SIGGRAPH 2008

OpenGL: What’s Coming Down the Graphics Pipeline

Dave Shreiner
ARM
shreiner@siggraph.org
Syllabus

- Introduction [Shreiner]
- Rendering Fundamentals [Shreiner]
- Shader Overview [Licea-Kane]
- Fundamental Techniques [Hart]
- Applications [Angel]

What Is OpenGL, and What Can It Do for Me?

- OpenGL is a computer graphics rendering API
  - Generate high-quality color images by rendering with geometric and image primitives
  - Create interactive applications with 3D graphics
  - OpenGL is
    - operating system independent
    - window system independent
OpenGL and Its Related APIs

Graphic Pipeline Varieties

- **Fixed-function version**
  - order of operations is fixed
  - can only modify parameters and disable operations
  - limited to what’s implemented in the pipeline

- **Programmable version**
  - interesting parts of pipeline are under your control
  - write shaders to implement those operations
  - boring stuff is still “hard coded”
  - rasterization & fragment testing
Course Ground-rules

- A new version of OpenGL is coming
- Emphasize the new way to program with OpenGL
  - we won’t discuss the fixed-function pipeline
- Updated notes and demo programs available at:
  
  
  • there are some things we can’t talk about yet

The Graphics Pipeline

- *Transformation stage* converts 3D models into pixel locations
- *Rasterization stage* fills the associated pixels
(Perhaps) The Simplest Vertex Shader

```cpp
attribute vec4 vertex;

void main()
{
    gl_Position = vertex;
}
```

The Simplest Fragment Shader

```cpp
void main()
{
    vec4 blue = vec4(0, 0, 1, 1);
    gl_FragColor = blue;
}
```
Representing Geometry

- We represent geometric primitives by their vertices

\[ \vec{v} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \]

- Vertices are specified as homogenous coordinates
  - 4-tuple of reals
  - most “vertex data” are homogenous coordinates
    - makes the math easier

Storing Vertex Attributes

- Vertex arrays are very flexible
  - store data contiguously as an array, or

```c
glVertexAttribPointer( vIndex, 3, GL_FLOAT, GL_FALSE, 0, v );
glVertexAttribPointer( cIndex, 4, GL_UNSIGNED_BYTE, GL_TRUE, 0, c );
glVertexAttribPointer( tcIndex, 2, GL_FLOAT, GL_FALSE, 0, tc );
```
Storing Vertex Attributes (cont’d)

- As “offsets” into a contiguous array of structures

```c
struct VertexData {
    GLfloat tc[2];
    GLubyte c[4];
    GLfloat v[3];
};
VertexData verts;
```

```
glVertexAttribPointer( vIndex, 3, GL_FLOAT, GL_FALSE, sizeof(VertexData), verts[0].v );
glVertexAttribPointer( cIndex, 4, GL_UNSIGNED_BYTE, GL_TRUE, sizeof(VertexData), verts[0].c );
glVertexAttribPointer( tcIndex, 2, GL_FLOAT, GL_FALSE, sizeof(VertexData), verts[0].tc );
```

“Turning on” Vertex Arrays

- Need to let OpenGL ES know which vertex arrays you’re going to use

```c
glEnableVertexAttribArray( vIndex );
glEnableVertexAttribArray( cIndex );
glEnableVertexAttribArray( tcIndex );
```
OpenGL’s Geometric Primitives

- All primitives are specified by vertices

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUAD_STRIP
GL_QUADS
GL_POLYGON

Drawing Geometric Primitives

- For contiguous groups of vertices

```c
glDrawArrays( GL_TRIANGLES, 0, n );
```

0 1 2 3 4 5
V V V V V
C C C C C
tc tc tc tc tc

0 1 2 3 4
...
Drawing Geometric Primitives

• For indexed groups of vertices

```c
glDrawElements( GL_TRIANGLES, 0, n, indices );
```

Compiling Shaders
Creating a Shader Program

- Similar to compiling a “C” program
  - compile, and link
    - OpenGL ES supports both online and offline compilation
- Multi-step process
  1. create and compile shader objects
  2. attach shader objects to program
  3. link objects into executable program

Shader Compilation (Part 1)

- Create and compile a Shader (with online compilation)
  ```c
  GLuint shader = glCreateShader( shaderType );
  const char* str = "void main() {...}";
  glShaderSource( shader, 1, &str, NULL );
  glCompileShader( shader );
  ```
- `shaderType` is either
  - GL_VERTEX_SHADER
  - GL_FRAGMENT_SHADER
Shader Compilation (Part 2)

- Checking to see if the shader compiled (online compilation)

```cpp
GLint compiled;
glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );
if ( !compiled ) {
    GLint len;
glGetShaderiv( shader, GL_INFO_LOG_LENGTH, &len );
    std::string msgs( ' ', len );
glGetShaderInfoLog( shader, len, &len, &msgs[0] );
    std::cerr << msgs << std::endl;
    throw shader_compile_error;
}
```

Shader Program Linking (Part 1)

- Create an empty program object

```cpp
GLuint program = glCreateProgram();
```

- Associate shader objects with program

```cpp
glAttachShader( program, vertexShader );
glAttachShader( program, fragmentShader );
```

- Link program

```cpp
glLinkProgram( program );
```
Shader Program Linking (Part 2)

• Making sure it worked

    GLint linked;
    glGetProgramiv( program, GL_LINK_STATUS, &linked );
    if ( !linked ) {
        GLint len;
        glGetProgramiv( program, GL_INFO_LOG_LENGTH, &len );
        std::string msgs( ' ', len );
        glGetProgramInfoLog( program, len, &len, &msgs[0] );
        std::cerr << msgs << std::endl;
        throw shader_link_error;
    }

Using Shaders in an Application

• Need to turn on the appropriate shader

    glUseProgram( program );
**Associating Shader Variables and Data**

- Need to associate a shader variable with an OpenGL data source
  - vertex shader attributes → app vertex attributes
  - shader uniforms → app provided uniform values
- OpenGL relates shader variables to indices for the app to set
- Two methods for determining variable/index association
  - specify association before program linkage
  - query association after program linkage

**Determining Locations After Linking**

- Assumes you already know the variables' name

```c
GLint idx =
    glGetAttribLocation( program, "name" );
```

```c
GLint idx =
    glGetUniformLocation( program, "name" );
```
Initializing Uniform Variable Values

- Uniform Variables

glUniform4f(index, x, y, z, w);

GLboolean transpose = GL_TRUE; // Since we're C programmers
GLfloat mat[3][4][4] = { ... };

glUniformMatrix4fv(index, 3, transpose, mat);
Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)

Transformations ... Magical Mathematics

- Transformations take us from one “space” to another
  - All of our transforms are 4×4 matrices
Camera Analogy and Transformations

- Projection transformations
  - adjust the lens of the camera
- Viewing transformations
  - tripod—define position and orientation of the viewing volume in the world
- Modeling transformations
  - moving the model
- Viewport transformations
  - enlarge or reduce the physical photograph

3D Transformations

- A vertex is transformed by 4 x 4 matrices
  - all affine operations are matrix multiplications
  - all matrices are stored column-major in OpenGL
    - this is opposite of what "C" programmers expect
  - matrices are always post-multiplied
  - product of matrix and vector is $\mathbf{M}\mathbf{v}$
Specifying What You Can See

• Set up a *viewing frustum* to specify how much of the world we can see
• Done in two steps
  – specify the size of the frustum (*projection transform*)
  – specify its location in space (*model-view transform*)
• Anything outside of the viewing frustum is *clipped*
  – primitive is either modified or discarded (if entirely outside frustum)

Specifying What You Can See (cont’d)

• OpenGL projection model uses *eye coordinates*
  – the “eye” is located at the origin
  – looking down the –z axis
• Projection matrices use a six-plane model:
  – near (image) plane
  – far (infinite) plane
    • both are distances from the eye (positive values)
  – enclosing planes
    • top & bottom
    • left & right
Specifying What You Can See (cont’d)

Orthographic View

Perspective View

\[ O = \begin{pmatrix} \frac{2}{z/2} & 0 & 0 & \frac{z/2}{z/2} \\ 0 & \frac{1}{f} & 0 & \frac{2}{f-z} \\ 0 & 0 & \frac{z}{f-z} & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

\[ P = \begin{pmatrix} \frac{2n}{z/2} & 0 & 0 & \frac{z/2}{z/2} \\ 0 & \frac{2}{f} & 0 & \frac{2}{f-z} \\ 0 & 0 & \frac{1}{f-z} & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \]

Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To “fly through” a scene
  - change viewing transformation and redraw scene

\[ \text{LookAt( } \text{eye}_x, \text{eye}_y, \text{eye}_z, \text{look}_x, \text{look}_y, \text{look}_z, \text{up}_x, \text{up}_y, \text{up}_z \text{ )} \]

- up vector determines unique orientation
- careful of degenerate positions
Creating the LookAt Matrix

\[ \hat{n} = \frac{\text{look-eye}}{\|\text{look-eye}\|} \]
\[ \hat{u} = \frac{\hat{n} \times \text{up}}{\|\hat{n} \times \text{up}\|} \]
\[ \hat{v} = \hat{u} \times \hat{n} \]

\[
\begin{pmatrix}
u_x & u_y & u_z & 0 \\
v_x & v_y & v_z & 0 \\
-n_x & -n_y & -n_z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

Translation

- Move the origin to a new location

\[
T(t_x, t_y, t_z) = \begin{pmatrix}
1 & 0 & 0 & t_x \\
0 & 1 & 0 & t_y \\
0 & 0 & 1 & t_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Scale

- Stretch, mirror or decimate a coordinate direction

\[ S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

Note, there’s a translation applied here to make things easier to see.

Rotation

- Rotate coordinate system about an axis in space

Note, there’s a translation applied here to make things easier to see.
Rotation (cont’d)

\[ \vec{v} = (x, y, z) \]
\[ \vec{u} = \frac{\vec{v}}{|\vec{v}|} = (x', y', z') \]

\[
M = \vec{u}' \vec{u} + \cos(\theta)(I - \vec{u}' \vec{u}) + \sin(\theta)S
\]

\[
S = \begin{pmatrix}
0 & -z' & y' \\
z' & 0 & -x' \\
-y' & x' & 0
\end{pmatrix}
\]

\[
R_+(\theta) = \begin{pmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & -1 & 0
\end{pmatrix}
\]
Double Buffering

Animation Using Double Buffering

1. Request a double buffered color buffer
   ```
   glutInitDisplayMode ( GLUT_RGB | GLUT_DOUBLE );
   ```
2. Clear color buffer
   ```
   glClear ( GL_COLOR_BUFFER_BIT );
   ```
3. Render scene
4. Request swap of front and back buffers
   ```
   glutSwapBuffers();
   ```
• Repeat steps 2 - 4 for animation
  – Use a glutIdleFunc() callback
Depth Buffering and Hidden Surface Removal

Depth Buffering Using OpenGL

1. Request a depth buffer
   ```
   glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
   ```
2. Enable depth buffering
   ```
   glEnable(GL_DEPTH_TEST);
   ```
3. Clear color and depth buffers
   ```
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   ```
4. Render scene
5. Swap color buffers
OpenGL Shading Language Overview

Bill Licea-Kane
AMD

OpenGL Shading Language

• Shading Language Details
• Trivial Examples
Shading Language Details

- The OpenGL Shading Language 1.20.08
- The OpenGL ES Shading Language 1.0.14
- DRAFT! The OpenGL Shading Language 1.3.tbd

Preprocessor

```
// Comment
#define /* Comment */

#if
#elif
#else
#elifdef
#ifndef
#else
#elif
#elifdef
#elseif
#else
#error
#pragma
```
Preprocessor

```shading_language_1.10
#version 110 // Shading Language 1.10 (IMPLICIT)
#version 120 // Shading Language 1.20
#version 130 // Shading Language 1.30 (DRAFT)

#extension: NAME : require|enable|warn|disable

#line LINE FILE
__LINE__
__FILE__
__VERSION__
```

Types

```shading_language_1.10
void

// Scalar
float int bool

// Vector
vec2 vec3 vec4
tvec2 tvec3 tvec4
bvec2 bvec3 bvec4

// Matrix
mat2 mat3 mat4 mat2x2

// Sampler
sampler1D sampler2D sampler3D samplerCube
sampler1DShadow sampler2DShadow
```
Containers

```c
// no qualifiers
// no bitfields
// no forward references
// no in-place definitions
// no anonymous structures

// 1D arrays (First class starting in version 1.20)
```

Scope

```c
// (Outside Global)
// Built-in functions

// Global
// User-defined functions (Can hide Built-in)
// Shared name space
// Shared globals must be same type

// Local
// RESTRICTION - No function prototypes
```
Storage Qualifiers

default
const

// global qualifiers
attribute
uniform
varying
centroid varying

// invariant qualifier
Invariant

// parameter qualifiers
in out inout

Operators

() // grouping
[] // array and component
() // constructor
. // field select and swizzle
++ -- // postfix
++ -- // prefix
+ - ! // prefix
Operators

+ - * /  // binary
< <= > >=  // relational
== !=  // equality
&& ^^ ||  // logical
?:  // selection
= += -= *= /=  // assignment

Integer operators

~  // prefix
%  // binary
<< >> & ^ |  // bitwise
%= <<= >>= &= ^= |=  // assignment
Constructors

// Scalar
float() int() bool()

// Vector
vec2() vec3() vec4()
ivec2() ivec3() ivec4()
bvec2() bvec3() bvec4()

// Matrix
mat2() mat3() mat4() matCxC()

// Struct
// Array

Components

// Vector
.xyzw .rgba .stpq [i]

// Matrix
[i] [i][j]
Flow Control

// expression ? TrueExpression : FalseExpression
// if, if-else
// for, while, do-while
// return, break, continue
// discard (fragment only)

Vector Matrix Operations

mat4 m4, n4;
vec4 v4;

vec4 first = m4 * v4; // matrix * vector
vec4 second = v4 * n4; // vector * matrix
mat4 third = m4 * n4; // matrix * matrix
Functions

// Parameter qualifiers
in out inout
count in
// Functions are call by value, copy in, copy out
// NOT exactly like C++
/
// Examples
vec3 function( count in vec3 N, count in vec3 L );
void f( inout float X, count in float Y );

Special Variables

// Vertex
vec4 gl_Position; // must be written to
vec4 gl_ClipVertex; // may be written to
count gl_PointSize; // may be written to

// Fragment
vec4 gl_FragCoord; // may be read from
bool gl_FrontFacing; // may be read from
vec4 gl_FragColor; // may be written to
vec4 gl_FragData[i]; // may be written to
count gl_FragDepth; // may be written to
Built-in attributes

This page intentionally blank
(there aren’t any in built-in attributes in GLSL 1.30)

Built-in varying

This page intentionally blank
(there also aren’t any in built-in varyings in GLSL 1.30)
Built-in uniforms

This page intentionally blank

(and guess what, there also aren't any in built-in uniforms in GLSL 1.30, either)

Built-in Functions

// angles and trigonometry
// exponential
// common
// interpolations
// geometric
// vector relational
// texture
// shadow
// noise
// vertex
vec4 ftransform(void);

// fragment
genType dFdx(genType P);
genType dFdy(genType P);
genType fwidth(genType P);
Really Simple Shaders

// Vertex Shader

//
// #version 120 // Shading Language 1.20

void main( void )
{
    gl_Position = vec4( 0.0 );
}

// Fragment Shader

//
// #version 120 // Shading Language 1.20

void main( void ) {}
Smallest OpenGL Shaders

// Vertex Shader
//
// #version 130 // Shading Language 1.30
void main() {  
  gl_Position = vec4( 0.0 );
}

// Fragment Shader
//
// #version 130 // Shading Language 1.30
void main() { }

Small OpenGL Shaders

// Vertex Shader
//
// #version 120 // Shading Language 1.20
uniform mat4 matMVP;
attribute vec4 mPosition;
attribute vec4 mColor;
varying vec4 fColor;
void main() {  
  gl_Position = matMVP * mPosition;
  fColor = mColor;
}

// Fragment Shader
//
// #version 120 // Shading Language 1.20
varying vec4 fColor;
void main() {  
  gl_FragData[0] = fColor;
}
Small OpenGL Shaders

// Vertex Shader
//
// #version 130 // Shading Language 1.30
uniform mat4 matMVP;
in vec4 mPosition;
in vec4 mColor;

void main( void )
{
    gl_Position = matMVP * mPosition;
    // fColor = mColor;
}

// Fragment Shader
//
// #version 130 // Shading Language 1.30
in vec4 mColor;

void main( void )
{
    gl_FragData[0] = mColor;
}

OpenGL Shading Language

• Acknowledgements
  – John Kessenich (Intel)
  – David Baldwin
  – Randi J. Rost (Intel)
  – Robert Simpson (AMD)
  – Benj Lipchack (AMD)
  – …and ARB Contributors
Lighting and Materials

Evan Hart
NVIDIA

Illumination with Shaders

• DIY Lighting
  – No built-in illumination support in modern API
• Back to the basics
  – Fundamental graphics algorithms are key
• Complete flexibility
  – Any lighting model desired
Illumination Components

- View dependent component
  - Specular
- View independent components
  - Diffuse
  - Ambient
  - Emissive

Mathematics of Illumination

- Sum of contributions from all lights
  - Diffuse
  - Specular
- Auxilliary Illumination
  - Ambient – bounce lighting
  - Emission - glow
Diffuse Illumination Component

- Lambertian model
  - Works well for many common materials
- Intensity derived from angle of incidence

Diffuse Illumination Diagram
Diffuse Illumination Code

```glsl
// Compute light direction
vec3 light_dir = normalize( light_pos - pos);

// Compute the lighting
float intensity = dot( normal, light_dir);
intensity = clamp( intensity, 0.0, 1.0);
```

Specular Illumination Component

- Blinn-Phong model
  - Simple and efficient
  - Good for plastic
  - OK for some smooth metal
- Intensity derived from view vector, normal vector, and light vector
Specular Illumination Diagram

Specular Illumination Code

// Compute the view vector
vec3 view_dir = normalize(-pos);

// Compute the half-angle vector
vec3 half = normalize(view_dir + light_dir);

// Compute the specular intensity
float spec = dot(half, normal);
spec = clamp(spec, 0.0, 1.0);
spec = pow(spec, shininess);
Coordinate Frames

- Consistent coordinate frame required for scene composition
  - Allows objects to appear in the proper place

- Common Solutions
  - World coordinates
  - Eye Coordinates
  - Surface-local Coordinates

Worldspace Coordinate Frame

- Most intuitive
  - Easy to reason about
  - Easy to debug
**Eyespace Coordinate Frame**

- Centers coordinate frame at eye
  - Viewer position becomes (0,0,0)
- View direction is typically (0,0,±1)
  - Fixed-function OpenGL used (0,0,-1)
- Somewhat more efficient than worldspace
  - Constant direction and eye position

**Surface-local Coordinate Frame**

- Typically not used for actual lighting computations
  - Used to convert from 2D coordinate to 3D
- Most common is ‘Tangent Space’
  - Defined by two vectors tangent to the surface and the normal
  - Normal typically specifies the Z direction
Tangent Space

Computational Frequency

- Per-polygon
  - Fairly difficult, mostly for diagrammatic purposes
- Per-vertex
  - Simple, fairly low-quality
- Per-fragment
  - Moderate difficulty, high-quality
- Hybrid
  - Different frequencies for different components
Interpolation

- Process of converting vertex values to fragment values
- Performed on all variables declared varying
- Interpolation mode and shaders determine shading frequency

Per-polygon shading

- Compute lighting in vertex shader
- Declare illumination components as varying
  - Often wish to defer summation
  - Allows per-pixel material application for material colors
Material Properties

- Shades of gray are boring
- Materials provide the proper look
- Can include all factors feeding into lighting
  - Colors
  - Surface roughness

Material Colors

- Can provide separate colors for all illumination components
  - Specular, ambient, diffuse, etc.
- Relative combinations emulate different physical materials
  - Metals: diffuse_material == specular_material
  - Plastics: specular_material == white
Shininess / Roughness

• Specular power \( (k) \)
  – Provides tightness on specular highlight
  – Often interpreted as \( k = 1 / \text{roughness} \)

Texturing

• Applying an image to the object surface
  – Most often 2D
• Thought of as a set of varying material properties
  – All properties mentioned so far may be provided via textures
Texture Coordinates

- Shading language functions
  - `texture2D(sampler2D map, vec2 p)`
    - Returns `vec4` from point `p` in map
  - `texture2DProj(sampler2D map, vec4 p)`
    - `p' = p.xy / p.w`
    - Returns `vec4` from point `p'` in map
Loading and Configuring Textures

- Bind texture object
- Load base image
- Load mipmap images [optional]
- Specify texture object parameters
  - Filter state
  - Wrap state

Specifying a Texture Image

- `glTexImage2D` arguments
  - Dimensions: width and height
  - Internal format: preferred HW format
    - RGBA8, RGB5_A1, LUMINANCE8
  - Format: format of input data
    - RGB, RGBA, etc
  - Type: data type of input data
    - UCHAR, FLOAT, etc
Mipmaps

- Smaller versions of base image
- Allow for better filtering
  - Reduce aliasing
- Smaller levels are ½ size in each dimension
  - 128x128 – 64x64 – 32x32 …
Filtering

- Specified through glTexParameteri
- Controls the manner of fetching a texel
  - GL_NEAREST – point sampling (worst)
  - GL_LINEAR – blend between 4 nearest texels
  - GL_LINEAR_MIPMAP_NEAREST – Select one mipmap and blend the 4 nearest texels
  - GL_LINEAR_MIPMAP_LINEAR – Select two mipmap and blend the 4 nearest texels from each
Wrap State

- Specified through glTexParameter
- Controls boundary behavior for texture filtering
  - GL_CLAMP_TO_EDGE
  - GL_REPEAT
  - GL_MIRRORED_REPEAT

Clamp Texture Wrapping
Repeat Texture Wrapping

Mirror Texture Wrapping
Applying the Texture to the Shader

// Declare the sampler
uniform sampler2D diffuse_mat;

// Apply the material color
vec3 diffuse = intensity * texture2D(diffuse_mat, coord).rgb;

Thanks

- Fellow presenters
- NVIDIA DevTech Team
- NVIDIA OpenGL Driver Team
Application Examples

Ed Angel
University of New Mexico

Shader Examples

• Vertex Shaders
  – Moving vertices: height fields
  – Per vertex lighting: height fields
  – Per vertex lighting: cartoon shading

• Fragment Shaders
  – Per vertex vs per fragment lighting: cartoon shader
  – Samplers: reflection Map
  – Bump mapping
Height Fields

• A height field is a function $y = f(x, z)$ where the $y$ value represents a quantity such as the height above a point in the $x$-$z$ plane.

• Heights fields are usually rendered by sampling the function to form a rectangular mesh of triangles or rectangles from the samples $y_{ij} = f(x_i, y_j)$

Displaying a Height Field

• Defining a rectangular mesh
  $$\text{for}(i=0;i<N;i++) \text{ for}(j=0;j<N;j++) \text{ data}[i][j]=f( i, j, \text{time});$$

• Displaying a mesh
  $$\text{glBegin(GL\_LINE\_LOOP);}$$
  $$\text{glVertex3f((float)i/N, data[i][j], (float)j/N);}$$
  $$\text{glVertex3f((float)i/N, data[i][j], (float)(j+1)/N);}$$
  $$\text{glVertex3f((float)(i+1)/N, data[i][j], (float)(j+1)/N);}$$
  $$\text{glVertex3f((float)(i+1)/N, data[i][j], (float)(j/N);}$$
  $$\text{glEnd();}$$
Time varying vertex shader

uniform float time; /* in milliseconds */

void main()
{
  vec4 t = gl_Vertex;
  t.y = 0.1*sin(0.001*time + 5.0*gl_Vertex.x)*
       sin(0.001*time+5.0*gl_Vertex.z);
  gl_Position = gl_ModelViewProjectionMatrix * t;
  gl_FrontColor = gl_Color;
}

Mesh Display
Adding Lighting

• Solid Mesh:
  
  `glBegin(GL_POLYGON);`
  
• We must add lighting
  
• Must do per vertex lighting in shader if we use a vertex shader for time-varying mesh

Mesh Shader

```glsl
uniform float time;
void main()
{
  vec4 t = gl_Vertex;
  t.y = 0.1*sin(0.001*time+5.0*gl_Vertex.x) *sin(0.001*time+5.0*gl_Vertex.z);
  gl_Position = gl_ModelViewProjectionMatrix * t;
  
  vec4 ambient;
  vec4 diffuse;
  vec4 specular;
  vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;
  vec4 eyeLightPos = gl_LightSource[0].position;
```

Mesh Shader (cont)

```glsl
vec3 N = normalize(gl_NormalMatrix * gl_Normal);
vec3 L = normalize(eyeLightPos.xyz - eyePosition.xyz);
vec3 E = -normalize(eyePosition.xyz);
vec3 H = normalize(L + E);
float Kd = max(dot(L, N), 0.0);
float Ks = pow(max(dot(N, H), 0.0), gl_FrontMaterial.shininess);
diffuse = Kd*gl_FrontLightProduct[0].diffuse;
specular = Ks*gl_FrontLightProduct[0].specular;
gl_FrontColor = ambient+diffuse+specular;
```
Cartoon Shader

• This vertex shader uses only two colors but the color used is based on the orientation of the surface with respect to the light source
• Normal vector provided by the application through `glNormal` function
• A third color (black) is used for a silhouette edge

Cartoon Shader Code

```c
void main()
{
    const vec4 yellow = vec4(1.0, 1.0, 0.0, 1.0);
    const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;
    vec4 eyeLightPos = gl_LightSource[0].position;
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec3 L = normalize(eyeLightPos.xyz - eyePosition.xyz);
    float Kd = max(dot(L, N), 0.0);
    if(Kd > 0.6) gl_FrontColor = yellow;
    else gl_FrontColor = red;
}
```
Adding a Silhouette Edge

const vec4 black = vec4(0.0, 0.0, 0.0, 1.0);
vec3 E = -normalize(eyePosition.xyz);
if(abs(dot(E,N))<0.25) gl_FrontColor = black;

Smoothing

• We can get rid of some of the jaggedness using the mix function in the shader

   gl_FrontColor = mix(yellow, red, Kd);
Fragment Shader Examples

- Per fragment lighting: Cartoon shader
- Texture Mapping: Reflection Map
- Bump Mapping

Per Fragment Cartoon Vertex Shader

```glsl
varying vec3 N;
varying vec3 L;
varying vec3 E;

void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    vec4 eyePosition = gl_ModelViewMatrix * gl_Vertex;
    vec4 eyeLightPos = gl_LightSource[0].position;

    N = normalize(gl_NormalMatrix * gl_Normal);
    L = normalize(eyeLightPos.xyz - eyePosition.xyz);
    E = -normalize(eyePosition.xyz);
}
```
Cartoon Fragment Shader

```glsl
varying vec3 N;
varying vec3 L;
varying vec3 E;

void main()
{
    const vec4 yellow = vec4(1.0, 1.0, 0.0, 1.0);
    const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
    const vec4 black = vec4(0.0, 0.0, 0.0, 1.0);

    float Kd = max(dot(L, N), 0.0);
    gl_FragColor = mix(red, yellow, Kd);
    if(abs(dot(E,N))<0.25) gl_FragColor = black;
}
```

Cartoon Fragment Shader Result

![Tea Pot Image]
Reflection Map

- Specify a cube map in application
- Use `reflect` function in vertex shader to compute view direction
- Apply texture in fragment shader

Reflection Map Vertex Shader

```cpp
varying vec3 R;

void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    vec3 N = normalize(gl_NormalMatrix*gl_Normal);
    vec4 eyePos = gl_ModelViewMatrix*gl_Vertex;

    R = reflect(eyePos.xyz, N);
}
```
Reflection Map Fragment Shader

```glsl
varying vec3 R;
uniform samplerCube texMap;

void main()
{
    vec4 texColor = textureCube(texMap, R);
    gl_FragColor = texColor;
}
```

Reflection mapped teapot
Bump Mapping

- Vary normal in fragment shader so that lighting changes for each fragment
- Application: specify texture maps that describe surface variations
- Vertex Shader: calculate vertex lighting vectors and transform to texture space
- Fragment Shader: calculate normals from texture map and shade each fragment

Bump Map Example
Thanks!

References

On-Line Resources

- [http://www.opengl.org](http://www.opengl.org)
  - start here; up to date specification and lots of sample code
  - online "man pages" for all OpenGL functions
- [http://www.mesa3d.org/](http://www.mesa3d.org/)
  - Brian Paul's Mesa 3D
  - very special thanks to Nate Robins for the OpenGL Tutors
  - source code for tutors available here!
Books

- The OpenGL Shading Language, 2nd Edition
- OpenGL Programming for the X Window System
- OpenGL: A Primer 3rd Edition
- OpenGL Distilled
- OpenGL Programming on Mac OS® X