EBook Gratis

APRENDIZAJE opengl

Free unaffiliated eBook created from
Stack Overflow contributors.

#opengl

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: opengl

It is an unofficial and free opengl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official opengl.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Opengl

Observaciones

OpenGL es un estándar abierto para renderizar gráficos en 2D y 3D aprovechando el hardware de gráficos. OpenGL se ha implementado en una impresionante variedad de plataformas que permiten que las aplicaciones dirigidas a OpenGL sean extremadamente flexibles.

Versiones

Versión	Fecha de lanzamiento
1.1	1997-03-04
1.2	1998-03-16
1.2.1	1998-10-14
1.3	2001-08-14
1.4	2002-07-24
1.5	2003-07-29
2.0	2004-09-07
2.1	2006-07-02
3.0	2008-08-11
3.1	2009-03-24
3.2	2009-08-03
3.3	2010-03-11
4.0	2010-03-11
4.1	2010-07-26
4.2	2011-08-08
4.3	2012-08-06
4.4	2013-07-22
4.5	2014-08-11

Examples

Obtención de OpenGL

Una de las ideas erróneas más comunes acerca de OpenGL es que era una biblioteca que podía instalarse desde fuentes de terceros. Este concepto erróneo lleva a muchas preguntas en la forma "cómo instalar OpenGL" o "dónde descargar el SDK de OpenGL".

Así no es como OpenGL encuentra el camino en el sistema informático. OpenGL en sí mismo es simplemente un conjunto de especificaciones sobre qué comandos debe seguir una implementación. Así que lo que importa es la implementación. Y por el momento, las implementaciones de OpenGL son parte de los controladores de GPU. Esto *podría* cambiar en el futuro, cuando la nueva interfaz de programación de GPU permita implementar verdaderamente OpenGL como una biblioteca, pero por ahora es una API de programación para los controladores de gráficos.

Cuando OpenGL se lanzó por primera vez, la API encontró su camino en el contrato ABI (Interfaz Binario de la Aplicación) de Windows, Solaris y Linux (Escritorio LSB-4), además de su origen, Sun Irix. Apple siguió y, de hecho, integró OpenGL tan profundamente en MacOS X, que la versión de OpenGL disponible está estrechamente unida a la versión de MacOS X instalada. Esto tiene el efecto notable de que los entornos de programación del sistema para estos sistemas operativos (es decir, la cadena de herramientas del compilador y el enlazador que se orienta de forma nativa a estos sistemas) también **deben** proporcionar definiciones de API OpenGL. Por lo tanto, no es necesario instalar realmente un SDK para OpenGL. Es técnicamente posible programar OpenGL en estos sistemas operativos sin el requisito de instalar un SDK dedicado, suponiendo que se instale un entorno de compilación después de la ABI objetivo.

Un efecto secundario de estas estrictas reglas ABI es que la versión OpenGL expuesta a través de la interfaz de enlace es el denominador común más bajo que los programas que se ejecutan en la plataforma de destino pueden esperar que estén disponibles. Por lo tanto, se puede acceder a las funciones modernas de OpenGL a través del mecanismo de extensión, que se describe en profundidad por separado.

Linux

En Linux es bastante común compartimentar los paquetes de desarrollo para diferentes aspectos del sistema, para que estos puedan actualizarse individualmente. En la mayoría de las distribuciones de Linux, los archivos de desarrollo para OpenGL están contenidos en un paquete dedicado, que suele ser una dependencia para un meta-paquete de desarrollo de aplicaciones de escritorio. Por lo tanto, la instalación de los archivos de desarrollo de OpenGL para Linux generalmente se realiza con la instalación del / los paquetes de desarrollo de escritorio. *

Microsoft Windows

La biblioteca de enlace de API `opengl32.dll` (llamada así para las versiones de Windows de 32 y 64 bits) se envía de forma predeterminada con todas las versiones de Windows desde Windows

NT-4 y Windows 95B (ambas hacia 1997). Sin embargo, esta DLL no proporciona una implementación real de OpenGL (aparte de un respaldo de software cuyo único propósito es actuar como una red de seguridad para los programas si no se instala otra implementación de OpenGL). Este DLL pertenece a Windows y **no** debe modificarse ni moverse. Las versiones modernas de OpenGL se envían como parte del llamado *Controlador de cliente instalable* (ICD) y se accede a ellas a través del `opengl32.dll` predeterminado que viene preinstalado con cada versión de Windows. Microsoft decidió, sin embargo, que los controladores de gráficos instalados a través de *Windows Update* no instalarían / actualizarían un ICD OpenGL. Como tales, las nuevas instalaciones de Windows con controladores instalados automáticamente carecen de soporte para las características modernas de OpenGL. Para obtener un ICD OpenGL con características modernas, los controladores de gráficos deben descargarse directamente desde el sitio web del proveedor de la GPU e instalarse manualmente.

En cuanto al desarrollo, no se deben tomar medidas adicionales per se. Todos los compiladores de C / C ++ que siguen las especificaciones de Windows ABI se envían con encabezados y el código auxiliar del vinculador (opengl32.lib) necesarios para compilar y vincular los ejecutables que utilizan OpenGL.

Configuración manual de OpenGL en Windows

Código de ejemplo completo incluido al final

Componentes de Windows para OpenGL

WGL

WGL (se puede pronunciar *meneo*) significa "Windows-GL", como en "una interfaz entre Windows y OpenGL", un conjunto de funciones de la API de Windows para comunicarse con OpenGL. Las funciones WGL tienen un prefijo *wgl* y sus tokens tienen un prefijo *WGL_* .

La versión predeterminada de OpenGL compatible con los sistemas de Microsoft es 1.1. Esa es una versión muy antigua (la más reciente es 4.5). La forma de obtener las versiones más recientes es actualizar sus controladores de gráficos, pero su tarjeta gráfica debe ser compatible con esas nuevas versiones.

La lista completa de funciones WGL se puede encontrar aquí .

Interfaz de dispositivo gráfico (GDI)

GDI (actualizado hoy a GDI +) es una interfaz de dibujo en 2D que le permite dibujar en una ventana en Windows. Necesita GDI para inicializar OpenGL y permitir que interactúe con él (pero en realidad no utilizará GDI).

En GDI, cada ventana tiene un *contexto de dispositivo (DC)* que se utiliza para identificar el destino del dibujo al llamar a las funciones (se pasa como parámetro). Sin embargo, OpenGL

utiliza su propio *contexto de representación (RC)* . Por lo tanto, DC se utilizará para crear RC.

Configuración básica

Creando una ventana

Entonces, para hacer cosas en OpenGL, necesitamos RC, y para obtener RC, necesitamos DC, y para obtener DC necesitamos una ventana. Crear una ventana usando la API de Windows requiere varios pasos. *Esta es una rutina básica, por lo que para una explicación más detallada, debe consultar otra documentación, ya que no se trata de usar la API de Windows.*

Esta es una configuración de Windows, por lo que se debe incluir `Windows.h` , y el punto de entrada del programa debe ser el procedimiento `WinMain` con sus parámetros. El programa también debe estar vinculado a `opengl32.dll` y a `gdi32.dll` (independientemente de si está en un sistema de 64 o 32 bits).

Primero necesitamos describir nuestra ventana usando la estructura `WNDCLASS` . Contiene información sobre la ventana que queremos crear:

```
/* REGISTER WINDOW */
WNDCLASS window_class;

// Clear all structure fields to zero first
ZeroMemory(&window_class, sizeof(window_class));

// Define fields we need (others will be zero)
window_class.style = CS_OWNDC;
window_class.lpfnWndProc = window_procedure; // To be introduced later
window_class.hInstance = instance_handle;
window_class.lpszClassName = TEXT("OPENGL_WINDOW");

// Give our class to Windows
RegisterClass(&window_class);
/* *************** */
```

Para una explicación precisa del significado de cada campo (y para una lista completa de campos), consulte la documentación de MSDN.

Luego, podemos crear una ventana usando `CreateWindowEx` . Una vez creada la ventana, podemos adquirir su DC:

```
/* CREATE WINDOW */
HWND window_handle = CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
                                    TEXT("OPENGL_WINDOW"),
                                    TEXT("OpenGL window"),
                                    WS_OVERLAPPEDWINDOW,
                                    0, 0,
                                    800, 600,
                                    NULL,
                                    NULL,
```

```
                                    instance_handle,
                                    NULL);

HDC dc = GetDC(window_handle);

ShowWindow(window_handle, SW_SHOW);
/* ************* */
```

Finalmente, necesitamos crear un bucle de mensajes que reciba eventos de ventana desde el SO:

```
/* EVENT PUMP */
MSG msg;

while (true) {
    if (PeekMessage(&msg, window_handle, 0, 0, PM_REMOVE)) {
        if (msg.message == WM_QUIT)
            break;

        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // draw(); <- there goes your drawing

    SwapBuffers(dc); // To be mentioned later
}
/* ********** */
```

Formato pixel

OpenGL necesita conocer cierta información sobre nuestra ventana, como el bitness de color, el método de almacenamiento en búfer, etc. Para ello, utilizamos un *formato de píxel* . Sin embargo, solo podemos sugerir al sistema operativo qué tipo de formato de píxel necesitamos, y el sistema operativo proporcionará *el más compatible* , no tenemos control directo sobre él. Es por eso que solo se llama *descriptor* .

```
/* PIXEL FORMAT */
PIXELFORMATDESCRIPTOR descriptor;

// Clear all structure fields to zero first
ZeroMemory(&descriptor, sizeof(descriptor));

// Describe our pixel format
descriptor.nSize = sizeof(descriptor);
descriptor.nVersion = 1;
descriptor.dwFlags = PFD_DRAW_TO_WINDOW | PFD_DRAW_TO_BITMAP | PFD_SUPPORT_OPENGL |
PFD_GENERIC_ACCELERATED | PFD_DOUBLEBUFFER | PFD_SWAP_LAYER_BUFFERS;
descriptor.iPixelType = PFD_TYPE_RGBA;
descriptor.cColorBits = 32;
descriptor.cRedBits = 8;
descriptor.cGreenBits = 8;
descriptor.cBlueBits = 8;
descriptor.cAlphaBits = 8;
descriptor.cDepthBits = 32;
```

```
descriptor.cStencilBits = 8;

// Ask for a similar supported format and set it
int pixel_format = ChoosePixelFormat(dc, &descriptor);
SetPixelFormat(dc, pixel_format, &descriptor);
/* ********************** */
```

Hemos habilitado el búfer doble en el campo `dwFlags` , por lo que debemos llamar a `SwapBuffers` para ver las cosas después del dibujo.

Contexto de representación

Después de eso, simplemente podemos crear nuestro contexto de representación:

```
/* RENDERING CONTEXT */
HGLRC rc = wglCreateContext(dc);
wglMakeCurrent(dc, rc);
/* ***************** */
```

Tenga en cuenta que solo un hilo puede usar el RC a la vez. Si desea usarlo desde otro hilo más tarde, debe llamar a `wglMakeCurrent` allí para activarlo nuevamente (esto lo desactivará en el hilo que está actualmente activo, y así sucesivamente).

Obteniendo funciones OpenGL

Las funciones de OpenGL se obtienen mediante el uso de punteros de función. El procedimiento general es:

1. De alguna manera obtener tipos de punteros de función (esencialmente los prototipos de función)
2. Declare cada función que nos gustaría usar (con su tipo de puntero de función)
3. Obtener la función real

Por ejemplo, considere glBegin:

```
// We need to somehow find something that contains something like this,
// as we can't know all the OpenGL function prototypes
typedef void (APIENTRY *PFNGLBEGINPROC)(GLenum);

// After that, we need to declare the function in order to use it
PFNGLBEGINPROC glBegin;

// And finally, we need to somehow make it an actual function
```

("PFN" significa "puntero a función", luego sigue el nombre de una función OpenGL, y "PROC" al final - ese es el nombre habitual del tipo de puntero de función OpenGL).

Así es como se hace en Windows. Como se mencionó anteriormente, Microsoft solo ofrece OpenGL 1.1. Primero, los tipos de punteros de función para esa versión se pueden encontrar incluyendo `GL/gl.h` Después de eso, declaramos todas las funciones que pretendemos usar como

se muestra arriba (hacer eso en un archivo de encabezado y declararlas "externas" nos permitiría usarlas todas después de cargarlas una vez, solo incluyéndolas). Finalmente, la carga de las funciones de OpenGL 1.1 se realiza abriendo la DLL:

```
HMODULE gl_module = LoadLibrary(TEXT("opengl32.dll"));

/* Load all the functions here */
glBegin = (PFNGLBEGINPROC)GetProcAddress("glBegin");
// ...
/* ************************** */

FreeLibrary(gl_module);
```

Sin embargo, probablemente queremos un poco más que OpenGL 1.1. Pero Windows no nos proporciona los prototipos de funciones ni las funciones exportadas para nada que se encuentre por encima de eso. Los prototipos deben ser adquiridos desde el registro de OpenGL . Hay tres archivos que nos interesan: `GL/glext.h` , `GL/glcorearb.h` , y `GL/wglext.h` .

Para completar `GL/gl.h` proporcionado por Windows, necesitamos `GL/glext.h` . Contiene (como se describe en el registro) "OpenGL 1.2 y superiores interfaces de extensión y perfil de compatibilidad" (más información sobre perfiles y extensiones más adelante, donde veremos que en **realidad no es una buena idea usar esos dos archivos**).

Las funciones reales deben ser obtenidas por `wglGetProcAddress` (no hay necesidad de abrir la DLL para este tipo, no están ahí, solo use la función). Con él, podemos obtener todas las funciones de OpenGL 1.2 y superiores (pero no 1.1). Tenga en cuenta que, para que funcione correctamente, **el contexto de representación OpenGL debe crearse y actualizarse** . Así, por ejemplo, `glClear` :

```
// Include the header from the OpenGL registry for function pointer types

// Declare the functions, just like before
PFNGLCLEARPROC glClear;
// ...

// Get the function
glClear = (PFNGLCLEARPROC)wglGetProcAddress("glClear");
```

En realidad, podemos construir un envoltorio procedimiento `get_proc` que use `wglGetProcAddress` y `GetProcAddress` :

```
// Get function pointer
void* get_proc(const char *proc_name)
{
    void *proc = (void*)wglGetProcAddress(proc_name);
    if (!proc) proc = (void*)GetProcAddress(gl_module, proc_name); // gl_module must be
somewhere in reach

    return proc;
}
```

Así que para terminar, crearíamos un archivo de encabezado lleno de declaraciones de punteros

de función como esta:

```
extern PFNGLCLEARCOLORPROC glClearColor;
extern PFNGLCLEARDEPTHPROC glClearDepth;
extern PFNGLCLEARPROC glClear;
extern PFNGLCLEARBUFFERIVPROC glClearBufferiv;
extern PFNGLCLEARBUFFERFVPROC glClearBufferfv;
// And so on...
```

Luego podemos crear un procedimiento como `load_gl_functions` que llamamos solo una vez, y funciona así:

```
glClearColor = (PFNGLCLEARCOLORPROC)get_proc("glClearColor");
glClearDepth = (PFNGLCLEARDEPTHPROC)get_proc("glClearDepth");
glClear = (PFNGLCLEARPROC)get_proc("glClear");
glClearBufferiv = (PFNGLCLEARBUFFERIVPROC)get_proc("glClearBufferiv");
glClearBufferfv = (PFNGLCLEARBUFFERFVPROC)get_proc("glClearBufferfv");
```

¡Y ya está todo listo! Solo incluye el encabezado con los punteros a la función y GL fuera.

Mejor configuracion

Perfiles de OpenGL

OpenGL ha estado en desarrollo durante más de 20 años, y los desarrolladores siempre fueron estrictos con respecto a la *compatibilidad con versiones anteriores (BC)* . Agregar una nueva característica es muy difícil debido a eso. Así, en 2008, se separó en dos "perfiles". *Núcleo* y *compatibilidad* . El perfil central rompe BC a favor de mejoras de rendimiento y algunas de las nuevas características. Incluso elimina por completo algunas características heredadas. El perfil de compatibilidad mantiene BC con todas las versiones hasta 1.0, y algunas características nuevas no están disponibles en él. Solo se debe utilizar para sistemas antiguos y heredados, todas las aplicaciones nuevas deben usar el perfil central.

Debido a eso, hay un problema con nuestra configuración básica: solo proporciona el contexto que es compatible con OpenGL 1.0. El formato de píxeles es limitado también. Hay un mejor enfoque, utilizando extensiones.

Extensiones de OpenGL

Cualquier adición a la funcionalidad original de OpenGL se llama extensiones. En general, pueden legalizar algunas cosas que no existían antes, extender el rango de valores de los parámetros, extender GLSL e incluso agregar una funcionalidad completamente nueva.

Hay tres grupos principales de extensiones: proveedor, EXT y ARB. Las extensiones de proveedor provienen de un proveedor específico y tienen una marca específica de proveedor, como AMD o NV. Extensiones EXT son hechas por varios proveedores trabajando juntos.

Después de algún tiempo, pueden convertirse en extensiones ARB, que son todas las que cuentan con el respaldo oficial y las aprobadas por ARB.

Para adquirir tipos de puntero de función y prototipos de función de todas las extensiones *y como se mencionó anteriormente, todos los tipos de puntero de función de OpenGL 1.2 y superiores* , uno debe descargar los archivos de encabezado del registro de OpenGL . Como se comentó, para las aplicaciones nuevas es mejor usar el perfil del núcleo, por lo que sería preferible incluir `GL/glcorearb.h` lugar de `GL/gl.h` y `GL/glext.h` (si está usando `GL/glcorearb.h` entonces don no incluye `GL/gl.h`).

También hay extensiones para el WGL, en `GL/wglext.h` . Por ejemplo, la función para obtener la lista de todas las extensiones compatibles es en realidad una extensión en sí misma, `wglGetExtensionsStringARB` (devuelve una cadena grande con una lista separada por espacios de todas las extensiones compatibles).

La obtención de extensiones también se maneja a través de `wglGetProcAddress` , por lo que podemos usar nuestro contenedor como antes.

Formato avanzado de píxeles y creación de contexto

La extensión `WGL_ARB_pixel_format` nos permite la creación avanzada de formato de píxeles. A diferencia de antes, no usamos una estructura. En su lugar, pasamos la lista de atributos deseados.

```
int pixel_format_arb;
UINT pixel_formats_found;

int pixel_attributes[] = {
    WGL_SUPPORT_OPENGL_ARB, 1,
    WGL_DRAW_TO_WINDOW_ARB, 1,
    WGL_DRAW_TO_BITMAP_ARB, 1,
    WGL_DOUBLE_BUFFER_ARB, 1,
    WGL_SWAP_LAYER_BUFFERS_ARB, 1,
    WGL_COLOR_BITS_ARB, 32,
    WGL_RED_BITS_ARB, 8,
    WGL_GREEN_BITS_ARB, 8,
    WGL_BLUE_BITS_ARB, 8,
    WGL_ALPHA_BITS_ARB, 8,
    WGL_DEPTH_BITS_ARB, 32,
    WGL_STENCIL_BITS_ARB, 8,
    WGL_ACCELERATION_ARB, WGL_FULL_ACCELERATION_ARB,
    WGL_PIXEL_TYPE_ARB, WGL_TYPE_RGBA_ARB,
    0
};

BOOL result = wglChoosePixelFormatARB(dc, pixel_attributes, NULL, 1, &pixel_format_arb,
&pixel_formats_found);
```

De manera similar, la extensión `WGL_ARB_create_context` nos permite la creación avanzada de contexto:

```
GLint context_attributes[] = {
```

```
    WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
    WGL_CONTEXT_MINOR_VERSION_ARB, 3,
    WGL_CONTEXT_PROFILE_MASK_ARB, WGL_CONTEXT_CORE_PROFILE_BIT_ARB,
    0
};

HGLRC new_rc = wglCreateContextAttribsARB(dc, 0, context_attributes);
```

Para una explicación precisa de los parámetros y funciones, consulte la especificación de OpenGL.

¿Por qué no empezamos con ellos? Bueno, eso es porque las *extensiones* nos permiten hacer esto, y para obtener las extensiones necesitamos `wglGetProcAddress` , pero eso solo funciona con un contexto activo válido. Entonces, en esencia, antes de que seamos capaces de crear el contexto que queremos, necesitamos tener algún contexto activo ya, y generalmente se lo denomina *contexto ficticio* .

Sin embargo, Windows no permite configurar el formato de píxeles de una ventana más de una vez. Por eso, la ventana necesita ser destruida y recreada para poder aplicar cosas nuevas:

```
wglMakeCurrent(dc, NULL);
wglDeleteContext(rc);
ReleaseDC(window_handle, dc);
DestroyWindow(window_handle);

// Recreate the window...
```

Código de ejemplo completo:

```
/* We want the core profile, so we include GL/glcorearb.h. When including that, then
   GL/gl.h should not be included.

   If using compatibility profile, the GL/gl.h and GL/glext.h need to be included.

   GL/wglext.h gives WGL extensions.

   Note that Windows.h needs to be included before them. */

#include <cstdio>
#include <Windows.h>
#include <GL/glcorearb.h>
#include <GL/wglext.h>

LRESULT CALLBACK window_procedure(HWND, UINT, WPARAM, LPARAM);
void* get_proc(const char*);

/* gl_module is for opening the DLL, and the quit flag is here to prevent
   quitting when recreating the window (see the window_procedure function) */

HMODULE gl_module;
bool quit = false;

/* OpenGL function declarations. In practice, we would put these in a
   separate header file and add "extern" in front, so that we can use them
   anywhere after loading them only once. */
```

```
PFNWGLGETEXTENSIONSSTRINGARBPROC wglGetExtensionsStringARB;
PFNWGLCHOOSEPIXELFORMATARBPROC wglChoosePixelFormatARB;
PFNWGLCREATECONTEXTATTRIBSARBPROC wglCreateContextAttribsARB;
PFNGLGETSTRINGPROC glGetString;

int WINAPI WinMain(HINSTANCE instance_handle, HINSTANCE prev_instance_handle, PSTR cmd_line,
int cmd_show) {
    /* REGISTER WINDOW */
    WNDCLASS window_class;

    // Clear all structure fields to zero first
    ZeroMemory(&window_class, sizeof(window_class));

    // Define fields we need (others will be zero)
    window_class.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
    window_class.lpfnWndProc = window_procedure;
    window_class.hInstance = instance_handle;
    window_class.lpszClassName = TEXT("OPENGL_WINDOW");

    // Give our class to Windows
    RegisterClass(&window_class);
    /* *************** */

    /* CREATE WINDOW */
    HWND window_handle = CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
                                        TEXT("OPENGL_WINDOW"),
                                        TEXT("OpenGL window"),
                                        WS_OVERLAPPEDWINDOW,
                                        0, 0,
                                        800, 600,
                                        NULL,
                                        NULL,
                                        instance_handle,
                                        NULL);

    HDC dc = GetDC(window_handle);

    ShowWindow(window_handle, SW_SHOW);
    /* ************* */

    /* PIXEL FORMAT */
    PIXELFORMATDESCRIPTOR descriptor;

    // Clear all structure fields to zero first
    ZeroMemory(&descriptor, sizeof(descriptor));

    // Describe our pixel format
    descriptor.nSize = sizeof(descriptor);
    descriptor.nVersion = 1;
    descriptor.dwFlags = PFD_DRAW_TO_WINDOW | PFD_DRAW_TO_BITMAP | PFD_SUPPORT_OPENGL |
PFD_GENERIC_ACCELERATED | PFD_DOUBLEBUFFER | PFD_SWAP_LAYER_BUFFERS;
    descriptor.iPixelType = PFD_TYPE_RGBA;
    descriptor.cColorBits = 32;
    descriptor.cRedBits = 8;
    descriptor.cGreenBits = 8;
    descriptor.cBlueBits = 8;
    descriptor.cAlphaBits = 8;
    descriptor.cDepthBits = 32;
    descriptor.cStencilBits = 8;
```

```
    // Ask for a similar supported format and set it
    int pixel_format = ChoosePixelFormat(dc, &descriptor);
    SetPixelFormat(dc, pixel_format, &descriptor);
    /* ********************** */

    /* RENDERING CONTEXT */
    HGLRC rc = wglCreateContext(dc);
    wglMakeCurrent(dc, rc);
    /* ***************** */

    /* LOAD FUNCTIONS (should probably be put in a separate procedure) */
    gl_module = LoadLibrary(TEXT("opengl32.dll"));

    wglGetExtensionsStringARB =
(PFNWGLGETEXTENSIONSSTRINGARBPROC)get_proc("wglGetExtensionsStringARB");
    wglChoosePixelFormatARB =
(PFNWGLCHOOSEPIXELFORMATARBPROC)get_proc("wglChoosePixelFormatARB");
    wglCreateContextAttribsARB =
(PFNWGLCREATECONTEXTATTRIBSARBPROC)get_proc("wglCreateContextAttribsARB");
    glGetString = (PFNGLGETSTRINGPROC)get_proc("glGetString");

    FreeLibrary(gl_module);
    /* ************** */

    /* PRINT VERSION */
    const GLubyte *version = glGetString(GL_VERSION);
    printf("%s\n", version);
    fflush(stdout);
    /* ******* */

    /* NEW PIXEL FORMAT*/
    int pixel_format_arb;
    UINT pixel_formats_found;

    int pixel_attributes[] = {
        WGL_SUPPORT_OPENGL_ARB, 1,
        WGL_DRAW_TO_WINDOW_ARB, 1,
        WGL_DRAW_TO_BITMAP_ARB, 1,
        WGL_DOUBLE_BUFFER_ARB, 1,
        WGL_SWAP_LAYER_BUFFERS_ARB, 1,
        WGL_COLOR_BITS_ARB, 32,
        WGL_RED_BITS_ARB, 8,
        WGL_GREEN_BITS_ARB, 8,
        WGL_BLUE_BITS_ARB, 8,
        WGL_ALPHA_BITS_ARB, 8,
        WGL_DEPTH_BITS_ARB, 32,
        WGL_STENCIL_BITS_ARB, 8,
        WGL_ACCELERATION_ARB, WGL_FULL_ACCELERATION_ARB,
        WGL_PIXEL_TYPE_ARB, WGL_TYPE_RGBA_ARB,
        0
    };

    BOOL result = wglChoosePixelFormatARB(dc, pixel_attributes, NULL, 1, &pixel_format_arb,
&pixel_formats_found);

    if (!result) {
        printf("Could not find pixel format\n");
        fflush(stdout);
        return 0;
    }
    /* ***************** */
```

```
    /* RECREATE WINDOW */
    wglMakeCurrent(dc, NULL);
    wglDeleteContext(rc);
    ReleaseDC(window_handle, dc);
    DestroyWindow(window_handle);

    window_handle = CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
                                   TEXT("OPENGL_WINDOW"),
                                   TEXT("OpenGL window"),
                                   WS_OVERLAPPEDWINDOW,
                                   0, 0,
                                   800, 600,
                                   NULL,
                                   NULL,
                                   instance_handle,
                                   NULL);

    dc = GetDC(window_handle);

    ShowWindow(window_handle, SW_SHOW);
    /* *************** */

    /* NEW CONTEXT */
    GLint context_attributes[] = {
        WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
        WGL_CONTEXT_MINOR_VERSION_ARB, 3,
        WGL_CONTEXT_PROFILE_MASK_ARB, WGL_CONTEXT_CORE_PROFILE_BIT_ARB,
        0
    };

    rc = wglCreateContextAttribsARB(dc, 0, context_attributes);
    wglMakeCurrent(dc, rc);
    /* ************ */

    /* EVENT PUMP */
    MSG msg;

    while (true) {
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
            if (msg.message == WM_QUIT)
                break;

            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        // draw(); <- there goes your drawing

        SwapBuffers(dc);
    }
    /* ********** */

    return 0;
}

// Procedure that processes window events
LRESULT CALLBACK window_procedure(HWND window_handle, UINT message, WPARAM param_w, LPARAM
param_l)
{
    /* When destroying the dummy window, WM_DESTROY message is going to be sent,
```

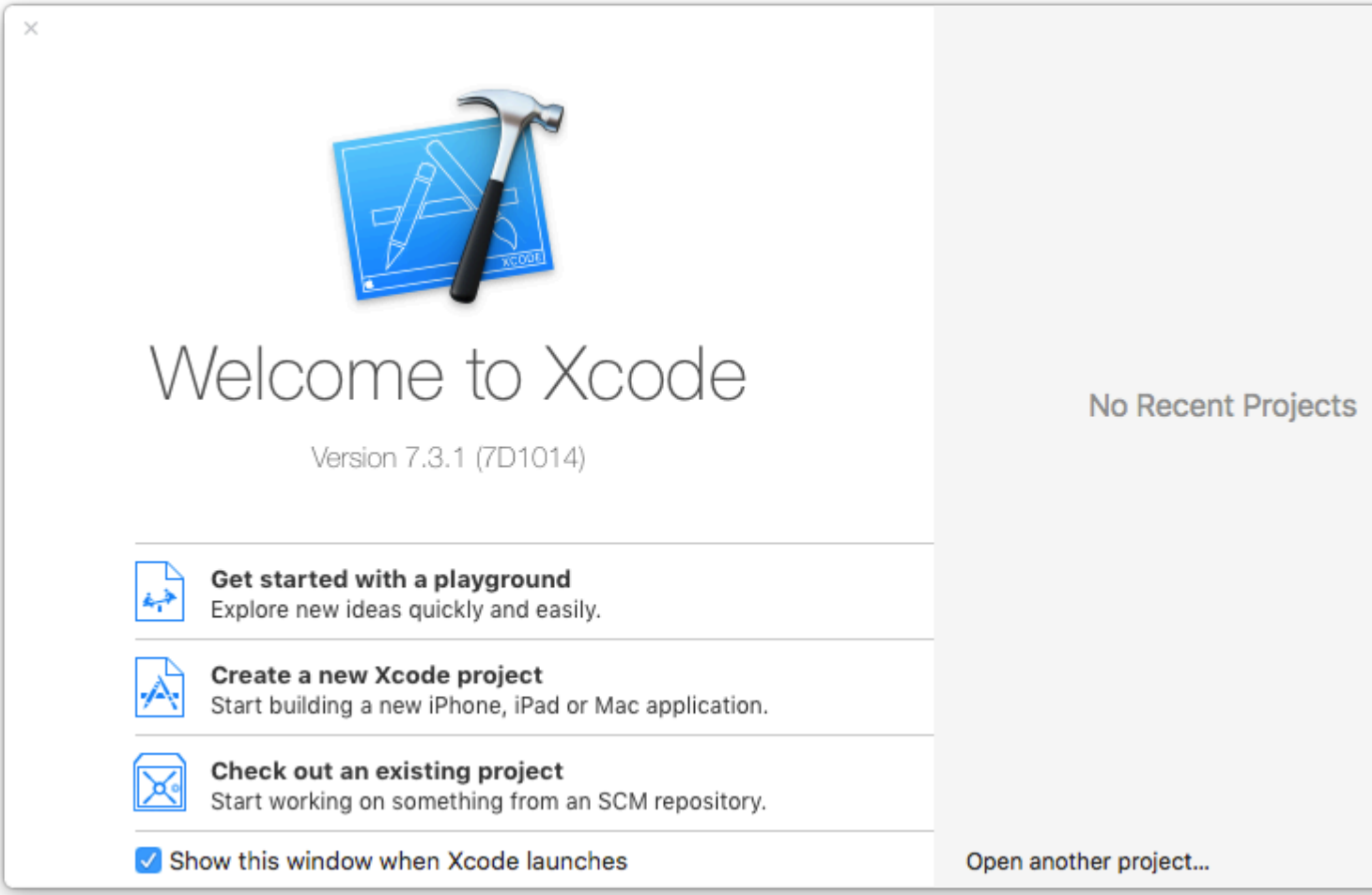
```
       but we don't want to quit the application then, and that is controlled by
       the quit flag. */

    switch(message) {
    case WM_DESTROY:
        if (!quit) quit = true;
        else PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(window_handle, message, param_w, param_l);
}

/* A procedure for getting OpenGL functions and OpenGL or WGL extensions.
   When looking for OpenGL 1.2 and above, or extensions, it uses wglGetProcAddress,
   otherwise it falls back to GetProcAddress. */
void* get_proc(const char *proc_name)
{
    void *proc = (void*)wglGetProcAddress(proc_name);
    if (!proc) proc = (void*)GetProcAddress(gl_module, proc_name);

    return proc;
}
```

Compilado con `g++ GLExample.cpp -lopengl32 -lgdi32` con MinGW / Cygwin o `cl GLExample.cpp opengl32.lib gdi32.lib user32.lib` con el compilador `cl GLExample.cpp opengl32.lib gdi32.lib user32.lib` . Sin embargo, asegúrese de que los encabezados del registro de OpenGL estén en la ruta de inclusión. Si no es así, utilice el indicador `-I` para `g++` o `/I` para `cl` para indicar al compilador dónde están.

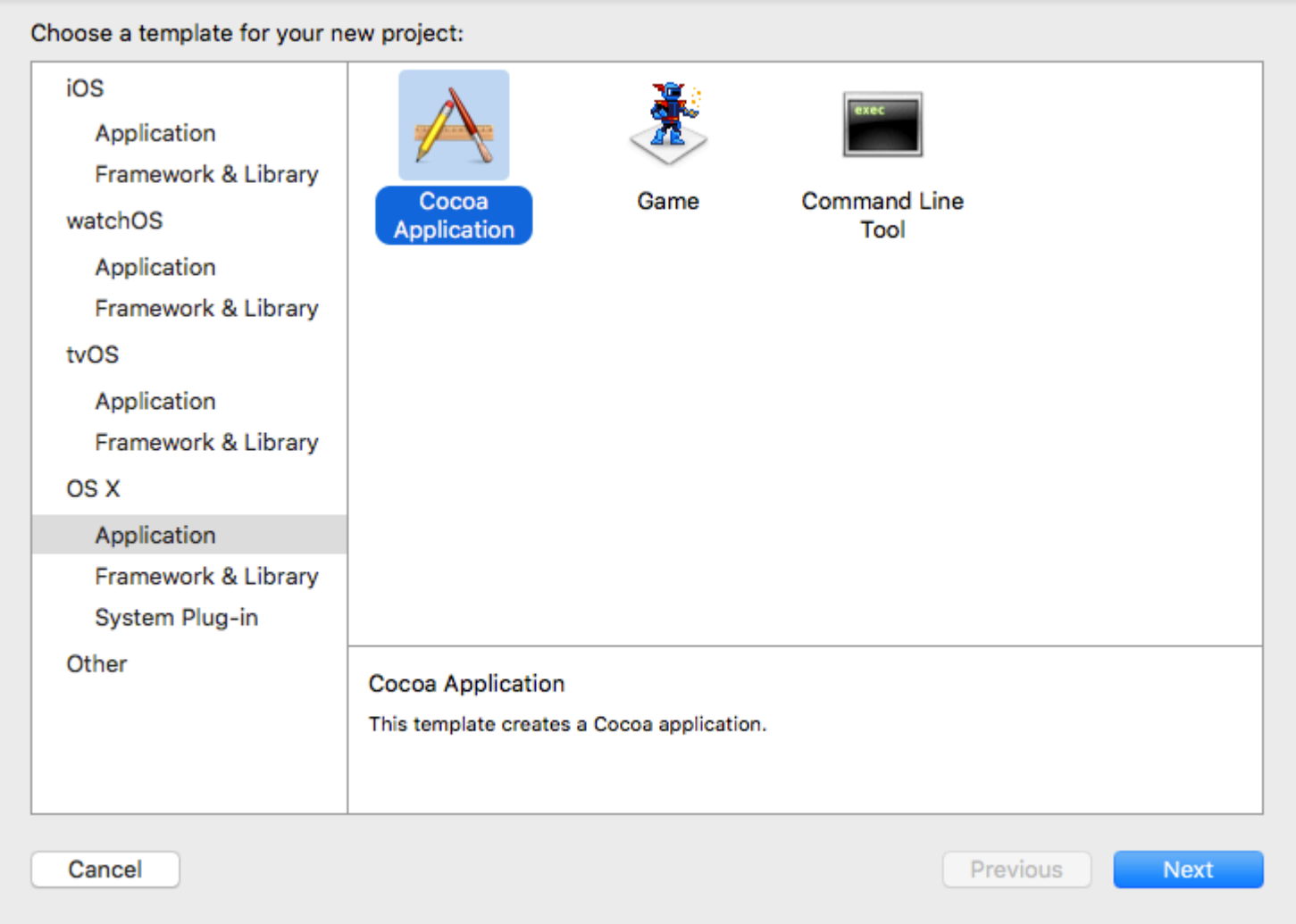
Creando OpenGL 4.1 con C ++ y Cocoa

Nota: Habrá algo de Objective-c en este ejemplo ... Haremos un envoltorio para C ++ en este ejemplo, así que no se preocupe mucho por eso.

Primero inicia Xcode y crea un proyecto.



Y selecciona una aplicación de cacao.

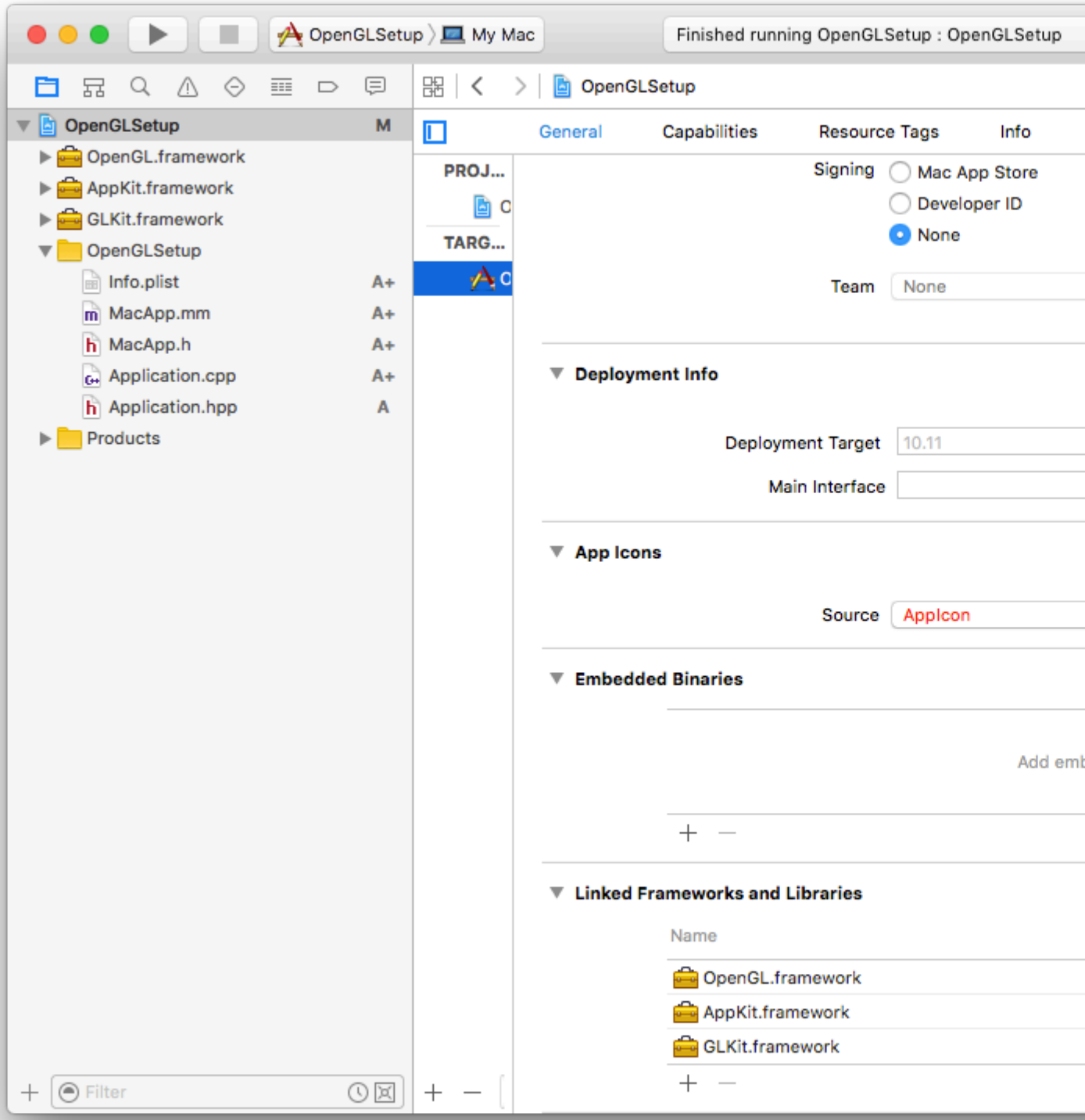


Elimine todas las fuentes excepto el archivo Info.plist. (Su aplicación no funcionará sin él)

Cree 4 nuevos archivos de origen: un archivo y un encabezado de Objective-c ++ (he llamado el mío MacApp) Una clase de C ++ (he llamado el mío (la aplicación)

En la parte superior izquierda (con el nombre del proyecto), haga clic en él y agregue marcos y bibliotecas vinculados. Añadir: OpenGL.Framework AppKit.Framework GLKit.Framework

Tu proyecto se verá probablemente así:



La aplicación NSA es la clase principal que utiliza al crear una aplicación MacOS. Te permite registrar ventanas y atrapar eventos.

Queremos registrar (nuestra propia) ventana en la aplicación NSA. Primero cree en su encabezado de object -c ++ una clase de object -c que herede de NSWindow e implementa NSApplicationDelegate. NSWindow necesita un puntero a la aplicación C ++, una vista openGL y un temporizador para el ciclo de dibujo.

```
//Mac_App_H
#import <Cocoa/Cocoa.h>
#import "Application.hpp"
#import <memory>
NSApplication* application;

@interface MacApp : NSWindow <NSApplicationDelegate>{
    std::shared_ptr<Application> appInstance;
}
@property (nonatomic, retain) NSOpenGLView* glView;
-(void) drawLoop:(NSTimer*) timer;
@end
```

Llamamos a esto desde el principio con

```
int main(int argc, const char * argv[]) {
    MacApp* app;
    application = [NSApplication sharedApplication];
    [NSApp setActivationPolicy:NSApplicationActivationPolicyRegular];
    //create a window with the size of 600 by 600
    app = [[MacApp alloc] initWithContentRect:NSMakeRect(0, 0, 600, 600)
styleMask:NSTitledWindowMask | NSClosableWindowMask |  NSMiniaturizableWindowMask
backing:NSBackingStoreBuffered defer:YES];
    [application setDelegate:app];
    [application run];
}
```

La implementación de nuestra ventana es bastante fácil. Primero, declaramos sintetizando nuestra vista global y agregamos un booleano objetivo-c global cuando la ventana debería cerrarse.

```
#import "MacApp.h"

@implementation MacApp

@synthesize glView;

BOOL shouldStop = NO;
```

Ahora para el constructor. Mi preferencia es usar el initWithContentRect.

```
-(id)initWithContentRect:(NSRect)contentRect styleMask:(NSUInteger)aStyle
backing:(NSBackingStoreType)bufferingType defer:(BOOL)flag{
if(self = [super initWithContentRect:contentRect styleMask:aStyle backing:bufferingType
defer:flag]){
    //sets the title of the window (Declared in Plist)
    [self setTitle:[[NSProcessInfo processInfo] processName]];

    //This is pretty important.. OS X starts always with a context that only supports openGL
2.1
    //This will ditch the classic OpenGL and initialises openGL 4.1
    NSOpenGLPixelFormatAttribute pixelFormatAttributes[] ={
        NSOpenGLPFAOpenGLProfile, NSOpenGLProfileVersion3_2Core,
            NSOpenGLPFAColorSize    , 24                           ,
            NSOpenGLPFAAlphaSize    , 8                            ,
            NSOpenGLPFADoubleBuffer ,
            NSOpenGLPFAAccelerated  ,
```

```
            NSOpenGLPFANoRecovery   ,
            0
    };

    NSOpenGLPixelFormat* format = [[NSOpenGLPixelFormat
alloc]initWithAttributes:pixelFormatAttributes];
    //Initialize the view
    glView = [[NSOpenGLView alloc]initWithFrame:contentRect pixelFormat:format];

    //Set context and attach it to the window
    [[glView openGLContext]makeCurrentContext];

    //finishing off
    [self setContentView:glView];
    [glView prepareOpenGL];
    [self makeKeyAndOrderFront:self];
    [self setAcceptsMouseMovedEvents:YES];
    [self makeKeyWindow];
    [self setOpaque:YES];

    //Start the c++ code
    appInstance = std::shared_ptr<Application>(new Application());

}
return self;
}
```

Bien ... ahora tenemos una aplicación ejecutable ... Es posible que veas una pantalla negra o parpadeos.

Vamos a empezar a dibujar un triángulo impresionante (en c ++)

Mi encabezado de aplicación

```
#ifndef Application_hpp
#define Application_hpp
#include <iostream>
#include <OpenGL/gl3.h>
class Application{
private:
    GLuint          program;
    GLuint          vao;
public:
    Application();
    void update();
    ~Application();

};

#endif /* Application_hpp */
```

La implementación:

```
Application::Application(){
 static const char * vs_source[] =
    {
        "#version 410 core                                                 \n"
        "                                                                  \n"
```

```
        "void main(void)                                                    \n"
        "{                                                                  \n"
        "    const vec4 vertices[] = vec4[](vec4( 0.25, -0.25, 0.5, 1.0),  \n"
        "                                   vec4(-0.25, -0.25, 0.5, 1.0),  \n"
        "                                   vec4( 0.25,  0.25, 0.5, 1.0)); \n"
        "                                                                  \n"
        "    gl_Position = vertices[gl_VertexID];                          \n"
        "}                                                                 \n"
    };

    static const char * fs_source[] =
    {
        "#version 410 core                                                 \n"
        "                                                                  \n"
        "out vec4 color;                                                   \n"
        "                                                                  \n"
        "void main(void)                                                   \n"
        "{                                                                 \n"
        "    color = vec4(0.0, 0.8, 1.0, 1.0);                             \n"
        "}                                                                 \n"
    };

    program = glCreateProgram();
    GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fs, 1, fs_source, NULL);
    glCompileShader(fs);

    GLuint vs = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vs, 1, vs_source, NULL);
    glCompileShader(vs);

    glAttachShader(program, vs);
    glAttachShader(program, fs);

    glLinkProgram(program);

    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);
}

void Application::update(){
    static const GLfloat green[] = { 0.0f, 0.25f, 0.0f, 1.0f };
    glClearBufferfv(GL_COLOR, 0, green);

    glUseProgram(program);
    glDrawArrays(GL_TRIANGLES, 0, 3);
}


Application::~Application(){
    glDeleteVertexArrays(1, &vao);
    glDeleteProgram(program);
}
```

Ahora solo necesitamos llamar a la actualización una y otra vez (si desea que algo se mueva) Implementar en su clase de objetivo-c

```
-(void) drawLoop:(NSTimer*) timer{

if(shouldStop){
```

```
    [self close];
    return;
}
if([self isVisible]){

       appInstance->update();
    [glView update];
    [[glView openGLContext] flushBuffer];
}

}
```

Y agregue este método en la implementación de su clase de objetivo-c:

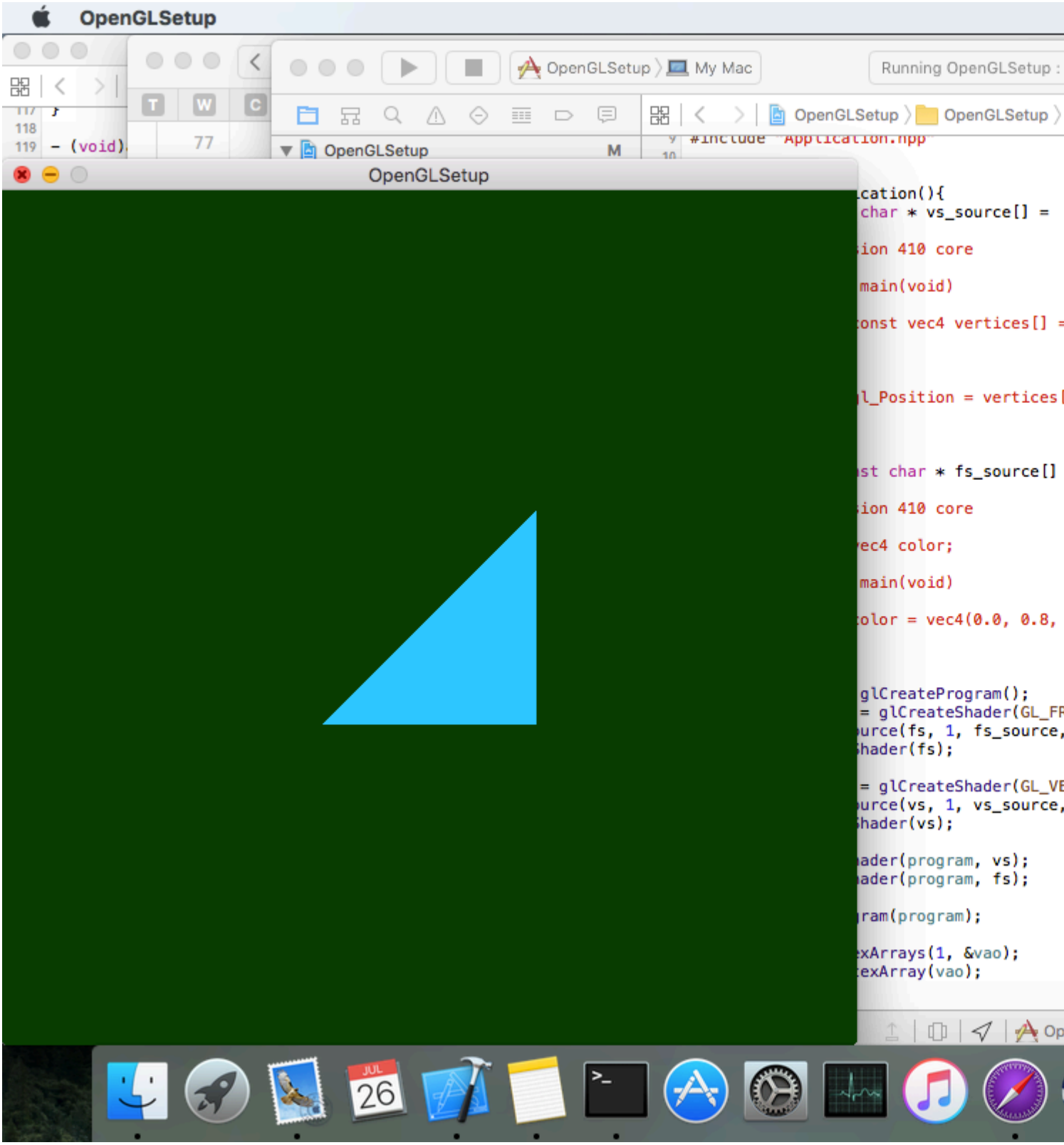
```
- (void)applicationDidFinishLaunching:(NSNotification *)notification {
    [NSTimer scheduledTimerWithTimeInterval:0.000001 target:self selector:@selector(drawLoop:)
userInfo:nil repeats:YES];
}
```

Esto llamará a la función de actualización de su clase c ++ una y otra vez (cada 0.000001 segundos para ser precisos)

Para finalizar cerramos la ventana cuando se presiona el botón de cerrar:

```
- (BOOL)applicationShouldTerminateAfterLastWindowClosed:(NSApplication     *)theApplication{
    return YES;
}

- (void)applicationWillTerminate:(NSNotification *)aNotification{
    shouldStop = YES;
}
```

Felicitaciones, ahora tiene una ventana increíble con un triángulo OpenGL sin ningún marco de terceros.



Creación de contexto OpenGL multiplataforma (utilizando SDL2)

Creación de una ventana con contexto OpenGL (extensión de carga a través de GLEW):

```
#define GLEW_STATIC

#include <GL/glew.h>
#include <SDL2/SDL.h>
```

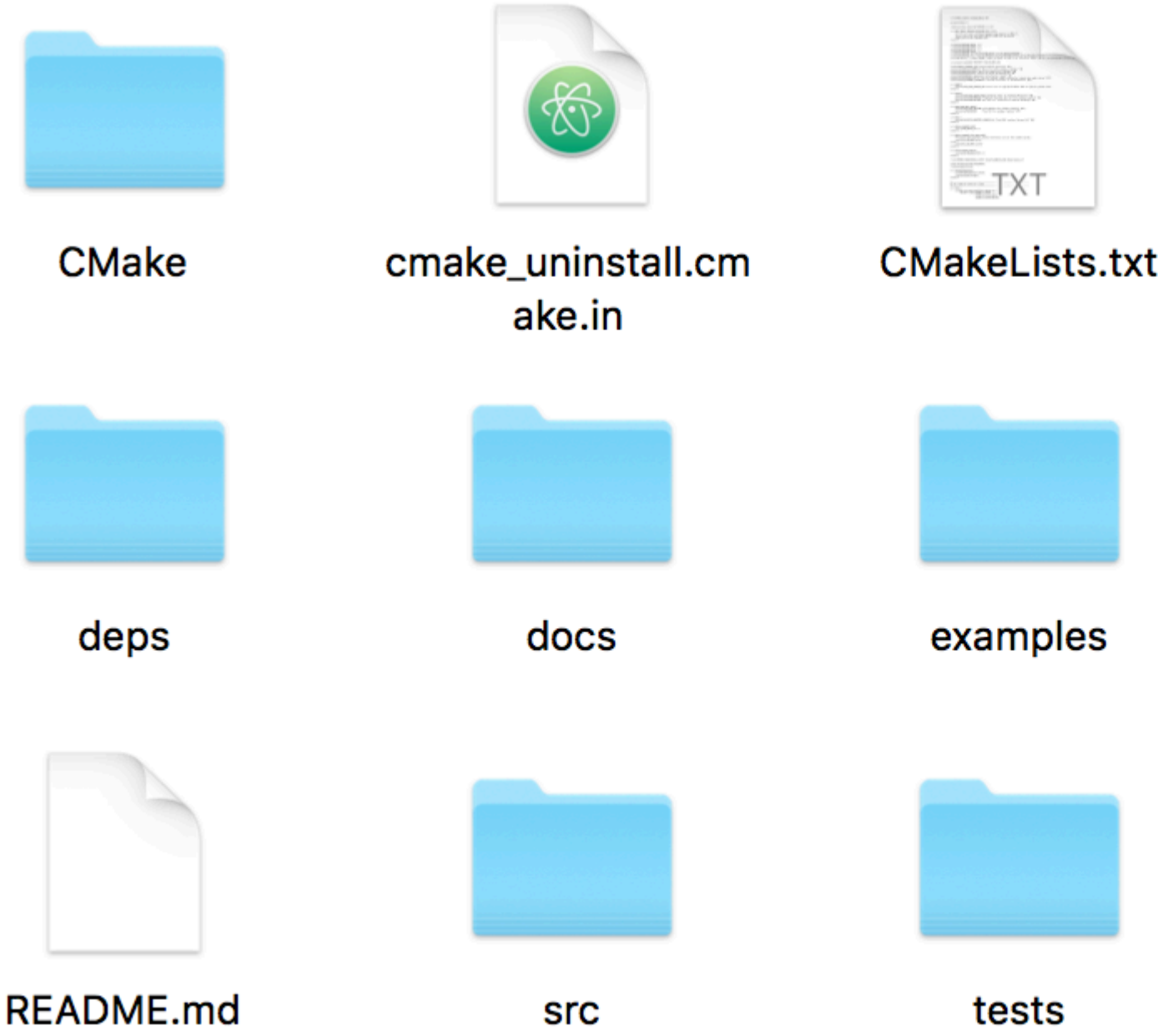
```
int main(int argc, char* argv[])
{
    SDL_Init(SDL_INIT_VIDEO); /* Initialises Video Subsystem in SDL */

    /* Setting up OpenGL version and profile details for context creation */
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);

    /* A 800x600 window. Pretty! */
    SDL_Window* window = SDL_CreateWindow
        (
        "SDL Context",
        SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
        800, 600,
        SDL_WINDOW_OPENGL
        );

    /* Creating OpenGL Context */
    SDL_GLContext gl_context = SDL_GL_CreateContext(window);

    /* Loading Extensions */
    glewExperimental = GL_TRUE;
    glewInit();

    /* The following code is for error checking.
    *  If OpenGL has initialised properly, this should print 1.
    *  Remove it in production code.
    */
    GLuint vertex_buffer;
    glGenBuffers(1, &vertex_buffer);
    printf("%u\n", vertex_buffer);
    /* Error checking ends here */

    /* Main Loop */
    SDL_Event window_event;
    while(1) {
        if (SDL_PollEvent(&window_event)) {
            if (window_event.type == SDL_QUIT) {
                /* If user is exiting the application */
                break;
            }
        }
        /* Swap the front and back buffer for flicker-free rendering */
        SDL_GL_SwapWindow(window);
    }

    /* Freeing Memory */
    glDeleteBuffers(1, &vertex_buffer);
    SDL_GL_DeleteContext(gl_context);
    SDL_Quit();

    return 0;
}
```

Configuración moderna de OpenGL 4.1 en macOS (Xcode, GLFW y GLEW)

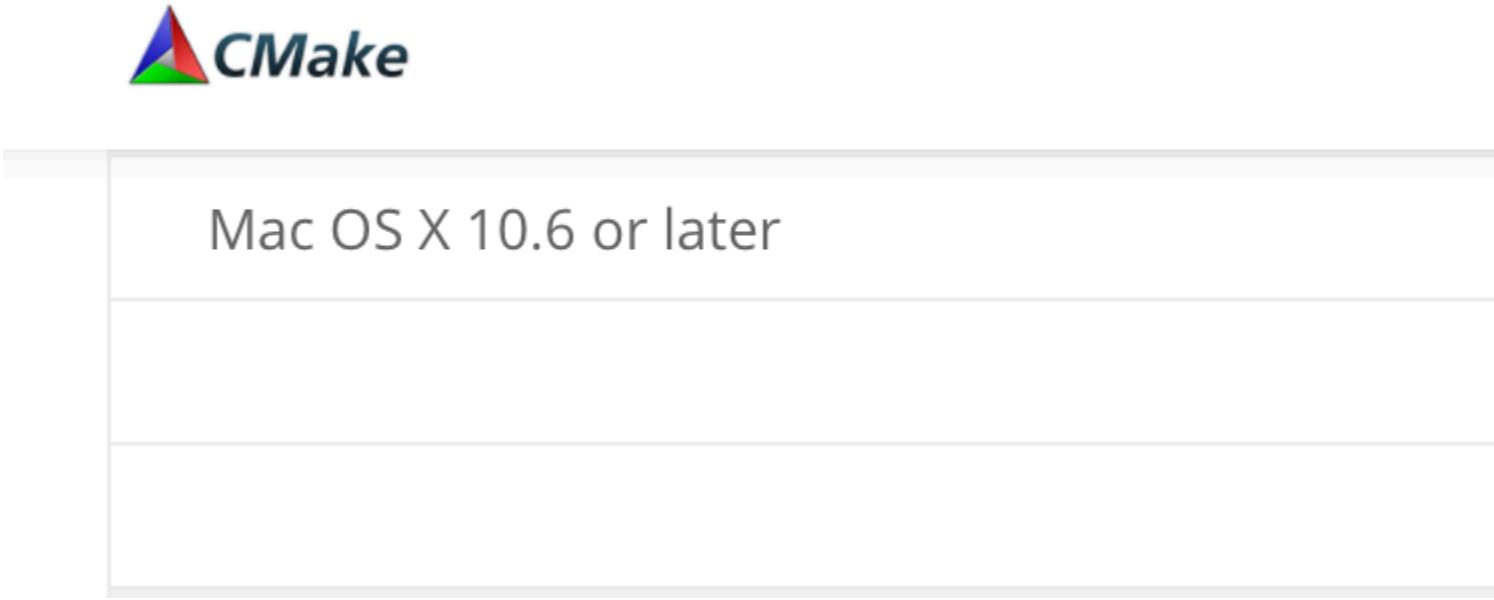
1. Instale GLFW

El primer paso es crear una ventana de OpenGL. GLFW es una biblioteca multiplataforma de código abierto para crear ventanas con OpenGL, para instalar GLFW, primero descargue sus archivos de www.glfw.org

Extrae la carpeta GLFW y su contenido se verá así

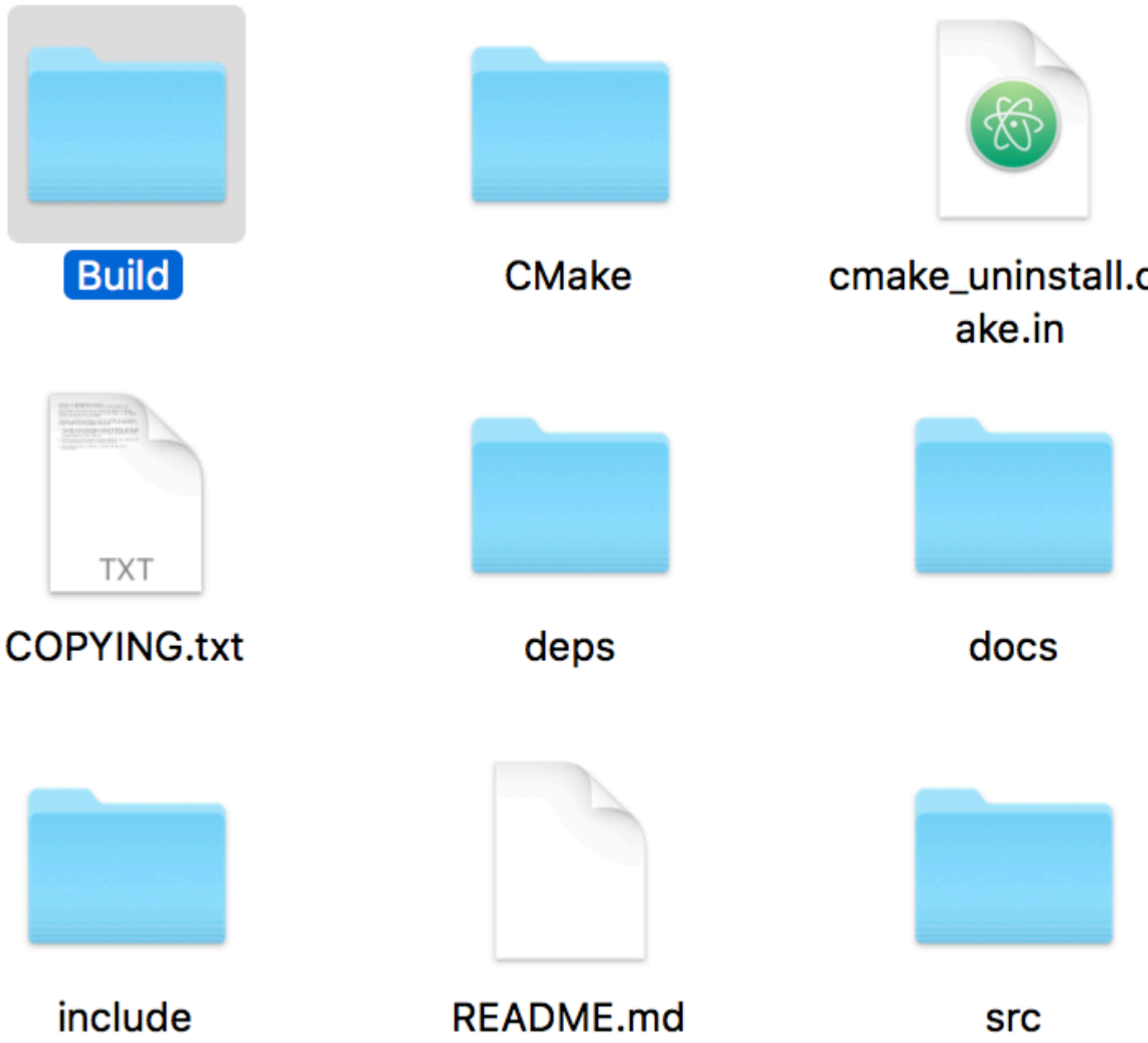


Descarga e instala CMake para compilar GLFW. Vaya a www.cmake.org/download/ , descargue CMake e instale para MAC OS X

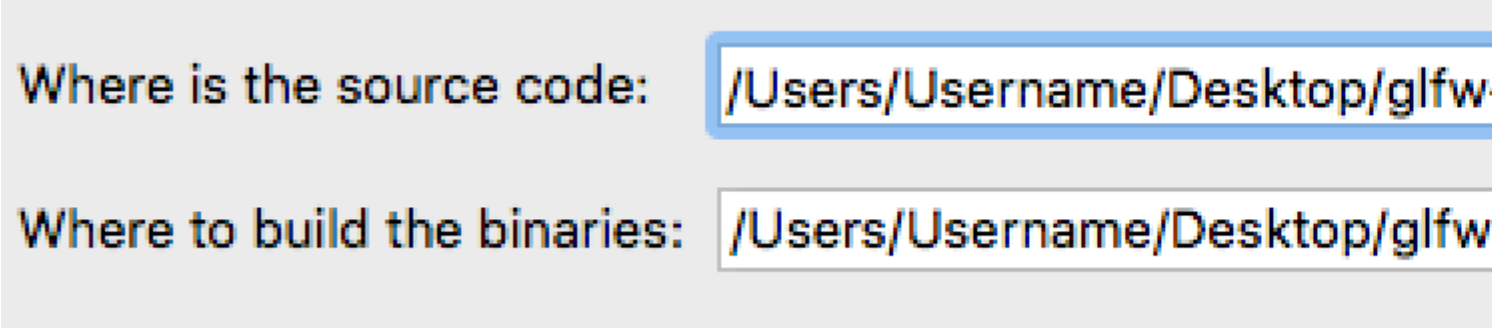


Si Xcode no está instalado. Descargue e instale Xcode desde Mac App Store.

Crear una nueva carpeta **Construir** dentro de la carpeta GLFW

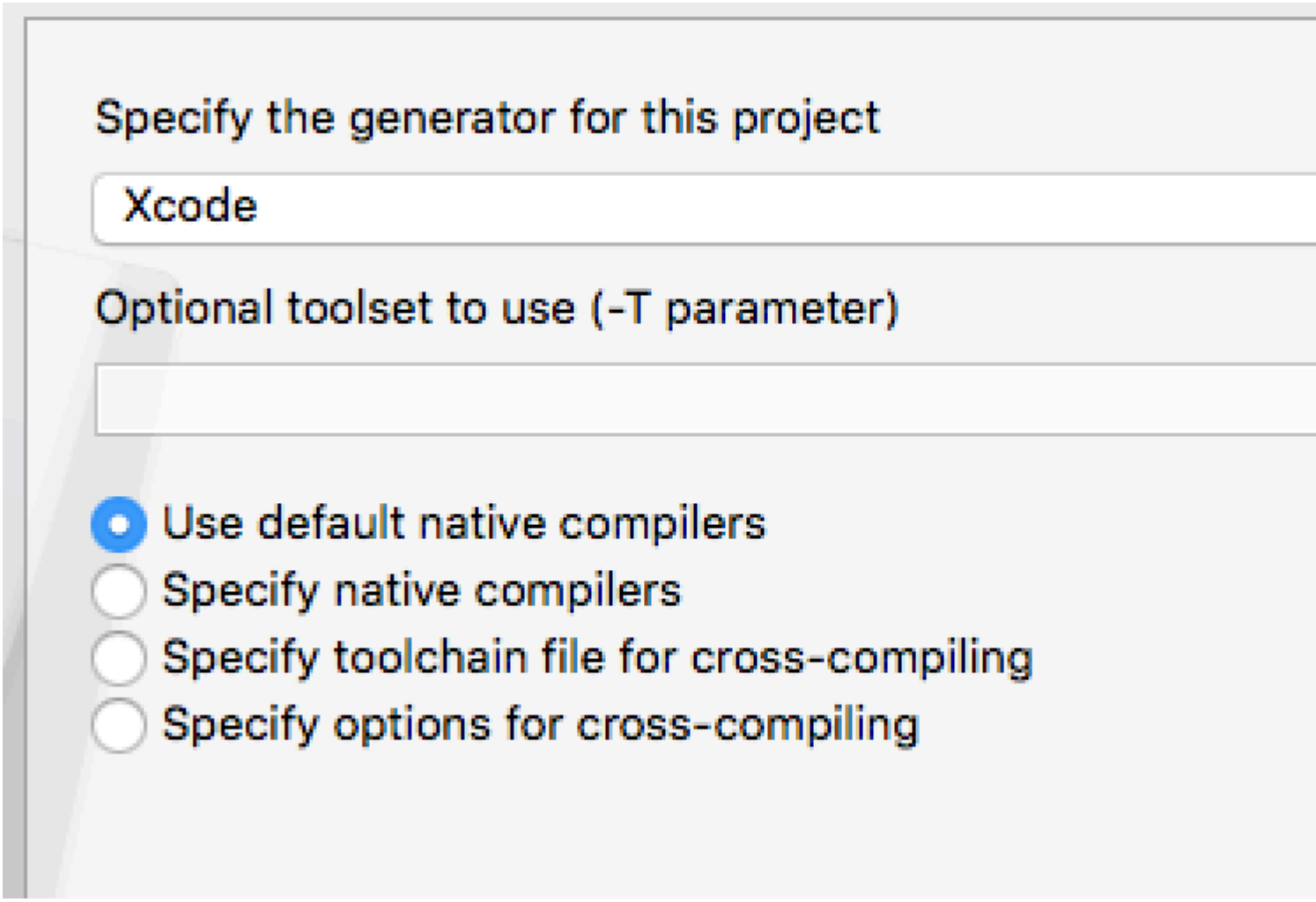


Abra CMake, haga clic en el botón **Buscar fuente** para seleccionar la carpeta GLFW (asegúrese de que CMakeLists.txt) se encuentre dentro de esa carpeta. Después de eso, haga clic en el botón **Explorar compilación** y seleccione la carpeta de **compilación** recién creada en el paso anterior.

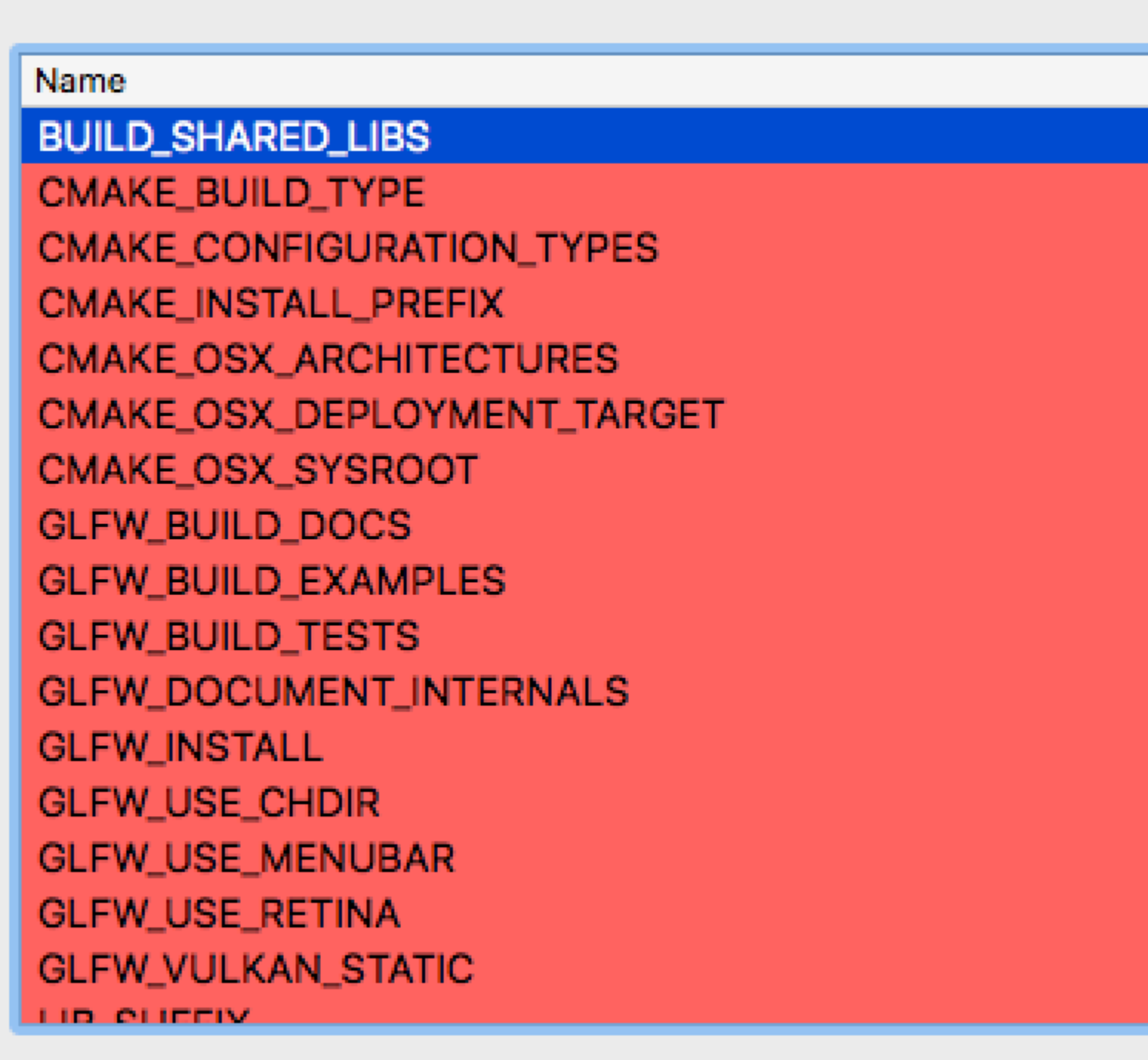


Ahora haga clic en el botón **Configurar** y seleccione **Xcode** como generador con la **opción Usar**

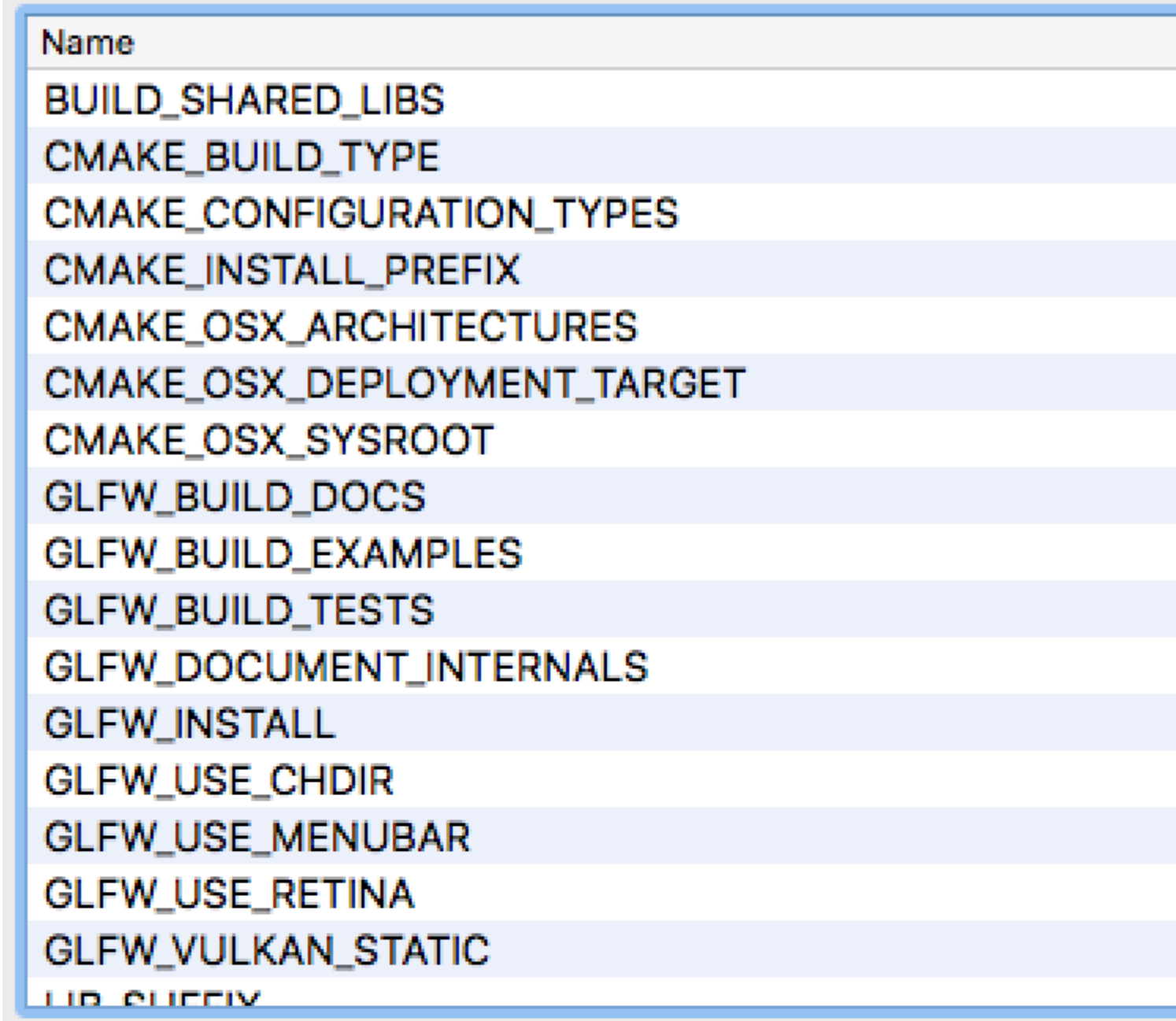
compiladores nativos predeterminados , y haga clic en **Listo** .



Marque la opción **BUILD_SHARED_LIBS** y luego haga clic nuevamente en el botón **Configurar** y finalmente haga clic en el botón **Generar** .

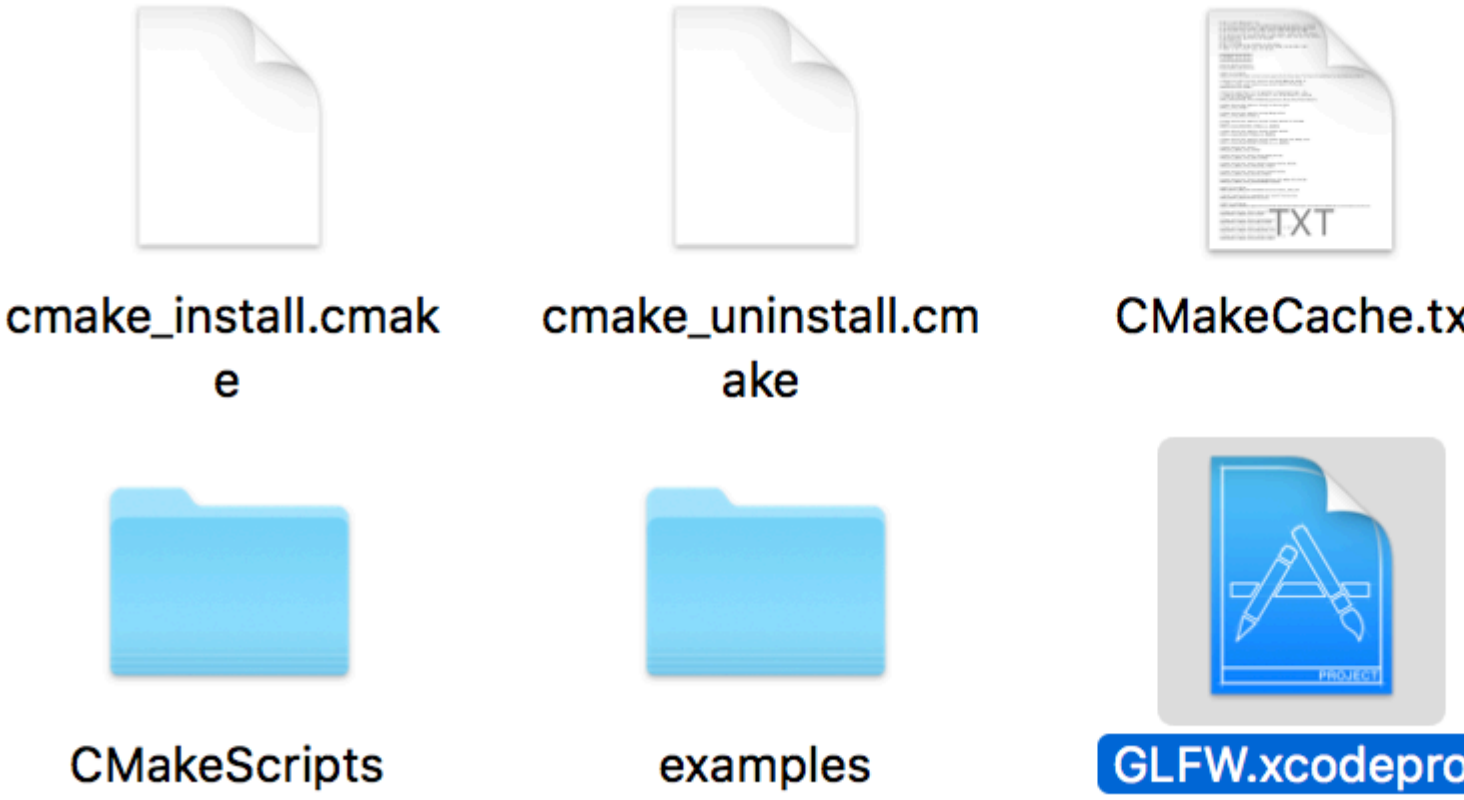


Después de la generación, CMake debería verse así.


Name
BUILD_SHARED_LIBS
CMAKE_BUILD_TYPE
CMAKE_CONFIGURATION_TYPES
CMAKE_INSTALL_PREFIX
CMAKE_OSX_ARCHITECTURES
CMAKE_OSX_DEPLOYMENT_TARGET
CMAKE_OSX_SYSROOT
GLFW_BUILD_DOCS
GLFW_BUILD_EXAMPLES
GLFW_BUILD_TESTS
GLFW_DOCUMENT_INTERNALS
GLFW_INSTALL
GLFW_USE_CHDIR
GLFW_USE_MENUBAR
GLFW_USE_RETINA
GLFW_VULKAN_STATIC
Press Configure to update and display new
Configure
Generate
Current Generator: Xcode
Could NOT find Vulkan (missing: VULKAN_LIBRARY
Could NOT find Doxygen (missing: DOXYGEN_EXECU
Using Cocoa for window creation
Configuring done
Generating done

si no está ya allí. Abra la carpeta **local** y cree dos carpetas **include** y **lib** si aún no está allí.

Ahora abra la carpeta GLFW y vaya a **Crear** (donde CMake había creado los archivos). Abra el archivo **GLFW.xcodeproj** en Xcode.



Seleccione **instalar> Mi Mac** y luego haga clic en **ejecutar** (botón Reproducir en forma).

Ahora está instalado con éxito (ignorar las advertencias).

Para asegurarse de que Open Finder y la carpeta goto **/ usr / local / lib** y tres archivos de la biblioteca GLFW ya estarán presentes allí (de lo contrario, abra la carpeta **Build** dentro de la carpeta GLFW y vaya a **src / Debug** copie todos los archivos a **/ usr / local / lib**)

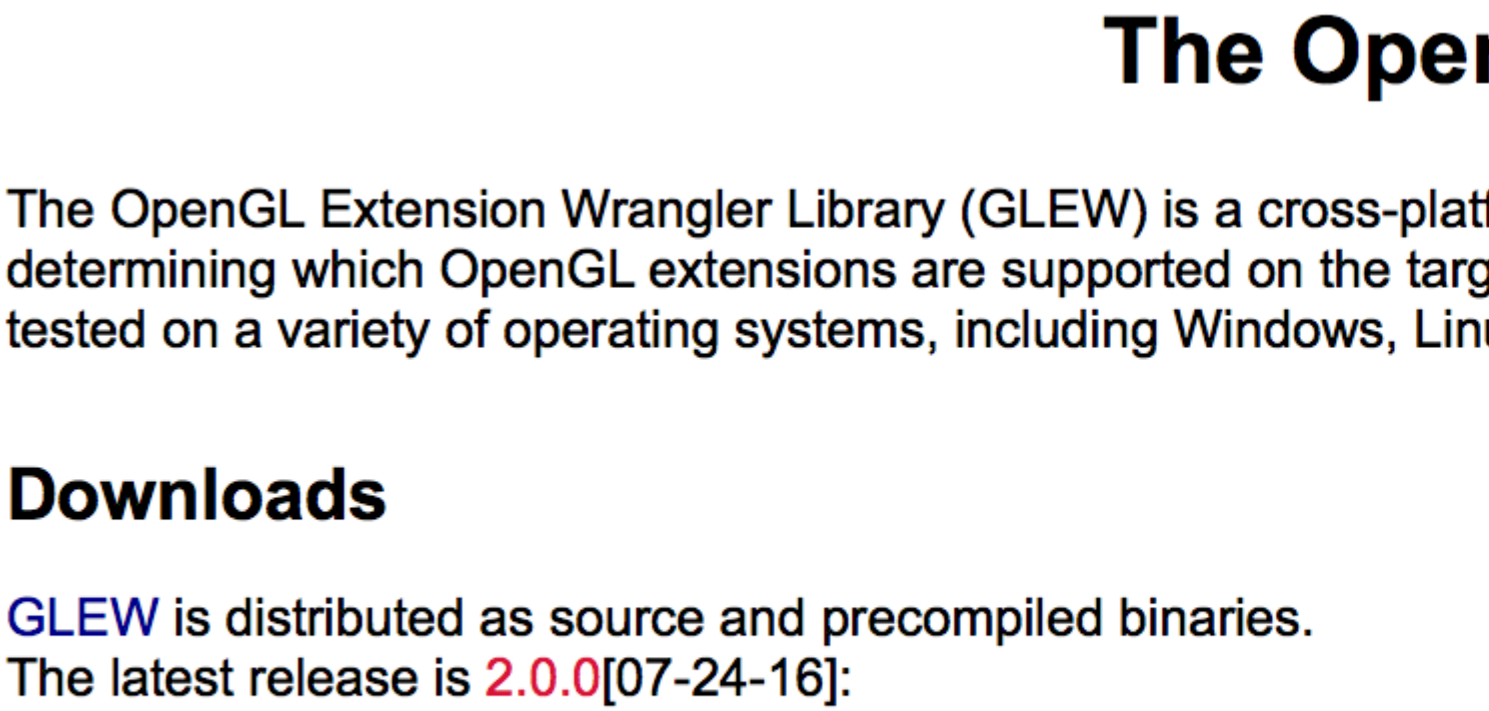
Open Finder y goto **/ usr / local / include** y una carpeta GLFW ya estarán presentes allí con dos archivos de encabezado dentro de él con el nombre de **glfw3.h** y **glfw3native.h**

glfw3.h

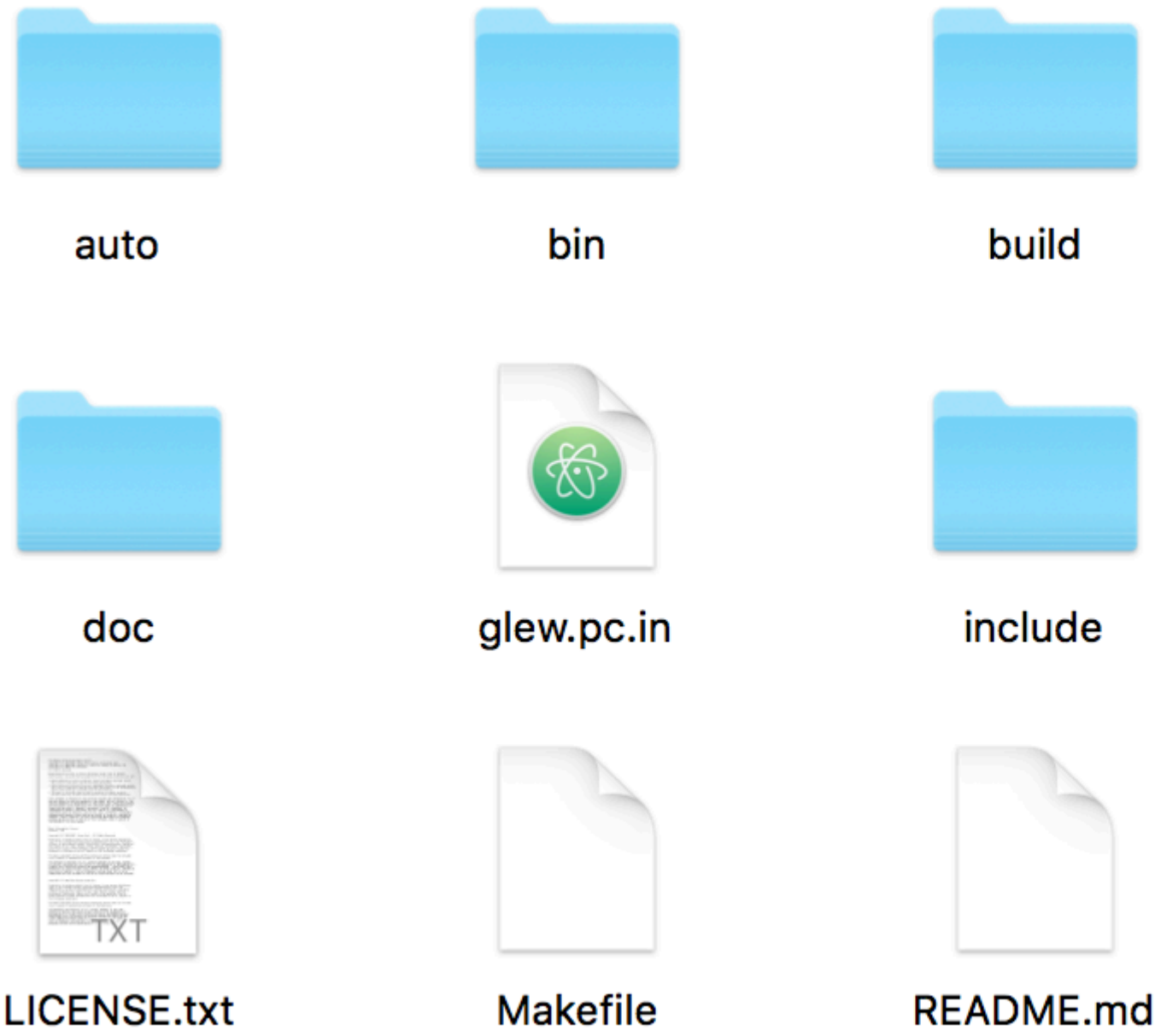
glfw3native.h

2. Instale GLEW

GLEW es una biblioteca multiplataforma que ayuda a consultar y cargar extensiones OpenGL. Proporciona mecanismos de tiempo de ejecución para determinar qué extensiones OpenGL son compatibles con la plataforma de destino. Es solo para OpenGL moderno (OpenGL versión 3.2 y superior que requiere que las funciones se determinen en tiempo de ejecución). Para instalar primero descargue sus archivos desde glew.sourceforge.net



Extraiga la carpeta GLFW y su contenido se verá así.



Ahora abra la Terminal, navegue a la carpeta GLEW y escriba los siguientes comandos

```
make
sudo make install
make clean
```

Ahora GLEW está instalado con éxito. Para asegurarse de que esté instalado, Open Finder, vaya a **/ usr / local / include** y una carpeta GL ya estará presente allí con tres archivos de encabezado dentro de él con el nombre de **glew.h** , **glxew.h** y **wglew.h**

glew.h

glxew.h

wglew.h

Abra el Finder y vaya a **/ usr / local / lib** y los archivos de la biblioteca GLEW ya estarán presentes allí

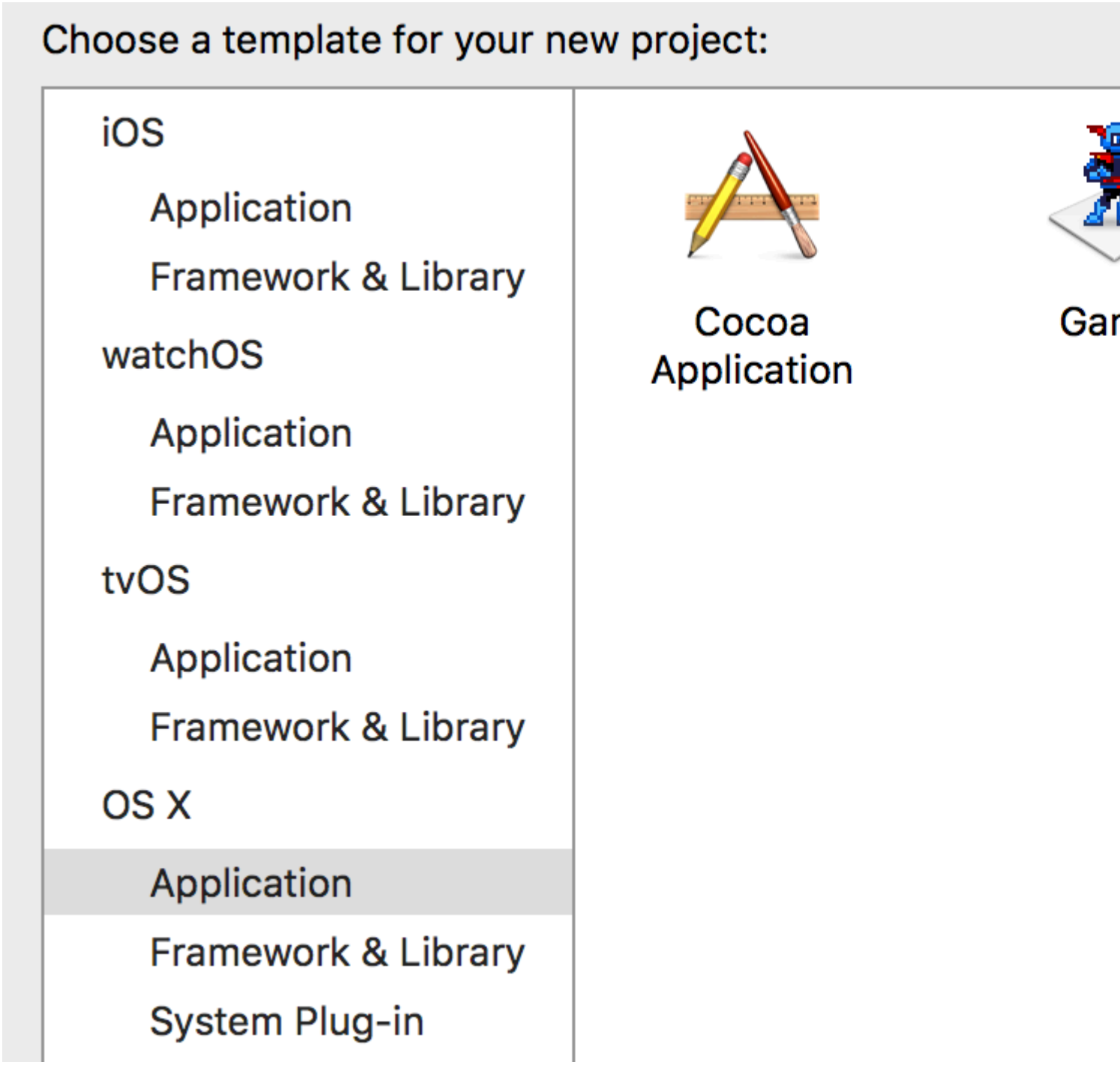
libGLEW.2.0.dylib

libGLEW.a

libGLEW.dylib

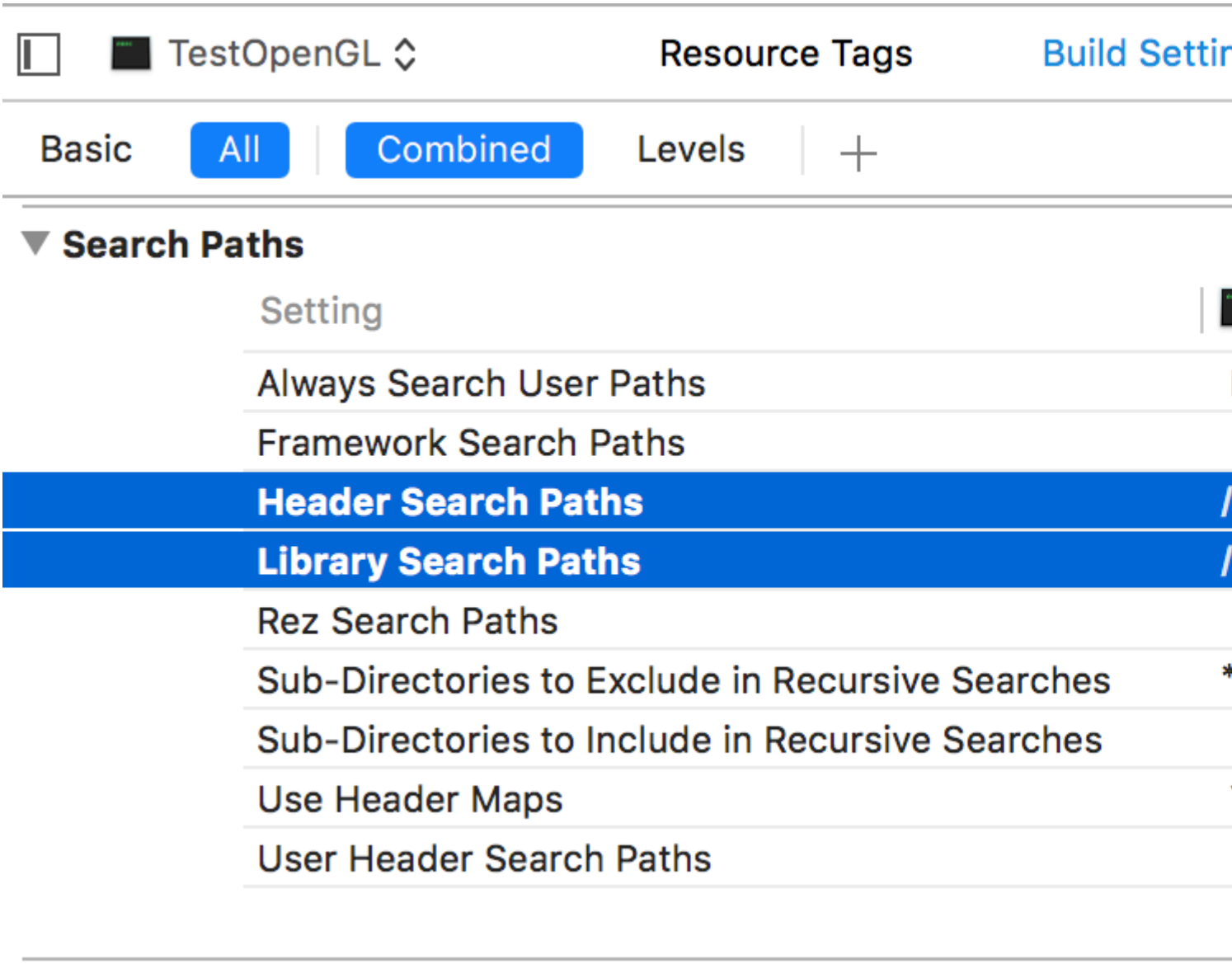
3. Probar y ejecutar

Ahora hemos instalado con éxito GLFW y GLEW. Es hora de codificar. Abre Xcode y crea un nuevo proyecto de Xcode. Seleccione la **herramienta de línea de comandos y** luego continúe y seleccione **C ++** como idioma.

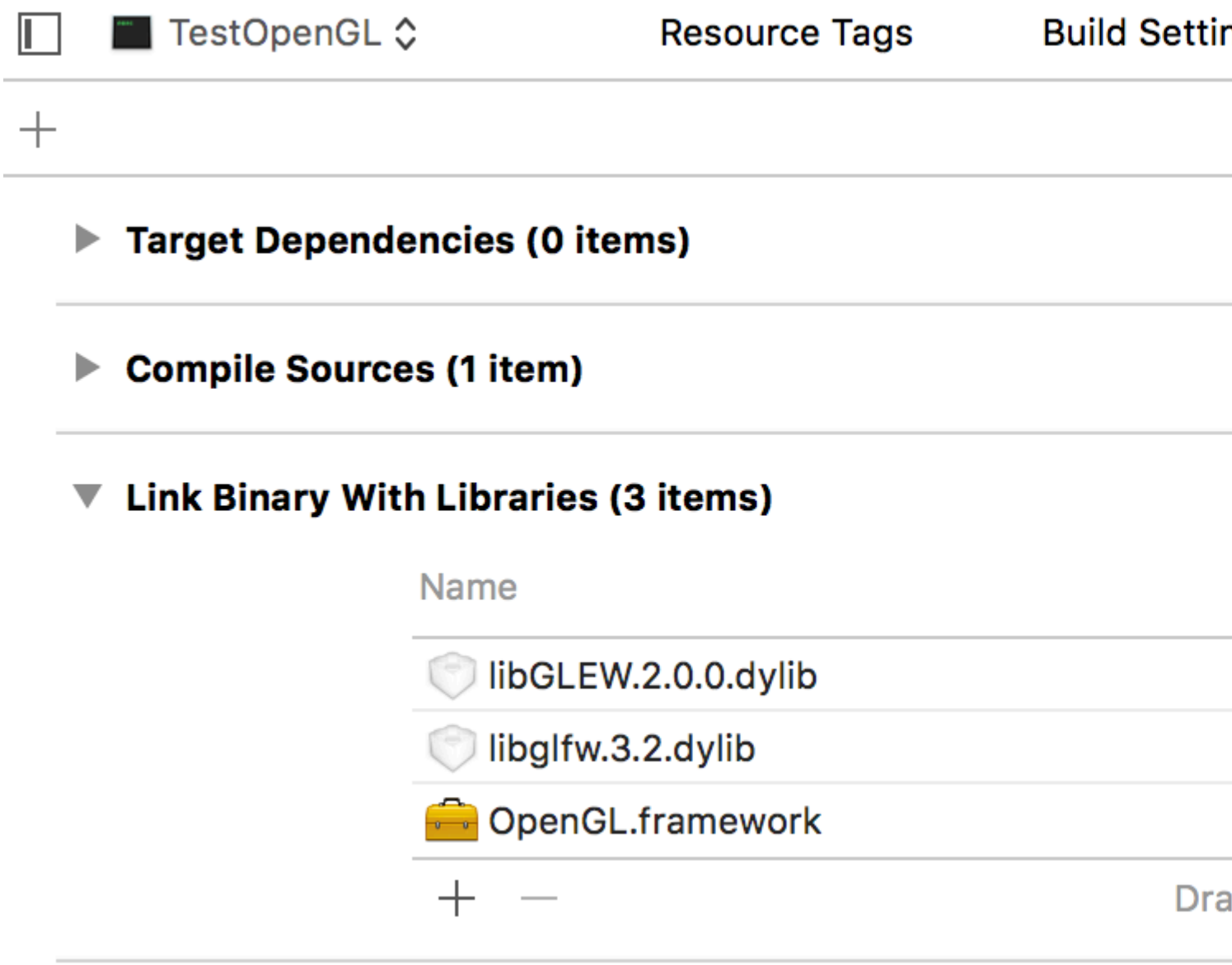


Xcode creará un nuevo proyecto de línea de comandos.

Haga clic en el nombre del proyecto y, en la pestaña **Configuraciones de compilación,** cambie de **Básico a Todo** , en la sección **Rutas de búsqueda** , agregue **/ usr / local / include** en Rutas de búsqueda de encabezado y agregue **/ usr / local / lib** en Rutas de búsqueda de biblioteca



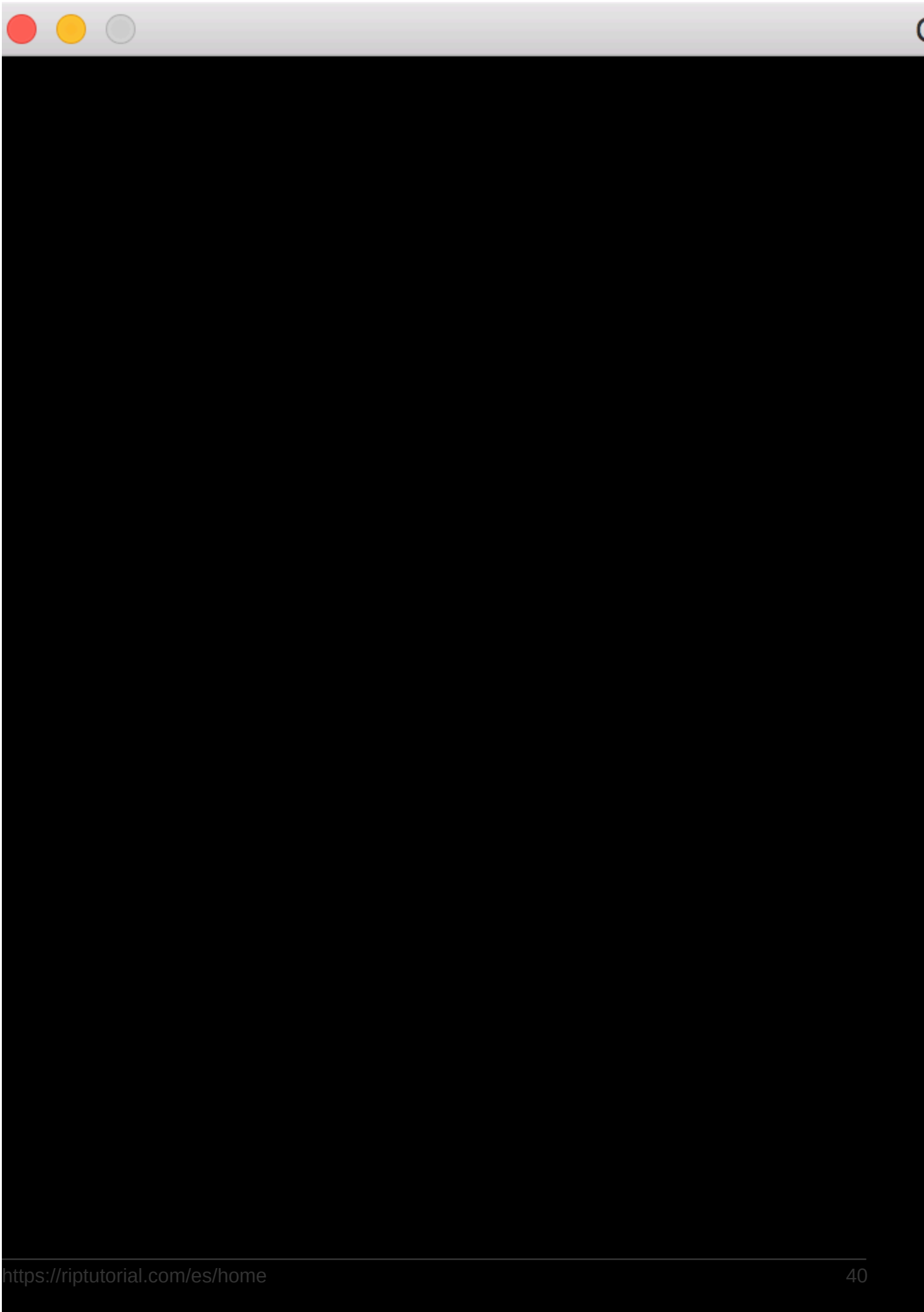
Haga clic en el nombre del proyecto y en la pestaña **Build Phases** y en **Link With Binary Libraries** agregue **OpenGL.framework** y también agregue las bibliotecas **GLFW** y **GLEW** creadas recientemente desde **/ usr / local / lib**



Ahora estamos listos para codificar en Modern Open GL 4.1 en macOS utilizando C ++ y Xcode. El siguiente código creará una ventana OpenGL usando GLFW con salida de pantalla en blanco.

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>

// Define main function
int main()
{
    // Initialize GLFW
    glfwInit();

    // Define version and compatibility settings
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
    glfwWindowHint(GLFW_OPENGL_PROFILE,GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

    // Create OpenGL window and context
    GLFWwindow* window = glfwCreateWindow(800, 600, "OpenGL", NULL, NULL);
    glfwMakeContextCurrent(window);
```

```
    // Check for window creation failure
    if (!window)
    {
        // Terminate GLFW
        glfwTerminate();
        return 0;
    }

    // Initialize GLEW
    glewExperimental = GL_TRUE; glewInit();

    // Event loop
    while(!glfwWindowShouldClose(window))
    {
        // Clear the screen to black
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f); glClear(GL_COLOR_BUFFER_BIT);
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // Terminate GLFW
    glfwTerminate(); return 0;
}
```



https://riptutorial.com/es/opengl/topic/814/empezando-con-opengl

Capítulo 2: Creación de contexto de OpenGL.

Examples

Creando una ventana básica

Este tutorial asume que ya tienes un entorno OpenGL en funcionamiento con todas las bibliotecas y encabezados necesarios disponibles.

```
#include <GL\glew.h>     //Include GLEW for function-pointers etc.
#include <GLFW\GLFW3.h>  //Include GLFW for windows, context etc.
                         //Important note: GLEW must NEVER be included after
                         //gl.h is already included(which is included in glew.h
                         //and glfw3.h) so if you want to include one of these
                         //headers again, wrap them
                         //into #ifndef __gl_h_ directives just to be sure

GLFWwindow* window;      //This is the GLFW handle for a window, it is used in several
                         //GLFW functions, so make sure it is accessible

void createWindow()
{
    glewExperimental = GL_TRUE; //This is required to use functions not included
                                //in opengl.lib

    if(!glfwInit())             //Inititalizes the library
        return;

    glfwDefaultWindowHints();   //See second example

    window = glfwCreateWindow(WIDTH, HEIGHT, title, NULL, NULL);
    //Creates a window with the following parameters:
    // + int width: Width of the window to be created
    // + int height: Height of the window to be created
    // + const char* title: Title of the window to be created
    // + GLFWmonitor* monitor: If this parameter is non-NULL, the window will be
    //   created in fullscreen mode, covering the specified monitor. Monitors can be
    //   queried using either glfwGetPrimaryMonitor() or glfwGetMonitors()
    // + GLFWwindow* share: For more information about this parameter, please
    //   consult the documentation

    glfwMakeContextCurrent(window);
    //Creates a OpenGL context for the specified window
    glfwSwapInterval(0);
    //Specifies how many monitor-refreshes GLFW should wait, before swapping the
    //backbuffer and the displayed frame. Also know as V-Sync
    glewInit();
    //Initializes GLEW
}
```

Añadiendo pistas a la ventana.

```
glfwDefaultWindowHints();
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
```

```
glfwWindowHint(GLFW_SAMPLES, 4);
glfwWindowHint(GLFW_VISIBLE, GL_FALSE);

//Window hints are used to manipulate the behavior of later created windows
//As the name suggests, they are only hints, not hard constraints, which means
//that there is no guarantee that they will take effect.
//GLFW_RESIZABLE controls if the window should be scalable by the user
//GLFW_SAMPLES specifies how many samples there should be used for MSAA
//GLFW_VISIBLE specifies if the window should be shown after creation. If false, it
//             can later be shown using glfwShowWindow()
//For additional windowhints please consult the documentation
```

Lea Creación de contexto de OpenGL. en línea:
https://riptutorial.com/es/opengl/topic/6194/creacion-de-contexto-de-opengl-

Capítulo 3: Encapsular objetos OpenGL con C ++ RAII

Introducción

Ejemplos de varias formas de hacer que los objetos OpenGL funcionen con C ++ RAII.

Observaciones

La encapsulación RAII de objetos OpenGL tiene peligros. Lo más inevitable es que los objetos OpenGL están asociados con el contexto OpenGL que los creó. Por lo tanto, la destrucción de un objeto RAII de C ++ debe realizarse en un contexto de OpenGL que comparta la propiedad del objeto OpenGL gestionado por ese objeto de C ++.

Esto también significa que si se destruyen todos los contextos que poseen el objeto, cualquier objeto OpenGL encapsulado RAII existente intentará destruir los objetos que ya no existen.

Debes tomar pasos manuales para lidiar con problemas de contexto como este.

Examples

En C ++ 98/03

Encapsular un objeto OpenGL en C ++ 98/03 requiere obedecer la regla de C ++ de 3. Esto significa agregar un constructor de copia, operador de asignación de copia y destructor.

Sin embargo, los constructores de copia deben copiar lógicamente el objeto. Y copiar un objeto OpenGL es una tarea no trivial. Igualmente importante, es casi seguro que es algo que el usuario no desea hacer.

Así que en su lugar haremos el objeto no copiable:

```
class BufferObject
{
public:
    BufferObject(GLenum target, GLsizeiptr size, const void *data, GLenum usage)
    {
        glGenBuffers(1, &object_);
        glBindBuffer(target, object_);
        glBufferData(target, size, data, usage);
        glBindBuffer(target, 0);
    }

    ~BufferObject()
    {
        glDeleteBuffers(1, &object_);
    }
```

```
    //Accessors and manipulators
    void Bind(GLenum target) const {glBindBuffer(target, object_);}
    GLuint GetObject() const {return object_;}

private:
    GLuint object_;

    //Prototypes, but no implementation.
    BufferObject(const BufferObject &);
    BufferObject &operator=(const BufferObject &);
};
```

El constructor creará el objeto e inicializará los datos del objeto del búfer. El destructor destruirá el objeto. Al declarar el constructor / asignación de copia *sin* definirlos, el vinculador dará un error si algún código intenta llamarlos. Y al declararlos privados, solo los miembros de `BufferObject` podrán llamarlos.

Tenga en cuenta que `BufferObject` no conserva el `target` pasado al constructor. Esto se debe a que un objeto de búfer OpenGL se puede usar con cualquier objetivo, no solo con el que se creó inicialmente. Esto es diferente a los objetos de textura, que siempre deben estar vinculados al objetivo con el que se crearon inicialmente.

Debido a que OpenGL es muy dependiente de los objetos de enlace al contexto para varios propósitos, a menudo es útil tener también un enlace de objeto de estilo RAII. Debido a que diferentes objetos tienen diferentes necesidades de enlace (algunos tienen objetivos, otros no), tenemos que implementar uno para cada objeto individualmente.

```
class BindBuffer
{
public:
    BindBuffer(GLenum target, const BufferObject &buff) : target_(target)
    {
        buff.Bind(target_);
    }

    ~BindBuffer()
    {
        glBindBuffer(target_, 0);
    }

private:
    GLenum target_;

    //Also non-copyable.
    BindBuffer(const BindBuffer &);
    BindBuffer &operator=(const BindBuffer &);
};
```

`BindBuffer` no se puede copiar, ya que copiarlo no tiene sentido. Tenga en cuenta que no retiene el acceso al objeto `BufferObject` se enlaza. Eso es porque es innecesario.

En C ++ 11 y posteriores.

C ++ 11 ofrece herramientas que mejoran la funcionalidad de los objetos OpenGL encapsulados en RAII. Sin las características de C ++ 11, como la semántica de movimiento, dichos objetos tendrían que asignarse dinámicamente si desea pasarlos, ya que no se pueden copiar. El soporte para mover permite que se pasen de un lado a otro como valores normales, aunque no copiando:

```
class BufferObject
{
public:
    BufferObject(GLenum target, GLsizeiptr size, const void *data, GLenum usage)
    {
        glGenBuffers(1, &object_);
        glBindBuffer(target, object_);
        glBufferData(target, size, data, usage);
        glBindBuffer(target, 0);
    }

    //Cannot be copied.
    BufferObject(const BufferObject &) = delete;
    BufferObject &operator=(const BufferObject &) = delete;

    //Can be moved
    BufferObject(BufferObject &&other) noexcept : object_(other.Release())
    {}

    //Self-assignment is OK with this implementation.
    BufferObject &operator=(BufferObject &&other) noexcept
    {
        Reset(other.Release());
    }

    //Destroys the old buffer and claims ownership of a new buffer object.
    //It's OK to call glDeleteBuffers on buffer object 0.
    GLuint Reset(GLuint object = 0)
    {
        glDeleteBuffers(1, &object_);
        object_ = object;
    }

    //Relinquishes ownership of the object without destroying it
    GLuint Release()
    {
        GLuint ret = object_;
        object_ = 0;
        return ret;
    }

    ~BufferObject()
    {
        Reset();
    }

    //Accessors and manipulators
    void Bind(GLenum target) const {glBindBuffer(target, object_);}
    GLuint GetObject() const {return object_;}

private:
    GLuint object_;
};
```

Este tipo puede ser devuelto por una función:

```
BufferObject CreateStaticBuffer(GLsizeiptr byteSize) {return BufferObject(GL_ARRAY_BUFFER,
byteSize, nullptr, GL_STATIC_DRAW);}
```

Lo que le permite almacenarlos en sus propios tipos (implícitamente de solo movimiento):

```
struct Mesh
{
public:
private:
    //Default member initializer.
    BufferObject buff_ = CreateStaticBuffer(someSize);
};
```

Una clase de enlace con ámbito también puede tener semántica de movimiento, lo que permite que el enlace se devuelva desde las funciones y se almacene en contenedores de biblioteca estándar de C ++:

```
class BindBuffer
{
public:
    BindBuffer(GLenum target, const BufferObject &buff) : target_(target)
    {
        buff.Bind(target_);
    }

    //Non-copyable.
    BindBuffer(const BindBuffer &) = delete;
    BindBuffer &operator=(const BindBuffer &) = delete;

    //Move-constructible.
    BindBuffer(BindBuffer &&other) noexcept : target_(other.target_)
    {
        other.target_ = 0;
    }

    //Not move-assignable.
    BindBuffer &operator=(BindBuffer &&) = delete;

    ~BindBuffer()
    {
        //Only unbind if not moved from.
        if(target_)
            glBindBuffer(target_, 0);
    }

private:
    GLenum target_;
};
```

Tenga en cuenta que el objeto se puede mover pero no se puede asignar. La idea con esto es evitar la reconexión de un enlace de búfer de ámbito. Una vez que se configura, la única cosa que lo puede desactivar es que se está moviendo.

Lea Encapsular objetos OpenGL con C ++ RAII en línea:

https://riptutorial.com/es/opengl/topic/7556/encapsular-objetos-opengl-con-c-plusplus-raii

Capítulo 4: Framebuffers

Examples

Fundamentos de framebuffers

Framebuffer es un tipo de búfer que almacena **los** valores de **color** , la **profundidad** y la información de la **plantilla** de los píxeles en la memoria. Cuando dibuja algo en OpenGL, la salida se almacena en el *framebuffer predeterminado* y luego se ven los valores de color de este búfer en la pantalla. También puede crear su propio framebuffer que se puede usar para muchos efectos de posprocesamiento geniales, como *escala de grises, desenfoque, profundidad de campo, distorsiones, reflejos* ...

Para comenzar necesitas crear un objeto de framebuffer (**FBO**) y enlazarlo como cualquier otro objeto en OpenGL:

```
unsigned int FBO;
glGenFramebuffers(1, &FBO);
glBindFramebuffer(GL_FRAMEBUFFER, FBO);
```

Ahora debe agregar al menos un **archivo adjunto** (color, profundidad o plantilla) al framebuffer. Un archivo adjunto es una ubicación de memoria que actúa como un búfer para el framebuffer. Puede ser una *textura* o un *objeto renderbuffer* . La ventaja de usar una textura es que puede usar esta textura fácilmente en un sombreado de post-procesamiento. Crear la textura es similar a una textura normal:

```
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

El `width` y la `height` deben ser iguales al tamaño de la ventana de renderizado. El puntero de datos de textura es `NULL` porque solo desea asignar la memoria y no rellenar la textura con ningún dato. La textura está lista, por lo que puedes adjuntarla al framebuffer:

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);
```

Su framebuffer debería estar listo para usar ahora, pero es posible que también desee agregar adjuntos de profundidad o adjuntos de profundidad y de plantilla. Si desea agregarlos como archivos adjuntos de textura (y usarlos para algún procesamiento) puede crear otras texturas como las de arriba. La única diferencia sería en estas líneas:

```
glTexImage2D(
    GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, width, height, 0,
    GL_DEPTH_COMPONENT, GL_FLOAT, NULL
);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, texture, 0);
```

O estos si desea utilizar profundidad **y** apego de plantilla en una sola textura:

```
glTexImage2D(
    GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, width, height, 0,
    GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL
);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_TEXTURE_2D, texture,
0);
```

También puede usar un **tampón** de **renderizado en** lugar de una textura como un archivo adjunto para los búferes de profundidad y de plantilla si no desea procesar los valores más adelante. (Se explicará en otro ejemplo ...)

Puede comprobar si el framebuffer se ha creado y completado correctamente sin ningún error:

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) == GL_FRAMEBUFFER_COMPLETE)
    // do something...
```

Y, por último, no olvides desenlazar el framebuffer para que no le rindas accidentalmente:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Límites

La función OGL glGetIntegerv puede determinar el número máximo de búferes de color que se pueden adjuntar a un solo búfer de cuadro mediante el parámetro `GL_MAX_COLOR_ATTACHMENTS` :

```
GLint maxColAttchments = 0;
glGetIntegerv( GL_MAX_COLOR_ATTACHMENTS, &maxColAttchments );
```

Usando el framebuffer

El uso es bastante sencillo. En primer lugar, unes tu **framebuffer** y renderizas tu escena en él. Pero aún no verás nada porque tu renderbuffer no es visible. Por lo tanto, la segunda parte es hacer que el framebuffer sea una textura de un cuadrante de pantalla completa en la pantalla. Solo puedes renderizarlo como está o hacer algunos efectos de post-procesamiento.

Aquí están los vértices para un quad de pantalla completa:

```
float vertices[] = {
//   positions     texture coordinates
```

```
    -1.0f,  1.0f,  0.0f, 1.0f,
    -1.0f, -1.0f,  0.0f, 0.0f,
     1.0f, -1.0f,  1.0f, 0.0f,

    -1.0f,  1.0f,  0.0f, 1.0f,
     1.0f, -1.0f,  1.0f, 0.0f,
     1.0f,  1.0f,  1.0f, 1.0f
};
```

Deberá almacenarlos en un VBO o renderizar utilizando punteros de atributos. También necesitarás un programa de sombreado básico para representar el cuadrante de pantalla completa con textura.

Sombreador de vértices:

```
in vec2 position;
in vec2 texCoords;

out vec2 TexCoords;

void main()
{
    gl_Position = vec4(position.x, position.y, 0.0, 1.0);
    TexCoords = texCoords;
}
```

Sombreador de fragmentos:

```
in vec2 TexCoords;
out vec4 color;

uniform sampler2D screenTexture;

void main()
{
    color = texture(screenTexture, TexCoords);
}
```

Nota: Es posible que deba ajustar los sombreadores para su versión de *GLSL* .

Ahora puede hacer la representación real. Como se describió anteriormente, lo primero es renderizar la escena en su FBO. Para hacerlo, simplemente enlace su FBO, bórrelo y dibuje la escena:

```
glBindFramebuffer(GL_FRAMEBUFFER, FBO);
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// draw your scene here...
```

Nota: En la función `glClear` debe especificar todos los archivos adjuntos de framebuffer que está utilizando (en este ejemplo, el archivo de color y profundidad).

Ahora puedes renderizar tu FBO como un cuadrante de pantalla completa en el framebuffer predeterminado para que puedas verlo. Para hacer esto, simplemente desvincula tu FBO y

renderiza el quad:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0); // unbind your FBO to set the default framebuffer
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

shader.Use(); // shader program for rendering the quad

glBindTexture(GL_TEXTURE_2D, texture); // color attachment texture
glBindBuffer(GL_ARRAY_BUFFER, VBO); // VBO of the quad
// You can also use VAO or attribute pointers instead of only VBO...
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

¡Y eso es todo! Si has hecho todo correctamente, deberías ver la misma escena que antes pero renderizada en un quad de pantalla completa. La salida visual es la misma que antes, pero ahora puede agregar fácilmente efectos de posprocesamiento simplemente editando el sombreador de fragmentos. (Agregaré efectos en otro ejemplo (s) y lo vincularé aquí)

Lea Framebuffers en línea: https://riptutorial.com/es/opengl/topic/7038/framebuffers

Capítulo 5: Iluminación básica

Examples

Modelo de iluminación phong

> NOTA: Este ejemplo es WIP, se actualizará con diagramas, imágenes, más ejemplos, etc.

¿Qué es Phong?

Phong es un modelo de luz muy básico, pero de aspecto real para superficies que tiene tres partes: iluminación ambiental, difusa y especular.

Iluminación ambiental:

La iluminación ambiental es la más simple de las tres partes para entender y calcular. La iluminación ambiental es una luz que inunda la escena e ilumina el objeto uniformemente en todas las direcciones.

Las dos variables en la iluminación ambiental son la fuerza del ambiente y el color del ambiente. En tu sombreador de fragmentos, lo siguiente funcionará para el ambiente:

```
in vec3 objColor;

out vec3 finalColor;

uniform vec3 lightColor;

void main() {
   float ambientStrength = 0.3f;
   vec3 ambient = lightColor * ambientStrength;
   finalColor = ambient * objColor;
}
```

Iluminación difusa:

La iluminación difusa es un poco más compleja que la ambiental. La luz difusa es luz direccional, lo que significa que las caras que miran hacia la fuente de luz estarán mejor iluminadas y las que apuntan hacia afuera se oscurecerán debido a la forma en que la luz las golpea.

Nota: la iluminación difusa requerirá el uso de normales para cada cara, que no mostraré cómo calcular aquí. Si quieres aprender cómo hacerlo, consulta la página de matemáticas 3D.

Para modelar el reflejo de la luz en gráficos de computadora se utiliza una función de distribución de reflectancia bidireccional (BRDF). BRDF es una función que proporciona la relación entre la luz reflejada a lo largo de una dirección saliente y la luz incidente de una dirección entrante.

Una superficie difusa perfecta tiene un BRDF que tiene el mismo valor para todas las direcciones

incidentes y salientes. Esto reduce sustancialmente los cálculos y, por lo tanto, se usa comúnmente para modelar superficies difusas, ya que es físicamente plausible, aunque no haya materiales difusos puros en el mundo real. Este BRDF se llama reflexión de Lambert porque obedece la ley de coseno de Lambert.

La reflexión de Lambert se utiliza a menudo como modelo para la reflexión difusa. Esta técnica hace que todos los polígonos cerrados (como un triángulo dentro de una malla 3D) reflejen la luz por igual en todas las direcciones cuando se procesa. El coeficiente de difusión se calcula a partir del ángulo entre el vector normal y el vector de luz.

```
f_Lambertian = max( 0.0, dot( N, L )
```

donde `N` es el vector normal de la superficie y `L` es el vector hacia la fuente de luz.

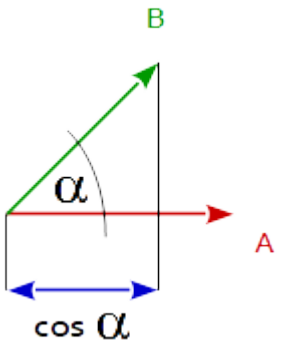
Cómo funciona

En general, el producto de *punto* de 2 vectores es igual al *coseno* del ángulo entre los 2 vectores multiplicado por la magnitud (longitud) de ambos vectores.

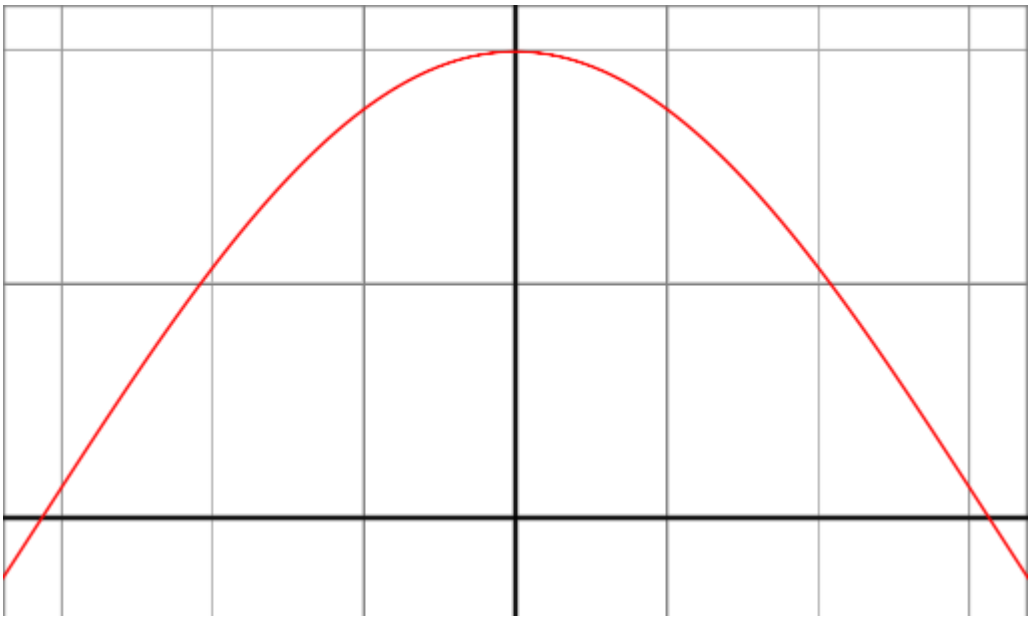
```
dot( A, B ) == length( A ) * length( B ) * cos( angle_A_B )
```

De esto se deduce que el producto de *puntos* de 2 vectores unitarios es igual al *coseno* del ángulo entre los 2 vectores, porque la longitud de un vector unitario es 1.

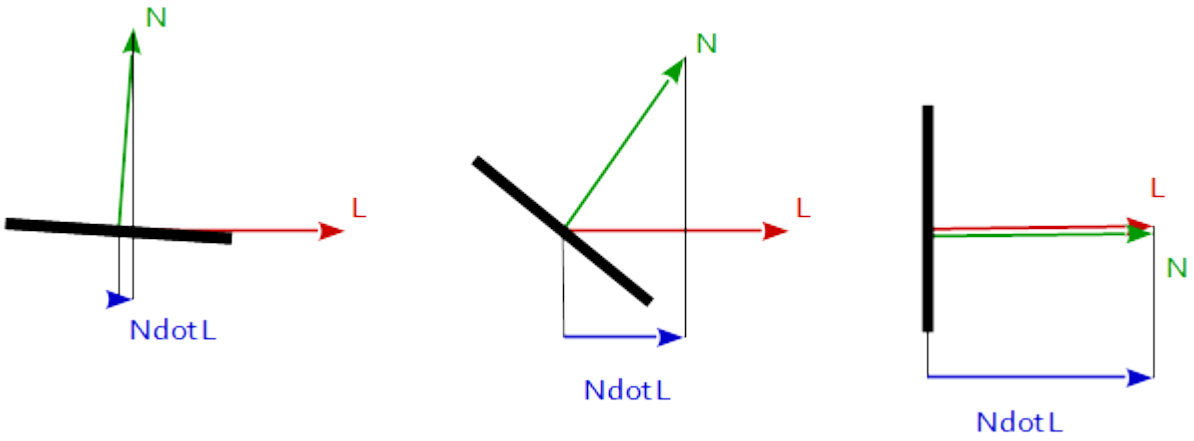
```
uA = normalize( A )
uB = normalize( B )
cos( angle_A_B ) == dot( uA, uB )
```



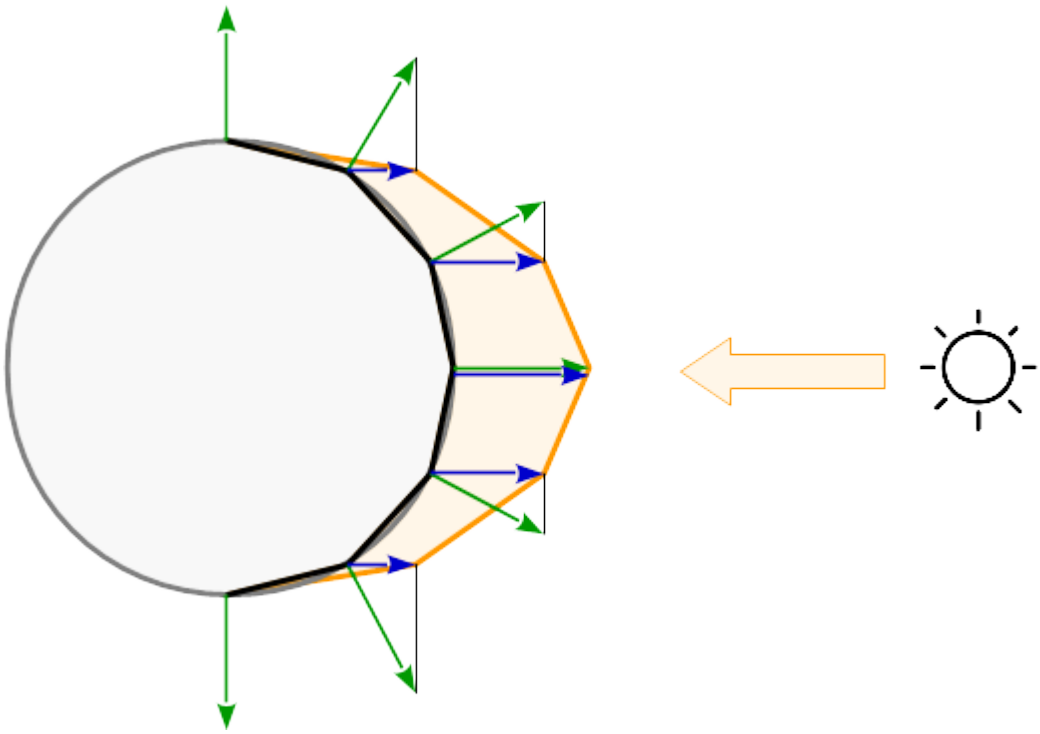
Si observamos la función *cos (x)* entre los ángulos -90 ° y 90 °, podemos ver que tiene un máximo de 1 en un ángulo de 0 ° y baja a 0 en los ángulos de 90 ° y -90 °.



Este comportamiento es exactamente lo que queremos para el modelo de reflexión. Cuando el vector nromal de la superficie y la dirección a la fuente de luz están en la misma dirección (el ángulo entre es 0 °), queremos un máximo de reflexión. En contraste, si los vectores son ortonormalizados (el ángulo intermedio es de 90 °), entonces queremos un mínimo de reflexión y queremos un funcionamiento funcional suave y continuo entre los dos bordes de 0 ° y 90 °.



Si la luz se calcula por vértice, la reflexión se calcula para cada esquina de la primitiva. Entre las primitivas, las reflexiones se interpolan según sus coordenadas baricéntricas. Ver las reflexiones resultantes sobre una superficie esférica:



Ok, para comenzar con nuestro sombreador de fragmentos, necesitaremos cuatro entradas.

- Normales de vértices (deben estar en el búfer y especificados por punteros de atributo de vértice)
- Posición del fragmento (debe salir del sombreado de vértice en el sombreador de fragmentos)
- Posición de la fuente de luz (uniforme)
- Color claro (uniforme)

```
in vec3 normal;
in vec3 fragPos;

out vec3 finalColor;

uniform vec3 lightColor;
uniform vec3 lightPos;
uniform vec3 objColor;
```

Dentro de main es donde necesitamos hacer algunos cálculos. Todo el concepto de iluminación difusa se basa en el ángulo entre la dirección normal y la luz. Cuanto mayor es el ángulo, menos luz hay hasta 90 ° donde no hay luz en absoluto.

Antes de que podamos comenzar a calcular la cantidad de luz, necesitamos el vector de dirección de la luz. Esto se puede recuperar simplemente restando la posición de la luz de la posición del fragmento que devuelve un vector de la posición de la luz apuntando a la posición del fragmento.

```
vec3 lightDir = lightPos-fragPos;
```

Además, siga adelante y normalice los vectores `normal` y `lightDir` para que tengan la misma longitud con la que trabajar.

```
normal  = normalize(normal);
lightDir = normalize(lightDir);
```

Ahora que tenemos nuestros vectores, podemos calcular la diferencia entre ellos. Para hacer esto, vamos a utilizar la función de producto punto. Básicamente, esto toma 2 vectores y devuelve el cos () del ángulo formado. Esto es perfecto porque a 90 grados producirá 0 y a 0 grados dará 1. Como resultado, cuando la luz apunta directamente al objeto, estará completamente iluminado y viceversa.

```
float diff = dot(normal, lightDir);
```

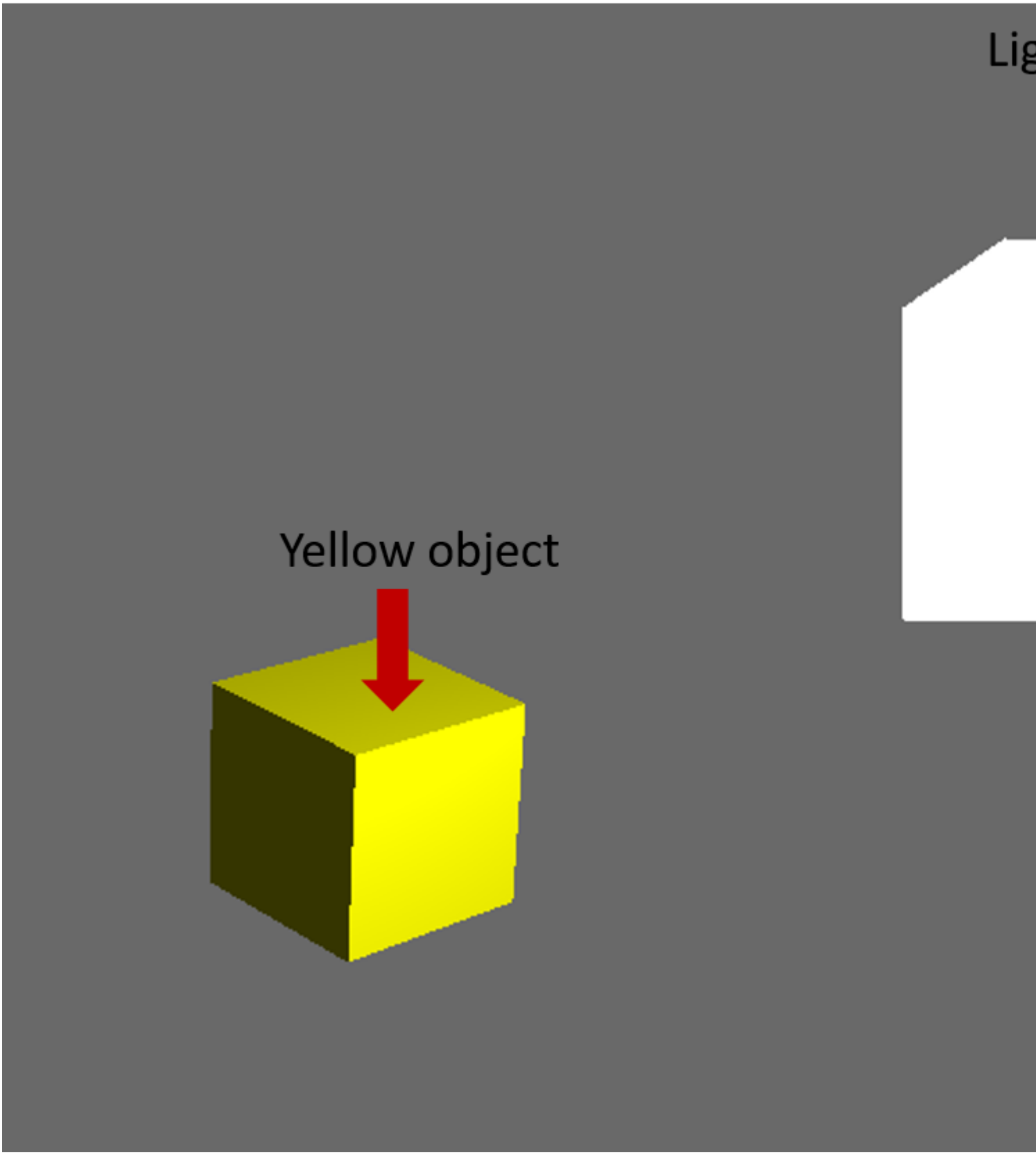
Hay una cosa más que tenemos que hacer con el número calculado, tenemos que asegurarnos de que siempre sea positivo. Si lo piensas, un número negativo no tiene sentido en contexto porque eso significa que la luz está detrás de la cara. Podríamos usar una sentencia if, o podemos usar la función `max()` que devuelve el máximo de dos entradas.

```
diff = max(diff, 0.0);
```

Una vez hecho esto, ahora estamos listos para calcular el color de salida final para el fragmento.

```
vec3 diffuse = diff * lightColor;
finalColor = diffuse * objColor;
```

Debe tener un aspecto como este:



Iluminación especular:

Trabajo en progreso, vuelva a consultar más tarde.

Conjunto

Trabajo en progreso, vuelva a consultar más tarde.

El siguiente código e imagen muestran estos tres conceptos de iluminación combinados.

Lea Iluminación básica en línea: https://riptutorial.com/es/opengl/topic/4209/iluminacion-basica

Capítulo 6: Instancia

Introducción

La creación de instancias es una técnica de representación que nos permite dibujar varias copias del mismo objeto en una llamada de sorteo. Generalmente se usa para hacer partículas, follaje o grandes cantidades de cualquier otro tipo de objetos.

Examples

Instancing by Vertex Attribute Arrays

3.3

La creación de instancias se puede realizar a través de modificaciones en la forma en que se proporcionan los atributos de vértice al sombreado de vértice. Esto introduce una nueva forma de acceder a las matrices de atributos, permitiéndoles proporcionar datos por instancia que parecen un atributo regular.

Una sola instancia representa un objeto o grupo de vértices (una hoja de hierba, etc.). Los atributos asociados con matrices de instancias solo avanzan entre las instancias; a diferencia de los atributos de vértice regular, no obtienen un nuevo valor por vértice.

Para especificar que una matriz de atributos es instanciada, use esta llamada:

```
glVertexAttribDivisor(attributeIndex, 1);
```

Esto establece el estado del objeto de matriz de vértice. El "1" significa que el atributo es avanzado para cada instancia. Al pasar un 0 se desactiva la creación de instancias para el atributo.

En el sombreador, el atributo instanciado se parece a cualquier otro atributo de vértice:

```
in vec3 your_instanced_attribute;
```

Para representar varias instancias, puede invocar una de las formas `Instanced` del valor de las llamadas `glDraw*` . Por ejemplo, esto dibujará 1000 instancias, y cada instancia constará de 3 vértices:

```
glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 1000);
```

Código de matriz instanciado

Configuración de VAOs, VBOs y los atributos:

```
// List of 10 triangle x-offsets (translations)
GLfloat translations[10];
GLint index = 0;
for (GLint x = 0; x < 10; x++)
{
    translations[index++] = (GLfloat)x / 10.0f;
}

// vertices
GLfloat vertices[] = {
     0.0f,   0.05f,
     0.05f, -0.05f,
    -0.05f, -0.05f,
     0.0f,  -0.1f,
};

// Setting VAOs and VBOs
GLuint meshVAO, vertexVBO, instanceVBO;
glGenVertexArrays(1, &meshVAO);
glGenBuffers(1, &instanceVBO);
glGenBuffers(1, &vertexVBO);

glBindVertexArray(meshVAO);

    glBindBuffer(GL_ARRAY_BUFFER, vertexVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat), (GLvoid*)0);

    glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(translations), translations, GL_STATIC_DRAW);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 1, GL_FLOAT, GL_FALSE, sizeof(GLfloat), (GLvoid*)0);
    glVertexAttribDivisor(1, 1); // This sets the vertex attribute to instanced attribute.

    glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindVertexArray(0);
```

Dibujar llamada:

```
glBindVertexArray(meshVAO);
glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, 4, 10); // 10 diamonds, 4 vertices per instance
glBindVertexArray(0);
```

Sombreador de vértices:

```
#version 330 core
layout(location = 0) in vec2 position;
layout(location = 1) in float offset;

void main()
{
    gl_Position = vec4(position.x + offset, position.y, 0.0, 1.0);
}
```

Sombreador de fragmentos:

```
#version 330 core
layout(location = 0) out vec4 color;

void main()
{
    color = vec4(1.0, 1.0, 1.0, 1.0f);
}
```

Lea Instancia en línea: https://riptutorial.com/es/opengl/topic/8647/instancia

Capítulo 7: Matemáticas 3d

Examples

Introducción a las matrices.

Cuando estés programando en OpenGL o en cualquier otra api de gráficos, chocarás contra una pared de ladrillos cuando no seas tan bueno en matemáticas. Aquí explicaré con código de ejemplo cómo puede lograr movimiento / escalado y muchas otras cosas interesantes con su objeto 3D.

Tomemos un caso de la vida real ... Has creado un cubo increíble (tridimensional) en OpenGL y quieres moverlo en cualquier dirección.

```
glUseProgram(cubeProgram)
glBindVertexArray(cubeVAO)
glEnableVertexAttribArray ( 0 );
glDrawArrays ( GL_TRIANGLES, 0,cubeVerticesSize)
```

En los motores de juego como Unity3d esto sería fácil. Simplemente llamaría transform.Translate () y terminaría con él, pero OpenGL no incluye una biblioteca matemática.

Una buena biblioteca de matemáticas es glm, pero para transmitir mi punto codificaré todos los (importantes) métodos matemáticos para que salgas.

Primero debemos entender que un objeto 3d en OpenGL contiene mucha información, hay muchas variables que dependen unas de otras. Una forma inteligente de gestionar todas estas variables es mediante el uso de matrices.

Una matriz es una colección de variables escritas en columnas y filas. Una matriz puede ser 1x1, 2x4 o cualquier número arbitrario.

```
[1|2|3]
[4|5|6]
[7|8|9] //A 3x3 matrix
```

Puedes hacer cosas realmente geniales con ellos ... pero, ¿cómo pueden ayudarme a mover mi cubo? Para entender esto, primero necesitamos saber varias cosas.

- ¿Cómo se hace una matriz desde una posición?
- ¿Cómo se traduce una matriz?
- ¿Cómo se pasa a OpenGL?

Hagamos una clase que contenga todos nuestros datos y métodos de matriz importantes (escritos en c ++)

```
template<typename T>
```

```
//Very simple vector containing 4 variables
struct Vector4{
    T x, y, z, w;
    Vector4(T x, T y, T z, T w) : x(x), y(y), z(z), w(w){}
    Vector4(){}

    Vector4<T>& operator=(Vector4<T> other){
        this->x = other.x;
        this->y = other.y;
        this->z = other.z;
        this->w = other.w;
        return *this;
    }
}

template<typename T>
struct Matrix4x4{
     /*!
     *  You see there are columns and rows like this
     */
    Vector4<T> row1,row2,row3,row4;

    /*!
     *  Initializes the matrix with a identity matrix. (all zeroes except the ones diagonal)
     */
    Matrix4x4(){
        row1 = Vector4<T>(1,0,0,0);
        row2 = Vector4<T>(0,1,0,0);
        row3 = Vector4<T>(0,0,1,0);
        row4 = Vector4<T>(0,0,0,1);
    }

    static Matrix4x4<T> identityMatrix(){
        return Matrix4x4<T>(
                        Vector4<T>(1,0,0,0),
                        Vector4<T>(0,1,0,0),
                        Vector4<T>(0,0,1,0),
                        Vector4<T>(0,0,0,1));
    }



    Matrix4x4(const Matrix4x4<T>& other){
        this->row1 = other.row1;
        this->row2 = other.row2;
        this->row3 = other.row3;
        this->row4 = other.row4;
    }

    Matrix4x4(Vector4<T> r1, Vector4<T> r2, Vector4<T> r3, Vector4<T> r4){
        this->row1 = r1;
        this->row2 = r2;
        this->row3 = r3;
        this->row4 = r4;
    }

      /*!
     *  Get all the data in an Vector
     *  @return rawData The vector with all the row data
     */
    std::vector<T> getRawData() const{
```

```
        return{
            row1.x,row1.y,row1.z,row1.w,
            row2.x,row2.y,row2.z,row2.w,
            row3.x,row3.y,row3.z,row3.w,
            row4.x,row4.y,row4.z,row4.w
        };
    }

}
```

Primero notamos algo muy peculiar en el constructor por defecto de una matriz de 4 por 4. Cuando se llama, no comienza todo en cero, pero como:

```
[1|0|0|0]
[0|1|0|0]
[0|0|1|0]
[0|0|0|1] //A identity 4 by 4 matrix
```

Todas las matrices deben comenzar con unas en diagonal. (solo porque>. <)

Bien, entonces declaremos en nuestro cubo épico una matriz de 4 por 4.

```
glUseProgram(cubeProgram)
Matrix4x4<float> position;
glBindVertexArray(cubeVAO)
glUniformMatrix4fv(shaderRef, 1, GL_TRUE, cubeData);
glEnableVertexAttribArray ( 0 );
glDrawArrays ( GL_TRIANGLES, 0,cubeVerticesSize)
```

Ahora que tenemos todas nuestras variables, ¡finalmente podemos comenzar a hacer algunos cálculos! Vamos a hacer la traducción. Si ha programado en Unity3d, puede recordar una función Transform.Translate. Vamos a implementarlo en nuestra propia clase de matriz.

```
 /*!
 *  Translates the matrix to
 *  @param vector, The vector you wish to translate to
 */
static Matrix4x4<T> translate(Matrix4x4<T> mat, T x, T y, T z){
    Matrix4x4<T> result(mat);
    result.row1.w += x;
    result.row2.w += y;
    result.row3.w += z;
    return result;
}
```

Esta es toda la matemática necesaria para mover el cubo (no la rotación ni la escala), funciona en todos los ángulos. Implementemos esto en nuestro escenario de la vida real.

```
glUseProgram(cubeProgram)
Matrix4x4<float> position;
position = Matrix4x4<float>::translate(position, 1,0,0);
glBindVertexArray(cubeVAO)
glUniformMatrix4fv(shaderRef, 1, GL_TRUE, &position.getRawData()[0]);
glEnableVertexAttribArray ( 0 );
```

```
glDrawArrays ( GL_TRIANGLES, 0,cubeVerticesSize)
```

Nuestro shader necesita usar nuestra maravillosa matriz.

```
#version 410 core
uniform mat4 mv_matrix;
layout(location = 0) in vec4 position;

void main(void){
    gl_Position = v_matrix * position;
}
```

Y debería funcionar ... pero parece que ya tenemos un error en nuestro programa. Cuando te mueves a lo largo del eje z, tu objeto parece desaparecer en el aire. Esto es porque no tenemos una matriz de proyección. Para resolver este error necesitamos saber dos cosas:

1. ¿Cómo se ve una matriz de proyección?
2. ¿Cómo podemos combinarlo con nuestra matriz de posición?

Bueno, podemos hacer una perspectiva (estamos usando tres dimensiones después de todo) matriz El código

```
template<typename T>
Matrix4x4<T> perspective(T fovy, T aspect, T near, T far){

    T q = 1.0f / tan((0.5f * fovy) * (3.14 / 180));
    T A = q / aspect;
    T B = (near + far) / (near - far);
    T C = (2.0f * near * far) / (near - far);

    return Matrix4x4<T>(
        Vector4<T>(A,0,0,0),
        Vector4<T>(0,q,0,0),
        Vector4<T>(0,0,B,-1),
        Vector4<T>(0,0,C,0));
}
```

Parece aterrador, pero este método en realidad calculará una matriz de cuán lejos desea mirar en la distancia (y qué tan cerca) y su campo de visión.

Ahora tenemos una matriz de proyección y una matriz de posición ... ¿Pero cómo los combinamos? Bueno, lo divertido es que en realidad podemos multiplicar dos matrices entre sí.

```
/*!
 *  Multiplies a matrix with an other matrix
 *  @param other, the matrix you wish to multiply with
 */
static Matrix4x4<T> multiply(const Matrix4x4<T>& first,const Matrix4x4<T>& other){
    //generate temporary matrix
    Matrix4x4<T> result;
    //Row 1
    result.row1.x = first.row1.x * other.row1.x + first.row1.y * other.row2.x + first.row1.z *
other.row3.x + first.row1.w * other.row4.x;
    result.row1.y = first.row1.x * other.row1.y + first.row1.y * other.row2.y + first.row1.z *
```

```
other.row3.y + first.row1.w * other.row4.y;
    result.row1.z = first.row1.x * other.row1.z + first.row1.y * other.row2.z + first.row1.z *
other.row3.z + first.row1.w * other.row4.z;
    result.row1.w = first.row1.x * other.row1.w + first.row1.y * other.row2.w + first.row1.z *
other.row3.w + first.row1.w * other.row4.w;

    //Row2
    result.row2.x = first.row2.x * other.row1.x + first.row2.y * other.row2.x + first.row2.z *
other.row3.x + first.row2.w * other.row4.x;
    result.row2.y = first.row2.x * other.row1.y + first.row2.y * other.row2.y + first.row2.z *
other.row3.y + first.row2.w * other.row4.y;
    result.row2.z = first.row2.x * other.row1.z + first.row2.y * other.row2.z + first.row2.z *
other.row3.z + first.row2.w * other.row4.z;
    result.row2.w = first.row2.x * other.row1.w + first.row2.y * other.row2.w + first.row2.z *
other.row3.w + first.row2.w * other.row4.w;

    //Row3
    result.row3.x = first.row3.x * other.row1.x + first.row3.y * other.row2.x + first.row3.z *
other.row3.x + first.row3.w * other.row4.x;
    result.row3.y = first.row3.x * other.row1.y + first.row3.y * other.row2.y + first.row3.z *
other.row3.y + first.row3.w * other.row4.y;
    result.row3.z = first.row3.x * other.row1.z + first.row3.y * other.row2.z + first.row3.z *
other.row3.z + first.row3.w * other.row4.z;
    result.row3.w = first.row3.x * other.row1.w + first.row3.y * other.row2.w + first.row3.z *
other.row3.w + first.row3.w * other.row4.w;

    //Row4
    result.row4.x = first.row4.x * other.row1.x + first.row4.y * other.row2.x + first.row4.z *
other.row3.x + first.row4.w * other.row4.x;
    result.row4.y = first.row4.x * other.row1.y + first.row4.y * other.row2.y + first.row4.z *
other.row3.y + first.row4.w * other.row4.y;
    result.row4.z = first.row4.x * other.row1.z + first.row4.y * other.row2.z + first.row4.z *
other.row3.z + first.row4.w * other.row4.z;
    result.row4.w = first.row4.x * other.row1.w + first.row4.y * other.row2.w + first.row4.z *
other.row3.w + first.row4.w * other.row4.w;

    return result;
}
```

Ooef ... eso es un montón de código que en realidad parece más aterrador de lo que realmente parece. Se puede hacer en un bucle for, pero (probablemente erróneamente) pensé que esto sería más claro para las personas que nunca han trabajado con matrices.

Mira el código y observa un patrón que se repite. Multiplique la columna por la fila, agréguela y continúe (esto es lo mismo para cualquier matriz de tamaño)

* Tenga en cuenta que la multiplicación con matrices no es como la multiplicación normal. AXB! = B x A *

Ahora que sabemos cómo proyectar y agregar esto a nuestra matriz de posición, nuestro código de la vida real probablemente se verá así:

```
glUseProgram(cubeProgram)
Matrix4x4<float> position;
position = Matrix4x4<float>::translate(position, 1,0,0);
position = Matrix4x4<float>::multiply(Matrix<float>::perspective<float>(50, 1 , 0.1f,
100000.0f), position);
```

```
glBindVertexArray(cubeVAO)
glUniformMatrix4fv(shaderRef, 1, GL_TRUE, &position.getRawData()[0]);
glEnableVertexAttribArray ( 0 );
glDrawArrays ( GL_TRIANGLES, 0,cubeVerticesSize)
```

Ahora nuestro bicho está aplastado y nuestro cubo parece bastante épico en la distancia. Si desea escalar su cubo, la fórmula es la siguiente:

```
 /*!
 *  Scales the matrix with given vector
 *  @param s The vector you wish to scale with
 */
static Matrix4x4<T> scale(const Matrix4x4<T>& mat, T x, T y, T z){
    Matrix4x4<T> tmp(mat);
    tmp.row1.x *= x;
    tmp.row2.y *= y;
    tmp.row3.z *= z;
    return tmp;
}
```

Solo necesitas ajustar las variables diagonales.

Para la rotación necesitas echar un vistazo más de cerca a los Cuaterniones.

Lea Matemáticas 3d en línea: https://riptutorial.com/es/opengl/topic/4063/matematicas-3d

Capítulo 8: Programa de introspección

Introducción

Se pueden obtener varias características de los objetos del programa a través de la API del programa.

Examples

Información del atributo vértice

La información sobre los atributos de vértice se puede recuperar con la función OGL glGetProgram y los parámetros `GL_ACTIVE_ATTRIBUTES` y `GL_ACTIVE_ATTRIBUTE_MAX_LENGTH` .

La ubicación de un atributo de sombreado activo puede determinarse por la función OGL glGetAttribLocation , por el índice del atributo.

```
GLuint shaderProg = ...;
std::map< std::string, GLint > attributeLocation;

GLint maxAttribLen, nAttribs;
glGetProgramiv( shaderProg, GL_ACTIVE_ATTRIBUTES, &nAttribs );
glGetProgramiv( shaderProg, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, &maxAttribLen
GLint written, size;
GLenum type;
std::vector< GLchar >attrName( maxAttribLen );
for( int attribInx = 0; attribInx < nAttribs; attribInx++ )
{
    glGetActiveAttrib( shaderProg, attribInx, maxAttribLen, &written, &size, &type,
&attrName[0] );
    attributeLocation[attrName] = glGetAttribLocation( shaderProg, attrName.data() );
}
```

Información uniforme

La información sobre los uniformes activos en un programa se puede recuperar con la función OGL glGetProgram y los parámetros `GL_ACTIVE_UNIFORMS` y `GL_ACTIVE_UNIFORM_MAX_LENGTH` .

La ubicación de una variable uniforme de sombreado activa puede determinarse por la función OGL glGetActiveUniform , por el índice del atributo.

```
GLuint shaderProg = ...;
std::map< std::string, GLint > unifomLocation;

GLint maxUniformLen, nUniforms;
glGetProgramiv( shaderProg, GL_ACTIVE_UNIFORMS, &nUniforms );
glGetProgramiv( shaderProg, GL_ACTIVE_UNIFORM_MAX_LENGTH, &maxUniformLen );

GLint written, size;
GLenum type;
```

```
std::vector< GLchar >uniformName( maxUniformLen );
for( int uniformInx = 0; uniformInx < nUniforms; uniformInx++ )
{
    glGetActiveUniform( shaderProg, uniformInx, maxUniformLen, &written, &size, &type,
&uniformName[0] );
    unifomLocation[uniformName] = glGetUniformLocation( shaderProg, uniformName.data() );
}
```

Lea Programa de introspección en línea: https://riptutorial.com/es/opengl/topic/10805/programa-de-introspeccion

Capítulo 9: Shader Loading y Compilación

Introducción

Estos ejemplos demuestran varias formas de cargar y compilar shaders. Todos los ejemplos ***deben incluir código de manejo de errores*** .

Observaciones

Los objetos de sombreado, tal como se crean a partir de `glCreateShader` , no hacen mucho. Contienen el código compilado para una sola etapa, pero ni siquiera tienen que contener el código compilado *completo* para esa etapa. En muchos sentidos, funcionan como archivos de objetos C y C ++.

Los objetos del programa contienen el programa final vinculado. Pero también tienen el estado para los valores uniformes del programa, así como una serie de otros datos estatales. Tienen API para realizar una introspección de los datos de la interfaz del sombreador (aunque solo se incluyó en GL 4.3). Los objetos de programa son los que definen el código de sombreado que se utiliza al renderizar.

Los objetos Shader, una vez utilizados para vincular un programa, ya no son necesarios a menos que tenga la intención de usarlos para vincular otros programas.

Examples

Cargar Shader separable en C ++

4.1

Este código carga, compila y vincula un solo archivo que crea un programa de sombreado separado para una sola etapa . Si hay errores, obtendrá el registro de información para esos errores.

El código utiliza alguna funcionalidad de C ++ 11 comúnmente disponible.

```
#include <string>
#include <fstream>

//In C++17, we could take a `std::filesystem::path` instead of a std::string
//for the filename.
GLuint CreateSeparateProgram(GLenum stage, const std::string &filename)
{
    std::ifstream input(filename.c_str(), std::ios::in | std::ios::binary | std::ios::ate);

    //Figure out how big the file is.
    auto fileSize = input.tellg();
    input.seekg(0, ios::beg);
```

```
    //Read the whole file.
    std::string fileData(fileSize);
    input.read(&fileData[0], fileSize);
    input.close();

    //Compile&link the file
    auto fileCstr = (const GLchar *)fileData.c_str();
    auto program = glCreateShaderProgramv(stage, 1, &fileCstr);

    //Check for errors
    GLint isLinked = 0;
    glGetProgramiv(program, GL_LINK_STATUS, &isLinked);
    if(isLinked == GL_FALSE)
    {
        //Note: maxLength includes the NUL terminator.
        GLint maxLength = 0;
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &maxLength);

        //C++11 does not permit you to overwrite the NUL terminator,
        //even if you are overwriting it with the NUL terminator.
        //C++17 does, so you could subtract 1 from the length and skip the `pop_back`.
        std::basic_string<GLchar> infoLog(maxLength);
        glGetProgramInfoLog(program, maxLength, &maxLength, &infoLog[0]);
        infoLog.pop_back();

        //The program is useless now. So delete it.
        glDeleteProgram(program);

        //Use the infoLog in whatever manner you deem best.

        //Exit with failure.
        return 0;
    }

    return program;
}
```

Compilación de objetos Shader individuales en C ++

El modelo de compilación GLSL tradicional implica compilar código para una etapa de sombreado en un objeto de sombreado, luego vincular múltiples objetos de sombreado (que cubren todas las etapas que desea usar) en un solo objeto de programa.

Desde 4.2, se pueden crear objetos de programa que tienen solo una etapa de sombreado. Este método vincula todas las etapas de sombreado en un solo programa.

Compilación de objetos Shader

```
#include <string>
#include <fstream>

//In C++17, we could take a `std::filesystem::path` instead of a std::string
//for the filename.
GLuint CreateShaderObject(GLenum stage, const std::string &filename)
{
```

```
    std::ifstream input(filename.c_str(), std::ios::in | std::ios::binary | std::ios::ate);

    //Figure out how big the file is.
    auto fileSize = input.tellg();
    input.seekg(0, ios::beg);

    //Read the whole file.
    std::string fileData(fileSize);
    input.read(&fileData[0], fileSize);
    input.close();

    //Create a shader name
    auto shader = glCreateShader(stage);

    //Send the shader source code to GL
    auto fileCstr = (const GLchar *)fileData.c_str();
    glShaderSource(shader, 1, &fileCstr, nullptr);

    //Compile the shader
    glCompileShader(shader);

    GLint isCompiled = 0;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &isCompiled);
    if(isCompiled == GL_FALSE)
    {
        GLint maxLength = 0;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &maxLength);

        //C++11 does not permit you to overwrite the NUL terminator,
        //even if you are overwriting it with the NUL terminator.
        //C++17 does, so you could subtract 1 from the length and skip the `pop_back`.
        std::basic_string<GLchar> infoLog(maxLength);
        glGetShaderInfoLog(shader, maxLength, &maxLength, &infoLog[0]);
        infoLog.pop_back();

        //We don't need the shader anymore.
        glDeleteShader(shader);

        //Use the infoLog as you see fit.

        //Exit with failure.
        return 0;
    }

    return shader;
}
```

Programa de enlace de objetos

```
#include <string>

GLuint LinkProgramObject(vector<GLuint> shaders)
{
    //Get a program object.
    auto program = glCreateProgram();

    //Attach our shaders to our program
    for(auto shader : shaders)
```

```
        glAttachShader(program, shader);

    //Link our program
    glLinkProgram(program);

    //Note the different functions here: glGetProgram* instead of glGetShader*.
    GLint isLinked = 0;
    glGetProgramiv(program, GL_LINK_STATUS, (int *)&isLinked);
    if(isLinked == GL_FALSE)
    {
        GLint maxLength = 0;
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &maxLength);

        //C++11 does not permit you to overwrite the NUL terminator,
        //even if you are overwriting it with the NUL terminator.
        //C++17 does, so you could subtract 1 from the length and skip the `pop_back`.
        std::basic_string<GLchar> infoLog(maxLength);
        glGetProgramInfoLog(program, maxLength, &maxLength, &infoLog[0]);
        infoLog.pop_back();

        //We don't need the program anymore.
        glDeleteProgram(program);

        //Use the infoLog as you see fit.

        //Exit with failure
        return 0;
    }

    //Always detach shaders after a successful link.
    for(auto shader : shaders)
        gldetachShader(program, shader);

    return program;
}
```

Lea Shader Loading y Compilación en línea: https://riptutorial.com/es/opengl/topic/8685/shader-loading-y-compilacion

Capítulo 10: Sombreadores

Sintaxis

- #version version_number // La versión GLSL que estamos usando
- void main () {/ * Code * /} // Función principal de Shader
- en nombre del tipo; // Especifica un parámetro de entrada - GLSL 1.30
- escriba el nombre; // Especifica un parámetro de salida - GLSL 1.30
- nombre de tipo inout; // Parámetro para entrada y salida - GLSL 1.30

Parámetros

Parámetro	Detalles
tipo	El tipo de parámetro, tiene que ser un tipo incorporado GLSL.

Observaciones

Para especificar qué versión de GLSL debe usarse para compilar un sombreado, use el **preprocesador de** versión, por ejemplo, `#version 330` . Cada versión de OpenGL es necesaria para admitir versiones específicas de GLSL . Si un preprocesador `#version` no está definido en la parte superior de un sombreado, se usa la versión predeterminada 1.10.

Examples

Shader para renderizar un rectángulo de color

Un programa de sombreado, en el sentido de **OpenGL** , contiene una serie de sombreadores diferentes. Cualquier programa de sombreado debe tener al menos un sombreado de **vértice** , que calcula la posición de los puntos en la pantalla, y un sombreador de **fragmentos** , que calcula el color de cada píxel. (En realidad, la historia es más larga y compleja, pero de todos modos ...)

Los siguientes sombreadores son para `#version 110` , pero deberían ilustrar algunos puntos:

Sombreador de vértices:

```
#version 110

// x and y coordinates of one of the corners
attribute vec2 input_Position;

// rgba colour of the corner. If all corners are blue,
// the rectangle is blue. If not, the colours are
// interpolated (combined) towards the center of the rectangle
attribute vec4 input_Colour;
```

```
// The vertex shader gets the colour, and passes it forward
// towards the fragment shader which is responsible with colours
// Must match corresponding declaration in the fragment shader.
varying vec4 Colour;

void main()
{
    // Set the final position of the corner
    gl_Position = vec4(input_Position, 0.0f, 1.0f);

    // Pass the colour to the fragment shader
    UV = input_UV;
}
```

Sombreador de fragmentos:

```
#version 110

// Must match declaration in the vertex shader.
varying vec4 Colour;

void main()
{
    // Set the fragment colour
    gl_FragColor = vec4(Colour);
}
```

Lea Sombreadores en línea: https://riptutorial.com/es/opengl/topic/5065/sombreadores

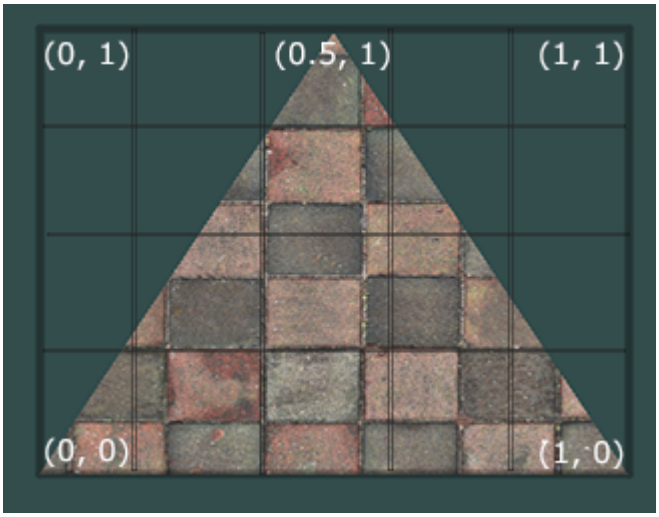
Capítulo 11: Texturizado

Examples

Fundamentos de la texturización.

Una textura es una forma de almacenamiento de datos que permite un acceso conveniente no solo a entradas de datos particulares, sino también a puntos de muestra que mezclan (interpolan) varias entradas juntas.

En OpenGL, las texturas se pueden usar para muchas cosas, pero lo más común es mapear una imagen a un polígono (por ejemplo, un triángulo). Para mapear la textura a un triángulo (u otro polígono) tenemos que decirle a cada vértice a qué parte de la textura corresponde. Asignamos una coordenada de textura a cada vértice de un polígono y luego se interpolará entre todos los fragmentos de ese polígono. Las coordenadas de la textura suelen oscilar entre 0 y 1 en los ejes x e y como se muestra en la siguiente imagen:



Las coordenadas de textura de este triángulo se verían así:

```
GLfloat texCoords[] = {
    0.0f, 0.0f,  // Lower-left corner
    1.0f, 0.0f,  // Lower-right corner
    0.5f, 1.0f   // Top-center corner
};
```

Coloque esas coordenadas en VBO (objeto de búfer de vértice) y cree un nuevo atributo para el sombreador. Ya debería tener al menos un atributo para las posiciones de vértice, así que cree otro para las coordenadas de textura.

Generando textura

Lo primero que se debe hacer es generar un objeto de textura al que se hará referencia mediante

una ID que se almacenará en una *textura* int sin signo:

```
GLuint texture;
glGenTextures(1, &texture);
```

Después de eso, debe estar vinculado para que todos los comandos de textura subsiguientes configuren esta textura:

```
glBindTexture(GL_TEXTURE_2D, texture);
```

Cargando imagen

Para cargar una imagen, puede crear su propio cargador de imágenes o puede usar una biblioteca de carga de imágenes como SOIL (Simple OpenGL Image Library) en c ++ o PNGDecoder de TWL en java.

Un ejemplo de carga de imagen con SOIL sería:

```
int width, height;
unsigned char* image = SOIL_load_image("image.png", &width, &height, 0, SOIL_LOAD_RGB);
```

Ahora puedes asignar esta imagen al objeto de textura:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
```

Después de eso debes desenlazar el objeto de textura:

```
glBindTexture(GL_TEXTURE_2D, 0);
```

Wrap parámetro para coordenadas de textura

Como se ve arriba, la esquina inferior izquierda de la textura tiene las coordenadas UV (st) (0, 0) y la esquina superior derecha de la textura tiene las coordenadas (1, 1), pero las coordenadas de textura de una malla pueden estar en cualquier rango. Para manejar esto, se debe definir cómo se ajustan las coordenadas de la textura a la textura.

El parámetro de ajuste para la coordenada de textura se puede configurar con glTextureParameter usando `GL_TEXTURE_WRAP_S` , `GL_TEXTURE_WRAP_T` y `GL_TEXTURE_WRAP_R` .

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

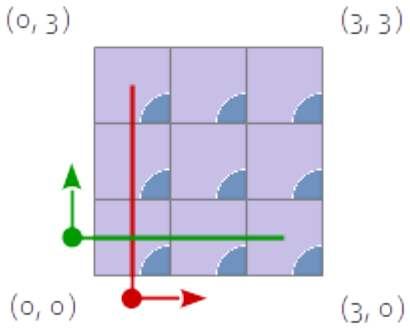
Los parámetros posibles son:

- `GL_CLAMP_TO_EDGE` hace que las coordenadas de la textura se `GL_CLAMP_TO_EDGE` al rango *[1 /*

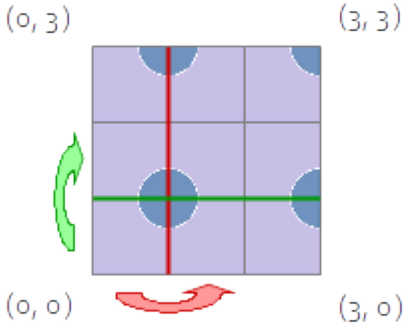
2N, 1 - 1 / 2N] , donde *N* es el tamaño de la textura en la dirección.

- `GL_CLAMP_TO_BORDER` hace lo mismo que `GL_CLAMP_TO_EDGE` , pero en los casos en que la sujeción, los datos de texel obtenidos se sustituyen con el color especificado por `GL_TEXTURE_BORDER_COLOR` .

- `GL_REPEAT` hace que se `GL_REPEAT` la parte entera de la coordenada de textura. La textura es de **baldosas** .



- `GL_MIRRORED_REPEAT` : Si la parte entera de la coordenada de textura es par, entonces se ignora. A diferencia de, si la parte entera de la coordenada de la textura es impar, entonces la coordenada de la textura se establece en *1 - frac (s)* . *fract (s)* es la parte fraccionaria de la coordenada de textura. Eso hace que la textura se **refleje** cada 2 veces.



- `GL_MIRROR_CLAMP_TO_EDGE` hace que la coordenada textue se repita como para `GL_MIRRORED_REPEAT` para una repetición de la textura, en cuyo punto la coordenada se fija como en `GL_CLAMP_TO_EDGE` .

Tenga en cuenta que el valor predeterminado para `GL_TEXTURE_WRAP_S` , `GL_TEXTURE_WRAP_T` y `GL_TEXTURE_WRAP_R` es `GL_REPEAT` .

Aplicando texturas

Lo último que hay que hacer es enlazar la textura antes del dibujo:

```
glBindTexture(GL_TEXTURE_2D, texture);
```

Textura y Framebuffer

Puede adjuntar una imagen en una textura a un framebuffer, para que pueda renderizar directamente a esa textura.

```
glGenFramebuffers (1, &framebuffer);
glBindFramebuffer (GL_FRAMEBUFFER, framebuffer);
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D,
                       texture,
                       0);
```

Nota: no puede leer y escribir desde la misma textura en la misma tarea de renderizado, ya que se trata de un comportamiento indefinido. Pero puede usar: `glTextureBarrier()` entre las llamadas de render para esto.

Leer datos de textura

Puede leer datos de textura con la función `glGetTexImage` :

```
char *outBuffer = malloc(buf_size);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);

glGetTexImage(GL_TEXTURE_2D,
              0,
              GL_RGBA,
              GL_UNSIGNED_BYTE,
              outBuffer);
```

Nota: el tipo y el formato de la textura son solo por ejemplo y pueden ser diferentes.

Usando PBOs

Si vincula un búfer a `GL_PIXEL_UNPACK_BUFFER` entonces el parámetro de `data` en `glTexImage2D` es un desplazamiento en ese búfer.

Esto significa que glTexImage2D no necesita esperar a que todos los datos se copien de la memoria de la aplicación antes de que pueda regresar, lo que reduce la sobrecarga en el subproceso principal.

```
glGenBuffers(1, &pbo);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
glBufferData(GL_PIXEL_UNPACK_BUFFER, width*height*3, NULL, GL_STREAM_DRAW);
void* mappedBuffer = glMapBuffer(GL_PIXEL_UNPACK_BUFFER, GL_WRITE_ONLY);

//write data into the mapped buffer, possibly in another thread.
int width, height;
unsigned char* image = SOIL_load_image("image.png", &width, &height, 0, SOIL_LOAD_RGB);
memcpy(mappedBuffer, image, width*height*3);
SOIL_free_image(image);

// after reading is complete back on the main thread
```

```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, 0);
```

Pero donde realmente brillan las PBO es cuando necesita leer el resultado de un render en la memoria de la aplicación. Para leer los datos de píxeles en un búfer, `GL_PIXEL_PACK_BUFFER` a `GL_PIXEL_PACK_BUFFER` entonces el parámetro de datos de `glGetTexImage` será un desplazamiento en ese búfer:

```
glGenBuffers(1, &pbo);
glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo);
glBufferData(GL_PIXEL_PACK_BUFFER, buf_size, NULL, GL_STREAM_COPY);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);

glGetTexImage(GL_TEXTURE_2D,
              0,
              GL_RGBA,
              GL_UNSIGNED_BYTE,
              null);
//ensure we don't try and read data before the transfer is complete
GLsync sync = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);

// then regularly check for completion
GLint result;
glGetSynciv(sync, GL_SYNC_STATUS, sizeof(result), NULL, &result);
if(result == GL_SIGNALED){
    glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo);
    void* mappedBuffer = glMapBuffer(GL_PIXEL_PACK_BUFFER, GL_READ_ONLY);

    //now mapped buffer contains the pixel data

    glUnmapBuffer(GL_PIXEL_PACK_BUFFER);

}
```

Usando texturas en sombreadores GLSL

El sombreador de vértices solo acepta las coordenadas de textura como un atributo de vértice y envía las coordenadas al sombreador de fragmentos. De forma predeterminada, también garantizará que el fragmento reciba la coordenada interpolada correctamente en función de su posición en un triángulo:

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoordIn;

out vec2 texCoordOut;

void main()
{
    gl_Position = vec4(position, 1.0f);
    texCoordOut = texCoordIn;
}
```

El sombreador de fragmentos acepta la variable de salida `texCoord` como una variable de entrada. Luego puede agregar una textura al fragmento sombreado declarando un `uniform sampler2D` . Para muestrear un fragmento de la textura usamos una `texture` función incorporada que tiene dos parámetros. Primero está la textura de la que queremos muestrear y la segunda es la coordenada de esta textura:

```
in vec2 texCoordOut;

out vec4 color;

uniform sampler2D image;

void main()
{
    color = texture(image, texCoordOut);
}
```

Tenga en cuenta que la `image` no es la identificación directa de la textura aquí. Es la identificación de la *unidad de textura* que se muestreará. A su vez, las texturas no están limitadas a los programas directamente; Ellos están obligados a unidades de textura. Esto se logra primero `glActiveTexture` unidad de textura con `glActiveTexture` , y luego llamando a `glBindTexture` afectará a esta unidad de textura en particular. Sin embargo, como la unidad de textura predeterminada es la unidad de textura `0` , los programas que usan una textura pueden simplificarse omitiendo esta llamada.

Lea Texturizado en línea: https://riptutorial.com/es/opengl/topic/3461/texturizado

Capítulo 12: Utilizando VAOs

Introducción

El objeto de matriz Vertex almacena cómo opengl debe interpretar un conjunto de VBO.

En esencia, le permitirá evitar llamar a glVertexAttribPointer cada vez que quiera renderizar una nueva malla.

Si no quiere lidiar con VAOs, simplemente puede crear uno y vincularlo durante la inicialización del programa y pretender que no existen.

Sintaxis

- void glEnableVertexAttribArray (GLuint attribIndex);
- void glDisableVertexAttribArray (GLuint attribIndex);
- void glVertexAttribPointer (GLuint attribIndex, tamaño de GLint, tipo GLenum, GLboolean normalizado, zancada GLsizei, const GLvoid * puntero);
- void glVertexAttribFormat (GLuint attribIndex, tamaño de GLint, tipo GLenum, GLboolean normalizado, relativo relativo GLuint);
- void glVertexAttribBinding (GLuint attribIndex, GLuint bindingIndex);
- void glBindVertexBuffer (GLuint bindingIndex, GLuint buffer, GLintptr offset, GLintptr stride);

Parámetros

parámetro	Detalles
attribIndex	la ubicación del atributo de vértice a la que la matriz de vértices alimentará datos
tamaño	El número de componentes que se extraerán del atributo.
tipo	El tipo C ++ de los datos de atributo en el búfer
normalizado	si se deben asignar tipos de enteros al rango de punto flotante [0, 1] (para sin signo) o [-1, 1] (para firmado)
puntero	el byte se desplaza en el búfer al primer byte de los datos del atributo (conversión a `void*` por razones heredadas)
compensar	el desplazamiento de bytes base desde el principio del búfer hasta donde comienzan los datos de la matriz

parámetro	Detalles
relativoOffset	el desplazamiento a un atributo particular, relativo al desplazamiento base para el búfer
paso	el número de bytes de los datos de un vértice al siguiente
buffer	el objeto de búfer donde se almacenan las matrices de vértices
bindingIndex	el índice al que se enlazará el objeto de búfer de origen

Observaciones

El formato de atributo separado VAO setup puede interoperar con `glVertexAttribPointer` (este último se define en términos del primero). Pero debes tener cuidado al hacerlo.

La versión de formato de atributo separada tiene equivalentes de acceso directo de estado (DSA) en 4.5. Estos tendrán los mismos parámetros, pero en lugar de usar el VAO vinculado, el VAO que se está modificando se pasa explícitamente. Cuando se utiliza DSA de index buffer para `glDrawElements` se puede configurar con `glVertexArrayElementBuffer(vao, ebo);`

Examples

Versión 3.0

Cada atributo está asociado con un recuento de componentes, tipo, normalizado, desplazamiento, zancada y VBO. El VBO no se pasa explícitamente como un parámetro, sino que es el búfer vinculado a GL_ARRAY_BUFFER en el momento de la llamada.

```
void prepareMeshForRender(Mesh mesh){
    glBindVertexArray(mesh.vao);
    glBindBuffer(GL_ARRAY_BUFFER, mesh.vbo);

    glVertexAttribPointer (posAttrLoc, 3, GL_FLOAT, false, sizeof(Vertex), mesh.vboOffset +
offsetof(Vertex, pos));//will associate mesh.vbo with the posAttrLoc
    glEnableVertexAttribArray(posAttrLoc);

    glVertexAttribPointer (normalAttrLoc, 3, GL_FLOAT, false, sizeof(Vertex), mesh.vboOffset +
offsetof(Vertex, normal));
    glEnableVertexAttribArray(normalAttrLoc);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mesh.ebo); //this binding is also saved.
    glBindVertexArray(0);
}

void drawMesh(Mesh[] meshes){
    foreach(mesh in meshes){
        glBindVertexArray(mesh.vao);
        glDrawElements(GL_TRIANGLES, mesh.vertexCount, GL_UNSIGNED_INT, mesh.indexOffset);
    }
}
```

Versión 4.3

4.3

OpenGL 4.3 (o ARB_separate_attrib_format) agrega una forma alternativa de especificar los datos de vértice, que crea una separación entre el formato de los datos enlazados para un atributo y el origen del objeto del búfer que proporciona los datos. Entonces, en lugar de tener un VAO por malla, puede tener un formato VAO por vértice.

Cada atributo está asociado con un formato de vértice y un punto de enlace. El formato de vértice consiste en el tipo, el recuento de componentes, si está normalizado y el desplazamiento relativo desde el inicio de los datos a ese vértice en particular. El punto de enlace especifica de qué búfer un atributo toma sus datos. Al separar los dos, puede vincular buffers sin volver a especificar ningún formato de vértice. También puede cambiar el búfer que proporciona datos a múltiples atributos con una sola llamada de enlace.

```
//accessible constant declarations
constexpr int vertexBindingPoint = 0;
constexpr int texBindingPoint = 1;// free to choose, must be less than the
GL_MAX_VERTEX_ATTRIB_BINDINGS limit

//during initialization
glBindVertexArray(vao);

glVertexAttribFormat(posAttrLoc, 3, GL_FLOAT, false, offsetof(Vertex, pos));
// set the details of a single attribute
glVertexAttribBinding(posAttrLoc, vertexBindingPoint);
// which buffer binding point it is attached to
glEnableVertexAttribArray(posAttrLoc);

glVertexAttribFormat(normalAttrLoc, 3, GL_FLOAT, false, offsetof(Vertex, normal));
glVertexAttribBinding(normalAttrLoc, vertexBindingPoint);
glEnableVertexAttribArray(normalAttrLoc);

glVertexAttribFormat(texAttrLoc, 2, GL_FLOAT, false, offsetof(Texture, tex));
glVertexAttribBinding(texAttrLoc, texBindingPoint);
glEnableVertexAttribArray(texAttrLoc);
```

Luego, durante el sorteo, mantendrá el vao enlazado y solo cambiará los enlaces del búfer.

```
void drawMesh(Mesh[] mesh){
    glBindVertexArray(vao);

    foreach(mesh in meshes){
        glBindVertexBuffer(vertexBindingPoint, mesh.vbo, mesh.vboOffset, sizeof(Vertex));
        glBindVertexBuffer(texBindingPoint, mesh.texVbo, mesh.texVboOffset, sizeof(Texture));
        // bind the buffers to the binding point

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mesh.ebo);

        glDrawElements(GL_TRIANGLES, mesh.vertexCount, GL_UNSIGNED_INT, mesh.indexOffset);
        //draw
    }
}
```

Lea Utilizando VAOs en línea: https://riptutorial.com/es/opengl/topic/9237/utilizando-vaos

Capítulo 13: Vista y proyección OGL

Introducción

Sobre matriz de modelo, matriz de vista, proyección ortográfica y perspectiva.

Examples

Implementar una cámara en OGL 4.0 GLSL 400.

Si queremos ver una escena como si la hubiéramos fotografiado con una cámara, primero debemos definir algunas cosas:

- La posición desde la cual se ve la escena, la posición del ojo `pos` .
- El punto que miramos en la escena (`target`). También es común definir la dirección en la que miramos. Técnicamente necesitamos una *línea de visión* . Una recta en el espacio se define matemáticamente por 2 puntos o por un punto y un vector. La primera parte de la definición es la posición del ojo y la segunda es el `target` o el vector de línea de visión `los` .
- La dirección hacia `up` hacia `up` .
- El campo de visión ' `fov_y` . Esto significa el ángulo entre las dos líneas rectas, comenzando en la posición del ojo y terminando en el punto más a la izquierda y el más a la derecha, que se puede ver simultáneamente.
- El gran tamaño y la relación de aspecto de la ventana gráfica a la que proyectamos nuestra imagen `vp` .
- El *plano* `near` y el *plano* `far` . El *plano cercano* es la distancia desde la posición del ojo al plano desde donde los objetos se hacen visibles para nosotros. El *plano lejano* es la distancia desde la posición del ojo al plano en el que los objetos de la escena son visibles para nosotros. Una explicación de lo *que* se necesita el *plano cercano* y el *plano lejano* seguirá más adelante.

Una definición de estos datos en C ++ y en Python puede tener este aspecto:

C ++

```
using TVec3 = std::array<float,3>;
struct Camera
{
    TVec3 pos    {0.0, -8.0, 0.0};
    TVec3 target {0.0, 0.0, 0.0};
    TVec3 up     {0.0, 0.0, 1.0};
    float fov_y  {90.0};
    TSize vp     {800, 600};
    float near   {0.5};
    float far    {100.0};
};
```

Pitón

```
class Camera:
    def __init__(self):
        self.pos    = (0, -8, 0)
        self.target = (0, 0, 0)
        self.up     = (0, 0, 1)
        self.fov_y  = 90
        self.vp     = (800, 600)
        self.near   = 0.5
        self.far    = 100.0
```

Para tomar en cuenta toda esta información al dibujar una escena, generalmente se usan una matriz de proyección y una matriz de vista. Para organizar las partes individuales de una escena en la escena, se utilizan matrices modelo. Sin embargo, estos se mencionan aquí solo por estar completos y no se tratarán aquí.

- Matriz de proyección: la matriz de proyección describe el mapeo desde puntos 3D en el mundo tal como se ven desde una cámara estenopeica, hasta puntos 2D de la ventana gráfica.
- Ver matriz: la matriz de vista define la posición del *ojo* y la dirección de visualización en la escena.
- Matriz modelo: la matriz modelo define la ubicación y el tamaño relativo de un objeto en la escena.

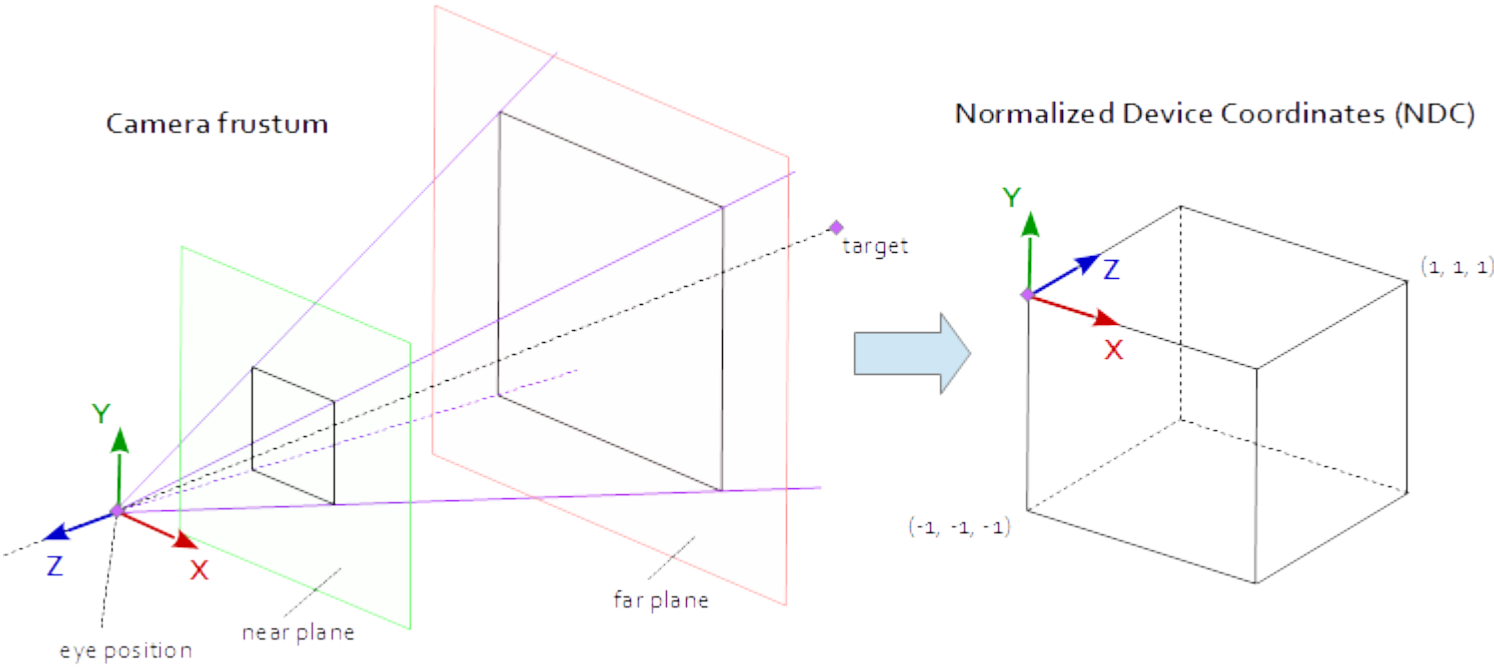
Después de que hayamos llenado las estructuras de datos anteriores con los datos correspondientes, tenemos que traducirlos a las matrices apropiadas. En el modo de compatibilidad OGL, esto se puede hacer con las funciones `gluLookAt` y `gluPerspective` que establecen los uniformes `gl_ModelViewMatrix` , `gl_NormalMatrix` , y `gl_ModelViewProjectionMatrix` . En OGL 3.1 y GLSL #version 150 se eliminaron los uniformes incorporados, ya que toda la matriz de matriz de función fija quedó obsoleta. Si queremos usar el sombreador de alto nivel OGL con GLSL versión 330 o incluso más, tenemos que definir y establecer los uniformes de la matriz (aparte del uso de la palabra clave de `compatibility` GLSL).

Configurar la perspectiva - Matriz de proyección.

Un punto en la ventana gráfica está visible cuando se encuentra en el AABB nativo (cuadro delimitador alineado con el eje) definido por los puntos `(-1.0, -1.0, -1.0)` y `(1.0, 1.0, 1.0)` . Esto se llama las Coordenadas del dispositivo normalizado (NDC). Un punto con las coordenadas `(-1.0, -1.0, z)` se pintará en la esquina inferior izquierda de la ventana gráfica y un punto con las coordenadas `(1.0, 1.0, z)` se pintará en la esquina superior derecha de la ventana gráfica. La coordenada Z se asigna desde el intervalo (-1.0, 1.0) al intervalo (0.0, 1.0) y se escribe en el Z-buffer.

Todo lo que podemos ver de la escena es dentro de una pirámide de 4 lados. La parte superior de la pirámide es la *posición del ojo* . Los 4 lados de la pirámide están definidos por el campo de vista (`fov_y`) y la relación de aspecto (`vp[0]/vp[1]`). La matriz de proyección debe asignar los puntos desde el interior de la pirámide al NDC definido por los puntos `(-1.0, -1.0, -1.0)` y `(1.0, 1.0, 1.0)` . En este punto, nuestra pirámide es infinita, no tiene un final en profundidad y no

podemos mapear un espacio infinito a uno finito. Para esto, ahora necesitamos el *plano cercano* y el *plano lejano* , ya que transforman la pirámide en un tronco cortando la parte superior y limitando la pirámide en la profundidad. El plano cercano y el plano lejano deben elegirse de tal manera que incluyan todo lo que debería ser visible desde la escena.



El mapeo desde los puntos dentro de un frustum hasta el NDC es matemática pura y generalmente se puede resolver. El desarrollo de las fórmulas a menudo se discutió y se publicó repetidamente en toda la web. Ya que no puede insertar una fórmula LaTeX en una documentación de desbordamiento de pila, se prescinde aquí y solo se agrega el código fuente de C ++ y Python completado. Tenga en cuenta que las coordenadas de los ojos se definen en el sistema de coordenadas de la mano derecha, pero NDC utiliza el sistema de coordenadas de la mano izquierda. La matriz de proyección se calcula a partir de, el *campo de visión* `fov_y` , la *relación de aspecto* `vp[0]/vp[1]` , el *plano* `near` y el *plano* `far` .

C ++

```
using TVec4  = std::array< float, 4 >;
using TMat44 = std::array< TVec4, 4 >;
TMat44 Camera::Perspective( void )
{
    float fn  = far + near;
    float f_n = far - near;
    float r   = (float)vp[0] / vp[1];
    float t   = 1.0f / tan( ToRad( fov_y ) / 2.0f );
    return TMat44{
        TVec4{ t / r, 0.0f,  0.0f,                    0.0f },
        TVec4{ 0.0f,  t,     0.0f,                    0.0f },
        TVec4{ 0.0f,  0.0f, -fn / f_n,               -1.0  },
        TVec4{ 0.0f,  0.0f, -2.0f * far * near / f_n,  0.0f } };
}
```

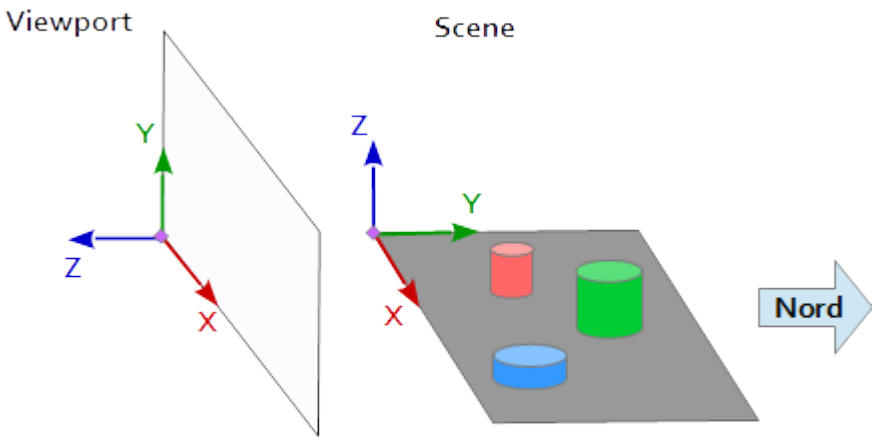
Pitón

```
def Perspective(self):
    fn, = self.far + self.near
    f_n = self.far - self.near
    r = self.vp[0] / self.vp[1]
    t = 1 / math.tan( math.radians( self.fov_y ) / 2 )
    return numpy.matrix( [
        [ t/r, 0,  0,                                0 ],
        [ 0,   t,  0,                                0 ],
        [ 0,   0, -fn/f_n,                           -1 ],
        [ 0,   0, -2 * self.far * self.near / f_n,   0 ] ] )
```

Configurar la mirada a la escena - Ver matriz

En el sistema de coordenadas de la ventana gráfica, el eje Y apunta hacia arriba `(0, 1, 0)` y el eje X apunta hacia la derecha `(1, 0, 0)` . Esto da como resultado un eje Z que apunta hacia afuera de la ventana gráfica (`(0, 0, -1) = cross( X-axis, Y-axis )`).

En la escena, el eje X apunta hacia el este, el eje Y hacia el norte y el eje Z hacia la parte superior.



El eje X de la ventana gráfica `(1, 0, 0)` coincide con el eje Y de la escena `(1, 0, 0)` , el eje Y de la ventana gráfica `(0, 1, 0 )` coincide con el eje Z de la escena `(0, 0, 1)` y el eje Z de la ventana gráfica `(0, 0, 1 )` coinciden con el eje Y negado de la escena `(0, -1, 0)` .

Por lo tanto, cada punto y cada vector del sistema de referencia de la escena deben convertirse primero en coordenadas de la ventana gráfica. Esto se puede hacer mediante algunas operaciones de intercambio e inversión en los vectores escalares.

```
 x  y  z
--------
 1  0  0  | x' =  x
 0  0  1  | y' =  z
 0 -1  0  | z' = -y
```

Para configurar una matriz de vista, la posición `pos` , el objetivo `target` y el vector `up` deben asignarse al sistema de coordenadas de la ventana gráfica, como se describe anteriormente. Esto

da los 2 puntos `p` y `t` y el vector `u` , como en el siguiente fragmento de código. El eje Z de la matriz de vista es la línea de visión inversa, que se calcula mediante `p - t` . El eje Y es el vector ascendente `u` . El eje X se calcula por el producto cruzado del eje Y y el eje Z. Para la ortonormalización de la matriz de vista, el producto cruzado se usa por segunda vez, para calcular el eje Y desde el eje Z y el eje X (Por supuesto, la ortogonalización Gram-Schmidt funcionaría igual de bien). Al final, los 3 ejes deben estar normalizada y la *posición* del *ojo* `pos` tiene que ser establecido como º origen de la matriz de vista.

El siguiente código define una matriz que encapsula exactamente los pasos necesarios para calcular una mirada a la escena:

1. Convertir las coordenadas del modelo en coordenadas de la ventana gráfica.
2. Gire en la dirección de la dirección de la vista.
3. Movimiento a la posición del ojo.

C ++

```
template< typename T_VEC >
TVec3 Cross( T_VEC a, T_VEC b )
{
    return { a[1] * b[2] - a[2] * b[1], a[2] * b[0] - a[0] * b[2], a[0] * b[1] - a[1] * b[0]
};
}

template< typename T_A, typename T_B >
float Dot( T_A a, T_B b )
{
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}

template< typename T_VEC >
void Normalize( T_VEC & v )
{
    float len = sqrt( v[0] * v[0] + v[1] * v[1] + v[2] * v[2] ); v[0] /= len; v[1] /= len;
v[2] /= len;
}

TMat44 Camera::LookAt( void )
{
    TVec3 mz = { pos[0] - target[0], pos[1] - target[1], pos[2] - target[2] };
    Normalize( mz );
    TVec3 my = { up[0], up[1], up[2] };
    TVec3 mx = Cross( my, mz );
    Normalize( mx );
    my = Cross( mz, mx );

    TMat44 v{
        TVec4{ mx[0], my[0], mz[0], 0.0f },
        TVec4{ mx[1], my[1], mz[1], 0.0f },
        TVec4{ mx[2], my[2], mz[2], 0.0f },
        TVec4{ Dot(mx, pos), Dot(my, pos), Dot(TVec3{-mz[0], -mz[1], -mz[2]}, pos), 1.0f }
    };

    return v;
}
```

pitón

```
def LookAt(self):
    mz = Normalize( (self.pos[0]-self.target[0], self.pos[1]-self.target[1], self.pos[2]-
self.target[2]) ) # inverse line of sight
    mx = Normalize( Cross( self.up, mz ) )
    my = Normalize( Cross( mz, mx ) )
    tx = Dot( mx, self.pos )
    ty = Dot( my, self.pos )
    tz = Dot( (-mz[0], -mz[1], -mz[2]), self.pos )
    return = numpy.matrix( [
        [mx[0], my[0], mz[0], 0],
        [mx[1], my[1], mz[1], 0],
        [mx[2], my[2], mz[2], 0],
        [tx, ty, tz, 1] ] )
```

Las matrices finalmente se escriben en uniformes y se usan en el sombreado de vértices para transformar las posiciones del modelo.

Sombreador de vértices

En el sombreado de vértices, se realiza una transformación después de la otra.

1. La matriz del modelo lleva el objeto (malla) a su lugar en la escena. (Esto solo se enumera por estar completo y no se ha documentado aquí ya que no tiene nada que ver con la vista de la escena)
2. La matriz de vista define la dirección desde la que se ve la escena. La transformación con la matriz de vista rota los objetos de la escena para que se vean desde la dirección de vista deseada con referencia al sistema de coordenadas de la ventana gráfica.
3. La matriz de proyección transforma los objetos de una vista paralela en una vista en perspectiva.

```
#version 400

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inCol;

out vec3 vertCol;

uniform mat4 u_projectionMat44;
uniform mat4 u_viewMat44;
uniform mat4 u_modelMat44;

void main()
{
    vertCol       = inCol;
    vec4 modelPos = u_modelMat44 * vec4( inPos, 1.0 );
    vec4 viewPos  = u_viewMat44 * modelPos;
    gl_Position   = u_projectionMat44 * viewPos;
}
```

Sombreador de fragmentos

El sombreador de fragmentos se enumera aquí solo por estar completo. El trabajo fue hecho

antes.

```
#version 400

in vec3 vertCol;

out vec4 fragColor;

void main()
{
    fragColor = vec4( vertCol, 1.0 );
}
```

Una vez que se compilan y gustan los sombreadores, las matrices pueden unirse a las variables uniformes.

C ++

```
int shaderProg = ;
Camera camera;

// ...

int prjMatLocation  = glGetUniformLocation( shaderProg, "u_projectionMat44" );
int viewMatLocation = glGetUniformLocation( shaderProg, "u_viewMat44" );
glUniformMatrix4fv( prjMatLocation,  1, GL_FALSE, camera.Perspective().data()->data() );
glUniformMatrix4fv( viewMatLocation, 1, GL_FALSE, camera.LookAt().data()->data() );
```

Pitón

```
shaderProg =
camera = Camera()

# ...

prjMatLocation  = glGetUniformLocation( shaderProg, b"u_projectionMat44" )
viewMatLocation = glGetUniformLocation( shaderProg, b"u_viewMat44" )
glUniformMatrix4fv( prjMatLocation,  1, GL_FALSE, camera.Perspective() )
glUniformMatrix4fv( viewMatLocation, 1, GL_FALSE, camera.LookAt() )
```

Además, he agregado el volcado de código completo de un ejemplo de Python (agregar el ejemplo de C ++ desafortunadamente superaría el límite de 30000 caracteres). En este ejemplo, la cámara se mueve elípticamente alrededor de un tetraedro en un punto focal de la elipse. La dirección de visualización siempre está dirigida al tetraeder.

Pitón

Para ejecutar el script de Python se debe instalar NumPy .

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import numpy
from time import time
```

```
import math
import sys

def Cross( a, b ): return ( a[1] * b[2] - a[2] * b[1], a[2] * b[0] - a[0] * b[2], a[0] * b[1]
- a[1] * b[0], 0.0 )
def Dot( a, b ): return a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
def Normalize( v ):
    len = math.sqrt( v[0] * v[0] + v[1] * v[1] + v[2] * v[2] )
    return (v[0] / len, v[1] / len, v[2] / len)

class Camera:
    def __init__(self):
        self.pos    = (0, -8, 0)
        self.target = (0, 0, 0)
        self.up     = (0, 0, 1)
        self.fov_y  = 90
        self.vp     = (800, 600)
        self.near   = 0.5
        self.far    = 100.0
    def Perspective(self):
        fn, f_n = self.far + self.near, self.far - self.near
        r, t = self.vp[0] / self.vp[1], 1 / math.tan( math.radians( self.fov_y ) / 2 )
        return numpy.matrix( [ [t/r,0,0,0], [0,t,0,0], [0,0,-fn/f_n,-1], [0,0,-
2*self.far*self.near/f_n,0] ] )
    def LookAt(self):
         mz = Normalize( (self.pos[0]-self.target[0], self.pos[1]-self.target[1], self.pos[2]-
self.target[2]) ) # inverse line of sight
    mx = Normalize( Cross( self.up, mz ) )
    my = Normalize( Cross( mz, mx ) )
    tx = Dot( mx, self.pos )
    ty = Dot( my, self.pos )
    tz = Dot( (-mz[0], -mz[1], -mz[2]), self.pos )
    return = numpy.matrix( [ [mx[0], my[0], mz[0], 0], [mx[1], my[1], mz[1], 0], [mx[2],
my[2], mz[2], 0], [tx, ty, tz, 1] ] )

# shader program object
class ShaderProgram:
    def __init__( self, shaderList, uniformNames ):
        shaderObjs = []
        for sh_info in shaderList: shaderObjs.append( self.CompileShader(sh_info[0],
sh_info[1] ) )
        self.LinkProgram( shaderObjs )
        self.__unifomLocation = {}
        for name in uniformNames:
            self.__unifomLocation[name] = glGetUniformLocation( self.__prog, name )
            print( "uniform %-30s at loaction %d" % (name, self.__unifomLocation[name]) )
    def Use(self):
        glUseProgram( self.__prog )
    def SetUniformMat44( self, name, mat ):
        glUniformMatrix4fv( self.__unifomLocation[name], 1, GL_FALSE, mat )
    # read shader program and compile shader
    def CompileShader(self, sourceFileName, shaderStage):
        with open( sourceFileName, 'r' ) as sourceFile:
            sourceCode = sourceFile.read()
        nameMap = { GL_VERTEX_SHADER: 'vertex', GL_FRAGMENT_SHADER: 'fragment' }
        print( '\n%s shader code:' % nameMap.get( shaderStage, '' ) )
        print( sourceCode )
        shaderObj = glCreateShader( shaderStage )
        glShaderSource( shaderObj, sourceCode )
        glCompileShader( shaderObj )
        result = glGetShaderiv( shaderObj, GL_COMPILE_STATUS )
```

```
        if not (result):
            print( glGetShaderInfoLog( shaderObj ) )
            sys.exit()
        return shaderObj
    # linke shader objects to shader program
    def LinkProgram(self, shaderObjs):
        self.__prog = glCreateProgram()
        for shObj in shaderObjs: glAttachShader( self.__prog, shObj )
        glLinkProgram( self.__prog )
        result = glGetProgramiv( self.__prog, GL_LINK_STATUS )
        if not ( result ):
            print( 'link error:' )
            print( glGetProgramInfoLog( self.__prog ) )
            sys.exit()

# vertex array object
class VAObject:
    def __init__( self, dataArrays, tetIndices ):
        self.__obj = glGenVertexArrays( 1 )
        self.__noOfIndices = len( tetIndices )
        self.__indexArr = numpy.array( tetIndices, dtype='uint' )
        noOfBuffers = len( dataArrays )
        buffers = glGenBuffers( noOfBuffers )
        glBindVertexArray( self.__obj )
        for i_buffer in range( 0, noOfBuffers ):
            vertexSize, dataArr = dataArrays[i_buffer]
            glBindBuffer( GL_ARRAY_BUFFER, buffers[i_buffer] )
            glBufferData( GL_ARRAY_BUFFER, numpy.array( dataArr, dtype='float32' ),
GL_STATIC_DRAW )
            glEnableVertexAttribArray( i_buffer )
            glVertexAttribPointer( i_buffer, vertexSize, GL_FLOAT, GL_FALSE, 0, None )
    def Draw(self):
        glBindVertexArray( self.__obj )
        glDrawElements( GL_TRIANGLES, self.__noOfIndices, GL_UNSIGNED_INT, self.__indexArr )

# glut window
class Window:
    def __init__( self, cx, cy ):
        self.__vpsize = ( cx, cy )
        glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH )
        glutInitWindowPosition( 0, 0 )
        glutInitWindowSize( self.__vpsize[0], self.__vpsize[1] )
        self.__id = glutCreateWindow( b'OGL window' )
        glutDisplayFunc( self.OnDraw )
        glutIdleFunc( self.OnDraw )
    def Run( self ):
        self.__startTime = time()
        glutMainLoop()

    # draw event
    def OnDraw(self):
        self.__vpsize = ( glutGet( GLUT_WINDOW_WIDTH ), glutGet( GLUT_WINDOW_HEIGHT ) )
        currentTime = time()
        # set up camera
        camera = Camera()
        camera.vp = self.__vpsize
        camera.pos = self.EllipticalPosition( 7, 4, self.CalcAng( currentTime, 10 ) )

        # set up attributes and shader program
        glEnable( GL_DEPTH_TEST )
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
```

```
        prog.Use()
        prog.SetUniformMat44( b"u_projectionMat44", camera.Perspective()  )
        prog.SetUniformMat44( b"u_viewMat44", camera.LookAt() )

        # draw object
        modelMat = numpy.matrix(numpy.identity(4), copy=False, dtype='float32')
        prog.SetUniformMat44( b"u_modelMat44", modelMat )
        tetVAO.Draw()

        glutSwapBuffers()

    def Fract( self, val ): return val - math.trunc(val)
    def CalcAng( self, currentTime, intervall ): return self.Fract( (currentTime -
self.__startTime) / intervall ) * 2.0 *     math.pi
    def CalcMove( self, currentTime, intervall, range ):
        pos = self.Fract( (currentTime - self.__startTime) / intervall ) * 2.0
        pos = pos if pos < 1.0 else (2.0-pos)
        return range[0] + (range[1] - range[0]) * pos
    def EllipticalPosition( self, a, b, angRag ):
        a_b = a * a - b * b
        ea = 0 if (a_b <= 0) else math.sqrt( a_b )
        eb = 0 if (a_b >= 0) else math.sqrt( -a_b )
        return ( a * math.sin( angRag ) - ea, b * math.cos( angRag ) - eb, 0 )

# initialize glut
glutInit()

# create window
wnd = Window( 800, 600 )

# define tetrahedron vertex array opject
sin120 = 0.8660254
tetVAO = VAObject(
    [ (3, [ 0.0, 0.0, 1.0, 0.0, -sin120, -0.5, sin120 * sin120, 0.5 * sin120, -0.5, -sin120 *
sin120, 0.5 * sin120, -0.5 ]),
      (3, [ 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, ])
    ],
    [ 0, 1, 2, 0, 2, 3, 0, 3, 1, 1, 3, 2 ]
)

# load, compile and link shader
prog = ShaderProgram(
    [ ('python/ogl4camera/camera.vert', GL_VERTEX_SHADER),
      ('python/ogl4camera/camera.frag', GL_FRAGMENT_SHADER)
    ],
    [b"u_projectionMat44", b"u_viewMat44", b"u_modelMat44"] )

# start main loop
wnd.Run()
```

Lea Vista y proyección OGL en línea: https://riptutorial.com/es/opengl/topic/10680/vista-y-proyeccion-ogl

Creditos

S. No	Capítulos	Contributors
1	Empezando con Opengl	ammar26, Ashwin Gupta, Bogdan0804, CaseyB, Community, datenwolf, Franko L. Tokalić, genpfault, L-X, Naman Dixit, Nicol Bolas, Nox, Valvy
2	Creación de contexto de OpenGL.	Dynamitos
3	Encapsular objetos OpenGL con C ++ RAII	0x499602D2, Nicol Bolas
4	Framebuffers	MarGenDo, Rabbid76
5	Iluminación básica	Ashwin Gupta, Elouarn Laine, Rabbid76
6	Instancia	MarGenDo, Nicol Bolas
7	Matemáticas 3d	Ashwin Gupta, Franko L. Tokalić, Valvy, Wyck
8	Programa de introspección	Nicol Bolas, Rabbid76
9	Shader Loading y Compilación	Nicol Bolas, Rabbid76
10	Sombreadores	FedeWar, genpfault, George, MarGenDo, nica.dan.cs, Nicol Bolas
11	Texturizado	Bartek Banachewicz, genpfault, George, MarGenDo, Nicol Bolas, Rabbid76, ratchet freak
12	Utilizando VAOs	Nicol Bolas, ratchet freak
13	Vista y proyección OGL	Matej Kormuth, Rabbid76, SurvivalMachine