OpenGL Errors [2.3.1]
enum GetError(void);

Graphics Reset Recovery [2.3.2]
enum GetGraphicsResetStatus(void);
 Returns: NO_ERROR, GUILTY_CONTEXT_RESET, (INNOCENT, UNKNOWN) CONTEXT_RESET
GetErrorNoResetNotification();
 Returns: NO_RESET_NOTIFICATION, LOSS_CONTEXT_ON_RESET
Flush and Finish [2.3.3]
void Flush(void); void Finish(void);

Sync Objects [4.1]
void DeleterSync_sync(sync);
sync FenceSyncsyncenumcondition,bitfield,flags);
condition:SYNC_CPU_COMMANDS_COMPLETE
flags:mustbe0

Buffer Objects [6]
void GenBuffers(sizei, n, uint *buffers);
void CreateBuffers(sizei, n, buffers);
void DeleteBuffers(sizei, n, uint *buffers);

Create and Bind Buffer Objects [6.1]
void BindBuffer(enum target, uint buffer);
target: [Table 6.1] (ARRAY, UNIFORM) BUFFER,
(UNIFORM, ATTACHED, QUERY) BUFFER,
COPY (READ, WRITE) BUFFER,
(DRAW) INDIRECT BUFFER,
(ATTACHED, DRAW) TEXTURE BUFFER,
PIXEL UNPACK 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 BUFFER,
TRANSFORM_FEEDBACK BUFFER
void BindBufferBase(enum target, uint index, buffer);
target:See BindBufferRange

Sync Objects and Fences [4.1.1]
void GenSync(const Enum target, uint *ids);
void CreateSync(const Enum target, uint *ids);
void DeleteSyncs(const Enum target, uint *ids);

Create/Modify Buffer Object Data [6.2]
void BufferData(uint target, const void *data, sizeiptr size, uint usage);
void BindBuffer(const Enum target, uint buffer);
void BufferSubData(uint target, const void *data, sizeiptr offset, sizeiptr size);
void BindBufferRange(const Enum target, uint first, sizeiptr count, const void *buffers, const sizeiptr *sizes);

Shaders and Programs
Shader Objects [7.1-2]
uint CreateShader(type);
type: (COMPRESS, FRAGMENT) SHADER,
(Geometry, Vertex) SHADER,
Tessellation Evaluation Control SHADER
void ShaderSource(uint shader, sizeiptr size, const char *const *strings, const int *length);

Copy Between Buffers [6.6]
void CopyBufferSubData(uint target, const void *data, sizeiptr size, uint readTarget, uint writeTarget);

Map/Unmap Buffer Data [6.3]
void *MapBufferRange(enum target, sizeiptr offset, sizeiptr size, const void *data, sizeiptr length);
void *UnmapBufferRange(void *map, sizeiptr offset, sizeiptr size, const void *data);

Flush and Finish [2.3.3]
void Flush(void); void Finish(void);

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](args[], type)[*];
The arguments enclosed in brackets [args[], type[*]] 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, GL type

Asynchronous Queries [4.2.4.1.2]
void GetQueryObjecti(uint target, sizei, int *ids);
void CreateQuery(const Enum target, sizei, uint *ids);
void DeleteQueries(const Enum target, sizei, const int *ids);
void BeginQuery(const Enum target, uint id);
void EndQuery(const Enum target);
void GetQueryObjectiv(uint id, enum name, int *params);

Timer Queries [4.3]
Timer queries track the amount of time needed to fully complete a set of GL commands.
void QueryCounter(const Enum target, const Enum name, const void *data);
GetQueryCounter(const Enum target, const Enum name, const void *data);

Copy Between Buffers [6.6]
void CopyBufferSubData(const Enum target, const void *data, sizeiptr size, const void *readTarget, const void *writeTarget);

Map/Unmap Buffer Data [6.3]
void *MapBufferRange(const Enum target, sizeiptr offset, sizeiptr size, const void *data, sizeiptr length);
void *UnmapBufferRange(void *map, sizeiptr offset, sizeiptr size, const void *data);

Shaders and Programs
Shader Objects [7.1-2]
uint CreateShader(type); type:
(COMPRESS, FRAGMENT) SHADER,
(Geometry, Vertex) SHADER,
Tessellation Evaluation Control SHADER
void ShaderSource(uint shader, sizeiptr size, const char *const *strings, const int *length);
void CompileShader(void);
void ReleaseShaderCompiler(void);
void DeleteShader(void);
void IsShader(void);
void ShaderBinary(sizei count, const uint *shaders, enum binaryformat, const void *binary, sizei length);

(Continued on next page)
Shaders and Programs (cont.)

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 CreateShaderProgram(enun type, uint count, const char * const * names);
void ProgramParameteri(uint program, enum pname, int value);

boolean IsProgram(uint program);

Program Interfaces [7.3.1]

getProgramInterface{f d}v(uint program, enum programInterface, enum pname, int *params);

programInterface:

- ATOMIC_COUNTER_BUFFER, BUFFER_VARABLE, UNIFORM{BLOCK, PROGRAM_INPUT_OUTPUT, SHADER_STORAGE_BUFFER, (GEOMETR{VEXEUR_SUBROUTINE, TES{SS_CONTROL, EVALUATION, SUBROUTINE, (FRAGMENT, COMPUTE), SUBROUTINE, TES{SS_CONTROL, SUBROUTINE, UNIFORM{GEOMETR{VEXEUR_SUBROUTINE, (FRAGMENT, COMPUTE), SUBROUTINE, TRANSFORM, FEEDBACK, BUFFER, VARYING}}

pname:

- PROGRAM_SEPARABLE, PROGRAM_BINARY_RETREIVABLE, HINT

DeleteProgram(uint program);

boolean IsProgram(uint program);

OpenGL 4.5 API Reference Card
void TexSubImage1D(enum target, int, level, int xoffset, width, format, type, const void *data);

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

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

void CopyTexSubImage1D(enum target, int, level, int xoffset, width, format, type, const void *data);

void CopyTexSubImage2D(enum target, int, level, int xoffset, int yoffset, width, height, format, type, const void *data);

void CopyTexSubImage3D(enum target, int, level, int xoffset, int yoffset, int zoffset, width, height, depth, format, type, const void *data);

void GetTexImage1D(enum target, int, level, sizei, format, type, const void *data);

void GetTexImage2D(enum target, int, level, sizei, sizei, format, type, const void *data);

void GetTexImage3D(enum target, int, level, sizei, sizei, sizei, format, type, const void *data);

void GetCompressedTexImage1D(enum target, int, level, sizei, format, type, const void *data);

void GetCompressedTexImage2D(enum target, int, level, sizei, sizei, format, type, const void *data);

void GetCompressedTexImage3D(enum target, int, level, sizei, sizei, sizei, format, type, const void *data);

void GetTexImageBuffer(enum target, int, level, sizei, sizei, sizei, format, type, const void *data);

void GetCompressedTexImageBuffer(enum target, int, level, sizei, sizei, sizei, format, type, const void *data);

voidTexImage3D(void *data, int sizei, format, type);
Textures and Samplers (cont.)

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.

void *for attributes of type float, VertexAttribI* for int or uint, or VertexAttribI* for double.

Separate Patches [10.1.15]

* for double.
**Vertex Arrays (cont.)**

void VertexArrayAttribBinding(uint index, uint base);
void VertexArrayAttribIndex(uint index, int size, enum type, size_t stride, const void *pointer);

**Vertex Attribute Divisors**

void VertexBindingDivisor(uint index, uint stride).googleapis.oauth
void VertexArrayBindingDivisor(uint index, uint stride);

**Primitive Restart**

Enable/Disable(IsEnabled)(target)

**Vertex Attributes** [11.1.1]

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

void BindAttribLocation(uint program, int index, const char *name);
void GetActiveAttribs(uint program, uint index, size_t size, GLsizei length, GLsizei *size, GLsizei *length, const void **pointer);

**Transform Feedback**

void GenTransformFeedbacks(uint n, uint *ids);
void DeleteTransformFeedbacks(size_t n, const uint *ids);
bool IsTransformFeedback(uint id);
void BindTransformFeedback(uint target, uint id);
void CreateTransformFeedbacks(size_t n, uint *ids);
void BeginTransformFeedback(uint primitiveMode);

**Rastorization** [13.4, 14]

Enable/Disable(IsEnabled)(target);

**Multisampling** [14.3.1]

Use to anti alias points and lines.

void GetMultisamplef(uint program, uint index, float *val);

**Points** [14.4]

void PointSize(float size);
void PointParameterI_i32f(enum program, T *params, uint program, const char *name);

**Shader Execution** [11.1.1]

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

**Tessellation Prim. Generation** [11.2.2]

void PatchParameterf(uint program, const float *values);

**Vertex Array Queries** [10.5]

void GetVertexAttribiv(uint program, enum pname, uint *params);

**Controlling Viewport** [13.6]

void DepthRangeArrayf(uint first, uint start, uint end, const void **pointer);
void DepthRangeIndexedf(uint index, double *vals);

**Fragment Shaders** [15.2]

void DispatchCompute(uint num_groups_x, uint num_groups_y, uint num_groups_z);
void DispatchComputeIndirect(intptr indirect);

**Compute Shaders** [19]

void DispatchComputeIndirect(intptr indirect);
void Enablei(enum func, int ref, uint mask);
void Enable{func}Separate(enum func, enum mode);
void Blend{func}Separate(enum modeRGB, enum modeAlpha);
void ClearBufferi(enum buffer, int r, int g, int b, int a);

void ClearColori(int r, int g, int b, int a);
void ClearDepthi(float depth);
void ClearDepthstencili(int depth, int stencil);
void Clear{func}Maski(float src, int ref, uint mask);
void Clear{func}MaskSeparate(int ref, uint mask);
void ClearStencilf(float stencil, uint mask);
void Clear{func}Stencili(int src, int ref, int x, int y, uint z, uint w);
void Clear{func}StenciliF(float src, int ref, int x, int y, uint z, uint w);
void Clear{func}StenciliD(int src, int ref, int x, int y, uint z, uint w);

// Fine Control of Buffer Updates
void ColorMaski(boolean b, boolean g, boolean r, boolean a);
void DepthMaski(boolean mask);
void StencilMaski(boolean mask);
void StencilSeparatei(boolean src, boolean dst, boolean keep, boolean replace, boolean incr, boolean decr, boolean wrap, boolean clamped);

// Depth Buffer Test
void DisableDepthTest();
void EnableDepthTest();
void Disable{func}SeparateDepthTest();
void Enable{func}SeparateDepthTest();

void ClearColorAttachmenti(int color, int ref, int x, int y, uint z, uint w);
void ClearDepthAttachmenti(float depth, int ref, int x, int y, uint z, uint w);
void ClearStencilAttachmenti(float stencil, int ref, int x, int y, uint z, uint w);
void Clear{func}MaskAttachmenti(float src, int ref, int x, int y, uint z, uint w);

// Stencil Test
void Enablei(int target, uint mask);
void Disablei(int target, uint mask);
void IsEnabledi(int target, uint mask);

// Multisample FragmentOps.

// Whole Framebuffer

// Reading and Copying Pixels

// Debug Output

// Debug Groups

// State and State Requests

// String Queries

// Hints

// Dithering

// Logical Operation

// Invalidated Framebuffers

// Invalidated Subframebuffers

// State menu

(Continued on next page)
OpenGL Compute Programming Model and Compute Memory Hierarchy

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

```cpp
void barrier();
```

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

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

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

```cpp
in vec3 gl_GlobalInvocationID;
in vec3 gl_WorkGroupID;
const vec3 gl_WorkGroupSize;
```

Use the `심_` function to synchronize invocations in a work group:

```cpp
gl_NumWorkGroups = (4,2,8);
```

OpenGL Texture Views and Texture Object State

Texture state set with TextureView():

```cpp
enum internalformat // base internal format
enum target // texture target
```

Texture Views and Texture Object State

```cpp
void GetInternalformativ(target, pname, internalformat, index, param);
void GetInternalformativ64(target, pname, internalformat, index, int64 *, int64 *);
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

New Texture Object

```cpp
newtextureobject mutable
```

Sampler Object

```cpp
samplerobject mutable
```

Sampler Parameters (mutable)

```cpp
samplerparameters mutable
```

Texture Parameters (mutable)

```cpp
textureparameters mutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture Parameters (immutable)

```cpp
textureparameters immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

TransformFeedback Queries

```cpp
void GetTransformFeedbackiv(uint xfb, enum pname, int *param);
void GetTransformFeedbackiv64(uint xfb, enum pname, uint index, int64 *param);
void TransformFeedbackBufferBinding(view, enum target);
```

Textures (cont.)

```cpp
ubyte
```

OpenGL Texture Views and Texture Object State

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture parameters (mutable)

```cpp
textureparameters mutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```

Texture View Parameters (mutable)

```cpp
textureviewparams mutable
```

Texture View Parameters (immutable)

```cpp
textureviewparams immutable
```
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.
- Texture binding
- 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.

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.
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
3. . . . shared
4. ++ -- prefix increment and decrement unary
5. * / % relational
6. + - additive
7. < <= > >= bit-wise shift
8. == != relational
9. & | ^ logical
10. bit-wise exclusive or
11. bit-wise inclusive or
12. logical and
13. logical or
14. assignment arithmetic assignments
15. = += -= %= <<= >>= &= |= sequence

Types [4.1]

Floating-Point Opaque Types

void no function return value
bool Boolean
int unsigned integer
float single-precision floating-point scalar
double double-precision floating scalar
vec2 vec3 vec4 double-precision floating vector
dvec2 dvec3 dvec4 double-precision floating vector
vec2vec3vec4vector floating point vector
dvec2 dvec3 dvec4 double-precision floating vector

Signed Integer Opaque Types

sampler1D sampler2D sampler3D 1D, 2D, or 3D integer texture
samplerCube sampler1DArray sampler2DArray 1D, 2D, or 3D array texture
samplerBuffer samplerCubeArray sampler1DArray, sampler2DArray, sampler3DArray 1D, 2D, or 3D array texture

Signed Integer Opaque Types (cont'd)

isampler1D isampler2D isampler3D 1D, 2D, or 3D integer texture
isamplerCube isampler1DArray isampler2DArray 1D, 2D, or 3D array texture

Implicit Conversions

int long int short int
float double float single-precision floating-point scalar
vec2 dvec4 vec4 float scalar
uint atom
vec2 dvec4 vec4 atomic

Qualifier Freeform Preprocessor Macros [3.3]

Preprocessor Directives

# if defined endif if define undef

Preprocessor Operators

Inverson 450
Inverson 450 profile

Extension behavior
Extension all - behavior

Required when using version 4.50.

Types  [4.1]

Predefined Macros

_LINE__ FILE__
_VERSION__

GL core profile
GL es profile
GL compatibility profile

Operators & Expressions [5.1]

Qualifiers

Storage Qualifiers [4.3]

Declarations may have one storage qualifier.

Const read-only variable

In linkage into shader from previous stage

Out out of shader to next stage

Uniform linkage into a shader, OpenGL, and the application

Access accessible by shaders and OpenGL, API

Shared accessible by all shaders stored 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 { // allowed restatement qualifier: mat4 ModelViewMatrix; uniform mat3 NormalMatrix; }

Layout Qualifiers [4.4]

The following table summarizes the use of layout qualifiers applied to non-opaque types and the kinds of declarations they may be applied to.

Op = Opaque types only, FC = gl_FragCoord only, FD = gl_FragDepth only.
In the OpenGL Shading Language 4.50 Reference Card, the section on Built-In Variables [7.1] discusses the variables that are automatically available to all shaders. These variables are designed to provide information about the current program execution state, such as the program ID, the input and output variables, and the state of the shader execution environment. The table lists a variety of these built-in variables, such as `gl_VertexID`, `gl_TessLevelOuter`, and `gl_TessLevelInner`, among others.

The section on Geometry Language [7.2] introduces the basic concepts of geometry processing, including vertex and tessellation evaluation. It explains how to use built-in variables like `gl_Coord` and `gl_ClipDistance` to access geometric properties, and how to perform operations like vertex interpolation using built-in variables such as `gl_FragCoord` and `gl_FragCoord2D`.

The Precision Qualifiers [4.9] section emphasizes the importance of precision in shading languages to ensure that operations are executed as intended. It points out that for multiple factors they may appear in order, but not initialized, but not initialized when passed in out.

The Order of Qualification [4.11] discusses the importance of the order in which declarations are made. The layout qualifier is the only qualifier that can qualify a previously declared variable, and it ensures that operations are executed in stated order with operator consistency.

The Structure & Array Operations [5.7] section covers the operations on matrices and vectors, including operations on matrices and vectors, and array subscripting syntax.

The Statements and Structure [6.1] section provides a comprehensive overview of the structure of shader programs, including basic control flow statements like if, switch, and while, and control flow statements like for, while, and do.

The Memory Qualifiers [5.2] section explains how to manage memory allocation and deallocation in shaders, including memory qualifiers like `const`, `uniform`, and `varying`.

The Invariant Qualifiers [4.8] section is crucial for understanding how to ensure that variables are used consistently throughout the shader, including how to qualify a previously declared variable.

The Built-In Constants [7.3] section provides a list of predefined constants that are available to all shaders, which can be used to simplify shader code and improve performance.

The Examples of Operations on Matrices and Vectors [5.4.2] section gives a range of examples of operations that can be performed on matrices and vectors, including matrix multiplication, addition, and scalar multiplication.

The Precision Qualifiers [4.9] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Structure & Array Operations [5.7] section provides a comprehensive overview of the operations that can be performed on arrays and matrices, including operations like matrix multiplication, addition, and scalar multiplication.

The Parameter Qualifiers [4.6] section explains how to use qualifiers like `in`, `out`, and `inout` to control the flow of data between the shader and the rest of the program.

The Interpolation Qualifiers [4.5] section discusses the use of interpolation qualifiers like `flat` to control how variables are interpolated across a function.

The Precision Qualifiers [4.7] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.

The Precision Qualifiers [4.10] section is important for understanding how to specify the precision of variables in a shader, including how to use precision qualifiers like `precision highp`, `precision mediump`, and `precision lowp`.
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 in units of radians. Tf=float, vecn.

- `Tf radians(Tf degrees)` degrees to radians
- `Tf degrees(Tf radians)` radians to degrees
- `Tf sign(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 atan2(Tf y, Tf 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 atan2(Tf y, Tf x)` hyperbolic tangent

Common Functions (cont.)

- Returns maximum value:
  - `Tf max(Tf x, Tf y)` Tf max(Tiu x, Tiu y)
  - `Tf max(Tf x, float y)` Tf max(Tiu x, int y)
  - `Tf max(Tf x, double y)` Tf max(Tiu x, uint y)

- Returns minimum value:
  - `Tf min(Tf x, Tf y)` Tf min(Tiu x, Tiu y)
  - `Tf min(Tf x, float y)` Tf min(Tiu x, int y)
  - `Tf min(Tf x, double y)` Tf min(Tiu x, uint y)

Exponential Functions [8.2]

- `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)` log2
- `Tf sqrt(Tf x)` square root
- `Tf inversesqrt(Tf x)` inverse square root

Common Functions [8.3]

- `Tf ui abs(Tf x)` Ti abs(Ti x)
- `returns -1.0, 0.0, or 1.0:`
  - `Tf sign(Tf x)` Ti sign(Ti x)
- `returns nearest integer < x:`
  - `Tf floor(Tf x)` Tf floor(Tf x)
- `returns nearest integer with absolute value <= x of:
  - `Tf trunc(Tf x)`
  - `returns integer, implementation-dependent rounding mode:
    - `Tf round(Tf x)`
  - `returns nearest integer, 0.5 rounds to nearest even integer:
    - `Tf roundEven(Tf x)`
  - `returns nearest integer >= x:
    - `Tf ceil(Tf x)`
  - `returns x - floor(x):`
    - `Tf fract(Tf x)`
- `returns modulus:
  - `Tf mod(Tf x, Tf y)` Ti mod(Ti x, y)
  - `returns separate integer and fractional parts:
    - `Tf mod(Tf x, Tfi y)`

Floating-Point Pack/Unpack [8.4]

These do not operate component-wise. Converts each component of x into 8- or 16-bit ints, packs results into the returned 32-bit unsigned integer:

- `uint packNorm2x16(vec2 x)`
- `uint packNorm4x8(vec4 x)`
- `uint packNorm2x16(vec2 x)`
- `uint packNorm4x8(vec4 x)`

Unpacks 32-bit into two 16-bit units, four 8-bit units, or signed ints. Then converts each component to a normalized float to generate a 2- or 4-component vector:

- `vec2 unpackNorm2x16(uint y)`
- `vec4 unpackNorm4x8(uint y)`
- `vec4 unpackNorm4x8(uint y)`

Integer Functions (cont.)

- Returns the bit that is least-significant bit of x:
  - `vec2 bitfieldExtract(vec2 x)`
  - `vec4 bitfieldExtract(vec4 x)`

Texture Lookup Functions [8.9]

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

Atomic Counter Functions [8.10]

Returns the value of an atomic counter.

Atomic Memory Functions [8.11]

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

Image Functions [8.12]

In the image functions below, IMAGE_PARAMS may be one of the following:
- gimage1D image, int P
- gimage2D image, int P
- gimage3D image, int P
- gimage2DRect image, int P
- gimageCube image, int P
- gimage2Dcubemap image, int P
- gimage1DArray image, int P
- gimage2DArray image, int P
- gimage2DcubemapArray image, int P
- gimage2DMS image, int P
- gimage2DMSArray image, int P

- Returns the dimensions of the images or images:
  - `int imageSize(IMAGE_PARAMS)`
  - `vec3 imageSize(IMAGE_PARAMS)`

- Stores data into the texture at the coordinate P from the image unit image:
  - `void imageStore(IMAGE_PARAMS, vec4 data)`

(Continued on next page)
Texture Functions [8.9]

Texture Size 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.

Texture query functions return the number of mipmaps accessible in the texture associated with sampler.

Textel 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 and the array layer comes from Fw. For non-shadow forms, the array layer comes from the last component of P.

Projective texture lookup with offset added before texture lookup.

Texture gather as in textureGather, and with explicit gradient as in textureGrad.

Texture Gather Instructions [8.9.3]

functions

Texture gather as in textureGather by offset as described in textureOffset except minimum and maximum offset values are given by (MIN, MAX, PROGRAM_TEXTURE_TEXTURE_OFFSET).

Texture gather as in textureGather by offset except offsets determined location of the four texels to sample.
## 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.

| A | ActiveShaderProgram | CreateTransformFeedbacks | 5 |
| B | BeginConditionalRender | BindTextureArray | 4 |
| C | Callback | CheckNamedFramebufferStatus | 4 |
| D | DebugMessage | DeleteVertexArrayAttrib | 5 |
| E | Enable | EndConditionalRender | 5 |
| F | Feedback | EndQueryIndexed | 1 |
| G | GenBuffers | EndTransformFeedback | 5 |
| H | Hint | GetActiveUniform | 5 |
| I | ImageFunctions | GetActiveAttrib | 5 |
| J | InterpolationFunctions | GetActiveSubroutine | 5 |
| K | Invariant | GetActiveSubroutineIndex | 5 |
| L | LayoutQualifiers | GetActiveUniformSubroutine | 5 |
| M | Macros | GetActiveUniformSubroutineIndex | 5 |
| N | Named | GetActiveUniforms | 5 |
| O | ObjectPtrLabel | GetActiveUniformsIndex | 5 |
| P | PackUnPackFunctions | GetActiveUniformsLength | 5 |
| Q | Qualifiers | GetActiveUniformsRange | 5 |
| R | Rasterization | GetAttachedShaders | 5 |
| S | SampleCoverage | GetAttribLocation | 5 |
| T | TesselationDiagram | GetAttribLocationIndexed | 5 |
| U | UniformQualifiers | GetAttribLocationIndex | 5 |
| V | ValidateProgramPipeline | GetAttribLocationIndexed | 5 |
| W | WaitSync | GetAttribLocationIndexed | 5 |

---

See www.opengl.org to learn more about OpenGL.

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

OpenGL is a registered trademark of Silicon Graphics International, used under license by Khronos Group.

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.khrons.org to learn more about the Khronos Group.

See www.opengl.org to learn more about OpenGL.