Tutorial 7 – Computer Graphics CSC418/2504
Illumination, Shading, and Colour

Remember:
We're talking about a simple *local* model of illumination, where we can compute shading for each polygon *independently* based on:

- material properties of the polygon
- orientation of the polygon (e.g. normals for faces and vertices)
- positions and parameters of the lights

More complicated *global* models of illumination also consider light inter-reflected *between* polygons. And ray-tracing methods can be used to model mirrored surfaces, refraction, etc. And there are other more advanced models.

1) What is the difference between ambient, diffuse, and specular reflection?

Ambient
- Approximates the effect of inter-reflections
- Sourceless – constant over entire surface
- Does not depend on surface normal
- Does not vary based on viewpoint

Diffuse
- Models rough surfaces (e.g. paper or drywall) – where light scatters equally in all directions
- Has a point or directional source
- Depends on surface normal – brightest where the surface is oriented toward the light, and falls off to zero at 90°
- Does not vary based on viewpoint

Specular
- Models highlights from smooth, shiny surfaces (e.g. opaque plastic)
- Has a point or directional source
- Depends on surface normal
- Depends on viewpoint

The Phong model puts these three terms together:

$$I_a k_a + \sum_{i=1}^{lights} \left[I_i k_{diff} (N \cdot L) + I_i k_{spec} (R \cdot V)^n \right]$$

2) Exercise: Light a triangle using the Phong Illumination model

$P_1 = (1,1,1)^T$	$k_a = 0.7$	white ambient intensity = 0.1
$P_2 = (0,2,1)^T$	$k_{diff} = 0.9$	white point light
$P_3 = (0,0,1)^T$	$k_{spec} = 0.6$	- position = $(1,1,5)^T$
	$n = 10$	- intensity = 0.5
viewer = $(1,2,5)^T$		

What's the intensity at the centroid of the triangle, $P = (0.333,1,1)^T$?
The following assumes a white object (r,g,b) = (1,1,1)
Because the light is white, the intensity will be the same for each colour channel (r,g,b)

Ambient $I_a k_a = 0.1(0.7) = 0.07$

Diffuse
$$N = (P_1-P_3) \times (P_2-P_3)$$
$$= (1,1,0)^T \times (0,2,0)^T$$
$$= (0,0,1)^T$$
$$L = (1,1,5)^T - (0.333,1,1)^T$$
$$= (0.164,0,0.986)^T \quad \text{(normalized)}$$
$$I_i k_{diff}(N \cdot L) = 0.5(0.9)(0.986) = 0.444$$

Specular
$$R = 2N(N \cdot L) - L$$
$$= 2(0,0,1)^T[0.986] - (0.164,0,0.986)^T$$
$$= (-0.164,0,0.986)$$
$$V = (1,2,5)^T - (0.333,1,1)^T$$
$$= (0.160,0.239,0.958)^T \quad \text{(normalized)}$$
$$R \cdot V = 0.971$$
$$I_i k_{spec}(R \cdot V)^n = 0.5(0.6)(0.971)^{10}$$
$$= 0.5(0.6)(0.745)$$
$$= 0.224$$

Total
$$I = 0.07 + 0.444 + 0.224$$
$$= 0.738$$
(if you were to get a value higher than 1.0, clamp it to 1.0)

What if the object were coloured?
The light reflected to the viewer is just a multiplication of

- incident light
- albedo (colour of the surface)

for every colour channel, (r,g,b).

For this example the incident light is 0.738*(1,1,1) – since the light is white

If the object, for example, were dark red (r,g,b) = (0.5,0,0), then the light reflected from P would be (0.5,0,0)·(0.738, 0.738, 0.738) = (0.369,0,0).

What if we wanted a different specular colour?
Okay, just apply a different colour to the specular term in the lighting model.

3) Shading

Flat shading

- Entire surface (polygon) has one colour
- Cheapest to compute, and least accurate (so you need a dense triangulation for decent-looking results)
- OpenGL – glShadeModel(GL_FLAT)

Phong shading

- Compute illumination for every pixel during scan conversion
- Interpolate normals at each pixel too
- Expensive, but more accurate
- Not supported in OpenGL (directly)

Gouraud shading

- Just compute illumination at vertices
- Interpolate vertex colours across polygon pixels
- Cheaper, but less accurate (spreads highlights)
- OpenGL - glShadeModel(GL_SMOOTH)

Phong illumination

- Don’t confuse shading and illumination!
- Shading describes how to apply an illumination model to a polygonal surface patch
- All these shading methods could use Phong illumination (ambient, diffuse, and specular) or any other local illumination model

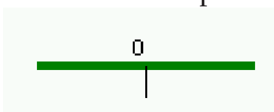
BSP trees

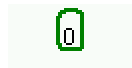
[Hill: 707-711. Foley & van Dam: p. 675-680]

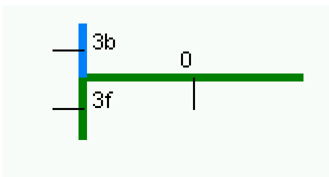
- *binary space partition* –object space, produces back-to-front ordering
- preprocess scene once to build BSP tree
- traversal of BSP tree is view dependent

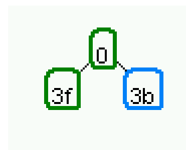
```
BSPtree *BSPmaketree(polygon list) {
   choose a polygon as the tree root
   for all other polygons
      if polygon is in front, add to front list
      if polygon is behind, add to behind list
      else split polygon and add one part to each list
   BSPtree = BSPcombinetree(BSPmaketree(front list),
                            root,
                            BSPmaketree(behind list) )
}
```

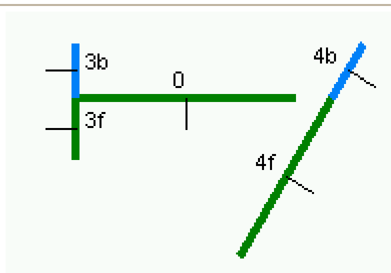
```
DrawTree(BSPtree) {
   if (eye is in front of root) {
        DrawTree(BSPtree->behind)
        DrawPoly(BSPtree->root)
        DrawTree(BSPtree->front)
   } else {
        DrawTree(BSPtree->front)
        DrawPoly(BSPtree->root)
        DrawTree(BSPtree->behind)
   }
}
```

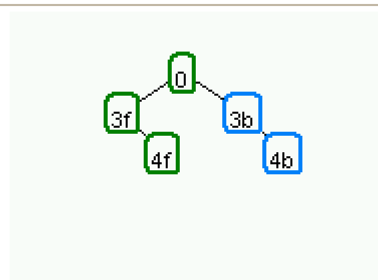
<table>
<tr>
<td>First, create a root node and partition plane.


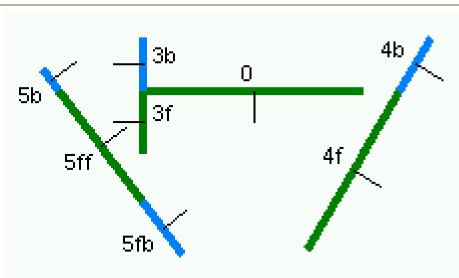
</td>
<td>Obviously the root does not have any children.


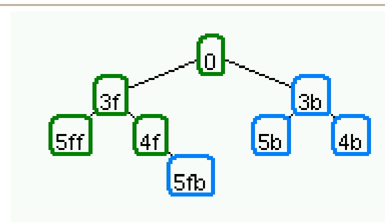
</td>
</tr>
<tr>
<td>


</td>
<td>


</td>
</tr>
<tr>
<td>


</td>
<td>


</td>
</tr>
<tr>
<td>


</td>
<td>


</td>
</tr>
</table>

We work through drawing the BSP from a point in the scene, following the algorithm.
Example: from a point in the extreme lower-right corner:

behind(0) 0 front(0)
front(3b) 3b behind(3b) 0 front(0)
….
5b 3b 4b 0 5ff 3f 5fb 4f