Overview of OpenGL 4.0

Chapter Objectives

This chapter provides an overview of OpenGL version 4.0, describing the following:

- Tessellation Shaders
- Indirect Rendering
- Shader subroutines
- Enhancements to the OpenGL Shading Language

Note: This is a work-in-progress. Note that some features of OpenGL 4.0, including enhancements to transform feedback, and sample shading are not included in this draft.
Overview of OpenGL 4.0 Features

The enhancements that form OpenGL 4.0 are as follows:

- Support for GLSL 4.00, including
  - new textureLOD functions for fragment shading that automatically compute the level-of-detail values that would be preformed as if a normal texture lookup was performed.
  - new textureGather functions that return the $2\times2$ set of texels that would have been used for bi-linear filtering. The first (red) component of each texel is returned in a vector of four values.
- The ability to set blend functions and equations for each color output
- Numerous new features for GLSL 4.00
- Support for double-precision values for uniforms (including vectors and matrices) and for double-precision values inside of shaders
- Request that a minimum number of unique samples are when multisampling
- Indirect function calls inside of a shader
- Addition of two new pipeline stages: tessellation control shaders and tessellation evaluation shaders for operating on patches
- Addition of three-component buffer formats: RGB32F, RGB32I, and RGB32UI
- Cube map texture arrays
- Additions for transform feedback:
- transform feedback objects to encapsulate transform feedback state
- the ability to pause and resume transform feedback operations
- the ability to draw primitives captured in transform feedback mode without querying the captured primitive count.
- increased flexibility in how vertex attributes are written to buffer objects and support for multiple separate vertex streams.

**Fundamental OpenGL Enhancements**

While most changes in OpenGL manifest themselves as new API entry points or shader stages, or additions to the OpenGL Shading Language (GLSL), some enhancements affect the core operation of OpenGL.

**Cube Map Arrays**

<<< Adds support of arrays of cube maps. 

There’s lots of support of texture arrays already in the API, so this isn’t too massive. No new API calls were added; just a few in GLSL, which may be the place to discuss further. Not quite sure how much to go into it. >>>

**RGB 32-bit Texture Buffers**

<<< This adds three new internal texture formats: GL_RGB32F, GL_RGB32I, and GL_RGB32UI. 

This just filled in some overlooked functionality when floating-point buffers formats were add into the specification >>>.
Tessellation Shaders

The likely most interesting addition to OpenGL in version 4.0 is *tessellation* using programmable shaders. These new shaders tessellate a *patch*—a new OpenGL rendering primitive—into lines, triangles or quads for subsequent rendering. The updated shading pipeline for OpenGL version 4.0 is illustrated in Figure 1.

**Figure 1** The OpenGL version 4.0 shading pipeline

The process of tessellation in OpenGL is done using two user-specified shaders, and a fixed-operation stage:

- a *tessellation control* shader that principally specifies how the patch primitive is to be tessellated into fundamental rendering primitives (like triangles), then
- the fixed-operation processing stage that uses the output of the tessellation control shader to generate the vertices of the triangles, quad, or lines of the tessellated patch,
- and finally, a *tessellation evaluation* shader, which specifies the position (and other per-vertex attributes) of the generated vertices

Either shader is optional, but you will likely find using the combination of them the most practical.

Tessellation Patches

Tessellation doesn’t operate on OpenGL’s classic geometric primitives: points, lines, and triangles, but uses a new primitive called a *patch*. Patches are new input types for OpenGL’s drawing commands, like `glDrawArrays()`[^1], and are operated on by all of the active shading stages in OpenGL version 4.0.

[^1]: They are also acceptable as input into `glBegin()` if you’re using a compatibility profile.

---

**Overview of OpenGL 4.0**
the pipeline (by comparison to the other primitive types, which can at most be processed by vertex, fragment, and geometry shaders).

Patches are an ordered collections of vertices, with the number of vertices in a patch being specified by `glPatchParameteri()`.

```c
void glPatchParameteri(GLenum pname, GLint value);
```

Specifies the number of vertices using `value` for a patch. `pname` must be set to `GL_PATCH_VERTICES`. A `GL_INVALUE_VALUE` error is generated if `value` is less than zero, or greater than `GL_MAX_PATCH_VERTICES`. The default number of vertices for a patch is three.

To specify a patch, use the input type `GL_PATCHES` into any OpenGL drawing command. Example 2 demonstrates issuing two patches, each with four vertices

```c
GLfloat vertices [][] = {
    {-0.75, -0.25}, {-0.25, -0.25}, {-0.25, 0.25}, {-0.75, 0.25},
    { 0.25, -0.25}, { 0.75, -0.25}, { 0.75, 0.25}, { 0.25, 0.25}
};
```

```c
glBindVertexArray( VAO );
glBindBuffer( GL_ARRAY_BUFFER, VBO);
glBufferData( GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW );
glVertexAttribPointer( vPos, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
glPatchParameteri( GL_PATCH_VERTICES, 4 );
glDrawArrays( GL_PATCHES, 0, 8 );
```

**Example 2**  Specifying tessellation patches

The vertices of each patch are first processed by the currently bound vertex shader, and then used to initialize the array `gl_in`, which is implicitly declared in the tessellation control shader. The number of elements in `gl_in` is the same as the patch size specified by `glPatchParameteri()`. Inside of a tessellation control shader, the variable `gl_PatchVerticesIn` provides the number of elements in `gl_in` (as does querying `gl_in.length()`).
Tessellation Control Shaders

Once your application issues a patch, the tessellation control shader will be called, provided one is bound (otherwise, the data values will be passed directly to the primitive generator). The shader has two tasks:

- generate the list of vertices to be passed tessellation primitive generator, as well as update any per-vertex, or per-patch attribute values.
- specify the tessellation factors that control the operation of the tessellation primitive generator. These are special tessellation control shader variables called gl_TessLevelInner and gl_TessLevelOuter.

Generating output patch vertices

Tessellation control shaders can use the input patch vertices specified by the application, and generate a set of vertices to be used in subsequent tessellation processing. The number of vertices that the control shader will output is controlled by a required line of GLSL like the following:

```glsl
layout (vertices = n) out;
```

The value set by the `vertices` parameter in the `layout` directive does two things: it sets the size of output vertex array in the control shader, which is called `gl_out`; and specifies how many times the control shader will execute. Control shaders execute once for each output vertex in a patch.

In order to determine which output vertex is being processed, the tessellation shader can use the `gl_InvocationID` variable. Its value is most often used as an index into the `gl_out` array. While a control shader is executing, it can have access to all vertex data—both input and output. This can lead to issues where a shader invocation might need data values for a shader that hasn’t executed yet. Control shaders can use the GLSL `barrier()` function which causes all of the control shaders for an input patch to wait until all of the shaders reach that point, thus guaranteeing that all of the data values you might set will be computed.
A common idiom of control tessellation shaders is just passing the input vertices out of the shader. The GLSL code for that is show in

```
gl_out[gl_InvocationID].gl_Position =
  gl_out[gl_InvocationID].gl_Position
```

**Example 3**  Passing through tessellation control shader patch vertices

### Tessellation Control Shader Variables

The `gl_in` array is actually an array of structures, with each element defined as:

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

The `gl_out` array is similarly defined, except for its dimensions, which is specified by the value set for `vertices` in the `out` layout qualifier.

Additionally, the following scalar values are provided for determining which primitive and output vertex invocation is being shaded

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gl_InvocationID</code></td>
<td>Invocation index for the output vertex of the current tessellation control shader</td>
</tr>
<tr>
<td><code>gl_PrimitiveID</code></td>
<td>Primitive index for current input patch</td>
</tr>
<tr>
<td><code>gl_PatchVerticesIn</code></td>
<td>Number of vertices in the input patch, which is the dimension of <code>gl_in</code></td>
</tr>
</tbody>
</table>

**Table 3-1**  Tessellation control shader input variables

### Controlling Tessellation

The other function of a control shader is to specify the tessellation of the output patch.

While we haven’t discussed them yet, the tessellation evaluation shaders control the type of output patch for rendering, and consequently, the domain where tessellation—that is, primitive generation by subdivision—
occurs. For the moment, accept that there are three types of tessellation domains: a quadrilateral, a triangle, and a collection of iso-lines.

Each type of domain is subdivided by using two sets of values, the tessellation levels, that you need to set, either inside of the tessellation control shader, or if you’re not using a control shader, then using the function \texttt{glPatchParameterfv()}

```c
void glGetPatchParameterfv(GLenum pname, const GLfloat *values);
```

Sets the default tessellation levels for when no tessellation control shader is bound. \texttt{pname} must be either \texttt{GL_PATCH_DEFAULT_OUTER_LEVEL}, or \texttt{GL_PATCH_DEFAULT_INNER_LEVEL}.

When \texttt{pname} is \texttt{GL_PATCH_DEFAULT_OUTER_LEVEL}, \texttt{values} must specify an array of four floating-point values that specify the four outer-tessellation levels.

Similarly, when \texttt{pname} is \texttt{GL_PATCH_DEFAULT_INNER_LEVEL}, \texttt{values} must specify an array of two floating-point values that specify the two inner-tessellation levels.

The amount of tessellation is controlled by specifying two sets of values, the \textit{inner-} and \textit{outer-tessellation levels}. The outer-tessellation level controls how the perimeter of the domain is subdivided, and is stored in an implicitly declared four-element array named \texttt{gl_TessLevelOuter}. Similarly, the inner-tessellation level specifies how the interior of the domain is subdivided, and are stored in a two-element array named \texttt{gl_TessLevelInner}. The number of values used from the arrays depend on the type of patch being tessellated, and all of the values are floating-point (with fractional values accepted).

Figure 4 shows the mapping of inner- and outer-tessellation levels to the domains of the patches. Any of the \texttt{gl_TessLevelOuter} values represent how many segments the respective edge should be partitioned into. Similarly, \texttt{gl_TessLevelInner} specify how many segments the interior should be divided into. With those values in place, the fixed-operation primitive generator will generate triangles (for quads and triangles), and lines for the iso-line patch type.
Figure 4    Quadrilateral and triangular tessellation factors

To provide a better illustration, the tessellation control shader shown in Example 5 generates the tessellation shown in Figure 6.

```cpp
#version 400 core
layout (vertices = 4) out;
void main()
{
  gl_TessLevelOuter[0] = 2.0;
  gl_TessLevelOuter[1] = 3.0;
  gl_TessLevelOuter[2] = 2.0;
  gl_TessLevelOuter[3] = 3.0;
  gl_TessLevelInner[0] = 3.0;
  gl_TessLevelInner[1] = 4.0;

  gl_out[gl_InvocationID].gl_Position =
      gl_in[gl_InvocationID].gl_Position;
}
```

Example 5     An example of tessellation levels in a tessellation control shader

The tessellation levels specified in Example 5 cause the top and bottom outside edges in domain space to be equally divided into two parts, while the left and right edges are equally divided into three. The inner levels, which may be somewhat less intuitive to understand, cause the domain to
be divided into four equal horizontal partitions, and divided vertically into three equal regions. From those partitions, the fixed-operation creates the triangular regions you see in Figure 6, and then call the control evaluation shader.

![Figure 6](image)

**Figure 6** The tessellation output from the Example 5 shader

**Tessellation Evaluation Shaders**

The final phase in OpenGL's tessellation pipeline is to allow shading on the vertices generated from the fixed-operation primitive generation stage. For each vertex that is associated with a primitive exported from the primitive generation stage, the tessellation evaluation shader, if one is bound is executed. As compared to geometry shaders, the geometry information (neighboring vertices, adjacency, etc.) is not presented in the evaluation shader; only any user-defined uniforms or attributes, and information for controlling the vertex’s position.
Controlling Primitive Generation

Similar to the tessellation control shader, the tessellation evaluation shader controls its input parameters using a `layout` directive. The parameters specified control the type of primitive the vertices originate from, their orientation (for face culling), and how the tessellation levels should be applied during primitive generation.

The evaluation shader specifies the type of primitive, among other options, that should be generated from the primitive generator. In return, the primitive generator provides `tessellation coordinates` in domain space for the generated primitive.

The tessellator’s domain space is the unit square, and is mapped across the entire input patch. There are three types of input primitive that an evaluation shader can specify, which control the input data that shader receives:

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Description</th>
<th>Domain Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>quads</td>
<td>A rectangular shaped subregion of the domain space</td>
<td>a ((u,v)) pair specifying the lower-left corner of the quad in domain space. ((1-u, 1-v)) specifies the opposite corner of the quad.</td>
</tr>
<tr>
<td>triangles</td>
<td>A triangular shaped subregion of the domain space</td>
<td>barycentric coordinates ((a, b, c)) of the vertices of the triangle</td>
</tr>
<tr>
<td>isolines</td>
<td>A collection of equally-spaced horizontal lines in domain space</td>
<td>a ((u, v)) pair, with (u) specifying the partition along the isoline, and (v) specifying which isoline the vertex belongs to</td>
</tr>
</tbody>
</table>

Table 6-1 Evaluation shader primitive types

As with any filled primitive in OpenGL, the order the vertices are issued determines the facedness of the primitive. Since we don’t issue the vertices in this case, but rather have the primitive generator do it on our behalf, we can specify the face winding of our primitives. Additional options to the `layout` directive, either `cw`, or `ccw` convey our request.

Additionally, we can control how the fractional values for the tessellation levels are used in determining the primitive generation. Recall that the tessellation level specifies how many partitions an edge should be divided
into. In the case of fractional values, we have some options for controlling how the fractional part is employed.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equal_spacing</td>
<td>Tessellation level is clamped to ([1, \text{max}]), and is then rounded up to next largest integer value</td>
</tr>
<tr>
<td>fractional_even_spacing</td>
<td>The value is clamped to ([2, \text{max}]), and then rounded up to next largest integer value (n). The edge is then divided into (n-2) equal length parts, and two other parts, one at either end, which may be shorter than the other lengths.</td>
</tr>
<tr>
<td>fractional_odd_spacing</td>
<td>The value is clamped to ([1, \text{max}-1]), and then rounded up to next largest integer value (n). The edge is then divided into (n-2) equal length parts, and two other parts, one at either end, which may be shorter than the other lengths.</td>
</tr>
</tbody>
</table>

**Table 6-2** Options for controlling tessellation level effects

Finally, should you want to output points, as compared to isolines or filled regions, you can supply the `point_mode` option, which will render a single point for each vertex processed by the evaluation shader.

**Specifying a Vertex’s Position**

The vertices output from the tessellation control shader (as `gl_out` in the control shader) are made available in the evaluation shader in the `gl_in` variable, which when combined with the tessellation coordinates, can be used to generate the output vertex’s position.

Tessellation coordinates are provided to the shader in the variable `gl_TessCoord`. The values depend on the primitive type (see Table 6-1) specified in the `layout` directive.

In Example 7, we use a combination of equal-spaced quads to render a simple patch. In this case, the tessellation coordinates are used to color the surface, and illustrates how to compute vertex’s position.\(^2\)

\(^2\) The direct computation of the \(u, v\), and their one-minus variant coefficients could easily be replaced by the `mix()`.
#version 400 core

layout (quads, equal_spacing, ccw) in;

out vec4 color;

void main()
{
    float u = gl_TessCoord.x;
    float omu = 1 - u;
    float v = gl_TessCoord.y;
    float omv = 1 - v;

    color = gl_TessCoord;

    gl_Position =
    omu * omv * gl_in[0].gl_Position +
    u   * omv * gl_in[1].gl_Position +
    u   *   v * gl_in[2].gl_Position +
    omu *   v * gl_in[3].gl_Position;
}

**Example 7**  A sample tessellation evaluation shader

**Tessellation Evaluation Shader Variables**

Similar to tessellation control shaders, evaluation shaders have a `gl_in` array that is actually an array of structures, with each element defined as:

```c
in gl_perVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[gl_PatchVerticesIn];
```
Additionally, the following scalar values are provided for determining which primitive and for computing the position of the output vertex.

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gl_PrimitiveID</td>
<td>Primitive index for current input patch</td>
</tr>
<tr>
<td>gl_PatchVerticesIn</td>
<td>Number of vertices in the input patch, which is the dimension of gl_in</td>
</tr>
<tr>
<td>gl_TessLevelOuter[4]</td>
<td>Outer-tessellation level values</td>
</tr>
<tr>
<td>gl_TessLevelInner[2]</td>
<td>Inner-tessellation level values</td>
</tr>
<tr>
<td>gl_TessCoord</td>
<td>Coordinates in patch domain space of the vertex being shaded in the evaluation shader</td>
</tr>
</tbody>
</table>

Table 7-1  Tessellation control shader input variables

The output vertex’s data is stored in an interface block defined as:

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

**Indirect Rendering**

As GPUs become increasingly faster, data locality for processing has become ever more important. Until the introduction of indirect rendering in OpenGL version 4.0, rendering required all parameters of a rendering call to be processed on the host CPU. Indirect rendering effectively allows storing the parameters of drawing calls in a buffer object in GPU memory.

Two new functions were introduced that reflect OpenGL-style rendering using data stored in a buffer object: `glDrawArraysIndirect()`, and `glDrawElementsIndirect()`. For each call, the data in the buffer object needs to be stored in an indirect buffer object, which is created by calling `glBindBuffer()` with the `target` parameter set to `GL_DRAW_INDIRECT_BUFFER`, which is created and initialized like any other type of buffer object. If no indirect buffer object is bound when an indirect rendering call is executed, no rendering is done.
The `glDrawArraysIndirect()` function requires that four values are stored in the indirect buffer object. It may be helpful to think of encapsulating the parameters of `glDrawArraysInstanced()` in a structure described in Figure 8:

```c
struct DrawArraysIndirectParameters {
    GLuint first;
    GLuint count;
    GLuint primCount;
    GLuint zero;  // must be set to zero
};
```

**Figure 8** Helper structure definition for `glDrawArraysIndirect()`

The last parameter is required to have a value of zero; the results if it is not is undefined. Additionally, the alignment of the values in the indirect buffer have restrictions, as described below.

```c
void glDrawArraysIndirect(GLenum mode, const GLvoid *indirect);
```

Effectively calls `glDrawArraysInstanced()` retrieving the values for `first`, `count`, and `primCount` from a buffer object of type GL_INDIRECT_BUFFER.

A GL_INVALID_OPERATION error is generated if `glDrawArraysIndirect()` attempts to read more values than are stored in the associated array buffer object, or if the value of `indirect` is not aligned to a multiple of `sizeof(GLuint)`.

Similar to `glDrawArraysIndirect()`, for `glDrawElementsIndirect()`, encapsulating the parameters of `glDrawElementsInstancedBaseVertex()` within a structure, as described

```c
struct DrawElementsIndirectParameters {
    GLuint count;
    GLuint primCount;
    GLuint firstIndex;
    GLint baseVertex;
    GLuint zero;  // must be set to zero
};
```

**Figure 9** Helper structure definition for `glDrawElementsIndirect()`
Again, the value of the last field must be zero, and the alignment of the structures in the indirect buffer object must be a multiple of the value of sizeof(GLuint).

```c
void glDrawElementsIndirect(GLenum mode, GLenum type, const GLvoid *indirect);
```

Effectively calls `glDrawElementsInstancedBaseVertex()` retrieving the values for `count`, `primCount`, `firstIndex`, and `baseVertex` from the memory pointed to by `indirect`.

A GL_INVALID_OPERATION error is generated if no element array buffer object is bound, or if `glDrawElementsIndirect()` attempts to read more values than are stored in the associated array buffer object, or if the value of `indirect` is not aligned to a multiple of sizeof(GLuint).

## Shader Subroutines

While GLSL allowed you to define functions in shaders, the call flow of those functions was always static. To dynamically select between multiple functions, you either created two distinct shaders, or used an if-statement to make a run-time selection, like demonstrated in Example 10.

```c
void func_1() { ... }
void func_2() { ... }

uniform int func;

void
main()
{
    if (func == 1)
        func_1();
    else
        func_2();
}
```

**Example 10**  Static shader control flow
Shader subroutines are conceptually similar to function pointers in C, for implementing dynamic subroutine selection. In your shader, you specify a subroutine type and use that type when declaring the set of subroutines eligible for dynamic use. Then, you choose which subroutine from the set to execute in the shader by setting a subroutine uniform variable.

GLSL Subroutine Setup

When you want to use subroutine selection inside of a shader, there are three steps required to set up the pool of subroutines:

1. Define the subroutine type using the subroutine keyword

   subroutine returnType subroutineType( type param, ... );

   where returnType is any valid type that a function can return, and subroutineType is any valid name. As with function prototypes, only the parameter types are required; the parameter names are optional.

   Hint: Think of this like a typedef in C, with subroutineType as the newly defined type.

2. Using the subroutineType you just defined, define the set of subroutines that you would like to dynamically select from using the subroutine keyword. The prototype for a subroutine function looks like:

   subroutine (subroutineType) returnType functionName( ... );

3. Finally, specify the subroutine uniform variable that will hold the “function pointer” for the subroutine you’ve selected in your application:

   subroutine uniform subroutineType variableName;

Demonstrating those steps together, consider the following example where we would like to dynamically select between ambient and diffuse lighting:

subroutine vec4 LightFunc( vec3 ); // Step 1

subroutine (LightFunc) vec4 ambient( vec3 n ) // Step 2
{ return Materials.ambient; }
subroutine (LightFunc) vec4 diffuse( vec3 n ) // Step 2 (again)
{
    return Materials.diffuse *
    max(dot(normalize(n), LightVec.xyz). 0.0);
}

subroutine uniform LightFunc materialShader; // Step 3

Example 11  Declaring a set of subroutines

A subroutine is not restricted to being a single type of subroutine (e.g., LightFunc in Example 11). If you have defined multiple types of subroutines, you can associate any number of the types with a subroutine by adding the type to the list when defining the subroutine, as demonstrated,

subroutine void Type_1();
subroutine void Type_2();
subroutine void Type_3();

subroutine (Type_1, Type_2) Func_1();
subroutine (Type_1, Type_3) Func_2();

subroutine uniform Type_1 func_1;
subroutine uniform Type_2 func_2;
subroutine uniform Type_3 func_3;

In this case, func_1 could use either Func_1, or Func_2. However, func_2 would be limited to only using Func_1, and likewise, func_3 could only use Func_2.

Selecting Shader Subroutines

Once you have all your subroutine types and functions defined in your shaders, you only need to query a few values from the linked shader program, and then use those values to select the appropriate function.

In step 3. described in “GLSL Subroutine Setup,” a subroutine uniform value was declared, and we will need its location in order to set its value. As compared to other shader uniforms, subroutine uniforms use glGetSubroutineUniformLocation() to retrieve their locations.
Once we have the subroutine uniform to assign values to, we need to
determine the indices of the subroutines inside of the shader. For that, we
can call \texttt{glGetSubroutineIndex()}. 

\begin{verbatim}
GLint glGetSubroutineUniformLocation(GLuint program, 
   GLenum shadertype, const char* name);
\end{verbatim}

Returns the location of the subroutine uniform named \textit{name} in \textit{program}
for the shading stage specified by \textit{shadertype}. \textit{name} is a null-terminated
character string, and \textit{shadertype} must be one of \texttt{GL_VERTEX_SHADER}, \texttt{GL_}
\texttt{TESS_CONTROL_SHADER}, \texttt{GL_TESS_EVALUATION_SHADER}, \texttt{GL_}
\texttt{GEOMETRY_SHADER}, or \texttt{GL_FRAGMENT_SHADER}.

If \textit{name} is not an active subroutine uniform, minus one (–1) is returned. If
\textit{program} is not a successfully linked shader program, a \texttt{GL_INVALID_}
\texttt{OPERATION} error will be generated.

Once we have the subroutine uniform to assign values to, we need to
determine the indices of the subroutines inside of the shader. For that, we
can call \texttt{glGetSubroutineIndex()}. 

\begin{verbatim}
GLuint glGetSubroutineIndex(GLuint program, GLenum shadertype, 
   const char* name);
\end{verbatim}

Returns the index of the shader function associated with \textit{name} from
\textit{program} for the shading stage specified by \textit{shadertype}. \textit{name} is a null-
terminated character string, and \textit{shadertype} must be one of \texttt{GL_VERTEX_}
\texttt{SHADER}, \texttt{GL_TESS_CONTROL_SHADER}, \texttt{GL_TESS_EVALUATION_}
\texttt{SHADER}, \texttt{GL_GEOMETRY_SHADER}, or \texttt{GL_FRAGMENT_SHADER}.

If \textit{name} is not an active subroutine for the shader for \textit{shadertype}, \texttt{GL_}
\texttt{INVALID_INDEX} is returned.

Once you have both the available subroutine indices, and subroutine
uniform location, use \texttt{glUniformSubroutinesuiv()} to specify which
subroutine should be executed in the shader. All active subroutine uniforms
for a shader stage must be initialized.
Assembling those steps, the following code snippet demonstrates the process for the vertex shader described in Example 11.

```c
GLint materialShaderLoc;
GLuint ambientIndex;
GLuint diffuseIndex;

glUseProgram( program );

materialShaderLoc = glGetSubroutineUniformLocation( program,
    GL_VERTEX_SHADER, "materialShader" );

if ( materialShaderLoc < 0 ) {
    // Error: materialShader is not an active subroutine
    // uniform in the shader.
}

ambientIndex = glGetSubroutineIndex( program,
    GL_VERTEX_SHADER, "ambient" );
diffuseIndex = glGetSubroutineIndex( program,
    GL_VERTEX_SHADER, "diffuse" );

if ( ambientIndex == GL_INVALID_INDEX ||
    diffuseIndex == GL_INVALID_INDEX ) {
    // Error: the specified subroutines are not active in
    // the currently bound program for the GL_VERTEX_SHADER
    // stage.
}
```

Does not contain any figures.
else {
  GLsizei n;
  glGetIntegerv( GL_MAX_SUBROUTINE_UNIFORM_LOCATIONS, &n );

  GLuint *indices = new GLuint[n];
  indices[materialShaderLoc] = ambientIndex;

  glUniformSubroutinesuiv( GL_VERTEX_SHADER, n, indices );

  delete [] indices;
}

Note: calling glUseProgram() will reset all of the subroutine uniform values to an implementation-dependent ordering.

Overview of GLSL 4.00 Features

New GLSL Version

As with any new GLSL release, the version number is updated, and in the case of GLSL, your shaders will need to begin with

#version 400

Optionally, you can include a profile indicator, which can either be core or compatibility with the version number to indicate which profile, and which implicit variables, should be defined.

Texture LOD Query

<<< Describe the textureQueryLOD() GLSL function.

It returns a vec2 that contains the clamped texture mipmap levels that would have been used by the normal texture sampling mechanisms.

If the texture is incomplete, the returned results are undefined.>>>
Texture Gather

<<< Describe the `textureGather()` function.

Samples a texture determining the four texels that would have been sampled using GL_LINEAR mode for the base level of the texture, and returns a vec4 of the specified component (which is an optional parameter) to the call.

There are also `textureGatherOffset()` functions that take a constant offset that is applied to the provided texture coordinates (like `textureOffset()`).

Additionally, `textureGatherOffsets()` takes an array of texture coordinate offsets that are combined with the texture coordinate to before sampling. The texture gather operation occurs like normal for each texture coordinate (i.e., the 2x2 texel square is computed for each offset like normal using GL_LINEAR sampling, but only the upper-left texel from each texel square is returned). >>>

Double-precision (64-bit) Floating-Point Support

Computations can be carried out in 64-bit floating-point precision, effectively adding double as a first-class type in GLSL.

This functionality also adds support for specifying double-precision uniform variables.

```c
void glUniform[1234]|ld ui](GLint location, TYPE value);
void glUniform[1234]|ld ui](GLint location, GLsizei count,
    const TYPE *values);
void glUniformMatrix[234]|fd](GLint location, GLsizei count,
    GLboolean transpose, const TYPE *values);
void glUniformMatrix[2x3,2x4,3x2,3x4,4x2,4x3]|fd](GLint location,
    GLsizei count, GLboolean transpose,
    const TYPE *values);
```
Sets the value for the uniform variable associated with the index location. The vector form loads count sets of values (from one to four values, depending upon which glUniform*() call is used) into the uniform variables starting location. If location is the start of an array, count sequential elements of the array are loaded.

The floating-point forms can be used to load a single- or double-precision floating-point value, a vector or an array of floating-point values, or an array of vectors of floating-point values.

The integer forms can be used to update a single integer, an integer vector, an array of integers, or an array of integer vectors. Additionally, individual and arrays of texture samplers can also be loaded.

For glUniformMatrix{234}*, count sets of 2 × 2, 3 × 3, or 4 × 4 matrices are loaded from values.

For glUniformMatrix{2x3,2x4,3x2,3x4,4x2,4x3}*, count sets of like-dimensioned matrices are loaded from values. If transpose is GL_TRUE, values are specified in row-major order (like arrays in “C”); or if GL_FALSE is specified, values are taken to be in column-major order (ordered in the same manner as glLoadMatrix()).