Compte rendu SAE 01 – 02



**Silver Man : Joyride**

TABBICHE Malik, GINET Tristan et SPORTIELLO Mathis

Sommaire :

1. Introduction
2. Cahier des charges
3. Réalisation
4. Résultat
5. Introduction

Dans le cadre de notre SAE 1-2, nous avions à créer un jeu en C# en utilisant le moteur Monogame. Nous avions donc à le maitriser afin de rendre une version de notre jeu aussi fidèle que possible de ce que nous imaginions.

Après quelques minutes et recherches sur internet, notre choix s’est tourné vers un remake de Jetpack Joyride. Ce jeu que tout le monde connait, celui qui a fait tant parler de lui à sa sortie en 2011 jusqu’à aujourd’hui. Nous savions alors dans quelle direction partir.



Logo Monogame

1. Cahier des charges

Pour mener à bien ce projet il a fallu établie un cahier des charges. Nous voulions obtenir un jeu dans l’esprit de Jetpack Joyride : un personnage qui se déplace verticalement dans un niveau qui défile jusqu’à échec du joueur.

Nous devions aussi concevoir ce jeu pour qu’un maximum de personne puisse y jouer, donc un maximum optimisé, intuitif, en bref simple d’utilisation !

L’objectif de ce jeu est aussi de reconnecter les anciens joueurs du jeu avec une version revisitée plus récente.

Nous avions besoin de pièges comme des « zappers » et des « rocket », d’un niveau « infini » et d’une difficulté croissante.

1. Réalisation

La première chose que nous avons implémentés dans le jeu est le personnage, il fallait qu’il soit assez petit pour rendre le jeu jouable dans l’écran propose mais aussi assez gros pour pouvoir rendre le jeu compétitif. Nous l’avons placé à la bordure interne gauche du jeu pour pouvoir prévoir et réagir aux pièges qui arrivent du côté droit.

Ce personnage peut seulement bouger sur le plan vertical. Nous avons alors rencontré un premier problème : ses mouvements étaient trop rectilignes. Nous avons donc dû simuler une gravité. Nous avons utilisé une suite u(n+1) pour accroitre sa position en Y chaque millisecondes un peu plus lorsque l’on appuie sur Espace et de la décroitre par la même suite facteur 3 lorsque l’on lâche la touche Espace.



Personnage du jeu

La seconde chose à considérer était le décor, comme dit précédemment, nous voulions créer un niveau infini, qui se génère automatiquement. Pour ce faire, nous avons simplement créé une classe qui, à chaque fois que l’image est centrée, crée une image similaire collée à l’ancienne. Nous trouvions ce système intéressant et avons donc voulu ajouter une sorte de parallaxe, un faux 3D. Pour cela, nous avons créé une liste de fond d’écran avec exactement les mêmes paramètres que le premier fond d’écran en modifiant juste leur vitesse qui varie de plus lente au dernier plan à plus rapide au premier plan. Puis nous avons augmenté le niveau de difficulté du jeu en fonction du temps passé à jouer pour rendre le jeu compétitif et apporter une expérience évolutive.

Une image contenant texte, usine, lumière, cité

Description générée automatiquement

Rendu final du niveau

En ce qui concerne les pièges, il y en a deux : le zapper et la rocket.

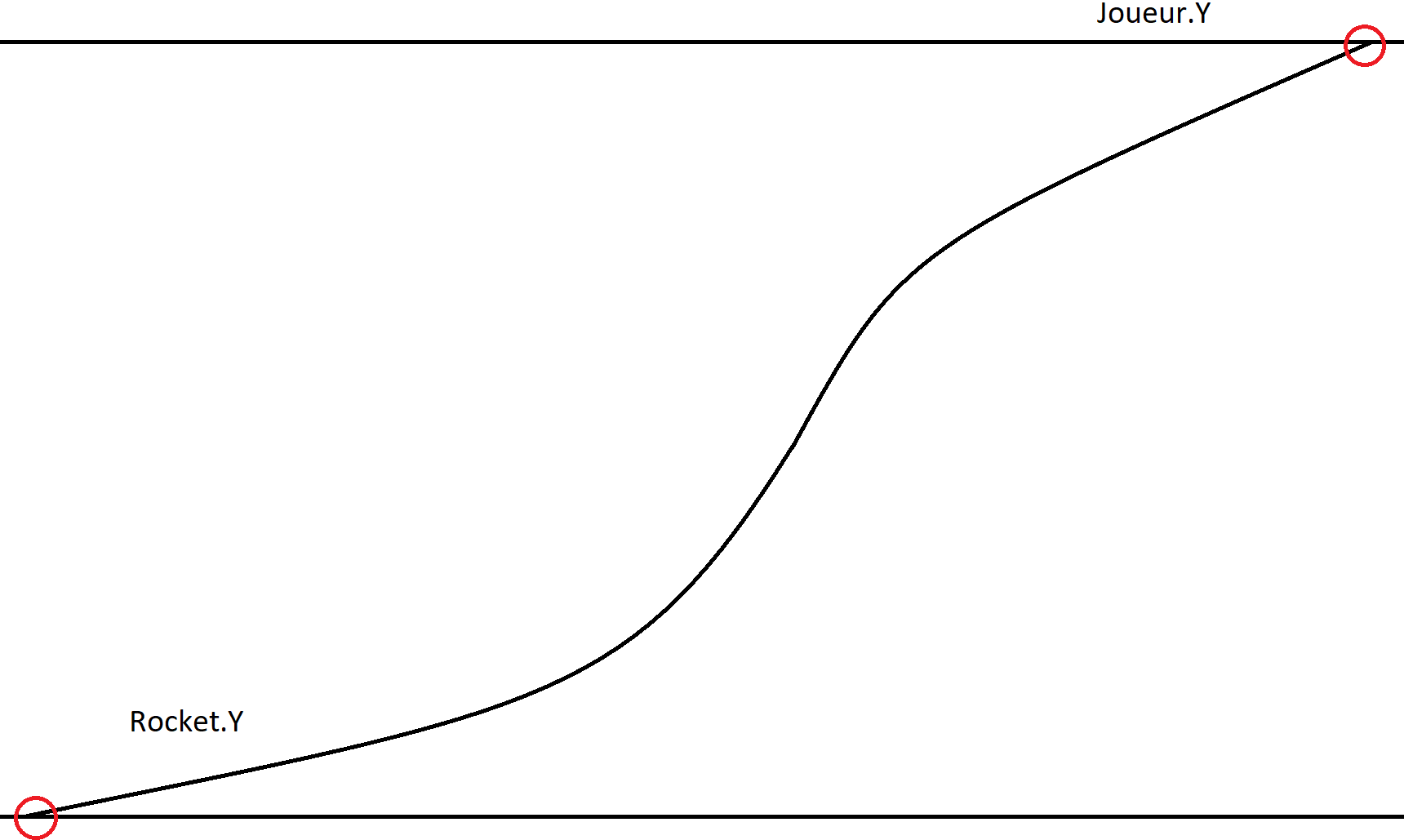
Le zapper est un arc électrique tournant sur lui-même qui a la même vitesse que le sol. Ce qui le rend difficile à éviter c’est sa grande taille, sa rotation et son positionnement aléatoire sur l’axe Y qui nous force à changer régulièrement de position. Pour le créer, nous lui avons donné une position aléatoire Y dès qu’il dépassait les quelques premiers pixels négatifs sur l’axe X ce qui le rendait totalement invisible. A ce point, nous lui redonnions une coordonnée X positive elle aussi invisible à l’écran pour qu’il puisse rechallenger le joueur. A cela s’ajoute sa rotation qui est générée par un ajout à chaque update d’angle.

Une image contenant lumière

Description générée automatiquement

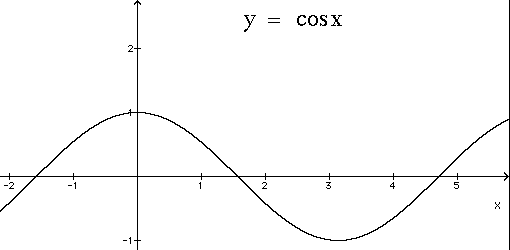
Représentation du zapper

La rocket est quant à elle une fusée qui est initialisée au milieu de l’écran en Y à une position X positive et aléatoire pour lui permettre d’acquérir un avantage sur le joueur et de le mettre en difficulté en étant le plus imprévisible possible. Cette rocket dès l’ors qu’elle est générée, suit le joueur sur l’axe Y. Pour rendre l’expérience un peu plus réaliste, nous avons décidé de créer un mouvement adouci. C’est-à-dire que quand le joueur essaie d’éviter une rocket, celle-ci le suit sans à-coups et sans linéarité. Pour répondre à ce problème, nous avons défini un point fictif à l’avant de la rocket avec les mêmes cordonnées Y que le personnage et une centaine de pixels devant la rocket. L’effet à été adouci en prenant le coefficient directeur de ce point fictif et de la rocket (bloqué à une valeur pour que la rocket ne puisse pas être trop compliquée à esquiver) en soustrayant à la position Y de la rocket ce coefficient directeur à chaque update (plus de détails sur les formules utilisées dans le code fourni).



Allure de l’IA rocket

Cette rocket était encore trop prévisible, alors comment la rendre plus compliquée à éviter ? Nous avons ajouté à cette rocket un mouvement sinusoïdal ce qui lui permet d’avoir une plus grande amplitude et peut être toucher le joueur plus facilement. Nous avons utilisé la classe Math C#.



Allure de la rocket

Ce piège est complètement vil mais vous pouvez compter sur l’alerte pour vous prévenir de son arrivée imminente. Ce signal s’enclenche lorsque la rocket est sur le point d’être visible et vous montre à quelle altitude il se situe. Pour coder ce signal, nous avons prit les coordonnées X de la rocket, puis nous avons mit une détection de celui-ci qui lorsqu’elle est activée déclenche le signal. Pour ses mouvements, nous avons lui avons attribué les coordonnées Y de la rocket et fixé sa position à la bordure interne droite de la fenêtre.



Signal warning !

Pour arriver à nous repérer dans les niveaux de difficulté du jeu, nous avons mis un système de secteur qui nous indique si nous somme dans une difficulté paisible (A), facile (B), intermédiaire (C) et difficile (D). Celui-ci s’actualise en fonction du temps passé en jeu et donc est corrélé à la vitesse du niveau.