Line Aliasing

Ideal raster line is one pixel wide

All line segments, other than vertical and horizontal segments, partially cover pixels:

• Simple algorithms color only whole pixels
• Lead to the “jaggies” or aliasing
• Similar issue for polygons
• Antialiasing is a process to remove the “jaggies” (= common name for jagged edges of rasterized geometric primitives).
Antialiasing by Area Averaging

Color multiple pixels for each $x$, depending on coverage by ideal line

original

antialiased

magnified
Antialiasing

Can try to color a pixel by adding a fraction of its color to the frame buffer
  Fraction depends on percentage of pixel covered by fragment
  Fraction depends on whether there is overlap

no overlap

overlap
Area Averaging

Use average area $a_1 + a_2 - a_1 a_2$ as blending factor
OpenGL Antialiasing

OpenGL supports antialiasing of all geometric primitives by enabling both GL_BLEND and one of the constants listed below. It can be enabled separately for points, lines, or polygons:

```c
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```
Polygon Aliasing

Aliasing problems can be serious for polygons

- Jaggedness of edges
- Small polygons neglected
- Need compositing so that color of one polygon does not totally determine color of pixel
- In certain cases, \texttt{GL\_POLYGON\_SMOOTH} may not provide sufficient results, particularly if polygons share edges. As such, using the accumulation buffer for full scene antialiasing may be a better solution.

\textit{example where all three polygons should contribute to color}
What about using the OpenGL Accumulation Buffer?

To address problems of compositing into color buffers:
  • limited color resolution
  • clamping
  • loss of accuracy

The accumulation buffer acts as a “floating point” color buffer:
  • accumulate into accumulation buffer
  • transfer results to frame buffer
Accessing Accumulation Buffer

\texttt{glAccum( op, value )}

operations:

within the accumulation buffer: \texttt{GL\_ADD, GL\_MULT}
from read buffer: \texttt{GL\_ACCUM, GL\_LOAD}
transfer back to write buffer: \texttt{GL\_RETURN}

\texttt{glAccum(\texttt{GL\_ACCUM, 0.5})} multiplies each value in write buffer by 0.5 and adds to accumulation buffer
Accumulation Buffer
Applications

• Compositing
• Depth of Field
• Filtering
• Motion Blur
• Full Scene Antialiasing

2017-04-24
Full Scene Antialiasing with the Accumulation Buffer: *Jittering the view*

Each time we move the viewer, the image shifts

- Different aliasing artifacts in each image
- Averaging images using accumulation buffer averages out these artifacts
Depth of Focus with the Accumulation Buffer: *Keeping a Plane in Focus*

Jitter the viewer to keep one plane unchanged

```
<table>
<thead>
<tr>
<th>Back Plane</th>
<th>Focal Plane</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front Plane</td>
<td></td>
</tr>
<tr>
<td>eye pos₁</td>
<td>eye pos₂</td>
</tr>
</tbody>
</table>
```
OpenGL ES Full-Screen Antialiasing?

one small problem:

OpenGL ES does not support the Accumulation Buffer...

...use render-to-texture instead!
OpenGL ES framebuffer objects (FBOs)

Framebuffer objects are the destination for rendering commands. When you create a framebuffer object, you have precise control over its storage for color, depth, and stencil data.

You provide this storage by attaching images to the framebuffer, as shown above.
OpenGL ES framebuffer objects (FBOs)

The most common image attachment is a renderbuffer object.

You can also attach an OpenGL ES texture to the color attachment point of a framebuffer, which means that any drawing commands are rendered into the texture. Later, the texture can act as an input to future rendering commands.

You can also create multiple framebuffer objects in an single rendering context. You might do this so that you can share the same rendering pipeline and OpenGL ES resources between multiple framebuffers.
OpenGL ES framebuffer objects (FBOs)

You can also attach an OpenGL ES texture to the color attachment point of a framebuffer, which means that any drawing commands are rendered into the texture.

1. Create the framebuffer object (using the same procedure as in Creating Offscreen Framebuffer Objects).
2. Create the destination texture, and attach it to the framebuffer's color attachment point.

```c
// create the texture
GLint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);
```

3. Allocate and attach a depth buffer (as before).
4. Test the framebuffer for completeness (as before).

Lecture 22: Texture Mapping

Introduce Mapping Methods
  * Texture Mapping
  * Environment Mapping
  * Bump Mapping

Consider basic strategies
  * Forward vs backward mapping
  * Point sampling vs area averaging
The Limits of Geometric Modeling

Although graphics cards can render over 10 million polygons per second, that number is insufficient for many phenomena:
- Clouds
- Grass
- Terrain
- Skin
Modeling an Orange

Consider the problem of modeling an orange (the fruit)
Start with an orange-colored sphere
   Too simple
Replace sphere with a more complex shape
   Does not capture surface characteristics (small dimples)
   Takes too many polygons to model all the dimples
Modeling an Orange (2)

Take a picture of a real orange, scan it, and “paste” onto simple geometric model
   This process is known as texture mapping
Still might not be sufficient because resulting surface will be smooth
   Need to change local shape
Bump mapping
We'll look at Three Types of Mapping

Texture Mapping
- Uses images to fill inside of polygons

Environment (reflection mapping)
- Uses a picture of the environment for texture maps
- Allows simulation of highly specular surfaces

Bump mapping
- Emulates altering normal vectors during the rendering process
Texture Mapping

geometric model

texture mapped
Environment Mapping
Bump Mapping
Where does mapping take place?

Mapping techniques are implemented at the end of the rendering pipeline

Very efficient because few polygons make it past the clipper
Texture Mapping Strategies

Basic mapping strategies
- Forward vs backward mapping
- Point sampling vs area averaging
Is it simple?

Although the idea is simple---map an image to a surface---there are 3 or 4 coordinate systems involved.
Coordinate Systems

Parametric coordinates
   May be used to model curves and surfaces
Texture coordinates
   Used to identify points in the image to be mapped
Object or World Coordinates
   Conceptually, where the mapping takes place
Window Coordinates
   Where the final image is really produced
Texture Mapping

texture coordinates

world coordinates

window coordinates
Mapping Functions

Basic problem is how to find the maps
Consider mapping from texture coordinates to a point a surface
Appear to need three functions
\[ x = x(s,t) \]
\[ y = y(s,t) \]
\[ z = z(s,t) \]
But we really want to go the other way
Backward Mapping

We really want to go backwards

Given a pixel, we want to know to which point on an object it corresponds

Given a point on an object, we want to know to which point in the texture it corresponds

Need a map of the form

\[ s = s(x, y, z) \]
\[ t = t(x, y, z) \]

Such functions are difficult to find in general
Two-part mapping

One solution to the mapping problem is to first map the texture to a simple intermediate surface
Example: map to cylinder
Cylindrical Mapping

parametric cylinder

\[ x = r \cos 2\pi u \]
\[ y = r \sin 2\pi u \]
\[ z = \frac{v}{h} \]

maps rectangle in \( u,v \) space to cylinder of radius \( r \) and height \( h \) in world coordinates

\[ s = u \]
\[ t = v \]

maps from texture space
Spherical Map

We can use a parametric sphere

\[ x = r \cos 2pu \]
\[ y = r \sin 2pu \cos 2pv \]
\[ z = r \sin 2pu \sin 2pv \]

in a similar manner to the cylinder but have to decide where to put the distortion

Spheres are used in environmental maps
Box Mapping

Easy to use with simple orthographic projection
Also used in environment maps
Two-stage Mapping

Map from intermediate object to actual object
Normals from intermediate to actual
Normals from actual to intermediate
Vectors from center of intermediate
Aliasing

Point sampling of the texture can lead to aliasing errors

miss blue stripes

point samples in texture space

point samples in u,v (or x,y,z) space
Area Averaging

A better but slower option is to use area averaging.

Note that *preimage* of pixel is curved.
Texture Mapping in OpenGL ES

Introduce OpenGL ES texture mapping

two-dimensional texture maps
assigning texture coordinates
forming texture images
Basic Strategy

Three steps to applying a texture

specify the texture
read or generate image
assign to texture
enable texturing

assign texture coordinates to vertices
Proper mapping function is left to application

specify texture parameters
wrapping, filtering
Texture Mapping

geometry

image

display
Texture Example

The texture is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective.
Texture Mapping and the OpenGL ES Pipeline

Images and geometry flow through separate pipelines that join during fragment processing. “Complex” textures do not affect geometric complexity.
Specifying a Texture Image

Define a texture image from an array of texels (texture elements) in CPU memory.

Use an image in a standard format such as JPEG:
- Scanned image
- Generate by application code

OpenGL ES supports only 2 dimensional texture maps:
- no need to enable as in desktop OpenGL
- desktop OpenGL supports 1-4 dimensional texture maps
Define Image as a Texture

`glTexImage2D( target, level, components, 
    w, h, border, format, type, texels )`

target: type of texture, e.g. `GL_TEXTURE_2D`
level: used for mipmapping (discussed later)
components: elements per texel
w, h: width and height of texels in pixels
border: used for smoothing (discussed later)
format and type: describe texels
texels: pointer to texel array

equivalent:
`glTexImage2D( GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB, 
    GL_UNSIGNED_BYTE, my_texels )`
A Checkerboard Image

def image1 = Array(4*texSize*texSize)
    for ( var i = 0; i < texSize; i++ ) {
        for ( var j = 0; j < texSize; j++ ) {
            patchx = floor(i/(texSize/numChecks))
            patchy = floor(j/(texSize/numChecks))
            if (patchx%2 ^ patchy%2)
                { c = 255 }
            else
                { c = 0 }
            image1[4*i*texSize+4*j] = c
            image1[4*i*texSize+4*j+1] = c
            image1[4*i*texSize+4*j+2] = c
            image1[4*i*texSize+4*j+3] = 255
        }
    }

Mapping a Texture

Based on parametric texture coordinates
Specify as a 2D vertex attribute

![Diagram showing mapping of a texture from texture space to object space]
Interpolation

OpenGL ES uses interpolation to find proper texels from specified texture coordinates

Can be distortions

- good selection of tex coordinates
- poor selection of tex coordinates
- texture stretched over trapezoid showing effects of bilinear interpolation
Texture Objects and Parameters

Introduce the OpenGL ES texture functions and options
  texture objects
  texture parameters
  example code
Using Texture Objects

specify textures in texture objects
set texture filter
set texture function
set texture wrap mode
set optional perspective correction hint
bind texture object
enable texturing
supply texture coordinates for vertex coordinates can also be generated
Texture Parameters

OpenGL ES has a variety of parameters that determine how texture is applied:

- Wrapping parameters determine what happens if $s$ and $t$ are outside the $(0,1)$ range.
- Filter modes allow us to use area averaging instead of point samples.
- Mipmapping allows us to use textures at multiple resolutions.
- Environment parameters determine how texture mapping interacts with shading.
Wrapping Mode

Clamping: if \( s,t > 1 \) use 1, if \( s,t < 0 \) use 0

Wrapping: use \( s,t \) modulo 1

\[
\text{glTexParameteri}(
GL\_TEXTURE\_2D, \\
GL\_TEXTURE\_WRAP\_S, GL\_CLAMP)
\]

\[
\text{glTexParameteri}(
GL\_TEXTURE\_2D, \\
GL\_TEXTURE\_WRAP\_T, GL\_REPEAT)
\]
Magnification and Minification

More than one texel can cover a pixel (*minification*) or more than one pixel can cover a texel (*magnification*)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values
Filter Modes

Modes determined by

\texttt{glTexParameteri( target, type, mode )}

\texttt{glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST)}

\texttt{glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR)}
Mipmapped Textures

Mipmapping allows for prefILTERED texture maps of decreasing resolutions
Lessens interpolation errors for smaller textured objects
Declare mipmap level during texture definition

```c
glTexImage2D(GL_TEXTURE_2D, level, … )
```
Example

point sampling

mipmapped point sampling

linear filtering

mipmapped linear filtering
Modeling an Orange (2)

We've seen that if we take a picture of a real orange, scan it, and “paste” onto simple geometric model, this process is known as *texture mapping*. Texture mapping might not be sufficient because resulting surface will be *smooth*.

Need to change local shape:

*Bump mapping*
Three Types of Mapping

Texture Mapping
Uses images to fill inside of polygons

Environment (reflection mapping)
Uses a picture of the environment for texture maps
Allows simulation of highly specular surfaces

Bump mapping
Emulates altering normal vectors during the rendering process
Bump Mapping

Main concepts:

• Perturb normal for each fragment
• Store perturbation as textures
Normalization Maps

Cube maps can be viewed as lookup tables
(Containing 1- 2- 3- 4- dimensional variables)
Vector from origin is pointer into table

Example: store normalized value of vector in the map
Same for all points on that vector
Use “normalization map” instead of normalization function
Lookup replaces sqrt, mults and adds
Bump Mapping

Mapping methods:

Texture mapping
Environmental (reflection) mapping
    Variant of texture mapping
Bump mapping
    Solves flatness problem of texture mapping
Modeling an Orange

Consider modeling an orange

Texture map a photo of an orange onto a surface
Captures dimples
Will not be correct if we move viewer or light
We have shades of dimples rather than their correct orientation

Ideally we need to perturb normal across surface of object and compute a new color at each interior point
Bump Mapping (as from Blinn's paper)

Consider a smooth surface

\[ \mathbf{n} \]

\[ \mathbf{p} \]
Rougher Version of Surface
Equations

• a point \( p \) on a parametric surface:

\[
p(u,v) = [x(u,v), y(u,v), z(u,v)]^T
\]

• the partial derivatives at the point are:

\[
p_u = \begin{bmatrix}
\frac{\partial x}{\partial u} & \frac{\partial y}{\partial u} & \frac{\partial z}{\partial u}
\end{bmatrix}^T
\]

\[
p_v = \begin{bmatrix}
\frac{\partial x}{\partial v} & \frac{\partial y}{\partial v} & \frac{\partial z}{\partial v}
\end{bmatrix}^T
\]

lie on the plane that is tangent to the surface at the point \( p \).

• the unit normal \( n \) at the point \( p \) is:

\[
n = \frac{(p_u \times p_v)}{|p_u \times p_v|}
\]
Tangent Plane
Displacement Function

\[ p' = p + d(u,v) \mathbf{n} \]

\( d(u,v) \) is the \textit{bump} or \textit{displacement} function, assumed to be known and small:

\[ |d(u,v)| \ll 1 \]
Perturbed Normal

• instead of perturbing $p$, which would be adding to the geometric complexity, we perturb the normal $n$:

$$n' = p'_u \times p'_v$$

• we differentiate the equation for $p'$ (from the previous slide):

$$p'_u = p_u + \frac{\partial d}{\partial u}n + d(u,v)n_u$$

$$p'_v = p_v + \frac{\partial d}{\partial v}n + d(u,v)n_v$$

If $d$ is small, we can neglect the last term
Computing the normal from the cross product 
(\( p'_u \times p'v \))
(\( n \times n = 0 \))

\[ n' = p'_u \times p'v \]

\[ \approx n + (\frac{\partial d}{\partial u})n \times p_v + (\frac{\partial d}{\partial v})n \times p_u \]

The vectors \( n \times p_v \) and \( n \times p_u \) lie in the tangent plane.

Hence the normal is displaced in the tangent plane.

Therefore:
- precompute the arrays \( \frac{\partial d}{\partial u} \) and \( \frac{\partial d}{\partial v} \)
- perturb the normal during shading
Tangent Space

• $p'_u$ and $p'_v$
  are not necessarily orthogonal to each other (even though they are both in the plane orthogonal to $n'$)

• we can obtain an orthogonal basis and a corresponding rotation matrix, by using the cross product of
  
  $n_1 = (n') / |n'|$ ---- the normalized normal
  and
  
  $t_1 = (p'_u) / |p'_u|$ ---- the tangent vector
Tangent Space

• the cross product of
  \( \mathbf{n}_1 = (\mathbf{n}') / |\mathbf{n}'| \)
  and
  \( \mathbf{t}_1 = (\mathbf{p}'_u) / |\mathbf{p}'_u| \)

  ---- the normalized normal
  ---- the tangent vector

• is the **binormal** vector:

  \( \mathbf{b}_1 = \mathbf{n}_1 \times \mathbf{t}_1 \)
Tangent Space

• The three vectors, when placed in a matrix:

$$M_1 = [t_1, b_1, n_1]^T$$

represent a rotation matrix, that will convert points represented in the original space into points represented in a new space defined by the three vectors:

$$t_1, b_1, n_1$$

• This new space is called **tangent space**.
Tangent Space

\[ \mathbf{m}_l = [\mathbf{t}_l, \mathbf{b}_l, \mathbf{n}_l]^T \]

- the tangent and binormal vectors can change at each point on the surface
- therefore, the tangent space is a local coordinate system.