COMP3421

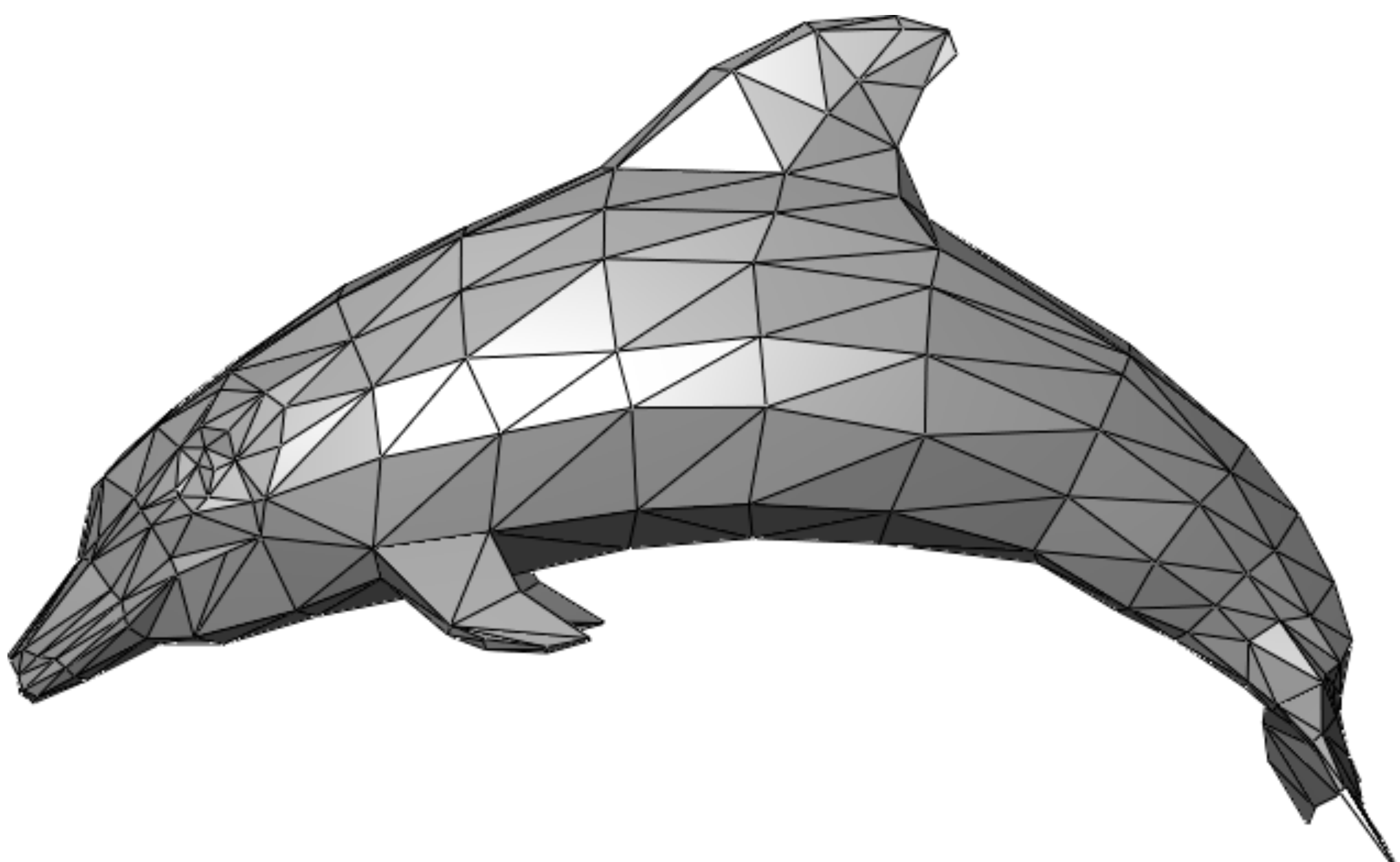
Meshes, Lighting

Robert Clifton-Everest

Email: robertce@cse.unsw.edu.au

Meshes

- We represent 3D objects as polygonal meshes.
- A mesh is a collection of polygons in 3D space that form the skin of an object.



Meshes

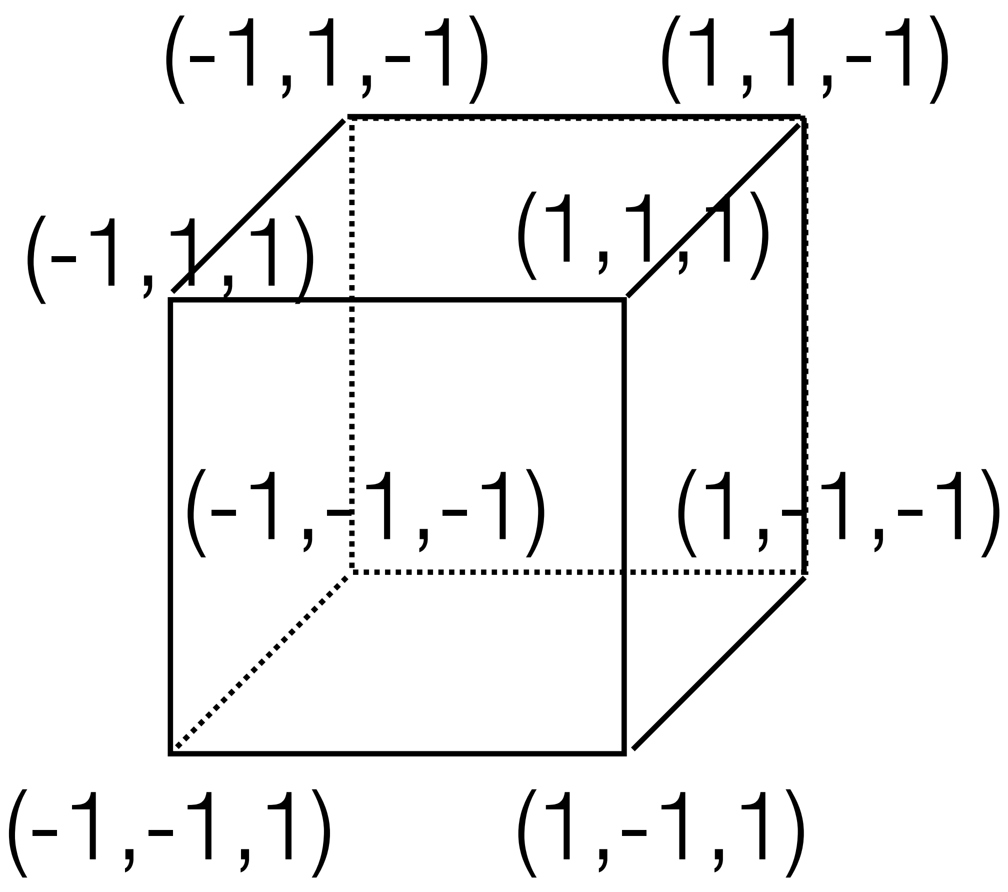
- Triangle meshes are polygonal meshes that only contain triangles
- They are generally easier to work with at the cost of requiring more memory
- Meshes of arbitrary polygons can be converted into triangle meshes by tessellating any polygons with more than 3 vertices

Mesh Data Structures

- It is common to represent a polygon mesh in terms of lists:
  - vertex list: all the vertices used in the mesh
  - face list: each face's vertices as indices into the above list.

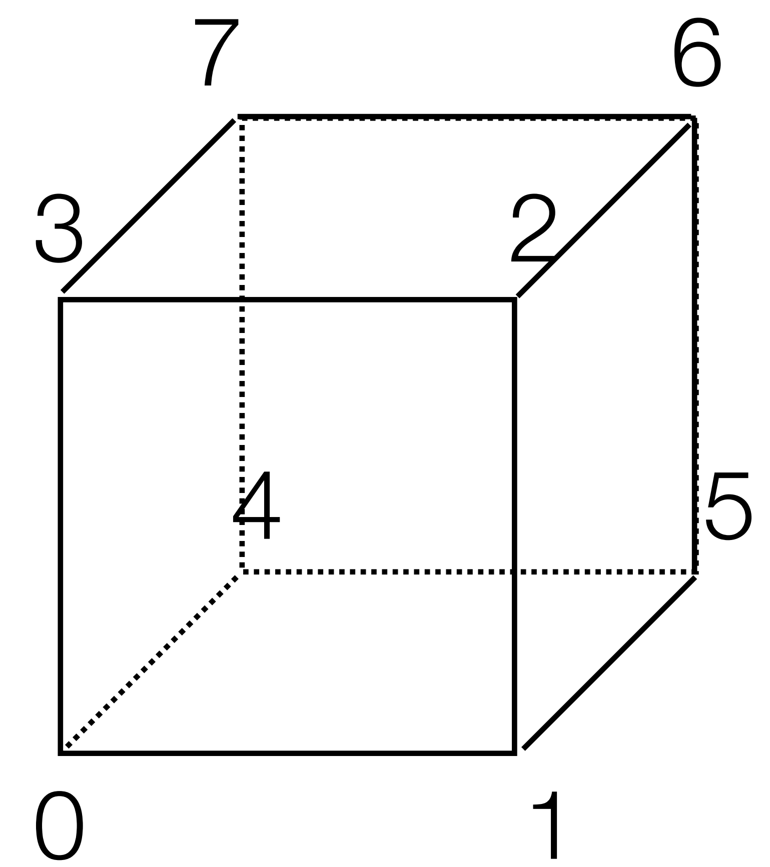
Cube

vertex	x	y	z
0	-1	-1	1
1	1	-1	1
2	1	1	1
3	-1	1	1
4	-1	-1	-1
5	1	-1	-1
6	1	1	-1
7	-1	1	-1



Cube

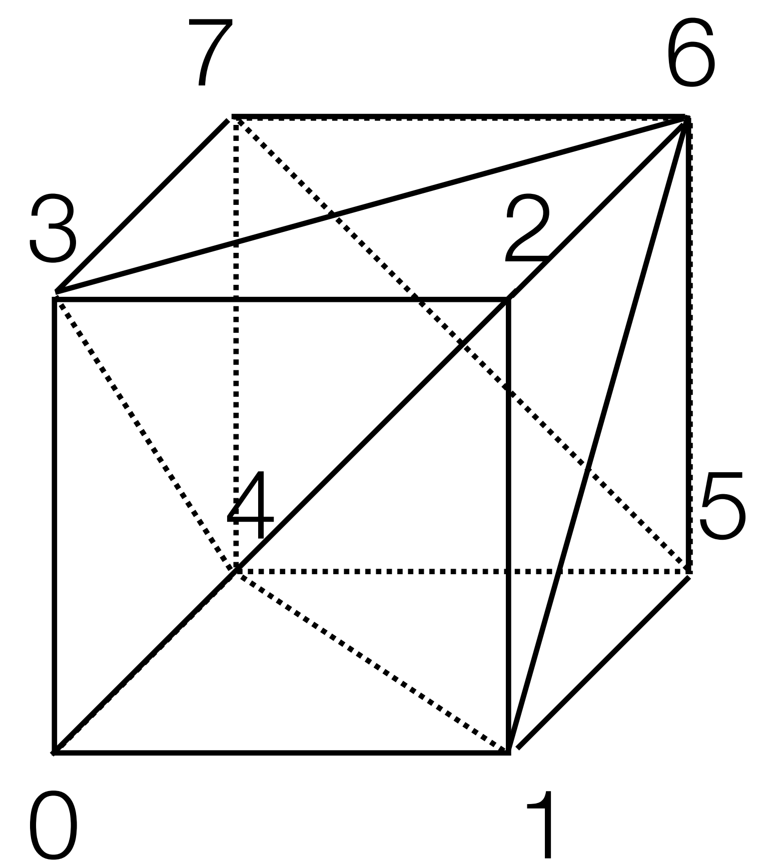
face	vertices
0	0,1,2,3
1	1,5,6,2
2	5,4,7,6
3	4,0,3,7
4	3,2,6,7
5	4,5,1,0



Cube (triangle mesh)

face	vertices
0	0,1,2
1	2,3,0
2	1,5,6
3	6,2,1
4	5,4,7
5	7,6,5

face	vertices
6	4,0,3
7	3,7,4
8	3,2,6
9	6,7,3
10	4,5,1
11	1,0,4



Indexed Drawing

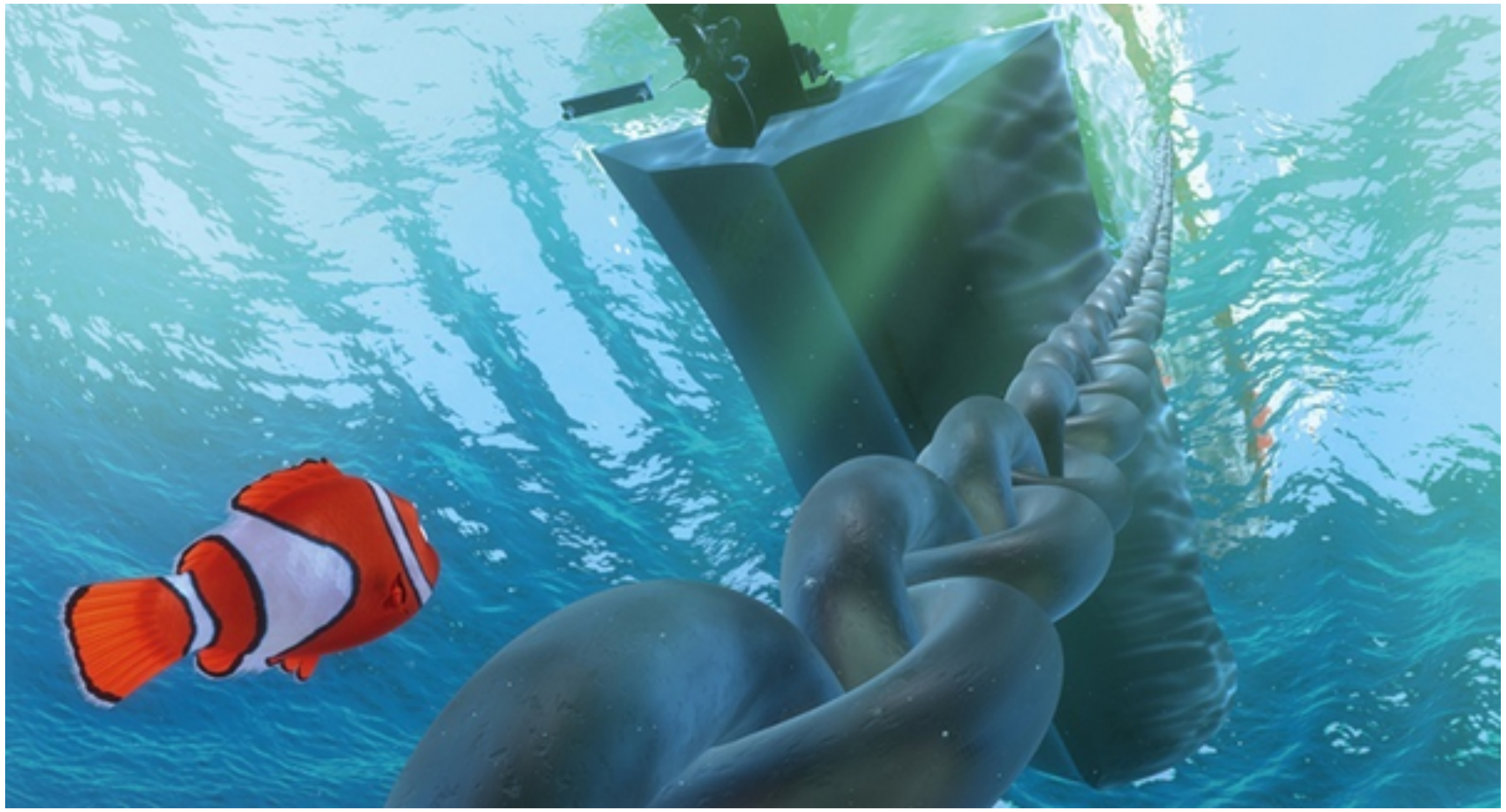
```
// Draw something using indexed data
gl.glDrawElements(int mode, int count,
                  int type, long offset);

//mode - GL_TRIANGLES etc
//count - number of indices to be used.
//type - type of index array
//offset - in bytes!
```

Indexed Drawing

- We can use indexed drawing to draw meshes.
- Triangle meshes are simplest as we can just use GL_TRIANGLES
- Other polygonal meshes are more complex and we won't cover in this course.
- See IndexedCube.java

Efficiency



Efficiency

- Transferring a large triangle mesh to the GPU is expensive
- If it does not change we only need to transfer it once at the start of the program
  - e.g. By overriding the Application3D.init() method

PLY format

- The PLY format is a simple file format for representing polygon data.
- https://en.wikipedia.org/wiki/PLY_(file_format)
- We can read this format via JPLY.
- https://github.com/smurn/jPLY

PLY format

- The header specifies how many vertices and how many faces there are and what format they're in.

```
ply
format ascii 1.0
comment simple cube
element vertex 8
property float x
property float y
property float z
element face 6
property list uchar int vertex_indices
end_header
```

PLY format

```
ply
format ascii 1.0
comment simple cube
element vertex 8
property float x
property float y
property float z
element face 6
property list uchar int vertex_indices
end_header
```

8 vertices

x, y, and z coordinates

6 faces

Each face is a list of ints

PLY format

- The body lists all the vertices and faces.

```
-1 -1  1
 1 -1  1
 1  1  1
-1  1  1
-1 -1 -1
 1 -1 -1
 1  1 -1
-1  1 -1
4 0 1 2 3
4 1 5 6 2
4 5 4 7 6
4 4 0 3 7
4 3 2 6 7
4 4 5 1 0
```

Vertices

Faces

PLY format

- JPLY automatically tessellates polygonal faces into triangles so we can draw using glDrawElements()
- See IndexedCube.java

Meshes in UNSW graph

- TriangleMesh.java lets us load in PLY files as triangle meshes, but:
  - Have to make sure we construct and initialise the mesh at the right time
  - Meshes can have different scales and positions, so may have to translate it.
- See ModelViewer.java

Illumination

- Why can’t we see the details in the bunny?
- We need lighting!
- In this section we will be considering how much light reaches the camera from the surface of an object.

Achromatic Light

- To start with we will consider lighting equations for achromatic light which has no colour, but simply an intensity.
- We will then extend this (next week) to include coloured lights.

Local Illumination

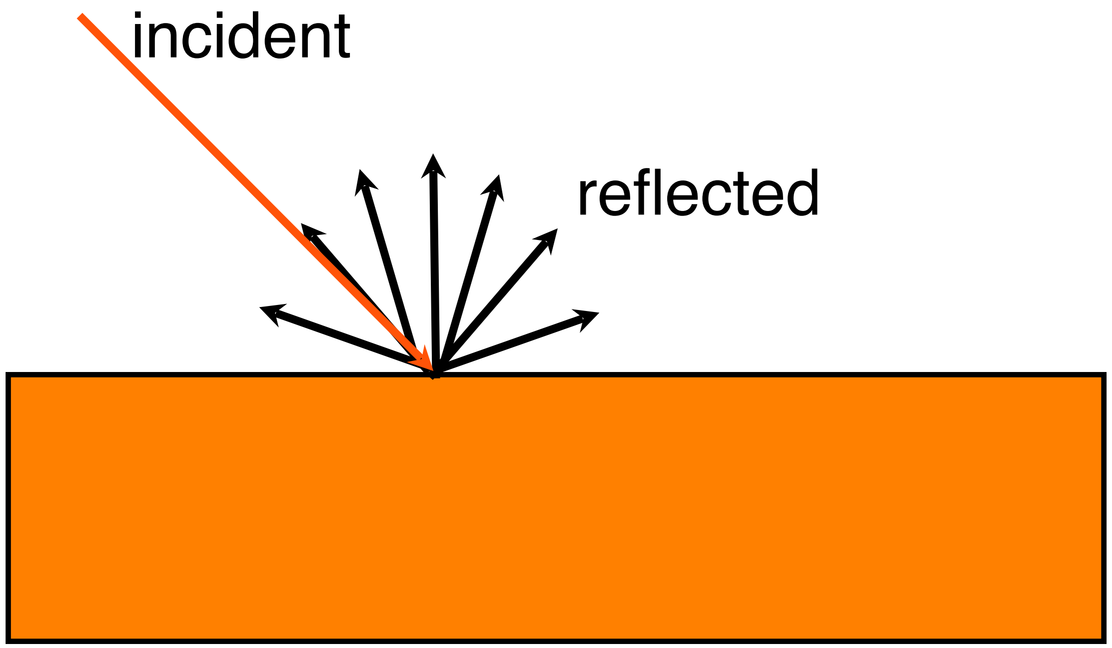
- Inter-reflections: In real life light reflects from a light off one object onto another object etc
- So objects with no direct light are not completely in darkness
- This is very costly to model.
- In OpenGL we use local illumination and only model reflections directly from a light source…. (and then add a fudge factor)

Illumination

- The colour of an object in a scene depends on:
  - The colour and amount of light that falls on it.
  - The colour and reflectivity of the object eg. Red object reflects red light
- There are two kinds of reflection we need to deal with: diffuse and specular.

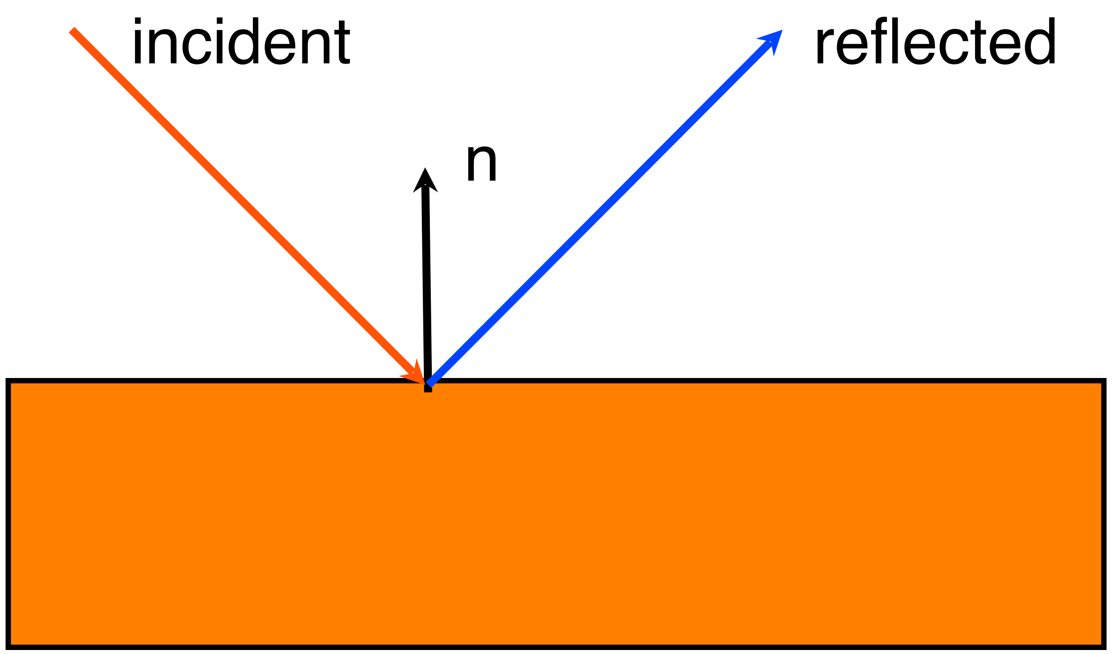
Diffuse reflection

- Dull or matte surfaces exhibit diffuse reflection. Light falling on the surface is reflected uniformly in all directions. It does not depend on the viewpoint.



Specular reflection

- Polished surfaces exhibit specular reflection. Light falling on the surface is reflected at the same angle. Reflections will look different from different view points.



Components

- Most objects will have both a diffuse and a specular component to their illumination.

- We will also include an ambient component to cover lighting from indirect sources.

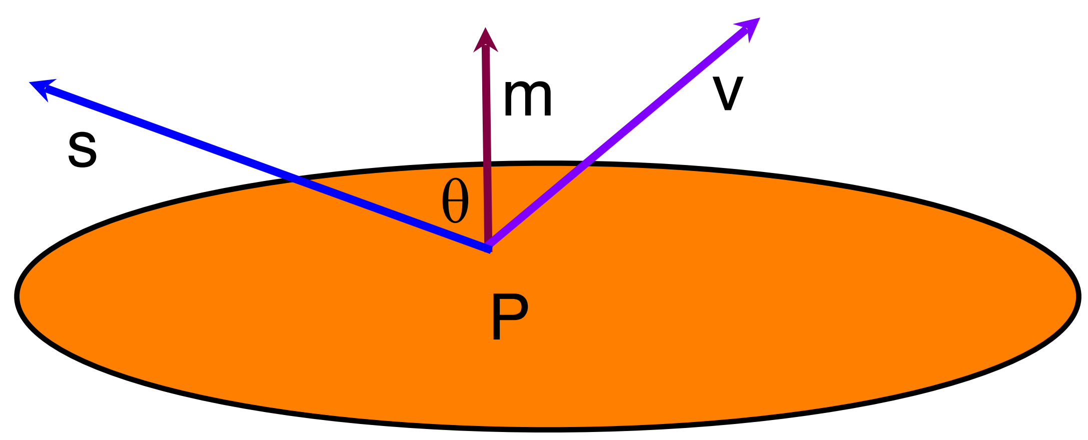
- We will build a lighting equation:

$$I(P) = I_{ambient}(P) + I_{diffuse}(P) + I_{specular}(P)$$

- I(P) is the amount of light coming from P to the camera.

Ingredients

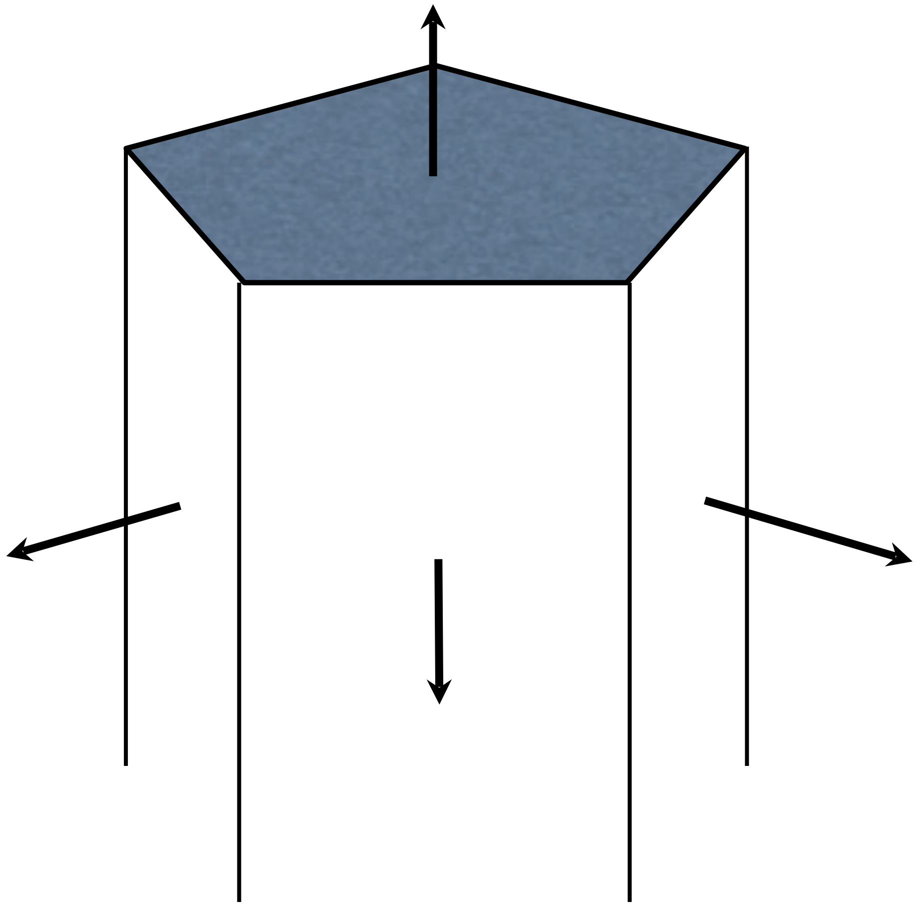
- To calculate the lighting equation we need to know three important vectors:
  - The normal vector $\mathbf{m}$ to the surface at P
  - The view vector $\mathbf{v}$ from P to the camera
  - The source vector $\mathbf{s}$ from P to the light source.



Modeling Normals

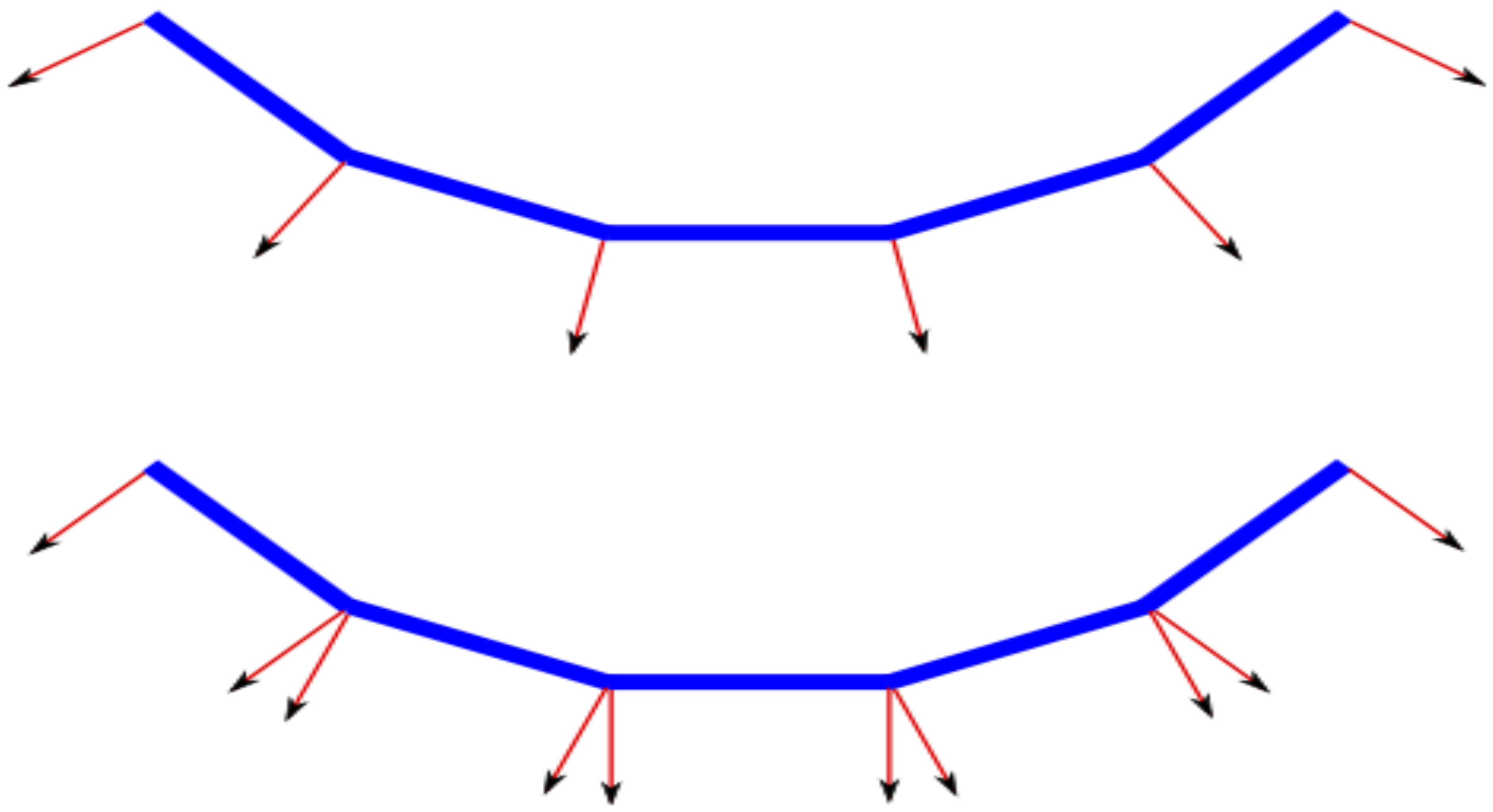
- Every vertex has an associated normal
- On flat surfaces, we want to use face normals set the normals perpendicular to the face.
- On curved surfaces, we may want to specify a different value for the normal, so the normals change more gradually over the curvature.

Face Normals



Smooth vs Flat Normals

Imagine this is a top down view of a prism



Calculation of Face Normals

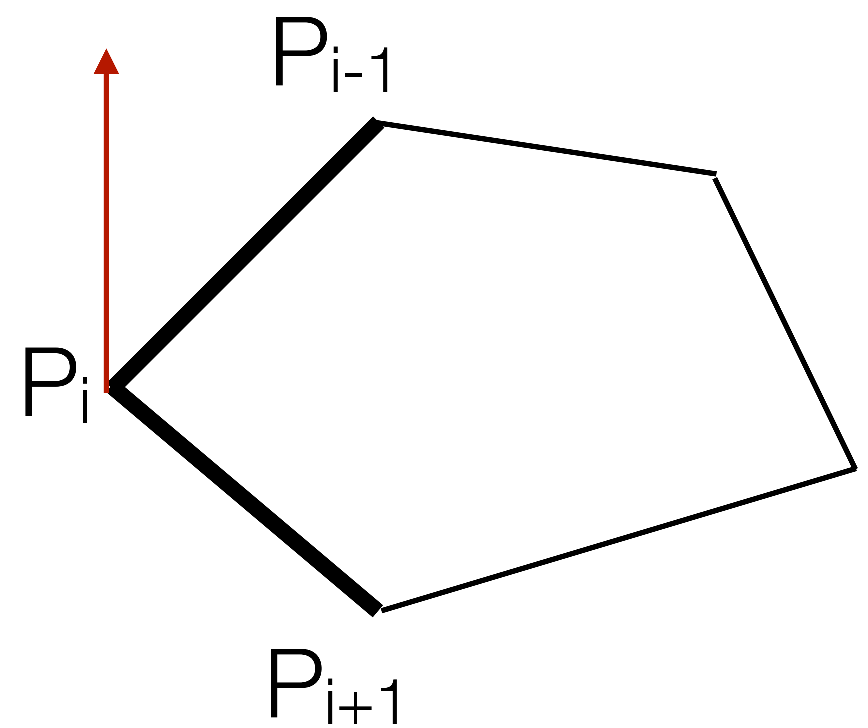
- Every vertex for a given face will be given the same normal.
- This normal can be calculated by
- Finding cross product of 2 sides if the polygon is convex (triangles are always planar)
- Using Newell's method for arbitrary polygons which may not be convex (or even planar)

Cross product method

- Works for any convex polygon (vertices MUST be in CCW order)

- Pick two (non-parallel) adjacent edges and calculate:

$$n = (P_{i+1} - P_i) \times (P_{i-1} - P_i)$$



Exercise

- Calculate the face normal for the triangle defined by points A=(0,0,1), B=(1,0,1), C =(1,1,0).

Newell's Method

- A robust approach to computing **face normal** for arbitrary polygons:

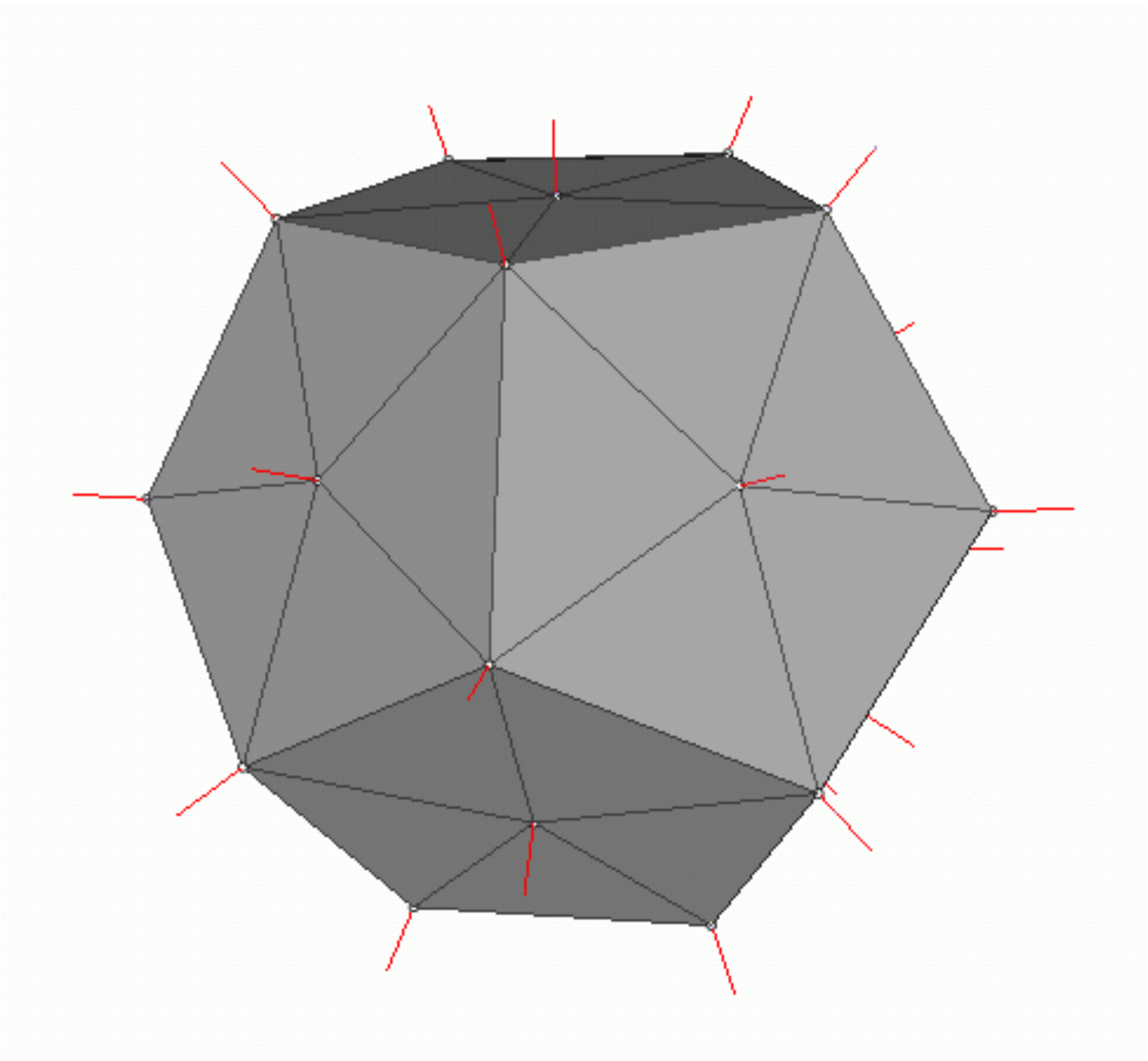
$$
\begin{aligned}
n_x &= \sum_{i=0}^{N-1} (y_i - y_{i+1})(z_i + z_{i+1}) \\
n_y &= \sum_{i=0}^{N-1} (z_i - z_{i+1})(x_i + x_{i+1}) \\
n_z &= \sum_{i=0}^{N-1} (x_i - x_{i+1})(y_i + y_{i+1})
\end{aligned}
$$

$$\text{where } (x_N, y_N, z_N) = (x_0, y_0, z_0)$$

Vertex Normals

- For smooth surfaces we can calculate each normal based on
  - maths if it is a surface with a mathematical formula
  - **averaging the face normals of adjacent vertices** (if this is done without normalising the face normals you get a weighted average). This is the basic way and can be fined tuned to exclude averaging normals that meet at a sharp edge etc.

Vertex Normals



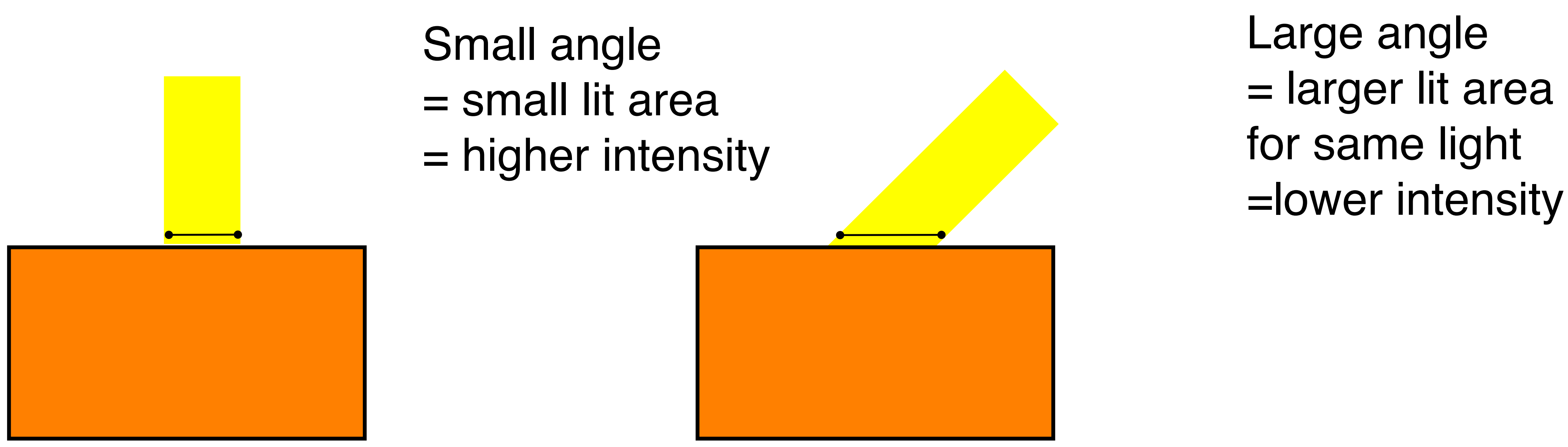
Vertex Normals

- We keep a separate buffer of normals for a mesh.
- In TriangleMesh.java we calculate this normal buffer by a weighted average of the face normals

Diffuse illumination

- Diffuse scattering is equal in all directions so does not depend on the viewing angle.

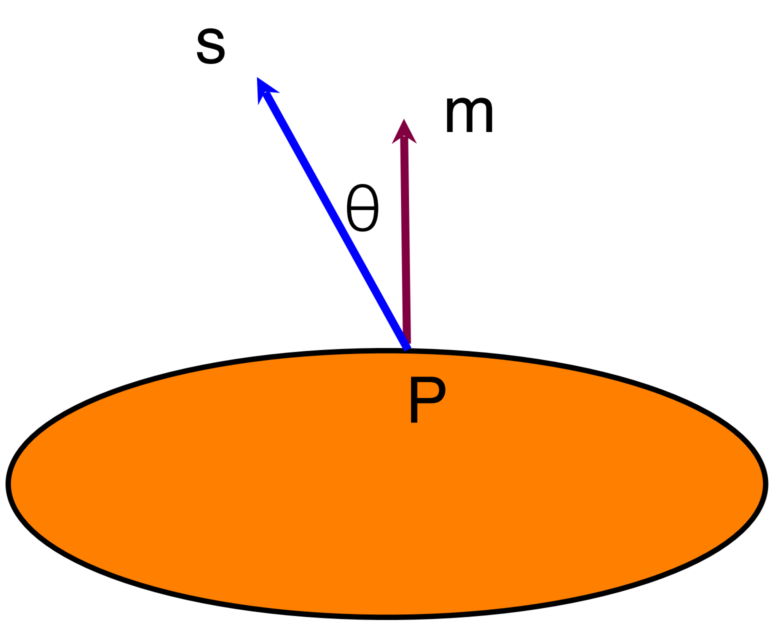
- The amount of reflected light depends on the angle of the source of the light



Lambert's Cosine Law

- We can formalise this as Lambert's Law:

$$
\begin{aligned}
I_d &\propto I_s \cos\theta \\
&= I_s \rho_d (\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})
\end{aligned}
$$



where:

- I_s is the source intensity, and
- ρ_d is the diffuse reflection coefficient in [0,1]
- NOTE: Both vectors are normalised!

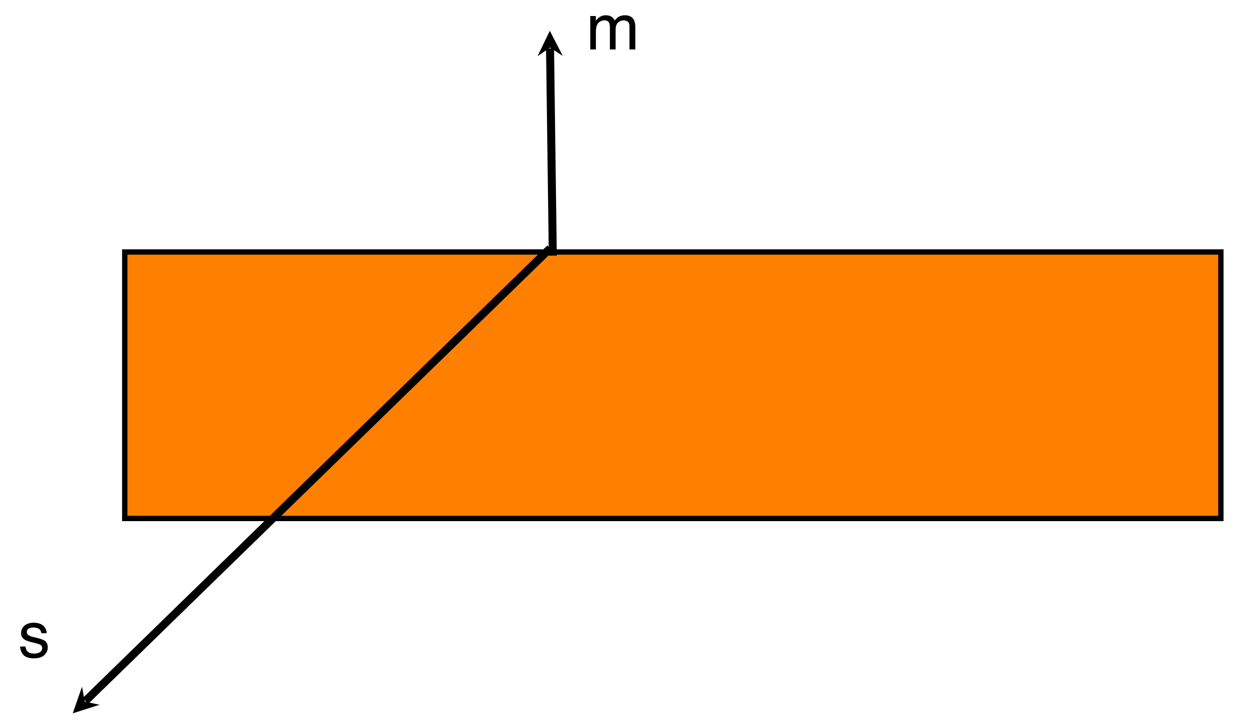
Lambert’s Cosine Law

- When the angle is 0 degrees the cosine is 1
  - You get all the reflected light back
- When the angle is 90 degrees
  - None of the light is reflected back
- When the angle is > 90 degrees
  - cos gives us a negative value! This is not what we want.

Lambert's Law

- If the angle > 90, then the light is on the wrong side of the surface and the cosine is negative. In this case we want the illumination to be zero. So:

$$I_d = \max\left(0, I_s \rho_d (\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})\right)$$

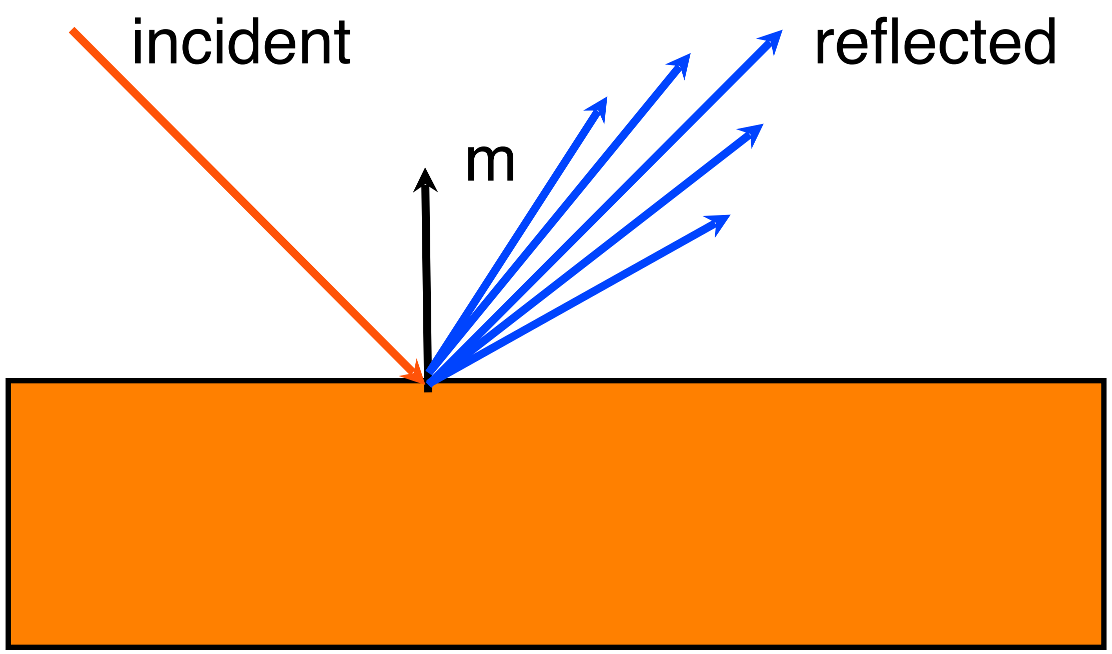


Diffuse reflection coefficient

- The coefficient ρ_d is a property of the surface.
  - Light surfaces have values close to 1 as they reflect more light
  - Dark surfaces have values close to 0 as they absorb more light

Specular reflection

- Only mirrors exhibit perfect specular reflection. On other surfaces there is still some scattering.



Phong model

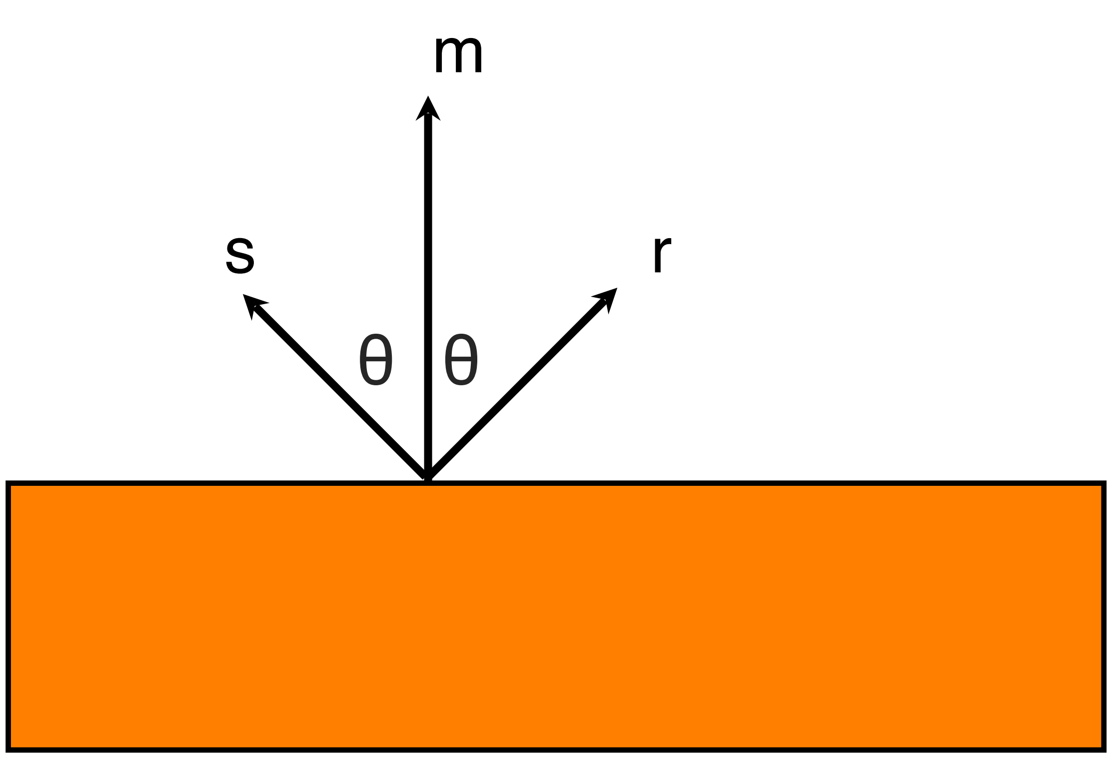
- The Phong model is an approximate model (not correct physics) of specular reflection. It allows us to add highlights to shiny surfaces.

- It looks good for plastic and glass but not good for polished metal (in which real reflections are visible).

Phong model

- Reflection is brightest around the reflection vector:

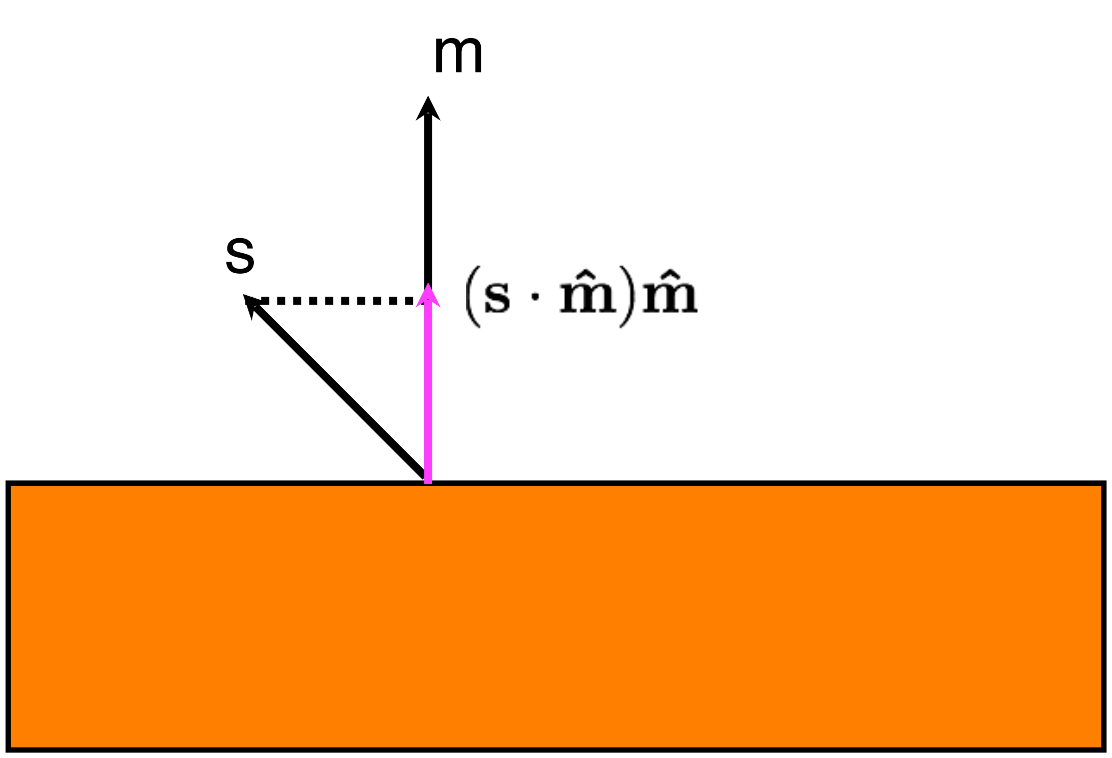
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

- Reflection is brightest around the reflection vector :

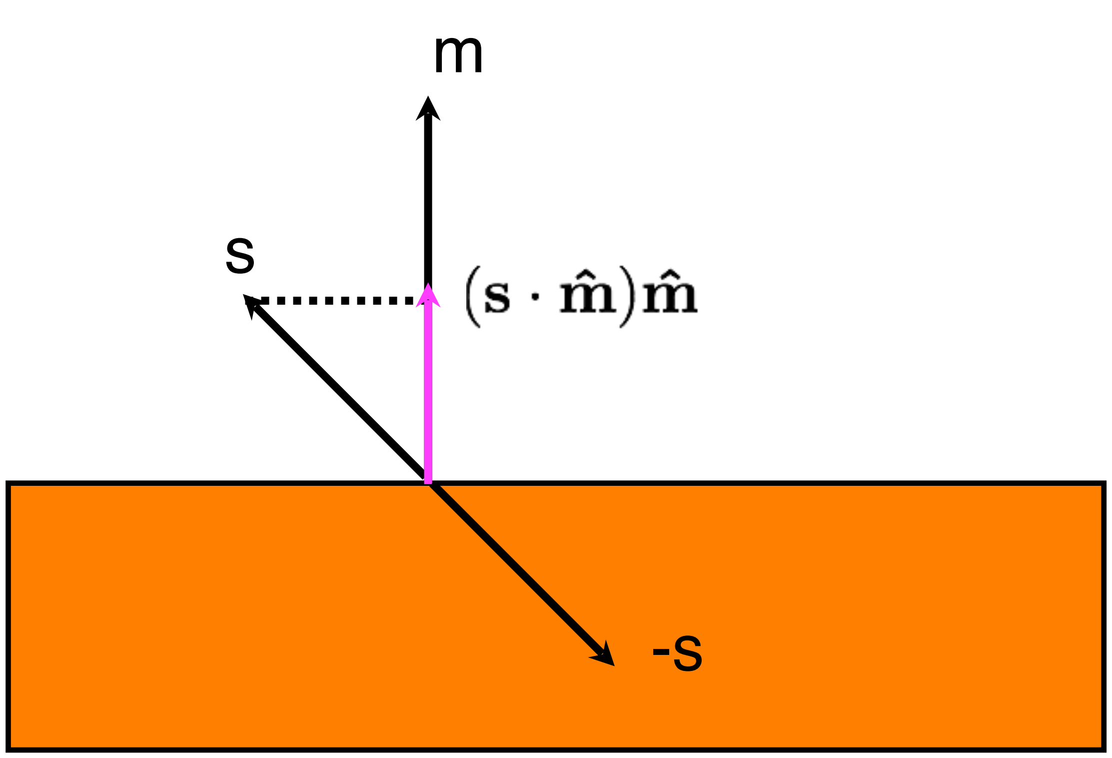
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

- Reflection is brightest around the reflection vector:

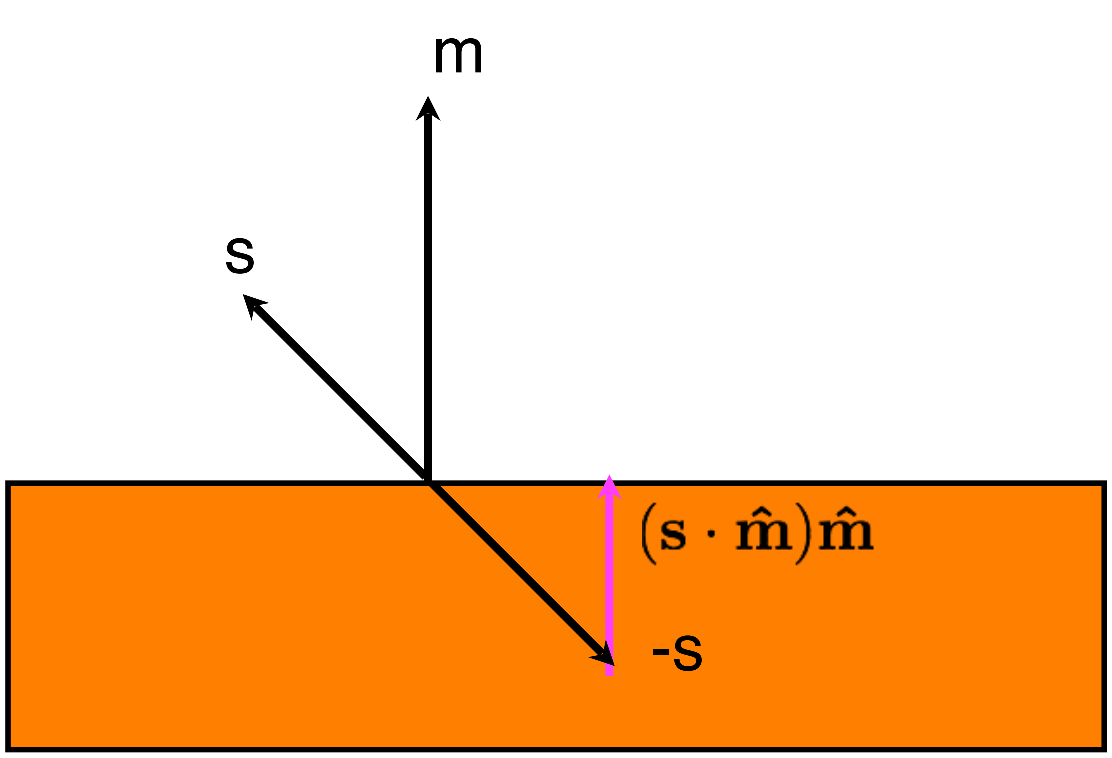
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s}\cdot\hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

- Reflection is brightest around the reflection vector:

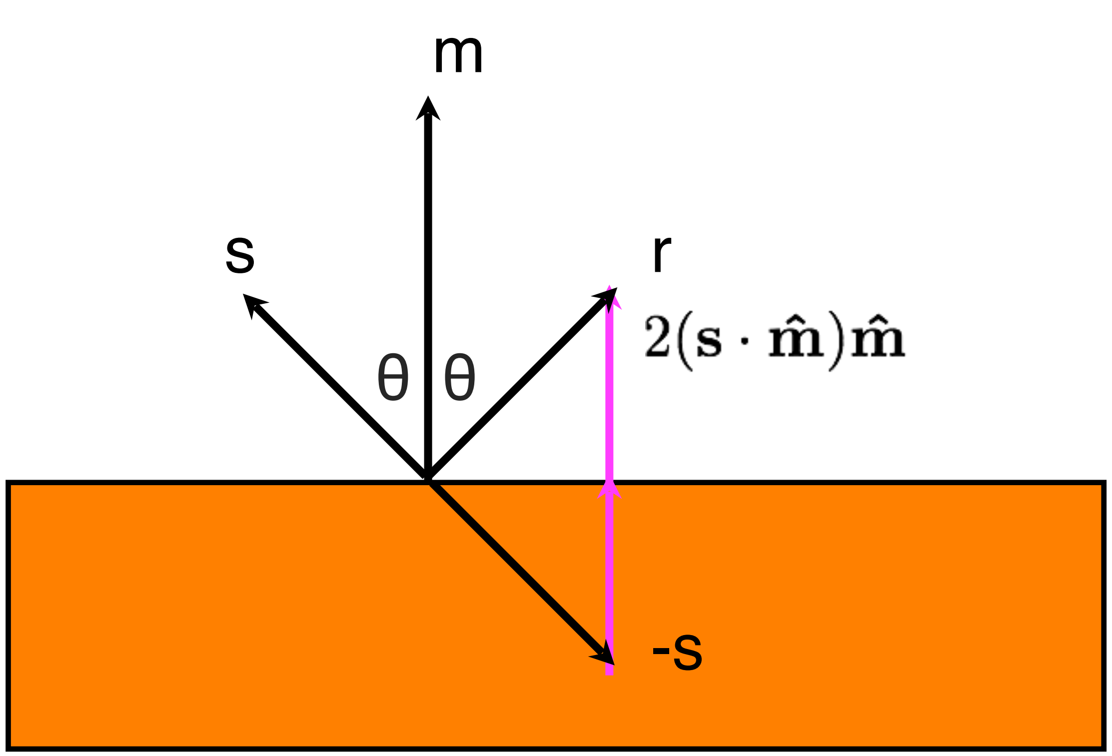
$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

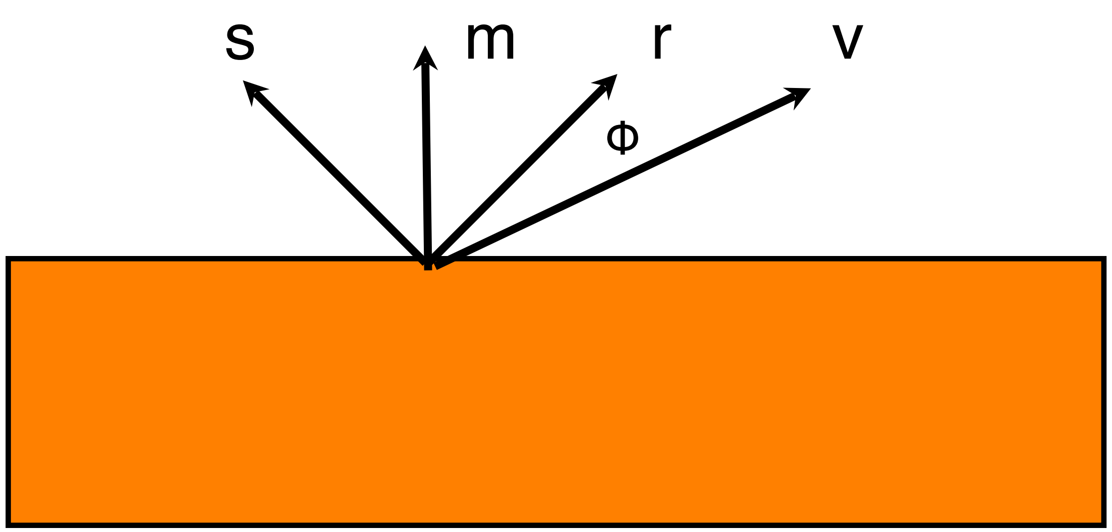
- Reflection is brightest around the reflection vector:

$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \hat{\mathbf{m}})\hat{\mathbf{m}}$$



Phong model

- The intensity falls off with the angle Φ between the reflected vector and the view vector (vector towards camera).



Phong model

- The Phong equation is:

$$
\begin{aligned}
I_{sp} &\propto I_s(\cos(\phi)^f) \\
&= \max(0, I_s \rho_{sp} (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^f)
\end{aligned}
$$

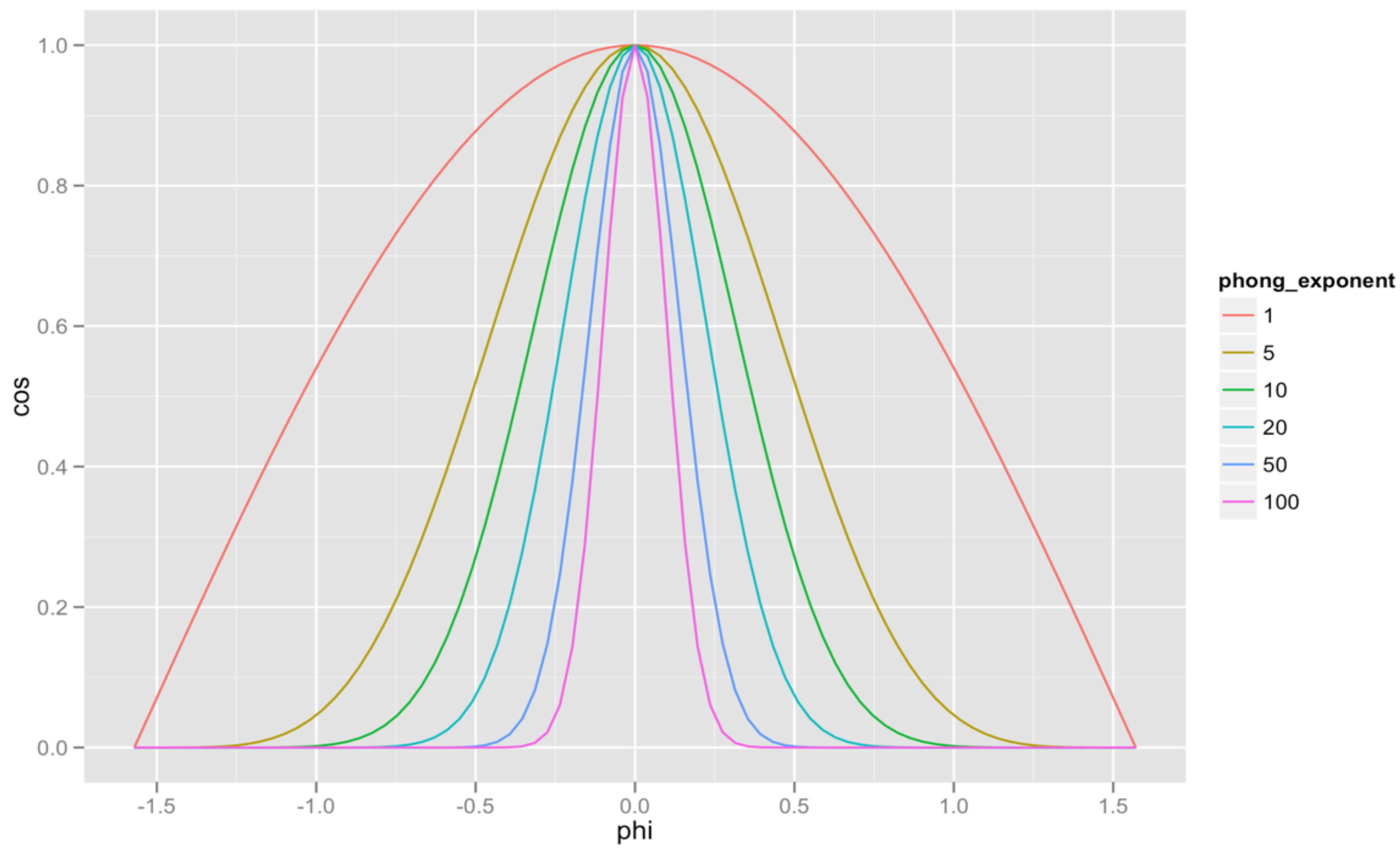
where:

ρ_{sp} is the specular reflection coefficient in the range [0,1]

f is the phong exponent, typically in the range [1,128]

Phong exponent

- Larger values of the Phong exponent f make $\cos(\Phi)^f$ smaller, produce less scattering, creating more mirror-like surfaces.



Blinn Phong Model

- The Blinn-Phong model uses a vector halfway between the source and the viewer instead of calculating the reflection vector.

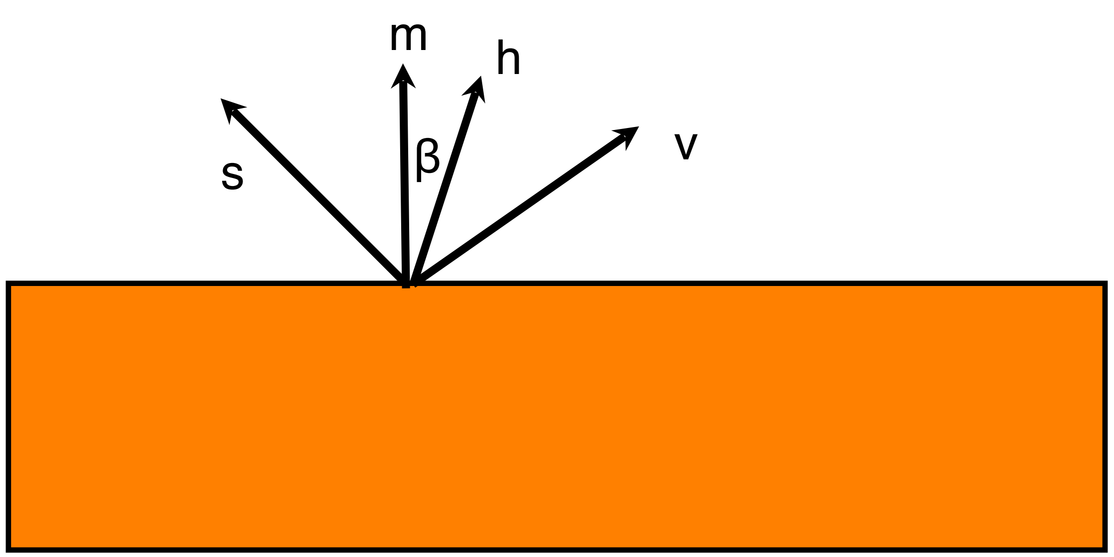
- Experimentally it has been found to produce more accurate reflections

  - https://people.csail.mit.edu/wojciech/BRDFValidation/index.html

Blinn-Phong Specular Light

- Find the halfway vector $\mathbf{h} = \dfrac{\hat{\mathbf{s}} + \hat{\mathbf{v}}}{2}$
- Then the angle B between h and m approximately measures the falloff of intensity

$$I_{sp} = \max(0, I_s \rho_{sp} (\hat{\mathbf{h}} \cdot \hat{\mathbf{m}})^f)$$



Reflection

- Note that the Phong/Blinn Phong model only reflects light sources, not the environment.

- It is good for adding bright highlights but cannot create a true mirror.

- Proper reflections are more complex to compute (as we'll see later).

Ambient light

- Lighting with just diffuse and specular lights gives very stark shadows.
- In reality shadows are not completely black.
- Light is coming from all directions, reflected off other objects, not just from ‘sources’
- It is too computationally expensive to model this in detail.

Ambient light

- The solution is to add an ambient light level to the scene for each light:

$$I_{ambient} = I_a \rho_a$$

- where:

    I_a is the ambient light intensity

    ρ_a is the ambient reflection coefficient

        in the range (0,1) (usually $\rho_a = \rho_d$)

Combining Light Contributions

- For a particular light source at a vertex

$$I = I_{ambient} + I_d + I_{sp}$$

$$I = I_a \rho_a + \max(0, I_s \rho_d (\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})) + \max(0, I_s \rho_{sp} (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^f)$$

Demo

- ModelViewer.java shows how we can use the lighting equations to generate very realistic looking renders in realtime
  - apple has ~1,700 triangles
  - bunny has ~70,000 triangles
  - dragon1 has ~871,000 triangles
  - dragon2 has 7,220,000 triangles!

Models

- The bunny and apple meshes are included with UNSWgraph. The dragons are too big. You can download them here

https://www.dropbox.com/s/tg2y5kvzbgb3pco/big.zip?dl=1

- More models are available here

https://people.sc.fsu.edu/~jburkardt/data/ply/ply.html

Limitations

- It is only a local model.
- Colour at each vertex V depends only on the light properties and the material properties at V
- It does not take into account
  - whether V is obscured from a light source by another object or shadows
  - light that strikes V having bounced off other objects

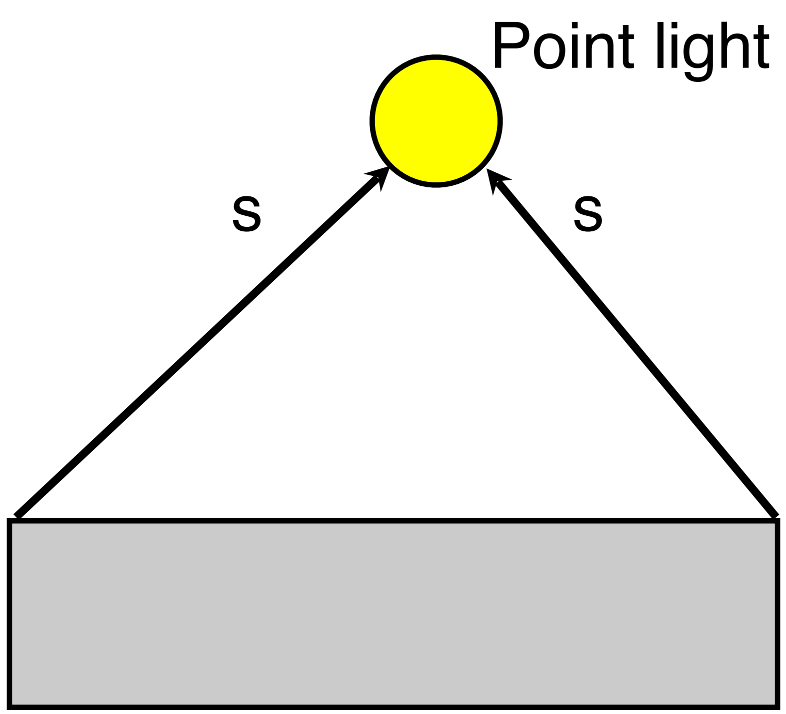
Combining all Light Sources

- If we have multiple lights, we add the ambient, diffuse and specular components from all of them.

$$I = \sum_{l \in lights} I_a^l \rho_a + \max(0, I_s^l \rho_d (\hat{\mathbf{s}} \cdot \hat{\mathbf{m}})) + \max(0, I_s^l \rho_{sp} (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^f)$$

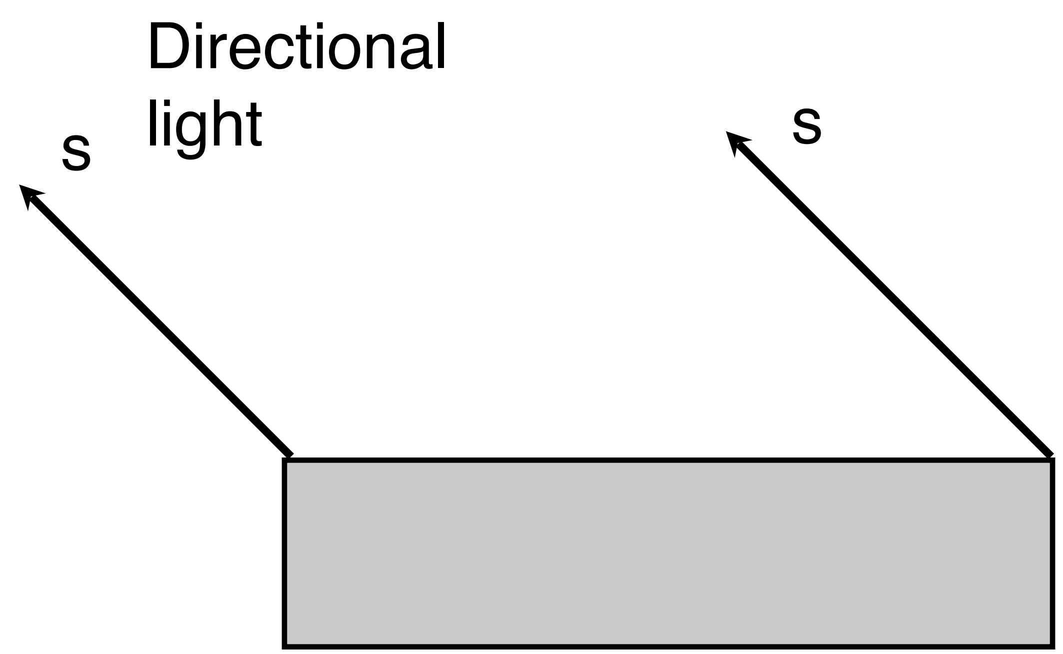
Point and directional lights

- We have assumed so far that lights are at a point in the world, computing the source vector from this.
- These are called point lights



Directional lights

- Some lights (like the sun) are so far away that the source vector is effectively the same everywhere.
- These are called directional lights.

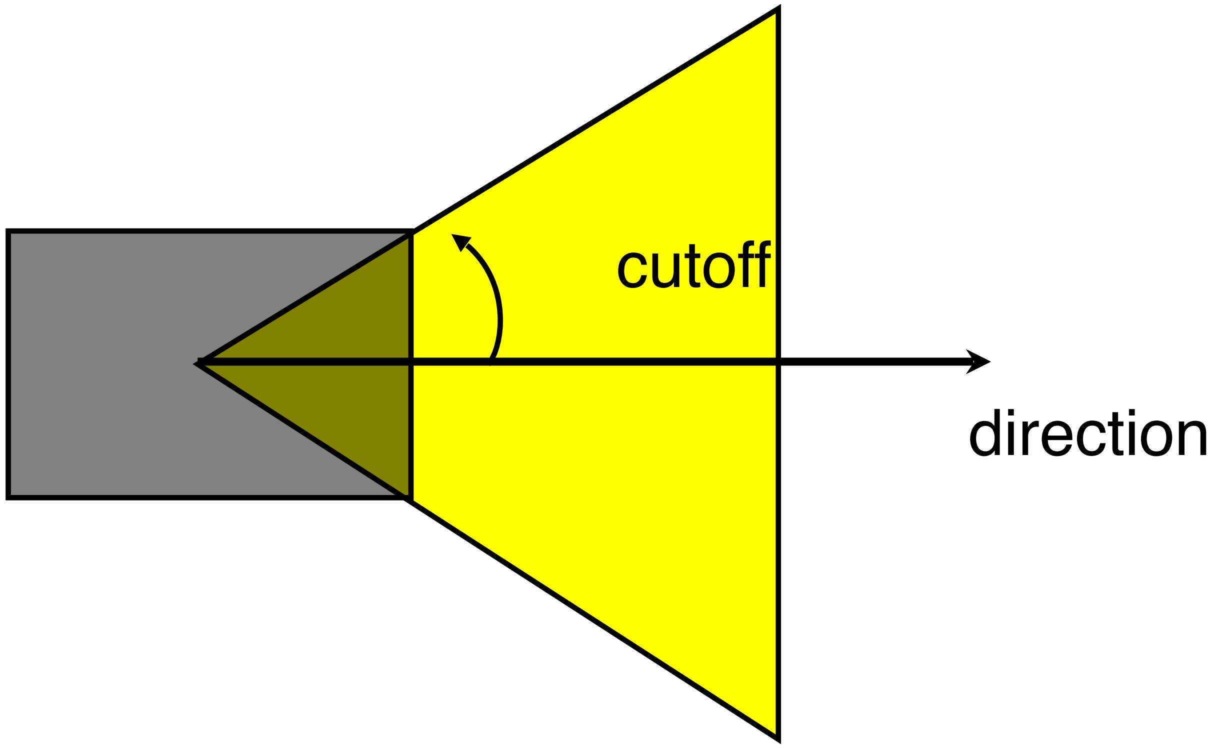


Moving Lights

- To make a light move with an object in the scene make sure it is subject to the same modelling transformation as the object

Spotlights

- Point sources emit light equally in all directions.
- For sources like headlights or torches it is more appropriate to use a spotlight.
- A spotlight has a direction and a cutoff angle,



Spotlights

- Spotlights are also attenuated, so the brightness falls off as you move away from the centre.

$$I = I_s(\cos(\beta))^{\varepsilon}$$

- where ε is the attenuation factor (exponent)