Where are we?

- Last lecture, we started the OpenGL pipeline with our example code
- This lecture we’ll continue that
OpenGL API Example

```c
// smooth color interpolation
glShadeModel(GL_SMOOTH);

// enable hidden surface removal
glEnable(GL_DEPTH_TEST);

// clear color and depth buffers
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

// begin drawing triangles
glBegin(GL_TRIANGLES);

// vertexes and colors
glColor4ub(255, 0, 0, 255); // RGBA= (1,0,0,100%)
glVertex3f(-0.8, 0.8, 0.3); // XYZ= (-8/10,8/10,3/10)

// vertexes and colors
glColor4ub(0, 255, 0, 255); // RGBA= (0,1,0,100%)
glVertex3f(0.8, 0.8, -0.2); // XYZ= (8/10,8/10,-2/10)

// vertexes and colors
glColor4ub(0, 0, 255, 255); // RGBA= (0,0,1,100%)
glVertex3f(0.0, -0.8, -0.2); // XYZ= (0,-8/10,-2/10)

// end drawing
glEnd();
```
#include <GL/glut.h>  // includes necessary OpenGL headers

void display() {
    // << insert code on prior slide here >>
    glutSwapBuffers();
}

void main(int argc, char **argv) {
    // request double-buffered color window with depth buffer
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInit(&argc, argv);
    glutCreateWindow("simple triangle");
    glutDisplayFunc(display);  // function to render window
    glutMainLoop();
}
NDC to Window Space

- Done transforming from NDC space to window space
- Next: Rasterize, then shade pixels (fragments)
Screen Space Coordinates of Triangle

- Assume the window is 500x500 pixels
- So `glViewport(0,0,500,500)` has been called

L=(50, 450, 0.65)
M=(250,50,0.4)
N=(450,450,0.4)

center at (250,250)
origin at (0,0)
Rasterization

- Process of converting a clipped triangle into a set of sample locations covered by the triangle
- Also can rasterize points and lines
Determining a Triangle

- **Classic view:** 3 points determine a triangle
  - Given 3 vertex positions, we determine a triangle
  - Hence glVertex3f/glVertex3f/glVertex3f

- **Rasterization view:** 3 oriented edge equations determine a triangle

Each oriented edge equation in form: 
\[ A*x + B*y + C \geq 0 \]
Oriented Edge Equations

\[ Ax + By + C > 0 \]

\[ Ax + By + C = 0 \]

\[ Ax + By + C < 0 \]
Step back: Why Triangles?

- Simplest linear primitive with area
  - If it got any simpler, the primitive would be a line (just 2 vertexes)
  - Guaranteed to be planar (flat) and convex (not concave)
- Triangles are compact
  - 3 vertexes, 9 scalar values in affine 3D, determine a triangle
  - When in a mesh, vertex positions can be “shared” among adjacent triangles
- Triangles are simple
  - Simplicity and generality of triangles facilitates elegant, hardware-amenable algorithms
- Triangles lacks curvature
  - BUT with enough triangles, we can piecewise approximate just about any manifold
- We can subdivide regions of high curvature until we reach flat regions to represent as a triangle
Concave vs. Convex

- Region is convex if any two points can be connected by a line segment where all points on this segment are also in the region.
  - Opposite is non-convex.
- Concave means the region is connected but NOT convex.
  - Connected means there’s some path (not necessarily a line) from every two points in the region that is entirely in the region.
7 Cases

\[ E_i(x,y) = A_i x + B_i y + C_i \]
Inside Triangle Test

- Evaluate edge equations at grid of sample points
  - If sample position is “inside” all 3 edge equations, the position is “within” the triangle
  - Implicitly parallel—all samples can be tested at once

- Good for hardware implementation
  - Pixel-planes
  - Pineda tiled extension
Other Rasterization Approaches

- **Subdivision approaches**
  - Easy to split a triangle into 4 triangles
  - Keep splitting triangles until they are slightly smaller than your samples
    - Often called micro-polygon rendering
    - Chief advantage is being able to apply displacements during the subdivision

- **Edge walking approaches**
  - Often used by CPU-based rasterizers
  - Much more sequential than Pineda approach
  - Work efficient and amendable to fixed-point implementation
Micropolygons

- Rasterization becomes a geometry dicing process
  - Approach taken by Pixar
    - For production rendering when scene detail and quality is at a premium; interactivity, not so much
  - High-level representation is generally patches rather than mere triangles

Displacement mapping of a meshed sphere [Pixar, RenderMan]
Find a “top” to the triangle
Now walk down edges
Scanline Rasterization

- Move down a scan-line, keeping track of the left and right ends of the triangle
Scanline Rasterization

- Repeat, moving down a scanline
- Cover the samples between the left and right ends of the triangle in the scan-line
Scanline Rasterization

- Process repeats for each scanline
- Easy to “step” down to the next scanline based on the slopes of two edges
Scanline Rasterization

- Eventually reach a vertex
- Transition to a different edge and continue filling the span within the triangle
Scanline Rasterization

- Until you finish the triangle
- Friendly for how CPU memory arranges an image as a 2D array with horizontal locality
- Layout is good for raster scan-out too
Creating Edge Equations

- Triangle rasterization need edge equations
  - How do we make edge equations?
- An edge is a line so determined by two points
  - Each of the 3 triangle edges is determined by two of the 3 triangle vertexes (L, M, N)

\[ N = (N_x, N_y) \]
\[ M = (M_x, M_y) \]
\[ L = (L_x, L_y) \]

How do we get

\[ A \times x + B \times y + C \geq 0 \]

for each edge from L, M, and N?
Edge Equation Setup

- How do you get the coefficients A, B, and C?
- Determinants help—consider the LN edge:

\[
\begin{vmatrix}
N_x - L_x & N_y - L_y \\
N_x - L_x & N_y - L_y \\
\end{vmatrix} > 0 \quad \text{or more succinctly} \quad \begin{vmatrix}
N - L \\
P - L \\
\end{vmatrix} > 0
\]

- **Expansion:** \((L_y-N_y)\times P_x + (N_x-L_x)\times P_y + N_y\times L_x - N_x\times L_y > 0\)
  - \(A_{LN} = L_y - N_y\)
  - \(B_{LN} = N_x - L_x\)
  - \(C_{LN} = N_y\times L_x - N_x\times L_y\)

- **Geometric interpretation:** twice the signed area of the triangle LPN
Assume the window is 500x500 pixels

So `glViewport(0, 0, 500, 500)` has been called

L = (50, 450, 0.65)

N = (450, 450, 0.4)

M = (250, 50, 0.4)

Center at (250, 250)

Origin at (0, 0)
Look at the LN edge

**Expansion:**

\[(Ly-Ny)\times Px + (Nx-Lx)\times Py + Ny\times Lx-Nx\times Ly > 0\]

- \(A_{LN} = Ly-Ny = 450-450 = 0\)
- \(B_{LN} = Nx-Lx = 50-450 = -400\)
- \(C_{LN} = Ny\times Lx-Nx\times Ly = 180,000\)

**Is center at (250,250) in the triangle?**

- \(A_{LN} \times 250 + B_{LN} \times 250 + C_{LN} = ???\)
- \(0 \times 250 - 400 \times 250 + 180,000 = 80,000\)
  - \(80,000 > 0\) so (250,250) is in the triangle
All Three Edge Equations

- All three triangle edge equations:

\[
\begin{vmatrix}
M - N \\
P - N
\end{vmatrix} > 0 
\quad \begin{vmatrix}
N - L \\
P - L
\end{vmatrix} > 0 \quad \begin{vmatrix}
L - M \\
P - M
\end{vmatrix} > 0
\]

- Satisfy all 3 and P is in the triangle
- And then rasterize at sample location P
- **Caveat:** if \( \begin{vmatrix}
N - L \\
M - L
\end{vmatrix} < 0 \) reverse the comparison sense
Water Tight Rasterization

- Two triangles often share a common edge
  - Indeed in closed polygonal meshes, every triangle shares its edges with as many as three other triangles
    - Called adjacent or “shared edge” triangles
- Crucial rasterization property
  - No double sampling (hitting) along the shared edge
  - No sample gaps (pixel fall-out) along the shared edge
  - Samples along the shared edge must be belong to exactly one of the two triangles
    - Not both, not neither
- Water tight rasterization is crucial to many higher-level algorithms; otherwise, rendering artifacts
  - Possible artifact: if pixels hit twice on an edge, the pixel could be double blended
  - Example application: Stenciled Shadow Volumes (SSV)
Water Tight Rasterization Solution

- First “snap” vertex positions to a grid
  - Grid can (and should) be sub-pixel samples
  - Results in fixed-point vertex positions
- Fixed-point math allows exact edge computations
  - **Surprising?** Ensuring robustness requires discarding excess precision
- Problem
  - What happens when edge equation evaluates to exactly zero at a sample position?
  - Need a consistent tie breaker
Tie Breaker Rule

- Look at edge equation coefficients
- Tie-breaker rule when edge equation evaluates to zero
  - “Inside” edge when edge equation is zero and
    A > 0 when A ≠ 0, or B > 0 when A = 0
- Complete coverage determination rule
  - if (E(x,y) > 0 || (E(x,y)==0 && (A != 0 ? A > 0 : B > 0)))
    sample at (x,y) is inside edge
Zero Area Triangles

- We reverse the edge equation comparison sense if the (signed) area of the triangle is negative.
- What if the area is zero?
  - Linear algebra indicates a singular matrix.
  - Need to cull the primitive.
- Also useful to cull primitives when area is negative.
  - OpenGL calls this face culling.
    - Enabled with `glEnable(GL_CULL_FACE)`.
  - When drawing closed meshes, back face culling can avoid drawing primitives assured to be occluded by front faces.
Back Face Culling Example

Torus drawn in wire-frame without back face culling

Notice considerable extraneous triangles that would normally be occluded

Torus drawn in wire-frame with back face culling

By culling back-facing (negative signed area) triangles, fewer triangles are rasterized
Simple Fragment Shading

- For all samples (pixels) within the triangle, evaluate the interpolated color
  - Requires having math to determine color at the sample (x,y) location

Application

Vertex batching & assembly

Clipping

NDC to window space

Rasterization

Fragment shading

Depth testing

Color update

Framebuffer

Depth buffer
Our simple triangle is drawn with smooth color interpolation

- Recall: `glShadeModel(GL_SMOOTH)`

How is color interpolated?

- Think of a plane equation to compute each color component (say red) as a function of \((x,y)\)
  - Just done for samples positions within the triangle

\[
"\text{redness}" = A_{\text{red}} x + B_{\text{red}} y + C_{\text{red}}
\]
Setup Plane Equation

Setup plane equation to solve for “red” as a function of \((x,y)\)

\[
\begin{bmatrix}
L_{\text{red}} \\
M_{\text{red}} \\
N_{\text{red}}
\end{bmatrix} =
\begin{bmatrix}
L_x & L_y & 1 \\
M_x & M_y & 1 \\
N_x & N_y & 1
\end{bmatrix}
\begin{bmatrix}
A_{\text{red}} \\
B_{\text{red}} \\
C_{\text{red}}
\end{bmatrix}
\]

Setup system of equations

Solve for plane equation coefficients \(A, B, C\)

\[
\begin{bmatrix}
L_x & L_y & 1 \\
M_x & M_y & 1 \\
N_x & N_y & 1
\end{bmatrix}^{-1}
\begin{bmatrix}
L_{\text{red}} \\
M_{\text{red}} \\
N_{\text{red}}
\end{bmatrix} =
\begin{bmatrix}
A_{\text{red}} \\
B_{\text{red}} \\
C_{\text{red}}
\end{bmatrix}
\]

Do the same for green, blue, and alpha (opacity)…
More Intuitive Way to Interpolate

Barycentric coordinates

\[
\begin{align*}
\text{Area}(PMN) & \alpha \\
\text{Area}(LMN) & \\
\text{Area}(LPN) & \beta \\
\text{Area}(LMP) & \gamma \\
\end{align*}
\]

Note: \( \alpha + \beta + \gamma = 0 \) by construction

\[
\text{attribute}(P) = \alpha \times \text{attribute}(L) + \beta \times \text{attribute}(M) + \gamma \times \text{attribute}(N)
\]
Top GPUs can setup over a billion triangles per second for rasterization.

Triangle setup & rasterization is just one of the (many, many) computation steps in GPU rendering.
Remaining Steps

- Depth interpolation
- Color update
- Scan-out to the display

Next time...
Programming tips

- 3D graphics, whether OpenGL or Direct3D or any other API, can be frustrating
- You write a bunch of code and the result is

Nothing but black window; where did your rendering go??
Things to Try

- Set your clear color to something other than black!
  - It is easy to draw things black accidentally so don’t make black the clear color
  - But black is the initial clear color
- Did you draw something for one frame, but the next frame draws nothing?
  - Are you using depth buffering? Did you forget to clear the depth buffer?
- Remember there are near and far clip planes so clipping in Z, not just X & Y
- Have you checked for glGetError?
  - Call glGetError once per frame while debugging so you can see errors that occur
  - For release code, take out the glGetError calls
- Not sure what state you are in?
  - Use glGetIntegerv or glGetFloatv or other query functions to make sure that OpenGL’s state is what you think it is
- Use glutSwapBuffers to flush your rendering and show to the visible window
  - Likewise glFinish makes sure all pending commands have finished
- Try reading
  - http://www.slideshare.net/Mark_Kilgard/avoiding-19-common-opengl-pitfalls
  - This is well worth the time wasted debugging a problem that could be avoided
Next Lecture

- Finish OpenGL pipeline
- Transforms and Graphics Math
  - Interpolation, vector math, and number representations for computer graphics
Thanks

- Presentation approach and figures from
  - David Luebke [2003]
  - Brandon Lloyd [2007]
  - Geometric Algebra for Computer Science
    [Dorst, Fontijne, Mann]
  - via Mark Kilgard