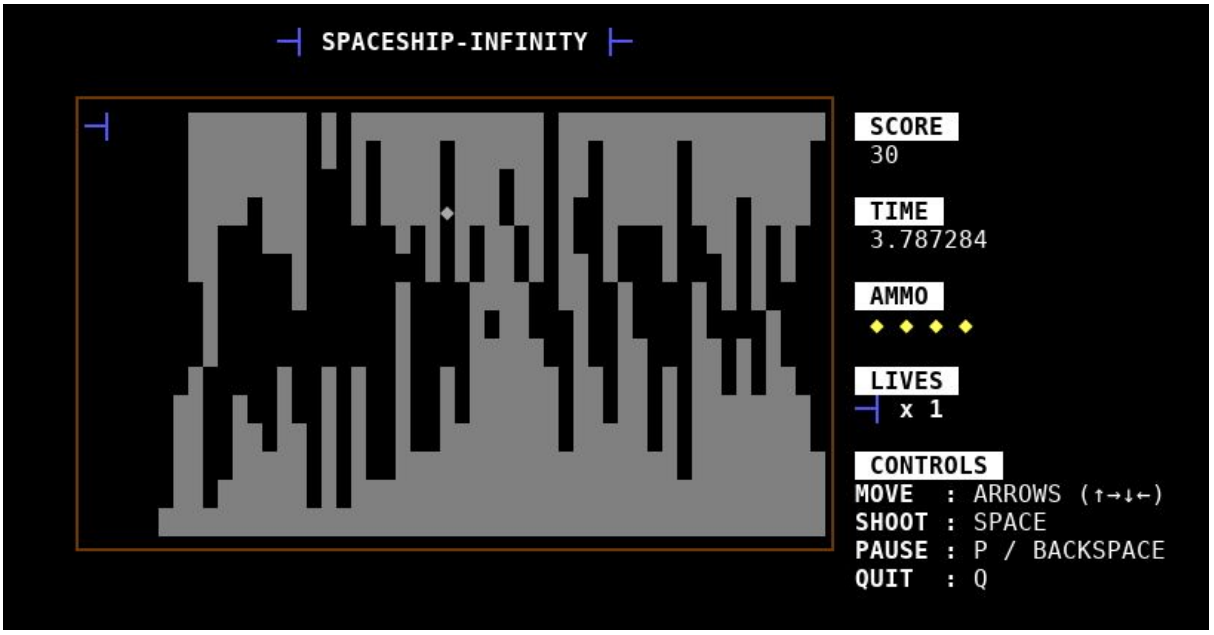
Rapport Projet Spaceship-Infinity

Structures de Données, Algorithmes



LAFORÊT Nicolas

Université de Strasbourg

Licence Informatique - Structures de Données, Algorithmes

L2S3

Introduction

Le but de ce projet est de compléter le code d'un jeu d'arcade dans lequel le joueur contrôle un vaisseau. Le vaisseau se déplace dans un monde en deux dimensions et peut y récolter divers objets : des bonus, des malus, des slots pour son arme ou bien des objets surprises. Le vaisseau peut tirer des projectiles pour détruire des éléments sur son passage.

Choix d'implémentation

Lors de ce projet, nous avons dû modifier trois fichiers afin de faire marcher le jeu Spaceship-Infinity.

Algorithme *`column_fall()`*

Cette fonction se trouve dans le fichier **`column.c`**, elle permet de faire «tomber» d'une case les cellules qui ne sont pas connectées au «plafond». Ainsi, une cellule peut tomber si et seulement si les conditions suivantes sont vérifiées :

1. Il ne s'agit pas de la première cellule de la colonne
2. Il ne s'agit pas de la dernière cellule de la colonne
3. Il existe une cellule vide entre cette cellule et la première cellule
4. Il existe une cellule vide entre cette cellule et la dernière cellule

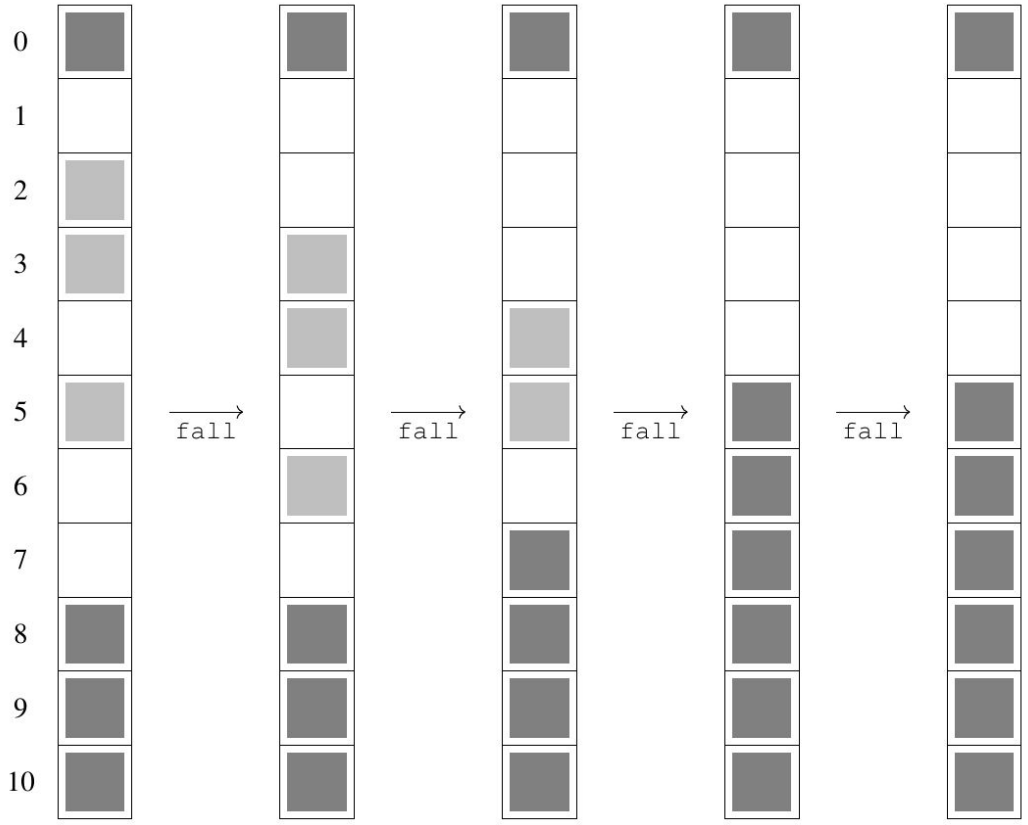


FIGURE 2 – Exemple d'appels successifs de la fonction `column_fall()` :

Dans la vidéo de démonstration du jeu disponible sur Moodle, on peut observer que même les bonus sont soumis à la gravité, mais pour des raisons de gameplay, j'ai décidé de ne pas les soumettre à la gravité. Car dans la plupart des jeux de ce style, les objets volent le plus souvent au milieu du jeu. Mais le passage de l'un à l'autre est facile à implémenter dans le code (il est en commentaire à la suite de la fonction dans le fichier **`column.c`**).

Pour implémenter la fonction ***`column_fall()`***, j'ai choisi de créer des blocs de cellules qui ne sont pas **CELL_EMPTY**. Pour cela j'ai donc créé un tableau à deux dimensions initialisé avec des coordonnées invalides :

```
int groups[c->height / 2 + 2][2];          // Création du tableau 2D
int i;
for (i = 0; i < c->height / 2 + 2; ++i) {    // Initialisation avec des coordonnées invalides
        groups[i][0] = -1;
        groups[i][1] = -1;
}
```

On choisi un tableau 2D avec la première dimension égale à la moitié de la hauteur d'une colonne + 2, car il est impossible d'avoir plus de groupes de **CELL_WALL** que c->height / 2 + 1. On fait +2 et pas seulement +1 pour pouvoir s'arrêter dans la boucle while de la troisième partie de la fonction.

Dans un second temps, on parcourt la colonne en faisant les groupes de **CELL_WALL** :

```
int index = 0;
for (i = 0; i < c->height; ++i) {
        if (column_get_cell(c, (size_t) i) == CELL_WALL) {
                groups[index][0] = i;
                while (column_get_cell(c, (size_t) i) == CELL_WALL) {
                        groups[index][1] = i;
                        i++;
                }
                index++;
        }
}
```

Si on prend la Figure 2 pour exemple, après le passage de cette boucle, on obtiendrait les groupes suivants :

groups[0][0] = 0 groups[0][1] = 0 **{0, 0}**	groups[1][0] = 2 groups[1][1] = 3 **{2, 3}**	groups[2][0] = 5 groups[2][1] = 5 **{5, 5}**	groups[3][0] = 8 groups[3][1] = 10 **{8, 10}**
groups[4][0] = -1 groups[4][1] = -1 **{-1, -1}**	groups[5][0] = -1 groups[5][1] = -1 **{-1, -1}**	groups[6][0] = -1 groups[6][1] = -1 **{-1, -1}**	

La troisième partie de la fonction consiste à faire tomber les blocs de cellules :

```
i = 0;
while (groups[i][0] != -1 && groups[i][1] != -1) {
        if ((groups[i][0] != 0)
           && (groups[i][1] != c->height - 1)
           && (column_get_cell(c, (size_t) groups[i][1] + 1) == CELL_EMPTY)) {

                column_set_cell(c, (size_t) groups[i][0], CELL_EMPTY);
                column_set_cell(c, (size_t) groups[i][1] + 1, CELL_WALL);
        }
        i++;
}
```

Pour faire tomber les blocs de cellules, on doit boucler dans tous les blocs et tester s'ils doivent tomber ou non. Comme je ne fais pas tomber les bonus, si un bloc doit tomber il suffit de mettre une **CELL_EMPTY** à la première coordonnée du groupe et de rajouter une **CELL_WALL** à la case après la dernière coordonnée du groupe.

Implémentation de *column_list*

Dans ce projet, il faut implémenter deux types de liste, le premier est **column_list** (les listes de **column**). Le terrain du jeu repose sur une liste de colonnes (**column_list**). Cette liste correspond aux colonnes affichées. Lorsque le décor bouge, il faut supprimer et ajouter de nouvelles colonnes en tête ou en queue de liste. Plusieurs mécanismes (affichage, test de collision, chute des cellules) nécessitent de pouvoir accéder à une colonne donnée à partir de son index.

Pour réaliser ces opérations, j'ai choisi d'utiliser des listes doublement chaînées afin de pouvoir naviguer dans les colonnes plus facilement. La structure **column_list** est donc la suivante :

```
struct column_list {
        column* col;
        column_list* next;
        column_list* prev;
};
```

Toutes les fonctions à implémenter étaient sensiblement les mêmes que celles du TP6, on a donc dû rajouter deux fonctions auxiliaires afin de faciliter l'accès au début et à la fin des listes.

```
static inline column_list* column_list_start(const column_list* const l);
static inline column_list* column_list_end(const column_list* const l);
```

Il ne faut pas oublier de détruire les colonnes elles-mêmes dans la fonction ***column_list_destroy()*** à l'aide de la fonction ***column_destroy()***.

Fonctions Projet Spaceship-Infinity	Fonctions TP6
`column_list_new()`	`liste_vide()`
`column_list_destroy()`	`liste_free()`
`column_list_get_column()`	
`column_list_get_size()`	`list_longueur()`
`column_list_push_front()`	`list_insertion_tete()`
`column_list_push_back()`	`list_insertion_queue()`
`column_list_pop_front()`	`list_suppression_tete()`
`column_list_pop_back()`	`list_suppression_queue()`
`column_list_start()`	`list_debut()`
`column_list_end()`	`list_fin()`

Implémentation de ***`column_list_get_column()`*** :

```
column* column_list_get_column(const column_list* const l, const size_t i) {
  column_list* tmp = column_list_start(l);
  for (size_t index = 0; index < i; ++index)
          tmp = tmp->next;

  return tmp->col;
}
```

Implémentation de *point_list*

Au cours de la partie, le joueur peut décider de tirer. Pour stocker la position des projectiles en col, une liste de points (**point_list**) est nécessaire. À chaque «tour», tous les projectiles se déplacent vers la droite. De même, le niveau recule lorsque le joueur recule trop (en difficulté 0) et il faut déplacer les projectiles en conséquence. Le code fourni suppose que ces opérations seront réalisées dans les fonctions ***point_list_shift_left()*** et ***point_list_shift_right()***.

Pour réaliser ces opérations, j'ai choisi d'utiliser des listes doublement chaînées comme pour les listes **column_list**. La structure **point_list** est donc la suivante :

```
struct point_list {
	point pt;
	point_list* next;
	point_list* prev;
};
```

Tout comme pour les colonnes, la plupart des fonctions sont les mêmes que celles du TP6. Pour les autres fonctions, nous allons voir les implémentations fonction par fonction.

```
bool point_list_contains(const point_list* const l, point p) {
	bool contains = false;
	const point_list* tmp;
	for (tmp = point_list_start(l); tmp; tmp = tmp->next) // Boucle dans tous les points
		if ((tmp->pt.x == p.x) && (tmp->pt.y == p.y))  // Test si égal au point p
			contains = true;

	return contains;
}

void point_list_set_point(point_list* const l, size_t i, point p) {
	point_list* tmp = point_list_start(l);
	for (size_t index = 0; index < i; ++index)	// On trouve le ième point
		tmp = tmp->next;
	tmp->pt = p;					// On le remplace par p
}

void point_list_shift_left(point_list* const l) {
	const point_list* tmp;
	size_t i = 0;
	// Boucle pour décaler tous les points vers la gauche
	for (tmp = point_list_start(l); tmp; tmp = tmp->next) {
		point_list_set_point(l, i, point_xy(tmp->pt.x-1, tmp->pt.y));
		i++;
	}
}
```

```
void point_list_shift_right(point_list* const l) {
	const point_list* tmp;
	size_t i = 0;
	// Boucle pour décaler tous les points vers la droite
	for (tmp = point_list_start(l); tmp; tmp = tmp->next) {
		point_list_set_point(l, i, point_xy(tmp->pt.x+1, tmp->pt.y));
		i++;
	}
}
```

Par ailleurs, il arrive que les positions de certains projectiles ne correspondent plus :

- lorsqu'un projectile explose, le jeu remplace sa position par un point «invalide» (voir le fichier `point.h`).
- lorsqu'un projectile sort des limites du jeu.

Pour que le jeu puisse gérer ces cas, il faut implémenter la fonction ***`point_list_out_of_bounds()`*** pour qu'elle supprime d'une liste toutes les positions qui correspondent à des points en dehors des limites du jeu.

Pour implémenter cette fonction, j'ai choisi de diviser la fonction en deux parties (pour des raisons qui seront expliquées dans la partie **Difficultés rencontrées**). La première partie consiste à mettre tous les points qui sortent de la zone de jeu en position invalide.

```
point_list* tmp = point_list_start(l);
for (size_t i = 0; tmp; ++i) {
	if (!point_is_in_rectangle(tmp->pt, up_left, bottom_right))
		point_list_set_point(tmp, i, point_invalid());
	tmp = tmp->next;
}
```

Et ensuite la deuxième partie permet de supprimer les points qui sont en position invalide. Il faut faire attention à gérer tous les cas, pour pouvoir supprimer n'importe quel point à n'importe quel indice dans la liste de point.

```
tmp = point_list_start(l);
while(tmp) {
	if (!point_is_valid(tmp->pt)) {
		if (!(tmp->prev) && tmp->next) {
			tmp = tmp->next;
			free(tmp->prev);
			tmp->prev = NULL;
			return tmp;
		} else if (tmp->prev && !(tmp->next)) {
			tmp = tmp->prev;
			free(tmp->next);
			tmp->next = NULL;
			return tmp;
		} else if (tmp->prev && tmp->next) {
			point_list* tmp2 = tmp;
			tmp = tmp->prev;
			tmp->next = tmp2->next;
			tmp2->next->prev = tmp;
			free(tmp2);
		} else {
			free(tmp);
			return NULL;
		}
	} else {
		tmp = tmp->next;
	}
}
```

Difficultés rencontrées

La plus grande difficulté rencontrée a été dans l'implémentation de la fonction ***`point_list_out_of_bounds()`***.

Pour l'implémentation de cette fonction, je voulais d'abord supprimer les points placés en position invalides suite à une collision avec un mur ou un objet et les points qui sortent de la zone de jeu en une seule fois. Mais suite à des tests après cette implémentation, le jeu avait des ***Segmentation Fault*** lorsqu'en difficulté 0, on tirait plusieurs projectiles au milieu du jeu et qu'on allait très vite vers la gauche. Car les projectiles qui sortaient de la zone de jeu réapparaissaient de l'autre côté du terrain, puis au prochain appel de ***`game_compute_turn()`***, le programme s'arrêtait avec un ***Segmentation Fault***.

Le reste des difficultés étaient dans l'implémentation des fonctionnalités supplémentaires. Notamment dans la gestion du Menu Pause et la durée du laser, mais nous verrons ces points dans les paragraphes suivant.

Fonctionnalités supplémentaires

L'implémentation du jeu de base a été assez rapide, c'est pourquoi j'ai pu prendre le temps d'ajouter plusieurs fonctionnalités supplémentaires. Nous allons voir ces fonctionnalités dans l'ordre dans lequel je les ai implémentées.

Queue vulnérable `--tail`

Cette fonctionnalité s'active à l'aide de l'option de compilation **`--tail`**. Pour ce faire j'ai du changer la structure **`spaceship_options`** en ajoutant le booléen **`tail`**, dont la valeur par défaut est **`false`**. Il a fallu aussi ajouter des références à cette fonctionnalité dans toutes les structures et fonctions des fichiers **`options.c`** et **`options.h`**, ainsi que dans l'affichage de la fonction **`print_help()`**.

Cette option permet d'étendre la hitbox du vaisseau non seulement à sa tête mais aussi à sa queue. Car dans la version fournie, le sprite du vaisseau recouvre deux cases alors que sa hitbox ne se trouve que sur sa tête.

Il faut tout de mettre faire attention car, le sprite du **vaisseau** ne recouvre **qu'une seule case** lorsque l'option **`pretty`** est mise à **`false`**. Il ne faut donc pas changer la taille de la hitbox du vaisseau dans ce cas là. Ainsi, dans la suite des explications, les changement n'apparaissent que lorsque **`pretty=true`**.

L'implémentation a été assez simple, il a suffit d'ajouter des tests sur la cellule de la queue du vaisseau dans la fonction ***`game_check_hit()`***. Sans oublier d'empêcher certains déplacements du vaisseau si cette option est activée. En effet, la queue du vaisseau ne peut pas se trouver dans la bordure de gauche du terrain lorsque cette option est activée.

De plus, dans la version fournie, on ne peut pas se déplacer dans un mur, c'est forcément le mur qui se déplace dans le vaisseau lors d'une collision. C'est pourquoi j'ai aussi empêché le fait de se déplacer avec la queue du vaisseau dans les murs.

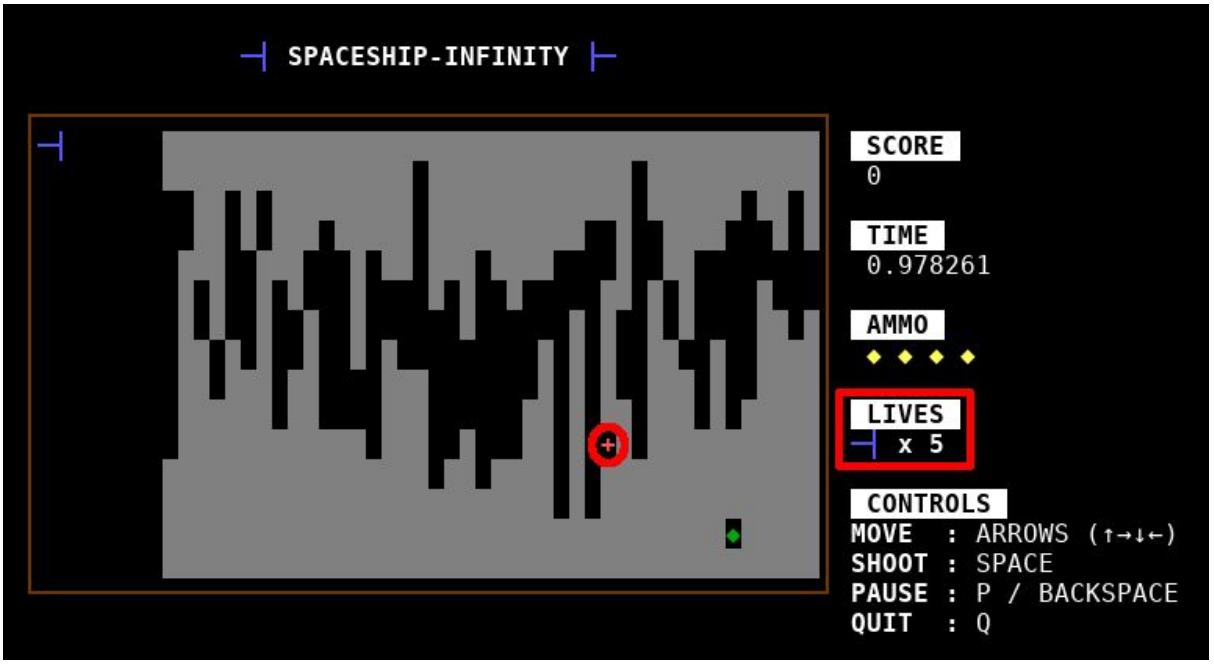
Le dernier point à ajouter est la possibilité de récupérer les objets avec la queue du vaisseau, cela se fait en ajoutant un test sur la cellule de la queue du vaisseau dans la fonction ***`game_check_special_cells()`***.

Friendly Fire `--friendly`

Cette fonctionnalité s'active à l'aide de l'option de compilation **`--friendly`**. Pour ce faire j'ai du changer la structure **`spaceship_options`** en ajoutant le booléen **`friendly`**, dont la valeur par défaut est **`false`**. Il a fallu aussi ajouter des références à cette fonctionnalité dans toutes les structures et fonctions des fichiers **`options.c`** et **`options.h`**, ainsi que dans l'affichage de la fonction **`print_help()`**.

Cette option permet aux projectiles tirés par le joueur d'infliger des dégâts au vaisseau s'ils se trouvent à la même position. Ce test se fait dans la fonction ***`game_check_hit()`***. Comme j'avais déjà implémenté l'option pour rendre la queue du vaisseau vulnérable, il faut aussi tester si le projectile touche la queue du vaisseau lorsque cette option est activée (seulement si **`pretty=true`**).

Ajout de vie `--lives=<value>`



Cette fonctionnalité s'active à l'aide de l'option de compilation **`--lives=<value>`**, où **`value`** est un nombre entier. Pour ce faire j'ai du changer la structure **`spaceship_options`** en ajoutant un intmax_t **`lives`**, dont la valeur par défaut est **`1`**. Il a fallu aussi ajouter des références à cette fonctionnalité dans toutes les structures et fonctions des fichiers **`options.c`** et **`options.h`**, ainsi que dans l'affichage de la fonction **`print_help()`**.

Si à la compilation on entre une valeur inférieur à 1, le nombre de vie sera remis à 1 par la fonction **`check_long_options()`**. De même, si la valeur est supérieure à 9, elle sera remise à 9, pour empêcher de commencer avec un nombre de vie trop important.

Le nombre de vie courant pendant la partie est stocké dans la variable intmax_t **`lives`** créée dans la structure **`game`**.

Pour implémenter cette fonctionnalité, j'ai dû ajouter quelques fonctions dans les fichiers **`options.c`** et **`options.h`**, permettant d'ajouter des vies, d'en enlever, de récupérer le nombre de vies courant.

Le fait de perdre une vie se déroule dans la fonction **`interface_game_loop()`**. Si **`game_check_hit(g)`** est **`true`** alors on met le jeu en pause, on appelle la fonction **`game_was_hit()`** qui permet de supprimer toutes les colonnes afin de ne pas perdre toutes ses vies dans le même mur si le nombre de vies est supérieur à 1, on perd aussi 100 points. De plus, cette fonction supprime tous les projectiles qui ont été tirés et fait perdre une vie. Après avoir fait toutes ces opérations, si le nombre de vie n'est pas égal à 0, on attend 1 seconde avant de remettre la partie en route (avec l'option still de la partie courante).

J'ai aussi décidé d'ajouter un nouveau type de cellule dans le fichier **`cell.h`** : CELL_LIFE, qui permet de gagner une vie lorsqu'on passe dessus, dont le sprite est "**+**". Pour qu'elle apparaisse dans le jeu, il a fallu l'ajouter dans la création de nouvelles colonnes, fonction **`terrain_new_column()`**, j'ai choisi de mettre un taux d'apparition de 15%. Il y a aussi un taux de 15% qu'une vie se cache dans une CELL_SECRET. On ajoute une vie grâce à la fonction **`game_add_life()`**.

Menu Pause / Options

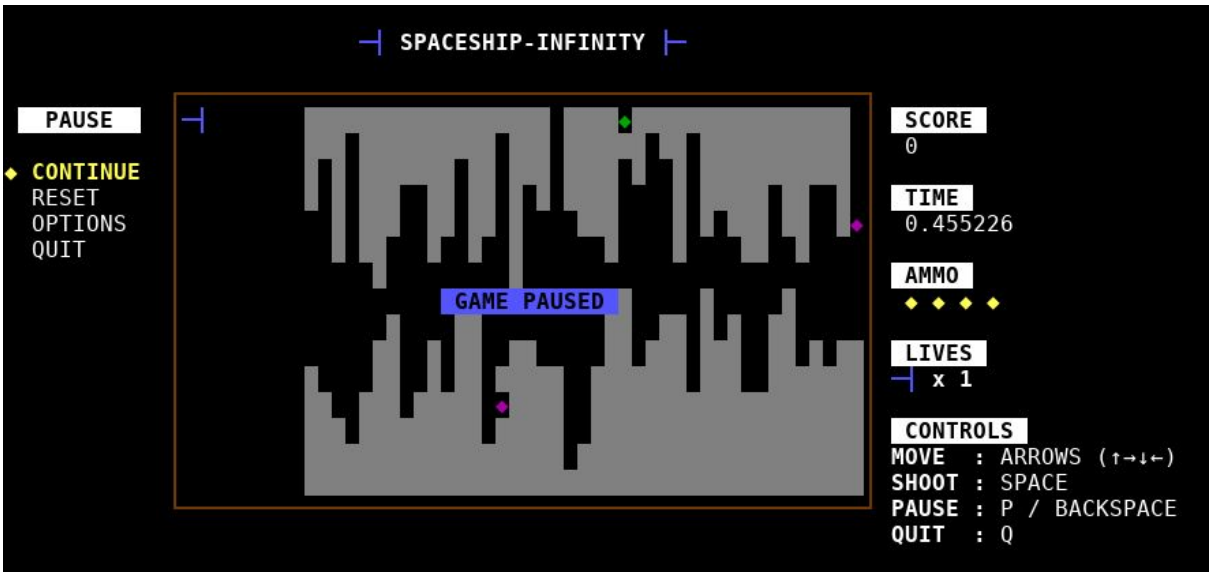
Je voulais pouvoir mettre le jeu en pause et avoir plusieurs choix qui allaient influer sur le jeu, c'est pourquoi je me suis dis qu'un menu pause (avec la possibilité de changer des options en plein milieu d'une partie) était nécessaire.

Pour cela, j'ai dû changer la structure interface dans le fichier ui.c, en ajoutant une nouvelle fenêtre (WINDOW*), où j'afficherai ce Menu Pause et Options. J'ai aussi ajouté par la suite un entier (**int** quit) pour pouvoir quitter le jeu sans appuyer sur la touche "q" dans le menu Pause, mais seulement en choisissant l'option "QUIT". De plus, j'ai ajouté un entier (**int** pause) dans la structure game, pour savoir quel choix on aura sélectionné dans le Menu en lui-même.

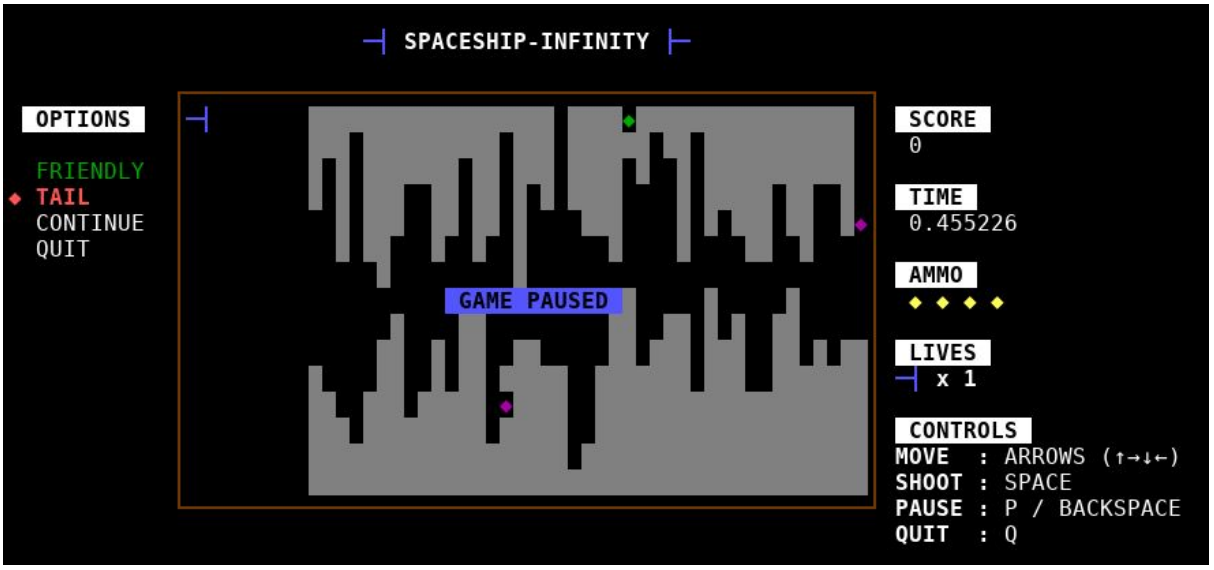
La gestion du Menu a été l'élément le plus compliqué de ce Projet, car lorsqu'on appuie sur la touche "p" ou "backspace" pour afficher le Menu, il fallait mettre le jeu en pause mais aussi le Timer sur le côté. Sinon, la vitesse du jeu aurait augmentée pendant la pause, et on aurait gagné des points pour avoir survécu un certain temps erroné. Sauf que ça ne doit pas se passer comme cela, quand le jeu est en pause, rien ne doit faire changer ces éléments.

La gestion du Menu Pause et Options se déroule dans les fonctions **`interface_game_loop()`**, **`game_pause_menu()`**, **`game_pause_process_input()`** et **`game_pause()`**.

- **`game_pause_menu()`** : renvoie l'indice du choix dans le menu.
- **`game_pause_process_input()`** : incrémente l'indice du choix dans le menu.
- **`game_pause(game* g, int p)`** : met l'indice du choix du menu à la valeur de p.

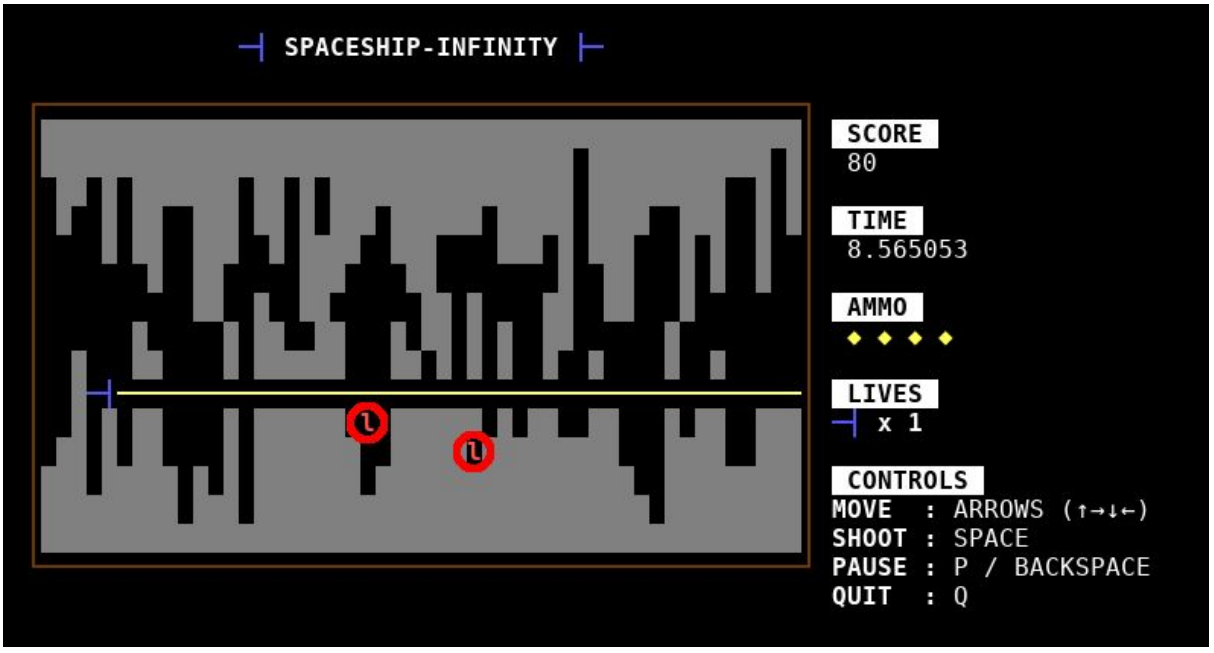


CONTINUE : Continue la partie.
RESET : Redémarre le jeu avec les valeurs réinitialisées.
OPTIONS : Ouvre le Menu des Options.
QUIT : Quitte le jeu (sans avoir à appuyer sur "q" une deuxième fois).



FRIENDLY : Active/désactive le Friendly Fire (Rouge = désactivé ; Vert = activé).
TAIL : Active/désactive la queue vulnérable (Rouge = désactivé ; Vert = activé).
CONTINUE : Continue la partie.
QUIT : Quitte le jeu (sans avoir à appuyer sur "q" une deuxième fois).

Laser `--laser=<value>`



Cette fonctionnalité s'active à l'aide de l'option de compilation **`--laser=<value>`**, où **`value`** est un nombre entier. Pour ce faire j'ai du changer la structure **`spaceship_options`** en ajoutant un intmax_t **`laserTime`**, dont la valeur par défaut est **3** (secondes). Il a fallu aussi ajouter des références à cette fonctionnalité dans toutes les structures et fonctions des fichiers **`options.c`** et **`options.h`**, ainsi que dans l'affichage de la fonction **`print_help()`**.

Si à la compilation on entre une valeur inférieur à 1, le nombre de vie sera remis à 1 par la fonction **`check_long_options()`**. De même, si la valeur est supérieure à 9, elle sera remise à 9, pour limiter la durée du laser.

J'ai ajouté une variable bool **`laser`** dans la structure **`game`**, afin de savoir si le laser est activé ou non.

Pour pouvoir activer le laser, j'ai décidé d'ajouter un nouveau type de cellule dans le fichier **`cell.h`** : CELL_LASER, dont le sprite est " ℓ ". Pour qu'elle apparaisse dans le jeu, il a fallu l'ajouter dans la création de nouvelles colonnes, fonction **`terrain_new_column()`**, j'ai choisi de mettre un taux d'apparition de 10%. Il y a aussi un taux de 10% qu'un laser se cache dans une CELL_SECRET. On active le laser grâce à la fonction **`game_set_laser(game* g, bool active)`**.

L'affichage du laser et la destruction des cellules par le laser se fait dans la fonction **`_display_game()`** car cela est plus rapide vu que le laser détruit toutes les cellules.

Nouvelles Fenêtres (WINDOW*)

Pour des raisons de cosmétique et de gameplay, j'ai décidé d'ajouter de l'affichage supplémentaire. C'est pourquoi j'ai ajouté le titre "**SPACESHIP-INFINITY**", centré au dessus du terrain de jeu. Pour cela, j'ai dû ajouter une variable **WINDOW* title_window** afin de créer une nouvelle fenêtre avec la librairie ncurses, pour afficher ce titre.

J'ai aussi ajouté une variable **WINDOW* pause_window**, comme nous l'avons vu auparavant, pour afficher le Menu Pause.

De plus pour des raisons de gameplay, j'ai affiché les touches qui permettent de jouer au jeu, afin que l'utilisateur puisse plus facilement s'y retrouver.

Conclusion

C'est un Projet que j'ai particulièrement apprécié de par la liberté qu'il nous laissait une fois les fonctions nécessaires au jeu de base implémentées. En effet, ces fonctions ont très vite été codées, j'ai donc pû prendre le temps de lire plus en profondeur le code fourni. Ainsi, je me suis mis à imaginer comment je pouvais améliorer l'expérience de jeu en ajoutant des fonctionnalités supplémentaires.

C'est dans ces ajouts que j'ai pu apprendre et me débrouiller seul sur l'implémentation de fonctions à l'aide de librairie que je ne connaissais que très peu. Je me rends bien compte que mon code n'est certainement pas optimisé pour les fonctionnalités supplémentaires, mais j'ai réussi à faire ce que j'avais en tête, sans trouver de bug majeur, ni de Segmentation Fault.

J'ai déposé ce Projet une première fois sur moodle la veille du rendu, sauf que dans la soirée je me suis rendu compte que j'avais toujours un petit bug dans mon jeu. En effet, lorsqu'on ramassais un laser et qu'on mettait le jeu en pause pendant que le laser était activé, le temps du laser continuait de s'écouler. J'ai donc corrigé ce bug le lendemain.

Pour finir, c'est avec une certaine satisfaction d'avoir réalisé ce que je voulais que je rends ce Projet, je continuerais de modifier mon code même après ce rendu pour ajouter d'autres fonctionnalités pour le plaisir.