Manea Palluat de Besset - Groupe 3 : manea.palluat@gmail.com
15/12/2023

Développement et Implémentation de XMAS TREATS : Un jeu de type puzzle en console

Manea Palluat de Besset - Groupe 3 : manea.palluat@gmail.com
15/12/2023

Table des Matières

I) Introduction..... 3
1) Brève présentation du jeu..... 3
2) Objectifs du rapport..... 3
II) Conceptualisation du Jeu..... 3
1) Étapes de création :..... 3
2) Explication technique du programme..... 5
A) GRILLE, MOUVEMENTS ET APPARITION DES BONBONS..... 5
B) GESTION DE LA PARTIE ET EXPÉRIENCE DU JOUEUR..... 14
III) MAIN..... 18
III) Tableau d'organisation..... 19
IV) Conclusion..... 20

I) Introduction

1) Brève présentation du jeu

Dans "XMAS TREATS", le joueur combine des bonbons sur un plateau 4x4 pour marquer des points. Les bonbons, réglisses, cookies et sucres d'orge valent respectivement 1, 3, 7 et 15 points. Les bonbons se combinent en déplaçant vers les directions cardinales avec le pavé numérique, formant des bonbons de valeur supérieure lorsqu'ils se rencontrent. Un nouveau bonbon apparaît après chaque mouvement. Le jeu se termine quand le nombre maximal de coups est atteint ou lorsqu'aucun mouvement n'est possible, avec un score basé sur les bonbons restants.

2) Objectifs du rapport

Ce rapport explore le processus de développement de "XMAS TREATS", un jeu de puzzle interactif basé sur la thématique de Noël. Le projet a été conçu pour illustrer la fusion entre une jouabilité intuitive et une expérience utilisateur engageante, en utilisant les principes de base de la programmation en C#.

Le but principal de ce projet était de développer un jeu simple mais captivant, qui mettrait en œuvre des compétences fondamentales en matière de programmation, telles que la manipulation de tableaux, la gestion des entrées utilisateur, et la logique de jeu. Le thème festif a été choisi pour rendre le jeu attractif et accessible à un large public.

II) Conceptualisation du Jeu

1) Étapes de création :

1) Définition de la grille et des règles de base : J'ai commencé par établir une grille de jeu 4x4 et définir les règles de base, y compris les différents types de bonbons et leurs points.

2) Développement des fonctionnalités clés : J'ai programmé les fonctionnalités essentielles telles que la création et l'initialisation de la grille, la gestion des mouvements des bonbons, et les règles pour fusionner les bonbons.

3) Gestion des entrées utilisateur : J'ai implémenté la gestion des entrées utilisateur pour les mouvements et le nombre de coups.

4) Ajout de conditions de fin de jeu : J'ai ajouté des conditions de fin de jeu, incluant l'atteinte du nombre maximal de coups et le blocage lorsque plus aucun mouvement n'est possible.

5) Calcul du score : J'ai créé une fonction pour calculer le score basé sur les bonbons présents sur le plateau à la fin du jeu.

6) Améliorations et débogage : En réponse à des erreurs et des besoins d'amélioration, j'ai modifié et débogué le code, notamment en ajoutant des vérifications pour les entrées invalides et en ajustant le comportement des bonbons lors de la fusion.

7) Ajout d'Instructions et d'aide : Pour améliorer l'expérience utilisateur, j'ai inclus des instructions et une option d'aide au début du jeu.

8) Optimisation et clarification du code : Enfin, j'ai affiné le code, j’ai “factorisé” et rassemblé certaines fonctions pour le rendre plus compact, et j’ai ajouté des sauts de lignes et des indentations pour une meilleure clarté.

2) Explication technique du programme

Au début de la création de "XMAS TREATS", j'ai rapidement compris que le cœur du jeu résiderait dans son plateau, la grille de jeu. C'est ici que toutes les actions se déroulent, où les bonbons sont déplacés et fusionnés. Ma première étape a donc été de créer et d'initialiser cette grille. J'ai pris soin de la concevoir de manière à ce qu'elle soit à la fois flexible et intuitive pour les mécaniques du jeu. Voici un aperçu plus détaillé de la manière dont j'ai abordé cette partie du développement.

A) GRILLE, MOUVEMENTS ET APPARITION DES BONBONS

Dans ce code, j'ai commencé par définir la taille de la grille, qui est un choix déterminant pour la dynamique du jeu. Une grille 4x4 offre un bon équilibre entre complexité et jouabilité.

Ensuite, j'ai créé la grille sous forme d'un tableau bidimensionnel char[,] grille. Chaque cellule

```
126  // Crée et initialise une grille de jeu 4x4
127  // Cette méthode crée et initialise une grille de jeu 4x4 pour le jeu.
128  char[,] CreerEtInitialiserGrille()
129  {
130      int taille = 4; // Définit la taille de la grille, ici 4x4.
131
132      // Création d'un tableau bidimensionnel de caractères de taille 4x4.
133      // Ce tableau représente la grille de jeu.
134      char[,] grille = new char[taille, taille];
135
136      // Double boucle for pour parcourir chaque cellule de la grille.
137      for (int i = 0; i < taille; i++) // Boucle sur les lignes de la grille.
138      {
139          for (int j = 0; j < taille; j++) // Boucle sur les colonnes de la grille.
140          {
141              // Initialisation de chaque cellule de la grille avec un espace ' '.
142              // Cela indique que la cellule est vide au début du jeu.
143              grille[i, j] = ' ';
144          }
145      }
146
147      // Ajoute deux bonbons aléatoires sur la grille.
148      // La fonction 'AjouterBonbonAleatoire' est appelée deux fois pour placer deux bonbons.
149      AjouterBonbonAleatoire(grille);
150      AjouterBonbonAleatoire(grille);
151
152      // Retourne la grille initialisée prête pour le jeu.
153      return grille;
154  }
```

de ce tableau peut contenir un caractère. Pour l'initialisation, j'ai opté pour une double boucle for, qui assigne un espace ' ' à chaque cellule. Cela signifie que toutes les cellules sont vides, prêtes à accueillir les bonbons.

La partie cruciale de cette méthode est l'ajout des premiers bonbons. J'utilise la méthode AjouterBonbonAleatoire() pour placer deux bonbons de manière aléatoire sur la grille. Cette étape est essentielle pour démarrer le jeu avec une configuration initiale intéressante.

En développant cette méthode, j'ai posé les bases du jeu, garantissant une expérience de départ cohérente à chaque nouvelle partie. Cette approche m'a permis de construire les autres composants du jeu sur une fondation solide.

Après avoir mis en place la grille initiale, mon prochain objectif était de la dynamiser avec des éléments interactifs, c'est-à-dire les bonbons. Pour cela, j'ai écrit la fonction AjouterBonbonAleatoire. Cette fonction ajoute un bonbon dans une case aléatoire de la grille. Voici comment j'ai procédé :

```
330  // Ajoute un bonbon aléatoire à la grille
331  void AjouterBonbonAleatoire(char[,] grille)
332  {
333      Random rnd = new Random();
334      int positionL, positionC;
335
336      do
337      {
338          positionL = rnd.Next(grille.GetLength(0));
339          positionC = rnd.Next(grille.GetLength(1));
340      }
341      while (grille[positionL, positionC] != ' ');
342
343      // On utilise rnd.Next(8) pour générer un nombre entre 0 et 7
344      // Si ce nombre est 0 (une chance sur 8), le bonbon sera '@'
345      // Sinon (les 7 autres cas sur 8), le bonbon sera '*'
346      char bonbon = rnd.Next(8) == 0 ? '@' : '*';
347      grille[positionL, positionC] = bonbon;
348  }
```

Dans cette fonction, j'ai d'abord instancié un objet Random pour obtenir des emplacements aléatoires sur la grille. Ensuite, j'ai utilisé une boucle do-while pour continuer à générer des positions aléatoires jusqu'à trouver une case vide (indiquée par ' '). Cela garantit que les bonbons ne remplacent pas ceux déjà présents sur la grille.

La partie la plus intéressante est la décision du type de bonbon à placer. J'ai utilisé rnd.Next(8) pour obtenir un nombre entre 0 et 7. Si le nombre généré est 0, ce qui a 1 chance sur 8 de se produire, un bonbon spécial '@' est placé. Sinon, un bonbon standard '*' est ajouté.

Ce n'était pas demandé dans la consigne mais je trouve que cela rajoute un plus au jeu. Cette méthode apporte un élément de surprise et de stratégie au jeu. Le joueur doit anticiper ces ajouts aléatoires pour planifier ses mouvements. Parfois, cela peut l'obliger à effectuer un mouvement qu'il n'avait pas anticipé en déjouant sa stratégie, s'il prévoit ce qu'il va effectuer deux mouvements à l'avance par exemple. Cela enrichit l'expérience de jeu et ajoute de la variété à chaque partie.

Après avoir géré l'initialisation de la grille et l'ajout aléatoire des bonbons, j'ai ensuite concentré mes efforts sur la gestion des mouvements des joueurs. J'ai créé une méthode GererMouvements qui permet aux joueurs d'interagir avec le jeu en déplaçant les bonbons. Voici comment elle fonctionne :

```
185   // Gère les entrées de l'utilisateur pour les mouvements
186   bool GererMouvements(char[,] grille)
187   {
188       Console.WriteLine("Entrez un mouvement (8: haut, 6: droite, 4: gauche, 2: bas):");
189       char mouvement = Console.ReadKey().KeyChar;
190       Console.ReadLine(); // Consomme le reste de la ligne
191
192       char[,] grilleAvantMouvement = (char[,])grille.Clone(); // Copie de la grille avant le mouvement
193
194       switch (mouvement)
195       {
196           case '8':
197               DeplacerVersHaut(grille);
198               FusionnerBonbonsHaut(grille);
199               DeplacerVersHaut(grille);
200               break;
201           // Autres cas
202           default:
203               Console.WriteLine("\nEntrée non valide. Réessayez.");
204               return false;
205       }
206
207       // Vérifie si un changement a été fait sur la grille
208       if (EstIdentique(grille, grilleAvantMouvement))
209       {
210           Console.WriteLine("\n ⚠  Mouvement sans effet. Réessayez.");
211           return false;
212       }
213
214       AjouterBonbonAleatoire(grille); // Ajoute un bonbon après un mouvement avec effet
215       return true;
216   }
```

Cette méthode est cruciale pour l'interaction avec le jeu. Elle commence par demander au joueur d'entrer un mouvement, puis utilise une structure switch pour traiter les différentes directions de mouvement. Chaque cas dans le switch appelle d'abord la méthode pour déplacer les bonbons, puis celle pour fusionner les bonbons si deux identiques se rencontrent, et enfin redéplace les bonbons pour combler les espaces vides laissés par la fusion. Ce bloc switch est fondamental pour la logique de jeu. Il permet de traiter les différentes entrées de l'utilisateur (8 pour haut, 6 pour droite, 4 pour gauche, 2 pour bas) et de déclencher les mouvements correspondants. Chaque case dans le switch correspond à une direction spécifique et exécute une série d'actions: déplacement des bonbons, fusionnement des bonbons identiques, et un second déplacement pour ajuster la grille après les fusions. Cette approche assure que les bonbons sont déplacés et fusionnés correctement selon les règles du jeu.

En cas d'entrée invalide (aucun des cas 8, 6, 4, ou 2), le jeu informe l'utilisateur et lui demande de réessayer, garantissant ainsi que seules les entrées valides sont traitées.

Si aucun changement n'est détecté dans la grille après le mouvement (vérifié par EstIdentique), le joueur est informé que son mouvement n'a pas eu d'effet et on lui demande de réessayer. Cela empêche les mouvements inutiles et incite le joueur à réfléchir stratégiquement.

Enfin, si un mouvement valide est effectué, un nouveau bonbon est ajouté aléatoirement sur la grille.

La méthode EstIdentique dont je viens de parler est expliquée ici. Son rôle est de comparer deux grilles de jeu pour vérifier si elles sont identiques.

```
233  bool EstIdentique(char[,] grille1, char[,] grille2)
234  {
235      // La première étape consiste à parcourir chaque ligne de la première grille.
236      // J'utilise 'grille1.GetLength(0)' pour obtenir le nombre de lignes de la grille.
237      for (int i = 0; i < grille1.GetLength(0); i++)
238      {
239          // Ensuite, pour chaque ligne, je parcours toutes les colonnes de cette ligne.
240          // 'grille1.GetLength(1)' donne le nombre de colonnes de la grille.
241          for (int j = 0; j < grille1.GetLength(1); j++)
242          {
243              // Ici, je compare les éléments situés à la position [i, j] dans les deux grilles.
244              // Si un élément à une position donnée est différent entre grille1 et grille2,
245              // cela signifie qu'un changement a été effectué à cette position.
246              if (grille1[i, j] != grille2[i, j])
247                  return false; // Si je trouve une différence, je retourne immédiatement false.
248          }
249      }
250
251      // Si je termine les boucles for sans trouver aucune différence,
252      // cela signifie que les deux grilles sont exactement les mêmes.
253      // Dans ce cas, je retourne true.
254      return true;
255  }
```

Cette fonction est fondamentale pour déterminer si un mouvement effectué par le joueur a entraîné des modifications sur le plateau. La logique commence par parcourir chaque ligne des deux grilles, grille1 et grille2, en utilisant une boucle for. Pour chaque ligne, une boucle for imbriquée parcourt ensuite chaque colonne.

À chaque itération, l'élément de la grille1 à la position actuelle (ligne i, colonne j) est comparé à l'élément correspondant dans la grille2. Si une différence est trouvée entre les deux éléments (signifiant qu'un élément à un emplacement donné dans grille1 est différent de l'élément au même emplacement dans grille2), la fonction renvoie immédiatement false, indiquant que les grilles ne sont pas identiques.

Si aucune différence n'est trouvée après avoir examiné toutes les lignes et colonnes, la fonction conclut que les grilles sont identiques et renvoie true. Cette vérification est essentielle pour s'assurer que le jeu réagit correctement aux mouvements du joueur, ne permettant pas des mouvements inutiles et mettant à jour la grille seulement quand un mouvement valide est effectué.

La fonction AfficherGrilleEtInfos est un élément central de l'interface utilisateur. Elle affiche non seulement l'état actuel de la grille de jeu, mais fournit également des informations clés au joueur. Voici une explication plus détaillée des aspects techniques de cette fonction :

```
299  // Affiche la grille et les informations du jeu
300  void AfficherGrilleEtInfos(char[,] grille, int coupsRestants, int coupsJoues)
301  {
302      int taille = grille.GetLength(1);
303      string ligneHorizontale = "+";
304
305      for (int i = 0; i < taille; i++)
306      {
307          ligneHorizontale += "---+";
308      }
309
310      // Afficher la ligne du haut de la grille
311      Console.WriteLine(ligneHorizontale);
312
313      for (int i = 0; i < grille.GetLength(0); i++)
314      {
315          for (int j = 0; j < grille.GetLength(1); j++)
316          {
317              Console.Write($"| {grille[i, j]} ");
318          }
319          Console.WriteLine("|");
320
321          // Afficher la ligne horizontale après chaque ligne de la grille
322          Console.WriteLine(ligneHorizontale);
323      }
324
325      Console.WriteLine($"Nombre de coups joués : {coupsJoues}");
326      Console.WriteLine($"Nombre de coups restants : {coupsRestants}");
327      Console.WriteLine("Rappel des commandes : 8 pour haut ⬆, 6 pour droite➡, 4 pour gauche⬅, 2 pour bas ⬇.");
328  }
```

Détermination de la Taille de la Grille et Création de la Ligne Horizontale:

int taille = grille.GetLength(1); : Je récupère la dimension de la grille, qui est carrée, pour déterminer combien de fois je dois répéter le motif de la ligne horizontale.
string ligneHorizontale = "+"; : J'initialise une chaîne pour représenter les bordures horizontales de la grille. Le signe "+" sert de coin pour chaque cellule.
La boucle for qui suit construit la ligneHorizontale en ajoutant "---+" pour chaque colonne de la grille. Cela crée un aspect visuel de cases séparées.
Affichage de la Grille:

La première utilisation de Console.WriteLine(ligneHorizontale); imprime la bordure supérieure de la grille.
Les boucles for imbriquées (for (int i = 0; i < grille.GetLength(0); i++) et son enfant) parcourent chaque cellule de la grille.
Console.Write($"| {grille[i, j]} "); : Pour chaque cellule, je récupère et affiche son contenu. L'opérateur $ permet d'intégrer la valeur de la cellule directement dans la chaîne.

Après avoir affiché toutes les cellules d'une ligne, j'utilise Console.WriteLine("|"); pour fermer la ligne de la grille.

Affichage des Informations de Jeu:

Console.WriteLine($"Nombre de coups joués : {coupsJoues}"); et Console.WriteLine($"Nombre de coups restants : {coupsRestants}"); fournissent des mises à jour en temps réel sur l'état du jeu.
Enfin, je rappelle les commandes disponibles pour les mouvements, ce qui est important pour l'accessibilité et l'expérience utilisateur.

La fonction DeplacerVersBas gère le déplacement des bonbons vers le bas de la grille lorsque le joueur choisit cette direction. Voici une explication technique de cette fonction :

```
353  // Déplace les bonbons vers le bas
354  void DeplacerVersBas(char[,] grille)
355  {
356      // On parcourt chaque colonne de la grille
357      for (int col = 0; col < grille.GetLength(1); col++)
358      {
359          // On initialise la position d'insertion au bas de la colonne
360          int positionInsertion = grille.GetLength(0) - 1;
361
362          // On parcourt la colonne de bas en haut
363          for (int row = grille.GetLength(0) - 1; row >= 0; row--)
364          {
365              // On vérifie si la case courante contient un bonbon (n'est pas vide)
366              if (grille[row, col] != ' ')
367              {
368                  // On déplace le bonbon à la position d'insertion actuelle
369                  grille[positionInsertion, col] = grille[row, col];
370
371                  // Si la position d'origine est différente de la position d'insertion, on vide la position d'origine
372                  if (row != positionInsertion)
373                  {
374                      grille[row, col] = ' ';
375                  }
376
377                  // On décrémente la position d'insertion pour le prochain bonbon à déplacer
378                  positionInsertion--;
379              }
380          }
381      }
382  }
```

La fonction commence par parcourir chaque colonne de la grille, de gauche à droite. Pour chaque colonne, j'initialise une variable positionInsertion à la ligne la plus basse de la colonne. Cette position indique où le bonbon le plus proche du bas de la grille doit être déplacé. Ensuite, je parcours la colonne de bas en haut pour examiner chaque cellule. Si une cellule contient un bonbon (c'est-à-dire, n'est pas vide), je déplace ce bonbon à la position d'insertion. Il est crucial de noter que si la position d'origine du bonbon est différente de la position d'insertion, je vide la position d'origine pour simuler le mouvement du bonbon vers le bas.

Un aspect technique important ici est la gestion des positions dans un tableau bidimensionnel. Chaque fois qu'un bonbon est déplacé vers le bas, la position d'insertion est mise à jour en se déplaçant d'une case vers le haut. Cela garantit que les bonbons suivants

seront placés correctement dans les espaces vides. Cette méthode illustre la manière dont la logique de déplacement et de gestion de l'espace est implémentée dans un jeu de grille, rendant l'interaction du joueur avec le jeu à la fois

Il en va du même principe pour les autres directions.

Je vais maintenant vous présenter la logique derrière la fusion des bonbons lorsqu'ils se déplacent vers le haut.

```
452  // Fusionne les bonbons lors du déplacement vers le haut
453  void FusionnerBonbonsHaut(char[,] grille)
454  {
455      for (int col = 0; col < grille.GetLength(1); col++)
456      {
457          for (int row = 0; row < grille.GetLength(0) - 1; row++)
458          {
459              // Ajouter une condition pour éviter de fusionner deux 'J'
460              if (grille[row, col] == 'J' && grille[row + 1, col] == 'J')
461                  continue;
462
463              if (grille[row, col] == grille[row + 1, col] && grille[row, col] != ' ')
464              {
465                  grille[row, col] = ProchainBonbon(grille[row, col]);
466                  grille[row + 1, col] = ' ';
467              }
468          }
469      }
470  }
```

La fonction FusionnerBonbonsHaut est conçue pour fusionner deux bonbons identiques lorsqu'ils se rencontrent dans une colonne pendant un mouvement vers le haut. La fusion des bonbons est une étape clé du jeu, car elle permet de créer des bonbons de valeur supérieure et d'accumuler plus de points.

Pour chaque colonne de la grille, j'examine les bonbons de bas en haut, à l'exception du dernier élément de la colonne (car il n'y a rien au-dessus avec lequel fusionner). Une nouveauté importante que j'ai intégrée est la vérification spécifique pour les bonbons de type 'J'. Si deux bonbons 'J' se rencontrent, ils ne doivent pas fusionner. Cette condition est cruciale pour préserver l'intégrité du gameplay, car le bonbon 'J' est le plus élevé dans la hiérarchie des bonbons et ne doit pas se transformer en un autre type.

Lorsque deux bonbons identiques (autres que 'J') se rencontrent, ils fusionnent en un bonbon de type supérieur, géré par la fonction ProchainBonbon. Le bonbon inférieur (celui situé juste au-dessus dans la colonne) est alors vidé (' '), simulant la fusion. Cette approche garantit une expérience de jeu dynamique où le joueur doit stratégiquement manœuvrer les bonbons pour atteindre des scores élevés.

Cette partie du code illustre comment les règles du jeu sont programmées pour offrir une expérience ludique tout en respectant les contraintes du gameplay, un aspect crucial dans la conception de jeux.

Les autres directions utilisent des fonctions similaires.

Un aspect crucial est la progression des bonbons à travers différentes étapes de fusion. Pour gérer cela, j'ai conçu la méthode ProchainBonbon, qui détermine le type de bonbon résultant après une fusion.

```
535  // Cette méthode détermine le prochain type de bonbon après une fusion
536  char ProchainBonbon(char bonbon)
537  {
538      // Utilisation d'une structure switch pour traiter les différents types de bonbons
539      switch (bonbon)
540      {
541          case '*':
542              // Si le bonbon actuel est '*', il fusionne en '@'
543              return '@';
544
545          case '@':
546              // Si le bonbon actuel est '@', il fusionne en 'o'
547              return 'o';
548
549          case 'o':
550              // Si le bonbon actuel est 'o', il fusionne en 'J'
551              return 'J';
552
553          case 'J':
554              return 'J'; // Garder le même type pour 'J'
555
556          default:
557              // Pour 'J' ou tout autre cas inattendu, la fusion ne produit aucun nouveau bonbon
558              // (Peut servir pour gérer une erreur ou une condition inattendue)
559              return ' ';
560      }
561  }
```

Cette méthode utilise une structure switch pour traiter les différents types de bonbons. Chaque case correspond à un type de bonbon spécifique. Voici comment elle fonctionne :

Bonbon '*' : Si le bonbon actuel est ", représentant le niveau le plus bas, il fusionne en '@' lorsqu'il rencontre un autre bonbon ". Cela augmente la valeur du bonbon et représente la première étape de progression dans le jeu.

Bonbon '@' : De manière similaire, si le bonbon actuel est '@', il fusionne en 'o' lorsqu'il rencontre un autre '@'. Cela représente une étape supplémentaire dans l'escalade de la valeur des bonbons.

Bonbon 'o' : Lorsque deux bonbons 'o' se rencontrent, ils fusionnent en 'J', le bonbon de la plus haute valeur dans le jeu.

Bonbon 'J' : J'ai ajouté une nouvelle condition pour le bonbon 'J'. Si le bonbon actuel est 'J', il ne change pas lorsqu'il rencontre un autre 'J'. Cela résout un problème antérieur où deux bonbons 'J' fusionnaient indûment en '@'. Maintenant, ils restent 'J', conservant ainsi leur statut de bonbon de plus haute valeur.

Cas par défaut : Pour tout autre caractère ou situation inattendue, la méthode retourne un espace ' ', ce qui peut être utile pour gérer des erreurs ou des conditions non prévues.

Cette méthode montre comment le jeu gère la complexité des interactions entre les différents types de bonbons, un élément clé pour créer un défi engageant et progressif pour les joueurs.

J’ai donc présenté ici toutes les fonctions de mouvements et d’initialisation pour la grille. Je vais maintenant détailler les fonctions utiles à l’expérience de jeu de l’utilisateur et le déroulement de la partie : affichage de l’aide, des règles, vérification de la fin de la partie, comptage des points.

B) GESTION DE LA PARTIE ET EXPÉRIENCE DU JOUEUR

Ainsi, après avoir discuté de la façon dont les bonbons sont fusionnés et gérés dans le jeu, je vais maintenant me pencher sur un aspect plus interactif : la méthode DemanderNombreDeCoups.

```
156  // Demande au joueur de saisir le nombre maximal de coups pour la partie
157  // Cette méthode demande à l'utilisateur de saisir le nombre maximal de coups pour la partie.
158  int DemanderNombreDeCoups()
159  {
160      int nombreDeCoups; // Déclaration de la variable pour stocker le nombre de coups saisi par l'utilisateur.
161
162      // Boucle infinie pour demander à l'utilisateur jusqu'à ce qu'une entrée valide soit fournie.
163      while (true)
164      {
165
166          // Demande à l'utilisateur de saisir le nombre de coups.
167          Console.WriteLine("Entrez le nombre maximal de coups pour cette partie :");
168
169          // Lecture de la saisie utilisateur. 'input' peut être null, donc il est déclaré
170          // comme 'string?' pour gérer la possibilité d'une valeur null.
171          string? input = Console.ReadLine();
172
173          // Vérification si l'entrée utilisateur n'est pas null et si elle peut être convertie en un entier.
174          // 'int.TryParse' essaie de convertir la saisie en un entier et stocke le résultat dans 'nombreDeCoups'.
175          // Si la conversion est réussie, 'int.TryParse' retourne 'true' et la condition if est exécutée.
176          if (input != null && int.TryParse(input, out nombreDeCoups))
177          {
178              return nombreDeCoups; // Si l'entrée est un nombre valide, retourne ce nombre et quitte la méthode.
179          }
180
181          // Si l'entrée n'est pas un nombre valide ou est null, affiche un message d'erreur et la boucle continue.
182          Console.WriteLine("Entrée invalide. Veuillez entrer un nombre entier.");
183      }
184  }
```

Le processus est assez direct :

Demande de saisie : D'abord, la méthode invite l'utilisateur à saisir le nombre maximal de coups pour la partie. Cette étape est essentielle car elle permet au joueur de définir son propre niveau de défi.

Boucle de validation : La méthode utilise une boucle while qui tourne indéfiniment jusqu'à ce qu'une entrée valide soit saisie. Cela garantit que le jeu ne progresse pas tant que l'utilisateur n'a pas fourni une valeur acceptable.

Gestion des entrées : L'utilisateur saisit une valeur, stockée temporairement dans la variable input. Cette variable est déclarée comme string?, ce qui signifie qu'elle peut être null. Cela est important pour éviter les erreurs si l'utilisateur appuie simplement sur 'Entrée' sans taper de chiffre.

Conversion et validation : La méthode int.TryParse tente de convertir l'entrée en un nombre entier et la stocke dans nombreDeCoups si elle est valide. Si la conversion réussit (c'est-à-dire si l'utilisateur a effectivement saisi un nombre), la méthode retourne ce nombre et la boucle s'arrête.

Gestion des erreurs : Si l'entrée n'est pas un nombre ou est null, un message d'erreur s'affiche, et la méthode redemande une saisie. Cela garantit que le jeu ne commence qu'avec une valeur de coup valide, permettant une expérience de jeu cohérente.

La méthode CalculerScore : Cette méthode est au cœur du jeu car elle définit l'objectif final pour le joueur, qui est de maximiser son score.

```
566   //Calcule le score final
567   // Cette méthode calcule le score total basé sur les bonbons présents dans la grille
568   int CalculerScore(char[,] grille)
569   {
570       int score = 0; // Initialise le score à 0
571
572       // On parcourt chaque cellule de la grille
573       for (int i = 0; i < grille.GetLength(0); i++) // On parcourt les lignes de la grille
574       {
575           for (int j = 0; j < grille.GetLength(1); j++) // On parcourt les colonnes de la grille
576           {
577               // Selon le type de bonbon dans la cellule, on ajoute des points au score
578               switch (grille[i, j])
579               {
580                   case '*':
581                       // Si le bonbon est de type '*', on ajoute 1 point
582                       score += 1;
583                       break;
584                   case '@':
585                       // Si le bonbon est de type '@', on ajoute 3 points
586                       score += 3;
587                       break;
588                   case 'o':
589                       // Si le bonbon est de type 'o', on ajoute 7 points
590                       score += 7;
591                       break;
592                   case 'J':
593                       // Si le bonbon est de type 'J', on ajoute 15 points
594                       score += 15;
595                       break;
596               }
597           }
598       }
599
600       // On retourne le score total après avoir parcouru toute la grille
601       return score;
602   }
```

Voici comment j'ai conçu et implémenté cette fonctionnalité :

Initialisation du score : Tout d'abord, je définis une variable score et l'initialise à 0. Cette variable va accumuler les points gagnés en fonction des bonbons présents sur la grille.

Parcours de la grille : J'utilise une double boucle for pour parcourir chaque cellule de la grille. La première boucle passe par toutes les lignes (i), tandis que la seconde passe par toutes les colonnes (j) de chaque ligne.

Attribution des points : À l'intérieur de ces boucles, j'utilise une structure switch pour traiter les différents types de bonbons présents dans chaque cellule. Chaque type de bonbon a une valeur de points différente :

Le bonbon de base * ajoute 1 point.
Le rouleau de réglisse @ ajoute 3 points.
Le cookie o ajoute 7 points.
Le sucre d'orge J, qui est le bonbon le plus élevé, ajoute 15 points.

Calcul cumulatif : À chaque fois qu'un type de bonbon est identifié dans une cellule, le score associé est ajouté au total. Cela permet de cumuler les points pour l'ensemble de la grille.

Retour du score final : Après avoir parcouru toute la grille, la méthode retourne le score total calculé. Ce score représente la performance du joueur dans cette partie spécifique.

```
105  void AfficherAide()
106  {
107      Console.WriteLine("\nBienvenue dans XMAS TREATS !");
108      Console.WriteLine("Objectif : Rassemblez des bonbons pour marquer des points.");
109      Console.WriteLine("Règles :");
110      Console.WriteLine(" - Déplacez les bonbons pour les faire fusionner deux à deux.");
111      Console.WriteLine(" - Chaque bonbon a une valeur de points différente.");
112      Console.WriteLine(" - Vous avez un nombre limité de mouvements pour atteindre le meilleur score.");
113      Console.WriteLine("\nCompte des Points :");
114      Console.WriteLine(" - Bonbon '*' : 1 point");
115      Console.WriteLine(" - Bonbon '@' : 3 points");
116      Console.WriteLine(" - Bonbon 'o' : 7 points");
117      Console.WriteLine(" - Bonbon 'J' : 15 points");
118      Console.WriteLine("\nCommandes :");
119      Console.WriteLine(" - 8 pour haut ⬆");
120      Console.WriteLine(" - 6 pour droite ➡");
121      Console.WriteLine(" - 4 pour gauche ⬅");
122      Console.WriteLine(" - 2 pour bas ⬇");
123      Console.WriteLine("\nBonne chance !\n");
124  }
```

J'ai créé la fonction AfficherAide. Voici comment elle fonctionne et ce qu'elle apporte au jeu :

Accueil et Introduction au Jeu : Au début de la fonction, j'affiche un message de bienvenue qui introduit le joueur au jeu "XMAS TREATS".

Explication de l'Objectif du Jeu : Je clarifie ensuite l'objectif principal du jeu, qui est de rassembler des bonbons pour marquer des points.

Détail des Règles : Je décris les règles de base du jeu, notamment la nécessité de déplacer les bonbons pour les fusionner et l'existence d'une limite de mouvements.

Système de Points : Une partie importante de l'aide est le détail du système de points. Chaque type de bonbon a une valeur différente, ce qui est clairement expliqué.

Commandes de Jeu : Enfin, j'inclus une section expliquant les commandes pour déplacer les bonbons dans différentes directions.

```
295  // Vérifie si la partie est terminée
296  bool VerifierFinDePartie(char[,] grille, int nombreDeCoups, int coupsJoues)
297  {
298      if (coupsJoues >= nombreDeCoups) return true; // Nombre maximal de coups atteint
299      return false;
300  }
```

J'ai développé la fonction VerifierFinDePartie pour déterminer si une partie est terminée, basée sur le nombre de coups joués. Elle compare simplement le nombre total de coups joués avec le maximum autorisé. Si le joueur a atteint ou dépassé ce nombre, la fonction renvoie true, indiquant que la partie est finie. Cette approche garantit que le jeu ne continue pas au-delà de la limite fixée.

III) MAIN

Dans la fonction Main de mon jeu "XMAS TREATS", j'ai effectué plusieurs améliorations pour une meilleure expérience utilisateur. Voici un résumé de mon code :

Titre et Aide : Au démarrage, le jeu affiche son titre. Ensuite, j'ai implémenté une boucle do-while pour demander au joueur s'il a besoin d'aide. Cette boucle garantit que l'utilisateur fournit une réponse valide ('1' pour oui, '0' pour non). Si l'utilisateur demande de l'aide, la fonction AfficherAide est appelée.

Initialisation : J'initialise le meilleur score à zéro et un indicateur premierePartie pour suivre si c'est la première partie jouée.

Boucle Principale du Jeu :

Création de la grille : La grille de jeu est créée et initialisée avec CreerEtInitialiserGrille.
Nombre de Coups : Je demande au joueur le nombre maximal de coups pour la partie avec DemanderNombreDeCoups.
Gestion du Jeu : Dans une boucle do-while, le jeu continue tant que la fin de partie n'est pas atteinte. À chaque tour, j'affiche la grille et les infos avec AfficherGrilleEtInfos, puis gère les mouvements du joueur avec GererMouvements. Si le joueur atteint le nombre maximal de coups ou s'il n'y a plus de mouvements possibles, la partie se termine.

Fin de Partie :

Calcul du Score : À la fin de chaque partie, je calcule le score avec CalculerScore et l'affiche.
Meilleur Score : Je mets à jour le meilleur score atteint et l'affiche. Si c'est la première partie, un message spécifique est affiché.
Option de Rejouer : Enfin, je demande au joueur s'il souhaite rejouer. Si la réponse est '1', la boucle principale se répète, sinon le jeu se termine.
Ce code structure l'expérience de jeu, en offrant une introduction claire, un déroulement de jeu fluide et des interactions utilisateur cohérentes.

III) Tableau d’organisation

Date	Tâche Effectuée	Description
24 Novembre	Initialisation du Projet	Définition des objectifs du jeu "XMAS TREATS", choix des technologies et de l'environnement de développement.
26 Novembre	Création de la Grille du Jeu	Programmation de la fonction CreerEtInitialiserGrille pour générer la grille de jeu 4x4.
28 Novembre	Gestion des Mouvements et Fusions	Développement des fonctions GererMouvements, DeplacerVersBas, FusionnerBonbonsHaut, etc., pour gérer les déplacements et les fusions des bonbons.
30 Novembre	Implémentation de la Logique de Fin de Partie	Codage de la fonction VerifierFinDePartie pour déterminer les conditions de fin du jeu.
2 Décembre	Ajout de la Fonction d'Aide et Améliorations UI/UX	Ajout de la fonction AfficherAide et amélioration de l'interface utilisateur pour une meilleure interaction.
4 Décembre	Gestion des Scores et Option de Rejouer	Programmation de la fonction CalculerScore et ajout de la possibilité de rejouer à la fin d'une partie.
6 Décembre	Tests et Débogage	Réalisation de tests complets du jeu pour identifier et corriger les bugs.
7 Décembre	Rédaction du Rapport - Partie 1	Début de la rédaction du rapport, couvrant l'introduction, la description du jeu et le début de l'explication du code.
10 Décembre	Rédaction du Rapport - Partie 2	Poursuite de la rédaction du rapport, détaillant les aspects techniques et les fonctionnalités clés du jeu.

12 Décembre	Finalisation et Révision du Rapport	Finalisation de la rédaction du rapport, révision et ajustements pour assurer la cohérence et la clarté.
14 Décembre	Finalisation du programme et transfert à Github	Transfert du programme vers Github et ajout de ce présent tableau au rapport

IV) Conclusion

Dans le cadre de la création du jeu "XMAS TREATS", j'ai pu approfondir mes connaissances en programmation, notamment en C#. Un aspect particulièrement instructif a été l'utilisation des structures de contrôle comme les instructions switch. Cela m'a aidé à gérer les entrées des utilisateurs de manière efficace et à contrôler la logique de fusion des bonbons dans le jeu. Cette expérience m'a montré l'importance d'écrire un code clair et bien structuré.

Par ailleurs, travailler avec des tableaux et des matrices pour représenter la grille de jeu a été crucial. J'ai ainsi pu gérer l'état du jeu de manière plus intuitive et précise.

Enfin, j'ai eu l'occasion d'utiliser des méthodes telles que Parse et TryParse pour le traitement des entrées, Console.ReadKey().KeyChar pour la saisie des commandes, et Clone pour dupliquer des objets. Ces techniques m'ont permis de gérer les entrées de l'utilisateur de façon fiable et de comprendre des aspects plus subtils de la manipulation d'objets en C#. Cette expérience m'a apporté des compétences techniques essentielles et a élargi ma compréhension de la programmation, tout en renforçant ma capacité à résoudre des problèmes complexes.