GameCodeur – Gaming Campus

Soutenance Monogame – C#

Simon Foucher

*« Dice Roll Breakout »*

Une image contenant dessin humoristique, Animation, affiche, fiction

Description générée automatiquement

COMMENT JOUER ?

**LE JEU**

« Dice Roll Breakout » est le prototype d’un jeu de casse-brique 2D dans lequel le joueur incarne un mage qui doit défendre son sanctuaire magique d’une invasion de monstres. Dans ce monde, la magie n’existe que sous la forme de dés, accordés par les dieux bons… comme les mauvais. Aussi, toute magie est fruit du hasard : l’apparition des monstres, les pouvoirs du mage…

**PRINCIPE DE GAMEPLAY**

Le gameplay repose sur les bases d’un jeu de casse-brique classique, mais comporte des mécaniques inspirées du jeu de rôle sur table :

Phase 1 : Le tirage de dés

1. A chaque début de niveau, le joueur fait face à des rangées de « briques », ici devenues des dés de rencontres : selon les niveaux, il y a des dés 3, des dés 4, des dés 6, des dés 8, des dés 10, des dés 12 et des dés 20.
2. Une image contenant capture d’écran, dessin humoristique, Logiciel de jeu vidéo, Animation

   Description générée automatiquementUne image contenant capture d’écran, Bleu Majorelle

   Description générée automatiquementEn cliquant sur chaque dés, le joueur obtient un chiffre définissant le niveau d’un monstre qu’il devra affronté. Chaque dés est alors remplacé par le monstre en question.

Phase 2 : Le combat

1. Chaque monstre a un nombre de points de vie que le joueur doit faire descendre à zéro pour le tuer. Pour cela, il dispose d’un projectile qu’il peut tirer : c’est la balle classique du casse-brique, qui rebondit contre les murs.
2. Quand le joueur a tué tous les monstres, il passe au niveau suivant, jusqu’à finir le dernier niveau et remporter la victoire.

Les actions possibles et événements

1. Lorsqu’il touche un monstre, le joueur collectionne des points qui remplissent une jauge. Lorsque cette jauge est pleine, le joueur obtient un dé magique qui se lance pour déterminer quel pouvoir il va obtenir. Ces pouvoirs sont au nombre de six :
   1. Avoir une deuxième balle
   2. Augmenter les dégâts de la balle pendant 5 secondes
   3. Tuer un monstre à l’impact
   4. Détruire une ligne de monstre à l’impact
   5. Détruire tous les monstres du même type à l’impact
   6. Détruire un carré de monstre 3x3 à l’impact
2. Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

   Description générée automatiquementUne image contenant texte, capture d’écran, Logiciel multimédia, logiciel

   Description générée automatiquementSi le joueur perd la balle en cours, les monstres descendent sur la grille : lorsqu’ils atteignent les limites de jeu, les monstres se mettent à attaquer le joueur qui a lui aussi une barre de PV.

*Exemple à gauche en mode normal, à droite avec un pouvoir activé (dégâts)*

**COMMANDES**

Les commandes de Roll Dice Breakout sont les suivantes :

* **Déplacement :** par touches ***Q gauche, D droite*** ou par **flèches directionnelles gauche droites.**
* **Tir :** Lancer de balle en visant avec la souris et **clic gauche** de la souris pour la lancer.
* **Pouvoir :** A l’obtention d’un pouvoir, **clic gauche** à nouveau pour l’activer : le pouvoir, une fois chargé, se déclenche au moment de l’impact avec un monstre.
* **Touche Echap** pour mettre en pause. **Espace** pour reprendre ou de nouveau **Echap** pour quitter.

**Touches cheat** : pour le debug ou pour tester plus rapidement, la touche **P** permet de remplir la jauge de pouvoir à ras-bord, la touche **M** fait descendre les monstres et la touche **W** fait passer au niveau suivant.

**MODE EDITEUR**

En mode éditeur, tout se fait au clic :

* **Ajouter un dé :** clic gauche sur un emplacement de la grille
* **Supprimer un dé :** clic droit sur un emplacement de la grille
* **Changer de dé :** clic gauche sur une des flèches

**COMMENT GAGNER ?**

Pour gagner le jeu, le joueur doit détruire l’intégralité des ennemis présents dans les 10 niveaux du jeu.

**COMMENT PERDRE ?**

Le joueur perd à force de perdre des balles. Il y a deux conséquences à la perte de balle, qui peuvent entrainer deux défaites différentes :

1. **Mort :** Lorsque le joueur perd une balle, les monstres descendent d’une case ou deux selon leur vitesse. S’ils atteignent la limite de jeu, ils attaquent le joueur qui perd alors des PV. Si le joueur tombe à zéro, il a perdu.
2. **Plus de munitions :** Lorsque le joueur tire une nouvelle balle, il perd une « munition » indiquée sous le portrait du mage. Lorsqu’il n’a plus de munition, la partie est perdue. Ces munitions sont remises au maximum à chaque changement de niveau, ce qui laisse dix essais au joueur.

ORIGINALITE

**SPECIFICITES DE GAMEPLAY**

Les spécificités du gameplay par rapport à un jeu de casse-brique classique sont doubles :

1. La première originalité est l’utilisation de monstres à PV au lieu de briques. Ces monstres descendent et peuvent venir vous agresser. Il s’agit principalement de la classe **Monster.cs** qui hérite de **Bricks.cs** et qui dispose d’un **MonsterAnimator.cs, MonsterFighter.cs** pour gérer le combat et **MonsterHealth.cs** et MonsterHealthUI.cs pour gérer la vie. **Le Player.cs,** qui remplace le pad ici, est aussi composé d’un **PlayerHealth, PlayerFighter, PlayerAnimator** etc. Monstre et joueur héritent tous deux des mêmes **Health.cs, Fighter.cs, Animator.cs** de base.
2. La seconde originalité vient de l’utilisation des dés : les dés de rencontre définissent les monstres, ce qui veut dire que même si on refait deux fois le niveau, ce n’est pas exactement le même. Le dés magique définissant les pouvoirs que le joueur va gagner. Pour cela, j’ai utilisé un **DicesManager.cs** qui, a à la demande de mon **LevelManager.cs** va être capable d’appeler deux factories : un **DicesFactory** pour créer des dés sur la grille de jeu et un **MonstersFactory** qui va créer les monstres à la place des dés une fois les dés lancés, avec le niveau adapté. Les dés existent sous la forme d’une classe **Dice.cs**, qui possède une interface IBrickable pour pouvoir être mis dans la grille, et mes monstres sont issus de **Monster.cs** qui hérite de Bricks.cs qui implémente également une interface **IBrickable.**
3. Enfin, l’éditeur de niveau intégré au jeu est original également pour les mêmes raisons : le joueur place des dés sur la grille au lieu de placer des briques, ce qui veut dire qu’il n’a pas la maitrise totale du niveau qu’il fait.

**ALGORITHMES SPECFIQUES**

Pour créer ces originalités, mes algorithmes spécifiques sont à la fois le **Dice.cs**, le **BaseGrid.cs**, le **Dice.cs** ainsi que ceux liés aux pouvoirs du joueur, les classes héritantes de **PowerManagers.cs** à savoir **PlayerPowerManager.cs** et **BallPowerManager.cs**.

**Dice.cs** est le plus simple, puisqu’il s’agit d’afficher le bon sprite de dé via un switch au constructeur, tirer un chiffre aléatoire et jouer une animation de dé avant de renvoyer un résultat.

**BaseGrid.cs** représente ma grille : il s’agit d’une liste de Vector2 qui peuvent être convertis en « slot », c’est-à-dire en rectangle de la taille de mes briques. Ces slots ont un ID chacun. Grâce à cela je peux attribuer un slot à un monstre pour savoir où il se trouve et le faire bouger sur cette grille : descendre, monter etc. En comparant l’ID slot voulu et l’ID slot sur lequel est le monstre (Les classes implémentant l’interface IBrickable.cs possèdent une information ID slot) je peux savoir si une case est occupée ou pas. C’est aussi grâce à ces slots que fonctionne mon éditeur de niveau qui regarde si au moment de cliquer la souris du joueur est sur l’un des slots.

Enfin, les **PowerManager** dépendent du résultat du **MagicalDice.cs** et insert dans une propriété Power du joueur ou de la ball le pouvoir correspondant au résultat.

CODE SOURCE

**STRUCTURE DU CODE**

**Les différentes scènes**

Mon projet commence par un MainGame.cs qui va Update et Draw la scène en cours. Ces scènes sont au nombre de 5 :

* **SceneMenu.cs** : menu de démarrage du jeu invitant le joueur soit à « Play », soit à « Editer » un niveau, soit à « Load » un niveau.
* **SceneGameplay.cs** : la scène comprenant le jeu à proprement parler.
* **SceneGameOver** : un écran de Game over.
* **SceneWin** : un écran de victoire.
* **SceneEditor** : un écran permettant l’édition d’un niveau.

**MAIN GAME**

Le main game enregistre tout d’abord les services pour mon ServicesLocator si ces services sont nécessaires dès maintenant et à l’ensemble du jeu, enregistre les input communs à tout le jeu, update et draw la scène courante en s’appuyant sur un Gamestate.cs qui contient la liste des scènes et celle en cours, ainsi que la méthode pour changer de scène. MainGame demande à Gamestate SceneMenu.cs comme première scène.

**SceneMenu**

SceneMenu possède trois boutons : le bouton Play permet de passer à la scène SceneGameplay. Le menu éditer à SceneEditor, le menu Load lance également la SceneGameplay, mais en modifiant un paramètre pour que le JSON à charger ne soit pas le même (savedLevels.json au lieu de levels.json).

**SceneGameplay**

ScèneGameplay est la scène du jeu, c’est ici que la plupart des choses se passent. La scène contient un GameManager qui va gérer la logique du jeu. Ce dernier crée :

* Un nouveau joueur (Player.cs)
* Un levelManager

GameManager gère alors les logiques d’interaction entre le joueur et les éléments que lui renvoie le LevelManager en s’appuyant notamment sur les états du LevelManager.

**LevelManager**

Le LevelManager a une machine a état : l’état « dices », l’état « play », l’était « win » et l’état « end ».

Il commence sur Dices : c’est la phase où le joueur doit lancer des dès, il ne peut pas « jouer ».  Quand tous les dès sont lancés, LevelManager passe en « play » : là, le joueur peut lancer une balle, taper les monstres etc. C’est le jeu. Lorsque le joueur gagne un niveau, le LevelManager passe en « Win ». Le GameManager lui demande alors de passer au niveau suivant. Lorsqu’il n’y a plus de niveau parce que c’était le dernier, le LevelManager passe en « End », ce qui permet au GameManager de passer le jeu en « IsGameWin = true ».

Pour fonctionner, le LevelManager s’appuie sur trois éléments :

* **LevelLoader**: il crée un levelLoader et lui demande de récupérer les informations du niveau en cours via un JSON. LevelLoader lui renvoie une liste de Int.
* **DicesManager** : LevelManager récupère la liste de Int et la transmet au **DicesManager**, la classe responsable de l’état des dès. Le DicesManager va convertir la liste en liste de dés via une **DicesFactory**.
* **BaseGrid** : ensuite le LevelManager va créer une grille et va demander au DicesManager de mettre les dés dans la grille qui lui transmet.

Le LevelManager interroge régulièrement le DicesManager pour lui demander l’état des dés. S’il reste des dés rollables ou non. Si un dé est roll et a un résultat, le LevelManager demande au DiceManager de les convertir en monstre. A ce moment-là, le DicesManager remplace le dé par un objet Monster du même niveau que le résultat du dé via une **MonstersFactory**. Quand il n’y a que des monstres dans la grille, LevelManager passe en « play ».

**Player.cs**

Le player possède un **PlayerPowerManager** pour gérer ses pouvoirs et plusieurs composants pour fonctionner : un **playerCollisionsManager**, un **PlayerHealth** et **PlayerHealthUI** pour la vie, un **PlayerAnimator** pour animer son sprite, un **PlayerInput** pour les touches utilisateur qui communique avec un **PlayerMovement** pour le déplacer. Mais surtout, il y a un **PlayerFighter** qui concerne sa capacité à combattre. Dans ce **PlayerFighter** on retrouve une liste de balles (Ball.cs) et les méthodes pour tirer.

Chaque balle possède également ses propres composants : un **BallCollisionManager**, un **BallHitManager** pour les dégâts qu’elle met, un **BallMovement**, un **BallParticlesManager** pour afficher des particules sur son trajet, et un **BallPowerManager** pour les pouvoirs qui lui sont propres.

**Monster.cs**

De la même manière, les monstres ont des composants **MonsterAnimator**, **MonsterFighter**, MonsterHealth etc. Ils héritent tous des mêmes classes abstraites pour le joueur ou les monstres : **Animator.cs**, **Fighter.cs**, **Health.cs**, **HealthUI.cs**…

**SCHEMA DE LA STRUCTURE**

*Une image contenant texte, capture d’écran, diagramme, nombre

Description générée automatiquement(Voir PDF pour meilleure lisibilité)*

**SERVICE LOCATOR**

**Liste des services**

Pour la création de mon ServiceLocator, j’ai créé plusieurs services :

**AssetsManagerService.cs** embarquant une interface **IAssetsServices** : gestion des assets récurrents dans le jeu.

**FontService.cs**, qui embarque une interface **IFontService** : gestion des fonts du jeu

**InputService.cs**, qui embarque une interface **IInputService**: gestion des inputs utilisateurs (clavier, souris)

**MediaPlayerService.cs**, qui embarque un **IMediaPlayerService**: lis les musiques et les sons.

**PathsService.cs**, qui embarque un **IPathsService** : définition des chemins aux images, son etc.

**PlayerSessionService**, qui a une interface **ISession** : contient le score du joueur, sa vie et ses points.

**Que facilitent-ils ?**

Le ServiceLocator et le système de services offre une alternative aux classes statiques parfois peu modulables. Il me permet de garder un certain niveau d’abstraction dans les classes qui n’ont pas forcément besoin de connaître une information (voir la partie exemple). Cela me permet également de faciliter la maintenance en centralisant les informations et les dépendances : si je souhaite modifier la touche sur laquelle l’utilisateur doit appuyer pour faire une action, j’ai seulement à modifier le contenu de mon **InputService.cs.** Il en va de même pour mon **PathsService.cs** qui me permet d’avoir un seul endroit où indiquer les chemins d’accès à mes fichiers. De même, si je veux changer d’implémentation, il me suffira d’enregistrer dans mon ServiceLocator une autre classe avec la même interface, plutôt que de modifier toutes les références comme j’aurais pu le faire avec une classe statique.

**Exemple de service**

Par exemple, mon **PlayerInput.cs**, classe qui gère le comportement de mon joueur/pad selon les inputs utilisateur, n’a pas besoin de connaître la touche sur laquelle j’appuie : je passe à la place par une méthode de mon **InpusService.cs**, ce qui me permet d’avoir un code très lisible et compréhensible, sans surcharger d’informations inutiles ou exposer ce qui n’a pas à être exposé :

**Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement**

**A**insi, le PlayerInput.cs n’a aucune idée des inputs utilisés. Demain, si je veux changer la façon dont c’est implémenté, je pourrais passer à mon ServiceLocator une autre classe implémentant une interface IInputService. Cela peut être une solution par exemple pour les changements de controller et enregistrer un IInputService différent selon le controller que le joueur a choisi, sans pour autant devoir mettre des « OR » partout dans mes conditions.

**EDITEUR DE NIVEAUX**

**Stratégie adoptée**

Pour mon éditeur de niveaux, j’ai choisi de séparer les niveaux de mon jeu des niveaux produits par le joueur, afin que le jeu soit toujours praticable. Ainsi, mon menu possède trois boutons :

Play, qui lance le jeu avec les niveaux classiques.   
Editor, qui permet d’éditer un niveau

Load, qui permet de charger et jouer le niveau qui a été édité.

Dans ce prototype, il n’y a qu’un niveau sauvegardable pour le joueur, mais implémenter la possibilité d’en sauvegarder plusieurs ne nécessiterait que de prévoir une UI permettant de passer d’un niveau à un autre dans l’éditeur.

Puisque mon jeu fonctionne via un levelManager qui va lire des données dans un JSON pour signaler à mon DicesManager quels dés afficher pour chaque niveau, il ne me semblait pas pertinent de changer de façon de faire pour les niveaux personnalisés, bien au contraire ! Le JSON est particulièrement adapté pour enregistrer des données, donc tout repose sur les mêmes fonctionnements :

* Le joueur dispose d’une grille et voit les dés qu’il peut utiliser. En cliquant gauche sur la grille, il peut ajouter un dé, en cliquant droit, en enlever. Lorsqu’il sauvegarde, la grille est convertie en un tableau qui est sauvegardé dans un JSON spécifique.
* Une image contenant capture d’écran, texte, Logiciel multimédia

  Description générée automatiquementAinsi, quand le joueur choisit de « Play » une partie classique ou « Load » son niveau personnalisé, il lance en fait le même GameManager, avec le même LevelManager etc. la seule différence est le JSON qui est lu : dans un cas **levels.json**, dans l’autre **savedlevel.json**.

**FORMAT DE DONNEES**

**Format de données des niveaux**

Mes niveaux sont donc sauvegardés soit dans **levels.json** pour les jeux classiques, préétablis, soit enregistrés à la volée dans un **savedLevel.json** pour le mode éditor.

La structure de ces deux fichiers est identique pour pouvoir être lu de la même manière par mon levelManager et est assez simple :

* « LevelsNb »
* « Levels » :
  + « LevelNb »
  + « Dices »
  + « LevelNb »
  + « Dices »  
    etc.

**Une image contenant texte, capture d’écran, conception

Description générée automatiquement**Il comporte donc un *LevelsNb*, qui contient le nombre total de niveau, ainsi qu’un array *Levels*, tableau de niveaux. Chaque niveau a un *levelNb*, le numéro du niveau et un array *Dices*. L’array est comporté de 0 (case vide) et de numéros représentant les dés à afficher (4, 6, 12, 20 etc).

Pour récupérer ces données en c#, j’ai donc créé une classe **Level.cs** qui contient un int levelNb et un array à deux dimensions int Dices, ainsi qu’une classe **LevelList.cs** qui contient le LevelsNb et Levels, l’array d’objet Level.

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement** **Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement**

**MON EXPERIENCE**

**Temps consacré au projet :** Durant un peu moins d’un mois, j’ai consacré environ 3 soirs par semaines et environ deux week-ends complet à ce projet. Je dirais une moyenne de 90 heures.

**Qu’ai-je appris ?**

Je connaissais déjà un peu la POO, mais j’ai découvert le vrai rôle des classes abstraites, virtual et des interfaces qui étaient pour moi des notions un peu floues et que je n’étais pas du tout capable d’utiliser. J’ai également découvert le pattern du ServicesLocator. De même, je ne connaissais pas d’autres environnements qu’Unity sur le C#, j’ai donc découvert Monogame.

**Quelles sont les difficultés que j’ai rencontrées ?**

Comme pour le projet précédent, mes difficultés ont été principalement organisationnelles au niveau du code savoir comment et à quel point découper le code en classes…

**Quels sont les domaines dans lesquels je pense devoir m’améliorer ?**

Je pense devoir m’améliorer sur le maintien d’un code propre durant la production. J’ai beaucoup trop tendance à faire des lignes et des lignes de code jusqu’à ce que ça marche, puis à « cleaner » le code en recoupant tout en plusieurs classes, mais cela prend beaucoup de temps. Je dois prendre plus le temps de réfléchir en amont pour avoir une idée du bon découpage dès le commencement.