API Details
Objects

- **Objects are OpenGL-managed data structures that consist of state and data**
- **Where GLSL is concerned, objects are named (given handles) by OpenGL, and these names are used by the application to subsequently refer to the created object**
- **Applications can provide data for objects and modify their state**
- **All objects can be shared across contexts**
Shader Source Code

• Source code intended for an OpenGL processor is called a shader

• Shaders are defined as an array of strings
  ▪ Strings need not be null-terminated, as length of strings are passed as well
  ▪ Pass a string length less than 0 to indicate a null-terminated string
Usage Model

- **Four steps to using a shader**
  - Send shader source to OpenGL
  - Compile the shader
  - Create an executable (i.e., link compiled shaders together)
  - Install the executable as part of current state

- **Goal was to mimic C/C++ source code development model**

- **Key benefits:**
  - Shader source is highly portable
  - No need to change app when compiler improvements occur
  - Shaders can be compiled at any time (e.g., at app initialization time or just before use)
  - Executables can be created at any time (e.g., at app initialization time or just before use)
  - Supports development of modular shaders
Shader Objects

- **Shader objects are created with:**
  - shaderID = glCreateShader(shaderType);

- **Shader source code is supplied to OpenGL using:**
  - glShaderSource(shaderID, numStrings, strings, lengths)

- **Shader objects are compiled with:**
  - glCompileShader(shaderID);
  - Call glGetShaderiv with the constant GL_COMPILE_STATUS to determine whether the shader was compiled successfully
Shader Objects

- **The shader object’s compiler information string can be obtained with:**
  - `glGetShaderInfoLog(shaderID, maxLen, actualLen, buffer)`
- **An executable for a programmable processor may be built from several shader objects**
  - One shader object might contain main, other shader objects might contain functions called by main
  - Resolved at link time
  - Supports modular development of complex shaders
Program Objects

- A program object is a container for shader objects
  - This establishes the set of shaders that need to be linked together when used
  - `programObj = glCreateProgram()`
  - `glAttachShader(programID, shaderID)`
  - `glDetachShader(programID, shaderID)`

- The shaders in a program object are linked with
  - `glLinkProgram(programID)`

- A program object is made current with:
  - `glUseProgram(programID)`
Object Deletion

- Shader objects and program objects are deleted with:
  - `glDeleteShader(shaderID)`
  - `glDeleteProgram(programID)`

  Data for a shader object isn’t actually deleted until it is no longer attached to any program object for any rendering context.

  Data for a program object is deleted when it is no longer in use by any context.
Shader Compatibility

- **Compatibility between the shaders in a program object can be checked with:**
  - `glGetProgramInfoLog(programID, maxLen, actualLen, buffer)`
  - Returns the info log for the specified program object
Linking and Using Program Objects

• **When `glLinkProgram` is called:**
  - Attached shader objects are linked together to create an executable program and the program object’s info log is updated
  - If the program object is currently in use, the re-linked executable is immediately made part of the current rendering state

• **When `glUseProgram` is called:**
  - If the program object contains compatible, valid shader objects (i.e., no link errors), then the executable programs it contains are made part of the current rendering state

• **Shaders in the program object are checked for compatibility**
  - If both a vertex shader and a fragment shader are supplied, they must be compatible
  - If only one of the two is supplied, it must be compatible with the fixed functionality interface defined by OpenGL
**Vertex Shader Input**

- **Vertex data is provided using the normal OpenGL mechanisms**
  - `glColor`, `glNormal`, `glTexCoord`, `glVertex`
  - Vertex arrays
  - Example: calling `glNormalf` results in setting the value of the built-in attribute `gl_Normal` in the current shader

- **Vertex shader is executed:**
  - Once when the `glVertex` command is called
  - Multiple times when `glDrawArrays` and other vertex array commands are called
**Vertex Shader Input**

- **Uniforms and Attributes**
  - Uniforms and attributes load data that is used in shaders

- **Attributes change per vertex**
  - Standard attributes are defined as GLSL built-ins (e.g. `gl_Vertex`, `gl_Normal`, `gl_Color`)
  - Generic attributes (tangent, temperature, pressure, velocity, etc.)
  - Implementations must allow at least 16 attributes that can hold up to the size of a vec4

- **Uniforms are constant per primitive or group of primitives**
  - Change relatively infrequently compared to attributes
  - At least 512 floats for a vertex shader and 64 for a fragment shader
Generic Attributes

- **New OpenGL 2.0 entry points can be used to provide generic attribute data**
  - `glVertexAttrib`  
- **Enhanced vertex arrays also allow generic attributes**
  - Call `glVertexAttribPointer` with the index of the user-defined array (a value from 0 to `GL_MAX_VERTEX_ATTRIBS - 1`
Generic Attributes

- **Generic attributes are bound to a variable name in a program object with:**
  - `glBindAttribLocation(programID, index, name)`
  - User-defined attributes may be bound explicitly before calling `glLinkProgram`, or they will be bound implicitly and the assigned location can be queried
  - `glGetActiveAttrib` is used to determine how many of the available attributes have been used by an executable program
  - `glGetActiveAttrib` should be called after calling `glLinkProgram`

- **Number of user-defined attributes is an implementation-dependent value that can be queried**
  - `GL_MAX_VERTEX_ATTRIBS`
  - Must be at least 16, each can contain up to four floats
Standard Vertex Attributes

Application calls
to set standard
vertex attributes

Current
attribute value

Built-in attribute variables
Generic Vertex Attributes

glVertexAttribARB(0, ...) → gl_Vertex or MyVertex
glVertexAttribARB(1, ...) → Opacity
glVertexAttribARB(2, ...) → Binormal
glVertexAttribARB(3, ...) → MyData
glVertexAttribARB(4, ...) → SpectralChannel03
... (for N, ...)

Application calls to set generic vertex attributes

Current attribute value

User-defined attribute variables (excluding gl_Vertex)
User-supplied Uniforms

- The location of a named uniform variable can be obtained with:
  - location = glGetUniformLocation(programID, name)
  - This call should be made after the call to glLinkProgram since the location of uniform variables is not known until linking occurs
  - A value of -1 is returned if the variable name is not found

- Data other than vertex data can be supplied to the current shaders with:
  - glUniform{1234|fi} (location, value)
  - glUniform{1234|fi}v (location, count, value)
  - glUniformMatrix{234}fv(location, count, transpose, matrix)
  - These calls cannot be issued between Begin/End
  - No API for supplying the complete contents of a structure in one call
Fragment Shader Input

- **Loading uniforms is done with the same API as for the vertex shader**
- **Fragment shaders can access the built-in variable**
  - `gl_FragCoord`
    - Contains window relative coordinates (x, y, z, 1/w) as computed by the preceding fixed functionality rasterization process
    - Z value is the depth value that may eventually be written into the depth buffer for the fragment
- **Fragment shaders can access the built-in variable**
  - `gl_FrontFacing`
    - Contains the result of the preceding fixed functionality “facingness” computation
    - True if fragment belongs to a primitive that is front-facing, false otherwise
    - Useful for implementing different shading for front/back faces
Fragment Shader Output

- Output of the fragment processor goes on to the fixed function fragment operations and frame buffer operations using built-in variables
  - `gl_FragColor`
  - `gl_FragDepth`
  - `gl_FragData[n]`

- Clamping or format conversion to the target buffer is done automatically outside of the fragment shader
Texture Access

- No restrictions on number of texture accesses or on number of dependent texture accesses
- Applications can continue to use standard OpenGL calls for loading textures and setting texture attributes
- Applications must define a “sampler” for each texture to be accessed by specifying the texture unit to be accessed
- When texture accesses occur within a shader, filtering, wrapping behavior, etc., are performed based on the attributes of the texture object being accessed
Simple Code
Example
Application Example

- The following application example is not complete, but illustrates how an application would create and use shaders.
- Complete source code examples are available on the 3Dlabs developer web site:
  - http://developer.3dlabs.com
int installBrickShaders(const GLchar *brickVertex,
   const GLchar *brickFragment)
{
    GLuint brickVS, brickFS, brickProg; // handles to objects
    GLint  vertCompiled, fragCompiled;  // status values
    GLint  linked;

    // Create a vertex shader object and a fragment shader object
    brickVS = glCreateShader(GL_VERTEX_SHADER);
    brickFS = glCreateShader(GL_FRAGMENT_SHADER);

    // Load source code strings into shaders
    glShaderSource(brickVS, 1, &brickVertex, NULL);
    glShaderSource(brickFS, 1, &brickFragment, NULL);
// Compile the brick vertex shader, and print out
// the compiler log file.

glCompileShader(brickVS);
glGetShaderiv(brickVS, GL_COMPILE_STATUS, &vertCompiled);
printShaderInfoLog(brickVS);

// Compile the brick fragment shader, and print out
// the compiler log file.

glCompileShader(brickFS);
glGetShaderiv(brickFS, GL_COMPILE_STATUS, &fragCompiled);
printShaderInfoLog(brickFS);

if (!vertCompiled || !fragCompiled)
    return 0;
Compiling and using shaders – 3 of 4

// Create a program object and attach the two compiled shaders

brickProg = glCreateProgram();
glAttachShader(brickProg, brickVS);
glAttachShader(brickProg, brickFS);

// Link the program object and print out the info log

glLinkProgram(brickProg);
glGetProgramiv(brickProg, GL_LINK_STATUS, &linked);
printProgramInfoLog(brickProg);

if (!linked)
    return 0;
Application Example

Compiling and using shaders – 4 of 4

// Install program object as part of current state

glUseProgram(brickProg);

// Set up initial uniform values

glUniform3f(getUniLoc(brickProg, "BrickColor"), 1.0, 0.3, 0.2);
glUniform3f(getUniLoc(brickProg, "MortarColor"), 0.85, 0.86, 0.84);
glUniform2f(getUniLoc(brickProg, "BrickSize"), 0.30, 0.15);
glUniform2f(getUniLoc(brickProg, "BrickPct"), 0.90, 0.85);
glUniform3f(getUniLoc(brickProg, "LightPosition"), 0.0, 0.0, 4.0);

return 1;
void printShaderInfoLog(GLuint shader)
{
    int infologLength = 0;
    int charsWritten = 0;
    GLchar *infoLog;

    printOpenGLError(); // Check for OpenGL errors
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infologLength);
    printOpenGLError(); // Check for OpenGL errors
    if (infologLength > 0)
    {
        infoLog = (GLchar*)malloc(infologLength);
        if (infoLog == NULL)
        {
            printf("ERROR: Could not allocate InfoLog buffer\n");
            exit(1);
        }
        glGetShaderInfoLog(shader, infologLength,
            &charsWritten, infoLog);
        printf("InfoLog:\n%s\n\n", infoLog);
        free(infoLog);
    }
    printOpenGLError(); // Check for OpenGL errors
}
Application Example

Getting the location of a uniform variable

GLint getUniLoc(GLuint program, const GLchar *name)
{
    GLint loc;

    loc = glGetUniformLocation(program, name);

    if (loc == -1)
        printf("No such uniform named \"%s\"\n", name);

    printOpenGLError(); // Check for OpenGL errors
    return loc;
}
Brick Shader – Lighting
Brick Vertex Shader

uniform vec3 LightPosition;
const float SpecularContribution = 0.3;
const float DiffuseContribution = 1.0 - SpecularContribution;
varying float LightIntensity;
varying vec2 MCposition;
void main( void )
{
    vec3 ecPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec   = normalize(LightPosition - ecPosition);
    vec3 reflectVec = reflect(-lightVec, tnorm);
    vec3 viewVec    = normalize(-ecPosition);
    float diffuse   = max(dot(lightVec, tnorm), 0.0);
    float spec      = 0.0;
    if (diffuse > 0.0)
    {
        spec = max(dot(reflectVec, viewVec), 0.0);
        spec = pow(spec, 16.0);
    }
    LightIntensity  = DiffuseContribution * diffuse + SpecularContribution * spec;
    MCposition     = gl_Vertex.xy;
    gl_Position     = ftransform();
}
Brick Shader - Parameters

BrickSize.y = 0.15
BrickSize.x = 0.30
BrickPct.x = 0.90
BrickPct.y = 0.85
BrickColor = (1.0, 0.3, 0.2)
MortarColor = (0.85, 0.86, 0.84)
Brick Shader – Step Function

BrickColor MortarColor

BrickPct.x  BrickPct.x+1  BrickPct.x+2
Fragment Shader Example

uniform vec3 BrickColor, MortarColor;
uniform vec2 BrickSize;
uniform vec2 BrickPct;
varying vec2 MCposition;
varying float LightIntensity;

void main(void)
{
    vec3 color;
    vec2 position, useBrick;
    position = MCposition / BrickSize;
    if (fract(position.y * 0.5) > 0.5)
        position.x += 0.5;
    position = fract(position);
    useBrick = step(position, BrickPct);
    color = mix(MortarColor, BrickColor, useBrick.x * useBrick.y);
    color *= LightIntensity;
    gl_FragColor = vec4(color, 1.0);
}