Clarification: Computing Edge Equations

**Solution:**
- We are only interested in the sign of the equation.
- Let's multiply the equation by denominator:
  - \( x_1 < x_c \); \((x_c, x_1)\) is positive, so the sign is preserved
    \[
    L(x,y) = \frac{(y-y_1)(x-x_1) + (y-y_1)(x-x_1)}{(x-x_1)} = -(y-y_1)(x-x_1) + (y-y_1)(x-x_1)
    \]
  - \( x > x_c \); \((x, x_c)\) is negative, multiply by -\((x_c-x)\) to preserve sign
    \[
    L(x,y) = -(x-x_1) \cdot \frac{(y-y_1)(x-x_1) - (y-y_1)(x-x_1)}{(x-x_1)} = -(y-y_1)(x-x_1) - (y-y_1)(x-x_1)
    \]

**Clarification: Computing Edge Equations**

**Summary:**
- Now we have only ONE equation
  \[
  L(x,y) = -(y-y_1)(x-x_1) + (y-y)(x-x_1)
  \]
- Works for both cases
- Also works for vertical lines!
Clarification: Interpolation with Plane Equation

Quantities vary linearly across image plane
- E.g.: $r = Ax + By + C$
  - $r$: red channel of the color
  - Same for $g, b, N_x, N_y, N_z, z$
- From info at vertices we know:
  - $r_1 = Ax_1 + By_1 + C$
  - $r_2 = Ax_2 + By_2 + C$
  - $r_3 = Ax_3 + By_3 + C$
- Solve for $A, B, C$
- One-time set-up cost per triangle

Clarification: Interpolation with Plane Equations

This approach works for colors, normals

Texture coordinates:
- For texture coordinates, we need to compute the image-space Barycentric coordinates $\alpha, \beta, \gamma$, then interpolate using formula from last lecture:
  $$ s = \frac{\alpha \cdot s_1 + \beta \cdot s_2 + \gamma \cdot s_3}{\alpha \cdot w_1 + \beta \cdot w_1 + \gamma \cdot w_2} $$

Observation:
- $\alpha, \beta, \gamma$ vary linearly across image plane!
- So: rather than setting up one plane equation for $s$, and one for $t$, we set up three plane equations, one each for $\alpha, \beta$, and $\gamma$, then use the above formula
  - Once we have these, we can also use $\alpha, \beta, \gamma$ for interpolating color, normal

Texture Mapping
- Real life objects have nonuniform colors, normals
- To generate realistic objects, reproduce coloring & normal variations = texture
- Can often replace complex geometric details

Texture image: 2D array of color values (texels)
Assigning texture coordinates $(s, t)$ at vertex with object coordinates $(x, y, z, w)$
- Use interpolated $(s, t)$ for texel lookup at each pixel
- Use value to modify a polygon’s color
  - Or other surface property
- Specified by programmer or artist

Texture Coordinate Interpolation

Perspective correct interpolation
- $\alpha, \beta, \gamma$:
  - Barycentric coordinates of a point $P$ in a triangle
  - $s_0, s_1, s_2$:
    - Texture coordinates of vertices
  - $w_0, w_1, w_2$:
    - Homogeneous coordinates of vertices
  - $$ s = \frac{\alpha \cdot s_0 + \beta \cdot s_1 + \gamma \cdot s_2}{\alpha \cdot w_0 + \beta \cdot w_1 + \gamma \cdot w_2} $$
**Reconstruction**
- How to deal with:
  - **Pixels that are much larger than texels?**
    - Apply filtering, “averaging”
  - **Pixels that are much smaller than texels?**
    - Interpolate

**Interpolating Textures**
- Nearest neighbor
- Bilinear
- Hermite

**MIPmapping**
use “image pyramid” to precompute averaged versions of the texture
- Without MIP-mapping
- With MIP-mapping
- Store whole pyramid in single block of memory

**Samples**
- Most things in the real world are **continuous**
- Everything in a computer is **discrete**
- The process of mapping a continuous function to a discrete one is called **sampling**
- The process of mapping a discrete function to a continuous one is called **reconstruction**
- The process of mapping a continuous variable to a discrete one is called **quantization**
- Rendering an image requires sampling and quantization
- Displaying an image involves reconstruction

**Images**
*An image is a 2D function I(x, y)*
- Specifies intensity for each point (x, y)
- (we consider each color channel independently)

**Point Sampling an Image**
- Simplest sampling is on a grid
- Sample depends solely on value at grid points
**Point Sampling**

*Multiply sample grid by image intensity to obtain a discrete set of points, or samples.*

**Image As Signal**

*Image as spatial signal*

**2D raster image**

- Discrete sampling of 2D spatial signal

**1D slice of raster image**

- Discrete sampling of 1D spatial signal

**Summing Waves I**

**1D Sampling and Reconstruction**

- Discrete sampling of 1D spatial signal

Examples from Foley, van Dam, Feiner, and Hughes

Pixel position across scanline

© Wolfgang Heidrich
1D Sampling and Reconstruction

**Problems**
- Jaggies – abrupt changes
- Lose data

**Falling Below Nyquist Rate**

*When sampling below Nyquist Rate, resulting signal looks like a lower-frequency one*
- This is aliasing!

**Aliasing**

*Incorrect appearance of high frequencies as low frequencies*

*To avoid: anti-aliasing*
- Supersample
  - Sample at higher frequency
- Low pass filtering
  - Remove high frequency function parts
  - Aka prefiltering, band-limiting

**Sampling Theorem**

- Continuous signal can be completely recovered from its samples

  \[ \text{iff} \]
  - Sampling rate greater than twice highest frequency present in signal

  *Claude Shannon*
The Rendering Pipeline

Geometry Processing
- Geometry Database
- Model/View Transform.
- Lighting
- Perspective Transform.
- Clipping
- Scan Conversion
- Texturing
- Depth Test
- Blending
- Frame-buffer

Blending

How might you combine multiple elements?
- New color A, old color B

Premultiplying Colors

Specify opacity with alpha channel: (r,g,b,a)
- \( a = 1 \): opaque, \( a = 0 \): transparent
- \( A \) over \( B \)
  - \( C = A \times \frac{1}{a} B \)

But what if \( B \) is also partially transparent?
- \( C = \alpha A + (1 - \alpha) B \)
- \( \gamma = \frac{\beta}{1 - \alpha} + \frac{\alpha}{1 - \beta} \)
- 3 multiplies, different equations for alpha vs. RGB

Premultiplying by alpha
- \( C' = \alpha C, B' = \frac{\beta}{1 - \alpha} B \)
- \( \gamma' = \gamma + \frac{\alpha}{1 - \beta} \)
- 1 multiply to find \( C \), same equations for alpha and RGB

OpenGL Blending

In OpenGL:
- Enable blending
  - glEnable( GL_BLEND )
- Specify alpha channel for colors
  - glColor4f( r, g, b, alpha )
- Specify blending function
  - E.g.: glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA )
  - \( C = \text{alpha}_\text{new} \times \text{new} + (1 - \text{alpha}_\text{new}) \times \text{old} \)

Double Buffering

Framebuffer:
- Piece of memory where the final image is written
- Problem:
  - The display needs to read the contents, cyclically, while the GPU is already working on the next frame
  - Could result in display of partially rendered images on screen
- Solution:
  - Have TWO buffers
    - One is currently displayed (front buffer)
    - One is rendered into for the next frame (back buffer)
**Interactive Object Selection**

*Move cursor over object, click*
- How to decide what is below?

*Ambiguity*
- Many 3D world objects map to same 2D point

*Common approaches*
- Manual ray intersection
- Bounding extents
- Selection region with hit list (OpenGL support)

---

**Manual Ray Intersection**

*Do all computation at application level*
- Map selection point to a ray
- Intersect ray with all objects in scene.

*Advantages*
- No library dependence

---

**Bounding Extents**

*Keep track of axis-aligned bounding rectangles*

*Advantages*
- Conceptually simple
- Easy to keep track of boxes in world space

---

**OpenGL Picking**

*"Render" image in picking mode*
- Pixels are never written to framebuffer
- Only store IDs of objects that would have been drawn

*Procedure*
- Set unique ID for each pickable object
- Call the regular sequence of glBegin/glVertex/glEnd commands
  - If possible, skip glColor, gNormal, glTexCoord etc. for performance
**Select/Hit**

*OpenGL support*
- Use small region around cursor for viewport
- Assign per-object integer keys (names)
- Redraw in special mode
- Store hit list of objects in region
- Examine hit list

**Viewport**

*Small rectangle around cursor*
- Change coord sys so fills viewport

*Why rectangle instead of point?*
- People aren’t great at positioning mouse
  - *Fitts’s Law*: time to acquire a target is function of the distance to and size of the target
- Allow several pixels of slop

**Tricky to compute**
- Invert viewport matrix, set up new orthogonal projection

*Simple utility command*
- `gluPickMatrix(x,y,w,h,viewport)`
  - `x,y`: cursor point
  - `w,h`: sensitivity/slop (in pixels)
- Push old setup first, so can pop it later

**Render Modes**

*`glRenderMode(mode)`*
- `GL_RENDER`: normal color buffer
  - default
- `GL_SELECT`: selection mode for picking
  - `GL_FEEDBACK`: report objects drawn

**Name Stack**

- “names” are just integers
- `glHitNames()`
- `glLoadName(name)`
- `glPushName(name);` or hierarchy supported by stack
- `glPopName(name);`
  - can have multiple names per object

**Hierarchical Names Example**

```c
for(int i = 0; i < 2; ++i) {
  glPushName();
  for(int j = 0; j < 2; ++j) {
    glPushName();
    glLoadName(HEAD);
    glCallList(nowManHeadDL);
    glLoadName(BODY);
    glCallList(nowManBodyDL);
    glPopName();
    glPopName();
  }
  glPopMatrix();
}
```

**Hit List**

- `glSelectBuffer(buffersize, *buffer)`
  - where to store hit list data
- on hit, copy entire contents of name stack to output buffer.
- hit record
  - number of names on stack
  - minimum and minimum depth of object vertices
    - depth lies in the z-buffer range [0,1]
    - multiplied by $2^{32} - 1 \text{ then rounded to nearest int}$

**Integrated vs. Separate Pick Function**

*Integrate: use same function to draw and pick*
- Simpler to code
- Name stack commands ignored in render mode

*Separate: customize functions for each*
- Potentially more efficient
- Can avoid drawing unpickable objects

**Select/Hit**

**Advantages**
- Faster
  - OpenGL support means hardware excel
  - Only do clipping work, no shading or rasterization
- Flexible precision
  - Size of region controllable
- Flexible architecture
  - Custom code possible, e.g. guaranteed frame rate

**Disadvantages**
- More complex

**Coming Up...**

**Monday:**
- Shadows, Modern GPU features

**Wednesday:**
- Color