Shading Models

There are two main types of rendering that we cover,

- polygon rendering
- ray tracing

Polygon rendering is used to apply illumination models to polygons, whereas ray tracing applies to arbitrary geometrical objects. Ray tracing is more accurate, whereas polygon rendering does a lot of fudging to get things to look real, but polygon rendering is much faster than ray tracing.

- With polygon rendering we must approximate NURBS into polygons, with ray tracing we don't need to, hence we can get perfectly smooth surfaces.
- Much of the light that illuminates a scene is indirect light (meaning it has not come directly from the light source). In polygon rendering we fudge this using ambient light. Global illumination models (such as ray tracing, radiosity) deal with this indirect light.
- When rendering we assume that objects have material properties which we denote $k_{(property)}$.
- We are trying to determine I which is the colour to draw on the screen.

We start with a simple model and build up,

Lets assume each object has a defined colour. Hence our illumination model is $I = k_i$, very simple, looks unrealistic.

Now we add ambient light into the scene. Ambient Light is indirect light (ie. did not come straight from the light source) but rather it has reflected off other objects (from diffuse reflection). $I = I_a k_a$. We will just assume that all parts of our object have the same amount of ambient light illuminating them for this model.

Next we use the diffuse illumination model to add shading based on light sources. This works well for non-reflective surfaces (matte, not shiny) as we assume that light reflected off the object is equally reflected in every direction.

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)."

Gouraud Shading

Use normals at each vertex to calculate the colour of that vertex (if we don't have them, we can calculate them from the polygon normals for each face). Do for each vertex in the polygon and interpolate the colour to fill the polygon. The vertex normals address the common issue that our polygon surface is just an approximation of a curved surface.

To use gouraud shading in OpenGL use glShadeModel(GL_SMOOTH). But we also need to define the vertex normals with glNormal3f() (which will be set to any glVertex that you specify after calling glNormal).

Highlights don't look realistic as you are only sampling at every vertex.

Interpolated shading is the same, but we use the polygon normal as the normal for each vertex, rather than the vertex normal.

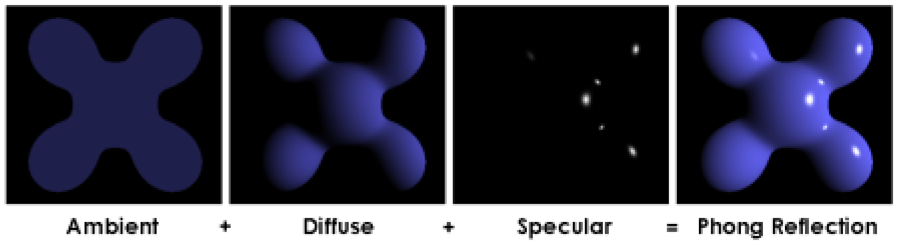
Phong Shading

Like gouraud, but you interpolate the normals and then apply the illumination equation for each pixel.

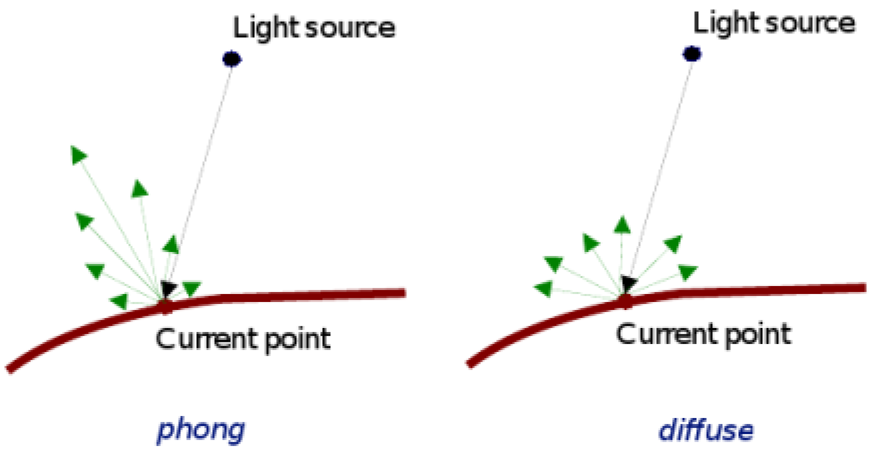
This gives much nicer highlights without needing to increase the number of polygons, as you are sampling at every pixel.

Phong Illumination Model

Diffuse reflection and specular reflection.



Components of the Phong Model (Brad Smith,http://commons.wikimedia.org/wiki/File:Phong_components_version_4.png)



(Source: COMP3421, Lecture Slides.)
$I_s = I_l k_s \cos^n(\alpha)$

n is the Phong exponent and determines how shiny the material (the larger n the smaller the highlight circle).

Flat shading. Can do smooth shading with some interpolation.

If you don’t have vertex normals, you can interpolate it using the face normals of the surrounding faces.

Gouraud interpolates the colour, phong interpolates the normals.

Attenuation

inverse square is physically correct, but looks wrong because real lights are not single points as we usually use in describing a scene, and

For now I assume that all polygons are triangles. We can store the normal per polygon. This will reneder this polygon, but most of the time the polygon model is just an approximation of some smooth surface, so what we really want to do is use vertex normals and interpolate them for the polygon.

Ray Tracing

For each pixel on the screen shoot out a ray and bounce it around the scene. The same as shooting rays from the light sources, but only very few would make it into the camera so its not very efficient.

Each object in the scene must provide an intersection(Line2D) function and a normal (Point3D) function

Ray Tree

Nodes are intersections of a light ray with an object. Can branch intersections for reflected/refracted rays. The primary ray is the original ray and the others are secondary rays.

Shadows

Can do them using ray tracing, or can use shadow maps along with the Z buffer. The key to shadow maps is to render the scene from the light's perspective and save the depths in the Z buffer. Then can compare this Z value to the transformed Z value of a candidate pixel.

==============

Rasterisation

Line Drawing

DDA

- You iterate over x or y, and calculate the other coordinate using the line equation (and rounding it).
- If the gradient of the line is > 1 we must iterate over y otherwise iterate over x. Otherwise we would have gaps in the line.
- Also need to check if x1 is > or < x2 or equal and have different cases for these.

Bresenham

- Only uses integer calcs and no multiplications so its much faster than DDA.
- We define an algorithm for the 1st octant and deal with the other octant's with cases.
- We start with the first pixel being the lower left end point. From there there are only two possible pixels that we would need to fill. The one to the right or the one to the top right. Bresenham's algorithm gives a rule for which pixel to go to. We only need to do this incrementally so we can just keep working out which pixel to go to next.
- The idea is we accumulate an error and when that exceeds a certain amount we go up right, then clear the error, other wise we add to the error and go right.
  We use Bresenham's algorithm for drawing lines this is just doing linear interpolation, so we can use Bresenham's algorithm for other tasks that need linear interpolation.

Polygon Filling

Scan line Algorithm

The Active Edge List (AEL) is initially empty and the Inactive Edge List (IEL) initially contains all the edges. As the scanline crosses an edge it is moved from the IEL to the AEL, then after the scanline no longer crosses that edge it is removed from the AEL.

To fill the scanline,

- On the left edge, round up to the nearest integer, with round(n) = n if n is an integer.
- On the right edge, round down to the nearest integer, but with round(n) = n-1 if n is an integer.

Its really easy to fill a triangle, so an alternative is to split the polygon into triangles and just fill the triangles.

===============

Anti-Aliasing

Ideally a pixel's colour should be the area of the polygon that falls inside that pixel (and is on top of other polygons on that pixel) times the average colour of the polygon in that pixel region then multiply with any other resulting pixel colours that you get from other polygons in that pixel that's not on top of any other polygon on that pixel.

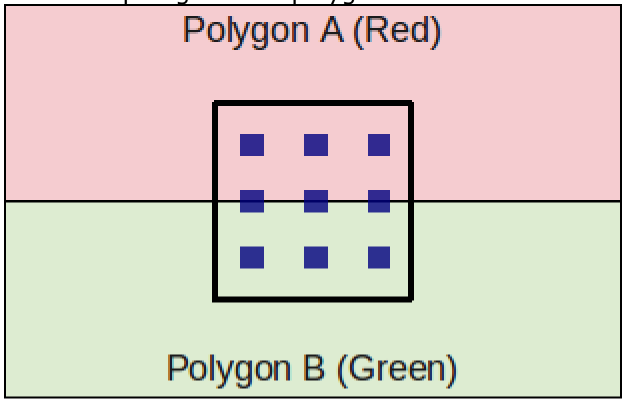
Aliasing Problems

- Small objects that fall between the centre of two adjacent pixels are missed by aliasing. Anti-aliasing would fix this by shading the pixels a gray rather than full black if the polygon filled the whole pixel.
- Edges look rough ("the jaggies").
- Textures disintegrate in the distance
- Other non-graphics problems.

Anti-Aliasing

In order to really understand this anti-aliasing stuff I think you need some basic understanding of how a standard scene is drawn. When using a polygon rendering method (such as is done with most real time 3D), you have a framebuffer which is just an area of memory that stores the RGB values of each pixel. Initially this framebuffer is filled with the background colour, then polygons are drawn on top. If your rending engine uses some kind of hidden surface removal it will ensure that the things that should be on top are actually drawn on top.

Using the example shown (idea fromhttp://cgi.cse.unsw.edu.au/~cs3421/wordpress/2009/09/24/week-10-tutorial/#more-60), and using the rule that if a sample falls exactly on the edge of two polygons, we take the pixel is only filled if it is a top edge of the polygon.



Anti-Aliasing Example Case. The pixel is the thick square, and the blue dots are samples.

No Anti-Aliasing

With no anti-aliasing we just draw the pixel as the colour of the polygon that takes up the most area in the pixel.

Pre-Filtering

- We only know what colours came before this pixel, and we don't know if anything will be drawn on top.
- We take a weighted (based on the ratio of how much of the pixel the polygon covers) averages along the way. For example if the pixel was filled with half green, then another half red, the final anti-aliased colour of that pixel would determined by,
  Green (0, 1, 0) averaged with red (1, 0, 0) which is (0.5, 0.5, 0). If we had any more colours we would then average (0.5, 0.5, 0) with the next one, and so on.
- Remember weighted averages,
  $\frac{Aa+Bb}{A+B}$
  where you are averaging a and b with weights A and B respectively.
- Pre-filtering is designed to work with polygon rendering because you need to know the ratio which by nature a tracer doesn't know (because it just takes samples), nor does it know which polygons fall in a given pixel (again because ray tracers just take samples).
- Pre-filtering works very well for anti-aliasing lines, and other vector graphics.

Post-Filtering

- Post-filtering uses supersampling.
- We take some samples (can jitter (stochastic sampling) them, but this only really helps when you have vertical or horizontal lines moving vertically or horizontally across a pixel, eg. with vector graphics)
- $\left(\frac{6}{9}\right)$ of the samples are Green, and $\left(\frac{3}{9}\right)$ are red. So we use this to take an average to get the final pixel colour of $\left(\frac{1}{3}, \quad \frac{2}{3}, \quad 0\right)$
- We can weight these samples (usually centre sample has more weight). The method we use for deciding the weights is called the **filter**. (equal weights is called the *box filter*)
- Because we have to store all the colour values for the pixel we use more memory than with pre-filtering (but don't need to calculate the area ratio).
- Works for either polygon rendering or ray tracing.
  Can use adaptive supersampling. If it looks like a region is just one colour, don't bother supersampling that region.

OpenGL

Often the graphics card will take over and do supersamling for you (full scene anti aliasing).

To get OpenGL to anti-alias lines you need to first tell it to calculate alpha for each pixel (ie. the ratio of non-filled to filled area of the pixel) using, glEnable(GL_LINE_SMOOTH) and then enable alpha blending to apply this when drawing using,

```
glEnable(GL_BLEND);

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

You can do post-filtering using the accumulation buffer (which is like the framebuffer but will apply averages of the pixels), and jittering the camera for a few times using accPerspective.

Anti-Aliasing Textures

A texel is a texture pixel whereas a pixel in this context refers to a pixel in the final rendered image.

When magnifying the image can use bilinear filtering (linear interpolation) to fill the gaps.

Mip Mapping

Storing scaled down images and choose closes and also interpolate between levels where needed. Called trilinear filtering.

Rip Mapping helps with non uniform scaling of textures. Anisotropic filtering is more general and deals with any non-linear transformation applied to the texture

Double Buffering

We can animate graphics by simply changing the framebuffer, however if we start changing the framebuffer and we cannot change it faster than the rate the screen will display the contents of the frame buffer, it gets drawn when we have only changed part of the framebuffer. To prevent this, we render the image to an off screen buffer and when we finish we tell the hardware to switch buffers.

Can do on-demand rendering (only refill framebuffer when need to) or continuois rendeing (draw method is called at a fixed rate and the image is redrawn regardless of whether the image needs to be updated.)

LOD

Mip Mapping for models. Can have some low poly models that we use when far away, and use the high res ones when close up.

Animation

Key-frames and tween between them to fill up the frames.

===============

Shaders

OpenGL 2.0 using GLSL will let us implement out own programs for parts of the graphics pipeline particularly the vertex transformation stage and fragment texturing and colouring stage.

Fragments are like pixels except they may not appear on the screen if they are discarded by the Z-buffer.

Vertex Shaders

- position tranformation and projection (set gl_Position), and
- lighting calculation (set gl_FrontColor)

Fragment Shaders

- interpolate vertex colours for each fragment
- apply textures
- etc.

set gl_FragColor.

Source: http://andrewharvey4.wordpress.com/2009/12/02/computer-graphics-notes/