Final Review

Remember…HW #4 due today!

Course web page:
http://goo.gl/EB3aA

Final Notes

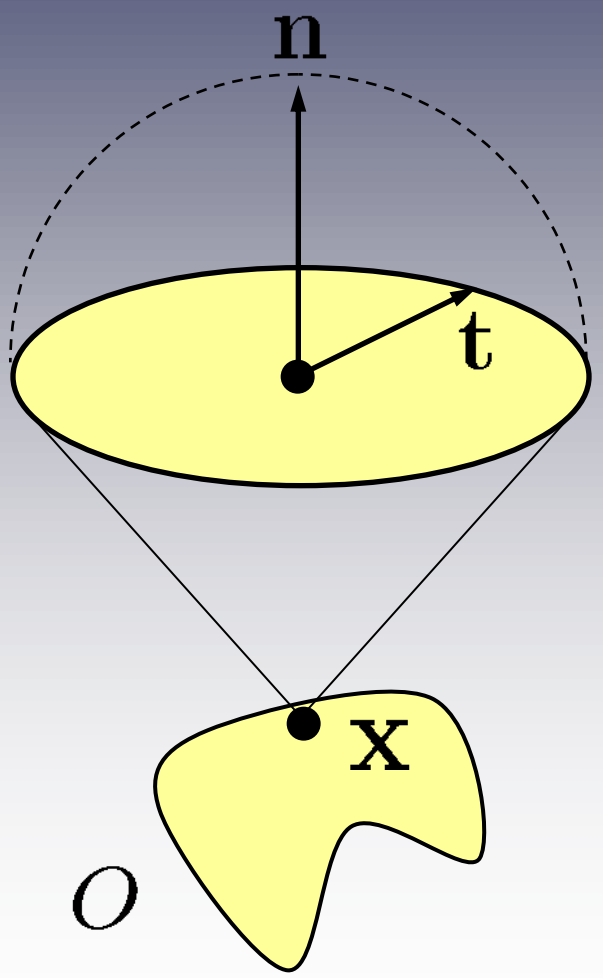
- When/where
  - Next Thursday, May 24
  - 10:30 am-12:30 pm, Gore 303
- Worth 20% of your grade, just like the midterm
- Covers second half of term (from spring break to May 10), but of course first-half knowledge may be assumed
  - A few topics not in book (like ambient occlusion, global illumination)
  - A bit less emphasis on topics addressed in HWs
- Closed book, no calculators, no notes
- Question format similar to midterm, including a few OpenGL- and GLSL-related questions

Final topics

- Shading
- Ray tracing
- Global illumination
- Texture mapping
- GLSL shaders
- Shape modeling

Incoming and Outgoing Light at a Surface

- **Irradiance** E (Wm^{-2})
  - Light arriving at a point on a surface from all visible directions
  - An image samples the irradiance at the pinhole
- **Radiosity** B (Wm^{-2})
  - Light leaving a surface in all directions (per patch)

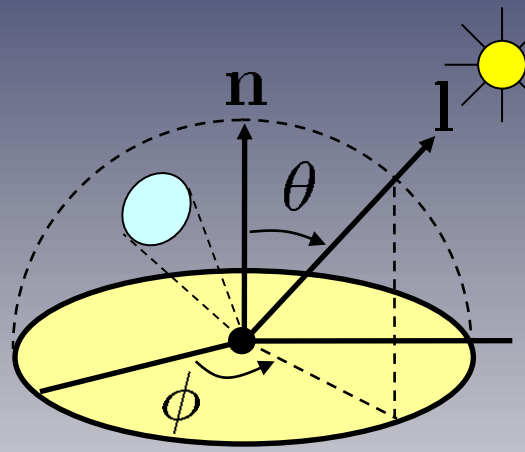


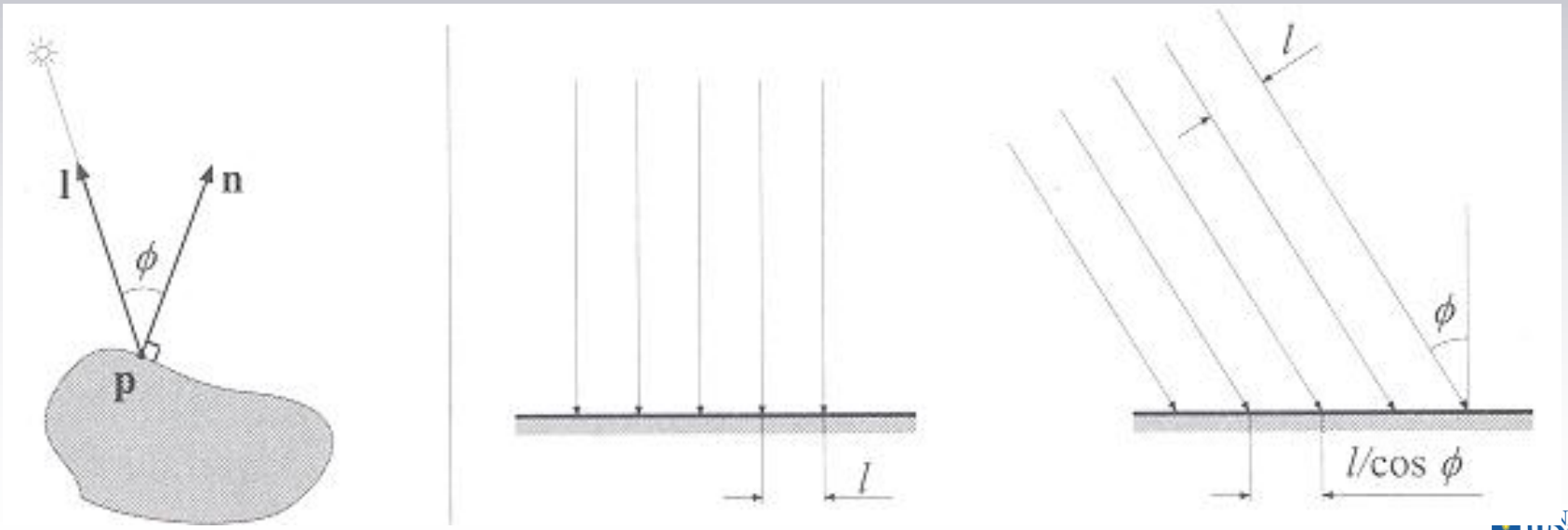
Radiance

- **Radiance** L ($Wm^{-2}sr^{-1}$)
  - Power at a point in space in a given direction, foreshortened, per solid angle
  - Can be incoming or outgoing
  - Does not attenuate with distance in vacuum
- What is stored in a pixel—the light energy arriving along a particular ray at a particular point

Foreshortening

- The more a surface is tilted away, the larger the area light energy is distributed over (and therefore is "diluted")
  - In 2-D, received intensity is proportional to **cosine** of angle between light direction **l** and surface normal **n**
  - Received intensity is greatest when **l** and **n** are parallel
- 3-D foreshortening factor for light coming from direction (θ, φ) is $\cos\theta$



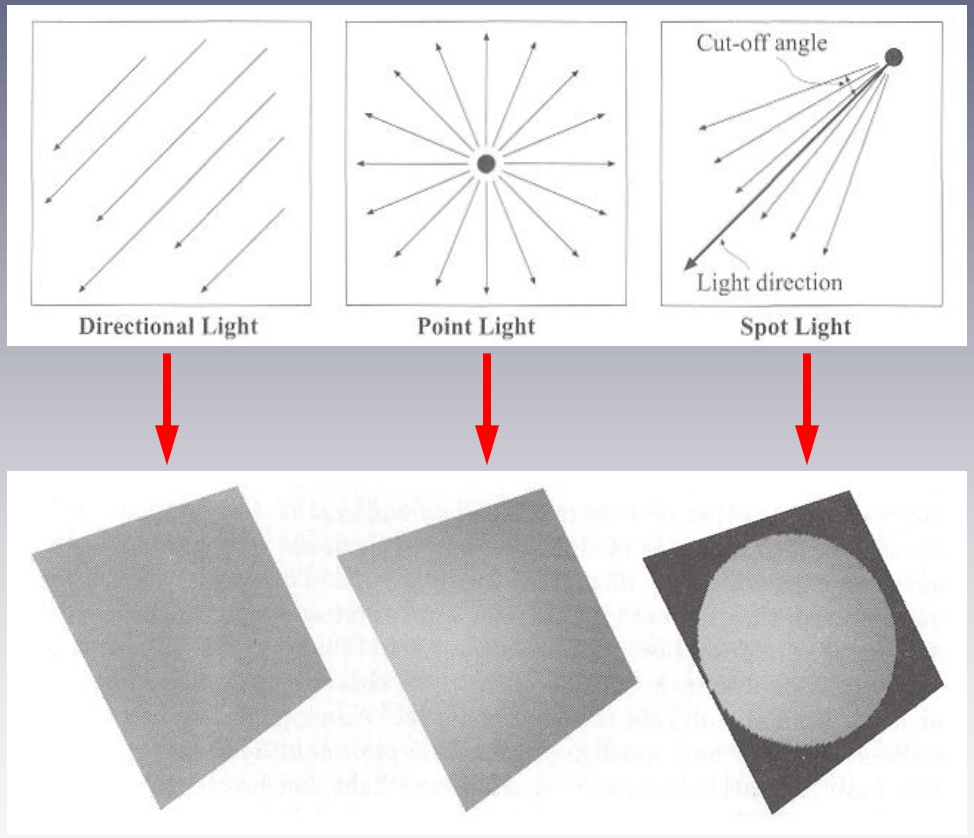


from Akenine-Moller & Haines

Light sources

- Properties
  - Intensity (total radiosity)
  - Color (intensity / wavelength)
- Geometry
  - **Point**: Shoots light in all directions
  - **Spotlight**: Angle-limited point source
  - **Directional**: Source distant enough that light rays are roughly parallel (e.g., like the sun relative to earth)
  - **Area**: Behaves like a continuous configuration of point sources inside, say, a polygon

Some light source types

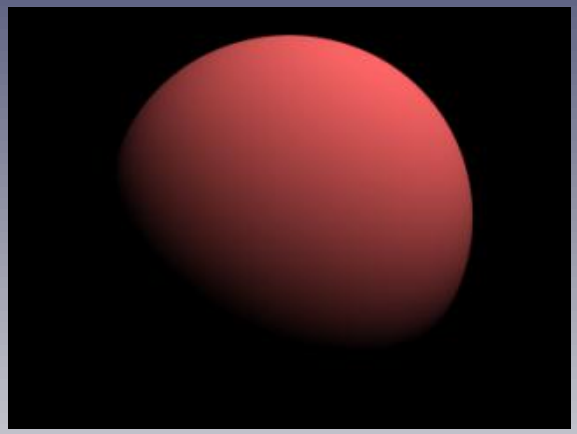


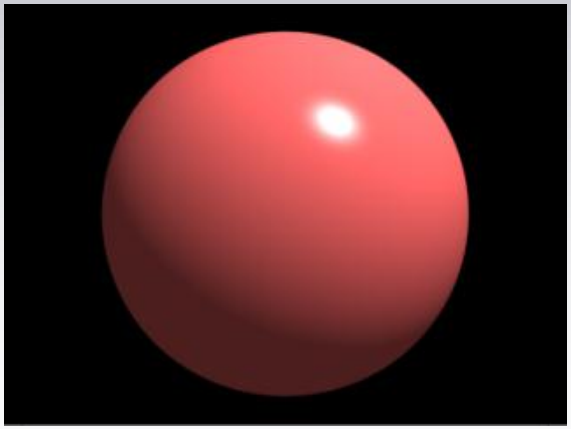
from Akenine-Moller & Haines

Light source types: Induced shad

Object surface properties

- General
  - Light/dark/color
  - Reflectivity (e.g., matte vs. shiny)
- Space-varying pattern
  - I.e., are above characteristics different in different locations?
  - We'll get to this when we cover texture-mapping

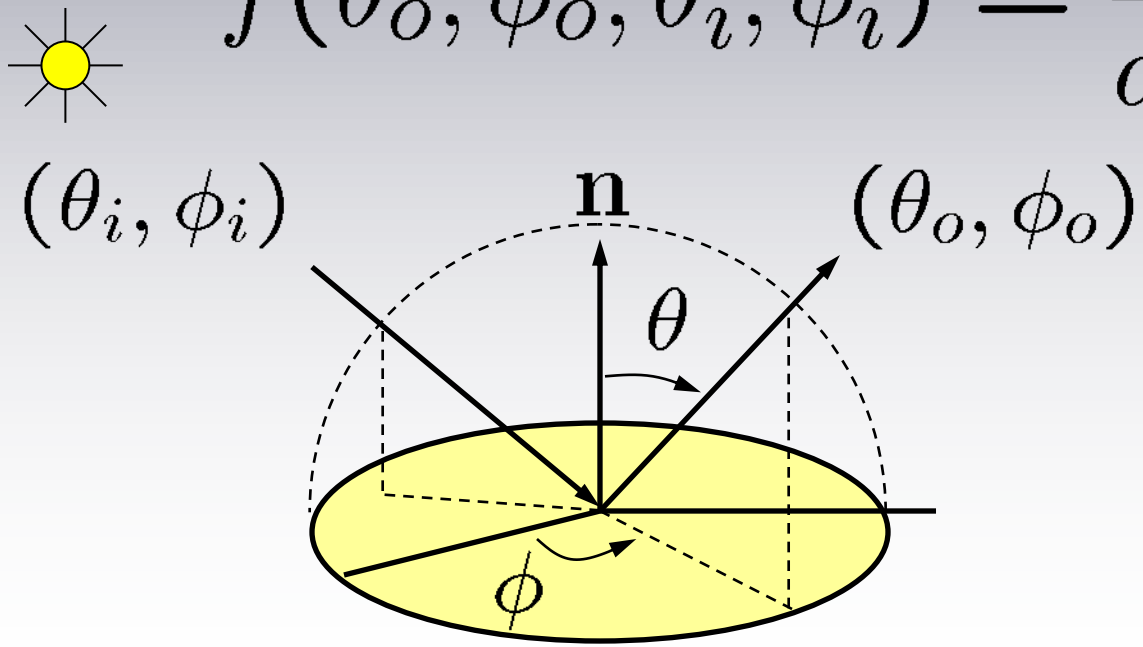




Capturing Surface Properties: BRDF

- **Bidirectional Reflectance Distribution Function** (BRDF): Ratio of outgoing radiance in one direction to incident irradiance from another

$$f(\theta_o, \phi_o, \theta_i, \phi_i) = \frac{L(\theta_o, \phi_o)}{dE(\theta_i, \phi_i)}$$



Reflectance equation

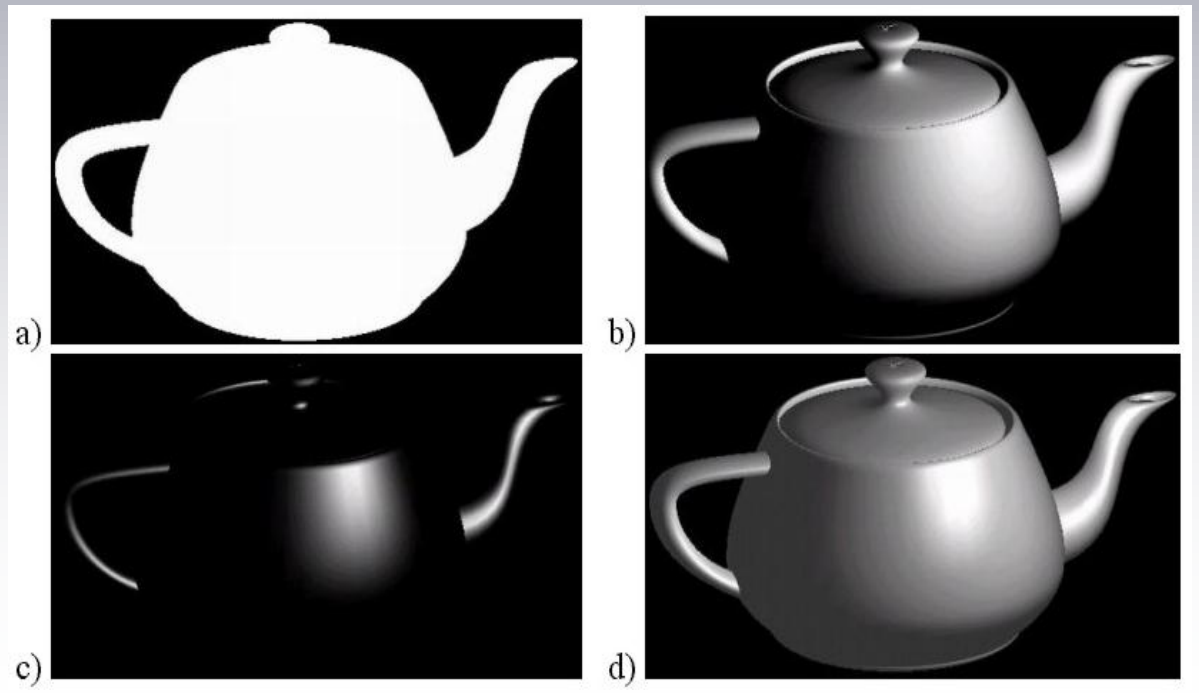
- Radiance for a viewing direction given all incoming light:

$$L_o(\mathbf{x}, \theta_o, \phi_o) = \int_\Omega f(\theta_o, \phi_o, \theta_i, \phi_i) L_i(\mathbf{x}, \theta_i, \phi_i) \cos\theta_i d\omega$$

- This is expensive to compute in general, so the standard local approach is approximation:
  - Approximate incoming light as **ambient** (whole hemisphere) + set of point light sources
  - Approximate BRDF of surface as combination of **diffuse** (matte) and **specular** (shiny) factors

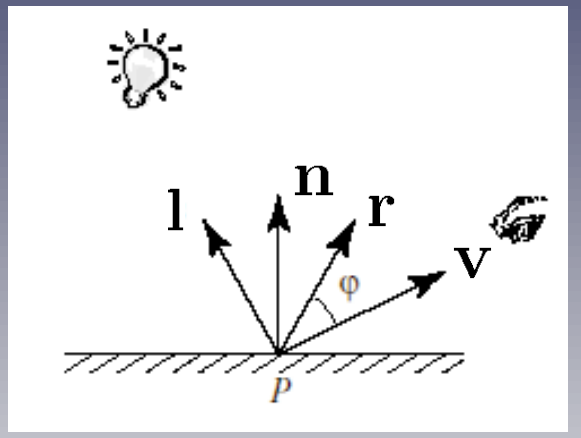
Standard local model for graphics

- Final perceived brightness is a combination of diffuse and specular reflectance, plus an **ambient** term to approximate global lighting effects



Lighting a point

- Let **c** = (r, g, b) be **perceived** material color (called **i** on previous slides), **s**(l) be color of light /
- Sum over all lights /for each color channel (clamp overflow to [0, 1]):



from Hill

$$\mathbf{c}_{total} = \sum_{l} \mathbf{c}_{amb}(l) + \mathbf{c}_{diff}(l) + \mathbf{c}_{spec}(l)$$

$$\mathbf{c}_{amb}(l) = \mathbf{m}_{amb} \otimes \mathbf{s}_{amb}(l)$$

$$\mathbf{c}_{diff}(l) = \max(0, \mathbf{n} \cdot \mathbf{l}(l))\mathbf{m}_{diff} \otimes \mathbf{s}_{diff}(l)$$

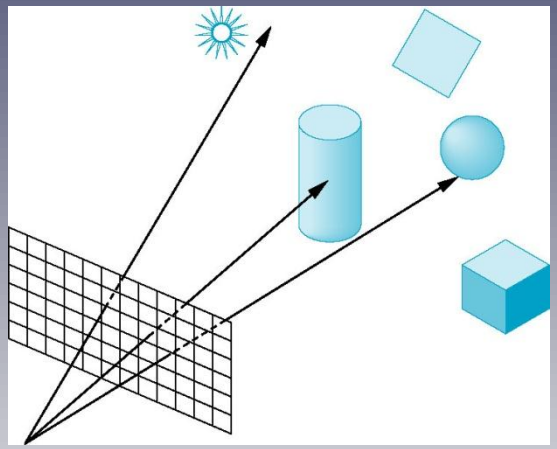
$$\mathbf{c}_{spec}(l) = \max(0, \mathbf{v} \cdot \mathbf{r}(l))^{shine}\mathbf{m}_{spec} \otimes \mathbf{s}_{spec}(l)$$

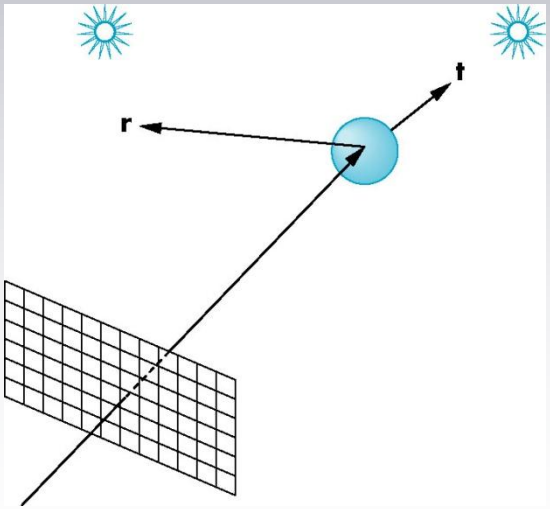
Illumination models

- Interaction between light sources and objects in scene that results in perception of intensity and color at eye
- Local vs. global models
  - Local illumination: Perception of a particular primitive only depends on light sources **directly** affecting that one primitive
    - Geometry
    - Material properties
  - Global illumination: Also take into account **indirect** effects on light of other objects in the scene
    - Shadows cast
    - Light reflected/refracted

Backward Ray "Following": Types

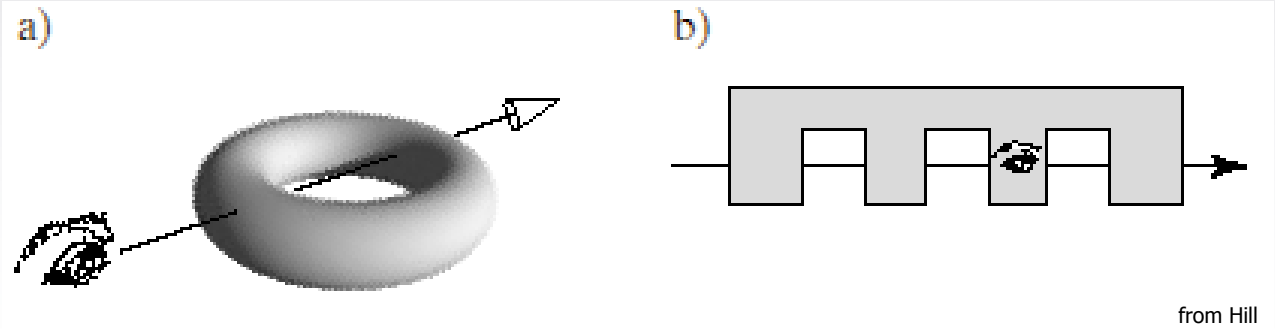
- **Ray casting**: Compute illumination at first intersected surface point only
  - Takes care of hidden surface elimination
- **Ray tracing**: Recursively spawn rays at hit points to simulate reflection, refraction, etc.





Does Ray Intersect any Scene Primitives?

- Test each primitive in scene for intersection individually
- Different methods for different kinds of primitives
  - Polygon
  - Sphere
  - Cylinder, torus
  - Etc.
- Make sure intersection point is **in front of eye** and **nearest one**



from Hill

Ray-Sphere Intersection I

- Combine implicit definition of sphere

$$|\mathbf{p} - \mathbf{p}_c|^2 - r^2 = 0$$

with ray equation

$$\mathbf{p} = \mathbf{o} + t\mathbf{d}$$

(where **d** is a unit vector) to get:

$$|\mathbf{o} + t\mathbf{d} - \mathbf{p}_c|^2 - r^2 = 0$$

Ray-Sphere Intersection II

- Substitute $\Delta\mathbf{p} = \mathbf{p}_c - \mathbf{o}$ and use identity

$$|\mathbf{a}+\mathbf{b}|^2 = |\mathbf{a}|^2 + 2\mathbf{a}\cdot\mathbf{b} + |\mathbf{b}|^2$$

to solve for t, resulting in a quadratic equation with roots given by:

$$t = \mathbf{d}\cdot\Delta\mathbf{p} \pm \sqrt{(\mathbf{d}\cdot\Delta\mathbf{p})^2 - (|\Delta\mathbf{p}|^2 - r^2)}$$

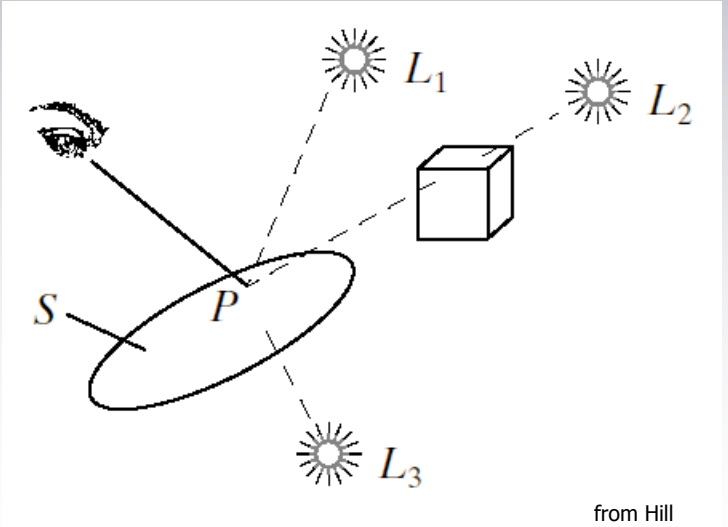
- Notes
  - Real solutions mean there actually are 1 or 2 intersections
  - Negative solutions are behind eye

Shadow Rays

- For point **p** being locally shaded, only add diffuse & specular components for light l if light is not occluded (i.e., blocked)
- Test for occlusion of l for **p**:
  - Spawn **shadow ray** for l with origin **p**, direction **l**(l)
  - Check whether shadow ray intersects any scene object
  - Intersection only "counts" if:

$$0 < t < |\mathbf{p}_l - \mathbf{p}|$$

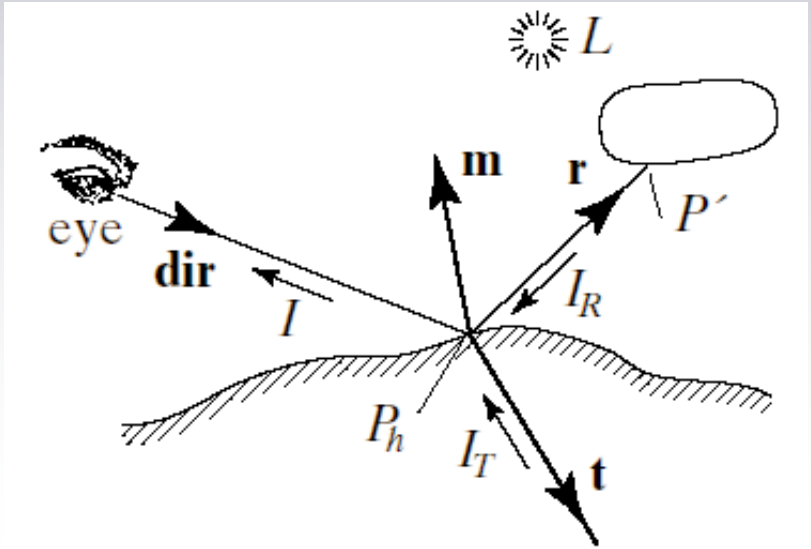
- More details in Shirley, Chap. 10.5



from Hill

Ray Tracing

- Model: Perceived color at point **p** is an additive combination of local illumination (e.g., Phong), reflection, and refraction effects
- Compute reflection, refraction contributions by **tracing** respective rays back from **p** to surfaces they came from and evaluating local illumination at those locations
- Apply operation **recursively** to some maximum depth to get:
  - Reflections of reflections of ...
  - Refractions of refractions of ...
  - And of course mixtures of the two



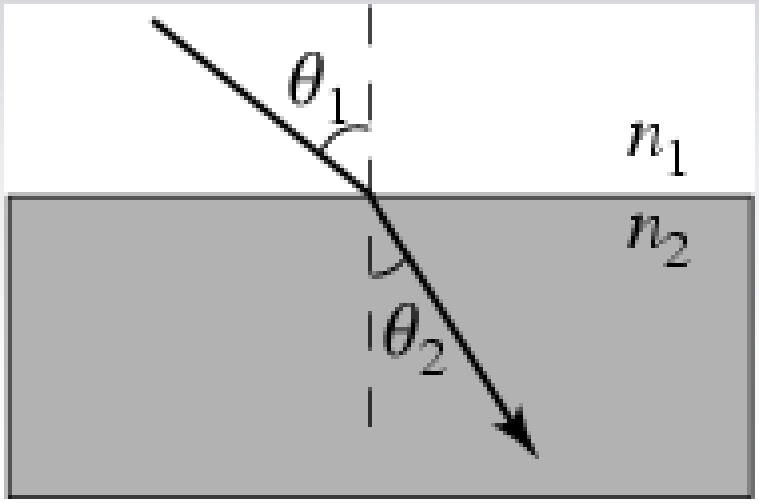
Ray Tracing Reflection Formula

- The formula used for Phong illumination is not what we want here because our incident ray **v** is pointing **in** toward the surface, whereas the light direction **l** was pointed **away** from the surface
- So just negate the formula to get:

$$\mathbf{r} = \mathbf{v} - 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n}$$

Refraction

- Definition: Bending of light ray as it crosses interface between media (e.g., air $\rightarrow$ glass or vice versa)
- Index of refraction (IOR) n for a medium: Ratio of speed of light in vacuum to that in medium (wavelength-dependent $\Rightarrow$ prisms)
  - By definition, $n \geq 1$
  - Examples: n_{air} (1.00) < n_{water} (1.33) < n_{glass} (1.52)



θ_1: Angle of incidence

θ_2: Angle of refraction

Basic Ray Tracing: Notes

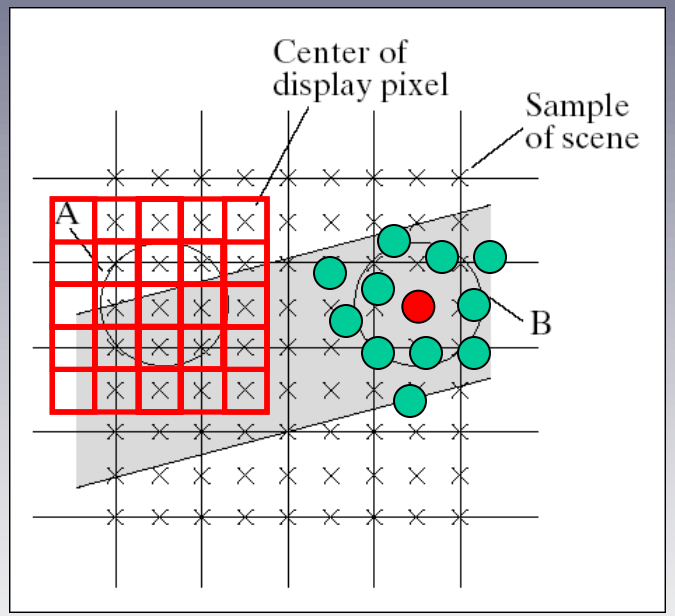
- Global illumination effects simulated by basic algorithm are shadows, purely specular reflection/transmission
- Some outstanding issues
  - Aliasing, aka jaggies
  - Shadows have sharp edges, which is unrealistic
  - No diffuse reflection from other objects
- Intersection calculations are expensive, and even more so for more complex objects
  - Not currently suitable for real-time (i.e., games)

Distributed (aka "distribution") Ray Tracing (DRT)

- Basic idea: Use multiple eye rays for each pixel rendered or multiple recursive rays at intersections
- Application #1: Improving image quality via **anti-aliasing**
  - Supersampling: Shoot multiple nearby eye rays per pixel and combine colors
  - Uniform vs. adaptive: Constant number of rays or change in areas where image is changing more quickly

Supersampling

- Rasterize at higher resolution
  - Regular grid pattern around each "normal" image pixel
  - Irregular **jittered** sampling pattern reduces artifacts
- Combine multiple samples into one pixel via **weighted average**
  - "Box" filter: All samples associated with a pixel have equal weight (i.e., directly take their average)
  - Gaussian/cone filter: Sample weights inversely proportional to distance from associated pixel



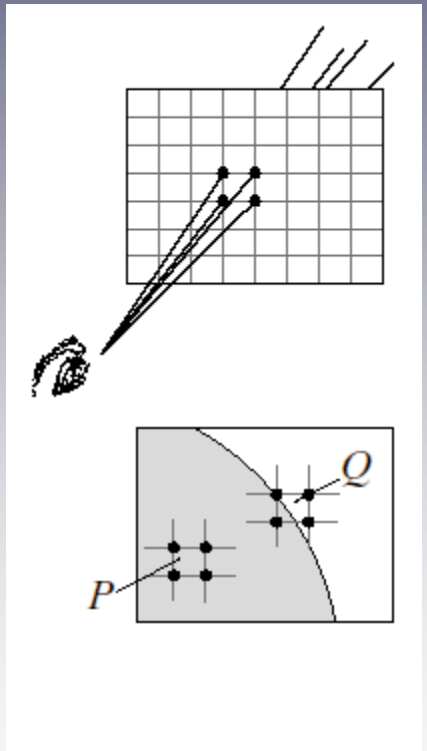
from Hill

Regular supersampling with 2x frequency

Jittered supersampling

Adaptive Supersampling (Whitted's method)

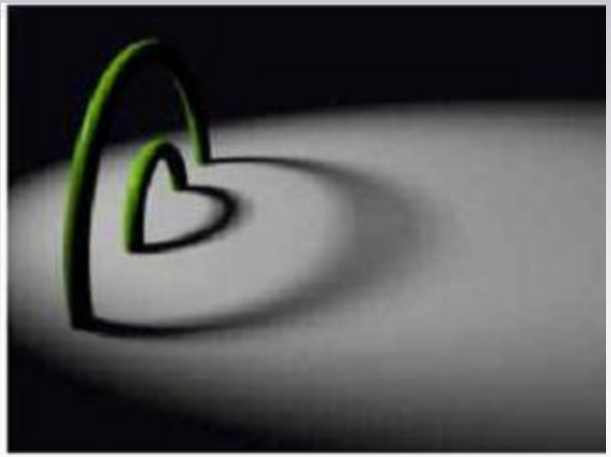
- Shoot rays through 4 pixel corners and collect colors
- Provisional color for entire pixel is **average** of corner contributions
  - If you stop here, the only overhead vs. center-of-pixel ray-tracing is another row, column of rays
- If any corner's color is too different, **subdivide** pixel into quadrants and recurse on quadrants
- Details
  - Subdivide if any corner is more than 25% different from average (try experimenting with different thresholds here)
  - Maximum depth of 2 subdivisions sufficient



from Hill

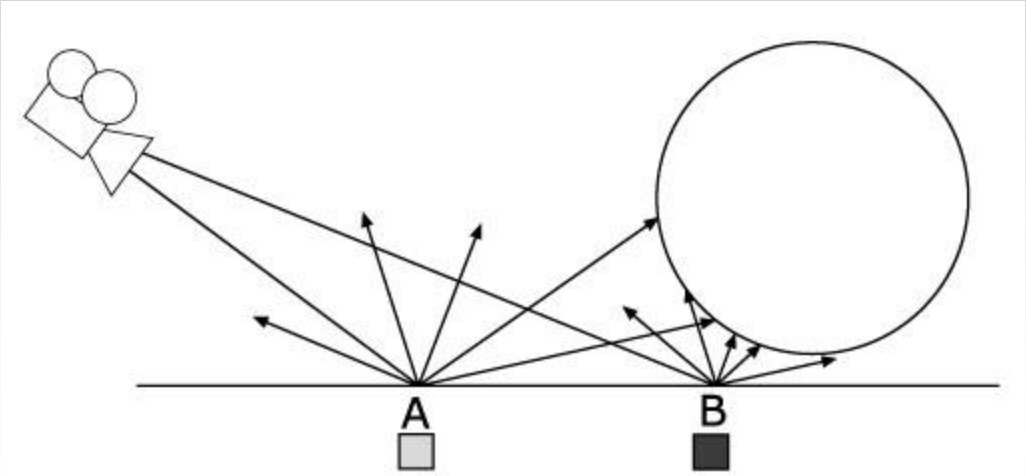
DRT: Soft Shadows

- For point light sources, sending a single shadow ray toward each is reasonable
  - But this gives hard-edged shadows
- Simulating soft shadows
  - Model each light source as sphere
  - Send multiple jittered shadow rays toward a light sphere; use **fraction** that reach it to attenuate color



DRT: Ambient Occlusion

- Extension of shadow ray idea—not every point should get full ambient illumination
- Cast random rays from each surface point to estimate percent of sky hemisphere that is visible (i.e., is there any intersection within a certain distance)
  - May use cosine weighting/distribution for foreshortening



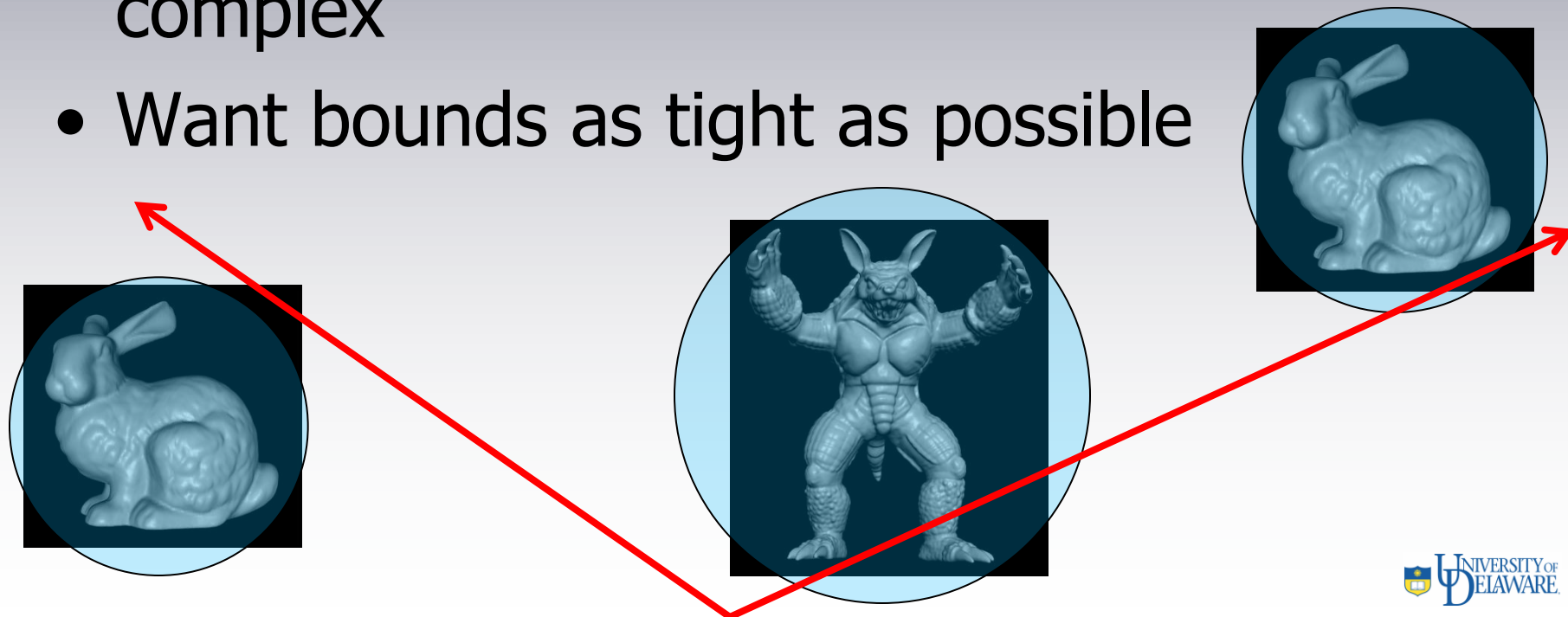
DRT: Glossy Reflections

- Analog of hard shadows are "sharp reflections"—every reflective surface acts like a perfect mirror
- To get glossy or blurry reflections, send out multiple jittered reflection rays and average their colors

Why is the reflection sharper at the top?

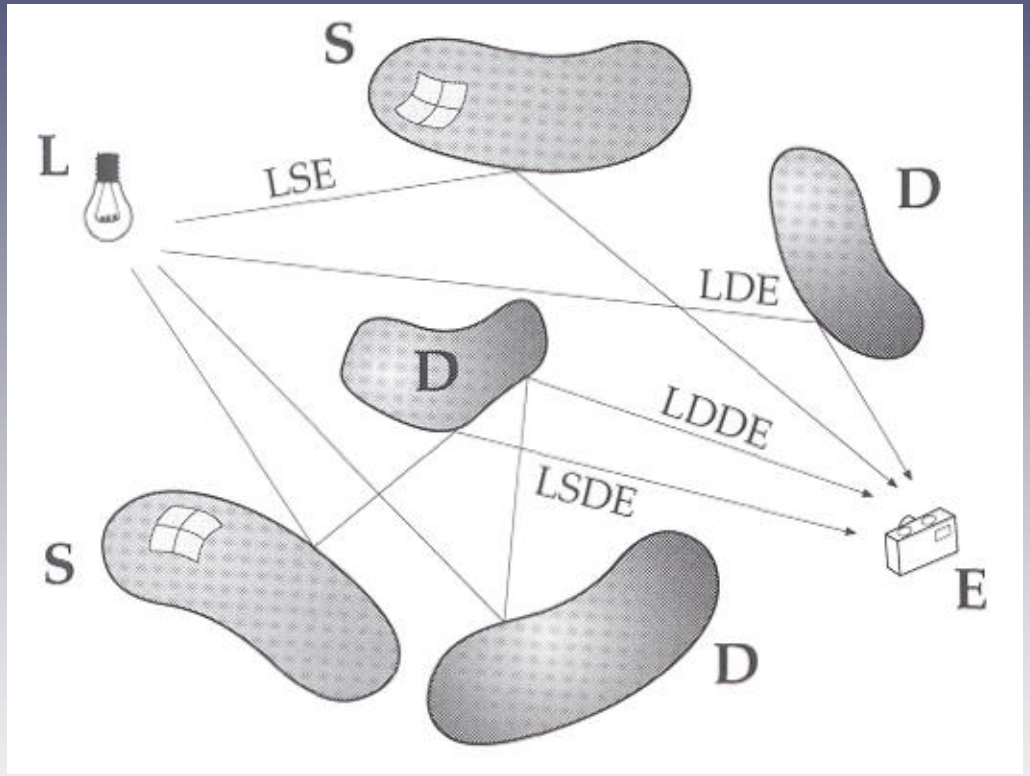
Bounding Volumes

- Idea: enclose complex objects (i.e., .obj models) in simpler ones (i.e., spheres) and test simple intersection before complex
- Want bounds as tight as possible



Light Paths

- Consider the path that a light ray might take through a scene between the light source **L** and the eye **E**
- It may interact with multiple diffuse (**D**) and specular (**S**) objects along the way

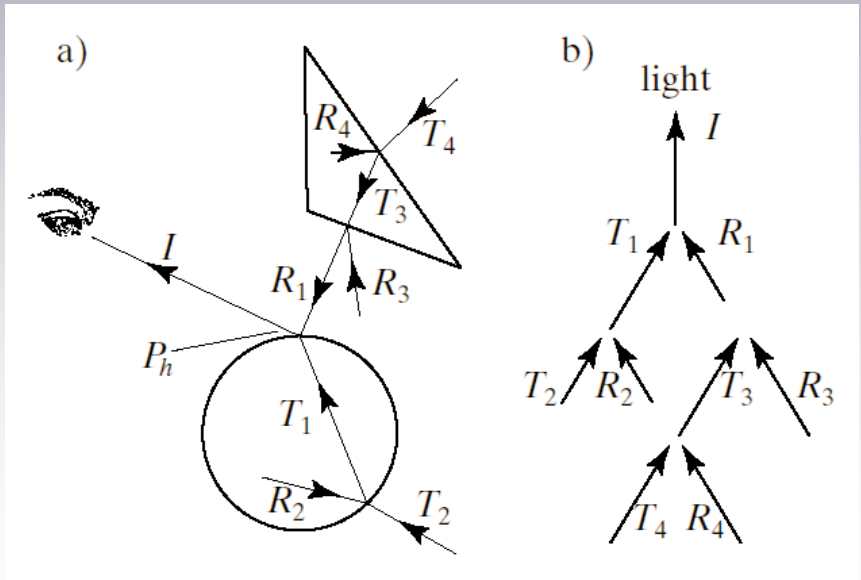


from Sillion & Puech

- We can describe this series of interactions with the regular expression **L** (**D** | **S**)* **E**
  - (If a surface is a mix of **D** and **S**, the combination is additive so it is still OK to treat in this manner)

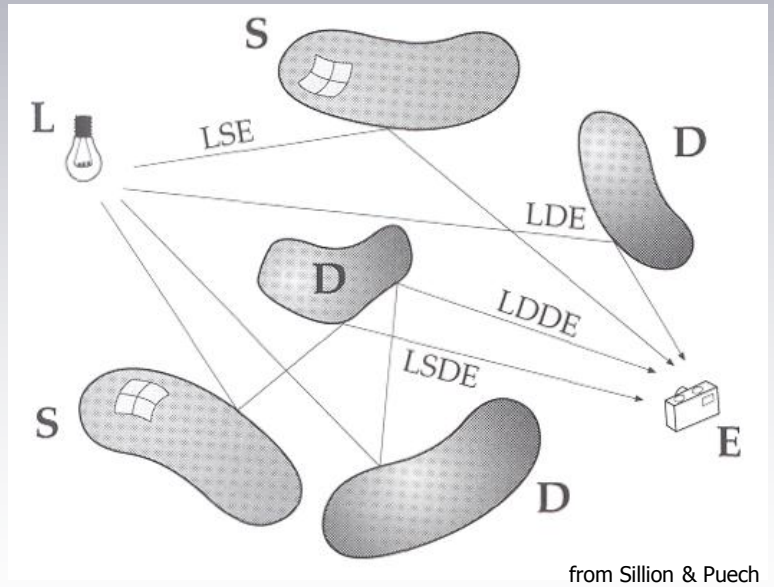
Light Paths: Examples

- Direct visualization of the light: **LE**
- Local illumination: **LDE**, **LSE**
- Ray tracing: **LS*E**, **LDS*E**



from Hill

Ray tracing light paths

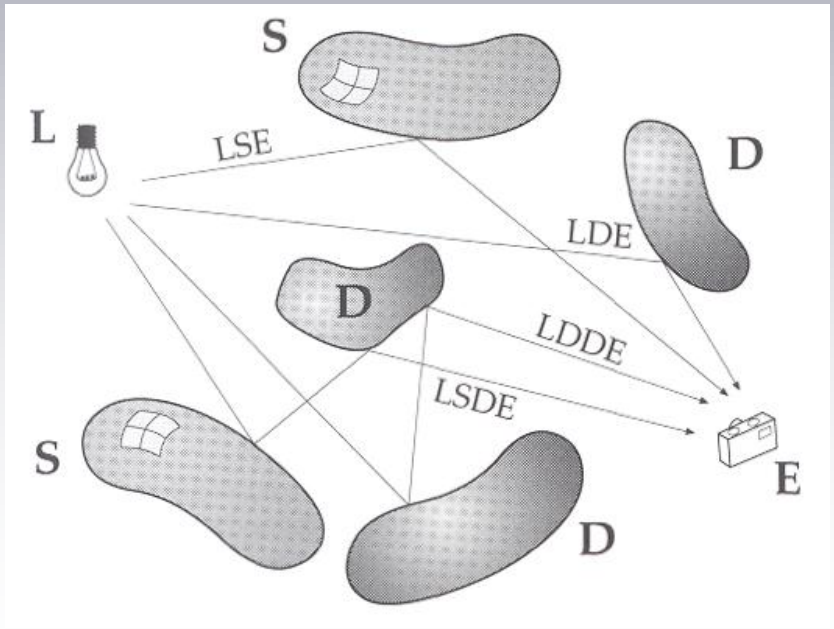


from Sillion & Puech

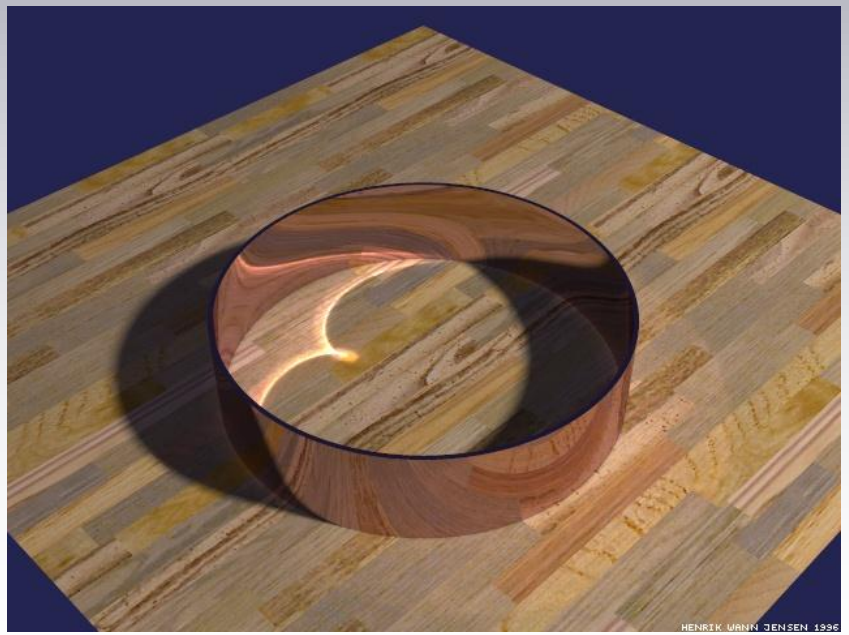
General light paths

Caustics

- Definition: (Concentrated) specular reflection/refraction onto a diffuse surface
  - In simplest form, follow an **LSDE** path
- Standard ray tracing **cannot** handle caustics—only paths described by **LDS*E**



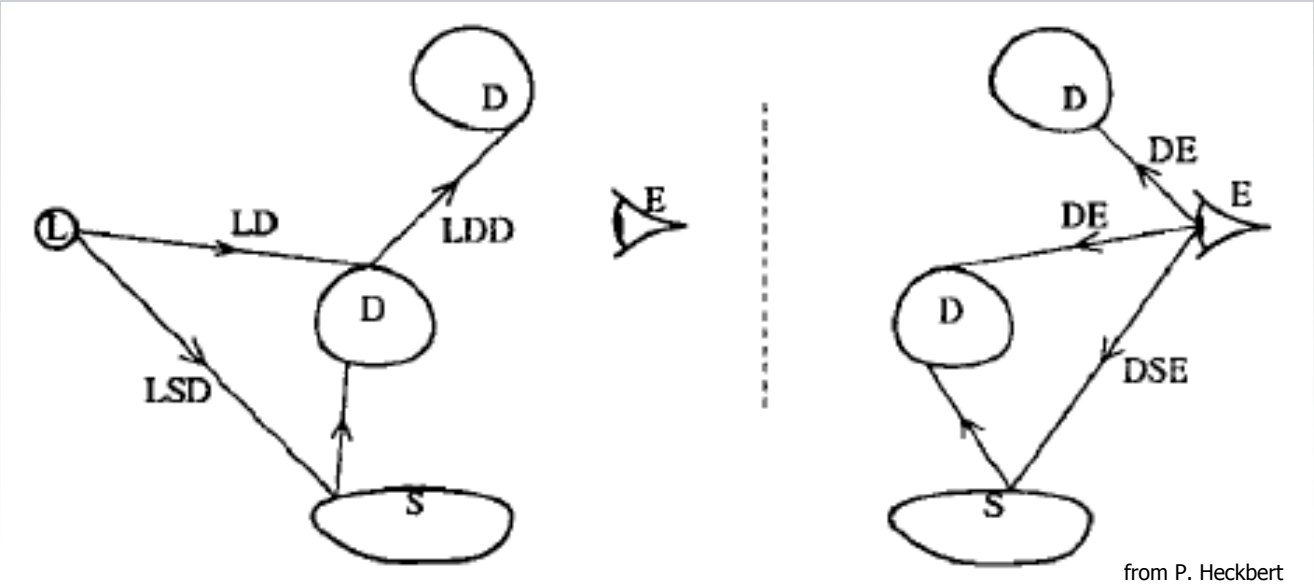
from Sillion & Puech



courtesy of H. Wann Jensen

Bidirectional Ray Tracing (P. Heckbert, 1990)

- Idea: Trace forward **light rays** into scene as well as backward **eye rays**
- At diffuse surfaces, light rays additively "deposit" photons in **radiosity textures**, or "rexes", where they are accessed up by eye rays
  - Summation approximates integral term in radiance computation
  - Light rays carry information on specular surface locations—they have no uncertainty



from P. Heckbert

Photon Mapping (H. Jensen, 1996)

- Two-pass algorithm somewhat like bidirectional ray tracing, but photons stored differently
- 1st pass: Build photon map
  - Shoot random rays from light(s) into scene
  - Each photon carries fraction of light's power
  - Follow specular bounces, but store photons in map at each diffuse surface hit (or scattering event)
- 2nd pass: Render scene
  - Modified ray tracing: follow eye rays into scene
  - Use photons near each intersection to compute light

What is Texture Mapping?

- Spatially-varying modification of surface appearance at the pixel level
- Characteristics
  - Color
  - Shininess
  - Transparency
  - Bumpiness
  - Etc.

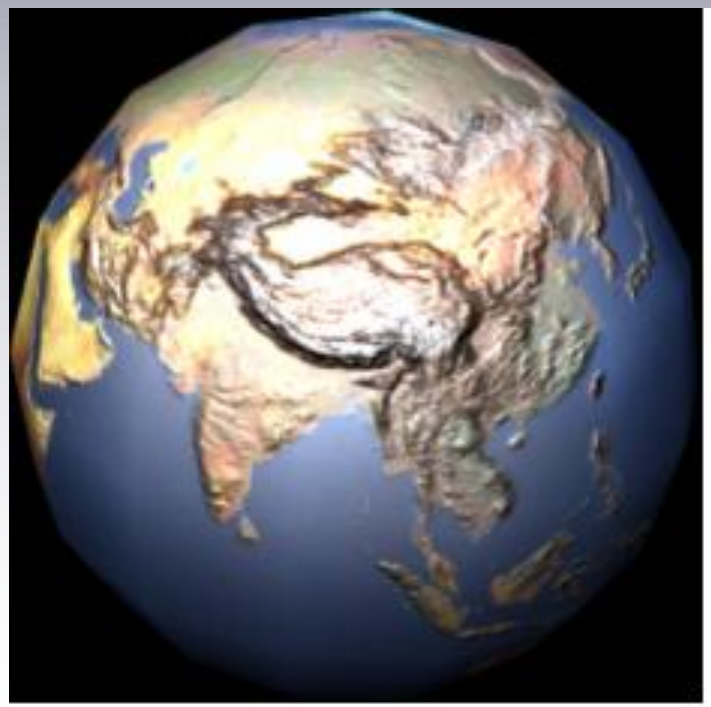
from Hill

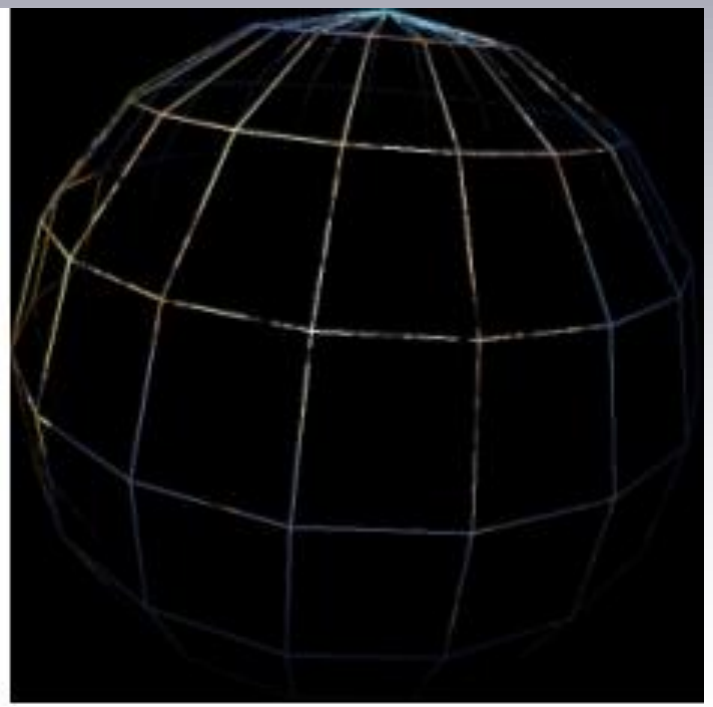
Bump Mapping

- So far we've been thinking of textures modulating color and transparency only
  - Billboards, decals, lightmaps, etc.
- But any other per-pixel properties are fair game...
- Pixel **normals** usually smoothly varying
  - Computed at vertices for Gouraud shading; color interpolated
  - Interpolated from vertices for Phong shading
- Textures allow setting per-pixel normal with a **bump map**

Bump mapping: Why?

- Can get a lot more surface detail without expense of more object vertices to light, transform

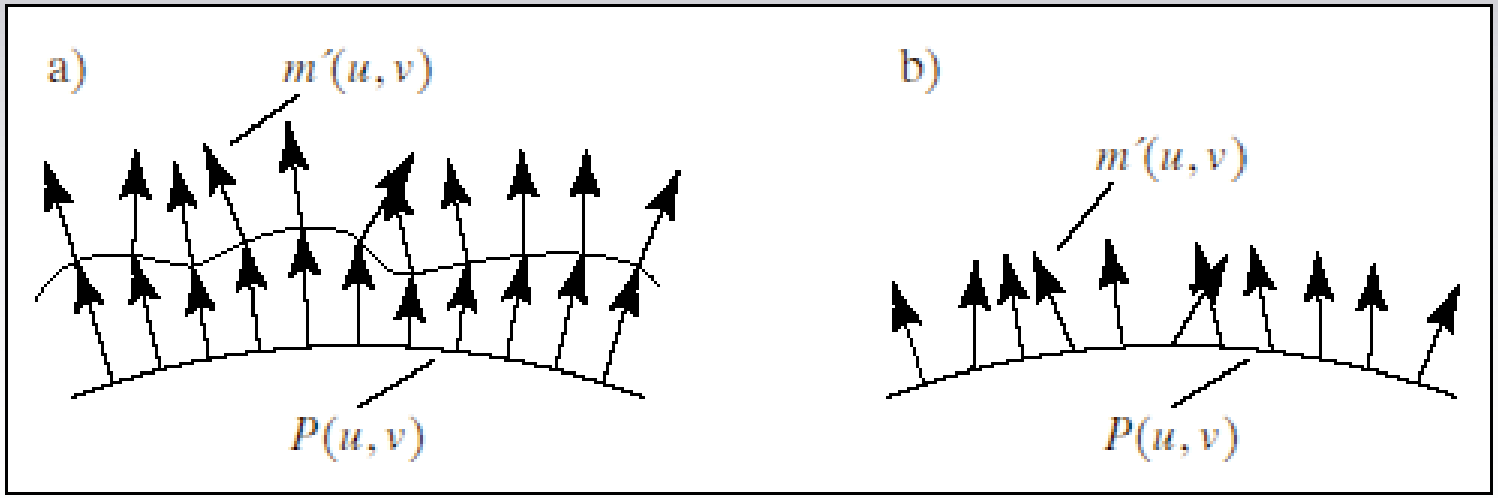




courtesy of Nvidia

Bump Mapping: How?

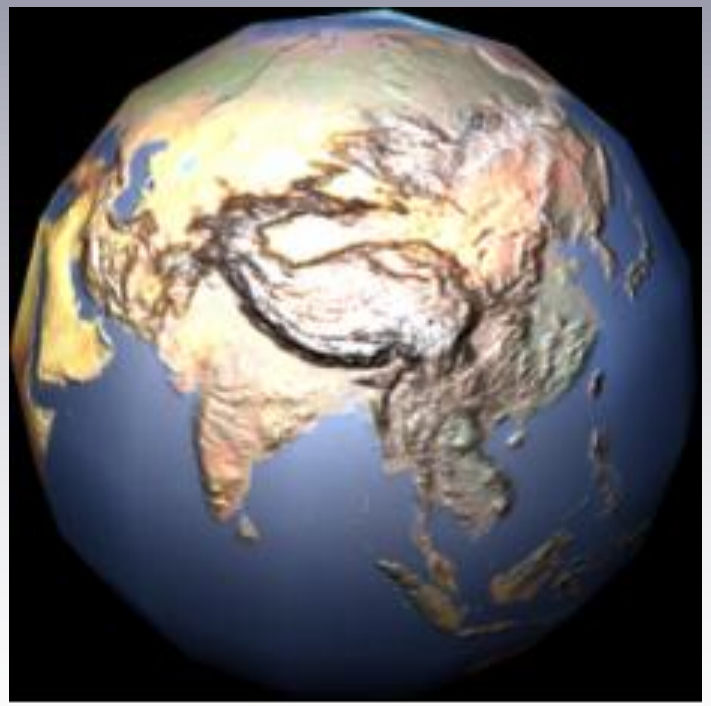
- Idea: Perturb pixel normals **n**(u, v) derived from object geometry to get additional detail for shading
- Compute lighting per pixel (like Phong)



from Hill

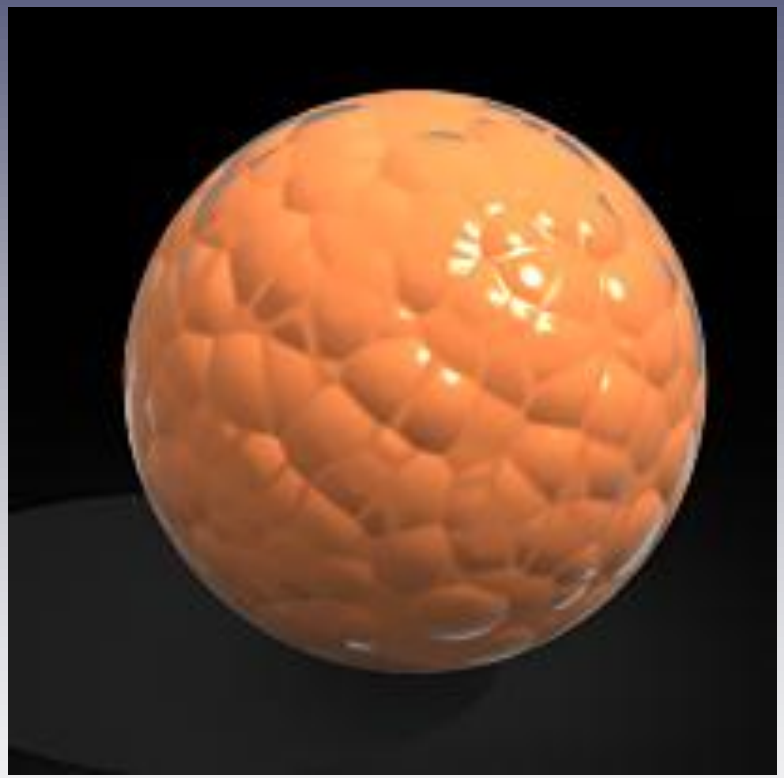
Bump mapping: Issues

- Bumps don't cast shadows
- Geometry doesn't change, so silhouette of object is unaffected
- Textures can be used modify underlying geometry with **displacement maps**

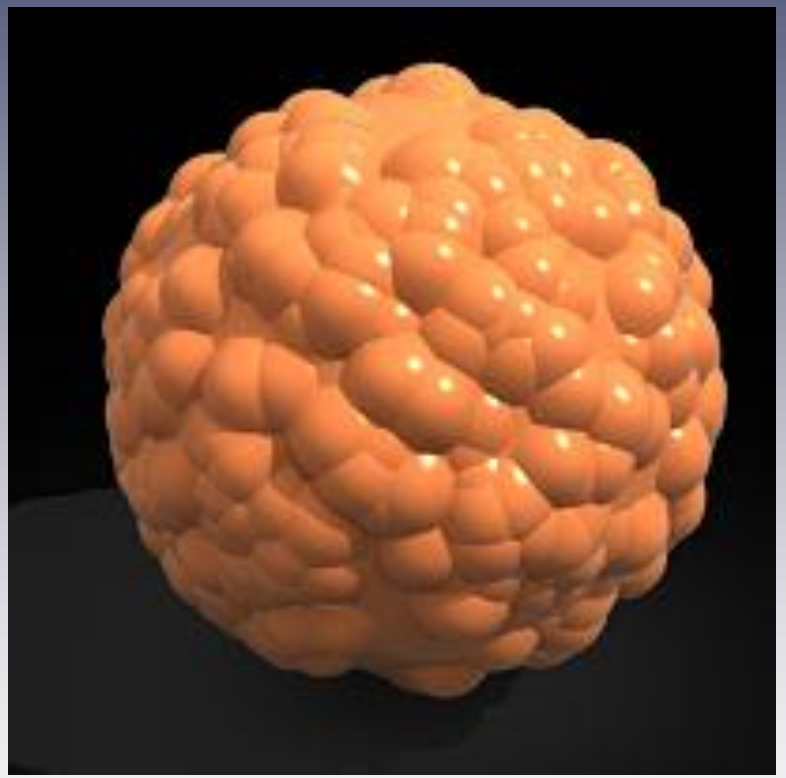


courtesy of Nvidia

Displacement Mapping



Bump mapping



Displacement mapping

courtesy of spot3d.com

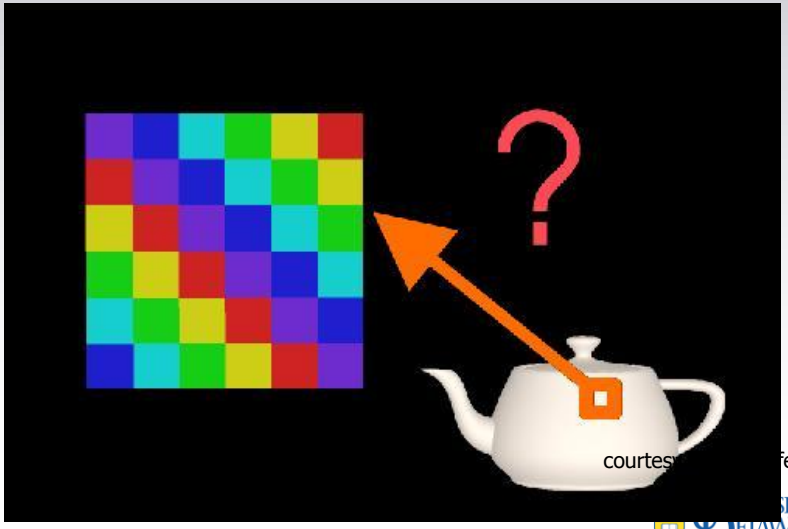
Texture mapping: Steps

- **Creation**: Where does the texture image come from?
- **Geometry**: Transformation from 3-D shape locations to 2-D texture image coordinates
- **Rasterization**: What to draw at each pixel
  - E.g., bilinear interpolation vs. nearest-neighbor

Texturing Pipeline (Geometry + Rasterization)

1. Compute object space location (x, y, z) from screen space location (i, j)
2. Use **projector** function to obtain object surface coordinates (u, v)
3. **Corresponder** function to find texel coordinates (s, t)
   - Scale, shift, wrap like viewport transform in geometry pipeline
4. Filter texel at (s, t)
5. Modify pixel (i, j)

Rasterization

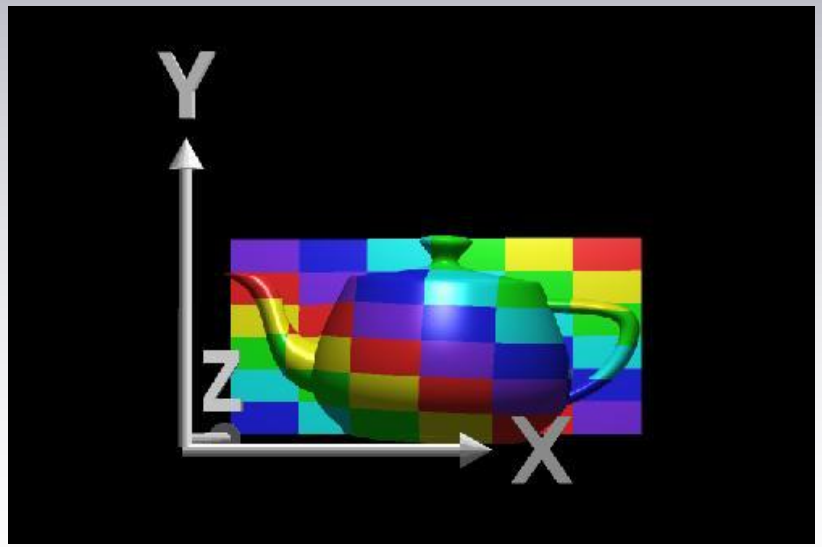


courtes fe

list adapted from Akenine-Moller & Haines

Projector Functions

- Want way to get from 3-D point to 2-D surface coordinates as an intermediate step
- Idea: Project complex object onto **simple object**'s surface with parallel or perspective projection (focal point inside object)
  - Plane
  - Cylinder
  - Sphere
  - Cube
  - Mesh: piecewise

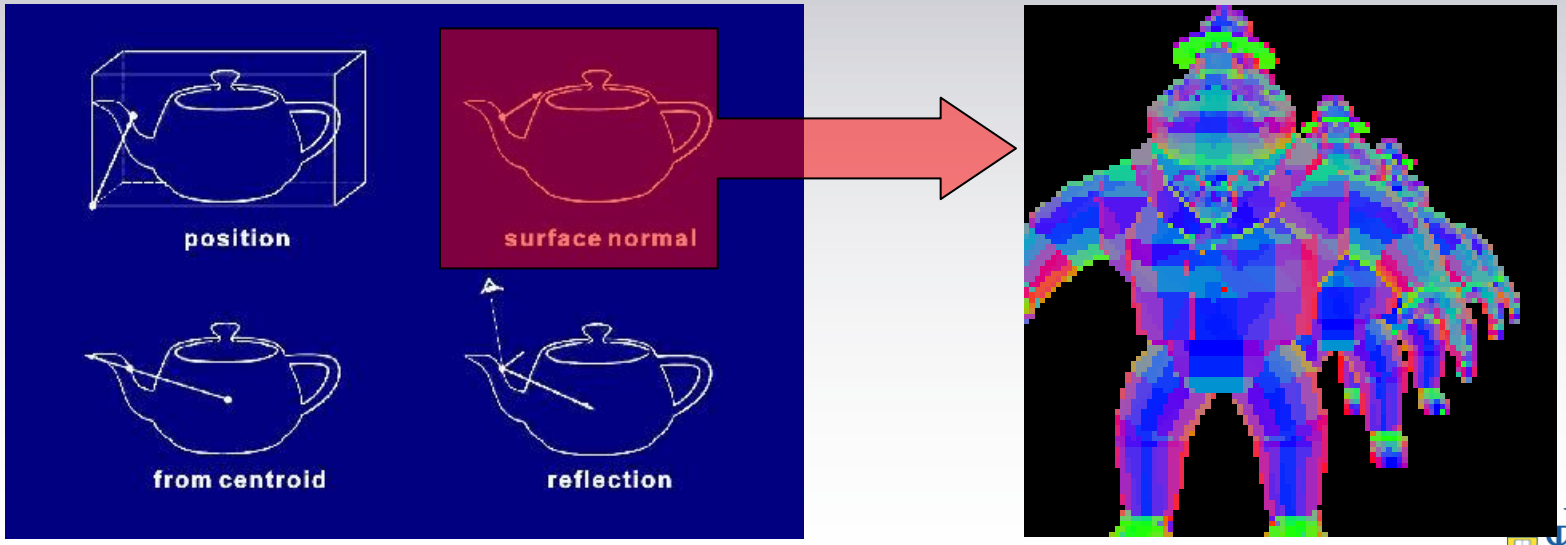


courtesy of R. Wolfe

Planar projector

Projecting in non-standard directions

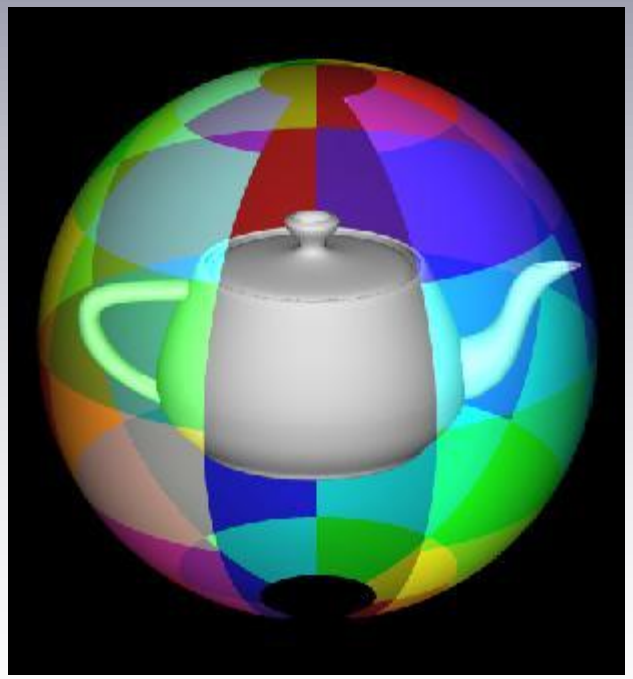
- Don't have to project ray from object center through position (x, y, z)—can use any attribute of that position. For example:
  - Ray comes from another location
  - Ray is surface normal **n** at (x, y, z)
  - Ray is reflection-from-eye vector **r** at (x, y, z)
  - Etc.

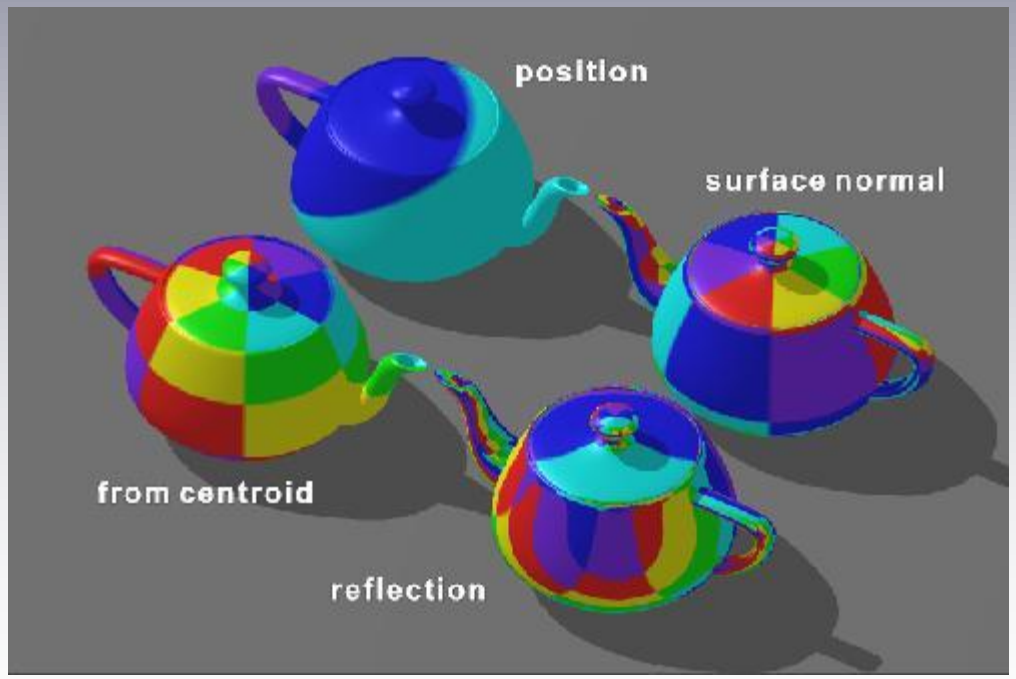


courtesy of R. Wolfe

Projecting in non-standard directions

- This can lead to interesting or informative effects



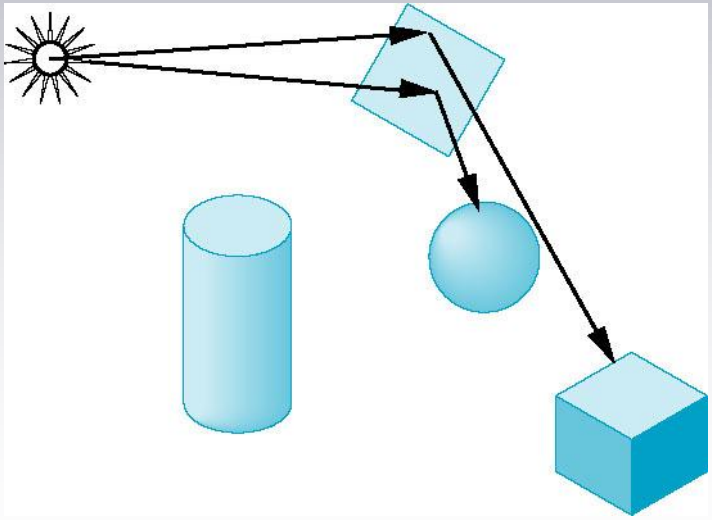


courtesy of R. Wolfe

Different ray directions for a spherical projector

Environment/Reflection Mapping

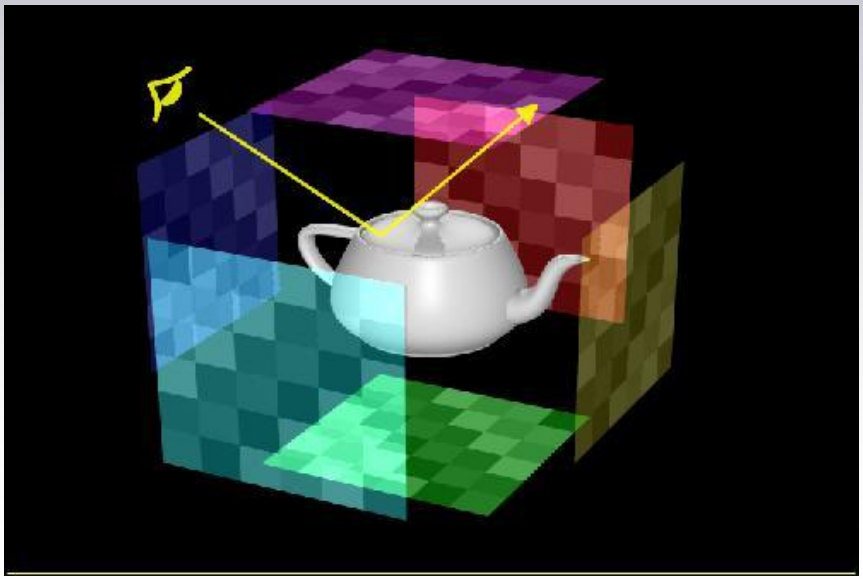
- Problem: To render pixel on mirrored surface correctly, we need to follow reflection of eye vector back to first intersection with another surface and get its color
- This is an expensive procedure with ray tracing
- Idea: Approximate with texture mapping



from Angel

Environment mapping: Details

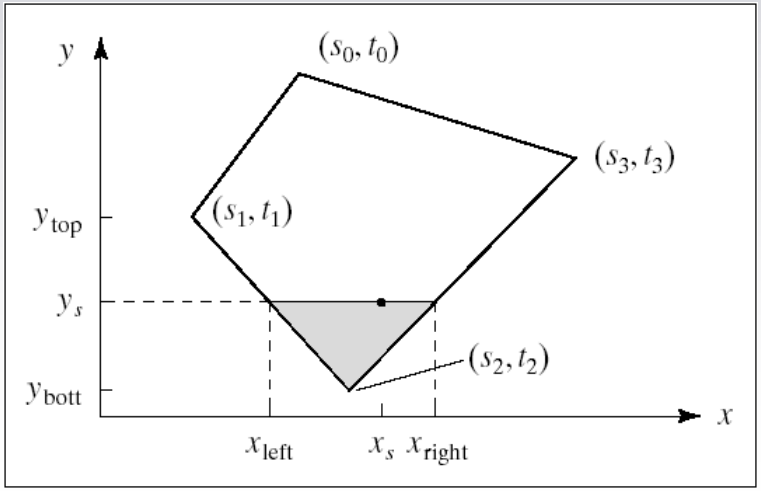
- Key idea: Render 360 degree view of environment **from center of object** with sphere or box as intermediate surface
- Intersection of eye reflection vector with intermediate surface provides texture coordinates for reflection/environment mapping



courtesy of R. Wolfe

Texture Rasterization

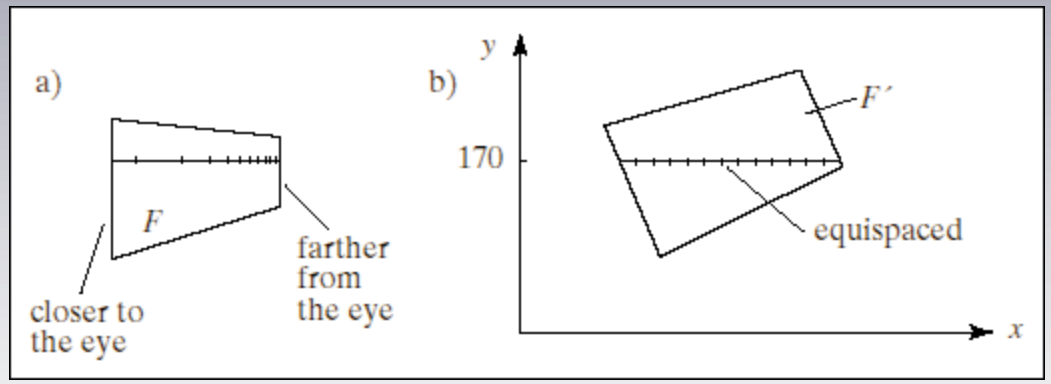
- Okay…we've got texture coordinates for the polygon vertices. What are (s, t) for the pixels inside the polygon?
- Use Gouraud-style linear interpolation of texture coordinates, right?
  - First along polygon edges between vertices
  - Then along scanlines between left and right sides



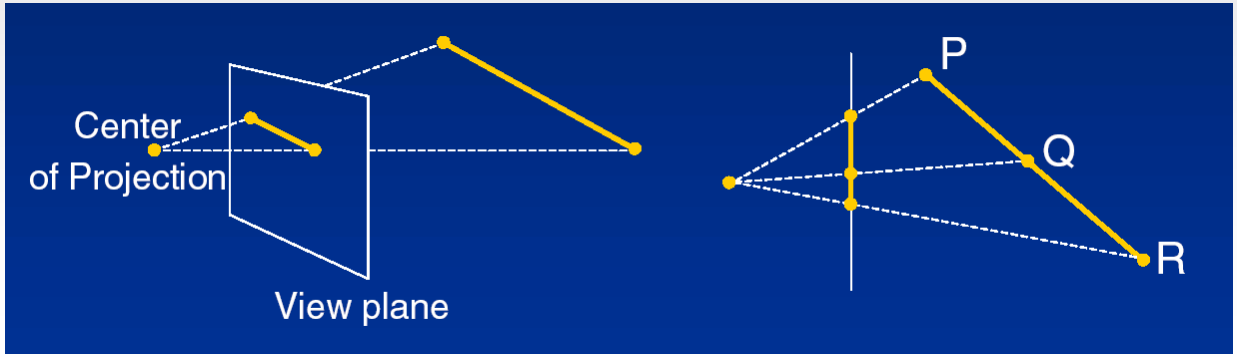
from Hill

Why not?

- Equally-spaced pixels do **not** project to equally-spaced texels under perspective projection
  - No problem with 2-D affine transforms (rotation, scaling, shear, etc.)
  - But different depths change things

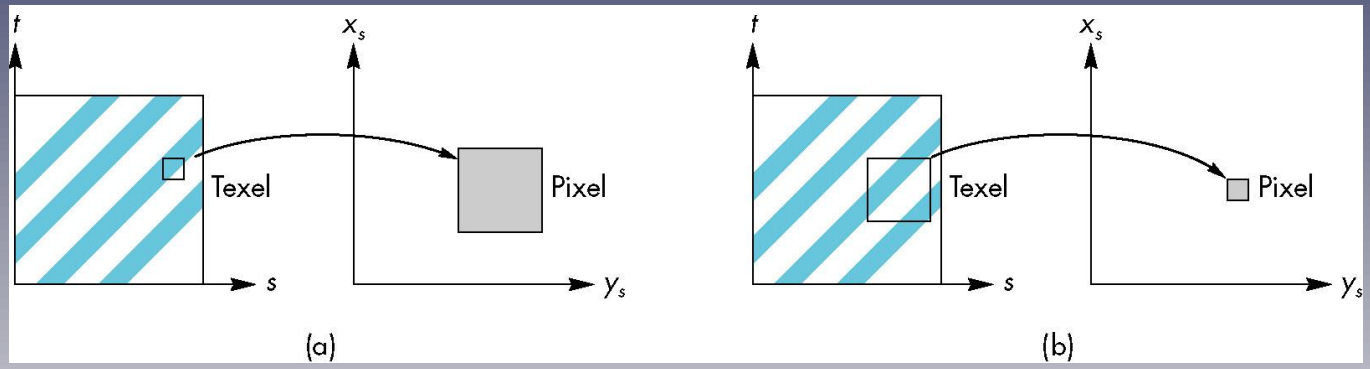


from Hill



courtesy of H. Pfister

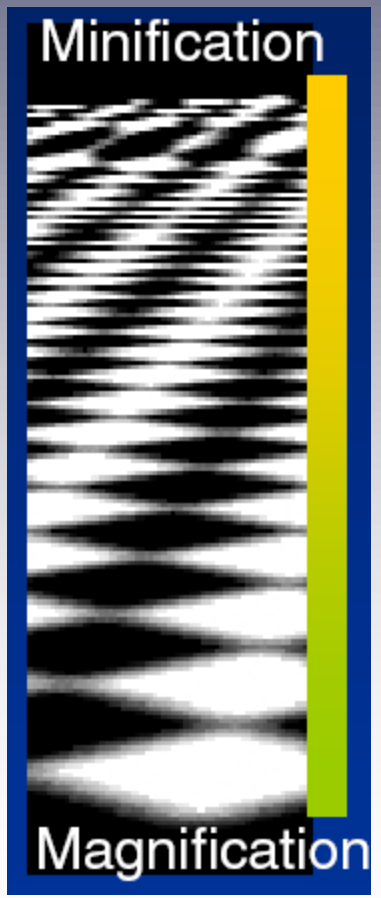
Magnification and minification



from Angel

Magnification

Minification

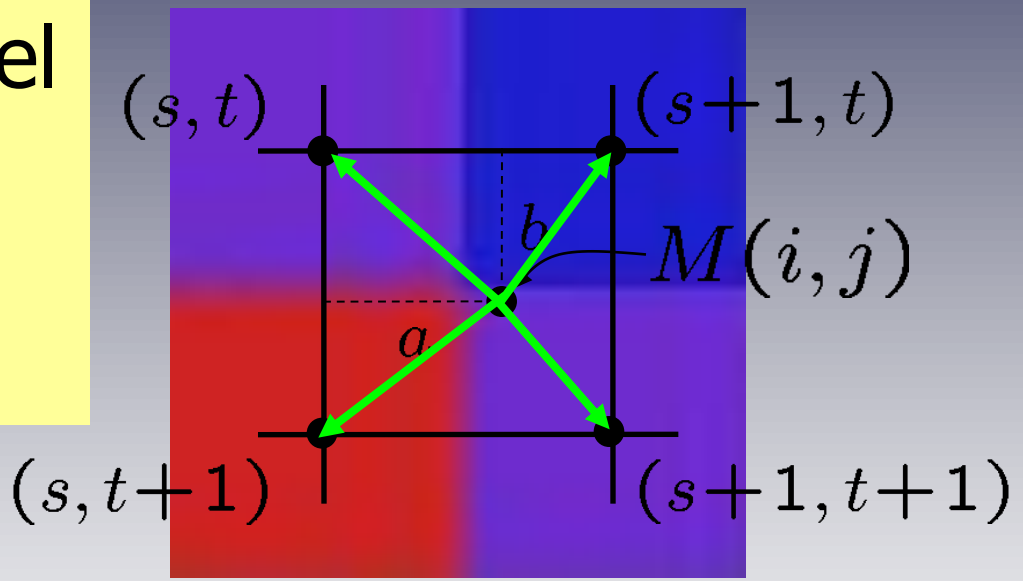


courtesy of H. Pfister

- **Magnification**: Single screen pixel maps to area less than or equal to one texel
- **Minification**: Single screen pixel area maps to area greater than one texel
  - If texel area covered is much greater than 4, even bilinear filtering isn't so great

Bilinear Interpolation (BLI)

- Idea: Blend four texel values surrounding source, weighted by nearness



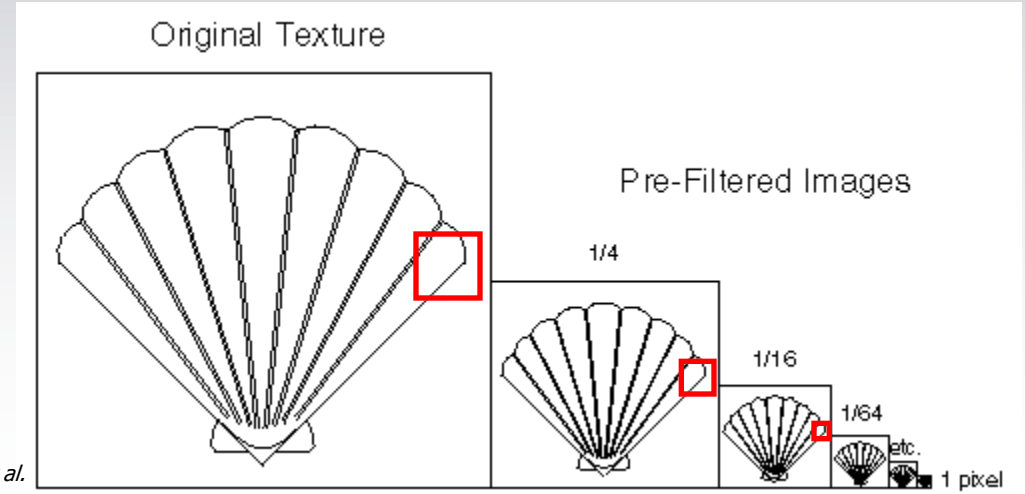
$$\mathbf{I}(i,j) = (1-b, b) \begin{bmatrix} \mathbf{I}_{tex}(s,t) & \mathbf{I}_{tex}(s+1,t) \\ \mathbf{I}_{tex}(s,t+1) & \mathbf{I}_{tex}(s+1,t+1) \end{bmatrix} \begin{pmatrix} 1-a \\ a \end{pmatrix}$$

Vertical blend

Horizontal blend

Mipmaps

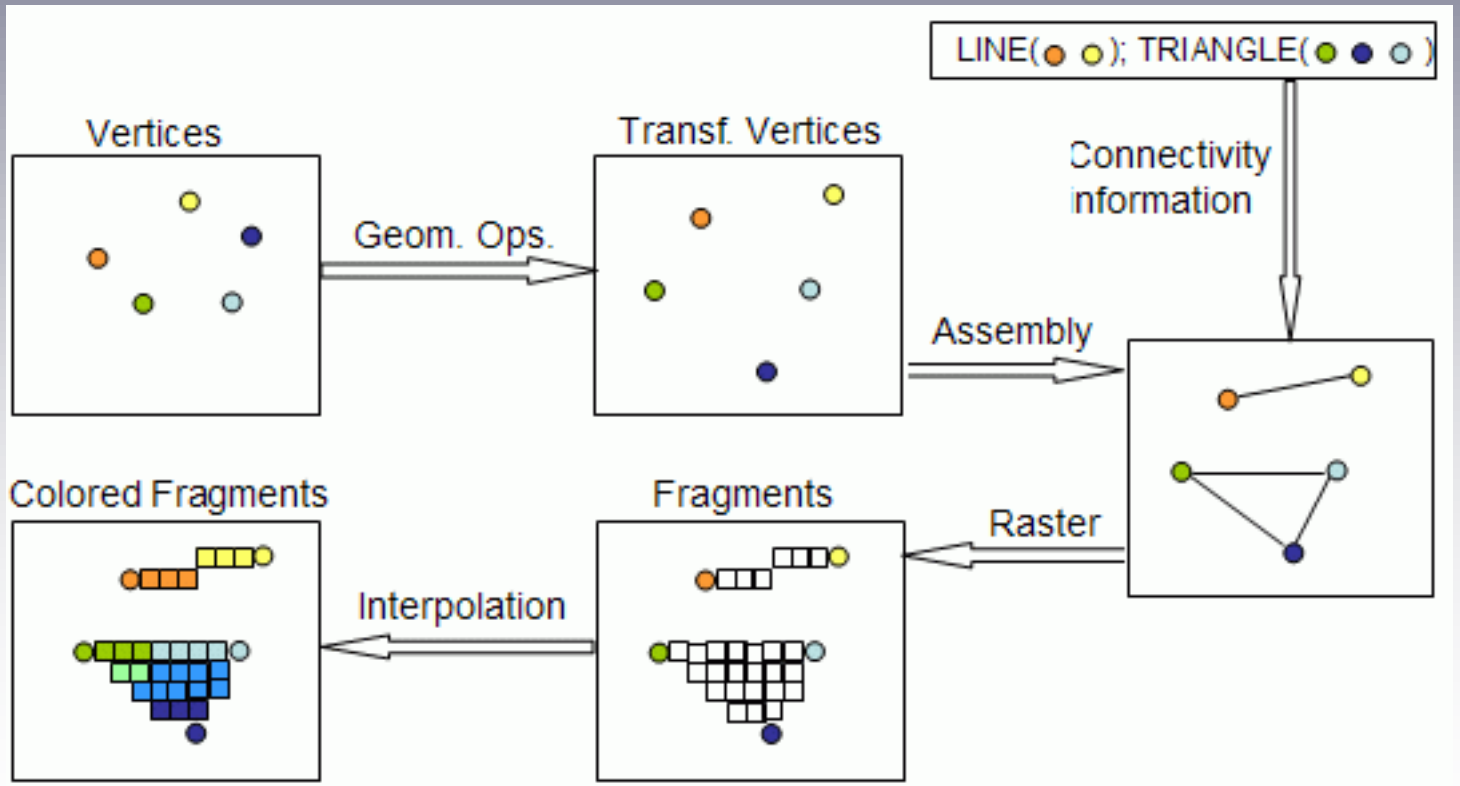
- Filtering for minification is expensive, and different areas must be averaged depending on the amount of minification
- Idea:
  - Prefilter entire texture image at different resolutions
  - For each screen pixel, pick texture in mipmap at **level of detail (LOD)** that minimizes minification (i.e., pre-image area closest to 1)
  - Do nearest or linear filtering in appropriate LOD texture image



from Woo, *et al.*

Pipeline Visual Summary

- From lighthouse3d



Vertex Shaders

- Operates on single vertex
- Input includes position, color, normals, etc.
- Must take care of
  - Modelview and perspective transformations
  - Normal transformations
  - Texture coordinate generation & transformations
  - Lighting per vertex
  - Etc.

Fragment Shaders

- Operates on single fragment, executed in parallel
- Alpha blending, depth test still in fixed pipeline
- Can be used to
  - Compute colors
  - Apply textures
  - Compute fog
  - Per-pixel lighting (e.g., Phong shading)

GLSL Syntax Overview

- GLSL is like C **without**
  - Pointers, objects (but structs are still there)
  - Recursion
  - Gotos, switches
  - Dynamic memory allocation
- GLSL is like C with
  - Built-in vector, matrix and sampler types
  - Constructors
  - A great math library
  - Input and output qualifiers

GLSL Syntax: Types

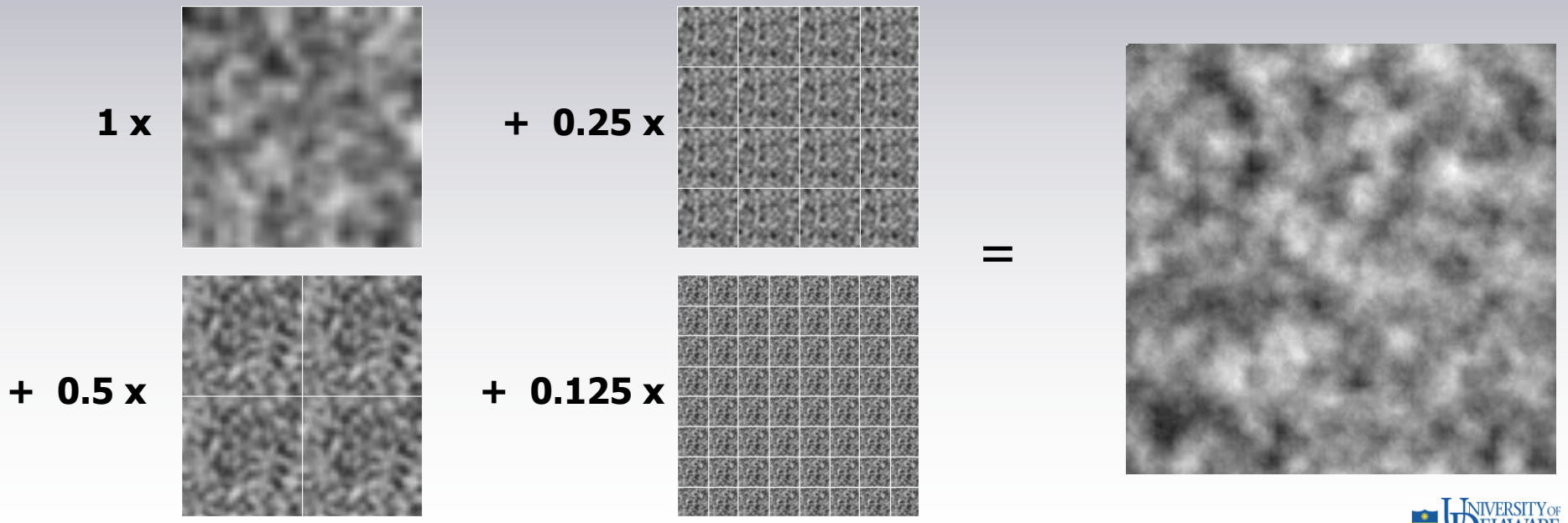
- Scalar types: `float`, `int`, `uint`, `bool`
  - Can do casts: `int i = int(0.0);`
- Vectors are first-class types:
  - `vec2`, `vec3`, `vec4`
  - `ivec`, `bvec`: Integer, boolean vectors
- Matrices
  - mat2, mat3, mat4 (2x2, 3x3, 4x4)
  - Can also do m x n; stored in column-major format
- Samplers
  - sampler1D, sampler2D, sampler3D

Noise as a Texture Generator

- Easiest texture to make: Random values for texels
  - `noise(x, y) = random()`
- If `random()` has limited range (e.g., [0, 1]), can control maximum value via **amplitude**
  - `a * noise(x, y)`
- But the results usually aren't very exciting visually

Perlin Noise for Turbulence

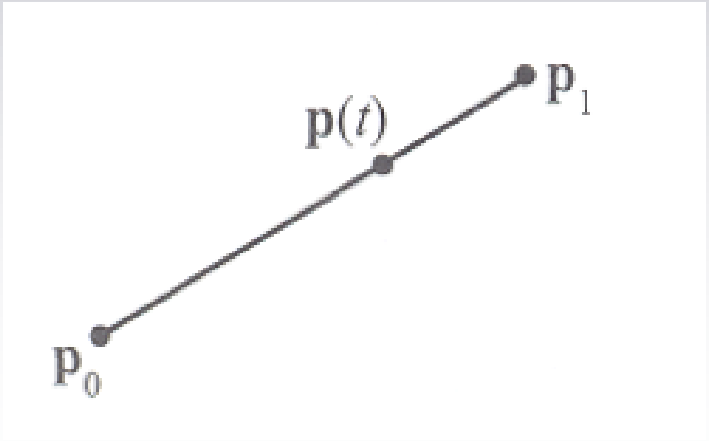
- Fractal noise: Many frequencies present, looks more natural
- Can get this by **summing** noise at different magnifications
- $\mathbf{turb(x, y, z) = \Sigma_i\ a_i * noise_i(x, y, z)}$
- Typical (but totally adjustable) parameters:
  - Magnification doubles at each level (**octave**)
  - Amplitude drops by half



Parametric Lines

- Parametric definition of a line segment:

$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0), \text{ where } t \in [0, 1]$$
$$= \mathbf{p}_0 - t\,\mathbf{p}_0 + t\,\mathbf{p}_1$$
$$= (1 - t)\mathbf{p}_0 + t\,\mathbf{p}_1$$



like a "blend" of the two endpoints

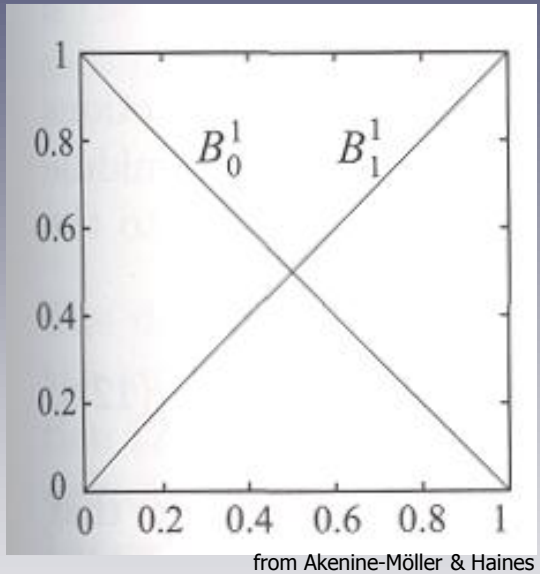
Linear Interpolation as Blending

- Consider each point on the line segment as a sum of **control points** $\mathbf{p}_i$ weighted by **blending functions** B_i :

degree n = 1 for linear blending

$$\mathbf{p}(t) = \sum_{i=0}^{n} B_i^n(t)\mathbf{p}_i$$

- Here we have $B_0 = 1 - t$ and $B_1 = t$

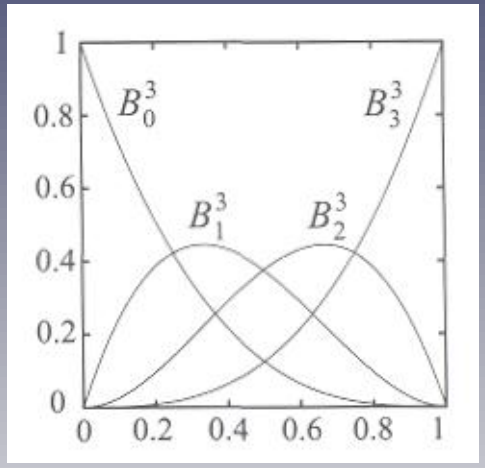


from Akenine-Möller & Haines

Blending functions for linear interpolation (2 control points)

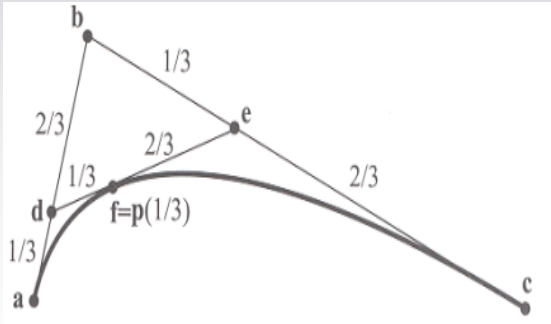
Bézier Curves

- Curve approximation through **recursive** application of linear interpolations
  - Linear: 2 control points, 2 linear Bernstein polynomials
  - Quadratic: 3 control points, 3 quadratic Bernstein polynomials
  - N control points = N - 1 degree curve
- Notes
  - Only endpoints are interpolated (i.e., on the curve)
  - Curve is tangent to linear segments at endpoints
  - Every control point affects every point on curve
    - Makes modeling harder



from Akenine-Möller & Haines

Cubic Bernstein polynomials for 4 control points

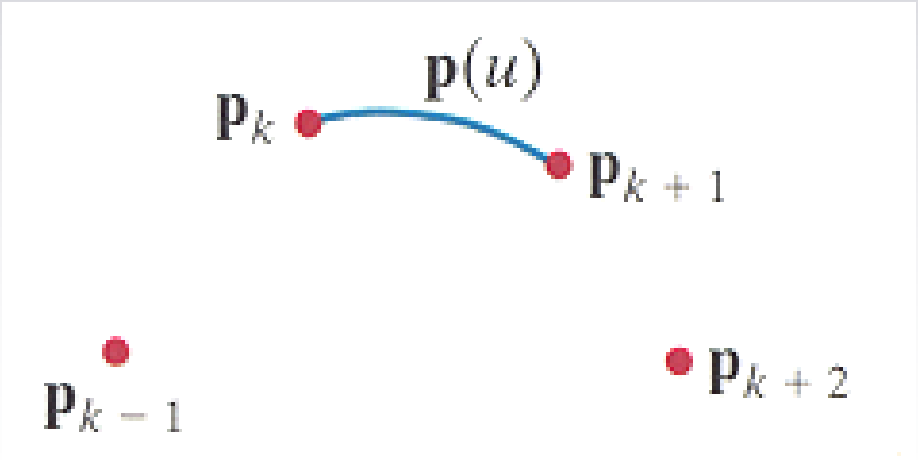


Interpolating Splines

- Idea: Use **key frames** to indicate a series of positions that must be "hit"
- For example:
  - Camera location
  - Path for character to follow
  - Animation of walking, gesturing, or facial expressions
    - Morphing
- Use splines for smooth interpolation
  - Must not be approximating!

Catmull-Rom spline

- Different from Bezier curves in that we can have arbitrary number of control points, but only 4 of them at a time influence each section of curve
  - And it's **interpolating** (goes through points) instead of **approximating** (goes "near" points)
- Four points define curve between 2nd and 3rd



Inferring the Coefficients

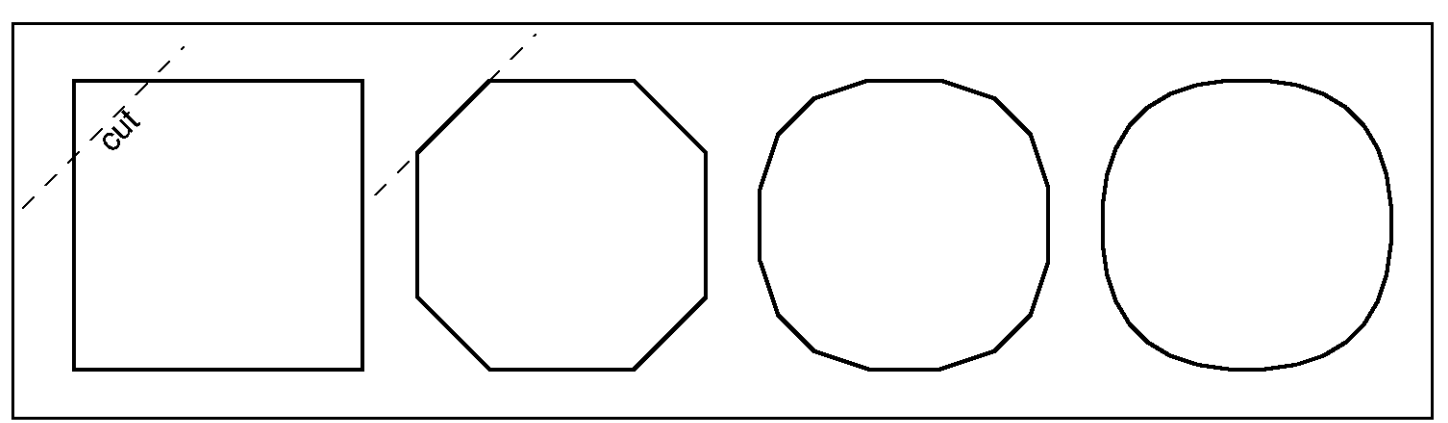
$$\mathbf{P}(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

- The control points are located at t = 0 and t = 1 on the curve segment, so we can relate them to the polynomial coefficients as follows:

$$\mathbf{P}(0) = a_0$$
$$\mathbf{P}(1) = a_0 + a_1 + a_2 + a_3$$
$$\mathbf{P}'(0) = a_1$$
$$\mathbf{P}'(1) = a_1 + 2a_2 + 3a_3$$

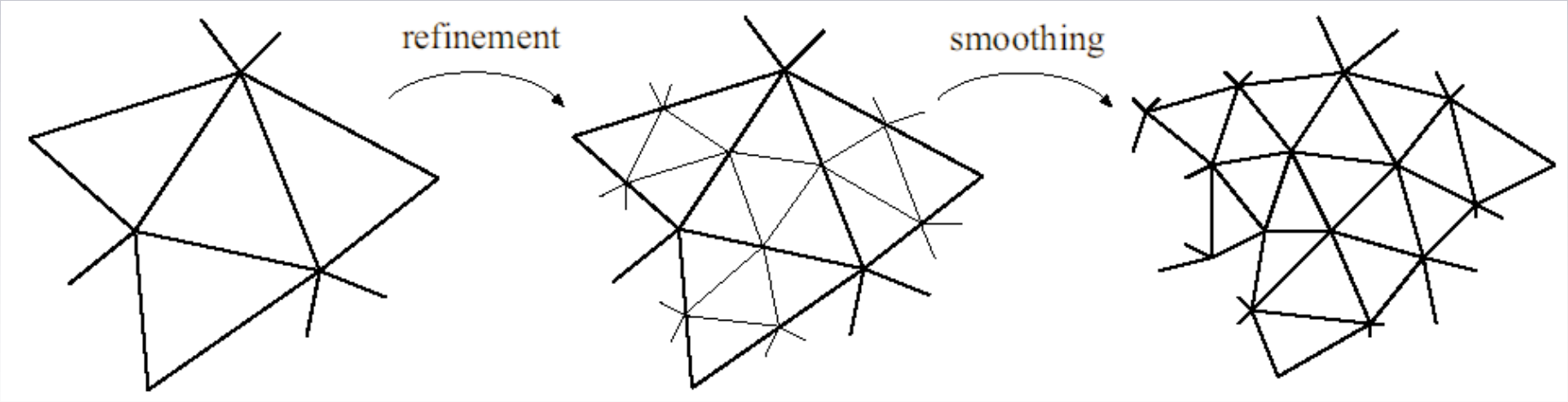
Curve Subdivision

- Goal: Algorithmically obtain smooth curves starting from small number of line segments
- One approach: **Corner-cutting** subdivision
  - Repeatedly chop off corners of polygon
  - Each line segment is replaced by two shorter segments
  - **Limit curve** is shape that would be reached after an infinite series of such subdivisions



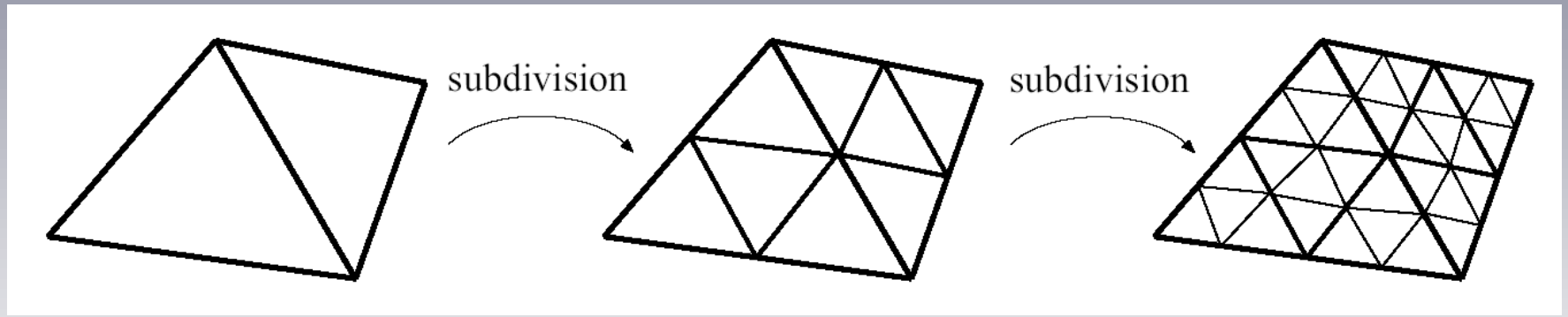
Surface Subdivision

- Analogous to curve subdivision:
  1. **Refine mesh**: Choose new vertices to make smaller polygons, update connectivity
  2. **Smooth mesh**: Move vertices to fit underlying object



Loop subdivision

- Smooths **triangle** mesh
- Subdivision replaces 1 triangle with 4



from Akenine-Möller & Haines

- Approximating scheme
  - Original vertices not guaranteed to be in subdivided mesh