OpenGL® is the only cross-platform graphics API that enables developers of software for PC, workstation, and supercomputing hardware to create high-performance, visually-compelling graphics software applications, in markets such as CAD, content creation, energy, entertainment, game development, manufacturing, medical, and virtual reality. **Specifications are available at www.opengl.org/registry**

- *see FunctionName* refers to functions on this reference card.
- [n.n.n] and [Table n.n] refer to sections and tables in the OpenGL 4.2 core specification.
- [n.n.n] refers to sections in the OpenGL Shading Language 4.20 specification.

OpenGL Errors [2.5]

enum **GetError**(void); Returns the numeric error code.

OpenGL Operation

Floating-Point Numbers [2.1.1 - 2.1.2]

16-Bit	1-bit sign, 5-bit exponent, 10-bit mantissa
Unsigned 11-Bit	no sign bit, 5-bit exponent, 6-bit mantissa
Unsigned 10-Bit	no sign bit, 5-bit exponent, 5-bit mantissa

Command Letters [Table 2.1]

Letters are used in commands to denote types.

b -	byte (8 bits)	**ub** -	ubyte (8 bits)
s -	short (16 bits)	**us** -	ushort (16 bits)
i -	int (32 bits)	**ui** -	uint (32 bits)
i64 -	int64 (64 bits)	**ui64** -	uint64 (64 bits)
f -	float (32 bits)	**d** -	double (64 bits)

Vertex Arrays [2.8]

void **VertexAttribPointer**(uint *index*, int *size*, enum *type*, boolean *normalized*, sizei *stride*, const void **pointer*);
type: SHORT, INT, FLOAT, HALF_FLOAT, DOUBLE, {UNSIGNED_}INT_2_10_10_10_REV, FIXED, BYTE, UINT, UNSIGNED_{BYTE, SHORT}

void **VertexAttribIPointer**(uint *index*, int *size*, enum *type*, sizei *stride*, const void **pointer*);
type: BYTE, SHORT, UNSIGNED_{BYTE, SHORT}, INT, UINT
index: [0, MAX_VERTEX_ATTRIBS - 1]

void **VertexAttribLPointer**(uint *index*, int *size*, enum *type*, sizei *stride*, const void **pointer*);
type: DOUBLE
index: [0, MAX_VERTEX_ATTRIBS - 1]

void **EnableVertexAttribArray**(uint *index*);

void **DisableVertexAttribArray**(uint *index*);
index: [0, MAX_VERTEX_ATTRIBS - 1]

void **VertexAttribDivisor**(uint *index*, uint *divisor*);

Enable/Disable(PRIMITIVE_RESTART);

void **PrimitiveRestartIndex**(uint *index*);

Drawing Commands [2.8.3]

For all the functions in this section:
mode: POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_{STRIP, FAN}, TRIANGLES, LINES_ADJACENCY, {LINE, TRIANGLE}_STRIP_ADJACENCY, PATCHES, TRIANGLES_ADJACENCY
type: UNSIGNED_{BYTE, SHORT, INT}

void **DrawArraysOneInstance**(enum *mode*, int *first*, sizei *count*, int *instance*, uint *baseinstance*);

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

void **DrawArraysInstanced**(enum *mode*, int *first*, sizei *count*, sizei *primcount*);

void **DrawArraysInstancedBaseInstance**(enum *mode*, int *first*, sizei *count*, sizei *primcount*, uint *baseinstance*);

void **DrawArraysIndirect**(enum *mode*, const void **indirect*);

void **MultiDrawArrays**(enum *mode*, const int **first*, const sizei **count*, sizei *primcount*);

void **DrawElements**(enum *mode*, sizei *count*, enum *type*, const void **indices*);

void **DrawElementsInstanced**(enum *mode*, sizei *count*, enum *type*, const void **indices*, sizei *primcount*);

void **DrawElementsInstancedBaseInstance**(enum *mode*, sizei *count*, enum *type*, const void **indices*, sizei *primcount*, uint *baseinstance*);

void **DrawElementsInstancedBaseVertexBaseInstance**(enum *mode*, sizei *count*, enum *type*, const void **indices*, sizei *primcount*, int *basevertex*, uint *baseinstance*);

void **DrawElementsOneInstance**(enum *mode*, sizei *count*, enum *type*, const void **indices*, int *instance*, uint *baseinstance*);

void **MultiDrawElements**(enum *mode*, sizei **count*, enum *type*, const void ***indices*, sizei *primcount*);

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

void **DrawElementsBaseVertex**(enum *mode*, sizei *count*, enum *type*, const void **indices*, int *basevertex*);

void **DrawRangeElementsBaseVertex**(enum *mode*, uint *start*, uint *end*, sizei *count*, enum *type*, const void **indices*, int *basevertex*);

void **DrawElementsInstancedBaseVertex**(enum *mode*, sizei *count*, enum *type*, const void **indices*, sizei *primcount*, int *basevertex*);

void **DrawElementsIndirect**(enum *mode*, enum *type*, const void **indirect*);

void **MultiDrawElementsBaseVertex**(enum *mode*, sizei **count*, enum *type*, const void ***indices*, sizei *primcount*, int **basevertex*);

OpenGL Command Syntax [2.3]

GL commands are formed from a return type, a name, and optionally up to 4 characters (or character pairs) from the Command Letters table (above), as shown by the prototype:

return-type **Name**{1234}{b s i i64 f d ub us ui ui64}{v} (*[args ,] T arg1 , . . . , T argN [, args]*);

The arguments enclosed in brackets ([args ,] and [, args]) may or may not be present.
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 from the Command Table (above). If "v" is present, an array of N items is passed by a pointer.
For brevity, the OpenGL documentation and this reference may omit the standard prefixes.

The actual names are of the forms: glFunctionName(), GL_CONSTANT, GLtype

Vertex Specification [2.7]

Vertices have 2, 3, or 4 coordinates. The VertexAttrib* commands specify generic attributes with components of type float (VertexAttrib*), int or uint (VertexAttribI*), or double (VertexAttribL*).

void **VertexAttrib{1234}{sfd}**(uint *index*, T *values*);

void **VertexAttrib{123}{sfd}v**(uint *index*, const T *values*);

void **VertexAttrib4{bsifd ub us ui}v**(uint *index*, const T *values*);

void **VertexAttrib4Nub**(uint *index*, T *values*);

void **VertexAttrib4N{bsi ub us ui}v**(uint *index*, const T *values*);

void **VertexAttribI{1234}{i ui}**(uint *index*, T *values*);

void **VertexAttribI{1234}{i ui}v**(uint *index*, const T *values*);

void **VertexAttribI4{bs ub us}v**(uint *index*, const T *values*);

void **VertexAttribL{1234}d**(uint *index*, T *values*);

void **VertexAttribL{1234}dv**(uint *index*, const T *values*);

void **VertexAttribP{1234}ui**(uint *index*, enum *type*, boolean *normalized*, uint *value*);

void **VertexAttribP{1234}uiv**(uint *index*, enum *type*, boolean *normalized*, const uint **value*);
type: INT_2_10_10_10_REV, UNSIGNED_INT_2_10_10_10_REV

Buffer Objects [2.9-10]

void **GenBuffers**(sizei *n*, uint **buffers*);

void **DeleteBuffers**(sizei *n*, const uint **buffers*);

Creating and Binding Buffer Objects [2.9.1]

void **BindBuffer**(enum *target*, uint *buffer*);
target: PIXEL_{PACK, UNPACK}_BUFFER, {UNIFORM, ARRAY, TEXTURE}_BUFFER, COPY_{READ, WRITE}_BUFFER, DRAW_INDIRECT_BUFFER, ELEMENT_ARRAY_BUFFER, {TRANSFORM_FEEDBACK, ATOMIC_COUNTER}_BUFFER

void **BindBufferRange**(enum *target*, uint *index*, uint *buffer*, intptr *offset*, sizeiptr *size*);
target: ATOMIC_COUNTER_BUFFER, {TRANSFORM_FEEDBACK, UNIFORM}_BUFFER

void **BindBufferBase**(enum *target*, uint *index*, uint *buffer*);
target: see BindBufferRange

Creating Buffer Object Data Stores [2.9.2]

void **BufferSubData**(enum *target*, intptr *offset*, sizeiptr *size*, const void **data*);
target: see BindBuffer

void **BufferData**(enum *target*, sizeiptr *size*, const void **data*, enum *usage*);
usage: STREAM_{DRAW, READ, COPY}, {DYNAMIC, STATIC}_{DRAW, READ, COPY}
target: see BindBuffer

Mapping/Unmapping Buffer Data [2.9.3]

void ***MapBufferRange**(enum *target*, intptr *offset*, sizeiptr *length*, bitfield *access*);
access: The logical OR of MAP_{READ, WRITE}_BIT, MAP_INVALIDATE_{BUFFER, RANGE}_BIT, MAP_{FLUSH_EXPLICIT, UNSYNCHRONIZED}_BIT
target: see BindBuffer

void ***MapBuffer**(enum *target*, enum *access*);
access: READ_ONLY, WRITE_ONLY, READ_WRITE

void **FlushMappedBufferRange**(enum *target*, intptr *offset*, sizeiptr *length*);
target: see BindBuffer

boolean **UnmapBuffer**(enum *target*);
target: see BindBuffer

Copying Between Buffers [2.9.5]

void **CopyBufferSubData**(enum *readtarget*, enum *writetarget*, intptr *readoffset*, intptr *writeoffset*, sizeiptr *size*);
readtarget and *writetarget*: *see BindBuffer*

Vertex Array Objects [2.10]

All states related to definition of data used by vertex processor is in a vertex array object.

void **GenVertexArrays**(sizei *n*, uint **arrays*);

void **DeleteVertexArrays**(sizei *n*, const uint **arrays*);

void **BindVertexArray**(uint *array*);

Vertex Array Object Queries [6.1.10]

boolean **IsVertexArray**(uint *array*);

Buffer Object Queries [6.1.9]

boolean **IsBuffer**(uint *buffer*);

void **GetBufferParameteriv**(enum *target*, enum *pname*, int **data*);
target: see BindBuffer
pname: BUFFER_SIZE, BUFFER_USAGE, BUFFER_ACCESS{_FLAGS}, BUFFER_MAPPED, BUFFER_MAP_{OFFSET, LENGTH}

void **GetBufferParameteri64v**(enum *target*, enum *pname*, int64 **data*);
target: see BindBuffer
pname: see GetBufferParameteriv,

void **GetBufferSubData**(enum *target*, intptr *offset*, sizeiptr *size*, void **data*);
target: see BindBuffer

void **GetBufferPointerv**(enum *target*, enum *pname*, void ***params*);
target: see BindBuffer
pname: BUFFER_MAP_POINTER

Shaders and Programs

Shader Objects [2.11.1-2]

uint **CreateShader**(enum *type*);
type: {VERTEX, FRAGMENT, GEOMETRY}_SHADER, TESS_{EVALUATION, CONTROL}_SHADER

void **ShaderSource**(uint *shader*, sizei *count*, const char ***string*, const int **length*);

void **CompileShader**(uint *shader*);

void **ReleaseShaderCompiler**(void);

void **DeleteShader**(uint *shader*);

void **ShaderBinary**(sizei *count*, const uint **shaders*, enum *binaryformat*, const void **binary*, sizei *length*);

Program Objects [2.11.3]

uint **CreateProgram**(void);

void **AttachShader**(uint *program*, uint *shader*);

void **DetachShader**(uint *program*, uint *shader*);

void **LinkProgram**(uint *program*);

void **UseProgram**(uint *program*);

uint **CreateShaderProgramv**(enum *type*, sizei *count*, const char ***strings*);

void **ProgramParameteri**(uint *program*, enum *pname*, int *value*);

(parameters ⮥)

pname: PROGRAM_SEPARABLE, PROGRAM_BINARY_{RETRIEVABLE_HINT}
value: TRUE, FALSE

void **DeleteProgram**(uint *program*);

Program Pipeline Objects [2.11.4]

void **GenProgramPipelines**(sizei *n*, uint **pipelines*);

void **DeleteProgramPipelines**(sizei *n*, const uint **pipelines*);

void **BindProgramPipeline**(uint *pipeline*);

void **UseProgramStages**(uint *pipeline*, bitfield *stages*, uint *program*);
stages: ALL_SHADER_BITS or the bitwise OR of TESS_{CONTROL, EVALUATION}_SHADER_BIT, {VERTEX, GEOMETRY, FRAGMENT}_SHADER_BIT

void **ActiveShaderProgram**(uint *pipeline*, uint *program*);

Program Binaries [2.11.5]

void **GetProgramBinary**(uint *program*, sizei *bufSize*, sizei **length*, enum **binaryFormat*, void **binary*);

(Shaders and Programs Continue >)

Shaders and Program (cont.)

void **ProgramBinary**(uint *program*,
enum *binaryFormat*, const void **binary*,
sizei *length*);

Vertex Attributes [2.11.6]

Vertex shaders operate on array of 4-component items numbered from slot 0 to MAX_VERTEX_ATTRIBS - 1.

void **GetActiveAttrib**(uint *program*,
uint *index*, sizei *bufSize*, sizei **length*,
int **size*, enum **type*, char **name*);
**type* returns: FLOAT_{VEC*n*, MAT*n*, MAT*nxm*}, FLOAT, {UNSIGNED_}INT, {UNSIGNED_}INT_VEC*n*

int **GetAttribLocation**(uint *program*,
const char **name*);

void **BindAttribLocation**(uint *program*,
uint *index*, const char **name*);

Uniform Variables [2.11.7]

int **GetUniformLocation**(uint *program*,
const char **name*);

uint **GetUniformBlockIndex**(uint *program*,
const char **uniformBlockName*);

void **GetActiveUniformBlockName**(
uint *program*, uint *uniformBlockIndex*,
sizei *bufSize*, sizei **length*,
char **uniformBlockName*);

void **GetActiveUniformBlockiv**(
uint *program*, uint *uniformBlockIndex*,
enum *pname*, int **params*);
pname: UNIFORM_BLOCK_{BINDING, DATA_SIZE}, UNIFORM_BLOCK_NAME_{LENGTH, UNIFORM}, UNIFORM_BLOCK_ACTIVE_UNIFORMS_INDICES, or UNIFORM_BLOCK_REFERENCED_BY_*x*_SHADER, where *x* may be one of VERTEX, FRAGMENT, GEOMETRY, TESS_CONTROL, or TESS_EVALUATION

void **GetActiveAtomicCounterBufferBindingsiv**(
uint *program*, uint *bufferBindingIndex*,
enum *pname*, int **params*);

(parameters ⤴)

pname: UNIFORM_BLOCK_REFERENCED_-BY_TESS_EVALUATION_SHADER or ATOMIC_COUNTER_BUFFER_*n*, where *n* may be BINDING, DATA_SIZE, ACTIVE_ATOMIC_{COUNTERS, COUNTER_INDICES}, REFERENCED_BY_{VERTEX, TESS_CONTROL}_SHADER, REFERENCED_BY_{GEOMETRY, FRAGMENT}_SHADER

void **GetUniformIndices**(uint *program*,
sizei *uniformCount*, const char ***uniformNames*,
uint **uniformIndices*);

void **GetActiveUniformName**(uint *program*,
uint *uniformIndex*, sizei *bufSize*,
sizei **length*, char **uniformName*);

void **GetActiveUniform**(uint *program*,
uint *index*, sizei *bufSize*, sizei **length*,
int **size*, enum **type*, char **name*);
**type* returns: DOUBLE, DOUBLE_{VEC*n*, MAT*n*, MAT*nxn*}, FLOAT, FLOAT_{VEC*n*, MAT*n*, MAT*nxn*}, INT, INT_VEC*n*, UNSIGNED_INT{_VEC*n*}, BOOL, BOOL_VEC*n*, or any value in [Table 2.13]

void **GetActiveUniformsiv**(uint *program*,
sizei *uniformCount*, const uint **uniformIndices*,
enum *pname*, int **params*);
pname: UNIFORM_{TYPE, SIZE, NAME_LENGTH}, UNIFORM_BLOCK_INDEX, UNIFORM_OFFSET, UNIFORM_{ARRAY, MATRIX}_STRIDE, UNIFORM_IS_ROW_MAJOR

Load Uniform Vars. In Default Uniform Block

void **Uniform{1234}{ifd}**(int *location*, T *value*);

void **Uniform{1234}{ifd}v**(int *location*,
sizei *count*, const T *value*);

void **Uniform{1234}ui**(int *location*, T *value*);

void **Uniform{1234}uiv**(int *location*,
sizei *count*, const T *value*);

void **UniformMatrix{234}{fd}v**(
int *location*, sizei *count*,
boolean *transpose*, const T **value*);

void **UniformMatrix{2x3,3x2,2x4,4x2,
3x4,4x3}{fd}v**(int *location*, sizei *count*,
boolean *transpose*, const T **value*);

void **ProgramUniform{1234}{ifd}**(
uint *program*, int *location*, T *value*);

void **ProgramUniform{1234}{ifd}v**(
uint *program*, int *location*, sizei *count*,
const T *value*);

void **ProgramUniform{1234}ui**(
uint *program*, int *location*, T *value*);

void **ProgramUniform{1234}uiv**(
uint *program*, int *location*, sizei *count*,
const T *value*);

void **ProgramUniformMatrix{234}{fd}v**(
uint *program*, int *location*, sizei *count*,
boolean *transpose*, const float **value*);

void **ProgramUniformMatrixf{2x3,3x2,2x4,
4x2,3x4,4x3}{fd}v**(
uint *program*, int *location*, sizei *count*,
boolean *transpose*, const float **value*);

Uniform Buffer Object Bindings

void **UniformBlockBinding**(uint *program*,
uint *uniformBlockIndex*,
uint *uniformBlockBinding*);

Subroutine Uniform Variables [2.11.9]

int **GetSubroutineUniformLocation**(
uint *program*, enum *shadertype*,
const char **name*);

uint **GetSubroutineIndex**(uint *program*, enum
shadertype, const char **name*);

void **GetActiveSubroutineUniformiv**(
uint *program*, enum *shadertype*,
uint *index*, enum *pname*, int **values*);
pname: {NUM_}COMPATIBLE_SUBROUTINES, UNIFORM_SIZE, UNIFORM_NAME_LENGTH

void **GetActiveSubroutineUniformName**(
uint *program*, enum *shadertype*,
uint *index*, sizei *bufsize*, sizei **length*,
char **name*);

void **GetActiveSubroutineName**(
uint *program*, enum *shadertype*,
uint *index*, sizei *bufsize*, sizei **length*,
char **name*);

void **UniformSubroutinesuiv**(enum *shadertype*,
sizei *count*, const uint **indices*);

Output Variables [2.11.12]

void **TransformFeedbackVaryings**(
uint *program*, sizei *count*,
const char ***varyings*, enum *bufferMode*);
bufferMode: {INTERLEAVED, SEPARATE}_ATTRIBS

void **GetTransformFeedbackVarying**(
uint *program*, uint *index*, sizei *bufSize*,
sizei **length*, sizei **size*, enum **type*,
char **name*);
**type* returns NONE, FLOAT{_VEC*n*}, DOUBLE{_VEC*n*}, {UNSIGNED_}INT, {UNSIGNED_}INT_VEC*n*, MAT*nxm*, {FLOAT, DOUBLE}_MAT*n*, {FLOAT, DOUBLE}_MAT*nxm*

Shader Execution [2.11.13]

void **ValidateProgram**(uint *program*);

void **ValidateProgramPipeline**(uint *pipeline*);

Shader Memory Access [2.11.14]

void **MemoryBarrier**(bitfield *barriers*);
barriers: ALL_BARRIER_BITS or the OR of *n*_BARRIER_BIT, where *n* may be UNIFORM, VERTEX_ATTRIB_ARRAY, ELEMENT_ARRAY, TEXTURE_FETCH, BUFFER_UPDATE, PIXEL_BUFFER, SHADER_IMAGE_ACCESS, COMMAND, TEXTURE_UPDATE, FRAMEBUFFER, TRANSFORM_FEEDBACK, ATOMIC_COUNTER

Tessellation Primitive Generation[2.12.2]

void **PatchParameterfv**(enum *pname*,
const float **values*);
pname: PATCH_DEFAULT_{INNER, OUTER}_LEVEL

Fragment Shaders [3.10.2]

void **BindFragDataLocation**(uint *program*,
uint *colorNumber*, const char **name*);

void **BindFragDataLocationIndexed**(
uint *program*, uint *colorNumber*,
uint *index*, const char **name*);

int **GetFragDataLocation**(uint *program*,
const char **name*);

int **GetFragDataIndex**(uint *program*,
const char **name*);

Shader and Program Queries

Shader Queries [6.1.12]

boolean **IsShader**(uint *shader*);

void **GetShaderiv**(uint *shader*, enum *pname*,
int **params*);
pname: SHADER_TYPE, FRAGMENT_SHADER, {GEOMETRY, VERTEX}_SHADER, TESS_{CONTROL, EVALUATION}_SHADER, INFO_LOG_LENGTH, {DELETE, COMPILE}_STATUS, SHADER_SOURCE_LENGTH

void **GetShaderInfoLog**(uint *shader*,
sizei *bufSize*, sizei **length*, char **infoLog*);

void **GetShaderSource**(uint *shader*,
sizei *bufSize*, sizei **length*, char **source*);

void **GetShaderPrecisionFormat**(
enum *shadertype*, enum *precisiontype*,
int **range*, int **precision*);
shadertype: {VERTEX, FRAGMENT}_SHADER
precisiontype: LOW_{FLOAT, INT}, MEDIUM_{FLOAT, INT}, HIGH_{FLOAT, INT}

void **GetProgramStageiv**(uint program, enum
shadertype, enum *pname*, int **values*);
pname: ACTIVE_SUBROUTINES, ACTIVE_SUBROUTINE_{UNIFORMS, MAX_LENGTH}, ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS, ACTIVE_SUBROUTINE_UNIFORM_MAX_LENGTH

Program Queries [6.1.12]

void **GetAttachedShaders**(uint *program*,
sizei *maxCount*, sizei **count*, uint **shaders*);

void **GetVertexAttrib{d f i}v**(uint *index*, enum
pname, T **params*);
pname: CURRENT_VERTEX_ATTRIB or VERTEX_ATTRIB_ARRAY_*x* where *x* is one of BUFFER_BINDING, DIVISOR, ENABLED, INTEGER, NORMALIZED, SIZE, STRIDE, or TYPE

void **GetVertexAttribl{i ui}v**(uint *index*,
enum *pname*, T **params*);
pname: see GetVertexAttrib{d f i}v

void **GetVertexAttribLdv**(uint *index*,
enum *pname*, double **params*);
pname: see GetVertexAttrib{d f i}v

void **GetVertexAttribPointerv**(uint *index*,
enum *pname*, void ***pointer*);
pname: VERTEX_ATTRIB_ARRAY_POINTER

void **GetUniform{f d i ui}v**(uint *program*,
int *location*, T **params*);

void **GetUniformSubroutineuiv**(
enum *shadertype*, int *location*,
uint **params*);

boolean **IsProgram**(uint *program*);

void **GetProgramiv**(uint *program*,
enum *pname*, int **params*);
pname: DELETE_STATUS, LINK_STATUS, VALIDATE_STATUS, INFO_LOG_LENGTH, ATTACHED_SHADERS, ACTIVE_ATTRIBUTES, ACTIVE_ UNIFORMS{_BLOCKS},

(more values for *pname* ⤴)

ACTIVE_ATTRIBUTES_MAX_LENGTH, ACTIVE_UNIFORM_MAX_LENGTH, TRANSFORM_FEEDBACK_BUFFER_MODE, TRANSFORM_FEEDBACK_ VARYINGS, TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, GEOMETRY_VERTICES_OUT, GEOMETRY_{INPUT, OUTPUT}_TYPE, GEOMETRY_SHADER_INVOCATIONS, TESS_CONTROL_OUTPUT_VERTICES, TESS_GEN_{MODE, SPACING, VERTEX_ORDER}, TESS_GEN_POINT_MODE, PROGRAM_SEPARABLE, PROGRAM_BINARY_{LENGTH, RETRIEVABLE_HINT}

boolean **IsProgramPipeline**(uint *pipeline*);

void **GetProgramPipelineiv**(uint *pipeline*,
enum *pname*, int **params*);

void **GetProgramInfoLog**(uint *program*,
sizei *bufSize*, sizei **length*, char **infoLog*);

void **GetProgramPipelineInfoLog**(
uint *pipeline*, sizei *bufSize*,
sizei **length*, char **infoLog*);

Viewport and Clipping

Controlling Viewport [2.14.1]

void **DepthRangeArrayv**(uint *first*,
sizei *count*, const clampd **v*);

void **DepthRangeIndexed**(uint *index*,
clampd *n*, clampd *f*);

void **DepthRange**(clampd *n*, clampd *f*);

void **DepthRangef**(clampf *n*, clampd *f*);

void **ViewportArrayv**(uint *first*, sizei *count*,
const float **v*);

void **ViewportIndexedf**(uint *index*, float *x*,
float *y*, float *w*, float *h*);

void **ViewportIndexedfv**(uint *index*,
const float **v*);

void **Viewport**(int *x*, int *y*, sizei *w*, sizei *h*);

Clipping [2.20]

Enable/Disable(CLIP_DISTANCE*i*);
i: [0, MAX_CLIP_DISTANCES - 1]

Rendering Control & Queries

Asynchronous Queries [2.15]

void **BeginQuery**(enum *target*, uint *id*);
target: PRIMITIVES_GENERATED{*n*}, {ANY_}SAMPLES_PASSED, TIME_ELAPSED, TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN{*n*}

void **EndQuery**(enum *target*);

void **BeginQueryIndexed**(enum *target*,
uint *index*, uint *id*);

void **EndQueryIndexed**(enum *target*,
uint *index*);

void **GenQueries**(sizei *n*, uint **ids*);

void **DeleteQueries**(sizei *n*, const uint **ids*);

Conditional Rendering [2.16]

void **BeginConditionalRender**(uint *id*,
enum *mode*);
mode: QUERY_WAIT, QUERY_NO_WAIT, QUERY_BY_REGION_{WAIT, NO_WAIT}

void **EndConditionalRender**(void);

Transform Feedback [2.17]

void **GenTransformFeedbacks**(sizei *n*, uint **ids*);

void **DeleteTransformFeedbacks**(sizei *n*,
const uint **ids*);

void **BindTransformFeedback**(enum *target*,
uint *id*);
target: TRANSFORM_FEEDBACK

void **BeginTransformFeedback**(
enum *primitiveMode*);
primitiveMode: TRIANGLES, LINES, POINTS

void **EndTransformFeedback**(void);

void **PauseTransformFeedback**(void);

void **ResumeTransformFeedback**(void);

void **DrawTransformFeedback**(
enum *mode*, uint *id*);
mode: see Drawing Commands [2.8.3] on this card

void **DrawTransformFeedbackInstanced**(
enum *mode*, uint *id*, sizei *primcount*);

void **DrawTransformFeedbackStream**(
enum *mode*, uint *id*, uint *stream*);

void
DrawTransformFeedbackStreamInstanced(
enum *mode*, uint *id*, uint *stream*,
sizei *primcount*);

(Rendering Control & Queries Continue >)

Rendering Control (cont.)

Asynchronous Queries[6.1.7]

void **GetQueryiv**(enum *target*,
enum *pname*, int **params*);

target: ***see BeginQuery***, plus TIMESTAMP
pname: CURRENT_QUERY, QUERY_COUNTER_BITS

boolean **IsQuery**(uint *id*);

void **GetQueryIndexediv**(enum *target*,
uint *index*, enum *pname*, int **params*);

target: ***see BeginQuery***
pname: CURRENT_QUERY, QUERY_COUNTER_BITS

void **GetQueryObjectiv**(uint *id*,
enum *pname*, int **params*);

void **GetQueryObjectuiv**(uint *id*,
enum *pname*, uint **params*);

void **GetQueryObjecti64v**(uint *id*,
enum *pname*, int64 **params*);

void **GetQueryObjectui64v**(uint *id*,
enum *pname*, uint64 **params*);

pname: QUERY_RESULT{_AVAILABLE}

Transform Feedback Query [6.1.11]

boolean **IsTransformFeedback**(uint *id*);

Lighting and Color

Flatshading [2.19]

void **ProvokingVertex**(enum *provokeMode*);

provokeMode: {FIRST, LAST}_VERTEX_CONVENTION

Reading Pixels [4.3.1]

void **ClampColor**(enum *target*, enum *clamp*);

target: CLAMP_READ_COLOR
clamp: TRUE, FALSE, FIXED_ONLY

Rasterization [3]

Enable/Disable(*target*);

target: RASTERIZER_DISCARD, MULTISAMPLE, SAMPLE_SHADING

Multisampling [3.3.1]

Use to antialias points, and lines.

void **GetMultisamplefv**(enum *pname*,
uint *index*, float **val*);

pname: SAMPLE_POSITION

void **MinSampleShading**(clampf *value*);

Points [3.4]

void **PointSize**(float *size*);

void **PointParameter{if}**(enum *pname*,
T *param*);

void **PointParameter{if}v**(enum *pname*, const
T *params*);

param, params: The fade threshold if *pname* is POINT_FADE_THRESHOLD_SIZE; {LOWER|UPPER}_LEFT if *pname* is POINT_SPRITE_COORD_ORIGIN. LOWER_LEFT, UPPER_LEFT, pointer to point fade threshold

pname: POINT_FADE_THRESHOLD_SIZE, POINT_SPRITE_COORD_ORIGIN

Enable/Disable (*target*);

target: VERTEX_PROGRAM_POINT_SIZE

Line Segments [3.5]

void **LineWidth**(float *width*);

Enable/Disable(LINE_SMOOTH);

Polygons [3.6]

Enable/Disable(*target*);

target: POLYGON_SMOOTH, CULL_FACE

void **FrontFace**(enum *dir*);

dir: CCW, CW

void **CullFace**(enum *mode*);

mode: FRONT, BACK, FRONT_AND_BACK

Polygon Rast. & Depth Offset [3.6.3-4]

void **PolygonMode**(enum *face*, enum *mode*);

face: FRONT_AND_BACK
mode: POINT, LINE, FILL

void **PolygonOffset**(float *factor*, float *units*);

Enable/Disable(*target*);

target: POLYGON_OFFSET_{POINT, LINE, FILL}

Pixel Storage Modes [3.7.1]

void **PixelStore{if}**(enum *pname*, T *param*);

pname: {UN}PACK_*x* (*x* may be SWAP_BYTES, LSB_FIRST, ROW_LENGTH, SKIP_{PIXELS, ROWS}, ALIGNMENT, IMAGE_HEIGHT, SKIP_IMAGES), UNPACK_ COMPRESSED_BLOCK_{WIDTH, HEIGHT, DEPTH, SIZE}

Texturing [3.9]

void **ActiveTexture**(enum *texture*);

texture: TEXTURE*i* where *i* is [0, max(MAX_TEXTURE_COORDS, MAX_COMBINED_TEXTURE_IMAGE_UNITS)-1]

Texture Objects [3.9.1]

void **BindTexture**(enum *target*,
uint *texture*);

target: TEXTURE_{1, 2}D{_ARRAY}, TEXTURE_{3D, RECTANGLE, BUFFER}, TEXTURE_CUBE_MAP{_ARRAY}, TEXTURE_2D_MULTISAMPLE{_ARRAY}

void **DeleteTextures**(sizei *n*,
const uint **textures*);

void **GenTextures**(sizei *n*, uint **textures*);

Sampler Objects [3.9.2]

void **GenSamplers**(sizei *count*,
uint **samplers*);

void **BindSampler**(uint *unit*, uint *sampler*);

void **SamplerParameter{if}v**(uint *sampler*,
enum *pname*, const T *param*);

void **SamplerParameterI{u ui}v**(uint *sampler*,
enum *pname*, const T **params*);

pname: TEXTURE_WRAP_{S, T, R}, TEXTURE_{MIN, MAG}_{FILTER, LOD}, TEXTURE_BORDER_COLOR, TEXTURE_LOD_BIAS, TEXTURE_COMPARE_{MODE, FUNC}

void **DeleteSamplers**(sizei *count*,
const uint **samplers*);

Texture Image Spec. [3.9.3]

void **TexImage3D**(enum *target*, int *level*,
int *internalformat*, sizei *width*, sizei *height*,
sizei *depth*, int *border*, enum *format*,
enum *type*, const void **data*);

target: TEXTURE_{3D, 2D_ARRAY, CUBE_MAP_ARRAY}, PROXY_TEXTURE_{3D, 2D_ARRAY, CUBE_MAP_ARRAY}

internalformat: DEPTH_COMPONENT, DEPTH_STENCIL, RED, INTENSITY, RG, RGB, RGBA; or a sized internal format from [Tables 3.12-3.13], COMPRESSED_{RED_RGTC1,RG_RGTC2}, COMPRESSED_SIGNED_{RED_RGTC1,RG_RGTC2}, or a specific compressed format in [Table 3.14]

format: DEPTH_COMPONENT, DEPTH_STENCIL, RED, GREEN, BLUE, RG, RGB, {RED, GREEN, BLUE}_INTEGER, {RG, RGB, RGBA, BGR}_INTEGER, BGRA_INTEGER, RGBA, BGR, BGRA [Table 3.3]

type: {UNSIGNED_}BYTE, {UNSIGNED_}SHORT, {UNSIGNED_}INT, HALF_FLOAT, FLOAT, or a value from [Table 3.2]

void **TexImage2D**(enum *target*, int *level*,
int *internalformat*, sizei *width*,
sizei *height*, int *border*, enum *format*,
enum *type*, const void **data*);

target: TEXTURE_{2D, RECTANGLE, CUBE_MAP}, PROXY_TEXTURE_{2D, RECTANGLE, CUBE_MAP}, TEXTURE_1D_ARRAY, PROXY_TEXTURE_1D_ARRAY, TEXTURE_CUBE_MAP_POSITIVE_{X, Y, Z}, TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}

(more values for *target* ⮥)

internalformat, format, and *type:* ***see TexImage3D***

void **TexImage1D**(enum *target*, int *level*,
int *internalformat*, sizei *width*, int *border*,
enum *format*, enum *type*,
const void **data*);

target: TEXTURE_1D, PROXY_TEXTURE_1D
type, internalformat, and *format:* ***see TexImage3D***

Alternate Texture Image Spec. [3.9.4]

void **CopyTexImage2D**(enum *target*,
int *level*, enum *internalformat*, int *x*,
int *y*, sizei *width*, sizei *height*, int *border*);

target: TEXTURE_{2D, RECTANGLE, 1D_ARRAY}, TEXTURE_CUBE_MAP_{POSITIVE, NEGATIVE}_{X, Y, Z}
internalformat: ***see TexImage3D***, except 1, 2, 3, 4

void **CopyTexImage1D**(enum *target*,
int *level*, enum *internalformat*, int *x*,
int *y*, sizei *width*, int *border*);

target: TEXTURE_1D
internalformat: ***see TexImage3D***, except 1, 2, 3, 4

void **TexSubImage3D**(enum *target*, int *level*,
int *xoffset*, int *yoffset*, int *zoffset*,
sizei *width*, sizei *height*, sizei *depth*,
enum *format*, enum *type*, const void **data*);

target: TEXTURE_3D, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY
format and *type:* ***see TexImage3D***

void **TexSubImage2D**(enum *target*, int *level*,
int *xoffset*, int *yoffset*, sizei *width*,
sizei *height*, enum *format*, enum *type*,
const void **data*);

target: ***see CopyTexImage2D***
format and *type:* ***see TexImage3D***

void **TexSubImage1D**(enum *target*, int *level*,
int *xoffset*, sizei *width*, enum *format*,
enum *type*, const void **data*);

target: TEXTURE_1D
format, type: ***see TexImage3D***

void **CopyTexSubImage3D**(enum *target*,
int *level*, int *xoffset*, int *yoffset*, int *zoffset*,
int *x*, int *y*, sizei *width*, sizei *height*);

target: ***see TexSubImage3D***

void **CopyTexSubImage2D**(enum *target*,
int *level*, int *xoffset*, int *yoffset*, int *x*,
int *y*, sizei *width*, sizei *height*);

target: TEXTURE_2D, TEXTURE_1D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_{POSITIVE, NEGATIVE}_{X, Y, Z}

void **CopyTexSubImage1D**(enum *target*,
int *level*, int *xoffset*, int *x*, int *y*, sizei *width*);

target: TEXTURE_1D

Compressed Texture Images [3.9.5]

void **CompressedTexImage3D**(enum *target*,
int *level*, enum *internalformat*, sizei *width*,
sizei *height*, sizei *depth*, int *border*,
sizei *imageSize*, const void **data*);

target: ***see TexImage3D***
internalformat: COMPRESSED_RED_RGTC1, COMPRESSED_SIGNED_RED_RGTC1, COMPRESSED_RG_RGTC2, COMPRESSED_SIGNED_RG_RGTC2

void **CompressedTexImage2D**(enum *target*,
int *level*, enum *internalformat*,
sizei *width*, sizei *height*, int *border*,
sizei *imageSize*, const void **data*);

target: ***see TexImage3D***, omitting compressed rectangular texture formats
internalformat: ***see CompressedTexImage3D***

void **CompressedTexImage1D**(enum *target*,
int *level*, enum *internalformat*,
sizei *width*, int *border*, sizei *imageSize*,
const void **data*);

target: TEXTURE_1D, PROXY_TEXTURE_1D
internalformat: values are implementation-dependent

void **CompressedTexSubImage3D**(
enum *target*, int *level*, int *xoffset*,
int *yoffset*, int *zoffset*, sizei *width*,
sizei *height*, sizei *depth*, enum *format*,
sizei *imageSize*, const void **data*);

target: ***see TexSubImage3D***
format: ***see internalformat for CompressedTexImage3D***

void **CompressedTexSubImage2D**(
enum *target*, int *level*, int *xoffset*,
int *yoffset*, sizei *width*, sizei *height*,
enum *format*, sizei *imageSize*,
cont void **data*);

target: ***see TexSubImage2D***
format: ***see TexImage3D***

void **CompressedTexSubImage1D**(
enum *target*, int *level*, int *xoffset*,
sizei *width*, enum *format*, sizei *imageSize*,
const void **data*);

target: ***see TexSubImage1D***
format: ***see TexImage3D***

Multisample Textures [3.9.6]

void **TexImage3DMultisample**(enum *target*,
sizei *samples*, int *internalformat*,
sizei *width*, sizei *height*, sizei *depth*,
boolean *fixedsamplelocations*);

target: {PROXY_}TEXTURE_2D_MULTISAMPLE_ARRAY
internalformat: RED, RG, RGB, RGBA, DEPTH_{COMPONENT, STENCIL}, STENCIL_INDEX, or sized internal formats corresponding to these base formats

void **TexImage2DMultisample**(enum *target*,
sizei *samples*, int *internalformat*,
sizei *width*, sizei *height*,
boolean *fixedsamplelocations*);

target: {PROXY_}TEXTURE_2D_MULTISAMPLE
internalformat: ***see TexImage3DMultisample***

Buffer Textures [3.9.7]

void **TexBuffer**(enum *target*,
enum *internalformat*, uint *buffer*);

target: TEXTURE_BUFFER
internalformat: R8{I,UI}, R16{F, I, UI}, R32{F, I, UI}, RG8{I, UI}, RG16{F, I, UI}, RG32{F, I, UI}, RGB32{F, I, UI}, RGBA8{I, UI}, RGBA16{F, I, UI}, RGBA32{F, I, UI}

Texture Parameters [3.9.8]

void **TexParameter{if}**(enum *target*,
enum *pname*, T *param*);

void **TexParameter{if}v**(enum *target*,
enum *pname*, const T **params*);

void **TexParameterI{i ui}v**(enum *target*,
enum *pname*, const T **params*);

target: TEXTURE_{1D,2D,3D}, TEXTURE_{1D,2D}_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP{_ARRAY}
pname: TEXTURE_WRAP_{S, T, R}, TEXTURE_{MIN, MAG}_FILTER, TEXTURE_LOD_BIAS, TEXTURE_BORDER_COLOR, TEXTURE_{MIN, MAX}_LOD, TEXTURE_SWIZZLE_{R, G, B, A, RGBA}, TEXTURE_COMPARE_{MODE, FUNC}, TEXTURE_{BASE, MAX}_LEVEL [Table 3.16]

Cube Map Texture Select [3.9.10]

Enable/Disable(
TEXTURE_CUBE_MAP_SEAMLESS);

Texture Minification [3.9.11]

void **GenerateMipmap**(enum *target*);

target: TEXTURE_{1D, 2D, 3D}, TEXTURE_{1D, 2D}_ARRAY, TEXTURE_CUBE_MAP{_ARRAY}

Immutable-Format Tex. Images [3.9.16]

void **TexStorage1D**(enum *target*,
sizei *levels*, enum *internalformat*,
sizei *width*);

target: TEXTURE_1D, PROXY_TEXTURE_1D
internalformat: any of the sized internal color, depth, and stencil formats in [Tables 3.12-13]

void **TexStorage2D**(enum *target*,
sizei *levels*, enum *internalformat*,
sizei *width*, sizei *height*);

target: TEXTURE_2D, PROXY_TEXTURE_2D, TEXTURE_{RECTANGLE, CUBE_MAP, 1D_ARRAY}, PROXY_TEXTURE_{RECTANGLE, CUBE_MAP, 1D_ARRAY}
internalformat: ***see TexStorage3D***

void **TexStorage3D**(enum *target*,
sizei *levels*, enum *internalformat*,
sizei *width*, sizei *height*, sizei *depth*);

target: TEXTURE_3D, PROXY_TEXTURE_3D, TEXTURE_{2D, CUBE_MAP}{_ARRAY}, PROXY_TEXTURE_{CUBE_MAP, 2D}{_ARRAY}
internalformat: ***see TexStorage3D***

(Texturing Continue >)

Texturing (cont.)

Texture Image Loads/Stores [3.9.20]

void **BindImageTexture**(uint *index*, uint *texture*, int *level*, boolean *layered*, int *layer*, enum *access*, enum *format*);

access: READ_ONLY, WRITE_ONLY, READ_WRITE

format: RGBA{32,16}F, RG{32,16}F, R{32,16}F, RGBA{32,16,8}UI, R11F_G11F_B10F, RGB10_A2UI, RG{32,16,8}UI, R{32,16,8}UI, RGBA{32,16,8}I, RG{32,16,8}I, R{32,16,8}I, RGBA{16,8}, RGB10_A2, RG{16,8}, R{16,8}, RGBA{16,8}_SNORM, RG{16,8}_SNORM, R{16,8}_SNORM [Table 3.21]

Enumerated Queries [6.1.3]

void **GetTexParameter{if}v**(enum *target*, enum *value*, T *data*);

void **GetTexParameterI{i ui}v**(enum *target*, enum *value*, T *data*);

target: TEXTURE_{1D, 2D, 3D,RECTANGLE}, TEXTURE_{1D, 2D}_ARRAY, TEXTURE_CUBE_MAP{_ARRAY}

(more parameters ↥)

value: IMAGE_FORMAT_COMPATIBILITY_TYPE, TEXTURE_IMMUTABLE_FORMAT, TEXTURE_{BASE, MAX}_LEVEL, TEXTURE_BORDER_COLOR, TEXTURE_LOD_BIAS, TEXTURE_COMPARE_{MODE, FUNC}, TEXTURE_{MIN, MAG}_FILTER, TEXTURE_MAX_{LEVEL, LOD}, TEXTURE_MIN_LOD, TEXTURE_SWIZZLE_{R, G, B, A, RGBA}, TEXTURE_WRAP_{S, T, R} [Table 3.16]

void **GetTexLevelParameter{if}v**(enum *target*, int *lod*, enum *value*, T *data*);

target: {PROXY_}TEXTURE_{1D, 2D, 3D}, TEXTURE_BUFFER, PROXY_TEXTURE_CUBE_MAP, {PROXY_}TEXTURE_{1D, 2D}_ARRAY, {PROXY_}TEXTURE_CUBE_MAP_ARRAY, {PROXY_}TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_{POSITIVE, NEGATIVE}_{X, Y, Z}, {PROXY_}TEXTURE_2D_MULTISAMPLE{_ARRAY}

(more parameters ↥)

value: TEXTURE_{WIDTH, HEIGHT, DEPTH}, TEXTURE_SAMPLES, TEXTURE_FIXED_SAMPLE_LOCATIONS, TEXTURE_{INTERNAL_FORMAT, SHARED_SIZE}, TEXTURE_COMPRESSED{_IMAGE_SIZE}, TEXTURE_BUFFER_DATA_STORE_BINDING, TEXTURE_*x*_{SIZE, TYPE} (where *x* can be RED, GREEN, BLUE, ALPHA, DEPTH, STENCIL)

Texture Queries [6.1.4]

void **GetTexImage**(enum *tex*, int *lod*, enum *format*, enum *type*, void *img);

tex: TEXTURE_{1, 2}D{_ARRAY}, TEXTURE_3D, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_CUBE_MAP_POSITIVE_{X, Y, Z}, TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}

format: ***see TexImage3D***

type: {UNSIGNED_}BYTE, UNSIGNED_{SHORT}, {UNSIGNED_}INT, {HALF_}FLOAT, or value from [Table 3.2]

void **GetCompressedTexImage**(enum *target*, int *lod*, void *img);

target: ***see "tex" for GetTexImage***

boolean **IsTexture**(uint *texture*);

Sampler Queries [6.1.5]

boolean **IsSampler**(uint *sampler*);

void **GetSamplerParameter{if}v**(uint *sampler*, enum *pname*, T *params);

void **GetSamplerParameterI{i ui}v**(uint *sampler*, enum *pname*, T *params);

pname: TEXTURE_WRAP_{S, T, R}, TEXTURE_{MIN, MAG}_FILTER, TEXTURE_BORDER_COLOR, TEXTURE_LOD_BIAS, TEXTURE_{MIN, MAX}_LOD, TEXTURE_COMPARE_{MODE, FUNC}

Per-Fragment Operations

Scissor Test [4.1.2]

Enable/Disable(SCISSOR_TEST);

Enablei/Disablei(SCISSOR_TEST, uint *index*);

void **ScissorArrayv**(uint *first*, sizei *count*, const int *v);

void **ScissorIndexed**(uint *index*, int *left*, int *bottom*, sizei *width*, sizei *height*);

void **ScissorIndexedv**(uint *index*, int *v);

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

Multisample Fragment Operations [4.1.3]

Enable/Disable(*target*);

target: SAMPLE_ALPHA_TO_{COVERAGE, ONE}, SAMPLE_{COVERAGE, MASK}, MULTISAMPLE

void **SampleCoverage**(clampf *value*, boolean *invert*);

void **SampleMaski**(uint *maskNumber*, bitfield *mask*);

Stencil Test [4.1.4]

Enable/Disable(STENCIL_TEST);

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

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

func: NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GREATER, GEQUAL, NOTEQUAL

void **StencilOp**(enum *sfail*, enum *dpfail*, enum *dppass*);

void **StencilOpSeparate**(enum *face*, enum *sfail*, enum *dpfail*, enum *dppass*);

face: FRONT, BACK, FRONT_AND_BACK

sfail, *dpfail*, and *dppass*: KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP, DECR_WRAP

Depth Buffer Test [4.1.5]

Enable/Disable(DEPTH_TEST);

void **DepthFunc**(enum *func*);

func: ***see StencilFuncSeparate***

Occlusion Queries [4.1.6]

BeginQuery(enum *target*, uint *id*);

EndQuery(enum *target*);

target: SAMPLES_PASSED, ANY_SAMPLES_PASSED

Blending [4.1.7]

Enable/Disable(BLEND);

Enablei/Disablei(BLEND, uint *index*);

void **BlendEquation**(enum *mode*);

void **BlendEquationi**(uint *buf*, enum *mode*);

void **BlendEquationSeparate**(enum *modeRGB*, enum *modeAlpha*);

mode, modeRGB, and *modeAlpha*: FUNC_ADD, FUNC_{SUBTRACT, REVERSE}_SUBTRACT, MIN, MAX

void **BlendEquationSeparatei**(uint *buf*, enum *modeRGB*, enum *modeAlpha*);

mode, modeRGB, and *modeAlpha*: ***see BlendEquationSeparate***

void **BlendFunc**(enum *src*, enum *dst*);

srd, dst: ***see BlendFuncSeparate***

void **BlendFunci**(uint *buf*, enum *src*, enum *dst*);

srd, dst: ***see BlendFuncSeparate***

void **BlendFuncSeparate**(enum *srcRGB*, enum *dstRGB*, enum *srcAlpha*, enum *dstAlpha*);

src, dst, *srcRGB, dstRGB*, *srcAlpha*, *dstAlpha*: ZERO, ONE, SRC_{COLOR, ALPHA}, DST_{COLOR, ALPHA}, SRC_ALPHA_SATURATE, CONSTANT_{COLOR, ALPHA}, ONE_MINUS_{SRCDST, CONSTANT}_{COLOR, ALPHA}, {ONE_MINUS_}SRC1_ALPHA

void **BlendFuncSeparatei**(uint *buf*, enum *srcRGB*, enum *dstRGB*, enum *srcAlpha*, enum *dstAlpha*);

dst, *dstRGB*, *dstAlpha*, src, *srcRGB*, *srcAlpha*: ***see BlendFuncSeparate***

void **BlendColor**(clampf *red*, clampf *green*, clampf *blue*, clampf *alpha*);

Dithering [4.1.9]

Enable/Disable(DITHER);

Logical Operation [4.1.10]

Enable/Disable(enum *op*);

op: INDEX_LOGIC_OP, {COLOR_}LOGIC_OP

void **LogicOp**(enum *op*);

op: CLEAR, AND, AND_REVERSE, COPY, AND_INVERTED, NOOP, OR, OR, NOR, EQUIV, INVERT, OR_REVERSE, COPY_INVERTED, OR_INVERTED, NAND, SET

Whole Framebuffer

Selecting Buffers for Writing [4.2.1]

void **DrawBuffer**(enum *buf*);

buf: NONE, FRONT{_LEFT, _RIGHT}, LEFT, RIGHT, FRONT_AND_BACK, BACK{_LEFT, _RIGHT}, COLOR_ATTACHMENT*i* (*i* = [0, MAX_COLOR_ATTACHMENTS - 1]), AUX*i* (*i* =[0, AUX_BUFFERS - 1])

void **DrawBuffers**(sizei *n*, const enum *bufs);

bufs: NONE, FRONT_{LEFT, RIGHT}, BACK_LEFT, BACK_RIGHT, COLOR_ATTACHMENT*i* where *i* = [0, MAX_COLOR_ATTACHMENTS - 1], AUX*i* where *i* =[0, AUX_BUFFERS - 1])

Fine Control of Buffer Updates [4.2.2]

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

void **ColorMaski**(uint *buf*, boolean *r*, boolean *g*, boolean *b*, boolean *a*);

void **StencilMask**(uint *mask*);

void **StencilMaskSeparate**(enum *face*, uint *mask*);

face: FRONT, BACK, FRONT_AND_BACK

void **DepthMask**(boolean *mask*);

Clearing the Buffers [4.2.3]

void **ClearColor**(clampf *r*, clampf *g*, clampf *b*, clampf *a*);

void **ClearDepth**(clampd *d*);

void **ClearDepthf**(clampf *d*);

void **ClearStencil**(int *s*);

void **ClearBuffer{if ui}v**(enum *buffer*, int *drawbuffer*, const T *value);

buffer: COLOR, DEPTH, STENCIL

void **ClearBufferfi**(enum *buffer*, int *drawbuffer*, float *depth*, int *stencil*);

buffer: DEPTH_STENCIL

drawbuffer: 0

Framebuffer Objects

Binding and Managing [4.4.1]

void **BindFramebuffer**(enum *target*, uint *framebuffer*);

target: {DRAW_ , READ_}FRAMEBUFFER

void **DeleteFramebuffers**(sizei *n*, const uint *framebuffers);

void **GenFramebuffers**(sizei *n*, uint *ids);

Attaching Images [4.4.2]

Renderbuffer Objects

void **BindRenderbuffer**(enum *target*, uint *renderbuffer*);

target: RENDERBUFFER

void **DeleteRenderbuffers**(sizei *n*, const uint *renderbuffers);

void **GenRenderbuffers**(sizei *n*, uint *renderbuffers);

void **RenderbufferStorageMultisample**(enum *target*, sizei *samples*, enum *internalformat*, sizei *width*, sizei *height*);

target: RENDERBUFFER

internalformat: ***see TexImage3DMultisample in the Texturing section of this card***

void **RenderbufferStorage**(enum *target*, enum *internalformat*, sizei *width*, sizei *height*);

target and *internalformat*: ***see RenderbufferStorageMultisample***

Attaching Renderbuffer Images

void **FramebufferRenderbuffer**(enum *target*, enum *attachment*, enum *renderbuffertarget*, uint *renderbuffer*);

(parameters ↥)

target: {DRAW_, READ_}FRAMEBUFFER

attachment: {DEPTH, STENCIL}_ATTACHMENT, DEPTH_STENCIL_ATTACHMENT, COLOR_ATTACHMENT*i* where *i* is [0, MAX_COLOR_ATTACHMENTS - 1]

renderbuffertarget: RENDERBUFFER

Attaching Texture Images

void **FramebufferTexture**(enum *target*, enum *attachment*, uint *texture*, int *level*);

target: {DRAW_, READ_}FRAMEBUFFER

attachment: ***see FramebufferRenderbuffer***

void **FramebufferTexture3D**(enum *target*, enum *attachment*, enum *textarget*, uint *texture*, int *level*, int *layer*);

textarget: TEXTURE_3D

target and *attachment*: ***see framebufferRenderbuffer***

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

textarget: TEXTURE_{2D, RECTANGLE}, TEXTURE_2D_MULTISAMPLE, TEXTURE_CUBE_MAP_POSITIVE_{X, Y, Z}, TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}

target, attachment: ***see FramebufferRenderbuffer***

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

textarget: TEXTURE_1D

target, attachment: ***see FramebufferRenderbuffer***

void **FramebufferTextureLayer**(enum *target*, enum *attachment*, uint *texture*, int *level*, int *layer*);

target, attachment: ***see FramebufferTexture3D***

Framebuffer Completeness [4.4.4]

enum **CheckFramebufferStatus**(enum *target*);

target: {DRAW, READ}_FRAMEBUFFER, FRAMEBUFFER

returns: FRAMEBUFFER_COMPLETE or a constant indicating the violating value

Framebuffer Object Queries [6.1.13]

boolean **IsFramebuffer**(uint *framebuffer*);

void **GetFramebufferAttachmentParameteriv**(enum *target*, enum *attachment*, enum *pname*, int *params);

target: {DRAW_, READ_}FRAMEBUFFER

attachment: FRONT_{LEFT, RIGHT}, BACK_{LEFT,RIGHT}, COLOR_ATTACHMENT*i*, DEPTH, STENCIL, {DEPTH, STENCIL}_ATTACHMENT, DEPTH_STENCIL_ATTACHMENT

pname: FRAMEBUFFER_ATTACHMENT_*x* (where *x* may be OBJECT_TYPE, OBJECT_NAME, RED_SIZE, GREEN_SIZE, BLUE_SIZE, ALPHA_SIZE, DEPTH_SIZE, STENCIL_SIZE, COMPONENT_TYPE, COLOR_ENCODING, TEXTURE_LEVEL, LAYERED, TEXTURE_CUBE_MAP_FACE, TEXTURE_LAYER)

Renderbuffer Object Queries [6.1.14]

boolean **IsRenderbuffer**(uint *renderbuffer*);

void **GetRenderbufferParameteriv**(enum *target*, enum *pname*, int *params);

target: RENDERBUFFER

pname: RENDERBUFFER_*x* (where *x* may be WIDTH, HEIGHT, INTERNAL_FORMAT, SAMPLES, {RED, GREEN, BLUE, ALPHA, DEPTH, STENCIL}_SIZE)

Reading, and Copying Pixels

Reading Pixels [4.3.1]

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

format: STENCIL_INDEX, DEPTH_{COMPONENT, STENCIL}, RED, GREEN, BLUE, RG, RGB, RGBA, BGR, BGRA {RED, GREEN, BLUE, RG, RGB}_INTEGER, {RGBA, BGR, BGRA}_INTEGER [Table 3.3]

type: {HALF_}FLOAT, {UNSIGNED_}BYTE, {UNSIGNED_}SHORT, {UNSIGNED_}INT, FLOAT_32_UNSIGNED_INT_24_8_REV, and UNSIGNED_{BYTE, SHORT, INT}_* values from [Table 3.2]

void **ReadBuffer**(enum *src*);

src: NONE, FRONT{_LEFT, _RIGHT}, LEFT, RIGHT, BACK{_LEFT, _RIGHT}, FRONT_AND_BACK, AUX*i* (*i* = [0, AUX_BUFFERS - 1]), COLOR_ATTACHMENT*i* (*i* = [0, MAX_COLOR_ATTACHMENTS - 1])

Copying Pixels [4.3.2]

void **BlitFramebuffer**(int *srcX0*, int *srcY0*, int *srcX1*, int *srcY1*, int *dstX0*, int *dstY0*, int *dstX1*, int *dstY1*, bitfield *mask*, enum *filter*);

mask: Bitwise OR of {COLOR, DEPTH, STENCIL}_BUFFER_BIT

filter: LINEAR, NEAREST

> *Also see **DrawPixels, ClampColor, PixelZoom** in the Rasterization section of this card.*

Timer Queries [5.1]

Timer queries use query objects to track the amount of time needed to fully complete a set of GL commands.

void **QueryCounter**(uint *id*, TIMESTAMP);

void **GetInteger64v**(TIMESTAMP, int64 **data*);

Synchronization

Flush and Finish [5.2]

void **Flush**(void);

void **Finish**(void);

Sync Objects and Fences [5.3]

void **DeleteSync**(sync *sync*);

sync **FenceSync**(enum *condition*, bitfield *flags*);
condition: SYNC_GPU_COMMANDS_COMPLETE
flags: must be 0

Waiting for Sync Objects [5.3.1]

enum **ClientWaitSync**(sync *sync*, bitfield *flags*, uint64 *timeout_ns*);
flags: SYNC_FLUSH_COMMANDS_BIT, or zero

void **WaitSync**(sync *sync*, bitfield *flags*, uint64 *timeout_ns*);
timeout_ns: TIMEOUT_IGNORED

Sync Object Queries [6.1.8]

void **GetSynciv**(sync *sync*, enum *pname*, sizei *bufSize*, sizei **length*, int **values*);
pname: OBJECT_TYPE, SYNC_{STATUS, CONDITION, FLAGS}

boolean **IsSync**(sync *sync*);

State and State Requests

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

Simple Queries [6.1.1]

void **GetBooleanv**(enum *pname*, boolean **data*);

void **GetIntegerv**(enum *pname*, int **data*);

void **GetInteger64v**(enum *pname*, int64 **data*);

void **GetFloatv**(enum *pname*, float **data*);

void **GetDoublev**(enum *pname*, double **data*);

void **GetBooleani_v**(enum *target*, uint *index*, boolean **data*);

void **GetIntegeri_v**(enum *target*, uint *index*, int **data*);

void **GetFloati_v**(enum *target*, uint *index*, float **data*);

void **GetInteger64i_v**(enum *target*, uint *index*, int64 **data*);

boolean **IsEnabled**(enum *cap*);

boolean **IsEnabledi**(enum *target*, uint *index*);

String Queries [6.1.6]

ubyte ***GetString**(enum *name*);
name: RENDERER, VENDOR, VERSION, SHADING_LANGUAGE_VERSION

ubyte ***GetStringi**(enum *name*, uint *index*);
name: EXTENSIONS
index: range is [0, NUM_EXTENSIONS - 1]

Hints [5.4]

void **Hint**(enum *target*, enum *hint*);
target: FRAGMENT_SHADER_DERIVATIVE_HINT, TEXTURE_COMPRESSION_HINT, {LINE, POLYGON}_SMOOTH_HINT,
hint: FASTEST, NICEST, DONT_CARE

OpenGL Shading Language 4.20 Reference Card

The OpenGL® Shading Language is used to create shaders for each of the programmable processors contained in the OpenGL processing pipeline. The OpenGL Shading Language is actually several closely related languages. Currently, these processors are the vertex, tessellation control, tessellation evaluation, geometry, and fragment processors.

[n.n.n] and [Table n.n] refer to sections and tables in the OpenGL Shading Language 4.20 specification at www.opengl.org/registry

Preprocessor [3.3]

Preprocessor Directives

#	*#define*	*#elif*	*#if*	*#else*
#extension	*#version*	*#ifdef*	*#ifndef*	*#undef*
#error	*#include*	*#line*	*#endif*	*#pragma*

Preprocessor Operators

#version 420 #version 420 *profile*	Required when using version 4.20. *profile* indicates core or compatibility.
#extension *extension_name* : *behavior* #extension all : *behavior*	• *behavior*: require, enable, warn, disable • *extension_name:* extension supported by compiler, or "all"

Predefined Macros

__LINE__	__FILE__	Decimal integer constants. FILE says which source string number is being processed, or the path of the string if the string was an included string

GL_compatibility_profile	Integer 1 if the implementation supports the compatibility profile
__VERSION__	Decimal integer, e.g.: 420

Operators & Expressions [5.1]

The following operators are numbered in order of precedence. Relational and equality operators evaluate to Boolean. Also see lessThan(), equal(), etc.

1.	( )	parenthetical grouping
2.	[ ] ( ) . ++ --	array subscript function call, constructor, structure field, selector, swizzler postfix increment and decrement
3.	++ -- + - ~ !	prefix increment and decrement unary
4.	* / %	multiplicative
5.	+ -	additive
6.	<< >>	bit-wise shift
7.	< > <= >=	relational
8.	== !=	equality
9.	&	bit-wise and
10.	^	bit-wise exclusive or
11.	|	bit-wise inclusive or
12.	&&	logical and
13.	^^	logical exclusive or
14.	| |	logical inclusive or
15.	? :	selects an entire operand.
16.	= += -= *= /= %= <<= >>= &= ^= |=	assignment arithmetic assignments
17.	,	sequence

Vector & Scalar Components [5.5]

In addition to array numeric subscript syntax, names of vector and scalar components are denoted by a single letter. Components can be swizzled and replicated. Scalars have only an *x*, *r*, or *s* component.

{x, y, z, w}	Points or normals
{r, g, b, a}	Colors
{s, t, p, q}	Texture coordinates

Types [4.1]

Transparent Types

void	no function return value
bool	Boolean
int, uint	signed/unsigned integers
float	single-precision floating-point scalar
double	double-precision floating scalar
vec2, vec3, vec4	floating point vector
dvec2, dvec3, dvec4	double precision floating-point vectors
bvec2, bvec3, bvec4	Boolean vectors
ivec2, ivec3, ivec4 **uvec2, uvec3, uvec4**	signed and unsigned integer vectors
mat2, mat3, mat4	2x2, 3x3, 4x4 float matrix
mat2x2, mat2x3, mat2x4	2-column float matrix of 2, 3, or 4 rows
mat3x2, mat3x3, mat3x4	3-column float matrix of 2, 3, or 4 rows
mat4x2, mat4x3, mat4x4	4-column float matrix of 2, 3, or 4 rows
dmat2, dmat3, dmat4	2x2, 3x3, 4x4 double-precision float matrix
dmat2x2, dmat2x3, dmat2x4	2-column double-precision float matrix of 2, 3, 4 rows
dmat3x2, dmat3x3, dmat3x4	3-column double-precision float matrix of 2, 3, 4 rows
dmat4x2, dmat4x3, dmat4x4	4-column double-precision float matrix of 2, 3, 4 rows

Floating-Point Opaque Types

sampler[1,2,3]D	1D, 2D, or 3D texture
image[1,2,3]D	1D, 2D, or 3D image
samplerCube	cube mapped texture
imageCube	cube mapped image
sampler2DRect	rectangular texture
image2DRect	rectangular image
sampler[1,2]DShadow	[1,2]D depth tex./compare
sampler2DRectShadow	rectangular tex./compare
sampler[1,2]DArray	1D or 2D array texture
image[1,2]DArray	1D or 2D array image
sampler[1,2]DArrayShadow	1D or 2D array depth texture/comparison
samplerBuffer	buffer texture
imageBuffer	buffer image
sampler2DMS	2D multi-sample texture
image2DMS	2D multi-sample image
sampler2DMSArray	2D multi-sample array tex.
image2DMSArray	2D multi-sample array img.
samplerCubeArray	cube map array texture
imageCubeArray	cube map array image
samplerCubeArrayShadow	cube map array depth texture with comparison

Signed Integer Opaque Types

isampler[1,2,3]D	integer 1D, 2D, or 3D texture
iimage[1,2,3]D	integer 1D, 2D, or 3D image
isamplerCube	integer cube mapped texture

Continue ⮥

Signed Integer Opaque Types (cont'd)

iimageCube	integer cube mapped image
isampler2DRect	integer 2D rectangular texture
iimage2DRect	integer 2D rectangular image
isampler[1,2]DArray	integer 1D, 2D array texture
iimage[1,2]DArray	integer 1D, 2D array image
isamplerBuffer	integer buffer texture
iimageBuffer	integer buffer image
isampler2DMS	integer 2D multi-sample texture
iimage2DMS	integer 2D multi-sample image
isampler2DMSArray	int. 2D multi-sample array tex.
iimage2DMSArray	int. 2D multi-sample array image
isamplerCubeArray	integer cube map array texture
iimageCubeArray	integer cube map array image

Unsigned Integer Opaque Types

atomic_uint	uint atomic counter
usampler[1,2,3]D	uint 1D, 2D, or 3D texture
uimage[1,2,3]D	uint 1D, 2D, or 3D image
usamplerCube	uint cube mapped texture
uimageCube	uint cube mapped image
usampler2DRect	uint rectangular texture
uimage2DRect	uint rectangular image
usampler[1,2]DArray	1D or 2D array texture
uimage[1,2]DArray	1D or 2D array image
usamplerBuffer	uint buffer texture
uimageBuffer	uint buffer image
usampler2DMS	uint 2D multi-sample texture

Continue ⮥

Unsigned Integer Opaque Types (cont'd)

uimage2DMS	uint 2D multi-sample image
usampler2DMSArray	uint 2D multi-sample array tex.
uimage2DMSArray	uint 2D multi-sample array image
usamplerCubeArray	uint cube map array texture
uimageCubeArray	uint cube map array image

Implicit Conversions

int	->	uint
int, uint	->	float
int, uint, float	->	double
ivec2|3|4	->	uvec2|3|4
ivec2|3|4, uvec2|3|4	->	vec2|3|4
vec2|3|4	->	dvec2|3|4
ivec2|3|4, uvec2|3|4	->	dvec2|3|4
mat2|3|4	->	dmat2|3|4
mat2x3|2x4	->	dmat2x3|2x4
mat3x2|3x4	->	dmat3x2|3x4
mat4x2|4x3	->	dmat4x2|4x3

Aggregation of Basic Types

Arrays	**float**[3] foo; **float** foo[3]; structures, blocks, and structure members can be arrays

Structures	**struct** *type-name* {
	members
	} *struct-name*[];
	// optional variable declaration

Blocks	**in/out/uniform** *block-name* {
	// interface matching by block name
	optionally-qualified members
	} *instance-name*[];
	// optional instance name, optionally an array

Qualifiers

Storage Qualifiers [4.3]

Declarations may have one storage qualifier.

none	(default) local read/write memory, or input parameter
const	global compile-time constant, read-only func parameter, or read-only local variable
in	linkage into shader from previous stage
out	linkage out of a shader to next stage
uniform	linkage between a shader, OpenGL, and the application

Auxiliary Storage Qualifiers

Use to qualify some input and output variables:

centroid	centroid-based interpolation
sampler	per-sample interpolation
patch	per-tessellation-patch attributes

Uniform Qualifiers [4.3.5]

Declare global variables with same values across entire primitive processed. Examples:

uniform vec4 lightPosition;

uniform vec3 color = vec3(0.7, 0.7, 0.2);

Layout Qualifiers [4.4]

layout(*layout-qualifiers*) *block-declaration*

layout(*layout-qualifiers*) in/out/uniform

layout(*layout-qualifiers*) in/out/uniform *declaration*

Input Layout Qualifiers [4.4.1]

For all shader stages:

location = *integer-constant*

For tessellation evaluation shaders:

triangles, **quads**, **equal_spacing**, **isolines**, **fractional_{even,odd}_spacing**, **cw**, **ccw**, **point_mode**

For geometry shader inputs:

points, **lines**, **{lines,triangles}_adjacency**, **triangles**, **invocations** = *integer-constant*

For fragment shaders only for redeclaring built-in variable gl_FragCoord:

origin_upper_left, pixel_center_integer

For "in" only (not with variable declarations):

early_fragment_tests

Output Layout Qualifiers [4.4.2]

For all shader stages:

location = *integer-constant*
index = *integer-constant*

For tessellation control shaders:

vertices = *integer-constant*

For geometry shader outputs:

points, line_strip, triangle_strip,
max_vertices = *integer-constant,*
stream = *integer-constant*

Fragment shader outputs:

depth_any, **depth_greater**, **depth_less**, **depth_unchanged**

For fragment shaders:

index = *integer-constant*

Uniform-Block Layout Qualifiers [4.4.3]

Layout qualifier identifiers for uniform blocks:

shared, packed, std140, {row, column}_major
binding = *integer-constant*

Opaque Uniform Layout Qualifiers [4.4.4]

Used to bind opaque uniform variables to specific buffers or units.

binding = *integer-constant*

Atomic Counter Layout Qualifiers [4.4.4.1]

binding = *integer-constant*
offset = *integer-constant*

Format Layout Qualifiers [4.4.4.2]

One qualifier may be used with variables declared as "image" to specify the image format.

For tessellation control shaders:

binding = *integer-constant*,
rgba{32,16}f, **rg{32,16}f**, **r{32,16}f**, **rgba{16,8}**, **r11f_g11f_b10f**, **rgb10_a2{ui}**, **rg{16,8}**, **r{16,8}**, **rgba{32,16,8}i**, **rg{32,16,8}i**, **r{32,16,8}i**, **rgba{32,16,8}ui**, **rg{32,16,8}ui**, **r{32,16,8}ui**, **rgba{16,8}_snorm**, **rg{16,8}_snorm**, **r{16,8}_snorm**

Interpolation Qualifiers [4.5]

Qualify outputs from vertex shader and inputs to fragment shader.

smooth	perspective correct interpolation
flat	no interpolation
noperspective	linear interpolation

Parameter Qualifiers [4.6]

Input values copied in at function call time, output values copied out at function return.

none	(default) same as **in**
in	for function parameters passed into function
const	for function parameters that cannot be written to
out	for function parameters passed back out of function, but not initialized when passed in
inout	for function parameters passed both into and out of a function

Precision Qualifiers [4.7]

Precision qualifiers have no effect on precision; they aid code portability with OpenGL ES:

highp, **mediump**, **lowp**

Invariant Qualifiers Examples [4.8.1]

#pragma STDGL **invariant**(all)	force all output variables to be invariant
invariant gl_Position;	qualify a previously declared variable
invariant centroid out vec3 Color;	qualify as part of a variable declaration

Precise Qualifier [4.9]

Ensures that operations are executed in stated order with operator consistency. Requires two identical multiplies, followed by an add.

precise out vec4 Position = a * b + c * d;

Memory Qualifiers [4.10]

Variables qualified as "image" can have one or more memory qualifiers.

coherent	reads and writes are coherent with other shader invocations
volatile	underlying values may be changed by other sources
restrict	won't be accessed by other code
readonly	read only
writeonly	write only

Order of Qualification [4.11]

When multiple qualifiers are present in a declaration they may appear in any order, but must all appear before the type. The layout qualifier is the only qualifier that can appear more than once. Further, a declaration can have at most one storage qualifier, at most one auxiliary storage qualifier, and at most one interpolation qualifier. Multiple memory qualifiers can be used. Any violation of these rules will cause a compile-time error.

Operations and Constructors

Vector & Matrix [5.4.2]

.**length**() for matrices returns number of columns
.**length**() for vectors returns number of components

```
mat2(vec2, vec2);                     // 1 col./arg.
mat2x3(vec2, float, vec2, float);     // col. 2
dmat2(dvec2, dvec2);                  // 1 col./arg.
dmat3(dvec3, dvec3, dvec3);           // 1 col./arg.
```

Structure Example [5.4.3]

.**length**() for structures returns number of members

struct light {*members;* };
light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));

Array Example [5.4.4]

.**length**() for arrays returns number of elements

const float c[3] = float[3](5.0, b + 1.0, 1.1);

Matrix Examples [5.6]

Examples of access components of a matrix with array subscripting syntax:

```
mat4 m;              // m is a matrix
m[1] = vec4(2.0);    // sets 2nd col. to all 2.0
m[0][0] = 1.0;       // sets upper left element to 1.0
m[2][3] = 2.0;       // sets 4th element, 3rd col. to 2.0
```

Examples of operations on matrices and vectors:

```
m = f * m;        // scalar * matrix component-wise
v = f * v;        // scalar * vector component-wise
v = v * v;        // vector * vector component-wise
m = m +/- m;      // matrix +/- matrix comp.-wise
m = m * m;        // linear algebraic multiply
f = dot(v, v);    // vector dot product
v = cross(v, v);  // vector cross product
```

Structure & Array Operations [5.7]

Select structure fields or **length**() method of an array using the period (.) operator. Other operators:

.	field or method selector
== !=	equality
=	assignment
[ ]	indexing (arrays only)

Array elements are accessed using the array subscript operator ([ ]), e.g.:

diffuseColor += lightIntensity[3]*NdotL;

Statements and Structure

Iteration and Jumps [6.3-4]

Function	call by value-return

Iteration	for (;;) { break, continue }
	while ( ) { break, continue }
	do { break, continue } while ( );

Selection	if ( ) { }
	if ( ) { } else { }
	switch ( ) { case integer: ... break; ... default: ... }

Entry	void main()
Jump	break, continue, return (There is no 'goto')
Exit	return in main() discard // Fragment shader only

Subroutines [6.1.2]

Subroutine type variables are assigned to functions through the **UniformSubroutinesuiv** command in the OpenGL API.

Declare types with the **subroutine** keyword:

```
subroutine returnType subroutineTypeName(type0
    arg0,
    type1 arg1, ..., typen argn);
```

Associate functions with subroutine types of matching declarations by defining the functions with the subroutine keyword and a list of subroutine types the function matches:

```
subroutine(subroutineTypeName0, ...,
    subroutineTypeNameN)
returnType functionName(type0 arg0,
    type1 arg1, ..., typen argn){ ... }
    // function body
```

Declare subroutine type variables with a specific subroutine type in a subroutine uniform variable declaration:

```
subroutine uniform subroutineTypeName
    subroutineVarName;
```

Built-In Variables [7]

Shaders communicate with fixed-function OpenGL pipeline stages and other shader executables through built-in variables.

Vertex Language

Inputs:

```
in int gl_VertexID;
in int gl_InstanceID;
```

Outputs:

```
gl_PerVertex {
    vec4  gl_Position;
    float  gl_PointSize;
    float  gl_ClipDistance[];
};
```

Tessellation Control Language

Inputs:

```
in gl_PerVertex {
    vec4  gl_Position;
    float  gl_PointSize;
    float  gl_ClipDistance[];
} gl_in[];

  in  int gl_PatchVerticesIn;
  in  int gl_PrimitiveID;
  in  int gl_InvocationID;
```

Outputs:

```
out gl_PerVertex {
    vec4  gl_Position;
    float  gl_PointSize;
    float  gl_ClipDistance[];
} gl_out[];

patch out  float  gl_TessLevelOuter[4];
patch out  float  gl_TessLevelInner[2];
```

Tessellation Evaluation Language

Inputs:

```
in gl_PerVertex {
    vec4  gl_Position;
    float  gl_PointSize;
    float  gl_ClipDistance[];
} gl_in[];
```

(more ↑)

```
in  int      gl_PatchVerticesIn;
in  int      gl_PrimitiveID;
in  vec3   gl_TessCoord;
patch in    float  gl_TessLevelOuter[4];
patch in    float  gl_TessLevelInner[2];
```

Outputs:

```
out  gl_PerVertex {
    vec4  gl_Position;
    float  gl_PointSize;
    float  gl_ClipDistance[];
};
```

Geometry Language

Inputs:

```
in gl_PerVertex {
    vec4  gl_Position;
    float  gl_PointSize;
    float  gl_ClipDistance[];
} gl_in[];

in int gl_PrimitiveIDIn;
in int gl_InvocationID;
```

Outputs:

```
out gl_PerVertex {
    vec4  gl_Position;
    float  gl_PointSize;
    float  gl_ClipDistance[];
};

out  int  gl_PrimitiveID;
out  int  gl_Layer;
out  int  gl_ViewportIndex;
```

Fragment Language

Inputs:

```
in  vec4  gl_FragCoord;
in  bool  gl_FrontFacing;
in  float  gl_ClipDistance[];
in  vec2  gl_PointCoord;
in  int     gl_PrimitiveID;
in  int     gl_SampleID;
in  vec2  gl_SamplePosition;
in  int     gl_SampleMask[];
```

Outputs:

```
out  float gl_FragDepth;
out  int    gl_SampleMask[];
```

(Built-In Variables Continue >)

Built-In Variables (cont.)

Built-In Constants [7.3]

The following are provided to all shaders. The actual values are implementation-dependent, but must be at least the value shown.

```
const  int gl_MaxVertexAttribs = 16;
const  int gl_MaxVertexUniformComponents = 1024;
const int  gl_MaxVaryingComponents = 60;
const  int gl_MaxVertexOutputComponents = 64;
const  int gl_MaxGeometryInputComponents = 64;
const  int gl_MaxGeometryOutputComponents = 128;
const  int gl_MaxFragmentInputComponents = 128;
const  int gl_MaxVertexTextureImageUnits = 16;
const  int gl_MaxCombinedTextureImageUnits = 80;
const  int gl_MaxTextureImageUnits = 16;
const  int gl_MaxImageUnits = 8;
const  int
   gl_MaxCombinedImageUnitsAndFragmentOutputs = 8;
const  int gl_MaxImageSamples = 0;
const  int gl_MaxFragmentUniformComponents = 1024;
const  int gl_MaxDrawBuffers = 8;
const  int gl_MaxClipDistances = 8;
const  int gl_MaxGeometryTextureImageUnits = 16;
const  int gl_MaxGeometryOutputVertices = 256;
const  int gl_MaxGeometryTotalOutputComponents = 1024;
const  int gl_MaxGeometryUniformComponents = 1024;
const  int gl_MaxGeometryVaryingComponents = 64;
const  int gl_MaxTessControlInputComponents = 128;
const  int gl_MaxTessControlOutputComponents = 128;
const  int gl_MaxTessControlTextureImageUnits = 16;
const  int gl_MaxTessControlUniformComponents = 1024;
const  int gl_MaxTessControlTotalOutputComponents = 4096;
const  int gl_MaxTessEvaluationInputComponents = 128;
const  int gl_MaxTessEvaluationOutputComponents = 128;
const  int gl_MaxTessEvaluationTextureImageUnits = 16;
const  int gl_MaxTessEvaluationUniformComponents = 1024;
const  int gl_MaxTessPatchComponents = 120;
const  int gl_MaxPatchVertices = 32;
const  int gl_MaxTessGenLevel = 64;
const  int gl_MaxViewports = 16;
const  int gl_MaxVertexUniformVectors = 256;
const  int gl_MaxFragmentUniformVectors = 256;
const  int gl_MaxVaryingVectors = 15;
const  int gl_MaxVertexAtomicCounters = 0;
const  int gl_MaxTessControlAtomicCounters = 0;
const  int gl_MaxTessEvaluationAtomicCounters = 0;
const  int gl_MaxGeometryAtomicCounters = 0;
const  int gl_MaxFragmentAtomicCounters = 8;
const  int gl_MaxCombinedAtomicCounters = 8;
const  int gl_MaxAtomicCounterBindings = 1;
const  int gl_MinProgramTexelOffset = -7;
const  int gl_MaxProgramTexelOffset = 8;
```

Built-In Functions

Type Abbreviations for Built-in Functions:

Tf=float, vec*n*. Td =double, dvec*n*. Tfd= float, vec*n*, double, dvec*n*. Tb=bool, bvec*n*. Tvec=vec*n*, uvec*n*, ivec*n*.
Tu=uint, uvec*n*. Ti=int, ivec*n*. Tiu=int, ivec*n*, uint, uvec*n*.

Use of T*n* or T*nn* within each function call must be the same. In vector types, *n* is 2, 3, or 4.

Angle & Trig. Functions [8.1]

Functions will not result in a divide-by-zero error. If the divisor of a ratio is 0, then results will be undefined. Component-wise operation. Parameters specified as *angle* are in units of radians. Tf=float, vec*n*.

Tf **radians**(Tf *degrees*)	degrees to radians
Tf **degrees**(Tf *radians*)	radians to degrees
Tf **sin**(Tf *angle*)	sine
Tf **cos**(Tf *angle*)	cosine
Tf **tan**(Tf *angle*)	tangent
Tf **asin**(Tf *x*)	arc sine
Tf **acos**(Tf *x*)	arc cosine
Tf **atan**(Tf *y*, Tf *x*) Tf **atan**(Tf *y_over_x*)	arc tangent
Tf **sinh**(Tf *x*)	hyperbolic sine
Tf **cosh**(Tf *x*)	hyperbolic cosine
Tf **tanh**(Tf *x*)	hyperbolic tangent
Tf **asinh**(Tf *x*)	hyperbolic sine
Tf **acosh**(Tf *x*)	hyperbolic cosine
Tf **atanh**(Tf *x*)	hyperbolic tangent

Exponential Functions [8.2]

Component-wise operation. Tf=float, vec*n*. Tfd= float, vec*n*, double, dvec*n*.

Tf **pow**(Tf *x*, Tf *y*)	x^y
Tf **exp**(Tf *x*)	e^x
Tf **log**(Tf *x*)	ln
Tf **exp2**(Tf *x*)	2^x
Tf **log2**(Tf *x*)	$\log_2$
Tfd **sqrt**(Tfd *x*)	square root
Tfd **inversesqrt**(Tfd *x*)	inverse square root

Common Functions [8.3]

Component-wise operation. Tf=float, vec*n*. Tfd= float, vec*n*, double, dvec*n*.

Tfd **abs**(Tfd *x*) Ti **abs**(Ti *x*)	absolute value
Tfd **sign**(Tfd *x*) Ti **sign**(Ti *x*)	returns -1.0, 0.0, or 1.0
Tfd **floor**(Tfd *x*)	nearest integer <= *x*
Tfd **trunc**(Tfd *x*)	nearest integer with absolute value <= absolute value of *x*
Tfd **round**(Tfd *x*)	nearest integer, implementation-dependent rounding mode
Tfd **roundEven**(Tfd *x*)	nearest integer, 0.5 rounds to nearest even integer
Tfd **ceil**(Tfd *x*)	nearest integer >= *x*
Tfd **fract**(Tfd *x*)	*x* - floor(*x*)
Tfd **mod**(Tfd *x*, Tfd *y*) Tf **mod**(Tf *x*, float *y*) Td **mod**(Td *x*, double *y*)	modulus
Tfd **modf**(Tfd *x*, out Tfd *i*)	separate integer and fractional parts
Tfd **min**(Tfd *x*, Tfd *y*) Tf **min**(Tf *x*, float *y*) Td **min**(Td *x*, double *y*) Tiu **min**(Tiu *x*, Tiu *y*) Ti **min**(Ti *x*, int *y*) Tu **min**(Tu *x*, uint *y*)	minimum value

(Common Functions continue ↑)

Common Functions (continued)

Tfd **max**(Tfd *x*, Tfd *y*) Tf **max**(Tf *x*, float *y*) Td **max**(Td *x*, double *y*) Tiu **max**(Tiu *x*, Tiu *y*) Ti **max**(Ti *x*, int *y*) Tu **max**(Tu *x*, uint *y*)	maximum value
Tfd **mix**(Tfd *x*, Tfd *y*, Tfd *a*) Tf **mix**(Tf *x*, Tf *y*, float *a*) Td **mix**(Td *x*, Td *y*, double *a*)	linear blend of *x* and *y*
Tfd **mix**(Tfd *x*, Tfd *y*, Tb *a*)	true if comps. in *a* select comps. from *y*, else from *x*
Tfd **step**(Tfd *edge*, Tfd *x*) Tf **step**(float *edge*, Tf *x*) Td **step**(double *edge*, Td *x*)	0.0 if *x* < *edge*, else 1.0
Tb **isnan**(Tfd *x*)	true if *x* is NaN
Tb **isinf**(Tfd *x*)	true if *x* is positive or negative infinity
Tfd **clamp**(Tfd *x*, Tfd *minVal*, Tfd *maxVal*) Tf **clamp**(Tf *x*, float *minVal*, float *maxVal*) Td **clamp**(Td *x*, double *minVal*, double *maxVal*) Tiu **clamp**(Tiu *x*, Tiu *minVal*, Tiu *maxVal*) Ti **clamp**(Ti *x*, int *minVal*, int *maxVal*) Tu **clamp**(Tu *x*, uint *minVal*, uint *maxVal*)	min(max(*x*, *minVal*), *maxVal*)
Tfd **smoothstep**(Tfd *edge0*, Tfd *edge1*, T *x*) Tf **smoothstep**(float *edge0*, float *edge1*, Tf *x*) Td **smoothstep**(double *edge0*, double *edge1*, Td *x*)	clip and smooth
Ti **floatBitsToInt**(Tf *value*) Tu **floatBitsToInt**(Tf *value*)	Returns signed int or uint value representing the encoding of a floating-point value
Tf **intBitsToFloat**(Tiu *value*)	Returns floating-point value of a signed int or uint encoding of a floating-point value
Tfd **fma**(Tfd *a*, Tfd *b*, Tfd *c*)	Computes and returns a*b + c. Treated as a single operation when using **precise**
Tfd **frexp**(Tfd *x*, out Ti *exp*)	Splits *x* into a floating-point significand in the range [0.5, 1.0) and an integral exponent of 2
Tfd **ldexp**(Tfd *x*, in Ti *exp*)	Builds a floating-point number from *x* and the corresponding integral exponent of 2 in *exp*.

Floating-Point Pack/Unpack [8.4]

These do not operate component-wise.

uint **packUnorm2x16**(vec2 *v*) uint **packSnorm2x16**(vec2 *v*) uint **packUnorm4x8**(vec4 *v*) uint **packSnorm4x8**(vec4 *v*)	Converts each comp. of *v* into 8- or 16-bit ints, packs results into the returned 32-bit unsigned integer

(Pack/Unpack Functions continue ↑)

Pack/Unpack Functions (continued)

vec2 **unpackUnorm2x16**(uint *p*) vec2 **unpackSnorm2x16**(uint *p*) vec4 **unpackUnorm4x8**(uint *p*) vec4 **unpackSnorm4x8**(uint *p*)	Unpacks 32-bit *p* into two 16-bit uints, four 8-bit uints, or signed ints. Then converts each component to a normalized float to generate a 2- or 4-component vector
double **packDouble2x32**(uvec2 *v*)	Packs components of *v* into a 64-bit value and returns a double-precision value
uvec2 **unpackDouble2x32**(double *v*)	Returns a 2-component vector representation of *v*
uint **packHalf2x16**(vec2 *v*)	Returns a uint by converting the components of a two-component floating-point vector
vec2 **unpackHalf2x16**(uint *v*)	Returns a two-component floating-point vector

Geometric Functions [8.5]

These functions operate on vectors as vectors, not component-wise. Tf=float, vec*n*. Td =double, dvec*n*. Tfd= float, vec*n*, double, dvec*n*.

float **length**(Tf *x*) double **length**(Td *x*)	length of vector
float **distance**(Tf *p0*, Tf *p1*) double **distance**(Td *p0*, Td *p1*)	distance between points
float **dot**(Tf *x*, Tf *y*) double **dot**(Td *x*, Td *y*)	dot product
vec3 **cross**(vec3 *x*, vec3 *y*) dvec3 **cross**(dvec3 *x*, dvec3 *y*)	cross product
Tfd **normalize**(Tfd *x*)	normalize vector to length 1
Tfd **faceforward**(Tfd *N*, Tfd *I*, Tfd *Nref*)	returns *N* if dot(*Nref*, *I*) < 0, else -*N*
Tfd **reflect**(Tfd *I*, Tfd *N*)	reflection direction *I* - 2 * dot(*N*,*I*) * *N*
Tfd **refract**(Tfd *I*, Tfd *N*, float *eta*)	refraction vector

Matrix Functions [8.6]

For the matrix functions, type *mat* is used in the single-precision floating point functions, and type *dmat* is used in the double-precision floating point functions. *N* and *M* are 1, 2, 3, 4.

mat **matrixCompMult**(mat *x*, mat *y*) dmat **matrixCompMult**(dmat *x*, dmat *y*)	component-wise multiply
mat*N* **outerProduct**(vec*N* *c*, vec*N* *r*) dmat*N* **outerProduct**(dvec*N* *c*, dvec*N* *r*)	outer product (where *N* != *M*)
mat*NxM* **outerProduct**(vec*M* *c*, vec*N* *r*) dmat*NxM* **outerProduct**(dvec*M* *c*, dvec*N* *r*)	outer product
mat*N* **transpose**(mat*N* *m*) dmat*N* **transpose**(dmat*N* *m*)	transpose

(Matrix Functions continue ↑)

Matrix Functions (continued)

mat*NxM* **transpose**(mat*MxN* *m*) dmat*NxM* **transpose**(dmat*MxN* *m*)	transpose (where *N* != *M*)
float **determinant**(mat*N* *m*) double **determinant**(dmat*N* *m*)	determinant
mat*N* **inverse**(mat*N* *m*) dmat*N* **inverse**(dmat*N* *m*)	inverse

Vector Relational Functions [8.7]

Compare *x* and *y* component-wise. Sizes of the input and return vectors for any particular call must match. Tvec=vec*n*, uvec*n*, ivec*n*.

bvec*n* **lessThan**(Tvec *x*, Tvec *y*)	<
bvec*n* **lessThanEqual**(Tvec *x*, Tvec *y*)	<=
bvec*n* **greaterThan**(Tvec *x*, Tvec *y*)	>
bvec*n* **greaterThanEqual**(Tvec *x*, Tvec *y*)	>=
bvec*n* **equal**(Tvec *x*, Tvec *y*) bvec*n* **equal**(bvec*n* *x*, bvec*n* *y*)	==
bvec*n* **notEqual**(Tvec *x*, Tvec *y*) bvec*n* **notEqual**(bvec*n* *x*, bvec*n* *y*)	!=
bool **any**(bvec*n* *x*)	true if any component of *x* is true
bool **all**(bvec*n* *x*)	true if all components of *x* are true
bvec*n* **not**(bvec*n* *x*)	logical complement of *x*

Integer Functions [8.8]

Component-wise operation. Tu=uint, uvec*n*. Ti=int, ivec*n*. Tiu=int, ivec*n*, uint, uvec*n*.

Tu **uaddCarry**(Tu *x*, Tu *y*, out Tu *carry*)	Adds 32-bit uint *x* and *y*, returning the sum modulo 2^{32}
Tu **usubBorrow**(Tu *x*, Tu *y*, out Tu *borrow*)	Subtracts *y* from *x*, returning the difference if non-negative, otherwise 2^{32} plus the difference
void **umulExtended**(Tu *x*, Tu *y*, out Tu *msb*, out Tu *lsb*) void **imulExtended**(Ti *x*, Ti *y*, out Ti *msb*, out Ti *lsb*)	Multiplies 32-bit integers *x* and *y*, producing a 64-bit result
Tiu **bitfieldExtract**(Tiu *value*, int *offset*, int *bits*)	Extracts bits [*offset*, *offset* + *bits* - 1] from *value*, returns them in the least significant bits of the result
Tiu **bitfieldInsert**(Tiu *base*, Tiu *insert*, int *offset*, int *bits*)	Returns the insertion the *bits* least-significant bits of *insert* into *base*
Tiu **bitfieldReverse**(Tiu *value*)	Returns the reversal of the bits of *value*
Ti **bitCount**(Tiu *value*)	Returns the number of bits set to 1
Ti **findLSB**(Tiu *value*)	Returns the bit number of the least significant bit set to 1
Ti **findMSB**(Tiu *value*)	Returns the bit number of the most significant bit

(Built-In Functions Continue >)

Built-In Functions (cont.)

Texture Lookup Functions [8.9]

Available to vertex, geometry, and fragment shaders. See Texture Function tables below.

Atomic-Counter Functions [8.10]

Returns the value of an atomic counter.

uint **atomicCounterIncrement**(atomic_uint *c*)	Atomically returns the value of counter for *c*, then increments.
uint **atomicCounterDecrement**(atomic_uint *c*)	Atomically decrements counter for *c*, then returns value of counter for *c*.
uint **atomicCounter**(atomic_uint *c*)	Atomically returns the counter for *c*.

Image Functions [8.11]

In these image functions, *IMAGE_PARAMS* may be one of the following:

gimage{1D, Buffer} *image*, int *P*
gimage{2D[Rect], 1DArray} *image*, ivec2 *P*
gimage{3D, Cube[Array], 2DArray} *image*, ivec3 *P*
gimage2DMS *image*, ivec2 *P*, int *sample*
gimage2DMSArray *image*, ivec3 *P*, int *sample*

gvec4 **imageLoad**(readonly *IMAGE_PARAMS*)	Loads the texel at the coordinate *P* from the image unit *image*.
void **imageStore**(writeonly *IMAGE_PARAMS*, gvec4 *data*)	Stores *data* into the texel at the coordinate *P* from the image specified by *image*.

(Image Functions continue ⤴)

Image Functions (continued)

uint **imageAtomicAdd**(*IMAGE_PARAMS*, uint *data*) int **imageAtomicAdd**(*IMAGE_PARAMS*, int *data*)	Adds the value of *data* to the contents of the selected texel.
uint **imageAtomicMin**(*IMAGE_PARAMS*, uint *data*) int **imageAtomicMin**(*IMAGE_PARAMS*, int *data*)	Takes the minimum of the value of *data* and the contents of the selected texel.
uint **imageAtomicMax**(*IMAGE_PARAMS*, uint *data*) int **imageAtomicMax**(*IMAGE_PARAMS*, int *data*)	Takes the maximum of the value *data* and the contents of the selected texel.
uint **imageAtomicAnd**(*IMAGE_PARAMS*, uint *data*) int **imageAtomicAnd**(*IMAGE_PARAMS*, int *data*)	Performs a bit-wise AND of the value of *data* and the contents of the selected texel.
uint **imageAtomicOr**(*IMAGE_PARAMS*, uint *data*) int **imageAtomicOr**(*IMAGE_PARAMS*, int *data*)	Performs a bit-wise OR of the value of *data* and the contents of the selected texel.
uint **imageAtomicXor**(*IMAGE_PARAMS*, uint *data*) int **imageAtomicXor**(*IMAGE_PARAMS*, int *data*)	Performs a bit-wise EXCLUSIVE OR of the value of *data* and the contents of the selected texel.
uint **imageAtomicExchange**(*IMAGE_PARAMS*, uint *data*) int **imageAtomicExchange**(*IMAGE_PARAMS*, int *data*)	Copies the value of *data*.

(Image Functions continue ⤴)

Image Functions (continued)

uint **imageAtomicCompSwap**(*IMAGE_PARAMS*, uint *compare*, uint *data*) int **imageAtomicCompSwap**(*IMAGE_PARAMS*, int *compare*, int *data*)	Compares the value of *compare* and contents of selected texel. If equal, the new value is given by *data*; otherwise, it is taken from the original value loaded from texel.

Fragment Processing Functions [8.12]

Available only in fragment shaders.
Tf=float, vec*n*.

Derivative fragment-processing functions

Tf **dFdx**(Tf *p*)	derivative in *x*
Tf **dFdy**(Tf *p*)	derivative in *y*
Tf **fwidth**(Tf *p*)	sum of absolute derivative in *x* and *y*: abs (dFdx (*p*)) + abs (dFdy (*p*));

Interpolation fragment-processing functions

Tf **interpolateAtCentroid**(Tf *interpolant*)	Return value of *interpolant* sampled inside pixel and the primitive.
Tf **interpolateAtSample**(Tf *interpolant*, int *sample*)	Return value of *interpolant* at the location of sample number *sample*.
Tf **interpolateAtOffset**(Tf *interpolant*, vec2 *offset*)	Return value of *interpolant* sampled at fixed offset *offset* from pixel center.

Noise Functions [8.13]

Returns noise value. Available to fragment, geometry, and vertex shaders.

float **noise1**(Tf *x*)	
vec*n* **noisen**(Tf *x*)	where *n* is 2, 3, or 4

Geometry Shader Functions [8.14]

Only available in geometry shaders.

void **EmitStreamVertex**(int *stream*)	Emits values of output variables to current output primitive stream *stream*.
void **EndStreamPrimitive**(int *stream*)	Completes current output primitive stream *stream* and starts a new one.
void **EmitVertex**()	Emits values of output variables to the current output primitive.
void **EndPrimitive**()	Completes output primitive and starts a new one.

Other Shader Functions [8.15-16]

void **barrier**()	Shader Invocation: Synchronizes across shader invocations.
void **memoryBarrier**()	Shader Memory Control: Control the ordering of memory transactions issued by a single shader invocation.

Texture Functions [8.9]

Available to vertex, geometry, and fragment shaders. gvec4=vec4, ivec4, uvec4. gsampler* = sampler*, isampler*, usampler*.

The *P* argument needs to have enough components to specify each dimension, array layer, or comparison for the selected sampler. The *dPdx* and *dPdy* arguments need enough components to specify the derivative for each dimension of the sampler.

Texture Query Functions [8.9.1]

textureSize functions return dimensions of *lod* (if present) for the texture bound to sampler. Components in return value are filled in with the width, height, depth of the texture. For array forms, the last component of the return value is the number of layers in the texture array.

{int,ivec2,ivec3} **textureSize**(gsampler{1D[Array],2D[Rect,Array],Cube} *sampler*[, int *lod*])

{int,ivec2,ivec3} **textureSize**(gsampler{Buffer,2DMS[Array]}*sampler*)

{int,ivec2,ivec3} **textureSize**(sampler{1D,2D[Rect],Cube}[Array]Shadow *sampler*[, int *lod*])

ivec3 **textureSize**(samplerCubeArray *sampler*, int *lod*)

textureQueryLod functions return the mipmap array(s) that would be accessed in the *x* component of the return value. Returns the computed level of detail relative to the base level in the *y* component of the return value.

vec2 **textureQueryLod**(gsampler{1D[Array],2D[Array],3D,Cube[Array]} *sampler*, {float,vec2,vec3} *P*)

vec2 **textureQueryLod**(sampler{1D[Array],2D[Array],Cube[Array]}Shadow *sampler*, {float,vec2,vec3} *P*)

Texel Lookup Functions [8.9.2]

Use texture coordinate *P* to do a lookup in the texture bound to *sampler*. For shadow forms, *compare* is used as D_{ref} and the array layer comes from *P*.w. For non-shadow forms, the array layer comes from the last component of *P*.

gvec4 **texture**(gsampler{1D[Array],2D[Array,Rect],3D,Cube[Array]} *sampler*, {float,vec2,vec3,vec4} *P* [, float *bias*])

(more ⤴)

float **texture**(gsamplerCubeArrayShadow *sampler*, vec4 *P*, float *compare*)

float **texture**(sampler{1D[Array],2D[Array,Rect],Cube}Shadow *sampler*, {vec3,vec4} *P* [, float *bias*])

Texture lookup with projection.

gvec4 **textureProj**(gsampler{1D,2D[Rect],3D} *sampler*, vec{2,3,4} *P* [, float *bias*])

float **textureProj**(sampler{1D,2D[Rect]}Shadow *sampler*, vec4 *P* [, float *bias*])

Texture lookup as in **texture** but with explicit LOD.

gvec4 **textureLod**(gsampler{1D[Array],2D[Array],3D,Cube[Array]} *sampler*, {float,vec2,vec3} *P*, float *lod*)

float **textureLod**(sampler{1D[Array],2D}Shadow *sampler*, vec3 *P*, float *lod*)

Offset added before texture lookup as in **texture**.

gvec4 **textureOffset**(gsampler{1D[Array],2D[Array,Rect],3D} *sampler*, {float,vec2,vec3} *P*, {int,ivec2,ivec3} *offset* [, float *bias*])

float **textureOffset**(sampler{1D[Array],2D[Rect]}Shadow *sampler*, vec3 *P*, {int,ivec2} *offset* [, float *bias*])

Use integer texture coordinate *P* to lookup a single texel from *sampler*.

gvec4 **texelFetch**(gsampler{1D[Array],2D[Array,Rect],3D} *sampler*, {int,ivec2,ivec3} *P*[, {int,ivec2} *lod*])

gvec4 **texelFetch**(gsampler{Buffer, 2DMS[Array]} *sampler*, {ivec2,ivec3} *P*, int *sample*)

Fetch single texel with *offset* added before texture lookup.

gvec4 **texelFetchOffset**(gsampler{1D[Array],2D[Array],3D} *sampler*, {int,ivec2,ivec3} *P*, int *lod*, {int, ivec2, ivec3} *offset*)

gvec4 **texelFetchOffset**(gsampler2DRect *sampler*, ivec2 *P*, ivec2 *offset*)

Projective texture lookup with *offset* added before texture lookup.

gvec4 **textureProjOffset**(gsampler{1D,2D[Rect],3D} *sampler*, vec{2,3,4} *P*, {int,ivec2,ivec3} *offset* [, float *bias*])

float **textureProjOffset**(sampler{1D,2D[Rect]}Shadow *sampler*, vec4 *P*, {int,ivec2} *offset* [, float *bias*])

Offset texture lookup with explicit LOD.

gvec4 **textureLodOffset**(gsampler{1D[Array],2D[Array],3D} *sampler*, {float,vec2,vec3} *P*, float *lod*, {int,ivec2,ivec3} *offset*)

float **textureLodOffset**(sampler{1D[Array],2D}Shadow *sampler*, vec3 *P*, float *lod*, {int,ivec2} *offset*)

Projective texture lookup with explicit LOD.

gvec4 **textureProjLod**(gsampler{1D,2D,3D} *sampler*, vec{2,3,4} *P*, float *lod*)

float **textureProjLod**(sampler{1D,2D}Shadow *sampler*, vec4 *P*, float *lod*)

Offset projective texture lookup with explicit LOD.

gvec4 **textureProjLodOffset**(gsampler{1D,2D,3D} *sampler*, vec{2,3,4} *P*, float *lod*, {int, ivec2, ivec3} *offset*)

float **textureProjLodOffset**(sampler{1D,2D}Shadow *sampler*, vec4 *P*, float *lod*, {int, ivec2} *offset*)

Texture lookup as in **texture** but with explicit gradients.

gvec4 **textureGrad**(gsampler{1D[Array],2D[Rect,Array],3D,Cube[Array]} *sampler*, {float, vec2, vec3,vec4} *P*, {float, vec2, vec3} *dPdx*, {float, vec2, vec3} *dPdy*)

float **textureGrad**(sampler{1D[Array], 2D[Rect, Array]}Shadow *sampler*, {vec3, vec4} *P*, {float, vec2} *dPdx*, {float, vec2} *dPdy*)

Texture lookup with both explicit gradient and offset.

gvec4 **textureGradOffset**(gsampler{1D[Array], 2D[Rect, Array], 3D} *sampler*, {float, vec2, vec3} *P*, {float, vec2, vec3} *dPdx*, {float, vec2, vec3} *dPdy*, {int, ivec2, ivec3} *offset*)

float **textureGradOffset**(sampler{1D[Array],2D[Rect, Array]}Shadow *sampler*, {vec3, vec4} *P*, {float, vec2} *dPdx*, {float, vec2} *dPdy*, {int, ivec2} *offset*)

Texture lookup both projectively as in **textureProj**, and with explicit gradient as in **textureGrad**.

gvec4 **textureProjGrad**(gsampler{1D,2D[Rect],3D} *sampler*, {vec2,vec3,vec4} *P*, {float,vec2,vec3} *dPdx*, {float,vec2,vec3} *dPdy*)

float **textureProjGrad**(sampler{1D,2D[Rect]}Shadow *sampler*, vec4 *P*, {float,vec2} *dPdx*, {float,vec2} *dPdy*)

Texture lookup projectively and with explicit gradient as in **textureProjGrad**, as well as with offset as in **textureOffset.**

gvec4 **textureProjGradOffset**(gsampler{1D,2D[Rect],3D} *sampler*, vec{2,3,4} *P*, {float,vec2,vec3} *dPdx*, {float,vec2,vec3} *dPdy*, {int,ivec2,ivec3} *offset*)

float **textureProjGradOffset**(sampler{1D,2D[Rect]Shadow *sampler*, vec4 *P*, {float,vec2} *dPdx*, {float,vec2} *dPdy*, {ivec2,int,vec2} *offset*)

Texture Gather Instructions [8.9.3]

These functions take components of a floating-point vector operand as a texture coordinate, determine a set of four texels to sample from the base level of detail of the specified texture image, and return one component from each texel in a four-component result vector.

gvec4 **textureGather**(gsampler{2D[Array,Rect],Cube[Array]} *sampler*, {vec2,vec3} *P* [, int *comp*])

vec4 **textureGather**(sampler{2D[Array,Rect],Cube[Array]}Shadow *sampler*, {vec2,vec3,vec4} *P*, float *refZ*)

Texture gather as in **textureGather** by offset as described in **textureOffset** except minimum and maximum offset values are given by {MIN, MAX}_PROGRAM_TEXTURE_GATHER_OFFSET.

gvec4 **textureGatherOffset**(gsampler2D[Array,Rect] *sampler*, {vec2,vec3} *P*, ivec2 *offset* [, int *comp*])

vec4 **textureGatherOffset**(sampler2D[Array,Rect]Shadow *sampler*, {vec2,vec3} *P*, float *refZ*, ivec2 *offset*)

Texture gather as in **textureGatherOffset** except *offsets* determines location of the four texels to sample.

gvec4 **textureGatherOffsets**(gsampler2D[Array,Rect] *sampler*, {vec2,vec3} *P*, ivec2 *offset*[4] [, int *comp*])

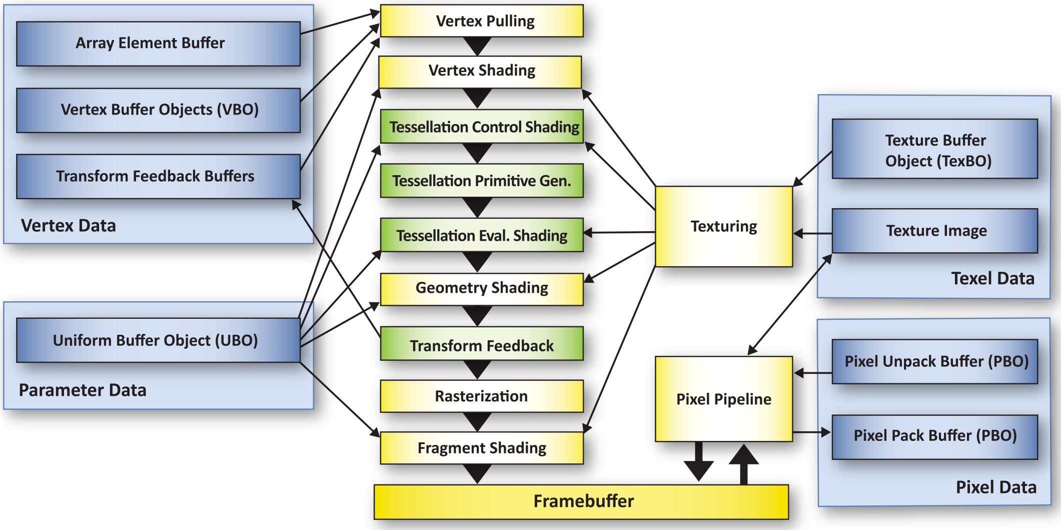
vec4 **textureGatherOffsets**(sampler2D[Array,Rect]Shadow *sampler*, {vec2,vec3} *P*, float *refZ*, ivec2 *offset*[4])

OpenGL Pipeline

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Calls are made to allocate a GL context which is then associated with the window, then OpenGL commands can be issued.

The heavy black arrows in this illustration show the OpenGL pipeline. In order to fully take advantage of modern OpenGL, pay close attention to how to most efficiently use the new buffer types.

- 🟦 Blue blocks indicate various buffers that feed or get fed by the OpenGL pipeline.
- 🟩 Green blocks indicate features new or significantly changed with OpenGL 4.x.

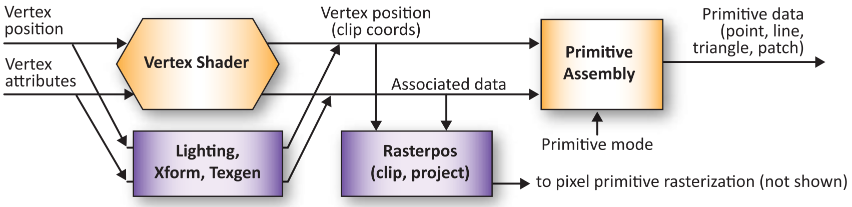


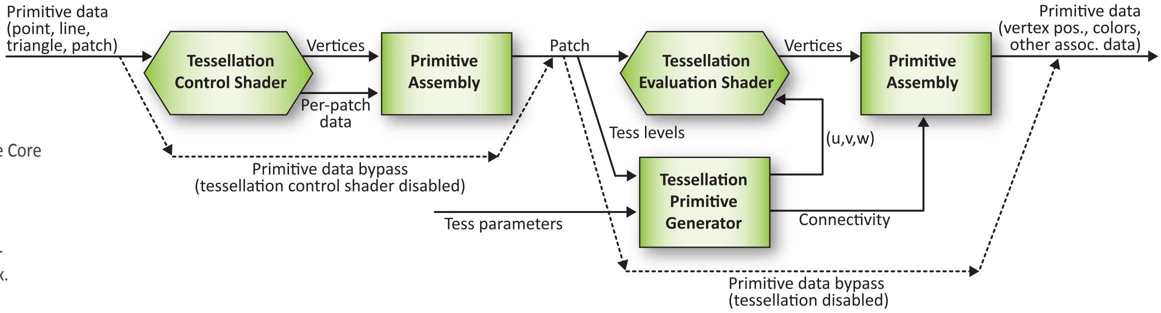
Vertex & Tessellation Details

Each vertex is processed either by a vertex shader or fixed-function vertex processing (compatibility only) to generate a transformed vertex, then assembled into primitives. Tessellation (if enabled) operates on patch primitives, consisting of a fixed-size collection of vertices, each with per-vertex attributes and associated per-patch attributes. Tessellation control shaders (if enabled) transform an input patch and compute per-vertex and per-patch attributes for a new output patch.

A fixed-function primitive generator subdivides the patch according to tessellation levels computed in the tessellation control shaders or specified as fixed values in the API (TCS disabled). The tessellation evaluation shader computes the position and attributes of each vertex produced by the tessellator.

- 🟧 Orange blocks indicate features of the Core specification.
- 🟪 Purple blocks indicate features of the Compatibility specification.
- 🟩 Green blocks indicate features new or significantly changed with OpenGL 4.x.


Vertex position
Vertex attributes
Vertex Shader
Lighting, Xform, Texgen
Vertex position (clip coords)
Associated data
Rasterpos (clip, project)
Primitive Assembly
Primitive mode
Primitive data (point, line, triangle, patch)
to pixel primitive rasterization (not shown)


Primitive data (point, line, triangle, patch)
Tessellation Control Shader
Vertices
Per-patch data
Primitive Assembly
Primitive data bypass (tessellation control shader disabled)
Patch
Tessellation Evaluation Shader
Vertices
Primitive Assembly
Primitive data (vertex pos., colors, other assoc. data)
Tess levels
(u,v,w)
Tessellation Primitive Generator
Tess parameters
Connectivity
Primitive data bypass (tessellation disabled)

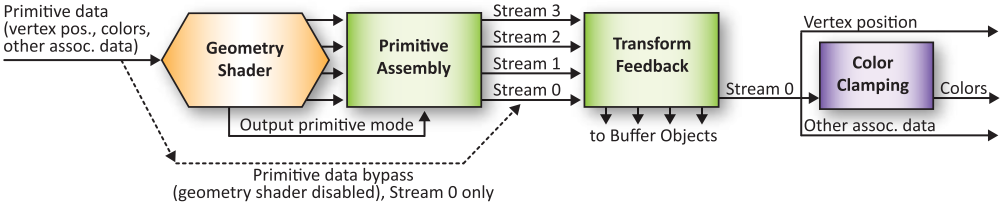
Geometry & Follow-on Details

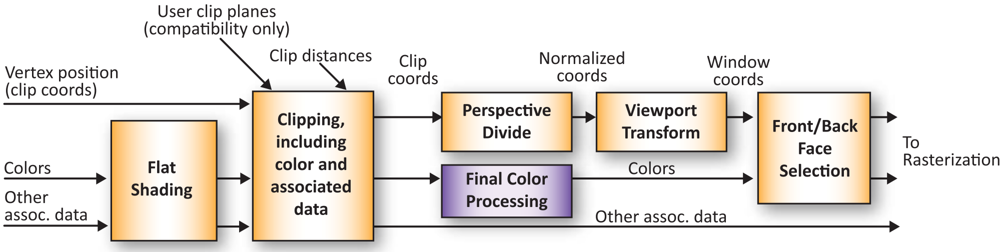
Geometry shaders (if enabled) consume individual primitives built in previous primitive assembly stages. For each input primitive, the geometry shader can output zero or more vertices, with each vertex directed at a specific vertex stream. The vertices emitted to each stream are assembled into primitives according to the geometry shader's output primitive type.

Transform feedback (if active) writes selected vertex attributes of the primitives of all vertex streams into buffer objects attached to one or more binding points.

Primitives on vertex stream zero are then processed by fixed-function stages, where they are clipped and prepared for rasterization.

- 🟧 Orange blocks indicate features of the Core specification.
- 🟪 Purple blocks indicate features of the Compatibility specification.
- 🟩 Green blocks indicate features new or significantly changed with OpenGL 4.x.





OpenGL Reference Card Index

The following index shows each item included on this card along with the page on which it is described. The color of the row in the table below is the color of the pane to which you should refer.

The Khronos Group is an industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics and dynamic media on a wide variety of platforms and devices.

See www.khronos.org to learn more about the Khronos Group.
See www.opengl.org to learn more about OpenGL.

Reference card production by Miller & Mattson www.millermattson.com