OpenGL® SC 2.0 is a safety-critical subset of OpenGL® ES 2.0, which a software interface to graphics hardware for producing high-quality color images of three-dimensional objects.

- [n.n.n] refers to sections and tables in the OpenGL SC 2.0 specification.
- [n.n.n] refers to sections in the OpenGL ES Shading Language 1.0 specification.
- In descriptions, curly braces { } mean “one of,” and square brackets [ ] mean “optional.”
- Specifications are available at www.khronos.org/registry/glscc/

Khrnos strongly encourages OpenGL SC implementations to also support EGL, but some implementations may provide alternate, platform- or vendor-specific APIs with similar functionality. Specifically, OpenGL SC is defined targeting EGL 1.4 with the EGL EXT-create context robustness and EGL KHR surfaceless context extensions.

OpenGL SC Command Syntax [2.3]

For brevity, the OpenGL documentation and this reference may omit the standard prefixes. The actual names are of the forms: glFunctionName(), GL_CONSTANT, GLtype

OpenGL SC commands are formed from a return type, a name, and optionally a type letter i for 32-bit int, or f for 32-bit float, as shown by the prototype below:

```
return-type-name[1234][if][v]([args ,] type T arg1 , . . . , type T argN [, args]);
```

The argument type T and the number N of arguments may be indicated by the command name suffixes.

N is 1, 2, 3, or 4 if present, or else corresponds to the type letters. If “v” is present, an array of

GL commands are formed from a return type, a name, and optionally a type letter i for 32-bit int, or f for 32-bit float.

### Buffer Objects [2.10]

Buffer objects hold vertex array data or indices in high-performance server memory.

- void GenBuffers(sizet n, uint *buffers);
- void BindBuffer(enum target, uint buffer);

### Creating and Binding Buffer Objects

- void BindBuffer(uint target, uint buffer);
  - target: ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER

### Creating Buffer Object Data Stores

- void BufferData(enum target, sizetptr size, const void *data, enum usage);
  - target: ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER
  - usage: STATIC_DRAW, STREAM_DRAW, DYNAMIC_DRAW

### Buffer Object Data Stores

- void GetBufferParameteri(ihex, enum target, enum pname, uint *params)
- void GetBufferParameteriv(ihex, enum target, enum pname, ihex *params)

### Shader Queries [6.1.5, 6.1.6]

- void GetVertexAttribf(ihex, enum index, enum pname, float *params);
- void GetVertexAttribfv(ihex, enum index, enum pname, float *params);

### Vertex Attributes

- void GetAttribLocation(ihex, enum program, const char *name);
- void GetUniformLocation(ihex, enum program, const char *name);

### Uniform Variables

- void Uniform1f(ihex, ihex, float *params);
- void Uniform1fv(ihex, ihex, float *params);
- void UniformMatrixf(ihex, ihex, ihex, boolean transpose, float *params);

### Program Queries [6.1.8]

- void GetProgramiv(ihex, enum program, enum pname, uint *params);
- void GetProgramf(ihex, enum program, enum pname, float *params);

### Rasterization [3]

#### Points [3.3]

Point size is taken from the shader built-in gl_PointSize and clamped to the implementation-dependent point size range.

#### Line Segments [3.4]

Line Width.

#### Polygons [3.5]

- void PolygonOffset(float factor, float units);

### Viewport and Clipping

#### Controlling the Viewport [2.13.1]

- void DepthRange(float n, float f);
- void Viewport(int x, int y, sizet w, sizet h);

### Pixel Rectangles [3.6, 4.3]

- void PixelStore(enum pname, T param);
  - pname: UNPACK_ALIGNMENT

---

### GL Data Types [2.3]

GL types are not C types.

<table>
<thead>
<tr>
<th>GL Type</th>
<th>Minimum Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
<td>Boolean</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>Signed binary integer</td>
</tr>
<tr>
<td>ubyte</td>
<td>8</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>Characters making up strings</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>Signed 2’s complement binary integer</td>
</tr>
<tr>
<td>ushort</td>
<td>16</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>Signed 2’s complement binary integer</td>
</tr>
<tr>
<td>uint</td>
<td>32</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>sizei</td>
<td>32</td>
<td>Non-negative binary integer size</td>
</tr>
<tr>
<td>enum</td>
<td>32</td>
<td>Enumerated binary integer value</td>
</tr>
<tr>
<td>intptr</td>
<td>ptrbits</td>
<td>Signed 2’s complement binary integer</td>
</tr>
<tr>
<td>sizeiptr</td>
<td>ptrbits</td>
<td>Non-negative binary integer size</td>
</tr>
<tr>
<td>bitfield</td>
<td>32</td>
<td>Bit field</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>Floating-point value</td>
</tr>
</tbody>
</table>

---

### Errors and Status Reset

#### Errors [2.5]

- enum GetError(void); //Returns one of:
  - INVALID_ENUM
  - INVALID_FRAMEBUFFER_OPERATION
  - INVALID_OPERATION
  - OUT_OF_MEMORY
  - CONTEXT_LOST
  - NO_ERROR

#### Graphics Reset Recovery [2.6]

- enum GetGraphicsResetStatus(void); //Returns one of:
  - NO_ERROR
  - GUILTY_CONTEXT_RESET
  - INNOCENT_CONTEXT_RESET
  - UNKNOWN_CONTEXT_RESET

#### Vertices

- **Current Vertex State [2.8]**
  - void VertexAttribPointer(ihex, ihex, enum type, boolean normalized, sizet stride, const void *pointer);
  - type: BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, FLOAT

- **Vertex Arrays [2.9]**
  - void VertexArrayAttribPointer(ihex, ihex, enum type, boolean normalized, sizet stride, const void *pointer);

- **EnableVertexArrayArray**
  - void EnableVertexArrayArray(ihex);

- **DrawArrays**
  - void DrawArrays(enum mode, int first, sizet count);

- **DrawRangeElements**
  - void DrawRangeElements(enum mode, uint start, uint end, enum count, enum type, const void *indices);

#### Rasterization

- void CullFace(enum mode);
  - mode: FRONT, BACK, FRONT_AND_BACK

Enable/Disable
- void PolygoneOffset(float factor, float units);
- Enable/Disable POLYGON_OFFSET_FILL
Per-Fragment Operations

Scissor Test [4.1.2]

Enable/Disable(SCISSOR_TEST)

void Scissor(int left, int bottom, sizei width, sizei height);

Multisample Operations [4.1.3]

Enable/Disable(cap)
cap: SAMPLE_ALPHA_TO_COVERAGE, SAMPLE_COVERAGE

void SampleCoverage(float value, boolean invert);

Stencil Test [4.1.4]

Enable/Disable(STERICAL_TEST)

void StencilFunc(enum func, int ref, uint mask);

void StencilFuncSeparate(enum func face, enum func ref, uint mask);

void StencilOp(uint face, enum func, enum func, enum func);

Whole Framebuffer Operations

Fine Control of Buffer Updates [4.2.2]

void ColorMask(boolean r, boolean g, boolean b, boolean a);

void DepthMask(boolean mask);

void stencilMask(uint mask);

void StencilMaskSeparate(uint face, uint mask);

Clearing the Buffers [4.2.3]

void Clear(bitfield buf);

buf: BITWISE OR of COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, STENCIL_BUFFER_BIT

void ClearColor(float r, float g, float b, float a);

void ClearDepth(float d);

void ClearStencil(int s);

Attaching Texture Images to a Framebuffer [4.3.1]

void FramebufferTexture2D(enum target, enum attachment, enum texTarget, uint texture, int level);

void FramebufferTexture3D(enum target, enum attachment, enum texTarget, uint texture, int level);

void FramebufferTexture1D(enum target, enum attachment, enum texTarget, uint texture, int level);

void FramebufferTextureStorage2D(enum target, enum attachment, uint texture, sizei width, sizei height);

void FramebufferTextureStorage3D(enum target, enum attachment, uint texture, sizei width, sizei height, sizei depth);

void FramebufferTextureStorage1D(enum target, enum attachment, uint texture, sizei width, sizei height);

void FramebufferTextureParameteri(enum target, enum attachment, int param, int value);

void FramebufferParametersi(enum target, enum attachment, int param, int value);

Manual Mipmap Generation [3.7.10]

void GenerateMipmap(enum target);

Texture Objects [3.7.12]

void BindTexture(enum target, uint texture);

void GenTextures(sizei n, uint *textures);

Enumerated Queries [6.1.3]

void GetTexParameterfv(enum target, enum parameter, float *data);

void GetTexParameteriv(enum target, enum parameter, int *data);

void GetTexEnvfv(enum target, enum parameter, float *data);

void GetTexEnviv(enum target, enum parameter, int *data);

void GetVertexAttribfv(enum target, int index, float *data);

void GetVertexAttribiv(enum target, int index, int *data);

void GetUniformfv(program target, int location, float *data);

void GetUniformiv(program target, int location, int *data);

Reading Pixels [4.3.1]

void ReadPixels(int x, int y, sizei width, sizei height, enum format, enum type, float *data);

Special Functions

Flush and Finish [5.1]

void Flush();

void Finish();

Hints [5.2]

Hint controls certain aspects of GL behavior.

void Hint(enum target, enum hint);

State and State Requests

A complete list of symbolic constants for states is shown in the tables in [6.2].

Simple Queries [6.1.3]

void GetBooleanv(enum value, boolean *data);

void GetIntegerv(enum value, int *data);

void GetFloatv(enum value, float *data);

void GetIntegerv(enum value, int *data);

boolean isEnabled(enum value);

Pointer and String Queries [6.1.4]

ubyte *GetString(enum name);

name: VENDOR, RENDERER, VERSION, SHADING_LANGUAGE_VERSION, EXTENSIONS

Find specifications, conformant product listings, and other resources at khorons.org/opengles/
The OpenGL® ES Shading Language is two closely-related languages which are used to create shaders for the vertex and fragment processors contained in the OpenGL ES processing pipeline.

[n.n.n] and [Table n.n] refer to sections and tables in the OpenGL ES Shading Language 1.0 specification at www.opengl.org/registry/gles

### Types [4.1]
A shader can aggregate these using arrays and structures to build more complex types. There are no pointer types.

#### Basic Types
- **void** no function return value or empty parameter list
- **bool** Boolean
- **int** signed integer
- **float** floating scalar
- **vec2, vec3, vec4** n-component floating point vector
- **bvec2, bvec3, bvec4** Boolean vector
- **ivec2, ivec3, ivec4** signed integer vector
- **mat2, mat3, mat4** 2x2, 3x3, 4x4 float matrix
- **sampler2D** access a 2D texture
- **samplerCube** access cube mapped texture

#### Structures and Arrays [4.1.8, 4.1.9]
- **Structures**
  - `struct type-name (members)`
  - `struct-name[]`
- **Arrays**
  - `float foo[3];`
  - `only 1-dimensional arrays supported`
  - `structure members can be arrays`

### Operators and Expressions [5.1]
Numbered in order of precedence. The relational and equality operators `>; <=` `=>` `!=` evaluate to a Boolean. To compare vectors component-wise, use functions such as `lessThan()`, `equal()`, etc.

#### Basic Operators
- `= + - * /` arithmetic
- `+= -= *= /=` arithmetic assignments
- `++` `--` prefix increment and decrement
- `!` logical not
- `& &` logical and
- `||` logical or
- `[]` array subscript
- `{` `}` structure members
- `? :` swizzler
- `+= -= *= /=` arithmetic assignments
- `sequence` sequence

#### Matrices and Vectors
- `vector` (Default) same as `vec1` for matrix operations that do not aggragate components
- `vector multiply` `*`
- `vector component-wise addition/subtraction` `+` `-`
- `vector cross product` `x`
- `vector dot product` `f`
- `vector length` `l`
- `vector magnitude` `m`
- `vector equality` `==`
- `vector relational` `< > <= >=` `!=` `==`

#### Assigment
- `=`
- `+= -= *= /=` arithmetic assignment

### Vector Components [5.5]
In addition to array numeric subscript syntax, names of vector components are denoted by a single letter. Components can be swizzled and replicated, e.g.: `pos.xx`, `pos.xy`

- `x, y, z` Use when accessing vectors that represent points or normals
- `g, b, a` Use when accessing vectors that represent colors
- `h, t, p, q` Use when accessing vectors that represent texture coordinates

### Preprocessor [3.4]
#### Preprocessor Directives
- `#define` `#undef` `#if` `#ifdef` `#ifndef` `#else`
- `#version` `#extension` `#extension_name`

#### Examples of Preprocessor Directives
- "#version 1.00" in a shader program specifies that the program is written in GLSL ES version 1.00. It is optional. If used, it must occur before anything else in the program other than whitespace or comments.
- `#extension extension_name : behavior, where behavior can be require, enable, warn, or disable, and where extension_name is the extension supported by the compiler`

#### Predefined Macros
- `_LINE_` Decimal integer constant that is one more than the number of preceding new-lines in the current source string
- `_FILE_` Decimal integer constant that says which source string number is currently being processed.
- `_VERSION_` Decimal integer, e.g., 100
- `GL_ES` Defined and set to integer 1 if running on an OpenGL-ES Shading Language.
- `GL_FRAGMENT_PRECISION_HIGH` 1 if highp is supported in the fragment language, else undefined [4.5.4]

### Qualifiers
#### Storage Qualifiers [4.3]
Variable declarations may be preceded by one storage qualifier.

- **none** (Default) local read/write memory, or input parameter
- **const** Compile-time constant, or read-only function parameter
- **attribute** Linkage between a vertex shader and OpenGL ES for per-vertex data
- **uniform** Value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL ES, and the application
- **varying** Linkage between a vertex shader and fragment shader for interpolated data

#### Uniform [4.3.4]
Use to declare global variables whose values are the same across the entire primitive being processed. All uniform variables are read-only. Use uniform qualifiers with any basic data types, to declare a variable whose type is a structure, or an array of any of those. For example:

- `uniform vec4 lightPosition;`

#### Varying [4.3.5]
The varying qualifier can be used only with the data types float, vec2, vec3, vec4, mat2, mat3, mat4, or arrays of these. Structures cannot be varying. Varying variables are required across the entire primitive being processed. All uniform variables are read-only. Use uniform qualifiers with any basic data types, to declare a variable whose type is a structure, or an array of any of those. For example:

- `vec2 Coord;`

#### Parameter Qualifiers [4.4]
Input values are copied in at function call time, output values are copied out at function return time.

- `in` For function parameters passed into a function
- `out` For function parameters passed back out of a function, but not used for when passed in
- `inout` For function parameters passed both into and out of a function

### Precision and Qualifiers [4.5]
Any floating point, integer, or sampler declaration can have the type preceded by one of these precision qualifiers:

- **highp** Satisfies minimum requirements for the vertex language. Optional in the fragment language.
- **mediump** Satisfies minimum requirements for the fragment language. Its range and precision is between that provided by lowp and highp.
- **lowp** Range and precision can be less than mediump, but still represents all color values for any color channel.

- `float lowp float color;` `lowp vec2 lowp float;` `lowp mat4;`

#### Ranges & precisions for precision qualifiers (FP=floating point):

<table>
<thead>
<tr>
<th>FP Range</th>
<th>FP Magnitude Range</th>
<th>FP Precision</th>
<th>Integer Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>highp</td>
<td>[2.0, 2.0]</td>
<td>Relative 1x2</td>
<td>[2.0, 2.0]</td>
</tr>
<tr>
<td>mediump</td>
<td>[2.0, 2.0]</td>
<td>Relative 1x2</td>
<td>[2.0, 2.0]</td>
</tr>
<tr>
<td>lowp</td>
<td>[2, 2]</td>
<td>Absolute 2x4</td>
<td>[2, 2]</td>
</tr>
</tbody>
</table>

A precision statement establishes a default precision qualifier for subsequent int, float, and sampler declarations, e.g.: `precision highp int;`

#### Invariant Qualifiers Examples [4.6]
- `#pragma STGDL invariant(all)` Force all output variables to be invariant
- `#pragma gl_Position;` Qualify a previously declared variable
- `#pragma varying mediump vec3 Color;` Qualify as part of a variable declaration

### Order of Qualification [4.7]
When multiple qualifications are present, they must follow a strict order. This order is as follows:

- `#pragma` as part of a variable declaration
- `#version` as part of a variable declaration
- `#extension` as part of a variable declaration
- `attribute` as part of a variable declaration
- `const` as part of a variable declaration
- `uniform` as part of a variable declaration
- `varying` as part of a variable declaration
- `in` as part of a variable declaration
- `out` as part of a variable declaration
- `inout` as part of a variable declaration
- `highp` as part of a variable declaration
- `mediump` as part of a variable declaration
- `lowp` as part of a variable declaration
- `int` as part of a variable declaration
- `float` as part of a variable declaration
- `vec2` as part of a variable declaration
- `vec3` as part of a variable declaration
- `vec4` as part of a variable declaration
- `mat2` as part of a variable declaration
- `mat3` as part of a variable declaration
- `mat4` as part of a variable declaration
- `bool` as part of a variable declaration
- `Sampler2D` as part of a variable declaration
- `SamplerCube` as part of a variable declaration
- `Sampler1DArray` as part of a variable declaration
- `Sampler2DArray` as part of a variable declaration
- `SamplerCubeArray` as part of a variable declaration

### Aggregate Operations and Constructors [5.4]

#### Matrix Constructor Examples [5.4]
- `mat2(float)` `mat2(vec2);` `mat2(vec2);` `mat2(float);` `mat2(float, float);` `mat2(float, float, float);`

#### Structure Constructor Example [5.4.3]
- `struct light { float _foo_; float _bar_; } long _lightData_;` `light = new long(_lightData_);`

#### Matrix Components [5.6]
Access components of a matrix with array subscripting syntax.

- `m[0][1] = m[1][0];` `m[0][1] = m[1][0];`

#### Structure Components [5.7]
Select structure fields using the period (.) operator. Other operators include:

- `.` `field selector`
- `==` `equality`
- `=` `assignment`

### Array Operations [4.1.9]
Array elements are accessed using the array subscript operator `[ ]`. For example:

- `array(6) = float[6];` `array(6)[3] = 12.34f;`
Built-In Inputs, Outputs, and Constants [7]

Output Special Variables may be read back after writing. Input Special Variables are read-only. All Special Variables have global scope.

Vertex Shader Special Variables [7.1]

<table>
<thead>
<tr>
<th>Outputs:</th>
<th>Description</th>
<th>Units or coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>highp vec4 gl_Position;</td>
<td>transformed vertex position</td>
<td>clip coordinates</td>
</tr>
<tr>
<td>mediump float gl_PointSize;</td>
<td>transformed point size (point rasterization only)</td>
<td>pixels</td>
</tr>
</tbody>
</table>

Fragment Shader Special Variables [7.2]

Fragment shaders may write to gl_FragColor or to one or more elements of gl_FragData[], but not both. The size of the gl_FragData array is given by the built-in constant gl_MaxDrawBuffers.

<table>
<thead>
<tr>
<th>Inputs:</th>
<th>Description</th>
<th>Units or coordinate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>mediump vec4 gl_FragCoord;</td>
<td>fragment position within frame buffer</td>
<td>window coordinates</td>
</tr>
<tr>
<td>bool gl_FrontFacing;</td>
<td>fragment belongs to a front-facing primitive</td>
<td>Boolean</td>
</tr>
<tr>
<td>mediump int gl_PointCoord;</td>
<td>fragment position within a point (point rasterization only)</td>
<td>0.0 to 1.0 for each component</td>
</tr>
</tbody>
</table>

Built-In Constants With Minimum Values [Appendix F7]

<table>
<thead>
<tr>
<th>Built-In Constant</th>
<th>Minimum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>const mediump int gl_MaxVertexAttributes</td>
<td>8</td>
</tr>
<tr>
<td>const mediump int gl_MaxVertexUniformVectors</td>
<td>128</td>
</tr>
<tr>
<td>const mediump int gl_MaxVertexVaryingVectors</td>
<td>8</td>
</tr>
<tr>
<td>const mediump int gl_MaxTextureImageUnits</td>
<td>8</td>
</tr>
<tr>
<td>const mediump int gl_MaxCombinedTextureImageUnits</td>
<td>8</td>
</tr>
<tr>
<td>const mediump int gl_MaxFragmentImageUnits</td>
<td>8</td>
</tr>
<tr>
<td>const mediump int gl_MaxFragmentUniformVectors</td>
<td>16</td>
</tr>
<tr>
<td>const mediump int gl_MaxDrawBuffers</td>
<td>1</td>
</tr>
</tbody>
</table>

Built-In Uniform State [7.5]

Specifies depth range in window coordinates. If an implementation does not support high precision in the fragment language, and state is listed as highp, then that state will only be available as mediump in the fragment language.

```cpp
struct gl_DepthRangeParameters {
  highp float near; // n
  highp float far; // f
  highp float diff; // F n
};
uniform gl_DepthRangeParameters gl_DepthRange;
```

Built-In Functions

**Angle & Trigonometry Functions [8.1]**

Component-wise operation. Parameters specified as angle & trigonometry functions. These functions operate on vectors as vectors, not component-wise. T is float, vec2, vec3, vec4.

- `float length(T)`
- `float distance(T p0, T p1)`
- `float dot(T x, T y)`
- `float cross(T x, T y)`
- `float normalize(T)`
- `float faceforward(N, T, T Kf)`
- `float reflect(T T Nx, T Nt)`
- `float refract(T T Nx, T Nt, float eta)`

**Exponential Functions [8.2]**

Component-wise operation. T is float, vec2, vec3, vec4.

- `float pow(T x, T y)`
- `float exp(T x)`
- `float log(T x)`
- `float log2(T x)`
- `float sqrt(T x)`
- `float inverseSqrt(T x)`

**Common Functions [8.3]**

Component-wise operation. T is float, vec2, vec3, vec4.

- `float abs(T x)`
- `float sign(T x)`
- `float floor(T x)`
- `float ceil(T x)`
- `float fract(T x)`
- `float mod(T x, T y)`
- `float min(T x, T y)`
- `float max(T x, T y)`
- `float clamp(T x, T minVal, T maxVal)`
- `float step(T edge, T x)`
- `float smoothstep(T edge0, T edge1, T x)`

**Geometric Functions [8.4]**

These functions operate on vectors as vectors, not component-wise. T is float, vec2, vec3, vec4.

- `float length(T)`
- `float distance(T p0, T p1)`
- `float dot(T x, T y)`
- `float cross(T x, T y)`
- `float normalize(T)`
- `float faceforward(N, T, T Kf)`
- `float reflect(T T Nx, T Nt)`
- `float refract(T T Nx, T Nt, float eta)`

**Matrix Functions [8.5]**

Type mat is any matrix type.

- `mat matrixCompMult(mat x, mat y)`
- `multiply(x, y)` component-wise

**Vector Relational Functions [8.6]**

Component-wise comparison. Sizes of input and return vectors for a particular call must match. Type bvec is bvec2, bvec3, bvec4.

- `bool lessThan(T x, T y)`
- `bool lessThanEqual(T x, T y)`
- `bool greaterThan(T x, T y)`
- `bool greaterThanEqual(T x, T y)`
- `bool equal(T x, T y)`
- `bool notEqual(T x, T y)`
- `bool all(bvec x)`
- `bool any(bvec x)`

**Texture Lookup Functions [8.7]**

Available only in vertex shader.

- `vec4 texture2Dlod(sampler2D sampler, vec2 coord, float lod)`
- `vec4 texture2DProj(sampler2D sampler, vec2 coord, float lod)`
- `vec4 texture2DProjRef(sampler2D sampler, vec3 coord, float lod)`
- `vec4 textureCube(samplerCube sampler, coord, float lod)`

**Statements and Structure**

**Iteration and Jumps [6]**

```cpp
Function Call call by value-return
Iteration for (;) { break, continue } while () { do, break, continue } while () {
Selection if () if () else ()
Jump break, continue, return discard
Entry void main()
```

**Sample Program**

A shader pair that applies diffuse and ambient lighting to a textured object.

**Vertex Shader**

```cpp
uniform mat4 mvp_matrix; // model-view-projection matrix
uniform vec3 ec_light_dir; // light direction in eye coords
uniform vec4 v_diffuse; // vertex position
uniform vec4 v_normal; // vertex normal
uniform vec2 v_texcoord; // texture coordinates
uniform mat4 model_matrix; // model matrix

// Put vertex normal into eye coords
v_normal = normalize(normal_matrix * v_normal);
// Emit diffuse scale factor, texture, and position
v_diffuse = max(dot(ec_light_dir, ec_normal), 0.0); v_texcoord = a_texcoord; gl_Position = mvp_matrix * v_position;
```

**Fragment Shader**

```cpp
precision mediump float;
uniform sampler2D t_reflectance;
uniform vec4 i_ambient;
uniform vec4 i_diffuse;
uniform vec2 i_texcoord;
void main(void) {
  vec4 color = texture2D(t_reflectance, i_texcoord);
  gl_FragColor = color * (vec4(i_diffuse) + i_ambient);
}
```