GLSL versions

- WebGL 1.0 requires GLSL 1.00
- WebGL 2.0 can use GLSL 1.00 or GLSL 3.00
  - some features require GLSL 3.00
- must use the same version for both vertex and fragment shader within a shader program
- book addresses both GLSL 1.00 and 3.00
  - we'll only cover GLSL 1.00
- refer to the book (section 6.3) for more info about GLSL

GLSL – Types

- variables are declared with a type
  - scalar types – `float`, `int`, `bool`
  - vector types – `vec2`, `vec3`, `vec4` (of `floats`)
    `ivec2`, `ivec3`, `ivec4` (of `ints`)
    `bvec2`, `bvec3`, `bvec4` (of `bools`)
  - matrix types – `mat2`, `mat3`, `mat4` (of `floats`)
  - structs
  - arrays

note that GLSL `vec` and `mat` types are distinct from the JavaScript types provided by `glmatrix` even though the names are the same

GLSL – Vector Types

- vectors – `vec2`, `vec3`, `vec4` and the forms for other types
  - can use array notation or several forms of dot notation
    - v[0], v[1], v[2], v[3] - use when vector represents a 1D array
    - v.x, v.y, v.z, v.w - use when vector represents a point or vector
    - v.r, v.g, v.b, v.a - use when vector represents a color
    - v.s, v.t, v.p, v.q - use when vector represents texture coordinates
  - constructor `veci(…)` takes a list of expressions which together provide the correct number of values

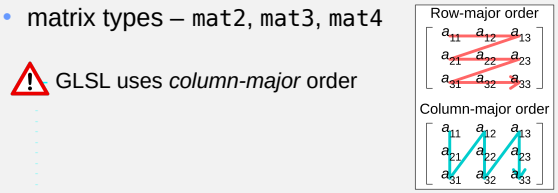
```
vec2 v = vec2( 1.0, 2.0 );
vec4 w = vec4( v, v );  // w is ( 1.0, 2.0, 1.0, 2.0 )
```

```
vec3 v = vec3( 1.0, 2.0, 3.0 );
vec3 w = vec3( v.zx, 4.0 );  // w is ( 3.0, 1.0, 4.0 )
```

    - extra values are dropped, but it isn't legal to include extra expressions where all of their values are dropped
    - special case: `veci(value)` creates a vector with all entries equal to `value`
    - supports automatic type conversion
      - for bool/int: true → 1, false → 0; 0 → false, non-zero → true
  - *swizzlers* allow convenient access and recombination of components
    - e.g. `v.xy`, `v.rrg`
    - can use swizzlers on the left side of an assignment if there are no repeated components

GLSL – Matrix Types

- matrix types – `mat2`, `mat3`, `mat4`



⚠ GLSL uses *column-major* order

  - use array notation
    - for `mat3 m`, `m[2][1]` is an element, `m[2]` is a column
  - constructor `mati(…)` takes a list of expressions which together provide the correct number of values
    - special case: `mati(value)` creates a matrix with all diagonal entries equal to `value` and other entries 0
      - e.g. `mat2(1)`, `mat3(1)`, `mat4(1)` yield identity matrices

GLSL – Structs

```
struct LightProperties {
  vec4 position;
  vec3 color;
  float intensity;
};
```

- structs
  - a collection of named fields, or a class with only public instance variables and not methods
  - defines a type which can be used to declare variables
    - e.g. `LightProperties light;`
  - fields are accessed by varname.field
    - e.g. `light.position`, `light.color`, `light.intensity`
  - construct with a list of values with exact types in the same order as declared
    - e.g. `light = LightProperties(vec4(x,y,z,1),vec3(r,g,b),1.0);`
    - no type conversion

GLSL – Arrays

- arrays
  - 1D only
  - base type can be a basic type or struct type
  - size is specified in the variable declaration
    - must be an integer constant
    - C-style – goes after the variable name rather than the type
    - e.g. `int A[10]`
  - use array notation
    - index expression can only contain integer constants and `for` loop variables (with one exception)
  - no bounds checking

GLSL Qualifiers

- storage qualifiers – `uniform`, `const` `attribute`, `varying`
  - `const` value cannot be changed after initialization
  - `uniform`
    - can be used by both vertex and fragment shaders
    - can use the same variable in both shaders (types must be the same)
    - can be any type, including array and struct
      - arrays and structs aren't supported directly by JavaScript – must treat each as a separate uniform value
    - use JavaScript `gl.uniformMatrixNfv()` for `matN`
      - 1D array in column major order
      - 2nd parameter must be `false` (can be true in GLSL 3.00 to indicate row-major order)

```
transformLoc = gl.getUniformLocation(prog, "transform");
gl.uniformMatrix3fv( transformLoc, false, [ 1,0,0, 0,1,0, 0,0,1 ] );
```


GLSL Qualifiers

- `attribute`
  - can only be used in vertex shaders
  - only for types `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, `mat4`
    - matrix attributes aren't supported directly by JavaScript so they are treated as a set of vector attributes (one per column)
- `varying`
  - only for types `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, `mat4` and arrays of those types
  - should be declared in both vertex and fragment shaders (must have the same type in both)
  - read-only in the fragment shader
  - should be written (and can be read) in vertex shader

GLSL Qualifiers

- precision qualifiers – `highp`, `mediump`, `lowp`
  - specifies the range and precision of values used by the shaders
  - GLSL specifies minimum requirements
  - defaults
    - vertex shader – `highp` for `int`s and `float`s
    - fragment shader – `mediump` for `int`s
  - must specify for `float`s in fragment shader
    - can set for individual variables
      - e.g. `varying highp float v;`
    - can set for all `float` variables
      - e.g. `precision mediump float;`
    - many fragment shaders support highp even though it is not required
      - book gives code to use `highp` if supported and `mediump` otherwise

GLSL Qualifiers

- `invariant`
  - requires that exactly the same value be used when the same expression occurs in different places
    - e.g. for a multipass algorithm where several shaders are invoked in turn – want to ensure that the `gl_Position` computed is the same in all cases
  - only applies to varying variables or predefined variables (e.g. `gl_Position`, `gl_FragCoord`)

GLSL Expressions

- operators +, -, *, /, ++, -- for `float` and `int`
  - no automatic type conversion (not even `int` → `float`)
- overloaded operators
  - mat*mat, mat*vec to multiply matrices and vectors
  - vec+scalar, vec-scalar, vec*scalar, vec/scalar applies the operation to each component of the vector
  - vec+vec, vec-vec, vec*vec, vec/vec applies the operation to each pair of components
- relational operators
  - <, >, <=, >= only apply to `int`, `float`
  - ==, != work with all built-in types except sampler types
    - vectors and matrices are only equal if all components are equal
- boolean operators – !, &&, ||, ^^ (XOR)
- =, +=, -=, *=, /= work as usual

GLSL Functions

require `float` params

- vector functions
  - `dot(u,v)`
  - `cross(u,v)`
  - `length(v)`
  - `distance(p,q)`
  - `normalize(v)`
- utility functions
  - `mix(x,y,t)`
    - $0 \le t \le 1$
    - computes x*(1-t)+y*t
  - `clamp(x,low,high)`
    - returns low if x < low, high if x > high, x otherwise
  - `smoothstep(s,t,x)`
    - s < t
    - returns 0 if x < s, 1 if x > t, interpolated value in range 0..1 otherwise

GLSL Functions

require `float` params

- trig functions – radians
  - `sin`, `cos`, `tan`
  - `asin`, `acos`, `atan`
- math functions
  - `log`, `exp`
  - `pow`, `sqrt`
  - `abs`
  - `floor`, `ceil`
  - `min`, `max`
    - also `min(v,f)`, `max(v,f)` where each element of vector `v` is compared to scalar `f`
  - `mod(x,y)`
    - returns x-y*floor(x/y)

GLSL User-Defined Functions

- must be declared before used
  - declaration can be full definition or prototype
- can be overloaded
- return type cannot include arrays (either directly or as part of a struct)
- array parameters must include size as part of the declaration
- parameters can be `in`, `out`, or `inout`
  - default is `in` (keyword can be omitted)
  - for `out` and `inout` parameters, actual parameter must be something that can be assigned to (variable or swizzler) rather than an expression
- cannot be recursive

GLSL Control Structures

- unless otherwise noted, syntax is C-style (also similar to Java)
- conditionals
  - `if`, including `else` and `else if`
- loops
  - only supports a limited version of `for`
    - loop variable must must be declared in the initialization part of the loop
      - can only be `int` or `float`
      - initial value must be a constant or constant expression (involving only literal constants or constant variables)
    - test condition can only be of the form `var op expr`
    - loop variable can only be updated in the update part of the loop
      - update must be `var++`, `var--`, `var += expr`, `var -= expr` where *expr* is a constant expression
    - can include `break`, `continue`

GLSL Limits

- limits specify what a WebGL implementation is required to provided
  - particular implementations may support more
- book lists, along with how to query the actual limits from JavaScript
  - e.g. number of attribute/uniform variables, textures
  - e.g. viewport, texture image size
  - e.g. line width, point size
- usage
  - for maximum portability, stick within the limits
  - if you need more, check so you can output an error if the device can't support your program