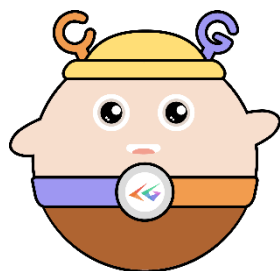




華中科技大學



# 计算机图形学课程

**实验：搭建 OpenGL 环境**

**&&绘制窗口**

---

## 目录

1 搭建 OpenGL 环境.....	1
1.1 GLFW .....	1
1.1.1 构建 GLFW .....	1
1.1.2 CMake .....	2
1.1.3 编译 .....	6
1.1.4 我们的第一个工程.....	7
1.1.5 链接 .....	7
1.2 GLAD.....	10
2 绘制一个窗口 .....	13
2.1 GLAD.....	15
2.2 视口 .....	15
2.3 渲染 .....	16
2.4 最后 .....	17
2.5 更改背景色 .....	18

---

# 1 搭建 OpenGL 环境

在正式搭建环境之前，我们先来介绍一下读完下面的部分你会了解些什么。

- 如何构建 GLFW
- 如何使用 CMake 工具
- 建立我们的第一个工程
- 如何编译我们的库，并将其链接至工程中
- 配置 GLAD 库

## 1.1 GLFW

GLFW 是一个 OpenGL 的 C 语言库，它提供了渲染物体所需要的最低限度的接口，它允许用户创建 OpenGL 上下文，定义窗口参数以及用户的输入。

【简单来说，GLFW 是对之前固定管线常用的 GLUT 的一种改进】

### 1.1.1 构建 GLFW

GLFW 可以从它官方网站的下载页上获取。我们需要下载源代码包。

【如果你要使用预编译的二进制版本的话，请下载 32 位的版本而不是 64 位的（除非你清楚你在做什么）】

下载源码包之后，将其解压并打开。我们只需要里面的这些内容：

- 编译生成的库
- include 文件夹

从源代码编译库可以保证生成的库是兼容你的操作系统和 CPU 的，而预编译的二进制文件可能会出现兼容问题（甚至有时候没提供支持你系统的文件）。提供源代码所产生的一个问题在于不是每个人都用相同的 IDE 开发程序，因而

提供的工程/解决方案文件可能和一些人的 IDE 不兼容。所以人们只能从.c/.cpp 和.h/.hpp 文件来自己建立工程/解决方案，这是一项枯燥的工作。但因此也诞生了一个叫做 CMake 的工具。

### 1.1.2 CMake

CMake 是一个工程文件生成工具。用户可以使用预定义好的 CMake 脚本，根据自己的选择（像是 Visual Studio, Code::Blocks, Eclipse）生成不同 IDE 的工程文件。这允许我们从 GLFW 源码里创建一个 Visual Studio 2017 工程文件，之后进行编译。首先，我们需要从这里下载安装 CMake。我选择的是 Win32 安装程序。

Latest Release (3.12.1)  
The release was packaged with CPack which is included as part of the release. The .sh files are self extracting gzipped tar files. To install a .sh file, run it with /bin/sh and follow the directions. The OS-machine.tar.gz files are gzipped tar files of the install tree. The OS-machine.tar.Z files are compressed tar files of the install tree. The tar file distributions can be untared in any directory. They are prefixed by the version of CMake. For example, the Linux-x86\_64 tar file is all under the directory cmake-Linux-x86\_64. This prefix can be removed as long as the share, bin, man and doc directories are moved relative to each other. To build the source distributions, unpack them with zip or tar and follow the instructions in Readme.txt at the top of the source tree. See also the [CMake 3.12 Release Notes](#). Source distributions:

Platform	Files
Unix/Linux Source (has \n line feeds)	<a href="#">cmake-3.12.1.tar.gz</a>
	<a href="#">cmake-3.12.1.tar.Z</a>
Windows Source (has \r\n line feeds)	<a href="#">cmake-3.12.1.zip</a>

Binary distributions:

Platform	Files
Windows win64-x64 Installer: <b>Installer tool has changed. Uninstall CMake 3.4 or lower first!</b>	<a href="#">cmake-3.12.1-win64-x64.msi</a>
Windows win64-x64 ZIP	<a href="#">cmake-3.12.1-win64-x64.zip</a>
Windows win32-x86 Installer: <b>Installer tool has changed. Uninstall CMake 3.4 or lower first!</b>	<a href="#">cmake-3.12.1-win32-x86.msi</a>
<u>Windows win32-x86 ZIP</u>	<a href="#">cmake-3.12.1-win32-x86.zip</a>
Mac OS X 10.7 or later	<a href="#">cmake-3.12.1-Darwin-x86_64.dmg</a>
	<a href="#">cmake-3.12.1-Darwin-x86_64.tar.gz</a>
Linux x86_64	<a href="#">cmake-3.12.1-Linux-x86_64.sh</a>
	<a href="#">cmake-3.12.1-Linux-x86_64.tar.gz</a>

图 1-2-1 cmake 版本

当 CMake 安装成功后，你可以选择从命令行或者 GUI 启动 CMake，由于我们不想让事情变得太过复杂，我们选择用 GUI。CMake 需要一个源代码目录和一个存放编译结果的目标文件目录。源代码目录我们选择 GLFW 的源代码的根目录，然后我们新建一个 build 文件夹，选中作为目标目录。

【CMake 的 GUI 在 bin 目录下】

名称	修改日期	类型	大小
 cmake.exe	2018/5/17 10:24	应用程序	7,365 KB
 cmake-gui.exe	2018/5/17 10:24	应用程序	17,979 KB
 cmcldeps.exe	2018/5/17 10:23	应用程序	792 KB
 cpack.exe	2018/5/17 10:24	应用程序	7,071 KB
 ctest.exe	2018/5/17 10:24	应用程序	7,814 KB

图 1-2-2 cmake 可执行文件

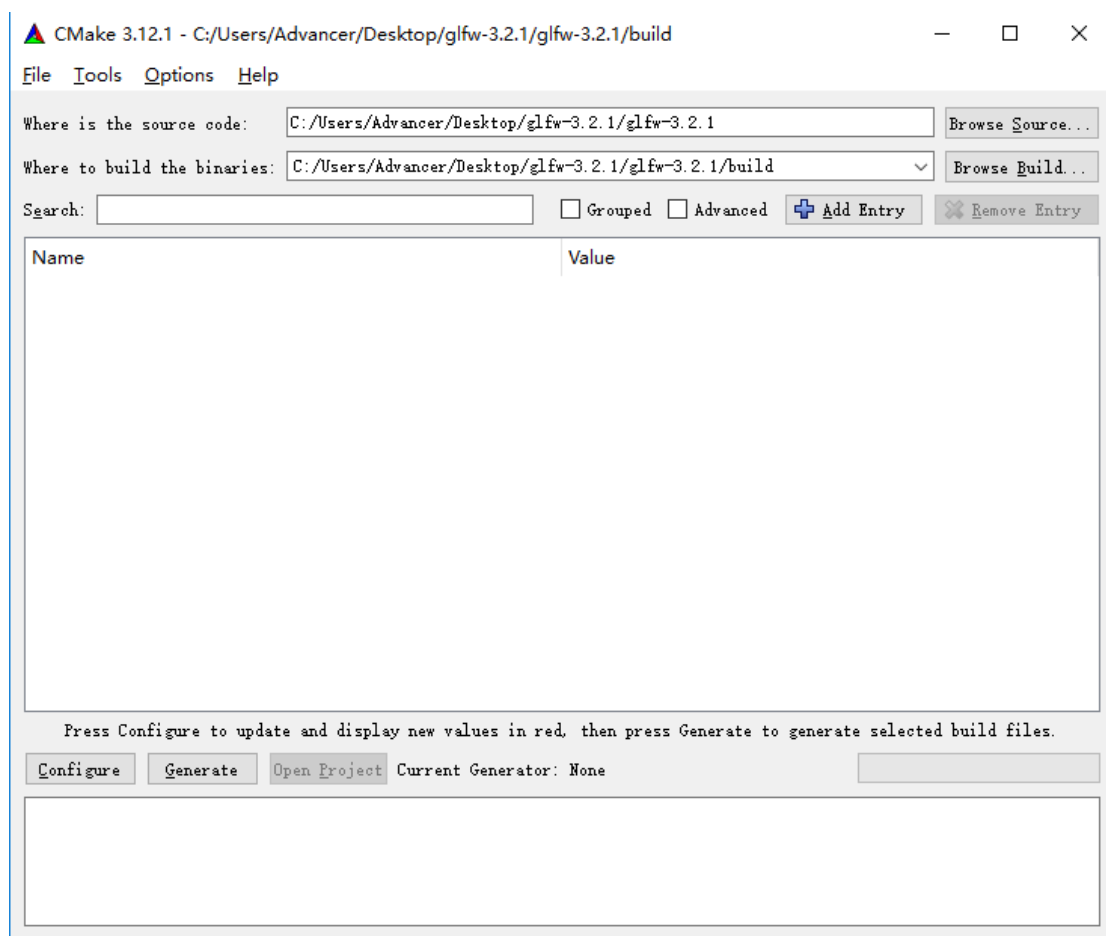


图 1-2-3 cmake 界面

在设置完源代码目录和目标目录之后，点击 **Configure**(设置)按钮，让 CMake 读取设置和源代码。我们接下来需要选择工程的生成器，由于我们使用的是 Visual Studio 2017，我们选择 **Visual Studio 15** 选项（因为 Visual Studio 2017 的内部版本号是 15）。

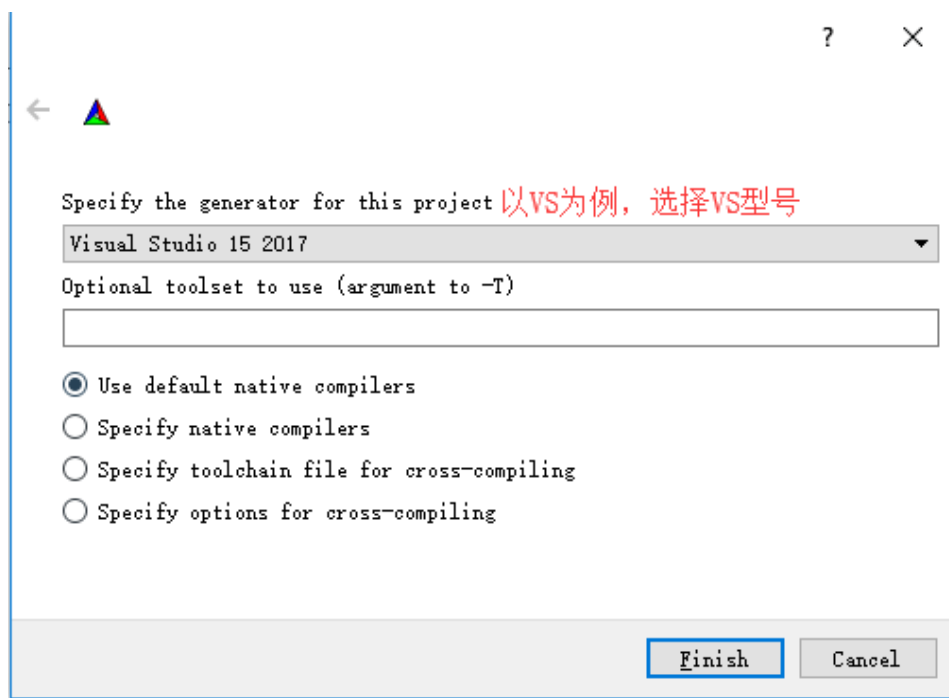


图 1-2-4 cmake 选择对应 IDE 版本

CMake 会显示可选的编译选项用来配置最终生成的库。这里我们使用默认设置，并再次点击 **Configure**(设置)按钮保存设置。保存之后，点击 **Generate**(生成)按钮，生成的工程文件会在你的 **build** 文件夹中。

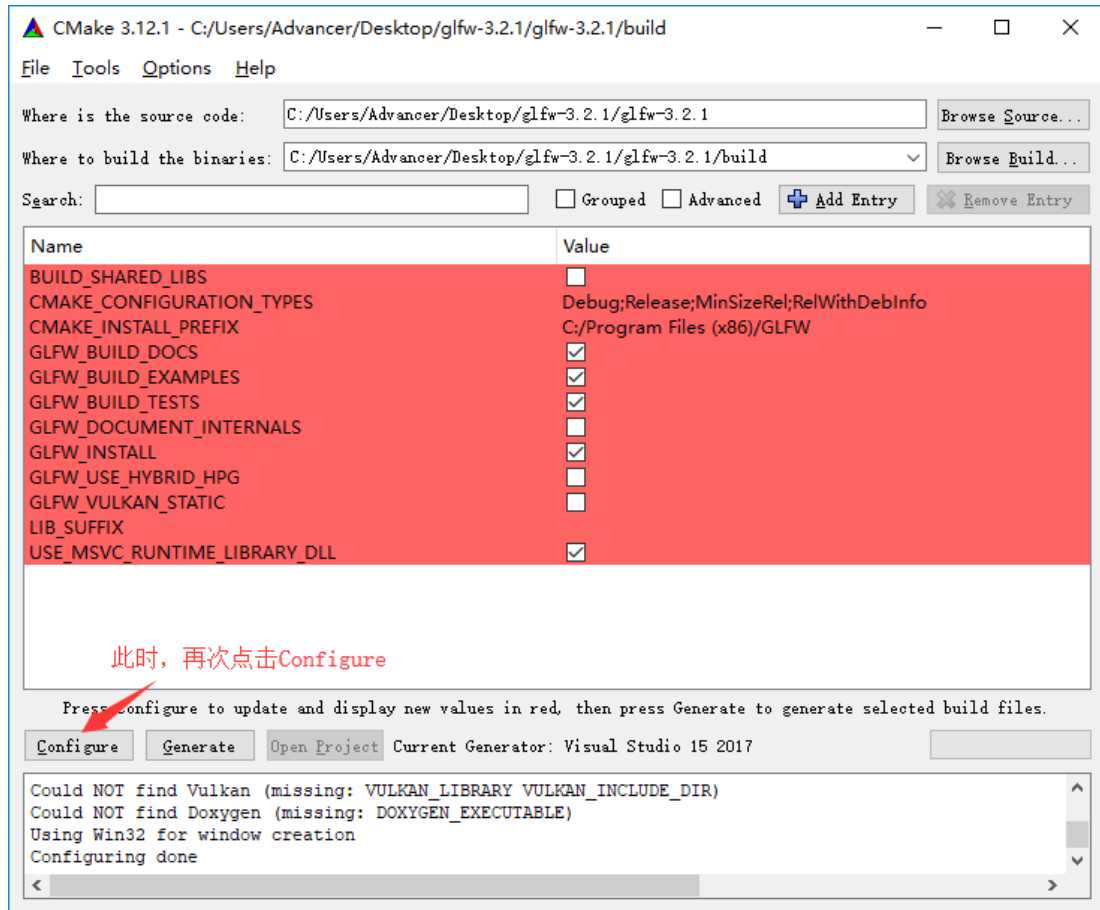


图 1-2-5 cmake 设置生成

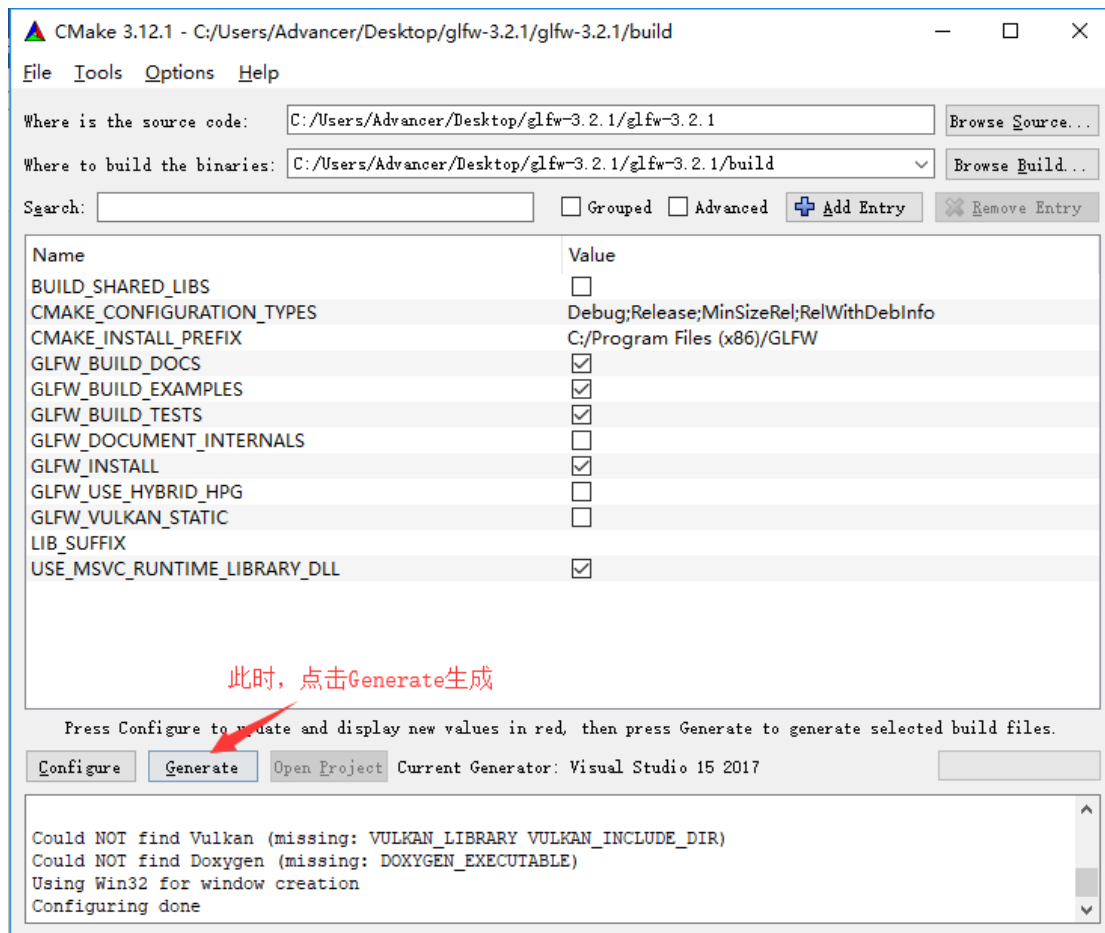


图 1-2-6 cmake 设置生成

### 1.1.3 编译

在 build 文件夹里可以找到 GLFW.sln 文件，用 Visual Studio 2017 打开。因为 CMake 已经配置好了项目，所以我们直接点击 Build Solution(生成解决方案)按钮，然后编译的库 glfw3.lib（注意我们用的是第 3 版）就会出现在 src/Debug 文件夹内。




glfw-3.2.1 > glfw-3.2.1 > build > src > Debug			
名称	修改日期	类型	大小
 glfw3.lib	2018/8/22 16:59	对象文件库	576 KB

图 1-3-1 编译生成的 glfw 库

库生成完毕之后，我们需要让 IDE 知道库和头文件的位置。有两种方法：

1. 找到 IDE 或者编译器的/lib 和/include 文件夹，添加 GLFW 的 include 文件夹里的文件到 IDE 的/include 文件夹里去。用类似的方法，将 glfw3.lib 添加到/lib 文件夹里去。虽然这样能工作，但这不是推荐的方式，因为这样会让你很难去管理库和 include 文件，而且重新安装 IDE 或编译器可能会导致这些文件丢失。
2. 推荐的方式是建立一个新的目录包含所有的第三方库文件和头文件，并且在你的 IDE 或编译器中指定这些文件夹。我个人会使用一个单独的文件夹，里面包含 Libs 和 Include 文件夹，在这里存放 OpenGL 工程用到的所有第三方库和头文件。这样我的所有第三方库都在同一个位置（并且可以共享至多台电脑）。然而这要求你每次新建一个工程时都需要告诉 IDE/编译器在哪能找到这些目录。

完成上面步骤后，我们就可以使用 GLFW 创建我们的第一个 OpenGL 工程了！

### 1.1.4 我们的第一个工程

首先，打开 Visual Studio，创建一个新的项目。如果 VS 提供了多个选项，选择 Visual C++，然后选择 Empty Project(空项目)（别忘了给你的项目起一个合适的名字）。现在我们终于有一个空的工作空间了，开始创建我们第一个 OpenGL 程序吧！

### 1.1.5 链接

为了使我们的程序使用 GLFW，我们需要把 GLFW 库链接(Link)进工程。这可以通过在链接器的设置里指定我们要使用 glfw3.lib 来完成，但是由于我们将第三方库放在另外的目录中，我们的工程还不知道在哪寻找这个文件。于是我们首先需要将我们放第三方库的目录添加进设置。

要添加这些目录（需要 VS 搜索库和 include 文件的地方），我们首先进入 Project Properties(工程属性，在解决方案窗口里右键项目)，然后选择 VC++ Directories(VC++ 目录)选项卡（如下图）。在下面的两栏(Include Directories 包含目录和 Library Directories 库目录)添加目录：

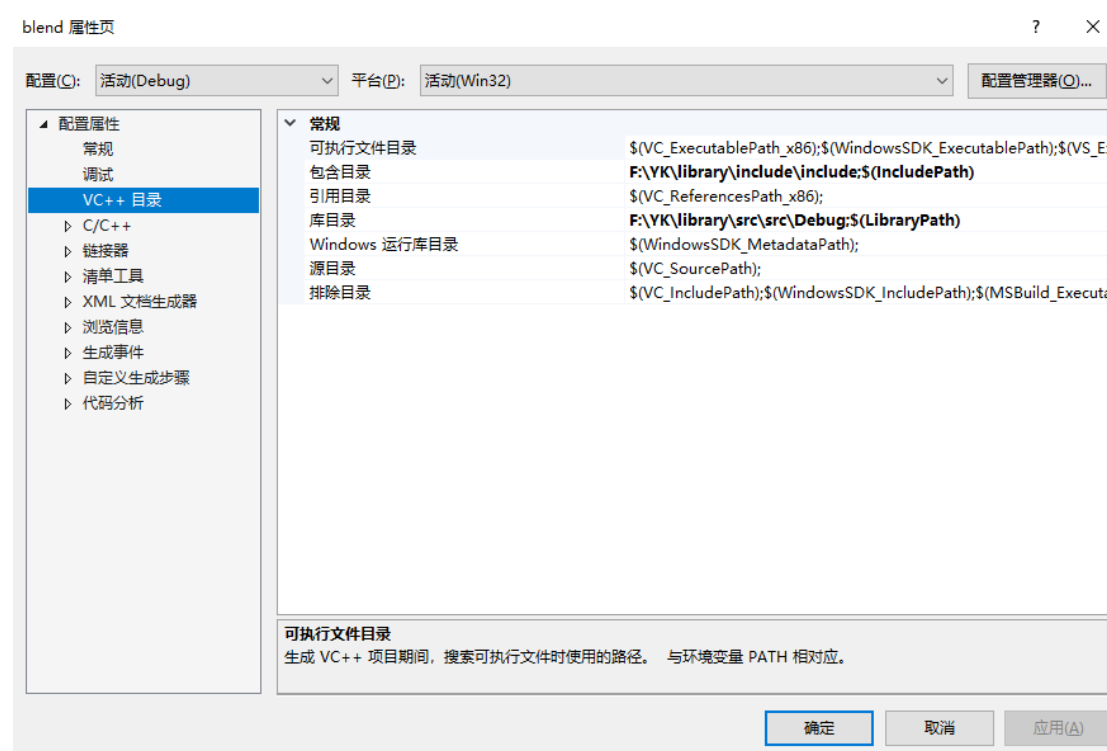


图 1-5-1 vs 配置界面

这里你可以把自己的目录加进去，让工程知道到哪去搜索。你需要手动把目录加在后面，也可以点击需要的位置字符串，选择选项，之后会出现类似下面这幅图的界面，图是选择 Include Directories(包含目录)时的界面：

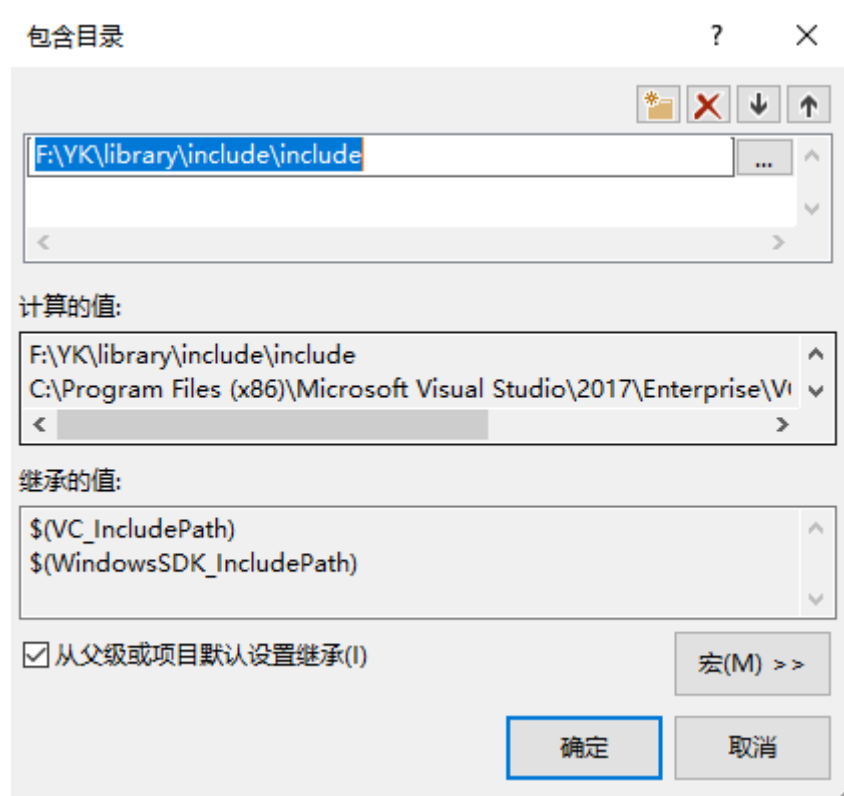


图 1-5-2 vs 配置界面

这里可以添加任意多个目录，IDE 会从这些目录里寻找头文件。所以只要你将 GLFW 的 Include 文件夹加进路径中，你就可以使用<GLFW/..>来引用头文件。库目录也是同样。

现在 VS 可以找到所需的所有文件了。最后需要在 Linker(链接器)选项卡里的 Input(输入)选项卡里添加 glfw3.lib 这个文件：

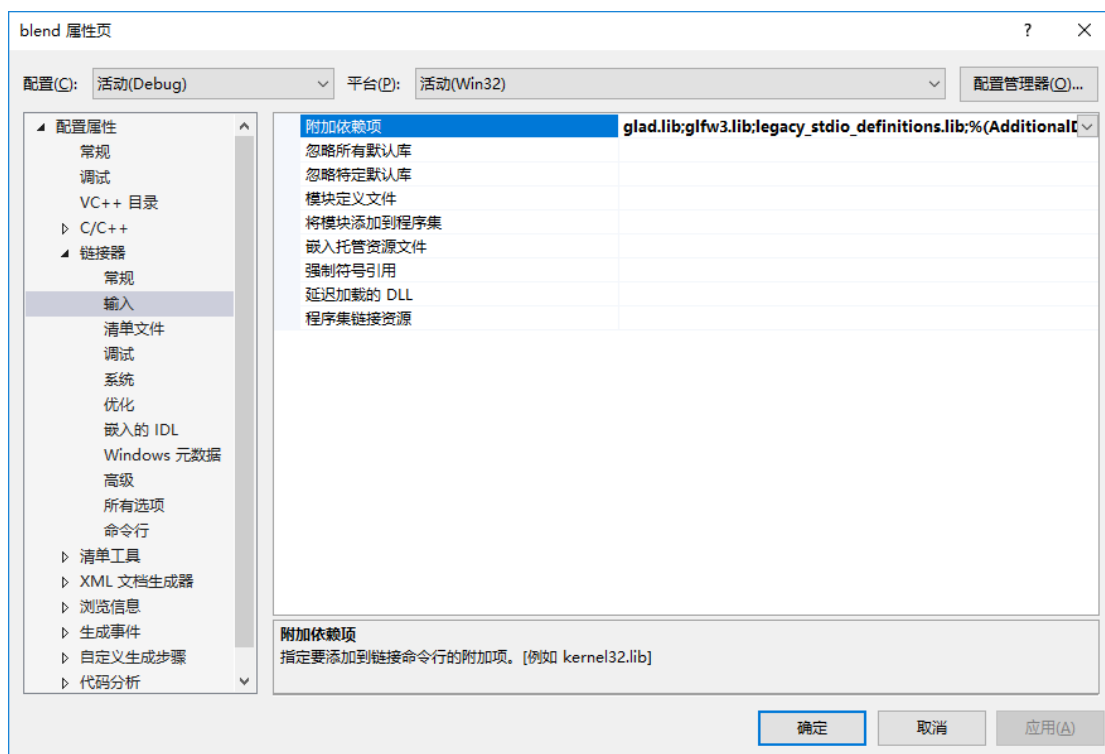


图 1-5-3 vs 配置界面

要链接一个库我们必须告诉链接器它的文件名。库的文件名是 `glfw3.lib`，我们把它加到 `Additional Dependencies`(附加依赖项)字段中(手动或者使用选项都可以)。这样 `GLFW` 在编译的时候就会被链接进来了。

**【GLFW 的安装与配置就到此为止】**

## 1.2 GLAD

到这里还没有结束，我们仍然还有一件事要做。因为 `OpenGL` 只是一个标准/规范，具体的实现是由驱动开发商针对特定显卡实现的。由于 `OpenGL` 驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。所以任务就落在了开发者身上，开发者需要在运行时获取函数地址并将其保存在一个函数指针中供以后使用。取得地址的方法因平台而异，代码非常复杂，而且很繁琐，我们还需要对每个可能使用的函数都要重复这个过程。幸运的是，有些库能简化此过程，其中 `GLAD` 是目前最新，也是最流行的库。

接下来我们来介绍一下如何配置 GLAD 库，GLAD 是一个开源的库，它能解决我们上面提到的那个繁琐的问题。GLAD 的配置与大多数的开源库有些许的不同，GLAD 使用了一个在线服务。在这里我们能够告诉 GLAD 需要定义的 OpenGL 版本，并且根据这个版本加载所有相关的 OpenGL 函数。

打开 GLAD 的在线服务，将语言(Language)设置为 C/C++，在 API 选项中，选择 3.3 以上的 OpenGL(gl)版本（我们的教程中将使用 3.3 版本，但更新的版本也能正常工作）。之后将模式(Profile)设置为 Core，并且保证生成加载器(Generate a loader)的选项是选中的。现在可以先（暂时）忽略拓展(Extensions)中的内容。都选择完之后，点击生成(Generate)按钮来生成库文件。

The screenshot displays the GLAD online service configuration interface. At the top, there are two dropdown menus: 'C/C++' for the language and 'OpenGL' for the API. Below these, the 'API' section contains four dropdowns for 'gl' (set to 'Version 3.3'), 'gles1' (set to 'None'), 'gles2' (set to 'None'), and 'glsc2' (set to 'None'). To the right, the 'Profile' dropdown is set to 'Core'. The 'Extensions' section features a search bar and a list of extensions, with 'GL\_3DFX\_tbuffer' currently selected. Below the extensions list are three buttons: 'ADD LIST', 'ADD ALL', and 'REMOVE ALL'. The 'Options' section at the bottom includes three checkboxes: 'Generate a loader' (checked), 'Omit KHR' (unchecked), and 'Local Files' (unchecked). A 'GENERATE' button is located at the bottom right of the interface.

图 2-1-1 GLAD 在线服务界面

GLAD 现在应该提供给你一个 zip 压缩文件，包含两个头文件目录，和一个 glad.c 文件。将两个头文件目录（glad 和 KHR）复制到你的 Include 文件夹

---

中（或者增加一个额外的项目指向这些目录），并添加 `glad.c` 文件到你的工程中。

经过前面的这些步骤之后，你就应该可以将以下的指令加到你的文件顶部了：

```
#include <glad/glad.h>
```

---

## 2 绘制一个窗口

在正式绘制窗口之前，我们先来介绍一下读完下面的部分你会了解些什么。

- 如何使用 GLAD
- 渲染的过程包括最后的释放资源
- 如何调整背景色

让我们先来试试能不能让 GLFW 正常工作。首先，新建一个.cpp 文件，然后把下面的代码粘贴到该文件的最前面

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

接下来我们创建 main 函数，在这个函数中我们将会实例化 GLFW 窗口：

```
int main()
{
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    //glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

    return 0;
}
```

首先，我们在 main 函数中调用 glfwInit 函数来初始化 GLFW，然后我们可以使用 glfwWindowHint 函数来配置 GLFW。glfwWindowHint 函数的第一个参数代表选项的名称，我们可以从很多以 GLFW\_开头的枚举值中选择；第二个参数接受一个整形，用来设置这个选项的值。如果你现在编译你的 cpp 文件会得到大量的 undefined reference (未定义的引用)错误，也就是说你并未顺利地链接 GLFW 库。

---

由于我们是基于 OpenGL3.3 来讨论的，我们将主版本号(Major)和次版本号(Minor)都设为 3。我们同样明确告诉 GLFW 我们使用的是核心模式(Core-profile)。明确告诉 GLFW 我们需要使用核心模式意味着我们只能使用 OpenGL 功能的一个子集（没有我们已不再需要的向后兼容特性）。如果使用的是 Mac OS X 系统，你还需要加下面这行代码到你的初始化代码中这些配置才能起作用（将上面的代码解除注释）：

```
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

当然我们也可以看到 GLFW 当前使用的版本号，GLFW 提供了很多函数可以让我们了解到当前使用的版本。

```
int Major, Minor, Rev;
glfwGetVersion(&Major, &Minor, &Rev);
printf("GLFW %d.%d.%d initialized\n", Major, Minor, Rev);
```

接下来我们创建一个窗口对象，这个窗口对象存放了所有和窗口相关的数据，而且会被 GLFW 的其他函数频繁地用到。

```
GLFWwindow* window = glfwCreateWindow(800, 600, "window", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

glfwCreateWindow 函数需要窗口的宽和高作为它的前两个参数。第三个参数表示这个窗口的名称（标题），这里我们使用"window"，当然你也可以使用你喜欢的名称。最后两个参数我们暂时忽略。这个函数将会返回一个 GLFWwindow 对象，我们会在其它的 GLFW 操作中使用到。创建完窗口我们就可以通知 GLFW 将我们窗口的上下文设置为当前线程的主上下文了。



---

## 2.1 GLAD

在之前我们已经提到，GLAD 是用来管理 OpenGL 的函数指针的，所以在调用任何 OpenGL 的函数之前我们需要初始化 GLAD。

```
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

我们给 GLAD 传入了用来加载系统相关的 OpenGL 函数指针地址的函数。GLFW 给我们的是 glfwGetProcAddress，它根据我们编译的系统定义了正确的函数。

## 2.2 视口

在我们开始渲染之前还有一件重要的事情要做，我们必须告诉 OpenGL 渲染窗口的尺寸大小，即视口(Viewport)，这样 OpenGL 才能知道怎样根据窗口大小显示数据和坐标。我们可以通过调用 glViewport 函数来设置窗口的维度(Dimension)：

```
glViewport(0, 0, 800, 600);
```

glViewport 函数前两个参数控制窗口左下角的位置。第三个和第四个参数控制渲染窗口的宽度和高度（像素）。

我们实际上也可以将视口的维度设置为比 GLFW 的维度小，这样子之后所有的 OpenGL 渲染将会在一个更小的窗口中显示，这样的话我们也可以将一些其它元素显示在 OpenGL 视口之外。

然而，当用户改变窗口的大小的时候，视口也应该被调整。我们可以对窗口注册一个回调函数(Callback Function)，它会在每次窗口大小被调整的时候被调用。这个回调函数的原型如下：

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
```

这个帧缓冲大小函数需要一个 `GLFWwindow` 作为它的第一个参数，以及两个整数表示窗口的新维度。每当窗口改变大小，`GLFW` 会调用这个函数并填充相应的参数供你处理。

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}
```

我们还需要注册这个函数，告诉 `GLFW` 我们希望每当窗口调整大小的时候调用这个函数：

```
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

当窗口被第一次显示的时候 `framebuffer_size_callback` 也会被调用。对于视网膜(Retina)显示屏，`width` 和 `height` 都会明显比原输入值更高一点。

## 2.3 渲染

我们希望程序在我们主动关闭它之前不断绘制图像并能够接受用户输入。因此，我们需要在程序中添加一个 `while` 循环，我们可以把它称之为渲染循环(Render Loop)，它能在我们让 `GLFW` 退出前一直保持运行。下面几行的代码就实现了一个简单的渲染循环：

```
while(!glfwWindowShouldClose(window))
{
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

- `glfwWindowShouldClose` 函数在我们每次循环的开始前检查一次 GLFW 是否被要求退出，如果是的话该函数返回 `true` 然后渲染循环便结束了，之后我们就可以关闭应用程序了。
- `glfwPollEvents` 函数检查有没有触发什么事件（比如键盘输入、鼠标移动等）、更新窗口状态，并调用对应的回调函数（可以通过回调方法手动设置）。
- `glfwSwapBuffers` 函数会交换颜色缓冲（它是一个储存着 GLFW 窗口每一个像素颜色值的大缓冲），它在这一迭代中被用来绘制，并且将会作为输出显示在屏幕上。
- **【双缓冲(Double Buffer)】**
- 应用程序使用单缓冲绘图时可能会存在图像闪烁的问题。这是因为生成的图像不是一下子被绘制出来的，而是按照从左到右，由上而下逐像素地绘制而成的。最终图像不是在瞬间显示给用户，而是通过一步一步生成的，这会导致渲染的结果很不真实。为了规避这些问题，我们应用双缓冲渲染窗口应用程序。**前**缓冲保存着最终输出的图像，它会在屏幕上显示；而所有的渲染指令都会在后缓冲上绘制。当所有的渲染指令执行完毕后，我们**交换(Swap)**前缓冲和后缓冲，这样图像就立即呈显出来，之前提到的不真实感就消除了。

## 2.4 最后

当渲染循环结束后我们需要正确释放/删除之前的分配的所有资源。我们可以在 `main` 函数的最后调用 `glfwTerminate` 函数来完成。

```
glfwTerminate();  
return 0;
```

这样便能清理所有的资源并正确地退出应用程序。现在你可以尝试编译并运行你的应用程序了，如果没做错的话，你将会看到如下的输出：

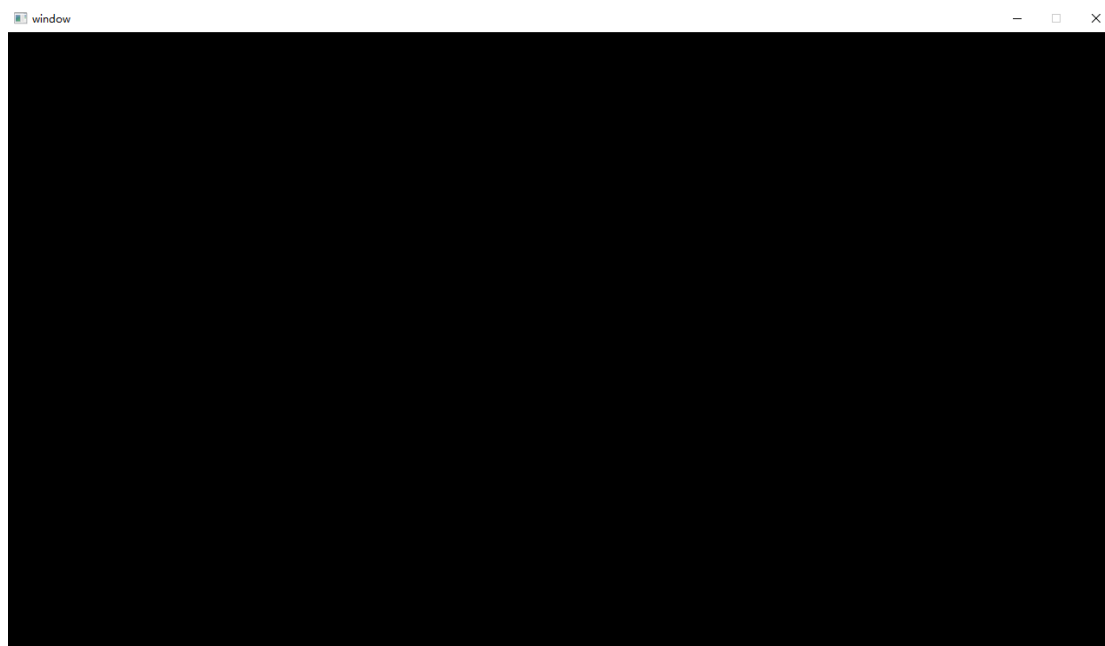


图 4-1 窗口图（黑色）

如果你看见了这样的一个黑色窗口，恭喜你，那就对了~，如果没有得到正确的结果，或者不知道怎样把东西放在一起，那么请参考源码：

## 2.5 更改背景色

我们要把所有的渲染(Rendering)操作放到渲染循环中，因为我们想让这些渲染指令在每次渲染循环迭代的时候都能被执行。代码将会是这样的：

```
// 渲染循环
while(!glfwWindowShouldClose(window))
{
    // 输入
    processInput(window);

    // 渲染指令
    ...

    // 检查并调用事件，交换缓冲
    glfwPollEvents();
}
```

```
glfwSwapBuffers(window);  
}
```

为了测试一切都正常工作，我们使用一个自定义的颜色清空屏幕。在每个新的渲染迭代开始的时候我们总是希望清屏，否则我们仍能看见上一次迭代的渲染结果（这可能是你想要的效果，但通常这不是）。我们可以通过调用 `glClear` 函数来清空屏幕的颜色缓冲，它接受一个缓冲位(Buffer Bit)来指定要清空的缓冲，可能的缓冲位有 `GL_COLOR_BUFFER_BIT`，`GL_DEPTH_BUFFER_BIT` 和 `GL_STENCIL_BUFFER_BIT`。由于现在我们只关心颜色值，所以我们只清空颜色缓冲。

```
glClearColor(0.0f, 0.34f, 0.57f, 1.0f);  
glClear(GL_COLOR_BUFFER_BIT);
```

注意，除了 `glClear` 之外，我们还调用了 `glClearColor` 来设置清空屏幕所用的颜色。当调用 `glClear` 函数，清除颜色缓冲之后，整个颜色缓冲都会被填充为 `glClearColor` 里所设置的颜色。在这里，我们将屏幕设置为了一种好看的蓝色。效果如下：



图 5-1 窗口图（蓝色）