Announcements

- No new homework assignment for this Friday
- Homework assignment #4 due Friday, Oct 29
  - To be presented between 2-4pm in lab 260
- Late submissions for project #3 accepted until this Friday
- Midterm exam: Thursday, Oct 21, 2-3:20pm, WLH 2005
- Midterm tutorial: Tuesday, Oct 19, noon-1:45pm, Atkinson Hall, room 4004
  - Tutors: Jurgen and Phi
  - We will have blank index cards for everybody
- Phi’s office hours on Oct 19 and 21 are cancelled
Lecture Overview

- Shader programming
  - Fragment shaders
  - GLSL
- Texturing
  - Overview
  - Texture coordinate assignment
  - Anti-aliasing
Fragment Programs

- **Fragment data**: Interpolated vertex attributes, additional fragment attributes

- **Uniform parameters**: OpenGL state, application specified parameters

- From rasterizer to fixed framebuffer access functionality (z-buffering, etc.)

- Fragment color, depth
Types of Input Data

**Fragment data**
- Change for each execution of the fragment program
- Interpolated from vertex data during rasterization, *varying* variables
- Interpolated fragment color, texture coordinates
- Standard OpenGL fragment data accessible through *predefined* variables
  ```
  varying vec4 gl_Color;
  varying vec4 gl_TexCoord[ ];
  etc.
  ```
- **Note** *varying* storage classifier, read-only
- User defined data possible, too
Types of Input Data

**Uniform parameters**
- Same as in vertex shader
- OpenGL state
- Application defined parameters
  - Use `glGetUniformLocation`, `glUniform*` in application
Output Variables

- Predefined outputs
  - gl_FragColor
  - gl_FragDepth

- OpenGL writes these to the frame buffer
- Result is undefined if you do not write these variables
“Hello World” Fragment Program

- `main()` function is executed for every fragment
- Use predefined variables
- Draws every pixel in green color

```c
void main()
{
    gl_FragColor = vec4(0.0,1.0,0.0,1.0);
}
```
Examples

- Fancy per pixel shading
  - Bump mapping
  - Displacement mapping
  - Realistic reflection models
  - Cartoon shading
  - Shadows

- Most often, vertex and fragment shaders work together to achieve a desired effect
Fragment Programs

Limitations

- Cannot read frame buffer (color, depth)
- Can only write to frame buffer pixel that corresponds to fragment being processed
  - No random write access to frame buffer
- Limited number of varying variables passed from vertex to fragment shader
- Limited number of application-defined uniform parameters
Lecture Overview

- **Shader programming**
  - Fragment shaders
  - GLSL

- **Texturing**
  - Overview
  - Texture coordinate assignment
  - Anti-aliasing
GLSL Main Features

- Similar to C language
- `attribute`, `uniform`, `varying` storage classifiers
- Set of predefined variables
  - Access per vertex, per fragment data
  - Access OpenGL state
- Built-in vector data types, vector operations
- No pointers
- No direct access to data, variables in your C++ code
Per-pixel Diffuse Lighting

// Vertex shader, stored in file diffuse.vert
varying vec3 normal, lightDir;
void main()
{
    lightDir = normalize(vec3(gl_LightSource[0].position));
    normal = normalize(gl_NormalMatrix * gl_Normal);
    gl_Position = ftransform();
}

// Pixel shader, stored in file diffuse.frag
varying vec3 normal, lightDir;
void main()
{
    gl_FragColor =
    gl_LightSource[0].diffuse * max(dot(normalize(normal), normalize(lightDir)),0.0) *
    gl_FrontMaterial.diffuse;
}
### GLSL Quick Reference Guide

#### OpenAL® Shading Language (GLSL) Quick Reference Guide

*GLSL version 1.10, as included in OpenAL® 2.0, and specified by 'The OpenAL® Shading Language', version 1.10.99. Section and page numbers refer to that version of the spec.*

### Data Types (4.1 p16)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float, vec2, vec3, vec4</td>
<td>Basic floating-point types</td>
</tr>
<tr>
<td>int, vec2, vec3, vec4</td>
<td>Integer and vector types</td>
</tr>
<tr>
<td>mat2, mat3, mat4</td>
<td>Matrix types</td>
</tr>
<tr>
<td>sampler2D, sampler3D, samplerCube</td>
<td>Texture sampler types</td>
</tr>
</tbody>
</table>

### Data Type Qualifiers (4.3 p22)

- `uniform` indicates a variable declared in a shader is constant and shared among all shaders in the same program
- `attribute` indicates a variable is declared in a vertex shader and passed to the fragment shader
- `varying` indicates a variable is declared in a fragment shader and passed to the vertex shader
- `constant` indicates a variable is declared in a shader and shared among all shaders in the same program

### Function Parameters

- `in` indicates a parameter is passed to a function
- `out` indicates a parameter is passed to a function and also returned
- `inout` indicates a parameter is passed to a function and can be modified but not returned

### Vector Components (5.5 p 20)

When using the GLSL type qualifier `s`, components may not be mixed across sets

<table>
<thead>
<tr>
<th>Type</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec2</td>
<td>x, y</td>
<td>2-component vector</td>
</tr>
<tr>
<td>vec3</td>
<td>x, y, z</td>
<td>3-component vector</td>
</tr>
</tbody>
</table>

### Preprocessor (1.3 p9)

- `#version 1.100` | GLSL version declaration and extensions protocol

---

http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf
Tutorials and Documentation

- **OpenGL and GLSL specifications**

- **GLSL tutorials**

- **OpenGL Programming Guide (Red Book)**
- **OpenGL Shading Language (Orange Book)**
Lecture Overview

- **Shader programming**
  - Fragment shaders
  - GLSL

- **Texturing**
  - Overview
  - Texture coordinate assignment
  - Anti-aliasing
Large Triangles

Pros:
- Often sufficient for simple geometry
- Fast to render

Cons:
- Per vertex colors look bad
- Need more interesting surfaces
  - Detailed color variation, small scale bumps, roughness
Texture Mapping

- Attach textures (images) onto surfaces
- Same triangle count, much more realistic appearance
Texture Sources

- Take photographs
- Paint directly on surfaces with a 3D modeling program (Maya, 3ds Max, Blender, etc.)
- Use existing images from disk

Images by Paul Debevec

Texture painting in Maya
Texture Mapping

- Goal: assign locations in texture to locations on 3D geometry
- Introduce texture space
  - Texture pixels (texels) have texture coordinates \((u,v)\)
- Convention
  - Bottom left corner of texture is at \((u,v) = (0,0)\)
  - Top right corner is at \((u,v) = (1,1)\)
Texture Mapping

- Store texture coordinates at each triangle vertex

$$v_1$$

$$(u,v) = (0.65,0.75)$$

$$v_0$$

$$(u,v) = (0.6,0.4)$$

$$v_2$$

$$(u,v) = (0.4,0.45)$$

Triangle (in any space before projection)
Texture Mapping

- Each point on triangle has barycentric coordinates $\alpha$, $\beta$, $\gamma$
- Use barycentric coordinates to interpolate texture coordinates

$(u,v) = (0.65, 0.75)$

$$v_1 = \alpha \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix} + \beta \begin{bmatrix} 0.65 \\ 0.75 \end{bmatrix} + \gamma \begin{bmatrix} 0.4 \\ 0.475 \end{bmatrix}$$

$(u,v) = (0.6, 0.4)$

$(u,v) = (0.4, 0.45)$

Triangle (in any space before projection)

Texture space
Texture Mapping

- Each point on triangle has corresponding point in texture

Triangle (in any space before projection)

\[ v_1 \]
\[ (u,v) = (0.65,0.75) \]

\[ v_0 \]
\[ (u,v) = (0.6,0.4) \]

\[ v_2 \]
\[ (u,v) = (0.4,0.45) \]
Rendering

- Given
  - Texture coordinates at each vertex
  - Texture image
- At each pixel, use barycentric coordinates to interpolate texture coordinates
- Look up corresponding texel
- Paint current pixel with texel color
- All computations are done on the GPU
Texture Mapping

Primitives

Modeling and viewing transformation

Shading

Projection

Rasterization

Fragment processing

Frame-buffer access (z-buffering)

Image

Includes texture mapping
Rendering

- Linear interpolation in image space does not correspond to linear interpolation in 3D
- Need to do perspectively correct interpolation!

Linear interpolation in image coordinates

Perspectively correct interpolation
Perspectively Correct Interpolation

- Point in image space with barycentric coordinates $\alpha$, $\beta$, $\gamma$
- Triangle vertices with homogeneous coordinates
  1. $a_w, b_w, c_w$
  2. $\frac{1}{w} = \alpha \frac{1}{a_w} + \beta \frac{1}{b_w} + \gamma \frac{1}{c_w}$
  3. $\frac{u}{w} = \alpha \frac{a_u}{a_w} + \beta \frac{b_u}{b_w} + \gamma \frac{c_u}{c_w}$
  
  \[ u = \frac{u}{w} \frac{1}{w} \]

- Same for $v$ texture coordinate
Texture Look-Up

- Given interpolated texture coordinates \((u, v)\) at current pixel
- Closest four texels in texture space are at \((u_0, v_0), (u_1, v_0), (u_1, v_0), (u_1, v_1)\)
- How to compute pixel color?
Nearest-Neighbor Interpolation

- Use color of closest texel

- Simple, but low quality and aliasing
Bilinear Interpolation

1. Linear interpolation horizontally

\[ w_u = \frac{u - u_0}{u_1 - u_0} \]

\[ c_b = tex(u_0, v_0)(1 - w_u) + tex(u_1, v_0)w_u \]

\[ c_t = tex(u_0, v_1)(1 - w_u) + tex(u_1, v_1)w_u \]
Bilinear Interpolation

1. Linear interpolation horizontally

\[ w_u = \frac{u - u_0}{u_1 - u_0} \]
\[ c_b = \text{tex}(u_0, v_0)(1 - w_u) + \text{tex}(u_1, v_0)w_u \]
\[ c_t = \text{tex}(u_0, v_1)(1 - w_u) + \text{tex}(u_1, v_1)w_u \]

2. Linear interpolation vertically

\[ w_v = \frac{v - v_0}{v_1 - v_0} \]
\[ c = c_b(1 - w_v) + c_t w_v \]
Basic Shaders for Texturing

// Need to initialize texture using OpenGL API calls.
// See base code.

// Vertex shader
void main()
{
    gl_Position = ftransform();
}

// Fragment shader
uniform sampler2D tex;
void main()
{
    gl_FragColor = texture2D(tex, gl_TexCoord[0].st);
}
Tiling

- Image exists from $[0,0] \times [1,1]$ in texture space
- $(u,v)$ texture coordinates may go outside that range
- *Tiling* and *wrapping* rules for out-of-range coordinates
Tiling

- Repeat the texture
  - Seams, unless the texture lines up on the edges
Clamping

- Use the edge value everywhere outside the data
- Or, ignore the texture outside 0-1
Mirroring

- Flip left-to-right and top-to-bottom
  - All the edges line up

Texture Space

37
Lecture Overview

- **Shader programming**
  - Fragment shaders
  - GLSL

- **Texturing**
  - Overview
  - *Texture coordinate assignment*
  - Anti-aliasing
Texture Coordinate Assignment

Surface parameterization

- Mapping between 3D positions on surface and 2D texture coordinates
  - In practice, defined by texture coordinates of triangle vertices
- Various options to establish a parameterization
  - Parametric mapping
  - Orthographic mapping
  - Projective mapping
  - Spherical mapping
  - Cylindrical mapping
  - Skin mapping
Parametric Mapping

- Surface given by parametric functions
  \[ x = f(u, v) \quad y = f(u, v) \quad z = f(u, v) \]
- Very common in CAD
- Use \((u, v)\) parameters as texture coordinates
Orthographic Mapping

- Use linear transformation of object’s xyz coordinates
- For example

\[
\begin{bmatrix}
  u \\
  v
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0
\end{bmatrix} \begin{bmatrix}
  x \\
  y \\
  z \\
  w
\end{bmatrix}
\]
Projective Mapping

- Use perspective projection of $xyz$ coordinates
  - OpenGL provides GL_TEXTURE matrix to apply perspective projection on texture coordinates
- Useful to achieve “fake” lighting effects
Spherical Mapping

- Use, e.g., spherical coordinates for sphere
- Place object in sphere
- “shrink-wrap” sphere to object
Cylindrical Mapping

- Similar as spherical mapping, but with cylinder
- Useful for faces
Skin Mapping

- Complex technique to unfold surface onto plane
- Preserve area and angle
- Sophisticated mathematics

[Piponi2000]
Lecture Overview

- **Shader programming**
  - Fragment shaders
  - GLSL

- **Texturing**
  - Overview
  - Texture coordinate assignment
  - Anti-aliasing
Example for Aliasing

- What causes this?
Aliasing

Sufficiently sampled, no aliasing

Insufficiently sampled, aliasing

High frequencies in the input data appear as low frequencies in the sampled signal
Antialiasing: Intuition

- Pixel may cover large area on triangle in camera space
Antialiasing: Intuition

- Pixel may cover large area on triangle in camera space
- Corresponds to many texels in texture space
- Need to compute average

![Diagram of Image plane, Camera space, and Texture space with pixel area and texels](image)
Antialiasing: Mathematics

- Pixels are samples, not little squares
  http://www.alvyray.com/memos/6_pixel.pdf
- Use frequency analysis to explain sampling artifacts
  - Fourier transforms
- Antialiasing is achieved through low-pass filtering
- For more information:
  - Glassner: Principles of digital image synthesis
Antialiasing Using Mip-Maps

- Averaging over texels is expensive
  - Many texels as objects get smaller
  - Large memory access and computation cost
- Precompute filtered (averaged) textures
  - Mip-maps
- Practical solution to aliasing problem
  - Fast and simple
  - Available in OpenGL, implemented in GPUs
  - Reasonable quality
Mipmaps

- MIP stands for multum in parvo = “much in little” (Williams 1983)

Before rendering

- Pre-compute and store down scaled versions of textures
  - Reduce resolution by factors of two successively
  - Use high quality filtering (averaging) scheme
- Increases memory cost by $1/3$
  - $1/3 = 1/4 + 1/16 + 1/64 + …$
- Width and height of texture need to be powers of two
Mipmaps

- Example: resolutions 512x512, 256x256, 128x128, 64x64, 32x32 pixels

“multum in parvo”
Mipmaps

- One texel in level 4 is the average of $4^4 = 256$ texels in level 0

“multum in parvo”
Mipmaps

Level 0

Level 1

Level 2

Level 3

Level 4
Rendering With Mipmaps

- “Mipmapping”
- Interpolate texture coordinates of each pixel as without mipmapping
- Compute approximate size of pixel in texture space
- Look up color in nearest mipmap
  - E.g., if pixel corresponds to 10x10 texels use mipmap level 3
  - Use nearest neighbor or bilinear interpolation as before
Mipmapping

- Image plane
- Camera space
- Texture space

- Texels
- Pixel area

- Mip-map level 0
- Mip-map level 1
- Mip-map level 2
- Mip-map level 3
Nearest Mipmap, Nearest Neighbor

- Visible transition between mipmap levels
Nearest Mipmap, Bilinear

- Visible transition between mipmap levels
Trilinear Mipmapping

- Use two nearest mipmap levels
  - E.g., if pixel corresponds to 10x10 texels, use mipmap levels 3 (8x8) and 4 (16x16)
- Perform bilinear interpolation in both mip-maps
- Linearly interpolate between the results
- Requires access to 8 texels for each pixel
- Standard method, supported by hardware with no performance penalty
Nearest Mipmap, Bilinear

- Visible transition between mipmap levels
Trilinear Mipmapping

- Smooth transition between mipmap levels
Next Lecture

- **Thursday:**
  - Midterm Exam

- **Next Tuesday:**
  - Advanced Texture Mapping Techniques