Announcements

- Find homework scores on Gradesource
- Homework Project 3 due this Friday, Oct 23, demo in lab 250 between 2-5pm
- Homework late submission policy:
  - This Friday accepting late submissions of project 2
- Midterm exam: Thursday, Oct 29
- No lab hours next Friday! If you need to get graded late you must submit by next Thursday.
Midterm Exam

- Midterm exam Thursday Oct 29 during lecture slot, 2-3:20pm
- Use previous exams on course web site to study
- No homework project for next Friday!
- Midterm tutorial with Jason:
  - Mon Oct 26, 2-5pm, Atkinson Hall room 4004
  - Walk in/out at any time
- TAs will answer questions about midterm during all lab hours next week
- Exam will be closed book, no calculators. Permitted is one single sided, hand-written index card (3x5 inch). You will need to bring a pen/pencil and a ruler, as well as a few blank sheets of paper.
- More information about midterm on course web site under *Grading and Exams*. 

Today

• Shader programming
  - Vertex shaders
  - Fragment shaders

• Texturing
  - Basics
  - Texture coordinate assignment
Programmable pipeline

- Primitives
- Modeling and viewing transformation
  - Shading
  - Projection
- Rasterization
- Fragment processing
  - Frame-buffer access (z-buffering)
- Image

**Vertex program**
- Executed once for each vertex

**Fragment program**
- Executed once for each fragment
GPU architecture

Pipeline

1. Scene data
2. Modeling and viewing transformation
3. Shading
4. Projection
5. Rasterization
6. Fragment processing
7. Frame-buffer access (z-buffering)
8. Image

GPU Architecture

NVidia NV80 (GeForce 8800 Series)
128 stream processors

GPU architecture

Pipeline

- Scene data
  - Modeling and viewing transformation
    - Shading
  - Projection
  - Rasterization
  - Fragment processing
    - Frame-buffer access (z-buffering)

Image

GPU Architecture

NVidia NV80 (GeForce 8800 Series)

128 stream processors

Vertex programs

**Vertex attributes**
Coordinates in object space, additional vertex attributes

From application

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

To rasterizer

**Vertex program**

Transformed vertices, processed vertex attributes
Types of input data

- **Vertex attributes**
  - Change for each execution of the vertex program
  - Predefined OpenGL attributes (color, position, etc.)
  - User defined attributes

- **Uniform parameters**
  - Normally the same for all vertices
  - OpenGL state variables
  - Application defined parameters
Vertex attributes

- “Data that flows down the pipeline with each vertex”
- Per-vertex data that your application specifies
- E.g., vertex position, color, normal, texture coordinates
- Declared using attribute storage classifier in your shader code
  - Read-only
Vertex attributes

• OpenGL vertex attributes accessible through predefined variables

```cpp
attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
attribute vec4 gl_Color;
```

etc.

• Optional user defined attributes
OpenGL state variables

• Provide access to state of rendering pipeline, which you set through OpenGL calls in application

• Predefined variables

```glsl
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform gl_LightSourceParameters
    gl_LightSource[gl_MaxLights];
```

e tc.

• Declared using `uniform` storage classifier
  - Read-only
Uniform parameters

- Parameters that are set by the application
- Should not change frequently
  - Not on a per-vertex basis!
- Will be the same for each vertex until application changes it again
- Declared using `uniform` storage classifier
  - Read-only
Uniform parameters

• To access, use `glGetUniformLocation`, `glUniform*` in application

• Example

  - In shader declare
    ```
    uniform float a;
    ```
  - In application, set a using
    ```
    GLuint p;
    //... initialize program p
    int i=glGetUniformLocation(p,"a");
    glUniform1f(i, 1.f);
    ```
Output variables

• Required output: homogeneous vertex coordinates
  vec4 gl_Position

• **varying** outputs
  - Mechanism to send data to the fragment shader
  - Will be interpolated during rasterization
  - Interpolated values accessible in fragment shader (using same variable name)

• Predefined **varying** outputs
  varying vec4 gl_FrontColor;
  varying vec4 gl_TexCoord[ ];
  etc.

• **User defined** **varying** outputs
Output variables

Note

• Any predefined output variable that you do not write will assume the value of the current OpenGL state

• E.g., your vertex shader does not write varying vec4 gl_TexCoord[ ]
  - Your fragment shader may still read it
  - The value will be the current OpenGL state
“Hello world” vertex program

• main() function is executed for every vertex

• Use predefined variables

```c
void main()
{
    gl_Position = // required output
    gl_ProjectionMatrix * // predefined uniform
    gl_ModelViewMatrix * // predefined uniform
    gl_Vertex;          // predefined attribute
}
```

• Alternatively, use

```c
gl_ModelViewProjectionMatrix or ftransform()
```
Vertex programs

Current limitations

• Cannot write data to memory accessible by application

• Cannot pass data between vertices
  - Each vertex is independent

• For each incoming vertex there is one outgoing vertex
  - Cannot generate new geometry
Examples

- Character skinning
- Particle systems
- Water
Today

- **Shader programming**
  - Vertex shaders
  - Fragment shaders

- **Texturing**
  - Basics
  - Texture coordinate assignment
Fragment programs

fragment data
Interpolated vertex attributes, additional fragment attributes

from rasterizer

uniform parameters
OpenGL state, application specified parameters

fragment program

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 don’t write these variables
“Hello world” fragment program

- `main()` function is executed for every fragment
- Use predefined variables
- Draws everything 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
  - Etc.

- Most often, vertex and fragment shader work together to achieve desired effect
Fragment programs

Current limitations

• Cannot read frame buffer (e.g., color, depth)

• Can only write to frame buffer pixel that corresponds to fragment being processed
  - No random write access to frame buffer

• Number of *varying* variables passed from vertex to fragment shader is limited

• Number of application-defined uniform parameters is limited
Demo

- NVIDIA Tidepool
Summary

• Shader programs replace part of the rendering pipeline

• Written in special shading language (GLSL in OpenGL)

• Sequence of OpenGL calls to compile/activate shaders

• Two types of shaders
  - Vertex shaders
  - Fragment shaders
GLSL main features

• Similar to C

• 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

OpenGL® Shading Language (GLSL) Quick Reference Guide

Describes GLSL version 1.10, as included in OpenGL 3.0, and specified by "The OpenGL Shading Language", version 1.5.9. Section and page numbers refer to that version of the spec.

DATA TYPES (4.1 p.10)

float, vec2, vec3, vec4
int, vec2, vec3, vec4
bool, bvec2, bvec3, bvec4
mat2, mat3, mat4
vec4

data1D, sampler2D, sampler3D

DATA TYPE QUALIFIERS (4.3 p.22)

const: value initialized on entry, and copied on return (default)

uniform: input to vertex and fragment shader from OpenGL or application (READ-ONLY)

attribute: input parameter to geometry shader from OpenGL or application (READ-ONLY), then input to fragment shader (READ-ONLY)

varying: output from vertex shader (READ/WRITE), interpolated, then output to fragment shader (READ-ONLY)

vec4

correct: constant-time constant (READ-ONLY)

function parameters:

in: value initialized on entry, not copied on return (default)

out: copied on return, but not initialized

inout: value initialized on entry, and copied on return

correct: constant-time constant (READ-ONLY)

VECTOR COMPONENTS (5.5 p.30)

component names may not be mixed across sets

x, y, z, a

BUILTIN FUNCTIONS (5.10 p.20)

GLSL version declaration and extensions protocol

extension (oneline name) aliased public enable disable extension (oneline name) aliased public enable disable

http://www.opengl.org/sdk/libs/OpenSceneGraph/gls3l_quirkref.pdf
GLSL quick reference

VERTEX SHADER VARIABLES

Special Output Variables (7.1 p42) access=RW

float gl_FragCoord;  // vertex coordinates
float gl_TexCoord[4];  // vertex texture coordinates

Attribute Inputs (7.3 p44) access=RO

attribute vec3 gl_Vertex;  // vertex position
attribute vec2 gl_Normal;  // vertex normal
attribute vec2 gl_Tangent;  // vertex tangent
attribute vec4 gl_POSITION;  // vertex position
attribute vec4 gl_NORMAL;  // vertex normal
attribute vec4 gl_TANGENT;  // vertex tangent
attribute vec4 gl_TEXCOORD0;  // vertex texture coordinate
attribute vec4 gl_TEXCOORD1;  // vertex texture coordinate
attribute vec4 gl_TEXCOORD2;  // vertex texture coordinate
attribute vec4 gl_TEXCOORD3;  // vertex texture coordinate

Varying Outputs (7.6 p48) access=RW

varying vec3 gl_FrontColor;  // fragment color
varying vec4 gl_Tangent;  // tangent
varying vec4 gl_Normal;  // normal
varying vec4 gl_TexCoord[4];  // texture coordinate

Built-in Uniforms (7.5 p45) access=RO

uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_TextureMatrix[4];
uniform mat4 gl_TextureMatrixInverse[4];
uniform mat4 gl_ModelViewInverseTranspose;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_TextureMatrixInverseTranspose[4];
uniform mat4 gl_NormalMatrix;
uniform float gl_PointSize;

struct gl_DepthRangeParameters {
  float near;
  float far;
};

struct gl_DepthRange {
  float depthRange;
};

GLSL quick reference

OPEN ScenesGraph Preset Uniforms

let csg_FrameNumber;  // frame number
let csg_FrameTime;  // frame time
let csg_DeltaFrameTime;  // delta frame time
let csg_ViewportTexture;
let csg_ViewportTransform;
let csg_ViewportMatrix;

struct gl_LightModelParameters {
  vec4 ambient;
}

struct gl_LightModelParameters gl_LightModel;

struct gl_LightProducts {
  vec4 color;
}

struct gl_LightProducts gl_LightProducts;

struct gl_MaterialParameters {
  vec4 emission;
}

struct gl_MaterialParameters gl_Material;

struct gl_PointParameters {
  float size;
  float depthBias;
  float depthMask;
  float alphaToCoverage;
  float distanceConstantAttenuation;
  float distanceLinearAttenuation;
  float distanceQuadraticAttenuation;
}

struct gl_RippleParameters gl_Ripple;

struct gl_LightSourceParameters {
  vec4 color;
  vec4 diffuse;
  vec4 specular;
  vec4 position;
  vec4 halfVector;
  vec4 spotDirection;
  vec4 spotExponent;
  vec4 spotCutoff;
  float constantAttenuation;
  float linearAttenuation;
  float quadraticAttenuation;
}

GLSL quick reference

http://www.opengl.org/sdk/libs/OpenSceneGraph/gls1_quickref.pdf
Tutorials and documentation

- OpenGL and GLSL specifications
  http://www.opengl.org/documentation/specs/

- GLSL tutorials
  http://www.lighthouse3d.com/opengl/gls1/
  http://www.clockworkcoders.com/oglsl/tutorials.html

- OpenGL Programming Guide (red book)
- OpenGL Shading Language (orange book)
Shader development tools

- ATI/AMD render monkey (OpenGL and DirectX shaders)

- NVidia FX composer (DirectX shaders only)
  http://developer.nvidia.com/object/fx_composer_home.html
Today

- Shader programming
  - Vertex shaders
  - Fragment shaders
- Texturing
  - Basics
  - Texture coordinate assignment
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
• Ideas?
Texture mapping

- Glue textures (images) onto surfaces
- Same triangles, much more interesting appearance
Textures

- Photographs
- Paint directly on surfaces using modeling program
- Stored as image files

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 \((u,v)=(0,0)\)
  - Top right corner is \((u,v)=(1,1)\)
Texture mapping

- Store texture coordinates at each triangle vertex

Triangle (in any space before projection)
Texture mapping

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

\[(u,v) = (0.65,0.75)\]
\[(u,v) = (0.6,0.4)\]
\[(u,v) = (0.4,0.45)\]

Triangle (in any space before projection)
Texture mapping

- Each point on triangle has corresponding point in texture

Triangle (in any space before projection)

Texture space
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 on 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!
Perspectively correct interpolation

- Point in image space with barycentric coordinates $\alpha, \beta, \gamma$

- Triangle vertices with homogeneous coordinates $a_w, b_w, c_w$

1. $\frac{1}{w} = \alpha \frac{1}{a_w} + \beta \frac{1}{b_w} + \gamma \frac{1}{c_w}$

2. $\frac{u}{w} = \alpha \frac{a_u}{a_w} + \beta \frac{b_u}{b_w} + \gamma \frac{c_u}{c_w}$

3. $u = \frac{u}{w} / \frac{1}{w}$

- 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 color of pixel?
Nearest-neighbor interpolation

- Use color of closest texel

- Simple, but low quality
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

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

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

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

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* or *wrapping* rules for out-of-range coordinates
Tiling

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

Texture Space
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
```

![Diagram of Mirroring](image)

- Points marked: (0,0) and (1,1)
Today

- Shader programming
  - Vertex shaders
  - Fragment shaders
- Texturing
  - Basics
  - Texture coordinate assignment
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 techniques to unfold surface onto plane
- Preserve area, angle
- Sophisticated math

[Piponi2000]
Next Lecture

- Antialiasing
- Advanced Texture Mapping