

BOUCLE DE JEU

Création du projet	2
La boucle de jeu	5
Librairie statique	8
SDL	11
Initialisation de SDL	12
Fermeture de SDL	13
Création d'une fenêtre	14
Rendu dans la fenêtre	15
Traitement de messages	16
Gestion du temps	18

La partie théorique de l'examen s'appuie quasiment entièrement sur les chapitres à lire dans le livre « *Game Programming Patterns* » de Robert Nystrom.

Certains sujets ne seront **pas directement abordés** dans le cours et c'est à vous de lire les chapitres et de poser des questions au besoin.

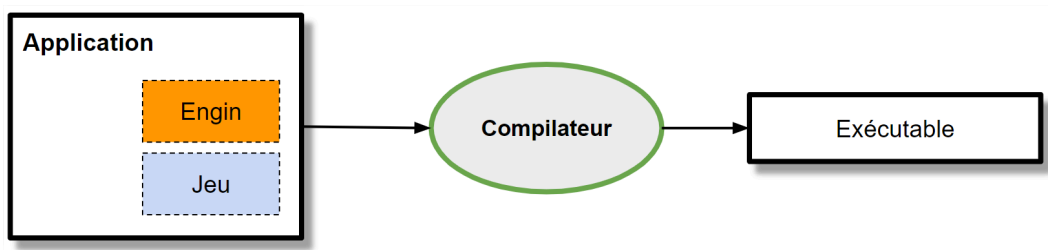
Lecture :

Robert Nystrom, Game Programming Patterns, Game Loop
<https://gameprogrammingpatterns.com/game-loop.html>

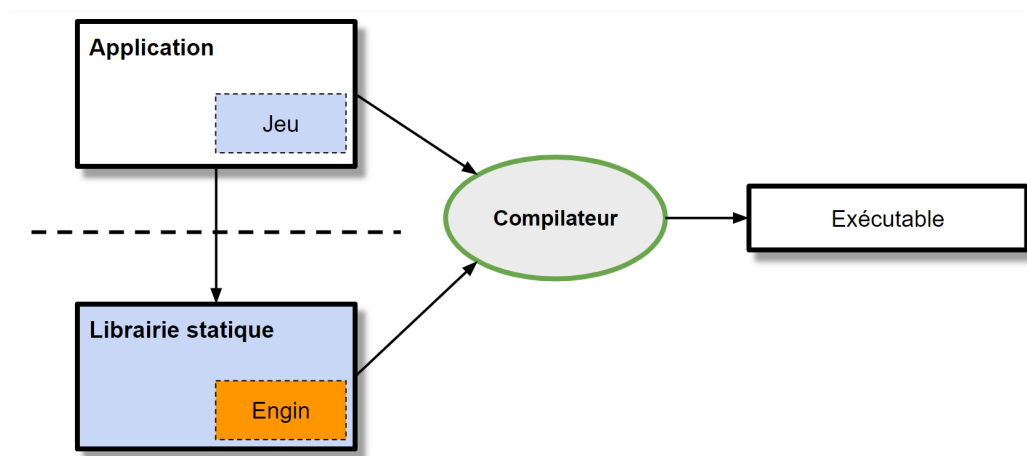
Création du projet

Nous n'allons pas coder un engin complet avec des outils, un éditeur et un langage de « scripting ». Nous n'aurions pas suffisamment de temps pour le faire. Mais, nous allons, un peu comme id-Software a dû le faire à l'époque, faire notre jeu en gardant une séparation de code engin/jeu la plus rigoureuse possible. Comme si vous alliez réutiliser le code « engin » pour faire d'autres jeux après vos études.

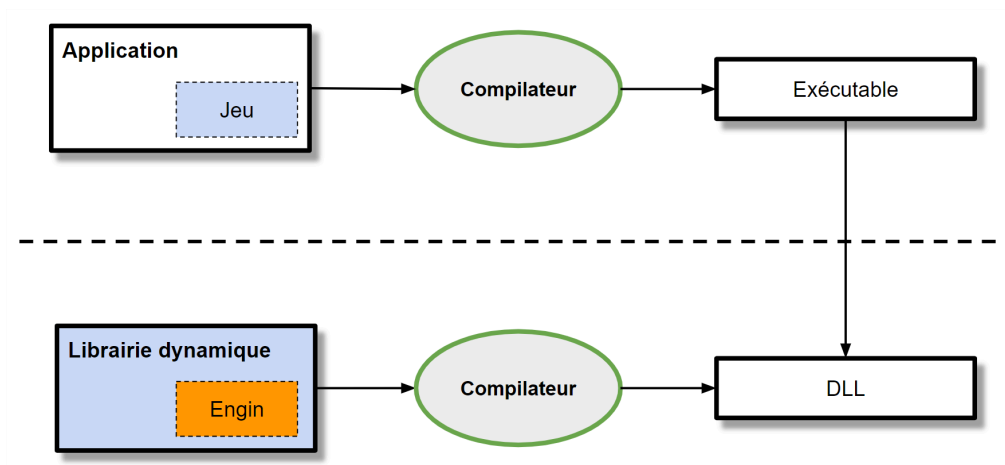
Lorsque vous faites un programme (contenant « engin » et « jeu »), tout votre code est traduit dans un exécutable par le compilateur.



En C++, une librairie statique (extension *.a* ou *.lib*) est un fichier contenant des fonctions qui seront intégrées à l'exécutable après la compilation.



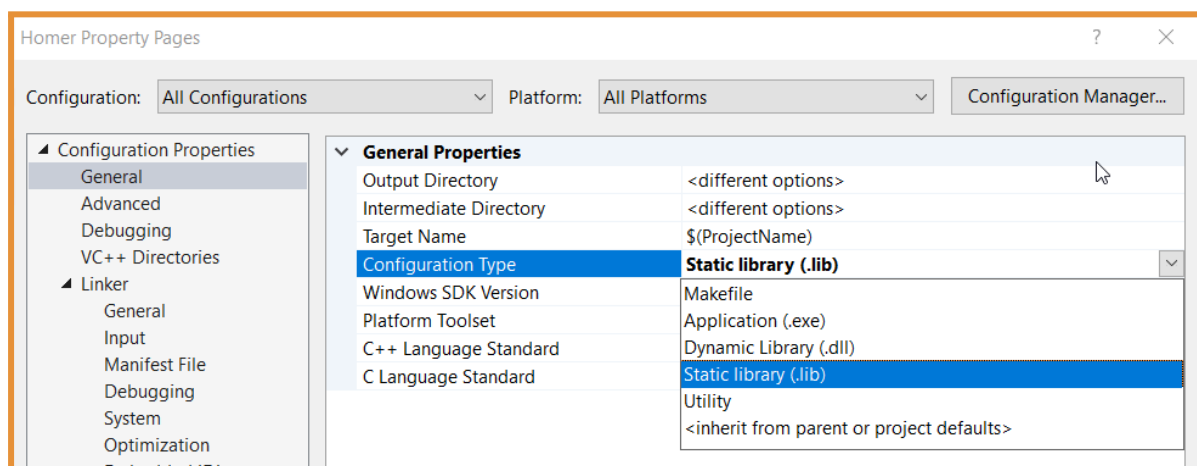
À l'inverse une librairie dynamique (extension .dll) est un fichier contenant des fonctions qui ne seront pas intégrées à l'exécutable, mais plutôt utilisées à l'exécution.



Pour favoriser la réutilisabilité et la séparation de code, nous allons faire une librairie. Une librairie statique nous donnera une certaine flexibilité dans le code lorsque vous aurez à le déboguer. La DLL est compilée et plus difficile à consulter lors de la recherche de bogues.

🔧 Dans Visual Studio, **solution vide** (Create new project > Blank Solution). Dans mon projet, j'ai baptisé mon engin : « Homer ». Ajoutez ensuite un projet vide (Empty project) qui contiendra le code de l'engin (J'ai nommé ce projet « TheEngine » dans mes notes).

Dans les propriétés du projet, nous devons spécifier le type de configuration. Choisissez « Static library (.lib) » pour faire notre librairie réutilisable d'un jeu à l'autre.

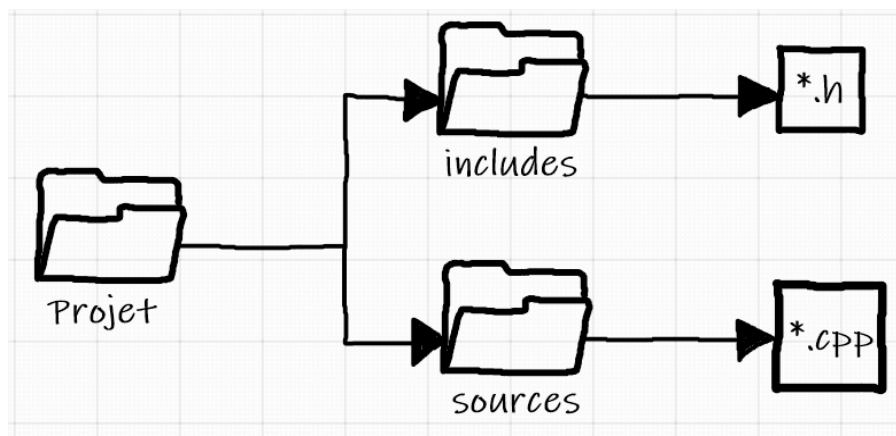


Conseil : pour éviter toute confusion, enlevez la configuration x86 et ne gardez que la x64 dans les options de « Configuration Manager ».

- 🔧 Ajoutez un fichier *Engine.h* dans un répertoire « *includes* » ;
- 🔧 Ajoutez un fichier *Engine.cpp* dans un répertoire « *sources* » ;
- 🔧 Dans Visual Studio, remplacez les **filtres** pour ces noms de répertoires.

Note : les *filtres* de Visual Studio, même s'ils portent le même nom que les répertoires, ne sont pas des répertoires, ce ne sont que des *filtres* pour Visual Studio.

Lorsque nous partageons notre librairie statique, nous devons aussi fournir les fichiers d'entête. Une organisation simple de répertoires simplifiera cette tâche.



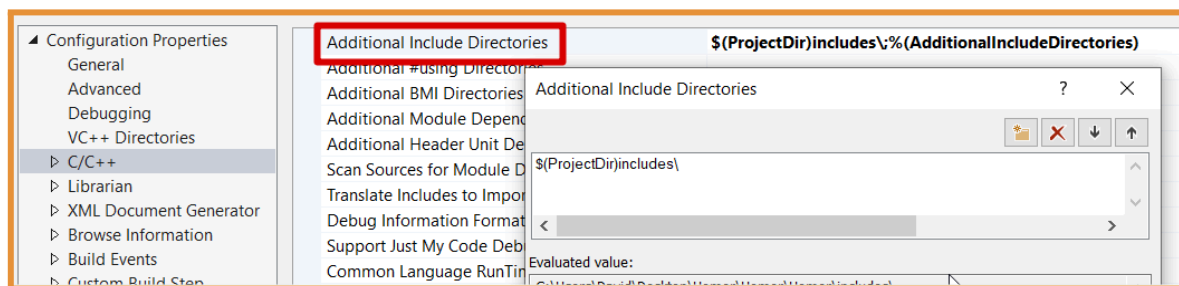
Attention: vous allez inévitablement mélanger les sources et les includes en cours de session, faites attention pour vous éviter des soucis.

Utilisez un « *namespace* » pour tout le code de la librairie. Ceci nous permettra d'utiliser les noms de classe que nous voulons sans causer de problème avec d'autres librairies qui utilise les mêmes noms que nous.

```
namespace homer {
    class Engine final {
    };
}
```

Le mot-clé « *final* » indique que l'on ne permet pas d'hériter de cette classe.

Dans les propriétés du projet, ajoutez le chemin vers votre répertoire «includes» pour que le programme trouve ces fichiers à la compilation.



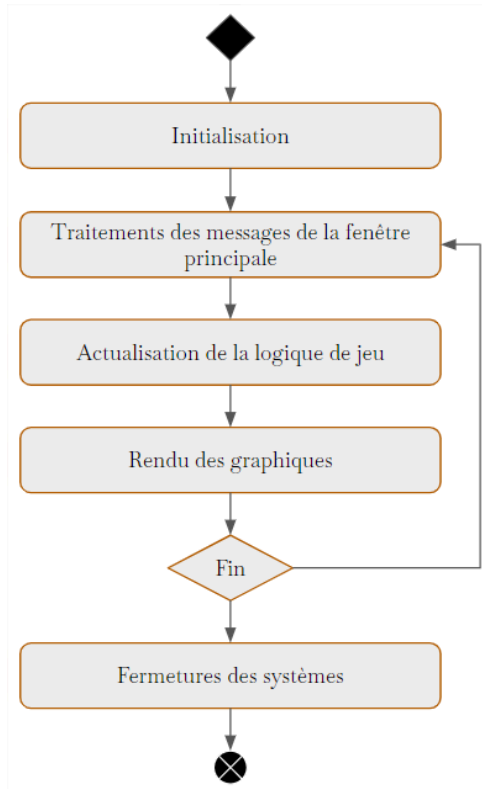
\$(ProjectDir) est une macro qui retourne le répertoire dans lequel se trouve notre projet. Si vous déplacez le projet, la macro retourne le nouvel emplacement. N'écrivez jamais les chemins directement. Utilisez toujours des macros.



[Youtube : configuration](#)

La boucle de jeu

Contrairement à un logiciel de traitement de texte, qui attend que l'utilisateur appuie sur une touche au clavier pour faire quelque chose, un jeu doit constamment se mettre à jour en temps réel. Les animations, les sons, la musique, les particules, tous ces systèmes doivent continuer sans attendre l'interaction du joueur. Les responsabilités de votre classe « engine » sont :



Initialisation : démarrage des services que l'engin partage dans le jeu.

Traitements des messages : gestion des messages que reçoit la fenêtre pour permettre de la manipuler et recevoir les entrées du joueur.

Actualisation : mise à jour de la logique du jeu.

Rendu : affichage de l'état actuel du jeu

Fermeture : lorsqu'on sort de la boucle, l'engin s'occupe aussi de fermer les services précédemment initialisés.

La boucle de jeu lit les entrées du joueur, mais elle n'attend pas qu'il y en ait avant d'actualiser le reste du monde: elle est hors de contrôle du joueur.

Nous connaissons maintenant ce que nous avons besoin dans notre classe d'engin pour implémenter une boucle de jeu, ajoutons le maintenant :

```

namespace homer {
    class Engine final {
    public:
        bool Init(const std::string& title, int w, int h);
        void Start();

    private:
        void ProcessInput();
        void Update();
        void Render();
        void Shutdown();

    private:
        bool m_IsRunning = false;
        bool m_IsInit = false;
    };
}
  
```

Nous avons séparé l'initialisation du démarrage de la boucle pour laisser au jeu la possibilité de faire quelque chose après l'initialisation, mais juste avant de donner le contrôle à l'engin. La méthode « Start » entre dans la boucle et « update » le jeu jusqu'à ce que le joueur décide de quitter.

```
void homer::Engine::Start()
{
    if (!m_IsInit)
    {
        if (!Init("Unknown title", 800, 600))
        {
            return;
        }
    }

    m_IsRunning = true;

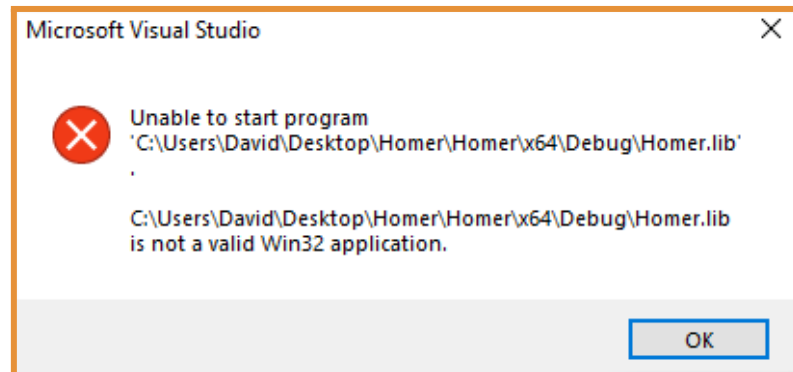
    while (m_IsRunning)
    {
        ProcessInput();
        Update();
        Render();
    }

    Shutdown();
}
```

La boucle commence par vérifier les entrées durant un «frame», puis la méthode «Update» fait avancer la logique du jeu d'un pas. Finalement, «Render» affiche ce qu'il s'est passé durant ce «frame». Lorsque le jeu sort de cette boucle, aussitôt nous fermons les services avant de quitter.

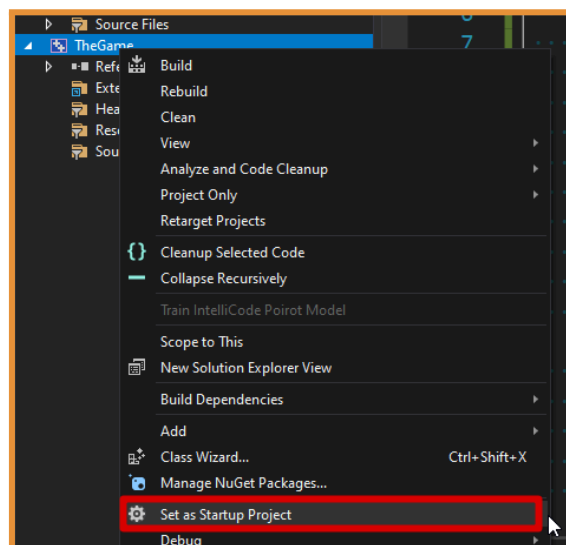
Librairie statique

Si vous tentez d'exécuter votre engin à ce stade, vous obtenez une erreur comme dans l'image suivante :



Ici, « *unable to start program* » vous indique que votre librairie n'est **pas un exécutable** (pas un .exe). Il nous faudra faire un programme hôte (le jeu) qui l'inclut dans sa compilation et qui l'utilise.

🔑 Dans la solution, ajoutez un autre projet vide. Nous appellerons ce projet : «TheGame». Vous pouvez remplacer ce nom si vous savez déjà le nom du jeu que vous désirez faire cette session. Pour que Visual Studio utilise votre nouveau projet lorsqu'il exécute la solution, vous devez utiliser l'option « *Set as Startup Project* ».



C'est ce projet que Visual Studio doit démarrer lorsque vous appuyez sur «Build».


Ajoutez un fichier source pour contenir le « main » du jeu. C'est le point d'entrée qui démarre l'engin.

```
#define WIN32_LEAN_AND_MEAN
#define VC_EXTRALEAN
#include <Windows.h>

INT WINAPI WinMain(_In_ HINSTANCE, _In_opt_ HINSTANCE, _In_ PSTR, _In_ INT) {
    return 0;
}
```


Lignes 1 et 2 : Grâce à ces préprocesseurs, «Windows.h» se limitera au strict nécessaire pour ce qu'il ajoute dans notre programme. Ceci aura pour effet de faire un programme plus rapide à compiler.

Ligne 5 : Pour une application en mode «Windows», la fonction d'entrée est «WinMain». Nous n'utilisons pas les paramètres, mais ils font partie de la signature de la fonction qui doit être présente dans notre programme.

 Dans les propriétés de ce projet, configurez le système pour qu'il soit non pas en mode « Console », mais plutôt en mode « Window ». **Linker > System > SubSystem : Window**

Visual Studio

Lorsque vous créez un projet avec Visual Studio, il utilise par défaut une version de «Windows SDK» (dans l'image : 10.0) et une version du «Platform Toolset» (dans l'image : v142). Si vous avez plusieurs membres dans votre équipe, ou si vous utilisez plusieurs ordinateurs pour le développement de votre projet, assurez-vous d'être aux mêmes versions pour éviter d'avoir des conflits entre les différents postes de travail.

 Dans la section « References » du projet « TheGame », ajoutez une référence à votre librairie. Maintenant, lorsque « TheGame » est compilé, la librairie sera incluse dans l'exécutable.



Challenge 2.1 : enlevez la configuration « x86 » comme nous l'avons fait pour la librairie et ajoutez les includes de l'engin en utilisant \$(SolutionDir).

Ce programme est une rampe de lancement pour l'engin. Si l'initialisation réussit, on laisse la chance au programme d'initialiser aussi des éléments de jeu et on démarre l'engin. Lorsque la méthode « Start » se termine, ça veut dire que nous sommes sortis de la boucle de jeu et le programme n'a plus qu'à se fermer.

```
#define WIN32_LEAN_AND_MEAN
#define VC_EXTRALEAN
#include <Windows.h>
#include <Engine.h>

void InitGameplay()
{
}

INT WINAPI WinMain(_In_ HINSTANCE, _In_opt_ HINSTANCE, _In_ PSTR, _In_ INT)
{
    homer::Engine theEngine;
    if (theEngine.Init("Test Game", 800, 600))
    {
        InitGameplay();
        theEngine.Start();
    }

    return 0;
}
```

Pour l'instant, le programme s'exécute à l'infini et vous devez le fermer en arrêtant l'exécution par Visual Studio.



Challenge 2.2 : ajoutez une méthode « Exit » qui ferme (**temporairement**) l'engin après la première mise à jour.

SDL

Que ce soit Windows, Linux ou Mac, chaque plateforme a sa propre façon de créer et d'afficher des fenêtres et des graphiques, gérer les entrées de l'utilisateur et accéder à tout système de bas niveau. SDL fournit un moyen uniforme et simple d'accéder à ces fonctionnalités spécifiques à chaque plateforme.

Cette uniformité se traduit concrètement par un gain de temps à améliorer votre jeu, plutôt que de vous soucier de la façon dont une plateforme particulière vous permet d'afficher une image ou comment obtenir les entrées de l'utilisateur, etc. La programmation de jeux peut être parfois difficile, et d'avoir une librairie telle que SDL peut rendre votre jeu opérationnel relativement rapidement.

Vous pouvez obtenir SDL à partir de l'adresse suivante : www.libsdl.org

Parmi les choix, téléchargez les «*Development Libraries*» pour Windows :

`SDL2-devel-2.0.xx-VC.zip (Visual C++ 32/64-bit)`

Pour ajouter SDL à votre projet, copiez la librairie dans un répertoire « sdl » dans le répertoire de votre solution, ouvrez les propriétés et servez-vous du tableau suivant pour faire votre configuration.

Configuration

Dans Visual Studio, vous pouvez utiliser de multiples configurations. Par défaut, Visual Studio ajoute DEBUG et RELEASE pour vous, mais vous pouvez aussi en ajouter d'autres. Lorsque vous changez les options de votre projet, **vous pouvez le faire pour toutes les configurations et plateformes en même temps** en utilisant les options dans le haut de la page de paramètres. Ceci vous évitera de devoir le faire à de multiples reprises pour chacune des configurations.

Il ne vous reste plus qu'à indiquer à votre projet où se trouve votre librairie de SDL pour que vous puissiez l'utiliser dans votre jeu.

Propriétés	
C/C++ - General	
Additional Include Directories	\$(SolutionDir)sdl\include\
Linker - General	
Additional Library Directories	\$(SolutionDir)sdl\lib\
Linker - Input	
Additional Dependencies	SDL2.lib SDL2main.lib

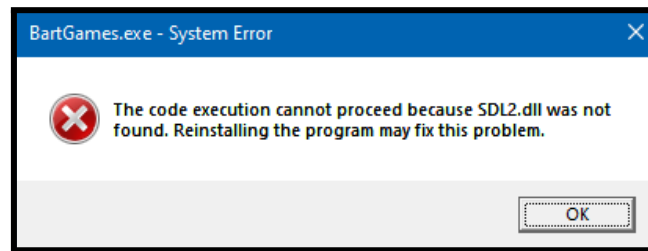
Initialisation de SDL

Lorsque vous initialisez SDL, vous avez le choix de démarrer que les systèmes nécessaires ou les utiliser tous.

```
if (SDL_Init(SDL_INIT EVERYTHING) != 0)
{
    SDL_Log(SDL_GetError());
    return false;
}
```

Options de SDL	
SDL_INIT_TIMER	timer subsystem
SDL_INIT_AUDIO	audio subsystem
SDL_INIT_VIDEO	video subsystem
SDL_INIT_JOYSTICK	joystick subsystem
SDL_INIT_EVENTS	events subsystem
SDL_INIT EVERYTHING	all of the above subsystems

Si vous initialisez SDL maintenant et que vous exécutez votre programme, vous aurez l'erreur suivante :



Pour utiliser SDL, votre programme doit «trouver» les DLL. Normalement, un joueur doit installer les «Runtime Binaries» qui ajouteront les DLL de SDL sur l'ordinateur du joueur. Pour le développement de votre jeu, vous pouvez simplement copier tous les DLL dans le même répertoire que votre projet.

Fermeture de SDL

Avant de quitter le programme, nous devons fermer les systèmes de SDL qui ont été ouverts, détruire la fenêtre (`SDL_Window`) et le renderer (`SDL_Renderer`) et toutes les ressources que votre jeu a utilisées.

```
SDL_DestroyRenderer(_renderer);  
SDL_DestroyWindow(_window);  
SDL_Quit();
```



Challenge 2.3 : initialisez et fermez SDL aux bons moments dans l'engin. Attention, **vous n'avez pas le droit d'inclure SDL dans les fichiers d'entête de l'engin.**

Création d'une fenêtre

Avant d'entrer dans la boucle de jeu, pour être en mesure de rendre votre jeu à l'écran, vous devez créer une fenêtre.

```
int _x = SDL_WINDOWPOS_CENTERED;
int _y = SDL_WINDOWPOS_CENTERED;
Uint32 _flag = SDL_WINDOW_TOOLTIP;

// _window est une variable de type SDL_Window*
_window = SDL_CreateWindow(title.c_str(), _x, _y, w, h, _flag);
if (!_window)
{
    SDL_Log(SDL_GetError());
    return false;
}
```

Après avoir fourni la position et la taille de la fenêtre, vous aurez à indiquer le type de fenêtre que vous voulez.

Quelques options que vous pouvez utiliser (ou combiner avec un «ou» binaire) sont :

Types de fenêtres	
SDL_WINDOW_FULLSCREEN	fullscreen window
SDL_WINDOW_FULLSCREEN_DESKTOP	fullscreen window at desktop resolution
SDL_WINDOW_BORDERLESS	no window decoration
SDL_WINDOW_RESIZABLE	window can be resized
SDL_WINDOW_UTILITY	utility window
SDL_WINDOW_TOOLTIP	window should be treated as a tooltip

Fenêtre plein écran

Attention, ne travaillez pas en «DEBUG» avec une fenêtre plein écran. Vous risquez de couvrir tout l'écran et si le programme est planté, il vous bloquera l'accès à Visual Studio et vous ne pourrez plus le fermer.

Rendu dans la fenêtre

Maintenant que vous avez initialisé SDL et créé une fenêtre, tout ce qu'il vous reste à faire est d'afficher le jeu pour le joueur. En SDL, vous devez d'abord initialiser un «SDL_Renderer» pour dessiner des graphiques dans la fenêtre.

```
_flag = SDL_RENDERER_ACCELERATED;

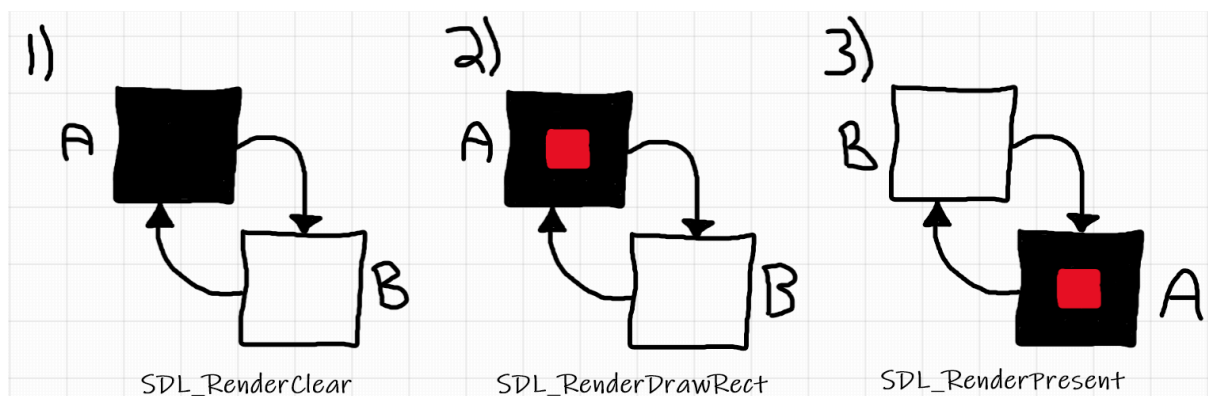
// _renderer est une variable de type SDL_Renderer*
_renderer = SDL_CreateRenderer(_window, -1, _flag);

if (!_renderer)
{
    SDL_Log(SDL_GetError());
    return false;
}
```

Pour configurer le «renderer», vous devez utiliser un (ou plusieurs) option suivante :

Options du «renderer»	
SDL_RENDERER_SOFTWARE	the renderer is a software fallback
SDL_RENDERER_ACCELERATED	the renderer uses hardware acceleration
SDL_RENDERER_PRESENTVSYNC	present is synchronized with the refresh rate
SDL_RENDERER_TARGETTEXTURE	the renderer supports rendering to texture

Le «renderer» représente l'état de la carte graphique. Si vous changez la couleur (SDL_SetRenderDrawColor) active, les prochains graphiques vont l'utiliser, car la couleur fait partie du «renderer». Par exemple, pour afficher un rectangle :





Challenge 2.4 : avec SDL, videz l'écran avec une couleur de votre choix, dessinez un rectangle avec une autre couleur puis finalement, présentez le résultat dans la fenêtre.

Traitement de messages

Si vous ne traitez pas les messages, vous ne pourrez pas déplacer ni fermer votre fenêtre et encore moins utiliser le clavier ou la souris. Il est donc important de rapidement ajouter le code pour traiter les messages à fin de pouvoir tester votre fenêtre.

```
void homer::Engine::ProcessInput()
{
    SDL_Event _event;
    while (SDL_PollEvent(&_event))
    {
        switch (_event.type)
        {
            case SDL_QUIT:
                Exit();
                break;
        }
    }
}
```

Il existe beaucoup trop de types de messages pour tous les afficher ici, mais voici un petit tableau des plus important pour faire un jeu :

Types de messages	
SDL_QUIT	user-requested quit;
SDL_KEYDOWN	key pressed
SDL_KEYUP	key released
SDL_MOUSEMOTION	mouse moved
SDL_MOUSEBUTTONDOWN	mouse button pressed
SDL_MOUSEBUTTONUP	mouse button released
SDL_MOUSEWHEEL	mouse wheel motion

Pour les événements liés aux boutons de la souris, vous devez premièrement savoir quel bouton a été activé. L'événement SDL contient cette information et vous pouvez l'avoir de cette façon :

```
case SDL_MOUSEBUTTONDOWN:
    SDL_MouseButtonEvent _buttonDown = _event.button;
    SDL_Log("Button down : %d", _buttonDown.button);
    SDL_Log("at (%d, %d)", _buttonDown.x, _buttonDown.y);
    break;

case SDL_MOUSEBUTTONUP:
    SDL_MouseButtonEvent _buttonUp = _event.button;
    SDL_Log("Button up : %d", _buttonUp.button);
    SDL_Log("at (%d, %d)", _buttonUp.x, _buttonUp.y);
    break;
```

Le bouton gauche correspond au bouton 1, le bouton du milieu à 2 et le bouton de droite à 3. Si votre souris a plus de boutons, ce n'est pas certain que SDL les supporte. Et de toute façon, ils sont probablement configurés pour entrer une touche du clavier. Notez aussi que l'événement contient la position de la souris lorsque le message a été envoyé.

Avec le message SDL_MOUSEMOTION, vous pouvez connaître la position du curseur de la souris, sans que le joueur ait à cliquer dans la fenêtre.

```
case SDL_MOUSEMOTION:
    SDL_MouseMotionEvent _motion = _event.motion;
    SDL_Log("%d, %d", _motion.x, _motion.y);
    break;
```

Pour les événements du clavier, ça nous fait beaucoup de boutons à vérifier. Nous allons plutôt opter pour une autre stratégie. SDL peut nous construire un tableau dans lequel chaque position du tableau correspond à une touche du clavier et le contenu sont état : enfoncé (1) ou relâché (0). Pour obtenir ce tableau, utilisez le code suivant :

```
const unsigned char* _keyStates = SDL_GetKeyboardState(nullptr);
```

Chaque état de touche du clavier est représenté par un 0 ou un 1, un «char» non signé est amplement suffisant pour ceci. Le paramètre de la fonction SDL nous permet d'obtenir le nombre de touches qui se retrouve dans le tableau. Si vous

n'en avez pas besoin, vous pouvez passer «`nullptr`». Si vous gardez une variable pour ce tableau, et que vous la mettez à jour chaque fois que les messages de clavier sont interceptés (`SDL_KEYUP` et `SDL_KEYDOWN`), le tableau contiendra l'état des touches à jour et il vous suffira de regarder à la bonne position pour savoir si une touche est appuyée ou pas.

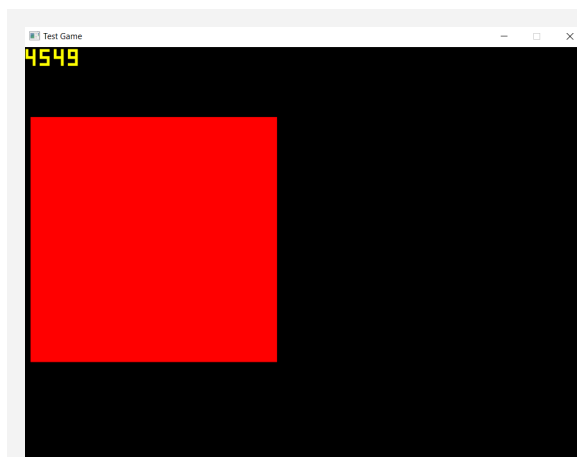


Challenge 2.5 : ajoutez le nécessaire à l'engin pour déplacer un rectangle à l'aide du clavier (avec les flèches haut, bas, gauche et droite). Utilisez les constantes `SDL_SCANCODE_####`.

Note : si vous trouvez que votre rectangle se déplace trop vite, c'est normal, votre application (pour l'instant) «roule» à plus de 3000 fps.

Gestion du temps

Actuellement, une fois entré dans sa boucle, votre jeu s'exécute le plus rapidement possible. Et plus que votre ordinateur est puissant, plus le jeu sera rapide. Un programme en temps réel n'attend pas une intervention de l'utilisateur et prend toutes les ressources qu'il peut. Sur un appareil mobile, vous pouvez littéralement drainer la pile.



Fraps

Pour tester si les FPS fonctionnent, j'utilise Fraps : un programme très simple qui ajoute les FPS (dans l'image, en haut à droite) dans tous les jeux qui démarrent sur mon ordinateur.

<https://fraps.com/>

Mais, est-ce vraiment nécessaire d'actualiser le jeu aussi rapidement? Sur l'ordinateur que j'utilise, notre boucle fait environ 3000 itérations à chaque seconde. Pour être fluide pour l'oeil humain, 30 itérations secondes seraient suffisantes.

Sur un ordinateur, souvent, nous opterons pour 60 itérations secondes (FPS pour «Frame per seconds»). Pour obtenir 60 itérations par secondes, chaque itération doit durer 0.016 seconde (1/60). Pour obtenir le temps de l'ordinateur, nous utilisons la fonction «clock» sur Windows. Par contre, cette fonction nous retourne des millisecondes. Pour avoir des secondes, il nous suffit de diviser notre résultat par 1000 : chaque itération doit durer 16.66 ms. La variable «_end» est initialisée en utilisant le temps actuel.

```
clock_t _end = clock();

while (m_IsRunning) {
    const clock_t _start = clock();
    float _dt = (_start - _end) * 0.001f;

    ProcessInput();
    Update(_dt);
    Render();

    _end = _start;
}
```

Ligne 5 : le «delta time» que l'on désire passer à la fonction «Update» est en fait la différence de temps entre deux mises à jour. Comme notre valeur est en millisecondes, nous la ramenons en secondes.

Challenge 2.6 : en vous basant sur le chapitre « game loop », faites-en sorte que le jeu s'exécute à une vitesse maximale de 60 FPS. Utilisez la méthode simple.

ATTENTION, il ne faut pas utiliser une valeur égale ou inférieure à 0 avec la fonction « Sleep » que vous pourrez utiliser en incluant « Windows.h ».