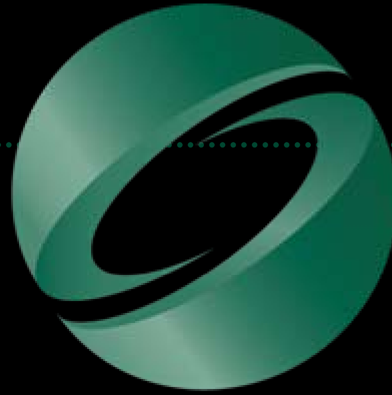


SIGGRAPH2006

GLSL for Open GL ES

An Introduction.

Robert J. Simpson
Architect, ATI Research

Programmer’s model

SIGGRAPH2006

Attributes
(~8)

Vertex Uniforms
(~96)

Vertex
Shader

Primitive
Assembly
& Rasterize

Varyings
(~8)

Fragment Uniforms
(~16)

Fragment
Shader

Per-Sample
Operations

Vertex Shader

Uniforms

Textures

Attribute 0
Attribute 1
Attribute 2
Attribute 3
Attribute 4
Attribute 5
Attribute 6
Attribute 7

Vertex Shader

Varying 0
Varying 1
Varying 2
Varying 3
Varying 4
Varying 5
Varying 6
Varying 7

gl_Position
gl_FrontFacing
gl_PointSize

Temporary variables

Fragment Shader

Uniforms

Textures

Varying 0
Varying 1
Varying 2
Varying 3
Varying 4
Varying 5
Varying 6
Varying 7

gl_FragCoord
gl_FrontFacing
gl_PointPosition

Fragment Shader

gl_FragColor

gl_Position

Temporary variables

GLSL ES Overview

SIGGRAPH2006

- 'C' – like language
- Many simplifications
  - No pointers
  - No implicit type conversion
  - Simplified preprocessor
- Some graphics-specific additions
  - Built-in vector and matrix types
  - Built-in functions
- Similar to desktop GLSL
  - Removal of most OpenGL fixed function state
  - Restrictions on shader complexity
  - Fewer sampler modes
  - No access to frag depth
  - Support for mixed precisions
  - More general invariance mechanism.

GLSL ES Preprocessor

- Comments
  ```
  //
  /*   */
  ```
- Macros
  ```
  #
  #define
  #undef
  ```
- Control
  ```
  #if
  #ifdef
  #ifndef
  #else
  #elif
  #endif
  #error
  ```
- Operators
  ```
  defined
  ```
- Extensions
  ```
  #pragma
  #extension
  #version
  #line
  ```

GLSL ES Types

- Scalar

  `void float int bool`

- Vector
  - boolean: **`bvec2 bvec3 bvec4`**
  - integer: **`ivec2 ivec3 ivec4`**
  - floating point: **`vec2 vec3 vec4`**

- Matrix

  `mat2 mat3 mat4`

- Sampler

  `sampler2D`

- Container

  `struct`

  Arrays

GLSL ES Storage Qualifiers

- `const`
  - Local constants within a shader.
- `uniform`
  - ‘Constant shader parameters’ (light position/direction, texture units, …)
  - Do not change per vertex.
- `attribute`
  - Per-vertex values (position, normal,…)
- `varying`
  - Generated by vertex shader
  - Interpolated by the rasterizer to generate per pixel values
  - Used as inputs to Fragment Shader
  - e.g. texture coordinates

Function Parameter Qualifiers

- Functions parameters can be used to pass values in or out or both
- Call by value 'copy in, copy out' semantics.
- Qualifiers:
  - `in` **(default)**
  - `out`
  - `inout`
- Can use 'const' with 'in'.
- Functions can still return a value
  - But use a parameter if returning an array
- e.g.

```
bool f(const in vec2 v, out int a[2])
{
    ...
}
```

GLSL ES Precision Qualifiers

- Rationale
  - ALU and register resources are scarce.
  - Many operations require only limited precision
- Available float precisions
  - `lowp float`
  - `mediump float`
  - `highp float`
- Available int precisions
  - `lowp int`
  - `mediump int`
  - `highp int`

GLSL ES Precision Qualifiers

- `lowp float`
  - Typically implemented by fixed point sign + 1.8 fixed point.
  - Range is -2.0 < x < 2.0
  - Resolution 1/256
  - Use for simple colour blending

- `mediump float`
  - Typically implemented by sign + 5.10 floating point
  - -16384 < x < 16384
  - Resolution 1 part in 1024
  - Use for HDR blending, some texture coordinate calculations

Precision Qualifiers

- `highp float`
  - Typically implemented by 24 bit float (16 bits of mantissa)
  - range $\pm 2^{62}$
  - Resolution 1 part in 2^{16}
  - Use of texture coordinate calculation e.g. environment mapping
- single precision
  - Not explicit in GLSL but usually available in the vertex shader

GLSL ES Precision Qualifiers

- Can specify per variable or set a default.
- Per Variable:

```
mediump float x;
```

- Set default:

```
precision mediump float;
```

- Can change the default:

```
{
    precision mediump float;
    float x; // x is a medium precision
             // float
    precision lowp float;
    float y; // y is a low precision float
}
```

GLSL ES Precision

- Precision of a sub-expression depends entirely on the operands
- Evaluated at the highest precision of

```
lowp float x;
mediump float y;
highp float z = x * y; // '*' evaluated
                       // at medium
                       // precision
```

- Literals do not have any defined precision

```
lowp float x;
highp float z = x * 2.0 + 1.2;
                          // evaluated at
                          // low precision
```

GLSL ES Precision

- Some special cases
  - If no operands have a precision, use outer-level sub-expression:

```
lowp float x = 1.0 / 3.0;  // evaluated
                           // at low
                           // precision
```

  - Use default precision if required

```
bool b = (1.0/3.0 > 0.33); // Must have
                           // default
                           // precision
                           // defined.
```

GLSL ES Constructors

- Replaces type casting
- All named types have constructors available
  - Includes built-in types, structs
  - Excludes arrays
- No implicit conversion: must use constructors
- Int to Float:

```
int n = 1;
float x,y;
x = float(n);
y = float(2);
```

- Concatenation:

```
float x = 1.0,y = 2.0;
vec2 v = vec2(x,y);
```

- Struct initialization

```
struct S {int a; float b;};
S s = S(2, 3.5);
```

GLSL ES Swizzle operators

- Use to select a set of components from a vector
- Can be used in L-values

```
vec2 u,v;
v.x = 2.0;                 // Assignment to single
                           // component
float a = v.x;             // Component selection
v.xy = u.yx;               // swap components
v = v.xx;                  // replicate components
v.xx = u;                  // Error
```

- Component sets

  Use one of:

  ```
  xyzw
  rgba
  stpq
  ```

- Indexing operator

```
vec4 u,v;
float x = u[0]; // equivalent to u.x
```

- Must use indexing operator for matrices

GLSL ES Features cont.

- Operators
  - ++ -- + - ! () []
  - * / + -
  - < <= > >=
  - == !=
  - && ^^ ||
  - ?:
  - = *= /= += -=
- Flow control
  - <expression> ? <expression_1> : <expression_2>
  - `if else`
  - `for while do`
  - `return break continue`
  - `discard` (fragment shader only)

GLSL Built-in Variables

- Aim of ES is to reduce the amount of fixed functionality
  - Ideal would be a totally pure programmable model
  - But still need some

- Vertex shader
  - `vec4  gl_Position;` // Write-only (required)
  - `float gl_PointSize;` // Write-only
- Fragment shader
  - `vec4  gl_FragCoord;` // Read-only
  - `bool  gl_FrontFacing;` // Read-only
  - `vec2  gl_PointCoord;` // Read-only
  - `float gl_FragColor;` // Write only

GLSL ES Built-in Functions 1

- General
  - `pow, exp2, log2, sqrt, inversesqrt`
  - `abs, sign, floor, ceil, fract, mod, min, max, clamp`
- Trig functions
  - `radians, degrees, sin, cos, tan, asin, acos, atan`
- Geometric
  - `length, distance, cross, dot, normalize, faceForward, reflect, refract`

GLSL ES Built-in Functions 2

- Interpolations
  - **mix(x,y,alpha)**
    x*(1.0-alpha) + y*alpha)
  - **step(edge,x)**
    x <= edge ? 0.0 : 1.0
  - **smoothstep(edge0,edge1,x)**
    t = (x-edge0)/(edge1-edge0);
    t = clamp(t, 0.0, 1.0);
    return t*t*(3.0-2.0*t);

- Texture
  - **texture1D, texture2D, texture3D, textureCube**
  - **texture1DProj, texture2DProj, textureCubeProj**

GLSL ES Built-in Functions

- Vector comparison (`vecn, ivecn`)
  - `bvecn lessThan(vecn, vecn)`
  - `bvecn lessThanEqual(vecn, vecn)`
  - `bvecn greaterThan(vecn, vecn)`
  - `bvecn greaterThanEqual(vecn, vecn)`
- Vector comparison (`vecn`, `ivecn`, `bvecn`)
  - `bvecn equal(vecn, vecn)`
  - `bvecn notEqual(vecn, vecn)`
- Vector (`bvecn`)
  - `bvecn any(bvecn)`
  - `bvecn all(bvecn)`
  - `bvecn not(bvecn)`
- Matrix
  - `matrixCompMult (matn, matn)`

Invariance: The Problem

- Mathematical operations are not precisely defined
- Same code may produce (slightly) different results
- Two cases to consider:
  - Invariance within a shader
  - Invariance between shaders

GLSL ES: Invariance – the problem

- Consider a simple transform in the vertex shader:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

x' = ax + by + cz + dw

But how is this calculated in practice?

- There may be several possible code sequences

GLSL ES: Invariance

e.g.

```
MUL R1, a, x
MUL R2, b, y
MUL R3, c, z
MUL R4, d, w
ADD R1, R1, R2
ADD R3, R3, R4
ADD R1, R1, R3
```

or

```
MUL R1, a, x
MADD R1, b, y
MADD R1, c, z
MADD R1, d, w
```

GLSL ES: Invariance

- Three reasons the result may differ:
  - Use of different instructions
  - Instructions executed in a different order
  - Different precisions used for intermediate results (only minimum precisions are defined)
- But it gets worse...

GLSL ES Invariance

- Modern compilers may rearrange your code
  - Values may lose precision when written to a register
  - Sometimes it is cheaper to recalculate a value rather than store it in a register.
    But will it be calculated the same way?

e.g.

```
uniform sampler2D tex1, tex2;
...
const vec2 pos = ...;
vec4 col1 = texture2D(tex1, pos);
...
vec4 col2 = texture2D(tex2, pos);// is this
  the same value?
gl_FragColor = col1 – col2;
```

Invariance – The solution

- Solution is in two parts:
  - invariant keyword to specify specific variables are invariant

```
invariant varying vec3 LightPosition;
```

    - Currently can only be used on outputs
  - Global switch to make all variable invariant

```
#pragma STDGL invariant(all)
```

- Invariance flag controls both invariance within shaders and invariance between shaders.
- Usage
  - Turn on invariance to make programs 'safe' and easier to debug
  - Turn off invariance to get the maximum optimization from the compiler.

Examples: ‘Null Shader’

- Vertex

```
attribute vec4 VertexPositionIn; // Input to
                                 // Vertex Shader
attribute vec4 VertexColourIn;   // Input to
                                 // Vertex Shader
varying vec4 VertexColorOut;     // Output from
                                 // Vertex shader

void main()
{
  VertexColorOut = VertexColorIn
  gl_Position = VertexPositionIn;
}
```

Examples: ‘Null Shader’

- Fragment

```
varying vec4 FragmentColorIn;//Input to
                             //Fragment Shader

void main()
{
  gl_FragColor = FragmentColorIn;
}
```

Vertex Shader functions

SIGGRAPH2006

- The vertex shader can do:
  - Transformation of position using model-view and projection matrices
  - Transformation of normals, including renormalization
  - Texture coordinate generation and transformation
  - Per-vertex lighting
  - Calculation of values for lighting per pixel
- The vertex shader cannot do:
  - Anything that requires information from more than one vertex
  - Anything that depends on connectivity.
  - Any triangle operations (e.g. clipping, culling)
  - Access colour buffer

Example Vertex Shader

- Diffuse lighting

```
uniform mat4 ModelViewProjectionMatrix, NormalMatrix;
uniform vec4 LightSourceDiffuse, LightSourcePosition, MaterialDiffuse;
attribute vec4 InputPosition, InputNormal, InputTextureCoordinates;
varying vec4 VertexColour;
varying vec4 TextureCoordinates;

void main()
{
   vec3 normal, lightDirection;
   vec4 diffuse;
   float NdotL;

   normal = normalize(NormalMatrix * Normal);
   lightDirection = normalize(vec3(LightSourcePosition));
   NdotL = max(dot(normal, lightDirection), 0.0);
   diffuse = MaterialDiffuse * LightSourceDiffuse;
   VertexColor = NdotL * diffuse;

   TextureCoordinates = InputTextureCoordinates;

   gl_Position = ModelViewProjectionMatrix * position;
}
```

Fragment Shader Functions

- The fragment shader can do:
  - Texture blending
  - Fog
  - Alpha testing
  - Dependent textures
  - Pixel discard
  - Bump and environment mapping
- The fragment shader cannot do:
  - Blending with colour buffer
  - ROP operations
  - Depth or stencil tests
  - Write depth

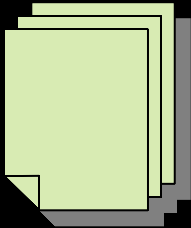
Example Fragment Shader

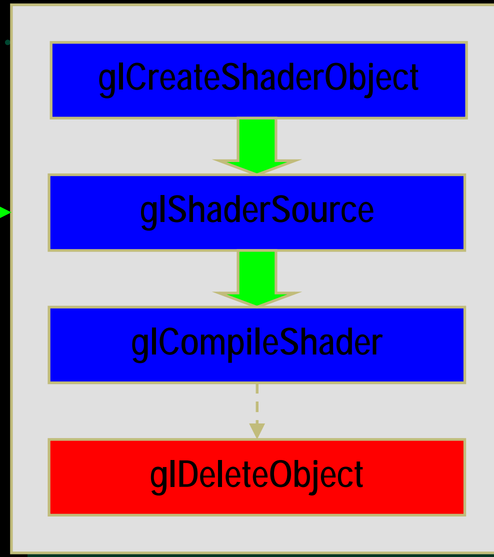
- Simple Texture Blend

```
uniform sampler2D TextureHandle;
varying vec2 TextureCoordinates;
varying vec4 VertexColour;
void main()
{
  vec4 texel = texture2D (TextureHandle,
  TextureCoordinates);
  gl_FragColor = texel * VertexColour;
}
```

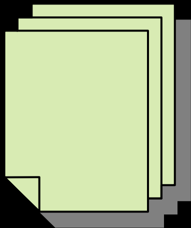
Compiling and using a shader

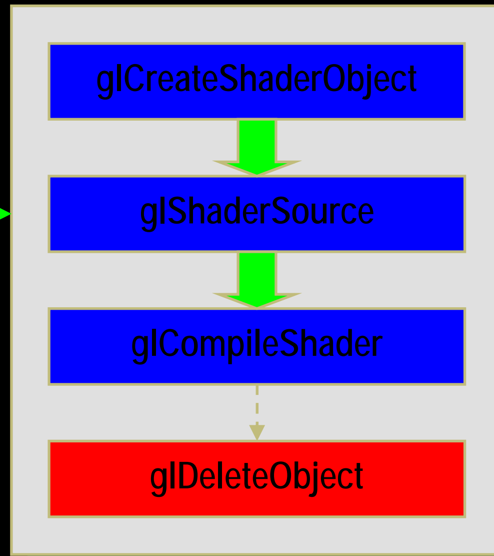
Vertex Shader

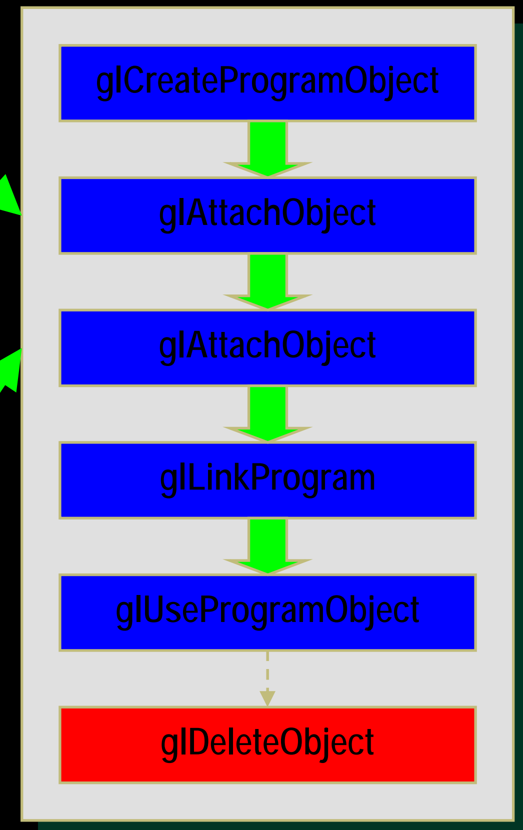




Fragment Shader







Compiling and using a shader: The code

```
// Create Shader Objects
GLhandleARB programObject = glCreateProgramObjectARB();
GLhandleARB vertexShaderObject =
    glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
GLhandleARB fragmentShaderObject =
    glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

GLcharARB *vertexShaderSource = readShaderFile(vertexShaderFilename);
GLcharARB *fragmentShaderSource = readShaderFile(fragmentShaderFilename);

// Load code into shader objects
glShaderSourceARB(vertexShaderObject, 1, vertexShaderSource, NULL);
glShaderSourceARB(fragmentShaderObject, 1, fragmentShaderSource, NULL);

glCompileShaderARB(vertexShaderObject);
glCompileShaderARB(fragmentShaderObject);

glAttachObjectARB(programObject, vertexShaderObject);
glAttachObjectARB(programObject, fragmentShaderObject);

glLinkProgramARB(programObject);

glUseProgramObjectARB(programObject);
```

Performance Tips

- Keep fragment shaders simple
  - Fragment shader hardware is expensive.
  - Early implementations will not have good performance with complex shaders.
- Try to avoid using textures for function lookups.
  - Calculation is quite cheap, accessing textures is expensive.
  - This is more important with embedded devices.
- Minimize register usage
  - Embedded devices do not support the same number of registers compared with desktop devices. Spilling registers to memory is expensive.
- Minimize the number of shader changes
  - Shaders contain a lot of state
  - May require the pipeline to be flushed
  - Use uniforms to change behaviour in preference to loading a new shader.

Future Directions

SIGGRAPH2006

- Sample Shaders
  - Enables alpha testing at per-sample resolution
  - Enables more of the fixed function pipeline to be removed.
  - Allows more programmability when using multi-sampling.
  - e.g. Read and write depth and stencil
- Object (Geometry) Shaders
  - Programmable tessellation
  - Higher order surfaces
  - Procedural geometry
  - Possibility of accelerating many more algorithms e.g. shadows, occlusion culling.

Future ES Pipeline?

API

Uniform Shader

Vertex Buffer Objects

Primitive Processing

Vertex Shader

Primitive Assembly

Object Shader

Rasterizer

Fragment Shader

Sample Shader

Frame Buffer

Any Questions?

SIGGRAPH2006

KHRONOS
GROUP