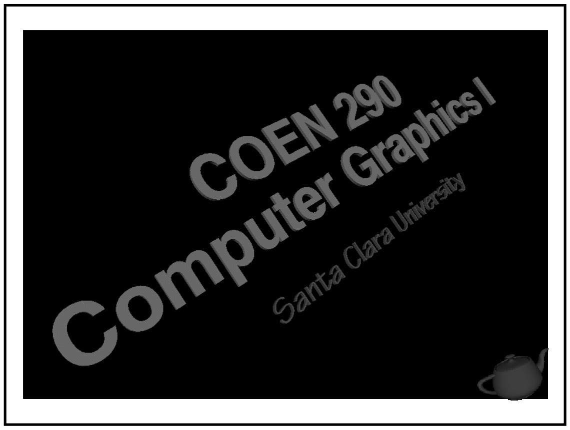


COEN 290
Computer Graphics I
Santa Clara University

Evening's Goals

- Discuss the fundamentals of lighting in computer graphics
- Analyze OpenGL's lighting model
- Show basic geometric rasterization and clipping algorithms

Simulating Lighting In CGI

- Lighting is a key component in computer graphics
- Provides cues on:
  - shape and smoothness of objects
  - distance from lights to objects
  - objects orientation in the scene
- Most importantly, helps CG images look more realistic

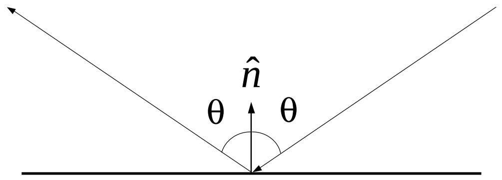
Lighting Models

- Many different models exist for simulating lighting reflections
  - we’ll be concentrating on the *Phong* lighting model
- Most models break lighting into constituent parts
  - *ambient* reflections
  - *diffuse* reflections
  - *specular* highlights

Lighting Model Components

- *Material* Properties
  - used to describe an objects reflected colors
- *Surface Normals*
- *Light* Properties
  - used to describe a lights color emissions
- *Light Model* Properties
  - “global” lighting parameters

Physics of Reflections



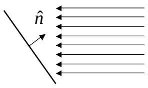
Ambient Reflections

- Color of an object when not directly illuminated
  - light source not determinable
- Think about walking into a room with the curtains closed and lights off

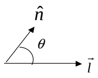
$$I_a = g_a + \sum_i l_{ia} \cdot m_a$$

Diffuse Reflections

- Color of an object when directly illuminated
  - often referred to as *base color*

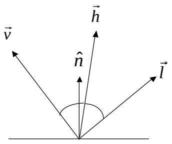


$$I_d = \sum_i l_{i_d} \cdot m_d \left(\hat{l} \bullet \hat{n}\right)$$



Specular Reflections

- Highlight color of an object
- *Shininess* exponent used to shape highlight



$$I_s = \sum_i l_{i_s} \cdot m_s \left(\hat{n} \bullet \hat{h}\right)^s$$

$\hat{h} = \frac{1}{2}\left(\hat{l} + \hat{v}\right)$

Phong Lighting Model

- Using surface normal
- OpenGL's lighting model based on Phong's

$$I = I_a + I_d + I_s$$

OpenGL Material Properties

- `GL_AMBIENT`
- `GL_DIFFUSE`
- `GL_SPECULAR`
- `GL_SHININESS`
- `GL_EMISSION`

Setting Material Properties

`glMaterial[fd]v``(` *`face, prop, params`* `);`

- *face* represents which side of a polygon
  - `GL_FRONT`
  - `GL_BACK`
  - `GL_FRONT_AND_BACK`
- polygon facedness controlled by **`glFrontFace`**`()`

OpenGL Lights

- OpenGL supports at least eight simultaneous lights
  - `GL_LIGHT0 - GL_LIGHT[n-1]`
- Inquire number of lights using

```
glGetIntegerv( GL_MAX_LIGHTS, &n );

glLight[fd]v( light, property, params );
```

OpenGL Light Color Properties

- `GL_AMBIENT`
- `GL_DIFFUSE`
- `GL_SPECULAR`

Types of Lights

- *Point* (also called *Local*)
- *Directional* (also called *Infinite*)
- Light's type determined by its w value
  - $w = 0$ infinite light
  - $w = 1$ local light

Positioning Lights

- Light’s positions are modified by ModelView matrix
- Three variations
  - fixed in space
  - fixed in a scene
  - total freedom

Setting up a Fixed Light

- Light positioned in eye coordinates
  - identity matrix on ModelView stack
- Special case - creating a headlamp
  - imagine wearing a miner’s helmet with a light
  - pass (0 0 0 *w*) for light’s position

```
GLfloat  pos[] = { 0.0, 0.0, 0.0, 1.0 };
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glLightfv( GL_LIGHT0, GL_POSITION, pos );
```

Positioning a Light in a Scene

- Light positioned in world coordinates
  - viewing transform only on ModelView stack

```
GLfloat pos[] = { 1.0, 2.0, 3.0, 0.0 };

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
glLightfv( GL_LIGHT0, GL_POSITION, pos );
```

Arbitrary Light Positioning

- Any modeling and viewing transforms on ModelView stack
- Transform light separately by isolating with **`glPushMatrix()`** and **`glPopMatrix()`**
- Unique motion variable allows light to animate independently of other objects

Arbitrary Light Positioning (cont.)

```
GLfloat pos[] = { 0.0, 0.0, 0.0, 1.0 };
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
glPushMatrix();
glRotatef( angle, axis.x, axis.y, axis.z );
glTranslatef( x, y, z );
glLightfv( GL_LIGHT0, GL_POSITION, pos );
glPopMatrix();
```

Light Attenuation

- Physical light’s brightness diminishes as the square of the distance $d = \left\| \vec{p} - \vec{l} \right\|^2$
- Simulate this in OpenGL
  - `GL_CONSTANT_ATTENUATION`
  - `GL_LINEAR_ATTENUATION`
  - `GL_QUADRATIC_ATTENUATION`

$$f = \frac{1}{a_c + a_l \cdot d + a_q d^2}$$

Everything Else …

- “Global” lighting parameters are held in the *light model*

`glLightModel[fd]v(` *`property, param`* `);`

  - `GL_LIGHT_MODEL_AMBIENT`
  - `GL_LIGHT_MODEL_TWO_SIDE`
  - `GL_LIGHT_MODEL_LOCAL_VIEWER`

Turning on the Lights

- To turn on lighting

  `glEnable(` *`GL_LIGHTING`* `);`

  - turns on the “power”, but not any lights
- To turn on an individual light

  `glEnable(` *`GL_LIGHTn`* `);`

OpenGL Lighting Model

- At each vertex
  - For each color component

$$c = g_a + m_e + \sum_i f_i \left(l_{i_a} m_a + l_{i_d} \cdot m_d \left(\hat{n} \bullet \hat{l}\right) + l_{i_s} m_s \left(\hat{n} \bullet \hat{h}\right)^s \right)$$

$$\begin{cases} g & \texttt{glLightModel}() \\ l & \texttt{glLight}() \\ m & \texttt{glMaterial}() \end{cases}$$

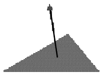
Computing Surface Normals

- Lighting needs to know how to reflect light off the surface
- Provide normals per
  - face - flat shading
  - vertex - Gouraud shading
  - pixel - Phong shading
    - OpenGL does not support Phong natively

Face Normals

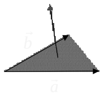
- Same normal for all vertices in a primitive
  - results in flat shading for primitive

```
glNormal3f( nx, ny, nz );
glBegin( GL_TRIANGLES );
  glVertex3fv( v1 );
  glVetrex3fv( v2 );
  glVertex3fv( v3 );
glEnd();
```



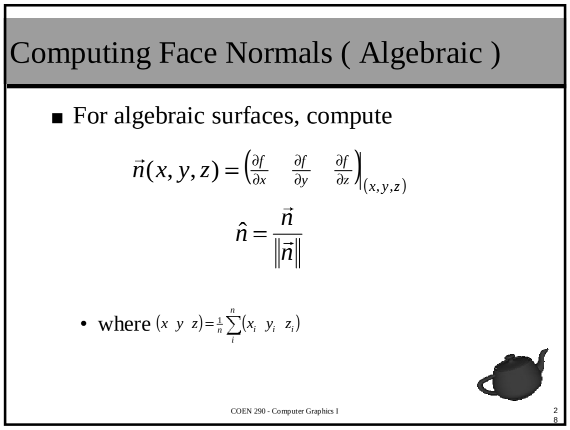
Computing Face Normals (Polygons)

- We’re using only planar polygons
- Can easily compute the normal to a plane
  - use a cross product



$$\vec{a} \times \vec{b} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

$$\hat{n} = \frac{\vec{a} \times \vec{b}}{\left\| \vec{a} \times \vec{b} \right\|}$$



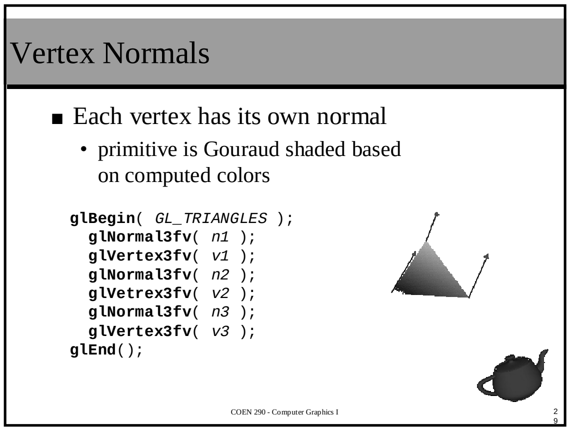
Computing Face Normals (Algebraic)

- For algebraic surfaces, compute

$$\vec{n}(x,y,z) = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{pmatrix}_{(x,y,z)}$$

$$\hat{n} = \frac{\vec{n}}{\|\vec{n}\|}$$

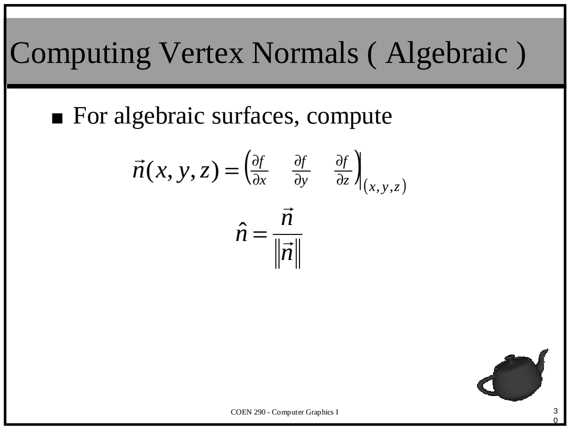
  - where $(x\ \ y\ \ z) = \frac{1}{n}\sum_{i}^{n}(x_i\ \ y_i\ \ z_i)$



Vertex Normals

- Each vertex has its own normal
  - primitive is Gouraud shaded based on computed colors

```
glBegin( GL_TRIANGLES );
  glNormal3fv( n1 );
  glVertex3fv( v1 );
  glNormal3fv( n2 );
  glVetrex3fv( v2 );
  glNormal3fv( n3 );
  glVertex3fv( v3 );
glEnd();
```



Computing Vertex Normals (Algebraic)

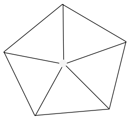
- For algebraic surfaces, compute

$$\vec{n}(x,y,z) = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{pmatrix}_{(x,y,z)}$$

$$\hat{n} = \frac{\vec{n}}{\|\vec{n}\|}$$

Computing Vertex Normals (Polygons)

- Need two things
  - face normals for all polygons
  - know which polygons share a vertex



$$\hat{n}_v = \left\| \sum_{i}^{m} \hat{n}_i \right\|$$

Sending Normals to OpenGL

`glNormal3f``( x, y, z );`

- Use between `glBegin()` / `glEnd()`
- Use similar to `glColor*()`

Normals and Scale Transforms

- Normals must be normalized
  - non-unit length skews colors
- Scales affect normal length
  - rotates and translates do not

`glEnable( GL_NORMALIZE );`

Why?

- Lighting computations are really done in eye coordinates
  - this is why there are the projection and modelview matrix stacks
- Lighting normals transformed by the inverse transpose of the ModelView matrix

Rasterizing Points

- Rendering a point should set one pixel
- Which pixel should we set?

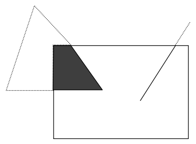
$$(x \quad y) \rightarrow \left(\left\lfloor x+\tfrac{1}{2} \right\rfloor \quad \left\lfloor y+\tfrac{1}{2} \right\rfloor \right)$$

- Use the following macro

```
#define ROUND(x) ((int)(x + 0.5))
```

Where should you draw

- *viewport* is the part of the window where you can render
- Need to *clip* objects to the viewport

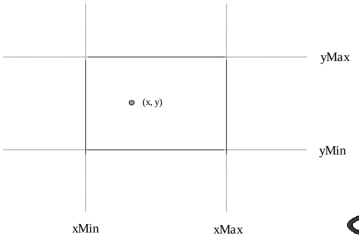


Clipping

- Determination of visible primitives
- Can clip to an arbitrary shape
  - we’ll only clip to rectangles
- Various clip boundaries
  - window
  - viewport
  - *scissor box*

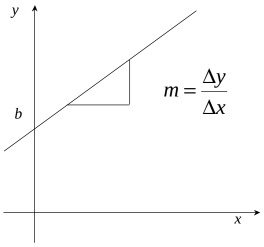
Point Clipping

- Simple point inside rectangle test



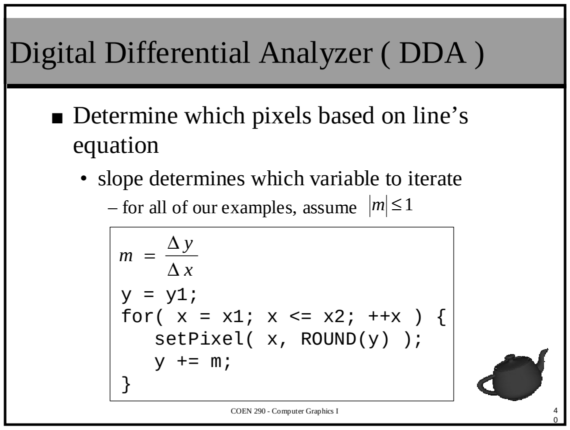
Mathematics of Lines

- Point-Intercept form of a line



$$m = \frac{\Delta y}{\Delta x}$$

$$y = mx + b$$

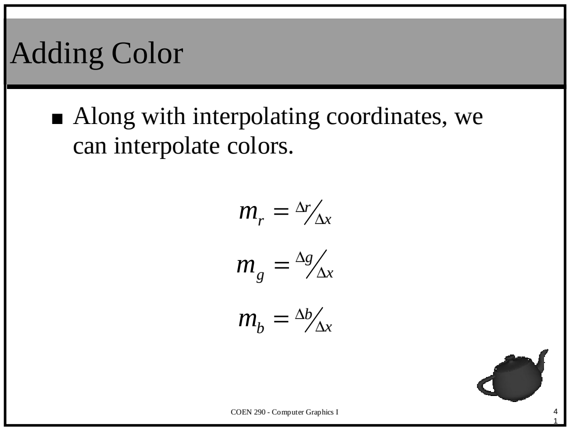


Digital Differential Analyzer (DDA)

- Determine which pixels based on line's equation
  - slope determines which variable to iterate
    - for all of our examples, assume $|m| \leq 1$

$$m = \frac{\Delta y}{\Delta x}$$

```
y = y1;
for( x = x1; x <= x2; ++x ) {
   setPixel( x, ROUND(y) );
   y += m;
}
```



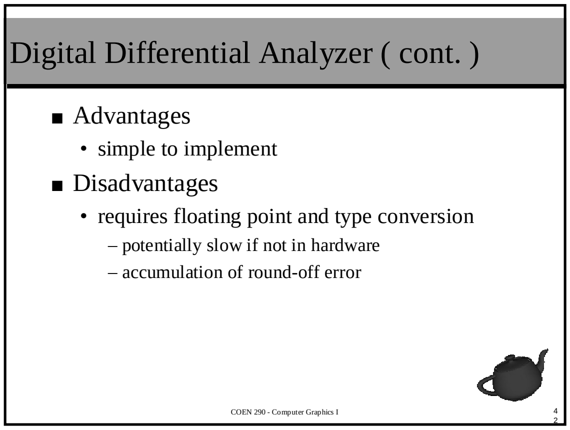
Adding Color

- Along with interpolating coordinates, we can interpolate colors.

$$m_r = {}^{\Delta r}\!/_{\Delta x}$$

$$m_g = {}^{\Delta g}\!/_{\Delta x}$$

$$m_b = {}^{\Delta b}\!/_{\Delta x}$$

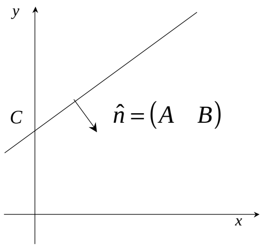


Digital Differential Analyzer (cont.)

- Advantages
  - simple to implement
- Disadvantages
  - requires floating point and type conversion
    - potentially slow if not in hardware
    - accumulation of round-off error

Explicit Form of a Line

$$f(x, y) = Ax + By + C$$

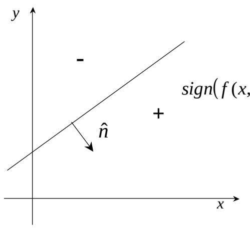


Another way of saying the same thing

$$f(\vec{p}) = \hat{n} \cdot \vec{p} + C$$

Why the Explicit Form is your Friend

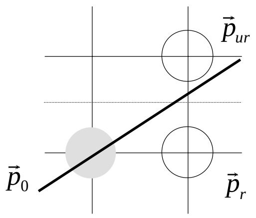
- Creates a *Binary Space Partition*
  - tells which side of the line your on

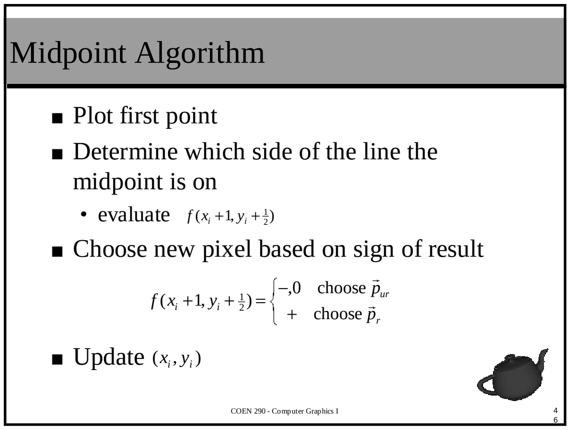


$$sign\left(f(x, y)\right) = \begin{cases} + & \text{same side as normal} \\ 0 & \text{point on line} \\ - & \text{opposite side of normal} \end{cases}$$

How does this help render lines?

- We can use the explicit form to determine which pixel is closest to the line



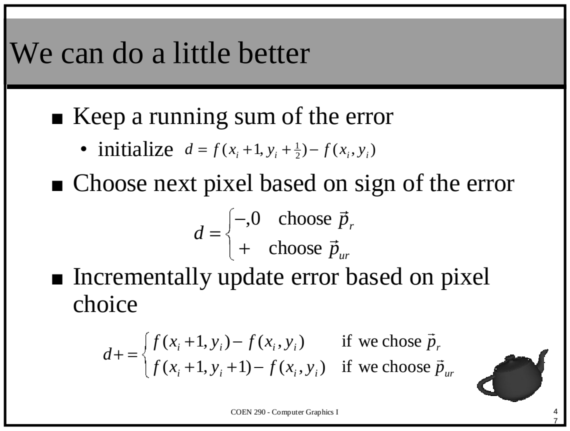


Midpoint Algorithm

- Plot first point
- Determine which side of the line the midpoint is on
  - evaluate $f(x_i+1, y_i+\frac{1}{2})$
- Choose new pixel based on sign of result

$$f(x_i+1, y_i+\tfrac{1}{2}) = \begin{cases} -,0 & \text{choose } \vec{p}_{ur} \\ + & \text{choose } \vec{p}_r \end{cases}$$

- Update (x_i, y_i)



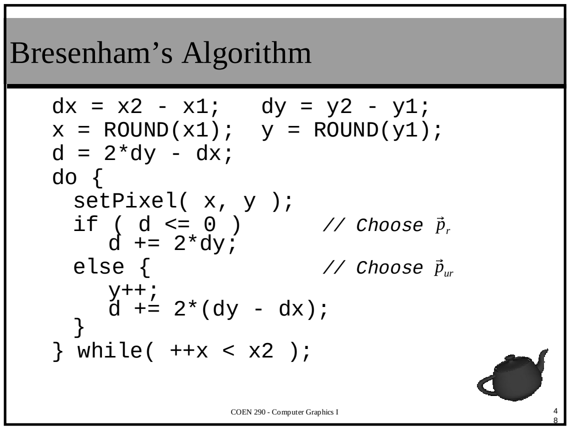
We can do a little better

- Keep a running sum of the error
  - initialize $d = f(x_i+1, y_i+\frac{1}{2}) - f(x_i, y_i)$
- Choose next pixel based on sign of the error

$$d = \begin{cases} -,0 & \text{choose } \vec{p}_r \\ + & \text{choose } \vec{p}_{ur} \end{cases}$$

- Incrementally update error based on pixel choice

$$d+= \begin{cases} f(x_i+1, y_i) - f(x_i, y_i) & \text{if we chose } \vec{p}_r \\ f(x_i+1, y_i+1) - f(x_i, y_i) & \text{if we choose } \vec{p}_{ur} \end{cases}$$

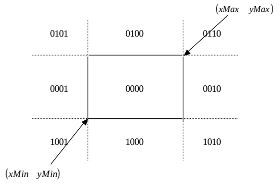


Bresenham's Algorithm

```
dx = x2 - x1;   dy = y2 - y1;
x = ROUND(x1);  y = ROUND(y1);
d = 2*dy - dx;
do {
  setPixel( x, y );
  if ( d <= 0 )       // Choose $\vec{p}_r$
     d += 2*dy;
  else {              // Choose $\vec{p}_{ur}$
     y++;
     d += 2*(dy - dx);
  }
} while( ++x < x2 );
```

Cohen-Sutherland Line Clipping

- Clip to rectangular region
- Partition space into regions
  - keep a bit-code to indicate which region a vertex is in



Cohen-Sutherland Line Clipping (cont.)

- Quickly determine if a line is outside the viewport

  if (mask$_{v1}$ & mask$_{v2}$)
    return False; ***// Don't render***

- Or inside

  if (!(mask$_{v1}$ | mask$_{v2}$))
    return True; ***// render! No clipping needed***

Cohen-Sutherland Line Clipping (cont.)

- If quick tests fail … need to clip vertices
- Render horizontal span between each edge's pixel