GL: graphics library

当一个对象创建后，Java虚拟机（JVM）就会给这个对象分配一个引用自身的指针，这个指针的名字就是this。this只能在类中的非静态方法中使用

super是用在子类中，目的是访问直接父类中被屏蔽的成员，注意是直接父类（就是类之上最近的超类）。当子类方法中的局部变量或者子类的成员变量与父类成员变量同名时，也就是子类局部变量覆盖父类成员变量时，用“super.成员变量名”来引用父类成员变量。

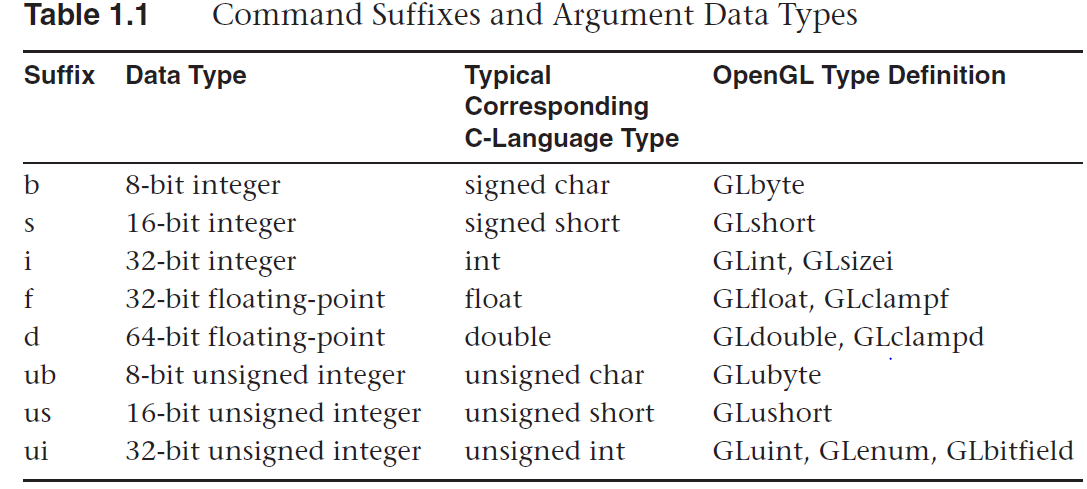
1. Introduction

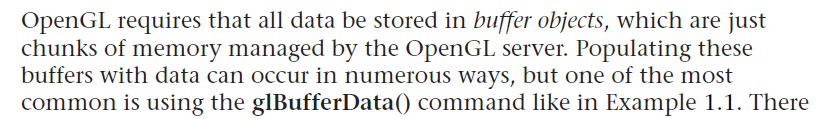
*Rendering*, which we’ve already used without defining previously, is the process by which a computer creates an image from models.

*Models*, or objects are constructed from geometric primitives---*point*s, lines, and triangles - that are specified by their *vertices*.

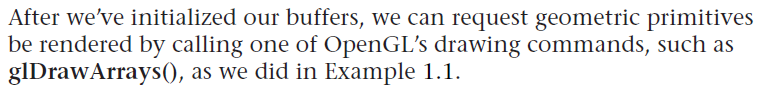
The best way to think of shaders is as little programs that are specifically compiled for your*graphics processing unit*---commonly called a graphics processing unit (GPU). There are four *shader stage*s that you can use. The most common are *vertex shader*s, which process vertex data, and fragment shaders, which operate on the fragments generated by the rasterizer. Both vertex and fragment shaders are required in every OpenGL program.

A *pixel* is the smallest visible element on your *display*. The pixels in your system are stored in a *framebuffer*, which is a chunk of memory that the graphics hardware manages, and feeds to your display device.





gl.glBufferData(GL.*GL\_ARRAY\_BUFFER*, vertexBufferData.capacity()\* Buffers.*SIZEOF\_FLOAT*, vertexBufferData, GL.*GL\_STATIC\_DRAW*);



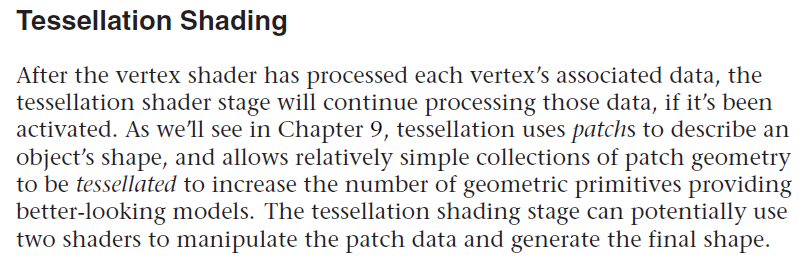
gl.glDrawArrays(GL.*GL\_TRIANGLE\_STRIP*, 0, 3);

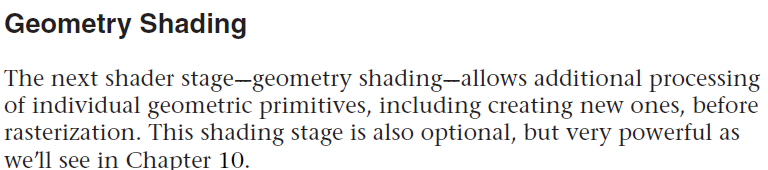


simpleProg = GLSLShaderProgram.*createAndLinkProgramResource*(gl,

"standalone/simple.vp", "standalone/simple.fp", **true**);

simpleProg.activate(gl);





Immediately after clipping, the updated primitives are sent to the rasterizer for fragment generation. Consider a fragment a ‘‘candidate pixel’’, in that pixels have a home in the framebuffer

Vertex shading (including tessellation and geometry shading) determine where on the screen a primitive is, while fragment shading uses that information to determine what color that fragment will be.

Init()

We’ll see numerous OpenGL commands of the form **glGen\***, for allocating names to the various types of OpenGL objects. Until you allocate some memory and have the name reference it, the name isn’t much help. VAO(vertext array object)

gl.glGenVertexArrays(1, iBuf, 0)

Our allocation scheme is called *binding an object:* **glBind\***.

gl.glBindVertexArray(vertexArrayID);

(VBO) Vertex-Buffer Objects. create some names for your vertex-buffer objects:

gl.glGenBuffers(1, iBuf, 0);

bring them into existence by calling **:**

gl.glBindBuffer(GL.*GL\_ARRAY\_BUFFER*, vertexBufferID);

**Loading Data into a Buffer Object**

After initializing our vertex-buffer object, we need to transfer the vertex data from our objects into the buffer object. This is done by the **glBufferData()** routine

gl.glBufferData(GL.*GL\_ARRAY\_BUFFER*,Buffers.*sizeOfBufferElem*(vertexBufferData)\* 3\* Buffers.*SIZEOF\_FLOAT*, vertexBufferData, GL.*GL\_STATIC\_DRAW*);(The last parameter called usage.U*usage* provides a hint as to how the data will be read and written after allocation. Gl\_ STREAM STATIC DYNAMIC \_ DRAW READ COPY)

OpenGL only knows how to draw geometric primitives into coordinate space. In fact, that range of coordinates are known as *normalized-device coordinates* (commonly called NDCs).

Every OpenGL program must provide at least two shaders: a vertex shader and a fragment shader. A shader is a small function written in the OpenGL Shading Language (OpenGL Shading Language (GLSL)). You provide your GLSL shader to OpenGL as a string of characters.

Vertex Shader

#version 330 core

layout(location = 0) in vec3 vertexPosition;

void main() {

gl\_Position.xyz = vertexPosition;

gl\_Position.w = 1.0;

}

#version 330 core, specifies which version of the OpenGL Shading Language we want to use. The ‘‘core’’ relates to wanting to use OpenGL’s core profile, which corresponds with our request to GLUT when we called **glutInitContextProfile()**.

Next, we allocate a *shader variable*. Shader variables are a shader’s connection to the outside world. we have one input variable named vertexPosition, which you can determine by the ‘‘in’’ on its declaration line.

The layout(location = 0) part is called a *layout qualifier*, and provides meta-data for our variable declaration.

The core of the shader is defined in its **main()** routine. there are several shader variables provided by OpenGL that you’ll use, and they all begin with the gl\_ prefix.

Fragment Shader

#version 330 core

out vec4 color;

void main() {

color = vec4(1, 0, 0, 1);

}

out vec4 color: the shader will output values through color. Each fragment is assigned

this vector of four values. In OpenGL, colors are represented in what’s called the *RGB color space*, with each color ranging from [0, 1]. OpenGL really uses an RGBA color space, with the fourth color not really being a color at all. It’s for a value called *alpha*, which is really a measure of translucency. 1.0 indicates the color is fully opaque.

To associate data going into our vertex shader, which is the entrance all vertex data take to get processed by OpenGL, we need to connect our shader ‘‘in’’ variables to a *vertex-attribute array*, and we do that with the **glVertexAttribPointer()** routine.

gl.glVertexAttribPointer(0, 3, GL.*GL\_FLOAT*, **false**, 0, 0L);

*index* 0---- the location value for the respective vertex shader input variable(vertexPosition)

*size* 3----the number of values for each vertex in our array.

*type* GL.*GL\_FLOAT* ----The enumerated value for the GLfloat type.

*normalized* **False----** positional coordinates values can basically take on any value, so we don’t want them

constrained to the range [*−*1, 1]; and, the values are not integer types (e.g., GLint, or GLshort).

*Stride 0----* As our data are ‘‘tightly packed’’, which implies that one set of data values is immediately contiguous in memory to the next, we can use the value zero.

*Pointer* 0L----We set this to zero because our data starts at the first byte (address zero) of our buffer object.

gl.glEnableVertexAttribArray(0);

*ndex 0----index* must be a value between zero and GL\_MAX\_VERTEX\_ATTRIBS *−* 1. passing the index of the attribute array pointer.

Display()

gl.glClear(GL.*GL\_COLOR\_BUFFER\_BIT*);// to black

Begin rendering by clearing our framebuffer. The argument is a bitwise logical OR combination of the values listed in

**Buffer Name**

Color Buffer GL\_COLOR\_BUFFER\_BIT

Depth Buffer GL\_DEPTH\_BUFFER\_BIT

Stencil Buffer GL\_STENCIL\_BUFFER\_BIT

To change the clear color, call **glClearColor()**.

glClearColor(1, 1, 1, 1).//white, set the value right before you call **glClear()** in

**display().**A more efficient solution would be to set the clear color in **init()**. Any values that will be constant over the execution of a program are set in **init()**.

We first call **glBindVertexArray()** to select the vertex array that we want to use as vertex data. gl.glBindVertexArray(vertexArrayID);

Next, we call **glDrawArrays()**, which actually sends vertex data to the OpenGL pipeline.

gl.glDrawArrays(GL.*GL\_LINE\_LOOP*, 0, 4);

Constructs a sequence of geometric primitives using the elements from the currently bound vertex array starting at *0* and ending at *0* + *4 −* 1. *GL\_LINE\_LOOP* specifies what kinds of primitives are constructed.

|  |  |  |
| --- | --- | --- |
| **glDraw command** | **output primtive** | **used in shaders** |
| GL\_POINTS | points | geometry; fragment |
| GL\_LINES | lines | geometry; fragment |
| GL\_LINE\_STRIP | lines | geometry; fragment |
| GL\_LINE\_LOOP | lines | geometry; fragment |
| GL\_LINES\_ADJACENCY | lines\_adjacency | geometry |
| GL\_LINE\_STRIP\_ADJACENCY | lines\_adjacency | geometry |
| GL\_TRIANGLES | triangles | geometry; fragment |
| GL\_TRIANGLE\_STRIP | triangles | geometry; fragment |
| GL\_TRIANGLE\_FAN | triangles | geometry; fragment |
| GL\_TRIANGLES\_ADJACENCY | triangles\_adjacency | geometry |
| GL\_TRIANGLE\_STRIP\_ADJACENCY | triangles\_adjacency | geometry |
| GL\_PATCHES | patches | tessellation control |

gl.glFlush()//requests that any pending OpenGL calls are flushed to the OpenGL server and processed, and then it returns immediately---it doesn’t wait until everything pending is completed.

**glFinish()** function waits until all OpenGL operations in flight are done, and then returns.

Most operational features are turned on and off by the **glEnable()** and **glDisable()** commands.

void **glEnable**(GLenum *capability*);

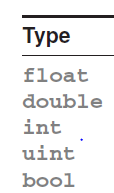
void **glDisable**(GLenum *capability*);

GL\_DEPTH\_TEST for turning on and off depth testing; GL\_BLEND to control blending and GL\_RASTERIZER\_DISCARD for advanced rendering control while doing transform feedback.

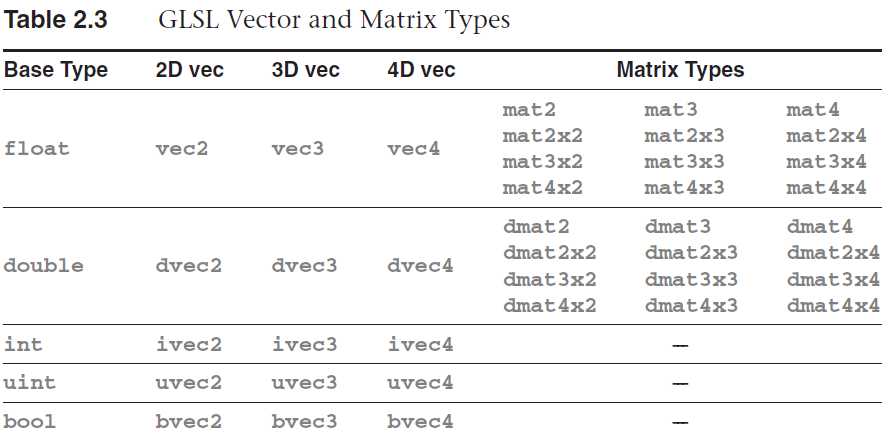
1. Shader Fundamentals

GLSL’s **main()** doesn’t take any arguments, but rather all data going into and out of a shader stage are passed using special global variables in the shade.

Basic Data Types in GLSL:



**Aggregate Types:**



matrices of **float** and **double** are available.

Variables declared with these types can be initialized similar to their scalar counterparts:

**vec3** velocity = **vec3**(0.0, 2.0, 3.0);

and converting between types is equally accessible:

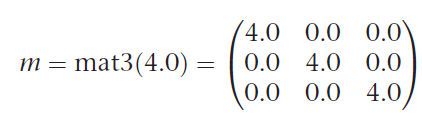
**ivec3** steps = **ivec3**(velocity);

**vec4** color;

**vec3** RGB = **vec3**(color); // now RGB only has three elements

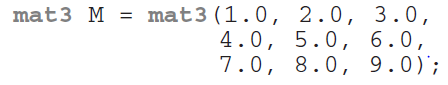
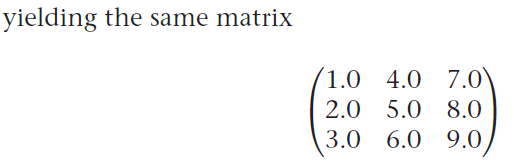
**vec3** white = **vec3**(1.0); // white = (1.0, 1.0, 1.0)

**vec4** translucent = **vec4**(white, 0.5);

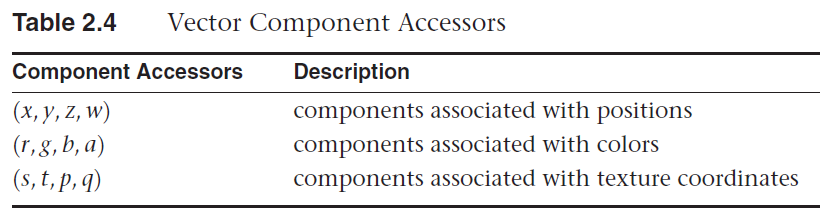


matrices are specified in column-major order, meaning the values are used to populate

columns before rows.

Vector Component Accessors



color = color.abgr; // reverse the components of a color

**vec4** zVec = m[2]; // get column 2 of the matrix

**float** yScale = m[1][1]; // or m[1].y works as well

**Arrays**

**float** coeff[3] = **float**[3](2.38, 3.14, 42.0);

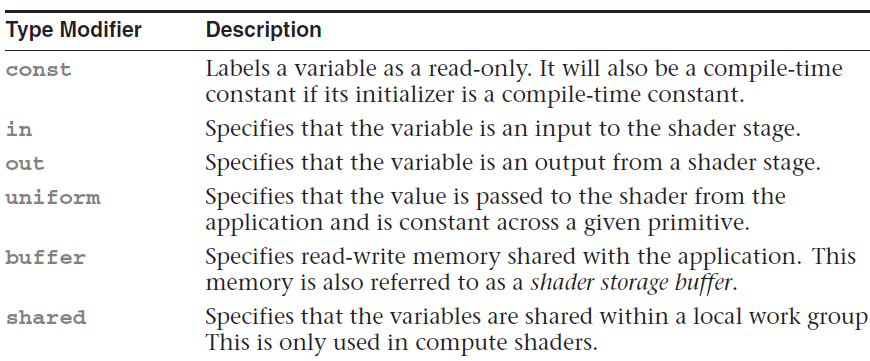
length():reporting their number of elements. a matrix’s length is the number of columns it contains.

**float** coeff[3][5]; // an array of size 3 of arrays of size 5.

coeff.length(); // this returns the constant 3

**Storage Qualifiers**

GLSL Type Modifiers



A shader cannot write to a uniform variable and change its value.

**uniform vec4** BaseColor;

To set BaseColor’s value from your application, you need to obtain the index of BaseColor in the table, which is done using the **glGetUniformLocation()** routine.

you can set the value of the uniform variable using the **glUniform\*()** or **glUniformMatrix\*()** routines.

GLint timeLoc; /\* Uniform index for variable "time" in shader \*/

GLfloat timeValue; /\* Application time \*/

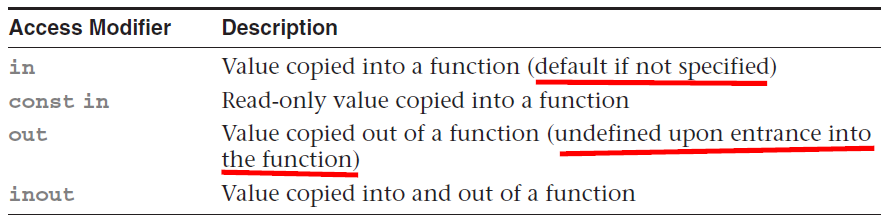
timeLoc = glGetUniformLocation(program, "time");

glUniform1f(timeLoc, timeValue);

One notable exception is that the multiplication of two vectors will result in component-wise multiplication of components; however, multiplying two matrices will result in normal matrix multiplication.

**Function Parameter Qualifiers**

parameters of functions have associated parameter qualifiers indicating if the value should be copied into, or out of, a function after execution.



**Computational Invariance**

GLSL does not guarantee that two identical computations in different shaders will result in exactly the same value. GLSL has two methods for enforcing this type of invariance between shaders, using the **invariant** or **precise** keywords.

The invariant qualifier may be applied to any shader output variable.

**invariant** gl\_Position;

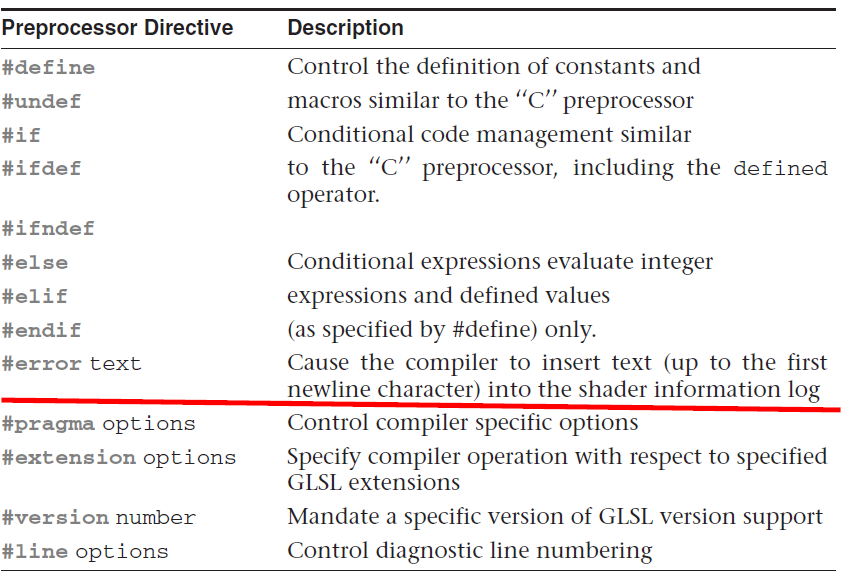
**invariant centroid out vec3** Color;

The **precise** qualifier may be applied to any computed variable or function return value.

**precise out vec3** Location;

**Shader Preprocessor**

GLSL Preprocessor Directives



**#define** LPos(n) gl\_LightSource[(n)].position

**#if** defined(NUM\_ELEMENTS) && NUM\_ELEMENTS > 3

...

**#elif** NUM\_ELEMENTS < 7

...

**#endif**

The #pragma directive provides the compiler additional information regarding how you would like your shaders compiled:

**#pragma** optimize(on) or **#pragma** optimize(off)

By default, optimization is enabled for all shaders.

**#pragma** debug(on) or **#pragma** debug(off)

The debug option enables or disables additional diagnostic output of the shader. by default, debugging is disabled for all shaders.

The GLSL preprocessor uses the #extension directive to provide instructions to the shader compiler regarding how extension availability should be handled during compilation.

**#extension** extension\_name : <directive>

**Interface Blocks**

Shader variables shared with the application or between stages can be, and sometimes must be, organized into *blocks* of variables.

**uniform** b { // "uniform" or "in" or "out" or "buffer"

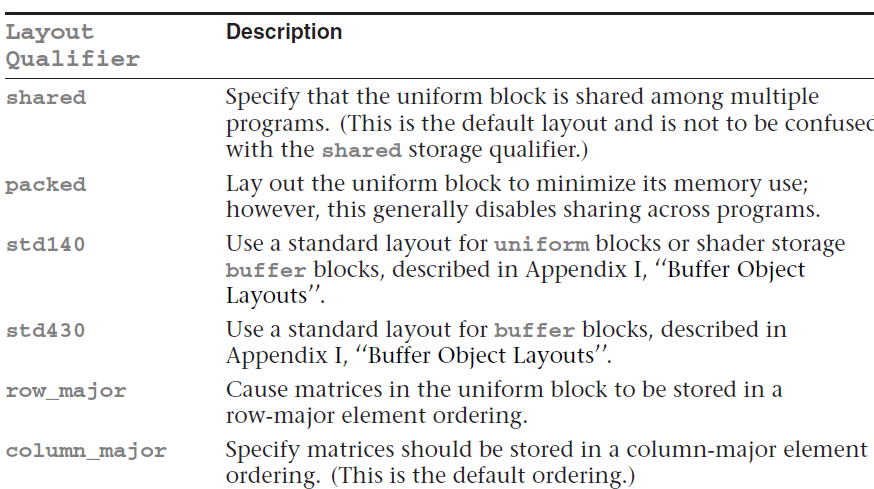
**vec4** v1; // list of variables

**bool** v2; // ...

} name; // access members as "name.v1" and "name.v2"

the block name at the beginning (b above) is used for interface matching or external identification, while the name at the end (name above) is used in the rest of the shader for accessing the members. Uniform blocks must be declared at global scope.

Layout Qualifiers for Uniform



**layout** (**shared**, row\_major) **uniform** { ... };

**Accessing Uniform Blocks:**

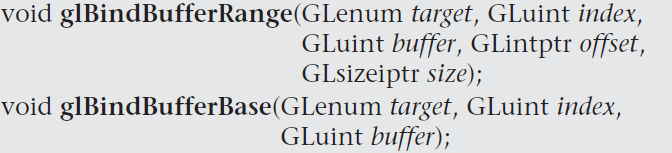
The first step in initializing the uniform variables in your uniform block is to obtain the

index of the block for a given program. Calling **glGetUniformBlockIndex()** returns an essential piece of information required to complete the mapping of uniform variables into your

application’s address space.

To initialize a buffer object to be associated with your uniform block, you’ll need to bind a buffer object to a GL\_UNIFORM\_BUFFER target using the **glBindBuffer()** routine.

Then we need to determine how large to make it to accommodate the variables in the named uniform block from our shader. To do so, we use the routine **glGetActiveUniformBlockiv()**,

After obtaining the index of the uniform block, we need to associate a buffer object with that block. The most common method for doing so is to call either **glBindBufferRange()** or, if all the buffer storage is used for the uniform block, **glBindBufferBase()**.  


1. **Drawing with OpenGL**

**Points**

OpenGL determines which pixels are covered by the point using a set of rules called *rasterization rules*. **glPointSize().** The default point size is 1.0.

**Lines, Strips, and Loops**

Individual lines are therefore represented by pairs of vertices, one for each endpoint of the

line. The closed sequence is known as a *line loop,* whereas the open sequence (one that is not closed) is known as a *line strip*. The rule for line rasterization is known as the *diamond exit rule*. OpenGL allows you to specify wider sizes for lines using the **glLineWidth()** function.

void **glLineWidth**(GLfloat *width*);

**Triangles, Strips, and Fans**

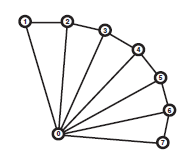
A triangle is rendered by projecting each of the three vertices into screen space and forming three edges running between the edges. A sample is considered covered if it lies on the positive side of all of the *half spaces* formed by the lines between the vertices.

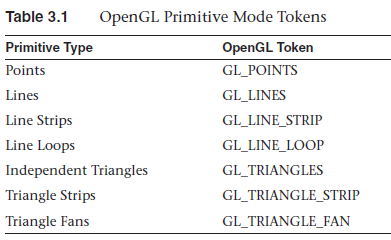
When a triangle strip is rendered, the first three vertices form the first triangle, then each

subsequent vertex forms another triangle along with the last two vertices of the previous triangle.



When rendering a triangle fan, the first vertex forms a shared point that is included in each subsequent triangle. Triangles are then formed using that shared point and the next two vertices. An arbitrarily complex *convex* polygon can be rendered as a triangle fan.





**Rendering Polygons As Points, Outlines, or Solids**

A *polygon* has two sides---front and back-. By default, both front and back *faces* are drawn in the same way. To change this, or to draw only outlines or vertices, use **glPolygonMode()**.

void **glPolygonMode**(GLenum *face*, GLenum *mode*);

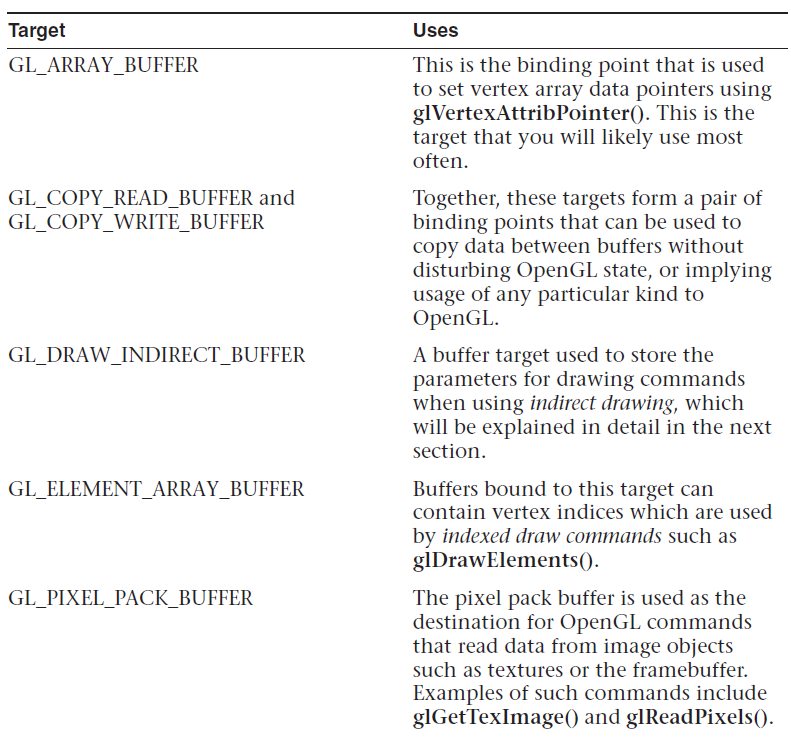
void **glFrontFace**(GLenum *mode*); Controls how front-facing polygons are determined.

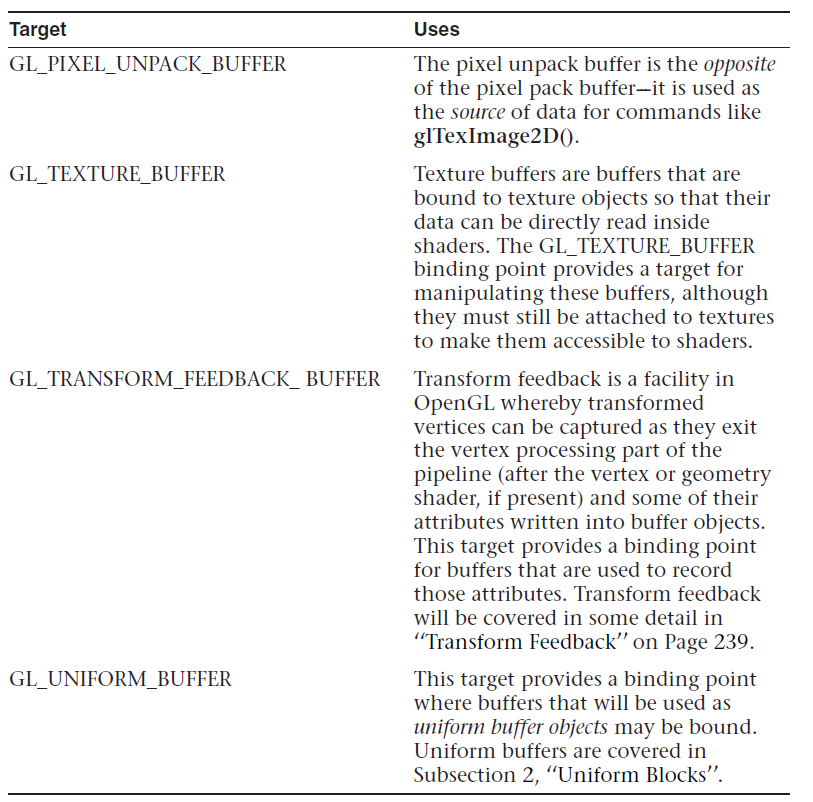
**Data in OpenGL Buffers**

void **glGenBuffers**(GLsizei *n*, GLuint *\*buffers*);

Returns *n* currently unused names for buffer objects in the array *buffers*.

The buffer objects themselves are not actually created until the name is first bound to one of the buffer binding points on the context. Buffer Binding Targets:



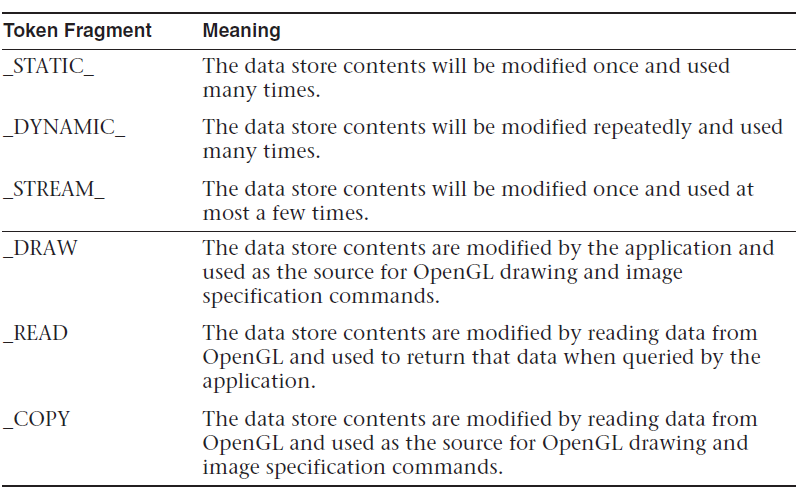


void **glBindBuffer**(GLenum *target*, GLuint *buffer*); Binds the buffer object named *buffer* to the buffer-binding point as specified by *target*.

**Getting Data into and out of Buffers**

void **glBufferData**(GLenum *target*, GLsizeiptr *size*, const GLvoid *\*data*, GLenum *usage*);

*usage* must be one of the standard usage tokens such as GL\_STATIC\_DRAW orGL\_DYNAMIC\_COPY. Buffer Usage Tokens:



**Initializing Part of a Buffer**

void **glBufferSubData**(GLenum *target*, GLintptr *offset*, GLsizeiptr *size*, const GLvoid *\*data*); Replaces a subset of a buffer object’s data store with new data. The section of the buffer object bound to *target* starting at *offset* bytes is updated with the *size* bytes of data addressed by *data*.

void **glClearBufferData**(GLenum *target*, GLenum *internalformat*, GLenum *format*, GLenum *type*, const void \* *data*);

void **glClearBufferSubData**(GLenum *target*, GLenum *internalformat*, GLintptr *offset*, GLintptr *size*, GLenum *format*, GLenum *type*, const void \* *data*);

void **glCopyBufferSubData**(GLenum *readtarget*, GLenum *writetarget*, GLintptr *readoffset*,

GLintprr *writeoffset*, GLsizeiptr *size*);

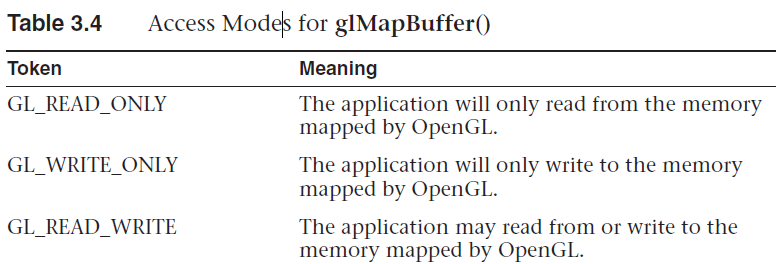
**Reading the Contents of a Buffer**

void **glGetBufferSubData**(GLenum *target*, GLintptr *offset*, GLsizeiptr *size*, GLvoid \* *data*);

**Accessing the Content of Buffers**

it’s very possible that the memory owned by OpenGL would be accessible to your application if only you had a pointer to it. Well, you can get that pointer using **glMapBuffer()**.

void \* **glMapBuffer**(GLenum *target*, GLenum *access*); When you call **glMapBuffer()**, the function returns a pointer to memory that represents the data store of the buffer object attached to *target*.



Once you are done using the data or writing data into the buffer object, you must unmap it using **glUnmapBuffer()**,

GLboolean **glUnmapBuffer**(GLenum *target*);

FILE \* f;

size\_t filesize;

// Open a file and find its size

f = fopen("data.dat", "rb");

fseek(f, 0, SEEK\_END);

filesize = ftell(f);

fseek(f, 0, SEEK\_SET);

glGenBuffers(1, &buffer);

glBindBuffer(GL\_COPY\_WRITE\_BUFFER, buffer);

glBufferData(GL\_COPY\_WRITE\_BUFFER, (GLsizei)filesize, NULL, GL\_STATIC\_DRAW);

**void** \* data = glMapBuffer(GL\_COPY\_WRITE\_BUFFER, GL\_WRITE\_ONLY);

fread(data, 1, filesize, f);// Once the buffer is mapped, the file can be read directly into

the buffer object’s data store.

glUnmapBuffer(GL\_COPY\_WRITE\_BUFFER);

fclose(f);

**glMapBufferRange()** uses flags to specify *access* more precisely.

void \* **glMapBufferRange**(GLenum *target*, GLintptr *offset*, GLsizeiptr *length*, GLbitfield *access*);

**Vertex Specification**

hook the data up to the shader.

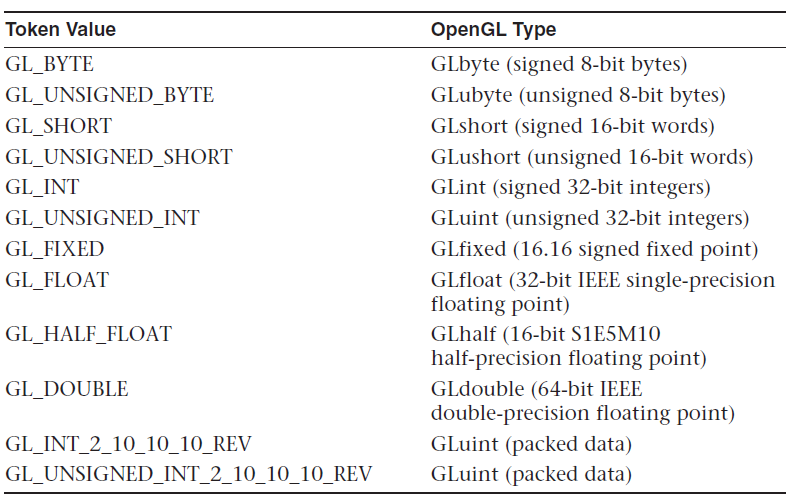
void **glVertexAttribPointer**(GLuint *index*, GLint *size*, GLenum *type*, GLboolean *normalized*, GLsizei *stride*, const GLvoid *\*pointer*);

gl.glVertexAttribPointer(0, 2, GL.*GL\_FLOAT*, **false**, 0, 0L);

Specifies where the data values for the vertex attribute with location *index* can be accessed.

*Size* represents the number of components to be updated per vertex. *stride* is the byte offset between consecutive elements in the array. If *stride* is zero, the data is assumed to be tightly packed.

Values of *Type* for **glVertexAttribPointer()**



Before OpenGL will read any data from your vertex buffers, you must enable the corresponding vertex attribute arrays with **glEnableVertexAttribArray()**.

**OpenGL Drawing Commands**

void **glDrawArrays**(GLenum *mode*, GLint *first*, GLsizei *count*);

void **glDrawElements**(GLenum *mode*, GLsizei *count*, GLenum *type*, const GLvoid *\*indices*);

void **glDrawElementsBaseVertex**(GLenum *mode*, GLsizei *count*, GLenum *type*, const GLvoid *\*indices*, GLint *basevertex*);

OpenGL has the ability to restart primitives within the same drawing command by specifying a special value, the *primitive restart index*, which is specially processed by OpenGL.

void **glPrimitiveRestartIndex**(GLuint *index*);