GLSL Language Details

Design Focus
- Based on syntax of ANSI C
- Some additions to support graphics functionality
- Some additions from C++
- Some differences for a cleaner language design

Additions for Graphics
- Vector types are supported for floats, integers, and booleans
  - Can be 2-, 3-, or 4-component
- Floating point matrix types are supported
  - 2x2, 3x3, or 4x4
- Type qualifiers attribute, uniform, and varying
- Built-in names for accessing OpenGL state and for communicating with OpenGL fixed functionality
- A variety of built-in functions are included for common graphics operations
  - Square root, trig functions, geometric functions, texture lookups, etc.
- Keyword discard to cease processing of a fragment
- Vector components are named (.rgba, .xyzw, .stpq) and can be swizzled
- Sampler data type is added for texture access

Additions from C++
- Function overloading based on argument types
- Function declarations are required
- Variables can be declared when needed
- Struct definition automatically performs a corresponding typedef
- Data type bool
ANSI C Features Not Supported

- Automatic promotion of data types
- Double, byte, short, long and unsigned byte/short/int/long
- Switch statements, goto statements, and labels
- Pointers and pointer-related capabilities
- Character and string literals
- Unions
- Enum
- Bit-fields
- Modulus and bit-wise operators
  - %, ~, >>, <<, |, &, %~, <<=, >>=, &=, ^=, !=
- File-based preprocessor directives
- Number sign-based preprocessor operators
  - #, #@, ##, etc.
- sizeof

Other Differences

- Constructors are used for conversion rather than type casts
- Function parameters are passed by value-return

Basics

- No inherent limit on hard-to-count resources such as registers or instructions
  - But limits may exist on early implementations
- Well-formed shaders are portable
- Ill-formed shaders may compile but are not portable
- Compilers must report lexical, grammatical, and syntactical errors
- Linkers must report compatibility errors, unresolved references, and out-of-resource errors
- Shaders containing errors cannot be executed
- Compilers may report warnings about code that limits performance
- Some slight differences between the language for vertex shaders and the language for fragment shaders
  - Built-in variables, type qualifiers, and built-in functions differ slightly

Source Code

- The source code for a shader consists of an array of strings
- Each string may contain multiple lines of source code, separated by new-lines
- A line of source code may be made of multiple strings
- Compiler diagnostic messages identify the source string and the line within the string that caused the error
- Source strings are numbered starting from 0
- When parsing, current line number is number of new-lines processed plus 1
Basic Structure

- A shader is a sequence of declarations and function bodies
- Curly braces are used to group sequences of statements
- A shader must have a `main` function
- Statements end with a semi-colon

Comments

- Comments are delimited by /* and */, or by // and a new-line
- Comments cannot be nested

Basic Types – 1 of 2

- `float`, `vec2`, `vec3`, `vec4`
  - 1, 2, 3, or 4 floating point values
  - Preferred data types for most processing
- `int`, `ivec2`, `ivec3`, `ivec4`
  - 1, 2, 3, or 4 integer values
  - Integer for loops and array index
  - Underlying hardware not expected to support integers natively
  - Limited to 16 bits of precision, plus sign
  - No guaranteed wrapping behavior
- `bool`, `bvec2`, `bvec3`, `bvec4`
  - 1, 2, 3, or 4 boolean values
  - As in C++, contains true or false
  - Used in expressions for conditional jumps
  - Underlying hardware not expected to support booleans natively
- `mat2`, `mat3`, `mat4`
  - Floating point square matrix
  - Used to perform transformation operations

Basic Types – 2 of 2

- `void`
  - Used for functions that do not return a value
- `sampler1D`, `sampler2D`, `sampler3D`
  - Handles for accessing 1D, 2D, and 3D textures
  - Used in conjunction with texture access functions
- `samplerCube`
  - Handle for accessing a cube map texture
  - Used in conjunction with texture access functions
- `sampler1DShadow`, `sampler2DShadow`
  - Handles for accessing 1D or 2D depth textures with an implicit comparison operation
  - Used in conjunction with texture access functions
Arrays

- An aggregation of variables of the same type
- All basic types and structures can be aggregated into arrays
- Only 1D arrays are supported
- Size of array can be expressed as an integral constant expression within square brackets ([ ])
- Arrays can be declared without a size, and then re-declared later with the same type and a size
- Using an index that goes beyond an array’s bounds results in undefined behavior
- Examples:
  
  ```
  float ramp[10];
  vec4 colors[4];
  bool results[3];
  ```

Structures

- User-defined types can be created using struct with previously defined types
- Example:
  ```
  struct surfMaterial
  {
    float ambient;
    float diffuse;
    float specular;
    vec3 baseColor;
  } surf;
  ```
  ```
  surfMaterial surf1, surf2;
  ```
- Creates a new type called `surfMaterial`
- Defines variables of this type called `surf`, `surf1`, and `surf2`
- Structures can include arrays
- Fields are selected using the period ( . )

Variables and Scoping

- Variables, types, functions must be declared before use
- No default type, everything must be declared with a type
- A variable's scope is determined by where it is declared
- Shared globals are permitted, types must match

Type Qualifiers

- `const`
  - variable is a constant and can only be written during its declaration
- `attribute`
  - per-vertex data values provided to the vertex shader
- `uniform`
  - (relatively) constant data provided by the application or by OpenGL for use in the shader
- `varying`
  - a perspective-correct interpolated value
  - output for vertex shader
  - input for fragment shader
- `in`
  - for function parameters copied into a function, but not copied out
- `out`
  - for function parameters copied out of a function, but not copied in
- `inout`
  - for function parameters copied into and out of a function
**Constants**

- Named constants are declared using the `const` qualifier, e.g.:
  - `const float epsilon = 0.0001;`
  - `const int loopCount = 8;`
  - `const vec3 position = vec3 (0.0, 0.0, 0.0);`
- `Const` qualifier can only be used by itself or with uniform
- Can be used to qualify local or global variables or function parameters
- Literal constants can be expressed as in C
  - Decimal (e.g., 1023, 4076, 5, 0)
  - Octal (e.g., 0777, 05, 02345)
  - Hexadecimal (e.g., 0xFFF, 0x11, 0xFEE)
  - Floating point (e.g., 1.0, 5839.37, 32.0)
  - Scientific notation (e.g., 0.1e-5, 5.333e6, 1.0E10, 2.1E+3)
- Character and string constants are not supported

**Uniform Variables**

- Input to the vertex processor
- Data provided by the application that changes every vertex
- Available as read-only in a vertex shader
- Can be a standard OpenGL vertex attribute
  - `gl_Colour`, `gl_Normal`, `gl_Vertex`, `gl_TexCoord`, etc.
- Can be user-defined
  - Temperature, weighting factor, glossiness, refraction factor, etc.
- API is provided to tie generic vertex attributes supplied by an application to attribute names in a shader
- Specification of vertex position causes execution of the vertex shader
- Can only be used as a qualifier for float, vec2/3/4, and mat2/3/4
- Global variables only
  - `attribute vec3 tangent;`
  - `attribute float density;`
  - `attribute vec3 binormal;`
- Can only be used to qualify global variables

**Varying Variables**

- Output from vertex processor
  - Can be read or written
- Input for fragment processor
  - Read-only
- Global variables only
  - Names/types must match or a link error will occur
  - Used to specify values that are interpolated across a primitive
- Can be standard OpenGL values
  - `gl_FrontColor`, `gl_TexCoord[0]`, `gl_TexCoord[1]`, etc.
- Can be user-defined values
  - `normal`, `halfAngle`, `thickness`, `modelCoordinate`, etc.
- Varying values are interpolated in a perspective-correct fashion
  - `varying vec3 Normal;`
  - `varying vec3 EyeDir;`
  - `varying float LightIntensity;`
Operators

- Same as ANSI C except no:
  - Modulus operator
  - Bit-wise operators
  - Address-of
  - Dereference
  - Type cast
- Operators work as expected on floats, ints, bools
- Operators work component-wise for vectors and matrices
  - Except for * which performs matrix multiplication
- Only assignment (=), equality (==, !), and field selection (.) operators work with structures
- Only array subscript operator ([ ]) works on arrays

Constructors

- Function call syntax is used to make a value of a desired type
- “Parameters” are used to initialize the constructed value
- Can be used to:
  - Do data type conversion
  - Build a larger type out of several smaller types
  - Reduce the size of a larger type
  - Do swizzling of components
- All lexically correct parameter lists are valid
- Parameter list must be of sufficient size and correct type
- Parameters are assigned to the constructed value from left to right

Scalar Constructors

- Some scalar constructor examples:

```cpp
int(bool) // converts a Boolean value to an int
int(float) // converts a float value to an int
float(bool) // converts a Boolean value to a float
float(int) // converts an integer value to a float
bool(float) // converts a float value to a Boolean
bool(int) // converts an integer value to a Boolean
float(vec3) // selects first component of the vector
```

- From float to int, fractional part is dropped
- From int or float to bool, 0 and 0.0 are converted to false, other values are converted to true
- From bool to int or float, false is converted to 0 or 0.0, true to 1 or 1.0

Vector Constructors

- A single scalar parameter will initialize all components of a vector
- Vector constructor examples:

```cpp
vec3(float)
vec4(ivec4)
vec2(float, float)
ivec3(int, int, int)
bvec4(int, int, float, float)
vec2(vec3)
vec3(vec4)
vec3(vec2, float)
vec3(float, vec2)
vec4(vec3, float)
vec4(float, vec3)
vec4(vec2, vec2)
```

- Usage:

```cpp
vec4 color = vec4(0.0, 1.0, 0.0, 1.0);
vec4 rgba = vec4(1.0);
vec3 rgb = vec3(color);
```
Matrix Constructors

- A single scalar parameter is used to initialize all components on the diagonal of the matrix, others are set to 0.0
- Matrices are constructed in column major order
- Examples:
  - mat2(float)
  - mat3(float)
  - mat4(float)
  - mat2(vec2, vec2);
  - mat3(vec3, vec3, vec3);
  - mat4(vec4, vec4, vec4, vec4);
  - mat2(float, float, float, float);
  - mat3(float, float, float, float, float, float);
  - mat4(float, float, float, float, float, float, float, float);

Structure Constructors

- Constructor for a structure is available once structure is defined
- Example:
  ```
  struct light {
    float intensity;
    vec3 position;
  }
  
  light newLight = light(3.0, vec3(1.0, 2.0, 3.0));
  ```

Vector Components

- Vector components can be referred to using array syntax or a single letter:
  - [0], [1], [2], [3]
  - r, g, b, a
  - x, y, z, w
  - s, t, p, q
- This syntax can be used to extract, duplicate, or swizzle components
  ```
  vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
  vec4 swiz = pos.wzyx;  // swiz = (4.0, 3.0, 2.0, 1.0)
  vec4 dup = pos.xxyy;   // dup = (1.0, 1.0, 2.0, 2.0)
  
  pos.xw = vec2(5.0, 6.0); // pos = (5.0, 2.0, 3.0, 6.0)
  pos.wx = vec2(7.0, 8.0); // pos = (8.0, 2.0, 3.0, 7.0)
  pos.xx = vec2(3.0, 4.0); // illegal -- 'x' used twice
  ```

Matrix Components

- Matrix components can be accessed using array subscripting syntax
- A single subscript selects a single column
- A second subscript selects a component within a column
- Examples:
  ```
  mat4 m;
  m[1] = vec4(2.0);  // sets the second column to all 2.0
  m[0][0] = 1.0;     // sets the upper left element to 1.0
  m[2][3] = 2.0;     // sets the 4th element of the third column to 2.0
  ```
Expressions

- Constants
- Constructors
- Variables
- Component field selectors
- Subscripted array names
- Scalar/vector/matrix operations as expected
  - +, -, *, /
- Ternary selection operation (?:)
- User-defined functions
- Built-in functions

Function Definitions

- Function names can be overloaded
  - Argument lists must differ
- Functions must be declared or defined before being called
- Must have a basic type as a return value
  - Can be void
- Arguments can be a basic type, arrays, or structures
- Return type can be a structure, but not an array
- A valid shader must have a function called `main`
- Recursion behavior is undefined

Function Calling Conventions

- Functions are called by value-return
- Arguments can include an optional qualifier
  - in – for function parameters copied into a function, but not copied out
  - out – for function parameters copied out of a function, but not copied in
  - inout – for function parameters copied into and out of a function
  - const – for function parameters that are constants
  - If no qualifier is specified, in is assumed

Function Examples

- Declaration
  
  ```
  vec3 computeColor (in vec3 c1, in vec3 c2);
  float radians (float degrees);
  ```

- Definition
  
  ```
  float myFunc (in float f1, // f1 is copied in
                 inout float f2) // f2 is copied in and out
  {
    float myResult;
    // do computations
    return myResult;
  }
  ```
### Conditional Statements
- if and if-else are supported
- if expression must be type bool
- Can be nested
- Examples:

```c
if (diffuse > 0.1)
    color1 = daytimeColor;

if (z < GrainThreshold)
    color += LightWood * LightGrains * noisevec[2];
else
    color -= LightWood * DarkGrains * noisevec[2];
```

### Iteration Statements
- for, while, and do-while loops are supported as in ANSI C
- Loops can be nested
- Examples:

```c
for (i = 0; i < 8; i++)
    color += contribution[i];

while (i > 0)
    color += contribution[--i];

do
    total += lightContrib[i--];
while (i > 0);
```

### Jump Statements
- continue, break, and return are supported as in ANSI C
- return can return an expression
- discard can be used in a fragment shader to abandon the operation on the current fragment
- Examples:

```c
return (color1 + color2 + color3);

if (intensity < 0.0)
    discard;
```

### Vertex Shader Built-in Variables
- The following special variables are available in a vertex shader:
  - `vec4 gl_Position; // must be written to`
  - `float gl_PointSize; // may be written to`
  - `vec4 gl_ClipVertex; // may be written to`
- Every execution of a vertex shader must write the homogeneous vertex position into gl_Position
  - Can use the built-in function `frtransform()` to achieve invariance with fixed functionality
- Vertex shaders may write the size of points to be rasterized (measured in pixels) into the built-in variable gl_PointSize
- Vertex shaders may write the transformed coordinate to be used in conjunction with user clipping planes into gl_ClipVertex
Vertex Shader Built-in Attributes

- The following are available from a vertex shader for accessing standard OpenGL vertex attributes:
  - `attribute vec4 gl_Color;`
  - `attribute vec4 gl_SecondaryColor;`
  - `attribute vec3 gl_Normal;`
  - `attribute vec4 gl_Vertex;`
  - `attribute vec4 gl_MultiTexCoord0;`
  - `attribute vec4 gl_MultiTexCoord1;`
  - `attribute vec4 gl_MultiTexCoordN-1;`
  - `attribute float gl_FogCoord;`

Built-in Constants

- The following built-in constants are defined:
  - `gl_MaxLights = 8`
  - `gl_MaxClipPlanes = 6`
  - `gl_MaxTextureUnits = 2`
  - `gl_MaxTextureCoords = 2`
  - `gl_MaxVertexAttribs = 16`
  - `gl_MaxVertexUniformComponents = 512`
  - `gl_MaxVaryingFloats = 32`
  - `gl_MaxVertexTextureImageUnits = 0`
  - `gl_MaxTextureImageUnits = 2`
  - `gl_MaxFragmentUniformComponents = 64`
  - `gl_MaxCombinedTextureImageUnits = 2`

- Can be used within a shader
- Have the same value as queriable values of the same name in OpenGL

State-Tracking

- Existing OpenGL state is available to shaders
  - Uniform variables with reserved prefix “gl_” are used to automatically track OpenGL 1.5 state
- Variables can be used by shaders to access current OpenGL state
  - These are built-in uniform variables so do not need to be declared or included
- State tracking will occur for all such variables that are used in a shader
- Examples:
  - `gl_ModelViewMatrix`
  - `gl_ModelViewProjectionMatrix`
  - `gl_LightSource[gl_MaxLights]`
  - `gl_Fog`
  - `gl_ClipPlane[gl_MaxClipPlanes]`

Built-in Varying Variables

- Available to be written in a vertex shader:
  - `varying vec4 gl_FrontColor;`
  - `varying vec4 gl_BackColor;`
  - `varying vec4 gl_FrontSecondaryColor;`
  - `varying vec4 gl_BackSecondaryColor;`
  - `varying vec4 gl_TexCoord[gl_MaxTextureCoords];`
  - `varying float gl_FogCoord;`
- Available to be read in a fragment shader
  - `varying vec4 gl_Color;`
  - `varying vec4 gl_SecondaryColor;`
  - `varying vec4 gl_TexCoord[gl_MaxTextureCoords];`
  - `varying float gl_FogCoord;`
- Can be used to interface to the fixed functionality of OpenGL
Fragment Shader Built-in Variables

- The following special variables are available as read-only in a fragment shader:
  - `vec4 gl_FragCoord;` // window relative coords
  - `bool gl_FrontFacing;` // is primitive frontfacing?

- The following special variables are available for writing in a fragment shader:
  - `vec4 gl_FragColor;` // final color value
  - `float gl_FragDepth;` // final depth value
  - `vec4 gl_FragData[n];` // arbitrary data

  gl_FragCoord and gl_FrontFacing contain values computed by fixed functionality in between the vertex processor and the fragment processor.

  gl_FragColor and gl_FragDepth should be written with the values to be used by the back end of the processing pipeline.

  If gl_FragDepth is not written, the depth value computed by fixed functionality will be used as the depth.

  gl_FragData[n] can be used to write arbitrary data to multiple render targets.

Built-in Functions

- **Trigonometry/angle**
  - radians, degrees, sin, cos, tan, asin, acos, atan

- **Exponential**
  - pow, exp2, log2, sqrt, inversesqrt

- **Common**
  - abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, smoothstep

- **Geometric and matrix**
  - length, distance, dot, cross, normalize, ftransform, faceforward, reflect, matrixCompMult

- **Vector relational**
  - lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, any, all

- **Texture lookup**
  - texture1D/2D/3D, texture1D/2D/3DProj, textureCube, texture1D/2DShadow, texture1D/2DShadowProj

- **Fragment shader only**
  - dFdx, dFdy, fwidth

- **Noise**
  - noise1/2/3/4

Preprocessor

- Preprocessor processes strings before they are compiled
- Support for all ANSI C preprocessor directives except file-based ones
  - e.g., #include
- Predefined macros `__LINE__`, `__FILE__`, `__VERSION__`
- No number sign operators or sizeof
- Two pragmas are defined:
  - Turn optimization on and off
  - Turn debugging on and off