eBook Gratuit

APPRENEZ opengl

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#opengl

Table des matières

À propos 1

Chapitre 1: Démarrer avec opengl 2

Remarques 2

Versions 2

Examples 3

Obtenir OpenGL 3

Linux 3

Microsoft Windows 3

Configuration manuelle OpenGL sous Windows 4

Composants Windows pour OpenGL 4

WGL 4

Interface de périphérique graphique (GDI) 4

Configuration de base 5

Créer une fenêtre 5

Format de pixel 6

Contexte de rendu 7

Obtenir les fonctions OpenGL 7

Meilleure configuration 9

Profils OpenGL 9

Extensions OpenGL 9

Format de pixel avancé et création de contexte 10

Créer OpenGL 4.1 avec C ++ et Cocoa 15

Création de contexte OpenGL Cross Platform (en utilisant SDL2) 23

Installez Modern OpenGL 4.1 sur macOS (Xcode, GLFW et GLEW) 24

Créer un contexte Opengl avec Java et LWJGL 3.0 41

Chapitre 2: Création de contexte OpenGL. 43

Examples 43

Créer une fenêtre de base 43

Ajout d'indices à la fenêtre 43

Chapitre 3: Éclairage de base 45

Examples ... 45
Modèle d'éclairage Phong ... 45
Comment ça marche ... 46
Chapitre 4: Encapsulation d'objets OpenGL avec C ++ RAII ... 52
Introduction ... 52
Remarques ... 52
Examples ... 52
En C ++ 98/03 ... 52
En C ++ 11 et versions ultérieures ... 53
Chapitre 5: Framebuffers ... 57
Examples ... 57
Bases du framebuffer ... 57
Limites ... 58
Utiliser le framebuffer ... 58
Chapitre 6: L'instanciation ... 61
Introduction ... 61
Examples ... 61
Instillation par tableaux d'attributs de vertex ... 61
Code de tableau instancié ... 61
Chapitre 7: Math 3d ... 64
Examples ... 64
Introduction aux matrices ... 64
Chapitre 8: Programme Introspection ... 70
Introduction ... 70
Examples ... 70
Informations d'attribut de sommet ... 70
Informations uniformes ... 70
Chapitre 9: Shader Chargement et Compilation ... 72
Introduction ... 72
Remarques ... 72
Examples ... 72

Charger le shader séparable en C ++ 72
Compilation d'objet Shader individuel en C ++ 73
Compilation d'objet Shader 73
Liaison d'objets de programme 74
Chapitre 10: Shaders 76
Syntaxe 76
Paramètres 76
Remarques 76
Examples 76
Shader pour le rendu d'un rectangle coloré 76
Chapitre 11: Texturation 78
Examples 78
Bases de texturation 78
Générer de la texture 78
Chargement de l'image 79
Paramètre Wrap pour les coordonnées de texture 79
Appliquer des textures 80
Texture et tampon 81
Lire les données de texture 81
Utiliser les PBO 81
Utiliser des textures dans les shaders GLSL 82
Chapitre 12: Utiliser des VAO 84
Introduction 84
Syntaxe 84
Paramètres 84
Remarques 85
Examples 85
Version 3.0 85
Version 4.3 86
Chapitre 13: Vue et projection OGL 88
Introduction 88

Examples 88
Implémenter une caméra dans OGL 4.0 GLSL 400 88
Mettre en place la perspective - matrice de projection 89
Mettre en place le regard sur la scène - Afficher la matrice 91
Crédits 98

À propos

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

Chapitre 1: Démarrer avec opengl

Remarques

OpenGL est un standard ouvert pour le rendu des graphiques 2D et 3D en utilisant le matériel graphique. OpenGL a été mis en œuvre sur un nombre impressionnant de plates-formes permettant aux applications ciblant OpenGL d'être extrêmement flexibles.

Versions

Version	Date de sortie
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

Obtenir OpenGL

L'une des idées fausses les plus courantes à propos d'OpenGL est qu'il s'agissait d'une bibliothèque pouvant être installée à partir de sources tierces. Cette idée fausse mène à de nombreuses questions sous la forme "comment installer OpenGL" ou "où télécharger le SDK OpenGL".

Ce n'est pas ainsi que OpenGL trouve le chemin dans le système informatique. OpenGL en soi n'est qu'un ensemble de spécifications sur les commandes qu'une implémentation doit suivre. Donc, c'est l'implémentation qui compte. Et pour le moment, les implémentations OpenGL font partie des pilotes GPU. Cela *pourrait* changer à l'avenir, lorsque la nouvelle interface de programmation GPU permettra de véritablement implémenter OpenGL en tant que bibliothèque, mais pour l'instant, il s'agit d'une API de programmation destinée aux pilotes graphiques.

Lorsque OpenGL a été publié pour la première fois, l'API s'est retrouvée dans le contrat ABI (Application Binary Interface) de Windows, Solaris et Linux (LSB-4 Desktop) en plus de son origine Sun Irix. Apple a suivi et intégré OpenGL si profondément dans MacOS X, que la version OpenGL disponible est étroitement liée à la version de MacOS X installée. Cela a pour effet notable que les environnements de programmation système pour ces systèmes d'exploitation (à savoir la chaîne d'outils du compilateur et de l'éditeur de liens qui cible en natif ces systèmes) **doivent** également fournir des définitions d'API OpenGL. Il n'est pas nécessaire d'installer un SDK pour OpenGL. Il est techniquement possible de programmer OpenGL sur ces systèmes d’exploitation sans qu’il soit nécessaire d’installer un SDK dédié, en supposant qu’un environnement de construction suivant l’ABI ciblée soit installé.

Un effet secondaire de ces règles ABI strictes est que la version OpenGL exposée via l’interface de liaison est le plus petit dénominateur commun que les programmes exécutés sur la plate-forme cible peuvent espérer être disponible. Par conséquent, les fonctionnalités OpenGL modernes sont accessibles via le mécanisme d'extension, décrit en détail séparément.

Linux

Sous Linux, il est assez courant de compartimenter les packages de développement pour différents aspects du système, afin de pouvoir les mettre à jour individuellement. Dans la plupart des distributions Linux, les fichiers de développement pour OpenGL sont contenus dans un package dédié, qui est généralement une dépendance pour un méta-package de développement d'applications de bureau. L'installation des fichiers de développement OpenGL pour Linux est généralement prise en charge lors de l'installation du / des package (s) de développement de méta.

Microsoft Windows

La bibliothèque de liaisons API `opengl32.dll` (nommée ainsi pour les versions 32 bits et 64 bits de Windows) est livrée par défaut avec chaque version de Windows depuis Windows NT-4 et

Windows 95B (toutes deux vers 1997). Cependant, cette DLL ne fournit pas d'implémentation OpenGL réelle (hormis un repli de logiciel dont le seul but est de servir de filet de sécurité pour les programmes si aucune autre implémentation OpenGL n'est installée). Cette DLL appartient à Windows et **ne doit pas** être modifiée ou déplacée! Les versions OpenGL modernes sont livrées dans le cadre du *pilote* ICD (*Installable Client Driver*) et accessibles via le `opengl32.dll` par défaut, préinstallé avec toutes les versions de Windows. Il a été décidé en interne par Microsoft, cependant, que les pilotes graphiques installés via *Windows Update* n'installeraient / ne mettraient pas à jour un ICD OpenGL. En tant que telles, de nouvelles installations de Windows avec des pilotes installés automatiquement ne prennent pas en charge les fonctionnalités OpenGL modernes. Pour obtenir un ICD OpenGL doté de fonctionnalités modernes, les pilotes graphiques doivent être téléchargés directement à partir du site Web du fournisseur du GPU et installés manuellement.

En ce qui concerne le développement, aucune mesure supplémentaire ne doit être prise en soi. Tous les compilateurs C / C ++ qui suivent les spécifications Windows ABI sont livrés avec des en-têtes et le stub de l'éditeur de liens (opengl32.lib) requis pour créer et lier des exécutables utilisant OpenGL.

Configuration manuelle OpenGL sous Windows

Exemple de code complet inclus à la fin

Composants Windows pour OpenGL

WGL

WGL (peut être prononcé *wiggle*) signifie "Windows-GL", comme dans "une interface entre Windows et OpenGL" - un ensemble de fonctions de l'API Windows pour communiquer avec OpenGL. Les fonctions WGL ont un préfixe *wgl* et ses jetons ont un préfixe *WGL_* .

La version OpenGL par défaut prise en charge sur les systèmes Microsoft est la version 1.1. C'est une très vieille version (la plus récente est la 4.5). La manière d'obtenir les versions les plus récentes est de mettre à jour vos pilotes graphiques, mais votre carte graphique doit prendre en charge ces nouvelles versions.

La liste complète des fonctions WGL peut être trouvée ici .

Interface de périphérique graphique (GDI)

GDI (aujourd'hui mis à jour vers GDI +) est une interface de dessin 2D qui vous permet de dessiner dans une fenêtre de Windows. Vous avez besoin de GDI pour initialiser OpenGL et lui permettre d'interagir avec lui (mais n'utilisera pas réellement GDI lui-même).

Dans GDI, chaque fenêtre a un *contexte de périphérique (DC)* utilisé pour identifier la cible de dessin lors de l'appel de fonctions (vous le transmettez en tant que paramètre). Cependant,

OpenGL utilise son propre *contexte de rendu (RC)* . Donc, DC sera utilisé pour créer RC.

Configuration de base

Créer une fenêtre

Donc, pour faire des choses avec OpenGL, nous avons besoin de RC, et pour obtenir RC, nous avons besoin de DC, et pour obtenir DC, nous avons besoin d'une fenêtre. La création d'une fenêtre à l'aide de l'API Windows nécessite plusieurs étapes. *Ceci est une routine de base, donc pour une explication plus détaillée, vous devriez consulter d'autres documentations, car il ne s'agit pas d'utiliser l'API Windows.*

Ceci est une installation de Windows, donc `Windows.h` doit être inclus et le point d'entrée du programme doit être la procédure `WinMain` avec ses paramètres. Le programme doit également être lié à `opengl32.dll` et à `gdi32.dll` (que vous soyez sur un système 64 ou 32 bits).

Nous devons d'abord décrire notre fenêtre en utilisant la structure `WNDCLASS` . Il contient des informations sur la fenêtre que nous voulons créer:

```
/* 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);
/* *************** */
```

Pour une explication précise de la signification de chaque champ (et pour une liste complète des champs), consultez la documentation MSDN.

Ensuite, nous pouvons créer une fenêtre à l'aide de `CreateWindowEx` . Une fois la fenêtre créée, nous pouvons acquérir son 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);
/* ************* */
```

Enfin, nous devons créer une boucle de message qui reçoit les événements de fenêtre du système d'exploitation:

```
/* 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
}
/* ********** */
```

Format de pixel

OpenGL a besoin de connaître certaines informations sur notre fenêtre, telles que le bitness de la couleur, la méthode de mise en mémoire tampon, etc. Pour cela, nous utilisons un *format de pixel* . Cependant, nous ne pouvons que suggérer au système d'exploitation le type de format de pixel dont nous avons besoin, et le système d'exploitation fournira *le* format *supporté le plus similaire* , nous n'avons pas de contrôle direct sur celui-ci. C'est pourquoi on l'appelle uniquement un *descripteur* .

```
/* 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);
/* ********************** */
```

Nous avons activé la double mise en mémoire tampon dans le champ `dwFlags` , nous devons donc appeler `SwapBuffers` pour voir les choses après le dessin.

Contexte de rendu

Après cela, nous pouvons simplement créer notre contexte de rendu:

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

Notez qu'un seul thread peut utiliser le RC à la fois. Si vous souhaitez l'utiliser à partir d'un autre thread plus tard, vous devez appeler `wglMakeCurrent` pour l'activer à nouveau (cela le désactivera sur le thread actuellement actif, etc.).

Obtenir les fonctions OpenGL

Les fonctions OpenGL sont obtenues en utilisant des pointeurs de fonctions. La procédure générale est la suivante:

1. D'une manière ou d'une autre, obtenir des types de pointeurs de fonctions (essentiellement les prototypes de fonctions)
2. Déclarez chaque fonction que nous aimerions utiliser (avec son type de pointeur de fonction)
3. Obtenir la fonction réelle

Par exemple, considérons 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" signifie "pointeur vers la fonction", suit le nom d'une fonction OpenGL et "PROC" à la fin - c'est le nom habituel du type de pointeur de la fonction OpenGL.)

Voici comment cela se passe sous Windows. Comme mentionné précédemment, Microsoft ne fournit que OpenGL 1.1. Tout d'abord, les types de pointeurs de fonction pour cette version

peuvent être trouvés en incluant `GL/gl.h` Après cela, nous déclarons toutes les fonctions que nous avons l'intention d'utiliser comme indiqué ci-dessus (le faire dans un fichier d'en-tête et les déclarer "extern" nous permettrait de les utiliser toutes après les avoir chargées une seule fois). Enfin, le chargement des fonctions OpenGL 1.1 se fait en ouvrant la DLL:

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

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

FreeLibrary(gl_module);
```

Cependant, nous voulons probablement un peu plus que OpenGL 1.1. Mais Windows ne nous donne pas les prototypes de fonction ou les fonctions exportées pour tout ce qui précède. Les prototypes doivent être acquis à partir du registre OpenGL . Trois fichiers nous intéressent: `GL/glext.h` , `GL/glcorearb.h` et `GL/wglext.h` .

Pour compléter `GL/gl.h` fourni par Windows, nous avons besoin de `GL/glext.h` . Il contient (comme décrit par la base de registre) "Interface de profil et d'extension de compatibilité OpenGL 1.2 et supérieur" (plus de détails sur les profils et les extensions plus tard, où nous verrons que ce n'est **pas une bonne idée d'utiliser ces deux fichiers**).

Les fonctions actuelles doivent être obtenues par `wglGetProcAddress` (pas besoin d'ouvrir la DLL pour ce type, elles ne sont pas là, utilisez simplement la fonction). Avec cela, nous pouvons récupérer toutes les fonctions d'OpenGL 1.2 et supérieur (mais pas 1.1). Notez que, pour fonctionner correctement, **le contexte de rendu OpenGL doit être créé et mis à jour** . Ainsi, par exemple, `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");
```

Nous pouvons en fait créer une procédure `get_proc` qui utilise à la fois `wglGetProcAddress` et `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;
}
```

Donc, pour conclure, nous créerions un fichier d'en-tête rempli de déclarations de pointeur de fonction comme ceci:

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

Nous pouvons alors créer une procédure comme `load_gl_functions` que nous appelons une seule fois et fonctionne comme ceci:

```
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");
```

Et vous êtes tous ensemble! Il suffit d'inclure l'en-tête avec les pointeurs de la fonction et GL loin.

Meilleure configuration

Profils OpenGL

OpenGL est en développement depuis plus de 20 ans et les développeurs ont toujours été stricts quant à la *compatibilité en amont (BC)* . L'ajout d'une nouvelle fonctionnalité est très difficile à cause de cela. Ainsi, en 2008, il a été séparé en deux "profils". *Core* et *compatibilité* . Le profil de base brise le BC en faveur des améliorations de performances et de certaines des nouvelles fonctionnalités. Il supprime même complètement certaines fonctionnalités héritées. Le profil de compatibilité conserve BC avec toutes les versions jusqu'à 1.0, et certaines nouvelles fonctionnalités ne sont pas disponibles. Il ne doit être utilisé que pour les anciens systèmes hérités, toutes les nouvelles applications doivent utiliser le profil de base.

À cause de cela, il y a un problème avec notre configuration de base - il ne fournit que le contexte rétrocompatible avec OpenGL 1.0. Le format de pixel est également limité. Il y a une meilleure approche, en utilisant des extensions.

Extensions OpenGL

Tout ajout à la fonctionnalité d'origine d'OpenGL s'appelle des extensions. En règle générale, ils peuvent soit rendre certaines choses légales qui ne l'étaient pas auparavant, étendre la plage de valeurs des paramètres, étendre GLSL, et même ajouter de nouvelles fonctionnalités.

Il existe trois groupes principaux d’extensions: fournisseur, EXT et ARB. Les extensions de fournisseur proviennent d'un fournisseur spécifique et elles ont une marque spécifique à un

fournisseur, comme AMD ou NV. Les extensions EXT sont réalisées par plusieurs fournisseurs travaillant ensemble. Après un certain temps, ils peuvent devenir des extensions ARB, qui sont toutes les extensions officiellement supportées et approuvées par ARB.

Pour acquérir des types de pointeurs de fonctions et des prototypes de fonctions de toutes les extensions *et comme mentionné précédemment, tous les types de pointeurs de fonctions d'OpenGL 1.2 et plus* , il faut télécharger les fichiers d'en-tête à partir du registre OpenGL . Comme nous l'avons vu, pour les nouvelles applications, il est préférable d'utiliser le profil de base, il serait donc préférable d'inclure `GL/glcorearb.h` au lieu de `GL/gl.h` et `GL/glext.h` (si vous utilisez `GL/glcorearb.h` alors don ne comprend pas `GL/gl.h`).

Il existe également des extensions pour le WGL, dans `GL/wglext.h` . Par exemple, la fonction permettant d'obtenir la liste de toutes les extensions prises en charge est en réalité une extension elle-même, `wglGetExtensionsStringARB` (elle retourne une grande chaîne avec une liste de toutes les extensions prises en charge séparées par des espaces).

Obtenir des extensions est également géré via `wglGetProcAddress` , nous pouvons donc utiliser notre wrapper comme avant.

Format de pixel avancé et création de contexte

L'extension `WGL_ARB_pixel_format` nous permet de créer un format de pixel avancé. Contrairement à avant, nous n'utilisons pas de structure. Au lieu de cela, nous passons la liste des attributs recherchés.

```
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 même, l'extension `WGL_ARB_create_context` nous permet de créer un contexte avancé:

```
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);
```

Pour une explication précise des paramètres et des fonctions, consultez la spécification OpenGL.

Pourquoi n'avons-nous pas simplement commencé avec eux? Eh bien, c'est parce que les *extensions* nous permettent de le faire, et pour obtenir des extensions, nous avons besoin de `wglGetProcAddress` , mais cela ne fonctionne qu'avec un contexte valide actif. Par conséquent, avant de pouvoir créer le contexte que nous souhaitons, nous devons déjà avoir un contexte actif, généralement appelé *contexte factice* .

Cependant, Windows ne permet pas de définir le format de pixel d'une fenêtre plus d'une fois. Pour cette raison, la fenêtre doit être détruite et recréée pour appliquer de nouvelles choses:

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

// Recreate the window...
```

Exemple de code complet:

```
/* 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
```

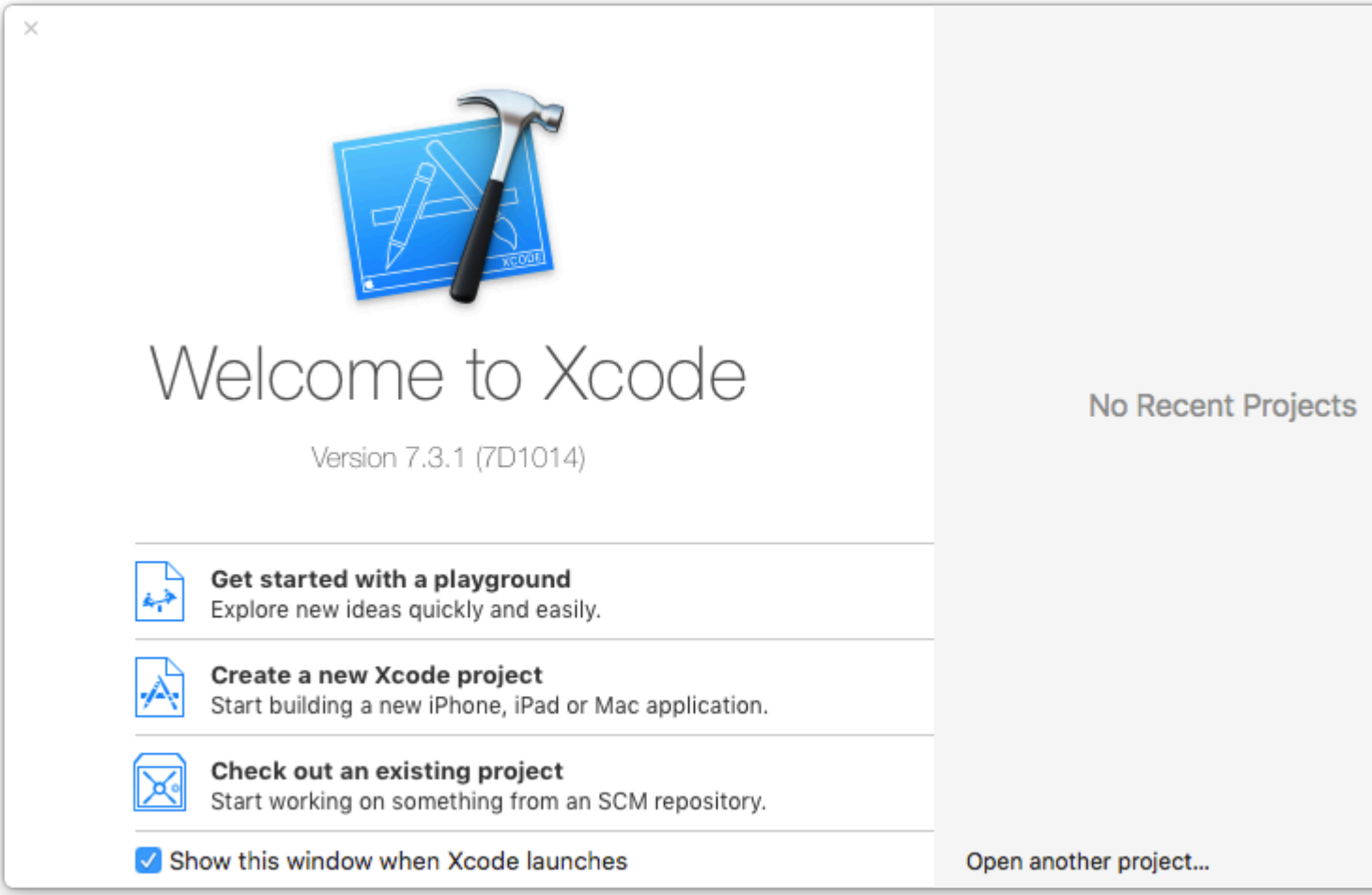
```
       the quit flag. */

    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;
}
```

Compilé avec `g++ GLExample.cpp -lopengl32 -lgdi32` avec MinGW / Cygwin ou `cl GLExample.cpp opengl32.lib gdi32.lib user32.lib` avec le compilateur `cl GLExample.cpp opengl32.lib gdi32.lib user32.lib` . Assurez-vous toutefois que les en-têtes du registre OpenGL sont dans le chemin d'inclusion. Sinon, utilisez l' `-I` pour `g++` ou `/I` pour `cl` afin d'indiquer au compilateur où elles se trouvent.

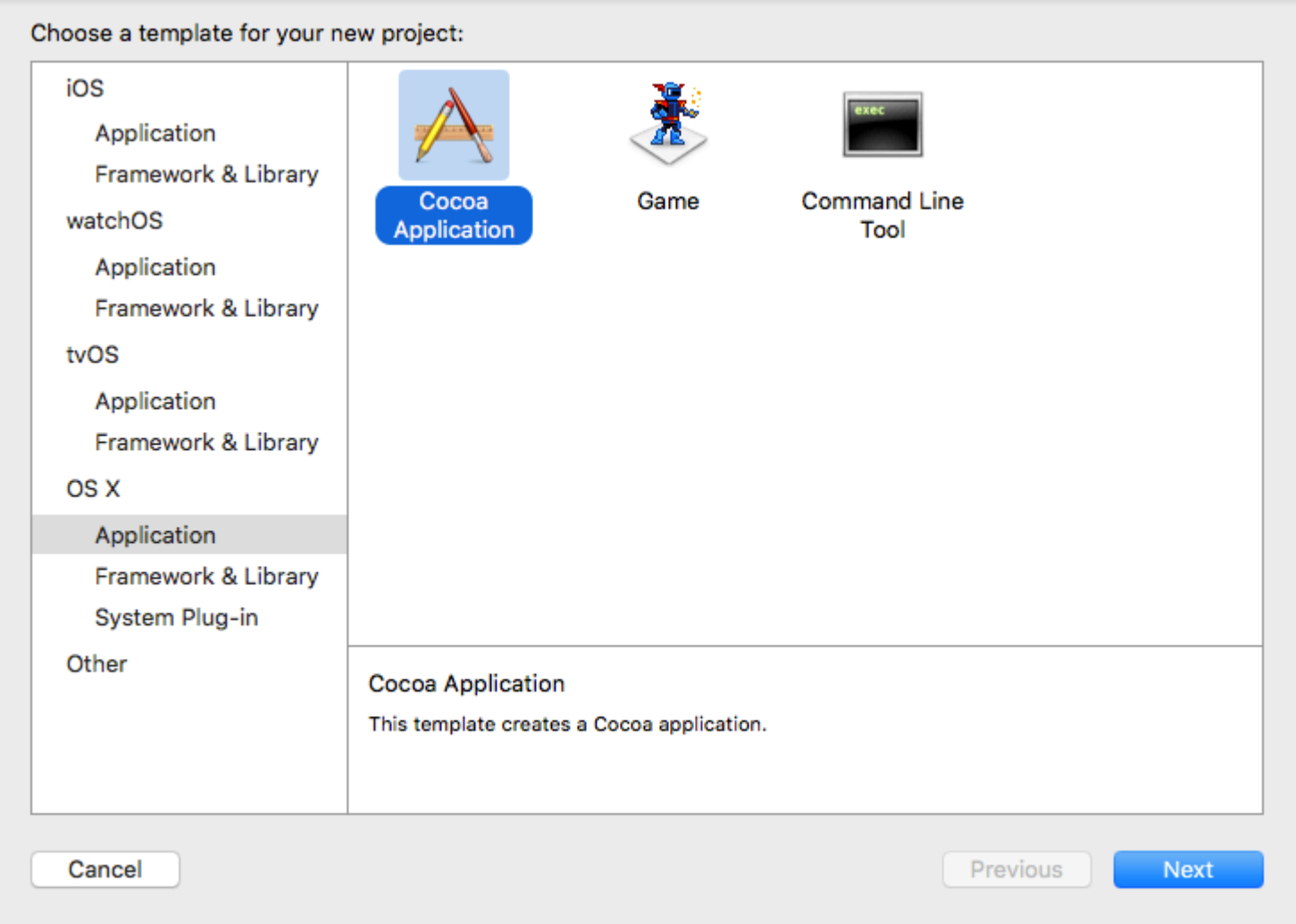
Créer OpenGL 4.1 avec C ++ et Cocoa

Note: Il y aura des Objective-c dans cet exemple. Dans cet exemple, nous allons créer un wrapper en C ++. Donc, ne vous en faites pas trop.

Commencez par démarrer Xcode et créez un projet.



Et sélectionnez une application Cocoa

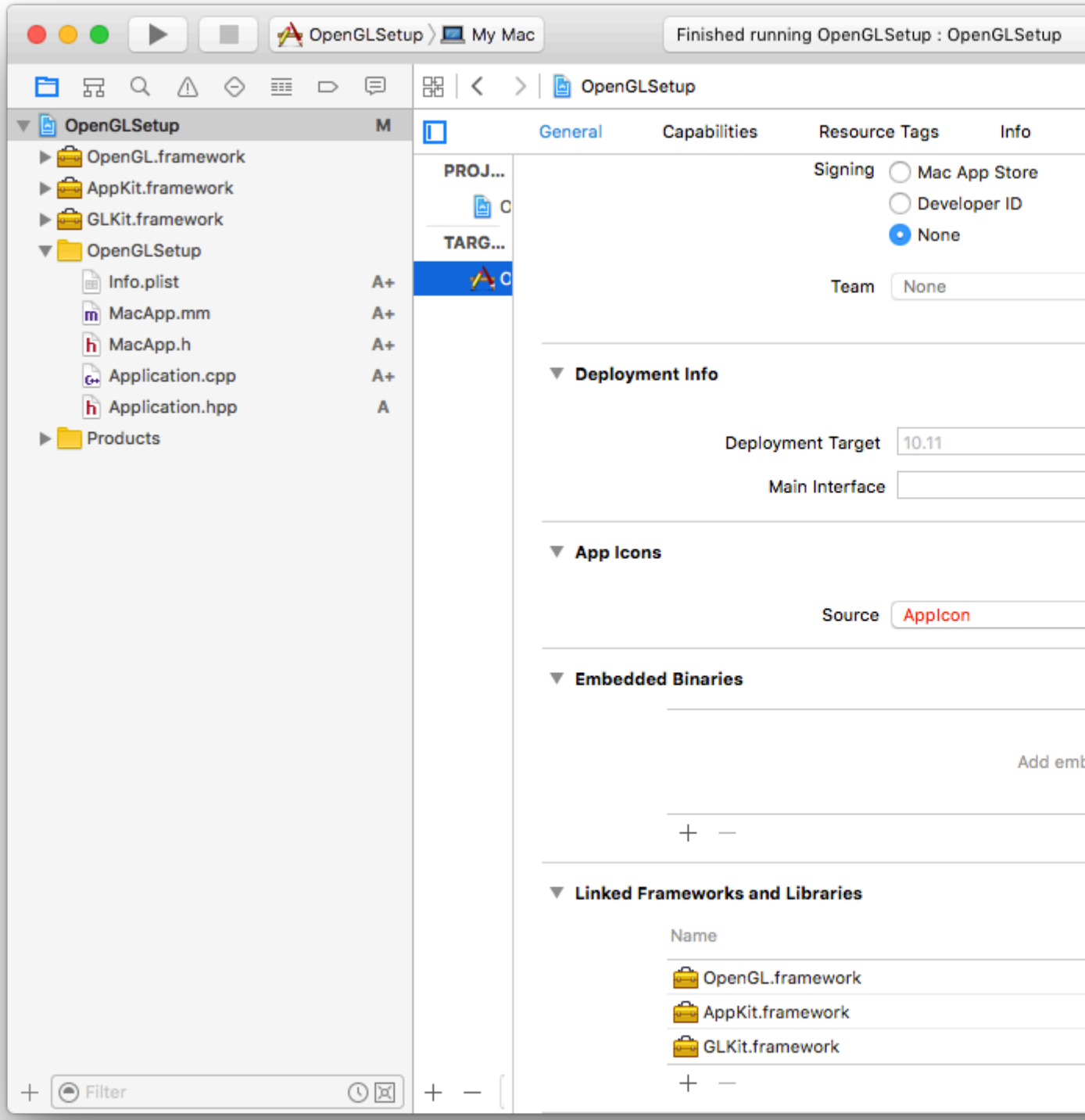


Supprimer toutes les sources sauf le fichier Info.plist (votre application ne fonctionnera pas sans elle)

Créez 4 nouveaux fichiers sources: Un fichier Objective-c ++ et un en-tête (j'ai appelé le mien MacApp) Une classe C ++ (j'ai appelé le mien (Application))

En haut à gauche (avec le nom du projet), cliquez dessus et ajoutez les frameworks et bibliothèques liés. Ajouter: OpenGL.Framework AppKit.Framework GLKit.Framework

Votre projet ressemblera probablement à ceci:



NSApplication est la classe principale que vous utilisez lors de la création d'une application MacOS. Il vous permet d'enregistrer des fenêtres et de capturer des événements.

Nous voulons enregistrer (notre propre) fenêtre sur la NSApplication. Commencez par créer dans votre en-tête objective-c ++ une classe objective-c qui hérite de NSWindow et implémente NSApplicationDelegate.

```
//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
```

Nous appelons cela du principal avec

```
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];
}
```

L'implémentation de notre fenêtre est en fait assez facile. Nous déclarons d'abord avec synthétiser notre vue et ajoutons un booléen global objectif-c lorsque la fenêtre devrait se fermer.

```
#import "MacApp.h"

@implementation MacApp

@synthesize glView;

BOOL shouldStop = NO;
```

Maintenant pour le constructeur. Ma préférence est d'utiliser 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;
}
```

Bon ... maintenant nous avons en fait une application exécutable .. Vous pourriez voir un écran noir ou scintiller.

Commençons à dessiner un triangle génial. (En c ++)

Mon en-tête d'application

```
#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 mise en oeuvre:

```
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);
}
```

Maintenant, il suffit d'appeler la mise à jour encore et encore (si vous voulez que quelque chose bouge) Implémentez dans votre classe objective-c

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

if(shouldStop){
    [self close];
```

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

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

}
```

Et ajoutez la méthode this dans l'implémentation de votre classe objective-c:

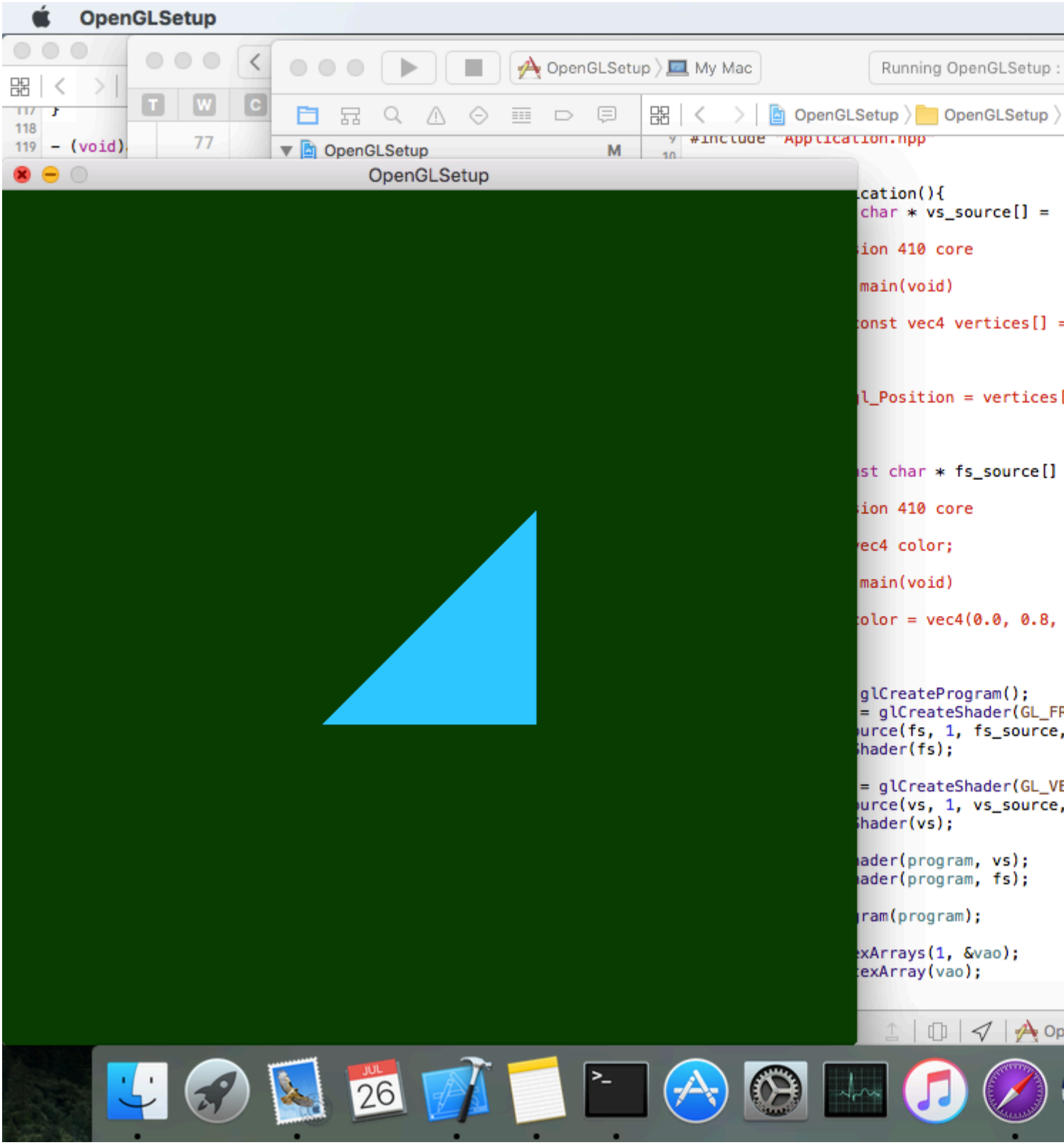
```
- (void)applicationDidFinishLaunching:(NSNotification *)notification {
    [NSTimer scheduledTimerWithTimeInterval:0.000001 target:self selector:@selector(drawLoop:)
userInfo:nil repeats:YES];
}
```

cela appellera la fonction de mise à jour de votre classe c ++ encore et encore (chaque 0,000001 seconde pour être précis)

Pour terminer, nous fermons la fenêtre lorsque le bouton Fermer est enfoncé:

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

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

Félicitations, vous avez maintenant une fenêtre impressionnante avec un triangle OpenGL sans framework tiers.



Création de contexte OpenGL Cross Platform (en utilisant SDL2)

Créer une fenêtre avec un contexte OpenGL (extension par GLEW):

```
#define GLEW_STATIC

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

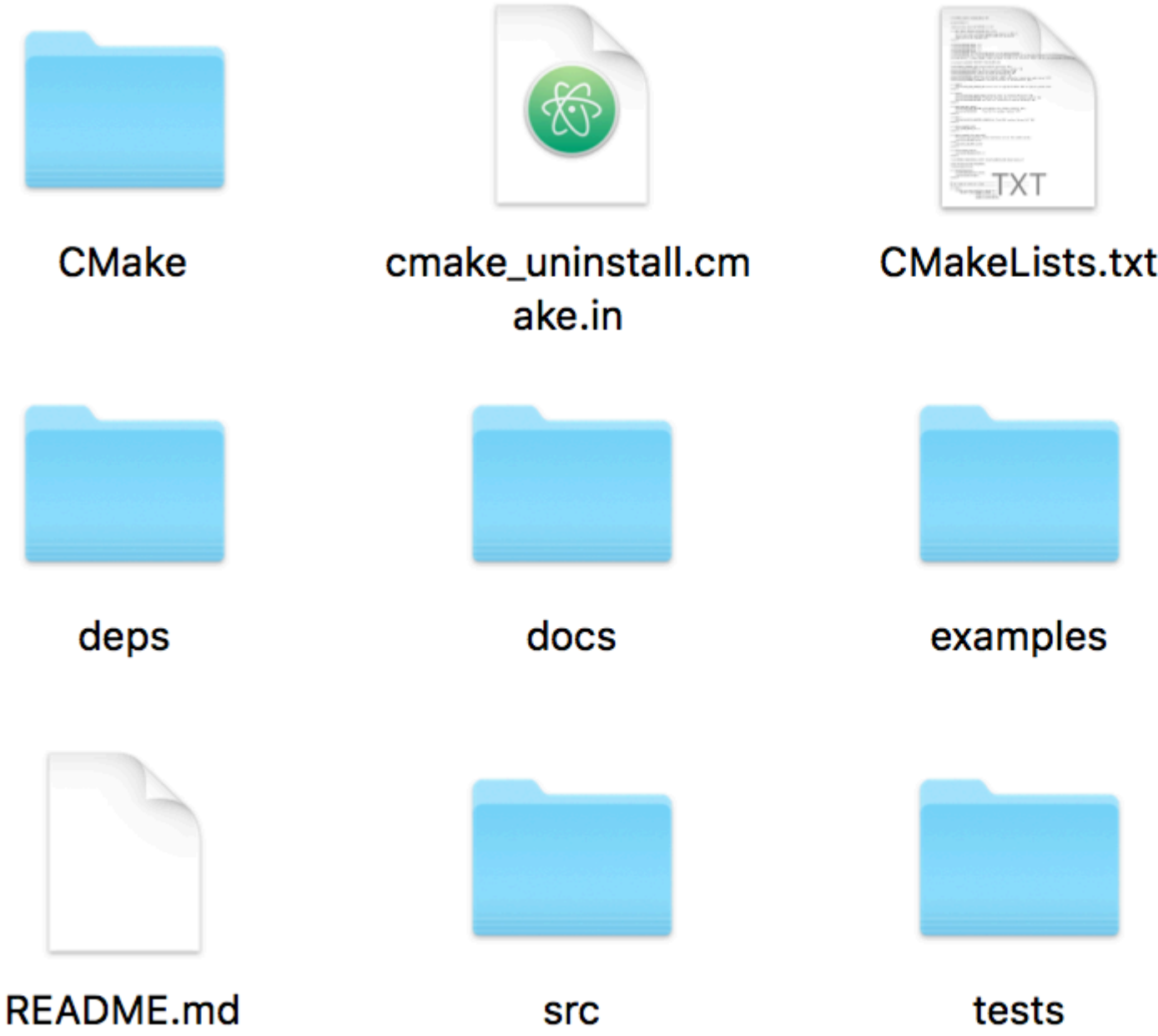
```
int main(int argc, char* argv[])
{
    SDL_Init(SDL_INIT_VIDEO); /* Initialises Video Subsystem in SDL */

    /* 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;
}
```

Installez Modern OpenGL 4.1 sur macOS (Xcode, GLFW et GLEW)

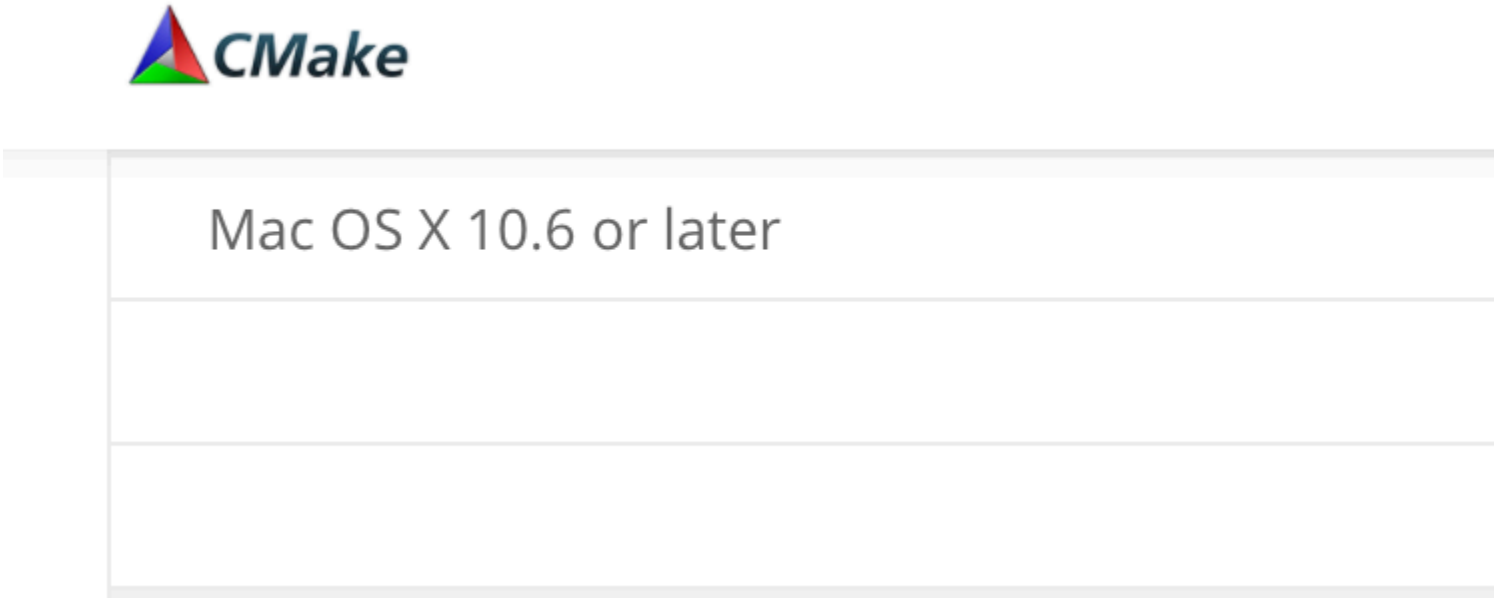
1. Installez GLFW

La première étape consiste à créer une fenêtre OpenGL. GLFW est une bibliothèque multi-plateforme Open Source pour créer des fenêtres avec OpenGL, pour installer GLFW en premier téléchargeant ses fichiers depuis www.glfw.org

Extraire le dossier GLFW et son contenu ressemblera à ceci

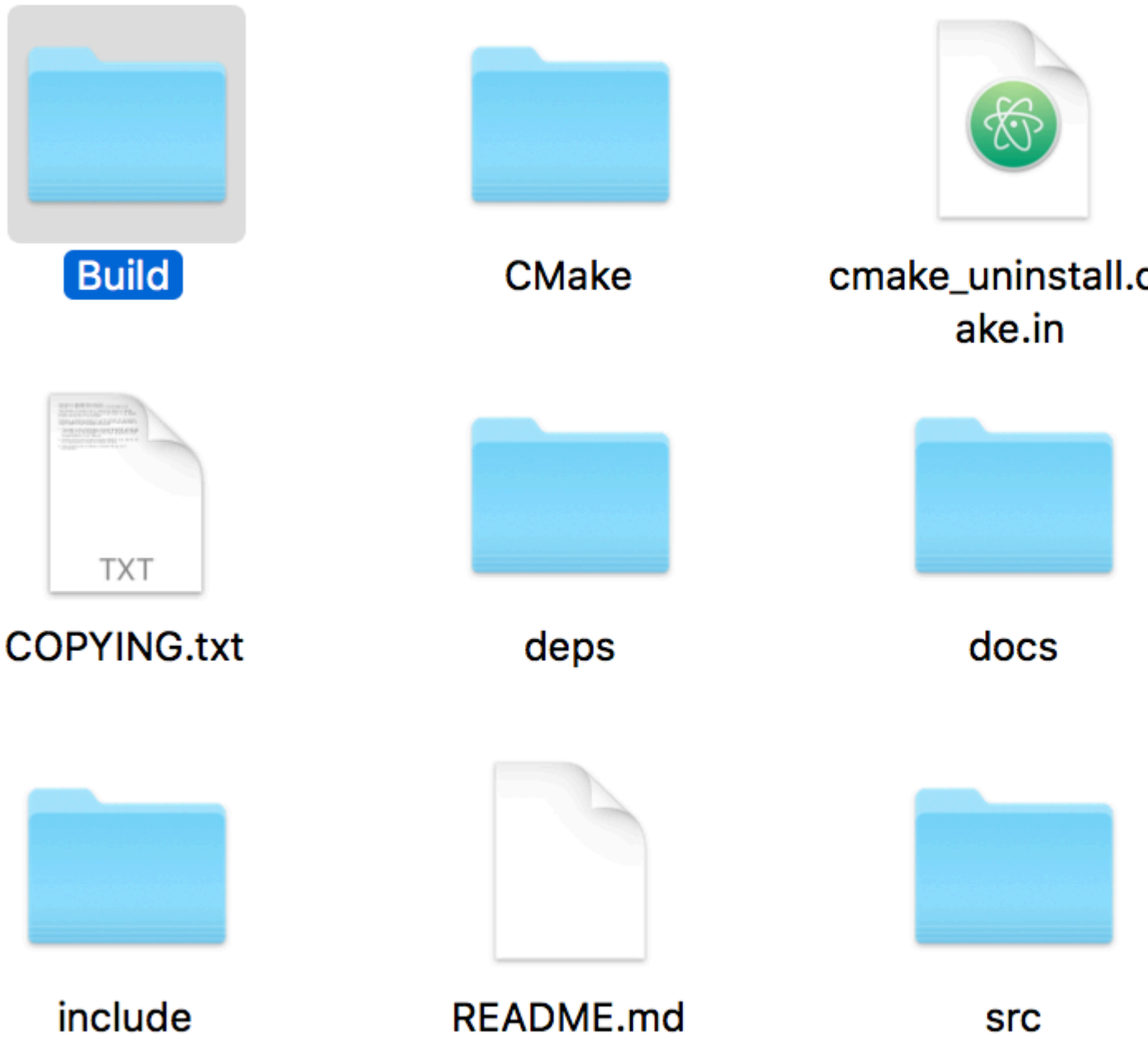


Téléchargez et installez CMake pour construire GLFW. Allez sur www.cmake.org/download/ , téléchargez CMake et installez pour MAC OS X

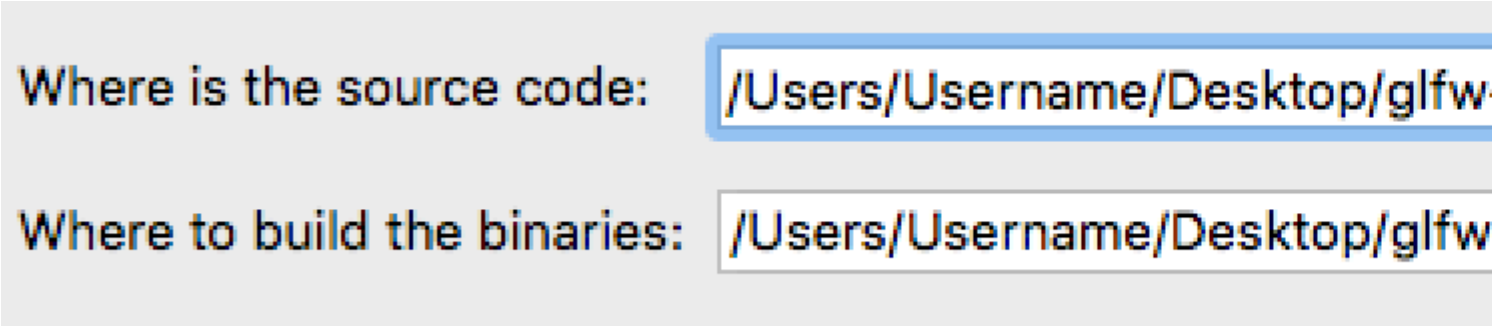


Si Xcode n'est pas installé. Téléchargez et installez Xcode à partir du Mac App Store.

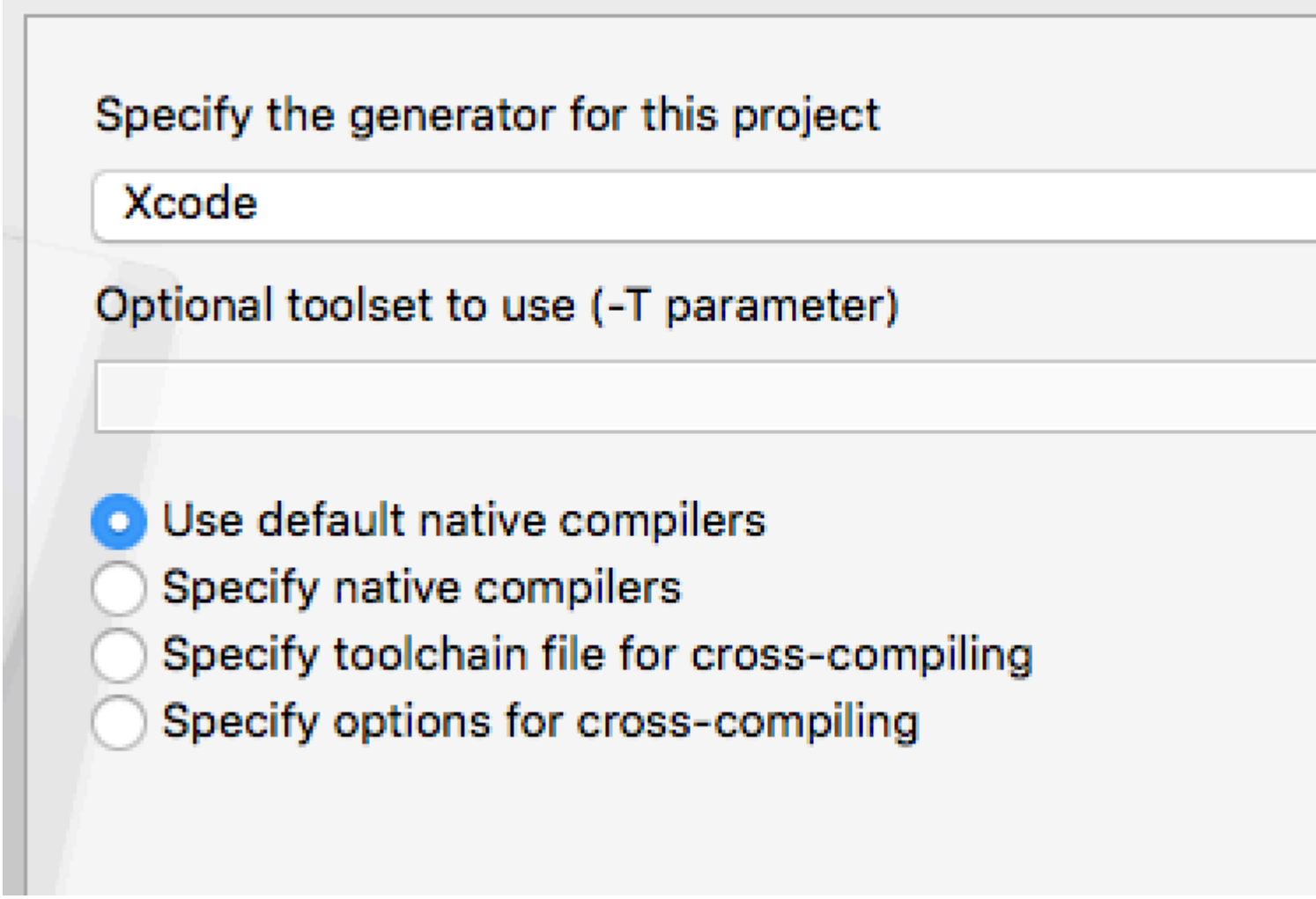
Créer un nouveau dossier **Construire** dans le dossier GLFW



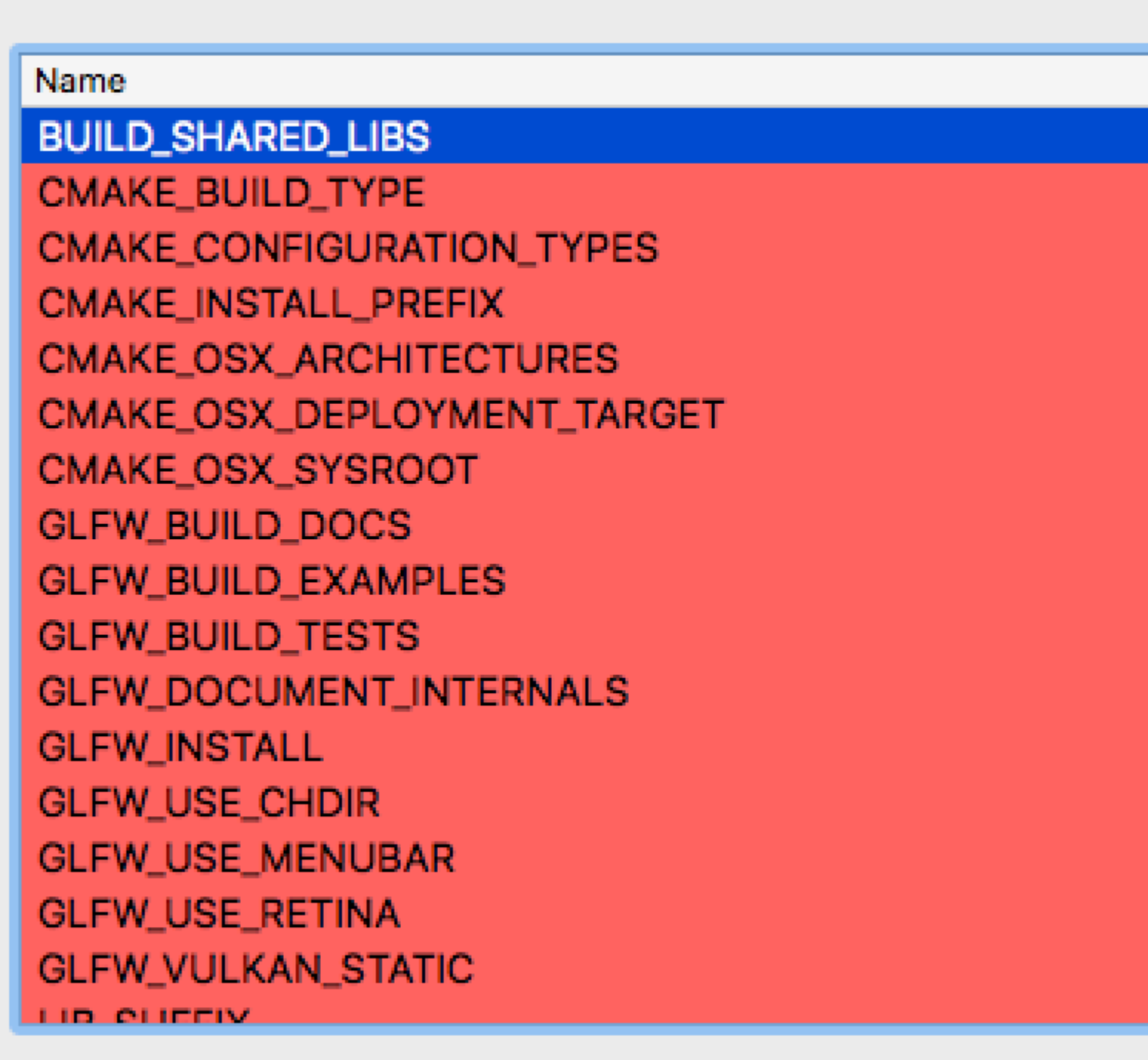
Ouvrez CMake, cliquez sur le bouton **Parcourir la source** pour sélectionner le dossier GLFW (assurez-vous que CMakeLists.txt) se trouve dans ce dossier. Après cela, cliquez sur le bouton **Browse Build** et sélectionnez le dossier **Build** nouvellement créé à l'étape précédente.



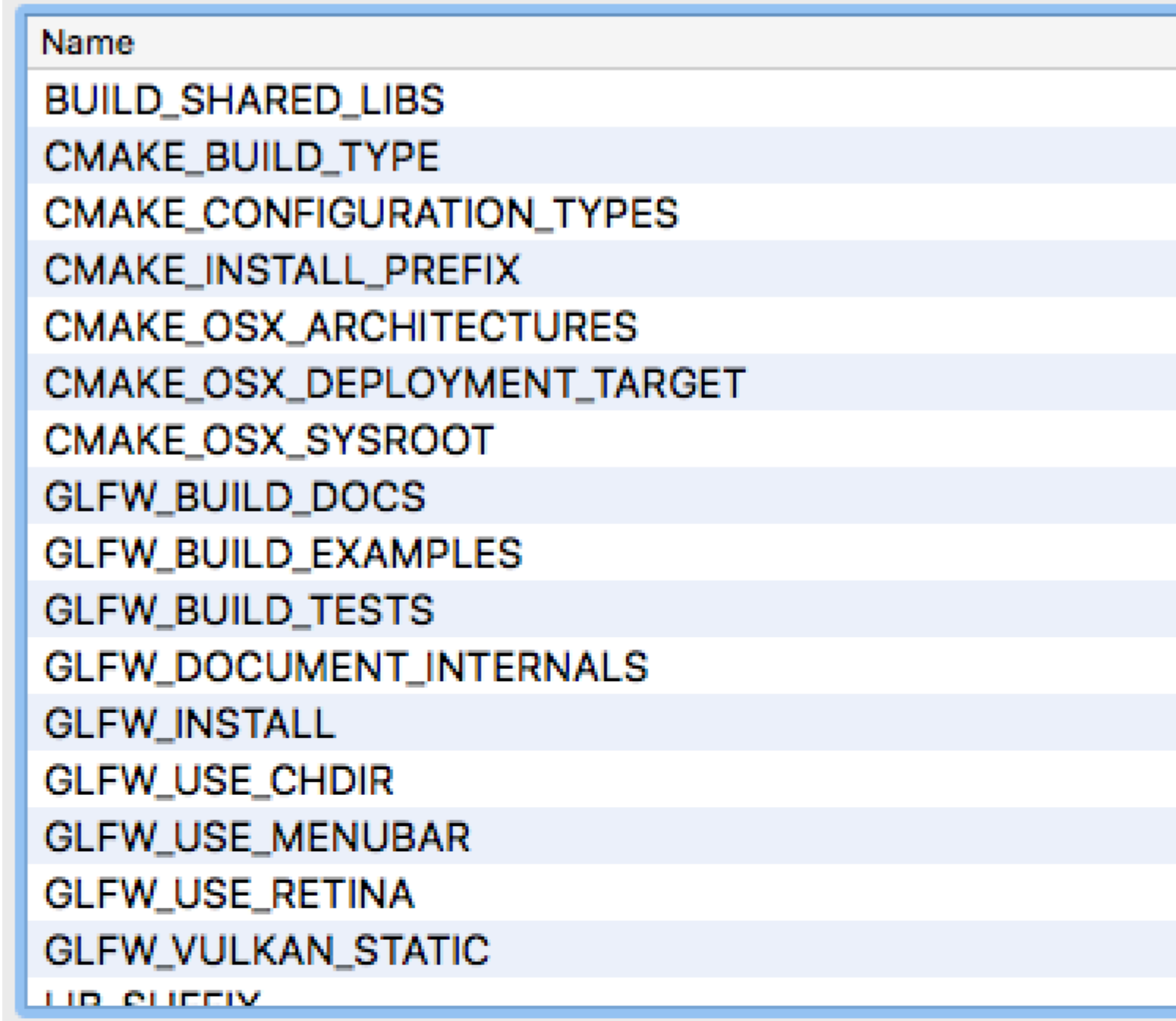
Maintenant Cliquez sur le bouton **Configurer** et sélectionnez **Xcode** comme générateur avec l' **option Utiliser les compilateurs natifs par défaut** , puis cliquez sur **Terminé** .



Cochez l'option **BUILD_SHARED_LIBS** puis cliquez à nouveau sur le bouton **Configurer** et enfin cliquez sur le bouton **Générer** .

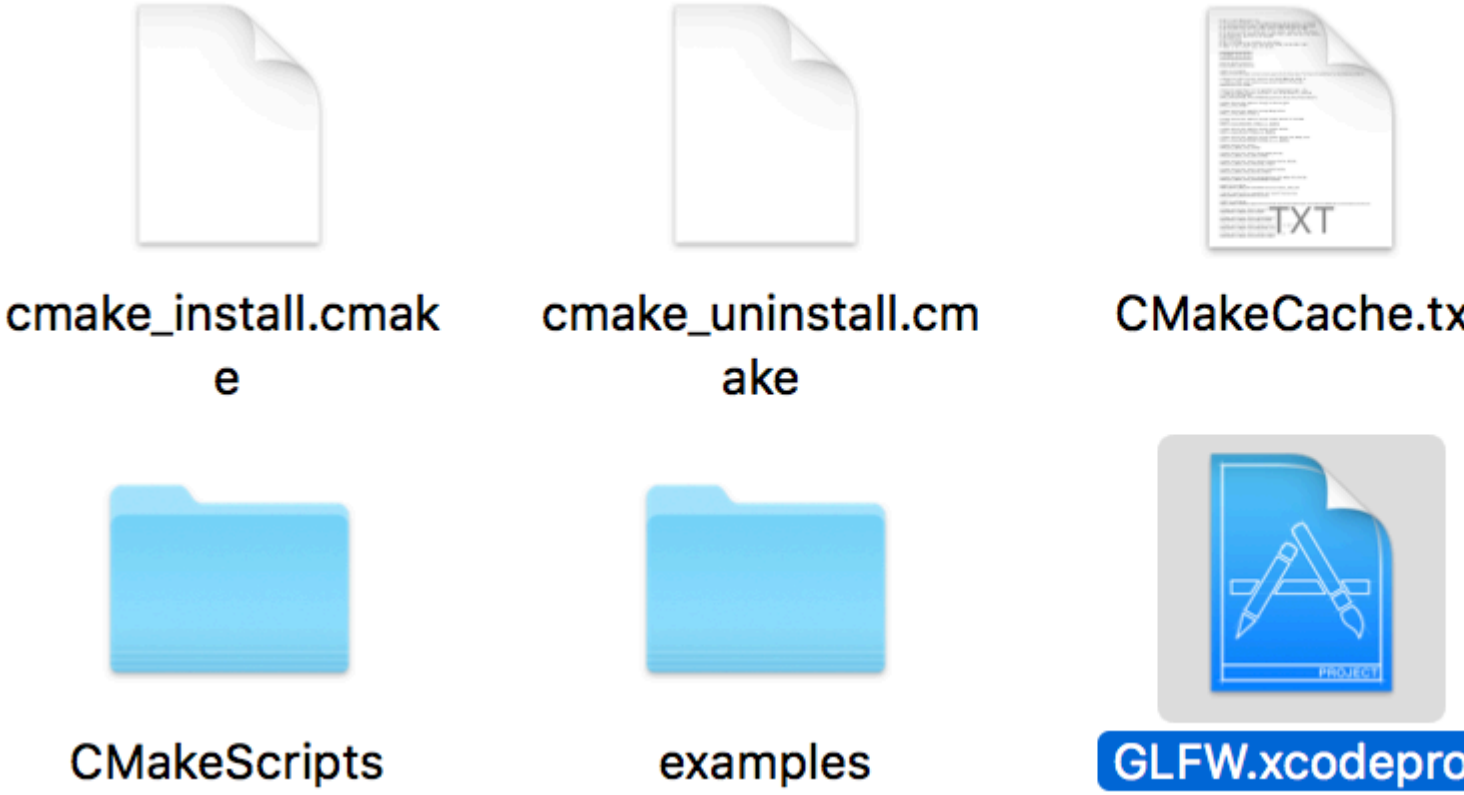


Après la génération CMake devrait ressembler à ceci


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

s'il n'y en a pas déjà. Ouvrez le dossier **local** et créez deux dossiers **include** et **lib** si ce n'est déjà fait.

Maintenant, ouvrez le dossier GLFW et allez dans **Build** (où CMake avait construit les fichiers). Ouvrez le fichier **GLFW.xcodeproj** dans Xcode.



Sélectionnez **Installer> Mon Mac** , puis cliquez sur **Exécuter** (bouton Jouer en forme).

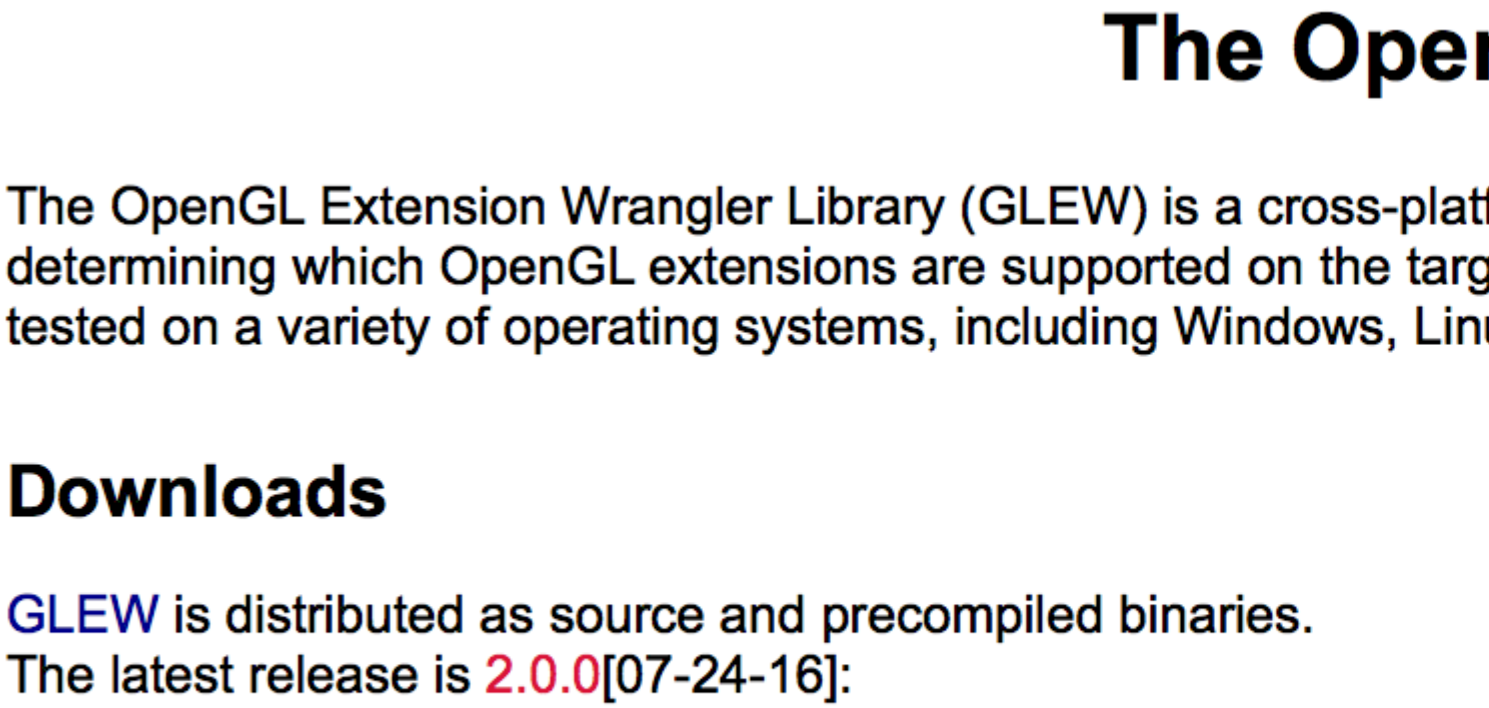
Il est maintenant installé avec succès (ignorez les avertissements).

Pour vous assurer que les dossiers Open Finder et goto **/ usr / local / lib** et trois fichiers de bibliothèque GLFW sont déjà présents (sinon, ouvrez le dossier **Build dans le** dossier GLFW et accédez à **src / Debug,** copiez tous les fichiers dans **/ usr / local / lib).**)

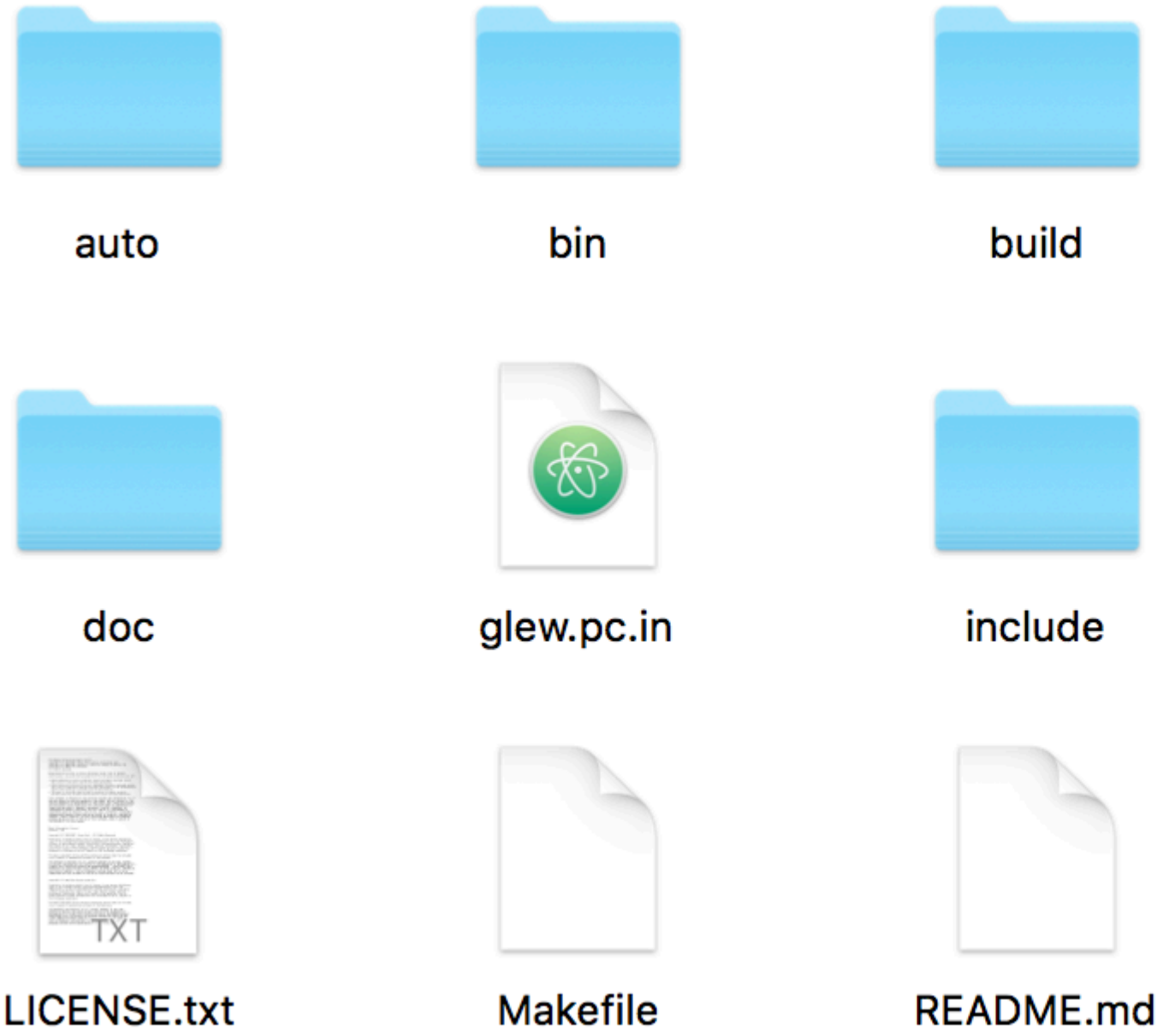
Open Finder et goto **/ usr / local / include** et un dossier GLFW y sera déjà présent avec deux fichiers d'en-tête à l'intérieur, **nommés glfw3.h** et **glfw3native.h**

2. Installez GLEW

GLEW est une bibliothèque multi-plateforme qui permet d'interroger et de charger des extensions OpenGL. Il fournit des mécanismes d'exécution pour déterminer quelles extensions OpenGL sont prises en charge sur la plate-forme cible. Ce n'est que pour OpenGL moderne (OpenGL version 3.2 et supérieure qui nécessite des fonctions à déterminer à l'exécution). Pour installer d'abord, téléchargez ses fichiers depuis glew.sourceforge.net



Extrayez le dossier GLFW et son contenu ressemblera à ceci.



Maintenant, ouvrez Terminal, accédez au dossier GLEW et tapez les commandes suivantes

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

Maintenant, GLEW est installé avec succès. Pour vous assurer qu'il est installé, ouvrez Finder, accédez à **/ usr / local / include** et un dossier GL y sera déjà présent avec trois fichiers d'en-tête à l'intérieur, **nommés glew.h** , **glxew.h** et **wglew.h.**

glew.h

glxew.h

wglew.h

Ouvrez le Finder et accédez à **/ usr / local / lib** et les fichiers de la bibliothèque GLEW y seront déjà présents.

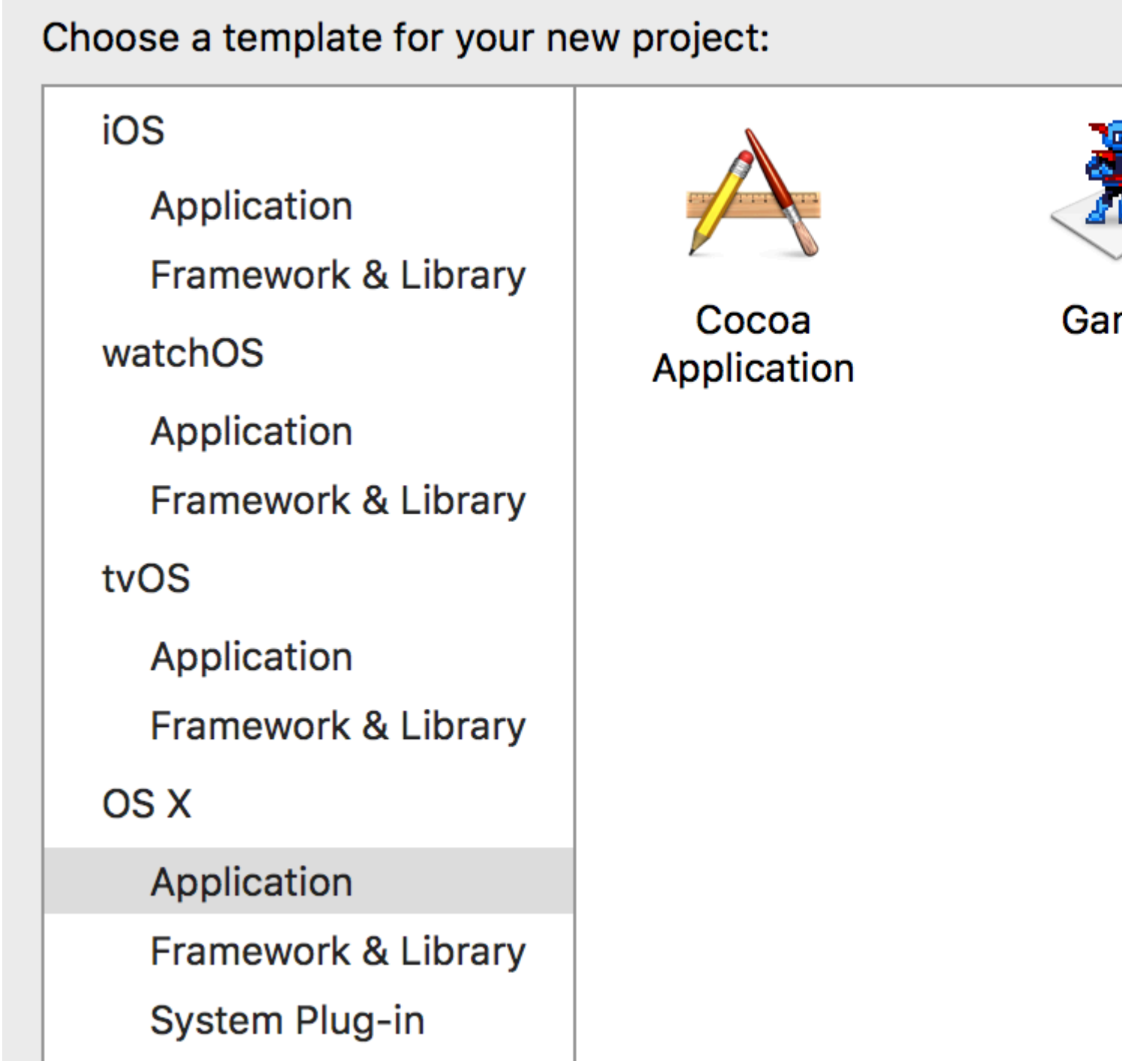
libGLEW.2.0.dylib

libGLEW.a

libGLEW.dylib

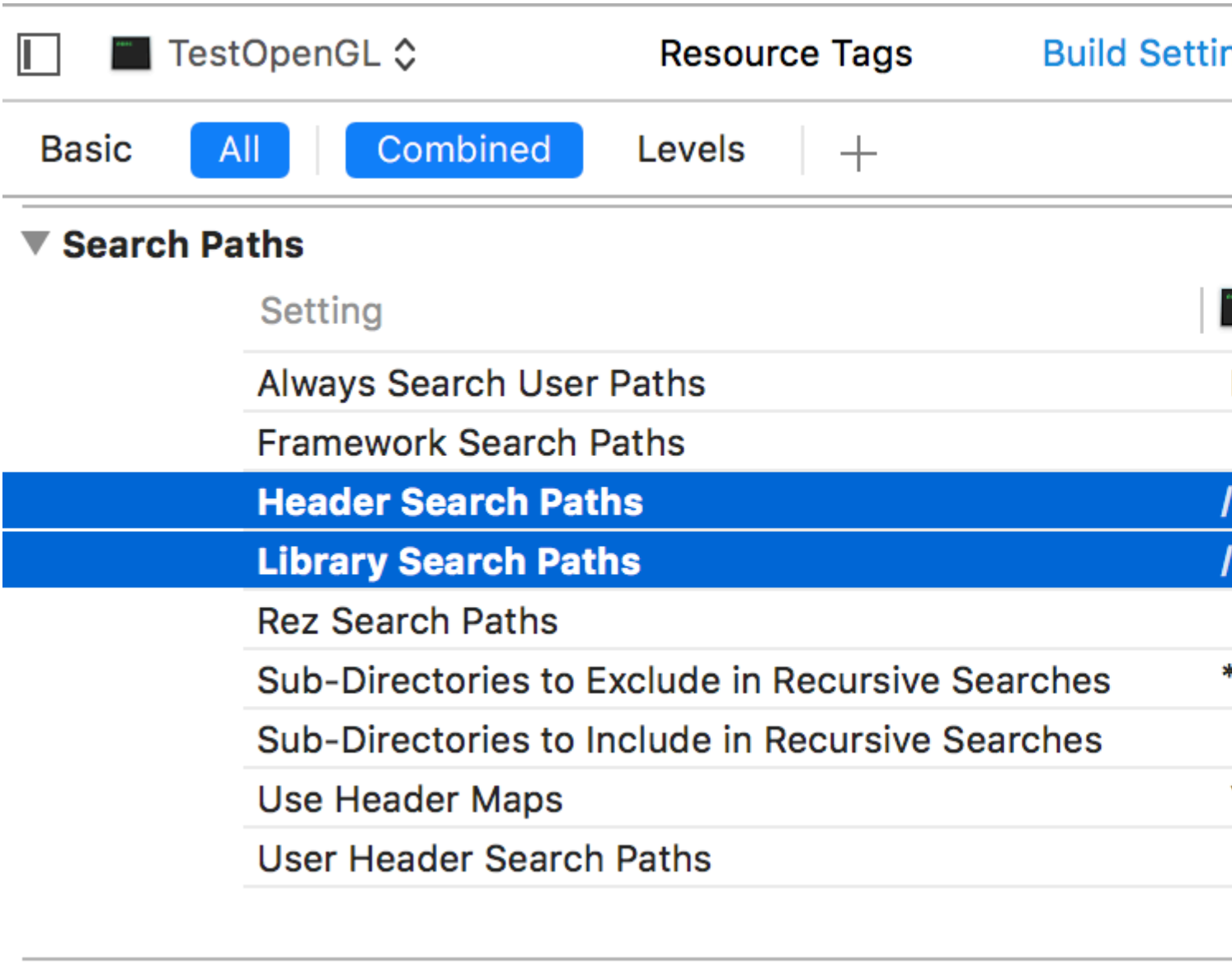
3. Tester et exécuter

Nous avons maintenant installé avec succès GLFW et GLEW. Il est temps de coder. Ouvrez Xcode et créez un nouveau projet Xcode. Sélectionnez l' **outil Ligne de commande,** puis continuez et sélectionnez **C ++** comme langage.

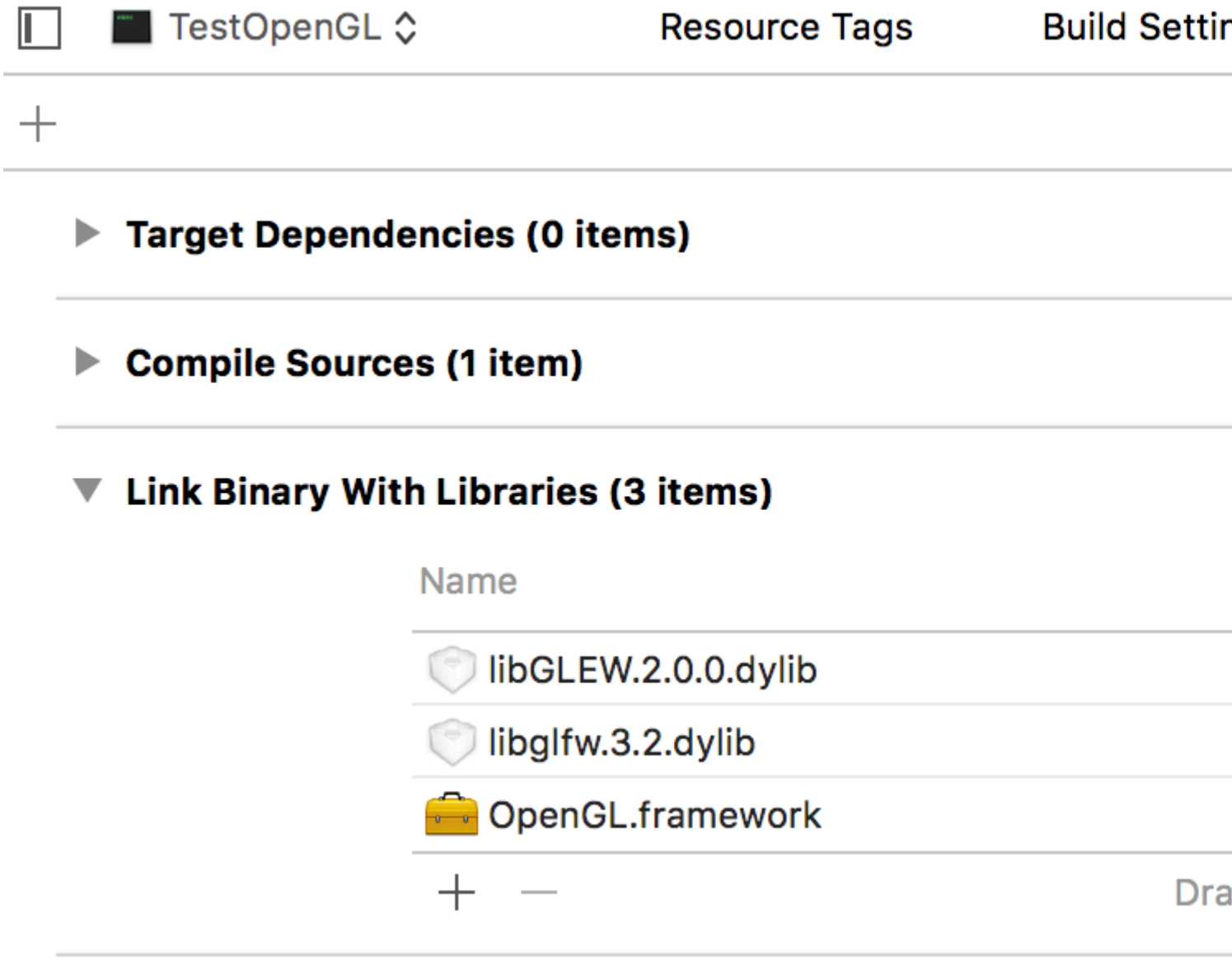


Xcode créera un nouveau projet de ligne de commande.

Cliquez sur le nom du projet et, sous l'onglet **Paramètres de** génération, passez de **Basique à Tous** , dans la section **Chemins de recherche** , ajoutez **/ usr / local / include** dans les chemins de recherche d'en-tête et ajoutez **/ usr / local / lib** dans les chemins de recherche de bibliothèque.



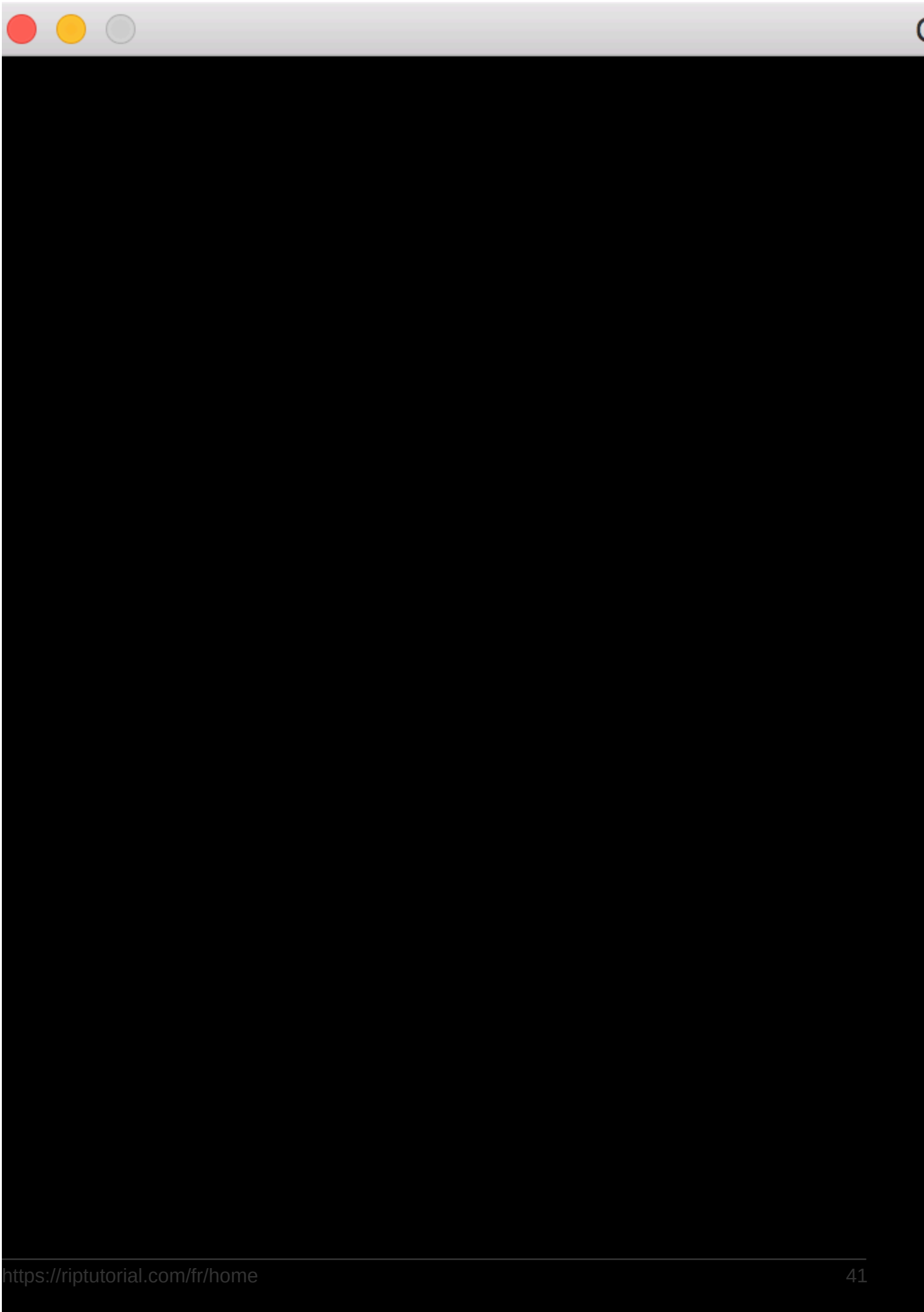
Cliquez sur le nom du projet et, sous l'onglet **Construire des phases** et sous **Lien avec les bibliothèques binaires,** ajoutez **OpenGL.framework** et ajoutez également les bibliothèques **GLFW** et **GLEW** récemment créées à partir de **/ usr / local / lib.**



Nous sommes maintenant prêts à coder dans Modern Open GL 4.1 sur macOS en utilisant C ++ et Xcode. Le code suivant créera une fenêtre OpenGL utilisant GLFW avec une sortie d'écran vide.

```
#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/fr/opengl/topic/814/demarrer-avec-opengl

Chapitre 2: Création de contexte OpenGL.

Examples

Créer une fenêtre de base

Ce tutoriel suppose que vous avez déjà un environnement OpenGL opérationnel avec toutes les bibliothèques et les en-têtes nécessaires 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
}
```

Ajout d'indices à la fenêtre

```
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
```

Lire Création de contexte OpenGL. en ligne: https://riptutorial.com/fr/opengl/topic/6194/creation-de-contexte-opengl-

Chapitre 3: Éclairage de base

Examples

Modèle d'éclairage Phong

> REMARQUE: Cet exemple est WIP, il sera mis à jour avec des diagrammes, des images, d'autres exemples, etc.

Qu'est ce que Phong?

Phong est un modèle de lumière très basique mais réel pour les surfaces à trois parties: éclairage ambiant, diffus et spéculaire.

Éclairage ambiant:

L'éclairage ambiant est la plus simple des trois parties à comprendre et à calculer. L'éclairage ambiant est une lumière qui inonde la scène et illumine l'objet de manière uniforme dans toutes les directions.

Les deux variables de l'éclairage ambiant sont la force de la température ambiante et la couleur de la pièce. Dans votre fragment shader, ce qui suit fonctionnera pour ambient:

```
in vec3 objColor;

out vec3 finalColor;

uniform vec3 lightColor;

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

Éclairage diffus:

L'éclairage diffus est légèrement plus complexe que l'ambiant. L'éclairage diffus est une lumière directionnelle, ce qui signifie essentiellement que les visages dirigés vers la source de lumière seront mieux éclairés et que les visages dirigés vers l'extérieur seront plus sombres en raison de la manière dont la lumière les frappe.

Remarque: un éclairage diffus nécessitera l'utilisation de normales pour chaque face que je ne montrerai pas comment calculer ici. Si vous voulez apprendre à faire cela, consultez la page de mathématiques 3D.

Pour modéliser la réflexion de la lumière dans l'infographie, on utilise une fonction de distribution de la réflectance bidirectionnelle (BRDF). BRDF est une fonction qui donne la relation entre la lumière réfléchie le long d'une direction sortante et la lumière incidente provenant d'une direction

entrante.

Une surface diffuse parfaite a un BRDF qui a la même valeur pour toutes les directions incidentes et sortantes. Cela réduit considérablement les calculs et, par conséquent, il est couramment utilisé pour modéliser les surfaces diffuses car il est physiquement plausible, même s'il n'y a pas de matériaux diffus purs dans le monde réel. Cette BRDF est appelée réflexion Lambertienne car elle obéit à la loi de cosinus de Lambert.

La réflexion de Lambert est souvent utilisée comme modèle pour la réflexion diffuse. Cette technique fait en sorte que tous les polygones fermés (tels qu'un triangle dans un maillage 3D) réfléchissent la lumière de la même manière dans toutes les directions. Le coefficient de diffusion est calculé à partir de l'angle entre le vecteur normal et le vecteur lumineux.

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

où `N` est le vecteur normal de la surface et `L` le vecteur vers la source de lumière.

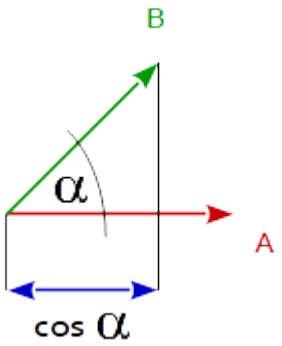
Comment ça marche

En général Le produit *scalaire* de 2 vecteurs est égal au *cosinus* de l'angle entre les 2 vecteurs multiplié par la grandeur (longueur) des deux vecteurs.

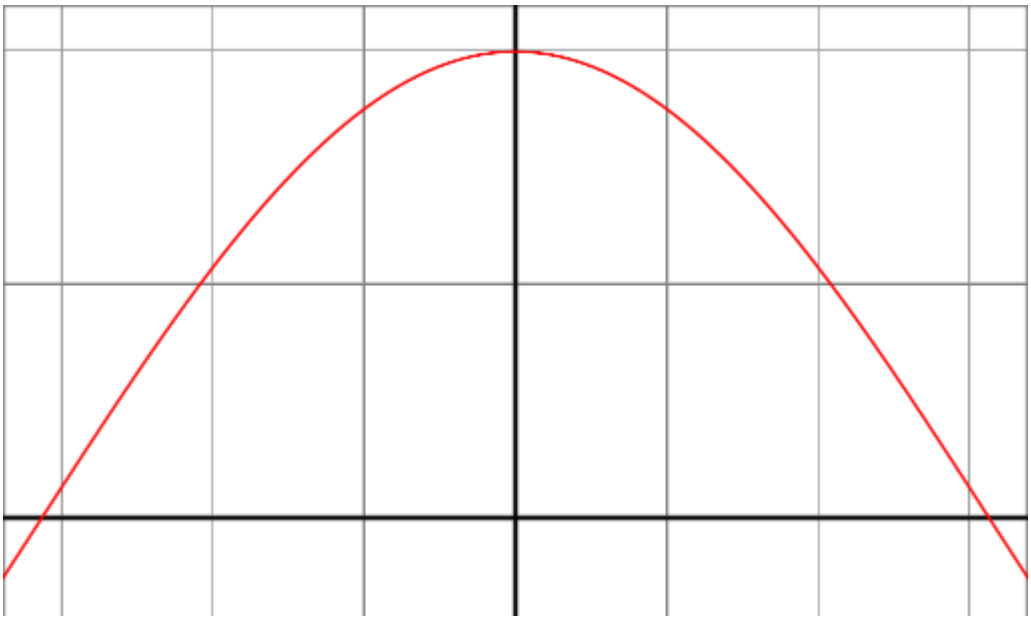
```
dot( A, B ) == length( A ) * length( B ) * cos( angle_A_B )
```

Ceci fait que le produit *scalaire* de 2 vecteurs unitaires est égal au *cosinus* de l'angle entre les 2 vecteurs, car la longueur d'un vecteur unitaire est 1.

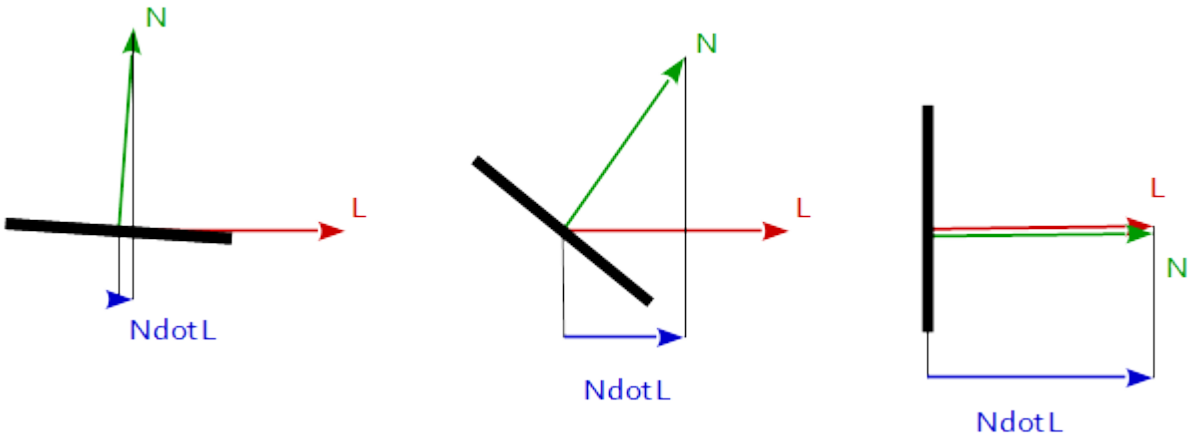
```
uA = normalize( A )
uB = normalize( B )
cos( angle_A_B ) == dot( uA, uB )
```



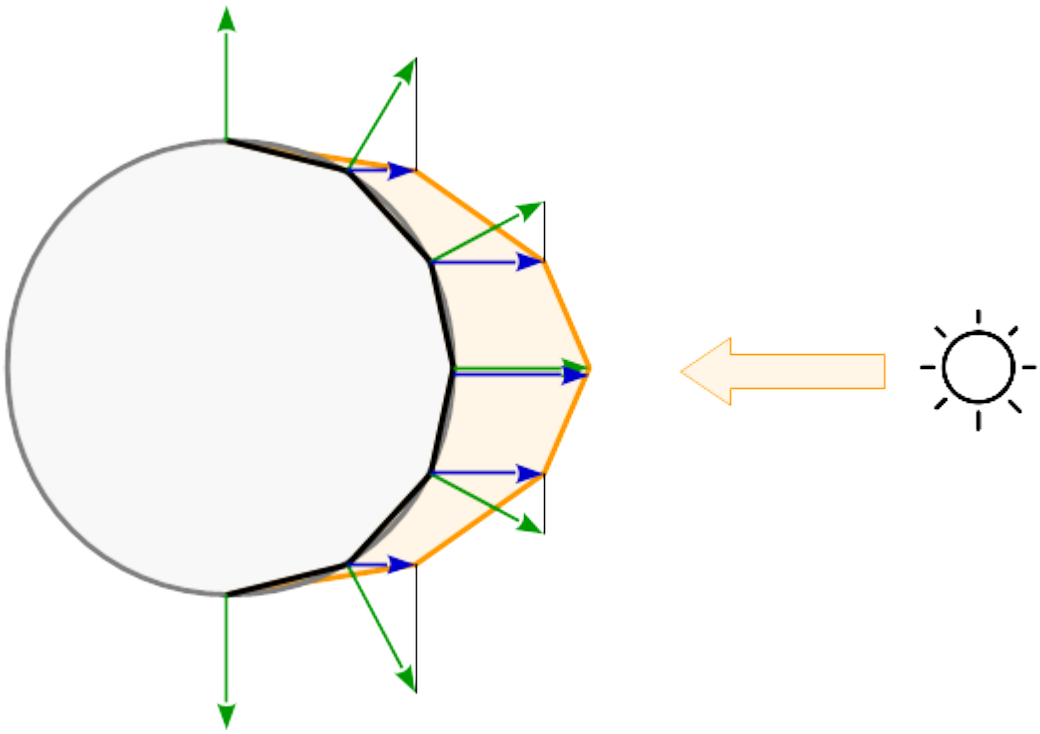
Si on regarde la fonction *cos (x)* entre les angles -90 ° et 90 °, on voit qu'elle a un maximum de 1 à un angle de 0 ° et qu'elle descend à 0 aux angles de 90 ° et -90 °.



Ce comportement est exactement ce que nous voulons pour le modèle de réflexion. Lorsque le vétor nromal de la surface et le diretion à la source lumineuse sont dans le même sens (l'angle compris entre 0 °), nous voulons un maximum de réflexion. En revanche, si les vecteurs sont orthonormés (l'angle compris entre 90 °), nous souhaitons un minimum de réflexion et nous souhaitons un fonctionnement fonctionnel régulier et continu entre les deux bordures de 0 ° et de 90 °.



Si la lumière est calculée par sommet, la réflexion est calculée pour chaque coin de la primitive. Entre les primitives, les réflexions sont interpolées en fonction de ses coordonnées barycentriques. Voir les reflets résultants sur une surface sphérique:



Ok, donc pour commencer avec notre fragment shader, nous aurons besoin de quatre entrées.

- Normales de sommet (devraient être dans le tampon et spécifiées par les pointeurs d'attribut de sommet)
- Position du fragment (doit être sortie du vertex shader dans frag shader)
- Position de la source lumineuse (uniforme)
- Couleur claire (uniforme)

```
in vec3 normal;
in vec3 fragPos;

out vec3 finalColor;

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

À l'intérieur du bâtiment principal, nous devons faire des calculs. Tout le concept d'éclairage diffus est basé sur l'angle entre la direction normale et la direction de la lumière. Plus l'angle est grand, moins il y a de lumière jusqu'à 90 ° où il n'y a pas de lumière du tout.

Avant de pouvoir commencer à calculer la quantité de lumière, nous avons besoin du vecteur de direction de la lumière. Cela peut être récupéré en soustrayant simplement la position de la lumière à la position du fragment qui renvoie un vecteur de la position de la lumière pointant vers la position du fragment.

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

Aussi, allez de l'avant et normalisez les vecteurs `normal` et `lightDir` pour qu'ils fonctionnent avec la même longueur.

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

Maintenant que nous avons nos vecteurs, nous pouvons calculer la différence entre eux. Pour ce faire, nous allons utiliser la fonction de produit scalaire. Fondamentalement, cela prend 2 vecteurs et renvoie le cos () de l'angle formé. Ceci est parfait car à 90 degrés, cela donnera 0 et à 0 degré, cela donnera 1. En conséquence, lorsque la lumière pointe directement sur l'objet, elle sera complètement allumée et vice versa.

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

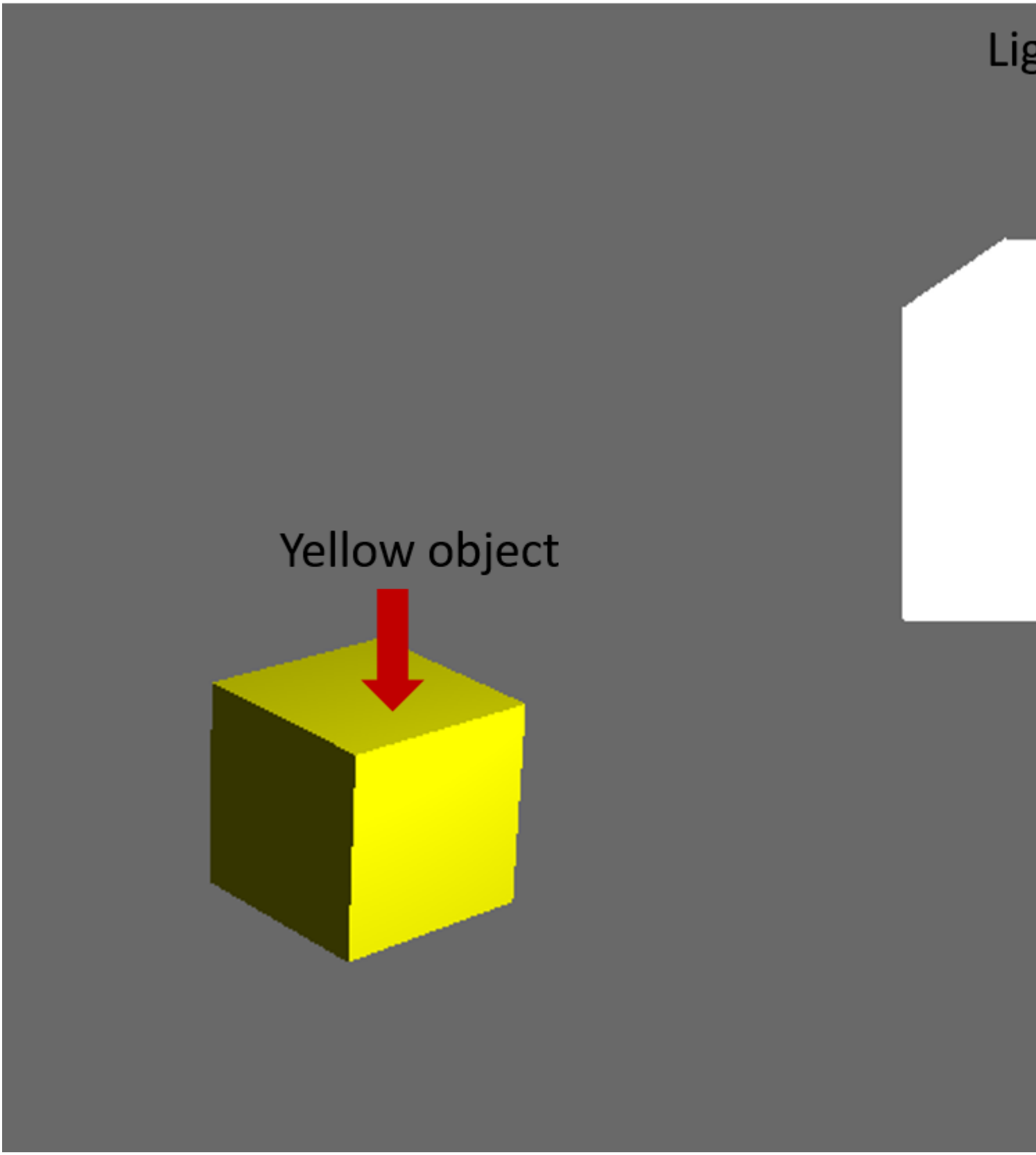
Il y a encore une chose à faire pour le nombre calculé, nous devons nous assurer qu'il est toujours positif. Si vous y réfléchissez, un nombre négatif n'a pas de sens dans le contexte car cela signifie que la lumière est derrière le visage. Nous pourrions utiliser une instruction if ou utiliser la fonction `max()` qui renvoie le maximum de deux entrées.

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

Ceci étant fait, nous sommes maintenant prêts à calculer la couleur de sortie finale du fragment.

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

Ça devrait ressembler à ça:



Éclairage spéculaire:

Travailler en cours, revenez plus tard.

Combiné

Travaux en cours, revenez plus tard.

Le code et l'image ci-dessous montrent ces trois concepts d'éclairage combinés.

Lire Éclairage de base en ligne: https://riptutorial.com/fr/opengl/topic/4209/eclairage-de-base

Chapitre 4: Encapsulation d'objets OpenGL avec C ++ RAII

Introduction

Exemples de différentes manières d'utiliser les objets OpenGL avec C ++ RAII.

Remarques

L'encapsulation RAII des objets OpenGL présente des dangers. Le plus inévitable est que les objets OpenGL sont associés au contexte OpenGL qui les a créés. La destruction d'un objet C ++ RAII doit donc se faire dans un contexte OpenGL qui partage la propriété de l'objet OpenGL géré par cet objet C ++.

Cela signifie également que si tous les contextes qui possèdent l'objet sont détruits, tous les objets OpenGL encapsulés dans RAII existants tenteront de détruire les objets qui n'existent plus.

Vous devez prendre des mesures manuelles pour traiter les problèmes de contexte comme celui-ci.

Examples

En C ++ 98/03

L'encapsulation d'un objet OpenGL dans C ++ 98/03 nécessite d'obéir à la règle C ++ de 3. Cela signifie qu'il faut ajouter un constructeur de copie, un opérateur d'attribution de copie et un destructeur.

Cependant, les constructeurs de copie doivent copier logiquement l'objet. Et copier un objet OpenGL est une entreprise non triviale. Tout aussi important, c'est presque certainement quelque chose que l'utilisateur ne souhaite pas faire.

Nous allons donc rendre l'objet non 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 &);
};
```

Le constructeur va créer l'objet et initialiser les données de l'objet tampon. Le destructeur détruira l'objet. En déclarant le constructeur / affectation de copie *sans les* définir, le lieur génère une erreur si un code tente de les appeler. Et en les déclarant privés, seuls les membres de `BufferObject` pourront même les appeler.

Notez que `BufferObject` ne conserve pas la `target` transmise au constructeur. En effet, un objet tampon OpenGL peut être utilisé avec n'importe quelle cible, pas seulement celle avec laquelle il a été initialement créé. Ceci est différent des objets de texture, qui doivent toujours être liés à la cible avec laquelle ils ont été initialement créés.

Comme OpenGL est très dépendant de la liaison d'objets au contexte à des fins diverses, il est souvent utile d'avoir également une liaison d'objets de type RAII. Étant donné que différents objets ont des besoins de liaison différents (certains ont des cibles, d'autres non), nous devons en implémenter un pour chaque objet individuellement.

```
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` n'est pas copiable, car la copier n'a aucun sens. Notez qu'il ne conserve pas l'accès au `BufferObject` qu'il lie. C'est parce que c'est inutile.

En C ++ 11 et versions ultérieures

C ++ 11 offre des outils qui améliorent les fonctionnalités des objets OpenGL encapsulés dans RAII. Sans les fonctionnalités C ++ 11 telles que la sémantique de déplacement, ces objets devraient être dynamiquement alloués si vous voulez les faire circuler, car ils ne peuvent pas être copiés. Le support de déplacement leur permet de passer comme des valeurs normales, mais pas en les copiant:

```
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_;
};
```

Un tel type peut être renvoyé par une fonction:

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

Ce qui vous permet de les stocker dans vos propres types (implicitement en déplacement uniquement):

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

Une classe de classeur de portée peut également avoir une sémantique de déplacement, permettant ainsi au classeur d'être renvoyé par des fonctions et stocké dans des conteneurs de bibliothèque standard 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_;
};
```

Notez que l'objet est déplaçable mais non déplaçable. L'idée est d'empêcher la reliure d'une liaison de tampon de portée. Une fois qu'il est défini, la seule chose qui peut le désélectionner est d'être déplacé.

Lire Encapsulation d'objets OpenGL avec C ++ RAII en ligne:
https://riptutorial.com/fr/opengl/topic/7556/encapsulation-d-objets-opengl-avec-c-plusplus-raii

Chapitre 5: Framebuffers

Examples

Bases du framebuffer

Le framebuffer est un type de tampon qui stocke **les** valeurs de **couleur** , les informations de **profondeur** et de **gabarit** des pixels en mémoire. Lorsque vous dessinez quelque chose dans OpenGL, la sortie est stockée dans le *framebuffer par défaut* , puis vous voyez réellement les valeurs de couleur de ce tampon à l'écran. Vous pouvez également créer votre propre framebuffer qui peut être utilisé pour beaucoup d'effets de post-traitement, tels que les *niveaux* de *gris, le flou, la profondeur de champ, les distorsions, les réflexions* ...

Pour commencer, vous devez créer un objet framebuffer (**FBO**) et le lier comme tout autre objet dans OpenGL:

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

Maintenant, vous devez ajouter au moins une **pièce jointe** (couleur, profondeur ou gabarit) au framebuffer. Une pièce jointe est un emplacement de mémoire qui agit comme un tampon pour le framebuffer. Il peut s'agir d'une *texture* ou d'un *objet renderbuffer* . L'avantage d'utiliser une texture est que vous pouvez facilement utiliser cette texture dans un shaders de post-traitement. La création de la texture est similaire à une texture normale:

```
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);
```

La `width` et la `height` doivent être identiques à la taille de la fenêtre de rendu. Le pointeur de données de texture est `NULL` car vous souhaitez uniquement allouer la mémoire et ne pas remplir la texture avec des données. La texture est prête pour que vous puissiez réellement l'attacher au framebuffer:

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

Votre framebuffer devrait être prêt à être utilisé maintenant, mais vous souhaiterez peut-être également ajouter des attachements de profondeur ou des attachements de profondeur et de gabarit. Si vous souhaitez les ajouter en tant que pièces jointes de texture (et les utiliser pour

certains traitements), vous pouvez créer d'autres textures comme ci-dessus. La seule différence serait dans ces lignes:

```
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);
```

Ou ceux-ci si vous souhaitez utiliser l'attachement de profondeur **et de** gabarit dans une texture unique:

```
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);
```

Vous pouvez également utiliser un **renderbuffer** au lieu d'une texture en tant que pièce jointe pour les tampons de profondeur et de gabarit si vous ne souhaitez pas traiter les valeurs ultérieurement. (Cela sera expliqué dans un autre exemple ...)

Vous pouvez vérifier si le framebuffer a été créé et exécuté sans erreur:

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

Et enfin, n'oubliez pas de délier le framebuffer pour ne pas lui rendre accidentellement:

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

Limites

Le nombre maximal de tampons de couleur pouvant être attachés à un seul tampon de trame peut être déterminé par la fonction OGL glGetIntegerv , en utilisant le paramètre `GL_MAX_COLOR_ATTACHMENTS` :

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

Utiliser le framebuffer

L'utilisation est assez simple. Tout d'abord, vous liez votre **framebuffer** et y rendez votre scène. Mais vous ne verrez rien encore car votre tampon de rendu n'est pas visible. La deuxième partie consiste donc à rendre votre framebuffer comme une texture de quad plein écran sur l'écran.

Vous pouvez simplement le rendre tel quel ou effectuer des effets de post-traitement.

Voici les sommets pour un quad plein écran:

```
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
};
```

Vous devrez les stocker dans un VBO ou les rendre en utilisant des pointeurs d'attributs. Vous aurez également besoin d'un programme de shader de base pour le rendu du quad plein écran avec texture.

Vertex Shader:

```
in vec2 position;
in vec2 texCoords;

out vec2 TexCoords;

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

Fragment shader:

```
in vec2 TexCoords;
out vec4 color;

uniform sampler2D screenTexture;

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

Note: Vous devrez peut-être ajuster les shaders pour votre version de *GLSL* .

Maintenant, vous pouvez faire le rendu réel. Comme décrit ci-dessus, la première chose à faire est de rendre la scène dans votre FBO. Pour ce faire, il vous suffit de lier votre FBO, de le vider et de dessiner la scène:

```
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...
```

Remarque: Dans la fonction `glClear` , vous devez spécifier toutes les pièces jointes du framebuffer que vous utilisez (Dans cet exemple, pièce jointe couleur et profondeur).

Maintenant, vous pouvez rendre votre FBO comme un quad plein écran sur le framebuffer par défaut pour que vous puissiez le voir. Pour ce faire, il vous suffit de déconnecter votre FBO et de rendre le 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);
```

Et c'est tout! Si vous avez tout fait correctement, vous devriez voir la même scène qu'auparavant, mais sur un quad plein écran. La sortie visuelle est la même que précédemment, mais vous pouvez désormais facilement ajouter des effets de post-traitement simplement en modifiant le fragment shader. (Je vais ajouter des effets dans un autre exemple et le lier ici)

Lire Framebuffers en ligne: https://riptutorial.com/fr/opengl/topic/7038/framebuffers

Chapitre 6: L'instanciation

Introduction

L'instanciation est une technique de rendu qui permet de dessiner plusieurs copies du même objet lors d'un appel. Il est généralement utilisé pour rendre des particules, du feuillage ou de grandes quantités de tout autre type d'objets.

Examples

Instillation par tableaux d'attributs de vertex

3.3

L'installation peut être effectuée via des modifications de la manière dont les attributs de vertex sont fournis au vertex shader. Cela introduit une nouvelle façon d'accéder aux tableaux d'attributs, leur permettant de fournir des données par instance qui ressemblent à un attribut normal.

Une seule instance représente un objet ou un groupe de sommets (une feuille d'herbe, etc.). Les attributs associés aux tableaux instanciés ne progressent qu'entre les instances. contrairement aux attributs de vertex réguliers, ils n'obtiennent pas de nouvelle valeur par sommet.

Pour spécifier qu'un tableau d'attributs est instancié, utilisez cet appel:

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

Ceci définit l'état d'objet du tableau de vertex. Le "1" signifie que l'attribut est avancé pour chaque instance. Passer un 0 désactive l'instanciation de l'attribut.

Dans le shader, l'attribut instancié ressemble à tout autre attribut vertex:

```
in vec3 your_instanced_attribute;
```

Pour rendre plusieurs instances, vous pouvez appeler l'une des formes `Instanced` de la valeur appels `glDraw*` . Par exemple, cela va dessiner 1000 instances, chaque instance étant constituée de 3 sommets:

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

Code de tableau instancié

Mise en place de VAO, de VBO et des attributs:

```
// 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);
```

Tirage au sort:

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

Vertex Shader:

```
#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);
}
```

Fragment shader:

```
#version 330 core
```

```
layout(location = 0) out vec4 color;

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

Lire L'instanciation en ligne: https://riptutorial.com/fr/opengl/topic/8647/l-instanciation

Chapitre 7: Math 3d

Examples

Introduction aux matrices

Lorsque vous programmez en OpenGL ou toute autre API graphique, vous frappez un mur de briques lorsque vous n'êtes pas si bon en maths. Ici, je vais expliquer avec un exemple de code comment vous pouvez réaliser un mouvement / une mise à l'échelle et bien d'autres choses intéressantes avec votre objet 3D.

Prenons un cas réel ... Vous avez créé un cube impressionnant (en trois dimensions) dans OpenGL et vous voulez le déplacer dans n'importe quelle direction.

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

Dans les moteurs de jeu comme Unity3d, ce serait facile. Vous appelez simplement transform.Translate () et vous en avez fini, mais OpenGL n'inclut pas de bibliothèque mathématique.

Une bonne bibliothèque de maths est glm mais pour vous faire comprendre, je coderai toutes les méthodes mathématiques (importantes) pour vous.

Tout d'abord, nous devons comprendre qu'un objet 3D dans OpenGL contient beaucoup d'informations, il y a beaucoup de variables qui dépendent les unes des autres. Une manière intelligente de gérer toutes ces variables consiste à utiliser des matrices.

Une matrice est une collection de variables écrites en colonnes et en lignes. Une matrice peut être 1x1, 2x4 ou n'importe quel nombre arbitraire.

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

Vous pouvez faire des choses vraiment cool avec eux ... mais comment peuvent-ils m'aider à déplacer mon cube? Pour comprendre cela, il faut d'abord connaître plusieurs choses.

- Comment faites-vous une matrice à partir d'une position?
- Comment traduisez-vous une matrice?
- Comment le transmettez-vous à OpenGL?

Faisons une classe contenant toutes nos données et méthodes matricielles importantes (écrites 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
        };
    }

}
```

Nous remarquons d'abord une chose très particulière dans le constructeur par défaut d'une matrice 4 par 4. Lorsqu'il est appelé, il ne démarre pas tous sur zéro mais comme:

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

Toutes les matrices doivent commencer par celles de la diagonale. (juste parce que>. <)

Bon alors déclarons à notre cube épique une matrice 4 par 4.

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

Maintenant, nous avons toutes nos variables, nous pouvons enfin commencer à faire des calculs! Faisons la traduction. Si vous avez programmé dans Unity3d, vous pourriez vous souvenir d'une fonction Transform.Translate. Implémentons-le dans notre propre classe de matrice

```
 /*!
 *  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;
}
```

C'est tout le calcul nécessaire pour déplacer le cube (pas de rotation ni de mise à l'échelle). Cela fonctionne à tous les angles. Implémentons cela dans notre scénario réel

```
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)
```

Notre shader doit utiliser notre merveilleuse matrice

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

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

Et ça devrait marcher .... mais il semble que nous ayons déjà un bug dans notre programme. Lorsque vous vous déplacez le long de l'axe z, votre objet semble disparaître dans l'air. C'est parce que nous n'avons pas de matrice de projection. Pour résoudre ce problème, nous devons connaître deux choses:

1. A quoi ressemble une matrice de projection?
2. Comment pouvons-nous le combiner avec notre matrice de position?

Eh bien, nous pouvons faire une perspective (nous utilisons trois dimensions après tout) matrice Le code

```
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));
}
```

Cela semble effrayant, mais cette méthode calcule en fait une matrice de la distance à parcourir (et de la distance) et de votre champ de vision.

Nous avons maintenant une matrice de projection et une matrice de position. Mais comment les combinons-nous? Ce qui est amusant, c'est que nous pouvons multiplier deux matrices les unes avec les autres.

```
/*!
 *  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 .. c'est beaucoup de code qui semble plus effrayant que ça. Cela peut être fait dans une boucle for, mais j'ai pensé (probablement à tort) que cela serait plus clair pour les personnes qui n'ont jamais travaillé avec des matrices.

Regardez le code et remarquez un motif répétitif. Multipliez la colonne par la ligne ajoutez-la et continuez (c'est la même chose pour toute matrice de taille)

* Notez que la multiplication avec les matrices n’est pas comme la multiplication normale. AXB! = B x A *

Maintenant que nous savons comment projeter et ajouter ceci à notre matrice de position, notre code réel ressemblera probablement à ceci:

```
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)
```

Maintenant, notre bogue est écrasé et notre cube semble assez épique au loin. Si vous souhaitez adapter votre cube, la formule est la suivante:

```
 /*!
 *  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;
}
```

Il vous suffit d'ajuster les variables diagonales.

Pour la rotation, vous devez regarder de plus près Quaternions.

Lire Math 3d en ligne: https://riptutorial.com/fr/opengl/topic/4063/math-3d

Chapitre 8: Programme Introspection

Introduction

Un certain nombre de fonctionnalités des objets de programme peuvent être récupérées via l'API du programme.

Examples

Informations d'attribut de sommet

Les informations sur les attributs de sommet peuvent être récupérées avec la fonction OGL glGetProgram et les paramètres `GL_ACTIVE_ATTRIBUTES` et `GL_ACTIVE_ATTRIBUTE_MAX_LENGTH` .

L'emplacement d'un attribut de shader actif peut être déterminé par la fonction OGL glGetAttribLocation , par l'index de l'attribut.

```
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() );
}
```

Informations uniformes

Les informations sur les uniformes actifs dans un programme peuvent être récupérées avec la fonction OGL glGetProgram et les paramètres `GL_ACTIVE_UNIFORMS` et `GL_ACTIVE_UNIFORM_MAX_LENGTH` .

L'emplacement d'une variable uniforme de shader active peut être déterminé par la fonction OGL glGetActiveUniform , par l'index de l'attribut.

```
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() );
}
```

Lire Programme Introspection en ligne: https://riptutorial.com/fr/opengl/topic/10805/programme-introspection

Chapitre 9: Shader Chargement et Compilation

Introduction

Ces exemples montrent différentes manières de charger et de compiler des shaders. Tous les exemples ***doivent inclure un code de gestion des erreurs*** .

Remarques

Les objets Shader, créés à partir de `glCreateShader` ne font pas grand chose. Ils contiennent le code compilé pour une seule étape, mais ils ne doivent même pas contenir le code compilé *complet* pour cette étape. À bien des égards, ils fonctionnent comme des fichiers objets C et C ++.

Les objets du programme contiennent le programme lié final. Mais ils contiennent également l'état des valeurs uniformes du programme, ainsi qu'un certain nombre d'autres données d'état. Ils ont des API pour l'introspection des données d'interface du shader (même si elles ne sont devenues complètes que dans GL 4.3). Les objets de programme sont ce qui définit le code de shader que vous utilisez lors du rendu.

Les objets Shader, une fois utilisés pour lier un programme, ne sont plus nécessaires à moins que vous ne souhaitiez les utiliser pour lier d'autres programmes.

Examples

Charger le shader séparable en C ++

4.1

Ce code charge, compile et lie un fichier unique qui crée un programme de shader distinct pour une seule étape . S'il y a des erreurs, il obtiendra le journal d'information pour ces erreurs.

Le code utilise certaines fonctionnalités C ++ 11 couramment disponibles.

```
#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;
}
```

Compilation d'objet Shader individuel en C ++

Le modèle de compilation GLSL traditionnel consiste à compiler du code pour une scène de shader dans un objet shader, puis à lier plusieurs objets shader (couvrant toutes les étapes que vous souhaitez utiliser) en un seul objet de programme.

Depuis 4.2, il est possible de créer des objets de programme comportant un seul stade de shader. Cette méthode lie toutes les étapes de shader en un seul programme.

Compilation d'objet 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;
}
```

Liaison d'objets de programme

```
#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;
}
```

Lire Shader Chargement et Compilation en ligne:
https://riptutorial.com/fr/opengl/topic/8685/shader-chargement-et-compilation

Chapitre 10: Shaders

Syntaxe

- #version version_number // Quelle version de GLSL nous utilisons
- void main () {/ * Code * /} // Fonction principale du shader
- dans le nom du type; // Spécifie un paramètre d'entrée - GLSL 1.30
- nom du type de sortie; // Spécifie un paramètre de sortie - GLSL 1.30
- nom de type inout; // Paramètre pour l'entrée et la sortie - GLSL 1.30

Paramètres

Paramètre	Détails
type	Le type du paramètre, il doit être un type intégré GLSL.

Remarques

Pour spécifier quelle version de GLSL doit être utilisée pour compiler un shader, utilisez le **préprocesseur de** version, par exemple `#version 330` . Chaque version d'OpenGL est nécessaire pour prendre en charge des versions spécifiques de GLSL . Si un préprocesseur `#version` n'est pas défini en haut d'un shader, la version par défaut 1.10 est utilisée.

Examples

Shader pour le rendu d'un rectangle coloré

Un programme de shader, au sens **OpenGL** , contient un certain nombre de shaders différents. Tout programme de shader doit avoir au moins un **vertex** shader, qui calcule la position des points sur l'écran et un **fragment** shader, qui calcule la couleur de chaque pixel. (En fait, l'histoire est plus longue et plus complexe, mais de toute façon ...)

Les shaders suivants sont pour `#version 110` , mais devraient illustrer quelques points:

Vertex Shader:

```
#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;
}
```

Fragment shader:

```
#version 110

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

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

Lire Shaders en ligne: https://riptutorial.com/fr/opengl/topic/5065/shaders

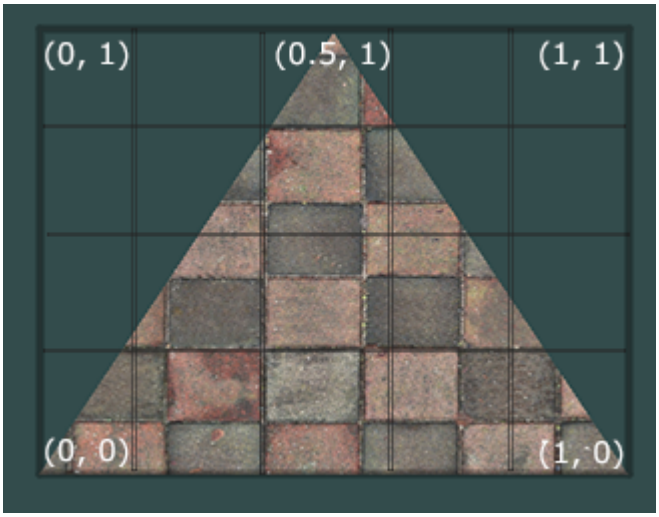
Chapitre 11: Texturation

Examples

Bases de texturation

Une texture est une forme de stockage de données qui permet un accès pratique non seulement à des entrées de données particulières, mais également à des échantillons de points mélangeant (interpolant) plusieurs entrées.

Dans OpenGL, les textures peuvent être utilisées pour beaucoup de choses, mais le plus souvent, elles mappent une image sur un polygone (par exemple un triangle). Afin de mapper la texture sur un triangle (ou un autre polygone), nous devons indiquer à chaque sommet la partie de la texture à laquelle il correspond. Nous assignons une coordonnée de texture à chaque sommet d'un polygone et celle-ci sera ensuite interpolée entre tous les fragments de ce polygone. Les coordonnées de texture sont généralement comprises entre 0 et 1 dans les axes x et y, comme illustré ci-dessous:



Les coordonnées de texture de ce triangle ressembleraient à ceci:

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

Mettez ces coordonnées dans VBO (objet tampon de vertex) et créez un nouvel attribut pour le shader. Vous devriez déjà avoir au moins un attribut pour les positions de sommet, créez-en une autre pour les coordonnées de texture.

Générer de la texture

La première chose à faire sera de générer un objet texture qui sera référencé par un identifiant qui sera stocké dans une *texture* int non signée:

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

Après cela, il doit être lié pour que toutes les commandes de texture suivantes configurent cette texture:

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

Chargement de l'image

Pour charger une image, vous pouvez créer votre propre chargeur d'image ou vous pouvez utiliser une bibliothèque de chargement d'image telle que SOIL (Simple OpenGL Image Library) dans c ++ ou PNGDecoder de TWL dans Java.

Un exemple de chargement d'image avec SOIL serait:

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

Vous pouvez maintenant affecter cette image à l'objet texture:

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

Après cela, vous devez dissocier l'objet texture:

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

Paramètre Wrap pour les coordonnées de texture

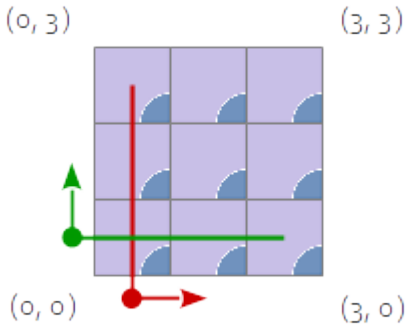
Comme vu ci-dessus, le coin inférieur gauche de la texture a les coordonnées UV (st) (0, 0) et le coin supérieur droit de la texture a les coordonnées (1, 1), mais les coordonnées de texture d'un maillage peuvent être dans toute gamme. Pour gérer cela, il faut définir comment les coordonnées de texture sont enveloppées dans la texture.

Le paramètre wrap de la coordonnée de texture peut être défini avec glTextureParameter à l' aide de `GL_TEXTURE_WRAP_S` , `GL_TEXTURE_WRAP_T` et `GL_TEXTURE_WRAP_R` .

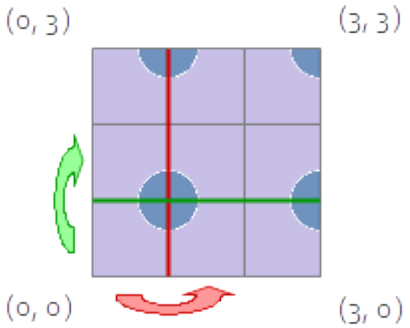
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Les paramètres possibles sont:

- `GL_CLAMP_TO_EDGE` provoque le `GL_CLAMP_TO_EDGE` des coordonnées de la texture dans la plage *[1 / 2N, 1 - 1 / 2N]* , où *N* est la taille de la texture dans la direction.
- `GL_CLAMP_TO_BORDER` fait la même chose que `GL_CLAMP_TO_EDGE` , mais dans les cas où le blocage, les données de texel récupérées sont remplacées par la couleur spécifiée par `GL_TEXTURE_BORDER_COLOR` .
- `GL_REPEAT` fait en sorte que la partie entière de la coordonnée de texture soit ignorée. La texture est en **mosaïque** .



- `GL_MIRRORED_REPEAT` : Si la partie entière de la coordonnée de texture est paire, alors elle est ignorée. Contrairement à, si la partie entière de la coordonnée de texture est impaire, la coordonnée de texture est définie sur *1 - frac (s)* . *fract (s)* est la partie fractionnaire de la coordonnée de texture. Cela fait que la texture est **reflétée** toutes les 2 fois.



- `GL_MIRROR_CLAMP_TO_EDGE` provoque la répétition de la coordonnée textue comme pour `GL_MIRRORED_REPEAT` pour une reptition de la texture, point auquel la coordonnée doit être bloquée comme dans `GL_CLAMP_TO_EDGE` .

Notez que la valeur par défaut pour `GL_TEXTURE_WRAP_S` , `GL_TEXTURE_WRAP_T` et `GL_TEXTURE_WRAP_R` est `GL_REPEAT` .

Appliquer des textures

La dernière chose à faire est de lier la texture avant l'appel de dessin:

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

Texture et tampon

Vous pouvez attacher une image dans une texture à un framebuffer, de sorte que vous pouvez rendre directement à cette texture.

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

Remarque: vous ne pouvez pas lire et écrire à partir de la même texture dans la même tâche de rendu, car elle appelle un comportement non défini. Mais vous pouvez utiliser: `glTextureBarrier()` entre les appels de rendu pour cela.

Lire les données de texture

Vous pouvez lire les données de texture avec la fonction `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);
```

Remarque: le type de texture et le format ne sont que par exemple et peuvent être différents.

Utiliser les PBO

Si vous liez un tampon à `GL_PIXEL_UNPACK_BUFFER` le paramètre `data` de `glTexImage2D` est un décalage dans ce tampon.

Cela signifie que glTexImage2D n'a pas besoin d'attendre que toutes les données soient copiées hors de la mémoire de l'application avant de pouvoir revenir, réduisant ainsi la surcharge dans le thread 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);
```

Mais où les PBO brillent vraiment, c'est lorsque vous devez lire le résultat d'un rendu dans la mémoire de l'application. Pour lire les données de pixel dans un tampon, liez-le à `GL_PIXEL_PACK_BUFFER` puis le paramètre data de `glGetTexImage` sera un décalage dans ce tampon:

```
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);

}
```

Utiliser des textures dans les shaders GLSL

Le vertex shader n'accepte que les coordonnées de texture en tant qu'attribut de sommet et transmet les coordonnées au fragment shader. Par défaut, cela garantira également que le fragment recevra la coordonnée correctement interpolée en fonction de sa position dans un triangle:

```
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;
```

```
}
```

Le fragment shader accepte alors la variable de sortie `texCoord` tant que variable d'entrée. Vous pouvez ensuite ajouter une texture au fragment shader en déclarant un `uniform sampler2D` . Pour échantillonner un fragment de la texture, nous utilisons une `texture` fonction intégrée dotée de deux paramètres. Le premier est la texture dont nous voulons échantillonner et le second est la coordonnée de cette texture:

```
in vec2 texCoordOut;

out vec4 color;

uniform sampler2D image;

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

Notez que l' `image` n'est pas l'id de texture directe ici. C'est l'id de l' *unité de texture* qui sera échantillonnée. À leur tour, les textures ne sont pas liées directement aux programmes; ils sont liés aux unités de texture. Ceci est obtenu en rendant d'abord l'unité de texture active avec `glActiveTexture` , puis en appelant `glBindTexture` affectera cette unité de texture particulière. Cependant, comme l'unité de texture par défaut est l'unité de texture `0` , les programmes utilisant une texture peuvent être simplifiés en omettant cet appel.

Lire Texturation en ligne: https://riptutorial.com/fr/opengl/topic/3461/texturation

Chapitre 12: Utiliser des VAO

Introduction

L'objet Vertex Array stocke comment opengl doit interpréter un ensemble de VBO.

Essentiellement, cela vous évitera d'appeler glVertexAttribPointer à chaque fois que vous voulez rendre un nouveau maillage.

Si vous ne voulez pas traiter avec les VAO, vous pouvez simplement en créer un et le lier pendant l'initialisation du programme et prétendre qu'ils n'existent pas.

Syntaxe

- annuler glEnableVertexAttribArray (GLuint attribIndex);
- void glDisableVertexAttribArray (GLuint attribIndex);
- void glVertexAttribPointer (GLIINT attribIndex, GLint size, type GLenum, GLboolean normalisé, GLsizei stride, const GLvoid * pointeur);
- vol glVertexAttribFormat (GLIINT attribIndex, taille GLint, type GLenum, GLboolean normalisé, GLuint relativeoffset);
- void glVertexAttribBinding (GLuint attribIndex, GLuint bindingIndex);
- annuler glBindVertexBuffer (GLuint bindingIndex, tampon GLuint, décalage GLintptr, foulée GLintptr);

Paramètres

paramètre	Détails
attribIndex	l'emplacement de l'attribut de sommet auquel le tableau de verte alimentera les données
Taille	le nombre de composants à extraire de l'attribut
type	Le type C ++ des données d'attribut dans le tampon
normalisé	s'il faut mapper des types entiers à la plage à virgule flottante [0, 1] (pour unsigned) ou [-1, 1] (for signed)
aiguille	le décalage d'octet dans le tampon au premier octet des données de l'attribut (converti en `void*` pour des raisons héritées)
décalage	le décalage d'octet de base depuis le début du tampon vers lequel les

paramètre	Détails
	données du tableau commencent
relativeOffset	le décalage à un attribut particulier, par rapport au décalage de base du tampon
foulée	le nombre d'octets des données d'un sommet vers le suivant
tampon	l'objet tampon où sont stockés les tableaux de sommets
bindingIndex	l'index auquel l'objet tampon source sera lié

Remarques

La configuration VAO de format d'attribut distinct peut interopérer avec `glVertexAttribPointer` (ce dernier est défini en termes de l'ancien). Mais vous devez faire attention en le faisant.

La version séparée du format d'attribut a des équivalents d'accès direct à l'état (DSA) dans 4.5. Ceux-ci auront les mêmes paramètres mais au lieu d'utiliser le VAO lié, le VAO en cours de modification est passé explicitement. Lors de l'utilisation de DSA, le tampon d'index pour `glDrawElements` peut être défini avec `glVertexArrayElementBuffer(vao, ebo);`

Examples

Version 3.0

Chaque attribut est associé à un nombre de composants, à un type, à une normalisation, à un décalage, à une foulée et à un VBO. Le VBO n'est pas passé explicitement en paramètre mais est la mémoire tampon liée à GL_ARRAY_BUFFER au moment de l'appel.

```
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);
    }
```

```
}
```

Version 4.3

4.3

OpenGL 4.3 (ou ARB_separate_attrib_format) ajoute une autre manière de spécifier les données de sommet, ce qui crée une séparation entre le format des données liées à un attribut et la source de l'objet tampon qui fournit les données. Donc, au lieu d'avoir un VAO par maillage, vous pouvez avoir un format VAO par sommet.

Chaque attribut est associé à un format de sommet et à un point de liaison. Le format de sommet se compose du type, du nombre de composants, de sa normalisation et du décalage relatif entre le début des données et ce sommet particulier. Le point de liaison spécifie le tampon à partir duquel un attribut prend ses données. En séparant les deux, vous pouvez lier des tampons sans spécifier de format de sommet. Vous pouvez également modifier le tampon qui fournit des données à plusieurs attributs avec un seul appel de liaison.

```
//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);
```

Ensuite, pendant le tirage au sort, vous maintenez le vao lié et ne modifiez que les liaisons de tampon.

```
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
    }
}
```

Lire Utiliser des VAO en ligne: https://riptutorial.com/fr/opengl/topic/9237/utiliser-des-vao

Chapitre 13: Vue et projection OGL

Introduction

A propos de la matrice du modèle, de la matrice de vue, de la projection orthographique et en perspective

Examples

Implémenter une caméra dans OGL 4.0 GLSL 400

Si nous voulons regarder une scène comme si nous l'avions photographiée avec un appareil photo, nous devons d'abord définir certaines choses:

- La position à partir de laquelle la scène est vue, la position des yeux `pos` .
- Le point que nous regardons dans la scène (`target`). Il est également courant de définir la direction dans laquelle nous regardons. Techniquement, nous avons besoin d'une *ligne de vue* . Une droite dans l'espace est définie mathématiquement soit par 2 points, soit par un point et un vecteur. La première partie de la définition est la position de l'œil et la seconde est la `target` ou la ligne de visée vectorielle `los` .
- La direction vers le `up` .
- Le champ de vision ' `fov_y` . Cela signifie que l'angle entre les deux lignes droites commence à la position de l'œil et se termine au point le plus à gauche et au point le plus à droite, ce qui peut être vu simultanément.
- Le grand et le format de la fenêtre dans laquelle nous projetons notre image `vp` .
- L' *avion* `near` et le *plan* `far` . Le *plan proche* est la distance entre la position de l'œil et le plan d'où les objets deviennent visibles pour nous. Le *plan lointain* est la distance entre la position de l'œil et le plan sur lequel les objets de la scène sont visibles. Une explication de ce que l' *avion proche* et l' *avion lointain* doivent suivre suivront plus tard.

Une définition de ces données en C ++ et en Python peut ressembler à ceci:

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};
};
```

Python

```
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
```

Afin de prendre en compte toutes ces informations lors du dessin d'une scène, une matrice de projection et une matrice de vue sont généralement utilisées. Afin d'organiser les différentes parties d'une scène dans la scène, des matrices de modèle sont utilisées. Cependant, ceux-ci ne sont mentionnés ici que par souci d'exhaustivité et ne seront pas traités ici.

- Matrice de projection: La matrice de projection décrit la correspondance entre les points 3D du monde tels qu'ils sont vus depuis une caméra à sténopé et les points 2D de la fenêtre.
- Afficher la matrice: La matrice de vue définit la position de l' *œil* et la direction de visualisation sur la scène.
- Matrice de modèle: La matrice de modèle définit l'emplacement et la taille relative d'un objet dans la scène.

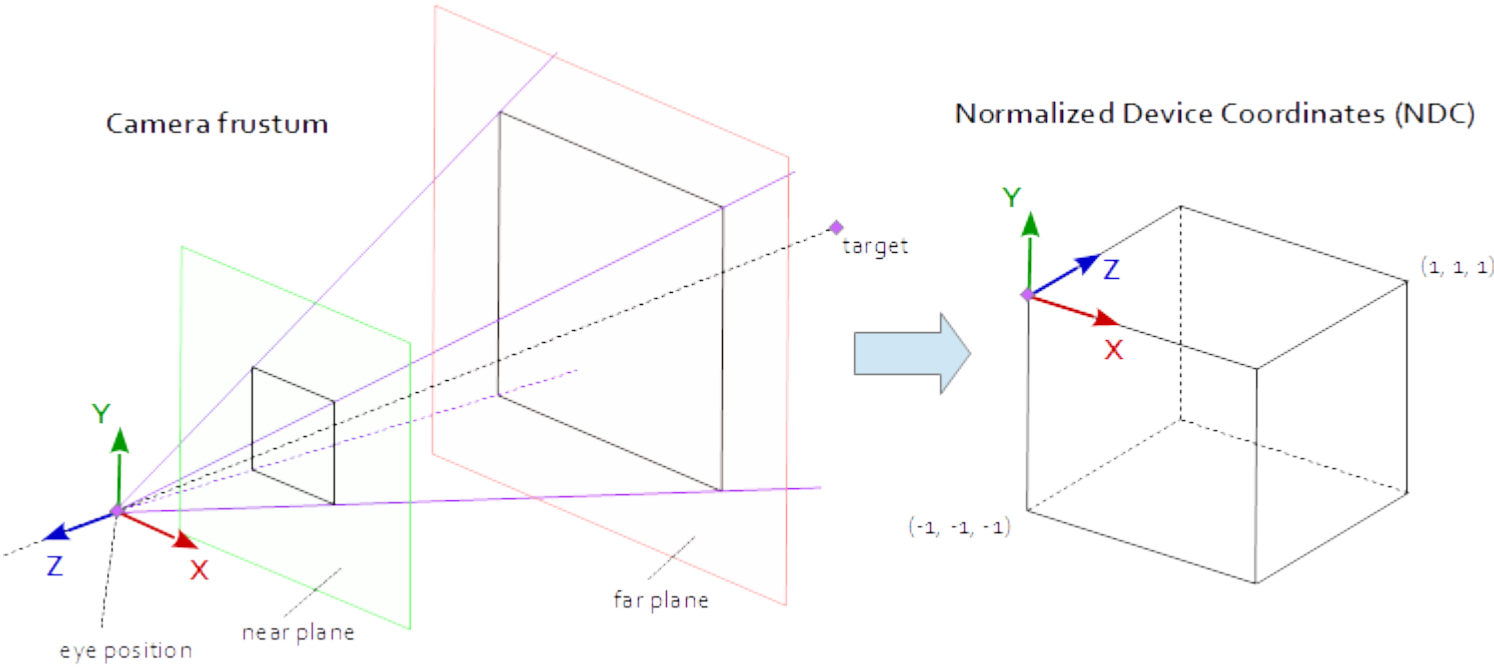
Après avoir rempli les structures de données ci-dessus avec les données correspondantes, nous devons les traduire dans les matrices appropriées. Dans le mode de compatibilité OGL, cela peut être fait avec les fonctions `gluLookAt` et `gluPerspective` qui définissent les uniformes `gl_ModelViewMatrix` , `gl_NormalMatrix` et `gl_ModelViewProjectionMatrix` . Dans OGL 3.1 et GLSL #version 150, les uniformes intégrés ont été supprimés car toute la pile de matrice à fonction fixe est devenue obsolète. Si nous voulons utiliser le shader de haut niveau OGL avec GLSL version 330 ou plus, nous devons définir et définir la matrice uniformément notre propre (à part l'utilisation du mot clé de `compatibility` GLSL).

Mettre en place la perspective - matrice de projection

Un point de la fenêtre d'affichage est visible lorsqu'il se trouve dans le bloc AABB natif défini par les points `(-1.0, -1.0, -1.0)` et `(1.0, 1.0, 1.0)` . C'est ce qu'on appelle les coordonnées de périphérique normalisées (NDC). Un point avec les coordonnées `(-1.0, -1.0, z)` sera peint dans le coin inférieur gauche de la fenêtre et un point avec les coordonnées `(1.0, 1.0, z)` sera peint dans le coin supérieur droit de la fenêtre. La coordonnée Z est mappée de l'intervalle (-1.0, 1.0) à l'intervalle (0.0, 1.0) et écrite dans le Z-buffer.

Tout ce que nous pouvons voir de la scène se trouve dans une pyramide à 4 faces. Le sommet de la pyramide est la *position de* l' *œil* . Les 4 côtés de la pyramide sont définis par le champ de vision (`fov_y`) et le format (`vp[0]/vp[1]`). La matrice de projection doit mapper les points de l'intérieur de la pyramide au NDC défini par les points `(-1.0, -1.0, -1.0)` et `(1.0, 1.0, 1.0)` . À ce stade, notre pyramide est infinie, elle n'a pas de fin en profondeur et nous ne pouvons pas mapper un espace infini à un espace fini. Pour cela, nous avons maintenant besoin du *plan proche* et du *plan lointain* , ils transforment la pyramide en un tronc en coupant le sommet et en limitant la

pyramide dans la profondeur. Le plan proche et le plan lointain doivent être choisis de manière à inclure tout ce qui doit être visible de la scène.



La correspondance entre les points d'un tronc et le NDC est purement mathématique et peut être généralement résolue. Le développement des formules a souvent été discuté et publié à plusieurs reprises sur le Web. Comme vous ne pouvez pas insérer une formule LaTeX dans une documentation Stack Overflow, cela n'est pas le cas ici et seul le code source C ++ et Python complété est ajouté. Notez que les coordonnées oculaires sont définies dans le système de coordonnées droitier, mais que NDC utilise le système de coordonnées gaucher. La matrice de projection est calculée à partir du *champ de vision* `fov_y` , du *ratio d'aspect* `vp[0]/vp[1]` , du *plan* `near` et du *plan* `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 } };
}
```

Python

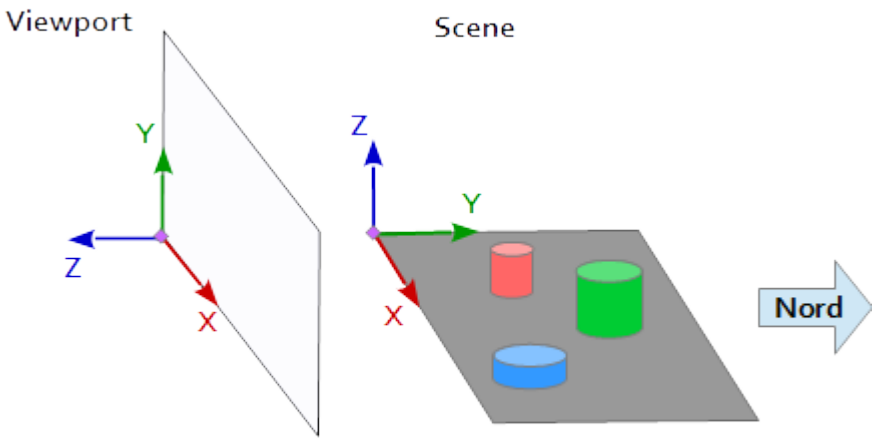
```
def Perspective(self):
    fn, = self.far + self.near
    f_n = self.far - self.near
```

```
    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 ] ] )
```

Mettre en place le regard sur la scène - Afficher la matrice

Dans le système de coordonnées de la fenêtre, l'axe Y est orienté vers le haut `(0, 1, 0)` et l'axe X est orienté vers la droite `(1, 0, 0)` . Il en résulte un axe Z qui pointe hors de la fenêtre (`(0, 0, -1) = cross( X-axis, Y-axis )`).

Dans la scène, l'axe X pointe vers l'est, l'axe Y vers le nord et l'axe Z vers le haut.



L'axe X de la fenêtre `(1, 0, 0)` correspond à l'axe Y de la scène `(1, 0, 0)` , l'axe Y de la fenêtre `(0, 1, 0 )` correspond à l'axe Z de la scène `(0, 0, 1)` et l'axe Z de la fenêtre `(0, 0, 1 )` correspond à l'axe Y négatif de la scène `(0, -1, 0)` .

Chaque point et chaque vecteur du système de référence de la scène doivent donc être convertis d'abord en coordonnées de fenêtre. Cela peut être fait par des opérations de permutation et d'inversion dans les vecteurs scalaires.

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

Pour configurer une matrice de vues, la position `pos` , la cible `target` et le vecteur `up` doivent être mappés dans le système de coordonnées de la fenêtre, comme décrit ci-dessus. Cela donne les 2 points `p` et `t` et le vecteur `u` , comme dans l'extrait de code suivant. L'axe Z de la matrice de vue est la ligne de visée inverse, calculée par `p - t` . L'axe Y est le vecteur ascendant `u` . L'axe X est calculé par le produit croisé de l'axe Y et de l'axe Z. Pour orthonormaliser la matrice de vue, le

produit croisé est utilisé une seconde fois pour calculer l'axe Y à partir de l'axe Z et de l'axe X (bien sûr, l'orthogonalisation de Gram-Schmidt fonctionnerait aussi bien). A la fin, tous les 3 essieux doivent être normalisées et la *position des yeux* `pos` doit être défini comme e origine de la matrice de vue.

Le code ci-dessous définit une matrice qui encapsule exactement les étapes nécessaires pour calculer un regard sur la scène:

1. Conversion des coordonnées du modèle en coordonnées de fenêtre.
2. Tournez dans le sens de la direction de vue.
3. Mouvement vers la position de l'oeil

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;
}
```

python

```
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] ] )
```

Les matrices sont finalement écrites en uniformes et utilisées dans le vertex shader pour transformer les positions du modèle.

Vertex Shader

Dans le vertex shader, une transformation après l'autre est effectuée.

1. La matrice modèle amène l'objet (maillage) à sa place dans la scène. (Ceci n'est indiqué que par souci d'exhaustivité et n'a pas été documenté ici car il n'a rien à voir avec la vue de la scène)
2. La matrice de vue définit la direction à partir de laquelle la scène est vue. La transformation avec la matrice de vues fait pivoter les objets de la scène de manière à ce qu'ils soient visualisés dans la direction de vue souhaitée par rapport au système de coordonnées de la fenêtre d'affichage.
3. La matrice de projection transforme les objets d'une vue parallèle en une vue en perspective.

```
#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;
}
```

Fragment shader

Le fragment shader est répertorié ici uniquement pour des raisons d'exhaustivité. Le travail a été fait avant.

```
#version 400

in vec3 vertCol;

out vec4 fragColor;

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

Une fois le shader compilé et aimé, les matrices peuvent être liées aux 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() );
```

Python

```
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() )
```

De plus, j'ai ajouté l'intégralité du code dump d'un exemple Python (ajouter l'exemple C ++ dépasserait malheureusement la limite de 30000 caractères). Dans cet exemple, la caméra se déplace elliptiquement autour d'un tétraèdre à un point focal de l'ellipse. La direction de visionnement est toujours dirigée vers le tetraeder.

Python

Pour exécuter le script Python, NumPy doit être installé.

```
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()
```

Lire Vue et projection OGL en ligne: https://riptutorial.com/fr/opengl/topic/10680/vue-et-projection-ogl

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec opengl	ammar26, Ashwin Gupta, Bogdan0804, CaseyB, Community, datenwolf, Franko L. Tokalić, genpfault, L-X, Naman Dixit, Nicol Bolas, Nox, Valvy
2	Création de contexte OpenGL.	Dynamitos
3	Éclairage de base	Ashwin Gupta, Elouarn Laine, Rabbid76
4	Encapsulation d'objets OpenGL avec C ++ RAII	0x499602D2, Nicol Bolas
5	Framebuffers	MarGenDo, Rabbid76
6	L'instanciation	MarGenDo, Nicol Bolas
7	Math 3d	Ashwin Gupta, Franko L. Tokalić, Valvy, Wyck
8	Programme Introspection	Nicol Bolas, Rabbid76
9	Shader Chargement et Compilation	Nicol Bolas, Rabbid76
10	Shaders	FedeWar, genpfault, George, MarGenDo, nica.dan.cs, Nicol Bolas
11	Texturation	Bartek Banachewicz, genpfault, George, MarGenDo, Nicol Bolas, Rabbid76, ratchet freak
12	Utiliser des VAO	Nicol Bolas, ratchet freak
13	Vue et projection OGL	Matej Kormuth, Rabbid76, SurvivalMachine