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 4.4 core specification.
- [n.n.n] refers to sections in the OpenGL Shading Language 4.40 specification.

OpenGL Errors [2.3.1]

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

OpenGL Operation

Floating-Point Numbers [2.3.3]

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 [Tables 2.1, 2.2]

Where a letter from the table below is used to denote type in a function name, T within the prototype is the same type.

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)

Synchronization

Flush and Finish [2.3.2]

void **Flush**(void);

void **Finish**(void);

Sync Objects and Fences [4.1]

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

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

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

Waiting for Sync Objects [4.1.1]

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

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

Sync Object Queries [4.1.3]

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

OpenGL Command Syntax [2.2]

GL commands are formed from a return type, a name, and optionally up to 4 characters (or character pairs) from the Command Letters table (to the left), 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. 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

Asynchronous Queries [4.2, 4.2.1]

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

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

void **BeginQuery**(enum *target*, uint *id*);
target: ANY_SAMPLES_PASSED[_CONSERVATIVE], PRIMITIVES_GENERATED, SAMPLES_PASSED, TIME_ELAPSED, TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN

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

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

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

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

void **GetQueryiv**(enum *target*, enum *pname*, int **params*);
target: see BeginQuery, plus TIMESTAMP
pname: CURRENT_QUERY, QUERY_COUNTER_BITS

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}, QUERY_RESULT_NO_WAIT

Timer Queries [4.3]

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*);

Buffer Objects [6]

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

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

Create and Bind Buffer Objects[6.1]

void **BindBuffer**(enum *target*, uint *buffer*);
target: [Table 6.1] {ARRAY, UNIFORM}_BUFFER, ATOMIC_COUNTER_BUFFER, COPY_{READ, WRITE}_BUFFER, {DISPATCH, DRAW}_INDIRECT_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_[UN]PACK_BUFFER, {QUERY, TEXTURE}_BUFFER, SHADER_STORAGE_BUFFER, TRANSFORM_FEEDBACK_BUFFER

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

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

void **BindBuffersRange**(enum *target*, uint *first*, sizei *count*, const uint **buffers*, const intptr **offsets*, const sizeiptr **size*);
target: see BindBufferRange

void **BindBuffersBase**(enum *target*, uint *first*, sizei *count*, const uint **buffers*);
target: see BindBufferRange

Create, Modify Buffer Object Data [6.2]

void **BufferStorage**(enum *target*, sizeiptr *size*, const void **data*, bitfield *flags*);
target: see BindBuffer
flags: Bitwise OR of MAP_{READ, WRITE}_BIT, {DYNAMIC, CLIENT}_STORAGE_BIT, MAP_{COHERENT, PERSISTENT}_BIT

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

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

void **ClearBufferSubData**(enum *target*, enum *internalFormat*, intptr *offset*, sizeiptr *size*, enum *format*, enum *type*, const void **data*);
target: see BindBuffer
internalformat: see TexBuffer on pg. 3 of this card
format: RED, GREEN, BLUE, RG, RGB, RGBA, BGR, BGRA,{RED, GREEN, BLUE, RG, RGB}_INTEGER, {RGBA, BGR, BGRA} _INTEGER, STENCIL_INDEX, DEPTH_{COMPONENT, STENCIL}

void **ClearBufferData**(enum *target*, enum *internalformat*, enum *format*, enum *type*, const void **data*);
target, internalformat, format: see ClearBufferSubData

Map/Unmap Buffer Data [6.3]

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

void ***MapBuffer**(enum *target*, enum *access*);
access: see MapBufferRange

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

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

Invalidate Buffer Data [6.5]

void **InvalidateBufferSubData**(uint *buffer*, intptr *offset*, sizeiptr *length*);

void **InvalidateBufferData**(uint *buffer*);

Copy Between Buffers [6.6]

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

Buffer Object Queries [6, 6.7]

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

void **GetBufferParameteriv**(enum *target*, enum *pname*, int **data*);
target: see BindBuffer
pname: [Table 6.2] BUFFER_SIZE, BUFFER_USAGE, BUFFER_{ACCESS[_FLAGS], BUFFER_MAPPED, BUFFER_MAP_{OFFSET, LENGTH}, BUFFER_IMMUTABLE_STORAGE, BUFFER_ACCESS_FLAGS

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*, const void ***params*);
target: see BindBuffer
pname: BUFFER_MAP_POINTER

Shaders and Programs

Shader Objects [7.1-2]

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

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

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

void **ReleaseShaderCompiler**(void);

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

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

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

Program Objects [7.3]

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 * const ** strings*);

void **ProgramParameteri**(uint *program*, enum *pname*, int *value*);
pname: PROGRAM_SEPARABLE, PROGRAM_BINARY_RETRIEVABLE_HINT
value: TRUE, FALSE

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

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

Program Interfaces [7.3.1]

void **GetProgramInterfaceiv**(uint *program*, enum *programInterface*, enum *pname*, int **params*);
programInterface: ATOMIC_COUNTER_BUFFER, BUFFER_VARIABLE, UNIFORM[_BLOCK], PROGRAM_{INPUT, OUTPUT}, SHADER_STORAGE_BLOCK, {GEOMETRY, VERTEX}_SUBROUTINE, TESS_{CONTROL, EVALUATION}_SUBROUTINE, {FRAGMENT, COMPUTE}_SUBROUTINE, TESS_CONTROL_SUBROUTINE_UNIFORM, TESS_EVALUATION_SUBROUTINE_UNIFORM, {GEOMETRY, VERTEX}_SUBROUTINE_UNIFORM, {FRAGMENT, COMPUTE}_SUBROUTINE_UNIFORM, TRANSFORM_FEEDBACK_{BUFFER, VARYING}
pname: ACTIVE_RESOURCES, MAX_NAME_LENGTH, MAX_NUM_ACTIVE_VARIABLES, MAX_NUM_COMPATIBLE_SUBROUTINES

uint **GetProgramResourceIndex**(uint *program*, enum *programInterface*, const char **name*);

void **GetProgramResourceName**(uint *program*, enum *programInterface*, uint *index*, sizei *bufSize*, sizei **length*, char **name*);

void **GetProgramResourceiv**(uint *program*, enum *programInterface*, uint *index*, sizei *propCount*, const enum **props*, sizei *bufSize*, sizei **length*, int **params*);
**props:* [see Table 7.2]

int **GetProgramResourceLocation**(uint *program*, enum *programInterface*, const char **name*);

int **GetProgramResourceLocationIndex**(uint *program*, enum *programInterface*, const char **name*);

(Continued on next page >)

Shaders and Programs (cont.)

Program Pipeline Objects [7.4]

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

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

boolean **IsProgramPipeline**(uint pipeline);

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, COMPUTE_SHADER_BIT

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

Program Binaries [7.5]

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

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

Uniform Variables [7.6]

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

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

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

void **GetActiveUniform**(uint program, uint index, sizei bufSize, sizei *length, int *size, enum *type, char *name);

*type returns: DOUBLE_{VECn, MATn, MATmxn}, DOUBLE, FLOAT_{VECn, MATn, MATmxn}, FLOAT, INT, INT_VECn, UNSIGNED_INT{_VECn}, BOOL, BOOL_VECn, or any value in [Table 7.3]

void **GetActiveUniformsiv**(uint program, sizei uniformCount, const uint *uniformIndices, enum pname, int *params);

pname: [Table 7.6] UNIFORM_{NAME_LENGTH, TYPE}, UNIFORM_{SIZE, BLOCK_INDEX, UNIFORM_OFFSET}, UNIFORM_{ARRAY, MATRIX}_STRIDE, UNIFORM_IS_ROW_MAJOR, UNIFORM_ATOMIC_COUNTER_BUFFER_INDEX

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_BLOCK_ACTIVE_UNIFORMS[_INDICES], UNIFORM_BLOCK_REFERENCED_BY_X_SHADER, where X may be one of VERTEX, FRAGMENT, COMPUTE, GEOMETRY, TESS_CONTROL, or TESS_EVALUATION [Table 7.7]

void **GetActiveAtomicCounterBufferiv**(uint program, uint bufferIndex, enum pname, int *params);

pname: see GetActiveUniformBlockiv, however replace the prefix UNIFORM_BLOCK_ with ATOMIC_COUNTER_BUFFER_

Load Uniform Vars. In Default Uniform Block

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

void **Uniform{1234}{i f d ui}v**(int location, sizei count, const T *value);

void **UniformMatrix{234}{f d}v**(int location, sizei count, boolean transpose, const float *value);

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

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

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

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

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

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

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

Uniform Buffer Object Bindings

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

Shader Buffer Variables [7.8]

void **ShaderStorageBlockBinding**(uint program, uint storageBlockIndex, uint storageBlockBinding);

Subroutine Uniform Variables [7.9]

Parameter shadertype for the functions in this section may be one of TESS_{CONTROL, EVALUATION}_SHADER, {COMPUTE, VERTEX}_SHADER, {FRAGMENT, GEOMETRY}_SHADER

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

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

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

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

void **GetActiveSubroutineUniformiv**(uint program, enum shadertype, uint index, enum pname, int *values);

pname: [NUM_]COMPATIBLE_SUBROUTINES

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

Shader Memory Access [7.12.2]

See diagram on page 6 for more information.

void **MemoryBarrier**(bitfield barriers);

barriers: ALL_BARRIER_BITS or the OR of X_BARRIER_BIT where X may be: VERTEX_ATTRIB_ARRAY, ELEMENT_ARRAY, UNIFORM, TEXTURE_FETCH, BUFFER_UPDATE, SHADER_IMAGE_ACCESS, COMMAND, PIXEL_BUFFER, TEXTURE_UPDATE, FRAMEBUFFER, TRANSFORM_FEEDBACK, ATOMIC_COUNTER, SHADER_STORAGE, CLIENT_MAPPED_BUFFER, QUERY_BUFFER

Shader|Program Queries [7.13]

void **GetShaderiv**(uint shader, enum pname, int *params);

pname: SHADER_TYPE, INFO_LOG_LENGTH, {DELETE, COMPILE}_STATUS, COMPUTE_SHADER, SHADER_SOURCE_LENGTH

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

pname: ACTIVE_ATOMIC_COUNTER_BUFFERS, ACTIVE_ATTRIBUTES, ACTIVE_ATTRIBUTE_MAX_LENGTH, ACTIVE_UNIFORMS, ACTIVE_UNIFORM_BLOCKS, ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, ACTIVE_UNIFORM_MAX_LENGTH, ATTACHED_SHADERS, COMPUTE_WORK_GROUP_SIZE, DELETE_STATUS, GEOMETRY_{INPUT, OUTPUT}_TYPE, GEOMETRY_SHADER_INVOCATIONS, GEOMETRY_VERTICES_OUT, INFO_LOG_LENGTH, LINK_STATUS, PROGRAM_SEPARABLE, PROGRAM_BINARY_RETRIEVABLE_HINT, TESS_CONTROL_OUTPUT_VERTICES, TESS_GEN_{MODE, SPACING}, TESS_GEN_{VERTEX_ORDER, POINT_MODE}, TRANSFORM_FEEDBACK_BUFFER_MODE, TRANSFORM_FEEDBACK_VARYINGS, TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, VALIDATE_STATUS

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

pname: ACTIVE_PROGRAM, VALIDATE_STATUS, {VERTEX, FRAGMENT, GEOMETRY}_SHADER, TESS_{CONTROL, EVALUATION}_SHADER, INFO_LOG_LENGTH, COMPUTE_SHADER

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

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

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

void **GetProgramPipelineInfoLog**(uint pipeline, 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, MEDIUM, HIGH}_{FLOAT, INT}

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

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

void **GetProgramStageiv**(uint program, enum shadertype, enum pname, int *values);

pname: ACTIVE_SUBROUTINES, ACTIVE_SUBROUTINES_X where X may be UNIFORMS, MAX_LENGTH, UNIFORM_LOCATIONS, UNIFORM_MAX_LENGTH

Textures and Samplers [8]

void **ActiveTexture**(enum texture);

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

Texture Objects [8.1]

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

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

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

void **BindTextures**(uint first, sizei count, const uint *textures);

target: see BindTexture

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

boolean **IsTexture**(uint texture);

Sampler Objects [8.2]

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

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

void **BindSamplers**(uint first, sizei count, const uint *samplers);

void **SamplerParameter{i f}**(uint sampler, enum pname, T param);

pname: TEXTURE_x where x may be WRAP_{S, T, R}, {MIN, MAG}_FILTER, {MIN, MAX}_LOD, BORDER_COLOR, LOD_BIAS, COMPARE_{MODE, FUNC} [Table 23.18]

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

pname: see SamplerParameter{if}

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

pname: see SamplerParameter{if}

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

boolean **IsSampler**(uint sampler);

Sampler Queries [8.3]

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

pname: see SamplerParameter{if}

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

pname: see SamplerParameter{if}

Pixel Storage Modes [8.4.1]

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

pname: [Tables 8.1, 18.1] [UN]PACK_X where X may be SWAP_BYTES, LSB_FIRST, ROW_LENGTH, SKIP_{IMAGES, PIXELS, ROWS}, ALIGNMENT, IMAGE_HEIGHT, COMPRESSED_BLOCK_WIDTH, COMPRESSED_BLOCK_{HEIGHT, DEPTH, SIZE}

Texture Image Spec. [8.5]

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

target: [PROXY_]TEXTURE_CUBE_MAP_ARRAY, [PROXY_]TEXTURE_3D, [PROXY_]TEXTURE_2D_ARRAY

internalformat: STENCIL_INDEX, RED, DEPTH_{COMPONENT, STENCIL}, RG, RGB, RGBA, COMPRESSED_{RED, RG, RGB, RGBA, SRGB, SRGB_ALPHA), a sized internal format from [Tables 8.12 - 8.13], or a specific compressed format in [Table 8.14]

format: DEPTH_{COMPONENT, STENCIL}, RED, GREEN, BLUE, RG, RGB, RGBA, BGR, BGRA, {BGRA, RED, GREEN, BLUE}_INTEGER, {RG, RGB, RGBA, BGR}_INTEGER, STENCIL_INDEX, [Table 8.3]

type: [UNSIGNED_]{BYTE, SHORT, INT}, [HALF_]FLOAT, or a value from [Table 8.2]

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

target: [PROXY_]TEXTURE_{2D, RECTANGLE}, [PROXY_]TEXTURE_1D_ARRAY, PROXY_TEXTURE_CUBE_MAP TEXTURE_CUBE_MAP_POSITIVE_{X, Y, Z}, TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}

internalformat, format, 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, format: see TexImage3D

Alternate Texture Image Spec. [8.6]

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

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

target: TEXTURE_1D

internalformat: see TexImage3D

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, 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, 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: see TexImage2D

(Continued on next page >)

Textures and Samplers (cont.)

void **CopyTexSubImage1D**(enum _target_, int _level_, int _xoffset_, int _x_, int _y_, sizei _width_);

target: _see TexSubImage1D_

Compressed Texture Images [8.7]

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_*X* where *X* may be [SIGNED_]RED_RGTC1, [SIGNED_]RG_RGTC2, {RGBA, SRGB_ALPHA}_BPTC_UNORM, RGB_BPTC_{SIGNED, UNSIGNED}_FLOAT

void **CompressedTexImage2D**(enum _target_, int _level_, enum _internalformat_, sizei _width_, sizei _height_, int _border_, sizei _imageSize_, const void *_data_);

target: _see TexImage2D_, omitting compressed rectangular texture formats

internalformat: _see CompressedTexImage3D_, plus COMPRESSED_*X* where *X* may be {RGB8, SRGB8}_ETC2, {RGB8, SRGB8}_PUNCHTHROUGH_ALPHA1_ETC2, {RGBA8, SRGB8_ALPHA8}_ETC2_EAC, [SIGNED_]R11_EAC, [SIGNED_]RG11_EAC

void **CompressedTexImage1D**(enum _target_, int _level_, enum _internalformat_, sizei _width_, int _border_, sizei _imageSize_, const void *_data_);

target: TEXTURE_1D, PROXY_TEXTURE_1D

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 internalformat for CompressedTexImage2D_

void **CompressedTexSubImage1D**(enum _target_, int _level_, int _xoffset_, sizei _width_, enum _format_, sizei _imageSize_, const void *_data_);

target: _see TexSubImage1D_

format: _see internalformat for CompressedTexImage1D_

Multisample Textures [8.8]

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, STENCIL_INDEX, DEPTH_{COMPONENT, STENCIL}, 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 [8.9]

void **TexBufferRange**(enum _target_, enum _internalFormat_, uint _buffer_, intptr _offset_, sizeiptr _size_);

void **TexBuffer**(enum _target_, enum _internalformat_, uint _buffer_);

target: TEXTURE_BUFFER

internalformat: [Table 8.15] R8, R8{I, UI}, R16, R16{F, I, UI}, R32{F, I, UI}, RG8, RG8{I, UI}, RG16, RG16{F, I, UI}, RG32{F, I, UI}, RGB32F, RGB32{I, UI}, RGBA8, RGBA8{I, UI}, RGBA16, RGBA16{F, I, UI}, RGBA32{F, I, UI}

Texture Parameters [8.10]

void **TexParameter{i f}**(enum _target_, enum _pname_, T _param_);

target: _see BindTexture_

void **TexParameter{i f}v**(enum _target_, enum _pname_, const T *_params_);

target: _see BindTexture_

void **TexParameterI{i ui}v**(enum _target_, enum _pname_, const T *_params_);

target: _see BindTexture_

pname: DEPTH_STENCIL_TEXTURE_MODE or TEXTURE_*X* where *X* may be one of WRAP_{S, T, R}, BORDER_COLOR, {MIN, MAG}_FILTER, LOD_BIAS,{MIN, MAX}_LOD, {BASE, MAX}_LEVEL, SWIZZLE_{R, G, B, A, RGBA}, COMPARE_{MODE, FUNC} [Table 8.16]

Enumerated Queries [8.11]

void **GetTexParameter{if}v**(enum _target_, enum _value_, T _data_);

target: _see BindTexture_

value: _see GetTexParameterI_

void **GetTexParameterI{i ui}v**(enum _target_, enum _value_, T _data_);

target: _see BindTexture_

value: _see pname for TexParameterI{i ui}v_, plus IMAGE_FORMAT_COMPATIBILITY_TYPE, TEXTURE_IMMUTABLE_{FORMAT, LEVELS}, TEXTURE_VIEW_NUM_{LEVELS, LAYERS}, TEXTURE_VIEW_MIN_{LEVEL, LAYER} [Table 8.16]

void **GetTexLevelParameter{i f}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,CUBE_MAP}_ARRAY, [PROXY_]TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}, TEXTURE_CUBE_MAP_POSITIVE_{X, Y, Z}, [PROXY_]TEXTURE_2D_MULTISAMPLE[_ARRAY]

value: TEXTURE_{WIDTH, HEIGHT, DEPTH}, TEXTURE_{SAMPLES, FIXED_SAMPLE_LOCATIONS}, TEXTURE_{INTERNAL_FORMAT, SHARED_SIZE}, TEXTURE_COMPRESSED[_IMAGE_SIZE], TEXTURE_BUFFER_DATA_STORE_BINDING, TEXTURE_BUFFER_{OFFSET, SIZE}, TEXTURE_STENCIL_SIZE, TEXTURE_ *X*_{SIZE, TYPE} where *X* can be RED, GREEN, BLUE, ALPHA, DEPTH

void **GetTexImage**(enum _tex_, int _lod_, enum _format_, enum _type_, void *_img_);

tex: TEXTURE_{1, 2}D[_ARRAY], TEXTURE_{3D, RECTANGLE, CUBE_MAP_ARRAY}, TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}, TEXTURE_CUBE_MAP_POSITIVE_{X, Y, Z}

format: _see ClearBufferSubData, pg 1 this card_

type: [UNSIGNED_]BYTE, SHORT, INT, [HALF_]FLOAT, or a value from [Table 8.2]

void **GetCompressedTexImage**(enum _target_, int _lod_, void *_img_);

target: _see tex for GetTexImage_

Cube Map Texture Select [8.13.1]

Enable/Disable/IsEnabled(TEXTURE_CUBE_MAP_SEAMLESS);

Manual Mipmap Generation [8.14.4]

void **GenerateMipmap**(enum _target_);

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

Texture Views [8.18]

void **TextureView**(uint _texture_, enum _target_, uint _origtexture_, enum _internalformat_, uint _minlevel_, uint _numlevels_, uint _minlayer_, uint _numlayers_);

target: TEXTURE_{1D, 2D,CUBE_MAP}[_ARRAY], TEXTURE_3D, TEXTURE_RECTANGLE, TEXTURE_2D_MULTISAMPLE[_ARRAY]

internalformat: [Table 8.21] R8{UI, I}, R8[_SNORM], RG8{F, UI, I}, RG8[_SNORM], RGB8[_SNORM], RGBA8{UI, I}, RGBA8[_SNORM], SRGB8[UI, I], SRGB8_ALPHA8, RGB9_E5, RGB10_A2[UI], R11F_G11F_B10F, RGBA16{F, UI, I}, RGBA16[_SNORM], RGB16{F, UI, I}, RGB16[_SNORM], RG16{F, UI, I}, RG16[_SNORM], R16{F, UI, I}, R16[_SNORM], RGBA32{F, UI, I}, RGB32{F, UI, I} , RG32{F, UI, I}, R32{F, UI, I}; COMPRESSED_*X* where *X* may be [SIGNED]_RED_RGTC1, [SIGNED]_RG_RGTC2, {RGBA, SRGB_ALPHA}_BPTC_UNORM, RGB_BPTC_[UN]SIGNED_FLOAT

Immutable-Format Tex. Images [8.19]

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 8.18-20]

void **TexStorage2D**(enum _target_, sizei _levels_, enum _internalformat_, sizei _width,_ sizei _height_);

target: [PROXY_]TEXTURE_{RECTANGLE, CUBE_MAP}, [PROXY_]TEXTURE_{1D_ARRAY, 2D}

internalformat: _see TexStorage1D_

void **TexStorage3D**(enum _target_, sizei _levels_, enum _internalformat_, sizei _width,_ sizei _height,_ sizei _depth_);

target: TEXTURE_3D, PROXY_TEXTURE_3D, [PROXY_]TEXTURE_{CUBE_MAP, 2D}[_ARRAY]

internalformat: _see TexStorage1D_

void **TexStorage2DMultisample**(enum _target_, sizei _samples_, enum _internalformat_, sizei _width_, sizei _height_, boolean _fixedsamplelocations_);

target: [PROXY_]TEXTURE_2D_MULTISAMPLE

void **TexStorage3DMultisample**(enum _target_, sizei _samples_, enum _internalformat_, sizei _width_, sizei _height_, sizei _depth_, boolean _fixedsamplelocations_);

target: [PROXY_]TEXTURE_2D_MULTISAMPLE_ARRAY

Invalidate Texture Image Data [8.20]

void **InvalidateTexSubImage**(uint _texture_, int _level_, int _xoffset_, int _yoffset_, int _zoffset_, sizei _width_, sizei _height_, sizei _depth_);

void **InvalidateTexImage**(uint _texture_, int _level_);

Clear Texture Image Data [8.21]

void **ClearTexSubImage**(uint _texture_, int _level_, int _xoffset_, int _yoffset_, int _zoffset_, sizei _width_, sizei _height_, sizei _depth_, enum _format_, enum _type_, const void *_data_);

format, type: _see TexImage3D, pg 2 this card_

void **ClearTexImage**(uint _texture_, int _level_, enum _format_, enum _type_, const void *_data_);

format, type: _see TexImage3D, pg 2 this card_

Texture Image Loads/Stores [8.26]

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, R11F_G11F_B10F, R{32,16}F, RGBA{32,16,8}UI, 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 8.25]

void **BindImageTextures**(uint _first_, sizei _count_, const uint *_textures_);

Framebuffer Objects

Binding and Managing [9.2]

void **BindFramebuffer**(enum _target_, uint _framebuffer_);

target: [DRAW_, READ_]FRAMEBUFFER

void **GenFramebuffers**(sizei _n_, uint *_framebuffers_);

void **DeleteFramebuffers**(sizei _n_, const uint *_framebuffers_);

boolean **IsFramebuffer**(uint _framebuffer_);

Framebuffer Object Parameters [9.2.1]

void **FramebufferParameteri**(enum _target_, enum _pname_, int _param_);

target: [DRAW_, READ_]FRAMEBUFFER

pname: FRAMEBUFFER_DEFAULT_*X* where *X* may be WIDTH, HEIGHT, FIXED_SAMPLE_LOCATIONS, SAMPLES, LAYERS

Framebuffer Object Queries [9.2.3]

void **GetFramebufferParameteriv**(enum _target_, enum _pname_, int *_params_);

target, pname: _see FramebufferParameteri_

void **GetFramebufferAttachmentParameteriv**(enum _target_, enum _attachment_, enum _pname_, int *_params_);

target: [DRAW_, READ_]FRAMEBUFFER

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

pname: FRAMEBUFFER_ATTACHMENT_ *X* where *X* may be OBJECT_{TYPE, NAME}, COMPONENT_TYPE, {RED, GREEN, BLUE}_SIZE, {ALPHA, DEPTH, STENCIL}_SIZE, COLOR_ENCODING, TEXTURE_{LAYER, LEVEL}, LAYERED, TEXTURE_CUBE_MAP_FACE

Attaching Images [9.2.4]

void **BindRenderbuffer**(enum _target_, uint _renderbuffer_);

target: RENDERBUFFER

void **GenRenderbuffers**(sizei _n_, uint *_renderbuffers_);

void **DeleteRenderbuffers**(sizei _n_, const uint *_renderbuffers_);

boolean **IsRenderbuffer**(uint _renderbuffer_);

void **RenderbufferStorageMultisample**(enum _target_, sizei _samples_, enum _internalformat_, sizei _width_, sizei _height_);

target: RENDERBUFFER

internalformat: _see TexImage3DMultisample_

void **RenderbufferStorage**(enum _target_, enum _internalformat_, sizei _width_, sizei _height_);

target: RENDERBUFFER

internalformat: _see TexImage3DMultisample_

Renderbuffer Object Queries [9.2.6]

void **GetRenderbufferParameteriv**(enum _target_, enum _pname_, int *_params_);

target: RENDERBUFFER

pname: [Table 23.27] RENDERBUFFER_*X* where *X* may be WIDTH, HEIGHT, INTERNAL_FORMAT, SAMPLES, {RED, GREEN, BLUE, ALPHA, DEPTH, STENCIL}_SIZE

Attaching Renderbuffer Images [9.2.7]

void **FramebufferRenderbuffer**(enum _target_, enum _attachment_, enum _renderbuffertarget_, uint _renderbuffer_);

target: [DRAW_, READ_]FRAMEBUFFER

attachment: [Table 9.2] {DEPTH, STENCIL, DEPTH_STENCIL}_ATTACHMENT, COLOR_ATTACHMENT*i* where *i* is [0, MAX_COLOR_ATTACHMENTS - 1]

renderbuffertarget: RENDERBUFFER

Attaching Texture Images [9.2.8]

void **FramebufferTexture**(enum _target_, enum _attachment_, uint _texture_, int _level_);

target: [DRAW_, READ_]FRAMEBUFFER

attachment: _see FramebufferRenderbuffer_

void **FramebufferTexture1D**(enum _target_, enum _attachment_, enum _textarget_, uint _texture_, int _level_);

textarget: TEXTURE_1D

target, attachment: _see FramebufferRenderbuffer_

void **FramebufferTexture2D**(enum _target_, enum _attachment_, enum _textarget_, uint _texture_, int _level_);

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

target, attachment: _see FramebufferRenderbuffer_

void **FramebufferTexture3D**(enum _target_, enum _attachment_, enum _textarget_, uint _texture_, int _level_, int _layer_);

textarget: TEXTURE_3D

target, attachment: _see FramebufferRenderbuffer_

void **FramebufferTextureLayer**(enum _target_, enum _attachment_, uint _texture_, int _level_, int _layer_);

target, attachment: _see FramebufferRenderbuffer_

Framebuffer Completeness [9.4.2]

enum **CheckFramebufferStatus**(enum _target_);

target: [DRAW_, READ_]FRAMEBUFFER

returns: FRAMEBUFFER_COMPLETE or a constant indicating the violating value

Vertices

Separate Patches [10.1.15]

void **PatchParameteri**(enum pname, int value);

pname: PATCH_VERTICES

Current Vertex Attribute Values [10.2]

Specify generic attributes with components of type float (**VertexAttrib***), int or uint (**VertexAttribI***), or double (**VertexAttribL***).

void **VertexAttrib{1234}{s f d}**(uint index, T values);

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

void **VertexAttrib4{b s i f d ub us ui}v**(uint index, const T *values);

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

void **VertexAttrib4N{b s i 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{b s 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: [UNSIGNED_]INT_2_10_10_10_REV, UNSIGNED_INT_10F_11F_11F_REV

Vertex Arrays

Generic Vertex Attribute Arrays [10.3.1]

void **VertexAttribFormat**(uint attribindex, int size, enum type, boolean normalized, unit relativeoffset);

type: [UNSIGNED_]BYTE, [UNSIGNED_]SHORT, [UNSIGNED_]INT, [HALF_]FLOAT, DOUBLE, FIXED, [UNSIGNED_]INT_2_10_10_10_REV, UNSIGNED_INT_10F_11F_11F_REV

void **VertexAttribIFormat**(uint attribindex, int size, enum type, unit relativeoffset);

type: [UNSIGNED_]BYTE, [UNSIGNED_]SHORT, [UNSIGNED_]INT

void **VertexAttribLFormat**(uint attribindex, int size, enum type, unit relativeoffset);

type: DOUBLE

void **BindVertexBuffer**(uint bindingindex, uint buffer, intptr offset, sizei stride);

void **BindVertexBuffers**(uint first, sizei count, const uint *buffers, const intptr *offsets, const sizei *strides);

void **VertexAttribBinding**(uint attribindex, uint bindingindex);

void **VertexAttribPointer**(uint index, int size, enum type, boolean normalized, sizei stride, const void *pointer);

type: **see VertexAttribFormat**

void **VertexAttribIPointer**(uint index, int size, enum type, sizei stride, const void *pointer);

type: **see VertexAttribIFormat**

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]

Vertex Attribute Divisors [10.3.2]

void **VertexBindingDivisor**(uint bindingindex, uint divisor);

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

Primitive Restart [10.3.5]

Enable/Disable/IsEnabled(target);

target: PRIMITIVE_RESTART{_FIXED_INDEX}

void **PrimitiveRestartIndex**(uint index);

Vertex Array Objects [10.4]

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);

boolean **IsVertexArray**(uint array);

Drawing Commands [10.5]

For all the functions in this section:

mode: POINTS, LINE_STRIP, LINE_LOOP, LINES, TRIANGLE_{STRIP, FAN}, TRIANGLES, PATCHES, LINES_ADJACENCY, TRIANGLES_ADJACENCY, {LINE, TRIANGLE}_STRIP_ADJACENCY,

type: UNSIGNED_{BYTE, SHORT, INT}

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

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

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

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

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

void **MultiDrawArraysIndirect**(enum mode, const void *indirect, sizei drawcount, sizei stride);

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

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

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

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

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 instancecount, int basevertex);

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

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

void **MultiDrawElementsIndirect**(enum mode, enum type, const void *indirect, sizei drawcount, sizei stride);

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

Vertex Array Queries [10.6]

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, LONG, NORMALIZED, SIZE, STRIDE, or TYPE

void **GetVertexAttribI{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, const void **pointer);

pname: VERTEX_ATTRIB_ARRAY_POINTER

Conditional Rendering [10.10]

void **BeginConditionalRender**(uint id, enum mode);

mode: {QUERY_BY_REGION, QUERY}_{WAIT, NO_WAIT}

void **EndConditionalRender**(void);

Vertex Attributes [11.1.1]

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

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

void **GetActiveAttrib**(uint program, uint index, sizei bufSize, sizei *length, int *size, enum *type, char *name);

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

Transform Feedback Variables [11.1.2]

void **TransformFeedbackVaryings**(uint program, sizei count, const char * const *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[_VECn], DOUBLE[_VECn], [UNSIGNED_]INT, [UNSIGNED_]INT_VECn, MATnxm, {FLOAT, DOUBLE}_{MATn, MATnxm}

Shader Execution [11.1.3]

void **ValidateProgram**(uint program);

void **ValidateProgramPipeline**(uint pipeline);

Tessellation Control Shaders [11.2.2]

void **PatchParameterfv**(enum pname, const float *values);

pname: PATCH_DEFAULT_{INNER, OUTER}_LEVEL

Vertex Post-Processing [13]

Transform Feedback [13.2]

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

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

boolean **IsTransformFeedback**(uint id);

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);

Transform Feedback Drawing [13.2.3]

void **DrawTransformFeedback**(enum mode, uint id);

mode: **see Drawing Commands [10.5] above**

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

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

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

Flatshading [13.4]

void **ProvokingVertex**(enum provokeMode);

provokeMode: {FIRST, LAST}_VERTEX_CONVENTION

Primitive Clipping [13.5]

Enable/Disable/IsEnabled(target);

target: DEPTH_CLAMP, CLIP_DISTANCEi where i = [0..MAX_CLIP_DISTANCES - 1]

Controlling Viewport [13.6.1]

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

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

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

void **DepthRangef**(float n, float 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);

Rasterization [13.4, 14]

Enable/Disable/IsEnabled(target);

target: RASTERIZER_DISCARD

Multisampling [14.3.1]

Use to antialias points, and lines.

Enable/Disable/IsEnabled(target);

target: MULTISAMPLE, SAMPLE_SHADING

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

pname: SAMPLE_POSITION

void **MinSampleShading**(float value);

Points [14.4]

void **PointSize**(float size);

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

pname, param: **see PointParameter{if}v**

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

pname: POINT_FADE_THRESHOLD_SIZE, POINT_SPRITE_COORD_ORIGIN

param, params: The fade threshold if pname is POINT_FADE_THRESHOLD_SIZE; {LOWER, UPPER}_LEFT if pname is POINT_SPRITE_COORD_ORIGIN.

Enable/Disable/IsEnabled(target);

target: PROGRAM_POINT_SIZE

Line Segments [14.5]

Enable/Disable/IsEnabled(target);

target: LINE_SMOOTH

void **LineWidth**(float width);

Polygons [14.6, 14.6.1]

Enable/Disable/IsEnabled(target);

target: POLYGON_SMOOTH, CULL_FACE

void **FrontFace**(enum dir);

dir: CCW, CW

(Continued on next page >)

Rasterization (cont.)

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

Polygon Rast. & Depth Offset [14.6.4-5]

void **PolygonMode**(enum *face*, enum *mode*);
face: FRONT_AND_BACK
mode: POINT, LINE, FILL

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

Enable/Disable/IsEnabled(*target*);
target: POLYGON_OFFSET_{POINT, LINE, FILL}

Per-Fragment Operations

Scissor Test [17.3.2]

Enable/Disable/IsEnabled(SCISSOR_TEST);

Enablei/Disablei/IsEnabledi(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 Ops. [17.3.3]

Enable/Disable/IsEnabled(*target*);
target: SAMPLE_ALPHA_TO_{COVERAGE, ONE}, SAMPLE_COVERAGE, SAMPLE_MASK

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

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

Stencil Test [17.3.5]

Enable/Disable/IsEnabled(STENCIL_TEST);

void **StencilFunc**(enum *func*, int *ref*, uint *mask*);
func: NEVER, ALWAYS, LESS, GREATER, EQUAL, LEQUAL, GEQUAL, NOTEQUAL

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

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, dppass: KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP, DECR_WRAP

Depth Buffer Test [17.3.6]

Enable/Disable/IsEnabled(DEPTH_TEST);

void **DepthFunc**(enum *func*);
func: see StencilFunc

Occlusion Queries [17.3.7]

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

EndQuery(enum *target*);
target: SAMPLES_PASSED, ANY_SAMPLES_PASSED, ANY_SAMPLES_PASSED_CONSERVATIVE

Blending [17.3.8]

Enable/Disable/IsEnabled(BLEND);

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

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

void **BlendEquationSeparate**(enum *modeRGB*, enum *modeAlpha*);
mode, modeRGB, modeAlpha: MIN, MAX , FUNC_{ADD, SUBTRACT, REVERSE_SUBTRACT}

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

void **BlendEquationSeparatei**(uint *buf*, enum *modeRGB*, enum *modeAlpha*);
mode, modeRGB, modeAlpha: see BlendEquationSeparate

void **BlendFunc**(enum *src*, enum *dst*);
src, dst: see BlendFuncSeparate

void **BlendFuncSeparate**(enum *srcRGB*, enum *dstRGB*, enum *srcAlpha*, enum *dstAlpha*);
src, dst, srcRGB, dstRGB, srcAlpha, dstAlpha: ZERO, ONE, SRC_ALPHA_SATURATE, {SRC, SRC1, DST, CONSTANT}_{COLOR, ALPHA}, ONE_MINUS_{SRC, SRC1}_{COLOR, ALPHA}, ONE_MINUS_{DST, CONSTANT}_{COLOR, ALPHA}

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

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

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

Dithering [17.3.10]

Enable/Disable/IsEnabled(DITHER);

Logical Operation [17.3.11]

Enable/Disable/IsEnabled(*COLOR_LOGIC_OP*);

void **LogicOp**(enum *op*);
op: CLEAR, AND, AND_REVERSE, COPY, AND_INVERTED, NOOP, XOR, OR, NOR, EQUIV, INVERT, OR_REVERSE, COPY_INVERTED, OR_INVERTED, NAND, SET

Fragment Shaders [15.2]

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

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

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

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

Reading and Copying Pixels

Reading Pixels [18.2]

void **ReadPixels**(int *x*, int *y*, sizei *width*, sizei *height*, enum *format*, enum *type*, void **data*);
format: STENCIL_INDEX, RED, GREEN, BLUE, RG, RGB, RGBA, BGR, DEPTH_{COMPONENT, STENCIL}, {RED, GREEN, BLUE, RG, RGB}_INTEGER, {RGBA, BGR, BGRA}_INTEGER, BGRA [Table 8.3]
type: [HALF_]FLOAT, [UNSIGNED_]BYTE, [UNSIGNED_]SHORT, [UNSIGNED_]INT, FLOAT_32_UNSIGNED_INT_24_8_REV, UNSIGNED_{BYTE, SHORT, INT}_* values in [Table 8.2]

void **ReadBuffer**(enum *src*);
src: NONE, {FRONT, BACK}_{LEFT, RIGHT}, FRONT, BACK, LEFT, RIGHT, FRONT_AND_BACK, COLOR_ATTACHMENT*i* (*i* = [0, MAX_COLOR_ATTACHMENTS - 1])

Final Conversion [18.2.6]

void **ClampColor**(enum *target*, enum *clamp*);
target: CLAMP_READ_COLOR
clamp: TRUE, FALSE, FIXED_ONLY

Copying Pixels [18.3]

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 or 0
filter: LINEAR, NEAREST

void **CopyImageSubData**(uint *srcName*, enum *srcTarget*, int *srcLevel*, int *srcX*, int *srcY*, int *srcZ*, uint *dstName*, enum *dstTarget*, int *dstLevel*, int *dstX*, int *dstY*, int *dstZ*, sizei *srcWidth*, sizei *srcHeight*, sizei *srcDepth*);
srcTarget, dstTarget: see target for BindTexture in section [8.1] on this card, plus GL_RENDERTARGET

Whole Framebuffer

Selecting a Buffer for Writing [17.4.1]

void **DrawBuffer**(enum *buf*);
buf: [Tables 17.4-5] NONE, {FRONT, BACK}_{LEFT, RIGHT}, FRONT, BACK, LEFT, RIGHT, FRONT_AND_BACK, COLOR_ATTACHMENT*i* (*i* = [0, MAX_COLOR_ATTACHMENTS - 1])

void **DrawBuffers**(sizei *n*, const enum **bufs*);
bufs: [Tables 17.5-6] {FRONT, BACK}_{LEFT, RIGHT}, NONE, COLOR_ATTACHMENT*i* (*i* = [0, MAX_COLOR_ATTACHMENTS - 1])

Fine Control of Buffer Updates [17.4.2]

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

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

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

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

void **StencilMaskSeparate**(enum *face*, uint *mask*);
face: FRONT, BACK, FRONT_AND_BACK

Clearing the Buffers [17.4.3]

void **Clear**(bitfield *buf*);
buf: 0 or the OR of {COLOR, DEPTH, STENCIL}_BUFFER_BIT

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

void **ClearDepth**(double *d*);

void **ClearDepthf**(float *d*);

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

void **ClearBuffer{i f 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

Invalidating Framebuffers [17.4.4]

void **InvalidateSubFramebuffer**(enum *target*, sizei *numAttachments*, const enum **attachments*, int *x*, int *y*, sizei *width*, sizei *height*);
target: [DRAW_ , READ_]FRAMEBUFFER
attachments: COLOR_ATTACHMENT*i*, DEPTH, {DEPTH, STENCIL}_ATTACHMENT, DEPTH_STENCIL_ATTACHMENT, COLOR, {FRONT, BACK}_{LEFT, RIGHT}, STENCIL

void **InvalidateFramebuffer**(enum *target*, sizei *numAttachments*, const enum **attachments*);
target, attachment: see InvalidateSubFramebuffer

Debug Output [20]

Enable/Disable/IsEnabled(DEBUG_OUTPUT);

Debug Message Callback [20.2]

void **DebugMessageCallback**(DEBUGPROC *callback*, void **userParam*);
callback: has the prototype:

void **callback**(enum *source*, enum *type*, uint *id*, enum *severity*, sizei *length*, const char **message*, void **userParam*);
source: DEBUG_SOURCE_*X* where *X* may be API, SHADER_COMPILER, WINDOW_SYSTEM, THIRD_PARTY, APPLICATION, OTHER
type: DEBUG_TYPE_*X* where *X* may be ERROR, MARKER, OTHER, DEPRECATED_BEHAVIOR, UNDEFINED_BEHAVIOR, PERFORMANCE, PORTABILITY, {PUSH, POP}_GROUP
severity: DEBUG_SEVERITY_{HIGH, MEDIUM}, DEBUG_SEVERITY_{LOW, NOTIFICATION}

Controlling Debug Messages [20.4]

void **DebugMessageControl**(enum *source*, enum *type*, enum *severity*, sizei *count*, const uint **ids*, boolean *enabled*);
source, type, severity: see callback (above), plus DONT_CARE

Externally Generated Messages [20.5]

void **DebugMessageInsert**(enum *source*, enum *type*, uint *id*, enum *severity*, int *length*, const char **buf*);
source: DEBUG_SOURCE_{APPLICATION, THIRD_PARTY}
type, severity: see DebugMessageCallback

Debug Groups [20.6]

void **PushDebugGroup**(enum *source*, uint *id*, sizei *length*, const char **message*);
source: see DebugMessageInsert

void **PopDebugGroup**(void);

Debug Labels [20.7]

void **ObjectLabel**(enum *identifier*, uint *name*, sizei length, const char **label*);
identifier: BUFFER, FRAMEBUFFER, RENDERBUFFER, PROGRAM_PIPELINE, PROGRAM, QUERY, SAMPLER, SHADER, TEXTURE, TRANSFORM_FEEDBACK, VERTEX_ARRAY

void **ObjectPtrLabel**(void* *ptr*, sizei *length*, const char **label*);

Synchronous Debug Output [20.8]

Enable/Disable/IsEnabled(DEBUG_OUTPUT_SYNCHRONOUS);

Debug Output Queries [20.9]

uint **GetDebugMessageLog**(uint *count*, sizei *bufSize*, enum **sources*, enum **types*, uint **ids*, enum **severities*, sizei **lengths*, char **messageLog*);

void **GetObjectLabel**(enum *identifier*, uint *name*, sizei *bufSize*, sizei **length*, char **label*);

void **GetObjectPtrLabel**(void* *ptr*, sizei *bufSize*, sizei **length*, char **label*);

Compute Shaders [19]

void **DispatchCompute**(uint *num_groups_x*, uint *num_groups_y*, uint *num_groups_z*);

void **DispatchComputeIndirect**(intptr *indirect*);

Hints [21.5]

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

State and State Requests

A complete list of symbolic constants for states is shown in the tables in [23].

Simple Queries [22.1]

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 **GetDoublei_v**(enum *target*, uint *index*, 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*);

(Continued on next page >)

States (cont.)

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

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

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

String Queries [22.2]

void **GetPointerv**(enum *pname*,
  void ***params*);

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

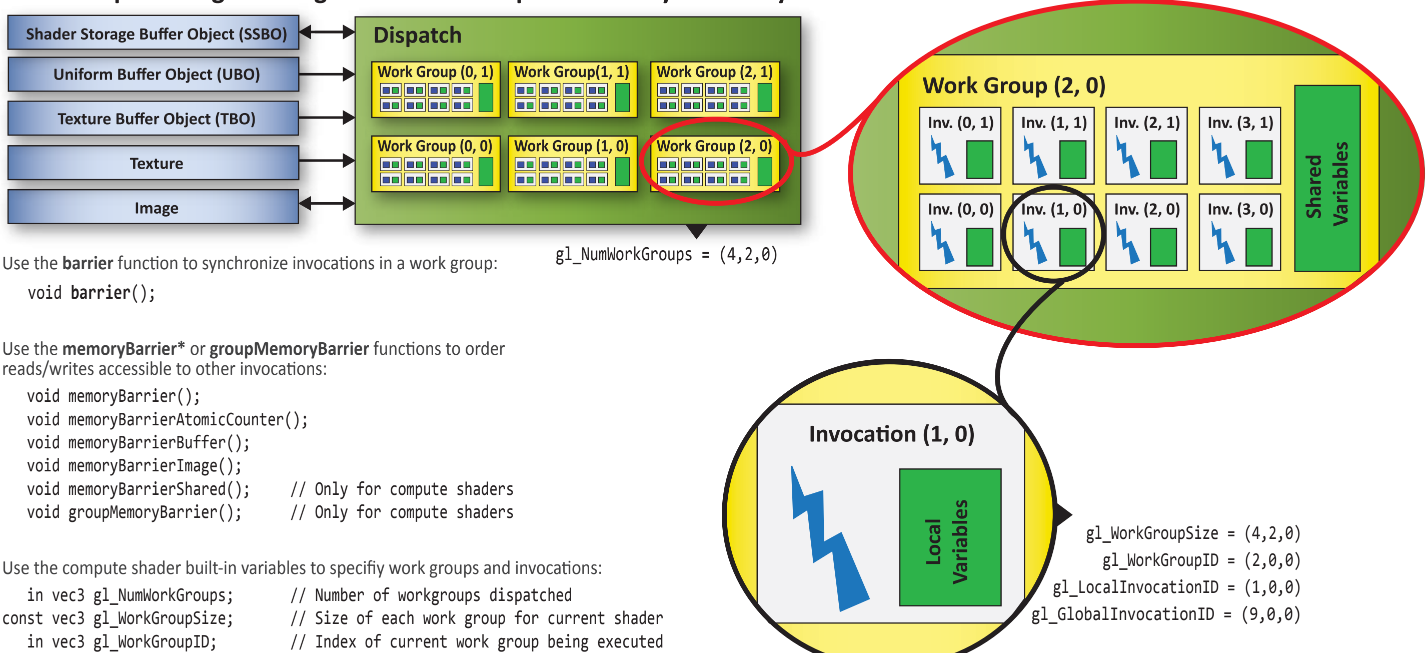
ubyte ***GetStringi**(enum *name*, uint *index*);
name: EXTENSIONS, SHADING_LANGUAGE_VERSION
index: EXTENSIONS range = [0, NUM_EXTENSIONS - 1]
  SHADING_LANGUAGE_VERSION range = [0, NUM_SHADING_LANGUAGE_VERSIONS-1]

Internal Format Queries [22.3]

void **GetInternalformati64v**(enum *target*,
  enum *internalformat*, enum *pname*,
  sizei *bufSize*, int64 **params*);
target: [Table 22.2]
  TEXTURE_{1D, 2D, 3D, CUBE_MAP}[_ARRAY],
  TEXTURE_2D_MULTISAMPLE[_ARRAY],
  TEXTURE_{BUFFER, RECTANGLE}, RENDERBUFFER
internalformat: any value
pname:
  CLEAR_{BUFFER, TEXTURE}, COLOR_ENCODING,
  COLOR_{COMPONENTS, RENDERABLE},
  COMPUTE_TEXTURE,
  DEPTH_{COMPONENTS, RENDERABLE},
  FILTER, FRAMEBUFFER_BLEND,
  FRAMEBUFFER_RENDERABLE[_LAYERED],
  {FRAGMENT, GEOMETRY}_TEXTURE,
  [MANUAL_GENERATE_]MIPMAP,
  IMAGE_COMPATIBILITY_CLASS,
  IMAGE_PIXEL_{FORMAT, TYPE},
  IMAGE_FORMAT_COMPATIBILITY_TYPE,
  IMAGE_TEXEL_SIZE,
  INTERNALFORMAT_{PREFERRED, SUPPORTED},
  INTERNALFORMAT_{RED, GREEN, BLUE}_SIZE,
  INTERNALFORMAT_{DEPTH, STENCIL}_SIZE,
  INTERNALFORMAT_{ALPHA, SHARED}_SIZE,
  INTERNALFORMAT_{RED, GREEN}_TYPE,
  INTERNALFORMAT_{BLUE, ALPHA}_TYPE,
  INTERNALFORMAT_{DEPTH, STENCIL}_TYPE,
  MAX_COMBINED_DIMENSIONS,
  MAX_{WIDTH, HEIGHT, DEPTH, LAYERS},
  NUM_SAMPLE_COUNTS,
  READ_PIXELS[_FORMAT, _TYPE],
  SAMPLES, SHADER_IMAGE_ATOMIC,
  SHADER_IMAGE_{LOAD, STORE},
  SIMULTANEOUS_TEXTURE_AND_DEPTH_TEST,
  SIMULTANEOUS_TEXTURE_AND_DEPTH_WRITE,
  SIMULTANEOUS_TEXTURE_AND_STENCIL_TEST,
  SIMULTANEOUS_TEXTURE_AND_STENCIL_WRITE,
  SRGB_{READ, WRITE},
  STENCIL_COMPONENTS,
  STENCIL_RENDERABLE,
  TESS_CONTROL_TEXTURE,
  TESS_EVALUATION_TEXTURE,
  TEXTURE_COMPRESSED,
  TEXTURE_COMPRESSED_BLOCK_HEIGHT,
  TEXTURE_COMPRESSED_BLOCK_WIDTH,
  TEXTURE_COMPRESSED_BLOCK_SIZE,
  TEXTURE_GATHER[_SHADOW],
  [GET_]TEXTURE_IMAGE_FORMAT,
  [GET_]TEXTURE_IMAGE_TYPE,
  TEXTURE_SHADOW,
  TEXTURE_VIEW,
  VERTEX_TEXTURE,
  VIEW_COMPATIBILITY_CLASS

void **GetInternalformativ**(enum *target*,
  enum *internalformat*, enum *pname*,
  sizei *bufSize*, int **params*);
target, pname, internalformat:
  see GetInternalformati64v,

OpenGL Compute Programming Model and Compute Memory Hierarchy



```
gl_NumWorkGroups = (4,2,0)
```

```
gl_WorkGroupSize = (4,2,0)
gl_WorkGroupID = (2,0,0)
gl_LocalInvocationID = (1,0,0)
gl_GlobalInvocationID = (9,0,0)
```

Use the **barrier** function to synchronize invocations in a work group:

```
void barrier();
```

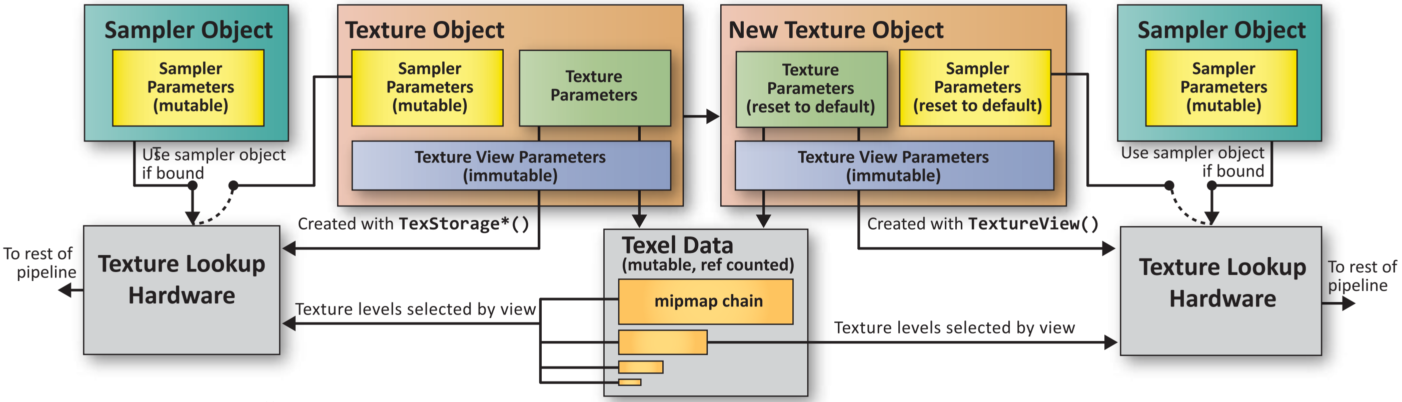
Use the **memoryBarrier*** or **groupMemoryBarrier** functions to order reads/writes accessible to other invocations:

```
void memoryBarrier();
void memoryBarrierAtomicCounter();
void memoryBarrierBuffer();
void memoryBarrierImage();
void memoryBarrierShared();     // Only for compute shaders
void groupMemoryBarrier();      // Only for compute shaders
```

Use the compute shader built-in variables to specifiy work groups and invocations:

```
   in vec3 gl_NumWorkGroups;      // Number of workgroups dispatched
const vec3 gl_WorkGroupSize;      // Size of each work group for current shader
   in vec3 gl_WorkGroupID;        // Index of current work group being executed
   in vec3 gl_LocalInvocationID;  // index of current invocation in a work group
   in vec3 gl_GlobalInvocationID; // Unique ID across all work groups and threads. (gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID)
```

OpenGL Texture Views and Texture Object State


Sampler Object
Sampler Parameters (mutable)
Use sampler object if bound
Texture Object
Sampler Parameters (mutable)
Texture Parameters
Texture View Parameters (immutable)
New Texture Object
Texture Parameters (reset to default)
Sampler Parameters (reset to default)
Texture View Parameters (immutable)
Sampler Object
Sampler Parameters (mutable)
Use sampler object if bound
Created with TexStorage*()
Created with TextureView()
To rest of pipeline
Texture Lookup Hardware
Texel Data (mutable, ref counted)
mipmap chain
Texture levels selected by view
Texture levels selected by view
Texture Lookup Hardware
To rest of pipeline

Texture state set with `TextureView()`

```
enum internalformat // base internal format       enum target     // texture target
uint minlevel       // first level of mipmap      uint numlevels  // number of mipmap levels
uint minlayer       // first layer of array texture uint numlayers // number of layers in array
```

Sampler Parameters (mutable)
TEXTURE_BORDER_COLOR
TEXTURE_COMPARE_{FUNC,MODE}
TEXTURE_LOD_BIAS
TEXTURE_{MAX,MIN}_LOD
TEXTURE_{MAG,MIN}_FILTER
TEXTURE_SRGB_DECODE
TEXTURE_WRAP_{S,T,R}

Texture Parameters (immutable)
TEXTURE_WIDTH
TEXTURE_DEPTH
TEXTURE_COMPRESSED
TEXTURE_IMMUTABLE_FORMAT
TEXTURE_HEIGHT
TEXTURE_FIXED_SAMPLE_LOCATIONS
TEXTURE_COMPRESSED_IMAGE_SIZE
TEXTURE_SAMPLES

Texture Parameters (mutable)
TEXTURE_SWIZZLE_{R,G,B,A}
TEXTURE_BASE_LEVEL
TEXTURE_MAX_LEVEL
DEPTH_STENCIL_TEXTURE_MODE

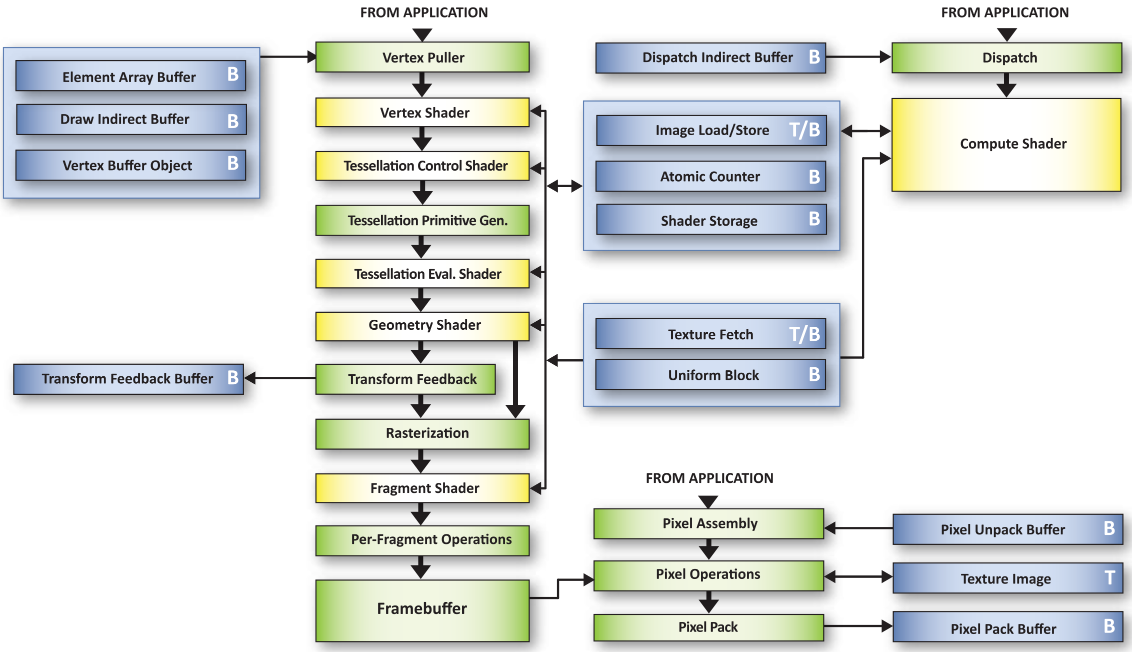
Texture View Parameters (immutable)
<target>
TEXTURE_INTERNAL_FORMAT
TEXTURE_VIEW_{MIN,NUM}_LEVEL
TEXTURE_IMMUTABLE_LEVELS
TEXTURE_{RED,GREEN,BLUE,ALPHA,DEPTH}_TYPE
TEXTURE_{RED,GREEN,BLUE,ALPHA,DEPTH,STENCIL}_SIZE
TEXTURE_SHARED_SIZE
TEXTURE_VIEW_{MIN,NUM}_LAYER
IMAGE_FORMAT_COMPATIBILITY_TYPE

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 and indicate data flow.

- 🟦 Blue blocks indicate various buffers that feed or get fed by the OpenGL pipeline.
- 🟩 Green blocks indicate fixed function stages.
- 🟨 Yellow blocks indicate programmable stages.
- T Texture binding
- B Buffer binding

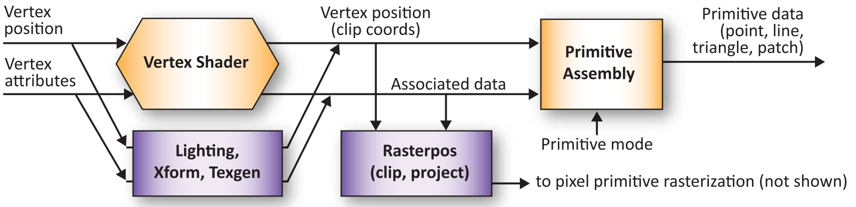


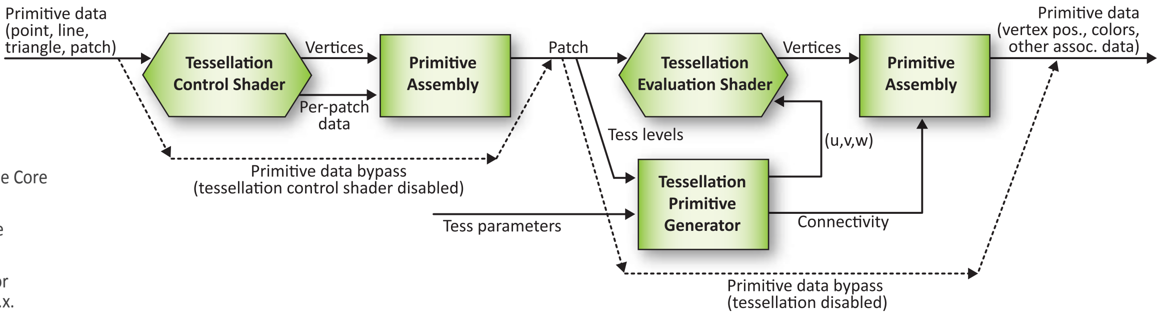
Vertex & Tessellation Details

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
Tess levels
Tessellation Primitive Generator
Tess parameters
(u,v,w)
Connectivity
Vertices
Primitive Assembly
Primitive data (vertex pos., colors, other assoc. data)
Primitive data bypass (tessellation disabled)

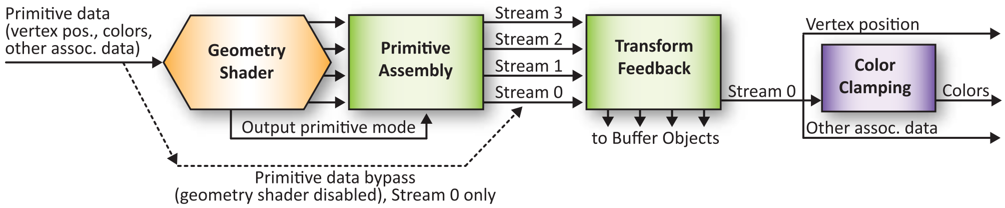
Geometry & Follow-on Details

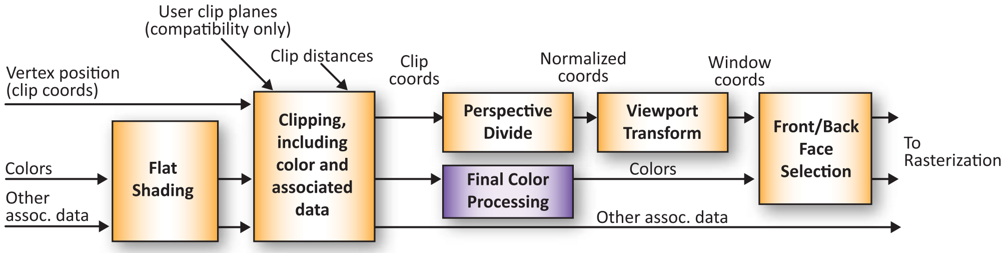
Geometry shaders (if enabled) consume individual primitives built in previous primitive assembly stages. For each input primitive, the geometry shader can output zero or more vertices, with each vertex directed at a specific vertex stream. The vertices emitted to each stream are assembled into primitives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





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, fragment, and compute shaders.

[n.n.n] and [Table n.n] refer to sections and tables in the OpenGL Shading Language 4.40 specification at www.opengl.org/registry

Preprocessor [3.3]

Preprocessor Directives

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

Preprocessor Operators

#version 440 #version 440 *profile*	Required when using version 4.40. *profile* is core, compatibility, or es.
#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 is being processed.
__VERSION__	Decimal integer, e.g.: 440
GL_core_profile	Defined as 1
GL_es_profile	1 if the implementation supports the es profile
GL_compatibility_profile	Defined as 1 if the implementation supports the compatibility profile.

Operators and Expressions [5.1]

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

1.	( )	parenthetical grouping
2.	[ ] ( ) . ++ --	array subscript function call, constructor, structure field, selector, swizzle 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-col. double-precision float matrix of 2, 3, 4 rows
dmat3x2, dmat3x3, dmat3x4	3-col. 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{1D,2D,3D}
image{1D,2D,3D}
	1D, 2D, or 3D texture

samplerCube **imageCube**	cube mapped texture
sampler2DRect **image2DRect**	rectangular texture
sampler{1D,2D}Array **image{1D,2D}Array**	1D or 2D array texture
samplerBuffer **imageBuffer**	buffer texture
sampler2DMS **image2DMS**	2D multi-sample texture
sampler2DMSArray **image2DMSArray**	2D multi-sample array texture
samplerCubeArray **imageCubeArray**	cube map array texture
sampler1DShadow **sampler2DShadow**	1D or 2D depth texture with comparison
sampler2DRectShadow	rectangular tex. / compare
sampler1DArrayShadow **sampler2DArrayShadow**	1D or 2D array depth texture with comparison
samplerCubeShadow	cube map depth texture with comparison
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
iimageCube	integer cube mapped image
isampler2DRect	int. 2D rectangular texture

Continue ⮥

Signed Integer Opaque Types (cont'd)

iimage2DRect	int. 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	int. 2D multi-sample texture
iimage2DMS	int. 2D multi-sample image
isampler2DMSArray	int. 2D multi-sample array tex.
iimage2DMSArray	int. 2D multi-sample array image
isamplerCubeArray	int. cube map array texture
iimageCubeArray	int. 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
uimage2DMS	uint 2D multi-sample image
usampler2DMSArray	uint 2D multi-sample array tex.

Continue ⮥

Unsigned Integer Opaque Types (cont'd)

uimage2DMSArray	uint 2D multi-sample array image
usamplerCubeArray	uint cube map array texture
uimageCubeArray	uint cube map array image

Implicit Conversions

int	->	uint	uvec2	->	dvec2
int, uint	->	float	uvec3	->	dvec3
int, uint, float	->	double	uvec4	->	dvec4
ivec2	->	uvec2	vec2	->	dvec2
ivec3	->	uvec3	vec3	->	dvec3
ivec4	->	uvec4	vec4	->	dvec4
ivec2	->	vec2	mat2	->	dmat2
ivec3	->	vec3	mat3	->	dmat3
ivec4	->	vec4	mat4	->	dmat4
uvec2	->	vec2	mat2x3	->	dmat2x3
uvec3	->	vec3	mat2x4	->	dmat2x4
uvec4	->	vec4	mat3x2	->	dmat3x2
ivec2	->	dvec2	mat3x4	->	dmat3x4
ivec3	->	dvec3	mat4x2	->	dmat4x2
ivec4	->	dvec4	mat4x3	->	dmat4x4

Aggregation of Basic Types

Arrays	float[3] foo; float foo[3]; int a [3][2];
	// Structures, blocks, and structure members
	// can be arrays. Arrays of arrays supported.

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	read-only 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
buffer	accessible by shaders and OpenGL API
shared	compute shader only, shared among work items in a local work group

Auxiliary Storage Qualifiers

Use to qualify some input and output variables:

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

Interface Blocks [4.3.9]

In, **out**, **uniform**, and **buffer** variable declarations can be grouped. For example:

```
uniform Transform {
  mat4 ModelViewMatrix;
  // allowed restatement qualifier
  uniform mat3 NormalMatrix;
};
```

Layout Qualifiers [4.4]

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

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

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

INPIT/OUTPUT layout qualifier for all shader stages except compute:

location = *integer-constant-expression*
component = *integer-constant-expression*

Tessellation

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

OUTPUT:
vertices = *integer-constant-expression*

Geometry Shader

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

OUTPUT:
points, line_strip, triangle_strip,
max_vertices = *integer-constant-expression*
stream = *integer-constant-expression*

Fragment Shader

INPUT: For redeclaring built-in variable gl_FragCoord: origin_upper_left, pixel_center_integer.

For **in** only (not with variable declarations): early_fragment_tests.

OUTPUT: gl_FragDepth may be redeclared using: depth_any, depth_greater, depth_less, depth_unchanged.
Additional qualifier for Fragment Shaders:
index = integer-constant-expression

Compute Shader

INPUT:
local_size_x = *integer-constant-expression*
local_size_y = *integer-constant-expression*
local_size_z = *integer-constant-expression*

Additional Output Layout Qualifiers [4.4.2]

Layout qualifiers for Transform Feedback:

The vertex, tessellation, and geometry stages allow the following on output declarations:

xfb_buffer = *integer-constant-expression*
xfb_offset = *integer-constant-expression*
xfb_stride = *integer-constant-expression*

Uniform Variable Layout Qualifiers [4.4.3]

location = *integer-constant-expression*

Subroutine Function Layout Qualifiers [4.4.4]

index = *integer-constant-expression*

Uniform/Storage Block Layout Qualifiers [4.4.5]

Layout qualifier identifiers for uniform blocks:
shared, packed, std140, std340,
{row, column}_major,
binding = *integer-constant-expression*
offset = *integer-constant-expression*
align = *integer-constant-expression*

Opaque Uniform Layout Qualifiers [4.4.6]

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

binding = *integer-constant-expression*

Atomic Counter Layout Qualifiers

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

(Continued on next page >)

Qualifiers (continued)

Format Layout Qualifiers

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

For tessellation control shaders:

binding = *integer-constant-expression*, 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]

Qualify individual variables:

{highp, mediump, lowp} *variable-declaration*;

Establish a default precision qualifier:

precision {highp, mediump, lowp} {int, float};

Invariant Qualifiers Examples [4.8]

These are for vertex, tessellation, geometry, and fragment languages.

#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. For example, a fused multiply-add cannot be used in the following; it 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]

```
const float c[3];
c.length()        // will return the integer 3
```

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 of 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

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, ..., type*n* arg*n*);

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, ...,
subroutineTypeName*N*)
returnType functionName(type0 arg0,
type1 arg1, ..., type*n* arg*n*){ ... }
// function body

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

subroutine uniform subroutineTypeName
subroutineVarName;

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

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	out 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[gl_MaxPatchVertices];
	
	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[gl_MaxPatchVertices];
	
	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_SampleMaskIn[];
	in int gl_Layer;
	in int gl_ViewportIndex;

Outputs	out float gl_FragDepth;
	out int gl_SampleMask[];

Compute Language

More information in diagram on page 6.

Inputs	**Work group dimensions**
	in uvec3 gl_NumWorkGroups;
	const uvec3 gl_WorkGroupSize;
	in uvec3 gl_LocalGroupSize;
	
	Work group and invocation IDs
	in uvec3 gl_WorkGroupID;
	in uvec3 gl_LocalInvocationID;
	
	Derived variables
	in uvec3 gl_GlobalInvocationID;
	in uint gl_LocalInvocationIndex;

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  ivec3 gl_MaxComputeWorkGroupCount =
            {65535, 65535, 65535} ;
const  ivec3 gl_MaxComputeWorkGroupSize[] =
            {1024, 1024, 64};
const  int gl_MaxComputeUniformComponents = 1024;
const  int gl_MaxComputeTextureImageUnits = 16;
const  int gl_MaxComputeImageUniforms = 8;
const  int gl_MaxComputeAtomicCounters = 8;
const  int gl_MaxComputeAtomicCounterBuffers = 1;
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_MaxCombinedImageUnitsAndFragment -
            Outputs = 8;
const  int gl_MaxImageSamples = 0;
const  int gl_MaxVertexImageUniforms= 0;
const  int gl_MaxTessControlImageUniforms = 0;
const  int gl_MaxTessEvaluationImageUniforms = 0;
const  int gl_MaxGeometryImageUniforms = 0;
const  int gl_MaxFragmentImageUniforms = 8;
const  int gl_MaxCombinedImageUniforms = 8;
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_MaxVertexAtomicCounterBuffers = 0;
const  int gl_MaxTessControlAtomicCounterBuffers = 0;
const  int gl_MaxTessEvaluationAtomicCounterBuffers = 0;
const  int gl_MaxGeometryAtomicCounterBuffers = 0;
const  int gl_MaxFragmentAtomicCounterBuffers = 1;
const  int gl_MaxCombinedAtomicCounterBuffers = 1;
const  int gl_MaxAtomicCounterBufferSize = 32;
const  int gl_MinProgramTexelOffset = -8;
const  int gl_MaxProgramTexelOffset =  7;
const int gl_MaxTransformFeedbackBuffers = 4;
const int gl_MaxTransformFeedbackInterleaved -
            Components = 64;
```

Built-In Functions

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*. Td= double, dvec*n*. Tfd= Tf, Td

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

Returns absolute value:
Tfd **abs**(Tfd *x*) Ti **abs**(Ti *x*)

Returns -1.0, 0.0, or 1.0:
Tfd **sign**(Tfd *x*) Ti **sign**(Ti *x*)

Returns nearest integer <= *x*:
Tfd **floor**(Tfd *x*)

Returns nearest integer with absolute value <= absolute value of *x*:
Tfd **trunc**(Tfd *x*)

Returns nearest integer, implementation-dependent rounding mode:
Tfd **round**(Tfd *x*)

Returns nearest integer, 0.5 rounds to nearest even integer:
Tfd **roundEven**(Tfd *x*)

Returns nearest integer >= *x*:
Tfd **ceil**(Tfd *x*)

Returns *x* - floor(*x*):
Tfd **fract**(Tfd *x*)

Returns modulus:
Tfd **mod**(Tfd *x*, Tfd *y*)
Tf **mod**(Tf *x*, float *y*)
Td **mod**(Td *x*, double *y*)

Returns separate integer and fractional parts:
Tfd **modf**(Tfd *x*, out Tfd *i*)

Returns minimum value:
Tfd **min**(Tfd *x*, Tfd *y*) Tiu **min**(Tiu *x*, Tiu *y*)
Tf **min**(Tf *x*, float *y*) Ti **min**(Ti *x*, int *y*)
Td **min**(Td *x*, double *y*) Tu **min**(Tu *x*, uint *y*)

(Continue ↰)

Common Functions (cont.)

Returns maximum value:
Tfd **max**(Tfd *x*, Tfd *y*) Tiu **max**(Tiu *x*, Tiu *y*)
Tf **max**(Tf *x*, float *y*) Ti **max**(Ti *x*, int *y*)
Td **max**(Td *x*, double *y*) Tu **max**(Tu *x*, uint *y*)

Returns min(max(*x*, *minVal*), *maxVal*):
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*)

Returns linear blend of *x* and *y*:
Tfd **mix**(Tfd *x*, Tfd *y*, Tfd *a*)
Tf **mix**(Tf *x*, Tf *y*, float *a*)
Td **mix**(Td *x*, Td *y*, double *a*)

Returns true if components in *a* select components from *y*, else from *x*:
Tfd **mix**(Tfd *x*, Tfd *y*, Tb *a*)

Returns 0.0 if *x* < *edge*, else 1.0:
Tfd **step**(Tfd *edge*, Tfd *x*)
Tf **step**(float *edge*, Tf *x*)
Td **step**(double *edge*, Td *x*)

Clamps and smoothes:
Tfd **smoothstep**(Tfd *edge0*, Tfd *edge1*, Tfd *x*)
Tf **smoothstep**(float *edge0*, float *edge1*, Tf *x*)
Td **smoothstep**(double *edge0*, double *edge1*, Td *x*)

Returns true if *x* is NaN:
Tb **isnan**(Tfd *x*)

Returns true if *x* is positive or negative infinity:
Tb **isinf**(Tfd *x*)

Returns signed int or uint value of the encoding of a float:
Ti **floatBitsToInt**(Tf *value*)
Tu **floatBitsToUint**(Tf *value*)

Returns float value of a signed int or uint encoding of a float:
Tf **intBitsToFloat**(Ti *value*) Tf **uintBitsToFloat**(Tu *value*)

Computes and returns a*b + c. Treated as a single operation when using **precise**:
Tfd **fma**(Tfd *a*, Tfd *b*, Tfd *c*)

Splits *x* into a floating-point significand in the range [0.5, 1.0) and an integer exponent of 2:
Tfd **frexp**(Tfd *x*, out Ti *exp*)

Builds a floating-point number from *x* and the corresponding integral exponent of 2 in *exp*:
Tfd **ldexp**(Tfd *x*, in Ti *exp*)

Floating-Point Pack/Unpack [8.4]

These do not operate component-wise.

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

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:
vec2 **unpackUnorm2x16**(uint *p*)
vec2 **unpackSnorm2x16**(uint *p*)
vec4 **unpackUnorm4x8**(uint *p*)
vec4 **unpackSnorm4x8**(uint *p*)

Packs components of *v* into a 64-bit value and returns a double-precision value:
double **packDouble2x32**(uvec2 *v*)

Returns a 2-component vector representation of *v*:
uvec2 **unpackDouble2x32**(double *v*)

Returns a uint by converting the components of a two-component floating-point vector:
uint **packHalf2x16**(vec2 *v*)

Returns a two-component floating-point vector:
vec2 **unpackHalf2x16**(uint *v*)

Type Abbreviations for Built-in Functions: In vector types, *n* is 2, 3, or 4.

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

Within any one function, type sizes and dimensionality must correspond after implicit type conversions. For example, float **round**(float) is supported, but float **round**(vec4) is not.

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]

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
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 comps. 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*.

Adds 32-bit uint *x* and *y*, returning the sum modulo 2^{32}:
Tu **uaddCarry**(Tu *x*, Tu *y*, out Tu *carry*)

Subtracts *y* from *x*, returning the difference if non-negative, otherwise 2^{32} plus the difference:
Tu **usubBorrow**(Tu *x*, Tu *y*, out Tu *borrow*)

(Continue ↰)

Integer Functions (cont.)

Multiplies 32-bit integers *x* and *y*, producing a 64-bit result:
void **umulExtended**(Tu *x*, Tu *y*, out Tu *msb*, out Tu *lsb*)
void **imulExtended**(Ti *x*, Ti *y*, out Ti *msb*, out Ti *lsb*)

Extracts bits [*offset*, *offset* + *bits* - 1] from *value*, returns them in the least significant bits of the result:
Tiu **bitfieldExtract**(Tiu *value*, int *offset*, int *bits*)

Returns the reversal of the bits of *value*:
Tiu **bitfieldReverse**(Tiu *value*)

Inserts the *bits* least-significant bits of *insert* into *base*:
Tiu **bitfieldInsert**(Tiu *base*, Tiu *insert*, int *offset*, int *bits*)

Returns the number of bits set to 1:
Ti **bitCount**(Tiu *value*)

Returns the bit number of the least significant bit:
Ti **findLSB**(Tiu *value*)

Returns the bit number of the most significant bit:
Ti **findMSB**(Tiu *value*)

Texture Lookup Functions [8.9]

Available to vertex, geometry, and fragment shaders. See tables on next page.

Atomic-Counter Functions [8.10]

Returns the value of an atomic counter.

Atomically increments *c* then returns its prior value:
uint **atomicCounterIncrement**(atomic_uint *c*)

Atomically decrements *c* then returns its prior value:
uint **atomicCounterDecrement**(atomic_uint *c*)

Atomically returns the counter for *c*:
uint **atomicCounter**(atomic_uint *c*)

Atomic Memory Functions [8.11]

Operates on individual integers in buffer-object or shared-variable storage. *OP* is Add, Min, Max, And, Or, Xor, Exchange, or CompSwap.

uint **atomic*OP***(inout uint *mem*, uint *data*)

int **atomic*OP***(inout int *mem*, int *data*)

Image Functions [8.12]

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

gimage1D *image*, int *P*
gimage2D *image*, ivec2 *P*
gimage3D *image*, ivec3 *P*
gimage2DRect *image*, ivec2 *P*
gimageCube *image*, ivec3 *P*
gimageBuffer *image*, int *P*
gimage1DArray *image*, ivec2 *P*
gimage2DArray *image*, ivec3 *P*
gimageCubeArray *image*, ivec3 *P*
gimage2DMS *image*, ivec2 *P*, int *sample*
gimage2DMSArray *image*, ivec3 *P*, int *sample*

Returns the dimensions of the images or images:
int **imageSize**(gimage{1D,Buffer} *image*)
ivec2 **imageSize**(gimage{2D,Cube,Rect,1DArray, 2DMS} *image*)
ivec3 **imageSize**(gimage{Cube,2D,2DMS}Array *image*)
vec3 **imageSize**(gimage3D *image*)

Loads texel at the coordinate *P* from the image unit *image*:
gvec4 **imageLoad**(readonly *IMAGE_PARAMS*)

Stores *data* into the texel at the coordinate *P* from the image specified by *image*:
void **imageStore**(writeonly *IMAGE_PARAMS*, gvec4 *data*)

(Continued on next page >)

Built-In Functions (cont.)

Image Functions (cont.)

Adds the value of *data* to the contents of the selected texel:

uint **imageAtomicAdd**(*IMAGE_PARAMS*, uint *data*)

int **imageAtomicAdd**(*IMAGE_PARAMS*, int *data*)

Takes the minimum of the value of *data* and the contents of the selected texel:

uint **imageAtomicMin**(*IMAGE_PARAMS*, uint *data*)

int **imageAtomicMin**(*IMAGE_PARAMS*, int *data*)

Takes the maximum of the value *data* and the contents of the selected texel:

uint **imageAtomicMax**(*IMAGE_PARAMS*, uint *data*)

int **imageAtomicMax**(*IMAGE_PARAMS*, int *data*)

Performs a bit-wise AND of the value of *data* and the contents of the selected texel:

uint **imageAtomicAnd**(*IMAGE_PARAMS*, uint *data*)

int **imageAtomicAnd**(*IMAGE_PARAMS*, int *data*)

Performs a bit-wise OR of the value of *data* and the contents of the selected texel:

uint **imageAtomicOr**(*IMAGE_PARAMS*, uint *data*)

int **imageAtomicOr**(*IMAGE_PARAMS*, int *data*)

(Continue ↑)

Image Functions (cont.)

Performs a bit-wise exclusive 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*)

Copies the value of *data*:

uint **imageAtomicExchange**(*IMAGE_PARAMS*, uint *data*)

int **imageAtomicExchange**(*IMAGE_PARAMS*, 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:

uint **imageAtomicCompSwap**(*IMAGE_PARAMS*, uint *compare*, uint *data*)

int **imageAtomicCompSwap**(*IMAGE_PARAMS*, int *compare*, int *data*)

Fragment Processing Functions [8.13]

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

Return value of *interpolant* sampled inside pixel and the primitive:

Tf **interpolateAtCentroid**(Tf *interpolant*)

Return value of *interpolant* at location of sample # *sample*:

Tf **interpolateAtSample**(Tf *interpolant*, int *sample*)

Return value of *interpolant* sampled at fixed offset *offset* from pixel center:

Tf **interpolateAtOffset**(Tf *interpolant*, vec2 *offset*)

Noise Functions [8.14]

Returns noise value. Available to fragment, geometry, and vertex shaders. *n* is 2, 3, or 4:

float **noise1**(Tf *x*)   vec*n* **noise*n***(Tf *x*)

Geometry Shader Functions [8.15]

Only available in geometry shaders.

Emits values of output variables to current output primitive stream *stream*:

void **EmitStreamVertex**(int *stream*)

Completes current output primitive stream *stream* and starts a new one:

void **EndStreamPrimitive**(int *stream*)

(Continue ↑)

Geometry Shader Functions (cont'd)

Emits values of output variables to the current output primitive:

void **EmitVertex**()

Completes output primitive and starts a new one:

void **EndPrimitive**()

Other Shader Functions [8.16-17]

See diagram on page 11 for more information.

Synchronizes across shader invocations:

void **barrier**()

Controls ordering of memory transactions issued by a single shader invocation:

void **memoryBarrier**()

Controls ordering of memory transactions as viewed by other invocations in a compute work group:

void **groupMemoryBarrier**()

Order reads and writes accessible to other invocations:

void **memoryBarrierAtomicCounter**()
void **memoryBarrierShared**()
void **memoryBarrierBuffer**()
void **memoryBarrierImage**()

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, 2DRect,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*)

textureQueryLevels functions return the number of mipmap levels accessible in the texture associated with *sampler*.

int **textureQueryLevels**(gsampler{1D[Array],2D[Array],3D,Cube[Array]} *sampler*)

int **textureQueryLevels**(sampler{1D[Array],2D[Array],Cube[Array]}Shadow *sampler*)

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*])

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

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

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.

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,Array]}Shadow *sampler*, {vec3, vec4} *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*, {int,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], Cube}Shadow *sampler*, {vec3,vec4} *P*, {float,vec2} *dPdx*, {float,vec2, vec3} *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,vec4} *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 *offsets*[4] [, int *comp*])

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

OpenGL API and OpenGL Shading Language 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.