

Introduction



par Jeff Molofee (NeHe) ([Autres Articles](#)) Traduit par Jean Christophe Beyler

Date de publication : 23/03/2007

Dernière mise à jour : 23/03/2007

Bienvenue aux tutoriels OpenGL. Je suis un mec ordinaire avec une passion pour OpenGL! La première fois que j'ai entendu parlé d'OpenGL était lorsque 3Dfx avait fourni leur pilote OpenGL pour la Voodoo 1. J'ai tout de suite su que je devais apprendre à programmer en OpenGL. Malheureusement, c'était très difficile de trouver des informations à propos d'OpenGL dans les livres ou sur internet. J'ai passé des heures tentant de faire fonctionner du code et encore plus de temps à demander de l'aide par mail et par IRC. J'ai découvert que les personnes qui comprenaient les entrailles d'OpenGL se considéraient comme des élites et n'avaient aucune volonté de partager leur connaissance. C'était très frustrant!

- 0 - Contributions
- 1 - Tutoriel
 - 1.1 - Introduction
 - 1.2 - Inclusions et variables globales
 - 1.3 - La fonction ReSizeGLScene
 - 1.4 - La fonction InitGLScene
 - 1.5 - La fonction DrawGLScene
 - 1.6 - La fonction KillGLScene
 - 1.7 - La fonction CreateGLWindow
 - 1.8 - La fonction WndProc
 - 1.9 - La fonction WinMain
 - 1.10 - Conclusion
- 2 - Téléchargements
- 3 - Remerciements
- 4 - Liens

0 - Contributions

Remarque : la traduction des autres tutoriels est en cours mais votre aide serait appréciée ! Si vous voulez aider, n'hésitez pas à envoyer un **message privé** ou un mail vers **nehe arrobase redaction-developpez point com**.

1 - Tutoriel

J'ai créé cette série de tutoriels pour que les personnes intéressées dans l'apprentissage de l'OpenGL aient quelque chose à se mettre sous la dent. Dans chacun de mes tutoriels, je tente d'expliquer, avec autant de détails que possible, ce que chaque ligne de code fait. Je tente de garder le code simple (pas de code MFC à apprendre). Même un vrai débutant en C++ et en OpenGL devrait pouvoir regarder le code présenté et avoir une idée assez précise de ce qui s'y passe. Si vous êtes déjà un bon programmeur OpenGL, ces tutoriels n'auront probablement pas grande chose à vous apprendre mais, si vous débutez, alors je pense que ces tutoriels auront beaucoup à offrir.

Ce tutoriel a été entièrement réécrit en Janvier 2000. Ce tutoriel va vous apprendre à mettre en place une fenêtre OpenGL. La fenêtre peut être mise en plein écran ou prendre la taille, la résolution et le format pixel que vous souhaitez. Le code est très flexible et peut être utilisé pour tous vos projets OpenGL. Tous les liens sont basés sur celui-ci ! J'ai écrit ce code pour qu'il soit flexible et puissant à la fois. Toutes les erreurs ont été, en principe, corrigées. Il ne devrait pas avoir de fuites de mémoire et le code est facile à lire et à modifier. Merci à Fredric Echols pour ses modifications du code.

1.1 - Introduction

Je vais commencer ce tutoriel en montrant directement le code. La première chose que vous devez faire est créer un projet sous Visual C++. Si vous ne savez pas comment faire cela, vous ne devriez pas tenter d'apprendre OpenGL mais plutôt le fonctionnement de Visual C++ (ou votre EDI préféré). Certaines versions de VC++ ont besoin que le type *bool* soit changé en **BOOL**, *true* doit être changé en **TRUE** et *false* doit être changé en **FALSE**. En faisant cela, j'ai même réussi à compiler le code sous Visual C++ 4.0 et 5.0 sans aucun problème.

Après avoir créé une nouvelle application Win32 - pas une application console - dans Visual C++, you devez lier les bibliothèques OpenGL. Dans Visual C++, allez à Projet->Propriétés->Propriétés de configuration->Editeur des liens. Dans Entrées/Bibliothèques, ajoutez OpenGL32.lib GLu32.lib and GLaux.lib. Une fois effectué, appuyez sur OK. Vous pouvez maintenant écrire un programme Windows OpenGL.

Remarque #1 : Beaucoup de compilateurs ne définissent pas **CDS_FULLSCREEN**. Si vous recevez un message d'erreur qui parle de **CDS_FULLSCREEN**, vous devez ajouter cette ligne de code au début de votre programme : `#define CDS_FULLSCREEN 4`.

Remarque #2 : Lorsque ces premiers tutoriels ont été écrits, **GLAUX** était très utilisé. Mais, au fil du temps, **GLAUX** n'a plus été maintenu par les développeurs. Beaucoup de ces tutoriels utilisent encore **GLAUX**. Si votre compilateur ne supporte pas **GLAUX** ou que vous ne voulez pas l'utiliser, télécharger le code de remplacement à partir de la page d'accueil <http://nehe.gamedev.net>.

1.2 - Inclusions et variables globales

Les premières 4 lignes incluent les fichiers d'en-tête pour chaque bibliothèque dont on aura besoin. Voici ces lignes :

Inclusions

```
#include <windows.h> // Fichier d'entete pour Windows
#include <gl\gl.h> // Fichier d'entete pour OpenGL32
#include <gl\glu.h> // Fichier d'entete pour GLu32
#include <gl\glaux.h> // Fichier d'entete pour GLaux
```

Ensuite, vous aurez besoin de mettre en place toutes les variables que vous allez utiliser dans votre programme. Ce programme va créer une fenêtre OpenGL vide, donc nous n'aurons pas encore besoin de beaucoup de variables.

Les quelques variables que nous allons utiliser ici sont très importantes et vous allez les utiliser dans presque tous les programmes OpenGL que vous écrirez.

La première ligne met en place un contexte de rendu. Tout programme OpenGL est lié à un tel contexte. Un contexte de rendu (RC) permet de lier les appels OpenGL au contexte de fenêtre. Le contexte de rendu OpenGL est défini comme étant hRC. Pour que votre programme puisse dessiner dans une fenêtre, vous devrez créer un contexte de fenêtre, ceci est fait dans la seconde ligne. Le contexte de fenêtre (DC pour *Device Context*) est défini comme étant HDC. Le DC connecte la fenêtre vers le GDI (L'interface graphique matériel). Le RC connecte OpenGL vers le DC.

Dans la troisième ligne, la variable **hWnd** va contenir l'identifiant assigné à notre fenêtre par le système d'exploitation Windows. Finalement, la quatrième ligne crée une instance pour notre programme.

Variables globales

```
HGLRC hRC=NULL; // Contexte de rendu permanent
HDC hDC=NULL; // Contexte GDI
HWND hWnd=NULL; // Contient un identifiant de fenêtre
HINSTANCE hInstance; // Contient une instance de l'application
```

La première ligne de la section du code qui suit met en place un tableau que nous allons utiliser pour surveiller l'état des touches de clavier. Il y a beaucoup de solutions pour gérer le clavier, mais c'est cette solution que j'utilise. C'est fiable et cela peut gérer l'appui de plusieurs touches en même temps.

La variable **active** sera utilisée pour savoir si notre fenêtre a été minimisée. Si la fenêtre a été minimisée, nous pouvons, par exemple, suspendre l'exécution du programme ou même quitter le programme. Je préfère suspendre le programme pour qu'il n'utilise pas les ressources processeurs lorsqu'il est minimisé.

La variable **fullscreen** est assez explicite. Si le programme est en plein écran, la variable vaudra TRUE, sinon le programme est en mode fenêtre et donc **fullscreen** vaudra FALSE. C'est important de définir cette variable en global pour que chaque fonction puisse savoir si le programme est en plein écran.

Autres variables globales

```
bool keys[256]; // Tableau utilisé pour la gestion du clavier
bool active=TRUE; // Fenêtre active mis à TRUE par défaut
bool fullscreen=TRUE; // Variable pour le plein écran mis à TRUE par défaut
```

Nous allons maintenant déclarer la fonction **WndProc**. Nous définissons ce prototype parce que la fonction **CreateGLWindow** fait un appel à **WndProc** mais elle est déclarée avant. En C, si nous voulons faire un appel à une fonction qui est définie après, nous devons définir le prototype au début du programme. Donc, dans le code qui suit, on définit le prototype de **WndProc**.

Prototype de WndProc

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration de WndProc
```

1.3 - La fonction ReSizeGLScene

Le travail de la prochaine section de code est de remettre la scène OpenGL lorsque la fenêtre est redimensionnée (en supposant que vous êtes en mode fenêtre et non plein écran). Même si vous ne pouvez pas redimensionner la fenêtre (par exemple, si le programme est en plein écran), cette fonction sera tout de même appelée lorsque le programme commence pour mettre en place la perspective. La scène OpenGL sera redimensionnée par rapport à la taille de la fenêtre qui la contient.

Fonction ReSizeGLScene

```
/* Redimensionne et initialise la fenêtre GL */
```

Fonction ReSizeGLScene

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
{
    if(height==0) /* Evite une division par zero */
    {
        height=1; /* Transforme la hauteur à 1 */
    }
    glViewport(0, 0, width, height); // Remet le Viewport à jour
}
```

Les lignes qui initialisent la perspective de la fenêtre. Ceci veut dire que plus les choses sont loin, plus elles deviendront petites. Ceci crée une scène plus réaliste. La perspective est calculée avec un angle de 45 degrés basé sur la largeur et la hauteur de la fenêtre. Le 0.1f et 100.0f sont la profondeur la plus proche et la plus lointaine qu'on peut dessiner dans la scène.

glMatrixMode(GL_PROJECTION) indique que le code qui suit va affecter la matrice de projection. Cette matrice est responsable de la gestion de la perspective de la scène. **glLoadIdentity()** est similaire à une remise à zéro. Cet appel remet la matrice à son état de départ. Après **glLoadIdentity()** a été appelé, on met la perspective de la scène en place. **glMatrixMode(GL_MODELVIEW)** indique que les prochaines transformations vont affecter la matrice de modélisation. Cette matrice est l'endroit où on met l'information pour les objets. Enfin, on remet la matrice de modélisation à zéro. Ne vous inquiétez pas si vous ne comprenez pas tout cela, ce sera de nouveau expliqué dans les prochains tutoriels. Sachez seulement que ce bout de code doit être fait si vous voulez avoir une jolie perspective.

Mise en place de la perspective

```
glMatrixMode(GL_PROJECTION); // Choisir la matrice de projection
glLoadIdentity(); // Remettre a zero de la matrice de projection

// Calculer le ratio pour la perspective de la fenêtre
gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, 0.1f, 100.0f);
glMatrixMode(GL_MODELVIEW); // Choisir la matrice de modélisation
glLoadIdentity(); // Remettre a zero la matrice de modélisation
}
```

1.4 - La fonction InitGLScene

Dans la prochaine section de code, nous initialisons OpenGL. Nous définissons la couleur qui sera utilisée pour vider la scène, nous mettons en place un tampon pour la profondeur, mettre en place un joli lissage, etc. Cette fonction ne sera pas appelée tant que la fenêtre OpenGL n'a pas été créée. Cette fonction retourne un entier mais, puisque ce code ne risque pas encore grand chose, nous n'allons pas nous en soucier.

Fonction d'initialisation

```
int InitGL(GLvoid) // Toute l'initialisation de OpenGL se trouve ici
{
```

La prochaine ligne permet d'avoir un joli rendu. Ce rendu permet d'avoir un bon dégradé. J'expliquerai comment obtenir un bon dégradé dans un autre tutoriel.

Utilisation de glShadeModel

```
glShadeModel(GL_SMOOTH); // Permet un joli ombrage
```

La ligne qui suit permet de choisir la couleur de fond lorsqu'on l'efface. Si vous ne savez pas comment fonctionne le système de couleur, je vais l'expliquer rapidement. L'intervalle des couleurs va de 0.0f à 1.0f. 0.0f étant le plus foncé et 1.0f étant le plus clair. Le premier paramètre après **glClearColor** est l'intensité de la couleur rouge, le second est l'intensité de la couleur verte et le troisième est l'intensité pour la couleur bleue. Si la valeur est proche de 1.0f, la couleur sera plus intense pour cette couleur. Le dernier nombre est la valeur Alpha. Lorsque nous nous occupons

d'effacer l'écran, nous ne nous intéressons pas de cette dernière valeur. Laissons cette valeur à 0.0f. J'expliquerai son utilité dans un autre tutoriel.

On crée différentes couleurs en mélangeant les trois couleurs primaires (rouge, vert, bleu). Donc, si vous appelez **glClearColor(0.0f,0.0f,1.0f,0.0f)**, vous allez effacer l'écran avec une couleur bleue claire. Si vous appelez **glClearColor(0.5f,0.0f,0.0f,0.0f)**, vous effacerez l'écran vers un rouge moyen. Pour avoir un fond blanc, il suffit de mettre toutes les couleurs à 1.0f. Pour avoir un fond noir, les trois couleurs doivent être mises à 0.0f.

Couleur de fond

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Fond noir
```

Les trois prochaines lignes font intervenir le tampon de profondeur. Considérez le tampon de profondeur comme étant des niveaux de couches sur l'écran. Ce tampon garde les profondeurs des objets qui sont affichés. Nous n'allons pas utiliser le tampon de profondeur dans ce programme, mais presque chaque programme OpenGL qui fait un rendu 3D utilise le tampon de profondeur. Ce tampon permet de dessiner un objet seulement s'il est devant un autre objet. Ce tampon est très important pour le rendu OpenGL.

Mise en place du test de profondeur

```
glClearDepth(1.0f); // Mis en place du tampon de profondeur  
glEnable(GL_DEPTH_TEST); // Mis en place du test de profondeur  
glDepthFunc(GL_LEQUAL); // Le type de test de profondeur
```

Ensuite, on explique à OpenGL que nous voulons la meilleure perspective qui peut être utilisé. Ceci provoque un coût de performance, mais nous allons améliorer un peu mieux la vue en perspective.

Mis en place des calculs de perspective

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Très jolis calculs de perspective
```

Finalement on retourne TRUE. Si nous voulons savoir si l'initialisation a eu des problèmes, on pourra tester le retour de cette fonction. Vous pourrez ajouter votre code et retourner FALSE si une erreur est survenue. Pour le moment, nous n'avons pas besoin de nous y intéresser.

La fin de la fonction

```
return TRUE; // L'initialisation s'est bien passé  
}
```

1.5 - La fonction DrawGLScene

C'est dans cette section que nous allons mettre tout le code de rendu. Tout ce que vous voulez afficher dans la fenêtre ira dans cette section de code. Chaque tutoriel qui va suivre celui-ci va ajouter du code dans cette section du programme. Si vous avez déjà une compréhension d'OpenGL, vous pouvez tenter de créer des formes de base après l'appel **glLoadIdentity()** et avant de retourner TRUE. Si vous êtes un(e) débutant(e) en programmation OpenGL, regardez les tutoriels qui vont suivre. Pour le moment, tout ce que nous allons faire, c'est mettre la couleur de base en place, réinitialiser le tampon de profondeur et réinitialiser la scène. Nous ne dessinerons rien pour le moment.

Le retour de TRUE dit à notre programme qu'il n'y pas eu de problèmes. Si vous voulez que le programme s'arrête pour une raison ou une autre, ajouter un *return FALSE* dans le code pour dire au programme que le code de rendu a échoué.

La fonction DrawGLScene

```
int DrawGLScene(GLvoid) // C'est ici qu'on fait tout le dessin  
{  
    // Effacer l'écran et le tampon de profondeur
```

La fonction DrawGLScene

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
glLoadIdentity(); // Remettre à zéro la matrice de modélisation  
return TRUE; // Tout s'est bien passé  
}
```

1.6 - La fonction KillGLScene

La prochaine section de code est appelée juste avant que le programme s'achève. Le travail de la fonction **KillGLWindow()** est de relâcher le contexte de rendu, le contexte de fenêtre et l'identifiant de fenêtre. On a ajouté beaucoup de vérifications d'erreurs. Si le programme est incapable de détruire n'importe quelle partie de la fenêtre, une boîte de message va apparaître, expliquant ce qui va échouer. Ceci va rendre le débogage de votre code beaucoup plus facile.

La fonction KillGLWindow

```
GLvoid KillGLWindow(GLvoid) // Fermer la fenêtre proprement  
{
```

La première chose que nous allons faire dans **KillGLWindow()** est de vérifier si nous sommes en plein écran. Si c'est le cas, nous allons retourner vers le bureau. Nous devons détruire la fenêtre avant d'enlever le mode plein écran car, si nous détruisons la fenêtre AVANT d'enlever le mode plein écran avec certaines cartes vidéos, le bureau va devenir corrompu. Donc nous devons enlever ce mode plein écran d'abord. Ceci empêchera le bureau d'être corrompu et fonctionne bien avec les cartes Nvidia et 3dfx!

Code de plein écran

```
if(fullscreen) // Sommes-nous en mode plein-écran?  
{
```

Nous utilisons `ChangeDisplaySettings(NULL,0)` pour nous remettre la configuration du bureau. Passer NULL comme premier paramètre et 0 comme le second paramètre force Windows à utiliser les valeurs qui seront stockées dans un registre Windows (la résolution par défaut, le nombre de bits de profondeur, la fréquence, etc). Ceci remettra les paramètres du bureau en place. Ensuite, nous pourrons de nouveau rendre le curseur visible.

Code pour le plein écran

```
ChangeDisplaySettings(NULL,0); // Pour remettre le bureau en place  
ShowCursor(TRUE); // Remettre la visibilité de la souris  
}
```

Le code ci-dessous vérifie si nous avons un contexte de rendu (hRC). Si ce n'est pas le cas, le programme va sauter cette section de code et vérifier si nous avons un contexte de fenêtre.

Test sur le contexte de rendu

```
if(hRC) // Est-ce que nous avons un contexte rendu?  
{
```

Si nous avons un contexte de rendu, le code qui suit va vérifier si nous sommes capables de le libérer (détaché le hRC du hDC). Remarquez comment je vérifie pour les erreurs. Je suis en train de dire au programme de tenter de le libérer (avec `wglMakeCurrent(NULL,NULL)`), ensuite je vérifie si la libération a été une réussite ou non.

Libération du DC et du RC

```
if(!wglMakeCurrent(NULL,NULL)) // Sommes-nous capables de libérer le DC et le RC?  
{
```

Si nous sommes incapables de libérer les deux contextes DC et RC, une boîte de message va apparaître un message d'erreur pour nous laisser savoir que les contextes n'ont pas été libérés. L'utilisation de NULL veut dire que la boîte de dialogue n'a pas de fenêtre parent. La chaîne de caractères qui suit sera le texte qui apparaît dans la boîte de dialogue. "SHUTDOWN ERROR" est le texte qui apparaît dans le titre de la boîte de dialogue. Ensuite nous avons MB_OK, cela veut dire que nous voulons une boîte de dialogue avec un bouton appelé "OK". MB_ICONINFORMATION fait apparaître un petit icône dans la boîte (ce qui le fait ressortir un peu plus).

Affichage d'une boîte de dialogue

```
MessageBox(NULL, "Release Of DC And RC Failed.",  
           "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION);  
}
```

Ensuite nous allons tenter de détruire le contexte de rendu. Si cela échoue, un message d'erreur va apparaître.

Destruction du contexte de rendu

```
if(!wglDeleteContext(hRC)) // Sommes-nous capables de détruire le RC?  
{
```

Si nous sommes incapables de détruire le rendu de contexte, le code qui suit va faire apparaître une autre boîte de dialogue qui nous dira que la libération du RC a échoué. hRC a été mis à NULL.

Boîte de dialogue

```
    MessageBox(NULL, "Release Rendering Context Failed.",  
              "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION);  
}  
hRC=NULL; // Mettre RC à NULL  
}
```

Maintenant nous allons vérifier si notre programme possède un contexte de fenêtre et, si c'est le cas, nous allons le libérer. Si nous sommes incapables de libérer le contexte de fenêtre, une autre boîte de message va apparaître et hDC sera mis à NULL.

Libération du contexte de fenêtre

```
if(hDC && !ReleaseDC(hWnd, hDC)) // Sommes-nous capable de libérer le DC  
{  
    MessageBox(NULL, "Release Device Context Failed.",  
              "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION);  
    hDC=NULL; // Mettre DC à NULL  
}
```

Maintenant, nous allons vérifier s'il y a un contexte de fenêtre, et tenter de le libérer utilisant DestroyWindow(hWnd). Si nous étions incapables de détruire la fenêtre, une boîte de message va apparaître et hWnd seront mis à NULL.

Destruction de la fenêtre

```
if(hWnd && !DestroyWindow(hWnd)) // Sommes-nous capables de détruire la fenêtre?  
{  
    MessageBox(NULL, "Could Not Release hWnd.",  
              "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION);  
    hWnd=NULL; // Mettre hWnd à NULL  
}
```

La dernière chose à faire est de désinscrire notre classe de fenêtre. Ceci nous permet de fermer la fenêtre et de rouvrir une autre fenêtre sans recevoir un message d'erreur "Class de fenêtre déjà inscrit".

Désinscription de la classe de fenêtre

```
// Sommes-nous capables de désinscrire cette classe
```

Désinscription de la classe de fenêtre

```
if( !UnregisterClass( "OpenGL" ,hInstance) )
{
    MessageBox(NULL, "Could Not Unregister Class.", "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION);
    hInstance=NULL; // Mettre hInstance à NULL
}
}
```

1.7 - La fonction CreateGLWindow

La prochaine section de code crée notre fenêtre OpenGL. J'ai passé beaucoup de temps à décider si nous devons créer une fenêtre plein écran qui ne nécessite pas beaucoup de code supplémentaire ou un code qui sera facilement modifiable par un utilisateur. J'ai décidé que la seconde option est un meilleur choix. Je reçois souvent ces questions : Comment créer une fenêtre au lieu de plein écran? Comment faire pour changer le titre de la fenêtre ? Comment changer la résolution ou le format des pixels de la fenêtre ? Le code qui suit permet de faire ceci! Donc c'est une bonne leçon et écrire les programmes OpenGL sera plus facile pour vous!

Comme vous le voyez, la fonction retourne un booléen (TRUE ou FALSE), il prend aussi 5 paramètres : titre de la fenêtre, largeur de la fenêtre, hauteur de la fenêtre, nombre de bits (16/24/32), et finalement le drapeau plein écran (si mis à TRUE) ou en mode fenêtre (si mis à FALSE). Nous retournons une valeur de booléen qui nous dira si la création de la fenêtre a réussi.

Fonction de création de fenêtre

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
```

Lorsqu'on demande à Windows de nous trouver un format pixel qui est compatible avec celui que nous voulons, le numéro du mode que Windows choisi sera stocké dans la variable PixelFormat.

Variable pour le format des pixels

```
GLuint PixelFormat; // Contiendra les résultats après avoir trouvé un format adéquat
```

Nous allons utiliser **wc** pour contenir la classe de fenêtre. La classe fenêtre contiendra l'information de notre fenêtre. En changeant les différents champs dans la classe, nous pouvons changer comment la fenêtre va apparaître et se comporte. Chaque fenêtre appartient à une classe de fenêtre. Avant de créer une fenêtre, vous devez inscrire une classe à la fenêtre.

Variable pour la classe de fenêtre

```
WNDCLASS wc; // La structure d'une classe de fenêtre
```

dwExStyle et dwStyle peuvent stocker l'information du style de la fenêtre normale et étendue. J'utilise des variables pour stocker les styles pour pouvoir changer les styles dépendant du type de fenêtre que j'ai besoin de créer (Une fenêtre sans bordure pour une application plein écran ou une fenêtre avec bordure pour un mode fenêtré).

Variables pour le style de fenêtre

```
DWORD dwExStyle; // Style étendu de fenetre
DWORD dwStyle; // Style de fenetre normale
```

Les 5 lignes qui suivent récupèrent les valeurs des positions haut, bas, gauche et droit d'un rectangle. Nous allons utiliser les valeurs pour ajuster la fenêtre pour que l'endroit sur lequel on dessine soit exactement à la bonne résolution. Normalement, si nous créons une fenêtre 640x480, les bordures de la fenêtre vont occuper un peu de notre résolution.

Position de la fenêtre

```
RECT WindowRect; // Rectangle pour la position Haut gauche / Bas droit

WindowRect.left=(long)0; // Mettre la valeur Gauche à 0
WindowRect.right=(long)width; // Mettre la valeur Droite pour la largeur voulue
WindowRect.top=(long)0; // Mettre la valeur Haut à 0
WindowRect.bottom=(long)height; // Mettre la valeur Bas pour la hauteur voulue
```

Dans la ligne de code qui suit nous initialisons la variable globale **fullscreen** à **fullscreenflag**.

Initialisation de la variable fullscreen

```
fullscreen=fullscreenflag; // Initialisation la variable fullscreen
```

Dans la prochaine section de code, nous allons récupérer l'instance de notre fenêtre puis nous déclarons la classe de fenêtre.

Le style CS_HREDRAW et CS_VREDRAW force la fenêtre à se redessiner lorsque la fenêtre est redimensionné. CS_OWNDC crée un contexte de fenêtre privé pour la fenêtre. Ceci veut dire que le DC n'est pas partagé entre les applications. WndProc est la fonction qui surveille les messages pour notre programme. Nous n'avons pas besoin de données supplémentaires pour la fenêtre donc nous mettons les deux champs Extra à 0. Ensuite, nous mettons en place l'instance de la fenêtre. Après, nous mettons hIcon à NULL disant que nous ne voulons pas d'icône de la fenêtre, et pour le pointeur souris nous allons utiliser un pointeur standard. La couleur de fond n'est pas importante (nous le ferons avec OpenGL). Nous ne voulons pas de menu avec cette fenêtre donc nous le mettons à NULL et le nom de la classe peut être n'importe quel nom. J'utiliserai "OpenGL" pour une raison de simplicité.

Mis en place de hInstance et wc

```
hInstance = GetModuleHandle(NULL); // Récupère une instance pour la fenêtre

// Redessine au fur et a mesure et possède propre DC pour la fenêtre
wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
wc.lpfnWndProc = (WNDPROC) WndProc; // WndProc s'occupera des messages
wc.cbClsExtra = 0; // Pas de données supplémentaire
wc.cbWndExtra = 0; // Pas de données supplémentaire
wc.hInstance = hInstance; // Mettre l'instance en place
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Charger l'icône de base
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Mettre en place le curseur de souris
wc.hbrBackground = NULL; // Pas d'image de fond nécessaire pour GL
wc.lpszMenuName = NULL; // Nous ne voulons pas de menu
wc.lpszClassName = "OpenGL"; // Mettre en place le nom de la classe
```

Nous allons maintenant inscrire la classe. Si quelque chose échoue, un message d'erreur va apparaître. Cliquer "OK" dans la boîte terminera le programme.

Inscription de la classe

```
if(!RegisterClass(&wc)) // Tenter d'inscrire une classe de fenêtre
{
    MessageBox(NULL, "Failed To Register The Window Class.", "ERROR", MB_OK | MB_ICONEXCLAMATION);
    return FALSE; // Quitter et retourner FALSE
}
```

Maintenant, nous allons vérifier si le programme va s'exécuter en plein écran ou en mode fenêtré. Si c'est en plein écran, nous allons tenter de se mettre en plein écran.

Code pour le cas plein écran

```
if(fullscreen) // Est-ce qu'on est en plein écran?
{
```

La prochaine section de code est quelque chose qui pose des problèmes à beaucoup de personnes : passer en mode plein écran. Il y a quelque points importants que vous devez garder à l'esprit lorsqu'on passe en mode plein écran. Soyez sûrs que la largeur et l hauteur que vous utilisez en plein écran sont les mêmes que la largeur et l hauteur que vous allez utiliser pour votre fenêtre. Mais, encore plus important, il faut se mettre en plein écran **avant** de créer la fenêtre. Dans ce code, vous n'avez pas besoin de vous inquiéter de la largeur et la hauteur, le plein écran et la taille de la fenêtre sont tous les deux à la taille voulue.

Code pour le mode de la fenêtre

```
DEVMODE dmScreenSettings; // Mode de la fenetre
memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Verifier que la memoire est mis a zero
dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Taille de la structure Devmode
dmScreenSettings.dmPelsWidth = width; // Largeur de la fenetre
dmScreenSettings.dmPelsHeight = height; // Hauteur de la fenetre
dmScreenSettings.dmBitsPerPel = bits; // Nombre de bits par pixel
dmScreenSettings.dmFields = DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;
```

Dans le code ci-dessus, nous avons créé de la place pour stocker nos paramètres vidéos. Nous allons mettre en place la largeur, la hauteur et le nombre de bits que nous voulons. Dans le code qui suit, nous tentons de mettre le programme en plein écran. Nous stockons toutes les informations (largeur, hauteur et nombre de bits) dans dmScreenSettings. Dans la ligne qui suit, **ChangeDisplaySettings** tente de passer aux paramètres qui sont compatibles avec ceux dans dmScreenSettings. J'utilise le paramètre CDS_FULLSCREEN lorsque nous changeons de mode parce c'est supposé enlever la barre des tâches, de plus cela garde la fenêtre au même endroit lorsqu'on passe en mode fenêtre au mode plein écran.

Mis en place du mode choisi

```
// Tenter de mettre le mode choisi et récupéré les résultats.
// REMARQUE : CSD_FULLSCREEN enlève la barre des tâches
if(ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
{
```

Si le mode n'a pas pu être mis en place, le code qui suit va être exécuté. Si un mode compatible plein écran n'a pas été trouvé, une boîte de dialogue va offrir deux solutions... Mettre en mode fenêtré ou quitter le programme.

Cas échéant

```
// Si le mode n'a pas été compatible, on offre deux solutions :
// Quitter ou se mettre en mode fenêtré
if(MessageBox(NULL,
"The Requested Fullscreen Mode Is Not Supported By\nYour Video Card. Use Windowed Mode Instead?",
"NeHe GL",MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
{
```

Si l'utilisateur décide d'utiliser un mode fenêtré, la variable plein écran devient FALSE, et le programme continuera son exécution.

Fin du if et début du else

```
fullscreen=FALSE; //Choisir le mode fenêtré
}
else
{
```

Si l'utilisateur décide de quitter, une boîte de dialogue fera savoir que le programme va se terminer. FALSE sera retourné pour dire au programme que la fenêtre n'a pas été créée correctement. Le programme va ensuite se terminer.

Affichage d'une boîte de dialogue

```
// Afficher une boite de message pour dire a l'utilisateur que le programme se terminera
MessageBox(NULL,"Program Will Now Close.,"ERROR",MB_OK|MB_ICONSTOP);
return FALSE; // Quitter et retourner FALSE
```

Affichage d'une boîte de dialogue

```
}  
}  
}
```

Puisque le code du plein écran peut avoir échoué et l'utilisateur peut avoir décidé de continuer le programme dans une fenêtre. Pour le savoir, on va vérifier la valeur de la variable **fullscreen**.

Test sur l'état du plein écran

```
if(fullscreen) // Sommes-nous toujours en plein écran?  
{
```

Si nous sommes toujours en mode plein écran, nous allons mettre le style étendu à **WS_EX_APPWINDOW** qui force une fenêtre vers la barre de tâches une fois que notre fenêtre est visible. Pour le style de la fenêtre, nous allons créer une fenêtre **WS_POPUP**. Ce genre de fenêtre n'a pas de bordure, ce qui est parfait pour le mode plein écran.

Finalement, on enlève le pointeur de souris. Si votre programme n'est pas interactif, il est toujours plus intéressant d'enlever le pointeur de souris en plein écran. C'est votre décision.

Style étendu et normal de la fenêtre et enlever la souris

```
dwExStyle=WS_EX_APPWINDOW; // Window Extended Style  
dwStyle=WS_POPUP; // Windows Style  
ShowCursor(FALSE); // Hide Mouse Pointer  
}  
else  
{
```

Si nous utilisons une fenêtre au lieu du mode plein écran, nous allons ajouter **WS_EX_WINDOWEDGE** au style étendu. Ceci va donner à la fenêtre un style plus 3D. Pour le style, nous allons utiliser **WS_OVERLAPPEDWINDOW** au lieu de **WS_POPUP**. **WS_OVERLAPPEDWINDOW** va créer une fenêtre avec un titre, une bordure, un menu et des boutons de minimisation et maximisation.

Style normal et étendu de la fenêtre

```
dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // Style étendu de la fenetre  
dwStyle=WS_OVERLAPPEDWINDOW; // Style de la fenetre  
}
```

La ligne qui suit ajuste notre fenêtre dépendant du style de fenêtre que nous sommes en train de créer. L'ajustement créera la résolution voulue. Normalement, les bordures vont recouvrir des parties de la fenêtre. En utilisant la fonction **AdjustWindowRectEx**, aucune partie de la scène OpenGL va être recouverte par les bords. A la place, la fenêtre sera légèrement agrandie. En plein écran, cet appel n'a aucune conséquence

Mise en place de la taille de la fenêtre

```
// Ajuste la fenetre a la fenetre a la taille voulue  
AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle);
```

Dans la prochaine section de code, nous allons créer une fenêtre et vérifier si elle a été créée correctement. Nous allons passer tous les paramètres nécessaires à la fonction **CreateWindowEx**. Les paramètres sont : le style étendu que nous voulons utiliser, le nom de classe (qui doit être le même que celui que vous avez donné à la classe Fenêtre), le titre de la fenêtre, le style de la fenêtre, la position en haut et à gauche de la fenêtre (0,0 est une bonne idée), la largeur et la hauteur de la fenêtre, on ne veut pas d'une fenêtre parent et nous n'avons pas de menu (donc ces deux derniers paramètres sont mis à NULL), nous passons l'instance de la fenêtre et enfin nous passons NULL.

Remarquez que nous avons inclus les styles **WS_CLIPSIBLINGS** et **WS_CLIPCHILDREN** avec le style de la fenêtre que nous avons décidé d'utiliser. **WS_CLIPSIBLINGS** et **WS_CLIPCHILDREN** sont tous les deux **nécessaires**

pour travailler correctement. Ces styles empêchent d'autres fenêtres de dessiner au-dessus ou dans notre fenêtre OpenGL.

Création de la fenêtre

```

if (!(hWnd=CreateWindowEx( dwExStyle, // Style étendu pour la fenetre
    "OpenGL", // Nom de la classe
    title, // Titre de la fenetre
    WS_CLIPSIBLINGS | // Style de la fenetre necessaire
    WS_CLIPCHILDREN | // Style de la fenetre necessaire
    dwStyle, // Style de la fenetre choisi
    0, 0, // Position des fenetres
    WindowRect.right-WindowRect.left, // Calcul de la largeur ajustee
    WindowRect.bottom-WindowRect.top, // Calcul de la hauteur ajustee
    NULL, // Pas de fenetre parent
    NULL, // Pas de menu
    hInstance, // Instance
    NULL))) // Ne rien passer a WM_CREATE
    
```

Ensuite, nous vérifions si notre fenêtre s'est créée correctement. Si notre fenêtre a été créée, **hWnd** va contenir le descripteur de fenêtre. Si la fenêtre n'a pas été créée alors le code ci-dessous va afficher une boîte de dialogue avant de quitter le programme.

En cas d'échec

```

{
    KillGLWindow(); // Remis à zéro de l'affichage
    MessageBox(NULL,"Erreur dans la création de la fenêtre.",
        "ERROR",MB_OK|MB_ICONEXCLAMATION);
    return FALSE; // Retourner FALSE
}
    
```

La prochaine section de code décrit le format du pixel. Nous devons choisir un format qui supporte OpenGL, l'utilisation d'un double tampon et l'utilisation du RGBA (rouge, vert, bleu et un canal alpha). On essaie de trouver un format de pixel qui convient par rapport au nombre de bits qu'on a décidé (16, 24 ou 32 bits). Finalement, nous mettons en place un tampon-Z de 16 bits. Les paramètres qui restent ne sont soit pas utilisés, soit pas importants (à part pour le tampon stencil et le tampon d'accumulation).

Format de pixel

```

static PIXELFORMATDESCRIPTOR pfd= // pfd dit aux fenetres quel format nous interesse
{
    sizeof(PIXELFORMATDESCRIPTOR), // Taille de ce descripteur de format pixel
    1, // Numero de version
    PFD_DRAW_TO_WINDOW | // Format doit fonctionner avec la fenetre
    PFD_SUPPORT_OPENGL | // Format doit fonctionner avec OpenGL
    PFD_DOUBLEBUFFER, // Format doit fonctionner le double tampon
    PFD_TYPE_RGBA, // Demander un format RGBA
    bits, // Choisir le nombre de bits par pixel
    0, 0, 0, 0, 0, 0, // Bits de couleur ignorés
    0, // Pas de tampon Alpha
    0, // Bit de decalage ignore
    0, // Pas de tampon d'accumulation
    0, 0, 0, 0, // Bits d'accumulation ignores
    16, // 16Bit du tampon Z (Tampon de profondeur)
    0, // Pas de tampon Stencil
    0, // Pas de tampon auxiliaire
    PFD_MAIN_PLANE, // Couche de dessin de base
    0, // Reserve
    0, 0, 0 // Masques de couche ignores
};
    
```

S'il n'y a pas eu d'erreurs lors de la création de la fenêtre, nous allons tenter de récupérer un contexte OpenGL. Si nous ne pouvons pas avoir un contexte, un message de dialogue donnera un message et le programme se terminera.

Récupération d'un contexte de fenêtre

```
if(!(hDC=GetDC(hWnd))) // Avons-nous un contexte de fenetre?
{
KillGLWindow(); // Detruire la fenetre
MessageBox(NULL,"Can't Create A GL Device Context.," "ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Retourner FALSE
}
```

Si nous avons réussi à avoir un contexte de fenêtre pour notre fenêtre OpenGL, nous allons tenter d'avoir un format de pixel qui correspond à celui décrit plus haut. Si Windows ne peut pas en trouver, une boîte de dialogue va apparaître et le programme quittera.

Choix du format de pixel

```
// Est-ce que la fenetre a trouve un format de pixel compatible?
if(!(PixelFormat=ChoosePixelFormat(hDC,&pfd))
{
KillGLWindow(); // Detruire la fenetre
MessageBox(NULL,"Can't Find A Suitable PixelFormat.," "ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Retourner FALSE
}
```

Si Windows a trouvé un format pixel compatible, nous allons tenter de l'utiliser. Si ce format ne peut pas être utilisé, un message d'erreur va apparaître et le programme va se terminer (retourner FALSE).

Mis en place du format de pixel

```
// Pouvons-nous mettre en place le format de Pixels ?
if(!SetPixelFormat(hDC,PixelFormat,&pfd)
{
KillGLWindow(); // Detruire la fenetre
MessageBox(NULL,"Can't Set The PixelFormat.," "ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Retourner FALSE
}
```

Si le format pixel a été mis en place correctement nous allons tenter d'avoir un contexte de rendu. Si nous n'y arrivons pas, un message d'erreur sera affiché et le programme quittera (retourner FALSE).

Création d'un contexte de rendu

```
if(!(hRC=wglCreateContext(hDC)) // Sommes-nous capable d'avoir un contexte de rendu ?
{
KillGLWindow(); // Detruire la fenetre
MessageBox(NULL,"Can't Create A GL Rendering Context.," "ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Retourner FALSE
}
```

Si nous n'avons pas eu d'erreurs, nous avons réussi à avoir un contexte de fenêtre et un contexte de rendu, tout ce qu'il reste à faire c'est rendre le contexte de rendu actif. Si nous n'arrivons pas à rendre le contexte de rendu actif, un message de dialogue apparaîtra et le programme quittera (retournant FALSE).

Activer le contexte de rendu

```
if(!wglMakeCurrent(hDC,hRC) // Tenter d'activer le contexte de rendu
{
KillGLWindow(); // Detruire la fenetre

MessageBox(NULL,"Can't Activate The GL Rendering Context.,"
"ERROR",MB_OK|MB_ICONEXCLAMATION);

return FALSE; // Retourner FALSE
}
```

Si tout s'est bien passé, notre fenêtre OpenGL a été créée et nous allons montrer la fenêtre, la mettre en premier plan (lui donner plus de priorité) et la rendre active. Ensuite, nous appellerons **ReSizeGLScene** avec, comme paramètre, la largeur et hauteur de l'écran pour mettre en place la perspective OpenGL.

Code de mise en place

```
ShowWindow(hWnd, SW_SHOW); // Montrer la fenetre
SetForegroundWindow(hWnd); // Mettre une priorite plus elevee
SetFocus(hWnd); // Rendre le clavier actif pour cette fenetre
ReSizeGLScene(width, height); // Mettre en place la perspective GL
```

Finalement, nous allons appeler `InitGL()`, où nous allons mettre en place l'éclairage, les textures et tout ce qu'il faut pour l'initialisation. Vous pouvez faire des tests d'erreurs dans cette fonction. Vous pourrez retourner `TRUE` (si tout se passe bien) ou `FALSE` (si quelque chose ne passe pas bien). Par exemple, si vous chargez les textures dans `InitGL` et avez eu une erreur, vous voudrez peut-être que le programme s'arrête. Si vous retournez `FALSE` à partir de la fonction `InitGL`, le code ci-dessous va le voir et va pouvoir se quitter.

Si l'initialisation OpenGL échoue

```
if(!InitGL()) // Initialise notre nouvelle fenetre GL
{
KillGLWindow(); // Detruire la fenetre
MessageBox(NULL, "Initialization Failed.", "ERROR", MB_OK | MB_ICONEXCLAMATION);
return FALSE; // Retourner FALSE
}
```

Si nous sommes arrivés jusqu'ici, on peut supposer que la création de la fenêtre a réussi. Nous pouvons retourner `VRAI` à `WinMain()` pour lui dire qu'il n'y a pas eu d'erreurs. Ceci va l'empêcher de quitter le programme.

En cas de succès

```
return TRUE; // Succes
}
```

1.8 - La fonction WndProc

C'est ici que tous les messages sont gérés. Lorsque nous avons inscrit la classe de fenêtre, nous lui avons dit d'appeler cette fonction pour gérer les messages de fenêtre.

Prototype de WndProc

```
HRESULT CALLBACK WndProc( HWND hWnd, // Identifiant de cette fenetre
UINT uMsg, // Message pour cette fenetre
WPARAM wParam, // Information supplementaire
LPARAM lParam) // Information supplementaire
{
```

Le code qui teste la valeur `uMsg` avec un `switch`. `uMsg` contient le type du message que nous allons devoir gérer.

Un switch sur les types de messages

```
switch (uMsg) // Verifier pour des messages Windows
{
```

Si `uMsg` vaut `WM_ACTIVATE`, on vérifie si notre fenêtre est encore active. Si notre fenêtre a été minimisée, la variable `active` vaudra `FALSE`. Si la fenêtre est `active`, la variable `active` vaudra `TRUE`.

Le cas WM_ACTIVATE

```
case WM_ACTIVATE: // Est-ce que le message est WM_ACTIVATE
{
```

Le cas WM_ACTIVATE

```
if(!HIWORD(wParam)) // Est-ce que la fenetre est minimisee?
{
    active=TRUE;      // La fenetre est active
}
else
{
    active=FALSE;    // La fenetre n'est pas active
}
return 0;           // On retourne a la boucle evenementielle
}
```

Si le message est égal à WM_SYSCOMMAND (commande système), nous allons comparer **wParam** avec les différentes possibilités de ce type de message. Si **wParam** vaut SC_SCREENSAVE ou SC_MONITORPOWER, soit un écran de veille est en train de tenter de démarrer, soit l'écran est en train de rentrer dans un mode d'économie d'énergie. En retournant 0, on évite que ces deux événements se produisent.

Le cas WM_SYSCOMMAND

```
case WM_SYSCOMMAND: // Intercepte les commandes systeme
{
    switch (wParam) // Verifie les appels systemes
    {
        case SC_SCREENSAVE: // Est-ce que l'ecran de veille commence?
        case SC_MONITORPOWER: // Est-ce que l'ecran tente de commencer l'economisation d'energie
            return 0; // On empeche que cela se passe
    }
    break; // On sort
}
```

Si **uMsg** vaut WM_CLOSE, la fenêtre va se fermer. On va envoyer un message pour quitter que la boucle de la fonction **main** va récupérer. La variable **done** sera mis à TRUE, la boucle globale s'arrêtera et le programme se terminera.

Le cas WM_CLOSE

```
case WM_CLOSE: // Est-ce qu'on a reçu un message de fermeture?
{
    PostQuitMessage(0); // Envoyer un message pour quitter
    return 0; // Sortir de la fonction
}
```

Si une touche du clavier est en train d'être appuyée, on peut découvrir quelle est la touche en regardant **wParam**. On mettra la case correspondante du tableau **keys** à TRUE. Cela me permet de vérifier dans le reste du code quelles touches sont appuyées. Ceci me permet aussi de gérer l'appui simultané de plusieurs touches.

Le cas WM_KEYDOWN

```
case WM_KEYDOWN: // Est-ce qu'on appuie sur une touche
{
    keys[wParam] = TRUE; // Si c'est le cas, on met à jour la case correspondante
    return 0; // Sortir de la fonction
}
```

Si une touche a été relâchée, on va savoir quelle est cette touche en regardant **wParam**. On va donc mettre la case correspondante du tableau **keys** à FALSE. Comme cela, nous allons pouvoir vérifier dans le reste du code si une touche est encore appuyée ou non. Chaque touche est représentée par un nombre entre 0 et 255. Par exemple, lorsque j'appuie sur la touche représentée par 40, **keys[40]** va valoir TRUE. Lorsque je relâche cette touche, cette case deviendra FALSE. C'est comme cela qu'on pourra gérer le clavier.

Le cas WM_KEYUP

```

case WM_KEYUP:           // Est-ce qu'une touche a ete relachee?
{
    keys[wParam] = FALSE; // La case associee devient FALSE
    return 0;           // Sortir de la fonction
}
    
```

Lorsqu'on redimensionne la fenêtre, **uMsg** va éventuellement valoir **WM_SIZE**. Nous pourrions utiliser les valeurs **LOWORD** et **HIWORD** de **lParam** pour récupérer les nouvelles dimensions de la fenêtre. Nous allons donc passer ces dimensions à **ReSizeGLScene**. La scène **OpenGL** est donc redimensionnée aux bonnes dimensions.

Le cas WM_SIZE

```

case WM_SIZE:           // Redimensionner la fenetre
{
    ReSizeGLScene(LOWORD(lParam),HIWORD(lParam)); // LoWord=Largeur, HiWord=Hauteur
    return 0;           // Sortir de la fonction
}
    
```

Tous les messages qui nous intéressent pas seront passés à la fonction **DefWindowProc** pour que Windows puissent s'en occuper.

Tous les autres messages

```

// Passer tous les messages qui ne sont pas geres à DefWindowProc
return DefWindowProc(hWnd,uMsg,wParam,lParam);
}
    
```

1.9 - La fonction WinMain

Ceci est le point d'entrée à notre application Windows. C'est ici que nous créons la fenêtre, gère les messages et regarde pour l'interaction de l'utilisateur.

La fonction WinMain

```

int WINAPI WinMain( HINSTANCE hInstance, // Instance
    HINSTANCE hPrevInstance, // Instance precedente
    LPSTR lpCmdLine, // Parametres de ligne de commande
    int nCmdShow) // L'etat d'affichage de la fenetre
{
    
```

On met en place deux variables. **msg** doit être utilisé pour vérifier s'il n'y a pas de messages en attente qui doivent être gérés. La variable **done** commence avec la valeur **FALSE**. Ceci veut dire que notre programme n'a pas fini d'être exécuté. Du moment que **done** demeure à **FALSE**, le programme va continuer. Dès qu'il passera à **TRUE**, le programme va quitter.

Variables locales

```

MSG msg; // Structure d'un message Windows
BOOL done=FALSE; // Variable pour gerer la boucle infinie
    
```

Cette portion de code est entièrement facultative. Il affiche une boîte de dialogue pour vous demander d'exécuter le programme en plein écran. Si l'utilisateur clique sur **NON**, la variable **fullscreen** passe à **FALSE** (la valeur par défaut est **TRUE**) et le programme se lancera dans une fenêtre.

Interaction avec l'utilisateur

```

// Demander a l'utilisateur quel mode d'execution (fenetre ou plein ecran)
if(MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?",
    "Start FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
    
```

Interaction avec l'utilisateur

```
{
  fullscreen=FALSE;    // Mode fenêtré
}
```

C'est comme ceci que nous créons une fenêtre OpenGL. Nous passons le titre, la largeur, la hauteur, le nombre de bit par pixel et le mode de fenêtrage à la fonction **CreateGLWindow**. C'est tout! Je suis assez content de la simplicité de ce code. Si la fenêtre n'a pas été codé pour une raison ou une autre, FALSE sera retourné et notre programme pourra quitter (avec return 0).

Création de la fenêtre

```
// Créer notre fenetre OpenGL
if(!CreateGLWindow("NeHe's OpenGL Framework",640,480,16,fullscreen))
{
  return 0; // Quitter si la fenetre n'a pas ete creee
}
```

C'est ici le début de notre boucle. Tant que **done** vaut FALSE, la boucle continuera.

Boucle globale

```
while(!done) // Boucle qui s'exécute tant que done=TRUE
{
```

La première chose que nous devons vérifier s'il y a des messages qui attendent. En utilisant la fonction **PeekMessage**, nous pouvons vérifier pour des messages sans arrêter notre programme. Beaucoup de programmes utilisent **GetMessage**. Ceci fonctionne aussi, mais avec **GetMessage**, votre programme ne fait rien jusqu'à la réception d'un message.

On vérifie si un message est en attente

```
if(PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Y-a-t-il un message en attente?
{
```

Dans la prochaine portion de code, nous vérifions si un message WM_QUIT a été envoyé. Si le message courant est WM_QUIT causé par **PostQuitMessage(0)**, la variable **done** est mise à TRUE. Ceci provoquera la fin du programme.

Allons-nous quitter?

```
if(msg.message==WM_QUIT) // Avons-nous reçu un message pour quitter?
{
  done=TRUE; // Si c'est le cas, on met done a TRUE
}
else // Sinon gere les messages
{
```

Si ce n'est pas un message pour quitter, nous traduisons le message, puis nous l'envoyons avec la fonction **DispatchMessage** pour que **WndProc** ou Windows puissent s'en occuper.

Traduire le message et l'envoyer

```
TranslateMessage(&msg); // Traduire le message
DispatchMessage(&msg); // Envoyer le message
}
else // S'il n'y a pas de message
{
```

S'il n'y avait pas de messages, nous allons dessiner la scène OpenGL. La première ligne de code ci-dessous vérifie si la fenêtre est active. Si la touche Echap a été appuyée alors la variable **done** est mise à TRUE, provoquant la fin du programme.

Si la fenêtre est active

```
// Dessine la scène. Verifier si Echap a ete appuye
if(active) // Est-ce que le programme est actif?
{
    if(keys[VK_ESCAPE]) // Est-ce que Echap a ete appuye?
    {
        done=TRUE; // ESC signale la fin du programme
    }
    else // On ne quitte pas encore, on va dessiner
    {
```

Si le programme est actif et echap n'a pas été appuyé, nous allons dessiner la scène et échanger les tampons (En utilisant un double tampon, nous obtiendrons animation sans scintillement). En utilisant un double tampon, nous dessinons la scène sur un écran *invisible*. Lorsque nous échangeons les tampons, la scène que nous voyons devient l'écran invisible et la scène qui était invisible devient visible. En faisant comme cela, nous ne voyons jamais la scène en train d'être dessiné. L'image apparaît directement entière.

Dessin de la scène

```
DrawGLScene(); // Dessine la scene
SwapBuffers(hdc); // Echange les tampons (Double tampon)
}
```

La prochaine section de code nous permet de passer entre le mode plein écran et le mode fenêtré.

Passage entre les modes fenêtré et plein écran

```
if(keys[VK_F1]) // Est-ce que F1 est appuye?
{
    keys[VK_F1]=FALSE; // On passe la touche a FALSE
    KillGLWindow(); // Fermer la fenetre OpenGL
    fullscreen=!fullscreen; // Passer entre le plein ecran / mode fenetre

    // Recréer la fenetre OpenGL
    if(!CreateGLWindow("NeHe's OpenGL Framework",640,480,16,fullscreen))
    {
        return 0; // Si la fenetre n'a pas ete creee
    }
}
}
```

Si la variable **done** n'est plus FALSE, le programme quitte. Nous fermons correctement la fenêtre OpenGL pour que tout soit libéré et nous quittons le programme.

Fermer la fenêtre

```
// Fermeture
KillGLWindow(); // Fermer la fenetre
return (msg.wParam); // Quitter le programme
}
```

1.10 - Conclusion

Dans ce tutoriel, j'ai tenté d'expliquer, avec autant de détail que possible, chaque étape nécessaire pour mettre en place et créer un programme OpenGL en plein écran. Ce programme quittera si la touche ESC a été appuyée et fait attention si la fenêtre est active ou non.

Jeff Molofee (NeHe)

2 - Téléchargements

Compte tenu du nombre de versions de code sources pour les tutoriels nehe, nous les laissons en anglais. En principe, si vous avez compris le code présenté dans ce tutoriel (et les tutoriels antérieurs), vous n'aurez pas de mal à le comprendre :

- **ASM (Conversion par Foolman)**
- **BeOS (Conversion par Rene Manqueros)**
- **Borland C++ Builder 6 (Conversion par Christian Kindahl)**
- **C# (Conversion par Joachim Rohde)**
- **Code Warrior 5.3 (Conversion par Scott Lupton)**
- **CygWin (Conversion par Stephan Ferraro)**
- **D (Conversion par Familia Pineda Garcia)**
- **Delphi (Conversion par Michal Tucek)**
- **Dev C++ (Conversion par Dan)**
- **Game GLUT (Conversion par Milikas Anastasios)**
- **Irix (Conversion par Lakmal Gunasekara)**
- **Java (Conversion par Jeff Kirby)**
- **Java/SWT (Conversion par Victor Gonzalez)**
- **JOGL (Conversion par Kevin J. Duling)**
- **LCC Win32 (Conversion par Robert Wishlaw)**
- **Linux (Conversion par Richard Campbell)**
- **Linux GLX (Conversion par Mihael Vrbanec)**
- **Linux SDL (Conversion par Ti Legget)**
- **LWJGL (Conversion par Mark Bernard)**
- **Mac OS (Conversion par Anthony Parker)**
- **Mac OS X/Cocoa (Conversion par Bryan Blackburn)**
- **MASM (Conversion par Nico (Scalp))**
- **Pelles C (Conversion par Pelle Orinius)**
- **Perl (Conversion par Cora Hussey)**
- **Power Basic (Conversion par Angus Law)**
- **Python (Conversion par John Ferguson)**
- **Scheme (Conversion par Jon DuBois)**
- **Solaris (Conversion par Lakmal Gunasekara)**
- **Visual Basic (Conversion par Ross Dawson)**
- **Visual Fortran (Conversion par Jean-Philippe Perois)**
- **Visual Studio**
- **VB.Net CsGL (Conversion par X)**
- **Visual Studio NET (Conversion par Grant James)**

3 - Remerciements

Merci à **Miles** et à **wichtounet** pour leur relecture.

4 - Liens

Sommaire

Tutoriel prochain : Coloration

