OpenGL

# OpenGL是什么，能为我们做什么？

OpenGL（Open Graphics Library）是一个跨平台、跨语言的开放式图形编程接口（API）标准。它用于渲染2D和3D矢量图形，通过与计算机的GPU（图形处理器）通信，实现硬件加速的图形渲染。简单来说，它是一套规范，定义了软件如何高效地命令显卡绘制复杂图形。

**OpenGL 只定义函数名、参数和功能标准，具体实现由显卡厂商（如NVIDIA/AMD）在驱动程序中完成。**开发者无需了解不同GPU的机器指令（如NVIDIA PTX或AMD GCN），只需调用统一的OpenGL API，驱动会将其“翻译”成当前硬件的可执行代码。

OpenGL只是一个允许我们控制显卡的规范

## 现代OpenGL的着色器：

OpenGL 的着色器（Shader）是用特殊的编程语言（GLSL，OpenGL Shading Language）编写的小型程序，它们直接在GPU上运行，负责控制图形渲染管线中的关键计算步骤。你可以将着色器理解为“注入到渲染流程中的自定义算法”，用于实现顶点变换、光照、颜色、纹理等效果的精细控制。

### 为什么需要着色器？

在早期OpenGL（固定管线时代），渲染效果（如光照、雾效）是由预定义的固定函数控制的，开发者只能通过参数调整，灵活性极低。而现代OpenGL（可编程管线）将核心计算过程开放给开发者，通过编写着色器，你可以：

* 完全掌控渲染的数学与物理过程
* 实现任意复杂的视觉效果（如卡通渲染、水波纹、动态光影）
* 最大化发挥GPU并行计算能力

现代OpenGL必须使用着色器：固定管线已在OpenGL 3.1+中被废弃。一句话概括：着色器是GPU上运行的微型程序，通过编写GLSL代码，开发者能像指挥物理定律一样控制虚拟世界的视觉规则——从移动顶点到点亮像素，皆由你定义。

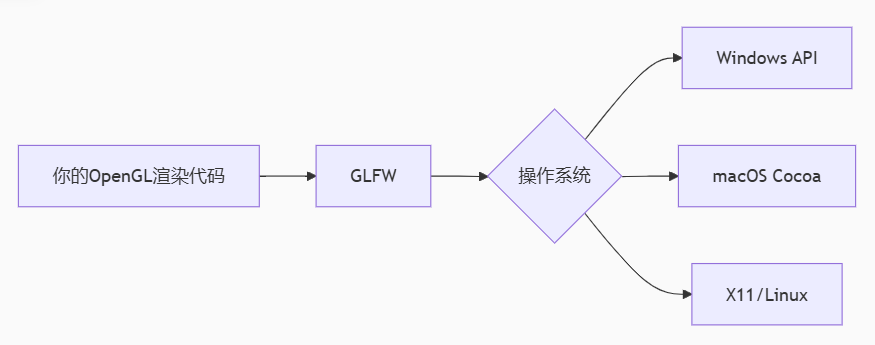
# OpenGL代码创建一个窗口并在窗口画一个三角形（GLFW库）

## GLFW是什么？

GLFW（OpenGL Framework）是一个轻量级、开源的跨平台库，专门用于创建和管理OpenGL渲染上下文所需的窗口、输入和事件系统。你可以将它理解为OpenGL的“启动器”和“控制台”——它不参与图形渲染本身，但为OpenGL提供了一个运行环境。

OpenGL本身只关注图形渲染，但渲染需要基础环境：

* ❌ 无法创建窗口或处理操作系统交互
* ❌ 无法读取键盘鼠标输入
* ❌ 无法管理窗口事件（如缩放、关闭）





GLFW 本质上是位于 OpenGL 之上的一层抽象 API，但它不参与图形渲染本身，而是专注于为 OpenGL（或 Vulkan/OpenGL ES）提供运行环境的基础设施。



类比：



GLFW 是 OpenGL 的“服务层”，它提供渲染所需的窗口环境，而 OpenGL 专注图形计算。两者协同工作，但各司其职。

1. 底层：操作系统与硬件

* 操作系统（Windows/macOS/Linux）：提供原生图形接口（如WGL/GLX/Cocoa）。
* 显卡驱动：实现 OpenGL 规范，将 API 调用转为 GPU 指令。
* GPU：执行实际渲染计算。

2. 中层：OpenGL API

图形渲染的核心层（负责绘制三角形、纹理贴图、着色器等）。

3. 上层：GLFW

角色：OpenGL 的“服务提供者”，解决以下问题：

* ✅ 窗口创建：生成一个支持 OpenGL 渲染的窗口。
* ✅ 上下文管理：绑定 OpenGL 与当前窗口。
* ✅ 输入处理：转换键盘/鼠标动作为可读事件。
* ✅ 事件循环：管理窗口缩放、关闭等系统事件。

## 下载GLFW

<https://www.glfw.org/>

自己不想编译看源码就下载官方编译好的文件。

## VS2022链接GLFW

### 静态链接和动态链接的区别

* ‌静态链接‌：在编译阶段完成，库代码（如.a或.lib文件）被直接合并到可执行文件中。‌‌‌‌
* ‌动态链接‌：在程序运行时加载共享库（如.so或.dll），仅记录库的引用信息。‌‌

‌文件与内存特性‌：

静态链接：可执行文件较大（包含所有库代码）。内存占用高（多程序重复加载相同库）

动态链接：可执行文件较小（仅含引用信息）。内存占用低（多程序共享同一库副本）。

维护与更新：

静态链接：库更新需重新编译程序，维护成本高。

动态链接：库可独立更新，无需重新编译程序，但可能引发版本冲突（如“DLL地狱”）。

适用场景：

静态链接：嵌入式系统、独立分发程序或对启动速度要求高的场景。

动态链接：大型软件、插件化架构或需频繁更新库的场景。??

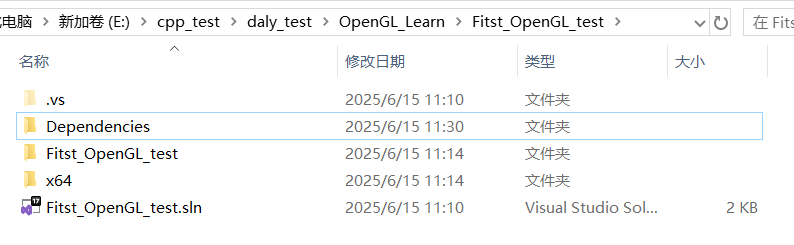
性能差异

静态链接启动更快（无库加载开销），但现代操作系统对动态库有缓存机制，实际差异可能不明显。

### 我们采用静态链接

1.拷贝GLFW官网的Documentation的示例文件，然后粘贴到自己的vs项目中。

2.打开自己VS项目的方案文件路径，创建一个Dependencies文件夹：

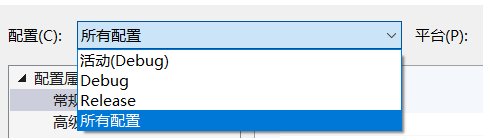


在Dependeices里面创建一个文件夹GLFW，把之前下载的压缩包中的include和lib-vc2015粘贴到GLFW文件夹内：

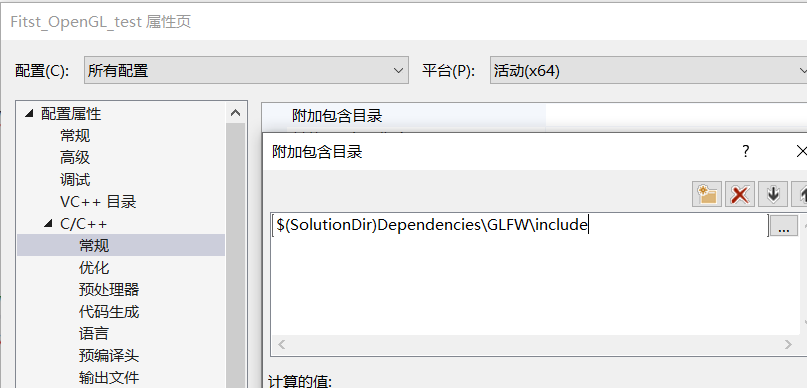


3.让自己的项目的“包含目录”包含include文件夹：

右键项目，属性，先选择所有配置：

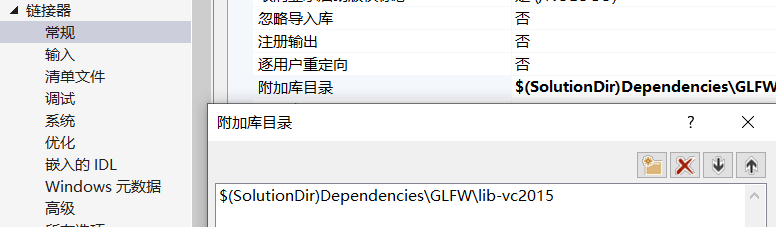


在C/C++，常规的附加包含目录输入：$(SolutionDir) 代表着项目的根目录，然后输入include的相对路径，这样可以在其他电脑上也能使用这个路径：

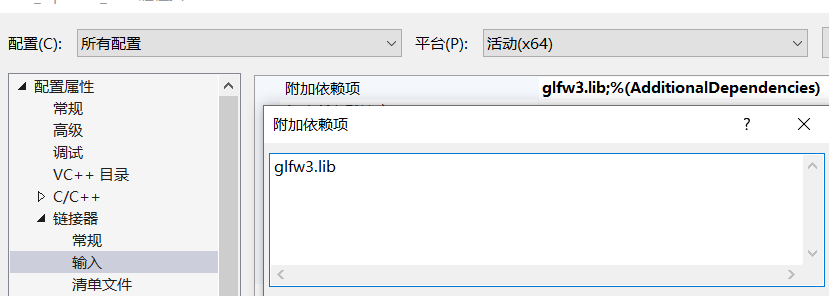


4.让自己项目的“库目录”包含lib文件：

在链接器，常规，附加库目录：包含lib-vc2015：

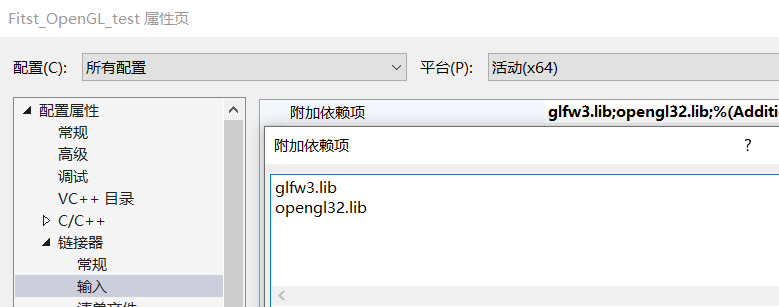


5.在链接器，输入，附加依赖项中添加： glfw3.lib

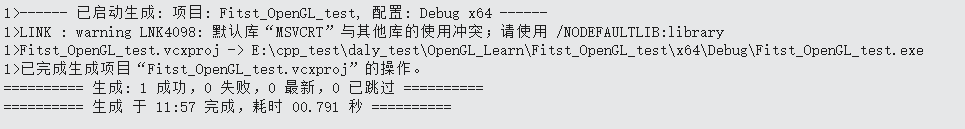


6.此时按f5，发现编译成功但是链接失败。是因为我们在实例中调用了OpenGL的函数，但是没链接OpenGL：

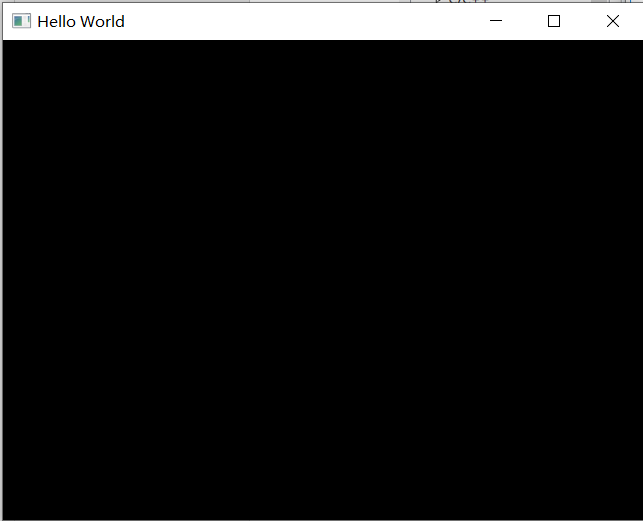
再次在链接器，输入，附加依赖项中添加： opengl32.lib



7.右键项目生成，可以看到，项目已经生成成功了：

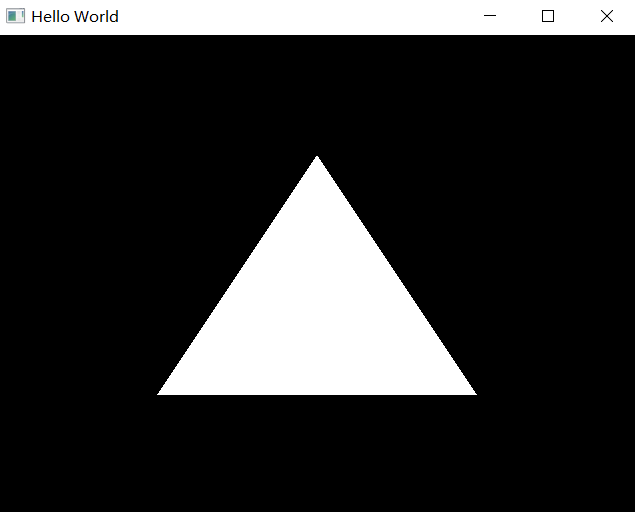
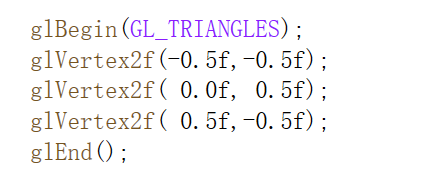


8.点击运行，可以看到生成了一个黑窗口：



## 在之前的黑窗口上创建一个三角形（GLFW库实现）

在代码中添加：



就会在之前画的黑色窗口中显示一个三角形，坐标为代码所示。

原点（0, 0）位于屏幕的中心。

* x 坐标的范围是从 -1 到 1，表示屏幕的水平方向。-1 表示屏幕的最左边，1 表示屏幕的最右边。
* y 坐标的范围是从 -1 到 1，表示屏幕的垂直方向。-1 表示屏幕的最下边，1 表示屏幕的最上边。

# GLEW库

## GLEW介绍

### GLEW产生的背景

GLEW（OpenGL Extension Wrangler Library）是用于处理OpenGL扩展的库。

OpenGL 是一个跨语言、跨平台的编程接口，用于渲染 2D、3D 矢量图形。随着图形硬件的发展，出现了许多新的 OpenGL 扩展功能。这些扩展功能允许开发者利用新的图形硬件特性来实现更复杂的图形效果。然而，不同厂商的显卡支持的 OpenGL 扩展可能不同，并且不同版本的 OpenGL 也有差异。GLEW帮助开发者处理这些复杂的扩展情况。

### GLEW工作原理

GLEW 会在程序运行时自动检测当前使用的 OpenGL 驱动程序所支持的扩展。它提供了一组易用的函数和宏定义，使得开发者可以方便地访问这些扩展功能。例如，对于每个 OpenGL 扩展函数，GLEW 会生成一个函数指针，并通过检测机制来确定该函数是否可用。如果可用，开发者就可以像使用普通的 OpenGL 函数一样来调用这些扩展函数。

### GLEW优点

* 自动检测 ：减少了开发者手动检查 OpenGL 扩展支持情况的工作量。在没有 GLEW 的情况下，开发者需要自己编写代码来判断显卡是否支持某个特定的扩展，这既繁琐又容易出错。而 GLEW 可以自动完成这个检测过程，提高了开发效率。
* 跨平台支持 ：在 Windows、Linux 和 macOS 等多个操作系统平台上都可以使用 GLEW。这使得开发者能够更容易地编写跨平台的图形应用程序，不用担心不同平台下 OpenGL 扩展处理的不同之处。
* 易于使用 ：它提供了一个简单易懂的接口。对于大多数扩展函数，开发者只需要包含 GLEW 的头文件并正确初始化，就可以立即使用这些函数。

### 典型应用场景

* 在游戏开发中，游戏通常需要利用各种高级的图形特效，这些特效往往依赖于 OpenGL 扩展。GLEW 可以帮助游戏开发者快速地集成这些扩展功能，如阴影贴图、高级纹理等功能，以提高游戏的视觉质量。
* 对于图形渲染软件，如 3D 建模工具和科学可视化软件，GLEW 也非常重要。这些软件需要充分利用现代显卡的硬件加速能力来渲染复杂的图形场景，GLEW 使得软件能够方便地访问各种 OpenGL 扩展来实现高效、高质量的图形渲染。

### GLEW 的核心功能

GLEW的核心使命是简化现代OpenGL函数指针的加载过程。

|  |  |
| --- | --- |
| 功能 | 说明 |
| 自动加载函数指针 | 一次性加载OpenGL 1.0到最新版（如4.6）的所有标准函数 |
| 扩展支持检测 | 检查硬件是否支持特定扩展（如 if(GLEW\_ARB\_geometry\_shader4) ） |
| 跨平台兼容 | 支持Windows/macOS/Linux（注意：macOS需额外处理） |
| 轻量级 | 仅头文件 glew.h + 源文件 glew.c（或预编译库），无复杂依赖 |
| 错误处理 | 明确返回初始化状态，避免崩溃 |

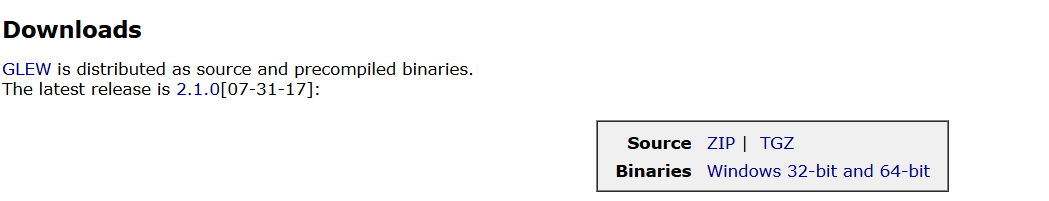
GLEW 是OpenGL开发的“钥匙扣”：

它把散落在显卡驱动中的函数指针（钥匙）统一收集管理，让你无需手动寻找即可直接调用 gl\* 系列函数。虽然现代替代品（如GLAD）更灵活，但理解GLEW仍能加深你对OpenGL底层机制的认识。

## 下载GLEW

访问GLEW官网：<https://glew.sourceforge.net/>进行下载。

如果不想看源码，直接下载二进制Binaries版本：



## VS2022链接GLEW

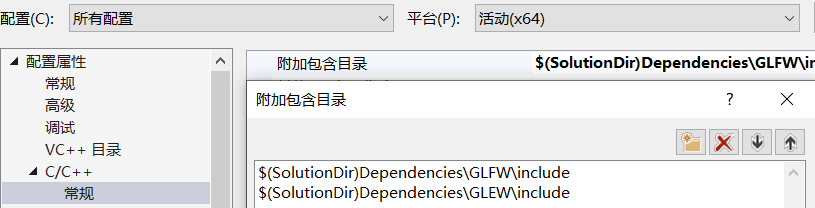
### 和之前链接GLFW一样，我们采用静态链接

1.把下载好的文件解压，也放在Dependencies文件夹内，改名为GLEW：



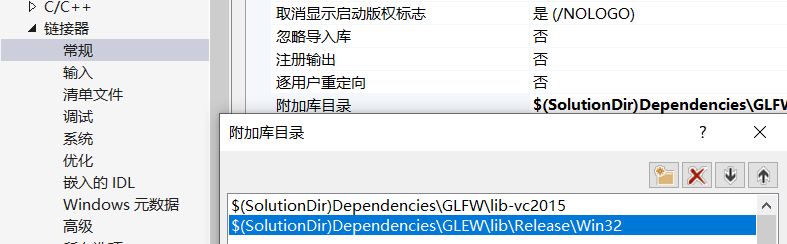
doc文件夹内包含了需要的各种文档。

2.同样在C/C++，常规，附加包含目录中加入include的相对路径：

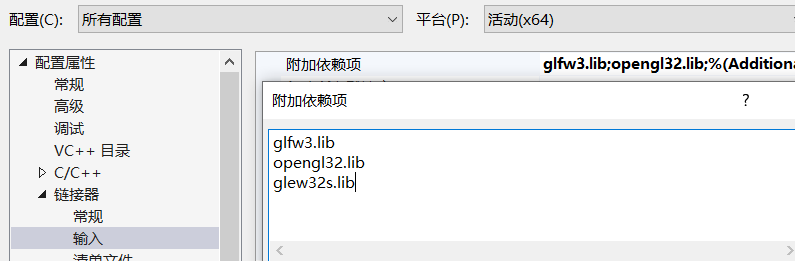


在链接器，常规，附加库目录中加入glew32s.lib的相对路径：

注意：32位在Win32文件夹，64位在x86文件夹



3.在链接器，输入中加入glew32s.lib：



至此就配置完成了。

### 使用现代OpenGL的函数

1.添加头文件：

为什么要GL/？因为我们的包含目录到glew.h之间还有一个GL文件夹。

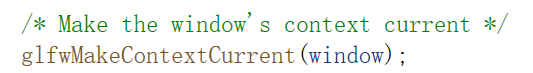
2.初始化glew：

但是在我们运行之后，发现报错。这是因为头文件glew.h要放在glfw3.h之前。所以我们要把glew.h放在所有OpenGL头文件之前。

3.这样右键生成，会发现编译没错了。但是有链接错误。在C/C++，预处理器，预处理器定义中添加：GLEW\_STATIC



自此点击生成成功生成。

注意：使用要先初始化OpenGL上下文，也就是该函数要放在后面。

我们使用GLEW就可以访问所有的OpenGL函数了，可以使用任何版本。我们浏览GL/glew.h文件中的内容，就会发现它里面全是OpenGL的函数指针。

### 打印OpenGL当前版本

在初始化上OpenGL下文之后就可以打印OpenGL当前版本：

std::cout << glGetString(GL\_VERSION) << std::endl;





# 现代OpenGL绘制三角形

电脑上的渲染通常是从三角形开始的‌。我们绘制三角形需要**顶点缓冲区**和**着色器**。

## 顶点缓冲区

顶点缓冲区 (Vertex Buffer)，是指一块存储在图形处理单元内存 (GPU Memory / VRAM) 中的特殊内存区域，专门用来存放构成渲染对象几何形状的顶点数据。（只是存数据的内存缓冲区？）

核心作用就是：高效、批量地将顶点数据从你的应用程序（CPU 端）传输并存储到 GPU 上，供 GPU 在渲染时快速访问。

它是OpenGL的内存缓存但是是在GPU上。

## 着色器

着色器运行在GPU，告知GPU如何解释顶点缓冲区中的数据以及如何把它放到我们屏幕上。告诉 GPU 如何“绘制”或“着色”物体的各个部分。

*我们用C/C++代码写的东西都是在CPU上运行，CPU 配置好渲染环境和数据，并通过绘制指令通知 GPU：“现在开始用当前设置执行渲染！”。一旦在CPU上发出了绘制指令，GPU 按固定管线流程调用着色器进行绘制，显示在屏幕。*

## 状态机

状态机（State Machine）用于描述一个系统如何基于当前状态和输入事件，按规则转移到新状态并执行相应动作的数学模型。一种将系统行为建模为 状态 + 事件 → 转移 + 动作 的范式。

OpenGL自身是一个巨大的状态机(State Machine)，一系列的变量描述OpenGL此刻应当如何运行。渲染行为由一组全局状态变量控制，API调用本质上是修改这些状态或触发基于当前状态的渲染操作。

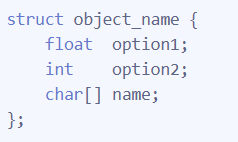
OpenGL的状态通常被称为OpenGL上下文(Context)。我们通常使用如下途径去更改OpenGL状态：设置选项，操作缓冲。最后，我们使用当前OpenGL上下文来渲染。

假设当我们想告诉OpenGL去画线段而不是三角形的时候，我们通过改变一些上下文变量来改变OpenGL状态，从而告诉OpenGL如何去绘图。一旦我们改变了OpenGL的状态为绘制线段，下一个绘制命令就会画出线段而不是三角形。

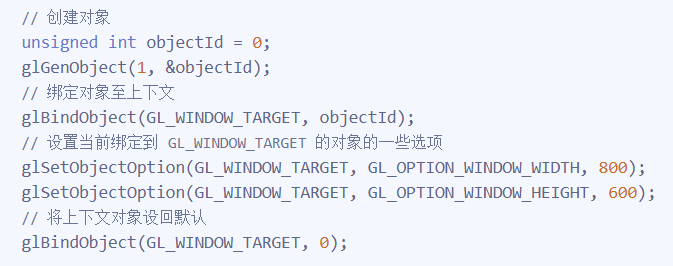
当使用OpenGL的时候，我们会遇到一些状态设置函数(State-changing Function)，这类函数将会改变上下文。以及状态使用函数(State-using Function)，这类函数会根据当前OpenGL的状态执行一些操作。只要你记住OpenGL本质上是个大状态机，就能更容易理解它的大部分特性。

## 对象

在OpenGL中一个对象是指一些选项的集合，它代表OpenGL状态的一个子集。比如，我们可以用一个对象来代表绘图窗口的设置，之后我们就可以设置它的大小、支持的颜色位数等等。可以把对象看做一个C风格的结构体(Struct)：

在这里我们使用的是C/C++自带的类型，但是其实可以使用OpenGL的类型，如“GLint”，使用OpenGL的类型的好处是保证了在各平台中每一种类型的大小都是统一的。你也可以使用其它的定宽类型(Fixed-width Type)来实现这一点。顶点缓冲区也算是对象。

下面这段操作展示了以后使用OpenGL时常见的工作流程

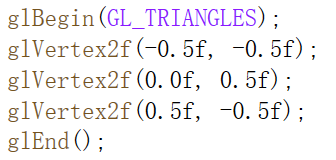


我们首先创建一个对象，然后用一个id保存它的引用（实际数据被储存在后台）。然后我们将对象绑定至上下文的目标位置（例子中窗口对象目标的位置被定义成GL\_WINDOW\_TARGET）。接下来我们设置窗口的选项。最后我们将目标位置的对象id设回0，解绑这个对象。设置的选项将被保存在objectId所引用的对象中，一旦我们重新绑定这个对象到GL\_WINDOW\_TARGET位置，这些选项就会重新生效。

使用对象的一个好处是在程序中，我们不止可以定义一个对象，并设置它们的选项，每个对象都可以是不同的设置。在我们执行一个使用OpenGL状态的操作的时候，只需要绑定含有需要的设置的对象即可。比如说我们有一些作为3D模型数据（一栋房子或一个人物）的容器对象，在我们想绘制其中任何一个模型的时候，只需绑定一个包含对应模型数据的对象就可以了（当然，我们需要先创建并设置对象的选项）。拥有数个这样的对象允许我们指定多个模型，在想画其中任何一个的时候，直接将对应的对象绑定上去，便不需要再重复设置选项了。

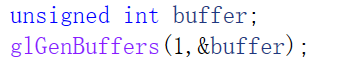
## 使用代码创建顶点缓冲区

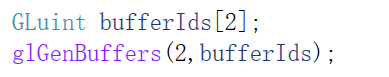
之前的代码使用的是传统OpenGL绘制三角形：

现在我们使用现代OpenGL进行绘制：

**创建顶点缓冲区**，顶点缓冲区包含顶点的数据。我们把它传到OpenGL的VRAM然后发出一个DrawCall指令：“根据缓冲区画出图形”，然后才能绘制出图形：

1.使用glGenBuffers()函数初始化顶点缓冲区：

第一个参数是要初始化的顶点缓冲区的数量，第二个参数是存储生成的缓冲区对象id（名称）。

当调用glGenBuffers时，OpenGL会为每个新生成的缓冲区对象分配一个唯一的id（名字，类似于标识符），这些名称会被存储在由第二个参数指向的内存空间中。如果想要生成2个顶点缓冲区，那么就用数组存储：

2.选择（绑定）顶点缓冲区：

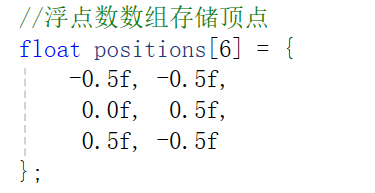
第一个参数是一个枚举值，用于指定缓冲区对象存储的目标，不同的目标值决定了缓冲区对象在 OpenGL 中的用途。第二个参数为缓冲区id（名字）。

GL\_ARRAY\_BUFFER 表示当前绑定的缓冲区对象将用于存储顶点属性数据（如顶点位置、颜色、法线、纹理坐标等）。

其他的枚举值如：

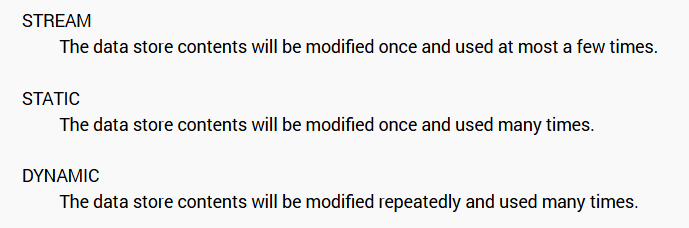
* GL\_ELEMENT\_ARRAY\_BUFFER：用于存储索引数据（index buffer），这些数据用于指示顶点的绘制顺序。
* GL\_UNIFORM\_BUFFER：用于存储统一变量（uniform variables），这些变量在着色器中是全局的。
* GL\_FRAMEBUFFER：用于绑定帧缓冲区对象（Framebuffer Object），用于自定义渲染目标。

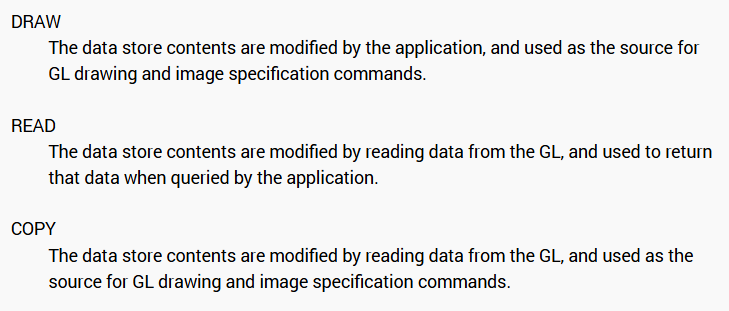
3.定义数据并且把数据放入顶点缓冲区：





这里，我们可以在<https://docs.gl> 中查询该函数：第四个参数usage，分别代表什么：





这里选择GL\_STATIC\_DRAW表示：数据存储内容将被修改一次并多次使用。自此传给OpenGL的数据就准备完毕了。

## OpenGL如何使用拿到的这些数据？

我们还未学习着色器，告知OpenGL这些数据是如何布局（使用和着色器紧密相连的函数glVertexAttributePointer()）。所以暂时使用glDrawArrays看看效果:

第二个参数：指定启用的数组中的起始索引。第三个参数：指定要渲染的索引数。

也可以使用glDrawElements（有索引缓冲区时使用）

疑问：glDrawArrays()函数是如何拿到之前创建的顶点缓冲区数据的？

我们之前使用了glBindBuffer(GL\_ARRAY\_BUFFER, buffer);

在这里绑定了buffer，表明了我们要绘制的是哪一个。

这就是状态机模式。类似于Photoshop的图册，我选择图层1，在图层1进行绘制，只会影响到图层1。如果没选择，它就不会绘制图层。所以在使用之前要选择绑定。

## 顶点属性和内存布局之glVertexAttribPointer()函数

之前代码的positions数组，只是表示了顶点的坐标，其实顶点属性不仅有这些。**顶点属性是用来描述顶点的几何和非几何信息的集合。**在创建和绑定顶点缓冲区对象之后（解绑顶点缓冲区之前），就可以告诉OpenGL,我们缓冲区的布局是什么。使用glVertexAttribPointer()函数（定义通用顶点属性数据的数组）：

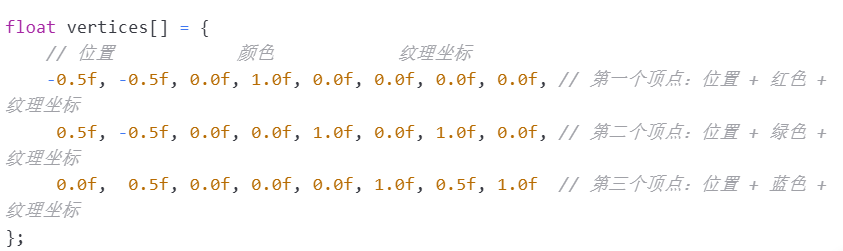
****

该函数的6个参数分别代表：

index：

指定要修改的通用顶点属性的索引。着色器要读取所有信息都是通过index索引。顶点属性索引的作用是将应用程序中定义的顶点数据与着色器程序中的输入变量关联起来。每个顶点属性（如位置、颜色、纹理坐标等）都需要一个唯一的索引来标识，这样 OpenGL 就知道如何将数据从应用程序传递到着色器中的相应变量。这里就定义0这个index为位置顶点属性索引。用来将应用程序中的顶点数据与着色器中的位置属性关联起来的。

如果还不理解，可以这样理解：我们一个float positions数组中包含了3种属性，顶点位置坐标（位置属性），颜色属性，纹理坐标属性（当然还可能包含更多属性）：

我们需要能说：我想让我的位置属性在index0处，颜色属性在index1处，纹理坐标属性在index2处。**这样当我开始从着色器和显卡读取数据时可以简单的引用它们。**如：给我index0，就能直接找到位置属性。当我们使用glVertexAttribPointer()函数定义颜色属性时，就可以把index设置为1。在着色器寻找颜色属性时就是用index1。

size

指定每个通用顶点属性的组件数。初始值为4。组件数：一个二维向量（如位置）有2个组件（x 和 y），一个三维向量（如法向量）有3个组件，一个四维向量（如颜色，包含 r、g、b、a）有4个组件。这里的2表示每个顶点属性有2个组件。

type

指定顶点数据的类型。

Normalized

让顶点数据的类型统一化（规范化）处理。如颜色我们是0-255表示rgb，但是要被规范为0-1才能在着色器中使用。可以让OpenGL替你完成这些操作。如果数据类型是整数类型（如GL\_INT或GL\_UNSIGNED\_INT），并且设置为GL\_TRUE，OpenGL会将整数数据映射到 [-1,1]（对于有符号整数）或 [0,1]（对于无符号整数）范围。对于浮点数类型（如 GL\_FLOAT），这个参数会被忽略。在这里，GL\_FALSE 表示不进行归一化。

stride

步长参数，表示连续顶点属性之间的字节偏移量。其实就是每个顶点之间的字节数(每个顶点的字节大小)。因为我们的float数组只包含了2维坐标信息，所以两个顶点之间的距离就为sizeof(float)\*2。如果是index中举的例子，那么距离就为sizeof(float)\*8。

pointer

偏移量参数，表示顶点属性在缓冲区中的起始位置。初始值为 0。其实就是每个属性的第一个组件在缓冲区的第几个字节偏移量。如代码中的0表示顶点属性从缓冲区的开头开始。如果是index中举的例子，那么颜色属性的就为：(void\*)(3 \* sizeof(float))，纹理坐标属性就为：(void\*)(6 \* sizeof(float))。要记得转换为指针类型。因为参数要求是 const void\* 类型，所以不能直接使用整数，需要将整数偏移量转换为 void\* 类型。这可以通过强制类型转换来实现，例如 (void\*)。后面如果确定了自己的参数，可以定义一些宏来进行表示，而不是直接数字表示。加强代码可读性。

**启动顶点属性：**

查询<https://docs.gl>中说：若要启用和禁用泛型顶点属性数组，请使用索引调用 glEnableVertexAttribArray 和 glDisableVertexAttribArray。

启动索引0：

自此我们就告知了OpenGL缓冲区布局是什么，接下来只需要着色器就能在屏幕显示三角形了。