

Dustopia - Compte rendu 6

Anouk OMS, Elora ROGER, Lysandre TERRISSE

Rapport Final



Table des matières

1	Introduction	3
2	Le jeu	4
2.1	L’histoire	4
2.2	Principe/but	4
3	Côté graphisme	5
3.1	Outils numériques	5
3.2	Background	5
3.3	Personnages/objets	6
4	Côté code	7
4.1	La vie	7
4.2	Les entités	8
4.3	Génération de monstres	9
4.4	Collisions	10
4.5	Musiques	14
4.6	Dialogues et Animations	15
4.7	Niveaux bonus	17
4.8	Sauvegards	18
5	Conclusion	19
6	Sitographie	20

1 Introduction

Dans le cadre d'un projet de groupe à réaliser en cours, nous avons choisi de créer un jeu vidéo. Nous nous sommes alors inspirés du jeu vidéo *Mario* ainsi que de l'univers *Ghibli*

Au début de ce projet, nous avons décidé de programmer ce jeu à l'aide du langage C. Puis, face à la difficulté de la tâche, nous avons changé d'avis, et avons décidé de programmer en Python. Les avantages de Python sont que c'est un langage que nous connaissions déjà tous, qu'il s'occupe lui-même de l'allocation mémoire, et qu'il peut faire de la Programmation Orientée Objet. De plus, Python nous permet d'utiliser la bibliothèque *Pygame*, qui est spécialisée dans la création de jeux vidéos, et qui est très rapide par le fait qu'elle est basée sur la bibliothèque SDL.

Pour plus d'efficacité, nous avons décidé de nous répartir les tâches en suivant deux catégories : l'aspect graphique du jeu et l'aspect code.

Elora - Graphisme

- *Design de Kaku et des Susuwatari*
- *Création des arrière-plans*
- *Design des autres entités*, telles que les monstres (balais, sprays, aspirateurs), l'étoile, ...
- *Coder la barre de vie*

Lysandre - Code

- *Création de l'algorithme de collisions*
- *Création des animations et des dialogues*
- *Création des musiques*

Anouk - Code

- *Création des monstre*, création d'une entité Monstre.
- *Génération des monstres* création de générateur, prenant la forme d'une poubelle.

2 Le jeu

2.1 L'histoire

Cette histoire se déroule dans une dystopie. Une entreprise dirigée par les humains, dont le dirigeant est un certain Bob, se propage de plus en plus dans la galaxie. Cette entreprise ravage des civilisations entières et exploite la puissance des étoiles pour seul but de préserver sa propre existence.

Kaku, un susuwatari comme les autres, décide d'empêcher cette catastrophe morale. En passant par une planète dénommée Terre, cette petite boule de suie découvre que les humains emprisonnent et exterminent les susuwataris un par un pour récupérer leur Dust.

Ainsi, le but de notre petit susuwatari est de libérer ses amis tout en récupérant un maximum d'étoiles, puis de rentrer sur leur planète d'origine. Mais, il doit faire attention, puisque les humains ont créé des gardiens pour empêcher les susuwataris de s'enfuir.

2.2 Principe/but

Le joueur contrôle Kaku. Il peut se déplacer à gauche, à droite et sauter en utilisant les flèches directionnelles.

Un niveau est composé de plusieurs blocs sur lesquels Kaku doit marcher, sauter ou contourner pour atteindre la fin du niveau où il trouvera un autre Susuwatari. Mais s'il tombe dans un trou, il meurt, et le joueur doit recommencer le niveau.

Sur le chemin, il peut rencontrer des poubelles qui génèrent trois types de monstre : des balais, des sprays, ou encore des aspirateurs, que le joueur devra esquiver (en sautant par dessus par exemple). Si par malheur Kaku ne parvient à les éviter il perd une vie et meurt s'il a perdu ses trois vies. La barre de vie étant réinitialisée à chaque début de niveau.

3 Côté graphisme

3.1 Outils numériques

Les sites que nous avons utilisés pour les graphismes sont : PixilArt ainsi que GIMP. PixilArt pour la création des backgrounds personnages...et GIMP afin de les redimensionner.

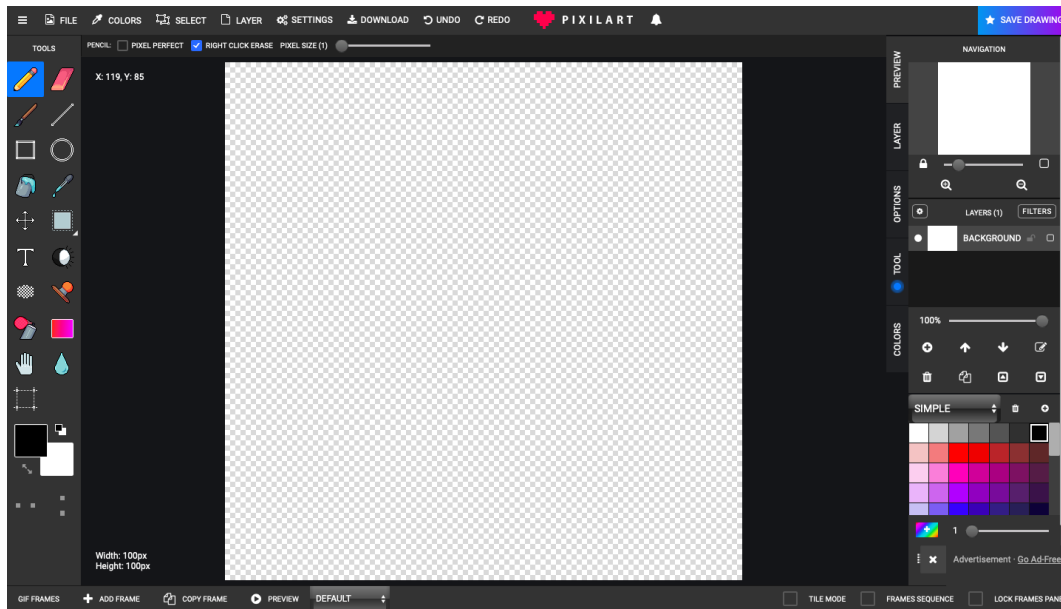


Figure 1: le site pixilArt

3.2 Background

Nous avons décidé que les niveaux de notre jeu refléteraient chacun un moment de la journée. Nous avons donc réalisé 7 différents fonds, de la nuit au coucher de soleil. Au fur et à mesure de la progression du jeu nous avons également décidé d'ajouter des niveaux bonus : dans le désert et dans la neige

Enfin, nous avons utilisé des images existant déjà sur internet (venant de la NASA et de PublicDomainPictures.net) pour les derniers niveaux ayant lieux dans l'espace.

3.3 Personnages/objets

Les ennemis de Kaku ont été représentés dans l'optique de nuire à ces boules de suie. Nous avons donc choisi de créer un générateur de monstres correspondant à une poubelle ainsi que des ennemis représentant un aspirateur, un spray et un balai.



Figure 2: Les trois type de monstres que le joueur peut rencontrer durant une partie

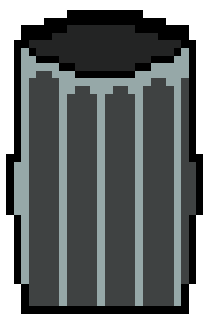


Figure 3: la poubelle, soit un générateur de monstre

4 Côté code

4.1 La vie

La vie... ne me parlez pas de la vie

Marvin, *Le Guide du voyageur galactique*

La classe `BarreVie` a comme attribut : un objet `vie` initialisé à trois, une variable correspondant au temps d’invincibilité, ainsi que deux objets initialisés à `False`, permettant respectivement de savoir si le joueur meurt, ou bien si il rentre en contact avec un ennemi.

La fonction `affichage` nous permet d’afficher le nombre de vies en haut à gauche de l’écran. La vie est représentée par un charbon, c’est ce qui sert de vitalité au susuwatari.



Figure 4: La barre de vie (entourée en gris pour la différencier de la page)

La fonction `GameOver` se déclenche lorsque Kaku n’a plus aucune vie, soit lorsque `self.estMort` passe à `True`. Ainsi un menu apparaît permettant au joueur de quitter ou bien de recommencer la partie.

4.2 Les entités

La classe **Entite** a comme attribut un nom, un objet **box** (utilisé pour gérer les collisions), une image et un dictionnaire **autresParametres**. Nous avons créé dans le fichier *main.py* un dictionnaire **etatDuJeu** contenant une liste **listeEntites** qui contient toutes les entités créées : *Joueur* (soit Kaku), *Susuwatari à sauver* (soit l'un des amis de Kaku), *Etoile*, *Bob* ainsi que les monstres. Nous avons fait en sorte que le joueur soit toujours le premier élément de **listeEntites**.

Afin pouvoir programmer le déplacement des monstres, il fallait filtrer entre les monstres et les autres entités. Il nous suffit donc de faire un test vérifiant si le nom de l'entité est dans **listeMonstre** du dictionnaire **etatDuJeu**.

Au début, Bob apparaissait au début de chaque niveau, puis nous avons décidé que Bob n'apparaîtrait qu'une fois par niveau. Ainsi si le joueur meurt durant le niveau, il pourra reprendre la partie sans que Bob ne revienne, rendant ainsi le jeu plus fluide.

4.3 Génération de monstres

Nous avons décidé d'implémenter trois types de monstres : balais, sprays, et aspirateurs. Aucun d'eux ne possède de capacités particulières, seule leur taille et leur design les différencient.

Chaque poubelle représente un générateur de monstre. À l'initialisation du jeu, deux nombres aléatoires définissent le temps (en seconde) entre chaque génération de monstre ainsi que le nombre de monstres générés. Le type de monstre généré est lui aussi aléatoire. Pour cela, nous avons utilisé la fonction `choice()` de la bibliothèque `random` qui est une fonction qui choisit aléatoirement un élément dans une liste passée en paramètre.

Nous avons dû définir un calcul permettant de définir la position des monstres selon la position du générateur :

$$\begin{cases} x = X_générateur + 1 \\ y = Y_générateur + monstre_height/2 \end{cases}$$

`X_générateur` et `Y_générateur` correspondent respectivement à la coordonnée `x` et `y` du générateur

`monstre_height` correspond à la hauteur du monstre

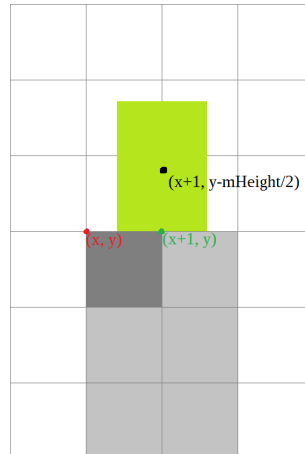


Figure 5: Le rectangle vert clair représente l'entité, et les blocs gris représentent la poubelle

4.4 Collisions

Non mais regardez mon programme comme il est parfait ! Franchement, on voudrait l'encadrer.

Pierre Pompidor, *HAI103I Utilisation des Systèmes Informatiques*

À cause des erreurs de flottants, l'algorithme de collisions, se trouvant dans `Box.py`, a été compliqué à coder. Nous nous étions mis d'accord dès le départ sur le fait de ne pas rajouter de constante ε pour résoudre cela, puisque ceux-ci ont tendance à rendre le jeu encore moins stable. À la place, nous avons décidé d'utiliser ce que nous appelons des alignements.

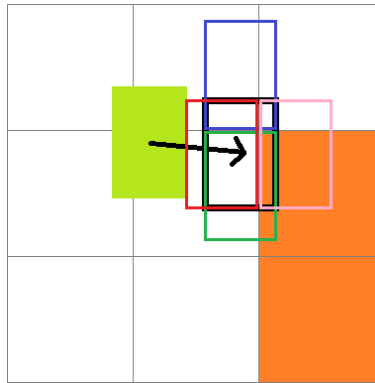


Figure 6: L'entité (vert clair), lorsqu'elle va bouger vers sa nouvelle position sans prendre en compte les collisions (noir), va entrer en collision avec un bloc (orange). Pour savoir où elle devrait se repositionner, nous devons générer quatre alignements. L'alignement par le haut (vert foncé) correspond à la position que devrait prendre l'entité si la collision venait sa la face du haut. Pareil pour l'alignement par le bas (bleu), par la gauche (rose), et par la droite (rouge).

Maintenant que l'on a des alignements, comment devons-nous les utiliser ? Une des réponses simples, mais erronées, est de dire que nous devons choisir l'alignement le plus proche de la position actuelle du joueur (donc le rectangle rouge dans ce cas, puisqu'il est le plus proche du rectangle vert clair). Mais considérez ce scénario.

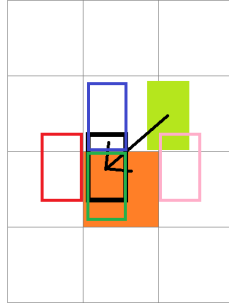


Figure 7: Ici, il est évident que l'entité devrait aller au niveau du rectangle bleu. Cependant, le rectangle rose est plus proche du rectangle vert clair.

Il faut donc ajouter une nouvelle étape dans cet algorithme. Quel est le problème dans le scénario ci-dessus ? Pour répondre à cela, il faut retourner au jeu qui nous a inspiré.



Figure 8: Dans Super Mario 64, lorsque Mario essaie de passer d'un point A à un point B, le jeu décide de vérifier pendant quatre *quartersteps* que Mario ne cogne aucun mur. Si un mur est touché, le mouvement est annulé.

Par la suite, nous dirons qu'un point est *valide* lorsqu'aucune de ses quartersteps n'annule le mouvement. Maintenant, nous pouvons voir quel était le problème de notre exemple précédent. Le rectangle rose est une position invalide pour le rectangle vert clair. Il faut ainsi privilégier les positions valides par rapport aux positions invalides. Ainsi, si il y a des alignements valides, alors nous choisissons le plus proche d'entre eux. Sinon, nous choisissons le plus proche des alignements invalides.

Les cas où il n'y a aucun alignement valide sont les cas où nous de-

vons répéter plusieurs fois l'algorithme. Plus précisément, il faut répéter l'algorithme en conservant la même position de départ (le même rectangle vert), mais en ayant notre vecteur de vitesse pointant vers l'alignement invalide le plus proche calculé précédemment.

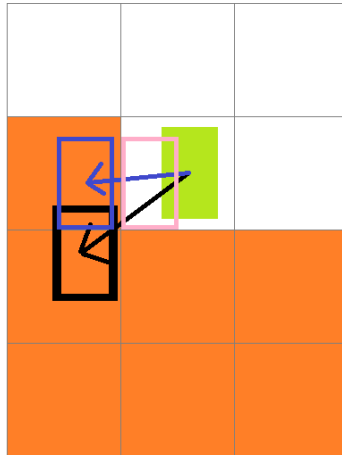


Figure 9: Ici, aucun des alignements n'est valide. Nous répétons donc le processus sur le rectangle bleu, puisqu'il est l'alignement le plus proche. Cette nouvelle exécution de l'algorithme nous dit d'aller au rectangle rose.

Cependant, en répétant l'algorithme, il ne faut pas aligner selon la même direction. Considérons ce scénario.

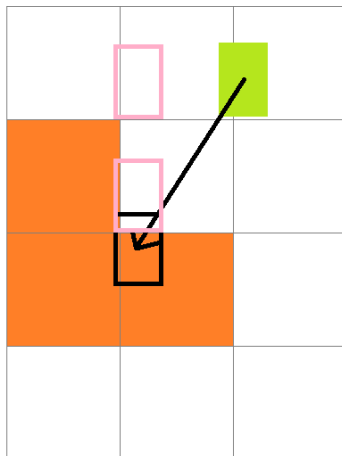


Figure 10: Si nous alignons deux fois selon la même face (bas), l'entité peut se retrouver à la mauvaise position. Cela s'explique par le fait que, lorsque la face du bas est alignée, nous nous trouvons déjà à la bonne ordonnée.

Pour cela, nous conservons une liste `alignementsPrecedents`, qui contient quels alignements ont déjà été faits ("Haut", "Bas", "Gauche", ou "Droite").

Cela est tout pour l'algorithme des collisions. Cependant, cet algorithme nécessite de vérifier si une boîte entre en contact avec un bloc. Pour déterminer cela, nous ne pouvons pas seulement vérifier les quatres coins de la boîte.

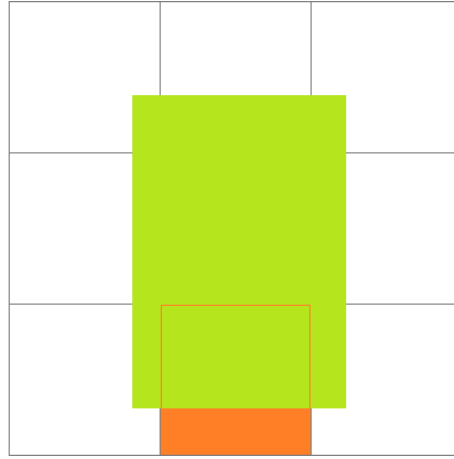


Figure 11: Si l'entité fait plus d'un bloc, regarder les coins n'est pas suffisant pour déterminer si l'entité touche un bloc.

À la place, si un côté a une longueur de n blocs, nous devons y générer $\lfloor n \rfloor$ nouveaux points.

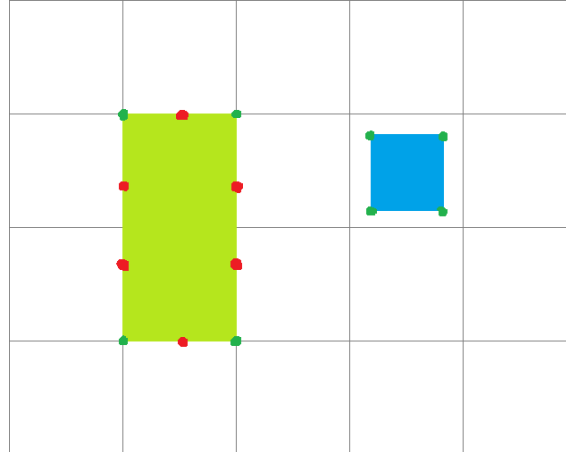


Figure 12: Pour l'entité de gauche, nous devons générer deux points pour chaque face verticale et un point pour chaque face horizontale. Tandis que pour l'entité de droite, nous n'avons pas besoin de générer de nouveaux points, puisqu'elle fait moins d'un bloc. Cela correspond bien à prendre la valeur au plancher de chaque côté.

4.5 Musiques

Ce n'est pas du Shakespeare, mais c'est authentique.

Cristof, *The Truman Show*

Pour composer les musiques, nous avons utilisé le format MIDI. Cependant, puisque celui-ci est interprété différemment selon les ordinateurs, nous les avons exportés au format .wav qui, même si il est plus lourd en terme de mémoire, nous permet d'avoir le même audio de partout.

Pour manipuler le format MIDI, nous avons utilisé un éditeur en ligne appelé Signal (voir sitographie).

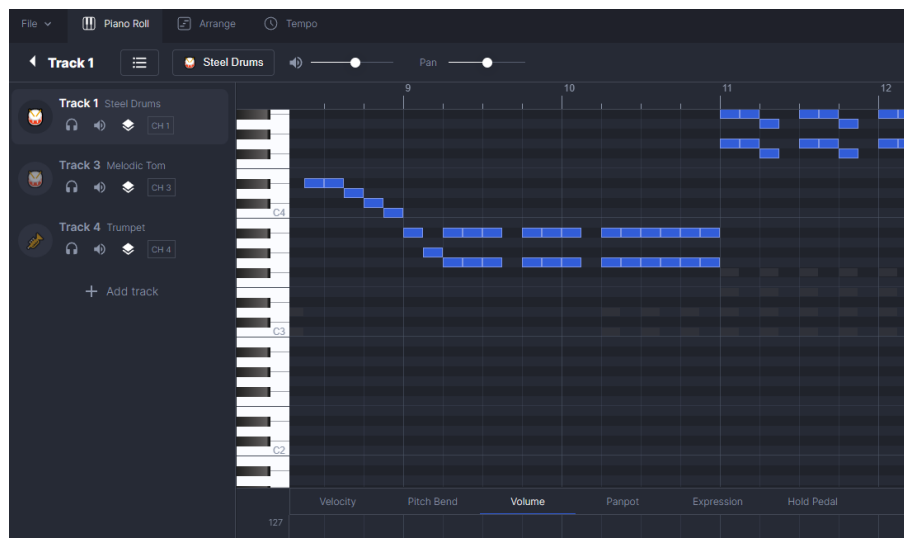


Figure 13: Représentation d'une partie de musique_basique par l'éditeur Signal

Nous avons créé six musiques : `musique_basique`, `musique_bonus`, `musique_evacuation`, `musique_evacuation_espace`, `musique_espace`, et `musique_mort`.

4.6 Dialogues et Animations

La voie du dialogue est une voie joyeuse pour soi et pour les autres ; c'est la voie du bonheur.

Daisaku Ikeda, *Dialogues avec la jeunesse*

Pour introduire le jeu dans son histoire, nous avons créé un système de dialogues. Lorsqu'un dialogue doit s'afficher, celui-ci s'affiche lettre par lettre à chaque image.

Les polices d'écriture ont été téléchargées grâce à Google Fonts, à l'exception de `Bob.ttf` qui a été téléchargée sur wFonts, et de `Boba Cups.ttf` sur DaFont. Nous avons choisi le format `.ttf` puisque, premièrement, celui-ci est le plus courant, mais aussi car il a une représentation en format vecteur, ce qui rend la police plus précise et moins pixelisée.



Figure 14: `Bob.ttf` est utilisée par Bob




Figure 15: `Kanit-Regular.ttf` est utilisée pour les sous-titres

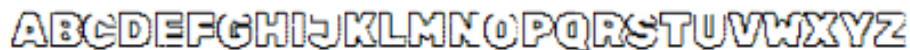


Figure 16: `RubikIso-Regular.ttf` est utilisée pour le menu de mort



Figure 17: `Boba Cups.ttf` est utilisée pour le menu principal

Nous avons deux animations, qui donnent une meilleure sensation de progression au jeu. Celles-ci sont les animations de Bob et du susuwatari à sauver, qui se font respectivement au début et à la fin de chaque niveau.



Figure 18: L'animation de Bob est un monologue. Au début du niveau, Bob tombe du ciel et pointe du doigt Kaku. Il lui fait des reproches, le nargue, ou le menace. Puis, il remonte grâce aux cordes situées dans son dos.

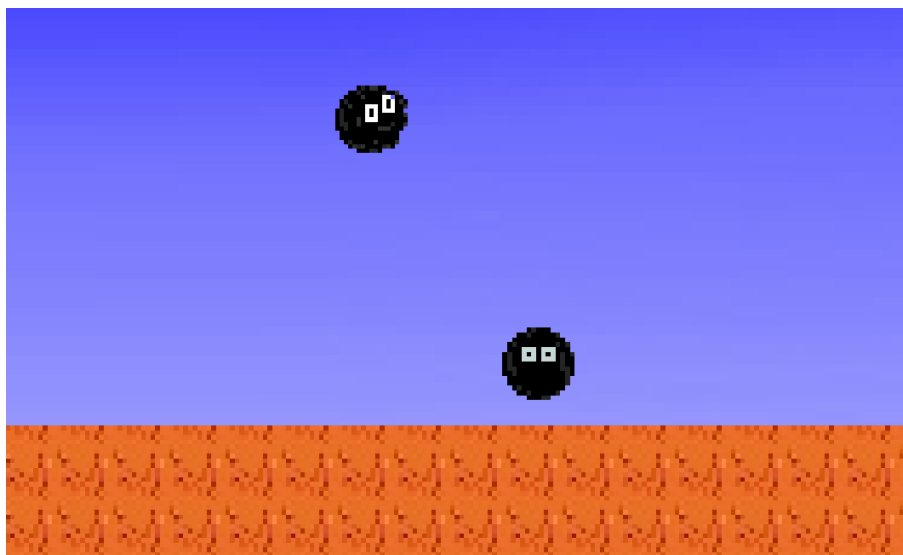


Figure 19: Pendant l'animation du susuwatari à sauver, Kaku sautille avec son ami après l'avoir libéré. Ensuite un fond noir apparaît. Puis, le niveau suivant est chargé.

4.7 Niveaux bonus

Pour que le joueur ait intérêt à récupérer les étoiles éparpillées dans les niveaux, nous avons ajouté deux niveaux bonus. Le premier s'obtient en récupérant les six premières étoiles, tandis que le deuxième s'obtient en récupérant l'étoile du premier niveau bonus.

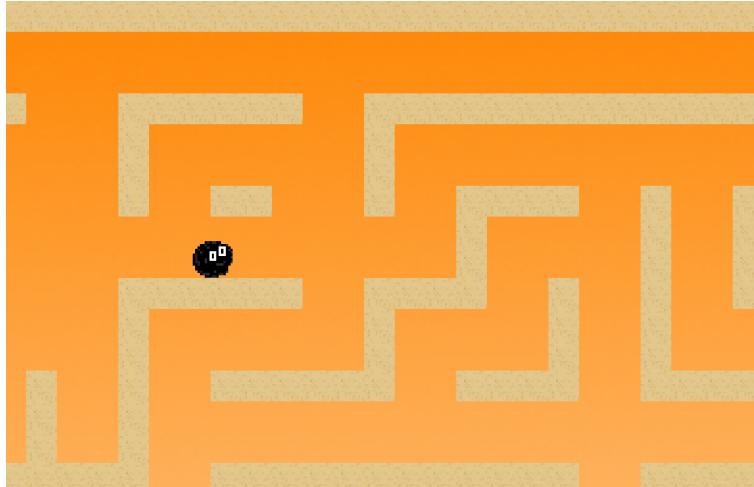


Figure 20: Le premier niveau bonus se déroule dans un désert. Êtes-vous bloqués dans le labyrinthe ? Ce n'est pas grave, nous allons vous donner un indice : Pensez en dehors de la boîte.

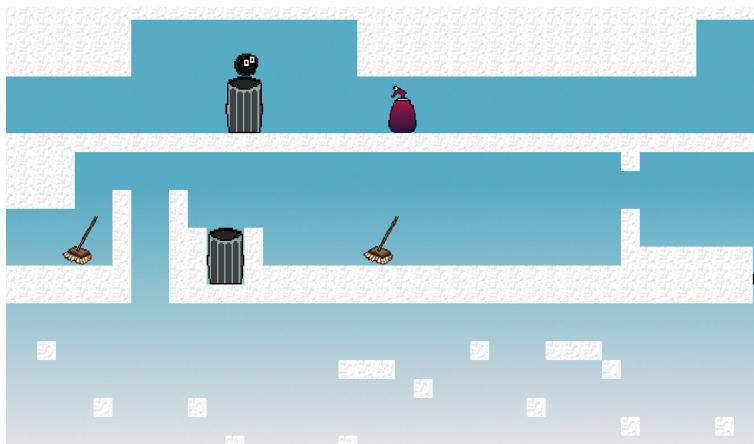


Figure 21: Le deuxième niveau bonus est enneigé. Puisque celui-ci est le dernier défi du jeu, il est plus complexe que les autres. Si vous n'y arrivez pas et voulez passer au niveau suivant, la prochaine section vous sera utile.

4.8 Sauvegardes

[Y]ou're just a dirty hacker, aren't you?

Sans, *Undertale*

Pour que le joueur puisse sauvegarder sa progression, et pour faciliter le débogage du jeu, nous avons implémenté un système de sauvegardes dans `sauvegarder.py`.

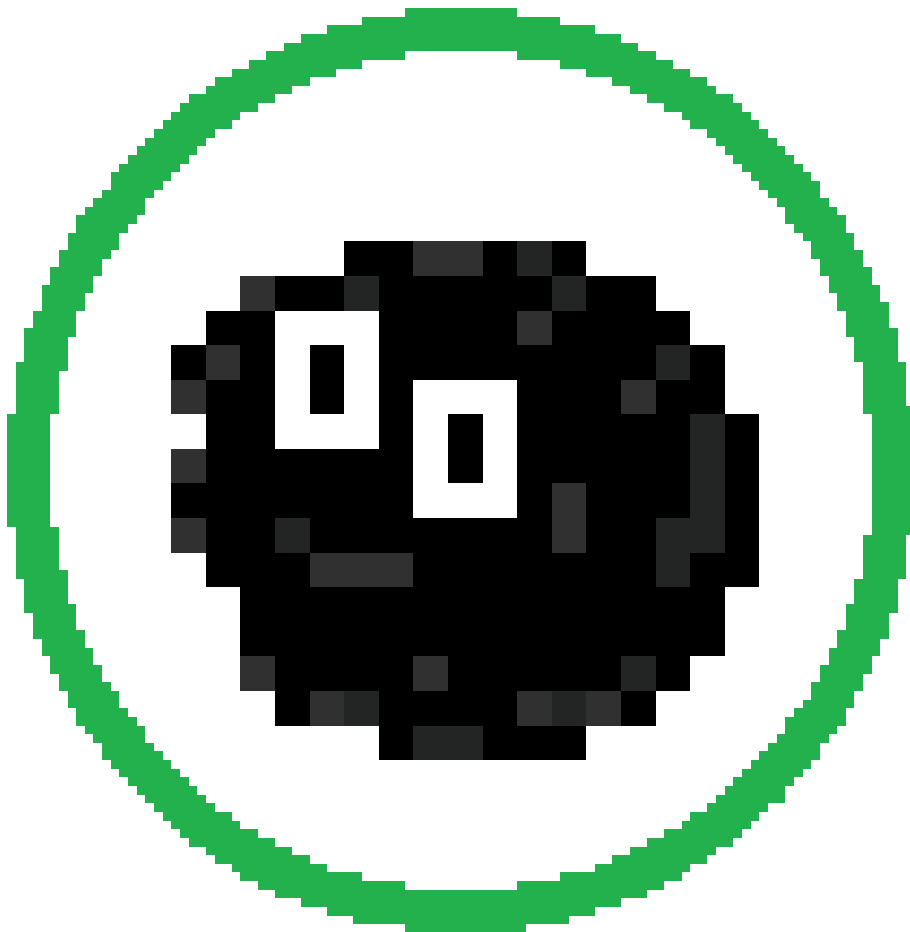
Ainsi, le joueur peut appuyer sur la touche S pour sauvegarder une progression, et sur la touche L pour charger la dernière sauvegarde. Cela sauvegarde le niveau actuel et le nombre d'étoiles récupérées. Cette sauvegarde est faite dans un fichier `sauvegarde_1.txt` se trouvant dans un fichier `sauvegardes`. Si vous êtes bloqués dans le niveau enneigé (par pur hasard), vous pouvez ouvrir ce fichier et incrémenter sa première valeur de 1 (la deuxième valeur étant le nombre d'étoiles).

5 Conclusion

Ce projet a été vraiment très intéressant à faire. Il nous a permis de travailler en équipe pendant une durée d'environ trois mois, ce que nous n'avions jamais fait auparavant. En plus de cela, nous avons à l'arrivée un résultat concret. Même si certaines parties ont été très compliquées (notamment l'algorithme de collisions), nous avons fini par avoir un jeu qui, nous l'espérons, sera agréable à jouer. Pour finir, une dernière citation :

Ah ! Et au cas où on ne se reverrait pas,
une bonne soirée et une excellente nuit !

Truman, *The Truman Show*



6 Sitographie

- Vidéos de Formation Vidéo sur l'utilisation de Pygame
<https://www.youtube.com/playlist?list=PLrSOXFDHBtfHg8fWBd7sKPxEmahwyVBkC>
- Site officiel de Pygame
<https://www.pygame.org>
- Le site Signal permettant d'éditer le format MIDI
<https://signal.vercel.app/edit>
- 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>
- Comment utiliser des surfaces transparentes en Pygame
<https://stackoverflow.com/questions/6339057>
- Comment obtenir les dimensions d'un texte en Pygame
<https://stackoverflow.com/questions/25149892>
- Image d'une fausse nébuleuse pour l'un des arrière-plans
<https://www.publicdomainpictures.net/fr/view-image.php?image=488420>
- Image d'une vraie nébuleuse prise par la NASA pour l'un des arrière-plans
<https://twitter.com/NASA/status/1546884608926646273>
- La police Kanit Regular sur Google Fonts
<https://fonts.google.com/specimen/Kanit>
- La police Rubik Iso sur Google Fonts
<https://fonts.google.com/specimen/Rubik+Iso>
- La police Bob sur wFonts.com
<https://www.wfonts.com/font/bob>
<https://www.wfonts.com/font/bob-filled>

- La police Boba Cups sur DaFont.com
<https://www.dafont.com/fr/search.php?q=boba+cups>
- Site de PixilArt :
<https://www.pixilart.com/>

Aucun contenu de ce jeu n'a été généré par une Intelligence Artificielle