CRÉATION D'UNE FENÊTRE ET D'UN RENDU SUR LA SDL 2

Introduction

Dans cette partie, nous allons parler plus de code. En effet dans le chapitre précédent nous avons vu, comment configurer un projet SDL, en ligne de commande, mais sans forcément comprendre le code que nous avions compilé.

> **Information**
>
> La plupart des notions/fonctionnalités que j'expliquerai ne sortent pas de mon chapeau magique , sachez qu'il est bon d'aller **regarder la documentation**, et c'est ce à quoi je vais vous amener d'ici à la fin de ce chapitre.

Je vous remets ci-dessous le code que vous avez précédemment compilé et juste aprés décortiquons-le !

```c
#include <SDL.h>

int main(int argc, char* argv[])
{
    SDL_Init(SDL_INIT_VIDEO);
    SDL_Quit();
    return 0;
}
```

```c
#include <SDL.h>
```

Au tout début du programme j'inclus ma bibliothèque SDL, pour par la suite pouvoir utiliser les fonctionnalités qu'elle propose.

```
int main(int argc, char** argv) ou int main(int argc, char* argv[])
```

Vous voyez que la signature de la fonction `main()`, est plus tôt assez longue, car en effet la SDL a besoin qu'on utilise cette signature la spécifiquement.

Initialisation de la SDL

Pour initialiser la SDL, il faut d'abord connaître le **module** de la bibliothèque qu'on souhaite utiliser.

Pour cela nous avons la fonction `SDL_Init()` cette fonction va s'occuper de nous charger, en mémoire les modules nécessaires pour le fonctionnement de la bibliothèque.

Voici sa signature :

```
int SDL_Init(Uint32 flags)
```

Elle retourne une valeur inférieure à 0 en cas d'erreur sinon elle retourne 0 si tout se passe bien.

Les flags symbolisent ici les modules de la SDL. Voici ci-dessous la liste des différents modules existants :

SDL_INIT_TIMER	timer subsystem
SDL_INIT_AUDIO	audio subsystem
SDL_INIT_VIDEO	video subsystem; automatically initializes the events subsystem
SDL_INIT_JOYSTICK	joystick subsystem; automatically initializes the events subsystem
SDL_INIT_HAPTIC	haptic (force feedback) subsystem
SDL_INIT_GAMECONTROLLER	controller subsystem; automatically initializes the joystick subsystem
SDL_INIT_EVENTS	events subsystem
SDL_INIT_EVERYTHING	all of the above subsystems
SDL_INIT_NOPARACHUTE	compatibility; this flag is ignored

On aura l'occasion sur d'autres modules du cours, de voir plus en détails ces modules.

Information

Si on souhaite par exemple utiliser le module VIDEO, alors on appellera la fonction `SDL_Init` de cette manière :

```
SDL_Init(SDL_INIT_VIDEO)
```

Il peut être parfois nécessaire d'utiliser plusieurs modules. Pour se faire les modules SDL peuvent être assemblés en utilisant l'opérateur `|` à ne pas confondre avec l'opérateur `||`.

Information

Pour information l'opérateur `|` pratique un "ou" bit a bit.

Par exemple pour activer le module AUDIO cumulé avec le module VIDEO, on aura le code suivant :

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO)
```

Après le bon déroulement de la fonction `SDL_Init()`, on ferra en sorte d'appeler directement `SDL_Quit()`, cette fonction nous permet de **libérer en mémoire les ressources** utilisées par la SDL.

Voici sa signature :

```
void SDL_Quit(void)
```

Elle reste quand même très simple à utiliser .

Donc si vous avez suivi jusqu'ici, vous aurez :

```
#include <SDL.h>

int main(int argc, char* argv[])
{
    SDL_Init(SDL_INIT_VIDEO);
    SDL_Quit();
    return 0;
}
```

Mais nous pouvons encore plus améliorer notre code, on gérant le cas ou `SDL_Init()` n'a pas réussi à fonctionner, d'où l'importance de vérifier le retour de la fonction avec un `if`.

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

int main(int argc, char* argv[])
{
    if(SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        return -1;
    }
    SDL_Quit();
    return 0;
}
```

Vous pouvez même inclure `cstdlib` pour remplacer les valeurs de retour par les constantes `EXIT_FAILURE` et `EXIT_SUCCESS`.

Dans ce `if` j'appelle une fonction `SDL_LogError()` qui va écrire dans le flux d'erreur. Le premier paramètre est la catégorie du message d'erreur. Voici une liste des différentes catégories :

SDL_LOG_CATEGORY_APPLICATION	application log
SDL_LOG_CATEGORY_ERROR	error log
SDL_LOG_CATEGORY_ASSERT	assert log
SDL_LOG_CATEGORY_SYSTEM	system log
SDL_LOG_CATEGORY_AUDIO	audio log
SDL_LOG_CATEGORY_VIDEO	video log
SDL_LOG_CATEGORY_RENDER	render log
SDL_LOG_CATEGORY_INPUT	input log
SDL_LOG_CATEGORY_TEST	test log
SDL_LOG_CATEGORY_RESERVED#	# = 1-10; reserved for future SDL library use
SDL_LOG_CATEGORY_CUSTOM	reserved for application use; see Remarks for details

`SDL_LogError()` n'est pas la seule fonction qui existe pour afficher les erreurs, il en existe beaucoup d'autres et qui fonctionnent pareil à quelque chose prêt.

Voici la documentation de `SDL_GetError()` : https://wiki.libsdl.org/SDL_GetError

Si vous allez en bas de la page vous trouverez des noms de fonctions en lien avec ce que `SDL_LogError()` fait, je vous laisse visiter la doc et voir ce qu'il est possible de faire en matière d'affichage d'erreurs.

Dans le `if` je retourne –1 pour dire que la fonction `main()` a échoué, sinon comme j'ai dit n'hésitez pas à un inclure la bibliothèque `cstdlib` pour utiliser les constant `EXIT_FAILURE` et `EXIT_SUCCESS`.

Création d'une fenêtre et du rendu

Création de la fenêtre

Après avoir initialisé la SDL, il est temps maintenant de **créer notre première fenêtre** ainsi que le rendu pour cette fenêtre.

Pour créer une fenêtre il existe la fonction `SDL_CreateWindow()` et pour créer le rendu de fenêtre il existe la fonction `SDL_CreateRenderer()`, vous remarquerez que toute fonction SDL commence par le mot `SDL_` , ceci est dû car la bibliothèque est écrite en langage C et qu'il a fallu faire en sorte d'éviter des conflits entre d'autres bibliothèques ayant le même nom de fonction. Ce type de problème est résolu avec l'arrivée des **namespaces** (espace de nom) dans les langages de programmation les plus modernes.

D'abord nous devons créer notre fenêtre en utilisant le prototype suivant :

```
SDL_Window* SDL_CreateWindow(const char* title, int x, int y, int w, int h, Uint32 fl
```

- Le premier paramètre est le titre de la fenêtre
- Le deuxième paramètre est la position x (horizontal) de la fenêtre
- Le troisième paramètre est la position y (vertical) de la fenêtre
- Le quatrième paramètre est la largeur de la fenêtre
- Le cinquième paramètre est la hauteur de la fenêtre
- Le sixième paramètre prend comme valeur un flag

Elle nous retourne une structure `SDL_Window*` et dans le cas contraire elle nous retourne un `nullptr`.

Pour le deuxième et troisième paramètres correspondant au positionnement de la fenêtre, il faut savoir qu'il existe trois façons pour placer sa fenêtre, soit par des coordonnées bien précisées (x, y), soit on laisse la SDL choisir pour ou sinon on peut aussi placer sa fenêtre au centre de mon écran.

Pour centrer ou laisser la SDL choisir la position pour nous, il faut utiliser les flags suivants :

- **SDL_WINDOWPOS_CENTERED** : au centre de l'écran
- **SDL_WINDOWPOS_UNDEFINED** : on laisse SDL choisir

Pour le dernier paramètre, voici la liste des flags possible :

SDL_WINDOW_FULLSCREEN	fullscreen window
SDL_WINDOW_FULLSCREEN_DESKTOP	fullscreen window at the current desktop resolution
SDL_WINDOW_OPENGL	window usable with OpenGL context
SDL_WINDOW_SHOWN	window is visible
SDL_WINDOW_HIDDEN	window is not visible
SDL_WINDOW_BORDERLESS	no window decoration
SDL_WINDOW_RESIZABLE	window can be resized
SDL_WINDOW_MINIMIZED	window is minimized
SDL_WINDOW_MAXIMIZED	window is maximized
SDL_WINDOW_INPUT_GRABBED	window has grabbed input focus
SDL_WINDOW_INPUT_FOCUS	window has input focus
SDL_WINDOW_MOUSE_FOCUS	window has mouse focus
SDL_WINDOW_FOREIGN	window not created by SDL
SDL_WINDOW_ALLOW_HIGHDPI	window should be created in high-DPI mode if supported (>= SDL 2.0.1)
SDL_WINDOW_MOUSE_CAPTURE	window has mouse captured (unrelated to INPUT_GRABBED, >= SDL 2.0.4)
SDL_WINDOW_ALWAYS_ON_TOP	window should always be above others (X11 only, >= SDL 2.0.5)
SDL_WINDOW_SKIP_TASKBAR	window should not be added to the taskbar (X11 only, >= SDL 2.0.5)
SDL_WINDOW_UTILITY	window should be treated as a utility window (X11 only, >= SDL 2.0.5)
SDL_WINDOW_TOOLTIP	window should be treated as a tooltip (X11 only, >= SDL 2.0.5)
SDL_WINDOW_POPUP_MENU	window should be treated as a popup menu (X11 only, >= SDL 2.0.5)

Prenons par exemple la demande suivante :

- Titre de la fenêtre : SDL Programme
- Position de la fenêtre : Centrer
- Dimension de la fenêtre : 800x600
- Flags de la fenêtre : Fenêtre Visible

Avant de vous dévoiler le code il est important de libérer votre fenêtre de la mémoire, pour cela il y a la fonction `SDL_DestroyWindow()`.

Voici la signature de la fonction `SDL_DestroyWindow()` :

```
void SDL_DestroyWindow(SDL_Window* window)
```

Elle prend en paramètre, un pointeur sur une structure SDL_Window correspondant à la fenêtre qu'on a créée.

Pour répondre à notre demande citée plus haut, on utilisera le code suivant :

```
#include <SDL.h>
#include <cstdlib>

int main(int argc, char* argv[])
{
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        return EXIT_FAILURE;
    }
    SDL_Window* pWindow{ nullptr };
    pWindow = SDL_CreateWindow("SDL Programme", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS
    if (pWindow == nullptr)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }
    SDL_DestroyWindow(pWindow);
```

```
    SDL_Quit();
    return EXIT_SUCCESS;
}
```

Création du rendu

À ce stade on a juste créé une fenêtre, maintenant il faut créer le rendu de la fenêtre, ce rendu aura pour type `SDL_Renderer*`, pour se faire nous aurons besoin de la fonction `SDL_CreateRenderer()`

Voici sa signature :

```
SDL_Renderer* SDL_CreateRenderer(SDL_Window* window, int index, Uint32 flags)
```

- Le premier paramètre correspond à notre fenêtre.
- Le deuxième paramètre est l'index du pilote à initialiser selon le flag demander en troisième paramètre
- Le troisième paramètre est un/des flag(s)

Pour, le deuxième paramètre, mettre -1 permet de laisser la SDL choisir pour vous le bon **pilote de la carte graphique**.

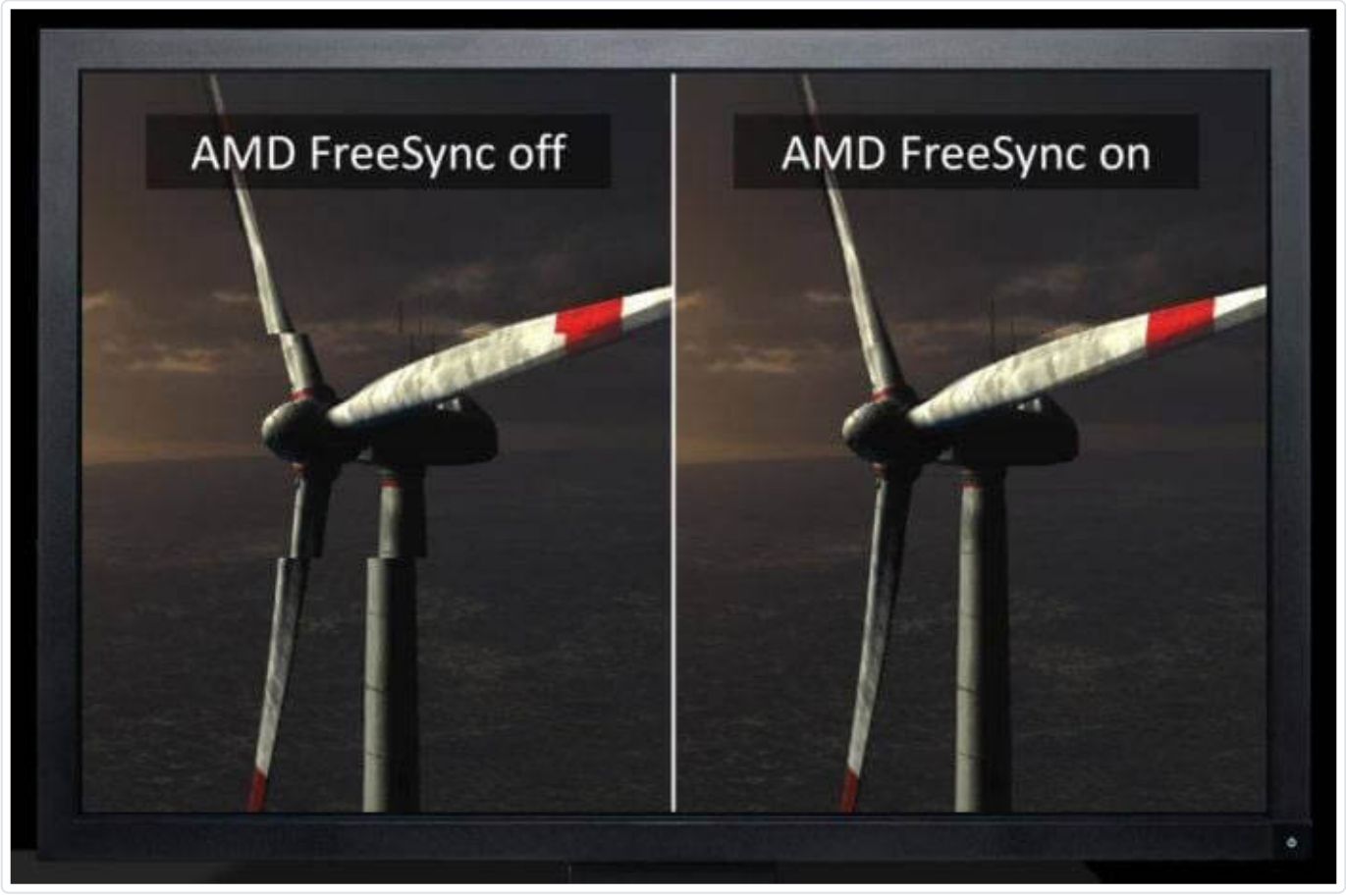
Pour le troisième paramètre, voici la liste des différents flags :

- `SDL_RENDERER_SOFTWARE` : utilise l'accélération logiciel pour faire les calculs de **rendu depuis le CPU**
- `SDL_RENDERER_ACCELERATED` : utilise l'accélération matériel pour faire les calculs de **rendu depuis la GPU**
- `SDL_RENDERER_PRESENTVSYNC` : synchronise l'affichage en fonction du taux de rafraîchissement de votre écran

- Exemple si j'ai un écran de 120 Hertz cela veut dire qu'il est capable d'afficher que 120 images par second or si ma carte graphique peut gérer 300 images par seconde alors j'aurai un effet de déchirure sur mon écran. D'où l'intérêt de cette option, qui va nous permettre d'adapter les calculs de la carte graphique à celle du taux de rafraîchissement
- `SDL_RENDER_TARGETTEXTURE` : autorise-le rendu sur une SDL_Texture (on verra plus en profondeur cette notion dans un chapitre dédié aux textures)

Information

Préférez utiliser le rendu calculé par la carte graphique (tag `SDL_RENDERER_ACCELERATED`) , car la carte graphique est faites pour ça !



La fonction `SDL_CreateRenderer()` retourne un pointeur `SDL_Renderer*` ou un `nullptr` si elle échoue.

Une fois qu'on aura réussi à créer le `SDL_Renderer*`, il ne faudra pas oublier de le libérer de la mémoire comme pour la fenêtre avec la fonction `SDL_DestroyRenderer()`.

Voici sa signature :

```
void SDL_DestroyRenderer(SDL_Renderer* renderer)
```

Si on reprend tout ce qu'on a pu apprendre jusqu'ici, alors on obtiendra le résultat suivant :

```
#include <SDL.h>
#include <cstdlib>

int main(int argc, char* argv[])
{
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        return EXIT_FAILURE;
    }
    SDL_Window* pWindow{ nullptr };
    SDL_Renderer* pRenderer{ nullptr };
    pWindow = SDL_CreateWindow("SDL Programme", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS
    if (pWindow == nullptr)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }
    pRenderer = SDL_CreateRenderer(pWindow, -1, SDL_RENDERER_ACCELERATED);
    if (pRenderer == nullptr)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }
    SDL_DestroyRenderer(pRenderer);     SDL_DestroyWindow(pWindow);
    SDL_Quit();
    return EXIT_SUCCESS;
```

```
}
```

Vous constatez que la création d'une fenêtre et d'un rendu pour la fenêtre se fait toujours de la même façon, mais sachez qu'il est possible que l'on puisse faire d'une pierre deux coups avec la fonction `SDL_CreateWindowAndRenderer()`. Cette fonction permet de créer à la fois la fenêtre et à la fois le rendu de fenêtre en même temps en appelant qu'une seule fonction .

Voici sa signature :

```
int SDL_CreateWindowAndRenderer(int width, int height, Uint32 window_flags, SDL_Windo
```

- Le premier paramètre est la largeur de la fenêtre
- Le deuxième paramètre est la hauteur de la fenêtre
- Le troisième paramètre est le flag de fenêtre
- Le quatrième paramètre est l'adresse d'un pointeur de SDL_Window
- Le cinquième paramètre est l'adresse d'un pointeur de SDL_Renderer

Elle retourne 0 si elle a réussi ou -1 si elle a échoué

Notre code peut se résumer simplement par ceci :

```
#include <SDL.h>
#include <cstdlib>

int main(int argc, char* argv[])
{
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        return EXIT_FAILURE;
    }
    SDL_Window* pWindow{ nullptr }; // ma fenêtre
    SDL_Renderer* pRenderer{ nullptr }; // mon rendu
    if (SDL_CreateWindowAndRenderer(800, 600, SDL_WINDOW_SHOWN, &pWindow, &pRenderer)
```

```
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }
    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    SDL_Quit();
    return EXIT_SUCCESS;
}
```

Pour les plus curieux d'entre vous, vous aurez peut-être remarqué qu'avec la fonction `SDL_CreateWindowAndRenderer()`, vous ne pouvez pas donner de titre à votre fenêtre. Vous avez raison, mais heureusement que la SDL a pensé à nous, en nous fournissant une fonction nommée `SDL_SetWindowTitle`, permettant d'assigner un titre à votre fenêtre, ainsi si vous le souhaitez vous pouvez changer le titre à volonté.

Voici son prototype :

```
const char* SDL_GetWindowTitle(SDL_Window* window)
```

Reprenons notre code avec cette nouvelle fonction

```
#include <SDL.h>
#include <cstdlib>

int main(int argc, char* argv[])
{
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        return EXIT_FAILURE;
    }
    SDL_Window* pWindow{ nullptr };
    SDL_Renderer* pRenderer{ nullptr };
    if (SDL_CreateWindowAndRenderer(800, 600, SDL_WINDOW_SHOWN, &pWindow, &pRenderer)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }
    SDL_SetWindowTitle(pWindow, "Hello !");
```

```
    SDL_Delay(1000); // mettre en pause pendant 1 seconde le rendu
    SDL_SetWindowTitle(pWindow, "World !");
    SDL_Delay(1000);
    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    SDL_Quit();
    return EXIT_SUCCESS;
}
```

La fonction `SDL_Delay()` permet de **mettre en pause votre rendu**, elle prend comme paramètre des millisecondes.

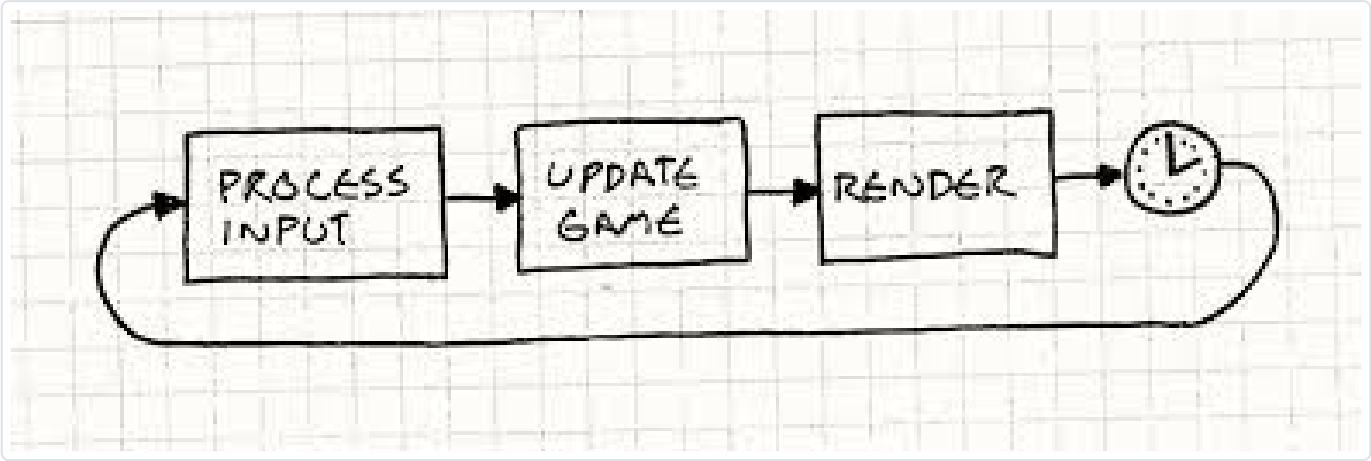
Voici son prototype :

```
void SDL_Delay(Uint32 ms)
```

GameLoop

Concept

Maintenant il vous faut que je vous parle de quelque chose d'indispensable. Plus communément la **GameLoop**, normalement j'ai prévu de parler de cette notion dans un autre article consacré au développement de jeux vidéo sous SDL mais en réalité je ne peux pas passer à côté, comme ça vous prenez dès le début du cours les bonnes habitudes.



Cette image représente bien ce qu'est une GameLoop. Si vous découvrez cette notion pour la première fois alors il est temps de démystifier cela .

1. **Process input** : ici on parle d'événements, clavier, souris, joystick. C'est ici qu'on capture les entrées utilisateurs
2. **Update Game** : c'est la mise à jour de tous ce qui compose votre jeu, en effet, lors de l'étape précédente [Process Input], vous avez par exemple appuyé sur la touche [FLECHE DE HAUT], et il doit y avoir un effet à comme par exemple déplacer le joueur vers le haut, dans ce cas le déplacement est géré par l'étape Update Game
3. **Render** : c'est la phase où on va dessiner tout ce qui compose notre scène 2D, par exemple la position du joueur par son sprite (animation) qui le représente
4. **Chronomètre**, ici est récupéré le temps d'exécution de la GameLoop. Cette manipulation permet généralement la **limitation des FPS (image par seconde)**

Nous allons voir ici très sommairement l'étape Process Input car j'ai réservé un chapitre dédié entièrement aux captures d'évènements. Nous allons juste nous contenter de pouvoir fermer la fenêtre en cliquant sur la croix en haut à droite.

Boucle du jeu

Dans un premier temps, il faut un moyen pour récupérer les évènements clavier, souris. Pour ça la SDL prévoit une une union `SDL_Event`, cette structure contiendra tous les évènements liés à nos périphériques.

Elle n'a pas besoin de fonction pour l'initialiser, il suffit juste de déclarer une union de type `SDL_Event` comme si c'était un type primitif du C++.

Avant de gérer les évènements, j'ai besoin d'abord de créer une **boucle principale** pour empêcher notre jeu de s'arrêter, avec un booléen qui me dira si je dois **quitter mon jeu** ou non. Voici donc à quoi va ressembler notre boucle :

```
SDL_Event events;
bool isOpen{ true };
while (isOpen)
{
    /* évènements de votre jeu */
}
```

Si vous compilez, vous aurez une boucle infinie car on ne gère pas le cas où on clique sur la croix rouge, qui mettra logiquement la variable `isOpen` a `false`.

Maintenant j'ai besoin de pouvoir quitter l'application quand je clique sur la croix rouge en fermant la fenêtre jeu. Pour cela je dois utiliser, la fonction `SDL_PollEvent()` pour lire d'éventuels événements clavier.

Voici sa signature :

```
int SDL_PollEvent(SDL_Event* event)
```

- Elle prend un unique paramètre, c'est l'adresse de l'union `SDL_Event`
- Elle retourne, 1 s'il y a un événement, et 0 s'il n'y en a aucun

Voici ce que vous devriez avoir :

```
SDL_Event events;
bool isOpen{ true };
while (isOpen)
{
    while (SDL_PollEvent(&events))
    {

    }
}
```

Nous devons maintenant, regarder quel type d'événements, on a pour cela l'union `SDL_Event` qui a un champ nommé `type` qui est un entier de type 32 bits, et permet de savoir quel type d'événements on capture. L'événement "clique sur la croix" est l'événement de type `SDL_QUIT`. Il suffit donc de vérifier que le champ "type" du `SDL_Event` vaut `SDL_QUIT`.

Voici le résultat final :

```
SDL_Event events;
bool isOpen{ true };
while (isOpen)
{
    while (SDL_PollEvent(&events))
    {
        switch (events.type)
        {
        case:
            SDL_QUIT:isOpen = false;
            break;
        }
    }
}
```

Bonus : Jouons avec notre fenêtre

À ce moment de l'article, vous avez une fenêtre qui peut-être fermée en cliquant sur la croix, Youpi !

Maintenant, discutons plus à propos, des choses qu'on peut faire, avec cette fenêtre ! Comme vu plus haut, on peut, changer le titre de la fenêtre avec la fonction `SDL_SetWindowTitle()`. Il existe aussi la fonction `SDL_GetWindowTitle()`, permettant de récupérer le titre.

Voici sa signature :

```
const char* SDL_GetWindowTitle(SDL_Window* window)
```

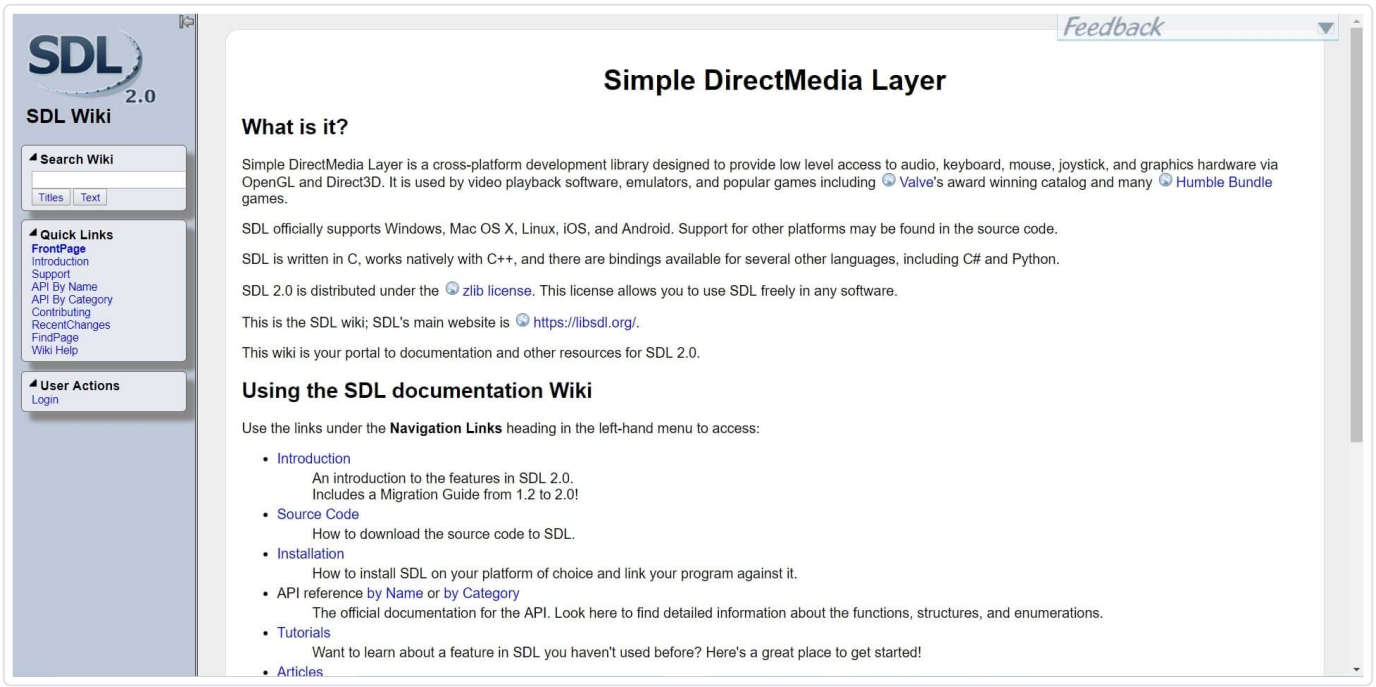
- Elle prend en paramètre un pointeur sur la fenêtre donc ici se sera `pWindow` en toute logique
- Elle retourne, le titre de la fenêtre, de type pointeur sur un type char

Son utilisation est assez simple :

```
const char* title = SDL_GetWindowTitle(pWindow);SDL_Log("Le titre de la fenêtre est %
```

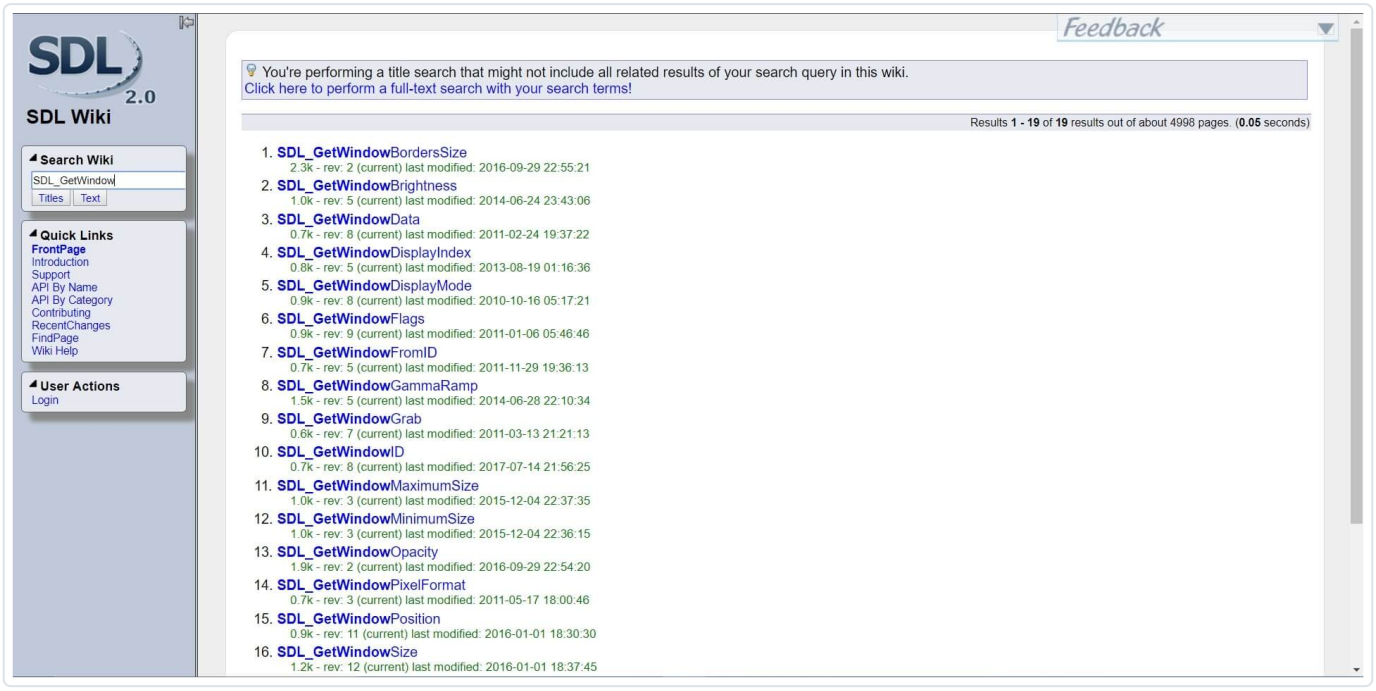
On remarque aisément que la SDL est bien faites ! Pour récupérer/changer un élement par rapport à la fenêtre, on obtient ce **pattern** : `SDL_SetWindowXXXX` et `SDL_GetWindowXXXX`

Nous pouvons aller vérifier dans la documentation ce qui existe déjà ! Pour cela, allez sur le site web de la SDL, et allez dans la documentation ! (Voici le lien : https://wiki.libsdl.org/)



Voici la page d'accueil de la SDL, maintenant on constate à gauche Search Wiki, il suffit de taper : "SDL_GetWindow" et vous retrouverez toutes les fonctions commençant par SDL_GetWindow, comme pour notre fonctionnalités `SDL_GetWindowTitle()` !

Voici ce que vous devriez obtenir en cherchant dans la documentation !



Bien entendu, il existe la même chose pour la structure SDL_Renderer !

Pour vous entraîner, je vous laisse regarder ce qui existe déjà, vous verrez c'est vraiment très simple !

Conclusion

Résumons, pour pouvoir, faire un programme SDL viable, il a fallu initialiser les modules de la SDL selon nos besoins, puis ensuite il a fallu créer une fenêtre et par la suite créer le contexte de rendu pour la fenêtre et enfin gérer la GameLoop.

Le code de fin donne :

```
#include <SDL2/SDL.h>
#include <cstdlib>

int main(int argc, char* argv[])
{
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        return EXIT_FAILURE;
    }
    SDL_Window* pWindow{ nullptr };
    SDL_Renderer* pRenderer{ nullptr };
    if (SDL_CreateWindowAndRenderer(800, 600, SDL_WINDOW_SHOWN, &pWindow, &pRenderer)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }
    SDL_Event events;
    bool isOpen{ true };
    while (isOpen)
    {
        while (SDL_PollEvent(&events))
        {
            switch (events.type)
            {
                case SDL_QUIT:
                    isOpen = false;
                    break;
            }
        }
    }
    SDL_DestroyRenderer(pRenderer);SDL_DestroyWindow(pWindow);
    SDL_Quit();
    return EXIT_SUCCESS;
}
```

On va se trouver pour le chapitre suivant qui sera plus sympa selon moi, car nous allons étudier l'affichage graphique avec la SDL !