

Projet de Programmation Avancée

Pac-Man « Puckman » (Namco, 1980)

Ce rapport décrit mes contributions au projet. La majorité de mon travail s'est orientée autour du niveau en lui-même, c'est-à-dire des classes `Cell` et `Terrain`. Une attention particulière a été accordée à la performance et à l'aisance de sa personnalisation.

Chargement du niveau

L'espace de jeu est décrit par un fichier lu à l'exécution (figure 1). Chaque caractère correspond à un type de cellule, ou pour les cellules vides, à l'objet qu'elles contiennent éventuellement.

Affichage du niveau

Le niveau affiché suit celui décrit dans le fichier, et non une surface monolithique. Il serait donc théoriquement possible de charger un niveau personnalisé.

De ce fait, on détermine l'aspect d'un mur en fonction de ses voisins. Cela est fait par l'utilisation d'un masque sur 4 bits déterminé trivialement à partir duquel on peut indexer dans un `std::array` pour récupérer le bon *sprite* pour la cellule isolée. On construit l'entière du niveau de cette manière. Le calcul du masque se fait de toute

manière uniquement au chargement du niveau, mais pour éviter complètement de redessiner les murs, qui ne changent jamais, on les dessine à la place une unique fois sur une surface cache, qui est copiée sur la surface rendue à l'écran à chaque *frame*.

Comme le *sprite* d'un mur est déterminé localement, on ne peut pas automatiquement reproduire l'aspect de certains murs, qui formant une boucle, ne sont pas visuellement connectés à leurs voisins du dessus ou du dessous. Pour remédier à cela, j'ai ajouté quatre nouveaux caractères qui déterminent qu'un mur ne peut se connecter que dans un sens, les deux verticaux (^ et V) étant utilisés, mais pas les deux horizontaux (< et >) qui sont tout de même utilisables.

```
#####
#0000000000#0000000000#
#0####0####0#0####0####0#
#G#.#0#.#0#0#.#0#.#G#
#0####0####0#0####0####0#
#000000000000000000000#
#0####0#0#^ ^ ^#0#0####0#
#00000#0#VVV#0#00000#
#####0#000#000#0####
. . . #0####.#.####0#. . .
. . . #0# . . . . . #0# . . .
#####0# . ## - ##.#0#####
. . . . . 0 . . # . . . 0 . . .
#####0# . #####.#0#####
. . . . #0# . . . . . #0# . . .
. . . . #0#.#^ ^ ^#.#0# . . .
#####0#0#VVV#0#0####
#0000000000#0000000000#
#0####0####0#0####0####0#
#Goo#00000000000000#ooG#
#^#0#0#0#^ ^ ^#0#0#0#^#
#V#0#0#0#VVV#0#0#0#V#
#00000#000#000#00000#
#0#####0#0#####0#
#00000000000000000000#
#####
```

Figure 1 : fichier de niveau

L'implémentation se base sur un masque négatif dans la méthode `Cell::set_wall`, appelée une première fois à la lecture du caractère représentant la cellule, où le masque négatif est donné en second argument, puis une seconde fois une fois les murs voisins déterminés, avec une valeur balise par défaut de -1.

Chaque cellule stocke une structure de donnée légère pointant vers son *sprite*. Le cas des murs a été évoqué plus haut, mais pour les autres types de cellules, au lieu de passer par un `switch`, `Cell::update_sprite_handle`, en outre du tableau des *sprites* des murs, reçoit des `std::unordered_map` de `Terrain::update_sprite_handles`, qui simplifient grandement la logique. L'objet, ou « fruit » de chaque niveau ainsi que la valeur de chacun sont aussi gérés de ces manières à des fins de lisibilité, à un léger coût en performance.

Enfin, l'échelle du niveau et des entités (nourriture, personnages) est contrôlable (figure 2) et la tinte de la surface de cache peut être changée pour effectuer le clignotement en blanc du jeu original sans passer par des sprites supplémentaires.

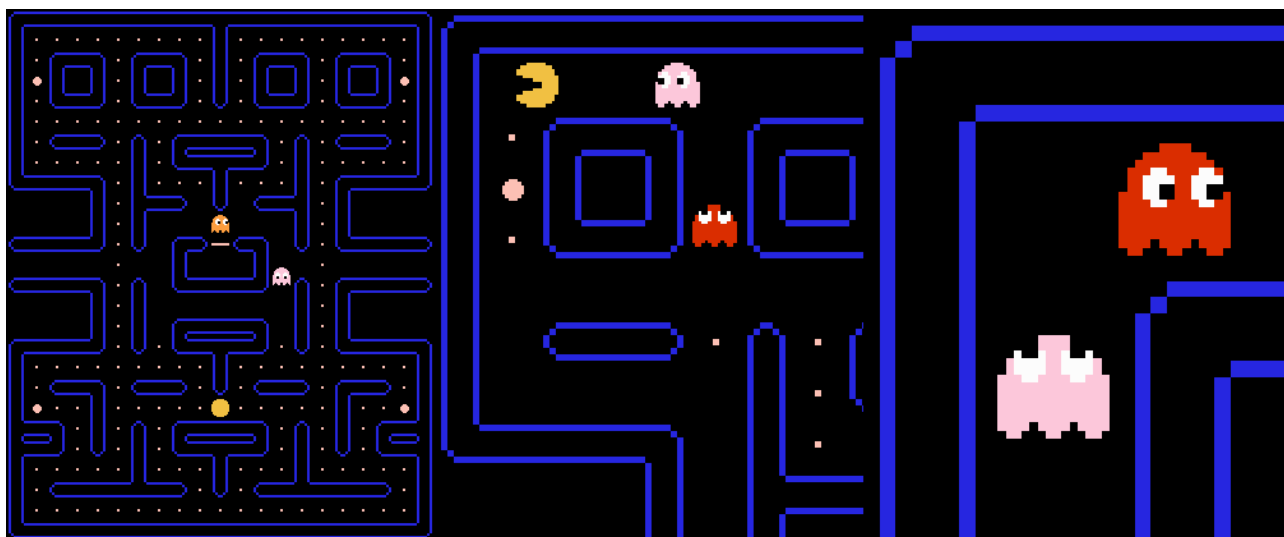


Figure 2 : échelle de dessin des cellules (1 ; 2,5 ; 6,25), affectant aussi les entités. Les *sprites* des cellules étant de moitié la résolution de celle des entités, leur échelle finale est effectivement doublée par rapport à la valeur donnée. Les arrondissements supportent des valeurs rationnelles non entières : 6,25 devient alternativement 12 ou 13.

Mécaniques du niveau

Les collisions sont déterminées en fonction de l'accessibilité de la cellule de la grille à l'avant du personnage qui se déplace, qui est testée par `Terrain::is_free`, or pour les fantômes, la porte de la cage peut parfois être traversée. Pour implémenter ce comportement, on utilise donc un argument optionnel, faux par défaut, qui indique si ladite porte est franchissable, ceci étant utilisé dans le comportement des fantômes.

Les tunnels sont implémentés en permettant aux personnages d'être d'une cellule en dehors du niveau, ce qui suit le comportement du jeu original. Une fois qu'ils se déplacent plus loin, ils sont téléportés de l'autre côté. On prétend donc qu'une cellule est accessible si et seulement si $x + w + 2 \bmod (w + 2) \geq w$, avec x la position entière en X et w la largeur du niveau, quoique dans une méthode séparée `Terrain::is_tunnel`, plutôt que `Terrain::is_free`, qui doit être utilisable comme garde pour les accès aux cellules.

Cette formule laisse deux cellules accessibles en bordure de la grille, l'une étant réellement traversable, et l'autre permettant juste aux personnages d'avancer suffisamment dans la précédente pour être téléportés. Comme les fantômes ne sont pas capables de faire demi-tour à moins de changer d'état, ils sont en mesure de pourchasser Pac-Man à travers les tunnels (figure 3).

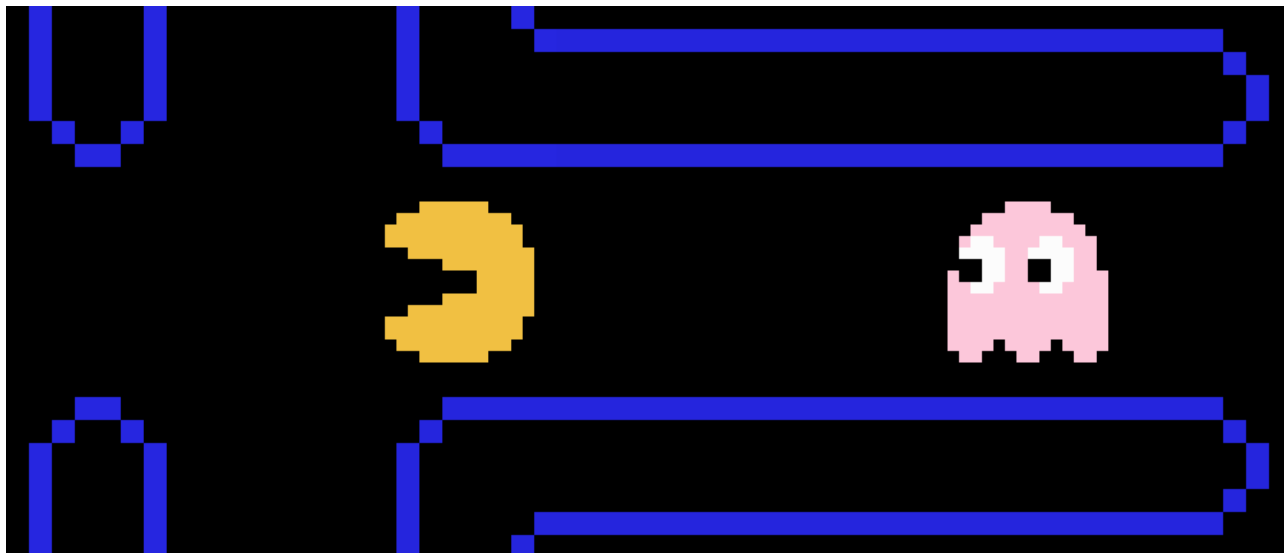


Figure 3 : Pac-Man pourchassé par Pinky à travers un tunnel, les deux étant entrés par la gauche et ayant été téléportés

Améliorations diverses

Nous utilisons, au départ, des `SDL_Point` pour les coordonnées en pixels, or la SDL étant une librairie C, aucun opérateur n'est surchargé pour cette structure. Afin de rendre le code plus lisible, ses opérateurs de multiplication ont été surchargés. Cette logique a été rendue obsolète par l'introduction de la classe `Vector2` par Pierre Even, et retirée.

Même s'il n'y a que cinq personnages dans le jeu, il devient redondant de gérer leur comportement commun par une liste d'appels. Cela a donc été remplacé par un vecteur de `shared_ptr` vers les entités, qui permet de simples appels *via* parcours par *for range* du vecteur en se reposant sur le polymorphisme.

Évidemment, j'ai participé à d'autres améliorations mineures du code, comme l'élimination de nombres magiques dépendants de l'échelle des textures ou encore la séparation des responsabilités de `tick` – comportement – et `draw` – affichage.

De plus, j'ai corrigé un bug où le fruit ne disparaissait pas à la mort de Pac-Man, et j'en ai profité pour factoriser du code et ajouter une surcharge à `Cell::update_sprite_handle`.

Pour finir, j'ai ajouté à la *sprite sheet* la texture « *GAME OVER* » qui, elle, n'était pas dans les ressources fournies – qui correspondent à la version NES et non la version arcade... – et implémenté son affichage, qui remplace la simple fermeture du jeu lorsqu'il est perdu.

Remerciements

En outre qu'évidemment nos enseignants de CM et de TP, Messieurs Thery et Ravaglia, je souhaite naturellement remercier mon ami et binôme Pierre Even, pour la pertinence de nos discussions sur les objectifs et les choix techniques et de conception, qui ont d'autant plus enrichi mes connaissances, ainsi que pour la qualité de l'architecture qu'il a constituée, fort de son expérience en conception de moteurs de jeu. Merci enfin à mon ami Baptiste Rostalski et au fameux jeu produit par le studio Team Cherry d'avoir empêché ma santé mentale de sombrer dans une spirale descendante malgré l'accumulation des projets et la nécessité de chercher un logement pour mon stage estival.



Figure 4 : « bulling » (« fungling »), *Hollow Knight* (Team Cherry, 2017)