An Introduction to OpenGL Programming

Ed Angel  University of New Mexico
Dave Shreiner  ARM, Inc.

SIGGRAPH 2013
The 40th International Conference and Exhibition on Computer Graphics and Interactive Techniques
What Is OpenGL?

• OpenGL is a computer graphics rendering *application programming interface*, or API (for short)
  – With it, you can generate high-quality color images by rendering with geometric and image primitives
  – It forms the basis of many interactive applications that include 3D graphics
  – By using OpenGL, the graphics part of your application can be
    • operating system independent
    • window system independent
Course Ground Rules

• We’ll concentrate on the latest versions of OpenGL
• They enforce a new way to program with OpenGL
  – Allows more efficient use of GPU resources
• Modern OpenGL doesn’t support many of the “classic” ways of doing things, such as
  – Fixed-function graphics operations, like vertex lighting and transformations
• All applications must use shaders for their graphics processing
  – we only introduce a subset of OpenGL’s shader capabilities in this course
Evolution of the OpenGL Pipeline
In the Beginning …

• OpenGL 1.0 was released on July 1\textsuperscript{st}, 1994
• Its pipeline was entirely \textit{fixed-function}
  – the only operations available were fixed by the implementation

• The pipeline evolved
  – but remained based on fixed-function operation through OpenGL versions 1.1 through 2.0 (Sept. 2004)
Beginnings of The Programmable Pipeline

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available
An Evolutionary Change

- OpenGL 3.0 introduced the *deprecation model* – the method used to remove features from OpenGL.
- The pipeline remained the same until OpenGL 3.1 (released March 24\textsuperscript{th}, 2009).
- Introduced a change in how OpenGL contexts are used.

<table>
<thead>
<tr>
<th>Context Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>Includes all features (including those marked deprecated) available in the current version of OpenGL</td>
</tr>
<tr>
<td>Forward Compatible</td>
<td>Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)</td>
</tr>
</tbody>
</table>
The Exclusively Programmable Pipeline

• OpenGL 3.1 removed the fixed-function pipeline
  – programs were required to use only shaders

• Additionally, almost all data is GPU-resident
  – all vertex data sent using buffer objects
What We Can’t DO

• Any use of the **fixed function** vertex or fragment operations; shaders are mandatory.
• Use of `glBegin/glEnd` and **Display lists** to define primitives; vertex buffers objects for geometry.
• Use of quad or polygon primitives; **only triangles**.
• Use of most of the built-in attribute and uniform variables in GLSL; **pass them manually to shaders**.
The main application of vertex shaders is to change the vertices of the primitives you already have defined and to setup variables such as lightening that depend of the vertices.

The vertex shader is a one vertex in, one vertex out process, and it can't create more vertices. (Geometry and Tessellation shaders do that on recent versions)
Vertex Shader (GLSL version < 1.2)

- Standard attributes:
  - `gl_Vertex`
  - `gl_Normal`
  - `gl_Color`
  - `gl_MultiTexCoord0`
  - ...

- Built-in constants:
  - `gl_MaxLights`, ...

- Texture maps

- Special variables:
  - `gl_Position` *(shader must write)*
  - ...
  - `gl_FrontColor`
  - `gl_TexCoord[ ]`
  - ...

- Built-in uniforms:
  - `gl_ModelViewMatrix`, ...

- User-defined uniforms

- User-defined varyings
Fragment Shader (GLSL version < 1.2)

- Discarding Pixels
- Anisotropic Shading
- Data Driven Shading
• OpenGL 3.2 (released August 3\textsuperscript{rd}, 2009) added an additional shading stage – geometry shaders
  – modify geometric primitives within the graphics pipeline
Geometry Shader (GLSL version 1.5)

- **Standard attributes**
  - `gl_PositionIn`[`gl_VerticesIn`]
  - `gl_FrontColorIn`[`gl_VerticesIn`]
  - `gl_TexCoordIn`[`gl_VerticesIn`]
  - ...

- **Built-in constants**
  - `gl_VerticesIn`, ...

- **Texture maps**

- **User-defined attributes**

- **Built-in uniforms**
  - `gl_ModelViewMatrix`, ...

- **User-defined uniforms**

- **User-defined varyings**

- **Special variables**
  - `{ gl_Position (shader must write) }
  - ...
  - `{ gl_FrontColor, gl_TexCoord[ ] }
  - ...

- **Standard varyings**

- **Functions**
  - `void EmitVertex();`
  - `void EndPrimitive();`
More Evolution – Context Profiles

• OpenGL 3.2 also introduced context profiles
  – profiles control which features are exposed
    • it's like GL_ARB_compatibility, only not insane 😊
  – currently two types of profiles: core and compatible

<table>
<thead>
<tr>
<th>Context Type</th>
<th>Profile</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>core</td>
<td>All features of the current release</td>
</tr>
<tr>
<td></td>
<td>compatible</td>
<td>All features ever in OpenGL</td>
</tr>
<tr>
<td>Forward Compatible</td>
<td>core</td>
<td>All non-deprecated features</td>
</tr>
<tr>
<td></td>
<td>compatible</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
The Latest Pipelines

• OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders

• Latest version is 4.5
OpenGL ES and WebGL

• OpenGL ES 2.0
  – Designed for embedded and hand-held devices such as cell phones
  – Based on OpenGL 3.1
  – Shader based

• WebGL
  – JavaScript implementation of ES 2.0
  – Runs on most recent browsers
OpenGL Application Development
• Modern OpenGL programs essentially do the following steps:
  – Create shader programs
  – Create buffer objects and load data into them
  – “Connect” data locations with shader variables
  – Render
Application Framework Requirements

- OpenGL applications need a place to render into
  - usually an on-screen window
- Need to communicate with native windowing system
- Each windowing system interface is different
- We use GLUT (more specifically, freeglut - http://freeglut.sourceforge.net)
  - simple, open-source library that works everywhere
  - handles all windowing operations:
    - opening windows
    - input processing
Simplifying Working with OpenGL

• Operating systems deal with library functions differently
  – compiler linkage and runtime libraries may expose different functions
• Additionally, OpenGL has many versions and profiles which expose different sets of functions
  – managing function access is cumbersome, and window-system dependent
• We use another open-source library, GLEW, to hide those details - http://glew.sourceforge.net
Representing Geometric Objects

• Geometric objects are represented using *vertices*
• A vertex is a collection of generic attributes
  – positional coordinates
  – colors
  – texture coordinates
  – any other data associated with that point in space
• Position stored in 4 dimensional homogeneous coordinates
• Vertex data must be stored in vertex buffer objects (VBOs)
• VBOs must be stored in vertex array objects (VAOs)
OpenGL’s Geometric Primitives

- All primitives are specified by vertices

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_LINE_LOOP
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
A First Program
Our First Program

• We’ll render a cube with colors at each vertex

• Our example demonstrates:
  – initializing vertex data
  – organizing data for rendering
  – simple object modeling
    • building up 3D objects from geometric primitives
    • building geometric primitives from vertices
Classic arrays with indices

Now we have to use triangles,
- Vertex attribute arrays with **8 elements** because we have one normal per vertex
- If different normals at the shared vertices it should be used arrays with **24 elements**
Initializing the Cube’s Data

• We’ll build each cube face from individual triangles
• Arrays without indices
• Need to determine how much storage is required
  – (6 faces)(2 triangles/face)(3 vertices/triangle)

\[
\text{const int NumVertices} = 36;
\]

• To simplify communicating with GLSL, we’ll use a \text{vec4} class (implemented in C++) similar to GLSL’s \text{vec4} type
  – we’ll also typedef it to add logical meaning

\[
\text{typedef vec4 point4;}
\]
\[
\text{typedef vec4 color4;}
\]
Initializing the Cube’s Data (cont’d)

• Before we can initialize our VBO, we need to stage the data
• Our cube has two attributes per vertex
  – position
  – color
• We create two arrays to hold the VBO data

  point4 vPositions[NumVertices];
  color4 vColors[NumVertices];
• Vertices of a unit cube centered at origin
  – sides aligned with axes

```
point4 positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```
• We’ll also set up an array of RGBA colors

    color4 colors[8] = {
        color4( 0.0, 0.0, 0.0, 1.0 ), // black
        color4( 1.0, 0.0, 0.0, 1.0 ), // red
        color4( 1.0, 1.0, 0.0, 1.0 ), // yellow
        color4( 0.0, 1.0, 0.0, 1.0 ), // green
        color4( 0.0, 0.0, 1.0, 1.0 ), // blue
        color4( 1.0, 0.0, 1.0, 1.0 ), // magenta
        color4( 1.0, 1.0, 1.0, 1.0 ), // white
        color4( 0.0, 1.0, 1.0, 1.0 )  // cyan
    };
Generating a Cube Face from Vertices

- To simplify generating the geometry, we use a convenience function `quad()`:
  - create two triangles for each face and assigns colors to the vertices

```c
int Index = 0;  // global variable indexing into VBO arrays

void quad( int a, int b, int c, int d )
{
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[b]; vPositions[Index] = positions[b]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[d]; vPositions[Index] = positions[d]; Index++;
}
```
• Generate 12 triangles for the cube
  – 36 vertices with 36 colors

```c
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```
Vertex Array Objects (VAOs)

- VAOs store the data of a geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an object
  - previously, you might have needed to make many calls to make all the data current
• Create a vertex array object

```c
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );
```
Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling
    
    ```
    glBindBuffer( GL_ARRAY_BUFFER, ... )
    ```
  - load data into VBO using
    
    ```
    glBufferData( GL_ARRAY_BUFFER, ... )
    ```
  - bind VAO for use in rendering `glBindVertexArray()`
• Create and initialize a buffer object

```c
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER,
    sizeof(vPositions) + sizeof(vColors),
    NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0,
    sizeof(vPositions), vPositions );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(vPositions),
    sizeof(vColors), vColors );
```
• Application vertex data enters the OpenGL pipeline through the vertex shader
• Need to connect vertex data to shader variables
  – requires knowing the attribute location
• Attribute location can either be queried by calling `glGetVertexAttribLocation()` after the linkage
• Or set by `glBindAttribLocation()` before the linkage
Vertex Array Code

- Associate shader variables with vertex arrays
  - do this after shaders are loaded

```c
GLuint vPosition =
    glGetAttribLocation(program, "vPosition" );
glEnableVertexAttribArray(vPosition );
glVertexAttribPointer(vPosition, 4, GL_FLOAT,
    GL_FALSE, 0,BUFFER_OFFSET(0) );

GLuint vColor =
    glGetAttribLocation(program,"vColor" );
glEnableVertexAttribArray(vColor );
glVertexAttribPointer(vColor, 4, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(sizeof(vPositions)) );
```
Drawing Geometric Primitives

• For contiguous groups of vertices

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

• Usually invoked in display callback
• Initiates vertex shader
Shaders and GLSL
GLSL Data Types

- **Scalar types:** `float`, `int`, `bool`
- **Vector types:** `vec2`, `vec3`, `vec4`  
  `ivec2`, `ivec3`, `ivec4`  
  `bvec2`, `bvec3`, `bvec4`
- **Matrix types:** `mat2`, `mat3`, `mat4`
- **Texture sampling:** `sampler1D`, `sampler2D`, `sampler3D`, `samplerCube`
- **C++ Style Constructors**
  ```cpp
  vec3 a = vec3(1.0, 2.0, 3.0);
  ```
Operators

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

```cpp
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```
Components and Swizzling

- Access vector components using either:
  - [ ] (c-style array indexing)
  - xyzw, rgba or strq (named components)

- For example:
  ```
  vec3 v;
  v[1], v.y, v.g, v.t - all refer to the same element
  ```

- Component swizzling:
  ```
  vec3 a, b;
  a.xy = b.yx;
  ```
• **in, out**
  - Copy vertex attributes and other variable into and out of shaders

    ```
    in vec2 texCoord;
    out vec4 color;
    ```

• **uniform**
  - shader-constant variable from application

    ```
    uniform float time;
    uniform vec4 rotation;
    ```
Functions

• Built in
  – Arithmetic: \( \text{sqrt}, \text{power}, \text{abs} \)
  – Trigonometric: \( \sin, \asin \)
  – Graphical: \( \text{length}, \text{reflect} \)

• User defined
Built-in Variables

• **gl_Position**
  – (required) output position from vertex shader

• **gl_FragCoord**
  – input fragment position

• **gl_FragDepth**
  – input depth value in fragment shader
#version 430

in vec4 vPosition;
in vec4 vColor;

out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
#version 430

in vec4 color;

out vec4 fColor; // fragment’s final color

void main()
{
    fColor = color;
}
Getting Your Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program.
- OpenGL provides the compiler and linker.
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional.

These steps need to be repeated for each type of shader in the shader program.
• We’ve created a routine for this course to make it easier to load your shaders
  – available at course website

      GLuint InitShaders( const char* vFile,
                          const char* fFile );

• InitShaders takes two filenames
  – vFile path to the vertex shader file
  – fFile for the fragment shader file

• Fails if shaders don’t compile, or program doesn’t link
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’ names

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

GLint loc = glGetUniformLocation( program, "name" );
```
• Uniform Variables

```c
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);
```
Finishing the Cube Program

```c
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
    glutInitWindowSize( 512, 512 );
    glutCreateWindow( "Color Cube" );

    glewInit();
    init();

    glutDisplayFunc( display );
    glutKeyboardFunc( keyboard );
    glutMainLoop();

    return 0;
}
```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}

void keyboard( unsigned char key, int x, int y )
{
    switch( key ) {
        case 033: case 'q': case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}
A vertex shader is initiated by each vertex output by `glDrawArrays()`.

A vertex shader must output a position in clip coordinates to the rasterizer.

Basic uses of vertex shaders:
- Transformations
- Lighting
- Moving vertex positions
Transformations
Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)
Transformations take us from one “space” to another
- All of our transforms are $4 \times 4$ 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
• A vertex is transformed by 4×4 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 $\mathbf{M} \mathbf{v}$

\[
\mathbf{M} = \begin{bmatrix}
  m_0 & m_4 & m_8 & m_{12} \\
  m_1 & m_5 & m_9 & m_{13} \\
  m_2 & m_6 & m_{10} & m_{14} \\
  m_3 & m_7 & m_{11} & m_{15}
\end{bmatrix}
\]
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 and 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**

\[
O = \begin{bmatrix}
\frac{2}{r} & 0 & 0 & \frac{r+l}{r} \\
0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\
0 & 0 & \frac{2}{f-n} & \frac{t+n}{f-n} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

**Perspective View**

\[
P = \begin{bmatrix}
\frac{2n}{r+l} & 0 & 0 & \frac{r+l}{r+l} \\
0 & \frac{2n}{t-b} & 0 & \frac{t+b}{t-b} \\
0 & 0 & \frac{2n}{(t+n)f-n} & \frac{2n}{(t+n)f-n} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
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
• LookAt( eyex, eyey, eyez, lookx, looky, lookz, upx, upy, upz )
  – up vector determines unique orientation
  – careful of degenerate positions
Creating the LookAt Matrix

\[ \hat{n} = \begin{bmatrix} \text{look} & \text{eye} \\ \text{look} & \text{eye} \end{bmatrix} \]

\[ \hat{u} = \begin{bmatrix} \hat{n} & \text{up} \\ \hat{n} & \text{up} \end{bmatrix} \]

\[ \hat{v} = \hat{u} \cdot \hat{n} \]

\[
\begin{bmatrix}
\hat{n} \\
\hat{u} \\
\hat{v}
\end{bmatrix} =
\begin{bmatrix}
\hat{u} & \hat{n}
\end{bmatrix}
\]

\[
\begin{bmatrix}
u_x & u_y & u_z & (\text{eye} \times \hat{u}) \\
v_x & v_y & v_z & (\text{eye} \times \hat{v}) \\
0 & 0 & 0 & (\text{eye} \times \hat{n})
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & 0 & 0 & 1
\end{bmatrix}
\]
• Move the origin to a new location

\[ T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
- 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
• Rotate coordinate system about an axis in space

Note, there’s a translation applied here to make things easier to see
\[ \vec{v} = (x \ y \ z) \]
\[ \vec{u} = \frac{\vec{v}}{||\vec{v}||} = (x \ y \ z) \]

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

\[ S = \begin{bmatrix} 0 & z & y \\ z & 0 & x \\ y & x & 0 \end{bmatrix} \]

\[ R_v(\ ) = M \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Vertex Shader for Rotation of Cube

```glsl
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
}
```
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0, 0.0, 0.0, 0.0, 
0.0, c.x, s.x, 0.0, 
0.0, -s.x, c.x, 0.0, 
0.0, 0.0, 0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0, 
0.0, 1.0, 0.0, 0.0, 
s.y, 0.0, c.y, 0.0, 
0.0, 0.0, 0.0, 1.0 );
mat4 rz = mat4( c.z, -s.z, 0.0, 0.0, 
               s.z,  c.z, 0.0, 0.0, 
               0.0,  0.0, 1.0, 0.0, 
               0.0,  0.0, 0.0, 1.0 );

color = vColor;
gl_Position = rz * ry * rx * vPosition;
Here, we compute our angles ($\Theta$) in our mouse callback.

```c
GLuint theta;  // theta uniform location
vec3 Theta;   // Axis angles

void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glUniform3fv( theta, 1, Theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}
```
Lighting
Lighting simulates how objects reflect light
  – material composition of object
  – light’s color and position
  – global lighting parameters

Usually implemented in
  – vertex shader for faster speed
  – fragment shader for nicer shading
Modified Phong Model

• Computes a color for each vertex using
  – Surface normals
  – Diffuse and specular reflections
  – Viewer’s position and viewing direction
  – Ambient light
  – Emission

• Vertex colors are interpolated across polygons by the rasterizer
  – *Phong shading* does the same computation per pixel, interpolating the normal across the polygon
    • more accurate results
Surface Normals

• Normals define how a surface reflects light
  – Application usually provides normals as a vertex attribute
  – Current normal is used to compute vertex’s color
  – Use unit normals for proper lighting
    • scaling affects a normal’s length
• Define the surface properties of a primitive

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuse</td>
<td>Base object color</td>
</tr>
<tr>
<td>Specular</td>
<td>Highlight color</td>
</tr>
<tr>
<td>Ambient</td>
<td>Low-light color</td>
</tr>
<tr>
<td>Emission</td>
<td>Glow color</td>
</tr>
<tr>
<td>Shininess</td>
<td>Surface smoothness</td>
</tr>
</tbody>
</table>

– you can have separate materials for front and back
// vertex shader

in vec4 vPosition;
in vec3 vNormal;
out vec4 color;

uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
void main()
{
    // Transform vertex position into eye coordinates
    vec3 pos = vec3(ModelView * vPosition);

    vec3 L = normalize(LightPosition.xyz - pos);
    vec3 E = normalize(-pos);
    vec3 H = normalize(L + E);

    // Transform vertex normal into eye coordinates
    vec3 N = normalize(vec3(ModelView * vNormal));
// Compute terms in the illumination equation
vec4 ambient = AmbientProduct;

float Kd = max( dot(L, N), 0.0 );
vec4 diffuse = Kd*DiffuseProduct;

float Ks = pow( max(dot(N, H), 0.0), Shininess );
vec4 specular = Ks * SpecularProduct;
if( dot(L, N) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0)

gl_Position = Projection * ModelView * vPosition;

color = ambient + diffuse + specular;
color.a = 1.0;
}
Fragment Shaders
Fragment Shaders

• A shader that’s executed for each “potential” pixel
  – fragments still need to pass several tests before making it to the framebuffer
• There are lots of effects we can do in fragment shaders
  – Per-fragment lighting
  – Texture and bump Mapping
  – Environment (Reflection) Maps
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
• A height field is a function \( y = f(x, z) \)
  – \( y \) represents the height of a point for a location in the \( x-z \) plane.
• Heights fields are usually rendered as a rectangular mesh of triangles or rectangles sampled from a grid
  – samples \( y_{ij} = f(x_i, z_j) \)
Displaying a Height Field

• First, generate a mesh data and use it to initialize data for a VBO

```cpp
float dx = 1.0/N, dz = 1.0/N;
for( int i = 0; i < N; ++i ) {
    float x = i*dx;
    for( int j = 0; j < N; ++j ) {
        float z = j*dz;
        float y = f( x, z );
        vertex[Index++] = vec3( x, y, z );
        vertex[Index++] = vec3( x, y, z + dz );
        vertex[Index++] = vec3( x + dx, y, z + dz );
        vertex[Index++] = vec3( x + dx, y, z );
    }
}
```

• Finally, display each quad using

```cpp
for( int i = 0; i < NumVertices ; i += 4 )
glDrawArrays( GL_LINE_LOOP, 4*i, 4 );
```
in vec4 vPosition;
in vec4 vColor;

uniform float time; // in milliseconds
uniform mat4 ModelViewProjectionMatrix;

void main()
{
    vec4 v = vPosition;
    vec4 u = sin( time + 5*v );
    v.y = 0.1 * u.x * u.z;
    gl_Position = ModelViewProjectionMatrix * v;
}
Adding Lighting

• Solid Mesh: create two triangles for each quad
• Display with

        glDrawArrays( GL_TRIANGLES, 0, NumVertices );

• For better looking results, we’ll add lighting
• We’ll do per-vertex lighting
  – leverage the vertex shader since we’ll also use it to vary the mesh
    in a time-varying way
uniform float time, shininess;
uniform vec4 vPosition, lightPosition, diffuseLight, specularLight;
uniform mat4 ModelViewMatrix, ModelViewProjectionMatrix, NormalMatrix;

void main()
{
    vec4 v = vPosition;
    vec4 u = sin( time + 5*v );
    v.y = 0.1 * u.x * u.z;

    gl_Position = ModelViewProjectionMatrix * v;

    vec4 diffuse, specular;
    vec4 eyePosition = ModelViewMatrix * vPosition;
    vec4 eyeLightPos = lightPosition;
vec3 N = normalize(NormalMatrix * Normal);
vec3 L = normalize(vec3(eyeLightPos - eyePosition));
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), shininess);
diffuse = Kd*diffuseLight;
specular = Ks*specularLight;
color = diffuse + specular;
Texture Mapping
Texture Mapping

Geometry

Image

Screen
Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer
  - “complex” textures do not affect geometric complexity
Applying Textures

• Three basic steps to applying a texture
  1. specify the texture
     • read or generate image
     • assign to texture
     • enable texturing
  2. assign texture coordinates to vertices
  3. specify texture parameters
     • wrapping, filtering
Texture Objects

• Have OpenGL store your images
  – one image per texture object
  – may be shared by several graphics contexts
• Generate texture names

    glGenTextures( n, *texIds );
• Create texture objects with texture data and state

```c
glBindTexture( target, id );
```

• Bind textures before using

```c
glBindTexture( target, id );
```
Specifying a Texture Image

- Define a texture image from an array of texels in CPU memory

```c
glTexImage2D( target, level, internal format, w, h, border, format, type, *texels );
```
Mapping a Texture

• Based on parametric texture coordinates
• coordinates needs to be specified at each vertex

\[(s, t) = (0.2, 0.8)\]

Texture Space

Object Space

\[(s, t) = (0.2, 0.8)\]

\((0.4, 0.2)\)

\((0.8, 0.4)\)
in vec4 texCoord;

// Declare the sampler
uniform float intensity;
uniform sampler2D diffuseMaterialTexture;

// Apply the material color
vec3 diffuse = intensity *
  texture(diffuseMaterialTexture, texCoord).rgb;
// add texture coordinate attribute to quad function

quad( int a, int b, int c, int d )
{
    vColors[Index] = colors[a];
    vPositions[Index] = positions[a];
    vTexCoords[Index] = vec2( 0.0, 0.0 );
    Index++;

    vColors[Index] = colors[b];
    vPositions[Index] = positions[b];
    vTexCoords[Index] = vec2( 1.0, 0.0 );
    Index++;

    ... // rest of vertices
}
// Create a checkerboard pattern
for ( int i = 0; i < 64; i++ ) {
    for ( int j = 0; j < 64; j++ ) {
        GLubyte c;
        c = (((i & 0x8 == 0) ^ (j & 0x8 == 0)) * 255;
        image[i][j][0] = c;
        image[i][j][1] = c;
        image[i][j][2] = c;
        image2[i][j][0] = c;
        image2[i][j][1] = 0;
        image2[i][j][2] = c;
    }
}
GLuint textures[1];
glGenTextures( 1, textures );

glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, textures[0] );

glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize, TextureSize, GL_RGB, GL_UNSIGNED_BYTE, image );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );

 gluTessBeginContour(&t)
in vec4 vPosition;
in vec4 vColor;
in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;

void main()
{
    color = vColor;
    texCoord = vTexCoord;
    gl_Position = vPosition;
}
in vec4 color;
in vec2 texCoord;

out vec4 fColor;

uniform sampler texture;

void main()
{
    fColor = color * texture( texture, texCoord );
}
Q & A

Thanks for Coming!
Resources
• Modern discussion
  – The OpenGL Superbible, 5th Edition

• Older resources
  – OpenGL and the X Window System
  – OpenGL Programming for Mac OS X
  – OpenGL ES 2.0 Programming Guide

• Not quite yet …
Online Resources

• The OpenGL Website: www.opengl.org
  – API specifications
  – Reference pages and developer resources
  – Downloadable OpenGL (and other APIs) reference cards
  – Discussion forums

• The Khronos Website: www.khronos.org
  – Overview of all Khronos APIs
  – Numerous presentations
• Feel free to drop us any questions:

angel@cs.unm.edu
shreiner@siggraph.org

• Course notes and programs available at

www.daveshreiner.com/SIGGRAPH
www.cs.unm.edu/~angel