



# Utiliser la SDL en langage C

---

2 février 2020



# Table des matières

<b>1. La SDL</b>	<b>4</b>
1.1. Présentation de la SDL	4
1.1.1. Qu'est-ce que la SDL	4
1.1.2. Que peut faire la SDL	4
1.1.3. Pourquoi choisir la SDL	5
1.2. Configurer un projet avec la SDL	6
1.2.1. Windows	6
1.2.2. Linux	6
1.2.3. OSX	7
1.3. Prérequis et état d'esprit du tutoriel	7
1.3.1. Prérequis	7
1.3.2. Le but du tutoriel	8
1.3.3. La documentation	8
<b>2. Les premières fenêtres</b>	<b>10</b>
2.1. Initialiser la SDL	10
2.1.1. Initialiser la SDL	10
2.1.2. Quitter la SDL	11
2.1.3. Gestion des erreurs	12
2.2. Créer des fenêtres	13
2.2.1. Créer une fenêtre	13
2.2.2. Détruire la fenêtre	14
2.2.3. Un bon code	15
2.3. Gérer la fenêtre	15
2.3.1. Les paramètres de la fenêtre	15
2.3.2. Agir sur la fenêtre	18
<b>3. Dessiner dans la fenêtre</b>	<b>21</b>
3.1. Gestion du rendu	21
3.1.1. Créer un renderer	21
3.1.2. Détruire le renderer	22
3.1.3. Créer le renderer et la fenêtre en même temps	23
3.2. Des rectangles et des couleurs	23
3.2.1. La base de tout, le point	23
3.2.2. On complète la base, le rectangle	24
3.2.3. Des fonctions utiles	25
3.2.4. Les couleurs	27
3.3. Des dessins	28
3.3.1. Le principe	28
3.3.2. Dessiner des points et des lignes	30

3.3.3. Dessiner des rectangles . . . . .	32
Contenu masqué . . . . .	33
<b>4. Les textures et les images</b>	<b>34</b>
4.1. Les textures . . . . .	34
4.1.1. Généralités sur les textures . . . . .	34
4.1.2. Dessiner sur une texture . . . . .	35
4.1.3. Afficher une texture . . . . .	36
4.2. Les surfaces . . . . .	38
4.2.1. Un vestige de la SDL 1 . . . . .	38
4.2.2. Opérations sur les surfaces . . . . .	39
4.2.3. Passer de la texture à la surface . . . . .	40
4.3. Les images . . . . .	41
4.3.1. Charger une image . . . . .	41
4.3.2. Dessiner sur l'image . . . . .	42
4.3.3. Se faire des fonctions . . . . .	42
<b>5. Modification pixels par pixels</b>	<b>44</b>
5.1. Les textures . . . . .	44
5.1.1. Le format des pixels . . . . .	44
5.1.2. Modifier les pixels . . . . .	46
5.1.3. Mettre à jour une texture . . . . .	47
5.2. Les surfaces . . . . .	48
5.2.1. Le format des pixels, encore et toujours . . . . .	48
5.2.2. Modifier les pixels . . . . .	49
5.2.3. Une nouvelle surface ? . . . . .	49
5.3. Lier surfaces et textures . . . . .	50
5.3.1. De la texture à la surface . . . . .	50
5.3.2. Du renderer à la surface ? . . . . .	51
5.3.3. Quelles opérations privilégier ? . . . . .	52
<b>6. La transparence</b>	<b>54</b>
6.1. La transparence alpha . . . . .	54
6.1.1. Un problème embêtant . . . . .	54
6.1.2. Les modes de fusion . . . . .	54
6.1.3. Les modes de fusion de la SDL . . . . .	55
6.2. Gérer la transparence alpha . . . . .	56
6.2.1. Avoir un renderer transparent . . . . .	56
6.2.2. Avoir une texture transparente . . . . .	57
6.2.3. Des collisions entre les textures et le renderer . . . . .	58
6.3. Avec les surfaces . . . . .	59
6.3.1. La transparence alpha . . . . .	59
6.3.2. Rendre une couleur transparente . . . . .	60
<b>7. TP - Effets sur des images</b>	<b>61</b>
7.1. Noirs et négatifs . . . . .	61
7.1.1. Principe . . . . .	61
7.1.2. Niveaux de gris . . . . .	62
7.1.3. Négatifs . . . . .	62

7.2.	De la luminosité . . . . .	63
7.2.1.	Éclaircir une image - version naïve . . . . .	63
7.2.2.	Éclaircir une image - version améliorée . . . . .	64
7.2.3.	Contraster une image . . . . .	65
7.3.	Floutage . . . . .	67
7.3.1.	Un peu de théorie . . . . .	67
7.3.2.	Le flou gaussien . . . . .	67
7.3.3.	Du détournement ? . . . . .	68
<b>8.</b>	<b>Les événements 1</b>	<b>69</b>
8.1.	Gérer les événements . . . . .	69
8.1.1.	Une file d'événements . . . . .	69
8.1.2.	La structure <code>SDL_Event</code> . . . . .	70
8.1.3.	Récupérer les événements . . . . .	71
8.2.	Analyser les événements . . . . .	73
8.2.1.	La fenêtre . . . . .	73
8.2.2.	Le clavier . . . . .	74
8.2.3.	La souris . . . . .	75
8.3.	Le statut des périphériques . . . . .	78
8.3.1.	La souris . . . . .	78
8.3.2.	Le clavier . . . . .	79
8.3.3.	Une structure pour la gestion des événements . . . . .	79
	Contenu masqué . . . . .	81

Vous connaissez le langage C et vous souhaitez l'utiliser pour faire des petits programmes graphiques? Vous êtes curieux et vous voulez découvrir la SDL? Ou encore, vous venez de finir d'apprendre les bases du langage C et voulez les mettre en pratique en utilisant une bibliothèque graphique? Vous êtes au bon endroit.

Dans ce tutoriel, nous verrons comment utiliser la SDL afin de faire des programmes graphiques.



### Prérequis

Savoir programmer en langage C (un tutoriel est disponible [ici](#) ).

### Prérequis optionnel

Avoir des bases en représentation des nombres en machines.

Connaître les opérateurs bits à bits (un tutoriel est disponible [ici](#) ).

Savoir utiliser les drapeaux.

### Objectifs

Apprendre à utiliser une nouvelle bibliothèque.

Introduire à la documentation de la SDL.

Apprendre à faire des petits jeux.

# 1. La SDL

Dans ce premier chapitre, nous allons voir ce qu'est la SDL et apprendre à faire un projet l'utilisant. C'est donc un chapitre d'introduction qui nous informera sur la SDL et sur le tutoriel. Nous y apprendrons les prérequis nécessaires pour suivre ce tutoriel et comment celui-ci se déroulera.

## 1.1. Présentation de la SDL

### 1.1.1. Qu'est-ce que la SDL

La SDL c'est-à-dire la Simple DirectMedia Layer est une bibliothèque multimédia. Elle permet un accès de bas-niveau à l'audio, au clavier, à la souris, au joystick, aux graphiques... Cela veut dire qu'elle permet d'afficher des fenêtres, d'afficher des images, de jouer des sons, de gérer le clavier...

Il s'agit d'une bibliothèque libre, multiplateforme et assez connue. Elle est écrite en C, mais peut également être utilisée en C++ et de nombreux portages existent dans d'autres langages, notamment en C#, en OCaml ou encore en Python. Depuis sa version 2, elle est placée sous la licence [zlib](#) [↗](#).

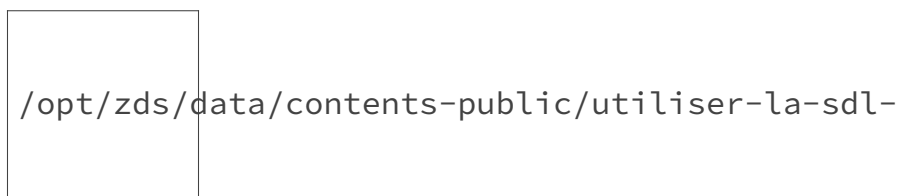


FIGURE 1.1. – Logo de la Simple Direct MediaLayer

### 1.1.2. Que peut faire la SDL

Très bien, la SDL est une bibliothèque multimédia.

?

Mais concrètement, quel type de projet peut-on faire avec?

La SDL permet de faire différents types de projet. Elle est typiquement utilisée pour faire des jeux 2D, mais ce n'est pas sa seule utilité. Elle se veut être une bibliothèque qui permette de faire assez de choses. Notamment, la version 2 de la bibliothèque, sortie en 2013, rajoute des fonctionnalités utiles.

## 1. La SDL

Pour faire un petit jeu 2D et s’amuser en C, c’est un bon choix. La SDL est assez généraliste pour nous permettre de faire n’importe quel type de jeu. Réaliser un Tétris, un Casse-brique, un Pong ou un Bomberman ne pose pas de problèmes. On peut même faire:

- des «*Shoot ’em up*» (Space Invaders);
- des jeux de combats;
- des jeux de plateforme (Super Mario Bros par exemple);
- des RPG 2D (Zelda par exemple).

Bien sûr, réaliser ces jeux demande plusieurs compétences. Par exemple, dans le cas du RPG, il faudra gérer:

- le «*tile mapping*»;
- les collisions;
- les évènements;
- les combats.

Tout ça sans oublier le son, les animations et tout le contenu. Ce travail est facilité par l’utilisation d’un moteur de jeu (nous pouvons créer un moteur de jeu minimaliste avec la SDL pour nous faciliter ce travail).



FIGURE 1.2. – Portage sous Android du jeu Blip & Blop (*Shoot ’em up* à scrolling horizontal).

La SDL peut également être utilisée pour créer de petits utilitaires, mais ce n’est clairement pas son rôle premier. Dans ce cas, il faut plutôt se renseigner sur des bibliothèques comme [GTK+](#) [↗](#).

### 1.1.3. Pourquoi choisir la SDL

C’est une question essentielle.



Quels sont les avantages de la SDL sur les autres bibliothèques?

Le plus grand avantage de la SDL est sa grande souplesse. En effet, comme nous l’avons dit, elle permet de faire des petits jeux plus ou moins facilement.

De plus, elle ne dispose pas de beaucoup de fonctions (moins de 600), et est donc assez rapide à appréhender. Son utilisation n’est cependant pas simple, mais elle reste un bon choix, notamment pour l’apprentissage. Notons que de petites bibliothèques viennent compléter la SDL ce qui nous permet par exemple de manipuler les réseaux ou le son plus facilement qu’avec seulement la SDL.

La licence peut aussi motiver le choix de cette bibliothèque. En voici les termes.

## 1. La SDL

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions :

1. The origin of this software must not be misrepresented ; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Nous pouvons utiliser la SDL dans n'importe quel projet, libre ou non, gratuit ou payant, et nous pouvons même modifier la SDL; tout cela avec très peu de restrictions. Ce qui nous intéresse dans ce tutoriel, ce n'est pas de modifier la SDL, mais seulement de l'utiliser. La seule restriction à cela est de ne pas mentir en revendiquant la création de la SDL. La licence ne nous oblige même pas à mentionner l'usage de la SDL (même si c'est apprécié)!

Un autre argument pour l'utilisation de la SDL est le fait qu'elle soit multiplateforme. Nous pouvons en effet réaliser des programmes pour Windows, Linux, OSX et pour une multitude d'autres plateformes. Nous pouvons même écrire des programmes pour Android!

Bien sûr, le meilleur argument reste les projets qu'elle permet de réaliser.

## 1.2. Configurer un projet avec la SDL

### 1.2.1. Windows

Les utilisateurs de Code::Blocks ou de MinGW peuvent utiliser [ce tutoriel](#) qui les aidera à configurer un projet qui utilise la SDL. Sinon, Internet et le site de la SDL donnent des informations à ce propos.

### 1.2.2. Linux

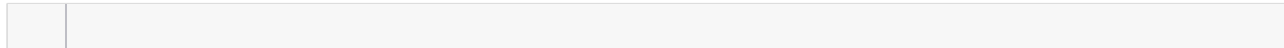
Sous Linux, la méthode la plus simple est d'utiliser son gestionnaire de paquet pour installer le paquet de développement de la SDL. Par exemple, sous Debian et ses dérivés, cette commande peut être utilisée.

Une grande majorité des distributions disposent du paquet de développement (peut-être sous un autre nom que `libSDL2-dev`) dans leurs paquets, mais s'il n'est pas disponible, il est également possible de compiler les sources soi-même (voir la [page d'installation de la SDL](#) ).

Une fois ceci fait, il nous faut juste modifier notre ligne de compilation pour indiquer que nous utilisons la SDL. Avec `gcc` par exemple, notre ligne pour compiler un fichier ressemblera à ça (nous conseillons de rajouter d'autres options de compilation comme `-Wall` et `-Wextra`).



## 1. La SDL



Nous indiquons juste qu'il faut lier la SDL à notre programme.

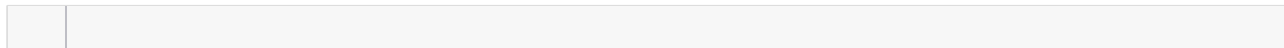
[Ce billet](#) fournit quelques explications pour une utilisation de la SDL avec Code::Blocks sous Linux.

### 1.2.3. OSX

Le site de la SDL (et d'autres ressources sur Internet) indiquent comment configurer un projet pour utiliser la SDL.

---

Pour vérifier que tout va bien, nous pouvons compiler ce code.



Normalement, la compilation devrait se faire sans erreur. Le programme créé ne fait rien, mais dès le prochain chapitre, nous verrons comment ouvrir une fenêtre.

## 1.3. Prérequis et état d'esprit du tutoriel

### 1.3.1. Prérequis

Pour suivre ce tutoriel, certaines connaissances sont requises. Des bases en langage C naturellement. Il n'est pas nécessaire d'être particulièrement à l'aise avec le langage (ce tutoriel peut d'ailleurs servir à pratiquer) quelques connaissances sont quand même indispensables:

- les pointeurs et les tableaux;
- la gestion de la mémoire;
- les structures;
- la gestion des fichiers.

De plus, l'instruction `goto` sera utilisée à des fins de gestion d'erreur. Il faut juste voir son fonctionnement et savoir à quoi elle sert, son utilisation dans ce tutoriel étant assez basique.



Il ne sert à rien d'essayer d'utiliser une bibliothèque en C sans avoir acquis les bases non seulement du langage, mais aussi de la programmation.

[Ce tutoriel](#) par exemple, apprend à manipuler les notions citées. Le but n'est pas de le survoler, mais de prendre son temps, de bien l'exploiter, de regarder d'autres ressources, afin de connaître les bases nécessaires pour suivre ce tutoriel.

### 1.3.2. Le but du tutoriel

Le but de ce tutoriel est de présenter la SDL bien entendu. Mais un but plus profond est de présenter une première bibliothèque, d'apprendre à l'utiliser sans se sentir perdu, et de devenir indépendant.

Un autre des buts de ce tutoriel est de donner de bonnes habitudes. Le C est un langage difficile à bien utiliser. Il demande de la rigueur et écrire un code propre n'est pas facile. Ainsi, en proposant un code propre et gérant les erreurs et autres cas critiques pouvant toucher le programme, nous espérons aider le lecteur à acquérir de bonnes pratiques. Cependant, tous les codes ne feront pas de vérification (mais il faudra quand même les faire).

Finalement, le but (et c'est quand même le principal) est que cette présentation de la SDL soit bénéfique et apprenne à utiliser une nouvelle bibliothèque. Si quelqu'un a pour but de faire un petit programme et réussit, grâce à ce tutoriel, à le faire avec la SDL, alors ce but a été atteint.




Ce tutoriel ne présente **que** la SDL et pas des méthodes de conception de projet qui sont nécessaires pour mener correctement un projet.

Ce tutoriel présente parfois des notions qui ne sont pas forcément nécessaires pour créer un jeu. Le chapitre sur la modification pixels par pixels n'est par exemple pas essentiel à la compréhension de ce tutoriel et est en plus assez technique (mais il reste très intéressant). Nous pourrions donc passer dessus rapidement voire le sauter quitte à revenir dessus plus tard.

Néanmoins, il nous faut veiller à passer suffisamment de temps sur chaque chapitre et à pratiquer ce que nous apprenons. La programmation est un domaine où lire passivement n'aide pas à la progression, et bien que le tutoriel possède des TPs, il faut également pratiquer de son côté et faire ses propres tests pour s'approprier les notions vues.

### 1.3.3. La documentation

En programmation en général, mais en particulier pour apprendre à utiliser une bibliothèque ou un outil inconnu, la documentation est un ami dont on ne peut pas se passer. Ce tutoriel s'appuiera beaucoup sur elle. Pour chaque fonction présentée, nous verrons la page de la documentation correspondante. Elle nous procurera notamment, le prototype de la fonction, ce qui nous permettra d'analyser ses arguments et sa valeur de retour. Elle nous renverra de plus vers d'autres fonctions proches qui pourront être utiles.

La documentation nous donnera également des informations sur la fonction (les erreurs qu'elle peut rencontrer par exemple) et des exemples d'utilisation et on peut même y trouver une [page au sujet de l'installation](#) .

Tout cela nous montre bien combien la documentation est utile. Savoir la lire et l'utiliser est donc une compétence non négligeable (voire indispensable). Espérons que ce tutoriel aidera à prendre conscience de son utilité.

## 1. La SDL



La documentation de la SDL (comme la documentation de la plupart des bibliothèques) est en anglais. Ce n'est pas de l'anglais très compliqué et avec un minimum d'efforts, même un débutant pourra comprendre ce qui est écrit assez rapidement.

Voici le [lien de la documentation](#)  . Nous pouvons déjà la parcourir et voir comment elle est construite avant de passer à la suite.

---

Dans ce chapitre introductif, nous n'avons pas du tout programmé, mais nous avons vu ce qu'était la SDL, les possibilités qu'elle donnait et comment ce tutoriel allait se présenter. Dans le chapitre suivant, nous commencerons à programmer, en voyant comment manipuler les fenêtres.

## 2. Les premières fenêtres

Dans ce chapitre, nous commencerons enfin à utiliser la SDL. Nous verrons comment initialiser la SDL, comment créer une fenêtre et comment la gérer.

### 2.1. Initialiser la SDL

Après avoir configuré notre projet, nous pouvons enfin commencer à utiliser la SDL.

#### 2.1.1. Initialiser la SDL

Voyons d'abord le code de base que nous devrons utiliser avec la SDL:

Un code tout à fait basique composé d'une fonction `main` et d'une directive de préprocesseur pour inclure la SDL. Maintenant, commençons à utiliser la SDL.

Pour être utilisée, la SDL doit d'abord être initialisée. Pour cela, nous devons utiliser la fonction [SDL\\_Init](#). Cette fonction doit être utilisée avant toutes les autres fonctions de la SDL car elle charge tout ce dont les autres fonctions ont besoin pour fonctionner. Voici son prototype.

Cette fonction prend en paramètre une liste de drapeaux sous la forme d'un entier, ceux-ci correspondant aux sous-systèmes que l'on veut initialiser. La SDL est en effet composée de plusieurs parties. Voici ses différentes parties et le drapeau à fournir pour l'initialiser.

Drapeaux	Description
<b>SDL_INIT_TIMER</b>	Initialise le système de gestion du temps
SDL_INIT_AUDIO	Initialise le système de gestion de l'audio
SDL_INIT_VIDEO	Initialise le système de gestion de rendu
SDL_INIT_JOYSTICK	Initialise le système de gestion des joysticks
SDL_INIT_GAMECONTROLLER	Initialise le système de gestion des contrôleurs de jeux

## 2. Les premières fenêtres

<code>SDL_INIT_EVENTS</code>	Initialise le système de gestion des évènements
<code>SDL_INIT EVERYTHING</code>	Permet de tout initialiser

Tous les drapeaux ne sont pas présents, mais nous avons le lien vers la documentation pour voir les autres.

*i*

Pour charger plusieurs systèmes à la fois, nous devons utiliser l'opérateur `|`.

Par exemple, pour charger les systèmes audio et vidéo, il nous faudra utiliser cette ligne.

La fonction `SDL_Init` peut retourner deux valeurs:

- `0` si l'initialisation s'est bien passée;
- une valeur négative si l'initialisation n'a pas pu se faire correctement.

Nous voulons faire des fenêtres, nous avons donc besoin du système de rendu de la SDL c'est-à-dire du drapeau `SDL_INIT_VIDEO`.

*i*

La fonction `SDL_Init` permet de charger la SDL. C'est donc le point de départ d'un programme en SDL. Aucune fonction de la SDL ne doit être utilisée avant elle.

Un code pour initialiser la SDL serait donc le suivant.

### 2.1.2. Quitter la SDL

Cependant, la fonction `SDL_Init`, même si nous ne savons pas comment elle fonctionne, fait certainement quelques allocations, et il faut donc désallouer tout ça. Pour cela, la SDL nous fournit une fonction, la fonction `SDL_Quit` [↗](#). Voici son prototype.

Elle ne prend pas d'arguments et ne renvoie rien. Elle permet juste de quitter proprement la SDL.



Toutes nos manipulations doivent être effectuées entre la fonction `SDL_Init` qui est le point de départ de notre programme et la fonction `SDL_Quit` qui est son point d'arrivée.

Au vu de cela, notre code n'est pas correct. Nous n'avons pas quitté la SDL. Le bon code...

### 2.1.3. Gestion des erreurs

La fonction `SDL_Init` comme beaucoup d'autres fonctions de la SDL que nous verrons peut échouer. Dans ce cas, elles renvoient une valeur d'erreur (une valeur négative pour `SDL_Init`). Cela nous permet de ne pas poursuivre le programme en cas d'erreur. Cependant, nous aimerions bien connaître la raison de cette erreur. Les fonctions `strerror` et `perror` (déclarées dans l'en-tête `stdio.h`) permettent d'obtenir des informations sur les erreurs des fonctions standards. La SDL dispose d'une fonction identique, la fonction `SDL_GetError` [↗](#) dont le prototype est le suivant.

Cette fonction ne prend aucun paramètre et renvoie une chaîne de caractères qui est un message à propos de la dernière erreur que la SDL a rencontrée. Nous pouvons donc l'écrire dans le flux de sortie standard d'erreur (c'est-à-dire `stderr`).



Pour utiliser `stderr`, il est nécessaire d'inclure l'en-tête `stdio.h`.

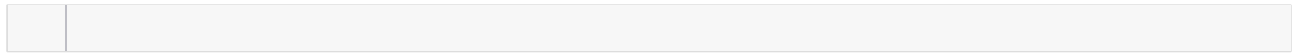
Grâce à ceci, on peut compléter notre code précédent.

Et après avoir appris tout ceci, nous pouvons visiter la page de la [documentation de la SDL à propos de l'initialisation](#) [↗](#). Nous pourrions y découvrir les fonctions de la SDL ayant un rapport avec son initialisation (après tout, comme nous l'avons dit, un tutoriel ne peut pas être exhaustif, et à un moment, la documentation devient essentielle pour apprendre de nouvelles choses).

### 2.2. Créer des fenêtres

#### 2.2.1. Créer une fenêtre

Notre code précédent initialise peut-être la SDL, mais il ne fait rien d'autre. Notre but est de créer des fenêtres, voyons comment faire. La SDL nous propose une fonction qui permet de créer une fenêtre très facilement, la fonction [SDL\\_CreateWindow](#). Voici son prototype.



Elle prend plusieurs paramètres:

- **title** est une chaîne de caractères et correspond au nom de la fenêtre;
- **x** et **y** correspondent respectivement aux positions de la fenêtre sur l'axe **x** et **y** de l'écran, le point **(0, 0)** étant placé en haut à gauche (les valeurs **SDL\_WINDOWPOS\_UNDEFINED** et **SDL\_WINDOWPOS\_CENTERED** peuvent être passées en paramètre pour indiquer de placer la fenêtre à n'importe quelle position sur cet axe ou de la centrer sur cet axe);
- **w** et **h** correspondent à la largeur (*width*) et à la hauteur (*height*) de la fenêtre;
- **flags** correspond à une série de drapeaux qui permettent de choisir des options pour la fenêtre.

Voici quelques drapeaux possibles.

Drapeaux	Description
<b>SDL_WINDOW_FULLSCREEN</b>	Crée une fenêtre en plein écran
<b>SDL_WINDOW_FULLSCREEN_DESKTOP</b>	Crée une fenêtre en plein écran à la résolution du bureau
<b>SDL_WINDOW_SHOWN</b>	Crée une fenêtre visible
<b>SDL_WINDOW_HIDDEN</b>	Crée une fenêtre non visible
<b>SDL_WINDOW_BORDERLESS</b>	Crée une fenêtre sans bordures
<b>SDL_WINDOW_RESIZABLE</b>	Crée une fenêtre redimensionnable
<b>SDL_WINDOW_MINIMIZED</b>	Crée une fenêtre minimisée
<b>SDL_WINDOW_MAXIMIZED</b>	Crée une fenêtre maximisée



Là encore, nous pouvons utiliser l'opérateur **|** pour choisir plusieurs drapeaux. De plus, le drapeau **SDL\_WINDOW\_SHOWN** est un drapeau par défaut. La fenêtre est toujours visible sauf si le drapeau **SDL\_WINDOW\_HIDDEN** a été passé en paramètre. Cependant, nous devons tout de même donner un paramètre à la fonction, donc, dans le cas où nous voulons juste que la fenêtre soit visible, nous pouvons passer **0** en paramètre (**0** signifie qu'on n'envoie pas de drapeaux, et donc la fenêtre sera visible puisqu'elle l'est par défaut). Cependant, dans un souci de clarté, nous préférons utiliser le drapeau **SDL\_WINDOW\_SHOWN**.

## 2. Les premières fenêtres

La fonction `SDL_CreateWindow` retourne un pointeur sur `SDL_Window`. C'est une structure de la SDL qui représente la fenêtre créée. On peut donc dire que la fonction `SDL_CreateWindow` retourne la fenêtre créée. Il nous faut récupérer ce pointeur car dès que nous voudrions agir sur la fenêtre, nous aurons besoin de lui.

En cas d'erreur, elle retourne `NULL` et les informations sur l'erreur peuvent être obtenus comme tout à l'heure à l'aide de la fonction `SDL_GetError`. Cela nous permet d'arriver à ce code.

Avec ce code, notre fenêtre apparaît et... Disparaît immédiatement. Ceci est tout à fait normal: juste après l'avoir ouverte on quitte la SDL. Pour laisser notre fenêtre à l'écran, nous allons utiliser une autre fonction de la SDL, la fonction `SDL_Delay` [↗](#) dont voici le prototype.

Cette fonction prend en paramètre un nombre entier. Ce nombre correspond à un nombre de millisecondes durant lequel le programme sera en pause. Ainsi, en arrêtant le programme pendant trois secondes avant de quitter la SDL, notre fenêtre restera à l'écran pendant trois secondes. On placera donc cette ligne avant notre `SDL_Quit`.

### 2.2.2. Détruire la fenêtre

Et là, tout comme il faut quitter la SDL après l'avoir initialisée, il faut **obligatoirement** détruire la fenêtre après l'avoir créée. Pour cela, nous allons utiliser la fonction `SDL_DestroyWindow` [↗](#) dont le prototype est le suivant.

Elle prend en argument un pointeur sur `SDL_Window` c'est-à-dire le pointeur qui représente la fenêtre qui doit être détruite (quand on disait qu'il fallait récupérer la valeur retournée par `SDL_CreateWindow` c'était pas pour rien) et ne retourne rien. Notre code devient le suivant.

?

Pourquoi doit-on passer à la fonction `SDL_DestroyWindow` la variable `window`? Il n'y a qu'une seule fenêtre à détruire, il peut très bien s'en passer, non?

Non, il ne peut pas s'en passer, pour la simple raison qu'avec la version 2 de la SDL, on peut créer plusieurs fenêtres. Essayons ce code.



## 2. Les premières fenêtres

Deux fenêtres sont créées. On aurait très bien pu vouloir fermer la première avant d'ouvrir la deuxième ou faire des opérations entre les deux fermetures. Cela n'est possible que parce qu'on ferme séparément chaque fenêtre grâce au paramètre de la fonction `SDL_DestroyWindow`.

### 2.2.3. Un bon code

Bon, maintenant que nous savons ouvrir une fenêtre et la refermer, analysons notre code. Avons-nous un bon code? Regardons :

- nous utilisons `SDL_Quit` pour quitter la SDL;
- nous détruisons la fenêtre créée;
- nous testons le retour des fonctions qui peuvent échouer.

Le code a l'air bien. Maintenant, posons-nous une autre question.

?

Faisons-nous ces actions dans tous les cas?

Et là, catastrophe, la réponse est non. Par exemple, si la création de la fenêtre a échoué, nous quittons le programme **sans quitter la SDL**.

Il faut faire en sorte de quitter la SDL dans tous les cas (et il faudra faire en sorte de fermer toutes nos fenêtres et de désallouer toute la mémoire allouée). Nous pourrions utiliser des `if` imbriqués, mais une fois que l'on fera des programmes un peu plus long ce ne sera plus maintenable. Nous allons plutôt utiliser la gestion des erreurs évoquée [ici](#) pour nous en tirer. Nous allons donc placer un label dans notre code, juste avant `SDL_Quit`. Plus tard, nous en placerons d'autre si besoin. On obtient ce code.

## 2.3. Gérer la fenêtre

Après avoir vu comment créer des fenêtres, nous allons les manipuler un peu. Nous n'allons rien faire de bien compliqué, juste voir quelques fonctions de la SDL à propos des fenêtres. Ceci permettra de pratiquer un peu et de voir déjà quelques-unes des possibilités offertes par la SDL.

### 2.3.1. Les paramètres de la fenêtre

Pour commencer, nous allons voir comment changer (et obtenir) les paramètres de la fenêtre.



### De quels paramètres parlons-nous?

Nous parlons des paramètres que nous avons choisis lors de la création de la fenêtre, c'est-à-dire:

- la taille de la fenêtre;
- la position de la fenêtre;
- le titre de la fenêtre;
- les drapeaux.

La SDL offre des fonctions pour changer tous ces paramètres et ce sont ces fonctions que nous allons maintenant voir. Pour nous faciliter la tâche, elle a le bon goût d'avoir toutes ses fonctions construites de la même manière:

- les fonctions pour changer les paramètres commencent toutes par `SDL_SetWindow`;
- les fonctions pour obtenir les paramètres commencent toutes par `SDL_GetWindow`;
- le premier paramètre de toutes ces fonctions est un pointeur sur `SDL_Window`, c'est-à-dire la fenêtre concernée.

Grâce à ceci nous pouvons quasiment deviner toutes les fonctions que nous allons voir.

#### 2.3.1.1. Le titre de la fenêtre

La fonction pour obtenir le titre d'une fenêtre est la fonction `SDL_GetWindowTitle` [↗](#). Son prototype:

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

Nous aurions pu le deviner, elle prend en paramètre (comme prévu) la fenêtre dont on veut déterminer le titre et retourne une chaîne de caractères correspondant au titre en question.

La fonction pour donner un titre à une fonction n'est guère plus compliquée à comprendre. Il s'agit de la fonction `SDL_SetWindowTitle` [↗](#). Son prototype:

```
void SDL_SetWindowTitle(SDL_Window * window, const char * title);
```

Elle ne retourne rien et prend deux paramètres, la fenêtre dont on veut changer le titre et le nouveau titre à donner sous la forme d'une chaîne de caractères.

#### 2.3.1.2. La position de la fenêtre

La position de la fenêtre est obtenue grâce à la fonction `SDL_GetWindowPosition` [↗](#). Son prototype:

```
void SDL_GetWindowPosition(SDL_Window * window, int * x, int * y);
```

## 2. Les premières fenêtres

On retrouve en paramètre la fenêtre, mais surprise, la fonction ne renvoie rien et prend aussi en paramètre deux pointeurs sur des entiers. Ces deux pointeurs correspondent à la position en X et en Y de la fenêtre. En effet, on ne peut pas renvoyer deux valeurs et donc la fonction modifie les deux valeurs pointées pour qu'elles valent finalement la position de la fenêtre.

La fonction pour changer la position de la fenêtre est la fonction [SDL\\_SetWindowPosition](#). Son prototype:

```
void SDL_SetWindowPosition(SDL_Window *window, int *x, int *y);
```

Son prototype est vraiment très proche de celui de la fonction `SDL_GetWindowPosition`. Elle prend en paramètre la fenêtre dont la position doit être changée, la nouvelle position en X et la nouvelle position en Y.

i

Notons que les nouvelles positions en X et Y peuvent aussi être `SDL_WINDOWPOS_CENTERED` ou `SDL_WINDOWPOS_UNDEFINED` que nous avons vu précédemment.

### 2.3.1.3. La taille de la fenêtre

Pour obtenir la taille de la fenêtre, il nous faut utiliser la fonction [SDL\\_GetWindowSize](#). Son prototype:

```
void SDL_GetWindowSize(SDL_Window *window, int *w, int *h);
```

Elle s'utilise comme la fonction `SDL_GetWindowPosition`. Elle prend en paramètre la fenêtre dont on veut obtenir la taille, et deux pointeurs sur `int`. La fonction modifie les valeurs pointées, et ce sont ces deux valeurs qui valent la largeur (paramètre `w`) et la hauteur (paramètre `h`) de la fenêtre.

De même, la fonction [SDL\\_SetWindowSize](#) qui permet de changer la taille d'une fenêtre s'utilise de la même manière que la fonction `SDL_GetWindowPosition`. Son prototype:

```
void SDL_SetWindowSize(SDL_Window *window, int w, int h);
```

Elle prend en paramètre la fenêtre dont on veut changer la taille, sa nouvelle largeur et sa nouvelle hauteur.

### 2.3.1.4. Les drapeaux de la fenêtre

Les drapeaux de la fenêtre peuvent être obtenus en utilisant la fonction [SDL\\_GetWindowFlags](#). Son prototype:

```
uint_t SDL_GetWindowFlags(SDL_Window *window);
```

## 2. Les premières fenêtres

Elle prend en paramètre la fenêtre dont on veut obtenir les drapeaux et retourne un entier.



Quoi? Un **entier**? Mais comment tester si un drapeau est présent alors?

La fonction renvoie le même type de données qu'on a passé à la fonction `SDL_Init`. Pour savoir si un drapeau est présent, il faut tout simplement utiliser l'opérateur `&` avec le drapeau et le retour de la fonction. Par exemple, pour regarder si la fenêtre est redimensionnable, on peut utiliser ce code:

```
if (SDL_GetWindowFlags(window) & SDL_WINDOW_RESIZABLE) {
```


Pour tester plusieurs drapeaux simultanément, il faut aussi utiliser l'opérateur `&` mais cette fois avec les différents drapeaux sur lesquels on aura utilisé l'opérateur `|`. Par exemple, pour tester si la fenêtre est redimensionnable **et** si elle n'a pas de bordures, on peut utiliser ce code:

```
if (SDL_GetWindowFlags(window) & (SDL_WINDOW_RESIZABLE | SDL_WINDOW_BORDERLESS)) {
```

### 2.3.2. Agir sur la fenêtre

On ne peut pas changer directement les drapeaux d'une fenêtre (il n'existe pas de fonction `SDL_SetWindowFlags`), mais on dispose de plusieurs fonctions pour faire des actions telles que réduire la fenêtre. Nous allons voir quelques-unes de ces fonctions.

#### 2.3.2.1. Agrandir, réduire et restaurer la fenêtre

Pour agrandir la fenêtre, nous devons utiliser la fonction [SDL\\_MaximizeWindow](#) . Son prototype:


```
void SDL_MaximizeWindow(SDL_Window *window);
```

Elle prend simplement en paramètre la fenêtre à agrandir.

La fonction permet de minimiser la fenêtre dans la barre des tâches. Son prototype:

```
void SDL_MinimizeWindow(SDL_Window *window);
```

Elle prend elle aussi comme unique paramètre la fenêtre à minimiser.

Et finalement, pour restaurer une fenêtre, il faut utiliser la fonction [SDL\\_RestoreWindow](#) . Son prototype:

```
void SDL_RestoreWindow(SDL_Window *window);
```

## 2. Les premières fenêtres


Nous aurions pu deviner son prototype, elle prend juste en paramètre la fenêtre à restaurer.

### 2.3.2.2. La visibilité de la fenêtre


Une fenêtre peut avoir trois états:

- cachée;
- visible;
- devant toutes les autres fenêtres.


La SDL nous offre des fonctions pour placer une fenêtre dans l'un de ces états. Là encore, nous verrons que les noms sont assez explicites.

Pour cacher la fenêtre (donc le drapeau `SDL_WINDOW_HIDDEN`) nous utiliserons la fonction [SDL\\_HideWindow](#) . Son prototype:

```
void SDL_HideWindow(SDL_Window *window);
```

Pour la montrer (le drapeau `SDL_WINDOW_SHOWN`), nous utiliserons la fonction [SDL\\_ShowWindow](#) . Son prototype:


```
void SDL_ShowWindow(SDL_Window *window);
```

Et finalement pour la mettre en avant-plan (devant toutes les autres fenêtres), nous utiliserons la fonction [SDL\\_RaiseWindow](#) . Son prototype:

```
void SDL_RaiseWindow(SDL_Window *window);
```

Nous pouvons le voir, ces fonctions sont très simples à utiliser. Elles prennent en paramètre la fenêtre sur laquelle on veut agir et ne renvoient pas de valeur.

### 2.3.2.3. Le plein écran

Nous pouvons placer notre fenêtre en plein écran durant sa création, mais nous pouvons également le faire plus tard. Par exemple, dans un jeu, nous pourrions proposer à l'utilisateur de mettre la fenêtre en plein écran ou non. La fonction [SDL\\_SetWindowFullscreen](#)  permet de le faire. Son prototype:

```
int SDL_SetWindowFullscreen(SDL_Window *window, Uint32 flags);
```

Elle prend en paramètre la fenêtre qu'on veut gérer et un drapeau. Ce drapeau peut-être:

- `SDL_WINDOW_FULLSCREEN` pour placer la fenêtre en plein écran;
- `SDL_WINDOW_FULLSCREEN_DESKTOP` pour placer la fenêtre en plein écran à la résolution du bureau;
- `0` pour placer la fenêtre en mode fenêtré.

## 2. Les premières fenêtres

Les drapeaux sont les mêmes que ceux que l'on utilisait déjà avec la fonction `SDL_CreateWindow`, alors nous ne sommes pas dépayés.

La fonction peut malheureusement échouer et sa valeur de retour, un entier, permet de savoir si elle a échoué. Elle renvoie `0` en cas de succès et une valeur négative s'il y a eu une erreur. Il suffit ensuite d'utiliser la fonction `SDL_GetError` pour obtenir l'erreur.

*i*

Généralement, on ne quitte pas un programme parce que cette fonction a échoué. On se contente de noter l'erreur et de continuer le programme après avoir averti l'utilisateur qu'il était impossible de changer de mode.

Maintenant, comme pour l'initialisation, nous pouvons consulter la [page de la documentation de la SDL à propos de la gestion des fenêtres](#) [↗](#). Elle est un peu plus grande que celle à propos de l'initialisation, mais nous y trouverons sûrement quelques fonctions intéressantes.

---

Nous savons maintenant comment créer et manipuler des fenêtres. Mais s'il y a une chose à retenir de ce chapitre c'est qu'il ne faut pas:

- oublier de vérifier le retour des fonctions à risque;
- oublier de libérer les ressources.

## 3. Dessiner dans la fenêtre

Après avoir vu comment manipuler des fenêtres, nous allons maintenant dessiner dedans.

### 3.1. Gestion du rendu

Il nous faut nous occuper du rendu de la fenêtre. Pour cela, nous devons créer un *renderer*. Il s'agit d'une structure (`SDL_Renderer`). Chaque *renderer* est associé à une fenêtre et nous permet de dessiner dans celle-ci. C'est de cette manière que nous allons modifier ce qu'il y a dans la fenêtre.

#### 3.1.1. Créer un renderer

Pour créer un *renderer*, il faut utiliser la fonction `SDL_CreateRenderer` [↗](#). Son prototype:

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

Elle prend ces paramètres:

- `window` correspond à la fenêtre à laquelle nous voulons que notre *renderer* soit associé;
- `index` correspond au pilote à utiliser pour gérer le *renderer* (on lui donne généralement la valeur `-1` qui permet de choisir le premier qui correspond);
- `flags` correspond à une liste de drapeaux.

Les différents drapeaux possibles sont:

drapeaux	Description
<code>SDL_RENDERER_SOFTWARE</code>	Le <i>renderer</i> est logiciel, le rendu sera effectué par le CPU et les données seront stockées en mémoire vive.
<code>SDL_RENDERER_ACCELERATED</code>	Le <i>renderer</i> utilise l'accélération matérielle. Les données sont en mémoire vidéo, plus rapide que la mémoire vive.
<code>SDL_RENDERER_PRESENTVSYNC</code>	La mise à jour de la fenêtre de rendu est synchronisé avec la fréquence de rafraîchissement de l'écran.

### 3. Dessiner dans la fenêtre

La plupart du temps, nous voulons un rendu avec l'accélération matérielle (utilisant la carte graphique), mais dans le cas où celle-ci n'est pas disponible on peut utiliser un rendu logiciel (c'est donc une solution de repli). Synchroniser le rendu avec la fréquence de rafraîchissement de l'écran peut être une bonne idée, mais la plupart du temps, on préférera le gérer nous-mêmes.

Nous pouvons également passer `0` comme argument, dans ce cas, la SDL essaye de fournir un *renderer* qui utilise l'accélération matérielle.

La fonction `SDL_CreateRenderer` retourne un pointeur sur `SDL_Renderer`, pointeur que nous devons récupérer car nous en aurons besoin dès que nous voudrions dessiner un peu. Si la création du *renderer* a échoué, elle retourne `NULL`. Il ne faut donc pas oublier de tester la valeur retournée. C'est là qu'intervient le *renderer* logiciel, on l'utilise généralement si on ne peut pas utiliser l'accélération matérielle. On devrait donc fonctionner de la manière suivante.

1. On essaye de créer un *renderer* avec le drapeau `SDL_RENDERER_ACCELERATED`.
2. Si la création du *renderer* a échoué on essaye avec le drapeau `SDL_RENDERER_SOFTWARE`.
3. Si aucun n'a marché, on quitte le programme.



En fait, la SDL fait ceci automatiquement quand l'index passé à `SDL_CreateRenderer` vaut `-1`.

Nous nous contenterons donc d'essayer le drapeau `SDL_RENDERER_ACCELERATED`. La fonction `SDL_RendererInfo` nous permet d'obtenir des informations sur un *renderer* et permet donc de connaître ses drapeaux après l'avoir créé, mais nous n'allons pas nous en soucier ici.

#### 3.1.2. Détruire le renderer

Désolé, mais nous allons encore une fois gérer nos ressources puisqu'il faut détruire le *renderer* créé. Pour cela, nous allons utiliser la fonction `SDL_DestroyRenderer`. Son prototype:

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

Elle prend en paramètre le *renderer* qu'il faut détruire et ne retourne rien. Grâce à ça, nous pouvons faire notre premier code avec un *renderer*:

```
SDL_Renderer *renderer = NULL;
```

Dans l'ordre, nous initialisons la SDL, puis nous créons la fenêtre et enfin nous créons le *renderer*. Nous faisons les destructions dans l'autre sens, le *renderer*, la fenêtre et enfin nous terminons en quittant la SDL. Sans oublier bien sûr de vérifier le retour de nos fonctions.



#### 3.1.3. Créer le `renderer` et la fenêtre en même temps

Nous avons lié le `renderer` à une fenêtre et avons dit que chaque `renderer` était associé à une fenêtre. Ce serait donc bien de pouvoir créer le `renderer` et la fenêtre en même temps. Tant mieux, la SDL a une fonction qui permet de faire ça. Il s'agit de la fonction `SDL_CreateWindowAndRenderer` [↗](#). Son prototype:

Juste en voyant son prototype, nous devrions comprendre comment elle marche. Elle prend en paramètre:

- la largeur de la fenêtre;
- la hauteur de la fenêtre;
- les drapeaux de la fenêtre (vus dans le [chapitre précédent](#) [↗](#) ;
- un double pointeur sur `SDL_Window`;
- un double pointeur sur `SDL_Renderer`.

Les deux doubles pointeurs sont compréhensibles: la fonction doit modifier la valeur des deux pointeurs pour qu'ils pointent sur la fenêtre et le `renderer` créé, or, pour modifier la valeur d'une variable dans une fonction, il faut lui passer l'adresse de cette variable, c'est-à-dire des doubles pointeurs dans notre cas. Aucun drapeau n'est demandé pour le `renderer`, la fonction essaye de créer un `renderer` avec le drapeau `SDL_RENDERER_ACCELERATED`.

La fonction retourne un entier. Il s'agit de:

- `0` si tout s'est bien passé;
- `-1` en cas d'erreur.

On peut comme d'habitude récupérer l'erreur en question avec la fonction `SDL_GetError`.

## 3.2. Des rectangles et des couleurs

Faisons maintenant, un aparté sur quelque chose qui nous sera très, mais vraiment très utile par la suite.

### 3.2.1. La base de tout, le point

La base de l'affichage d'un ordinateur est le pixel. Et un pixel c'est un point, un petit point d'un écran. Il paraît donc logique que la SDL puisse représenter un point.

?

Mais comment représenter un point?

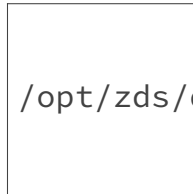
Comme en mathématiques, on représente un point avec son abscisse et son ordonnée. Ainsi, un point est représenté en SDL par une structure qui a pour champ ces deux composantes. Il s'agit de la structure `SDL_Point` [↗](#). Ses champs:

### 3. Dessiner dans la fenêtre

- `x` représente l'abscisse du point ;
- `y` représente l'ordonnée du point.



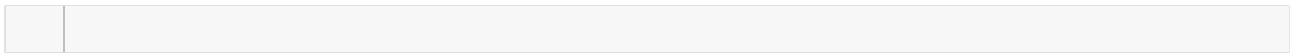
Le point de coordonnées  $(0, 0)$  est en haut à gauche, et si les abscisses augmentent en allant vers la droite, les ordonnées, elles, augmentent en allant vers le bas.



/opt/zds/data/contents-public/utiliser-la-sdl-

FIGURE 3.1. – Le système de coordonnées de la SDL.

On crée donc un point de cette manière:



#### 3.2.2. On complète la base, le rectangle

On a dit que la base de tout était le pixel, mais en fait, avec la SDL, ce que l'on peut vraiment considérer comme base est plutôt le rectangle. On pourrait se dire qu'un rectangle n'est qu'une amélioration du point et que le point reste donc la base, mais on utilise le rectangle vraiment beaucoup plus que le point.



Comment pouvons-nous définir un rectangle?

On peut répondre plusieurs choses:

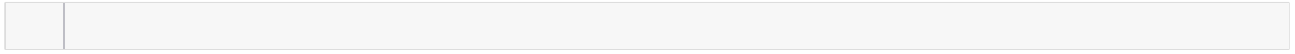
- on peut définir un rectangle grâce à quatre points (ses quatre coins) ;
- on peut définir un rectangle grâce à deux points (deux coins opposés) ;
- on peut définir un rectangle grâce à deux droites perpendiculaires ;
- plusieurs autres solutions.

Pour faire son rectangle la SDL a choisi une autre solution. Afin de simplifier les choses, son rectangle a toujours ses côtés parallèles aux axes. Ce rectangle est représenté grâce à la structure `SDL_Rect` [↗](#). Ses champs:

- `x` représente l'abscisse du coin en haut à gauche du rectangle ;
- `y` représente l'ordonnée du coin en haut à gauche du rectangle ;
- `w` représente la largeur du rectangle ;
- `h` représente la hauteur du rectangle.

### 3. Dessiner dans la fenêtre

Comme nous pouvons le voir, ces quatre variables permettent bien de définir entièrement le rectangle. Par exemple, il permet de savoir si un point appartient à un rectangle. Écrivons le code qui permet de le savoir:



Le code est assez simple à comprendre:

- on vérifie que l'abscisse du point est plus grande que l'abscisse du point en haut à gauche du rectangle ;
- on vérifie que son abscisse est plus petite que l'abscisse du point en haut à gauche du rectangle plus la largeur du rectangle ;
- on fait la même chose pour l'ordonnée.

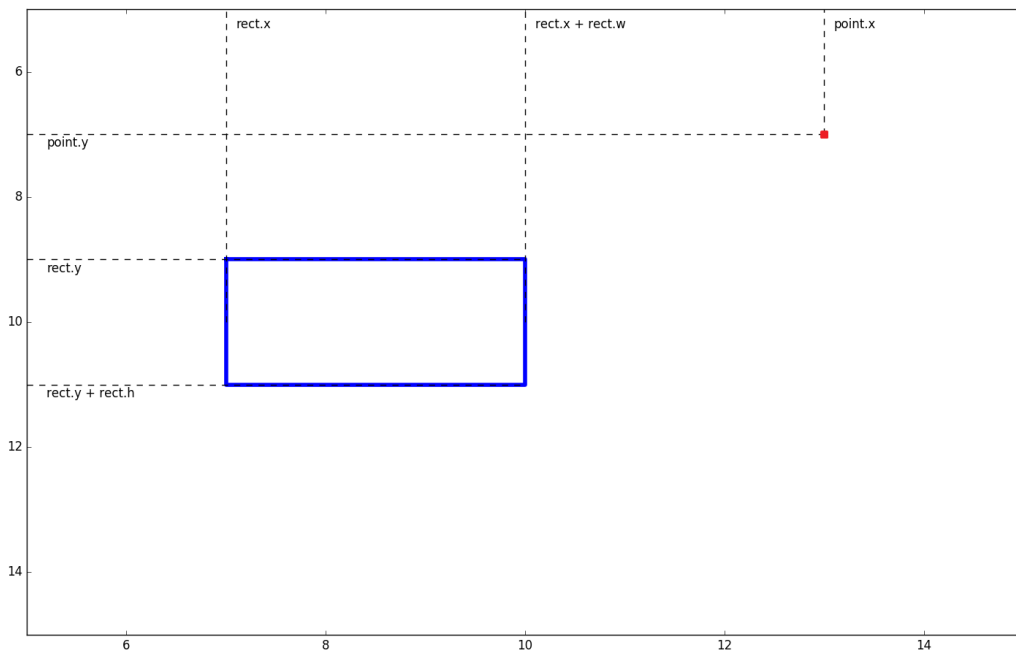


FIGURE 3.2. – Ici, le point rouge n'est pas dans le rectangle.

Notons que ce que l'on vient de coder s'appelle une fonction de collision.

#### 3.2.3. Des fonctions utiles

On vient d'écrire une fonction qui permet de tester si un point est dans un rectangle. On voudrait peut-être aussi écrire une fonction pour savoir si deux rectangles se touchent ou encore pour avoir l'intersection de deux rectangles.



Quoi? Mais, on ne va pas écrire tout ça?

Heureusement, on n'a pas à les écrire. La SDL a pensé à nous et offre plusieurs fonctions de collisions. Voyons en quelques-unes.

Commençons par la fonction qui permet de savoir si deux rectangles se touchent (ce dont on parlait précédemment). Il s'agit de la fonction [SDL\\_HasIntersection](#) . Son prototype:

```
int SDL_HasIntersection(SDL_Rect *a, SDL_Rect *b);
```

Elle prend en paramètre deux pointeurs sur `SDL_Rect`, c'est-à-dire deux pointeurs sur les deux rectangles sur lesquelles doit porter la collision et renvoie un `SDL_bool` qui vaut `SDL_TRUE` s'ils se touchent et `SDL_FALSE` sinon.

En fait, nous pouvons même obtenir le rectangle qui correspond à l'intersection grâce à la fonction [SDL\\_IntersectRect](#) . Son prototype:

```
void SDL_IntersectRect(SDL_Rect *a, SDL_Rect *b, SDL_Rect *c);
```

Elle agit exactement comme la fonction `SDL_HasIntersection`: elle prend en paramètre les pointeurs sur deux rectangles à tester et retourne `SDL_TRUE` s'il y a intersection et `SDL_FALSE` sinon. Cependant, elle prend un troisième argument qui correspond au rectangle résultant de l'intersection.

Pour finir, voyons une dernière fonction. Elle permet de tester si un point est dans un rectangle (en fait, c'est la fonction que l'on a codé tout à l'heure). Il s'agit de la fonction [SDL\\_PointInRect](#) . Son prototype:

```
int SDL_PointInRect(SDL_Point *p, SDL_Rect *r);
```

Nous ne sommes pas surpris en apprenant qu'elle prend en paramètre un pointeur sur `SDL_Point` et un autre sur `SDL_Rect` et renvoie `SDL_TRUE` si le point est dans le rectangle et `SDL_FALSE` sinon.

Et nous renvoie à la [page de la documentation de la SDL à propos des rectangles](#) pour y voir les autres fonctions à ce propos (oui oui, nous renvoyons beaucoup à la documentation).

Comme entraînement pour nous familiariser avec les rectangles et les points, nous pouvons essayer de réécrire toutes ces fonctions.

### 3.2.4. Les couleurs

Le système de couleurs de la SDL est le même que celui utilisé sur les ordinateurs, c'est-à-dire qu'il se base sur 3 nombres entiers dont la valeur est comprise entre 0 et 255. Ces trois valeurs correspondent aux composantes rouge, verte et bleue de la couleur qu'on veut représenter (on parle de système RGB pour *Red-Green-Blue*) et permettent de représenter toutes les autres couleurs. Ainsi, chaque couleur est représentée par:

- sa quantité de rouge;
- sa quantité de vert;
- sa quantité de bleu.

Notons que ce système s'appelle la [synthèse additive](#) ↗ .

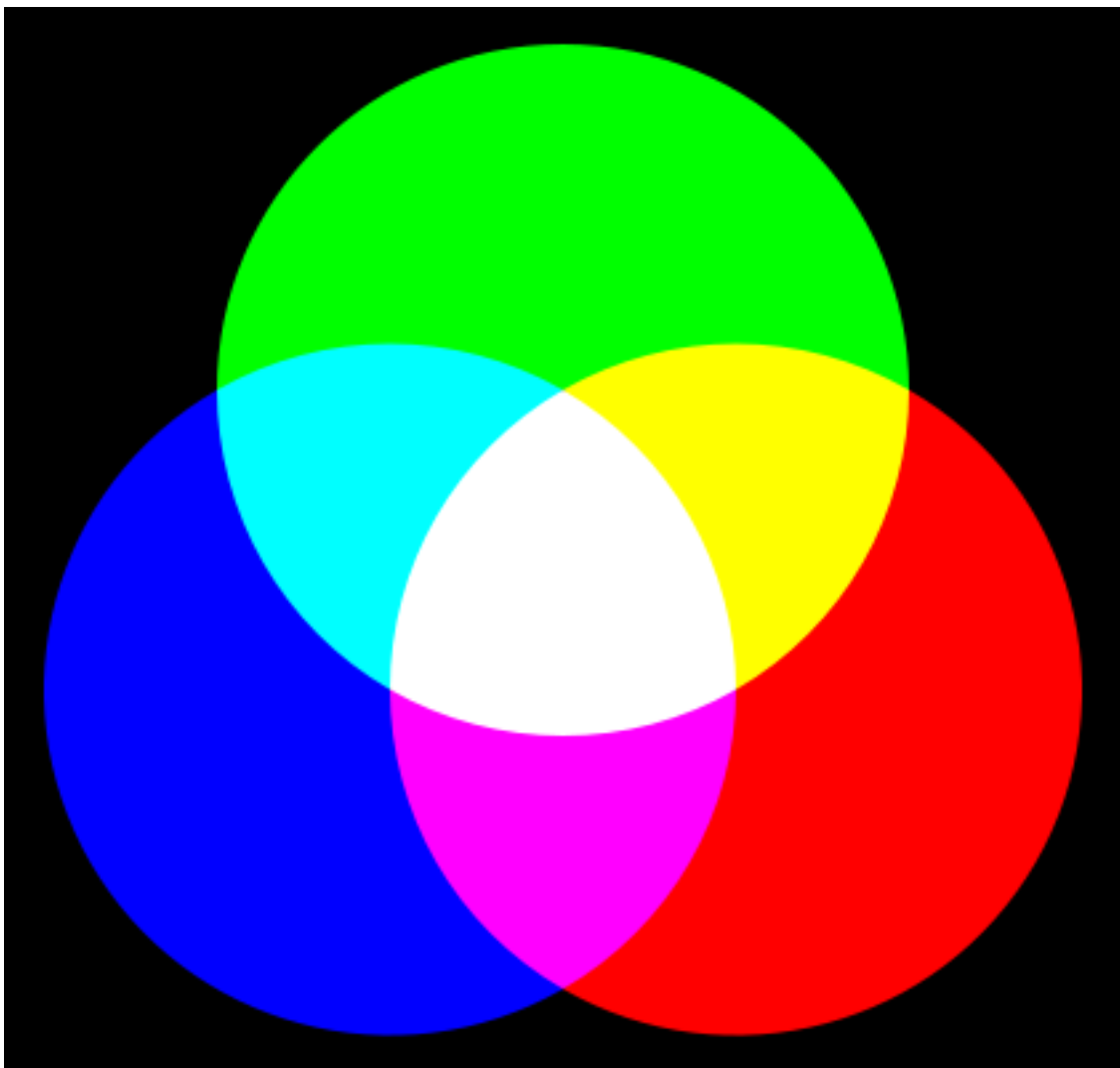


FIGURE 3.3. – La synthèse additive.

On obtient par exemple un jaune particulier avec le triplet `(255, 255, 0)` (le jaune est un mélange de rouge et de vert).

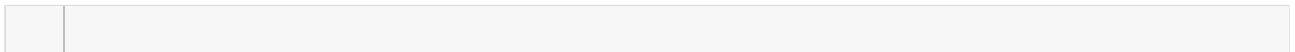
### 3. Dessiner dans la fenêtre

À ces trois valeurs, on ajoute parfois une quatrième qui correspond à la **composante alpha** [↗](#) de la couleur. Cette composante ne change pas la couleur mais permet de gérer la transparence, 0 correspondant à de la transparence totale et 255 à de l'opacité totale.

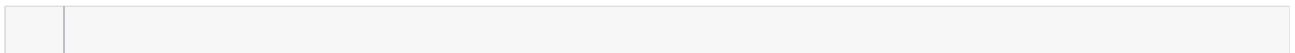
La plupart des fonctions de la SDL nous demanderont les 3 valeurs (ou les 4 pour certaines). De plus, la SDL dispose d'une structure pour représenter une couleur. Il s'agit de la structure **SDL\_Color** [↗](#). Ses champs:

- **r** est un entier entre 0 et 255 et représente la composante rouge de la couleur ;
- **g** est un entier entre 0 et 255 et représente la composante verte de la couleur ;
- **b** est un entier entre 0 et 255 et représente la composante bleue de la couleur ;
- **a** est un entier entre 0 et 255 et représente la composante alpha de la couleur.

Cette structure est intéressante dans le cas où on manipule plusieurs couleurs en même temps. Par exemple, supposons que l'on ait une fonction **colorier** qui prend en paramètre les quatre composantes d'une couleur et colorie la fenêtre. Si on veut la colorier en bleu, on utilisera ce code.



Là ça va, pas de difficulté. Par contre, si on veut colorier en bleu, puis en orange, puis en rose fuchsia, il vaudrait mieux faire ainsi.

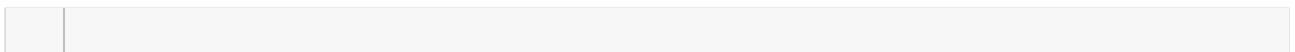


Ou encore mieux : on pourrait faire une fonction qui prend en paramètre un **SDL\_Color** et se charge d'appeler la fonction **colorier** ou bien modifier directement la fonction **colorier** afin qu'elle utilise un **SDL\_Color** comme paramètre.

## 3.3. Des dessins

### 3.3.1. Le principe

Le principe des *renderer* de la SDL est vraiment très simple. Voyons un exemple.

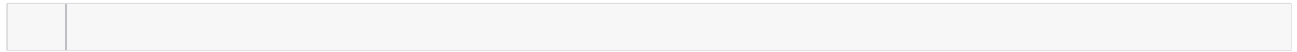


Ce programme change la couleur de la fenêtre en orange avant de se fermer. Expliquons le pas à pas. Bien sûr, nous n'allons pas revenir sur les initialisations.

#### 3.3.1.1. Choisir la couleur de dessin

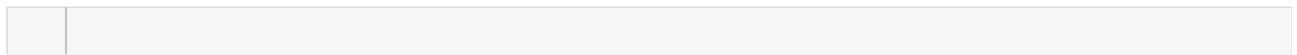
Pour commencer, nous choisissons une «couleur de travail» pour le *renderer*. En fait, il faut voir le *renderer* comme un outil de dessin; nous choisissons quelle couleur utiliser avec cet outil. Pour cela, nous utilisons la fonction **SDL\_SetRendererDrawColor** [↗](#). Son prototype:

### 3. Dessiner dans la fenêtre



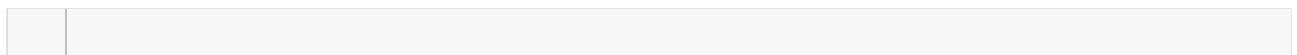
Elle prend en paramètre le *renderer* et les composantes de la couleur en question. Elle renvoie comme d'habitude `0` si tout s'est bien passé et une valeur négative en cas d'erreur. Nous devrions donc savoir l'utiliser. La couleur que l'on utilise pour dessiner sera celle-là jusqu'au prochain changement de couleur.

Dans notre code d'exemple, nous choisissons l'orange comme «couleur de travail» grâce à ces lignes.



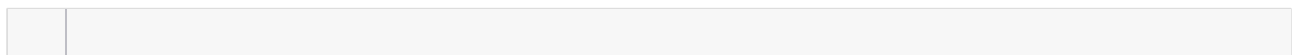
#### 3.3.1.2. Changer la couleur

Ensuite, il nous faut nettoyer le *renderer*, c'est-à-dire l'effacer entièrement en le «peignant» de la couleur souhaitée (celle que l'on utilise actuellement). On le fait à l'aide de la fonction [SDL\\_RenderClear](#) [↗](#). Son prototype:



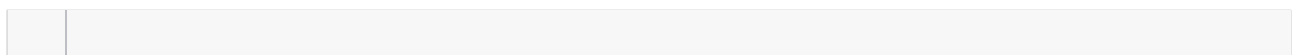
Elle prend en paramètre le *renderer* qui doit être nettoyé et renvoie `0` en cas de succès et une valeur négative sinon.

C'est donc cette partie de notre code d'exemple.



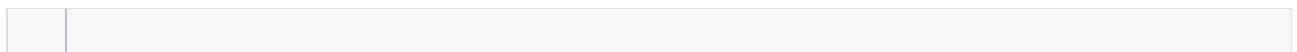
#### 3.3.1.3. Mise à jour de l'affichage

Pourtant, même après avoir fait tout ça, notre fenêtre n'est toujours pas en orange. En effet, on a modifié le *renderer*, mais on n'a pas mis à jour l'écran. La mise à jour de l'écran à partir du *renderer* se fait avec la fonction [SDL\\_RenderPresent](#) [↗](#). Son prototype:



Elle prend en paramètre le *renderer* à mettre à jour et ne retourne rien.

Donc, la mise à jour de l'écran se fait avec cette ligne.

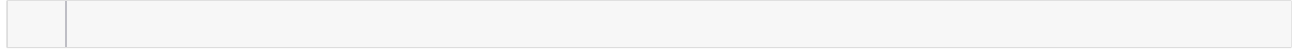


Nous pouvons remarquer que nous avons placé un `SDL_Delay` avant cette ligne. En fait, sans cela, notre fenêtre passe instantanément à l'orange. Le `SDL_Delay` nous permet de ne pas

### 3. Dessiner dans la fenêtre

changer la couleur de la fenêtre tout de suite. C'est grâce à cela que l'on voit le passage à l'orange.

Pour avoir un meilleur code, on pourrait créer une fonction pour changer la couleur de la fenêtre, pour avoir un code de ce genre avec une fonction réutilisable.



*i*

Dans ce dernier code, la réussite de la fonction `setWindowColor` ne nous intéresse pas. Ce sera souvent le cas lorsque nous allons dessiner. L'échec d'une fonction de dessin ne provoquera pas de crash, vérifier qu'elle a réussi n'est donc pas fondamental (à moins bien sûr que notre programme ait absolument besoin de sa réussite).

Finalement, dessiner revient à suivre deux étapes.

1. On dessine à l'aide du *renderer*.
2. On met à jour l'écran avec le *renderer*.

C'est ce que l'on fait tout le long d'un programme. On dessine et on met à jour, on redessine, on remet à jour...

L'étape 2 se fait très simplement à l'aide de la fonction `SDL_RenderPresent`.

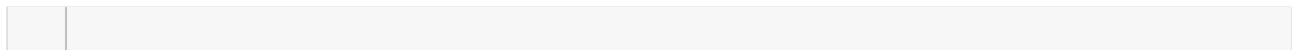
La première étape peut se décomposer en plusieurs étapes (choix d'une couleur de travail, et utilisation de cette couleur pour dessiner). Dans notre exemple, nous avons changé la couleur de l'écran, mais il y a plusieurs manières de dessiner:

- on peut changer la couleur de fond de la fenêtre ;
- on peut afficher un point ;
- on peut afficher une image.
- etc.

Toutes ces opérations se font à l'aide du *renderer* qui est, comme nous l'avons dit, notre outil de dessin.

#### 3.3.2. Dessiner des points et des lignes

Maintenant que nous connaissons la méthode, voyons comment dessiner un point. Pour cela, nous allons choisir notre couleur, puis utiliser la fonction `SDL_RenderDrawPoint` [↗](#). Son prototype:



Elle prend en paramètre le *renderer* sur lequel dessiner et les coordonnées du point à dessiner et retourne `0` en cas de succès et une valeur négative en cas d'erreur.



### 3. Dessiner dans la fenêtre

Pour dessiner un point rouge, puis un point bleu et finalement un point vert, nous pourrions écrire ceci.

Nous obtiendrons ainsi trois points.

Et si nous avons besoin d'afficher plusieurs points d'un coup (donc de la même couleur), la SDL propose la fonction `SDL_RenderDrawPoints` [↗](#). Son prototype:

Elle prend en paramètre le *renderer* sur lequel dessiner, un tableau de `SDL_Point` et le nombre de points à dessiner (donc ce nombre doit être plus petit ou égal à la taille de ce tableau). Elle retourne comme d'habitude une valeur négative en cas d'erreur et `0` en cas de succès.

Utilisons la pour dessiner une ligne sur l'écran.

On a utilisé un tableau de `SDL_Point`. Le champ `y` de tous ces points est égal à `200` et le champ `x` va en croissant, finalement on obtient donc une ligne horizontale d'ordonnée `200`.

?

Et pour dessiner une ligne sur la diagonale de l'écran?

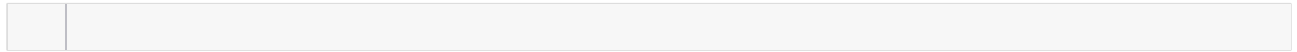
Il existe plusieurs [algorithmes de tracés de segment](#) [↗](#). Nous pourrions les implémenter et créer une fonction qui trace un segment reliant deux points. Mais ce n'est même pas la peine: la SDL possède déjà une fonction pour tracer un segment, la fonction `SDL_RenderDrawLine` [↗](#). Son prototype:

Elle prend en paramètre le *renderer* sur lequel dessiner le segment et les coordonnées des deux points à relier. Elle retourne comme d'habitude `0` en cas de succès et une valeur négative en cas d'erreur.

Pour relier les coins haut gauche et bas droit d'une fenêtre en `640x480`, nous allons donc faire ceci.

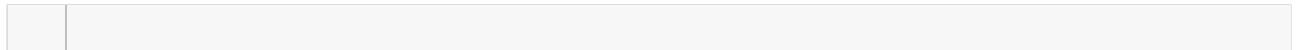
Et là encore, on peut dessiner plusieurs lignes à l'aide de la fonction `SDL_RenderDrawLines` [↗](#). Son prototype:

### 3. Dessiner dans la fenêtre



Elle prend en paramètre le `render`, un tableau de points et la taille de ce tableau. Elle retourne (encore une fois) `0` en cas de succès et une valeur négative en cas d'erreur.

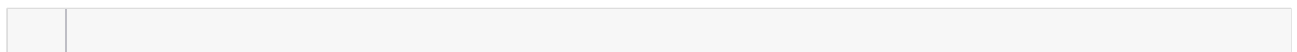
Cette fonction dessine `count - 1` segments: elle relie le premier point au second, le second au troisième, le troisième au quatrième, ..., l'avant-dernier au dernier. Ainsi, pour dessiner un carré de longueur 20 pixels, nous pourrions faire ceci.



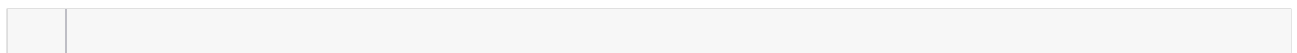
On crée 5 points parce qu'il ne faut pas oublier de relier le dernier point au premier pour « fermer » le carré. Le dernier point est donc le même que le premier.

#### 3.3.3. Dessiner des rectangles

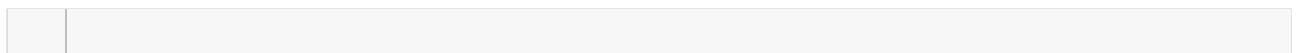
La SDL a aussi des fonctions pour dessiner des rectangles. Voyons pour commencer la fonction `SDL_RenderDrawRect` [↗](#). Son prototype:



Elle prend en paramètre le `render` sur lequel dessiner et un pointeur sur `SDL_Rect` qui représente le rectangle à dessiner et retourne `0` en cas de succès et une valeur négative en cas d'erreur. Ainsi, pour dessiner le carré de l'exemple précédent, nous pouvons faire ceci.

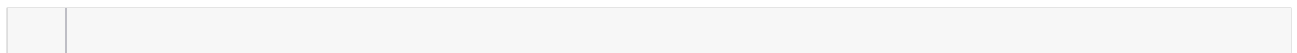


La fonction `SDL_RenderDrawRect` dessine un rectangle (c'est-à-dire les contours du rectangle). On peut aussi vouloir dessiner un rectangle plein. Dans ce cas, il nous faut une fonction qui remplira un rectangle de couleur (« *fill* » en anglais). Dès lors, le nom de la fonction qui fait cette action est toute trouvée. Il s'agit de la fonction `SDL_RenderFillRect` [↗](#). Son prototype:



C'est exactement le même prototype que celui de la fonction `SDL_RenderDrawRect`. Les paramètres et les retours sont exactement les mêmes.

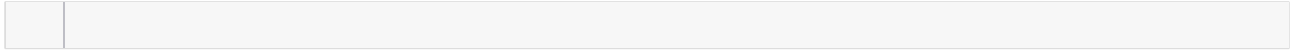
En changeant le `SDL_RenderDrawRect` de l'exemple précédent en `SDL_RenderFillRect`, on obtiendra donc un rectangle plein (avec la couleur que nous aurons choisie bien sûr).



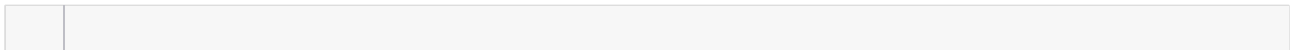
### 3. Dessiner dans la fenêtre

Le second paramètre de [SDL\\_RenderFillRect](#) (et aussi de [SDL\\_RenderDrawRect](#)), peut être `NULL`. Dans ce cas, le rectangle qui sera rempli (dont le contour sera dessiné) est le *renderer* en entier.

Et maintenant, voyons les variantes pour dessiner plusieurs rectangles. Il s'agit (et nous pouvions facilement le deviner) des fonctions [SDL\\_RenderFillRects](#) et [SDL\\_RenderDrawRects](#). Leurs prototypes ne devraient pas non plus nous surprendre:



Et



Comme d'habitude, elles retournent `0` en cas de succès et une valeur négative en cas d'erreur. Elles prennent en paramètre le *renderer* sur lequel dessiner, un tableau de `SDL_Rect` et le nombre de rectangles à dessiner.

Essayons par exemple d'utiliser `SDL_RenderFillRects` pour dessiner un damier. Voici un code possible.

👁 Contenu masqué n°1

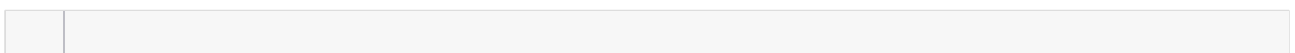


Dans ce chapitre nous n'avons quasiment pas fait de vérifications du retour des fonctions de la SDL. Il ne faut pas oublier de les faire dans notre code. Le plus simple est de créer des fonctions qui se chargent de cela.

## Contenu masqué

### Contenu masqué n°1

Avec une fenêtre `500x500` (donc des cases `50x50`):



Bien sûr, il existe d'autres méthodes pour faire ce travail.

[Retourner au texte.](#)

## 4. Les textures et les images

Dans ce chapitre, nous allons traiter des textures. Maintenant que nous savons comment nous occuper du rendu de notre fenêtre, il nous faut voir comment faire des dessins plus évolués et c'est le but des textures. Pour commencer, nous pouvons voir les textures comme de simples paquets de pixels qu'on va coller sur la fenêtres. Par exemple, nous pourrions charger les pixels d'une image dans une texture et ensuite coller cette texture sur l'écran c'est-à-dire afficher l'image.

### 4.1. Les textures

#### 4.1.1. Généralités sur les textures

Une texture est une structure `SDL_Texture` [↗](#). Nous avons dit qu'on pouvait le voir comme un paquet de pixels (un rectangle plus précisément). Ce paquet de pixels, on pourra l'afficher, le modifier, etc.

##### 4.1.1.1. Créer une texture

Voyons maintenant comment créer une texture. Cette opération se fait à l'aide de la fonction `SDL_CreateTexture` [↗](#). Son prototype:

--	--

Elle prend en paramètre:

- `renderer` est le *renderer* auquel on veut que notre texture soit associée.
- `format` correspond au format de pixel (il en existe plusieurs donnés dans l'énumération `SDL_PixelFormatEnum` [↗](#)). Nous allons généralement choisir la valeur `SDL_PIXELFORMAT_RGBA8888` qui correspond à la représentation que nous connaissons avec 4 chiffres entre 0 et 255.
- `access` correspond aux restrictions d'accès de notre structure. On peut lui donner trois valeurs (voir l'énumération `SDL_TextureAccess` [↗](#)).
- `w` et `h` correspondent à la largeur (*width*) et à la hauteur (*height*) de la texture (qui est, rappelons le, une sorte de rectangle de pixels).

Les trois valeurs possibles pour `access` sont celles-ci.

valeur	Description
--------	-------------

#### 4. Les textures et les images

<code>SDL_TEXTUREACCESS_STATIC</code>	La texture est rarement modifiée
<code>SDL_TEXTUREACCESS_STREAMING</code>	La texture est souvent modifiée
<code>SDL_TEXTUREACCESS_TARGET</code>	La texture peut être utilisée comme cible de rendu (comme un <i>renderer</i> )

Celles que nous allons utiliser ici sont `SDL_TEXTUREACCESS_TARGET` et `SDL_TEXTUREACCESS_STREAMING`.

La fonction `SDL_CreateTexture` renvoie un pointeur sur `SDL_Texture`. Si la création de la texture échoue, ce pointeur vaut `NULL`, sinon il pointe sur la texture créée (il ne faudra donc pas oublier de récupérer et tester la valeur retournée).

##### 4.1.1.2. Détruire une texture

Nous pouvons le deviner. Il faut détruire la texture une fois qu'on a fini de l'utiliser. Cette action se fait avec la fonction `SDL_DestroyTexture` [↗](#) dont le prototype est:

```
SDL_DestroyTexture(SDL_Texture *texture);
```

Comme d'habitude, la fonction de destruction ne renvoie rien et prend en paramètre la texture à détruire. On peut maintenant écrire notre code.

```
SDL_DestroyTexture(texture);
```

Notons que nous pourrions faire une fonction qui se chargerait de toutes les initialisations et des créations nécessaires.

##### 4.1.2. Dessiner sur une texture

Nous avons précédemment dit que si l'on utilisait la valeur `SDL_TEXTUREACCESS_TARGET`, c'était pour pouvoir utiliser notre texture comme un *renderer*. Pour être plus précis, c'est pour pouvoir utiliser notre texture comme **cible** («target») de rendu. En fait, nous allons utiliser les fonctions de dessin vu au chapitre précédent, mais la cible du rendu ne sera plus le *renderer* mais la texture. Pour changer la cible de rendu, nous allons utiliser la fonction `SDL_SetRenderTarget` [↗](#). Son prototype:

```
SDL_SetRenderTarget(SDL_Renderer *renderer, SDL_Texture *texture);
```

Elle prend en paramètre un *renderer* et une texture. Elle retourne `0` en cas de succès et une valeur négative en cas d'erreur (une erreur peut par exemple être que la texture passée en paramètre n'a pas été créée avec le paramètre `SDL_TEXTUREACCESS_TARGET`).

## 4. Les textures et les images

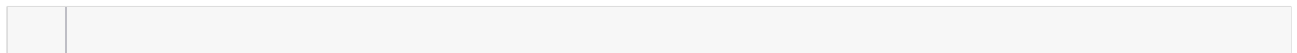
Après avoir appelé cette fonction, toutes les fonctions de dessin modifiant le *renderer* modifieront la texture passée en paramètre. Ainsi, pour dessiner sur notre texture, nous allons:

1. Appeler `SDL_SetRenderTarget` pour que notre texture soit la cible de rendu ;
2. Faire nos dessins ;
3. Remettre le *renderer* en tant que cible de rendu.



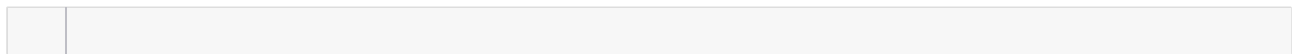
Pour faire la troisième étape, il suffit d'appeler la fonction `SDL_SetRenderTarget` en lui passant `NULL` comme second argument.

Par exemple, dessinons un carré sur une texture.



En essayant ce code, on se rend compte que même après avoir mis à jour le *renderer*, la fenêtre garde la même couleur, cela veut bien dire que le dessin ne s'est pas fait sur elle. Nous verrons bientôt comment afficher une texture et ainsi afficher les modifications effectuées.

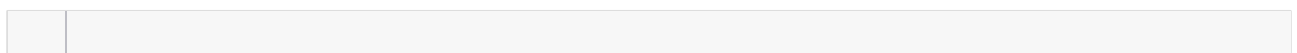
Notons qu'il existe une fonction `SDL_GetRenderTarget` qui permet d'obtenir la cible de rendu (donc de savoir si les dessins se font sur le *renderer* ou sur une texture). Son prototype:



Elle prend en paramètre un *renderer* et retourne un pointeur sur `SDL_Texture` qui correspond à la texture qui est la cible de rendu. Si la cible de rendu est le *renderer* lui-même, la fonction renverra `NULL`. Cette fonction peut être utile dans le cas où on veut, dans une fonction, obtenir l'adresse de la cible du rendu (ce n'est donc pas la peine de la passer en paramètre) ou encore si on a plusieurs textures et qu'on ne sait pas laquelle est la cible du rendu. Pour le moment, elle ne nous sera pas utile.

### 4.1.3. Afficher une texture

Afficher une texture consiste à copier la texture sur le *renderer* puis à mettre à jour le *renderer*. Ainsi, on verra bien la texture à l'écran. La copie de la texture se fait avec la fonction `SDL_RenderCopy` [↗](#) dont le prototype est:



Elle prend en paramètre:

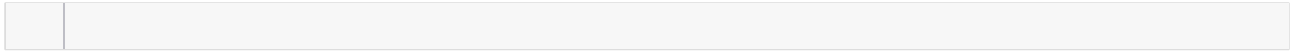
- le *renderer* sur lequel doit être fait la copie;
- la texture à copier;
- un pointeur sur un rectangle qui correspond à la partie de la texture à copier (en passant `NULL`, on copie toute la texture);

#### 4. Les textures et les images

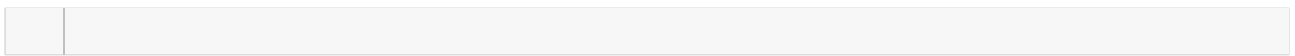
- un pointeur sur un rectangle qui correspond à l'endroit du *renderer* où doit être copié la texture (en passant `NULL`, la texture remplira tout le *renderer*).

La fonction retourne 0 en cas de succès et une valeur négative en cas d'erreur.

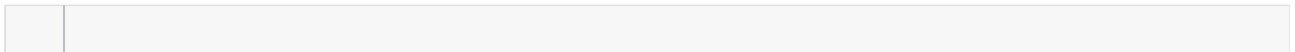
Faisons quelques tests pour voir comment elle fonctionne (on supposera avoir une texture de longueur et de hauteur `50` et un *renderer* associé à une fenêtre de longueur `600` et de hauteur `480`).



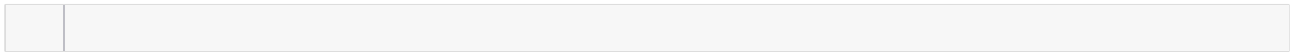
Ce cas est le plus simple. On copie toute la texture dans un rectangle de même dimension qui est placé dans le coin en haut à gauche du *renderer*.



Ici, on ne copie qu'une partie de la texture. On copie cette partie dans un rectangle qui a les mêmes dimensions et qui est placé en haut à gauche.



Ici, nous copions une partie de la texture, mais nous la copions dans un rectangle deux fois plus grand. Résultat: ce qui est affiché est la texture redimensionnée (étirée) pour remplir le rectangle de destination. C'est toujours ce qui se passera, la partie de la texture à afficher sera redimensionnée pour remplir le rectangle de destination. Voyons un dernier exemple.

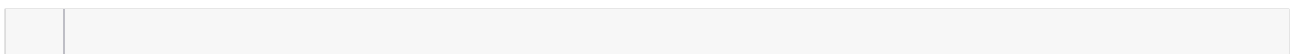


Ici, on copie toute la texture, et on veut qu'elle remplisse tout le *renderer*. Notre texture sera donc redimensionnée pour remplir le *renderer*.



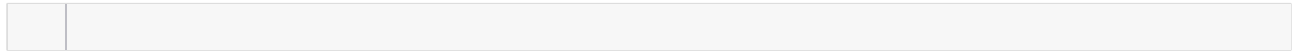
La vraie texture n'est pas redimensionnée. Il s'agit d'un redimensionnement à la volée, au moment de la copie. Notre variable `texture` n'est pas modifiée.

Tout ce que nous venons de dire à propos du redimensionnement implique que pour afficher une texture dans ses vraies dimensions, il faut connaître ses dimensions. Si nous ne les avons pas, il est possible de les récupérer avec la fonction `SDL_QueryTexture` [↗](#) comme nous allons le faire dans le code qui suit.



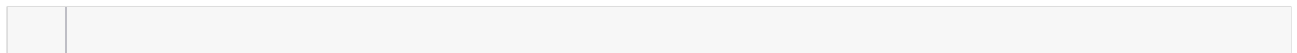
Comme nous aurions pu le deviner grâce au code précédent, le prototype de `SDL_QueryTexture` est:

## 4. Les textures et les images



Elle renvoie 0 en cas de succès et une valeur négative en cas d'erreur et prend en paramètre la texture dont on veut les paramètres et quatre pointeurs qui seront remplis avec, dans l'ordre, le format d'accès de la texture, son type d'accès, sa largeur et sa hauteur.

Dans le code suivant, nous avons passé **format** et **access** à **NULL** car leur valeur ne nous intéresse pas. Celui-ci crée une surface, dessine dessus (fond bleu et rectangle rouge dans le coin en bas à droite) et la colle à l'écran.



## 4.2. Les surfaces

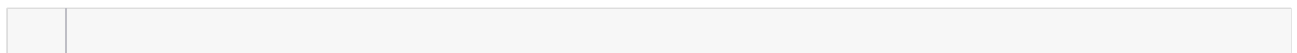
### 4.2.1. Un vestige de la SDL 1

Nous allons maintenant parler des surfaces. Les surfaces sont l'équivalent des textures dans les anciennes versions de la SDL. On représente une surface avec le type `SDL_Surface`. La plus grande différence entre les surfaces et les textures est que les textures sont gérées par le GPU et les surfaces par le CPU. Cela donne quelques avantages aux textures qui sont ainsi affichées plus rapidement que ne l'étaient les surfaces et qui peuvent être redimensionnées à l'affichage (comme nous l'avons vu précédemment).

Les surfaces restent cependant utiles. Par exemple, elles peuvent être modifiées pixel par pixel plus facilement que les textures. Mais là où elles nous seront utiles, c'est pour gérer les images.

#### 4.2.1.1. Créer une surface

La création de surface se fait avec la fonction `SDL_CreateRGBSurface`. Son prototype :



Elle prend en paramètre:

- une liste de drapeaux (ce paramètre doit être à 0);
- la largeur de la surface;
- la hauteur de la surface;
- le nombre de bits par pixels (en général, on travaille avec 32 bits par pixels);
- les quatre derniers paramètres permettent à la SDL de savoir comment extraire la couleur de chaque pixel. Nous allons passer la valeur 0 pour ces quatre paramètres.

La fonction retourne un pointeur sur `SDL_Surface` qui vaudra **NULL** en cas d'échec de la fonction et qui dans le cas contraire pointe sur la surface créée.



## 4. Les textures et les images

### 4.2.1.2. Détruire une surface

Comme les autres ressources, il nous faut détruire chaque surface créée. La libération des données se fait avec la fonction `SDL_FreeSurface` dont le prototype est :

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

Elle ne renvoie rien et prend en paramètre la surface à libérer.

Finalement, avec le code qui suit, on crée une surface de longueur `300` et de hauteur `200`.

```
SDL_Surface * surface = SDL_CreateSurface(300, 200, SDL_PIXELFORMAT_RGBA32);
```

### 4.2.2. Opérations sur les surfaces

#### 4.2.2.1. Colorer une surface

La SDL nous offre des fonctions pour dessiner sur des surfaces, mais il y en a beaucoup moins que celles pour dessiner sur les textures. En fait, il n'y en a qu'une seule. Il s'agit de la fonction `SDL_FillRect` qui nous permet, comme son nom l'indique, de «remplir» un rectangle avec une couleur. Son prototype :

```
int SDL_FillRect(SDL_Surface * surface, SDL_Rect * rect, Uint32 color);
```

Elle prend en paramètre la surface qui doit être remplie, un pointeur sur `SDL_Rect` qui représente la partie de la surface à remplir (en passant `NULL`, on demande à remplir toute la surface) et la couleur voulue. Elle retourne `0` en cas de succès et une valeur négative en cas d'erreur.

Notons que la couleur passée en paramètre doit avoir le même format que la surface. Nous devons donc transformer le triplet `RGB` représentant notre nombre. Pour ce faire, nous allons utiliser la fonction `SDL_MapRGB`. Son prototype :

```
Uint32 SDL_MapRGB(SDL_Surface * surface, Uint8 r, Uint8 g, Uint8 b);
```

Elle prend en paramètre un format de pixel et les trois composantes d'une couleur et retourne un pixel de cette couleur dans le format voulu.

Le format d'une surface est obtenu avec le champ `format` de cette surface.

Par exemple, pour colorer une surface en rouge, nous pouvons utiliser le code suivant.

```
SDL_FillRect(surface, NULL, SDL_MapRGB(surface->format, 255, 0, 0));
```

## 4. Les textures et les images

### 4.2.2.2. Coller une surface sur une autre

Une autre opération possible sur les surfaces est le «*blit*». Cette opération consiste à copier une surface (ou une partie d'une surface) sur une autre surface. On peut donc la voir comme l'équivalent de `SDL_RenderCopy`. Le *blit* s'effectue grâce à la fonction `SDL_BlitSurface` [↗](#). Son prototype :

```
SDL_Surface* SDL_BlitSurface(SDL_Surface* src, const SDL_Rect* srcrect, SDL_Surface* dst, const SDL_Rect* dstrect);
```

Ses différents paramètres sont:

- `src`, la surface source est la surface qui sera copiée;
- `srcrect` est le rectangle source, c'est-à-dire la partie de la surface source qui sera copiée;
- `dst` est la surface de destination, celle sur laquelle sera copiée la source;
- `dstrect` est le rectangle de destination, celui où sera copiée la source.

Notons que cette fonction ne fait pas de redimensionnement à la volée, les champs `h` et `w` de `dstrect` n'ont aucune incidence sur la copie.

### 4.2.2.3. Une fenêtre avec une icône

La SDL nous permet de donner une icône à notre programme grâce à la fonction `SDL_SetWindowIcon` [↗](#). Son prototype est :

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

Elle prend en paramètre la fenêtre dont on veut changer l'icône et une surface qui correspond à l'icône que l'on veut donner à la fenêtre. Cette fonction ne retourne rien. Par exemple, ici, on va faire une icône composée de quatre carrés.

```
SDL_Surface* createIcon(SDL_Surface* src, int w, int h);
```

### 4.2.3. Passer de la texture à la surface

Tout ça, c'est très bien, mais on n'a toujours pas vu comment afficher une surface. En fait, on ne peut pas afficher directement une surface. Il nous faut passer par les textures. Il nous faut donc transformer notre surface en texture, puis afficher la texture obtenue. Le passage de la surface à la texture se fait avec la fonction `SDL_CreateTextureFromSurface` [↗](#) dont le prototype est le suivant.

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

Elle prend en paramètre la surface qui nous permettra de créer la texture et le *renderer* auquel la texture créée doit être associée et retourne cette texture ou `NULL` en cas d'erreur.

## 4. Les textures et les images

Après avoir créé une texture à partir d'une surface, il ne faut pas oublier de libérer cette surface. Pour obtenir une texture à partir d'une surface on peut donc faire ceci.

Bien sûr, nous pouvons libérer la surface à la fin de notre programme, mais autant la libérer dès que l'on n'en a plus besoin.

### 4.3. Les images

#### 4.3.1. Charger une image

Afficher une image est une opération simple en théorie: une image est un paquet de pixels, on lit ce paquet de pixels, on le met dans une texture et on affiche cette texture. Nous pourrions faire une fonction qui prend en paramètre le chemin d'une image et retourne la texture associée à cette image. Heureusement, ce n'est pas la peine, les images sont gérées par la SDL.



Seul le format `BMP` est géré nativement par la SDL.

Pour charger une image, nous allons utiliser la fonction `SDL_LoadBMP` [↗](#). Son prototype:

Elle prend en paramètre le chemin (relatif ou absolu) de l'image à charger et retourne une surface contenant cet objet ou `NULL` en cas d'erreur. Si nous avons parlé des surfaces, c'est parce que nous les utilisons ensuite pour les images.

On peut donc obtenir une surface contenant l'image grâce à `SDL_LoadBMP`. Il nous suffit de créer une texture à partir de cette image, et on peut ensuite l'afficher. Sans oublier de libérer la surface.



Il est conseillé de toujours vérifier le retour de `SDL_LoadBMP`. En effet, si nous ne le faisons pas, nous risquons ensuite de faire des opérations sur un pointeur nul, ce qui causera des problèmes.

On peut alors écrire un code de ce genre pour charger une image dans une texture.

Ici, on aurait aussi pu libérer la surface à la fin en même temps que nos autres libérations, mais autant la libérer directement et ne pas occuper de la mémoire inutilement. Imaginons un code

## 4. Les textures et les images

dans lequel nous chargerions plusieurs dizaines d'images. Il vaudrait mieux que chaque surface soit libérée aussitôt qu'elle n'a plus d'utilité.

### 4.3.2. Dessiner sur l'image

Selon la [documentation](#) [↗](#), le type d'accès de la texture créée est `SDL_TEXTUREACCESS_STATIC`. On ne pourra donc pas modifier cette texture en dessinant dedans à moins de faire en sorte d'obtenir une texture avec le bon type d'accès. Pour cela, il nous suffit de copier notre texture dans une nouvelle texture qui, elle, a le bon type d'accès. Cela va donc se faire de cette manière.

- Charger notre image.
- Créer une texture à partir de notre image.
- Créer une autre texture de même dimension que la surface.
- Placer cette dernière texture en tant que cible de rendu.
- Copier la première texture sur la deuxième (avec `SDL_RenderCopy`).

Bien sûr, il faudra faire les libérations adéquates au bon moment et ne pas oublier de vérifier les valeurs retournées par les fonctions à risque. On se retrouve donc avec ce code.

### 4.3.3. Se faire des fonctions

Ici, notre but sera de pouvoir écrire au minimum un code de ce genre.

Écrivons les fonctions. Commençons par la fonction `init`. Pour que les pointeurs passés en paramètre soient modifiés, il faut lui passer en paramètre des doubles pointeurs. On écrit donc cette fonction.

La fonction `loadImage` doit charger une image et renvoyer la texture correspondante. Elle doit donc charger l'image dans une surface, convertir cette surface en texture et renvoyer cette texture sans oublier de vérifier les retours des fonctions employées.

On aurait pu faire la fonction `loadImage` pour que la texture renvoyée ait un accès `SDL_TEXTUREACCESS_TARGET`. La fonction `setWindowColor` a déjà été écrite dans le [chapitre précédent](#) [↗](#).

On pourrait même rendre ces fonctions plus personnalisables et en faire plus.

#### 4. *Les textures et les images*

La manière de charger des images nous montre bien que les surfaces restent utiles et ne sont pas totalement dépréciées. Pour en savoir plus à leur propos, consultons la [page de la documentation de la SDL à propos des surfaces](#) ↗ .

---

Avec ce chapitre, nous sommes maintenant capable de dessiner sur la fenêtre, de charger des images et de les afficher. Au prochain chapitre, nous verrons une autre manière de modifier les textures et les surfaces en changeant la couleur des pixels que l'on veut.

## 5. Modification pixels par pixels

Après avoir vu le fonctionnement des textures et des surfaces, nous allons voir comment les manipuler pixels par pixels.

### 5.1. Les textures

Nous pouvons également modifier une texture pixels par pixels. En fait, les pixels d'une texture sont stockés dans un tableau. Nous allons récupérer ce tableau, et le modifier.

#### 5.1.1. Le format des pixels

*i*

Cette partie sur le format des pixels est assez technique. Elle traite notamment de représentation des nombres en machine. Il n'est pas nécessaire de la comprendre parfaitement pour poursuivre le tutoriel.

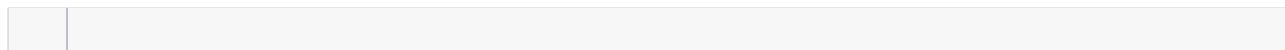
Le problème est que pour modifier un pixel et lui donner une autre couleur, il faut savoir comment il est stocké et comment ses couleurs sont représentées. Heureusement, c'est nous qui choisissons cette représentation lorsque nous créons une texture en spécifiant le format de pixel. La plupart du temps, nous choisissons le format `SDL_PIXELFORMAT_RGBA8888`. Cela veut dire qu'un pixel est représenté avec 8 *bits* pour chacune de ses composantes. Dans l'ordre :

- 8 *bits* pour le rouge;
- 8 *bits* pour le vert;
- 8 *bits* pour le bleu;
- 8 *bits* pour la composante alpha.

Ainsi, `0xFF0000FF` représente du rouge, `0x00FF00FF` représente du vert et `0x0000FFFF` représente du bleu.

Les pixels sont donc codés sur 32 *bits* dans ce format. Nous allons coder ces pixels avec le type `Uint32` (il s'agit d'un type de 32 *bits* permettant de représenter des entiers) et chacune de ses composantes avec le type `Uint8`. Dans ce format, l'octet de poids fort représente le rouge et celui de poids faible représente le canal alpha.

Nous pouvons dès lors créer une fonction qui transformera une couleur donnée par ses quatre composantes en pixel de cette couleur dans le format `SDL_PIXELFORMAT_RGBA8888`.



## 5. Modification pixels par pixels

Bien sûr, si le format de pixel de la texture n'est pas le même, cette fonction devra être modifiée. Nous n'allons pas utiliser cette solution, mais il est intéressant de la voir pour comprendre comment ça se passe.

Dans le chapitre précédent, nous avons vu la fonction `SDL_MapRGB`. Il y a également une fonction `SDL_MapRGBA` à qui l'on fournit les composantes `RGBA` d'une couleur et qui nous renvoie la valeur du pixel associé. Voici son prototype (qui ne devrait pas vous surprendre).

```
uint32_t SDL_MapRGBA(SDL_PixelFormat *format, Uint8 r, Uint8 g, Uint8 b, Uint8 a);
```

C'est le même que celui de la fonction `SDL_MapRGB` avec le paramètre `a` en plus. C'est cette fonction que nous allons utiliser pour obtenir un pixel d'une couleur donnée.

?

Oui, cette fonction fait ce que l'on veut, mais comment savoir quel premier paramètre lui donner? Les surfaces ont un champ `format` qui correspond à ce paramètre, mais que pouvons-nous faire pour les textures?

Cette question est en effet importante. Nous connaissons le format de notre texture sous la forme d'un `Uint32` (par exemple, la constante `SDL_PIXELFORMAT_RGBA8888`). Pour passer de cet entier à une variable du type `SDL_PixelFormat`, nous allons utiliser la fonction `SDL_AllocFormat`.

```
SDL_PixelFormat *SDL_AllocFormat(uint32_t format);
```

Elle prend en paramètre le format d'un pixel sous la forme d'un entier et retourne un pointeur sur `SDL_PixelFormat` (donc un pointeur sur le format du pixel). En cas d'erreur, elle retourne `NULL`.

Après avoir fini d'utiliser notre format, il nous faut le libérer avec la fonction `SDL_FreeFormat` de prototype suivant (elle prend en paramètre le pointeur sur `SDL_PixelFormat` et ne retourne rien).

```
void SDL_FreeFormat(SDL_PixelFormat *format);
```

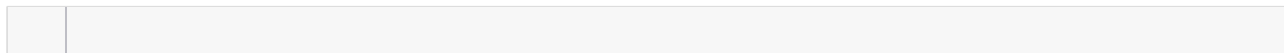
On peut alors faire un code de ce genre.

```
SDL_PixelFormat *format = SDL_AllocFormat(SDL_PIXELFORMAT_RGBA8888);
```

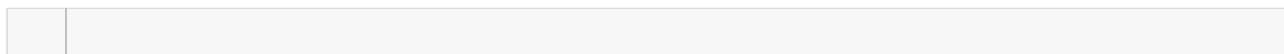
### 5.1.1.1. Obtenir les composantes d'un pixel

Il pourra parfois être utile d'obtenir les composantes d'un pixel donné (donc passer d'un `Uint32` à un `SDL_Color`). Pour faire cette opération, nous utilisons la fonction `SDL_GetRGBA`.

## 5. Modification pixels par pixels



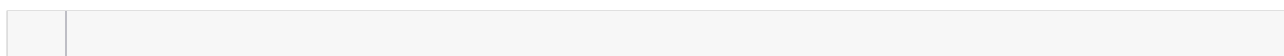
Elle prend en paramètre le pixel, un pointeur sur `SDL_PixelFormat` (comme `SDL_MapRGBA`) et des pointeurs sur quatre `Uint8` qui auront ensuite comme valeur les composantes de la couleur. Pour récupérer les composantes d'un pixel, on utilise alors ce code (après avoir récupéré le pixel et le format).



Notons l'existence de la fonction `SDL_GetRGB` qui permet de ne récupérer que les composantes rouge, bleue et verte. Son prototype ne diffère de celui de `SDL_GetRGBA` que par la disparition du pointeur pour la composante alpha.

### 5.1.2. Modifier les pixels

Pour obtenir le tableau de pixels d'une texture, nous allons utiliser la fonction `SDL_LockTexture`. Elle permet de «bloquer» une partie de la texture en lecture seule. Son prototype :



Elle prend en paramètre :

- une texture;
- un pointeur sur `SDL_Rect` qui représente la partie de la texture que nous voulons bloquer;
- un double pointeur qui pointera sur l'adresse mémoire du tableau de pixels;
- un pointeur sur un entier qui contiendra à l'issue de la fonction la longueur d'une ligne, en *octet*.

La fonction retourne 0 en cas de succès et une valeur négative en cas d'échec. Le paramètre `pitch` peut paraître abstrait. Prenons donc un exemple : si on veut bloquer toute une texture de longueur 200 et que chaque pixel est stocké sur 32 *bits* (donc 4 octets), `pitch` vaudra 800 ( $200 * 4$ ).



La texture que l'on veut bloquer doit avoir un accès `SDL_TEXTUREACCESS_STREAMING`.

Voyons comment les pixels sont arrangés dans le tableau en prenant l'exemple d'une image 3x3 (donc `w = 3` et `h = 3`).

Pixel 1	Pixel 2	Pixel 3	Pixel 4	Pixel 5	Pixel 6	Pixel 7	Pixel 8	Pixel 9
---------	---------	---------	---------	---------	---------	---------	---------	---------

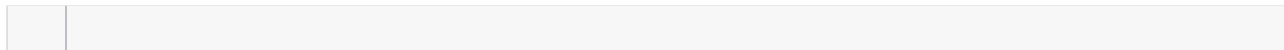


## 5. Modification pixels par pixels

$0*w + 0$	$0*w + 1$	$0*w + 2$	$1*w + 0$	$1*w + 1$	$1*w + 2$	$2*w + 0$	$2*w + 1$	$2*w + 2$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

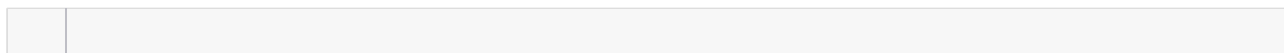
Notons que le *pitch* est alors de 12 (3 pixels par ligne avec chaque pixel faisant 4 octets).

Créons une texture dégradée grâce à cela.

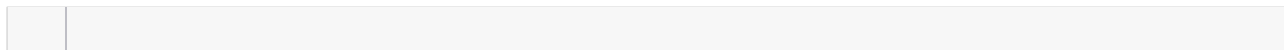


Ici, on bloque notre texture et ensuite on modifie chaque case du tableau de pixels. La case  $i * \text{WIDTH} + j$  correspond au pixel d'abscisse  $j$  et d'ordonnée  $i$ .

Après avoir réalisé nos modifications, il nous faut débloquer la texture. Cette opération se fait avec la fonction `SDL_UnlockTexture` [↗](#) de prototype suivant.



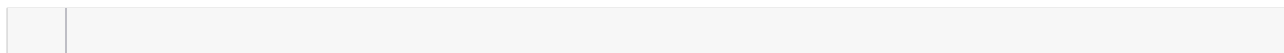
Elle prend comme paramètre la texture à débloquer et ne retourne rien. Il faut débloquer la texture avant de l'afficher ou de faire d'autres opérations. Pour afficher notre dégradé, nous allons donc rajouter ceci à notre code précédent.



Notons que nous aurions pu créer une texture de largeur 1. Le redimensionnement de la texture à l'affichage nous aurait quand même permis d'obtenir le dégradé pour toute la fenêtre. Nous aurions alors fait moins de calcul et économisé de la mémoire.

### 5.1.3. Mettre à jour une texture

Les fonctions `SDL_LockTexture` et `SDL_UnlockTexture` sont parfaites dans le cas où l'on veut modifier quelques pixels de la texture ou si on a besoin de connaître les pixels pour les modifier (par exemple pour un flou gaussien), mais si on veut complètement changer tout ou partie d'une texture et modifier chacun des pixels, on peut faire appel à la fonction `SDL_UpdateTexture` [↗](#) de prototype

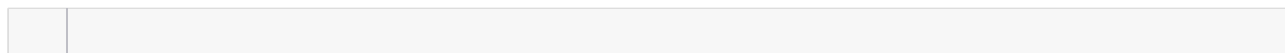


Elle prend en paramètre la texture, un pointeur sur `SDL_Rect` qui correspond à la partie de la texture que l'on veut mettre à jour (comme d'habitude on passe `NULL` pour tout mettre à jour), un tableau de pixels (les pixels doivent bien sûr être dans le bon format) et le «*pitch*» de notre tableau de pixels. Ce paramètre est assez simple à calculer: on veut mettre à jour une texture de longueur  $w$  et chaque pixel est codé sur  $s$  pixels, on a alors  $\text{pitch} = w * s$ .



Si la fonction `SDL_UnlockTexture` ne peut fonctionner qu'avec une structure d'accès `SDL_TEXTUREACCESS_STREAMING`, `SDL_UpdateTexture`, elle, fonctionne avec toutes les textures.

Faisons une fonction qui renvoie une texture avec le même dégradé que dans notre exemple précédent.



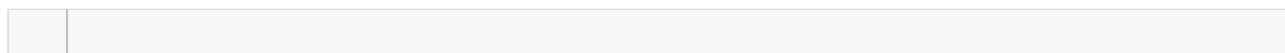
Ici, on s'est débarrassé de la seconde boucle car on a créé une surface de largeur 1. On a d'ailleurs `pitch = sizeof(Uint32) * 1`.

## 5.2. Les surfaces

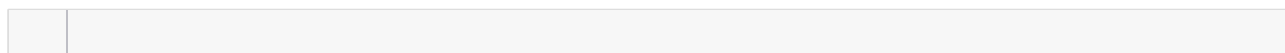
### 5.2.1. Le format des pixels, encore et toujours

Les pixels d'une surface sont stockés de la même manière que ceux d'une texture et pour les modifier, il nous faut donc connaître le format dans lequel ils sont stockés. Nous choisissons ce format lorsque nous créons une surface en spécifiant une valeur pour les quatre derniers paramètres et en spécifiant le nombre de *bits* par pixel. Généralement, le nombre de *bits* par pixel est de 32.

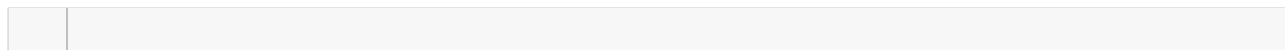
Jusqu'à maintenant, nous ne nous sommes pas occupés des valeurs des masques (nous passons 0 en paramètre), et en fait, nous allons continuer à ne pas nous en occuper. Pour cela, nous allons utiliser la fonction `SDL_CreateRGBSurfaceWithFormat` [↗](#).



Comme `SDL_CreateRGBSurface`, elle prend en paramètre le drapeau (qui doit, rappelons-le, être 0), les dimensions de la surface, le nombre de *bits* par pixels (généralement 32), et le format des pixels (sous la forme d'un `Uint32`). On pourra alors utiliser le format `SDL_PIXELFORMAT_RGBA8888` pour créer notre surface de cette manière.



En fait, une fois qu'on a le format sous la forme d'un `SDL_PixelFormat` (obtenu par exemple à l'aide de la fonction `SDL_AllocFormat`), on peut obtenir les quatre derniers paramètres à passer à `SDL_CreateRGBSurface` pour avoir une surface du même format (ils sont dans les champs `Rmask`, `Gmask` et `Bmask` et `Amask` du format). La ligne de code précédent est alors équivalente à celle-ci.



## 5. Modification pixels par pixels

Nous allons préférer la première version, notamment parce qu'elle ne demande pas d'avoir déjà un pointeur sur le format voulu.

### 5.2.2. Modifier les pixels

Contrairement aux textures, on peut accéder directement au tableau de pixels d'une surface. Il s'agit du champ `pixels` de `SDL_Surface`. Néanmoins, avant de lire et de modifier ce tableau, il nous faut bloquer la surface avec la fonction `SDL_LockSurface` [↗](#) de prototype :

```
int SDL_LockSurface(SDL_Surface *surface);
```

Elle prend en paramètre la surface à bloquer et retourne 0 en cas de succès et une valeur négative en cas d'erreur.

Après avoir fini nos modifications il faut, tout comme pour les textures, débloquer la surface. Nous le devinons facilement, cela se fait avec la fonction `SDL_UnlockSurface` [↗](#) de prototype :

```
void SDL_UnlockSurface(SDL_Surface *surface);
```

On peut alors faire un dégradé sur une surface à l'aide du code suivant.

```
void SDL_FillRect(SDL_Surface *surface, SDL_Rect *rect, Uint32 color);
```

Faisons maintenant une fonction `setPixel` qui prend en paramètre une surface débloquée, les composantes d'une couleur et les coordonnées d'un pixel et donne à ce pixel la couleur associée. On va considérer que les coordonnées passées en paramètre existent.

```
void setPixel(SDL_Surface *surface, int x, int y, Uint32 color);
```

### 5.2.3. Une nouvelle surface ?

En fait, une fois que l'on a un tableau de pixels, on peut s'en servir pour créer une nouvelle surface. On peut par exemple créer une surface de la bonne taille, puis copier le tableau de pixels dans le champ `pixels` de cette surface. Mais, il y a mieux. La SDL offre en effet une fonction qui se charge de créer la surface et de faire cette copie à notre place. Il s'agit de la fonction `SDL_CreateRGBSurfaceWithFormatFrom` [↗](#). Son prototype est le suivant.

```
SDL_Surface *SDL_CreateRGBSurfaceWithFormatFrom(
    Uint32 *pixels, int width, int height, int bpp, Uint32 fmt);
```

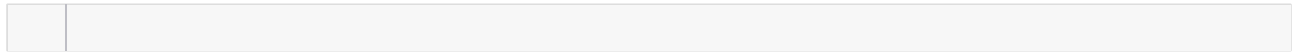
Ses paramètres ne nous sont pas étrangers. On a :

- `pixels` correspond au tableau de pixels que l'on veut assigner à notre surface;
- `width` et `height` sont la largeur et la hauteur de la surface;

## 5. Modification pixels par pixels

- `depth` est le nombre de *bits* par pixels;
- `pitch` est le nombre d'octets pris par une ligne de la surface (il est stocké dans le champ `pitch` de la surface);
- `format` est le format des pixels (sous la forme d'un `Uint32`).

Pour avoir la couleur associée à un pixel, nous allons donc utiliser `SDL_AllocFormat` pour avoir le format sous la forme d'un `SDL_PixelFormat`. Ceci nous permettra ensuite d'utiliser `SDL_MapRGBA`.



Puisque le tableau à allouer peut être de taille vraiment conséquente (par exemple pour une très grande image), la plupart du temps, nous allons allouer le tableau de pixels dynamiquement. Et bien sûr, nous allons éviter d'avoir des nombres en durs dans notre code, et plutôt utiliser des constantes.

Ici, on crée d'abord un tableau de `Uint32`. Ensuite, ce tableau est initialisé dans une boucle `for` avant d'être utilisé dans la fonction `SDL_CreateRGBSurfaceWithFormatFrom`. On a un `pitch` de `255 * sizeof(Uint32)` car la surface que l'on crée a une largeur de 255 pixels et que chaque pixel est codé sur un `Uint32` (puisque son format est `SDL_PIXELFORMAT_RGBA8888`).



Il y a également une fonction `SDL_CreateRGBSurfaceFrom` [↗](#) qui est à `SDL_CreateRGBSurface` ce que `SDL_CreateRGBSurfaceWithFormatFrom` est à `SDL_CreateRGBSurfaceWithFormat`. Nous ne l'utilisons pas ici, car nous préférons pouvoir donner le format à la fonction.

## 5.3. Lier surfaces et textures

### 5.3.1. De la texture à la surface

Nous avons vu dans la partie précédente comment passer de la surface à la texture avec la fonction `SDL_CreateTextureFromSurface`. Mais comment faire l'opération inverse? Il n'existe malheureusement pas de fonction pour créer une surface à partir d'une texture. Si nous voulons le faire, il nous faudra donc gérer tout ça à la main. Pour cela, nous allons:

- bloquer la texture;
- créer un tableau de pixels de même dimension;
- copier les pixels de la texture dans ce tableau;
- créer une surface avec ces pixels.

Il faudra notamment faire attention au format des pixels. En effet, si le format de la texture n'est pas celui que nous voulons pour notre surface, nous ne devons pas juste copier les pixels de la texture dans le tableau, mais nous devons aussi changer leur format. Pour cela, nous

## 5. Modification pixels par pixels

pourrions utiliser `SDL_GetRGBA` pour récupérer les composantes d'un pixel de la texture, puis utiliser `SDL_MapRGBA` pour obtenir un pixel de cette couleur dans le format voulu.

Mais si nous créons une surface de même format que la texture, cela signifie que le tableau de pixels de la texture peut être utilisé directement pour créer la surface, le tableau de pixels d'une surface et celui d'une texture sont équivalents. On a alors un code plus général en récupérant le format de la texture (grâce à `SDL_QueryTexture`), afin de pouvoir utiliser la fonction avec n'importe quelle texture. On écrit alors ce code.

### 5.3.2. Du renderer à la surface ?

Nous venons de voir comment passer d'une texture à une surface. Mais comment faire pour passer de notre cible de rendu à une surface? En gros, comment accéder à ses pixels? Cette opération est possible grâce à la fonction `SDL_RenderReadPixels` [↗](#).

Elle prend en paramètre le *renderer* dont on veut les pixels, un `SDL_Rect` qui correspond à la partie du *renderer* dont on veut récupérer les pixels (`NULL` si on veut la totalité), le format dans lequel on veut que ces pixels nous soient remis (pour ne pas changer, nous utiliserons `SDL_PIXELFORMAT_RGBA8888`), un pointeur qui pointera sur le tableau de pixels après l'exécution de la fonction et le *pitch* de notre tableau (on le détermine comme d'habitude avec la taille d'un pixel et la largeur du tableau).

La fonction renvoie 0 en cas de succès et une valeur négative en cas d'erreur.

Ainsi, si notre fenêtre a une largeur `WIDTH` et une hauteur `HEIGHT`, nous pouvons récupérer ses pixels avec le code suivant (on suppose que le *renderer* est rattaché à la fenêtre).



La fonction `SDL_RenderReadPixels` écrit dans la zone mémoire qu'on lui donne avec l'argument `pixels`. Il faut donc que le pointeur qu'on lui passe en paramètre pointe sur une zone mémoire valide et l'espace réservé doit être de taille suffisante (ici, on l'alloue avec `malloc`).

Nous pouvons alors faire une capture d'écran grâce à ça en créant une surface avec ces pixels et en la sauvegardant avec la fonction `SDL_SaveBMP`.

Notons de plus que `SDL_RenderReadPixels` permet de lire les pixels de la **cible de rendu**. Cela signifie que si la cible de rendu est la cible par défaut (la fenêtre), nous obtiendrons les pixels de la fenêtre. Mais si la cible de rendu est une texture, ce sont ses pixels que nous

## 5. Modification pixels par pixels

obtiendrons. Nous avons alors un moyen d'obtenir les pixels d'une texture dont l'accès est `SDL_TEXTUREACCESS_TARGET`.

### 5.3.3. Quelles opérations privilégier ?

Bon, après avoir vu tout cela, il nous faut nous rendre compte de quelque chose: la plupart de ces opérations sont coûteuses et lentes; enfin, relativement lente. Il ne faut donc pas en abuser. La documentation nous prévient d'ailleurs de cela. Par exemple, sur la page de `SDL_RenderReadPixel`, nous pouvons lire cet avertissement.

WARNING: This is a very slow operation, and should not be used frequently.

[Documentation de la SDL](#) ↗

Elle ne doivent pas être utilisées fréquemment.

Encore une fois, c'est en lisant la documentation que nous pouvons accéder à ces informations. Par exemple, regardons la partie remarque de la documentation de `SDL_UpdateTexture`.

This is a fairly slow function, intended for use with static textures that do not change often. If the texture is intended to be updated often, it is preferred to create the texture as streaming and use the locking functions referenced below. While this function will work with streaming textures, for optimization reasons you may not get the pixels back if you lock the texture afterward.

[Documentation de la SDL](#) ↗

On apprend ici que `SDL_UpdateTexture` est une fonction lente, prévue pour être utilisée sur des textures statiques (donc d'accès `SDL_TEXTUREACCESS_STATIC`). La documentation conseille de préférer l'utilisation de `SDL_LockTexture` si le but est de modifier une texture d'accès `SDL_TEXTUREACCESS_STREAMING`.

En plus de tout cela, il faut savoir que les textures ne sont pas faites pour une modification pixels par pixels. Elles ont l'avantage d'être affichables rapidement, de pouvoir être redimensionnée à la volée et copiée rapidement, mais elles ne sont pas prévues pour de la modification pixels par pixels. Les surfaces sont plus adaptées à cela. Ainsi, si nous voulons modifier une image sans l'afficher (nous le feront dans le chapitre suivant), nous préférons utiliser une surface. Et surtout, nous éviterons au maximum d'avoir à modifier les pixels d'une texture.



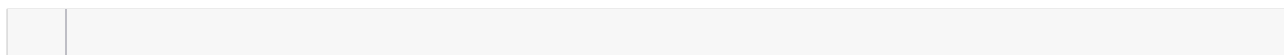
Pour changer facilement les pixels et travailler avec nos surfaces et nos textures, il nous faudra veiller à ce qu'elles aient toutes le même format.

Pensons notamment aux surfaces obtenues avec `SDL_LoadBMP`. Nous ne savons pas quelle est leur format de pixel. Mais pas d'inquiétude, la SDL nous fournit la fonction `SDL_ConvertSurfaceFormat` ↗ qui convertit une surface dans un format choisi.

## 5. Modification pixels par pixels

Elle prend en paramètre la surface à convertir, le format dans lequel doivent être convertis les pixels et des drapeaux (nous devons passer 0 pour cet argument).

Grâce à cette fonction, on peut faire en sorte que nos surfaces chargées avec `SDL_LoadBMP` aient bien le format voulu. On peut même faire une fonction qui nous renvoie la surface dans le bon format.



Bien sûr, cela n'est utile que dans le cas où nous allons effectivement lire ou modifier les pixels de nos surfaces. Si nous n'accédons jamais aux pixels, la manière dont ils sont stockés nous importe peu.

---

Ce chapitre est maintenant fini. Dans le chapitre suivant, nous allons nous attaquer à la transparence, que nous n'avons jamais utilisée jusqu'à maintenant.

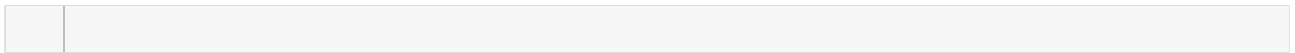
## 6. La transparence

À ce niveau du tutoriel, nous savons jouer avec la fenêtre, afficher des images et des textures diverses, etc. Il nous manque cependant quelque chose avant de pouvoir dire qu'on a à peu près fait le tour de tout ce qui concerne l'affichage. Il nous faut savoir jouer avec la transparence.

### 6.1. La transparence alpha

#### 6.1.1. Un problème embêtant

Nous avons vu que nous pouvions décomposer une couleur en quatre composantes, sa composante rouge, sa composante verte, sa composante bleue et sa composante alpha. La composante alpha correspond au niveau de transparence de l'image. Pour le moment, nous ne l'avons jamais vraiment essayé. Remplissons le *render* avec une couleur transparente.



Notre fenêtre est rouge et ce rouge n'est pas du tout transparent. Même en mettant la composante alpha à zéro, notre fenêtre est toujours rouge.

Pourtant, la composante alpha correspond bien au niveau de transparence. Pourquoi notre code ne fonctionne-t-il pas? En fait, cela est dû au fait que la transparence n'est tout simplement pas activée. Nous pourrions penser qu'il n'y a pas besoin d'activation et qu'il suffit de donner la composante alpha voulue, mais ce n'est pas aussi simple. Le fait est qu'il y a plusieurs types de transparence et qu'il faut indiquer à la SDL lequel nous voulons utiliser.

#### 6.1.2. Les modes de fusion

Voyons donc quels sont les différents types de transparence. Mais avant ça, il est nécessaire de nous poser une question.



Comment fait-on pour rendre un pixel transparent?

Car, même si nous disons qu'il y a plusieurs types de transparence, il faut encore savoir ce que signifie rendre un pixel transparent.



## 6. La transparence

En fait, la transparence d'un pixel n'est pas réelle et n'existe qu'à l'affichage. Un pixel n'est jamais transparent. L'ordinateur fait juste en sorte de nous le faire apparaître transparent. Pour cela, il **fusionne** la couleur du pixel à afficher avec la couleur du pixel qui est en dessous. La composante alpha d'un pixel indique juste à quel point il faut prendre en compte la couleur du pixel pour avoir la couleur à afficher. Ainsi, une composante **alpha** nulle signifie qu'il ne faut pas prendre en compte la couleur du pixel et donc la couleur affichée sera celle du pixel du dessous.



Dans la suite, nous nommerons «pixel source» ou «source» le pixel que l'on veut afficher, «pixel destination» ou «destination» le pixel qui est déjà à l'écran et «pixel résultant» ou «résultat», le pixel résultant de la fusion de la source et de la destination.

Cependant, on peut fusionner les pixels source et destination de plusieurs manières. Ces différentes méthodes s'appellent des **modes de fusion** ou *blend mode* en anglais. On pourrait imaginer plusieurs modes de fusion plus ou moins fantaisistes. Le mode le plus simple étant celui ou

$$= .$$

Ce mode correspond en fait à une absence totale de transparence (c'est ce que nous avons obtenu précédemment avec notre exemple de fenêtre rouge). Mais ce n'est **pas** un mode de fusion, puisqu'il n'y a justement **pas** de fusion.

En gros, les modes de fusion consistent à appliquer une fonction  $f$  telle que  $= f(,)$ .

### 6.1.3. Les modes de fusion de la SDL

Maintenant que nous voyons un peu mieux en quoi consiste la transparence, nous pouvons nous renseigner sur les différents modes de fusion de la SDL. Ces différents modes peuvent être trouvés dans l'énumération `SDL_BlendMode` [↗](#). En regardant les différentes valeurs de cette énumération, on voit que la SDL dispose de quatre modes de fusion (trois en fait puisque l'une des valeurs possibles pour l'énumération correspond à l'absence de transparence).

#### 6.1.3.1. Mode alpha

Le premier mode possible est celui qui correspond le plus à l'idée que l'on se fait d'une fusion puisqu'il s'agit en fait d'une moyenne pondérée de la source et de la destination, le poids étant déterminé grâce à la valeur de la composante alpha. Ce mode est appelé *alpha blending*. Il est associé à la valeur `SDL_BLENDMODE_BLEND` et au calcul

$$= \frac{\times}{255} + \frac{(255-\times)}{255}.$$

Si la composante alpha est nulle, il n'y a pas de transparence, si elle vaut 255, seule la couleur de la destination est prise en compte, si elle vaut 128, les deux couleurs sont parfaitement mélangées.

## 6. La transparence

### 6.1.3.2. Mode d'addition

Le second mode est appelé (à raison) *additive blending* puisqu'il consiste à ajouter la source à la destination. Il permet ainsi de rendre les couleurs plus vives. Il est associé à la valeur `SDL_BLENDMODE_ADD` et au calcul

$$= \frac{\times}{255} + .$$

Nous remarquons que plus la valeur de la composante alpha est élevée, plus la valeur ajoutée sera grande c'est-à-dire qu'encore une fois, plus la valeur de la composante alpha est grande, plus la couleur du pixel source sera prise en compte. Notons également que si la valeur du pixel résultant est supérieure à 255, la SDL la ramène à 255. En fait, on a plutôt

$$= \min \left( 255, \frac{\times}{255} + \right).$$

### 6.1.3.3. Mode de modulation

Le dernier mode, appelé *color modulate*, n'utilise même pas la valeur de la composante alpha du pixel source. Il est associé à la valeur `SDL_BLENDMODE_MOD` et au calcul

$$= \frac{\times}{255}.$$

Il multiplie la source par le résultat, ce qui explique qu'on l'appelle parfois «mode de multiplication».

---

Chaque transformation est appliquée à chacune des composantes rouge, bleue et verte du pixel. La composante alpha, elle, subit ces transformations.

- Dans le cas du mode alpha, on a  $() = () + () \times \frac{255-()}{255}$ .
- Dans les cas du mode d'addition et du mode de modulation, on a  $() = ()$ .

On utilise généralement le mode alpha qui correspond à l'idée qu'on se fait de la transparence.

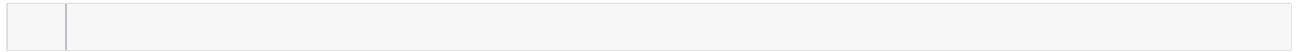
## 6.2. Gérer la transparence alpha

Il ne nous reste plus qu'à voir comment utiliser ces différents modes.

### 6.2.1. Avoir un renderer transparent

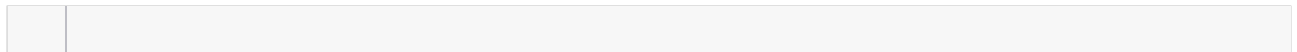
Pour appliquer un mode à un *renderer*, il nous faut connaître une seule fonction. Il s'agit de la fonction `SDL_SetRenderDrawBlendMode` [↗](#) dont le prototype est le suivant.

## 6. La transparence



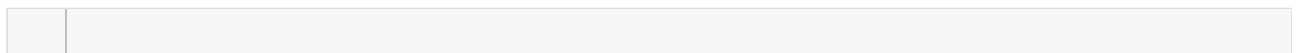
Elle permet d'indiquer à la SDL le mode de fusion que l'on veut donner à notre *renderer*. Pour cela, on lui envoie le *renderer* et le mode de fusion voulu. Elle retourne 0 en cas de succès et une valeur négative en cas d'erreur.

Reprenons le code du début de ce chapitre et modifions-le pour rendre notre fond transparent.



Ça ne fonctionne toujours pas!

C'est normal. Nous avons changé le mode de fusion de notre *renderer* **après** avoir dessiné sur l'écran. Il nous faut donc utiliser `SDL_SetRenderDrawBlendMode` avant d'utiliser `SDL_RenderFillRect`.

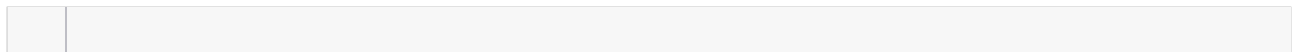


Cette fois, c'est bon. Le fond de notre fenêtre est devenu plus sombre (parce que le fond de notre fenêtre est noir). On a donc bien fusionné la couleur de la source et la couleur de la destination.

Notons que cette fonction active la transparence alpha pour les fonctions de dessin (lignes, rectangles, ect.). La fonction `SDL_RenderClear` n'est pas une fonction de dessin, mais de nettoyage. Le fond ne sera donc pas transparent s'il est fait avec `SDL_RenderClear`.

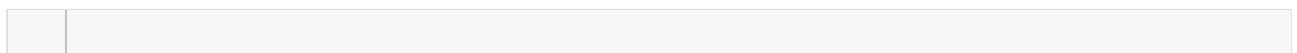
### 6.2.2. Avoir une texture transparente

Maintenant, voyons comment avoir une texture transparente. En effet, mettre la texture en tant que surface de rendu (avec `SDL_SetRenderTarget`) puis utiliser `SDL_SetRenderDrawBlendMode` ne fonctionne pas. Il nous faut utiliser la fonction `SDL_SetTextureBlendMode` [🔗](#) pour appliquer un mode à notre texture. Voici son prototype.



Elle prend en paramètre la texture pour laquelle on veut activer la transparence et le mode voulu. Elle retourne 0 en cas de succès et une valeur négative en cas d'erreur. Notons de plus que si le mode demandé n'est pas supporté par la texture, la fonction renverra -1 et activera le mode de fusion le plus proche de celui demandé (ce cas arrive rarement).

On peut alors afficher une texture transparente.



### 6.2.3. Des collisions entre les textures et le renderer

Nous savons maintenant appliquer de la transparence au *renderer* et aux textures. Mais que se passe-t-il lorsque l'on affiche une texture transparente sur un *renderer* transparent? Comment les textures interagissent-elles avec le *renderer*?

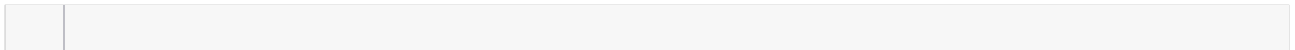


Le mode de transparence de la zone de rendu doit être `SDL_BLENDMODE_NONE` lorsqu'on travaille sur des textures, sans quoi nous aurons des problèmes d'affichage. Après avoir utilisé `SDL_SetRenderTarget` pour choisir la texture en tant que zone de rendu, nous changeons donc, si besoin est, le mode de transparence de la zone de rendu en `SDL_BLENDMODE_NONE`.

On peut observer ces différents cas de figures.

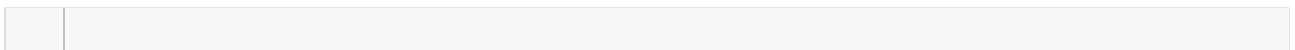
#### 6.2.3.1. Renderer opaque, texture opaque

C'est le cas le plus simple et le plus classique. La texture est opaque et le *renderer* est opaque, donc on ne verra pas le *renderer* à travers la texture. On dessine un rectangle bleu sur un fond rouge, le rectangle bleu cache une partie du fond rouge.



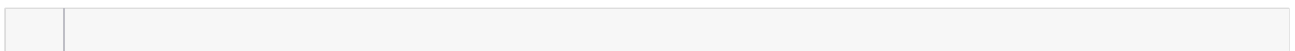
#### 6.2.3.2. Renderer transparent, texture opaque

Le résultat de cette expérience n'est pas trop surprenant. Le *renderer* est transparent, la texture est opaque, donc on voit la texture (le fait que le fond soit transparent ne change rien à cela). On dessine un rectangle bleu sur un fond transparent, donc on voit un rectangle bleu.



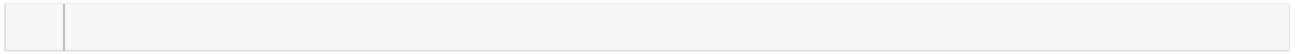
#### 6.2.3.3. Renderer opaque, texture transparente

C'est encore une fois un cas simple et plutôt classique. La texture est transparente et le *renderer* est opaque, donc on verra le *renderer* à travers la texture (plus ou moins suivant la valeur de la composante alpha de la texture). On dessine un rectangle bleu, mais transparent, sur un fond rouge, donc on voit le fond rouge à travers le rectangle.



### 6.2.3.4. Renderer transparent, texture transparente

Ce cas est le seul un peu difficile à appréhender. Le calcul se fait normalement suivant le mode de transparence choisi. Si la transparence du *renderer* et celle de la texture sont totales (la composante **alpha** est nulle), aucune couleur n'apparaît puisque toutes les couleurs sont transparentes, ce qui peut être un peu déconcertant. On dessine un rectangle transparent sur un fond transparent, donc on obtient quelque chose de transparent.

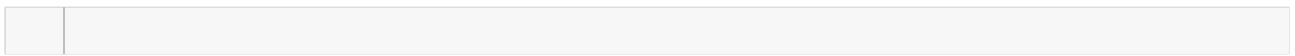


Cependant, nous n'aurons que rarement à nous soucier de cas comme ceux-ci. En général, on veut un fond opaque sur lequel on dessine des choses éventuellement transparentes.

## 6.3. Avec les surfaces

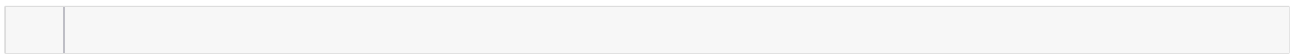
### 6.3.1. La transparence alpha

Il nous est également possible d'appliquer de la transparence alpha sur une surface. Pour cela, nous devons utiliser l'équivalent de la fonction `SDL_SetTextureBlendMode` pour les surfaces. Elle s'appelle, de façon non originale, `SDL_SetSurfaceBlendMode` [↗](#).



Son prototype ne nous surprend guère puisqu'il s'agit de celui de `SDL_SetTextureBlendMode`, la seule différence étant qu'il faut, bien entendu, passer une surface et non une texture en premier argument. Elle s'utilise donc de la même manière et renvoie 0 en cas de succès et une valeur négative en cas d'échec.

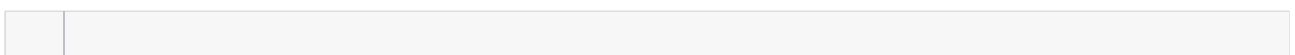
Notons également l'existence de la fonction `SDL_GetSurfaceBlendMode` [↗](#) qui, comme son nom l'indique, permet d'obtenir le mode de fusion d'une surface.



Elle prend en paramètre une surface et un pointeur sur l'énumération `SDL_BlendMode`. À la fin de la fonction, la valeur pointée par `blendmode` vaudra le mode de fusion de la surface passée en paramètre. La fonction renvoie 0 si elle réussit et une valeur négative sinon.

Nous aurons rarement (voire jamais) à l'utiliser, mais elle a le mérite d'exister.

Nous ne l'avons pas vu, mais elle existe également pour les textures sous le nom de `SDL_GetTextureBlendMode` [↗](#).



Là encore, nous n'aurons pas souvent à l'utiliser.

### 6.3.2. Rendre une couleur transparente

Il est également possible de ne rendre qu'une couleur transparente. Imaginons par exemple que l'on veuille afficher une image sur notre fenêtre. Notre image a malheureusement une couleur de fond, et ce fond ne permet pas une bonne intégration de notre image sur la fenêtre. Par exemple, on veut afficher l'image qui suit sur une fenêtre blanche et on veut faire disparaître son fond noir.

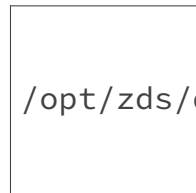
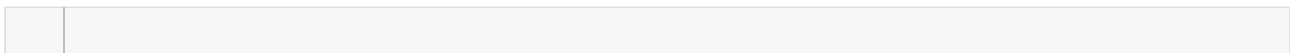


FIGURE 6.1. – Exemple

Il est possible de supprimer la couleur noire de l'image grâce à la fonction [SDL\\_SetColorKey](#) [↗](#).

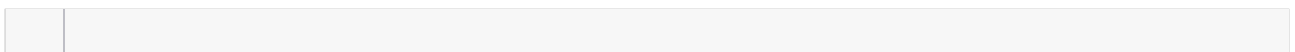


Elle prend en paramètre une surface, un drapeau (`SDL_TRUE` pour activer la transparence, `SDL_FALSE` pour la désactiver) et un `Uint32` qui correspond à la couleur à rendre transparente. Tous les pixels de cette couleur seront alors transparents.



Cela signifie que si, dans notre exemple, d'autres pixels que le fond avaient la couleur noire, ils seraient également transparent. Il faut faire attention à cela.

Pour rendre une couleur transparente, voici finalement le code que nous pouvons utiliser.



En gros, rien de bien compliqué à utiliser.

---

Et c'est un nouveau chapitre qui se termine ici. Nous pouvons (enfin) manipuler la transparence. Il nous faudra néanmoins faire attention à ceci.



La transparence est gérée par notre programme, mais pas par les images au format `BMP`. Ainsi, si nous enregistrons une surface (avec `SDL_SaveBMP`), la transparence ne sera pas conservée.

## 7. TP - Effets sur des images

Nous allons maintenant faire un petit TP qui consiste à modifier des images pour augmenter leur luminosité ou encore les flouter.

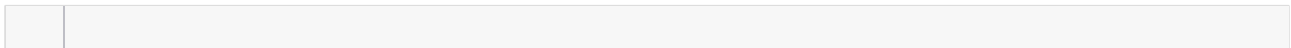
### 7.1. Noirs et négatifs

#### 7.1.1. Principe

Bon, avant de commencer à modifier des images, il faut que nous sachions bien de quoi nous allons parler dans ce chapitre et que nous voyions quelles méthodes nous allons utiliser. Notre but ici est d'effectuer une transformation sur une image telle qu'augmenter sa luminosité. Pour cela, nous allons appliquer une fonction à chacun des pixels de notre image. Par exemple, pour obtenir la même image exactement, on applique à chaque pixel la fonction identité, c'est-à-dire la fonction qui à un élément associe cet élément.

Il nous faudra alors trouver des fonctions pour faire ce que l'on veut. Grâce à elle, nous pourrons créer une fonction qui renvoie une surface à laquelle on aura appliqué une transformation avec cette fonction.

Nous aurons alors un code comme ceci.



Dans l'ordre on bloque la surface pour accéder à ses pixels, on crée un tableau de pixels de la bonne taille, on le modifie, puis on renvoie la surface créée à partir de ce tableau. Le champ `format` d'une variable de type `SDL_Format` correspond à une valeur de l'énumération `SDL_PixelFormatEnum`. Ainsi, si notre surface de départ a le format `SDL_PIXELFORMAT_RGBA8888`, on retrouvera cette valeur dans `s->format->format`. On l'utilise alors pour créer notre nouvelle surface. Bien sûr, on pourrait également décider de créer la nouvelle surface dans un autre format.

Ici, nous renvoyons une nouvelle surface, mais nous aurions également pu modifier directement la surface passée en paramètre.

Dans les codes qui vont suivre, nous n'allons pas réécrire toute la fonction, mais seulement le contenu de la boucle `for`.



Pour utiliser ces fonctions sur une surface créée à partir d'une image, il nous faudra au préalable utiliser la fonction `SDL_ConvertSurface` pour que le tableau de pixels obtenu



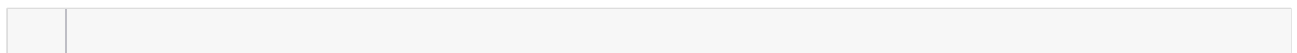
soit bien au format `SDL_PIXELFORMAT_RGBA8888` (une surface chargée depuis une image au format *Bitmap* a généralement le format `SDL_PIXELFORMAT_RGB888`). Sans cela, nous risquons des erreurs de segmentation.

En fait, il nous faudrait juste nous assurer que la surface est dans un format où les pixels sont codés sur quatre octets (quand ils sont codés sur moins d'octet, nous faisons des lectures en dehors du tableau de pixels). Convertir la surface dans un format qui convient est une manière simple de s'assurer de cela.

### 7.1.2. Niveaux de gris

Nous allons commencer par faire une fonction pour transformer une image couleur en niveau de gris. La question qui vient naturellement est: comment obtenir le niveau de gris d'une couleur? Déjà, il nous faut comprendre qu'une couleur est grise si ses trois composantes ont la même valeur. Ainsi, un pixel est gris s'il s'agit du pixel `(i, i, i)`. Plus `i` est grand, plus on va vers le blanc. `(255, 255, 255)` correspond à du blanc et `(0, 0, 0)` correspond à du noir.

Pour avoir le niveau de gris d'une couleur, nous allons faire la moyenne de ses trois composantes. Nous pouvons alors faire une fonction qui prend en paramètre une surface et renvoie une nouvelle surface qui correspond à la surface passée en paramètre en niveau de gris.



Notons qu'ici, nos pixels gris sont codés sur 32 *bits*. Pourtant, on aurait pu ne les stocker que sur 8 pixels, puisque les composantes **R**, **G** et **B** sont les mêmes. Si nous enregistrons l'image obtenue avec cette fonction, nous obtiendrons une image dont le poids en mémoire sera le même que l'image originale, alors que si nous utilisons *Paint* par exemple pour convertir notre image en niveau de gris, son poids aura diminué.



Nous utilisons `SDL_MapRGB` et pas `SDL_MapRGBA` car la composante alpha des pixels ne nous intéresse pas. Une des [remarques de la documentation](#) est que quand elle est utilisée avec un format de pixel qui a une composante alpha, elle renvoie un pixel dont la composante alpha est 255 (le pixel est totalement opaque). Si nous ne voulons pas perdre la composante alpha de notre image, il nous suffit d'utiliser `SDL_MapRGBA` et `SDL_GetRGBA`.

### 7.1.3. Négatifs

Nous allons maintenant faire en sorte d'obtenir le négatif d'une image. Mais avant d'aller plus loin, qu'est-ce que le négatif d'une image? Regardons Wikipédia rapidement.

Une image négative est une image dont les couleurs ont été inversées par rapport à l'originale; par exemple le rouge devient cyan, le vert devient magenta, le bleu devient jaune et inversement. Les régions sombres deviennent claires, le noir devient blanc. Source : [Wikipédia](#)



## 7. TP - Effets sur des images

Le négatif d'une image s'obtient donc en «inversant» chacun des pixels. La fonction mathématique qui lui est associée est donc la fonction de  $[0; 255]$  dans  $[0; 255]$  définie par  $f(x) = 255 - x$  et dont la représentation graphique est la suivante.



FIGURE 7.1. – Fonction associée au négatif.

Nous pouvons alors écrire ce code.

```
def negatif(image):  
    for i in range(image.shape[0]):  
        for j in range(image.shape[1]):  
            image[i, j] = 255 - image[i, j]
```

La composante de chaque pixel est `255 - composante_initiale`. Il suffit alors de le faire pour chaque pixel.

## 7.2. De la luminosité

Nous allons maintenant nous attarder sur la luminosité et faire des fonctions pour éclaircir, assombrir et changer le contraste d'une image.

### 7.2.1. Éclaircir une image - version naïve

Comment pouvons-nous éclaircir une image? Tout simplement, éclaircir une image, c'est lui ajouter de la luminosité. Le blanc est la couleur la plus lumineuse que nous avons, et le noir est la plus sombre. Pour éclaircir une image, nous pouvons alors faire « tendre » tous ses pixels vers le blanc.

Pour une image en niveau de gris, un pixel de niveau de gris `100` pourrait être modifié en `120` et un pixel de niveau de gris `0` changé en `20`. En gros, on rajoute `20` à chaque fois. Plus la valeur ajoutée est grande, plus l'image devient lumineuse.



Il faut cependant faire attention à ne pas dépasser la valeur `255`.

Il nous faut un entier  $n$  qui correspond à la valeur à ajouter et la fonction qu'il nous faut utiliser est alors la fonction  $f$  définie pour  $x$  appartenant à  $[0; 255]$  par

$$f(x) = \begin{cases} 255 & \text{si } x + n > 255 \\ x + n & \text{sinon} \end{cases}.$$

## 7. TP - Effets sur des images

Pour assombrir une image, le principe est le même. C'est juste qu'il faut enlever une valeur plutôt qu'en rajouter une. En fait, cela revient à éclaircir avec une valeur négative. Nous n'allons donc pas faire de nouvelle fonction pour cela mais modifier l'ancienne (il ne faut pas passer en dessous de 0). On a alors

$$f(x) = \begin{cases} 255 & \text{si } x + n > 255 \\ 0 & \text{si } x + n < 0 \\ x + n & \text{sinon} \end{cases}.$$

On a alors ces deux courbes pour  $n = 50$  (en rouge) et  $n = -50$  (en vert).

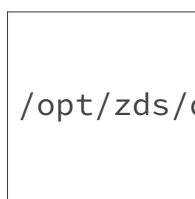
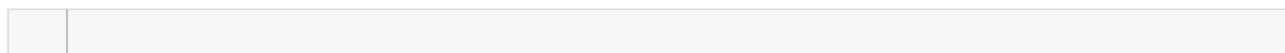
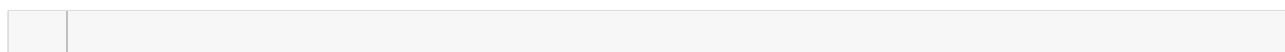


FIGURE 7.2. – Courbes de la fonction de changement de luminosité.

La courbe en bleue est celle de l'identité, c'est-à-dire qu'il s'agit du pixel normal sans aucun éclaircissement. On va d'abord faire une fonction `f` qui applique notre transformation à une composante d'un pixel (donc à un `Uint8`).



On a alors notre boucle `for`.



Notons que le prototype de la fonction est alors `SDL_Surface *creerSurface(SDL_Surface *s, int n)`.

### 7.2.2. Éclaircir une image - version améliorée

Notre fonction pour éclaircir fonctionne bien, mais a un grand défaut: toutes les composantes de pixel qui sont supérieures à `255 - valeur`, avec `valeur` la valeur ajoutée à cette composante, sont mises à 255. Les nuances de couleurs entre ces pixels seront annulées. Ainsi, si on veut éclaircir notre image d'une valeur de 50 et que toutes les composantes de tous les pixels de notre image sont supérieures à 205, on se retrouvera avec une image totalement blanche ce qui ne correspond pas à ce que l'on veut obtenir.

En fait, ce qu'il nous faut, c'est augmenter les valeurs des composantes différemment suivant si ces dernières sont grandes ou faibles. Ce qu'il nous faudrait, c'est que les noirs restent noirs, que les blancs restent blancs, et que les gris soient éclaircis. En gros, il nous faut quelque chose qui suit ce genre de courbe.

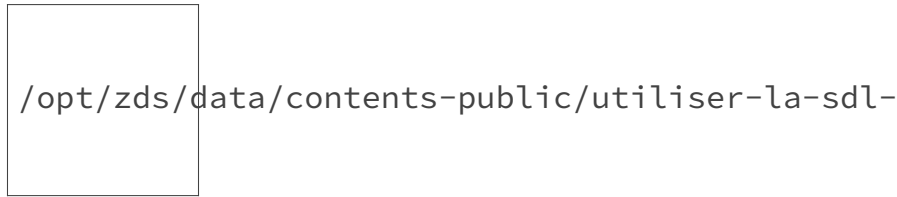


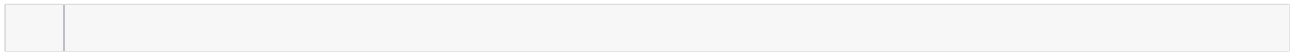
FIGURE 7.3. – Le genre de courbe que l'on veut

Ce genre de courbe est obtenu à l'aide des fonctions puissances qui ont en effet ce type de comportement sur  $[0; 1]$ . Pour «étendre» ce comportement à  $[0; 255]$ , nous allons utiliser la fonction  $f$  définie pour  $x$  appartenant à  $[0; 255]$  par

$$f(x) = 255 \times \left( \frac{x}{255} \right)^n.$$

La courbe bleue est obtenue pour  $n = 0.5$ , la verte du dessus pour  $n = 1/3$  et la verte du dessous pour  $n = 2$ . Pour  $n = 1$ , on obtiendra bien sûr une droite (ce qui correspond à un changement nul de luminosité). On remarque que pour  $n > 1$ , la courbe est «en dessous» de la droite d'équation  $y = x$  (ce qui correspond à un assombrissement) et que pour  $n < 1$ , elle est «au-dessus» (ce qui correspond à une hausse de la luminosité). Par ailleurs, plus  $n$  est grand devant 1 (respectivement petit devant 1), plus la courbe descendra (respectivement montera) et donc plus l'image sera assombrie (respectivement éclairée).

Notre nouvelle fonction d'éclairage prendra alors en paramètre ce coefficient  $n$  et appliquera à chaque pixel  $p$  la fonction  $f$ . Commençons comme tout-à-l'heure par écrire notre fonction  $f$ .



Notre boucle reste la même, on appelle la fonction pour chaque composante de chaque pixel.

Non seulement, cela nous permet de garder les nuances de notre image, mais en plus, notre code est raccourci. Il faut juste faire attention à *caster*  $c$  en **double** avant de faire la division pour bien faire une division flottante et non entière.

### 7.2.3. Contraster une image

Maintenant, faisons une opération plus compliquée: créons une fonction pour augmenter (ou diminuer) le contraste d'une image.



Mais avant tout qu'est-ce que le contraste d'une image?

Nous pouvons voir le contraste d'une image comme une propriété qui indique si la différence de luminosité entre les différents pixels de l'image est grande. Ainsi, le contraste de l'image est maximum si les pixels sombres sont entièrement noir et ceux blancs entièrement blanc, et le contraste est nul si l'image est uniformément grise.

## 7. TP - Effets sur des images

Cela permet de comprendre que pour augmenter le contraste, il faut en fait rendre les pixels lumineux encore plus lumineux et ceux sombres encore plus sombre. On peut par exemple faire cela en augmentant le niveau de gris d'un pixel s'il est entre 128 et 255 et en le diminuant sinon. Pour diminuer le contraste de l'image, il faudrait bien sûr faire le contraire, c'est-à-dire augmenter le niveau de gris des pixels s'ils sont entre 0 et 128 et les diminuer s'ils sont entre 128 et 255.

Nous voulons donc une courbe de ce genre.

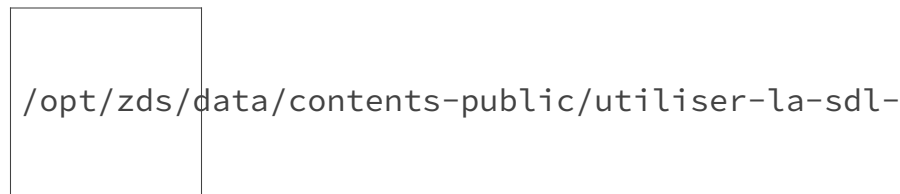


FIGURE 7.4. – La courbe de contraste.

Pour l'obtenir, il faut juste ruser un peu et utiliser les fonctions utilisées pour la luminosité. En effet, on obtient des courbes de ce genre avec la fonction  $f$  définie sur  $[0; 255]$  par

$$f(x) = \begin{cases} \frac{255}{2} \times \left(\frac{2x}{255}\right)^n & \text{si } x \in \left[0; \frac{255}{2}\right] \\ 255 - \frac{255}{2} \times \left(\frac{2(255-x)}{255}\right)^n & \text{si } x \in \left[\frac{255}{2}; 255\right] \end{cases}.$$

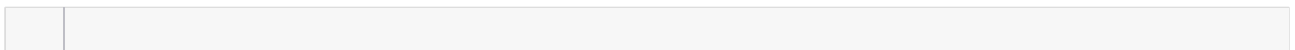
En fait, la courbe obtenue est en deux bouts. Tout d'abord la courbe d'éclaircissement ramenée sur  $\left[0; \frac{255}{2}\right]$  et son symétrique sur  $\left[\frac{255}{2}; 255\right]$ .

La courbe bleue est obtenue pour  $n = 2$ , la verte pour  $n = 1/3$  et la rouge pour  $n = 1/2$ . On remarque alors que plus  $n$  sera grand devant 1 (respectivement petit devant 1), plus l'image sera contrastée (respectivement moins l'image sera contrastée). Avec  $n = 0$ , le contraste de l'image obtenue sera nul.

Nous allons simplifier notre fonction  $f$  en remarquant que  $x > \frac{255}{2}$ ,  $f(x) = 255 - f(255 - x)$ . De plus, puisque nos valeurs sont toujours entières, nous allons prendre 127 pour la moitié de 255 et donc

$$f(x) = \begin{cases} 127 \times \left(\frac{2x}{255}\right)^n & \text{si } x \in [0; 127] \\ 255 - f(255 - x) & \text{si } x \in [128; 255] \end{cases}.$$

Il est maintenant temps de coder. Nous allons tout d'abord coder une fonction qui prend en paramètre une composante de pixel et un entier et renvoie l'image de ce pixel par la fonction  $f$ .



Là encore, notre boucle principale ne change pas.

## 7.3. Floutage

### 7.3.1. Un peu de théorie

Tout comme les autres effets que nous avons codés jusqu'ici, le floutage peut-être vu comme l'application d'une fonction à une image. La question qui se pose est: quelle fonction utiliser? Quelles opérations faut-il faire pour que notre image soit floue?

Il faut se dire que flouter l'image revient à changer certains de ses pixels pour que l'image soit moins reconnaissable. Par exemple, si on remplace un pixel sur trois par un pixel blanc, l'image obtenue sera différente mais restera proche de l'originale.

Le problème est que si par exemple on a une image toute rouge, placer des pixels blancs ne la floutera pas. En fait, on veut un effet de floutage, mais on veut que les couleurs de l'image obtenue restent proches de celles de départ. Il faut donc remplacer les pixels par des pixels de couleurs différentes, mais proches. Dans cette partie, nous allons voir l'algorithme de flou gaussien qui est assez intuitif.

### 7.3.2. Le flou gaussien

L'algorithme de flou gaussien est un algorithme assez simple à appréhender. Il consiste à remplacer un pixel par la moyenne des pixels qui l'entourent. Ainsi, on aura à peu près mélangé les pixels proches, ce qui donnera comme on le veut des couleurs proches de l'image de départ, tout en ayant du flou.

Puisque cet algorithme modifie les pixels en utilisant les valeurs d'autres pixels, cela veut dire que nous ne pouvons pas modifier directement le tableau de pixels. Nous sommes obligés dans ce cas de créer un nouveau tableau de pixels.

FIGURE 7.5. – Flou gaussien sur un pixel.

Dans l'exemple ci-dessus, le pixel rouge sera remplacé par la moyenne de ce pixel et des huit pixels qui l'entourent.

Pour obtenir une image encore plus floue, nous pouvons également remplacer un pixel par la moyenne de ce pixel, des deux pixels en hauts, des deux pixels en bas, des deux à gauche, des deux sur chaque diagonale. Nous le remplaçons par la moyenne des pixels du carré de cinq pixels dont il est le centre.

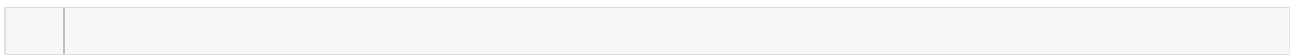
FIGURE 7.6. – Flou gaussien avec le carré de cinq pixels.

## 7. TP - Effets sur des images

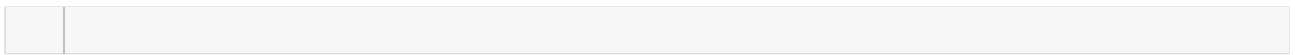
En fait, nous le remplaçons par la moyenne d'un carré dont il est le centre et plus ce carré est gros, plus l'image semblera floue. Pour un entier  $n$  donné (dans les deux images données en exemple,  $n$  prend respectivement les valeurs 1 et 2) on obtient alors cette formule générale (en considérant notre image comme une matrice) :

$$a_{i,j} = \frac{1}{(2n+1)^2} \sum_{k=i-n}^{i+n} \left( \sum_{l=j-n}^{j+n} a_{k,l} \right).$$

En gros, on fait la moyenne des pixels situés à au plus  $n$  lignes et  $n$  colonnes du pixel de départ. Il faut juste faire attention aux pixels qui se trouvent trop près du bord de l'image. On peut alors faire le code qui nous donne le flou voulu. Nous allons d'abord faire une fonction qui calcule la moyenne pour un pixel c'est-à-dire pour chacune de ses composantes.



On calcule d'abord nos bornes pour ne pas dépasser les limites de notre tableau. Puis on fait notre somme à l'aide de deux boucles et enfin on la divise par le nombre d'éléments. Notre flou est alors très simple à faire, notre double boucle est la suivante.

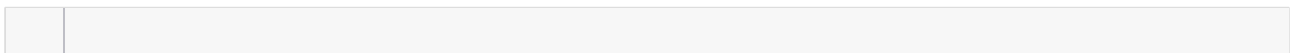


### 7.3.3. Du détournage ?

Nous n'allons pas nous attarder sur le pourquoi du comment, mais cette fonction de floutage nous permet également de faire du détournage, c'est-à-dire d'obtenir par exemple le contour d'un personnage dans une image. Pour cela, il nous faut faire cette opération, avec  $p$  le pixel que l'on traite et  $m$  la moyenne des pixels qui l'entourent.

$$a_{i,j} = 255 - |p - m|.$$

La valeur absolue permet d'être sûr d'avoir un résultat plus petit que 255. En gros, il s'agit de ce code.



---

Ce TP est maintenant fini. Mais nous n'avons touché que la surface du sujet. Nous pouvons par exemple faire une image «tirée» vers le rouge en augmentant seulement la composante rouge de ses pixels (un éclaircissement rouge en fait) ou faire le NON logique d'une image. On pourrait même jouer avec deux images de même taille en effectuant leur OU logique, leur ET logique, ou encore leur XOR logique. Il y a vraiment de quoi s'amuser.

## 8. Les évènements 1

Maintenant que nous avons joué un peu (beaucoup) avec l’affichage, il est temps de voir comment faire en sorte que notre programme réagisse aux actions du joueur comme l’appui sur une touche du clavier, le déplacement de la souris, etc.

### 8.1. Gérer les évènements

#### 8.1.1. Une file d’évènements

Nous appelons évènement toute action extérieure à notre programme et qui peut avoir un effet sur lui. L’appui sur une touche du clavier, le déplacement de la souris, le redimensionnement d’une fenêtre, et même une demande de fermeture du programme sont des évènements.

Durant toute la durée de vie de notre application, elle reçoit de nombreux évènements. Ceux-ci sont placés dans une file et attendent d’être traités par notre programme. Nous pouvons ensuite interroger la SDL pour savoir s’il y a eu un évènement. À ce moment, elle nous donne des informations sur l’évènement qui n’a pas été traité. Cela signifie notamment qu’il faudra réinterroger la SDL à chaque fois pour savoir s’il y a eu un nouvel évènement.

Bien sûr, il peut y avoir plusieurs évènements en attente. Dans ce cas, c’est l’évènement le plus vieux qui sera lu, conformément au système de file: le premier arrivé est également le premier à être traité. On peut donc voir cela comme n’importe quelle file d’attente dans la vie courante (avec les tricheurs en moins). On peut alors imaginer quelque chose de ce genre (ici, on suppose qu’on a prévu de fermer la fenêtre quand il y a une demande de fermeture du programme):

- début de notre programme;
- appui sur la touche `A`, `file = [appui A];`
- déplacement de la souris, `file = [appui A, déplacement souris];`
- lecture de la file, on apprend qu’il y a eu appui sur `A`, `file = [déplacement souris];`
- clic droit, `file = [déplacement souris, clic droit];`
- fermeture fenêtre, `file = [déplacement souris, clic droit, fermeture fenêtre];`
- lecture de la file, `file = [clic droit, fermeture fenêtre];`
- déplacement souris, `file = [clic droit, fermeture fenêtre, déplacement souris];`
- lecture de la file, `file = [fermeture fenêtre, déplacement souris];`
- clic gauche, `file = [fermeture fenêtre, déplacement souris, clic gauche];`
- lecture de la file, `file = [déplacement souris, clic gauche];`
- on a lu un évènement «fermeture fenêtre», on ferme le programme.

## 8. Les évènements 1

Ce petit exemple nous apprend beaucoup de choses à propos de la gestion des évènements que nous aurons à adopter. Lire un évènement à la fois est une très mauvaise idée, car pendant qu'on en lit un plusieurs peuvent être en train de se produire. Cela signifie qu'il faudra soit s'arranger pour lire un évènement dès qu'il survient, soit les lire tous quand on décide de lire.

### 8.1.2. La structure `SDL_Event`

La structure `SDL_Event` est la structure primordiale pour la gestion des évènements. Lorsque nous demandons à la SDL si un évènement a eu lieu, elle remplit cette structure avec les données de l'évènement correspondant (s'il y en a un). Elle contient alors toutes les informations sur le type d'évènements qui a eu lieu (par exemple, quelle touche a été pressée, à quelle position la souris a été déplacée, etc.) et nous est donc essentielle.

La structure `SDL_Event` possède un champ `type` qui prend pour valeur le type d'évènement lu, et ses autres champs (un par type d'évènement) sont des structures qui contiennent des informations précises sur l'évènement qui a eu lieu. Les valeurs pouvant être prises par ce champ sont celles de l'énumération `SDL_EventType`. Voici un tableau récapitulant les données les plus importantes.

Type d'évènements	Valeur du champ <code>type</code>	Champ de <code>SDL_Event</code> correspondant	Description
Évènements de l'application	<code>SDL_QUIT</code>	<code>quit</code>	Demande de fermeture du programme
Évènements de la fenêtre	<code>SDL_WINEVENT</code>	<code>window</code>	Changement d'état de la fenêtre
	<code>SDL_SYSWMEVENT</code>	<code>syswm</code>	Évènement dépendant du système
Évènements du clavier	<code>SDL_KEYDOWN</code>	<code>key</code>	Une touche est pressée
	<code>SDL_KEYUP</code>	<code>key</code>	Une touche est relâchée
	<code>SDL_TEXTEDITING</code>	<code>edit</code>	Édition de texte
	<code>SDL_TEXTINPUT</code>	<code>text</code>	Saisie de texte
Évènements de la souris	<code>SDL_MOUSEMOTION</code>	<code>motion</code>	Déplacement de la souris
	<code>SDL_MOUSEBUTTONDOWN</code>	<code>button</code>	Une touche de la souris est pressée
	<code>SDL_MOUSEBUTTONUP</code>	<code>button</code>	Une touche de la souris est relâchée
	<code>SDL_MOUSEWHEEL</code>	<code>wheel</code>	La molette est utilisée

Il y a quelques autres types d'évènements, comme les évènements relatifs aux mobiles (Android et iOS), ou encore ceux relatifs aux écrans tactiles ou aux manettes et autres contrôleurs.



### 8.1.3. Récupérer les évènements

Pour lire un évènement de la file, la SDL nous propose plusieurs fonctions. Chacune récupère l'évènement le plus ancien, mais elles ont un comportement différent.

#### 8.1.3.1. `SDL_WaitEvent`

La première fonction que nous verrons est la fonction `SDL_WaitEvent` [↗](#). Son prototype est le suivant.

```
int SDL_WaitEvent(SDL_Event *event);
```

Elle prend en paramètre un pointeur sur un `SDL_Event` qu'elle remplira avec les informations sur l'évènement. Elle retourne 0 en cas d'erreur et 1 en cas de succès (ce qui est contraire à beaucoup de fonctions de la SDL).

La fonction `SDL_WaitEvent` est une fonction bloquante. Quand on l'utilise, notre programme reste bloqué au niveau de cette fonction jusqu'à ce qu'un évènement ait lieu. Cela signifie que s'il n'y a aucun évènement dans la file, la fonction va attendre jusqu'à ce qu'un évènement arrive. Et dès qu'un évènement est arrivé, la fonction remplit le `SDL_Event` et termine. Faisons un code qui attend une demande de fermeture de la fenêtre pour quitter le programme.

```
int main(void) {  
    SDL_Event event;  
    SDL_WaitEvent(&event);  
    return 0;  
}
```

#### 8.1.3.2. `SDL_PollEvent`

La fonction `SDL_PollEvent` [↗](#) est l'équivalent non bloquant de `SDL_WaitEvent`. Cela signifie que s'il n'y a aucun évènement dans la file, le programme va juste continuer sa route.

```
int main(void) {  
    SDL_Event event;  
    while(SDL_PollEvent(&event)) {  
        // ...  
    }  
    return 0;  
}
```

Tout comme `SDL_WaitEvent`, elle prend en paramètre un pointeur sur un `SDL_Event` qu'elle remplira avec les informations sur l'évènement. Cependant, sa valeur de retour n'est pas liée à la présence d'erreur, mais à la présence d'évènements. Si elle a lu un évènement (donc si la file n'était pas vide), elle retourne 1, sinon elle retourne 0.

Nous avons précédemment dit qu'«il faudra soit s'arranger pour lire un évènement dès qu'il survient, soit les lire tous quand on décide de lire». La fonction `SDL_WaitEvent` correspond vraisemblablement au premier cas, la fonction `SDL_PollEvent` fait plus partie du second cas. Pour lire tous les évènements dans la file, nous allons appeler `SDL_PollEvent` tant que la file n'est pas vide, c'est-à-dire tant que le retour de `SDL_PollEvent` est différent de 0. Faisons avec `SDL_PollEvent` le même exemple que nous avons fait avec `SDL_WaitEvent`.

```
int main(void) {  
    SDL_Event event;  
    while(SDL_PollEvent(&event)) {  
        // ...  
    }  
    return 0;  
}
```

## 8. Les évènements 1

Nous avons juste placé le `SDL_PollEvent` dans une boucle. Le problème de cette méthode est que la boucle tourne indéfiniment tant qu'il n'y a pas d'évènements rencontrés, ce qui n'est pas souhaitable niveau performance. Pour régler ce problème, nous allons effectuer une petite pause à chaque tour de boucle, à l'aide de `SDL_Delay`, ce qui nous mène au code suivant.

Maintenant, un petit exercice: simulons le comportement de la fonction `SDL_WaitEvent` en utilisant la fonction `SDL_PollEvent`.

👁 Contenu masqué n°2

### 8.1.3.3. `SDL_WaitEventTimeout`

Finalement, la SDL propose une dernière fonction, la fonction `SDL_WaitEventTimeout` [↗](#).

Elle prend en paramètre un pointeur sur `SDL_Event`, mais aussi un entier. Cet entier correspond à un nombre de millisecondes. La fonction `SDL_WaitEventTimeout` attend un évènement pendant ce nombre de millisecondes. S'il y a un évènement pendant le délai imparti, elle retourne 1, sinon, s'il n'y en a pas eu ou s'il y a eu une erreur, elle retourne 0.

### 8.1.3.4. Comment est remplie la file d'évènements ?

La file d'évènements ne se remplit pas toute seule. À chaque appel de `SDL_WaitEvent`, de `SDL_WaitEventTimeout` et de `SDL_PollEvent`, la fonction `SDL_PumpEvents` [↗](#) est appelée.

Elle ne prend aucun argument et ne renvoie rien. Elle se contente d'analyser les périphériques afin de récupérer les nouveaux évènements et de les mettre dans la file.

Nous n'avons pas à l'appeler explicitement puisque les fonctions `SDL_WaitEvent`, `SDL_PollEvent` et `SDL_WaitEventTimeout` y font appel. Cependant, nous verrons plus tard comment elle peut être utilisée.

### 8.1.3.5. Comment bien gérer ses évènements

Bien gérer ses évènements n'est pas vraiment compliqué. Il suffit de suivre quelques règles simples. Pour commencer, regardons cette citation de la documentation de `SDL_PumpEvents`.

**WARNING** : This should only be run in the thread that initialized the video subsystem, and for extra safety, you should consider only doing those things on the main thread in any case.

`SDL_PumpEvents` doit être appelée uniquement dans le *thread* initiateur du mode vidéo et il est même conseillé pour plus de sécurité de l'appeler uniquement dans le *thread* principal.

De plus, il est conseillé de gérer ses évènements (donc de faire appel à `SDL_WaitEvent` par exemple) à un seul endroit du code.

Appeler une fonction de gestion des évènements une seule fois dans la boucle suffit. En fait, notre programme sera généralement de cette forme.

Finalement, pour bien gérer ses évènements, il faut bien choisir la fonction à utiliser. D'un côté, nous avons la fonction bloquante `SDL_WaitEvent`, d'un autre côté nous avons la fonction non bloquante `SDL_PollEvent` et au milieu des deux nous avons l'hybride `SDL_WaitEventTimeout`.

Nous allons privilégier `SDL_PollEvent` qui n'est pas bloquante et utiliser `SDL_WaitEvent` dans le seul cas où, sans intervention de l'utilisateur, il n'y rien à faire. `SDL_WaitEventTimeout` est un peu plus rarement utilisée.

## 8.2. Analyser les évènements

### 8.2.1. La fenêtre

Lorsque la SDL détecte un évènement de type `SDL_WINDOWEVENT`, nous pouvons récupérer les informations sur l'évènement dans le champ `window` du `SDL_Event` correspondant. Il s'agit d'une structure, `SDL_WindowEvent` [↗](#). Ses différents champs sont les suivants.

Ce qui nous intéresse le plus est son champ `event` qui correspond au type d'évènement détecté. Il peut avoir plusieurs valeurs qui correspondent aux valeurs de l'énumération `SDL_WindowEventID` [↗](#). On a alors par exemple ces différentes valeurs possibles:

- `SDL_WINDOWEVENT_SHOWN` si la fenêtre a été rendue visible;
- `SDL_WINDOWEVENT_MOVED` si la fenêtre a été déplacée;
- `SDL_WINDOWEVENT_RESIZED` si la fenêtre a été redimensionnée;
- `SDL_WINDOWEVENT_MINIMIZED` si la fenêtre a été minimisée;
- `SDL_WINDOWEVENT_RESTORED` si la fenêtre a été restaurée;
- `SDL_WINDOWEVENT_ENTER` si le focus de la souris est sur la fenêtre;
- `SDL_WINDOWEVENT_LEAVE` si le focus de la souris n'est plus sur la fenêtre;

## 8. Les évènements 1

- `SDL_WINDOWEVENT_CLOSE` si le gestionnaire de fenêtre demande la fermeture de la fenêtre.

Et d'autres que nous pouvons voir sur la page de documentation de `SDL_WindowEventID`. Par exemple, on peut savoir si la fenêtre a été redimensionnée avec ce code.

Le champ `windowID` est utile dans le cas où l'on a plusieurs fenêtres et nous permet de savoir par quelle fenêtre l'évènement a été reçu.

Nous pouvons comparer sa valeur à celle retournée par la fonction `SDL_GetWindowID` qui prend en paramètre une fenêtre et retourne son identifiant. Ainsi, si `event.window.windowID == SDL_GetWindowID(window1)`, alors c'est la fenêtre `window1` qui a été redimensionnée par exemple.

Nous pouvons également agir autrement en utilisant la fonction `SDL_GetWindowFromID` qui prend en paramètre un identifiant et renvoie la fenêtre qui a cet identifiant. Ainsi, `SDL_GetWindowFromID(event.window.windowID)` renvoie un pointeur sur `SDL_Window` qui correspond à la fenêtre ayant reçu l'évènement.

Les champs `data1` et `data2` sont utiles dans le cas d'un redimensionnement ou d'un déplacement. Dans le cas du redimensionnement, `data1` et `data2` valent respectivement la nouvelle largeur et la nouvelle hauteur de la fenêtre. Dans le cas du déplacement, `data1` et `data2` correspondent aux nouvelles positions `x` et `y` de la fenêtre.

Le champ `type` contient le type d'évènements et vaut donc `SDL_WINDOWEVENT`.

?

Pourquoi un évènement `SDL_WINDOWEVENT_CLOSE` alors qu'il y a déjà l'évènement `SDL_QUIT`?

Cette question est légitime, mais il ne faut pas oublier que nous pouvons ouvrir plusieurs fenêtres, la fermeture d'une fenêtre ne signifie donc pas forcément que l'on quitte le programme. Quand plusieurs fenêtres sont utilisées, cliquer sur la croix de l'une d'entre elles n'entraînera pas l'évènement `SDL_QUIT`.

### 8.2.2. Le clavier

Lorsque c'est un évènement de type `SDL_KEYDOWN` ou `SDL_KEYUP`, les informations sur l'évènement sont dans le champ `key` de `SDL_EVENT`. Il s'agit cette fois de la structure `SDL_KeyboardEvent` dont les champs sont les suivants.

On retrouve, comme pour les évènements de type `SDL_WINDOWEVENT`, le champ `windowID` qui correspond à l'identifiant de la fenêtre sur laquelle l'évènement a été détecté et le champ `type` qui vaut alors `SDL_KEYDOWN` ou `SDL_KEYUP`. Le champ `state` contient l'état de la touche et

## 8. Les évènements 1

vaut `SDL_PRESSED` si la touche est pressée et `SDL_RELEASED` si elle est relâchée. Le champ `repeat` est différent de 0 s'il y a eu répétition de l'appui.

Le champ qui nous intéresse le plus est `keysym`. Il s'agit d'une structure [SDL\\_Keysym](#). C'est dans ses champs que l'on retrouvera des informations sur la touche pressée ou libérée. Ses champs sont les suivants.

--	--

Les champs `scancode` et `sym` correspondent tous les deux à la touche pressée (ou relâchée), mais le premier correspond au code de la touche physique alors que le second correspond au code de la touche virtuelle. Ils prennent comme valeur celles des énumérations [SDL\\_Scancode](#) et [SDL\\_Keycode](#). Par exemple, pour savoir si on a appuyé sur la touche physique `A`, on vérifiera la condition `event.key.keysym.scancode == SDL_SCANCODE_A` et pour la touche virtuelle on fera le test `event.key.keysym.sym == SDLK_A`.

?

Mais quelle est la différence entre touche physique et touche virtuelle?

En fait, la touche physique ne dépend pas du système et est la même sur tous les ordinateurs. Ainsi, `SDL_SCANCODE_A` correspond à l'appui sur la touche `A` d'un clavier Qwerty, mais à la touche `Q` d'un clavier Azerty et à la touche `A` d'un clavier Bépo car ce qui compte ici, c'est la place physique de la touche sur le clavier. Au contraire, la touche virtuelle dépend du système et non du type de clavier, `SDLK_a` correspond donc toujours à l'appui sur la touche `A`. Pour bien voir cette différence, essayons ce code:

--	--

En testant ce code sur un clavier Azerty, il nous faudra appuyer sur `A` pour que le message «`keysym A`» s'affiche, mais il nous faudra appuyer sur `Q` pour que «`scancode A`» s'affiche.

Le champ `mod` de `SDL_Keysym` nous permet de savoir si l'utilisateur a appuyé sur une (ou plusieurs) touche(s) spéciale(s) (`Alt`, `Ctrl`, `Shift`, etc.). Sa valeur correspond au OU logique de valeurs de l'énumération [SDL\\_Keymod](#). Par exemple, pour savoir si on a appuyé sur `Ctrl`, on vérifiera si `event.key.keysym.mod & KMOD_CTRL != 0`.

### 8.2.3. La souris

Pour la souris, il y a, comme nous l'avons vu, trois types d'évènements. Les évènements du type déplacement de la souris, les appuis sur les touches et le déplacement de la molette.

### 8.2.3.1. Boutons de la souris

Lorsqu'il y a un évènement `SDL_MOUSEBUTTONDOWN` ou un évènement `SDL_MOUSEBUTTONUP`, les informations sur l'évènement sont placées dans le champ `button` de `SDL_Event` qui est une structure [SDL\\_MouseButtonEvent](#) . Ses champs sont les suivants.

--	--

On a comme d'habitude l'identifiant de la fenêtre dans le champ `windowID` et le type d'évènement dans le champ `type` (il vaut alors `SDL_MOUSEBUTTONDOWN` ou `SDL_MOUSEBUTTONUP`). Le champ `which` contient l'identifiant de la souris. Il peut servir dans le cas d'un appareil tactile (s'il y a appui sur l'écran, `which` vaudra `SDL_TOUCH_MOUSEID`). Le champ `state` correspond comme pour le clavier à l'état de la touche qui a déclenché l'évènement. Il vaut `SDL_PRESSED` si le bouton est pressé et `SDL_RELEASED` s'il est relâché. Les champs `x` et `y` correspondent à la position de la souris (par rapport à la fenêtre) au moment de l'évènement.

Le champ `button` contient les informations sur la touche pressée et peut prendre plusieurs valeurs suivant la touche pressée. En particulier, on a les valeurs suivantes:


- `SDL_BUTTON_LEFT` si c'est un clic gauche;
- `SDL_BUTTON_RIGHT` si c'est un clic droit;
- `SDL_BUTTON_MIDDLE` s'il y a appui sur la molette.

Finalement, le champ `clicks` peut s'avérer intéressant car il nous permet de savoir combien de clic il y a eu. Il vaut 1 dans le cas d'un clic simple, 2 dans le cas d'un double clic, etc.

Par exemple, on testera s'il y a eu double clic gauche avec ce code.

--	--

### 8.2.3.2. Déplacement de la souris

Dans le cas d'un évènement du type `SDL_MOUSEMOTION`, les données sont stockées dans le champ `motion` de `SDL_Event` qui est une structure [SDL\\_MousMotionEvent](#)  qui a les champs suivants.

--	--

Les champs `windowID`, `which`, `x` et `y` fonctionnent de la même manière que dans la structure `SDL_MousButtonEvent`. Et comme d'habitude, `type` contient le type d'évènement (ici `SDL_MOUSEMOTION`).

Le champ `state` nous permet de savoir sur quelles touches de la souris l'utilisateur appuyait pendant le déplacement de la souris. C'est une combinaison de différents masques qu'on trouve dans la documentation. Par exemple, on peut tester si `event.motion.state & SDL_BUTTON_LMASK` pour savoir s'il y a appui sur la touche gauche pendant le déplacement.

## 8. Les évènements 1

Les champs `xrel` et `yrel` caractérisent le mieux le déplacement puisqu'ils nous donnent respectivement les déplacements relatifs sur les axes des `x` et des `y`. Par exemple, si on s'est déplacé de 1 pixel vers le haut et de 2 pixels vers la droite, alors `xrel` vaudra 2 et `yrel` vaudra -1.

Avec le code qui suit, on affiche la position de la souris et les déplacements relatifs à chaque fois qu'il y a déplacement et appui sur la touche gauche de la souris.

### 8.2.3.3. Utilisation de la molette

Et finalement, lorsqu'un évènement du type `SDL_MOUSEWHEEL` est détecté, les informations sont placées dans le champ `wheel` de `SDL_Event`. Il s'agit d'une structure [SDL\\_MouseWheelEvent](#). Voici ses champs.

On retrouve les champs `which`, `type` (qui vaut cette fois `SDL_MOUSEWHEEL`), et `windowID`. Le champ `y` caractérise le défilement vertical. Cette valeur est positive s'il y a eu défilement vers le haut et négative si le défilement était vers le bas. `x` caractérise le déplacement horizontal de la molette (encore faut-il que la molette en soit capable) et est positif si le déplacement s'est fait vers la droite et négatif sinon.

Ainsi, avec ce code, on affiche la valeur de déplacement à chaque déplacement de la molette.

En testant ce code, on s'aperçoit qu'il est en fait très compliqué, mais vraiment très compliqué d'avoir une valeur de déplacement supérieure à 1 (ou inférieure à -1).

Nous n'avons jamais parlé du champ `timestamp` qui est présent dans toutes les structures que nous avons vues ici. Ce champ a pour valeur le nombre de millisecondes écoulées entre l'initialisation de la SDL et l'évènement que l'on est en train d'analyser. Puisque nous parlons de *timestamp*, profitons-en pour présenter la fonction [SDL\\_GetTicks](#) qui permet d'obtenir le temps en millisecondes depuis l'initialisation de la SDL. Son prototype est le suivant.

## 8.3. Le statut des périphériques

Dans certains cas, ce qui nous intéresse est de savoir l'état de chaque touche (clavier et souris) à n'importe quel moment. Donc en gros, on veut juste pouvoir savoir si telle ou telle touche est pressée ou pas à n'importe quel moment. La SDL propose un mécanisme pour cela en nous proposant des fonctions pour récupérer l'état des touches.

Ces fonctions, contrairement à celles que nous avons précédemment vues, n'appellent pas `SDL_PumpEvents`. C'est donc à nous de faire cet appel, et ensuite, nous appellerons nos fonctions.

### 8.3.1. La souris

Pour récupérer l'état de la souris, il nous faut utiliser la fonction `SDL_GetMouseState` [↗](#). Voici son prototype.

```
SDL_MouseState SDL_GetMouseState(int* x, int* y);
```

Elle prend en paramètre deux pointeurs sur `int` qui seront remplis avec les positions de la souris. Elle renvoie un `Uint32` qui nous permettra de savoir quelles touches sont pressées en utilisant des *flags*. On l'utilise comme ceci.

```
SDL_BUTTON(SDL_GetMouseState(&x, &y));
```

`SDL_BUTTON` est une macro qu'il faudra utiliser pour tester si une touche est pressée.

Si la position de la souris ne nous intéresse pas, nous pouvons lui passer `NULL` en paramètre.

#### 8.3.1.1. Déplacement de la souris

Si nous voulons récupérer le déplacement de la souris, nous pouvons utiliser la fonction `SDL_GetRelativeMouseState` [↗](#), dont le prototype est le suivant.

```
SDL_MouseState SDL_GetRelativeMouseState(int* x, int* y);
```

Elle a le même prototype que `SDL_GetMouseState` et s'utilise de la même manière. La seule différence est que `x` et `y` ne vaudront pas la position de la souris par rapport à la fenêtre, mais sa position par rapport à sa dernière position, c'est-à-dire par rapport à la position enregistrée lors du dernier appel à `SDL_GetRelativeMouseState`. C'est donc bien le déplacement que l'on obtient.



### 8.3.2. Le clavier

La fonction pour obtenir l'état du clavier est étonnamment la fonction `SDL_GetKeyboardState` [↗](#).

```
int SDL_GetKeyboardState(SDL_KeyboardState *state)
```

Elle prend en paramètre un pointeur sur `int` et retourne un pointeur sur `Uint8`. Le pointeur qu'elle renvoie pointe sur la première case d'un tableau dont les éléments sont les états des différentes touches du clavier (1 si la touche est pressée et 0 sinon). Il s'agit d'un tableau géré par la SDL. Il est valide pendant toute l'exécution du programme et nous n'avons pas besoin de le libérer manuellement. Pour savoir quelle case du tableau correspond à quelle touche, il nous faut utiliser les valeurs de l'énumération `SDL_Scancode`. Par exemple, `tab[SDL_SCANCODE_RETURN] == 1` signifie que la touche `Entrée` est pressée.

La variable sur laquelle pointe `numkeys` sera modifiée et vaudra la longueur du tableau. Comme nous n'avons généralement pas besoin de connaître la longueur de ce tableau, nous passons généralement `NULL` en argument. On peut alors écrire ce code qui boucle tant qu'on n'appuie pas sur `Échap` ou sur `Entrée`.

```
while (1) {
```

#### 8.3.2.1. Les touches spéciales du clavier

Pour les touches spéciales du clavier, il nous faut encore utiliser une autre fonction. Il s'agit de la fonction `SDL_GetModState` [↗](#) dont le prototype est le suivant.

```
SDL_KeyboardModState SDL_GetModState(void)
```

Elle ne prend aucun paramètre et renvoie une combinaison de OU logique des touches spéciales pressées. Pour tester si une touche est pressée, on va alors utiliser le ET logique.

```
if (SDL_GetModState() & KMOD_SHIFT)
```

#### 8.3.3. Une structure pour la gestion des évènements

Cela peut être encore mieux si nous utilisons notre propre structure pour gérer tout ça. Nous pourrions alors placer une touche à zéro à la main (imaginons un jeu où l'on voudrait que l'utilisateur ne puisse pas maintenir la touche de tir enfoncée, mais doive la relâcher avant de ré-appuyer pour tirer à nouveau). Pour la mettre à jour, nous n'utiliserons pas les fonctions d'acquisition d'état mais une boucle avec un `SDL_PollEvent` qui nous permettra d'y mettre également l'évènement `SDL_QUIT`. D'ailleurs, on va commencer par gérer cet évènement et le clavier.

## 8. Les évènements 1

La constante `SDL_NUM_SCANCODES` est une constante qui est plus grande que la plus grande des valeurs de l'énumération `SDL_Scancode`. Nous sommes alors sûrs que notre tableau est suffisamment grand pour contenir toutes les touches du clavier. `x` et `y` représentent la position de la souris, `xrel` et `yrel` le déplacement de la souris, et `xwheel` et `ywheel` le déplacement de la molette. Pour savoir quelle valeur prendre pour le tableau `mouse`, nous pouvons regarder les déclarations des différentes valeurs des boutons de la souris.

Dès lors, il nous suffirait d'un tableau de 5 éléments, mais il nous faudrait décaler les indices de 1. Nous allons plutôt déclarer un tableau de six éléments (de plus, le jour où nous voudrions gérer les appareils tactiles, nous pourrions utiliser la première case pour `SDL_TOUCH_MOUSEID`).

Cela donne lieu à cette fonction de mise à jour de notre structure.

Et là, nous pouvons écrire ce genre de code.

Ici, en restant appuyé sur `C` et `D`, le second message n'arrêtera pas de s'afficher, mais nous sommes obligés de relâcher la touche `A`, puis de ré-appuyer pour ré-afficher le premier message.

Nous pourrions rajouter à notre structure d'autres fonctionnalités telles que la gestion de certains des évènements de la fenêtre ou la gestion des touches spéciales du clavier.

Notre structure est très intéressante. Elle nous permet de séparer la gestion des évènements du reste du programme. On ne met à jour notre structure qu'au début de notre boucle. On se contente ensuite de vérifier sur quelle touche il y a appui.

Notons qu'avant d'utiliser la structure, il faut l'initialiser en la remplissant de `SDL_FALSE` pour que l'on ne puisse pas détecter d'évènements alors qu'en fait il n'y en a pas. Pour cela, nous pouvons par exemple utiliser la fonction `memset`.

---

Maintenant que nous avons vu comment gérer les événements, nous pouvons enfin faire des programmes interactifs.

---

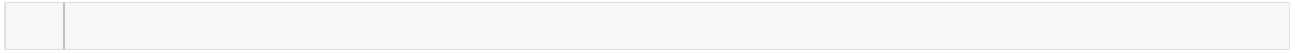
Ce tutoriel n'est pas encore fini et d'autres chapitres risquent bien de voir le jour. L'apprentissage non plus n'est pas fini, il reste plein de choses à apprendre et l'univers de la SDL est très vaste.

Un grand merci à tous les membres qui ont apporté leur aide lors de la bêta, à [Taurre](#) qui a fait un travail monstrueux pour valider ce tutoriel et surtout merci aux lecteurs!

## Contenu masqué

### Contenu masqué n°2

Le principe est assez simple, il suffit de rappeler `SDL_PollEvent` tant qu'il n'y a pas d'évènements, c'est-à-dire tant que la file est vide et donc tant que la valeur retournée est 0. Dans notre fonction, nous n'aurons alors que cette ligne (et on peut y rajouter une petite pause).



Ici, la seule chose que l'on pourrait nous reprocher est de ne pas gérer les erreurs. De plus, la fonction que nous codons ici n'est pas tout à fait similaire à `SDL_WaitEvent` car cette dernière implique que le programme ne soit plus exécuté tant qu'un évènement n'est pas rencontré.

[Retourner au texte.](#)