Description de la documentation

Description diagramme de cas :

Au début de notre projet nous avions prévu de faire un système d'inventaire permettant de nous aider dans les différents niveaux. Et aussi un système de temps limité pour finir les différents niveaux. Nous avons changé d'axe, et avons développé un système de rang en fonction du temps passé avant que le joueur ne finisse le niveau. Le joueur peut toujours interagir avec le décor en l'occurrence les différentes entités présentes sur le niveau par le biais de collisions.

Un patron de conception dans notre architecture serait la façade par le biais de notre classe game.

Description diagramme de classe :

Package Launch :

Launch : Notre classe Launch est notre classe centrale réalisant le lancement de notre application et d'un stage.

Package View :

Game : Notre classe game prends le rôle d'un game manager appelant, dans un processus tournant avec une boucle infinie (thread), notre controller de mouvement de collision et notre controller récupérant nos actions clavier. Et c'est à partir de ses controllers que seront appelé nos différentes classes métiers.

Accueil : La classe accueil représente les fonctionnalités liées à la vue de même nom (l'écran principal). Elle possède deux méthodes qui sont :
cliqueSurBoutonNewGame() : qui lancera notre jeu une fois cliquée dessus, et la fonction cliqueSurBoutonQuittter() : qui permettra de fermer cet onglet.

Package Model :

Manager : Cette classe sert à gérer notre persistance, en outre a gérer la sauvegarde d'un score à la fin d'une partie. Il gère les implémentations au niveau des scores à travers les classes de persistances.

IPersistance : C'est une interface de notre classe de persistance ISauvegardeTxt. Cette classe contient uniquement les méthodes de cette classe.

ISauvegardeTxt : Cette classe va gérer l'écriture et la lecture de la sauvegarde du temps qu'a mis un joueur pour finir un niveau. Les différentes méthodes sont : sauverTemps() qui va servir à mettre en « mémoire » le temps, et « chargerTemps » qui va nous donner la liste des temps en « mémoire ».

Entite : Cette classe mère représente l'ensemble des éléments dit « entités » que notre jeu peut implémenter. Il y aura au total 5 classes filles héritant de cette classe étant : consommable, joueur, piege, plateforme et checkpoint. Ce sont tous des éléments de notre niveau qui peuvent interagir entre eux. Toutes nos « entités » étant des ImageView, cette classe possède un getter et un setter.

Checkpoint : Cette classe fille d'Entite, représente l'éléments de notre niveau qui lorsqu'il rentrera en contact avec un joueur, déclenchera la fin de ce dernier.

Joueur : Cette classe héritant d'Entite est la classe implémentant un personnage jouable. Elle possède notamment des méthodes permettant l'arrêt des déplacements et le saut d'un joueur.

Plateforme : Cette autre classe fille d'Entite permet de proposer un sol à notre niveau permettant au joueur de se déplacer dessus librement.

Piège : Cette énième classe fille d'Entite aurait permis de complexifier la complétion d'un niveau en proposant une entité, qui une fois entrée en contact avec, réduit notre vie (variable propre au joueur) qui une fois à zéro, aurait mis à la partie.

Consommable : La dernière classe fille d'Entite, cette classe, si implémentée, aurait permis d'ajouter une nouvelle fonctionnalité à notre jeu, car elle aurait l'ajout de consommables. Consommables qui auraient pu rendre plus dur ou plus simple la validation d'un niveau.

Score : Une classe nous permettant de de définir ce qu'est qu'un score afin de pouvoir le faire persister. Elle correspond à l'association d'un temps et d'un nom de joueur.

ScoreParties : Cette autre classe liée à la persistance de nos données va permettre de gérer un score, soit l'association d'un temps et d'un pseudo comme dis plus haut. Cette classe implémente des getter et des setter mais aussi la possibilité d'ajouter un score.

Package Controller :

ActionClavier : Ce controller possède un méthode : ActionLectureListe() : qui récupère une liste d'input et fait des appels à des méthodes Joueur, en fonctions des inputs reçu dans cette liste. Inputs étant de trois types Gauche, Droite ou Escape (pour quitter le jeu).

Collision : Ce controller possède trois méthodes permettant à elles de déterminer s'il y a une collision entre un Joueur et une Entite. La méthode verify() appelle deux autres méthodes vérifiant respectivement les collisions haut / bas et gauche / droite (checkCollisionHautBas() et checkCollisionDroiteGauche()) et retourne un booléen à true si l'entité rencontrer est un CheckPoint sinon false.

Mouvement : Le dernier controller possède deux méthodes qui sont : UpdateJoueur() permet la gestion des mouvements d'un Joueur et l'actualisation de ces mouvements selon si le Joueur se collisionne à gauche, à droite, en haut ou en bas. Et gestionSaut() qui vérifie les collisions actuelle du Joueur et celle qu'il peut avoir en plein air. Nous gérons aussi le saut de pars sa durée car nous avions rencontré certains problèmes, non voulu, sans cela.

Description app :

Notre est jeu est un « plateformer » en 2D, ou le but est de finir le niveau le plus rapidement possible. Pour ce faire il faut déplacer le joueur sur une plateforme jaune. Une fois atteinte le niveau prends fin et le joueur voit son temps enregistré avec son pseudo. Il était prévu de faire un ensemble de niveau afin de développer plus de jouabilité.

Les déplacements se font à l'aide des flèches directionnelles gauche et droite pour les mouvements dans ces directions respectives, mais aussi à l'aide de la flèche haut permettant au joueur d'effectuer un saut. Un système de collision dynamique a été implémenté pour répondre à un certain nombre de besoins.