Today’s Outline

Triangle primitives
Triangle rasterization
Triangle (barycentric) interpolation

libst and libsgl
OpenGL Shape Primitives

OpenGL Examples

primitives.c, concave.c
Fundamental Primitive: Triangles

Why triangles?
- Lowest common denominator
- Easily break convex polygons into triangles
- Optimize the implementation of one primitive
- Triangles have unique properties
  - Guaranteed to be planar
  - Guaranteed to have a well-defined inside
  - Well-defined method (barycentric interpolation) for interpolating values in the interior

Graphics Types (Objects)

libst
STVector and STPoint Types

C++ classes

class STVector2;
class STPoint2;

class STVector3;
class STPoint3;

Overloaded operators for point and vector ops
Methods on points and vectors are different
See vect.cpp

Vectors and the Parallelogram Rule

Vector addition define for any number of dimensions
Vector Operations

Vectors: $u, v, w$

Parallelogram rule

$\langle \text{Vector}\rangle = \langle \text{Scalar}\rangle \times \langle \text{Vector}\rangle$

$v = \alpha w$

$\langle \text{Vector}\rangle = \langle \text{Vector}\rangle + \langle \text{Vector}\rangle$

$u = v + w$

Point Operations

Points: $p, q, r$

$\langle \text{Point}\rangle = \langle \text{Point}\rangle + \langle \text{Vector}\rangle$

$q = p + v$

$\langle \text{Vector}\rangle = \langle \text{Point}\rangle - \langle \text{Point}\rangle$

$v = q - p$

A point is an origin and a vector displacement
Illegal Operations

\[ \langle \text{Point} \rangle = \langle \text{Scalar} \rangle \times \langle \text{Point} \rangle \]
\[ p = \alpha q \]
\[ \langle \text{Point} \rangle = \langle \text{Point} \rangle + \langle \text{Point} \rangle \]
\[ p = q + r \]
\[ \langle \text{Vector} \rangle = \langle \text{Point} \rangle + \langle \text{Vector} \rangle \]
\[ v = p + w \]
\[ \langle \text{Point} \rangle = \langle \text{Point} \rangle - \langle \text{Point} \rangle \]
\[ p = q - r \]

STColor and STImage

```c
STImage* frog = new STImage("./frog.png");

STColor4ub pixel = frog->GetPixel(10,20);

frog->SetPixel(20,10,
              STColor4ub(255,0,0,255));

frog->Draw();

frog->Save("./badfrog.png");
```
sgl

```c
sglBeginTriangles(); // Triangle strip
  sglVertex(x, y);
  sglColor(r, g, b);
sglEnd();
```

sglLoadIdentity()
sglTranslate(), sglRotate(), sglScale()
sglPush(), sglPop()

---

**Rasterization**

![Diagram of rasterization process](image)
Rasterization Rules: Area Primitives

Output fragment if pixel center is **inside area**

Note that the vertices are not rounded to the nearest integer!

Triangle Rasterization Rule

Output fragment if pixel center is **inside the triangle**
Directed Line

\[ t = p_1 - p_0 = (x_1 - x_0, y_1 - y_0) \]

Perpendicular Vector in 2D

\[ \text{Perp}((x,y)) = (-y, x) \]
Normal to the Line

\[ \mathbf{n} \]
\[ \mathbf{t} \]
\[ \mathbf{p}_1 = (x_1, y_1) \]
\[ \mathbf{p}_0 = (x_0, y_0) \]

\[ \mathbf{t} = \mathbf{p}_1 - \mathbf{p}_0 = (x_1 - x_0, y_1 - y_0) \]

\[ \mathbf{n} = \text{Perp}(\mathbf{t}) = (y_0 - y_1, x_1 - x_0) \]

---

Line Equation

\[ \mathbf{t} \cdot \mathbf{n} = 0 \]
\[ \mathbf{p}_0 \]

\[ (\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} = 0 \]

This equation must be true for all point \( \mathbf{p} \) on the line
Line Equation

\[ n = (A, B) \]
\[ A = y_1 - y_0 \]
\[ B = x_0 - x_1 \]
\[ C = x_0 y_1 - y_0 x_1 \]

Line Divides Plane into 2 Half-Spaces

Normal \( n \) points to the left of the line
Inside (negative values) to the right
Triangle Rasterization

```c
rasterize( vert v[3] )
{
    for( int y=0; y<YRES; y++ )
        for( int x=0; x<XRES; x++ )
            if( inside3(v,x,y) )
                fragment(x,y);
}
```

Triangle Orientation

CCW (Front Facing)  CW (Back Facing)
Line Equation

Convention: Inside on the left for CCW polygons

```
makeline( vert& v0, vert& v1, line& l )
{
  l.a = v1.y - v0.y;
  l.b = v0.x - v1.x;
  l.c = -(l.a * v0.x + l.b * v0.y);
}
```

Point Inside Triangle Test

```
rasterize( vert v[3] )
{
  line 10, 11, 12;
  makeline(v[0],v[1],12);
  makeline(v[1],v[2],10);
  makeline(v[2],v[0],11);
  for( y=0; y<YRES; y++ )
  {
    for( x=0; x<XRES; x++ )
    {
      e0 = 10.a * x + 10.b * y + 10.c;
      e1 = 11.a * x + 11.b * y + 11.c;
      e2 = 12.a * x + 12.b * y + 12.c;
      if( e0<=0 & & e1<=0 & & e2<=0 )
        fragment(x,y);
    }
  }
}
```
Singularities

Singularities: Edges that touch pixels (e == 0)
Causes two fragments to be generated
- Wasted effort drawing duplicated fragments
- Problems with transparency (later lecture)

Not including singularities (e < 0) causes gaps

Handling Singularities

Create shadowed edges (thick lines)
Don’t draw pixels on shadowed edges
Solid drawn; hollow not drawn

```c
int shadow( line l ) {
    return (l.a>0) || (l.a == 0 && l.b > 0);
}
int inside( value e, line l ) {
    return (e == 0) ? !shadow(l) : (e < 0);
}
```
Better Point Inside Triangle Test

```cpp
rasterize( vert v[3] )
{
    line 10, 11, 12;
    makeline(v[0],v[1],12);
    makeline(v[1],v[2],10);
    makeline(v[2],v[0],11);
    for( y=0; y<YRES; y++ )
        for( x=0; x<XRES; x++ )
            e0 = 10.a * x + 10.b * y + 10.c;
            e1 = 11.a * x + 11.b * y + 11.c;
            e2 = 12.a * x + 12.b * y + 12.c;
            if( inside(e0,10)&inside(e1,11)&inside(e2,12) )
                fragment(x,y);
}
}
```

CS148 Lecture 3  Pat Hanrahan, Fall 2011

Compute Bounding Rectangle (BBox)

```cpp
bound3( vert v[3], bbox& b )
{
    b.xmin = ceil(min(v[0].x, v[1].x, v[2].x));
    b.xmax = ceil(max(v[0].x, v[1].x, v[2].x));
    b.ymin = ceil(min(v[0].y, v[1].y, v[2].y));
    b.ymax = ceil(max(v[0].y, v[1].y, v[2].y));
}
```

Calculate tight bound around the triangle
Round coordinates upward (ceil) to the nearest integer

CS148 Lecture 3  Pat Hanrahan, Fall 2011
Compute Bounding Rectangle (BBox)

```c
bound3( vert v[3], bbox& b )
{
    b.xmin = ceil(min(v[0].x, v[1].x, v[2].x));
    b.xmax = ceil(max(v[0].x, v[1].x, v[2].x));
    b.ymin = ceil(min(v[0].y, v[1].y, v[2].y));
    b.ymax = ceil(max(v[0].y, v[1].y, v[2].y));
}
```

Tested points indicated by filled circles
Don’t need to test hollow circles

More Efficient Bounding Box Version

```c
rasterize( vert v[3] )
{
    bbox b; bound3(v, b);
    line 10, 11, 12;
    makeline(v[0],v[1],12);
    makeline(v[1],v[2],10);
    makeline(v[2],v[0],11);

    for( y=b.ymin; y<b.ymax, y++ )
        for( x=b.xmin; x<b.xmax, x++ )
            e0 = 10.A * x + 10.B * y + 10.C;
            if( inside(e0,10)&inside(e1,11)&inside(e2,12) )
                fragment(x,y);
}
```
Barycentric Interpolation

Triangle

\[ \mathbf{p} = \alpha_0 \mathbf{p}_0 + \alpha_1 \mathbf{p}_1 + \alpha_2 \mathbf{p}_2 \]

\[ \alpha_0 + \alpha_1 + \alpha_2 = 1 \]
Finding Barycentric Coordinates

\[
\alpha_0 = \frac{\text{area}(p \ p_1 \ p_2)}{\text{area}(p_0 \ p_1 \ p_2)}
\]

\[
\alpha_1 = \frac{\text{area}(p_0 \ p \ p_2)}{\text{area}(p_0 \ p_1 \ p_2)}
\]

\[
\alpha_2 = \frac{\text{area}(p_0 \ p_1 \ p)}{\text{area}(p_0 \ p_1 \ p_2)}
\]

How to Compute Area?
Recall: Dot Product

\[ \mathbf{a} = (x_a, y_a, z_a) \]
\[ \mathbf{b} = (x_b, y_b, z_b) \]

\[ \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta = x_a x_b + y_a y_b + z_a z_b \]

The projection of \( \mathbf{a} \) onto \( \mathbf{b} \)

N. B. the projection is 0 if \( \mathbf{a} \) is perpendicular to \( \mathbf{b} \)

Recall: Vector Cross Product

\[ \mathbf{c} = \mathbf{a} \times \mathbf{b} \]
\[ x_c = y_a z_b - z_a y_b \]
\[ y_c = z_a x_b - x_a z_b \]
\[ z_c = x_a y_b - z_a x_b \]

c perpendicular to both \( \mathbf{a} \) and \( \mathbf{b} \)
\[ |\mathbf{c}| \text{ is equal to the area of quadrilateral } \mathbf{a} \mathbf{b} \]
Recall: Triple Product

Triple product is \( \text{volume}(a,b,c) \propto \text{area}(b,c) \times \text{length}(c) \)

Positive if \( a, b, c \) are right handed

Make 2D Point a 3D Vector!

\[
\begin{align*}
\mathbf{p}_1 &= (x_1, y_1, 1) \\
\mathbf{p}_2 &= (x_2, y_2, 1) \\
\mathbf{p}_3 &= (x_3, y_3, 1)
\end{align*}
\]
Compute 2D Area via 3D Volume

\[ l = \mathbf{p}_1 \times \mathbf{p}_2 \]
\[ z = 1 \]

area(\(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\)) \(\propto\) volume(\(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\))

volume(\(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\)) = \(\mathbf{p}_1 \times \mathbf{p}_2 \cdot \mathbf{p}_3 = l \cdot \mathbf{p}_3\)

Homogenous Coordinates

**Defn.** Homogenous coordinates: Convert 2D point to 3D point by adding a 1

**Line equation**

\[ l = \mathbf{p}_1 \times \mathbf{p}_1 \]
\[ = (a, b, c) \]
\[ = (y_1 - y_2, x_2 - x_1, x_1y_2 - x_2y_1) \]

**Inside test, area** \( l \cdot \mathbf{p} = ax + by + c \)

Neat!!
Barycentric Interpolation on Triangles

Can be used to interpolate colors
\[ r = a_0 r_0 + a_1 r_1 + a_2 r_2 \]
\[ g = a_0 g_0 + a_1 g_1 + a_2 g_2 \]
\[ b = a_0 b_0 + a_1 b_1 + a_2 b_2 \]

Can be used to interpolate texture coordinates
\[ u = a_0 u_0 + a_1 u_1 + a_2 u_2 \]
\[ v = a_0 v_0 + a_1 v_1 + a_2 v_2 \]

Can be used to interpolate z or depth
\[ z = a_0 z_0 + a_1 z_1 + a_2 c_2 \]

Can be used to interpolate normals ...

Things to Remember

Why triangles?

Triangle rasterization using inside tests

Barycentric interpolation