GLSL: Language - Overview

- GLSL was designed to be portable, following the philosophy that motivated the design of OpenGL
- GLSL based on C/C++
  - Control structures are the same:
    if, *switch*, *for*, *while*, *do-while*, *continue*, *break*, *return*
- A number of new data types have been added to enhance graphics processing (e.g., vector, matrix)
- A number of data types have been dropped, being of no relevance to graphics processing (e.g., char, string)
- Operations associated with the above data types have been either added or dropped (as appropriate)
- Additional restrictions:
  1. No pointers
  2. No *sizeof()* function
  3. No *enum* types
  4. Only 1D arrays allowed
  5. Type conversions more restricted
  6. New functions for graphics processing have been included (e.g., dot products, reflection calculations)
  7. New parameter types have been introduced
- NOTE: If mention of a language feature does not appear in the following, it is the same as in C/C++ (unless it was an oversight)

GLSL: Language - Scalar Data Types

- Types:
  1. *int*
  2. *uint*
  3. *float*
  4. *double*
  5. *bool*
- Implicit type conversions made as follows:

lvalue type	rvalue type implicitly converted from
uint	int
float	int, uint
double	int, uint, float

- Type casts are performed using *constructors*
  - Syntax like a function call:
    type(arg)
  - For booleans,
    - Non-zero values converted to $true$
    - Zero values converted to $false$

GLSL: Language - Vector Data Types

- Vector types:
  - [{ibud}]vec$n$, where $2 \leq n \leq 4$
  - They are represented internally as 1D arrays with 2, 3, or 4 arguments
  - While this type of data could be represented as a regular 1D array, the built-in vector operations (see below) are not applicable to the standard array types
- Vector elements can be accessed in the same way as array elements
- Similar to Java, vectors have an implicit *length()* function
  - Accessed using *dot* notation
  - Returns the number of elements in the vector

GLSL: Language - Vector Name Sets

- *Name sets* are a notational convenience for accessing vector components
- There are three name sets:
  1. $xyzw$
  2. $rgba$
  3. $stpq$
- These correspond to spatial coordinates, colors, and texture coordinates, which are the primary data that vector types are used to represent
- Note that the standard $strq$ is being represented as $stpq$ to avoid ambiguity wrt r
- Each letter of a name space corresponds to a positional index into the vector
- To use name space access of a vector, use standard dot notation
  E.g.,

```
v.rgba
v.rgb
v.xyzw
v.xyz
v.xz
```

- The name set is independent of the data that a vector represents
- You cannot mix and match letters from different name sets

GLSL: Language - Vector Constructors

- All initialization is done with constructors
  - NOTE: This applies to *all* aggregate types (e.g., arrays, matrices) - not just vectors
- Syntax: [t]vec*n*(arg [, arg]...)
- There are several flavors:
  1. Single scalar argument
     - All elements of the vector are initialized to the single value
       E.g.,
       ```
       ivec4 v = ivec4(8);
       ```
  2. Multiple scalar arguments
     - Each element is initialized to the respective argument
       E.g.,
       ```
       ivec4 v = ivec4(8, 4, 6, 10);
       ```
  3. Multiple arguments / different structural types
     - Multiple types may be supplied
     - All arguments must supply data to the vector
       * There may not be unused arguments
     - Values are assigned to the vector by processing the arguments from left to right
     - Unused values are ignored (not components - every component must be used, at least in part)
       E.g.,
       ```
       ivec3 v = ivec3(int, ivec2);
         //v[0] = int, v[1] = ivec2[0], v[2] = ivec2[1]
       ivec3 v = ivec3(ivec2, int);
         //v[0] = ivec2[0], v[1] = ivec2[1], v[2] = int
       ivec3 v = ivec3(ivec4);
         //v[0] = ivec4[0], v[1] = ivec4[1], v[2] = ivec4[2]
       ivec3 v = ivec3(int, ivec4);
         //v[0] = int, v[1] = ivec4[0], v[2] = ivec4[1]
       ```

GLSL: Language - Vector Operators and Functions

1. Extensions of C operators
   - Bitwise operators are applied component-wise
   - Math operators +, -, *, / are applied component-wise when
     - One argument is a scalar
     - Both arguments are vectors
   - Relational operators
     - Are not valid for vectors, *but* can be applied to arrays
2. Existing C functions
   - Many of C's functions have been extended to work with vectors
   - They are applied component-wise to the vector's elements
   - These functions include (See Appendix C pp 686 - 692)
     (a) Standard trig functions
     (b) Standard trig inverse functions
     (c) Standard trig hyperbolic functions
     (d) Conversion functions between degrees and radians
     (e) Exponentiation, log, square root, ...
     (f) Numeric functions like *sqrt*, *abs*, ...
   - Swizzle
     - This operation reorders a vector's elements
     - It uses the name space notation
     - The vector returned has its elements in the order specified by the swizzle
       E.g.,

```
ivec4 v1, v2, v3;
v1 = ivec4(1, 2, 3, 4);
v2 = v1.rrbb;
v3 = v1.wzyx;
```

GLSL: Language - Vector Operators and Functions (2)

3. New functions (Appendix C pp 692 - 698)

(a) Numeric

Syntax	Description
genType sign (genType x)	Returns 1.0 if $x > 0$, 0.0 if $x = 0$, and -1.0 if $x < 0$
genType fract (genType x)	Returns $x - floor(x)$
genType mod (genType x, genType y)	Returns $x - y * floor(x/y)$
genType min (genType x, genType y)	Returns y if $y < x$; otherwise x
genType max (genType x, genType y)	Returns y if $x < y$; otherwise x
genType clamp (genType x, genType minVal, genType maxVal)	Returns $min(max(x, minVal), maxVal)$
genType mix (genType x, genType y, genType a)	Returns $x * (1.0 - a) + y * a$
genType step (genType edge, genType x)	Returns 0 if $x \leq edge$, otherwise 1.0
genType smoothstep (genType edge0, genType edge1, genType x)	Returns 0 if $x \leq edge0$ and 1.0 if $x \geq edge1$, otherwise smooth Hermite interpolation

GLSL: Language - Vector Operators and Functions (3)

(b) Geometric (Appendix C pp 700 - 702)

Syntax	Description
float length (genType x) double length (genDType x)	Returns the length of vector x
float dot (genType x, genType y) double dot (genDType x, genDType y)	Returns the dot product of x and y
genType cross (genType x, genType y) genDType cross (genDType x, genDType y)	Returns the cross product of x and y
genType normalize (genType x) genDType normalize (genDType x)	Returns a vector in the same direction as x but with a length of 1
[d]mat*m* outerProduct (vec*m* c, vec*m* r) [d]mat*m*x*n* outerProduct (*t*vec*n* c, *t*vec*m* r)	Treats the first parameter c as a column vector and the second parameter r as a row vector and does a linear algebraic matrix multiply $c * r$, yielding a matrix whose number of rows is the number of components in c and whose number of columns is the number of components in r
genType faceforward (genType N, genType I, genType Nref) genDType faceforward (genDType N, genDType I, genDType Nref)	If $dot(Nref, I) < 0$ return N, otherwise return $-N$
genType reflect (genType I, genType N) genDType reflect (genDType I, genDType N)	For the incident vector I and surface orientation N, returns the reflection direction: $I - 2 * dot(N, I) * N$ N must already be normalized in order to achieve the desired result
genType refract (genType I, genType N, float eta) genDType refract (genDType I, genDType N, float eta)	For the incident vector I and surface normal N, and the ratio of indices of refraction eta, return the refraction vector. The result is computed by $k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I))$ if $(k < 0.0)$ return genType(0.0) // or genDType(0.0) else return $eta * I - (eta * dot(N, I) + sqrt(k)) * N$ The input parameters for the incident vector I and the surface normal N must already be normalized to get the desired results

(c) Relational (Appendix C pp 703 - 705)

- The following table indicates the legal data types for return and argument types

Placeholder	Specific types allowed
bvec	bvec2, bvec3, bvec4
ivec	ivec2, ivec3, ivec4
uvec	uvec2, uvec3, uvec4
vec	vec2, vec3, vec4, dvec2, dvec3, dvec4

GLSL: Language - Vector Operators and Functions (5)

- The sizes of all the input and return vectors for any particular call must match

Syntax	Description
bvec lessThan (vec x, vec y) bvec lessThan (ivec x, ivec y) bvec lessThan (uvec x, uvec y)	Returns the component-wise compare of x < y
bvec lessThanEqual (vec x, vec y) bvec lessThanEqual (ivec x, ivec y) bvec lessThanEqual (uvec x, uvec y)	Returns the component-wise compare of x ≤ y
bvec greaterThan (vec x, vec y) bvec greaterThan (ivec x, ivec y) bvec greaterThan (uvec x, uvec y)	Returns the component-wise compare of x > y
bvec greaterThanEqual (vec x, vec y) bvec greaterThanEqual (ivec x, ivec y) bvec greaterThanEqual (uvec x, uvec y)	Returns the component-wise compare of x ≥ y
bvec equal (vec x, vec y) bvec equal (ivec x, ivec y) bvec equal (uvec x, uvec y) bvec equal (bvec x, bvec y)	Returns the component-wise compare of x == y
bvec notEqual (vec x, vec y) bvec notEqual (ivec x, ivec y) bvec notEqual (uvec x, uvec y) bvec notEqual (bvec x, bvec y)	Returns the component-wise compare of x ! = y
bool any (bvec x)	Returns true if any component of x is true
bool all (bvec x)	Returns true only if all components of x are true
bvec not (bvec x)	Returns the component-wise logical complement of x

GLSL: Language - Matrix Data Types

- Matrices only support floats
- There are two general types
  1. Square matrices
     - Syntax: [d]mat$n$, where $2 \leq n \leq 4$
  2. General matrices
     - Syntax: [d]mat$m$xn, where $2 \leq m, n \leq 4$
  3. Note that 1X1, mX1, and 1Xn matrices do not exist
     - 1X1 is just a single float
     - mX1 and 1Xn are superfluous - they can be represented as a vec$n$ data type
- Matrices also have implicit *length()* function
  - returns the number of *columns*

GLSL: Language - Matrix Constructors

- These work similarly to vector constructors
- As is standard in OpenGL, matrices are stored in column-major order
  - When values from a constructor are added to a matrix, they are added column-by-column
- Syntax: [d]mat$n$(arg [, arg]...)
- There are several flavors:
  1. Single scalar argument
     - Fills the diagonal of a square matrix E.g.,

```
mat3 m = mat3(f);
```

$$\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & f \end{bmatrix}$$

  2. Multiple scalar arguments
     - Each element is initialized to the respective argument
       E.g.,

```
mat2 m = mat2(f1, f2, f3, f4);
```

$$\begin{bmatrix} f1 & f3 \\ f2 & f4 \end{bmatrix}$$

  3. Single matrix argument
     - The upper left quadrant of the argument fills the upper left quadrant of the matrix
     - Remaining part of diagonal receives ones E.g.,

```
//mat4 m4 = (f1, f2, ..., f16);
mat2 m2 = mat2(m4);
```

$$\begin{bmatrix} f1 & f5 \\ f2 & f6 \end{bmatrix}$$

```
//mat2 m2 = (f1, f2, f3, f4);
mat4 m4 = mat4(m2);
```

$$\begin{bmatrix} f1 & f3 & 0 & 0 \\ f2 & f4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

GLSL: Language - Matrix Constructors (2)

4. Multiple arguments / different structural types
   - Works the same as with vectors
   - Note: A matrix argument cannot be used with any other arguments E.g.,

```
vec2 v = vec2(f1, f2);
vec3 u = vec3(f3, f4, f5);
float f6, f7, f8, f9;
mat3 m = mat3(v, u, f6, f7, f8, f9);
```

$$\begin{bmatrix} f1 & f4 & f7 \\ f2 & f5 & f8 \\ f3 & f6 & f9 \end{bmatrix}$$

GLSL: Language - Matrix Operators and Functions

1. Extensions of C operators
   - Bitwise operators and math operators +, -, / are applied component-wise when
     - One argument is a scalar
     - Both arguments are matrices
   - Math operator *
     - Applied component-wise when one argument is a scalar
     - When both arguments are matrices or one is a vector
       * Standard matrix multiplication is performed
       * Compatible dimensions are assumed in all cases
       (a) If v is $1 \times d$ and m is $d \times e$, the result of $v * m$ is a $1 \times e$ vector
       (b) If v is $1 \times d$ and m is $e \times d$, the result of $m * v$ is a $1 \times e$ vector
       (c) If m is $d \times e$ and n is $e \times f$, the result of $m * n$ is a $d \times f$ matrix
2. Existing C functions
   - Work the same as with vectors
3. New functions

   (a) Geometric

Syntax	Description
mat matrixCompMult (mat x, mat y)	Multiply matrix x by matrix y component-wise, i.e., result[i][j] is the scalar product of x[i][j] and y[i][j]
[d]mat$m$ transpose ([d]mat$n$ m) [d]mat$m$xn transpose ([d]mat$n$xm n)	Returns a matrix that is the transpose of n. The input matrix n is not modified
type determinant ([d]mat$m$ n)	Returns the determinant of n
[d]mat$m$ inverse ([d]mat$m$ n)	Returns a matrix that is the inverse of n. The input matrix n is not modified. The values in the returned matrix are undefined if n is singular or poorly conditioned (nearly singular)

GLSL: Language - Precedence

Precedence	Operators	Accepted Types	Description
1	()	all	Grouping
2	[ ]	arrays, matrices, vectors	Subscripting
	f()	functions	Function calls, constructors
	.	structs	Struct component access
	++ −−	arithmetic	post inc/dec
3	++ −−	arithmetic	pret inc/dec
	+ −	arithmetic	pre
	∼	integer	Unary bitwise not
	!	bool	Unary logical not
4	* / %	arithmetic	
5	+ −	arithmetic	infix
6	<< >>	integer	Bitwise ops
7	< > <= >=	arithmetic	relops
8	== !=	any	eqops
9	&	integer	Bitwise and
10	∧	integer	Bitwise exclusive or
11	|	integer	Bitwise inclusive or
12	&&	bool	Logical and
13	^^	bool	Logical exclusive or
14	||	bool	Logical or
15	a ? b : c	bool ? any : any	Ternary selection
16	=	any	Equality
	+= −=	arithmetic	Arith assignment
	*= /=	arithmetic	
	%= <<= >>=	arithmetic	
	&= ^= |=	arithmetic	
17	,	any	Sequence

GLSL: Language - Storage Qualifiers

- GLSL includes a number of modifiers that restrict the use of variables
  - Most are available only in OpenGL2.1

Modifier	Semantics
const	The variable value cannot be changed.
in	Receive data from the previous shader in the pipeline. In vertex shader, the values are set by the OpenGL API. Only applied to *float*, *int*, *vec*, *mat* types, or arrays of these types. Read only. Must be globally declared.
out	Send data on to the next shader in the pipeline. In fragment shader, only applied to *float* or *int* scalars, *vec*, or arrays of these types. Write only. There must be an *in* variable in a shader for every *out* variable in the preceding shader in the pipeline. They must agree in name, data type, etc.
uniform	Define global variables whose values are constant (across a graphics primitive) Can be of any built-in data type (including arrays and structs) Accessible to all shaders. Must be globally declared.

GLSL: Language - Functions

- Same as C
- The following are new parameter qualifiers:

Syntax	Description
in	The argument value is copied to the formal parameter. Any changes made to the formal by the function are lost. The argument may be a general expression.
const in	The argument value is copied to the formal parameter, but cannot be changed.
out	The value of the formal parameter must be set by the function. The argument has no value on entry to the function. The value is copied to the argument on the function's return. The argument must be an lvalue.
inout	The argument value is copied to the formal parameter. Changes may be made to the formal. The final value is copied to the argument on the function's return. The argument must be an lvalue.

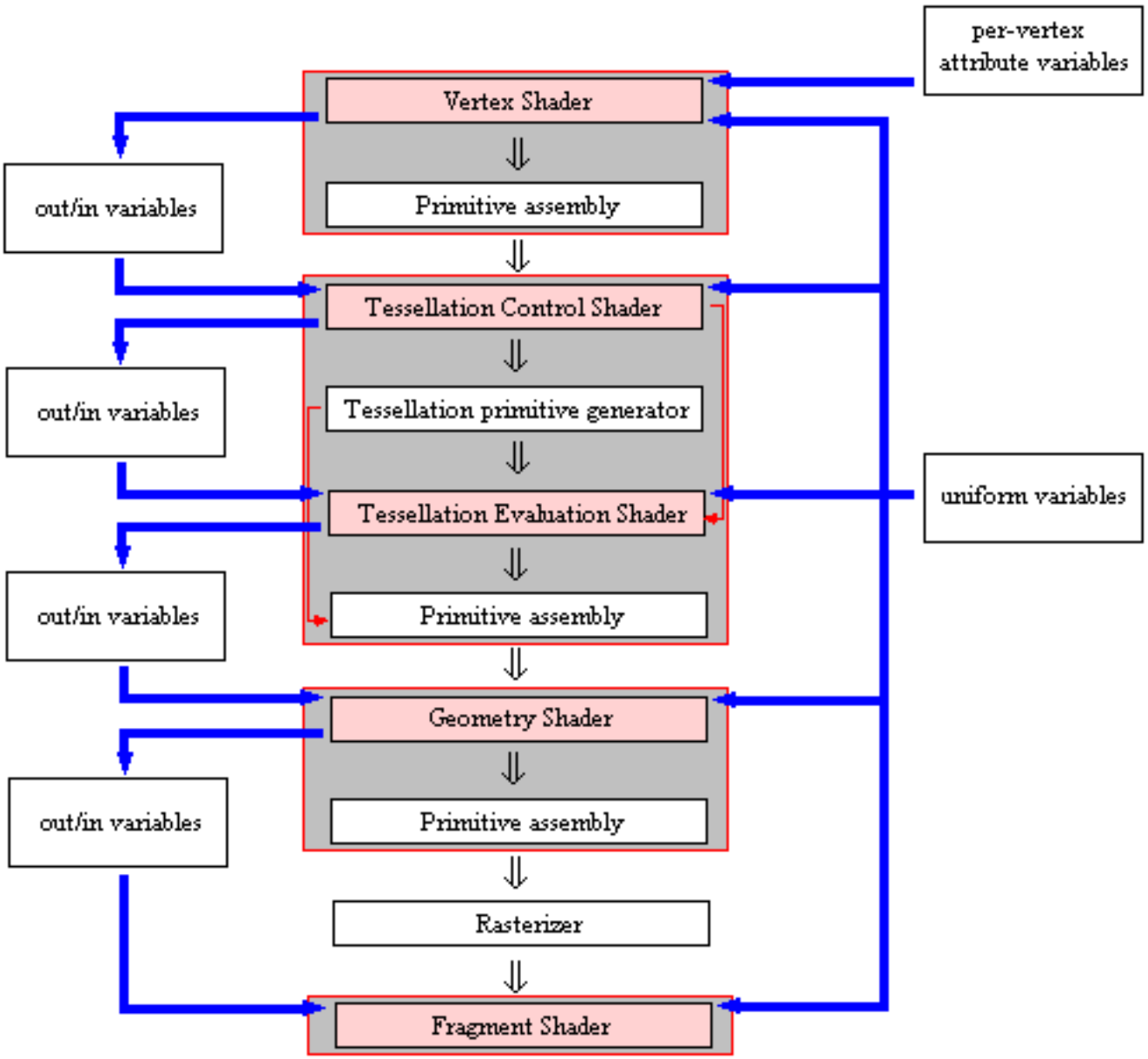
GLSL: Language - Variable Naming Conventions

- The following naming conventions are generally used:

Prefix	Origin
a	application (attribute variables)
u	application (uniform variables)
v	vertex shader
tc	tessellation control shader
te	tessellation evaluation shader
g	geometry shader
f	fragment shader

GLSL: Language - Data Flow

- The following diagram represents the flow of data among the shaders:



GLSL: Language - Data Flow (2)

- Variable types:
  1. Attribute variables
     - Specified in the application program
     - Specified per-vertex
     - Input only to the vertex shader
  2. Uniform variables
     - Specified in the application program
     - Accessible to all shaders
     - Describe large-scale characteristics (not per-vertex)
     - Read-only wrt shaders
  3. In variables
     - Are read into a shader
     - Read-only wrt shader
     - Values originate in previous shader in the pipeline
  4. Out variables
     - Are output by a shader
     - Write-only wrt shader
     - Can be result of a computation performed by the shader
     - Can simply be an In variable passed through from the previous shader in the pipeline to the next shader in the pipeline
     - Out variables of a shader are the In variables of the next shader in the pipeline
- Keep in mind that in the above, "*next/previous shader in the pipeline*" refers to only those that have been actually implemented