

(1) Installer GL4D

Tout est là <https://gl4d.api8.fr/FR/>
(On se donne une heure max)

Université Paris 8 - Vincennes à Saint-Denis



Annexe : Installation

Farès Belhadj

Date de MAJ : 6 octobre 2021

email : <mailto:amsi@up8.edu>
github : <https://github.com/noalien/GL4Dummies>
web [GL420] : <https://gl4d.api8.fr>

Annexe A

Installation de la Bibliothèque *GL4Dummies*

GL4Dummies étant une bibliothèque multiplateformes et *Open Source*, son code est disponible en téléchargement¹ et il est possible de la recompiler spécifiquement pour votre système d'exploitation.

La première section A.1 traite de l'installation multi-OS de *GL4Dummies* à partir de son code source.

La seconde section A.2, spécifique aux utilisateurs de *Windows 10*, détaille l'utilisation de *GL4Dummies*, avec l'IDE *Visual Studio Community 2019*, sans recours à la recompilation de la bibliothèque.

A.1 À partir du code source

Cette section décrit l'installation de la bibliothèque *GL4Dummies* à partir de son code source. Dans chaque sous-partie vous trouverez les indications à suivre en fonction de votre système d'exploitation : Linux / Unix-like / Mac OS X et Windows.

A.1.1 Prérequis : la bibliothèque *SDL2*

SDL2 (<http://www.libsdl.org/>) est une bibliothèque de développement multi-plateformes permettant un accès *hardware* aux ressources graphiques présentes sur le système. Elle permet aussi l'accès bas niveau aux ressources

1. Vous pouvez télécharger *GL4Dummies* à partir de l'adresse :
<https://github.com/noalien/GL4Dummies>

d'interface (clavier, souris et joystick). Aussi, des extensions de la bibliothèque donnent un accès simple aux ressources audio, à la gestion des *pthread*, à l'utilisation de *fonts Truetype*, à l'ouverture et l'enregistrement de multiples formats d'images, etc.

GL4Dummies dépend de la bibliothèque *SDL2* principalement pour ce qui concerne la gestion de fenêtres et du contexte OpenGL® (pour faire court : zone de l'écran dans laquelle OpenGL® effectue ses rendus). Il est nécessaire d'avoir la bibliothèque de développement (soit les paquets contenant les *headers* et les bibliothèques statiques) de *SDL2* afin de pouvoir recompiler *GL4Dummies*. Aussi, pour quelques exemples proposés avec la bibliothèque, il sera nécessaire d'avoir certaines extensions de la bibliothèque, telles que : *SDL_images*, *SDL_mixer* et *SDL_ttf*. Faire attention à récupérer les versions pour *SDL2* et non *SDL1.2* (l'ancienne version de la bibliothèque ne permet pas de gérer un contexte OpenGL® 3 et plus).

Afin de vérifier que *SDL2* est bien installée, et pour faire simple, nous pouvons considérer que la bibliothèque de développement est présente sur le système si la commande *sdl2-config* est présente et répond en donnant les bons chemins vers les différents fichiers (voir *sdl2-config --help*). Sinon, en fonction de l'OS (système d'exploitation) faire :

Pour les utilisateurs Linux, utiliser le gestionnaire de paquets afin d'installer la bibliothèque de développement (lib dev) de *SDL2*. Par exemple un utilisateur Ubuntu fera :

```
sudo apt install libsdl2-dev
```

Pour les utilisateurs Mac OS X, commencer par installer les *MacPorts*² (<http://www.macports.org>). Une fois les *MacPorts* pleinement fonctionnels, installer *SDL2* en tapant (dans un terminal) :

```
sudo port install libsdl2
```

Pour les utilisateurs Windows, pas besoin d'installer *SDL2* tout est dans l'archive *GL4Dummies* sauf si vous souhaitez utiliser une version plus récente de *SDL2*.

Enfin, il est aussi possible d'installer *SDL2* à partir de son source, néanmoins, cette procédure n'est pas recommandée car certaines dépendances liées aux exemples ne seront pas automatiquement incluses dans la bibliothèque ; par exemple : le support de différents formats d'images ou de fichiers audio.

2. Concernant les débutants, par expérience nous ne recommandons pas l'usage du gestionnaire de paquets *Homebrew*. Ce gestionnaire est flexible et offre un très bon niveau de contrôle sur ce qui est installé et comment les paquets sont installés. Le revers de la médaille est qu'un débutant en fera souvent un mauvais usage l'emmenant assez rapidement à des installations bancales.

A.1.2 Compilation et installation de *GL4Dummies*

① Récupérer le code source de la bibliothèque :

- Soit en récupérant, dans le sous-dossier releases, une release *GL4Dummies* et en la décompressant (par exemple : tar zxvf gl4dummies-X.Y.Z.tar.gz où X.Y.Z est la version de la release) afin d'obtenir le dossier sources de *GL4Dummies* ;
- Soit en récupérant directement le GIT (installer git, pkg-config et les outils autotools automake, autoconf et libtool) :

```
git clone https://github.com/noalien/GL4Dummies.git GL4Dummies
cd GL4Dummies
make -f Makefile.autotools
```

② Attention cette étape est découpée en trois parties, chacune correspond à une situation particulière de l'utilisateur (*Posix* avec droits administrateur, *Posix* sans droits administrateur et enfin *Windows*). Ainsi, depuis le dossier contenant le source de *GL4Dummies*, faire selon la situation :

Sous Posix (Linux, OSX, ...) avec droits administrateur, faire :

```
./configure
make
sudo make install
```

Puis, sur certains Linux, il sera nécessaire de spécifier que les variables d'environnements \$PATH et \$LD_LIBRARY_PATH pointent respectivement sur /usr/local/bin et /usr/local/lib. Ainsi, selon votre OS, à la racine de votre compte, vous devriez avoir (par ordre de préférence) un ou plusieurs des fichiers suivants : .bashrc, .bash_profile, .profile ou encore .zprofile. Editez l'un d'eux en ajoutant les lignes suivantes à la fin du fichier :

```
export PATH=$PATH:/usr/local/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

Ces fichiers permettent de paramétrer votre environnement utilisateur au moment du login (.bash_profile ou .zprofile) ou lors de l'ouverture d'un nouveau bash (.bashrc). Pour forcer la relecture d'un de ces fichiers de paramétrage, utilisez la commande source. Par exemple, si vous aviez ajouté les deux lignes au .bashrc, faire :

```
source ~/.bashrc
```

Sous Posix (Linux, OSX, ...) sans droits administrateur, il peut être nécessaire de créer un dossier local à la racine de votre compte et y installer la bibliothèque (bibliothèque statique et dynamique, *headers* et autres fichiers nécessaires). En premier lieu, vérifiez que votre variable d'environnement `$HOME` pointe bien à la racine de votre compte (`echo $HOME`) et tapez :

```
[ -d $HOME/local ] || mkdir $HOME/local  
./configure --prefix=$HOME/local  
make  
make install
```

et là il vous reste à paramétrer vos chemins en fonction de `$HOME/local` où l'installation a été faite.

Pour que le système ait connaissance de ce chemin d'installation particulier (les dossiers `bin` `include` `lib` et `share` ont été ajoutés dans `$HOME/local`) il faut renseigner ses variables d'environnement `$PATH` et `$LD_LIBRARY_PATH`. Selon votre OS, à la racine de votre compte, vous devriez avoir (par ordre de préférence) un ou deux des fichiers suivants `.bashrc`, `.bash_profile`, `.profile` ou encore `.zprofile`. Éditez l'un d'eux en ajoutant les deux lignes suivantes à la fin du fichier :

```
export PATH=$PATH:$HOME/local/bin  
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/local/lib
```

Ces fichiers permettent de paramétrer votre environnement utilisateur au moment du login (`.bash_profile` ou `.zprofile`) ou lors de l'ouverture d'un nouveau bash (`.bashrc`). Pour forcer la relecture d'un de ces fichiers de paramétrage, utilisez la commande source. Par exemple, si vous aviez ajouté les deux lignes au `.bashrc`, faire :

```
source ~/.bashrc
```

Sous Windows, ouvrir le répertoire *Windows* à l'emplacement contenant le code source de *GL4Dummies*, et, selon votre IDE : *Code::blocks* ou *Visual Studio*, vous trouverez respectivement un projet `GL4Dummies.cbp` ou `GL4Dummies.sln`. Ouvrez celui qui correspond à votre IDE et compiler le projet. Dans le cas de *Visual Studio*, il compilera la bibliothèque puis les exemples utilisant la bibliothèque, il ne reste plus qu'à exécuter. Dans le cas de *Code::blocks* la bibliothèque est compilée et les binaires créés dans le dossier `bin/Win32` à partir de la racine du dossier contenant les sources. Les projets exemples (fichiers `.cbp`) se trouvant dans *Windows* peuvent maintenant être compilés et exécutés.

A.1.3 Paramétrage de vos futurs projets

En fonction de l'emplacement où est installée la *GL4Dummies*, il faudra paramétrier vos futurs projets (par exemple paramétrier le *Makefile*) pour qu'ils puissent compiler et *linker* avec *GL4Dummies*. Vous devez ainsi vérifier que la ligne de compilation contient bien le chemin vers le dossier *include* de *GL4Dummies* et que la ligne de *link* (*ld*) contient le chemin vers le dossier *lib* et demande à utiliser *GL4Dummies* (par exemple *-lGL4Dummies*).

Par exemple, quand la bibliothèque est installée dans un *\$HOME/local*, en utilisant un *Makefile*, vérifier dans ce dernier que les variables *CPPFLAGS* et *LDLFLAGS* contiennent au minimum :

```
CPPFLAGS = -I. -I$(HOME)/local/include $(shell sdl2-config --cflags)
LDLFLAGS = -lm -L$(HOME)/local/lib -lGL4Dummies $(shell sdl2-config --libs)
```

Pour la situation spécifique d'utilisateurs travaillant sous *Visual Studio*, se référer à la sous-section A.2.1 qui détaille le paramétrage de projets dans ce cas.

A.2 À partir des binaires : Windows 10 – Visual Studio Community 2019

Pour cette partie, nous considérons que le lecteur utilise la version pré-compilée – pour Windows 10 et Visual Community 2019 – de *GL4Dummies* disponible à partir de l'adresse suivante :

https://github.com/noalien/GL4Dummies/tree/master/builds/MSVC_19

Une fois le dossier *MSVC_19* récupéré (cloner le *repository* ou en récupérer une archive), vous devez :

1. Copier les DLLs depuis le dossier *DLLs* de la manière suivante :
 - (a) Copier toutes les DLLs depuis *DLLs/x86/* vers "*%systemroot%\SysWoW64*" (habituellement "*C:\Windows\SysWoW64*");
 - (b) Copier toutes les DLLs depuis *DLLs/x64/* vers "*%systemroot%\System32*" (habituellement "*C:\Windows\System32*");
2. Copier le répertoire *GL4D* vers "*C:*", de manière à avoir "*C:\GL4D*" sur votre système³

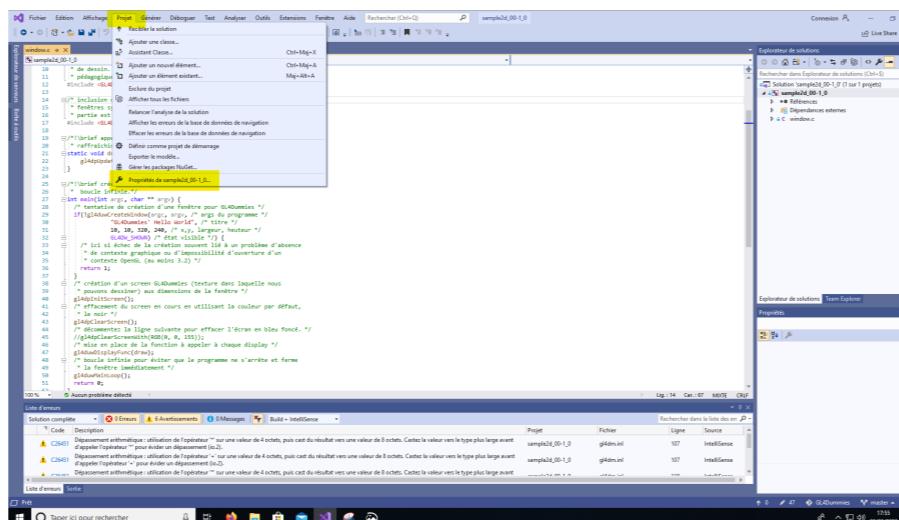
³. Vous pouvez aussi modifier les options de Visual Studio pour ajouter respectivement "*C:\GL4D\include;C:\GL4D\include\SDL2*" comme répertoires additionnels d'inclusion (*includes* ou *headers*), et "*C:\GL4D\lib\x86*" et/ou "*C:\GL4D\lib\x64*" comme répertoires additionnels respectivement pour des bibliothèques (les *.lib*) de plateformes *Win32* ou *x64*.

3. Retourner au *github* de *GL4Dummies*, téléchargez un *Sample Code* (dossier **samples**) vers votre répertoire de travail, utilisez le fichier **sln – the Visual Studio Solution file** (s'il y en a un, sinon créer le vôtre en mettant vos bons chemins vers les *includes* et les *libs*) – afin de compiler et exécuter l'exemple.

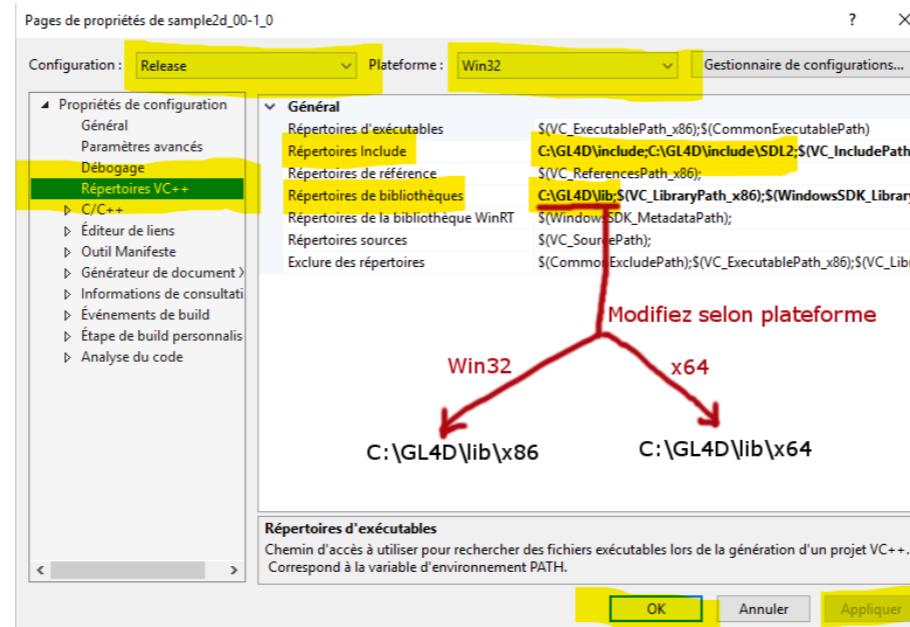
A.2.1 Paramétriser les projets pour utiliser *GL4Dummies*

Nous reprenons, au travers des captures d'écran qui suivent, quelques éléments essentiels à la bonne configuration d'un projet Visual Studio afin que ce dernier puisse utiliser les bonnes bibliothèques permettant de compiler, lier et exécuter un programme graphique utilisant *GL4Dummies*, *SDL2* et *OpenGL®*. La typologie de projet généralement utilisée est "Application console" car elle permet de garder une sortie texte pour les messages de type informatif ou d'erreur mais elle n'est pas toujours nécessaire.

La capture d'écran ci-après montre le menu contextuel permettant d'accéder aux propriétés du projet mis au premier plan (une solution pouvant être composée de plusieurs projets) :

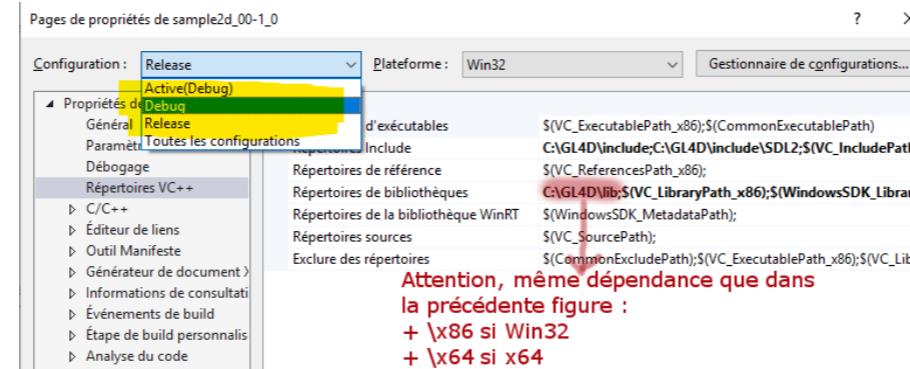


La fenêtre "page de propriétés" du projet permet d'accéder aux multiples options de "fabrication" du projet. Nous attirons votre attention sur le fait que ces options sont potentiellement liées à plusieurs configurations (ici *Debug* ou *Release*) et à plusieurs plateformes d'exécution (par exemple *Win32* ou *x64*). Après chaque changement dans une configuration ou au niveau de la plateforme, vous devez valider le changement via le bouton "Appliquer" avant de passer à une autre configuration ou plateforme. Ici nous mettons en valeur une catégorie d'options – *Répertoires VC++* – pour laquelle nous paramétrons les dossiers d'inclusion (où aller chercher les fichiers "**.h*") et les dossiers contenant les bibliothèques (où aller chercher les fichiers "**.lib*") en ajoutant des chemins liés à *GL4Dummies* :

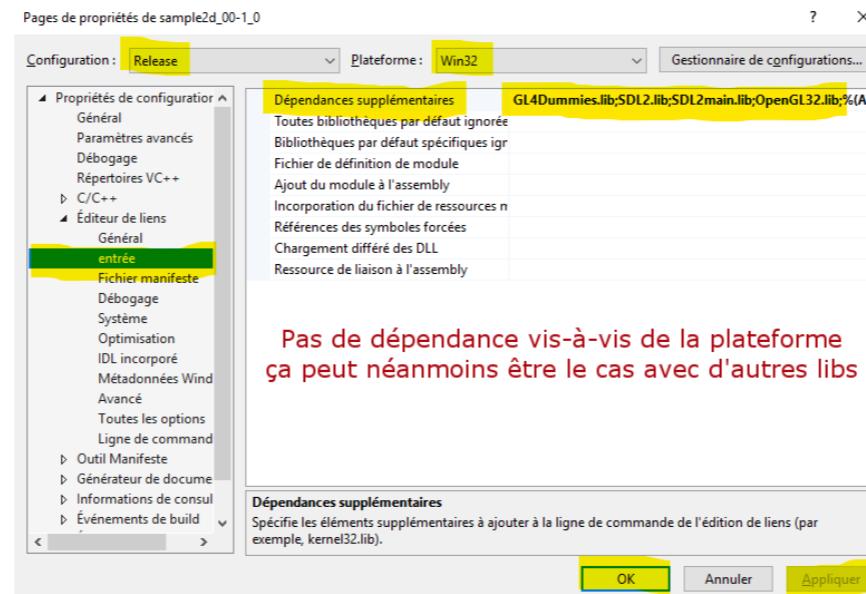


La capture d'écran ci-dessous montre justement un changement de configuration. Chacune a ses propres options, parfois certaines options sont communes et peuvent être modifiées en une fois via la sélection de la configuration "Toutes les configurations" (ceci est aussi valable pour les plateformes) :

Installation de la Bibliothèque GL4Dummies



Enfin, cette capture d'écran illustre les modifications de la propriété "éditeur de liens → entrée". Le champ "Dépendances supplémentaires" indique nommément les bibliothèques utilisées lors de l'édition de liens. Dans notre cas, ceci rend le programme dépendant des DLLs *GL4Dummies*, *SDL2* et *OpenGL*® :



(2)

Introduction à l'interaction

Appliquée à GL4Dummies

Voir aussi la doc de référence à l'adresse <https://gl4d.api8.fr/doxygen/html/index.html>

Démarrage

```
création de la fenêtre;  
initiation des paramètres;  
initiation des données;  
atexit(quit);  
...  
paramétrage des fonctions callBack;  
-> boucle event-simu-draw;
```

Evénements

```
...  
-> exit(e);  
...
```

Boucle event-simu-draw

```
-> événements  
-> simulation  
-> dessin  
-> SwapBuffers
```

Quit

```
destruction;
```

Démarrage

```
création de la fenêtre;  
initiation des paramètres;  
initiation des données;  
atexit(quit);  
...  
paramétrage des fonctions callBack;  
-> boucle event-simu-draw;
```

Quels types d'événements ?

Boucle event-simu-draw

- > événements
- > simulation
- > dessin
- > SwapBuffers

Événements

```
...  
-> exit(e);  
...
```

Quit

destruction;

Quels types d'événements ?

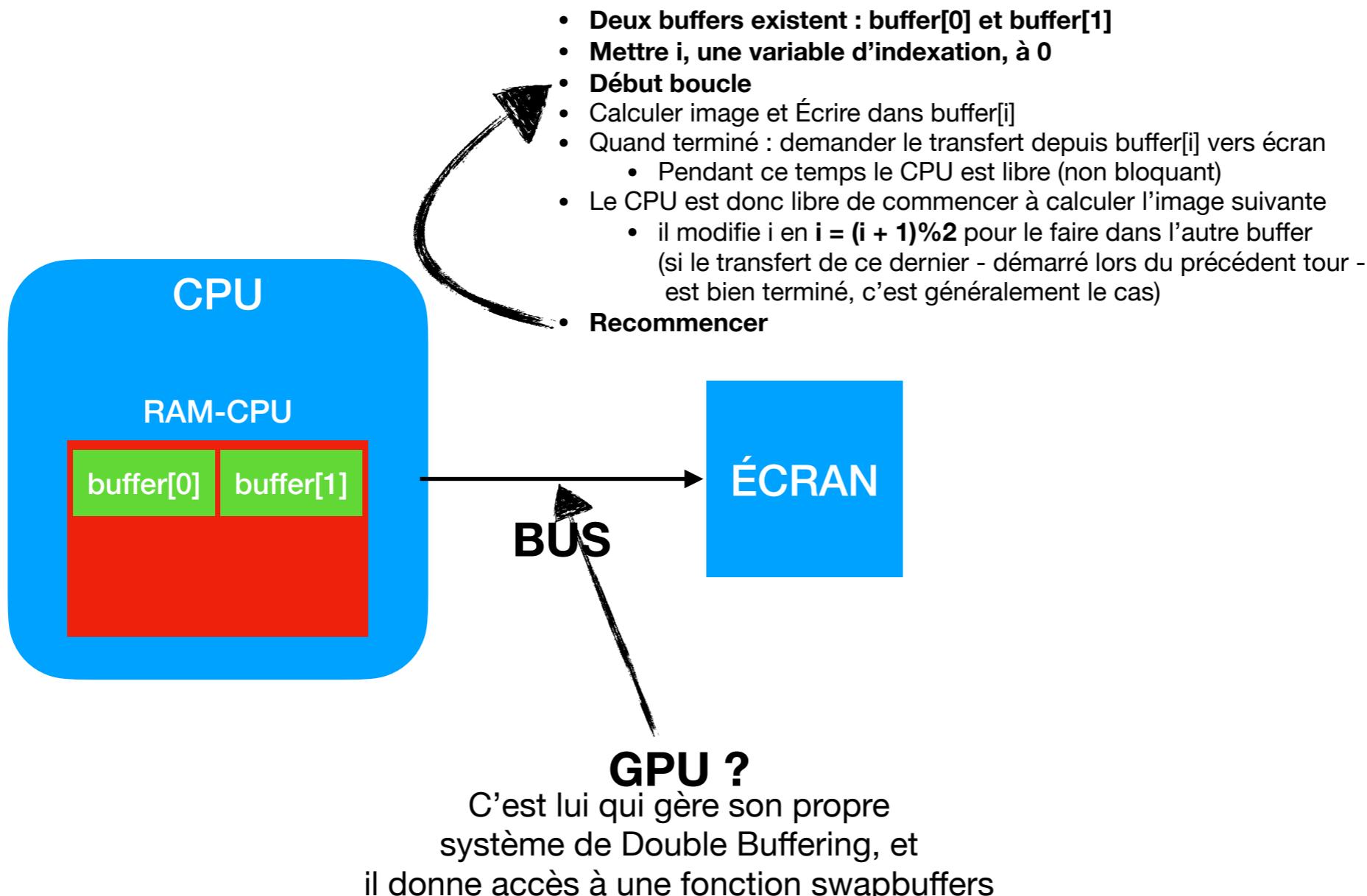
- Interactions clavier (plusieurs types)
- Interactions souris (plusieurs types)
- Actions sur la fenêtre :
 - retailler, perte de focus, fermeture, ...
- Autres interactions possibles (extérieurs) :
 - Divers capteurs, caméra RGB, caméra RGBD, Motion Leap, ...

Quels types d'événements ? (GL4Dummies)

- Interactions clavier (plusieurs types)
 - ➔ gl4duwKeyDownFunc(void(*)(int keycode) **mafondction**) => void **mafondction**(int code_de_latouche) { ... }
 - ➔ gl4duwKeyUpFunc(void(*)(int keycode) **mafondction**) => void **mafondction**(int code_de_latouche) { ... }
- Interactions souris (plusieurs types)
 - ➔ gl4duwMouseFunc(void(*)(int button, int state, int x, int y) **mafondction**) => void **mafondction**(int button, int state, int x, int y) { ... }
 - ➔ gl4duwMotionFunc(void(*)(int x, int y) **mafondction**) => void **mafondction**(int, x, int y) { ... }
 - ➔ gl4duwPassiveMotionFunc(void(*)(int x, int y) **mafondction**) => void **mafondction**(int, x, int y) { ... }
- Actions sur la fenêtre :
 - retailler, perte de focus, fermeture, ...
 - ➔ gl4duw...
- Autres interactions possibles (extérieurs) :
 - Divers capteurs, caméra RGB, caméra RGBD, Motion Leap, ...
 - ➔ manuellement dans gl4duwIdleFunc(void(*)(void) **mafondction**) => void **mafondction**(void) { ... }

La parenthèse SwapBuffers (double buffering)

Améliorer la fluidité de l'application graphique



Université Paris 8 - Vincennes à Saint-Denis



Petite introduction à GL4D

Farès Belhadj

Date de MAJ : 4 octobre 2022

email : <mailto:amsi@up8.edu>
github : <https://github.com/noalien/GL4Dummies>
web [?] : <http://api8.fr/GL4D/>

Chapitre 1

Mon **HelloWorld** *GL4Dummies*

Commençons par le premier *Sample Code* (code échantillon) présent dans les sources de la bibliothèque *GL4Dummies*. Il s'agit du projet situé dans `samples/sc_00_00_blank-1.0` que l'utilisateur peut compiler (sous un système Posix¹) en se rendant dans le dossier et en saisissant la commande `make ↵` puis, si tout se déroule correctement, lancer l'exécutable produit en saisissant `./blank ↵`. Ce qui donne :

```
MacBook-Pro-de-Fares:GL4Dummies amsi$ cd samples/sc_00_00_blank-1.0/
MacBook-Pro-de-Fares:sc_00_00_blank-1.0 amsi$ ls
COPYING  Makefile  blank.sln  blank.vcxproj  documentation  window.c
MacBook-Pro-de-Fares:sc_00_00_blank-1.0 amsi$ make
gcc -I . -I/usr/local/include -I/opt/local/include/SDL2 -D_THREAD_SAFE -Wall -O3 -mmacosx-version-min=10.8
-c window.c -o window.o
gcc window.o -lm -L/usr/local/lib -framework OpenGL -mmacosx-version-min=10.8 -lGL4Dummies -L/opt/local/lib
-lSDL2 -o blank
MacBook-Pro-de-Fares:sc_00_00_blank-1.0 amsi$ ./blank
OpenGL version: 4.1 NVIDIA-10.17.5 355.10.05.45f01
Supported shaders version: 4.10
GL4D/gl4du.c (344): Création du programme 1 a l'aide des Shaders :
gl4dp_basic_with_transforms.vs : vertex shader
gl4dp_basic.fs : fragment shader
```

Les utilisateurs de Visual Studio se contenteront d'ouvrir le fichier de la solution `blank.sln` de compiler (F7) et d'exécuter le programme (ctrl-F5).

Ainsi, une exécution réussie du *Sample Code sc_00_00_blank-1.0* ouvre une fenêtre de 320×240 pixels, portant le titre “GL4Dummies’ Hello World” et affichant un fond noir telle qu’illustrée par la figure 1.1. Il est à noter que cette fenêtre possède un bouton permettant sa fermeture, un autre pour la réduire et que le bouton permettant le redimensionnement (s’il est présent) n’est pas actif. L’application quitte (donc la fenêtre se ferme) si l’utilisateur clique sur le bouton fermer. Ses comportements sont programmés et

1. Posix concerne tous les systèmes *Unix-Like* comprenant *Linux*, *FreeBSD*, *Mac OS X* et même l’utilisation de *cygwin* ou *mingw32* sous *Windows*. Par contre, si vous utilisez des IDE tels que *Code::blocks* ou *Visual Studio* voir les solutions présentes dans le dossier *Windows*.

le lecteur les découvrira au fur et à mesure. Enfin, notons que si l'application est lancée depuis un terminal des nombres apparaissent toutes les 5 secondes. Ces nombres correspondent au *Framerate* de l'application ; soit le nombre d'images par seconde qu'elle produit. Il est fréquent que ce nombre corresponde à 60 car classiquement les *drivers* (pilotes) de cartes graphiques bloquent la fréquence de rafraîchissement afin de la synchroniser avec celle de l'écran (télévision ou vidéo-projecteur) qui est souvent à $60Hz$. Enfin, en fonction du système et des pilotes fournis, il existe des outils permettant de désactiver cette **synchronisation verticale**.



FIGURE 1.1 – Résultat du *Sample Code sc_00_00_blank-1.0*

1.1 Le source `window.c` de `sc_00_00_blank-1.0`

Dans ce *Sample Code* le seul fichier strictement nécessaire à la fabrication du programme est `window.c`, il est listé dans le CODE SOURCE 1.1. Pour des raisons de lisibilité, nous avons supprimé les commentaires contenus dans le code téléchargeable et commenterons les principaux éléments de ce programme.

```
#include <GL4D/gl4dp.h>
#include <GL4D/gl4duw_SDL2.h>
int main(int argc, char ** argv) {
    if(!gl4duwCreateWindow( argc, argv, "GL4Dummies'Hello_World",
                           10, 10, 320, 240, GL4DW_SHOWN ) ) {
        return 1;
    }
    gl4dpInitScreen();
    gl4dpClearScreen();
    gl4dpUpdateScreen(NULL);
    gl4duwMainLoop();
    return 0;
}
```

Code source 1.1 – Code source du projet sc_00_00_blank-1.0 : écran noir

Dans les grandes lignes, ce programme créé une fenêtre à l'aide de la fonction `gl4duwCreateWindow`; créé, efface en noir puis met à jour un *screen* respectivement avec les fonctions `gl4dpInitScreen`, `gl4dpClearScreen` et `gl4dpUpdateScreen`; et enfin se lance dans une boucle d'affichage infinie.

Le fichier *header* `gl4dp.h` contient les macros, structures et prototypes de fonctions liées aux primitives de dessin 2D ; la lettre **p** de `gl4dp.h` correspondant à **primitives** mais aussi à **pédagogique** car cette partie de la bibliothèque n'a pas pour vocation de proposer des outils de primitives optimisés mais possède une priorité pédagogique.

Le fichier *header* `gl4duw_SDL2.h` contient pour sa part les macros, structures et prototypes de fonctions liées à la gestion des fenêtres et à la gestion des fonctionnalités qui y sont liées.

1.1.1 La fonction `gl4duwCreateWindow` :

Cette fonction permet de créer une fenêtre pour *GL4Dummies* et initialise la bibliothèque. Son prototype est le suivant :

```
GLboolean gl4duwCreateWindow(int argc, char ** argv,
                            const char * title, int x, int y,
                            int width, int height, Uint32 wflags);
```

les deux premiers arguments proviennent des arguments du `main` et permettent à la bibliothèque d'initialiser son contexte (principalement utile pour connaître les chemin vers le binaire et pouvoir trouver les dépendances – comme les fichiers *shaders* – de manière relative). Le troisième argument est le titre de la fenêtre, il peut aussi servir à retrouver la fenêtre parmi plusieurs autres afin de lui affecter le statut “courante”. `x`, `y`, `width` et `height` donnent la position et dimensions de la fenêtre lors de la création. `wflags` est un entier non signé permettant de stocker des options relatives à la fenêtre comme par exemple : est-elle redimensionnable, visible, ... Nous laisserons le lecteur découvrir plus de détails en le renvoyant sur la documentation de référence. Enfin, cette fonction revoie un booléen afin d'indiquer si la création de la fenêtre a réussi ou échoué. L'échec peut être causé par l'absence de contexte

graphique ou l'impossibilité d'ouvrir un contexte OpenGL® 3.2 (valeur par défaut dans *GL4Dummies*, voir la fonction `gl4duwSetGLAttributes`).

1.1.2 Le *screen* et la fonction `gl4dpInitScreen` :

Le *screen* est un écran virtuel géré par *GL4Dummies* et représentant un tableau de données proportionnel aux dimensions de la fenêtre quand le *screen* est créé à l'aide de la fonction `gl4dpInitScreen` (voir aussi la fonction `gl4dpInitScreenWithDimensions` pour des *screens* aux dimensions différentes de la fenêtre). Ci-après le prototype des fonctions `InitScreen` :

```
GLuint gl4dpInitScreen(void);  
GLuint gl4dpInitScreenWithDimensions(GLuint w, GLuint h);
```

Ces fonctions renvoient donc un identifiant de *screen* permettant, en cas de multiples *screens*, de passer de l'un à l'autre, afin d'effacer, dessiner, supprimer, ... Par ailleurs, la structure intrinsèque à un *screen* est simple, elle contient autant d'entiers non signés que de pixels. Pour faire simple, soit pour un *screen* de dimensions $w \times h$ la partie données du *screen* est comparable à un :

```
unsigned int data[w * h];
```

Nous invitons le lecteur à se rendre à la section 4 pour en savoir un peu plus sur ce qu'est un pixel, une de ses représentations simples adoptées dans la partie "pédagogique" (`gl4dp.h`) de *GL4Dummies*, et comment faire son premier programme créant un dégradé de pixels.

1.1.3 La fonction `gl4dpClearScreen` :

Cette fonction efface le *screen* en cours en y mettant des zéros (0). En pratique, cette fonction utilise un `memset` avec comme second argument `0` afin de remplir tous les octets de zéros ce qui produit un écran noir (les pixels contenant l'intensité de chaque couleur, l'omniprésence de zéros produit une image à intensité nulle donc noire). Pour un effacement avec une autre couleur voir la fonction `gl4dpClearScreenWith(Uint32 color)`.

1.1.4 La fonction `gl4dpUpdateScreen` :

Sans rentrer dans les détails, cette fonction permet de communiquer entre la mémoire centrale, la *RAM-CPU*, et la mémoire graphique, la *RAM-GPU*. Les données sont partiellement ou dans sa globalité transférée vers OpenGL® pour que son état actuel en *RAM-CPU* soit reproduit à l'écran. Ci-après le prototype de la fonction :

```
void gl4dpUpdateScreen(GLint * rect);
```

où `rect` représente le rectangle, sous partie du *screen*, à mettre à jour côté *GPU*. C'est le pointeur vers les quatre entiers positifs `x, y` – coin haut gauche du rectangle – et `w, h` – ses dimensions.

1.1.5 La fonction `gl4duwMainLoop` :

Enfin, nous revenons vers une fonction liée à la gestion du fenêtrage. Une application fenêtrée est un processus qui demeure *alive* (en quelques sortes “vivant”) au sein du système tant qu'il ne reçoit pas d'événement ou de signal lui indiquant ou le forçant à quitter. Donc, par définition, dans ce type d'applications il est nécessaire de produire une boucle infinie² dont le corps est en attente d'un signal de sortie. Afin de ne pas accaparer les ressources du système cette boucle doit ordonner les tâches à faire pour permettre à la fois d'écouter les événements, réagir en fonction et procéder à l'affichage. C'est le rôle de la fonction `gl4duwMainLoop` dont nous abordons le principe générale dans la section 2 ci-après.

2. À moins d'utiliser une sorte variable globale servant de test de boucle et passant à faux par le biais d'un événement extérieur.

Chapitre 2

Principe de fonctionnement d'une application graphique

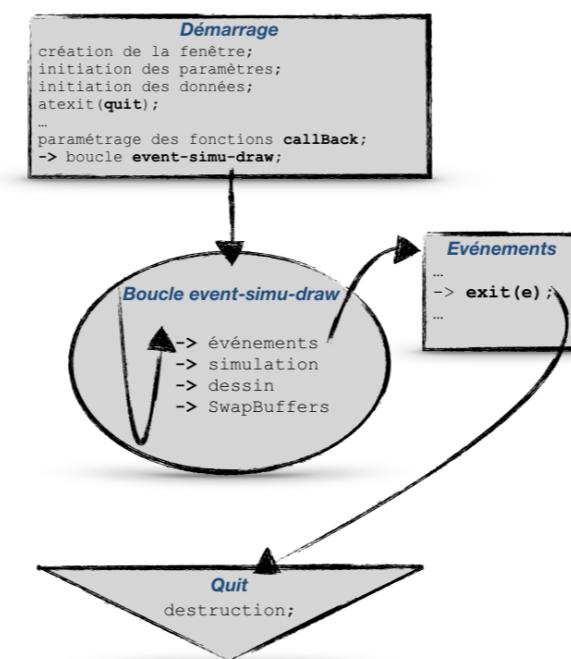


FIGURE 2.1 – Principe général du fonctionnement d'une application graphique avec la boucle événements–simulation–dessin

Le schéma donné Figure 2.1 illustre les grandes lignes du fonctionnement

d'une application graphique. Une application graphique passe par la création d'au moins une fenêtre contenant un contexte graphique dans lequel des éléments graphiques sont dessinés ; il arrive qu'on interagisse avec ces derniers. Dans notre cas, il s'agit de créer une fenêtre donnant accès à un contexte OpenGL® dans lequel la bibliothèque (ses paramètres, ses données et ses fonctions) fonctionne.

Avant d'arriver à la boucle principale d'affichage, il est généralement utile de paramétrier les options de l'API graphique, parfois ses propres données initiales ainsi que les fonctions *callBack*. Concernant ces dernières, la boucle d'affichage est une boucle infinie dans laquelle entre le programme et où il passe son temps à gérer les événements avec l'utilisateur (via le clavier, la souris, ...), met éventuellement à jour les données puis redessine et rafraîchit l'affichage. Ceci peut ressembler à un code tel que :

```
for(;;) { /* le programme entre dans une boucle infinie */
    /* 1) s'il y a une fonction de gestion d'événements, l'appeler */
    /* 2) s'il y a une fonction pour MAJ des données, l'appeler */
    /* 3) s'il y a une fonction de dessin, l'appeler */
    /* 4) rafraîchir la fenêtre (SwapBuffers) */
}
/* ce point de code n'est jamais atteint */
/* on ne quitte la boucle infinie que par le biais d'un exit */
/* d'où l'importance du atexit pour ajouter d'éventuelles fonctions à exécuter au moment du exit */
```

Les fonctions citées aux points (1), (2) et (3) peuvent être implémentées comme des fonctions dites de *callBack*¹ afin qu'elles soient appelées par la boucle principale d'affichage si elle prend la forme d'une fonction comme dans notre cas actuel, c'est à dire **gl4duwMainLoop()**. Ainsi, les principales fonctions de *callBack* peuvent être *settées*² par le biais de fonctions telles que :

gl4duwKeyDownFunc permet de *setter* la fonction à appeler en cas d'événement clavier *key down* ou “touche enfoncee” ;

gl4duwKeyUpFunc permet de *setter* la fonction à appeler en cas d'événement clavier *key up* ou “touche relâchée” ;

gl4duwResizeFunc permet de *setter* la fonction à appeler en cas de redimensionnement, par l'utilisateur, de la fenêtre si cette dernière est redimensionnable ;

gl4duwMouseFunc permet de *setter* la fonction à appeler en cas de click de souris dans la fenêtre ;

1. Généralement, une fonction *callBack* est une fonction passée en argument d'une autre fonction pour que cette dernière l'appelle si besoin. Nous pouvons aussi utiliser des variables “pointeurs de fonction” ainsi que des *Setters* (type de fonction qui a pour but de “set” une variable/propriété) pour les modifier.

2. Désolé pour cet anglicisme, mais je ne trouve pas de meilleur mot que *set* pour indiquer l'action de mettre une valeur à une variable/propriété.

Principe de fonctionnement d'une application graphique

gl4duwIdleFunc permet de *setter* la fonction à appeler afin de mettre régulièrement (à chaque frame) à jour les données. Cette mise à jour correspond à une action souvent appelée “simulation” car elle sert à mettre à jour les données liées au(x) modèle(s) simulé(s) par l'application graphique ;

gl4duwDisplayFunc permet de *setter* la fonction à appeler afin de mettre à jour le “dessin” de chaque frame. C'est la fonction qui dessine sur le *buffer* courant ;

gl4duwMotionFunc permet de *setter* la fonction à appeler lorsque l'utilisateur déplace la souris dans la fenêtre tout en ayant un ou plusieurs boutons enfoncés ;

gl4duwMotionFunc permet de *setter* la fonction à appeler lorsque l'utilisateur déplace la souris dans la fenêtre sans qu'aucun bouton ne soit enfoncé.

Chapitre 3

Le HelloWorld multi-coloré

Dans le premier exemple d'application graphique (cf. CODE SOURCE 1.1) le programme créé une fenêtre, puis un écran aux dimensions de la fenêtre (*i.e. screen*), efface l'écran en noir et le met à jour. À partir de là, le programme entre dans une sa boucle – infinie – d'affichage sans avoir, au préalable, *setté* de fonction *callBack*. Ainsi, le résultat reste statique (rien de nouveau ne se passe) jusqu'à ce que l'utilisateur décide de fermer la fenêtre.

Dans le CODE SOURCE 3.1 deux nouvelles fonctions font leur apparition : **quitte** et **dessine**. Nous pouvons remarquer que ces deux fonctions ne sont JAMAIS explicitement appelées, elles sont néanmoins présentent comme argument respectifs des fonctions **atexit** et **gl4duwDisplayFunc**, ce qui en fait des fonctions de *callBack*.

```
#include <GL4D/gl4dp.h>
#include <GL4D/gl4duw_SDL2.h>

static void quitte(void) {
    gl4duClean(GL4DU_ALL);
}

static void dessine(void) {
    GLubyte r = rand() % 256, g = rand() % 256, b = rand() % 256;
    gl4dpClearScreenWith(RGB(r, g, b));
    gl4dpUpdateScreen(NULL);
}

int main(int argc, char ** argv) {
    if(!gl4duwCreateWindow(argc, argv, "GL4Dummies'_Hello_World",
        10, 10, 320, 240, GL4DW_SHOWN) ) {
        return 1;
    }
    gl4dpInitScreen();
    atexit(quitte);
    gl4duwDisplayFunc(dessine);
    gl4duwMainLoop();
    return 0;
}
```

Code source 3.1 – Code source du projet sc_00_01_multicolorClear-1.0 : écran coloré changeant à chaque *Frame*

La fonction `atexit` dont voici un extrait du manuel (`man atexit`) :

NAME
atexit -- register a function to be called on exit

SYNOPSIS

```
#include <stdlib.h>

int
atexit(void (*function)(void));

int
atexit_b(void (^function)(void));
```

DESCRIPTION
The atexit() function registers the given function to be called at program exit, whether via exit(3) or via return from the program's main(). Functions so registered are called in reverse order; no arguments are passed.

If the provided function is located in a library that has been dynamically loaded (e.g. by dlopen()), it will be called when the library is unloaded (due to a call to dclose()) or at program exit.

The atexit_b() function is like atexit() except the callback is a block pointer instead of a function pointer.

Note: The Block_copy() function (defined in <Blocks.h>) is used by atexit_b() to make a copy of the block, especially for the case when a stack-based block might go out of scope when the subroutine returns.

These functions must not call exit(); if it should be necessary to terminate the process while in such a function, the _exit(2) function should be used. (Alternatively, the function may cause abnormal process termination, for example by calling abort(3).)

At least 32 functions can always be registered, and more are allowed as long as sufficient memory can be allocated.

RETURN VALUES
The atexit() and atexit_b() functions return the value 0 if successful; otherwise the value -1 is returned and the global variable errno is set to indicate the error.

ERRORS
[ENOMEM] No memory was available to add the function to the list. The existing list of functions is unmodified.

SEE ALSO
exit(3)

STANDARDS
The atexit() function conforms to ISO/IEC 9899:1990 ("ISO C90").

BSD September 6, 2002 **BSD**
est une fonction système qui permet d'empiler des callbacks à (de la dernière empilée à la première) suite à un exit rencontré où dans le code ou à un return au sein du main. Ainsi la fonction

BSD September 6, 2002 BSD
est une fonction système qui permet d'empiler des *callBacks* à exécuter (de la dernière empilée à la première) suite à un **exit** rencontré n'importe où dans le code, ou à un **return** au sein du **main**. Ainsi la fonction **quitte** n'est appelée qu'au moment où l'utilisateur ferme la fenêtre de l'application ; ceci peut être vérifié en ajoutant un *print* dans le corps de la fonction ou en ajoutant un *breakpoint* et en exécutant le programme en mode *Debug*. Elle contient l'instruction `gl4duClean(GL4DU_ALL)` ; qui appelle la fonction *GL4Dummies* libérant les ressources utilisées par la bibliothèque.

Concernant la fonction `dessine`, il faut comprendre que la conséquence de l'appel `gl4duwDisplayFunc(dessine)`; l'affecte comme fonction *callBack* à appeler à chaque fois que la boucle principale d'affichage (représentée par le `gl4duwMainLoop();`) dessine une nouvelle *frame*. Elle a pour objectif de sélectionner aléatoirement¹ une intensité entre 0 et 255 (à cause du modulo

1. En pratique, la fonction `rand` renvoie une série de nombres pseudo-aléatoires compris entre 0 et `RAND_MAX`. Voir le manuel pour plus de détails.

“%” 256) et l'affecte à l'octet non signé (*i.e.* `GLubyte`) `r` – pour *red* – puis `g` – pour *green* – et enfin `b` – pour *blue*. Le triplet (`r`, `g`, `b`) est transformé en entier non signé 32 bits représentant la couleur RGB qui correspond à ces trois intensités (synthèse additive de couleurs). C'est la macro `RGB` qui réalise le codage sur 32 bits de la couleur, sa définition se trouve dans le fichier `gl4dp.h` et nous donnerons plus de détails sur cette représentation dans la section suivante.

Ainsi cette couleur est transmise à la fonction `gl4dpClearScreenWith` qui s'en sert pour effacer l'écran qui n'est rien de plus qu'une zone mémoire de $320 \times 240 = 76800$ entiers 32 bits en RAM-CPU². Pour que cet “écran” s'affiche, une **synchronisation** est nécessaire avec la mémoire GPU et ceci est fait à l'aide de l'appel `gl4dpUpdateScreen(NULL)`; qui transfert l'ensemble³ de la mémoire-écran depuis la RAM-CPU vers la RAM-GPU.

2. Pour faire simple, la RAM-CPU est la mémoire vive utilisée par le processeur central de la machine. Cette distinction est faite par opposition à la RAM-GPU qui est la mémoire vive appartenant et utilisée par la carte graphique de la machine.

3. Le paramètre qui ici vaut `NULL` indique que nous ne souhaitons pas transférer une sous-partie de la zone mémoire mais bien l'ensemble de cette zone. Quand ce paramètre est un pointeur valide vers un rectangle (pointeur vers quatre entiers consécutifs représentant respectivement (x, y) et (w, h) coin supérieur-gauche et largeur-hauteur du rectangle) seul la zone délimitée par ce rectangle est mise à jour depuis la RAM-CPU vers la RAM-GPU. Ceci peut-être utile quand peu de zones changent à l'écran, réduisant ainsi les transferts CPU-GPU. Ce cas se présente par exemple lors d'un usage optimal de *sprites* qui se déplacent à l'écran.

Chapitre 4

Le HelloPixels avec *GL4Dummies*

Cette section a pour premier objectif de donner quelques notions de base de la représentation des pixels, donc des couleurs, en mémoire. Elle se termine par un exemple simple de création d'un dégradé de niveaux de gris appliqué à une représentation RGBA 32 bits.

4.1 Modèles de représentation des couleurs en mémoire

Sans être exhaustifs, nous reprenons dans la Figure 4.1 les modèles classiques de représentation d'un pixel¹ en mémoire. Le quasi point commun entre tous ces modèles est que chaque élément représente une quantité numérique qui indique un degré d'intensité. Cela peut être l'intensité de blanc, donc le niveau de gris, ou en séparant la lumière en somme de trois composantes primaires (le rouge, le vert et le bleu), l'intensité de chaque composante. Le seul modèle présenté ici et qui déroge à la règle est le modèles de couleurs indexées (le troisième de la Figure 4.1 en partant du haut). En effet, dans ce cas la valeur indiquée dans un pixel ne correspond pas à une intensité mais à l'indice, dans un autre tableau appelé LUT (pour *Look Up Table*), auquel nous trouvons la ou les intensités correspondant à ce numéro de couleur ; une analogie peut être faite avec le jeu pour enfants *draw by numbers*.

1. Pixel est la contraction de *picture* et *element*, il est donc la brique de base compasant une image numérique sur un format matriciel, c'est-à-dire sous la forme d'une grille.

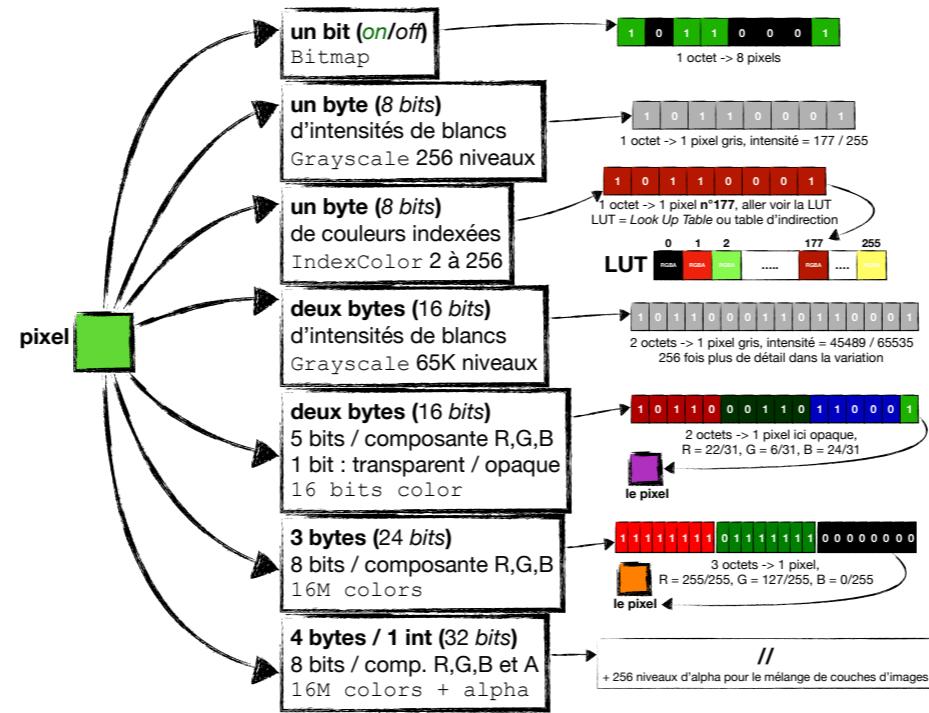


FIGURE 4.1 – Quelques modèles classiques de représentation d'un pixel (ou couleur) en mémoire.

Donnons quelques détails sur les différents modèles présentés Figure 4.1 :

Bitmap : est en un sens l'ancêtre du *PixMap* (abréviation de *pixel map*). Ce modèle est donc une *map* (carte ou simplement tableau qui cartographie une image matricielle) de bits indiquant à chaque position si cette dernière est allumée – *on* ou éteinte – *off* ; ce qui était suffisant pour les premiers modèles d'écrans qui étaient monochromes. Côté mémoire, c'est donc une suite d'octets (ou *bytes*) où chacun représente 8 points consécutifs, de gauche à droite, à l'écran. Dès que nous arrivons à la fin d'une ligne, les octets suivants représentent la ligne d'après (celle du bas), puis ainsi de suite jusqu'à arriver à la fin de l'image (généralement en bas à droite) ;

GrayScale-256 : Appelée aussi image en niveaux de gris. Ce modèle cherche à coder l'intensité de blanc en chaque pixel, ce qui peut aussi se rapporter à la notion de luminance de chaque pixel. C'est donc

un modèle simple où l'octet stocke une valeur numérique de 0 à 255 correspondant à une intensité par pixel – 0 intensité minimale ou noir, 255 intensité maximale ou blanc.

IndexColor : Ce modèle peut être généralisé au codage indexé pour un nombre de couleurs quelconques. Prenons le cas le plus simple, chaque octet est un pixel, et la valeur dans l'octet n'indique aucunement une intensité mais plutôt un indice dans un “autre” tableau contenant 256 éléments où chacun est une couleur codée en plus d'un octet, comme par exemple en “vraies” couleurs 24 bits ou 16 millions de couleurs (sinon ça n'aurait aucun intérêt). Un format répandu utilisant ce modèle est le *GIF*.

GrayScale-65536 : Est un modèle équivalent au *GrayScale-8-bits* sauf que les nuances de gris sont codées sur 16 bits, soit 2 octets, pour plus de finesse² dans les variations d'intensité ;

16-bits-colors : Ce modèle tend à être de moins en moins répandu. Il est souvent lié à une limitation matérielle (couleurs gérées par l'écran et quantité mémoire) que nous pouvions retrouver sur les premiers modèles de *Smatphones*, des *devices* portables tels que la *Nintendo DS* ou même encore certains modèles “haut de gamme” de micro-ordinateurs³ ou parmi les premières générations de consoles de jeux. Les 16 bits étaient exploités de la manière suivante : 5 bits pour le rouge, 5 bits pour le vert, 5 bits pour le bleu et 1 bit pour indiquer si le pixel est transparent ou opaque (très utile pour utiliser des *sprites* dont les frontières ne sont pas nécessairement rectangulaires). Ce format permet donc d'avoir 32 nuances d'intensité par composante pour un total de 32768 couleurs.

2. Cette finesse dans la variation n'est pas forcément nécessaire pour la perception humaine. Par le passé, il y avait même certaines études qui annonçaient que 32 à 64 niveaux de gris étaient des bonnes bornes supérieures pour un rendu sur un dispositif d'affichage (*i.e.* écran), qu'au delà l'humain ne distinguait pas les variations. Ceci est relativement faux, car les dispositifs d'affichage se sont bien améliorés depuis et 256 niveaux de gris est plutôt une borne inférieure raisonnable. En pratique, nous percevons mieux les variations dans les tons sombres et une échelle linéaire sur les 256 nuances montre des défauts liés au manque de nuances. Par ailleurs, une image n'a pas nécessairement pour vocation d'être directement “vue” par l'œil humain, elle peut contenir plus de détails que ce que nous pouvons percevoir – prenons pour exemple des captures d'imagerie médicale ou astronomique – et ces détails peuvent être révélés par le biais de filtres de traitement d'images.

3. L'avènement des micro-ordinateur coïncide avec l'époque où la grande majorité des machines grand-public étaient 8-bits. Les modèles 16-bits faisaient donc partie du haut de gamme de l'époque.

16 millions de couleurs : Ce modèle et le suivant sont les plus classiques, même si, avec le temps, ils laisseront probablement la place à des modèles plus “riches” en terme bit par composante – donc de nuances par composante. Ici, le pixel comprend un octet par composante, donc 8 bits pour le rouge, 8 bits pour le vert et 8 bits pour le bleu pour un total de 24 bits soit $2^{24} = 16M$ de couleurs.

RGBA sur 32 bits : Le format précédent était idéal pour la majorité des dispositifs d'affichage qui permettaient l'affichage de 16 millions de couleurs. Par contre, stocker les pixels par lots de 3 octets n'est pas optimal par rapport aux architectures machines. En effet, sur une architecture 32-bits, il est plus optimal de parcourir ou de réaliser une opération sur un `int` 32-bits que d'en faire 3 ou 4 sur 3 ou 4 octets. Ainsi, il est préférable de stocker les 24 bits du format RGB ci-dessus dans un `int` 32-bits. Et étant donné qu'il reste 8 bits de libres, autant les utiliser ; d'où la composante alpha qui est rajoutée et qui permet de stocker 256 niveaux d'opacité par pixel. Ceci donne la possibilité d'avoir plusieurs “couches” (ou *layers*) par image, chacune étant elle-même une image et l'ensemble représentant une composition formée de ces “couches”.

Remarque : Il est possible d'utiliser plus d'un octet par composante, comme par exemple utiliser un flottant pour chaque composante, soit 32 bits par composante et un total de 128 bits par pixel. C'est notamment un cas facilement géré par les GPUs car ces dernières ont pour format natif le `vec4` qui est un vecteur de 4 flottants, et les bus de communication entre CPU et GPU ont actuellement une largeur au moins égale à 128 bits.

4.2 Cas simplifié de la sous-bibliothèque `gl4dp`

Un sous-ensemble de la bibliothèque *GL4Dummies* sert à se familiariser avec la manipulation d'images dans un format simple avant d'aborder des formes plus générales de représentation ou encore aborder des notions liées à OpenGL® et aux *Shaders*. Ce sous-ensemble de fonctions et de macros est regroupé dans le fichier `gl4dp.h` (et par extension dans le fichier source `gl4dp.c` mais ce dernier concerne l'implémentation et n'est pas utile pour la compréhension des éléments à venir).

Dans `gl4dp`, nous avons fait le choix de n'avoir qu'un seul modèle de représentation de couleurs. Une couleur est donc un entier non-signé 32 bits,

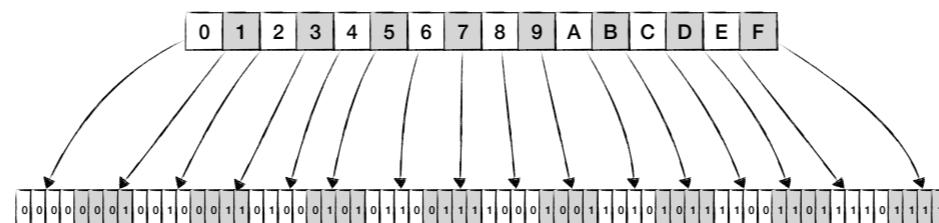
et un **screen** est une image, donc un tableau composé de plusieurs couleurs – autant que la largeur \times la hauteur de l'image – que nous pouvons afficher à la fenêtre. Un **screen** peut avoir des dimensions différentes de celles de la fenêtre.

Le format choisi est donc le RGBA 32-bits, ce qui signifie que chaque case de l'image (ou du tableau) est un entier non signé de 32 bits. Ainsi dans le code suivant :

```
Uint32 img[] = {
    0xFF0000FF, 0x00FF00FF, 0x0000FFFF,
    0xFFFF00FF, 0x00FFFFFF, 0xFF00FFFFFF
};
```

nous pouvons imaginer que **img** est une image de 3×2 pixels dont les couleurs sont respectivement (de gauche à droite et de haut en bas) : rouge, vert, bleu, jaune, cyan et magenta. Pour information, le type **Uint32** est lié à *GL4Dummies* et aussi à *SDL2* et garantit la création d'un entier non signé (**U** ou **Unsigned**) dont la taille en bits est de 32.

Pour rappel, en hexadécimal un chiffre (0, 1, 2, ..., 9, A, B, C, D, E, F) oui il y en a 16) donne lieu à 16 possibilités. En faisant l'analogie avec le binaire, il faut avoir 4 chiffres (ou cases) binaires pour pouvoir représenter 16 possibilités, car $2^4 = 16$. Le schéma ci-après donne cette première correspondance :



La Figure 4.2 donne une schématisation d'une représentation mentale et mémoire correspondant au code de la *data img* donnée ci-dessus. En pratique dans ce cas nous avons ajouté une ligne telle que :

```
Uint8* p = (Uint8*)img;
```

Ici **Uint8** est un *byte* (donc un octet et aussi 8 bits) non signé. Donc **p** est un pointeur vers **img** mais à la différence de cette dernière **p** la voit comme une succession d'octets et non d'entiers 32-bits. Par exemple, **p[0]**, **p[1]**, **p[2]** et **p[3]** sont les quatre octets qui, concaténés, représentent **img[0]**.

Maintenant si nous souhaitons afficher, à l'aide d'un *print*, la valeur hexadécimale de, par exemple, **img[3]** nous pouvons le faire directement via **img[3]**

Le HelloPixels avec GL4Dummies

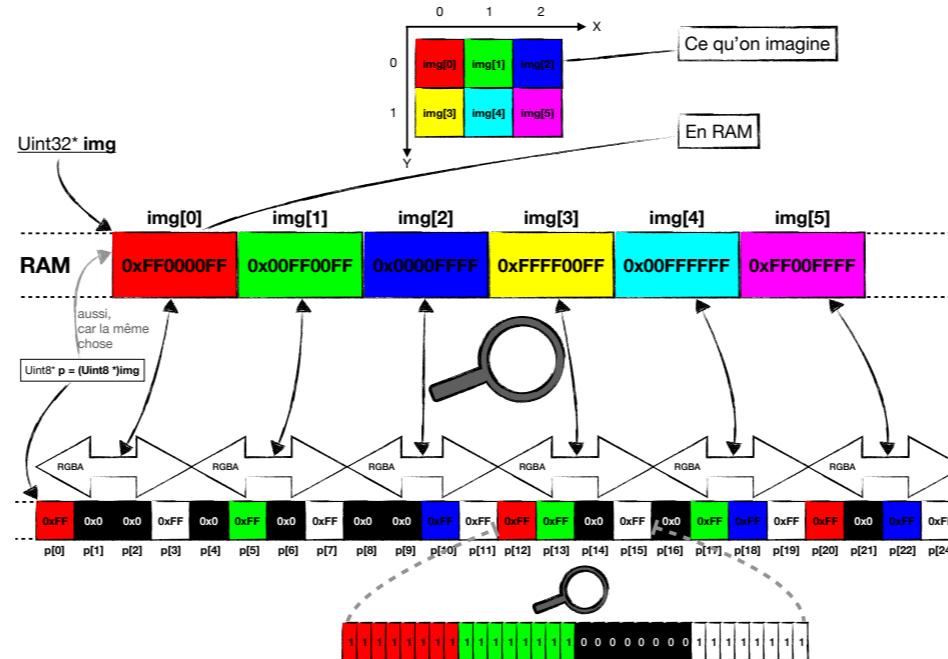


FIGURE 4.2 – ZOOMS sur la mémoire d'une image RGBA-32-bits de 3×2 . Attention ici le *Endianness* n'est pas traité, nous considérons que nous sommes dans une configuration *Big-endian*.

ou indirectement via $p[12]$, $p[13]$, $p[14]$ et $p[15]$. Voici un bout de code⁴
– CODE SOURCE 4.1 – qui le fait pour notre exemple :

```
#include <stdio.h> /* pour la fonction printf */
#include <stdint.h> /* pour les types standards uint32_t (comme Uint32) et uint8_t (comme Uint8)
/* n'hésitez pas à m'avertir si cet include n'est pas présent dans vos bibliothèques
standards */

int main(void) {
    uint32_t img[] = {
        0xFF0000FF, 0x00FF00FF, 0x0000FFFF,
        0xFFFF00FF, 0x00FFFFFF, 0xFF00FFFF
    };
    uint8_t * p = (uint8_t *)img;
    printf("img[3]_->_%08x\n", img[3]);
    printf("concatenation_de_p[12],_p[13],_p[14]_et_p[15]_->_%02x%02x%02x%02x\n", p[12], p[13], p[14], p[15]);
    printf("Alors_?_quel_Boutisme_(ou_Endianness)_?\n");
    return 0;
}
```

Code source 4.1 – Récupérer de deux manières la même couleur.

4. Ce code est plus ou moins complet car l'ordre n'est pas toujours respecté selon que l'architecture machine de la machine l'exécutant est *Little-endian* ou *Big-endian*.

Or, l'exécution de ce code sur notre machine équipée d'un processeur *Intel i7* ne donne pas réellement le résultat escompté. Voici la trace :

```
$ gcc -Wall code_rgba_zoom_memoire.c && ./a.out
img[3] -> fffff0ff
concatenation de p[12], p[13], p[14] et p[15] -> ff00ffff
Alors ? quel Boutisme (ou Endianness) ?
```

Nous remarquons que la concaténation des quatre octets a donné un résultat correspondant au miroir du nombre stocké dans l'entier ; comme si au lieu d'avoir RGBA nous avons produit ABGR (vous pouvez vous en assurer en utilisant une autre valeur pour `img[3]` telle que `0x1A2B3C4D`). Ceci est dû au fait que le *Boutisme*⁵ ou *Endianness*⁶ de notre architecture *Intel* est de type *Little-endian*, c'est-à-dire que les octets d'un entier sont placés de gauche à droite du plus faible au plus fort, ce qui est contre-intuitif car inverse à l'ordre des bits dans un octet où le bit de poids fort est à gauche, et en allant vers la droite, nous rencontrons des bits de poids plus en plus faibles jusqu'au bit de faible, complètement à droite. Aussi, lors d'un cours d'informatique, l'enseignant au tableau a plus tendance à naturellement utiliser la représentation *Big-endian*.

La Figure 4.3 résume la problématique liée à gestion de la différence entre une architecture *Big-endian* ou *Little-endian* ; ici nous l'illustrons avec le besoin d'extraire la composante verte d'une couleur mais le problème se présente aussi lors de l'écriture ou la modification d'une composante (plus généralement la composition d'une couleur à l'aide de ses composantes).

Ainsi, à titre informatif nous donnons dans le CODE SOURCE 4.2 comment peuvent être définies des fonctions ou (ici) des macros pour pouvoir lire ou écrire une composante dans une couleur. Ce code prend automatiquement en compte la détection du type d'architecture en terme d'*Endianness*, pour cela il utilise la macro `SDL_BYTEORDER` fournie de manière homogène et multiplateforme par la bibliothèque *SDL2*.

5. Personnellement, je trouve que le terme choisi en français est ridicule. Ca ne doit pas vous empêcher de jeter un œil à la doc :
<https://fr.wikipedia.org/wiki/Boutisme>

6. C'est plus joli en anglais :)
<https://en.wikipedia.org/w/index.php?title=Endianness>

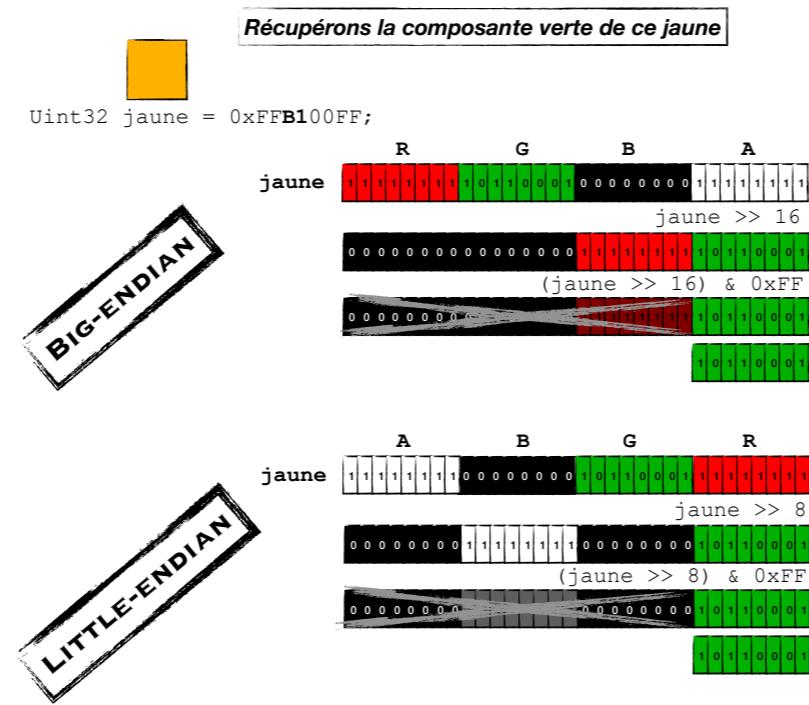


FIGURE 4.3 – Exemple d'extraction d'une composante (ici la verte) selon l'Endianness de l'architecture.

```

#ifndef SDL_BYTEORDER == SDL_BIGENDIAN
/*!\brief décalage pour la composante rouge */
#define R_SHIFT 24
/*!\brief décalage pour la composante verte */
#define G_SHIFT 16
/*!\brief décalage pour la composante bleue */
#define B_SHIFT 8
/*!\brief décalage pour la composante alpha */
#define A_SHIFT 0
/*!\brief masque pour la composante rouge */
#define R_MASK 0xff000000
/*!\brief masque pour la composante verte */
#define G_MASK 0x00ff0000
/*!\brief masque pour la composante bleue */
#define B_MASK 0x0000ff00
/*!\brief masque pour la composante alpha */
#define A_MASK 0x000000ff
#else
/*!\brief décalage pour la composante rouge */
#define R_SHIFT 0
/*!\brief décalage pour la composante verte */
#define G_SHIFT 8
/*!\brief décalage pour la composante bleue */
#define B_SHIFT 16
/*!\brief décalage pour la composante alpha */
#define A_SHIFT 24
/*!\brief masque pour la composante rouge */
#define R_MASK 0x000000ff
/*!\brief masque pour la composante verte */
#define G_MASK 0x0000ff00
/*!\brief masque pour la composante bleue */
#define B_MASK 0x00ff0000
/*!\brief masque pour la composante alpha */
#define A_MASK 0xff000000
#endif
/*!\brief macro qui convertie un r, un g, un b et un a en couleur Uint32 rgba */
#define RGBA(r, g, b, a) (((Uint32)(uint8)(r)) << R_SHIFT | \
((Uint32)(uint8)(g)) << G_SHIFT | \
((Uint32)(uint8)(b)) << B_SHIFT | \
((Uint32)(uint8)(a)) << A_SHIFT )
/*!\brief macro qui convertie un r, un g et un b en couleur Uint32 rgba dont l'alpha est 'a' 255 */
#ifndef RGB //FOR MSVC
#define RGB
#endif
#define RGB(r, g, b) RGBA(r, g, b, 255)
/*!\brief macro permettant d'extraire une composante en utilisant l'argument \a shift */
#define EXTRACT_COMP(coul, shift) (((Uint32)(coul)) >> (shift)) & 0xFF
/*!\brief macro permettant d'extraire une composante rouge de l'Uint32 \a coul */
#define RED(coul) EXTRACT_COMP(coul, R_SHIFT)
/*!\brief macro permettant d'extraire une composante verte de l'Uint32 \a coul */
#define GREEN(coul) EXTRACT_COMP(coul, G_SHIFT)
/*!\brief macro permettant d'extraire une composante bleue de l'Uint32 \a coul */
#define BLUE(coul) EXTRACT_COMP(coul, B_SHIFT)
/*!\brief macro permettant d'extraire une composante alpha de l'Uint32 \a coul */
#define ALPHA(coul) EXTRACT_COMP(coul, A_SHIFT)

```

Code source 4.2 – Macros pour la composition d'une couleur et l'extraction de ses composantes.

4.3 Faire un dégradé de niveaux de gris et jouer avec les couleurs

Partant des éléments vus précédemment, le CODE SOURCE 4.3⁷ montre comment réaliser un dégradé de niveaux de gris dans un *screen* de 16×16 pixels mappé sur une fenêtre de 320×320 pixels.

7. L'ensemble de cet exemple est téléchargeable à partir de l'adresse :
https://github.com/noalien/GL4Dummies/tree/master/samples/sc_00_01_gradient-1.0

```
#include <GL4D/gl4dp.h>
#include <GL4D/gl4duw	SDL2.h>

static void quitte(void) {
    gl4duClean(GL4DU_ALL);
}

static void dessin(void) {
    int x, y;
    Uint32 c;
    for(y = 0; y < 16; y++) {
        for(x = 0; x < 16; x++) {
            c = y * 16 + x;
            c = c | (c << 8) | (c << 16) | (c << 24);
            gl4dpSetColor(c);
            gl4dpPutPixel(x, y);
        }
        gl4dpUpdateScreen(NULL);
    }
}

int main(int argc, char ** argv) {
    if(!gl4duwCreateWindow(argc, argv, "GL4Dummies'_Hello_Pixels",
                           10, 10, 320, 320, GL4DW_SHOWN)) {
        return 1;
    }
    gl4dpInitScreenWithDimensions(16, 16);
    atexit(quitte);
    gl4duwDisplayFunc(dessin);
    gl4duwMainLoop();
    return 0;
}
```

Code source 4.3 – Code source du projet sc_00_01_gradient-1.0 : dégradé de niveaux de gris sur un *screen* de résolution inférieure à celle de la fenêtre.

Même si cela est difficilement distinguable à l'œil (zomez sur la fenêtre illustrée en Figure 4.4 pour le voir), nous pouvons constater que l'intensité du dégradé dépend de la coordonnée *x* et *y* du pixel dans le *screen*.

La ligne : *c* = *y* * 16 + *x*; calcule cette intensité de manière à avoir une valeur commençant à zéro, en bas à gauche, et augmentant de 1 de gauche à droite et de 16 d'une ligne à l'autre de bas en haut, arrivant au final à 255 qui représente l'intensité maximale sur une composante codée sur un octet.

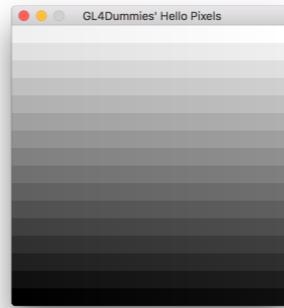


FIGURE 4.4 – Résultat de l'exécution de l'exemple sc_00_01_gradient-1.0 effectuant un dégradé de niveaux de gris sur un *screen* de résolution inférieure à celle de la fenêtre.

Ainsi, sur une architecture *little-endian* (et aussi pour vérifier que dans ce cas le rouge est à droite, le vert est au milieu-droit, le bleu est au milieu-gauche et l'alpha est à gauche), en remplaçant la ligne d'après :

```
c = c | (c << 8) | (c << 16) | (c << 24);
```

par, respectivement :

1. `c = (255 << 24) | c;`
2. `c = (255 << 24) | (c << 8);`
3. `c = (255 << 24) | (c << 16);`
4. `c = (255 << 24) | ((rand()&255) << 16) | ((rand()&255) << 8) | (rand()&255);`

nous obtenons les résultats, illustrés de gauche à droite, de la Figure 4.5. Vous remarquerez que le dernier cas produit une palette différente de couleurs à chaque frame, ceci est dû à la “continuité” de génération de nombres pseudo-aléatoires à chaque fois que la fonction `dessin` est appelée.

Pour finir, nous proposons de modifier légèrement la fonction `dessin` afin d'introduire une intensité initiale qui modifie l'ensemble des pixels. Nous l'introduisons sous la forme d'une variable locale *statique* – le *qualifier static* est ajouté avant le type de la variable – ce qui a pour conséquence que cette variable garde sa précédente valeur d'un appel à l'autre de la fonction⁸.

8. En quelque sorte, une variable locale statique réside en mémoire dans la même zone que la fonction qui la contient, son adresse ne change pas d'un appel à l'autre ce qui

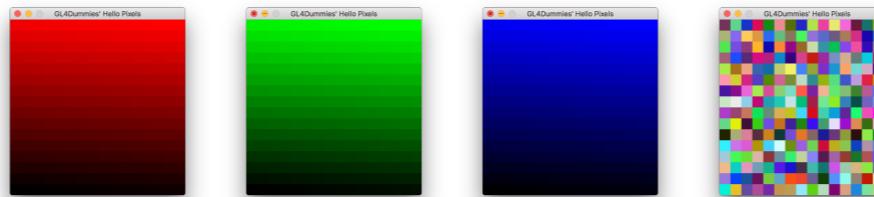


FIGURE 4.5 – Résultats de l'exécution de l'exemple `sc_00_01_gradient-1.0` après modification de la ligne composant la couleur utilisée par le `putPixel`.

Comme nous augmentons la valeur de cette variable à la fin de la fonction, l'intensité initiale augmente d'autant. Afin d'éviter de déborder de la valeur maximale par composante (*i.e.* 255) nous bornons `c` à l'aide d'un `& 0xFF` ce qui équivaut à un modulo 256. Voici le code modifié :

```
static void dessin(void) {
    int x, y;
    Uint32 c;
    static Uint32 intensite_initiale = 0;
    for(y = 0; y < 16; y++) {
        for(x = 0; x < 16; x++) {
            /* pour chacun des 256 pixels, calculer l'intensité */
            c = (intensite_initiale + y * 16 + x) & 0xFF; /* pareil que & 255 */
            /* appliquer cette intensité sur chacune des composantes rouge,
             * vert, bleu et alpha du pixel */
            c = (255 << 24) | (c << 16) | (c << 8) | c;
            /* met cette couleur comme couleur de dessin courante */
            gl4dpSetColor(c);
            /* affecte le pixel x,y avec la couleur en cours */
            gl4dpPutPixel(x, y);
        }
        gl4dpUpdateScreen(NULL);
        ++intensite_initiale;
    }
}
```

Nous laissons le lecteur effectuer la modification et constater le résultat obtenu.

4.4 Exercices

1. Partant du code `samples/sc_00_01_multicolorClear-1.0`, faire en sorte d'obtenir le même résultat visuel que celui de la Figure 4.6. Pour se faire, vous pouvez utiliser la fonction `gl4dpLine` précédée, à

lui permet de maintenir sa précédente valeur. Il est toujours préférable d'initialiser ce type de variables. Les variables statiques sont placées et initialisées une seule fois, lors du chargement du code en mémoire.

Le HelloPixels avec GL4Dummies

chaque fois, d'un appel à la fonction `gl4dpSetColor` qui proposera une couleur aléatoire. Étant donné que les couleurs changent à chaque exécution dans la fonction `dessin`, nous observons que ce changement de couleurs est effectif à chaque frame.

2. Vous pouvez modifier le prototype et le corps de votre propre fonction `drawLine` (celle du DM par exemple) afin d'utiliser `gl4dpPutPixel` ou bien `gl4dpGetPixels`⁹ pour dessiner chaque pixel. L'utiliser à la place de `gl4dpLine` pour réaliser la même chose que la question précédente.



FIGURE 4.6 – Résultat attendu pour l'exercice 1 et 2 après modification du *Sample Code sc_00_01_multicolorClear-1.0*

9. Cette fonction vous permet de récupérer le pointeur vers l'ensemble de la pixmap. Pour rappel, ici chaque pixel est un entier non signé de 32 bits, vous pouvez donc les addresser via un pointeur de type `GLuint *`, `Uint32 *` ou encore `uint32_t *`. Les macros `RGB` ou `RGBA` peuvent vous servir à composer vos couleurs.

Chapitre 5

Allons un peu plus loin

Nous abordons ici quelques aspects algorithmiques pour lesquels nous exploitons les premières connaissances en programmation graphique afin de produire des résultats visuellement intéressants.

5.1 Trier des barres

Au travers de présentation de quelques algorithmes de tri, nous en profitons pour proposer une visualisation temps-réel de la donnée en cours de traitement. Nous commençons par citer quelques méthodes de tri, proposer pour certaines une implémentation puis donnons un exemple de visualisation de données que nous appliquons de manière à visualiser les principales étapes d'un algorithme de tri.

Il existe plusieurs algorithmes de tri, chacun plus ou moins efficace¹, parfois selon la situation. Ici, il ne s'agira pas d'être exhaustif ni de détailler les algorithmes les plus efficaces. Citons néanmoins quelques uns parmi les plus connus², pour chacun, nous mettons une note de bas de page exprimant un point de vue totalement personnel et arbitraire :

— Le tri par sélection³;

1. L'efficacité d'un algorithme est inversement proportionnelle à une notion appelée complexité souvent notée $\mathcal{O}(1), \mathcal{O}(\log n), \mathcal{O}(n), \mathcal{O}(n^2), \dots$ pour respectivement une complexité constante, logarithmique, linéaire ou quadratique avec n représentant le nombre d'éléments dans la donnée à traiter.

2. Ce choix d'algorithmes cités est purement arbitraire, il tente au mieux d'exprimer une certaine variété en terme d'efficacité et de stratégie. Pour plus de détails, nous conseillons au lecteur de parcourir la page WIKIPÉDIA “Algorithme de tri” : https://fr.wikipedia.org/wiki/Algorithme_de_tri

3. Simple et intuitif.

- Le tri à bulles⁴ ;
- Le tri par insertion⁵ ;
- Le tri fusion⁶ ;
- Le tri rapide ou le *Quicksort*⁷.



FIGURE 5.1 – *AlgoRhythmic* – Université de Sapientia en Roumanie : Vidéo d'une chorégraphie reprenant le déroulement de l'algorithme de tri à bulle.

Aussi, profitons de ce point pour mettre en avant une initiative originale et fort intéressante de l'Université de Sapientia en Roumanie qui propose depuis 2011 un ensemble de vidéos permettant de comprendre le fonctionnement de certains algorithmes de tri. Un groupe de danse folklorique d'Europe centrale est mis en scène de manière à reproduire séquentiellement des algorithmes de tri ; chacun des dix danseurs et danseuses porte un numéro – voir Figure 5.1 – et ils exécutent en détail les étapes liés à chaque algorithme simulé. La chaîne *AlgoRhythmic*, disponible en ligne⁸, regroupe ces vidéos d'algorithmes de tri et d'autres.

5.1.1 Le tri par sélection

Le principe du tri par sélection⁹ est simple et peut être résumé à :

4. Simpliste, local et pas toujours intuitif.
5. Élégant et minimaliste, peut donner le meilleur comme le pire.
6. Stratège et constant dans son efficacité, gourmand.
7. Stratège et, en général, le plus rapide.
8. La chaîne est disponible à l'adresse :

<https://www.youtube.com/channel/UCIqiLefbVHsOAXDAxQJH7Xw>

Et le tri à bulles est directement accessible depuis ce lien :

<https://www.youtube.com/watch?v=1yZQPjUT5B4>

9. Ici et pour l'ensemble des autres tris présentés dans cette section, nous considérons que nous trions les éléments d'un tableau dans un ordre croissant.

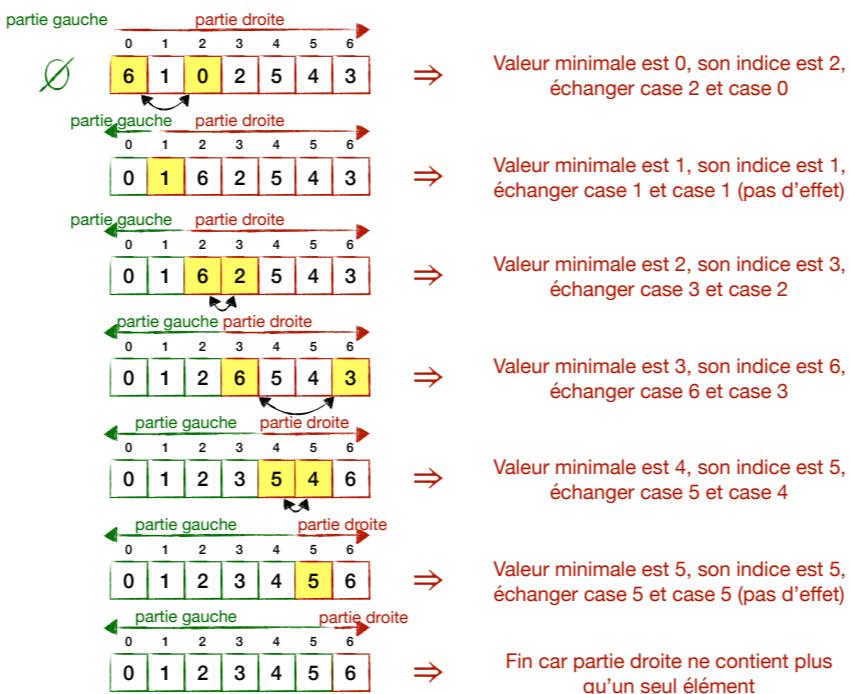


FIGURE 5.2 – Déroulement d'un tri par sélection sur un tableau à 7 éléments.

- Considérer qu'une partie gauche du tableau est triée là où la partie droite ne l'est pas – au début, la partie gauche est vide et celle de droite représente tout le tableau ;
- Parcourir l'ensemble de la partie droite en cherchant l'indice de la plus petite valeur ;
- Échanger les valeurs de la première case de la partie droite avec la case ayant l'indice de la plus petite valeur ;
- La partie gauche grossit d'un élément là où la partie droite perd un élément (en pratique il suffit de décaler un indice de début de partie droite) ;
- S'arrêter dès que la partie droite ne possède plus qu'un seul élément (donc forcément ce dernier est trié car il est seul).

La Figure 5.2 illustre au travers d'un exemple le fonctionnement d'un tri par sélection.

Ci-après, dans CODE SOURCE 5.1, nous donnons une implémentation du tri par sélection ainsi que l'ensemble des fonctions nécessaires pour que le programme soit complet. La fonction `init` initialise pseudo-aléatoirement le tableau `t` de `n` entiers passé en argument, la fonction `print` permet de l'afficher sur le terminal de commandes, la fonction `triSelection` exécute le tri par sélection sur le tableau désordonné et enfin la fonction `main` orchestre le tout.

```
#include <stdio.h>
#include <stdlib.h>
/* Initialise aléatoirement les valeurs du tableau passé en argument
 * avec des valeurs allant de 0 à n - 1. Utiliser srand (par exemple
 * srand(time(NULL))) pour avoir des valeurs différentes à chaque
 * exécution. */
void init(int * t, int n) {
    for(int i = 0; i < n; ++i)
        t[i] = n * (rand() / (RAND_MAX + 1.0));
}
/* Affiche les valeurs du tableau passé en argument. */
void print(int * t, int n) {
    for(int i = 0; i < n; ++i)
        printf("%d ", t[i]);
    printf("\n");
}
/* Tri par sélection du tableau passé en argument. */
void triSelection(int * t, int n) {
    int i, j, min, v;
    for(i = 0; i < n - 1; ++i) {
        min = i;
        for(j = i + 1; j < n; ++j)
            if(t[j] < t[min]) min = j;
        if(i != min) {
            v = t[min];
            t[min] = t[i];
            t[i] = v;
        }
    }
}
int main(void) {
    int t[64];
    init(t, 64);
    print(t, 64);
    triSelection(t, 64);
    print(t, 64);
    return 0;
}
```

Code source 5.1 – Exemple d'implémentation C d'un tri par sélection.

5.1.2 Visualiser le tri par sélection

Nous réalisons cet objectif en deux sous-parties. La première consiste à préparer le code pour une visualisation de la donnée, dans le désordre ou triée. La seconde aborde les modifications à apporter à la première pour pouvoir visualiser pas à pas le travail réalisé par l'algorithme de tri. Le lecteur aura le choix entre intégrer les éléments par lui-même ou bien télécharger directement le code source des deux étapes attendues :

https://github.com/noalien/GL4Dummies/tree/master/samples/sc_00_03_visualSort-1.0
https://github.com/noalien/GL4Dummies/tree/master/samples/sc_00_03_visualSort-1.1

Commençons par récupérer le code vu précédemment et permettant de dessiner un dégradé de gris :

https://github.com/noalien/GL4Dummies/tree/master/samples/sc_00_01_gradient-1.0

Nous insérons dans ce code¹⁰ les fonctions `init` et `triSelection` données dans CODE SOURCE 5.1. Pour dessiner chaque valeur stockée dans le tableau à trier sous la forme d'une barre verticale, nous allons avoir besoin d'une fonction spécifique (car plus optimale) au dessin de segments verticaux. Nous proposons le prototype suivant `static void vLine(int x, int y0, int y1, GLuint color)`; où `x` est l'abscisse du segment vertical, `y0` et `y1` l'ordonnée de départ et d'arrivée (sans distinguer forcément laquelle est en haut et laquelle est en bas) et enfin `color` est la couleur en 32-bits à utiliser pour le dessin de cette ligne verticale. Voici donc une proposition d'implémentation à insérer quelque part avant l'usage de cette fonction¹¹ :

```
static void vLine(int x, int y0, int y1, GLuint color) {
    int w = gl4dpGetWidth(), h = gl4dpGetHeight(), i, p;
    GLuint * pixels = gl4dpGetPixels();
    /* tous les cas à éviter, ne donnant rien */
    if(x < 0 || x >= w || (y0 < 0 && y1 < 0) || (y0 >= h && y1 >= h))
        return;
    y0 = MIN(MAX(y0, 0), h - 1);
    y1 = MIN(MAX(y1, 0), h - 1);
    p = y0 < y1 ? w : -w;
    y0 = y0 * w + x;
    y1 = y1 * w + x;
    for(i = y0; i != y1; i += p)
        pixels[i] = color;
}
```

Vous remarquerez que nous récupérons dans `w` et `h` la largeur et la hauteur

10. De préférence en début de fichier juste après les macros (tous les `#include` et autres `#define`) afin de ne pas avoir besoin de prototyper les fonctions pour les utiliser avant leur définition.

11. Ou bien optez pour un prototypage en haut du fichier et une définition où bon vous semble.

du *screen* en cours, que **pixels** est un pointeur vers les pixels de ce *screen*, que les coordonnées sont testées et éventuellement modifiées pour toujours se retrouver à l'intérieur de la grille du *screen* et enfin que *y0* et *y1* sont modifiés afin de devenir les indices (incluant l'abscisse et les sauts *w* en fonction de la ligne) du point de départ et d'arrivée (ou inversement) du segment ; nous allons de l'un à l'autre en utilisant un pas *p* qui vaut *w* ou *-w*.

Étant donnée que dans ce code nous accèderons à plusieurs endroits à la données “tableau trié ou à trier”, nous optons pour une déclaration en variable globale mais, attention, avec le *qualifier* qui la rend statique, donc accessible qu’au sein de ce fichier. Nous ajoutons ainsi en haut du fichier (toujours après les macros et autres prototypes) :

```
/* inclusion des entêtes pour rand/srand */
#include <stdlib.h>
/* inclusion des entêtes pour time (voir son usage dans main) */
#include <time.h>

/* nombre d'élément du tableau à trier */
#define N 128
/* le tableau à trier */
static int _a_trier[N];
```

Nous ajoutons aussi une fonction **calcul** qui sera appelée comme *callBack* de type *idle* pour jouer le rôle de la phase simulation dans la boucle infinie d'affichage événement-simulation-dessin, modifions aussi la fonction **dessin** :

```
/* appelée à chaque idle par gl4duwMainLoop */
static void calcul(void) {
    triSelection(_a_trier, N);
}
/* appelée à chaque draw par gl4duwMainLoop */
static void dessin(void) {
    int i;
    for(i = 0; i < N; ++i)
        vLine(i, 0, _a_trier[i], RGB(128, 128, 128));
    gl4dpUpdateScreen(NULL);
}
```

Finissons par quelques modifications dans la fonction **main** :

```
int main(int argc, char ** argv) {
    /* initialisation de la chaîne pseudo-aléatoire en utilisant le temps */
    srand(time(NULL));
    /* remplir aléatoirement le tableau */
    init(_a_trier, N);
    if(!gl4duwCreateWindow(argc, argv, /* args du programme */
                           "GL4Dummies'_Hello_Pixels", /* titre */
                           10, 10, 4 * N, 2 * N, /* x,y, largeur, hauteur */
                           GL4DW_SHOWN) /* état visible */) {
        return 1; /* échec si ici */
    }
    /* screen aux dim de la data à trier (nb-élém. et valeurs dedans) */
    gl4dpInitScreenWithDimensions(N, N);
    /* ajoute quitte à la pile des TODO en sortant du programme */
    atexit(quitte);
    /* met en place la display au sein de la boucle event-simu-draw */
```

```
gl4duwDisplayFunc(dessin);
/* met en place la idle correspondant à la phase "simulation".
 * Commentez cette ligne pour visualiser le tableau non trié. */
gl4duwIdleFunc(calcul);
/* boucle infinie pour éviter l'arrêt prématué du programme */
gl4duwMainLoop();
return 0;
}
```

Nous soulignons que la donnée `_a_trier`¹² est initialisée en tout début de programme, précédée d'un `srand` qui permet de modifier l'étape initiale de la chaîne pseudo-aléatoire¹³. Aussi, que la fenêtre et le *screen* sont à des dimensions relatives à N , le nombre d'éléments dans `_a_trier` mais aussi un de plus que la valeur maximale qui peut être mise dedans par la fonction `init`. Enfin, remarquez la présence du *set* de la *callback* de simulation.

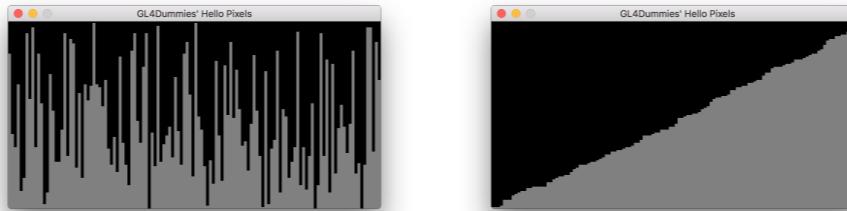


FIGURE 5.3 – Résultats figés de la visualisation de la donnée non triée (à gauche) et triée (à droite).

Ce code ainsi modifié / complété, donne par exemple les résultats présents sur la Figure 5.3 sachant que l'image de gauche et de droite correspondent respectivement au fait d'avoir commenté ou laissé telle quelle la ligne de code `gl4duwIdleFunc(calcul);`.

Notons que dans un cas ou dans l'autre, le résultat est toujours figé, soit le tableau n'est pas trié, soit il l'est mais rien ne change visuellement. Nous apportons donc les modifications au code qui vont nous permettre de visualiser les modifications quand ces dernières se produisent. Ici il ne s'agit pas de piloter le graphique par la simulation¹⁴ mais plutôt de faire en sorte que

12. Pour la distinguer des autres, j'ai pour habitude personnelle de précéder le nom d'une variable statique et globale par un *underscore* ... et non un tiret du 8 comme disent certains :)

13. Connaissez-vous l'effet papillon ?

14. Ceci est possible, voire même plus facile pour une personne maîtrisant les appels bas niveau de la bibliothèque de gestion du fenêtrage (forcer la mise à jour de la fenêtre quand l'algorithme de tri le souhaite), mais cela va à l'encontre du paradigme *event-simu-draw*.

la simulation, au rythme de la visualisation, apporte le peu de nouveautés étape par étape pour que nous puissions visualiser le travail de l'algorithme de tri en œuvre ici. Ce résultat attendu est visualisable dans la vidéo présente à cette url :

<https://youtu.be/fZm-jPDJnqM>

Ainsi, nous commençons par ajouter deux éléments de type entier qui nous permettent de savoir, à l'issue d'une étape de l'algorithme, quelles sont les deux cases (donc quels sont les indices) du tableau que nous devons échanger. De cette manière, nous pourrons les surligner en rouge comme dans la vidéo juste avant de les échanger. Nous optons pour un tableau statique et global à deux éléments pour stocker les deux indices qui nous intéressent :

```
/* les indices des deux données qui échangeront leurs places */  
/* ligne à placer par exemple après la déclaration de _a_trier */  
static int _to_swap[2] = {-1, -1};
```

Notons que la valeur -1 correspond à un indice que nous considérons non valide. Ainsi, quand nous rencontrons -1 nous ne surlignerons aucune entrée.

La partie qui subit le plus de changements est l'algorithme lui-même, donc la fonction **triSelection**. Nous souhaitons qu'à chaque fin de phase de simulation nous soyons prêt (tant que cela n'est pas fini) à échanger deux nouveaux éléments dans le tableau car nous aurons "sélectionné" le plus petit du sous-tableau de droite et il sera prêt à être échangé avec le premier élément de ce sous-tableau, c'est-à-dire le i de la boucle **for** de la version initiale. Donc à chaque appel, il s'agit d'un seul i et d'un seul \min ; i est alors transformé en variable statique commençant à zéro et qui ne s'incrémentera qu'au maximum une fois à la fin pour préparer l'étape suivante, c'est-à-dire trouver le meilleur pour la place $i=1 \dots$. Le **for** sur le i disparaît donc au détriment d'un simple test $i < n - 1$. Par ailleur, au lieu de faire l'échange dès que nous avons les bons candidats i et \min , nous nous contentons de les stocker dans **_to_swap** pour le faire à l'étape d'après, se laissant ainsi le temps de surligner les deux valeurs à échanger avant de le faire réellement. Ainsi, nous savons qu'il faudra – dès le début d'une nouvelle étape, et si nous avons des valeurs différentes dans les deux cases de **_to_swap** – faire un échange pour rendre effectif l'étape précédente. Tout ceci nous amène à la proposition suivante :

```
static void triSelection(int * t, int n) {  
    /* i devient statique car la boucle disparaît à ce niveau */  
    static int i = 0;  
    int j, min;  
    /* échanger après coup les deux valeurs de l'étape précédente */  
    if(_to_swap[0] != _to_swap[1]) {
```

```
    int v = t[_to_swap[1]];
    t[_to_swap[1]] = t[_to_swap[0]];
    t[_to_swap[0]] = v;
}
/* dire qu'aucune donnée n'est à échanger */
_to_swap[0] = _to_swap[1] = -1;
/* plus de boucle, on attend le prochain tour idle */
if(i < n - 1) {
    min = i;
    for(j = i + 1; j < n; ++j)
        if(t[j] < t[min])
            min = j;
    /* marquer les deux données à échanger au prochain appel */
    _to_swap[0] = i; _to_swap[1] = min;
    /* i se prépare pour la prochaine fois en augmentant de 1 */
    ++i;
    /* ajouter un délai en ms (ralentir 1/4 de seconde) pour
     * avoir le temps de visualiser l'étape */
    SDL_Delay(250);
}
}
```

Pour compléter, la ligne optionnelle contenant `SDL_Delay(250);` n'est là que pour ralentir le rendu et laisser le temps à l'utilisateur de voir ce qui se passe. Ce ralentissement peut-être placé ailleurs, par exemple dans `dessin`.

Les derniers ajouts concernent la fonction `dessin` qui aura deux tâches supplémentaires à effectuer. La première (on va faire simple), effacer l'ensemble du *screen* afin que les dessins ne viennent pas s'ajouter en surimpression. La seconde, surligner en rouge les deux valeurs aux positions échangeables stockées dans `_to_swap`. Ce qui donne :

```
static void dessin(void) {
    int i;
    /* effacer l'écran à chaque fois car la donnée peut être
     * différente d'une frame à l'autre */
    gl4dpClearScreen();
    for(i = 0; i < N; ++i)
        vLine(i, 0, _a_trier[i], RGB(128, 128, 128));
    /* surligner en rouge les deux positions à échanger */
    if(_to_swap[0] >= 0) {
        vLine(_to_swap[0], 0, _a_trier[_to_swap[0]], RGB(255, 0, 0));
        vLine(_to_swap[1], 0, _a_trier[_to_swap[1]], RGB(255, 0, 0));
    }
    gl4dpUpdateScreen(NULL);
}
```

5.1.3 Le tri à bulles ou d'autres

Le tri à bulles fonctionne sur le principe suivant : parcourir du début à la fin (soit de gauche à droite) le tableau en regardant la case courante et celle qui est juste à sa droite, si la courante est plus grande que celle de droite alors les échanger et continuer jusqu'au bout (en pratique le bout - 1 car on regarde toujours un coup à l'avance à droite). Conclusion à la fin

de ce parcours, on imagine que la valeur la plus grande du tableau serait remontée de proche en proche de la gauche (le bas) vers la droite (le haut), comme une bulle qui remonte dans un liquide. Il est sûr que la plus grande valeur se retrouve complètement à droite, mais au passage, nous savons que quelques valeurs qui sont grandes, mais pas les plus grandes, ont aussi réussi à remonter légèrement. Il suffit ainsi de considérer que nous avons trouvé et classé (au bout à droite) la plus grande valeur du tableau et qu'il reste à faire de même pour la deuxième plus grande sur le sous-tableau allant un cran moins loin. Puis ainsi de suite, pour la troisième plus grande, la quatrième ... Ceci ressemble assez à l'algorithme de tri par sélection mais il faut noter qu'ici des échanges peuvent intervenir plusieurs fois à chaque grande étape (trouver le plus grand du sous tableau de gauche) alors que dans un tri par sélection nous n'échangeons qu'une seule fois. C'est ce qui fait de l'algorithme de tri à bulles l'un des moins efficaces.

Nous proposons une implémentation de ce tri, sur le même modèle que le tri par sélection :

```
#include <stdio.h>
#include <stdlib.h>
void init(int * t, int n) {
    for(int i = 0; i < n; ++i)
        t[i] = n * (rand() / (RAND_MAX + 1.0));
}
void print(int * t, int n) {
    for(int i = 0; i < n; ++i)
        printf("%d ", t[i]);
    printf("\n");
}
void triBulles(int * t, int n) {
    int i, j;
    for(i = n - 1; i >= 0; --i)
        for(j = 1; j <= i; ++j)
            if(t[j - 1] > t[j]) {
                /* vous vous souvenez ? échange de 2 variables
                 * sans utiliser une troisième */
                t[j]      ^= t[j - 1];
                t[j - 1] ^= t[j];
                t[j]      ^= t[j - 1];
            }
}
int main(void) {
    int t[64];
    init(t, 64);
    print(t, 64);
    triBulles(t, 64);
    print(t, 64);
    return 0;
}
```

Code source 5.2 – Exemple d'implémentation C d'un tri à bulles.

Enfin, sans aller jusqu'à expliquer le fonctionnement d'autres tris, nous proposons au lecteur une implémentation d'un tri par insertion et d'un tri par fusion, chacune respectivement donnée dans les CODE SOURCE 5.3 et 5.4.

```
#include <stdio.h>
#include <stdlib.h>
void init(int * t, int n) {
    for(int i = 0; i < n; ++i)
        t[i] = n * (rand() / (RAND_MAX + 1.0));
}
void print(int * t, int n) {
    for(int i = 0; i < n; ++i)
        printf("%d ", t[i]);
    printf("\n");
}
void triInsertion(int * t, int n) {
    int i, j, v;
    for(i = 1; i < n; i++) {
        v = t[j = i];
        while(j > 0 && t[j - 1] > v) {
            t[j] = t[j - 1];
            j--;
        }
        t[j] = v;
    }
}
int main(void) {
    int t[64];
    init(t, 64);
    print(t, 64);
    triInsertion(t, 64);
    print(t, 64);
    return 0;
}
```

Code source 5.3 – Exemple d'implémentation C d'un tri par insertion.

```
#include <stdio.h>
#include <stdlib.h>
void init(int * t, int n) {
    for(int i = 0; i < n; ++i)
        t[i] = n * (rand() / (RAND_MAX + 1.0));
}
void print(int * t, int n) {
    for(int i = 0; i < n; ++i)
        printf("%d ", t[i]);
    printf("\n");
}
int s[64]; /* pas beau ..., l'allocation dynamique
             * c'est pour plus tard */
void triFusion(int * t, int g, int d) {
    int i, j, k, m;
    if(d > g) {
        m = (d + g) >> 1;
        triFusion(t, g, m);
        triFusion(t, m + 1, d);
        for(i = m; i >= g; i--)
            s[i] = t[i];
        for(j = m; j < d; j++)
            s[d + m - j] = t[j + 1];
        for(k = g, i = g, j = d; k <= d; k++)
            t[k] = (s[i] < s[j]) ? s[i++] : s[j--];
    }
}
int main(void) {
    int t[64];
    init(t, 64);
    print(t, 64);
    triFusion(t, 0, 63);
    print(t, 64);
    return 0;
}
```

Code source 5.4 – Exemple d'implémentation C d'un tri par fusion.

5.1.4 Exercice

1. Sur le modèle de ce qui a été fait pour visualiser le fonctionnement du tri par sélection, faire de même pour le tri à bulles ;
2. Sur le modèle de ce qui a été fait pour visualiser le fonctionnement du tri par sélection, faire de même pour au choix : le tri par insertion, le tri par fusion, ou bien un autre tri qui vous intéresse ;

5.2 Simuler un feu de forêt

Pas aujourd’hui (plus tard dans le semestre)

(3) Analyse Discrète Différentielle (ADD)

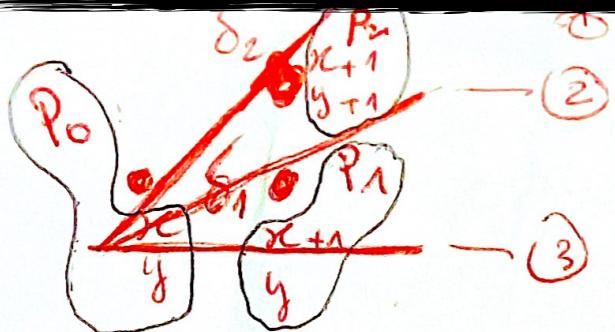
Bresenham 1965 => la droite :

- https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
- Mais aussi gl4dpLine (voir code source de GL4Dummies fichier gl4dp.c)

Bresenham 1977 => le cercle :

- <http://public.callutheran.edu/~reinhart/CSC505/Week1/BresenhamCircles.pdf>
- Mais aussi gl4dpCircle (voir code source de GL4Dummies fichier gl4dp.c)

A exposer en cours



- ① $\delta_1 < 0 \text{ et } \delta_2 < 0 \rightarrow P_2$
- ② $\delta_1 < 0 \text{ et } \delta_2 > 0 \rightarrow ?$
- ③ $\delta_1 > 0 \text{ et } \delta_2 > 0 \rightarrow P_1$

~~(3)~~ \Rightarrow

- ① $\delta_1 + \delta_2 < 0 \rightarrow P_2$
- ③ $\delta_1 + \delta_2 > 0 \rightarrow P_1$
- ② ? mais est-ce que les conditions de ① et ② fonctionnent \rightarrow oui !

$$\Delta = \delta_1 + \delta_2$$

$$\Delta = (u \cancel{y} - v(x+1)) + (u(y+1) - v \cancel{(x+1)})$$

$$\boxed{\Delta = 2uy - 2vx + u - 2v}$$

Comment varie Δ ?

$$x \rightarrow x+1$$

* si P_1 choisi : $\delta\Delta_H = (2uy - 2v(x+1) + u - 2v) - (2uy - 2vx + u - 2v)$

$$\delta\Delta_H = -2v \Rightarrow \boxed{\text{linc H} = -2v}$$

* si P_2 choisi : $\delta\Delta_O = (2u(y+1) - 2v(x+1) + u - 2v) - (2uy - 2vx + u - 2v)$

$$\delta\Delta_O = 2u - 2v \Rightarrow \boxed{\text{linc O} = 2u - 2v}$$

eq: $y - \frac{v}{u}x = 0$
ou $\boxed{uy - vx = 0}$

qd $x = y = 0$

$$\boxed{\Delta_1 = u - 2v} \quad \boxed{\Delta_0 = 0}$$

$$\boxed{\Delta_1 = u - 2v} \quad \boxed{\Delta_0 = 0}$$

(4)

Écrivons ensemble une fonction qui dessine des cercles

Rappel de l'équation d'un cercle : $(x - x_0)^2 + (y - y_0)^2 = r^2$
où (x_0, y_0) est le centre du cercle et r son rayon

(AIDE AU SLIDE SUIVANT)

POINT DE DÉPART

https://github.com/noalien/GL4Dummies/tree/master/samples/sc_00_00_blank-1.0

il suffit de calculer ce tronçon
et reproduire le reste par symétrie

