Lighting
Scan Conversion

Wolfgang Heidrich

Course News

Assignment 2
- Due Monday, Feb 28

Homework 3
- Discussed in labs next wee

Quiz 1
- Discussed in labs this week

Reading
- Chapter 9, 3

Out of Town Friday
- Anika will fill in for me
The Rendering Pipeline

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

Rasterization → Fragment Processing

Lighting

**Goal**
- Model the interaction of light with surfaces to render realistic images

**Contributing Factors**
- Light sources
  - Shape and color
- Surface materials
  - How surfaces reflect light
- Transport of light
  - How light moves in a scene (global illumination, later in the course)
Types of Reflection

- Specular (a.k.a. mirror or regular) reflection causes light to propagate without scattering.

- Diffuse reflection sends light in all directions with equal energy.

- Mixed reflection is a weighted combination of specular and diffuse.

Types of Reflection

- retro-reflection occurs when incident energy reflects in directions close to the incident direction, for a wide range of incident directions.

- gloss is the property of a material surface that involves mixed reflection and is responsible for the mirror-like appearance of rough surfaces.
Specular Reflection

Geometry of specular (mirror) reflection

\[ r = -l + 2(n \cdot l)n \]

Intuitively: cross-sectional area of the “beam” intersecting an element of surface area is smaller for greater angles with the normal.

Lambert’s “Law”

Lambert's Cosine Law

Intuitively: cross-sectional area of the “beam” intersecting an element of surface area is smaller for greater angles with the normal.
Computing Diffuse Reflection

- Depends on angle of incidence: angle between surface normal and incoming light
  \[ I_{\text{diffuse}} = k_d I_{\text{light}} \cos \theta \]
- In practice use vector arithmetic
  \[ I_{\text{diffuse}} = k_d I_{\text{light}} (n \cdot l) \]
- Always normalize vectors used in lighting
  - \( n \), \( l \) should be unit vectors
- Scalar (B/W intensity) or 3-tuple or 4-tuple (color)
  - \( k_d \): diffuse coefficient, surface color
  - \( I_{\text{light}} \): incoming light intensity
  - \( I_{\text{diffuse}} \): outgoing light intensity (for diffuse reflection)

Glossy Materials – Empirical Approximation

Angular falloff

\[ \vec{n} \]

how might we model this falloff?
**Phong Lighting**

*Most common lighting model in computer graphics*

(Phong Bui-Tuong, 1975)

\[ I_{\text{specular}} = k_s I_{\text{light}} (\cos \phi)^{n_s} \]

- \( n_s \): purely empirical constant, varies rate of falloff
- \( k_s \): specular coefficient, highlight color
- no physical basis, works ok in practice

**Alternative Model**

*Blinn-Phong model (Jim Blinn, 1977)*

- Variation with better physical interpretation
  - \( h \): halfway vector; \( r \): roughness

\[ I_{\text{out}}(x) = k_s \cdot (h \cdot n)^{1/r} \cdot I_{\text{in}}(x); \text{ with } h = (l + v) / 2 \]
**Simple Light Sources**

**Types of light sources**
- Directional/parallel lights
  - *E.g. sun*
  - *Homogeneous vector*
- (Homogeneous) point lights
  - *Same intensity in all directions*
  - *Homogeneous point*
- Spot lights
  - *Limited set of directions*
  - *Point+direction+cutoff angle*

**Light Sources**

**Area lights:**
- Light sources with a finite area
- Can be considered a continuum of point lights
- Not available in many rendering systems
Light Source Falloff

**Quadratic falloff (point- and spot lights)**
- Brightness of objects depends on power per unit area that hits the object
- The power per unit area for a point or spot light decreases quadratically with distance

\[
\text{Area } 4\pi r^2
\]

\[
\text{Area } 4\pi (2r)^2
\]

**Non-quadratic falloff**
- Many systems allow for other falloffs
- Allows for faking effect of area light sources
- OpenGL / graphics hardware
  - \( I_o \): intensity of light source
  - \( x \): object point
  - \( r \): distance of light from \( x \)

\[
I_{in}(x) = \frac{1}{ar^2 + br + c} \cdot I_0
\]
Light Sources

Ambient lights
- No identifiable source or direction
- Hack for replacing true global illumination
  - *light bouncing off from other objects*

Ambient Light Sources
- Scene lit only with an ambient light source

- Light Position Not Important
- Viewer Position Not Important
- Surface Angle Not Important

Wolfgang Heidrich
### Directional Light Sources

- Scene lit with directional and ambient light

### Point Light Sources

- Scene lit with ambient and point light source
Light Sources & Transformations

Geometry: positions and directions

- Standard: world coordinate system
  - Effect: lights fixed wrt world geometry
  - Demo: http://www.xmission.com/~nate/tutors.html
- Alternative: camera coordinate system
  - Effect: lights attached to camera (car headlights)
- Points and directions undergo normal model/view transformation

Illumination calculations: camera coords

Lighting Review

Lighting models

- Ambient
  - Normals don’t matter
- Lambert/diffuse
  - Angle between surface normal and light
- Phong/specular
  - Surface normal, light, and viewpoint
Lighting in OpenGL

**Light source: amount of RGB light emitted**
- Value represents percentage of full intensity
  - E.g., (1.0,0.5,0.5)
- Every light source emits ambient, diffuse, and specular light

**Materials: amount of RGB light reflected**
- Value represents percentage reflected
  - E.g., (0.0,1.0,0.5)

**Interaction: multiply components**
- Red light (1,0,0) x green surface (0,1,0) = black (0,0,0)

```c
glLightfv(GL_LIGHT0, GL_AMBIENT, amb_light_rgba);
glLightfv(GL_LIGHT0, GL_DIFFUSE, dif_light_rgba);
glLightfv(GL_LIGHT0, GL_SPECULAR, spec_light_rgba);
glLightfv(GL_LIGHT0, GL_POSITION, position);
glEnable(GL_LIGHT0);

glMaterialfv( GL_FRONT, GL_AMBIENT, ambient_rgba );
glMaterialfv( GL_FRONT, GL_DIFFUSE, diffuse_rgba );
glMaterialfv( GL_FRONT, GL_SPECULAR, specular_rgba );
glMaterialfv( GL_FRONT, GL_SHININESS, n );
```
Lighting in Rendering Pipeline

**Notes:**
- Lighting is applied to every **vertex**
  - *i.e. the three vertices in a triangle*
  - **Per-vertex lighting**
- Will later see how the interior points of the triangle obtain their color
  - *This process is called shading*
  - *Will discuss in the context of scan conversion*

Scan Conversion

Wolfgang Heidrich
The Rendering Pipeline

Scan Conversion - Rasterization

Convert continuous rendering primitives into discrete fragments/pixels

- Lines
  - Midpoint/Bresenham
- Triangles
  - Flood fill
  - Scanline
  - Implicit formulation
- Interpolation

Wolfgang Heinrich
Scan Conversion - Lines

Wolfgang Heidrich
Scan Conversion - Lines

First Attempt:
- Line (s,e) given in device coordinates
- Create the thinnest line that connects start point and end point without gap

Assumptions for now:
- Start point to the left of end point: \( xs < xe \)
- Slope of the line between 0 and 1 (i.e. elevation between 0 and 45 degrees):

\[
0 \leq \frac{ye - ys}{xe - xs} \leq 1
\]

---

Scan Conversion of Lines - Digital Differential Analyzer

First Attempt:

```c
dda( float xs, ys, xe, ye ) {
    // assume xs < xe, and slope m between 0 and 1
    float m = (ye-ys)/(xe-xs);
    float y = round( ys );
    for( int x = round( xs ) ; x<= xe ; x++ ) {
        drawPixel( x, round( y ) );
        y = y+m;
    }
}
```

Wolfgang Heidrich
Scan Conversion of Lines

**DDA:**

---

Scan Conversion of Lines

**Midpoint Algorithm**

*Moving horizontally along x direction*
- Draw at current y value, or move up vertically to y+1?
  - Check if midpoint between two possible pixel centers above or below line

**Candidates**
- Top pixel: (x+1, y+1)
- Bottom pixel: (x+1, y)

**Midpoint: (x+1, y+.5)**

**Check if midpoint above or below line**
- Below: top pixel
- Above: bottom pixel

*Key idea behind Bresenham Alg.*
Scan Conversion of Lines

**Idea: decision variable**

```c
dda( float xs, ys, xe, ye ) {
    float d= 0.0;
    float m= (ye-ys)/(xe-xs);
    int y= round( ys );
    for( int x= round( xs ) ; x<= xe ; x++ ) {
        drawPixel( x, y );
        d= d+m;
        if( d>= 0.5 ) { d= d-1.0; y++; }
    }
}
```

Scan Conversion of Lines
Bresenham Algorithm (’63)

- Use decision variable to generate purely integer algorithm
- Explicit line equation:
  
  \[ y = \frac{(y_e - y_s)}{(x_e - x_s)}(x - x_s) + y_s \]

- Implicit version:
  
  \[ L(x, y) = \frac{(y_e - y_s)}{(x_e - x_s)}(x - x_s) - (y - y_s) = 0 \]

- In particular for specific x, y, we have
  - \( L(x, y) > 0 \) if \((x, y)\) below the line, and
  - \( L(x, y) < 0 \) if \((x, y)\) above the line
Scan Conversion of Lines

Bresenham Algorithm

- Decision variable: after drawing point \((x,y)\) decide whether to draw
  - \((x+1,y)\): **case E** (for “east”)
  - \((x+1,y+1)\): **case NE** (for “north-east”)

Check whether \((x+1,y+1/2)\) is above or below line

\[
d = L(x+1, y + \frac{1}{2})
\]

Point above line if and only if \(d<0\)

---

Scan Conversion of Lines

**Bresenham Algorithm**

- Problem: how to update \(d\)?

- Case E (point above line, \(d \leq 0\))
  - \(x= x+1;\)
  - \(d= L(x+2, y+1/2) = d + (y_e-y_s)/(x_e-x_s)\)

- Case NE (point below line, \(d > 0\))
  - \(x= x+1; y= y+1;\)
  - \(d= L(x+2, y+3/2) = d + (y_e-y_s)/(x_e-x_s) - 1\)

- Initialization:
  - \(d= L(x_s, y_s+1/2) = (y_e-y_s)/(x_e-x_s) - 1/2\)
Scan Conversion of Lines

Bresenham Algorithm

- This is still floating point
- But: only sign of \( d \) matters
- Thus: can multiply everything by \( 2(x_e-x_s) \)

```c
Bresenham( int xs, ys, xe, ye ) {
    int y = ys;
    incrE = 2(ys - ye);
    incrNE = 2((ye - ys) - (xe-xs));
    for( int x = xs ; x<= xe ; x++ ) {
        drawPixel( x, y );
        if( d<= 0 ) d += incrE;
        else { d += incrNE; y++; }
    }
}
```
Scan Conversion of Lines

**Discussion**
- Bresenham sets same pixels as DDA
- Intensity of line varies with its angle!

![Diagram of Scan Conversion of Lines](image)

---

Scan Conversion of Lines

**Discussion**
- Bresenham
  - *Good for hardware implementations (integer!)*
- DDA
  - *May be faster for software (depends on system)!*
  - *Floating point ops higher parallelized (pipelined)*
    - E.g. RISC CPUs from MIPS, SUN
  - *No if statements in inner loop*
    - More efficient use of processor pipelining
Scan Conversion of Polygons

One possible scan conversion
Scan Conversion of Polygons

A General Algorithm
- Intersect each scanline with all edges
- Sort intersections in x
- Calculate parity to determine in/out
- Fill the ‘in’ pixels

Scan Conversion of Polygons
- Works for arbitrary polygons
- Efficiency improvement:
  - Exploit row-to-row coherence using “edge table”
**Edge Walking**

*Past graphics hardware*
- Exploit continuous L and R edges on trapezoid

\[
\text{scanTrapezoid}(x_L, x_R, y_B, y_T, \Delta x_L, \Delta x_R)
\]

```c
for (y=yB; y<=yT; y++) {
  for (x=xL; x<=xR; x++)
    setPixel(x, y);
  xL += DxL;
  xR += DxR;
}
```

Wolfgang Heinrich
Edge Walking Triangles

Split triangles into two regions with continuous left and right edges

\[
\text{scanTrapezoid}(x_3, x_m, y_3, y_m, \frac{1}{m_{13}}, \frac{1}{m_{12}})
\]

\[
\text{scanTrapezoid}(x_2, x_2', y_2, y_2', \frac{1}{m_{23}}, \frac{1}{m_{12}})
\]

Issues

- Many applications have small triangles
  - Setup cost is non-trivial
- Clipping triangles produces non-triangles
  - This can be avoided through re-triangulation, as discussed
Coming Up:

**Friday**

- More scan conversion
- Lecture by Anika