2I013 Groupe 1

Projet Foot

Maxime Sangnier – Nicolas Baskiotis
maxime.sangnier@lip6.fr

2019

Laboratoire d'Informatique de Paris 6 (LIP6)
Sorbonne Université

Outline

Séance précédente

Bilan du TME

Vous avez :

- pris en main **git** (`git pull, add, commit -am "MON MESSAGE", push`) ;
- créé un compte **github** et un dépôt pour développer votre code ;
- cloné la plateforme de simulation dans un dossier **différent** ;
- programmé un joueur aléatoire ;
- programmé un joueur fonceur ;
- fonctionne-t-il des **deux côtés** ?

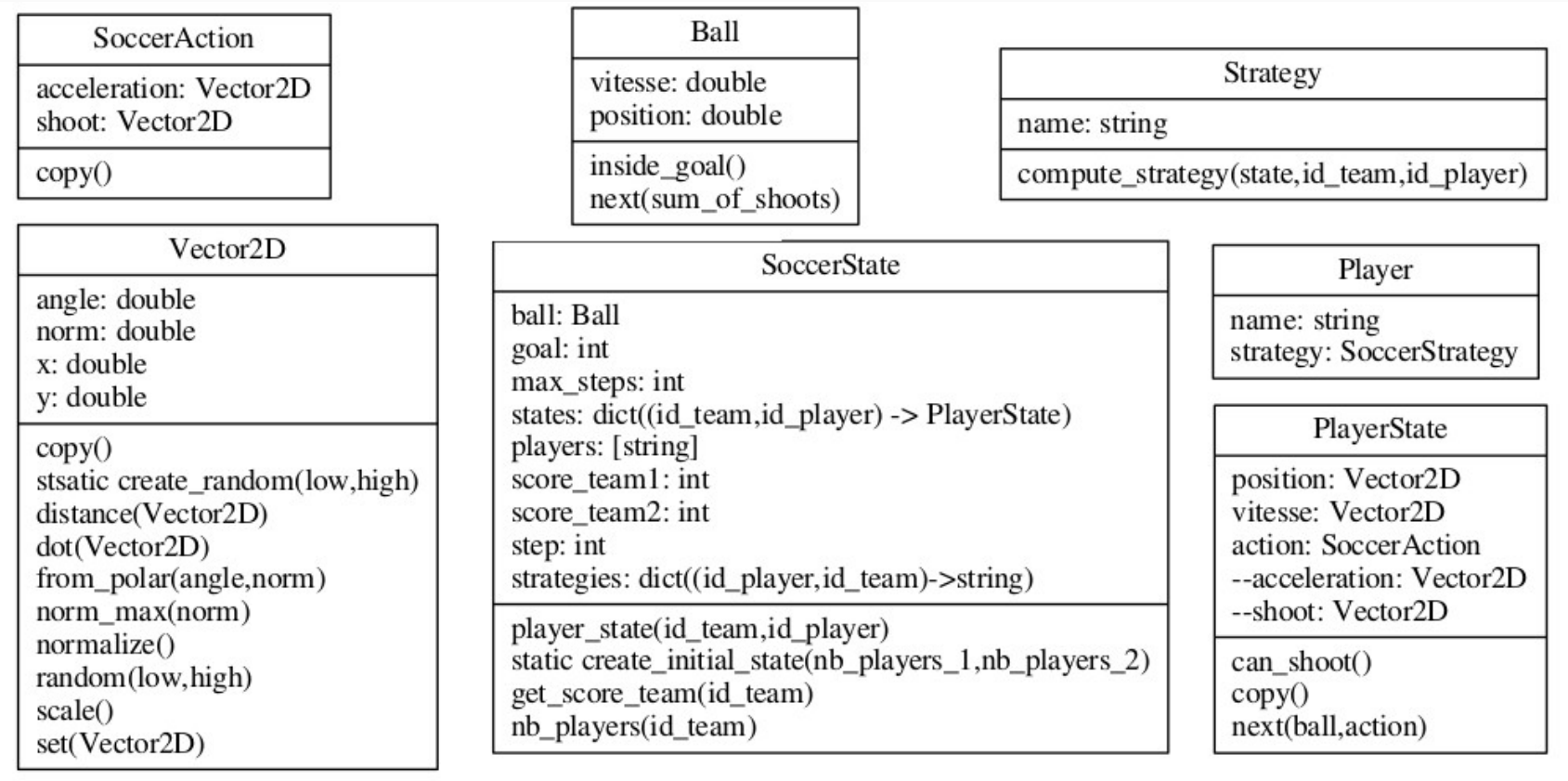
```
class GoStrategy(Strategy):
    def __init__(self):
        Strategy.__init__(self, "Go-getter")

    def compute_strategy(self, state, id_team, id_player):
        ball = state.ball.position
        player = state.player_state(id_team, id_player).position
        goal = ...
        if player.distance(ball) < PLAYER_RADIUS+BALL_RADIUS:
            return SoccerAction(shoot=goal-player)
        else:
            return SoccerAction(acceleration=ball-player)
```

Le simulateur décortiqué

Les objets en présence (et leurs attributs)

- `Vector2D` : représente un point ou un vecteur ;
- `MobileMixin` : représente un objet mobile (position, vitesse) ;
- `SoccerAction` : représente l'action d'un joueur (accélération, shoot) ;
- `Player` : représente un joueur (nom, stratégie) ;
- `Strategy` : représente une stratégie ;
- `PlayerState` : représente un état d'un joueur (position, vitesse)
- `SoccerState` : représente un état du jeu (balle, joueurs, score)
- `SoccerTeam` : liste de joueurs et de leur stratégie
- `Simulation` : une simulation de match

Les objets en présence (et leurs attributs)



Les commandes utiles

Étant donné un état `state`...

La balle

- `state.ball` : MobileMixin de la balle
- `state.ball.position` : la position de la balle (`state.ball.position.x, state.ball.position.y`)
- `state.ball.vitesse` : la vitesse de la balle

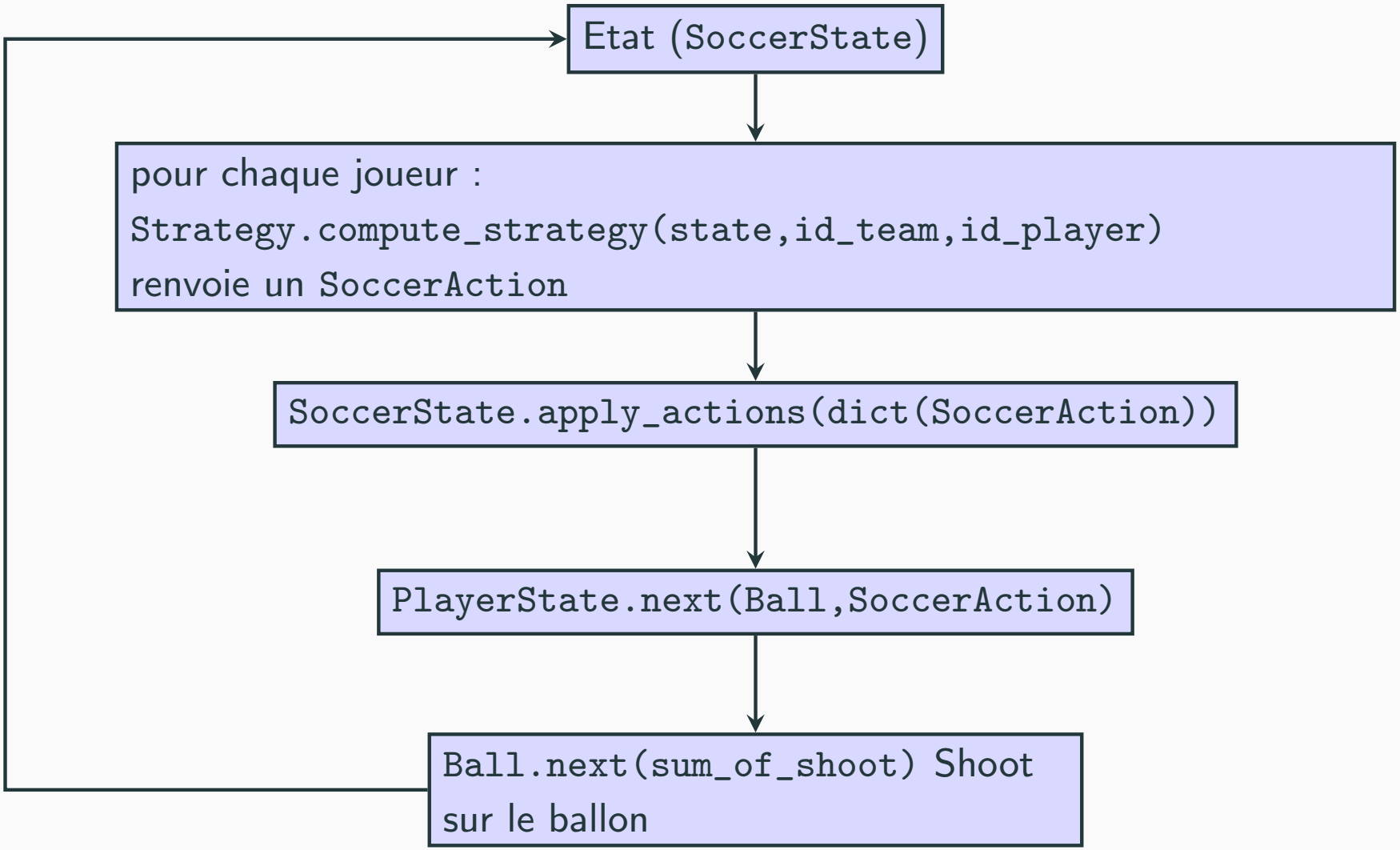
Le joueur

- `state.player_(idteam,idplayer)` : la configuration d'un joueur
- `state.player_state(idteam,idplayer)` : MobileMixin du joueur
- `state.player_state(idteam,idplayer).position` : position du joueur
- `state.player_state(idteam,idplayer).vitesse` : vitesse du joueur

Autre

- `state.players` : liste des clés `(idteam,idplayer)` de tous les joueurs

Le cœur du simulateur



Le moteur d'un joueur

```
def next(self, ball, action=None):
  if not (hasattr(action,"acceleration") and hasattr(action,"shoot")):
    action = SoccerAction()
    self.action = action.copy()
    self.vitesse *= (1 - settings.playerBrackConstant)
    self.vitesse = (self.vitesse +\
        self.acceleration).norm_max(settings.maxPlayerSpeed)
    self.position += self.vitesse
    if self.position.x < 0 or self.position.x > settings.GAME_WIDTH \
      or self.position.y < 0 \
      or self.position.y > settings.GAME_HEIGHT:
    self.position.x=max(0,min(settings.GAME_WIDTH, self.position.x))
    self.position.y=max(0,min(settings.GAME_HEIGHT, self.position.y))
    self.vitesse = Vector2D()
    if self.shoot.norm == 0 or not self.can_shoot():
      self._dec_shoot()
      return Vector2D()
    self._reset_shoot()
    if self.position.distance(ball.position)\
        >(settings.PLAYER_RADIUS+settings.BALL_RADIUS):
        return Vector2D()
    return self._rd_angle(self.shoot,(self.vitesse.angle-self.shoot.angl
        self.position.distance(ball.position)/(settings.PLAYER_RADIUS+se
```

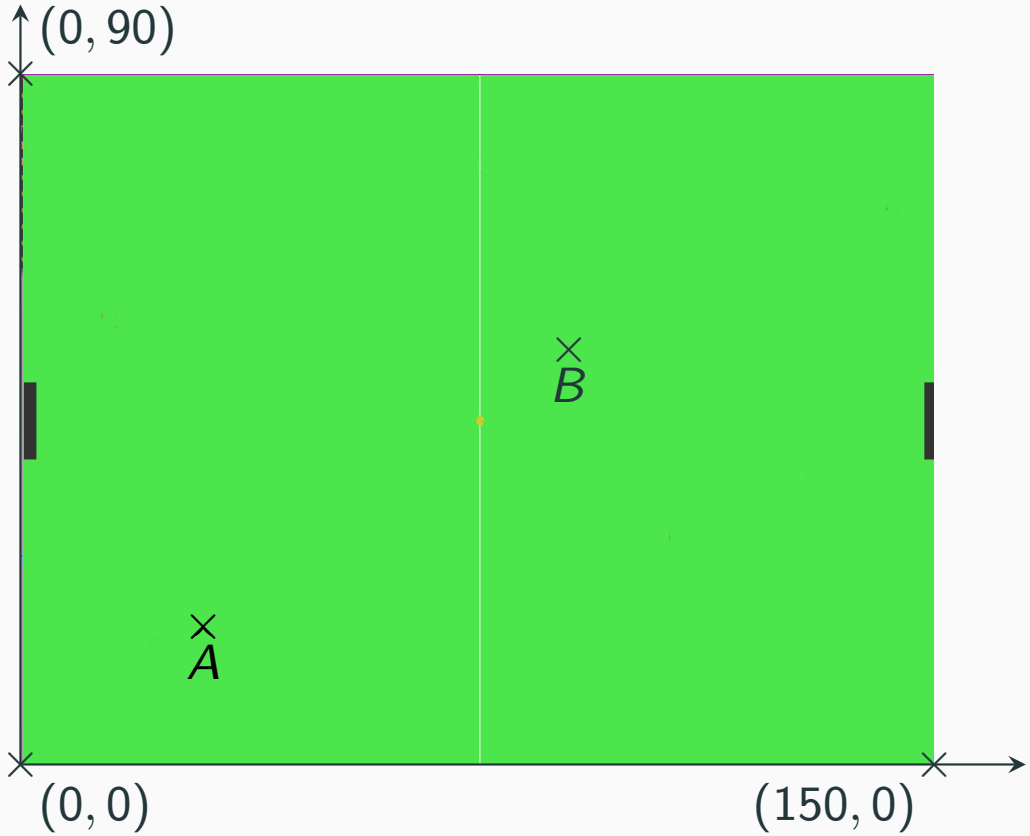
Le moteur du ballon

```
def next(self,sum_of_shoots):
  vitesse = self.vitesse.copy()
  vitesse.norm = self.vitesse.norm -\
                 settings.ballBrakeSquare*\
                 self.vitesse.norm**2-\
                 settings.ballBrakeConstant*self.vitesse.norm
  ## decomposition selon le vecteur unitaire de ball.speed
  snorm = sum_of_shoots.norm
  if snorm > 0:
    u_s = sum_of_shoots.copy()
    u_s.normalize()
    u_t = Vector2D(-u_s.y, u_s.x)
    speed_abs = abs(vitesse.dot(u_s))
    speed_ortho = vitesse.dot(u_t)
    speed_tmp = Vector2D(speed_abs * u_s.x\
       - speed_ortho * u_s.y,\
       speed_abs * u_s.y + speed_ortho * u_s.x)
    speed_tmp += sum_of_shoots
    vitesse = speed_tmp
  self.vitesse = vitesse.norm_max(settings.maxBallAcceleration)
  self.position += self.vitesse
```

Retour sur la stratégie fonceur

Aller vers un point ?

- A : position courante
- B : destination
- Quelle action ?

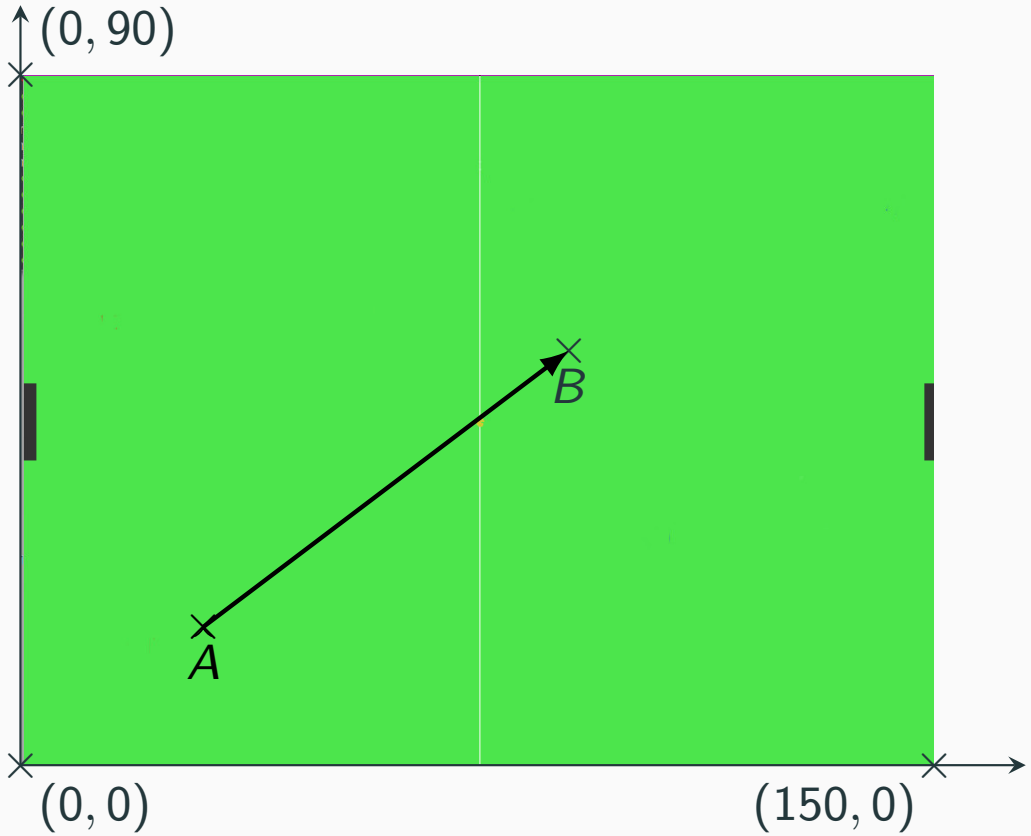


Aller vers un point ?

- A : position courante
- B : destination
- Quelle action ?
- Vecteur vitesse :

$$\overrightarrow{v} = \overrightarrow{AB}$$

- Importance de la norme ?




```
def compute_strategy(self, state, id_team, id_player):
    return SoccerAction(state.ball.position
                -state.player_state(id_team,id_player).position,
                Vector2D())
```

Proposition de stratégie

Stratégies naïves

- Fonceur
- Gardien / défenseur
- ...

Proposition de stratégie

Stratégies naïves

- Fonceur
- Gardien / défenseur
- ...
- Comment choisir entre les différentes stratégies ?
- Comment le faire de manière élégante ?

$\Rightarrow$ Coder des petites fonctions légères et génériques !

Vers une bonne organisation du code

Création d'une Toolbox

Dans un fichier séparé (par exemple `tools.py`)

Inclure les fonctions usuelles pour :

- aller vers un point
- shooter vers le but
- trouver l'adversaire le plus proche
- ... (toutes les petites fonctions récurrentes dont vous aurez besoin)

Puis dans votre fichier

```
from tools import *
...
```

Réfléchissez à la structure de vos fonctions :

- Elles doivent être générique (situation miroir selon l'identifiant de l'équipe)
- Facile à manier.
- Possibilité d'encapsuler l'objet `SoccerState`.

Encapsuler un objet

Il s'agit

- d'enrichir un objet de nouvelles fonctionnalités ;
- de *traduire* certaines de ses propriétés (par exemple objet miroir)
- d'en faciliter l'utilisation.

Exemple

```
class SuperState(object):
  def __init__(self, state, id_team, id_player):
    self.state = state
    self.id_team = id_team
    self.id_player = id_player

  @property
  def ball(self):
    return self.state.ball.position

  @property
  def player(self):
    return self.state.player_state(self.id_team, self.id_player).position

  @property
  def goal(self):
```

Encapsuler un objet

Il s'agit

- d'enrichir un objet de nouvelles fonctionnalités ;
- de *traduire* certaines de ses propriétés (par exemple objet miroir)
- d'en faciliter l'utilisation.

Exemple

```
class GoStrategy(Strategy):
def __init__(self):
  Strategy.__init__(self, "Go-getter")

def compute_strategy(self, state, id_team, id_player):
  s = SuperState(state, id_team, id_player)
  if s.player.distance(s.ball) < PLAYER_RADIUS+BALL_RADIUS:
   return SoccerAction(shoot=s.goal-s.player)
  else:
    return SoccerAction(acceleration=s.ball-s.player)
```

Utiliser l'héritage

Il s'agit (toujours)

- de faciliter l'utilisation d'un objet (ici `Strategy`).

Différences avec l'encapsulation

- on hérite **naturellement** des attributs et des méthodes de la classe parente ;
- on doit construire **proprement** l'instance de l'objet parent.

Exemple

```
class SimpleStrategy(Strategy):
  def __init__(self, action, name):
    super().__init__(name)
    self.action = action

  def compute_strategy(self, state, id_team, id_player):
    s = SuperState(state, id_team, id_player)
    return self.action(s)
```

Utiliser l'héritage

Il s'agit (toujours)

- de faciliter l'utilisation d'un objet (ici `Strategy`).

Différences avec l'encapsulation

- on hérite **naturellement** des attributs et des méthodes de la classe parente ;
- on doit construire **proprement** l'instance de l'objet parent.

Exemple

```
def gobetter(state):
  if state.player.distance(state.ball) < PLAYER_RADIUS + BALL_RADIUS:
   return SoccerAction(shoot=state.goal - state.player)
  else:
   return SoccerAction(acceleration=state.ball - state.player)

team2.add("Go", SimpleStrategy(gobetter, 'Go-better'))
```

Création des équipes

Création des équipes

Dans la vraie vie

- discutez avec vos camarades ;
- mettez-vous d'accord sur des binômes ;
- ces binômes perdureront tout le semestre et seront utilisés pour l'évaluation (rapport, soutenance, code).

Sur github

- choisissez un compte **github** ;
- partagez votre dépôt avec l'autre ;
- renseignez le formulaire : `https://goo.gl/forms/kL0dERD8L0pEpGwu1`

Objectifs du TME

Objectifs du TME

- organiser son code à l'aide d'une toolbox et de scripts ;
- définir et enrichir un objet `SuperState` ;
- affiner le joueur fonceur pour en faire un joueur attaquant qui :
  - tire correctement ;
  - sait faire face à un fonceur ;
- programmer un joueur défenseur.

A chaque TME

Mettre à jour le dépôt contenant le simulateur :

```
cd [DOSSIER DU SIMULATEUR]
git pull
```