CSC 4356
Interactive Computer Graphics
Final Review

Jinwei Ye

http://www.csc.lsu.edu/~jye/CSC4356/

Tue & Thu: 10:30 - 11:50am
218 Tureaud Hall

What We Have Learned

- 2D & 3D Transformations
- Rasterization
- Viewing Transformation
- Projection Transformation
- 3D Object Representation
- User Interaction
- Hidden Surface Removal
- Illumination Models
- Texture Mapping
- Ray Tracing
- Image-Based Rendering

What We Have Learned

- 2D & 3D Transformations
- Rasterization
- Viewing Transformation
- Projection Transformation
- 3D Object Representation
- User Interaction
- Hidden Surface Removal
- Illumination Models
- Texture Mapping
- Ray Tracing
- Image-Based Rendering

Illumination Model in OpenGL

- Final surface reflectance models as combination of **ambient**, **diffuse**, and **specular** components
  - Simplified empirical illumination model
  - Approximate global lighting effects

$$I_{total} = I_{ambient} + I_{diffuse} + I_{specular}$$

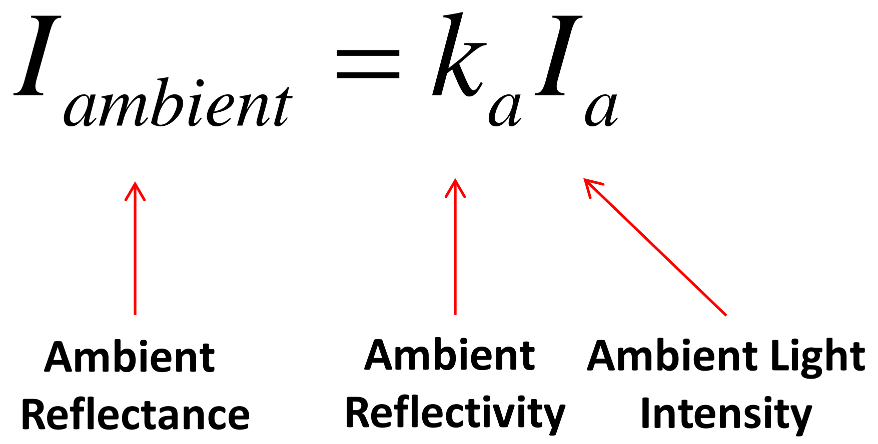
Ambient + Diffuse + Specular = Phong Reflection

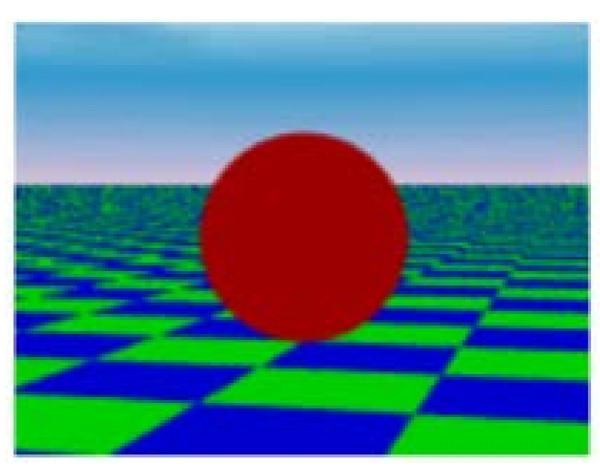
Ambient Light Source

- Even though an object in a scene is not directly lit it will still be visible. This is because light is reflected indirectly from nearby objects
- A simple hack that is commonly used to model this indirect illumination is to use of an ambient light source
- Ambient light source:
  - No spatial or directional characteristics
  - The amount of ambient light incident on each object is a constant for all surfaces in the scene (minimum illumination)
  - An ambient light can have a color

Ambient Reflectance

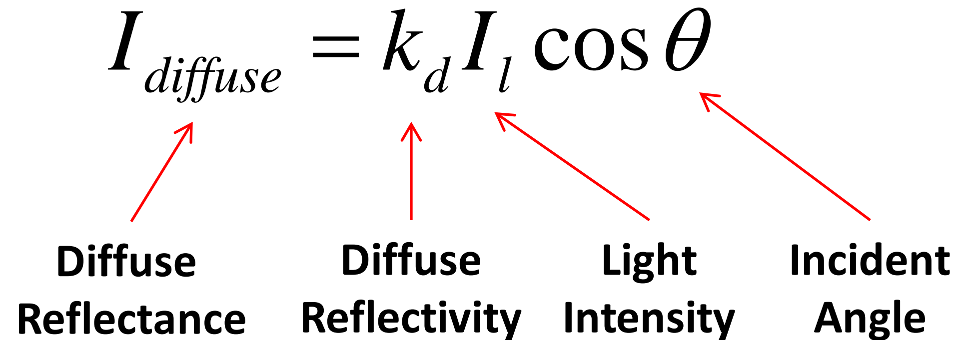
- The amount of ambient light that is reflected by an object is independent of the object's position or orientation
- Surface properties are used to determine how much ambient light is reflected





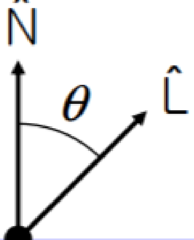
Computing Diffuse Reflection

- The angle between the surface normal and the incoming light ray is called the angle of incidence and we can express a intensity of the light in terms of this angle θ



- In practice, we can use dot product to compute $\cos\theta$
  - If both the surface normal and the lighting direction are normalized (unit length) then diffuse reflectance can be computed as

$$I_{diffuse} = k_d I_l (\hat{n} \cdot \hat{l})$$



Diffuse Light Examples

- Below are several examples of a spherical diffuse reflector with a varying lighting angles.
  - Why consider a spherical surface?
  - We need only consider angles from 0 to 90 degrees
  - Greater angles (where the dot product is negative) are blocked by the surface and the reflectance is zero

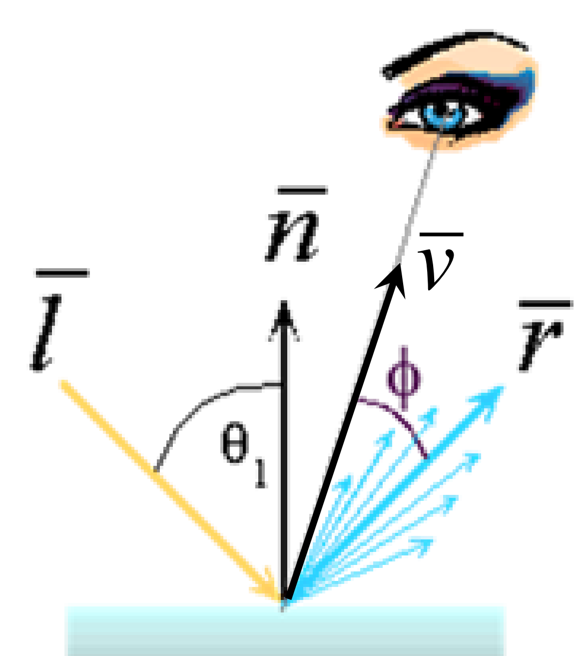
Phong Illumination

- Phong Illumination model approximates this fall off
  - This model has no physical basis
  - Yet it is one of the most commonly used illumination models in computer graphics

$$I_{specular} = k_s I_l (\cos\phi)^{n_{shiny}}$$

$$= k_s I_l (\hat{v} \cdot \hat{r})^{n_{shiny}}$$

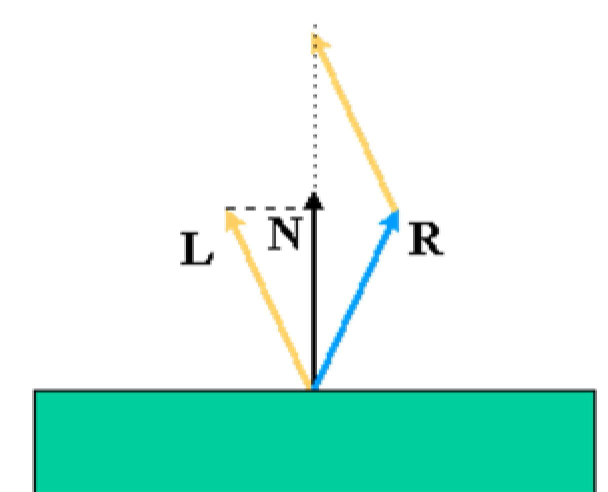
- $\hat{v}$ is the direction to the viewer
- The n_{shiny} controls how quickly the highlight falls off
  - The larger the exponent, the faster fall off



How to Compute Reflection Vector?

- The vector reflection vector R can be computed from the incoming light direction and the surface normal as shown below:

$$\hat{r} + \hat{l} = (2(\hat{n} \cdot \hat{l}))\hat{n} \quad \Rightarrow \quad \hat{r} = (2(\hat{n} \cdot \hat{l}))\hat{n} - \hat{l}$$

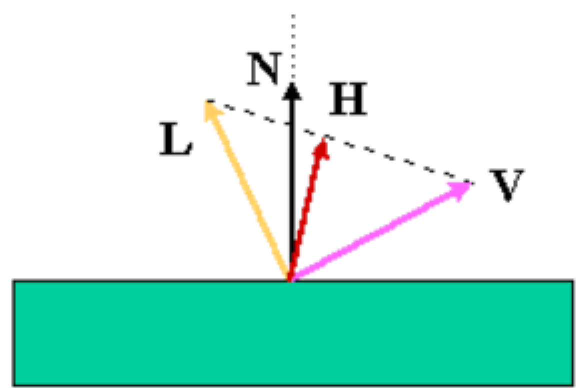


Blinn & Torrance Variation

- Jim Blinn introduced another approach for computing Phong-like illumination based on the work of Ken Torrance

$$I_{specular} = k_s I_l (\hat{n} \cdot \hat{H})^{n_{shiny}}$$

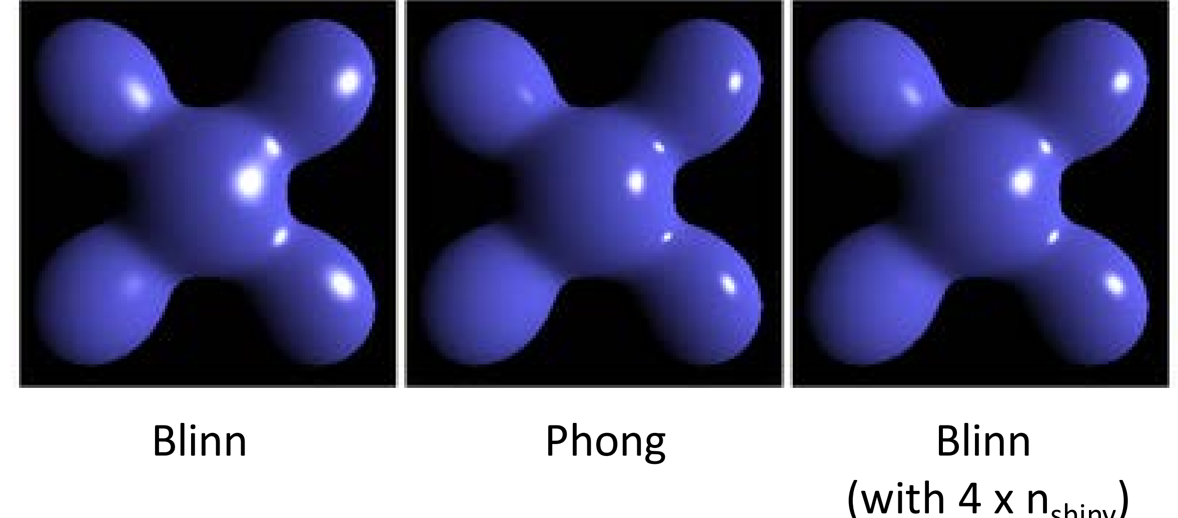
- Halfway vector H: a vector bisecting the incoming light direction and the viewing direction



$$\hat{H} = \frac{\hat{l} + \hat{v}}{|\hat{l} + \hat{v}|}$$

What is the difference?

- The angle between the halfway vector and the surface normal is likely to be smaller than the angle between *R* and *V* used in Phong's model
  - unless the surface is viewed from a very steep angle, the angle between H and N is likely to be larger
- We can set larger exponent (shininess) for Blinn



Blinn   Phong   Blinn (with 4 x n_{shiny})

Putting It All Together

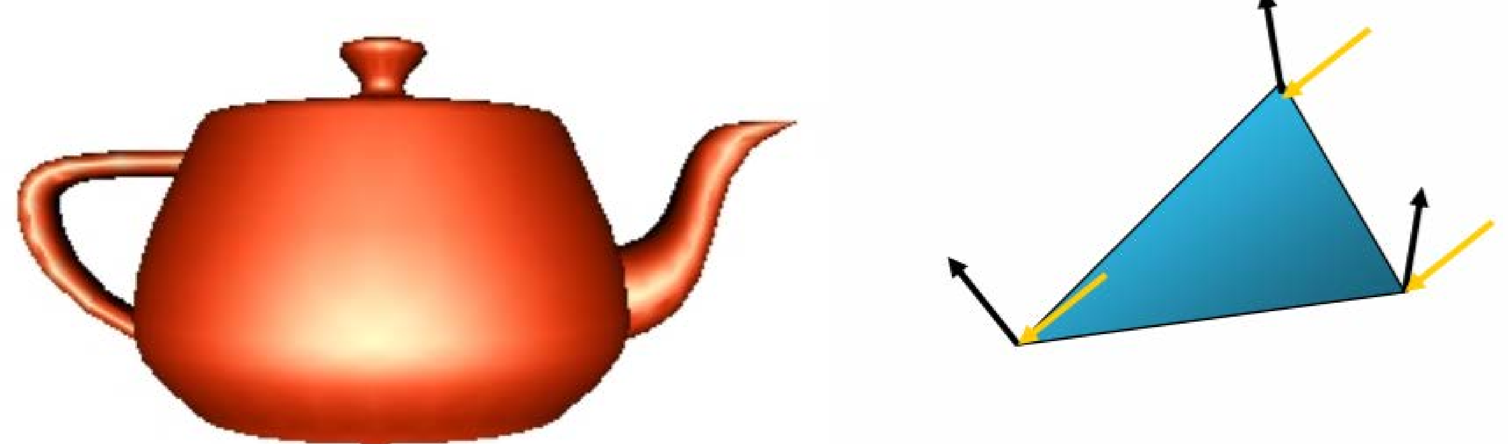
- Our final empirical illumination model is:

$$I_{total} = k_a I_a + \sum_{i=1}^{lights} I_i (k_d (\hat{n} \cdot \hat{l}) + k_s (\hat{v} \cdot \hat{r})^{n_{shiny}})$$

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

Gouraud Shading

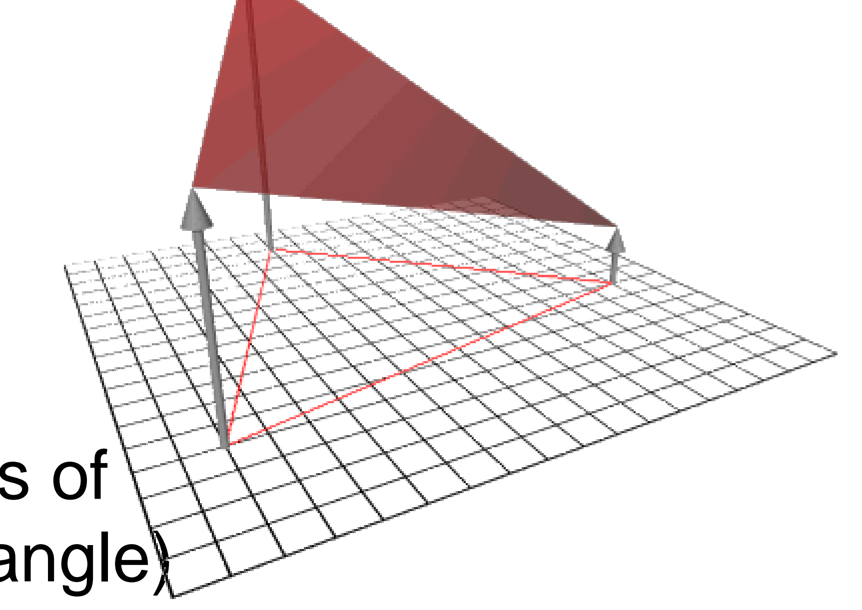
- The Gouraud Shading applies the illumination model on a subset of surface points and *interpolates the intensity* of the remaining points on the surface
  - In the case of a polygonal mesh the illumination model is applied at each vertex and the colors in the triangles interior are linearly interpolated from these vertex values
  - The linear interpolation can be accomplished using the plane equation method discussed in the lecture on rasterizing polygons
  - Notice that facet artifacts are still visible



Interpolating Color

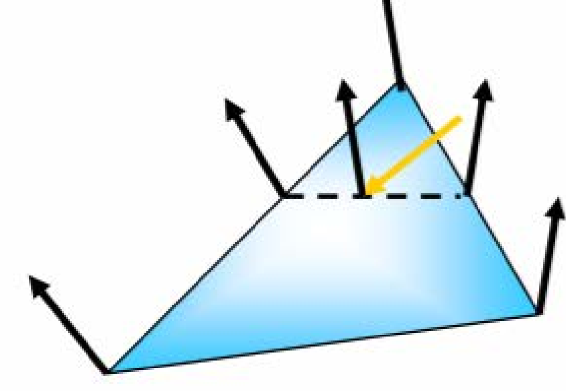
- Now we know how to draw a solid triangle (All vertices have the same color)
- What if they have different colors (or other parameters, e.g. depth)? How to interpolate?
- Idea: triangles are planar in any space:
  - This is the “redness” parameter space
  - Also need to do this for green and blue
  - Plane equation
    $z = A_r x + B_r y + C_r$

(here z stands for redness of a point (x,y) inside the triangle)

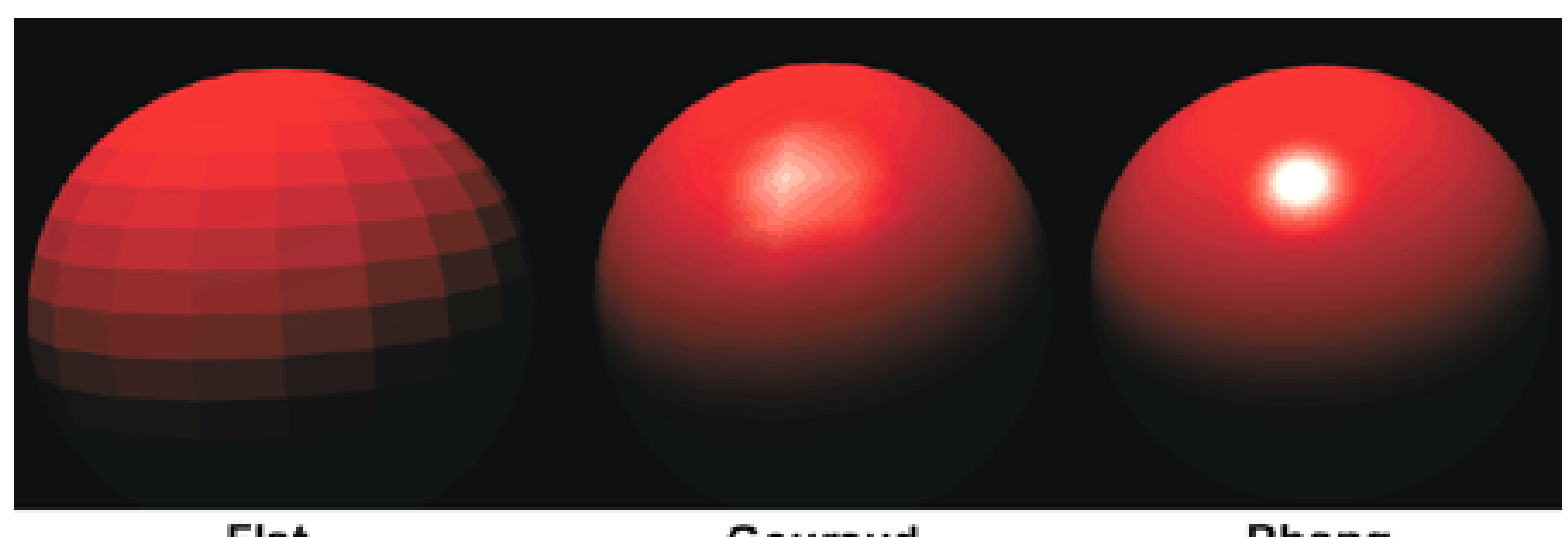


Phong Shading

- In Phong shading (not to be confused with Phong's illumination model), the *surface normal is linearly interpolated* across polygonal facets, and the illumination model is applied at every point
  - Better handling on specular high lights and usually results in a very smooth appearance
  - Slower than Gouraud shading
  - NOT built into OpenGL (OpenGL uses Gouraud)
  - Can be implemented on graphics card using fragment shader



Flat vs. Gouraud vs. Phong



Compute illumination model once on facet centroid

Compute illumination model on the vertices of a facet and then interpolate color for interior points

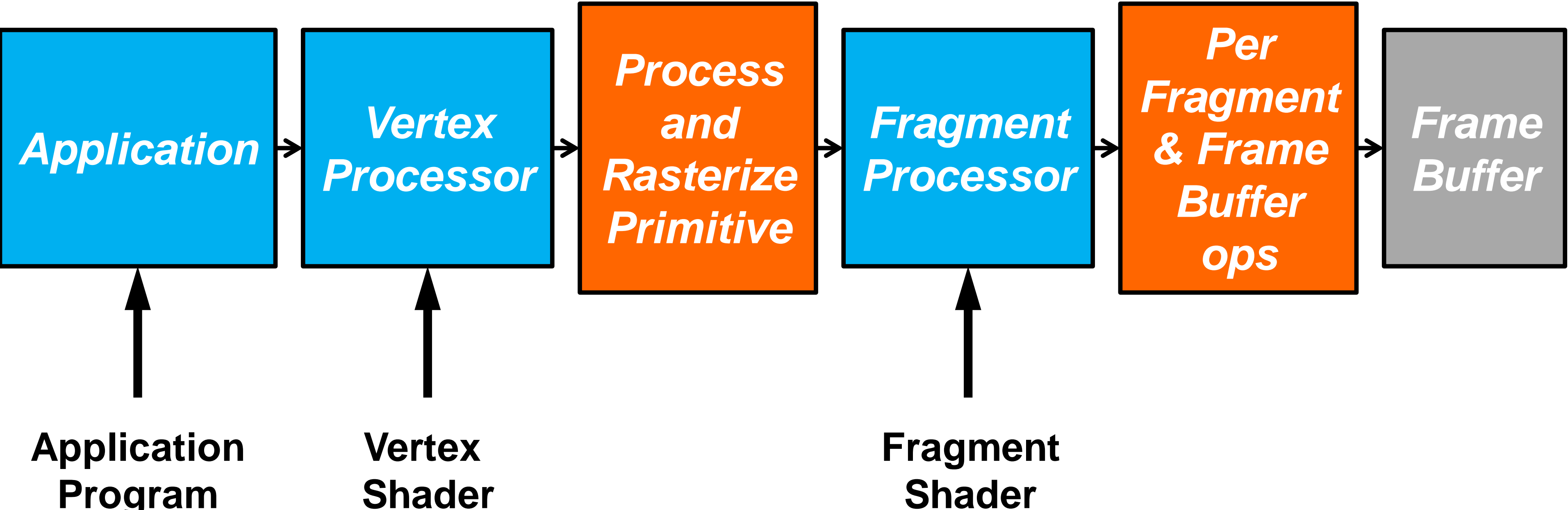
Interpolate normal for every point and then compute illumination model for each point on a facet

OpenGL Shading Language (GLSL)

- A C-like language and incorporated into OpenGL 2.0
- Used to write vertex shader and fragment shader
- No distinction in the syntax between a vertex shader and a fragment shader
- Platform independent compared to CG

GPU Programmability

- Programmable Processing units
  - Programmable per-Vertex Processors
  - Programmable per-Fragment Processors



Vertex Shader

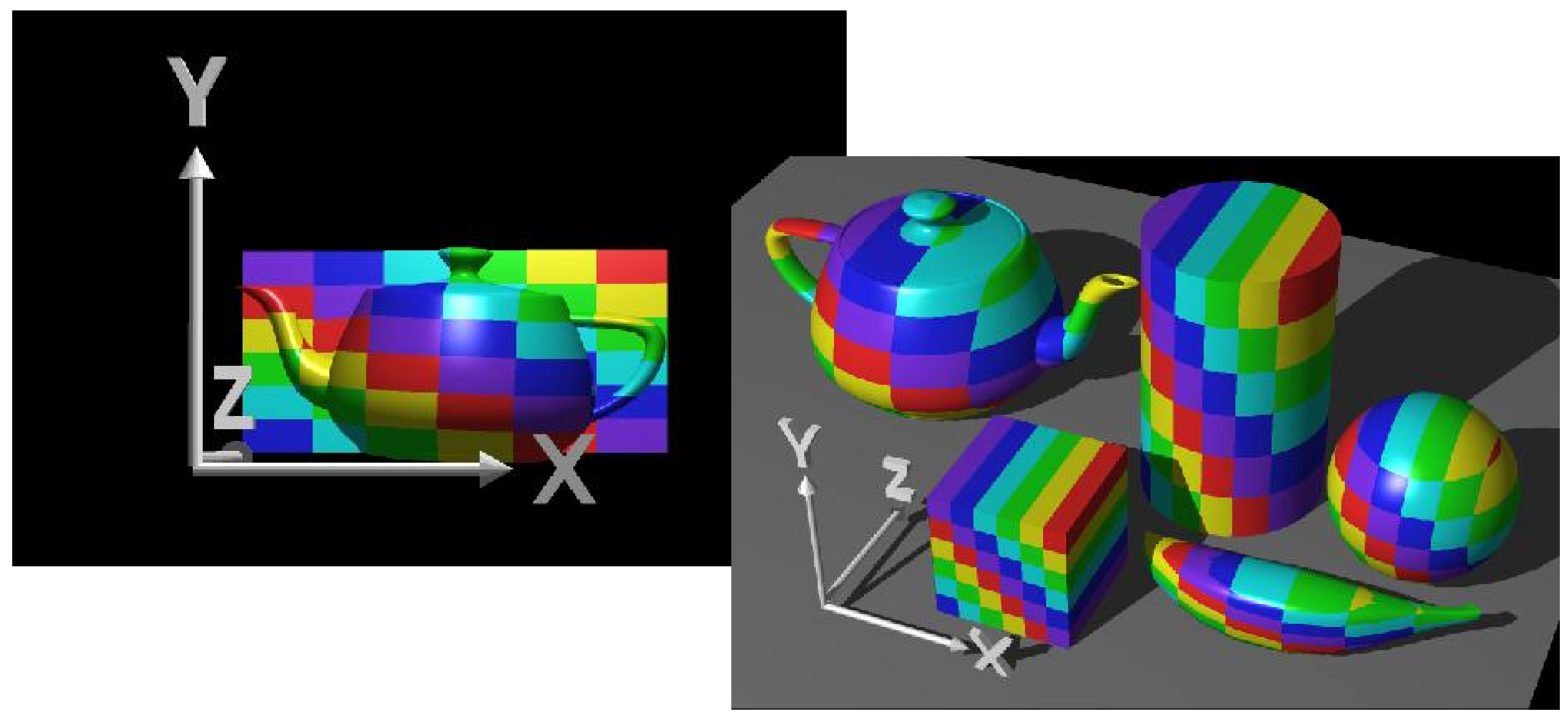
- A vertex shader is executed on each vertex triggered by glVertex*()
- Each vertex shader must output the information that the rasterizer needs
  - At a minimum: transforms the vertex position
- The program can access all OpenGL states
  - Current color, texture coordinates, material properties, transformation matrices, etc
- The application can also supply additional input variables to the vertex program

Fragment Shader

- The fragment shader is executed after rasterization and operate on each fragment
  - Per-pixel operations
- Vertex attributes (colors, positions, texture coordinates, etc.) are interpolated across a primitive automatically as the input to the fragment program
- Fragment shader can access OpenGL state, (interpolated) output from vertex program, and user defined variables

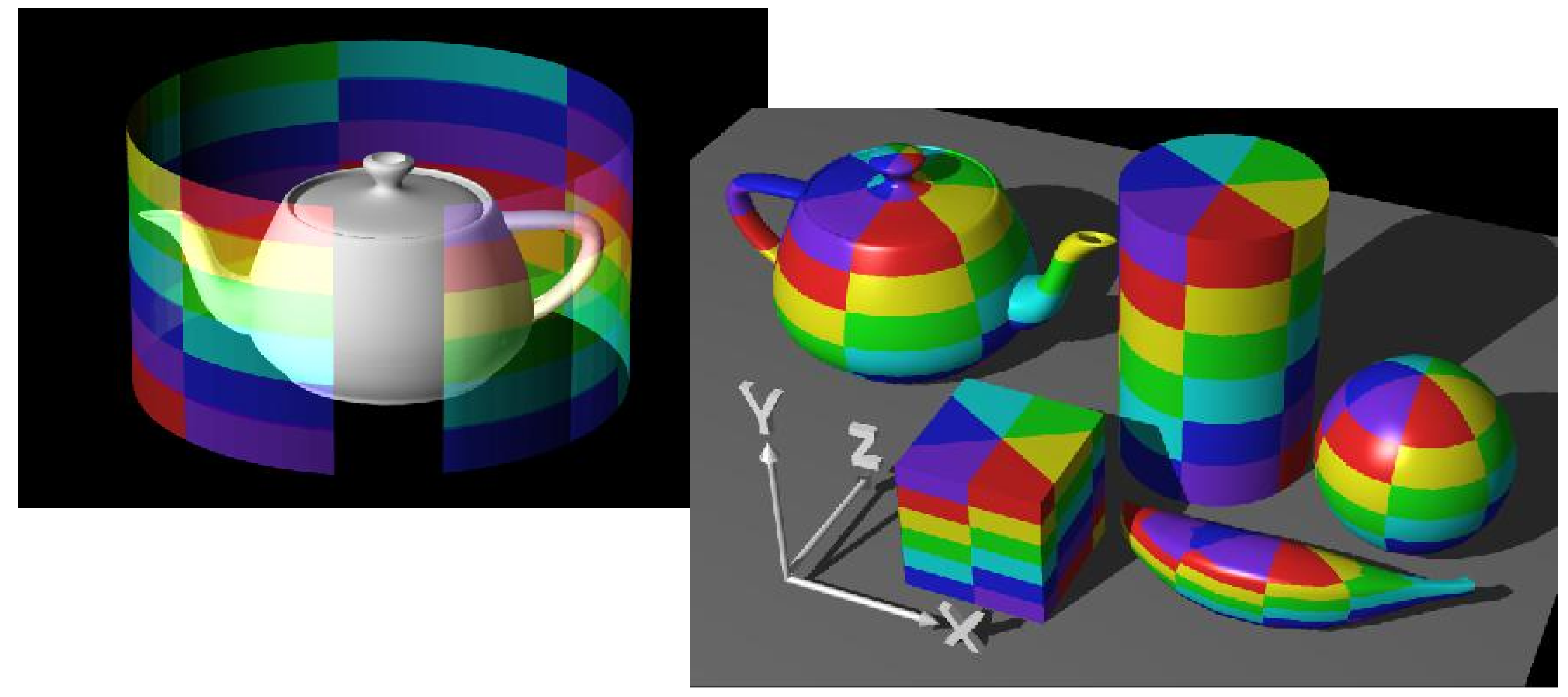
Planar Mapping

- Like projections, take vertex coordinate (x,y,z) and throw away one dimention
  - e.g., drop z such that texture coord (u,v) = (x/W,y/H)



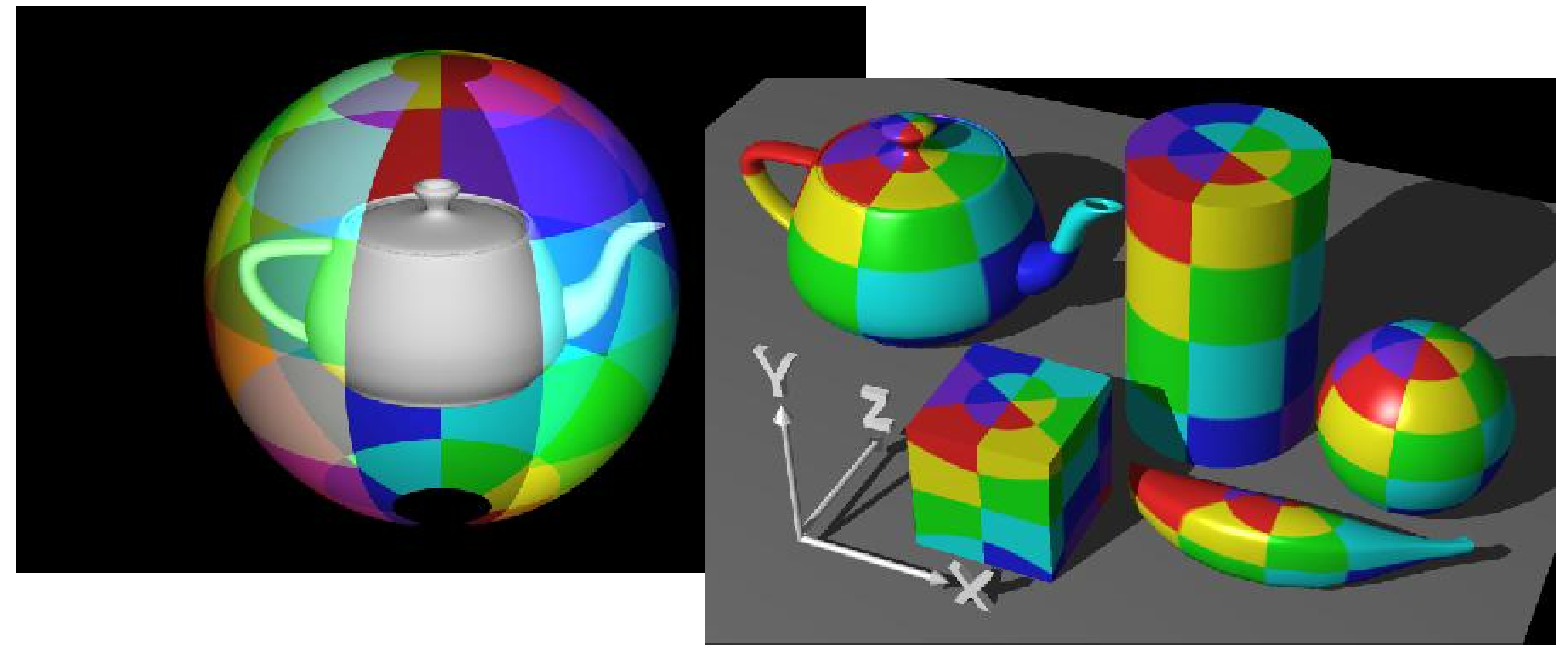
Cylindrical Mapping

- Cylinder: r, θ, z with $(u,v) = (\theta/(2\pi), z)$
  - Note seams when wrapping around ($\theta = 0$ or 2π)

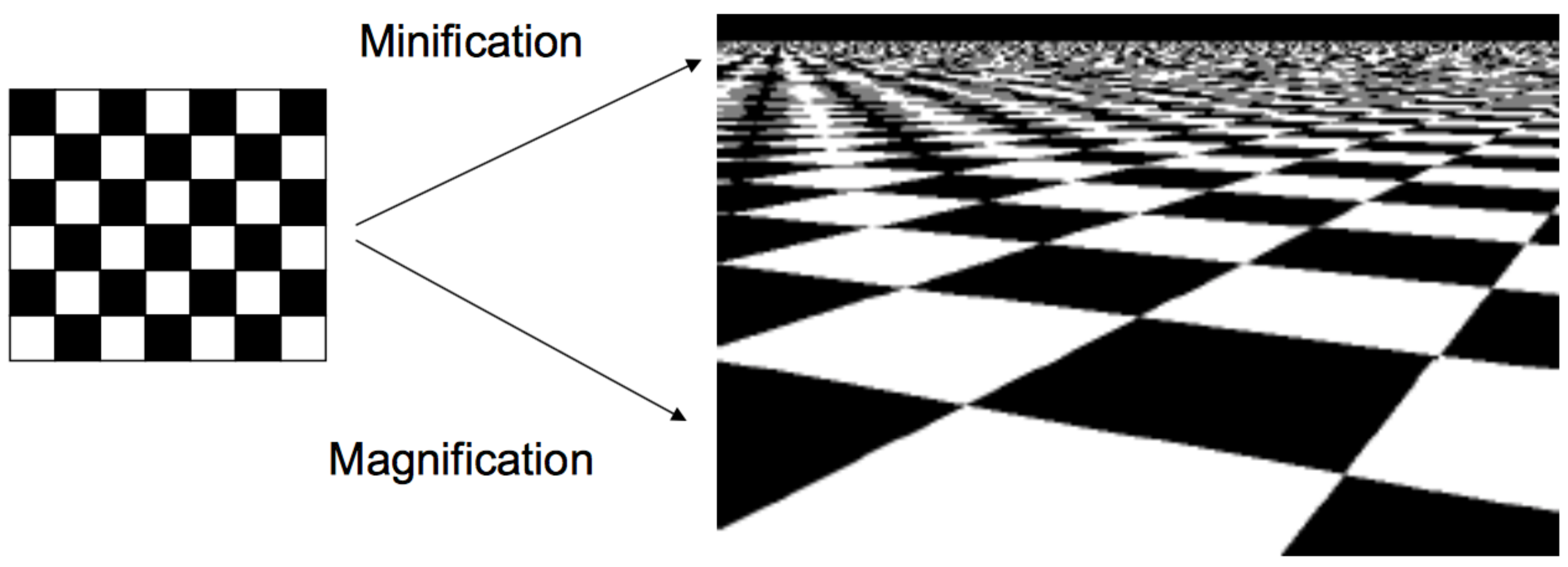


Spherical Mapping

- Convert to spherical coordinates: use latitude/longitude
  - Singularities at north and south poles



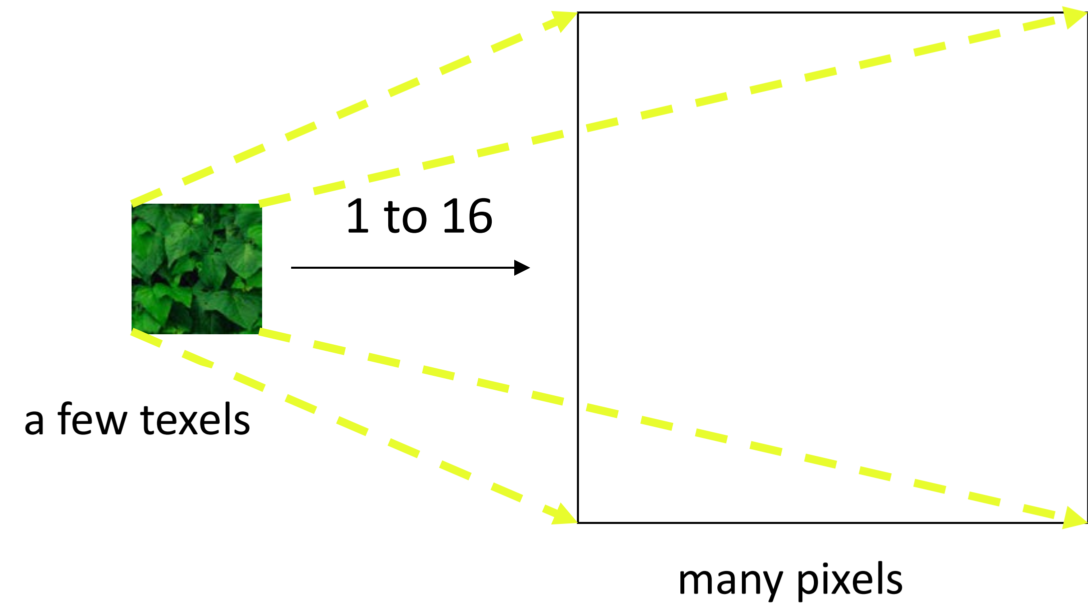
Minification & Magnification



(Note that minification is not handled very well here)

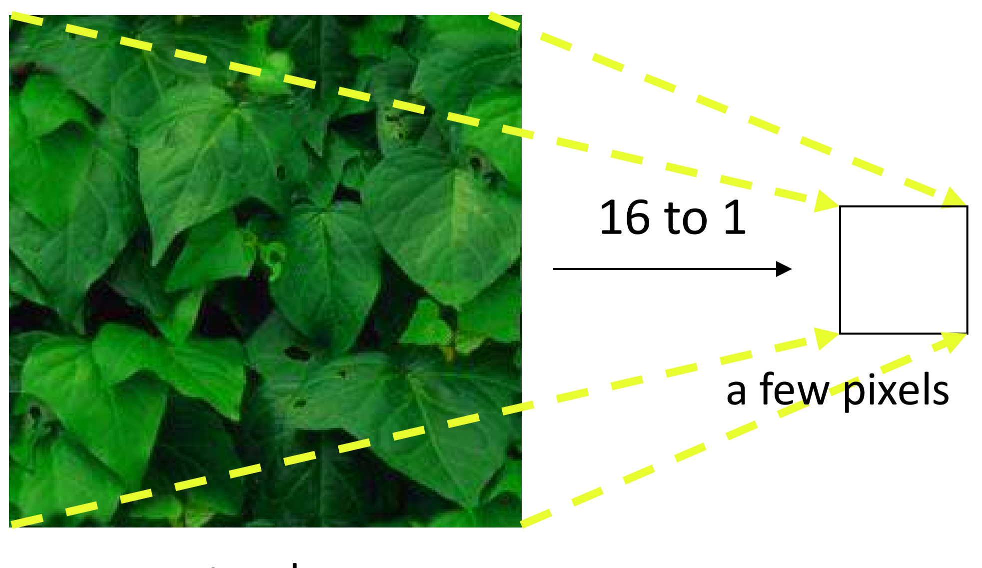
Magnification

- In magnification, one texel is mapped to many pixels



Minification

- In minification, many texels are mapped to one pixel



MIP Mapping

- MIP Mapping is one popular technique for precomputing and performing this pre-filtering. MIP is an acronym for the Latin phrase *multium in parvo*, which means “many in a small place”. The technique was first described by Lance Williams. The basic idea is to construct a pyramid of images that are pre-filtered and resampled at sampling frequencies that are a binary fractions (1/2, 1/4, 1/8, etc) of the original image's sampling.

- While rasterizing we compute the index of the decimated image that is sampled at a rate closest to the density of our desired sampling rate
  - Try to maintain pixel to texel ratio close to 1

Final sub-map is only one texel

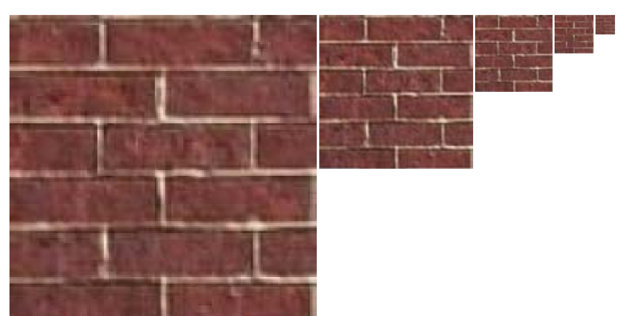
S

V

U

Each sub-map is 1/2 the size (1/4 the area) of the preceeding map

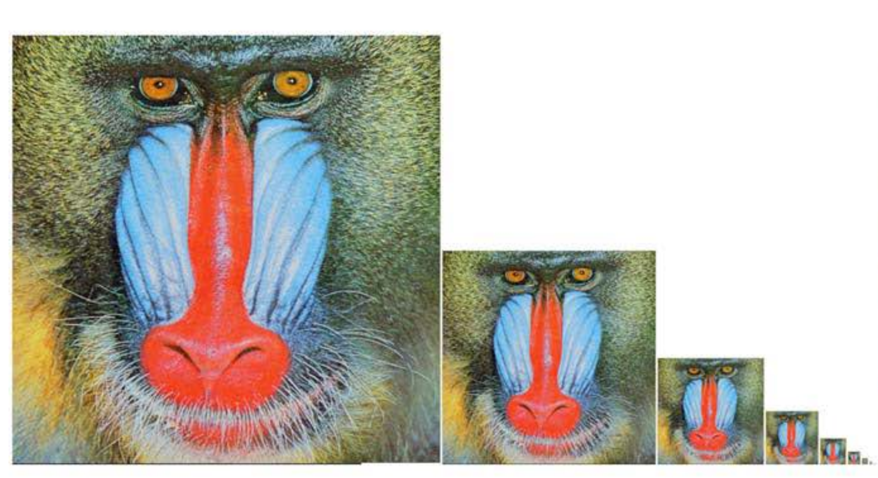
Main (full-resolution) texture map



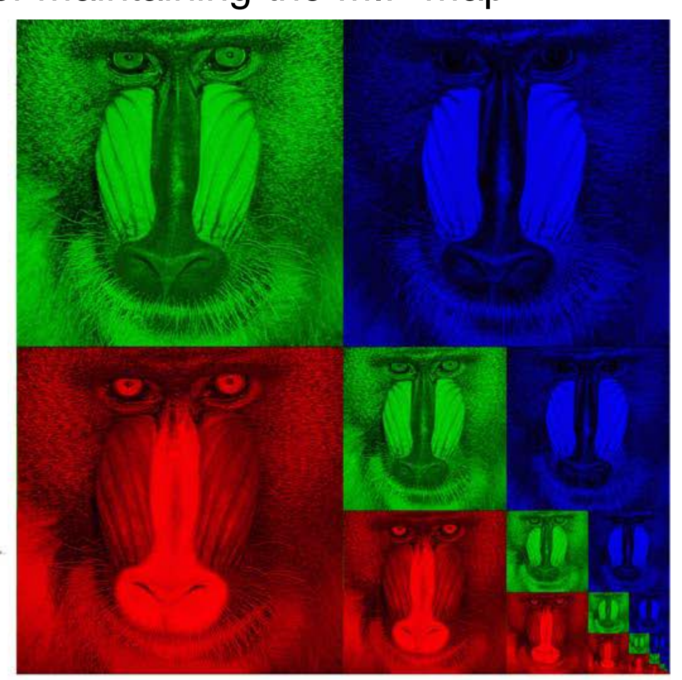
- Computing this series of filtered images requires only a small fraction of additional storage over the original texture (How small of a fraction?)

Storing MIP Maps

- One convenient method of storing a MIP map is shown on the right image
  - It also nicely illustrates the 1/3 overhead of maintaining the MIP map



10-level mip map



Memory format of a mip map

$$mip\ map\ size = \sum_{i=0}^{\infty} \left(\frac{1}{4}\right)^i = \frac{1}{1 - \frac{1}{4}} = \frac{4}{3}$$

Finding MIP Level

- Idea: Use the projection of a pixel in screen into texture space to figure out which level to use

$$u^*(x, y) = u / w = A_u x + B_u y + C_u$$

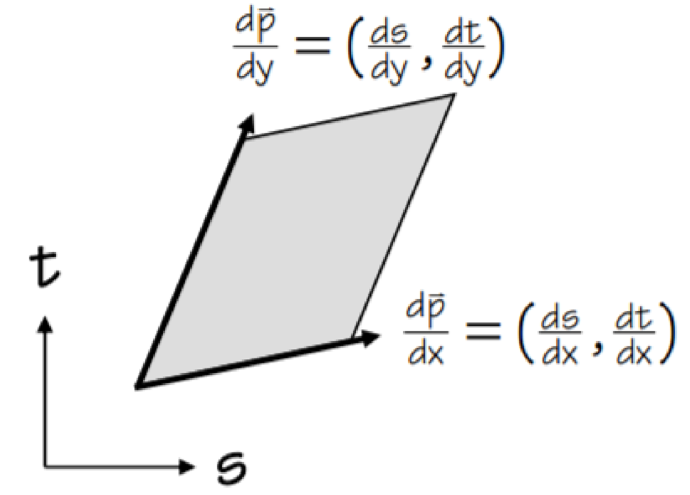
$$o^*(x, y) = 1 / w = A_o x + B_o y + C_o$$

$$s = u \cdot textureWidth$$

Applying chain rule: $\frac{ds}{dx} = \frac{ds}{du}\frac{du}{dx}$

$$\frac{ds}{du} = textureWidth$$

$$\frac{du}{dx} = \frac{d(u^*(x, y) / o^*(x, y))}{dx} = \frac{A_u o^*(x, y) - A_o u^*(x, y)}{o^*(x, y)^2}$$



Other derivatives can be computed in the same way.

Finding MIP Level

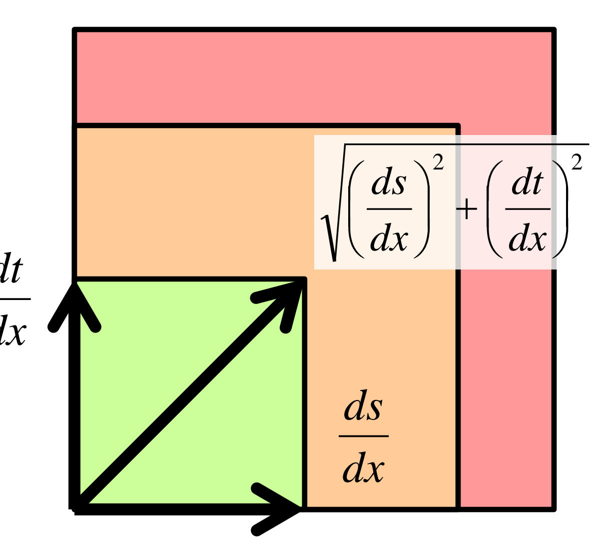
- Use the lengths of the projected pixel in texture space to define a measure of mismatch between sampling densities:

$$m = \max(\|\frac{d\vec{p}}{dx}\|, \|\frac{d\vec{p}}{dy}\|) = \max(\sqrt{(\frac{ds}{dx})^2 + (\frac{dt}{dx})^2}, \sqrt{(\frac{ds}{dy})^2 + (\frac{dt}{dy})^2})$$

$$\approx \max(\max(\frac{ds}{dx}, \frac{dt}{dx}), \max(\frac{ds}{dy}, \frac{dt}{dy}))$$

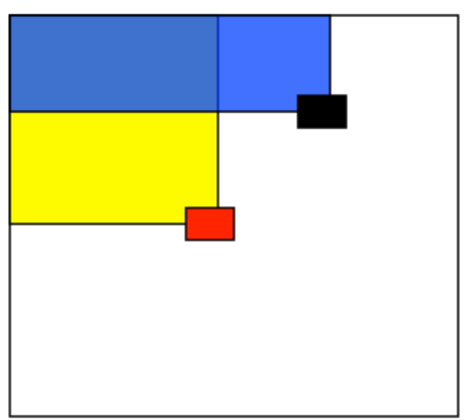
- Now choose the appropriate level:

$$level = \lfloor \log_2(m) \rfloor$$



Summed-Area Tables

- Summed-area tables perform anisotropic filtering
  - It can be used to compute the average color for any arbitrary rectangular region in the texture space at a constant speed
- Summed-area table is a two dimensional array that has the same size as the texture



Each entry stores the sum of all the texel colors above and to the left

Summed-Area Tables

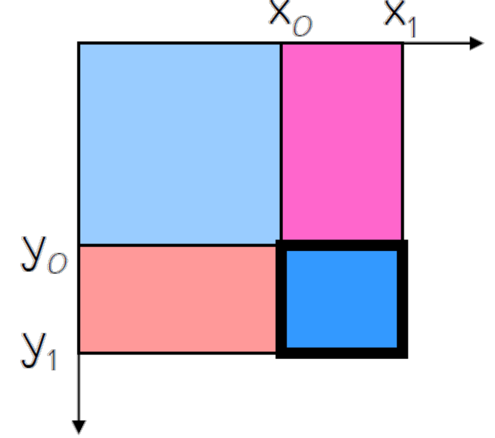
- Each entry in the summed area table is the sum of all entries above and to the left:

1	6	8	3
0	0	3	7
4	7	8	8
5	0	9	9

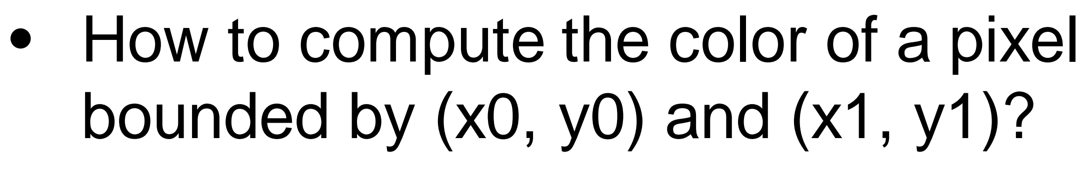
Original Texture

1	7	15	18
1	7	18	28
5	18	37	55
10	23	51	78

Summed-Area Table

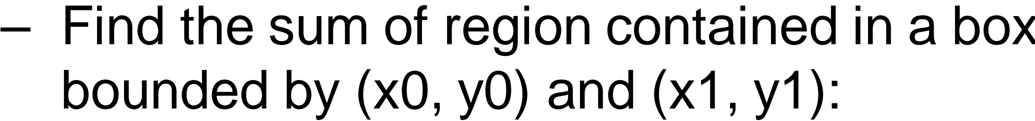


- How to compute the color of a pixel bounded by (x0, y0) and (x1, y1)?



  - Find the sum of region contained in a box bounded by (x0, y0) and (x1, y1):

$$T(x_1, y_1) - T(x_0, y_1) - T(x_1, y_0) + T(x_0, y_0)$$



  - Divide out area

$$(y_1 - y_0)(x_1 - x_0)$$

Shadow Map

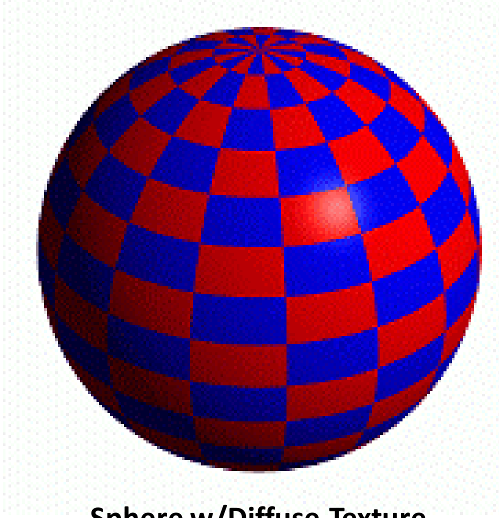
- Render an image from the light's point of view
  - Camera look-from point is the light position
  - Aim camera to look at objects in scene
  - Render only the z-buffer depth values
    - Don't need colors
    - Don't need to compute lighting or shading
      - (unless a procedural shader would make an object transparent)
- Store result in a *shadow map (*a.k.a. depth map*)*
  - Store the depth values (z-buffer)
  - Also store the (inverse) camera & projection transform
- Remember, z-buffer pixel holds depth of closest object to the camera
  - A shadow map pixel contains the distance of the closest object to the light (because camera is in the position of light)

Environment Mapping Steps

- Create a 2D environment map
- For each pixel on a reflective object, compute the normal
- Compute the reflection vector based on the eye position and surface normal
- Use the reflection vector to compute an index into the environment texture
- Use the corresponding texel to color the pixel

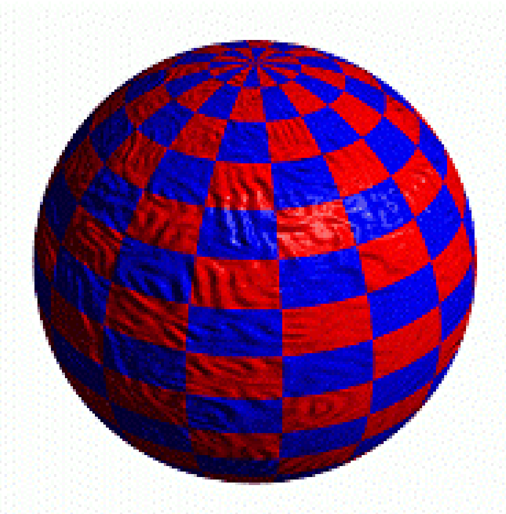
Bump Mapping

- Textures can be used to alter the surface normals of an object. This does not actual shape of the surface -- we are only shading it as if it were a different shape! This technique is called ***bump mapping***.
- The texture map is treated as a single-valued height function. The value of the function is not actually used, just its partial derivatives. The partial derivatives tell how to alter the true surface normal at each point on the surface to make the object appear as if it were deformed by the height function.



Sphere w/Diffuse Texture

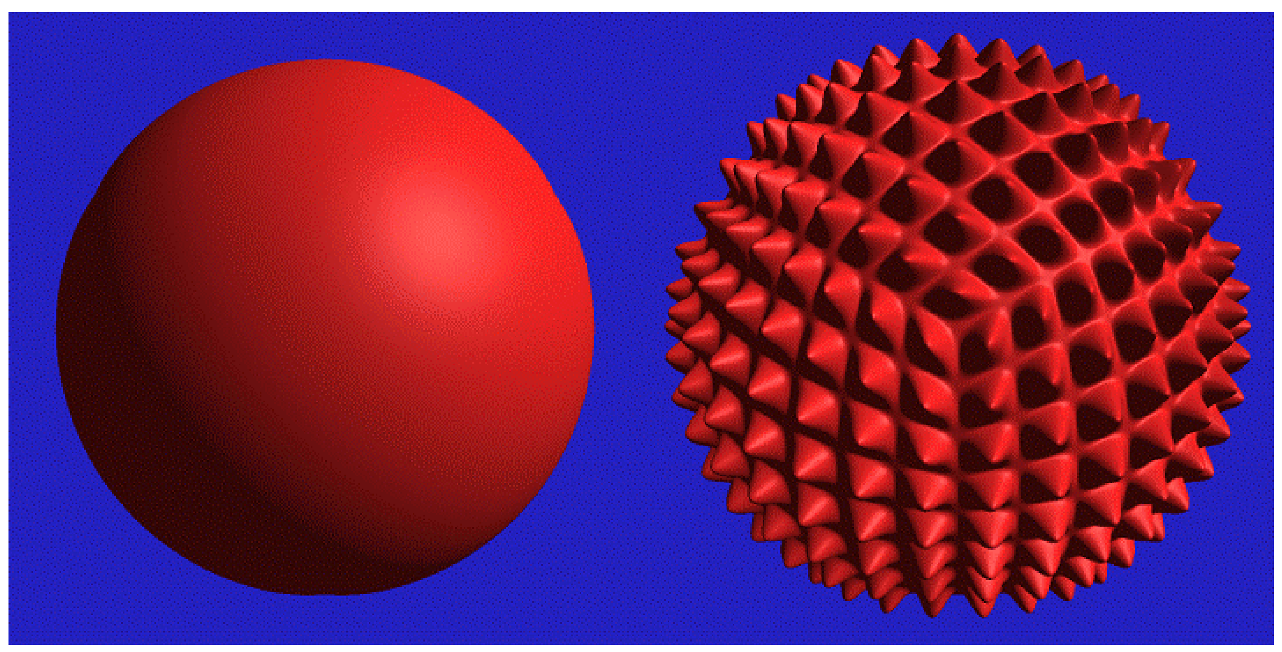
Swirly Bump Map



Sphere w/Diffuse Texture & Bump Map

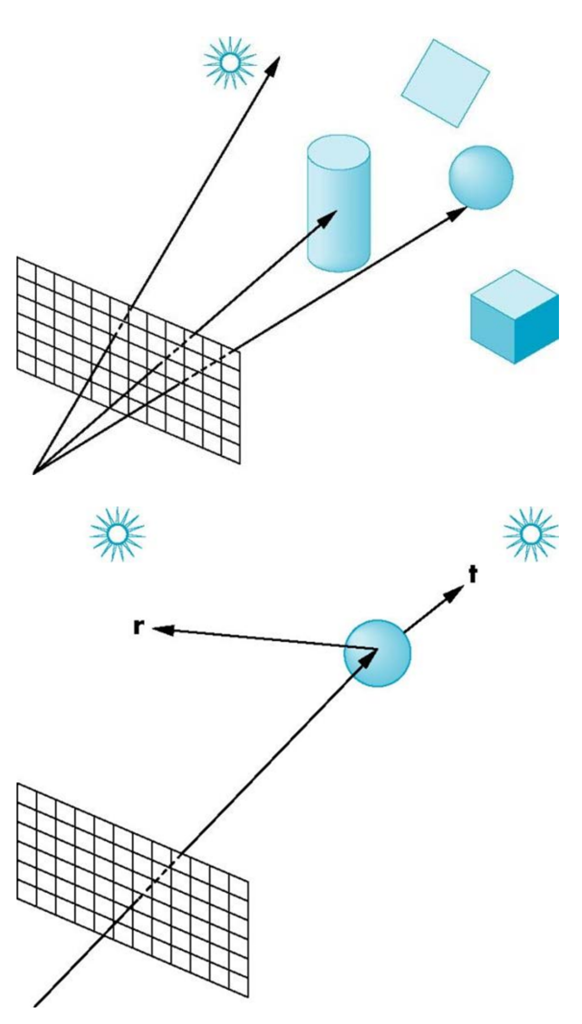
Displacement Mapping

- Texture maps can be used to actually move surface points.
- This is called ***displacement mapping***. How is this fundamentally different than bump mapping?



Backward Ray Tracing

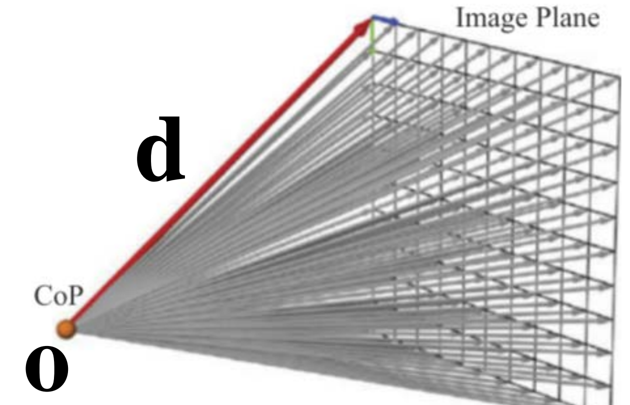
- **Ray casting**: Compute illumination at first intersected surface point only
  - Takes care of hidden surface elimination
- **Ray tracing**: Recursively spawn rays at hit points to simulate reflection, refraction, etc.



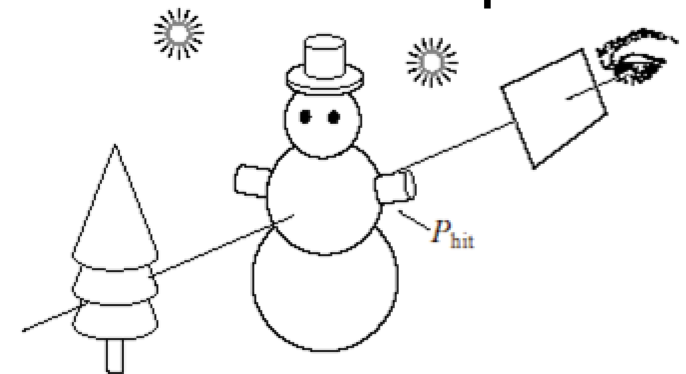
Ray Casting: Details

- Must compute 3D ray into scene for each 2D image pixel

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



- Compute 3-D **position** of ray's intersection with nearest object and normal at that point
- Apply lighting model such as Phong to get color at that point and fill in pixel with it



Ray-Sphere Intersection I

- Combine implicit definition of sphere

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

with ray equation

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

Thus we have

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

Ray-Sphere Intersection II

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

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

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

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

  - d is a unit vector |d| = 1
- Notes
  - Real solutions mean there actually are 1 or 2 intersections -- what does this correspond to?
  - Negative solutions are behind eye

Ray-Polygon Intersection

- Express point **p** on a ray as some distance t along direction d from origin **o**: **p** = **o** + t**d**
- Use plane equation **n** · **x** + m= 0, substitute **o** + t**d** for **x**, and solve for t
- Only positive t's mean the intersection is in front of the eye
- Then plug t back into **p** = **o** + t**d** to get **p**
- Is the 2-D location of p on the plane inside the 2-D polygon?
  - For convex polys, Cohen-Sutherland-style outcode test will work

Ray-Triangle Intersection

- Direct barycentric coordinates expression

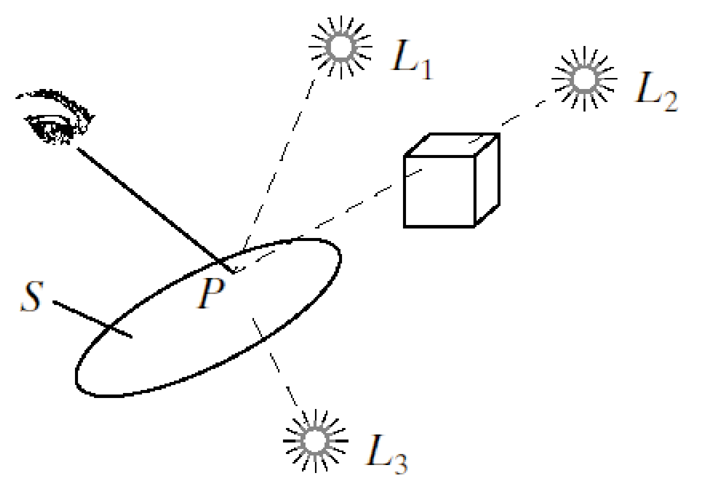
$$\mathbf{t}(u,v) = (1-u-v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$$

- Set this equal to parametric form of ray **o** + t**d** and solve for intersection point (t, u, v)
- Only inside triangle if u, v, and 1 – u – v are between 0 and 1

Shadow Rays

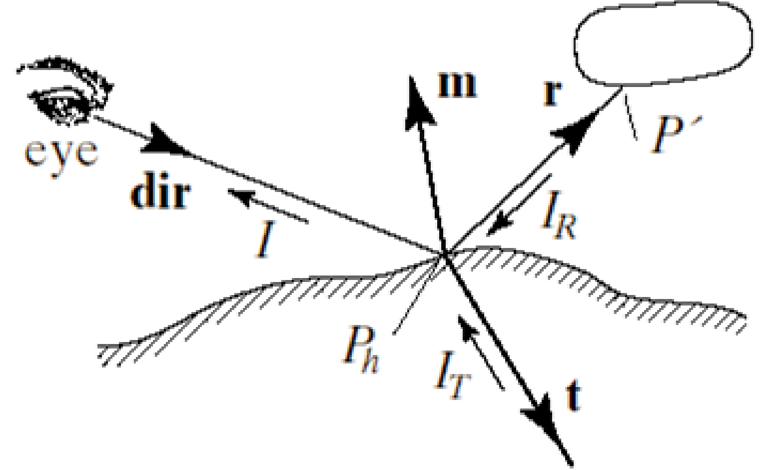
- For point **p** being locally shaded, only add diffuse & specular components for light l if light is not blocked
- Test for occlusion of l for **p**:
  - Spawn **shadow ray** for l with origin **p**, direction **l**(l)
  - Check whether shadow ray intersects any scene object
  - Intersection only "counts"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

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

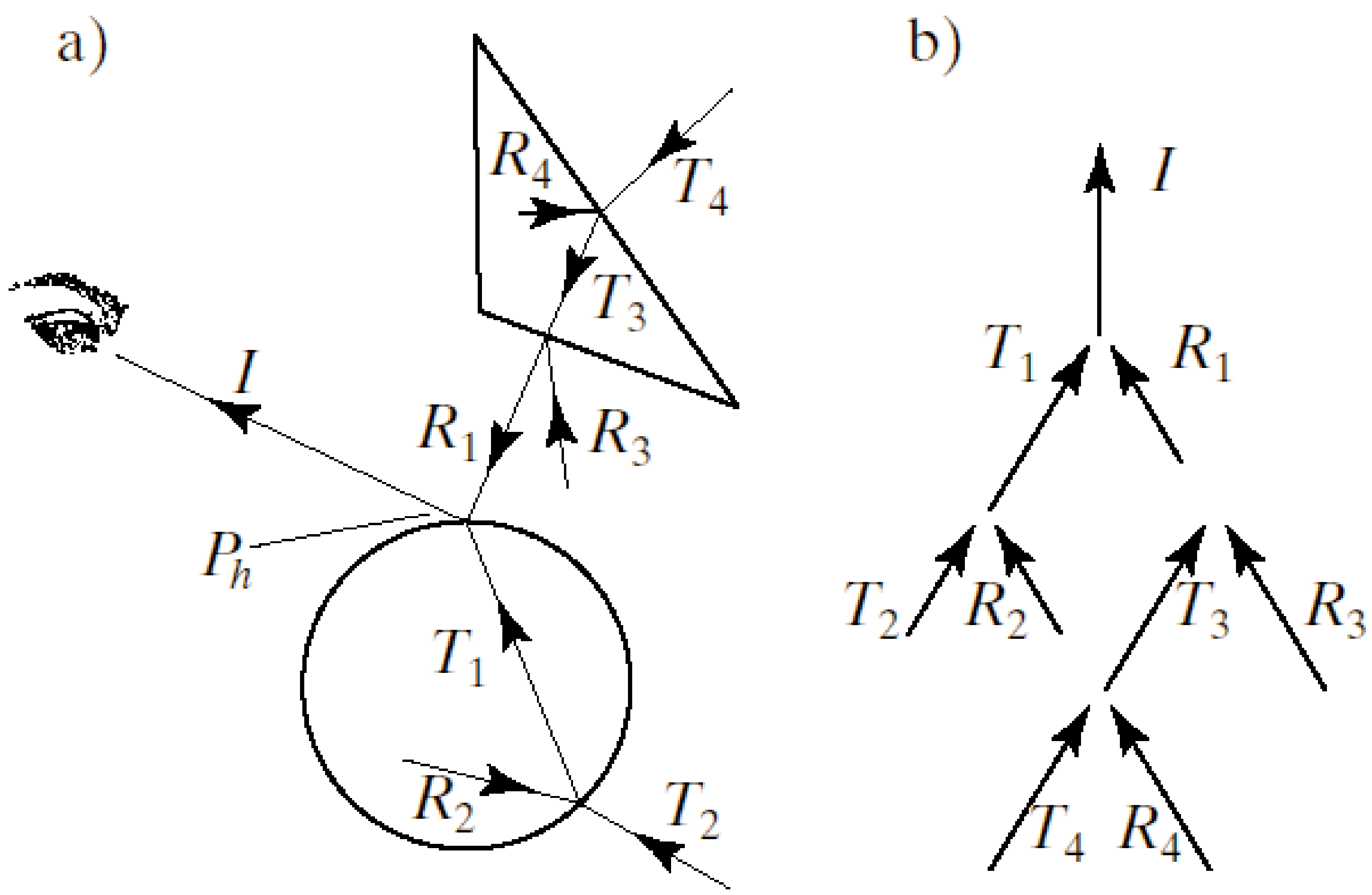


Ray Tracing

- Model: Perceived color at point **p** is an additive combination of local illumination (e.g., Phong) + reflection + refraction effects
  - Weights on last two terms are additional material properties
- 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: Recursion



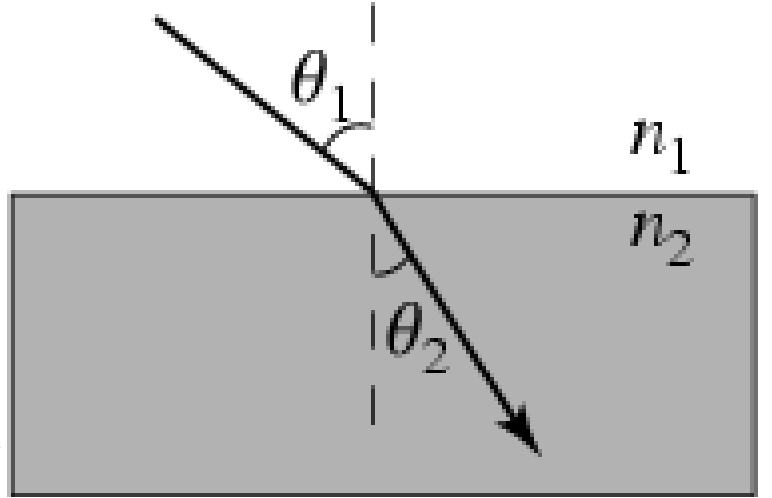
Ray Tracing Reflection Formula

- 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:

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

Refraction

- Definition: Bending of light ray as it crosses interface between media (e.g., air → glass or vice versa)
- Index of refraction (**IOR**) n for a medium: Ratio of speed of light in vacuum to speed in that medium (wavelength-dependent ⇒ prisms)
- By definition, $n \geq 1$
- Examples: n_{air} (1.0003) < n_{water} (1.33) < n_{glass} (1.52)



courtesy of Wolfram

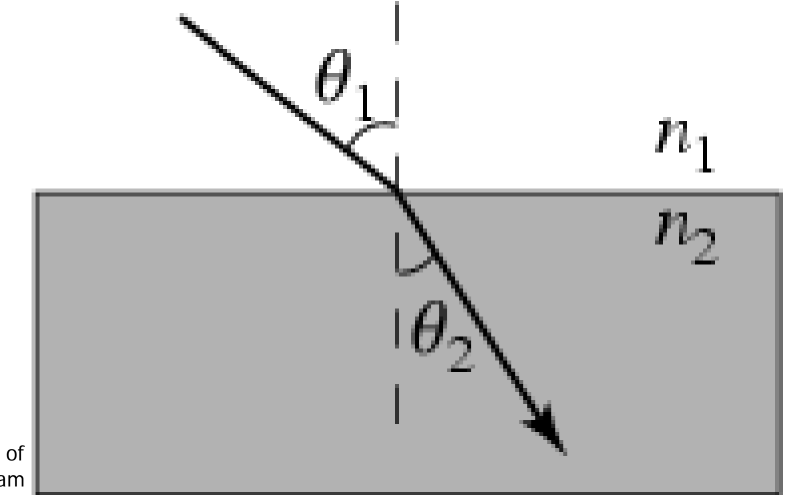
θ_1: Angle of incidence

θ_2: Angle of refraction

Snell's Law

- The relationship between the angle of incidence and the angle of refraction is given by:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2$$



courtesy of Wolfram

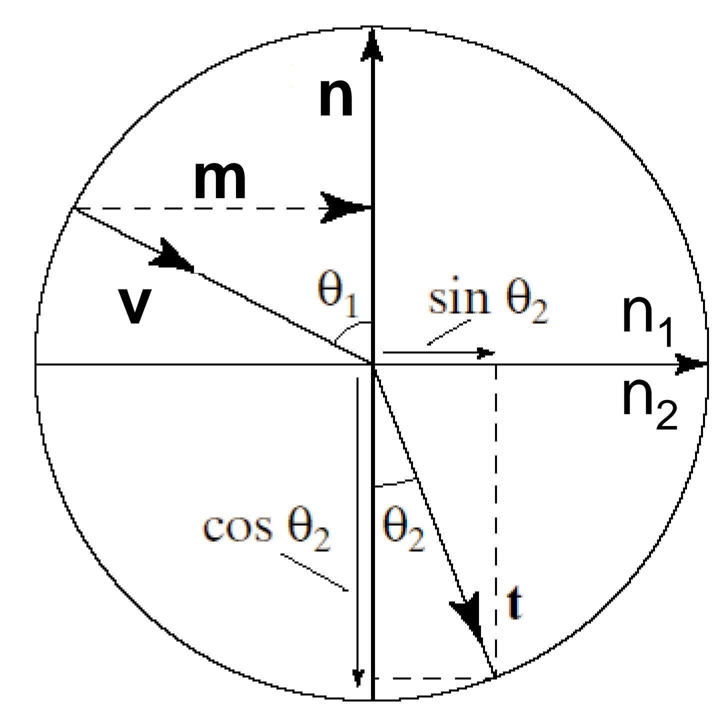
Computing the Transmission Direction t

$$n = \frac{n_1}{n_2}$$

$$c_1 = \cos\theta_1 = -\mathbf{v}\cdot\mathbf{n}$$

$$c_2 = \cos\theta_2 = \sqrt{1 - n^2(1 - c_1^2)}$$

$$\mathbf{t} = n\mathbf{v} + (nc_1 - c_2)\mathbf{n}$$



Total internal reflection happens when the term in the square root above isn't positive, which is when

$$n^2(1 - c_1^2) \geq 1$$

Distributed Ray Tracing (DRT)

- **Main idea**: Replace our single ray approximations with a *distribution of rays*
- Improvements to this image:
  - Anti-aliased edges
  - Objects in/out of focus according to a lens
  - Motion blur of fast moving objects
  - Soft shadows
  - Glossy reflection
  - “Glossy” translucency

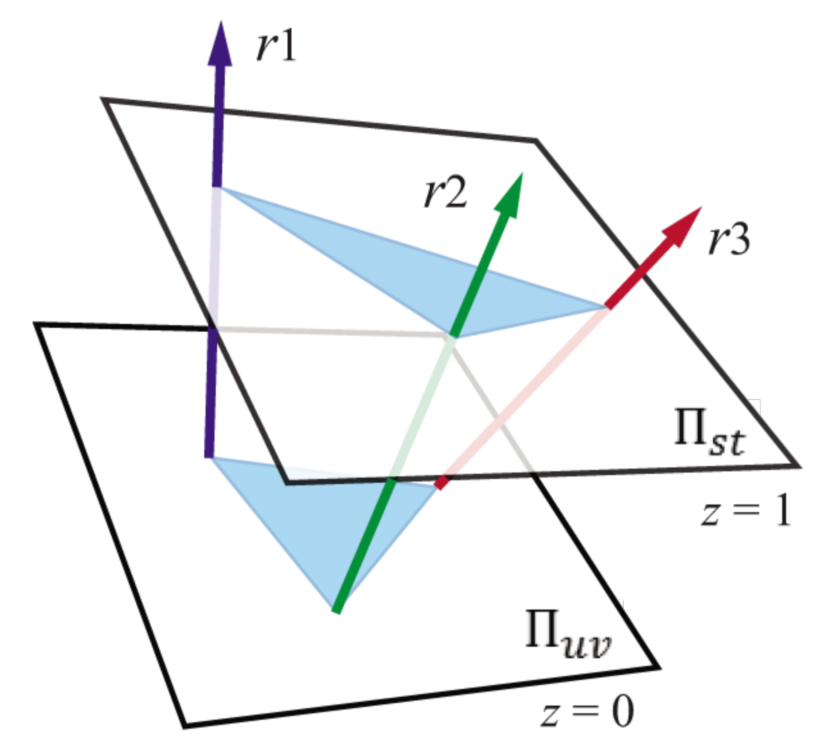
Two-Plane Parameterization(2PP)

- Parameterized over two parallel planes (2PP)
- Each ray maps to a 4D point [*u, v, s, t*]
- Relative 2PP

$$\vec{r} = [u, v, \sigma, \tau]$$

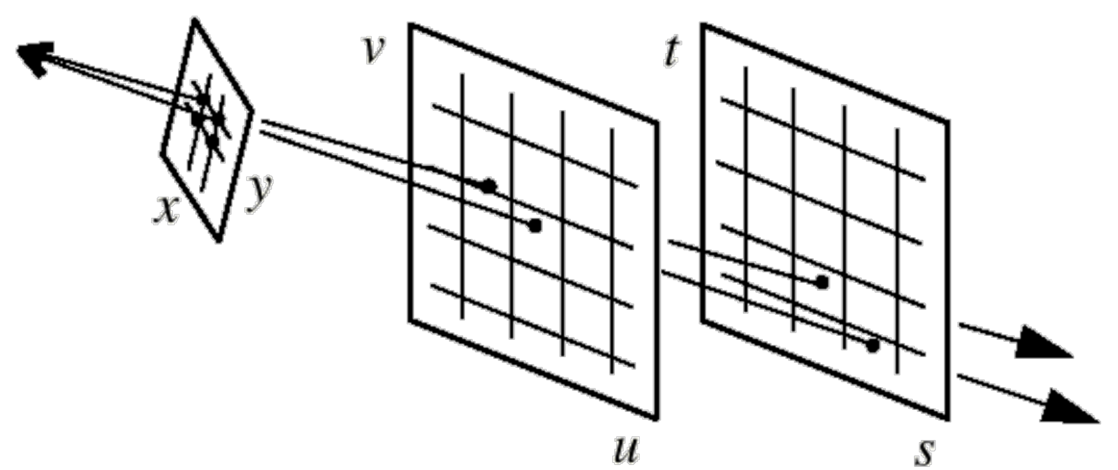
$$\sigma = s - u, \tau = t - v$$

- Ray direction: [*σ, τ, 1*]



Resampling

- For each pixel
  - generate a ray
  - find the closest rays in the light field
  - return a combination of the radiance of those rays



Reminders

- Final exam
  - 12/8 from 3:00-5:00pm
  - 218 Tureaud Hall
  - You can bring one page single-sided note and calculator
- Programming Assignment 4 is due on 12/5
- Course evaluation
  - Open until Dec 3
  - Online: www.cae.lsu.edu/eval
  - Course ID: CSC 4356 001