Name

    KHR_vulkan_glsl

Name Strings

    GL_KHR_vulkan_glsl

Contact

    John Kessenich (johnkessenich 'at' google.com), Google

Contributors

    Jeff Bolz, NVIDIA
    Kerch Holt, NVIDIA
    Kenneth Benzie, Codeplay
    Neil Henning, Codeplay
    Neil Hickey, ARM
    Daniel Koch, NVIDIA
    Timothy Lottes, Epic Games
    David Neto, Google

Notice

Status

    Approved by Vulkan working group 03-Dec-2015.
    Ratified by the Khronos Board of Promoters 15-Jan-2016.

Version

    Last Modified Date: 21-May-2017
    Revision: 40

Number

    TBD.

Dependencies

    This extension can be applied to OpenGL GLSL versions 1.40
    (#version 140) and higher.

    This extension can be applied to OpenGL ES ESSL versions 3.10

(#version 310) and higher.

All these versions map GLSL/ESSL semantics to the same SPIR-V 1.0 semantics (approximating the most recent versions of GLSL/ESSL).

Overview

This is version 100 of the GL_KHR_vulkan_glsl extension.

This extension modifies GLSL to be used as a high-level language for the Vulkan API. GLSL is compiled down to SPIR-V, which the Vulkan API consumes.

The following features are removed:
* default uniforms (uniform variables not inside a uniform block), except for opaque types
* atomic-counters (those based on atomic_uint)
* subroutines
* shared and packed block layouts
* the already deprecated texturing functions (e.g., texture2D())
* compatibility-mode-only features
* gl_DepthRangeParameters and gl_NumSamples
* gl_VertexID and gl_InstanceID

The following features are added:
* push-constant buffers
* shader-combining of separate textures and samplers
* descriptor sets
* specialization constants
* gl_VertexIndex and gl_InstanceIndex
* subpass inputs
* 'offset' and 'align' layout qualifiers for uniform/buffer blocks for versions that did not support them

The following features are changed:
* precision qualifiers (mediump and lowp) will be respected for all versions, not dropped for desktop versions (default precision for desktop versions is highp for all types)
* gl_FragColor will no longer indicate an implicit broadcast
* arrays of opaque uniforms take only one binding number for the entire object, not one per array element
* the default origin is origin_upper_left instead of origin_lower_left

Each of these is discussed in more detail below.

Enabling These Features

This extension is not enabled with a #extension as other extensions are. It is also not enabled through use of a profile or #version. The intended level of GLSL/ESSL features, independent from Vulkan-specific usage, comes from the traditional use of #version, profile, and #extension.

Instead, use of this extension is an effect of using a GLSL front-end in a mode that has it generate SPIR-V for Vulkan. Such tool use is outside the scope of using the Vulkan API and outside the definition of GLSL and this extension. See the documentation of the compiler to see how to request generation of SPIR-V for Vulkan.

When a front-end is used to accept this extension, it must error check and reject shaders not adhering to this specification, and accept those that do. Implementation-dependent maximums and capabilities are supplied to, or part of, the front-end, so it can do error checking against them.

A shader can query the level of Vulkan support available, using the predefined

```
    #define VULKAN 100
```

This allows shader code to say, for example,

```
    #ifdef VULKAN
        layout(set = 1, binding = 0) uniform sampler s;
        layout(set = 1, binding = 1) uniform texture2D t;
        #if VULKAN > 100
            ...
        #endif
    #else
        layout(binding = 0) uniform sampler2D ts;
    #endif
```

Push Constants

Push constants reside in a uniform block declared using the new layout-qualifier-id "push_constant" applied to a uniform-block declaration. The API writes a set of constants to a push-constant buffer, and the shader reads them from a push_constant block:

```
    layout(push_constant) uniform BlockName {
        int member1;
        float member2;
        ...
    } InstanceName; // optional instance name

    ... = InstanceName.member2; // read a push constant
```

The memory accounting used for the push_constant uniform block is different
than for other uniform blocks: There is a separate small pool of memory
it must fit within. By default, a push_constant buffer follows the std430
packing rules.

Combining separate samplers and textures

A sampler, declared with the keyword 'sampler', contains just filtering
information, containing neither a texture nor an image:

```
    uniform sampler s;    // a handle to filtering information
```

A texture, declared with keywords like 'texture2D', contains just image
information, not filtering information:

```
    uniform texture2D t;  // a handle to a texture (an image in SPIR-V)
```

Constructors can then be used to combine a sampler and a texture at the
point of making a texture lookup call:

```
    texture2D(sampler2D(t, s), ...);
```

Note, layout() information is omitted above for clarity of this feature.

Descriptor Sets

Bound objects can further declare which Vulkan descriptor set they belong
to, using 'set':

```
    layout(set = N, ...) ... // declared object belongs to descriptor set N
```

For example, two combined texture/sampler objects can be declared in two
different descriptor sets as follows

```
    layout(set = 0, binding = 0) uniform sampler2D ts3;
    layout(set = 1, binding = 0) uniform sampler2D ts4;
```

See the API documentation for more detail on the operation model of
descriptor sets.

Specialization Constants

SPIR-V specialization constants, which can be set later by the client API,
can be declared using "layout(constant_id=...)". For example, to make a

specialization constant with a default value of 12:

```
    layout(constant_id = 17) const int arraySize = 12;
```

Above, "17" is the ID by which the API or other tools can later refer to this specific specialization constant. The API or an intermediate tool can then change its value to another constant integer before it is fully lowered to executable code. If it is never changed before final lowering, it will retain the value of 12.

Specialization constants have const semantics, except they don't fold. Hence, an array can be declared with 'arraySize' from above:

```
    vec4 data[arraySize];  // legal, even though arraySize might change
```

Specialization constants can be in expressions:

```
    vec4 data2[arraySize + 2];
```

This will make data2 be sized by 2 more than whatever constant value 'arraySize' has when it is time to lower the shader to executable code.

An expression formed with specialization constants also behaves in the shader like a specialization constant, not a like a constant.

```
    arraySize + 2       // a specialization constant (with no constant_id)
```

Such expressions can be used in the same places as a constant.

The constant_id can only be applied to a scalar *int*, a scalar *float* or a scalar *bool*.

Only basic operators and constructors can be applied to a specialization constant and still result in a specialization constant:

```
    layout(constant_id = 17) const int arraySize = 12;
    sin(float(arraySize));    // result is not a specialization constant
```

While SPIR-V specialization constants are only for scalars, a vector can be made by operations on scalars:

```
    layout(constant_id = 18) const int scX = 1;
    layout(constant_id = 19) const int scZ = 1;
    const vec3 scVec = vec3(scX, 1, scZ);  // partially specialized vector
```

A built-in variable can have a 'constant_id' attached to it:

```
    layout(constant_id = 18) gl_MaxImageUnits;
```

This makes it behave as a specialization constant. It is not a full redeclaration; all other characteristics are left intact from the original built-in declaration.

The built-in vector gl_WorkGroupSize can be specialized using special layout local_size_{xyz}_id's applied to the "in" qualifier. For example:

```
    layout(local_size_x_id = 18, local_size_z_id = 19) in;
```

This leaves gl_WorkGroupSize.y as a non-specialization constant, with gl_WorkGroupSize being a partially specialized vector. Its x and z components can be later specialized using the ID's 18 and 19.

gl_VertexIndex and gl_InstanceIndex

Adds two new built-in variables, gl_VertexIndex and gl_InstanceIndex to replace the existing built-in variables gl_VertexID and gl_InstanceID.

In the situations where the indexing is relative to some base offset, these built-in variables are defined, for Vulkan, to take on values as follows:

```
    gl_VertexIndex             base, base+1, base+2, ...
    gl_InstanceIndex           base, base+1, base+2, ...
```

Where it depends on the situation what the base actually is.

Subpass Inputs

Within a rendering pass, a subpass can write results to an output target that can then be read by the next subpass as an input subpass. The "Subpass Input" feature regards the ability to read an output target.

Subpasses are read through a new set of types, available only to fragment shaders:

```
     subpassInput
     subpassInputMS
    isubpassInput
    isubpassInputMS
    usubpassInput
    usubpassInputMS
```

Unlike sampler and image objects, subpass inputs are implicitly addressed by the fragment's (x, y, layer) coordinate.

A subpass input is selected by using a new layout qualifier identifier 'input_attachment_index'. For example:

```
    layout(input_attachment_index = i, ...) uniform subpassInput t;
```

An input_attachment_index of i selects the ith entry in the input pass list. (See API specification for more information.)

These objects support reading the subpass input through the following functions:

```
    gvec4 subpassLoad(gsubpassInput   subpass);
    gvec4 subpassLoad(gsubpassInputMS subpass, int sample);
```

gl_FragColor

The fragment-stage built-in gl_FragColor, which implies a broadcast to all outputs, is not present in SPIR-V. Shaders where writing to gl_FragColor is allowed can still write to it, but it only means to write to an output:
 - of the same type as gl_FragColor
 - decorated with location 0
 - not decorated as a built-in variable.
There is no implicit broadcast.

Mapping to SPIR-V

For informational purposes (non-specification), the following is an expected way for an implementation to map GLSL constructs to SPIR-V constructs:

Mapping of storage classes:

```
  uniform sampler2D...;        -> UniformConstant
  uniform blockN { ... } ...;  -> Uniform, with Block decoration
  in / out variable            -> Input/Output, possibly with block (below)
  in / out block...            -> Input/Output, with Block decoration
  buffer  blockN { ... } ...;  -> Uniform, with BufferBlock decoration, or
                                  StorageBuffer, when requested
  N/A                          -> AtomicCounter
  shared                       -> Workgroup
  <normal global>              -> Private
```

Mapping of input/output blocks or variables is the same for all versions of GLSL or ESSL. To the extent variables or members are available in a version, its location is as follows:

These are mapped to SPIR-V individual variables, with similarly spelled
built-in decorations (except as noted):

  Any stage:

```
      in gl_NumWorkGroups
      in gl_WorkGroupSize
      in gl_WorkGroupID
      in gl_LocalInvocationID
      in gl_GlobalInvocationID
      in gl_LocalInvocationIndex

      in gl_VertexIndex
      in gl_InstanceIndex
      in gl_InvocationID
      in gl_PatchVerticesIn      (PatchVertices)
      in gl_PrimitiveIDIn        (PrimitiveID)
      in/out gl_PrimitiveID      (in/out based only on storage qualifier)
      in gl_TessCoord

      in/out gl_Layer
      in/out gl_ViewportIndex

      patch in/out gl_TessLevelOuter  (uses Patch decoration)
      patch in/out gl_TessLevelInner  (uses Patch decoration)
```

  Fragment stage only:

```
      in gl_FragCoord
      in gl_FrontFacing
      in gl_ClipDistance
      in gl_CullDistance
      in gl_PointCoord
      in gl_SampleID
      in gl_SamplePosition
      in gl_HelperInvocation
      out gl_FragDepth
      in gl_SampleMaskIn        (SampleMask)
      out gl_SampleMask         (in/out based only on storage qualifier)
```

These are mapped to SPIR-V blocks, as implied by the pseudo code, with
the members decorated with similarly spelled built-in decorations:

  Non-fragment stage:

```
      in/out gl_PerVertex {   // some subset of these members will be used
          gl_Position
```

```
            gl_PointSize
            gl_ClipDistance
            gl_CullDistance
        }                       // name of block is for debug only
```

  There is at most one input and one output block per stage in SPIR-V.
  The subset and order of members will match between stages sharing an
  interface.

Mapping of precision qualifiers:

  lowp   -> RelaxedPrecision, on storage variable and operation
  mediump -> RelaxedPrecision, on storage variable and operation
  highp   -> 32-bit, same as int or float

  portability tool/mode -> OpQuantizeToF16

Mapping of precise:

  precise -> NoContraction

Mapping of images

  subpassInput -> OpTypeImage with 'Dim' of SubpassData
  subpassLoad() -> OpImageRead
  imageLoad()   -> OpImageRead
  imageStore() -> OpImageWrite
  texelFetch() -> OpImageFetch

  imageAtomicXXX(params, data) -> %ptr = OpImageTexelPointer params
                                          OpAtomicXXX %ptr, data

  XXXQueryXXX(combined) -> %image = OpImage combined
                                    OpXXXQueryXXX %image

Mapping of layouts

  std140/std430 -> explicit offsets/strides on struct
  shared/packed -> not allowed
  <default>   -> not shared, but std140 or std430

  max_vertices   -> OutputVertices

Mapping of other instructions

  %   -> OpUMod/OpSMod
  mod() -> OpFMod
  N/A   -> OpSRem/OpFRem

```
  atomicExchange()      -> OpAtomicExchange
  imageAtomicExchange() -> OpAtomicExchange
  atomicCompSwap()      -> OpAtomicCompareExchange
  imageAtomicCompSwap() -> OpAtomicCompareExchange
  N/A                   -> OpAtomicCompareExchangeWeak
```

Changes to Chapter 1 of the OpenGL Shading Language Specification

Change the last paragraph of "1.3 Overview": "The OpenGL Graphics System Specification will specify the OpenGL entry points used to manipulate and communicate with GLSL programs and GLSL shaders."

Add a paragraph: "The Vulkan API will specify the Vulkan entry points used to manipulate SPIR-V shaders. Independent offline tool chains will compile GLSL down to the SPIR-V intermediate language. Vulkan use is not enabled with a #extension, #version, or a profile. Instead, use of GLSL for Vulkan is determined by offline tool-chain use. See the documentation of such tools to see how to request generation of SPIR-V for Vulkan."

"GLSL -> SPIR-V compilers must be directed as to what SPIR-V *Capabilities* are legal at run-time and give errors for GLSL feature use outside those capabilities. This is also true for implementation-dependent limits that can be error checked by the front-end against constants present in the GLSL source: the front-end can be informed of such limits, and report errors when they are exceeded."

Changes to Chapter 2 of the OpenGL Shading Language Specification

Change the name from

"2 Overview of OpenGL Shading"

to

"2 Overview of OpenGL and Vulkan Shading"

Remove the word "OpenGL" from three introductory paragraphs.

Changes to Chapter 3 of the OpenGL Shading Language Specification

Add a new paragraph at the end of section "3.3 Preprocessor": "When shaders are compiled for Vulkan, the following predefined macro is available:

```
    #define VULKAN 100
```

Add the following keywords to section 3.6 Keywords:

```
    texture1D        texture2D        texture3D
    textureCube      texture2DRect    texture1DArray
    texture2DArray   textureBuffer    texture2DMS
    texture2DMSArray textureCubeArray

    itexture1D        itexture2D       itexture3D
    itextureCube      itexture2DRect   itexture1DArray
    itexture2DArray   itextureBuffer
    itexture2DMS      itexture2DMSArray
    itextureCubeArray

    utexture1D        utexture2D        utexture3D
    utextureCube      utexture2DRect    utexture1DArray
    utexture2DArray   utextureBuffer    utexture2DMS
    utexture2DMSArray utextureCubeArray

    sampler    samplerShadow

    subpassInput      isubpassInput     usubpassInput
    subpassInputMS    isubpassInputMS   usubpassInputMS
```

Move the following keywords in section 3.6 Keywords to the reserved section:

```
    atomic_uint
    subroutine
```

Changes to Chapter 4 of the OpenGL Shading Language Specification

Add into the tables in section 4.1 Basic Types, interleaved with the existing types, using the existing descriptions (when not supplied below):

Floating-Point Opaque Types

```
        texture1D
        texture2D
        texture3D
        textureCube
        texture2DRect
        texture1DArray
        texture2DArray
        textureBuffer
        texture2DMS
        texture2DMSArray
        textureCubeArray
        subpassInput            | a handle for accessing a floating-point
```

```
                          GL_KHR_vulkan_glsl.txt
                               | subpass input
        subpassInputMS         | a handle for accessing a multi-sampled
                               | floating-point subpass input

    Signed Integer Opaque Types

        itexture1D
        itexture2D
        itexture3D
        itextureCube
        itexture2DRect
        itexture1DArray
        itexture2DArray
        itextureBuffer
        itexture2DMS
        itexture2DMSArray
        itextureCubeArray
        isubpassInput     | a handle for accessing an integer subpass input
        isubpassInputMS   | a handle for accessing a multi-sampled integer
                          | subpass input

    Unsigned Integer Opaque Types

        utexture1D
        utexture2D
        utexture3D
        utextureCube
        utexture2DRect
        utexture1DArray
        utexture2DArray
        utextureBuffer
        utexture2DMS
        utexture2DMSArray
        utextureCubeArray
        usubpassInput     | a handle for accessing an unsigned integer
                          | subpass input
        usubpassInputMS   | a handle for accessing a multi-sampled unsigned
                          | integer subpass input

Remove the entry from the table in section 4.1 Basic Types:

    atomic_uint

Add a new category in this section

  "Sampler Opaque Types

      sampler       |    a handle for accessing state describing how to
```

```
                |    sample a texture (without comparison)"
        ------------------------------------------------------------------
        samplerShadow |    a handle for accessing state describing how to
                |    sample a depth texture with comparison"
```

Remove "structure member selection" from 4.1.7 and instead add a sentence "Opaque types cannot be declared or nested in a structure (struct)."

Modify subsection 4.1.3 Integers, for desktop versions of GLSL, to say:

"Highp unsigned integers have exactly 32 bits of precision. Highp signed integers use 32 bits, including a sign bit, in two's complement form. Mediump and lowp integers are as defined by the RelaxedPrecision decoration in SPIR-V."

Add a subsection to 4.1.7 Opaque Types:

"4.1.7.x Texture, *sampler*, and *samplerShadow* Types

"Texture (e.g., *texture2D*), *sampler*, and *samplerShadow* types are opaque types, declared and behaving as described above for opaque types. When aggregated into arrays within a shader, these types can only be indexed with a dynamically uniform expression, or texture lookup will result in undefined values. Texture variables are handles to one-, two-, and three-dimensional textures, cube maps, etc., as enumerated in the basic types tables. There are distinct texture types for each texture target, and for each of float, integer, and unsigned integer data types. Textures can be combined with a variable of type *sampler* or *samplerShadow* to create a sampler type (e.g., sampler2D, or sampler2DShadow). This is done with a constructor, e.g., sampler2D(texture2D, sampler) or sampler2DShadow(texture2D, samplerShadow), and is described in more detail in section 5.4 "Constructors"."

"4.1.7.x Subpass Inputs

"Subpass input types (e.g., subpassInput) are opaque types, declared and behaving as described above for opaque types. When aggregated into arrays within a shader, they can only be indexed with a dynamically uniform integral expression, otherwise results are undefined.

"Subpass input types are handles to two-dimensional single sampled or multi-sampled images, with distinct types for each of float, integer, and unsigned integer data types.

"Subpass input types are only available in fragment shaders. It is a compile-time error to use them in any other stage."

Remove the section 4.1.7.3 Atomic Counters

Change section 4.3.3 Constant Expressions:

  Add a new very first sentence to this section:

    "SPIR-V specialization constants are expressed in GLSL as const, with
    a layout qualifier identifier of constant_id, as described in section
    4.4.x Specialization-Constant Qualifier."

  Add to this sentence:

    "A constant expression is one of...
      * a variable declared with the const qualifier and an initializer,
        where the initializer is a constant expression"

  To make it say:

    "A constant expression is one of...
      * a variable declared with the const qualifier and an initializer,
        where the initializer is a constant expression; this includes both
        const declared with a specialization-constant layout qualifier,
        e.g., 'layout(constant_id = ...)' and those declared without a
        specialization-constant layout qualifier"

  Add to "including getting an element of a constant array," that

    "an array access with a specialization constant as an index does
    not result in a constant expression"

  Add to this sentence:

    "A constant expression is one of...
      * the value returned by a built-in function..."

  To make it say:

    "A constant expression is one of...
      * for non-specialization-constants only: the value returned by a
        built-in function... (when any function is called with an argument
        that is a specialization constant, the result is not a constant
        expression)"

  Rewrite the last half of the last paragraph to be its own paragraph
  saying:

    "Non-specialization constant expressions may be evaluated by the

compiler's host platform, and are therefore not required ...
[rest of paragraph stays the same]"

Add a paragraph

"Specialization constant expressions are never evaluated by the front-end, but instead retain the operations needed to evaluate them later on the host."

Add to the table in section 4.4 Layout Qualifiers:

	Individual Variable	Block	Allowed Interface
constant_id =	scalar only		const
push_constant		X	uniform
set =	opaque only	X	uniform
input_attachment_index	subpass types only		uniform

(The other columns remain blank.)

Also add to this table:

	Qualifier Only	Allowed Interface
local_size_x_id =	X	in
local_size_y_id =	X	in
local_size_z_id =	X	in

(The other columns remain blank.)

Expand this sentence in section 4.4.1 Input Layout Qualifiers:

"Where integral-constant-expression is defined in section 4.3.3 Constant Expressions as 'integral constant expression'"

To include the following:

", with it being a compile-time error for integer-constant-expression to be a specialization constant: The constant used to set a layout identifier X in layout(layout-qualifier-name = X) must evaluate to a front-end constant containing no specialization constants."

Change the rules about locations and inputs for doubles, by removing

"If a vertex shader input is any scalar or vector type, it will consume

  a single location. If a non-vertex shader input is a scalar or vector
  type other than dvec3 or dvec4..."

Replacing the above with

  "If an input is a scalar or vector type other than dvec3 or dvec4..."

(Making all stages have the same rule that dvec3 takes two locations...)

At the end of the paragraphs describing the *location* rules, add this
paragraph:

  "When generating SPIR-V, all *in* and *out* qualified user-declared
  (non built-in) variables and blocks (or all their members) must have a
  shader-specified *location*. Otherwise, a compile-time error is
  generated."

[Note that an earlier existing rule just above this says "If a block has
no block-level *location* layout qualifier, it is required that either all
or none of its members have a *location* layout qualifier, or a compile-
time error results."]

Change section 4.4.1.3 "Fragment Shader Inputs" from

  "By default, gl_FragCoord assumes a lower-left origin for window
  coordinates ... For example, the (x, y) location (0.5, 0.5) is
  returned for the lowerleft-most pixel in a window. The origin can be
  changed by redeclaring gl_FragCoord with the
  origin_upper_left identifier."

To

  "The gl_FragCoord built-in variable assumes an upper-left origin for
  window coordinates ... For example, the (x, y) location (0.5, 0.5) is
  returned for the upper-left-most pixel in a window. The origin can be
  explicitly set by redeclaring gl_FragCoord with the origin_upper_left
  identifier. It is a compile-time error to change it to
  origin_lower_left."

Add to the end of section 4.4.3 Uniform Variable Layout Qualifiers:

  "The /push_constant/ identifier is used to declare an entire block, and
  represents a set of "push constants", as defined by the API. It is a
  compile-time error to apply this to anything other than a uniform block
  declaration. The values in the block will be initialized through the
  API, as per the Vulkan API specification. A block declared with
  layout(push_constant) may optionally include an /instance-name/.
  There can be only one push_constant

  block per stage, or a compile-time or link-time error will result. A
  push-constant array can only be indexed with dynamically uniform indexes.
  Uniform blocks declared with push_constant use different resources
  than those without; and are accounted for separately. See the API
  specification for more detail."

After the paragraphs about binding ("The binding identifier..."), add

  "The /set/ identifier specifies the descriptor set this object belongs to.
  It is a compile-time error to apply /set/ to a standalone qualifier or to
  a member of a block. It is a compile-time error to apply /set/ to a block
  qualified as a push_constant. By default, any non-push_constant uniform
  or shader storage block declared without a /set/ identifier is assigned to
  descriptor set 0. Similarly, any sampler, texture, or subpass input type
  declared as a uniform, but without a /set/ identifier is also assigned
  to descriptor set 0.

  "If applied to an object declared as an array, all elements of the array
  belong to the specified /set/.

  "It is a compile-time error for either the /set/ or /binding/ value
  to exceed a front-end-configuration supplied maximum value."

Remove mention of subroutine throughout section 4.4 Layout Qualifiers,
including removal of section 4.4.4 Subroutine Function Layout Qualifiers.

Change section 4.4.6 Opaque-Uniform Layout Qualifiers:

  Change

  "If the binding identifier is used with an array, the first element of
  the array takes the specified unit and each subsequent element takes the
  next consecutive unit."

  To

  "If the binding identifier is used with an array, the entire array
  takes just the provided binding number. The next consecutive binding
  number is available for a different object."

Remove section 4.4.6.1 Atomic Counter Layout Qualifiers

Add a new subsection at the end of section 4.4:

  "4.4.x Specialization-Constant Qualifier

  "Specialization constants are declared using "layout(constant_id=...)".
  For example:

```
  layout(constant_id = 17) const int arraySize = 12;
```

"The above makes a specialization constant with a default value of 12.
17 is the ID by which the API or other tools can later refer to
this specific specialization constant. If it is never changed before
final lowering, it will retain the value of 12. It is a compile-time
error to use the constant_id qualifier on anything but a scalar bool,
int, uint, float, or double.

"Built-in constants can be declared to be specialization constants.
For example,

```
  layout(constant_id = 31) gl_MaxClipDistances;  // add specialization id
```

"The declaration uses just the name of the previously declared built-in
variable, with a constant_id layout declaration. It is a compile-time
error to do this after the constant has been used: Constants are strictly
either non-specialization constants or specialization constants, not
both.

"The built-in constant vector gl_WorkGroupSize can be specialized using
the local_size_{xyz}_id qualifiers, to individually give the components
an id. For example:

```
    layout(local_size_x_id = 18, local_size_z_id = 19) in;
```

"This leaves gl_WorkGroupSize.y as a non-specialization constant, with
gl_WorkGroupSize being a partially specialized vector. Its x and z
components can be later specialized using the ids 18 and 19. These ids
are declared independently from declaring the work-group size:

```
  layout(local_size_x = 32, local_size_y = 32) in;   // size is (32,32,1)
  layout(local_size_x_id = 18) in;                   // constant_id for x
  layout(local_size_z_id = 19) in;                   // constant_id for z
```

"Existing rules for declaring local_size_x, local_size_y, and
local_size_z are not changed by this extension. For the local-size ids,
it is a compile-time error to provide different id values for the same
local-size id, or to provide them after any use. Otherwise, order,
placement, number of statements, and replication do not cause errors.

"Two arrays sized with specialization constants are the same type only if
sized with the same symbol, involving no operations.

```
  layout(constant_id = 51) const int aSize = 20;
  const int pad = 2;
  const int total = aSize + pad; // specialization constant
```

```
    int a[total], b[total];        // a and b have the same type
    int c[22];                     // different type than a or b
    int d[aSize + pad];            // different type than a, b, or c
    int e[aSize + 2];              // different type than a, b, c, or d
```

"Types containing arrays sized with a specialization constant cannot be compared, assigned as aggregates, declared with an initializer, or used as an initializer. They can, however, be passed as arguments to functions having formal parameters of the same type.

"Arrays inside a block may be sized with a specialization constant, but the block will have a static layout. Changing the specialized size will not re-layout the block. In the absence of explicit offsets, the layout will be based on the default size of the array."

Add a new subsection at the end of section 4.4:

"4.4.y Subpass Qualifier

"Subpasses are declared with the basic 'subpassInput' types. However, they must have the layout qualifier "input_attachment_index" declared with them, or a compile-time error results. For example:

```
  layout(input_attachment_index = 2, ...) uniform subpassInput t;
```

This selects which subpass input is being read from. The value assigned to 'input_attachment_index', say i (input_attachment_index = i), selects that entry (ith entry) in the input list for the pass. See the API documentation for more detail about passes and the input list.

"If an array of size N is declared, it consume N consecutive input_attachment_index values, starting with the one provided.

"It is a compile-time or link-time error to have different variables declared with the same input_attachment_index. This includes any overlap in the implicit input_attachment_index consumed by array declarations.

"It is a compile-time error if the value assigned to an input_attachment_index is greater than or equal to gl_MaxInputAttachments."

Remove all mention of the 'shared' and 'packed' layout qualifiers.

Change section 4.4.5 Uniform and Shader Storage Block Layout Qualifiers

"The initial state of compilation is as if the following were declared:

```
  layout(std140, column_major) uniform;  // without push_constant
```

```
    layout(std430, column_major) buffer;
```

"However, when push_constant is declared, the default layout of the buffer will be std430. There is no method to globally set this default."

Add to section 4.4.5 Uniform and Shader Storage Block Layout Qualifiers, for versions not having 'offset' and 'align' description language, or replace with the following for versions that do have 'offset' and 'align' description language:

"The 'offset' qualifier can only be used on block members of 'uniform' or 'buffer' blocks. The 'offset' qualifier forces the qualified member to start at or after the specified integral-constant-expression, which will be its byte offset from the beginning of the buffer. It is a compile-time error to have any offset, explicit or assigned, that lies within another member of the block. Two blocks linked together in the same program with the same block name must have the exact same set of members qualified with 'offset' and their integral-constant-expression values must be the same, or a link-time error results. The specified 'offset' must be a multiple of the base alignment of the type of the block member it qualifies, or a compile-time error results.

"The 'align' qualifier can only be used on block members of 'uniform' or 'buffer' blocks. The 'align' qualifier makes the start of each block buffer have a minimum byte alignment. It does not affect the internal layout within each member, which will still follow the std140 or std430 rules. The specified alignment must be greater than 0 and a power of 2, or a compile-time error results.

"The actual alignment of a member will be the greater of the specified 'align' alignment and the standard (e.g., std140) base alignment for the member's type. The actual offset of a member is computed as follows: If 'offset' was declared, start with that offset, otherwise start with the offset immediately following the preceding member (in declaration order). If the resulting offset is not a multiple of the actual alignment, increase it to the first offset that is a multiple of the actual alignment. This results in the actual offset the member will have.

"When 'align' is applied to an array, it affects only the start of the array, not the array's internal stride. Both an 'offset' and an 'align' qualifier can be specified on a declaration.

"The 'align' qualifier, when used on a block, has the same effect as qualifying each member with the same 'align' value as declared on the block, and gets the same compile-time results and errors as if this had been done. As described in general earlier, an individual member can specify its own 'align', which overrides the block-level 'align', but just for that member."

Remove the following preamble from section 4.7, which exists for desktop versions, but not ES versions. Removal:

> "Precision qualifiers are added for code portability with OpenGL ES, not for functionality. They have the same syntax as in OpenGL ES, as described below, but they have no semantic meaning, which includes no effect on the precision used to store or operate on variables.
>
> "If an extension adds in the same semantics and functionality in the OpenGL ES 2.0 specification for precision qualifiers, then the extension is allowed to reuse the keywords below for that purpose.
>
> "For the purposes of determining if an output from one shader stage matches an input of the next stage, the precision qualifier need not match."

Add:

> "For interface matching, uniform variables and uniform and buffer block members must have the same precision qualification. For matching *out* variables or block members to *in* variables and block members, the precision qualification does not have to match.
>
> "Global variables declared in different compilation units linked into the same shader stage must be declared with the same precision qualification."

More generally, all versions will follow OpenGL ES semantic rules for precision qualifiers.

Section 4.7.2 Precision Qualifiers (desktop only)

> Replace the table saying "none" for all precisions with this statement:
>
> > "Mediump and lowp floating-point values have the precision defined by the RelaxedPrecision decoration in SPIR-V."

Section 4.7.4 Default Precision Qualifiers:

> For desktop versions, replace the last three paragraphs that state the default precisions with the following instead:
>
> > "All stages have default precision qualification of highp for all types that accept precision qualifiers."

Changes to Chapter 5 of the OpenGL Shading Language Specification

Add a new subsection at the end of section 5.4 "Constructors":

"5.4.x Sampler Constructors

"Sampler types, like *sampler2D* can be declared with an initializer that is a constructor of the same type, and consuming a texture and a sampler. For example:

```
  layout(...) uniform sampler s;   // handle to filtering information
  layout(...) uniform texture2D t; // handle to a texture
  layout(...) in vec2 tCoord;
  ...
  texture(sampler2D(t, s), tCoord);
```

The result of a sampler constructor cannot be assigned to a variable:

```
  ... sampler2D sConstruct = sampler2D(t, s);  // ERROR
```

Sampler constructors can only be consumed by a function parameter.

Sampler constructors of arrays are illegal:

```
  layout(...) uniform texture2D tArray[6];
  ...
  ... sampler2D[](tArray, s) ...  // ERROR
```

Formally:
 * every sampler type can be used as a constructor
 * the type of the constructor must match the type of the variable being declared
 * the constructor's first argument must be a texture type
 * the constructor's second argument must be a scalar of type *sampler* or *samplerShadow*
 * the dimensionality (1D, 2D, 3D, Cube, Rect, Buffer, MS, and Array) of the texture type must match that of the constructed sampler type (that is, the suffixes of the type of the first argument and the type of the constructor will be spelled the same way)
 * the presence or absence of depth comparison (Shadow) must match between the constructed sampler type and the type of the second argument
 * there is no control flow construct (e.g., "?:") that consumes any sampler type

Change section 5.9 Expressions

Add under "The sequence (,) operator..."

"Texture and sampler types cannot be used with the sequence (,) operator."

Change under "The ternary selection operator (?:)..."

"The second and third expressions can be any type, as long their types match."

To

"The second and third expressions can be any type, as long their types match, except for texture and sampler types, which result in a compile-time error."

Add a section at the end of section 5

"5.x Specialization Constant Operations"

Only some operations discussed in this section may be applied to a specialization constant and still yield a result that is as specialization constant. The operations allowed are listed below. When a specialization constant is operated on with one of these operators and with another constant or specialization constant, the result is implicitly a specialization constant.

- int(), uint(), and bool() constructors for type conversions from any of the following types to any of the following types:
  * int
  * uint
  * bool
- vector versions of the above conversion constructors
- allowed implicit conversions of the above
- swizzles (e.g., foo.yx)
- The following when applied to integer or unsigned integer types:
  * unary negative (-)
  * binary operations (+ , - , * , / , %)
  * shift (<<, >>)
  * bitwise operations (& , | , ^)
- The following when applied to integer or unsigned integer scalar types:
  * comparison (== , != , > , >= , < , <=)
- The following when applied to the Boolean scalar type:
  * not (!)
  * logical operations (&& , || , ^^)
  * comparison (== , !=)
- The ternary operator (? :)

Changes to Chapter 6 of the OpenGL Shading Language Specification

Remove mention of subroutine throughout, including removal of section 6.1.2 Subroutines.

Changes to Chapter 7 of the OpenGL Shading Language Specification

Changes to section 7.1 Built-In Language Variables

Replace gl_VertexID and gl_InstanceID, for non-ES with:

"in int gl_VertexIndex;"
"in int gl_InstanceIndex;"

For ES, add:

"in highp int gl_VertexIndex;"
"in highp int gl_InstanceIndex;"

The following definition for gl_VertexIndex should replace the definition for gl_VertexID:

"The variable gl_VertexIndex is a vertex language input variable that holds an integer index for the vertex, [See issue 7 regarding which name goes with which semantics] relative to a base. While the variable gl_VertexIndex is always present, its value is not always defined. See XXX in the API specification."

The following definition for gl_InstanceIndex should replace the definition for gl_InstanceID:

"The variable gl_InstanceIndex is a vertex language input variable that holds the instance number of the current primitive in an instanced draw call, relative to a base. If the current primitive does not come from an instanced draw call, the value of gl_InstanceIndex is zero." [See issue 7 regarding which name goes with which semantics]

Changes to section 7.3 Built-In Constants

Add

"const int gl_MaxInputAttachments = 1;"

Remove section 7.4 Built-In Uniform State (there is none in Vulkan).

Changes to Chapter 8 of the OpenGL Shading Language Specification

Add the following ES language to desktop versions of the specification:

"The operation of a built-in function can have a different precision qualification than the precision qualification of the resulting value. These two precision qualifications are established as follows.

"The precision qualification of the operation of a built-in function is based on the precision qualification of its input arguments and formal parameters: When a formal parameter specifies a precision qualifier, that is used, otherwise, the precision qualification of the calling argument is used. The highest precision of these will be the precision qualification of the operation of the built-in function. Generally, this is applied across all arguments to a built-in function, with the exceptions being:

- bitfieldExtract and bitfieldInsert ignore the 'offset' and 'bits' arguments.
- interpolateAt* functions only look at the 'interpolant' argument.

"The precision qualification of the result of a built-in function is determined in one of the following ways:

- For the texture sampling, image load, and image store functions, the precision of the return type matches the precision of the sampler type:

```
uniform lowp sampler2D sampler;
highp vec2 coord;
...
lowp vec4 col = texture (sampler, coord); // texture() returns lowp
```

Otherwise:

- For prototypes that do not specify a resulting precision qualifier, the precision will be the same as the precision of the operation. (As defined earlier.)

- For prototypes that do specify a resulting precision qualifier, the specified precision qualifier is the precision qualification of the result."

Add precision qualifiers to the following in desktop versions:

```
genIType floatBitsToInt (highp genFType value)
genUType floatBitsToUint(highp genFType value)
genFType intBitsToFloat (highp genIType value)
genFType uintBitsToFloat(highp genUType value)

genFType frexp(highp genFType x, out highp genIType exp)
genFType ldexp(highp genFType x,  in highp genIType exp)

highp uint packSnorm2x16(vec2 v)
vec2 unpackSnorm2x16(highp uint p)
highp uint packUnorm2x16(vec2 v)
vec2 unpackUnorm2x16(highp uint p)
vec2 unpackHalf2x16(highp uint v)
```

```
  vec4 unpackUnorm4x8(highp uint v)
  vec4 unpackSnorm4x8(highp uint v)

  genIType bitfieldReverse(highp genIType value)
  genUType bitfieldReverse(highp genUType value)
  genIType findMSB(highp genIType value)
  genIType findMSB(highp genUType value)
  genUType uaddCarry(highp genUType x, highp genUType y,
                     out lowp genUType carry)
  genUType usubBorrow(highp genUType x, highp genUType y,
                      out lowp genUType borrow)
  void umulExtended(highp genUType x, highp genUType y,
                    out highp genUType msb, out highp genUType lsb)
  void imulExtended(highp genIType x, highp genIType y,
                    out highp genIType msb, out highp genIType lsb)
```

Remove section 8.10 Atomic-Counter Functions

Add a section

"8.X Subpass Functions

"Subpass functions are only available in a fragment shader.

"Subpass inputs are read through the built-in functions below. The gvec... and gsubpass... are matched, where they must both be the same floating point, integer, or unsigned integer variants.

Add a table with these two entries (in the same cell):

```
"gvec4 subpassLoad(gsubpassInput   subpass)
 gvec4 subpassLoad(gsubpassInputMS subpass, int sample)"
```

With the description:

"Read from a subpass input, from the implicit location (x, y, layer) of the current fragment coordinate."

Changes to the grammar

Arrays can no longer require the size to be a compile-time folded constant expression. Change

```
  | LEFT_BRACKET constant_expression RIGHT_BRACKET
```

to

```
  | LEFT_BRACKET conditional_expression RIGHT_BRACKET
```

and change

  | array_specifier LEFT_BRACKET constant_expression RIGHT_BRACKET

to

  | array_specifier LEFT_BRACKET conditional_expression RIGHT_BRACKET

Remove the ATOMIC_UINT type_specifier_nonarray.

Remove all instances of the SUBROUTINE keyword.

Issues

1. Can we have specialization sizes in an array in a block? That prevents putting known offsets on subsequent members.

   RESOLUTION: Yes, but it does not affect offsets.

2. Can a specialization-sized array be passed by value?

   RESOLUTION: Yes, if they are sized with the same specialization constant.

3. Can a texture array be variably indexed? Dynamically uniform?

   Resolution (bug 14683): Dynamically uniform indexing.

4. Are arrays of a descriptor set all under the same set number, or does, say, an array of size 4 use up 4 descriptor sets?

   RESOLUTION: There is no array of descriptor sets. Arrays of resources are in a single descriptor set and consume a single binding number.

5. Which descriptor set arrays can be variably or non-uniformly indexed?

   RESOLUTION: There is no array of descriptor sets.

6. Do we want an alternate way of doing composite member specialization constants? For example,

       layout(constant_id = 18) gl_WorkGroupSize.y;

   Or

       layout(constant_id = 18, local_size_y = 16) in;

   Or

```
    layout(constant_id = 18) wgy = 16;
    const ivec3 gl_WorkGroupSize = ivec3(1, wgy, 1);
```

RESOLUTION: No. Use local_size_x_id etc. for workgroup size, and defer any more generalized way of doing this for composites.

7. What names do we really want to use for

```
    gl_VertexIndex            base, base+1, base+2, ...
    gl_InstanceIndex          base, base+1, base+2, ...
```

RESOLUTION: Use the names above.

Note that gl_VertexIndex is equivalent to OpenGL's gl_VertexID in that it includes the value of the baseVertex parameter. gl_InstanceIndex is NOT equivalent to OpenGL's gl_InstanceID because gl_InstanceID does NOT include the baseInstance parameter.

8. What should "input subpasses" really be called?

RESOLVED: subpassInput.

9. The spec currently does not restrict where sampler constructors can go, but should it? E.g., can the user write a shader like the following:

```
  uniform texture2D t[MAX_TEXTURES];
  uniform sampler s[2];

  uniform int textureCount;
  uniform int sampleCount;
  uniform bool samplerCond;

  float ShadowLookup(bool pcf, vec2 tcBase[MAX_TEXTURES])
  {
      float result = 0;

      for (int textureIndex = 0; textureIndex < textureCount; ++textureIndex)
      {
          for (int sampleIndex = 0; sampleIndex < sampleCount; ++sampleIndex)
          {
              vec2 tc = tcBase[textureIndex] + offsets[sampleIndex];
              if (samplerCond)
                  result += texture(sampler2D(t[textureIndex], s[0]), tc).r;
              else
                  result += texture(sampler2D(t[textureIndex], s[1]), tc).r;
          }
```

Or, like this?

```
 uniform texture2D t[MAX_TEXTURES];
 uniform sampler s[2];

 uniform int textureCount;
 uniform int sampleCount;
 uniform bool samplerCond;

 sampler2D combined0[MAX_TEXTURES] = sampler2D(t, s[0]);
 sampler2D combined1[MAX_TEXTURES] = sampler2D(t, s[1]);

 float ShadowLookup(bool pcf, vec2 tcBase[MAX_TEXTURES])
 {
     for (int textureIndex = 0; textureIndex < textureCount; ++textureIndex) {
         for (int sampleIndex = 0; sampleIndex < sampleCount; ++sampleIndex) {
             vec2 tc = tcBase[textureIndex] + offsets[sampleIndex];
             if (samplerCond)
                 result += texture(combined0[textureIndex], tc).r;
             else
                 result += texture(combined1[textureIndex], tc).r;
         }
     ...
```

RESOLUTION (bug 14683): Only constructed at the point of use, where passed as an argument to a function parameter.

Revision History

Rev.	Date	Author	Changes
40	21-May-2017	JohnK	Require in/out explicit locations
39	14-Apr-2017	JohnK	Update overview for StorageBuffer storage class.
38	14-Apr-2017	JohnK	Fix Vulkan public issue #466: texture2D typo.
37	26-Mar-2017	JohnK	Fix glslang issue #369: remove gl_NumSamples.
36	13-Feb-2017	JohnK	Fix public bug 428: allow anonymous push_constant blocks.
35	07-Feb-2017	JohnK	Add 'offset' and 'align' to all versions
34	26-Jan-2017	JohnK	Allow the ternary operator to result in a specialization constant
33	30-Aug-2016	JohnK	Allow out-of-order offsets in a block
32	1-Aug-2016	JohnK	Remove atomic_uint and more fully subroutine
31	20-Jul-2016	JohnK	Have desktop versions respect mediump/lowp
30	12-Apr-2016	JohnK	Restrict spec-const operations to non-float
29	5-Apr-2016	JohnK	Clarify disallowance of spec-const arrays in initializers
28	7-Mar-2016	JohnK	Make push_constants not have sets
27	28-Feb-2016	JohnK	Make the default by origin_upper_left

26	17-Feb-2016	JohnK	Expand specialized array semantics
25	10-Feb-2016	JohnK	Incorporate resolutions from the face to face
24	28-Jan-2016	JohnK	Update the resolutions from the face to face
23	6-Jan-2016	Piers	Remove support for gl_VertexID and gl_InstanceID since they aren't supported by Vulkan.
22	29-Dec-2015	JohnK	support old versions and add semantic mapping
21	09-Dec-2015	JohnK	change spelling *subpass* -> *subpassInput* and include this and other texture/sample types in the descriptor-set-0 default scheme
20	01-Dec-2015	JohnK	push_constant default to std430, opaque types can only aggregate as arrays
19	25-Nov-2015	JohnK	Move "Shadow" from texture types to samplerShadow
18	23-Nov-2015	JohnK	Bug 15206 - Indexing of push constant arrays
17	18-Nov-2015	JohnK	Bug 15066: std140/std43 defaults
16	18-Nov-2015	JohnK	Bug 15173: subpass inputs as arrays
15	07-Nov-2015	JohnK	Bug 14683: new rules for separate texture/sampler
14	07-Nov-2015	JohnK	Add specialization operators, local_size_*_id rules, and input dvec3/dvec4 always use two locations
13	29-Oct-2015	JohnK	Rules for input att. numbers, constant_id, and no subpassLoadMS()
12	29-Oct-2015	JohnK	Explain how gl_FragColor is handled
11	9-Oct-2015	JohnK	Add issue: where can sampler constructors be
10	7-Sep-2015	JohnK	Add first draft specification language
9	5-Sep-2015	JohnK	- make specialization id's scalar only, and add local_size_x_id... for component-level workgroup size setting - address several review comments
8	2-Sep-2015	JohnK	switch to using the *target* style of target types (bug 14304)
7	15-Aug-2015	JohnK	add overview for input targets
6	12-Aug-2015	JohnK	document gl_VertexIndex and gl_InstanceIndex
5	16-Jul-2015	JohnK	push_constant is a layout qualifier VULKAN is the only versioning macro constantID -> constant_id
4	12-Jul-2015	JohnK	Rewrite for clarity, with proper overview, and prepare to add full semantics
3	14-May-2015	JohnK	Minor changes from meeting discussion
2	26-Apr-2015	JohnK	Add controlling features/capabilities
1	26-Mar-2015	JohnK	Initial revision