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

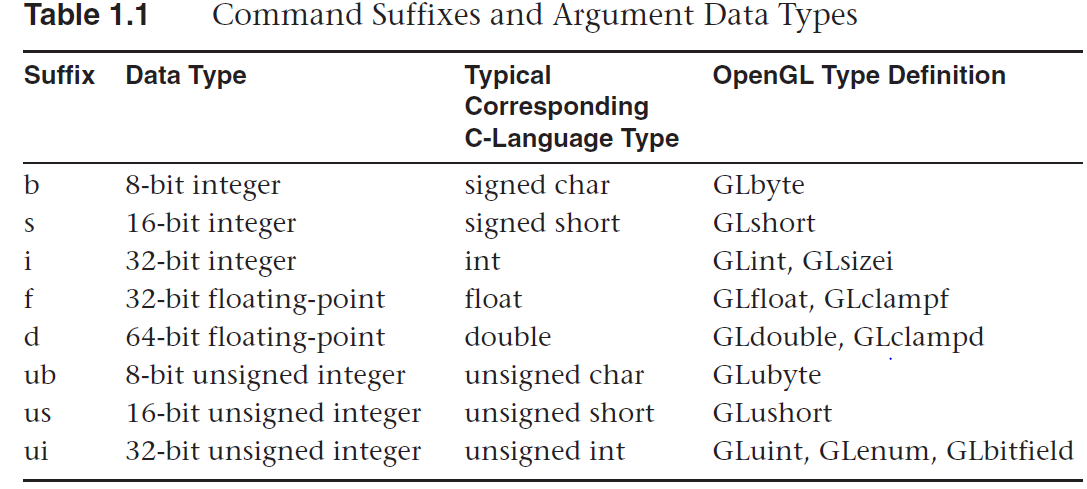
super是用在子类中，目的是访问直接父类中被屏蔽的成员，注意是直接父类（就是类之上最近的超类）。当子类方法中的局部变量或者子类的成员变量与父类成员变量同名时，也就是子类局部变量覆盖父类成员变量时，用“super.成员变量名”来引用父类成员变量。

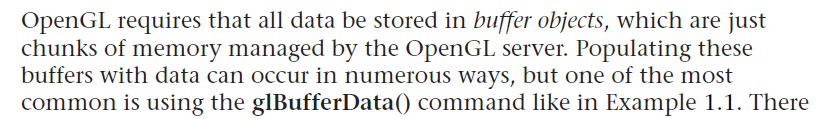
*Rendering*, which we’ve already used without defining previously, is the process by which a computer creates an image from models.

*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.

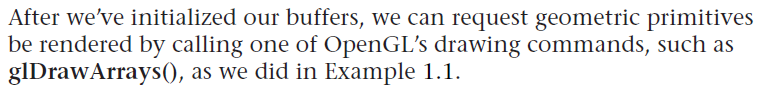
A *pixel* is the smallest visible element on your *display*. The pixels in your system are stored in a *framebuffer*, which is a chunk of memory that the graphics hardware manages, and feeds to your display device.





gl.glBufferData(GL.*GL\_ARRAY\_BUFFER*, 9 \* Buffers.*SIZEOF\_FLOAT*,

vertexBufferData, GL.*GL\_STATIC\_DRAW*);



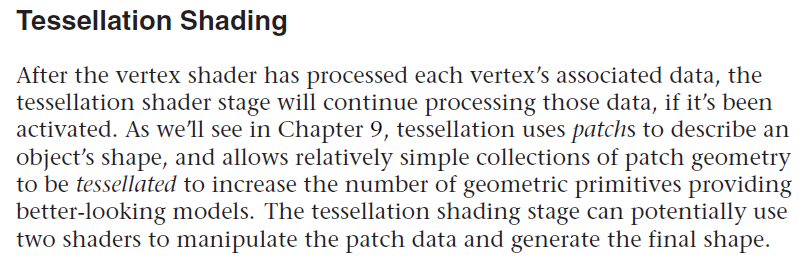
gl.glDrawArrays(GL.*GL\_TRIANGLE\_STRIP*, 0, 3);

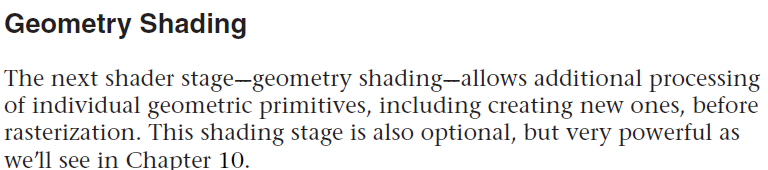


simpleProg = GLSLShaderProgram.*createAndLinkProgramResource*(gl,

"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.