

Réalisation d'un moteur 3D en C++ - partie II : le renderer



par [Laurent Gomila](#)

Date de publication : 13/12/2004

Dernière mise à jour : 01/08/2005

Après avoir mis au point divers outils utiles à notre futur coding, nous allons dans cette seconde partie nous attaquer à un gros morceau du moteur : le coeur du système de rendu, le renderer.

- 1 - Introduction
- 2 - Fonctionnement du renderer
 - 2.1 - Principe
 - 2.2 - Séparation des renderers en DLL
- 3 - Contenu
 - 3.1 - Initialisations
 - 3.2 - Gestion des matrices
 - 3.3 - Gestion des couleurs
- 4 - Les enums
- 5 - Conclusion
- 6 - Téléchargements
- 7 - Remerciements

1 - Introduction

La gestion des API 3D est une tâche délicate, elle constitue le coeur de notre moteur. Absolument tout le reste sera construit sur ces classes de base, elles devront donc être très performantes et offrir les fonctionnalités souhaitées le plus simplement possible. Il faudra également veiller à ce que ces classes, qui seront utilisées et réutilisées sans cesse dans le moteur, n'introduisent ni bug ni fuite de mémoire, mais pour cela nous avons déjà quelques outils bien utiles ! Enfin, il sera indispensable que nos classes encapsulent totalement les API graphiques, ainsi nous disposerons pour la suite d'une couche qui fait totalement abstraction de l'API utilisée.

Rentrons à présent un peu plus dans le détail de ces classes, quelles seront-elles exactement et quel va être leur rôle ? Tout d'abord nous devons gérer la géométrie, c'est-à-dire créer des classes de **vertex buffers** et **index buffers** pour stocker nos triangles. Ensuite nous devons également gérer les **textures** et tous les types dont on pourra avoir besoin : texture de rendu, texture dynamique, texture statique, ... Outil devenus indispensables depuis quelques années, nous gérerons également les **vertex shaders** et les **pixel shaders**, qui permettront un haut degré de liberté pour le rendu et qui nous serviront à mettre en oeuvre les plus chouettes effets graphiques. Enfin, et c'est certainement l'élément le plus important, nous aurons un **renderer**. C'est lui qui va gérer nos buffers, nos textures, nos shaders, ainsi que tout ce qui est relatif au rendu de base. La grande majorité du code spécifique aux API 3D sera encapsulé dans cette classe.

C'est donc sur cet élément charnière du moteur que nous allons nous attarder dans cette partie, et nous étudierons en détail les autres classes dans les tutoriels suivants.

La conception de ces différentes classes peut se révéler très problématique, car il faut qu'elles puissent parfaitement encapsuler les mécanismes correspondant dans les diverses API 3D. Par exemple, si l'on munit nos vertex buffers d'une méthode qui est utilisée par les textures d'OpenGL mais qui n'existe pas avec celles de DirectX, nous allons être bloqués. Il faut donc analyser avec le plus grand soin les API visées, et en extraire une base commune qui nous permettra de construire notre couche abstraite. Ce qui peut vraiment mener à de grosses prises de tête ! Si je peux vous donner un conseil, c'est donc de prendre tout le temps nécessaire à analyser et concevoir ces classes, car le moindre hic pourra vous conduire à recoder entièrement le coeur de votre moteur.

Cette introduction d'ordre général étant faite, attardons-nous dès à présent sur notre fameux renderer.

2 - Fonctionnement du renderer

2.1 - Principe

Comme expliqué en introduction, le renderer sera en fait une grosse encapsulation des API 3D. C'est lui qui servira en quelque sorte d'interface entre le moteur et l'API, c'est également lui qui gèrera et manipulera les autres objets directement issus des API 3D (buffers, textures, shaders, ...). Il faudra donc créer un renderer spécialisé pour chaque API que l'on voudra gérer. Notre moteur quant à lui accèdera à ce renderer via une classe de base abstraite, *IRenderer*, de laquelle vont donc dériver nos renderers spécialisés. Cette classe ne contiendra a priori que des fonctions virtuelles pures, qui seront à redéfinir dans les classes dérivées et implémentées avec l'API concernée.

Prenons un exemple simple pour illustrer ceci : les fonctions de début et fin de rendu.

Classe de base

```
class YES_EXPORT IRenderer
{
public :

    // Démarre le rendu de la scène
    virtual void BeginScene() const = 0;

    // Termine le rendu de la scène
    virtual void EndScene() const = 0;
};
```

Spécialisation DirectX9

```
void CDX9Renderer::BeginScene() const
{
    m_Device->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER | D3DCLEAR_STENCIL,
0x00008080, 1.0f, 0x00);
    m_Device->BeginScene();
}

void CDX9Renderer::EndScene() const
{
    m_Device->EndScene();
    m_Device->Present(NULL, NULL, NULL, NULL);
}
```

Spécialisation OpenGL

```
void COGLRenderer::BeginScene() const
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
}

void COGLRenderer::EndScene() const
{
    SwapBuffers(m_Handle);
}
```

Et voilà, nous avons maintenant un gestionnaire de rendu indépendant de l'API. Il n'y a plus qu'à faire de même pour tout le reste, et nous allons voir de quoi est composé exactement ce reste, puis comment l'implémenter avec chaque API. Nous ne pourrons pas tout détailler dans ce tutoriel car beaucoup de fonctions sont liées à des classes que nous ne verrons que plus tard (buffers, textures, shaders, ...), notre renderer s'enrichira donc au gré des prochains articles.

2.2 - Séparation des renderers en DLL

Nous avons vu comment allait fonctionner notre renderer, mais qu'en est-il du côté du moteur ? Comment va-t-on agencer tout cela pour pouvoir utiliser telle ou telle API à la volée ? Le plus simple est de créer une bibliothèque dynamique (DLL) pour chaque API, puis d'y exporter notre renderer. Ainsi tout le code spécifique à une API sera entièrement contenu dans une DLL, qui sera chargée et déchargée à souhait par notre moteur. L'avantage est qu'on peut ainsi recompiler le moteur ou un renderer spécifique indépendamment du reste. On peut également ajouter un nouveau renderer sans toucher au moteur, simplement en créant une spécialisation de `IRenderer` dans une nouvelle DLL.

Voici le code correspondant à toutes ces pensées philosophiques, vous remarquerez l'utilisation des outils développés dans le premier tutoriel : **CPlugin** pour gérer la DLL, et **Assert** pour sécuriser au maximum notre code. Les fonctions de chargement / dechargement / récupération sont directement intégrées à notre classe `IRenderer`, en tant que membres statiques :

Gestion du renderer

```
class YES_EXPORT IRenderer
{
public :

    // ...

    // Charge un renderer à partir d'une DLL
    static void Load(const std::string& DllName);
    {
        Destroy();
        s_Instance = s_Library.Load(DllName);
        Assert(s_Instance != NULL);
    }

    // Détruit le renderer
    static void Destroy();
    {
        delete s_Instance;
        s_Instance = NULL;
    }

    // Renvoie l'instance du renderer
    static IRenderer& Get();
    {
        Assert(s_Instance != NULL);
        return *s_Instance;
    }

private :

    // Données membres
    static IRenderer* s_Instance; // Instance du renderer chargée
    static CPlugin<IRenderer> s_Library; // Helper pour manipuler la DLL
};

// Chargement du renderer DirectX9
IRenderer::Load("DirectX9.dll");
IRenderer::Get().DrawMeASheep();

// Le mouton DirectX9 n'est pas assez beau, allons voir celui d'OpenGL
IRenderer::Destroy();
IRenderer::Load("OpenGL.dll");
IRenderer::Get().DrawMeASheep();
```

Le changement d'API graphique à la volée parait facile, mais attention cependant : il faudra par la suite détruire tous les objets créés (textures, shaders, buffers, ...) puis les recréer à l'identique avec le nouveau renderer. C'est une tâche assez fastidieuse, c'est pourquoi on choisit en général le renderer

au lancement du programme et on n'y touche plus par la suite.

L'utilisation de la classe CPlugin est relativement aisée, mais si vous avez une bonne mémoire peut-être vous rappellerez-vous ce qui a été dit dans le premier tutoriel : il faut que les DLL manipulées par CPlugin se plient à certaines règles, pour le moins assez simple. J'avais notamment dit que nos DLL devraient exporter une fonction, StartPlugin, qui renverrait un pointeur sur un objet, ici un IRenderer. Voici donc cette fameuse fonction, que nous prendrons soin d'adjoindre à nos DLL de renderer :

Exportation du renderer

```
// Dans DirectX9.dll
#include "DX9Render.h"
extern "C" __declspec(dllexport) IRenderer* StartPlugin()
{
    return &CDX9Renderer::Instance();
}

// Dans OpenGL.dll
#include "OGLRender.h"
extern "C" __declspec(dllexport) IRenderer* StartPlugin()
{
    return &COpenGL::Instance();
}
```

Vous l'aurez deviné : CDX9Renderer et COGLRenderer sont tous deux des singletons, ce qui est bien naturel. Notez bien ici que nous n'avons pas utilisé la macro YES_EXPORT pour exporter cette fonction. En effet, nous allons l'importer dynamiquement depuis notre moteur, nous n'aurons donc pas besoin de savoir que StartPlugin est exportée. Nous (le moteur en fait) n'avons même pas besoin que celle-ci existe, nous pouvons donc même nous passer de sa déclaration ! Bien sûr l'import dynamique n'a pas que des avantages : la moindre erreur sur cette fonction (faute de frappe, prototype différent) sera cette fois non pas signalé par une erreur d'édition de liens, mais provoquera un de ces plantages qu'on aime tant. Heureusement notre classe CPlugin est suffisamment robuste, et les erreurs de ce genre seront identifiées immédiatement.

3 - Contenu

Maintenant que nous avons conçu efficacement notre renderer, nous pouvons utiliser l'API de notre choix à la volée. Il va à présent falloir remplir ce renderer et le munir de toutes les fonctionnalités dont on pourra avoir besoin pour notre rendu. Bien sûr on ne pourra pas tout prévoir à l'avance, et notre renderer s'enrichira au fur et à mesure, mais voici déjà les fonctions vitales.

3.1 - Initialisations

La toute première chose à faire avec notre API 3D sera de l'initialiser correctement, ce que nous allons reléguer à une fonction que nous nommerons **IRenderer::Initialize**. Rien de bien méchant du côté de DirectX :

```
void CDX9Renderer::Initialize(HWND Hwnd)
{
    // Récupération de l'instance de l'objet D3D9
    if ((m_Direct3D = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
        throw CDX9Exception("Direct3DCreate9", "Initialize");

    // Récupération de la taille de la fenêtre
    RECT Rect;
    GetClientRect(Hwnd, &Rect);

    // Initialisation des paramètres du device
    D3DPRESENT_PARAMETERS PresentParameters;
    ZeroMemory(&PresentParameters, sizeof(D3DPRESENT_PARAMETERS));
    PresentParameters.FullScreen_RefreshRateInHz = 0;
    PresentParameters.PresentationInterval      = D3DPRESENT_INTERVAL_IMMEDIATE;
    PresentParameters.SwapEffect                 = D3DSWAPEFFECT_DISCARD;
    PresentParameters.BackBufferWidth           = Rect.right - Rect.left;
    PresentParameters.BackBufferHeight          = Rect.bottom - Rect.top;
    PresentParameters.BackBufferFormat          = D3DFMT_X8R8G8B8;
    PresentParameters.AutoDepthStencilFormat     = D3DFMT_D24S8;
    PresentParameters.EnableAutoDepthStencil    = true;
    PresentParameters.Windowed                  = true;

    // Création du device
    if (FAILED(m_Direct3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
                                       Hwnd, D3DCREATE_HARDWARE_VERTEXPROCESSING,
                                       &PresentParameters, &m_Device)))
        throw CDX9Exception("CreateDevice", "Initialize");

    // States par défaut
    m_Device->SetRenderState(D3DRS_DITHERENABLE, true);
    m_Device->SetRenderState(D3DRS_LIGHTING, false);
    m_Device->SetRenderState(D3DRS_ZENABLE, true);
    m_Device->SetRenderState(D3DRS_FOGENABLE, false);
    m_Device->SetRenderState(D3DRS_ALPHATESTENABLE, false);
    m_Device->SetRenderState(D3DRS_ALPHABLENDENABLE, false);
    m_Device->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    m_Device->SetRenderState(D3DRS_STENCILMASK, 0xFF);
    m_Device->SetRenderState(D3DRS_STENCILWRITEMASK, 0xFF);

    m_Device->SetSamplerState(0, D3DSAMP_ADDRESSU, D3DTADDRESS_CLAMP);
    m_Device->SetSamplerState(0, D3DSAMP_ADDRESSV, D3DTADDRESS_CLAMP);
    m_Device->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
    m_Device->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
    m_Device->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_NONE);
}
```

L'initialisation de DirectX se résume en gros à la création de l'objet D3D, de l'objet Device, puis le paramétrage des states par défaut. Vous noterez que cette fonction d'initialisation est très limitée au niveau des choix : on se place en mode fenêtré, 32 bits, stencil buffer 8 bits, pas de synchronisation

verticale, etc... Si vous êtes novice en programmation DirectX et que ce code ne vous parle pas beaucoup, vous pourrez trouver toutes les explications nécessaires dans la documentation et les samples du SDK DirectX. Plus tard nous rendrons l'initialisation beaucoup plus flexible, avec la possibilité de changer tous ces paramètres, ainsi qu'une gestion des capacités du hardware et choix du mode par défaut approprié. Mais restons pour le moment dans des choses simples, l'important dans ce genre de projet est de coder progressivement et de n'ajouter des fonctionnalités que lorsqu'on est certains que le code fonctionne correctement.

Au niveau d'OpenGL, c'est basiquement le même mécanisme. La seule chose réellement différente par rapport à DirectX est la gestion des extensions, que nous allons devoir charger dynamiquement lors de l'initialisation.

```
void COGLRenderer::Initialize(HWND Hwnd)
{
    // Paramètres de rendu
    PIXELFORMATDESCRIPTOR PixelDescriptor =
    {
        sizeof(PIXELFORMATDESCRIPTOR), // size of this pfd
        1, // version number
        PFD_DRAW_TO_WINDOW | // support window
        PFD_SUPPORT_OPENGL | // support OpenGL
        PFD_DOUBLEBUFFER, // double buffered
        PFD_TYPE_RGBA, // RGBA type
        32, // 32-bit color depth
        0, 0, 0, 0, 0, 0, // color bits ignored
        0, // no alpha buffer
        0, // shift bit ignored
        0, // no accumulation buffer
        0, 0, 0, 0, // accum bits ignored
        32, // 32-bit z-buffer
        32, // 32-bits stencil buffer
        0, // no auxiliary buffer
        PFD_MAIN_PLANE, // main layer
        0, // reserved
        0, 0, 0 // layer masks ignored
    };

    // Récupération du Hwnd et du HDC de la fenêtre de rendu
    m_Hwnd = Hwnd;
    m_Handle = GetDC(Hwnd);

    // Choix du meilleur format de pixel
    if (!SetPixelFormat(m_Handle, ChoosePixelFormat(m_Handle, &PixelDescriptor),
    &PixelDescriptor))
        throw COGLException("SetPixelFormat", "Initialize");

    // Création du contexte de rendu
    m_Context = wglCreateContext(m_Handle);
    if (!wglMakeCurrent(m_Handle, m_Context))
        throw COGLException("wglMakeCurrent", "Initialize");

    // Récupération des extensions
    m_Extensions = reinterpret_cast<const char*>(glGetString(GL_EXTENSIONS));

    // Chargement des extensions OpenGL
    LoadExtensions();

    // States par défaut
    glClearColor(1.0f, 0.5f, 0.0f, 0.0f);
    glClearDepth(1.0f);
    glClearStencil(0x00);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
    glShadeModel(GL_SMOOTH);
}
}
```

Même remarque que pour DirectX : si ce code vous semble obscur, vous pourrez trouver les

explications nécessaires dans n'importe quel bon tutoriel d'introduction à OpenGL, par exemple chez [NeHe](#). La récupération dynamique des fonctions OpenGL étant plutôt laborieuse, nous allons utiliser notre ami le C++ pour la rendre un peu plus sympathique, à coups de macros et de templates. Voici donc à quoi ressemblerait notre fonction **LoadExtensions** et sa copine **CheckExtension** :

```
void COGLRenderer::LoadExtensions()
{
    // Vérification des extensions
    CHECK_EXTENSION("GL_ARB_vertex_buffer_object");
    CHECK_EXTENSION("GL_ARB_multitexture");
    // Etc...

    // Chargement des extensions
    LOAD_EXTENSION(glBindBufferARB);
    LOAD_EXTENSION(glGenBuffersARB);
    // Etc...
}

bool COGLRenderer::CheckExtension(const std::string& Extension) const
{
    return m_Extensions.find(Extension) != std::string::npos;
}
```

Et les macros / templates permettant d'écrire ce code sympathique :

```
// Chargement d'extension
template <class T> inline void LoadExtension(T& Proc, const char* Name)
{
    if (!(Proc = reinterpret_cast<T>(wglGetProcAddress(Name))))
        throw COGLException("wglGetProcAddress", "LoadExtension");
}
#define LOAD_EXTENSION(Ext) LoadExtension(Ext, #Ext)

// Vérification d'extension
#define CHECK_EXTENSION(Ext) if (!CheckExtension(Ext)) throw COGLException(#Ext, "CheckExtensions");
```

Ce genre de bidouille peut paraître sans importance, mais derrière le fait que nous allons économiser plusieurs lignes de code et des copier / coller, il y a plus important. En effet si nous avons dû nous passer de ces macros et templates, aurions-nous testé chaque retour de fonction et lancé l'exception appropriée en cas d'erreur ? N'aurions-nous pas fait une petite faute de frappe en recopiant tout ce code redondant ? Même pour des petits bouts de code comme ceux-ci, une bonne factorisation (et une bonne utilisation du C++) permet d'avoir un code plus sûr et plus riche.

Voilà, après l'appel à cette fonction d'initialisation, l'API est maintenant normalement prête pour afficher nos triangles. Mais pour le faire correctement cela ne sera pas suffisant : il va maintenant nous falloir aller trifouiller dans les matrices.

3.2 - Gestion des matrices

Nous n'avons pas encore abordé les matrices dans cette série d'articles, mais elles tiennent elles aussi une place très importante au sein de notre renderer, et dans la programmation 3D en général. En effet c'est au travers de ces matrices que nous allons faire subir à notre géométrie toutes les transformations dont elle aura besoin. Nous n'en avons pas encore parlé, mais il apparaît évident qu'il faudra une classe appropriée pour gérer ces matrices. Il nous faudra d'ailleurs toute une bibliothèque mathématique : matrices, vecteurs, quaternions, ... Nous manipulerons beaucoup de notions mathématiques tout au long de notre coding, ce sera donc un élément plus que primordial. Comme je

ne me sens pas l'âme d'un prof de maths, et qu'il n'y a guère plus à dire sur ces classes, je ne m'étends pas sur le sujet. J'ai simplement inclus leur code source intégral au zip que vous trouverez à la fin de cet article. Bien sûr si vous avez des interrogations ou des remarques quant à ces classes, elles seront les bienvenues. Si je reçois suffisamment de demande pour des explications plus poussées, je pourrais peut-être écrire un petit tuto sur le sujet, en attendant je vous renvoie à [la bible des matrices et des quaternions](#), elle contient tout ce dont vous aurez besoin !

Revenons donc à nos moutons. Globalement, que ce soit DirectX ou OpenGL, l'API 3D gère plusieurs matrices de transformations. Il y a plusieurs type de matrices :

- **La matrice de vue**, qui permet de placer la caméra dans notre scène.
- **La matrice de projection**, qui permet de projeter nos points 3D sur notre écran 2D.
- **Les matrices de texture** (rarement utilisées, surtout depuis l'arrivée des shaders) qui permettent divers contrôles sur nos coordonnées de texture.
- Par dessus ça, DirectX possède une matrice supplémentaire : **la matrice "monde"**, qui permet de placer un modèle dans l'espace 3D indépendamment de la caméra.

OpenGL fusionne cette dernière avec la matrice de vue (qu'il appelle d'ailleurs *model-view*), nous ferons donc de même dans notre moteur. La matrice monde de DirectX restera quant à elle toujours à l'identité, comme ça elle n'embêtera personne.

On pourra également remarquer qu'OpenGL fournit gracieusement des piles de matrices et leur gestion, ce que ne fait pas DirectX (cela a été ajouté récemment à la bibliothèque D3DX, mais tant qu'à gérer tout ça séparément, autant le faire nous-même). Nous allons donc nous inspirer d'OpenGL pour la gestion de nos matrices, il faudra : changer une matrice courante, récupérer une matrice courante, empiler une matrice (la sauvegarder), dépiler une matrice (la restaurer), et changer une matrice en la multipliant à la matrice courante (on pourrait le faire séparément, mais vu que ça arrive très souvent et qu'OpenGL fournit cette fonction, autant faire de même).

Voici donc ce que tout cela donne :

```
// Différents types de matrices de transformation
enum TMatrixType
{
    MAT_MODELVIEW,
    MAT_PROJECTION,
    MAT_TEXTURE_0,
    MAT_TEXTURE_1,
    MAT_TEXTURE_2,
    MAT_TEXTURE_3
};

class YES_EXPORT IRenderer
{
public :

    // ...

    // Empile la matrice courante
    virtual void PushMatrix(TMatrixType Type) = 0;

    // Dépile la matrice en haut de la pile
    virtual void PopMatrix(TMatrixType Type) = 0;

    // Charge une matrice
    virtual void LoadMatrix(TMatrixType Type, const CMatrix4& Matrix) = 0;

    // Charge une matrice en la multipliant avec la précédente
```

```
virtual void LoadMatrixMult(TMMatrixType Type, const CMatrix4& Matrix) = 0;  
// Récupère la matrice courante  
virtual void GetMatrix(TMMatrixType Type, CMatrix4& Matrix) const = 0;  
};
```

L'implémentation OpenGL est quasi-directe : on utilise simplement les fonctions correspondantes (*glMatrixMode*, *glPushMatrix*, *glPopMatrix*, *glLoadMatrixf*, *glMultMatrixf*). Pour DirectX ce n'est pas immédiat mais pas beaucoup plus compliqué : on utilise des **std::vector** de matrices en guise de piles (une pile pour chaque type de matrice), puis on les gère à coup de *push_back*, *pop_back*, *back*, etc... Si vous n'êtes pas familier avec **std::vector**, un petit tour par [la FAQ C++ de developpez.com](#) vous permettra d'y voir plus clair.

3.3 - Gestion des couleurs

Il était une fois au pays de l'API 3D, deux grands royaumes. D'un côté les Directiksses, de l'autre les Opengéhèles. Bien entendu, les rois de ces deux contrées étaient de farouches ennemis, et ils n'étaient jamais d'accord sur un même sujet. C'est ainsi que, le jour où ils durent définir le codage de leurs couleurs 32 bits, ils prirent des options différentes, le roi des Directiksses choisissant le ARGB et le roi des Opengéhèles prenant lui du ABGR. Pour notre plus grand bonheur ! Tout ça donc pour dire que selon l'API que nous utiliserons, le codage interne de nos couleurs ne sera pas le même. A chaque fois que nous allons remplir un vertex buffer ou autre, il nous faudra convertir notre couleur dans la bonne représentation interne. C'est fastidieux, mais bon... ! Pour le bien-être des habitants des deux royaumes, nous dotons donc notre renderer d'une fonction de conversion :

```
// Chez les Directiksses  
unsigned long CDX9Renderer::ConvertColor(const CColor& Color) const  
{  
    return Color.ToARGB();  
}  
  
// Chez les Opengéhèles  
unsigned long COGLRenderer::ConvertColor(const CColor& Color) const  
{  
    return Color.ToABGR();  
}
```

CColor est une classe facilitant la manipulation des couleurs, elle est incluse aux sources qui accompagnent cet article.

Cela peut paraître un détail (qui a tout de même son importance si l'on veut des résultats corrects !), mais c'est un exemple parfait du genre de piège dans lesquels on peut tomber lorsqu'on gère plusieurs API. Donc, gardez l'oeil !

4 - Les enums

Si vous êtes attentifs, peut-être aurez vous remarqué qu'une bonne partie du code sera redondant dans notre renderer. En effet on va manipuler beaucoup de types énumérés, de flags, de constantes, qu'il faudra à chaque fois faire correspondre aux valeurs de l'API. On va donc se retrouver avec des switches et des successions de if partout dans les fonctions du renderer, ce qui n'est pas la plus belle des façons de programmer. L'idée est donc d'aller encapsuler toutes ces correspondances dans une seule classe (pour chaque API), et d'utiliser un appel de fonction pour remplacer chaque switch / if de notre code. Imaginez cette situation :

```
Beurk
void COGLRenderer::PushMatrix(TMATRIX Type)
{
    // - Gestion des matrices de texture coupée ici pour aller à l'essentiel -
    switch(Type)
    {
        case MAT_MODELVIEW : glMatrixMode(GL_MODELVIEW); break;
        case MAT_PROJECTION : glMatrixMode(GL_PROJECTION); break;
        case ... : glMatrixMode(...); break;
    }
    glPushMatrix();
}

void COGLRenderer::PopMatrix(TMATRIX Type)
{
    // - Gestion des matrices de texture coupée ici pour aller à l'essentiel -
    switch(Type)
    {
        case MAT_MODELVIEW : glMatrixMode(GL_MODELVIEW); break;
        case MAT_PROJECTION : glMatrixMode(GL_PROJECTION); break;
        case ... : glMatrixMode(...); break;
    }
    glPopMatrix();
}
```

Imaginons maintenant la même fonction, mais en mettant en application notre brillante idée :

```
Miam
void COGLRenderer::PushMatrix(TMATRIX Type)
{
    // - Gestion des matrices de texture coupée ici pour aller à l'essentiel -
    glMatrixMode(COGLEnum::Get(Type));
    glPushMatrix();
}

void COGLRenderer::PopMatrix(TMATRIX Type)
{
    // - Gestion des matrices de texture coupée ici pour aller à l'essentiel -
    glMatrixMode(COGLEnum::Get(Type));
    glPopMatrix();
}
```

Nettement plus chouette n'est-ce pas ? Sachant que notre renderer aura à faire ce genre de manip de nombreuses fois, et parfois plusieurs fois avec le même type énuméré (comme ici pour les matrices), c'est un gain de temps et de lignes de code appréciable. Sans compter les fautes dûes aux mauvais copier / coller qui arrivent toujours dans ce genre de situation, que nous éviterons aussi. Pour finir en

beauté, nous allons même appeler toutes nos fonctions **Get** et utiliser la surcharge, comme ça nous n'aurons même pas à chercher quelle fonction appeler, il n'y aura qu'à faire `COGLEnum::Get(Machin)` pour avoir un truc ! Quand je vous le dis, que c'est chouette le C++.

Voyons maintenant ce que va contenir notre nouvelle classe **COGLEnum** (la même existe dans la version DirectX : **CDX9Enum**). Comme prévu, toute une tripotée de fonctions Get ainsi que les tableaux effectuant la conversion, le tout statique car créer des instances de `COGLEnum` n'aurait aucun intérêt ici.

```
class COGLEnum
{
public :

    // Type des matrices
    static unsigned long Get(TMatrixType Value);

    // Opérations sur les unités de texture
    static unsigned long Get(TTextureOp Value);

    // Opérateurs des unités de texture
    static unsigned long Get(TTextureArg Value);

    // Etc...

private :

    // Données membres
    static unsigned long MatrixType[];
    static unsigned long TextureOp[];
    static unsigned long TextureArg[];
    // Etc...
};
```

Libre à vous d'ajouter à cette classe tout ce qui vous semblera pratique, n'oubliez pas que le but est au final de se fatiguer le moins possible.

Vient ensuite le remplissage des tableaux, de manière à faire correspondre les valeurs du moteur aux valeurs de l'API :

```
unsigned long COGLEnum::MatrixType[] =
{
    GL_MODELVIEW,
    GL_PROJECTION,
    GL_TEXTURE,
    GL_TEXTURE,
    GL_TEXTURE,
    GL_TEXTURE
};
unsigned long COGLEnum::TextureOp[] =
{
    GL_REPLACE,
    GL_ADD,
    GL_MODULATE,
    GL_REPLACE,
    GL_ADD,
    GL_MODULATE
};
unsigned long COGLEnum::TextureArg[] =
{
    GL_PRIMARY_COLOR_EXT,
    GL_TEXTURE,
    GL_PREVIOUS_EXT,
    GL_CONSTANT_EXT
};
```

Puis enfin l'implantation des fonctions Get, qui ne font rien d'autre que renvoyer le bon élément :

```
unsigned long COGLEnum::Get(TMATRIXType Value)
{
    return MatrixType[Value];
}

unsigned long COGLEnum::Get(TTextureOp Value)
{
    return TextureOp[Value];
}

unsigned long COGLEnum::Get(TTextureArg Value)
{
    return TextureArg[Value];
}
```

Et voilà les loulous !

5 - Conclusion

Nous avons vu dans cette partie quels mécanismes utiliser pour encapsuler nos API 3D, et comment en changer à la volée. Nous avons également vu les bases du renderer, qui sera véritablement le coeur du rendu 3D. Nous sommes à présent capables d'effectuer un rendu indépendamment de l'API choisie, nous allons pouvoir empiler les briques suivantes et progressivement donner de plus en plus de moyens à notre moteur 3D. La prochaine étape nous permettra notamment d'afficher nos premiers polygones et construire nos modèles, via le codage des vertex et index buffers. Nous améliorerons également notre renderer, qui devra être bien plus riche pour remplir parfaitement son rôle. Nous devons lui ajouter toute sorte de fonctions, pour gérer toute sorte de paramètres et effets relatifs à l'API 3D. On aura par exemple la gestion du viewport, des states, des unités de texture (en gros, du FFP), également des choses moins importantes du genre récupérer le backbuffer pour faire une capture d'écran, etc... Mais ce ne sera pas le plus dur : une fois qu'on aura identifié une fonction qui nous semblera nécessaire, il suffira de l'ajouter à l'interface de IRenderer et de coder l'implémentation correspondante pour chaque API. En plus ces fonctions se limiteront la plupart du temps à quelques lignes (appel de la bonne fonction de l'API concernée), ce sera donc du gâteau. Miam !

Encore un mot avant de terminer : on ne le repètera jamais assez, faites attention aux différences (parfois on se demande s'ils le font exprès) entre les diverses API que vous comptez gérer, en l'occurrence DirectX et OpenGL ici. Cela peut mener à de vrais casse-têtes de conception, et malheureusement la plupart du temps à une refonte plus ou moins importante du code. Ne négligez pas l'analyse !

6 - Téléchargements

Les sources fournies dans les précédents tutoriels sont entièrement intégrées à celles-ci, de manière à ce que vous ayez toujours un package complet et à jour.

Les codes source livrés tout au long de cette série d'articles ont été réalisés sous Visual Studio.NET 2003. Aucun test n'a été effectué sur d'autres compilateurs, mais vous ne devriez pas rencontrer de problème si vous possédez un compilateur récent, le code respectant autant que possible le standard.

[Télécharger les sources de cet article \(zip, 74 Ko\)](#)

Une version PDF de cet article est disponible : [Télécharger \(81 Ko\)](#)

Si vous avez des suggestions, remarques, critiques, si vous avez remarqué une erreur, ou bien si vous souhaitez des informations complémentaires, n'hésitez pas à me contacter !

7 - Remerciements

Un grand merci à toutes les personnes qui m'aident, de près ou de loin, à réaliser ou améliorer cette série d'article : Stoomm, Nico, Vincent, toute l'équipe de developpez.com, les lecteurs bien sûr, et enfin Mathieu pour sa grande contribution.