Programmable Graphics Hardware

- OpenGL Extensions
- Shading Languages
- Vertex Program
- Fragment Program

[Angel Ch. 9]
Introduction

- Recent major advance in real time graphics is the *programmable* pipeline:
  - First introduced by NVIDIA GeForce 3 (in 2001)
  - Supported by all modern high-end commodity cards
    - NVIDIA, ATI
  - Software Support
    - Direct X 8, 9, 10
    - OpenGL

- This lecture: *programmable pipeline and shaders*
OpenGL Extensions

• Initial OpenGL version was 1.0
• Current OpenGL version is 4.2

• As graphics hardware improved, new capabilities were added to OpenGL
  – multitexturing
  – multisampling
  – non-power-of-two textures
  – shaders
  – and many more
OpenGL Grows via Extensions

• Phase 1: vendor-specific: GL_NV_multisample

• Phase 2: multi-vendor:
  GL_EXT_multisample

• Phase 3: approved by OpenGL’s review board
  GL_ARB_multisample

• Phase 4: incorporated into OpenGL (v1.3)
OpenGL 2.0 Added Shaders

• Shaders are customized programs that replace a part of the OpenGL pipeline

• They enable many effects not possible by the fixed OpenGL pipeline

• Motivated by Pixar’s Renderman (offline shader)
Shaders Enable Many New Effects

- Complex materials
- Lighting environments
- Shadowing
- Advanced mapping
The Rendering Pipeline

CPU → Vertex Processor → Rasterizer → Fragment Processor → Frame Buffer

vertices → vertices → fragments → fragments
Shaders Replace Part of the Pipeline

- CPU
- Vertex Processor
- Rasterizer
- Fragment Processor
- Frame Buffer

vertices → vertices → fragments → fragments

customizable by a vertex program
customizable by a fragment program
Shaders

• Vertex shader (= vertex program)

• Fragment shader (= fragment program)

• Geometry shader (recent addition)

• Default shaders are provided by OpenGL (fixed-function pipeline)

• Programmer can install her own shaders as needed
Shaders Are Written in Shading Languages

• Early shaders: assembly language

• Since ~2004: high-level shading languages
  – OpenGL Shading Language (GLSL)
    • highly integrated with OpenGL
  – Cg (NVIDIA and Microsoft), very similar to GLSL
  – HLSL (Microsoft), almost identical to Cg
  – All of these are simplified versions of C/C++
Vertex Program

- **Input:** vertices, and per-vertex attributes:
  - color
  - normal
  - texture coordinates
  - many more

- **Output:**
  - vertex location in clip coordinates
  - vertex color
  - vertex normal
  - many more are possible
Simple Vertex Program in GLSL

/* pass-through vertex shader */

void main()
{
   gl_Position = gl_ProjectionMatrix
                * (gl_ModelViewMatrix * gl_Vertex);
}
Fragment Program

• Input: **pixels**, and per-pixel attributes:
  – color
  – normal
  – texture coordinates
  – many more are possible

• Inputs are outputs from vertex program, interpolated (by the GPU) to the pixel location!

• Output:
  – pixel color
  – depth value
Simple Fragment Program

/* pass-through fragment shader */

void main()
{
    gl_FragColor = gl_Color;
}

Simple Fragment Program #2

/* all-red fragment shader */

void main()
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
GLSL: Data Types

- **Scalar Types**
  - float - 32 bit, very nearly IEEE-754 compatible
  - int - at least 16 bit
  - bool - like in C++

- **Vector Types**
  - vec[2 | 3 | 4] - floating-point vector
  - ivec[2 | 3 | 4] - integer vector
  - bvec[2 | 3 | 4] - boolean vector

- **Matrix Types**
  - mat[2 | 3 | 4] - for 2x2, 3x3, and 4x4 floating-point matrices

- **Sampler Types**
  - sampler[1 | 2 | 3]D - to access texture images
GLSL: Operations

- Operators behave like in C++
- Component-wise for vector & matrix
- Multiplication on vectors and matrices

Examples:
- Vec3 t = u * v;
- float f = v[2];
- v.x = u.x + f;
GLSL: Swizzling

- Swizzling is a convenient way to access individual vector components

```glsl
vec4 myVector;
myVector.rgba;  // is the same as myVector
myVector.xy;    // is a vec2
myVector.b;     // is a float
myVector[2];    // is the same as myVector.b
myVector.xb;    // illegal
myVector.xxx;   // is a vec3
```
GLSL: Global Qualifiers

• Attribute
  – Information specific to each vertex/pixel passed to vertex/fragment shader
  – No integers, bools, structs, or arrays
  
  Example: Vertex Color

• Uniform
  – Constant information passed to vertex/fragment shader
  – Cannot be written to in a shader

  Example: Light Position
          Eye Position

• Varying
  – Info passed from vertex shader to fragment shader
  – Interpolated from vertices to pixels
  – Write in vertex shader, but only read in fragment shader

  Example: Vertex Color
          Texture Coords

• Const
  – To declare non-writable, constant variables

  Example: pi, e, 0.480
GLSL: Flow Control

- Loops
  - C++ style if-else
  - C++ style for, while, and do

- Functions
  - Much like C++
  - Entry point into a shader is void main()
  - No support for recursion
  - Call by value-return calling convention

- Parameter Qualifiers
  - in - copy in, but don’t copy out
  - out - only copy out
  - inout - copy in and copy out

Example function:

```cpp
void ComputeTangent(
    in vec3 N,
    out vec3 T,
    inout vec3 coord)
{
    if (dot(N, coord)>0)
        T = vec3(1,0,0);
    else
        T = vec3(0,0,0);
    coord = 2 * T;
}
```
GLSL: Built-in Functions

• Wide Assortment
  – Trigonometry (cos, sin, tan, etc.)
  – Exponential (pow, log, sqrt, etc.)
  – Common (abs, floor, min, clamp, etc.)
  – Geometry (length, dot, normalize, reflect, etc.)
  – Relational (less than, equal, etc.)

• Need to watch out for common reserved keywords
• Always use built-in functions, don’t implement your own
• Some functions aren’t implemented on some cards
GLSL: Accessing OpenGL State

• Built-in Variables
  – Always prefaced with gl_
  – Accessible to both vertex and fragment shaders

• Uniform Variables
  – Matrices (ModelViewMatrix, ProjectionMatrix, inverses, transposes)
  – Materials (in MaterialParameters struct, ambient, diffuse, etc.)
  – Lights (in LightSourceParameters struct, specular, position, etc.)

• Varying Variables
  – FrontColor for colors
  – TexCoord[] for texture coordinates
GLSL: Accessing OpenGL State

• Vertex Shader:
  – Have access to several vertex attributes: `gl_Color`, `gl_Normal`, `gl_Vertex`, etc.
  – Also write to special output variables: `gl_Position`, `gl_PointSize`, etc.

• Fragment Shader:
  – Have access to special input variables: `gl_FragCoord`, `gl_FrontFacing`, etc.
  – Also write to special output variables: `gl_FragColor`, `gl_FragDepth`, etc.
Example: Phong Shader ("per-pixel lighting")

- Questions?

- Goals:
  - C/C++ Application Setup
  - Vertex Shader
  - Fragment Shader
  - Debugging
Phong Shading Review

\[ I = \frac{1}{a + bq + cq^2} \left( k_d L_d (l \cdot n) + k_s L_s (r \cdot v)^\alpha \right) + k_a L_a \]
Phong Shader: Setup Steps

- Step 1: Create Shaders
  - Create handles to shaders
- Step 2: Specify Shaders
  - Load strings that contain shader source
- Step 3: Compiling Shaders
  - Actually compile source (check for errors)
- Step 4: Creating Program Objects
  - Program object controls the shaders
- Step 5: Attach Shaders to Programs
  - Attach shaders to program objects via handle
- Step 6: Link Shaders to Programs
  - Another step similar to attach
- Step 7: Enable Shaders
  - Finally, let OpenGL and GPU know that shaders are ready
Phong Shader: Vertex Program

varying vec3 n;
varying vec3 vtx;
void main(void)
{
    // transform vertex position to eye coordinates:
    vtx = vec3(gl_ModelViewMatrix * gl_Vertex);
    // transform normal:
    n = normalize(gl_NormalMatrix * gl_Normal);
    // transform vertex position to clip coordinates:
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

these will be passed to fragment program (interpolated by hardware)
Phong Shader: Fragment Program

```cpp
varying vec3 n;
varying vec3 vtx;
void main (void)
{
    // we are in eye coordinates, so eye pos is (0,0,0)
    vec3 l = normalize(gl_LightSource[0].position.xyz - vtx);
    vec3 v = normalize(-vtx);
    vec3 r = normalize(-reflect(l,n));
    // calculate ambient, diffuse, specular terms:
    vec4 lamb = gl_FrontLightProduct[0].ambient;
    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,l), 0.0);
    vec4 lspec = gl_FrontLightProduct[0].specular * pow(max(dot(r,v),0.0), gl_FrontMaterial.shininess);
    // write total color:
    gl_FragColor = gl_FrontLightModelProduct.sceneColor + lamb + Idiff + lspec;
}
```
Debugging Shaders

• More difficult than debugging C programs

• Common show-stoppers:
  – Typos in shader source
  – Assuming implicit type conversion
  – Attempting to pass data to undeclared varying/uniform variables

• Extremely important to check error codes, use status functions like:
  – `glGetObjectParameter{I|f}vARB (GLhandleARB shader, GLenum whatToCheck, GLfloat * statusVals)`

• Subtle Problems
  – Shader too long
  – Use too many registers
Summary

- OpenGL Extensions
- Shading Languages
- Vertex Programs
- Fragment Programs
- Phong Shading in GLSL