2 – OpenGL Overview

OpenGL vs. DirectX

- DirectX
  - Powerful
  - Proprietary
  - Runs only under Windows

- OpenGL
  - Powerful
  - Open Standard
  - Runs on: Windows, Linux, Mac OS, ...
  - Bindings for several languages available

Khronos Group

The Khronos Group: an “Open Standard API” industry consortium. OpenGL Architecture Review Board (ARB) is a “Khronos Working Group”

OpenGL Versions

- GL 1.0
- GL 1.1
- GL 1.2
- GL 1.3
- GL 1.4
- GL 1.5
- ES 1.0
- ES 1.1
- ES 2.0
- GL 2.0
- GL 2.1
- GL 3.0
- GL 3.1
- GL 3.2
- GL 3.3
- GL 4.0, 4.1, 4.2, 4.3, 4.4, 4.5

JOGL: the Java OpenGL binding

- Originally a joint project between Sun & SGI
- Now an Open-Source community project (“jogamp”)
- Supports up to OpenGL 4.5

JOGL Program Structure
**JOGL Interface Methods**

```java
/** This interface defines the methods which must be implemented by a
 * component (e.g. a JFrame) that wishes to respond to "GLEvents" (for
 * example, a "display callback" from a GLCanvas's display() method.)
 */

public interface GLEventListener {
    /** Called when OpenGL is initialized */
    void init(GLAutoDrawable drawable);
    /** Called when the drawable object is repainted */
    void display(GLAutoDrawable drawable);
    /** Called when the drawable object's container is resized */
    void reshape(GLAutoDrawable d, int x, int y, int width, int height);
    /** Called to notify the listener to release any OpenGL resources
     * which it may have associated with the specified drawable */
    void dispose(GLAutoDrawable d);
}
```

**JOGL Interface Methods (cont.)**

```java
/** This interface defines the methods which are implemented by
 * "GLAutoDrawable"s (for example, JOGL "GLCanvas" objects). These
 * methods allow a listener to register to be "called back" when the
 * GLAutoDrawable's display() method has been called. */

public interface GLAutoDrawable {
    /** Adds a "callback" listener to a drawable object */
    public void addGLEventListener(GLEventListener listener);
    /** Called by the application or system to repaint the
     * drawable object; invokes display(this) in all registered listeners. */
    public void display();
    /** Returns the GL object which knows how to draw on this
     * drawable object */
    public GL getGL();
    //... other interface methods ...
}
```

**The Graphics “Pipeline”**

- Implemented by the combination of the graphics driver and
  graphics (hardware) card
- Can be FIXED-FUNCTION or PROGRAMMABLE, depending on implementation version

**Hardware Architecture**

**Early structure:**

```
Main Memory

CPU

Frame Buffer (Memory)

Display
```

**Next stage: graphics processors (GPU’s)**

```
Memory

CPU

GPU

VRAM

Frame Buffer
```

**Hardware Architecture (cont.)**

**First evolution: simple processors**

```
Memory

CPU

Processor

Register

Frame Buffer
```

**Hardware Architecture (cont.)**

**Next stage: graphics processors (GPU’s)**

```
Memory

CPU

GPU

VRAM

Frame Buffer
```

**The Graphics “Pipeline”**

```
Application Code

OpenGL API

Other Data

"Client Side" "Server Side"
```

**Hardware Architecture**

```
Application code

OpenGL Driver (software)

Vertex Processing

Primitive Assembly

(Rendering Pipeline)

Projection

Clipping/Culling/etc.

Rasterization

Fragment Processing

Display
```

**Hardware Architecture (cont.)**

```
Application code

OpenGL Driver (software)

Frame Buffer (Memory)

"Firmware"
```

**Hardware Architecture (cont.)**

```
Application code

OpenGL Driver (software)

Frame Buffer (Memory)

"Firmware"
```

**Hardware Architecture (cont.)**

```
Application code

OpenGL Driver (software)

Frame Buffer (Memory)

"Firmware"
```

**Hardware Architecture (cont.)**

```
Application code

OpenGL Driver (software)

Frame Buffer (Memory)

"Firmware"
```
THE PROGRAMMABLE PIPELINE

Hardware ("Shader") Programming

High-level languages:
- HLSL ("High-Level Shading Language")
  - Proprietary (Microsoft)
  - Powerful
  - Specific to DirectX
- Cg ("C for graphics")
  - Proprietary (nVidia)
  - Supports both DirectX and OpenGL APIs (more complex)
- GLSL ("OpenGL Shading Language")
  - Open standard
  - Compiles to all common vendor chips
  - Can run "on top of" DirectX, or directly on hardware

GLSL Overview

Based on C; looks much like C/C++/Java

Every Shader has a "main":
```c
void main () { ... }
```

Other similarities:
- Constants, identifiers, most operators, expression syntax
- Statement forms (assignment, if/then/else; for, while; switch)
- Function calls
  - Parameter passing is "Call By Value-Return":
    - Input parameters are copied in at call time
    - Output parameters are copied back to caller at exit
  - Parameters have a "mode qualifier": in, out, or inout
```c
int myFunc (in float a, out float b) { .. }
```

GLSL Versions

<table>
<thead>
<tr>
<th>OpenGL Version</th>
<th>GLSL Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>1.0</td>
<td>2003; Partial GLSL</td>
</tr>
<tr>
<td>2.0</td>
<td>1.1</td>
<td>2004</td>
</tr>
<tr>
<td>2.1</td>
<td>1.2</td>
<td>2006; ≈ Direct3D 9</td>
</tr>
<tr>
<td>3.0</td>
<td>1.3</td>
<td>2008</td>
</tr>
<tr>
<td>3.1</td>
<td>1.4</td>
<td>2009</td>
</tr>
<tr>
<td>3.2</td>
<td>1.5</td>
<td>2009</td>
</tr>
<tr>
<td>3.3</td>
<td>3.3</td>
<td>2010; ≈ D3D 10 (&quot;Shader Model 4.0&quot;)</td>
</tr>
<tr>
<td>4.0</td>
<td>4.0</td>
<td>2010</td>
</tr>
<tr>
<td>4.1</td>
<td>4.1</td>
<td>2010</td>
</tr>
<tr>
<td>4.2</td>
<td>4.2</td>
<td>2011; ≈ D3D 11 (&quot;Shader Model 5.0&quot;)</td>
</tr>
<tr>
<td>4.3</td>
<td>4.3</td>
<td>2012</td>
</tr>
<tr>
<td>4.4</td>
<td>4.4</td>
<td>2013</td>
</tr>
<tr>
<td>4.5</td>
<td>4.5</td>
<td>2014</td>
</tr>
</tbody>
</table>

Features NOT In GLSL

- No automatic type promotion
  ```c
  float f = 0; //compiler error!
  ```
- No pointers
- No Strings or Chars
- No unions
- No enumerated types
- No bitfield structures or bitwise operators
- No references to files
  - No `#include`
  - (But there is a preprocessor; `#define` works...)

GLSL Variables

- Functions can declare "local variables" of any type
- Variables can also be declared outside the scope of any function
  - Used for input/output communication with shaders
  - Must have a qualifier indicating how the variable is used: `const`, `in`, or `out`
  - "Use qualifier" is in addition to "type"
Example: "Identity" Vertex Shader

// A simple GLSL "Identity" vertex shader -- simply copies the input vertex position and color to the output

//Version 430
//indicate this shader requires GLSL Version 3.3 or higher

in vec3 vertPos;  // the vertex geometric position
in vec4 vertColor;  // the vertex color
out vec4 varyingColor;  // the varying (interpolated) color passed to the frag shader

void main(void)
{
    varyingColor = vertColor;  // pass the input color to the output
    gl_Position = vec4(vertPos,1);  // pass the input position to the output
}

"Identity" Fragment Shader

// A simple GLSL "Identity" fragment shader. All it does is pass the input color from the rasterizer to the shader color output.

//Version 430

in vec4 varyingColor;  // the interpolated color from the rasterizer
out vec4 fragColor;  // the color assigned to the fragment

void main (void)
{
    fragColor = varyingColor;
}

Shader Program Organization

Creating & Installing Shaders

1. Create one or more (empty) shader objects:
   shaderID = glCreateShader(shaderType)

2. Provide source code (an array of Strings) for each shader:
   glShaderSource(shaderID, numStrings, arrayOfStrings, arrayOfStringLengths)

3. Compile each shader:
   glCompileShader(shaderID)

4. Create one or more (empty) program objects:
   programID = glCreateProgram()

5. Attach shader object(s) to program objects:
   glAttachShader(programID, shaderID)

6. Link program objects:
   glLinkProgram(programID)

7. Install a program object into OpenGL pipeline:
   glUseProgram(programID)