

# Rapport Projet Lightup

Nicolas Dubuisson\*

8 mai 2022

---

\*Bordeaux University

## Table des matières

<b>1</b>	<b>Le teminal</b>	<b>3</b>
1.1	Objectif . . . . .	3
1.1.1	La mise en place des fichiers . . . . .	3
1.1.2	Les structures et les allocations . . . . .	3
1.1.3	Les fonctions main() . . . . .	4
1.1.4	Les fonctions de sauvegarde . . . . .	6
1.1.5	Les fonctions test . . . . .	7
1.1.6	Makefile-CMake . . . . .	7
1.2	Problèmes rencontrés . . . . .	8
1.2.1	Solutions . . . . .	8
<b>2</b>	<b>L'interface graphique SDL2</b>	<b>9</b>
2.1	Objectif . . . . .	9
2.1.1	Explication de code . . . . .	9
2.2	Problèmes rencontrés . . . . .	10
<b>3</b>	<b>Android</b>	<b>11</b>
3.1	Objectif . . . . .	11
3.2	Problèmes rencontrés . . . . .	12
<b>4</b>	<b>Web</b>	<b>12</b>
4.1	Objectif . . . . .	12
4.1.1	les scripts . . . . .	13
4.2	Problèmes rencontrés . . . . .	14
<b>5</b>	<b>GitLab</b>	<b>14</b>
<b>6</b>	<b>Conclusion</b>	<b>14</b>

# 1 Le teminal

- Langague : C
- Logiciel : Visual Studio Code
- Technologie : Git, Makefile, CMake

## 1.1 Objectif

Dans cette partie, je vais vous présenter les points dont je juge les plus importants du projet.

### 1.1.1 La mise en place des fichiers

L'objectif de cette partie était de nous apprendre à programmer un jeu en mode text et jouable dans le terminal.

Dans un premier temps, nous avons dû apprendre à maîtriser le langage C et le Makefile. C'est-à-dire maîtriser la manipulation et la compilation des fichiers \*.c et \*.h.

Pour le TP, nous avons créé les fichiers suivant :

- **game\_aux.c game\_aux.h** pour l'interface 2D sur le terminal.
- **game\_text.c** et **game\_solve.c** sont des programmes exécutables (disposant de la fonction main) qui permettent à un utilisateur de jouer interactivement en mode texte.
- **game.c game.h** pour l'interface des fonctions principales du jeu.
- **game\_ext.c game\_ext.h** pour l'allocation d'un nouveau jeu et les fonctions undo-redo.
- **game\_tools.c game\_tools.h** pour la fonction solveur et les fonctions de sauvegarde.
- **queue.c queue.h** leurs fonctions vont permettre de mettre le jeu dans une pile et ainsi de pouvoir utiliser les fonctionnalités undo-redo.
- **game\_examples.c game\_examples.h** contient des exemples de jeu à afficher.

Dans les fichiers \*.h, nous devons juste définir les fonctions que nous voulons utiliser dans les fichiers \*.c.

### 1.1.2 Les structures et les allocations

Pour rendre le code le plus lisible possible, le projet utilise de **typedef struct** et le **typedef enum** pour créer le jeu. Chaque case correspond à une constante de enum **square** (ci dessous). Ainsi, nous pouvons identifier quelle case correspond à quoi.

Un extrait de `game_aux.c`

```
1 game game_default(void) {
2     square jeu[49] = {
3         S_BLANK, S_BLANK, S_BLACK1, ...
4     }
5     return game_new(jeu);
}
```

**Typedef struct** contient des variables qui sont stockées dans la mémoire. **Typedef enum** contient des constantes qui sont stockées dans la mémoire.

Nous utilisons un **malloc** pour l'allocation de mémoire de taille la structure **game\_s**. Et nous utilisons un **calloc** pour allouer un espace mémoire de la taille d'un tableau. Afin d'éviter les erreurs, on utilise des **assert** qui vérifient si la variable allouée n'est pas **NULL**. Ainsi, pour chaque variables contenus dans la structure, on lui attribue une valeur.

Un extrait de `game_ext.c`

```
1 game game_new_empty_ext(uint nb_rows, uint nb_cols,
2                           bool wrapping){
3     game g = (game) malloc(sizeof(struct game_s));
4     assert(g);
5     g->nb_rows = nb_rows;
6     g->nb_cols = nb_cols;
7     g->squares=(square *) calloc(g->nb_rows*g->nb_cols,
8                                 sizeof(uint));
9     assert(g->squares);
10    ...
11 }
```

Remarque : Lorsque l'on assigne le type **cgame** à une variable, on ne peut pas modifier les valeurs des variables contenu dans la structure. Tandis qu'avec **game**, on peut le faire.

```
1 typedef struct game_s *game;
2 typedef const struct game_s *cgame;
```

### 1.1.3 Les fonctions main()

Les fonctions **main()** sont les points de départ de l'exécutable créé. On peut mettre des valeurs en paramètre de l'exécutable qui seront directement misent dans le tableau **argv[]**. Pour le projet, on a créé quatre exécutables :

`./game_text`, `./game_solve`, `./game_sdl` (ici) et `./game_test` (ici). L'objectif de `./game_text` est d'afficher un jeu par défaut ou de charger un jeu qu'on lui a mis en paramètre `./game_text Monjeu.txt`. La variable **argc** correspond au nombre d'argument qu'on a rentré en paramètre de l'exécutable.

Un extrait de `game_text.c`

```
1 int main(int argc, char *argv[]) {
2     if (argc > 2) {
3         fprintf(stderr, "error_too_many_arguments\n");
4         return EXIT_FAILURE;
5     } else if (argc == 2) {
6         game g = NULL;
7         g = game_load(argv[1]);
8         assert(g);
9         game_print(g);
10        ...
11    } else {
12        game g = NULL;
13        g = game_default();
14        ...
15    } ...
16 }
```

L'objectif de `./game_solve` n'est pas de jouer au jeu, même si sa fonction **main()** ressemble à celle de `game_text.c`. L'exécutable comprend une série de paramètre, le but étant de prendre un fichier \*.txt ("input") avec un jeu écrit dessus (ici) et d'afficher en sortie sur le terminal (ou sur un fichier ("output")) soit la solution (option "-s"), soit le nombre de solution du jeu (option "-c").

```
1 $ ./game_solve <option> <input> [<output>]
```

### 1.1.4 Les fonctions de sauvegarde

C'est l'une des parties que j'ai trouvées intéressantes à travailler. Nous avons créé deux fichiers : Le premier, **game\_load(char \*filename)** prend et lit un fichier au format text. Pour enregistrer un jeu au format text, nous avons ajouté une commande - **press 'w <filename.txt>' to save the game**. Ainsi dans **game\_load()**, on prend un fichier text avec un jeu, on l'ouvre "fopen" et on le lit "r". La lecture se fait grâce au **fscanf** et il ne faut pas oublier de fermer la lecture **fclose(f)** à la fin.

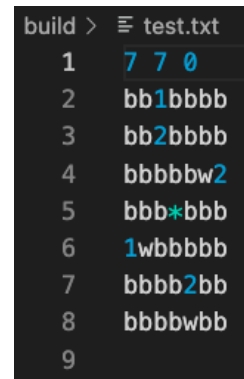


FIGURE 1 – Jeu text

Un extrait de **game\_tools.c**

```
1 game game_load(char *filename) {
2     FILE *f;
3     f = fopen(filename, "r");
4     assert(f != NULL);
5     uint row, col, wrap;
6     int ret = fscanf(f, "%u_%u_%u", &row, &col, &wrap);
7     ...
8     bool wrapping = wrap == 1;
9     fclose(f);
10    game g = game_new_ext(row, col, game_layout, wrapping);
11    game_update_flags(g);
12    return g;
13 }
```

La seconde fonction **game\_save(cgame g, char \*filename)** prend le jeu et l'écrit dans un fichier text. La fonction **fopen** prend alors comme paramètre le fichier text et "w" pour write. Puis on utilise **fprintf** pour écrire dans le fichier text.

```
1 void game_save(cgame g, char *filename) {
2     assert(g);
3     assert(g->nb_rows < 10 && g->nb_cols < 10);
4     FILE *f;
5     f = fopen(filename, "w");
6     assert(f != NULL);
```

```

7     fprintf(f, "%u_%u_%u\n", g->nb_rows, g->nb_cols,
8                                     g->wrapping);
9     ...
10    fclose(f);
11 }

```

### 1.1.5 Les fonctions test

Les fonctions de test ont pour but de tester les fonctions du jeu que l'on a créé. Ainsi, dans chacune des fonctions test, on devait utiliser la fonction à tester et vérifier si elle était non vide, si elle renvoyait la bonne chose, si il n'y avait pas une erreur de segmentation ou de mémoire, ...

Nous avons créé un nouvel exécutable qui permet de tester chacune de ces fonctions tests. La fonction "main()" de **game\_test.c** prend comme paramètre un text (ci-dessous) et, en fonction, elle appelle la fonction test définie dans une structure.

Un extrait de **CMakelist.txt**

```

1 add_executable(game_test game_test.c ...)
2 target_link_libraries(game_test game)
3
4 add_test(testv1_default ./game_test "default")
5 add_test(testv1_print ./game_test "print")
6 ...
7 {"default", test_default},
8 {"print", test_print},

```

Un extrait de **game\_test.c**

```

1 struct test tests[] = {
2     {"default", test_default},
3     {"print", test_print},
4     ...
5 }

```

### 1.1.6 Makefile-CMake

Le Makefile va nous permettre de générer un fichier exécutable à partir de nos fichiers \*.c . Dans l'exemple ci-dessous, on définit dans un premier temps le type du langage C utilisé. Puis l'exécutable **game** qui doit générer les fichiers \*.o . Enfin, on met en place un système de nettoyage de fichier avec "rm".

Commandes utilisées : make et make rm

```

1 CFLAGS= -Wall -g -std=c99
2 all : game
3 game : game.o game_print.o game_save.o
4 rm:
5     rm -f *.o game *.txt
6 .PHONY: all

```

Le même exemple mais en utilisant CMake. On génère dans un premier temps tout les fichiers que l'exécutable aura besoin, puis dans un second temps on génère l'exécutable. C'est une technologie intéressante car cela nous permet de générer plusieurs exécutables, mais aussi comme on l'a vu, de pouvoir effectuer des tests.

Commandes utilisées : mkdir build, cd build ; cmake .. ; make

```

1 cmake_minimum_required(VERSION 3.0)
2 project("Morpion" C)
3
4 set(CMAKE_C_FLAGS "-std=c99 -Wall")
5 set(CMAKE_C_FLAGS_DEBUG "-g -coverage")
6
7 add_library(game_lib game_print.c game_save.c)
8 add_executable(game game.c)
9 target_link_libraries(game game_lib)

```

## 1.2 Problèmes rencontrés

Parmi tout ce que j'ai expliqué sur cette partie **Terminal**, avec mon équipe, nous avons rencontré de légers problèmes, comme sur les fuites mémoire, les segmentation faults et l'écriture des fonctions test. Concernant le travail, j'ai trouvé l'écriture de la fonction **game\_update\_flags()** très compliquée.

Le moment le plus compliqué que je peux relever a été au rebut du TD 2 lorsqu'il a fallu écrire la fonction **game\_print()**. Je tiens à le souligner car plusieurs camarades de classes se sont retrouvés dans la même situation. J'aurais souhaité plus d'information sur ce sujet. La méthode éducative ne m'a pas correspondu pour cette partie.

### 1.2.1 Solutions

Grâce au CMake, on peut ajouter des options afin de visualiser après compilation, les fuites de mémoire et les parties du code qui ne sont pas utilisés. Mais aussi grâce aux fonctions tests, nous avons pu vérifier si nos fonctions étaient correctes. Il suffisait de faire "make test" pour les vérifier et



on pouvait en plus faire "make ExperimentalMemCheck" afin de visualiser les potentielles fuites mémoire.

## 2 L'interface graphique SDL2

- Langague : C
- Logiciel : Visual Studio Code
- Technologie : SDL2

### 2.1 Objectif

Le but de cette partie est d'apprendre à maîtriser le développement d'une interface graphique.

Pour ce faire nous avons un fichier **main.c** qui contient la fonction "main()" et qui va servir de point de départ. Nous avons par ailleurs un fichier **game\_sdl.c** qui inclu quatre fonctions :

- init() qui initialise les différentes variables.
- render() pour charger les différentes textures.
- process() pour lire en boucle et ainsi détecter chaque action que l'utilisateur apporte.
- clean() pour désallouer la mémoire et détruire les textures.

Nous avons ainsi créer un nouvel exécutable qui doit inclure les librairies **SDL image et ttf**.

```
1 include_directories(${SDL2_ALL_INC})
2 add_executable(game_sdl main.c game_sdl.c)
3 target_link_libraries(game_sdl ${SDL2_ALL_LIBS} m game)
```

#### 2.1.1 Explication de code

Pour la fonction **init()**, nous avons d'abord commencé par allouer de la mémoire afin de pouvoir utiliser les variables dans toutes les fonctions. Nous avons utilisé **SDL\_GetWindowSize(win, &w, &h)** pour avoir la taille de la fenêtre et **load\_texture(ren, BUTTON)** pour charger un bouton. Dans **render()**, il a fallu calculer l'emplacement exacte de chacunes des cases et insérer la texture adéquate. Nous avons donc dans un premier temps tracé une grille à l'aide de **SDL\_RenderDrawLine()**. Puis, nous calculons l'emplacement de chaque case à l'aide de cette formule :

```
1 square_width=((SCREEN_WIDTH/15)*w/SCREEN_WIDTH);
2 square_height=((SCREEN_WIDTH/15)*h/SCREEN_HEIGHT);
3
4 env->grid_x = (w - env->g->nb_cols*square_width)/2;
```

```

5 | env->grid_y = (h - env->g->nb_rows*square_height)/2;
6 |
7 | rect.x = (env->grid_x) + move_x + (5*w/SCREEN_WIDTH);
8 | rect.y = (env->grid_y) + move_y + (5*h/SCREEN_HEIGHT);

```

Imaginons une fenêtre de 600 par 600 pixels. Afin d'avoir un tableau assez grand, nous avons décidé de laisser 160px de chaque côté du tableau. Ainsi **square\_width** et **square\_height** prennent la valeur de 40. On peut donc calculer **env->grid\_x** et **env->grid\_y** qui prennent 160. On peut désormais placer point (en rouge) à la position (160,160).

Puis à l'aide de deux boucles "for", on trace le trait **env->grid\_x + square\_width** = (200,160) jusqu'au bout (440,160) et on revient au début en ajoutant **square\_height** à **env->grid\_y** (flèche bleu). Grâce à cela, on peut désormais calculer le centre de chacune des cases qui correspond à la taille des cases divisées par deux, (180,180) pour le point vert.

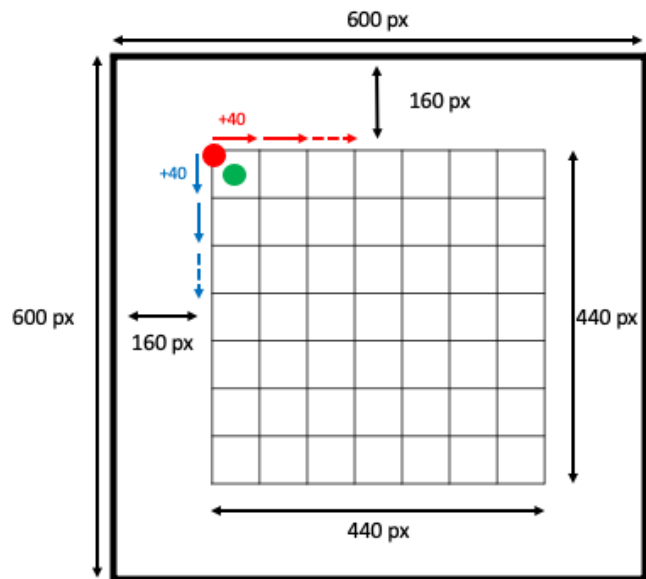


FIGURE 2 – Dimension du jeu

Nous plaçons les textures grâce à **SDL\_RenderCopy()**. La fonction **process()** doit détecter l'emplacement de la souris, c'est donc pour cela qu'on utilise **SDL\_GetMouseState()**. Notre algorithme calcule où se trouve le pointeur de la souris dans un premier temps. Dans un second temps, il arrive à savoir s'il est compris entre une certaine ligne x et une certaine ligne y, et alors de connaître la case. Ainsi on peut donc placer les différentes squares. Pour finir dans **clean()** on utilise **SDL\_DestroyTexture()** pour détruire les images et **free(env)** pour désallouer la structure.

## 2.2 Problèmes rencontrés

Le téléchargement des libraries sur nos ordinateurs fonctionnait, cependant elles se téléchargeaient dans les mauvais répertoire. En plus ces réper-

toires étaient visible qu'avec le terminal, donc j'ai du faire des `cp -r /répertoire_source /répertoire_destination"`..

### 3 Android

- Langague : C
- Logiciel : Visual Studio Code, Android Studio
- Technologie : SDL2

#### 3.1 Objectif

Le but de cette partie est d'apprendre à créer un fichier APK à partir des fichiers que l'on possède.

Pour ce projet, nous avons dû créer plusieurs dossiers. Le premier dossier **assets** possède toutes les images du jeu. Nous avons utilisé des liens symbolique `ln -s [fichier cible] [Nom de fichier symbolique]`. Le second dossier **jni/src** possède tous les fichiers \*.c et \*.h que l'on souhaite utiliser. Une fois encore nous avons utilisé des liens symboliques. Nous avons modifié les fichiers sources à copier dans le fichier "Android.mk". Le troisième dossier **res** possède l'icon du jeu. Et pour finir dans le dossier **.gitignore**, nous avons rajouté les lignes suivantes pour la compilation :

```
android/bin/*
android/jni/SDL*
android/libs/*
android/obj/*
android/gen/*
```

Pour la création du fichier APK, une série de commande doit être écrite :

```
./init-cremi.sh
./download-cremi.sh
source crema-env.sh
ndkbuild # libs/x86/*.so
ant debug # *.apk
android avd # simulateur
```

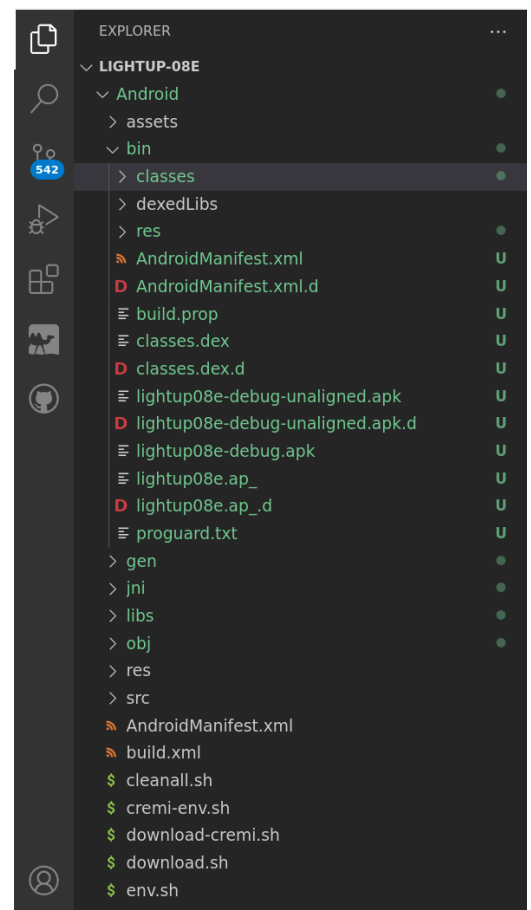


FIGURE 3 – Dossiers Android

Une fois le simulateur lancé, on doit choisir la version d'Android, la taille de l'écran,... Puis nous faisons un drag and drop du fichier **lightup08e-debug.apk** (photo ci dessus) dans le smartphone.

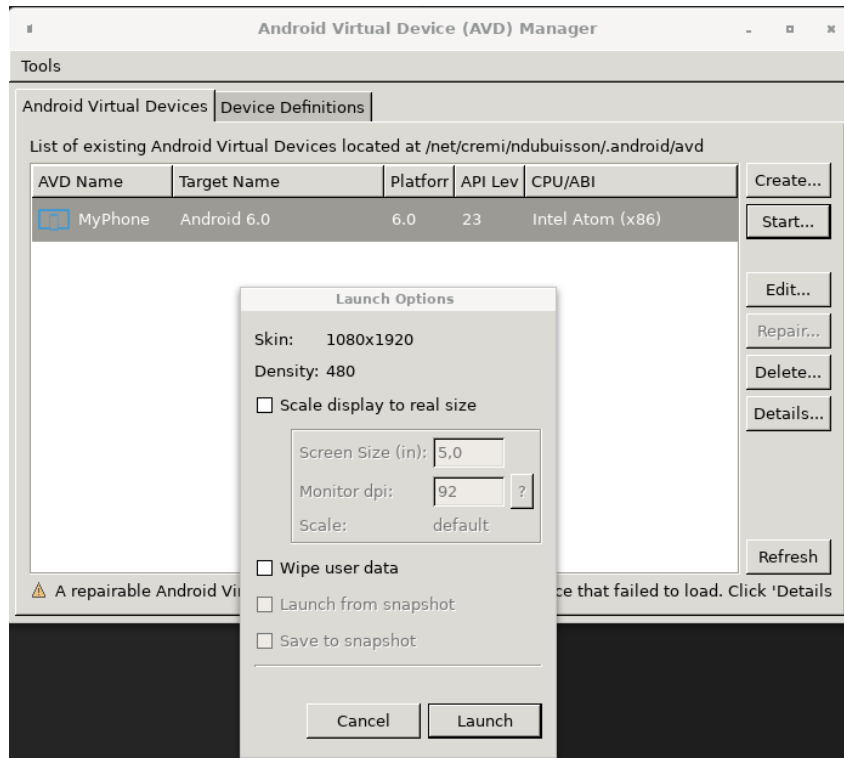


FIGURE 4 – Simulateur Android

### 3.2 Problèmes rencontrés

Contrairement aux autres groupes de ma classe, nous avons rencontré aucun problème. A part le fait que je n'ai pas réussi à créer le fichier APK sur mon ordinateur personnel.

## 4 Web

- Langage : C, HTML, CSS, Javascript
- Logiciel : Visual Studio Code
- Technologie : WebAssembly

### 4.1 Objectif

Le but de cette partie est de pouvoir jouer au jeu sur le web. Nous avons créé un fichier **index.html** qui permet de définir la structure de ma page web, c'est à dire l'emplacement de mon jeu, de mes boutons, du text,... Pour

avoir un espace propre, nous avons créé trois dossiers.

Le premier dossier **CSS** contient mon fichier **Page1.css** et permet de rendre la page web plus lisible. Le deuxième dossier **script** contient tous les fichiers Javascript, le **wrapper.c** et le **Makefile**. Et le troisième dossier **src** contient tous les fichiers \*.c et \*.h, ainsi que les images. Tous ces fichiers sont des liens symboliques comme pour Android. Le site du jeu.

#### 4.1.1 les scripts

Le fichier **wrapper.c** contient toutes les fonctions utiles pour la création du jeu Web. Le **Makefile** compile chaque fichier \*.c dans le dossier **src**. Cette compilation est possible grâce à **WebAssembly**. Etant sur MacOS, j'avais juste besoin de télécharger **emscripten** à l'aide de la commande **brew install emscripten**. J'ai pu compiler sans problème et cela a créé un fichier **game.js**. Je trouve cette technologie très intéressante.

Nous avons créé un fichier **canvas.js** pour l'ensemble des fonctions javascript que nous utilisons pour le jeu web. Il y a trois fonctions principales, la première **start** est utilisée lors du chargement de la page web.

```
1 var g;
2 var nb_rows, nb_cols;
3
4 function start() {
5     g = Module.__new_default();
6     nb_rows = Module.__nb_rows(g);
7     nb_cols = Module.__nb_cols(g);
8     drawCanvas();
9 }
```

Comme on peut le voir nous avons décidé de mettre le jeu "g" et son nombre de lignes/colonnes en variable globale. Cela nous permet donc de les utiliser dans mes autres fonctions sans mettre de paramètre.

Les autres fonctions sont **function drawCanvas()**. Elles permettent d'afficher la grille et les images (la manière utilisée est la même que celle de SDL avec deux boucles "for").

Et enfin **function canvasLeftClick(event)** qui est utilisé si on clique avec le bouton gauche de la souris **canvas.addEventListener('click', canvasLeftClick)**; Ici aussi, l'algorithme ressemble beaucoup à celui de SDL, on détecte l'emplacement de la souris dans la case et en fonction on a les coordonnées.

## 4.2 Problèmes rencontrés

Le sujet était très clair, le premier rendu "demo" nous a permis de nous familiariser avec les langages HTML CSS et Javascript. Je tiens à souligner que les nombreux exemples, la feuille de TD et le dossier "make-game-web" avec le fichier **wrapper.c** nous a beaucoup aidé.

## 5 GitLab

Pour ce projet, nous avons utilisé GitLab pour stocker l'ensemble de nos fichiers. En debut d'année, nous avons configuré nos ordinateurs avec une clé ssh, grâce à cela, on peut cloner le projet en ssh sur notre ordinateur. C'est un système que je trouve intéressant car cela m'a permis avec mon groupe de pouvoir travailler ensemble et de voir les différentes mise à jour que chacun a apporté.

Afin d'éviter les conflits entre les différents **git push**, nous avons décidé de créer une **branch** chacun pour travailler sur nos fonctions respectives. Avant de commencer à travailler, il faut faire un **git pull** pour charger les modifications des collègues. Ensuite pour créer une branch on fait **git branch MaVersion** puis **git switch MaVersion**, je peux désormais travailler dessus. Enfin pour déposer mon travail je fais un commentaire **git commit -m "modification du fichier"** pour expliquer ce que j'ai fait, cela permettra à mes collègues de savoir ce que j'ai fait. Puis je fais **git push** pour tout déposer.

Les commandes utilisent mais que j'ai moins utilisé sont **git log** pour voir les derniers commit et **git merge** pour fusionner deux branches.

## 6 Conclusion

J'ai trouvé ce projet très intéressant à réaliser, car j'ai pu apprendre beaucoup de chose qui me servent aujourd'hui dans mes projets personnels. Le fait d'apprendre la manipulation de fichier, comment fonctionne la compilation de plusieurs fichiers nous permet de mieux comprendre comment fonctionne une machine. Le fait que le projet a du être écrit en C m'a permis de l'utiliser avec plein de technologie comme SDL, Android et le Web. C'est un projet qui a duré toute une année universitaire, il a donc fallu être rigoureux avec mon équipe. Le fait d'avoir travaillé en groupe nous a tous permis d'élaborer une certaine méthode de travail. Malgré quelques difficultés au début du projet, je suis content du rendu final.