Documentation

Goby Roads

Description du jeu

Le projet est une réinterprétation du jeu Crossy road en vue 2D isométrique.

Il se compose d'un écran de menu, d'un écran d'introduction, d'un écran de jeu, et d'un écran de game over.

L'écran de menu permet de choisir des options pour la partie, quant à celui d'introduction il permet d'en lancer une.

Le jeu consiste en un quadrillage composé de lignes de différents types : routes, rails, rivières, herbe et éventuellement glace. Chacune de ces lignes comporte des obstacles différents. Un personnage sur une des cases du quadrillage représente le joueur. L'objectif du jeu est de traverser un maximum de ligne sans se faire toucher par un obstacle. Pour cela, le joueur peut se déplacer selon 4 directions, via les touches ZQSD. Lorsque le joueur franchit une ligne, son score augmente de 1. A ce moment la grille défile vers le bas, la dernière ligne est supprimée et une nouvelle est créée en haut, à condition que le joueur soit assez haut (y du joueur égale à 3).

Les lignes d'herbe contiennent des arbres placés aléatoirement. Il est impossible de se déplacer sur les cases contenant des arbres. Ces lignes ne peuvent pas faire perdre le joueur, leur présence est donc de plus en plus rare à mesure que le score du joueur augmente.

Les lignes de routes contiennent des voitures de vitesse et de longueur aléatoire. Les voitures se déplacent de cases en cases en suivant leur ligne. Les voitures d'une ligne peuvent se déplacer vers la droite ou la gauche. Toucher une voiture fait perdre la partie.

De la même manière, les lignes de rivières contiennent des radeaux de vitesse et de longueur aléatoire, qui se déplacent comme les voitures. Un joueur sur une case de radeau est déplacé avec celui-ci. Toucher une rivière sans radeaux, ou être sorti de l'écran par le déplacement d'un radeau fait perdre la partie.

La vitesse moyenne de ces véhicules (voitures et radeaux) augmente avec le score du joueur.

Les lignes de rails cyclent entre trois état : normal, warning et passage de train. Toucher des rails pendant le passage d'un train fait perdre la partie.

Enfin, les lignes de glace font glisser le joueur dans la direction de son déplacement précédent, après un léger délai. Cela force le joueur à anticiper le déplacement des véhicules.

L’écran de game over affiche le score final de la partie, et donne la possibilité de lancer une nouvelle partie ou de quitter le jeu.

Aspects techniques

Logique du jeu

Le jeu fonctionne via une boucle while principale, qui actualise le jeu 60 fois par seconde via les fonctions de compte de ticks de SDL.

À chaque frame, on commence par vérifier les actions du joueur (c'est à dire les touches pressées) via la fonction SDL_PollEvent, puis on applique ces actions : déplacement, sortie du jeu si on se trouve dans le jeu, ou démarrer, redémarrer le jeu ou changer les options si on se trouve dans un menu.

Si on se trouve dans le jeu, on va alors actualiser l'état du jeu (scrolling, déplacement des véhicules, …). Cet état est stocké dans une structure de données appelée Gamestate, qui contient l'intégralité des informations sur la partie en cours. On affiche ensuite le jeu en utilisant la bibliothèque SDL_image.

Si on se trouve dans un menu, on actualise l'affichage, qui est principalement composé de texte, via la bibliothèque SDL_ttf.

Structures de données

Le Gamestate contient les informations relatives à une partie :

-le score

-la position du joueur

-la liste chainée des véhicules, qui est utilisée pour les déplacer l'un après l'autre. Les véhicules contiennent leur position et leur taille.

-la grille actuelle

La grille contient elle-même un tableau à double entrées, contenant le type de chaque case. Ce tableau est utilisé pour actualiser l'écran de jeu, et pour vérifier les collisions entre le joueur et les obstacles.

La grille contient également un rowManager, un "gestionnaire de ligne", qui contient les informations spécifiques à une ligne, pour éviter qu'elles ne soient dupliquées sur toutes les cases de la ligne. Ces informations comprennent le sens de déplacement des véhicules, et leur délai de déplacement (ou le délai de passage pour les trains).

Implémentation SDL

La boucle de jeu dans la version SDL ressemble beaucoup à celle de la version terminal. Cependant, elle va y ajouter certains concepts. En effet, ici nous utiliserons une multitude de librairies externes que nous initialiserons, nous devons aussi prendre en compte de l'état des différents éléments interactifs comme la lecture de la musique de fond mais aussi décider de quel écran afficher.

L'ajout de la surcouche graphique SDL2 se fait en encapsulant certains concepts de la version core avec notamment le fichier "core_wrapper". Celui-ci permet de définir un UIGameState qui va lui même contenir le GameState précédemment mentionné mais en y ajoutant des propriétés propres à SDL : playerOffset pour l'affichage du joueur, menuHandler qui permet de suivre l'évolution de l'écran de menu et enfin les entiers running, intro et menu qui permettent de savoir quel écran afficher.

IA

Algorithme utilisé :

Dans un compromis entre efficacité, complexité d'implémentation et temps de calcul, nous avons choisi d'implémenter l'algorithme A* pour créer une IA jouant au jeu. Cet algorithme de pathfinding utilise à la fois la distance du chemin parcouru et une fonction heuristique pour estimer le coût restant jusqu'à la fin. Cela permet à l'IA de prédire ses déplacements de manière efficace, sans calculer tous les chemins comme Dijkstra, tout en prenant en compte les obstacles présents sur la carte.

Voici les étapes de l'algorithme :

1) On crée une liste avec les positions à explorer. Au début, elle contient seulement le point de départ
2) On garde aussi une liste des positions déjà explorées, pour éviter de repasser au même endroit
3) À chaque étape, on choisit dans la liste ouverte la position qui paraît la plus prometteuse, par rapport à son coût f (déjà parcouru + estimation)
4) Depuis la position choisie, on regarde les cases voisines qui ne sont pas des obstacles.
5) Pour chaque voisin, s'il n'est pas dans la liste ouverte on l'ajoute et on définit son parent, sinon si le chemin pour y arriver et plus petit qu'avant, on l'actualise et on met à jour son parent
6) On fait cela jusqu'au point d'arrivée ou qu'il n'y ait plus de noeuds à explorer
7) On construit le chemin en remontant tous les parents du noeud d'arrivée jusqu'au départ

Structure de données :

La structure Noeud représente une case de la grille ainsi que les caractéristiques qui y sont associées et qui seront utiles à l'algorithme.
On a donc

- les coordonnées de la case
- g : coût depuis le début jusqu'à ce noeud
- h : coût heuristique estimé (distance de Manhattan car déplacements haut-bas-gauche-droite)
- f : coût total (g + h)
- le noeud parent/précédent dans le chemin

La structure priorityQueue qui est une structure de tas minimum. Cela permet de récupérer efficacement le nœud qui possède le plus petit f. Cette fonction est appelée à chaque nœud analysé et permet donc d'économiser de nombreuses recherches de minimum dans une liste.

Gestion des voitures :

On cherche à savoir si une voiture sera présente au moment où on explore la case. Pour cela on a une formule calculant le nombre de cases que la voiture aura parcourue avant que le joueur n'y arrive.
Prenons ce nombre de case n, si une voiture allant vers la droite est présente à la position du joueur - n alors elle y sera en même temps que lui est ce sera un obstacle.
Pareil pour une voiture allant vers la gauche et position + n.

Intégration dans le jeu :

Fonction qui permet d'appeler l'algorithme A* avec la position du joueur comme départ et le point le plus éloigné au centre de la grille qui n'est pas un obstacle en tant qu'arrivée.
Cela renvoie une liste de nœuds correspondant au chemin ordonné. Chaque nœud va être passé à une fonction qui va faire déplacer le joueur vers ce nœud.
Quand tous les noeuds ont été passé, on réitère l'algorithme.