Dustopia - Compte rendu 4

Anouk OMS, Lysandre TERRISSE, Elora ROGER, Thomas DAILLY

Séance du 21 mars 2023

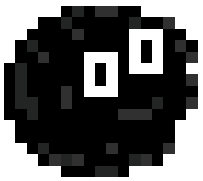

Table des matières

1 Introduction

Nous avons beaucoup avancé depuis la dernière fois sur Dustopia. Nous avons pu implémenter l'algorithme de mouvements, de gestion du temps et de la fenêtre, d'apparition des monstres, et de points de vie. Nous avons aussi implémenté certains graphismes, comme les blocs et l'arrière plan, ainsi que quelques musiques. Pour finir, nous avons créé un niveau de test dans lequel le joueur doit éviter de tomber dans le vide. Les tâches principales qui nous restent à faire sont les transitions entre les niveaux, l'intelligence des ennemis, et la création des menus. Les problèmes les plus importants que nous devons régler sont la lenteur de l'algorithme des graphismes, et le fait que les images de l'arrière plan n'entrent pas correctement en collision.

2 Discrétisation du temps

Les versions précédentes de notre programme ne discrétisaient pas le temps, ce qui veut dire que les périodes de calcul (ou les ticks : les appels de notre fonction `update()`) s'exécutaient dès qu'elles le pouvaient. Cela implique que, lorsque l'on exécutait notre programme sur deux ordinateurs différents, l'un d'entre eux pouvait être beaucoup plus rapide que l'autre, et cela par plusieurs ordres de grandeur. Pour éviter cela, nous stockons le temps exact du dernier tick, puis nous attendons que la différence entre ce temps et le temps actuel est plus grand qu'une certaine constante k (en ms). Si nous voulons 60 images par seconde, alors il nous faut choisir $k = \frac{1000}{60}$.

3 Implémentation des mouvements

L'implémentation des mouvements se fait dans le fichier `evenements.py`. Celui-ci s'occupe de toutes les entrées de l'utilisateur. Avant, nous avions programmé que, si une flèche directionnelle (par exemple celle de droite) était pressée en front montant, alors notre vitesse horizontale était incrémentée, et ne diminuait pas. Cela faisait que notre personnage glissait sur le sol, et que le joueur pouvait appuyer autant de fois qu'il le voulait sur la flèche pour accélérer indéfiniment. Maintenant, la vitesse est augmentée seulement lorsque la flèche est maintenue. Nous avons aussi fait en sorte que la vitesse soit constamment multipliée par un nombre proche mais plus petit que 1, ce qui permet de freiner. Contre-intuitivement, nous n'avons pas besoin de mettre des bornes à la vitesse, puisque l'étape précédente le fait déjà pour nous (en effet, toute suite arithmético-géométrique $u_{n+1} = a \cdot u_n + b$ telle que $|a| < 1$ est bornée).

4 Implémentation des sous-titres

Pour le premier niveau, nous avons décidé de faire des sous-titres. Cette implémentation se fait dans le fichier `Dialogue.py` Celui-ci s'affiche en bas de l'écran lettre par lettre tout en faisant du son. Comme le nom du fichier l'indique, les sous-titres pourront aussi être utilisés comme des dialogues, même si nous ne nous sommes pas encore décidés sur si nous allons l'utiliser de cette manière.

5 Musiques

Nous avons déjà crée 6 musiques : `musique_basique`, `musique_decollage`, `musique_espace`, `musique_mort`, `musique_peur`, et `musique_souterrain`. Pour faire ces musiques, nous avons utilisé un éditeur MIDI en ligne appelé Signal.

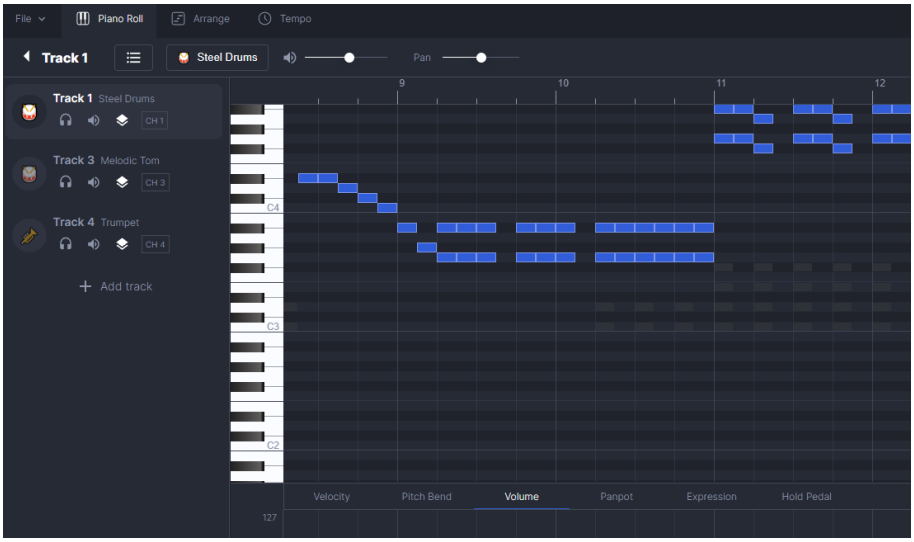

Figure 1: Représentation d'une partie de `musique_basique` par l'éditeur Signal

MIDI est un format entièrement numérique dans lequel on choisit quelles notes vont être activées à quel moment. Nous pouvons voir dans la Figure 1 que `musique_basique` contient trois pistes différentes qui vont être jouées en même temps, chacune avec des instruments différents. Puisque la première piste est sélectionnée, les autres sont grisées. Puisque le MIDI a plusieurs interprétations possibles (les instruments de musique par défaut varient selon les ordinateurs), nous les avons exporté au format .wav qui, même si il est beaucoup plus lourd en terme d'espace (car il stocke le son brut plutôt que les notes), a la qualité d'être lu de la même manière par tous les ordinateurs.

6 Création de sauvegardes

Pour sauvegarder, nous avons décidé de stocker les informations importantes en les stockant dans un dictionnaire `etatDuJeu`, puis d'écrire le résultat de `str(etatDuJeu)` dans un .txt se situant dans le dossier `sauvegardes`. Nous utilisions déjà ce dictionnaire pour pouvoir à la fois utiliser tous les objets importants et pour ne pas avoir de variables globales. Nous avons eu besoin de régler deux problèmes. Le premier est que certains objets ne renvoient pas toutes les informations en étant affichées, et affichent quelque chose ressemblant à `<__main__.A object at 0x0000010994D63220>`. Nous avons eu besoin de définir des méthodes `str()` à chaque objet. Le deuxième problème est que l'on doit retransformer le texte en un dictionnaire. Pour cela, nous avons utilisé la fonction `exec()`, qui prend en entrée une chaîne de caractères, et qui renvoie du code. Mais l'environnement de `exec()` est différent de celui de la fonction qui l'appelle. Heureusement, nous avons trouvé sur internet une manière de le récupérer (voir la sitographie).

Pour la gestion des fichiers, nous avons aussi eu besoin de faire des chemins relatifs. Il faut prendre en compte le fait que les séparateurs des chemins varient selon les systèmes d'exploitation. Cela est d'autant plus important par le fait que certains d'entre nous utilisent Windows, et d'autres Linux. Heureusement, la librairie `os` a une fonction qui renvoie ce séparateur.

7 Création de la barre de vie

Pour ce jeu, nous avons choisi de donner trois vies au joueur. L'implémentation de cette fonctionnalité a été faite dans le fichier `barreDeVie.py`. Il ne sera pas possible de regagner de la vie. Le joueur pourra seulement en perdre au contact d'un ennemi. Lorsque l'ennemi touchera le joueur, il perdra un point de vie, et il lui en restera donc 2 pour finir le niveau (s'il avait encore toutes ses vies). Lorsque le joueur a perdu toutes ses vies, une fenêtre s'ouvre avec un message lui demandant de choisir entre recommencer le niveau et quitter.

Nous avons choisi d'utiliser la la police d'écriture RubikIso-Regular, que nous avons importée dans le dossier principal de notre programme afin que, peu importe les polices de base stockées sur la machine de l'utilisateur, le programme puisse accéder à celle-ci. Nous aurons besoin d'importer d'autres polices par la suite pour créer des boutons, qui permettront au joueur de faire des choix.

Nous avons importé la police au format .ttf car celui-ci permet d'utiliser les polices au format vecteur et au format pixel. Le format vecteur est plus précis. Nous nous attendions à ce que le format vecteur soit plus lent à utiliser, avant de réaliser que la vitesse est identique.

8 Gérer les monstres et les générateurs

Au lieu d'utiliser le terme de Poubelles nous avons préféré utiliser Générateurs, rendant ainsi le terme plus général. Il y a donc eu la création de la clé `generateurs` dans le dictionnaire `etatDuJeu` dont l'attribut est un tableau de couple (x,y) de coordonnées de chaque générateur.

Équation permettant de définir la position des monstres selon la position du générateur :

$$\begin{cases} x = (\texttt{X\_generateur} + 1) + (\texttt{monstre\_width}/2) \\ y = \texttt{Y\_generateur} + (1.9 - \texttt{monstre\_height}) * 0.5 \end{cases}$$

`X_generateur` correspond à la coordonnée x du générateur
`monstre_width` correspond à la largeur du monstre
`Y_generateur` correspond à la coordonnée y du générateur
`monstre_height` correspond à la hauteur du monstre

Création d'un fichier `Generation.py`, qui gère la génération des monstres. Le fichier contient :

- L'objet `Generateur` :
  - Pour initialiser le Generateur (à la méthode `__init__`), on donne en paramètre de départ :
    - Les `coordonnes` du générateur.
    - Le type de monstre (`typeMonstre`) généré. En l'occurrence, il s'agit ici de balais et de spray (d'autres seront peut-être implantés)
    - Un nombre maximum de monstre généré (`nbGenereMax`)
    - Et enfin, l'intervalle de temps entre deux générations (`intervalleTemps`)
  - La procédure `genereMonstre` génère un type de monstre selon la position d'un générateur
  - La procédure `generation` vérifie si :
    - Le nombre de monstres générés par le générateur n'a pas dépassé le nombre maximal de générations possibles,

    - Le générateur ne vient pas de générer un monstre,
    - La différence entre l'instant où la fonction a été appelée et celui de sa dernière génération est bien égale à l'intervalle de temps du générateur

- La fonction `initialiserGenerateurs` permet la création des objets Générateurs en utilisant les coordonnées des générateurs du niveau, l'intervalle entre chaque génération de monstre et le nombre maximal de génération est aléatoire
- La fonction `regirGenerateurs` lance la méthode `generation` de l'objet `Generation.py` pour chaque générateur de la liste à l'aide d'une boucle.

9 Conclusion

Nous avons maintenant un jeu dans lequel le personnage peut se se déplacer, sauter, croiser des ennemis, et tomber dans le vide. Cela nous a permis de faire un premier niveau de plateformes, dans lequel le joueur doit essayer d'arriver au bout. Nous avons encore des tâches à accomplir. Celles-ci sont la progression du jeu, comme la transition entre les niveaux et la collecte des étoiles, ainsi qu'un menu pour pouvoir recommencer et sauvegarder proprement le niveau. Pour finir, nous voudrions que vous appréciez ce magnifique lever de soleil terrestre.

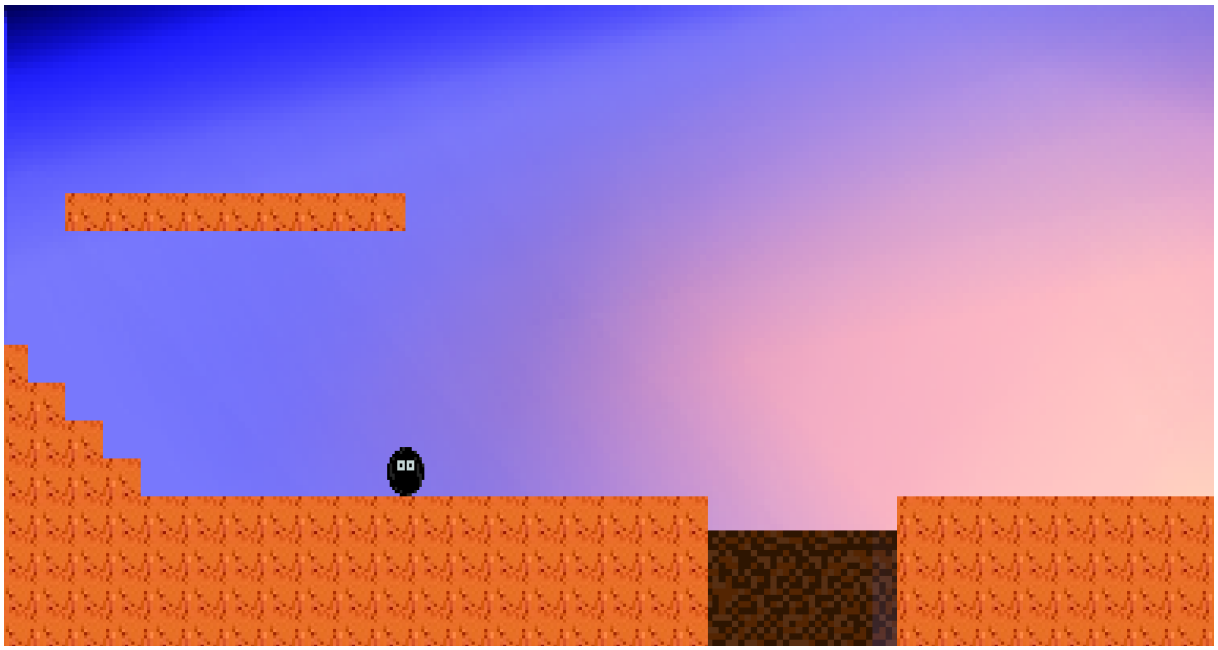

Figure 2: Kaku face à un lever de soleil terrestre

10 Sitographie

- Le site Signal permettant d'éditer le format MIDI
  https://signal.vercel.app/edit
- La page Wikipédia sur les suites arithmético-géométriques
  https://fr.wikipedia.org/wiki/Suite_arithm%C3%A9tico-g%C3%A9om%C3%A9trique
- Comment récupérer l'environnement de la fonction `exec()`
  https://stackoverflow.com/questions/1463306
- Comment faire des chemins relatifs
  https://towardsthecloud.com/get-relative-path-python
- Google Fonts qui nous a permi d'installer des polices d'écriture
  https://fonts.google.com