

CS3500 Computer Graphics

Module: Lighting and Shading

P. J. Narayanan

Spring 2009

Lighting/Shading

- We know which pixels of the frame buffer belongs to which object after visibility and scan conversion.
- What colour to give to the pixel? “Current colour”??
- Depends on: the colour of the object, the material properties of the object, the colour of the light source, the angle of viewing with respect to object/lights, etc.
- **Lighting** and **shading**: Finding the colours for each pixel, perhaps after finding it on the extrema of the primitives.
- What is our guide? **Physics!**

Different Terms

- **Illumination Model:** How to “light” an object point given its material properties, the light sources, and the camera?
- **Shading Model:** How the illumination model applies to objects such as polygons and points.
- **Lighting:** Different types of lights.
- **Shadows:** How are shadows cast by objects.

Confessions of a Graphics System

- Physics of lights and their effects on objects and ultimately on the camera image are well understood.
- Simulating it exactly on the computer is possible.
But could be **very** time consuming!
- Use simplifying *tricks* or *hacks* or *kludges* that look almost right. It is, after all, easy to fool the human eye!
- Graphics systems have always relied on them!
- Enhanced hardware capabilities are bringing Lighting and Shading closer to physics.

Illumination Models

- Modelling of the interaction with light and an object point from the point of view of the camera image.
- Three factors come into play: light source properties, material properties, and atmospheric effects.
- Light sources emit light.
Properties: Colour and Directionality.
- Materials interact with light differently.
Properties: Reflectivity and Colour.
- Atmospheric effects: attenuation.

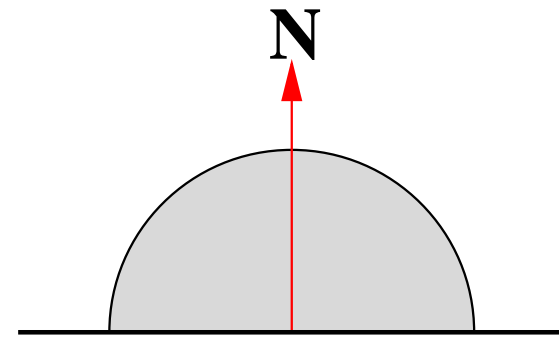
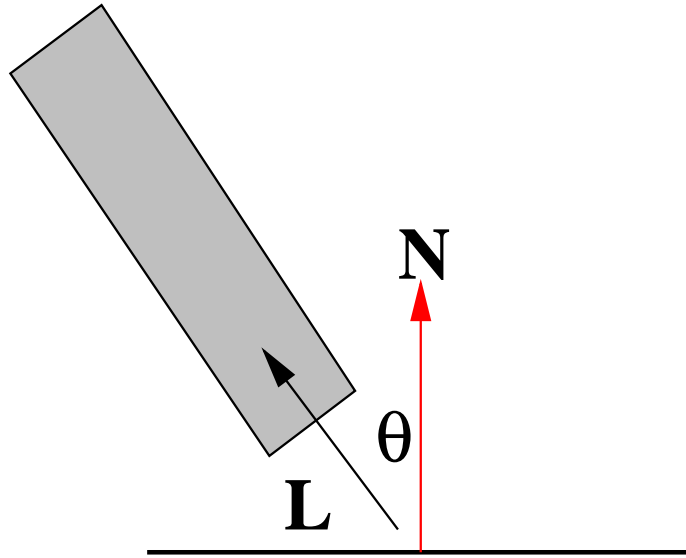
Material Properties

- **colour**: the paint used on the surface of the material with which the object is made.
- **Reflectivity**: how the material interacts with light. Shiny, dull, grainy, etc.
- **Diffuse reflection**: For dull, rough objects.
- **Specular reflection**: For smooth, shiny objects.

Diffuse or Lambertian Reflection

- Objects that obey Lambert's law of reflection: cloth, rough wall, etc.
- The normal component of light falling on it is absorbed by the object and is then reflected back equally in all directions.
- No preferred direction for reflecting light falling on it. Appearance of the point is independent of the view angle.
- Normal component of the light is proportional to $\cos \theta$.

Light Source



**Reflection in
all directions**

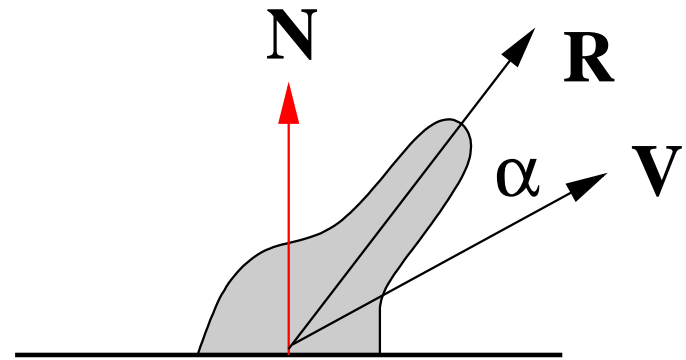
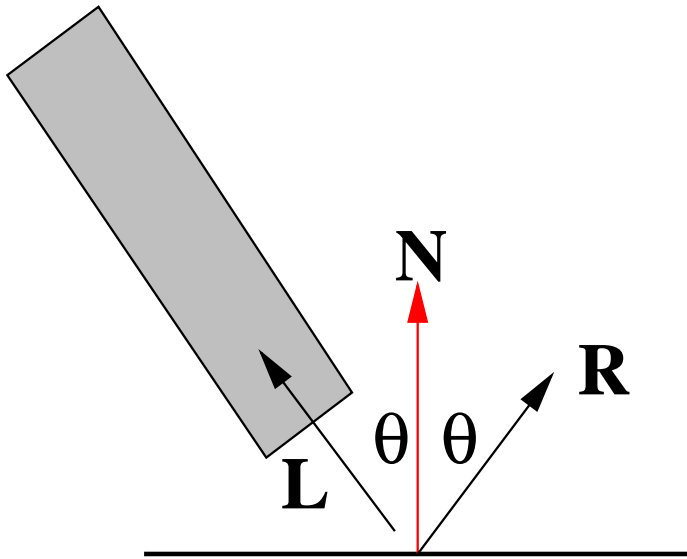
- Reflected light $I \propto I_L \cos \theta$ in all viewing directions.

Shiny Objects and Highlights

- Shiny objects (smooth metal, polished marble) behave differently.
- A *highlight* can be seen on them due to the light source. Highlight has the colour of the light source, irrespective of object colour.
- Highlight moves with the viewing angle. Appearance of the object point depends on the view angle.
- This is called **specular reflection**.

Specular Reflections

- A distinguished direction exists for reflection, depending on the incident light direction and normal direction.
- Reflection falls off quickly as view moves away from this angle.
- A mirror is an ideal specular reflector.
- Reflect the light vector about the normal vector to get the specular reflection direction.



**Maximum Reflection
along direction R**

Diffuse Illumination Equation

- The formula for computing the intensity at a point.
- For diffuse reflection, if I_p is the light falling at the surface,
$$I_d = I_p k_d \cos \theta = I_p k_d (\mathbf{N} \cdot \mathbf{L})$$
- k_d is the diffuse reflection coefficient.
- θ should be between 0 and 90. Otherwise, the light has no effect. (Object is self-occluding!)
- For correct effects, the real normal at the point is necessary. (Normals are coming of use finally!)

Material Colour

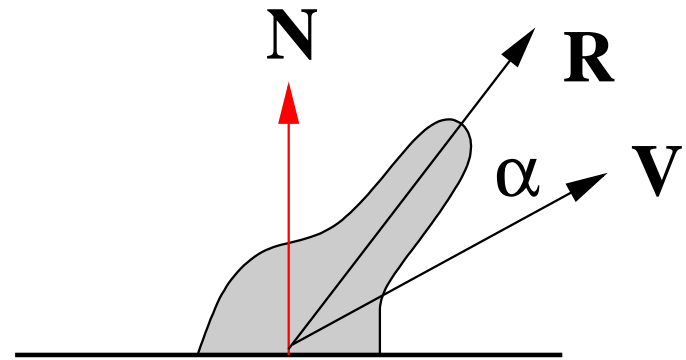
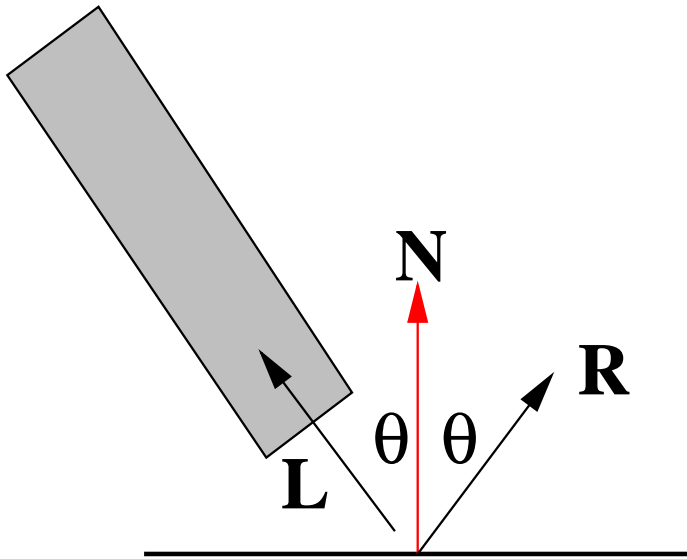
- What is the colour of an object? The light it reflects. Rest is absorbed.
- A fully red object has colour (1, 0, 0). It reflects all of the red falling on it and none of the green or blue.
- How is that done?
- We also need to know the spectral composition of the light falling on the object.
- Illumination equation: $I_{d\lambda} = I_{p\lambda} k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L})$
- $k_d O_{d\lambda}$ can be called the **diffuse colour** of object.

Ambient Light

- In graphics, some light is always present.
- This is called **ambient light** I_a , present everywhere.
- The net effect of all the light that reflects from the environment etc.
- This light helps see objects even when no explicit light source is present.
- Illumination equation: $I_\lambda = I_{a\lambda} k_a O_{a\lambda} + I_{p\lambda} k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L})$
- k_a is the ambient reflection coefficient.

Atmospheric Effects

- Most straightforward is attenuation.
- The light that reaches the point $I = f_{\text{att}} I_p$.
- Physics says: $f_{\text{att}} = 1/d_L^2$ by inverse square law.
- In practice, it doesn't work well. Objects that are far become indistinguishable.
- In graphics, we use $f_{\text{att}} = 1/(c_1 + c_2 d_L + c_2 d_L^2)$
- Illum Equn: $I_\lambda = I_{a\lambda} k_a O_{a\lambda} + f_{\text{att}} I_{p\lambda} k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L})$



**Maximum Reflection
along direction R**

Specular Lighting: Phong Model

- α is the angle between the reflection direction R and the view direction V .
- Phong's model: $I_s = I_p k_s \cos^n \alpha = I_p k_s (\mathbf{V} \cdot \mathbf{R})^n$
- As n becomes larger, reflection becomes sharper. k_s is the specular reflection coefficient.
- Sometimes, specular colour $O_{s\lambda}$ is also given to the object!

Multiple Light Sources

- Total illumination equation for one light source:

$$I_{\lambda} = I_{a\lambda} k_a O_{a\lambda} + f_{\text{att}} I_{p\lambda} [k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L}) + k_s O_{s\lambda} (\mathbf{V} \cdot \mathbf{R})^n]$$

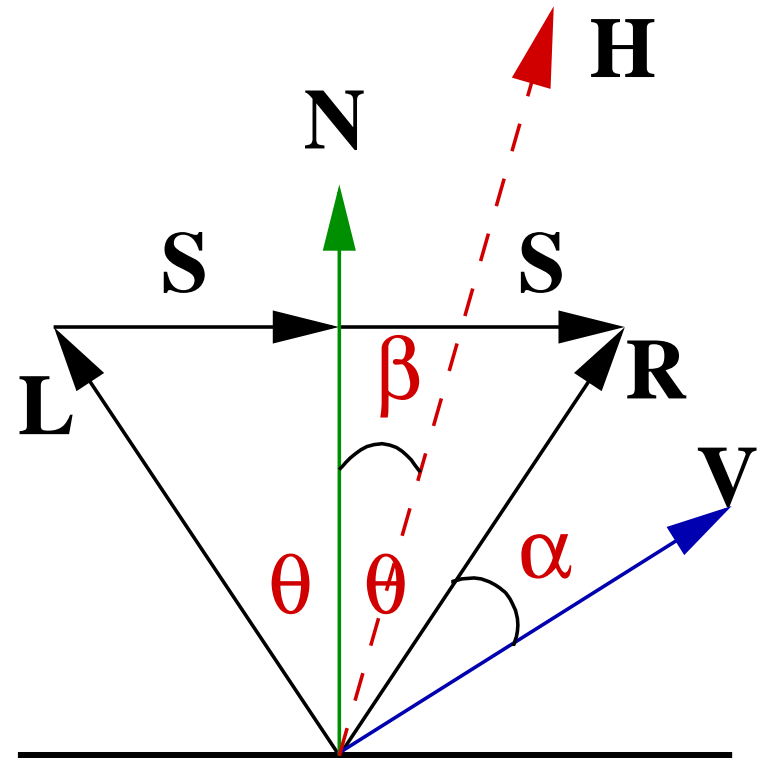
- When multiple light sources are involved:

$$I_{\lambda} = I_{a\lambda} k_a O_{a\lambda} + \sum_i f_{\text{att}_i} I_{p\lambda i} [k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L}_i) + k_s O_{s\lambda} (\mathbf{V} \cdot \mathbf{R}_i)^n]$$

- Contributions of ambient, diffuse, and specular reflections are added together.
- Contributions of multiple light sources are added together.

Computing Reflection Vector R

- $\mathbf{L} + \mathbf{S} = \mathbf{N} (\mathbf{N} \cdot \mathbf{L})$
- $\mathbf{R} = \mathbf{L} + 2\mathbf{S} = ??$
- The halfway vector \mathbf{H} may also be used: $\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|}$.
- Maximum highlight when \mathbf{H} and \mathbf{N} coincide.
- Deviation from it $\mathbf{N} \cdot \mathbf{H}$ can be used instead of $\mathbf{V} \cdot \mathbf{R}$.



Lighting: Summary

- **Material:** Diffuse colour, specular colour, ambient colour.
- **Light Source:** Type, directionality and colour. Usually, diffuse, specular, and ambient colours could be attached to light sources.
- **Reflection:** Diffuse and Specular.
- Total illumination equation for multiple light sources:

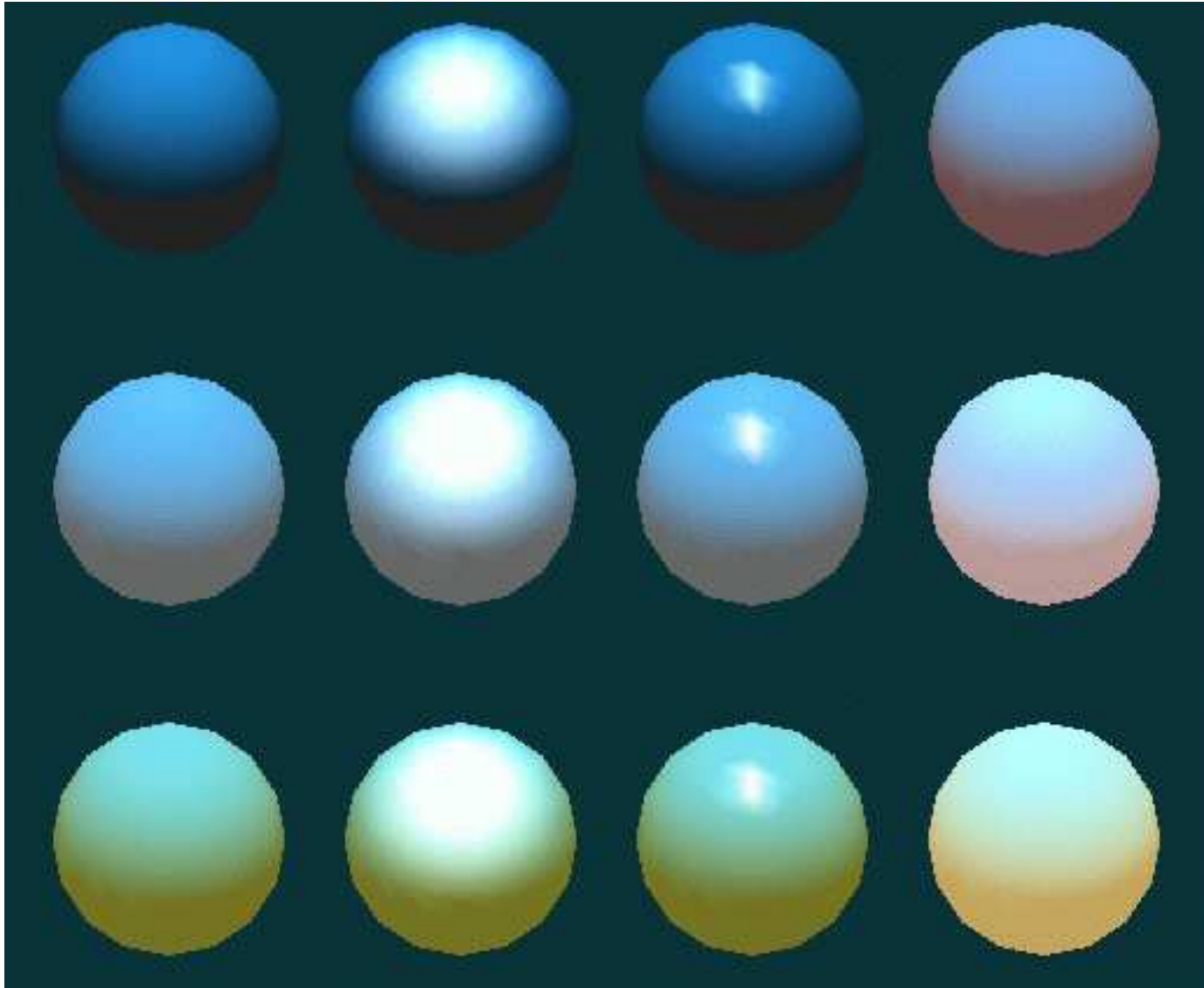
$$I_{\lambda} = I_{a\lambda} k_a O_{a\lambda} + \sum_i f_{\text{att}_i} I_{p\lambda i} [k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L}_i) + k_s O_{s\lambda} (\mathbf{V} \cdot \mathbf{R}_i)^n]$$

Emissive Colour

- Objects that emit colour such as a tubelight as an object.
- Material property can include emissive colours for such self luminous objects.
- The emissive colour is added to every point of the object.
- Only the appearance of the object is affected.
- Emissive objects do not work as light sources automatically.

Material in OpenGL

- **colours**: the material colour times the appropriate reflection coefficient.
- Ambient, Diffuse, Specular, Emissive colours.
- Shininess: like the n in the $\cos^n \alpha$ term.
- Produces good effects when combined with light sources.
- `glMaterial()` changes the current material properties.
Read!



Light Sources

- When the light source is infinitely far (like the Sun), only the direction matters. *Directional Light*
- When light has a position, **L** vector can be computed from it. *Point Light Source*
- Point light sources illuminate in all directions.
- Alternately, light source can have a position, a direction, and a drop off formula. Lighting is maximum in the given direction and drops off as you move away from that direction. *Spot Light*
- Colour of the light is important to compute effects.

Light Sources in OpenGL

- Position: Set $(x, y, z, 0)$ for directional light sources, located at ∞ . Finite position for others. Default: point light source.
- Each has: Ambient, Diffuse, Specular colours. Spot direction, cut-off, exponent. Attenuation.
- Since each polygon is drawn independently, shadows do not appear automatically.
- Read about `glLight()`, `glLightModel()`.
And `glEnable()` for `GL_LIGHTi`, `GL_LIGHTING`.

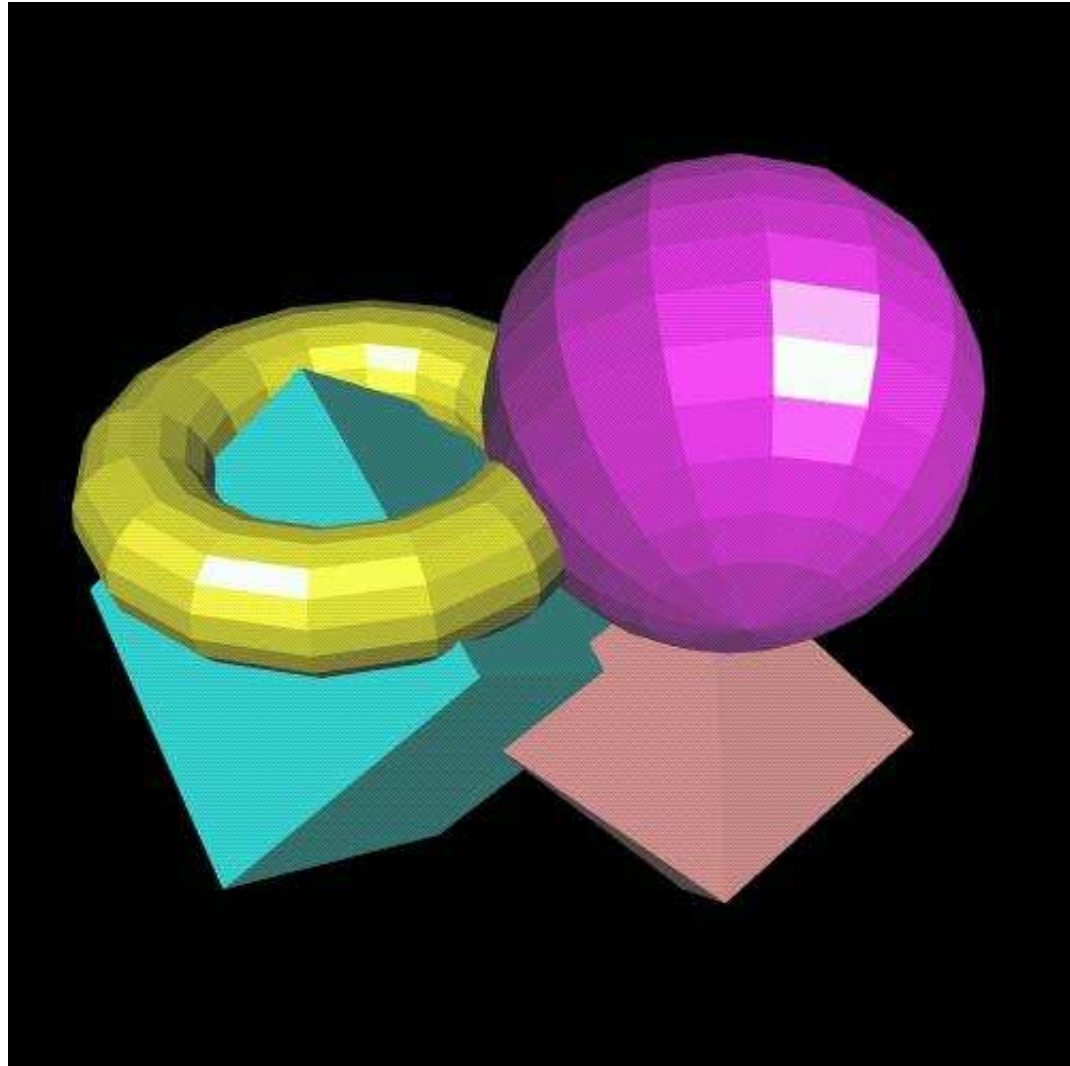
Shading Models for Polygons

- The illumination equation can be evaluated at every pixel to compute the colour/intensity there.
- This is expensive computationally.
(Does it give the correct results?)
- Can we take advantage of the coherence or the fact that we are computing for a planar polygon?
- A number of different options exist.

Constant or Flat Shading

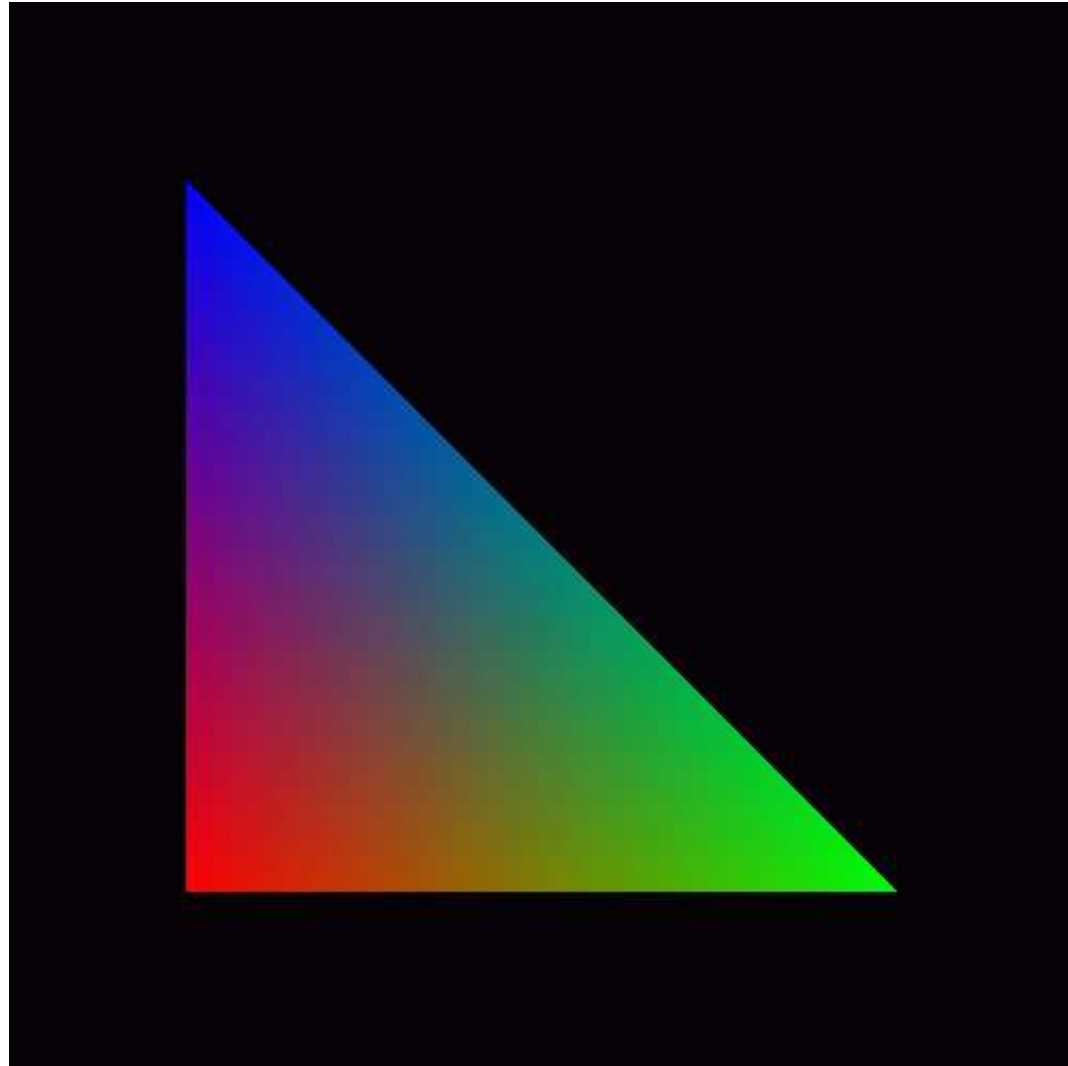
- Evaluate the illumination equation only once for the polygon.
- Apply the results (the intensity values) to the whole polygon.
- Each polygon gets a constant colour. They look flat.
- Most easy computationally.

- The results will be correct if:
 - $\mathbf{N} \cdot \mathbf{L}$ is constant across the polygon and
 - $\mathbf{N} \cdot \mathbf{V}$ is constant across the polygon and
 - Object is polyhedral, not an approximation.
- Use which point? Centre? First vertex? Results vary.



Interpolated Shading

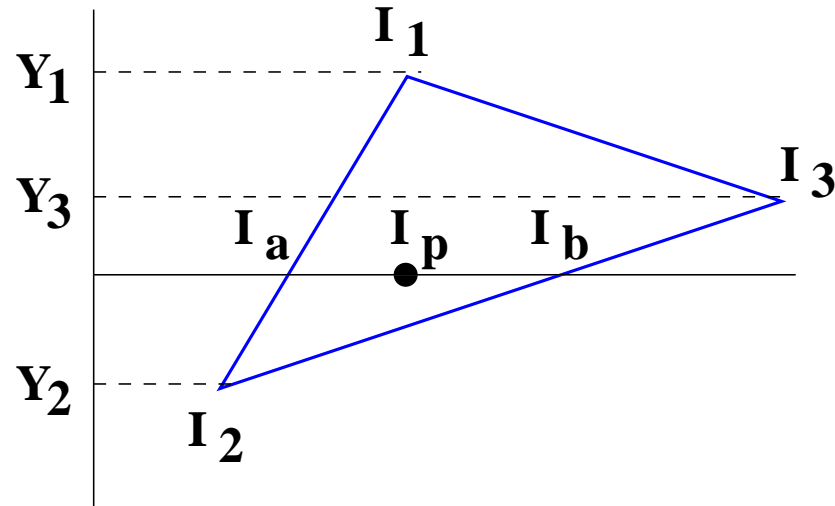
- Results of flat shading are unsatisfactory.
- Interpolated shading assumes that properties can be calculated at the vertices and can be interpolated for the interior points.
- The interpolation can be done along with the interpolation of the Z values.
- Use the normal, light and view vectors for the vertices to compute the values.
- Since the polygon is planar, the normals should be the same!?!?



Gourard Shading

- Also called *colour interpolation shading*.
- Evaluate the illumination equation at each of the polygon vertices.
- Use exact normals – stored already or computed on the fly – if available.
- Otherwise, use the average of all polygons that meet at the vertex!
- Interpolate intensities along the edges that connect vertices.

- Interpolate along scan lines using intensities at the edges. This can be combined nicely with the computation we perform on spans of polygons nicely.
- Produces good results, not correct according to Physics!



Highlights under Interpolation

- Specular highlights are localized bright areas.
- If the highlight falls in the middle of a polygon, it will be completely missed as illumination equation is computed only for the vertices.
- If highlight falls on a vertex, it will be interpolated across, making it less local.
- These cannot be handled by interpolation of intensities.

Polygon Approximation

- When a curved object is approximated using a polygon, adjacent polygons could have different intensity values.
- Flat shading will bring it out sharply; object will appear *faceted*.
- Interpolated shading may not help much if neighbouring polygons have different normals.
- Will it help to evaluate the illumination equation at each pixel?

Normals at Points

- If a parametric or analytic representation of the object that is approximated is available, exact normals can be computed at each point.
- Keep the equations along with the objects so that exact normals can be computed.
- Alternately, compute the normals for each vertex when converting the smooth object to a polygon mesh.
- Keeping the normal vector at each vertex along with the 3D coordinates is quite popular.

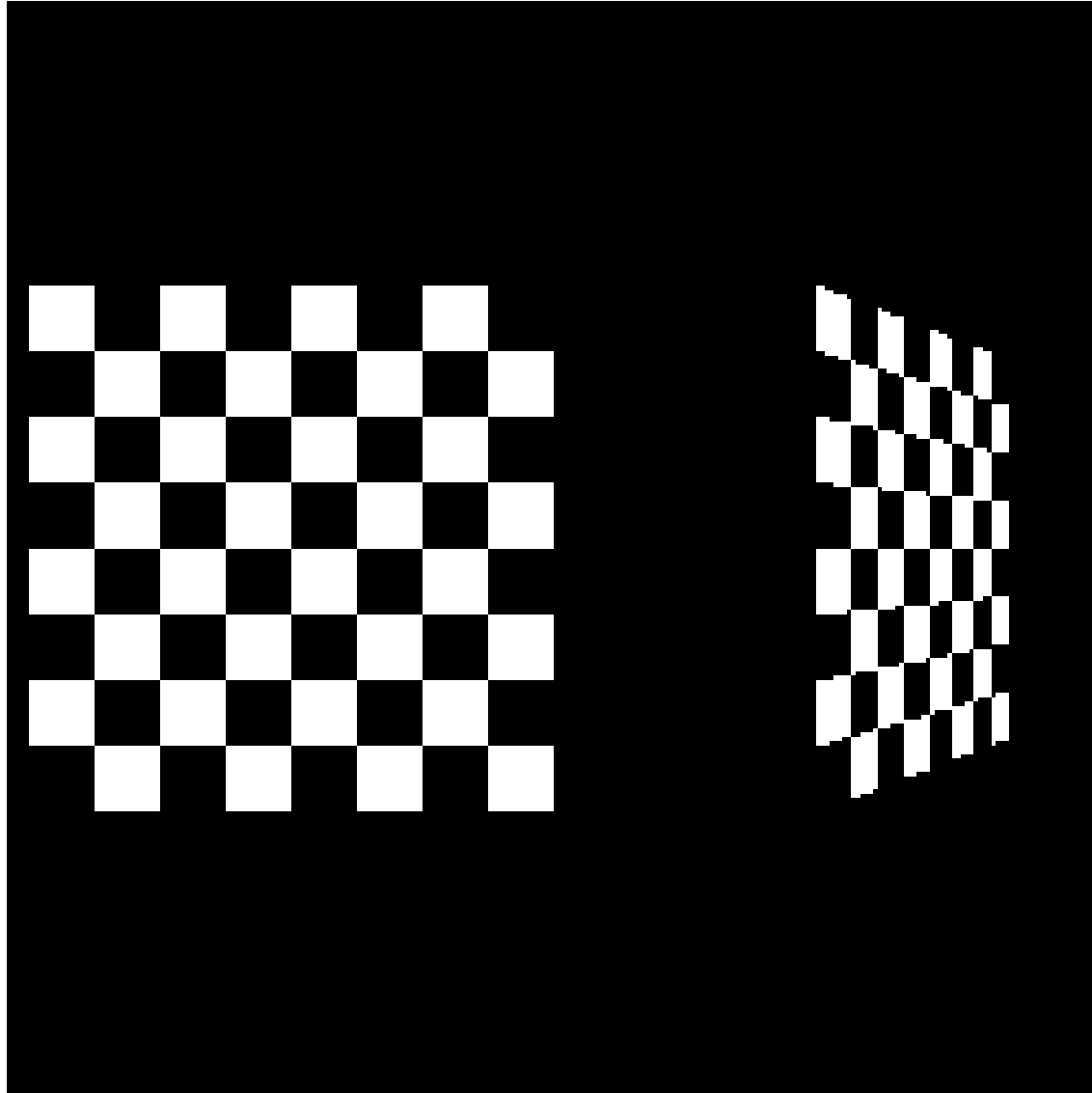
Phong Shading

- Interpolate surface normals across the polygon, given the normals at the vertices.
- Compute illumination equation with these interpolated normals for each pixel.
- Computes highlights very well.
- Computation time is more as the equation is evaluated at every pixel.
- Interpolation of normals is not physically based!
- Also called *normal vector interpolation shading*

Texture Mapping

- Shading can produce only uniform or smooth surfaces. They look plastic and artificial.
- Map a real or synthetic image onto a polygon surface.
- Each 3D point is also associated with a 2D texture coordinate pair.
- These refer to points in the image to associate with the points.

- While scan converting the polygon, the pixel coordinates are mapped to the texture image coordinates.
- The texture colour for the pixel is obtained by interpolating the texture image using these coordinates.
- Finally, a texture colour is obtained for the pixel.
- This can overwrite the surface, modulate it, blend it, etc.



Surface Detail

- Shading can produce only uniform or smooth surfaces
They look plastic and artificial.
- Surface-Detail Polygons: Special polygons that are coplanar with a base polygon. They are marked specially.
- Surface detail polygons can be used to draw a window on a planar wall.
- VSD is done using base polygons, ignoring surface detail.
- Shading is done on surface detail polygons.



CS3500

Lighting/Shading

Bump Mapping

- Surfaces look absolutely smooth even with textures.
- Displace surface points along the normal slightly to give it a rough look.
- A **bump map** gives the values by which the points have to be displaced along the normal.
- The displaced normals can be approximated easily and used in the lighting calculations.
- Produces nice results.

Transparency

- How do we deal with transparent objects?
- **Ignore them!** Leads to non-refractive transparency, which is still OK.
- Refraction takes considerable effort to handle. We will see it later.
- Other tricks: Interpolated transparency and Filtered transparency.

Interpolated Transparency

- When two objects overlap at a pixel, use a weighted average of the two colours at the pixel.
- $I_\lambda = (1 - k_{t1}) I_{\lambda1} + k_{t1} I_{\lambda2}$
- k gives the transmission coefficient or the transparency of each polygon. If $k_{t1} = 0$, polygon 1 is totally opaque.
- $(1 - k)$ is opacity.

RGBA Colours

- Graphics systems (OpenGL, for example) use 4-component colours, with A being the α channel.
- Alpha measures the opacity of the colour. Equals 1 for completely opaque objects and 0 for totally transparent ones.
- A similar interpolation formula is used when an object with alpha is drawn on top of another object.
- If the Z-buffer test succeeds, the colours of the new polygon are blended into the frame buffer using α instead of replacing it entirely.

Filtered Transparency

- Object is a transparent filter with colour.
- $I_\lambda = I_{\lambda 1} + k_{t1} O_{t\lambda} I_{\lambda 2}$
- $O_{t\lambda}$ is the object's *transparency colour*.
- $I_{\lambda 2}$ is the colour stored in the frame buffer, which could have had other transparent objects.

Screen-Door Transparency

- We need to perform extra computations at every pixel for interpolated transparency.
- Can we achieve 50% transparency by showing the front object at half the pixels and the back object at others?
- Yes, as the human eye is integrating it spatially and seeing both!
- This is called *screen-door transparency* as the back object is visible through a screen door.
- A 2-D bit mask has 1's for the pixels where the new object should be shown and 0's for others.

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	1
1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	1
1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	1
1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	1

- Possible masks for 50% and 75% transparency.
- Different combination of bit patterns will have slightly different effects.

Implementing Transparency

- List priority, back-to-front algorithms: Need capability to read framebuffer values back.
- Screen-door transparency with Z-buffer: Transparent/opaque objects can be rendered in any order.
- Otherwise transparency is difficult with Z-buffering.
- One idea: Render all opaque objects first normally. Follow this with rendering all transparent objects. Colour is blended with transparency; z -values are not changed.

- Produces reasonable but incorrect results. Depth ordering among transparent objects is lost.
- Correct rendering: Find the farthest transparent object, render it. Repeat this till all objects are exhausted!
- Computationally expensive.

Interobject Reflections

- Shiny objects reflect the environment. Mirrors are extreme examples.
- Called *Environment Mapping* or *Reflection Mapping*.
- Find a centre of reflection and surface of reflection.
- Render the scene with the camera at a chosen *centre of reflection* onto a chosen *surface of reflection*.
- Use the image as a texture map onto the surface of reflection.

- In practice, the environment is mapped to a sphere. Sections of this image are used as texture to map to the surface.
- Alternately, projection of the environment to the 6 sides of a cube can be used.
- Provides approximate solutions, not exact ones.

Shadows

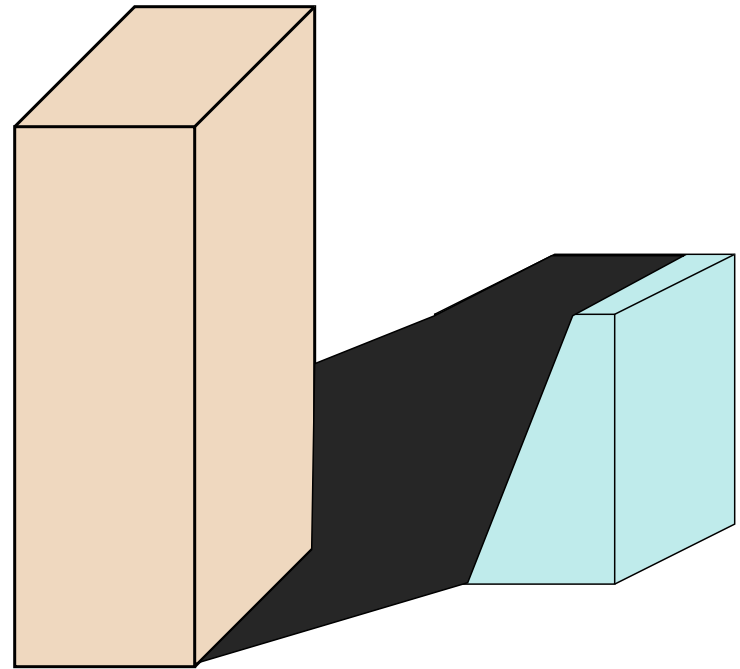
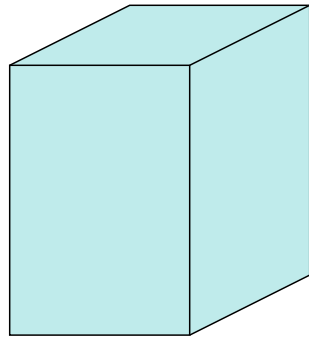
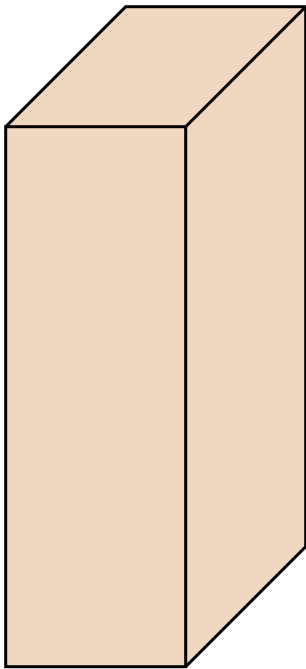
- Shadows are the results of light **not** reaching some part of the scene.
- Surfaces that are not visible from the light sources are in shadow. VSD algorithms can be used to determine this.
- **Shadow Map:** A bitmap in image space containing projections of shadowed regions.

- Modify the illumination equation to:

$$I_\lambda = I_{a\lambda} k_a O_{a\lambda} + \sum_{1 \leq i \leq m} S_i f_{\text{att}_i} I_{p\lambda i} \{ k_d O_{d\lambda} (\mathbf{N} \cdot \mathbf{L}_i) + k_s O_{s\lambda} (\mathbf{V} \cdot \mathbf{R}_i)^n \}$$

- Shadow map S_i indicates if the pixel is under the shadow for the light source i . $S_i = 0$ if under shadow, $S_i = 1$ otherwise.

Object Precision Algorithm



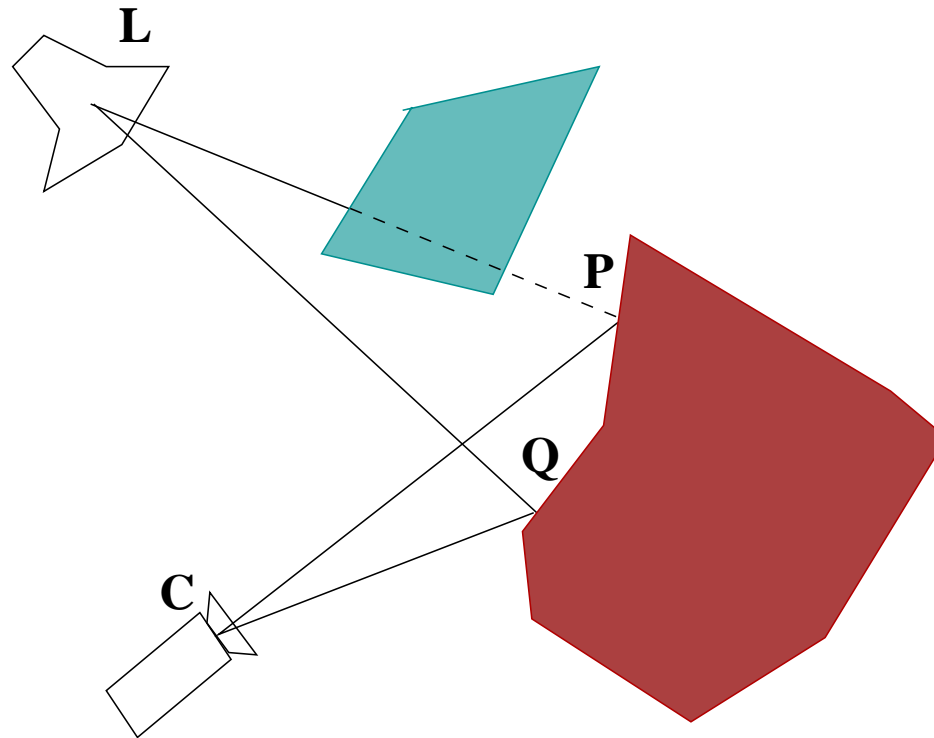
2-Pass Object Precision Algorithm

- Transform objects to light source's point of view.
- Find visible surfaces by splitting polygons. These will be the lit polygons.
- Transform visible polygons to ORC (Object Reference Coordinates).
- Make the lit polygons surface detail polygons on the base polygons.

- These can be done as pre-processing as the shadows will not change if lights and objects don't move.
- Transform objects to camera coordinates, perform VSD and draw lit polygons.

2-Pass Z-buffer Shadow Algorithm

- Draw the scene with light source as the camera using Z-buffering.
- Read the depth values from the Z-buffer into a z_l buffer. This is the *Shadow Map*.
- Draw scene from camera's viewpoint with Z-buffering.
- Transform each visible point (x_o, y_o, z_o) to (x'_o, y'_o, z'_o) in the light source's coordinates.
- If $z'_o > z_l(x'_o, y'_o)$, there is another point that is closer to the light source. Do not light the pixel.



- Point P will be shadowed and Q will not be.

Global Illumination

- Our calculations include only light that comes from a source to the point and reflected from it.
- Indirect reflections between objects, refraction, etc., are not handled.
- Only *local illumination* is used. *Global illumination* takes into account all light that falls on the object.
- Two techniques: **Ray Tracing** and **Radiosity Algorithms**.

Ray Tracing

- Perform exactly what our image-precision algorithm described.
 - **for each pixel in the image**
 - Determine closest object in the direction of projector
 - Draw the pixel with appropriate colours
- Send rays from CoP through each image pixel to the world.
- Called **ray tracing** or **ray casting**.
- Equation of the ray is known. Need to intersect it with objects in the world.

Ray Equation

- If the CoP is (x_0, y_0, z_0) and pixel point is (x_1, y_1, z_1) , the ray is given by $(x_0 + t\Delta x, y_0 + t\Delta y, z_0 + t\Delta z)$, $t > 0$
- Negative values of t line behind CoP. $t = 1$ is at the projection or pixel plane.
- If the scene is in front of the plane, the region of interest is $t > 1$.
- Compute intersections with other objects. Closest object is the one with the smallest t value.

Intersection with Polygons

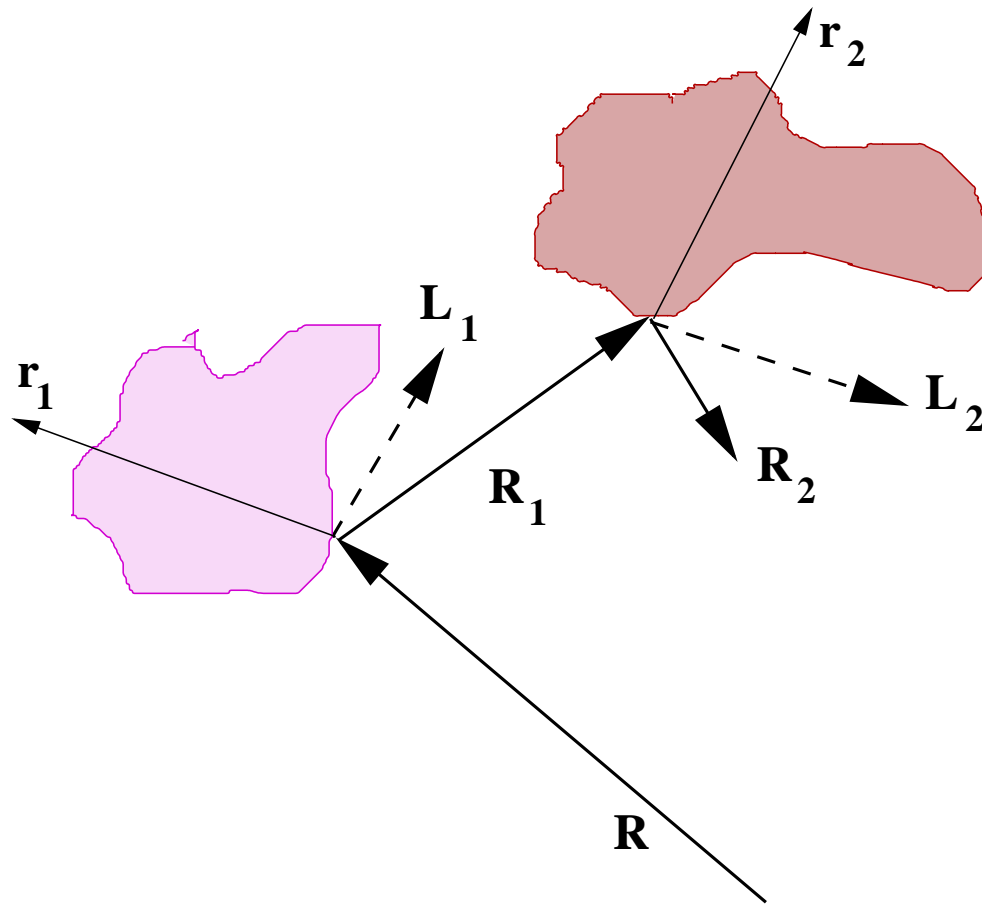
- Plane of the polygon is given by $Ax + By + Cz + D = 0$
- Intersection point: $t = -\frac{Ax_0 + By_0 + Cz_0 + D}{A\Delta x + B\Delta y + C\Delta z}$
- Does it lie within the polygon?
- Project to a coordinate plane and check for 2D polygon containment.
- Use the plane with largest area. This is determined by the largest absolute value of A, B, C .

Intersection with a Sphere

- Sphere is given by $(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2$.
- Substituting, we get: $(\Delta x^2 + \Delta y^2 + \Delta z^2) t^2 + 2[\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] t + (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0$
- A quadratic equation. Solve for t . Real solution with smaller positive t is the one of interest.
- Can normalize such that the coefficient of t^2 is 1, since we are interested only in the relative values of t .

Recursive Ray Tracing

- When a ray from CoP through a pixel hits an object, it can
 - Get reflected off the surface about the normal
 - Get transmitted into the object as per Snell's law of refraction
 - Get light from all light sources by diffuse reflections
- Each of these *secondary rays* can bring in a colour/intensity by recursively applying the above principle.
- Net appearance is a combination of the individual colours.



- Rays will be spawned from every point.

Main Algorithm

- Call the recursive ray tracing routine for every pixel to compute its colour.

for each scan-line do

for each pixel in scan line do

determine the ray for the pixel

pixelColour \leftarrow RT_Trace(ray, 1)

Ray Tracing Algorithm

- Intersect ray with closest object and compute colour using a shading routine

RT_Trace(ray, depth)

Find the closest object for the ray

if (object found)

 compute normal at intersection point

 return RT_Shade(obj, ray, intersect, normal, dpth)

else

 return BackgroundColour

Shading Algorithm

- Combine all effects to compute colour

```
RT_Shade(obj, ray, pnt, n, d)
```

```
  clr = ambient term
```

```
  for each light L do
```

```
    IRay = ray to light from pnt
```

```
    Compute how much light reaches pnt from L
```

```
    clr +=  $k_d$  * diffuse component due to L
```

```
  if (d >= maxDepth) return clr
```

```
  // Onto recursive processing now
```

```
if (object is reflective)
    rRay = reflected ray from pnt
    rClr = RT_Trace(rRay, d + 1)
    clr +=  $k_s$  * rClr
if (object is transparent)
    tRay = refracted ray from pnt
    if (no total internal reflection)
        tClr = RT_Trace(tRay, d + 1)
        clr +=  $k_t$  * tClr
return clr
```

Ray Tracing: Discussion

- Ray tracing is very good to compute specular effects.
- It is very compute intensive as the ray tree can grow exponentially with spawning of new rays.
- It is subject to numerical precision as small changes in secondary and tertiary rays can have large impact.
- Several simplifications: Trace a set of rays (beams, cones, pencils) to take advantage of coherence, stochastic sampling to reduce aliasing effects, etc.
- Used today when really high quality rendered images are required at the expense of time.

Radiosity Algorithms

- Ray tracing is a view-dependent global illumination algorithm.
- Can we model the light that leaves each surface patch independently, in a view-independent manner?

$$B_i = E_i + \rho_i \sum_{1 \leq j \leq n} B_j F_{ij}$$

- Light leaving a patch – **radiosity** – is the light emitted by it plus the light that comes to it from other patches that is reflected.

- *Form factor* F_{ij} gives the fraction of the energy that leaves patch j that arrives at the patch i .
- Light sources can be treated as regular objects with high emissive components.

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

- The above equation needs to be solved for B_i s iteratively.

Radiosity: Discussion

- Radiosity methods are excellent for diffuse environments. Specularity is not handled very well.
- View-independent radiosity can be rendered from any point of view using a normal VSD algorithm.
- Good results depend on the fine-ness of the patches.
- Iterative procedure, hence computation intensive.