OpenGL ES (Open Graphics Library for Embedded Systems) is a software interface to graphics hardware. The interface consists of a set of procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

Specifications are available at www.khronos.org/registry/gles/

- **[n.n.n]** refers to sections and tables in the OpenGL ES 3.1 specification.
- **[n.n.n]** refers to sections in the OpenGL ES Shading Language 3.10 specification.

OpenGL ES Command Syntax [2.2]

Commands are formed from a return type, a name, and optionally letters to denote type: i for 32-bit int, i64 for int64, f for 32-bit float, or ui for 32-bit uint, shown in the prototype below:

return-type **Name**{1234}{i i64 f ui}{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

Command Execution

OpenGL Errors [2.3.1]

enum **GetError**(void); //Returns one of the values shown in the table to the right.

NO_ERROR	No error encountered
INVALID_ENUM	Enum argument out of range
INVALID_VALUE	Numeric arg. out of range
INVALID_OPERATION	Operation illegal
INVALID_FRAME-BUFFER_OPERATION	Framebuffer is incomplete
OUT_OF_MEMORY	Not enough memory left to execute command

Flush and Finish [2.3.2]

void **Flush**(void); void **Finish**(void);

Synchronization

Sync Objects and Fences [4.1]

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

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

Waiting for Sync Objects [4.1.1]

enum **ClientWaitSync**(sync *sync*, bitfield *flags*, uint64 *timeout*);
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}

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

Programs and Shaders

Shader Objects [7.1-2]

uint **CreateShader**(enum *type*);
type: FRAGMENT_SHADER, VERTEX_SHADER, COMPUTE_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*);

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*);

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

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, TRANSFORM_FEEDBACK_VARYING
pname: ACTIVE_RESOURCES, MAX_NAME_LENGTH, MAX_NUM_ACTIVE_VARIABLES

uint **GetProgramResourceIndex**(uint *program*, enum *programInterface*, const char *name);
programInterface: *See GetProgramInterfaceiv*, omitting ATOMIC_COUNTER_BUFFER

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

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

int **GetProgramResourceLocation**(uint *program*, enum *programInterface*, const char *name);
programInterface: UNIFORM, PROGRAM_{INPUT, OUTPUT}

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 {VERTEX, FRAGMENT, 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 **GetUniformIndices**(uint *program*, sizei *uniformCount*, const char * const *uniformNames, uint *uniformIndices);

void **GetActiveUniform**(uint *program*, uint *index*, sizei *bufSize*, sizei *length*, int *size, enum *type, char *name);
*type returns: [Table 7.3] FLOAT[_VEC{2, 3, 4}], INT[_VEC{2, 3, 4}], UNSIGNED_INT[_VEC{2, 3, 4}], BOOL[_VEC{2, 3, 4}], FLOAT_MAT{2, 3, 4}, FLOAT_MAT2x{3, 4}, FLOAT_MAT3x{2, 4}, FLOAT_MAT4x{2, 3}, SAMPLER_{2D, 3D, CUBE}, [UNSIGNED_]INT_SAMPLER_{2D, 3D, CUBE}, SAMPLER_{CUBE, 2D[_ARRAY]}_SHADOW, SAMPLER_2D_{ARRAY, MULTISAMPLE}, [UNSIGNED_]INT_SAMPLER_2D_{ARRAY, MULTISAMPLE}, IMAGE_{2D[_ARRAY], 3D, CUBE}, [UNSIGNED_]INT_IMAGE_{2D[_ARRAY], 3D, CUBE}, UNSIGNED_INT_ATOMIC_COUNTER

void **GetActiveUniformsiv**(uint *program*, sizei *uniformCount*, const uint *uniformIndices, enum *pname*, int *params);
pname: [Table 7.6] UNIFORM_{NAME_LENGTH, TYPE, SIZE}, UNIFORM_{BLOCK_INDEX, OFFSET}, UNIFORM_{ARRAY, MATRIX}_STRIDE, UNIFORM_IS_ROW_MAJOR

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, UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, UNIFORM_BLOCK_REFERENCED_BY_X_SHADER, where X may be one of VERTEX, FRAGMENT [Table 7.7]

Asynchronous Queries [4.2]

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

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

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

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

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

void **GetQueryiv**(enum *target*, enum *pname*, int *params);
target: *See BeginQuery*
pname: CURRENT_QUERY

void **GetQueryObjectuiv**(uint *id*, enum *pname*, uint *params);
pname: QUERY_RESULT[_AVAILABLE]

Buffer Objects [6]

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

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

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

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, 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*

Buffer Object Data Stores [6.2]

void **BufferData**(enum *target*, sizeiptr *size*, const void *data, enum *usage*);
target: *See BindBuffer*
usage: DYNAMIC_{DRAW, READ, COPY}, {STREAM, STATIC}_{DRAW, READ, COPY}

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

Map/Unmap Buffer Data [6.3]

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

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

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

Copy Between Buffers [6.5]

void **CopyBufferSubData**(enum *readTarget*, enum *writeTarget*, intptr *readOffset*, intptr *writeOffset*, sizeiptr *size*);
readtarget and *writetarget:* *See BindBuffer*

Buffer Object Queries [6.6]

void **GetBufferParameteri[64]v**(enum *target*, enum *pname*, int[64]*data);
target: *See BindBuffer*
pname: [Table 6.2] BUFFER_SIZE, BUFFER_USAGE, BUFFER_MAP_{OFFSET, LENGTH}, BUFFER_MAPPED, BUFFER_ACCESS_FLAGS

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

(Continued on next page) ▶

◀ Programs and Shaders (cont.)

Load Uniform Vars. In Default Uniform Block

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

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

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

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

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

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

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

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

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

void **ProgramUniformMatrixf{2x3,3x2, 2x4,4x2, 3x4, 4x3}{f}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 Memory Access [7.11]

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,

void **MemoryBarrierByRegion**(bitfield *barriers*);

barriers:
ALL_BARRIER_BITS or the OR of *X*_BARRIER_BIT where *X* may be: ATOMIC_COUNTER, FRAMEBUFFER, SHADER_IMAGE_ACCESS, SHADER_STORAGE, TEXTURE_FETCH, UNIFORM

Shader, Program, Pipeline Queries [7.12]

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

pname: SHADER_{SOURCE_LENGTH, TYPE}, INFO_LOG_LENGTH, {DELETE, COMPILE}_STATUS

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

pname:
ACTIVE_ATOMIC_COUNTER_BUFFERS, ACTIVE_ATTRIBUTES, ACTIVE_UNIFORMS, ACTIVE_ATTRIBUTE_MAX_LENGTH, ACTIVE_UNIFORM_MAX_LENGTH, ACTIVE_UNIFORM_BLOCKS, ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, ATTACHED_SHADERS, COMPUTE_WORK_GROUP_SIZE, {DELETE, LINK}_STATUS, INFO_LOG_LENGTH, PROGRAM_SEPARABLE, PROGRAM_BINARY_RETRIEVABLE_HINT, 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, {COMPUTE, FRAGMENT, VERTEX}_SHADER, INFO_LOG_LENGTH

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 i ui}v**(uint *program*, int *location*, T **params*);

Textures and Samplers [8]

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

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

Texture Objects [8.1]

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

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

target:
TEXTURE_2D_ARRAY, TEXTURE_3D, TEXTURE_CUBE_MAP, TEXTURE_2D_MULTISAMPLE

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 **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, COMPARE_{MODE, FUNC} [Table 20.11]

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

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}***

Pixel Storage Modes [8.4.1]

void **PixelStorei**(enum *pname*, T *param*);

pname: [Tables 8.1, 18.1] [UN]PACK_ALIGNMENT, [UN]PACK_ROW_LENGTH, [UN]PACK_SKIP_PIXELS, [UN]PACK_SKIP_ROWS, UNPACK_IMAGE_HEIGHT, UNPACK_SKIP_IMAGES

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: TEXTURE_3D, TEXTURE_2D_ARRAY

format:
ALPHA, RGBA, RGB, RG, RED, {RGBA, RGB, RG, RED}_INTEGER, DEPTH_{COMPONENT, STENCIL}, LUMINANCE_ALPHA, LUMINANCE

type:
{UNSIGNED_}BYTE, {UNSIGNED_}SHORT, {UNSIGNED_}INT, {HALF_}FLOAT, UNSIGNED_SHORT_4_4_4_4, UNSIGNED_SHORT_5_5_5_1, UNSIGNED_SHORT_5_6_5, UNSIGNED_INT_2_10_10_10_REV, UNSIGNED_INT_24_8, UNSIGNED_INT_10F_11F_11F_REV, UNSIGNED_INT_5_9_9_9_REV, FLOAT_32_UNSIGNED_INT_24_8_REV

internalformat:
R8, R8I, R8UI, R8_SNORM, R16I, R16UI, R16F, R32I, R32UI, R32F, RG8, RG8I, RG8UI, RG8_SNORM, RG16I, RG16UI, RG16F, RG32I, RG32UI, RG32F, RGB, RGB5_A1, RGB565, RGB8, RGB8I, RGB8UI, RGB8_SNORM, SRGB8, SRGB8_ALPHA8, RGB9_E5, RGB10_A2, RGB10_A2UI, RGB16I, RGB16UI, RGB16F, RGB32I, RGB32UI, RGB32F, RGBA, RGBA4, RGBA8, RGBA8I, RGBA8UI, RGBA8_SNORM, RGBA16I, RGBA16UI, RGBA16F, RGBA32I, RGBA32UI, RGBA32F, R11F_G11F_B10F, LUMINANCE_ALPHA, ALPHA, LUMINANCE, DEPTH_COMPONENT{16, 24, 32F}

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

target: TEXTURE_2D, TEXTURE_CUBE_MAP_POSITIVE_{X, Y, Z}, TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}

internalformat: ***See TexImage3D***

format, type: ***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: ***See TexImage2D***

internalformat: ***See TexImage3D,*** except for DEPTH* values

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

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 TexImage2D***

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: TEXTURE_3D, TEXTURE_2D_ARRAY

format, type: ***See TexImage3D***

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

target: ***See TexImage2D***

Compressed Texture Images [8.9]

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

target: ***See TexImage2D***

internalformat: [Table 8.19] COMPRESSED_*X* where *X* may be one of [SIGNED_]R11_EAC, [SIGNED_]RG11_EAC, [S]RGB8_ETC2, [S]RGB8_PUNCHTHROUGH_ALPHA1_ETC2, RGBA8_ETC2_EAC, SRGB8_ALPHA8_ETC2_EAC

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: ***See TexImage3D***

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

target: ***See TexImage2D***

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: TEXTURE_2D_ARRAY, TEXTURE_3D

Multisample Textures [8.8]

void **TexStorage2DMultisample**(enum *target*, sizei *samples*, int *sizedinternalformat*, sizei *width*, sizei *height*, boolean *fixedsamplelocations*);

target: TEXTURE_2D_MULTISAMPLE

sizedinternalformat:
R8, R8I, R8UI, R16I, R16UI, R32I, R32UI, RG8, RG8I, RG8UI, RG16I, RG16UI, RG32I, RG32UI, RGB5_A1, RGB565, RGB8, RGB10_A2, RGB10_A2UI, RGBA4, RGBA8, RGBA8I, RGBA8UI, RGBA16I, RGBA16UI, RGBA32I, RGBA32UI, SRGB8_ALPHA8, STENCIL_INDEX8, DEPTH24_STENCIL8, DEPTH32F_STENCIL8, DEPTH_COMPONENT{16, 24, 32F}, LUMINANCE, ALPHA

Texture Parameters [8.9]

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

target: TEXTURE_{2D, 3D}, TEXTURE_CUBE_MAP, TEXTURE_2D_{ARRAY, MULTISAMPLE}

pname: [Tables 8.20] DEPTH_STENCIL_TEXTURE_MODE TEXTURE_COMPARE_{MODE, FUNC}, TEXTURE_{BASE, MAX}_LEVEL, TEXTURE_{MIN, MAX}_LOD, TEXTURE_{MIN, MAG}_FILTER, TEXTURE_SWIZZLE_{R,G,B,A}, TEXTURE_WRAP_{S,T,R}

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

target, pname: ***See TexParameter{if},***

Texture Queries [8.10]

void **GetTexParameter{i f}v**(enum *target*, enum *pname*, T * *params*);

target:
TEXTURE_{2D, 3D}, TEXTURE_CUBE_MAP, TEXTURE_2D_{ARRAY, MULTISAMPLE}

pname:
See TexParameter{if}v, plus IMAGE_FORMAT_COMPATIBILITY_TYPE, TEXTURE_IMMUTABLE_{FORMAT, LEVELS}

void **GetTexLevelParameter{i f}v**(enum *target*, int *lod*, enum *pname*, T **params*);

target:
TEXTURE_2D[_MULTISAMPLE], TEXTURE_3D, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_POSITIVE_{X, Y, Z}, TEXTURE_CUBE_MAP_NEGATIVE_{X, Y, Z}

pname:
TEXTURE _*Y*, where *Y* may be WIDTH, HEIGHT, DEPTH, SAMPLES, INTERNAL_FORMAT, FIXED_SAMPLE_LOCATIONS, SHARED_SIZE, COMPRESSED, STENCIL_SIZE; or TEXTURE_*X*_{SIZE, TYPE} where *X* can be RED, GREEN, BLUE, ALPHA, DEPTH

Manual Mipmap Generation [8.13.4]

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

target: TEXTURE_{2D, 3D, 2D_ARRAY, CUBE_MAP}

Immutable-Format Tex. Images [8.17]

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

target: TEXTURE_2D, TEXTURE_CUBE_MAP

internalformat: ***See TexImage3D***

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

target: TEXTURE_3D, TEXTURE_2D_ARRAY

internalformat: ***See TexImage3D***

Texture Image Loads/Stores [8.22]

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

access: READ_ONLY, WRITE_ONLY, READ_WRITE

format: [Table 8.27] R32{I, F, UI}, RGBA32{I, F, UI}, RGBA16{I, F, UI}, RGBA8 , RGBA8{I, UI}, RGBA8_SNORM

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

Framebuffer Object Queries [9.2.3]

void **GetFramebufferParameteriv**(enum *target*, enum *pname*, int **params*);
target: [DRAW_, READ_]FRAMEBUFFER
pname: ***See FramebufferParameteri***

void **GetFramebufferAttachmentParameteriv**(enum *target*, enum *attachment*, enum *pname*, int **params*);
target: [DRAW_, READ_]FRAMEBUFFER
attachment:
BACK, DEPTH, STENCIL, COLOR_ATTACHMENT*i*, {DEPTH, STENCIL, DEPTH_STENCIL}_ATTACHMENT
pname:
FRAMEBUFFER_ATTACHMENT_*X* where *X* may be OBJECT_{NAME, TYPE}, COLOR_ENCODING, COMPONENT_TYPE, {RED, GREEN, BLUE}_SIZE, {ALPHA, DEPTH, STENCIL}_SIZE, TEXTURE_{LAYER, LEVEL}, TEXTURE_CUBE_MAP_FACE

Renderbuffer Objects [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 sizedinternalformat for TexStorage2DMultisample***

void **RenderbufferStorage**(enum *target*, enum *internalformat*, sizei *width*, sizei *height*);
target: RENDERBUFFER
internalformat: ***See TexStorage2DMultisample***

Renderbuffer Object Queries [9.2.6]

void **GetRenderbufferParameteriv**(enum *target*, enum *pname*, int **params*);
target: RENDERBUFFER
pname: [Table 20.16]
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 9.1]
{DEPTH, STENCIL, DEPTH_STENCIL}_ATTACHMENT, COLOR_ATTACHMENT*i* where *i* is [0, MAX_COLOR_ATTACHMENTS - 1]
renderbuffertarget: RENDERBUFFER if *renderbuffer* is non-zero, else undefined

Attaching Texture Images [9.2.8]

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, 2D_MULTISAMPLE} if *texture* is zero, else undefined
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

Vertex Arrays

Generic Vertex Attributes [10.3.1]

void **VertexAttribFormat**(uint *attribindex*, int *size*, enum *type*, boolean *normalized*, uint *relativeoffset*);
type: [UNSIGNED_]BYTE, [UNSIGNED_]SHORT, [UNSIGNED_]INT, [HALF_]FLOAT, FIXED, [UNSIGNED_]INT_2_10_10_10_REV

void **VertexAttribIFormat**(uint *attribindex*, int *size*, enum *type*, uint *relativeoffset*);
type: ***See VertexAttribFormat***

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

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

void **DisableVertexAttribArray**(uint *index*);

Vertex Attribute Divisors [10.3.2]

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

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

Primitive Restart [10.3.4]

Enable/Disable/IsEnabled(*target*);
target: PRIMITIVE_RESTART_FIXED_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, TRIANGLE_FAN, TRIANGLE_STRIP, LINES, TRIANGLES
type: UNSIGNED_{BYTE, SHORT, INT}

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

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

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

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

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

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

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

Vertex Array Queries [10.6]

void **GetVertexAttrib{f i}v**(uint *index*, enum *pname*, T **params*);
pname: VERTEX_ATTRIB_[BUFFER_]BINDING, VERTEX_ATTRIB_RELATIVE_OFFSET, CURRENT_VERTEX_ATTRIB, or VERTEX_ATTRIB_ARRAY_*X* where *X* may be one of DIVISOR, ENABLED, INTEGER, NORMALIZED, SIZE, STRIDE, TYPE

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

Vertices

Current Vertex Attribute Values [10.2.1]

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

void **VertexAttrib{1234}{f}**(uint *index*, float *values*);

void **VertexAttrib{123}fv**(uint *index*, const float **values*);

void **VertexAttribI4{1234}{i ui}**(uint *index*, T *values*);

void **VertexAttribI4{1234}{i ui}v**(uint *index*, const T **values*);

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

Vertex Attributes [11.1]

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*);

Vertex Shader 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[_VEC*n*], INT, UNSIGNED_INT, [UNSIGNED_]INT_VEC*n*, FLOAT_MAT*nxm*, FLOAT_MAT*n*, *BOOL, BOOL_VEC2, BOOL_VEC3, BOOL_VEC4*

Shader Validation [11.1.3]

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

Vertex Post-Processing [12]

Transform Feedback [12.1]

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);

Controlling Viewport [12.5.1]

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

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

Shader Execution [14.2.3]

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

Rasterization

Multisampling [13.2.1]

Use to antialias points and lines.

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

Points [13.3]

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

Line Segments [13.4]

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

Polygons [13.5, 13.5.1]

void **FrontFace**(enum *dir*);
dir: CCW, CW

Enable(CULL_FACE)
Disable(CULL_FACE)
IsEnabled(CULL_FACE)

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

Enable(POLYGON_OFFSET_FILL)
Disable(POLYGON_OFFSET_FILL)
IsEnabled(POLYGON_OFFSET_FILL)

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

Per-Fragment Operations

Scissor Test [15.1.2]

Enable/Disable(SCISSOR_TEST);

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

Multisample Fragment Ops. [15.1.3]

Enable/Disable(*cap*);
cap: SAMPLE_ALPHA_TO_COVERAGE, SAMPLE_COVERAGE

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

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

Stencil Test [15.1.4]

Enable/Disable(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***
face: FRONT, BACK, FRONT_AND_BACK

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

(Continued on next page) ▶

◀ Per-Fragment Operations (continued)

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 [15.1.5]

Enable/Disable(DEPTH_TEST);

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

Blending [15.1.7]

Enable/Disable/IsEnabled(BLEND);

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

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

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

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, DST, CONSTANT}_COLOR, {SRC, DST, CONSTANT}_ALPHA, ONE_MINUS_SRC_{COLOR, ALPHA}, ONE_MINUS_{DST, CONSTANT}_COLOR, ONE_MINUS_{DST, CONSTANT}_ALPHA

Dithering [15.1.9]

Enable/Disable/IsEnabled(DITHER);

Reading and Copying Pixels

Reading Pixels [16.1]

void **ReadBuffer**(enum *src*);
src: BACK, NONE, or COLOR_ATTACHMENT*i* where *i* may range from zero to the value of MAX_COLOR_ATTACHMENTS - 1

void **ReadPixels**(int *x*, int *y*, sizei *width*, sizei *height*, enum *format*, enum *type*, void **data*);
format: RGBA, RGBA_INTEGER
type: INT, UNSIGNED_INT_2_10_10_10_REV, UNSIGNED_{BYTE, INT}
Note: *[4.3.1] ReadPixels() also accepts a queriable implementation-chosen format/type combination.*

Copying Pixels [16.1.2, 8.4.2]

void **BlitFramebuffer**(int *srcX0*, int *srcY0*, int *srcX1*, int *srcY1*, int *dstX0*, int *dstY0*, int *dstX1*, int *dstY1*, bitfield *mask*, enum *filter*);
mask: Zero or Bitwise OR of {COLOR, DEPTH, STENCIL}_BUFFER_BIT
filter: LINEAR or NEAREST

Hints [18.1]

void **Hint**(enum *target*, enum *hint*);
target: FRAGMENT_SHADER_DERIVATIVE_HINT, GENERATE_MIPMAP_HINT,
hint: FASTEST, NICEST, DONT_CARE

Compute Shaders [17]

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

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

Whole Framebuffer Operations

Selecting Buffers for Writing [15.2.1]

void **DrawBuffers**(sizei *n*, const enum **bufs*);
bufs points to an array of *n* BACK, NONE, or COLOR_ATTACHMENT*i* where *i* = [0,MAX_COLOR_ATTACHMENTS - 1].

Fine Control of Buffer Updates [15.2.2]

void **ColorMask**(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 [15.2.3]

void **Clear**(bitfield *buf*);
buf: Zero or Bitwise OR of COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, STENCIL_BUFFER_BIT

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

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 Framebuffer Contents [15.2.4]

void **InvalidateSubFramebuffer**(enum *target*, sizei *numAttachments*, const enum **attachments*, int *x*, int *y*, sizei *width*, sizei *height*);
target: FRAMEBUFFER, {DRAW, READ}_FRAMEBUFFER
attachments: points to an array of COLOR, STENCIL, {DEPTH, STENCIL}_ATTACHMENT, COLOR_ATTACHMENT*i*

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

Context State Queries

A complete list of symbolic constants for states is shown in the tables in [20].

Simple Queries [19.1]

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

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

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

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

String Queries [19.2]

ubyte ***GetString**(enum *name*);
name: VENDOR, RENDERER, EXTENSIONS, [SHADING_LANGUAGE_]VERSION

ubyte ***GetStringi**(enum *name*, uint *index*);
name: EXTENSIONS

Internal Format Queries [19.3]

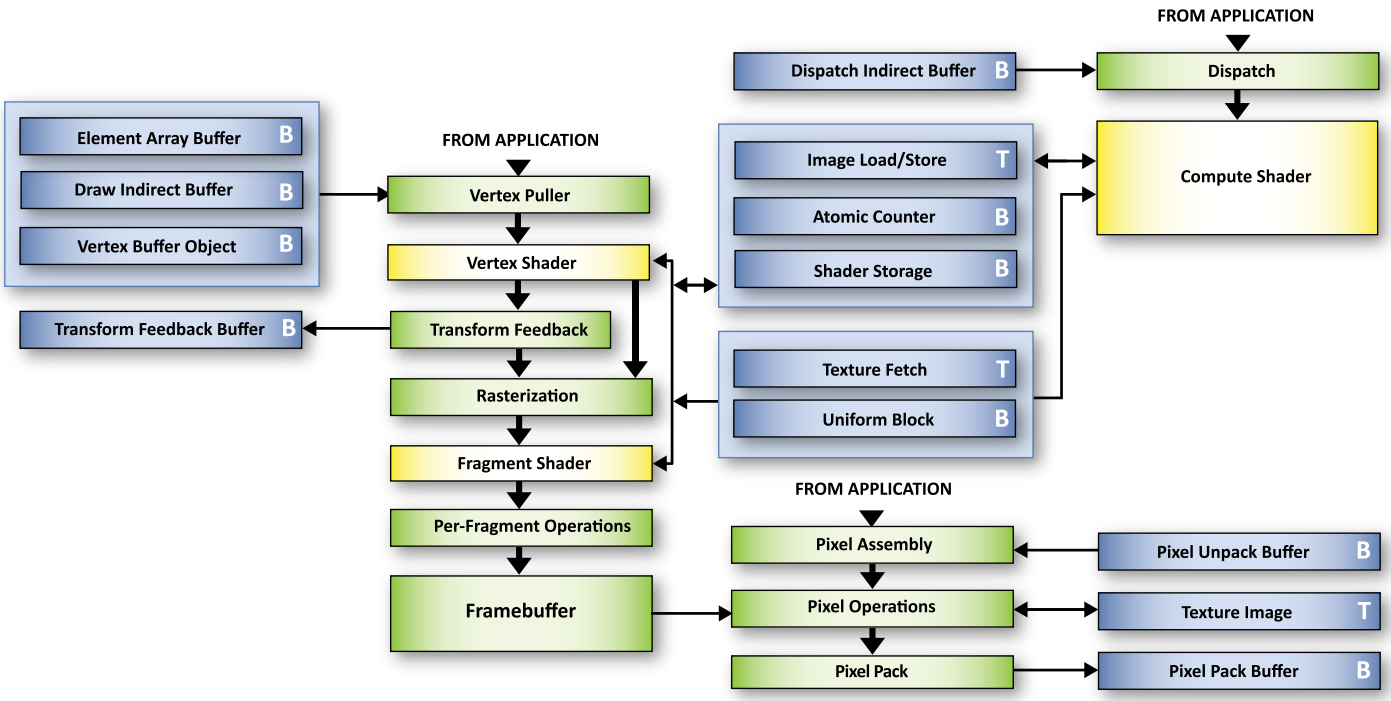
void **GetInternalformativ**(enum *target*, enum *internalformat*, enum *pname*, sizei *bufSize*, int **params*);
target: TEXTURE_2D_MULTISAMPLE, RENDERBUFFER
internalformat: See RenderbufferStorageMultisample
pname: SAMPLES, NUM_SAMPLES_COUNTS

OpenGL ES Pipeline

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

The heavy black arrows in this illustration show the OpenGL ES pipeline and indicate data flow.

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



The OpenGL® ES Shading Language is three closely-related languages which are used to create shaders for the vertex and fragment processors contained in the OpenGL ES processing pipeline.

[n.n.n] and **[Table n.n]** refer to sections and tables in the OpenGL ES Shading Language 3.10 specification at www.khronos.org/registry/gles/

Types [4.1]

A shader can aggregate these using arrays and structures to build more complex types. There are no pointer types.

Basic Types

void	no return value or empty parameter list
bool	Boolean
int, uint	signed, unsigned integer
float	floating scalar
vec2, vec3, vec4	n-component floating point vector
bvec2, bvec3, bvec4	Boolean vector
ivec2, ivec3, ivec4	signed integer vector
uvec2, uvec3, uvec4	unsigned integer vector
mat2, mat3, mat4	2x2, 3x3, 4x4 float matrix
mat2x2, mat2x3, mat2x4	2x2, 2x3, 2x4 float matrix
mat3x2, mat3x3, mat3x4	3x2, 3x3, 3x4 float matrix
mat4x2, mat4x3, mat4x4	4x2, 4x3, 4x4 float matrix

Signed Integer Sampler Types (opaque)

isampler2D, isampler3D	access an integer 2D or 3D texture
isamplerCube	access integer cube mapped texture
isampler2DArray	access integer 2D array texture

Preprocessor [3.4]

Preprocessor Directives

The number sign (#) can be immediately preceded or followed in its line by spaces or horizontal tabs.

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

Examples of Preprocessor Directives

- "#version 310 es" must appear in the first line of a shader program written in GLSL ES version 3.10. If omitted, the shader will be treated as targeting version 1.00.
- #extension *extension_name* : *behavior*, where *behavior* can be require, enable, warn, or disable; and where *extension_name is* the extension supported by the compiler
- #pragma optimize({on, off}) - enable or disable shader optimization (default on)
  #pragma debug({on, off}) - enable or disable compiling shaders with debug information (default off)

Predefined Macros

__LINE__	Decimal integer constant that is one more than the number of preceding newlines in the current source string
__FILE__	Decimal integer constant that says which source string number is currently being processed.
__VERSION__	Decimal integer, e.g.: 310
GL_ES	Defined and set to integer 1 if running on an OpenGL-ES Shading Language.

Signed Integer Sampler Types (continued)

isampler2DMS	access an integer 2D multisample texture
iimage2D	access an integer 2D image
iimage3D	access an integer 3D image
iimageCube	access an integer image cube
iimage2DArray	access a 2D array of integer images

Unsigned Integer Sampler Types (opaque)

usampler2D, usampler3D	access unsigned integer 2D or 3D texture
usamplerCube	access unsigned integer cube mapped texture
usampler2DArray	access unsigned integer 2D array texture
atomic_uint	access an unsigned atomic counter

Unsigned Integer Sampler Types (continued)

usampler2DMS	access unsigned integer 2D multisample texture
uimage2D	access an unsigned integer 2D image
uimage3D	access an unsigned integer 3D image
uimageCube	access an unsigned integer image cube
uimage2DArray	access a 2D array of unsigned integer images

Floating Point Sampler Types (opaque)

sampler2D, sampler3D	access a 2D or 3D texture
samplerCube	access cube mapped texture
samplerCubeShadow	access cube map depth texture w/comparison
sampler2DShadow	access 2D depth texture with comparison
sampler2DArray	access 2D array texture
sampler2DArrayShadow	access 2D array depth texture with comparison
sampler2DMS	access a 2D multisample texture
image2D	access a 2D image
image3D	access a 3D image
imageCube	access an image cube
image2DArray	access a 2D array of images

Structures and Arrays [4.1.8, 4.1.9]

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

Arrays	float foo[3]; Structures, blocks, and structure members can be arrays. Only 1-dimensional arrays supported.

Qualifiers

Storage Qualifiers [4.3, 4.5]

Variable declarations may be preceded by one storage qualifier specified in front of the type.

const	Compile-time constant, or read-only function parameter.
in	Linkage into a shader from a previous stage
out	Linkage out of a shader to a subsequent stage
uniform	Value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL ES, and the application
buffer	Buffer object
shared	Compute shader storage shared across local workgroup

Auxiliary Storage Qualifiers

Some input and output qualified variables can be qualified with at most one additional auxiliary storage qualifier.

centroid	Centroid-based interpolation

Interpolation Qualifiers

Shader inputs and outputs can be further qualified with one of these interpolation qualifiers.

smooth	Perspective-corrected interpolation
flat	No interpolation

Uniform Variables [4.3.5]

Used to declare read-only global variables whose values are the same across the entire primitive being processed, for example:

uniform vec4 lightPosition;

Buffer Qualifier and Interface Blocks [4.3.7-9]

A group of **uniform** or **buffer** variable declarations. Buffer variables may only be declared inside interface blocks. Example:

```
buffer BufferName {  // externally visible name of buffer
    int count;       // typed, shared memory...
    ...              // ...
    vec4 v[];        // last element may be an unsized array
                     // until after link time (dynamically sized)
} Name;              // name of block within the shader
```

Layout Qualifiers [4.4]

layout-qualifier :
layout(*layout-qualifier-id-list*)

Input Layout Qualifier Examples

All shaders except compute shaders allow a location qualifier:
layout(**location** = 3) in vec4 normal;

Fragment shaders allow:
layout(**early_fragment_tests**) in;

Compute shaders allow work-group size qualifiers:
layout(**local_size_x** = 32, **local_size_y** = 32) in;

Output Layout Qualifier Example

Vertex and fragment shaders allow a location qualifier:
layout(**location** = 3) out vec4 color;

Uniform Variable Layout Qualifier Example

For all default-block uniform variables but not for variables in uniform or shader storage blocks:

layout-qualifier-id :
location = *integer-constant*

Uniform and Shader Storage Block Layout Qualifiers

Layout qualifiers can be used for uniform and shader storage blocks, but not for non-block uniform declarations:

layout-qualifier-id :
shared
packed
std140
std430
row_major
column_major
binding = *integer-constant*

Opaque Uniform Layout Qualifier Example

Sampler, image and atomic counter types take the uniform layout qualifier identifier for binding:

```
layout(binding = 3) uniform sampler2D s;
                                  // s bound to unit 3
```

Atomic Counter Layout Qualifier Example

layout(**binding** = 2, **offset** = 4) uniform atomic_uint a;

Format Layout Qualifiers

layout-qualifier-id :
float-image-format-qualifier
int-image-format-qualifier
uint-image-format-qualifier
binding = *integer-constant*

Format Layout Qualifiers (continued)

float-image-format-qualifier:	int-image-format-qualifier:	uint-image-format-qualifier:
rgba32f	**rgba32i**	**rgba32ui**
rgba16f	**rgba16i**	**rgba16ui**
r32f	**rgba8i**	**rgba8ui**
rgba8	**r32i**	**r32ui**
rgba8_snorm		

Parameter Qualifiers [4.6]

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

none	(Default) same as **in**
in	For parameter passed into a function
out	For values passed out of a function
inout	Function parameters passed in and out

(Continued on next page) ▶

◀ Qualifiers (continued)

Precision and Precision Qualifiers [4.7]

Example of precision qualifiers:

```
lowp float color;
out mediump vec2 P;
lowp ivec2 foo(lowp mat3);
highp mat4 m;
```

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

```
precision mediump int;
precision lowp sampler2D;
precision highp atomic_uint;
```

Invariant Qualifiers Examples [4.8.1]

```
invariant gl_Position;  // make built-in gl_Position be invariant
invariant centroid out vec3 Color;
```

To force all output variables to be invariant:

```
#pragma STDGL invariant(all)
```

Memory Access Qualifiers [4.9]

coherent	Reads and writes are coherent with other shader invocations
volatile	Underyling value can change at any time
restrict	A variable that is the exclusive way to access a value
readonly	Read only
writeonly	Write only

Statements and Structure

Iteration and Jumps [6.3, 6.4]

Entry	void main()

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

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

Jump	break, continue, return discard // Fragment shader only

Operators and Expressions

Operators [5.1]

Numbered in order of precedence. The relational and equality operators > < <= >= == != evaluate to a Boolean. To compare vectors component-wise, use functions such as lessThan(), equal(), etc. [8.7].

	Operator	Description	Assoc.
1.	( )	parenthetical grouping	N/A
2.	[ ] ( ) . ++ --	array subscript function call & constructor structure field or method selector, swizzler postfix increment and decrement	L - R
3.	++ -- + - ~ !	prefix increment and decrement unary	R - L
4.	* % /	multiplicative	L - R
5.	+ -	additive	L - R
6.	<< >>	bit-wise shift	L - R
7.	< > <= >=	relational	L - R
8.	== !=	equality	L - R
9.	&	bit-wise and	L - R
10.	^	bit-wise exclusive or	L - R
11.	|	bit-wise inclusive or	L - R
12.	&&	logical and	L - R
13.	^^	logical exclusive or	L - R
14.	||	logical inclusive or	L - R
15.	? :	selection (Selects an entire operand. Use mix() to select individual components of vectors.)	R - L
16.	= += -= *= /= %= <<= >>= &= ^= |=	assignment arithmetic assignments	R - L
17.	,	sequence	L - R

Vector Components [5.5]

In addition to array numeric subscript syntax, names of vector components are denoted by a single letter. Components can be swizzled and replicated, e.g.: pos.xx, pos.zy

{x, y, z, w}	Use when accessing vectors that represent points or normals
{r, g, b, a}	Use when accessing vectors that represent colors
{s, t, p, q}	Use when accessing vectors that represent texture coordinates

Aggregate Operations and Constructors

Matrix Constructor Examples [5.4.2]

```
mat2(float)                    // init diagonal
mat2(vec2, vec2);              // column-major order
mat2(float, float,
     float, float);            // column-major order
```

Structure Constructor Example [5.4.3]

```
struct light {
  float intensity;
  vec3 pos;
};
light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));
```

Matrix Components [5.6]

Access components of a matrix with array subscripting syntax. For example:

```
mat4 m;          // m represents a matrix
m[1] = vec4(2.0); // sets second column 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 column 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 component-wise addition/subtraction
m = m * m;       // linear algebraic multiply
m = v * m;       // row vector * matrix linear algebraic multiply
m = m * v;       // matrix * column vector linear algebraic multiply
f = dot(v, v);   // vector dot product
v = cross(v, v); // vector cross product
m = matrixCompMult(m, m);  // component-wise multiply
```

Structure Operations [5.7]

Select structure fields using the period (.) operator. Valid operators are:

.	field selector
== !=	equality
=	assignment

Array Operations [5.7]

Array elements are accessed using the array subscript operator "[ ]". For example:

```
diffuseColor += lightIntensity[3] * NdotL;
```

The size of an array can be determined using the **.length()** operator. For example:

```
for (i = 0; i < a.length(); i++)
    a[i] = 0.0;
```

Built-In Inputs, Outputs, and Constants [7]

Shader programs use special variables to communicate with fixed-function parts of the pipeline. Output special variables may be read back after writing. Input special variables are read-only. All special variables have global scope.

Vertex Shader Special Variables [7.1.1]

```
in   highp   int   gl_VertexID;
in   highp   int   gl_InstanceID;
out  highp   vec4  gl_Position;
out  highp   float gl_PointSize;
```

Fragment Shader Special Variables [7.1.2]

```
in   highp   vec4  gl_FragCoord;
in   bool          gl_FrontFacing;
out  highp   float gl_FragDepth;
in   mediump vec2  gl_PointCoord;
in           bool  gl_HelperInvocation;
```

Compute Shader Special Variables [7.1.3]

Work group dimensions:

```
in     uvec3  gl_NumWorkGroups;
const  uvec3  gl_WorkGroupSize;
```

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 With Minimum Values [7.2]

Built-in Constant	Min.
const mediump int gl_MaxVertexAttribs	16
const mediump int gl_MaxVertexUniformVectors	256
const mediump int gl_MaxVertexOutputVectors	16
const mediump int gl_MaxFragmentInputVectors	15
const mediump int gl_MaxFragmentUniformVectors	224
const mediump int gl_MaxDrawBuffers	4
const mediump int gl_MaxVertexTextureImageUnits	16
const mediump int gl_MaxCombinedTextureImageUnits	48
const mediump int gl_MaxTextureImageUnits	16
const mediump int gl_MinProgramTexelOffset	-8
const mediump int gl_MaxProgramTexelOffset	7
const mediump int gl_MaxImageUnits	4
const mediump int gl_MaxVertexImageUniforms	0

Built-in Constant	Min.
const mediump int gl_MaxFragmentImageUniforms	0
const mediump int gl_MaxComputeImageUniforms	4
const mediump int gl_MaxCombinedImageUniforms	4
const mediump int gl_MaxCombinedShaderOutputResources	4
const highp ivec3 gl_MaxComputeWorkGroupCount = ivec3(65535, 65535, 65535);	
const highp ivec3 gl_MaxComputeWorkGroupSize = ivec3(128, 128, 64);	
const mediump int gl_MaxComputeUniformComponents	512
const mediump int gl_MaxComputeTextureImageUnits	16
const mediump int gl_MaxComputeAtomicCounters	8
const mediump int gl_MaxComputeAtomicCounterBuffers	1
const mediump int gl_MaxVertexAtomicCounters	0
const mediump int gl_MaxFragmentAtomicCounters	0
const mediump int gl_MaxCombinedAtomicCounters	8

Built-in Constant	Min.
const mediump int gl_MaxAtomicCounterBindings	1
const mediump int gl_MaxVertexAtomicCounterBuffers	0
const mediump int gl_MaxFragmentAtomicCounterBuffers	0
const mediump int gl_MaxVertexAtomicCounterBuffers	0
const mediump int gl_MaxCombinedAtomicCounterBuffers	1
const mediump int gl_MaxAtomicCounterBufferSize	32

Built-In Uniform State [7.4]

As an aid to accessing OpenGL ES processing state, the following uniform variables are built into the OpenGL ES Shading Language.

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

Built-In Functions

Angle & Trigonometry Functions [8.1]

Component-wise operation. Parameters specified as *angle* are assumed to be in units of radians. T is float, vec2, vec3, vec4.

T **radians** (T *degrees*);	Degrees to radians
T **degrees** (T *radians*);	Radians to degrees
T **sin** (T *angle*);	Sine
T **cos** (T *angle*);	Cosine
T **tan** (T *angle*);	Tangent
T **asin** (T *x*);	Arc sine
T **acos** (T *x*);	Arc cosine
T **atan** (T *y*, T *x*); T **atan** (T *y_over_x*);	Arc tangent
T **sinh** (T *x*);	Hyperbolic sine
T **cosh** (T *x*);	Hyperbolic cosine
T **tanh** (T *x*);	Hyperbolic tangent
T **asinh** (T *x*);	Arc hyperbolic sine; inverse of sinh
T **acosh** (T *x*);	Arc hyperbolic cosine; non-negative inverse of cosh
T **atanh** (T *x*);	Arc hyperbolic tangent; inverse of tanh

Exponential Functions [8.2]

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

T **pow** (T *x*, T *y*);	x^y
T **exp** (T *x*);	e^x
T **log** (T *x*);	ln
T **exp2** (T *x*);	2^x
T **log2** (T *x*);	$\log_2$
T **sqrt** (T *x*);	Square root
T **inversesqrt** (T *x*);	Inverse square root

Common Functions [8.3]

Component-wise operation. T is float and vec*n*, Ti is int and ivec*n*, Tu is uint and uvec*n*, and Tb is bool and bvec*n*, where *n* is 2, 3, or 4.

T **abs**(T *x*); Ti **abs**(Ti *x*);	Absolute value
T **sign**(T *x*); Ti **sign**(Ti *x*);	Returns -1.0, 0.0, or 1.0
T **floor**(T *x*);	Nearest integer <= *x*
T **trunc** (T *x*);	Nearest integer a such that |*a*| <= |*x*|
T **round** (T *x*);	Round to nearest integer
T **roundEven** (T *x*);	Round to nearest integer
T **ceil**(T *x*);	Nearest integer >= *x*
T **fract**(T *x*);	*x* - **floor**(*x*)
T **mod**(T *x*, T *y*); T **mod**(T *x*, float *y*); T **modf**(T *x*, out T *i*);	Modulus
T **min**(T *x*, T *y*); Ti **min**(Ti *x*, Ti *y*); Tu **min**(Tu *x*, Tu *y*); T **min**(T *x*, float *y*); Ti **min**(Ti *x*, int *y*); Tu **min**(Tu *x*, uint *y*);	Minimum value
T **max**(T *x*, T *y*); Ti **max**(Ti *x*, Ti *y*); Tu **max**(Tu *x*, Tu *y*); T **max**(T *x*, float *y*); Ti **max**(Ti *x*, int *y*); Tu **max**(Tu *x*, uint *y*);	Maximum value
T **clamp**(Ti *x*, T *minVal*, T *maxVal*); Ti **clamp**(V *x*, Ti *minVal*, Ti *maxVal*); Tu **clamp**(Tu *x*, Tu *minVal*, Tu *maxVal*); T **clamp**(T *x*, float *minVal*, float *maxVal*); Ti **clamp**(Ti *x*, int *minVal*, int *maxVal*); Tu **clamp**(Tu *x*, uint *minVal*, uint *maxVal*);	**min**(**max**(*x*, *minVal*), *maxVal*)

Common Functions (continued)

T **mix**(T *x*, T *y*, T *a*); T **mix**(T *x*, T *y*, float *a*);	Linear blend of *x* and *y*
T **mix**(T *x*, T *y*, Tb *a*);	Selects vector source for each returned component
T **step**(T *edge*, T *x*); T **step**(float *edge*, T *x*);	0.0 if *x* < *edge*, else 1.0
T **smoothstep**(T *edge0*, T *edge1*, T *x*); T **smoothstep**(float *edge0*, float *edge1*, T *x*);	Clamp and smooth
Tb **isnan**(T *x*);	True if *x* is a NaN
Tb **isinf**(T *x*);	True if *x* is positive or negative infinity
Ti **floatBitsToInt**(T *value*); Tu **floatBitsToUint**(T *value*);	highp integer, preserving float bit level representation
T **intBitsToFloat**(Ti *value*); T **uintBitsToFloat**(Tu *value*);	highp float, preserving integer bit level representation
highp T **frexp**(highp T *x*, out highp Ti *exp*);	Splits each single-precision floating point number
highp T **ldexp**(highp T *x*, in highp Ti *exp*);	Builds a single-precision floating point number

Floating-Point Pack and Unpack Functions [8.4]

highp uint **packSnorm2x16**(vec2 *v*); highp uint **packUnorm2x16**(vec2 *v*);	Convert two floats to fixed point and pack into an integer
highp uint **packSnorm4x8**(mediump vec4 *v*); highp uint **packUnorm4x8**(mediump vec4 *v*);	Convert four floats to fixed point and pack into an integer
highp vec2 **unpackSnorm2x16**(highp uint *p*); highp vec2 **unpackUnorm2x16**(highp uint *p*);	Unpack fixed point value pair into floats
mediump vec4 **unpackSnorm4x8**(highp uint *p*); mediump vec4 **unpackUnorm4x8**(highp uint *p*);	Unpack fixed point values into floats
highp uint **packHalf2x16**(mediump vec2 *v*);	Convert two floats into half-precision floats and pack into an integer
mediump vec2 **unpackHalf2x16**(highp uint *v*);	Unpack half value pair into full floats

Geometric Functions [8.5]

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

float **length**(T *x*);	Length of vector
float **distance**(T *p0*, T *p1*);	Distance between points
float **dot**(T *x*, T *y*);	Dot product
vec3 **cross**(vec3 *x*, vec3 *y*);	Cross product
T **normalize**(T *x*);	Normalize vector to length 1
T **faceforward**(T *N*, T *I*, T *Nref*);	Returns *N* if **dot**(*Nref*, *I*) < 0, else -*N*
T **reflect**(T *I*, T *N*);	Reflection direction *I* - 2 * **dot**(*N*,*I*) * *N*
T **refract**(T *I*, T *N*, float *eta*);	Refraction vector

Matrix Functions [8.6]

Type mat is any matrix type.

mat **matrixCompMult**(mat *x*, mat *y*);	Multiply *x* by *y* component-wise
mat2 **outerProduct**(vec2 *c*, vec2 *r*); mat3 **outerProduct**(vec3 *c*, vec3 *r*); mat4 **outerProduct**(vec4 *c*, vec4 *r*);	Linear algebraic column vector * row vector
mat2x3 **outerProduct**(vec3 *c*, vec2 *r*); mat3x2 **outerProduct**(vec2 *c*, vec3 *r*); mat2x4 **outerProduct**(vec4 *c*, vec2 *r*); mat4x2 **outerProduct**(vec2 *c*, vec4 *r*); mat3x4 **outerProduct**(vec4 *c*, vec3 *r*); mat4x3 **outerProduct**(vec3 *c*, vec4 *r*);	Linear algebraic column vector * row vector

Matrix Functions (continued)

mat2 **transpose**(mat2 *m*); mat3 **transpose**(mat3 *m*); mat4 **transpose**(mat4 *m*); mat2x3 **transpose**(mat3x2 *m*); mat3x2 **transpose**(mat2x3 *m*); mat2x4 **transpose**(mat4x2 *m*); mat4x2 **transpose**(mat2x4 *m*); mat3x4 **transpose**(mat4x3 *m*); mat4x3 **transpose**(mat3x4 *m*);	Transpose of matrix *m*
float **determinant**(mat2 *m*); float **determinant**(mat3 *m*); float **determinant**(mat4 *m*);	Determinant of matrix *m*
mat2 **inverse**(mat2 *m*); mat3 **inverse**(mat3 *m*); mat4 **inverse**(mat4 *m*);	Inverse of matrix *m*

Vector Relational Functions [8.7]

Compare *x* and *y* component-wise. Input and return vector sizes for a particular call must match. Type bvec is bvec*n*; vec is vec*n*; ivec is ivec*n*; uvec is uvec*n*; (where *n* is 2, 3, or 4). T is union of vec and ivec.

bvec **lessThan**(T *x*, T *y*); bvec **lessThan**(uvec *x*, uvec *y*);	*x* < *y*
bvec **lessThanEqual**(T *x*, T *y*); bvec **lessThanEqual**(uvec *x*, uvec *y*);	*x* <= *y*
bvec **greaterThan**(T *x*, T *y*); bvec **greaterThan**(uvec *x*, uvec *y*);	*x* > *y*
bvec **greaterThanEqual**(T *x*, T *y*); bvec **greaterThanEqual**(uvec *x*, uvec *y*);	*x* >= *y*
bvec **equal**(T *x*, T *y*); bvec **equal**(bvec *x*, bvec *y*); bvec **equal**(uvec *x*, uvec *y*);	*x* == *y*
bvec **notEqual**(T *x*, T *y*); bvec **notEqual**(bvec *x*, bvec *y*); bvec **notEqual**(uvec *x*, uvec *y*);	*x* != *y*
bool **any**(bvec *x*);	True if any component of *x* is true
bool **all**(bvec *x*);	True if all components of *x* are true
bvec **not**(bvec *x*);	Logical complement of *x*

Integer Functions [8.8]

Ti **bitfieldExtract**(Ti *value*, int *offset*, int *bits*); Tu **bitfieldExtract**(Tu *value*, int *offset*, int *bits*);	Extracts bits, returning them in the least significant bits of corresponding component of the result
Ti **bitfieldInsert**(Ti *base*, T *insert*, int *offset*, int *bits*); Tu **bitfieldInsert**(Tu *base*, Tu *insert*, int *offset*, int *bits*);	Inserts bits into the corresponding component of base
highp Ti **bitfieldReverse**(highp Ti *value*); highp Tu **bitfieldReverse**(highp Tu *value*);	Reverses the bits of *value*
lowp Ti **bitCount**(Ti *value*); lowp Ti **bitCount**(Tu *value*);	Returns number of one bits in *value*
lowp Ti **findLSB**(Ti *value*); lowp Ti **findLSB**(Tu *value*);	Returns the bit number of the least significant one bit
lowp Ti **findMSB**(highp Ti *value*); lowp Ti **findMSB**(highp Tu *value*);	Returns the bit number of the most significant one bit
highp Tu **uaddCarry**(highp Tu *x*, highp Tu *y*, out lowp Tu *carry*);	Adds 32-bit integer or vector *y* to *x*
highp Tu **usubBorrow**(highp Tu *x*, highp Tu y, out lowp Tu *borrow*);	Subtracts 32-bit unsigned integer or vector *y* from *x*
void **umulExtended**(highp Tu *x*, highp Tu *y*, out highp Tu *msb*, out highp Tu *lsb*); void **imulExtended**(highp Ti *x*, highp Ti *y*, out highp Ti *msb*, out highp Ti *lsb*);	Multiply 32-bit integers or vectors to produce a 64-bit result

(Continued on next page) ▶

◀ Built-In Functions (continued)

Texture Query Functions [8.9]

The function textureSize returns the dimensions of level *lod* for the texture bound to sampler, as described in [11.1.3.4] of the OpenGL ES 3.1 specification, under "Texture Queries". The initial "g" in a type name is a placeholder for nothing, "i", or "u".

highp ivec2	**textureSize**(gsampler2D *sampler*, int *lod*);
highp ivec3	**textureSize**(gsampler3D *sampler*, int *lod*);
highp ivec2	**textureSize**(gsamplerCube *sampler*, int *lod*);
highp ivec2	**textureSize**(gsampler2DMS *sampler*);
highp ivec3	**textureSize**(gsampler2DArray *sampler*, int *lod*);
highp ivec2	**textureSize**(samplerCubeShadow *sampler*, int *lod*);
highp ivec2	**textureSize**(sampler2DShadow *sampler*, int *lod*);
highp ivec3	**textureSize**(sampler2DArrayShadow *sampler*, int *lod*);

Texture Lookup Functions

Texture lookup functions using samplers are available to vertex and fragment shaders. The initial "g" in a type name is a placeholder for nothing, "i", or "u".

gvec4	**texture**(gsampler{2,3}D *sampler*, vec{2,3} *P* [, float *bias*]);
gvec4	**texture**(gsamplerCube *sampler*, vec3 *P* [, float *bias*]);
float	**texture**(sampler2DShadow *sampler*, vec3 *P* [, float *bias*]);
float	**texture**(samplerCubeShadow *sampler*, vec4 *P* [, float *bias*]);
gvec4	**texture**(gsampler2DArray *sampler*, vec3 *P* [, float *bias*]);
float	**texture**(sampler2DArrayShadow *sampler*, vec4 *P*);
gvec4	**textureProj**(gsampler2D *sampler*, vec{3,4} *P* [, float *bias*]);
gvec4	**textureProj**(gsampler3D *sampler*, vec4 *P* [, float *bias*]);
float	**textureProj**(sampler2DShadow *sampler*, vec4 *P* [, float *bias*]);
gvec4	**textureLod**(gsampler{2,3}D *sampler*, vec{2,3} *P*, float *lod*);
gvec4	**textureLod**(gsamplerCube *sampler*, vec3 *P*, float *lod*);
float	**textureLod**(sampler2DShadow *sampler*, vec3 *P*, float *lod*);
gvec4	**textureLod**(gsampler2DArray *sampler*, vec3 *P*, float *lod*);
gvec4	**textureOffset**(gsampler2D *sampler*, vec2 *P*, ivec2 *offset* [, float *bias*]);
gvec4	**textureOffset**(gsampler3D *sampler*, vec3 *P*, ivec3 *offset* [, float *bias*]);
float	**textureOffset**(sampler2DShadow *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]);
gvec4	**textureOffset**(gsampler2DArray *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]);
gvec4	**texelFetch**(gsampler2D *sampler*, ivec2 *P*, int *lod*);
gvec4	**texelFetch**(gsampler3D *sampler*, ivec3 *P*, int *lod*);
gvec4	**texelFetch**(gsampler2DArray *sampler*, ivec3 *P*, int *lod*);
gvec4	**texelFetch**(gsampler2DMS *sampler*, ivec2 *P*, int *sample*);
gvec4	**texelFetchOffset**(gsampler2D *sampler*, ivec2 *P*, int *lod*, ivec2 *offset*);
gvec4	**texelFetchOffset**(gsampler3D *sampler*, ivec3 *P*, int *lod*, ivec3 *offset*);
gvec4	**texelFetchOffset**(gsampler2DArray *sampler*, ivec3 *P*, int *lod*, ivec2 *offset*);
gvec4	**textureProjOffset**(gsampler2D *sampler*, vec3 *P*, ivec2 *offset* [, float *bias*]);
gvec4	**textureProjOffset**(gsampler2D *sampler*, vec4 *P*, ivec2 *offset* [, float *bias*]);
gvec4	**textureProjOffset**(gsampler3D *sampler*, vec4 *P*, ivec3 *offset* [, float *bias*]);
float	**textureProjOffset**(sampler2DShadow *sampler*, vec4 *P*, ivec2 *offset* [, float bias]);
gvec4	**textureLodOffset**(gsampler2D *sampler*, vec2 *P*, float *lod*, ivec2 *offset*);
gvec4	**textureLodOffset**(gsampler3D *sampler*, vec3 *P*, float *lod*, ivec3 *offset*);
float	**textureLodOffset**(sampler2DShadow *sampler*, vec3 *P*, float *lod*, ivec2 *offset*);
gvec4	**textureLodOffset**(gsampler2DArray *sampler*, vec3 *P*, float *lod*, ivec2 *offset*);
gvec4	**textureProjLod**(gsampler2D *sampler*, vec3 *P*, float *lod*);
gvec4	**textureProjLod**(gsampler2D *sampler*, vec4 *P*, float *lod*);
gvec4	**textureProjLod**(gsampler3D *sampler*, vec4 *P*, float *lod*);
float	**textureProjLod**(sampler2DShadow *sampler*, vec4 *P*, float *lod*);
gvec4	**textureProjLodOffset**(gsampler2D *sampler*, vec3 *P*, float *lod*, ivec2 *offset*);
gvec4	**textureProjLodOffset**(gsampler2D *sampler*, vec4 *P*, float *lod*, ivec2 *offset*);
gvec4	**textureProjLodOffset**(gsampler3D *sampler*, vec4 *P*, float *lod*, ivec3 *offset*);
float	**textureProjLodOffset**(sampler2DShadow *sampler*, vec4 *P*, float *lod*, ivec2 *offset*);
gvec4	**textureGrad**(gsampler2D *sampler*, vec2 *P*, vec2 *dPdx*, vec2 *dPdy*);
gvec4	**textureGrad**(gsampler3D *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy*);
gvec4	**textureGrad**(gsamplerCube *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy*);
float	**textureGrad**(sampler2DShadow *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*);
float	**textureGrad**(samplerCubeShadow *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*);
gvec4	**textureGrad**(gsampler2DArray *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*);
float	**textureGrad**(sampler2DArrayShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*);

Texture Lookup Functions (continued)

gvec4	**textureGradOffset**(gsampler2D *sampler*, vec2 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*);
gvec4	**textureGradOffset**(gsampler3D *sampler*, vec3 *P*, vec3 *dPdx*, vec3 *dPdy*, ivec3 *offset*);
float	**textureGradOffset**(sampler2DShadow *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*);
gvec4	**textureGradOffset**(gsampler2DArray *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*);
float	**textureGradOffset**(sampler2DArrayShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*);
gvec4	**textureProjGrad**(gsampler2D *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*);
gvec4	**textureProjGrad**(gsampler2D *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*);
gvec4	**textureProjGrad**(gsampler3D *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*);
float	**textureProjGrad**(sampler2DShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*);
gvec4	**textureProjGradOffset**(gsampler2D *sampler*, vec3 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*);
gvec4	**textureProjGradOffset**(gsampler2D *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*);
gvec4	**textureProjGradOffset**(gsampler3D *sampler*, vec4 *P*, vec3 *dPdx*, vec3 *dPdy*, ivec3 *offset*);
float	**textureProjGradOffset**(sampler2DShadow *sampler*, vec4 *P*, vec2 *dPdx*, vec2 *dPdy*, ivec2 *offset*);

Texture Gather Functions

Texture gather functions take components of a single floating-point vector operand as a texture coordinate and return one component from each texel in a four-component result vector.

gvec4	**textureGather**(gsampler2D *sampler*, vec2 *P*[, int *comp*]);
gvec4	**textureGather**(gsampler2DArray *sampler*, vec3 *P*[, int *comp*]);
gvec4	**textureGather**(gsamplerCube *sampler*, vec3 *P*[, int *comp*]);
vec4	**textureGather**(sampler2DShadow *sampler*, vec2 *P*, float *refZ*);
vec4	**textureGather**(sampler2DArrayShadow *sampler*, vec3 *P*, float *refZ*);
vec4	**textureGather**(samplerCubeShadow *sampler*, vec3 *P*, float *refZ*);
gvec4	**textureGatherOffset**(gsampler2D *sampler*, vec2 *P*, ivec2 *offset*[,int *comp*]);
gvec4	**textureGatherOffset**(gsampler2DArray *sampler*, vec3 *P*, ivec2 *offset*[, int *comp*]);
vec4	**textureGatherOffset**(sampler2DShadow *sampler*, vec2 *P*, float *refZ*, ivec2 *offset*);
vec4	**textureGatherOffset**(sampler2DArrayShadow *sampler*, vec3 *P*, float *refZ*, ivec2 *offset*);

Atomic-Counter Functions [8.10]

Returns the value of an atomic counter.

uint **atomicCounterIncrement**(atomic_uint *c*);	Increments the counter and returns its value prior to the increment
uint **atomicCounterDecrement**(atomic_uint *c*);	Decrements the counter and returns its value prior to the decrement
uint **atomicCounter**(atomic_uint *c*);	Returns counter value

Atomic Memory Functions [8.11]

Atomic memory functions perform atomic operations on an individual signed or unsigned integer stored in buffer-object or shared-variable storage.

uint **atomicAdd**(coherent inout uint *mem*, uint *data*); int **atomicAdd**(coherent inout int *mem*, int *data*);	Adds the value of *data* to *mem*
uint **atomicMin**(coherent inout uint *mem*, uint *data*); int **atomicMin**(coherent inout int *mem*, int *data*);	Minimum of value of *data* and *mem*
uint **atomicMax**(inout uint *mem*, uint *data*); int **atomicMax**(inout int *mem*, int *data*);	Maximum of value of *data* and *mem*
uint **atomicAnd**(coherent inout uint *mem*, uint *data*); int **atomicAnd**(coherent inout int *mem*, int *data*);	Bit-wise AND of value of *data* and *mem*
uint **atomicOr**(coherent inout uint *mem*, uint *data*); int **atomicOr**(coherent inout int *mem*, int *data*);	Bit-wise OR of value of *data* and *mem*

Atomic Memory Functions (continued)

uint **atomicXor**(coherent inout uint *mem*, uint *data*); int **atomicXor**(coherent inout int *mem*, int *data*);	Bit-wise EXCLUSIVE of value of *data* and *mem*
uint **atomicExchange**(coherent inout uint *mem*, uint *data*); int **atomicExchange**(coherent inout int *mem*, int *data*);	Copy the value of *data*
uint **atomicCompSwap**(coherent inout uint *mem*, uint *compare*, uint *data*); int **atomicCompSwap**(coherent inout int *mem*, int *compare*, int *data*);	Compares *compare* and the contents of *mem*. If equal, returns *data*; else *mem*

Image Functions [8.12]

Image functions read and write individual texels of a texture. Each image variable references an image unit, which has a texture image attached. Type gvec is ivec or uvec. The placeholder gimage may be image, iimage, or uimage.

The IMAGE_PARAMS placeholder is replaced by one of the following parameter lists:

gimage2D *image*, ivec2 *P*
gimage3D *image*, ivec3 *P*
gimageCube *image*, ivec3 *P*
gimage2DArray *image*, ivec3 *P*

highp ivec2 **imageSize**(readonly writeonly gimage2D *image*);
highp ivec3 **imageSize**(readonly writeonly gimage3D *image*);
highp ivec2 **imageSize**(readonly writeonly gimageCube *image*);
highp ivec3 **imageSize**(readonly writeonly gimage2DArray *image*);
highp gvec4 **imageLoad**(readonly *IMAGE_PARAMS*);
void **imageStore**(writeonly *IMAGE_PARAMS*, gvec4 *data*);

Fragment Processing Functions [8.13]

Approximated using local differencing.

T **dFdx**(T *p*);	Derivative in *x*
T **dFdy**(T *p*);	Derivative in *y*
T **fwidth**(T *p*);	**abs** (**dFdx** (*p*)) + **abs** (**dFdy** (*p*));

Shader Invocation Control Function [8.15]

The shader invocation control function controls the relative execution order of multiple shader invocations.

void **barrier**();	All invocations for a single work group must enter **barrier**() before any will continue beyond it

Shader Memory Control Functions [8.16]

Shader memory control functions control the ordering of memory transactions issued by or within a single shader invocation.

void **memoryBarrier**();	Control the ordering of memory transactions
void **memoryBarrierAtomicCounter**();	Control the ordering of accesses to atomic counter variables
void **memoryBarrierBuffer**();	Control the ordering of memory transactions to buffer variables
void **memoryBarrierImage**();	Control the ordering of memory transactions to images
void **memoryBarrierShared**();	Control the ordering of memory transactions to shared variables. Available only in compute shaders.
void **groupMemoryBarrier**();	Control the ordering of all memory transactions. Available only in compute shaders.