Rapport Projet PFA – Casse-briques

Badreddine Channoufi, Chang-Young Jung

1 Introduction

À partir du squelette du TP6, nous avons créé un casse briques simple qui utilise le modèle ECS. On a augmenté le code de base avec de nouveaux composants, de nouvelles entités, un système de collisions adapté à la mécanique du casse-briques et un affichage graphique avancé. Chacun des ajouts sera détaillé dans le rapport. Finalement, on expliquera aussi comment on aurait pu intégrer un bonus "multiballes".

2 Logique du jeu

Le joueur contrôle une raquette séparée en trois parties qu'il ne peut déplacer séparément. La raquette se déplace horizontalement et est en bas de l'écran. Une balle se déplace entre les briques situées en haut de l'écran et la raquette. À chaque fois que la balle touche une brique, la brique est détruite. La partie est gagnée si toutes les briques sont détruites et perdue si la balle arrive sous la raquette. Le joueur doit donc réceptionner la balle et l'envoyer vers les briques. Les briques renvoient la balle avec une vitesse accélérée, avec un coefficient qui peut aller jusqu'à 1.05 pour les briques les plus en haut. La difficulté augmente donc au fil de la partie. Les raquettes gauches et droites de la balle sont orientées et permettent de faire des tirs visés qui ont toujours la même direction.

3 Organisation du code

Le code suit la même organisation que celle du TP6, c'est-à-dire que le programme consiste en une boucle infinie (gérée par la biliothèqye gfx) qui termine lorsque la partie est terminée. À chaque itération de boucle, on gère la logique du jeu puis on appelle successivement les *systèmes* qui vont se mettre à jour. Un système est un module qui permet :

- d'enregistrer des *entités*
- de les mettre à jour en modifiant leurs *composants*
- de les supprimer du système

Le jeu suivant la logique *ecs*, on a donc dans dans la source du code des fichiers séparés pour les entités, les systèmes et les composants.

On doit donc créer des systèmes qui gèrent le déroulement du jeu. On a un système pour le dessin qui enregistre des entités qui peuvent être dessinées, un système pour les collisions qui gèrent des entités ayant une masse, une dimension et une position, etc. On a autant de systèmes que dans le TP6 (forces, collisions, draw, move) donc il n'y a pas d'ajout à ce niveau-là. Les systèmes sont dans le dossier "systems" mais sont créés par un foncteur *System* qui lui est dans la bilbiothèque ecs fournie.
Les composants sont définis dans le fichier "Component_defs.ml".
Les entités sont toutes définies dans le module "Entity" en tant que sous-modules. C'est plus simple car elles héritent toute d'une entité basique "block" qu'on modifie après.
Dans le dossier utils, il y a les types utiles au calcul vectoriel et à la gestion des surfaces mais également un type ajouté qui représente la nature d'une entité. Ce nouveau type sera utile pour faire du pattern-matching sur la nature des entités dans les systèmes.

4 Les ajouts faits aux systèmes et composants

- Le système draw enregistre des entités qui ont une taille, une position et une texture. La mise à jour de ce système se fait en effacant tout ce qu'il y a à l'écran puis en redessinant tout en itérant sur les entités enregistrées. Cependant, ceci ne garantit en rien l'ordre dans lequel les objets sont dessinés. C'est gênant si on veut un fond d'écran, car on doit le dessiner en premier si on ne veut qu'il occulte d'autres objets.
  On modifie donc la fonction update pour qu'elle affiche en premier une surface *wp*. Cette variable est en fait de type ref (Some Surface). Si elle est égale à None, on affiche un fond d'écran blanc, si elle est égale à Some s où s est une surface, on affiche s en tant que fond d'écran.

```
let wp = ref None

let update _dt el =
  let win = Game_state.get_window () in
  let ctx = Gfx.get_context win in
  let win_surf = Gfx.get_surface win in
  let w, h = Gfx.get_context_logical_size ctx in
  let () = Gfx.set_color ctx white in
  let () =
  match !wp with
  | Some s -> Gfx.blit_scale ctx win_surf s 0 0 w h
  | None -> Gfx.fill_rect ctx win_surf 0 0 w h
...
```

- Le système collisions lui a été augmenté en ce qu'il enregistre maintenant des entités qui ont comme composants supplémentaires un booléen touched, une nature de type Nature et une résistance.
  Si on détecte une collision entre deux objets, on met leur booléen touched à true. Ceci sera utile pour la logique du jeu implémentée ailleurs.
  S'il y a une collision entre la balle et une raquette gauche ou droite (qu'on détecte avec du pattern-matching sur le composant nature), on veut que la balle parte dans une direction fixée mais avec la même vitesse. Pour ça, on prend le vecteur vitesse de la balle et on calcule sa norme. Ensuite, selon que l'on va à gauche ou à droite, on prend un vecteur fixé de norme un qu'on multiplie par la norme. Finalement, on dit que la somme des forces de la balle est égale à ce nouveau vecteur.
  S'il y a une collision entre la balle et une brique, le calcul usuel se fait mais en intégrant cette fois l'élasticité de la brique qui est une composante. Ce coefficient augmentera ou baissera la magnitude du nouveau vecteur vitesse.

- Pour intégrer ces fonctionnalités on a ajouté trois nouveaux composants : un float resistance pour un calcul des collisions plus complexe, une nature de type nature (définie dans le dossier utils) qui représente la nature de l'entité, un booléen touched qui dit si un objet est touché ou non (pour savoir si on détruira les briques).

5 Les entités

Les entités sont autant que les éléments logiques du jeu : les murs, les briques, le joueur, la balle. Chaque entité est un sous-module du module entity, et implémente "make" (on enregistre l'entité dans les systèmes) et "destroy" (on

la retire de tous les systèmes). Les composants des entités sont ceux définis dans components_def.ml (dimensions, résistance, masse...). En ayant des entités séparées, on peut se permettre de ne pas enregistrer certaines entités dans certains systèmes. Par exemple, on enregistre jamais un mur dans le système des forces, car un mur a une masse infinie. Voir entity.ml pour les détails, mais ce fichier est assez simple.

6 game.ml

C'est dans le fichier game.ml que s'écrit la boucle principale et les fonctions utiles pour la logique du jeu.
On crée les entités qui composent le jeu avant tout. En particulier, les briques sont représentées par une référence de liste d'entités Briques. Le jeu est gagné lorsque la liste est vide.
Dans la boucle de mise à jour, on détecte si la partie est finie, c'est-à-dire si la liste des briques est vide ou que la balle est en dehors du tableau. Auquel cas on affiche l'écran final puis on renvoie faux pour arrêter la boucle. Sinon, on regarde si le joueur appuie sur une touche et on le déplace en conséquence. Puis on update la liste des briques en supprimant les briques touchées (information venant de la dernière itération de boucle lorsqu'on a mis à jour le système collision).

```
bricks := List.filter
          (fun b -> let touched = b#touched#get in
                    if touched then
                      Entity.Brique.destroy b; (* enlève la brique des systèmes *)
                    not touched
          ) !bricks
```

On aurait pu séparer ce fichier pour avoir les entités créées dans un module séparé, ou bien créer un nouveau système pour les briquer qui implémente update plutôt que d'appeler update_bricks nous même.

7 Ajout d'un bonus multiballes

Finalement, le jeu est fonctionnel mais est assez basique. Avec notre structure de code actuelle, comment pourrait-on implémenter un bonus "multiballes" ? C'est-à-dire, lorsqu'une certaine brique est touchée, deux autres balles apparaissent sur l'écran. Le fait qu'une nouvelle balle disparaisse de l'écran ne termine pas la partie, qui se termine plutôt lorsqu'il n'y a plus aucune balle à l'écran.
On pourrait ajouter un composant booléen "bonus" à l'entité Brique, et un compteur modifiable de type ref Int. On ajoute donc un paramètre bonus à la fonction Entity.Brick.Make pour déterminer si une brique est bonus ou pas.
Lors de l'appel à update_bricks (ou dans un monde plus sophistiqué, au système Brick) on regarde si une briquee cassée avait l'attribut bonus à true. Si oui, on crée deux balles avec Entity.Ball.Make dont les coordonnées sont fixées (par exemple, x=100 et y=500 et x=500 et y=500 ; assez bas pour qu'elles ne soient pas en collision avec la balle qui vient de toucher une braque un instant dt avant) avec une force verticale (de façon analogue à la création de la première balle). On incrémente aussi le compteur de 2.
À chaque tour de boucle, on regarde ainsi si une balle est sortie de l'écran et si c'est le cas on décrémentre le compteur. La condition de fin de partie devient alors "le compteur est égal à zéro".