Basic Computer Graphics

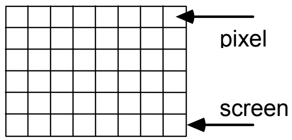
Kadi Bouatouch
IRISA
Email: kadi@irisa.fr

Summary

Introduction

Image

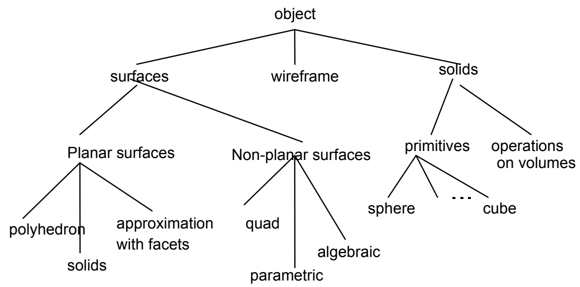
- Image = set of pixels
- Synthesis ⇔ assign an intensity value (color) to each pixel
- Intensity depends on the used light sources, the location of the viewpoint and of the objects within the scene.



What is Computer Graphics?

3D models:

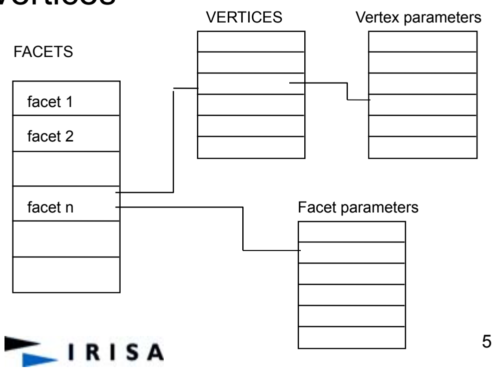
- Scene = set of different kinds



What is Computer Graphics?

3D models: facets

- object = {planar facets}
- scenes = list of facets
- facet = list of vertices

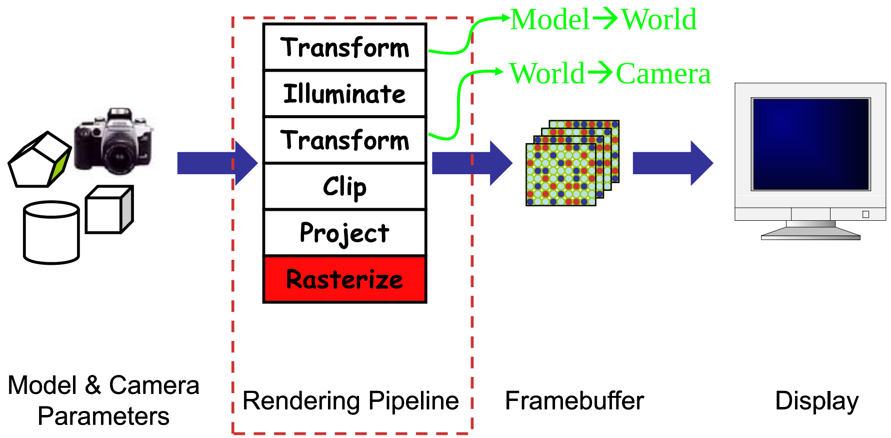


Introduction

The different processing:

- Geometric transformations.
- Clipping: Pyramidal view volume.
- Hidden surface removal.
- Cast shadows.
- Polygon filling.
- Transparent objects.
- Aliasing
- Texture mapping.

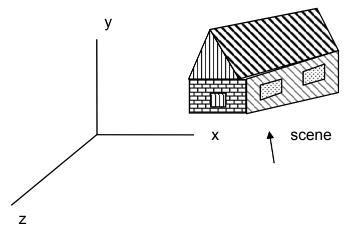
Introduction
The Rendering Pipeline



Coordinate systems

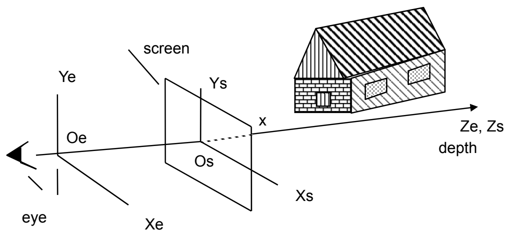
At least 3 coordinate systems:

- Word Coordinate System (o,x,y,z): in which is described the scene.



Coordinate systems

- View Coordinate System (oe,xe,ye,ze).
- z = depth axis.
- Screen Coordinate System: (os,xs,ys,zs)



Geometric transformations

- Interest: enlarge a scene, translate it, rotate it for changing the viewpoint (also called camera) .
- 2 ways for expressing a transformation:
  - with a matrix.
  - by composing transformations such as:
    - translation
    - Scaling
    - rotation

Geometric transformations

- Translation
- Scaling

$$T(t_x,t_y,t_z)=\begin{bmatrix}1 & 0 & 0 & t_x\\ 0 & 1 & 0 & t_y\\ 0 & 0 & 1 & t_z\\ 0 & 0 & 0 & 1\end{bmatrix}\quad S(s_x,s_y,s_z)=\begin{bmatrix}s_x & 0 & 0 & 0\\ 0 & s_y & 0 & 0\\ 0 & 0 & s_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Geometric transformations

- Rotation
  - One axis and one angle
  - Expression: matrix
- Also composition of rotation around each axis of the coordinate system

Geometric transformations

- Rotations around axes
  - X
  - Y
  - Z

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Geometric transformations

- Rotation around an axis : the sub-matrix A is orthogonal : At*A = I

$$R = \begin{bmatrix} R11 & R12 & R13 & 0 \\ R21 & R22 & R23 & 0 \\ R31 & R32 & R33 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \text{sub} \\ \text{matrix A} \\ \text{(3*3)} \end{bmatrix} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix}$$

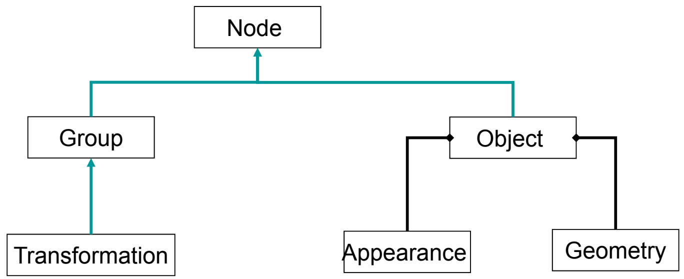
Geometric transformations

- Position the objects one with respect to the others
  - the vase is on the commode

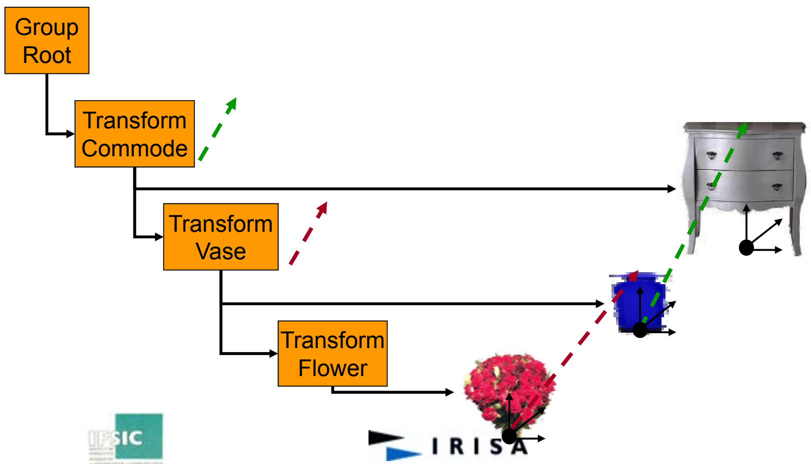
- Modeling without accounting for the final position

Geometric transformations

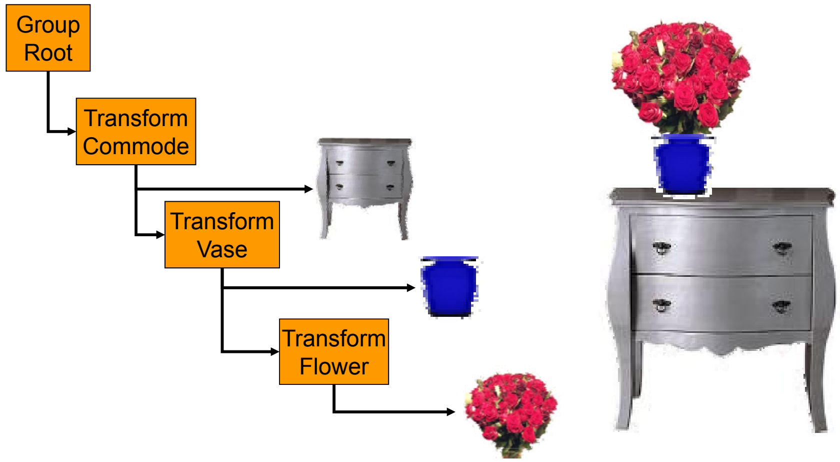
- Scene graph



Geometric transformations



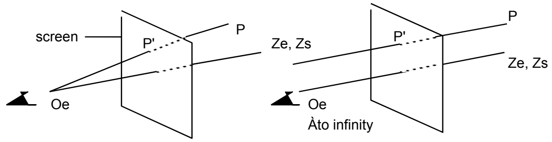
Geometric transformations



Geometric transformations for rendering

Projections

- **Plane projection = transformation which associates a screen point with a scene point**
- **It is defined by:**
  - **a centre of projection**
  - **a projection plane**
- **Two types : perspective and parallel**



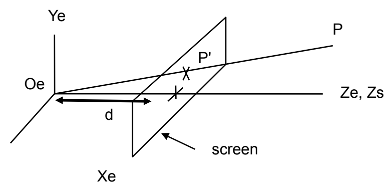
Perspective projection

Parallel projection

Geometric transformations for rendering

- **P'(xp, yp, d) = projection of P(x, y, z)**
- **d = focal distance**
- **We get:**

yp = d * y / z   et   xp = d * x / z



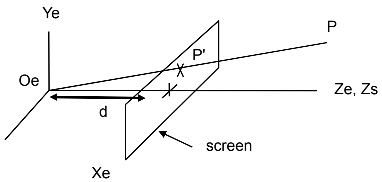
$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Geometric transformations for rendering

Homogeneous coordinates

- **Homo. Coord. : (X, Y, Z, W) = (x, y, z, 1) * Mper**
- **We get: (X, Y, Z, W) = (x, y, z, z / d)**
- **Perspective projection of P :**

(X/W, Y/W, Z/W, 1) = (xp, yp, zp, 1) = (x * d / z, y * d / z, d, 1)

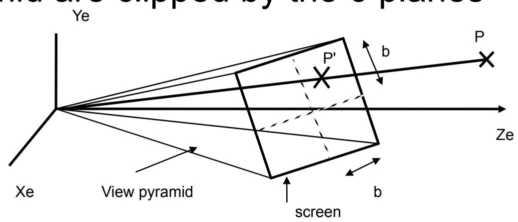


$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Geometric transformations for rendering

Clipping

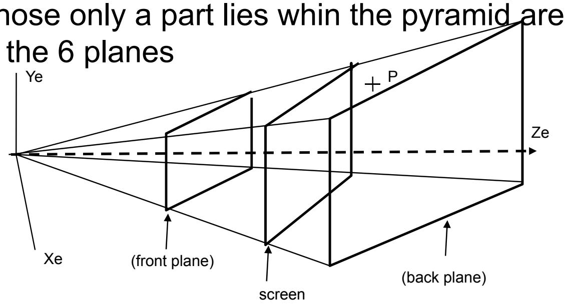
- Visible objects: inside the view pyramid
- Made up of 6 planes
- Objects whose only a part lies whin the pyramid are clipped by the 6 planes



Geometric transformations for rendering

Clipping

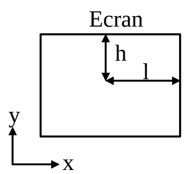
- Visible objects: inside the view pyramid
- Made up of 6 planes
- Objects whose only a part lies whin the pyramid are clipped by the 6 planes

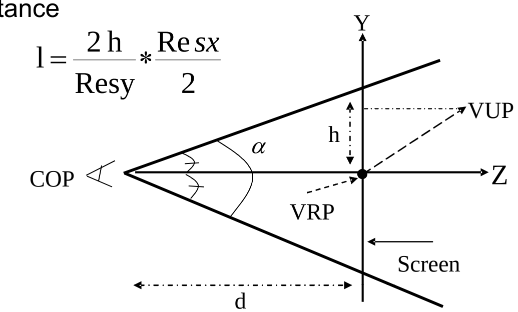


Choosing the camera parameters

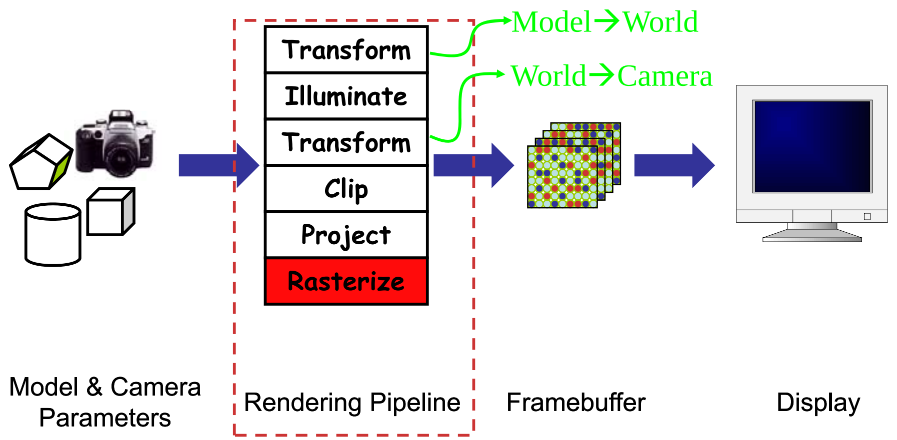
- **Resolution :** *Resx* x *Resy* (Nbr of columns) x (Nbre of rows)
  COP = Center Of Projection (observer).
  VRP = View Reference Point (targetetted point).
  VPN = View Point Normal (screen normal).
  VUP = View Up Vector
  d = focal distance

$$h = (\mathrm{tg}\,\alpha / 2) * d \qquad l = \frac{2\,h}{Resy} * \frac{Resx}{2}$$





Introduction: Rasterization



Rasterizing: Polygons

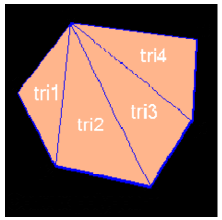
- In interactive graphics, polygons rule the world
- Two main reasons:
  - Lowest common denominator for surfaces
    - Can represent any surface *with arbitrary accuracy*
    - Splines, mathematical functions, volumetric isosurfaces…
  - Mathematical simplicity lends itself to simple, regular rendering algorithms
    - Like those we're about to discuss…
    - Such algorithms embed well in hardware

Rasterizing: Polygons

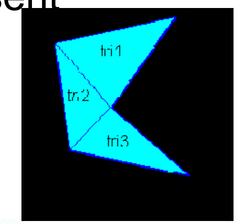
- Triangle is the *minimal unit* of a polygon
  - All polygons can be broken up into triangles
  - Triangles are guaranteed to be:
    - Planar
    - Convex

Rasterizing: Triangulation

- Convex polygons easily triangulated (Delaunay)



- Concave polygons present a challenge

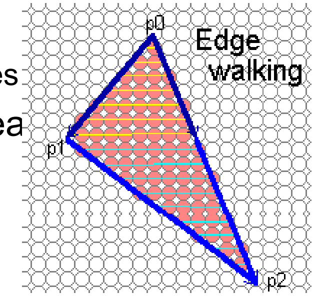


Rasterizing Triangles

- Interactive graphics hardware commonly uses *edge walking* or *edge equation* techniques for rasterizing triangles

Rasterization: Edge Walking

- Basic idea:
  - Draw edges vertically
    - Interpolate colors down edges
  - Fill in horizontal spans for each scanline
    - At each scanline, interpolate edge colors across span



Rasterization: Edge Walking

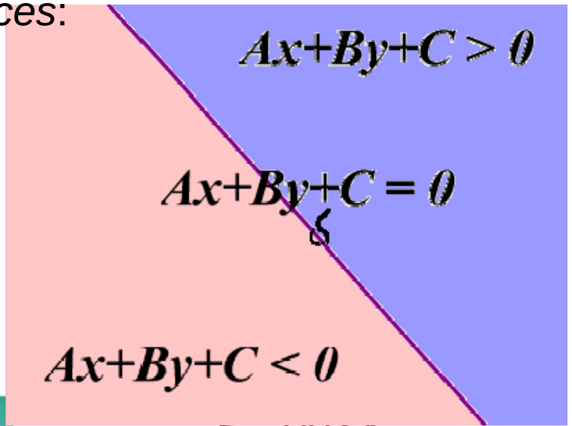
- Order three triangle vertices in x and y
  - Find middle point in y dimension and compute if it is to the left or right of polygon. Also could be flat top or flat bottom triangle
- We know where left and right edges are.
  - Proceed from top scanline downwards
  - Fill each span
  - Until breakpoint or bottom vertex is reached

Rasterization: Edge Equations

- An edge equation is simply the equation of the line defining that edge
  - Q: ***What is the implicit equation of a line?***
  - A: $Ax + By + C = 0$
  - Q: ***Given a point (x,y), what does plugging x & y into this equation tell us?***
  - A: Whether the point is:
    - On the line: $Ax + By + C = 0$
    - "Above" the line: $Ax + By + C > 0$
    - "Below" the line: $Ax + By + C < 0$

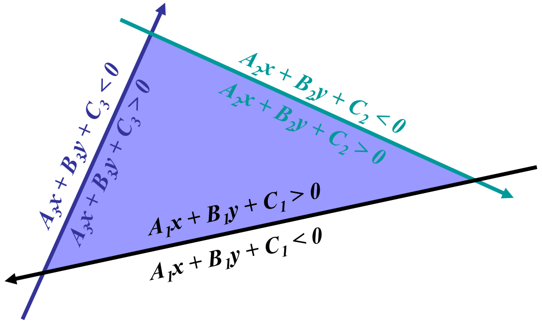
Rasterization: Edge Equations

- Edge equations thus define two *half-spaces*:



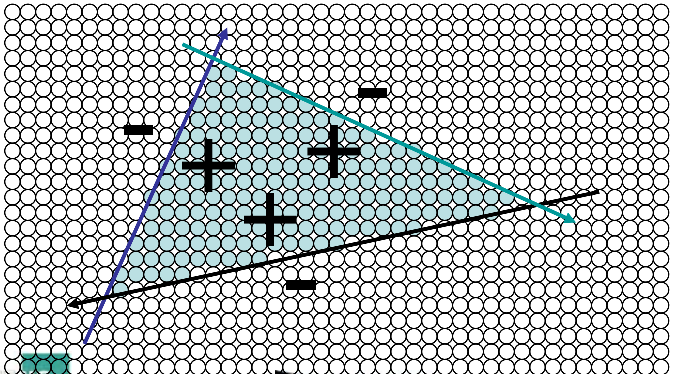
Rasterization: Edge Equations

- And a triangle can be defined as the intersection of three positive half-spaces:



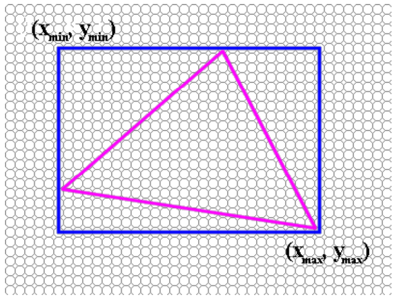
Rasterization: Edge Equations

- So…simply turn on those pixels for which all edge equations evaluate to > 0:



Rasterization: Using Edge Equations

- Which pixels: compute min,max bounding box

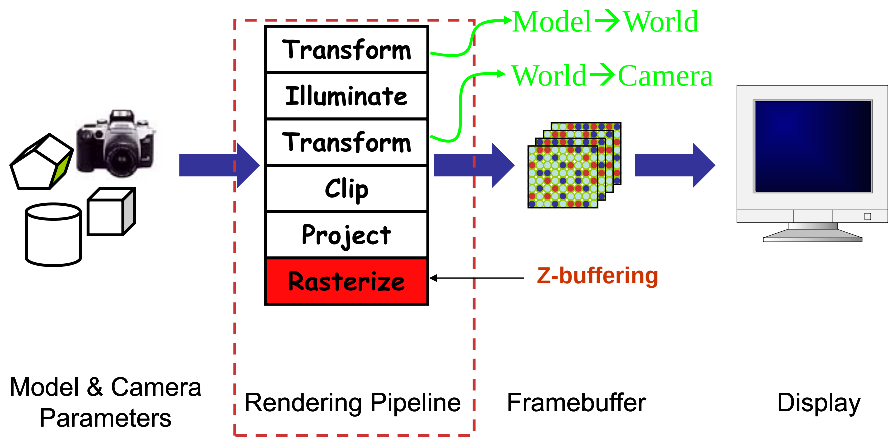


- Edge equations: compute from vertices

Rasterization: Edge Equations: Code

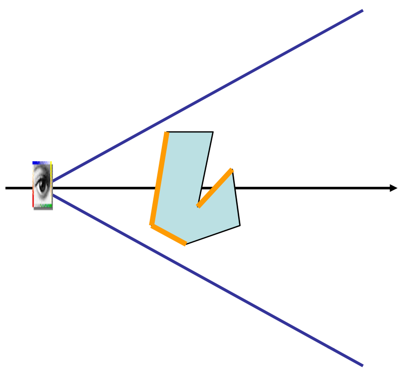
- Basic structure of code:
  - Setup: compute edge equations, bounding box
  - (Outer loop) For each scanline in bounding box...
  - (Inner loop) …check each pixel on scanline, evaluating edge equations and drawing the pixel if all three are positive

Hidden Surface Removal Z-buffering



Hidden Surface Removal Back Face Culling & Clipping

- Back Face Culling
  - Simple test
    - Normal: N
    - View direction: V
    - Dot produit: V·N
- Clipping



Hidden Surface Removal Z-buffering

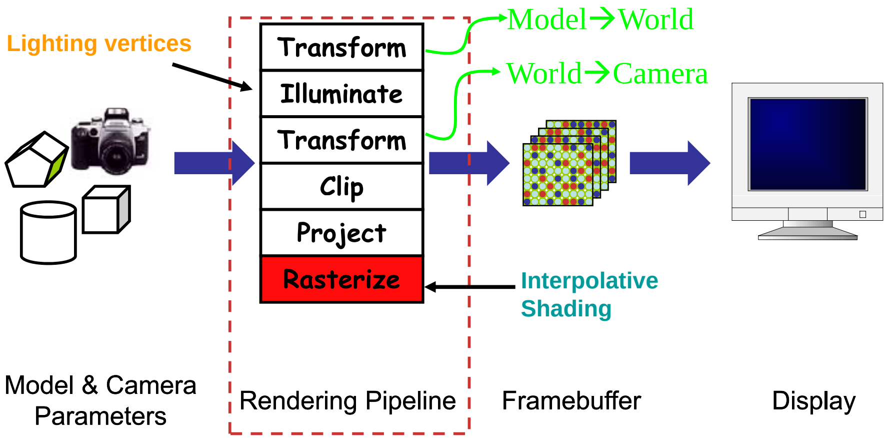
- Real-time
  - Z-Buffer (Catmull in 1974)
    - Depth memory for each pixel
  - Two 2D arrays
    - Frame buffer for intensities (colors): FB [i] [j]
    - Z-buffer for depths (z coordinate) ZB [i] [j]
  - Facets (triangles) are processed without any ordering

Hidden Surface Removal
Z-buffering

```
algorithm Z-Buffer ()
begin
    for (for pixels i,j do)
          FB [i][j] ← back plane's color ; ZB [i][j] ← z (back plane)
    endfor
    for (all facets) do
          for (all pixels within the projection of the facet) do
               compute_intensity_and_z for all pixels using interpolation
               if (zFrontPlane ≤ z of polygone at point i,j ≤ ZB [i][j]) then
                    ZB [i][j] ← z of facet at pixel (i,j)
                    FB [i][j] ← color of facet at (i,j)
               endif
          endfor
     endfor
end
```

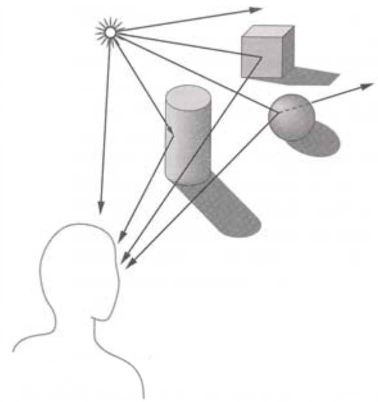

Lighting

Lighting



Lighting: Illumination

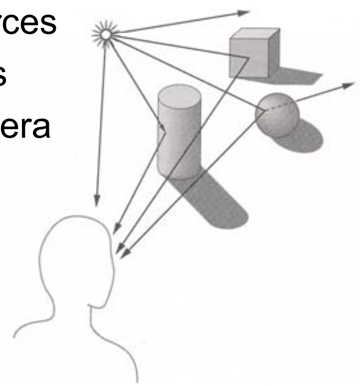
- How do we compute radiance for a sample ray?



Angel Figure 6.2

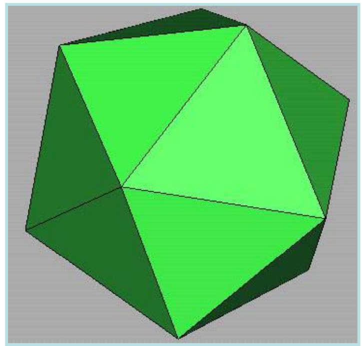
Lighting: Goal

- Must derive computer models for ...
  - Emission at light sources
  - Scattering at surfaces
  - Reception at the camera
- Desirable features …
  - Concise
  - Efficient to compute
  - "Accurate"



Lighting: Overview

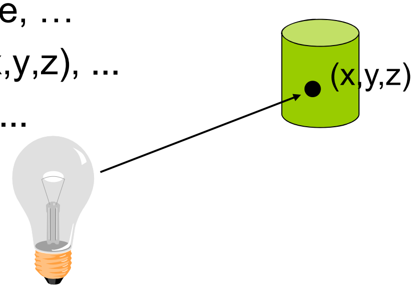
- Direct (Local) Illumination
  - Emission at light sources
  - Scattering at surfaces
- Global illumination
  - Shadows
  - Refractions
  - Inter-object reflections



Direct Illumination

Lighting: Modeling Light Sources

- $I_L(x,y,z,\theta,\phi,\lambda)$ ...
  - describes the intensity of energy,
  - leaving a light source, …
  - arriving at location(x,y,z), ...
  - from direction (θ,ϕ), ...
  - with wavelength λ



Lighting: Ambient Light Sources

- Objects not directly lit are typically still visible
  - e.g., the ceiling in this room, undersides of desks
- This is the result of *indirect illumination* from emitters, bouncing off intermediate surfaces
- Too expensive to calculate (in real time), so we use a hack called an *ambient light source*
  - No spatial or directional characteristics; illuminates all surfaces equally
  - Amount reflected depends on surface properties

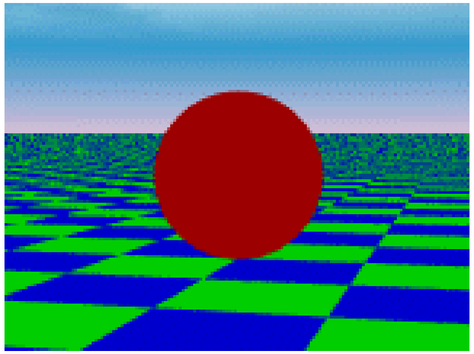
Lighting: Ambient Light Sources

- For each sampled wavelength (R, G, B), the ambient light reflected from a surface depends on
  - The surface properties, $k_{ambient}$
  - The intensity, $I_{ambient,}$ of the ambient light source (constant for all points on all surfaces)
    - $I_{reflected} = k_{ambient} I_{ambient}$

IRISA

Lighting: Ambient Light Sources

- A scene lit only with an ambient light source:



Light Position Not Important

Viewer Position Not Important

Surface Angle Not Important

IRISA

Lighting: Ambient Term

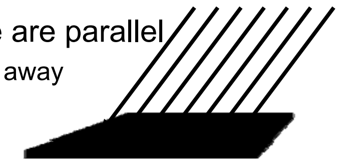
- Represents reflection of all indirect illumination

This is a total hack (avoids complexity of global illumination)!

IRISA

Lighting: Directional Light Sources

- For a *directional light source* we make simplifying assumptions
  - Direction is constant for all surfaces in the scene
  - All rays of light from the source are parallel
    - As if the source were infinitely far away from the surfaces in the scene
    - A good approximation to sunlight

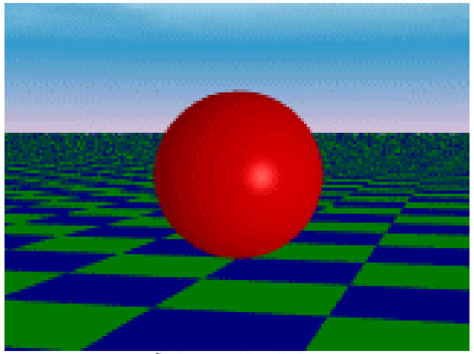


- The direction from a surface to the light source is important in lighting the surface

IRISA

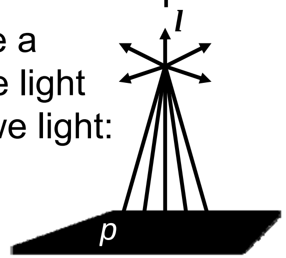
Lighting: Directional Light Sources

- The same scene lit with a directional and an ambient light source



Lighting: Point Light Sources

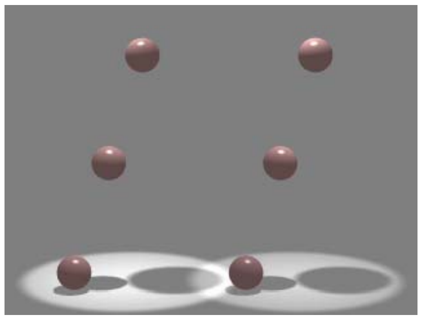
- A *point light source* emits light equally in all directions from a single point
- The direction to the light from a point on a surface thus differs for different points:
  - So we need to calculate a normalized vector to the light source for every point we light:



$$\bar{d} = \frac{\bar{p} - \bar{l}}{\left\| \bar{p} - \bar{l} \right\|}$$

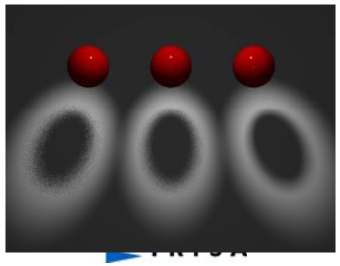
Lighting: Other Light Sources

- *Spotlights* are point sources whose intensity falls off directionally.
  - Requires color, point direction, falloff parameters
  - Supported by OpenGL



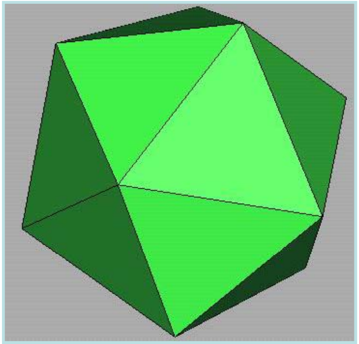
Lighting: Other Light Sources

- *Area light sources* define a 2-D emissive surface (usually a disc or polygon)
  - Good example: fluorescent light panels
  - Capable of generating *soft shadows* (*why?*)



Lighting: Overview

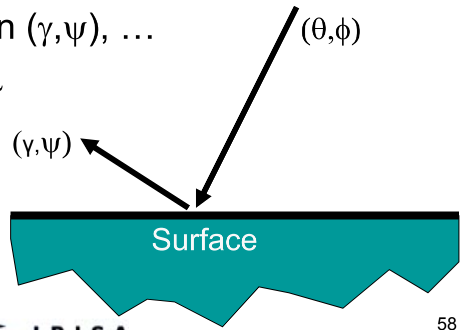
- Direct (Local) Illumination
  - Emission at light sources
  - Scattering at surfaces
- Global illumination
  - Shadows
  - Refractions
  - Inter-object reflections



Direct Illumination

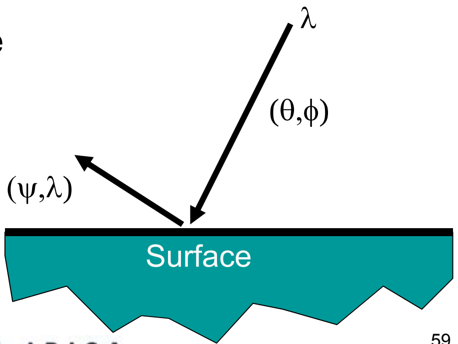
Lighting: Modeling Surface Reflectance

- $R_s(\theta,\phi,\gamma,\psi,\lambda)$ ...
  - describes the amount of incident energy,
  - arriving from direction (θ,ϕ), ...
  - leaving in direction (γ,ψ), …
  - with wavelength λ



Lighting: Empirical Models

- Ideally measure radiant energy for “all” combinations of incident angles
  - Too much storage
  - Difficult in practice



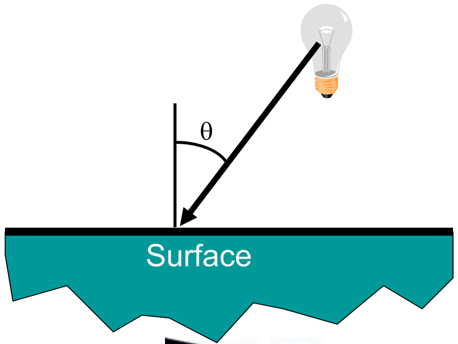
Lighting: The Physics of Reflection

- Ideal diffuse reflection
  - An *ideal diffuse reflector*, at the microscopic level, is a very rough surface (real-world example: chalk)
  - Because of these microscopic variations, an incoming ray of light is equally likely to be reflected in any direction over the hemisphere:

  - *What does the reflected intensity depend on?*

Lighting: Diffuse Reflection

- How much light is reflected?
  - Depends on angle of incident light



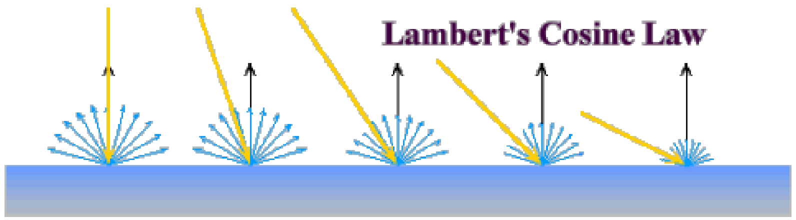
Lighting: Lambert's Cosine Law

- Ideal diffuse surfaces reflect according to *Lambert's cosine law*:

  The energy reflected by a small portion of a surface from a light source in a given direction is proportional to the cosine of the angle between that direction and the surface normal

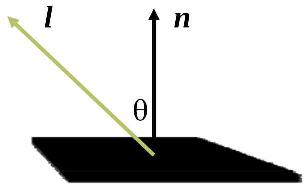
- These are often called *Lambertian surfaces*
- Note that the reflected intensity is independent of the viewing direction, but does depend on the surface orientation with regard to the light source

Lighting: Lambert's Law



Lighting: Computing Diffuse Reflection

- The angle between the surface normal and the incoming light is the *angle of incidence:*



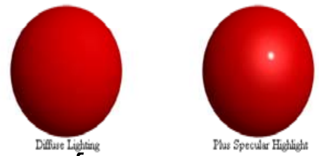
  - $I_{diffuse} = k_d \, I_{light} \, \mathbf{cos}\, \theta$
- In practice we use vector arithmetic:
  - $I_{diffuse} = k_d \, I_{light} \, (\mathbf{n} \bullet \mathbf{l})$

Lighting: Diffuse Lighting Examples

- We need only consider angles from 0° to 90° (*Why?*)
- A Lambertian sphere seen at several different lighting angles:

Lighting: Specular Reflection

- Shiny surfaces exhibit *specular reflection*
  - Polished metal
  - Glossy car finish



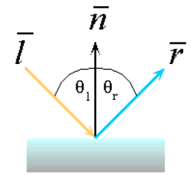
- A light shining on a specular surface causes a bright spot known as a *specular highlight*
- Where these highlights appear is a function of the viewer's position, so specular reflectance is view dependent

Lighting: The Physics of Reflection

- At the microscopic level a specular reflecting surface is very smooth
- Thus rays of light are likely to bounce off the microgeometry in a mirror-like fashion
- The smoother the surface, the closer it becomes to a perfect mirror

Lighting: The Optics of Reflection

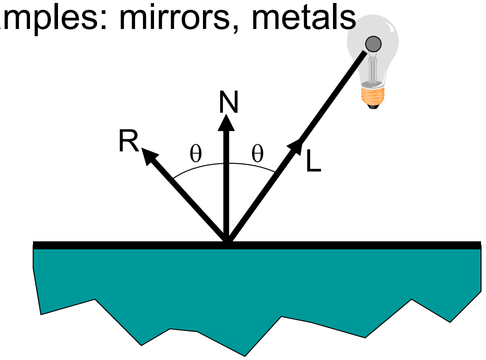
- Reflection follows *Snell's Laws:*
  - The incoming ray and reflected ray lie in a plane with the surface normal
  - The angle that the reflected ray forms with the surface normal equals the angle formed by the incoming ray and the surface normal:



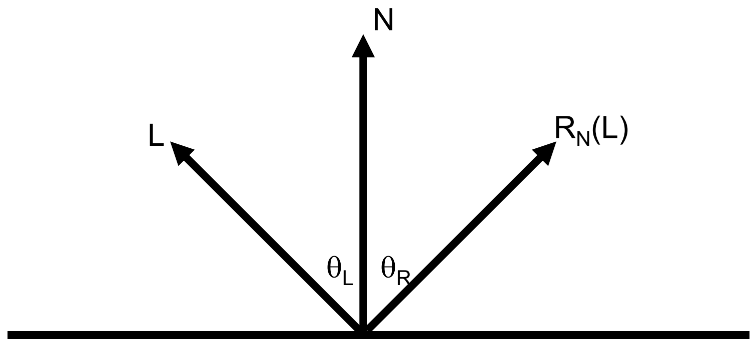
$$\theta_{(l)ight} = \theta_{(r)eflection}$$

Lighting: Specular Reflection

- Reflection is strongest near mirror angle
  - Examples: mirrors, metals

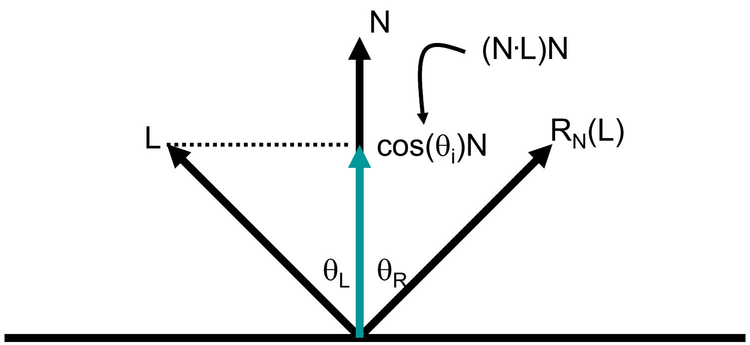


Lighting: Geometry of Reflection



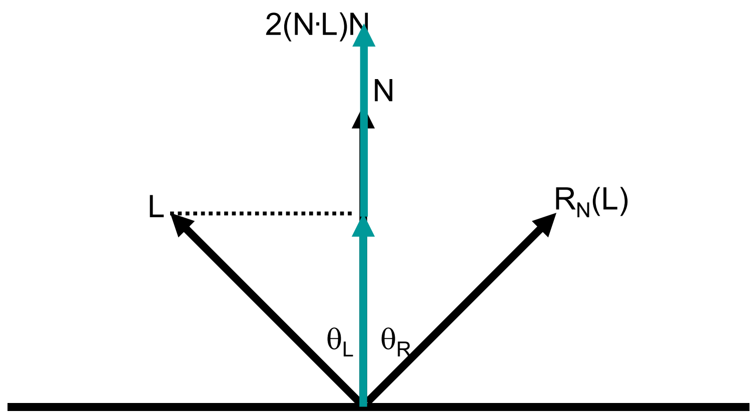
$\theta_L = \theta_R$

Lighting: Geometry of Reflection



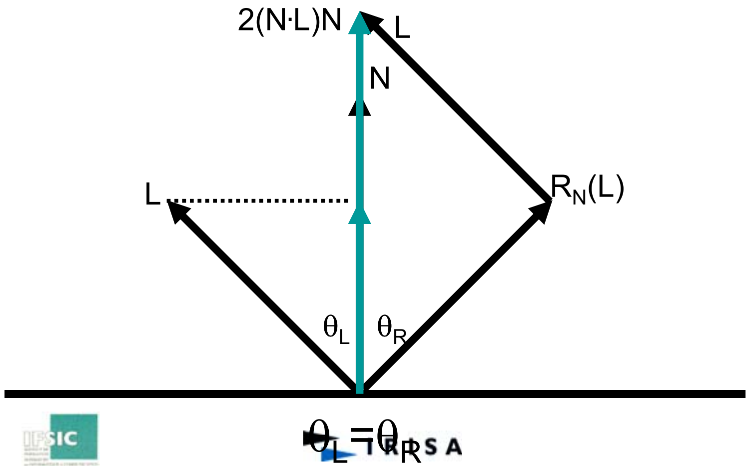
$\theta_L = \theta_R$

Lighting: Geometry of Reflection

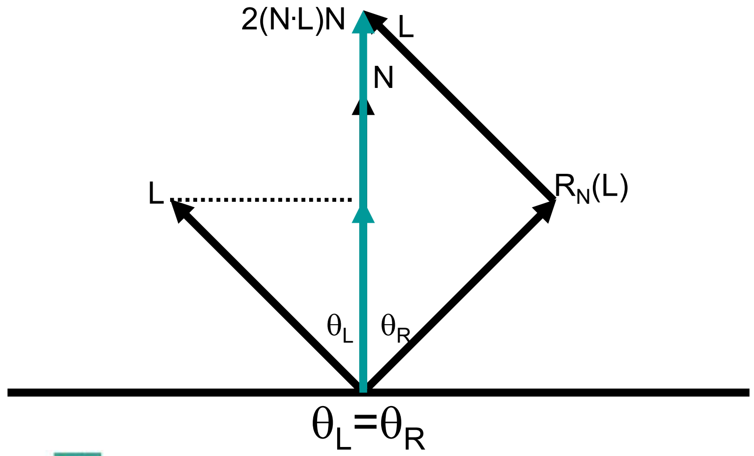


$\theta_L = \theta_R$

Lighting: Geometry of Reflection

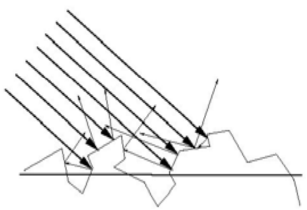


Lighting: Geometry of Reflection



Lighting: Non-Ideal Specular Reflectance

- Snell's law applies to perfect mirror-like surfaces, but aside from mirrors (and chrome) few surfaces exhibit perfect specularity
- How can we capture the "softer" reflections of surface that are glossy rather than mirror-like?



- One option: model the microgeometry of the surface and explicitly bounce rays off of it
- Or…

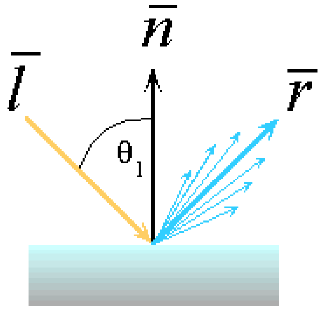
Lighting: Non-Ideal Specular Reflectance

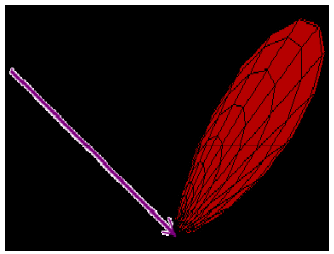
An Empirical Approximation

- Hypothesis: most light reflects according to Snell's Law
  - But because of microscopic surface variations, some light may be reflected in a direction slightly off the ideal reflected ray
- Hypothesis: as we move from the ideal reflected ray, some light is still reflected

Lighting: Non-Ideal Specular Reflectance

- An illustration of this angular falloff:





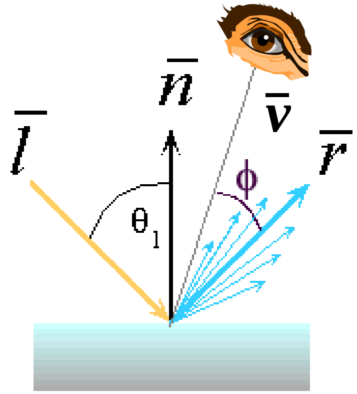
s falloff?

Lighting: Phong Lighting

- The most common lighting model in computer graphics was suggested by Phong:

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

- The n_{shiny} term is a purely empirical constant that varies the rate of falloff
- Though this model has no physical basis, it works (sort of) in practice



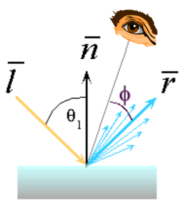
Lighting: Calculating Phong Lighting

- The **cos** term of Phong lighting can be computed using vector arithmetic:

$$\boxed{I_{specular} = k_s I_{light} (\bar{v} \cdot \bar{r})^{n_{shiny}}}$$

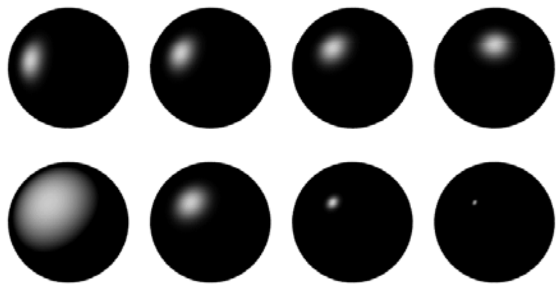
v

- v is the unit vector towards the viewer
- r is the ideal reflectance direction



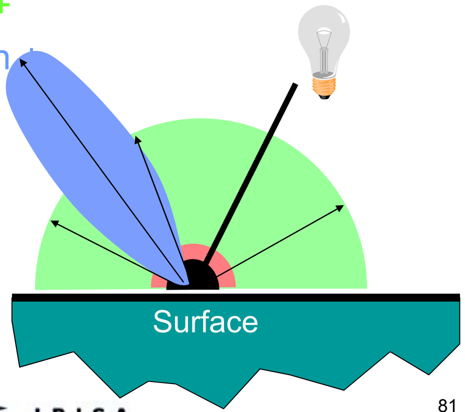
Lighting: Phong Examples

- These spheres illustrate the Phong model as l and n_{shiny} are varied:



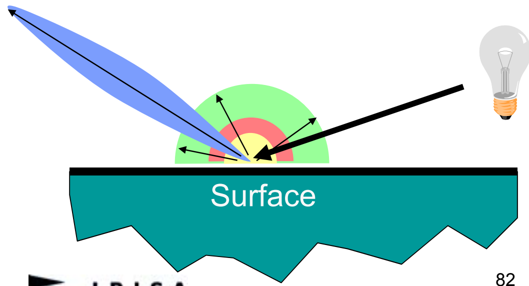
Lighting: Combining Everything

- Simple analytic model:
  - diffuse reflection +
  - specular reflection +
  - emission +
  - "ambient"



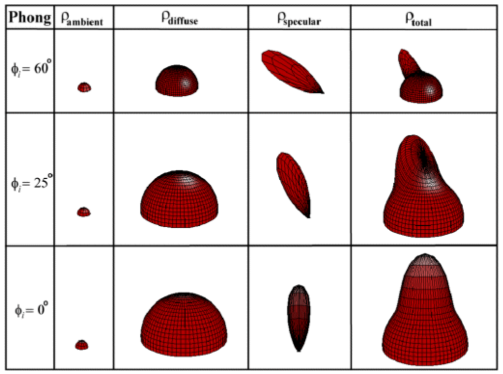
Lighting: Combining Everything

- Simple analytic model:
  - diffuse reflection +
  - specular reflection +
  - emission +
  - "ambient"



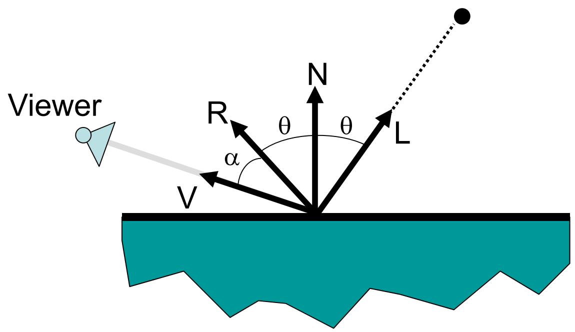
Lighting: OpenGL Reflectance Model

- Sum diffuse, specular, emission, and ambient



Lighting: The Final Combined Equation

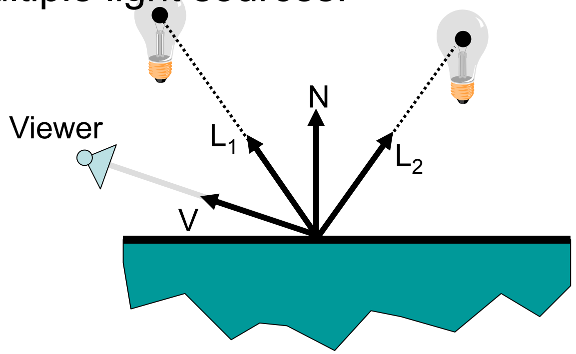
- Single light source:



$$I = I_E + K_A I_{AL} + K_D (N \bullet L) I_L + K_S (V \bullet R)^n I_L$$

Lighting: Final Combined Equation

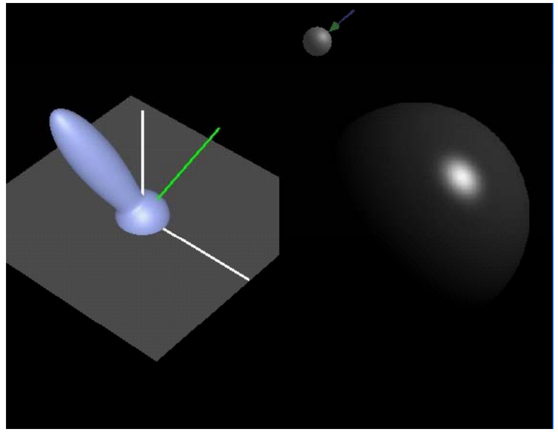
- Multiple light sources:



$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

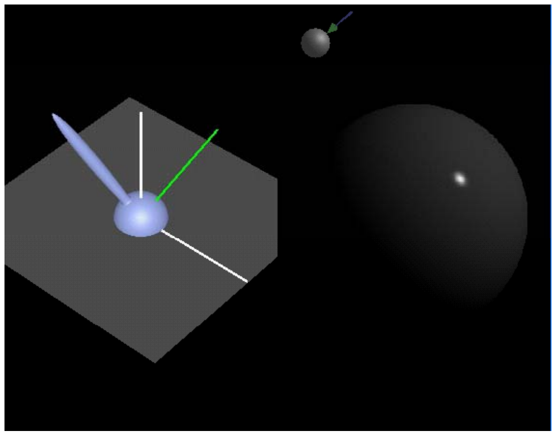
Lighting: Examples

•Paramètres :
- K_d =0.25
- K_s =0.75
- n=50.0



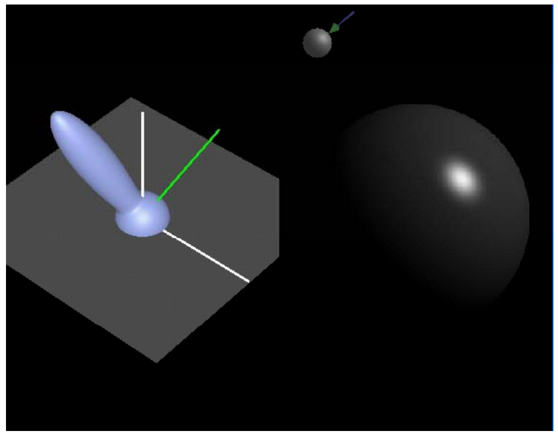
Lighting: Examples

•Paramètres :
- K_d =0.25
- K_s =0.75
- n=200.0



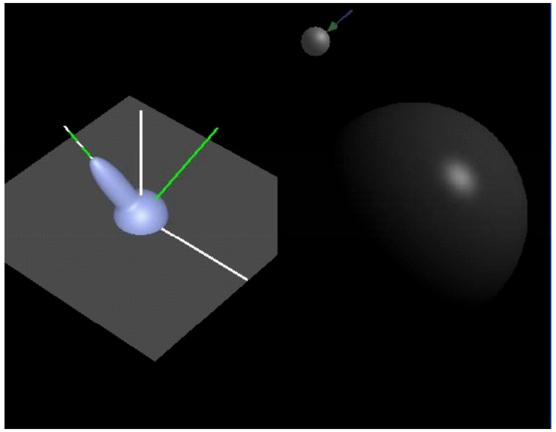
Lighting: Examples

•Paramètres :
- K_d =0.25
- K_s =0.75
- n=50.0



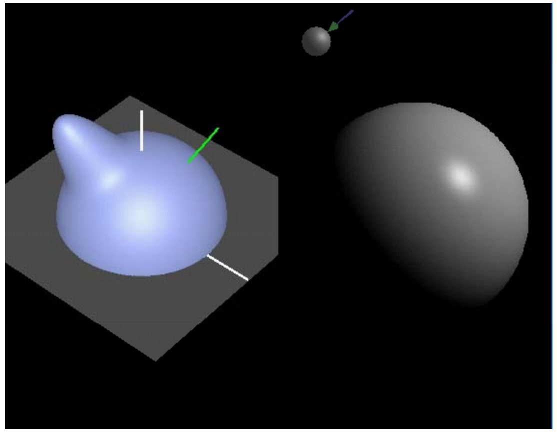
Lighting: Examples

•Paramètres :
- K_d =0.25
- K_s =0.25
- n=50.0



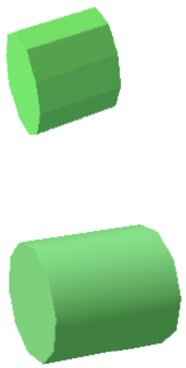
Lighting: Examples

•Paramètres :
- K_d =0.75
- K_s =0.25
- n=50.0

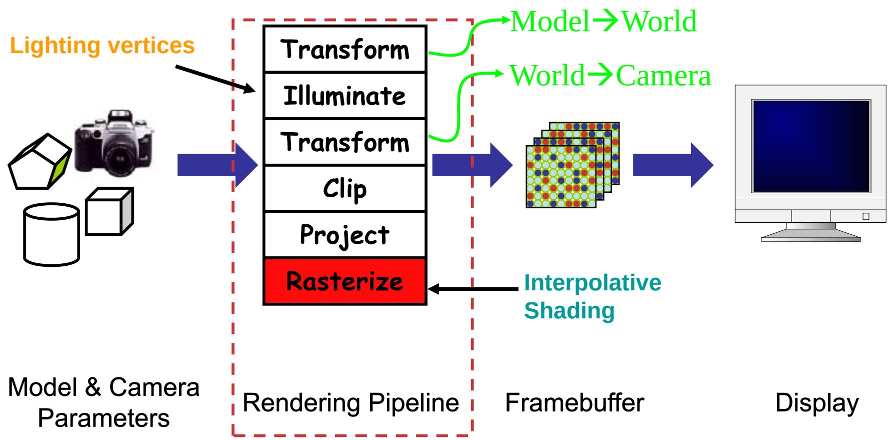


Lighting: interpolative shading

- Smooth D object representations: Quadrics superquadrics splines
  - Computing surface normals can be very expensive
- Interpolative shading:
  - approximate curved objects by polygonal meshes,
  - compute a surface normal that varies smoothly from one face to the next
  - computation can be very cheap Many objects are well approximated by polygonal meshes
  - silhouettes are still polygonal
- Done: rasterization step



Lighting



Lighting: interpolative shading

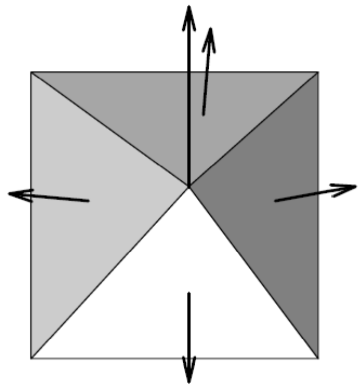
Two kinds of interpolative shading

- Gouraud Shading: cheap but gives poor highlights.
- Phong Shading: slightly more expensive but gives excellent highlights.

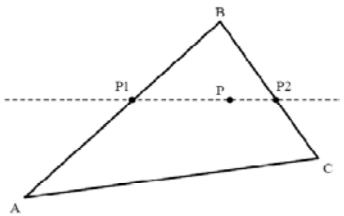
Lighting: interpolative shading

Vertex Normals

- All interpolative shading methods rely on vertex normals.
- A vertex normal is the average of the normals of all of the faces sharing that vertex.



Lighting: Gouraud Shading



- **Compute RGB Color at Each Vertex.** Use the Phong illumination model (or any other).
- **Compute RGB Colors at P_1 and P_2.** By linear interpolation:

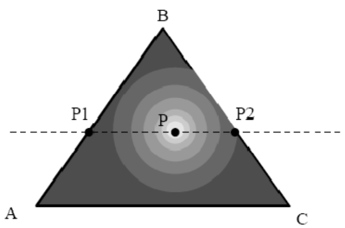
$$s = \frac{\|P_1 - B\|}{\|A - B\|} \qquad (R_{P_1}, G_{P_1}, B_{P_1}) = s(R_A, G_A, B_A) + (1 - s)(R_B, G_B, B_B)$$

- **Compute RGB Colors at P.** By linear interpolation:

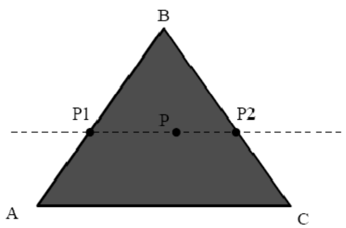
$$s = \frac{\|P - P_2\|}{\|P_1 - P_2\|} \qquad (R_P, G_P, B_P) = s(R_{P_1}, G_{P_1}, B_{P_1}) + (1 - s)(R_{P_2}, G_{P_2}, B_{P_2})$$

Lighting
Poor Highlights from Gouraud Sading

- Suppose we are approximating a sphere and that the true highlight should appear in the *center* of a face (e.g., at P).
- The computed RGBs at A, B, and C will not have highlights (because they are far away from P).
- Any point in the interior will therefore not have a highlight.



Desired

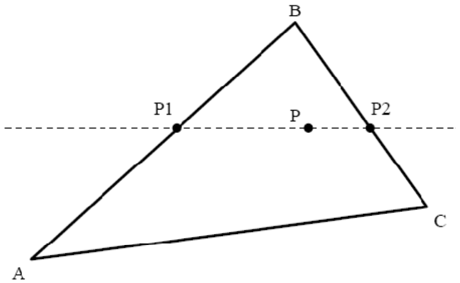


Gouraud Result

Lighting: Phong Shading

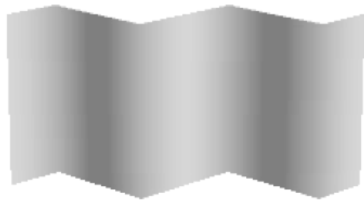
Interpolate the normals instead of the RGB values.

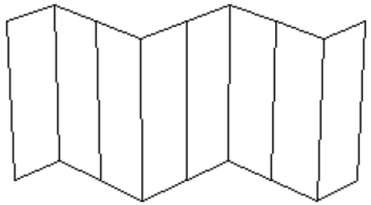
- Compute normals at each vertex A B and C.
- Compute the normals at P_1 and P_2 By interpolation using the normals from A and B and C and B.
- Compute the normal at P By interpolating the normals from P_1 and P_2.
- Compute RGB values at P Using Phong's rule.



Lighting: Phong Shading

Interpolating the normals

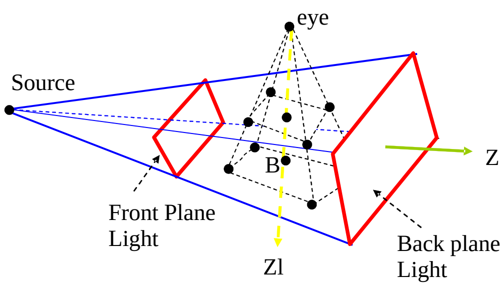




Cast Shadows

Cast Shadows: Shadow Map

Two Z-buffers: one from the light source and another from the viewer

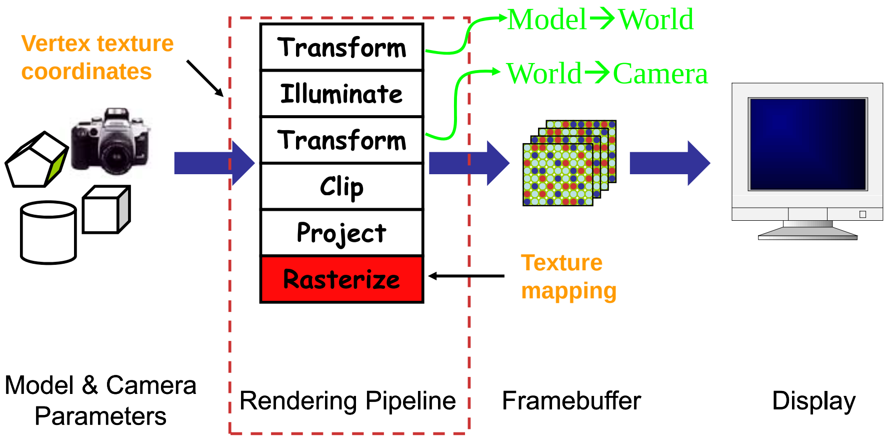


Cast Shadows: Shadow Map

- Compute an image as seen by the point light source
- Compute an image as seen by the eye
- Let (X,Y,Z) be a point seen through a pixel (x,y) of the camera screen
- Let (Xl,Yl,Zl) be the coordinates of this point in the source coordinate system and (xl,yl) the coordinates of the associated pixel on the source's screen
- If Zl > Z_Buffer_Ecl[xl][yl] then the point is shadowed
- Else the point is lit and we compute its intensity.

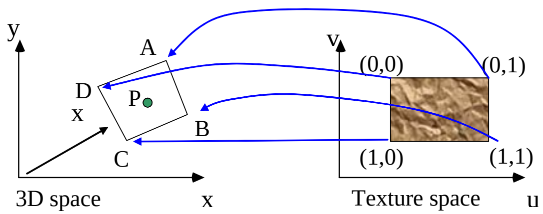
Texture Mapping

Introduction
The Rendering Pipeline



Texture Mapping

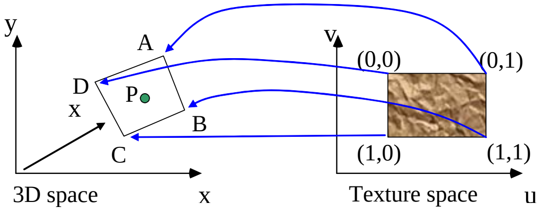
- Texture = 2D Image (we do not consider 3D textures)
- Texture : represented by a 2D array of RGB triplets
- Triplet: Color, Normal or any other thing
- Normalized texture space: (u,v), u et v ranging from 0 to 1
- For an intersection point P: compute its texture coordinates either directly or by interpolation



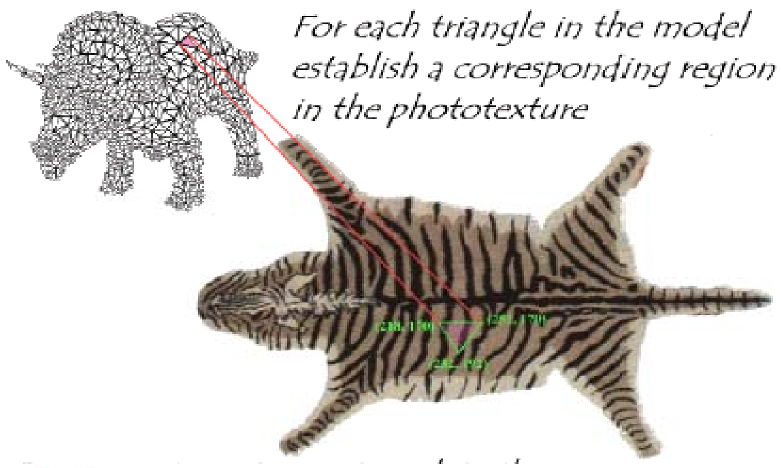
Texture Mapping

Polygon: Triangle or quadrilateral

- Each vertex of the 3D polygon is assigned texture coordinates
- For an intersection point P: compute its texture by bilinear interpolation of the texture coordinates of the polygon's vertices

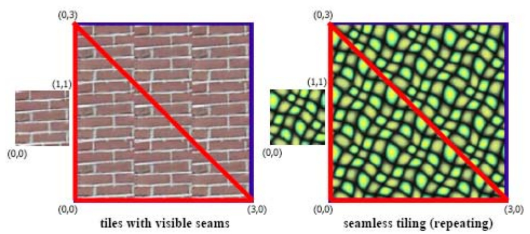


Texture Mapping



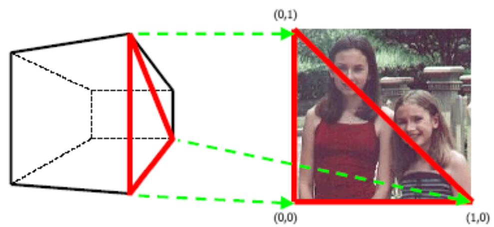
Texture Mapping

- Specify a texture coordinate (u,v) at each vertex
- Canonical texture coordinates (0,0) → (1,1)

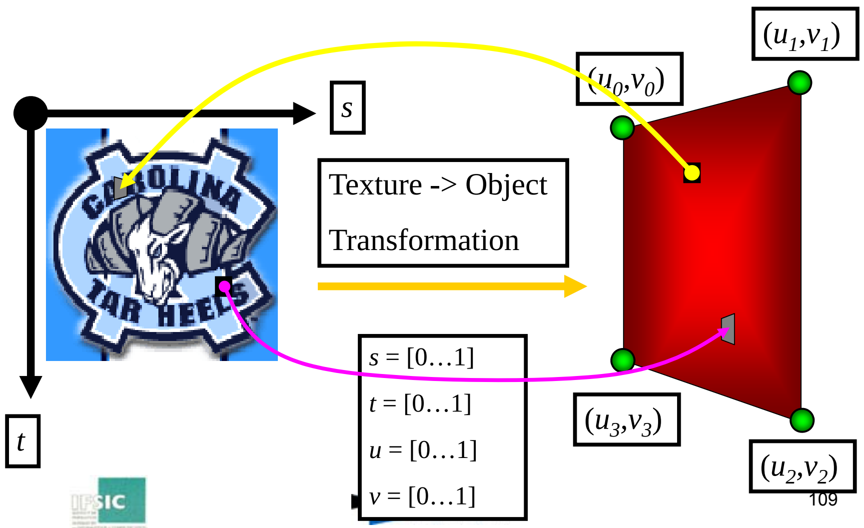


Texture Mapping: Interpolation

- Specify a texture coordinate (u,v) at each vertex
- Interpolate the texture values of intersection points lying on the polygon using those of its vertices



Texture Mapping: Interpolation



Texture Mapping

- Common texture coordinate mapping
  - planar
  - Cylindrical
  - Spherical

Texture Mapping

- Planar: Triangle
  - Barycentric coordinates: a way to parameterize a triangle with P_0, P_1 and P_2 as vertices
  - Let P be a point on the triangle and $\beta_0, \beta_1, \beta_2$ its barycentric coordinates

-Thus: $P = \beta_0 P_0 + \beta_1 P_1 + \beta_2 P_2$

$$P = (1 - \beta_1 - \beta_2) P_0 + \beta_1 P_1 + \beta_2 P_2$$

Since: $\beta_0 + \beta_1 + \beta_2 = 1$

Texture Mapping

- Planar: Triangle
  - Given an intersection P lying on a triangle
  - Compute its texture coordinates (s,t), say (β_1,β_2) by solving the following linear system:

$$P = (1 - \beta_1 - \beta_2) P_0 + \beta_1 P_1 + \beta_2 P_2$$

$$P - P_0 = \left(P_2 - P_1 \ , P_1 - P_0\right) \begin{pmatrix} \beta_2 \\ \beta_1 \end{pmatrix}$$

  - Unknowns: β_1 and β_2

Texture Mapping

- Planar: Quadrilateral
  - Given an intersection P lying on a quadrilateral

  $P(u,v)=(1-u)(1-v)P_0 + u(1-v)P1 + uvP_2 + (1-u)vP_3$
  - Compute its texture coordinates (s,t) by solving the following linear system (for each coordinate):

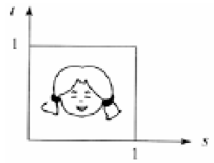
  $P=(1-u)(1-v)P_0 + u(1-v)P1 + uvP_2 + (1-u)vP_3$

  - Unknowns: u and v

Texture Mapping

Cylinder

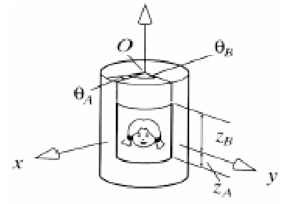
- P(x,y,z): point on the cylinder
- (s,t): texture coordinates



$$\theta \in [\theta_A, \theta_B],\ \ z \in [z_A, z_B]$$

$$x=r\cos\theta,\ y=r\sin\theta,\ z\in[z_A,z_B]$$

$$s=\frac{\theta-\theta_A}{\theta_B-\theta_A} \qquad t=\frac{z-z_A}{z_B-z_A}$$



Texture Mapping

Cylinder

- How to computeTexture coordinates
- Given an intersection P lying on a cylinder
- Compute its texture coordinates (s,t) as:

$$s=(a\cos(x/r)-\theta_A)/(\theta_B-\theta_A)$$

$$t=(z-z_A)/(z_B-z_A)$$

Texture Mapping

Sphere

- P=(x,y,z) on the sphere
- (s,t): texture coordinates

$$x=r\sin\theta\cos\varphi,\ \ y=r\sin\theta\sin\varphi,\ \ z=r\cos\theta$$

$$\theta\in[0,\pi],\varphi\in[0,2\pi] \qquad s=\frac{\theta}{\pi},\ t=\frac{\varphi}{2\pi}$$

Texture Mapping

Sphere

- How to compute Texture coordinates
- Given an intersection P lying on a sphere
- Compute its texture coordinates (s,t) as:

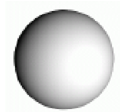
$$s = a\cos(z/r)/\pi$$

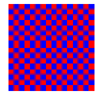
$$t = a\cos(x/(r\sin(\pi\, s)))/2\pi$$

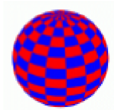
Texture Mapping & Illumination

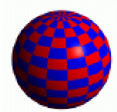
- Texture mapping can be used to alter some or all of the constants in the illumination equation:
  - pixel color, diffuse color, alter the normal, ….
- Classical texturing: diffuse color k_d changes over a surface and is given by a 2D texture which is an image

$$I_{local} = \sum_{i=0}^{nbLum} I_i \times \frac{vis(i)}{d_i^{\,2}} \times \left(k_d\left(\vec{N}\cdot\vec{L_i}\right) + k_s\left(\vec{R_i}\cdot\vec{V}\right)^n\right)$$


Constant Diffuse Color

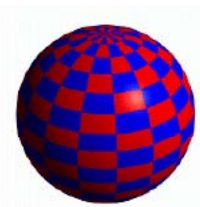

Diffuse Texture Color


Texture used as Label


Texture used as Diffuse Color

Bump Mapping

- Use textures to alter the surface normal
  - Does not change the actual shape of the surface
  - Just shaded as if it was a different shape

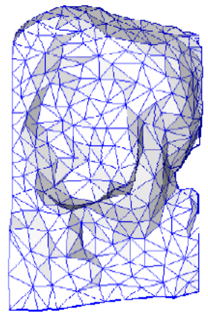

Sphere w/Diffuse Texture


Swirly Bump Map

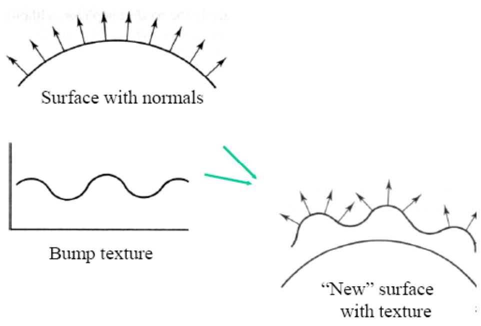

Sphere w/Diffuse Texture & Bump Map

Bump Mapping

- Add more realism to synthetic images without adding a lot of geometry



Bump Mapping



Bump Mapping

- Normal of bumped surface, so-called *perturbed* normal:
- Derivation can be found in "Simulation of Wrinkled Surfaces"

  James F. Blinn

  SIGGRAPH '78 Proceedings, pp. 286-292, 1978

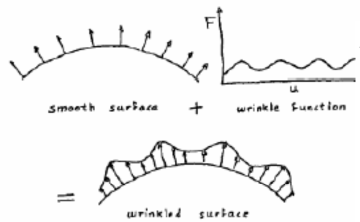
  (Pioneering paper...)
- Use texture to store either:
  - perturbed normal map
  - bump–map itself

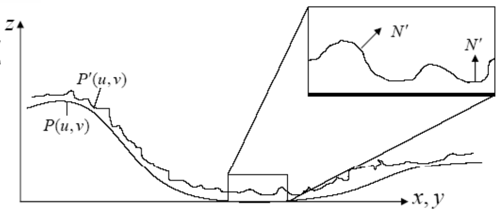
Bump Mapping

- The light at each point depends on the normal at that point.
- Take a smooth surface and perturb it with a function B.
- But we don't really perturb that surface (that is not displacement mapping).
- We modify the normals with the function B(u,v), measuring the displacement of the irregular surface compared to the ideal one.
- we are only shading it as if it was 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.

Bump Mapping

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 was deformed by the height function.





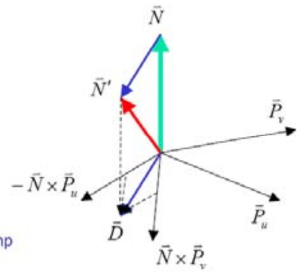
Bump mapping

General case

$\vec{P} = [x(u,v), y(u,v), z(u,v)]^T$ Initial point

$\vec{N} = \vec{P}_u \times \vec{P}_v$ Normal

$\vec{P}' = \vec{P} + \frac{B(u,v)\vec{N}}{\|\vec{N}\|}$ Simulated elevated point after bump



$$\vec{N}' \approx \vec{N} + \underbrace{\frac{B_u\left(\vec{N}\times\vec{P}_v\right) - B_v\left(\vec{N}\times\vec{P}_u\right)}{\|\vec{N}\|}}_{\vec{D}}$$

Variation of normal in u direction

$$B_u = \frac{B(s-\Delta,t) - B(s+\Delta,t)}{2\Delta}$$

$$B_v = \frac{B(s,t-\Delta) - B(s,t+\Delta)}{2\Delta}$$

Variation of normal in v direction

Compute bump map partials by numerical differentiation

Bump mapping derivation

$$\vec{P}' = \vec{P} + \frac{B(u,v)\vec{N}}{\|\vec{N}\|}$$

$$\vec{P}'_u = \vec{P}_u + \frac{B_u\vec{N}}{\|\vec{N}\|} + \cancel{\frac{B\vec{N}_u}{\|\vec{N}\|}} \approx 0$$

Assume B is very small...

$$\vec{P}'_v = \vec{P}_v + \frac{B_v\vec{N}}{\|\vec{N}\|} + \cancel{\frac{B\vec{N}_v}{\|\vec{N}\|}} \approx 0$$

$$\vec{N}' = \vec{P}'_u \times \vec{P}'_v$$

$$\vec{N}' \approx \vec{P}_u \times \vec{P}_v + \frac{B_u(\vec{N}\times\vec{P}_v)}{\|\vec{N}\|} + \frac{B_v(\vec{P}_u\times\vec{N})}{\|\vec{N}\|} + \frac{B_u B_v(\vec{N}\times\vec{N})}{\|\vec{N}\|^2}$$

But $\vec{P}_u \times \vec{P}_v = \vec{N}$, $\vec{P}_u \times \mathrm{N} = -\vec{N}\times\vec{P}_u$ and $\vec{N}\times\vec{N} = 0$ so

$$\vec{N}' \approx \vec{N} + \frac{B_u(\vec{N}\times\vec{P}_v)}{\|\vec{N}\|} - \frac{B_v(\vec{N}\times\vec{P}_u)}{\|\vec{N}\|}$$

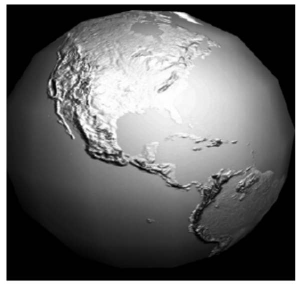
Bump Mapping

Choice of function B(u,v)

- Blinn has proposed various techniques:
- B(u,v) defined analytically as a polynomial with 2 variables or a Fourier serie (very expensive approach)
- B(u,v) defined by 2-entry table (poor results, requires large memory)
- B(u,v) defined by 2-entry table smaller and an interpolation is performed to find in-between values

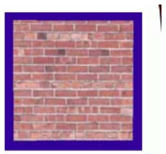
Bump Mapping

- Treat the texture as a single- valued height function
- Compute the normal from the partial derivatives in the texture



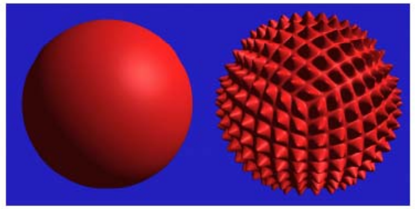
Bump Mapping

- There are no bumps on the silhouette of a bump-mapped object
- Bump maps don't allow self-occlusion or self-shadowing
- Problem solved with Displacement Mapping



Displacement Mapping

- Use the texture map to actually move the surface point along the normal to the intersected point.
- The geometry must be displaced before visibility is determined, which is different from bump mapping



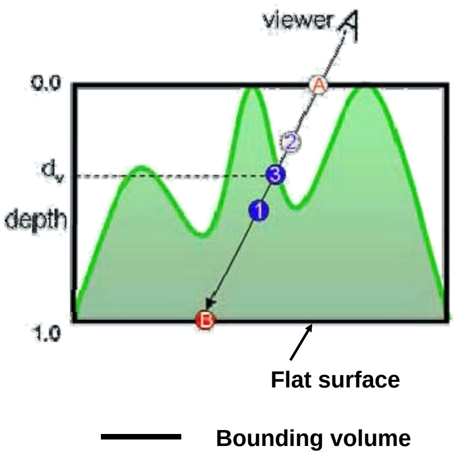
Displacement Mapping

•Compute intersection between ray and bounding volume

•Result: points A and B

•Height (or depth) is stored in a texture

•Use a search technique for the first intersection point: here point 3



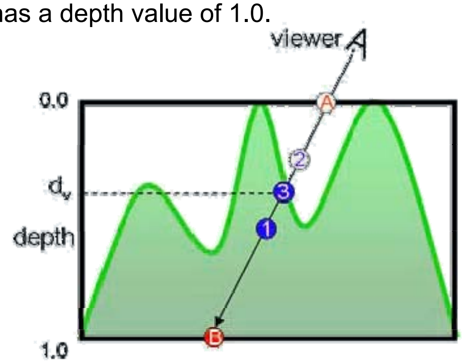
Displacement Mapping

- *A* has a depth vale of 0 and *B* has a depth value of 1.0.

•At each step, compute the midpoint of the current interval and assign it the average depth and texture coordinates of the end points. (used to access the depth map).

•If the stored depth is smaller than the computed value, the point along the ray is inside the height-field surface (point 1).

•In this case it takes three iterations to find the intersection between the height-field and the ray



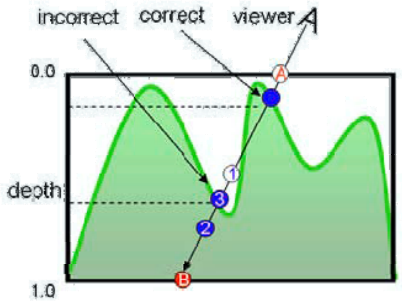
•However, the binary search may lead to incorrect results if the viewing ray intersects the height-field surface more than once

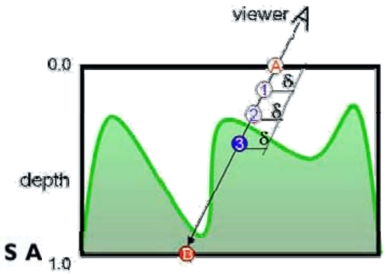
Displacement Mapping

•However, the binary search may lead to incorrect results if the viewing ray intersects the height-field surface more than once:

•In this situation, since the value computed at 1 is less than the value taken from the height-field, the search will continue down.

•In order to remedy this, the algorithm starts with a linear search

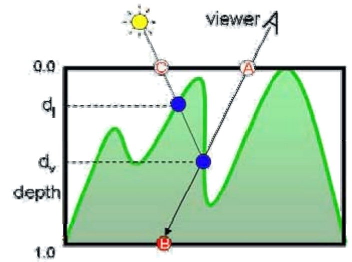




Displacement Mapping

- The technique can also handle surface self-shadowing:

•We must decide if the light ray intersects the height-field surface between point C and the point where the viewing ray first hits the surface.



Displacement Mapping

- Image from:
  Geometry Caching for
  Ray-Tracing Displacement Maps
- by Matt Pharr and Pat Hanrahan.
- *note the detailed shadows cast by the stones*

Displacement Mapping

- Bump Mapping combined with texture