Programmation impérative

Projet 2015 – Mini jeu 2040boum

Mehdi Khadir
04/01/2016

Introduction

Le 2040boum est un mini jeu implémenté dans ce projet en C en mode texte. L'interface est un tableau du nombre de colonnes et de lignes défini par l'utilisateur en début de partie. Il y a deux familles d'éléments dans ce tableau : des nombres et des bombes. L'utilisateur peut déplacer l'ensemble des éléments à chaque tour, qui s'additionnent s'ils sont de même valeur, ou faire exploser des bombes. Apparaît ensuite aléatoirement un nouvel élément avant le tour suivant.

J'expose dans ce document les choix que j'ai pris, les problèmes techniques que j'ai rencontrés et les solutions que j'ai trouvées afin de mener à bien ce projet. Je présente enfin les limites du jeu tel que je l'ai implémenté.

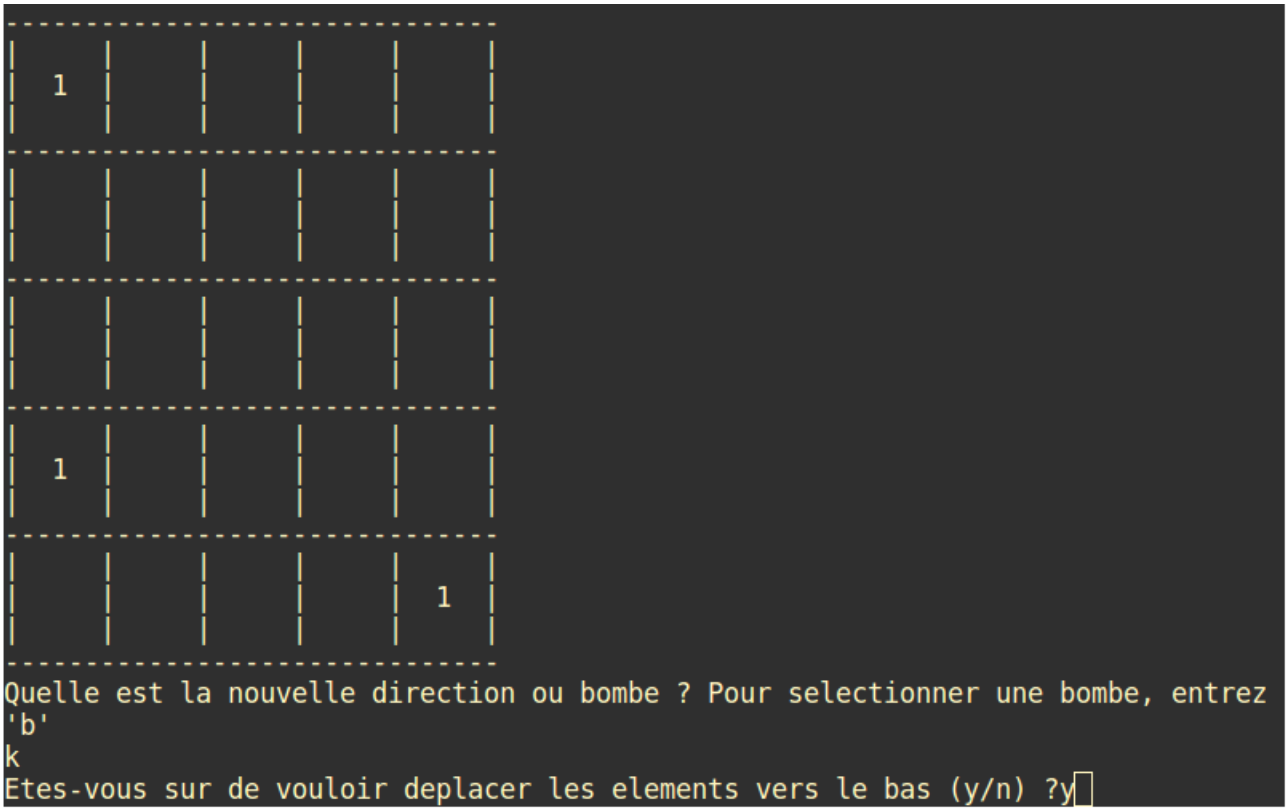
I – Fonctionnement global

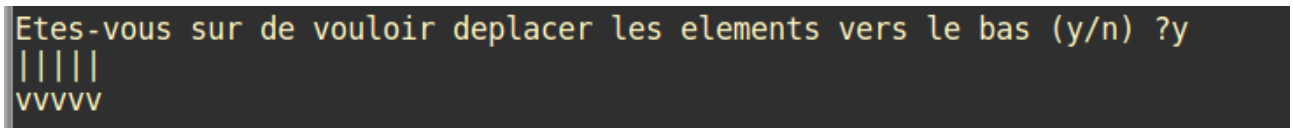
J'ai tout d'abord décidé d'adopter une structure de tableau à deux dimensions pour enregistrer les éléments. C'est un tableau d'entiers positifs, ou négatifs s'il s'agit de bombes, 0 étant réservé à la case vide. Une fonction d'affichage « correspondance » permet de traduire par un char les bombes de type int. J'ai adopté cette convention pour que les additions soient plus simples, sachant que deux bombes de même valeur qui se choquent forment une bombe deux fois plus puissante.

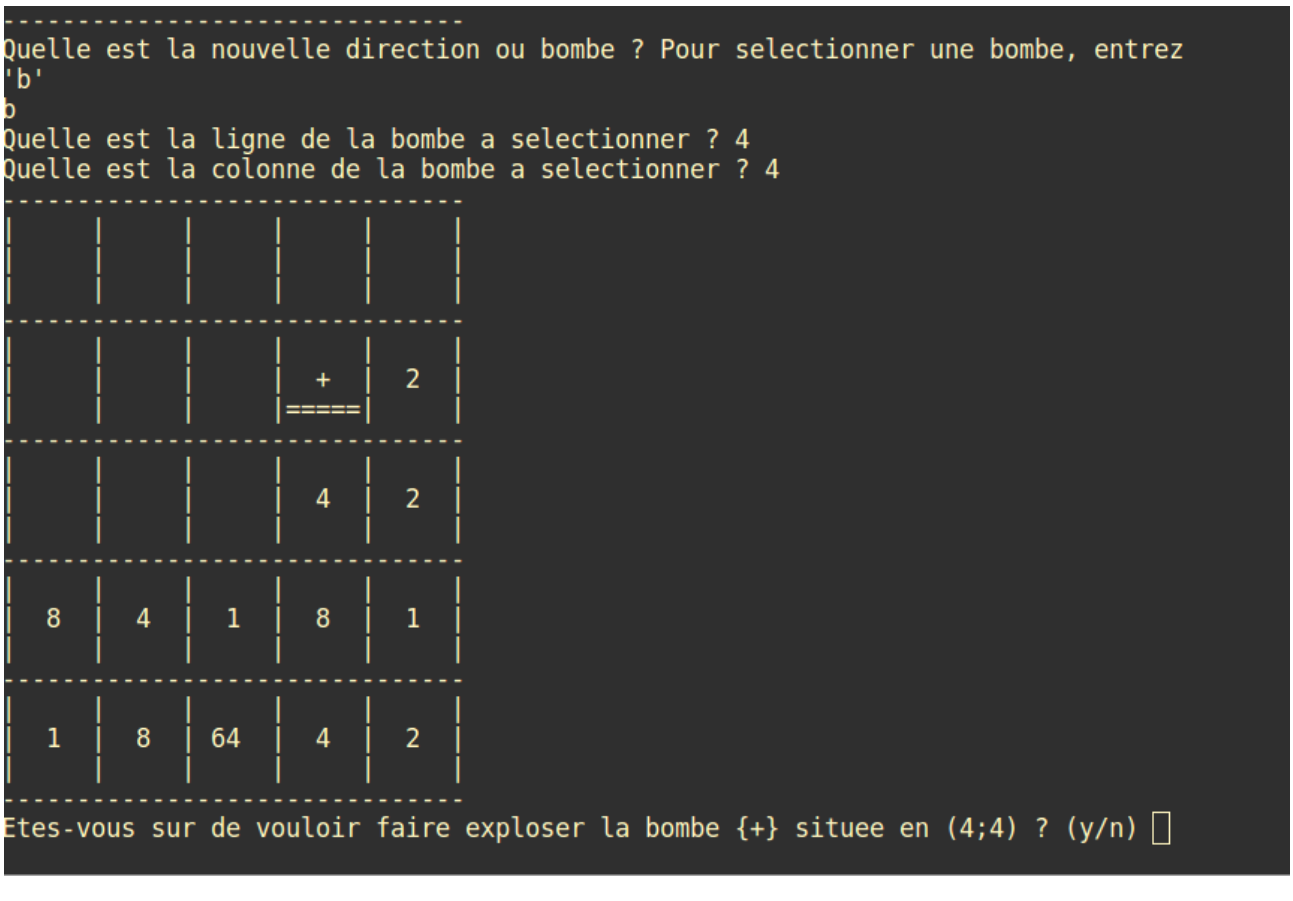
Pour gérer les bombes et leurs compteurs qui s'incrémentent à chaque tour ou sont initialisés à 1 pour une nouvelle bombe, j'ai utilisé un second tableau du même format que le tableau de jeu « tjeu ». Ce second tableau « tbombe » réagit en même temps que tjeu lorsqu'il y a des déplacements ou explosions. « tbombe » et « tjeu » sont initialisés dès le début de la partie : ils se remplissent de 0 et tjeu comporte un 1 placé aléatoirement.

Le jeu se termine lorsque l'objectif que s'est fixé au début l'utilisateur est atteint (un élément du tableau a une valeur supérieure ou égale à celle de l'objectif), ou lorsqu'il n'est plus possible d'insérer des éléments. J'ai traduit cette condition par une boucle while, avec un pointeur sur max, qui vaut toujours l'élément de plus haute valeur dans le tableau, pour que le maximum puisse facilement être modifié par les fonctions de déplacement. A chaque nouveau tour, il est demandé à l'utilisateur s'il

souhaite effectuer un déplacement ou faire exploser une bombe. Le choix de l'utilisateur est mis en évidence par le soulignage de la bombe sélectionnée ou par l'affichage de flèches directionnelles dans le cas d'un déplacement. Une confirmation est demandée.







II – L'affichage

A chaque tour, on affiche le tableau avec la fonction « affichetab ». Pour simplifier cette fonction, j'ai décidé de la découper en deux fonctions : la fonction « afficheligne » ne fait qu'afficher une ligne du tableau et « affichetab » fait appel à « afficheligne » i fois où i est le nombre de lignes du tableau.

Le problème principal était de trouver une solution pour souligner la bombe sélectionnée de manière claire. J'ai choisi d'inclure cette possibilité directement dans les fonctions « afficheligne » et « affichetab » en entrant en paramètres deux entiers supplémentaires « ligne_b » et « colonne_b » qui valent les indices de la bombe à souligner. Ces paramètres ne sont alors pris en compte par les fonctions d'affichage que s'ils sont positifs ou nuls. Pour chaque case à afficher, la fonction soulignera alors de « ==== » si les indices traités correspondent à « ligne_b » et « colonne_b ».

Pour mettre en évidence les directions choisies, j'affiche simplement des flèches indiquant la direction d'attraction.

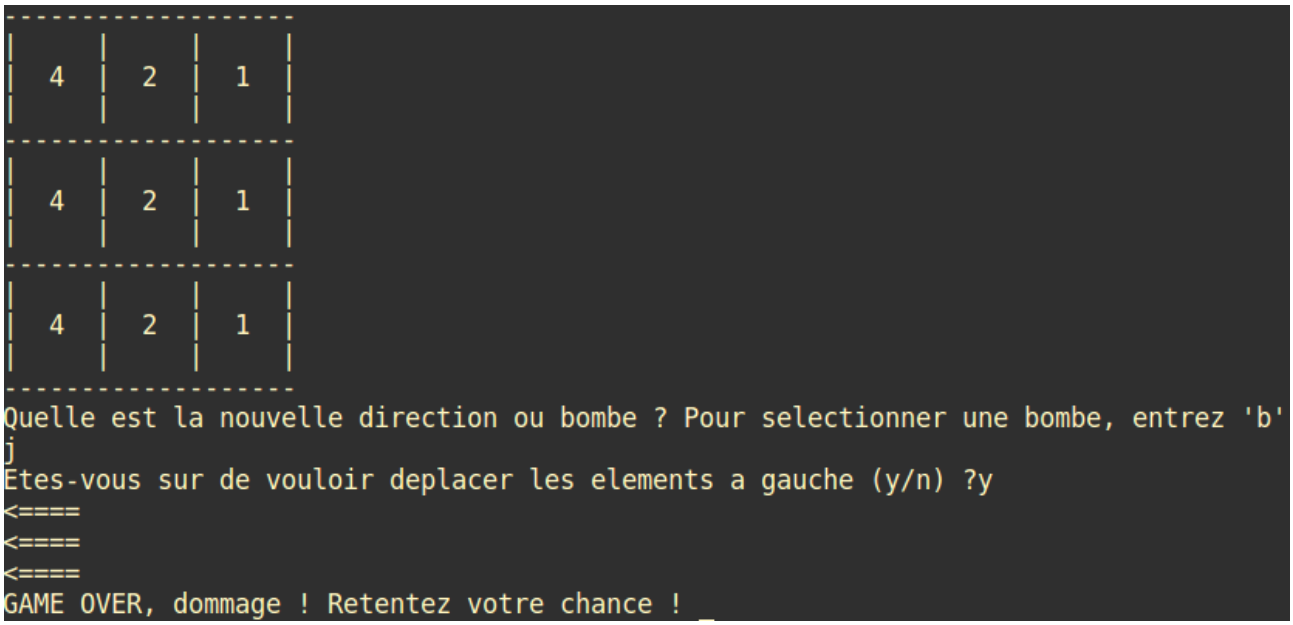
Pour un affichage clair, chaque ligne est composée de 3 lignes de texte et chaque colonne de 3 colonnes de texte. Les séparateurs sont « - » et « | ».

III– Les éléments placés aléatoirement

Plusieurs fonctions permettent de placer aléatoirement un nombre aléatoire à chaque fin de tour. Il faut d'abord choisir aléatoirement l'élément à placer, dont la valeur dépend du maximum. Ceci se fait avec la fonction « aleaobj » qui, en faisant intervenir la fonction rand(), renvoie des entiers positifs, ou négatifs si des bombes sont placées. Les pourcentages de chance d'apparition d'un élément dépendent uniquement de la valeur du maximum.

Puis il faut déterminer l'emplacement où l'élément sera positionné aléatoirement. La difficulté était de trouver aléatoirement un emplacement vide. J'ai donc décidé de créer une fonction « vacuite » qui vérifie que le tableau n'est pas complet. La fonction « newelement », qui place le nouvel objet, fait tout d'abord appel à « vacuite » pour vérifier qu'il y a une place vide. Puis tant que le nouvel élément n'a pas été placé, on choisi aléatoirement une ligne et une colonne dans le tableau. Cette boucle est finie, puisqu'on a préalablement vérifié qu'il y a bien une case vide.

Si la fonction vacuite renvoie 0, cela signifie que le tableau est complet. Un message indiquant la fin du jeu apparaît alors. Ceci explique la terminaison de la boucle while du main : en effet « newelement », et donc « vacuite », est appelé à chaque tour de boucle.



IV – Déplacements

Les fonctions de déplacement étaient les plus compliquées à implémenter, puisqu'il fallait prendre en compte les possibles fusions et multiplication par 2 lorsqu'il s'agit de deux éléments de même valeur. Il n'y a que 4 déplacements possibles, que j'ai implémentés en 8 fonctions : pour chaque déplacement, il y a une fonction qui déplace tous les éléments d'une ligne ou d'une colonne, et une autre qui appelle plusieurs fois cette première fonction pour l'appliquer à toutes les lignes ou colonnes. On effectue les mêmes déplacement sur la tableau de compteurs de bombes pour le tenir à jour.

Prenons l'exemple du déplacement vers la gauche sur une ligne, implémenté par la fonction « depgauche1 », tous les autres déplacements fonctionnant de la même manière. On part de la gauche pour aller vers la droite, en traitant chaque élément. Tant qu'il n'y a rien à gauche, on déplace l'élément traité vers la gauche, en gardant en mémoire la nouvelle colonne de cet élément avec l'entier « k » que l'on décrémente à chaque tour de boucle. Si l'élément de gauche est de même valeur, celui-ci est multiplié par deux et l'élément traité est remplacé par une case vide. Je traite aussi le cas particulier de deux bombes * qui se rencontrent avec la condition

`if(tab.t[i][k-1]==-8)` qui est testée après la multiplication par 2 et qui explose les éléments aux alentours et met à jour le tableau de compteurs de bombes si elle est vérifiée. Si l'élément de gauche n'est pas de la même valeur, on s'arrête et traite l'élément suivant.

Le problème avec cette méthode était que dans le même déplacement, un élément multiplié par 2 pouvait se voir à nouveau multiplié par 2 si sa nouvelle valeur était celle de l'élément de droite. J'ai réglé ce problème à l'aide d'un entier « bloque » qui vaut -1 dès le départ, et vaut ensuite l'indice du dernier élément multiplié par 2. Si lors du déplacement à gauche l'élément sur la gauche vaut la valeur de l'élément traité mais que « bloque » a pour indice celui de l'élément de gauche, la multiplication par 2 ne s'effectue pas.

La fonction « depgauche » effectue le déplacement à gauche pour chaque ligne et après chaque déplacement parcourt à nouveau les éléments pour mettre à jour la valeur du maximum.

V – Explosion d'une bombe

L'explosion d'une bombe est implémenté par la fonction « explose » qui prend pour argument l'entier correspondant à la bombe, les indices de position de cette bombe et le tableau de jeu. Elle agit directement sur ce tableau en mettant des cases vides là où il faut après l'explosion de la bombe.

Enfin la fonction « check_bombe », appelée à chaque tour de boucle, agit sur le tableau comportant les compteurs de bombes. Elle incrémente tous les éléments lorsqu'un tour se termine, ou explose une bombe une fin de vie, c'est-à-dire une bombe dont le compteur vaut 8.

Les limites du programme

La valeur maximum est mise à jour durant les déplacements, mais cela se fait en repassant deux fois par les mêmes cases, après que les déplacements aient eu lieu. Ce choix n'est pas le plus optimal.

De même, l'algorithme recherche à chaque tour s'il y a une case vide afin de placer le nouvel élément en parcourant l'ensemble des cases. Et sachant qu'il y a une case vide, on la cherche en boucle de manière aléatoire, ce qui peut prendre beaucoup de temps si les indices choisis au hasard ne correspondent pas aux cases vides.