# CPS - Projet StreetFighter

Aymeric ROBINI
Marco PONTI

Mai 2017

## 1 Introduction

Dans le cadre de l'UE CPS, nous nous proposons pour ce projet de spécifier et développer un jeu de type Street Fighter. Nous avons décomposé notre application en différents services ; ce document propose un mode d'emploi pour celle ci, ainsi qu'une spécification détaillée de ces derniers. Nous fournissons en annexe une implémentation, des contrats, ainsi qu'une série de tests pour ces services.

## 2 Installation, lancement du jeu

Supposons que l'archive du jeu s'appelle *street.jar*.
Executez :

```
> mkdir street-fighter
> unzip -d street-fighter street.jar
> cd street-fighter
> ant
```

## 3 Jouer au jeu

| | Player 1 | Player 2 |
|---|---|---|
| Déplacement à droite | RIGHT | D |
| Déplacement à gauche | LEFT | Q |
| Saut | UP | Z |
| Accroupissement | DOWN | S |
| Protection | NUMPAD3 | LSHIFT |
| Attaque droite | NUMPADENTER | SPACE |
| Attaque basse | NUMPADENTER+DOWN | SPACE+S |
| Attaque haute | NUMPADENTER+UP | SPACE+Z |

Un changement de couleur d'un personnage indique qu'on entre (ou qu'on quitte) en mode "garde".

Lorsqu'un coup est porté, la hitbox du coup est d'abord representée en gris (phase "startup"), puis en vert (phase "hit", c'est à ce moment que la collision a un impact), puis de nouveau en gris (phase "recovery").

# 4 Specifications

## 4.1 Engine

### 4.1.1 Besoins

Dans le cadre de ce projet nous souhaitons spécifier un service Engine. Celui doit être capable de :

- Gérer la fin du jeu (game over)
- Remettre en place deux personnages qui se tournent le dos
- Faire avancer les deux personnages (step)
- Placer correctement les joueurs en début de partie

### 4.1.2 Specification

**Service**: Engine
**Types**: CommandData
**Observators**:

const height : [Engine] → int
const width : [Engine] → int
char : [Engine] × int → [Character]
char(E, i) requires $i \in \{1, 2\}$
player : [Engine] × int → [Player]
player(E, i) requires $i \in \{1, 2\}$
gameOver : [Engine] → bool

**Constructor**:

init : int × int × int × Player × Player × Character × Character → [Engine]
init(h,w,s,p1,p2,c1,c2) requires $h > 0 \wedge w > s \wedge s > 0 \wedge p1 \neq p2 \wedge c1 \neq c2$

**Operators**:

step : [Engine] → [Engine]
step(E) requires ¬ gameOver(E)

**Observation**:

invariant : gameOver(E) = $\exists i \in \{1, 2\}$ Character ::dead(char(E, i))
init :
height(init(h, w, s, p1, p2, c1, c2)) = h

width(init(h, w, s, p1, p2, c1, c2)) = w
player(init(h, w, s, p1, p2, c1, c2), 1) = p1
player(init(h, w, s, p1, p2, c1, c2), 2) = p2
char(init(h, w, s, p1, p2, c1, c2), 1) = c1
char(init(h, w, s, p1, p2, c1, c2), 2) = c2
Character ::positionX(c1) = w//2 - s//2
Character ::positionX(c2) = w//2 + s//2
Character ::positionY(c1) = 0
Character ::positionY(c2) = 0
Character ::faceRight(c1)
¬Character ::faceRight(c2)

step :

char(step(E), 1) = step(char(E, 1), Player::getCommand(player(E,1))
char(step(E), 2) = step(char(E, 2), Player::getCommand(player(E,2))

(C::posX(char(E,1)) < C::posX(char(E,2)) ≠
C::posX(char(step(E), 1))) < C::posX(char(step(E), 2))) ⇒
(C::faceRight(char(step(E),1)) = ¬
C::faceRight(char(E,1)) ∧
(C::faceRight(char(step(E),2)) = ¬
C::faceRight(char(E,2))

## 4.2 Hitbox

### 4.2.1 Besoins

Nous souhaitons un service capable de gérer une hitbox, nous souhaitons ce que celui ci ai les propriétés suivantes :

- Capable de détecter si un point est dans cette hitbox (belongsTo)
- Capable de détecter la collision avec une autre hitbox (collidesWith)
- Capable de tester l'égalité avec une autre hitbox (equals)

De plus, nous souhaitons que ce service soit le plus général possible, de telle sorte que nous puissons le raffiner pour obtenir des hitboxes plus précises (ex : circulaire, rectangulaire, ensemble de hitboxes, etc).

### 4.2.2 Spécifications

**Service**: Hitbox
**Types**: bool, int
**Observators**:

PositionX: [Hitbox] → int
PositionY: [Hitbox] → int
BelongsTo: [Hitbox] × int × int → bool

CollidesWith: [Hitbox] × Hitbox → bool
EqualsTo: [Hitbox] × Hitbox → bool

**Constructor**:

init: int × int → [Hitbox]

**Operators**:

MoveTo: [Hitbox] × int × int → [Hitbox]

**Observation**:

invariant:

CollidesWith(H,H1) = ∃ x,y:int × int,
BelongsTo(H,x,y) ∧ BelongsTo(H1,x,y)

EqualsTo(H,H1) = ∀ x,y:int × int,
BelongsTo(H,x,y) = BelongsTo(H1,x,y)

init:

PositionX(init(x,y)) = x
PositionY(init(x,y)) = y

MoveTo:

PositionX(MoveTo(H,x,y)) = x
PositionY(MoveTo(H,x,y)) = y
∀ u,v:int × int, BelongsTo(MoveTo(H,x,y),u,v) =
Belongsto(H,u-(x-PositionX(H)),v-(y-PositionY(H))

## 4.3 HitboxRect

### 4.3.1 Besoins

Nous désirons un service capable de gérer des Hitboxes rectangulaires (celles ci étant à la fois simples à mettre en place, rapides à calculer, et plutôt précises pour nos besoins), et qui possède les mêmes propriétés que notre service Hitbox.

### 4.3.2 Specification

**Service**: HitboxRect
**Types**: bool, int
**Observators**:

Width: [Hitbox] → int
Height: [Hitbox] → int

**Constructor**:

init: int × int × int × int → [HitboxRect]

**Operators**:
**Observation**:

invariant:

init:

PositionX(init(x,y,w,h)) = x
PositionY(init(x,y,w,h)) = y
Width(init(x,y,w,h)) = w
Height(init(x,y,w,h)) = h

resize:

Height(resize(H, w, h)) = h
Width(resize(H, w, h)) = w
PositionX(resize(H, w, h)) = PositionX(H)
PositionY(resize(H, w, h)) = PositionY(H)

belongsTo:

$\forall$ (x,y) , x $\leq$ PositionX(H) $\wedge$ x $\geqslant$ PositionX(H) + Width(H)
$\wedge$ y $\leq$ PositionY(H) $\wedge$ y $\geq$ PositionY(H) + Height(H)
$\Leftrightarrow$ belongsTo(H,x,y)

equalsTo:

equalsTo(H,H2) $\Leftrightarrow$
PositionX(H) = PositionX(H2)
$\wedge$ PositionY(H) = PositionY(H2)
$\wedge$ Width(H) = Width(H2)
$\wedge$ Height(H) = Height(H2)

collidesWith:

CollidesWith(H,H2) $\Leftrightarrow$
PositionX(H) < PositionX(H2) + Width(H2)
$\wedge$ PositionY(H) < PositionY(H2) + Height(H2)
$\wedge$ PositionX(H) + Width(H) > PositionX(H2)
$\wedge$ PositionY(H) + Height(H) > PositionY(H2)

## 4.4 Character

### 4.4.1 Besoins

Nous avons besoin d'un service capable de gérer un personnage de jeu, plus précisement, chacune des actions qu'il peut effectuer, en respectant les contraintes suivantes :

- Deux personnages, ne peuvent entrer en collision
- Un personnage peut sauter, dans différentes directions
- Un personnage meurt si sa vie est négative
- Un personnage qui saute ne peut effectuer aucune autre action

Pour garder plus de liberté sur l'implémentation, nous omettons volontairement de spécifier certains points, notemment les positions exactes des déplacements. Cela nous permet par exemple d'adapter la physique du jeu, de faire "bump" les personnages lors d'une collision, etc.

### 4.4.2 Spécification

**Service**: Character
**Types**: bool, int, Commande
**Observators**:

positionX: [Character] → int
positionY: [Character] → int
engine: [Character] → Engine
charBox: [Character] → Hitbox
life: [Character] → int
const speed: [Character] → int
faceRight: [Character] → bool
dead: [Character] → bool
isCrouch: [Character] → bool
isJumpHigh: [Character] → bool
isJumpRightHigh: [Character] → bool
isJumpLeftHigh: [Character] → bool
isJumping: [Character] → bool
const maxY: [Character] → int

**Constructor**:

init: int × int × bool × Engine → [Character]
pre init(l,s,f,e) requires $l > 0 \wedge s > 0$

**Operators**:

moveLeft: [Character] → [Character]
moveRight: [Character] → [Character]
switchSide: [Character] → [Character]
step: [Character] × Commande → [Character]
pre step() requires ¬dead
damaged: [Character] × int → [Character]
crouch: [Character] → [Character]
rise: [Character] → [Character]
jump: [Character] → [Character]
jumpright: [Character] → [Character]
jumpleft: [Character] → [Character]

**Observations**:

invariant :
¬ Hitbox::collideswith(charbox(Engine::char(1)),charbox(Engine::char()))
positionX(C) > 0 ∧ positionX(C) < Engine::width(engine(C))
positionY(C) > 0 ∧ positionY(C) < Engine::height(engine(C))
isJumping(C) = isJumpHigh(C) ∨ isJumpRightHigh(C) ∨ isJumpLeft-High(C)

Toutes ces observations sont vraies si et seulement s'il n'y aurait pas de collision après ces actions. Sinon, elles ne font rien.

moveLeft:

$\neg$ isJumping(C) $\Rightarrow$ positionX(moveLeft(C)) = max(positionX(C)-speed, 0)

isJumping(C) $\Rightarrow$ positionX(moveLeft(C)) = positionX(C)

moveRight:

$\neg$ isJumping(C) $\Rightarrow$ positionX(moveRight(C)) = min(positionX(C)+speed, Engine::width(engine(C)))

isJumping(C) $\Rightarrow$ positionX(moveRight(C)) = positionX(C)

switchSide:

faceRight(switchSide(C)) = $\neg$ faceRight(C)

damaged:

life(damaged(C, d)) = life(C) - d

crouch:

$\neg$ isJumping(C) $\wedge\neg$ isCrouched(C) $\Rightarrow$ Hitbox::height(charbox(crouch(C))) = Hitbox::height(charbox(C))- CST

rise:

$\neg$ isJumping(C) $\wedge$ isCrouched(C) $\Rightarrow$ Hitbox::height(charbox(rise(C))) = Hitbox::height(charbox(C))+ CST

jump:

isJumpHigh(jump(C))

jumpright:

isJumpHighRight(jump(C))

jumpleft:

isJumpHighLeft(jump(C))

step:

isJumping(C) $\wedge$ positionY(C) > maxY(C) $\Rightarrow$ $\neg$ isJumping(step(C,_))

$\neg$ isJumping(C) $\wedge$ positionY(C) > 0 $\Rightarrow$ positionY(step(C,_)) = max(positionY(C) - CST, 0)

positionY(C) = 0 $\Rightarrow$

step(C, RIGHT) = moveRight(C)

$\wedge$ step(C, LEFT) = moveLeft(C)

$\wedge$ step(C, JUMP) = jump(C)

$\wedge$ step(C, JUMPRIGHT) = jumpright(C)

$\wedge$ step(C, JUMPLEFT) = jumpleft(C)

$\wedge$ step(C, CROUCH) = crouch(C)

$\wedge$ step(C, RISE) = rise(C)

## 4.5 FightChar

### 4.5.1 Besoins

Nous souhaitons désormais raffiner notre service Character pour obtenir la gestion du combat, c'est à dire :

- Un personnage peut porter des coups
- Si un coup touche un adversaire, l'adversaire est étourdi.
- Si un personnage est étourdi, il ne peut effectuer aucune action.
- Un personnage peut se proteger des coups, pour ne pas être blessé.

#### 4.5.2 Spécification

**Service**: FightChar refines Character
**Types**: bool, int, Commande
**Observators**:

isBlocking: [FightChar] → bool
isBlockstunned: [FightChar] → bool
isHitstunned: [FightChar] → bool
stuned : [FightChar] → bool
isTeching: [FightChar] → bool
tech: [FightChar] → Tech
pre tech(C) requires isTeching(C)
techFrame: [FightChar] → bool
pre techFrame(C) requires isTeching(C)
techHasAlreadyHit: [FightChar] → bool
pre techHasAlreadyHit(C) requires isTeching(C)

**Constructor**:

init: int × int × bool × Engine → [FightChar]
pre init(l,s,f,e) requires $l > 0 \wedge s > 0$

**Operators**:

startTech: [FightChar] × Tech → [FightChar]
pre startTech(C,T) requires ¬isTeching(C)
∧ ¬isHitstunned(C) ∧ ¬isBlockstunned(C)

**Observation**:

invariants :
isBlockstunned(C) ⇒ ¬isHitstunned(C)
isHitstunned(C) ⇒ ¬isBlockstunned(C)
stuned(C) = isBlockstunned(C) ∨ isHitStunned(C)
init :
¬ isBlocking(init(_))
¬ isBlockstunned(init(_))
¬ isHitstunned(init(_))
¬ isTeching(init(_))
moveLeft :
isHitstunned(C) ∨ isBlockstunned(C) ∨ isTeching(C)

⇒ positionX(moveLeft(C)) = positionX(C)

moveRight :

isHitstunned(C) ∨ isBlockstunned(C) ∨ isTeching(C) ⇒ positionX(moveRight(C)) = positionX(C)

Toutes ces observations sont vraies si et seulement s'il n'y aurait pas de collision après ces actions. Sinon, elles ne font rien.

crouch:

¬ stunned(C) ∧¬ teching(C) ⇒ Hitbox::height(charbox(crouch(C))) = Hitbox::height(charbox(C))- CST

rise:

¬ stunned(C) ⇒ Hitbox::height(charbox(rise(C))) = Hitbox::height(charbox(C))+ CST

jump:

¬ stunned(C) ∧¬ teching(C) ⇒ isJumpHigh(jump(C))

jumpright:

¬ stunned(C) ∧¬ teching(C) ⇒ isJumpHighRight(jump(C))

jumpleft:

¬ stunned(C) ∧¬ teching(C) ⇒ isJumpHighLeft(jump(C))

step:

isJumping(C) ∧ positionY(C) > maxY(C) ⇒ ¬ isJumping(step(C,_))
¬ isJumping(C) ∧ positionY(C) > 0 ⇒ positionY(step(C,_)) = max(positionY(C) - CST, 0)
¬ stuned(C) ∧¬ teching(C) ⇒
∧ step(C, PUNCHUP) = startech(C, TechData.punchUp)
∧ step(C, PUNCHDOWN) = startech(C, TechData.punchDown)
∧ step(C, PUNCH) = startech(C, TechData.punch)
∧ step(C, GUARD) = guard(C)

damaged :

isBlocking(C) ⇒ (life(damaged(C,deg,hstun,bstun)) = life(C)) ∧ isBlockstunned(C)
¬isBlocking(C) ⇒ (life(damaged(C,deg,hstun,bstun)) = life(C)-deg) ∧ isHitstunned(C)

## 4.6 Player

### 4.6.1 Besoins

La spécification du service Player nous a semblé superflue. En effet, le Player est celui qui contrôle le Character. Or, tout ce que nous souhaitons pour le player est de pouvoir récuperer des commandes à envoyer au Character, la source de ces commande peut être très variée :

- Un joueur physique (c'est le cas dans notre implémentation)
- Une IA

- Un joueur depuis le réseau
- etc...

Ainsi, il est difficile de spécifier Player sans se poser des barrières pour l'évolutivité de notre application.

#### 4.6.2 Specifications

**Service**: Player
**Types**: bool, int
**Observators**:

command: [Player] $\rightarrow$ CommandData

**Constructor**:

init: int $\rightarrow$ [Player]
init(n) requires n $\in \{1, 2\}$

**Operators**:

getCommand: [Player] $\rightarrow$ CommandData

**Observation**: