News
- midterm review Wednesday
  - plus Kangaroo Hall of Fame
- midterm Friday (be on time!)
  - covering through lighting/shading
  - not color or rasterization
- homework 1 solutions out
  - no more late homework accepted
- program 2 writeup out
  - due Thu Feb 24

Program 2: Terrain Navigation
- make bumpy terrain
  - 100x100 rectangular grid
  - vertex height varies randomly by 20%
  - vertex color varies randomly
  - switch between per-face, per-vertex normals
  - explicitly draw normals (hedgehog mode)
- lighting and shading
  - headlamp, plus at least one fixed light
  - switch between smooth and flat shading

Navigating
- two flying modes: absolute and relative
- absolute
  - keyboard keys to increment/decrement
    - x/y/z position of eye, lookat, up vectors
- relative
  - mouse drags
  - incremental wrt current camera position
  - forward/backward motion
  - roll, pitch, and yaw angles

Hint: Incremental Motion
- motion is wrt current camera coords
  - maintaining cumulative angles wrt world coords would be difficult
  - computation in coord system used to draw previous frame is simple
- OpenGL modelview matrix has the info!
  - but multiplying by new matrix gives $p' = Cp$
  - you want to do $p' = ICp$
  - trick:
    - dump out modelview matrix
    - wipe the stack with glIdentity
    - apply incremental update matrix
    - apply current camera coord matrix

Reading
- Color (reading from Friday)
  - FCG Chap 17 Human Vision (pp 293-298)
  - FCG Chap 18 Color (pp 301-311)
    - until Section 18.9 Tone Mapping
  - FCG Sec 3.2 Gamma Correction
  - FCG Sec 3.3 RGB Color
- Rasterization
  - FCG Chap 3 Raster Algorithms (pp 49-67)
  - FCG Section 2.11 Barycentric Coordinates
FCG Errata

- p 54
  - triangle at bottom of figure shouldn’t have black outline
- p 63
  - The test if numbers \( a[x] \) and \( b[y] \) have the same sign can be implemented as the test \( ab[xy] > 0 \).

Font Correction: Lighting in OpenGL

```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);
```

- warning: `glMaterial` is expensive and tricky
- use cheap and simple `glColor` when possible
- see OpenGL Pitfall #14 from Kilgard’s list
- http://www.opengl.org/resources/features/KilgardTechniques/oglpitfall/

Correction/Review: Computing Normals

- per-vertex normals by interpolating per-facet normals
  - OpenGL supports both
- computing normal for a polygon
  - three points form two vectors
  - cross: normal of plane direction
  - normalize: make unit length
- which side of plane is up?
  - counterclockwise point order convention

Review: Trichromacy and Metamers

- three types of cones
- color is combination of cone stimuli
- metamer: identically perceived color caused by very different spectra

Review: Color Constancy

![Image of color constancy](image-courtesy-of-John-McCann.png)

Review: Measured vs. CIE Color Spaces

- measured basis
  - monochromatic lights
  - physical observations
  - negative lobes
- transformed basis
  - “imaginary” lights
  - all positive, unit area
  - \( Y \) is luminance
Review: Device Color Gamuts

- compare gamuts on CIE chromaticity diagram
- gamut mapping

Review: RGB Color Space

- define colors with \((r, g, b)\) amounts of red, green, and blue
- used by OpenGL
- RGB color cube sits within CIE color space
- subset of perceivable colors

Review: HSV Color Space

- hue: dominant wavelength, "color"
- saturation: how far from grey
- value/brightness: how far from black/white

Review: YIQ Color Space

- \(YIQ\) is the color model used for color TV in America. \(Y\) is brightness, \(I\) & \(Q\) are color
- same \(Y\) as CIE, backwards compatibility with black and white TV
- blue is more compressed

\[
\begin{bmatrix}
Y \\
I \\
Q
\end{bmatrix} = \begin{bmatrix}
0.30 & 0.59 & 0.11 \\
0.60 & -0.28 & -0.32 \\
0.21 & -0.52 & 0.31
\end{bmatrix} \begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

Review: Gamma Correction

\[
\gamma_{DS} = \gamma_D \left(1/\gamma_{OS}\right)
\]

Rasterization
Scan Conversion - Rasterization
- convert continuous rendering primitives into discrete fragments/pixels
  - lines
  - Bresenham
  - triangles
  - flood fill
  - scanline
  - implicit formulation
  - interpolation

Scan Conversion
- given vertices in DCS, fill in the pixels
  - start with lines

Basic Line Drawing
\[ y = mx + b \]
\[ y' = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) + y_1 \]
- goals
  - integer coordinates
  - thinnest line with no gaps
- assume
  - \( x_1 < x_2 \), slope \( 0 < \frac{dy}{dx} < 1 \)
- how can we do this quickly?

Midpoint Algorithm
- moving incrementally along \( x \) direction
  - draw at current \( y \) value, or move up 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

Making It Fast
- maintain error value
  - test
    - if \((y+e+m) < y+.5\)
    - \( e+m < .5 \)
  - if top pixel picked
    - \( e = y+e+m-y = e+m \)
  - if bottom pixel picked
    - \( e = y+e+m-(y+1) = e+m-1 \)
- convert to use only integer arithmetic (remember \( m=dy/dx \))
  - test: multiply by \( 2^*dx \), then check if \((2*e*dx+dy) < dx \)
  - top: multiply by \( dx \), then \( e*dx = e*dx+dy \)
  - bottom: multiply by \( dx \), then \( e*dx = e*dx+dy-1 \)
- \( E= e*dx \)
Bresenham Line Drawing Algorithm

```c
y=y0; e=0;
for (x=x0; x <= x1; x++) {
    draw(x,y);
    if (2(e+dy) < dx) {
        e = e+dy;
    } else {
        y=y+1;
        e=e+dy-dx;
    }
}
```

Bresenham Line Drawing Algorithm

```c
y=y0; e=0;
for (x=x0; x <= x1; x++) {
    draw(x,y);
    if (2(e+dy) < dx) {
        e = e+dy;
    } else {
        y=y+1;
        e=e+dy-dx;
    }
}
```

**Polygons**

- basic surface representation in rendering
- why?
  - lowest common denominator
  - can approximate any surface with arbitrary accuracy
    - all polygons can be broken up into triangles
- guaranteed to be:
  - planar
  - triangles - convex
- simple to render
- can implement in hardware

**Triangulation**

- convex polygons easily triangulated
- concave polygons present a challenge

**OpenGL Triangulation**

- simple convex polygons
  - break into triangles, trivial
    - glBegin(GL_POLYGON) ... glEnd()
- concave or non-simple polygons
  - break into triangles, more effort
    - gluNewTess(), gluTessCallback(), ...
Problem

- input: closed 2D polygon
- problem: fill its interior with specified color on graphics display
- assumptions
  - simple - no self intersections
  - simply connected
- solutions
  - flood fill
  - scan conversion
  - implicit test

Flood Fill

- simple algorithm
  - draw edges of polygon
  - use flood-fill to draw interior

Flood Fill

- start with seed point
- recursively set all neighbors until boundary is hit

Flood Fill Drawbacks

- pixels visited up to 4 times to check if already set
- need per-pixel flag indicating if set already
  - must clear for every polygon!

Scanline Algorithms

- scanline: a line of pixels in an image
**Scanline Algorithms**
- set pixels inside polygon boundary along horizontal lines one pixel apart
- use bounding box to speed up

**Edge Walking**
- basic idea:
  - draw edges vertically
  - interpolate colors down edges
  - fill in horizontal spans for each scanline
  - at each scanline, interpolate edge colors across span

**Triangle Rasterization Issues**
- moving slivers
- shared edge ordering

**General Polygon Rasterization**
- consider the following polygon:
- how do we know whether a given pixel on the scanline is inside or outside the polygon?

**Triangle Rasterization Issues**
- exactly which pixels should be lit?
  - pixels with centers inside triangle edges
- what about pixels exactly on edge?
  - draw them: order of triangles matters (it shouldn’t)
  - don’t draw them: gaps possible between triangles
  - need a consistent (if arbitrary) rule
  - example: draw pixels on left or top edge, but not on right or bottom edge
  - example: check if triangle on same side of edge as offscreen point

**General Polygon Rasterization**
- idea: use a parity test

```c
for each scanline
    edgeCnt = 0;
    for each pixel on scanline (l to r)
        if (oldpixel->newpixel crosses edge)
            edgeCnt ++;
        // draw the pixel if edgeCnt odd
        if (edgeCnt % 2)
            setPixel(pixel);
```
Scan Conversion

- done:
  - how to determine pixels covered by a primitive
- next:
  - how to assign pixel colors
    - interpolation of colors across triangles
    - interpolation of other properties

Interpolation

Interpolation During Scan Conversion

- interpolate values between vertices
  - $z$ values
  - $r,g,b$ colour components
    - use for Gouraud shading
  - $u,v$ texture coordinates
  - $N_x,N_y,N_z$ surface normals
- equivalent methods (for triangles)
  - bilinear interpolation
  - barycentric coordinates

Bilinear Interpolation

- interpolate quantity along $L$ and $R$ edges, as a function of $y$
- then interpolate quantity as a function of $x$

3. Barycentric Coordinates

- weighted combination of vertices

\[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]
\[ \alpha + \beta + \gamma = 1 \]
\[ 0 \leq \alpha, \beta, \gamma \leq 1 \]

“convex combination of points”

Computing Barycentric Coordinates

- for point $P$ on scanline

\[ P_x = \frac{d_1}{d_1 + d_2} (P_3 - P_2) + \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_3 = \]
\[ (1 - \frac{d_1}{d_1 + d_2}) P_2 + \frac{d_1}{d_1 + d_2} P_3 = \]
\[ \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_3 \]
Computing Barycentric Coords

- similarly

\[ P_R = P + \frac{b_1}{b_1 + b_2}(P_1 - P_2) \]

\[ = \left(1 - \frac{b_1}{b_1 + b_2}\right)P_2 + \frac{b_1}{b_1 + b_2}P_1 \]

\[ = \frac{b_1}{b_1 + b_2}P_2 + \frac{b_1}{b_1 + b_2}P_1 \]

- combining

\[ P = \frac{c_2}{c_1 + c_2} \left( \frac{d_2}{d_1 + d_2}P_2 + \frac{d_1}{d_1 + d_2}P_1 \right) + \frac{c_1}{c_1 + c_2} \left( \frac{b_2}{b_1 + b_2}P_2 + \frac{b_1}{b_1 + b_2}P_1 \right) \]

- gives

\[ P = \frac{c_2}{c_1 + c_2} \left( \frac{d_2}{d_1 + d_2}P_2 + \frac{d_1}{d_1 + d_2}P_1 \right) + \frac{c_1}{c_1 + c_2} \left( \frac{b_2}{b_1 + b_2}P_2 + \frac{b_1}{b_1 + b_2}P_1 \right) \]

- can verify barycentric properties

\[ a_1 + a_2 + a_3 = 1 \]

\[ 0 \leq a_1, a_2, a_3 \leq 1 \]