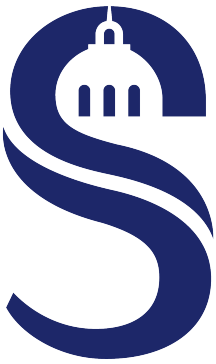

SORBONNE
UNIVERSITÉ

Rapport de projet
Street Fighter en Haskell

Adan Bougherara
Vivien Demeulenaere

Remis le 08 Mai 2022

Table des matières

1 Introduction

Ce projet vise a reproduire partiellement un jeu de combat du type *Street Fighter*. Pour se faire, nous utiliserons le langage **Haskell** permettant une programmation sûre guidée par les types. Le code accompagnant ce rapport contient donc le jeu dans le dossier `src` ainsi qu'une batterie de tests contenue dans le dossier `test`.

Figure 1 – Aperçu du jeu

2 Manuel d'utilisation du jeu

L'objectif de cette partie est d'expliquer le fonctionnement du jeu. Pour lancer ce dernier, il faut disposer de la bibliothèque **SDL 2** sur linux et du compilateur `lts-18.28` **Haskell**.
Afin de lancer le jeu, il conviendra d'utiliser la commande `stack run` depuis le répertoire `PAF-STREETFIGHTER-2022`.
Pour lancer les tests, il faudra utiliser la commande `stack test`, toujours depuis ce répertoire.

Au lancement, l'utilisateur (joueur 1) peut choisir entre jouer contre un autre joueur (joueur 2) ou contre l'ordinateur. Les actions suivantes sont alors possible :

Action	Touche pour le Joueur 1	Touche pour le Joueur 2
Se déplacer à droite	D	→
Se déplacer à gauche	G	←
Sauter	Z	↑
S'accroupir	S	↓
Donner un coup de poing	A	↵
Donner un coup de pied	E	⇧ (droite)
Envoyer un projectile	F	P
Se protéger (des coups uniquement)	R	P
Quitter le jeu	Esc	Esc

Table 1 – Touches associées aux différentes actions

3 Propositions

Cette partie vise à présenter brièvement les différentes propositions écrites afin de rendre le jeu plus sûr.

3.1 Invariants

Tous les types de données du fichier `Model.hs` contiennent des invariants à l'exception de `Technique` et de `Coord` qui ne sont respectivement que des énumérations des différentes actions réalisables par un joueur (en dehors des déplacements) et des directions possibles pour un mouvement.

3.1.1 Zone

On définit comme suit une zone de jeu à l'aide de deux entiers, l'un représentant la largeur du terrain et l'autre la hauteur.

```
data Zone = Zone Integer Integer deriving Show
```

L'invariant de `Zone` s'assure que sa hauteur et sa largeur sont strictement positives (cf. `prop_Zone_inv`).

3.1.2 Coord

Pour définir l'emplacement d'un point dans le terrain, on définit le type de données suivant :

```
data Coord = Coord Integer Integer deriving (Show, Eq)
```

Le premier entier précise la coordonnée sur l'axe de abscisses et la seconde sur l'axe des ordonnées. Il est à noter que la coordonnée en haut à gauche du terrain est (0,0), l'axe des abscisses augmente donc en allant à droite et celui des ordonnées en allant en bas. Naturellement, `prop_Coord_inv` vérifiera qu'une coordonnée est bien dans le terrain.

3.1.3 Mouvement

Afin de déplacer une coordonnées, on introduit les types de données `Mouvement` et `Direction`.

```
data Direction = H | B | G | D deriving (Show, Eq)
data Mouvement = Mouv Direction Integer deriving Show
```

L'invariant `prop_Mouvement_inv` se contentera de vérifier que le déplacement est supérieur à 0.

3.1.4 Hitbox

Dans le but de représenter une entité sur le terrain, on met en place des `Hitbox`. Celles-ci peuvent être un simple rectangle (défini par son coin supérieur gauche, sa largeur et sa hauteur) ou bien une `Sequence` de `Hitbox`.

```
data Hitbox =
    Rect Coord Integer Integer
    | Composite (Seq Hitbox)
    deriving (Show, Eq)
```

La propriété `prop_Hitbox_inv` considère alors comme valide toute `Hitbox` entièrement contenue dans la zone et dont l'aire est non nulle.

3.1.5 EtatCombattant

On ajoute également un état au combattant. Il est soit `KO`, soit en vie avec un certain nombre de points de vie.

```
data EtatCombattant =
    Ko
    | Ok Integer
    deriving (Show)
```

On s'assurera à l'aide de la propriéte `prop_EtatCombattant_inv` que les points sont toujours positifs.

3.1.6 Combattant

Nous avons maintenant à disposition les ingrédients pour créer un type de données `Combattant`.

```
data Technique = Rien
    |CoupDePoing
    |CoupDePied
    |Protection
    |Encaisse
    |Accroupi
    |Lancer
    deriving (Eq, Show)

data Combattant =
    Comb {
    hitboxc :: Hitbox,
    facec :: Direction ,
    etatx :: EtatCombattant,
    hitTimer :: Integer,
    dammage :: Integer,
    techniquec :: Technique,
    hitStun :: Integer,
    combo :: Integer
    }
    deriving(Show)
```

Un `Combattant` possédera donc une `Hitbox` pour le détourer, une direction pour savoir de quel côté il regarde, un état, un compteur pour qu'il ne puisse pas enchaîner trop de coups d'affilée, les dégâts qu'il fait, la technique qu'il est en train de réaliser ou encore un compteur dans le cas où il prendrait un coup.
La propriété `prop_Combattant_inv` vérifie qu'un `Combattant` possède une `Hitbox` valide et que ni ses compteurs, ni ses dégâts sont négatifs.

3.1.7 Projectile

Les `Combattant` pouvant lancer des projectiles, on introduit le type de données suivant :

```
data Projectile =
    Proj {
        proprietaire :: Integer,
        hitbox :: Hitbox,
        puissance :: Integer,
        mouvp :: Mouvement
    }
    deriving(Show)
```

On vérifiera alors à l'aide de `prop_Projectile_inv` qu'un `Projectile` a pour propriétaire le joueur 1 ou 2 (respectivement représentés par les entiers 1 et 2), que sa `Hitbox` ainsi que son `Mouvement` sont valides et enfin que les dégâts qu'il inflige soient positifs.

3.1.8 Jeu

A présent, il est possible de créer un `Jeu` composé de deux joueurs, d'un terrain et de projectiles.

```
data Jeu =
    GameOver Integer -- le numero du joueur vainqueur
    | EnCours {
        joueur1 :: Combattant ,
        joueur2 :: Combattant ,
        zoneJeu :: Zone,
        projectiles :: Seq Projectile
    }
    deriving(Show)
```

L'invariant `prop_Jeu_inv` s'assure qu'un jeu fini indique quel joueur a gagné la partie et qu'un jeu en cours est un jeu dont les deux combattants sont en vie et respectent leur invariant `prop_Projectile_inv`. De plus, la zone de jeu et les projectiles doivent également respecter leurs invariants (respectivement `prop_Zone_inv` et `prop_Projectile_inv`).

3.2 Pré-conditions et Post-conditions

Cette partie énumère les pré-conditions et post-conditions des fonctions du fichier `Model.hs` incluant à la fois le jeu de base et les extensions. Il est toutefois à noter qu'aucune propriété n'a pu être mise en place pour les jauges de santé, cette extension s'appuyant sur les points de vie des joueurs et étant visuelle.

3.2.1 appartient

La fonction `appartient` permet de déterminer si une `Coord` appartient ou non à une `Hitbox`. Elle ne possède qu'une pré-condition `prop_pre_Appartient` vérifiant que la coordonnées et la zone vérifient bien leur invariant.

3.2.2 bougeHitBoxSafe

La fonction bougeHitBoxSafe permet de déplacer une `Hitbox` si le déplacement ne la fait pas sortir de la zone. La fonction `prop_pre_BougeHitboxSafe` veillera à ce que la `Hitbox` et le `Mouvement` respectent leur invariant. De plus, la fonction `prop_post_BougeHitboxSafe` vérifiera qu'en cas de succès la `Hitbox` calculée respecte bien son invariant.

3.2.3 bougeCoordSafe

La fonction `bougeCoordSafe` est dans le même esprit que la fonction précédente et n'a qu'une post-condition, `prop_post_BougeCoordSafe` qui s'assure que la coordonnée calculée soit bien valide.

3.2.4 bougeCoord

Cette fonction permet de déplacer une coordonnée sans se soucier de l'invariant. On aura donc une seule propriété, `prop_post_bougeCoord`, vérifiant qu'un déplacement d'une certaine distance vers la droite puis vers la gauche reviendra bien à ne pas bouger.

3.2.5 collision

La fonction `collision` permet de calculer si deux `Hitbox` rentrent en collision. On vérifiera simplement que ces dernières vérifient leur invariant grâce à `prop_pre_Collision`.

3.2.6 initJoueur1

Il s'agit d'une fonction permettant d'initialiser le joueur 1 selon les dimensions de ses images. La fonction `prop_pre_Initjoueur1` vérifie alors que les points de vie, le `hitTimer` et le `hitStun` sont positifs. De plus, les coordonnées doivent être dans la zone et l'aire du personnage doit être non nulle. Une post-condition nommée `prop_post_Initjoueur1` permet de s'assurer que la fonction calcule bel et bien un `Combattant` respectant son invariant.

3.2.7 initJoueur2

Cette fonction est en tout point similaire à la précédente au niveau des propriétés qui lui sont applicables.

3.2.8 initHitboxProj (extension)

Cette fonction calcule une `Hitbox` pour un projectile. Afin de faire apparaître ce dernier au bon endroit, il est nécessaire de connaître son propriétaire. Il est représenté soit par 1 soit par 2. C'est la fonction `prop_pre_InitHitboxProj` qui permet de s'en assurer.

3.2.9 positionJoueurX & positionJoueurY

Ces fonctions permettent de calculer les abscisses et les ordonnées d'un combattant (ceux de la `Hitbox` principale, les autres représentant des surfaces pour frapper). Les propriétés `prop_pre_PositionJoueurX` et `prop_pre_PositionJoueurY` vérifient que le `Combattant` est valide.

3.2.10 positionProjectileX & positionProjectileY

Ces fonctions sont similaires aux deux précédentes et les propriétés `prop_pre_PositionProjectileX` et `prop_pre_PositionProjectileY` aussi. La seule différence résidant dans le fait que l'on traite du type de données `Projectile` et non pas de `Combattant`.

3.2.11 acualiserFace

La fonction `actualiserFace` permet de calculer de quel côté regarde un joueur. Elle prend en argument un numéro de joueur et le jeu. La propriété `prop_pre_ActualiserFace` vérifiera donc que le numéro est bien 1 ou 2. Enfin, `prop_post_ActualiserFace` s'assurera que les joueurs regardent bien soit à gauche soit à droite.

3.2.12 bougeJoueur & bougeJoueurJeu

Cette fonction sert à déplacer un joueur. Comme précédemment, il faudra passer le numéro du joueur à déplacer en paramètre. La propriété `prop_pre_BougeJoueur` s'assurera donc à la fois que le numéro est soit 1 soit 2 mais aussi que le `Mouvement` et le `Jeu` respectent leur invariant respectif. Une propriété `prop_post_BougeJoueur` est également disponible afin de vérifier que le `Jeu` calculé est valide.
La fonction `bougeJoueurJeu` est similaire à `bougeJoueur`, tout comme ses propriétés.

3.2.13 bougeProjectile (extension)

Il s'agit d'une fonction pour déplacer un `Projectile`. La propriété `prop_pre_bougeProjectilesJeu` exige que le `Jeu` passé en paramètre soit valide. La fonction retourne un couple de la forme `(Maybe Projectile, Combattant)` en enlevant des points de vie au `Combattant` adverse s'il y a lieu et calculant la nouvelle position du projectile s'il est toujours dans le terrain. La propriété `prop_post_bougeProjectilesJeu` vérifiera alors que le `Combattant` retourné soit toujours valide et que le `Projectile` éventuellement retourné le soit aussi.

3.2.14 ajouteProjectile (extension)

La fonction `ajouteProjectile` ajoute à la `Sequence` de `Projectile` du jeu un nouveau `Projectile`. Il sera nécessaire de vérifier à travers la propriété `prop_pre_ajouteProjectile` que le numéro de joueur fourni est bien le 1 ou le 2 et que le `Jeu` est valide. En échange, la propriété `prop_post_ajouteProjectile` assure qu'il n'y a pas plus d'un projectile par joueur dans le jeu.

3.2.15 pointsDeVie

Cette fonction calcule les points de vie d'un `Combattant`. On vérifiera à travers la propriété `prop_post_PointsDeVie` que le résultat est bien positif.

3.2.16 changerPosture & techniqueJeu

Ces deux fonctions requirent un numéro de joueur valide (cf `prop_pre_techniqueJeu` et `prop_pre_changerposture`). Elles préservent toutes deux l'invariant de `Jeu` (`prop_post_techniqueJeu` et `prop_post_changerposture`)

3.2.17 degats

La fonction `degats` prend en argument un `Combattant` et un `Integer` représentant les dégâts à appliquer à ce dernier. La propriété `prop_post_degats` vérifie donc que l'opération ait bien été effectuée sur le `Combattant` retourné.

3.2.18 prochaineDirection (extension)

Cette fonction est utilisée pour calculer la prochaine direction que l'ordinateur va prendre lorsqu'il incarne un joueur. La propriété `prop_post_prochaineDirection` s'assure que la direction retournée est soit `G` soit `D`.

3.2.19 gameStep & gameStepRandom (extension)

Il s'agit des deux fonctions calculant le nouvel état du `Jeu` à partir d'un `KeyBoard`. La version `random` est celle utilisée lorsque l'on joue contre l'ordinateur et prend un `Float` généré aléatoirement au préalable. Ces deux fonctions préservent l'invariant de `Jeu` (cf `prop_post_gameStep` et `prop_post_gameStepRandom`).

4 Jeu de base

Le jeu de base est un jeu de combat (en 2D et 60 images par secondes) qui nécessite deux joueurs, contrôlant chacun un personnage et dont les actions sont lues sur le même clavier.
Les règles du jeu sont les suivantes : les deux joueurs ont au départ 100 points de vie. Plusieurs actions sont disponibles pour un personnage, il peut se déplacer à gauche et à droite, sauter, s'accroupir, attaquer avec son bras ou sa jambe où enfin se protéger.
Un combattant qui pratique une technique ne peut pas se déplacer (hitTimer), tout comme un combattant qui subit des dégâts (hitStun).
Le temps du hitStun (10 frames) a été établi comme plus court que celui du hitTimer (15 frames) afin d'empêcher les combos infinis et d'offrir une porte de sortie au joueur qui reçoit l'attaque.
Dans ce projet, toutes les méthodes qui n'ont pas de style fonctionnel pures, c'est-à-dire les monades `IO()` et les entrées/sorties sont utilisées uniquement dans le fichier `Main.hs`.
La boucle de jeu est représentée par la méthode `gameLoop` dans le fichier `Main.hs`. Cette méthode retourne une monade `IO()`. Elle délègue dans un premier temps la gestion des évènements à `handleEvents` de `Keyboard.hs`, qui s'occupe de transformer les évènements en touches de claviers. Puis, elle va afficher les objets de l'interface graphique, à savoir le fond d'écran et les joueurs. Il est à noter que toutes les images sont pré-chargées au début de la monade du main. Enfin la boucle de jeu finit par calculer le nouveau `Jeu` avec la méthode `gameStep` de `Model.hs`.
On applique au jeu une composée de fonctions qui associent les touches du claviers qui sont appuyées avec les actions correspondantes (cf. Section 2) pour calculer le nouvel état du jeu.
Une action peut être un déplacement ou une technique.
Un déplacement est géré par la méthode `bougeJoueurJeu`, qui va appliquer de manière sûre le déplacement d'un combattant dans le jeu en utilisant `bougeHitboxSafe` qui annule l'action si elle implique une collision.
Il est à noter que tout déplacement est interdit lorsqu'un combattant possède un hitTimer ou un hitStun non nul, à l'exception de la redescente d'un combattant suite à un saut, qui est automatiquement gérée par la fin de la composée de fonction de `gameStep`.
Une technique est gérée par la méthode `techniqueJeu`, qui va dans un premier temps s'assurer que la technique est réalisable de manière sûre.
En effet, changer la posture d'un combattant revient à changer la taille de la hitbox de celui-ci voir même de modifier sa position. Il convient donc de s'assurer que cette nouvelle posture n'entraîne pas de comportement non désiré comme une sortie de terrain ou une collision incorrecte avec l'autre combattant. Les techniques présentes dans le jeu de base sont les suivantes :

- Se protéger - Immunise des attaques
- Attaquer avec le poing - Enlève 2pv (Joueur1) ou 5pv (Joueur 2)
- Attaquer avec la jambe - Enlève 5pv (Joueur1) ou 2pv (Joueur 2)

5 Extensions réalisées

4 extensions ont été réalisées pour ce jeu. Les fonctions présentées ci-dessous dont la provenance n'est pas explicitée viennent du fichier `Model.hs`

5.1 Combos

Cette extension permet à un joueur d'effectuer un combo en utilisant le coup de poing. Le coup de poing est la seule attaque qui a un hitTimer plus faible (5 frames) que le hitStun des combattants (10 frames). Il peut ainsi enchaîner un combo à partir du moment ou il réussit à toucher le premier coup malgré la portée faible de cette technique. Il faut cependant veiller à interdire les combos infinis.
Pour implémenter cette extension, nous avons ajouté un entier dans le type Combattant qui correspond au combo que le joueur est en train de subir.
On incrémente sa valeur lorsqu'un joueur reçoit des dégâts (fonction `degats`). On réinitialise le compteur à 0 lorsqu'un joueur change sa posture (fonction `changePosture`) pour une posture autre que `Encaisse`.

Afin d'éviter les boucles infinies, nous avons ajouté dans `gameStep` (et `gameStepRandom`) la condition qu'un coup de poing ne peut pas être lancé si la valeur de combo de l'adversaire est supérieure ou égale à 5.
Par conséquent, un joueur subissant un combo par un adversaire qui l'enchaîne de coups de poing aura la possibilité de s'enfuir, se protéger ou autre après 5 coups subits d'affilés.

5.2 Jauge de vie

Cette extension permet l'ajout de deux rectangles sur l'interface graphique agissant comme des barres de vie pour chaque combattant. Pour implémenter cette extension, qui est relative à l'affichage, le main appelle `loadHealthState` qui va ajouter un rectangle au rendu graphique, puis `fillHealthState` va remplir le rectangle en fonction de la vie du joueur. Ces deux fonctions proviennent du fichier `TextureMap.hs`

5.3 Intelligence artificielle

Un mode PvE est disponible au démarrage du jeu, permettant de jouer une partie contre un bot basé sur du pseudo aléatoire. Le bot joue avec le deuxième combattant.
S'il choisit de jouer contre l'IA, la boucle de jeu appelle `gameStepRandom` et non plus `gameStep`.
Cette méthode lit les touches du joueur 1 comme précédemment. Les actions du joueur 2 sont basés sur de l'aléatoire :

- 50% de chance de se déplacer latéralement
- 40% de chance d'attaquer
- 5% de chance de se protéger
- 5% de chance de sauter

Afin de rendre le bot plus fort, nous avons également apporté les modifications suivantes :
Le déplacement latéral est géré par `prochaineDirection` (accompagnée de `prop_post_prochaineDirection`).
Le bot va toujours vers le joueur. De plus, s'il détecte qu'il est sur une extrêmité du terrain, il partira dans le sens opposé (utile dans le cas ou le joueur 1 est au dessus/en dessous du bot).
Lors d'une attaque, le bot détermine s'il à la portée pour mettre un coup physique ou s'il doit lancer un projectile.
À noter que si le joueur 1 est à portée d'un coup physique, le bot a 20% de chance d'attaquer avec le poing et 20% d'attaquer avec le pied.
C'est `aPortee` qui calcule si l'adversaire est dans la portée du bot.
Enfin, `estAuDessusOuDessous` permet de calculer si un joueur est au-dessus d'un autre, ce qui permet d'interdire les actions qui modifient les tailles des hitbox comme s'accroupir notamment.

5.4 Projectiles

Chaque combattant se voit maintenant attribuer une technique `Lancer` permettant de lancer un projectile vers la gauche ou la droite. Un combattant ne peut avoir qu'un projectile lui appartenant sur le terrain à la fois.
Pour implémenter cette extension, nous avons tout d'abord rajouté `Lancer` comme constructeur du type `Technique`. De plus, nous avons ajouté un nouveau type Projectile qui permet d'identifier un projectile par son propriétaire, sa hitbox, sa puissance et son mouvement. Nous avons ainsi pu ajouter une liste de projectile dans `Jeu`.
La fonction `ajouteProjectile` (accompagnée de `prop_pre_ajouteProjectile` et `prop_post_ajouteProjectile`) qui est appelée dans `gameStep` lorsque le joueur 1 appuit sur F ou que le joueur 2 appuit sur P, permet d'ajouter un projectile au jeu.
Elle vérifie tout d'abord que le joueur peut changer sa posture pour lancer son projectile, c'est-à-dire que le changement de taille de sa hitbox lors du lancer ne va pas lui causer une sortie des limites du terrain. Elle s'assure également que le joueur ne possède pas d'autre projectile sur le terrain.
Si ces deux conditions sont bien respectées, `ajouteProjectile` retourne alors le jeu avec sa liste de projectiles contenant le nouveau. Concernant la partie graphique, nous avons rajouté dans `Main.hs` une méthode `loadProjectile` permettant de charger les images au début du jeu avec toutes les autres.

6 Mise en place de tests

Cette partie vise à détailler la mise en place des tests du projet. Ces derniers se situent dans le module `ModelSpec.hs` et sont appelés dans `Spec.hs`.

6.1 Les générateurs

Générateur	Type de données	Caractéristiques
`genTechnique`	`Gen Technique` (instancie `Arbitrary`)	— 40 % de chance de générer la Technique `Rien` — 10 % de chance de générerer un `CoupDePied` — 10 % de chance de générerer un `CoupDePoing` — 10 % de chance de générerer la technique `Protection` — 10 % de chance de générerer la technique `Accroupi` — 10 % de chance de générerer la technique `Encaisse` — 10 % de chance de générerer la technique `Lancer`
`genDirection`	`Gen Direction` (instancie `Arbitrary`)	— 50 % de chance de générer la direction `D` — 50 % de chance de générer la direction `G`
`genCombattant1`	`Gen Combattant`	Génère un `Combattant` en passant en paramètre à `initJoueur1` un `x`, un `y`, des `pv`, une `Technique`, une `Direction`, un `hitTimer` et un `hitStun` générés aléatoirement. La `largeur` du joueur vaut 64 et sa hauteur 119.
`genCombattant2`	`Gen Combattant`	Génère un `Combattant` en passant en paramètre à `initJoueur2` un `x`, un `y`, des `pv`, une `Technique`, une `Direction`, un `hitTimer` et un `hitStun` générés aléatoirement. La `largeur` du joueur vaut 63 et sa hauteur 129.
`genJeuEnCours`	`Gen Jeu`	Génère un `Jeu` en utilisant les deux générateurs de `Combattant`, une zone de 640 sur 480, ainsi qu'une liste de projectiles vide.
`genJeu`	`Gen Jeu` (instancie `Arbitrary`)	— 10 % de chance de générer un `GameOver gagnant` avec `gagnant` un nombre tiré parmi {1,2} de manière aléatoire. — 90 % de chance de tirer un `EnCours` généré par `genJeuEnCours`
`genKeycode`	`Gen KeyCode` (instancie `Arbitrary`)	Générateur de touches utilisées par le jeu générées comme suit : — 50 % de chance pour les touches [Q] [D] [←] [→] — 5 % de chance pour les touches [Z] [S] [↑] [↓] — 33 % de chance pour les touches [A] [E] [F] [P] [↲] [⇧] (droite)

Table 2 – Caractéristiques des générateurs

Afin d'utiliser le puissant outil `quickCheck` d'**Haskell**, des générateurs ont été codés. Les caractéristiques de ces derniers sont détaillées dans le tableau ci-dessus. Il est également à noter que pour que

l'instanciation de `Arbitrary Zone` se limite à un générateur qui génère toujours une `Zone` de 640 par 480. De plus, pour instancier `Arbitrary Combattant`, il a été choisi d'utiliser les deux générateurs de `Combattant` ci-dessus à fréquences égales.

6.2 Description des tests

6.2.1 coordSpec

- Vérifie que déplacer la coordonnée (50,50) de 10 dans une `Direction` puis à l'opposé d'une distance de 10 revient à ne rien faire.
- Vérifie que l'utilisation de `bougeCoordSafe` sur la coordonnée (50,50) a gauche avec un déplacement de 10 permet d'obtenir `Maybe (Coord 40 50)`
- Vérifie que l'utilisation de `bougeCoordSafe` sur la coordonnée (50,50) a gauche avec un déplacement de 51 interdit bien le déplacement en renvoyant `Nothing`

6.2.2 zoneSpec

S'assure que la zone de 640 par 480 vérifie bien son invariant.

6.2.3 combattantSpec

- Vérifie à l'aide d'un `quickCheck` que la propriété sur l'invariant d'un `Combattant`.
- Vérifie à l'aide d'un `quickCheck` que la fonction `actualiserFace` respecte bien sa post-condition quelque soit la `Direction` passée en paramètre.
- Vérifie à l'aide d'un `quickCheck` que les points de vie d'un `Combattant` sont toujours situés entre 0 et 100.

6.2.4 jeuSpec

- S'assure à l'aide d'un `quickCheck` que l'invariant de `Jeu` est respecté.
- S'assure à l'aide d'un `quickCheck` que l'invariant de `Jeu` est respecté après l'utilisation de `gameStep`
- S'assure à l'aide d'un `quickCheck` que l'invariant de `Jeu` est respecté après l'utilisation de `gameStepRandom`

Références

[1] Image du terrain. `https://www.spriters-resource.com/genesis_32x_scd/vrfighterstaken2/sheet/34570/`.

[2] Images du joueur 1. `https://www.spriters-resource.com/pc_computer/dungeonfighteronline/sheet/169383/`.

[3] Images du joueur 2. `https://www.spriters-resource.com/pc_computer/dungeonfighteronline/sheet/169873/`.