LES BASES FONDAMENTALES DE L'AFFICHAGE GRAPHIQUE EN SDL2

Introduction

Dans cette, que je juge vraiment très intéressante, nous verrons :

- **Faire du coloriage en SDL2**
- **Affichage des formes primitives en SDL2**
- **Gérer la transparence en SDL2**
- **Gérer les collisions en SDL2**
- **Gérer textures en SDL2**
- **Gérer les surfaces en SDL2**
- **Afficher des images en SDL2**
- **Affichier du texte en SDL2**
- **Gérer les rotations en SDL2**

Cette, partie est vraiment importante car ce sont **les bases fondamentales de la SDL**, n'hésitez par à relire plusieurs fois ce chapitre si cela vous paraît difficile à comprendre. Je vous mets en garde ce chapitre sera long.

Le coloriage

Dans sur le cours précedent, nous avions réussi à créer notre **Game Loop** prête à l'emploi, nous pouvions quitter notre fenêtre facilement, en cliquant sur la croix

rouge en haut, à droite de la fenêtre. Maintenant, la question légitime c'est **comment dessiner en SDL2** ? Tout d'abord il faut vous expliquer comment correctement dessiner.

En effet, on ne dessine pas en début de la Game Loop ou lors de la gestion des événements. Souvenez-vous des étapes de la Game Loop :

1. Gestion des événements
2. Mise à jour
3. rendu

Nous allons ici travailler sur l'étape 3 de la Game Loop. Pour dessiner il faut suivre l'ordre suivant :

1. **Nettoyer le contenu actuel de la fenêtre**
2. **Dessiner ce qu'on a à dessiner**
3. **Mettre à jour la fenêtre**

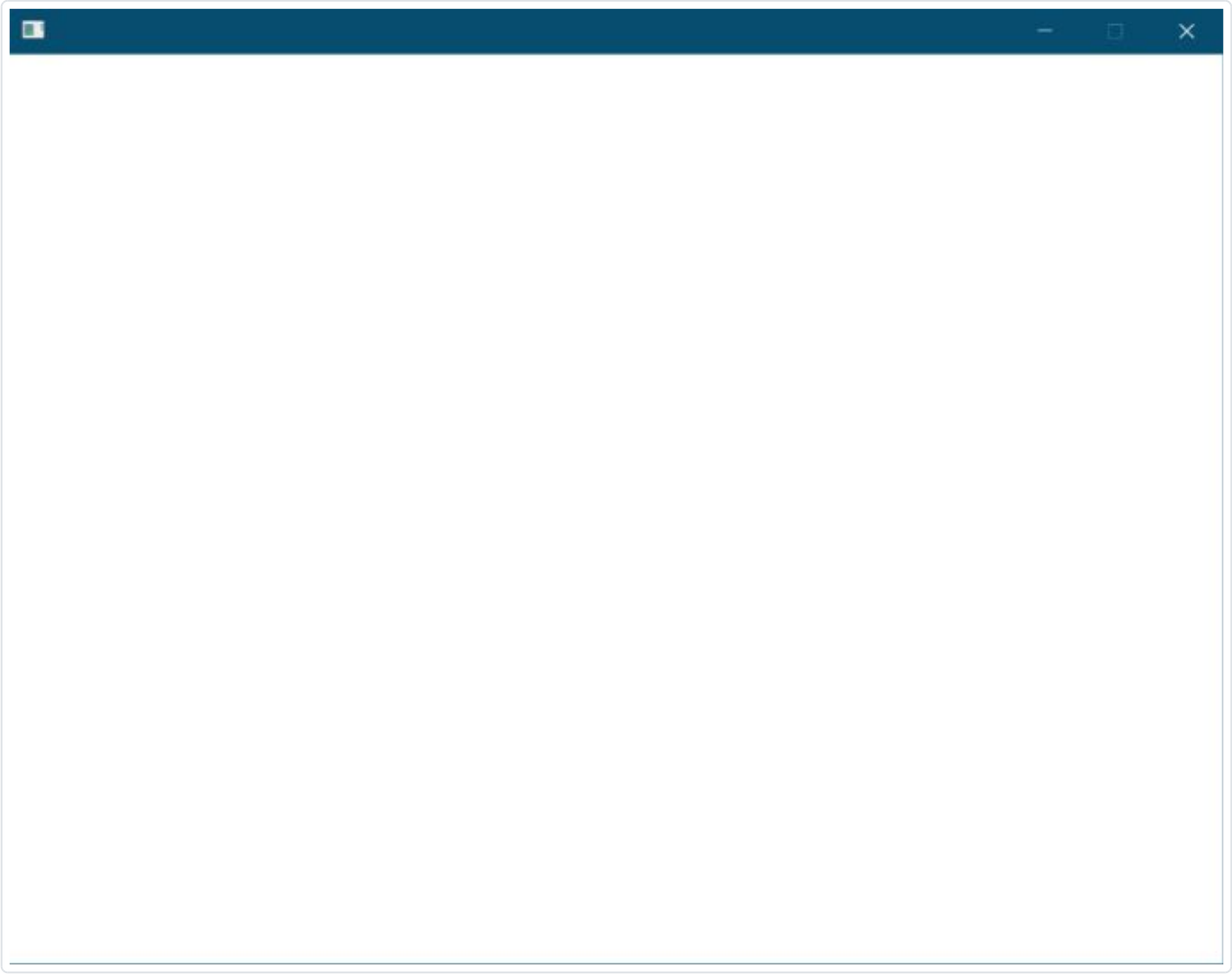
Pour cela nous, devons étudier 3 fonctions.

```
SDL_SetRenderDrawColor()();
SDL_RenderClear();
SDL_RenderPresent();
```

Respectivement, ces fonctions servent à :

- `SDL_SetRenderDrawColor()` : choix de la couleur pour dessiner.
- `SDL_RenderClear()` : nettoie le contenu de la fenêtre.
- `SDL_RenderPresent()` : mets à jour la fenêtre.

Pour l'instant votre fenêtre est toute blanche normalement comme ceci :



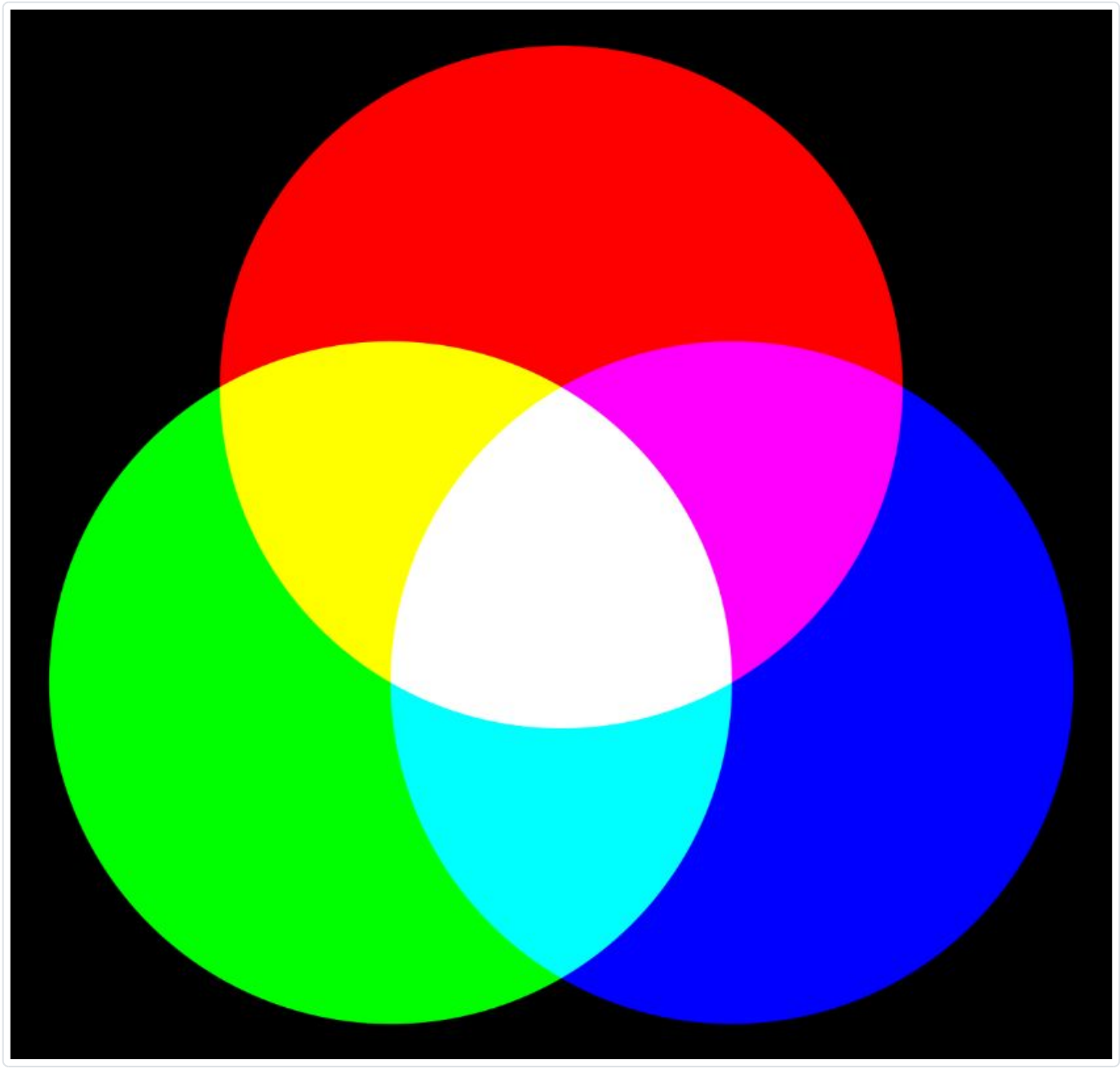
Il est temps de commencer par la première fonction `SDL_SetRenderDrawColor()`, voici son prototype :

```
int SDL_SetRenderDrawColor(SDL_Renderer* renderer,
                           Uint8         r,
                           Uint8         g,
                           Uint8         b,
                           Uint8         a)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est la quantité de rouge.

- Le troisième paramètre est la quantité de vert.
- Le quatrième paramètre est la quantité de bleu.
- Le cinquième paramètre est la transparence.
- Elle retourne, 0 si elle a réussi ou une valeur négative s'il y a une erreur.

Généralement, en informatique on représente les couleurs avec une quantité de rouge, vert, bleu, et parfois un composant alpha. Cette façon de définir une couleur s'appelle la synthèse additive.



Vous pouvez lire l'article wikipédia pour savoir ce que c'est réellement. Mais sachez que cette représentation de couleur est vraiment très présente en informatique graphique .

Nous constatons que les paramètres pour les couleurs sont des `Uint8`, c'est un type que SDL a créé pour pouvoir représenter des entiers non signé (c'est-à-dire qu'ils ne peuvent pas être négative) et de taille 8 bits. Par conséquent ils peuvent

aller jusqu'à la valeur 255 soit 256 symboles 2^8-1 (valeur) et 2^8 (symbole).

> **Information**
>
> Je ne vais plus trop vérifier, les retours de fonction sauf exception pour simplifier le code pour vous, mais aussi moi.

```
SDL_SetRenderDrawColor()(pRenderer, 0, 0, 0, 255);
```

Rien de compliqué dans cette fonction, sur cette exemple nous avons choisi de dessiner avec la couleur noire.

Maintenant il faut remplir toute la fenêtre, de cette couleur, vous allez devoir utiliser la fonction `SDL_RenderClear()`, et vous allez rire. Elle est toute aussi simple à utiliser voici son prototype.

```
int SDL_RenderClear(SDL_Renderer* renderer)
```

- Le premier paramètre est le rendu de fenêtre.
- Elle retourne 0 en cas de succès ou une valeur négative en cas d'échec.

Voici comment l'utiliser :

```
SDL_RenderClear(pRenderer);
```

Sur l'échelle de la simplicité d'utilisation je mets 10 / 10 . Maintenant à ce stade vous avez ceci :

```
#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_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);
    }

    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    SDL_Quit();

    return EXIT_SUCCESS;

}
```

Si vous compilez ce programme, vous tomberez encore sur une fenêtre blanche. Et c'est tout à fait normal car il manque la partie **mise à jour de la fenêtre**. Pour mettre à jour la fenêtre il faut utiliser la fonction encore plus simple `SDL_RenderPresent()` voici son prototype :

```
void SDL_RenderPresent(SDL_Renderer* renderer)
```

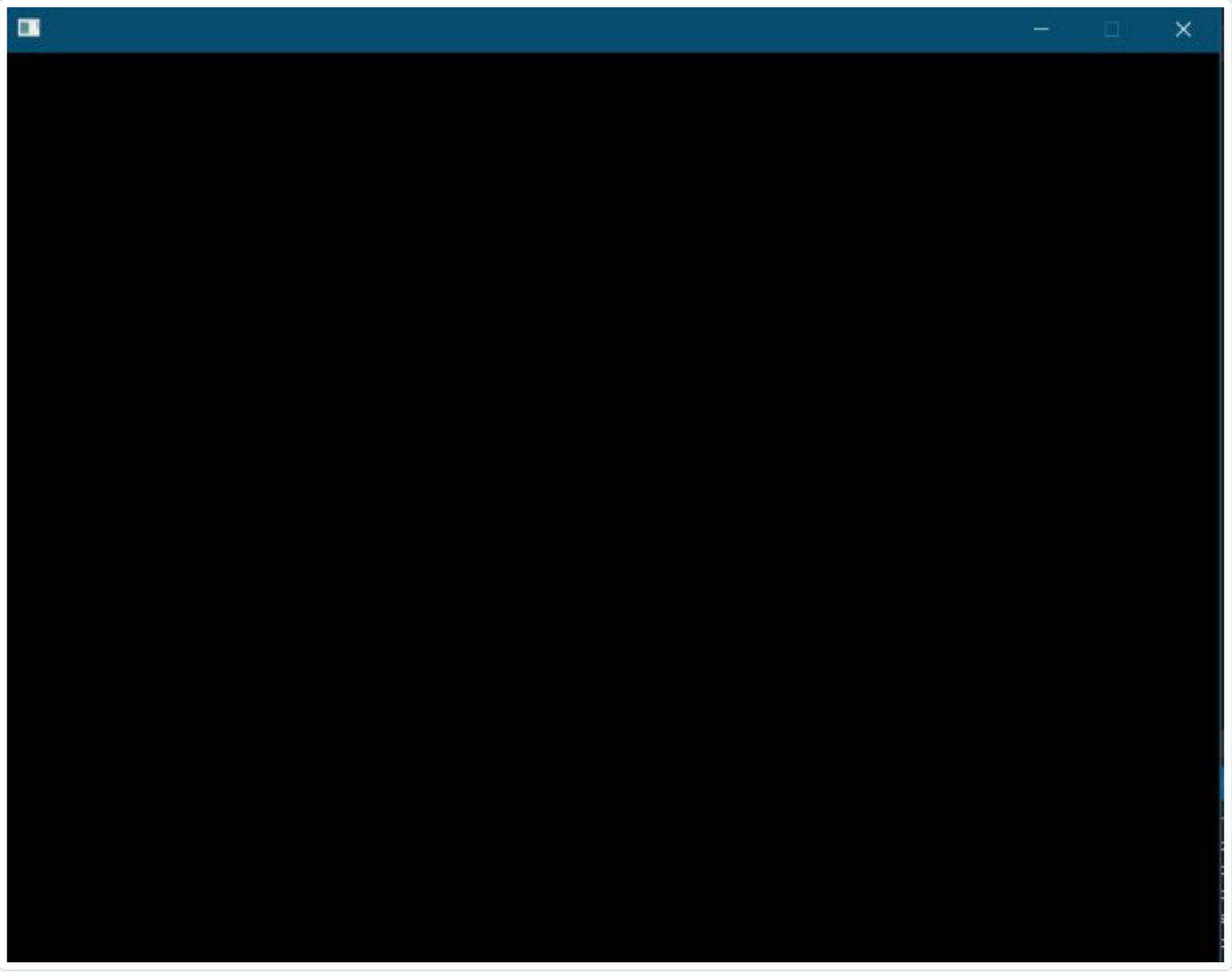
- Elle prend uniquement le rendu en paramètre.
- Elle retourne, rien du tout.

Voici le résultat final :

```
#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_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);
        SDL_RenderPresent(pRenderer);
    }
```

```
    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
     SDL_Quit();

    return EXIT_SUCCESS;

}
```

Maintenant, si vous compilez, votre fenêtre sera remplie de la couleur noire comme ceci :



Maintenant, vous avez appris à **choisir une couleur**, il est temps de **dessiner des formes géométriques**.

L'affichage des formes primitives

Comme je l'ai dit en introduction, la SDL est une bibliothèque minimaliste elle permet d'afficher que des formes géométriques simples à savoir :

- **Dessiner des points**
- **Dessiner des rectangles**
- **Dessiner des lignes**

Nous allons, maintenant dessiner ces 3 primitives, nous commencerons par les points, les rectangles puis nous finirons par les lignes.

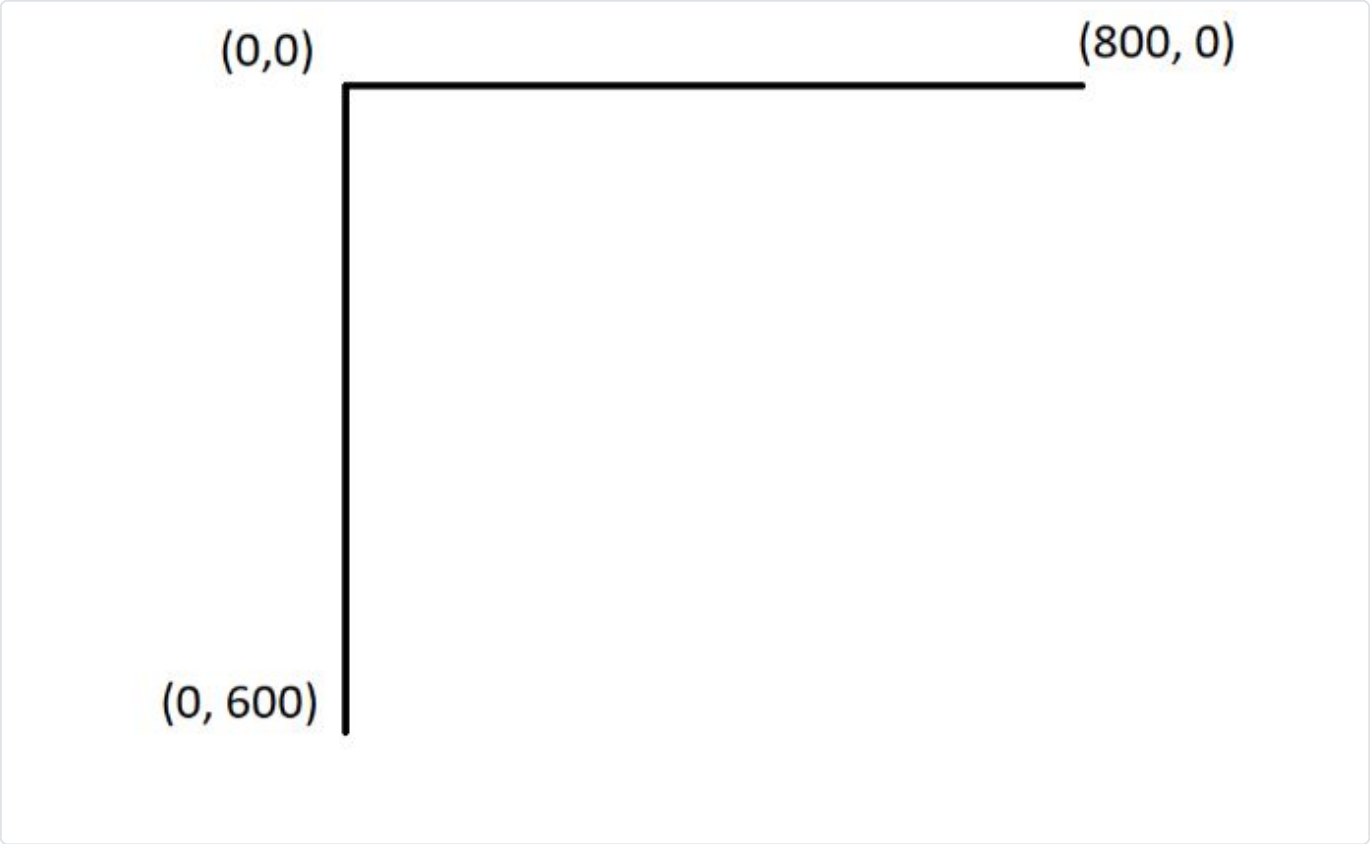
Les points

SDL, représente les points via à une structure `SDL_Point`, voici le code qui représente un `SDL_Point` :

```
typedef struct SDL_Point
{
    int x;
    int y;
} SDL_Point;
```

Cette structure, représente un champ x, qui représente la position x du point, et ensuite un champ nommé y qui symbolise la position y du point.

En effet, un point en mathématiques, peut représenter une position, par exemple la position d'un joueur. Mais il faut que je vous explique quelque chose, en informatique graphique 2D, on représente un repère orthonormé de cette manière :



Ici, ceci peut gêner, pour ceux qui ont l'habitude d'avoir un repère où l'origine se retrouve au centre, or en informatique graphique 2D l'origine est située en haut à gauche.

C'est-à-dire que si vous voulez placer un point au centre de l'écran, alors il faudra pas mettre les coordonnés de cette manière (0, 0), mais plutôt de cette façon (800 / 2, 600 / 2), soit au centre de l'écran.

Faisons ceci en déclarant un `SDL_Point`, au centre de l'écran :

```
SDL_Point point{ 800 / 2, 600 / 2};
```

Maintenant, ce qu'il faut faire c'est dessiner ce point, et je veux que ce point soit de couleur ROUGE, pour pouvoir dessiner un point à l'écran, il faut utiliser la fonction `SDL_RenderDrawPoint()`, voici sa signature :

```
int SDL_RenderDrawPoint(SDL_Renderer* renderer,
                        int           x,
                        int           y)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est la position en x.
- Le troisième paramètre est la position en y.
- Elle retourne 0 en cas de succès et une valeur négative en cas d'échecs.

Pour cela, je dois dessiner entre `SDL_RenderClear()` et `SDL_RenderPresent()`, parce que si nous le faisons avant `SDL_RenderClear()`, on aura dessiné notre point mais toute suite après à cause de la fonction `SDL_RenderClear()` il disparaîtra sous la couleur qu'on aura décidé via la fonction `SDL_SetRenderDrawColor()`, un exemple vaut mieux que des explications donc voici :

```
SDL_SetRenderDrawColor()(pRenderer, 0, 0, 0, 255); // On choisis la couleur de rempli
SDL_RenderClear(pRenderer); // On colorie toutes notre fenêtre en noir

SDL_SetRenderDrawColor(pRenderer, 255, 0, 0, 255); // On choisis la couleur rouge
SDL_RenderDrawPoint(pRenderer, point.x, point.y); // On dessiner un point

SDL_RenderPresent(pRenderer); // On mets a jour notre fenêtre
```

Je vous propose maintenant, de faire quelque chose de sympa. Créer plusieurs points avec des positions aléatoires et afficher 5000 points de couleur blanche, je vous donne ceci comme exercice de pratique.

CORRECTION :

J'espère que vous avez réussi à faire, cet exercice voici donc la correction :

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

#include <random>
#include <chrono>
```

```
#include <array>

template<typename T>
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };

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(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }

    SDL_Event events;
    bool isOpen{ true };


    std::default_random_engine generator{ static_cast<unsigned int>(std::chrono::syst
    std::uniform_int_distribution distribution{ 0, WIDTHSCREEN<int>};

    std::array<SDL_Point, TOTAL_POINTS<int>> points;

    for (auto& point : points)
        point = { distribution(generator), distribution(generator) };


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

```
        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);

        SDL_SetRenderDrawColor(pRenderer, 255, 255, 255, 255);

        for (const auto& point : points)
            SDL_RenderDrawPoint(pRenderer, point.x, point.y);

        SDL_RenderPresent(pRenderer);
    }

    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
     SDL_Quit();

    return EXIT_SUCCESS;

}
```

Si vous avez réussi, je vous félicite. Pour celui qui n'a pas réussi n'hésitez pas à me contacter pour que je puisse vous aider en programmation (je suis disponible et proche de mes lecteurs).

On constate ici que pour pouvoir dessiner tous mes points, j'ai dû faire une boucle qui parcourt mon tableau de `SDL_Point`, SDL propose une fonction `SDL_RenderDrawPoints()` (attention ici on a un S à la fin) cette fonction permet de dessiner plusieurs points, voici son prototype ;

```
int SDL_RenderDrawPoints(SDL_Renderer*    renderer,
                         const SDL_Point* points,
                         int              count)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est un pointeur sur le premier élément du tableau de `SDL_Point`.
- Le troisième paramètre est le nombre de `SDL_Point` à dessiner.

Voici la correction mais en utilisant cette fonction `SDL_RenderDrawPoints()` !

```
// Inclure la bibliothèque SDL
#include <SDL2/SDL.h>


// Inclusion des bibliothèque standard C++
#include <random>
#include <chrono>
#include <array>
#include <cstdlib>

// Définition des constante
template<typename T>
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };

int main(int argc, char* argv[])
{
    // Chargement du module vidéo de la SDL
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        return EXIT_FAILURE;
    }

    // Ressource : Fenêtre et rendu
    SDL_Window* pWindow{ nullptr };
    SDL_Renderer* pRenderer{ nullptr };

    // Création d'une fenêtre, en 800x600, et visible
    if (SDL_CreateWindowAndRenderer(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }


    SDL_Event events;
    bool isOpen{ true };

    // Gestion de l'aléatoire
    std::default_random_engine generator{ static_cast<unsigned int>(std::chrono::syst
    std::uniform_int_distribution distribution{ 0, WIDTHSCREEN<int>}; // Selon la loi

    std::array<SDL_Point, TOTAL_POINTS<int>> points; // Tableau des 5000 SDL_Point

    // Création des SDL_Point avec des coordonnée aléatoire
    for (auto& point : points)
```

```
        point = { distribution(generator), distribution(generator) };

    // Game loop
    while (isOpen)
    {
        // Input
        while (SDL_PollEvent(&events))
        {
            switch (events.type)
            {
            case SDL_QUIT:
                isOpen = false;
                break;
            }
        }


        // Rendering
        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255); // Choisir la couleur noir
        SDL_RenderClear(pRenderer); // Colorier en noir toutes la fenêtre

        SDL_SetRenderDrawColor(pRenderer, 255, 255, 255, 255); // Choisir la couleur

        SDL_RenderDrawPoints(pRenderer, &points[0], points.size()); // Dessiner mon t

        SDL_RenderPresent(pRenderer); // Mise à jour de la fenêtre
    }

    // Libération des ressource en mémoire
    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
     SDL_Quit();

    return EXIT_SUCCESS;

}
```

Ici je vous ai commenté tous le code par sympathie .

Les rectangles

Bon, maintenant nous avons vu les `SDL_Point` et nous savons ou placer notre code pour dessiner, passons maintenant au rectangle, pour la SDL.

Pour pouvoir dessiner des rectangles, SDL nous propose la structure `SDL_Rect` voici son implementation fait par SDL :

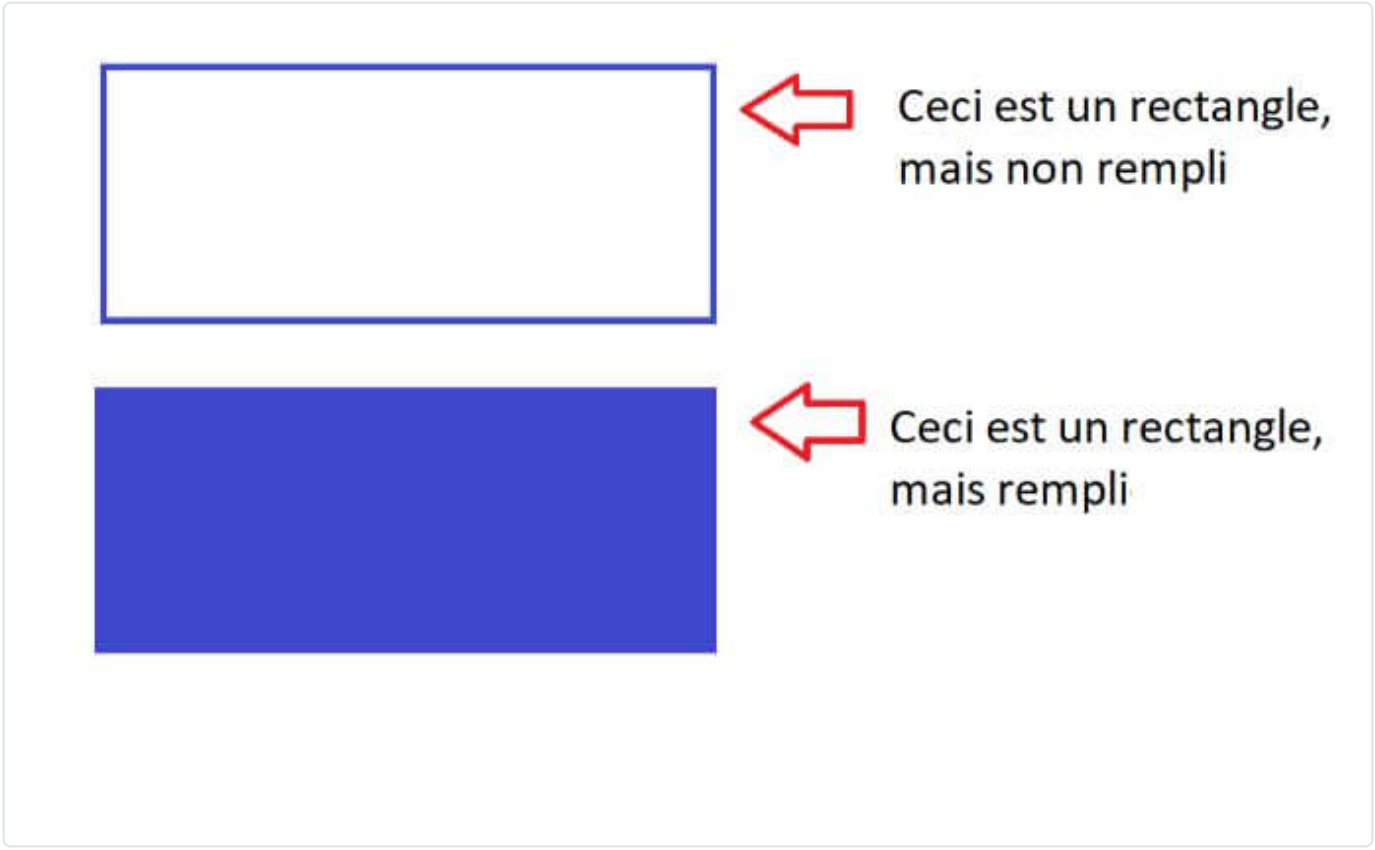
```
typedef struct SDL_Rect
{
    int x, y;
    int w, h;
} SDL_Rect;
```

Cette structure, contient un `SDL_Point` en effet nous constatons, deux champs x et y cela signifie que c'est une structure de type `SDL_Point`. Nous avons aussi deux nouveaux champs, le champ w et h, ce qui signifie en premier un width (largeur) et un height (hauteur).

Pour afficher notre premier rectangle. Nous allons devoir voir la chose la plus magnifique que DIEU a pu crée sur Terre, les rectangles ! Avec les rectangles on peut tout faire. vous verrez que les jeux 2Ds ne sont constitués quasiment que de rectangles. N'oubliez pas de remercier DIEU pour cette magnifique création. "MERCI DIEU". Pour les athées remercier la nature. "MERCI nature". Ne vous inquiétez pas je n'ai pas bu .

Poursuivons hmm. Pour afficher notre premier rectangle il faut déclarer une structure du type `SDL_Rect` et ceci ce fait comme un `SDL_Point` mais en renseignant la position x, y, w, h dans cet ordre.

Maintenant il éxiste deux façons pour afficher un rectangle, soit on fait un rectangle qui peut être rempli soit un rectangle non rempli, en anglais on pourrait parler de "fill" et de "not fill" rectangle comme ceci :



Il existe plusieurs façons de dessiner votre rectangle. Soit vous réinventez la roue, soit vous utilisez deux fonctions que SDL met en place pour vous.

Je vous propose en guise d'exercice de réinventer la roue. Oui réinventer la roue est bien pour apprendre . Normalement vous avez tout en main pour pouvoir réussir cet exercice, si vous avec vue les `SDL_Point` :p.

Il suffit juste de réfléchir. Ceci est important réellement car si vous ne réussissez pas instinctivement, c'est que vous n'avez pas assez réfléchi. Si je vous donne les schémas pour pouvoir penser, la solution sera trop évidente ! Car oui c'est facile de re-coder certaines fonctions de SDL mais pas tous :).

Pour dessiner un rectangle la structure `SDL_Rect` vous montre les infos qu'il faut :

CORRECTION

```
// Inclure la bibliothèque SDL
#include <SDL2/SDL.h>
```

```
// Inclusion des bibliothèque standard C++
#include <cstdlib>

// Définition des constante
template<typename T>
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };

// Prototype
void draw_rectangle_fill(SDL_Renderer* renderer, const SDL_Rect& rectangle, const SDL
void draw_rectangle_not_fill(SDL_Renderer* renderer, const SDL_Rect& rectangle, const

int main(int argc, char* argv[])
{
    // Chargement du module vidéo de la SDL
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        return EXIT_FAILURE;
    }

    // Ressource : Fenêtre et rendu
    SDL_Window* pWindow{ nullptr };
    SDL_Renderer* pRenderer{ nullptr };

    // Création d'une fenêtre, en 800x600, et visible
    if (SDL_CreateWindowAndRenderer(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }


    SDL_Event events;
    bool isOpen{ true };


    SDL_Rect rectangle1{20, 20, 100, 50};


    // Game loop
    while (isOpen)
    {
        // Input
        while (SDL_PollEvent(&events))
        {
```

```
            switch (events.type)
            {
            case SDL_QUIT:
                isOpen = false;
                break;
            }
        }


        // Rendering
        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255); // Choisir la couleur noir
        SDL_RenderClear(pRenderer); // Colorier en noir toutes la fenêtre

        draw_rectangle_not_fill(pRenderer, rectangle1, SDL_Color{ 0, 0, 255, 255 });

        SDL_RenderPresent(pRenderer); // Mise à jour de la fenêtre
    }

    // Libération des ressource en mémoire
    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    SDL_Quit();

    return EXIT_SUCCESS;
}

// Implementation

void draw_rectangle_fill(SDL_Renderer* renderer, const SDL_Rect& rectangle, const SDL
{
    SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, color.a);

    for (auto y = rectangle.y; y < rectangle.y + rectangle.h; y++)
    {
        for (auto x = rectangle.x; x < rectangle.x + rectangle.w; x++)
        {
            SDL_RenderDrawPoint(renderer, x, y);
        }
    }
}
void draw_rectangle_not_fill(SDL_Renderer* renderer, const SDL_Rect& rectangle, const
{
    SDL_SetRenderDrawColor(renderer, color.r, color.g, color.b, color.a);

    for (auto y = rectangle.y; y < rectangle.y + rectangle.h; y++)
    {
        for (auto x = rectangle.x; x < rectangle.x + rectangle.w; x++)
        {
            if(x == rectangle.x)
                SDL_RenderDrawPoint(renderer, x, y);
            if(x == rectangle.x + rectangle.w - 1)
                SDL_RenderDrawPoint(renderer, x, y);
            if(y == rectangle.y)
                SDL_RenderDrawPoint(renderer, x, y);
```

```
            if(y == rectangle.y + rectangle.h - 1)
                SDL_RenderDrawPoint(renderer, x, y);
        }
    }
}
```

Un peu d'explication s'exige, vous voyez que je fais commencer `y` a la valeur du champ `y` du `SDL_Rect` mais je fais un calcul lors de la condition de la boucle `for` je fais `y + rectangle.h` en effet ce calcul est nécessaire, pour pouvoir parcourir en hauteur, si vous ne le faite par vous aurez un bug (testez et vous verrez). Il faut gérer le cas où le rectangle n'est pas placer à l'origine.

Donc ce que je fais réellement c'est cette démarche intellectuelle :

1. Je souhaite placer mon rectangle avec des coordonnées x et y qui seront donc le point de départ de mon rectangle.
2. j'ai la largeur et la hauteur à gérer donc dans ma boucle ma position x et y, dois aller jusqu'à x + w et y + h.
3. Pour les conditions, je dessine mes pixels que dans 4 cas (la ligne de gauche, la ligne de droite, la ligne du bas et la ligne du haut) :
    - La ligne de gauche c'est quand y vaut la position x du rectangle
    - La ligne de droite c'est quand x vaut la largeur du rectangle qu'on souhaite afficher
    - La ligne en haut c'est quand y vaut la position y du rectangle
    - La ligne en bas c'est quand y vaut la hauteur du rectangle

Bon si tout ça vous semble un peu difficile, sachez alors que SDL vous propose deux fonctions pour dessiner vos rectangles, voici les deux prototypes :

```
int SDL_RenderDrawRect(SDL_Renderer*   renderer,
                       const SDL_Rect* rect)

int SDL_RenderFillRect(SDL_Renderer*   renderer,
                       const SDL_Rect* rect)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre et le rectangle.
- Elle retourne 0 en cas de succès ou une valeur négative en cas d'erreur

La première fonction `SDL_RenderDrawRect()` dessinera un rectangle non rempli. La deuxième fonction `SDL_RenderFillRect()` dessinera un rectangle rempli.

Sachez qu'il existe leur variante `SDL_RenderDrawRects()` et `SDL_RenderFillRects()`, j'en ai parlé de ces fonctions dans la correction de l'exercice sur les `SDL_Points` vous avez ces fonctions qui permettent de dessiner plusieurs points en un seul appel de fonction. Ben ici c'est pareil sauf que ce sont des rectangles.

Je vous laisse aller voir la documentation :

- Documentation `SDL_RenderDrawRects()` : https://wiki.libsdl.org/SDL_RenderDrawRects
- Documentation `SDL_RenderFillRects()` : https://wiki.libsdl.org/SDL_RenderFillRects

En effet j'utilise souvent la fonction `SDL_RenderDrawRects()` en mode debug, car ça me permet de voir les **colliders box** comme ceci :

Imaginons que je veux :

- couleur : Bleu (REMPLIE)
- Position : centrer au milieu de l'écran
- Dimension : 300x100

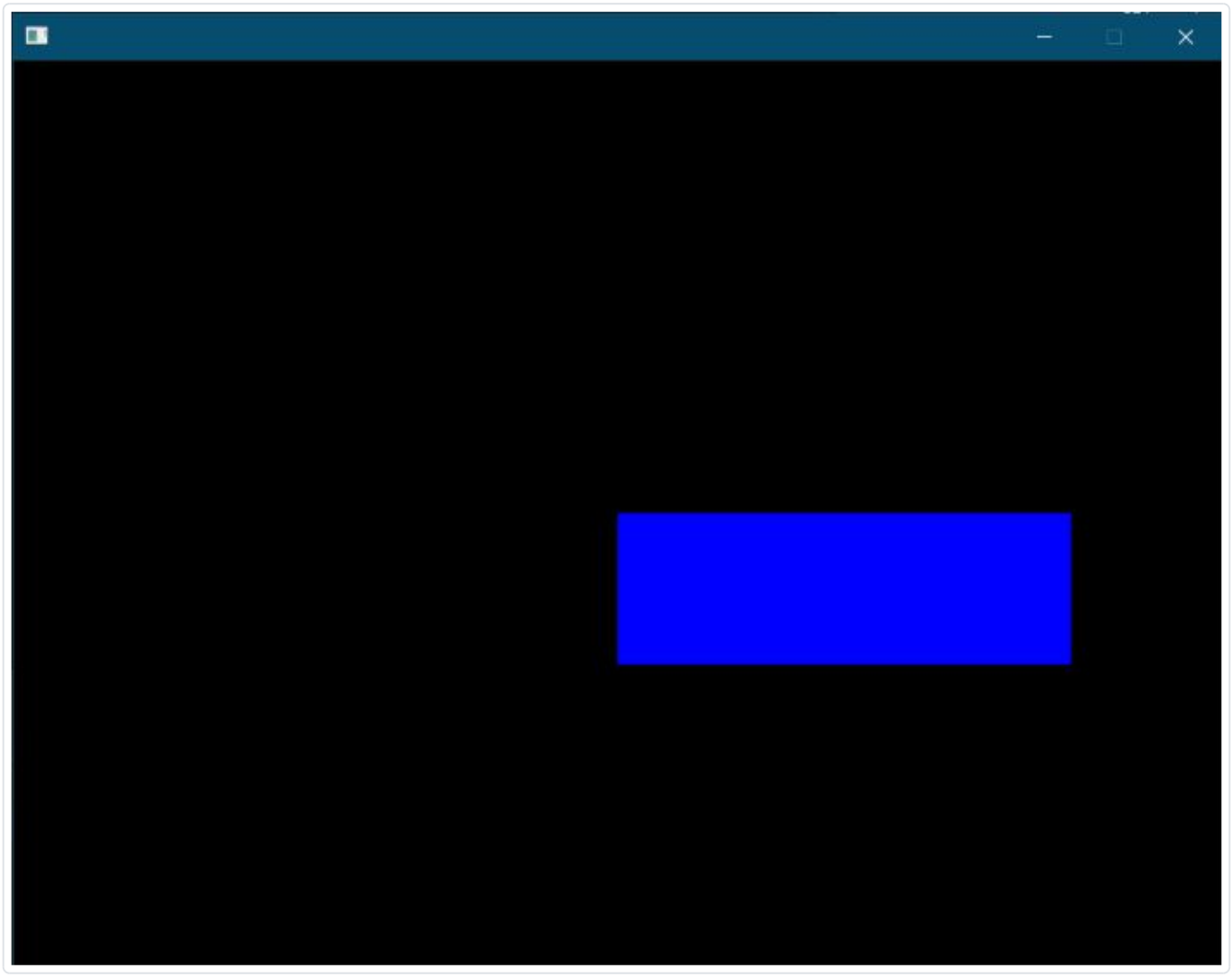
Voilà à quoi va ressembler notre code :

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

#include <cstdlib>

template<typename T>
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };
```

```
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(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }


    SDL_Event events;
    bool isOpen{ true };

    // Définition du rectangle
    SDL_Rect rectangle{ WIDTHSCREEN<int> / 2, HEIGHTSCREEN<int> / 2, 300, 100 };


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


        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);

        // Dessin du rectangle
        SDL_SetRenderDrawColor(pRenderer, 0, 0, 255, 255);
        SDL_RenderFillRect(pRenderer, &rectangle);

        SDL_RenderPresent(pRenderer);
    }



    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
```

```
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

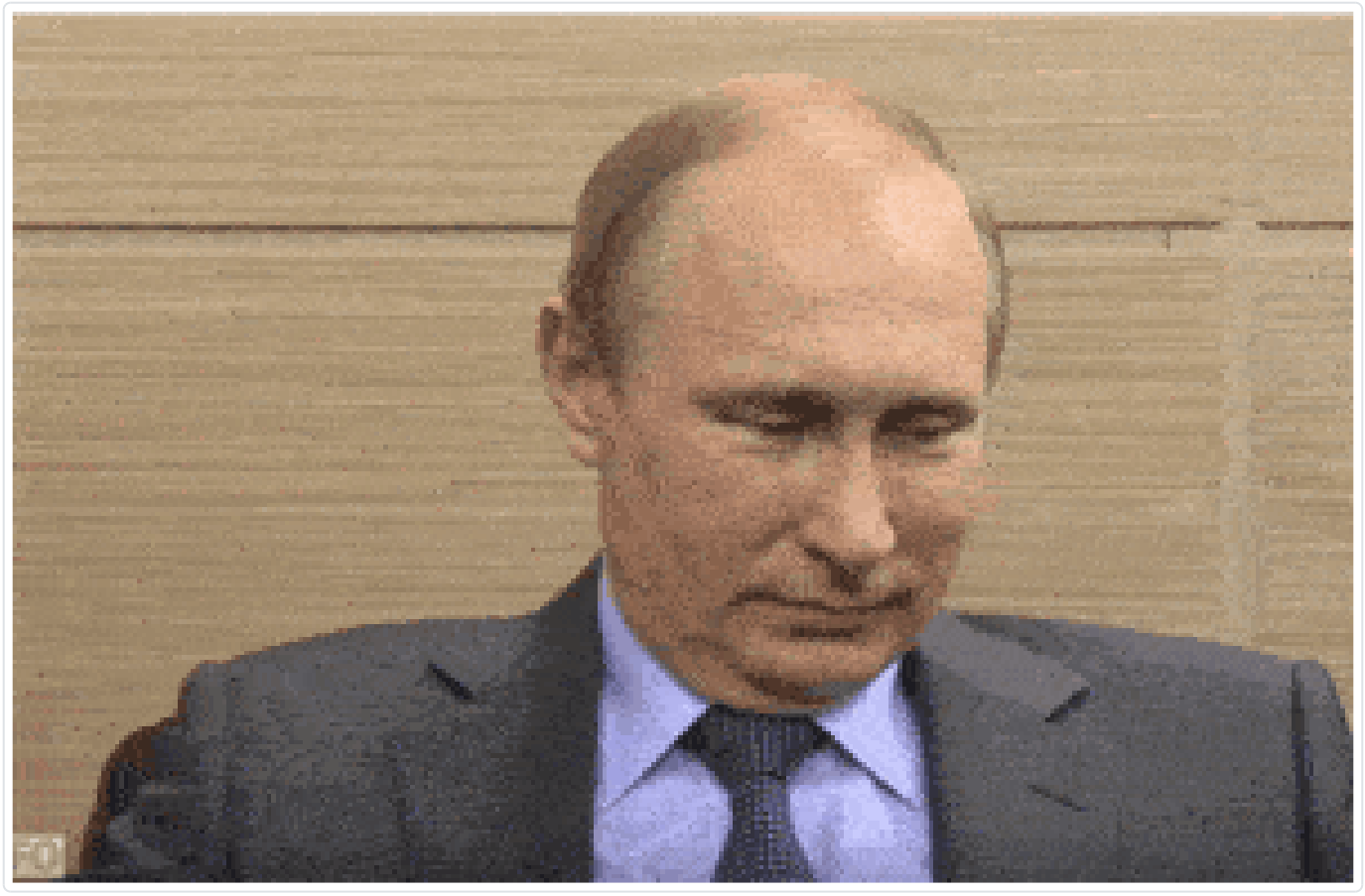
Voici le résultat du rendu :



Bon, nous voyons que c'est très simple rien de compliqué.

Vous : Tu me prend pour un con il n'est pas centré CYKA BLYAT !

Moi : : Ok, ben ça c'est normal, il est centré mais c'est juste que l'origine du repère est situé en haut a gauche. j'en ai déja parlé au début de l'article souvenez-vous. Pour cela il faut changer la position x et y du rectangle et le faire intelligemment. Il faut décaler a gauche le x de la moitié de la largeur du rectangle et déclarer en hauteur de la moitié de la hauteur du rectangle.

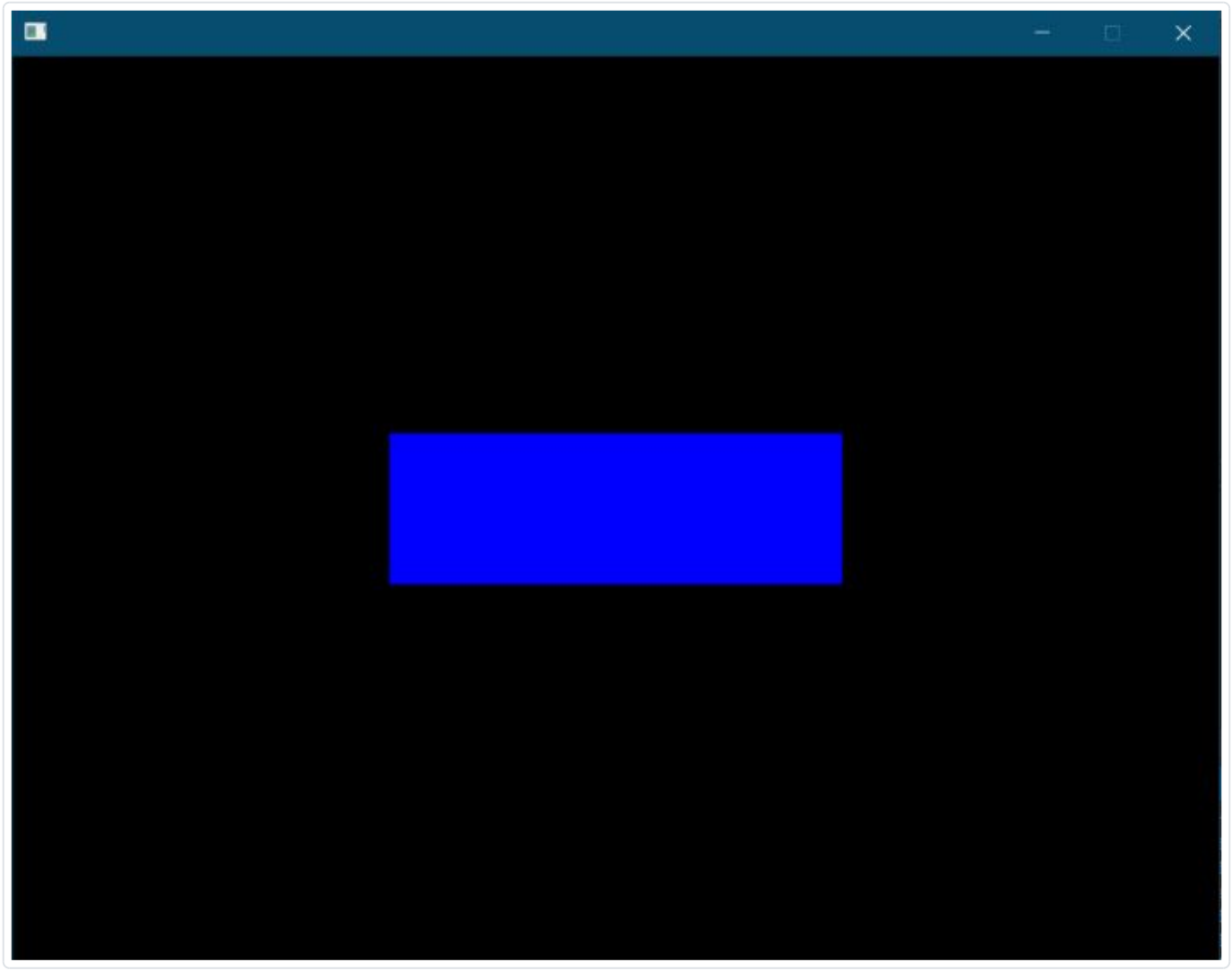


voici les calculs :

```
rectangle.x = WIDTHSCREEN<int> / 2 - rectangle.w / 2;
rectangle.y = HEIGHTSCREEN<int> / 2 - rectangle.h / 2;
```

Avec ces calcules, on retire bien la moitié de la largeur et de la hauteur :)

Voici le résultat :



Les lignes

Bon, il nous reste plus qu'à voir comment **tracer des lignes**. Voyons mathématiquement comment on peut tracer des droites.

En mathématiques :

- Méthode 1 : il est possible de tracer des droites en ayant l'ordonnée à l'origine et le coefficient directeur.
- Méthode 2 : il faut tous simplement placer deux points dans le repaire et les relier entre eux.

La méthode choisie par la SDL est la méthode 2, en effet pour tracer deux points, il faut deux `SDL_Point` la coordonnée du point A et la coordonnée du point B.

Nous allons dessiner un segment passant par l'origine du repère et qui se termine dans le coin en bas à droite de la fenêtre. Pour cela :

1. Déclarons deux `SDL_Point`, nommé point A et point B.
2. Donnez les bonnes coordonnées au point A et point B.
3. Ensuite choisir la couleur verte.
4. Dessinons en appelant la fonction `SDL_RenderDrawLine()` voici son prototype :

```
int SDL_RenderDrawLine(SDL_Renderer* renderer,
                       int           x1,
                       int           y1,
                       int           x2,
                       int           y2)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est la position en x (Horizontal) du premier point.
- Le troisième paramètre est la position en y (Vertical) du premier point.
- Le quatrième paramètre est la position en x (Horizontal) du deuxième point.
- Le cinquième paramètre est la position en y (Vertical) du deuxième point.

```
// Définition des deux Point
SDL_Point pointA;
pointA.x = 0;
pointA.y = 0;

SDL_Point pointB;
pointB.x = WIDTHSCREEN<int>;
pointB.y = HEIGHTSCREEN<int>;
```

```
SDL_SetRenderDrawColor()(pRenderer, 0, 0, 0, 255);
SDL_RenderClear(pRenderer);

// Dessin de la ligne
SDL_SetRenderDrawColor(pRenderer, 0, 255, 0, 255);
SDL_RenderDrawLine(pRenderer, pointA.x, pointA.y, pointB.x, pointB.y);
SDL_RenderPresent(pRenderer);
```

Voici le résultat final en code :

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


#include <cstdlib>

template<typename T>
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };


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(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }


    SDL_Event events;
    bool isOpen{ true };

    // Définition des deux Point
    SDL_Point pointA;
    pointA.x = 0;
    pointA.y = 0;
```

```
    SDL_Point pointB;
    pointB.x = WIDTHSCREEN<int>;
    pointB.y = HEIGHTSCREEN<int>;



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


        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);

        // Dessin de la ligne
        SDL_SetRenderDrawColor(pRenderer, 0, 255, 0, 255);
        SDL_RenderDrawLine(pRenderer, pointA.x, pointA.y, pointB.x, pointB.y);
        SDL_RenderPresent(pRenderer);
    }



    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Nous avons vu pas mal de fonctions de dessins. on en a pas encore fini. Reposez-vous. n'attaquez pas tout en même ce chapitre sera long très long.

La transparence

Cette partie sera très courte, nous allons apprendre à **déssiner de façon transparente**. Vous verrez c'est très simple .

Nous constatons qu'à chaque fois que j'écris `SDL_SetRenderDrawColor()` le paramètre alpha est toujours mis à 255. Pour ceux qui ont essayé de le mettre sur une autre valeur, vous constaterez que rien ne change, en effet c'est normal il faut **activer l'option transparence en SDL** car par défaut elle est désactivé. Il faut dire à la SDL, quel type de transparence il faut utiliser, il en existe plusieurs. Il existe plusieurs manières de gérer de la transparence, maintenant la question à se poser est : "Quelle formule mathématique allons-nous utiliser pour la transparence ?"

On peut déjà par se commencer la question suivante : Qu'est-ce qu'un pixel transparent ?

En réalité, pour rendre un pixel transparent il faut fusionner la couleur du pixel de dessous au pixel de dessus. Le pixel de dessous c'est le pixel déjà affiché et le pixel de dessus c'est celui qu'on veut afficher.

Si vous mettez un nombre au composant alpha, il indiquera la manière de prendre plus ou moins en compte la couleur du pixel que vous souhaitez afficher. Plus le composant alpha tend vers 0 plus se sera transparent et plus elle tend vert 255 plus elle sera opaque.

Il y a au total 4 modes pour **gérer la transparence**, voici un tableau qui montre les modes possibles :

SDL_BLENDMODE_NONE	no blending
	dstRGBA = srcRGBA
SDL_BLENDMODE_BLEND	alpha blending
	dstRGB = (srcRGB * srcA) + (dstRGB * (1-srcA))
	dstA = srcA + (dstA * (1-srcA))
SDL_BLENDMODE_ADD	additive blending
	dstRGB = (srcRGB * srcA) + dstRGB
	dstA = dstA
SDL_BLENDMODE_MOD	color modulate
	dstRGB = srcRGB * dstRGB
	dstA = dstA

- Le premier utilise cette formule : Ce qui rentre est égal a ce qui sort. En gros elle dit juste que le pixel que vous voulez afficher et celui qui va être affiché. c'est celui par défaut donc il ne prend pas en compte la transparence.
- Les autres utilisent des formules mathématiques qui sont écrites sur le tableau, on ne va pas se soucier à quoi correspondent ces formules en réalité on s'en fiche un peu :p

Si vous voulez plus d'explication sur comment fonctionne la transparence en infographie, je vous conseille de lire cet article en anglais.

Pour pouvoir utiliser la transparence en SDL, il faut specifier à la SDL quelle est la façon de faire. Pour cela on va choisir par exemple la méthode `SDL_BLENDMODE_BLEND` et disons à la SDL que nous allons l'utiliser via la fonction `SDL_SetRenderDrawBlendMode()`.

Voici son prototype :

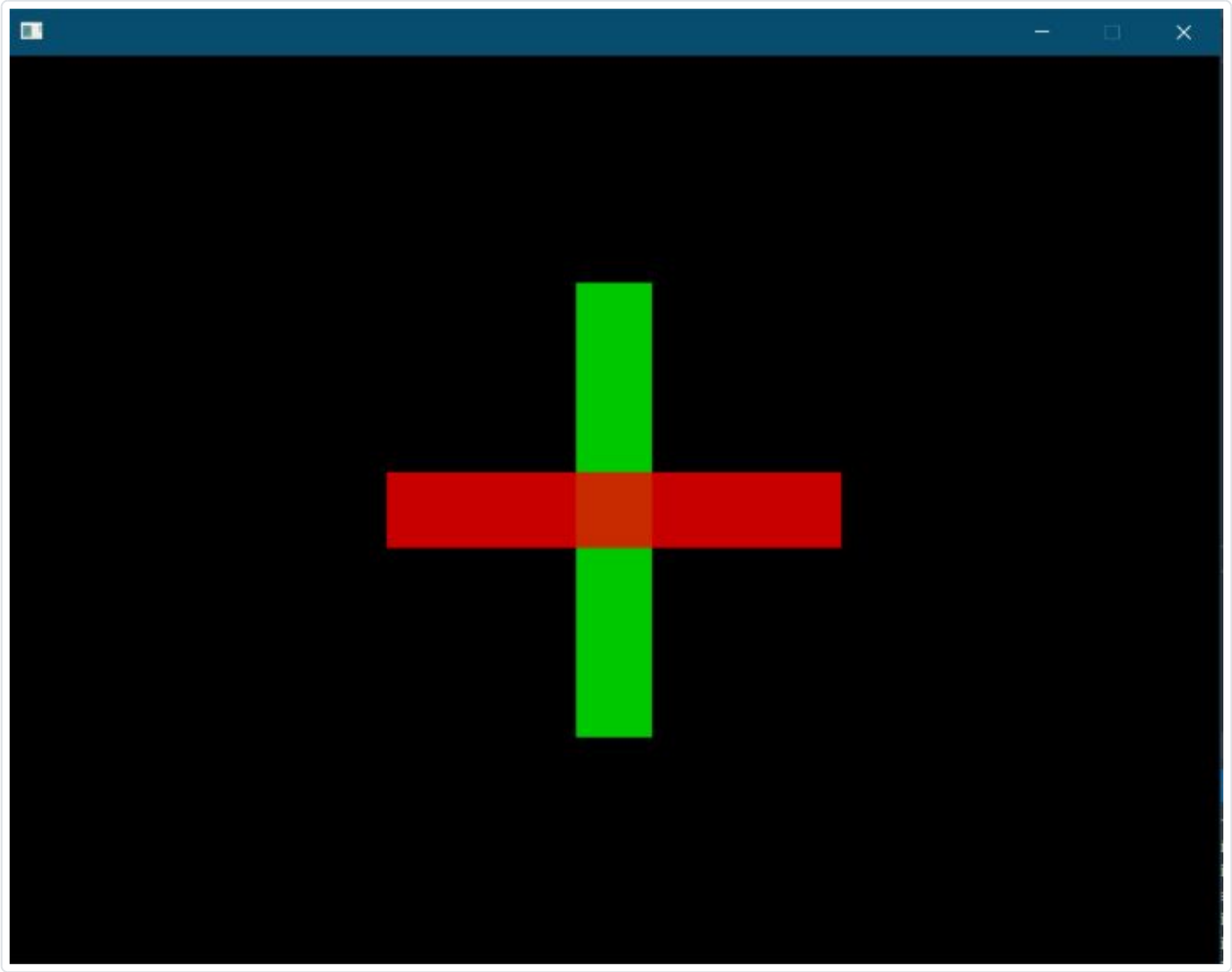
```
int SDL_SetRenderDrawBlendMode(SDL_Renderer* renderer,
                               SDL_BlendMode blendMode)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est le mode de fusion de couleur a utilisé.
- Elle retourne 0 en cas de succès ou une valeur négative si elle a échoué.

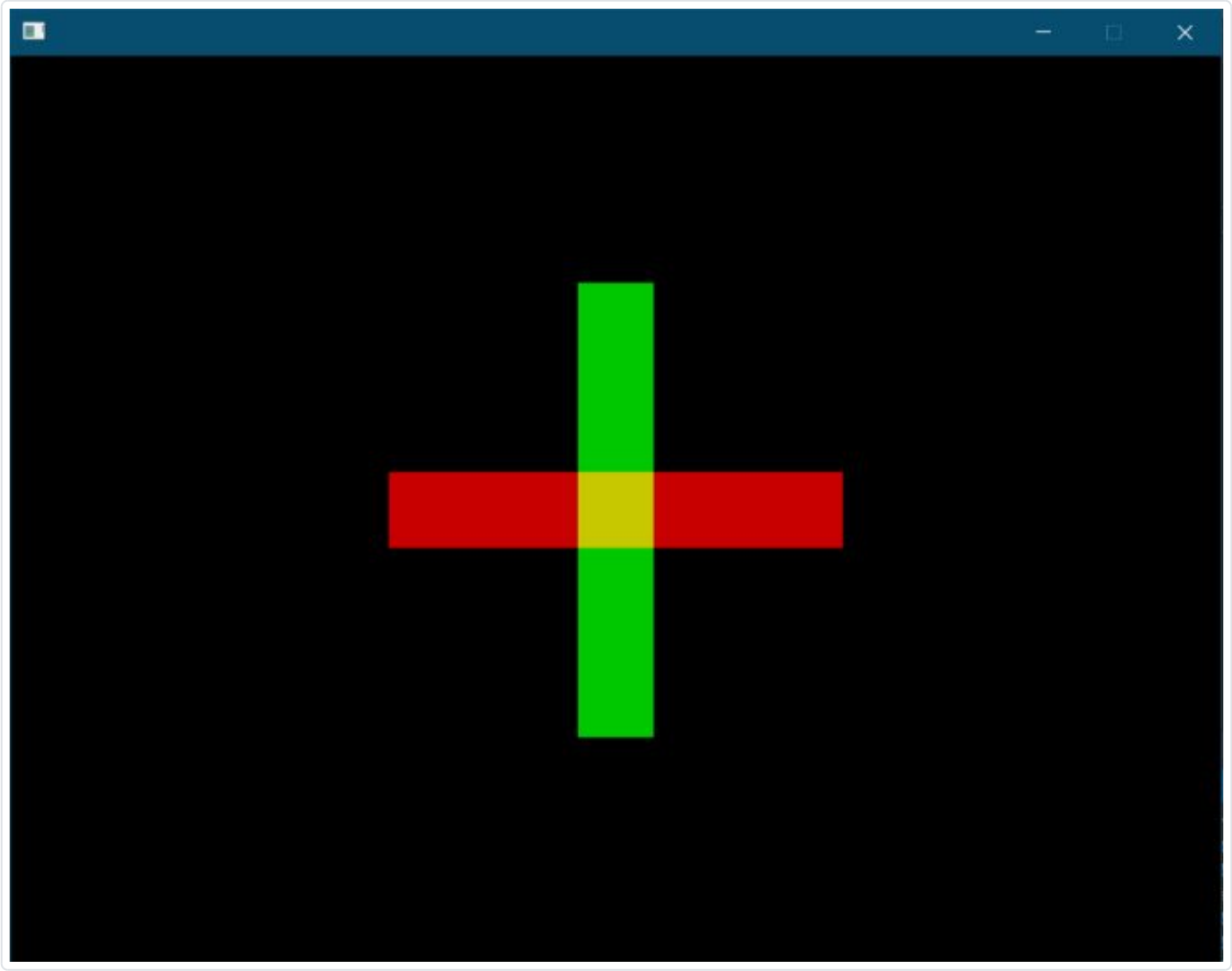
Cette fonction je vais l'appeler avant la Game Loop comme ceci :

```
SDL_SetRenderDrawBlendMode()(pRenderer, SDL_BLENDMODE_BLEND);

while (isOpen)
{
     ...
}
```

lorsque j'affiche un rectangle vert et rouge qui ont une transparence alpha mise à 255/2 et avec le mode `SDL_BLENDMODE_BLEND` j'obtiens ceci :



lorsque j'affiche un rectangle vert et rouge qui ont une transparence alpha mise à 255/2 et avec le mode `SDL_BLENDMODE_ADD` j'obtiens ceci :



Vous pouvez constater la différence, on remarque que le deuxième la couleur est plus vive.

Les collisions

Il est maintenant tant de découvrir les collisions et voir ce que la SDL a dans son ventre. SDL sait faire les types de collision suivante :

1. **Collision d'un rectangle avec un point**.
2. **Collision d'un rectangle avec un rectangle**.

3. **Collision d'un rectangle avec une ligne**.

Il est temps de voir ça. commençons par les intersections `SDL_Rect` et `SDL_Point` :

il existe une fonction pour ce type de collision, et voici son prototype :

```
SDL_bool SDL_PointInRect(const SDL_Point* p,
                         const SDL_Rect*  r)
```

- Le premier paramètre est un pointeur sur une structure `SDL_Point`.
- Le deuxième paramètre est un pointeur sur une structure `SDL_Point`.
- Elle retourne `SDL_TRUE` s'il y a collision sinon `SDL_FALSE` si aucune collision

Maintenant, avec uen collision entre une structure `SDL_Rect` et une même structure `SDL_Rect`, il existe deux fonctions, voici la première :

```
SDL_bool SDL_HasIntersection(const SDL_Rect* A,
                             const SDL_Rect* B)
```

- Le premier paramètre est un pointeur sur une structure `SDL_Rect`.
- Le deuxième paramètre est un pointeur sur une structure `SDL_Rect`.
- Elle retourne `SDL_TRUE` s'il y a collision sinon `SDL_FALSE` si aucune collision

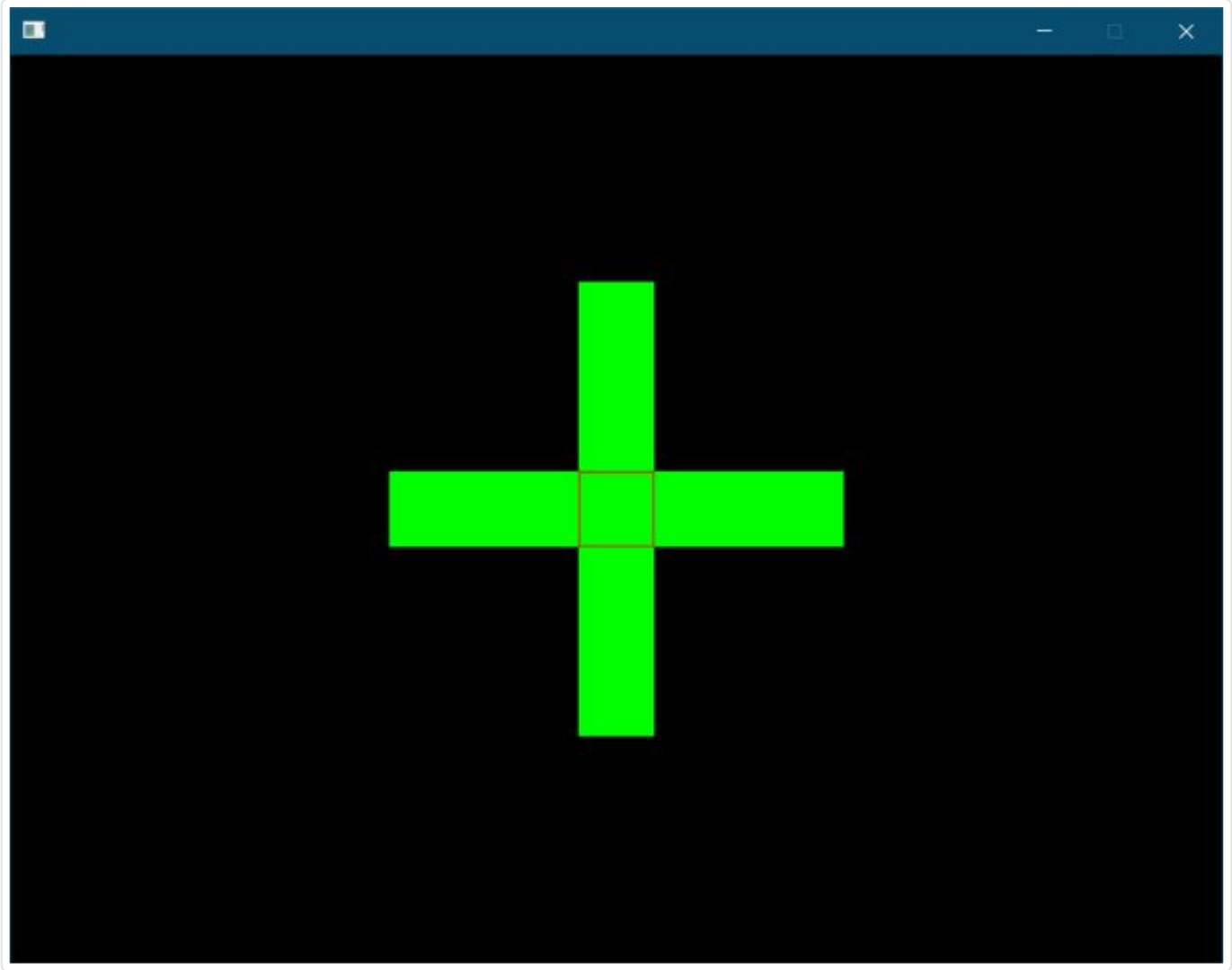
la deuxième fonction est :

```
SDL_bool SDL_IntersectRect(const SDL_Rect* A,
                           const SDL_Rect* B,
                           SDL_Rect*       result)
```

- Le premier paramètre est un pointeur sur une structure `SDL_Rect`.
- Le deuxième paramètre est un pointeur sur une structure `SDL_Rect`.

- Elle retourne `SDL_TRUE` s'il y a collision sinon `SDL_FALSE` si aucune collision

Voici une explication sur ce troisième paramètre, En effet SDL a trouvé bon, de pouvoir récupérer le rectangle de la collision. Voici une image qui permet de mieux comprendre ce que permet de récupérer ce paramètre :



Sur l'image, on aperçoit la collision formée par les deux rectangles, que je récupère et je redessine avec une bordure rouge, voici le code source :

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


#include <cstdlib>

template<typename T>
```

```
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };


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(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }


    SDL_Event events;
    bool isOpen{ true };

    SDL_Rect rectangle1{ 0, 0, 300, 50 };
    rectangle1.x = WIDTHSCREEN<int> / 2 - rectangle1.w / 2;
    rectangle1.y = HEIGHTSCREEN<int> / 2 - rectangle1.h / 2;



    SDL_Rect rectangle2{ 0, 0, 50, 300 };
    rectangle2.x = WIDTHSCREEN<int> / 2 - rectangle2.w / 2;
    rectangle2.y = HEIGHTSCREEN<int> / 2 - rectangle2.h / 2;

    SDL_Rect collisionRectangle; // Rectangle par le quel je vais récupérer la collis

    SDL_SetRenderDrawBlendMode(pRenderer, SDL_BLENDMODE_NONE);

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

```
            }
        }

        if (SDL_IntersectRect(&rectangle2, &rectangle1, &collisionRectangle))
        {
            SDL_Log("[DEBUG] Il y 'a collision ici ..."); // j'affiche ce message lor
        }

        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);



        // Dessin du rectangle
        SDL_SetRenderDrawColor(pRenderer, 0, 255, 0, 255 / 2);
        SDL_RenderFillRect(pRenderer, &rectangle2);

        SDL_SetRenderDrawColor(pRenderer, 0, 255, 0, 255/2);
        SDL_RenderFillRect(pRenderer, &rectangle1);

        // Je dessine le rectangle former par la collision
        SDL_SetRenderDrawColor(pRenderer, 255, 0, 0, 255); // couleur rouge
        SDL_RenderDrawRect(pRenderer, &collisionRectangle); // Je dessine le rectangl

        SDL_RenderPresent(pRenderer);
    }



    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

vous voyez c'est assez simple à utiliser, c'est vraiment bonus.

Maintenant, voyons voir une collision avec `SDL_Rect` avec une ligne :

```
SDL_bool SDL_IntersectRectAndLine(const SDL_Rect* rect,
                                  int*            X1,
                                  int*            Y1,
                                  int*            X2,
                                  int*            Y2)
```

- Le premier paramètre est un pointeur sur le rectangle.

- Le deuxième paramètre est l'adresse de la position horizontale (x) du premier point.
- Le troisième paramètre est l'adresse de la position verticale (y) du premier point.
- Le quatrième paramètre est l'adresse de la position horizontale (x) du deuxième point.
- Le cinquième paramètre est l'adresse de la position verticale (y) du deuxième point.
- Elle retourne `SDL_TRUE` s'il y a collision sinon `SDL_FALSE` si aucune collision

Voila, nous avons vu toutes les fonctions de collision qui existent en SDL.

Les textures

Nous allons nous intéresser aux textures, souvenez-vous on en a vu vite fait lors de la création du `SDL_Renderer`, on avait appris qu'il existait des flags possibles et notamment un flag nommé `SDL_RENDERER_TARGETTEXTURE`.

Nous allons récupérer quelque information à propos du `SDL_Renderer`, et pour ça nous allons utiliser la fonction `SDL_GetRenderInfo()` voici son prototype :

```
int SDL_GetRendererInfo(SDL_Renderer*     renderer,
                        SDL_RendererInfo* info)
```

- Elle prend le rendu de fenêtre.
- Elle prend un pointeur sur une `SDL_RendererInfo`.

Mais avant tout qu'est ce qu'un `SDL_RendererInfo` ? C'est une structure, son nom est bien choisi car elle contient toutes les informations de la `SDL_Renderer`.

```
typedef struct SDL_RendererInfo
{
    const char *name;           /**< The name of the renderer */
    Uint32 flags;               /**< Supported ::SDL_RendererFlags */
    Uint32 num_texture_formats; /**< The number of available texture formats */
    Uint32 texture_formats[16]; /**< The available texture formats */
    int max_texture_width;      /**< The maximum texture width */
    int max_texture_height;     /**< The maximum texture height */
} SDL_RendererInfo;
```

Elle nous affiche quelques informations, le paramètre flags, ce sont les flags quels contient.

Nous allons vérifier si le flag `SDL_RENDERER_TARGETTEXTURE` est présent. Voici le code

```
SDL_RendererInfo infoRenderer;
SDL_GetRendererInfo(pRenderer, &infoRenderer);

if (infoRenderer.flags & SDL_RENDERER_ACCELERATED)
{
    SDL_Log("Le rendu est gérer par la carte graphique...");

}
if (infoRenderer.flags & SDL_RENDERER_SOFTWARE)
{
    SDL_Log("Le rendu est gérer par la carte graphique...");
}

if (infoRenderer.flags & SDL_RENDERER_TARGETTEXTURE)
{
    SDL_Log("Le rendu est autoriser sur des texture...");
}
```

Résultat :

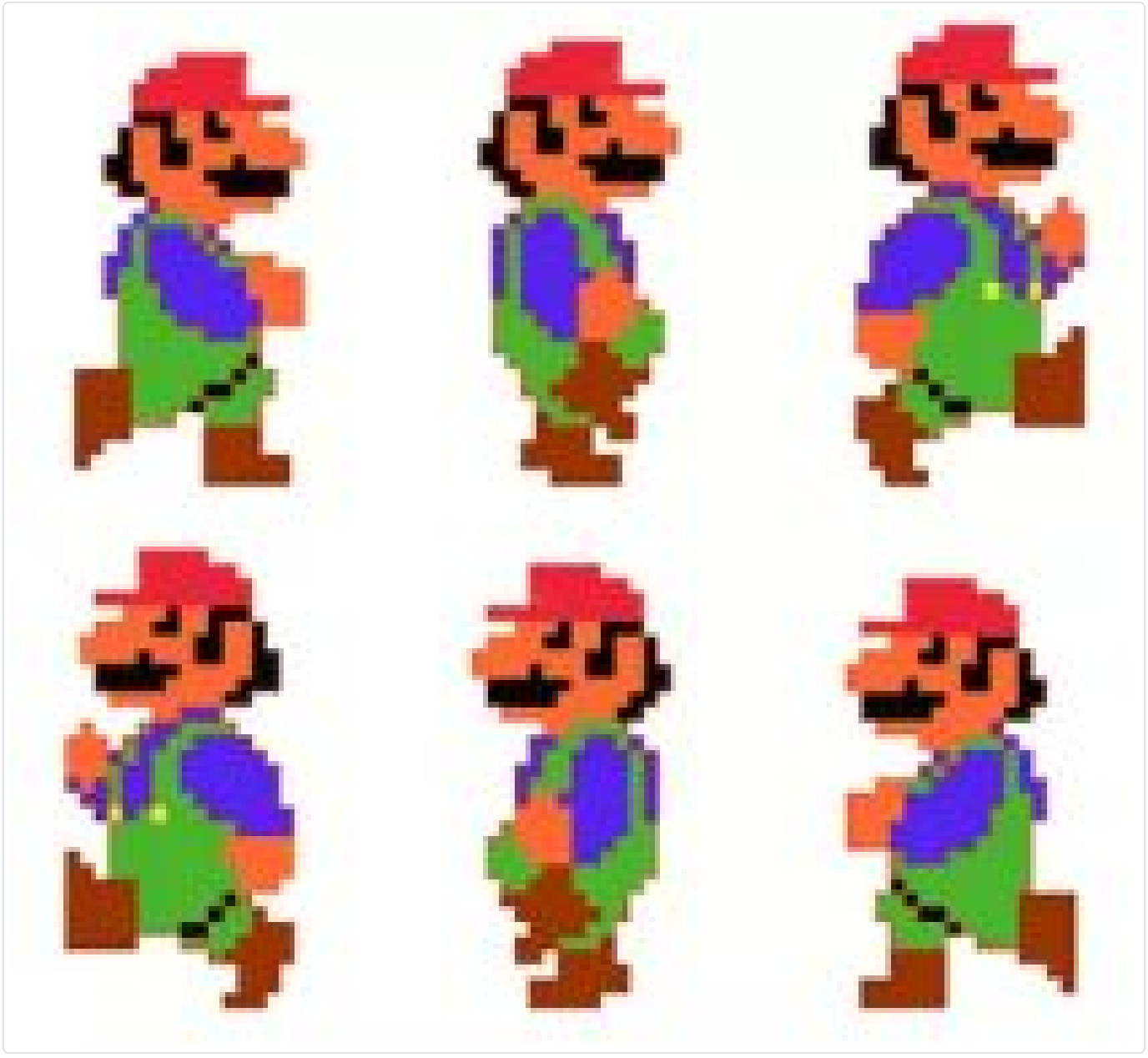
```
INFO: Le rendu est géré par la carte graphique...
INFO: Le rendu est autoriser sur des texture...
```

Eh oui il n'y a pas besoin de le rajouter. peu importe si vous utilisez la fonction `SDL_CreateWindow()` et `SDL_CreateRenderer()` ensemble ou `SDL_CreateWindowAndRenderer()`.

Eh si vous affichez le nom du système de rendu vous verrez marquer sous Windows dans man cas "Direct3D". Pour info, Direct3D est ce que Windows propose pour faire des affichages en rendre 3D nous allons bien sûr ne pas étudier Direct3D c'est un cours de SDL mais il faut savoir que SDL peut utiliser certaine chose venant d'autre part, donc si vous êtes trop curieux vous en avait pas fini et vous serez en sueur comme ce mec :

Bon maintenant nous avons appris que le flag `SDL_RENDERER_TARGETTEXTURE` est déjà présent pas besoin donc de le rajouter. Ça veut dire qu'on peut faire du rendu sur une structure `SDL_Texture`.

Nous allons par la suite **apprendre à charger des images**, et pour les afficher on aura besoin de la structure `SDL_Texture`, comme par exemple l'affichage du sprite Mario



ou bien carrément une texture (pour faire par exemple une map) grâce à la structure `SDL_Texture`.

La **différence entre un sprite et une texture**, c'est qu'un sprite et censé être élement d'un jeu qu'ont peut animer et qui se déplace, à l'inverse d'une texture dans un jeu-vidéo qu'on plaque quelque part pour afficher par exemple le sol.

Un sprite et texture restent tout de même des images et en tant qu'image ils ont des choses en commun, tel que

1. C'est deux images. Ils ont en commun leur tableau de pixels. c'est-à-dire que nous pouvons voir cette image en tant que tableau de deux dimensions et qui contiennent des pixels ayant un certain nombre de lignes et un certain nombre de colonnes.
2. C'est deux images ont un sens de l'information exemple RGB, où BGR (oui ça existe d'inverser comme le format BMP c'est du BGR)
3. C'est deux images ont le nombre d'octets par pixel généralement de 3 ou 4 (3 sans le composant alpha, 4 avec le composant alpha)
4. Le nombre de bits par pixels, généralement 8 bits soit 1 octet.
5. Plein d'autres choses comme largeur, et hauteur de l'image etc.

Bon, maintenant, il faut parler code...

Pour créer une `SDL_Texture` il faut utiliser la fonction `SDL_CreateTexture`, voici son prototype :

```
SDL_Texture* SDL_CreateTexture(SDL_Renderer* renderer,
                               Uint32        format,
                               int           access,
                               int           w,
                               int           h)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est le format.
- Le troisième paramètre est l'accès à la texture.
- Le quatrième paramètre est la largeur de la texture.
- Le cinquième paramètre est la hauteur de la texture.

- Elle retourne un pointeur sur une `SDL_Texture` et `nullptr` si ça n'a pas pu crée la texture.

Un peu d'explication sur le troisième paramètre qui est le format. Il permet d'informer la SDL comment seront stocker les pixels dans la texture. Si on choisi de stocker sous cette forme "RGBA", on stockera alors 4 octets RGBA :

- R = 1 octet
- G = 1 octet
- B = 1 octet
- A = 1 octet

Pour cela nous avons une enumeration, qui contient tous les formats, et ce tableau est assez long donc je vous exhorte à aller voir la documentation suivante : https://wiki.libsdl.org/SDL_PixelFormatEnum

nous allons ici utiliser le flag `SDL_PIXELFORMAT_RGBA8888`

Bon maintenant, voyons voir le quatrième paramètre qui est l'accès à la texture. Voici les valeurs possibles :

SDL_TEXTUREACCESS_STATIC	changes rarely, not lockable
SDL_TEXTUREACCESS_STREAMING	changes frequently, lockable
SDL_TEXTUREACCESS_TARGET	can be used as a render target

1. `SDL_TEXTUREACCESS_STATIC` : signifie que la texture ne change pas souvent et donc elle ne peut pas être verrouillable.

2. `SDL_TEXTUREACCESS_STREAMING` : Signifie que la texture changera souvent et qu'elle est verrouillable.
3. `SDL_TEXTUREACCESS_TARGET` : signifie que la texture peut être une cible de rendu.

Bon qu'est ça veut dire être **texture verrouillable** ? Ça veut simplement dire si il est possbile de changer le tableau de pixels, donc changer les couleurs du tableau par exemple pour la rendre plus sombre ou plus claire. (Sympa pour un logiciel d'édition d'image). Et être ciblé du rendu, veut dire qu'on peut utiliser le rendu pour dessiner sur la texture et dessiner comme si on dessinerait sur la fenêtre. Je vais vous passer directement la fonction qui permet de libérer en mémoire la `SDL_Texture` voici son prototype ;

```
void SDL_DestroyTexture(SDL_Texture* texture)
```

- le premier paramètre prend un pointeur sur une `SDL_Texture`.
- Elle ne retourne rien du tout.

Bon maintenant il faut créer une texture. Nous allons choisir ce flag `SDL_PIXELFORMAT_RGBA8888` pour le format et `SDL_TEXTUREACCESS_TARGET` pour l'accès à la texture.

Voici le code

```
SDL_Texture* pTexture = SDL_CreateTexture(pRenderer, SDL_PIXELFORMAT_RGBA8888, SDL_TE
```

N'oubliez pas de libérer en mémoire la texture :

```
SDL_DestroyTexture(pTexture);
SDL_DestroyRenderer(pRenderer);
SDL_DestroyWindow(pWindow);
SDL_Quit();
```

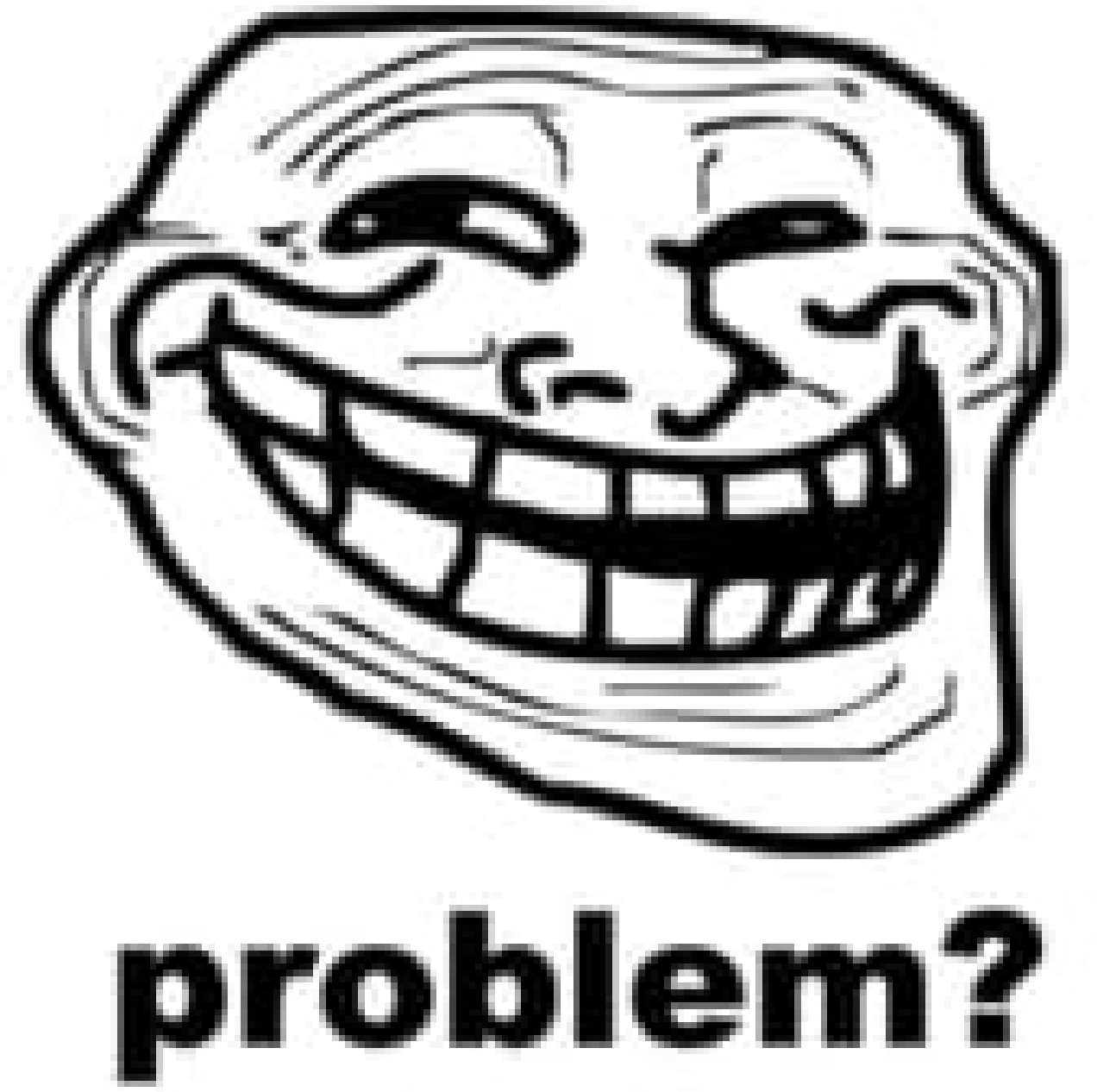
```
return EXIT_SUCCESS;
```

Et aussi, il est bon de vérifier les erreurs, moi je le ne fait pas car ça n'a pas vocation à être en production mais vous pour apprendre il faut que sa devienne un réflexe (une seconde nature).

Bon, une fois que j'ai ma `SDL_Texture` qui est créée, nous pouvons dessiner avec, c'est parti, nous allons dessiner un rectangle.

Vous : Tu te fous de ma gueule toi.!!

Moi : Non.



Bon, pour faire mieux nous allons dessiner un rectangle de couleur bleue et des lignes qui se croiseront au centre de couleur verte.

Pour dessiner sur une texture, il faut dire au rendu qu'il va dessiner sur la texture, puis ensuite nous pouvons dessiner comme on le ferait normalement sur une fenêtre.

Ensuite quand aura finit notre dessin, il faut repasser la main sur la fenêtre. Pour dire au rendu, qu'il va dessiner sur la texture, on doit utiliser une fonction qui s'appelle `SDL_SetRenderTarget()` voici son prototype :

```
int SDL_SetRenderTarget(SDL_Renderer* renderer,
                        SDL_Texture*  texture)=
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est la texture.
- Elle retourne 0, si elle a réussi sinon une valeur négative.

Nous allons l'utiliser comme ceci :

```
SDL_Texture* pTexture = SDL_CreateTexture(pRenderer, SDL_PIXELFORMAT_RGBA8888, SDL_TE
SDL_SetRenderTarget(pRenderer, pTexture);
```

Ensuite on dessine, ce qui donne :

```
SDL_Texture* pTexture = SDL_CreateTexture(pRenderer, SDL_PIXELFORMAT_RGBA8888, SDL_TE

SDL_SetRenderTarget(pRenderer, pTexture);

SDL_SetRenderDrawColor(pRenderer, 0, 0, 255, 255);
SDL_RenderClear(pRenderer);

SDL_SetRenderDrawColor(pRenderer, 0, 100, 0, 255);
SDL_RenderDrawLine(pRenderer, 0, 0, 100, 100);
SDL_RenderDrawLine(pRenderer, 100, 0, 0, 100);
```

Une fois qu'on a dessiné, il faut dire au rendu de prendre en compte notre dessin, et pour cela on utilisera la fonction `SDL_SetRenderTarget()` mais cette fois-ci on mettra le paramètre en deuxième paramètre un `nullptr`

```
SDL_SetRenderTarget(pRenderer, pTexture);

SDL_SetRenderDrawColor(pRenderer, 0, 0, 255, 255);
SDL_RenderClear(pRenderer);
```

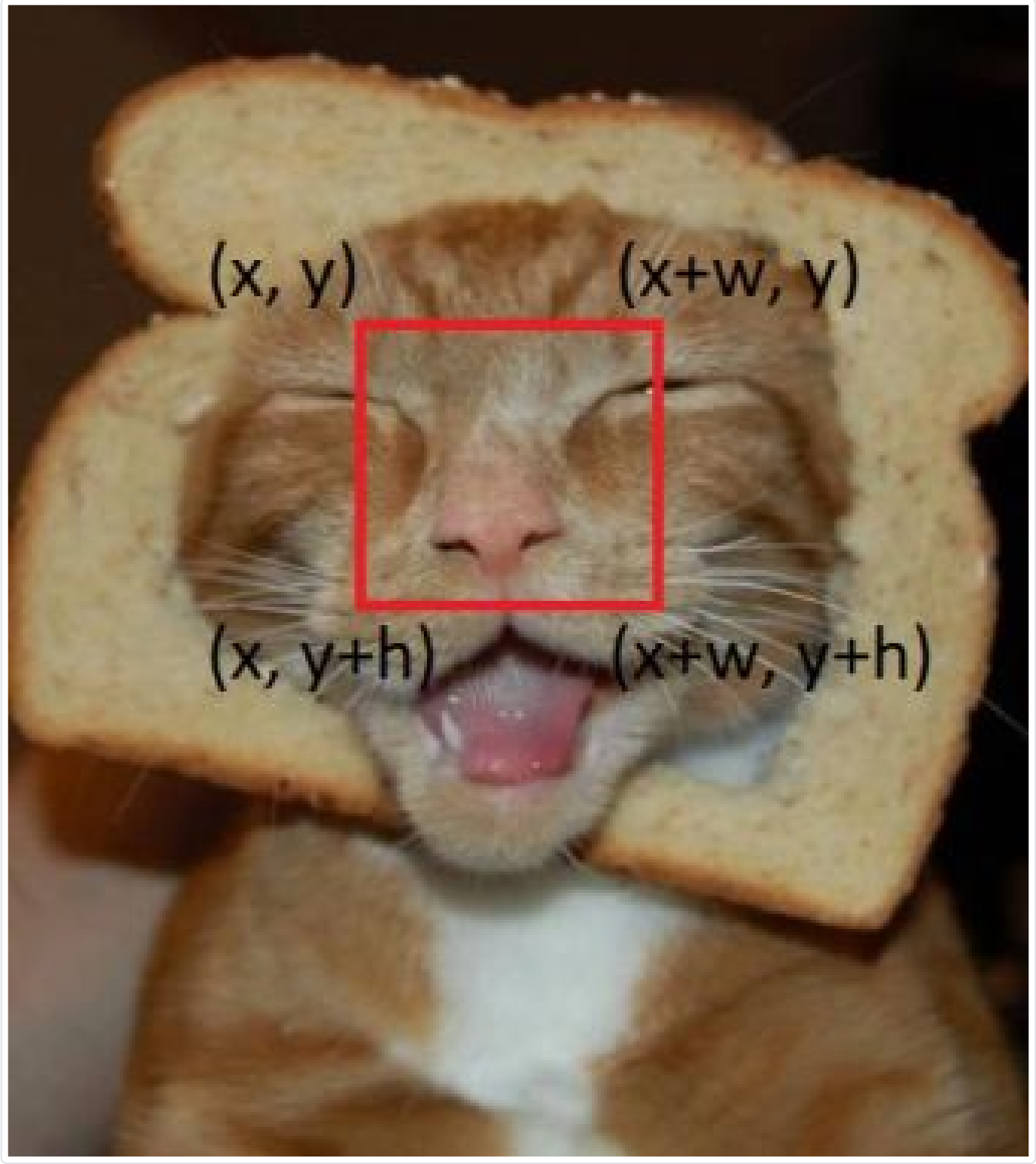
```
SDL_SetRenderDrawColor(pRenderer, 0, 255, 0, 255);
SDL_RenderDrawLine(pRenderer, 0, 0, 100, 100);
SDL_RenderDrawLine(pRenderer, 100, 0, 0, 100);

SDL_SetRenderTarget(pRenderer, nullptr);
```

Bon maintenant, ce que vous avez fait c'est bien, mais il manque juste à afficher la texture dans la fenêtre et pour cela il existe une fonction `SDL_RenderCopy()` qui va copier votre texture sur l'écran, voici son prototype :

```
int SDL_RenderCopy(SDL_Renderer*   renderer,
                   SDL_Texture*    texture,
                   const SDL_Rect* srcrect,
                   const SDL_Rect* dstrect)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est la texture.
- Le troisième paramètre est le rectangle source.
- Le quatrième paramètre est le rectangle de destination.
- Elle retourne 0 si elle réussit ou une valeur négative en cas d'erreurs

Bon il est bon de vous expliquer les deux derniers paramètres car il est important que vous les compreniez. Le rectangle, source signifie qu'elle est partie du rectangle à prendre en compte. Si vous souhaitez par exemple prendre que la partie du nez de ce chat vous aurez mis comme rectangle source ceci :

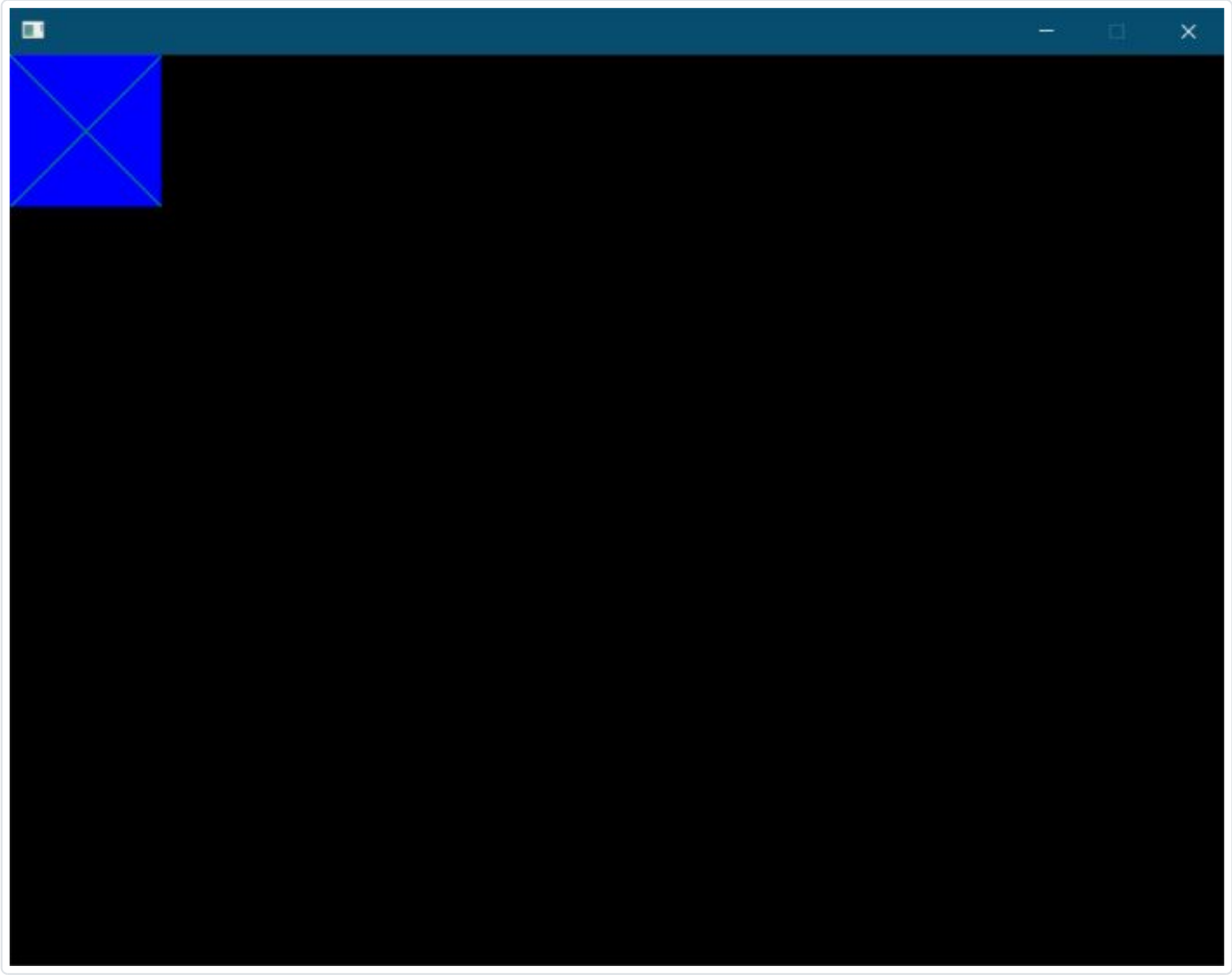


Pour le rectangle, de destination c'est sur quelle dimension que je vais la copier, si vous mettez la même hauteur et largeur que votre rectangle source, alors il va se copier naturellement sans redimensionner si vous mettez un rectangle plus grand

ou plus petit SDL fera en sorte de redimensionner pour que l'image que vous avez choisie de votre texture rentre dans le rectangle de destination.

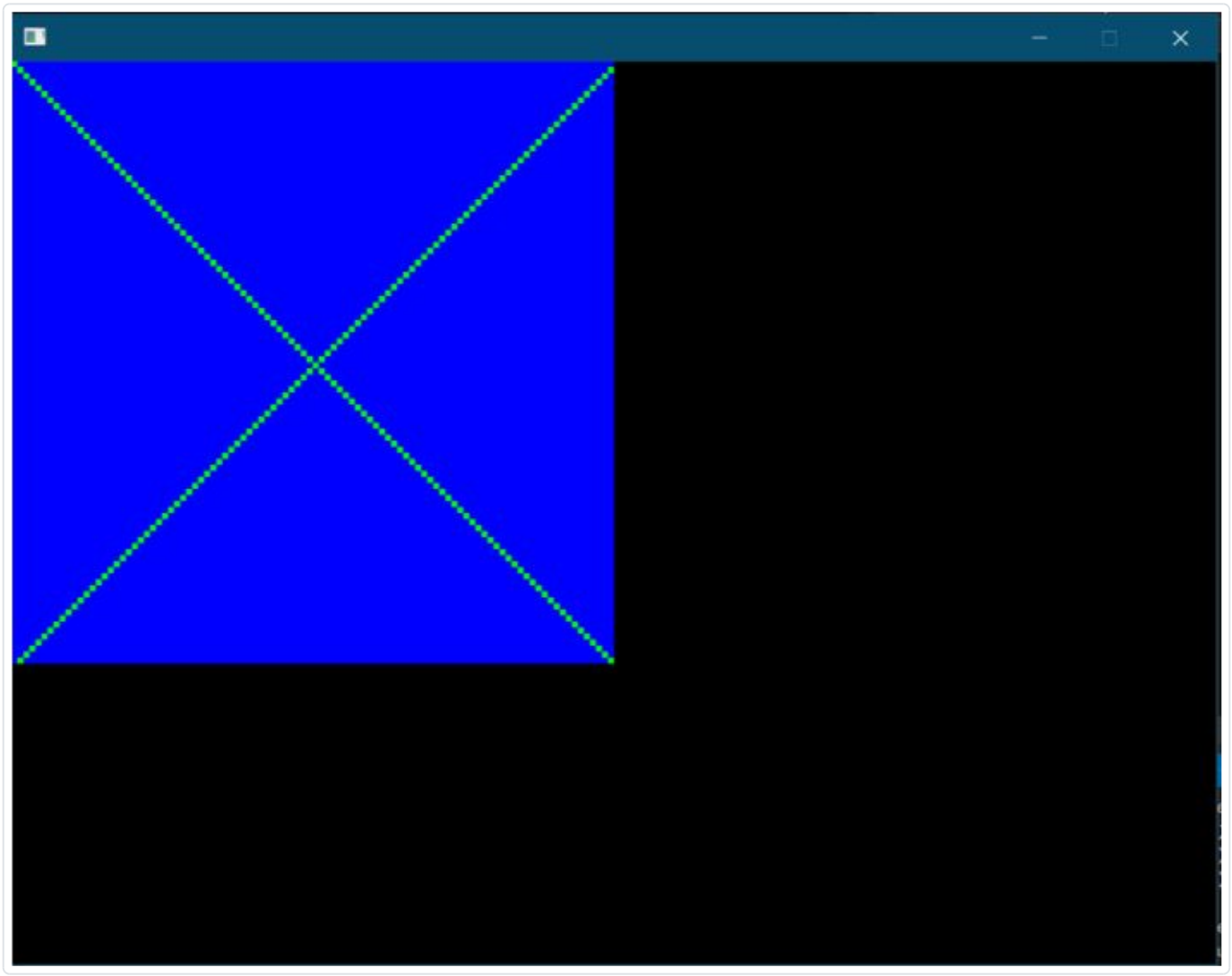
Par exemple, si je veux copier en haut à gauche de mon écran l'image sans la redimensionner je ferais ceci :

```
SDL_Rect src{ 0, 0, 100, 100 };

SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
SDL_RenderClear(pRenderer);

SDL_Rect src{ 0, 0, 100, 100 }; // Je prend toutes la texture sachant quel est de dim
SDL_RenderCopy(pRenderer, pTexture, &src, &src); // j'affiche la texture, a partir de

SDL_RenderPresent(pRenderer);
```



Amusez-vous à changer le paramètre dst :

```
SDL_Rect src{ 0, 0, 100, 100 };
SDL_Rect dst{ 0, 0, 400, 400 };
SDL_RenderCopy(pRenderer, pTexture, &src, &dst);
```



On remarque comme je vous les dis que Src a été redimensionné (on voit même quel est pixeliser) pour rentrer dans le rectangle dst.

Remarque si vos mettez :

```
SDL_RenderCopy(pRenderer, pTexture, nullptr, nullptr);
```

Ça va s'afficher sur tout l'écran. En effet mettre `nullptr` au rectangle Src signifie que prendre toutes la dimension de la texture. Et mettre `nullptr` à dst signifie se

prendre la dimension de l'écran. Il est réellement d'important de comprendre le rôle du dst et du Src.

Les surfaces

Bon, on en a pas encore fini, les surfaces sont les ancêtres de la `SDL_Texture`, en effet les `SDL_Surface` sont des structures, qui datent de la version 1 de la SDL et les `SDL_Texture` datent de la version 2 de la SDL, donc privilégier les `SDL_Texture` mais il y a des cas ou la SDL2 utilise encore les `SDL_Surface`, si vous avez cherché à comment **mettre une icône** sur votre programme vous êtes tombé sur ce prototype :

```
void SDL_SetWindowIcon(SDL_Window*  window,
                       SDL_Surface* icon)
```

- Le premier paramètre est la fenêtre `SDL_Window`
- Le deuxième paramètre est une `SDL_Surface`
- Elle ne retourne rien.

Il serait temps de pouvoir mettre une icône, ça vous tente ? Mettez une icône qui serait un rectangle bleu.

Vous : Bon, ben vas-y...

Moi: Ta vu, tu dis plus rien hein ;)

Bon comment, crée une `SDL_Surface`

voici le prototype ;

```
SDL_Surface* SDL_CreateRGBSurface(Uint32 flags,
                                  int    width,
                                  int    height,
                                  int    depth,
                                  Uint32 Rmask,
```

```
                                    Uint32 Gmask,
                                    Uint32 Bmask,
                                    Uint32 Amask)
```

- Le premier paramètre est un flag, en SDL version 1 il avait du sens, maintenant en SDL 2 il n'y en a plus mettez 0
- Le deuxième paramètre est la largeur de la surface.
- Le troisième paramètre est la hauteur de la surface.
- Le quatrième paramètre est le nombre de bits/pixels.
- Le cinquième est le masque de la couleur rouge.
- Le sixième est le masque de la couleur verte.
- Le septième est le masque de la couleur bleue.
- Le huitième est le masque de du composant alpha.
- Elle retourne la `SDL_Surface` ou `nullptr` en cas d'erreur

Bon trop de paramètres tue le paramètre, de toute manière ça sera la première et dernière fois que nous verrons cette fonction j'en parle parce qu'il faut bien savoir que ça existe.

- Pour raccourcir ce tutoriel, tous ces masks sont la juste question de savoir dans quel ordre sont les couleurs RGBA, BGRA etc.
- Les bits par pixels, est le nombre de bits utilisés pour coder une couleur, plus le nombre de bits par pixels est grand plus il y' a la possibilité d'afficher des couleurs.

Pour les masks nous allons mettre 0 c'est un moyen de mettre les valeurs par défaut si vous le souhaitez dans d'autre articles je pourrai parler plus en profondeur

de ce que c'est :).

La fonction pour la libérer en mémoire a pour prototype :

```
void SDL_FreeSurface(SDL_Surface* surface)
```

Voici maintenant le code résultat :

```
SDL_Surface* surface = SDL_CreateRGBSurface(0, 32, 32, 32, 0, 0, 0, 0);
SDL_FreeSurface(surface);
```

Maintenant nous allons le remplir de la couleur bleue pour faire une icône en rectangle de couleur bleue. Voici la fonction pour remplir une `SDL_Surface` :

```
int SDL_FillRect(SDL_Surface*    dst,
                 const SDL_Rect* rect,
                 Uint32          color)
```

- Le premier paramètre est la `SDL_Surface`.
- Le deuxième est la partie de la surface dans la quelle on va remplir le rectangle.
- Le troisième paramètre est la couleur.

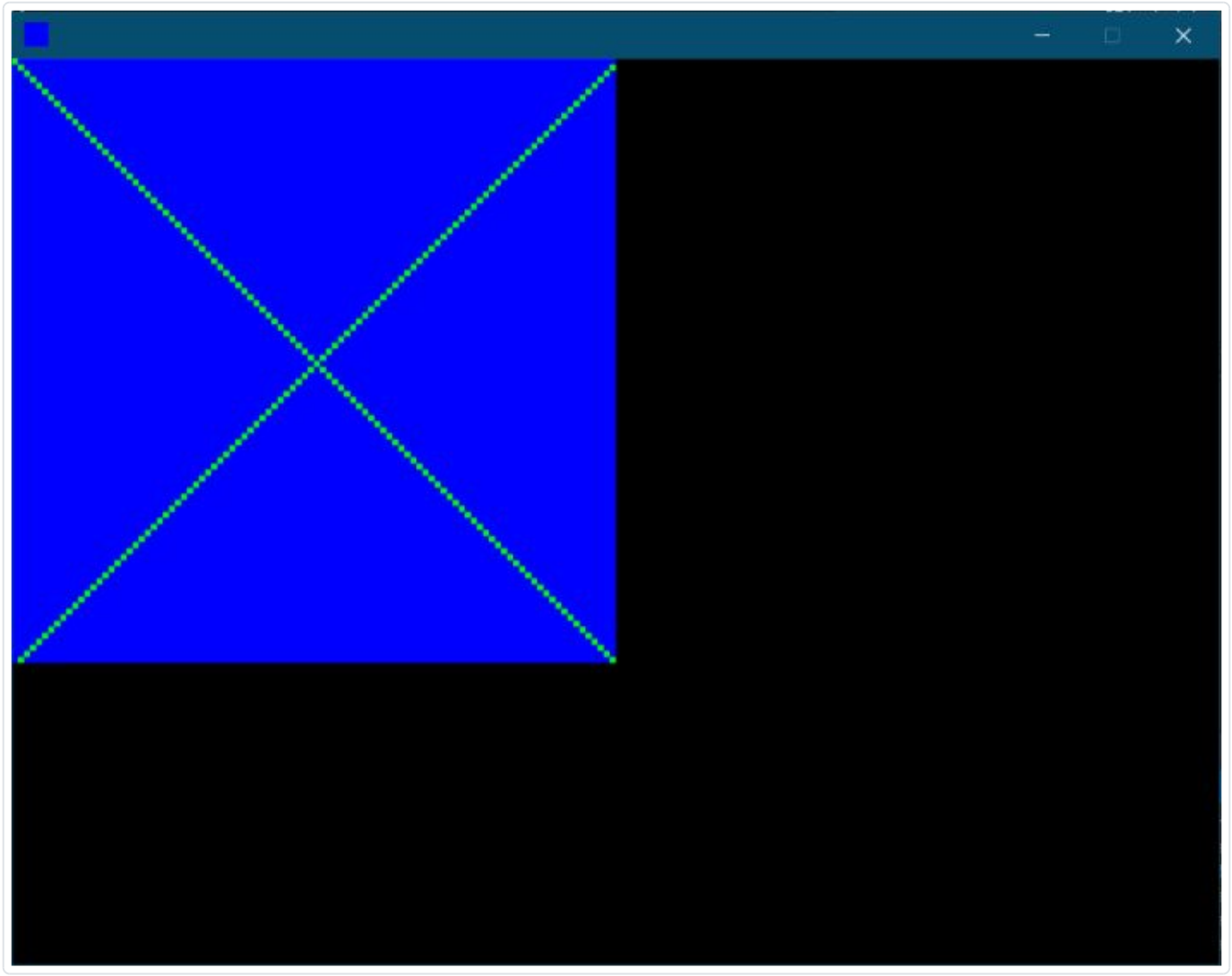
```
SDL_Surface* surface = SDL_CreateRGBSurface(0, 32, 32, 32, 0, 0, 0, 0);
SDL_Rect dst{ 0, 0, 32, 32 };


SDL_FillRect(surface, &dst, SDL_MapRGBA(surface->format, 0, 0, 255, 255));
SDL_SetWindowIcon(pWindow, surface);

SDL_FreeSurface(surface);
```

Bon je ne vous ai pas expliquer la fonction `SDL_MapRGBA()` voici sont prototype :

```
Uint32 SDL_MapRGBA(const SDL_PixelFormat* format,
                   Uint8                  r,
                   Uint8                  g,
                   Uint8                  b,
                   Uint8                  a)
```

- Le premier paramètre est le format (on en reparlera durant les `SDL_Texture`)
- Le deuxième paramètre est le taux de rouge.
- Le troisième paramètre est le taux de vert.
- Le quatrième paramètre est le taux de bleu.
- Le cinquième paramètre est le taux de transparence alpha.

Elle retourne une couleur sous forme de 32 bits par exemple du rouge donnera 0 x Ff0000. Ensuite dans le code je rajoute la surface pour faire une icône et que je libère avec la fonction `SDL_FreeSurface()` car j'en aurais plus besoin ne voila ce que sa donne :



Une belle icon bleue. si ce n'est pas beau ça ? Vu que nous n'avons pas encore vu le chargement d'image en SDL, nous allons utiliser une nouvelle fonction pour créer une surface, elle permet de créer une surface en fonction d'un tableau de pixels vu que j'en parle depuis les `SDL_Texture` il est temps de voir ce que c'est. Voici son prototype :

```
SDL_Surface* SDL_CreateRGBSurfaceFrom(void*  pixels,
                                      int    width,
                                      int    height,
                                      int    depth,
                                      int    pitch,
                                      Uint32 Rmask,
                                      Uint32 Gmask,
                                      Uint32 Bmask,
```

```
                                        Uint32 Amask)
```

- Le premier paramètre est le tableau de pixels.
- Le deuxième paramètre est la largeur de l'image.
- Le troisième paramètre est la hauteur de l'image.
- Le quatrième paramètre est le nombre de bits/pixels.
- Le cinquième paramètre est la longueur d'une ligne de pixels en octet.
- Le sixième est le masque de la couleur rouge.
- Le septième est le masque de la couleur verte.
- Le huitième est le masque de la couleur bleue.
- Le neuvième est le masque de du composant alpha.

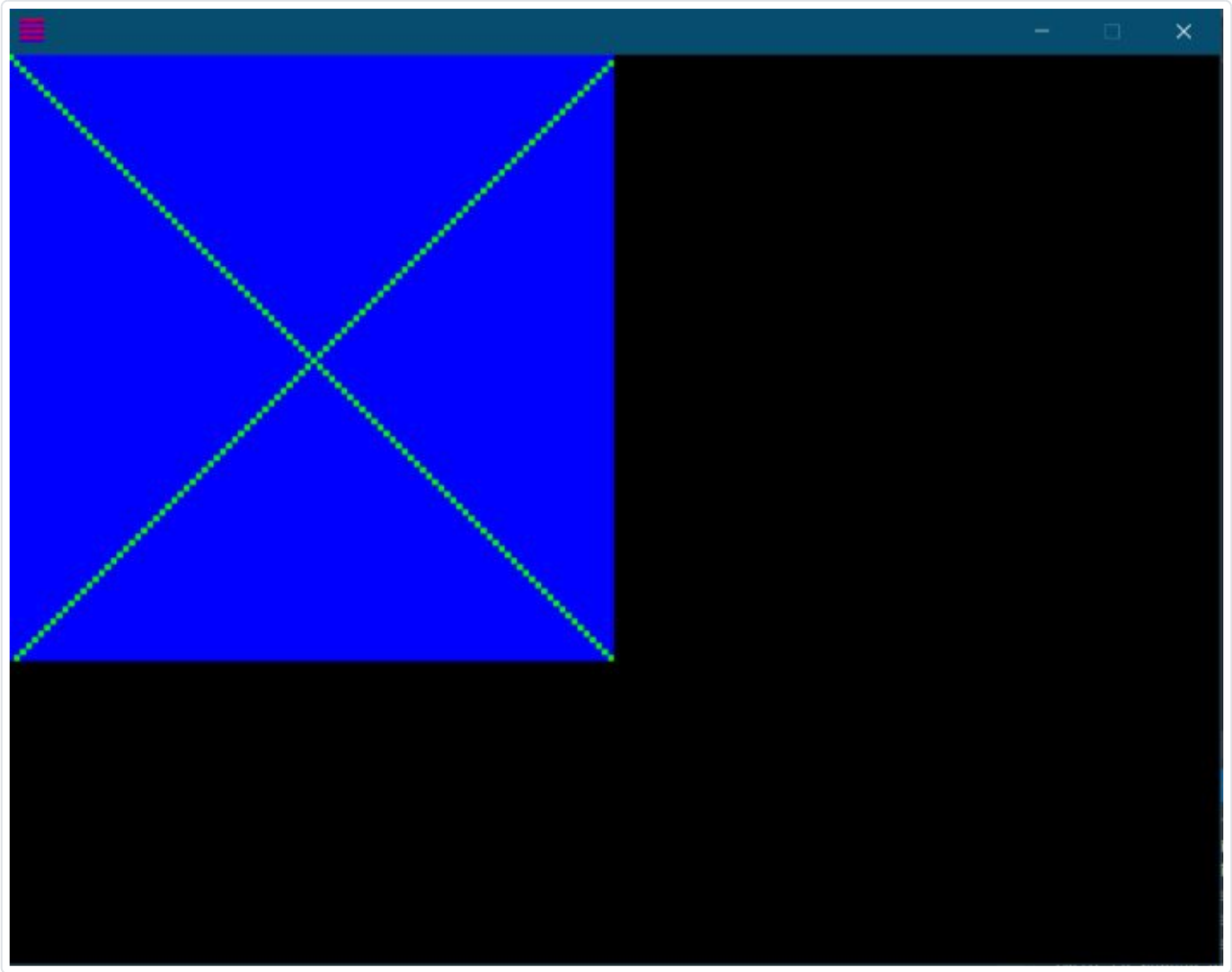
Le premier sera un tableau qui contiendra tous nous pixels, et pour le cinquième paramètre si j'ai 4 octets pour représenter une ligne, et que j'ai une image de 32 en largeur, pitch faudra (4 * 32) Coder un tableau qui aura pour dimension 32x32 et qui utilisera 4 octets pour la couleur d'un pixel, nous allons faire une icône qui s'alterne de rouge et de bleu. Voici mon code pour une icône personnalisée :

```
Uint32 pixels[32 * 32] = {
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
```

```
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
```

```
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f

        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000
        0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff0000, 0xff000

        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
        0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000ff, 0x0000f
};

SDL_Surface* surface = SDL_CreateRGBSurfaceFrom(pixels, 32, 32, 32, 32 * 4, 0, 0, 0,

SDL_SetWindowIcon(pWindow, surface);

SDL_FreeSurface(surface);
```

Bon, je ne vous conseille pas du tout, cette façon de faire pour mettre une icône, mais il était bon de savoir que ça existe :p.



Il est temps de simplifier tout ça avec un véritable affichage d'images et de vraies icônes, digne d'une SDL 2 .

Les images

Bon nous voila, arrivé presque à la fin de ce chapitre sur l'affiche graphique avec la SDL 2, je vous avez dit que la **SDL ne gère que le format d'image BMP**, pour le moment nous allons étudier d'abord juste ce que la SDL 2 propose nativement, mais ensuite, nous allons, géré beaucoup plus de format d'images via a le module `SDL_image`. Voici **comment charger une image** BMP en SDL :

```
SDL_Surface* SDL_LoadBMP(const char* file)
```

- Premier paramètre est le chemin du fichier.
- Elle retourne un pointeur sur une structure `SDL_Surface`, et `nullptr` s'il y a une erreur.

Cette fonction toute simple nous retourne un pointeur sur une `SDL_Surface`, qui sera la surface qui contiendra notre image, or nous n'avons pas vu comment afficher une image sur l'écran, et je ne veux pas vous montrer son affichage depuis les surfaces car c'est plus compliqué (pour les curieux yas la documentation qui existe).

Nous avons appris ici jusqu'ici à afficher une `SDL_Texture`, donc il faut à partir de la `SDL_Surface` créée une `SDL_Texture`. Pour créer une `SDL_Texture` après avoir créée une `SDL_Surface` il existe une fonction qui est :

```
SDL_Texture* SDL_CreateTextureFromSurface(SDL_Renderer* renderer,
                                          SDL_Surface*  surface)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est la `SDL_Surface` qui contient l'image BMP.

Voici en code ce que nous avons, n'oubliez pas juste après avoir appelé `SDL_CreateTextureFromSurface()` de libérer en mémoire la `SDL_Surface` qui contient l'image BMP, car nous en aurons plus besoin.

Utilisé l'image ci-dessous, si vous n'avez pas d'image BMP.

```
SDL_Surface* image = SDL_LoadBMP("imageBMP.bmp");
SDL_Texture* pTextureImage = SDL_CreateTextureFromSurface(pRenderer, image);
SDL_FreeSurface(image);
```

Voilà, ici nous avons chargé notre image BMP. maintenant il faut l'afficher :

```
SDL_SetRenderDrawColor()(pRenderer, 0, 0, 0, 255);
SDL_RenderClear(pRenderer);

SDL_RenderCopy(pRenderer, pTextureImage, nullptr, nullptr); // Affiche ma texture sur

SDL_RenderPresent(pRenderer);
```

Bon maintenant, l'image est un peu grande, j'aimerais déjà connaître, la taille de l'image, et ensuite la redimensionner en 400x300.

Pour récupérer, le `SDL_Rect`, qui représente l'image pour ainsi récupérer le taille de l'image, nous utiliserons cette fonction :

```
int SDL_QueryTexture(SDL_Texture* texture,
                     Uint32*      format,
                     int*         access,
                     int*         w,
                     int*         h)
```

- Le premier paramètre est la texture.

- Le deuxième paramètre est un pointeur sur le format de l'image.
- Le troisième paramètre est un pointeur sur le type d'accès de la texture.
- Le quatrième paramètre est la largeur de la texture.
- Le cinquième paramètre est la hauteur de la texture.
- Elle retourne 0 en cas de succès, ou une valeur négative en cas d'erreur.

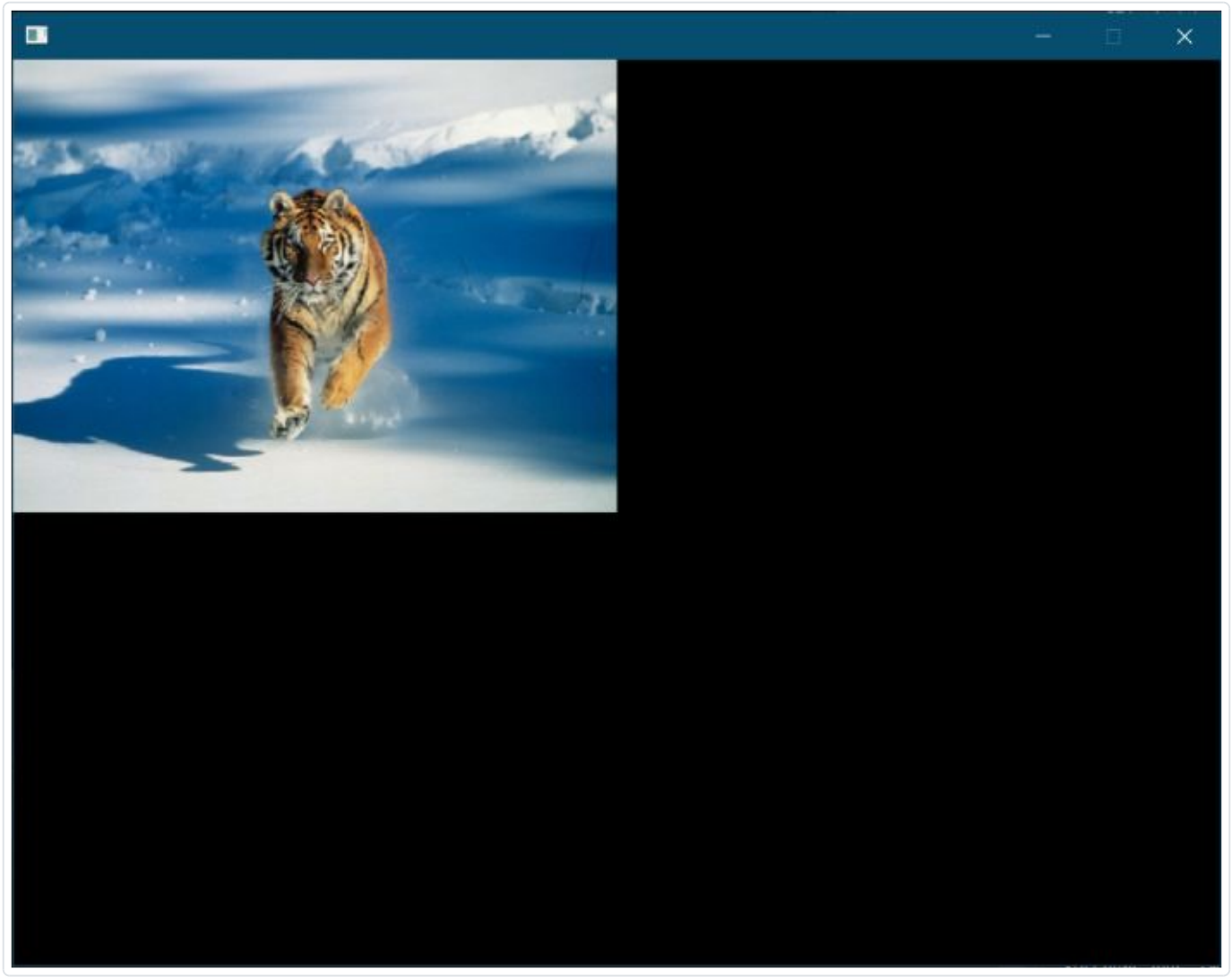
Voici comment l'utiliser pour récupérer la dimension de l'image, et ensuite l'afficher en la redimensionnant en 400x300.

```
SDL_Surface* image = SDL_LoadBMP("imageBMP.bmp");
SDL_Texture* pTextureImage = SDL_CreateTextureFromSurface(pRenderer, image);

SDL_FreeSurface(image);
SDL_Rect src{0, 0, 0, 0};
SDL_Rect dst{ 0, 0, 400, 300 };
SDL_QueryTexture(pTextureImage, nullptr, nullptr, &src.w, &src.h);
```

Vous constatez que le format et l'accès sont mis à `nullptr` faite de même car ils ne sont pas utiles ici. Ensuite je dessine :

```
SDL_SetRenderDrawColor()(pRenderer, 0, 0, 0, 255);
SDL_RenderClear(pRenderer);

SDL_RenderCopy(pRenderer, pTextureImage, &src, &dst);

SDL_RenderPresent(pRenderer);
```

Voici le résultat :



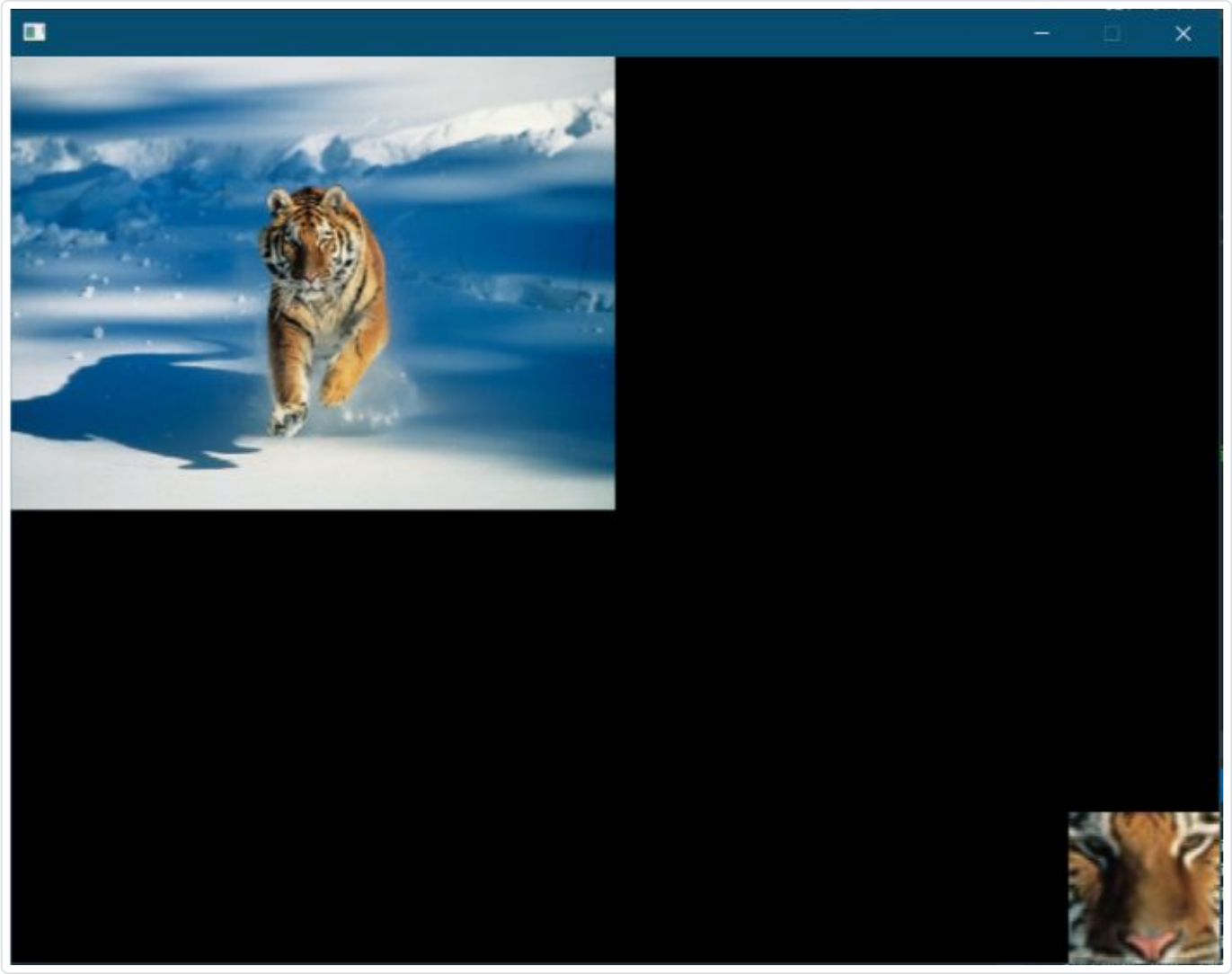
Bon, maintenant, par exemple je veux récupérer le lion et l'afficher en bas à droite. Pour ceux qui ont bien compris le Src et le dst de la fonction `SDL_RenderCopy()`, ce sera facile pour vous.

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


#include <cstdlib>

template<typename T>
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };
```

```
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(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }

    SDL_Event events;
    bool isOpen{ true };

    SDL_Surface* image = SDL_LoadBMP("imageBMP.bmp");
    SDL_Texture* pTextureImage = SDL_CreateTextureFromSurface(pRenderer, image);

    SDL_FreeSurface(image);
    SDL_Rect src1{0, 0, 0, 0};
    SDL_Rect src2{ 0, 0, 0, 0 };

    SDL_Rect dst1{ 0, 0, 400, 300 };
    SDL_Rect dst2{ WIDTHSCREEN<int> - 100, HEIGHTSCREEN<int> - 100, 100, 100};

    SDL_QueryTexture(pTextureImage, nullptr, nullptr, &src1.w, &src1.h);

    // Récuperer les bons coordonnées x et y pour afficher la tête du tigre
    src2.x = src1.w / 2 - 30;
    src2.y = src1.h / 2 - 50;
    src2.w = 50;
    src2.h = 50;


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

```
        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);

        SDL_RenderCopy(pRenderer, pTextureImage, &src1, &dst1); // Affiche la texture
        SDL_RenderCopy(pRenderer, pTextureImage, &src2, &dst2); // Affiche sur une pa

        SDL_RenderPresent(pRenderer);
    }

    SDL_DestroyTexture(pTextureImage);
    SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Voilà, ce que je vous dois avoir normalement :



Bon, nous avons vu comment, charger que des images au format BMP, il serait temps de voir comment, afficher des images avec d'autres formats.

Il est temps de parler du module `SDL_image`. vous verrez ça sera court mais avant il faut passer par l'installation de cette extension de la SDL, nous allons d'abord commencer par installation sous Linux.

Linux

Pour Linux, il faut utiliser le gestionnaire de paquets. On va commencer d'abord par mettre à jour nos paquets :

```
sudo apt-get update && sudo apt-get upgrade
```

Ensuite on télécharger `SDL_image` :

```
sudo apt-get install libsdl2-image-dev
```

En suite, vous pouvez retourner dans votre code et ajouter le header suivant :

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

Ensuite vous compilez avec cette ligne de commande :

```
g++ src/main.cpp -o bin/prog -lSDL2main -lSDL2_image -lSDL2
gcc src/main.c -o bin/prog -lSDL2main -lSDL2_image -lSDL2
```

Ne oublier pas c'est gcc pour du langage C et g++ pour du C++ ;)

Windows

Pour Windows, il faut aller sur le site officiel de `SDL_image`, qui est celui-ci : https://www.libsdl.org/projects/SDL_image/

Et vous descendez dans la partie **Développement Libraries:** :

prenez la **SDL2_image-Devel-2.0.4-Mingw.tar Gz (Mingw 32/64-bits)** et l'installation se fait en suivant les mêmes règles, pour le choix du dossier entre l'i686-w64-mingw32 et le x86_64-w64-mingw32.

Pour moi ça sera i686-w64-mingw32, ensuite en cliquant dessus on retrouve les 3 dossiers **bin** , **include** et **lib** , nous allons copier le contenu de ces dossiers dans les dossiers **bin** , **include** et **lib** de la SDL. Ensuite, vous pouvez retourner dans votre code et ajouter l'header suivant :

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

Ensuite vous compilez avec cette ligne de commande :

```
g++ src/main.cpp -o bin/prog -lmingw32 -lSDL2main -lSDL2_image -lSDL2
gcc src/main.c -o bin/prog -lmingw32 -lSDL2main -lSDL2_image -lSDL2
```

Ne oublier pas c'est gcc pour du langage C et g++ pour du C++ ;)

le module SDL_image

Si tous se compile sans aucune erreur alors il est temps maintenant pour vous d'oublier `SDL_LoadBMP()` et retenir `IMG_Load()`.

```
SDL_Surface * IMG_Load(const char *file);
```

Elle a le même prototype que la fonction `SDL_LoadBMP()`, mais sauf qu'elle sait gérer beaucoup plus de format d'images. Elle permet aussi de gérer la transparence d'une image. Or le BMP ne gère pas la transparence (même si il est possible de la gérer avec la SDL mais j'ai decidé de ne pas en parler au profit d'une utilisation de `SDL_image`).

Sachez qu'il existe une fonction `IMG_GetError()` qui permet de gérer les erreurs du module `SDL_image`. Elle s'utilise de la même manière que la fonction `SDL_GetError()` vu sur le cours précédent.

Le texte

Pour pouvoir, afficher du texte, il y existe deux façons soit :

- Recoder soi même une police de caractères et l'utiliser directement dans son programme, ce qui peut être long et fastidieux.
- Soit utiliser la bibliothèque `SDL_TTF` , qui se chargera de le faire pour nous et de pouvoir utiliser n'importe quelle police d'écriture.

Tout comme `SDL_image`, sont des sortes d'extensions de la SDL, il va falloir donc les télécharger et les installer. Pour cela, on procédera de la même manière d'installation que la `SDL_image`

Linux

Pour Linux, il faut encore utiliser encore le gestionnaire de paquets.

On va commencer d'abord par mettre à jour nos paquets comme on a l'habitude de le faire :

```
sudo apt-get update && sudo apt-get upgrade
```

Ensuite on télécharger `SDL_TTF` :

```
sudo apt-get install libsdl2-ttf-dev
```

Ensuite, vous pouvez retourner dans votre code et ajouter l'header suivant :

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

Ensuite vous compilez avec cette ligne de commande :

```
g++ src/main.cpp -o bin/prog -lSDL2main -lSDL2_image -lSDL2_tff -lSDL2
gcc src/main.c -o bin/prog -lSDL2main -lSDL2_image -lSDL2_ttf -lSDL2
```

Windows

Pour Windows, il faut visiter sur le site officiel de `SDL_TTF` , qui est celui-ci : Et descendre dans la partie Développement Libraries comme sur cette :

prenez : SDL2_ttf-devel-2.0.15-mingw.tar.gz (MinGW 32/64-bit) Eh l'installation se fait en suivant les mêmes règles, pour le choix du dossier entre l'i686-w64-mingw32 et le x86_64-w64-mingw32.

Pour moi ça sera i686-w64-mingw32, ensuite en cliquant dessus on retrouve les 3 dossiers **bin** , **include** et **lib** , nous allons copier le contenu de ces dossiers dans les dossiers **bin** , **include** et **lib** de la SDL. Ensuite, vous pouvez retourner dans votre code et ajouter l'header suivant :

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

Ensuite vous compilez avec cette ligne de commande :

```
g++ src/main.cpp -o bin/prog -lmingw32 -lSDL2main -lSDL2_image -lSDL2_ttf -lSDL2
gcc src/main.c -o bin/prog -lmingw32 -lSDL2main -lSDL2_image -lSDL2_ttf -lSDL2
```

Bon vous voyez que l'installation une fois qu'on sait faire, reste très répétitif, c'est pour ça que je me permet de moins détailler.

Parlons code, pour pouvoir utiliser cette bibliothèque il faut initialiser `SDL_TTF` . avec ce code :

```
if (TTF_Init() < 0)
{
    SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());
    return EXIT_FAILURE;
}
```

```
extern DECLSPEC int SDLCALL TTF_Init(void);
#define TTF_GetError    SDL_GetError
```

Une fois initialiser pensez à libérer les ressources en mémoire utilisez par `SDL_TTF` à la fin du programme en appelant la fonction `TTF_Quit()` voici son prototype :

```
void SDLCALL TTF_Quit(void);
```

```
// Libération des ressource en mémoire
SDL_DestroyRenderer(pRenderer);
SDL_DestroyWindow(pWindow);
TTF_Quit();
SDL_Quit();
```

Vous voyez que ça fonctionne un peu comme `SDL_Init()` sauf qu'il n'y a pas de paramètres, et qu'elle retourne 0 s'il y a succès ou une valeur négative en cas d'erreur. Et `TTF_GetError()` qui fonctionne comme `SDL_GetError()` vu sur le cours précedent.

Après avoir initialisé `SDL_TTF` avec `TTF_Init`, il faut **importer un font (une police d'écriture)**, ceci se fait avec la structure `TTF_Font`, pour créer une font il faudra utiliser la fonction `TTF_OpenFont()` voici son prototype :

```
TTF_Font*  TTF_OpenFont(const char *file, int ptsize);
```

- Le premier paramètre est le chemin où se situe la police d'écriture.
- Le deuxième paramètre est la taille de la police.
- Elle retourne une `TTF_Font` en cas de succès ou `nullptr` en cas d'erreur.

Si vous n'avez pas de police d'écriture à utiliser je vous conseille de télécharger une font sur ce site web : https://www.wfonts.com/font/arial . Cette page permet de télécharger la police d'écriture arial.

Voici le code qui permet de créer une font :

```
TTF_Font* font = TTF_OpenFont("ARIALI.TTF", 18);
```

Toute suite après il est bien d'appeler la fonction qui permet de libérer en mémoire la structure `TTF_Font`. La fonction à appeler est `TTF_CloseFont()` et voici son prototype :

```
void SDLCALL TTF_CloseFont(TTF_Font *font);
```

- Le premier paramètre est l'adresse de la structure `TTF_Font`
- Elle ne retourne rien.

On se retrouve avec ce code suivant :

```
TTF_Font* font = TTF_OpenFont("ARIALI.TTF", 18);
TTF_CloseFont(font);
```

Maintenant, il faut créé une surface qui contiendra notre texte par un exemple un jolie "Hello, World". Pour cela existe trois fonctions :

```
TTF_RenderText_Solid
TTF_RenderText_Shaded
TTF_RenderText_Blended
```

Voici leurs prototypes :

```
extern DECLSPEC SDL_Surface * SDLCALL TTF_RenderText_Solid(TTF_Font *font,
                const char *text, SDL_Color fg);
extern DECLSPEC SDL_Surface * SDLCALL TTF_RenderText_Shaded(TTF_Font *font,
                const char *text, SDL_Color fg, SDL_Color bg);
extern DECLSPEC SDL_Surface * SDLCALL TTF_RenderText_Blended(TTF_Font *font,
```

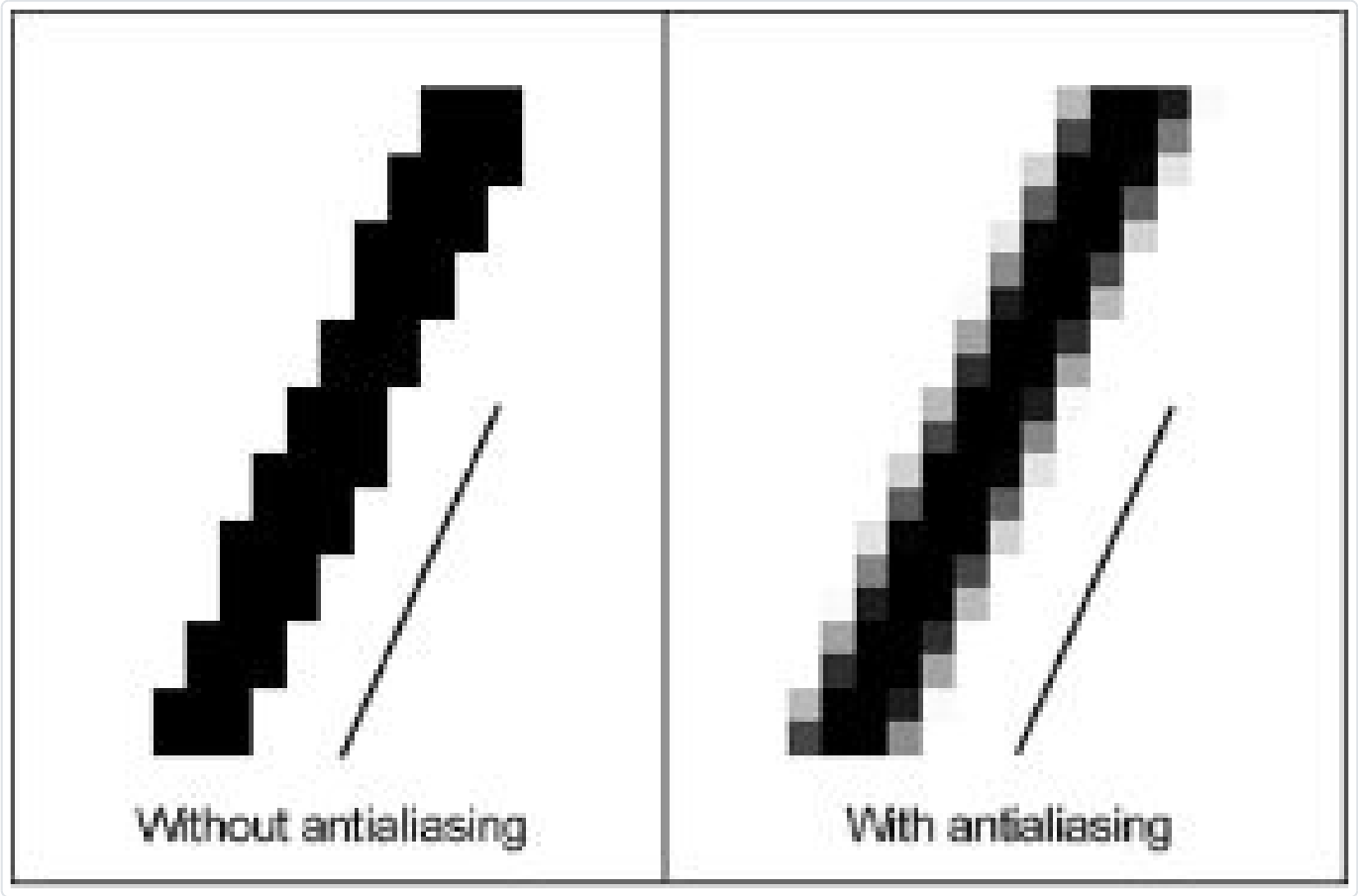
```
                const char *text, SDL_Color fg);
```

- Le premier paramètre est la police d'écriture un `TTF_Font*`.
- Le deuxième paramètre est le texte à écrire.
- Le troisième paramètre est la couleur du texte.
- Elle retourne un pointeur sur une `SDL_Surface` en cas de réussite sinon `nullptr`.

On constate que c'est 3 fonctions a le même prototype, or il y a bien une différence entre c'est 3 fonctions. Voici leurs différences :

- `TTF_RenderText_Solid` : le texte sera transparent, mais il n'y aura pas de lissage c'est-à-dire que le texte sera moins joli, il y aura un effet escalier. À utiliser pour le texte qui change souvent, comme par exemple lorsque vous affichez la position du joueur en mode debug de votre application, ou les FPS.
- `TTF_RenderText_Shaded` : le texte ne sera pas transparent, mais il y aura un effet de lissage (**antialiasing**).
- `TTF_RenderText_Blended` : le texte sera transparent, et il y aura un effet de lissage (antialiasing).

Voici, une idée de ce que c'est l'antialiasing, c'est un lisage, sur les contours et ça évite l'effet escalier.



Généralement, utiliser `TTF_RenderText_Blended` sinon `TTF_RenderText_Solid`, dans notre cas nous allons utiliser `TTF_RenderText_Blended`, voici le code qui permet de créer la surface qui contiendra le texte :

```
SDL_Surface* text = TTF_RenderText_Blended(font, "Hello, World", SDL_Color{ 0, 255, 0
```

N'oubliez pas de libérer la surface en effet on aura plus besoin, libéré là au même niveau ou vous libérez en mémoire la structure `TTF_Font` voici le code :

```
TTF_Font* font = TTF_OpenFont("ARIALI.TTF", 18);
SDL_Surface* text = TTF_RenderText_Blended(font, "Hello, World", SDL_Color{ 0, 255, 0

SDL_FreeSurface(text);
TTF_CloseFont(font);
```

Maintenant, il nous manque plus qu'à créer une texture et à l'afficher. Nnous savons déjà faire ceci, nous l'avons appris dans le chapitre sur les images ;) avec la fonction `SDL_CreateTextureFromSurface()`

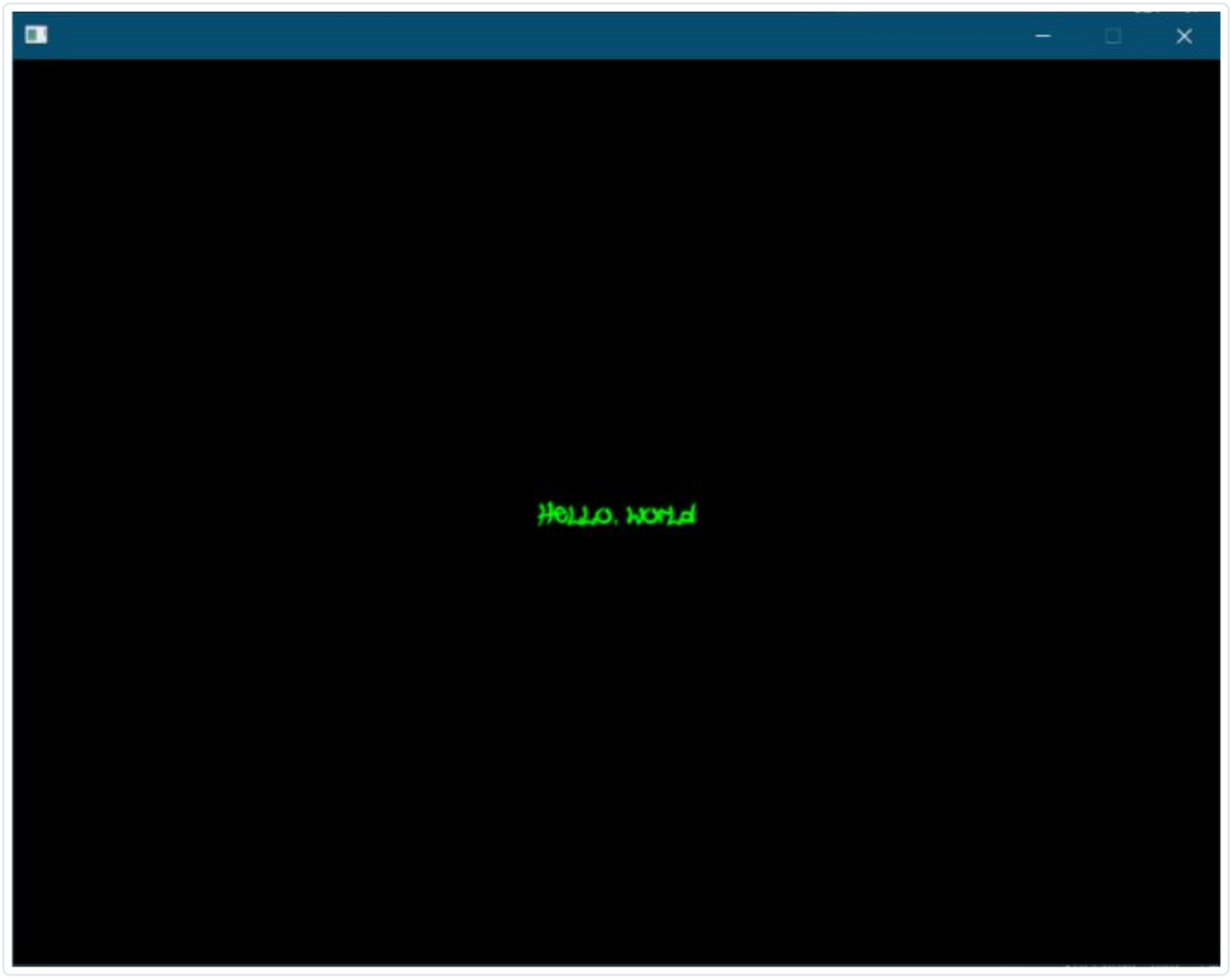
.

Voici donc le code complet :

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

#include <cstdlib>

template<typename T>
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };


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;
    }

    if (TTF_Init() < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());
        return EXIT_FAILURE;
    }

     SDL_Window* pWindow{ nullptr };
    SDL_Renderer* pRenderer{ nullptr };

     if (SDL_CreateWindowAndRenderer(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }

    SDL_Event events;
    bool isOpen{ true };


    TTF_Font* font = TTF_OpenFont("ARIALI.TTF", 18); // Crée un police avec la police

    if (font == nullptr)
    {
```

```
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());

    }

    SDL_Surface* text = TTF_RenderText_Blended(font, "Hello, World", SDL_Color{ 0, 25

    if (text == nullptr)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());
    }

    SDL_Texture* texture = SDL_CreateTextureFromSurface(pRenderer, text); // Crée la

    if (texture == nullptr)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());
    }

    SDL_Rect position;

    SDL_QueryTexture(texture, nullptr, nullptr, &position.w, &position.h); // Récuper

    // Centre la texture sur l'écran
    position.x = WIDTHSCREEN<int> / 2 - position.w / 2;
    position.y = HEIGHTSCREEN<int> / 2 - position.h / 2;


    // Libération des resource de la police et de la surface qui contient le texte
    SDL_FreeSurface(text);
    TTF_CloseFont(font);


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


        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);

        SDL_SetRenderDrawColor(pRenderer, 0, 255, 0, 255);
        SDL_RenderCopy(pRenderer, texture, nullptr, &position);

        SDL_RenderPresent(pRenderer);
    }
```

```
    SDL_DestroyTexture(texture);
     SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    TTF_Quit();
    SDL_Quit();

    return EXIT_SUCCESS;
}
```



Bon, nous avons vraiment vu pas mal de chose, et si vous maitrisez tous ce qu'on vient de voir, je vous assure que vous aurez un bon niveau en SDL.

Les rotations

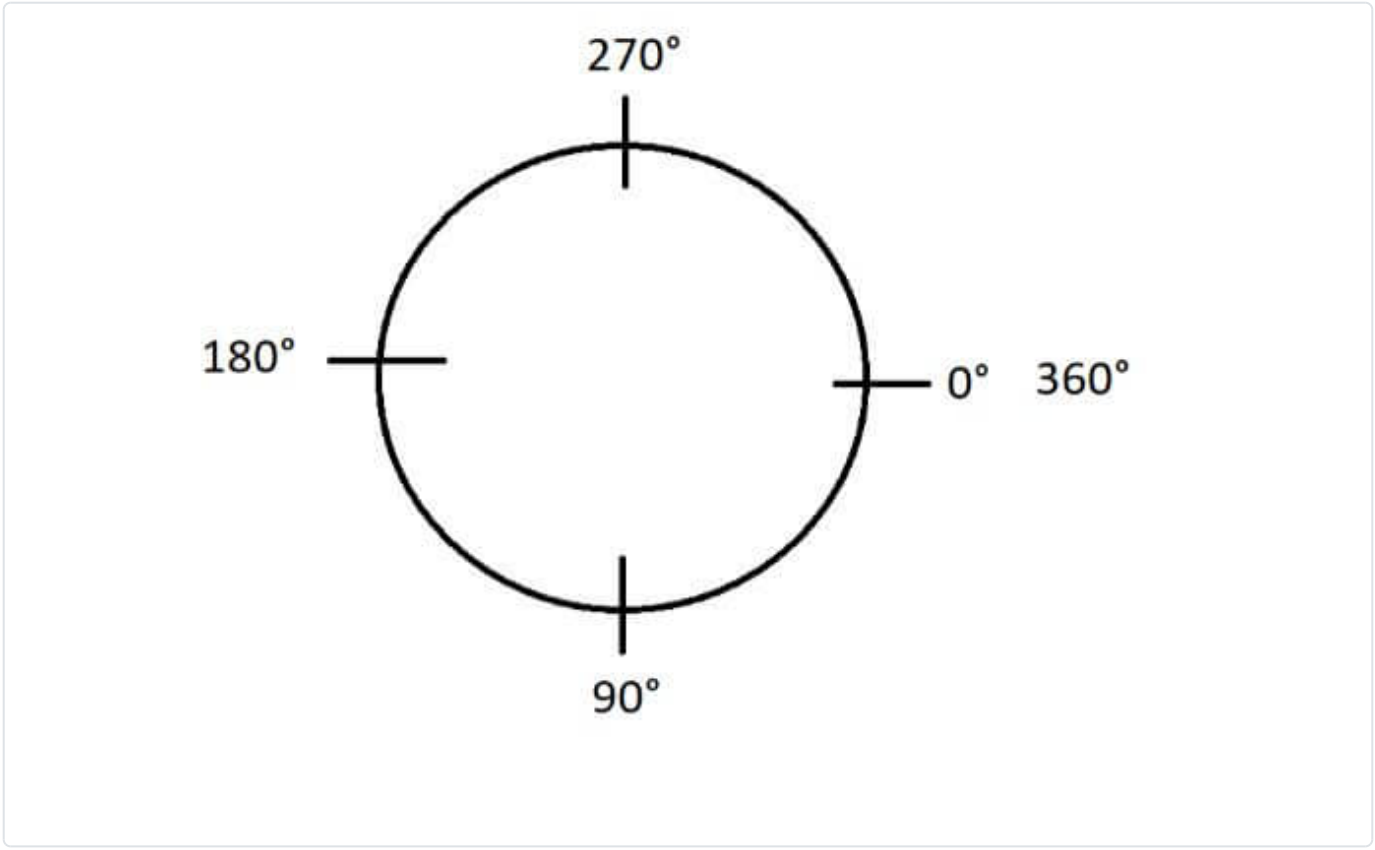
Ce chapitre sera vraiment très court. Eh oui le dernier chapitre sera sans doute le plus court que vous aurez vu jusqu'à l'instant car nous allons étudier qu'une seule fonction.

Nous connaissant maintenant bien la fonction `SDL_RenderCopy()` mais nous connaissons moins `SDL_RenderCopyEx()`. Voici son prototype :

```
int SDL_RenderCopyEx(SDL_Renderer*          renderer,
                     SDL_Texture*           texture,
                     const SDL_Rect*        srcrect,
                     const SDL_Rect*        dstrect,
                     const double           angle,
                     const SDL_Point*       center,
                     const SDL_RendererFlip flip)
```

- Le premier paramètre est le rendu de fenêtre.
- Le deuxième paramètre est la texture à afficher.
- Le troisième paramètre est le rectangle source.
- Le quatrième paramètre est le rectangle de destination.
- Le cinquième paramètre est l'angle de rotation de la texture.
- Le sixième paramètre est l'origine ou se fera la rotation.
- Le septième paramètre c'est le retournement.

Le premier paramètre jusqu'au quatrième on connaît déjà. Ce qui a de nouveau est le cinquième paramètre qui est un angle en degrés. Et SDL utilise le sens horaire et non trigonométrique, le sens horaire est parfois appeler le sens antitrigonométrique



Le sixième paramètre est le point de rotation, prenez une feuille A4 et mettez votre doigt sur la feuille et faite la pivoter comme ceci, si vous mettez votre doigt en haut à gauche, la feuille pivotera autour de ce point, si vous mettez au centre elle pivotera au centre de la feuille A4 tous dépend ou vous placer votre doigt en SDL c'est pareil la rotation dépend ou vous placer votre point.

Si vous mettez la largeur de la texture divisée par 2 et la hauteur de la texture diviser par deux et que vous affichez ça au centre de l'écran vous aurez une surprise.

Le septième est le paramètre qui indique la symétrie, et voici les valeurs possibles pour le retournement :

SDL_FLIP_NONE	do not flip
SDL_FLIP_HORIZONTAL	flip horizontally
SDL_FLIP_VERTICAL	flip vertically

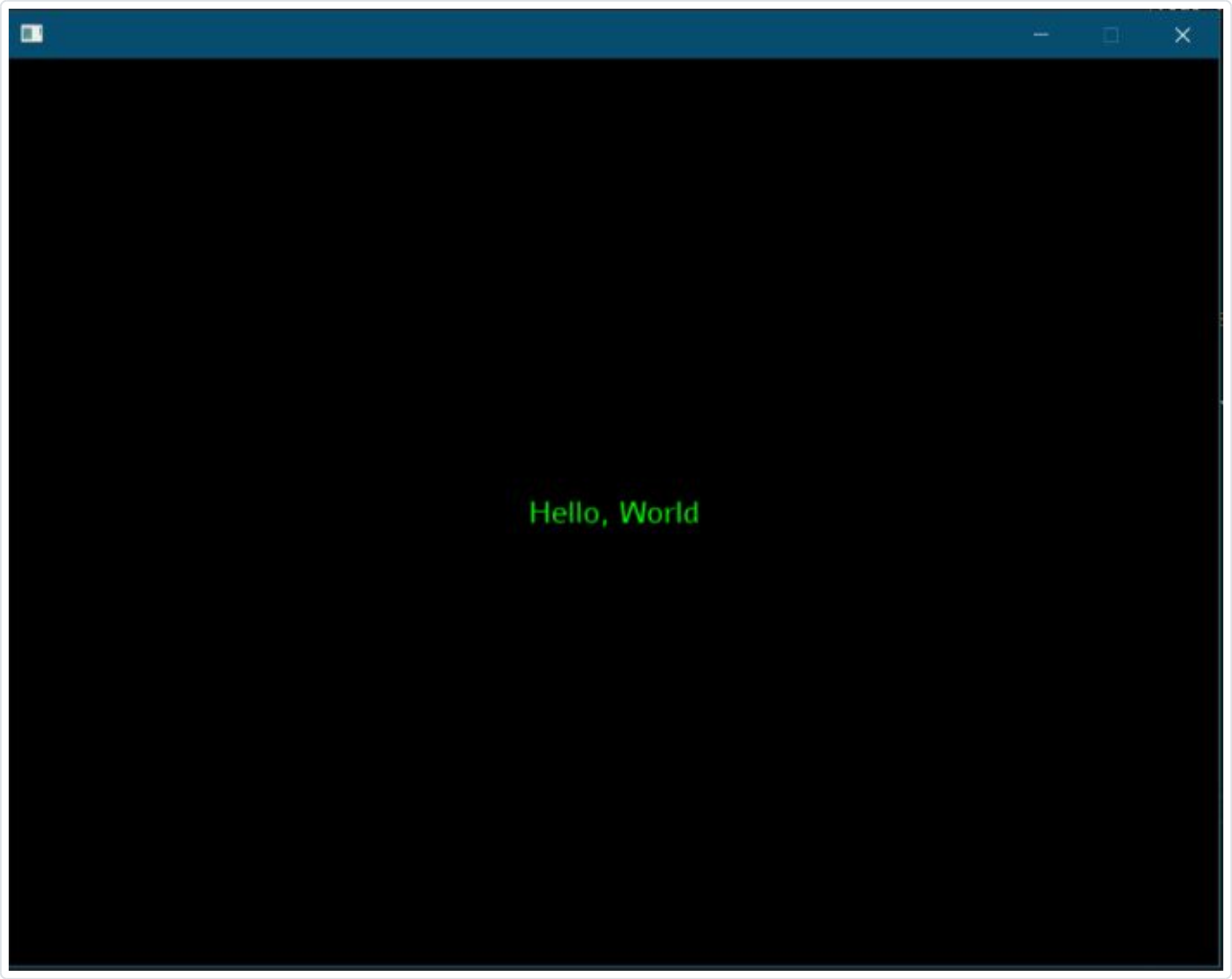
- Le premier indique qu'on ne retourne pas la texture.
- Le deuxième indique qu'on retourne la texture de façon horizontale.
- Le troisième indique qu'on retourne la texture de façon verticale.

Passons au code, pour l'angle de rotation mettons 0, on va afficher l'Hello, World normalement. Pour le point centre mettons la largeur divisée par 2 et hauteur divisée par 2, et pour le retournement mettons `SDL_FLIP_NONE` voici le code :

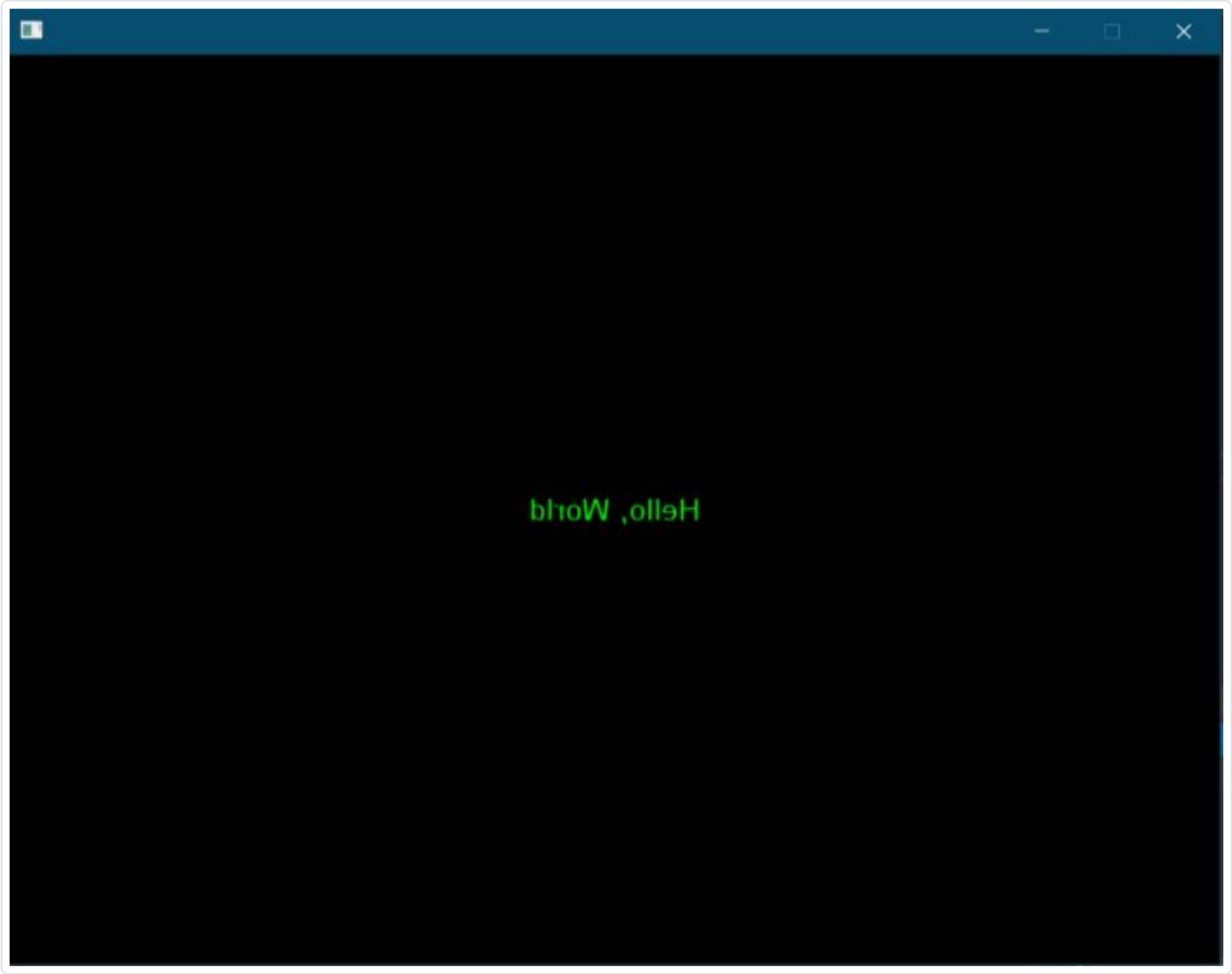
```
SDL_Point center = { position.w / 2, position.h / 2 }; // Place le "doigt" au centre
double angle = 0; // l'angle de rotation est de 0
SDL_RendererFlip flip = static_cast<SDL_RendererFlip>(SDL_FLIP_NONE); // Pas de retou
```

```
SDL_SetRenderDrawColor()(pRenderer, 0, 0, 0, 255);
SDL_RenderClear(pRenderer);

SDL_RenderCopyEx(pRenderer, texture, nullptr, &position, 0, &center, flip);

SDL_RenderPresent(pRenderer);
```

Vous aurez ceci :

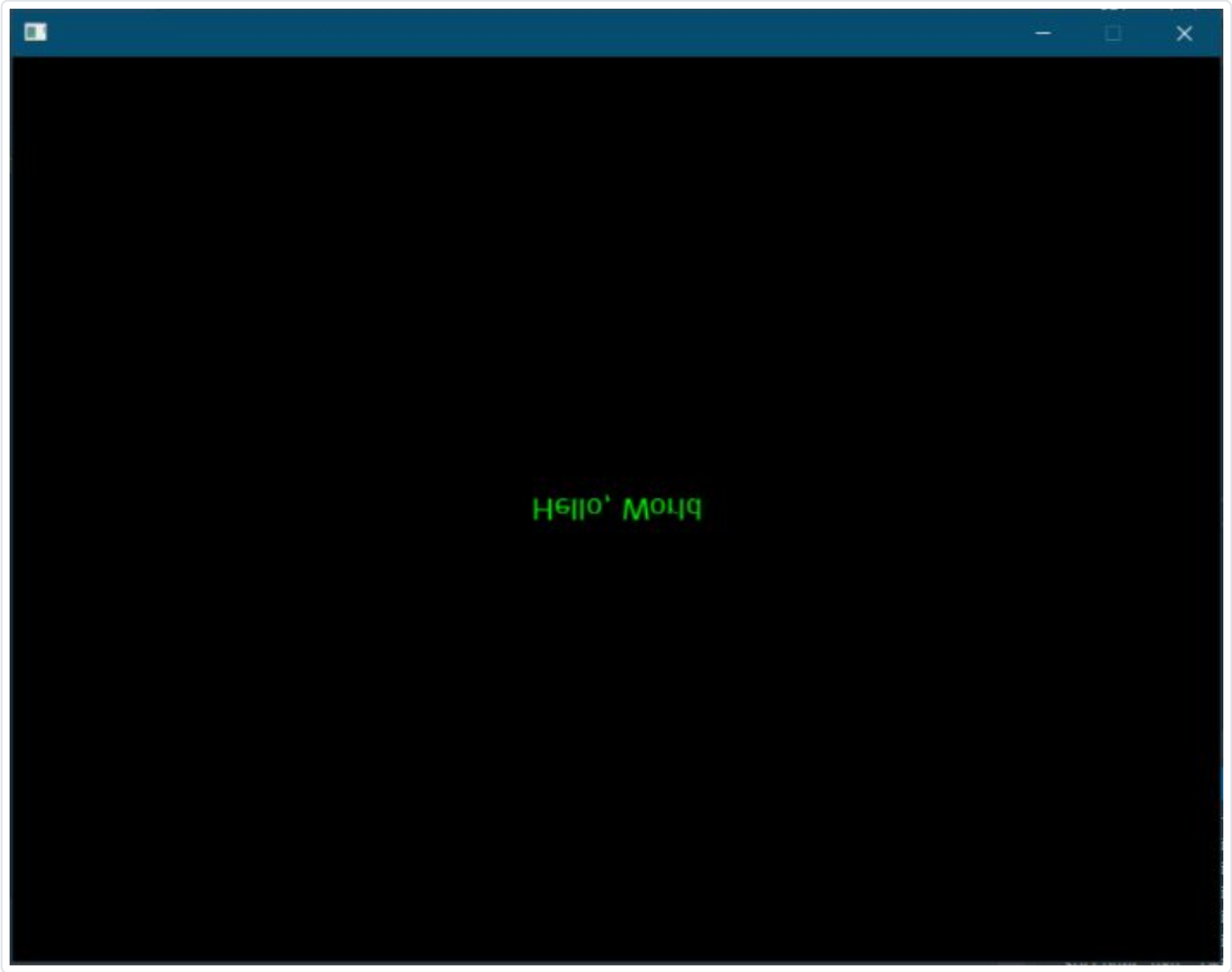


Vous remarquez que j'ai changé la police pour qu'on voit mieux. Vous remarquez aussi qu'ici il n'y a pas de retournement maintenant, si vous mettez, un retournement horizontal avec `SDL_FLIP_HORIZONTAL`.



On constate ici que le H se retrouve tout à droite.

Maintenant, un retournement vertical avec `SDL_FLIP_VERTICAL`.



Il est possible de combiner les deux de cette manière `SDL_FLIP_VERTICAL | SDL_FLIP_HORIZONTAL` mais je crois que vous aurez compris ce que ça fera.

Maintenant je vous donne comme exercice de faire une animation de rotation de l'hello world.

Corection.!!

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

#include <cstdlib>

template<typename T>
```

```
constexpr T WIDTHSCREEN{ 800 };

template<typename T>
constexpr T HEIGHTSCREEN{ 600 };

template<typename T>
constexpr T TOTAL_POINTS{ 5000 };


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;
    }

    if (TTF_Init() < 0)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());
        return EXIT_FAILURE;
    }

     SDL_Window* pWindow{ nullptr };
    SDL_Renderer* pRenderer{ nullptr };

     if (SDL_CreateWindowAndRenderer(WIDTHSCREEN<unsigned int>, HEIGHTSCREEN<unsigned
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", SDL_GetError());
        SDL_Quit();
        return EXIT_FAILURE;
    }

    SDL_Event events;
    bool isOpen{ true };


    TTF_Font* font = TTF_OpenFont("Alef-Regular.ttf", 20); // Crée un police avec la

    if (font == nullptr)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());

    }

    SDL_Surface* text = TTF_RenderText_Blended(font, "Hello, World", SDL_Color{ 0, 25

    if (text == nullptr)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());
    }

    SDL_Texture* texture = SDL_CreateTextureFromSurface(pRenderer, text); // Crée la
```

```
    if (texture == nullptr)
    {
        SDL_LogError(SDL_LOG_CATEGORY_APPLICATION, "[DEBUG] > %s", TTF_GetError());
    }

    SDL_Rect position;

    SDL_QueryTexture(texture, nullptr, nullptr, &position.w, &position.h); // Récuper

    // Centre la texture sur l'écran
    position.x = WIDTHSCREEN<int> / 2 - position.w / 2;
    position.y = HEIGHTSCREEN<int> / 2 - position.h / 2;


    // Libération des resource de la police et de la surface qui contient le texte
    SDL_FreeSurface(text);
    TTF_CloseFont(font);

    SDL_Point center = { position.w / 2, position.h / 2 };
    double angle = 0;
    SDL_RendererFlip flip = static_cast<SDL_RendererFlip>(SDL_FLIP_NONE);

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

        // Faire en sorte que l'angle soit compris entre 0° et 360°
        if (angle < 0)
            angle = 360;
        if (angle > 360)
            angle = 0;

        // Incrémentation de l'angle
        angle++;


        SDL_SetRenderDrawColor(pRenderer, 0, 0, 0, 255);
        SDL_RenderClear(pRenderer);

        SDL_RenderCopyEx(pRenderer, texture, nullptr, &position, angle, &center, flip

        SDL_RenderPresent(pRenderer);
    }


    SDL_DestroyTexture(texture);
```

```
     SDL_DestroyRenderer(pRenderer);
    SDL_DestroyWindow(pWindow);
    TTF_Quit();
    SDL_Quit();

    return EXIT_SUCCESS;
}
```

Bon, je pense, sincèrement que ce n'était pas trop dur.

C'est ainsi que ce conclu ce gros chapitre, sur l'affichage graphique sous SDL2 et nous avons encore un long chemin à passer ensemble