Rapport projet PFA

Erwan LEMATTRE Ewen DUFOUR

Avril 2024

1 Le jeu

Le jeu *Arthur la quête de la cuillère* a pour but d'offrir aux joueurs une expérience de jeu agréable avec différents niveaux ayant chacun sa spécificité tout en racontant une histoire. L'histoire est le fil conducteur du jeu : elle permet de donner un sens aux différents niveaux.

Arthur la quête de la cuillère s'inspire en partie de la légende du roi Arthur et de la série *Kaamelott*. Nous avons essayé d'ajouter des références à la série tout en gardant un jeu cohérent pour les non-connaisseurs. On retrouve également une référence à *Star Wars* avec les opening des niveaux inspirés de ceux des films. Enfin, le joueur est accompagné d'une série de musiques médiévales tout au long du jeu qui le plonge d'avantage dans l'aventure du roi arthur.

Le jeu se compose de 4 niveaux et d'un menu. Le **menu** sert uniquement d'accueil du jeu. Le **niveau 1** introduit la problématique de l'histoire (opening) et permet au joueur de prendre en main le jeu avec un niveau plutôt simple. Le **niveau 2** lance réellement la quête du roi. Le joueur ayant une maîtrise des commandes du jeu, il doit maintenant parcourir un niveau plus difficile. On reste dans le même environnement plutôt plaisant. Avec le **niveau 3** les choses se compliquent. L'environnement devient plus hostile et la difficulté augmente. Le sol est maintenant glissant, les blocs sont plus espacés, certains blocs sont invisibles et les ennemis sont plus nombreux. Enfin le **niveau 4** est le niveau final dans lequel le joueur doit faire face au boss final. L'environnement devient lugubre, la musique indique au joueur la bataille finale. Une fois le squelette éliminé la quête est terminée, on retourne au menu.

Il existe 3 ennemis différents dans notre jeu. Les archers (niveaux 2 et 3), les chevaliers (niveaux 2 et 3) et **Alexandre Le Petit** l'ennemi final (niveau 4). Les **archers** ne se déplacent pas, ils tirent des flèches lorsque le joueur leur fait face. Les **chevaliers** suivent le joueur pour l'attaquer avec leurs épées. L'ennemi final peut tirer des boules de feu et mettre des coups d'épée. Ils suit également le joueur.

Le joueur de son côté a différentes capacités qui s'ajoutent à mesure que les niveaux augmentent. Au niveau 1 le joueur n'a aucune capacité particulière il peut seulement se déplacer et mettre des **coups d'épée**. Au niveau 2 le joueur peut utiliser la **téléportation** qui lui permet de se déplacer très rapidement en un coup. Attention la téléportation ne permet cependant pas de passer au travers des objets de l'environnement ou des ennemis. Au niveau 3 s'ajoute les **boules de feu** qui est l'attaque à distance du joueur. Au niveau 4 le joueur a les mêmes capacités qu'au niveau 3. Enfin le joueur peut trouver des **soins** dans les niveaux. Les soins sont représentés par les items poulet.

Nous avons pu mentionner précédemment que chaque niveau possède une musique différente afin de varier l'ambience. Nous avons voulu faire en sorte que plus le joueur avance dans le jeu, plus la musique devient pesante.

Pour terminer, le jeu se joue avec les touches `zqd` (ou directionnelles) pour les déplacements. Les touches utiles aux pouvoirs dépendent des préférences des joueurs. On utilise par défaut `Shift`, `space` et `s`. D'autres touches servent au débogage, nous le verrons dans une section dédiée. Les niveaux ont été conçus de manière à ce que le joueur se serve au moins une fois de chaque pouvoir mis à sa disposition.

2 Organisation du code

2.1 Les systèmes

L'organisation générale du jeu est proche de celle du code modèle utilisé au début de ce projet. On retrouve dans `src/systems/` l'ensemble des systèmes du jeu. Le système **control** permet d'avoir accès aux entrées du joueur. Il est utilisé par l'entité player et les boutons du jeu. Ce système va appeler une fonction dans l'entité qui va ensuite gérer les actions en fonction des entrées. Le système **real_time** permet de gérer les actions qui doivent se passer sans latence pour le joueur. À chaque frame il appelle une fonction spécifique à chaque entité présente dans le système. Il permet par exemple de gérer les ennemis. Le système **music** s'occupe de la gestion des musiques du jeu. Il contient une variable `current_track` qui permet de sélectionner la track à jouer. Les track sont prédéfinies dans le code elles ne peuvent pas être modifiées depuis l'extérieur. Le système **view** permet de gérer la caméra. Nous reviendrons sur cette partie plus en profondeur dans la suite de ce rapport.

2.2 Le répertoire `src`

Le répertoire `src/` contient trois nouveaux fichiers.

Le fichier `level_loader.ml` permet le chargement les niveaux depuis des fichiers texte. C'est une partie qui a été particulièrement intéressante à implémenter. L'objectif a été de pouvoir créer des niveaux sans avoir besoin de connaître le fonctionnement interne du jeu. Nous pensons être plutôt proche de cet objectif même si certaines choses pourraient être améliorée. Les fichiers textes doivent contenir un ensemble de lignes chacune permettant la création d'un élément du jeu. On utilise la notation `id:XxY|WxH|param` avec id l'identifiant de l'entité (Les identifiants des entités et leur description sont disponibles dans le fichier `map.md`.), XxY la position XY, WxH la largeur et hauteur de l'entité et param les paramètres associés à l'entité. Cette ligne est découpée pour récupérer chacune des parties. Les paramètres entrés sont ajoutés dans une table de hachage. Lors de la création d'une entité on récupère un type énuméré `setting` définit dans `const.ml`. Ce type contient les différents paramètres possibles. Les entités peuvent ensuite récupérer les paramètres dont elles ont besoin dans cette structure. Ce fonctionnement permet d'avoir une flexibilité dans les déinitions de niveau. On peut par exemple donner des paramètres qui n'existent pas, cela n'aura aucune influence. On peut également ne pas définir un paramètre qui est nécessaire et dans ce cas une valeur par défaut sera donnée sans causer d'erreur.

Le fichier `resources.ml` gère le chargement des fichiers image, texte, audio et police du jeu. Le chargement est effectué dans le fichier `game.ml` avant d'afficher le menu au démarrage du jeu.

Le fichier `config.ml` contient la définition des touches pour Javascript et Sdl. La configuration est utilisée par les entités utilisant le système contrôle afin d'avoir accès au nom des touches.

2.3 Le répertoire `core`

Le répertoire `src/core/` contient 2 nouveaux fichiers.

Le fichier `const.ml` contient l'ensemble des constantes du jeu. On y trouve par exemple la vitesse du joueur, la taille des blocs, etc...Ce regroupement permet de modifier facilement les valeurs importantes du jeu sans avoir besoin de chercher tous les endroits où elles sont utilisées. Ce fichier contient également le type `setting` dont nous avons parlé dans la section précédente.

Le fichier `state.ml` contient la définition du type `state`. Ce type est utilisé par le joueur et les ennemis afin de contrôler les animations des attaques. L'état 0 de state signifie qu'aucune attaque est en cours. L'état 1 signifie que l'attaque est en cours. Elle est mise à jour grâce à une fonction `update` qui gère l'animation en fonction du numéro de la frame courante.

Dans le fichier `global.ml` nous avons ajouté la caméra et le joueur. On peut ainsi accéder au joueur depuis tous les fichiers, ce qui est utile pour gérer le comportement des ennemis.

2.4 Components

De nombreux composants ont été créés pour notre jeu. Les composants avec box dans leur nom permettent de créer des variations de box avec des comportement différents. Par exemple les *hide_ box* sont des box qui sont ajoutés dans les systèmes *draw* et *view* seulement lorsque la box est en collision avec le joueur. Ci-dessous une rapide description des composants de notre jeu.

Composant	Description
alexandre	Boss final du jeu *Alexandre Le Petit*.
arch	Archer, tire ne se déplace pas, tire des flèches.
arrow	Flèche tirée par l'archer. Détruite lorsqu'elle entre en collision avec n'importe quel objet.
audio	Composant audio contenant la piste d'écoute courante et le numéro de la musque écouté dans la piste.
background	Utilisé pour créer le fond du menu.
box	Une box simple avec toutes les paramètres possibles dessus.
bullet	Boule de feu pour le joueur et l'ennemi final. Fonctionnement similaire à `arrow`.
button	Bouton, notamment utilisé dans le menu.
camera	Composant caméra contenant le focus (discuté dans la suite).
decor	Affiche une texture tout en ayant aucune intéraction avec les autres éléments du jeu.
exit_box	Box permettant de changer de niveau lorsque le joueur entre en collision avec.
fall_box	Box qui est affectée par la gravité lorsque le joueur entre en collision avec.
hide_box	Box qui est affichée seulement lorsque le joueur entre en collision avec.
hitbox	Outil de débogage, voir section tests.
hpbar	Barre affichant la vie de l'ennemi final. Affiche un rectangle proportionnel à la vie du composant `alexandre`.
jump_box	Box permettant de faire rebondir le joueur. Le fait de pouvoir rebondir plus haut implique une élasticité supérieure à 1. Or une élasticité supérieure à 1 fait rebondir de plus en plus haut le joueur. C'est pourquoi on à limité la vélocité dans le système *collisions* en utilisant `clamp` du module `Vector` (ajouté).
knight	Chevalier, attaque seulement avec une épée et suit le joueur lorsqu'il est assez près.
medkit	Soin pour le joueur.
opening	Opening en début de niveau. Crée le texte de l'opening et gère la caméra. Une fois l'opening terminé le texte est retiré des systèmes et on laisse les commandes au joueur (`Control.disable` devient faux).
player	Joueur avec ses fonctions de contrôle, collisions, mise à jour en temps réel.
superuser	Outil de débogage, voir section tests.
sword_box	Zone de collision de l'épée. L'épée du joueur et des ennemis étant associée à la texture, nous avons créé une box qui apparaît lorsque le joueur utilise l'épée. On peut ensuite simplement utiliser le système de collisions pour savoir si on a été touché par une épée.
text	Un élément de texte

3 Organisation du travail

4 Les fonctionnalités

4.1 La caméra (Erwan)

Une fonctionnalité que j'ai pu implémenter pour le jeu et que j'ai trouvé intéressante est la caméra. Au début, nous avions un jeu avec un joueur que l'on pouvait déplacer mais lorsqu'on se déplaçait trop d'un côté le joueur disparaissait. Il a donc fallu trouver un moyen de suivre le joueur dans le jeu. J'ai donc créé un objet caméra qui permettrait de voir le jeu non plus en partant de la position 0,0 mais à l'endroit où se trouve la caméra. Il a ensuite suffit de donner à la caméra la position du joueur pour pouvoir suivre le joueur. Je vais expliquer dans cette section comment a été implémenté la caméra.

Pour commencer la caméra a deux modes : `position` qui place la caméra en fonction de son attribut `pos` et le mode `player` qui place la caméra à la position du joueur. Ce second mode utilise l'accès au joueur du fichier `global.ml`. La vue est gérée dans le système `view.ml`. Chaque entité affichable (*drawable*) possède un composant *camera_position* qui est la position à l'écran. On différencie donc maintenant la position à l'écran et la position dans l'espace du jeu qui est le composant *position*. Le système *view* va calculer à partir de la position de la caméra et du composant la position à l'écran du composant. L'algorithme est plutôt simple, pour chaque entité faire `position_entité - position_caméra`.

5 Les tests effectués

Nous avons développé différents outils permettant de tester notre jeu. Nous allons expliquer dans cette section les différents moyens mis en place afin de tester notre code.

5.1 Les hitbox

Les hitbox sont les zones de collision des objets. Au départ, les collisions se faisaient en fonction de la taille des objets (un rectangle de 10x10 a pour collision l'ensemble de la zone de taille 10x10). Le problème avec cette méthode est que s'il y a une zone de transparence sur une texture, l'utilisateur voit une collision avec du vide. C'est la raison pour laquelle nous avons créé les hitbox. Une hitbox est simplement représentée par un décalage en x et en y et une taille w, h. Dans le système de collision on prend ensuite compte le décalage et la taille de la hitbox. Une hitbox n'est donc pas réellement une nouvelle box. Afin d'ajuster les hitbox au mieux nous avons donc créé un composant hitbox qui affiche la hitbox comme un rectangle au-dessus de l'entité. Ce premier outil nous a permis d'avoir des hitbox collisions réglées au pixel près.

5.2 Le mode superuser

Le mode superuser est un outil qui a été indispensable dans le développement du jeu. D'abord il permet d'accéder directement à un niveau en utilisant les touches 0 à 4, 0 étant le menu. Ensuite il permet également de modifier certaines constantes. Dans la

configuration actuelle, on a par exemple les touches u et j pour augmenter et diminuer la vitesse du joueur.