Topics covered:
HIdden Surface Removal

Shading

Hidden Surface Removal

usually implemented in two ways

binary space partitioning (BSP) tree algorithm

SIGGRAPH 1980 paper by Henry Fuchs (now at UNC)

zbuffering

Ed Catmull SIGGRAPh 1978 (now at Pixar)

BSP

order surfaces from front to back

key aspect: preprocessing to create data structure so that info is useful from any viewpoint

BSP tree algorithm is example of painter's algorithm

draw every object from back to front

potentially drawing over previous drawn polygons

sort objects back to front relative to viewpoint

for each object

draw object on screen

Problems cycle (3D objects)

Preprocess all polygons into BSP tree based up implicit plane equation.
Start thinking about this with 2 triangle (see slide 1)
for all points p+ on one sides of plane f1(p+) > 0
for all points p- on other side of plane f1(p-) < 0

use binrary tree data structure with T1 as root
negative banch contains all triangles who vertices have f_i(p) < 0
positive branch contains all triangles who vertices have f_i(p) > 0

```
draw(bsptree tree, point e)
  if (tree.empty) then
    return
  if (f_tree.root(e) < 0) then
    draw (tree.plus,e)
    rasterize tree.triangle
    draw (tree.minus, e)
  else
    draw(tree.minus, e)
```

rasterize tree.triangle
draw(tree.plus, e)

Plane equation
implicit equation for a point p on a plane containing three non-coliear points, a,b,c:
$f(p) = ((b-a) \times (c-a)) \cdot (p - a) = 0$

or $f(x,y,z) = Ax + By + Cz + D = 0$

where normal of triangle $= n = (A, B, C) = (b - a) \times (c - a)$

solve for D by plugging any point on the plane: say a

$D = -Ax_a - By_a - Cz_a = -n \cdot a$

$f(p) = n \cdot p - n \cdot a$
$= n \cdot (p-a)$
$= 0$

One catch: what about triangels that are not uniquely on one side of a plane or the other?

Answer: split triangles into smaller triangles using the plane to cut them

See Shirley book for more details

----

z buffering (also called depth-buffering)

found in hardware and almost every video game and PC graphics
also useful software algorithm
if our problem is pixel level (center of pixel)
easier than finding true depth order in continuous screen space

At each pixel store real z value (distance from pixel to eye of closest triangle rasterized so far)
Use barycentric coordinates to find z values

only write rgb to pixel and update z
if z stored in z buffer is larger than new z

Problems: memory

Assume for discussion,

z
near & far plane
all positive

Use non negative integers
Prefer true floats, but zbuffers need to be fast and cheap

However integers cause precision problems

map 0 to near plane
map B-1 to far plane
then bucket for zbuffer
    delta z = (f-n) / B

Let B = 2^b, where b is number of bits used to store z-values

need to have a delta z that doesn't cause precision problems,
ie if triangel have speration of at least one meter, then delta z < 1 should yield images without problems

Fix problems by moving n & f closer or increase b

b is usually fixed in most APIs and hardware, so adjusting n & f are only choices

Above assumes numbers after perspective

to get real 3D world depth precision:

z = n + f - fn/z_w

z_w = world depth

approximate bin size by differentiating both sides

delta z approx eq fn delta z_w / (z_w)^2

so world space

delta z_w approxeq (z_w)^2 delta z / (fn)

biggest bin is when derrivative of z = f,

so

max(delta z_w) = (f)^2 delta z / (fn)
            =   f delta z / n

choose n =0 natural choice if don't want to loose objects right in
fron of eye, however end up with infinitely large bin... very bad

to make max(delta z_w) as small as possible,
want to minimize f, maximize n

Shading

"surface is painted with light"

Diffuse & Phong (1970x)

Many objects surface appearance loosely described as "matte" ie not at all shiny

Examples include paper, unfinished wood, dry, unpolished stones

such object do not change color with a change in viewpoint

Matte objects behave like "Lambertian" bojects
obeys Lambert's law
color c proportional to cosine of angle between the surface normal and the direction to the light source

c proportional cos theta

c proportional n . l

light is assumed to be distant relative to size of object
directional light (no position, just facing)

surface lighter or darker by changing intensity of light source or the reflectance of the surface

diffuse reflectance c_r is fraction of light reflected by the surface

fraction is different for different color components
ie is surface is red it reflect higher fraction of red incident light than blue

if we assume that surface color is proportional to light reflected from a surface, then diffuse reflection (color of surface) must be included

c proportional c_r n . l

add effects o flight intensity (and color)

$c = c_r c_l \; n . l$

problem is that if all colors are on a scale of [0,1], this equation can produce # outside this range (dot product can be negative)

so

$c = c_r c_l \; \max(0, n.l)$ or

$c = c_r c_l \; | n . l |$ often called two-sided lighting

Ambient shading

Problem of any face facing away from light is black
doesn't happen in real world

(light reflected from other objects) and
ambient lighting (sky lighting)

common trick: dim light at eye
another: two sided lighting (can be non-intuitive for naive viewer)
more comon: add an ambient term

$c = c_r (c_a + c_l \; \max(0, n.l))$

think of c_a avg color of all surfaces in scene.

problems in code: $c_a + c_l <= (1,1,1)$ or
clamp (common)

Result is faceted shading of triangulated objects
solution: add normals to vertices, calc c at each vertex
interpolate across polygon for smooth shading

How to get normals of faceted polygonal models: average
Normalize light vectors!

Specular

Problem with this shading: not all surfaces are matte

they have highlights
tile floors, apples, gloss paint, white boards

highlights move as viewpoint moves
need to know vector e to equation

highlights are reflections of th light, color of the light

placing highlight, where reflection of light would be, call it r
r is the reflected light from the light
so given vector l, make angle to normal on surface (theta),
r is also theta off of n

function for highlight: bright where e = r, falls off gradually

think about c = c_l (e . r)
problem: range of values can be negative; too wide
solution: narrow by rasing to power & add max term

c = c_l max(0, e . r)^p

p is called the Phong Exponent
 a positive real number

(Need image of examples)

Calculating r

r = -l + 2(l . n)n

(l.n) gets you the cos theta, so r = -l + 2* (n cos theta)

An alternative uses half way vector, h, a unit vector halfway between L and e
 h = (e + l) / (ll e + l ll)

highlight occurs when h is near n, ie when cos phi = h . n is near 1

c = c_l (h . n)^p

this is different than using r. angle between h and n is half size of angle between e and r

advantage of using h: alway s positive for eye and ligh above plane

in practive3 want diffuse and a highlight so:

$c = c_r (c_a + c_l \max(0,n.l)) + c_l (h . n)^p$

we can dim and color the highlight by using c_p:

$c = c_r (c_a + c_l \max(0,n.l)) + c_l c_p (h . n)^p$

good for coding: $c_p = 1 - M$ where M is max component of c_r to keep color below one for light source and no ambient term

----

Gouraud & Phong Shading

Interpolate normals, calculate lighting at each vertex, then
interpolate colors
this is Gouraud shading (Communication of ACM June 1971)
(intensity interpolation shading or color interpolation shading)
but lighting chages too fast.. what happens if
highlight lands on one vertex? what if highlights lands in middle of polygon

Phong shading (Communications of the ACM June 1975)
(normal vector interpolation shading)
so instead beter to interpolate normals and vertex colors,
interpolate a normal and color for each pixel,
then light the pixel

Lighting in OpenGL

Specifing the lights thru
4 independent components
emissive, ambient, diffuse and specular
computed independently and then added together

ambient: light been scattered by environment, origin unknown
- light added to entire scene, regardless of viewpoint

diffuse: light comes from one direction
brighter if comes squarely down on surface than if it arely glances off the surface
once it hits a surface, it is scattered equally in all directions
(ie viewpoint doesn't change this)
- Color of the light

specular light: particular direction

tens to bounc off in a preferred direction
(shinniness)

each of these is represented by a color

for if have white light bouncing off a blue room, the scattered light
tends to be blue, but light directly htiting surface is white

--

OpenGL Object Materials:
determine what surface of object is made of

red ball -> reflects incomiing red light, absorbs all green and blue light that hits it
in white light: appears red
in red light: appears red
in blue light: appears? .... black! no light is reflected

materials are specified by emissive, ambient, diffuse and specular too
ambient & diffuse are similar if not the same
specular is usually white or grey so the specular highlighgts end up being the color of the light source's specular intensity

emmisive: simulate light originating from object
doesn't add light to other objects

Start lightmaterial demo
```
cd ~/Teaching/CS351_IntroCG/NatesTutors]
./lightmaterial
```

Diffuse

diffuse plays most important role in determining perceived color of object
it is affected by color of incident diffuse light and the angle of the incident light relative to normal of surface... most intense when incident light falls perpendicular to surface (parallel to srf normal)
not affected by position of light

Amient

ambient reflectance affects the overall color of the object
most noticable when object receives no direct illumination
affected by global ambient light and ambient light from individual light sources
no affected by position of light

for real world objects, diffuse and ambient reflectance are normally the same color

```
GLfloat mat_amb_diff[] = { 0.1, 0.5, 0.8, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
        mat_amb_diff);
```

Specular reflection

dependent upon location of viewpoint

effect that the material has on reflected light (with GL_SPECULAR) and

control the size and brightness of the highlight (with GL_SHININESS).

[0.0, 128.0] for GL_SHININESS

the higher the value, the smaller and brighter (more focused) the highlight.

Emission

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

make light appear to give off color

Note: most real objects (except lights) don't emit light

useful for simulating lamps or other light sources in a scene

-----

glColorMaterial

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
/* now glColor* changes diffuse reflection  */
glColor3f(0.2, 0.5, 0.8);
/* draw some objects here */
glColorMaterial(GL_FRONT, GL_SPECULAR);
/* glColor* no longer changes diffuse reflection  */
/* now glColor* changes specular reflection  */
glColor3f(0.9, 0.0, 0.2);
/* draw other objects here */
glDisable(GL_COLOR_MATERIAL);
```

You should use glColorMaterial() when only need to change a single material parameter

if need to change more than one material parameter, use glMaterial*()

Creating light sources

OpenGL: usually at most can only specify 8 lights; performance hit for # of lights

color
position
direction

start lighting demo
    cd ~/Teaching/CS351_IntroCG/NatesTutors
    ./lightposition

directional light
infinite location rays of light are parallel by time reach object
ex: sun

positional source
or can make it a
position & directional source (see pg 183)
ex: desk lamp

Attenuation
real world lights attenuate, ie less light reaches farther objects

attenuation factor: 1 / (k_c k_l d   +   k_q   d^2)

d is distance to light to vertex
k_c = =constant attenuation (default to 1)
k_l = linear attenuation (default to 0)
k_q = quadratic attenuation (default to 0)

Spot light
cone of light
specify angle between the axis of the cone and a ray along the edge of the cone (half angle [0,90])
specify direction in object coordinates
can set attenuation factor
can set GL_SPOT_EXPONENT to control how concentrated the light is
  highest in center of cone

Code Examples
(see powerpoint slides)

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>

void init(void)
{
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 50.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);

   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glutSolidSphere (1.0, 20, 16);
   glFlush ();
}

void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
         1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
         1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char** argv)
{
```

```
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

Add Spot light

```
GLfloat light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat light1_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_position[] = { -2.0, 2.0, 1.0, 1.0 };
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };

glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.2);

glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);

glEnable(GL_LIGHT1);
```

Example 5-4 : Stationary Light Source

```
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
if (w <= h)
    glOrtho (-1.5, 1.5, -1.5*h/w, 1.5*h/w, -10.0, 10.0);
else
```

```
    glOrtho (-1.5*w/h, 1.5*w/h, -1.5, 1.5, -10.0, 10.0);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity();

/* later in init() */
GLfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

As you can see, the viewport and projection matrices are established first. Then, the identity matrix is loaded as the modelview matrix, after which the light position is set. Since the identity matrix is used, the originally specified light position (1.0, 1.0, 1.0) isn't changed by being multiplied by the modelview matrix. Then, since neither the light position nor the modelview matrix is modified after this point, the direction of the light remains (1.0, 1.0, 1.0).

Example 5-5 : Independently Moving Light Source

```
static GLdouble spin;

void display(void)
{
    GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
        gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
        glPushMatrix();
            glRotated(spin, 1.0, 0.0, 0.0);
            glLightfv(GL_LIGHT0, GL_POSITION, light_position);
        glPopMatrix();
        glutSolidTorus (0.275, 0.85, 8, 15);
    glPopMatrix();
    glFlush();
}
```

Light is rotated around torus, eyepoint is unchanged

OpenGL Lighting Equation

color produced by lighting a vertex:
vertex color = material emission at the vertex +
        global ambient light scaled by material's ambient property at vertex +

ambient, diffuse, and specular contributions from all light source, properly attenuated

color clamped to [0,1]

vertex color = emission_material +

ambient_(light model) * ambient_material +

sum (i=0 to n-1) (1/(k_c + k_i*d + k_q d^2)*(spotlight effect)_i *

[ambient_light *ambient_material +

(max { L · n , 0}) * diffuse_light * diffuse_material +

(max { s · n , 0})shininess * specular_light * specular_material]_i

----

Artistic Shading

Taking this a step further, modivate artistic shading

Silhouettes

draw silhouette if (e.n1)(e.n2)<= 0

Cool to Warm shading:

think about the sun and the sky

warm light & use the cosine to modulate color
warmth constant

k_w = (1+(n . l))/2

c = k_w c_w + (1 - k_w)c_c

Toon Shading

Examples:

Nvidia: developer.nvidia.com/object/Toon_Shading.html

Blender: w3imagis.imag.fr/Membres/Jean-Dominique.Gascuel/DEAIVR/

Cours2002/17%20janvier/Blender-tutorial80.pdf

Intel: http://www.intel.com/labs/media/3dsoftware/nonphoto.htm