

Where we are

- So far:
 - Can model the world using the modelling transform.
 - Can move the camera around.
 - Can model perspective.
 - Can work out which parts of the world we can see.
 - Can make it all happen faster.
 - What's left?

And now ...

- Lighting and colouring ... how to decide colour of pixels
 - Ambient lighting
 - Combining diffuse and ambient
 - Multiple light sources
 - Specular colour model
 - Light attenuation with distance
- Applying local illumination model in practice

Lighting & Illumination

- Now we've worked out what to show, how to fill in polygons and more ...
- We can use computer to automatically work out colour of things, based on things like:
 - Light sources.
 - Material colour.
 - Textures of material (bumpy/smooth).
 - Reflection and refraction.

Plan ...

- Discuss local illumination models.
- Start with simple model and gradually add more and more to it.
- Apply illumination models to polygons, called *polygon rendering*.
- Apply illumination models to arbitrary geometrical objects, working out how light gets to the eye, called *ray tracing*.

Assumptions

- We assume that the objects we want to draw have intrinsic properties called *material properties*, indicated by a $k_{\text{(something)}}$.
- We are trying to work out the intensity I , the colour that gets drawn on the screen.

Simplest illumination model

- Each object simply has a defined colour.

$$I = k_i$$

- No lighting or anything.
- I = intensity that gets drawn on the screen
- k_i = inherent colour of object.
- Also remember: repeat once for each object.
- This is what `glColor3f(...)` really does

In OpenGL

- Really, this is what we have been doing in 2D.
- In OpenGL, material properties are set using `glMaterial(face, property, values)`
- `face` can be `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`.
- Material settings are stateful; i.e. material setting applies to everything that follows until changed.

Ambient illumination model

- Adds "background light"
- Diffuse, non-directional light source, as a result of many different reflections from the environment.

$$I = I_a k_a$$

where

I_a = ambient light intensity

k_a = ambient reflection coefficient

Colour?

- Repeat for each of red, green and blue.

In OpenGL

- Setting I_a :

```
float [] ambient_light =  
{0.4f, 0.4f, 0.4f, 1.0f};  
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,  
ambient_light);
```
- Setting k_a :

```
float [] ambient_mat =  
{0.8f, 0.2f, 0.2f, 1.0f};  
glMaterialfv(GL_FRONT, GL_AMBIENT,  
ambient_mat);
```

Diffuse illumination model

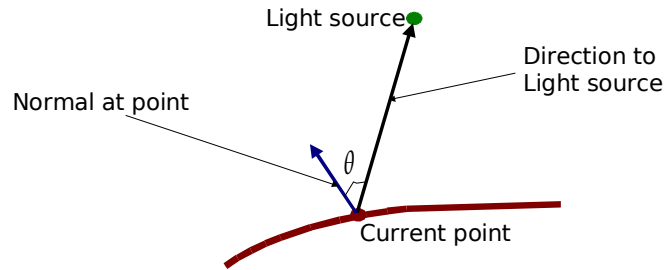
- Now we add point light sources.
- Point light source:
 - Light is emitted from a single point in space.
 - Light is emitted equally in all directions from that single point.
 - Light source has a brightness I_p .

Diffuse illumination model - contd

- Works well for matte, non-reflective surfaces.
- Each surface has a *diffuse reflection coefficient* k_d .
- Matte surfaces reflect light equally in all directions - so same colour from all directions.
- Does not depend where the person is standing.
- **Does** depend on the direction to the light source.

What does "direction to light source" mean?

- Direction to light source = difference between current point and the light source.

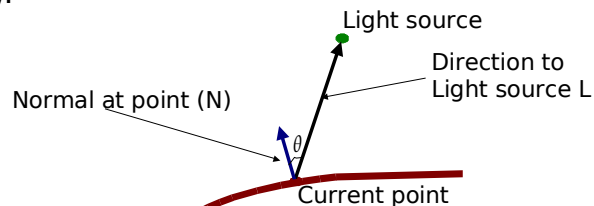


Why does colour depend on direction to light source?

- Think of amount of light covering an area.
- Or think of a rough sphere.
- What we know:
 - Object is brightest when the light source is directly on top of the normal at that point.
 - Object is darkest at 90 degrees or more (remember back face culling?)
- What function has this pattern?

Lambert's law

- Yes, it really is called Lambert's law.
- "Intensity of light reflected from a surface is proportional to the cosine of the angle between L (vector to light source) and N (normal at the point)."



The equation

- Hence equation is:

$$I_d = I_l k_d \cos(\theta)$$

where

I_l = intensity of light source

k_d = diffuse reflection coefficient

θ = angle between N and L

But wait ...

- cos of angle between vectors is easy...
- So, if N and L are normalised, then this is:

$$I_d = I_l k_d \vec{N} \cdot \vec{L}$$

where

I_l = intensity of light source

k_d = diffuse reflection coefficient

\vec{N} = Normal at point

\vec{L} = Vector from point to light source

Combining the two

- We usually have ambient light as well as diffuse reflection.
- So now combine ambient and diffuse light equations:

$$I = I_a k_a + I_l k_d \vec{N} \cdot \vec{L}$$

Multiple light sources?

- What if we have more than 1 light source?
- Each light source has its own intensity. Can simply sum the effects.

$$I = I_a k_a + \sum_{l_k=l_1..l_n} I_{l_k} k_d \vec{N} \cdot \vec{L}_k$$

where

$l_1..l_n$ are the point light sources

\vec{L}_k = Vector from point to light source k

I_{l_k} = Intensity of light source k

Lighting in OpenGL

- OpenGL supports at least 8 light sources
- One constant for each light source: `GL_LIGHT0`, `GL_LIGHT1`, ... `GL_LIGHT7`.
- `GL_LIGHT1 = GL_LIGHT0+1`
- Light properties set using `glLight(light, property, value)`
- Many different properties, including ambient and diffuse intensities.

Typical lighting commands

```
float[] lightpos =
{0f, 0.0f, 4.0f, 1.0f};
float[] lightmat =
{0.8f, 0.8f, 0.8f, 1.0f};
glLightfv(GL_LIGHT0, GL_POSITION,
lightpos);
glLightfv(GL_LIGHT0, GL_DIFFUSE,
lightmat);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

Diffuse Lighting in OpenGL

- Setting k_d :

```
float [] diffuse_mat =
{0.8f, 0.2f, 0.2f, 1.0f};
glMaterialfv(GL_FRONT, GL_DIFFUSE,
diffuse_mat);
```

Position vector

- Light position is treated as a vector of 4 values.
- If $w=1$, treated as a point light source (i.e. a point).
- If $w=0$, treated as a directional light source (i.e. because it's a vector!)

Question: 4-tuples for colour?

- Why are we using 4-tuples for colour?
- Answer: OpenGL supports alpha channel
- Alpha channel controls how transparent an object is.
- But for now, ignore. $\alpha=1$ means object is opaque.

Demos

- ToyFinal
- Shutterbug
- light

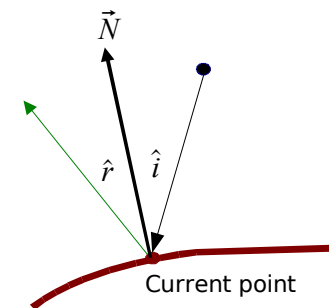
What's the problem with the diffuse model?

- Not all objects are "matte".
- Some surfaces are shiny.
- But what does "shiny" mean?
- Shiny means reflective.
- So we introduce *specular model*.
- Speculum is latin for "mirror".
- Allows us to model shiny objects and highlights.

How would a perfectly shiny object behave?

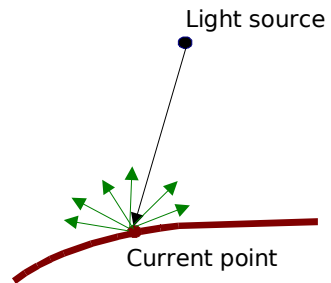
- Consider a mirror.
- Close to perfectly shiny object.
- How does it behave when struck by light?
- Think back to physics:
- Angle of incidence = angle of reflection.
- All of the intensity goes in one direction.

Perfectly shiny objects

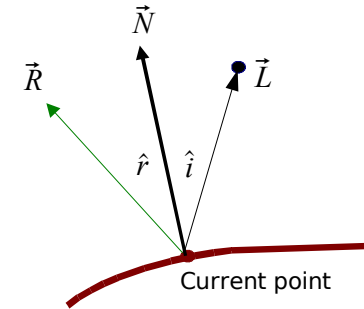


How is this different?

- Compare with diffuse:



What is the reflected vector?



What is the equation of the reflected vector?

- Can work it out ... see overhead.

$$\vec{R} = 2\vec{N}(\vec{N} \cdot \vec{L}) - \vec{L}$$

What does this mean?

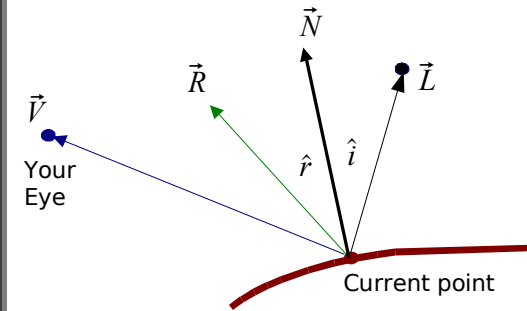
- When could we see the light source?
- For a perfect reflector, can only see light source *if our eye is directly along the R vector*.
- Why? Because for a perfect reflector, the only emitted light is along the R vector, so the only way it can reach our eye is if it goes "straight into" our eye.

Implications

- Specular reflection is *viewer dependent*.
- Diffuse reflection could be calculated in advance, since it is *viewer independent*.
- But specular reflection depends on where the viewer is.
- Think of the "highlight".

Introducing V

- V is the vector from the current point to our eye.



Our model of a mirror

- Mirror has a k_s - *specular reflection coefficient*.
- This allows us to have tinted mirrors.
- For a good mirror k_s is almost 1.
- Equation for specular reflection for perfect mirror is thus

$$I_s = I_l k_s, \text{ if } \vec{R} = \vec{V}$$
$$I_s = 0, \text{ otherwise}$$

Where do highlights come from

- If you have a curved surface, there may be some point where $R = V$

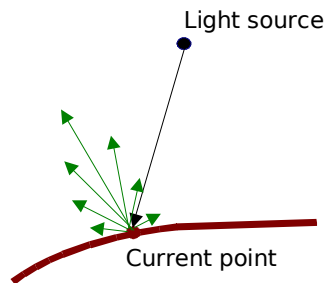
But what about other objects?

- Some objects are in the middle ... somewhere between.
- Example: Shiny green apple.
- Not a perfect reflector, but is "shiny".
- So how do we model it?

Phong illumination model

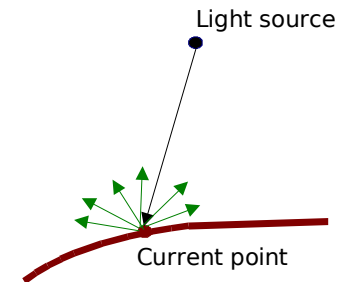
- Most of light is reflected close to the angle of reflection, but also some in other directions.
- Some objects spread light out more than others.
- Example: Perfectly reflective surface reflects all light at angle of reflection.
- Example: Apple reflects most light at angle of reflection, but gradually decreases as the angle increases from angle of reflection.

Phong model



How is this different?

- Compare with diffuse:



The equation

$$I_s = I_l k_s \cos^n(\alpha)$$

where

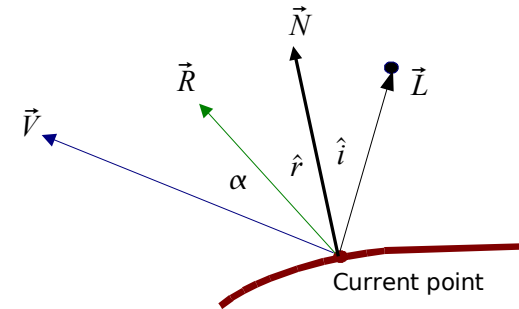
n = Phong exponent

k_s = specular reflection coefficient

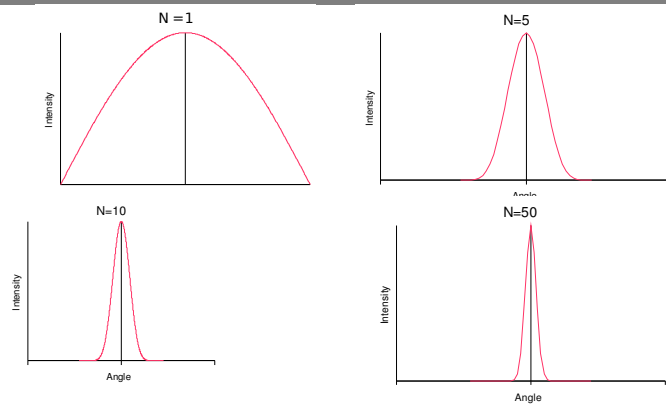
α = Angle between R and V

- Why? Because:
 - It looks ok in practice.
 - It is easy to compute.

Angle from reflected light source



What happens as we vary n ?

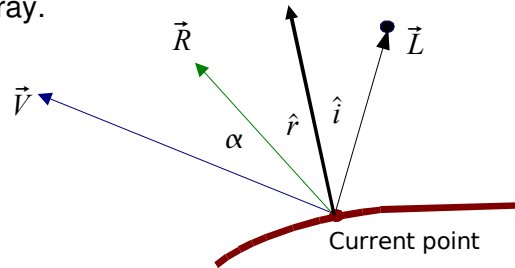


Demos

- See OpenGL + light + Nate demo
- Note:
 - No "headlight"
 - The centre of the highlight is when $R = V$
 - Highlight moves around.

And now the maths

- Alpha is the angle between us and the reflected ray.



The equation again

- Previous equation

$$I_s = I_l k_s \cos^n(\alpha)$$

where

n = Phong exponent

k_s = specular reflection coefficient

α = Angle from reflected light source

But ...

- Remember

$\cos(\alpha)$ is the angle between \vec{V} and \vec{R}
 hence if \vec{V} and \vec{R} are normalised
 then $\cos(\alpha) = \vec{V} \cdot \vec{R}$ and hence
 $I_s = I_l k_s (\vec{V} \cdot \vec{R})^n$

Multiple light sources

- To update our overall equation:

$$I = I_a k_a + \sum_{l_k=l_1..l_n} I_{l_k} (k_d \vec{N} \cdot \vec{L}_k + k_s (\vec{V} \cdot \vec{R}_k)^n)$$

where

$l_1..l_n$ are the point light sources

\vec{L}_k = Vector from point to light source k

I_{l_k} = Intensity of light source k

$$\vec{R}_k = 2 \vec{N} (\vec{N} \cdot \vec{L}_k) - \vec{L}_k$$

Specular Reflection in OpenGL

- Setting k_s :

```
float [] spec_mat =  
{0.8f, 0.8f, 0.8f, 1.0f};  
glMaterialfv(GL_FRONT, GL_SPECULAR,  
spec_mat);
```

- k_s is usually close to white.
- But also need to set phong exponent! In OpenGL, called shininess.

```
glMaterialf(GL_FRONT, GL_SHININESS, 20);
```

OpenGL lighting

- OpenGL lighting is flexible.
- Default settings: ambient light is (0,0,0), diffuse light is (1,1,1) for GL_LIGHT0, and (0,0,0) for others, similarly for specular light.
- Most commonly, per-light ambient is not used. and `GL_DIFFUSE==GL_SPECULAR`

A less common refinement

- Light attenuation:
- Further we get from light source, the dimmer it gets.
- Only depends on distance to light source.

Illum eqn for light attenuation

- Let the distance to light source k be d_k , then:

$$I = I_a k_a + \sum_{l_k=l_1..l_n} f_{att}(d_k) I_{l_k} (k_d \vec{N} \cdot \vec{L}_k + k_s (\vec{V} \cdot \vec{R}_k)^n)$$

where

$l_1..l_n$ are the point light sources

\vec{L}_k = Vector from point to light source k

f_{att} = Light attenuation function

d_k = Distance to light source l_k

What is f_{att} ?

- Function should be between 0 and 1.
- Physically correct (inverse square law):

$$f_{att} = \frac{1}{d^2}$$

But it looks wrong!!

- Why?
 - Because real lights aren't points.
 - Not "capped" at 1.

Practical f_{att} ?

- Typical f_{att} used:

$$f_{att} = \max\left(\frac{1}{c_1 + c_2 d + c_3 d^2}, 1\right)$$

with appropriate values of c_1 , c_2 , c_3

- For simplicity in following equations, will ignore f_{att} .

Attenuation in OpenGL

- OpenGL supports.

$c_1 = \text{GL_CONSTANT_ATTENUATION}$

$c_2 = \text{GL_LINEAR_ATTENUATION}$

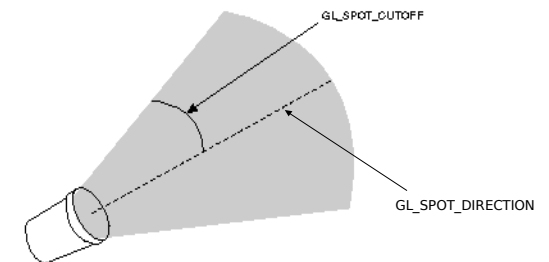
$c_3 = \text{GL_QUADRATIC_ATTENUATION}$

- E.g

```
glLightf(GL_LIGHT0,  
GL_QUADRATIC_ATTENUATION, 0.5);
```

Spotlights

- Sometimes want to constrain angle of light to create that spotlight effect.



Stuff that needs to be set

- Location of spotlight: set using GL_POSITION
- Direction of spotlight. Set using GL_SPOT_DIRECTION
- Angle of spotlight. Set using GL_SPOT_CUTOFF
- Can also set exponent: GL_SPOT_EXPONENT
- Exponent makes brighter in middle than on edge

Example

```
float[] hlCol = {1.0f, 1.0f, 1.0f, 1.0f};
float[] hlPos = { 1.0f, 2.0f, 3.0f, 1.0f };
float[] hlDir = { 1.0f, 0.0f, 0.0f, 0.0f };
gl.glLightfv(GL.GL_LIGHT1, GL.GL_DIFFUSE, hlCol);
gl.glLightfv(GL.GL_LIGHT1, GL.GL_POSITION, hlPos);
gl.glLightf(GL.GL_LIGHT1, GL.GL_SPOT_EXPONENT, 0);
gl.glLightf(GL.GL_LIGHT1, GL.GL_SPOT_CUTOFF, 60);
gl.glLightfv(GL.GL_LIGHT1, GL.GL_SPOT_DIRECTION,
hlDir);
gl.glEnable(GL.GL_LIGHT1);
```

Great ... how do we use it?

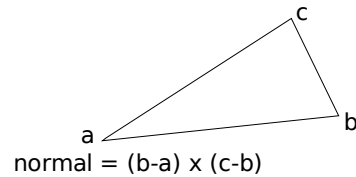
- Will explore two techniques
 - Polygon rendering
 - Ray tracing

Polygon rendering

- World is modelled as a group of polygons.
- To start off with, we assume we have the normal to each polygon.
- Takes each polygon and as we scan-convert the polygon, works out the colour of each pixel.
- Scan conversion - taking a polygon and drawing it on screen.

What if we don't have normal?

- Easy to work out, using our old friend, cross product - then normalise.
- Take cross product of first edge with second edge will give us a normal consistent with right hand rule.



How to find equation of plane

- If you have the normal, how can you work out equation of plane?
- Easy: Normal to plane means perpendicular to any vector on the plane. Let $n = (a,b,c)$
- If p_1 is a point on the plane, then for any point $p=(x,y,z)$ on the plane:
 $n \cdot (p-p_1) = 0$.
- Hence $ax+by+cz - n \cdot p_1 = 0$ <- equation of plane

Warning: Important

- Don't forget to normalise normals.
- Make sure normal is consistent with right hand rule or OpenGL will get confused.
- What happens when you scale?
- By default OpenGL does nothing
- If you want it to scale normals correctly, must use `glEnable(GL_NORMALIZE)`

Pseudocode - Brute force

```
public renderScene(PolygonList polys, LightList lights, Point3D view)
{
    for poly from polys {
        for each (x, y) in poly {
            Point3D origPoint = poly.originalPoint(x,y);
            // denotes the original coordinates of the polygon in world
            // space
            Colour colour = applyIllum(origPoint,
                poly.material, poly.normal, lights, view);
            WritePixel(x, y, colour);
        }
    }
}
```

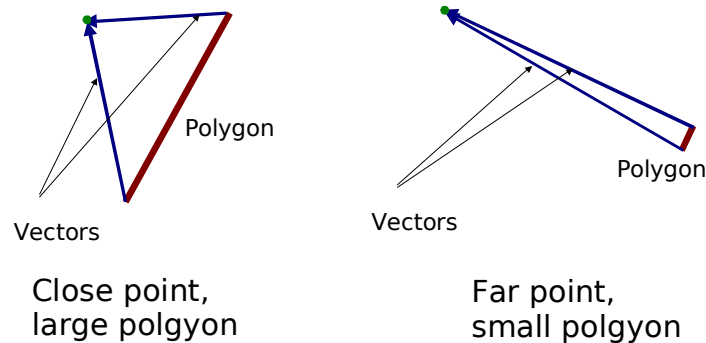
How is this?

- It works.
- It is really slow - we are applying the illumination equation at **every pixel**.
- Maybe we can make some assumptions.

Some easy assumptions

- If we assume that:
 - The viewer is sufficiently far away and the polygon sufficiently small that the vector to the viewer is almost the same across the polygon.
and
 - All light sources are sufficiently far away and the polygon is sufficiently small that the vector to the light source is almost the same across the polygon.
- then each polygon has a single colour.

Diagram of vector variation



Pseudocode - Flat shading

```
public renderScene(PolygonList polys, LightList lights, Point3D view)
{
  for poly from polys {
    Point3D polyCentre = poly.origCentre();
    Colour colour = applyIllum(polyCentre,
      poly.material, poly.normal, lights, view);
    for each (x, y) in poly {
      WritePixel(x, y, colour);
    }
  }
}
```

How is this approach?

- This approach is called flat shading.
- Much faster ... only apply illumination equation once per polygon, not once per pixel. Illumination equation is out of the loop.
- Base assumptions turns out to be OK in most cases, if it isn't we can split big close polygons.
- Looks cool for cubes, polyhedrons etc.
- Shutterbug demo.

Flat shading in OpenGL

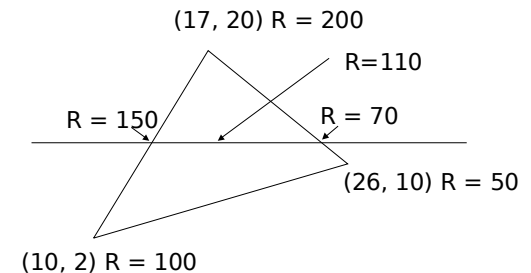
- The default. But can be enabled explicitly using:
`glShadeModel (GL_FLAT) ;`
- Note that if the light source is directional, then flat shading is correct, as normal remains same.

Slightly more clever approach

- How about .. we work out the colour of each of the corners of the polygon and linearly interpolate between them?
- Interpolate along edges and then to middle.

Linear interpolation

- Interpolating red component:



Pseudocode - Interpolated shading

```
public renderScene(PolygonList polys, LightList lights,
    Point view){
    for poly from polys {
        Colour vertexCol[];
        for vertex[i] in poly {
            Colour vertexCol[i] =
                applyIllum(poly.originalPoint(vertex[i]),
                    poly.material, poly.normal, lights, view);
            for each (x, y) in poly {
                // original coordinates of the polygon in world
                space
                colour = interpolate(x, y, poly, vertexCol);
                WritePixel(x, y, colour);
            }
        }
    }
}
```

COMP3421: Comp Graphics - shading

Slide 73

File: /home/ambert/graphics/slides/shade/COMP3421-shade.odp

How is this?

- Better, but still doesn't look right.
- Still misses some things, e.g. won't do specular highlights, unless they're at the corner of the polygon.
- A little more expensive than flat shading, but not much. One application of illumination equation per point + a few extra adds per pixel.
- This should look good, but it looks almost the same as flat shading.

COMP3421: Comp Graphics - shading

Slide 74

File: /home/ambert/graphics/slides/shade/COMP3421-shade.odp

We made an implicit assumption!!

- Turns out we made an implicit assumption which was not so good.
- The assumption was:
That the polygons represent the surface being modelled, and are **not** an approximation to a curved surface.
- This turns out not to be a good thing to assume - many times this is patently wrong.

COMP3421: Comp Graphics - shading

Slide 75

File: /home/ambert/graphics/slides/shade/COMP3421-shade.odp

Why is this so bad?

- Eye is not just a camera.
- Eye enhances edges. This can lead to "banding" effects (Mach bands).
- Have to be more clever when modelling surfaces.



COMP3421: Comp Graphics - shading

Slide 76

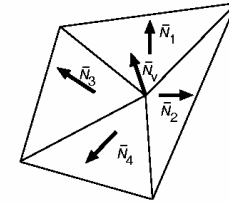
File: /home/ambert/graphics/slides/shade/COMP3421-shade.odp

What is the trick?

- We also assumed that there was one normal per polygon.
- But what is the effect if we actually store normals **per vertex**?
- We compute the normals at each vertex **from our curved surface**, rather than from the polygon itself.
- Store the normal with each vertex.

What if we don't have vertex normals?

- Can average out the normals from all the faces that meet at that vertex.



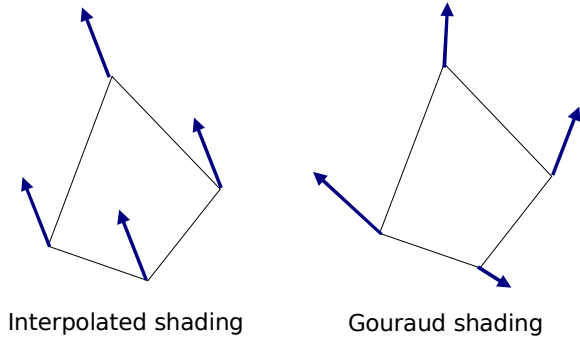
Result: Smoother looking polygons

- This works really well, but there's an additional issue:
- How do we make use of this idea?

Technique 1 - Gouraud shading

- Like interpolated shading ...
- Except that when we work out the colours of the corners, we use the normal for that corner.

Difference



Gouraud shading pseudocode

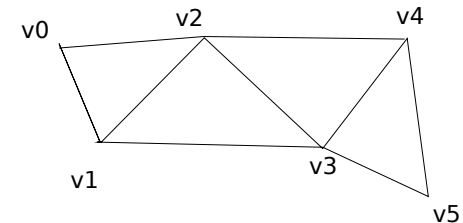
```
public renderScene(PolygonList polys, LightList lights, Point view){
    for poly from polys {
        Colour vertexCol[];
        for vertex[i] in poly {
            Colour vertexCol[i] = applyIllum(poly.OrigPoint(vertex[i]),
                poly.material, vertex[i].normal, lights, view);
        }
        for each (x, y) in poly {
            // denotes the original coordinates of the polygon in world
            // space
            colour = interpolate(x, y, poly, vertexCol);
            WritePixel(x, y, colour);
        }
    }
}
```

Gouraud shading in OpenGL

- Gouraud shading is called smooth shading in OpenGL
- To enable:
`glShadeModel(GL_SMOOTH);`
- But also have to define normals!
`glNormal3f(...);`
- Is stateful in OpenGL: applies to each glVertex that follows.

Consequences for triangle strips

- Triangle strips are kind of funny.



- Draw v0,v1,v2 then v2,v1,v3 then v2,v3,v4 then v4,v3,v5. Why? Because of normal.

Normals & lighting in OpenGL

- One normal per vertex ALWAYS. To do flat shading, just set same normal for all vertices.
- Can do two-sided or one-sided lighting.
- In one-sided: only front-facing side (as determined by vertex order) is rendered.
- In two-sided: front-facing side is rendered as usual. Back-facing side: use negative normal of front-facing side and render.

Implications for assignment

- What happens to polygons? MUST ensure front-facing convention and normal value agree, otherwise, polygons won't be lit correctly.
- What happens to triangle strips?

Gouraud shading pros & cons

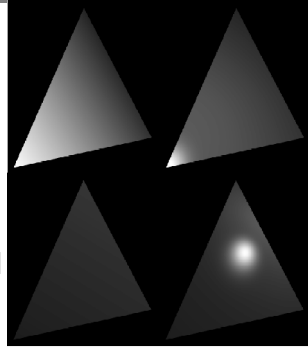
- Pros:
 - Quick: One illum eqtn per vertex, a few adds per pixel.
 - Simple - easy to implement in hardware.
 - Looks good for objects with matte surfaces.
- Cons
 - Doesn't look so good with specular objects
 - Some inherent problems with polygon rendering (more on that soon).

So ... what's the alternative?

- Go back to our original idea of working out colour per pixel.
- BUT, interpolate normal to work out the normal at each point from the corners.
- Called Phong shading
- Not to be confused with Phong illumination model.

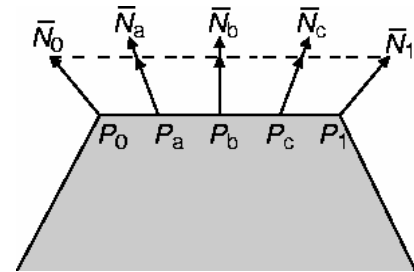
Problem with Gouraud for specular

- If specular point is in middle of polygon, Gouraud shading misses it completely.
- If it is at a vertex, you get artificial looking highlight which shows the polygonal nature of the shape.



Gouraud Phong

Diagram



Phong shading pseudocode

```
public renderScene(PolygonList polys, LightList lights, Point view){
  for poly from polys {
    for each (x, y) in poly {
      Point3D origPoint = poly.originalPoint(x,y);
      // denotes the original coordinates of the polygon in world
      // space
      Vector3D normal = poly.interpNormal(x,y);
      Colour colour = applyIllum(origPoint,
        poly.material, normal, lights, view);
      WritePixel(x, y, colour);
    }
  }
}
```

Phong shading pros & cons

- Pro
 - Beautiful looking! Does specular really well. Even improves diffuse rendering.
 - More physically accurate
- Con
 - Slow!! One illum equation **per pixel**, plus interpolating normals **and normalising them** is quite expensive.
 - General problems of polygon rendering.

Phong shading in OpenGL

- Can be done, using environment mapping or using a fragment shader (we'll learn about these later.)
- See ToyShader.

Polygon rendering pros and cons

- Fast and simple.
- Hardware support
- But:
 - Silhouettes.
 - Doesn't easily handle object to object reflections, shadows, transparent or refractive objects.