Dustopia - Compte rendu 3

Anouk OMS, Lysandre TERRISSE, Elora ROGET, Thomas DAILLY

February 18, 2023

Introduction - Changement de langage :

Lors de la dernière séance de projet, nous avons décidé de changer de langage de programmation après avoir compris que nous ne pouvions pas manipuler d'objets en C, et que l'utilisation de la mémoire n'aurait pas été optimale.

Nous avons donc décidé de changer de langage de programmation et d'utiliser ***Python*** ainsi que ***Pygame*** qui remplacera CSFML pour l'aspect graphique du jeu.

Nous avons donc été obligés de revoir certaines répartitions des tâches car certaines fonctions que nous devions coder sont déjà proposées par cette bibliothèque, notamment celles concernant les graphismes et le son.

Nous n'avons pas pris de retard par rapport à nos objectifs de la semaine. Les premiers algorithmes ont été commencés, et certains ont été revus, comme l'algorithme de collision qui s'est révélé fallacieux.

Changement du planning :

Nous avons revu le planning de Lysandre suite à ce changement de langage (*cf Figure 1*):

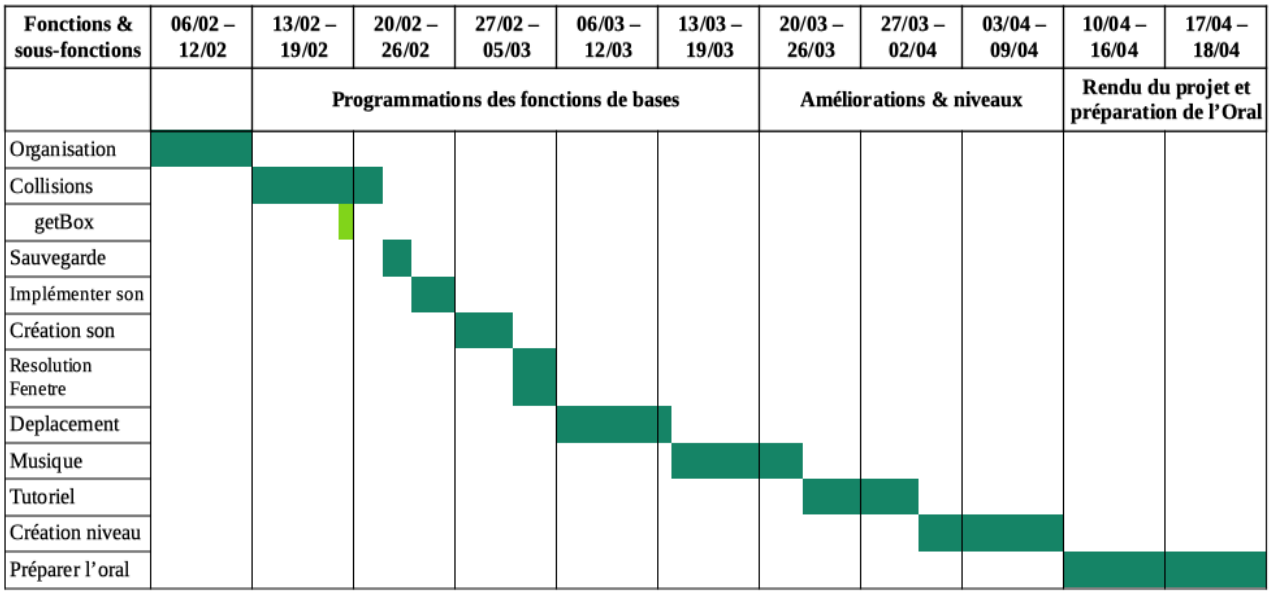

Figure 1 : Modification du planning de Lysandre

- *Tutoriel* : Ce niveau sera un niveau 0 disponible tout au début du jeu. Celui-ci permettra au joueur de découvrir les différentes commandes et fonctionnalités du jeu, tout en étant guidé par des sous-titres. Ce niveau nous permettra aussi de faire des tests: il contiendra ainsi quelques obstacles et ennemis faciles à éviter.
- La création des formats *PixelArt* et *ObjectArt* est désormais inutile grâce à Pygame.
- La dernière fois, nous avions oublié comme tâche de s'occuper de la **résolution de la fenêtre**. Nous expliquons nos idées quant à cette fonctionnalité ci-dessous.

Les fonctions et algorithmes :

L'objet `Personnage`

Le langage Python nous permet de faire de la programmation orientée objet. Ainsi nous avons décidé de faire un objet `Personnage` qui sera utilisé pour les monstres et pour *Kaku* (le joueur), car la plupart des fonctions qu'ils utilisent sont semblables, telle que celle du déplacement. L'objet `Personnage` de Python contiendra ainsi les méthodes suivantes :

- `collision` (paramètres : coordonnées de Box, vitesse) : calcule la prochaine position du personnage selon sa vitesse de déplacement du personnage
- `getBox` : définir les dimensions des rectangles dans lesquels vont se déplacer les personnages (*Kaku* et les monstres), leurs dimensions ...
- `déplacement` : appelant la méthode `collision` pour détecter les obstacles, cette méthode permet le déplacement d'un personnage. Selon la nature de ce personnage, la méthode change :
  - *Kaku, le joueur* : plus le joueur maintient une touche appuyée, plus sa vitesse augmente, impactant les autres méthodes liées aux déplacements telles que la `chute`, le `saut`, ...
  - *monstres* : le déplacement sera horizontal. Le monstre se déplace sans changer de vitesse et ne change de direction que lorsqu'il rencontre un obstacle, auquel cas il fait demi-tour. S'il touche le joueur, alors une fonction `attaque` sera lancée

Autres fonctions nouvellement définies

- `Sauvegarde` : Sauvegarde la progression du joueur et des contrôles personnalisés
- `ResolutionFenetre` : Permet d'agrandir et de rétrécir la fenêtre sans avoir d'impacts négatifs sur le jeu ni modifier la taille des éléments du jeu (Par exemple, rétrécir la fenêtre ne devrait pas écraser l'image, ce qui nous arrivait avec CSFML). Il faut que, lorsque la fenêtre sera agrandie par le joueur, celle-ci doit afficher plus de terrain.

Avancée du code

Depuis la dernière fois, nous avons avancé sur la programmation de la fonction `collision`, et nous avons réalisé que notre concept ne pouvait pas marcher. Nous avions dit que nous devions trouver un point étant sur la droite définie par la position centrale du joueur et son vecteur de vitesse. Mais en programmant, nous avons réalisé que cela n'était pas ce que nous voulions, car cela faisait un effet de ralenti à chaque fois que le joueur retombait sur le sol.

À la place, nous avons découvert une technique plus simple: Il suffit juste de déterminer quelles faces de l'entité entrent en contact avec un bloc, puis de faire une projection axiale dans ces directions (ce qui est trivial car toutes les droites sont verticales ou horizontales). Ensuite, il suffit juste de faire reculer le joueur de la moitié de sa longueur ou largeur.

Par exemple, si nous détectons que la face droite du joueur touche un bloc, il faut conserver la même ordonnée, puis prendre comme abscisse le plancher de l'ancienne abscisse, puis enlever la moitié de sa largeur. Puisque l'on a aligné notre personnage à droite, nous devons mettre sa vitesse horizontale à 0, et conserver la même vitesse verticale. Une fois cela fait, il faut ajouter à notre nouvelle position ce vecteur de vitesse.

Nous avions dit que pour déterminer si une entité entre en collision avec un bloc, il suffit de regarder chacun de ces coins. Premièrement, cela est faux lorsque l'entité est plus longue ou large qu'un bloc. Mais deuxièmement, cela nous est désormais inutile, puisque nous devons maintenant déterminer si **une face** entre en collision avec un bloc. Pour déterminer cela, il faut générer, entre les deux extrémités a et b de cette face, $\lfloor distance(a, b) \rfloor$ points. Ces points seront séparés au maximum d'un bloc, permettant, en calculant les positions où seront ces points, de n'oublier aucun bloc se trouvant strictement entre ces coins.

Cela dit, il reste quand-même un problème. Dans un monde parfait, lorsque le joueur est aligné par rapport à sa face droite, cette face ne touche plus de blocs. Mais puisque nous travaillons avec des flottants, cela devient faux (idem pour la face haute. Par contre, cela reste vrai pour la face gauche et basse). Pour résoudre cela, la plupart des gens rajouteraient une constante `epsilon`. Mais nous avons refusé cette option, puisque cela rendrait notre programme moins stable. À la place, lorsque l'on aligne notre entité à une face, nous faisons en sorte que notre fonction calculant si une face touche un bloc renvoie False, même dans le cas où ses points touchent réellement ce bloc. Cela permet d'être bien plus précis. Les seules autres solutions que nous connaissons sont de reconstruire les réels à l'aide des suites de Cauchy ou des coupures de Dedekind, mais cela prendra trop de temps à apprendre et à programmer.

Sitographie

- Vidéos de FormationVidéo sur l'utilisation de Pygame :
  https://www.youtube.com/watch?v=gx4yVcJqBaI&list=PLrSOXFDHBtfHg8fWBd7sKPxEmahwyVBkC&index=34
- Site officiel de Pygame :
  https://www.pygame.org
- Les constructions des réels pour éviter les erreurs de flottants
  https://en.wikipedia.org/wiki/Cauchy_sequence
  https://en.wikipedia.org/wiki/Dedekind_cut