ArcheoBeat : Documentation Technique

Comment jouer ?

Dans *ArcheoBeat* on incarne un petit robot chargé d'explorer d'étranges ruines mécaniques.

Pour ce faire, le joueur doit parcourir une série de niveaux du départ à l'arrivée en esquivant les pièges et sans tomber dans le vide.

Cependant, les ruines évoluent au rythme d'une étrange musique et le robot est lui-même soumis à cette règle : les déplacements du joueur se font case par case et doivent être en synchronisation avec le tempo, sous peine de défaillance mécanique.

*Pour la suite de ce document on utilisera le terme **beat** pour parler d'un temps de la musique.*

Pour l'aider dans sa tâche, le robot dispose d'un canon laser permettant de détruire certains obstacles. Celui-ci peut être tiré à condition d'être en rythme et prend un beat à se recharger.

Le robot possède un certain nombre de vies. Lorsqu'il chute dans le vide, le joueur perd une vie et réapparaît au départ ou au dernier checkpoint sans que le niveau ne se réinitialise. Si il chute sans qu'il ne lui reste de vie, il repart à 0 et le niveau est réinitialisé.

Concepts / Originalité

ArcheoBeat ne reprend pas à proprement parler les mécaniques du *Snake* mais possède bien des mécaniques de grille avec des déplacements discrets case par case.

Il s'inspire en revanche de jeux comme *Crypt of the NecroDancer* avec un système de rythme venant conditionner les déplacements et les actions du joueur.

Un autre concept assez central (et qui, je pense, crée l'originalité par rapport à ces jeux) est le réarrangement de certaines parties des niveaux en rythme avec la musique. Je me suis inspiré pour cela des algorithmes de type *wave function collapse* pour que les niveaux restent toujours cohérents et réalisables.

Le code

La structure du code s'organise autour d'un *Service Locator* servant d'aiguilleur entre les principaux services du jeu.

Certains services sont globaux au jeu, comme l'*AudioManager* ou le *SceneManager*, tandis que d'autres sont chargés et déchargés lors des changements de scènes.

La majeure partie de ces services sont nommés *[objet/concept]Manager*, cela correspond à des classes dont la fonction est de gérer et centraliser la manipulation de certains types d'objets. Les managers possèdent d'ailleurs (presque) tous une méthode Draw et Update.

Services globaux principaux :

- **GameParameters** : contient des paramètres généraux du jeu (propriétés de la fenêtre, target FPS…)
- **MyCamera** : caméra du jeu (version *class* du *struct* "*Camera*" de Raylib)
- **InputController** : écoute les *inputs* du joueur et *invoke* des évènements correspondant auxquels tous les objets du jeux peuvent souscrire
- **SceneManager** : charge et décharge les différentes scènes, actualise et affiche la scène en cours, charge et sauvegarde la progression des niveaux
- **AudioManager** : actualise les musiques en cours, joue les *sound FX* et gère le tempo des niveaux notamment en faisant des *invoke* d'évènements pour les beats, auxquels tous les objets peuvent souscrire

Le script principal (*program.cs*) charge ces services globaux, charge la première scène via le *SceneManager*, puis lance la boucle while de Raylib dans laquelle il actualise l'*InputController*, actualise la scène en cours et affiche la scène en cours (via le *SeneManager*). La première scène chargée étant le menu principal permettant de lancer les différents niveaux, tout peut s'enchaîner à partir de là.

Il existe deux types de scènes, tous deux héritant de la *class Scene* : *SceneMenu* et *SceneLevel.* Le premier type correspond au menu principal et le second correspond à un niveau. *SceneLevel* possède un constructeur particulier auquel on passe l'identifiant du niveau à charger, celui-ci sera utilisé par différents managers dont l'initialisation est propre au niveau (ex : *MapManager*).

Services de scène principaux :

- **UIManager** : gère l'affichage et la maj des différents éléments du GUI, comme la vie, l'animation du tempo, le menu de pause. etc… Il donne également accès aux différentes polices du jeu via des champs statiques.
- **PlayerManager** : gère et donne accès au personnage joueur

- **MapManager** : initialise les paramètres de la grille ainsi que toute la structure du niveau (celle-ci est composée d'un ensemble de blocs de 3x3 cellules disposés sur la grille). Gère la maj des blocs dynamiques en fonction du tempo ainsi que les différents effets que peuvent avoir les cellules sur le joueur et inversement (certaines cellules sont détruites avec le passage du joueur, d'autres sont détruites par un tir de celui-ci…)
- **ScissorsManager** : gère les ennemis du jeux, à savoir des ciseaux volant qui poussent le joueur dans le vide

Autres class importantes :

- **MapSpecifics/MobSpecifics** : contiennent dans des champs statiques les données nécessaires à l'initialisation de chaque niveau (respectivement pour la map et pour les ennemis). Ces champs sont récupérés dynamiquement par les managers correspondant à partir de l'identifiant de niveau de la scène en cours
- **Sprite** : élément graphique de base permettant d'afficher une image, animée ou non, dont on peut modifier la position, l'angle, le scale, l'origine, l'animation, etc… Permet également de renseigner des callbacks à appeler sur des frames d'animation spécifiques.
  La *class* possède également une partie statique permettant de lister des sprites à afficher dans un ordre forcé (avant le *draw()* du *MapManager* ou après celui du *PlayerManager* par exemple), pratique si le *draw()* du manager en charge du sprite n'est pas idéalement ordonné.
- **TempAnimManager** : permet de créer et gérer des animations éphémères (type *Anim* hérité de *Sprite*)

Zoom sur la Map

La map est constituée d'un ensemble de blocs (*CellsBlock*) eux-mêmes constitués de 9 cellules (*Cell*) positionnées sur la grille.

Les dimensions et caractéristiques globales d'un bloc se trouvent dans *CellsBlockDefinition* et les dimensions et caractéristiques globales d'une cellule dans *CellsDefinition.*

Le type du bloc détermine de quelles cellules il est constitué, les patterns de chaque type étant référencés dans le champ *Blocks* de *CellsBlockDefinition*.

Il existe différents types de cellules qui déterminent entre autres l'apparence d'une cellule ainsi que les effets qu'elle peut avoir lorsqu'activée par le MapManager. On trouve la liste de ces cellules en commentaire dans *CellsBlockDefinition* et *CellsDefinition.* Les caractéristiques d'une cellule sont encodées dans un identifiant à deux digits, le premier correspondant au type et le second à un marqueur de forme (cf listes en commentaire des *[...]Definition*).

En plus d'un tableau de cellules, un bloc possède plusieurs caractéristiques :

- *IsFixed* : indique que le bloc est statique et que son type ne doit donc pas être modifié en fonction des *beats*
- *IsAllwaysFixed* : indique que le bloque sera toujours fixe (permet de faire la distinction avec un bloc temporairement fixe parce que le joueur est dessus)
- *IsHectic* : indique que le bloc non fixe change de type à un rythme deux fois plus élevé que normal
- *Blockades* : une liste d'objets correspondant à des barrière situées sur le bloc et à leurs orbes qui doivent être détruites pour les désactiver

Tous les n *beats* le MapManager change le type de tous les blocs non fixes. Cela se fait en deux temps : on détermine les nouveaux types de blocs au beat n-1 (de façon à pouvoir afficher une preview pour que le joueur anticipe) et on applique les types déterminés lors du *beat* suivant.

La détermination des types de blocs se fait selon un algorithme inspiré du *wave function collapse* :

- Chaque bloc se trouve dans un état indéterminé (*IsCollapsed* = *false*) et possède un tableau de tous les types qu'il peut possiblement prendre
- Le MapManager vient "collapser" tour à tour tous les blocs
- Lorsqu'un bloc est "collapsé" le type à déterminer prend une valeur aléatoire parmi les éléments du tableau des possibilités, passe dans un état déterminé (*IsCollapsed* = *true*) et met à jour les tableaux de possibilités des blocs voisins
- La maj du tableau des possibilité se fait en fonction de tableaux de compatibilités (situés dans CellsBlockDefinition) qui spécifient pour

chaque type de bloc les types qui ne sont pas compatibles à gauche et au dessus, de façon à avoir des enchaînements de blocs cohérents et réalisables

- Lorsqu'un tableau de possibilité se voit réduit à 1 élément le bloc est automatiquement "collapsé"

A la fin de ce processus tous les blocs ont un futur état de déterminé. On l'applique aux blocs lors du *beat* suivant (sauf cas des blocs hectics qui déterminent et appliquent en même temps à chaque *beat*) cad qu'on change toutes les cellules de ces blocs en fonction du nouveau type. On exécute aussi une méthode *SmoothTiles* qui permet de modifier le marqueur de forme sur chaque cellule en fonction des cellules voisines sur les autres blocs, de façon à avoir des visuels continus.