

# RAPPORT PROJET C



FLORIAN TALOUR  
PAUL BAUDY  
ELODIE COCQ

# SOMMAIRE

I.	Le projet.....	1
	A. Cahier des charges.....	1
	B. Répartition des tâches de travail.....	2
II.	Les sources et la compilation.....	3
	A. Bibliothèques utilisées.....	3
	B. Arborescence des dossiers.....	3
	C. Commandes de compilation.....	4
III.	Les choix algorithmiques.....	4
	A. Les structures.....	4
	B. Les sets.....	6
	C. Les fonctions.....	6
	D. Un programme modulable.....	8
IV.	Les difficultés rencontrées.....	9
	A. Les changements globaux.....	9
	B. Les allocations de la mémoire.....	9
V.	Les annexes.....	10
	A. Le menu.....	10
	B. Le mode histoire.....	11
	C. Le mode multijoueurs.....	11

## I. Le projet

### A. Cahier des charges

**T**out d'abord, nous avons choisi de créer un Bomberman. Le principe de base d'un tel jeu est assez simple: on incarne un personnage sur une carte, qui doit éliminer ses adversaires grâce aux bombes dont il dispose dans un temps imparti. Pour ce faire, il devra exploser les briques qui le séparent de ceux-ci avant d'espérer une victoire, sans mourir suite à l'explosion d'une bombe.

**N**ous avons mis en place un système de manches au sein d'une partie pour rendre le jeu plus compétitif. Ainsi, le gagnant de la partie sera le joueur ayant le score le plus élevé, c'est-à-dire celui qui a été le plus souvent victorieux dans l'ensemble des manches. Le nombre de manches d'une partie est réglable, tout comme le nom des deux joueurs contrôlables grâce au clavier.

**C'**est en naviguant à travers le menu du jeu qu'on peut accéder à l'ensemble de ses fonctionnalités. Au lancement du Bomberman, le menu principal nous offre la possibilité de modifier le volume sonore des bruitages par l'intermédiaire des options. Nous avons de plus jugé utile d'informer l'utilisateur des contrôles dans l'onglet Aide. Il est possible de lancer une partie "multijoueurs", à 1 ou 2 joueurs contre l'ordinateur, ou encore de commencer le mode histoire.

**N**ous avons décidé d'ajouter une campagne pour enrichir le contenu du jeu. Les règles sont expliquées dans le menu de l'histoire mais le principe initial est de ramasser des fruits, gardés par des monstres qu'il ne faut pas toucher pour survivre. On considère que le joueur ne peut passer au niveau suivant uniquement s'il a réussi à ramasser la totalité des fruits présents sur le terrain, sans n'exploser aucune des cases spéciales dans le temps imparti. Un dur combat pour garder une alimentation saine et équilibrée! Un joueur solo peut donc tenter d'arriver au bout des 5 niveaux actuellement disponibles de l'histoire pour en sortir gagnant.

**T**outefois, un niveau final contenant un ultime ennemi dans la campagne était initialement prévu, mais le temps disponible n'a pas été suffisant à son élaboration. De la même façon, il n'y a aucune sauvegarde de l'avancement dans le mode histoire, ni de score en fonction du temps ou du nombre de morts. Nous aurions aussi aimé pouvoir afficher un tableau des meilleurs scores, tant pour la campagne que pour le multijoueurs.

**A**ussi, nous aurions pu élaborer différentes cartes en mode multijoueurs, que l'utilisateur aurait pu sélectionner dans les réglages ou au lancement d'une partie. Une personnalisation du joueur encore plus approfondie aurait pu être disponible, comme le choix de l'avatar ou de sa tête dans l'interface par exemple. Les sons auraient pu être plus présents ou différents selon les personnages et réglages, au choix de l'utilisateur. D'autres bonus auraient pu voir le jour, comme d'autres types de bombes.

De plus, l'intelligence artificielle, bien que fonctionnelle, n'est pas parfaite. Celle-ci est capable de poser des bombes lorsqu'elle se retrouve devant une brique ou sur un autre joueur et d'en esquiver l'explosion. Bien qu'elle cherche à se rapprocher du joueur le plus proche d'elle, l'IA n'a pas été assez perfectionnée pour évaluer le meilleur chemin afin d'échapper à une explosion. En effet, elle cherchera par défaut la première direction disponible, même si cela la conduit à être condamnée par un nouveau danger. Encore une fois, le manque de temps nous a conduit à rester sur une IA simple, capable de survivre le plus longtemps possible sans anticiper ces mouvements par rapport à l'ensemble du terrain.

Nous n'avons pu gérer proprement les appuis sur la touche Echap: nous aurions aimé faire apparaître un menu Pause qui permettrait à l'utilisateur de quitter la partie ou simplement figer le jeu un instant.

## B. Répartition des tâches de travail

Pour commencer, nous avons réfléchi ensemble afin d'élaborer un cahier des charges à notre projet. Nous avons ensuite noté l'ensemble des tâches à accomplir, dans un ordre de priorité précis, dans un document partagé en ligne. Nous avons donc pu suivre l'évolution de notre projet ainsi que les tâches accomplies et ajoutées, au fur et à mesure.

Après avoir mis en place la base du projet, avec les structures, carte et fonctions principales comme le déplacement ou la pose de bombe, nous avons décidé de dispatcher le travail. Florian s'est vu attribuer la mise en place de la campagne tandis que les autres gèreraient le reste. Le travail n'en a été que facilité en évitant les conflits sur les fichiers sources. Toutefois, il fallait rester conscient des avancées mutuelles pour aider en cas de problème, harmoniser certaines fonctions ou affichage, comme l'interface en jeu, et réutiliser les fonctions déjà prêtes et abouties.

Pour gérer au mieux notre projet, nous avons aussi noté dans le document partagé les différentes versions du programme, jointes à l'espace partagé, avec les ajouts et modifications associées. Ceci devait permettre de maintenir le contact entre les deux groupes, en dehors des séances de TP organisées et garder ainsi une cohérence des différents codes. Cependant, même avec un tel document, certaines informations n'ont pas été assez bien transmises puisque certaines fonctions presque identiques sont présentes dans notre programme, en multi et en campagne. En effet, après lecture de l'intégralité du code, nous nous sommes rendu compte que certaines fonctions auraient pu être regroupées en une unique. Ajouter simplement un paramètre supplémentaire indiquant s'il s'agissait d'un appel dans la campagne ou dans le multi aurait suffi. Des optimisations de fonctions sont donc bien évidemment possibles.

## II. Les sources et la compilation

### A. Bibliothèques utilisées

Nous avons choisi d'utiliser la SDL 1.2 comme interface graphique, en y ajoutant les bibliothèques permettant d'afficher des images, [SDL-image](#), ainsi que du texte, [SDL-TTF](#). Pour dynamiser le contenu, nous avons de plus intégré la bibliothèque [SDL-mixer](#) pour insérer des sons. Nous avons inséré les liens de téléchargement de ces bibliothèques sur Windows, mais voici la liste des bibliothèques utilisées à récupérer sur Linux via le gestionnaire de paquets APT:

- ⊙ `libsdl1.2-dev`
- ⊙ `libsdl-ttf2.0-0` (Pour SDL 1.2)
- ⊙ `libsdl-ttf2.0-dev` (Pour SDL 1.2)
- ⊙ `libsdl-image1.2`
- ⊙ `libsdl-image1.2-dev`
- ⊙ `libsdl-mixer1.2`
- ⊙ `libsdl-mixer1.2-dev`

Nous avons de plus inclus les headers "math.h" pour calculer des distances à l'aide de la racine carrée et "time.h" pour la génération aléatoire des blocs grâce à la fonction `rand()`.

### B. Arborescence des dossiers

#### Racine

- `/src`: contient l'ensemble des `.c` et `.h` du projet
- `makefile`: mis à disposition pour compiler le programme
- `/data`
  - `/sons`: contient notre bibliothèque de sons utilisés
  - `/sets`: contient les regroupements d'images des blocs des maps et des animations des personnages
  - `/maps`: contient les fichiers texte de création des cartes fixes de la campagne
  - `/images`: contient l'ensemble des images utilisées pour l'interface, du menu jusqu'à l'affichage du résultat des parties
  - `icone.png`: icône associée à la fenêtre du jeu
  - `ARJulian.ttf`: police utilisée

## C. Commandes de compilation

Un Makefile est mis à disposition à la racine du projet pour compiler le programme. La commande “make” crée l'exécutable, il ne reste plus qu'à utiliser “./bomberman” pour lancer le jeu. Il est également possible de se servir de “make clean” pour supprimer les fichiers générés.

Pour jouer sur Linux, il faut penser à mettre le clavier en anglais, c'est-à-dire en qwerty pour que les contrôles indiqués soient corrects.

## III. Les choix algorithmiques

### A. Les structures



#### Joueurs

La structure `t_joueur` contient toutes les variables relatives à celui-ci, dont son rendu visuel avec avatar et face ainsi que ses coordonnées. Chaque type de joueur possède ses propres contrôles pour qu'ils soient gérés indépendamment les uns des autres.










Avatar contient l'ensemble des images pour l'animation du personnage, et face celle de l'interface. On sélectionne l'avatar approprié selon l'animation désirée, mouvement ou mort, grâce aux coordonnées de `src`.

On utilise la dimension d'une tuile, case de la carte, pour déterminer la hitbox du personnage, ci-contre en rouge. On se sert alors de `DEPASSEMENT_TETE` pour réajuster la hitbox en fonction de la taille du personnage, contenu dans `src.h`. On adapte cette hitbox pour faciliter le mouvement en jeu.

#### Bombes

Une bombe est définie par la structure `t_bombe`, contenant ses coordonnées et autres informations qui lui sont relatives, dont `mapb`. Initialement, nous n'utilisons pas cette dernière, cependant nous nous sommes vite rendu compte qu'il serait indispensable d'enregistrer l'état d'avancement de l'explosion. De plus, elle nous permet d'enregistrer les limites de l'explosion, bloquée par les blocs durs, et donc d'optimiser le programme. Le tableau `mapb` définit les images, tiles, qu'il faut sélectionner pour l'animation de l'explosion. Il est alloué dynamiquement en fonction de la taille de l'explosion que la bombe produira. Une telle allocation est efficace pour n'importe quel type ou taille de bombe.

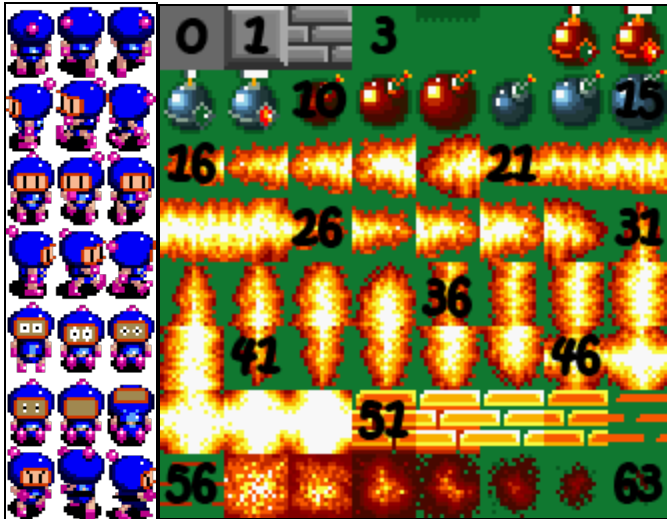
On aurait pu allouer uniquement 7 cases pour chaque image différente, cependant, cette méthode n'aurait pas pu gérer les éventuels événements spécifiques, comme l'explosion d'un bonus dans l'axe. On aurait voulu joindre les explosions entre-elles lorsqu'elles se touchent, mais le temps ne nous a pas été suffisant pour gérer ces cas-ci. Toutefois, nous avons lancé l'explosion d'une bombe si celle-ci est touchée par une autre.

	mapb[0]	tile associée		tile associée	mapb[1]	
centre	0				0 à w/2-2	gauche
haut	1 à h/2-1				w/2-1	
	h/2				w/2 à w-3	droite
bas	h/2+1 à h-2				w-2	
	h-1					

Ci-dessus la structure choisie pour garder en mémoire l'état de l'explosion. Par exemple, mapb[0] correspond aux images verticales, avec mapb[0][0] le centre de l'explosion. On retrouve aussi l'extrémité haute en mapb[0][h/2] et basse en mapb[0][h-1], séparées par autant d'images d'explosion verticale qu'il en faut pour remplir le tableau alloué en fonction de la hauteur de la bombe.

On place l'ensemble des bombes dans une file. On pourra donc regarder le sommet de la file pour déterminer les explosions et ainsi supprimer toute bombe dont l'explosion est terminée. On ajoute chaque bombe posée à la fin de la file.

## B. Les sets



On place dans le tableau map les numéros des tuiles à afficher. Par la suite, pour pouvoir récupérer la bonne tuile, il faut moduler par rapport à NB\_TUILES, le nombre de tuiles par ligne de l'image set.

Nous ne nous sommes pas simplement contentés de récupérer des images sur internet: nous les avons retravaillées, redimensionnées en 32 pixels en largeur, et ordonnées de façon à optimiser l'ordre des animations, des flammes de l'explosion des bombes aux mouvements des personnages. Il est désormais très facile d'ajouter ou modifier n'importe quelle image de n'importe quel set, ce qui nous offre des grandes possibilités d'expansion du jeu.

## C. Les fonctions

### Chargement des cartes

La fonction de chargement des cartes de l'histoire, `load_fichier_campagne`, permet de les charger depuis un fichier texte. Celui-ci est préalablement rempli en y plaçant les valeurs des cases que l'on veut, ordonnées de la même façon que ce que l'on cherche à afficher en jeu. D'autres méthodes de création de cartes existent, par le biais d'algorithmes plus ou moins complexes. Pour la campagne, il est plus judicieux de préparer des cartes statiques, qui permet de visualiser aisément l'allure générale et donc de déterminer l'emplacement idéal des cases spéciales, placées ultérieurement.

À l'inverse, la fonction de chargement de la carte multijoueurs, `load_map_1`, utilisée est définie par un algorithme simple pour obtenir un quadrillage en blocs durs. On y ajoute aléatoirement des blocs cassables avec une fréquence d'apparition contrôlée par une constante globale. On prend garde à ne pas faire apparaître de blocs cassables autour des points d'apparition des joueurs.



### Collision des joueurs

Pour déterminer si un joueur peut avancer dans la direction choisie, on sélectionne les deux coins du personnage associés à cette direction. Par exemple, pour se diriger vers la droite, on vérifie si un des coins droits de la hitbox du personnage qui prend en compte `DEPASSEMENT_TETE`, va toucher un bloc dur. On autorise une certaine marge quant au déplacement afin de le rendre plus tolérant. Si c'est cette marge qui autorise le déplacement, on repositionne correctement le personnage dans l'axe.

### Gestion des bombes

Un joueur ne peut poser qu'une seule bombe à la fois. Il doit ensuite attendre un temps donné avant d'être autorisé à en poser une nouvelle, déterminé par `wait_bombe`. Si il n'y a aucune autre bombe à cet endroit, le joueur pourra poser sa bombe et sera alors repositionné sur celle-ci. On évite ainsi au personnage de se retrouver bloqué par sa propre bombe lorsqu'il vient à l'instant de la lâcher. Sinon, toute bombe est considérée comme un bloc dur et est donc de ce fait infranchissable.

Une fois la bombe posée, elle est ajoutée à la fin de la file des bombes de la manche en cours avec l'ensemble de ses caractéristiques. De façon générale, on alloue les emplacements mémoire en définissant proprement la taille dont on a besoin. Après utilisation, on libère tous ces emplacements pour ne pas avoir de fuites mémoire. Cependant, pour les recherches de bombes, on alloue une variable temp qui parcourt l'ensemble de celles-ci. Il faut alors faire attention à ne pas libérer ce temp tant qu'il n'a pas fini son parcours: on supprimerait l'une des bombes de la file, et détruirait de ce fait le lien avec les suivantes!

### L'intelligence artificielle

Le mouvement des monstres de la campagne est géré par `controle_monstre`, qui détermine selon son environnement la direction que le monstre doit suivre. Différents cas sont gérés, suivant le nombre de blocs durs dont il est entouré, mais le principe reste simple: il suit le chemin disponible tant qu'il n'est pas confronté à un mur et fait demi-tour lorsqu'il détecte une explosion devant lui.

L'IA des ordinateurs en mode multijoueurs est, quant à elle, un peu plus développée. Elle vérifie l'emplacement des autres joueurs et tente de s'en rapprocher, tout en esquivant les bombes et respecte ainsi le cahier des charges.

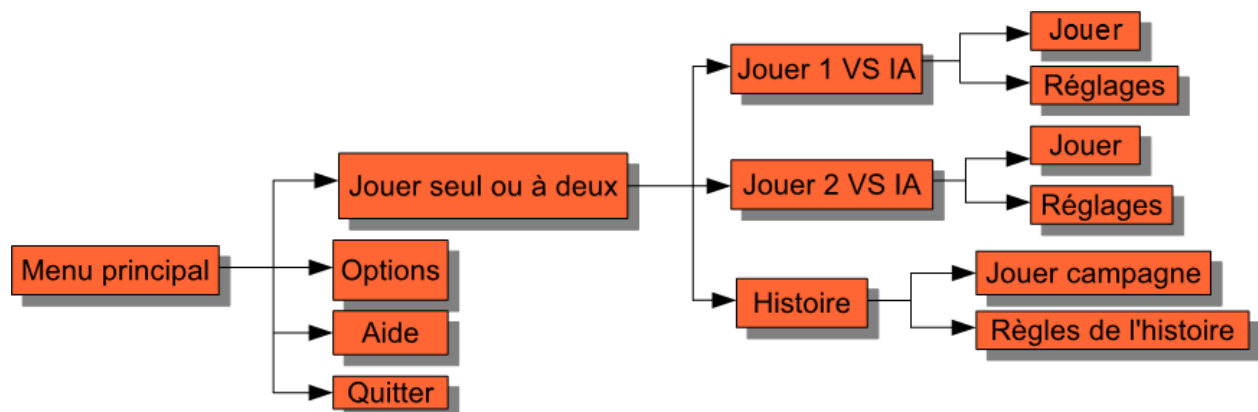
### Cases spéciales de la campagne

Tout d'abord, il faut initialiser la position des fruits et autres cases spéciales. Théoriquement, nous aurions pu les inclure dans les fichiers de création des cartes mais il aurait alors fallu modifier la structure de ceux-ci. En effet, avec plus de 9 valeurs différentes, il n'est pas possible d'en sélectionner une unique avec un seul chiffre! Complexifier la structure de ces documents nous a paru inutile dans notre cas: nous avons donc une fonction annexe plaçant ces autres cases par la suite, aux coordonnées passées en argument.

Ensuite, le programme vérifie si le joueur passe sur une case spéciale, un fruit ou une cible, et met alors la partie à jour grâce à `verif_event`. S'il active une cible, celle-ci fait alors disparaître un bloc dur et ouvre un nouvel accès à une partie de la carte, jusqu'ici inaccessible. Si le joueur récupère un fruit, on l'affiche dans l'interface et vérifie s'il a gagné le niveau. Toutefois, aucune de ces cases spéciales ne doit être détruite par une bombe, auquel cas le niveau est perdu et le joueur doit le recommencer.

### Menus

Nous avons séparé le menu du jeu en plusieurs fonctions de menus associés au cas sélectionné. Cela nous permet d'alléger le code du menu et par la même occasion d'éviter de charger toutes les images. En effet, on ne charge que celles dont on a besoin dans le menu courant, que l'on libère avant de passer à une autre fonction. Voici comment est organisé le menu:



## D. Un programme modulaire

Pour le programmeur, il est possible de modifier un large panel de constantes globales afin d'adapter le jeu à sa guise. Beaucoup de paramètres dépendent de ces constantes, qui, une fois modifiées impacteront l'ensemble du programme. Par exemple, il est possible de modifier les fréquences d'apparition des blocs cassables, comme des bonus.

L'utilisateur a, quant à lui, la possibilité de modifier un certain nombre de réglages. Il peut d'une part ajuster le volume des effets sonores de l'intégralité du jeu. Nous avons eu le temps de lui offrir jusque là l'opportunité de changer le nombre de manches d'une partie ainsi que les noms des deux joueurs contrôlables au clavier. Toutefois, nous avons installé des contrôles bornant le nombre de manches possibles ainsi que le nombre de caractères maximum et minimum qu'un nom peut avoir.

## IV. Les difficultés rencontrées

### A. Les changements globaux

Lorsque l'on modifiait l'une des fonctions utilisées à de nombreuses reprises dans d'autres endroits du programme, il fallait adapter l'ensemble du code pour que celui-ci fonctionne correctement. Par exemple, si on inversait les valeurs de retour d'une fonction, il fallait trouver et réajuster la façon dont on se servait de cette valeur à d'autres endroits du programme. Aussi, lorsque l'on se rendait compte qu'il manquait un paramètre pour ajouter de nouvelles fonctionnalités, il fallait s'assurer qu'à chaque appel de cette fonction-ci, nous disposions bien de l'argument nécessaire.

De plus, séparer les tâches en deux possède des avantages mais aussi des inconvénients. D'une part, cela nous a permis de développer rapidement à la fois le multijoueurs et la campagne. D'autre part, il nous a fallu prendre du temps pour comprendre le travail de l'autre, mettre en commun les deux parties du code et l'ajuster.

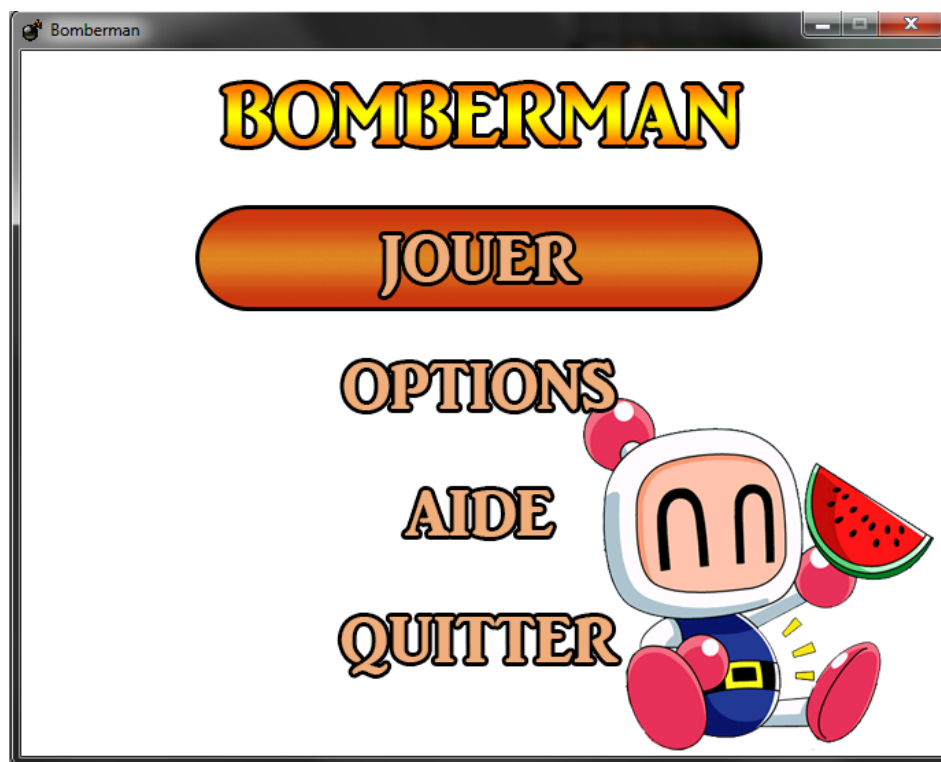
### B. Les allocations de la mémoire

Nous avons eu plusieurs difficultés quant à l'utilisation de la file de bombes. En effet, le programme cessait de fonctionner par moment lorsque l'on manipulait des bombes pendant la partie. Après un temps de recherches assez conséquent, nous avons compris que cette erreur venait de l'explosion suivie de la suppression d'une bombe. Nous libérions la variable temp lors du parcours de la file de bombe alors qu'elle n'existait plus. Il nous a suffi de mettre une valeur de retour à la fonction explosion, pour indiquer si elle avait déjà libérée la bombe considérée.

Aussi, nous avons pris beaucoup de temps pour vérifier les moments et les endroits où libérer ou non les variables préalablement allouées. Il en est de même pour toutes les libérations des images, surfaces, sons, ainsi que les fermetures des librairies et fichiers ouverts. Il s'agissait de concevoir un programme propre et bien pensé, sans fuite de mémoire.

## V. Les annexes

### A. Le menu



## B. Le mode histoire



## C. Le mode multijoueurs

