations to BRDF or empirical estimates.

Acronyms

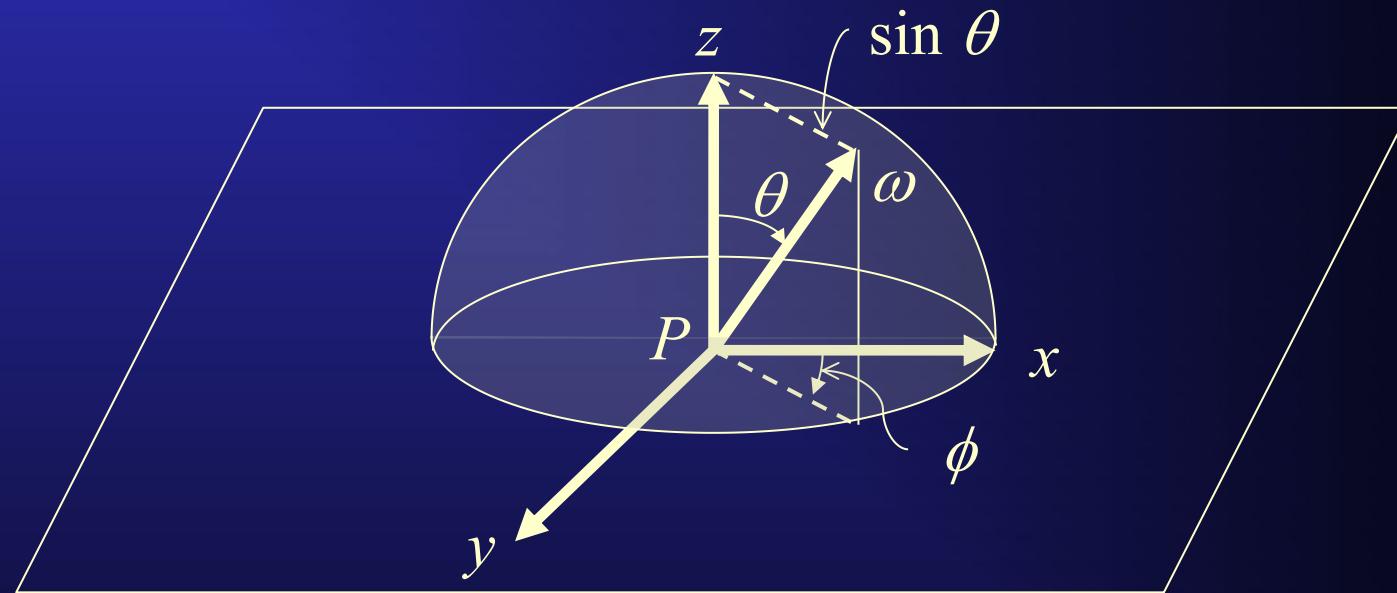
- BRDF – Bidirectional Reflectance Distribution Function
- SPDF – Scattering Probability Density Function
- BSDF – Bidirectional Scattering Distribution Function
- BTDF – Bidirectional Transmission Distribution Function
- BSSRDF – Bidirectional Scattering Surface Reflectance Distribution Function

Light & Matter

- Interaction depends on the physical characteristics of the light and its wavelength, as well as the physical composition and characteristics of the matter:
 - Reflection
 - Absorption
 - Transmittance



Reflectance Geometry at a Surface Point



- Local coordinate frame on surface.
- Incident ω_i or outgoing ω_o light direction, normalized.
- When ϕ matters, surface reflectance is *anisotropic*.

Classes and Properties of BRDFs

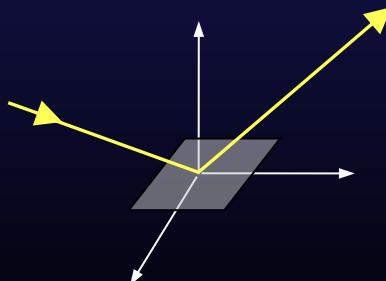
- There are two classes of BRDFs:
 - Isotropic BRDFs represent reflectance properties that are invariant with respect to rotation of the surface around the surface normal vector, e.g., smooth plastics.
 - Anisotropic BRDFs exhibit change with respect to rotation of the surface around the surface normal vector: e.g., most real world surfaces.
- The important properties of BRDFs are reciprocity and conservation of energy.
- BRDFs that have these properties are considered to be physically plausible.

Conservation of Energy

light incident at surface =

light reflected + light absorbed + light transmitted

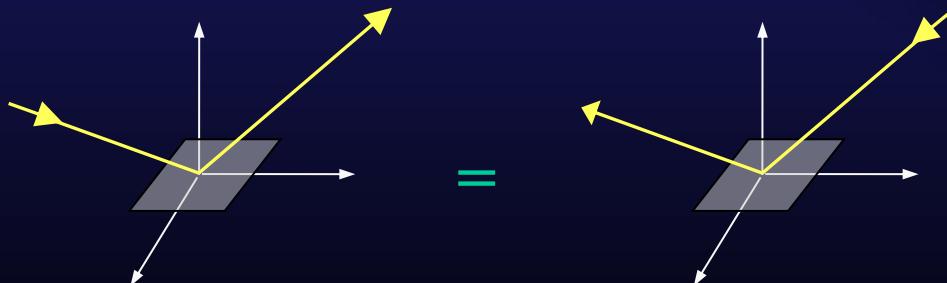
- A BRDF describes how much light is *reflected* when light makes contact with a certain material.
- The degree to which light is reflected depends on the viewer and light position relative to the surface normal and tangent.
- BRDF is a function of incoming (light) direction and outgoing (view) direction relative to a local orientation at the light interaction point.



Reciprocity

- If the incoming and outgoing directions are swapped at the same surface point λ , the value of the BRDF does not change.

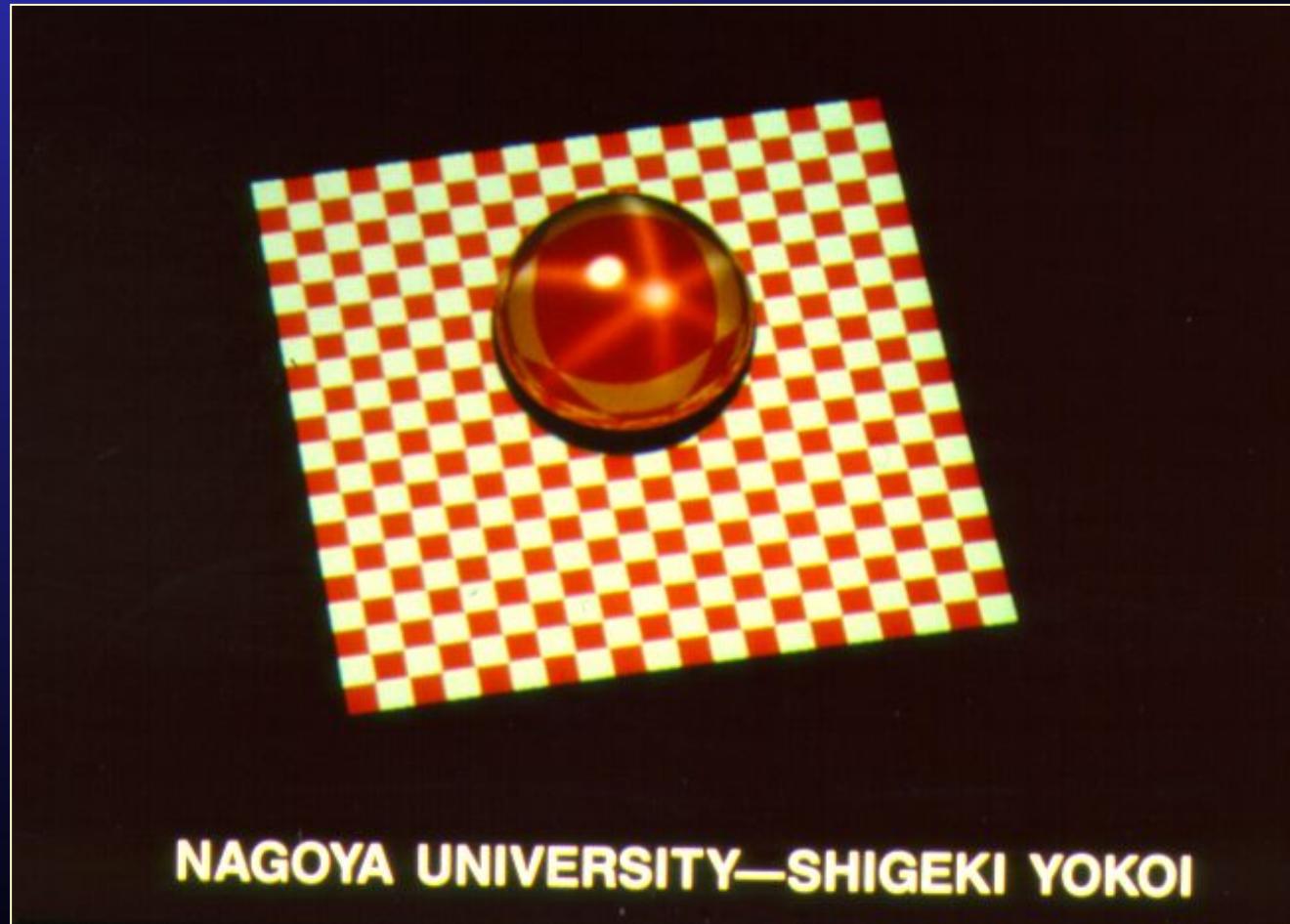
$$\text{BRDF}_\lambda(\theta_i, \phi_i, \theta_o, \phi_o) = \text{BRDF}_\lambda(\theta_o, \phi_o, \theta_i, \phi_i)$$



Positional Variance

- When light interacts differently with different regions of the surface (i.e., varies over u and v), it is called positional variance.
- Occurs in materials that reflect light with notable surface detail (wood, marble, etc.).

Anisotropic Reflection: Amount or Kind of Reflectance BRDF varies with Direction of Normal



NAGOYA UNIVERSITY—SHIGEKI YOKOI

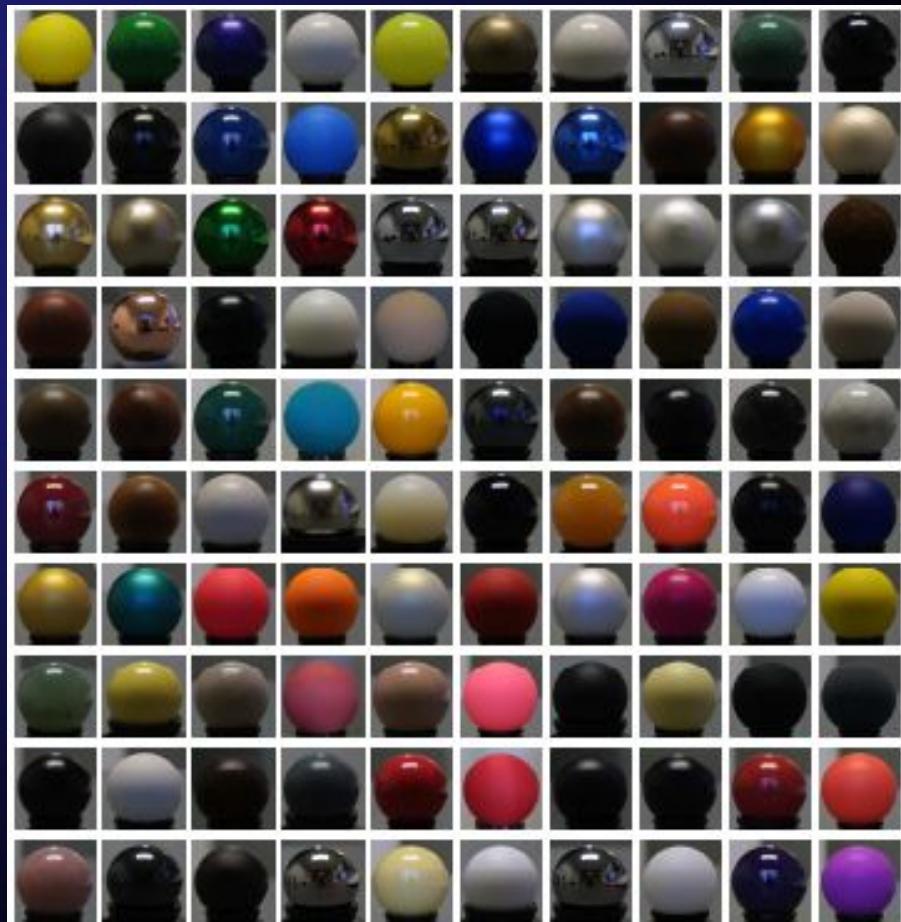
Position-Invariant BRDFs

$$\text{BRDF}_\lambda(\theta_i, \phi_i, \theta_o, \phi_o)$$

- Only valid for homogenous materials: when the spatial position is not included as a parameter to the function the reflectance properties of the material do not vary with spatial position.
- A spatially variant BRDF can be approximated by adding or modulating the result of a BRDF lookup with a texture (u, v) .

Data-Driven (Empirical, Measured) BRDF's of Real Materials

- Dense BRDF measurements on surfaces of 100 different materials.
- 20M-80M samples/BRDF.
- PCA dimensionality reduction into both linear and nonlinear model spaces.



User Navigation in BRDF Space

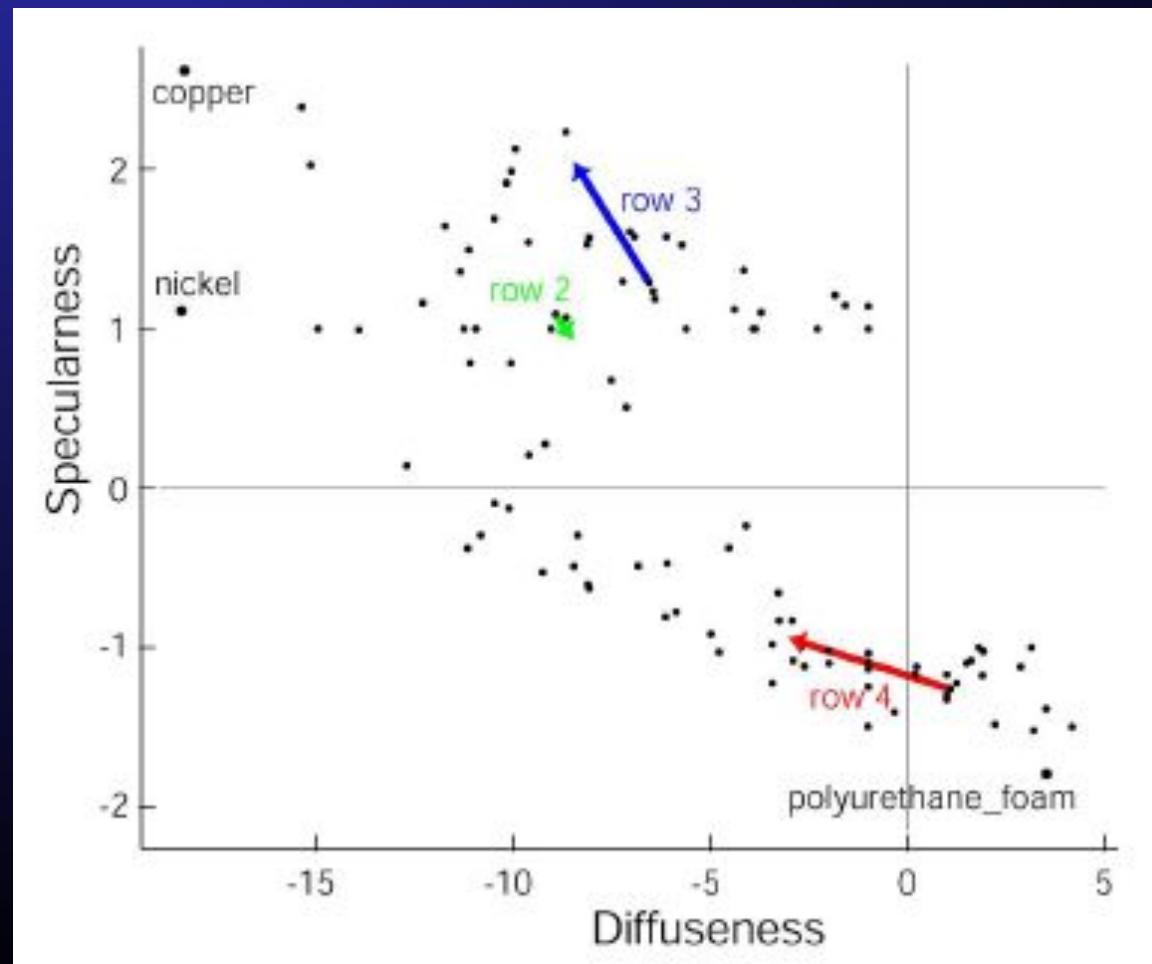
- Subjects assigned (arbitrary) *traits* to each material:
 - redness, greenness, blueness, specularity, diffuseness, glossiness, metallic-like, plastic-like, roughness, silveryness, gold-like, fabric-like, acrylic-like, greasiness, dustiness, rubber-like.
 - With values *possesses*, *not-possesses* or *unclear*.
- Project trait vectors into both linear (45D) and nonlinear (15D) BRDF model spaces.

Parameterizing the Model Spaces

- Used Support Vector Machine(SVM) [Vapnik 1995].
- SVM finds the hyperplane which separates datapoints in one material class from datapoints in a second class.
- The partitioning hyperplane has maximal distance to the closest datapoints (called support vectors) in both material classes.
- The parameterization direction is the (directed) normal to this hyperplane.
- The hyperplane is defined in 15D space for the non-linear model and in 45D space for the linear model.

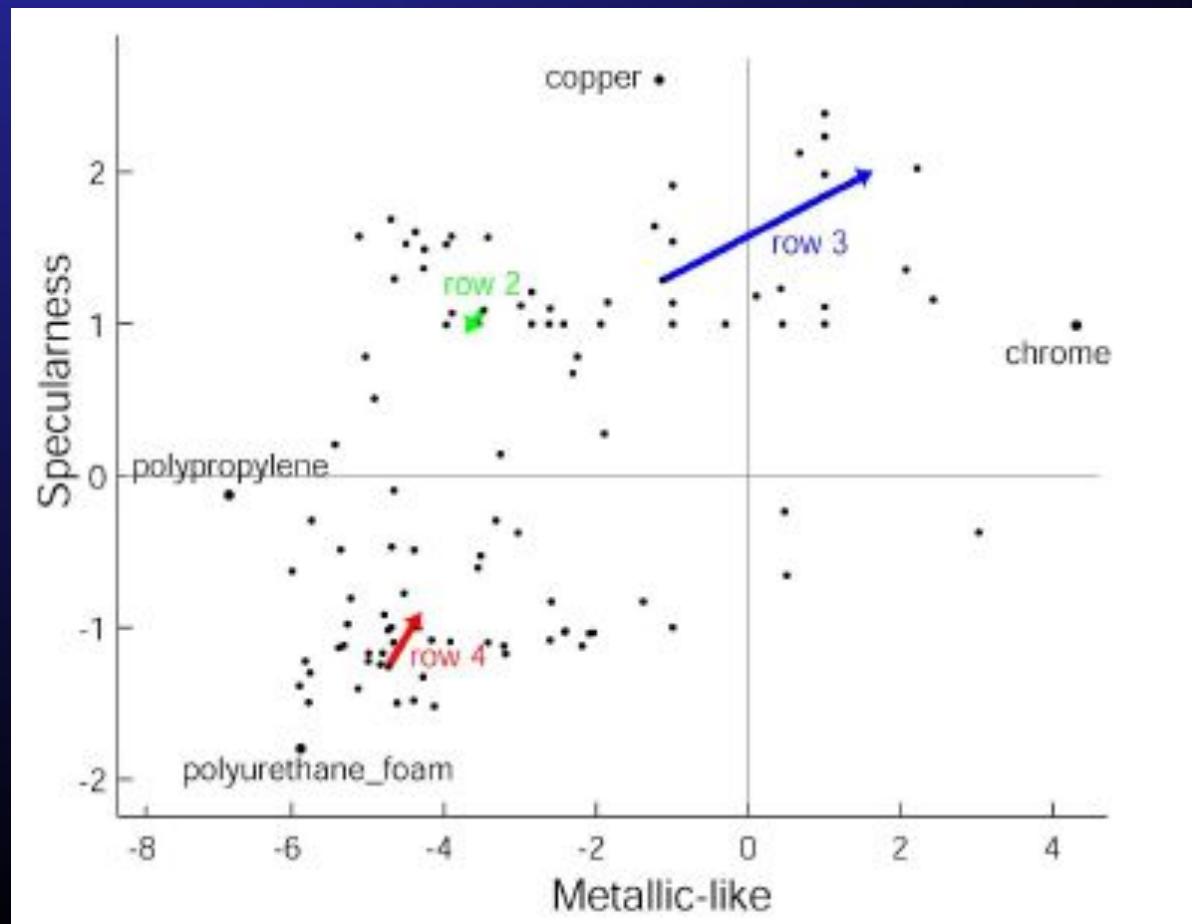
Diffuseness vs. Specularness

Show weak inverse correlation.



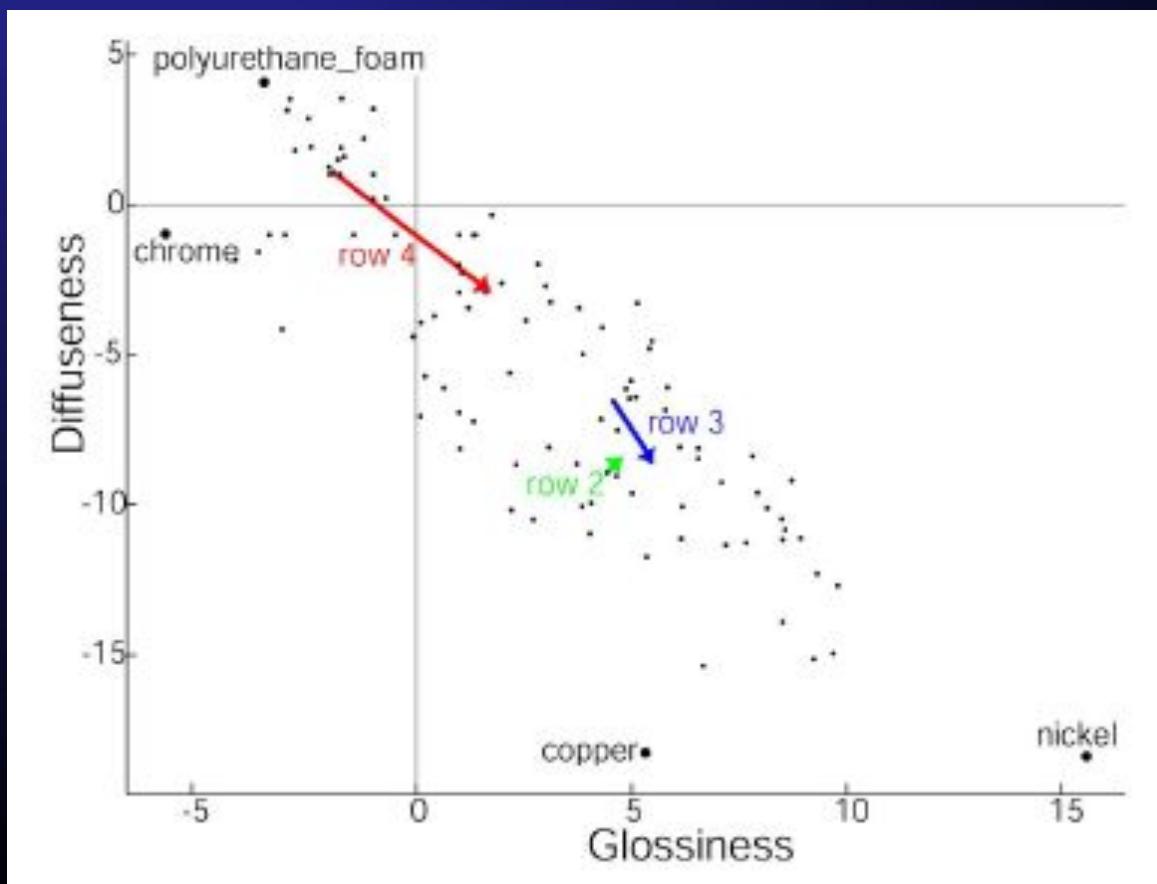
Metallic-like vs. Specularness

Show weak correlation.



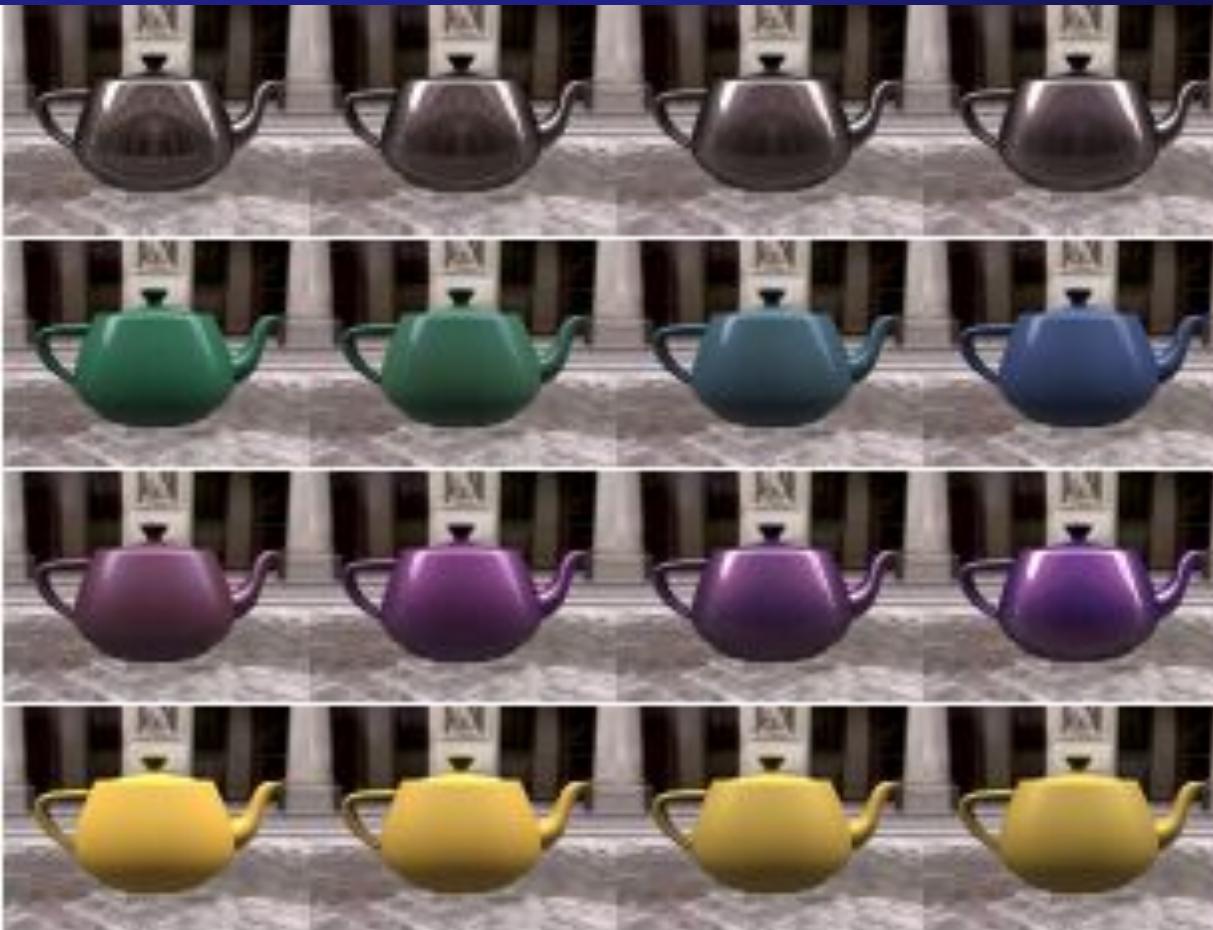
Glossiness vs. Diffuseness

Show inverse correlation.



Results of Moving Along Vectors in Illustrated Subspaces

- The red, green and blue vectors in the previous charts show the projections of the BRDF interpolations along single parameters in the non-linear model in rows 2, 3, and 4 in the image below.



Increase *roughness* applied to *Copper* BRDF.

Increase *blueness* applied to the *GreenAcrylic* BRDF.

Apply metallic trait to *VioletAcrylic* BRDF.

Increase *glossiness* trait to *YellowDiffusePaint* BRDF.

Subsurface Scattering

- Marble, skin, milk other materials reflect light after it bounces around material internal to the surface: **subsurface scattering**.
- Measure scattering empirically from real substance (BSSRDF).

Surface BRDF reflection only

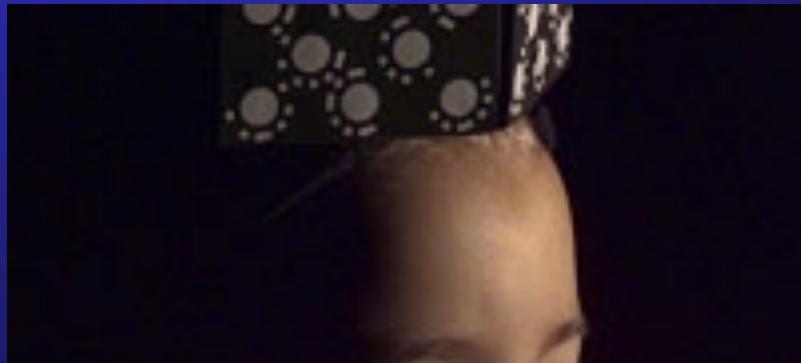


Subsurface scattering



Jensen, Marshner Levoy & Hanrahan

Subsurface Scattering Examples



Skin



Ponytail



Skim milk vs. whole milk
(looks like white paint)

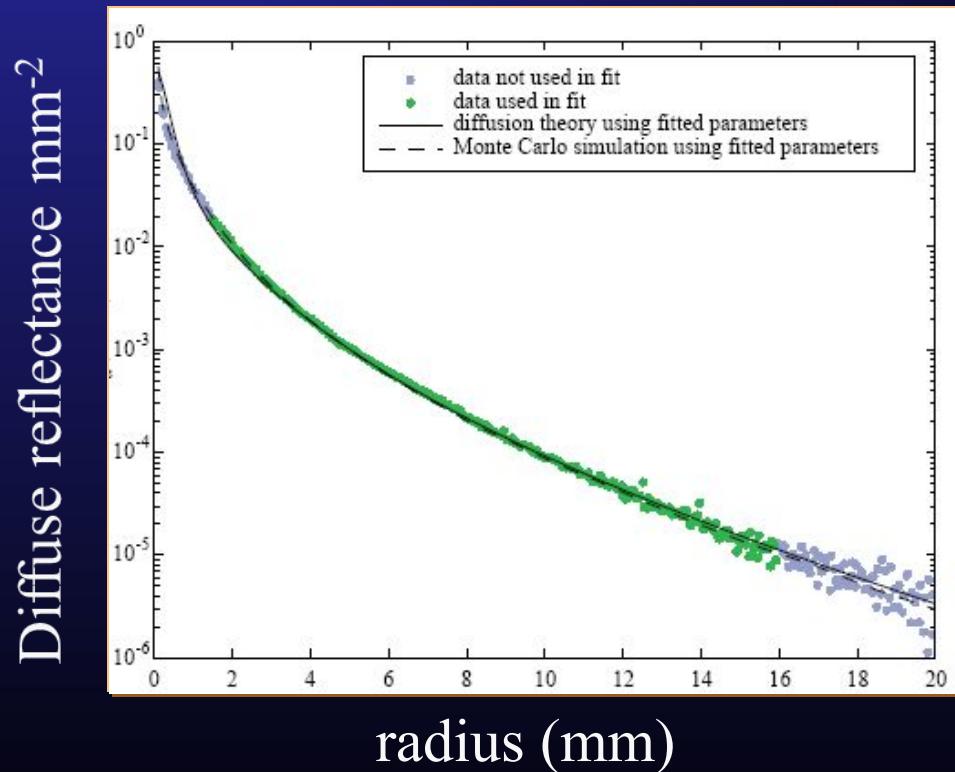
Hair close-up



Steve Marshner

Example Empirical Measurements

- From light probe; green wavelength for marble (Jensen *et al.*).
- Fit by exponential, e.g.



Fast Subsurface Scattering Rendering

- Approximate rapid fall-off empirical function with a spike plus a Gaussian blur: $1/(c+\text{radius})^{\text{power}}$
(where c avoids divide by zero and power ≈ 2).



- Use for diffuse only.
- Slower in the red channel.
- For faces, ensure enough pore-level detail.

Results Used in the Movies

Matrix Reloaded (digital doubles) and also *Lord of the Rings* (Gollum).



Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading ✓
- Mapping
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Light, Transparency, Shadow, Mapping and Shaders

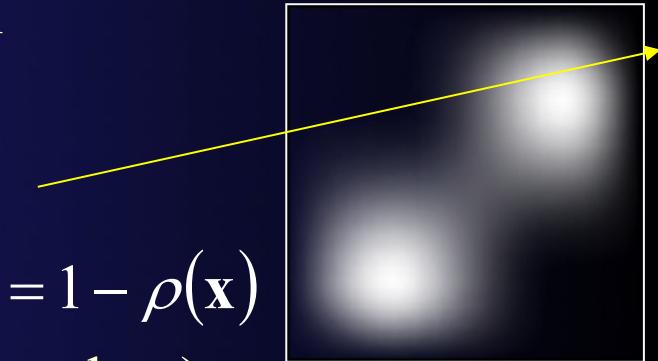
- Light transmittance:
 - Formalization.
 - Various roles for transmittance.
 - Volume rendering.
- Shadows:
 - Shadows identify light source directions.
 - Shadows emphasize contact and depth.
- Mappings (of all sorts):
 - Texture, Bump, Normal, Displacement.
 - “Baking”.
- Some shader examples.

Transmittance and Translucency

- A ray need not only see the front-most opaque surface – it might be the integration of material density over its entire length.
- For example, fog or haze (a “participating media”) acts like a depth-dependent obscurant to the surface color the ray actually hits.
- Transmittance can be used graphically in a wide variety of ways (not only as a real matter property).
- Key to volume rendering of variable density voxels.

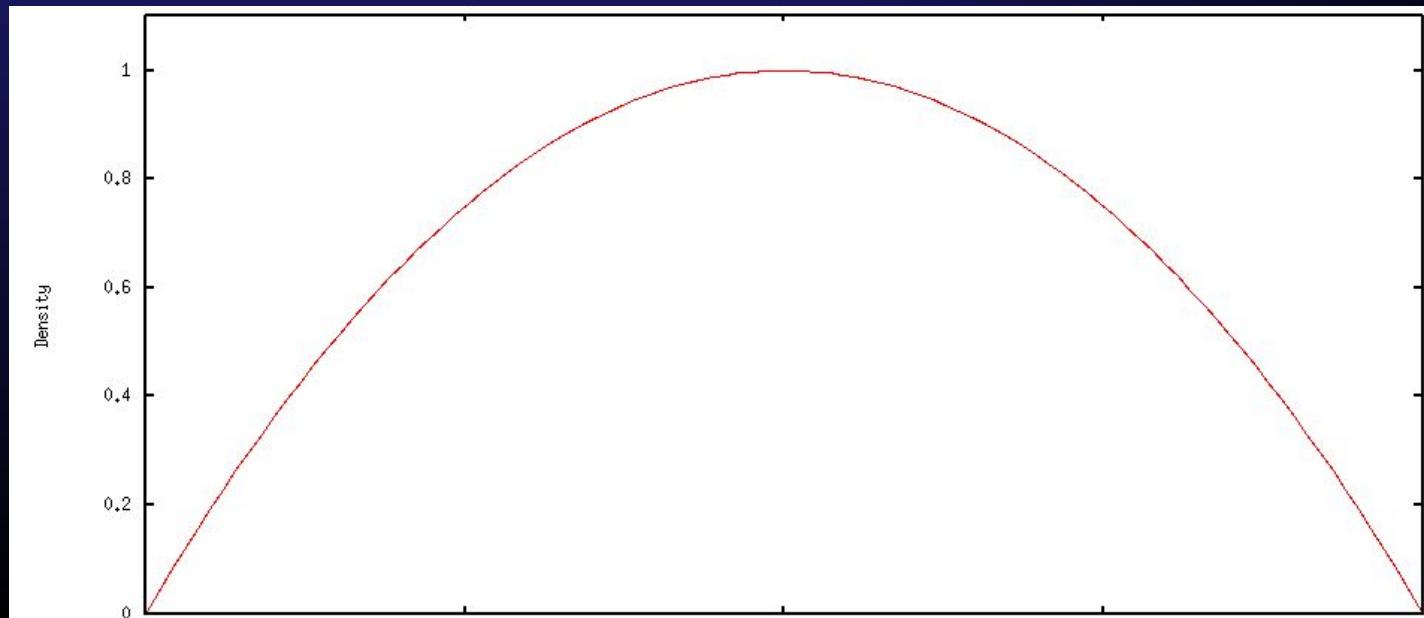
Formalization of Translucency

- Denote space $f(x, y, z)$ as $f(\mathbf{x})$, with both **density** $\rho(\mathbf{x})$ and **color** $C(\mathbf{x})$ fields.
- Consider a ray from the eye e : $r = \mathbf{x}_e + \hat{\mathbf{n}}s$
 - Transmittance (inverse of density): $\tau(\mathbf{x}) = 1 - \rho(\mathbf{x})$
 - ($\rho=1 \Rightarrow \tau=0 \Rightarrow$ opaque; $\rho=0 \Rightarrow \tau=1 \Rightarrow$ clear)
 - Attenuation along ray: $\alpha(t) = e^{-\int \tau(s)ds}$
 - (Check: $\tau=0 \Rightarrow \alpha=1$; $\tau=1 \Rightarrow \alpha=0$)
 - Attenuated color at t is $\alpha(t) C(t)$
 - Accumulate attenuated colors along ray: $C = \int \alpha(t) C(t) dt$



Example: Soft White Sphere

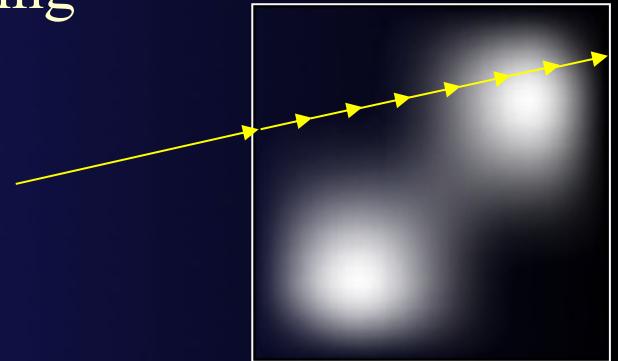
- Color everywhere in sphere: $C(p) = (1,1,1)$
- Density $\rho(\mathbf{x}) = \begin{cases} 1 - \frac{|\mathbf{x}|^2}{r^2} & \text{for } 1 - \frac{|\mathbf{x}|^2}{r^2} > 0 \\ 0 & \text{for } 1 - \frac{|\mathbf{x}|^2}{r^2} \leq 0 \end{cases}$
- Sphere has “soft” edges because density tapers off at edges:



Simplifying the Volume Integral

- Integrate ray through volume by approximating samples from a fixed step size d :

$$e^{-\int \tau dt} \approx e^{-\sum \tau_i d} = \prod e^{-\tau_i d} = \prod (1 - \alpha_i)$$



- The color of each ray segment is $\alpha_i C_i, i = 1, \dots, n$
- So:

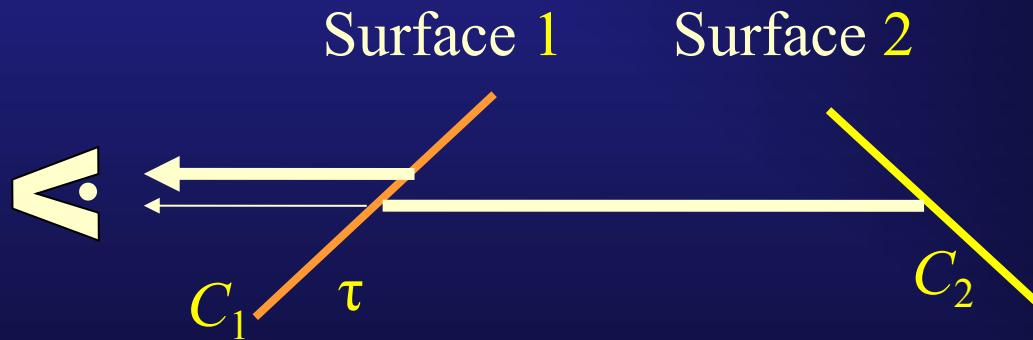
$$C \approx \sum_{i=1}^n \alpha_i C_i \prod_{j=1}^{i-1} (1 - \alpha_j)$$

- For $n = 2$ samples, this is just a back-to-front linear interpolation:

$$C'_1 = \alpha_1 C_1 + (1 - \alpha_1) C_2 \quad \text{or} \quad C'_1 = (1 - \tau_1) C_1 + \tau_1 C_2$$

A Simple Surface Translucency Model: Requires Known Back-to-Front Surface Order

- Surface 1 is translucent and only allows fraction τ (transmittance) of the light reflected from surface 2 (behind it) to pass through:



τ = transmittance of surface 1

$$C = (1 - \tau)C_1 + \tau C_2 \quad C_1 = \text{color of surface 1}$$

$C_2 = \text{color of surface 2}$

$\tau = 0 \Rightarrow$ surface 1 is opaque $\Rightarrow C = C_1$

$\tau = 1 \Rightarrow$ surface 1 is transparent $\Rightarrow C = C_2$

Transmittance can be a Function

- $\tau = f(\text{material})$
- $\tau = f(\text{depth})$
- $\tau = f(\text{normal})$
- $\tau = f(\text{thickness})$
- $\tau = f(\text{sampling})$
- $\tau = f(\text{space}) = f(x,y,z)$

$\tau = f(\text{material})$, e.g., Representing Glass; Use 2 Surface Formula

Say, $\tau \approx 0.9$.

Note black tiles are whitened

[white=(1,1,1);
black=(0,0,0)]:

$$(1-0.9)(1,1,1) +$$

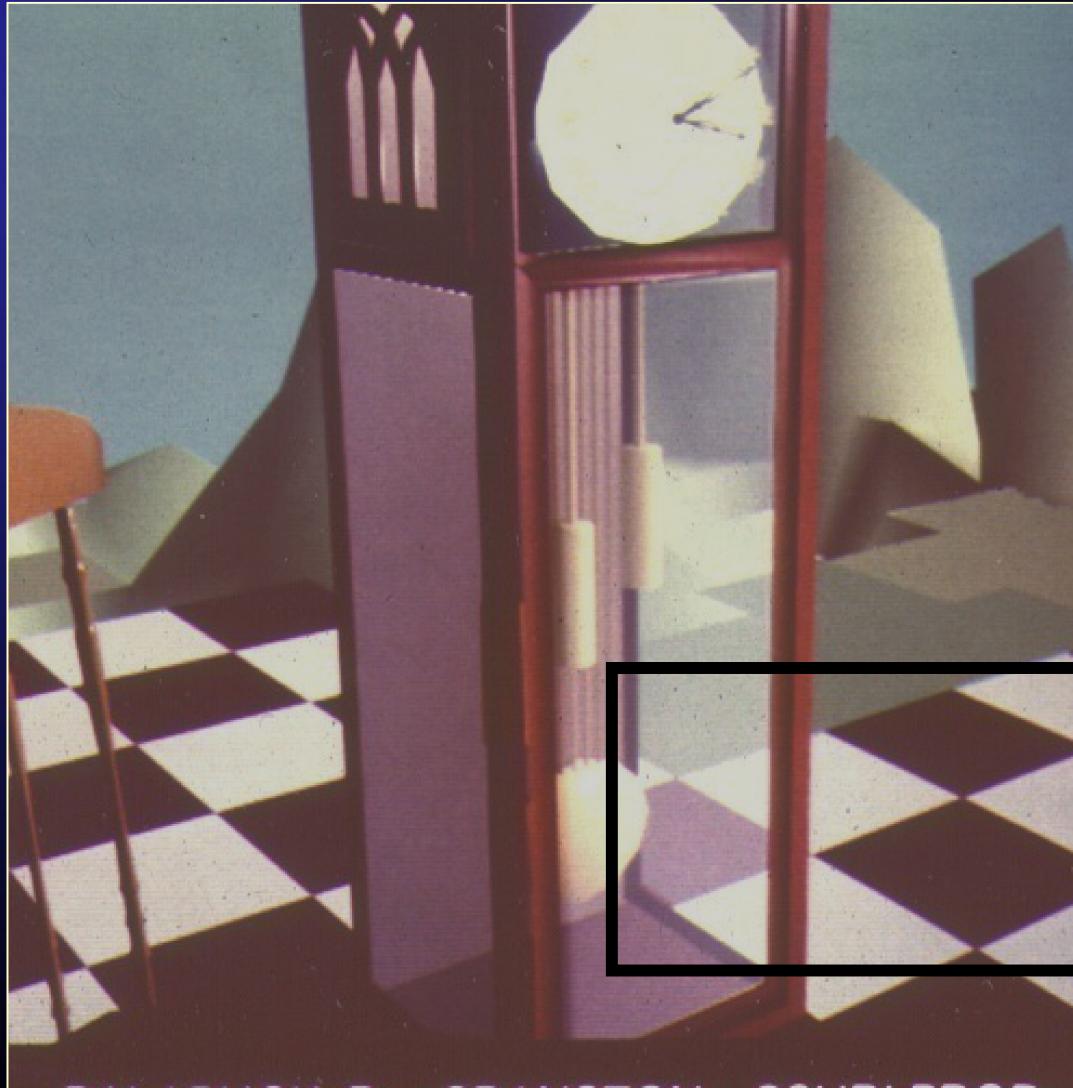
$$0.9(0,0,0) =$$

(0.1,0.1,0.1) (grey);

white tiles stay white:

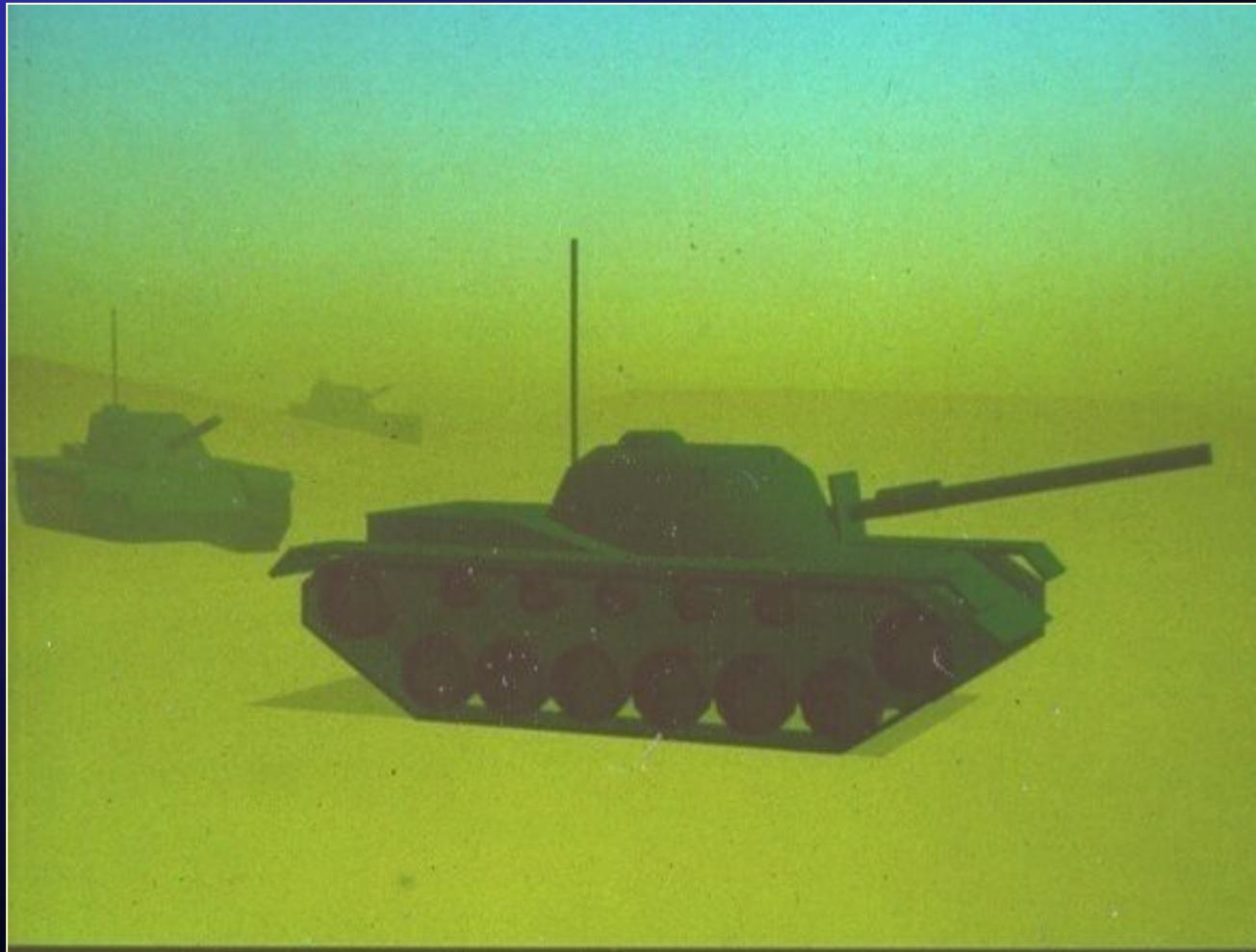
$$(1-0.9)(1,1,1) +$$

$$0.9(1,1,1) = (1,1,1)$$



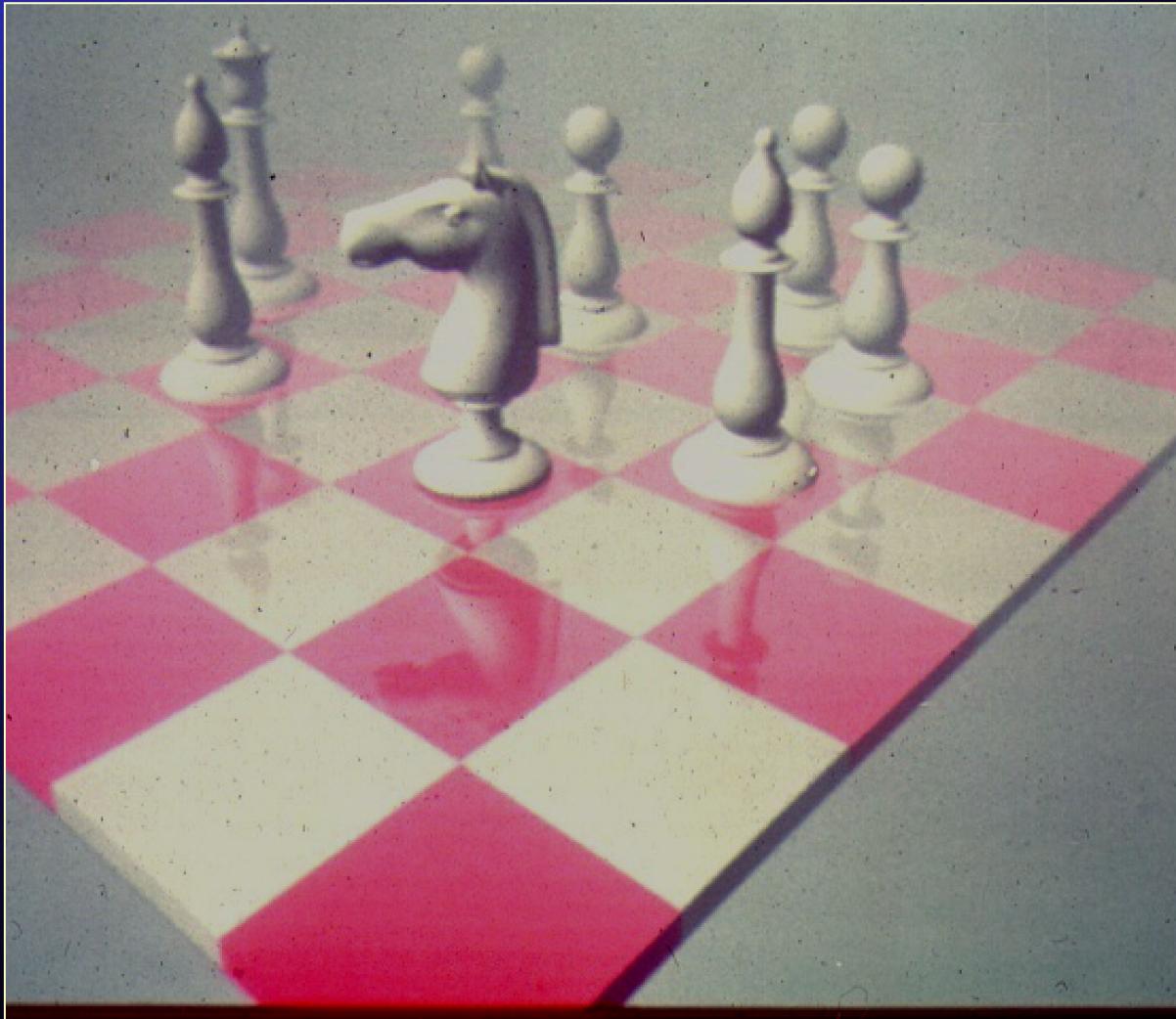
BALABUCK, R.—CRANSTON—CSURI PROD.

$\tau = f(\text{depth})$, e.g., Fog or Haze; Use 2 Surface Formula with $C_1 = \text{fog color}$ and $\tau = (\text{maxdepth} - \text{depth}) / \text{maxdepth}$



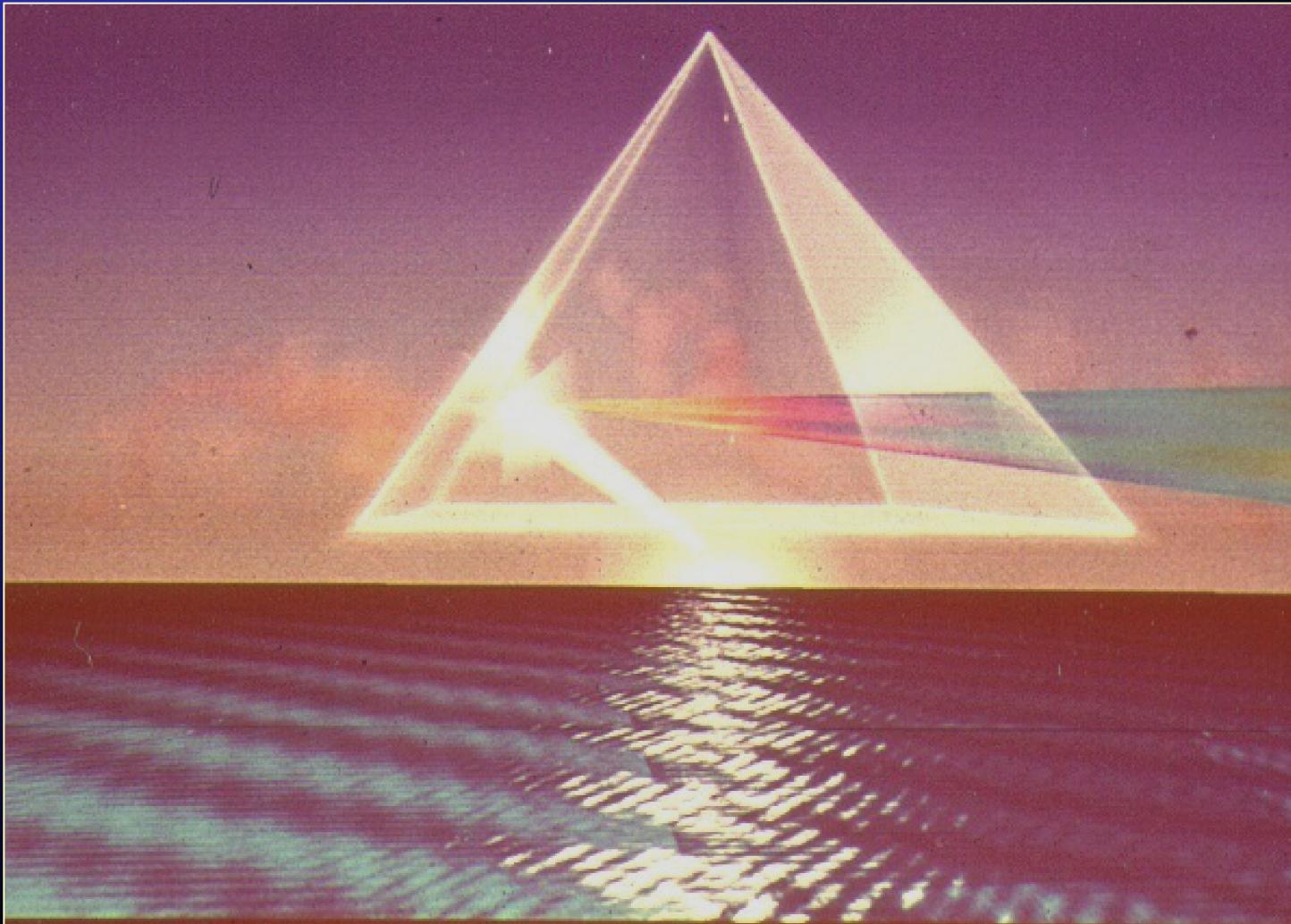
© 1983 STIPE, D.—ADVANCED TECHNOLOGY SYSTEMS

Foggy Chessmen (a Cheat)



WHITTED, T. & WEIMER, D.—BELL LABS

$\tau = f(\text{normal})$, e.g., to Show Surface Shape;
 $\tau = |\text{normal} \cdot \text{viewdirection}| / \|\text{viewdirection}\|$

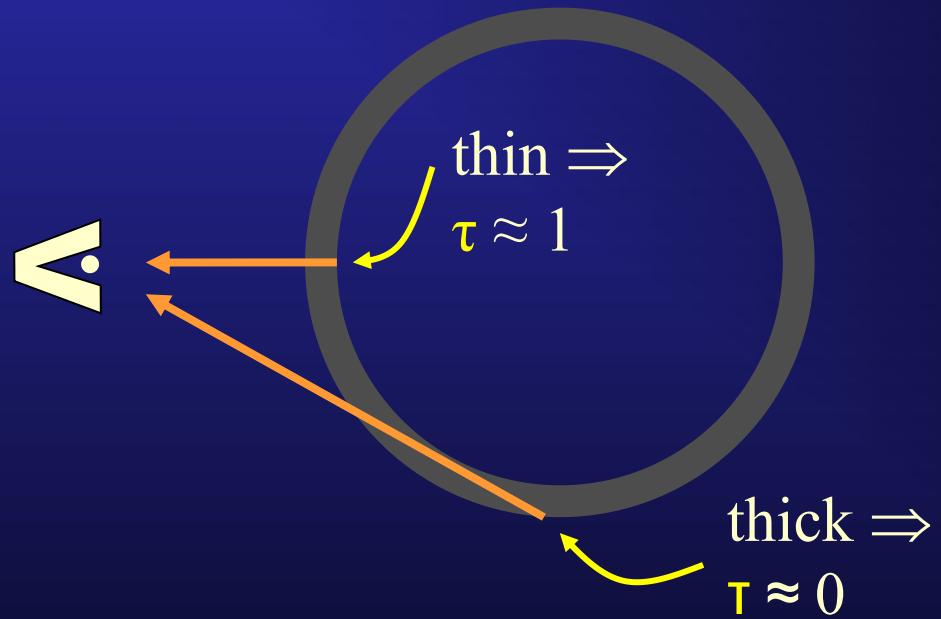


“Eye Candy”: Changing Transmittance by Lambert’s Law



MIRALABS, UNIVERSITY OF MONTREAL

$\tau = f(\text{thickness})$ of Material Relative to Viewpoint



Thus: $\tau = a |(N \bullet E)| + b$ where $a + b \leq 1$

a is “native” transmittance

$$0 \leq b \leq (1-a)$$

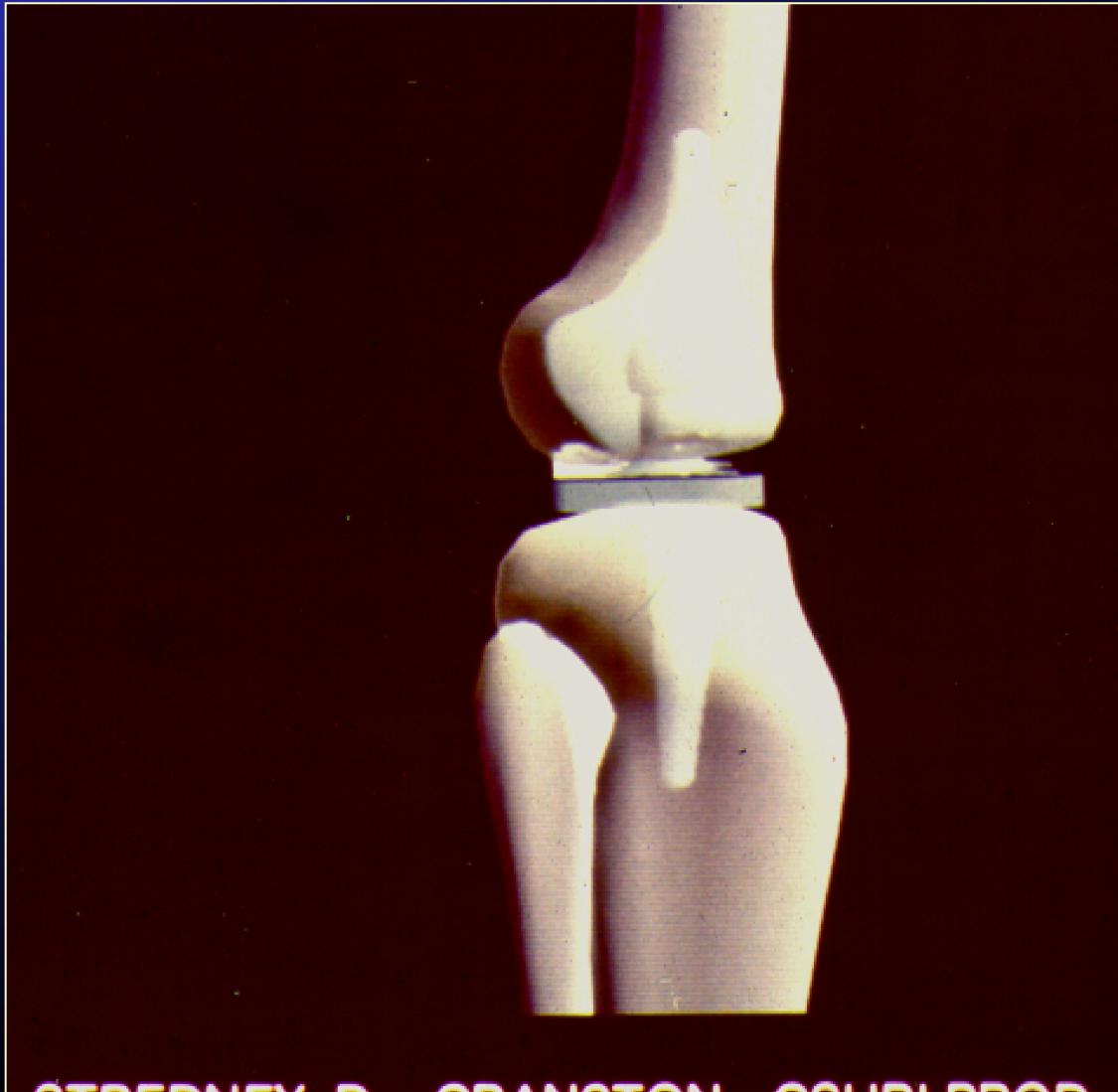
(thick .. very thin)

Closer to Real Glass



NAGOYA UNIVERSITY—SHIGEKI YOKOI

Transparency Used to See Internal Structure
(of otherwise opaque surfaces)

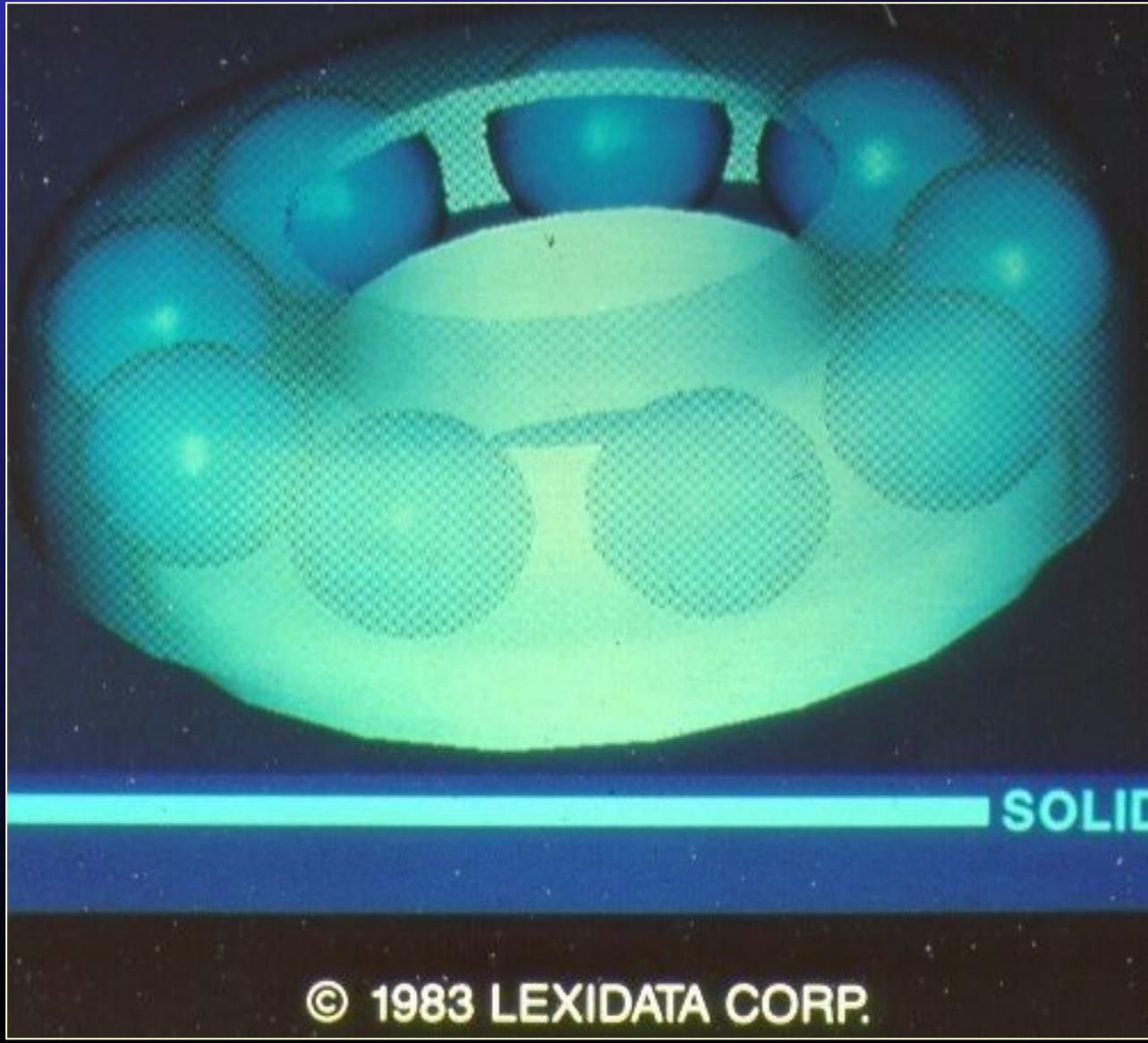


STREDNEY, D.—CRANSTON—CSURI PROD.

Seeing Transparency: Specular Highlights Reduce Transmittance



$\tau = f(\text{sampling})$: Dithering; e.g. % of pixels rasterized per polygon = $(1 - \tau) \times 100\%$



$\tau = f(\text{space})$: Varies over Volume $f(x,y,z)$; e.g.,
Surface of Ellipsoid Shapes



© 1985 G. GARDNER — GRUMMAN CORP.

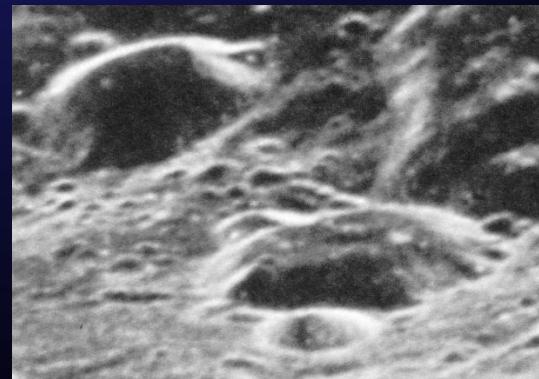
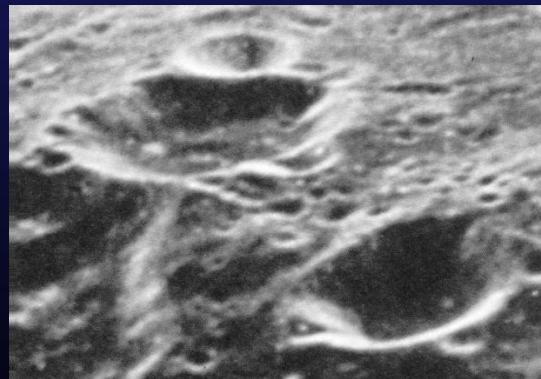
Using a Denser Transmittance Function
($\tau \approx 0$ for clouds; $\tau \approx 0.1$ for trees; $\tau = 0$ for ground)



© 1985 G. GARDNER — GRUMMAN CORP.

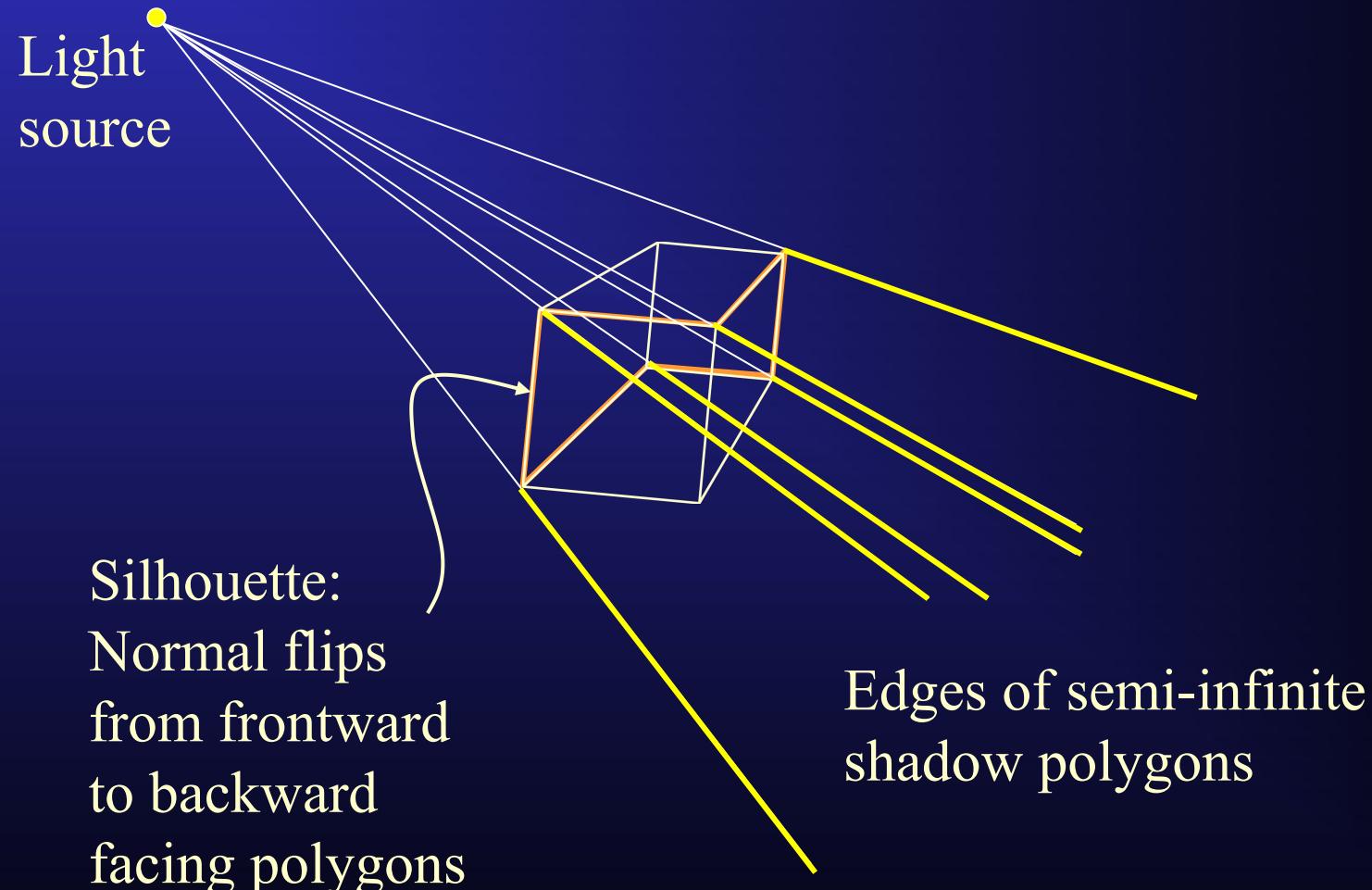
Shadows

- Shadows are a consequence of light and substance.
- Shadows help depth perception and spatial relationships.*
- Shadows can be hard- or soft-edged.
- Various shadow algorithms:
 - Finding silhouette edges and creating shadow polygons
 - Ray tracing
 - Radiosity

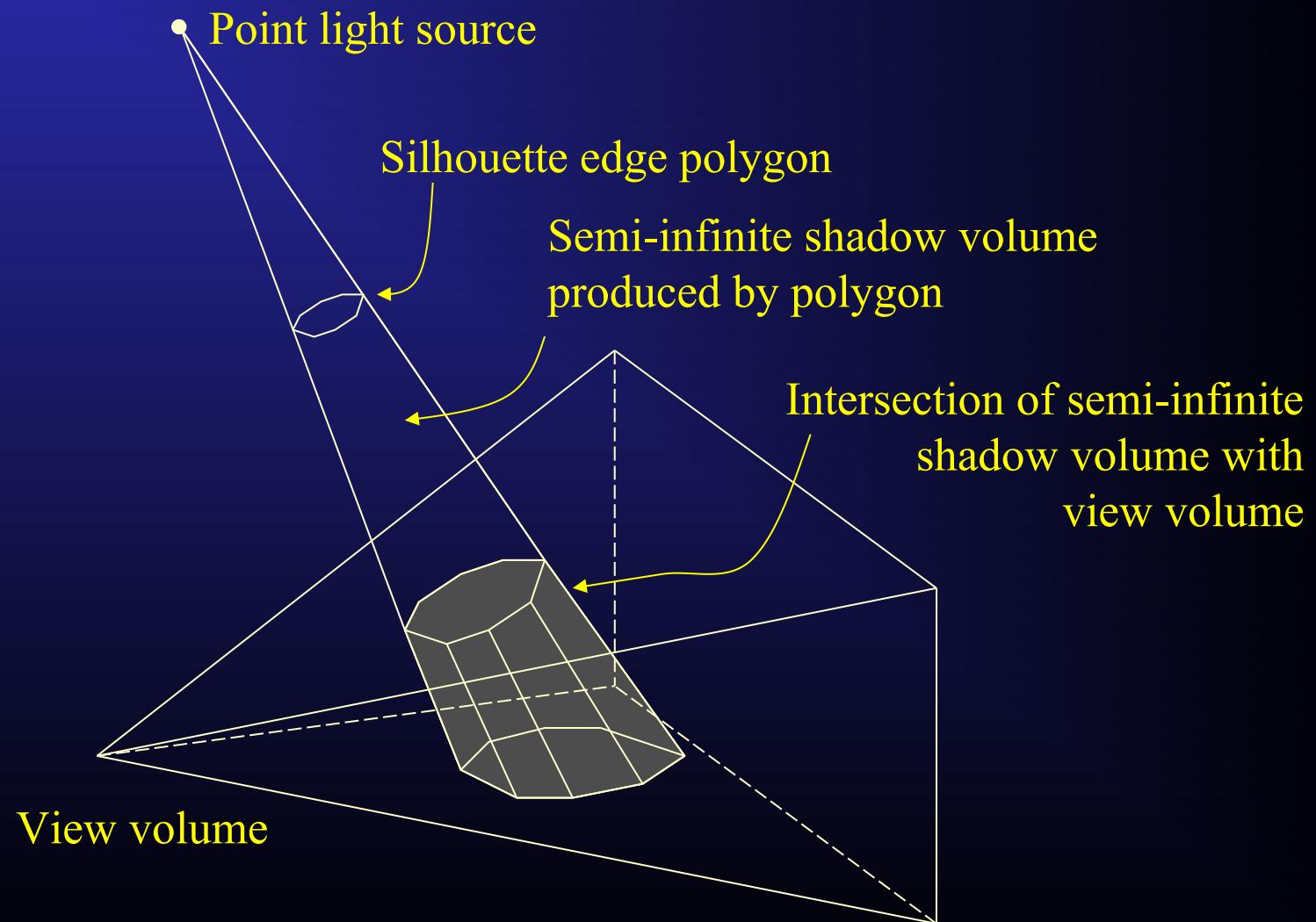


*These are the exact same images, only one is rotated 180°!

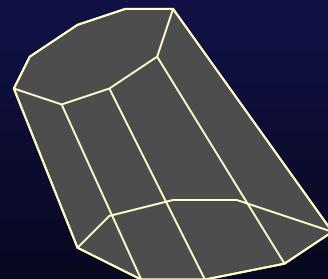
Shadow Volumes for Non-Ray-Traced Renderings



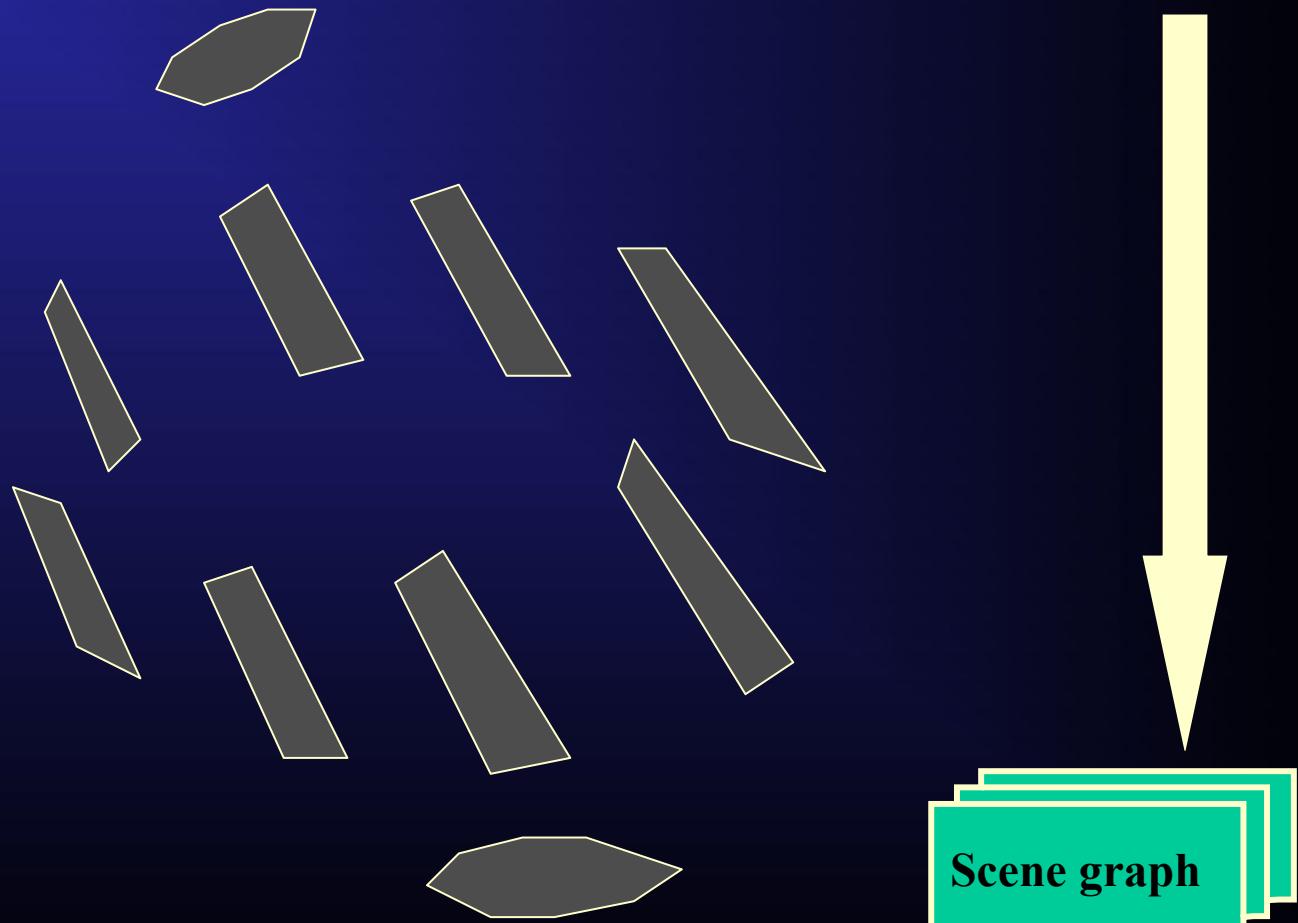
Clipping Shadow Volumes to the View Pyramid to Form Shadow Polygons



Clipped Shadow Volume Polygons Added to Scene



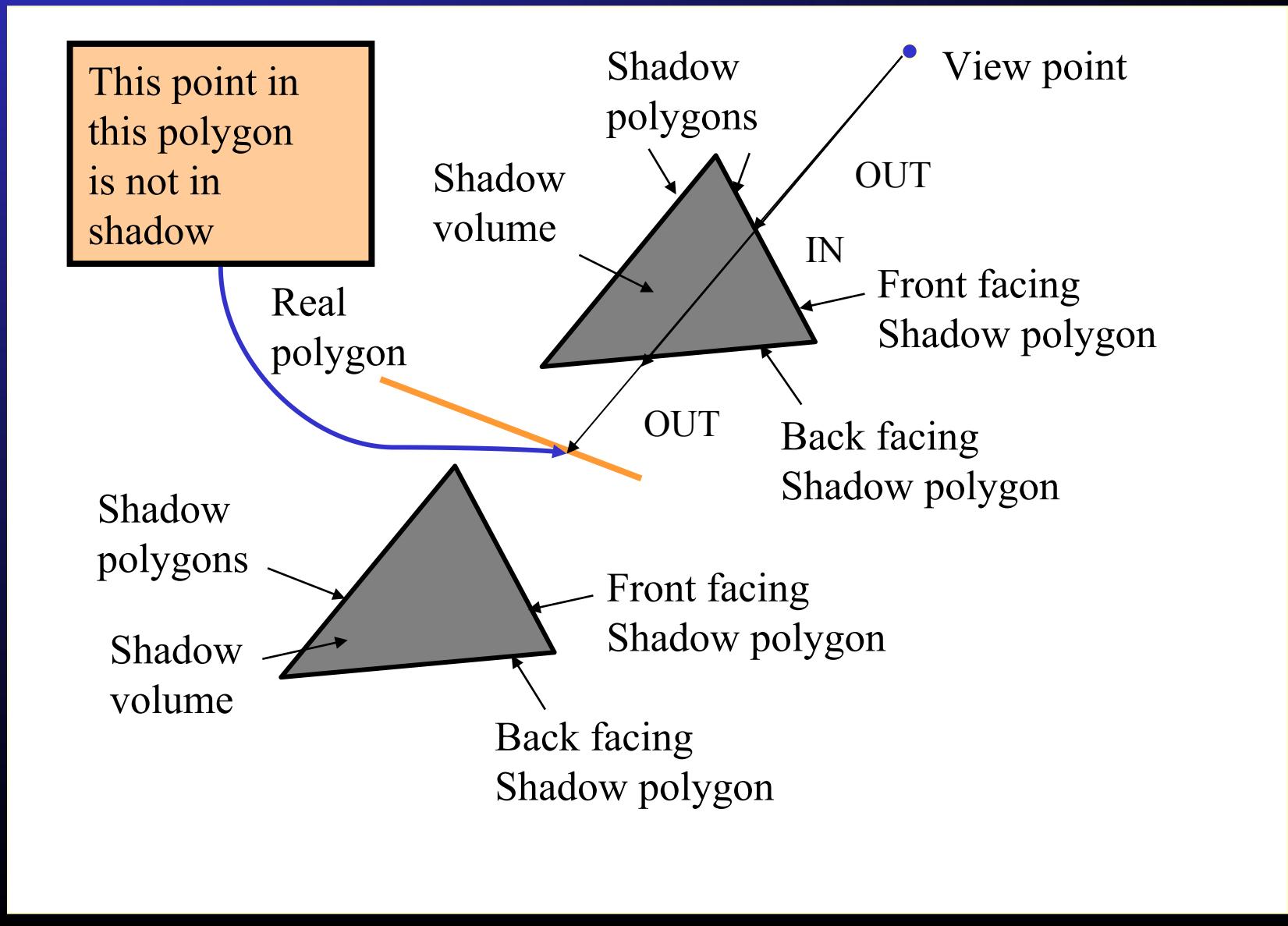
These Shadow Polygons (exploded view) –
Are Added into the Scene Graph



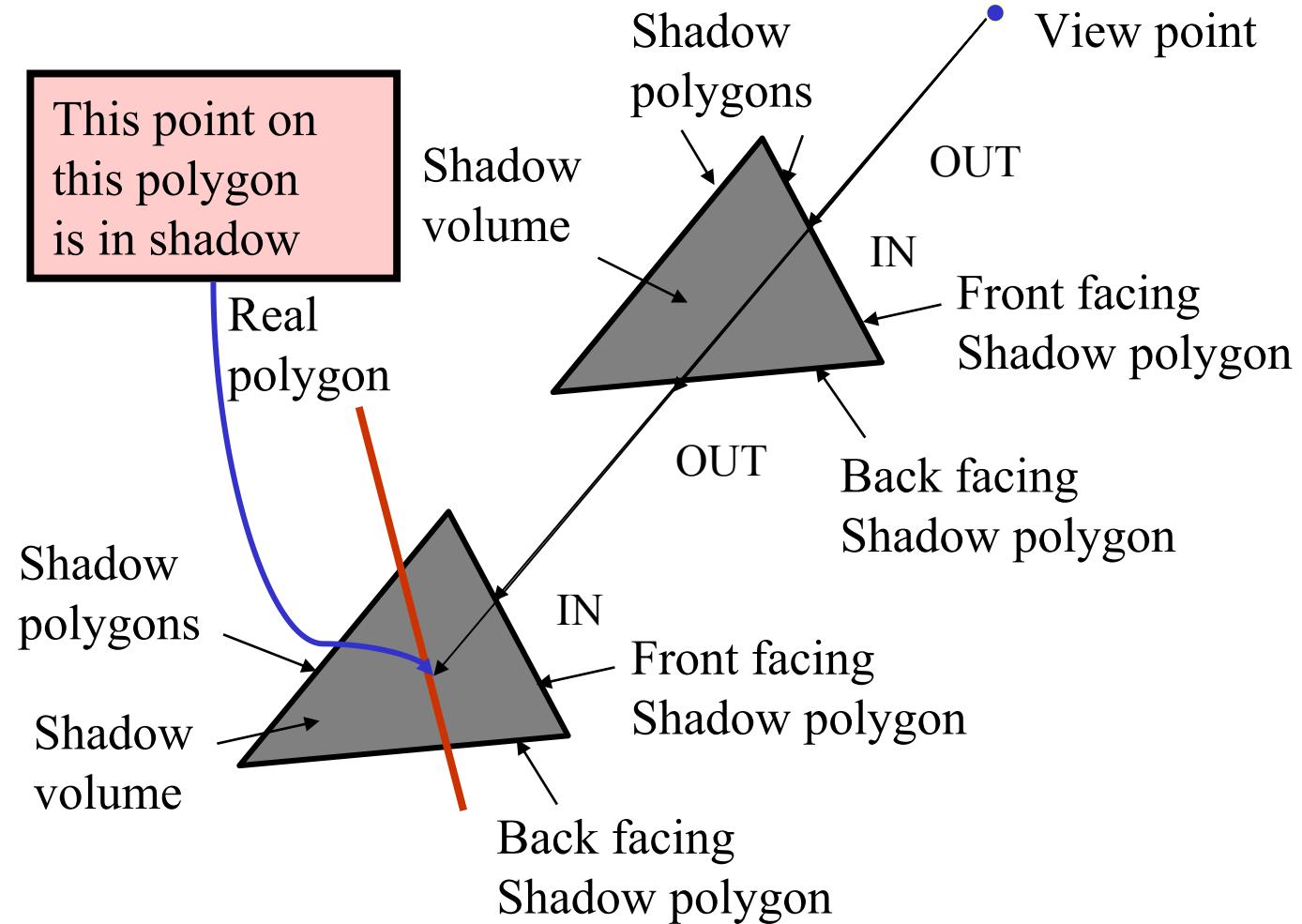
Shadow Polygons Processed for Visibility Along with Real Polygons

- If a real polygon is not in any shadow volume, then use the local illumination model to compute polygon color.
- If a real polygon is inside a shadow volume, then the light source that generated that shadow volume is **not** used in the local illumination model to compute polygon color.
- While this is easy to do in a simple *raycast*, that's rather expensive for just rendering a *polygon mesh*.
- Shadow polygons can be easily interpreted in a *scan-line algorithm*.

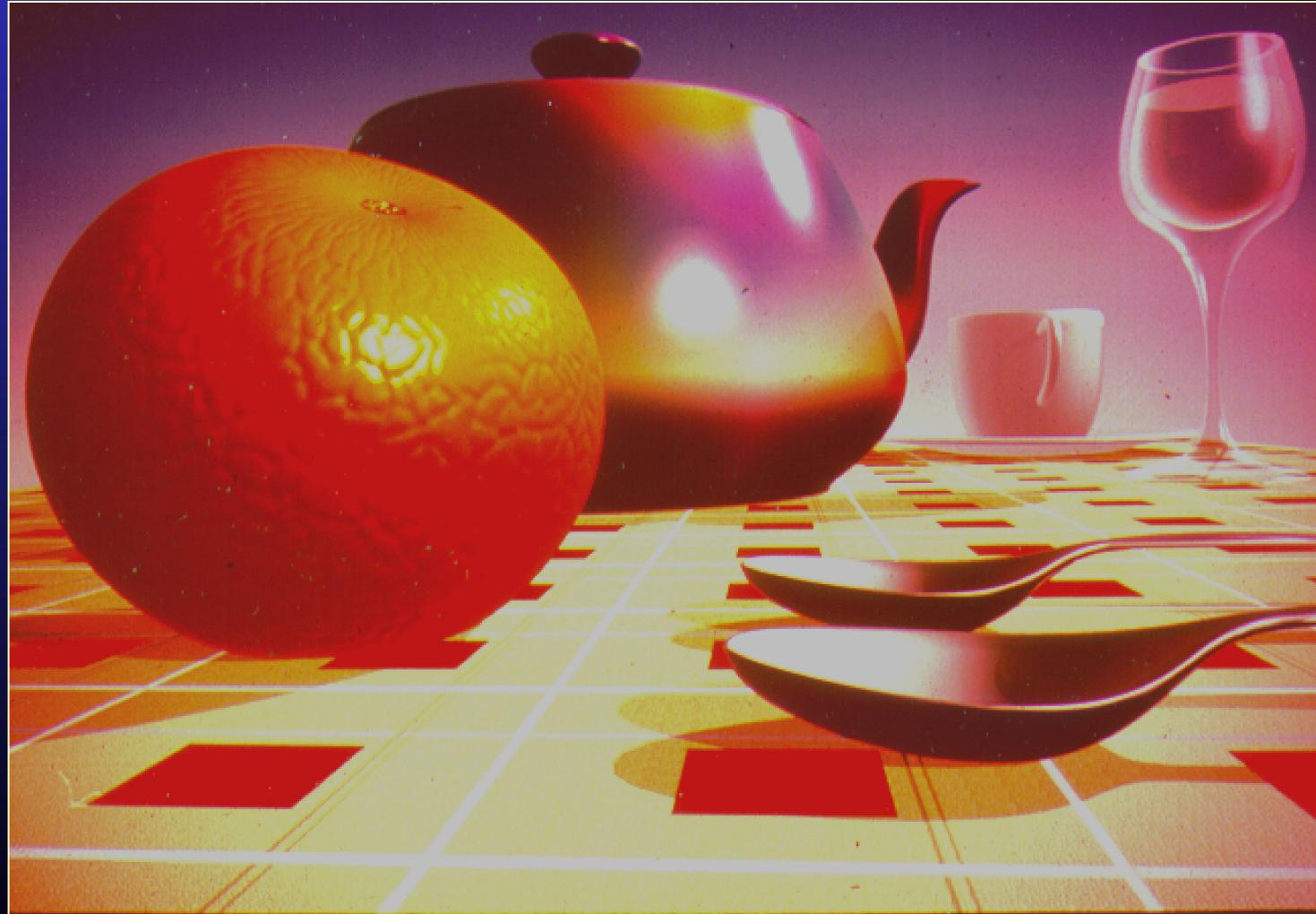
Does View See Visible Polygon in Shadow? (Active edges in a scan line algorithm)



Does View See Visible Polygon in Shadow? (Active edges in a scan line algorithm)

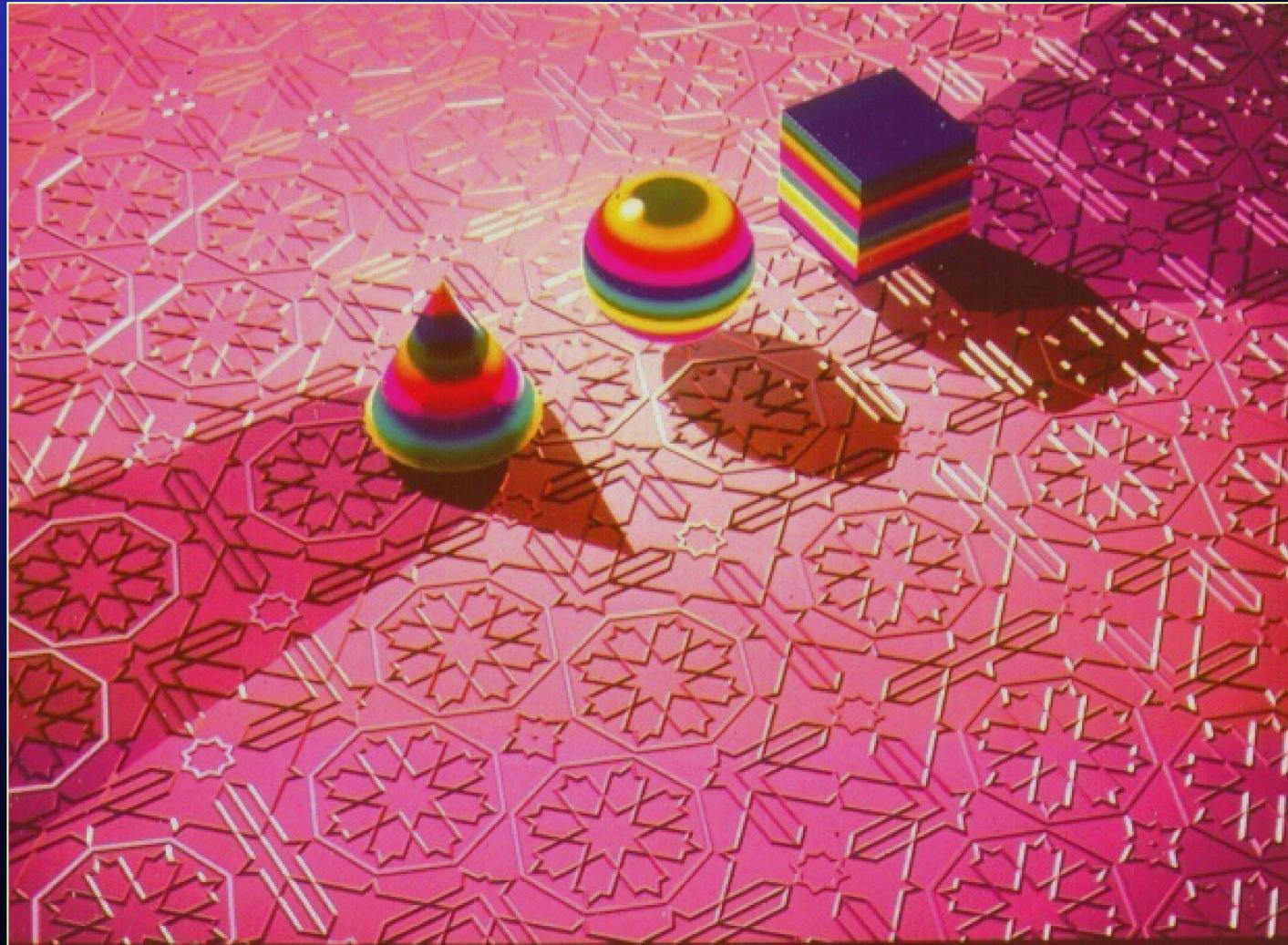


Shadows Help Show Contacts



INFORMATION INTERNATIONAL INC. COPYRIGHT 1981

Shadows Show Us Where Lights Might Be



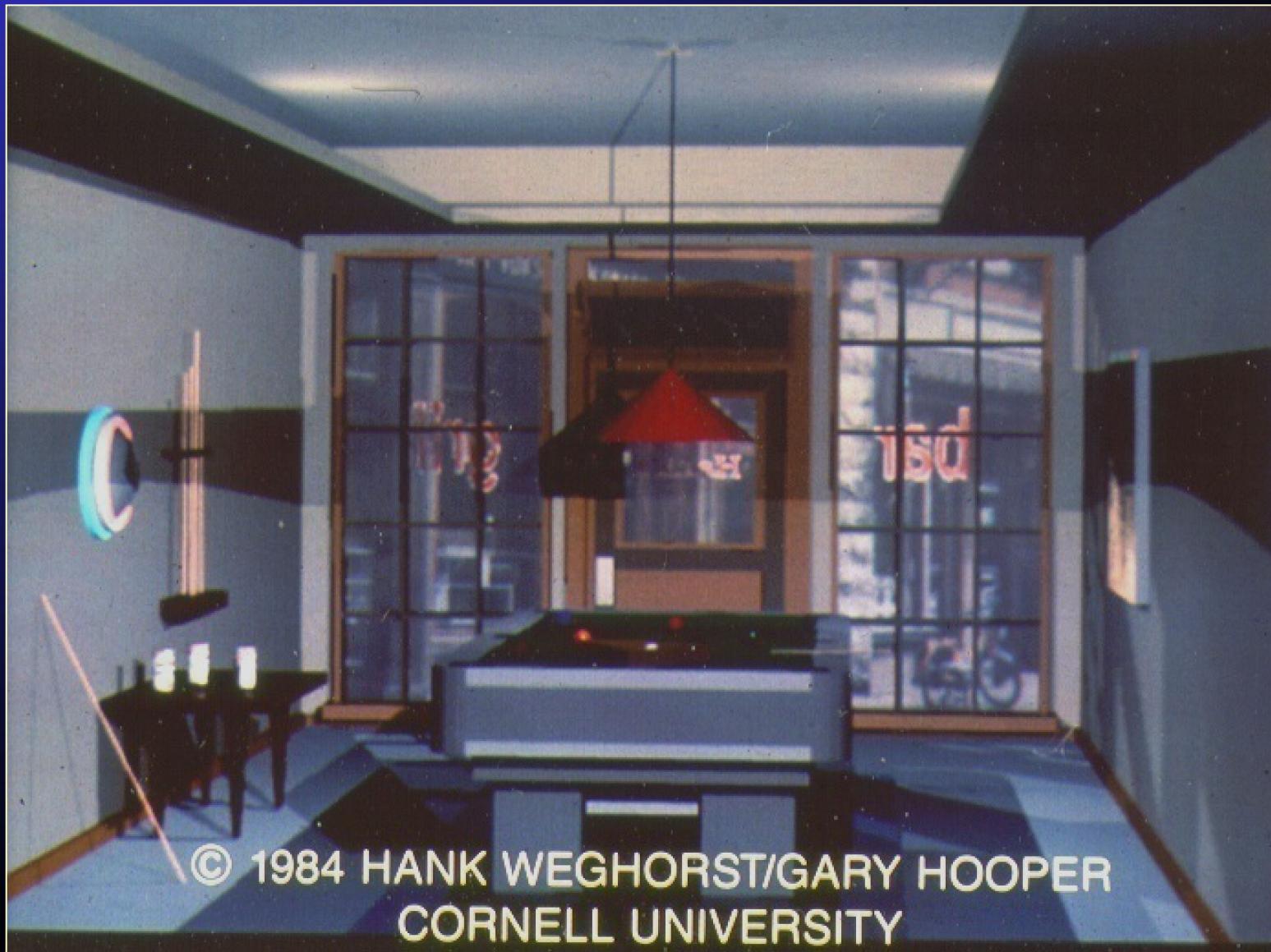
INFORMATION INTERNATIONAL INC. COPYRIGHT 1981

Shadows Accent Otherwise Like Color Features



DICK LUNDIN © 1986 NYIT

Light and Shadow over the Edge of Believability



© 1984 HANK WEGHORST/GARY HOOPER
CORNELL UNIVERSITY

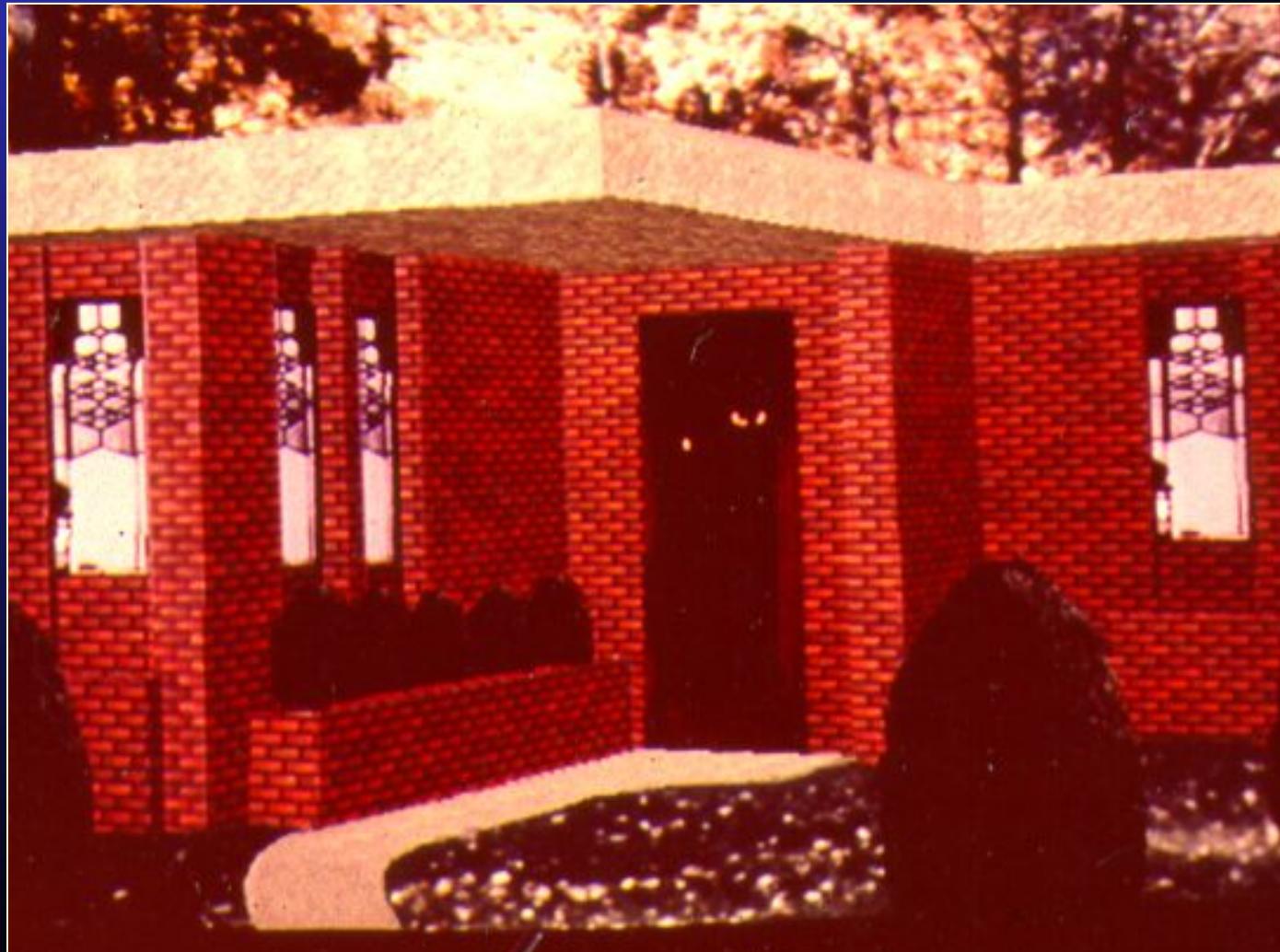
Mapping

- *Mapping* is an extraordinarily useful concept in computer graphics.
- Mapping means taking a 2D (or sometimes 3D) function and applying it to any of the **attributes** of an object or object surface.
- Maps can be **explicit arrays** of values (such as 2D images) or **procedurally-defined functions** $F(u,v)$.
- Maps can modify colors, transmittance, reflective properties, shape, etc.
- We'll see some of these...

Texture Mapping

- Advantages:
 - Map 2D texture onto clipped, visible parts of a 2D surface.
 - No great overhead in visible surface processor (except for anti-aliasing -- which is handled through a variety of techniques).
 - Easy texture definition.
 - Extends to 3D texture using volume (voxel) maps or procedural textures.
- Disadvantages:
 - Concern for correct anti-aliasing.

Example Showing Efficiency of Texture Maps over Detailed Geometric Modeling



CORNELL - E. FEIBUSH, M. LEVOY, R. COOK

Texture Mapping

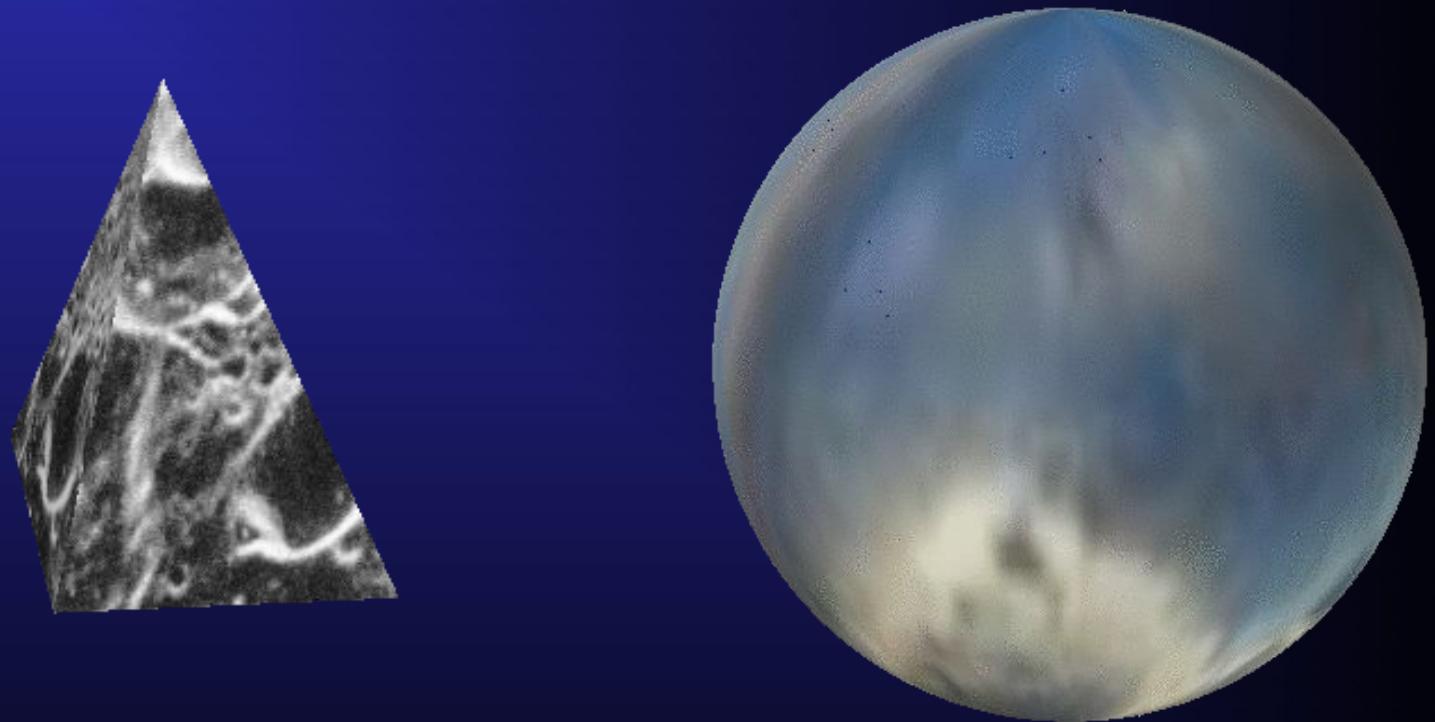
- Source: any 2D image, drawing, photo, procedural pattern.
- Target: any surface expressible as a function of two parameters:
 - spheres (latitude, longitude)
 - polygons (u,v)
 - curved patches (s,t)
 - cylinders or sweeps (height, radius(height))
 - polygon extrusions (boundary, axis)
 - etc.
 - Demo: 
- If textures are in a 2D array, the individual values are called **texels** (texture elements).

Texture Mapping Examples

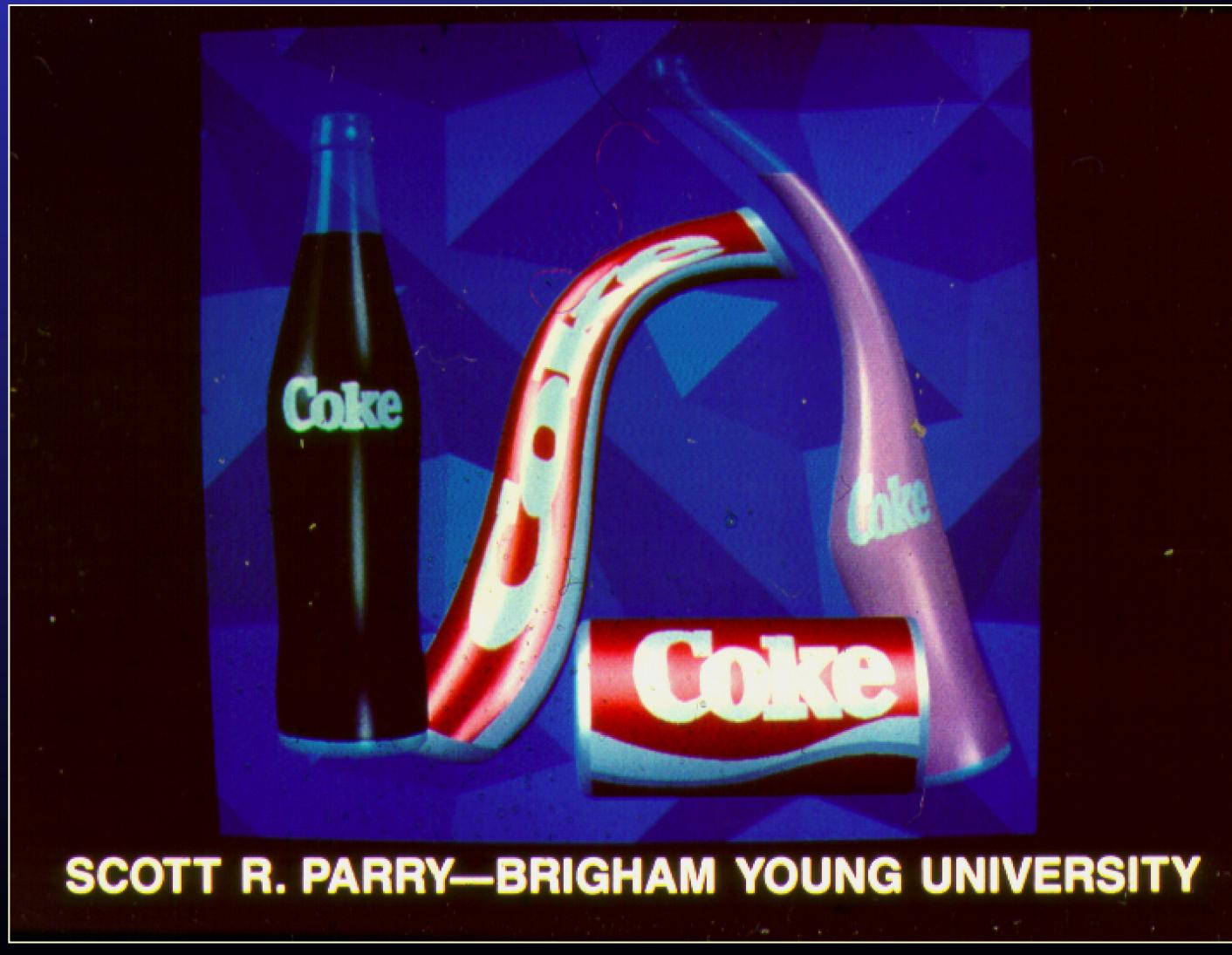


All based on same 2D checkerboard texture

Texture Mapping Example



Why use a Texture Map?
Texture is Rigidly Attached to Geometry!



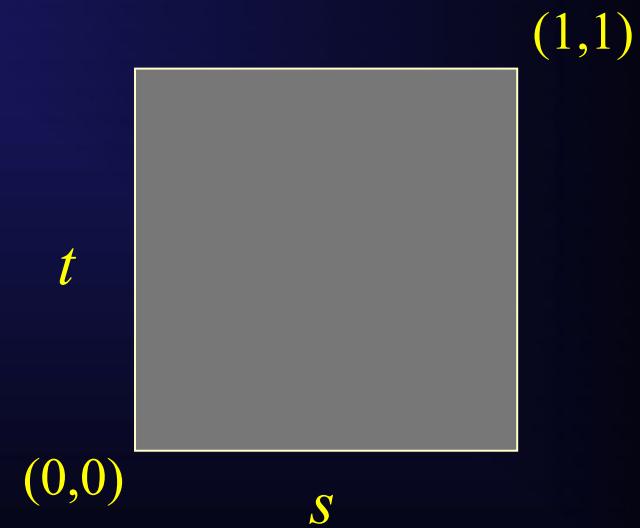
SCOTT R. PARRY—BRIGHAM YOUNG UNIVERSITY

A Very Simple Example of Texture Mapping

- Texture is image (u,v) with 100×100 texels (discrete values)
- Polygon is unit square (s,t) with $0 \leq s \leq 1$ and $0 \leq t \leq 1$ (floats)
- Texture mapping: $s = u/99$; $t = v/99$
- Inverse texture mapping: $u = \text{round}(99 s)$; $v = \text{round}(99 t)$

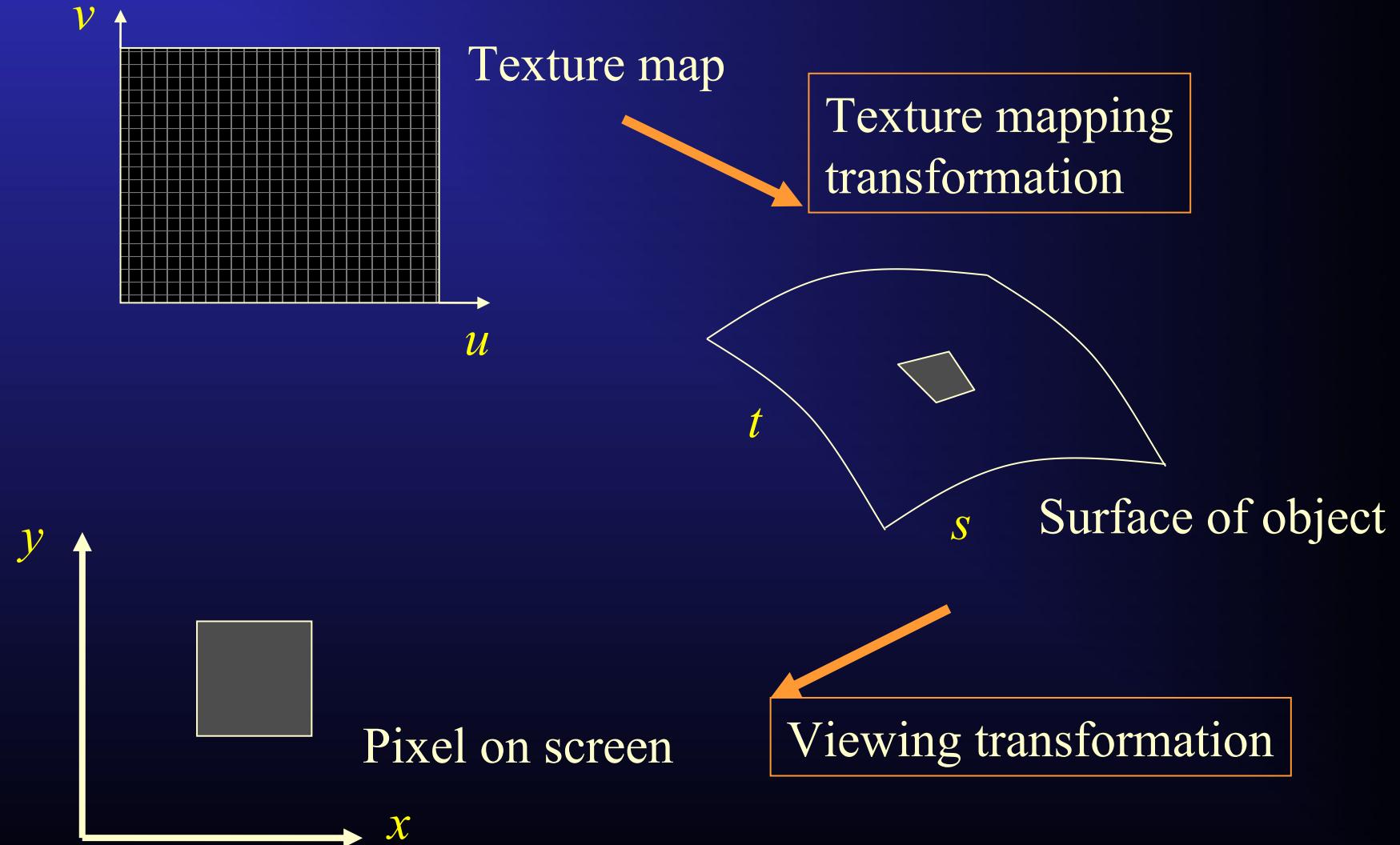


Texture (u,v) coordinates

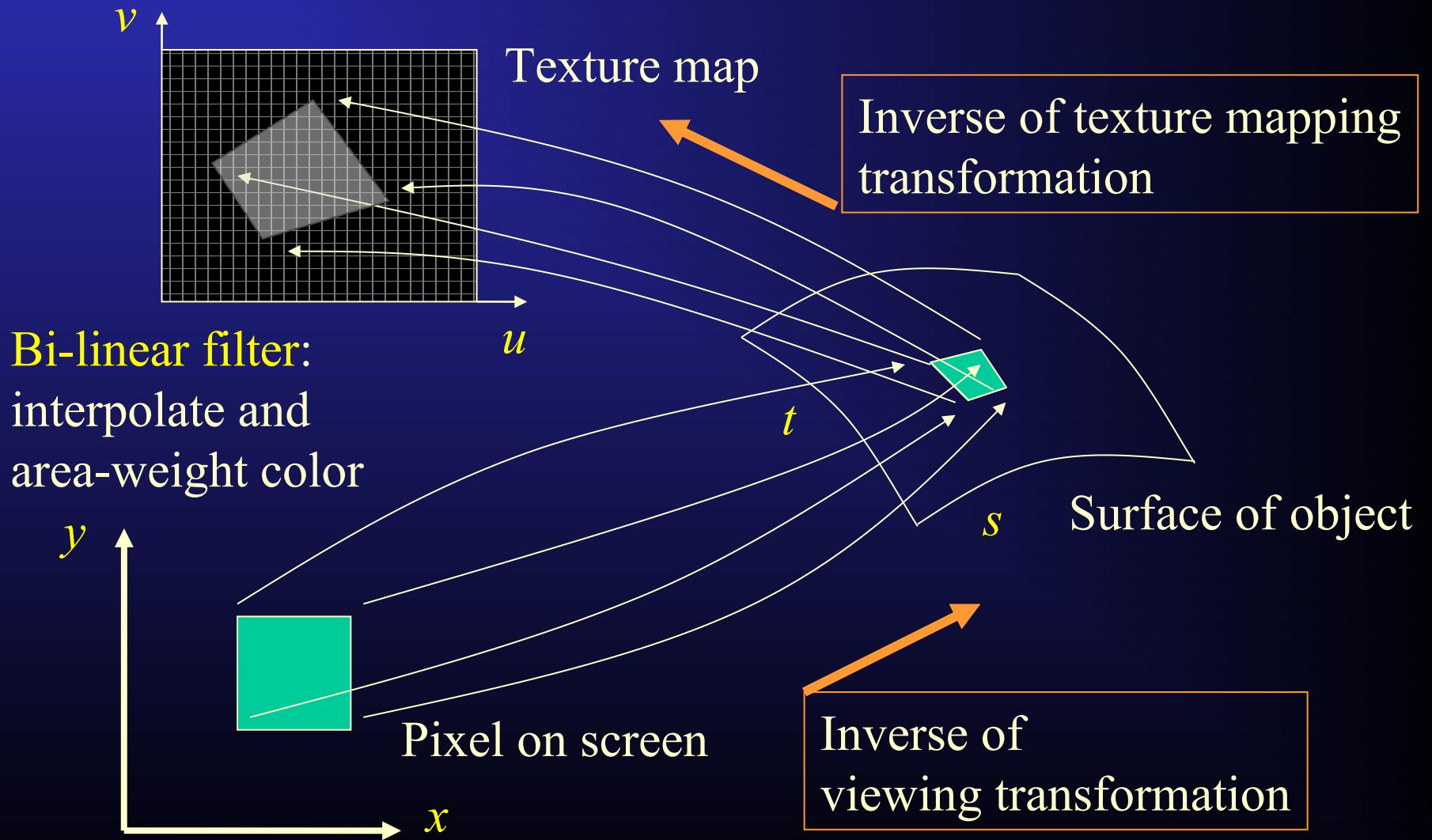


Polygon (s,t) coordinates

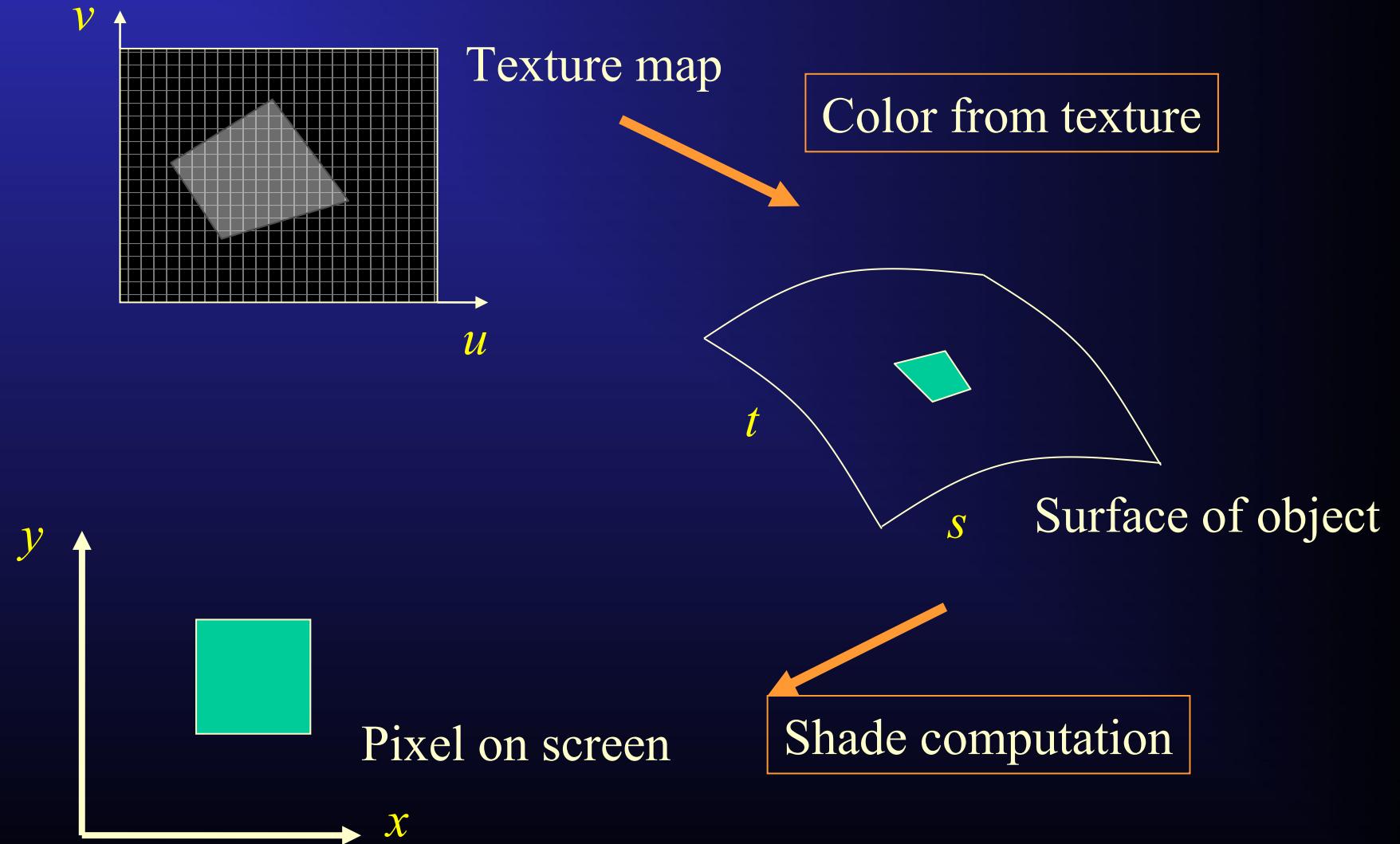
Texture Mapping Concept



Texture Mapping Computation



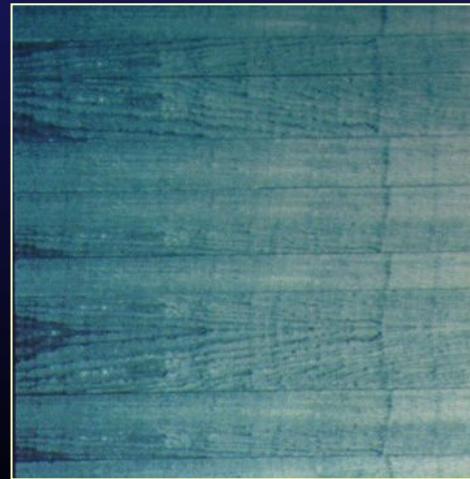
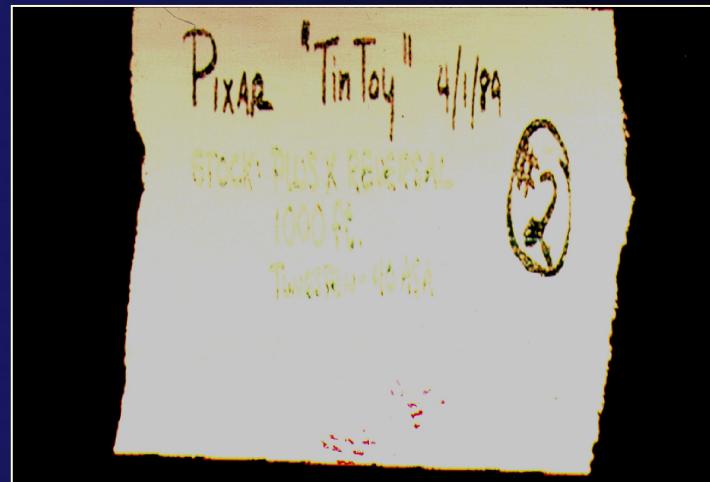
Mapping Texture Color Back to Screen



Textured Map Scene

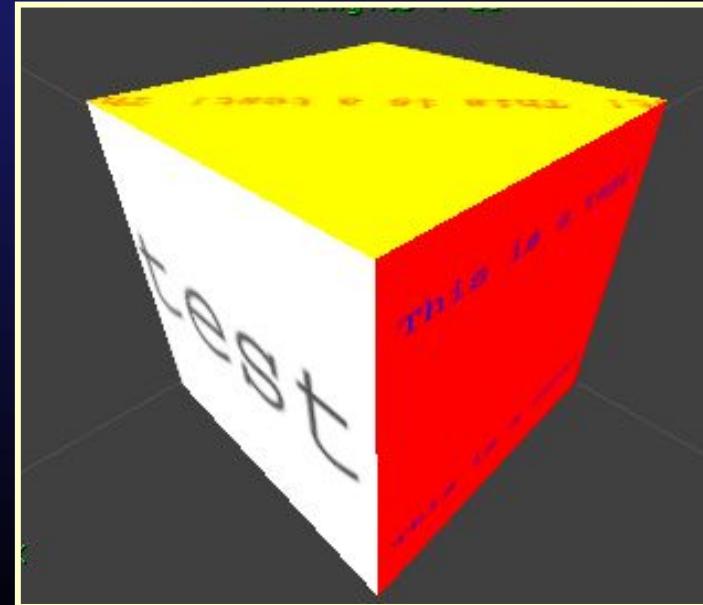
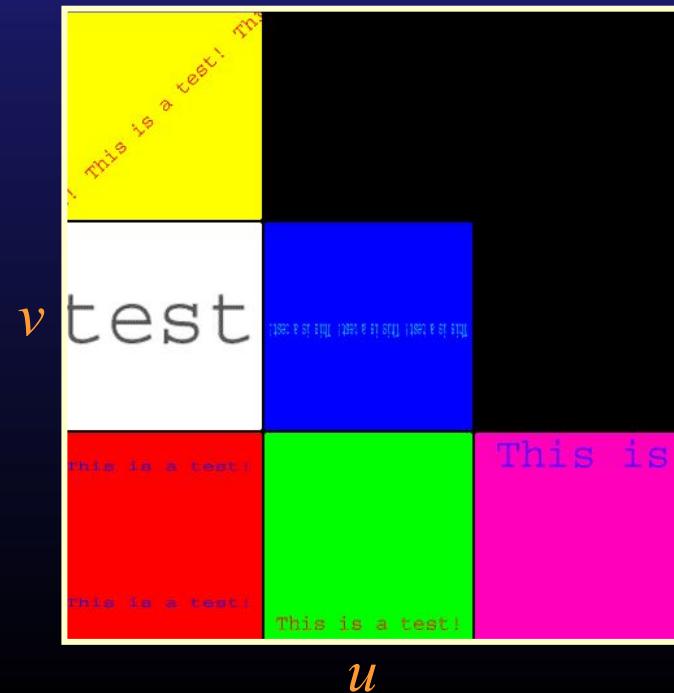


Some of the Textures Used



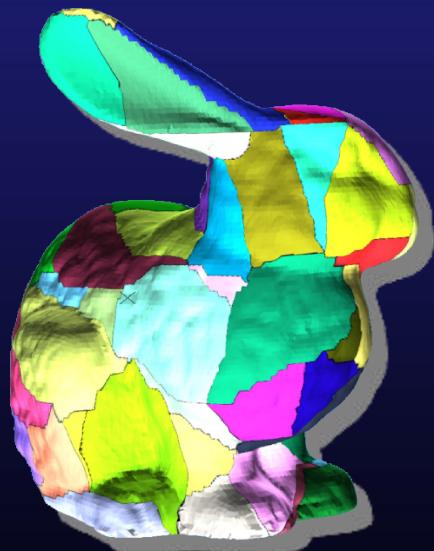
Texture Atlas

- Texture maps as images only need to have as much resolution as needed for the surface size they are mapped to.
- So for a surface mesh of many polygons, e.g., it is awkward to have a separate texture image (array) for each polygon.
- Therefore, pack the textures together into a *texture atlas*.
- Just need (u, v) offsets for each polygon.

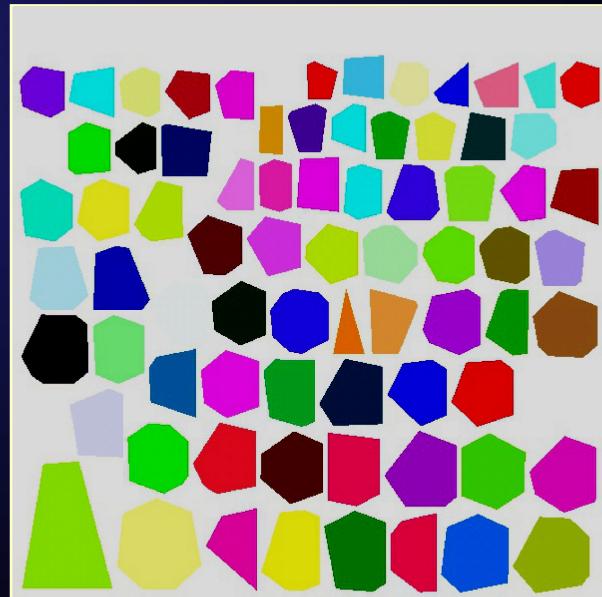


Multi-Face “Chart” Atlas

- In a large mesh, may not want to fix one texture to each face, so use a group of faces.
- Helps for varying level of detail in model while preserving texture.



Bunny model divided into charts.



Pack charts into atlas.

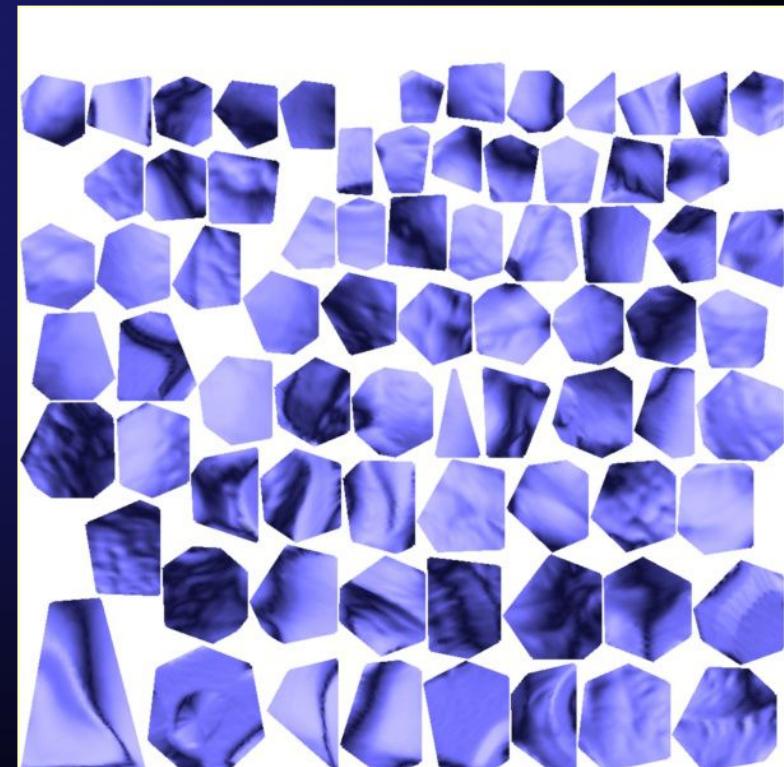
Texture Charts

- Insert any sort of texture map into each chart:

Color

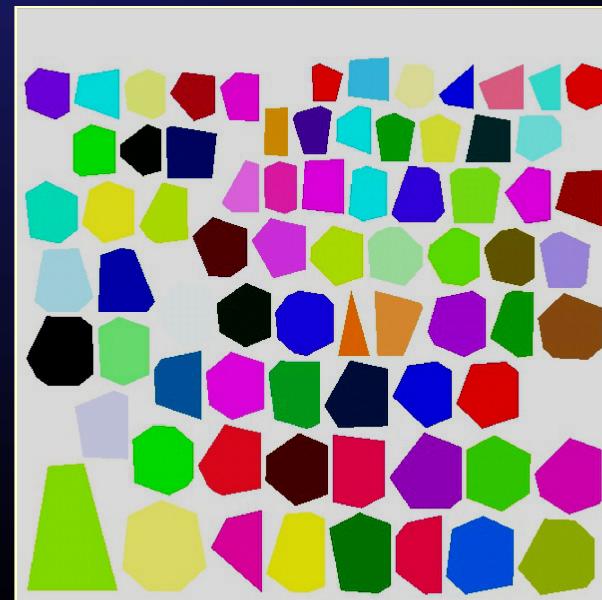


Normal

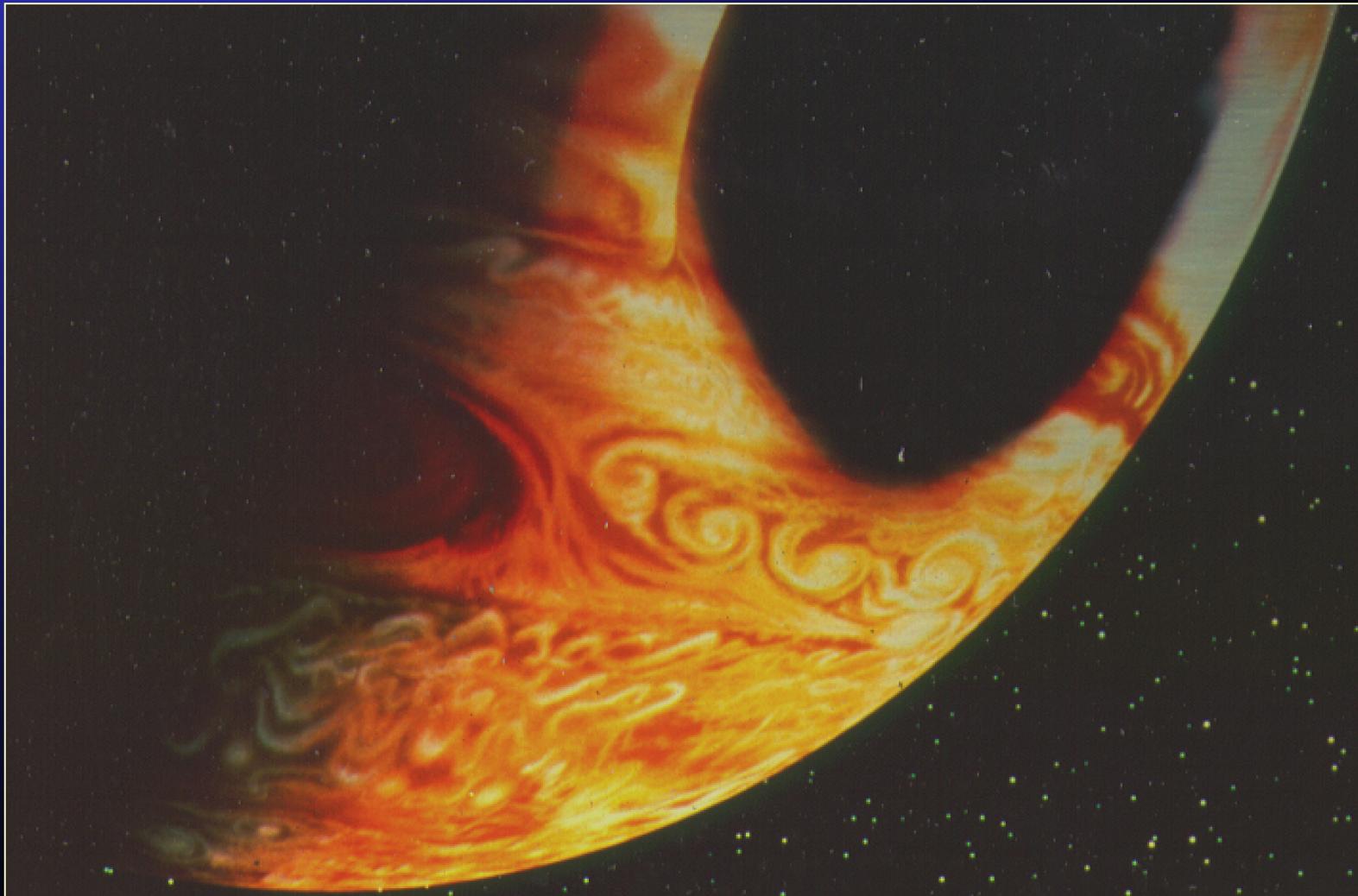


Packing Charts

- NP-Hard problem to optimize chart packing (minimize waste).
- Heuristic (similar to Igarashi '01):
 - Calculate bounding rectangle
 - Make rectangle “vertical”
 - Sort rectangles by height
 - Place largest to smallest, left-to-right, then right-to-left
(going bottom to top)



Using an Animated Texture Map



DIG SCENE SIM/D.P. FOR 2010/MGM UA © 1984

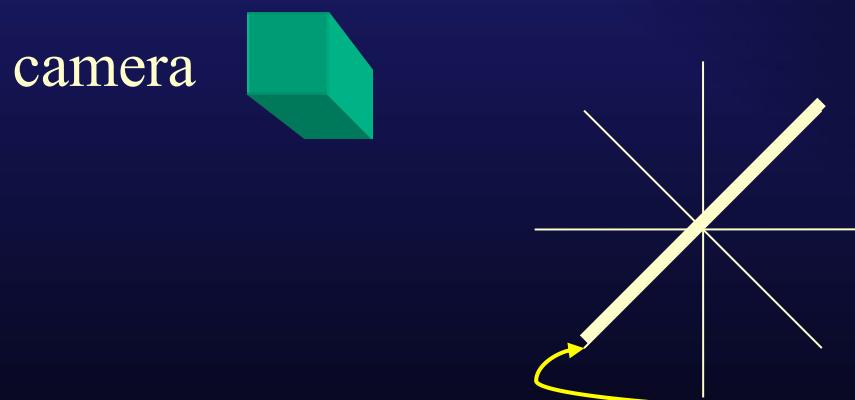
Texture Billboards (Real-Time Interaction)

- Use the negative of the camera view direction as the normal vector of a polygon to be textured.
- Polygon thus always faces the camera. (Don't fly over the top)
- Can use alpha-blending (i.e., don't draw non-object pixels) to remove texture/image "background".



Texture Billboards (Real-Time Interaction)

- By making different textures for different camera views, we can create view-dependent 3D appearance on a billboard polygon (e.g., game character sprites).
- Top view:

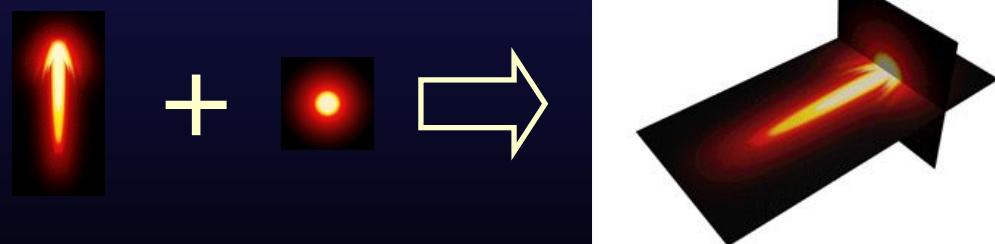
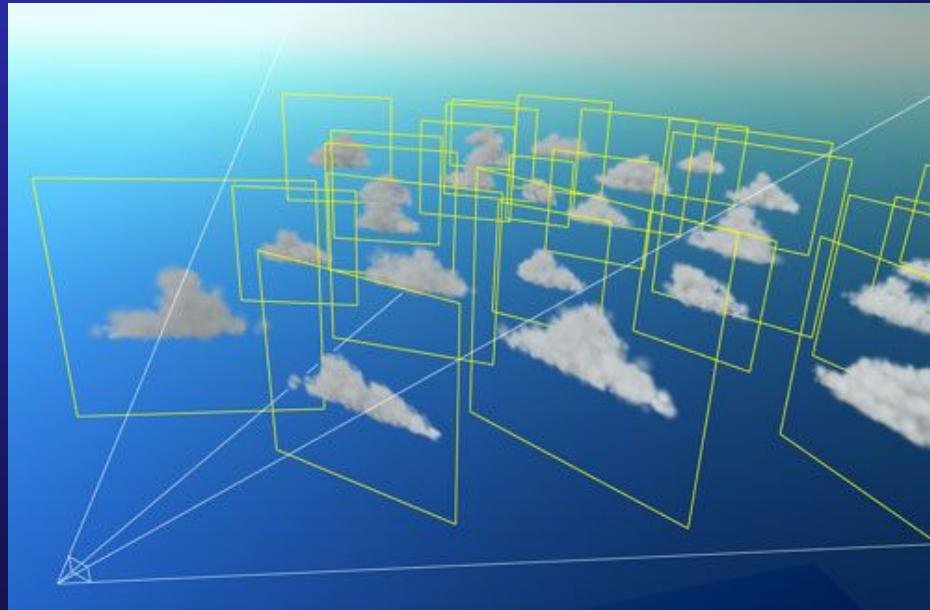


8 textures: one
on each face
of polygon

Use polygon billboard texture which lies on polygon with normal pointing closest to camera position, e.g. this one.

Common in Games and Virtual Environments

- E.g., Clouds, explosions, lens flare, laser trails, ...



<http://www.cs.unc.edu/~harrism/clouds/>

Using a Map to Vary the Reflectance Function:
e.g., n in \cos^n specular reflectance



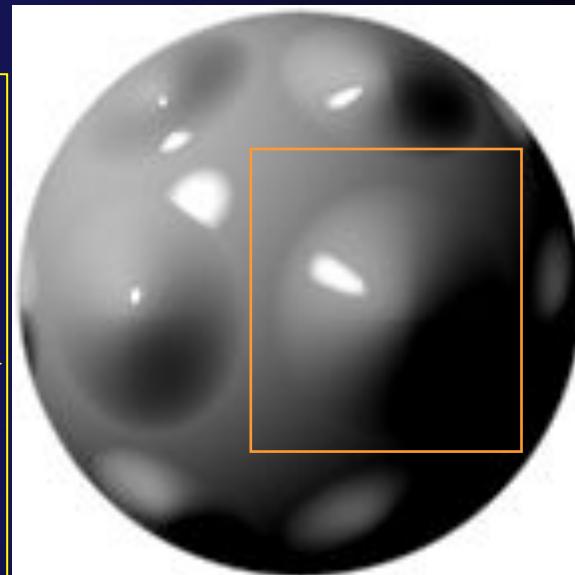
© 1984 ALAN GREEN—DIGITAL EFFECTS, INC.

Bump Mapping (2D Map to 3D Effect; Blinn)

0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	2	5	5	2	0	0	0
0	1	5	9	9	5	1	0	0
0	1	5	9	9	5	1	0	0
0	0	2	5	5	2	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0

2D Bump or
elevation map $\mathbf{b}(u,v)$

applied
like
texture
map to →
surface
 $\mathbf{p}(u,v)$,
but...



Bump map perturbs the local surface normal vector by the partial derivatives $\mathbf{b}_u(u,v)$ and $\mathbf{b}_v(u,v)$ of the map values, giving the illusion of curvature.

Bump Map Partial Derivatives $b_u(u,v)$ and $b_v(u,v)$ [e.g.:]

$\mathbf{b}(u,v)$ bump map:

0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	2	5	2	0	0	0	0
0	1	5	9	9	5	1	0	0
0	1	5	9	9	5	1	0	0
0	0	2	5	5	2	0	0	0
0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0

$\uparrow v$

$u \rightarrow$

$$\begin{aligned} \mathbf{b}_u(u_i, v_j) &\cong \mathbf{b}(u_{i+1}, v_j) - \mathbf{b}(u_i, v_j) \\ \mathbf{b}_v(u_i, v_j) &\cong \mathbf{b}(u_i, v_{j+1}) - \mathbf{b}(u_i, v_j) \end{aligned}$$

$\mathbf{b}_u(u,v)$

$\mathbf{b}_v(u,v)$

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	1	0	-1	0	0	0	0
0	0	0	-1	-1	0	0	0	0
0	2	3	0	-3	-2	0	0	0
0	0	-2	-4	-4	-2	0	0	0
1	4	4	0	-4	-4	-1	0	0
0	-1	-3	-4	-4	-3	-1	0	0
1	4	4	0	-4	-4	-1	0	0
0	0	0	0	0	0	0	0	0
0	2	3	0	-3	-2	0	0	0
0	1	3	4	4	3	1	0	0
0	0	1	0	-1	0	0	0	0
0	0	2	4	4	2	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0

Bump Function (1)

- Surface X parameterized by $\mathbf{p}(u,v)$ has a normal vector $\mathbf{n}(u,v)$ which is the cross-product of the partial derivatives:

$$\mathbf{n}(u, v) = \mathbf{p}_u(u, v) \times \mathbf{p}_v(u, v)$$

- Perturb surface \mathbf{p} along its normals by the bump map $\mathbf{b}(u,v)$ to get a new surface Y parameterized by $\mathbf{P}(u,v)$ as:

$$\mathbf{P}(u, v) = \mathbf{p}(u, v) + \mathbf{b}(u, v) \frac{\mathbf{n}(u, v)}{\|\mathbf{n}(u, v)\|}$$

- The new normal vectors for Y at $\mathbf{P}(u,v)$ are:

$$\mathbf{N}(u, v) = \mathbf{P}_u(u, v) \times \mathbf{P}_v(u, v)$$

- So we need to compute them.
- First, let's drop the (u,v) parts from the notation to make it simpler.

Bump Function (2)

$$\mathbf{P}_u = \mathbf{p}_u + \mathbf{b}_u \frac{\mathbf{n}}{|\mathbf{n}|} + \mathbf{b} \frac{\partial}{\partial u} \left(\frac{\mathbf{n}}{|\mathbf{n}|} \right) \quad \text{and} \quad \mathbf{P}_v = \mathbf{p}_v + \mathbf{b}_v \frac{\mathbf{n}}{|\mathbf{n}|} + \mathbf{b} \frac{\partial}{\partial v} \left(\frac{\mathbf{n}}{|\mathbf{n}|} \right)$$

- Assuming the effects of the bump map are relatively small, we can ignore the final term in each to get the approximation:

$$\mathbf{N}' \cong \mathbf{p}_u \times \mathbf{p}_v + \mathbf{b}_u \frac{\mathbf{n} \times \mathbf{p}_v}{|\mathbf{n}|} + \mathbf{b}_v \frac{\mathbf{n} \times \mathbf{p}_u}{|\mathbf{n}|} = \mathbf{n} + \mathbf{b}_u \frac{\mathbf{n} \times \mathbf{p}_v}{|\mathbf{n}|} + \mathbf{b}_v \frac{\mathbf{n} \times \mathbf{p}_u}{|\mathbf{n}|}$$

- So to approximate the partial derivatives use, for a suitable small ε :

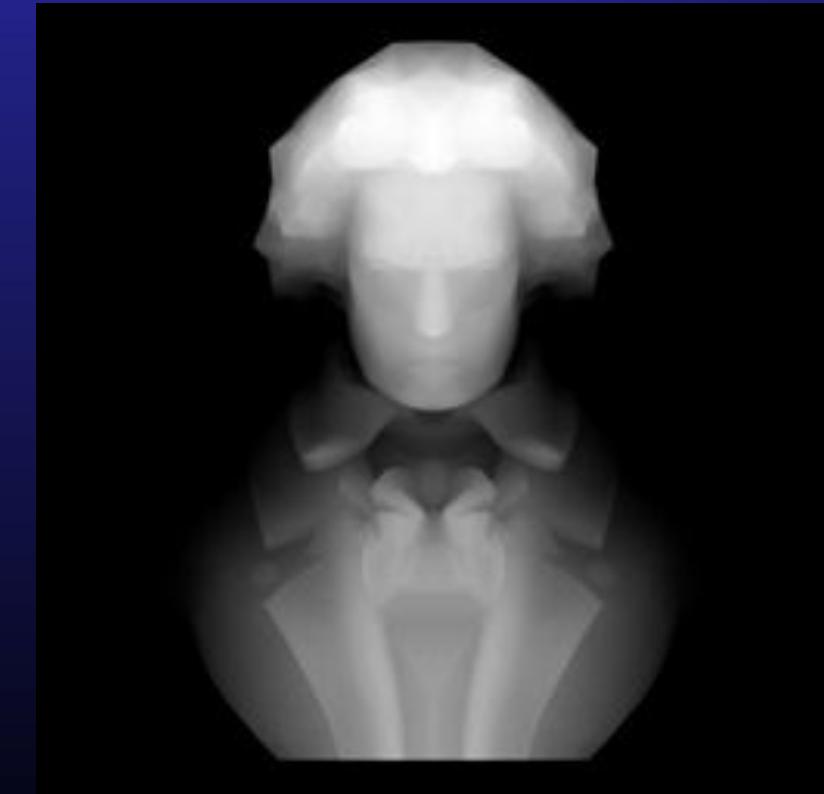
$$\mathbf{b}_u(u, v) = \frac{1}{2\varepsilon} \mathbf{b}(u + \varepsilon, v) - \mathbf{b}(u - \varepsilon, v)$$

$$\mathbf{b}_v(u, v) = \frac{1}{2\varepsilon} \mathbf{b}(u, v + \varepsilon) - \mathbf{b}(u, v - \varepsilon)$$

- Interpolate on the discrete values of $\mathbf{b}(i,j)$ to find the necessary $\mathbf{b}(u,v)$, or compute directly if given as a function.

Bump Map uses Elevation (Height) Only

Heights



Bump Mapped to single quad edge-on



Bump Mapping Adds Visual Complexity Cheaply



© 1982 DIGITAL EFFECTS

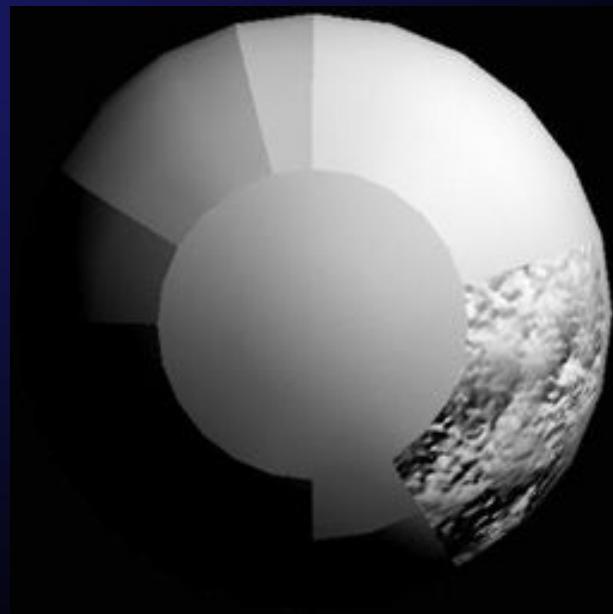
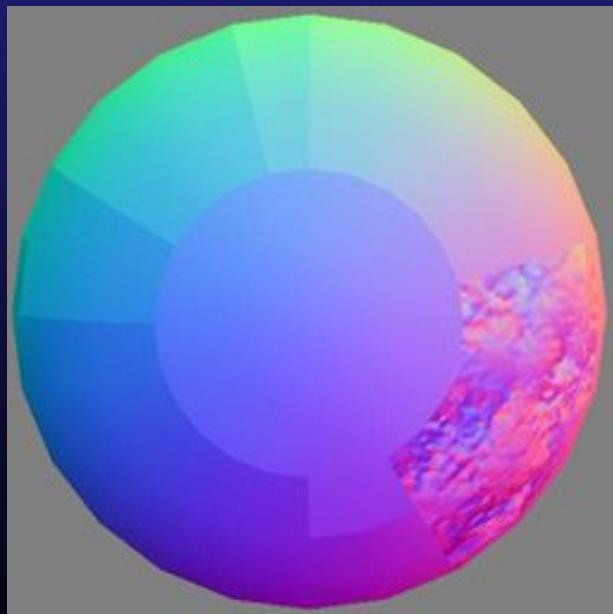
Certain Textures Model Well with Bump Maps



NATIONAL RESEARCH COUNCIL OF CANADA

Bump Map Generalizes to a Normal Map

- “Invent” surface “normals” as desired.
- Specify (x,y,z) normals everywhere on a parameterized surface by any convenient method.
- Encode x in red, y in green, z (depth) in blue.
- Still doesn’t change geometry, though.



Normal Map



RGB encoded normal map

Dave McCoy

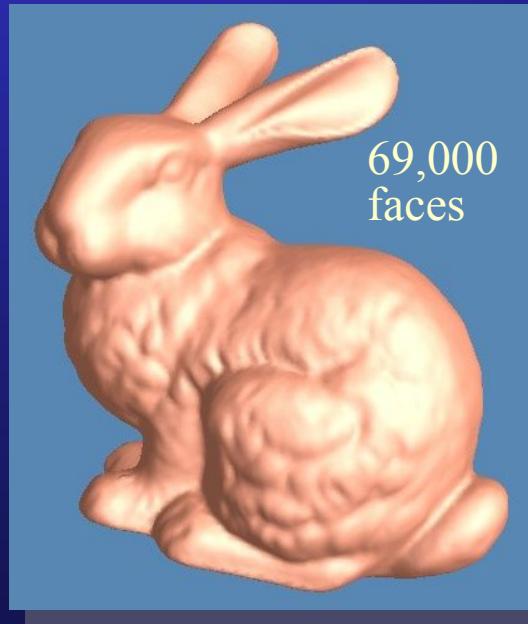


Rendered front view
with lighting

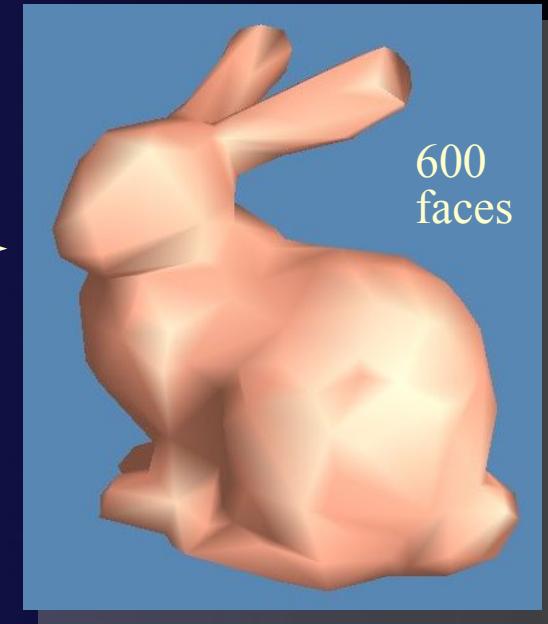
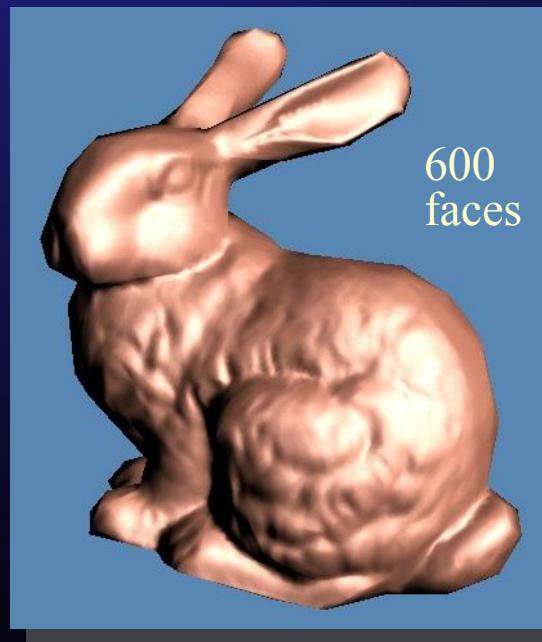
Edge view

895

Normal Maps Reduce Polygon Mesh Size



Geometric
simplification



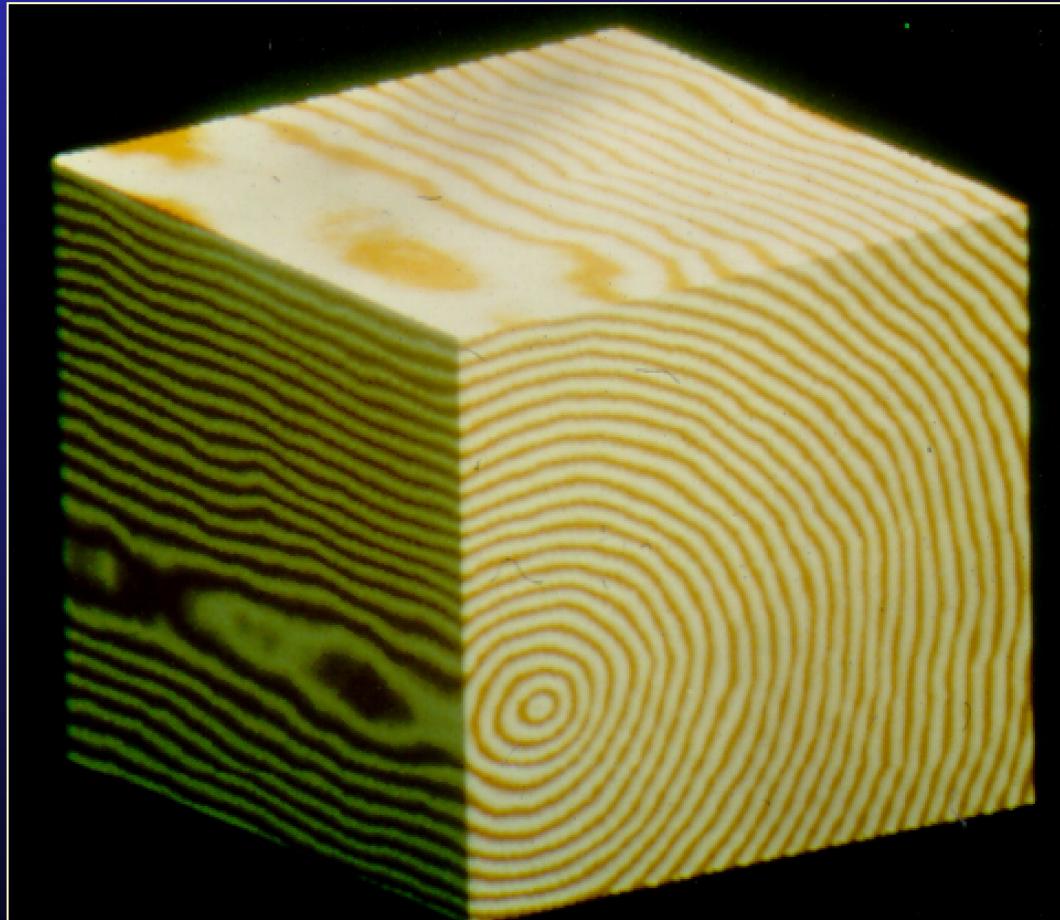
Simplified mesh +
normal map:
Conveys visual
detail of original
geometry

But note edge artifacts (!)

3D Texture Mapping

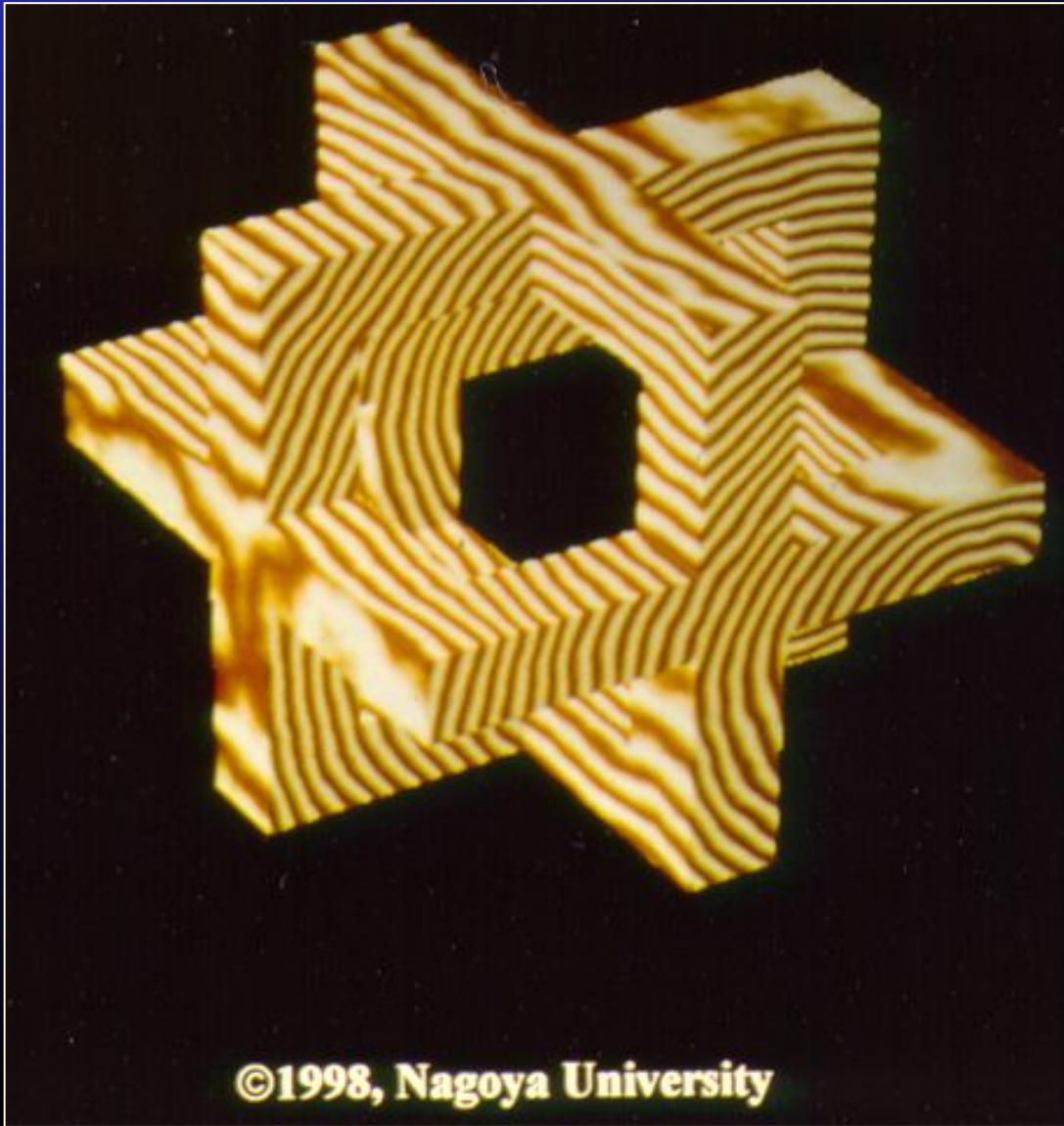
- Texture is a 3D array of voxels, oct-trees, or a procedure $T(u,v,w)$.
- Define the texture embedding (mapping) from (u,v,w) into (x,y,z) .
- Whenever a surface color (or other property) at a point (x,y,z) is needed, look it up (or compute it) from $T(u,v,w)$.
- Texture is now defined everywhere **inside** the object as well as on the surface; if the surface is cut away, the inside texture is consistent with the surface (like stone, marble, wood, etc.)

Wood Grain



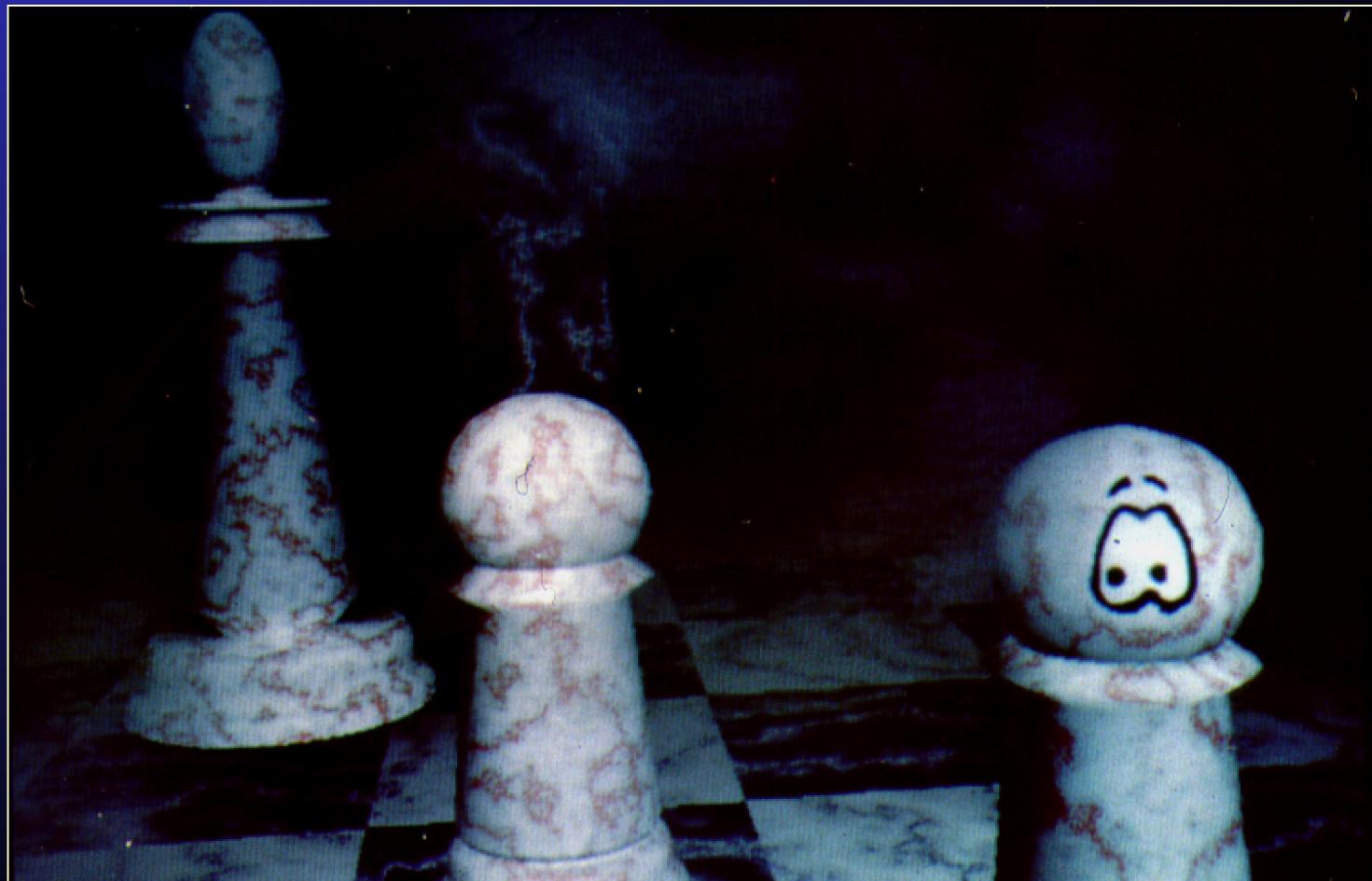
©1998, Nagoya University

Wood with Cut-Outs



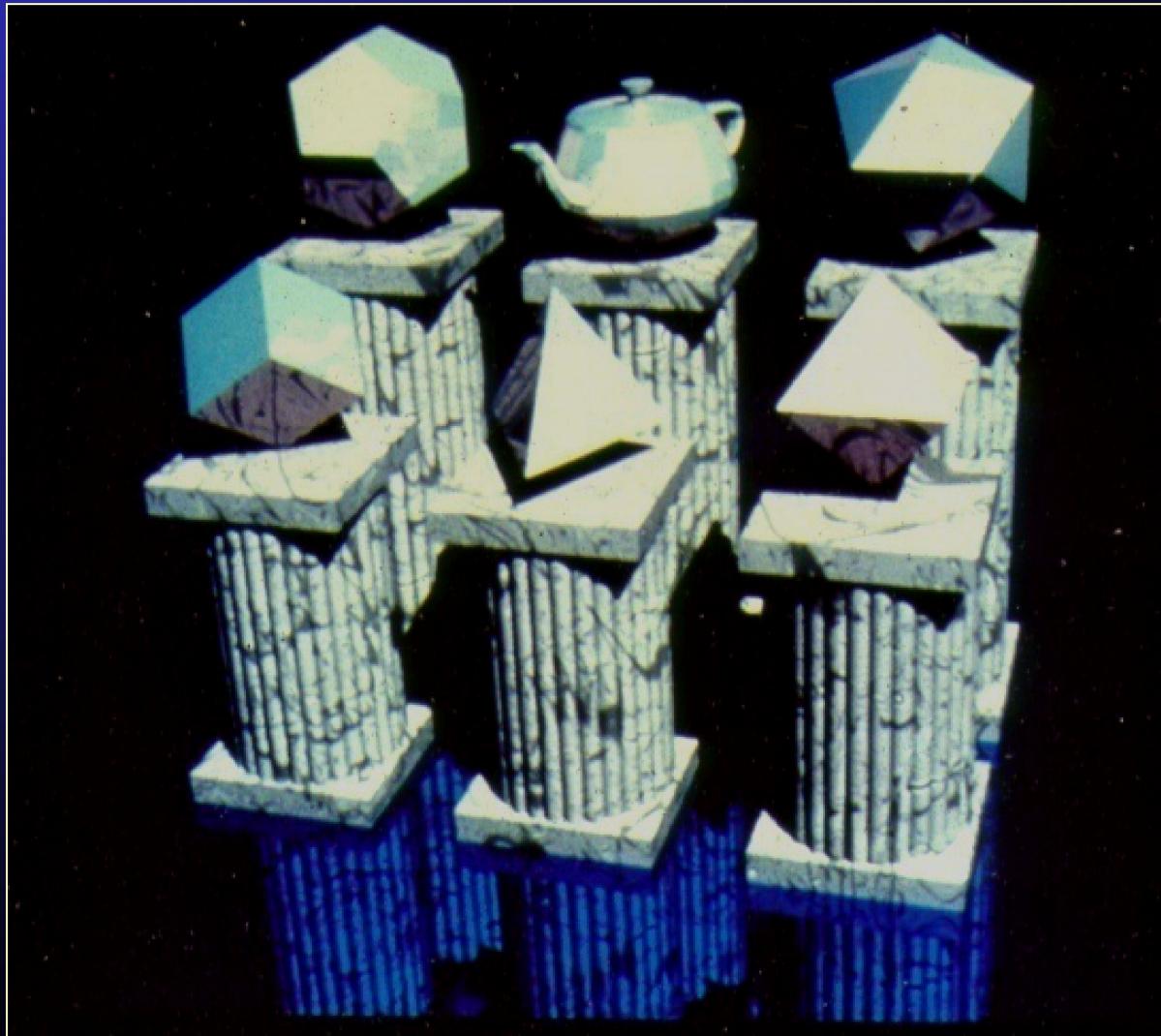
©1998, Nagoya University

3D Solid Texture -- Allows Complex Geometries to be Textured



© 1989 DAVID S. EBERT

The Platonic Solids According to Computer Graphics



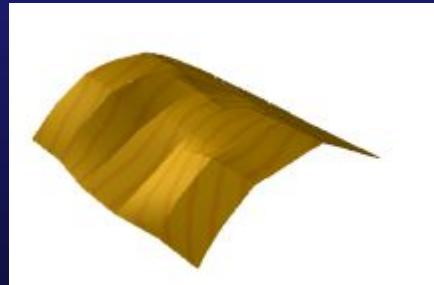
© 1987 ARVO/KIRK @ APOLLO COMPUTER

3D Solid Texture Map in Room Scene



Displacement Mapping

- Start with parameterized object surface $\mathbf{S}(u,v)$.
- Displacement map: a 2D height field or function $\mathbf{D}(u,v)$.
- Apply corresponding height (displacement) $\mathbf{S}(u,v) + \mathbf{D}(u,v)$.

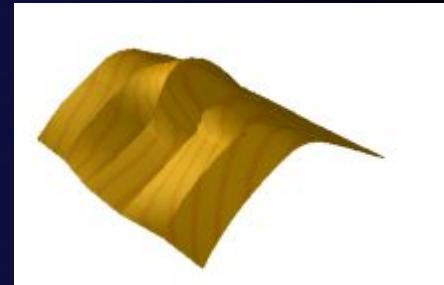


Original surface
 $\mathbf{S}(u,v)$

Displacement map
 $\mathbf{D}(u,v)$

A 8x8 grid representing a displacement map. The values are as follows:

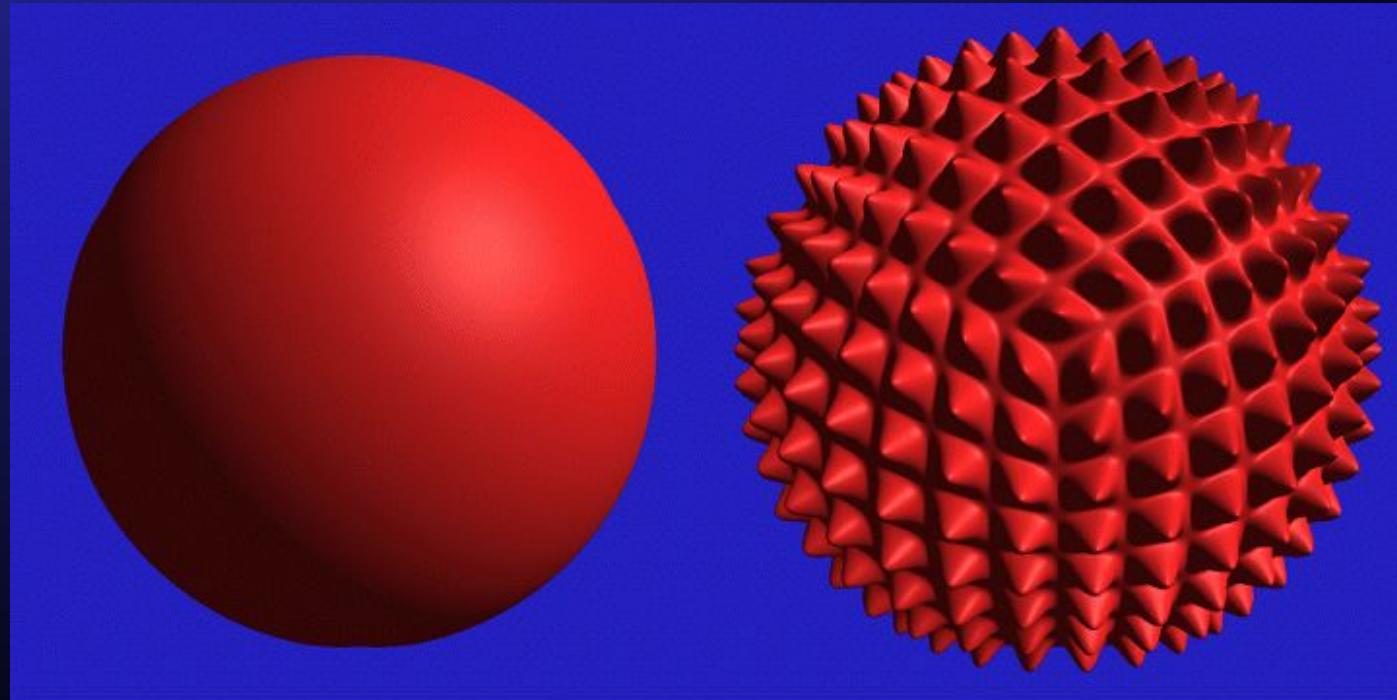
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	2	5	5	2	0	0
0	1	5	9	9	5	1	0
0	1	5	9	9	5	1	0
0	0	2	5	5	2	0	0
0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0



New surface with
actual displacements
 $\mathbf{S}(u,v) + \mathbf{D}(u,v)$

Displacement Mapping

- The geometry must be displaced before visibility is determined.
- Can render via a ray tracer: transform viewing ray into object space; displace object; do intersections.



Scene with Displacement Maps



Reflection Map



Scene with Displacement and Reflection Maps



Light Maps

- An efficient technique for static objects and lighting.
- Pre-calculate light intensity and color across polygon surface and make into texture map (bake).
- Linear filter (e.g., Gouraud) for pixel shade at run time.
- Add other dynamic lighting components at run time.



Policarpo
and Watt

Baking Textures

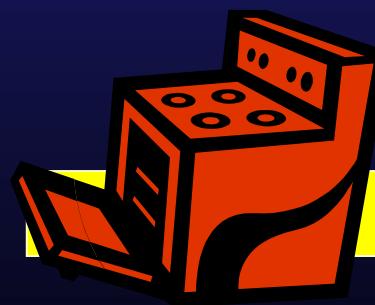
- To save time during rendering, many shading effects -- if static -- can be embedded or *baked* into a single texture map.
- This is frequently done for fixed lights, shadows, bump maps, etc.

Light/shadow 1 map

Light/shadow 2 map

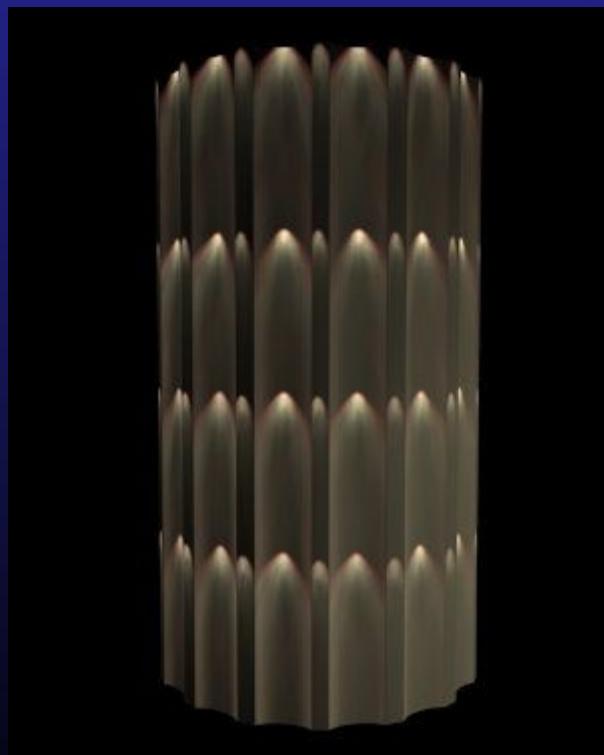
⋮

Light/shadow 3 map

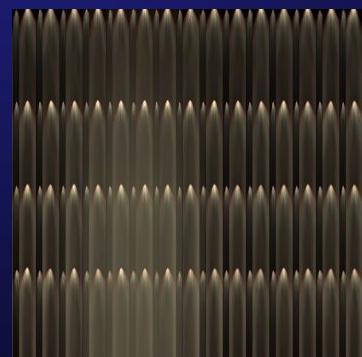


Example

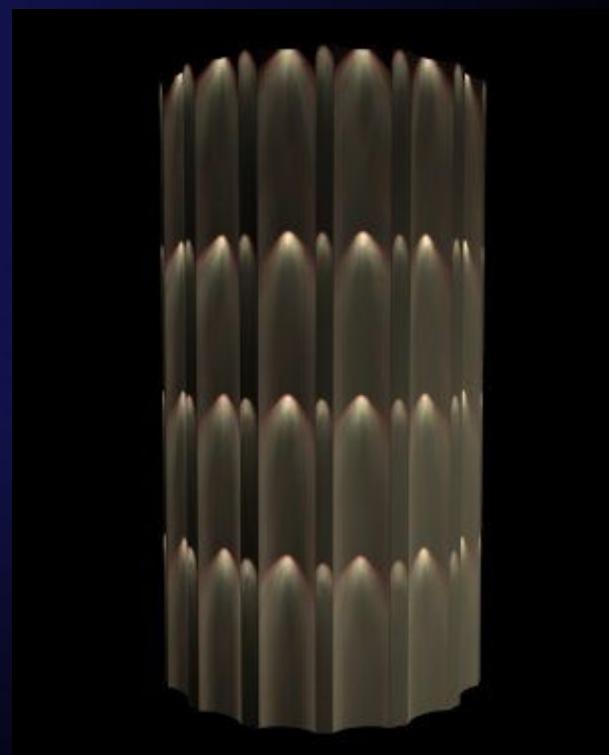
121 Lights:



Bake Texture:



No lights!:

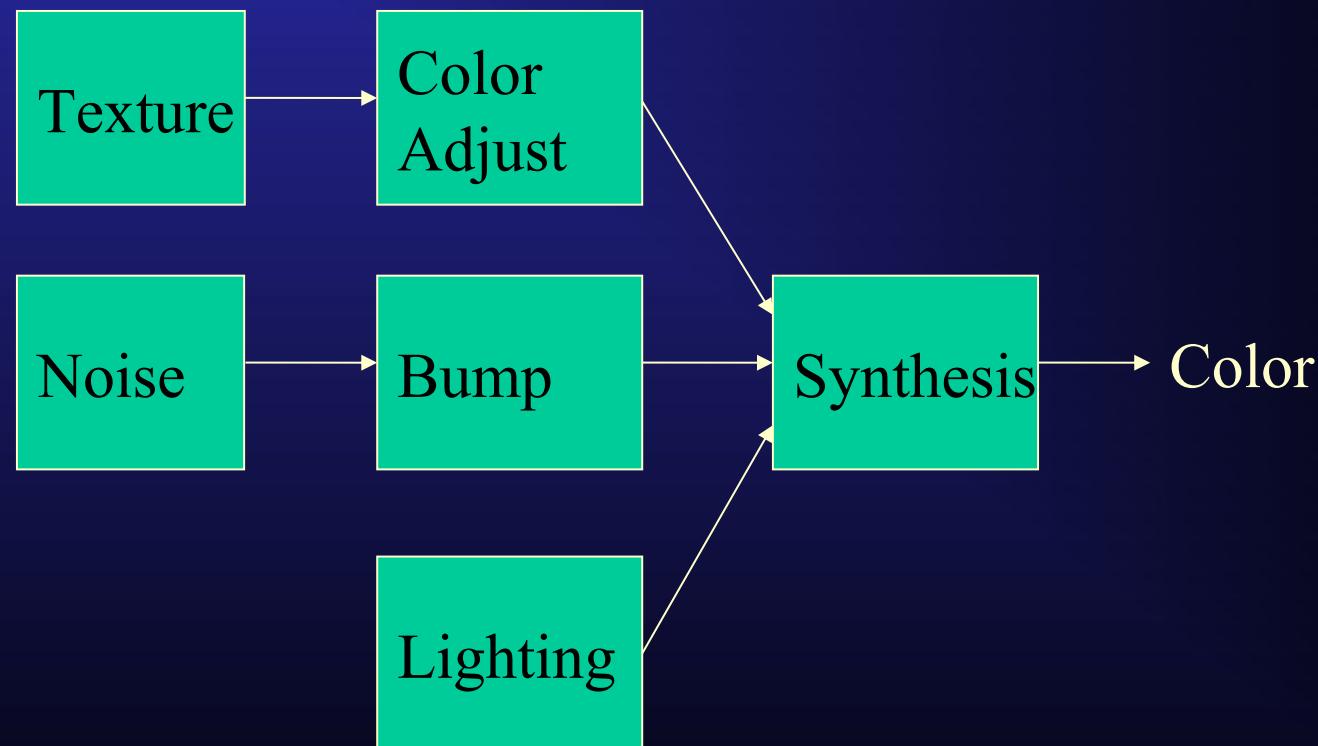


Shaders

- Store with every object or part of an object an indication of the type of shading algorithm to apply to it -- a **shader**, e.g.:
 - Diffuse, Specular, Metallic
 - Smooth, Rough
 - Texture, 2D or 3D
 - Bumps
 - Displacement map
 - Transmittance
- Done for **efficiency**, **flexibility**, and **customization** (Renderman, Maya, ...)
- Basis for GPU (board level) implementations

Simple Shader

- Organized as tree or network:

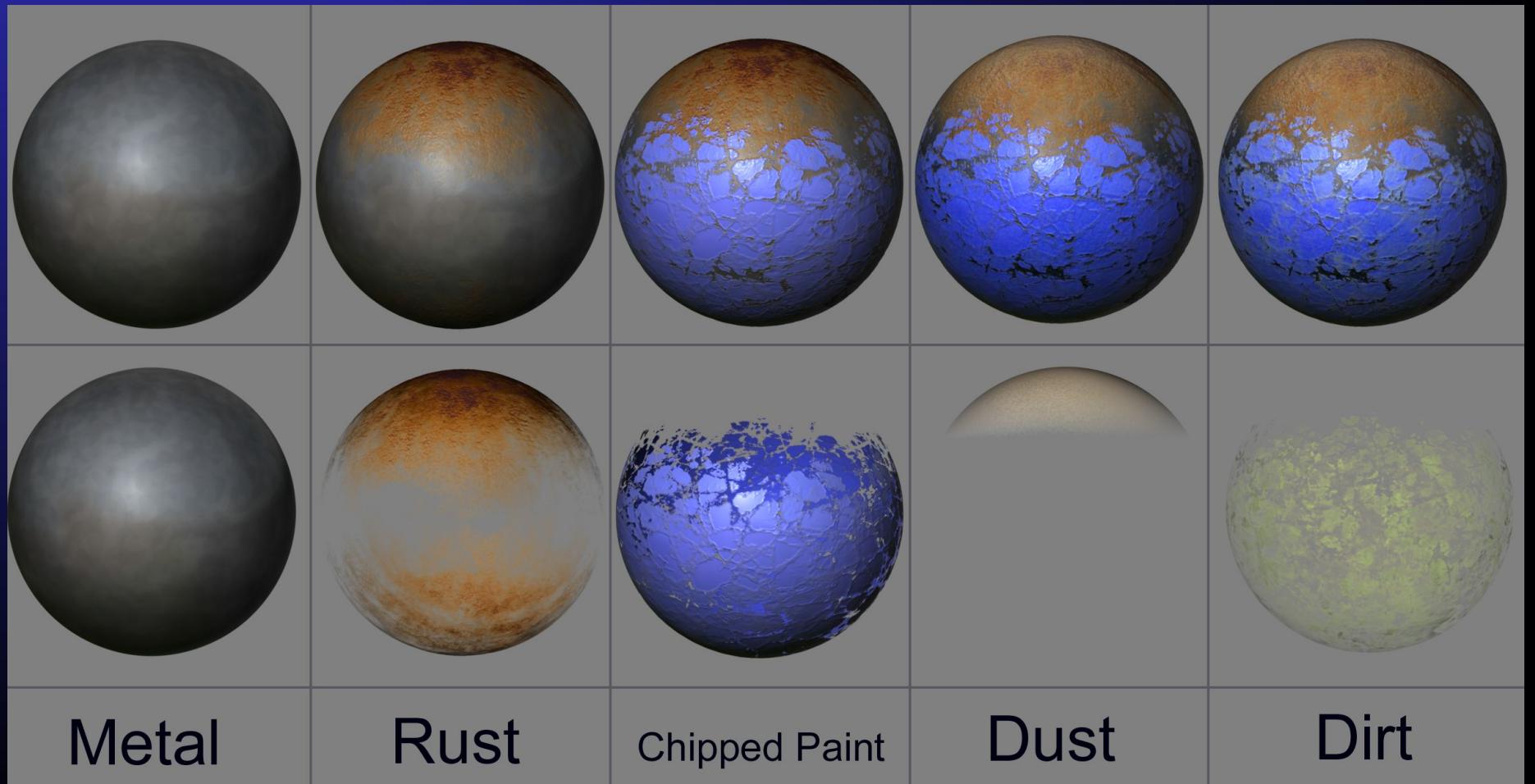


Aged Materials (Kanyuk and Teich)

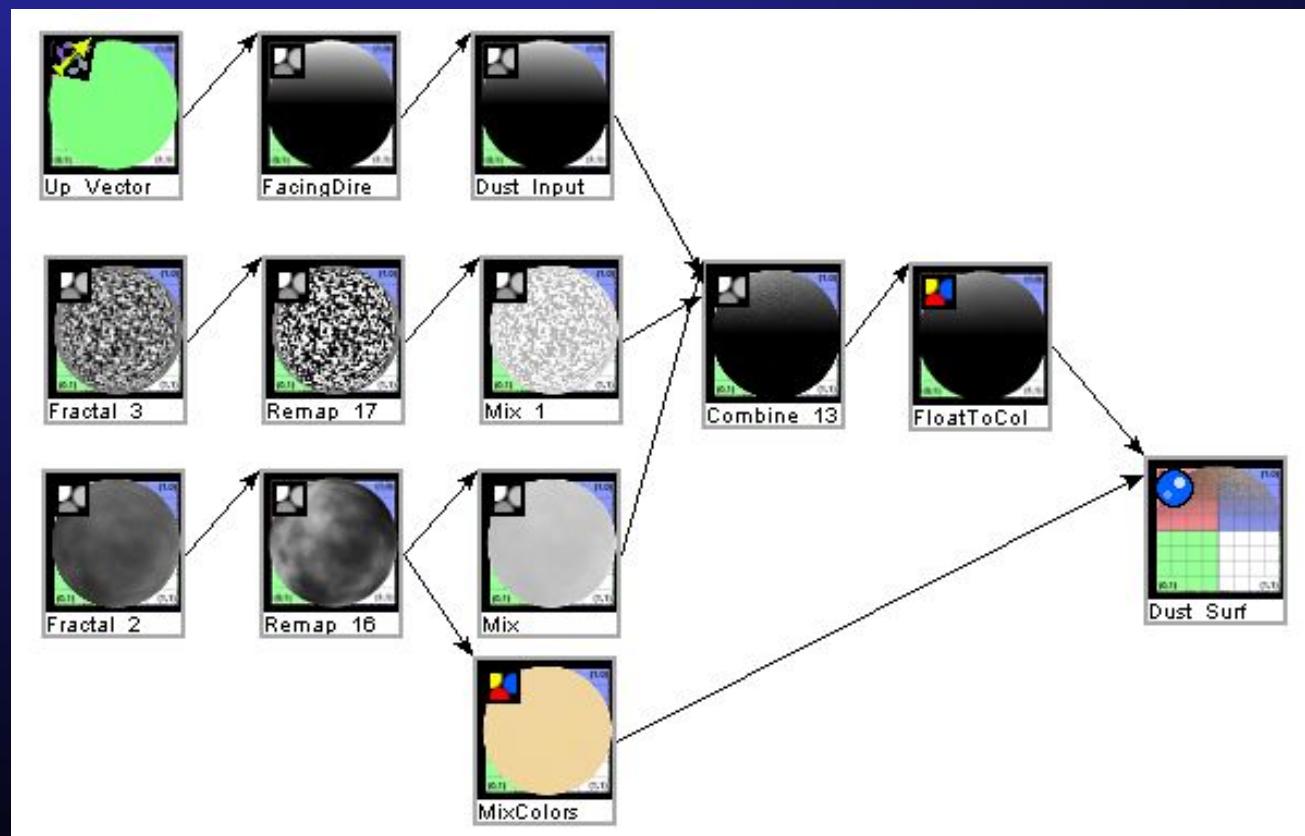
- Consider aged materials which may be representative of dirty environments:
 - Dust
 - Rust
 - Chipped Paint
 - Dirt
 - Concrete
 - Brick
 - Metal



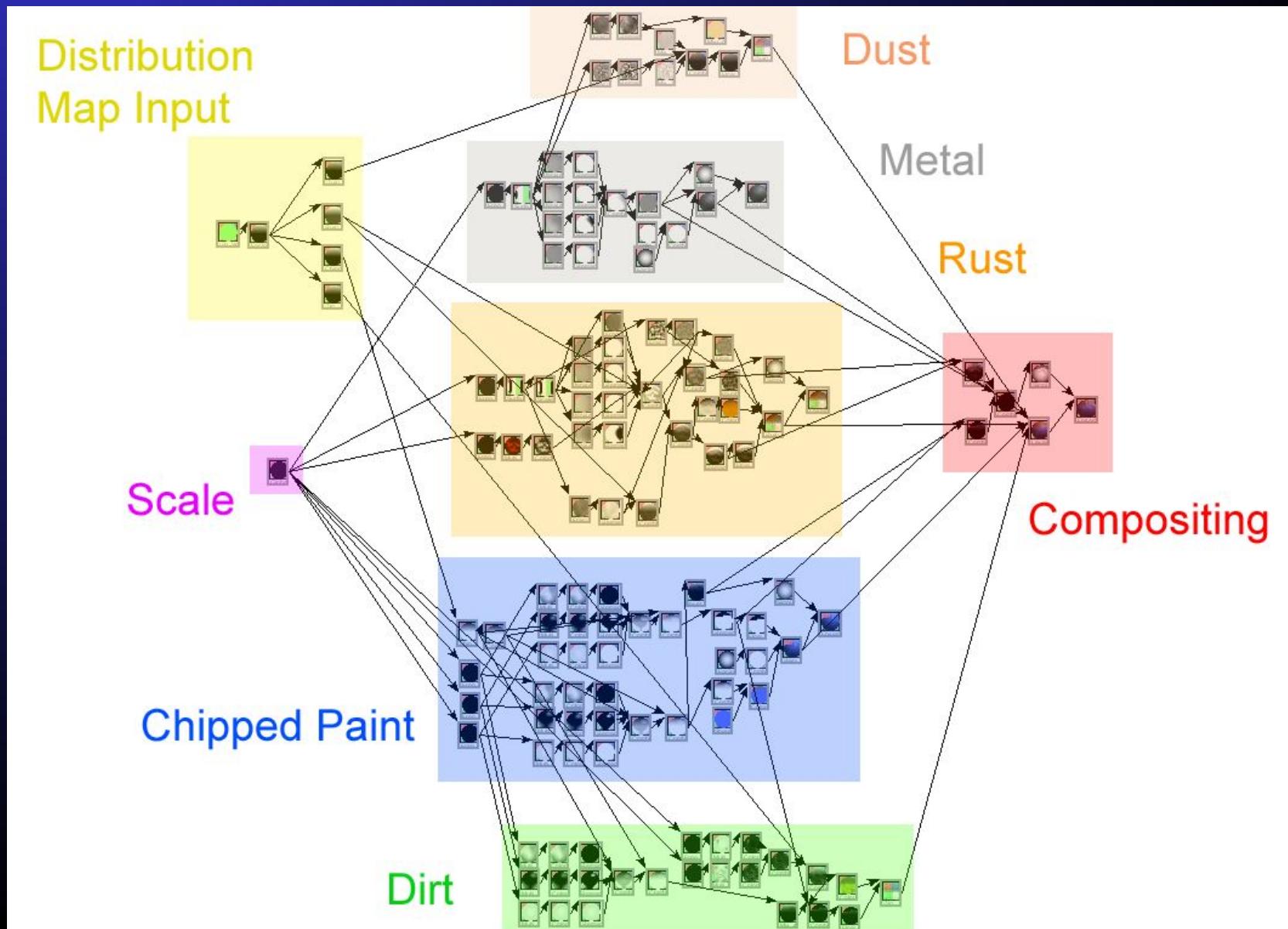
Aged Materials: Shaders



Dust Example

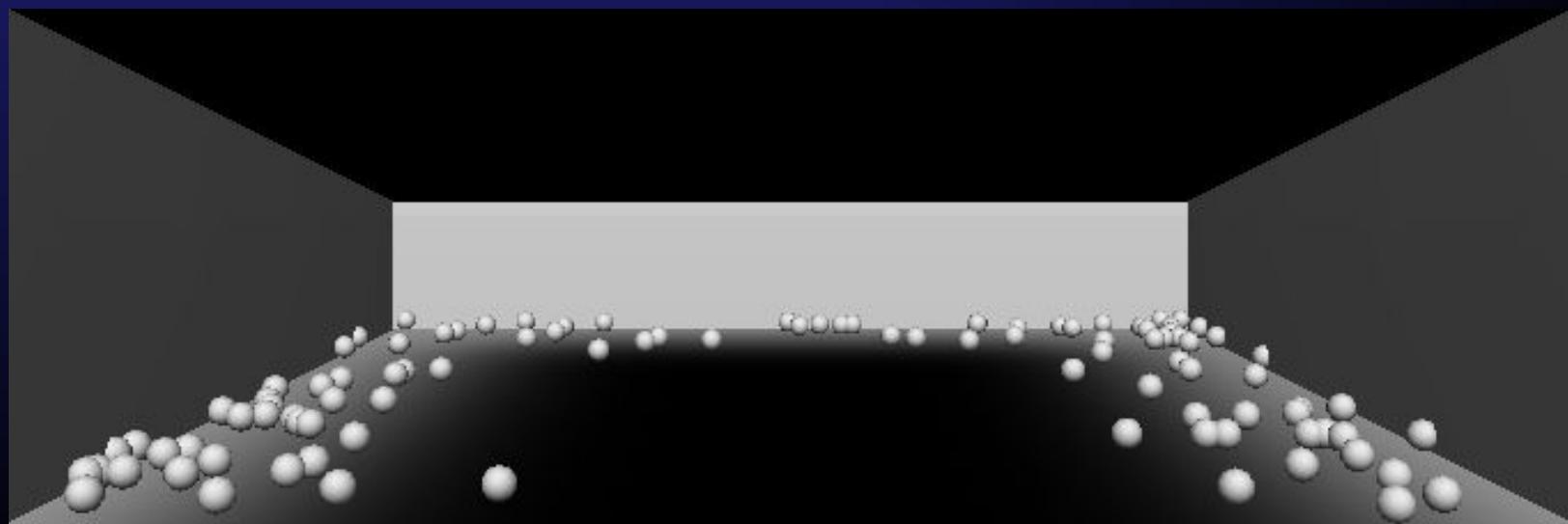


Full Shading Network



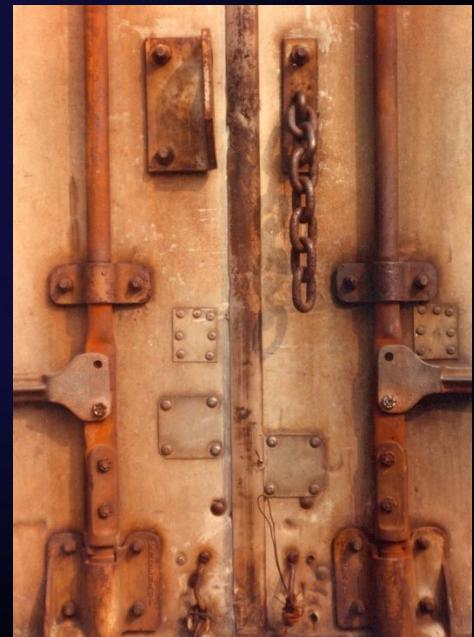
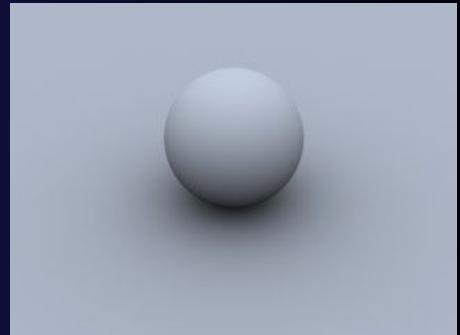
Distribution Maps (Roberts, Forziati, Kanyuk, Teich)

In a grayscale image, let white pixels represent a maximum relative probability that an object will be randomly placed in on an area mapped to the given pixel. Black pixels represent an area of zero probability and no object will ever be distributed on surface coordinates corresponding to these pixels. Gray pixels represent intermediate probabilities.



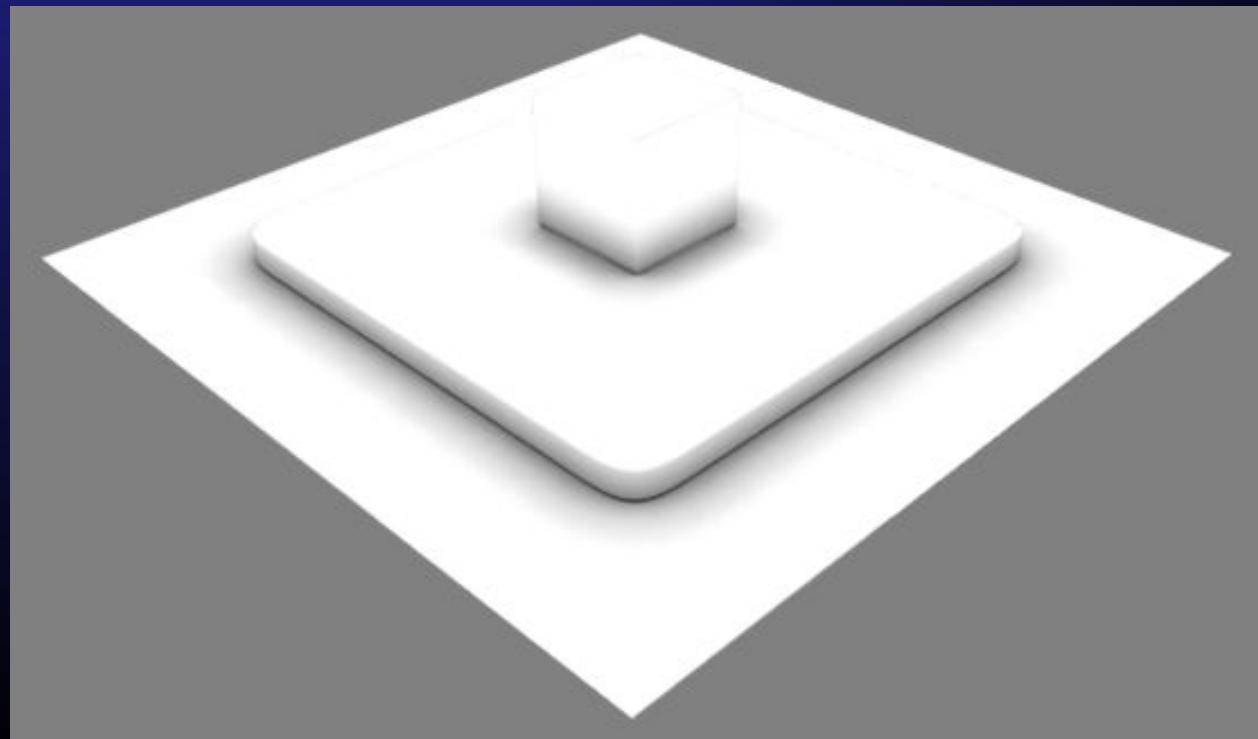
Ambient Occlusion as a Distribution Map

- There is a general tendency of materials to age more or less rapidly in “occluded” areas.
 - Moisture tends to collect in covered areas.
- **Ambient Occlusion** is a measure of geometric proximity (shadowing from a directionless light), and can characterize such regions.
 - Occlusion is calculated by casting a hemisphere of rays for each point being shaded and darkening it by the percentage of rays which intersect another surface within a fixed distance.

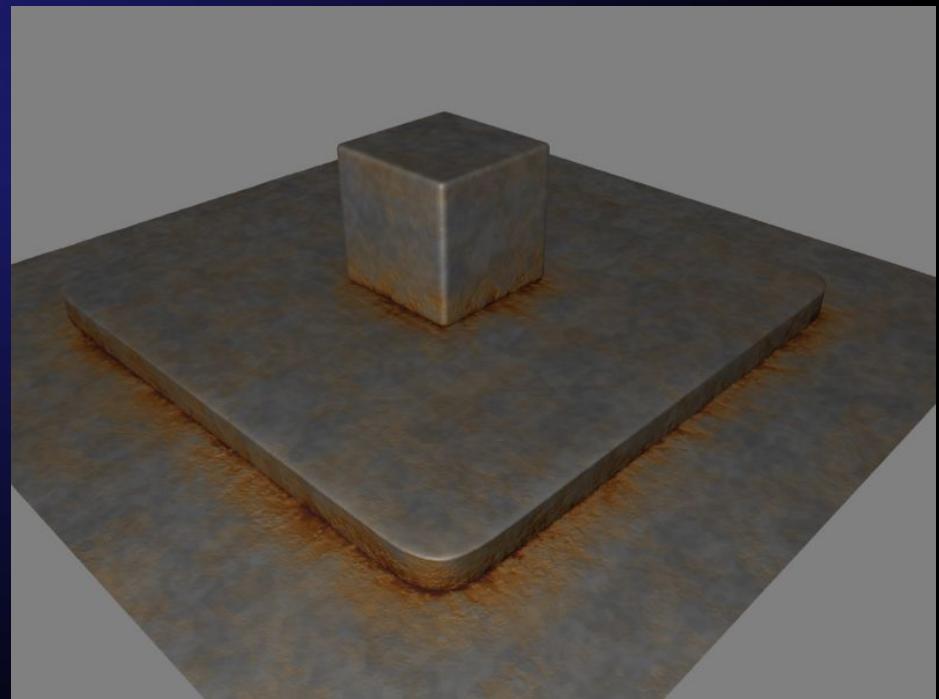
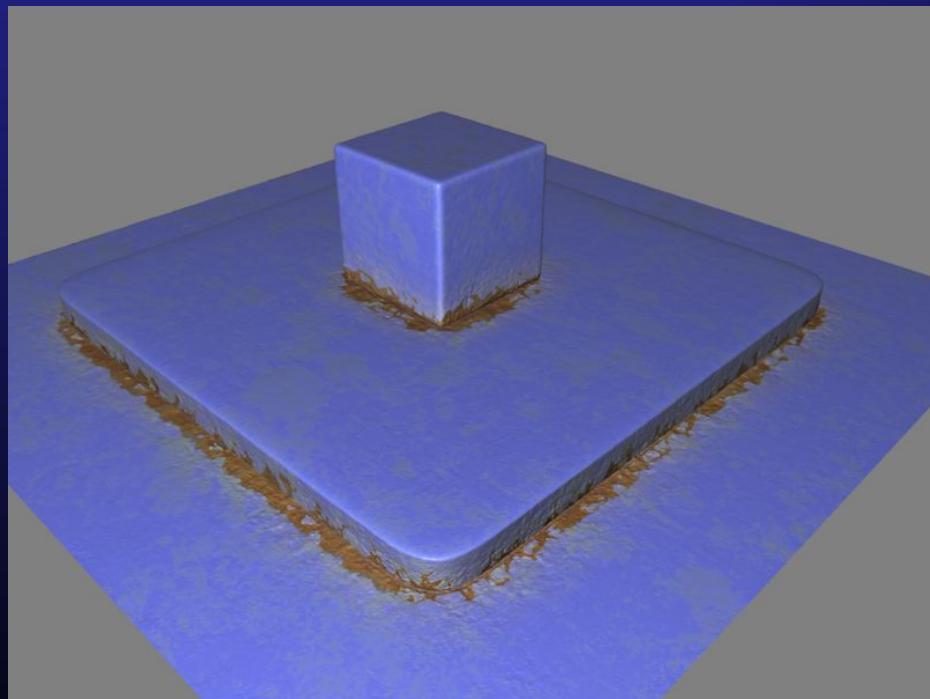


Ambient Occlusion Distribution Map

- An ambient occlusion map is an approximation to the actual distribution of “bounced” light (global illumination) in the scene (i.e., less in corners).
- We’ll see how to do this accurately when we talk about radiosity, path tracing and photon mapping.

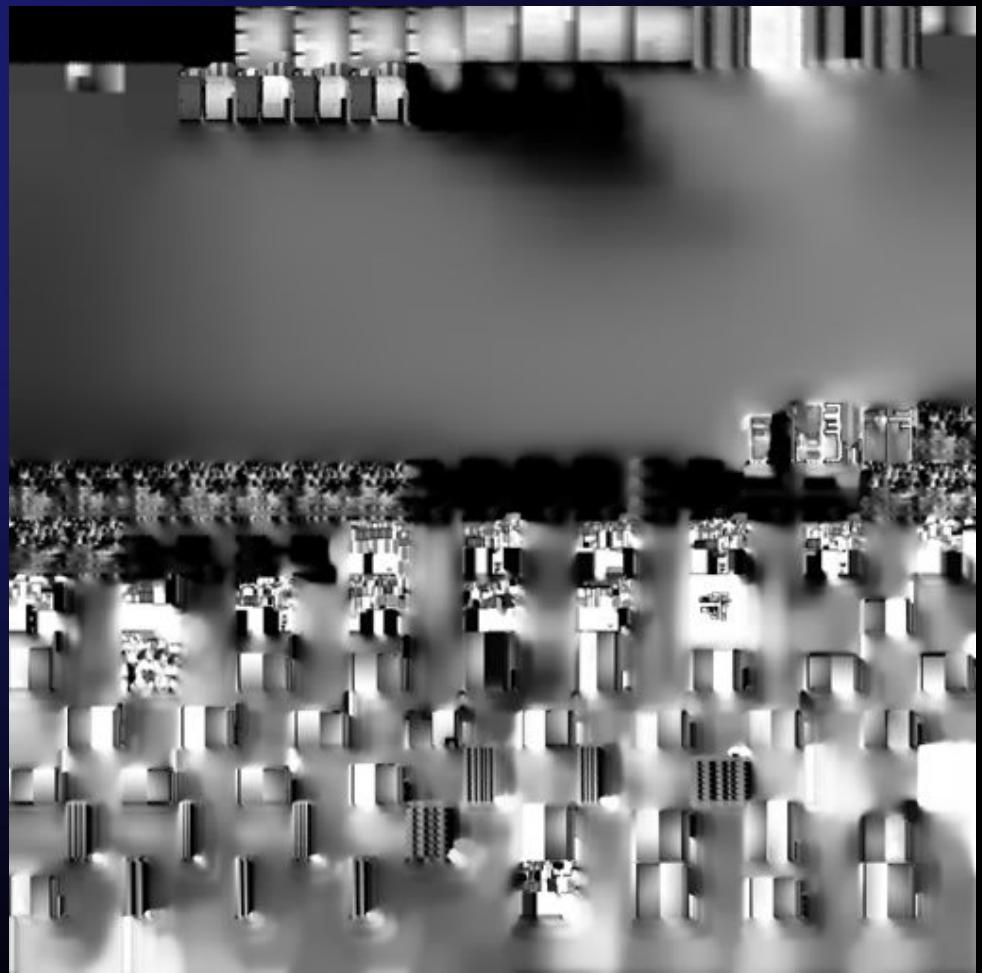


Ambient Occlusion Distribution Map with Applied Shader



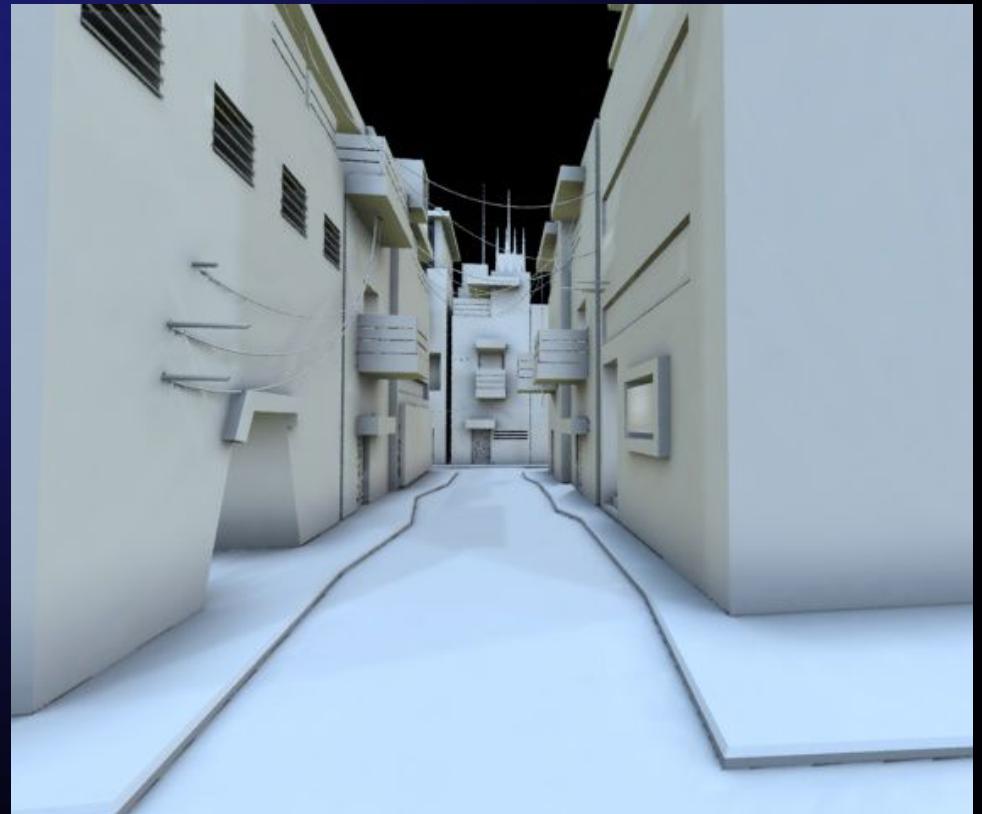
Baking Occlusion into a Texture Map

- Ambient Occlusion is very expensive to compute due to the large number of rays cast.
- Percentage of occlusion between stationary objects is a constant. Such occlusion can be computed once (baked), and reused for all subsequent renders.
- Here's a shader which bakes all of a scene's occlusion into a single 2D texture (in uv space).



Application: An Alley

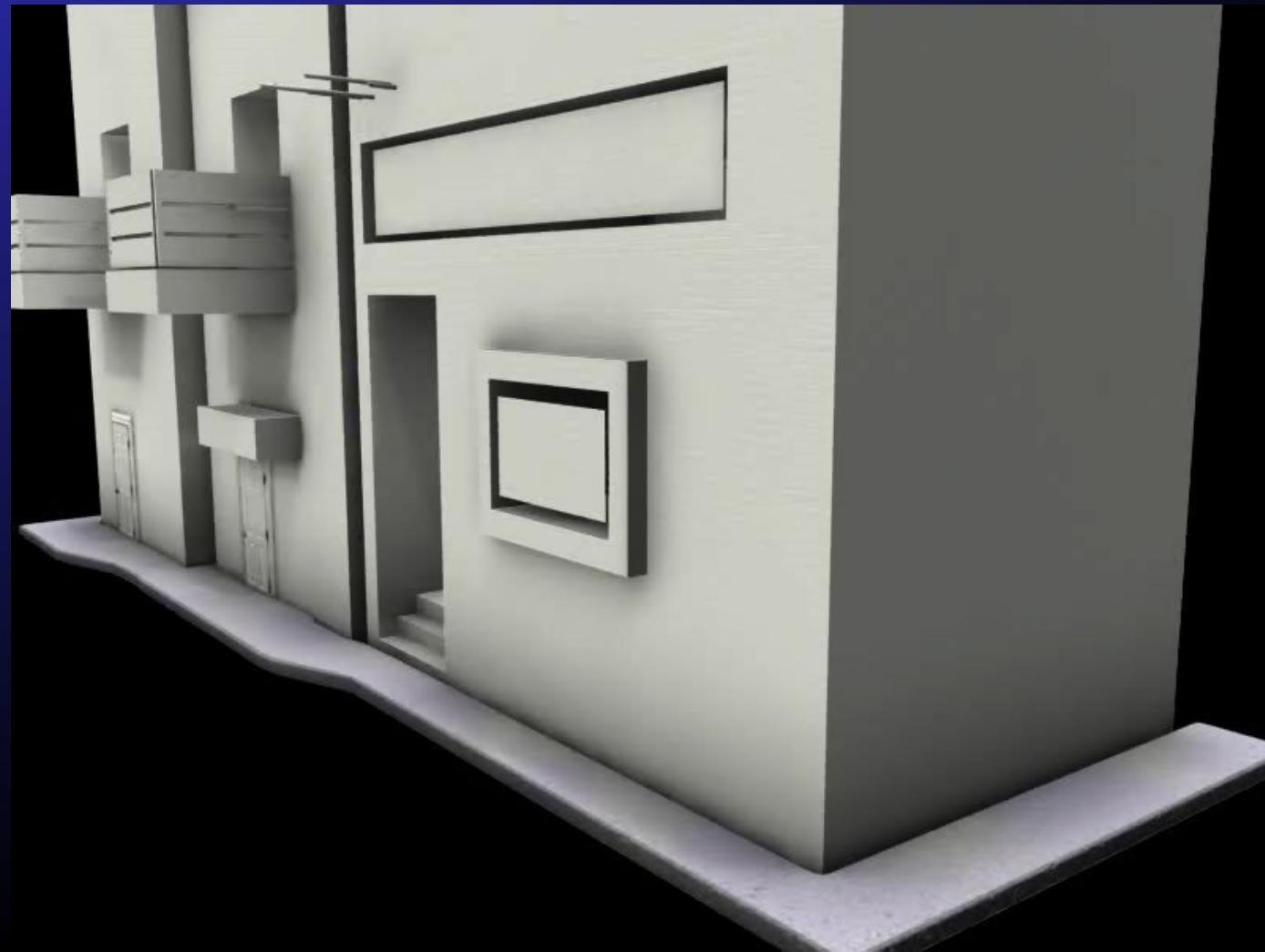
- Urban alley ways are renown for their accumulation of trash graffiti. Such a scene is an excellent test for an aging system.
- We purchased an unshaded street scene from Turbosquid to serve as the test case.



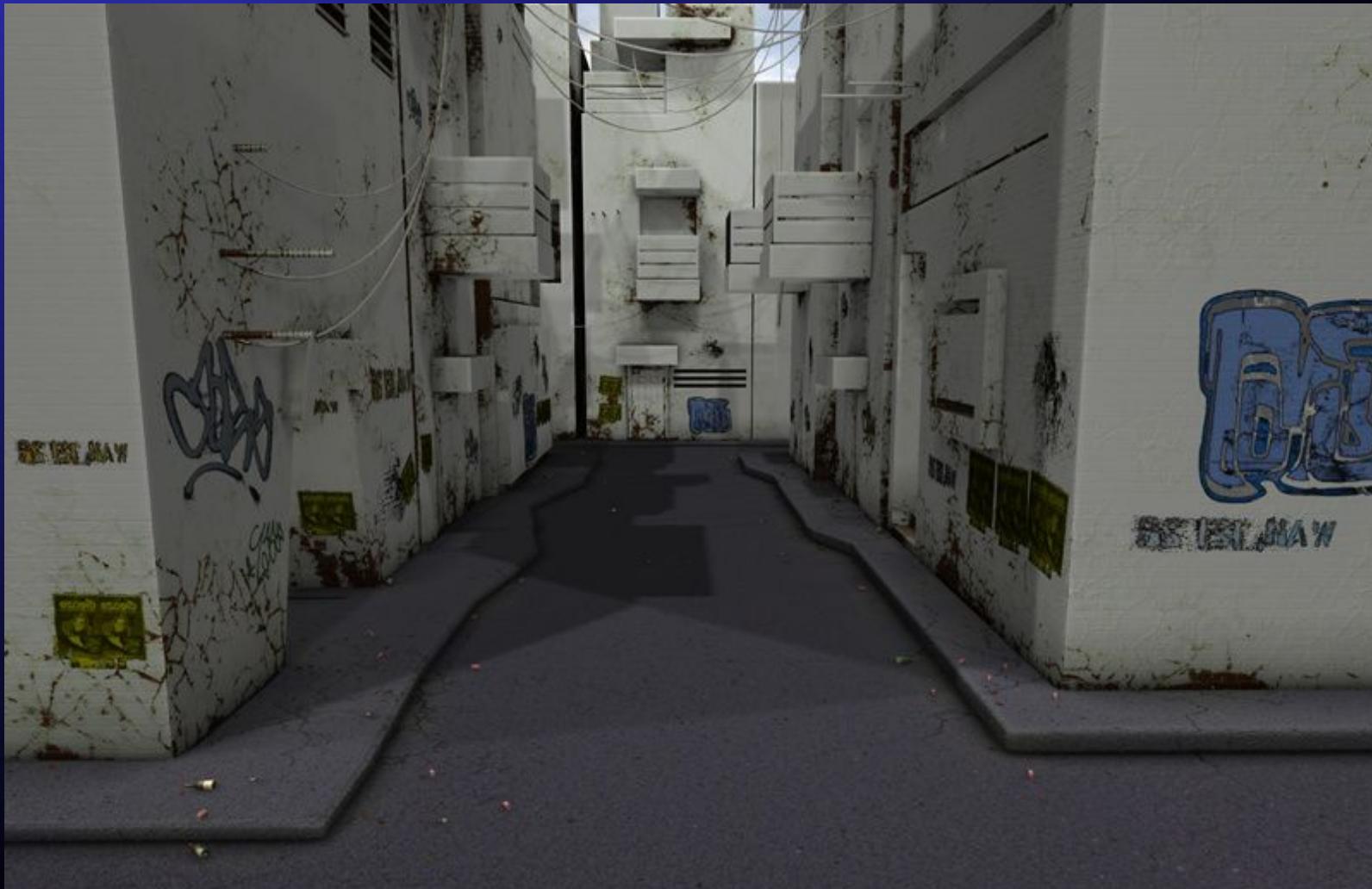
Application: An Alley Distribution Map for Surface Features



Alley Occlusion Map



Result!











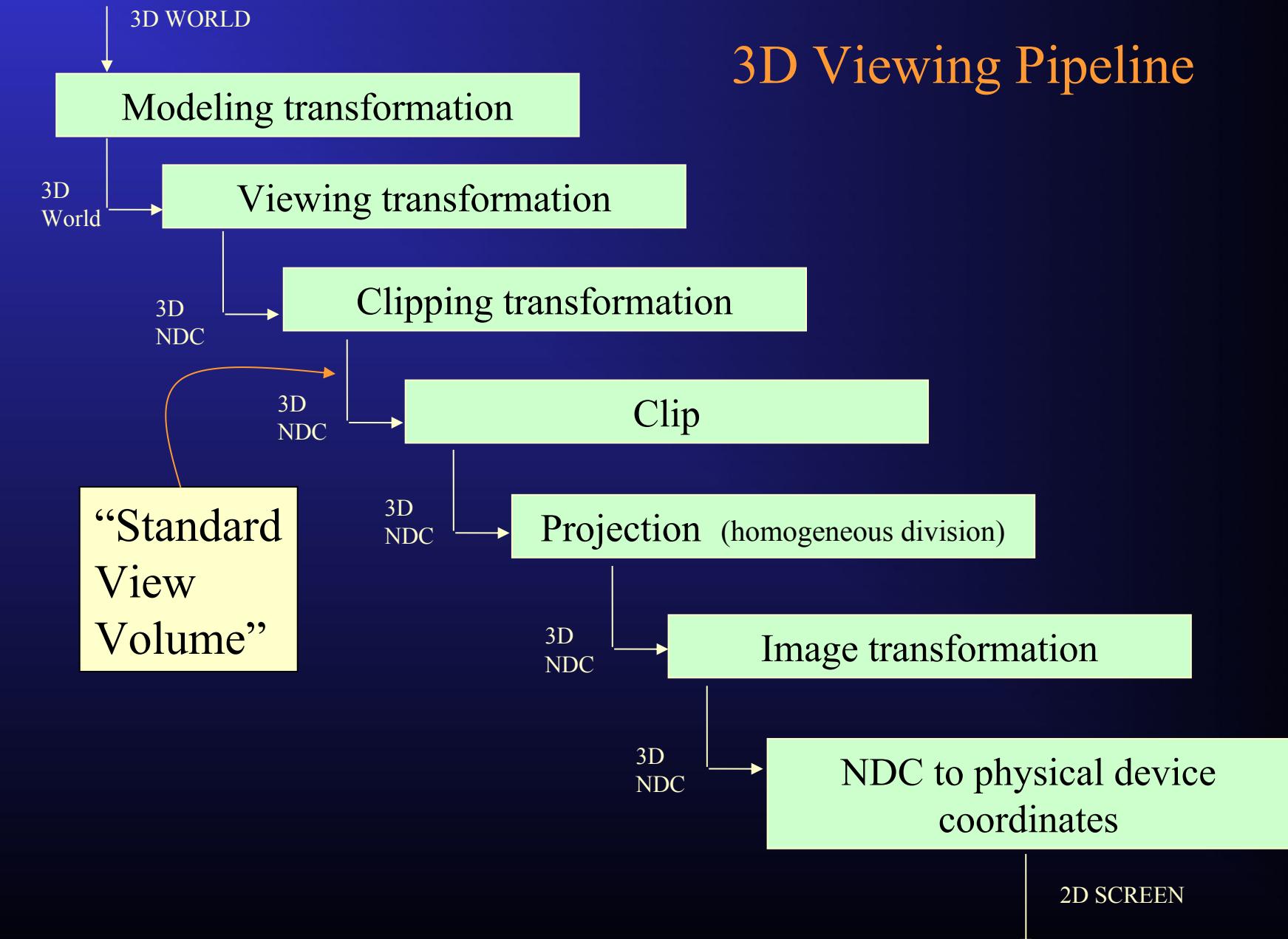
Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading ✓
- Mapping ✓
- 3D Viewing Transformations
- Anti-aliasing & Compositing
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

3D Viewing Transformations

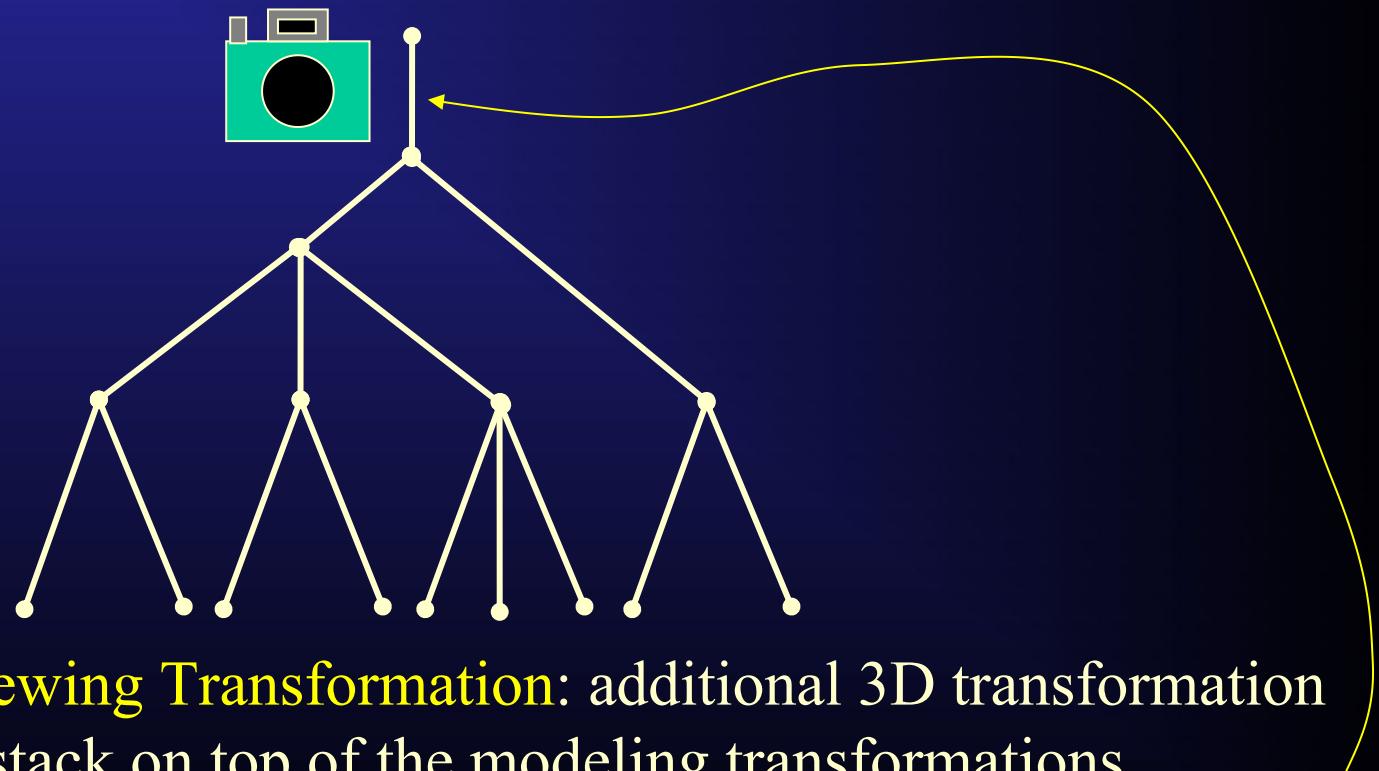
- 3D viewing pipeline
- Perspective and parallel projections
- Mathematics of the 3D viewing transformations
- Perspective transformation
- Polygon clipping
- Depth comparisons
- Perception of 3D on 2D displays

3D Viewing Pipeline



Scene Graph and Viewing Transformations

- Use 3D transformations to construct **scene graph** of objects desired.

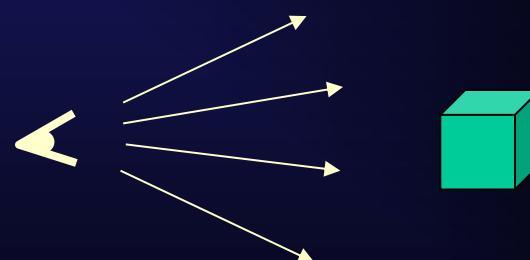
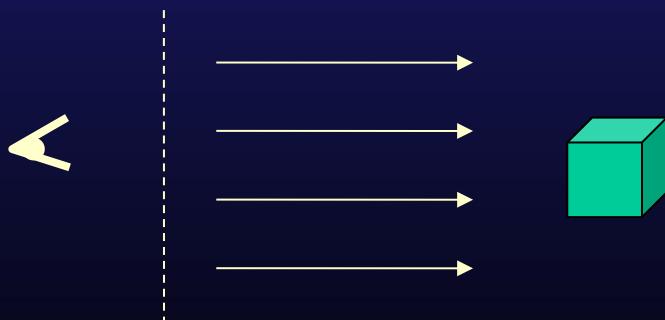


Camera or Viewing Transformation: additional 3D transformation matrices that stack on top of the modeling transformations.

They transform the (arbitrary) object world into a known camera coordinate system (e.g, see OpenGL camera).

3D Viewing Transformation(s)

- Parallel or Orthographic projection
 - Eye at infinity
 - Need direction of projection
 - “Projectors” are parallel
- Perspective or Central projection
 - Eye at point (x,y,z) in world coordinates
 - “Projectors” emanate from eye position



Projections to be Considered

- Perspective
 - 1-point
 - 2-point
 - 3-point
- Parallel
 - Axonometric (isometric and dimetric)
 - Oblique

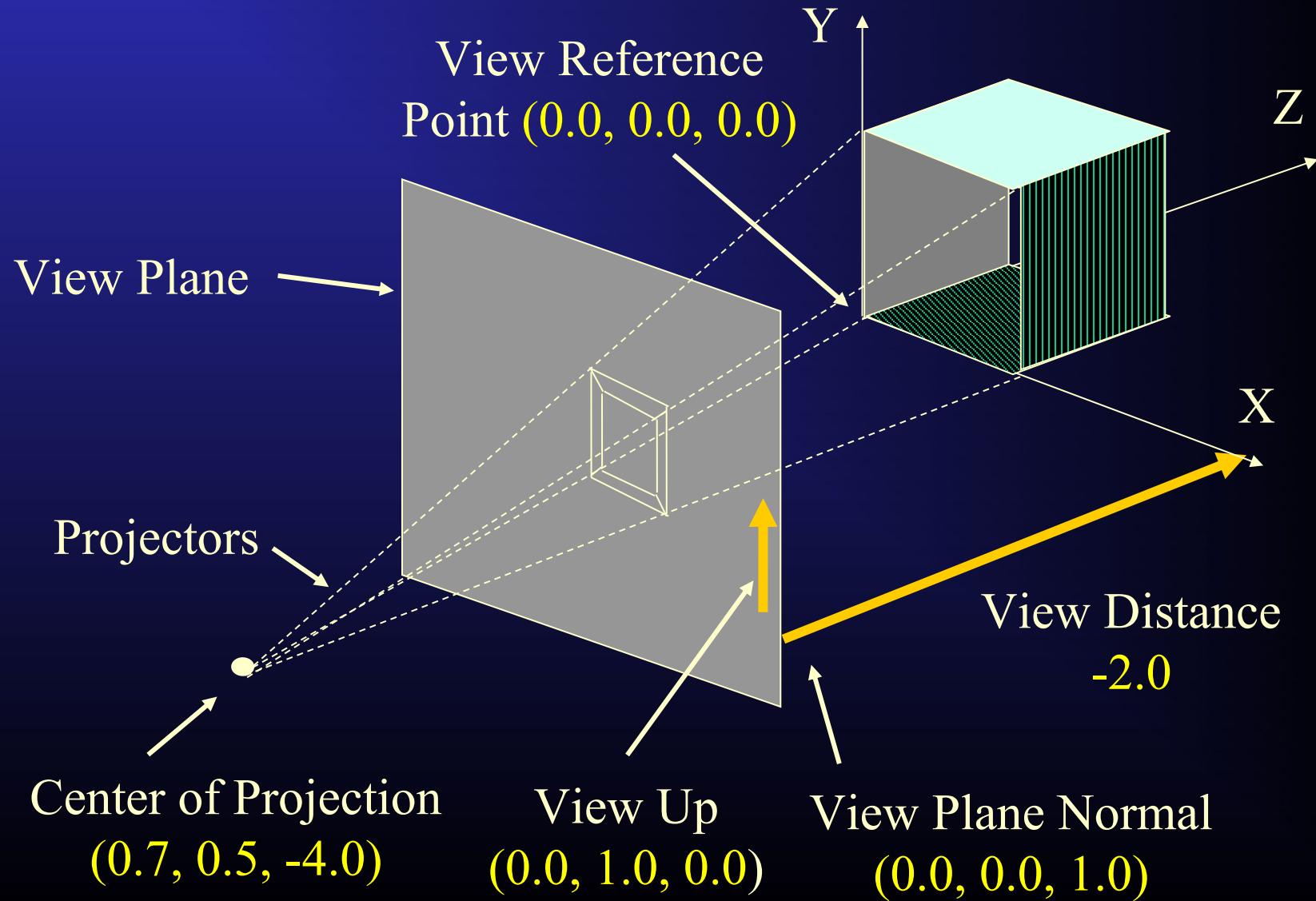
Defining a Perspective Projection

- Need:
 - Projection data and Center of Projection
- How to define these:
 1. **View reference point**: a point of interest (for now, assume point on the object)
 2. **View plane normal**: a direction vector
 3. **Center of projection**: a point defined relative to the view reference point
 4. **View plane distance**: defines a distance along the view plane normal from the view reference point. The view plane intersects the view plane normal at this distance from the view reference point.
 5. **View up vector**: The direction vector that will become “up” on the final image.

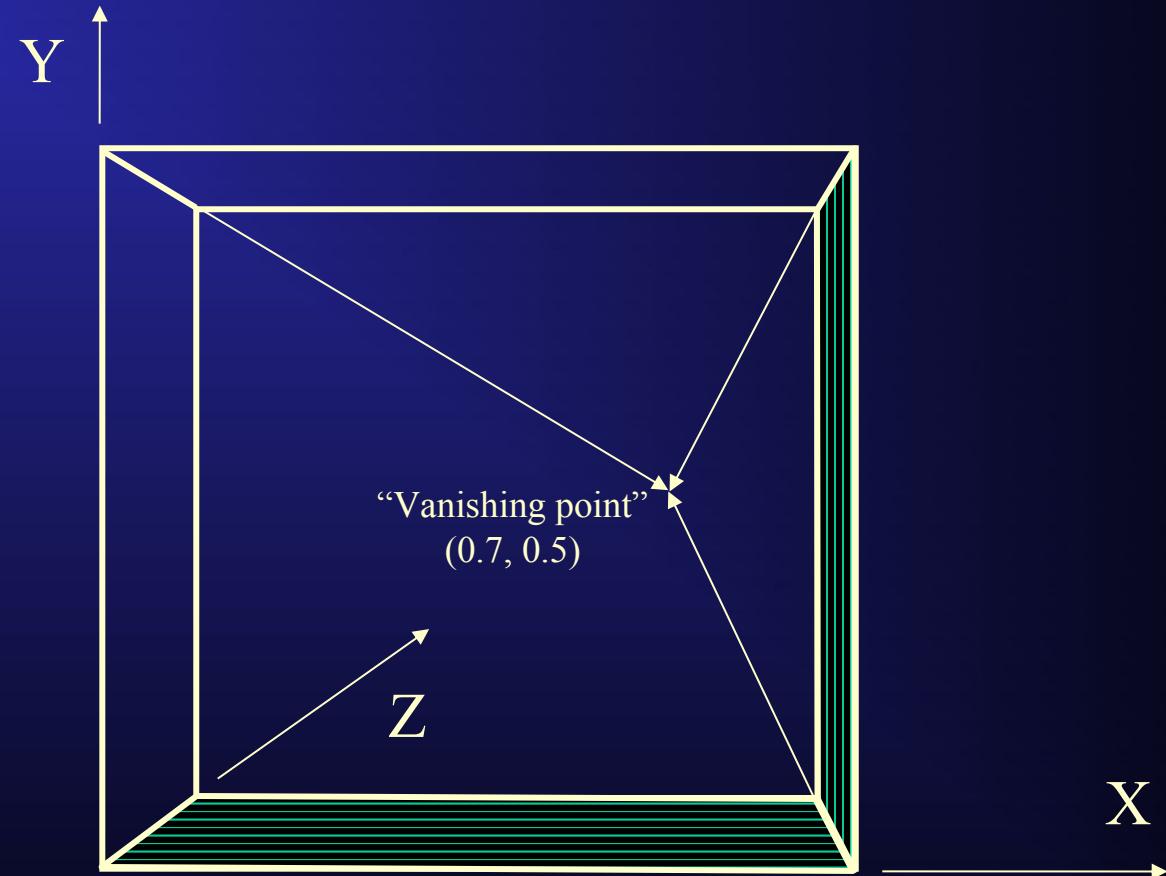
Defining a 1-Point Perspective

- Specifications for a 1-point perspective projection
 1. Let view reference point be origin.
 2. 1-point perspective \Rightarrow projection plane parallel to a principal face of object. Let it be parallel to xy-plane. View plane normal is $(0, 0, 1)$.
 3. Make center of projection so as to view xy-plane, to right of center of face of cube.
 4. Place the view plane between the object and the center of projection. (view distance = -2.0)
- Program segment to set up view:
 - SET_VIEW_REFERENCE_POINT (0.0, 0.0, 0.0)
 - SET_VIEW_PLANE_NORMAL (0.0, 0.0, 1.0)
 - SET_PROJECTION (PERSPECTIVE, 0.7, 0.5, -4.0)
 - SET_VIEW_PLANE_DISTANCE (-2.0)

One-Point Perspective Construction

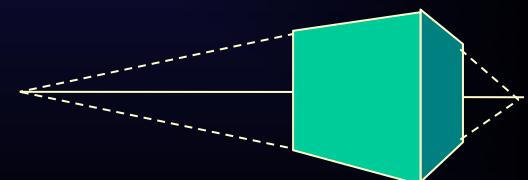


1-Point Perspective Projection of Cube



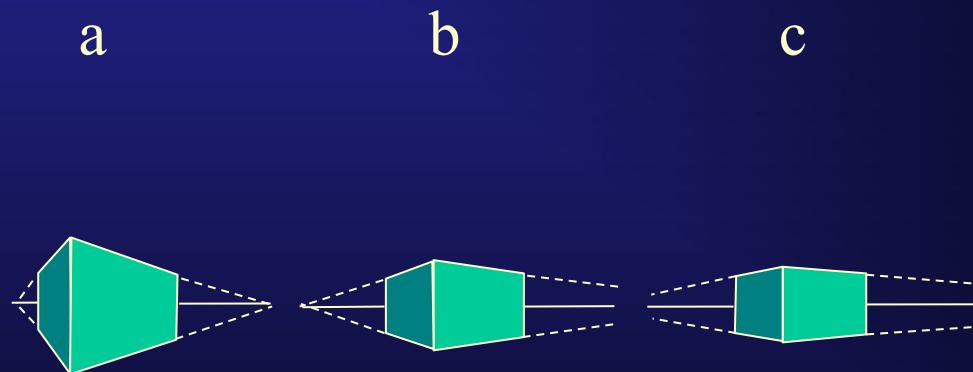
Defining a 2-point Perspective of a Cube

- Specification for a 2-point perspective projection
 - View reference point at origin $(0, 0, 0)$
 - 2-point perspective \Rightarrow projection plane intersects 2 of the principal axes. Assume view plane intersects x and z-axes at a 45 degree angle. View plane normal is $(1, 0, 1)$.
 - Center of projection directly opposite corner at $(-1.0, 0.5, -1.0)$
 - View distance is 0.71, putting view plane along diagonal of the cube.
- Program segment to obtain view shown:
 - SET_VIEW_REFERENCE_POINT $(0.0, 0.0, 0.0)$
 - SET_VIEW_PLANE_NORMAL $(1.0, 0.0, 1.0)$
 - SET_PROJECTION (PERSPECTIVE, $-1.0, 0.5, -1.0$)
 - SET_VIEW_PLANE_DISTANCE (0.71)



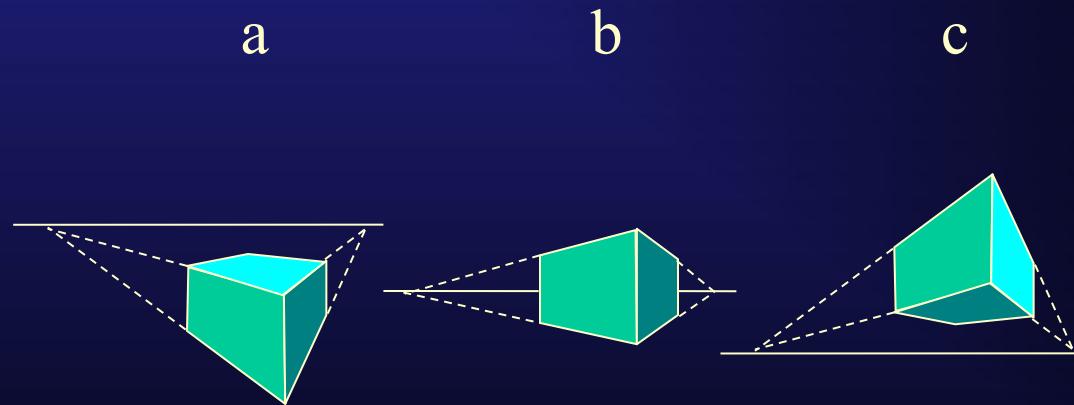
Varying the 2-Point Perspective Center of Projection Closer or Farther

- Variations achieved by moving the center of projection closer (a) or farther (c) from the view reference point.



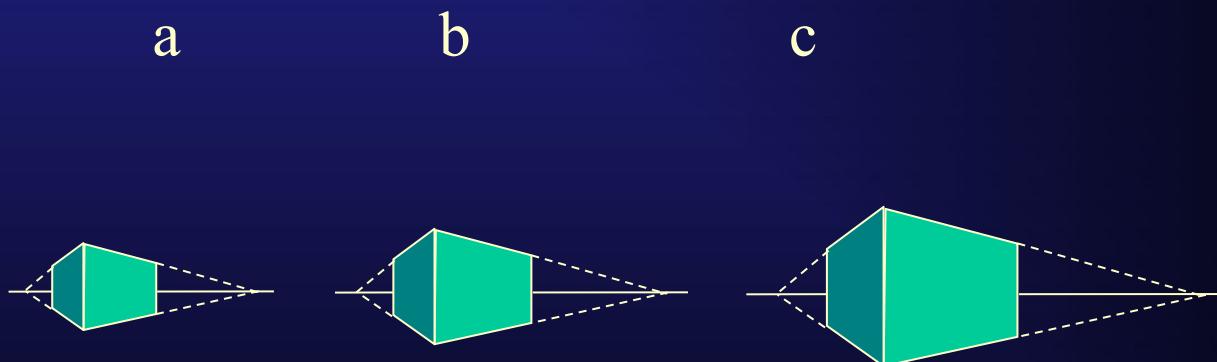
Varying the 2-Point Perspective Center of Projection Higher or Lower

- Variations achieved by moving the center of projection so as to show the top of the object (a) or the bottom of the object (c).



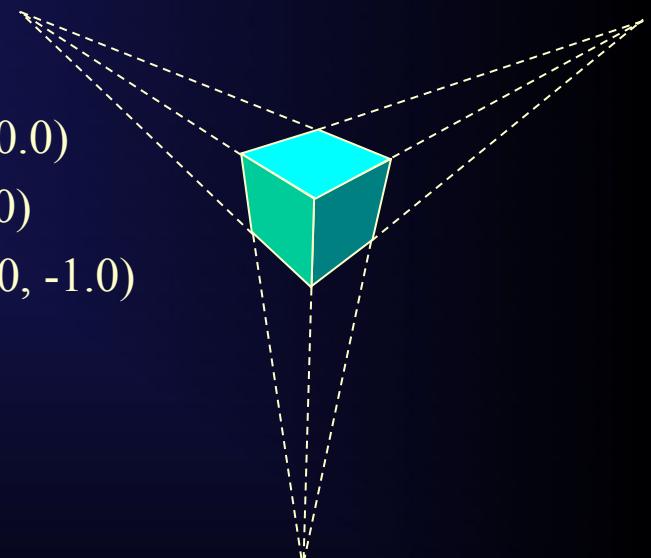
Varying the 2-Point Perspective View Plane Distance

- Effects achieved by varying the view plane distance. (a) has a large view plane distance, (c) a small view plane distance. The effect is the same as a scale change.



Defining a 3-Point Perspective Projection

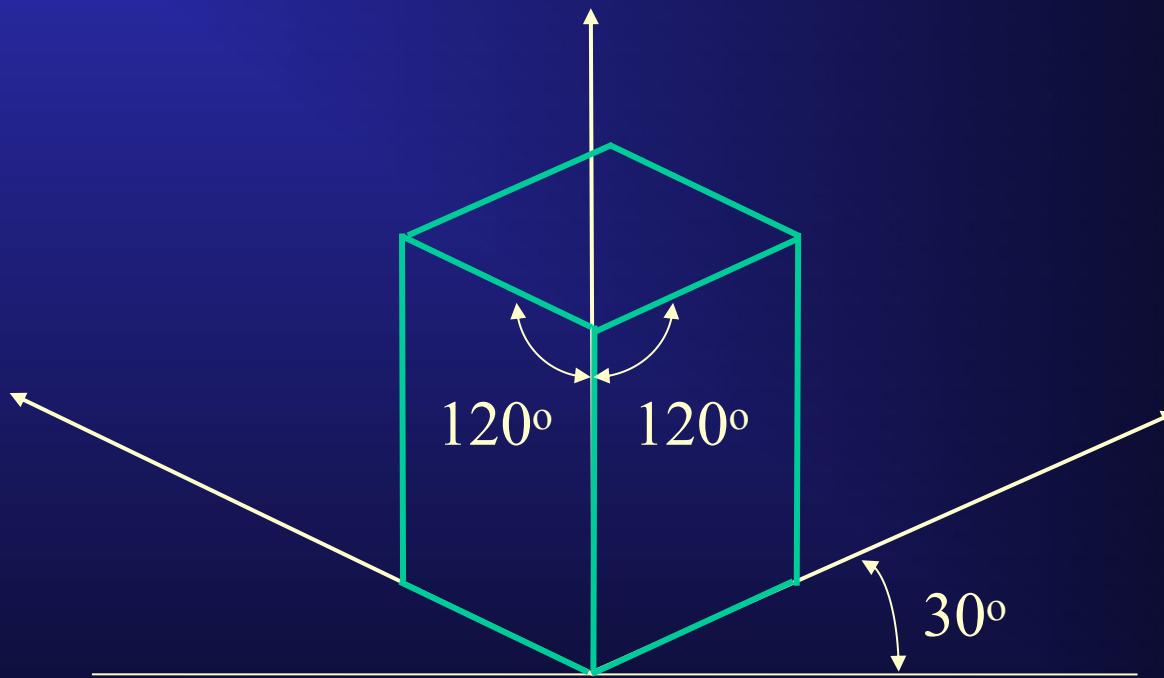
- Only difference from 2-point is that the projection plane intersects all three major axes.
- The view shown is generated by:
 - `SET_VIEW_REFERENCE_POINT (0.0, 1.0, 0.0)`
 - `SET_VIEW_PLANE_NORMAL (1.0, -1.0, 1.0)`
 - `SET_PROJECTION (PERSPECTIVE, -1.0, 1.0, -1.0)`
 - `SET_VIEW_PLANE_DISTANCE (-1.0)`



Defining Parallel Projections

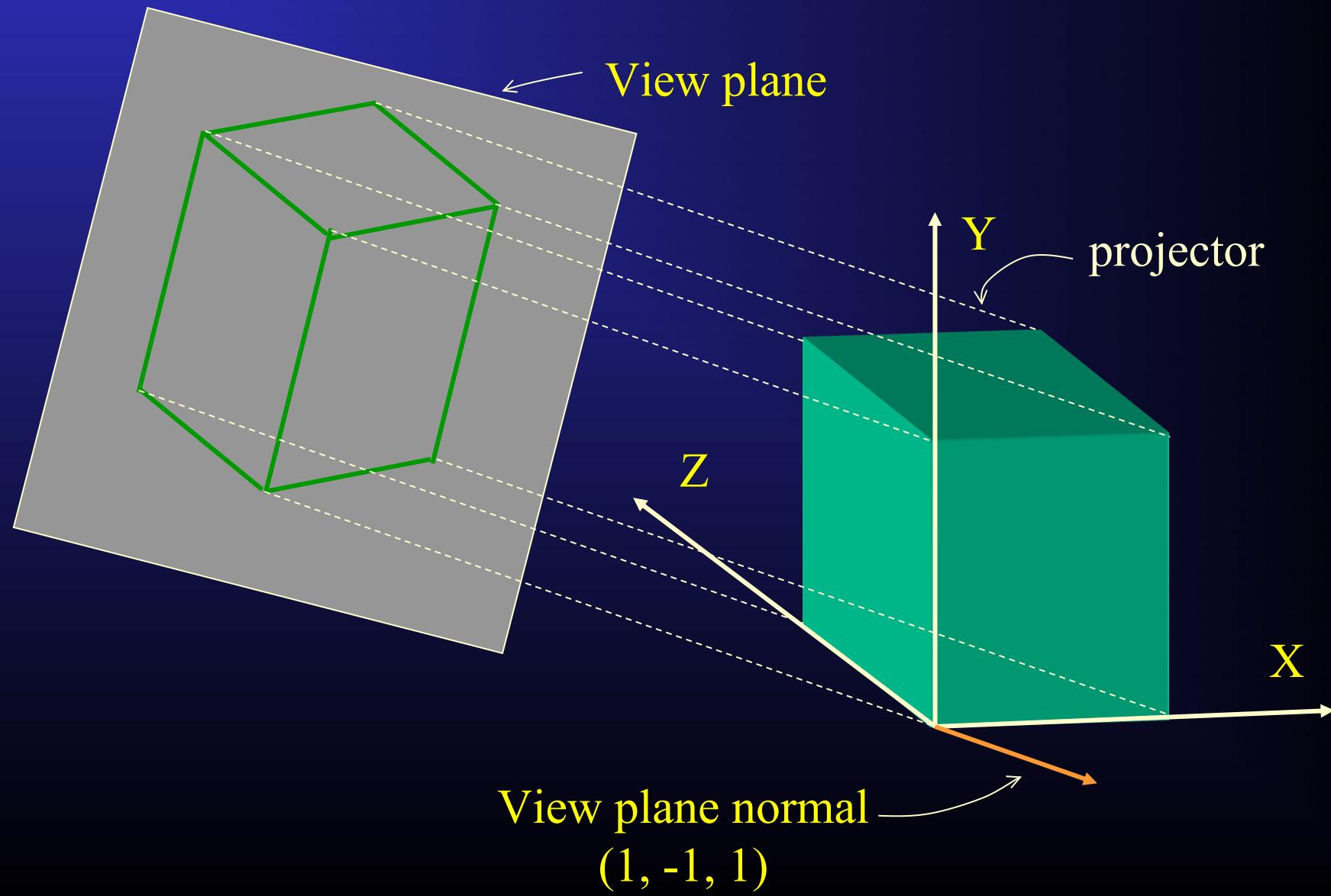
- Same as perspective except need a **direction** of projection rather than **center** of projection.
- Isometric projection: projection plane intersects the three coordinate axes at the same angle and the direction of projection is perpendicular to the projection plane.
- The isometric view in the next slide is specified by the following:
 - SET_VIEW_REFERENCE_POINT (0.0, 0.0, 0.0)
 - SET_VIEW_PLANE_NORMAL (1.0, -1.0, 1.0)
 - SET_PROJECTION (PARALLEL, 1.0, -1.0, 1.0)
 - SET_VIEW_PLANE_DISTANCE (-2.5) (* no effect on projection *)

Parallel Projection of Cube



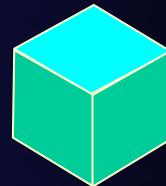
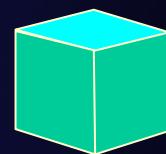
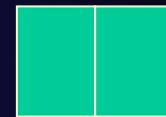
Notice how all parallel line families map into parallel lines in the projection.

Construction of Parallel Projection



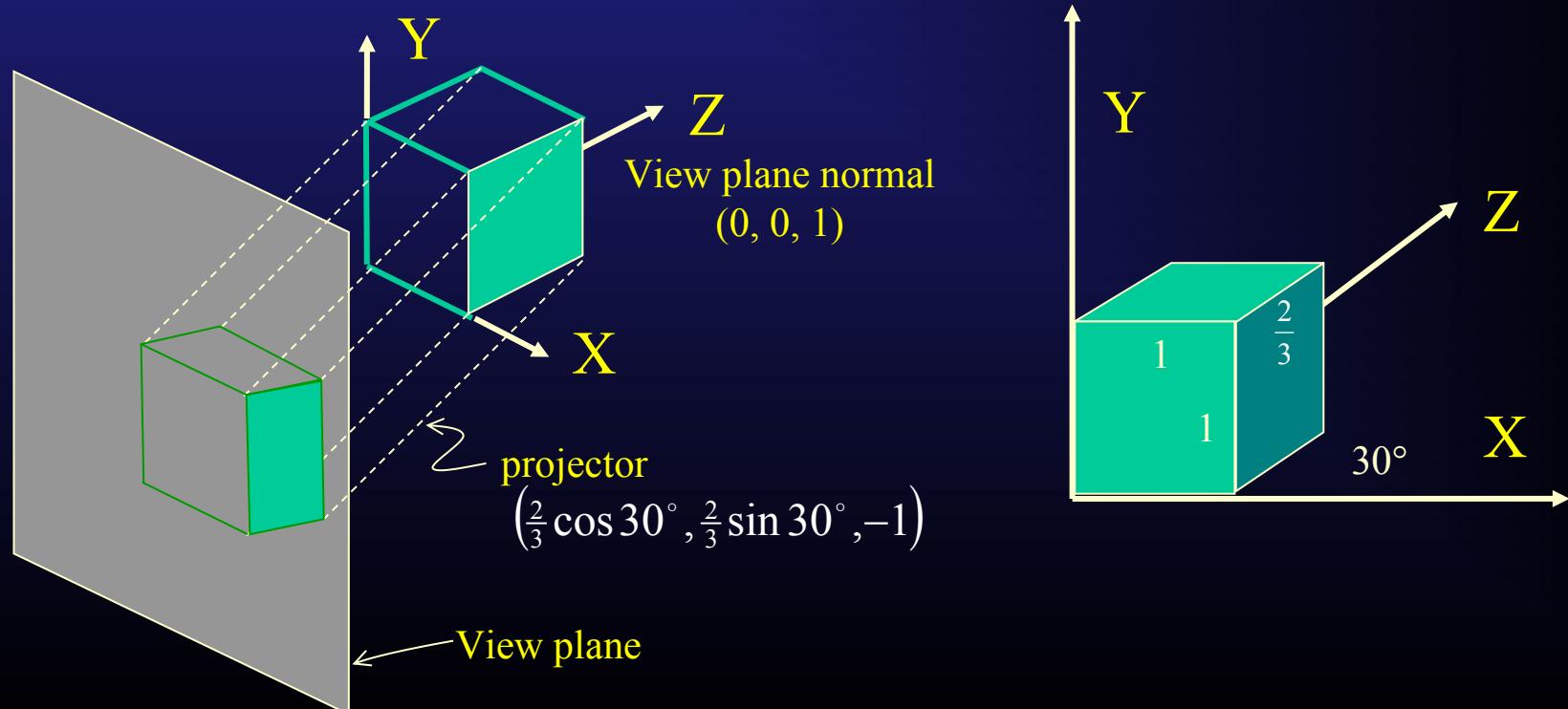
Dimetric Projections

- By changing the view plane normal and direction of projection, we can get the series of views shown next. (Note that for all axonometric projections the parameters to SET_VIEW_PLANE_NORMAL can be used to define the direction of projection.)
 - a. SET_VIEW_PLANE_NORMAL (1.0, 0.0, 1.0)
SET_PROJECTION(PARALLEL, 1.0, 0.0, 1.0)
 - b. SET_VIEW_PLANE_NORMAL (1.0, -0.5, 1.0)
SET_PROJECTION(PARALLEL, 1.0, -0.5, 1.0)
 - c. SET_VIEW_PLANE_NORMAL (1.0, -1.0, 1.0)
SET_PROJECTION(PARALLEL, 1.0, -1.0, 1.0)
 - d. SET_VIEW_PLANE_NORMAL (1.0, -1.5, 1.0)
SET_PROJECTION(PARALLEL, 1.0, -1.5, 1.0)
 - e. SET_VIEW_PLANE_NORMAL (1.0, -1.0, 1.0)
SET_PROJECTION(PARALLEL, 1.0, -1.0, 1.0)



Oblique Parallel Projections

- The direction of projection is not perpendicular to the view plane.
- The view shown is defined by the following:
 - SET_VIEW_REFERENCE_POINT (0.0, 0.0, 0.0);
 - SET_VIEW_PLANE_NORMAL (0.0, 0.0, 1.0);
 - SET_PROJECTION (PARALLEL, $\frac{2}{3}\cos 30^\circ$, $\frac{2}{3}\sin 30^\circ$, -1);
 - SET_VIEW_PLANE_DISTANCE (0.0);



Varying the Direction of Projection Angle

- SET_PROJECTION (PARALLEL,

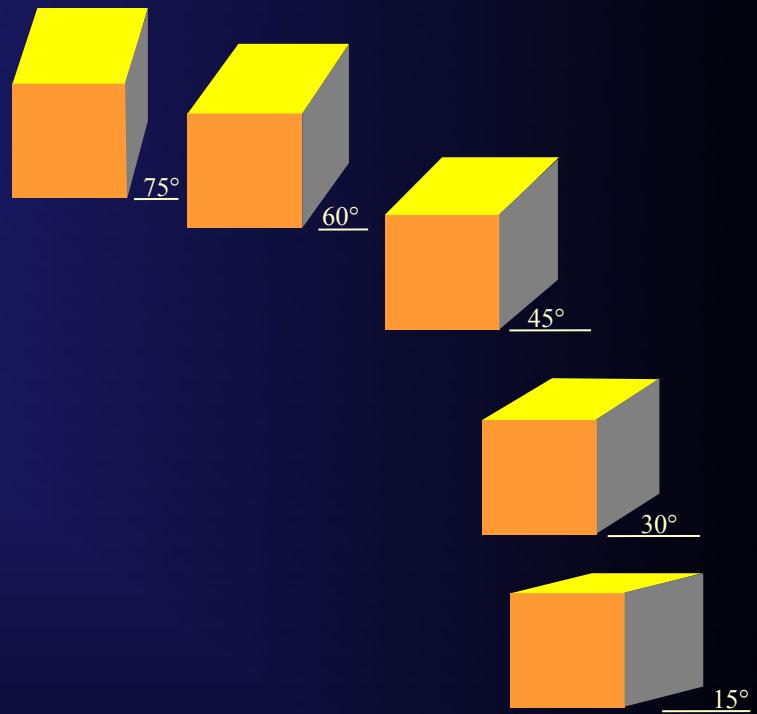
$$\frac{2}{3} \cos 75^\circ, \frac{2}{3} \sin 75^\circ, -1)$$

$$\frac{2}{3} \cos 60^\circ, \frac{2}{3} \sin 60^\circ, -1)$$

$$\frac{2}{3} \cos 45^\circ, \frac{2}{3} \sin 45^\circ, -1)$$

$$\frac{2}{3} \cos 30^\circ, \frac{2}{3} \sin 30^\circ, -1)$$

$$\frac{2}{3} \cos 15^\circ, \frac{2}{3} \sin 15^\circ, -1)$$



Varying the Foreshortening Ratio

- SET_PROJECTION (PARALLEL,

$$\cos 45^\circ, \sin 45^\circ, -1)$$



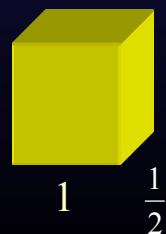
$$\frac{3}{4} \cos 45^\circ, \frac{3}{4} \sin 45^\circ, -1)$$



$$\frac{2}{3} \cos 45^\circ, \frac{2}{3} \sin 45^\circ, -1)$$

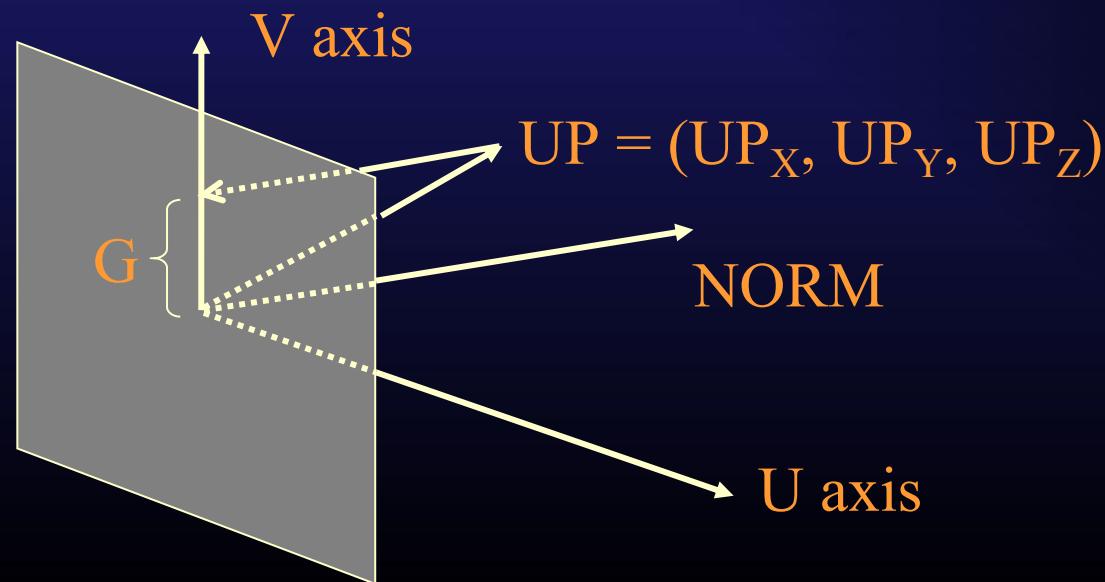


$$\frac{1}{2} \cos 45^\circ, \frac{1}{2} \sin 45^\circ, -1)$$

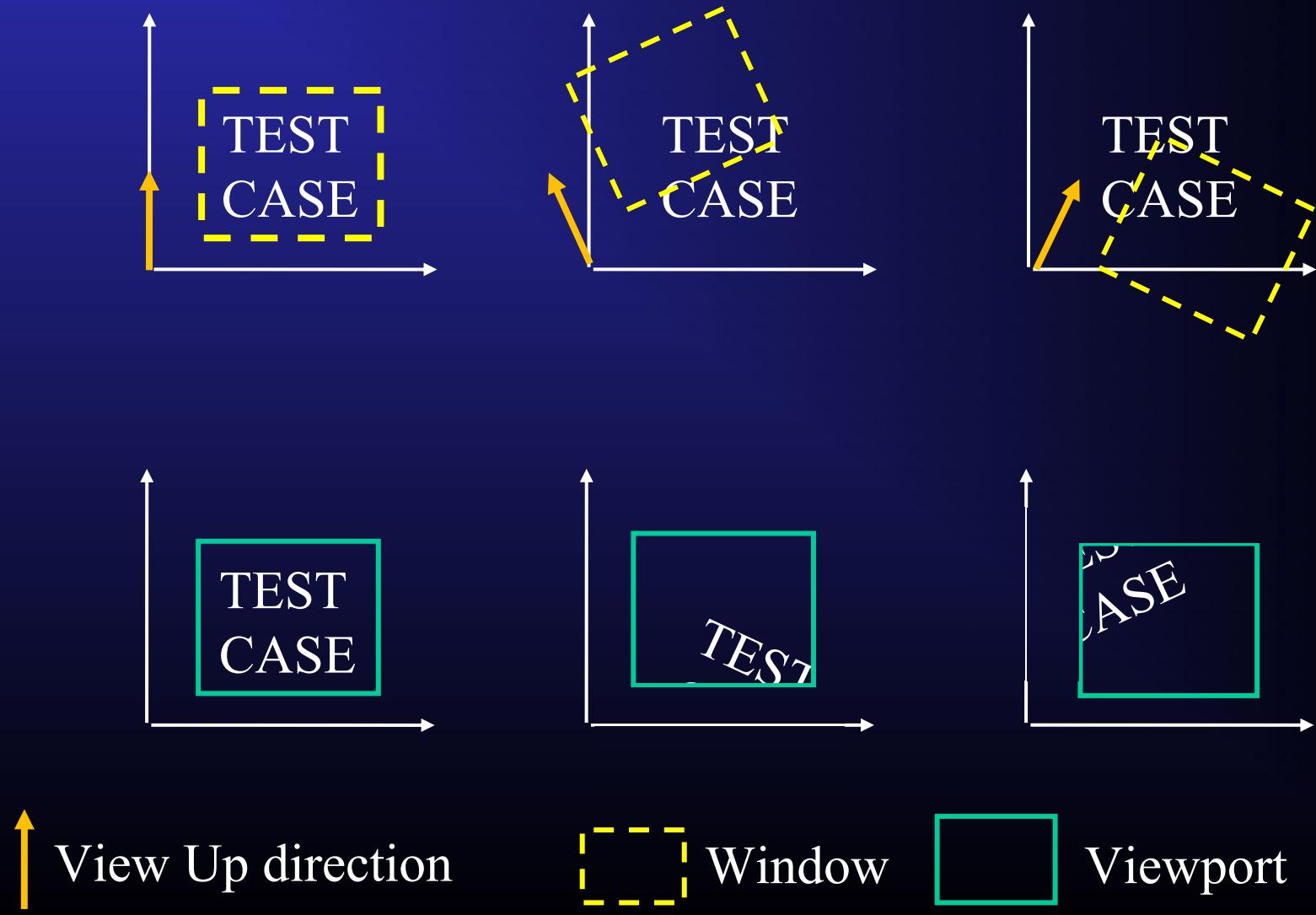


The View Up Vector

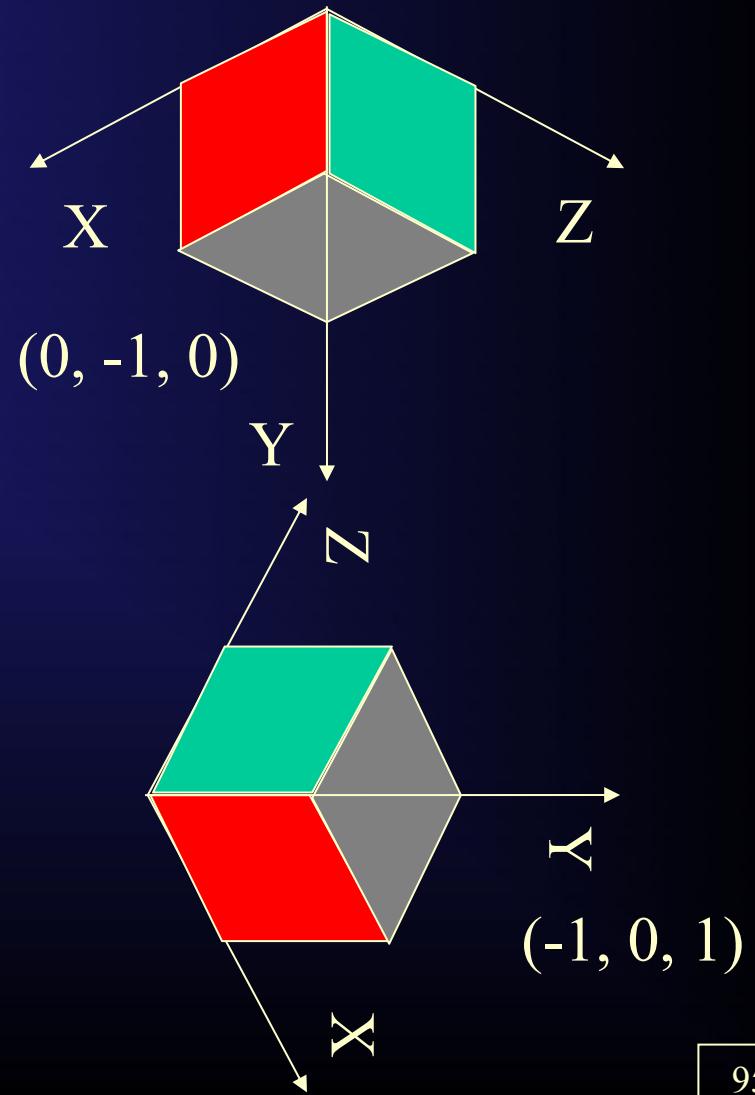
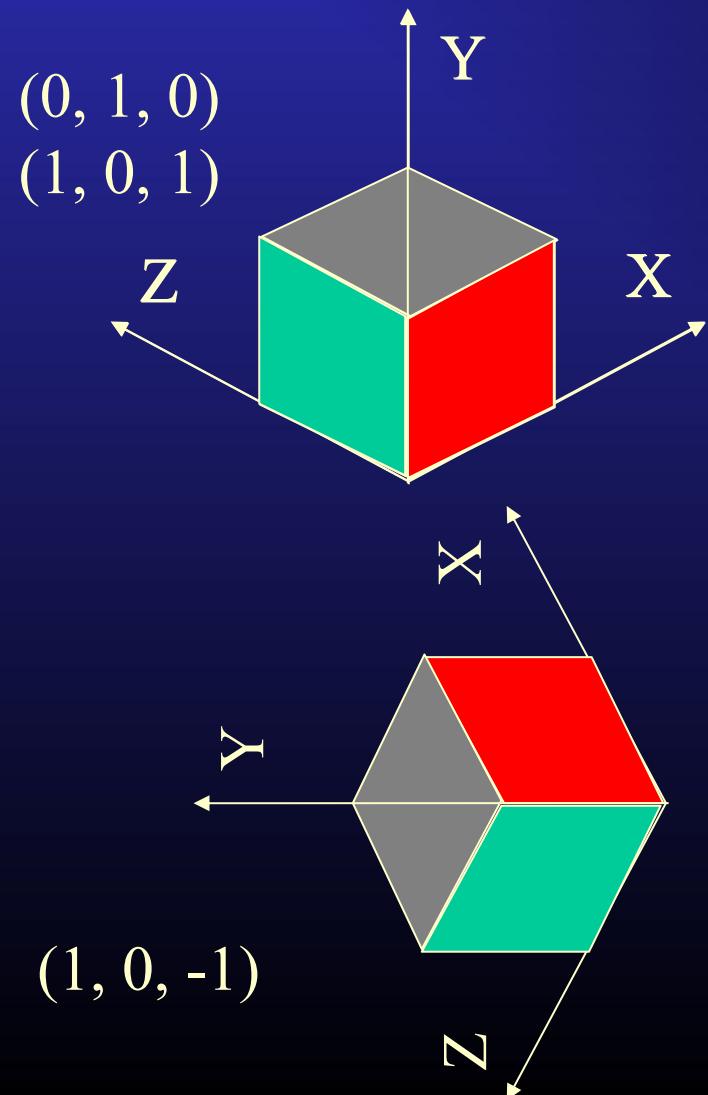
- Specify direction which will become vertical in the final image: **View Up**.
- Project **View Up** vector onto **view plane** in **view plane normal** direction; call it **G** (like “**gravity**”).
- Watch for degenerate case:
 - when **view up** points in same direction as **view plane normal**.



Effect of View Up Vector



Examples of View Up in 3D (VIEW UP =)



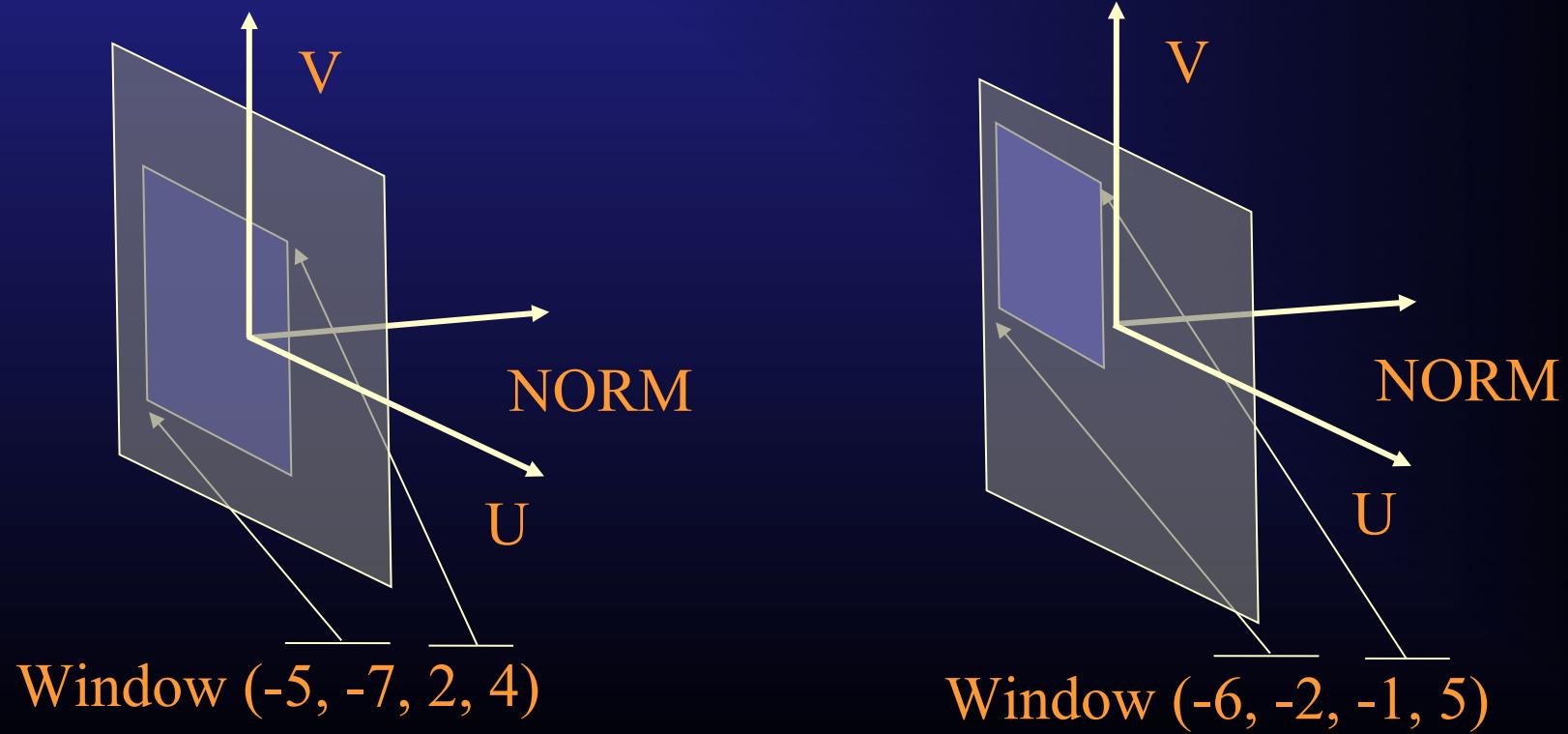
View Plane U - V Coordinate System

- Origin is the view reference point REF.
- Create a parameterization for this plane to specify the 2D coordinates of the window:
- V axis computed from view up vector G.
- U axis computed from view plane normal \times V.



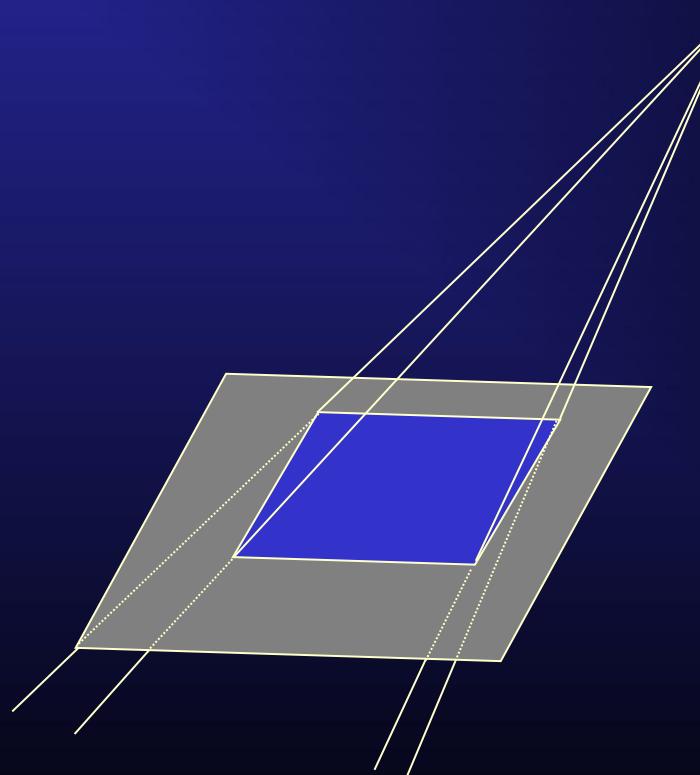
Specifying the Window

- Needed for window to viewport mapping (as in 2D case).
- Creates top, bottom, left, and right clipping limits.

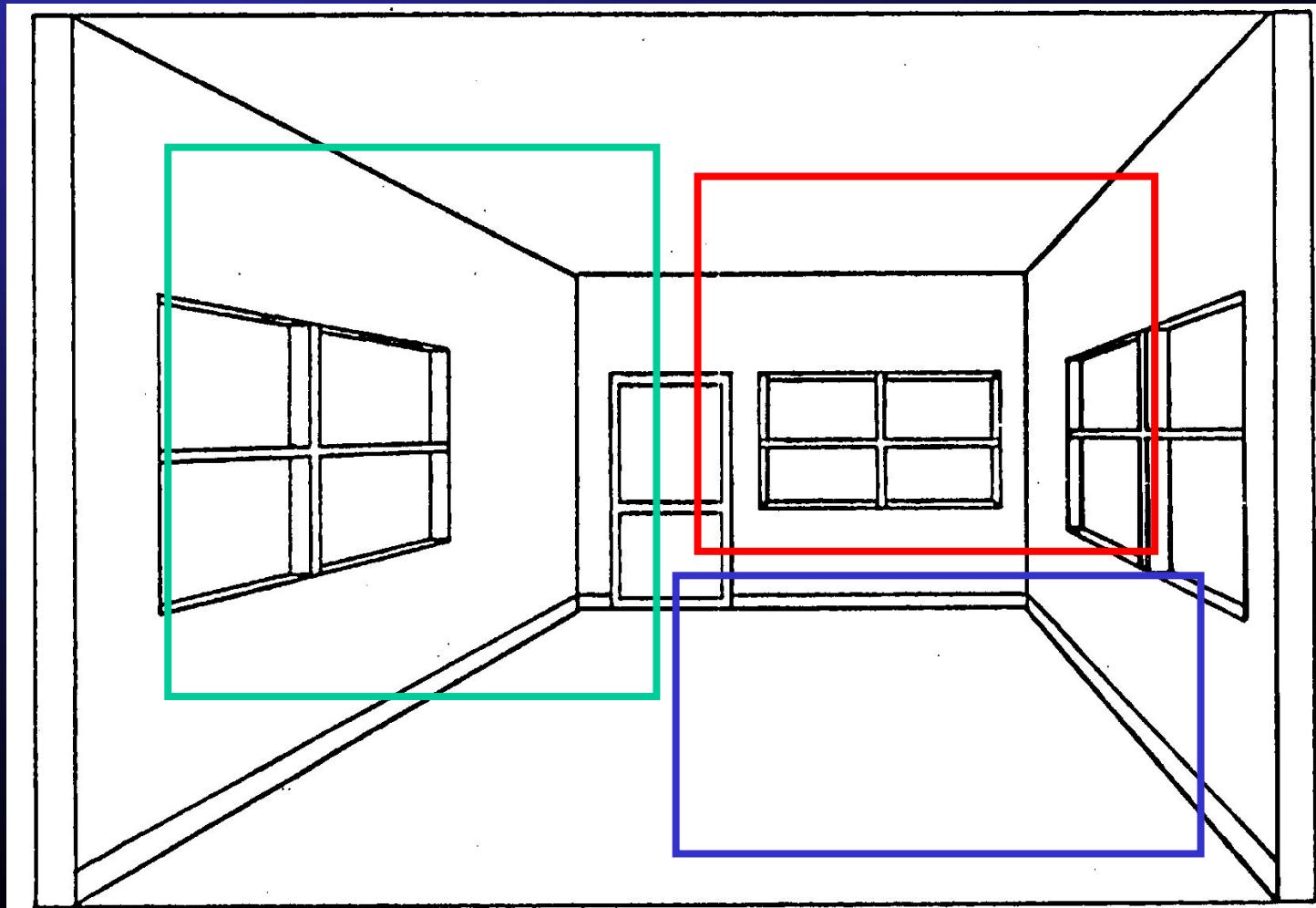


Window Changes Create Cropping Effects

- Window need not be centered in U - V coordinate system.
- Like cropping a photograph.

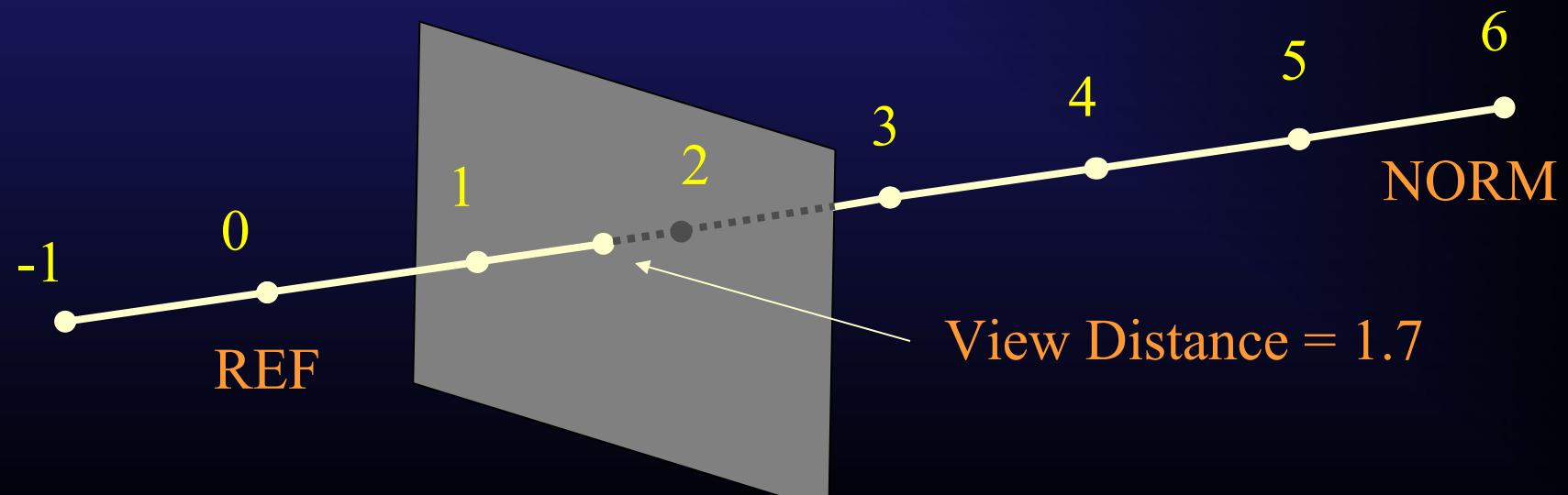


Cropping Example



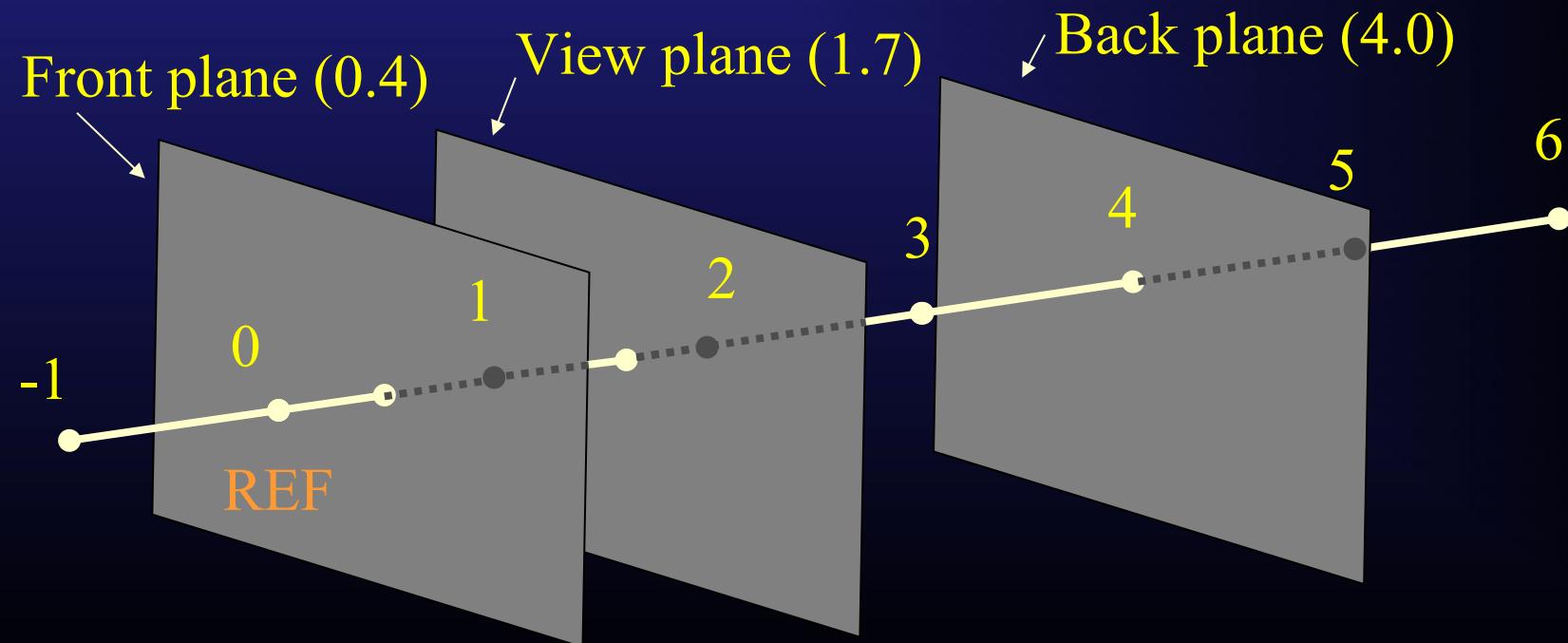
Setting the View Plane Distance

- Distance from view reference point to view plane along view plane normal (NORM).
- View reference point (REF) can be anywhere convenient:
 - on the object
 - at the center of the perspective projection
- Used to zoom in and out.



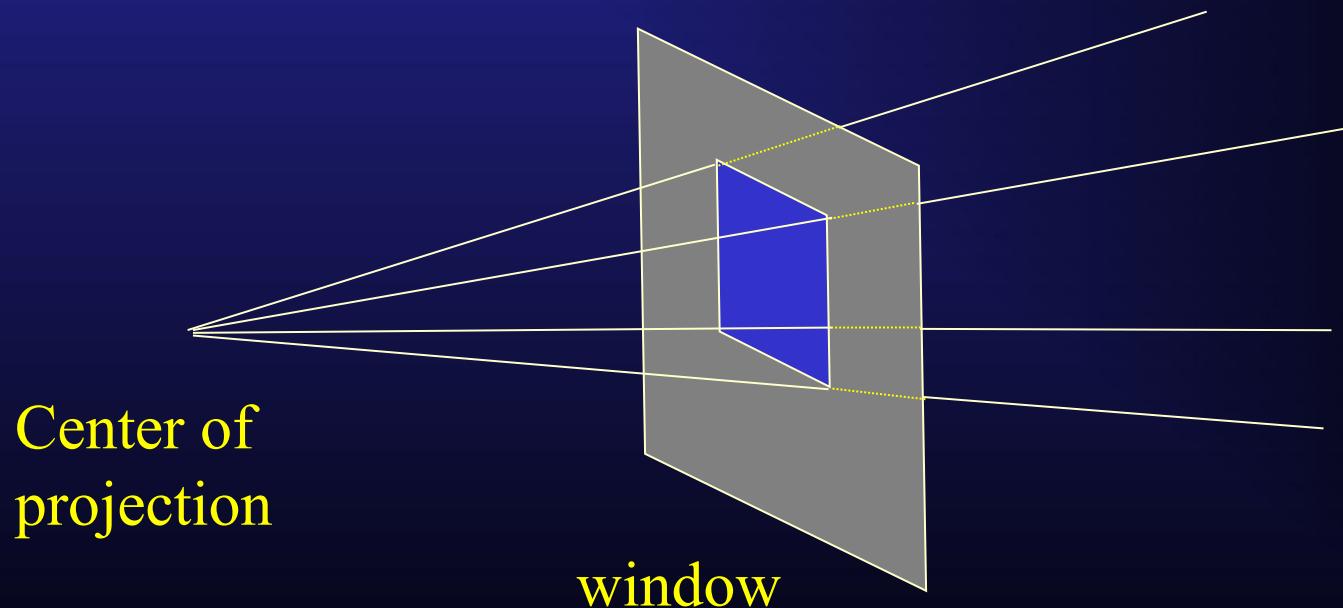
Setting the View Depth -- Front and Back Planes

- Front and Back are planes parallel to view plane.
- Forms front and back of view volume.
- (Do not confuse with front and back buffers!)
- Used for both perspective and parallel projections.
- Any order as long as front < back.



3D Clipping and the View Volume

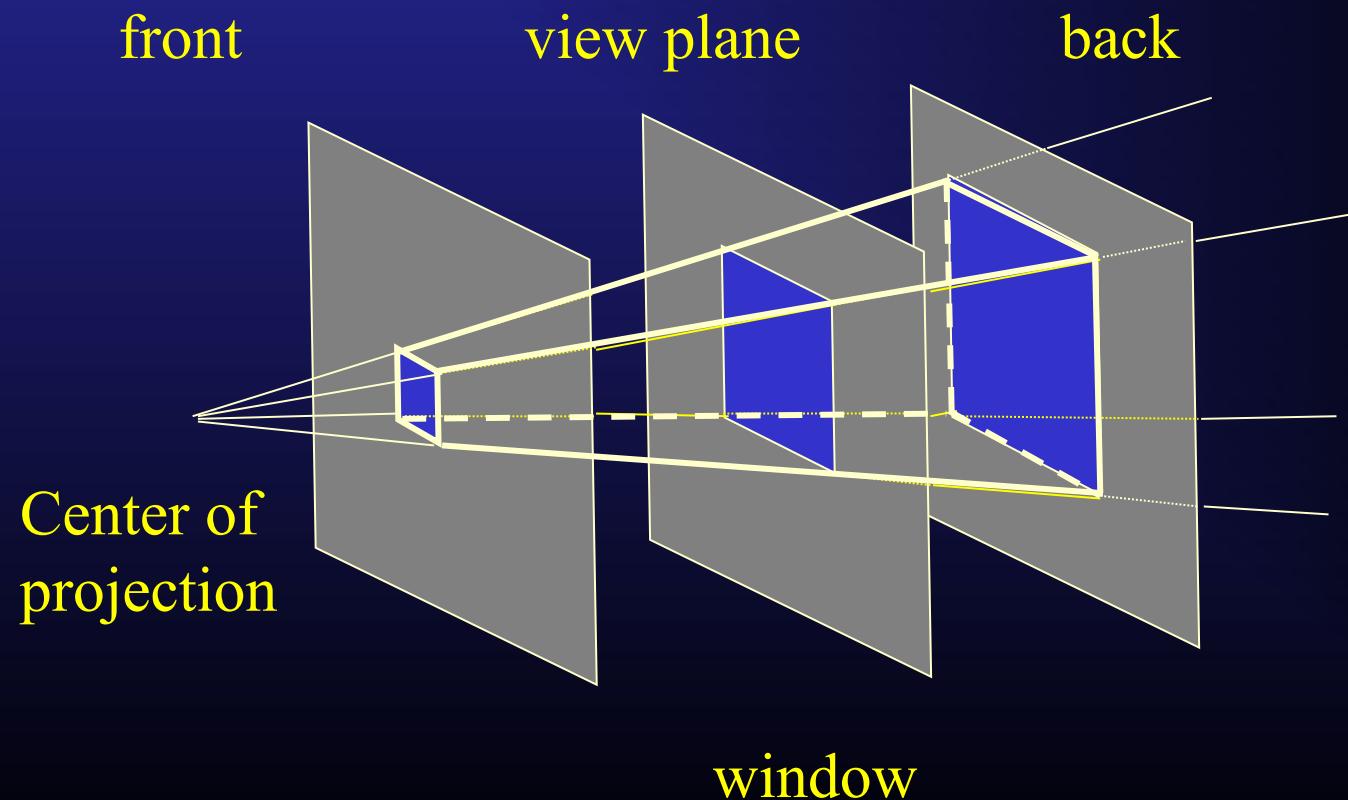
Graphic primitives are clipped to the **view volume** formed by:
the 4 planes formed by the top, bottom, left, and right
edges of the **window** and the **center of projection**, and...



The contents of the **view volume** are projected onto the **window**.

3D Clipping uses the Front and Back Planes, too

- Front and back planes truncate the view volume pyramid.



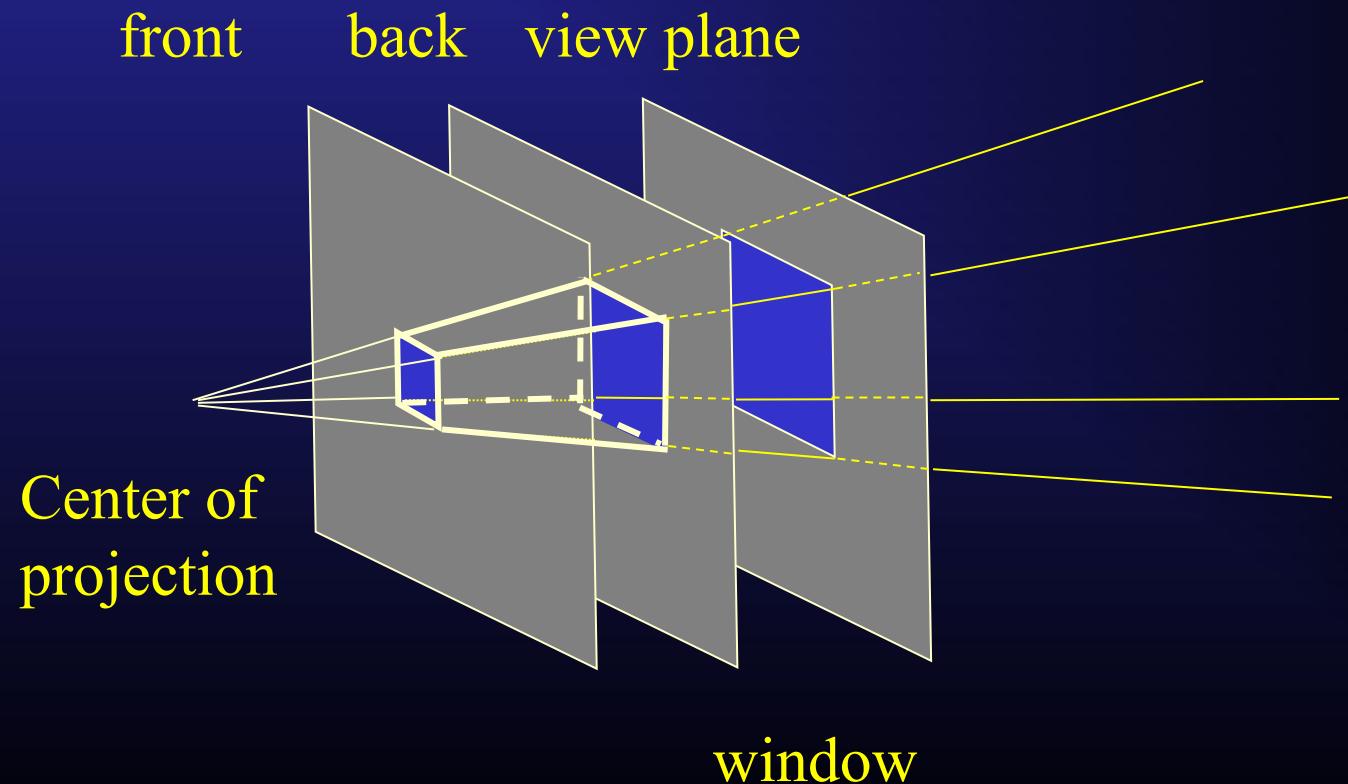
The Perspective Projection View Volume

Voilà! The **truncated pyramid view volume** or **frustum**.

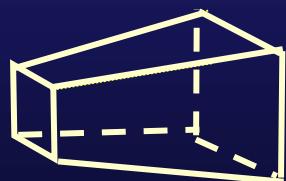


Only Condition on Front and Back: Front < Back

- These are used to clip data, not to specify the projection.

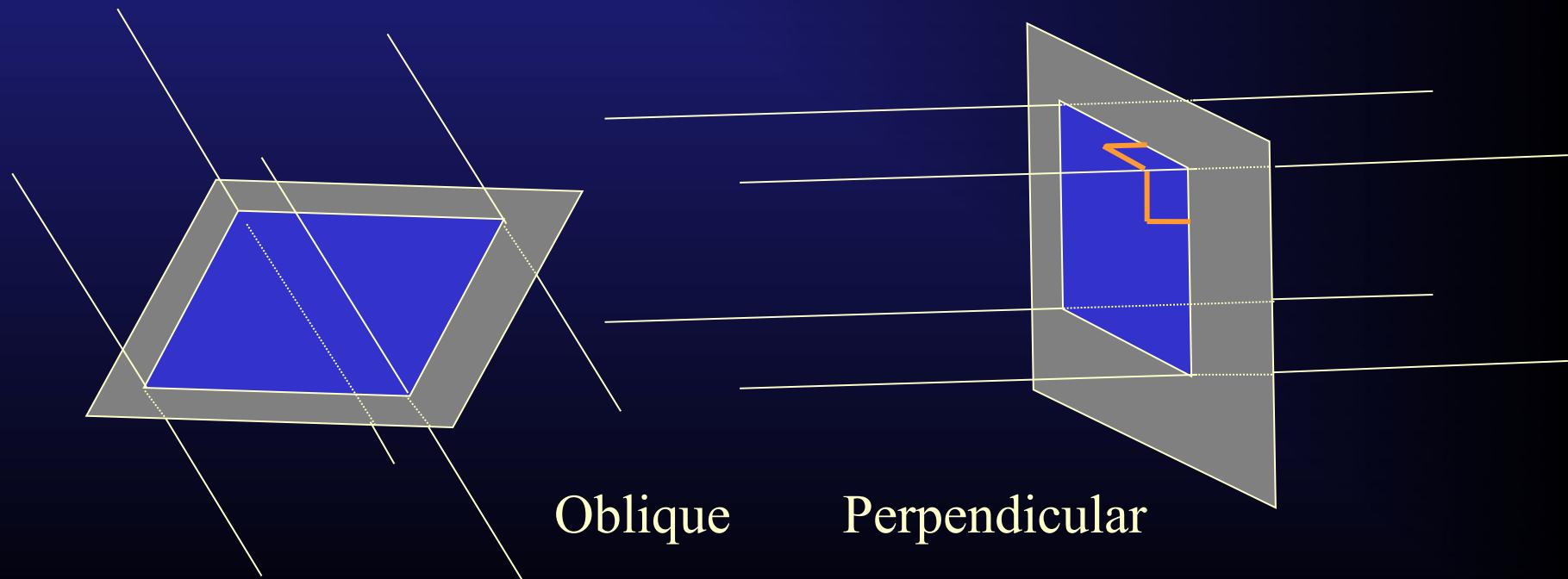


Only Condition on Front and Back: Front < Back



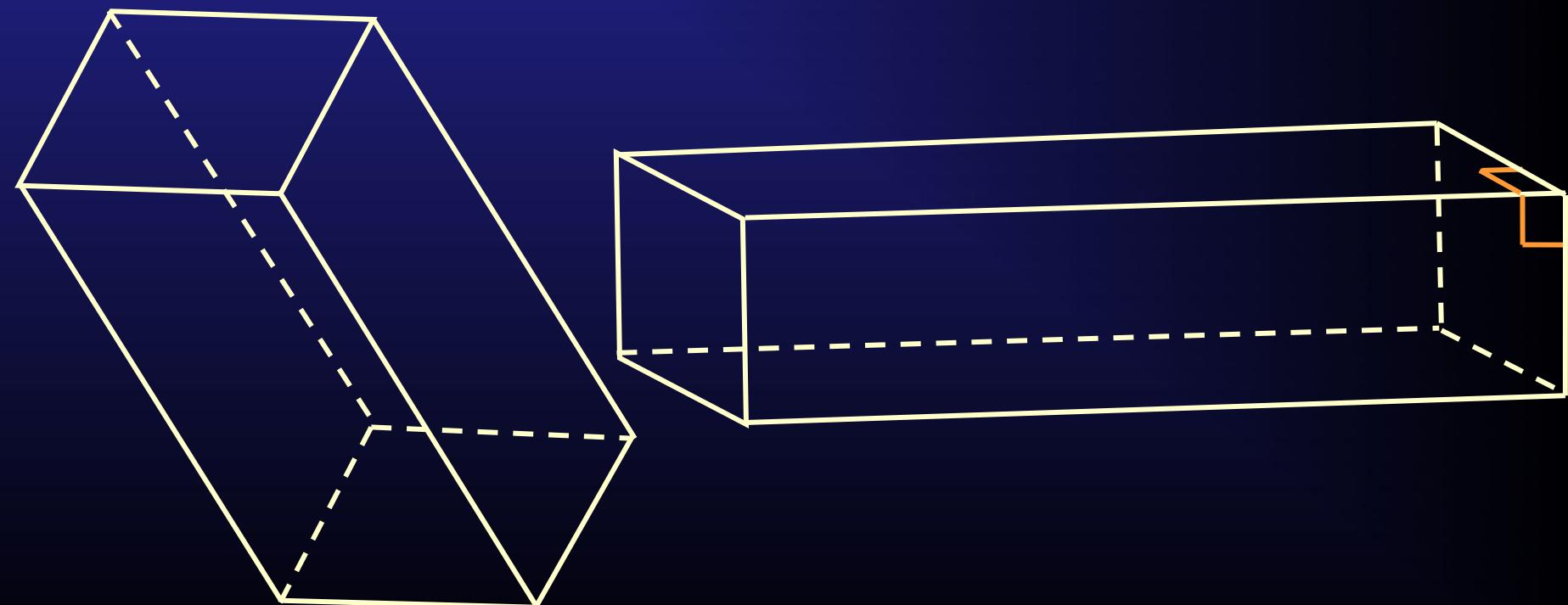
Parallel Projection Clipping View Volume

- View Volume determined by the direction of projection and the window (as defined in the perspective case).



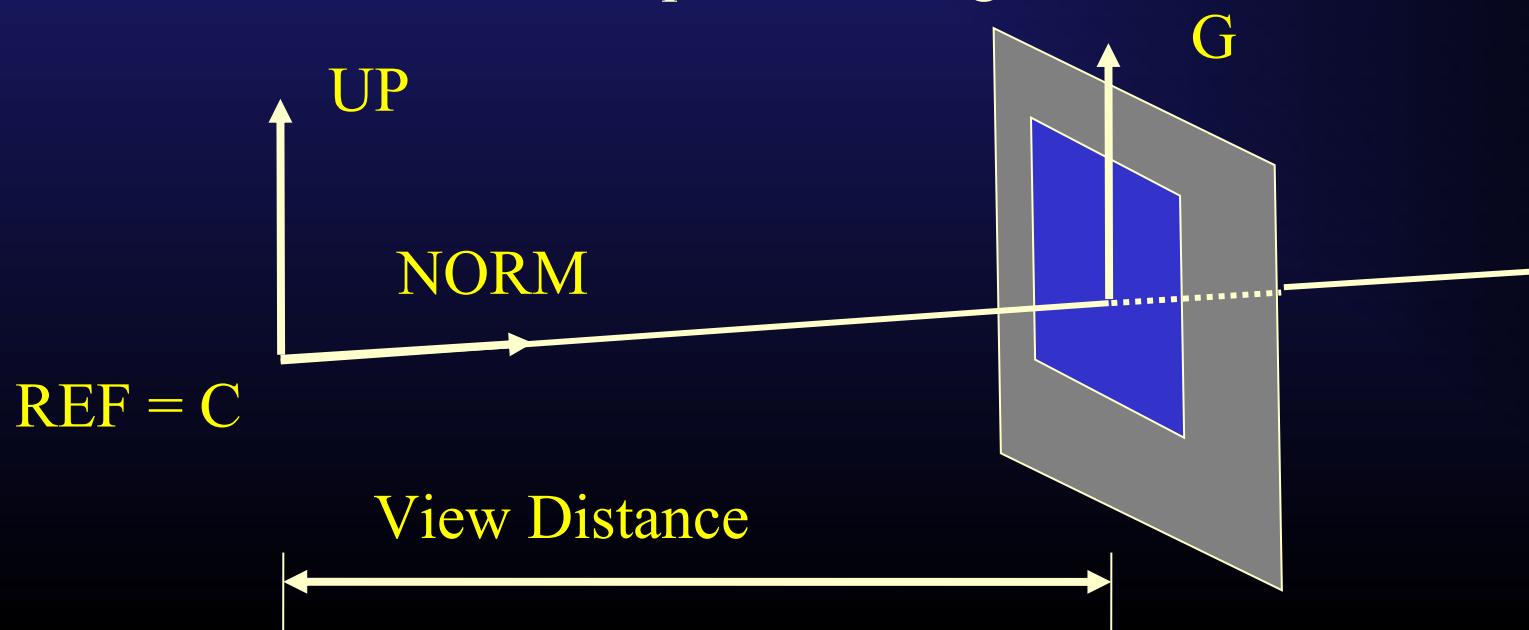
Parallel Projection View Volume Clipped to Front and Back Planes

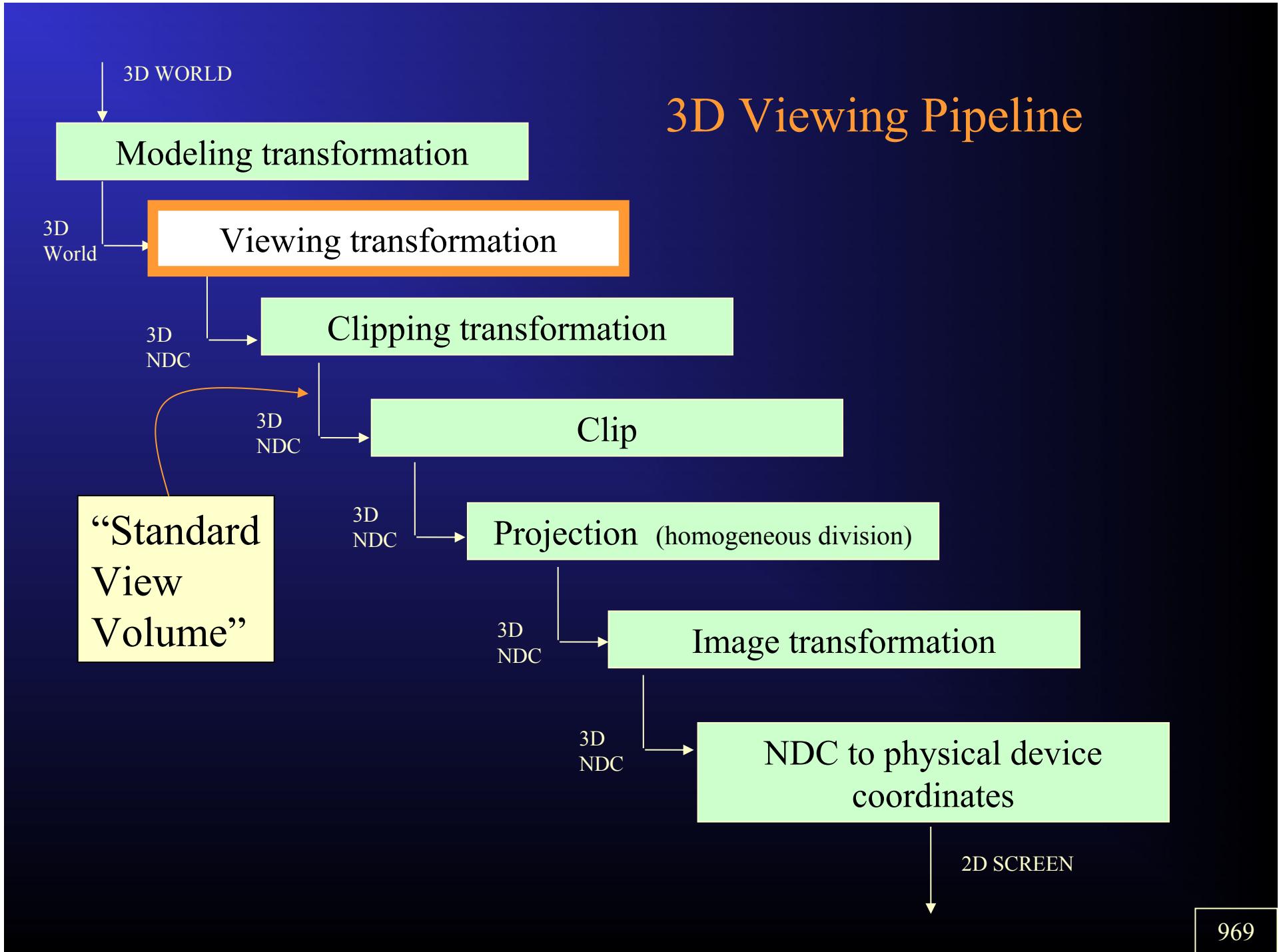
- View Volume is now a **parallelopiped**.



A Synthetic “Camera”

- If perspective center of projection = view reference point, image appears like a *synthetic camera*:
 - Translated via **REF** changes.
 - Rotated via **UP** changes.
 - Redirected via **View Plane Normal** changes (e.g. panning).
 - Zoom via changes in **View Distance**
 - No **Front** and **Back** planes, though!





Advantages of a Standard Coordinate System

- Clipping is easier with clipping planes in a standard position.
- The calculations are easier (and hence easier to do in hardware).
- Allows arbitrary world coordinate systems.
- Application independent.
- Subscript **C** will denote the clipping (transformed) coordinates.

Transform World Coordinates to Eye Coordinates

Approximate steps:

- Put eye (center of projection) at $(0, 0, 0)_C$
- Make $+X_C$ point to right.
- Make $+Y_C$ point up.
- Make $+Z_C$ point forward (away from eye in depth).
- (This is now a *left-handed* coordinate system!)

$$[x_e \quad y_e \quad z_e \quad 1]^T = [V][x \quad y \quad z \quad 1]^T \quad \text{for} \quad [V] = [R][T]$$

$[T]$ translates eye to $(0, 0, 0)_C$

$[R]$ rotates view direction to lie along $+Z_C$

World to Eye Transformation

How do we see the model from the view direction?

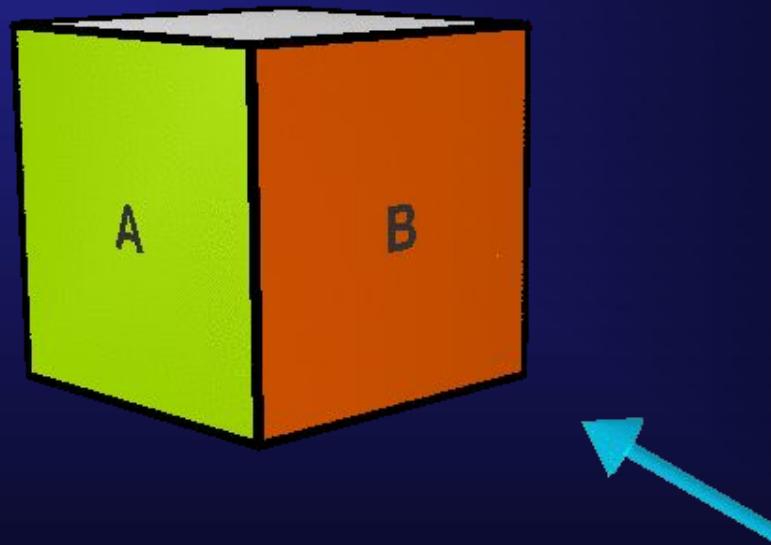
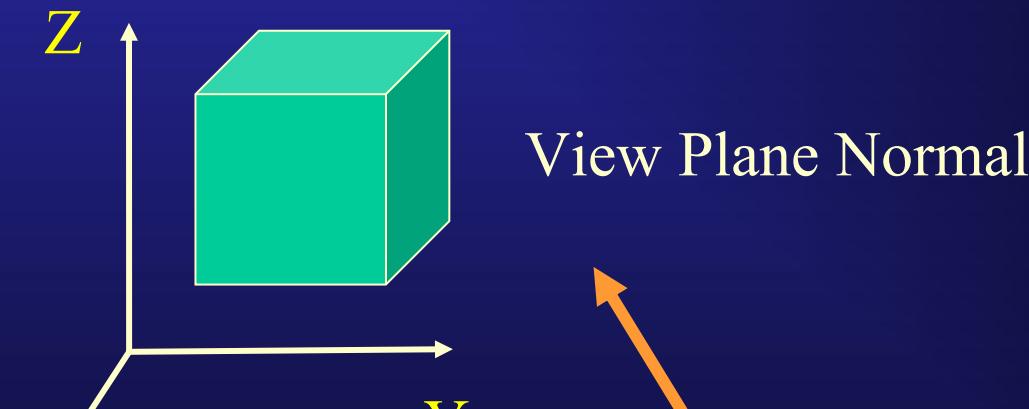


Diagram of World to Eye Transformation

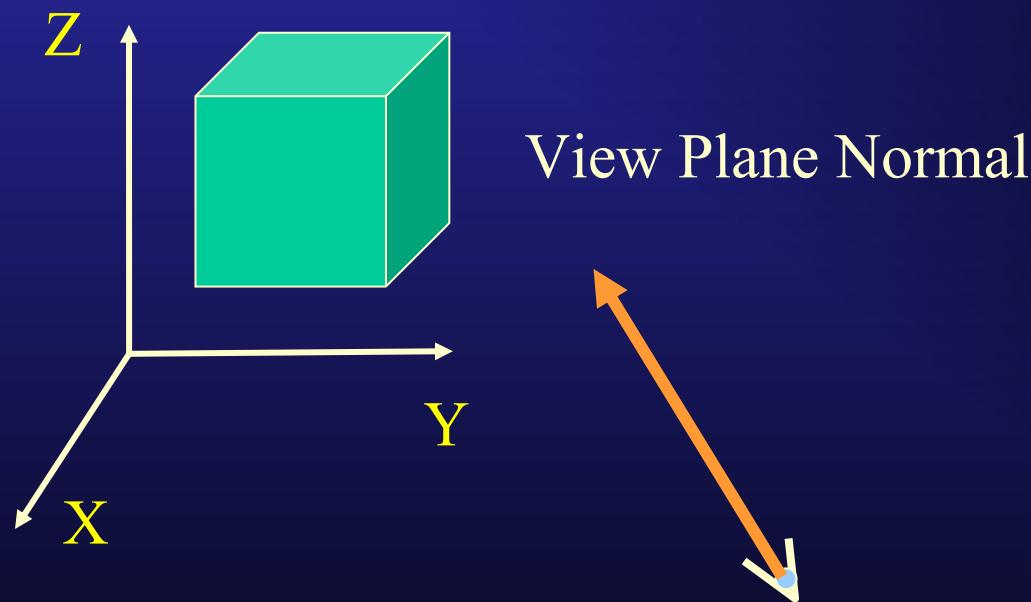
START



Eye =
center of projection

Diagram of World to Eye Transformation

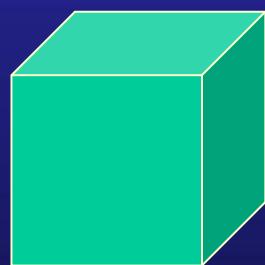
Translate eye to $(0, 0, 0)$



Eye =
center of projection

Diagram of World to Eye Transformation

Translate eye to $(0, 0, 0)$



View Plane Normal

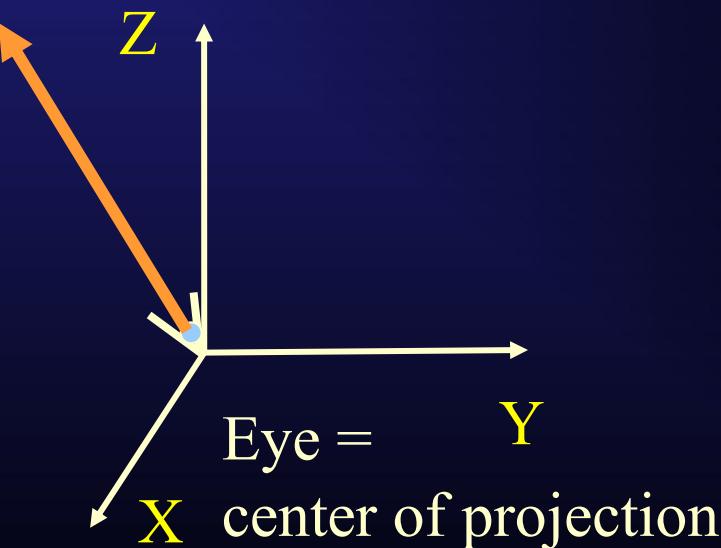


Diagram of World to Eye Transformation
Rotate to align View Plane Normal with +Z

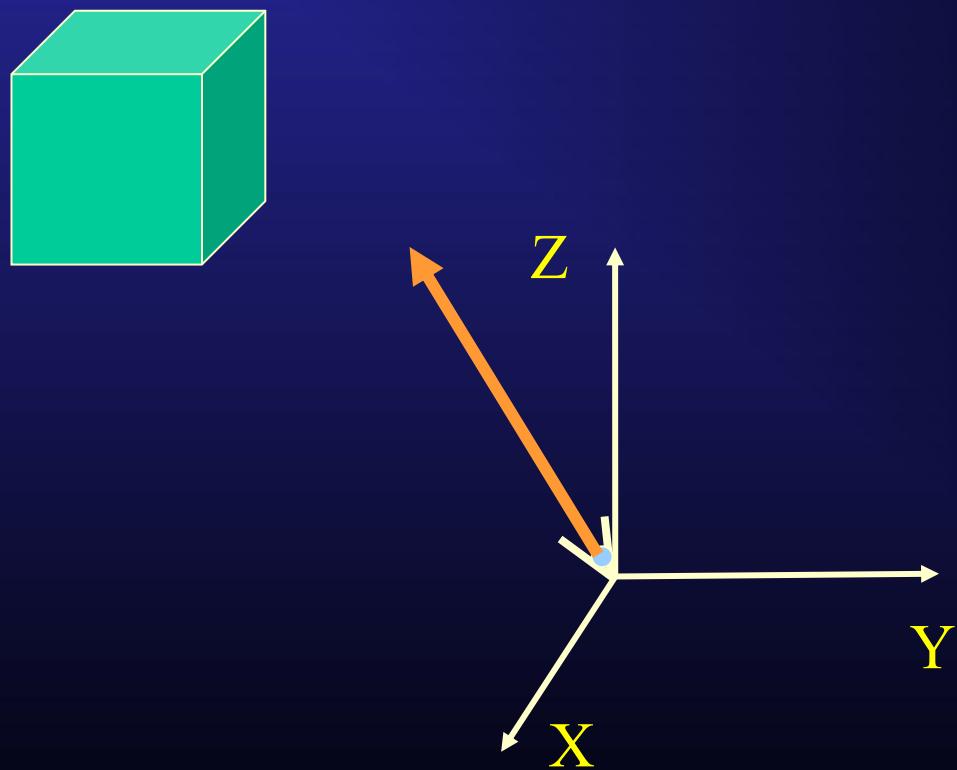


Diagram of World to Eye Transformation

Rotate to align G direction (up) with +Y

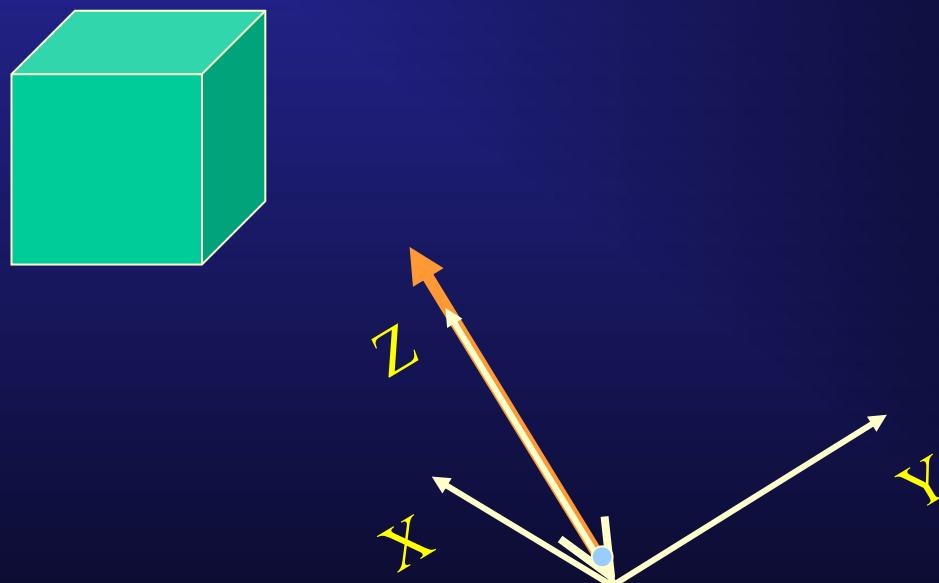


Diagram of World to Eye Transformation
Rotate to align G direction (up) with +Y

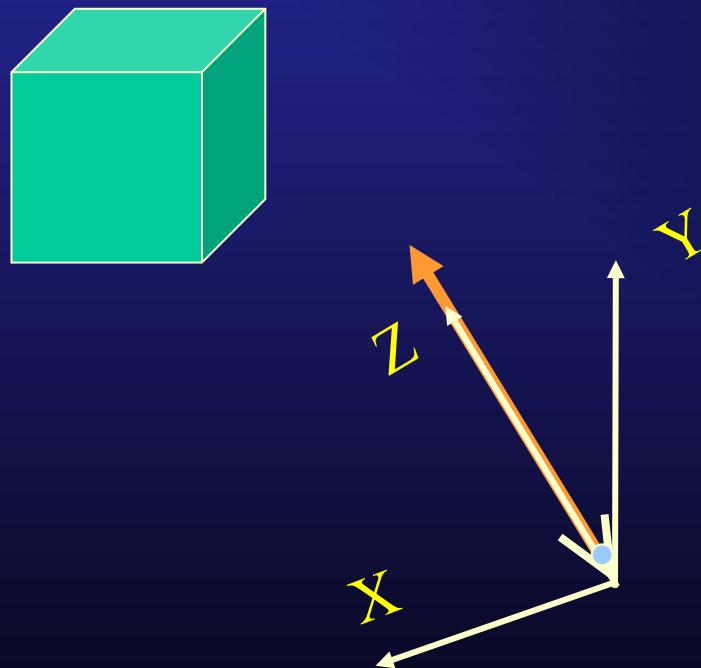
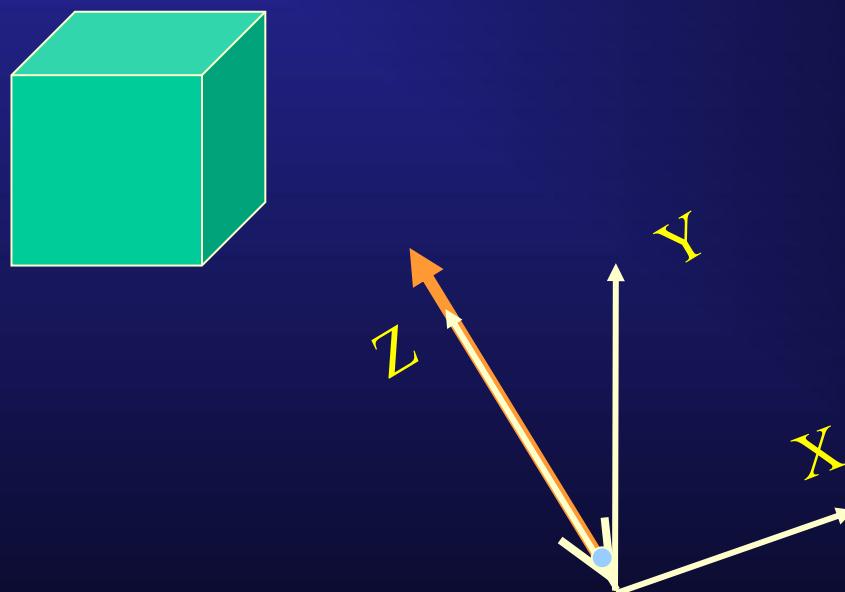


Diagram of World to Eye Transformation

Scale to left-handed coordinate system



Derivation of 3D Perspective Clipping Volume

We'll derive both viewing and clipping transformations.

- Translate center of projection to origin T
- Rotate view plane normal to Z_C axis R_N
- Rotate view up to the $Y_C - Z_C$ plane so that its Y_C is > 0 R_U
- Transform right-handed coordinate system to left-handed, if necessary L
- Shear window center to Z_C axis H
- Scale so back of view volume is in $Z_C = 1$ plane with $-1 \leq X, Y \leq 1$ S

Resulting matrix is $S H L R_U R_N T$

T

Translate center of projection to origin

- REF = view reference point
- PR = center of projection relative to REF
- world coordinates of center of projection = $REF + PR$

$$T = \begin{bmatrix} 1 & 0 & 0 & - (REF_X + PR_X) \\ 0 & 1 & 0 & - (REF_Y + PR_Y) \\ 0 & 0 & 1 & - (REF_Z + PR_Z) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Direct Method for Finding R_N and R_U

Let R be the 3x3 rotation matrix which rotates

- $NORM$ to +Z direction and
- the V axis in the viewplane to the +Y direction
- Thus:

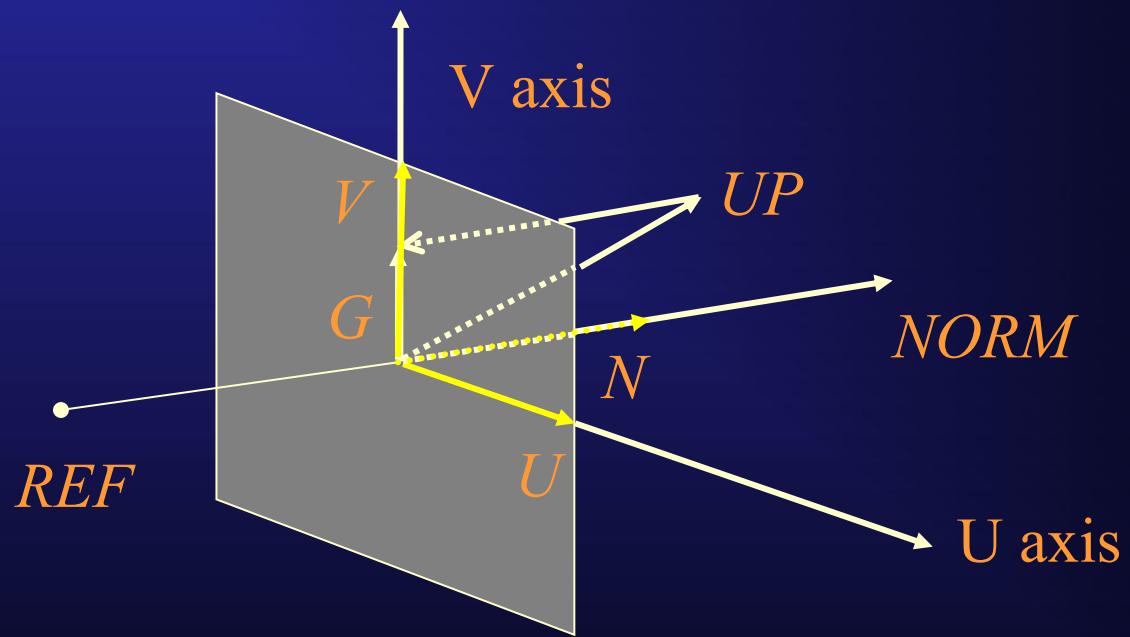
$$R_U \cdot R_N = \left[\begin{array}{ccc|c} & & & 0 \\ & R & & 0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

Define some unit vectors:

- Let N be the unit vector parallel to $NORM$

$$N = \frac{NORM}{\sqrt{NORM_X^2 + NORM_Y^2 + NORM_Z^2}}$$

Reference Diagram for Finding R_N and R_U



Note that U , V , and N are all unit vectors

Computing V

Let V be the unit vector parallel to the V axis on the viewplane.

- Since G is the orthographic projection of the view up vector UP onto the viewplane:

$$G = UP - N(N \cdot UP) \text{ where}$$

$$(N \cdot UP) = N_x UP_x + N_y UP_y + N_z UP_z$$

$$V = \frac{G}{\|G\|}$$

Computing U

Finally, let U be the unit vector parallel to the viewplane U axis:

$$U = N \times V = (N_Y V_Z - N_Z V_Y, N_Z V_X - N_X V_Z, N_Y V_X - N_X V_Y)^T$$

if the world is right-handed

$$U = -N \times V$$

if the world is left-handed

Filling in R

By the definition of R ,

- $RN = (0, 0, 1)^T$
- $RV = (0, 1, 0)^T$

Because U is perpendicular to N and V , the vector RU is perpendicular to vectors RN and RV . That is, U gets rotated to lie along the X axis. Thus:

- $RU = (\alpha, 0, 0)^T$

U being a unit vector means $\alpha = \pm 1$
 $\alpha = +1$ if the world is left-handed
 $\alpha = -1$ if the world is right-handed

The Action of R is Known...

Therefore we have

$$\alpha RU = (1,0,0)^T$$

$$RV = (0,1,0)^T$$

$$RN = (0,0,1)^T$$

hence

$$R \begin{bmatrix} \alpha U_X & V_X & N_X \\ \alpha U_Y & V_Y & N_Y \\ \alpha U_Z & V_Z & N_Z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

R^{-1}

Aha!

Since R is a ROTATION Matrix

As a rotation matrix, R enjoys the special property that

$$R^T = R^{-1} \quad \text{or}$$

$$R = (R^{-1})^T$$

so

$$R = \begin{bmatrix} \alpha U_X & \alpha U_Y & \alpha U_Z \\ V_X & V_Y & V_Z \\ N_X & N_Y & N_Z \end{bmatrix}$$

Changing World Handedness: L

Change right-handed to left-handed;
it suffices to reverse a single axis, so choose the X axis.

Recall

$\alpha = +1$ if the world is left-handed

$\alpha = -1$ if the world is right-handed, so

$$L = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and since $\alpha^2 = 1$, no matter what,

$$LR_U R_N = L \begin{bmatrix} R & & & | & 0 \\ & 0 & & | & 0 \\ & & 0 & | & 0 \\ \hline 0 & 0 & 0 & | & 1 \end{bmatrix} = \begin{bmatrix} U_X & U_Y & U_Z & 0 \\ V_X & V_Y & V_Z & 0 \\ N_x & N_Y & N_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

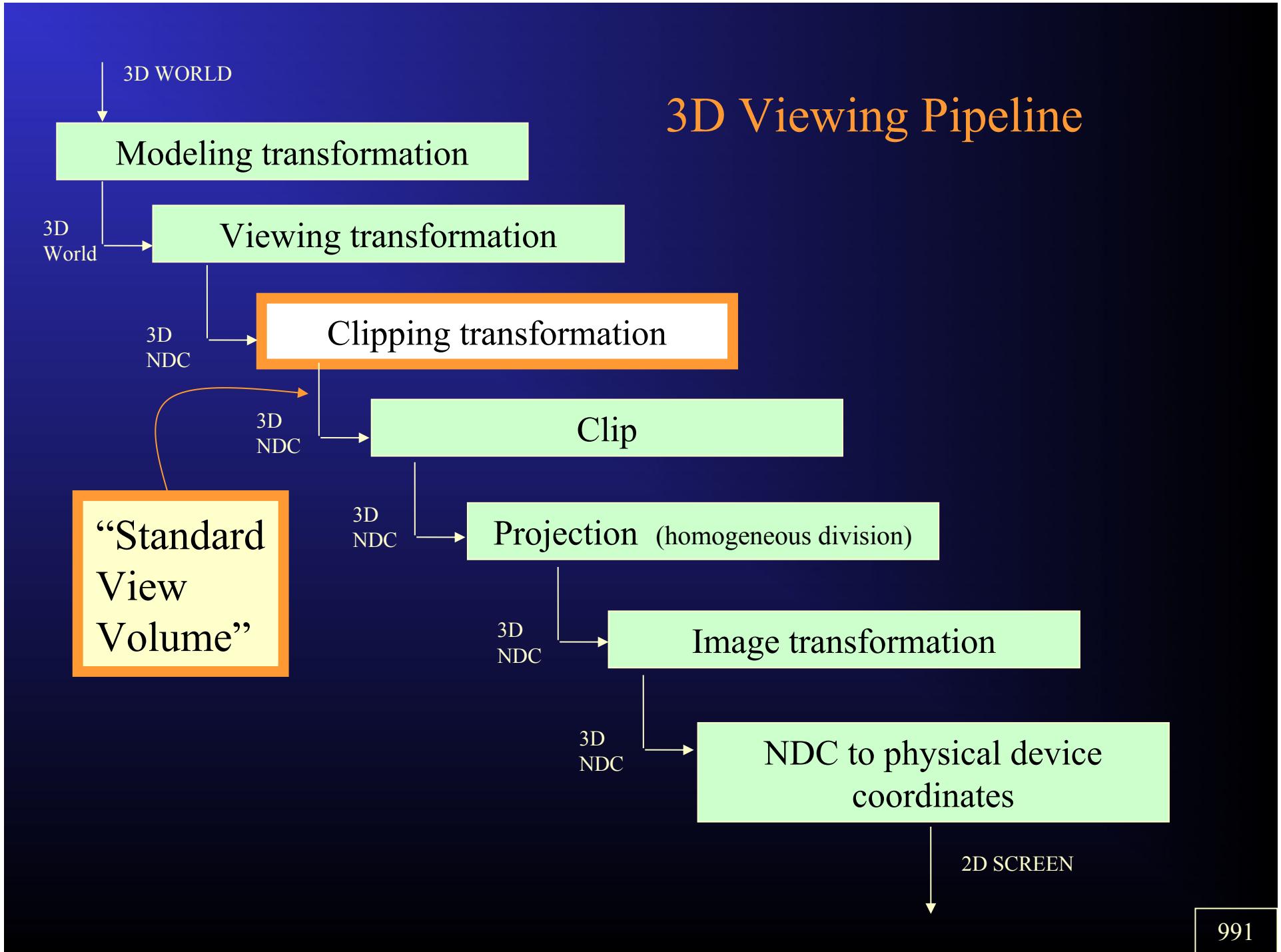
Derivation of 3D Perspective Clipping Volume

So far we have:

- Translate center of projection to origin T
- Rotate view plane normal to Z_C axis R_N
- Rotate view up to the $Y_C - Z_C$ plane so that its Y_C is > 0 R_U
- Transform right-handed coordinate system to left-handed, if necessary L

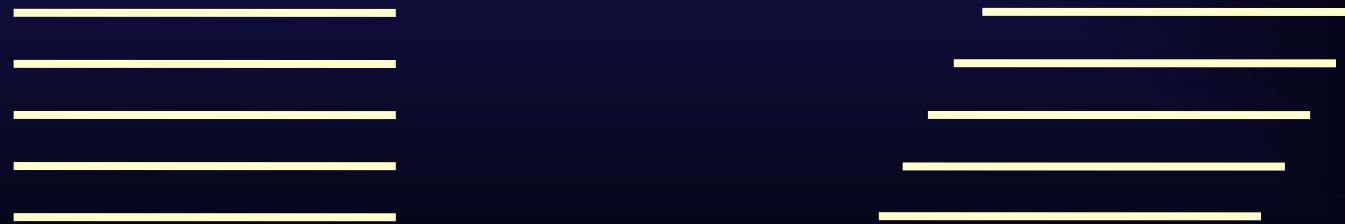
Next we shear and scale to the standard view volume:

- Shear window center to Z_C axis H
- Scale so back of view volume is in $Z_C = 1$ plane with $-1 \leq X, Y \leq 1$ S



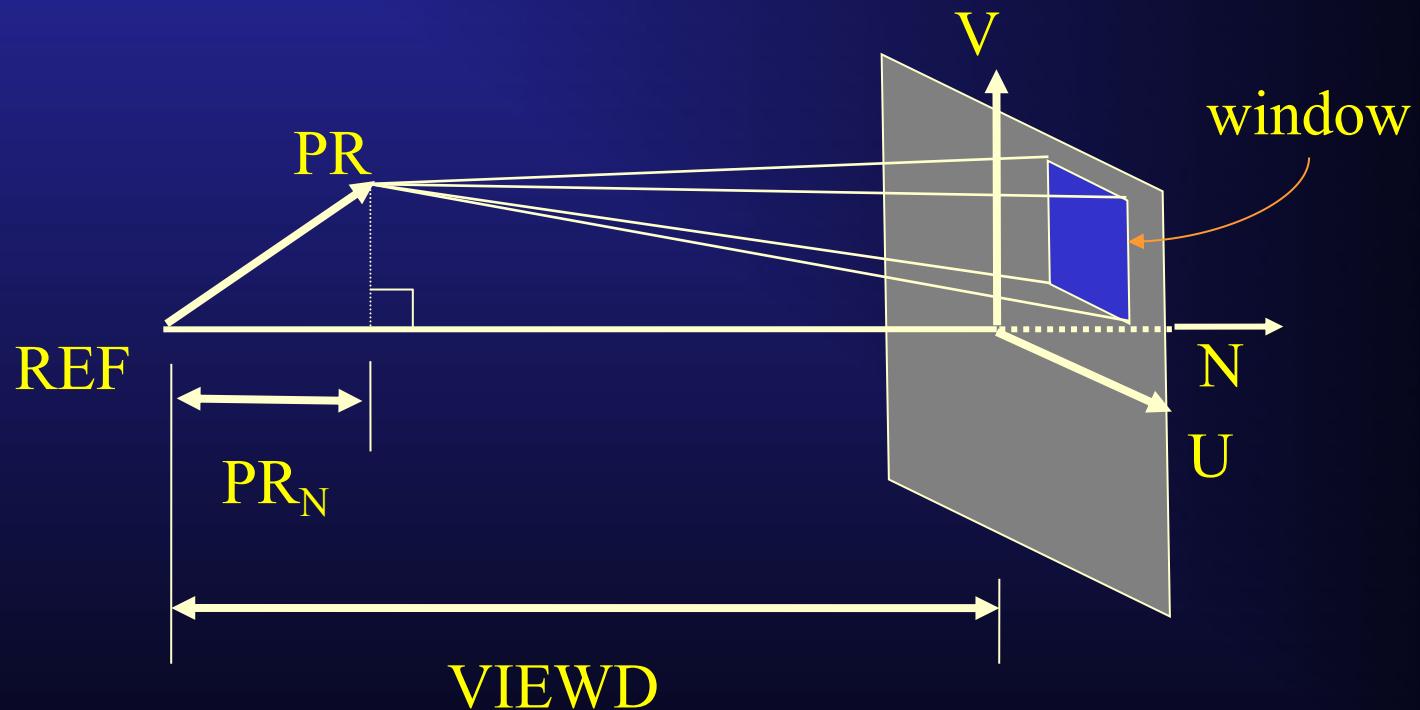
On to the Clipping Transformation

- We now have the first part of the viewing transformation done.
- It remains to do the transformations that put these coordinates into the clipping coordinate system -- the standard perspective pyramidal view volume.
 - We have to **shear** it to get it upright.
 - (A shear is an another affine transformation.)
 - Shearing slides parallel planes like a stack of cards:



Shear Layout

- Recall the overall layout:



Notice that the view pyramid is not a right pyramid.
We must make it so with the shear transformation

Shear the Window Center to the Z_C Axis

- The shearing matrix

$$H = \begin{bmatrix} H_{1,1} & H_{1,2} & H_{1,3} & H_{1,4} \\ H_{2,1} & H_{2,2} & H_{2,3} & H_{2,4} \\ H_{3,1} & H_{3,2} & H_{3,3} & H_{3,4} \\ H_{4,1} & H_{4,2} & H_{4,3} & H_{4,4} \end{bmatrix}$$

must be derived.

- We will do this by finding expressions for all the $H_{i,j}$.
- A shear transformation is sufficiently simple that how a few points are transformed determines the matrix entries.
- This will take 4 steps.

Shear Step 1

- Assume that H leaves the homogeneous coordinate alone.

$$H \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \begin{bmatrix} X' \\ Y' \\ Z' \\ W \end{bmatrix}$$

So

$$H_{4,1}X + H_{4,2}Y + H_{4,3}Z + H_{4,4}W = W$$

$$H = \begin{bmatrix} ? & & & \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear Step 2

- H must map any Z_C plane onto itself. (That's like sliding the cards -- their distance from the table top doesn't change.)

$$H \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} X' \\ Y' \\ Z \\ 1 \end{bmatrix}$$

$$H = \begin{bmatrix} ? & & & \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear Step 3

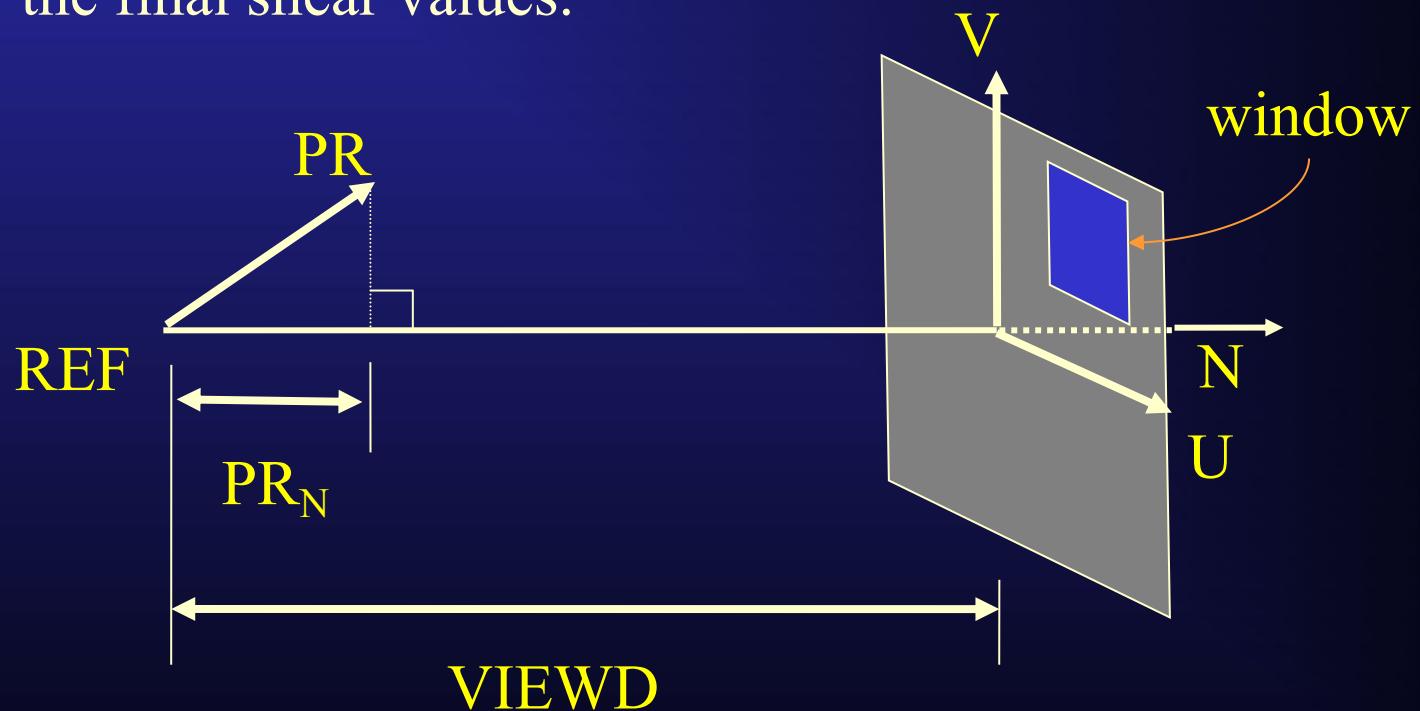
- H must leave the plane $Z_C = 0$ alone. (The table top doesn't move.)

$$H \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix}$$

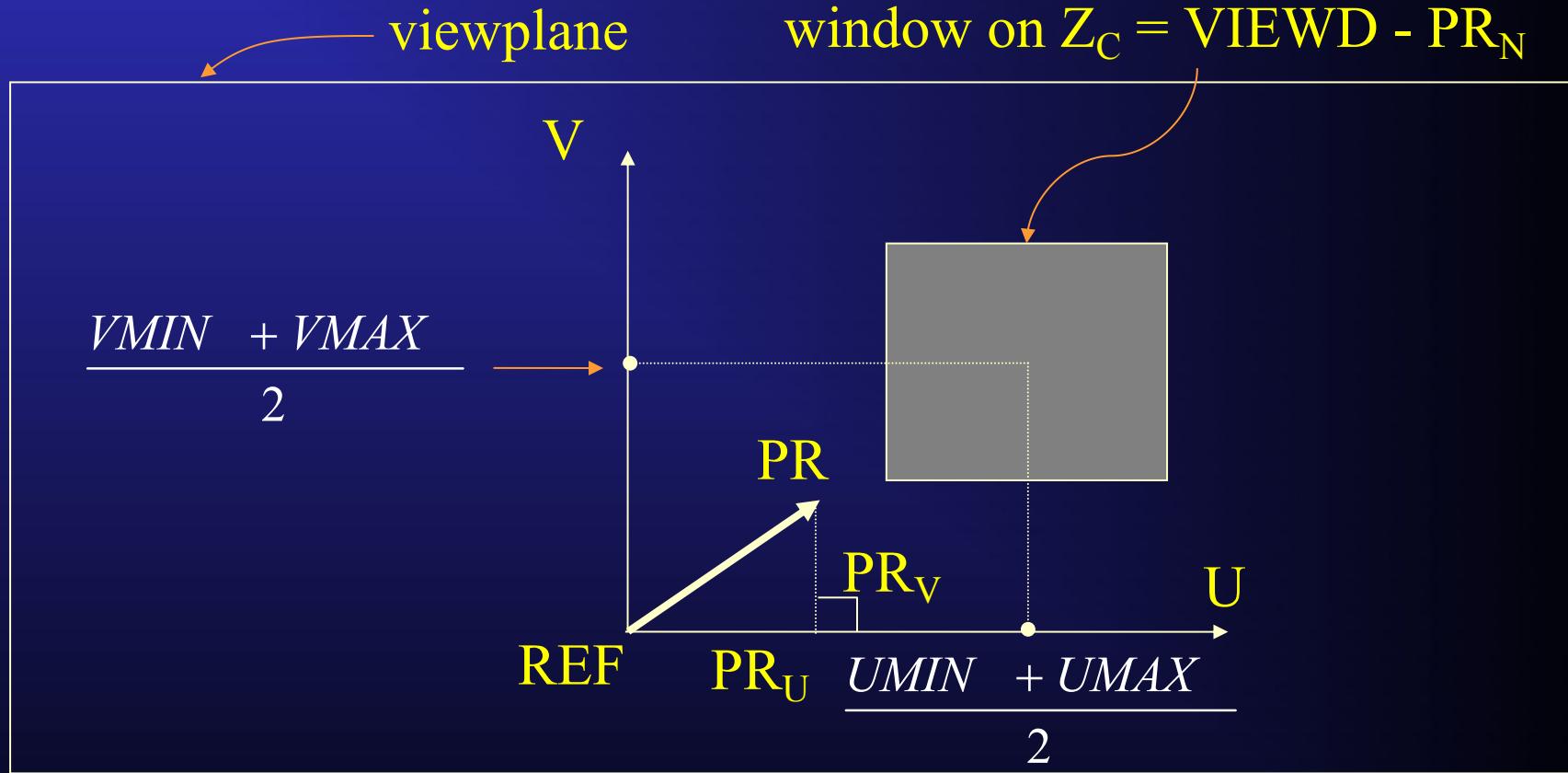
$$H = \begin{bmatrix} 1 & 0 & ? & 0 \\ 0 & 1 & ? & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Recall the Geometry of the Situation

- We'll look at it along the View Plane Normal to compute the final shear values.



Shear Layout: “End on” in World Coordinates

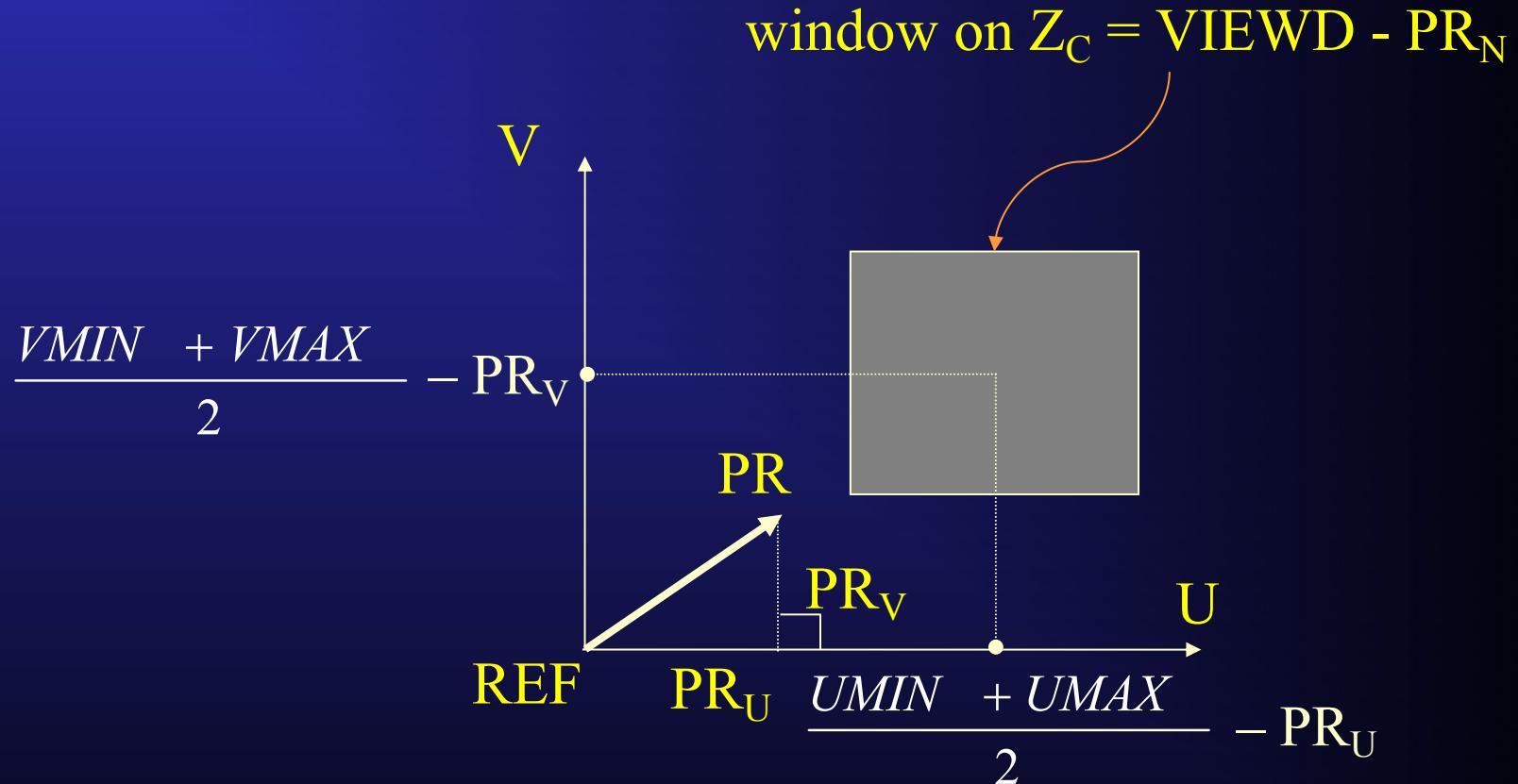


where $PR_U = U \cdot PR = U_X PR_X + U_Y PR_Y + U_Z PR_Z$

and $PR_V = V \cdot PR = V_X PR_X + V_Y PR_Y + V_Z PR_Z$

(calculated in world coordinates)

Shear Layout: “End on” in World Coordinates



$$H \begin{pmatrix} \frac{U_{MIN} + U_{MAX}}{2} - PR_U & \frac{V_{MIN} + V_{MAX}}{2} - PR_V & \text{VIEWD} - PR_N & 1 \end{pmatrix}^T$$

$$= (0 \quad 0 \quad \text{VIEWD} - PR_N \quad 1)^T$$

1000

Shear Step 4

- Solving for $H_{1,3}$:

$$\left[\frac{1}{2}(U_{MIN} + U_{MAX}) - PR_U \right] + H_{1,3} [VIEWD - PR_N] = 0$$

thus,
$$H_{1,3} = \frac{PR_U - \frac{1}{2}(U_{MIN} + U_{MAX})}{VIEWD - PR_N}$$

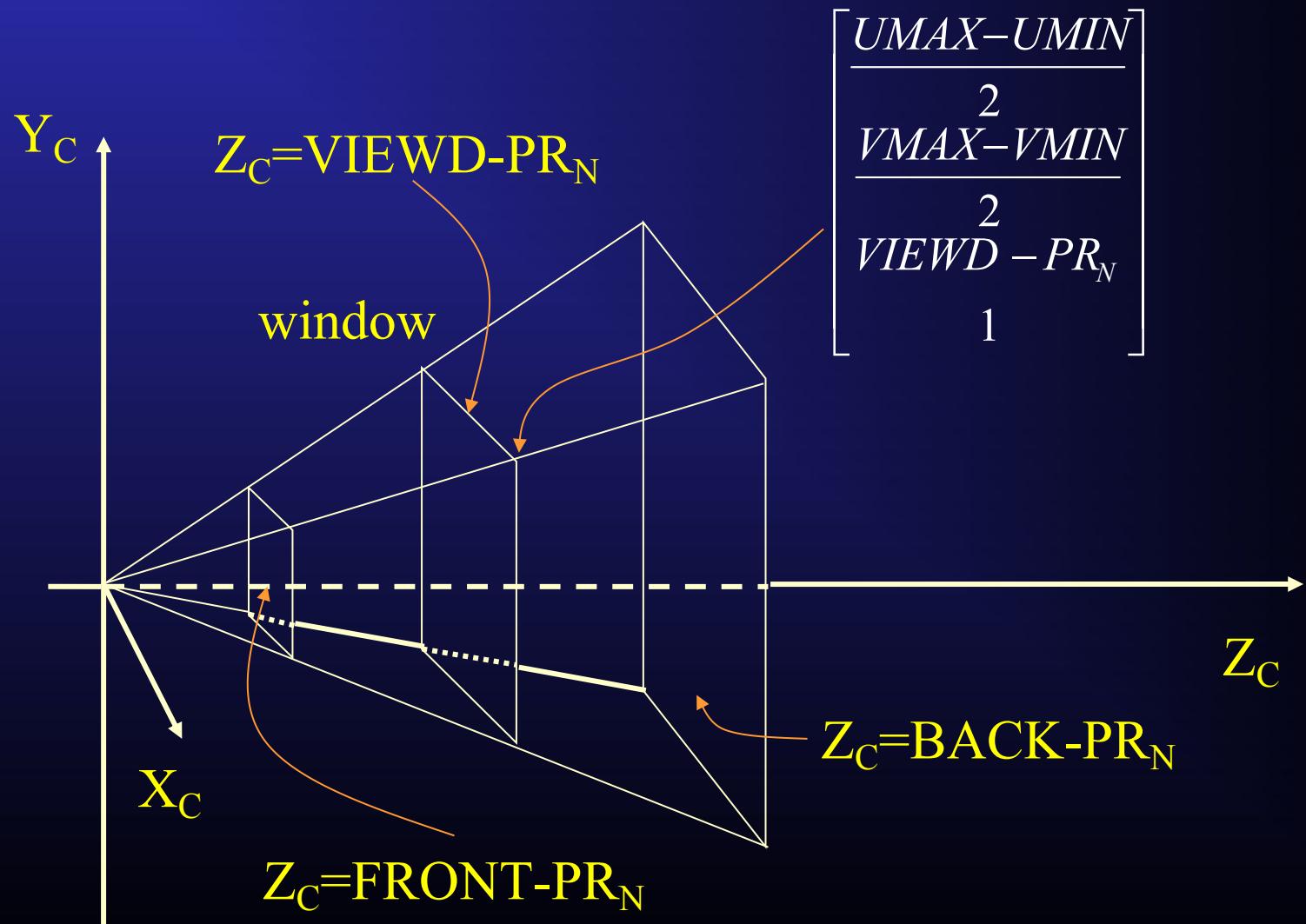
and similarly,
$$H_{2,3} = \frac{PR_V - \frac{1}{2}(V_{MIN} + V_{MAX})}{VIEWD - PR_N}$$

Shear Step 4 (continued)

- Finally:

$$H = \begin{bmatrix} 1 & 0 & \frac{PR_U - \frac{1}{2}(UMIN + UMAX)}{VIEWD - PR_N} & 0 \\ 0 & 1 & \frac{PR_V - \frac{1}{2}(VMIN + VMAX)}{VIEWD - PR_N} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling to Standard View Volume



Scale Factors

$$\text{Let } D = \frac{\text{VIEWD} - \text{PR}_N}{\text{BACK} - \text{PR}_N}$$

then scales in X_C Y_C Z_C are

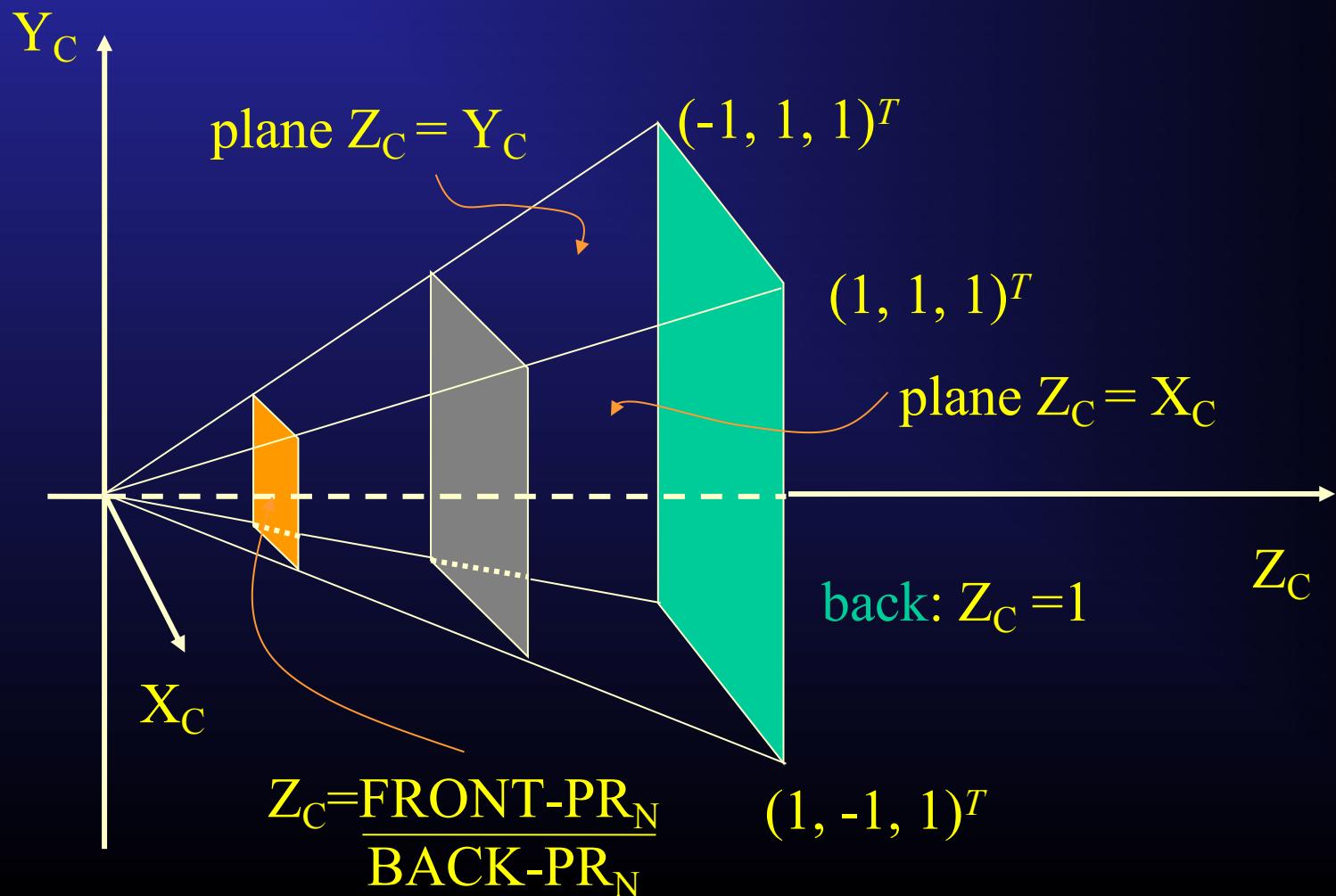
$$X_C : \frac{2D}{UMAX - UMIN}$$

$$Y_C : \frac{2D}{VMAX - VMIN}$$

$$Z_C : \frac{1}{BACK - PR_N}$$

The Standard View Volume for Perspective Case

- Standard clipping coordinate system x_c, y_c, z_c .



So Now we Know S

$$S = \begin{bmatrix} \frac{2(VIEWD - PR_N)}{(U_{MAX} - U_{MIN})(BACK - PR_N)} & 0 & 0 & 0 \\ 0 & \frac{2(VIEWD - PR_N)}{(V_{MAX} - V_{MIN})(BACK - PR_N)} & 0 & 0 \\ 0 & 0 & \frac{1}{BACK - PR_N} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Summary of Transformation to Clipping View Volume for Perspective Case

$$S \ H \ L \ R_U \ R_N \ T = (S \ H)(L \ R_U \ R_N \ T) =$$

$$\begin{bmatrix}
\frac{2}{UMAX - UMIN} * \frac{VIEWD - PR_N}{BACK - PR_N} & 0 & \frac{2PR_U - (UMIN + UMAX)}{(UMAX - UMIN)(BACK - PR_N)} & 0 \\
0 & \frac{2}{VMAX - VMIN} * \frac{VIEWD - PR_N}{BACK - PR_N} & \frac{2PR_V - (VMIN + VMAX)}{(VMAX - VMIN)(BACK - PR_N)} & 0 \\
0 & 0 & \frac{1}{BACK - PR_N} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \\
* \begin{bmatrix}
U_X & U_Y & U_Z & -(REF + PR) \cdot U \\
V_X & V_Y & V_Z & -(REF + PR) \cdot V \\
N_X & N_Y & N_Z & -(REF + PR) \cdot N \\
0 & 0 & 0 & 1
\end{bmatrix}$$

“Match Move”: Perspective Reconstruction from Multiple Camera Views

- The perspective projection allows an inverse process called **perspective reconstruction**.
- If we have the 2D screen coordinates of the same 3D point in two successive image frames (times), we can reconstruct the perspective projection matrix (up to a translation).
- (That's because a ray from the camera projects into a point.)
- This process is used in the movies to **match move** real images to arbitrary camera motions: the camera is difficult to track in space directly, so the camera motions are computed and can then be used to control a synthetic CG camera following the same track in space to add views of virtual 3D objects into the scene.

Derivation of 3D Parallel Clipping Volume

We'll derive both viewing and clipping transformations.

- Translate view reference point to origin T_1'
- Rotate view plane normal to Z_C axis R_N
- Rotate view up to the $Y_C - Z_C$ plane R_U
- Transform right-handed coordinate system
to left-handed, if necessary L
- Translate lower left corner of window to origin T_2'
- Shear to align projection direction with
 Z_C direction H'
- Scale and shift so view volume is the unit cube S'

Resulting matrix is $S' H' T_2' L R_U R_N T_1'$

T_1' and T_2'

- Translate view reference point to origin:

$$T_1' = \begin{bmatrix} 1 & 0 & 0 & -REF_X \\ 0 & 1 & 0 & -REF_Y \\ 0 & 0 & 1 & -REF_Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation is exactly the same as for perspective case.
- Later we need to translate lower left corner of window to origin:

$$T_2' = \begin{bmatrix} 1 & 0 & 0 & -UMIN \\ 0 & 1 & 0 & -VMIN \\ 0 & 0 & 1 & -VIEWD \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear for Parallel Projection

- Let J be the projection direction as transformed by $T_2' L R_U R_N T_1'$
 J can be written $(J_U \ J_V \ J_N)^T$ where

$$J_U = U \cdot (DX_PROJ \ DY_PROJ \ DZ_PROJ)^T$$

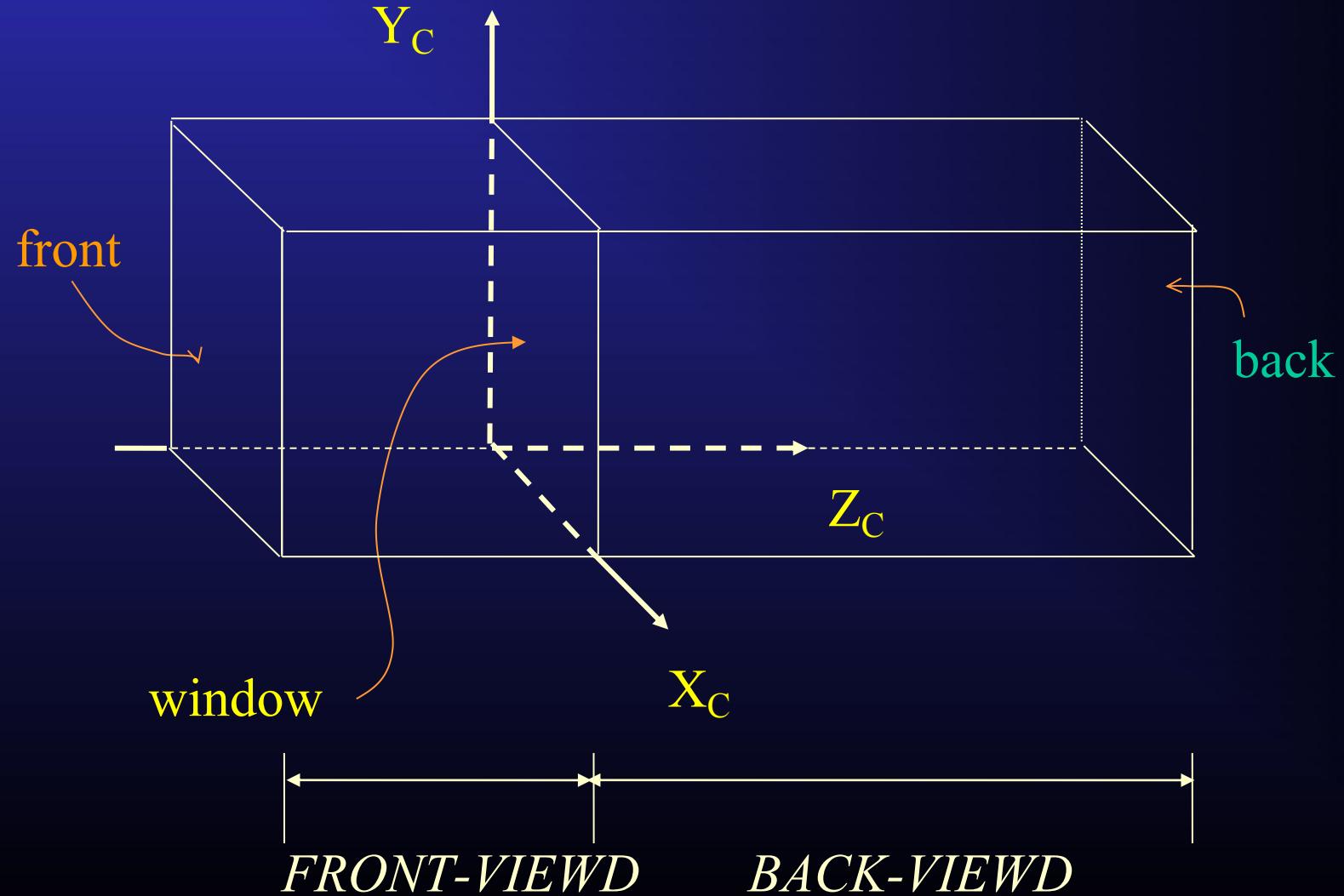
$$J_V = V \cdot (DX_PROJ \ DY_PROJ \ DZ_PROJ)^T$$

$$J_N = N \cdot (DX_PROJ \ DY_PROJ \ DZ_PROJ)^T$$

- The point J before shearing must be mapped to the point $(0 \ 0 \ J_N)^T$ after shearing:

$$H' = \begin{bmatrix} 1 & 0 & -\frac{J_U}{J_N} & 0 \\ 0 & 1 & -\frac{J_V}{J_N} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling to Standard View Volume



Scale Factors

Scales in X_c Y_c Z_c are

$$X_c : \frac{1}{UMAX - UMIN}$$

$$Y_c : \frac{1}{VMAX - VMIN}$$

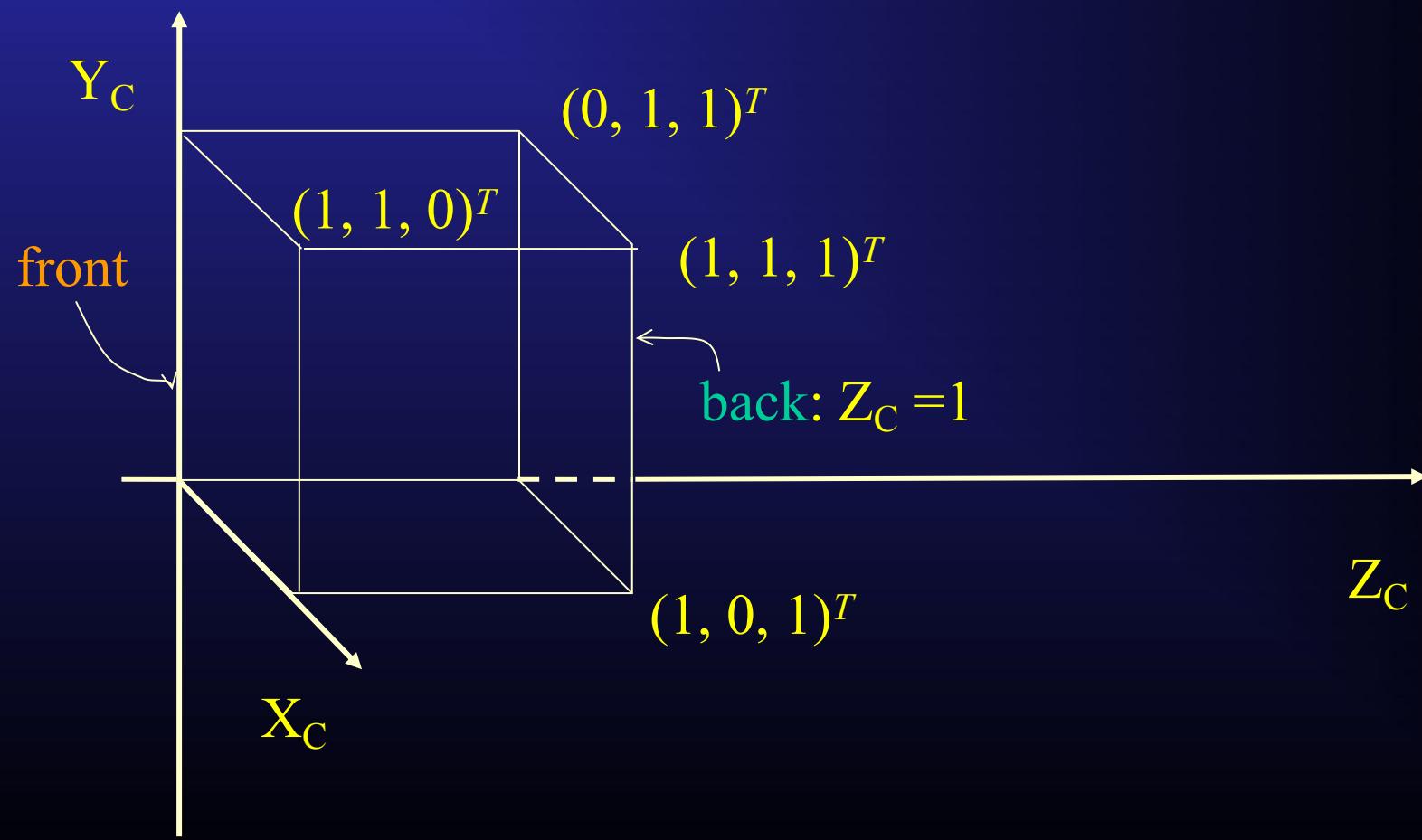
first shift Z_c by $VIEWD - FRONT$

$$\text{then scale } Z_c : \frac{1}{BACK - FRONT}$$

This yields the unit cube as the view volume.

The Standard View Volume for Parallel Case

The Unit Cube $[0, 1]^3$



So Now we Know S'

$S' =$

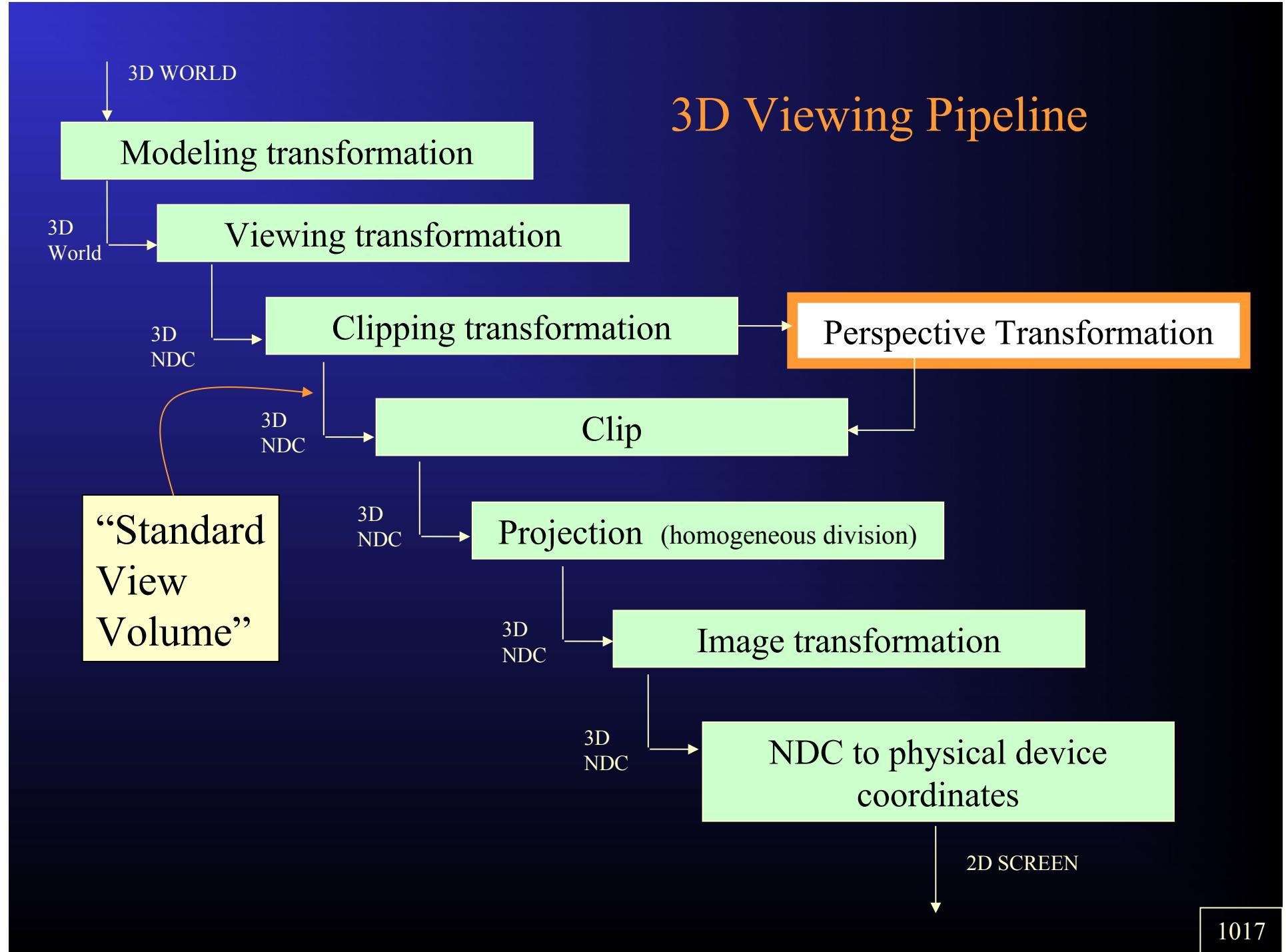
$$\begin{bmatrix} \frac{1}{UMAX - UMIN} & 0 & 0 & 0 \\ 0 & \frac{1}{VMAX - VMIN} & 0 & 0 \\ 0 & 0 & \frac{1}{BACK - FRONT} & \frac{VIEWD - FRONT}{BACK - FRONT} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Summary of Transformation to Clipping View Volume for Parallel Case

$$S' H' T_2' L R_U R_N T_I' = (S' H') (T_2' L R_U R_N T_I')$$

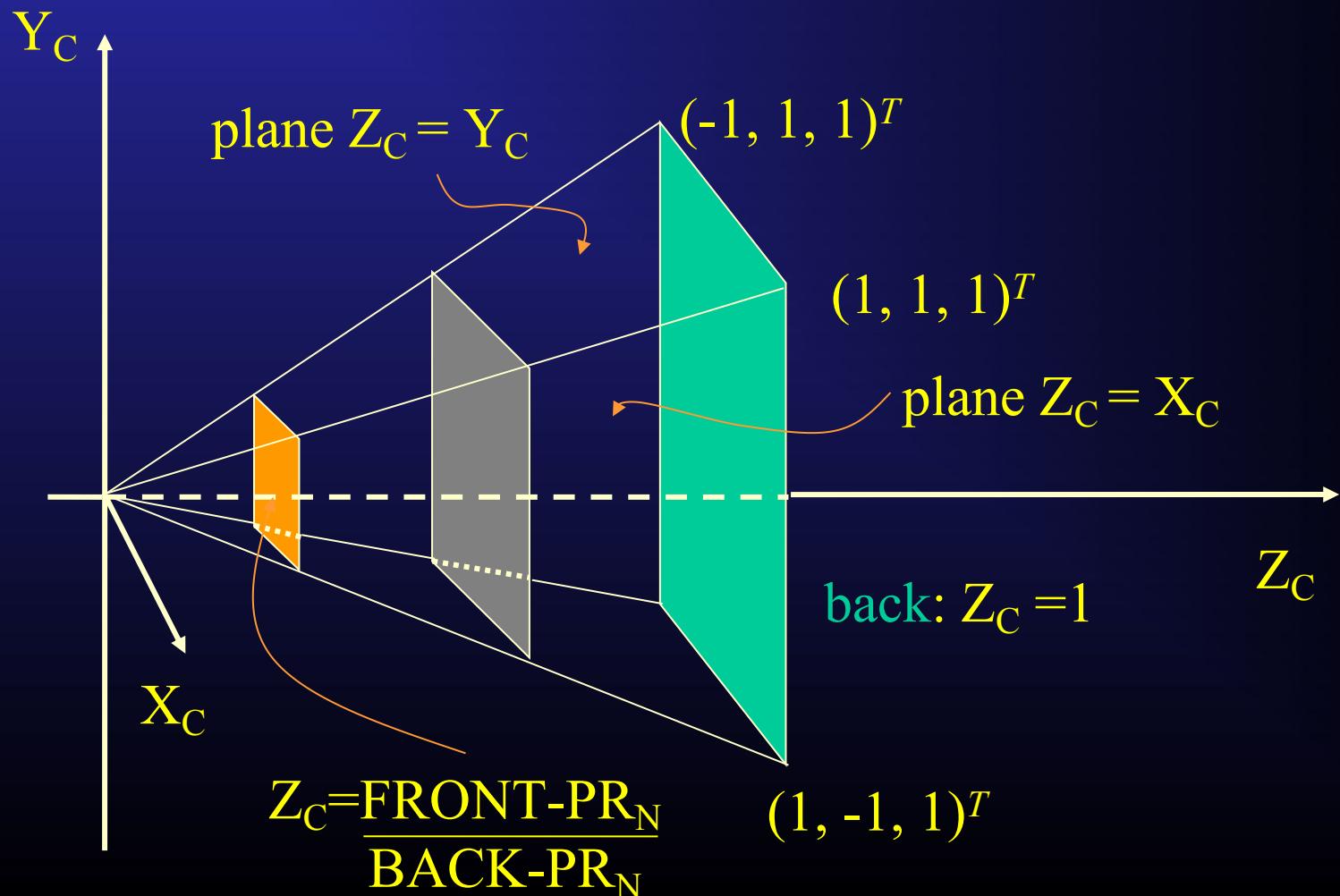
$$\begin{bmatrix} \frac{1}{UMAX - UMIN} & 0 & \frac{-J_U}{J_N(UMAX - UMIN)} & 0 \\ 0 & \frac{1}{VMAX - VMIN} & \frac{-J_V}{J_N(VMAX - VMIN)} & 0 \\ 0 & 0 & \frac{1}{BACK - FRONT} & \frac{VIEWD - FRONT}{BACK - FRONT} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$* \begin{bmatrix} U_X & U_Y & U_Z & -REF \cdot U - UMIN \\ V_X & V_Y & V_Z & -REF \cdot V - VMIN \\ N_X & N_Y & N_Z & -REF \cdot N - VIEWD \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Recall the Standard View Volume:

- Standard clipping coordinate system x_c, y_c, z_c .



The Perspective Transformation (for the Perspective Case only)

- Now that we have a normalized perspective view volume we apply one more matrix to it in order to permit simple depth comparisons -- which will be needed later.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1 - Z_{front}} & \frac{-Z_{front}}{1 - Z_{front}} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- Where Z_{front} is the value of the front clipping z coordinate after viewing transformation =

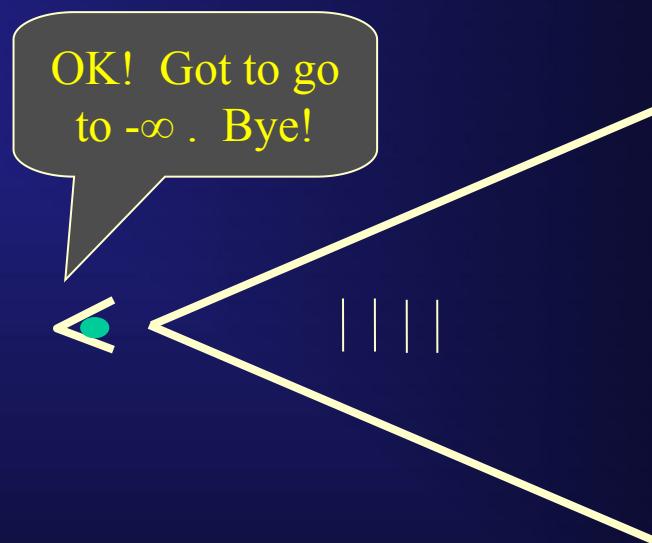
$$(\text{Front} - PR_N) / (\text{Back} - PR_N)$$

What does M do?

- Notice that M does not affect the x or y coordinates.
- M sets the homogeneous coordinate $w \leftarrow z$.
- z is changed to lie in the range $[0, 1]$.
- Check: if $z = Z_{front}$ then new $z \leftarrow 0$;
if $z = 1$ then new $z \leftarrow 1$.

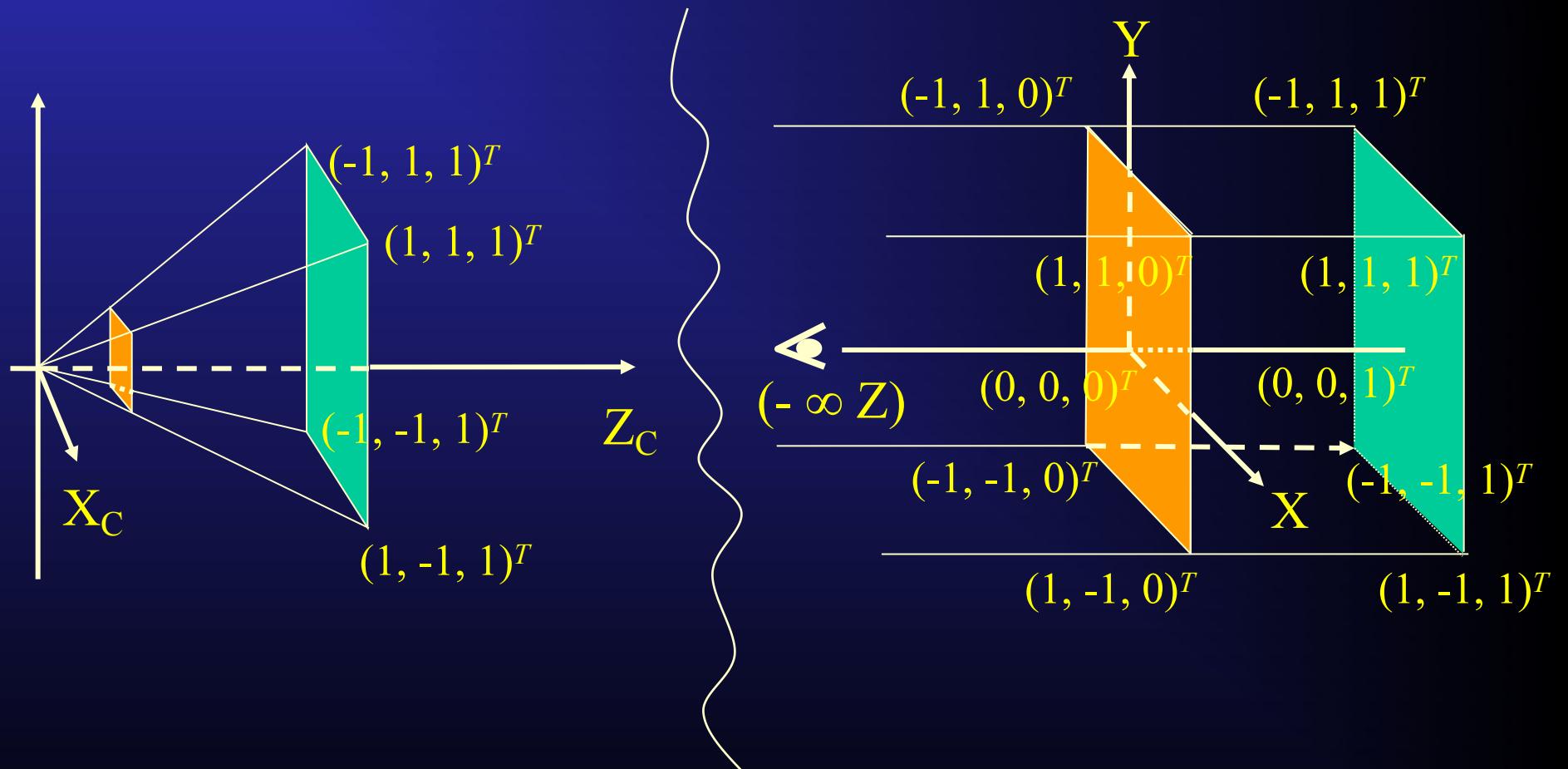
M Creates the Perspective Foreshortening Effect

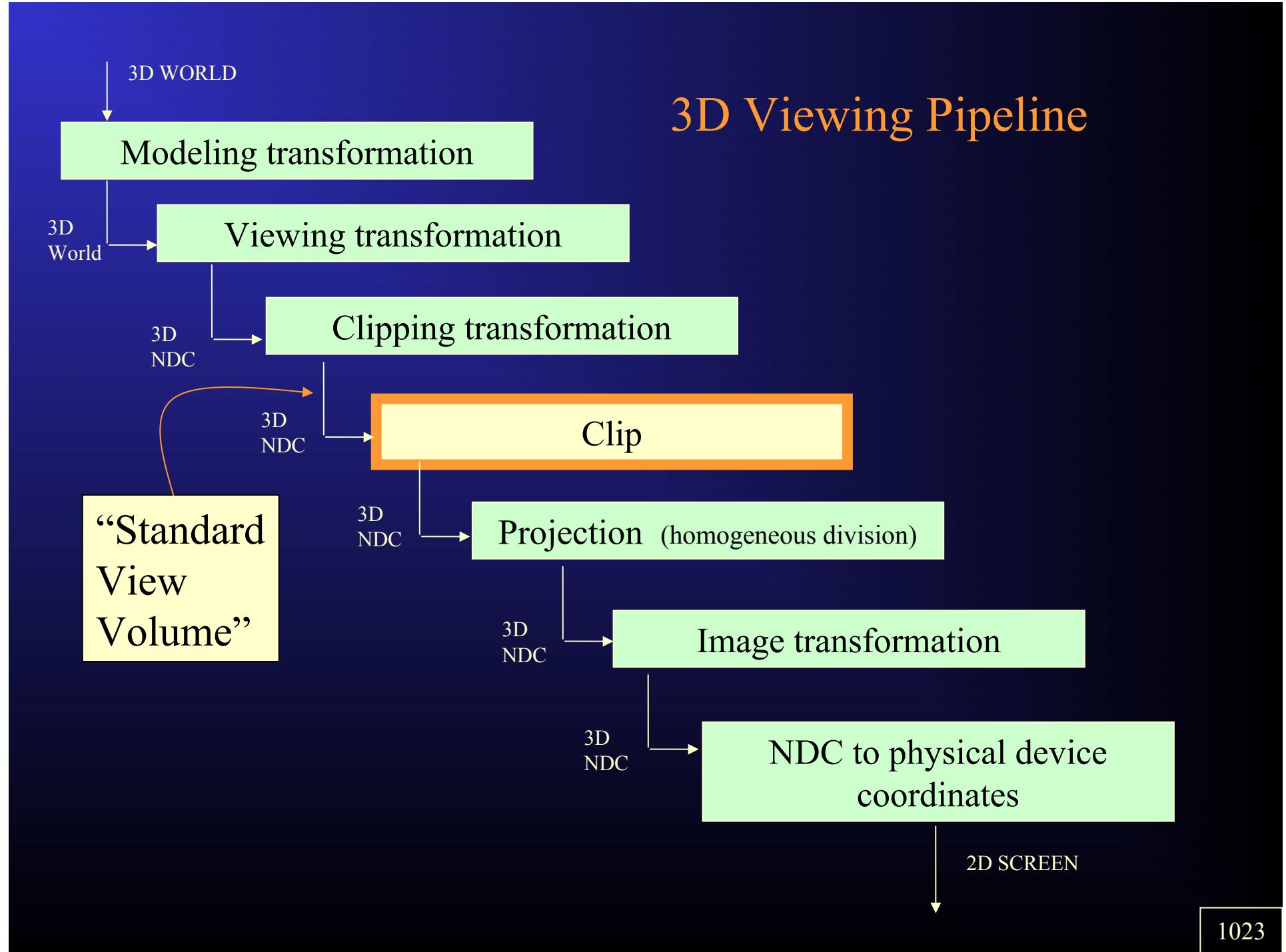
- Thus M makes the projectors parallel:



- This allows simple depth comparisons later:
- Preserves relative depth, linearity, and planarity.
- Still permits clipping -- just use W coordinate.

M Changes Standard Pyramid to This...





Clipping

- Points
- Lines
- Polygons

View Volume Clipping Limits

	Parallel	Perspective
Above	$y > 1$	$y > w$
Below	$y < 0$	$y < -w$
Right	$x > 1$	$x > w$
Left	$x < 0$	$x < -w$
Behind (yon)	$z > 1$	$z > 1$
In Front (hither)	$z < 0$	$z < 0$

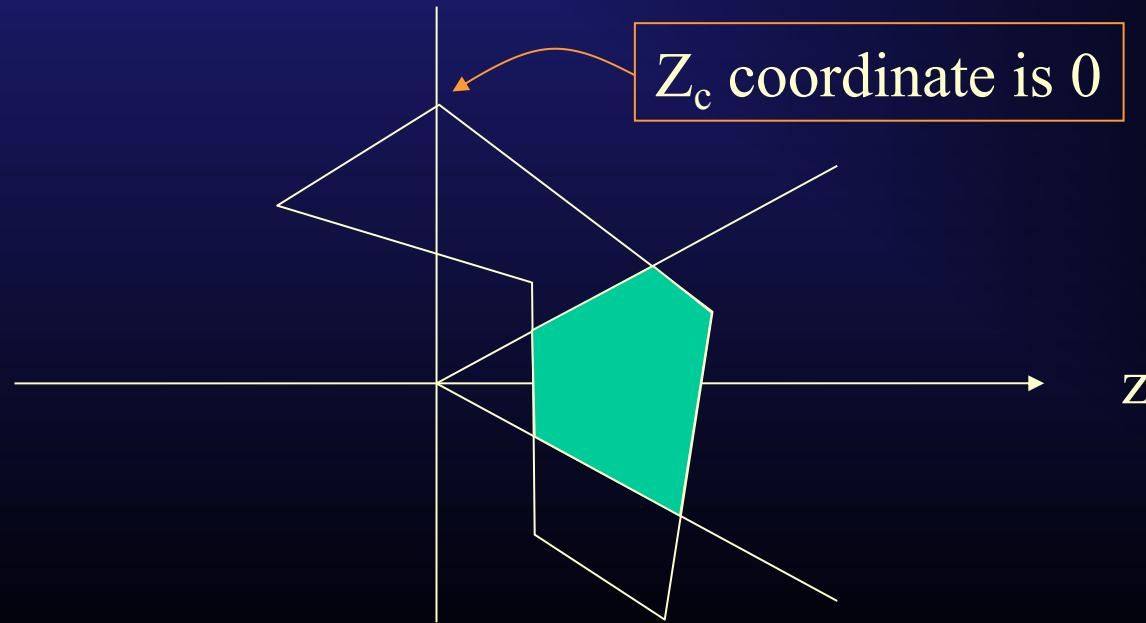
A point (x, y, z) is in the view volume if and only if it lies inside these 6 planes.

Clipping Lines

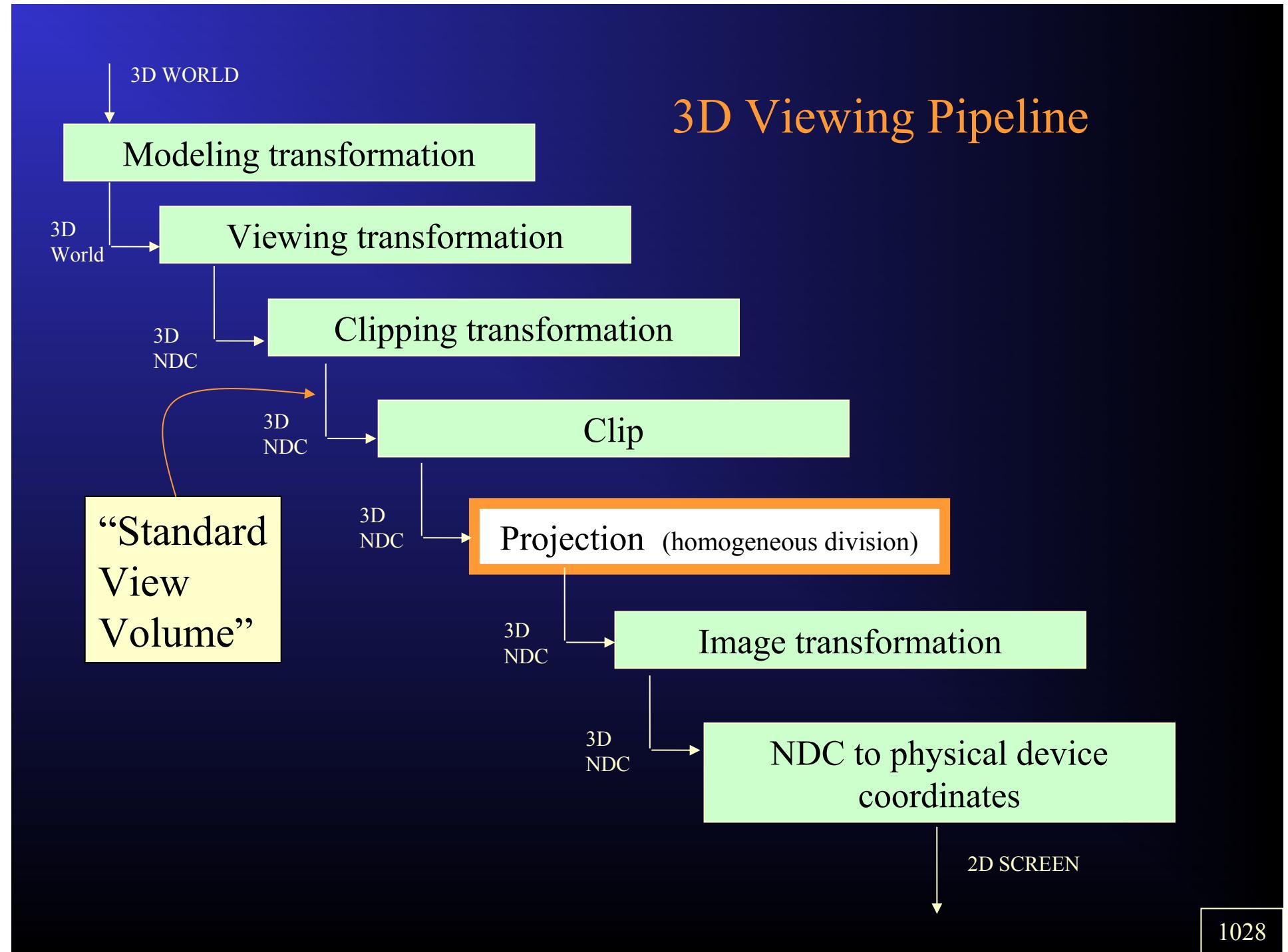
- Extend 2D case to 3D planes.
- Now have 6-bit code rather than 4-bit (above, below, left, right, in-front, behind).
- Only additional work is to find intersection of a line with a clipping plane.

Clipping 3D Polygons

- Important to clip polygons for visible surface synthesis.
- Preserve polygon properties (for rasterization).
- Avoid viewing pipeline degeneracies (e.g., divide by zero)
Sutherland and Hodgman, CACM June 1974.
- Already looked at 2D case; 3D is simple extension, but have 6 clipping planes (adding front and back) not just 4.



3D Viewing Pipeline



Normalize Homogeneous Coordinates (Perspective Only)

$$x' = \frac{x}{w}$$

$$y' = \frac{y}{w}$$

$$z' = \frac{z}{w}$$

provided $w \neq 0$

Returns x' and y' in range $[-1, 1]$
 z' in range $[0, 1]$

3D Window to 3D Viewport in (3D NDC)

- Parallel:
- Standard view volume is unit cube, so nothing to do!

$$X = x_c$$

$$Y = y_c$$

$$Z = z_c$$

- Perspective:
- Must translate view volume by +1 and scale it by 0.5:

$$X = (x_c + 1) / 2$$

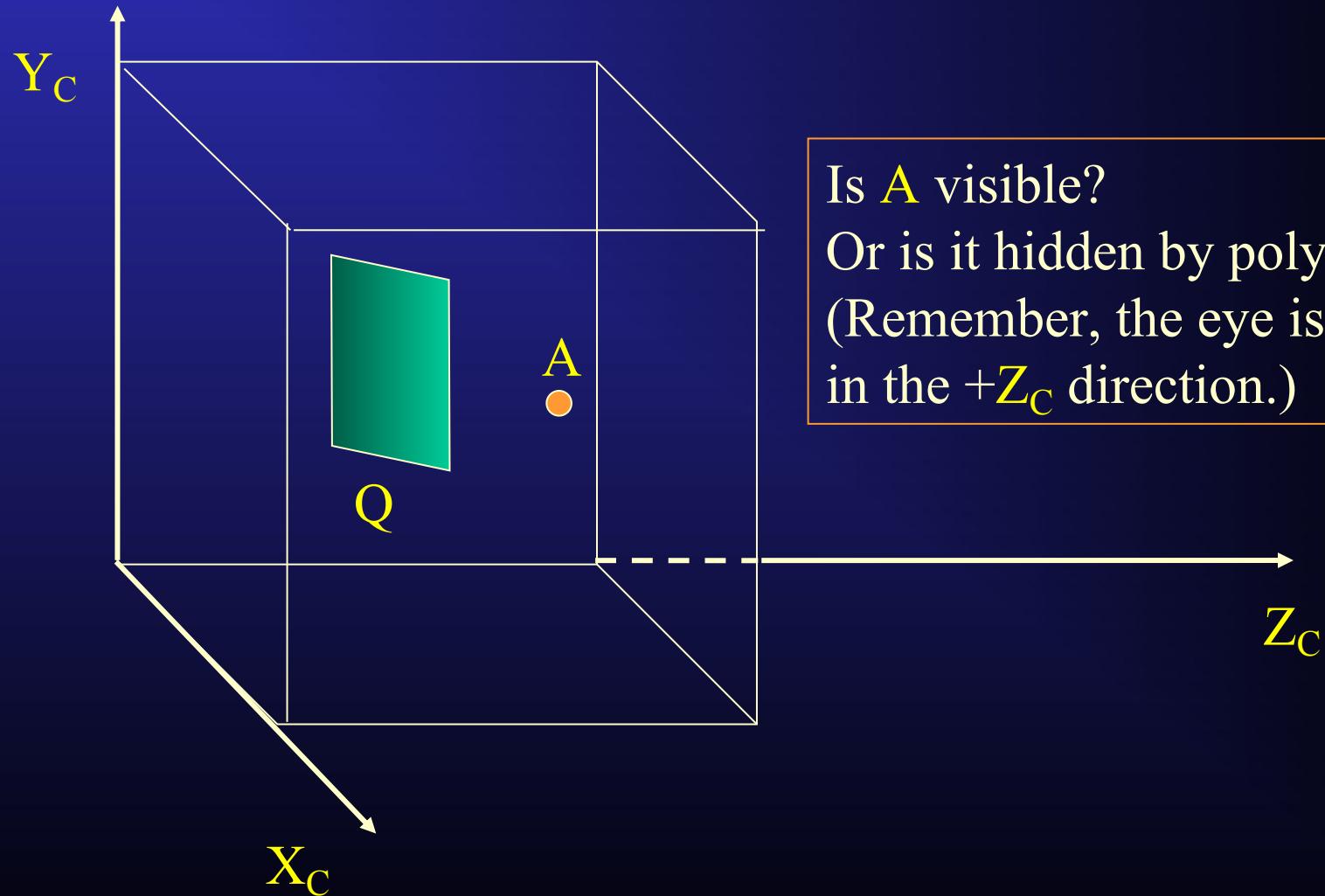
$$Y = (y_c + 1) / 2$$

$$Z = z_c$$

Depth Comparisons

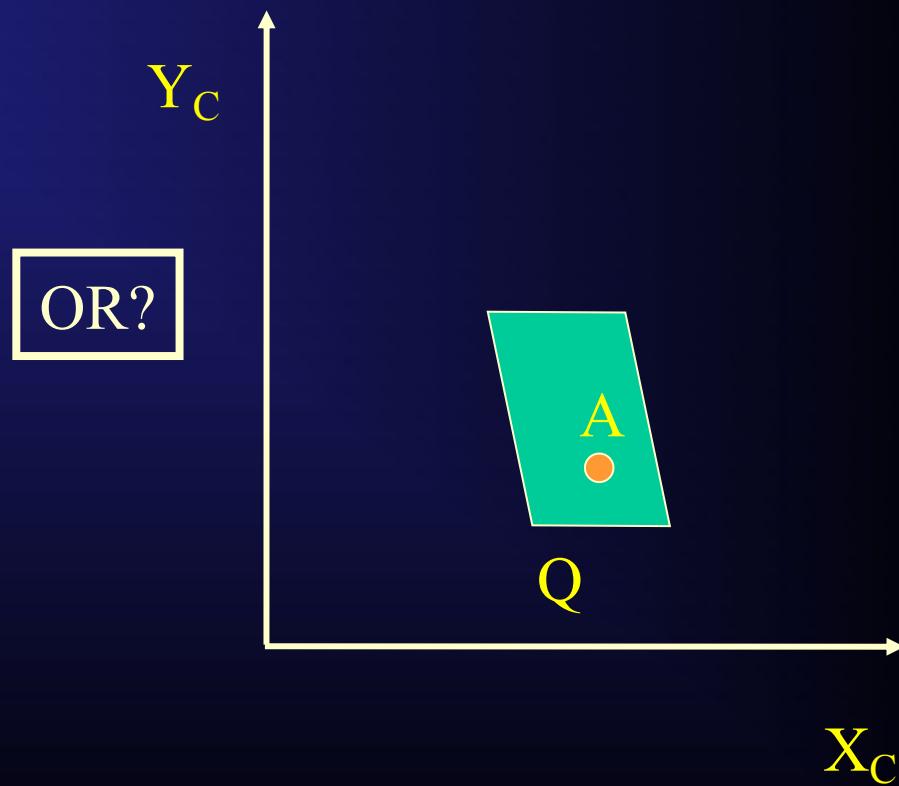
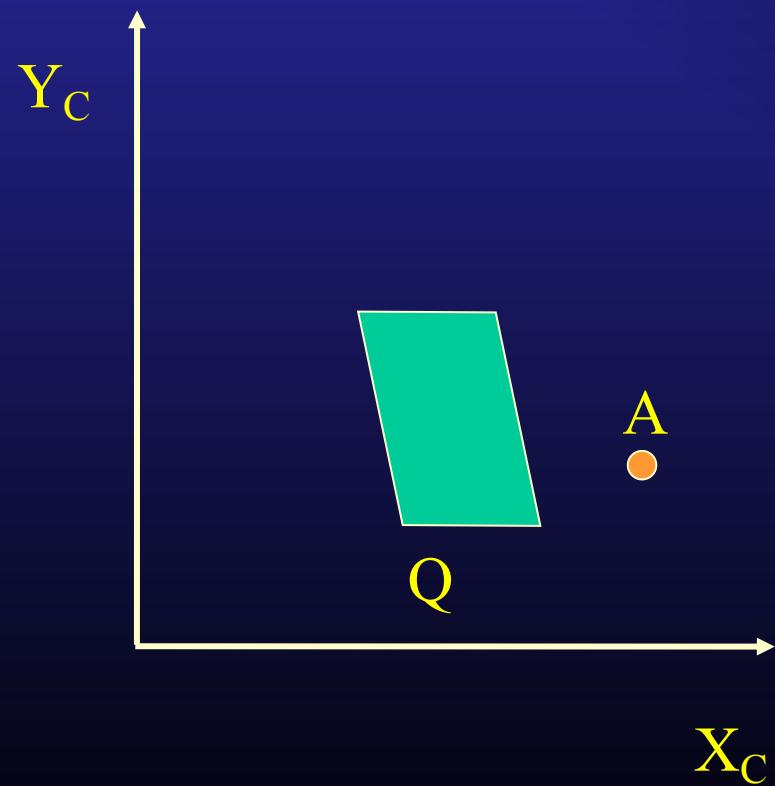
- These transformations have preserved depth relationships in 3D.
- Because of this we can remove hidden parts of the scene (visible surface algorithms).
- Let's see in general how these depth relationships determine visibility: Can a particular polygon obscure (hide) a given point?
- From the preceding derivations, it is sufficient to work in the unit cube.
- Moreover, the computation is simple because we have already taken the possibility of perspective into account.

Example for Depth Comparisons -- Polygon and Point



Is A visible?
Or is it hidden by polygon Q ?
(Remember, the eye is looking
in the $+Z_C$ direction.)

Looking along the $+Z_c$ direction,
A is either inside the projection of Q or not



OR?

Depth Comparison Algorithm

- A is hidden by polygon Q if
 - A is within polygon image of Q when both are projected onto the $Z_C = 0$ plane. (This is easy: just ignore the Z_C coordinates of A and the vertices of Q.) [Point-in-polygon algorithm]
 - AND
 - $Z_C(A) > Z_Q(A)$ -- where $Z_Q(A)$ is the corresponding depth of plane of Q at $X_C(A), Y_C(A)$:

for plane Q given by: $ax + by + cz + d = 0$

then in plane Q, plugging in $X_C(A), Y_C(A)$ and solving for z:

$$Z_Q(A) = - (1/c) (a X_C(A) + b Y_C(A) + d)$$

3D Viewing Pipeline

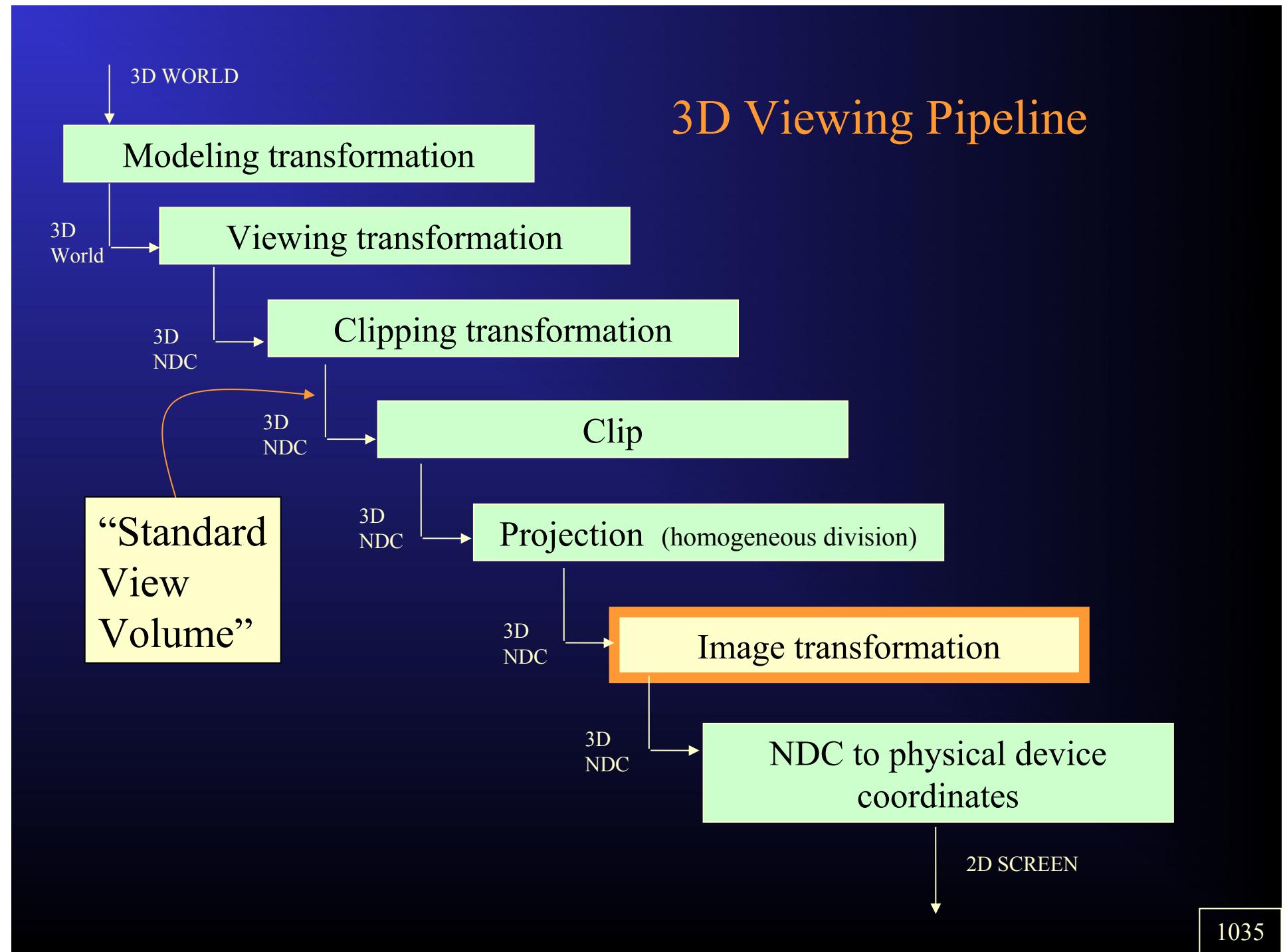
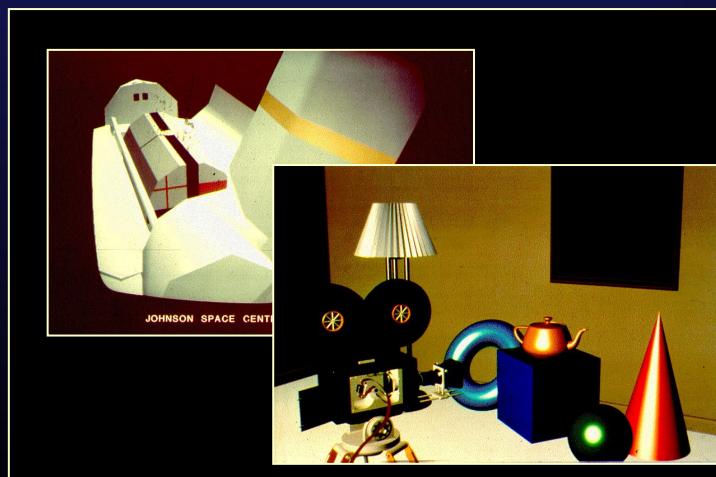
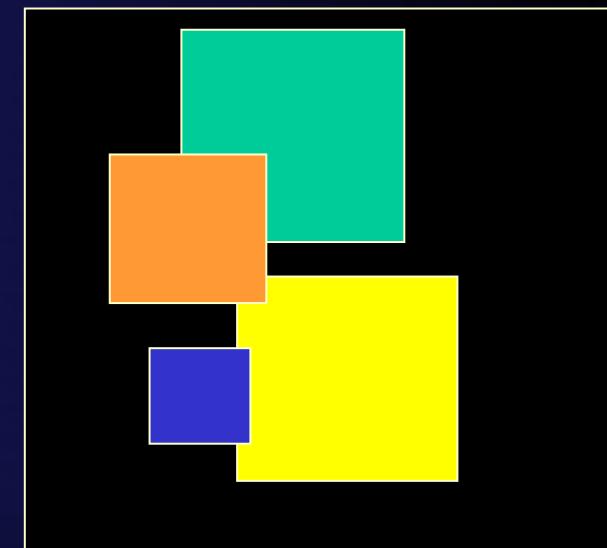
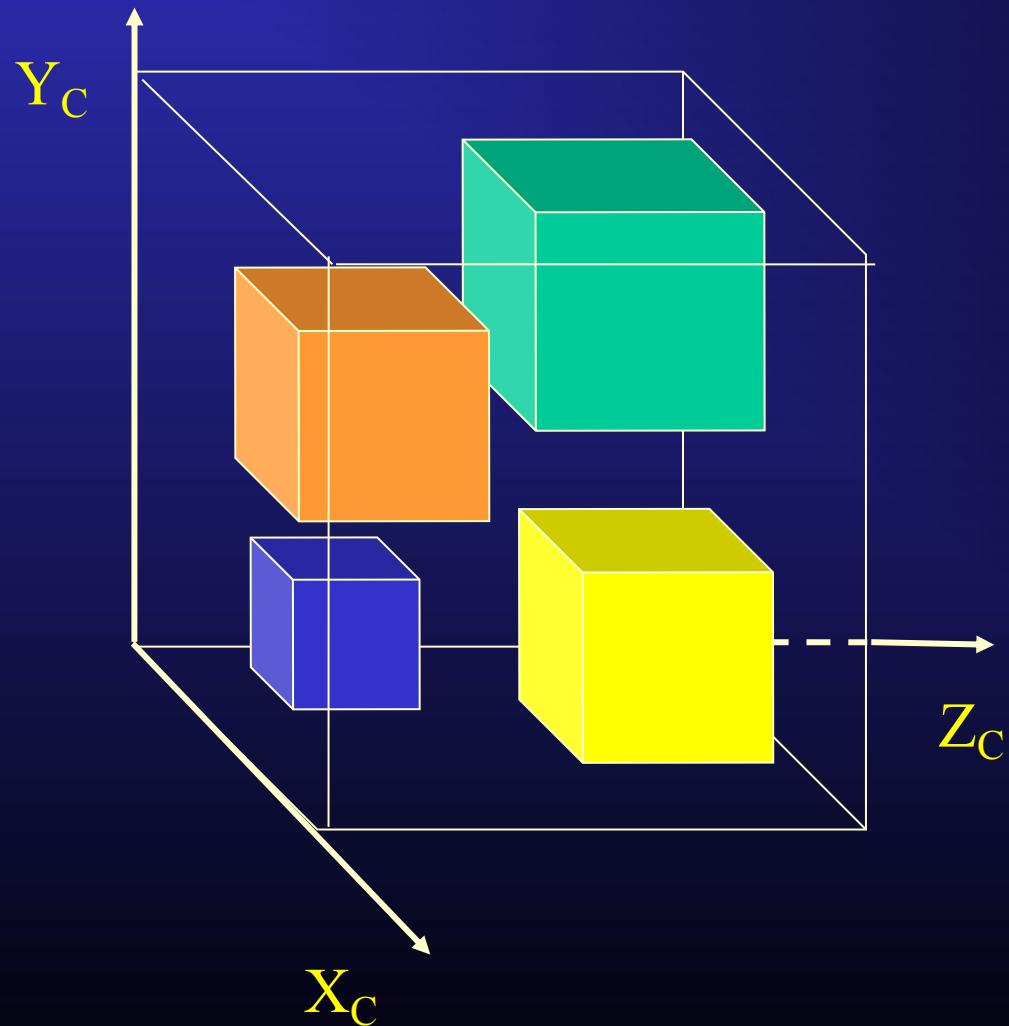


Image Transformations

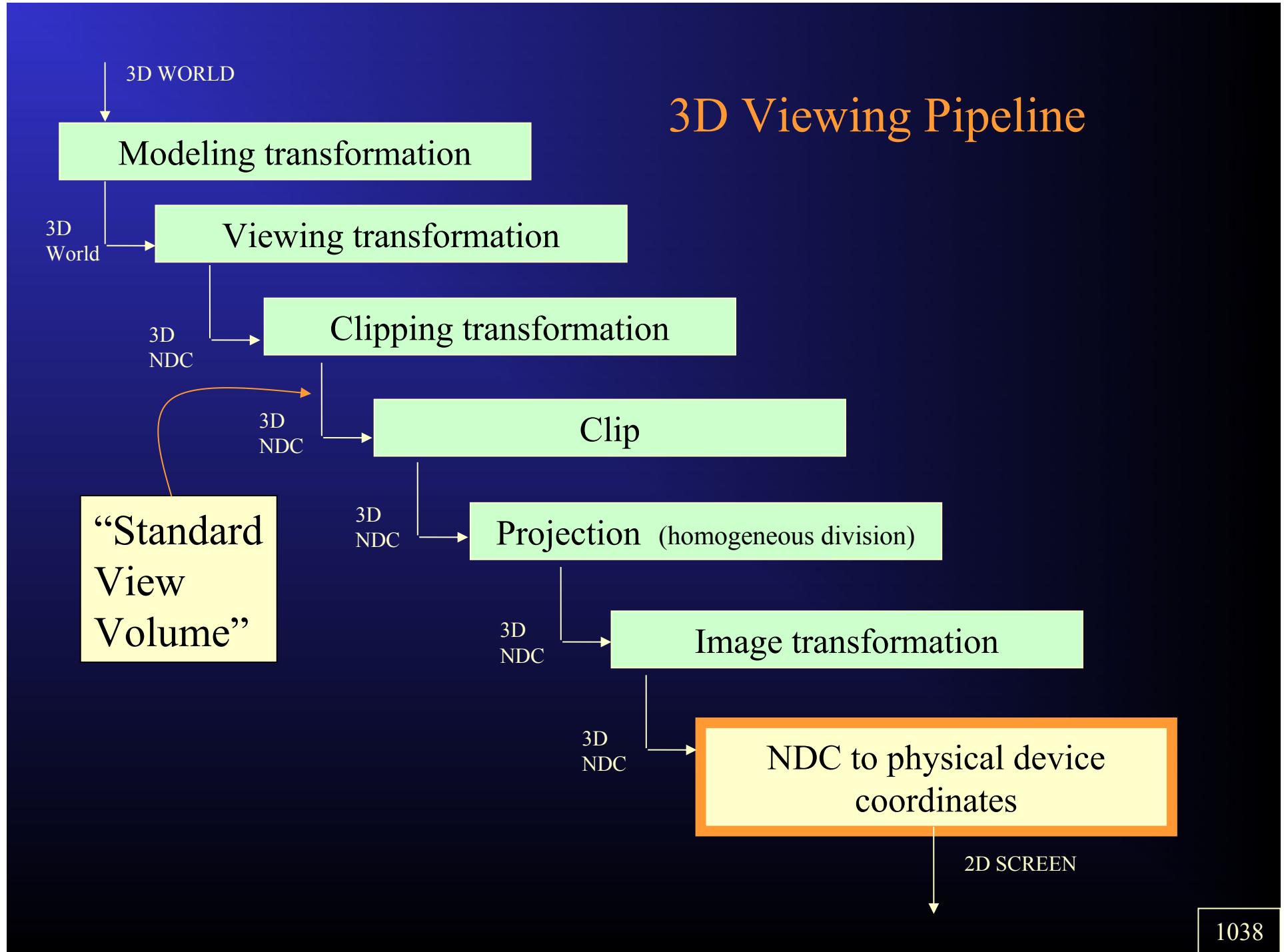
- We have now transformed the scene into a unit cube $[0,1]^3$.
- We can position this unit cube containing the scene anywhere on the display.
- Manipulating it will not change the 3D appearance of the contents, though it may be obscured in a layered viewport (e.g. Windows) system.



Viewport Volumes



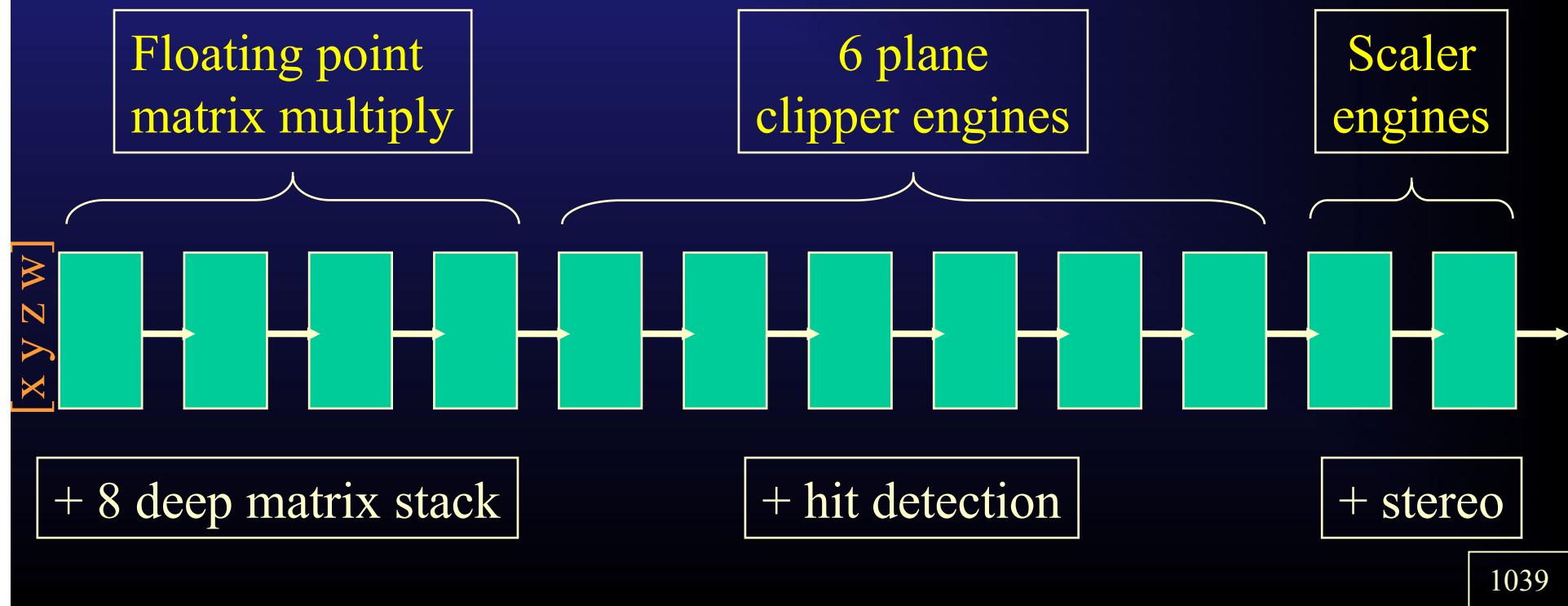
Screen Appearance
(layers displayed
back to front)



The VLSI Geometry Engine

Jim Clark and Forest Baskett → Silicon Graphics (c. 1990)

- Embed 3D perspective pipeline in VLSI
- Re-use common element: **Geometry Engine**
- Computes transformation of input vector $[x, y, z, w]$:



Outline

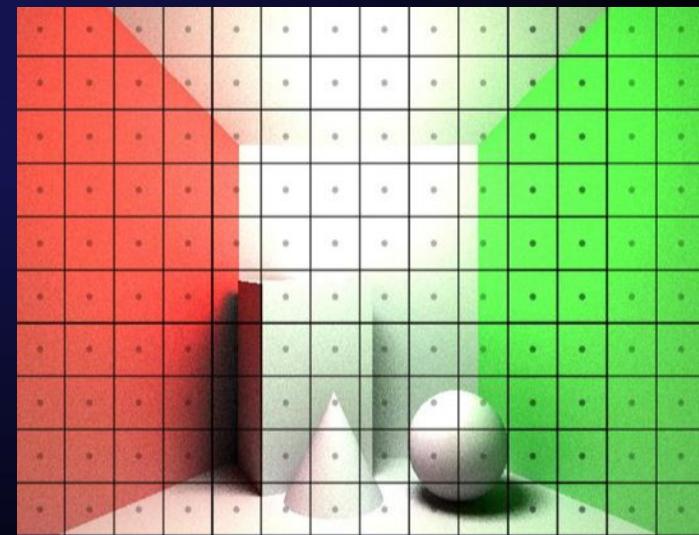
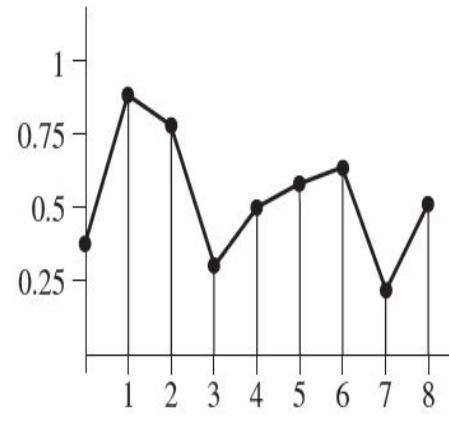
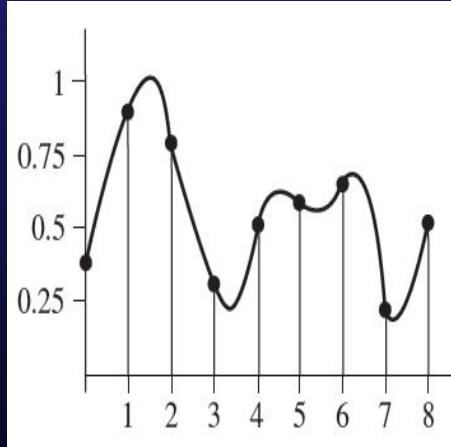
- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading ✓
- Mapping ✓
- 3D Viewing Transformations ✓
- **Anti-aliasing & Compositing**
- Global Illumination
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Anti-Aliasing and Compositing

- Sampling
- Anti-aliasing definition
- Consequence of rasterization
- Prefiltering, supersampling and jittering
- Lines and polygons
- Textures
- Compositing

Sampling and Reconstruction

- To compute the discrete pixel values in a synthesized digital image, we must *sample* the original continuously defined image function. (The geometry is in \Re^3 .)
- *Reconstruction* restores information from the discrete samples as best it can.



Continuous signal \Rightarrow Discrete samples
Linear reconstruction between samples

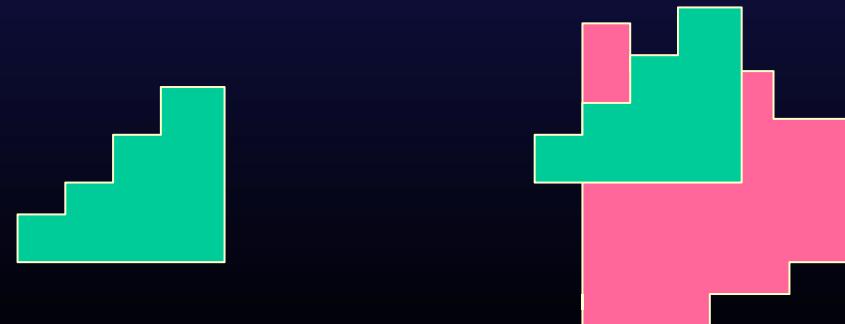
Aliasing = False Frequencies

- Sampling and reconstruction involve *approximation*, which introduces errors called *aliasing*.
- Signal Processing often uses a band-limited (bounded frequencies) assumption to help sampling and reconstruction, but that is not an option in graphics as geometry, lights and shadows will have sharp and thus unbounded frequency edge color transitions.
- Aliasing artifacts include rastering, staircasing, or jaggies, as well as noise, blur, and unwanted color gradients.

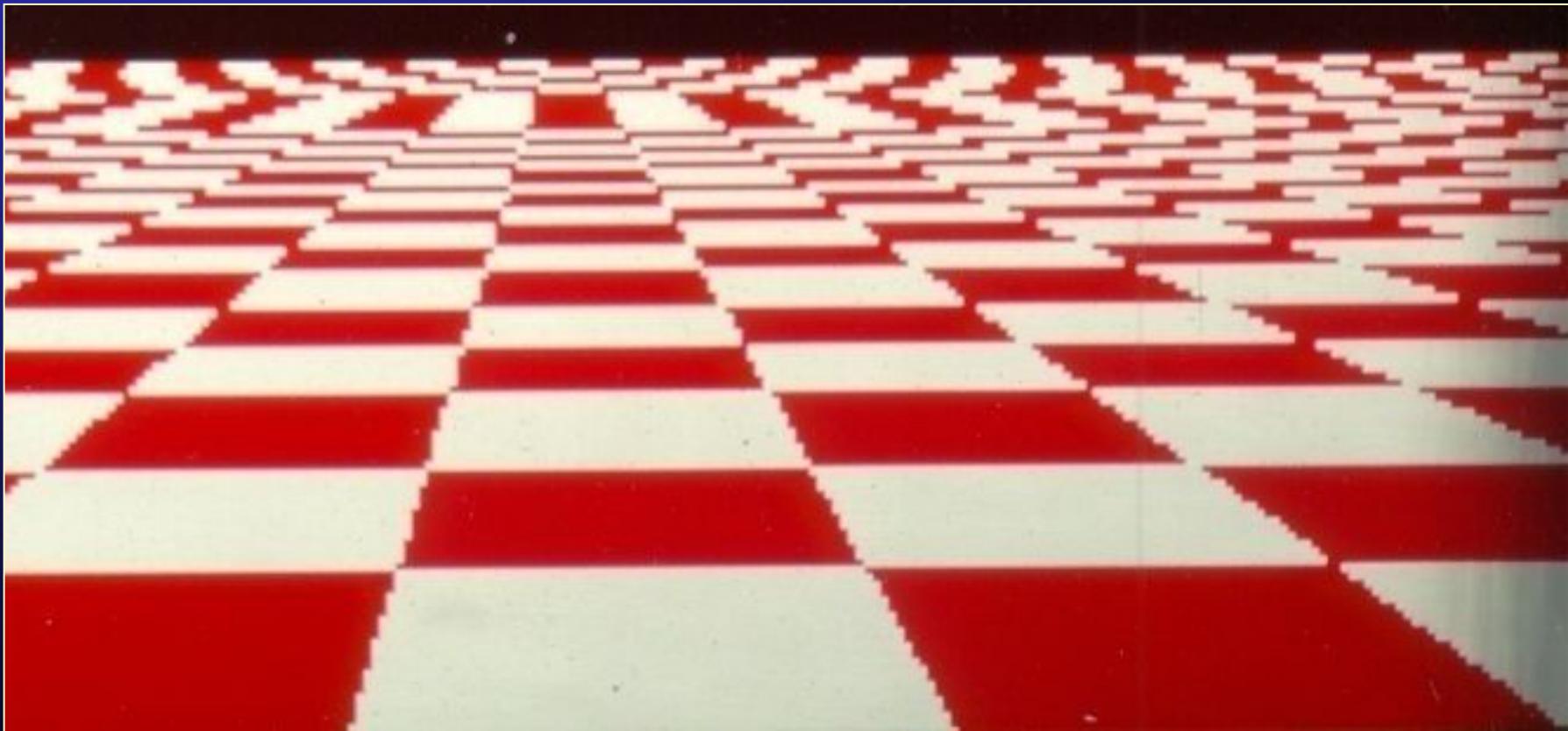
Anti-Aliasing: Sample well!

Anti-aliasing tries to remove or minimize such artifacts by:

- Reducing raster quantization artifacts from lines and shapes.
- Smoothly combining overlapping surfaces/shapes.
- Rendering sub-pixel detail.
- Reducing interference patterns (Moiré patterns)
- Reducing other undesired beats or patterns in the image.
- Avoiding texture disintegration.
- Reducing high frequency noise (flickering) in animation.



Disintegrating Textures



slide

Moiré Patterns

rotate

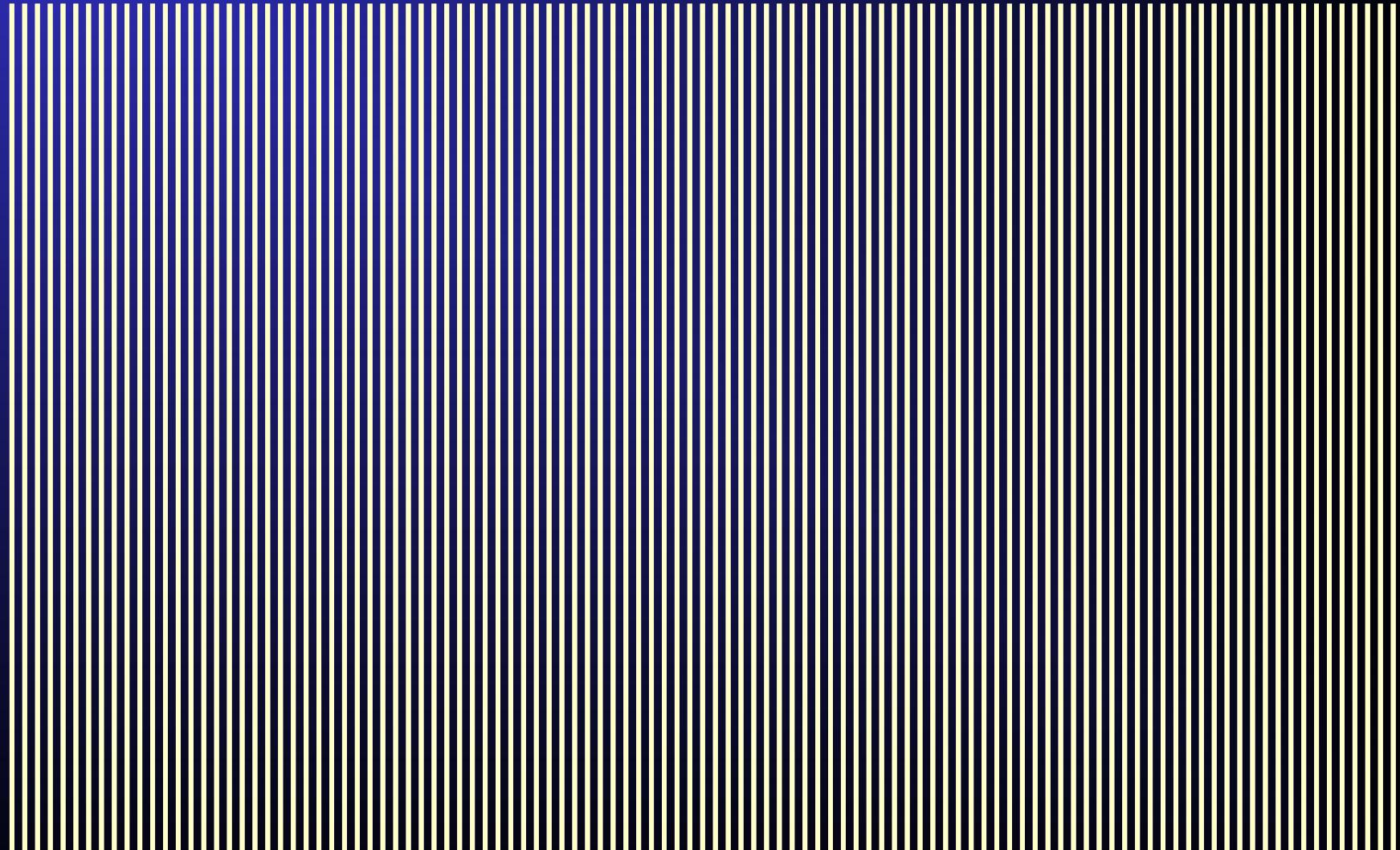
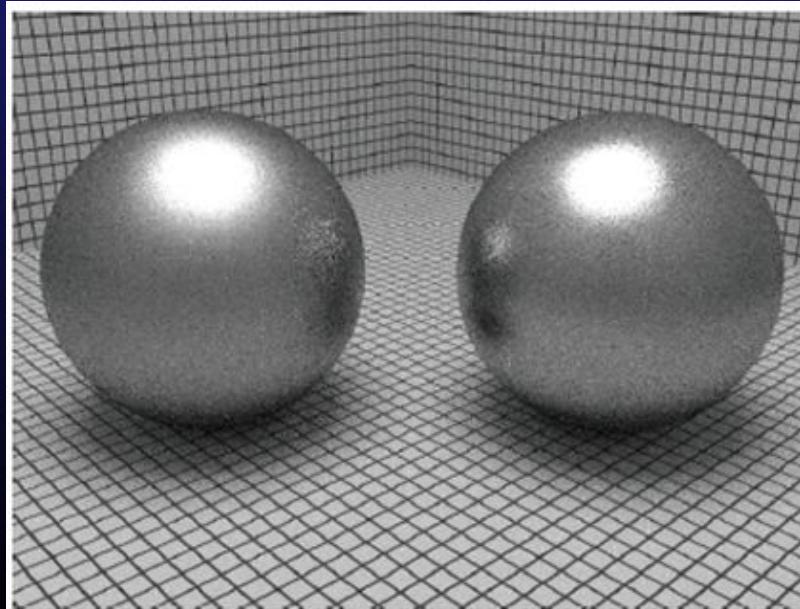


Image Sampling Comparison

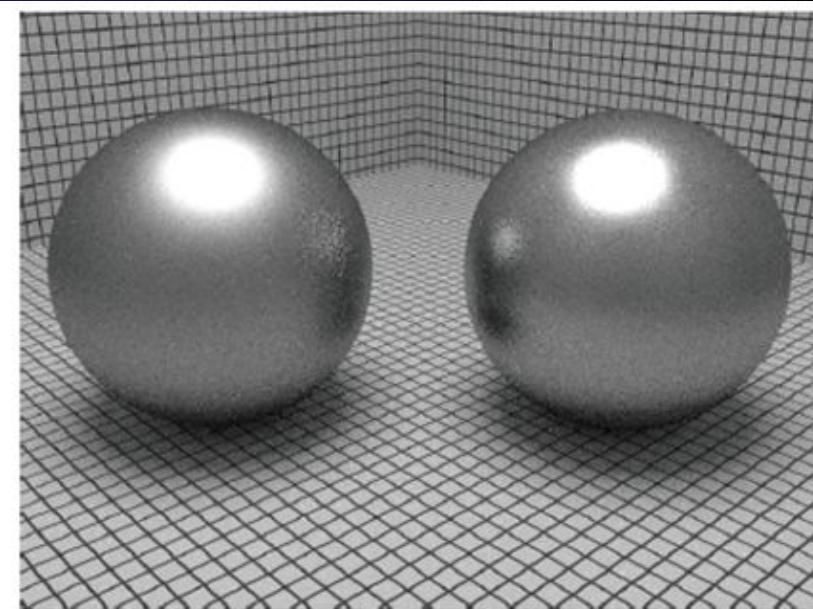
For rendering using ray methods, a good sampling pattern is desirable because it requires fewer rays to create a high quality image than if a lower quality pattern were used. Every ray is potentially expensive.

(Same number of rays)

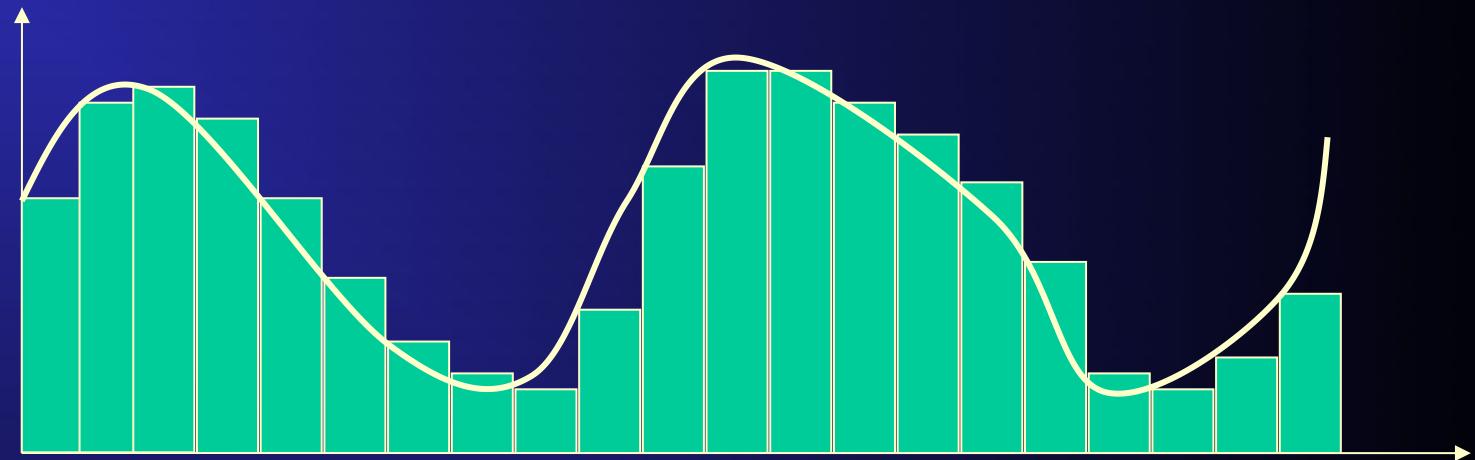
Poor sampling



Better sampling

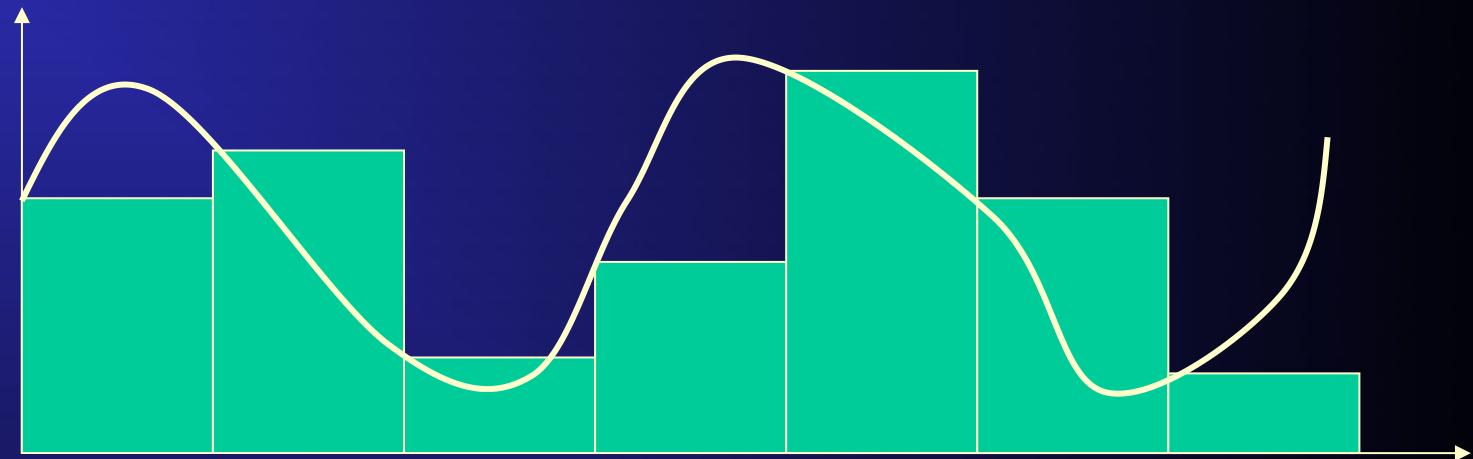


A Brief Look at Sampling Theory (1)



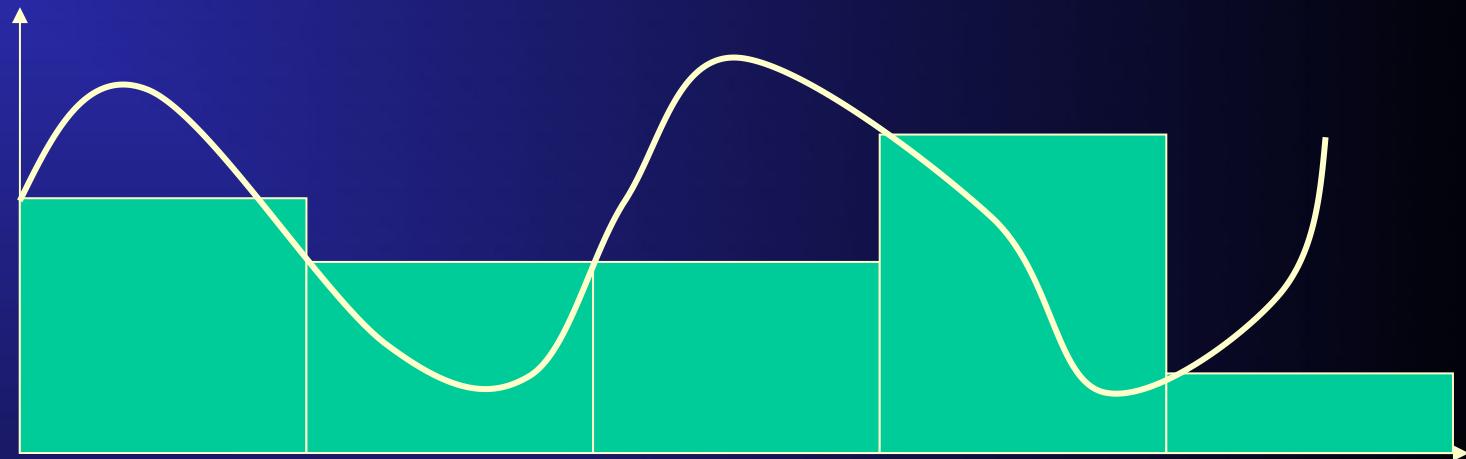
- Closely spaced samples give relatively good approximation to original signal.
- In general, samples reproduce the original frequency f if
$$f < \frac{1}{2} \text{sampling frequency}$$
(Nyquist Limit)

A Brief Look at Sampling Theory (2)



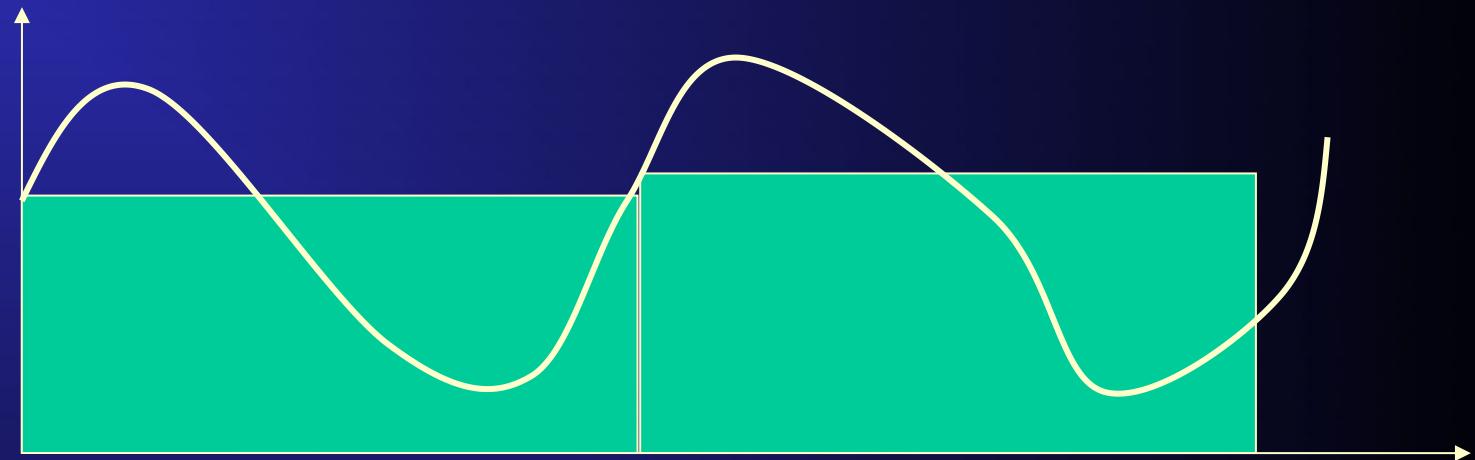
- Widely spaced samples give poor approximation to original signal.
- In general, samples reproduce the original frequency f if
$$f < \frac{1}{2} \text{sampling frequency}$$
(Nyquist Limit)

A Brief Look at Sampling Theory (3)



- Very widely spaced samples give wrong frequencies.
- In general, samples reproduce the original frequency f if
$$f < \frac{1}{2} \text{sampling frequency}$$
(Nyquist Limit)

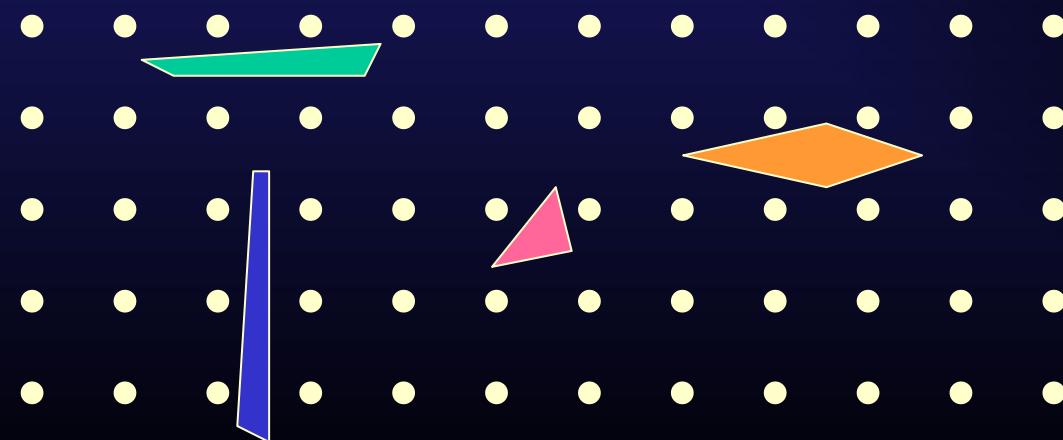
A Brief Look at Sampling Theory (4)



- Extremely widely spaced samples are really aliased.
- Demo: 
- In general, samples reproduce the original frequency f if
 $f < \frac{1}{2}$ sampling frequency
(Nyquist Limit)

Two Principal Techniques

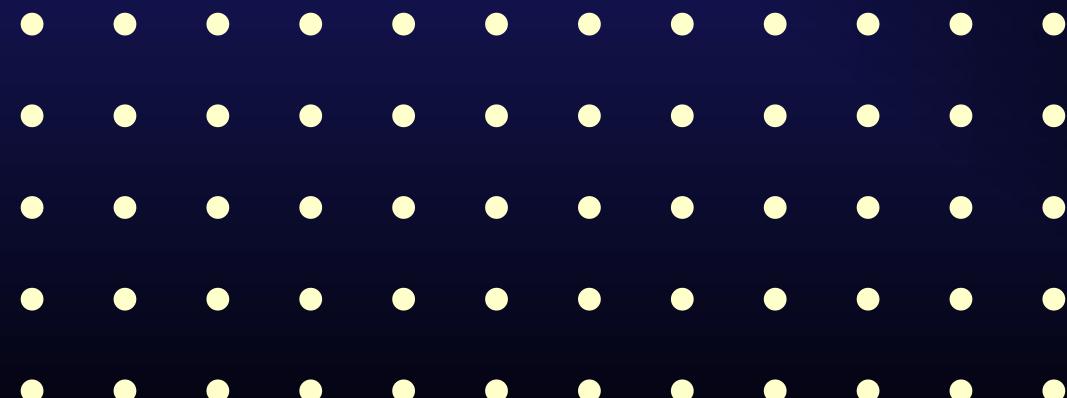
- Prefiltering at display resolution.
- Supersampling and filtering or averaging:
 - Often done at higher than display resolution, otherwise is just a blurring function.
 - Note that averaging or filtering without supersampling will result in the omission of small details which fail to be sampled by the raster grid:



- Looks like this ...

Two Principal Techniques

- Prefiltering at display resolution.
- Supersampling and filtering or averaging.
 - Often done at higher than display resolution, otherwise is just a blurring function.
 - Note that averaging or filtering without supersampling will result in the omission of small details which fail to be sampled by the raster grid:



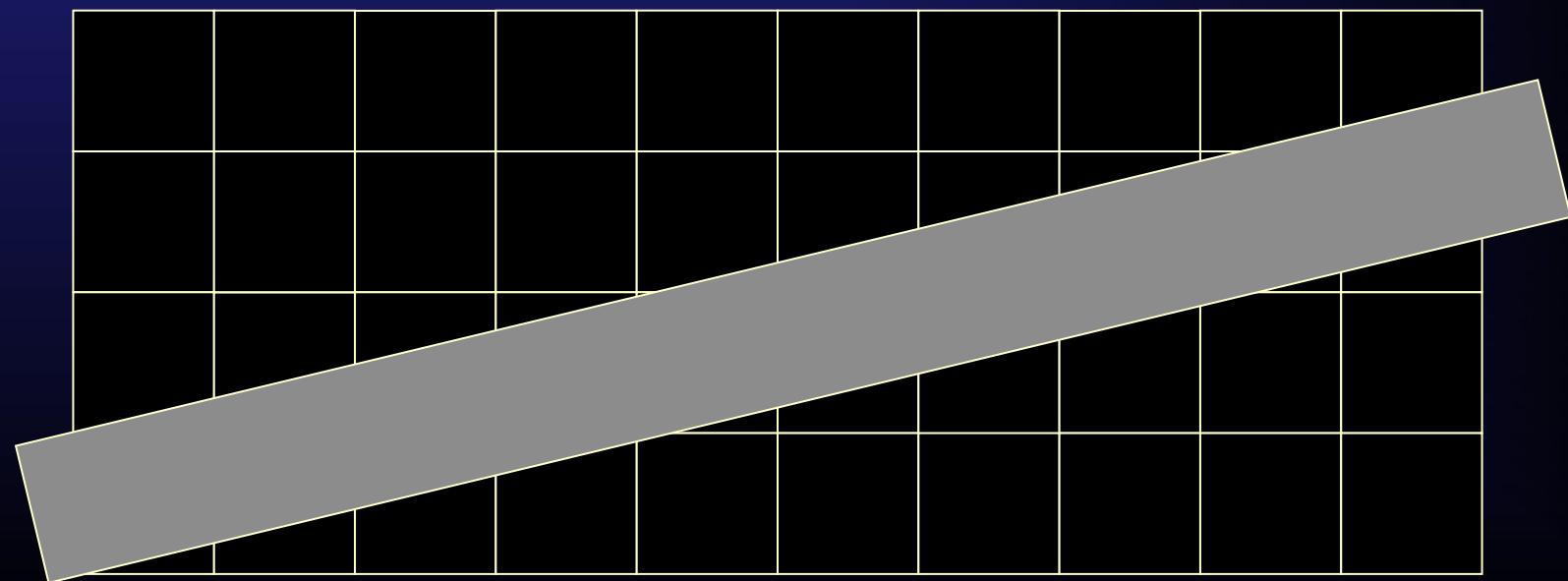
- Looks like this ... especially disruptive during animation!

Prefiltering Polygons

- Treat each pixel as an AREA rather than as a point sample.
- Pixels then cover the display viewport, eliminating the sampling problem just shown.
- Calculate sub-area within a pixel's area covered by a polygon that overlaps it.
- Can use polygon clipping algorithm to compute area where an edge passes through a pixel area.
- Efficient for simple shading, but costly for highly textured surfaces and non-planar geometry.
- First we'll do lines, then areas (polygons), then overlapping combinations.

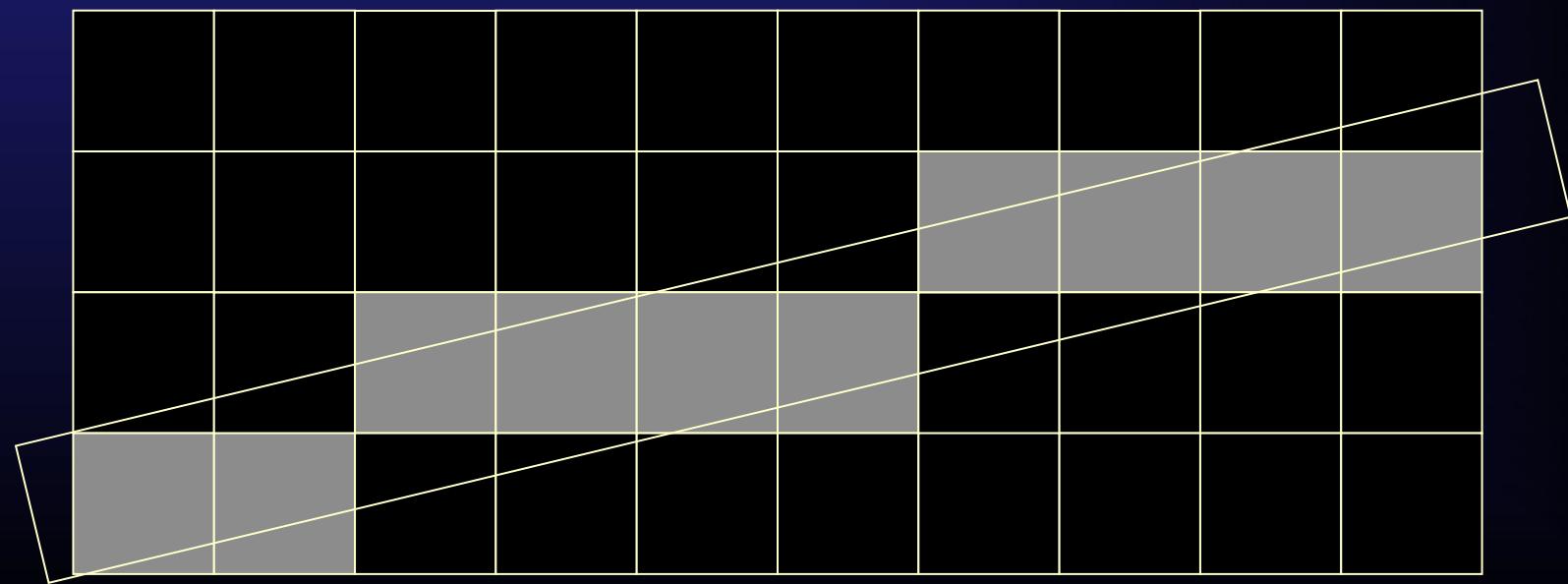
Aliased Lines

- If we only light pixels whose centers lie in the line boundary (assuming the line is one pixel “thick”) we get:



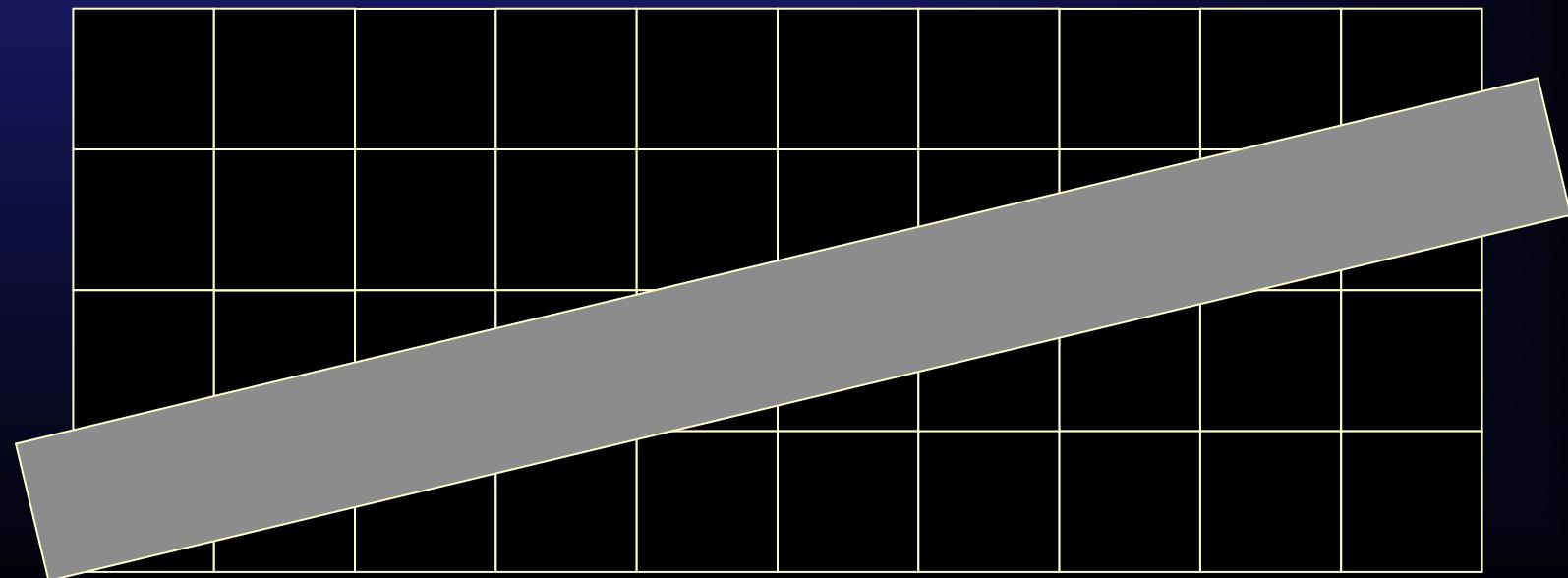
Aliased Lines

- This badly stair-stepped image:

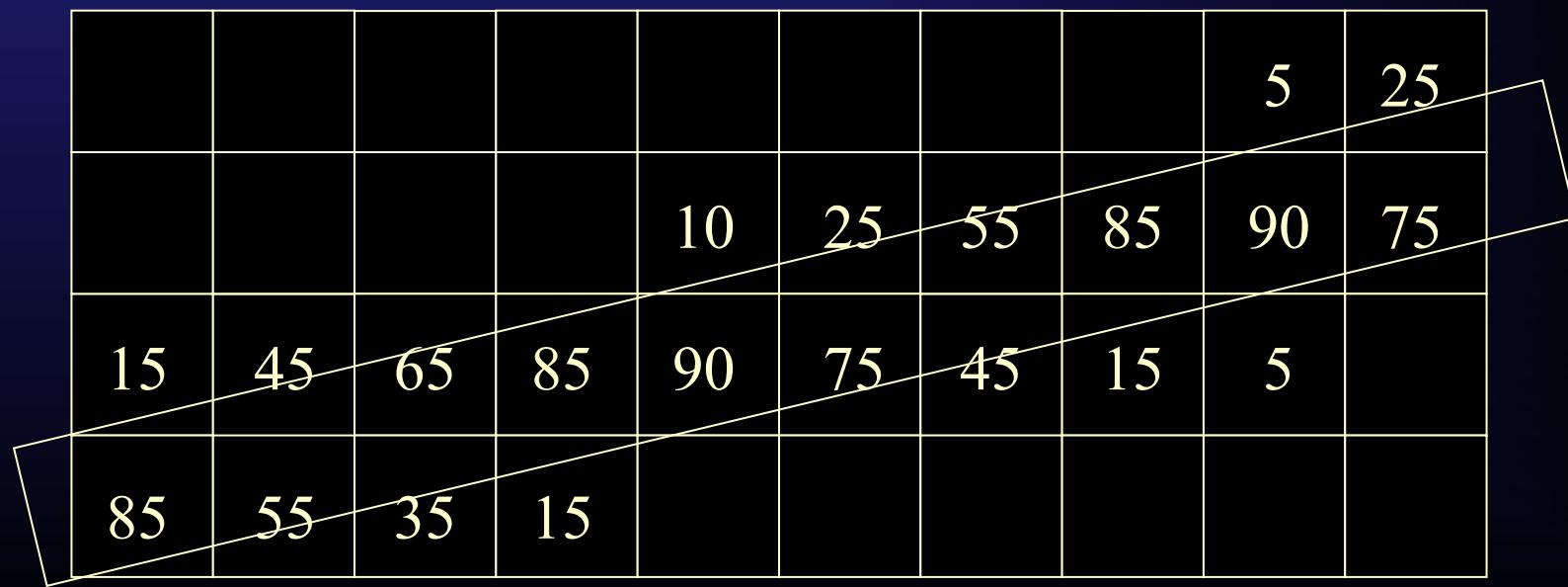


Anti-Aliasing Lines

- So instead, consider the line as a thin polygon, and use percentage of area overlap on each affected pixel as a weight on the line intensity or color.

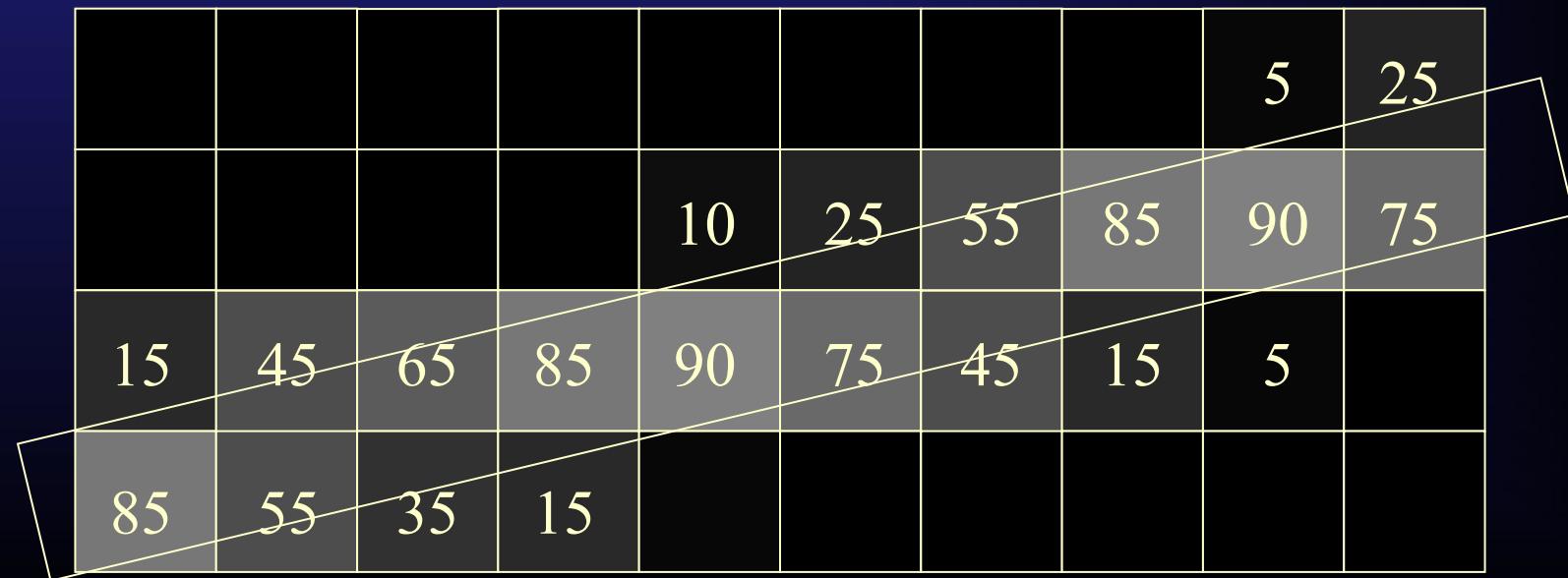


Coverage Percentages

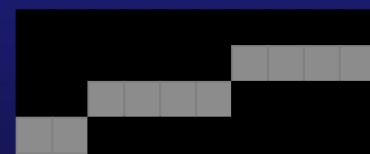
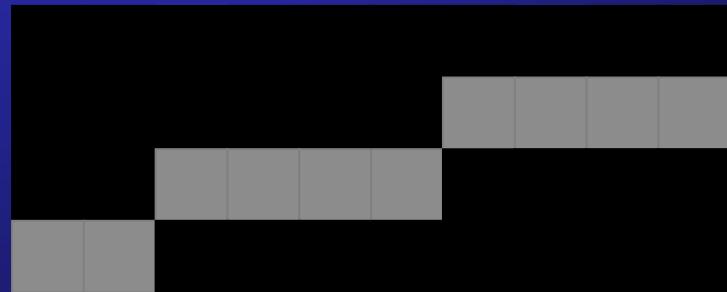


Coverage Percentages Drawn in Gray Scale

Notice that each column adds up to 100%.



Reduced ($1/2^n$), $n=1,2,3,4,5$ to Show Effect

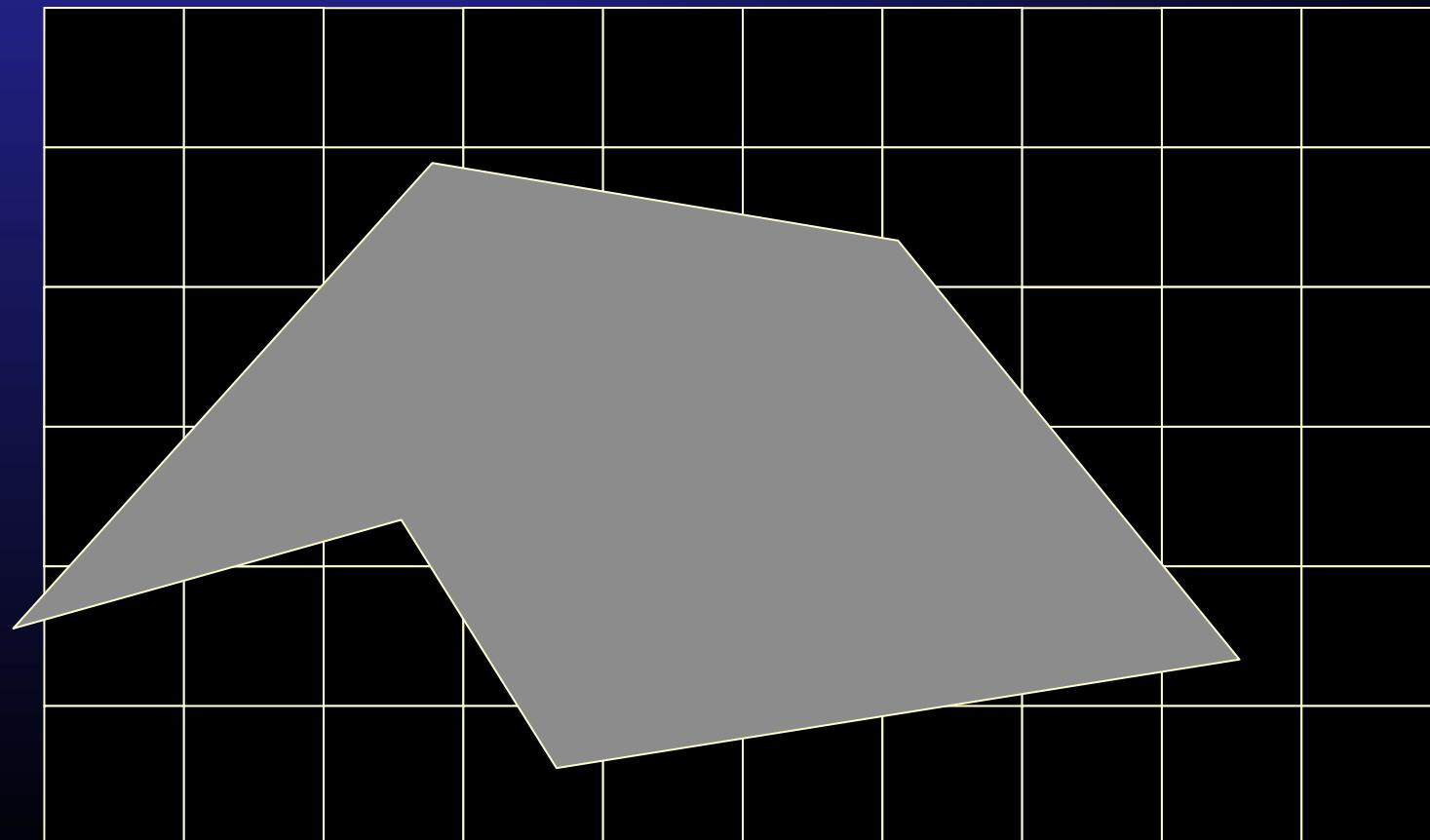


Demo:

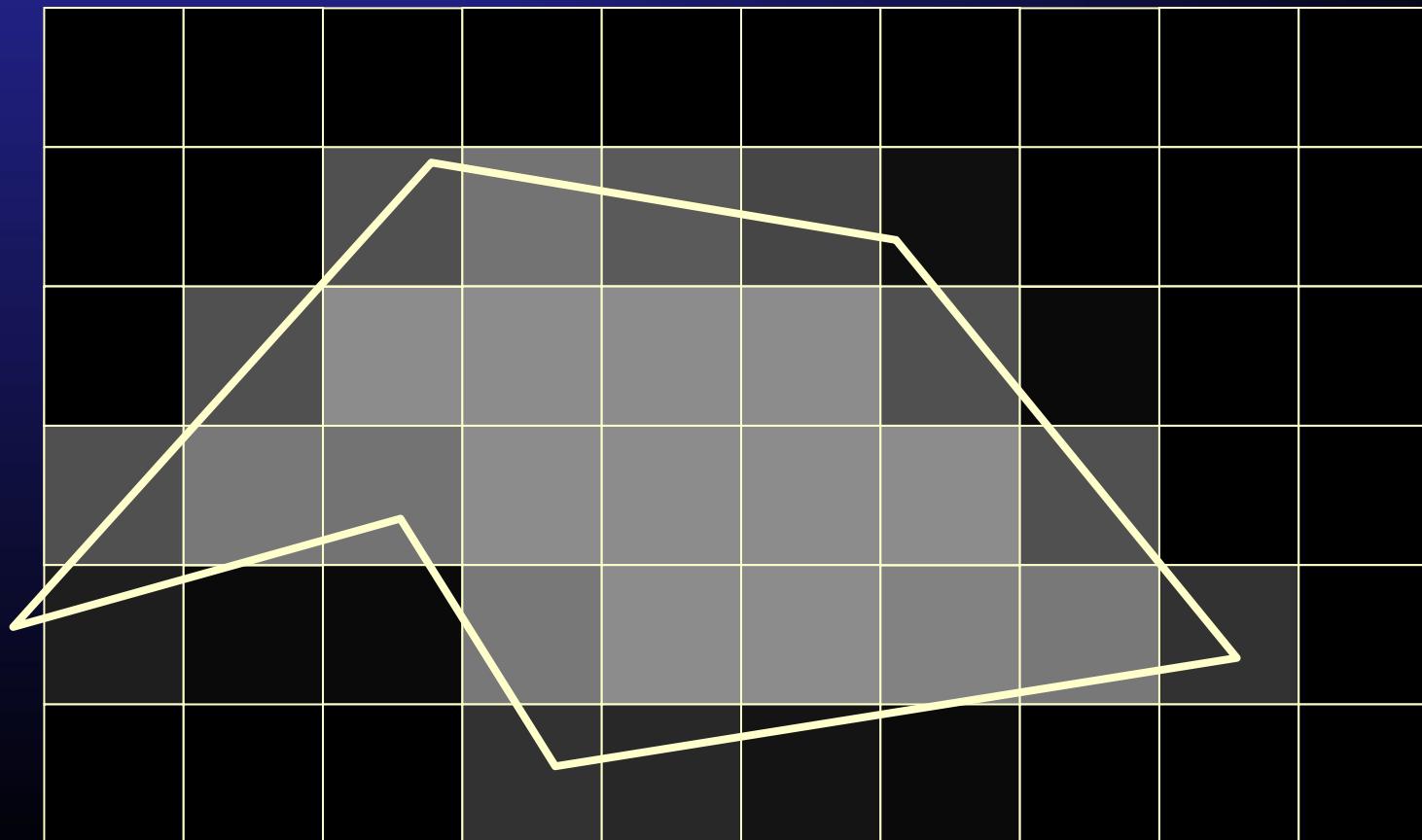


Anti-Aliasing Polygons

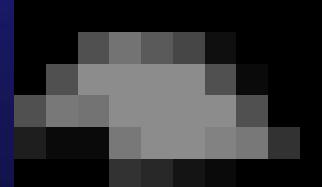
- If we can do lines by considering them as thin polygons, then we can do polygons by the same area coverage process.



Rasterized with Coverage Percentages

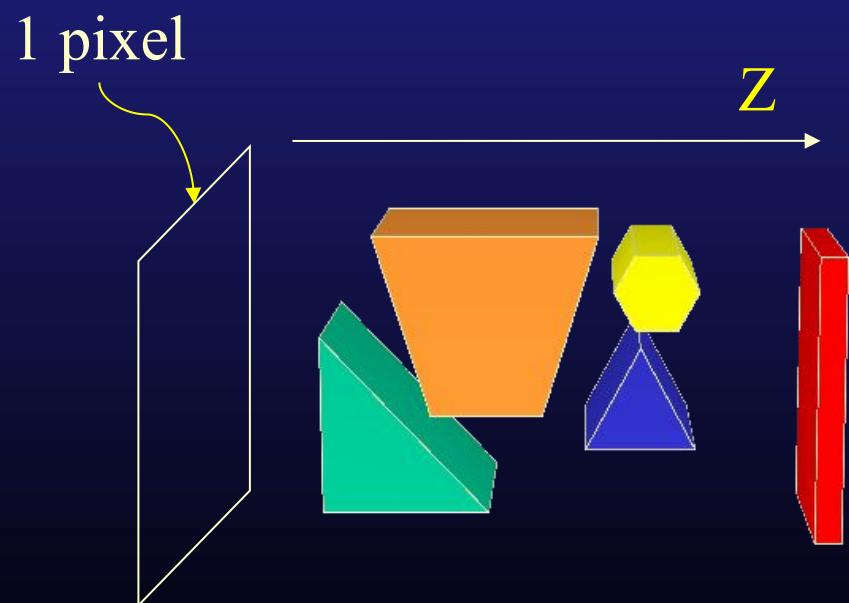


Reduced ($1/2^n$), $n=1,2,3,4,5$;
The Result isn't too Bad...



What about Overlapping Shapes?

- An accurate prefiltering approach requires keeping a depth-sorted list of coverage fragments as well as the numerical coverage fraction:
- Used in modern GPUs!



Looking down the Z
axis through the pixel:



The final area-weighted
average color

Supersampling and Filtering

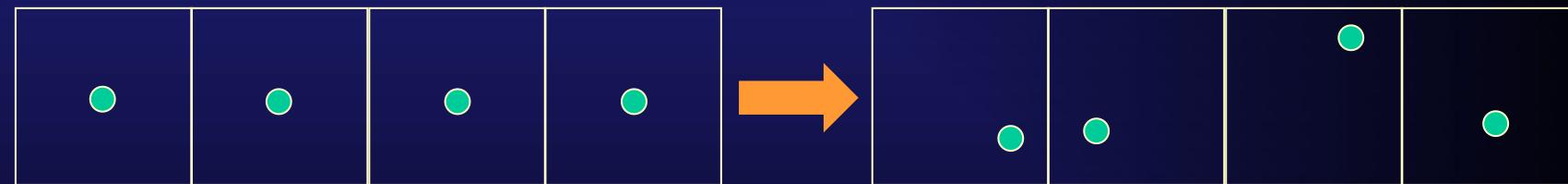
- Supersample image at higher than screen resolution.
- Reduce to screen resolution by averaging or filtering by various kernels: multiplicative masks with weights over some support (square or round domain), e.g., some 3x3 kernels:
 - Averaging or Fourier = uniform weights


$$\begin{matrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{matrix}$$
$$\begin{matrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{matrix}$$

- Bartlett or Gaussian = Gaussian distribution of weights

Stratified Sampling and Jittering

- Improve overall image quality by adding noise through *stratified sampling* and *jittering*.
- Stratified sampling creates regular sub-areas in the image.
- Jittering involves taking one sample per sub-area, but moving the sample from the center to a **random position** within the sub-area:

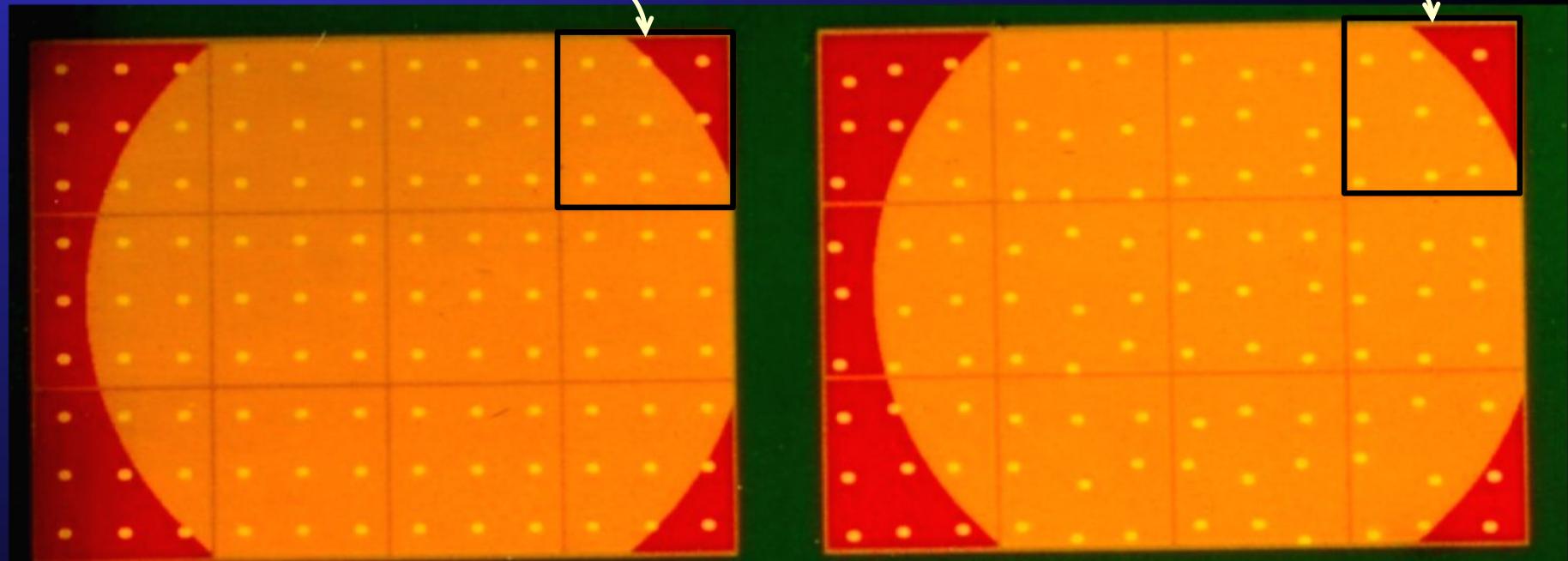


- Filter weights are the same as the non-jittered cases, only the location of the sample changes.
- Guarantees minimum sample separations.
- Adds white noise to break up moiré patterns and other sampling artifacts.

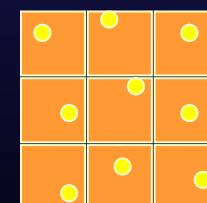
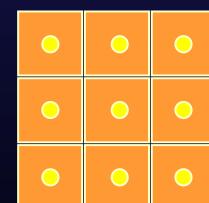
Comparison of 3x3 Regular and Jittered Supersamples

2/9 red + 7/9 yellow

1/9 red + 8/9 yellow



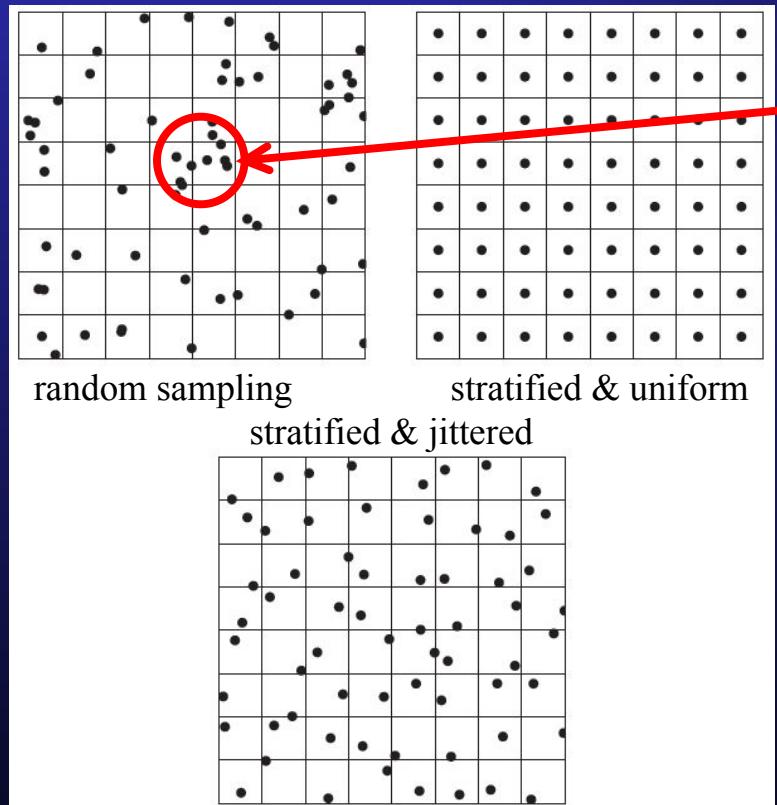
Each regular supersample stays at the center of its subarea within the pixel



While each jittered sample is taken somewhere inside its subarea within the pixel

Stratified Sampling and Jittering

- It does make a difference!

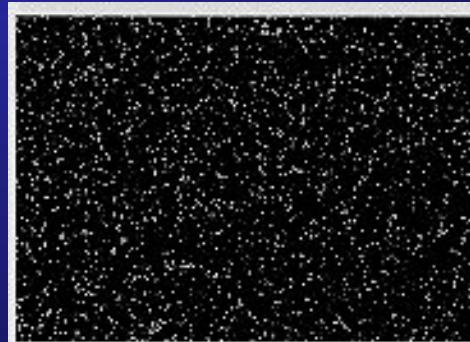


Helps avoid accidental clustering of random samples

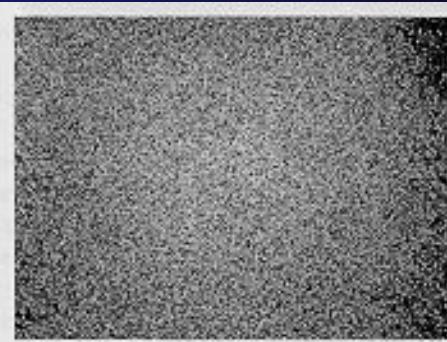


ground truth random sampling stratified sampling

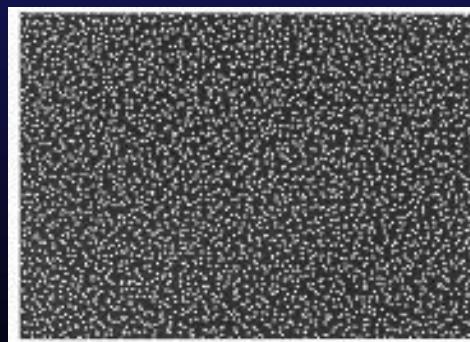
Spatial Frequency Comparison of Random vs. Jittered Image Samples



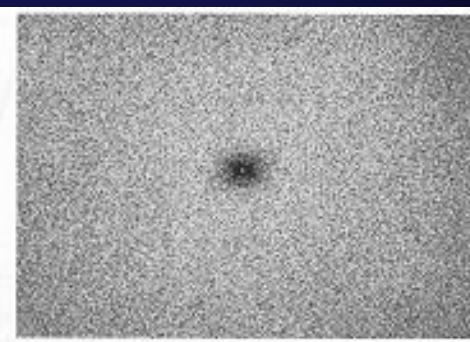
Random Samples



Fourier Distribution

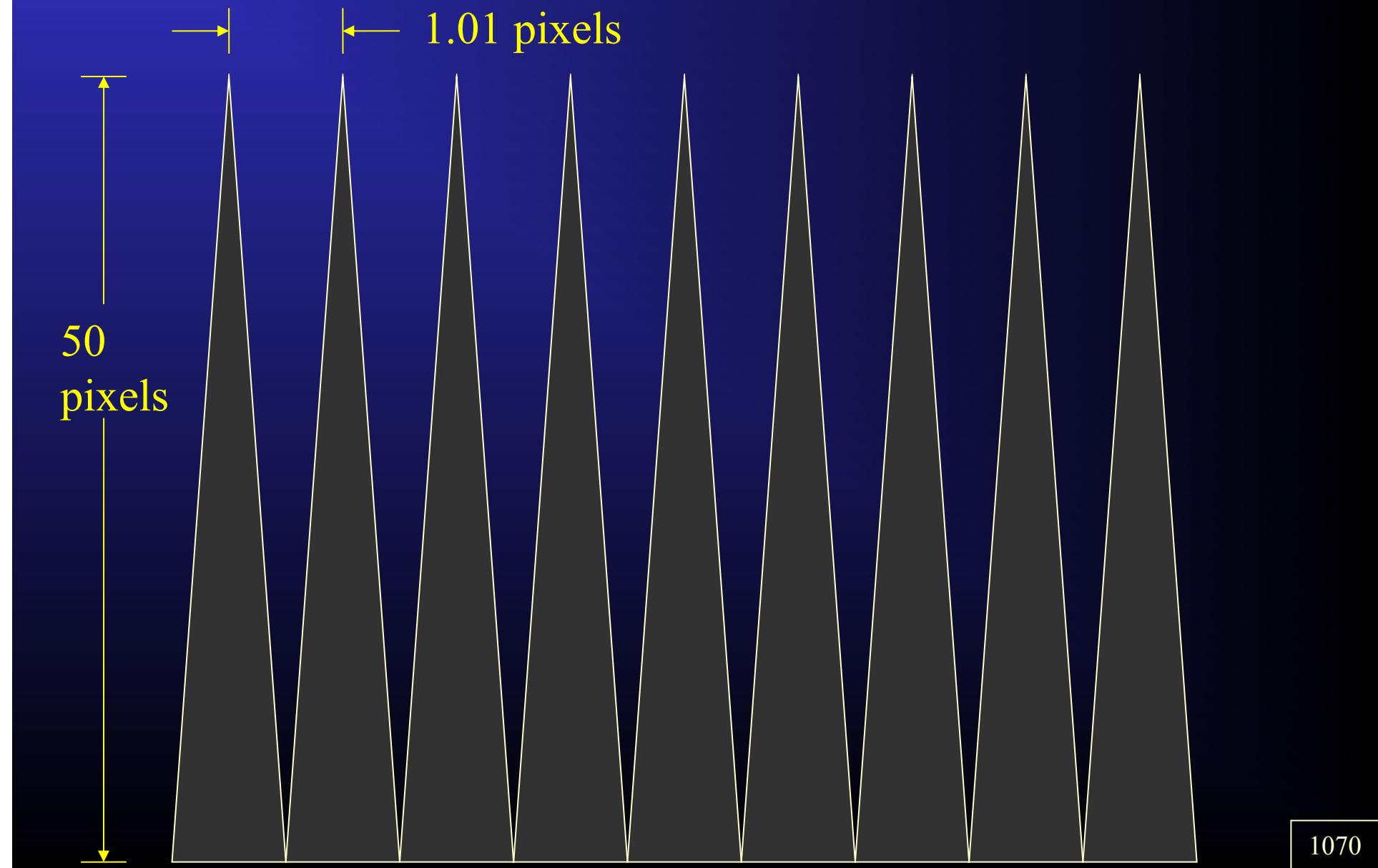


Jittered Samples

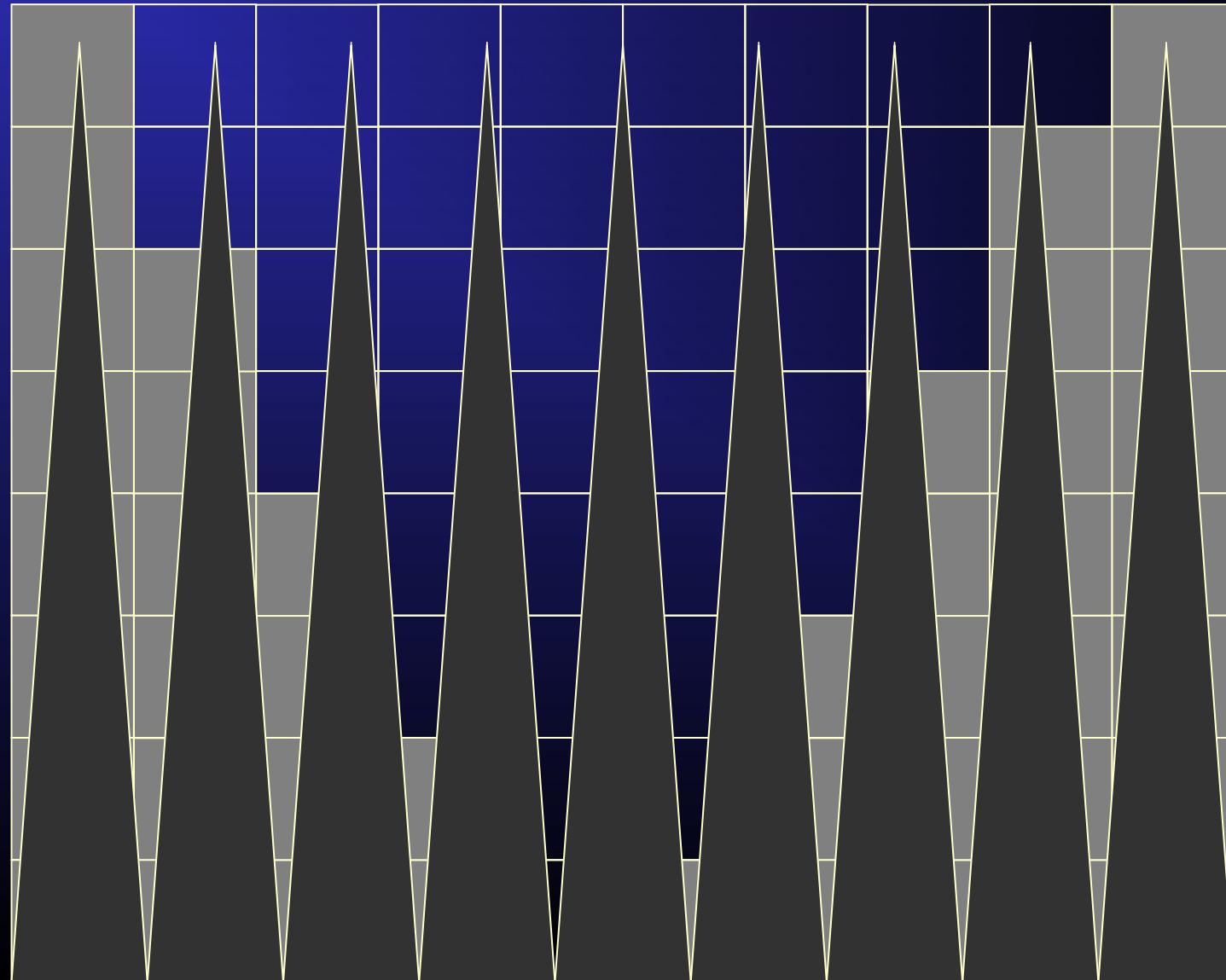


Fourier Distribution

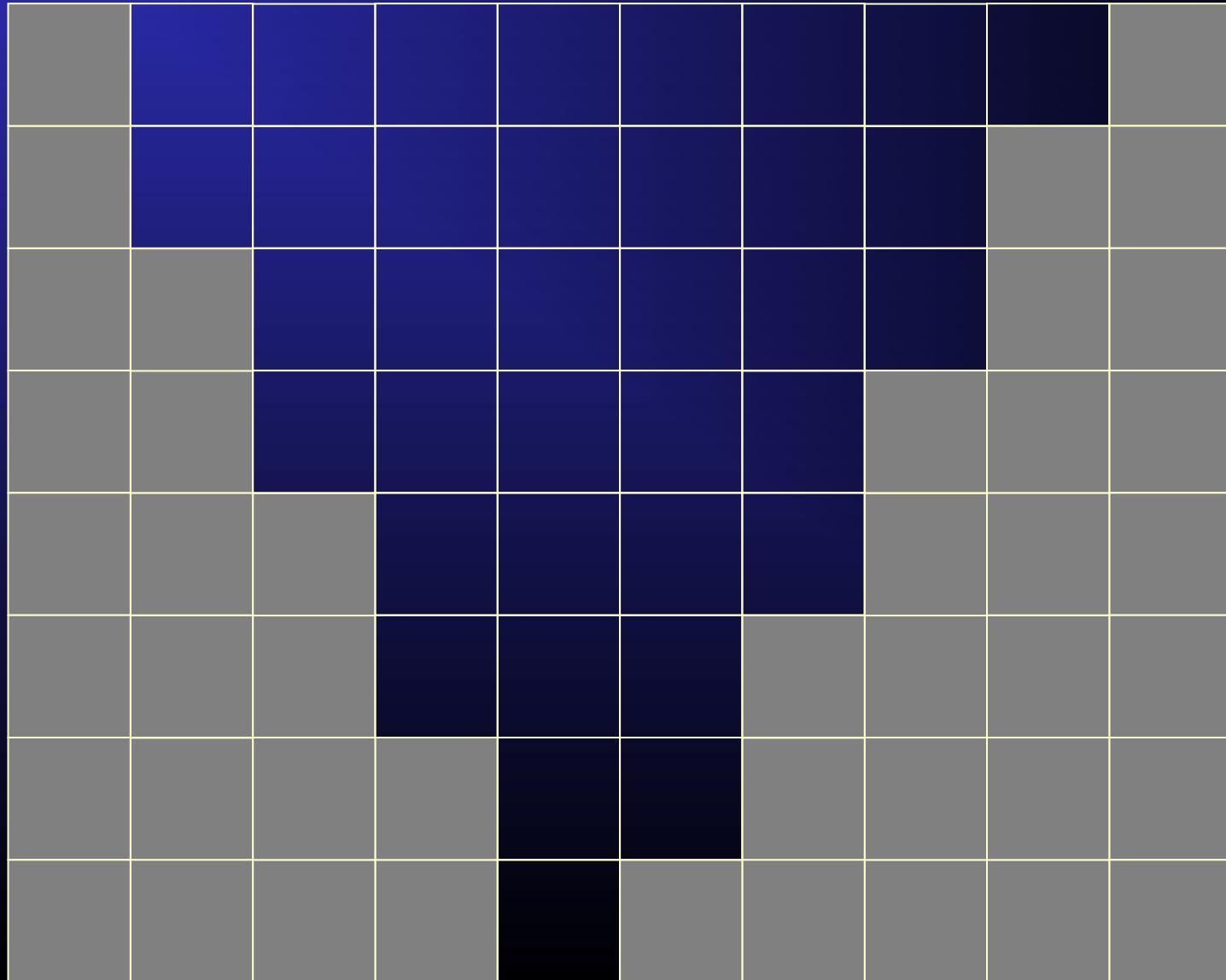
The Comb Image (Schematic/Animated)



The Comb Image, Center Pixel Sampled

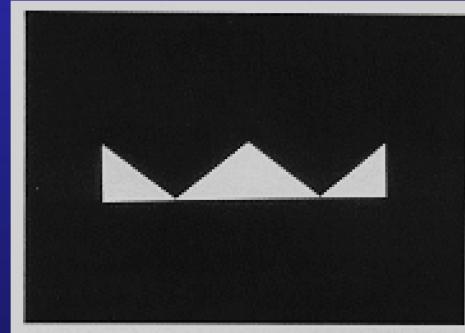


Center Pixel Samples Alias Badly

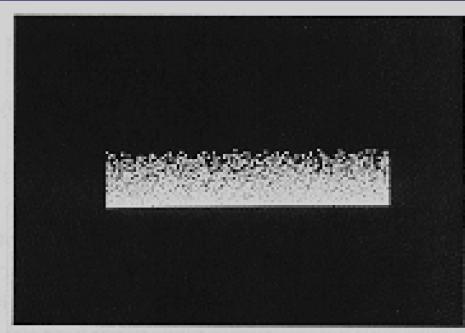


1072

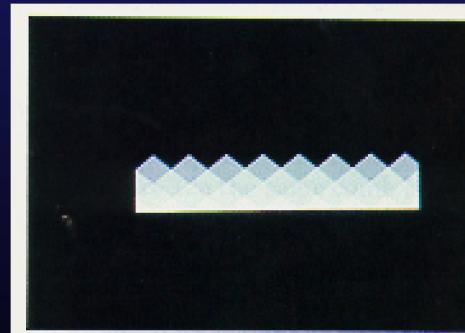
Comb Results: (50 x 1.01 pixel combs)



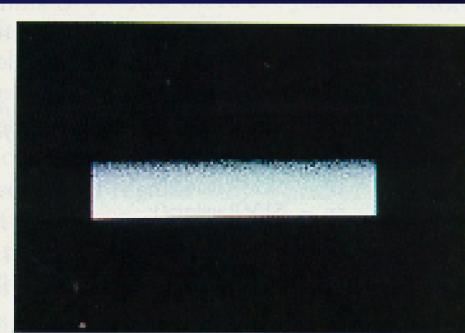
regular grid, 1 sample/pixel



jittered grid, 1 sample/pixel



regular grid, 16 samples/pixel



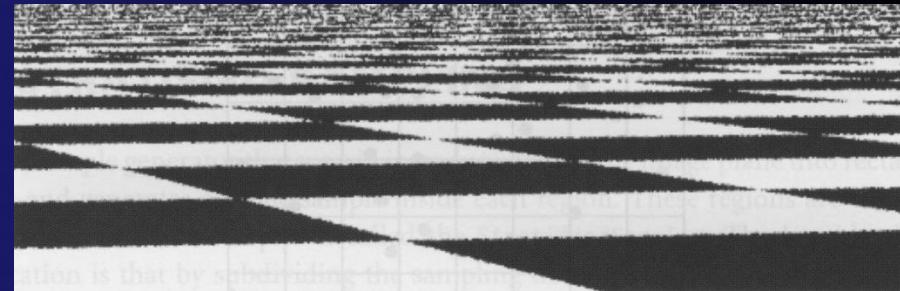
jittered grid, 16 samples/pixel

Uniform Sampled vs. Stratified/Jittered

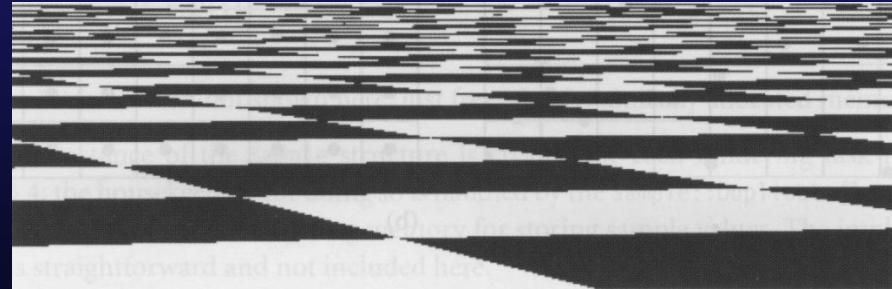
256 sample/pixel reference



4 uniform samples/pixel



1 uniform sample/pixel; no jitter



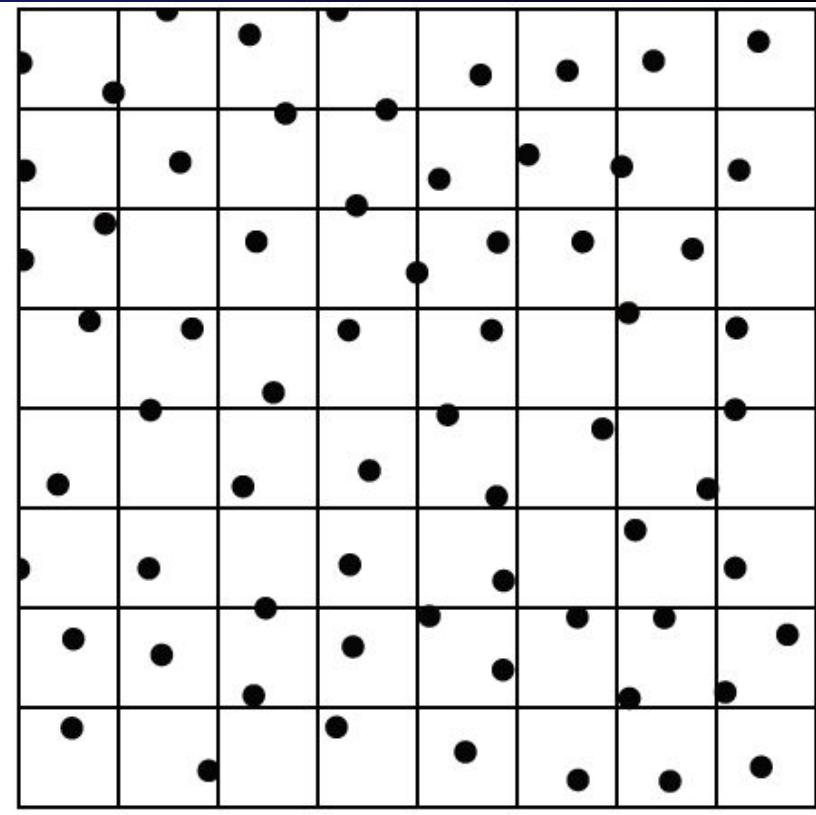
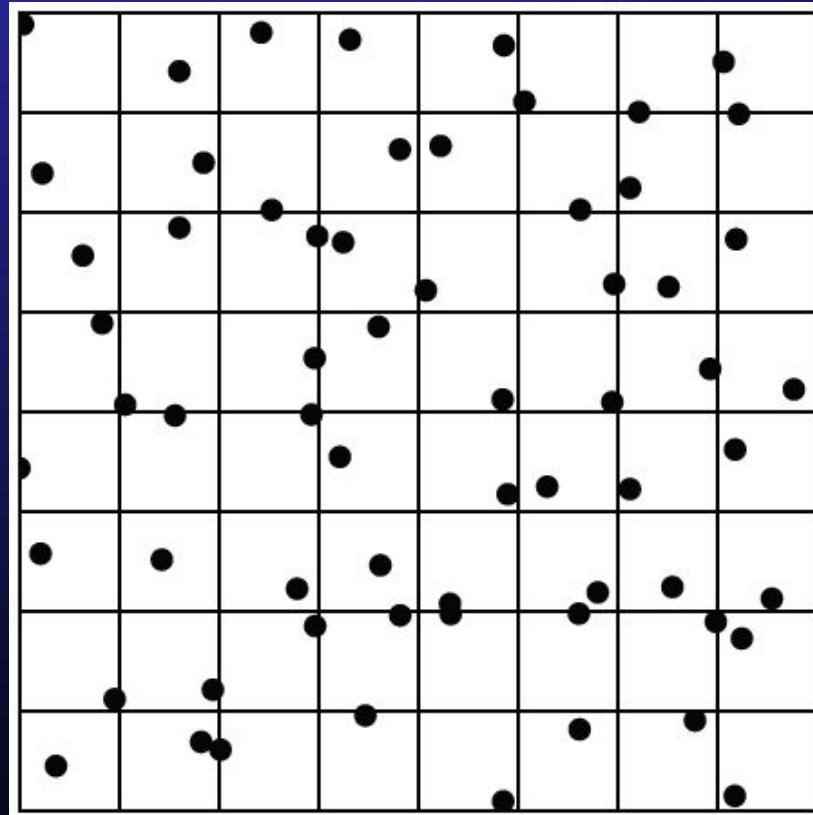
4 samples/pixel with jitter



Best-Candidate Sampling

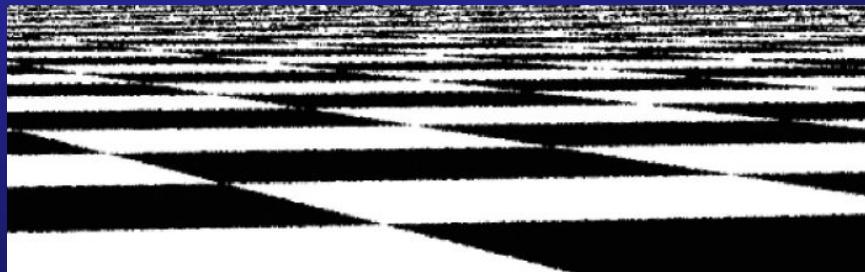
- Poisson disk pattern: no two points are closer than some minimum distance.
- One possible implementation: “Dart-throwing”:
 - Generate random sample, then discard if it is too close to an existing one.
- Best-candidate
 - Generate a limited (large) number of random samples, discard all but the best.
 - Poisson-like, but not guaranteed.
 - Advantage is that any prefix set of samples is well-distributed.

Jittered vs. Best-Candidate



Stratified/Jittered vs. Best-Candidate

1 sample/pixel jittered



1 sample/pixel best-candidate



4 samples/pixel jittered



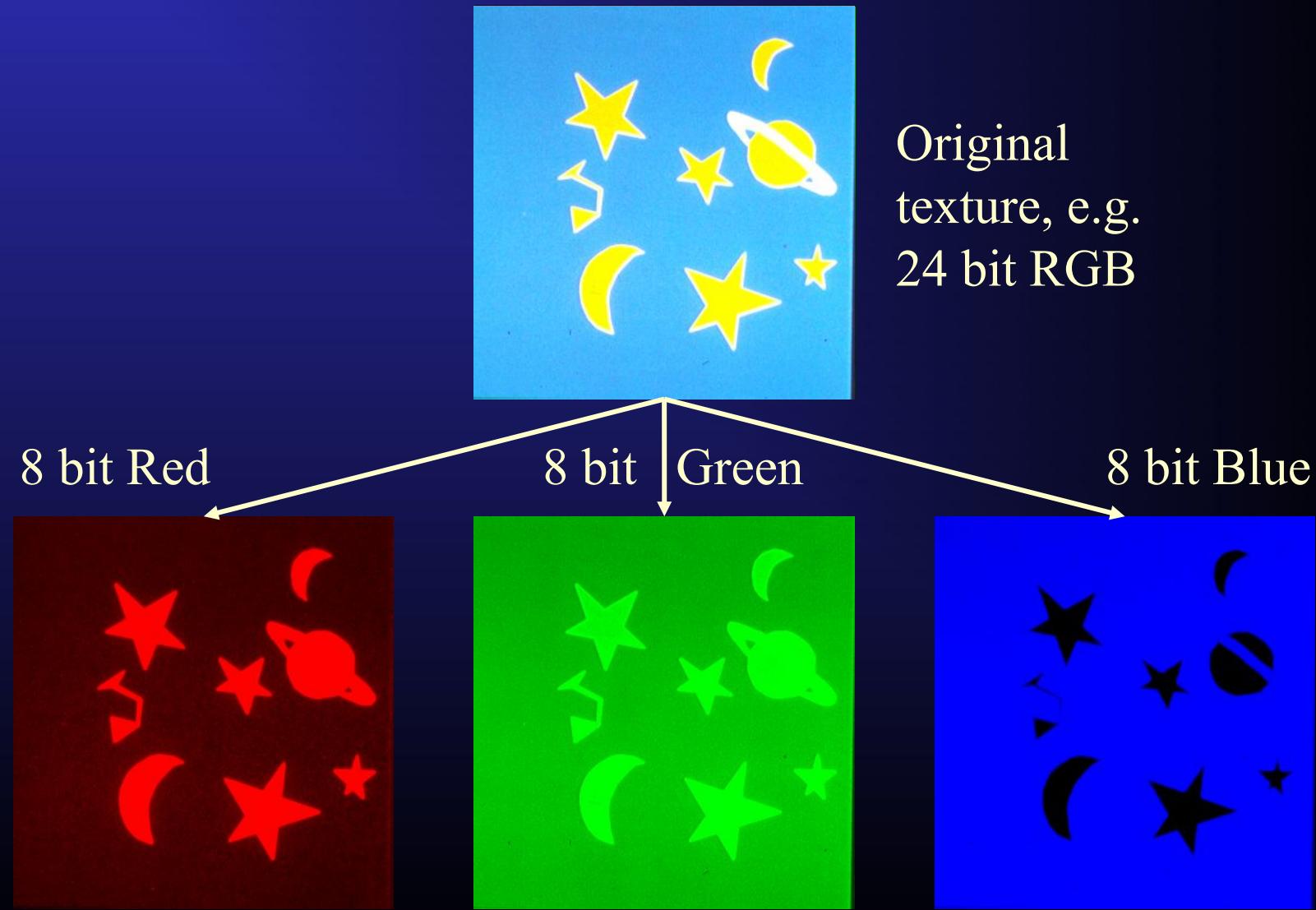
4 samples/pixel best-candidate



Anti-Aliasing Textures by Prefiltering

- Texture RGB pixel called a *texel* (texture element).
- Create *prefiltered* copies of the texture at all resolutions from original down to a single pixel: a **mip-map**.
- Pyramidal scheme due to Lance Williams.
- The texture memory needed will be only 33% larger than the original image size. Thus a 512 x 512 x 3 byte RGB texture will take up 1024 x 1024 bytes (512 x 2 x 512 x 2).
- Perform area averaging *bi-linear interpolation* within a particular resolution level.
- Final texture color computed by *tri-linear interpolation*: linear interpolation between original filtered pyramid levels plus in-level (bi-linear) filtering .

Texture Pyramids: First create RGB Separations



Consider each Separation as an 8-bit Gray Image

Red



Green

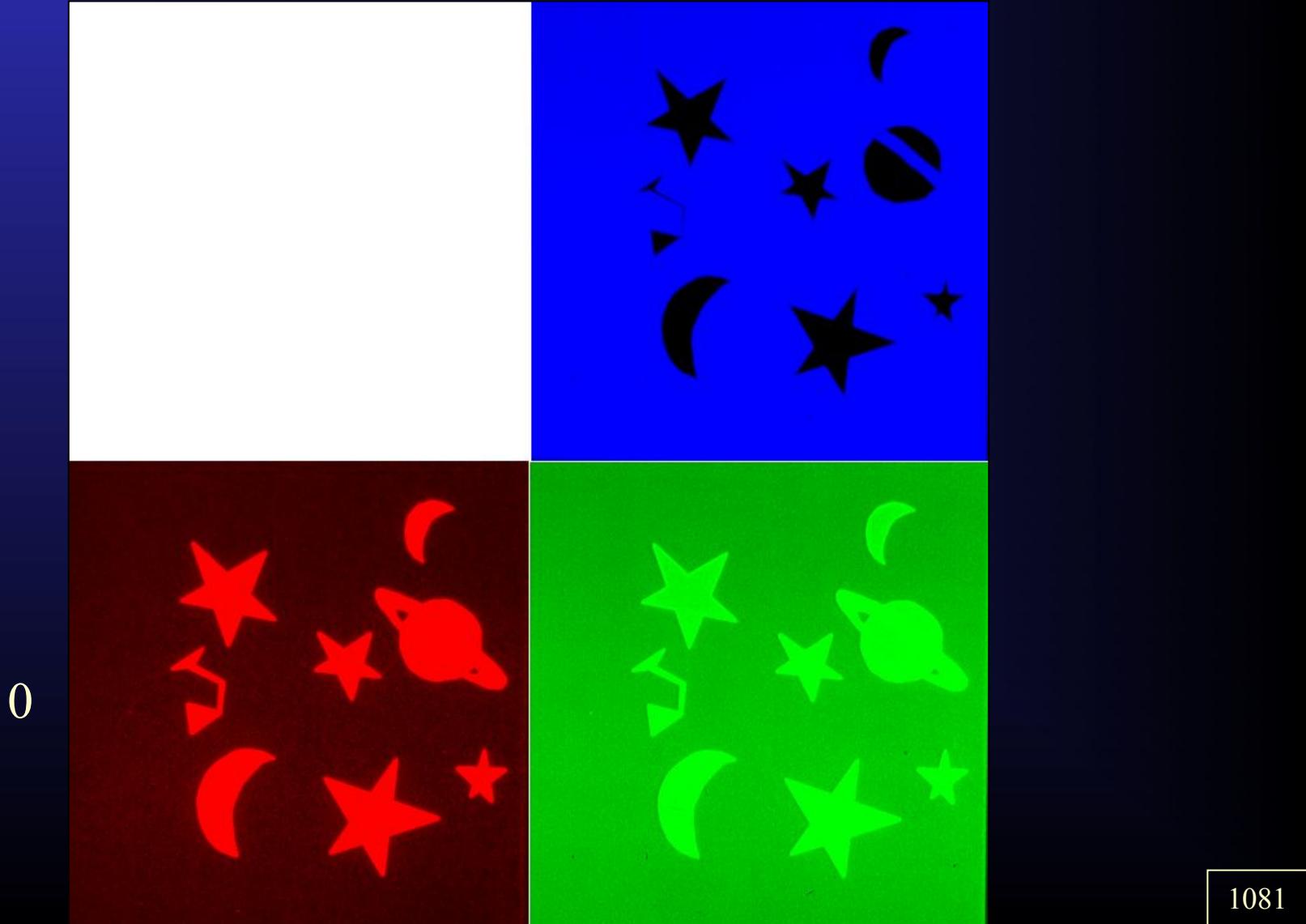


Blue

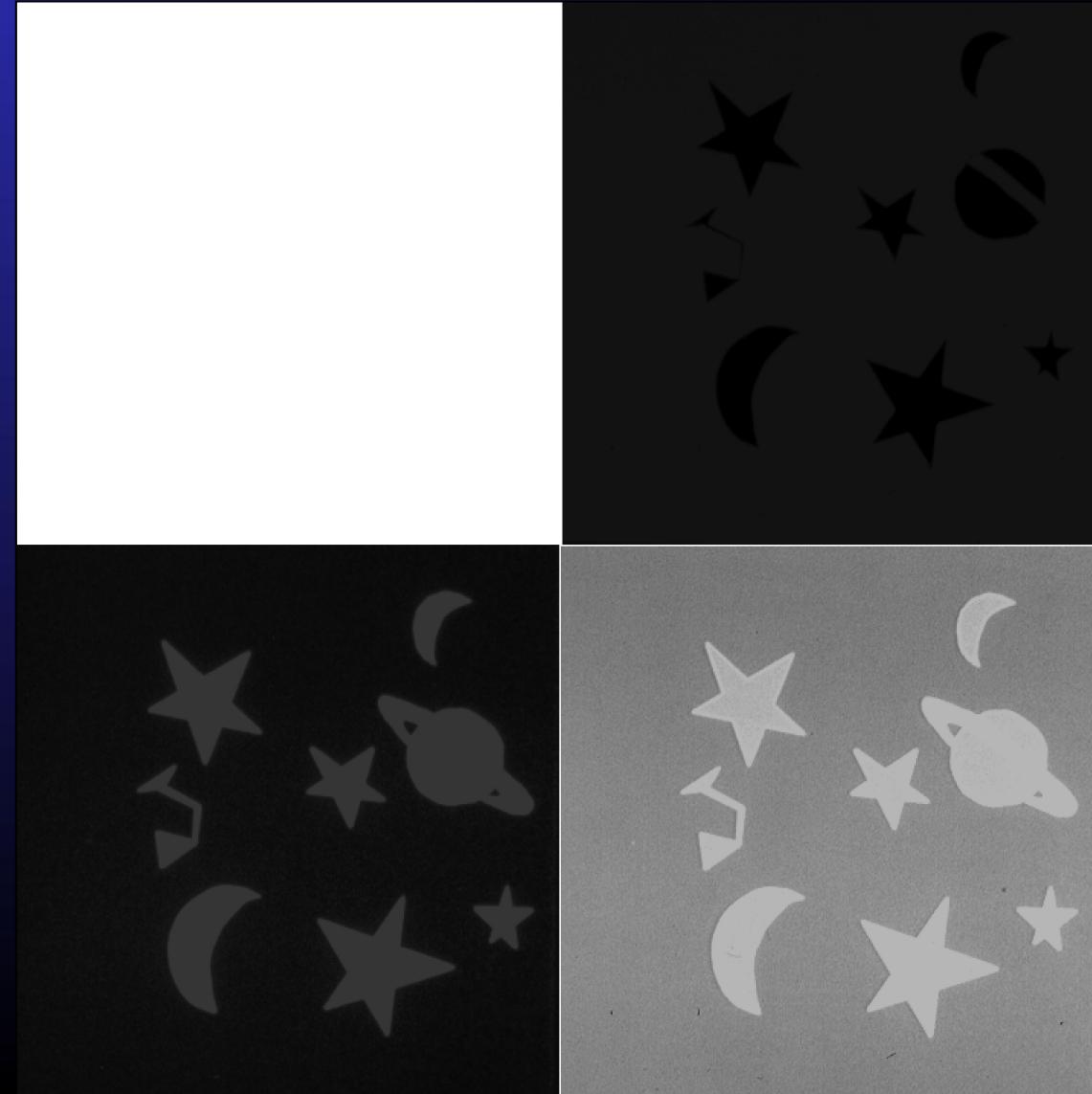


But for easier visualization, we'll just show the RGB color instead.

Create 0th level of Pyramid with RGB Separations

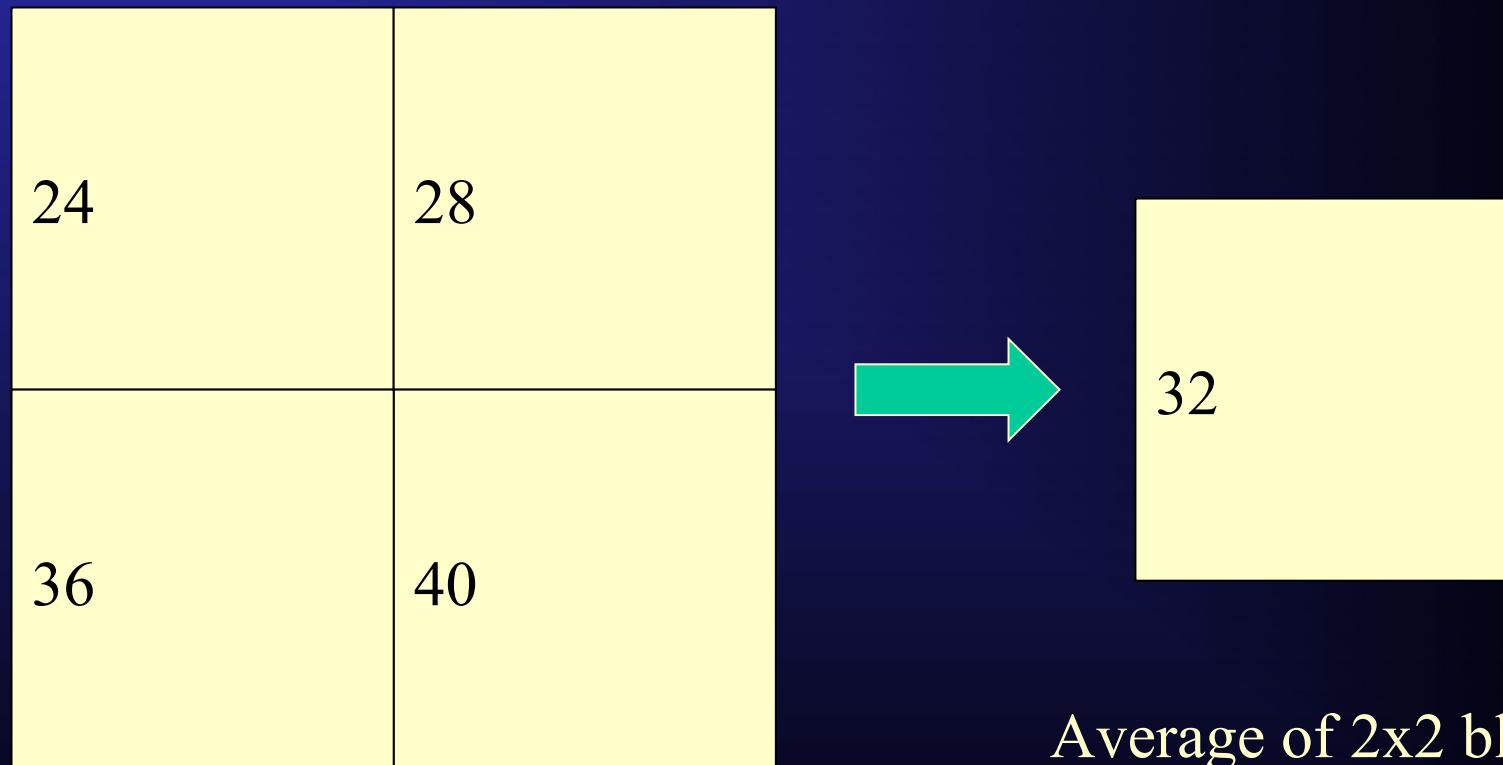


Create 0th level of Pyramid with RGB Separations
(Remember, it's really 1 byte per texel!:)

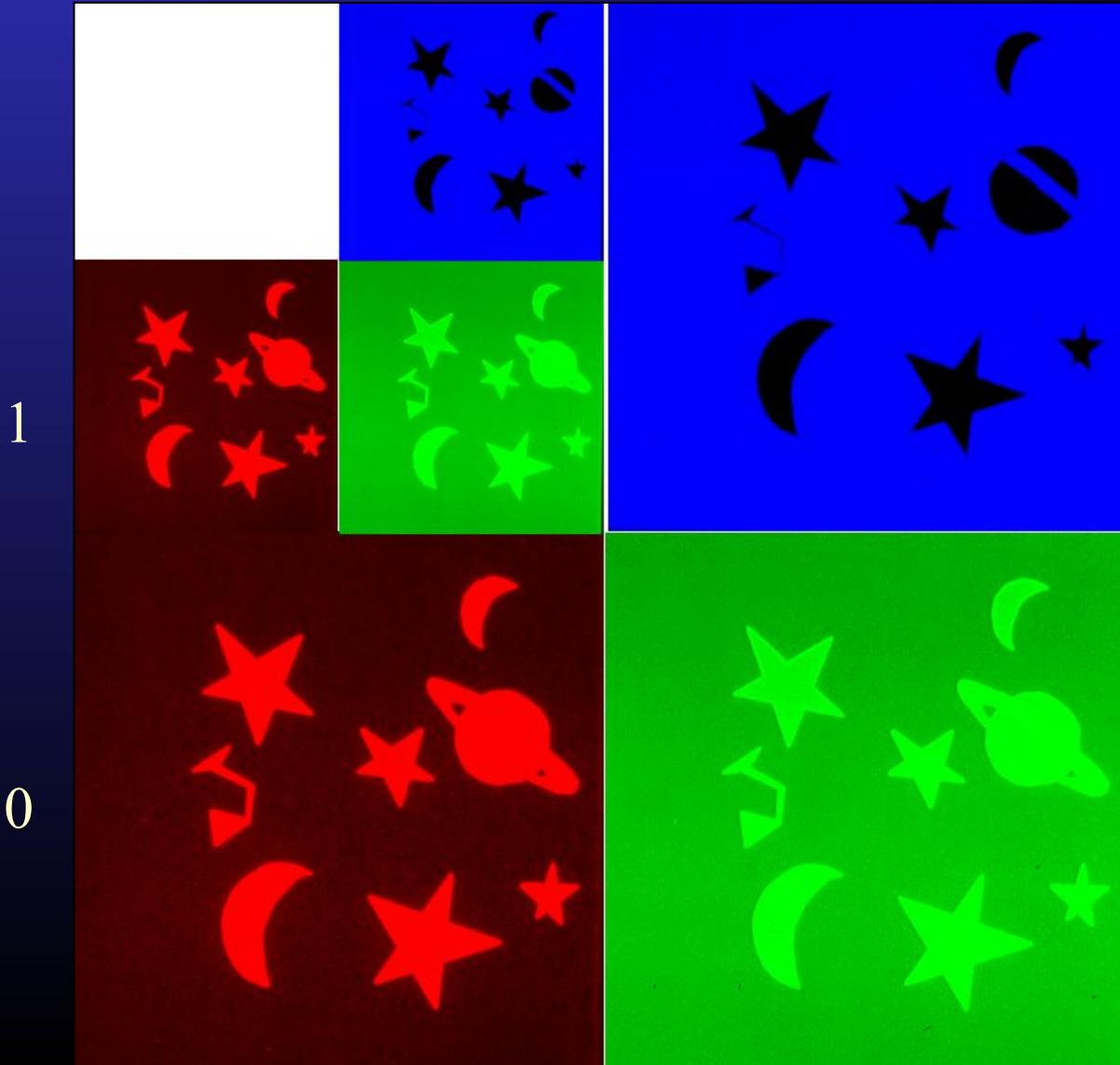


Average 2x2 Texel Blocks to Create Lower Resolution Texture

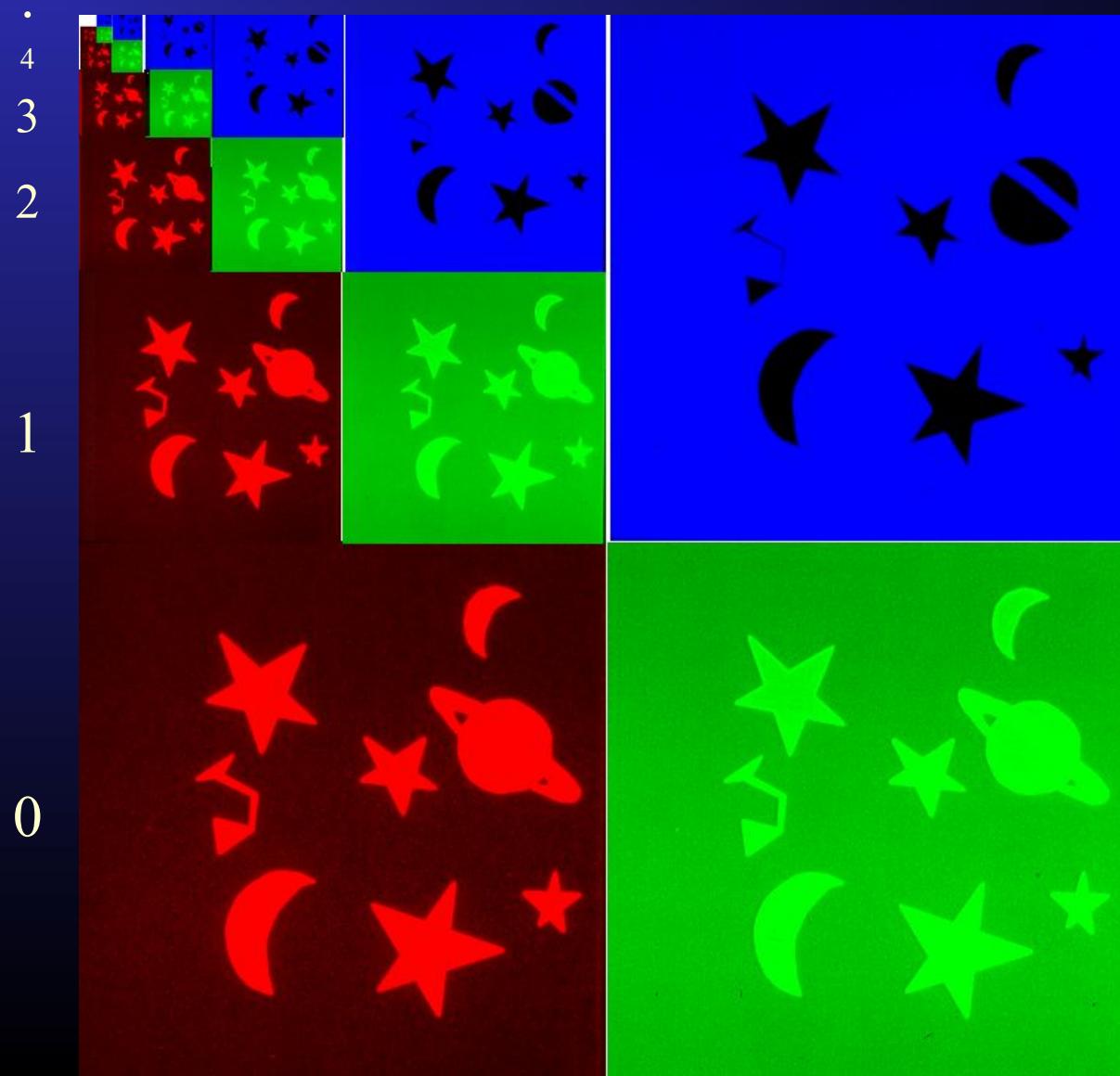
Each texel is 8 bits of grayscale



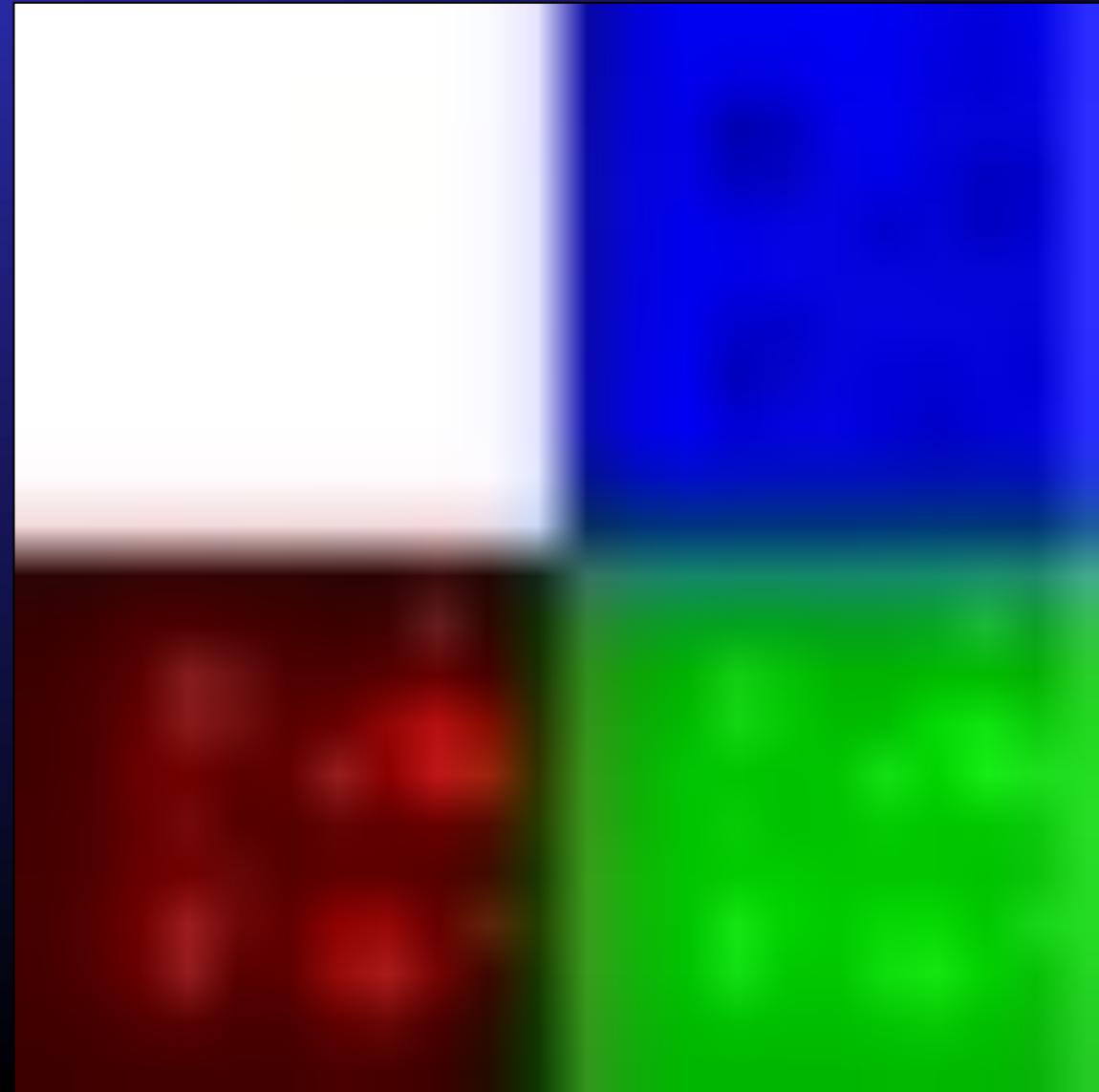
Create 1st Level of Pyramid with RGB Separations/2



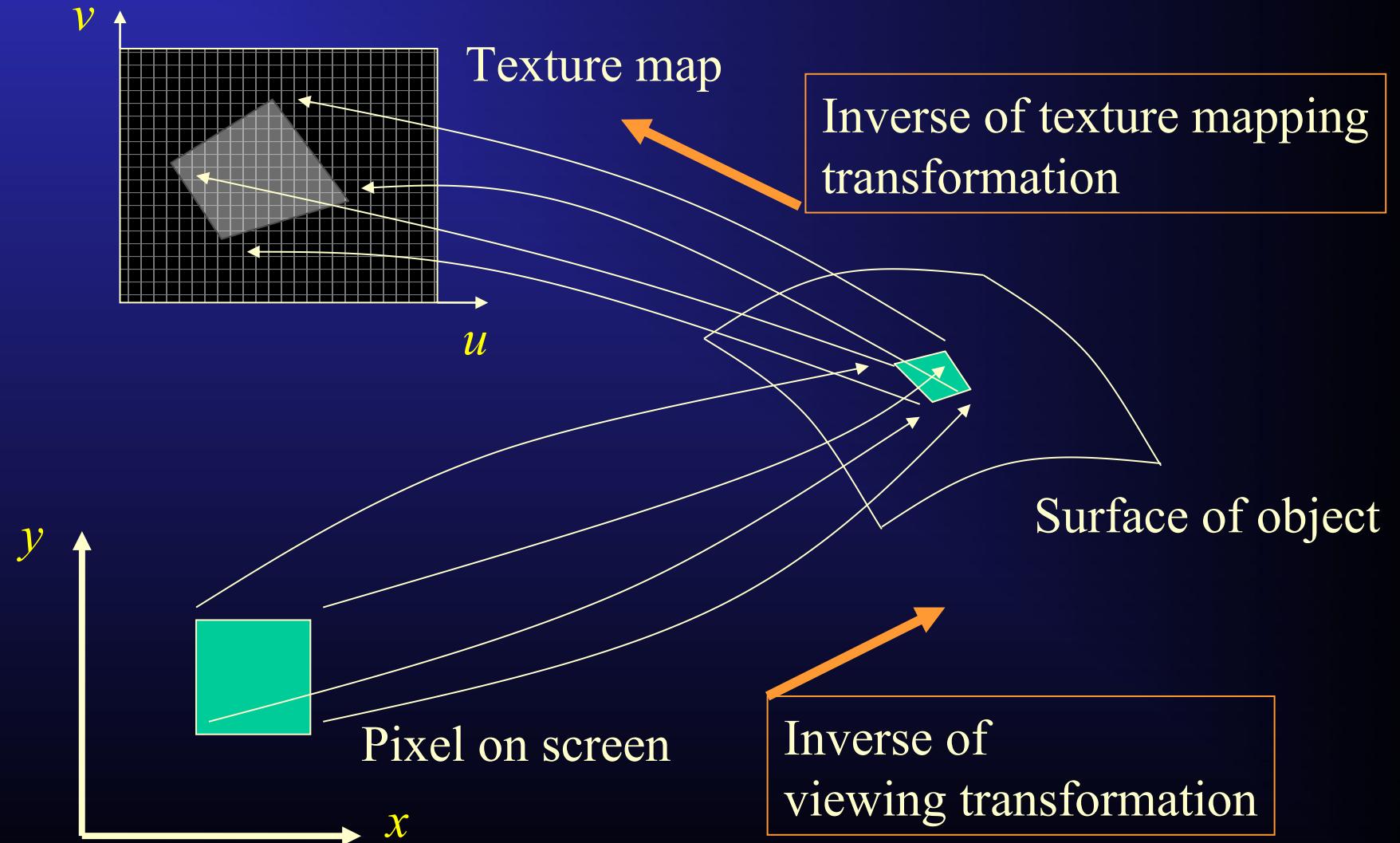
Continue until n^{th} Level, Dividing by 2^n :
This is now the prefiltered stored texture map



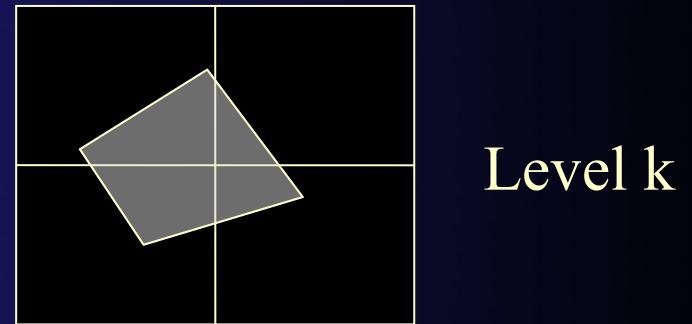
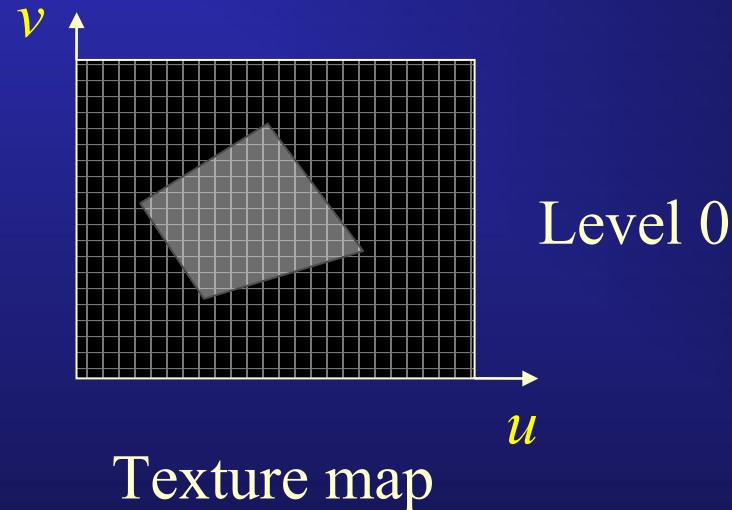
Close-up of Low Resolution Texels at 5th Level



Recall: Texture Mapping Transformations



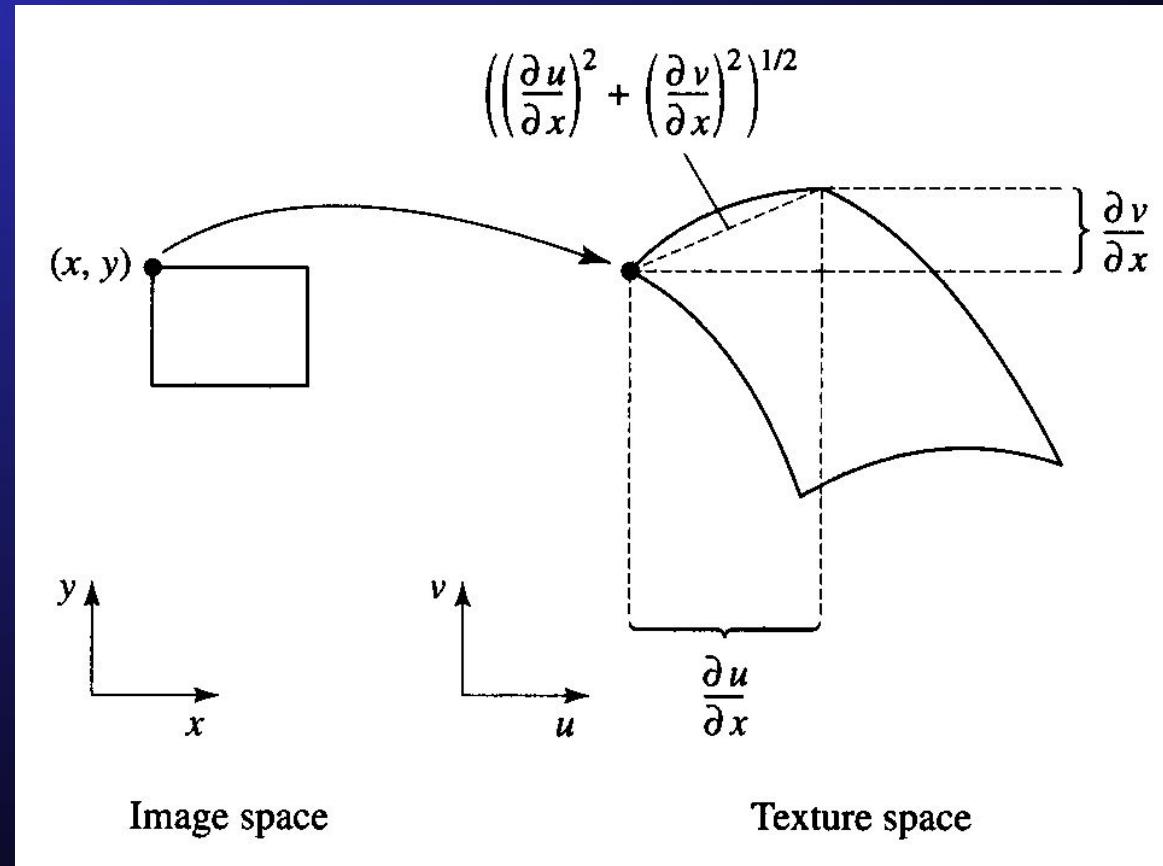
Computing the Resolution Level for Pixel Pre-Image



Since the pixel pre-image in the texture space covers some area, choose the pyramid resolution which is closest to mapping the pre-image into a *single* texel.

This is the color given to the reflectance function of the surface at that pixel.

Bi-Linear Texture Map Interpolation



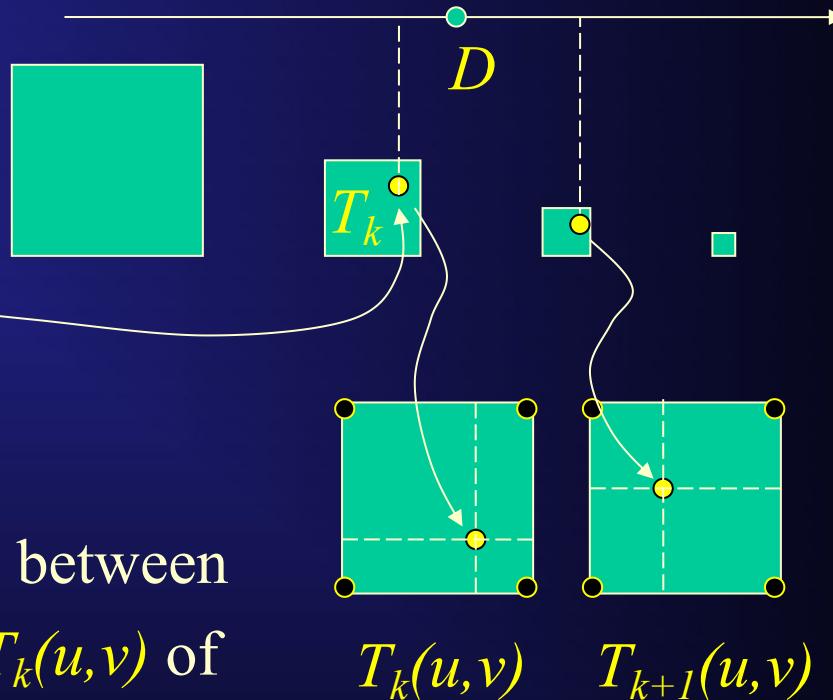
Policarpo
and Watt

For a square pixel , $\partial x = \partial y = 1$;

D = maximum of the 4 possible lengths from each pixel boundary.

Tri-Linear Texture Map Interpolation

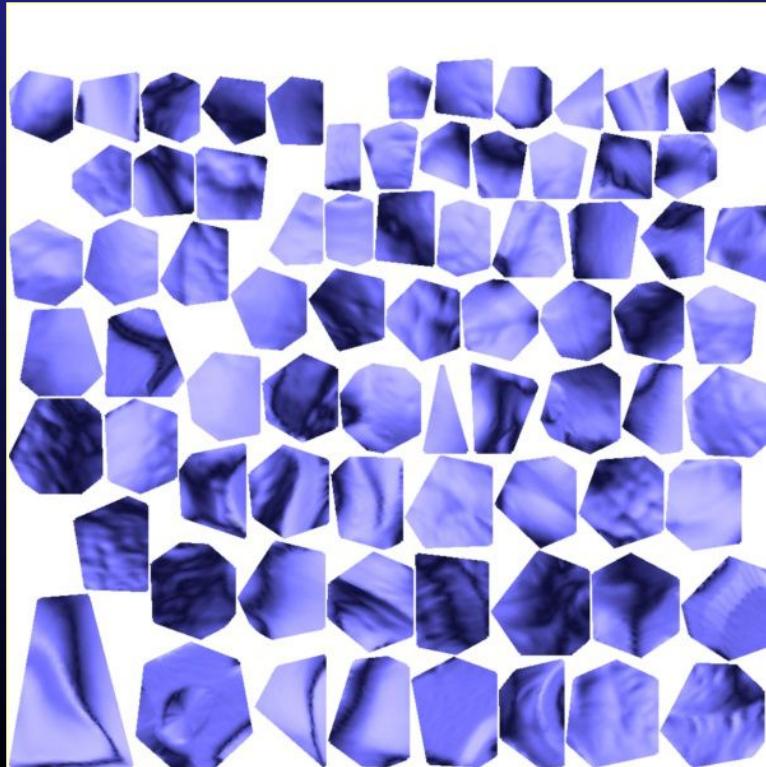
- Calculate D



- Calculate $T_k(u, v)$
- Bi-linearly interpolate between four texels nearest to $T_k(u, v)$ of pixel pre-image.
- Repeat for the next coarser level T_{k+1} of the texture mip-map and linearly interpolate the two values.

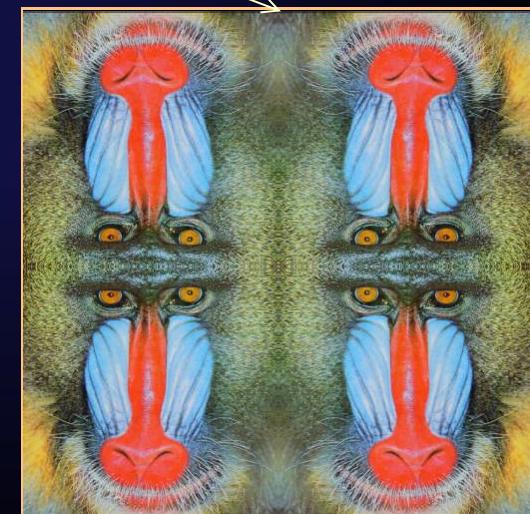
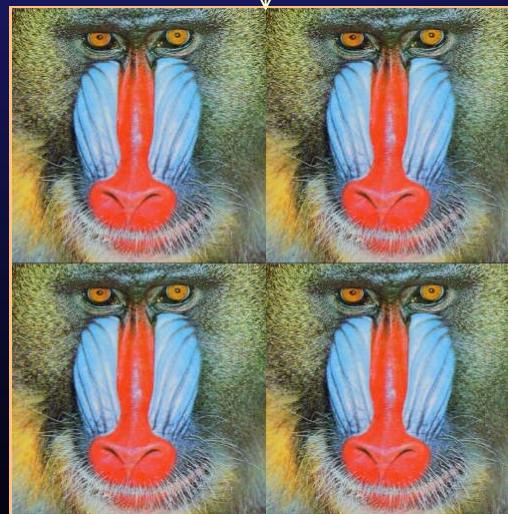
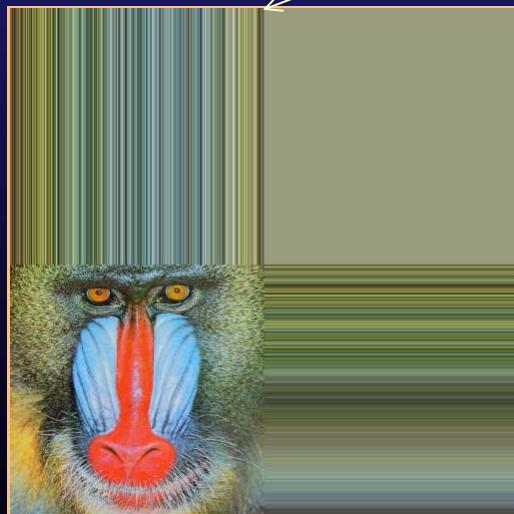
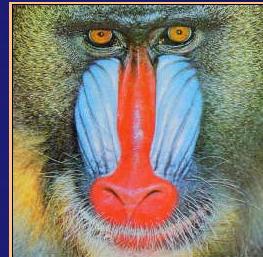
Avoiding Texture Atlas Sampling Artifacts

- Averaging values in texture atlases:
- To shade properly must “fill in” gaps between charts with interpolated values:



Texture Edge Options

- Clamp to edge values
- Repeat
- Mirror



Sampling and Advanced Rendering Methods

- We will return to sampling methods after discussing advanced rendering methods, to motivate using even better techniques.
- These rendering methods, such as Monte Carlo path tracing or photon mapping, all depend on making good choices for generating rays.
- “Making good choices” is equivalent to sampling rays to generate illumination, shadows, indirect lighting, caustics, and so on, while minimizing image artifacts such as sample clumping, noise, and blur.

Compositing Images

- Divide image into separately created or rendered components or layers.
- Simplify image generation (divide and conquer in software).
- Different kinds of objects can be separately processed then combined.
- Each component has an associated **matte**: an area containing any pixel touched by the component.
- Compositing must not introduce aliasing errors.
- Compositing allows for post-process effects such as dissolves and fades.
- Additional memory required for storing mattes -- often augmented with stencil buffers.

The Alpha Channel

- Stores matte (coverage or “mixing”) information.
- Mixing value cannot be directly stored into element color (R , G , B) -- needs additional bit planes called α .
- α plane interpretation:
 - $\alpha = 0 \Rightarrow$ no coverage
 - $\alpha = 1 \Rightarrow$ full coverage
 - $\alpha \in (0, 1) \Rightarrow$ partial coverage (percentage) [recall anti-aliasing prefiltering!]
- Pixel stored is thus (R, G, B, α)
- Interpretation: Pixel is α covered by the color
$$(R / \alpha, G / \alpha, B / \alpha)$$
- Thus a pixel half covered by a full red object is stored as
$$(0.5, 0, 0, 0.5)$$

Alpha Channel Properties

- $\alpha = 0$ usually forces color components to be 0.
- Anti-aliased edges are preserved.
- Black = $(0, 0, 0, 1)$ [opaque]
- Clear = $(0, 0, 0, 0)$ [transparent]
- Pre-multiplication in the alpha channel of (R, G, B) by α means that standard hardware interpretation of pixel color can be used.
- There is a compositing algebra to describe how two elements may combine at a single pixel.
- The α channel model assumes that the shape of the matte coverage at a pixel is not significant, only the sub-pixel coverage percentage. Relative depths are accommodated.
- Acts like pixel-wise fog or haze.

Compositing Combinations at a Pixel

- Consider 2 pictures A and B. They divide each pixel into 4 sub-pixel areas.
- \underline{A} means the complement of A.
- Based on the haze assumption, the area covered by the composite is defined by the following table:

B	A	name	description	who can contribute to composite	coverage
0	0	0	$\underline{A} \cap \underline{B}$	0	$(1-\alpha_A)(1-\alpha_B)$
0	1	A	$A \cap \underline{B}$	0, A	$\alpha_A(1-\alpha_B)$
1	0	B	$\underline{A} \cap B$	0, B	$(1-\alpha_A)\alpha_B$
1	1	AB	$A \cap B$	0, A, B	$\alpha_A\alpha_B$

Compositing Algebra

Operations compositing 2 pictures A and B:

operation	$(0, A, B, AB)$	diagram	coverage _A	coverage _B
clear	$(0, 0, 0, 0)$		0	0
A	$(0, A, 0, A)$		1	0
B	$(0, 0, B, B)$		0	1
A over B	$(0, A, B, A)$		1	$1 - \alpha_A$
A in B	$(0, 0, 0, A)$		α_B	0
A out B	$(0, A, 0, 0)$		$1 - \alpha_B$	0
A atop B	$(0, 0, B, A)$		α_B	$1 - \alpha_A$
A xor B	$(0, A, B, 0)$		$1 - \alpha_B$	$1 - \alpha_A$

Color of result is = $(R_A, G_A, B_A) \cdot \text{coverage}_A + (R_B, G_B, B_B) \cdot \text{coverage}_B$

Example

- Areas A and B may overlap within a pixel to form 4 regions:
 $AB=(\text{in A, in B})$, $A=(\text{in A, out B})$, $B=(\text{out A, in B})$, $0=(\text{out A, out B})$.
- Suppose A covers the pixel 0.3 and B covers it 0.6. In the area outside of B ($1-0.6$), A's coverage is $0.3(0.4) = 0.12$. Symmetrically for B outside of A, B's coverage is $0.6(1-0.3) = 0.42$. The area not covered by either A or B = $(1-0.3)(1-0.6) = 0.28$. The area covered by BOTH A and B is the product $(0.3)(0.6) = 0.18$. (If you add up the 4 coverages: $(0.12+0.42+0.18+0.28) = 1$.)
- Suppose A is 30% red (0.3, 0, 0, 0.3) and B is 60% blue (0, 0, 0.6, 0.6). Use the second chart to find the color:
- If the operation is A OVER B, e.g., then the color is:

$$\begin{aligned} & A * \text{coverage A} + \text{color B} * \text{coverage B} \\ &= (0.3, 0, 0)*1 + (0, 0, 0.6)(1-0.3) \\ &= (0.3, 0, 0) + (0, 0, 0.6)0.7 \\ &= (0.3, 0, 0) + (0, 0, 0.42) \\ &= (0.3, 0, 0.42) \end{aligned}$$

which is sensible as it has all of A's red but less of B's blue.

Road to Pt. Reyes



© 1983 COOK, R.—LUCASFILM LTD.

Other Compositing Operations

- Darken $(A, \phi) \equiv (\phi R_A, \phi G_A, \phi B_A, \alpha_A)$

$$\phi \begin{cases} 0 & \text{black} \\ 1 & \text{normal} \\ >1 & \text{brighter} \end{cases}$$

- Dissolve $(A, \delta) \equiv (\delta R_A, \delta G_A, \delta B_A, \delta \alpha_A)$

$$\delta \begin{cases} 0 & \text{disappear} \\ & \quad \text{fade} \\ 1 & \text{normal} \end{cases}$$

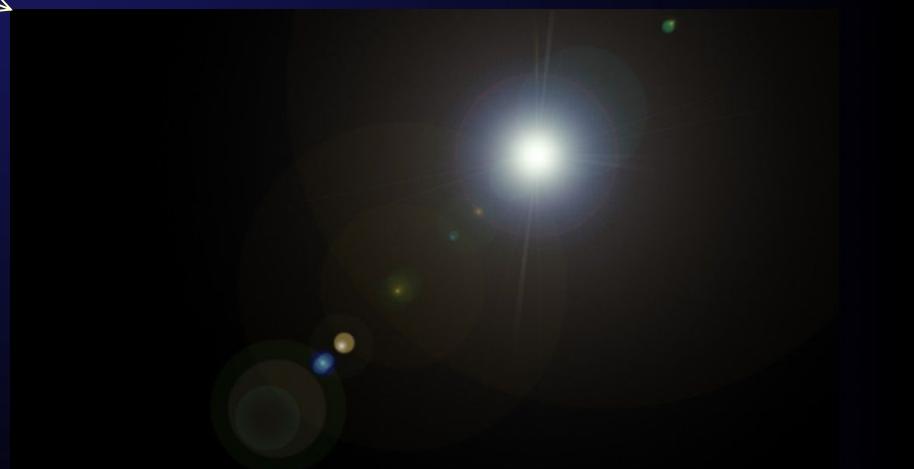
- Opaque $(A, \omega) \equiv (R_A, G_A, B_A, \omega \alpha_A) \quad 0 \leq \omega \leq 1$

$$\omega \begin{cases} 1 & \text{normal} \\ 0 < \omega < 1 & \text{less obscuration -- clip colors to [0,1]} \\ 0 & \text{obscures nothing -- lets colors through} \end{cases}$$

for translucent objects

A Compositing Example from the Movies

- “SHAKE” software
- Tutorial by Tahl Niran [MIA]
- Acted scene 
- Background plate (fixed)
- Lens flare effect
- Compositing narrative:
 - Note mattes
 - Note compositing algebra (tree structure)
 - Animated mattes
 - “Fix” lighting
 - Add lens flare and adjust



Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading ✓
- Mapping ✓
- 3D Viewing Transformations ✓
- Anti-aliasing & Compositing ✓
- **Global Illumination**
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

Global Illumination

- We saw earlier that **RAY TRACING** is a method to establish the shade (color) at a model surface point by following rays from a model surface point into scene. Easy to do if you know (compute) surface normals.
 - Strength: specular effects and computational simplicity.
 - Weakness: diffuse lighting, and in general lack of accommodation of all light affecting that surface point since it depends on bouncing and bending a finite tree of rays.
- To take into account **all** light that might influence the shading at a model surface point we use **GLOBAL illumination methods**.
- The one we'll talk about next is **RADIOSITY**. After that discussion, we'll look the the “ultimate” rendering methods, **MONTE CARLO PATH TRACING** and **PHOTON MAPPING**.

Significant Rendering Milestones

Notation:

L = Light; D = Diffuse surface; S = Specular surface; E = Eye

- Kajiya's Rendering Equation
- Whitted's Ray Tracing – $LD?S^*E$
- Cook's Distributed Ray Tracing
- Finite Element (Radiosity) Methods – LD^*E
- Monte Carlo Path Tracing – $\underline{L(S|D)}^*E$
- Lafortune and Veach's Bi-directional Path Tracing – $LS+DE$
- Veach's Metropolis Light Transport – $\underline{L(S|D)}^*E$
- Jenson's Photon Mapping – $\underline{L(S|D)}^*E$

Thanks to Ghulam Lashari, Pharr and Humphreys' book,
and various Internet sources for much of this material.

Global Illumination Overview

- Radiosity and the finite element method
- The rendering equation (Light transport equation)
- Light fields
- High dynamic range imagery and tone mapping
- Monte Carlo path tracing
- Importance sampling
- Photon mapping
- Comparison of rendering techniques

Radiosity

- Ray-tracing gives “hard-edge” and specular effects easily.
- Radiosity or global illumination does better diffuse lighting.
- Radiosity eliminates arbitrary ambient light term.
- Surface radiosities, once computed, become attributes of surface “patches” (actually small planar polygons). This is sometimes called a “finite element method”.
- The shade of each colored patch is view independent: a patch can be rendered without re-computing its shade, assuming that nothing that reflects light moves.

Radiosity “Finite Element” Patches

- Must divide the environment into small enough planar patches so that radiosity does not vary too much across the patch (otherwise causes color contouring artifacts) .
- Surface patches can be rendered directly by (real-time) z-buffer algorithm.
- The downsides to this are:
 - Increased number of surface primitives, even in large flat areas.
 - Patch decomposition actually depends on lighting!

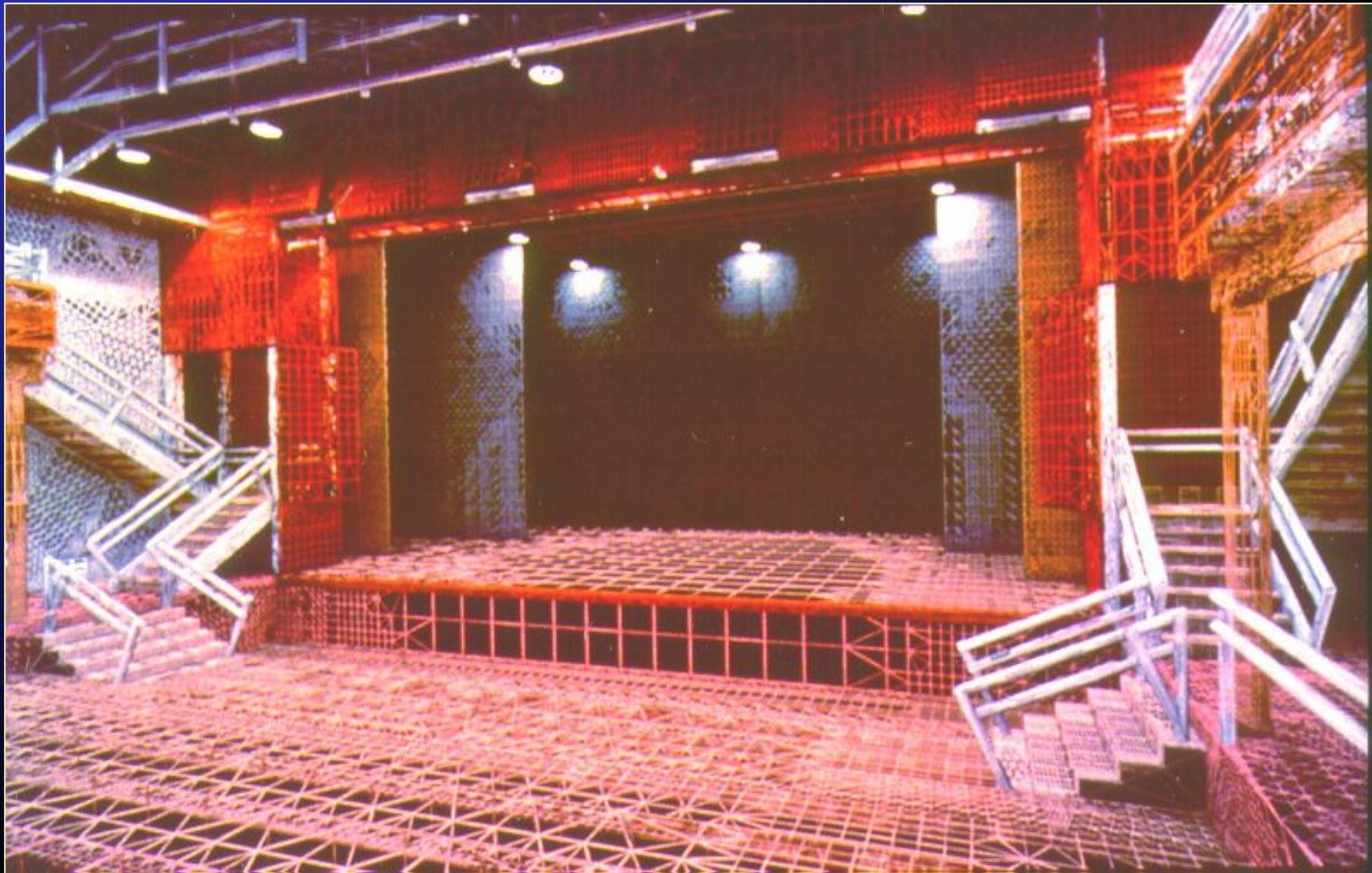
Stage Scene



Cornell

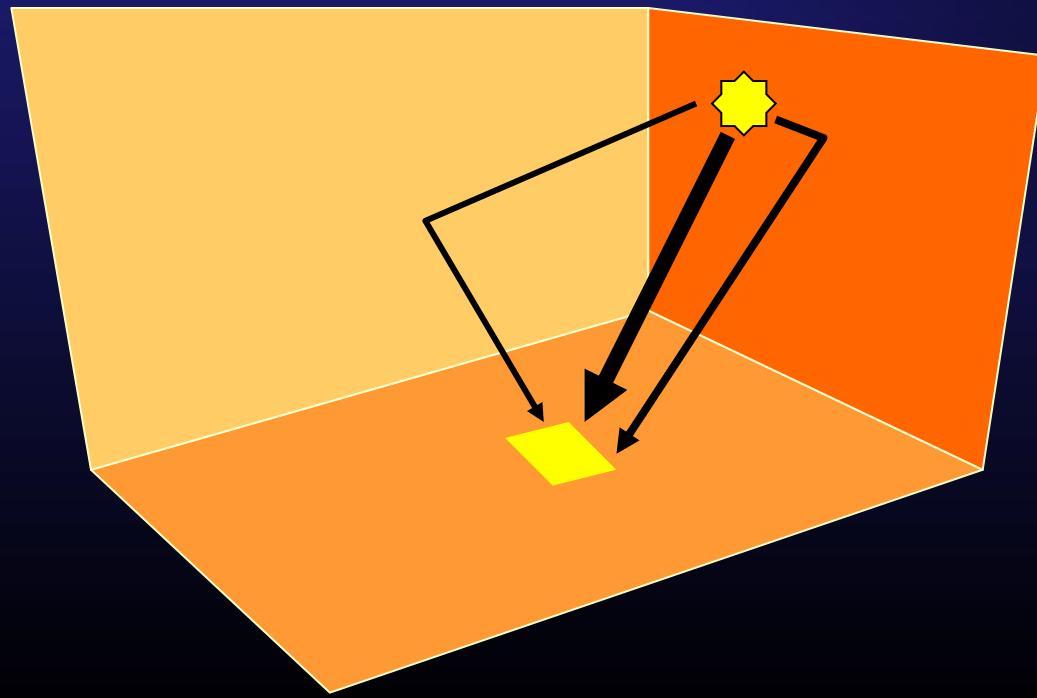
1109

Stage Scene Patches



Basic Principle of Radiosity Global Illumination

- Light reaching a surface patch gets there by
 - Direct route from any emitter directly visible to it.
 - Indirect routes from reflection (one or more times) from every other surface patch visible to it.



Ambient Light and Ray-Tracing

Scanline or z-buffer rendering;
constant ambient light value.



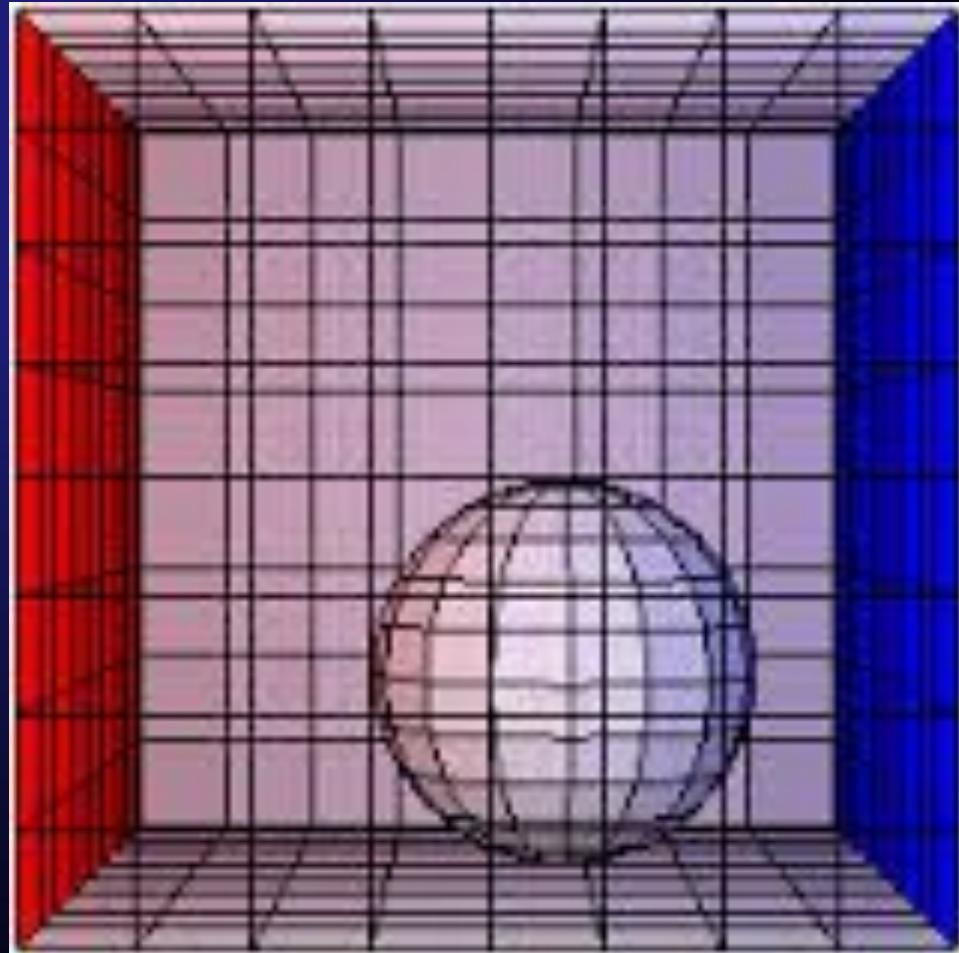
Ray trace rendering;
0 ambient light value.



Radiosity rendering;
ambient term not needed.

“Cornell Box”: Used for Closed Environment Illumination Studies

- Mesh closed cubical space into rectangular patches.
- Assign colors to (wall/floor/etc.) patches.
- Assign lights to patches.
- Insert objects as desired.



Diffuse Reflection and Color Bleeding

Example:

- Blue, red and white walls; lights on ceiling; yellow box.
- Note color bleeding:
 - blue and red onto white walls and into shadows;
 - red and blue onto yellow box.
- Get “soft shadows” for free.



Radiosity – Basic Definitions

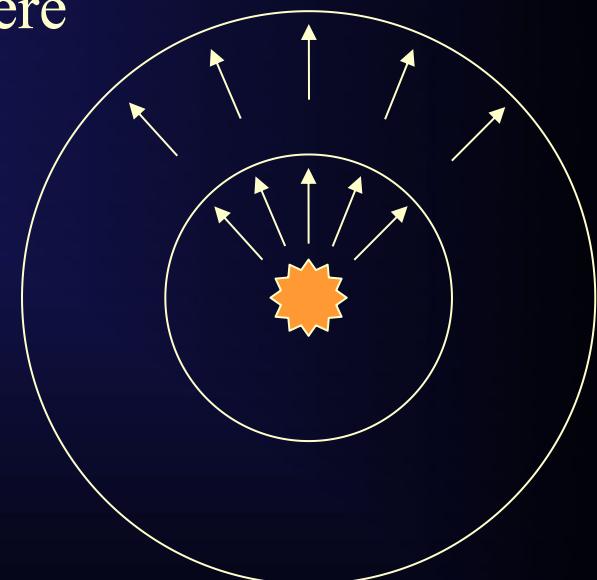
- Radiosity is the rate at which energy leaves a surface: energy per unit time (power) per unit area in watts/m².
 - Energy can be emitted (by an emitter).
 - Energy can be reflected (by a reflector).
- Irradiance is power arriving at a surface.
- Exitant radiosity leaving a surface is used to calculate shading.
- Note that radiosity is an unbounded real number.
- Radiosity may be discretized by wavelength bins.
- Equations describing radiosity in an enclosed environment originated in heat transfer analyses.
- Computer graphics revised and applied them to visible energy (illumination); and also made them computationally more efficient.

Radiant Flux

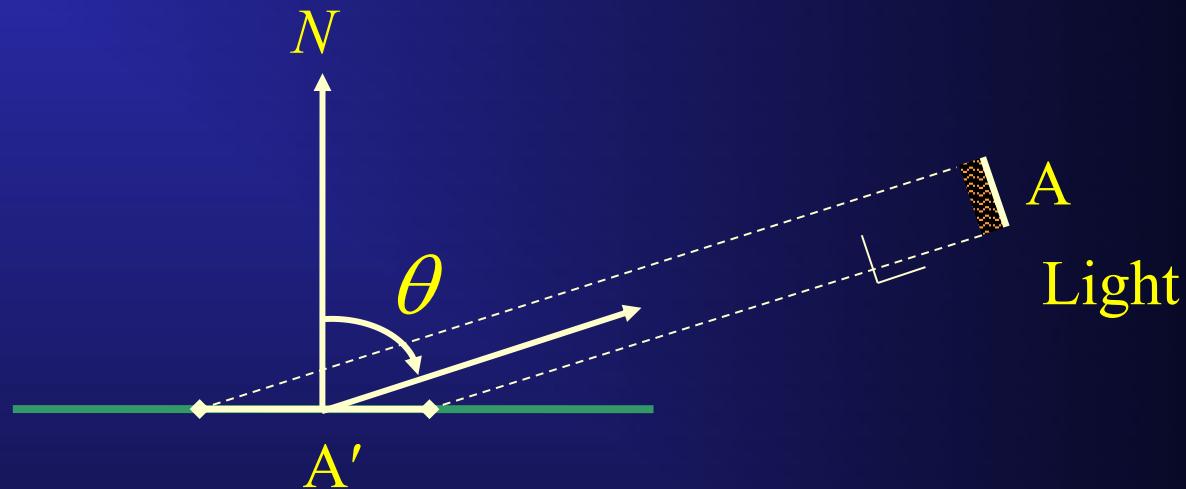
- Flux Φ is energy through a region of space.
- Irradiance E at a point on the outer sphere is less than irradiance at a point on the inner sphere, as the surface area of the outer is larger than the inner.
- Hence, irradiance is flux spread over sphere surface area ($4\pi r^2$):

$$E = \frac{\Phi}{4\pi r^2}$$

- Note fall-off as the radius squared!



Connection to Lambert's Law

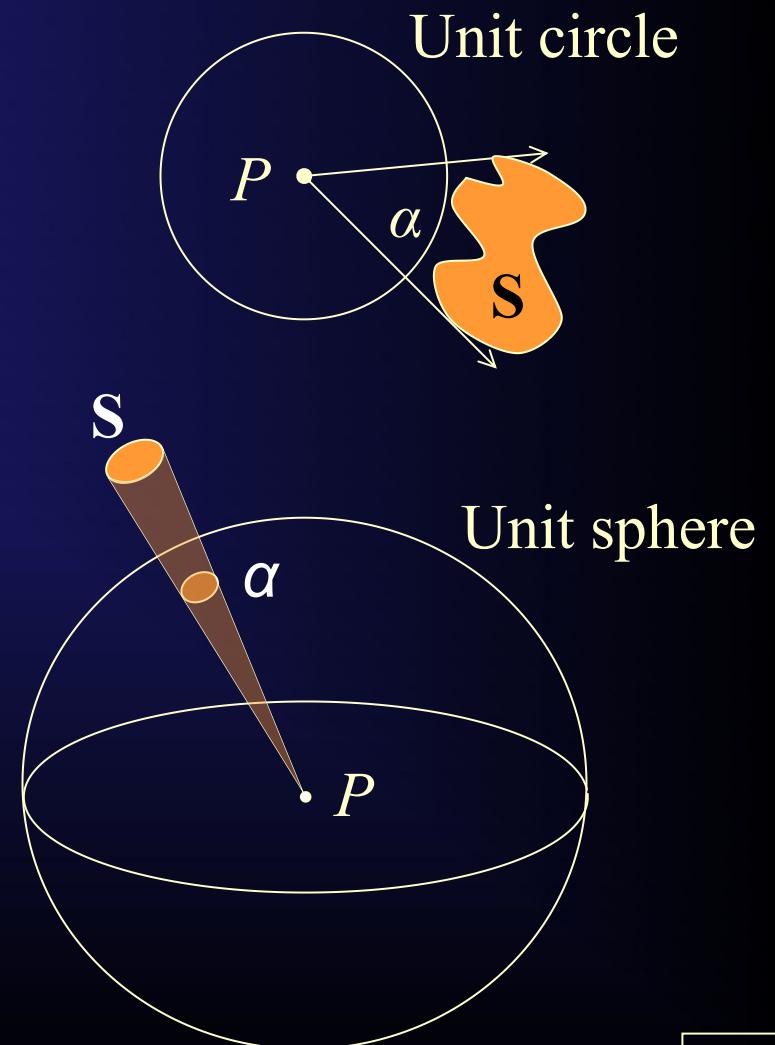


- Irradiance E from A arriving at a surface area A' varies as the cosine of the incidence angle.
- Within A' , flux is divided by the surface area of $A' = A / \cos \theta$, or

$$E = \frac{\Phi}{(A / \cos \theta)} = \frac{\Phi \cos \theta}{A}$$

Solid Angles

- Planar angle $\alpha = \text{arclength of projection of the object } S \text{ (area of unit circle it subtends) in radians.}$
- Maximum arclength is 2π .
- Extend to 3D unit sphere: total area subtended by S is α *steradians*.
- Maximum area is complete unit sphere surface = 4π steradians.
- Surface area of the 3D unit hemisphere H^2 is 2π .

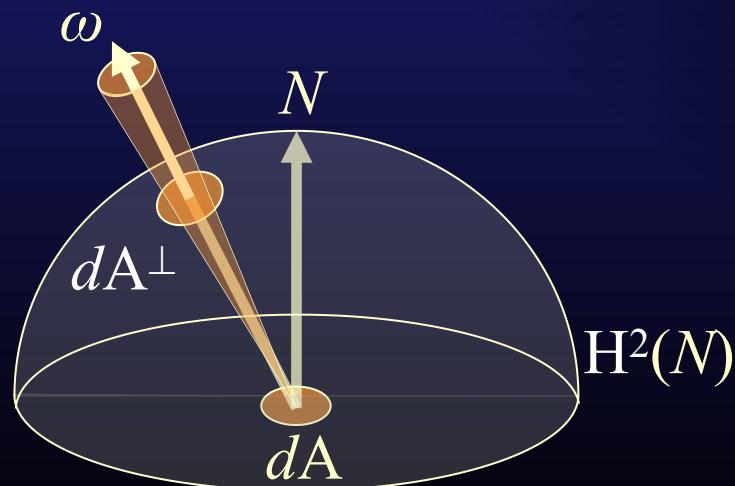


Radiance

- All points on the unit sphere centered at P describe vectors from P , and will be denoted by normalized vector direction ω .
- Flux density per unit area, per unit solid angle:

$$L = \frac{d\Phi}{d\omega dA^\perp}$$

- where dA^\perp is the projected area of dA on a hypothetical surface perpendicular to vector ω .



Incident and Exitant Radiance

- Incident radiance function $L_i(P, \omega)$ is the distribution of radiance toward the surface as a function of position P and direction ω .
- Exitant radiance function $L_o(P, \omega)$ is the distribution of radiance leaving the surface as a function of position P and direction ω .
- Vector ω is oriented away from the surface.
- In a vacuum $L_o(P, \omega) = L_i(P, -\omega)$.
- In general, $L_i(P, -\omega) \neq L_o(P, \omega)$



The Radiosity Equation

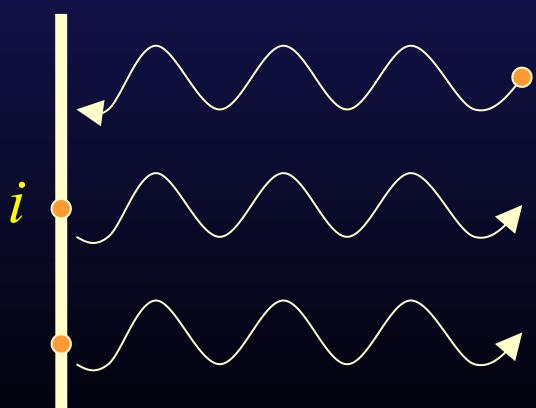
$$B_i = E_i + \rho_i \sum F_{ij} B_j \quad (\text{sum over all patches } j)$$

B_i, B_j = radiosity of surface i and j , respectively

E_i = emissivity of surface i

ρ_i = reflectivity of surface i

F_{ij} = form factor of surface j relative to surface i



$\sum F_{ij} B_j$ (Energy reaching this surface from other surfaces)

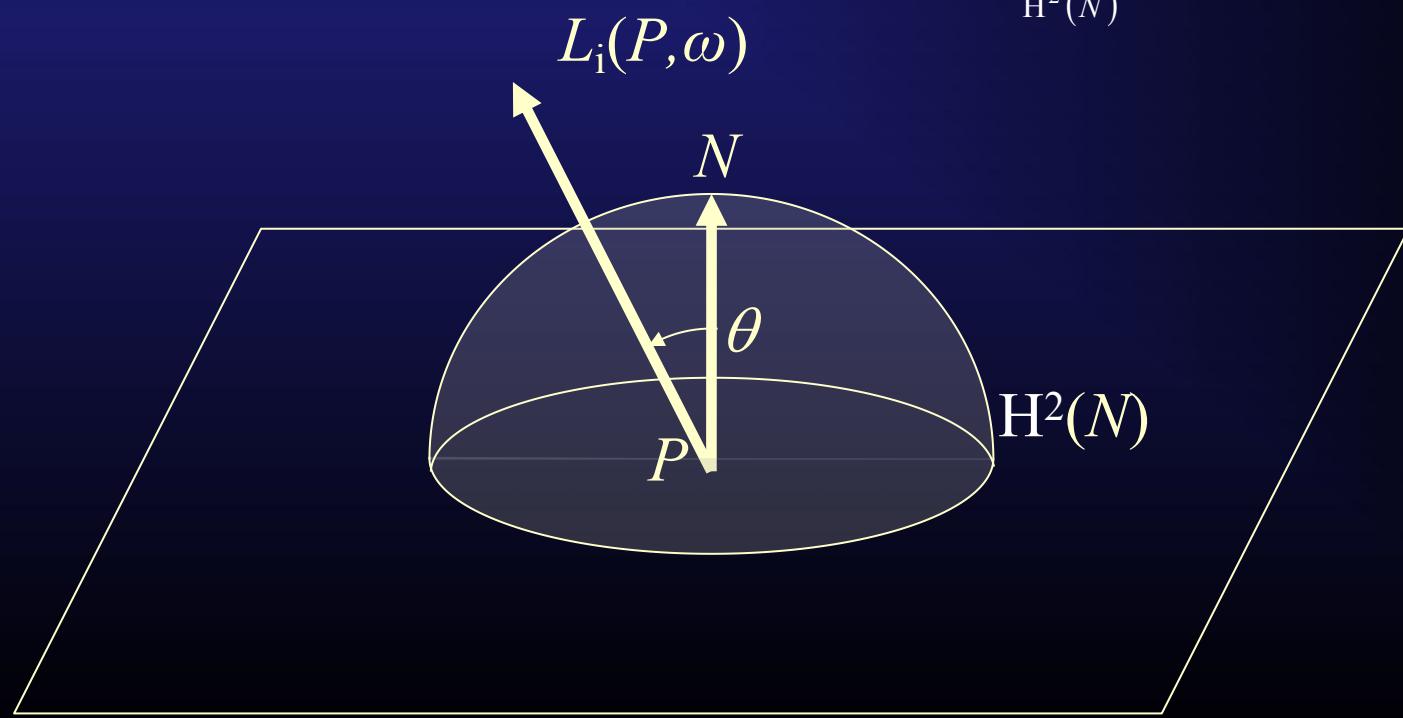
E_i (Energy emitted by this surface)

$\rho_i \sum F_{ij} B_j$ (Energy reflected from this surface)

Irradiance at a Point

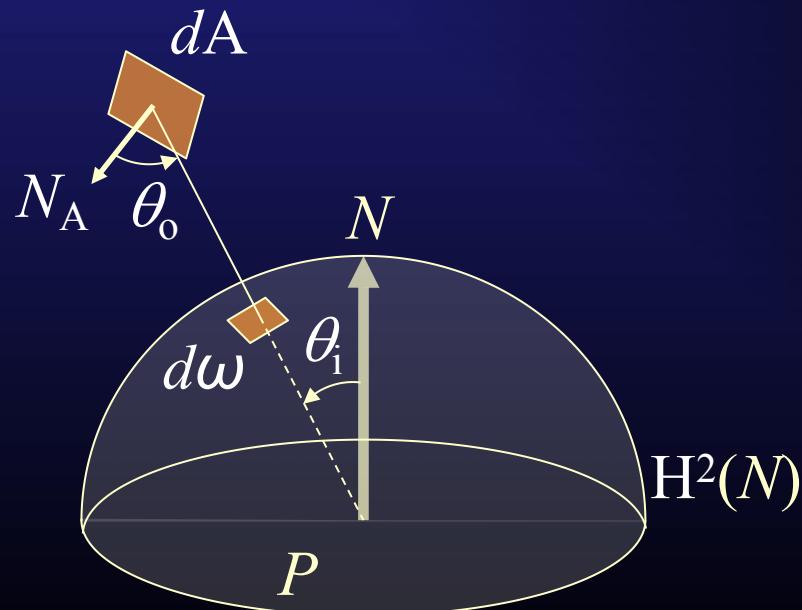
- Irradiance at point P is computed from incoming radiance $L_i(P, \omega)$ over the entire upper hemisphere $H^2(N)$ (i.e., over all vectors ω in $H^2(N)$) about P .
- Radiance must be multiplied by cosine of incident angle (like Lambert's law):

$$E(P, N) = \int_{H^2(N)} L_i(P, \omega) |\cos \theta| d\omega$$



Incident Radiance from a Quadrilateral Patch

- Differential area is related to differential solid angle from point P via $d\omega = \frac{dA \cos \theta}{r^2}$
- So $E(P, N) = \int_A L \cos \theta_i \frac{\cos \theta_o}{r^2} dA$



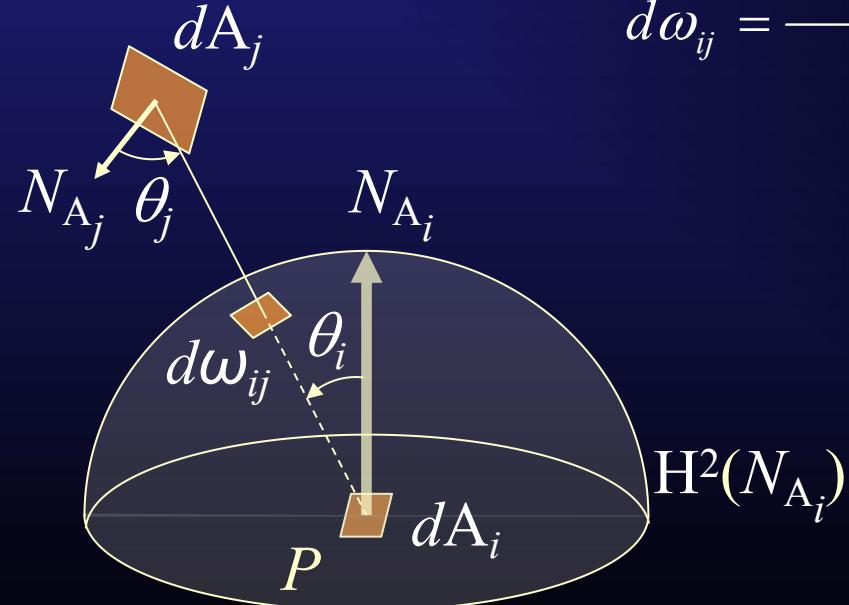
Incident Radiance from a Quadrilateral Patch to a Quadrilateral Patch

- From

$$d\omega = \frac{dA \cos \theta}{r^2}$$

- The $2\pi(1^2)$ surface area of $H^2(N_{A_i})$ “covered” by dA_j is:

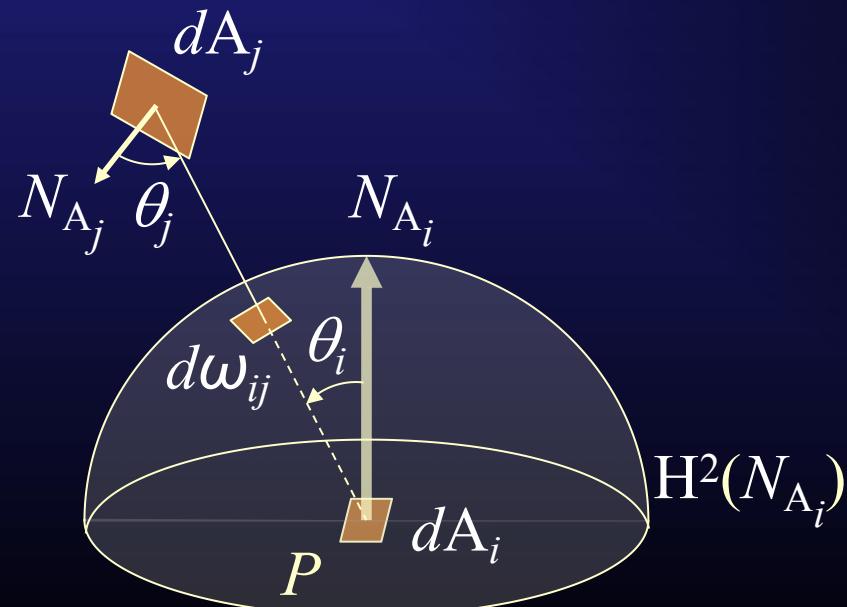
$$d\omega_{ij} = \frac{dA_j \cos \theta_j}{2\pi r^2}$$



Incident Radiance from a Quadrilateral Patch to a Quadrilateral Patch

- But by Lambert's Law, the fraction of light leaving dA_j that would reach dA_i is $\cos \theta_j$, so $d\omega_{ij}$ must be multiplied by this factor:

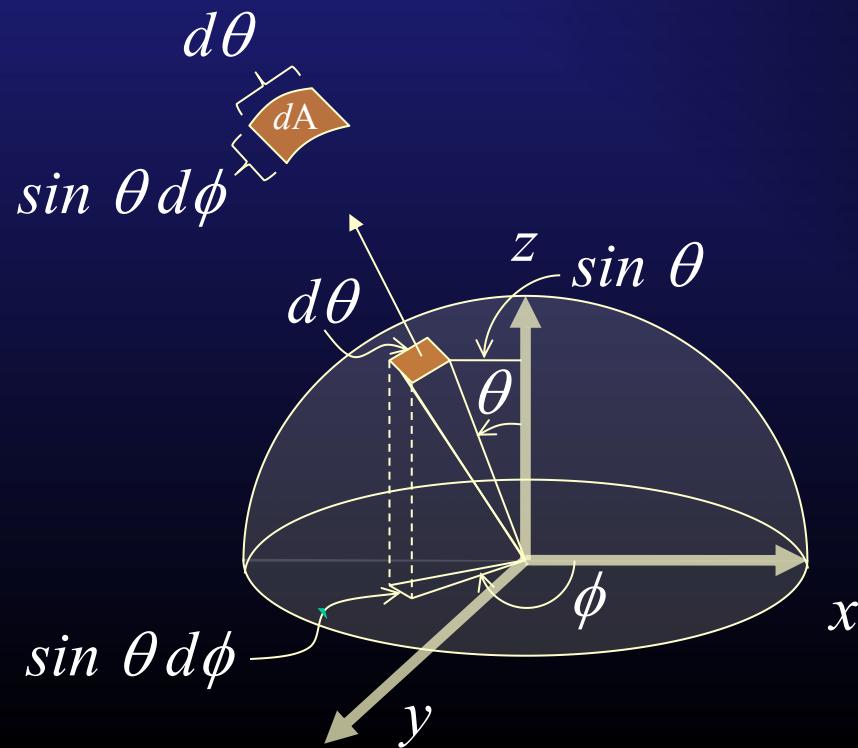
$$\cos \theta_i d\omega_{ij} = \cos \theta_i \frac{dA_j \cos \theta_j}{2\pi r^2}$$



Converting Integrals over Solid Angles to Integrals over Spherical Angles

- Differential area dA subtended by a solid angle = differential edge length $\sin \theta d\phi$ times $d\theta$: $d\omega = \sin \theta d\theta d\phi$
- Also $d\omega^\perp = |\cos \theta| d\omega$ so:

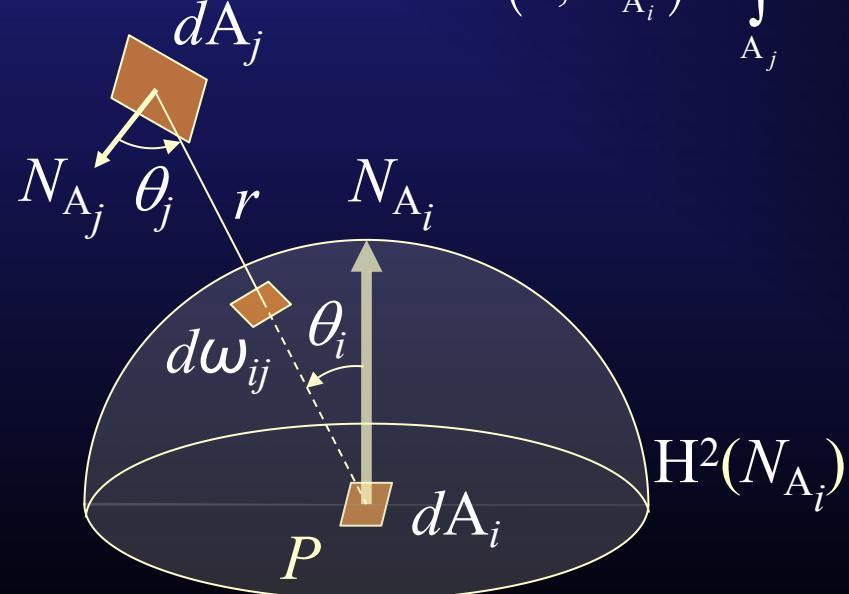
$$E(P, N) = \int_{H^2(N)} L_i(P, \omega) d\omega^\perp = \int_0^{2\pi} \int_0^{\pi/2} L_i(P, \theta, \phi) \cos \theta \sin \theta d\theta d\phi$$



Incident Radiance from a Quadrilateral Patch to a Quadrilateral Patch

- Differential area is related to solid angle from P by: $d\omega_{ij} = \frac{dA_j \cos \theta_j}{r^2}$
- (Check this is intuitively sensible: if $r=1$ and $\theta_j=0$ then $d\omega_{ij} = dA_j$)
- Over a quadrilateral source A_j (where L is light emitted from A_j):

$$E(P, N_{A_i}) = \int_{A_j} L \cos \theta_i \frac{\cos \theta_j dA_j}{r^2}$$



Incident Radiance from a Quadrilateral Patch to a Quadrilateral Patch

- Were the radiance the same in every direction about the hemisphere (evaluating that integral, without proof): $E(P, N_{A_i}) = \pi L_i$

$$E(P, N_{A_i}) = \pi L_i = \int_{A_j} L \cos \theta_i \frac{\cos \theta_j dA_j}{r^2} \quad (\text{from preceding slide})$$

$$\begin{aligned} \text{So: } L_i &= \frac{1}{\pi} \int_{A_j} L \cos \theta_i \frac{\cos \theta_j dA_j}{r^2} \\ &= \int_{A_j} L \cos \theta_i \frac{\cos \theta_j dA_j}{\pi r^2} = \int_{A_j} L \frac{\cos \theta_i \cos \theta_j}{\pi r^2} dA_j \\ &= \int_{A_j} L F_{ij} dA_j \end{aligned}$$

where F_{ji} is called the Form Factor.

The Form Factor

- The Form Factor is fraction of energy that leaves one surface j and reaches another i (along a direct path).
- Form Factor is purely geometric: its value is independent of viewpoint or surface attributes.
- We just derived F_{ji} between two differential surface areas dA_i of surface i and dA_j of surface j :

$$dA_j F_{ji} = \frac{\cos \theta_i \cos \theta_j}{\pi r^2} V(i, j)$$

- where

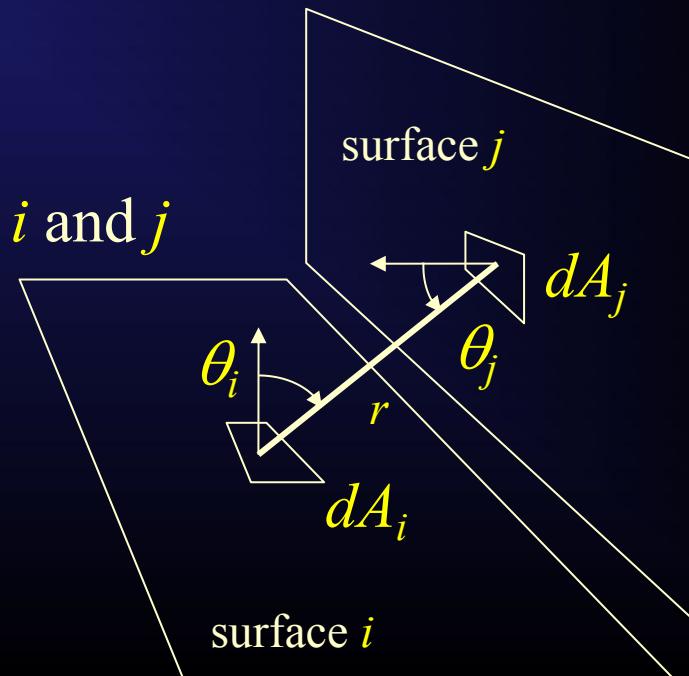
dA_i, dA_j = differential area of surfaces i and j

r = distance from dA_i to dA_j

θ_i = angle between Normal to A_i and r

θ_j = angle between Normal to A_j and r

$V(i, j) = \text{visible}(dA_i, dA_j) \quad [\in \{0,1\}]$



Form Factor Between Two Surfaces

- Integrate over all such differential surface patches:

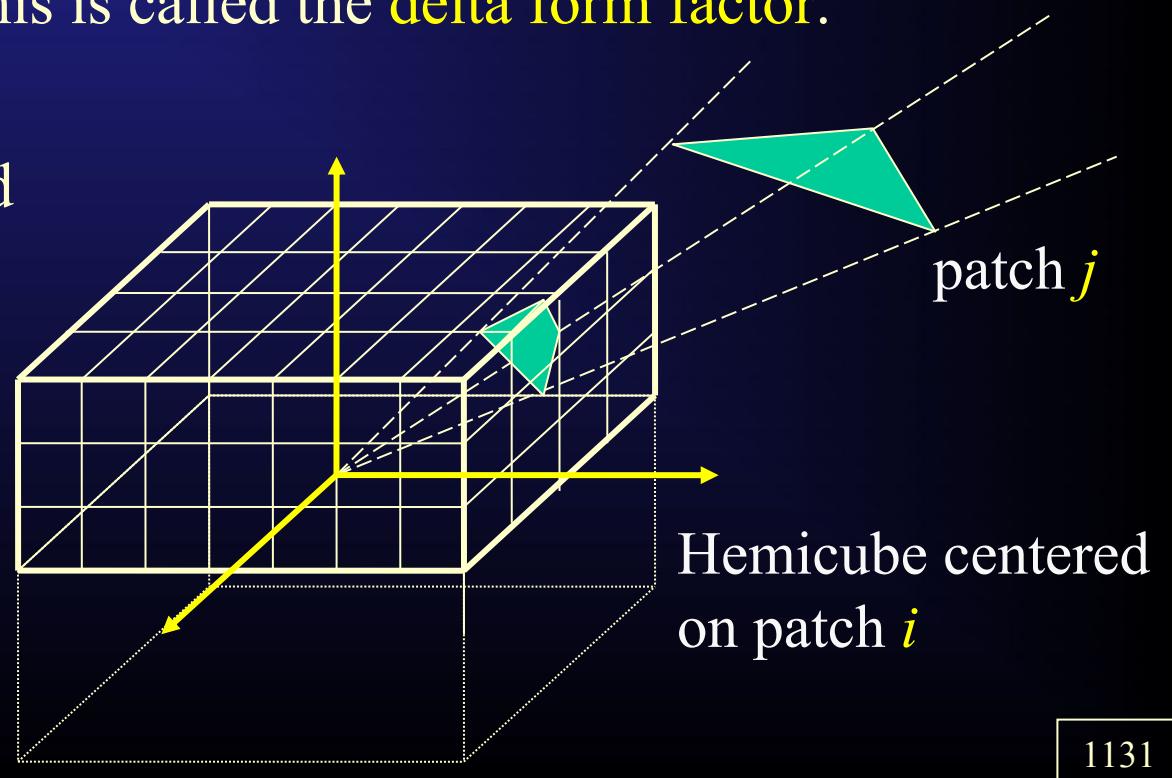
$$A_j F_{ji} = \int_{A_j} \int_{A_i} \frac{\cos \theta_i \cos \theta_j}{\pi r^2} dA_i dA_j$$

or $F_{ji} = \frac{1}{A_j} \int_{A_j} \int_{A_i} \frac{\cos \theta_i \cos \theta_j}{\pi r^2} dA_i dA_j$

- Note that $F_{ij} A_i = F_{ji} A_j$ (reciprocity relationship).
- Computing an approximation to this can be done with a **hemisphere** or a **hemicube**.
- The hemicube is an easier implementation.

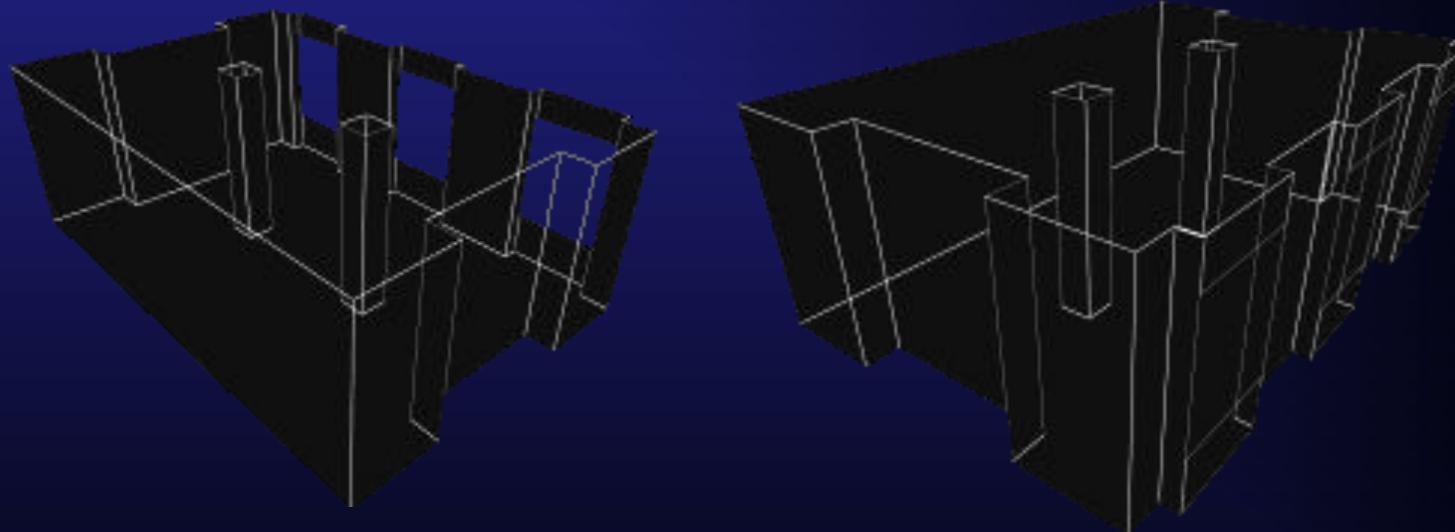
The Hemicube Approximation to the Form Factor

- Erect a hemicube (half a cube) centered over the surface patch i .
- The hemicube surface is subdivided into cells.
- Project surface patch j onto the hemicube cells.
- The contribution of each cell on the surface patch to the total form factor is computed : this is called the delta form factor.
- Sum the delta form factors for the covered cells to approximate the true form factor.

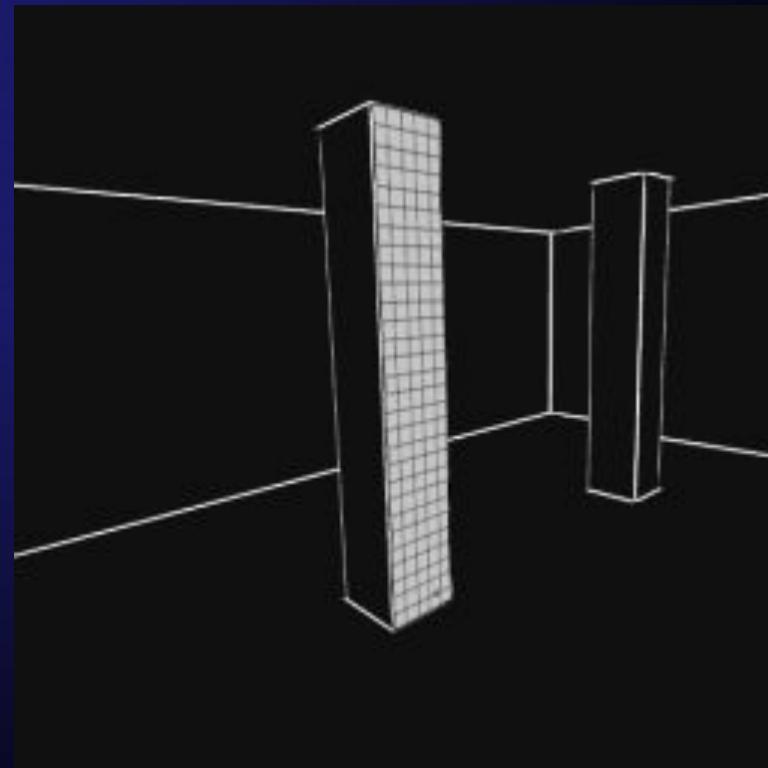
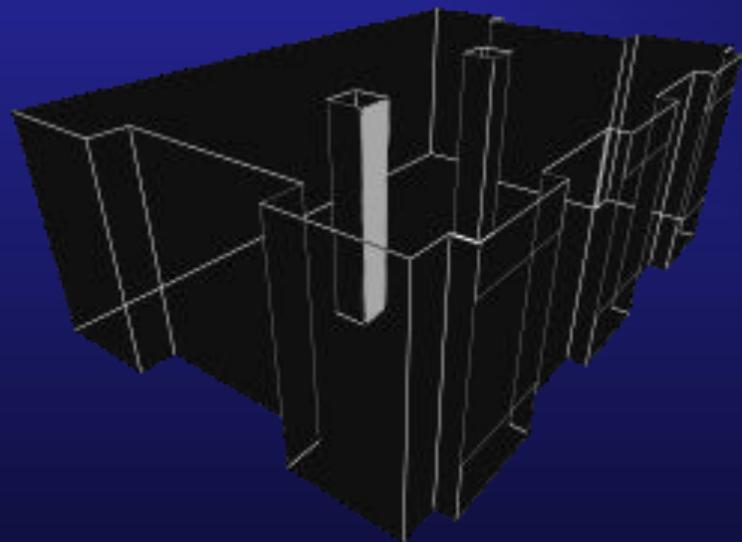


Let's do an Example...

- Polygon mesh room scene.
- Only light source is the sun outside the room.
- Sun is a finite size disk – not a point!

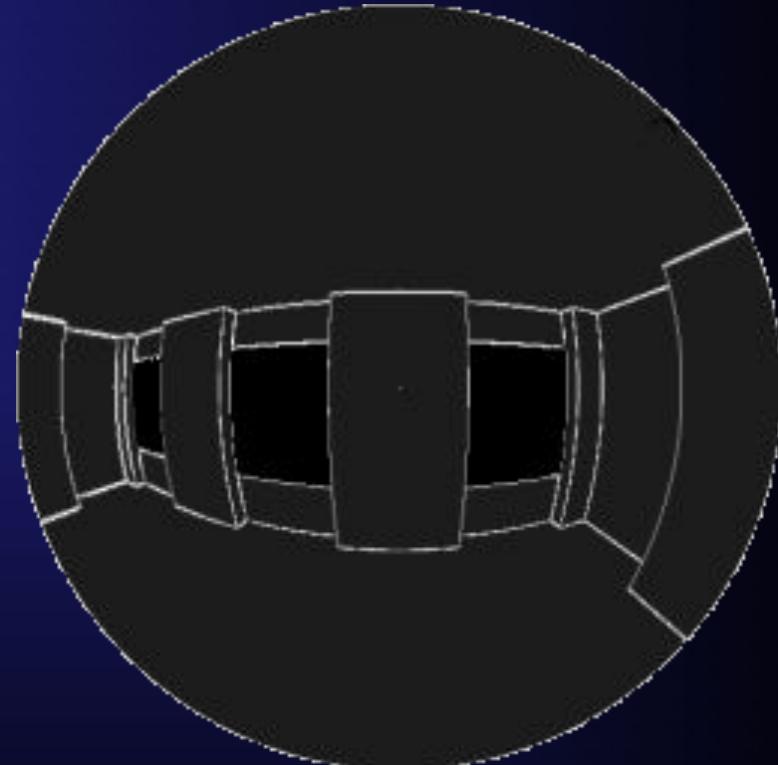
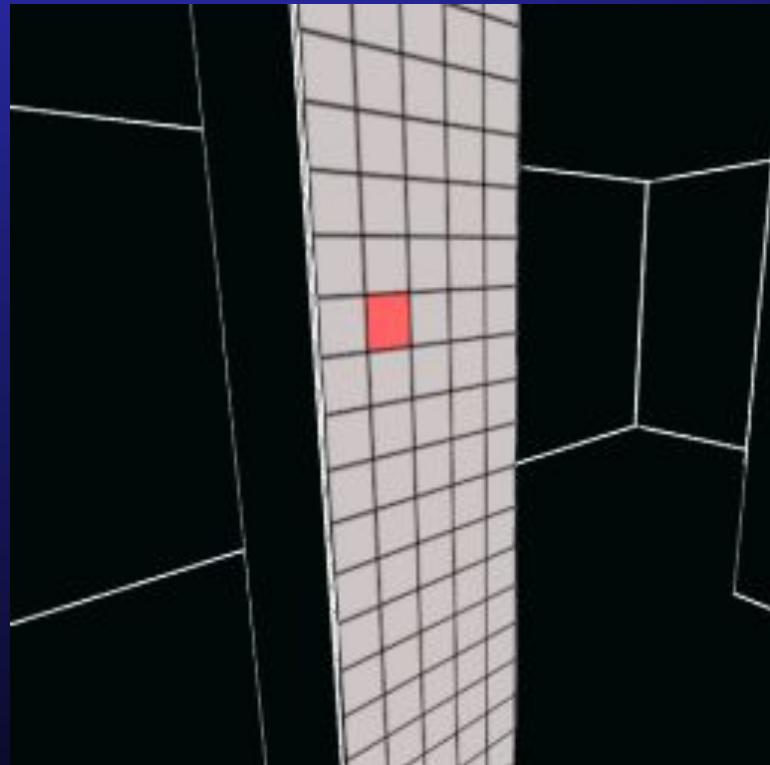


Patch Generation (per Polygon)



View from a Patch

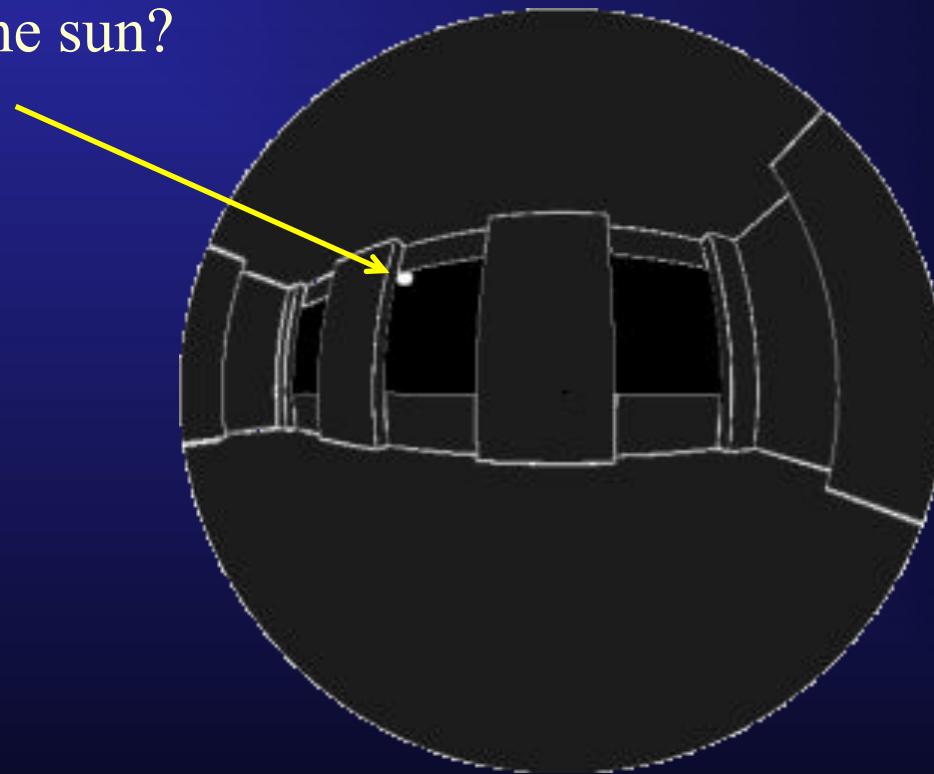
- Each patch stores irradiance (received light).



- The “red” one shown here receives no light from the sun patch, as shown in the image on the right (sun is blocked).

View from a Lower Patch

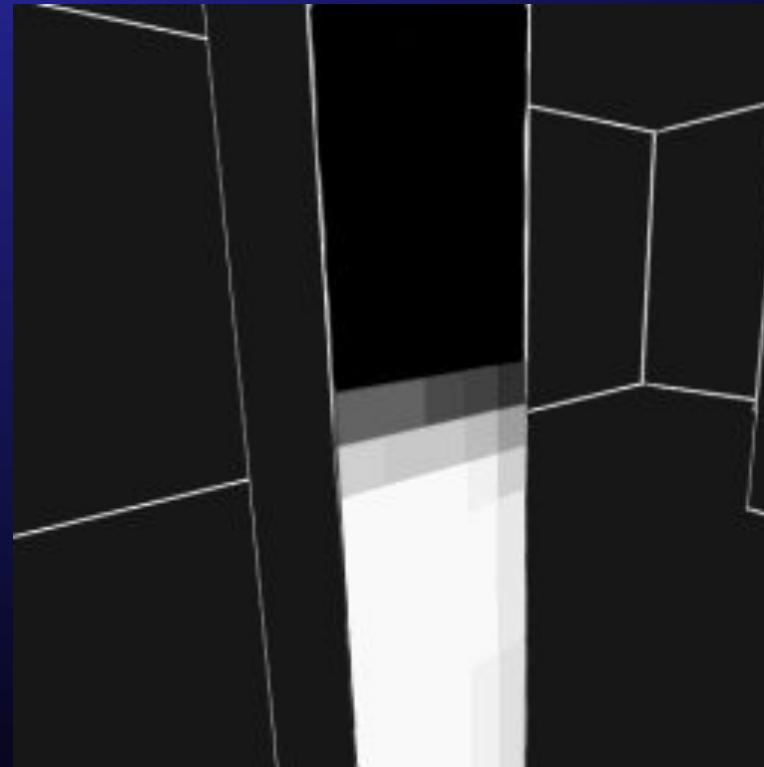
- See the sun?



- So this patch receives some light (irradiance).

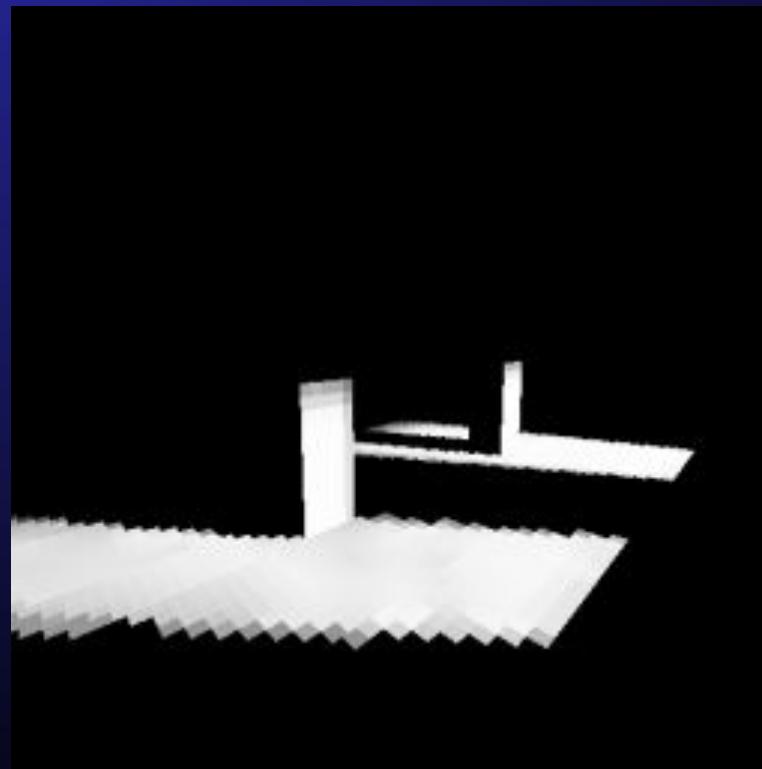
Do this for all Patches (Shown just for the pillar)

- Gray values show variation in irradiance due to partial coverage form factors between the patch and the finite sun disk “patch”.

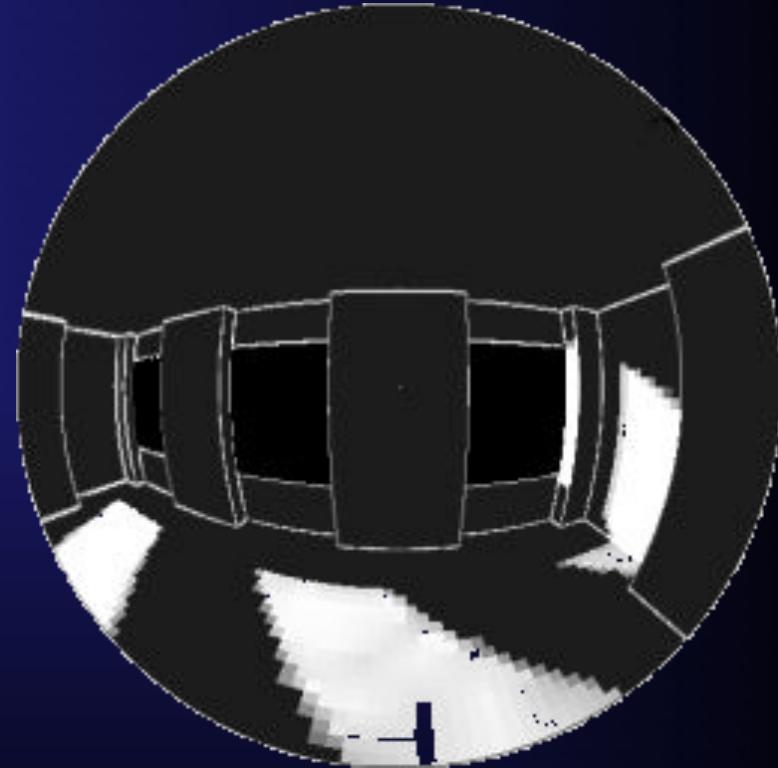
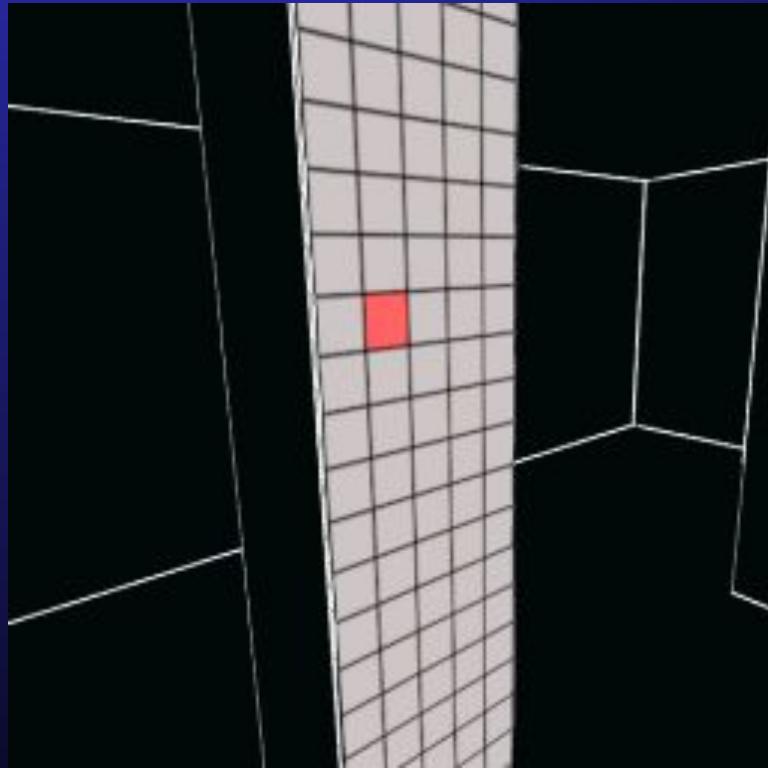


Do Patches for Entire Room (1st Pass)

- Clearly depends on whether or not a patch sees the sun patch (and what percentage of it).

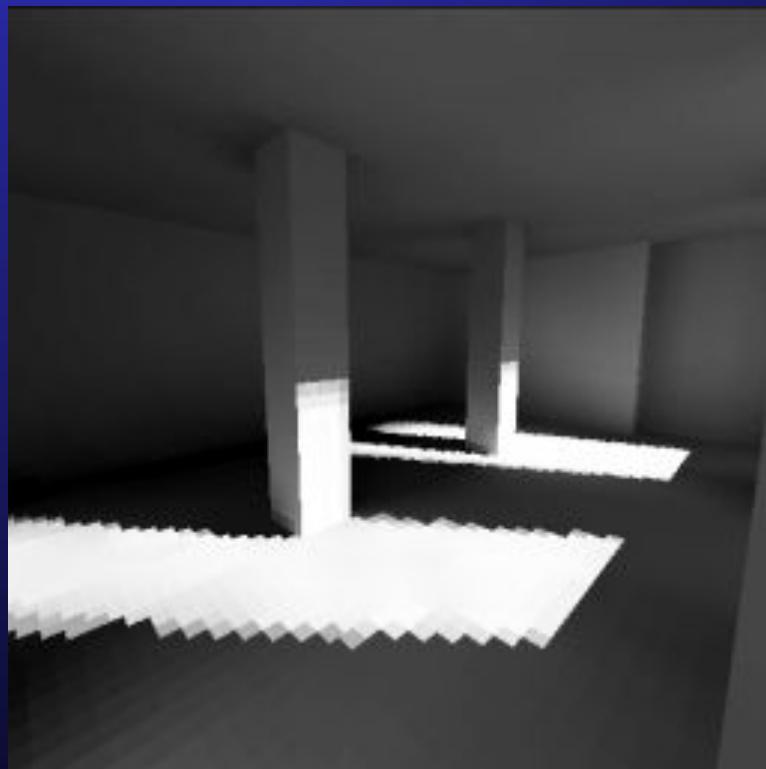


But our Sample Patch (that saw no light before)
NOW sees light from other patches...

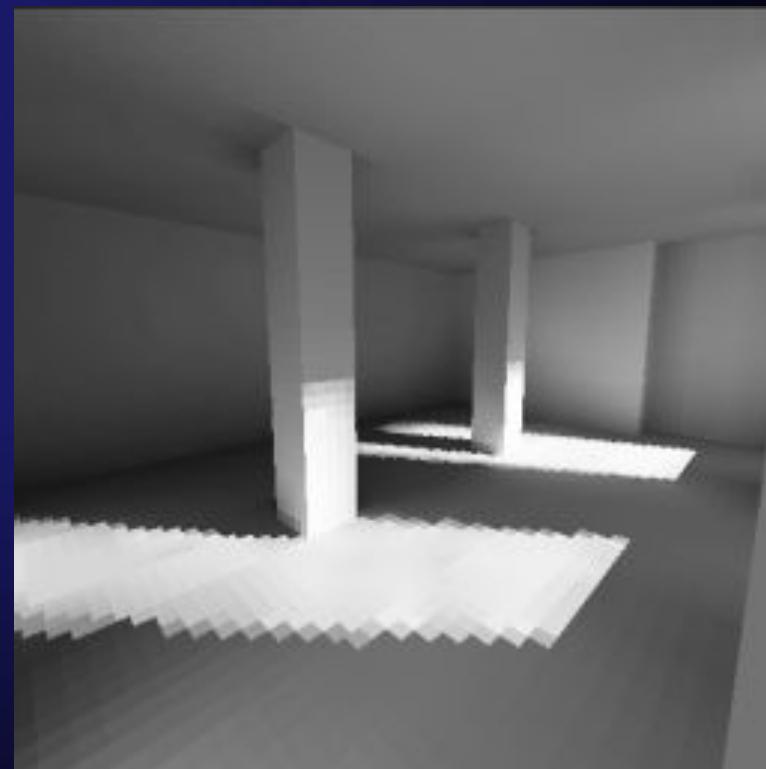


- Add this irradiance to the radiosity of each patch, as per the rendering equation (note form factor is handled explicitly by the visible surface computation).

So Making a 2nd and then a 3rd Pass for All Patches:



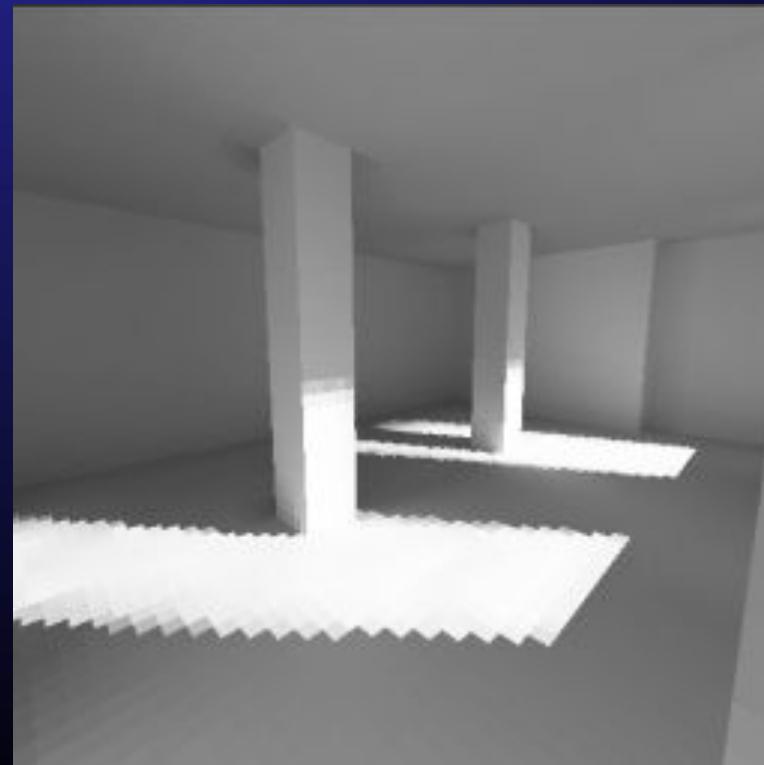
2nd



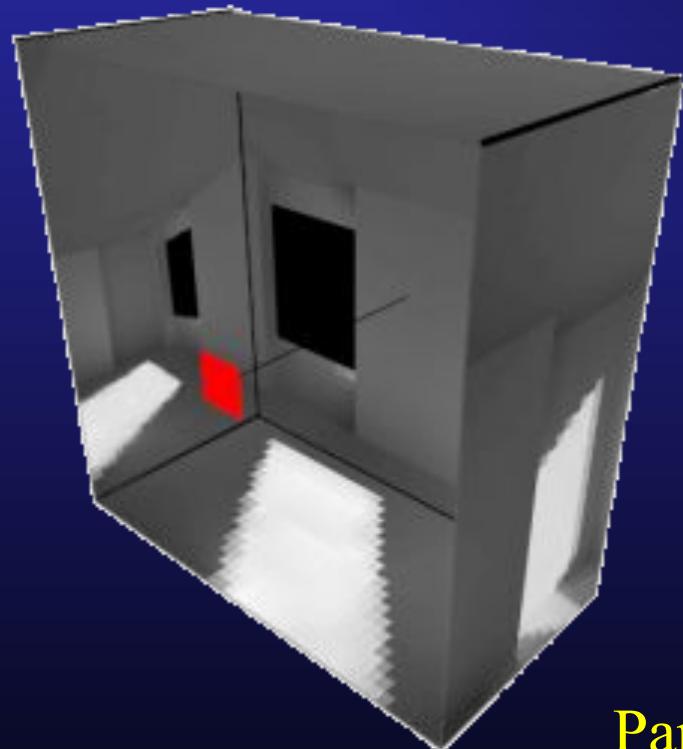
3rd

By the 16th Iteration

- Note the blockiness is due to the patches: each has a CONSTANT illumination across its extent.
- Remember, this is just from the initial distant sun light patch!

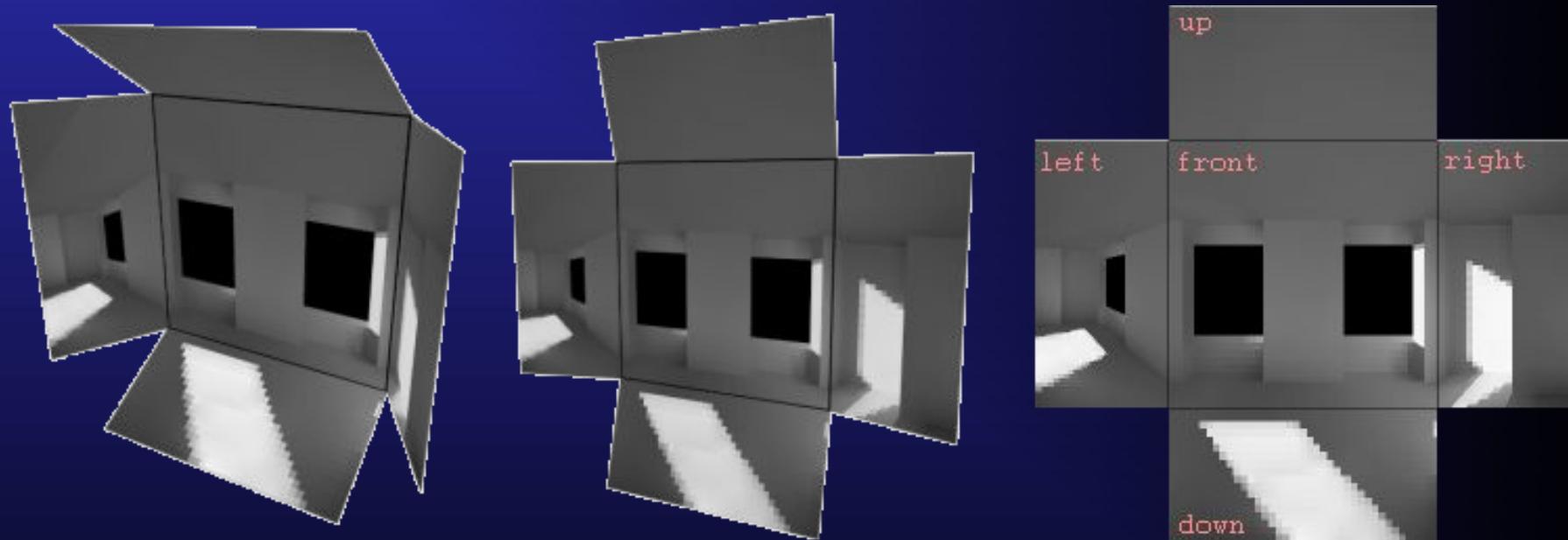


Let's Look at that Patch's View of the World from
the Hemicube on its Center



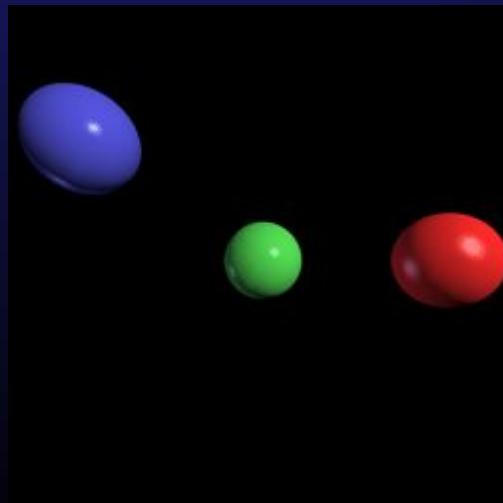
Partial rendering from center of hemicube

Unfold Hemicube Surfaces;
Now can see that each hemicube cell is a pixel



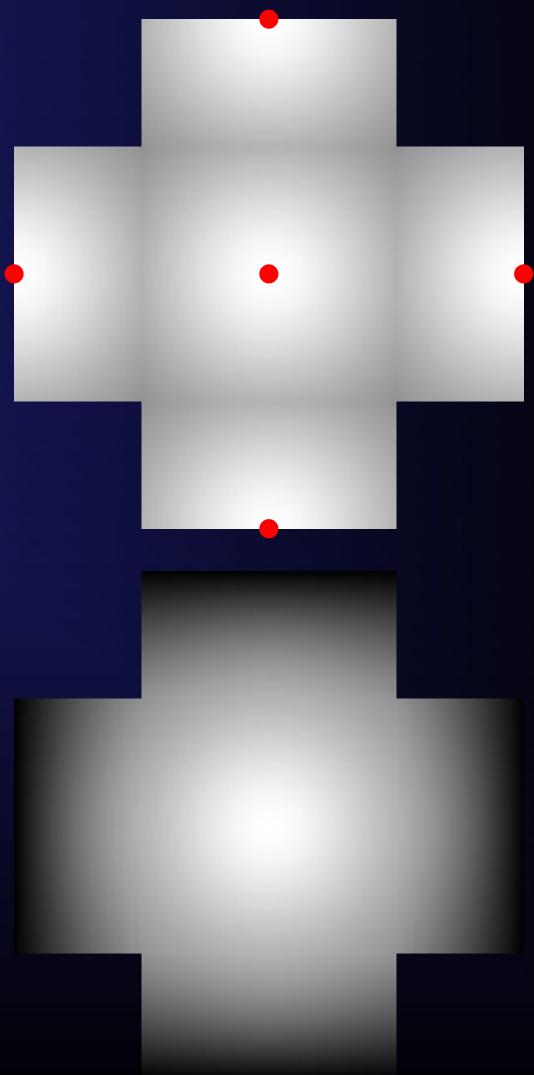
Compensating for the Hemicube's Shape

- The illumination on every pixel of the hemicube map is not equal since some pixels represent more coverage in space than others. (Note: While this is a property of the hemicube, if we used a hemisphere we would still have to worry about equalizing any discretization of its surface.)
- For example, if these were three equally bright, equally distant light sources, the lights near the edge would cast more light than they should (because they cover more area; unfairly due to the distortion!).



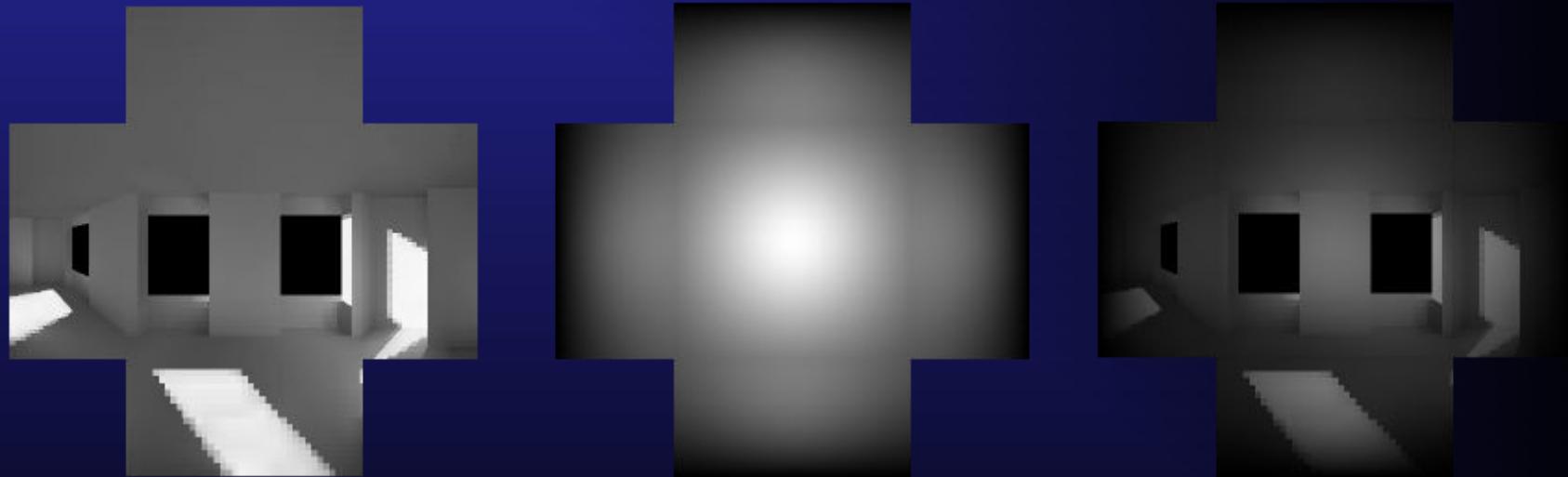
Create a Compensation Map C^*D

- $C = \text{Cosine of the angle between camera direction and the pixel center on each face of the hemicube:}$
- (These points • are bright ($=1$), e.g., because the 5 camera directions are perpendicular to each face of the hemicube.)
- $D = \text{Lambert's Law, just the cosine of angle just from the camera patch normal direction. normalized so the sum of all pixels here} = 1.$ (Only the patch normal is the maximum, in the center.)



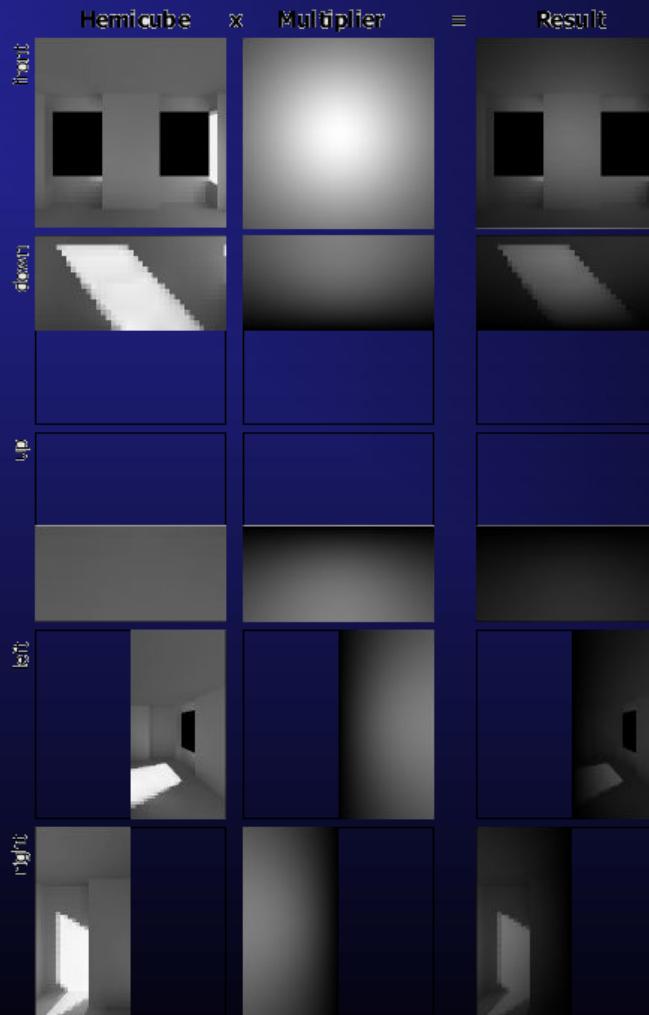
Pixelwise Multiply Hemicube by C*D to Create the Result

$$\text{Hemicube irradiance} * \text{C*D Multiplier} = \text{Result}$$



- Thus each hemicube pixel contributes its visible illumination (irradiance) weighted by incident angle and pixel area.

Showing Results for each Hemicube Area

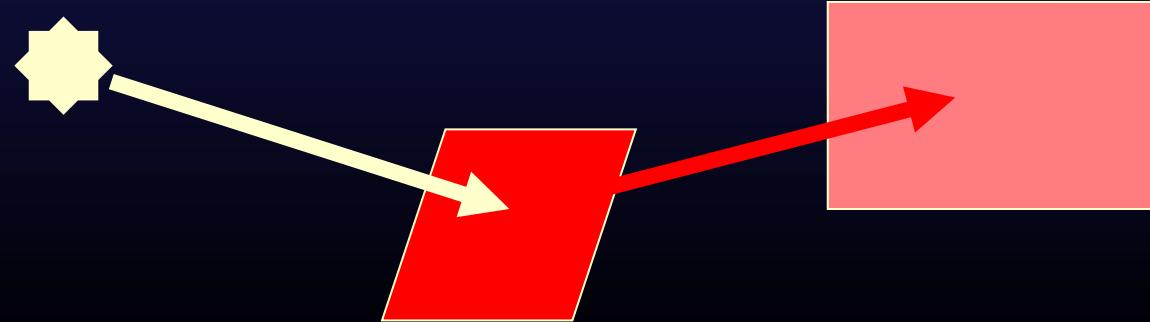


Z-Buffer Implementation of Hemicube

- The F_{ij} for surface i can be approximated for all the surface patches in the scene. (Assume the color of surface patch j is denoted by the unique integer identifier j – e.g., a lookup table index will do this nicely if there aren't too many patches).
- For each surface of the hemicube:
 - Use the hardware frame and z-buffers to create a low resolution rendering of the scene from the surface patch i center with the camera aimed along the patch i normal.
- The sum of the visible pixels of a given index j (each multiplied by the multiplier matrix) is the form factor estimate F_{ij} .

Computing the Surface Color

- For color (R,G,B) have to compute radiosity for each primary -- the good news is that the form factors of course don't change.
- Emitters can be a white or colored light.
- Reflectivities are the intrinsic color of the surface.
- So once the radiosities are computed for (R,G,B), that triple can be used (baked on) as the color of the surface patch.
- Since, e.g., a white surface will reflect, say, red light as red, color bleeding effects occur naturally.



Heat as Energy for Radiosity



The Radiosity Equation

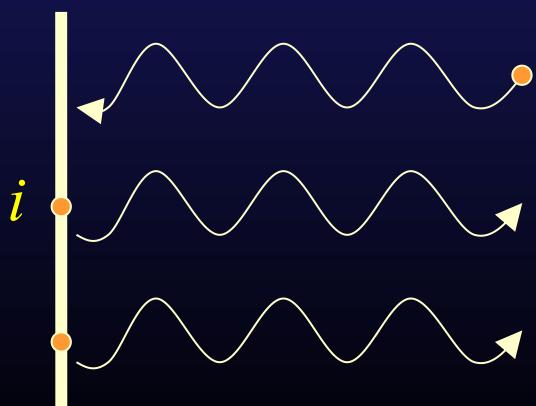
$$B_i = E_i + \rho_i \sum F_{ij} B_j \quad (\text{sum over all patches } j)$$

B_i, B_j = radiosity of surface i and j , respectively

E_i = emissivity of surface i

ρ_i = reflectivity of surface i

F_{ij} = form factor of surface j relative to surface i



$\sum F_{ij} B_j$ (Energy reaching this surface from other surfaces)

E_i (Energy emitted by this surface)

$\rho_i \sum F_{ij} B_j$ (Energy reflected from this surface)

Solving the Radiosity Equation

$$B_i = E_i + \rho_i \sum F_{ij} B_j \quad (\text{sum over all patches } j)$$

B_i, B_j = radiosity of surface i and j , respectively

E_i = emissivity of surface i

ρ_i = reflectivity of surface i

F_{ij} = form factor of surface j relative to surface I

Re-write as:

$$B_i - \rho_i \sum F_{ij} B_j = E_i$$

Now can cast in convenient matrix form:

Solving for the Radiosity

- For each pair of surfaces i and j , calculate F_{ij} .
- Form full matrix radiosity solution (from the equation we saw before) and solve for the B_i :

$$\begin{bmatrix} 1 - \rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 - \rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1 - \rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

- Notice that n can be rather large for complex scenes!

Solving for the Radiosity

- For each pair of surfaces i and j , calculate F_{ij} ($F_{ii} = 0$).
- That means n^2 form factors for n surface patches.
- Moreover, this matrix is not necessarily sparse, though we can truncate very small F_{ij} to 0 without much harm.
- Note that changing geometry requires recomputing the F_{ij} though!
- Cannot solve matrix equation by Gaussian elimination.
- Use iterative methods:
 - “Gathering” via Gauss-Seidel relaxation.
 - “Shooting” via progressive (iterative) refinement.

Direct Gathering Method (Gauss-Seidel)

- Recall $B_i = E_i + \rho_i \sum F_{ij} B_j$
- Let $K = [K_{ij}]$, where $K_{ij} = \delta_{ij} - \rho_i F_{ij}$
- Hence $KB = E$.
- Make an initial guess $B^{(0)}$ (for example, $B = E$) and iterate corrections to hopeful convergence.
- It is called a “gathering” radiosity method as each patch gathers incoming radiosity from all other patches at each iteration: $O(n^2)$.
- Let $B^{(k)}$ be the k^{th} guess; define the residual $r^{(k)}$ by $r^{(k)} = K B^{(k)} - E$
- Algorithm (based on an initial guess $B^{(0)}$):
while B has not yet converged (relative to an appropriate metric)
for $i=1$ to n

$$B_i = E_i - \sum_{j=1, j \neq i}^n \frac{K_{ij} B_j}{K_{ii}}$$

Progressive Refinement “Shooting” Method

- Update all patches each iteration.
- Shoot radiosity from “larger” emitters and reflectors earlier.
- Each patch may “shoot” more than once, so we only need to shoot any added radiosity (ΔB_i) since its last turn.
- $\rho_i F_{ij} B_j$ is the radiosity contribution of patch j to patch i .
- Hence $\rho_i F_{ij} B_j (A_i/A_j)$ is the radiosity contribution of patch i to patch j .
- Algorithm...

Progressive Refinement Algorithm

$B = E$

$\Delta B = E$

while B has not yet converged begin

 pick i so that $A_i \Delta B_i$ is the largest (to help convergence)

 for each $j \neq i$ begin

$$\Delta \text{radiosity} = \Delta B_i \rho_j F_{ij} B_j (A_i/A_j)$$

$$\Delta B_j =+ \Delta \text{radiosity}$$

$$B_j =+ \Delta \text{radiosity}$$

 end

$$\Delta B_i = 0$$

end

Pure Finite Element Techniques

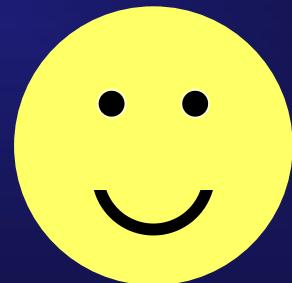
- The finite element solution and pre-meshing costs are too high to make this practical.
- We'll eventually look at better ways to organize the radiance estimates to make the process more efficient.
- Part of the solution will be better sampling techniques, since naïve finite element methods are stuck doing something for every [tiny] patch multiple times.

Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading ✓
- Mapping ✓
- 3D Viewing Transformations ✓
- Anti-aliasing & Compositing ✓
- Global Illumination ✓
- Light Fields & HDR
- Path Tracing/Photon Mapping
- Futures

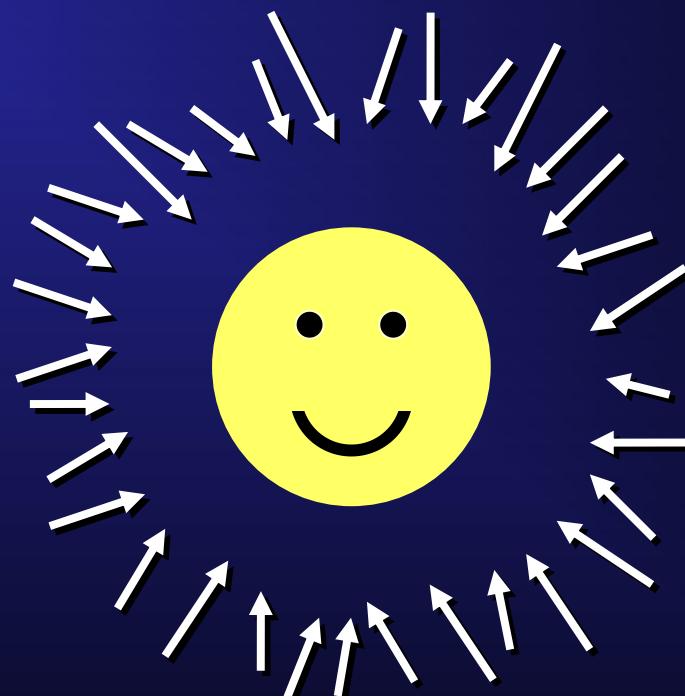
An(y) Object Transforms Incident Illumination into Radiant Illumination

- These illumination spaces are called *light fields*.

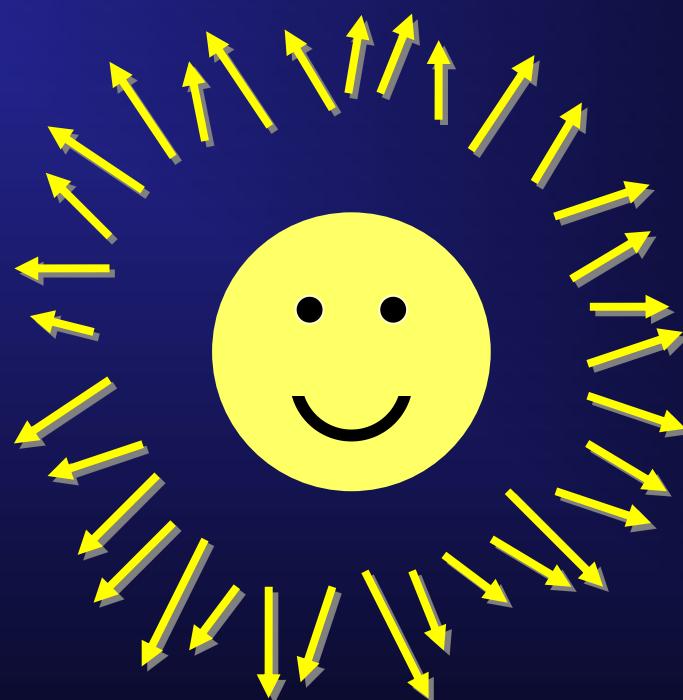


Much of this material is adapted from work by Paul Debevec, *et al.*

Incident Illumination Field (all directions in 3D spherical domain)

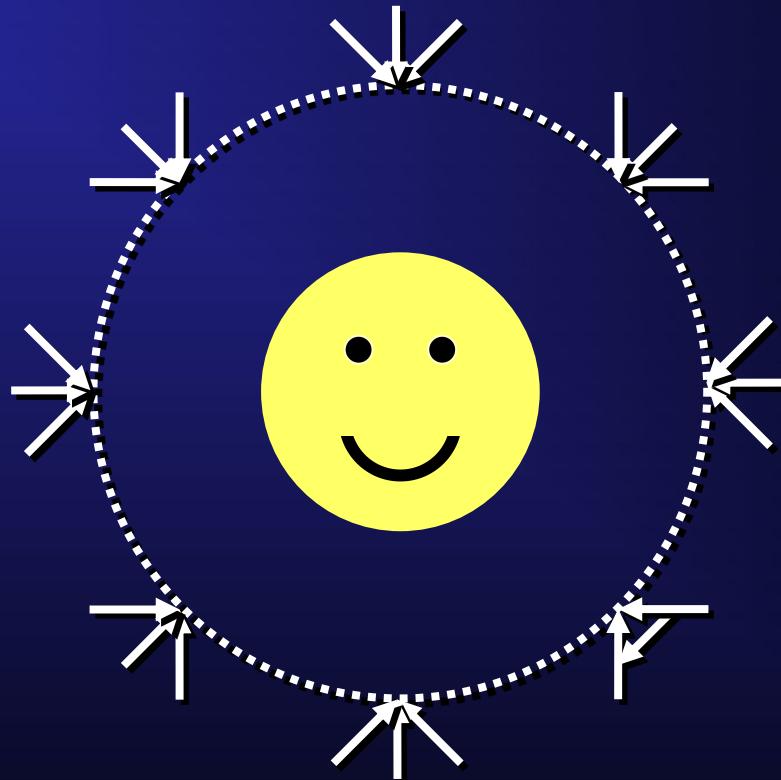


The Reflectance Field (Reflected Illumination in all 3D Spherical Directions)



Incident Light Field

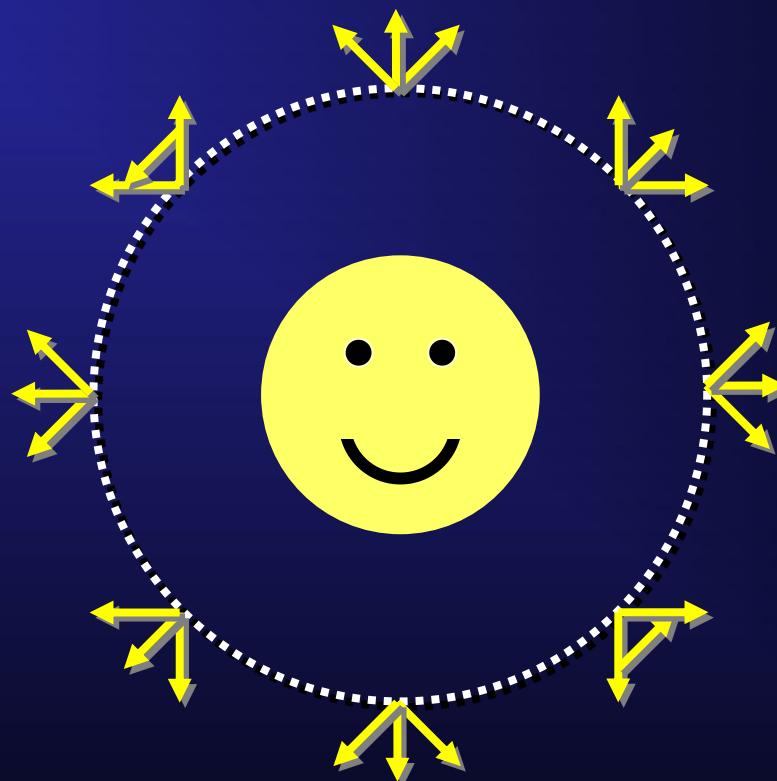
- Select sample incident directions: $R_i(u_i, v_i, \theta_i, \phi_i)$



- (u_i, v_i) on object; (θ_i, ϕ_i) incident direction

Radiant Light Field

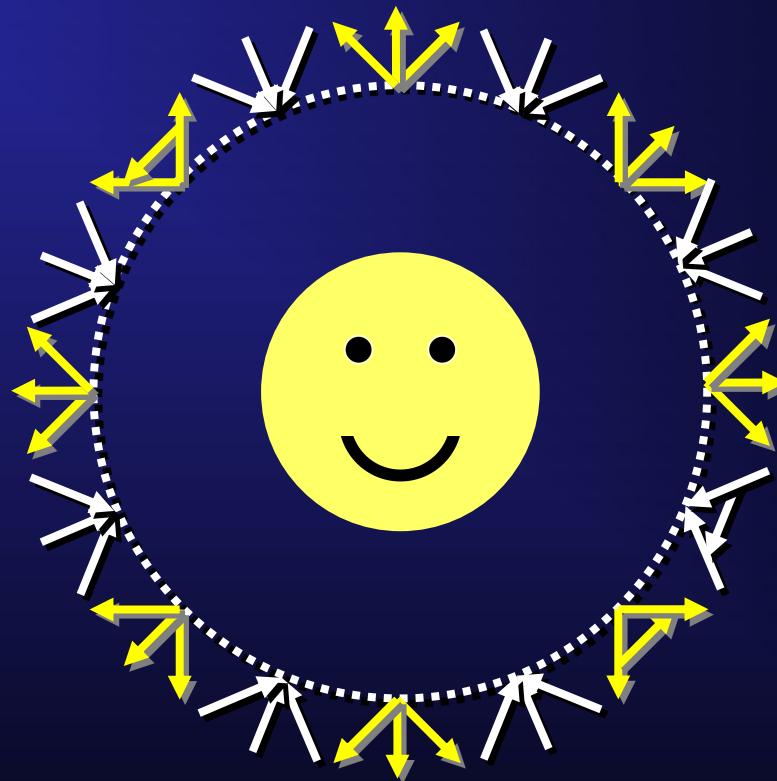
- Select sample reflected directions: $R_r(u_r, v_r, \theta_r, \phi_r)$



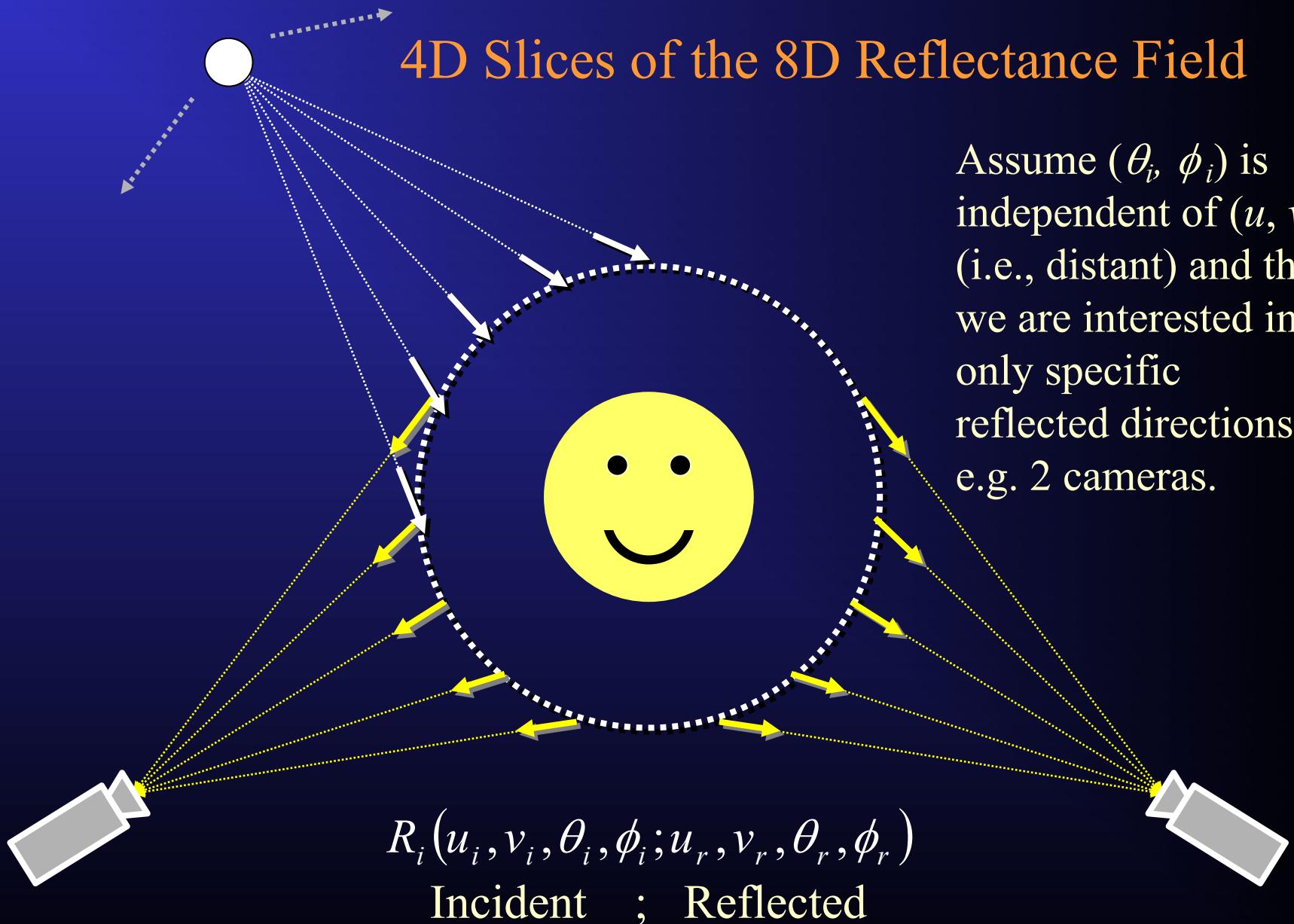
- (u_r, v_r) on object; (θ_r, ϕ_r) reflected direction

8D Total Reflectance Field

$$R_i(u_i, v_i, \theta_i, \phi_i; u_r, v_r, \theta_r, \phi_r)$$



4D Slices of the 8D Reflectance Field



$$R_i(\theta_i, \phi_i; u_r, v_r, camera1) \text{ and } R_i(\theta_i, \phi_i; u_r, v_r, camera2)$$

White (or RGB Triplets) LEDs in Light Stage

- Rapidly pulsed *one-at-a-time* to create incident light basis.

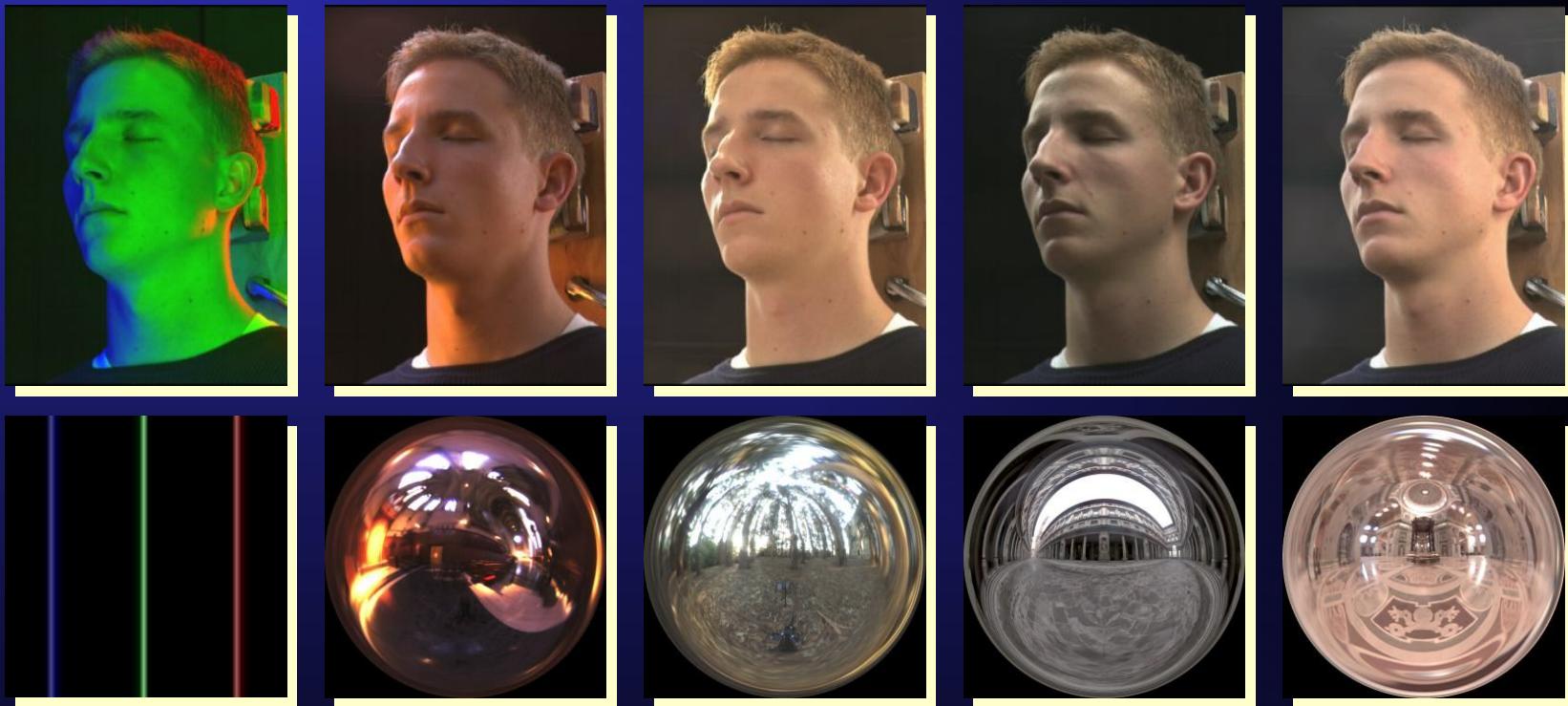


Example Light Stage Data (One Camera View; Original Resolution 64x32)



(Lighting through image recombination: Haeberli '92, Nimeroff '94, Wong '97)

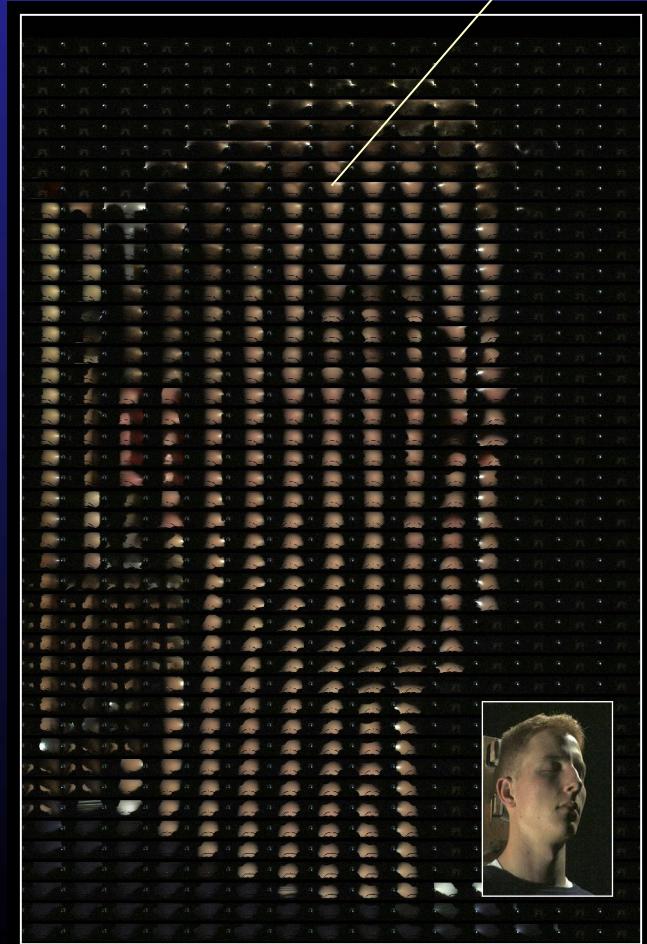
Light Stage Results



Environments from the Light Probe Image Gallery: www.debevec.org

Pixel-wise Reflectance in Dense Sampled Incident Directions

(Enlarged sample)



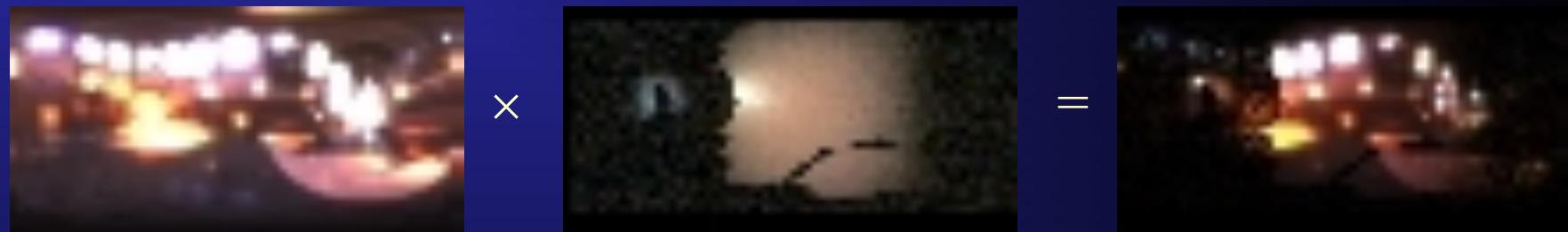
- Composite map of reflectance at each pixel.
- Each 64×32 reflectance function consists of the corresponding pixel location's appearance under 2000 incident lighting directions distributed throughout a (distant) sphere surrounding the face.
- The inset shows the same view of the face under a combination of three lighting directions.
- (The functions have been brightened by a factor of four from the original data.)

Relighting the Face Object

- Suppose that we wish to generate an image of the face in a novel illumination field. Since each $R_{xy}(\theta, \phi)$ represents how much light is reflected toward the camera by pixel (x, y) as a result of illumination from direction (θ, ϕ) , and since light is additive, we can compute an image of the face $L(x, y)$ under any combination of the original light sources $L_i(\theta, \phi)$ as:
$$L(x, y) = \sum_{\theta, \phi} R_{xy}(\theta, \phi)L_i(\theta, \phi)$$
- Each color channel is computed separately.
- Since the light sources densely sample the viewing sphere, any form of sampled incident illumination can be computed using this basis.
- Subsequent work expanded this approach to animation and actual skin reflectance models.

Diagrammatically, for each image pixel:

Normalized light map \times pixel reflectance function = light product



Light product summed & normalized = pixel value

$$\left\| \text{[image]} \right\|_1 = \text{[red square]}$$
A diagram illustrating the summation and normalization of the light product to produce a pixel value. On the left is the light product image from the previous diagram. To its right is a vertical double bar symbol followed by a subscript '1', indicating summation. An equals sign follows this, leading to a solid red square representing the final pixel value.

Light Field Model Demo

- Virtually *relight* real faces with real lighting environments or position and adjust up to 3 virtual lights.
- The demo program exploits the additivity of the light basis by computing a weighted sum of 512 images, each capturing the appearance of the face when illuminated from a particular direction. The relighting computation is performed in real-time.
- See <http://gl.ict.usc.edu/Research/FaceDemo/>



Video Examples

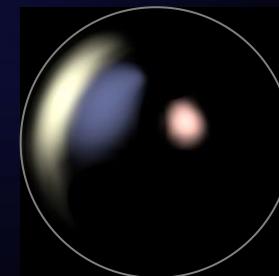
- Can also be applied (with extensions) to animated movements.



“Marissa”



Marissa
light field



Wenger et al. Performance Relighting and Reflectance Transformation with Time-Multiplexed Illumination. SIGGRAPH 2005.

Light Fields with Natural Incident (Emitter) Radiosity

- Radiosity, a physical quantity, is an *unbounded real number!*
- But from the rendering equation, radiosity and surface (R,G,B) are multiplied and summed to get the reflected color, and the emitter radiosity values are unlimited.
- So how can we map this large range of *real* radiosities to the *finite* color space of our graphics display devices?
- And how do we get the radiosity of a *natural* emitter anyway?
 - Take *high dynamic (illumination) range* (“HDR”) photographic images to establish emitter radiosities.
 - Photograph in small silvered sphere to see entire environment.
 - Use radiosities as an environment light map.
 - *Tone map* HDR range to finite color resolution

Much of this work is Paul Debevec’s.

Use Natural Emitter Radiosity, e.g., Light from the Sky



NO PANORAMA USED



PANORAMICA™
LAND & SKY



E.g., this point sees
this area of sky map

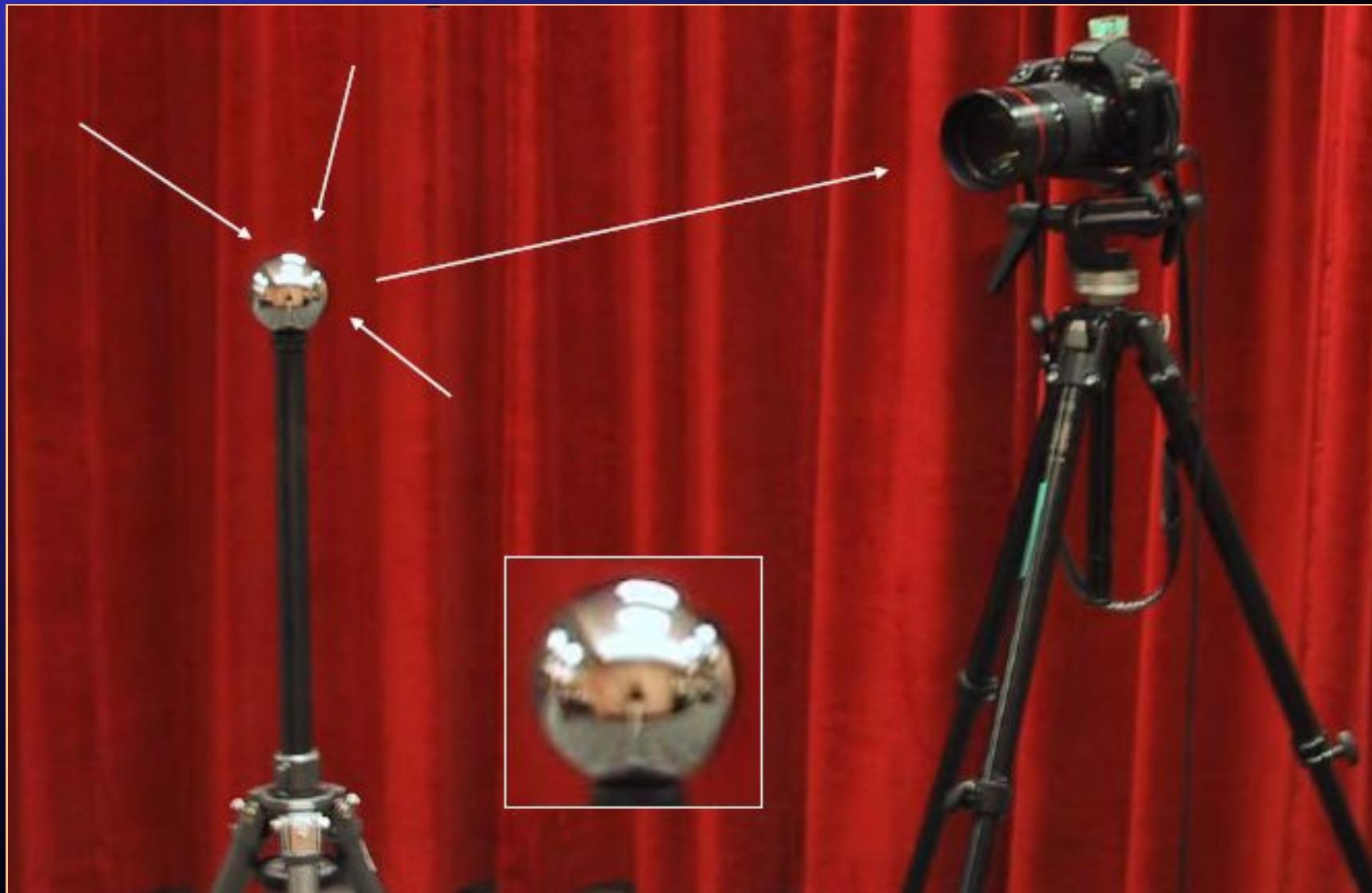
Note that this is an *environment map* for reflections, but it's also used for shadows via hemispherical occlusion of the sky map. Look at ground shadow variations!

Real Environment Maps

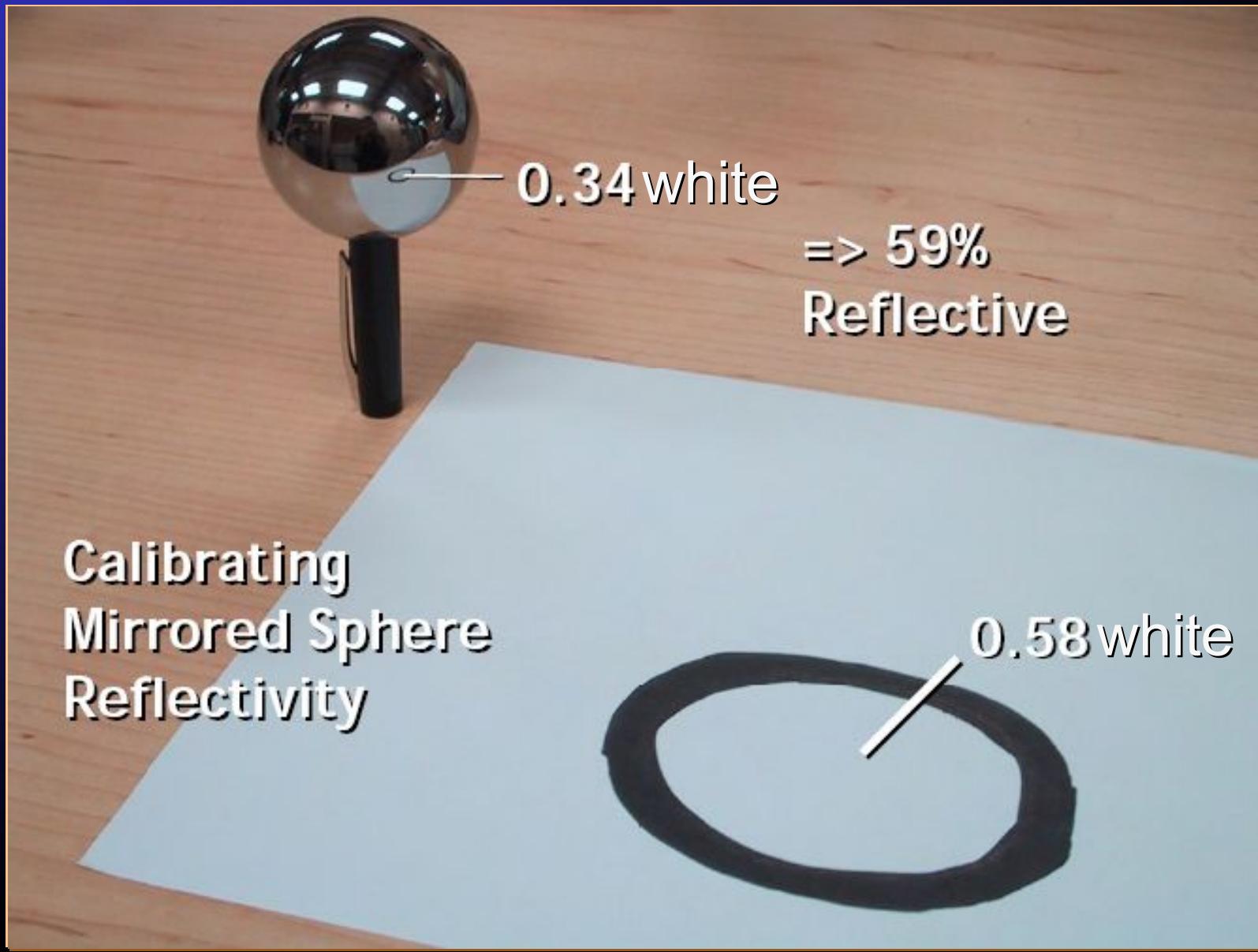
- Fisheye lenses
- Panoramic photographs
- Mirrored chrome spheres



Mirrored Sphere



Calibration is Essential! (Nothing is perfect)



Some Real World Lighting Environments

Funston
Beach



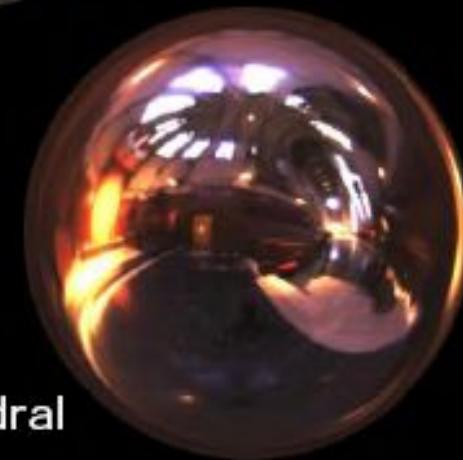
Eucalyptus
Grove



Uffizi
Gallery



Grace
Cathedral



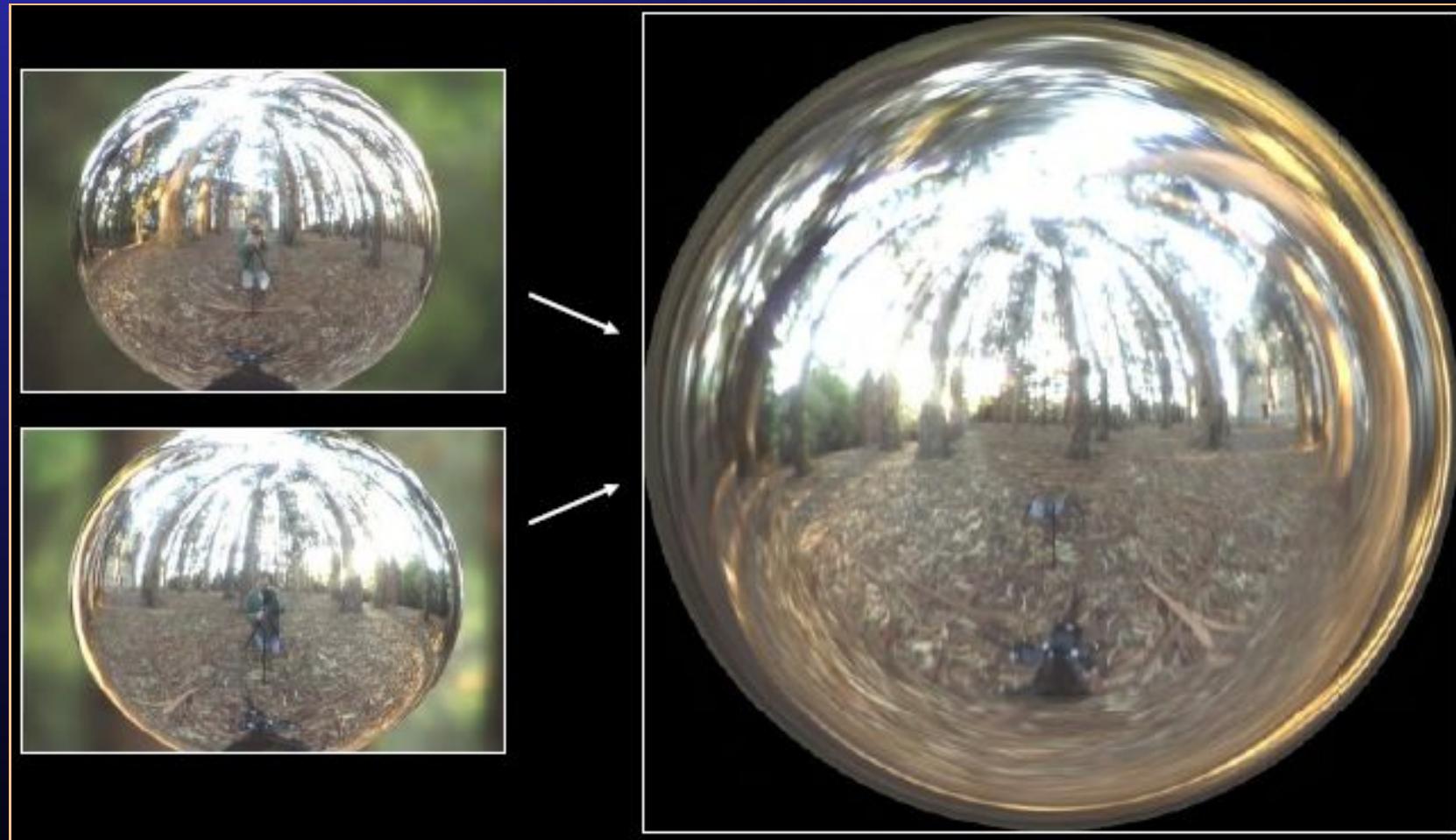
Lighting Environments from the Light Probe Image Gallery:
<http://www.debevec.org/Probes/>

High Dynamic Range Imagery from Multiple Exposures

Acquiring the Light Probe

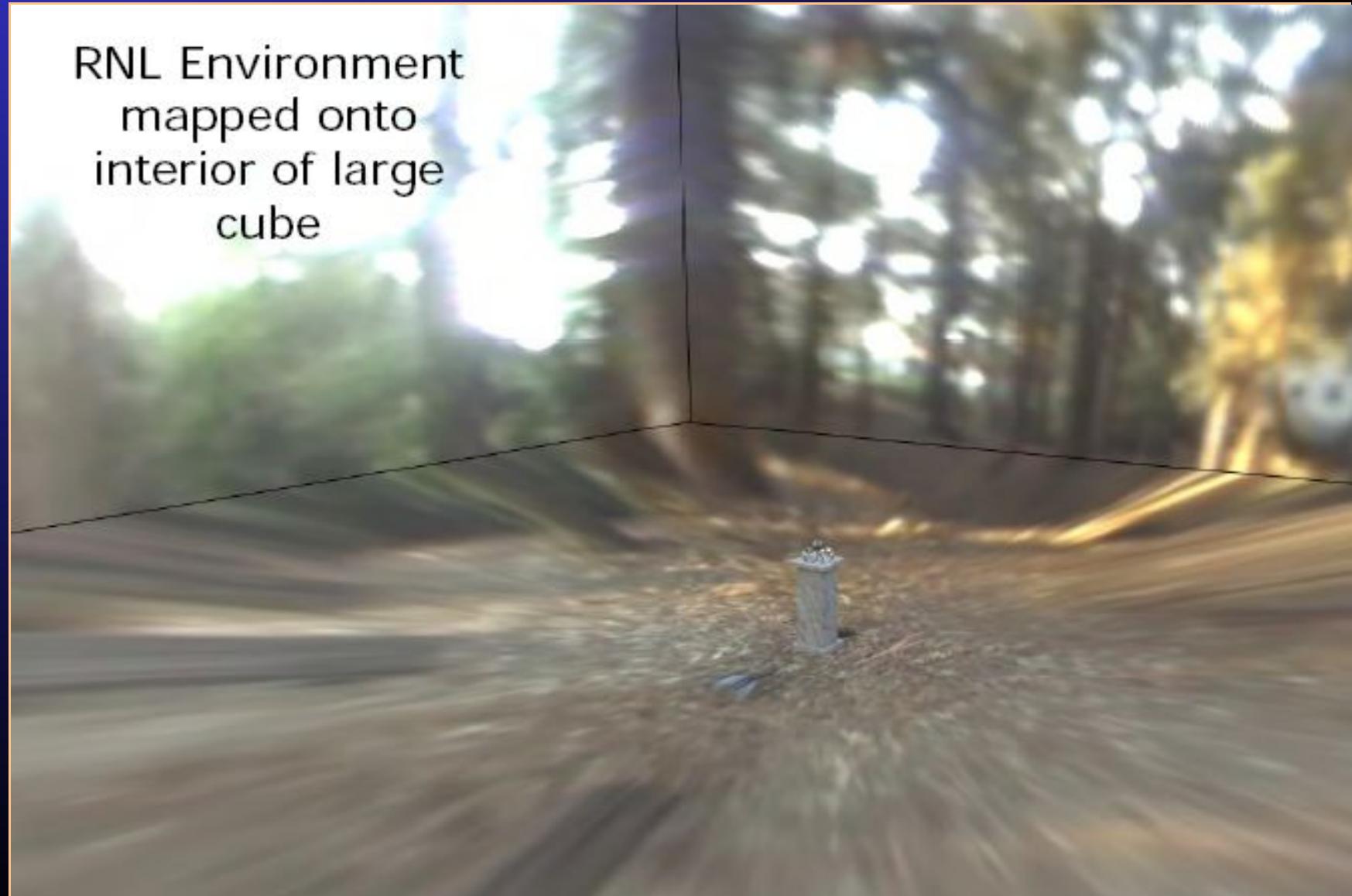


Assembling the Light Probe from Two Hemispheres



Use as Cubical Environment Map

RNL Environment
mapped onto
interior of large
cube

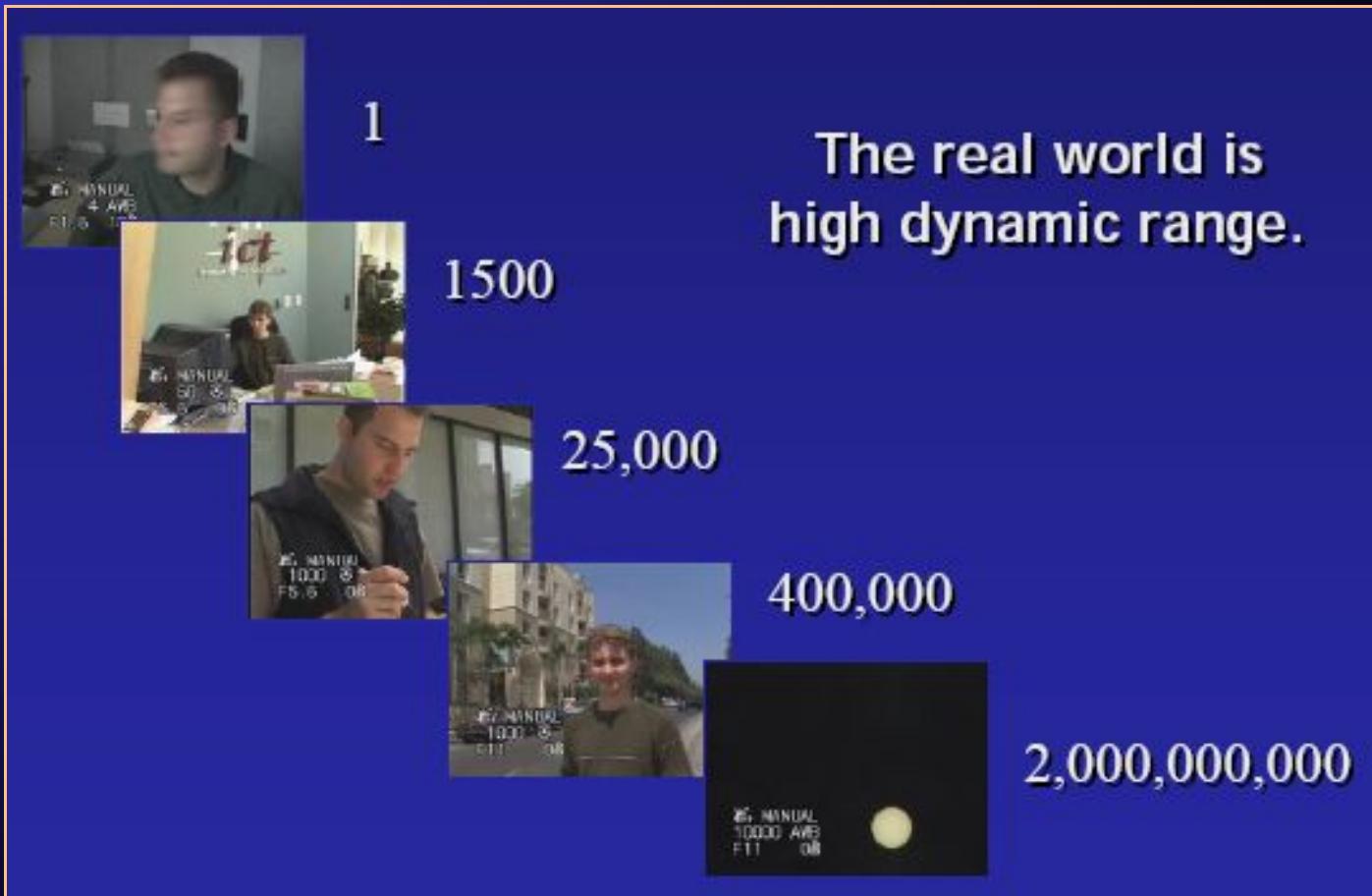


Rendering with Natural Light (Image-Based Rendering)

Paul Debevec, SIGGRAPH '98 Electronic Theater



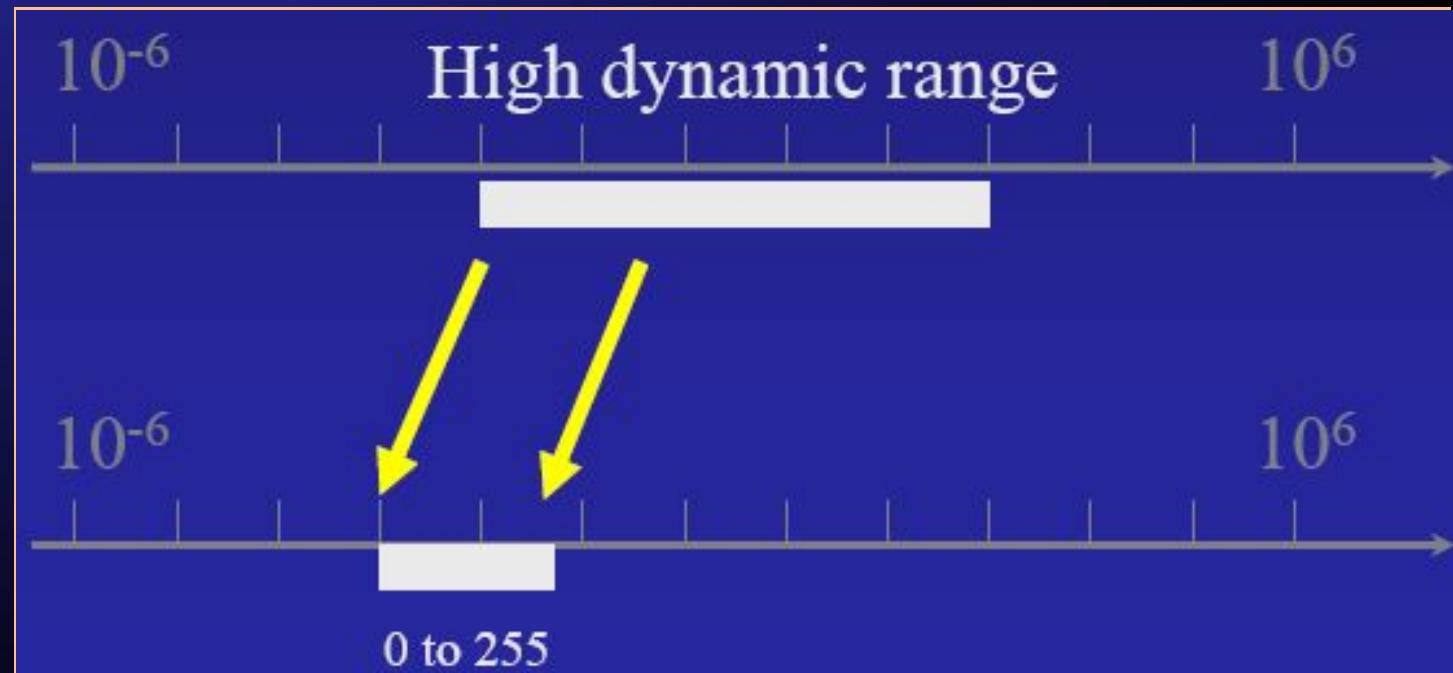
Problem: The *Real World*



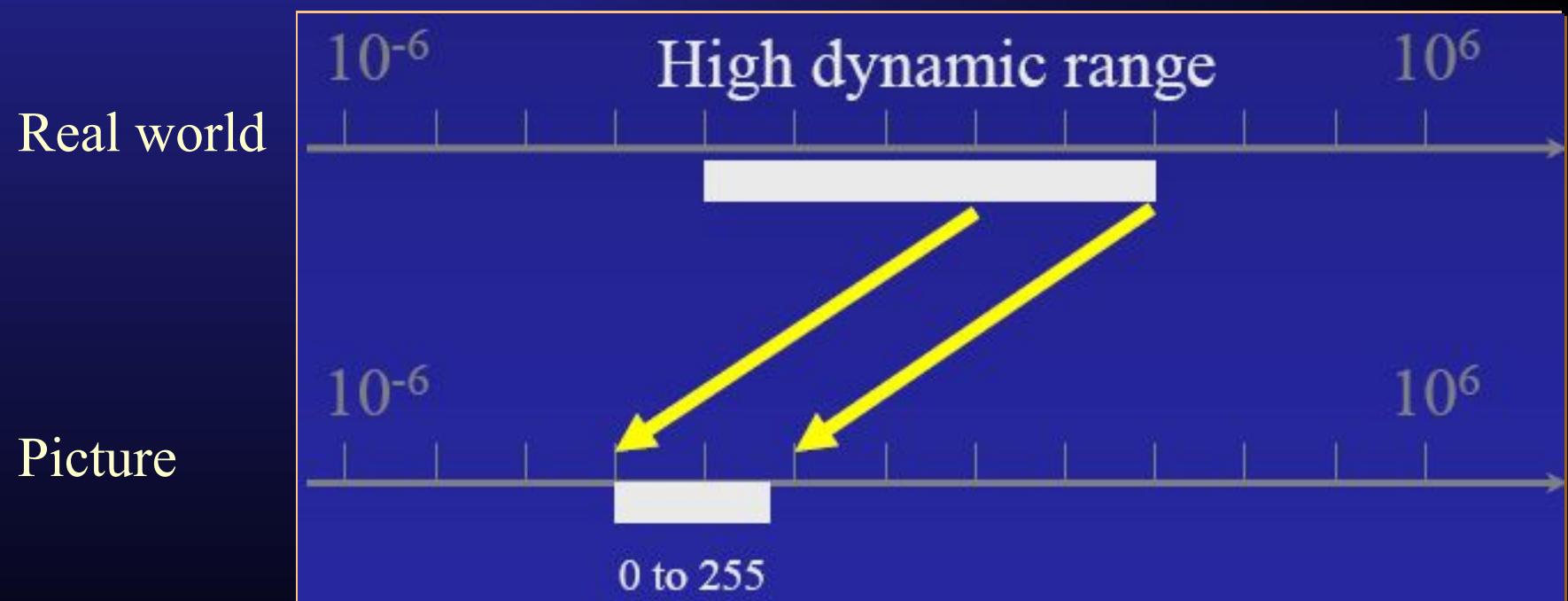
- How can this range be mapped into the available color resolution?
- E.g., 8 bits dynamic range = 256: off by factor of ~8M
- Even with 12 bits dynamic range \approx 4K: off by factor of ~500K.

A Long Exposure Washes out Part of the Dynamic Range

Real world

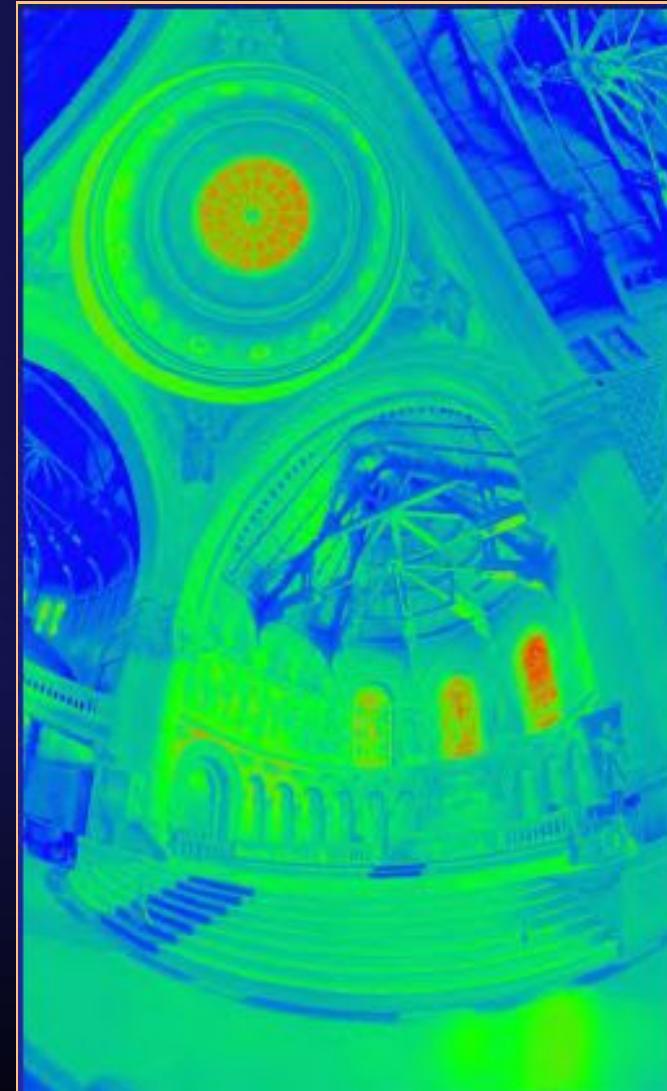


In Photography we “Solve” this Problem with Short Exposure Times



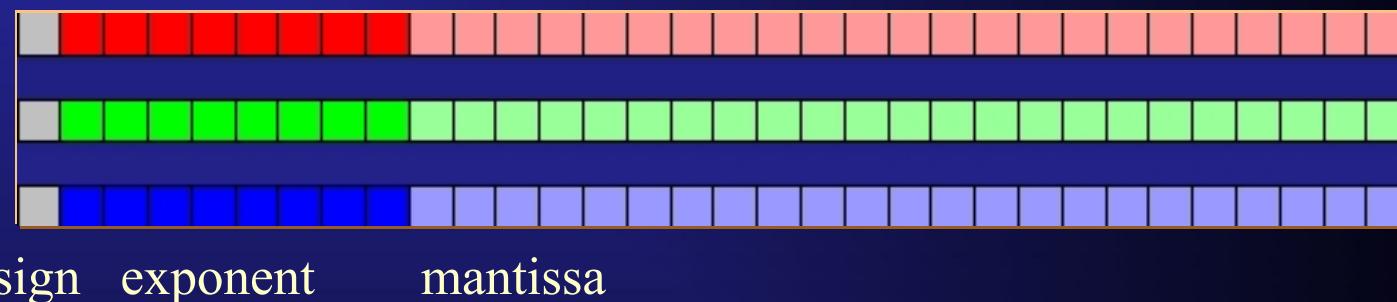
Make a Radiance Map

- Spectral radiance distribution: Radiosity per spherical area.
- Here, e.g., range covers 121.741 to 0.005, or a ratio of 24,348.2 to 1.
- Equi-level tone map would wash out lower or upper values (each pixel would have to cover a range of about 24348.2 / 256 \approx 95*0.005 values \approx 0.476 for each 8 bit color channel value.)



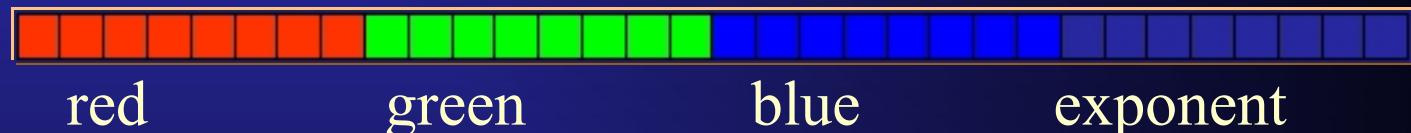
Storing HDR Images (1): Use Floats rather than Integers

- Portable Floatmap (.pfm): 12 bytes/pixel, 4 bytes/RGB channel



Storing HDR Images (2)

- *Radiance** format (.pic, .hdr): 4 bytes/pixel!



$$(R, G, B, e) \Rightarrow (R, G, B) * 2^{e-128}$$

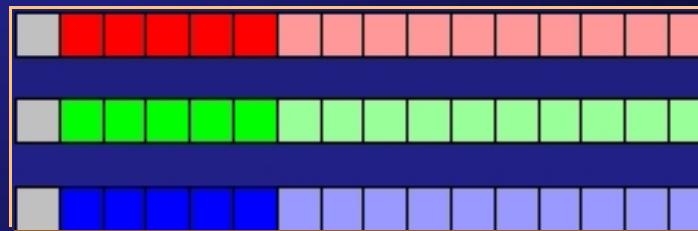
$$\begin{aligned} (145, 215, 87, 149) &= \\ (145, 215, 87) * 2^{149-128} &= \\ (1190000, 1760000, 713000) & \end{aligned}$$

$$\begin{aligned} (145, 215, 87, 103) &= \\ (145, 215, 87) * 2^{103-128} &= \\ (0.00000432, 0.00000641, 0.00000259) & \end{aligned}$$

* Greg Ward, “Real Pixels”, Graphics Gems IV, 1994

Storing HDR Images (3)

- ILM's OpenEXR (.exr): 6 bytes/pixel, 2 for each channel, compressed:

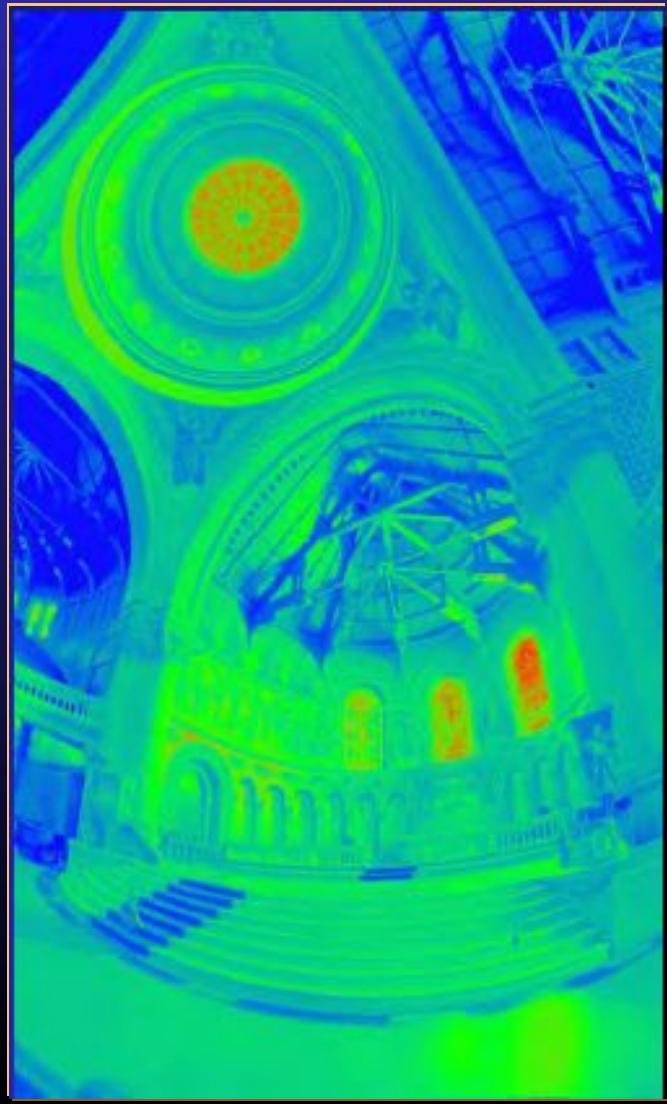


sign exponent mantissa

- Several lossless compression options (2:1 typical).
- Compatible with “half” datatype in nVidia's CG language.
- Available at <http://www.openexr.net/>.

Start with a Radiance Map

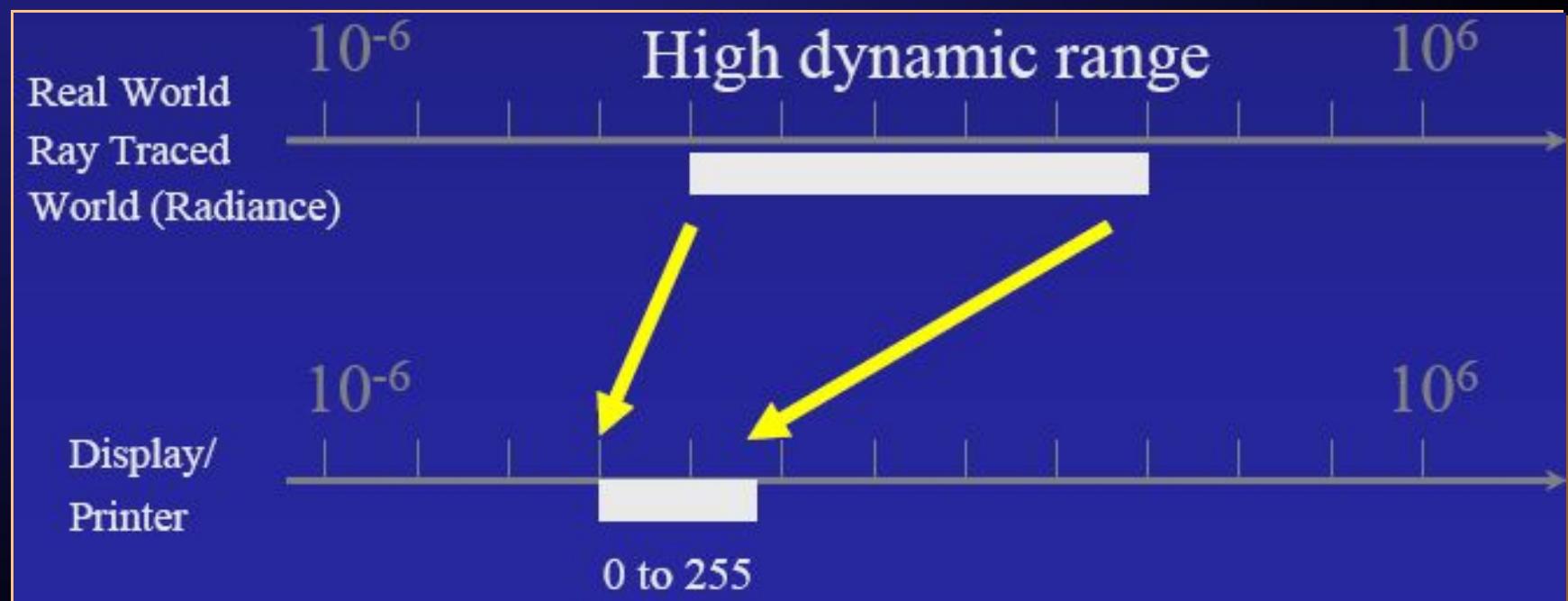
- Need to scale this to a limited linear range...



But this doesn't look too good...

Tone Mapping

- How to do this? Linear scaling? Thresholding to cleverly chosen limits? Give up?



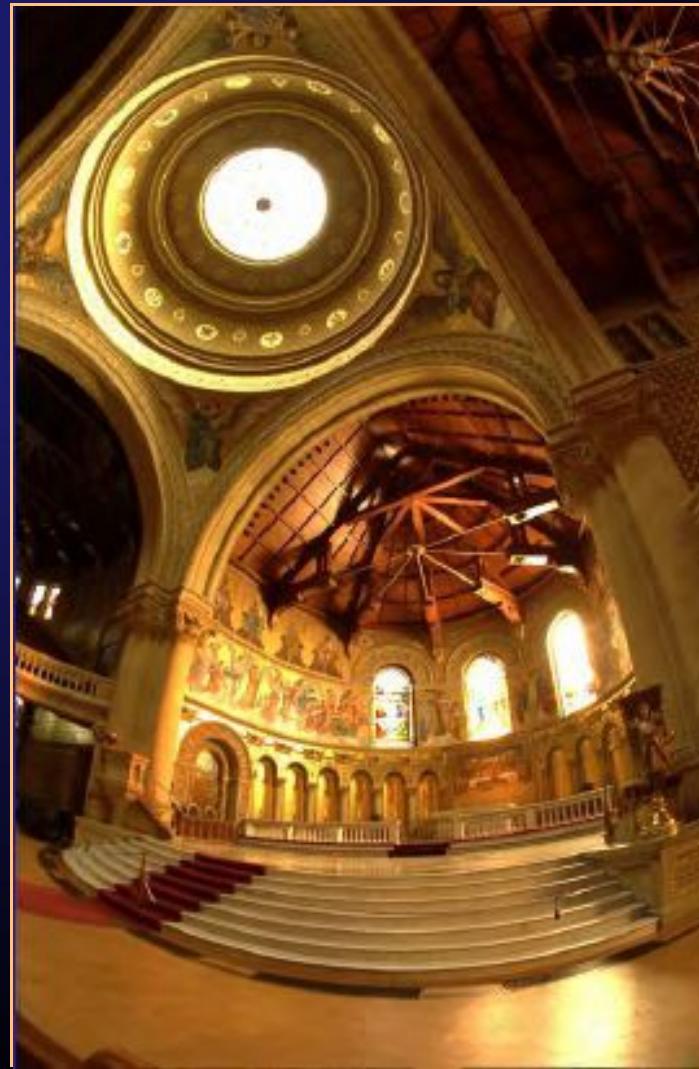
Linear Mapping

- Not very satisfactory.



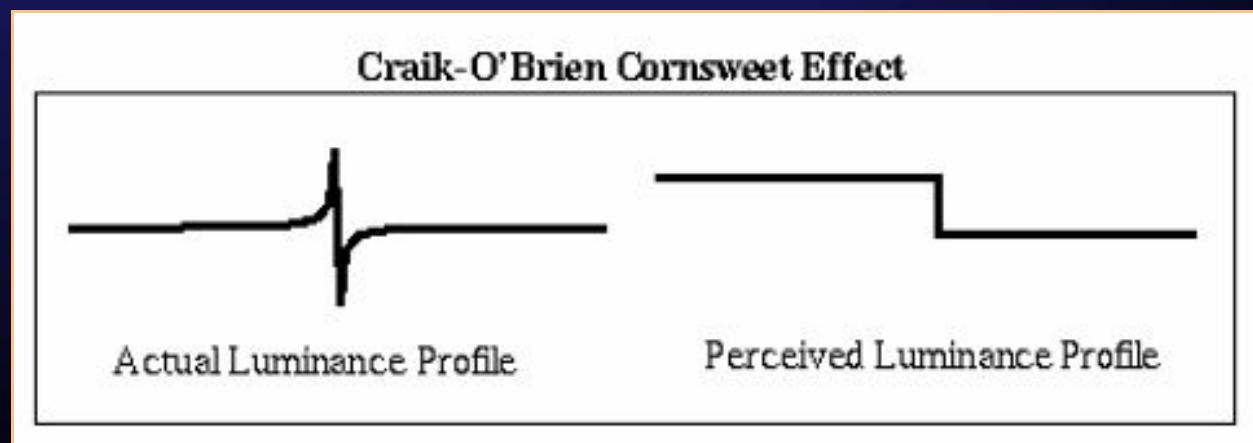
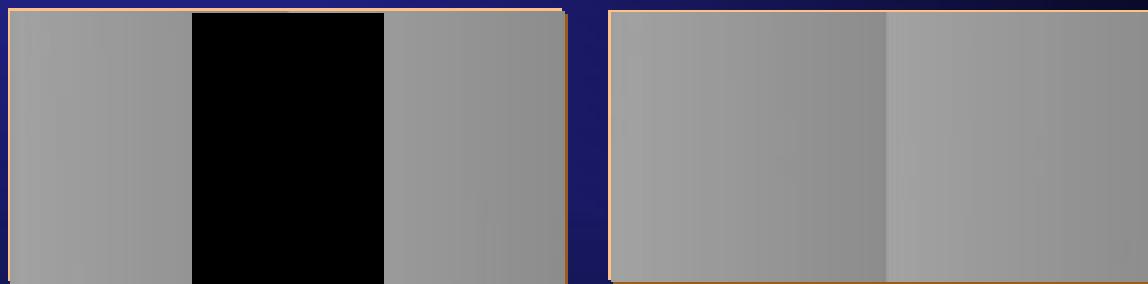
Darkest 0.1% Scaled to Display

- Better, but still washes out brightest areas (windows have no detail).



Exploit “Craik-O’Brien” Effect

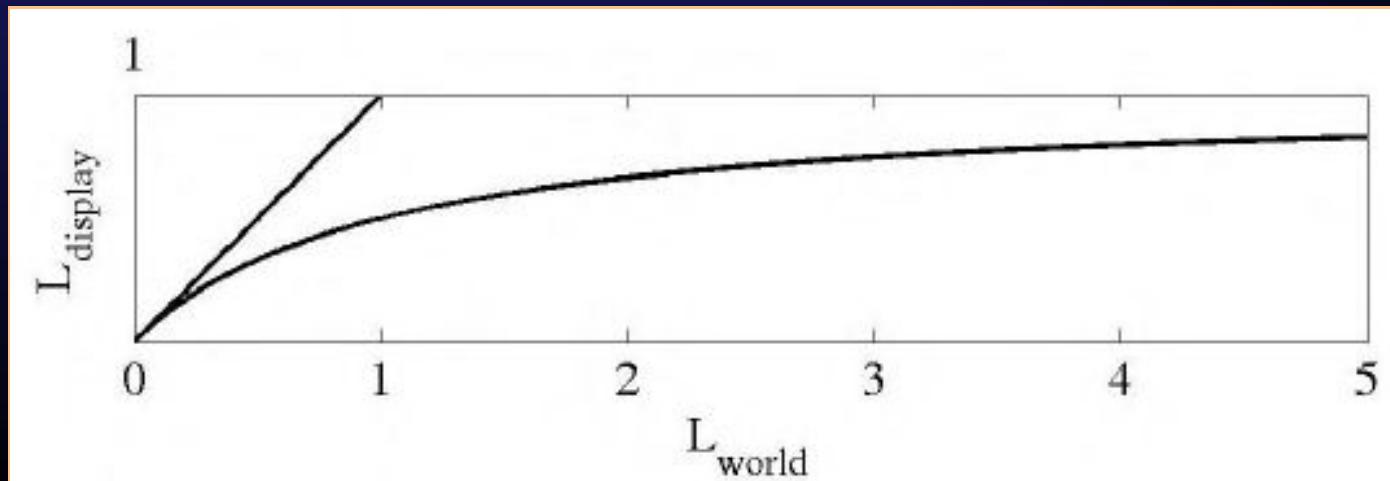
- Same luminance (gray values) but perceived as if different.



Global Operator for Compressing Dynamic Range

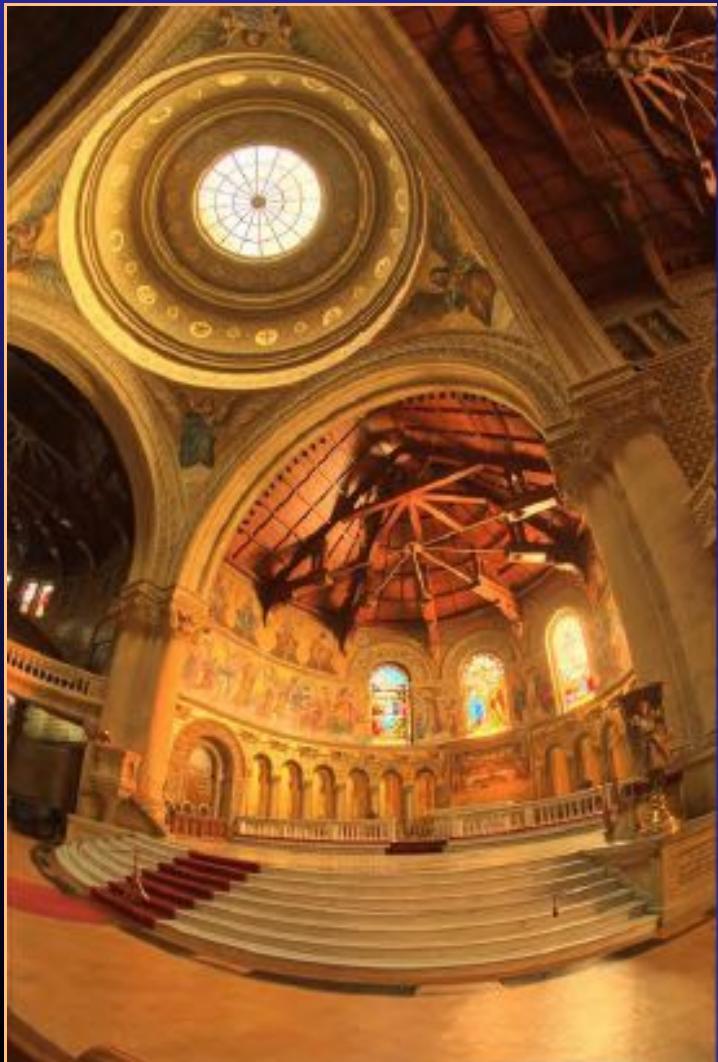
- Compression operator needs to bring everything into 0-255 range (Reinhart et al.) and leave dark areas alone.
- $L_{display}$ is luminance on display; L_{world} is radiosity in world.
- Try :
 - Asymptote at 1 (maximum luminance on display)
 - Derivative of 1 at 0 (linear change near 0)

$$L_{display} = \frac{L_{world}}{1 + L_{world}}$$



Compare!

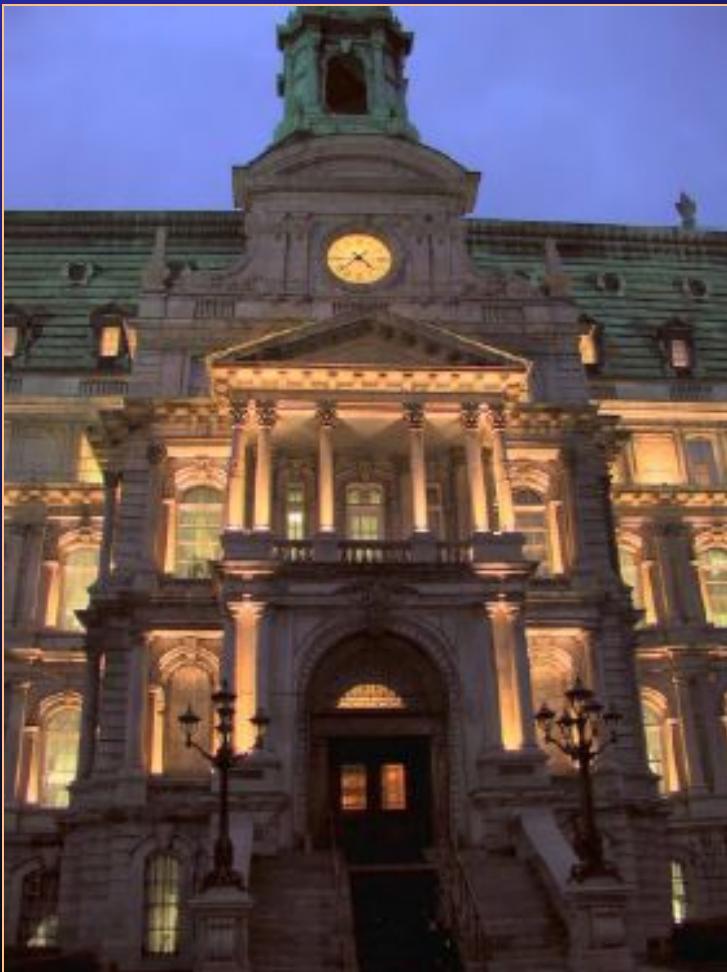
Reinhart Operator



Darkest 0.1% scaled to display



Global Operator Results



Extreme HDR Sun and Sky Image Acquisition

- Fisheye lens view aimed toward zenith.

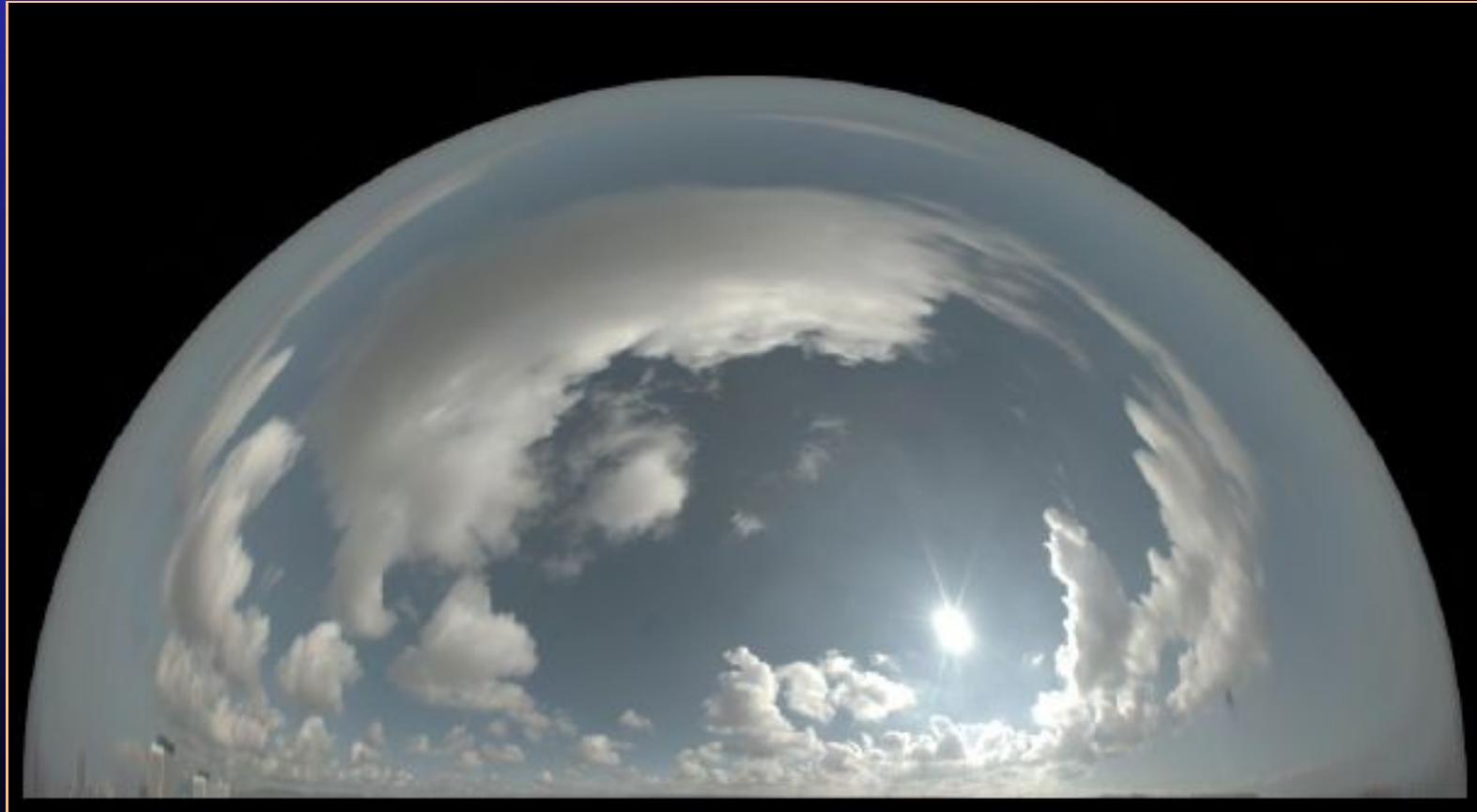


Close-Ups of Sun Image

- Note how we get saturation “flare” for free.



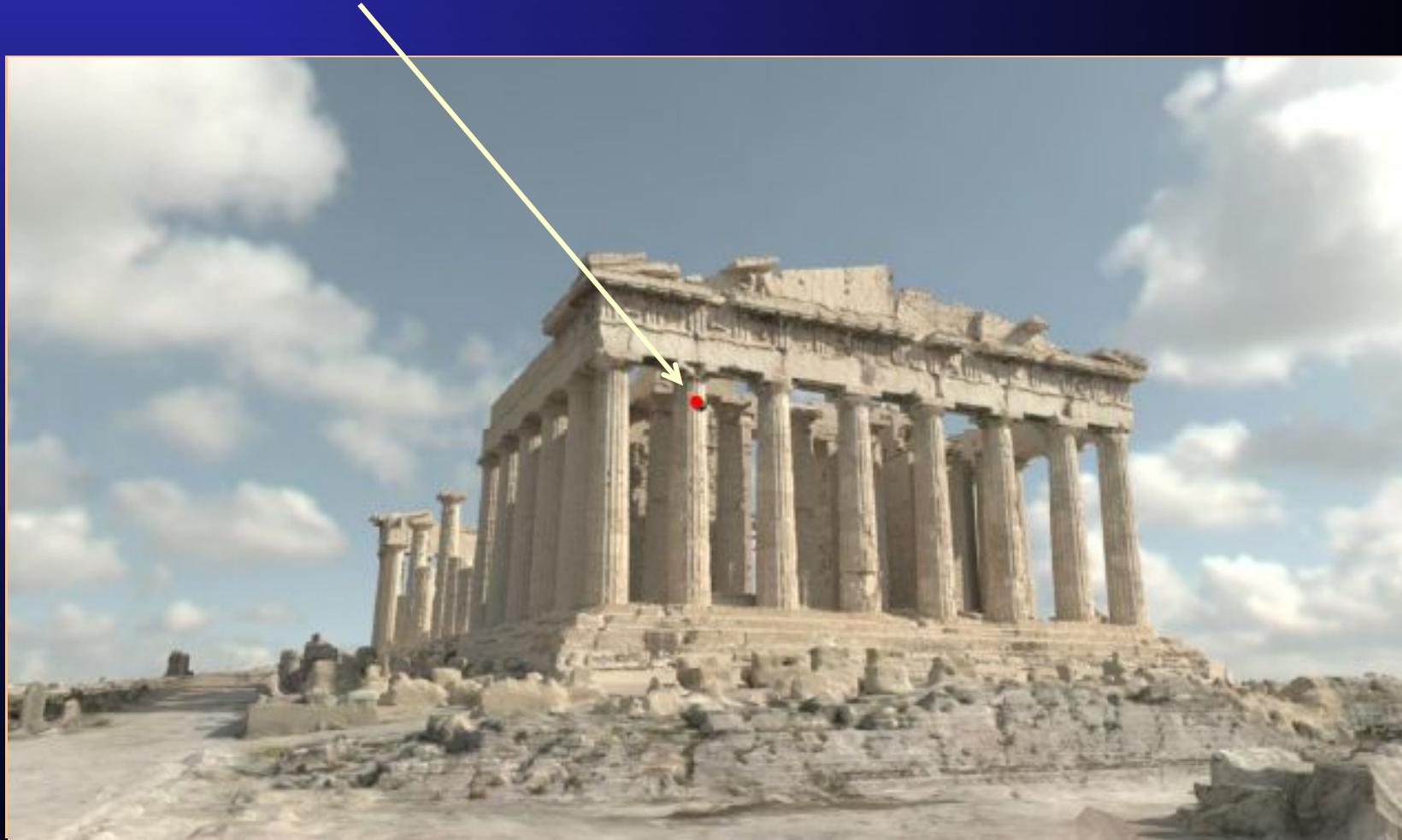
HDR Image Sky Probe



Connection to Radiosity

- HDR environment map (light probe) is a natural hemisphere around the camera location.
- So instead of using a hemicube and the synthetic 5 view camera approach we saw earlier, just...
- Point sample the hemisphere to directly approximate the irradiance!
- Uses HDR radiosity values directly (i.e., samples are HDR).
- No need to correct for camera view (“ $C*D$ ” weight matrix), but sampling function can incorporate this directly (e.g., bias towards center of field).
- Examples...

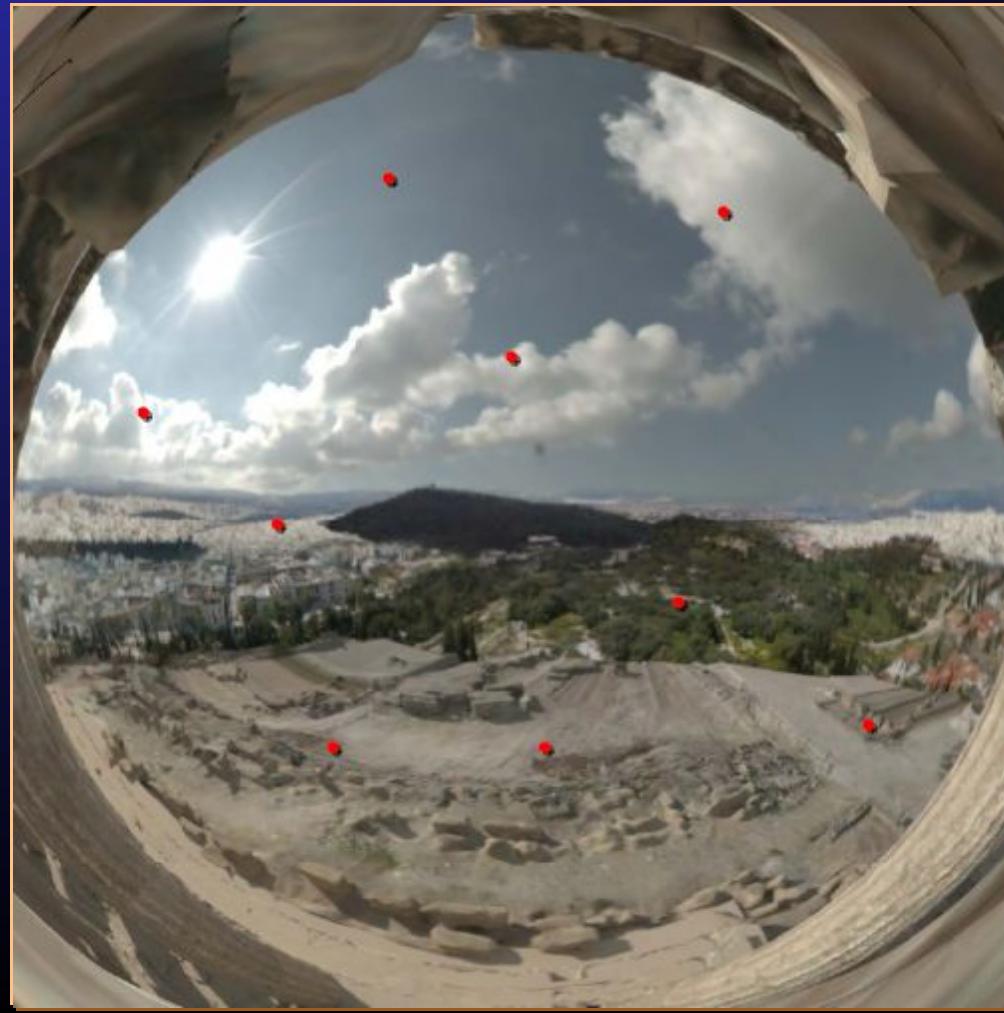
A Sunlit Sample Point (Hemispherical Radiosity Sampled)



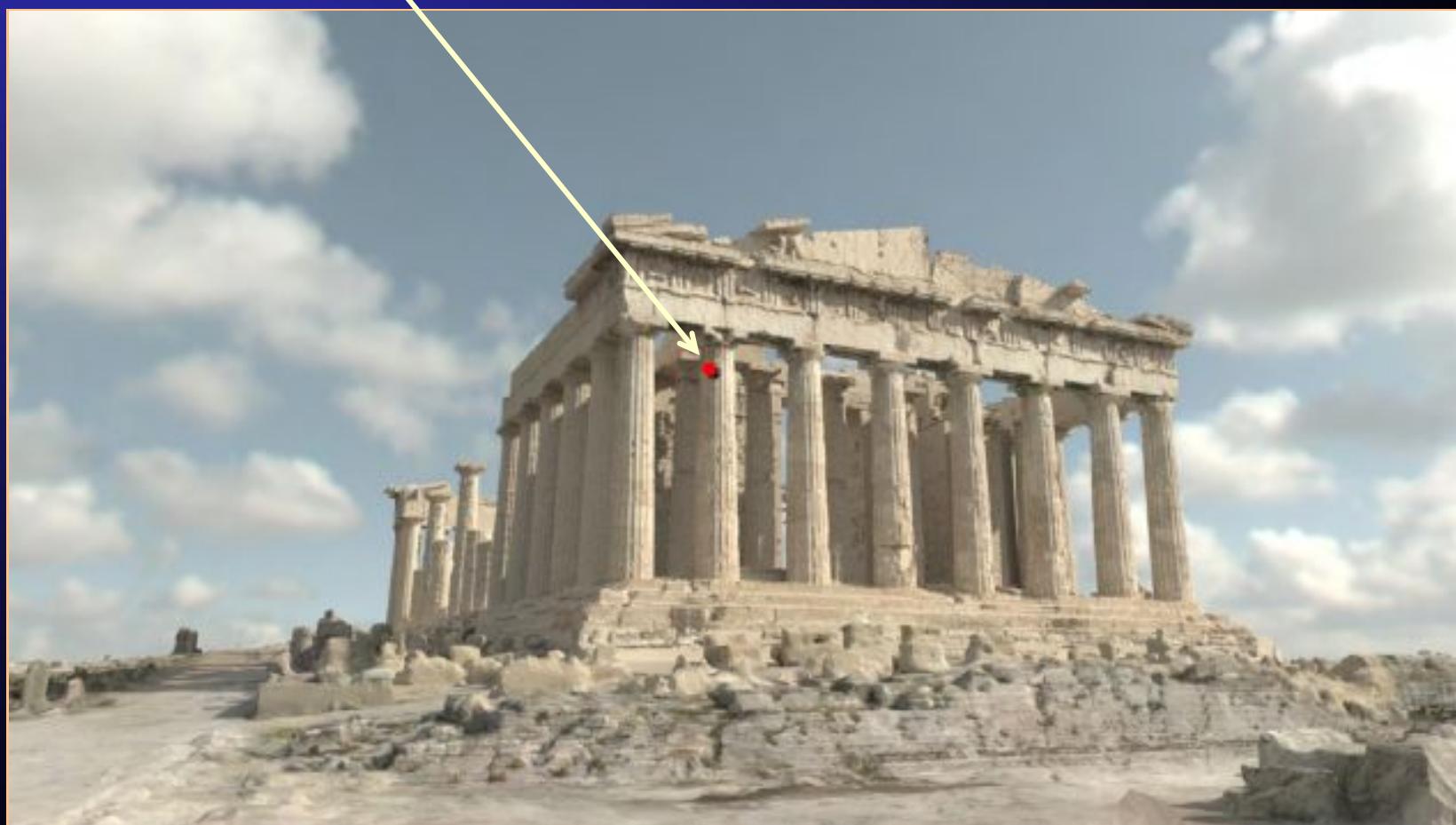
Hemispherical View from Sample



Some Typical Samples from Light Field



A Shadowed Sample Point (Hemispherical Radiosity Sampled)



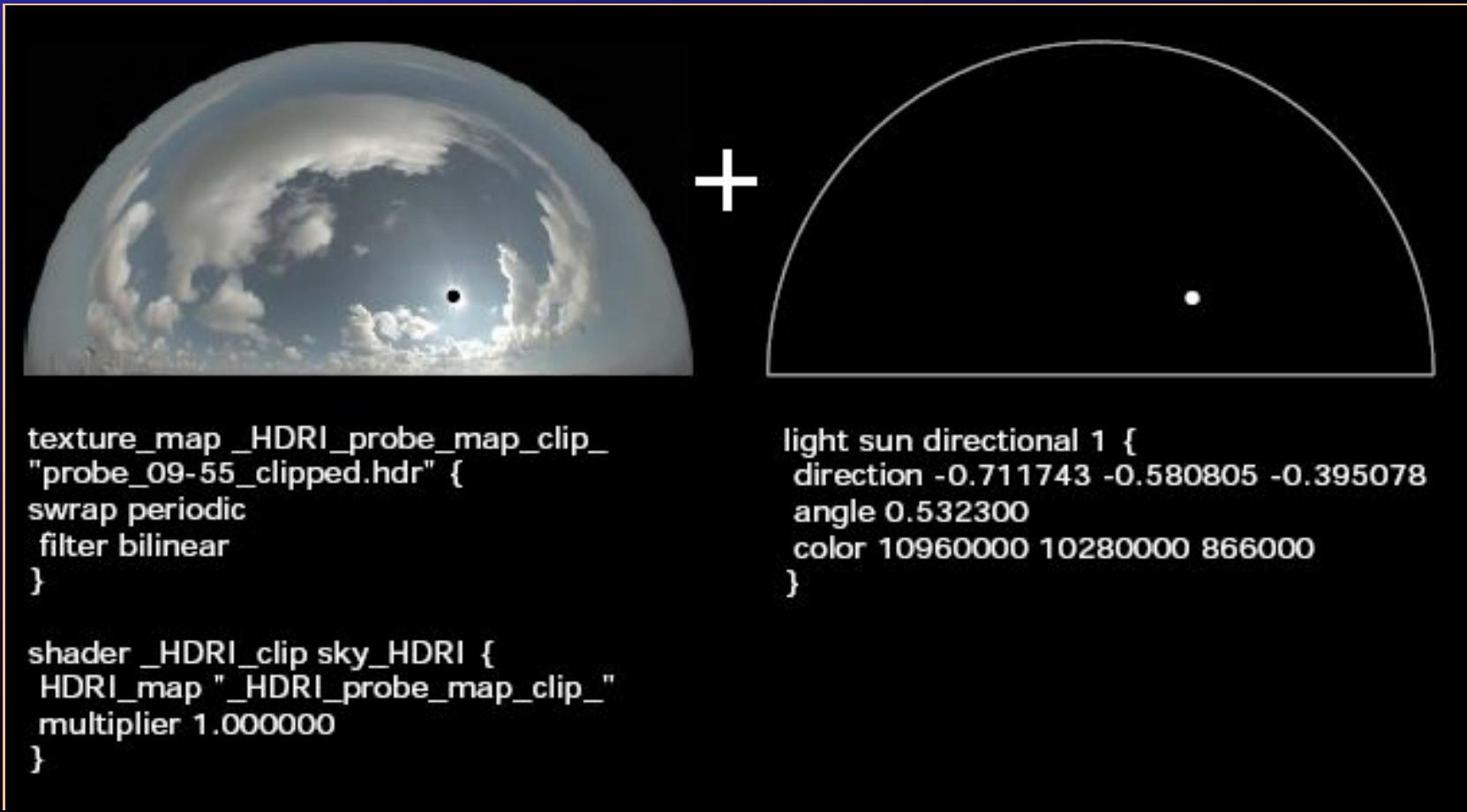
Hemispherical View from Shadowed Sample



Typical Samples from Light Field



Separate Sky (Texture) and Sun (Directional Light) Sources



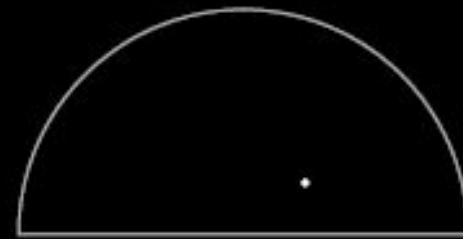
Sky Lighting Only

Lit by sky only, 17 min.



Sun Lighting Only

Lit by sun only, 21 min.



Lit by Both

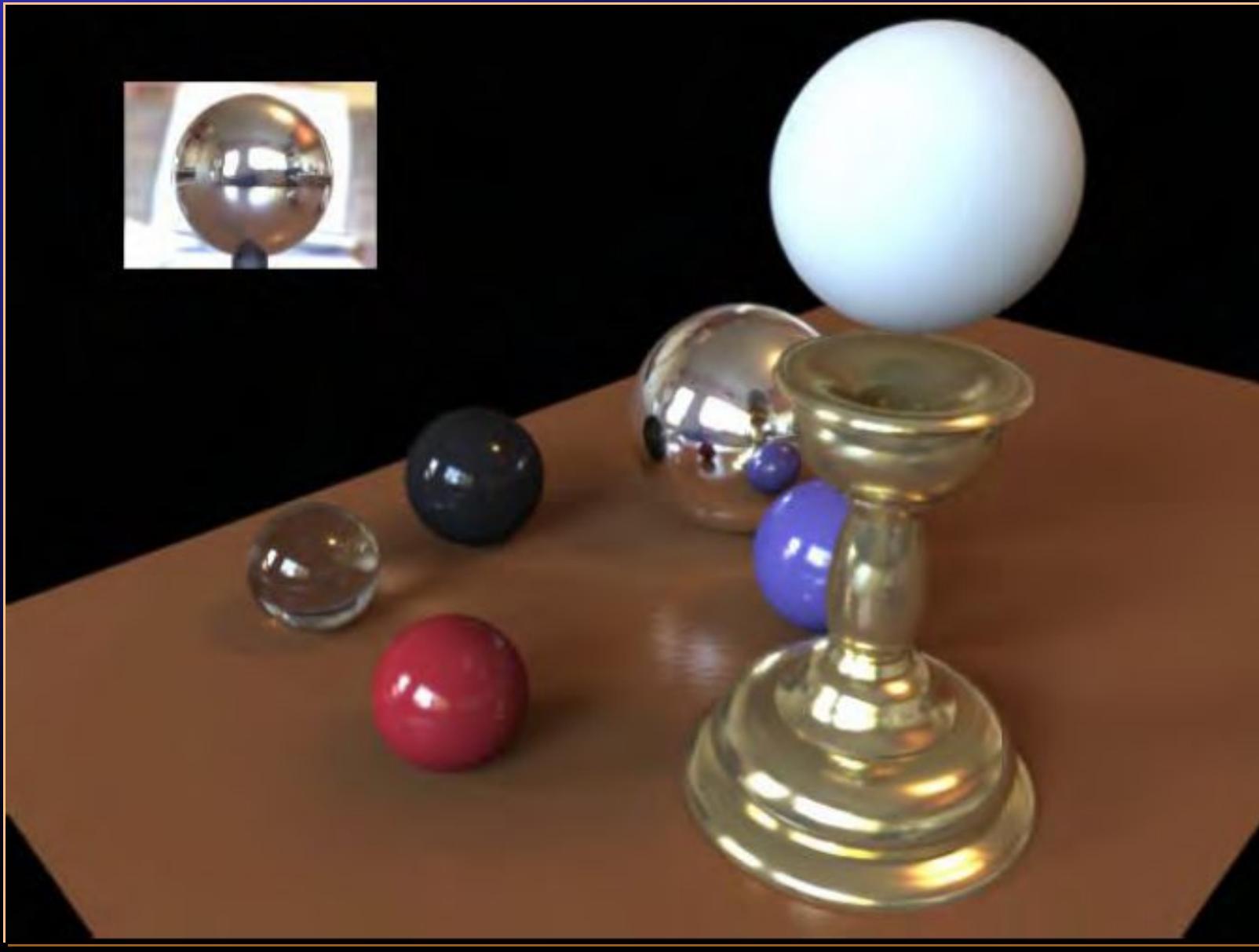
Lit by sun and sky, 25 min



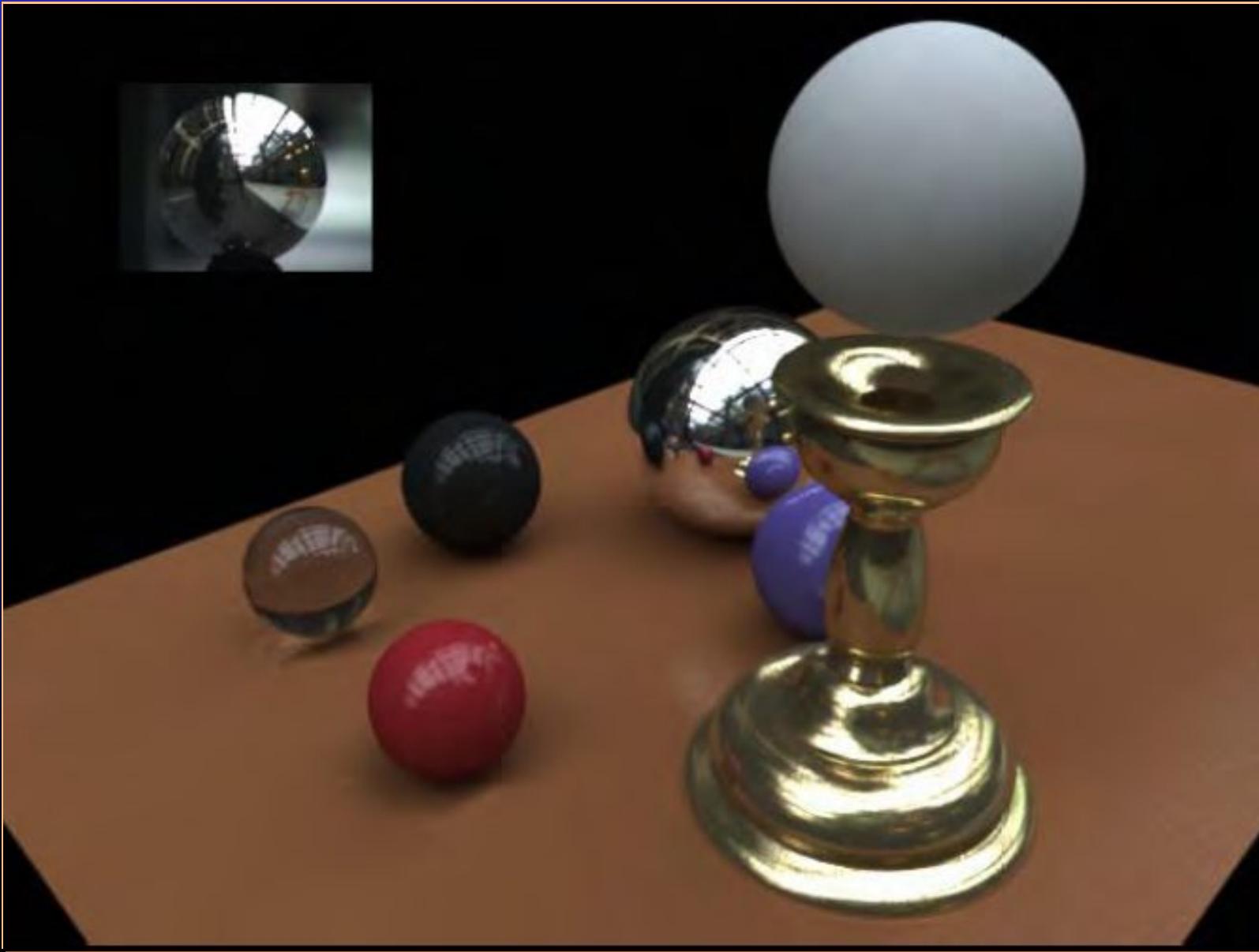
Parthenon with Varied Marina del Ray Light Fields Rendering (& Movie)



Ray-Tracing with Natural Light Environment Map



Change the Light Map; Change the Image



Real and Virtual

- This technique allows synthetic objects (such as the ones we just saw) to be lit by natural (real) light.
- So now we can add synthetic objects such as these to a real scene.

Key to making seamless movie special effects with virtual objects in live shots!

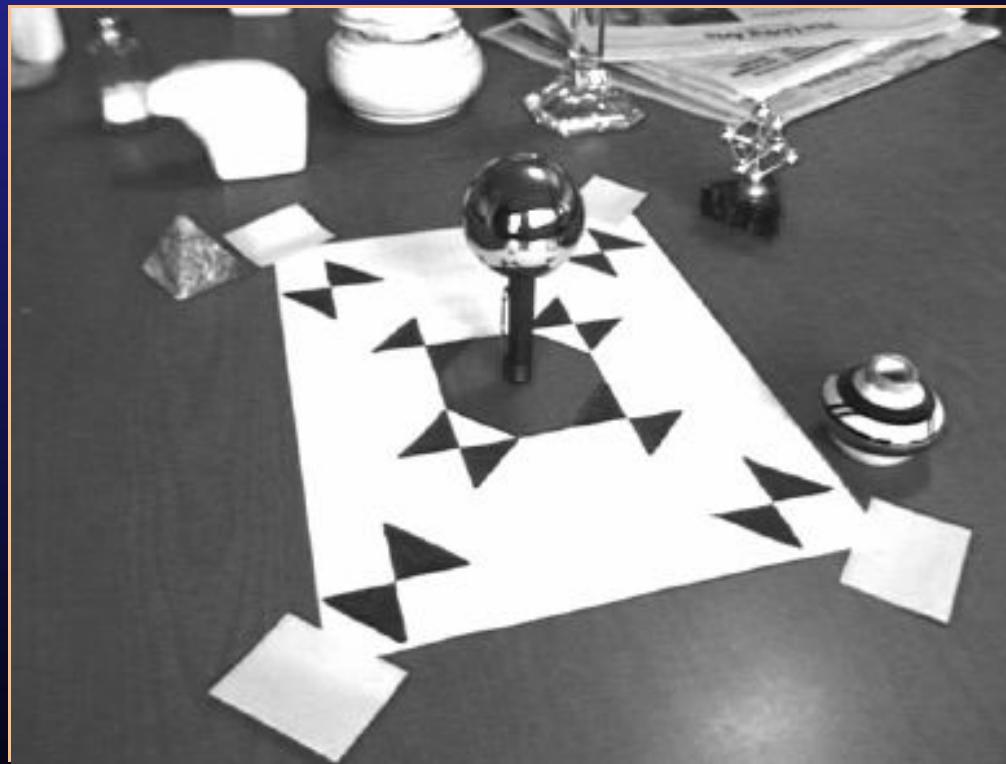
Real Scene Example (Debevec, SIGGRAPH 1998)

- Goal: Place synthetic objects on real table in this context.



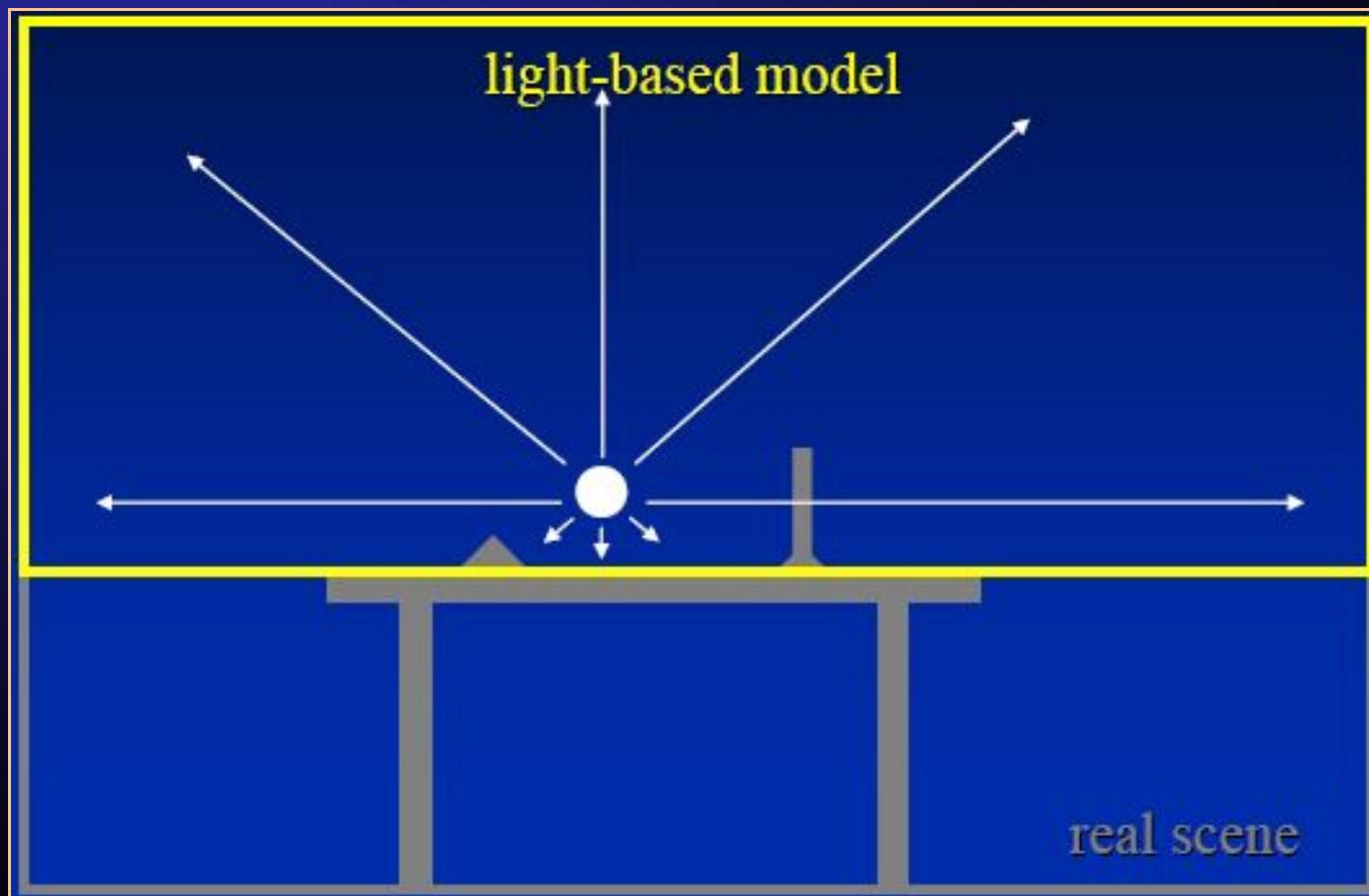
Light Probe on Real Table with Calibration Grid

- Note chrome mirror ball, black/white contrast areas, directional reference.



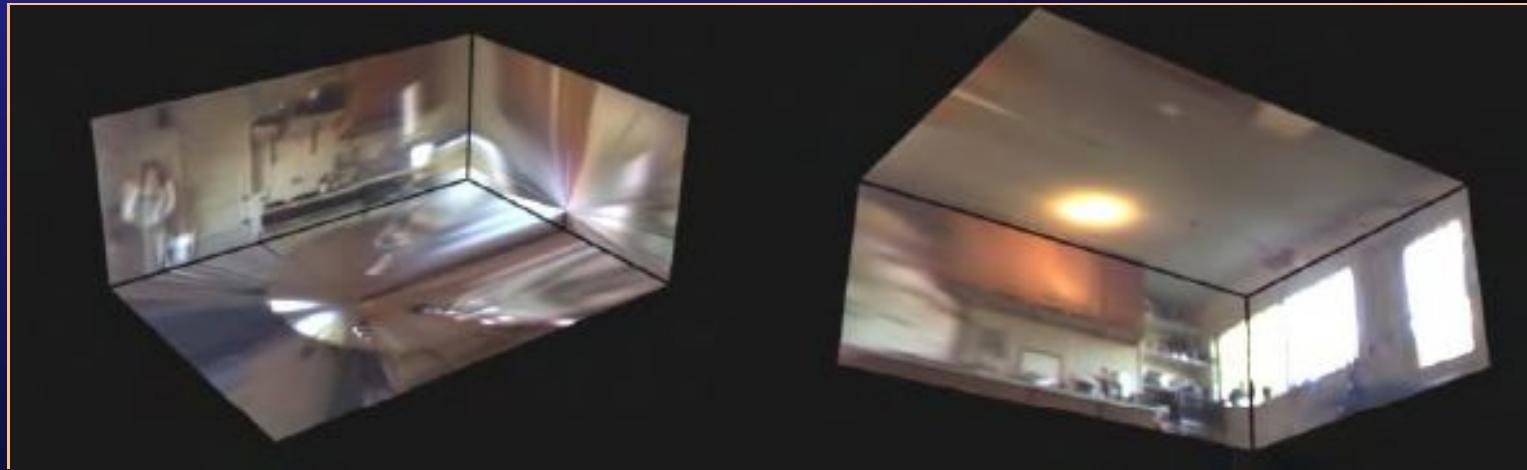
Modeling the Scene

- Need only create the hemisphere above the table surface.



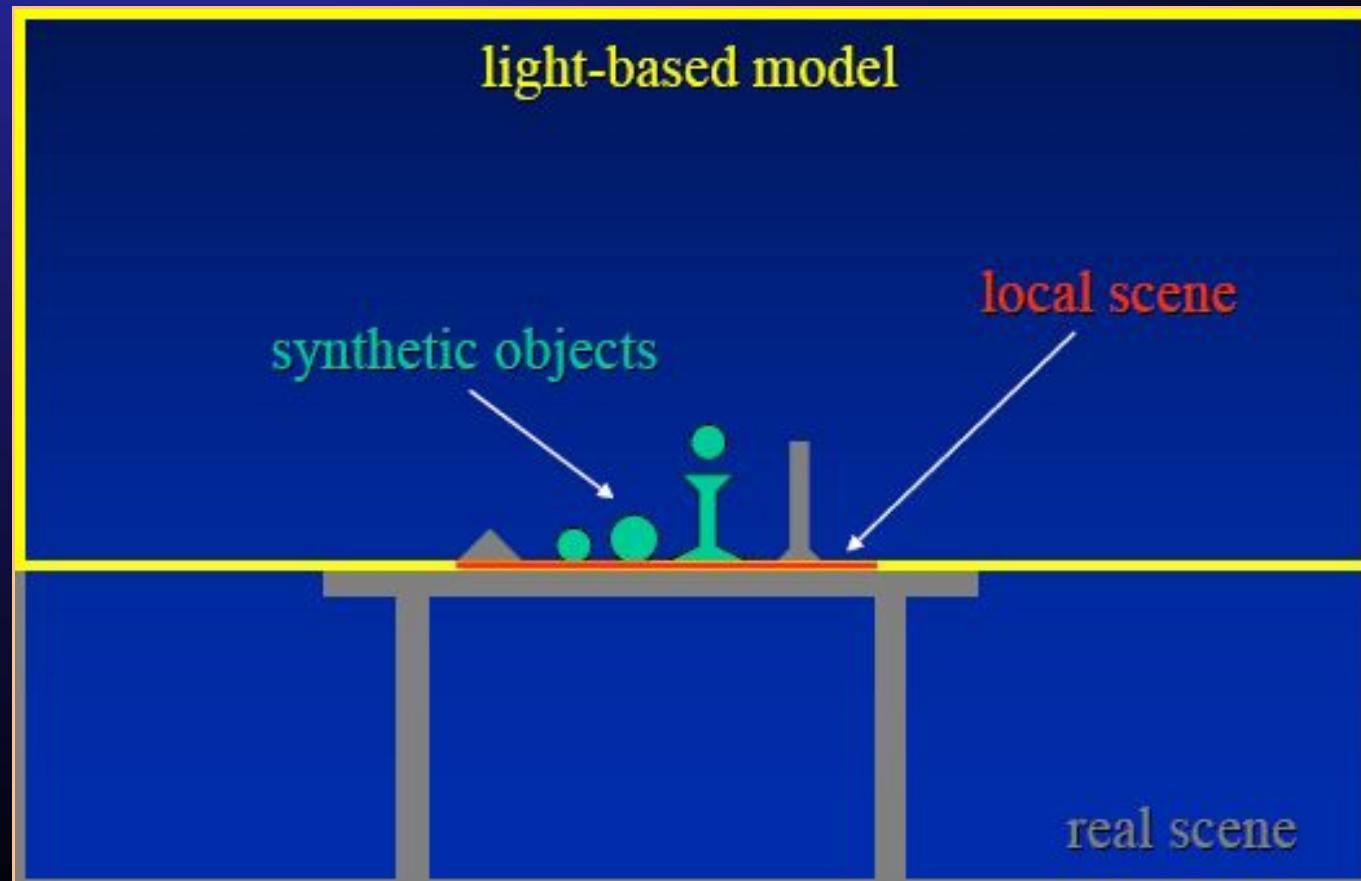
Create the Light-Based (Upper) Room Model

- The 6 surfaces of the environment hemicube.



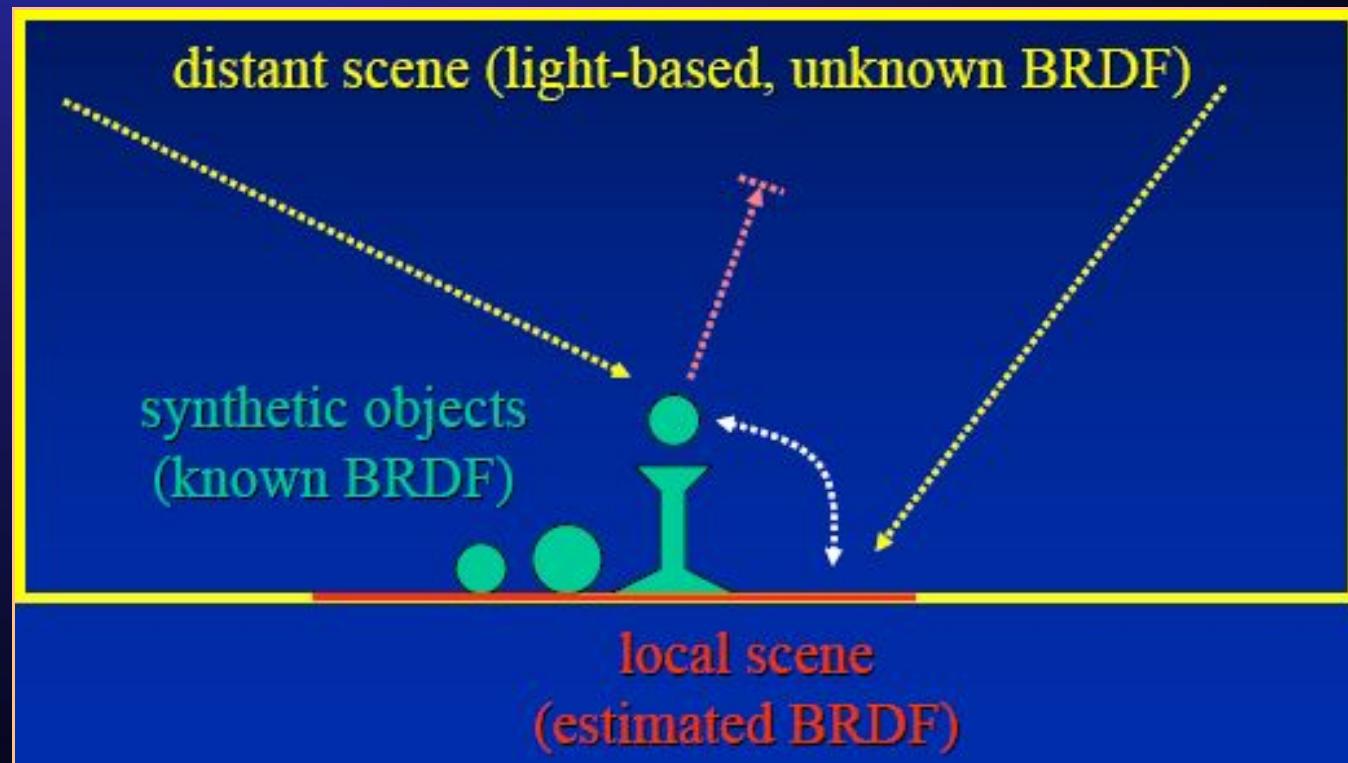
Model the Synthetic Objects

- Position the synthetic objects in the table space.



Lighting Computation

- Global illumination determines new appearance of models.



Background Plate

- LS_b = Original real scene.



Rendering into the Scene

- LS_{obj} = Objects and local scene matched in space.



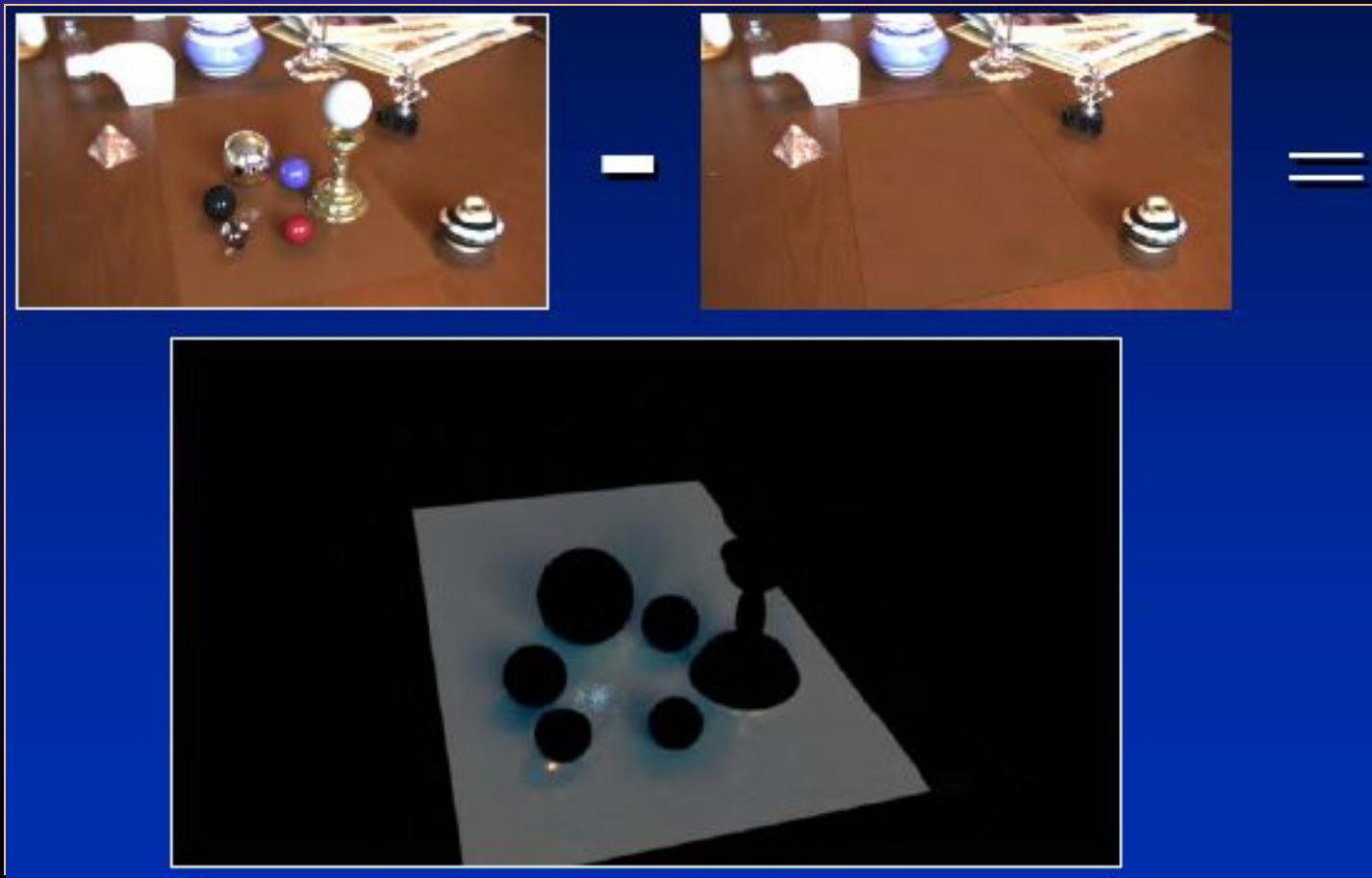
Differential Rendering

- LS_{noobj} = Local scene without objects; illuminated by light model.



Differential Rendering: Difference in Local Scene

- Creates a matte that removes foreground synthetic objects and difference term that adds in global illumination components from model.



Result!

- $LS_{final} = LS_b + (LS_{obj} - LS_{noobj})$



Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading ✓
- Mapping ✓
- 3D Viewing Transformations ✓
- Anti-aliasing & Compositing ✓
- Global Illumination ✓
- Light Fields & HDR ✓
- Path Tracing/Photon Mapping
- Futures

Combining Radiosity and Ray Tracing: Can it be done?

- Radiosity (e.g., by finite element method): accounts for *global diffuse illumination*:
 - Get area lights, soft shadows, color bleeding, participating media.
 - View independent.
 - No arbitrary ambient terms.
 - But expensive meshing and form factor pre-process step.
- Ray tracing accounts for *local specular reflections* and *translucency*.
 - View dependent
- Problem is that these two computations are not really ordered nor independent, but both are essential for natural-looking (arbitrary BRDF, BTDF, and BSSDF) object rendering!

Disadvantage to Both:

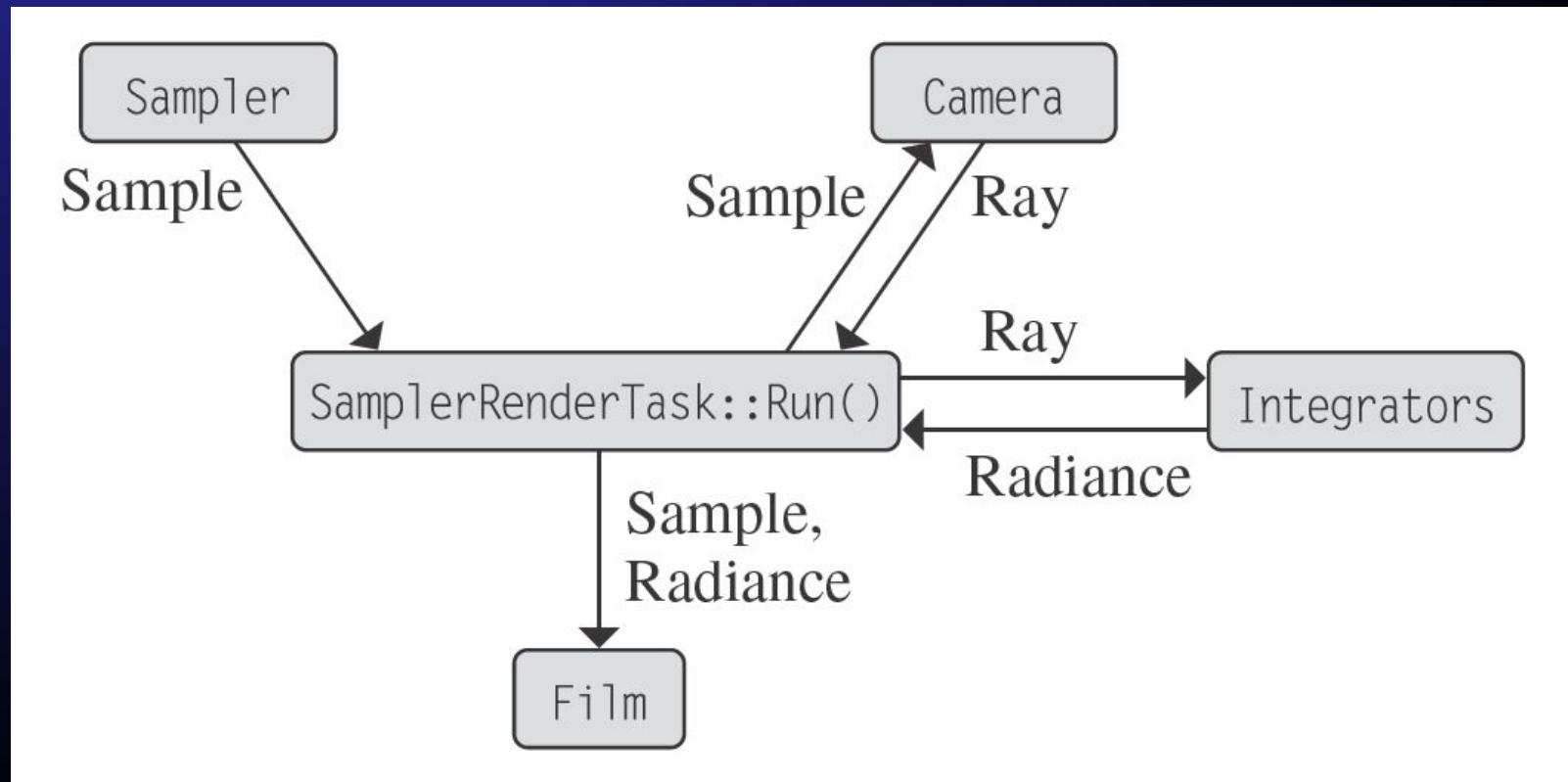
- Finally, neither approach can easily account for **caustics**: bright and shaped areas caused by the focusing and concentration of light by curved (lens-like) surfaces.



- Leads to Monte Carlo Path Tracing and Photon Mapping.

Assembling the Big Picture: Physically Based Rendering

- Need *radiance*, *sampling*, *ray tracing* and *integration!*
- Need to figure out *what*, *when*, and *how*.

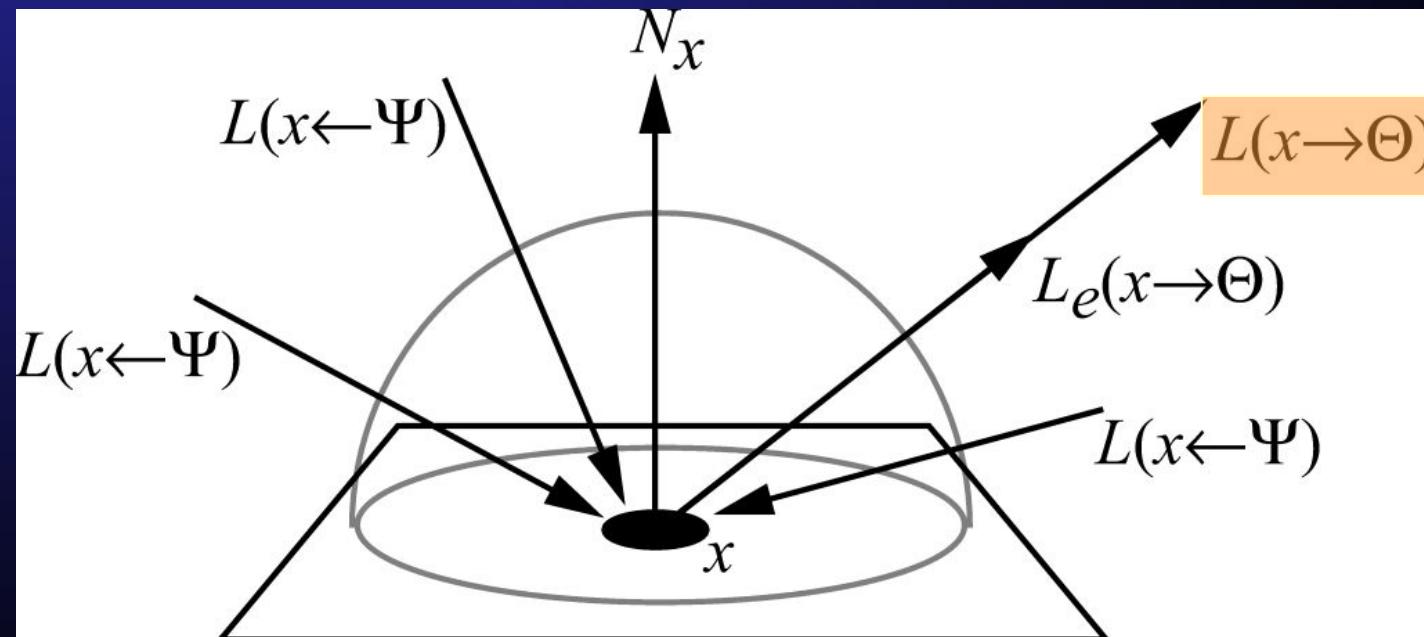


Much of this material adapted from Pharr and Humphreys: *Physically Based Rendering*, MK, 2010

The Rendering Equation: Hemisphere Formulation

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta)$$

$$= L_e(x \rightarrow \Theta) + \int_{\Omega_x} L(x \leftarrow \Psi) f_r(x, \Theta \leftrightarrow \Psi) \cos(\Psi, N_x) d\omega_\Psi$$

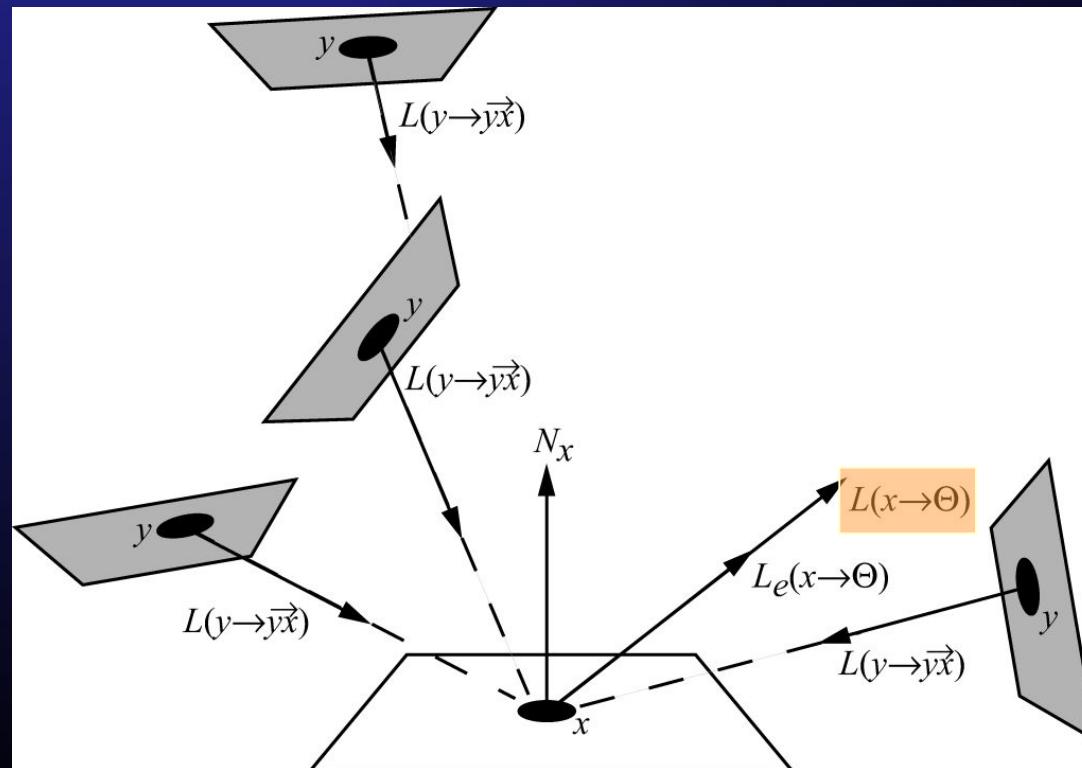


f_r is the Bidirectional Reflectance Distribution Function (BRDF) for the surface at point x

The Rendering Equation: Area Formulation

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta)$$

$$= L_e(x \rightarrow \Theta) + \int_A L(y \rightarrow \vec{yx}) f_r(x, \Theta \leftrightarrow \Psi) \frac{\cos(\Psi, N_x) \cos(-\Psi, N_y)}{r_{xy}^2} V(x, y) dA_y$$



Integration via Randomization Algorithm

- In either of these formulations we have an integral over some domain (hemisphere or surface) that looks quite complicated to solve, since it is inherently *recursive*.
- Solution: *estimate* integral over a *finite number of samples*.
- But how to sample?
- Use a *randomized* algorithm.
- But how to randomize “nicely”?
- We need to look into this question before we can return to the rendering equation itself.

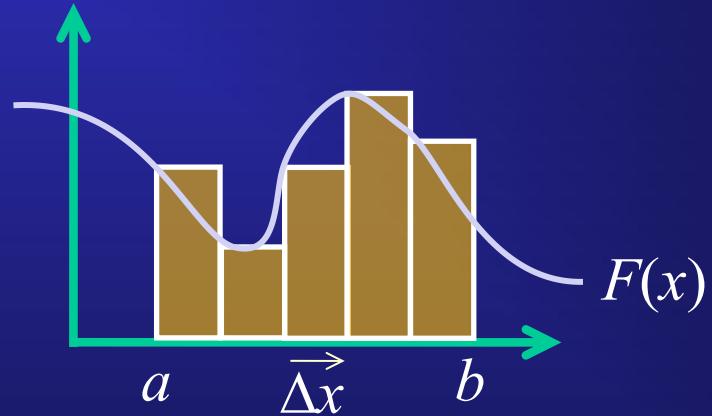


Las Vegas vs. Monte Carlo Random Sampling



- Las Vegas
 - Always gives the same answer
 - E.g., QuickSort (selecting the pivot element randomly gives $O(n \log n)$ complexity, but it always converges to the sorted list whatever the pivot selection method.)
- Monte Carlo
 - Gives the right answer on average
 - Used on functions that are difficult to evaluate or have no closed forms (such as our recursive integrals)
 - Slow convergence $O\left(\frac{1}{\sqrt{N}}\right)$
 - But independent of dimensionality!
 - Artifacts manifest as *noise*

Naïve Integration via Discretization



- Can also sample via Trapezoidal or Gaussian quadrature methods: same general idea.
- Good in 1D
- Error is $O(1/N)$ or $O(1/N^2)$

$$\begin{aligned} I &= \int_a^b f(x) dx \\ &\approx \sum_{i=1}^N f(x_i) \Delta x \\ &= \frac{1}{N} \sum_{i=1}^N f(x_i) \\ \text{Error} &= O\left(\frac{1}{N}\right) \end{aligned}$$

Monte Carlo Terminology

- X, Y, \dots random variables (chosen by a random process).
- Function of a random variable is a random variable , e.g., $Y=f(X)$.
- ξ is a canonical uniform random variable which takes values in domain $[0,1)$ with equal probability.
- x, y, \dots random values.
- \Pr ≡ “Probability of”.
- $P(x)$: Cumulative Distribution Function (CDF): $P(x) = \Pr\{X \leq x\}$
- $p(x)$: Probability Density Function (PDF) $p(x) = \frac{d \Pr(x)}{dx}$

Example for Uniform Random Variable

PDF $p(x)$ properties

$$p(x) \geq 0$$

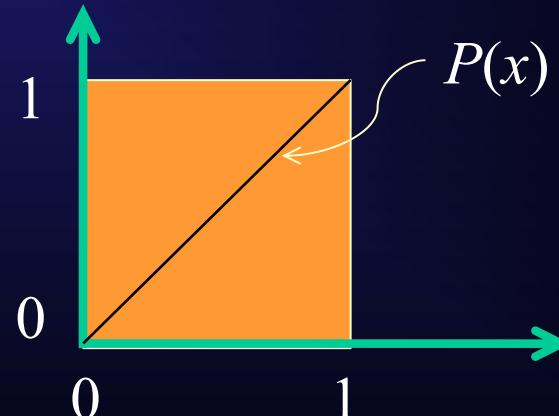
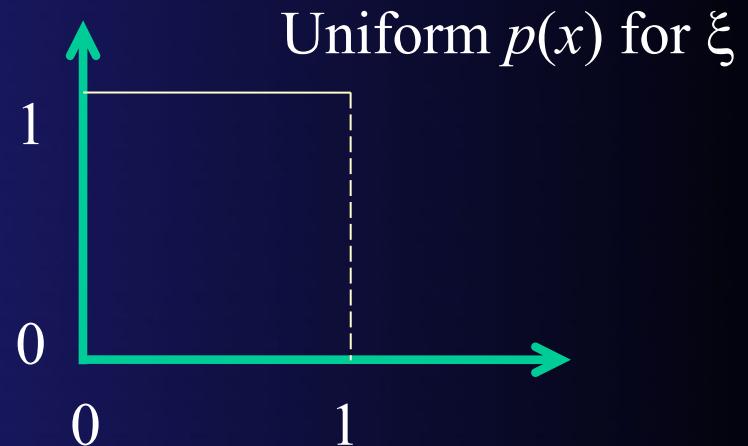
$$\int_D p(x) = 1$$

CDF $P(x)$

$$P(x) = \int_0^x p(x) dx$$

$$P(x) = \Pr(X < x); \quad P(1) = 1$$

$$\begin{aligned} \Pr(x \in [\alpha, \beta]) &= \int_{\alpha}^{\beta} p(x) dx \\ &= P(\beta) - P(\alpha) \end{aligned}$$

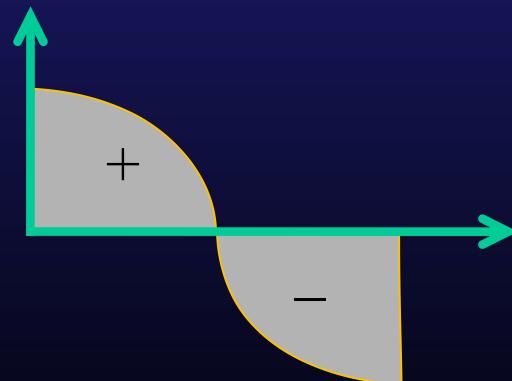


Expected Value

- The *Expected Value* is the mean or average value of the distribution:

$$E_p[f(x)] = \int_D f(x)p(x) dx$$

- E.g., the expected value of a cosine distribution between 0 and π :



$$p(x) = \frac{1}{\pi} \quad \left(\text{since } \int_0^\pi p(x) dx = \int_0^\pi \frac{1}{\pi} dx = \pi \frac{1}{\pi} = 1 \right) \quad \text{so}$$

$$E[\cos(x)] = \int_0^\pi \frac{\cos x}{\pi} dx = \frac{1}{\pi}(-\sin \pi + \sin 0) = 0$$

- ...which is what we would expect from its graph.

Expected Value Properties

- Homogeneity (Linear) $E[af(x)] = aE[f(x)]$
- Additivity (Linear) $E\left[\sum_i f(X_i)\right] = \sum_i E[f(X_i)]$

Monte Carlo Estimator (Uniform random variable)

- Goal: Evaluate $\int_a^b f(x) dx$
- Let X_i be a uniform random variable over $[a,b]$.
- The estimator F_N of $f(x)$ over the interval $[a,b]$ is:

$$F_N = \frac{b-a}{N} \sum_{i=1}^N f(X_i)$$

- The PDF $p(x)$ must be a constant and integrate to 1 over $[a,b]$. i.e.,
 $p(x) = 1/(b-a)$ {check: $(b-a)p(x) = 1$ }
- The expected value $E[F_N]$ of the estimator F_N is indeed the integral:

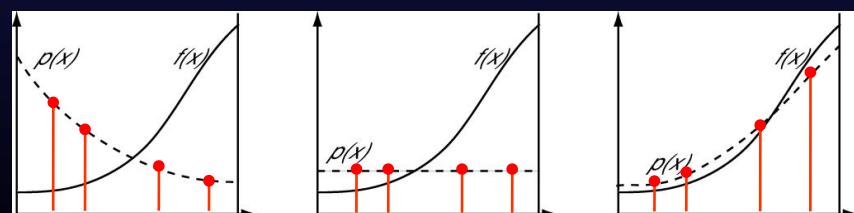
$$\begin{aligned} E[F_N] &= E\left[\frac{b-a}{N} \sum_{i=1}^N f(X_i)\right] = \frac{b-a}{N} \sum_{i=1}^N E[f(X_i)] \\ &= \frac{b-a}{N} \sum_{i=1}^N \int_a^b f(x)p(x) dx = \frac{1}{N} \sum_{i=1}^N \int_a^b f(x) dx \\ &= \int_a^b f(x) dx \end{aligned}$$

Monte Carlo Estimator (Arbitrary random variable)

- If X is a random variable drawn from an arbitrary PDF $p(x)$, then the estimator is:

$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}$$

- Choosing a “good” PDF is important. (Imagine $p(x_i)$ are “weights” that integrate to 1.)



$$\begin{aligned} E[F_N] &= E\left[\frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N} \sum_{i=1}^N E\left[\frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N} \sum_{i=1}^N \int_a^b \frac{f(x_i)}{p(x_i)} p(x) dx \\ &= \frac{1}{N} \sum_{i=1}^N \int_a^b f(x) dx \\ &= \int_a^b f(x) dx \end{aligned}$$

Monte Carlo Estimator (Multiple Dimensions)

- Suppose that N samples X_i are drawn from an arbitrary multidimensional (“joint”) PDF $p(X)$, and the 3D integral is:

$$\int_{x_0}^{x_1} \int_{y_0}^{y_1} \int_{z_0}^{z_1} f(x, y, z) dx dy dz$$

- If samples are chosen uniformly from the box (x_0, y_0, z_0) to (x_1, y_1, z_1) , then PDF $p(X)$ is the constant

$$\frac{1}{(x_1 - x_0)} \frac{1}{(y_1 - y_0)} \frac{1}{(z_1 - z_0)}$$

- And the estimator is

$$\frac{(x_1 - x_0)(y_1 - y_0)(z_1 - z_0)}{N} \sum_i f(X_i)$$

Monte Carlo Properties

- The number of samples can be chosen independently of the dimensionality of the integrand.
- (Standard numerical quadrature is exponential in the dimension!)
- The integral domain need not be a Cartesian box, but any domain (e.g., a hemisphere) as long as we can establish a suitable PDF.
- Monte Carlo convergence rate is $O\left(\frac{1}{\sqrt{N}}\right)$.
- (Standard quadrature converges faster in 1D, but exponentially worse in higher dimensions.)
- Monte Carlo is therefore *the only practical numerical integration technique for high dimensional integrals!*
- Moreover, in computer graphics rendering using path tracing, the integrands are *infinite dimensional!*

So what do we need for Monte Carlo Integration?

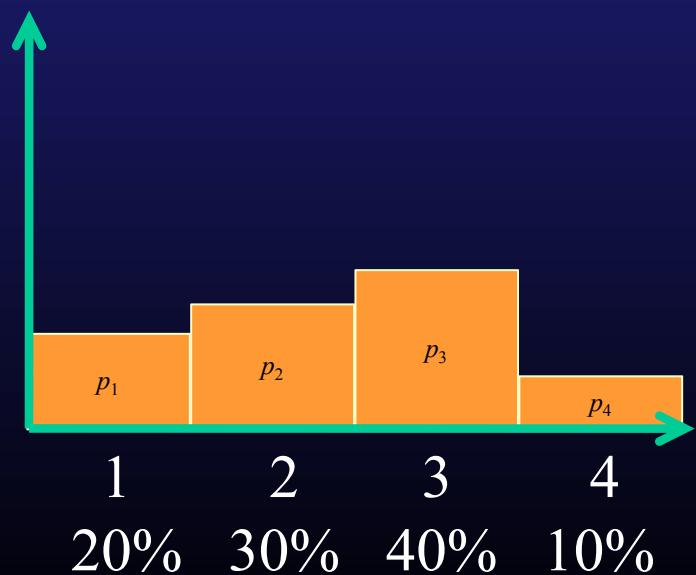
- N samples X_i and an “appropriate” PDF $p(X_i)$:

$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}$$

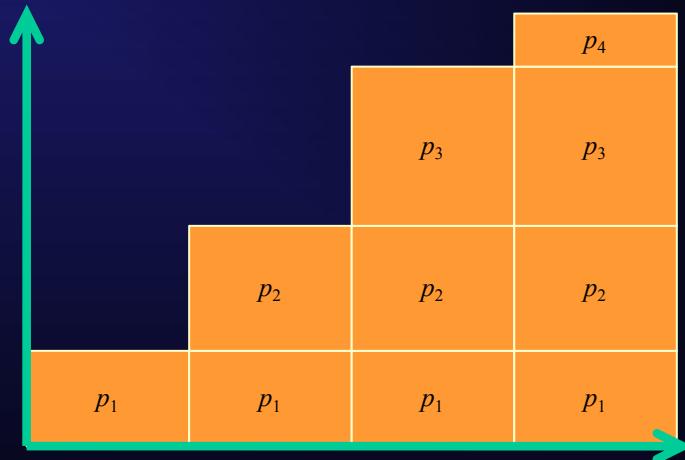
- Sampling methods include
 - Discrete bins
 - Inversion
 - Rejection
 - Metropolis (not done here)
- Assume uniform random variable $\xi \in [0,1]$ generated by a pseudo-uniform random number generator.

Toy Example: Discrete Bins

Generate the following PDF using ξ (note $\sum_{i=1}^4 p_i = 1$):



Stack PDF to get CDF:
 $P_i = \sum_{j=1}^i p_j$



Toy Example: Discrete Bins

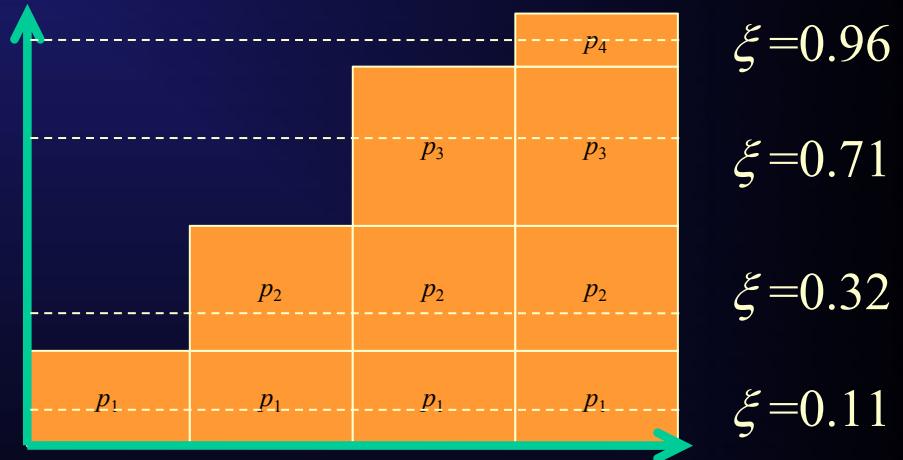
Generate a value for ξ ...

Suppose $\xi \in [0.9, 1)$,
then are in p_4

Suppose $\xi \in [0.5, 0.9)$,
then are in p_3

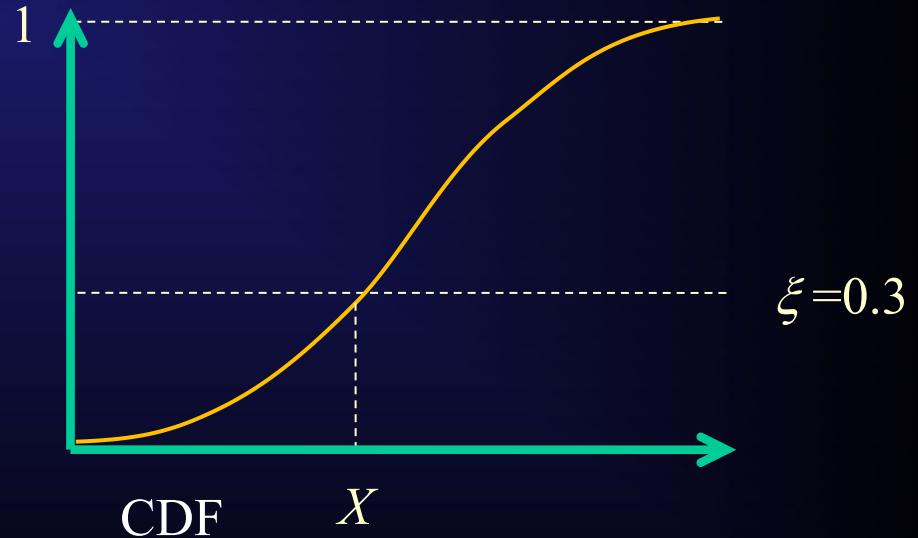
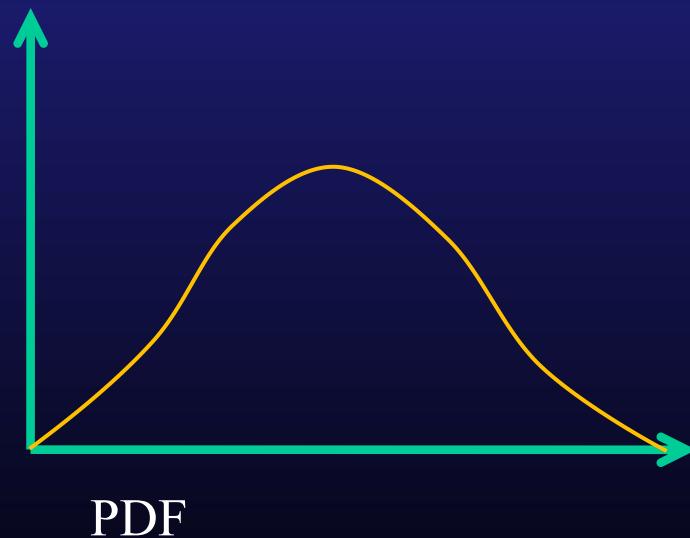
Suppose $\xi \in [0.2, 0.5)$,
then are in p_2

Suppose $\xi \in [0, 0.2)$,
then are in p_1



Inversion Method (If CDF is continuous and invertible)

- Get PDF $p(x)$
- Compute CDF $P(x) = \int_0^x p(x') dx'$
- Compute inverse $P^{-1}(x)$
- Get ξ
- Compute $X = P^{-1}(\xi)$



Example: Power Distribution (arises from microfacet model)

Sampling from the power distribution $p(x) \propto x^n$

PDF $p(x) = cx^n$

Since $\int_0^1 cx^n dx = 1$, $c \frac{x^{n+1}}{n+1} \Big|_0^1 = 1$ so

$$c = n + 1$$

Hence :

$$P(x) = \int_0^1 p(x) dx = x^{n+1}$$

and

$$P^{-1}(x) = \sqrt[n+1]{x}$$

Thus

$$X = \sqrt[n+1]{\xi}$$

The Rejection Method (Throw darts)

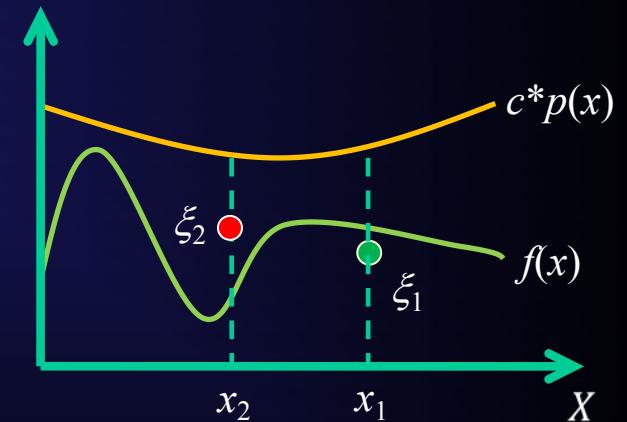
- Can't integrate $f(x)$ to find its PDF?
- Can't analytically invert its CDF?
- $p(x)$ is “ugly” or “weird” or just complicated?
- Then throw darts at it!
- Suppose $f(x) < (c p(x))$ for some constant scalar c and we know how to sample from $p(x)$.

loop forever:

sample X from p 's distribution

if $\xi < f(X) / (c p(X))$ then

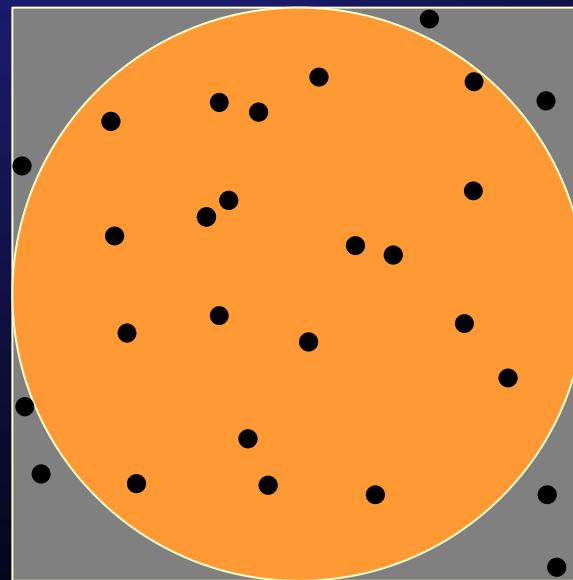
return X



- Works for any number of dimensions.

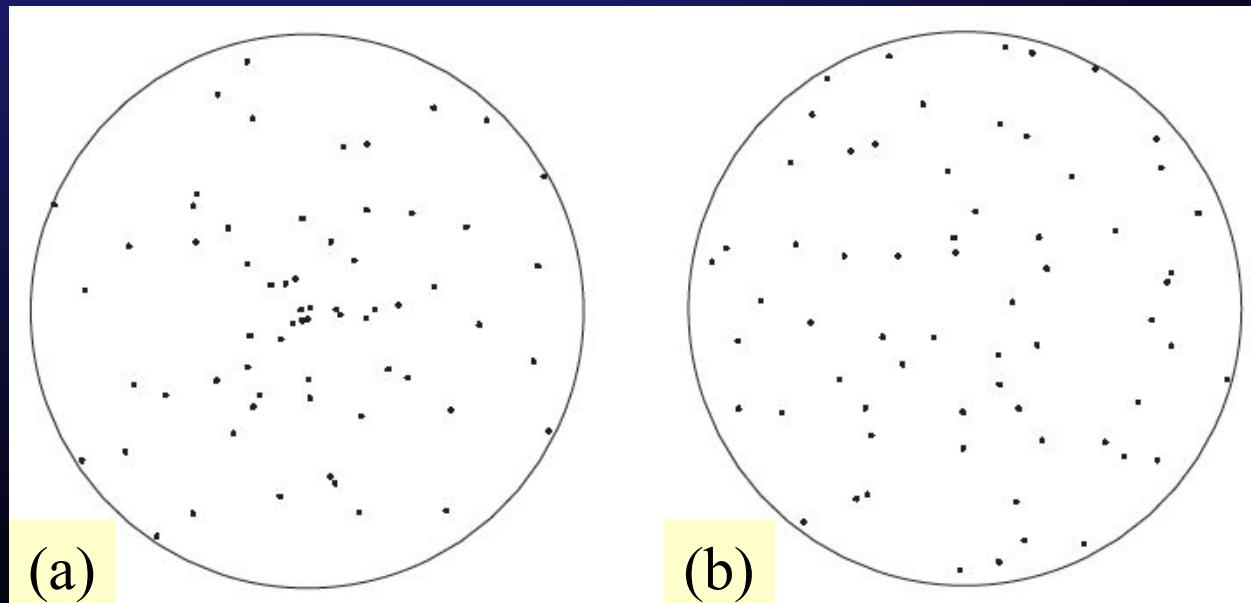
Examples of Rejection Sampling

- Uniformly sample a disk (embedded in a unit square) by rejection sampling.
- (Result IS uniform but “clumpy” in undesirable ways. Recall stratified sampling discussion.)



Direct Approach to Disk Sampling

- Compute samples *uniformly* over a unit disk in *polar* coordinates.
- An obvious sampling strategy is $(r, \theta) = (\xi_1, 2\pi\xi_2)$ but this causes samples to “clump” toward the disk center (a).
- Without proof, a good uniform sampling function for the unit disk is (b): $(r, \theta) = (\sqrt{\xi_1}, 2\pi\xi_2)$
- (The $\sqrt{\xi_1}$ counteracts the disk “clumping”.)



$$(r, \theta) = (\xi_1, 2\pi\xi_2)$$

$$(r, \theta) = (\sqrt{\xi_1}, 2\pi\xi_2)$$

Monte Carlo Integration: Sampling over a Hemisphere

- We need to sample irradiance due to a light source over a hemisphere:

$$\begin{aligned} I &= \int_{\Omega} L(\Theta) \cos \theta d\omega_{\Theta} \\ &= \int_0^{2\pi} \int_0^{\frac{\pi}{2}} L(\Theta) \cos \theta \sin \phi d\phi \\ E[I] &= \frac{1}{N} \sum_{i=1}^N \frac{L(\Theta_i) \cos \theta_i \sin \phi_i}{p(\Theta_i)} \end{aligned}$$

- How to do this?

Sampling over Spheres and Hemispheres

- Uniform direction samples over a complete sphere:

$$x = \cos(2\pi\xi_2)\sqrt{1-z^2} = \cos(2\pi\xi_2) 2\sqrt{\xi_1(1-\xi_1)}$$

$$y = \sin(2\pi\xi_2)\sqrt{1-z^2} = \sin(2\pi\xi_2) 2\sqrt{\xi_1(1-\xi_1)}$$

$$z = 1 - 2\xi_1$$

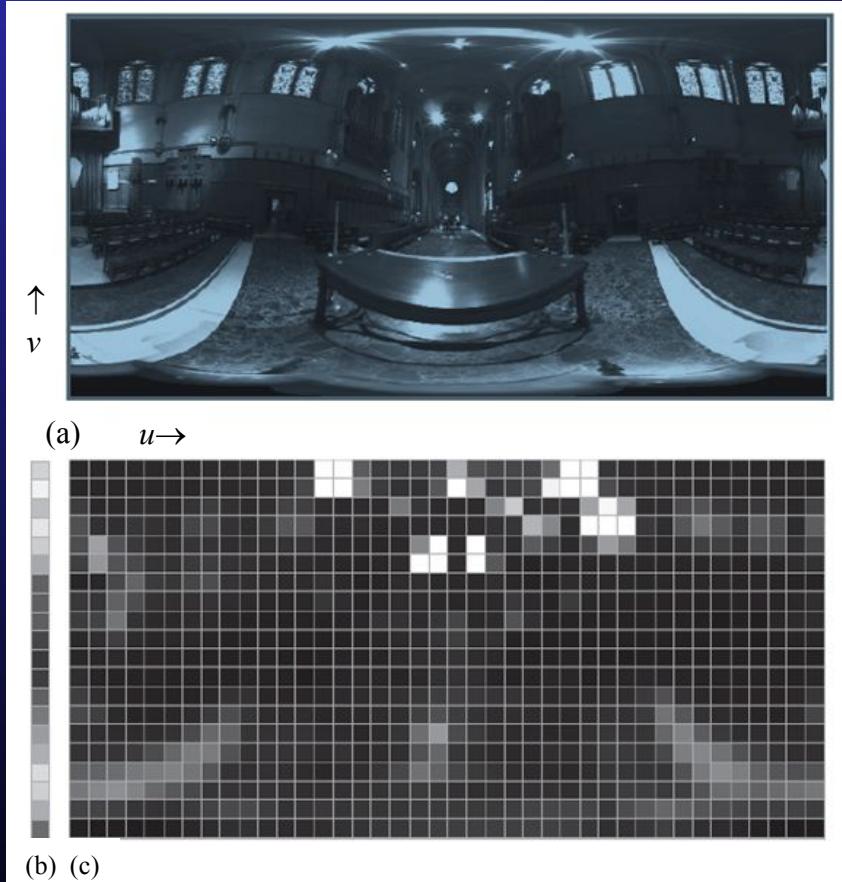
- And uniform samples over a hemisphere:

$$x = \sin \theta \cos \phi = \cos(2\pi\xi_2)\sqrt{1-\xi_1^2}$$

$$y = \sin \theta \sin \phi = \sin(2\pi\xi_2)\sqrt{1-\xi_1^2}$$

$$z = \cos \theta = \xi_1$$

Sampling 2D Piecewise-Constant Functions (Used for texture maps and environment maps)



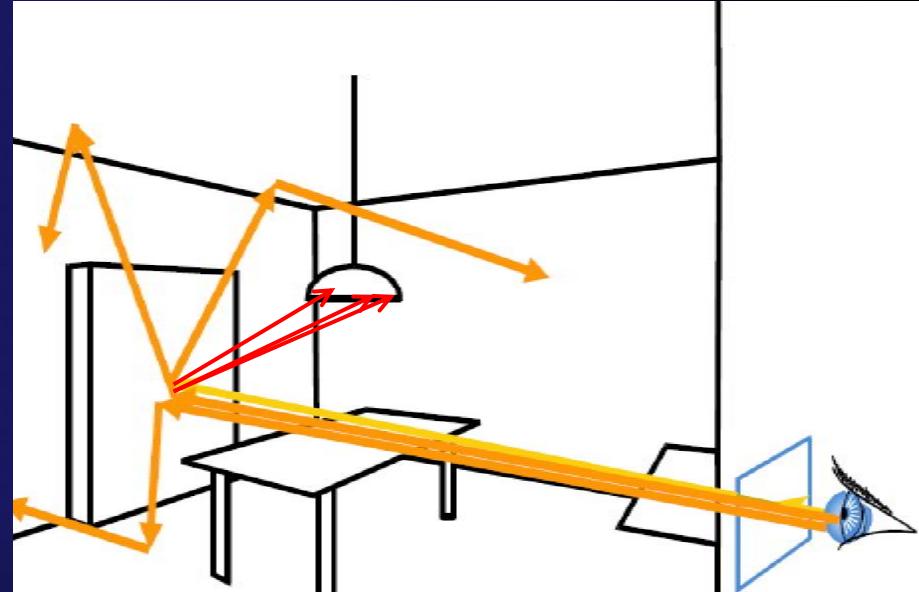
- (a) The original HDR environment map $[u,v]$.
- (b) A low-resolution version of the row-wise $[v]$ density function.
- (c) The piece-wise constant distributions for rows of the image.
- First (b) is used to select a row $[v]$ of the image to sample. Rows with bright pixels are more likely to be sampled.
- Then, given a row, a value u is sampled from that row's 1D distribution.
- (HDR map from Paul Debevec.)

Back to Monte Carlo Path Tracing

- Summary
 1. Choose sample paths according to an appropriate Probability Density Function.
 2. Evaluate function at sample.
 3. Average these appropriately weighted values.
- Error $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right) \Rightarrow$ slow convergence (twice as good \Rightarrow need 4 times more samples).
- Variance in estimator \Rightarrow noise in image.
- Good for high dimensional functions.
- Theoretically best we can do (good estimator and more samples \Rightarrow better image).

Monte Carlo Path Tracing

- Estimate radiance value as (appropriately chosen and weighted average of (lots of) random paths along rays from eye through [each] pixel.
- Compute radiance value for each visible point from Rendering Equation:
 - Self Emission
 - Direct Illumination (shadow rays)
 - Indirect Illumination (Recursive)
- Keep lengths of recursive paths manageable – Russian Roulette.

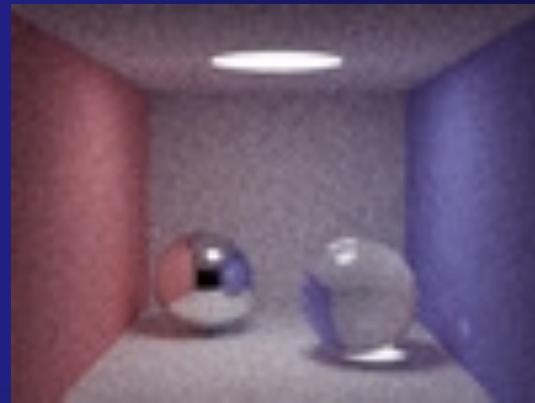


Cutler & Durand, MIT

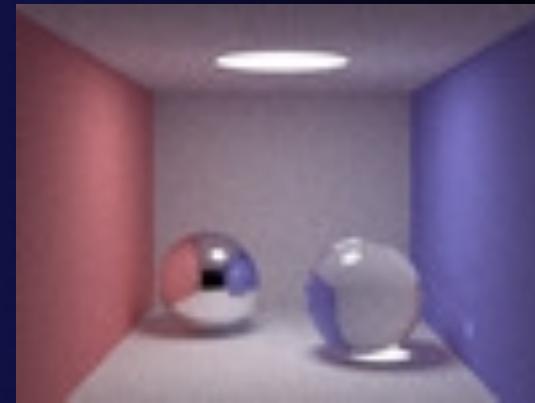
More Samples = Higher Cost but Less Noise!



8 samples/pixel
13 sec



40 samples/pixel
63 sec



200 samples/pixel
5 min



1K samples/pixel
25 min



5K samples/pixel
124 min



25K samples/pixel
10.3 hrs

2.4 GHz Intel Core 2 Quad CPU using 4 threads. See www.kevinbeason.com/smallpt

Evaluating the Radiance Arriving at the Eye

$$L_{pixel} = \int_{imageplane} L(x \rightarrow eye) h(p) dp$$

pixel-wise sampling filter

$$L(x \rightarrow eye) = L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta)$$

emitted

reflected light

$$E[L_r(x \rightarrow \Theta)] = \frac{1}{N} \sum_{i=1}^N \frac{L(x \leftarrow \Psi_i) f_r(x, \Theta \leftrightarrow \Psi_i) \cos(\mathbf{Y}_i, \mathbf{N}_x)}{p(\Psi_i)}$$

estimator for reflected light

$$L(x \leftarrow \Psi_i) = L(r(x, \Psi_i) \rightarrow -\Psi_i)$$

Recursive step

How to Stop Recursion?

- Typically use “Russian Roulette”
 - Unbiased
 - Absorption probability P : $\alpha = 1 - P$
 - α is proportional to hemispherical reflectance
 - Check:
 - black non-reflective surface, $\alpha = 0, P = 1$, so recursion must stop here;
 - mirror (specular) surface, $\alpha = 1, P = 0$, so recursion cannot stop here.
 - Threshold (fixed or adaptive)

Direct Illumination: First Option

1. Sample each light source separately

$$E[L_{direct}(x \rightarrow \Theta)] = \frac{1}{N_s} \sum_{i=1}^{N_s} \frac{L_e(y_i \rightarrow \vec{y_i x}) f_r(x, \Theta \leftrightarrow \vec{y_i x}) G(x, y_i) V(x, y_i)}{p(y_i)}$$

- Possible PDFs:
 - Uniform light area sampling
 - Uniform sampling of solid angle subtended by light source

Direct Illumination: Second Option

2. Combine light sources as one integral domain

$$E[L_{direct}(x \rightarrow \Theta)] = \frac{1}{N} \sum_{i=1}^N \frac{L_e(y_i \rightarrow \vec{y_i x}) f_r(x, \Theta \leftrightarrow \vec{y_i x}) G(x, y_i) V(x, y_i)}{p_L(k_i) p(y_i | k_i)}$$

- PDFs:
 - Uniform source selection; uniform sampling of light area.
 - Power proportional source selection; uniform sampling of light area.
- Works better for “natural” light scenes (non-discrete sources), e.g., an HDR light field environment map.

Indirect Illumination

- Uniform hemisphere sampling
- Importance sampling (PDF choice) of any of the following:
 - Cosine
 - BRDF
 - Incident radiance
 - A combination of any of the above

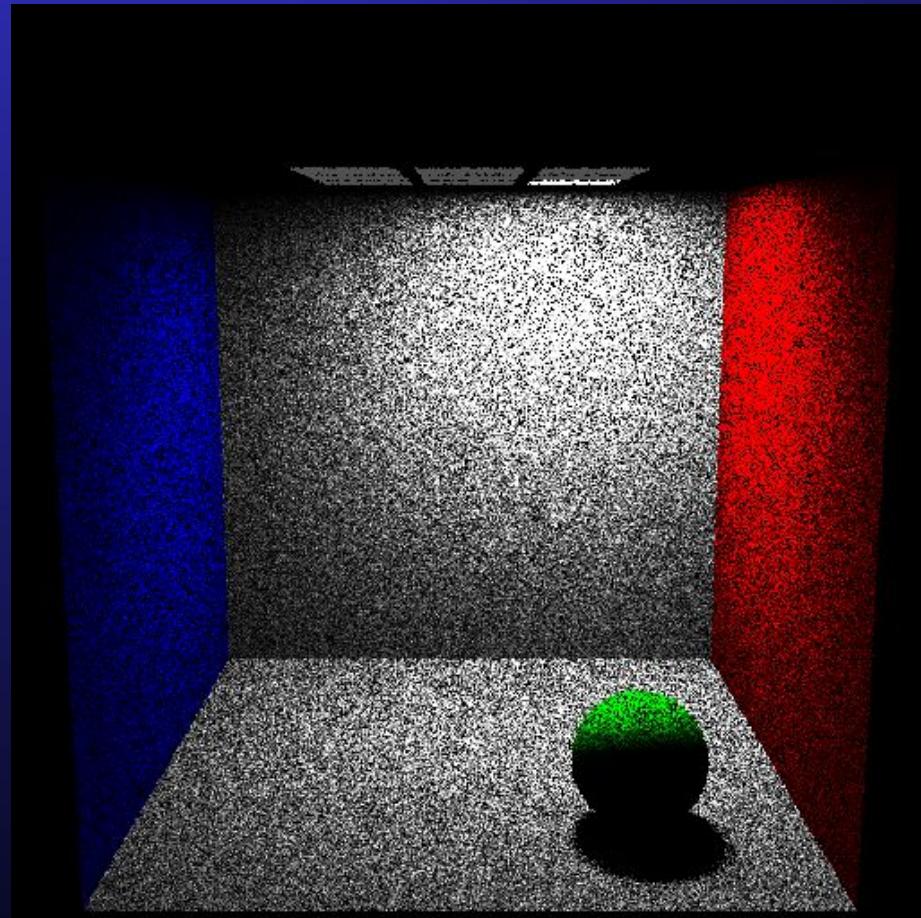
Importance Sampling

- Variance dependent upon the choice of PDF of the estimator.
- Importance sampling the integral means taking samples that are likely to contribute most significantly to the result.

$$L_{indirect}(x \rightarrow \Theta) = \int_{\Omega_x} L_r(r(x, \Psi) \rightarrow -\Psi) f_r(x, \Theta \leftrightarrow \Psi) \cos(\Psi, N_x) d\omega_\Psi$$

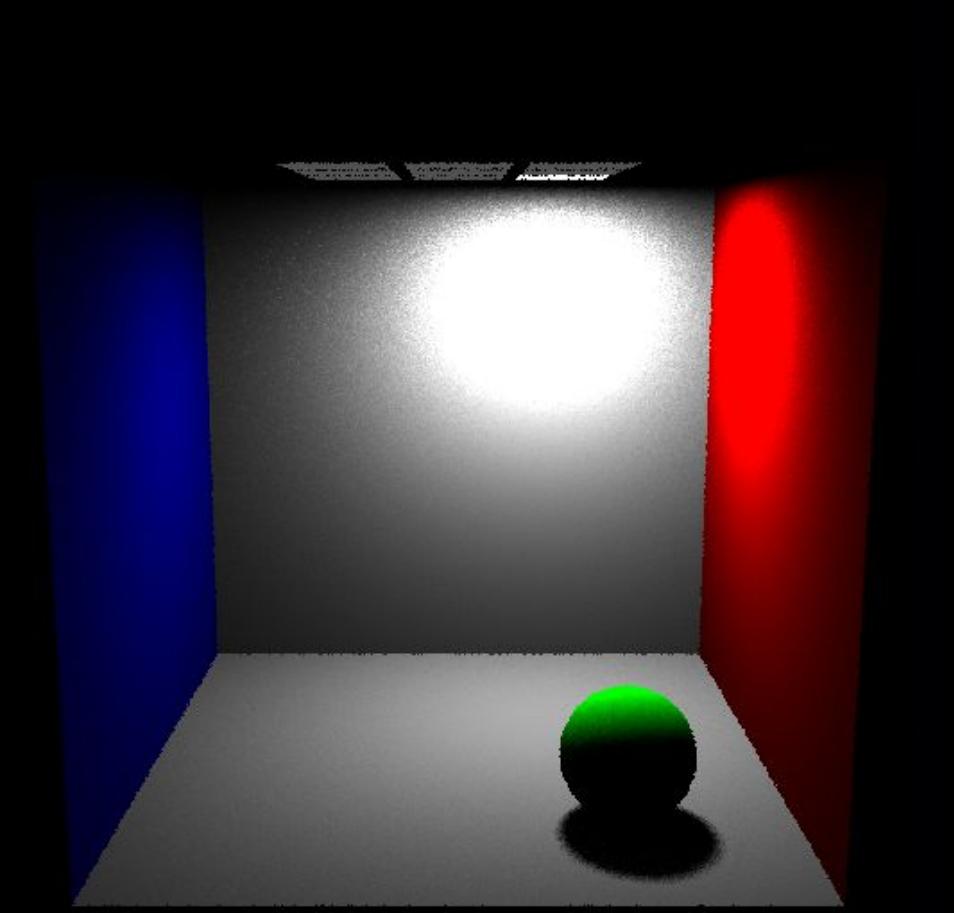
- Reminds one of progressive radiosity method.

Multiple Light Source Sampling



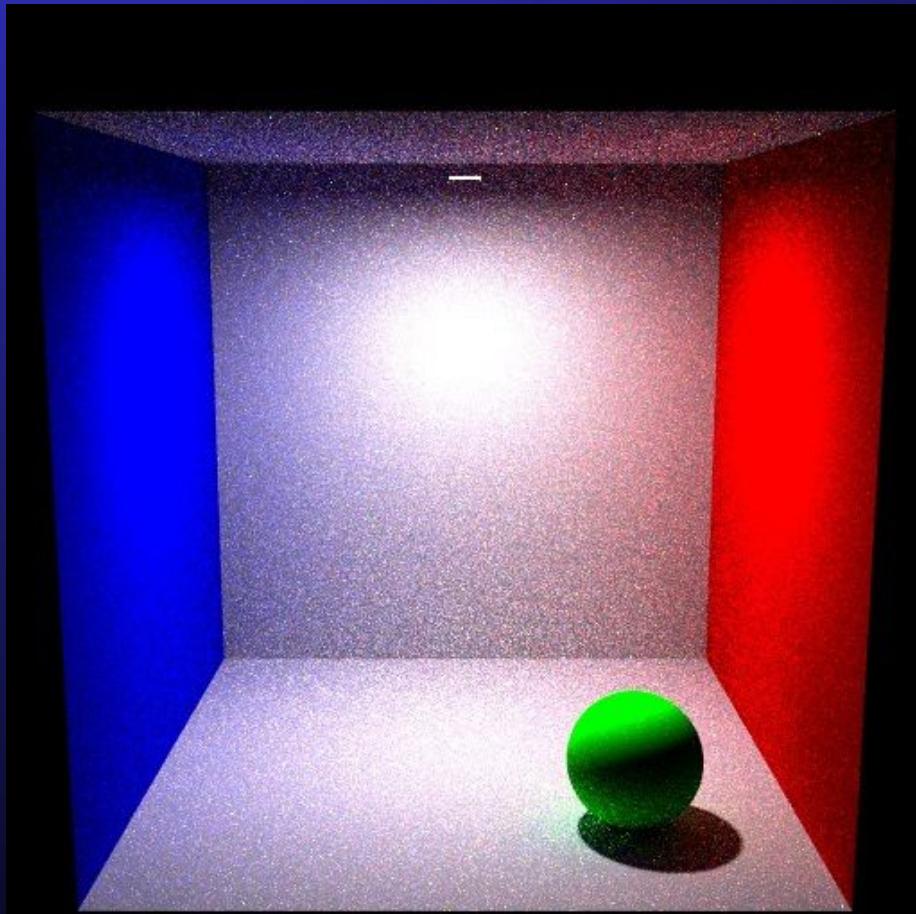
Uniform Light Source Sampling

1 path per pixel – 1 shadow ray per point – Direct Illumination Only



Power based Light Source Sampling

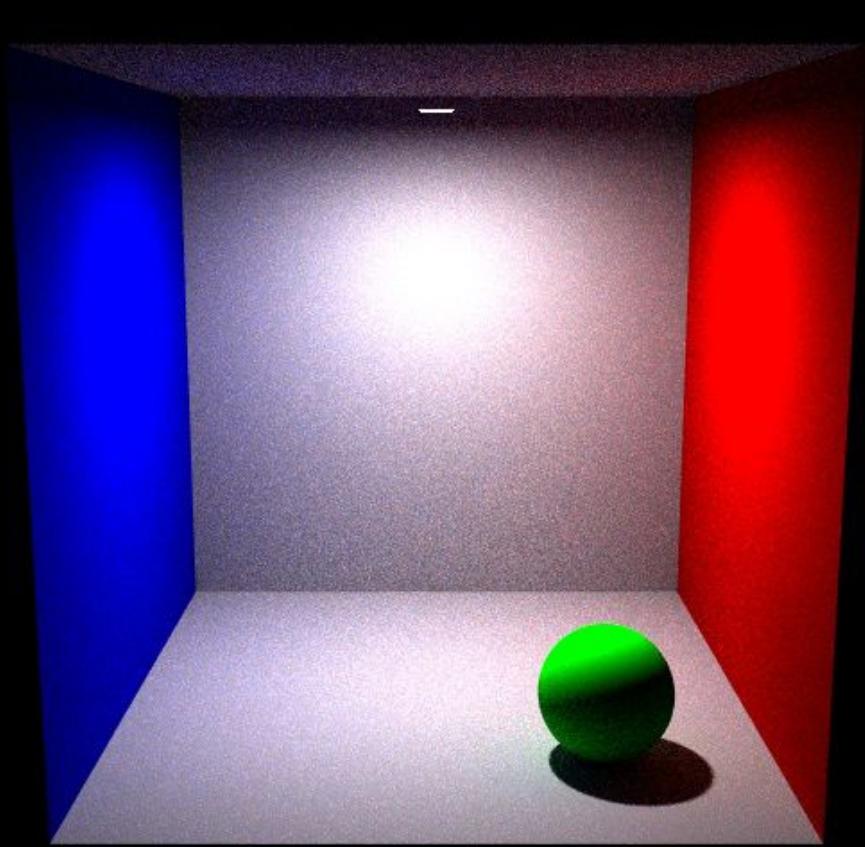
Importance Sampling: Cosine



Uniform Sampling of Hemisphere

49 paths per pixel – 1 shadow ray per point – Max eye path depth = 10

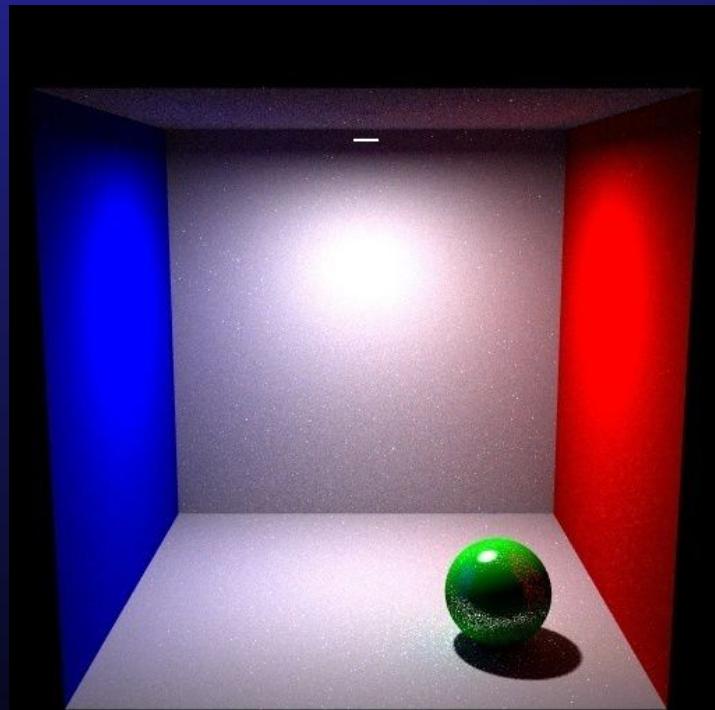
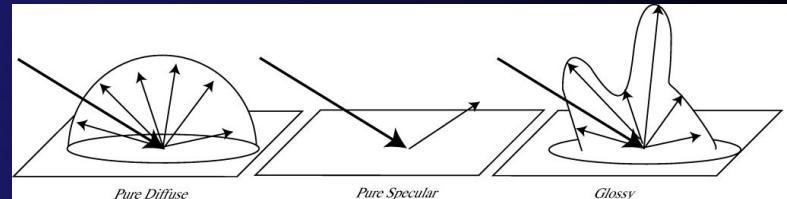
Direct + Indirect Illumination



Importance Sampling (cosine) of Hemisphere

Importance Sampling: BRDF

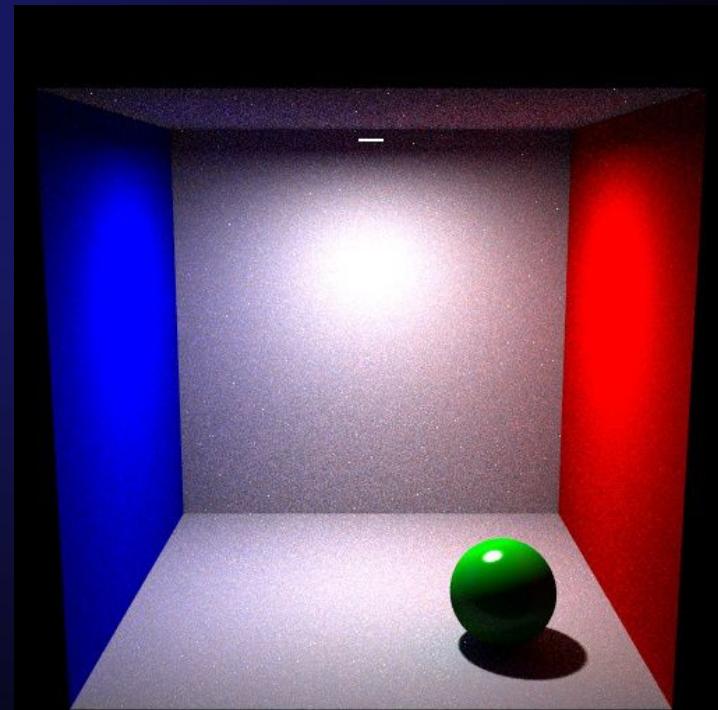
Uniformly sampling a directional
BRDF produces noise



Uniform Sampling of BRDF

49 paths per pixel – 1 shadow ray per point – Max eye path depth = 10

Direct + Indirect Illumination



Importance Sampling of BRDF

Temporal Anti-Aliasing -- Motion Blur

- Blur image of moving object to combat high frequency jerkiness or strobing caused by frame-rate sampling.
- Earliest work in 1984:
 - Korein and Badler did spheres and polygons.
 - Lucasfilm introduced spatial and temporal “noise” into ray-tracing: Monte Carlo or distribution path tracing.
- Also applied to camera depth-of-field



Monte Carlo Path Tracing and Temporal Anti-Aliasing

- Extend Monte Carlo path trace into time domain.
- Because a small amount of noise is added, it is best to supersample and filter.
- Can also do camera depth-of-field

Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

1. Jitter spatial location on image plane within the pixel.

Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

2. Jitter time T within inter-frame interval.

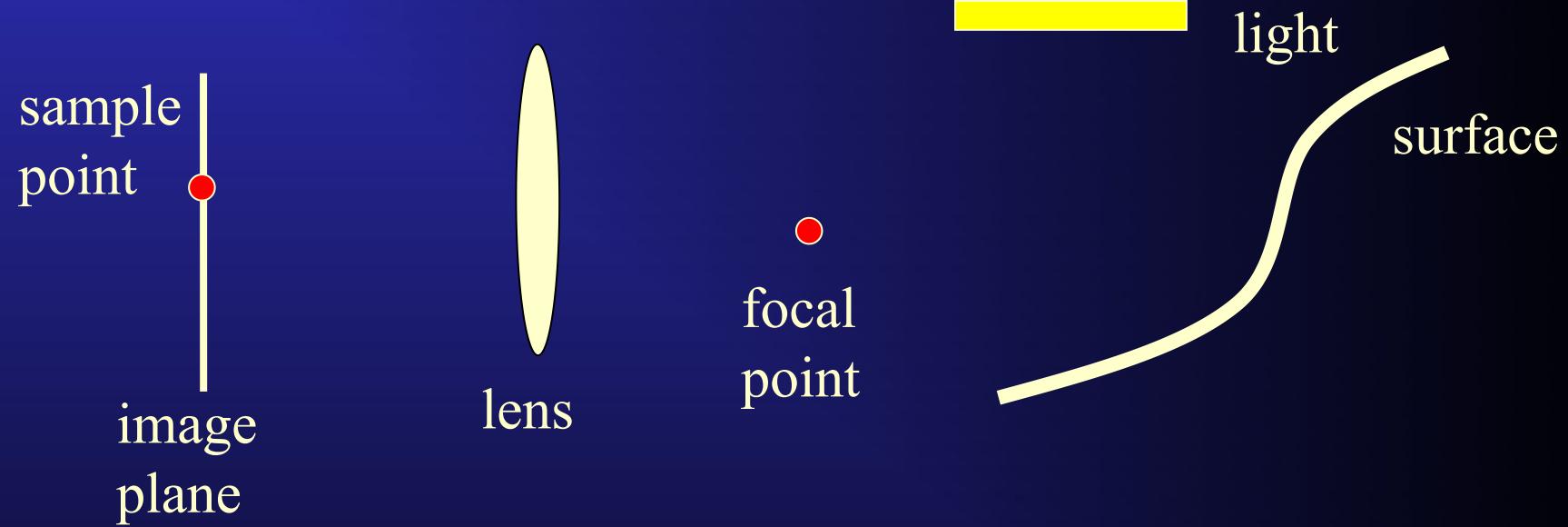
Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

3. Move camera and objects to position at that time T .

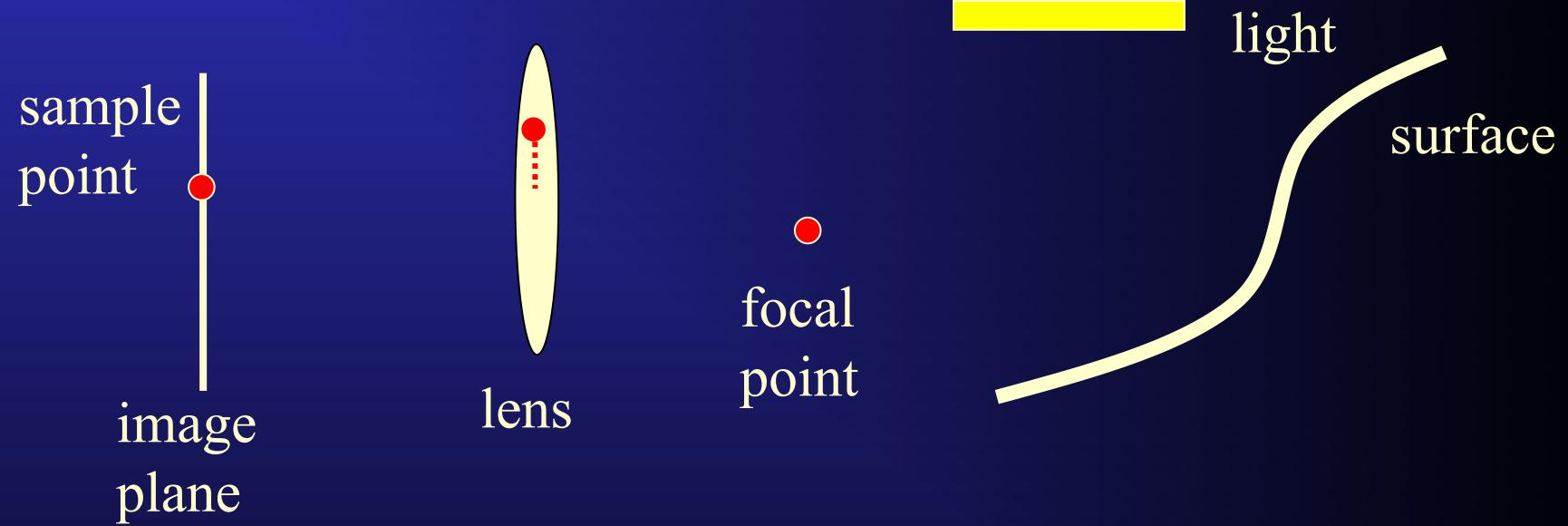
Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

4. Determine focal point.

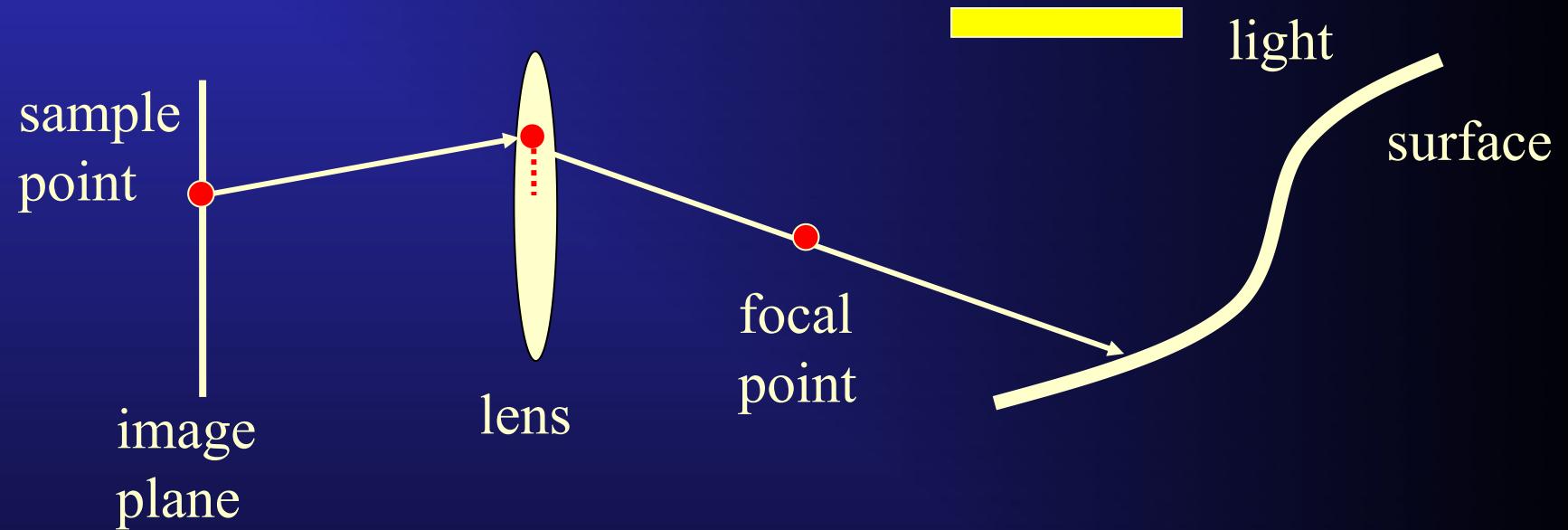
Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

5. Jitter lens location.

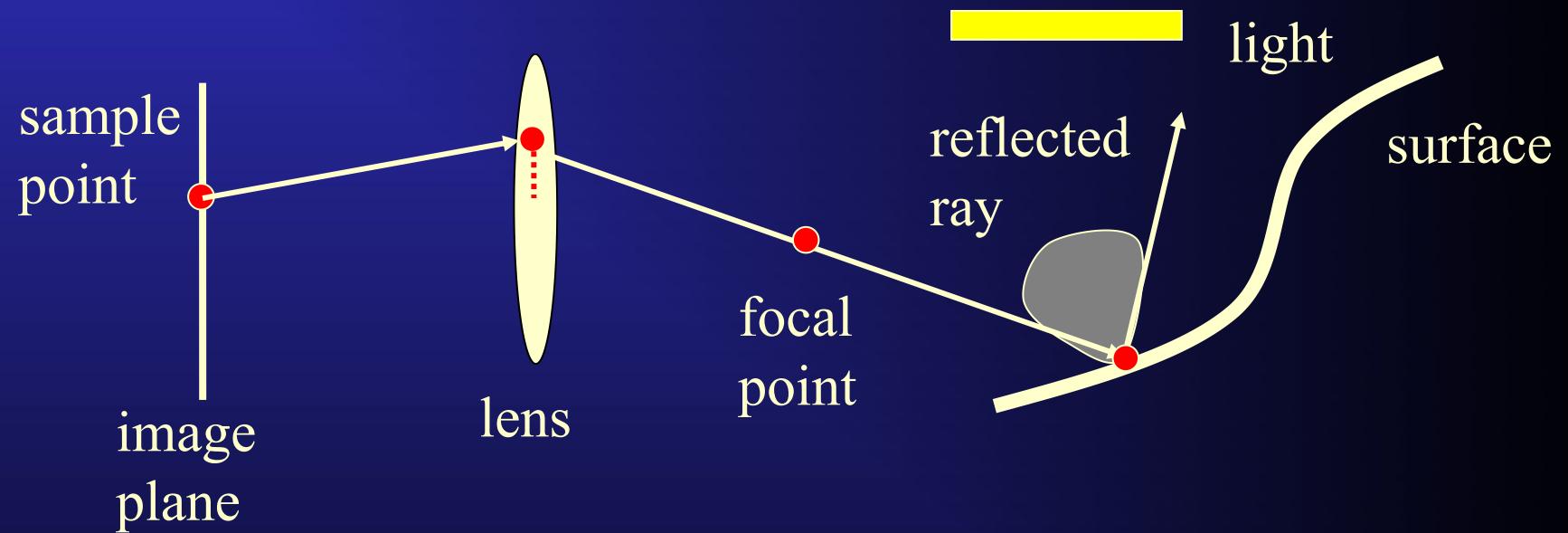
Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

6. Determine surface intersection via standard ray trace with rays going through sampled lens point and focal point.

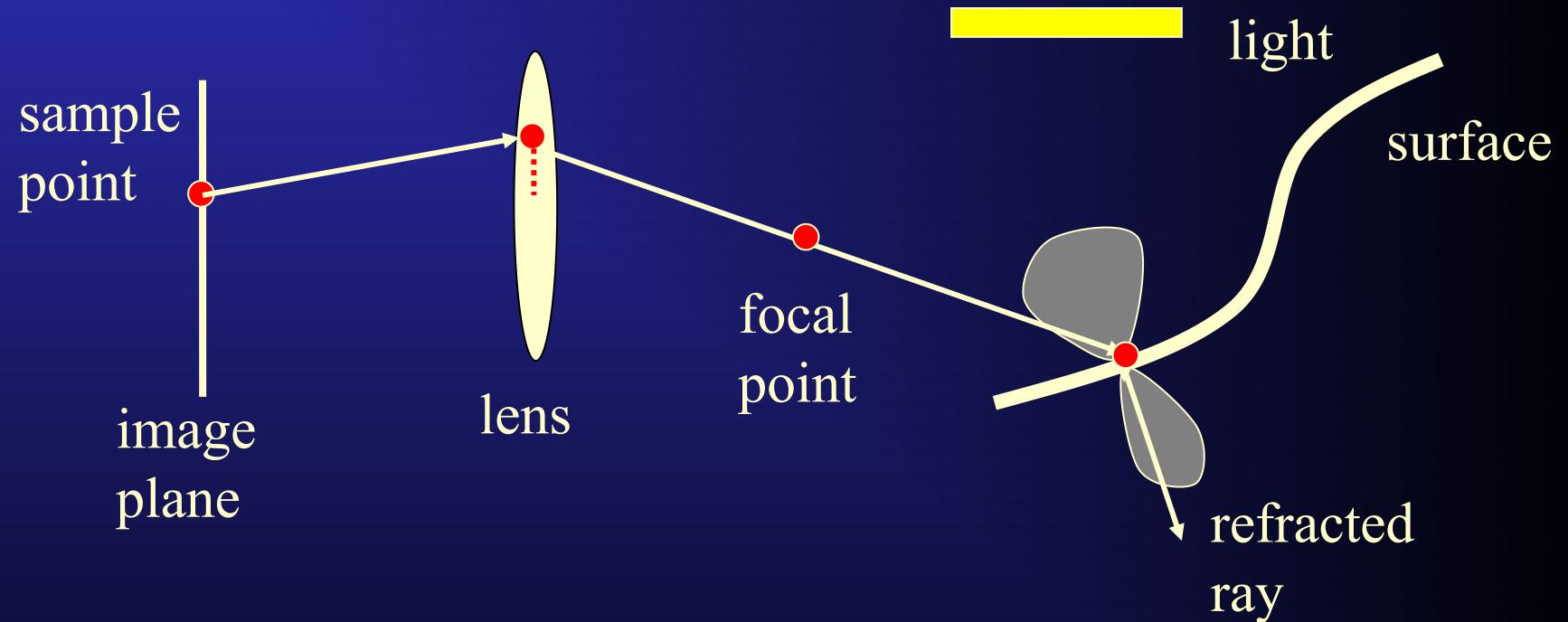
Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

7. Trace reflection ray, importance sampling according to BRDF.

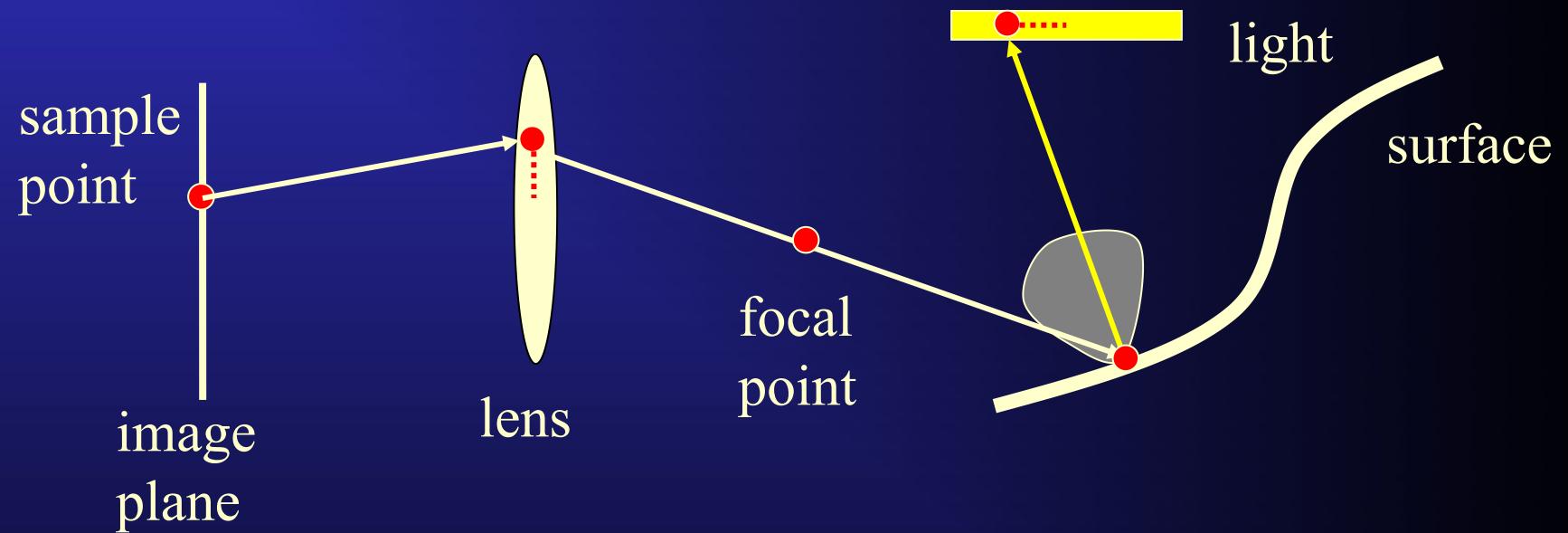
Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

8. Trace a refraction ray, if appropriate, sampling according to specular transmission function (STDF).

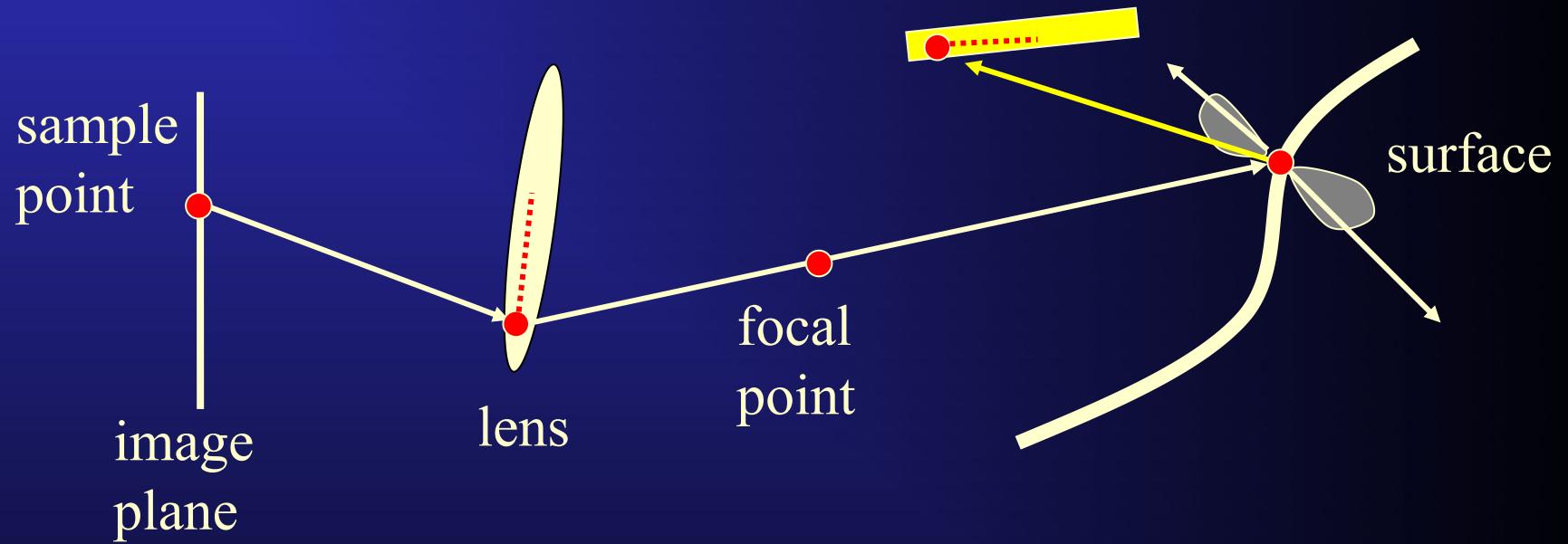
Monte Carlo Path Tracing in Time/Camera Space



For each primary ray:

9. Trace shadow ray, sampling location on light based on light (power) distribution function.

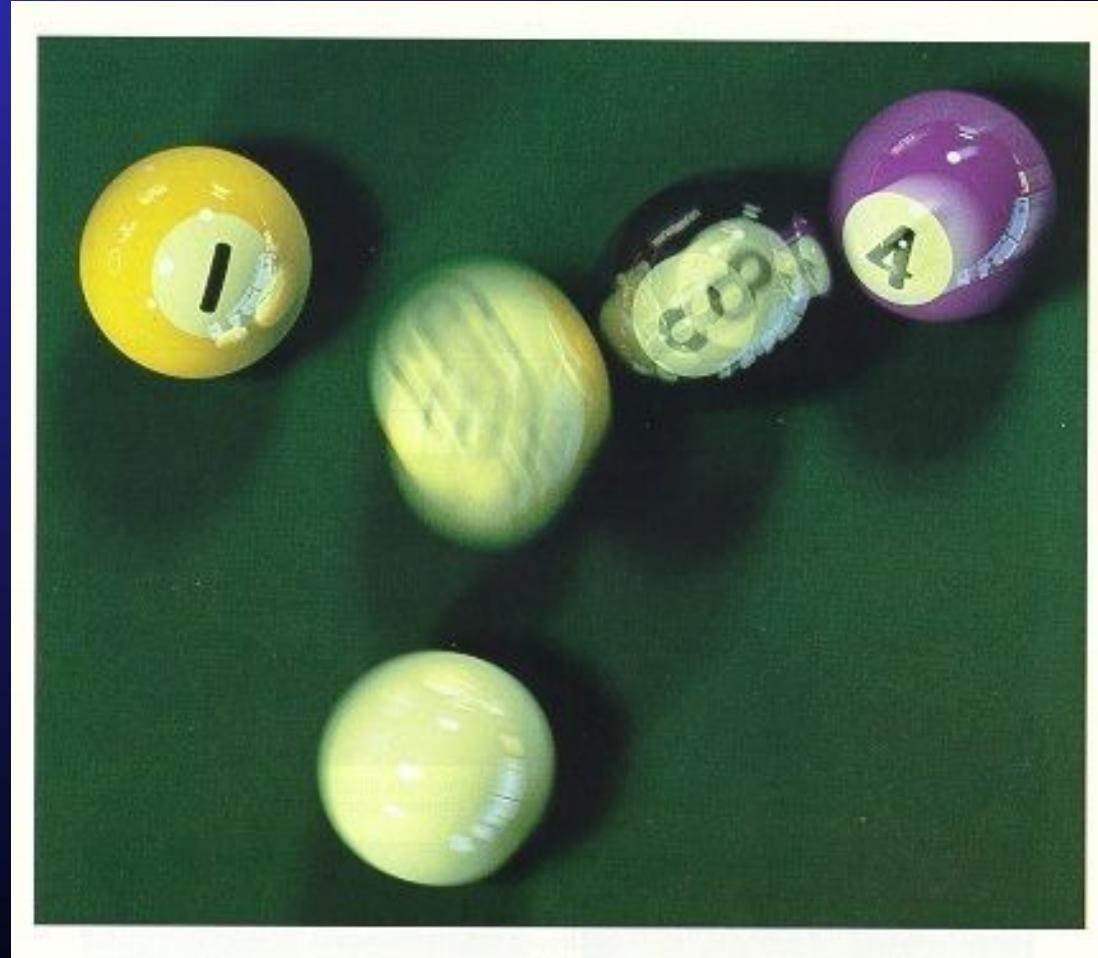
Monte Carlo Path Tracing in Time/Camera Space



Look at another possible sample for the same pixel...

Notice how the MC distribution samples some other contributing but possibly VERY DIFFERENT part of the scene.

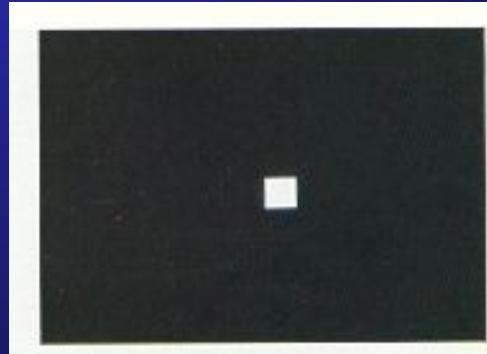
Monte Carlo Path Tracing With Temporal Anti-Aliasing



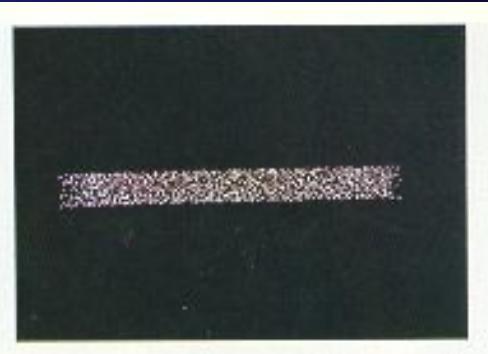
Detail from the Example



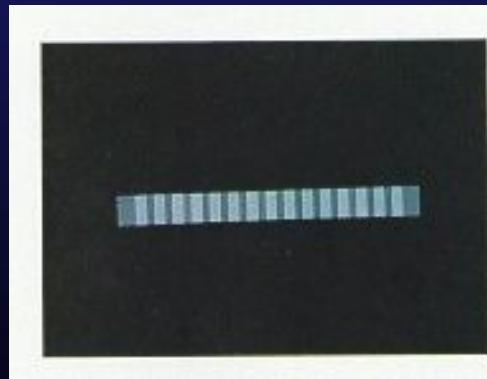
Motion Blur Test Cases



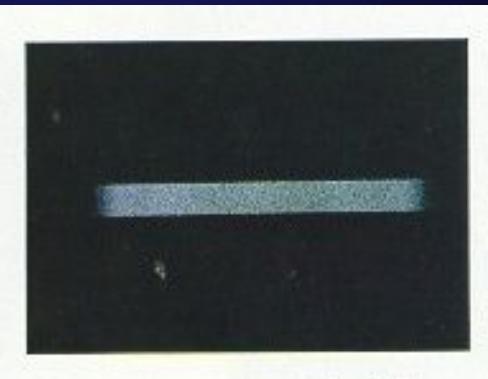
regular grid, 1 sample/pixel



jittered grid, 1 sample/pixel



regular grid, 16 samples/pixel



jittered grid, 16 samples/pixel

Monte Carlo Path Tracing: Pros and Cons

Pro:

- Arbitrary geometry – point primitive.
- Very low memory consumption.
- Theoretically “best” possible with infinite (time) resources.
- Paths in both time and space.

Con:

- View dependent (paths start at eye).
- Does not reuse illumination information.
- Not efficient for diffuse surfaces.
- Slow.
- Noisy.
- Expensive on caustics.



Photon Mapping

Based on work by Henrik Wann Jensen:

- For efficiency *re-use illumination* like finite-element (radiosity) methods, but do not pre-mesh (hence compute no form factors)!
- Retain other characteristics of Monte Carlo (recursive) path tracing.
- Subjective perceptual quality trade-off: *Blur* is better than *noise*!

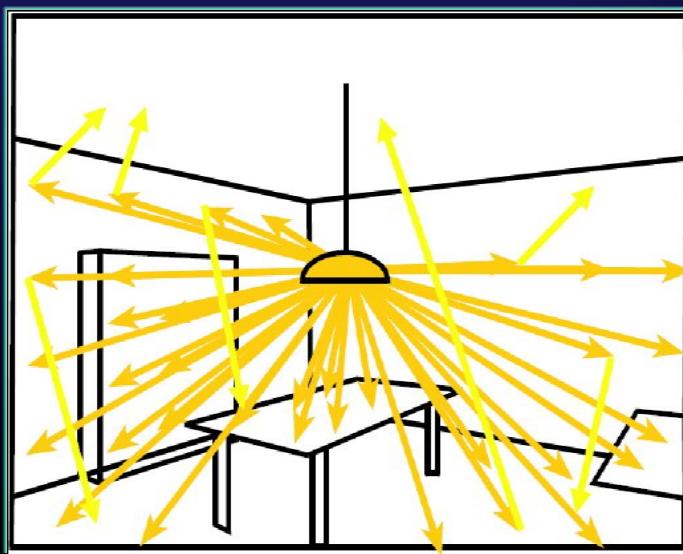


Photon Mapping

- Main idea: Decouple illumination information from geometry
 - Store (and re-use) radiance information plus incoming directions as points on surfaces;
 - do this efficiently so we can readily find neighboring samples;
 - Smooth discrete illumination samples via filtering.
- **Reverse** of ray-tracing: This is, trace light in the direction it flows (photon-like).
- Need to insure an adequate number of photons are emitted from light sources to “cover” the entire scene.
- Initially this was thought to be prohibitively expensive to compute: How many photons do you need to guarantee everything is covered? Millions? Trillions? More??

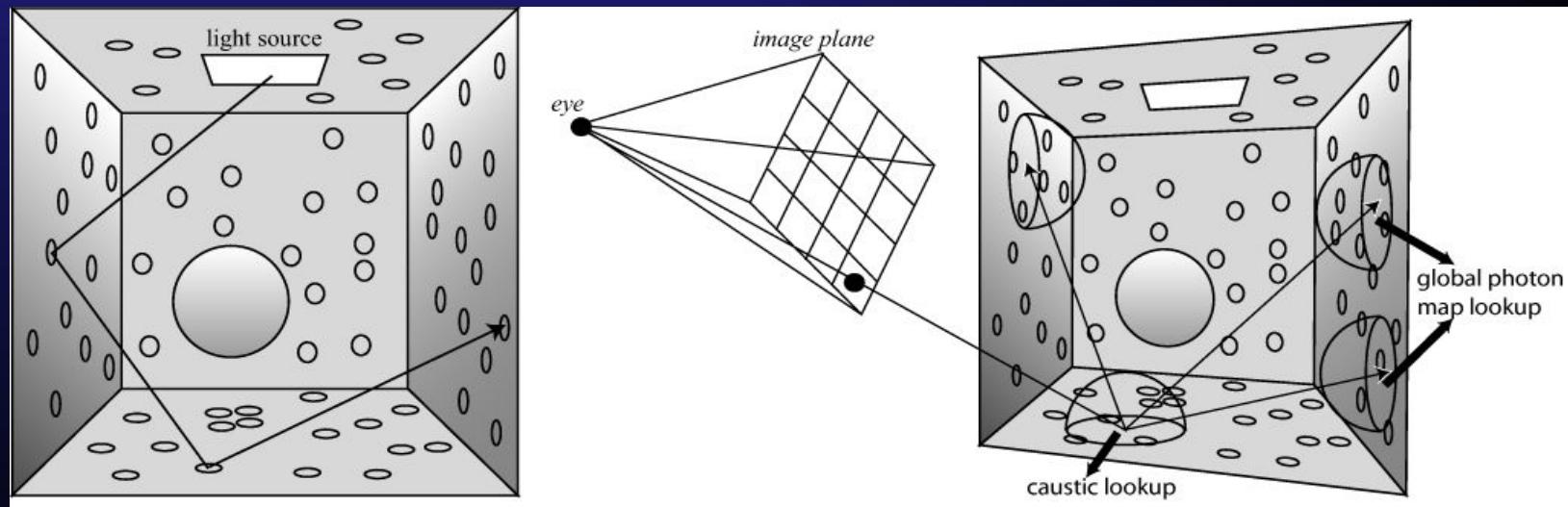
Key Properties of Photon Mapping

- Like radiosity, handles global illumination, diffuse inter-reflections (color bleeding), and participating media, but also does non-diffuse surfaces and caustics better.
- And like radiosity, the photon map is view independent! (Irradiance/colors are “baked on”.)
- But unlike radiosity, no pre-radiosity patch meshing is used.



Two Principal Steps in Algorithm

- Emit photons into the scene and **store** them in a photon map when they encounter a non-specular surface.
- Render using statistical methods to **estimate** locally (at any point) irradiance and reflected radiance from the photon map.



Pass 1: Shoot Photons

Pass 2: Find Nearest Neighbors

Three Photon Maps

- **Global** photon map: approximates global illumination for all diffuse surfaces.
- **Caustic** photon map: contains photons experiencing at least one transmission before hitting a diffuse surface.
- **Volume** photon map: used for participating media.

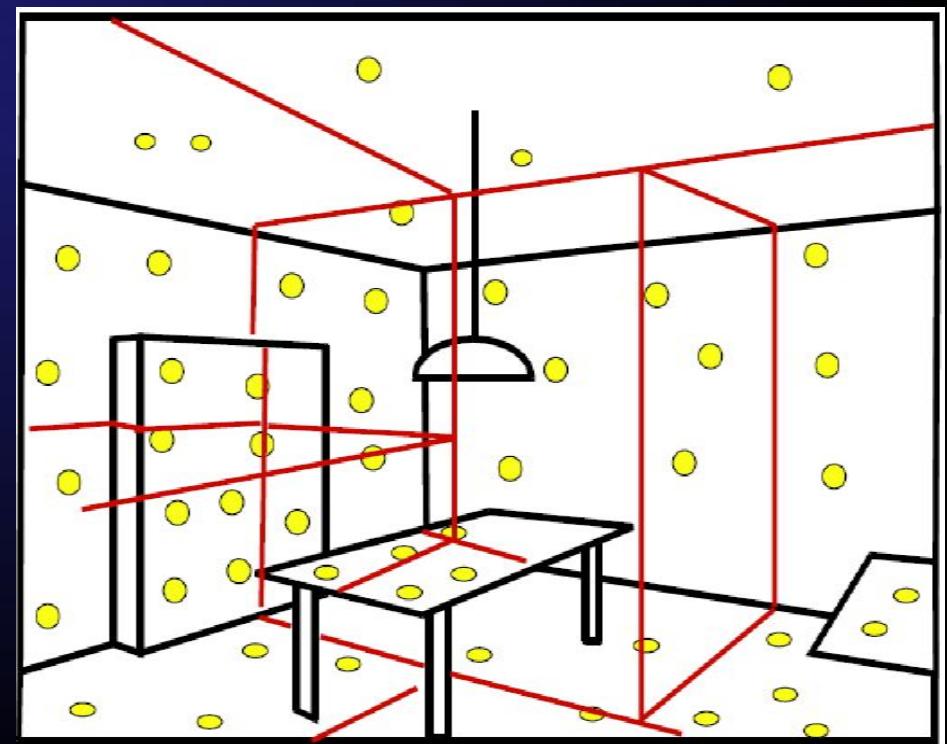
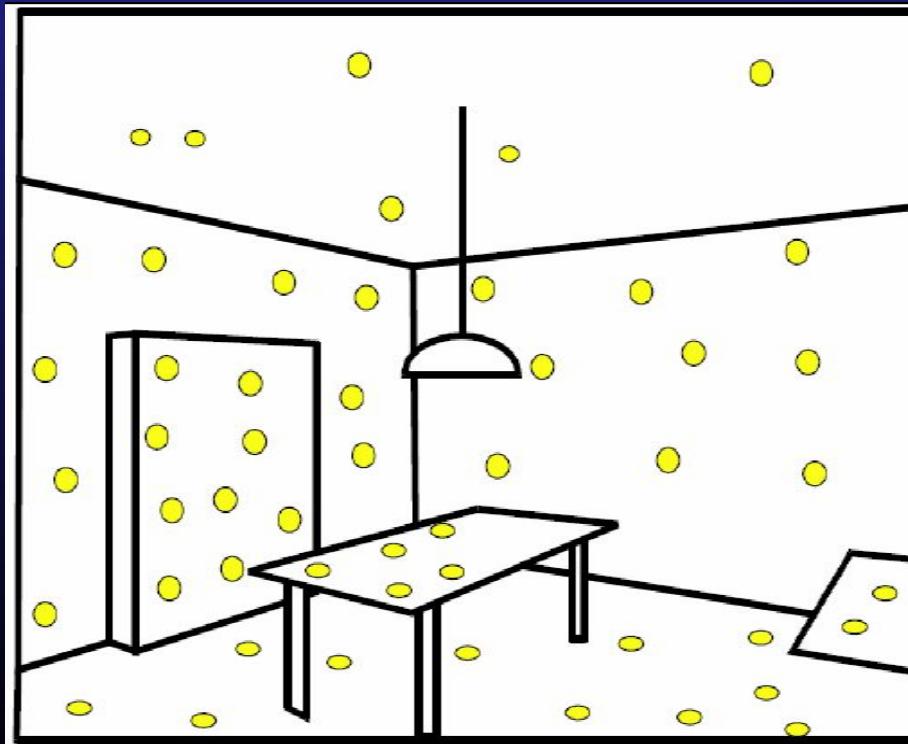
A separate caustic rendering step requires higher quality (more photons) data.

Photon maps are converted to balanced *K-d trees* in time $O(N \log N)$; this allows finding specific nearest K neighboring photons during the rendering pass with $O(K + \log N)$ search time.

Exploit K-d Tree Spatial Data Structure

- Index photon hits on geometry via K-d tree (essentially about equal numbers of hits per cell volume).
- Idea is to collect neighboring cached photons quickly!
- Photon caching

Spatial data structure (K-d tree)

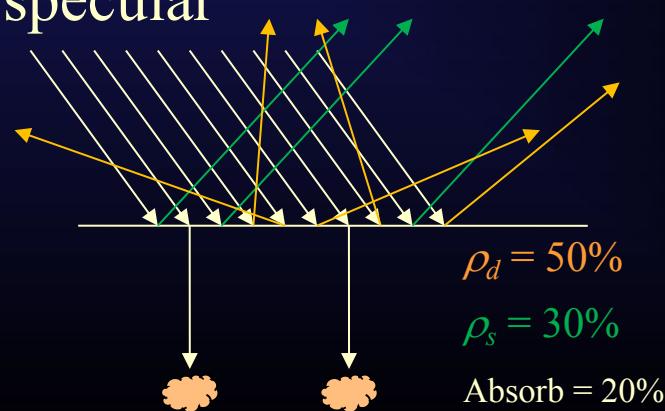


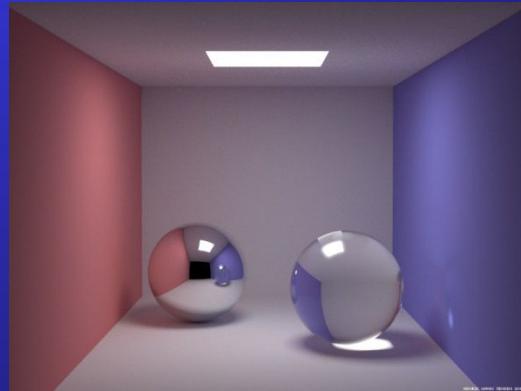
Photon Tracing

- Trace emitted photons through scene (“backward ray tracing”) except photons *propagate flux* (power/radiosity).
- (Rather than ray tracing which *gathers* reflected light as the recursion unwinds.)
- Photons interact with materials differently than rays (e.g., refraction does not effect photon flux).
- Photon interactions with a surface or object are computed probabilistically from material properties.
 - Reflected (BRDF)
 - Transmitted (BTDF)
 - Absorbed (ends trace for this photon – decided via “Russian roulette”, exactly as in Monte Carlo path tracing)

Handling Mixed Diffuse and Specular Surfaces

- Most surfaces have both specular and diffuse components in their BRDF:
 - ρ_d is diffuse reflectance
 - ρ_s is specular reflectance
 - $\rho_d + \rho_s < 1$ (conservation of energy)
- When photon hits this surface, compute a uniform random value $0 \leq \zeta \leq 1$:
 - If $\zeta < \rho_d$ then reflect diffuse
 - Else if $\zeta < \rho_d + \rho_s$ then reflect specular
 - Otherwise absorb photon

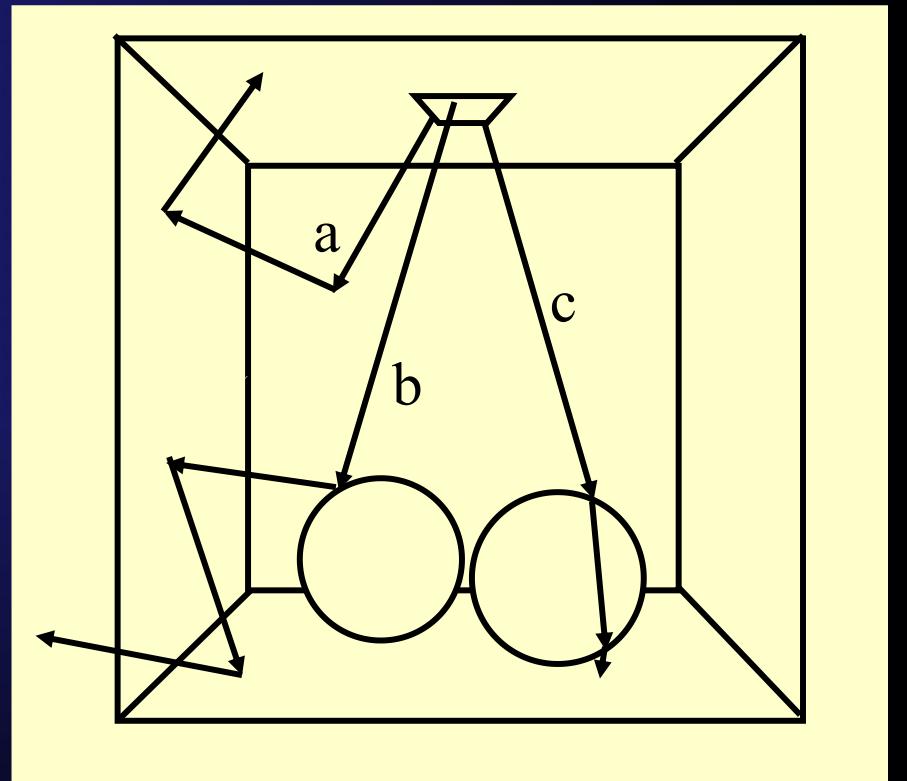




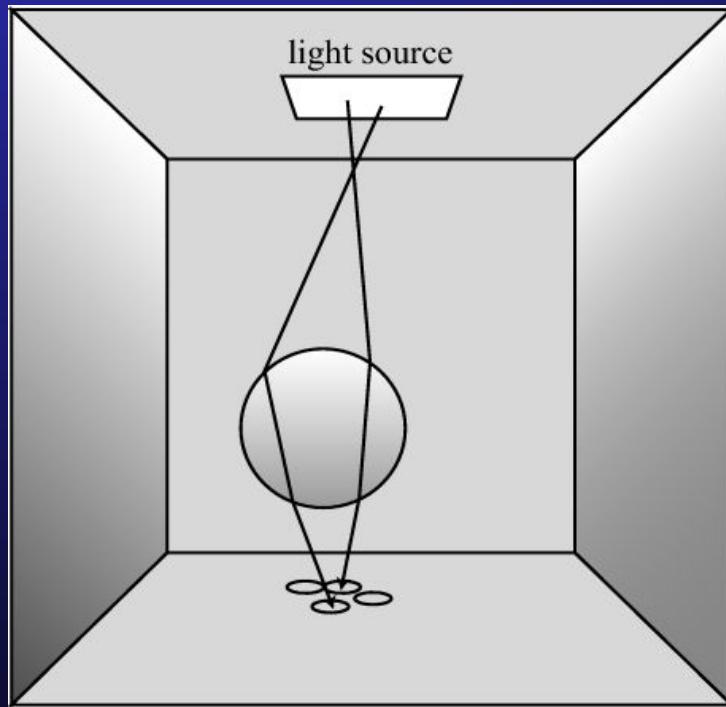
Sample Photon Paths

3 photon paths in a scene inside the Cornell box. Chrome sphere on left and glass sphere on right.

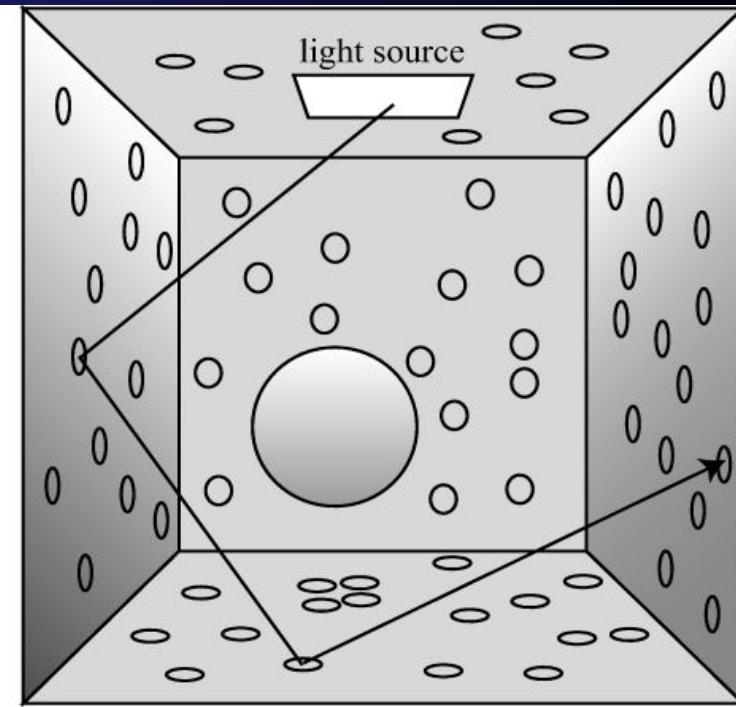
- (a) two diffuse reflections then absorption.
- (b) one specular reflection followed by two diffuse reflections.
- (c) two transmissions then absorption (cached on caustic map).



Photon and Caustic Maps Accumulate “Hits”: Incoming Direction and Irradiance

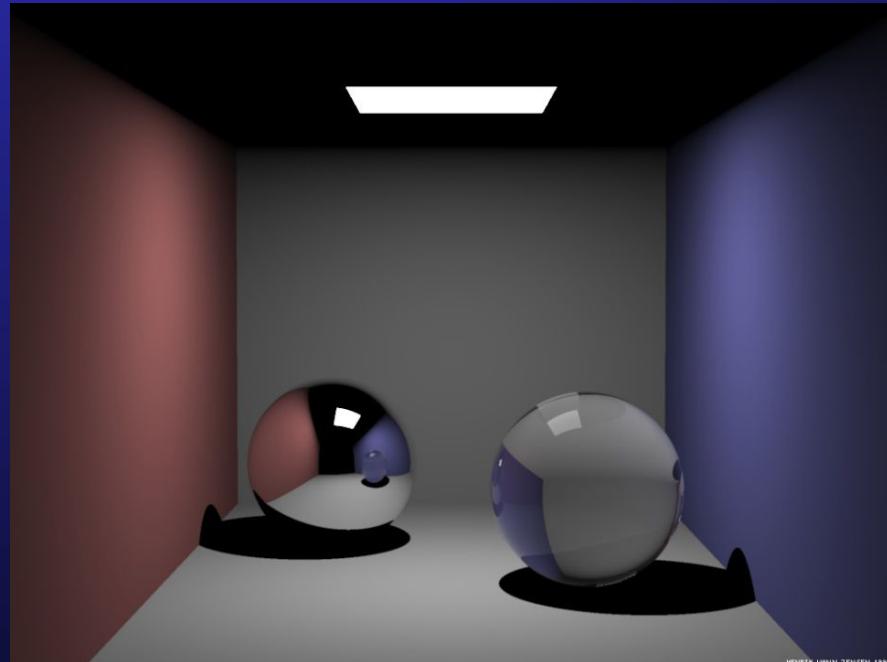


Caustic Map



Global Photon Map

“Cornell Box” with Chrome and Glass Spheres



Ray-traced image
Every pixel accounted for



Photons in photon maps
Can't guarantee complete coverage

Photon Storing

- Photons stored whenever they hit diffuse (non-specular) surfaces.
- Photon can be stored several times along its path.
- Photon information also stored where absorbed.
- Stored:
 - Position (x,y,z) and incident direction (x,y,z) of hit
 - Incoming photon power (magnitude in watts)
 - Color (r,g,b)
 - Housekeeping flag for photon map K-d tree data structure
- Overall, ~20 bytes per photon.
- Storage cost: #photons \times #bounces \times 20 bytes (not too bad)
- We'll see later what #photons might be.

Photon Emission

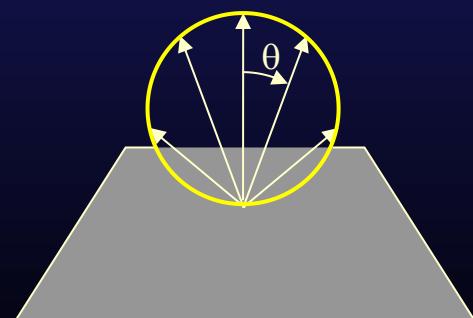
- Distribution of photons corresponds to distribution of emissive power P (“wattage”):

$$P_{photon} = \frac{P_{lightsource}}{n_{emitted\text{-}photons}}$$

- For multiple sources it is desirable that all emitted photons have about the same power, thus $n_{emitted\text{-}photons}$ will vary from emitter to emitter.
- E.g., for a 60W bulb sending out 100K photons, each has 0.6mW flux.

Photon Distribution Models Emitter Type

- Diffuse light source: photons emitted in uniformly distributed random directions.
- Directional light (outside the scene): photons emitted in that direction.
- Area emitter*: photons emitted in cosine distribution (maximum at normal direction).
- General distributions in shape and direction.



* E.g., rectangular area emitter with power Φ and n photons:
Cosine power distribution: each photon gets $(1/n) \Phi \cos \theta$

Projection Maps in Photon Emission

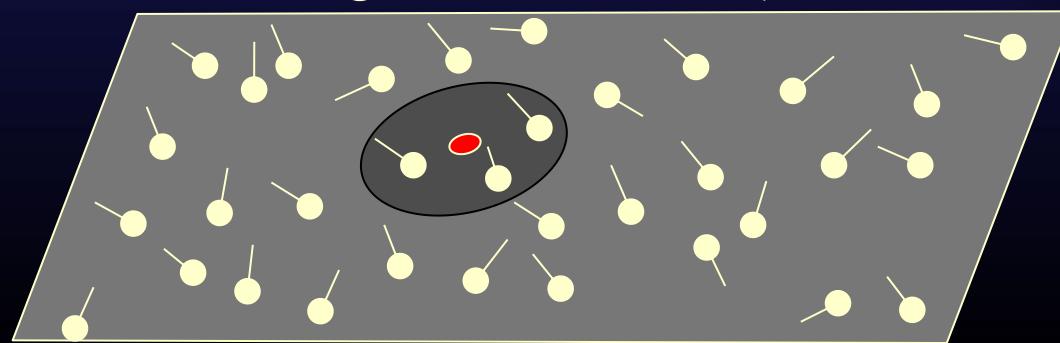
- **Projection maps:** a discrete cellular (2D array) map of the geometry as seen from the light source (compare with the hemicube of radiosity) – only photons that hit cells with geometry in them are emitted.

$$P_{photon} = \frac{P_{lightsource}}{n_{emitted-photons}} \quad \frac{\text{cells with objects}}{\text{total number of cells}}$$

- Planar projection map for directional light; spherical map for point light.
- Identify specular objects in cells (for generation of caustics).

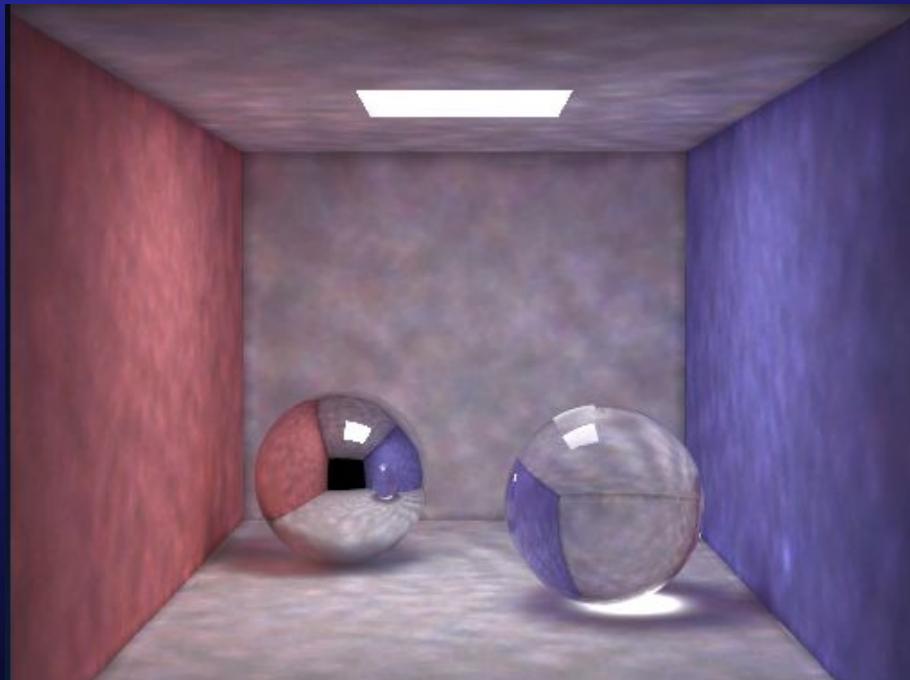
Radiance Estimate

- Photon map is an estimator of incoming flux (irradiance) to a surface.
- To compute irradiance we need to integrate (average) this information over a small area.
- Do integration by locating a set of m photons that are closest to the point ● for which we wish to compute irradiance. (Here, e.g., $m = 3$; generally $50 \leq m \leq 500$.)
- Use cone-shaped filter to bias estimate toward photons nearest the center of the sampling disk. (Can use other filters; e.g. Gaussian.)
- Various methods to select and optimize the number of photons needed for the estimate (a function of accuracy required, e.g.).
- (Need to be careful at edges and corners!)



Number of Samples Matters

- 50 photons/area



- 500 photons/area



Cone Filter

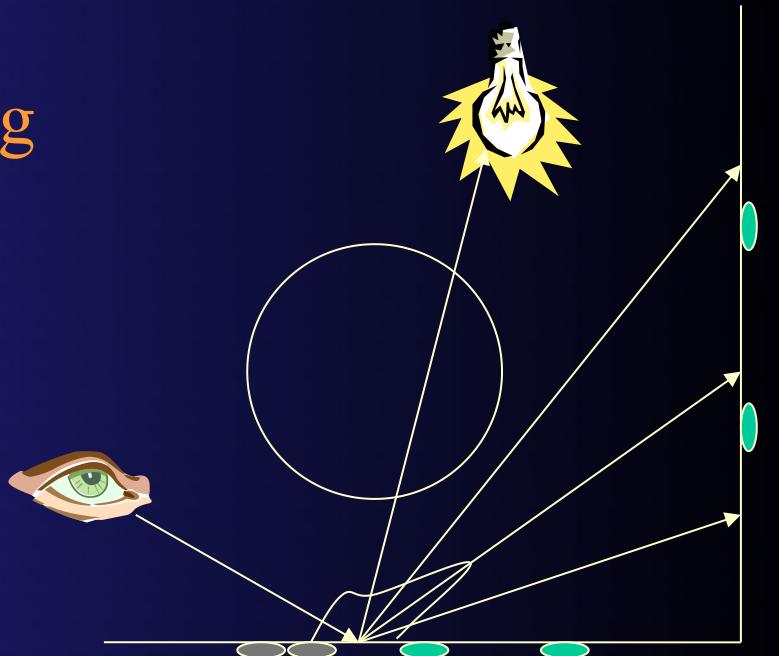
$$L_r(x \rightarrow \Theta) \approx \frac{1}{1 - \frac{2}{3k} \pi r^2} \sum_{p=1}^n \left(1 - \frac{\|x - x_p\|}{k r} \right) f_r(\omega_p, \omega_r) \text{(cached photon flux)}$$

Reflected light cone weight BRDF(incoming – reflected)

k = cone scale factor relative to cone volume ($k = 1$ fails to count samples at radius r , k is probably >1)

Rendering

- Rendered by distributed ray tracing (rays gather irradiance).
- When ray hits first diffuse surface...
 - Compute direct irradiance from light sources
 - Compute reflected radiance of caustic map photons, ignoring global map photons
 - Importance sample BRDF f_r
 - Use global photon map to importance sample incident radiance function L_i
 - Evaluate reflectance integral by casting rays and accumulating radiances from global photon map



First diffuse intersection.
Return radiance of caustic
map photons here, but
ignore global map photons

Use global
map photons
to return
radiance
when
evaluating L_i
at first
diffuse
intersection.



Photon Mapping: Computing Image (Details)

- Split BRDF and incoming radiance

$$f_r(x, \Theta \leftrightarrow \Psi) = f_{r,S}(x, \Theta \leftrightarrow \Psi) + f_{r,D}(x, \Theta \leftrightarrow \Psi)$$

Specular

Diffuse

$$L_i(x \leftarrow \Psi) = L_{i,l}(x \leftarrow \Psi) + L_{i,c}(x \leftarrow \Psi) + L_{i,d}(x \leftarrow \Psi)$$

Direct illumination Caustic map Diffuse

Computing Image

- Reflected radiance is then given by specular-surface distributed ray tracing. When ray first hits surface:

$$\begin{aligned}
 L_r(x \rightarrow \Theta) &= \int_{\Omega_x} L_i(x \leftarrow \Psi) f_r(x, \Theta \leftrightarrow \Psi) \cos(\Psi, N_x) d\omega_\Psi \\
 &= \int_{\Omega_x} L_{i,l}(x \leftarrow \Psi) f_r(x, \Theta \leftrightarrow \Psi) \cos(\Psi, N_x) d\omega_\Psi + \\
 &\quad \int_{\Omega_x} (L_{i,c}(x \leftarrow \Psi) + L_{i,d}(x \leftarrow \Psi)) f_{r,S}(x, \Theta \leftrightarrow \Psi) \cos(\Psi, N_x) d\omega_\Psi + \\
 &\quad \int_{\Omega_x} L_{i,c}(x \leftarrow \Psi) f_{r,D}(x, \Theta \leftrightarrow \Psi) \cos(\Psi, N_x) d\omega_\Psi + \\
 &\quad \int_{\Omega_x} L_{i,d}(x \leftarrow \Psi) f_{r,D}(x, \Theta \leftrightarrow \Psi) \cos(\Psi, N_x) d\omega_\Psi
 \end{aligned}$$

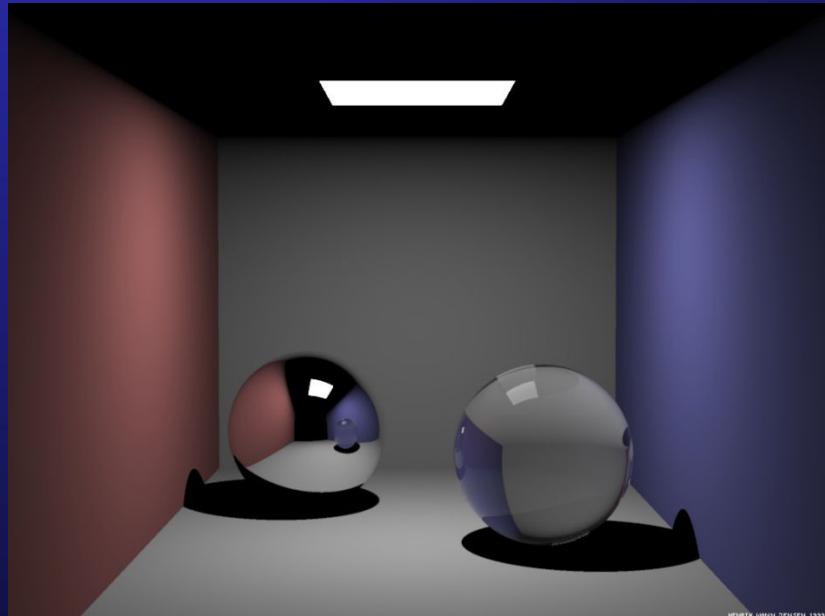
RT, direct emitters,
area sampled

 MC RT:
importance sample
based on BRDF $f_{r,S}$

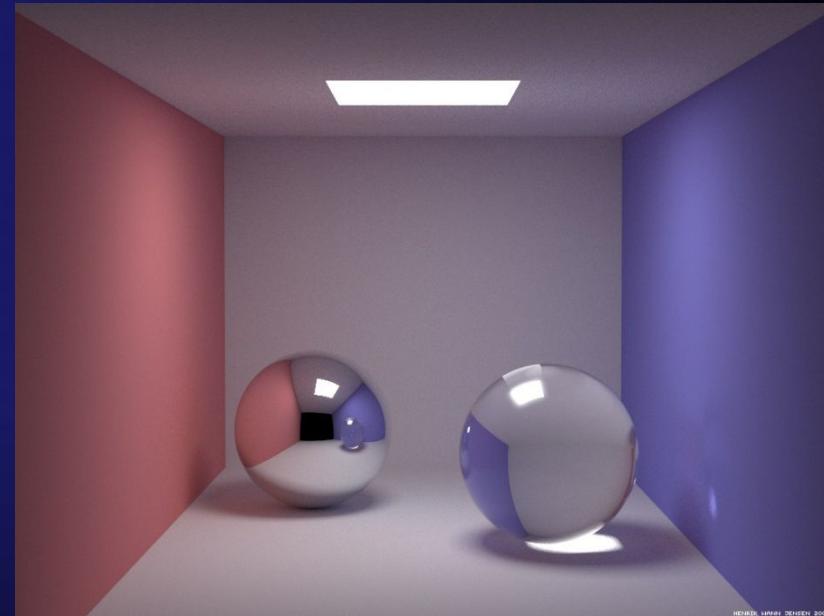
 Radiance estimate
using caustic map
photons

 MC RT:
importance sample
BRDF using
global photon map

Photon Mapping – Steps, Timing, and Comparisons



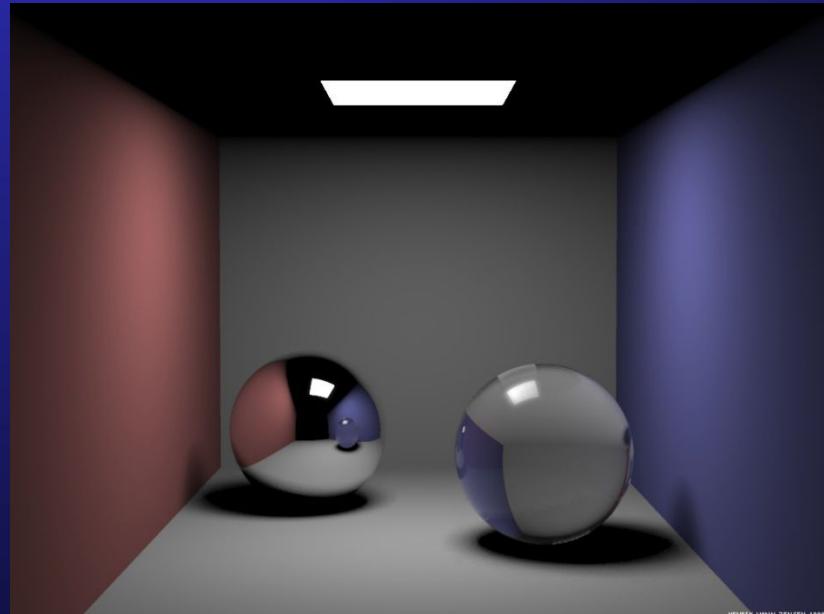
Ray tracing
1.5 seconds



MC Path Tracing
(for reference)

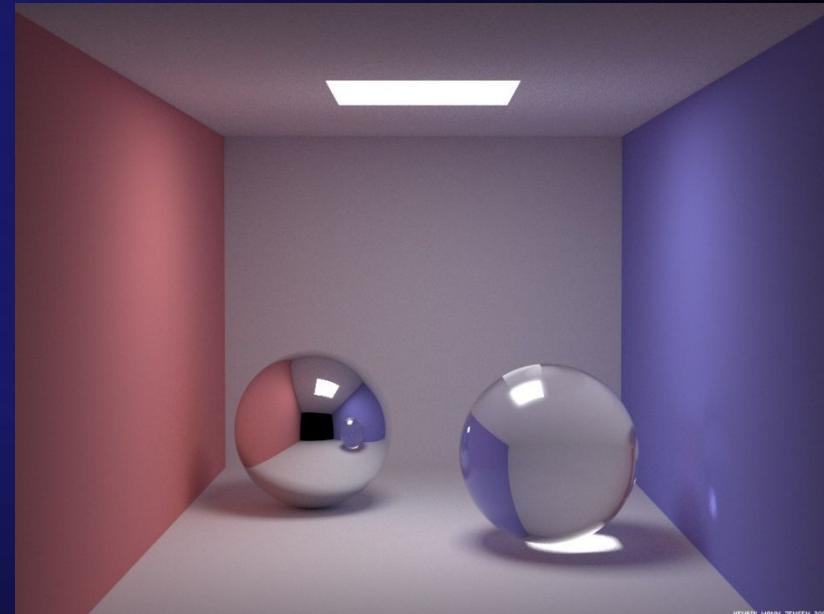
Timings based on Dual P3 800MHz Linux PC

Photon Mapping



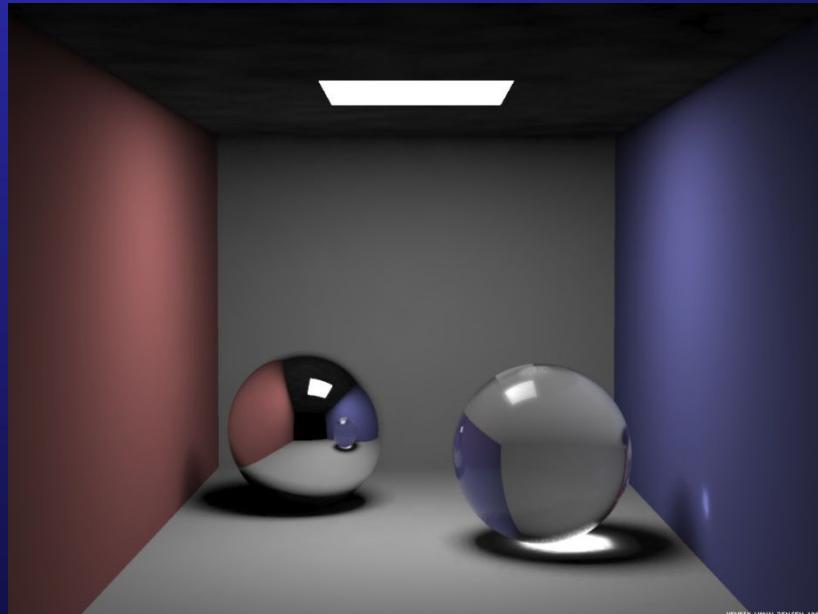
Ray tracing + Soft Shadows

$1.5+5.5 = 7$ seconds

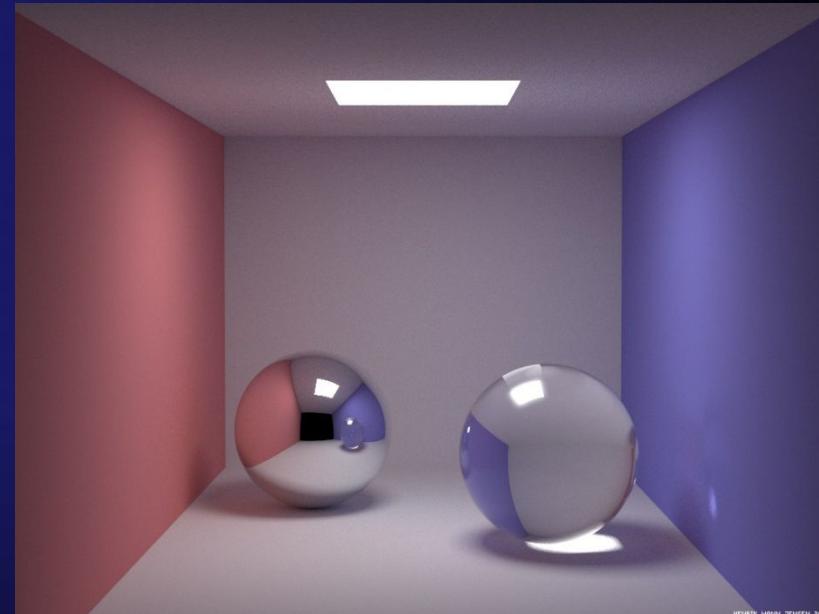


MC Path Tracing
(for reference)

Photon Mapping



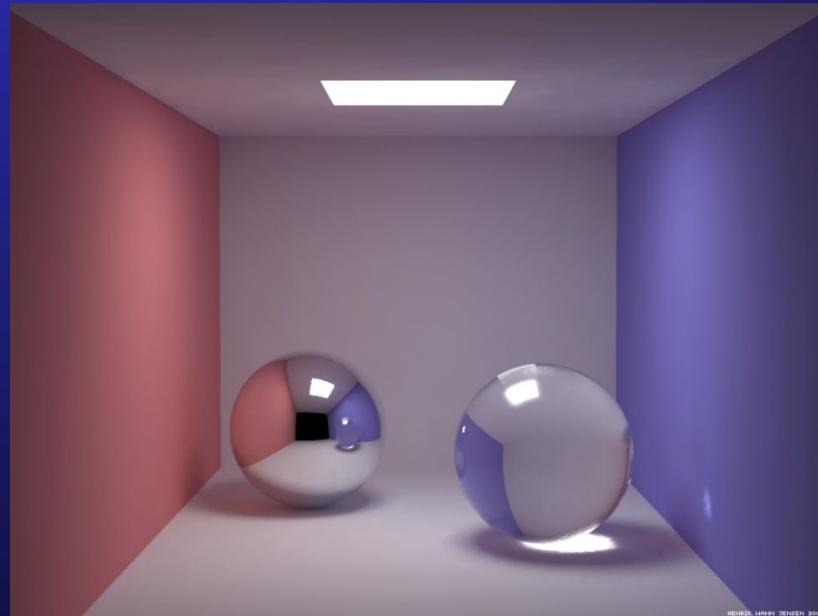
Ray tracing + Soft Shadows + Caustics
 $1.5+5.5+5 = 12$ seconds



MC Path Tracing
(for reference)

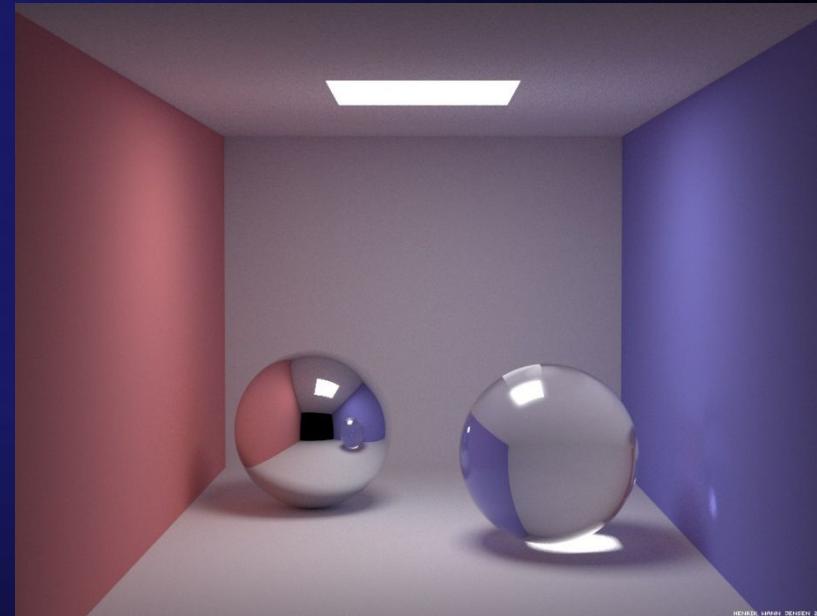
50,000 hits in caustics photon map

Photon Mapping



Ray tracing + Soft Shadows + Caustics
+ global illumination
 $1.5+5.5+5+3 = 15$ seconds

200,000 photons in photon map

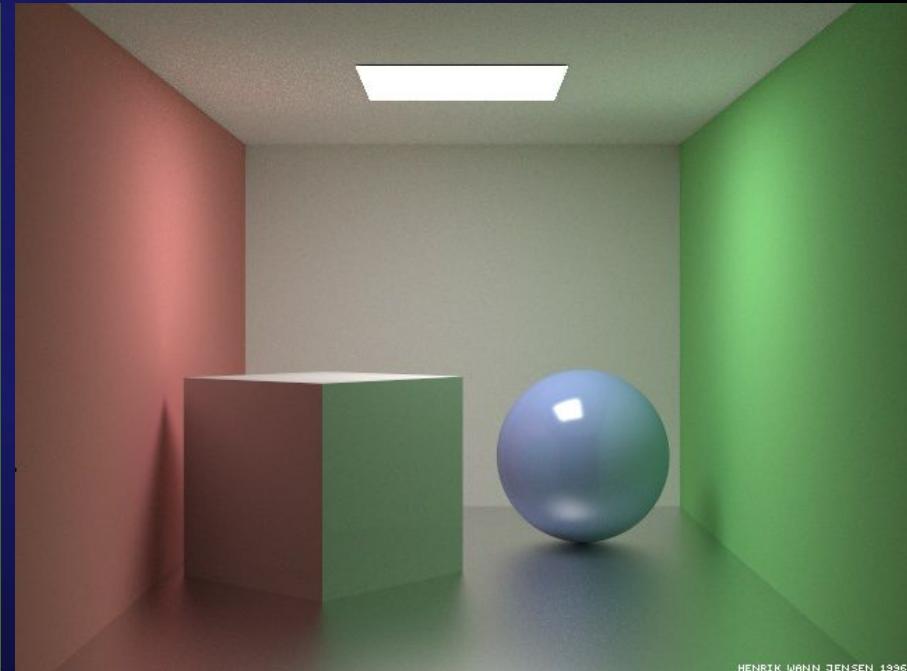


MC Path Tracing
(for reference)

Photon Mapping



Photon Mapping (16 samples/pixel)

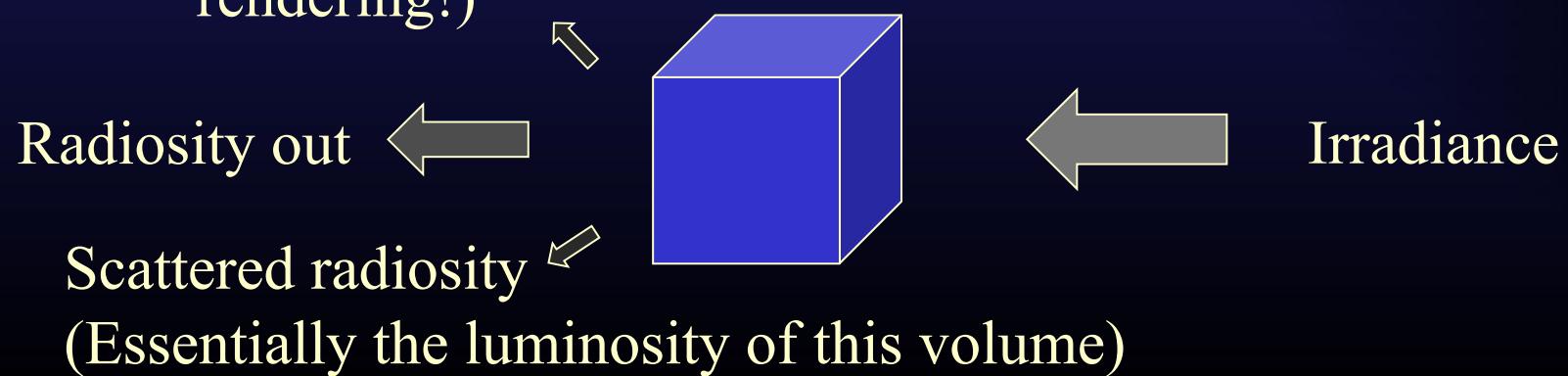


MC Path Tracing (1000 samples/pixel)

MC Path tracing image still has noise!
Photon Mapping still has some blur.

Light and a Participating Medium

- Model haze of smoke, dust, water vapor, or particles in air.
- These can emit, absorb, reflect, or scatter light, making the participating media visible.
- Divide space into voxel-like **zones**.
- Calculate absorption, scattering, and transmittance for each zone. E.g., more scattering will make a zone more visible, thus adding haze to that volume of space.
- (We already saw the transmittance formulation for volumes; this combines radiosity with volumetric rendering!)



Hazy Room

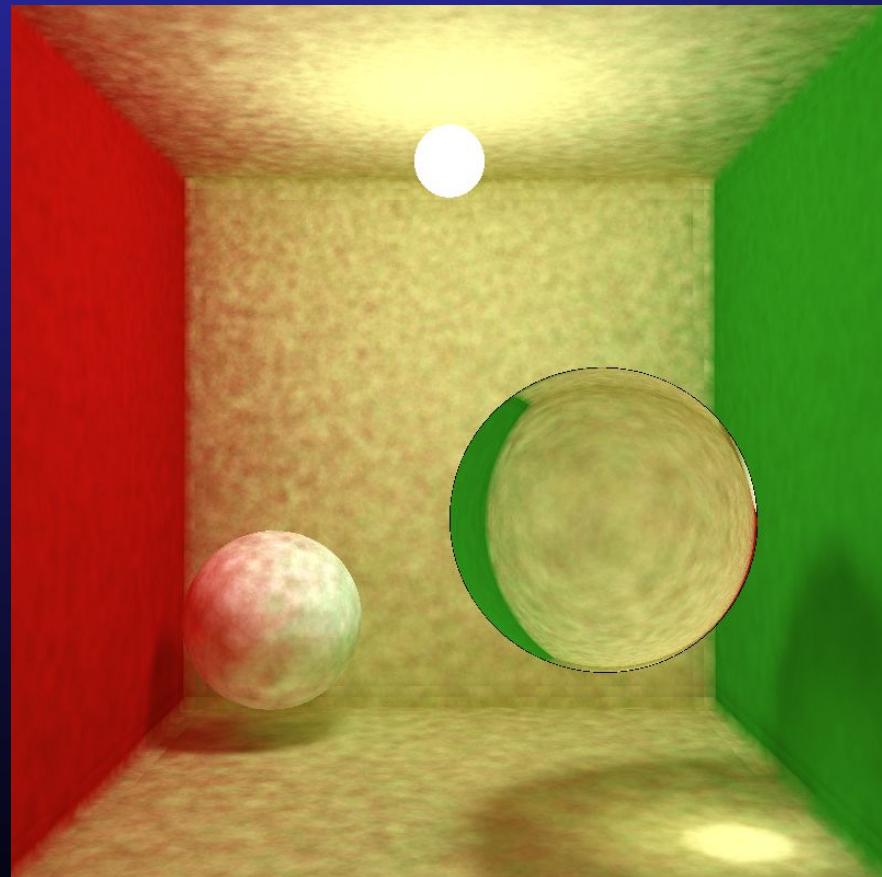


Chapel at Ronchamp

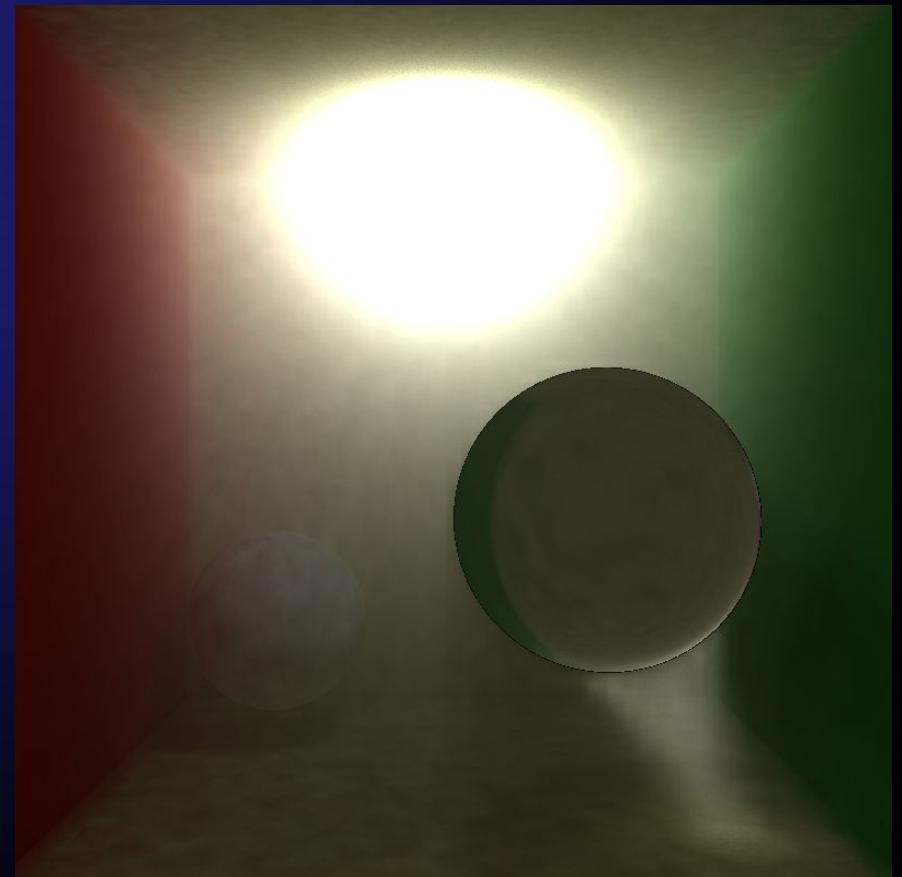


Photon Mapping in Participating Media

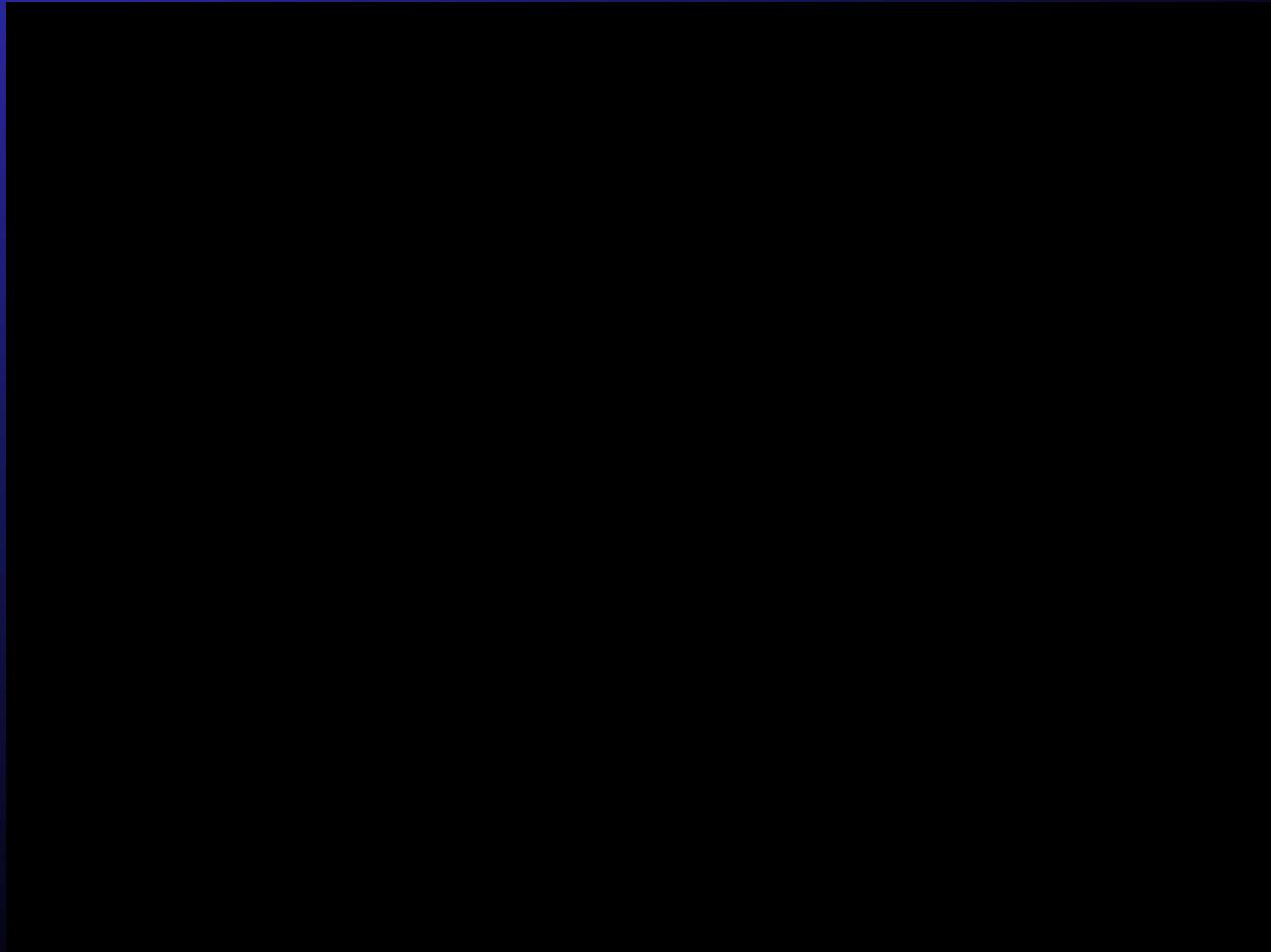
Without participating media



With participating media



Visualizing the Photon Wavefront from the Light Source



Photon Mapping: Pros and Cons

- + An algorithmic improvement on Monte Carlo path tracing
- + Arbitrary geometry – point primitive
- + Suitable for effects like caustics
- + Handles diffuse surfaces efficiently
- + Re-uses illumination information
- + Good for approximate solutions
- + Photon mapping (in static environments) is view independent (walkthroughs)
- + Interactive update in modern implementations on GPUs (see <http://blog.yiningkarlli.com/2013/07/nvidia-optix-lighting-preview-demo.html>)

- High memory consumption
- Low frequency error (blur)
- True real-time performance not quite there
- Too few photons: blurred, bumpy images, lack of detail

Rendering Summary

- Rendering shades object surfaces via ray-tracing, radiosity, and photon maps.
- Ray-tracing establishes visibility. Good for specular materials.
- Radiosity establishes global illumination for diffuse surfaces.
- Photon mapping provides the best of both worlds, providing global illumination without pre-meshing and adding caustics for correct specular and transmission effects.

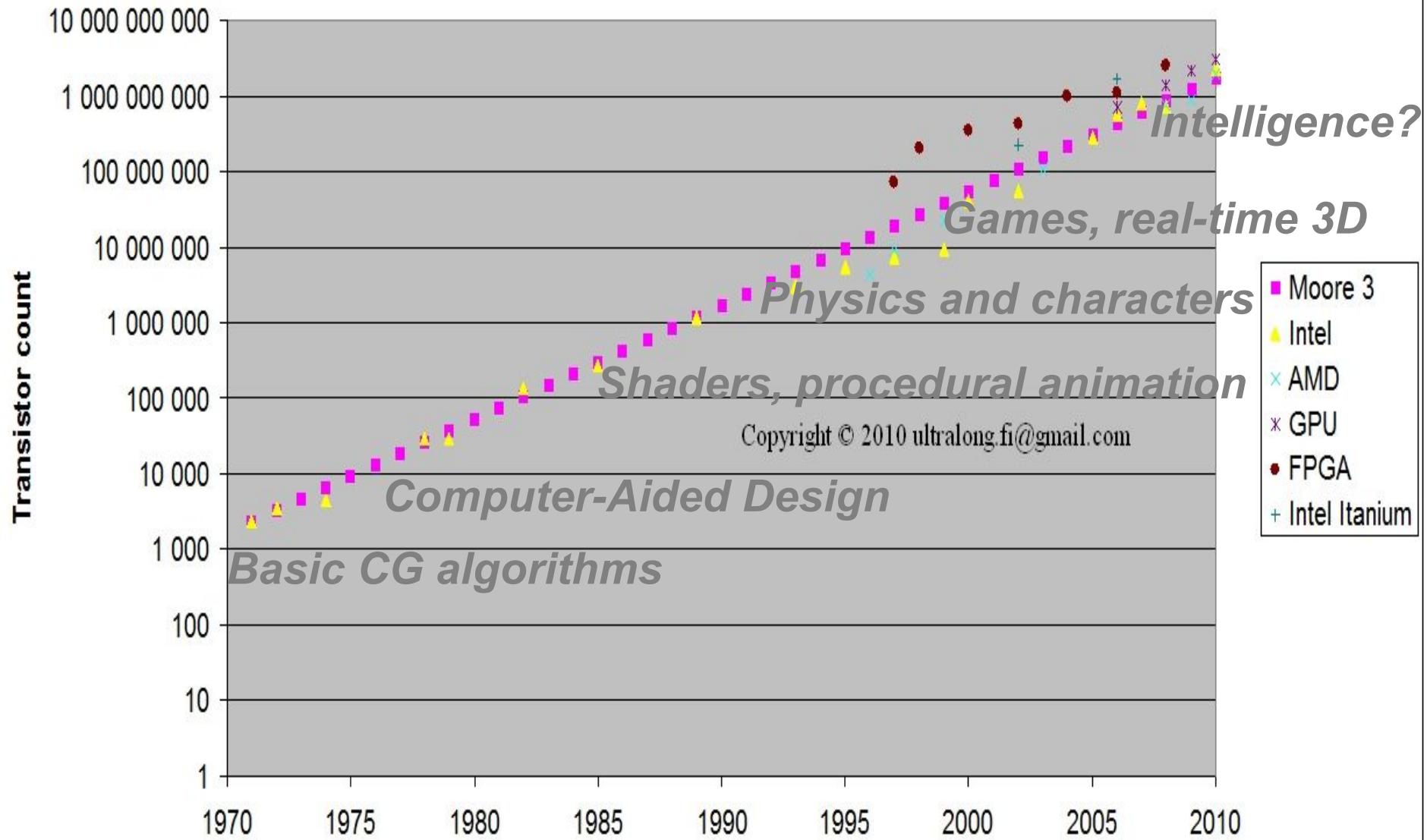
Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading ✓
- Mapping ✓
- 3D Viewing Transformations ✓
- Anti-aliasing & Compositing ✓
- Global Illumination ✓
- Light Fields & HDR ✓
- Path Tracing/Photon Mapping ✓
- Futures

Computer Graphics: The Future

- Applications
- Hardware
- Software
- Challenges

Moore's law vs. actual development



Applications (not so future!)

- Virtual environments
- Character animation
- Facial animation
- Digital photography and video production
- Synchronized animation and sound
- Non-photorealistic rendering
- Interactive medicine and surgery
- Model capture (using computer vision)
- Games and interactive societies
- Special effects; virtual sets, pre-visualization
- Real-time motion capture of people and animals

Hardware

- Continued migration of software techniques onto graphics boards and multi-core CPU architectures.
- Native C parallel programming on board (nVidia CUDA)
- Smaller footprints (mobile platforms).
- Flexible displays (e.g., OLED) = anywhere graphics.
- Cheap HUGE displays (billboards, buildings).
- Wireless ubiquitous connectivity.

Software

- Application software with great value at low or no cost.
- Migration of applications to graphics boards.
- Research developments migrate into commercial products on relatively rapid schedule.
- “Higher level” APIs for virtual humans, tutoring systems, emotion and cognitive processing.
- Mark-up languages for semantic annotation for objects, animation, even characters.

Challenges

- Automated capture of graphics data
 - Shape, deformations, and motion -- connection with computer vision. (Kinect!)
- Virtual environments for training
 - Multiple detailed players, smart agents, world and behavior authoring.
- Realistic *behaving* humans and other characters
 - Movies and games, but also educational software.
 - Connections with cognitive and behavioral psychology.
- Smarter objects and characters
 - Not just geometry, but behavior and function, too.
 - Connections with AI, natural languages, and speech.

Outline

- Great Ideas ✓
- Virtual Environments & Devices ✓
- 3D Perception ✓
- Architecture, Color & Rasterization ✓
- Vector Geometry & Transformations ✓
- 3D Modeling ✓
- Embedding, Hierarchies & Contours ✓
- Model Generation & Deformation ✓
- Visible Surface Algorithms ✓
- Polygon Algorithms ✓
- Image Synthesis & Shading ✓
- Mapping ✓
- 3D Viewing Transformations ✓
- Anti-aliasing & Compositing ✓
- Global Illumination ✓
- Light Fields & HDR ✓
- Path Tracing/Photon Mapping ✓
- Futures ✓

Thank you!

Perspector data - do not edit

