CS-107 : Mini-projet 2
Jeux sur grille : « ICWars »

S. Abuzakuk, J. Sam, B. Jobstmann

Version 1.5

Table des matières

1 Présentation

> Ce document utilise des couleurs et contient des liens cliquables (textes soulignés). Il est préférable de le visualiser en format numérique.

Vous vous êtes familiarisés ces dernières semaines avec les fondamentaux d'un petit moteur de jeux adhoc (voir tutoriel) vous permettant de créer des jeux sur grille en deux dimensions. L'ébauche simple obtenue s'apparente à ce que l'on peut trouver dans un jeu de type RPG. Le but de ce mini-projet est d'en tirer parti pour créer des déclinaisons concrètes d'un autre type de jeu. Le jeu de base qu'il vous sera demandé de créer est fortement inspiré du célèbre Advance Wars. La figure 1 montre un exemple de l'ébauche de base[1] que vous pourrez enrichir ensuite à votre guise, au gré de votre fantaisie et imagination.

Outre son aspect ludique, ce mini-projet vous permettra de mettre en pratique de façon naturelle les concepts fondamentaux de l'orienté-objet. Il vous permettra d'expérimenter le fait qu'une conception située à un niveau d'abstraction adéquat permet de produire des programmes facilement extensibles et adaptables à différents contextes.

Vous aurez concrètement à complexifier, étape par étape, les fonctionnalités souhaitées ainsi que les interactions entre composants.

FIG. 1 : Exemple d'un niveau du jeu où le « joueur » (curseur blanc) se déplace afin de sélectionner des unités pour les repositionner et/ou leur faire réaliser des actions en vue de battre l'adversaire.

[1] voir la vidéo de démonstration

Le projet comporte quatre étapes :

- Étape 1 (« Composants de base ») : au terme de cette étape vous aurez créé, en utilisant les outils du moteur de jeu fourni, une instance basique de « ICWars » avec un acteur sous forme de curseur capable de se déplacer et de changer de niveau. Vous aurez aussi créé des acteurs « Unité » qui pourront par la suite être utilisés par le joueur. Dans ce qui suit, nous appellerons « acteur principal » le curseur contrôlé par le joueur.
- Étape 2 (« Interactions avec les unités ») : lors de cette étape il vous sera demandé de faire interagir l'acteur principal de façon basique avec les unités pour les déplacer.
- Étape 3 (« Actions et IA ») : lors de cette étape il vous sera demandé de mettre en place les actions réalisables par les unités. Votre acteur principal devra également pouvoir affronter un joueur virtuel doté d'une « intelligence artificielle » basique. Le moteur de jeu sera enrichi de fonctionnalités permettant l'affichage de graphismes statiques dans le but de donner des informations sur l'état du jeu.
- Étape 4 (Extensions) : durant cette étape, diverses extensions plus libres vous seront proposées et vous pourrez enrichir à votre façon le jeu créé à l'étape précédente ou en créer d'autres.

Coder quelques extensions (à choix) vous permet de gagner des points bonus et/ou de valoriser votre projet pour participer au concours.

Voici les consignes/indications principales à observer pour le codage du projet :

1. Le projet sera codé avec les outils Java standard (import commençant par `java.` ou `javax.`). Si vous avez des doutes sur l'utilisation de telle ou telle librairie, posez-nous la question et surtout faites attention aux alternatives que Eclipse vous propose d'importer sur votre machine. Le projet utilise notamment la classe `Color`. Il faut utiliser la version `java.awt.Color` et non pas d'autres implémentations provenant de divers packages alternatifs.
2. Vos méthodes seront documentées selon les standard javadoc (inspirez-vous du code fourni). Votre code devra respecter les conventions usuelles de nommage et être bien **modularisé et encapsulé**. En particulier, les getters intrusifs, publiquement accessibles, sur des objets modifiables seront à éviter.
3. Les indications peuvent être parfois très détaillées. **Cela ne veut pas dire pour autant qu'elles soient exhaustives**. Les méthodes et attributs nécessaires à la réalisation des traitements voulus ne sont évidemment pas tous décrits et ce sera à vous de les introduire selon ce qui vous semble pertinent et en respectant une bonne encapsulation.
4. Votre projet **ne doit pas être stocké sur un dépôt public** (de type github). Pour ceux d'entre vous familier avec git, nous recommandons l'utilisation de GitLab : https ://gitlab.epfl.ch/, mais tout type de dépôt est acceptable pour peu qu'il soit privé.

2 Composants de base (étape 1)

Le but de cette étape est de commencer à créer votre propre petit jeu sur grille qui prendra la forme d'une variation simplifiée du jeu « Advance Wars »®(https://en.wikipedia.org/wiki/Advance_Wars).

La version de base à produire lors de cette étape contiendra un acteur principal prenant la forme (très épurée) d'un simple curseur déplaçable et qui pourra transiter sur différentes aires (niveaux). Vous introduirez également la notion d'« unité » utilisable par les joueurs. L'interaction avec ces unités et leur utilisation concrète ne seront abordées qu'à l'étape suivante.

Vous coderez le projet dans le paquetage fourni `ch.epfl.cs107.play.game.icwars`, désigné de façon raccourcie dans l'énoncé comme `game.icwars`.

2.1 Préparation du jeu `ICWars`

Tout d'abord, en utilisant une logistique analogue à celle de nos précédentes ébauches de jeu, nous allons complètement changer de décor ! Il est conseillé pour cela de partir de la version du corrigé (fourni dans la page de description du projet).

Préparez un jeu `ICWars` en vous inspirant de `Tuto2`. Ce dernier sera constitué pour commencer des composants suivants :

- La classe `RealPlayer` qui modélise le joueur « humain » (équivalent de l'acteur principal ébauché dans le tutoriel) et qui sera contrôlable manuellement ; il est à placer dans `game.icwars.actor.players` ; laissez cette classe vide pour le moment ; nous y reviendrons un peu plus bas.
- La classe `ICWars`, équivalente à `Tuto2` à placer dans le paquetage `game.icwars` la méthode `update` de `ICWars` se contentera pour le moment d'appeler celle de sa superclasse. N'oubliez pas d'adapter la méthode `getTitle()` qui retournera un nom de votre choix (par exemple *`"ICWars"`*;-)) ; Les aires spécifiques du jeu `ICWars` sont décrites un peu plus bas.
- La classe `ICWarsArea` équivalente à `Tuto2Area`, à placer dans un sous-paquetage `game.icwars.area`. Vous associerez un facteur d'échelle commun à toutes les aires de ce type (`10.f` par exemple ; augmentez cette valeur pour voir de plus grandes parties des aires).
- Les classes `Level0` et `Level1`, héritant de `ICWarsArea`, à placer dans le paquetage `game.icwars.area` (elles sont équivalentes aux classes `Ferme`, et `Village` de `game.tutos.area.tuto2`) ; les intitulés associés seront respectivement *`"icwars/Level0"`* et *`"icwars/Level1"`* ;
- de la classe `ICWarsBehavior` analogue à `Tuto2Behavior` à placer dans `game.icwars.area` et qui contiendra une classe publique `ICWarsCell` équivalente de `Tuto2Cell` (les adaptations nécessaires sont décrites ci-dessous).

2.1.1 Adaptation de `ICwarsBehavior`

Dans l'esprit, `ICwarsBehavior` et `ICWarsCell` sont équivalentes à `Tuto2Behavior` et `Tuto2Cell`. Néanmoins, contrairement à l'ébauche de jeu du tutoriel, la nature du « décor » n'est pas ce qui va conditionner le déplacement de l'acteur principal (curseur). Dans le cas de ce nouveau type de jeu, entrer dans les cellules de la grille se fera indépendamment de leur nature et la seule chose qui pourra entraver le déplacement est la présence d'un autre acteur qui ne se laisserait pas « marcher dessus ».

Concrètement, la méthode `canEnter` des `ICWarsCell` retournera faux si l'entité qui souhaite y entrer n'est pas traversable et que la cellule contient d'autres acteurs non traversables (« traversable » veut dire que la méthode `takeCellSpace` retourne `false`). Dans tous les autres cas, la cellule donnera son feu vert à l'entité qui souhaite y entrer.

Si la nature des cellules ne joue pas de rôle dans le déplacement, elle sera néanmoins impactante pour les choix de déplacement : à chaque type de cellule sera en effet associé un certain « nombre d'étoiles de défense » qui indiquent la protection offerte par la cellule (la mise en oeuvre de cette protection sera expliqué en temps voulu). Le type énuméré décrivant le types des cellules et le nombre d'étoiles associé à chaque type pourra donc être décrit comme suit :

```
NONE(0,0),              // Should never be used except
                        // in the toType method
ROAD(-16777216, 0),     // the second value is the number
                        // of defense stars
PLAIN(-14112955, 1),
WOOD(-65536, 3),
RIVER(-16776961, 0),
MOUNTAIN(-256, 4),
CITY(-1,2);
```

2.2 Acteurs de « ICwars » et unités

> Pour programmer cette partie, il est important de bien lire l'énoncé dans son intégralité (ne codez pas linéairement au fur et à mesure que vous lisez). Il est important de bien réfléchir à où placer les attributs et méthodes nécessaires en fonction des liens d'héritage.

Maintenant que les bases fondamentales sont posées, il vous est demandé de modéliser quelques acteurs essentiels des jeux de type `ICWars`. Ils seront codés dans le sous-paquetage `icwars.actor`.

Vous considérerez que les acteurs impliqués dans un jeu `ICWars`, les `ICWarsActor`, sont capables de se déplacer sur une grille (`MoveableAreaEntity`) et qu'à ce niveau d'abstraction, ils n'évoluent pas de façon spécifique. Ils appartiennent par ailleurs nécessairement à une *faction* qui peut être *"alliée"* ou *"ennemie"* (un type énuméré est conseillé).

La faction de départ d'un `ICWarsActor` est donnée à sa construction, tout comme sa position et l'aire à laquelle il appartient. Il sera orienté par défaut vers le haut à la construction . Il sera aussi capable d'entrer dans une aire et de quitter son aire à la manière du `GhostPlayer`. La seule différence et qu'il n'a pas forcément besoin de voir la caméra se centrer sur lui lorsqu'il entre dans une aire (les appels à `setViewCandidate` et `resetMotion` ne lui sont pas utiles à ce niveau de généralité). Enfin, les cellules qu'il occupe se définiront aussi comme pour le `GhostPlayer` (méthode `getCurrentCells()`) .

Plusieurs sous-classes de `ICWarsActor` seront à définir, parmi lesquelles celles représentant ce que l'on va appeler par la suite les « joueurs ». Vous vous occuperez à cette étape du joueur « humain » ébauché au travers de la classe `RealPlayer` et qui sera complété en partie un peu plus bas. Le joueur artificiel (automatique, `AIPlayer`) qui affrontera le joueur humain sera quant à lui codé lors d'une prochaine étape.

Chacun des joueurs va se servir d'*unités de combat* qui sont elles-mêmes des `ICWarsActor`. Elles lui permettent d'affronter son adversaire. Dans ce qui suit, vous trouverez des indications pour commencer à coder la notion d'unité ainsi que le joueur humain (`RealPlayer`).

2.2.1 Unités

Les unités sont des `ICWarsActor` et appartiennent de ce fait à une faction.

Une unité (classe `Unit`) a pour particularités :

- d'avoir un *nom* (une chaîne de caractères, accessible via une méthode `getName`) et des *points de vie* ;
  les points de vie ne peuvent pas être négatifs et ils ont une valeur maximale qui dépend du type spécifique de l'unité ; une unité est considérée comme morte si son nombre de points de vie est nul ; ils doivent pouvoir être accessibles via une méthode `getHp` ;
- de pouvoir être dessinée sous la forme d'un sprite : le sprite spécifique à choisir est évidemment dépendant du type concret de l'unité ;
- de pouvoir subir des dommages : cela revient à diminuer ses points de vie d'une certain nombre de points (donnés en paramètre) ;
- de pouvoir se réparer : ce qui revient cette fois à augmenter ses points de vie d'une certain nombre de points (aussi donnés en paramètre) ;
- de pouvoir infliger un nombre fixe de points de dommage à autrui (à chaque attaque) ; le calcul, via une méthode que vous nommerez `getDamage()` de ce nombre dépend du type spécifique de l'unité et ne peut pas être défini concrètement pour une unité quelconque ;
- et de ne pouvoir être déplacée que dans un « rayon » défini par un entier.

Les unités sont par ailleurs des acteurs pour lesquels `takeCellSpace` retourne `true`[2]. Vous considérerez pour le moment qu'elles n'acceptent que les interactions de contact (mais

[2] ils sont non traversables, ce qui veut dire que s'ils occupent une cellule, seuls des acteurs traversables comme le curseur pourront cohabiter avec dans cette cellule

pourrez par la suite leur permettre d'accepter des interactions à distance si vous le souhaitez). Vous introduirez à ce stade deux types spécifiques d'unités : les *soldats* et les *tanks*. Les caractéristiques spécifiques à chaque type d'unité sont décrites dans le tableau ci-dessous :

Caractéristiques	Soldat	Tank
Rayon de déplacement	2	4
Dommages infligés par attaque	2	7
Points de vie max.	5	10

On considère donc à ce stade que le calcul du nombre de dommages infligés revient simplement à retourner un entier spécifique.

Les sprites à utiliser seront intilialisés selon la tournure :

```
new Sprite(nomDuSprite, 1.5f, 1.5f, this, null, new
   Vector(-0.25f, -0.25f))
```

où `nomDuSprite` vaut : *`"icwars/friendlySoldier"`* resp. *`"icwars/friendlyTank"`* pour les soldat resp. tanks alliés et *`"icwars/enemySoldier"`* resp. *`"icwars/enemyTank"`* pour les soldats resp. tanks ennemis.

Contrainte à respecter : le constructeur des sous-classes doit avoir pour seuls paramètres l'aire d'appartenance de l'unité à sa création, sa position et sa faction. Les points de vie seront initialisés à la construction au nombre de points maximaux possible pour l'unité concernée.

2.2.2 Ebauche de joueur générique

Les joueurs peuvent être codés dans le sous-paquetage `game.icwars.actor.players`.

Chacun des joueurs impliqués dans une partie de `ICWars`, qu'il soit humain ou artificiel, doit disposer d'un ensemble d'unités lui permettant d'affronter son adversaire. Il est donc naturel d'envisager une classe commune, `ICWarsPlayer`, regroupant les caractéristiques de tout type de *joueurs*. Un `ICWarsPlayer` est évidemment aussi un `ICWarsActor`. Vous considérerez que les unités dont dispose un `ICWarsPlayer` ainsi que sa faction sont paramétrables à la construction.

Le `ICWarsPlayer` est en charge d'enregistrer ses unités comme acteurs dans son aire à sa création. Il évolue au cours du temps comme tout acteur de jeu sur grille mais en plus, il devra se préoccuper d'éliminer les unités qui ne sont plus opérationnelles : à chaque fois qu'il évolue d'un pas de temps (méthode `update`), il devra vérifier lesquelles parmi ses unités sont détruites et dans ce cas, les supprimer de son ensemble d'unités et les dés-enregistrer de son aire de jeu. Un `ICWarsPlayer` quitte son aire de jeu à la manière d'un `ICWarsActor` mais il doit au préalable dés-enregistrer ses unités de l'aire qu'il quitte.

Comme le `GhostPlayer` du tutoriel, un joueur peut être suivi par la caméra et donc il est utile de le doter d'une méthode `centerCamera()`.

En tant que `Interactable`, `ICWarsPlayer` sera l'objet d'interactions de contact uniquement (il devra entrer en contact de ses unité par exemple pour pouvoir s'en servir). Contrairement au `GhostPlayer` du tutoriel, il sera traversable (`takeCellSpace` retourne `false`, ce qui permettra au joueur d'entrer dans n'importe quelle cellule).

Enfin, il doit être possible de tester si un `ICWarsPlayer` est vaincu. Ce sera le cas lorsque sa liste d'unités est vide (plus d'arme pour combattre).

Indications

- une ellipse peut être utilisée pour fournir une liste variables d'unités au constructeur
- les unités doivent être enregistrées comme acteurs du jeu (pour que leur dessin et mise à jour puissent se faire). Il est aussi utile de les répertorier (dans leur aire d'appartenance, `Area`) dans une liste constituée uniquement d'unités. Cela facilitera les certains traitements ultérieurs en évitant de devoir chercher parmi tous les acteurs du jeu, lesquels sont des unités.

Il n'est pas rare en programmation orientée-objet de référencer des mêmes objets dans différents types de collection. Ceci permet d'avoir un prisme de vue différent en fonction du traitement que l'on veut réaliser. Dans notre cas les unités sont à la fois répertoriées dans la collection hétérogène d'acteurs de l'aire et une autre fois plus spécifiquement dans une collection homogène ne contenant que des unités. Ceci nous permet de les manipuler tantôt comme acteurs génériques (pour le dessin et la mise à jour) , tantôt comme acteurs spécifiques. Attention, ici cela nous astreint à une discipline stricte quant à la cohérence des deux ensembles : une unité enregistrée dans l'aire comme acteur doit apparaître dans l'ensemble spécifique des unités. De même, pour le dés-entregistrement.

2.2.3 Le joueur `RealPlayer`

La classe `RealPlayer`, suggérée plus haut sera codée pour le moment dans le même esprit que `GhostPlayer`. Vous devez cependant réviser les liens d'héritage car il s'agit évidemment aussi d'un `ICWarsPlayer`. L'image associée sera donnée directement dans le corps du constructeur. Il s'agira de *`"icwars/allyCursor"`* si le joueur est de faction amie, et *`"icwars/enemyCursor"`* s'il est ennemi.

Un `RealPlayer` se comporte (se met à jour) comme `ICWarsPlayer` mais en plus il doit pouvoir être déplacé au moyen des flèches directionnelles à la manière du `GhostPlayer` codé dans le tutoriel.

FIG. 2 : de gauche à droite : position initiale du « joueur et placements attendus des acteurs unités dans `Level0` et `Level1`.

2.3 Première ébauche de `ICWars`

Au lancement du jeu, un joueur de type `RealPlayer` sera créé à la position qui lui est destinée dans l'aire de démarrage (prenez $(0, 0)$ dans `Level0` et $(2, 5)$ dans `Level1`). Sa faction sera considérée comme alliée.

Il sera doté de deux unités de sa propre faction : un soldat et un tank. Pour simplifier vous pourrez donner des valeurs « en dur » pour la position de ces unités (par exemple `(2,5)` pour le tank et `(3,5)` pour le soldat). Libre à vous de sophistiquer le positionnement initial des unités et de les rendre dépendante de l'aire dans laquelle le joueur est créé.

Le jeu devra démarrer sur l'aire *`"icwars/Level0"`*.

Vous doterez ce jeu de deux contrôles :

1. la touche `N` doit permettre de passer au niveau suivant et s'il n'y a plus de niveau, décréter la fin de partie (appel de la méthode `end()` qui se contentera d'afficher le message *`"Game Over"`* dans la console) ;
2. la touche `R` doit permettre de faire une réinitialisation (« reset ») du jeu c'est à dire le redémarrer dans les même conditions que la toute première fois qu'on le lance.

2.3.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données.

Lancez votre jeu `ICWars`. Vous vérifierez :

1. que le jeu démarre en affichant un curseur représentant le `RealPlayer` selon la figure 2 ;
2. qu'il peut être déplacé librement sur toute l'aire au moyen des flèches directionnelles mais qu'il ne doit pas pouvoir en sortir ;
3. que les acteurs unités sont placés conformément à la spécification de l'énoncé(voir la

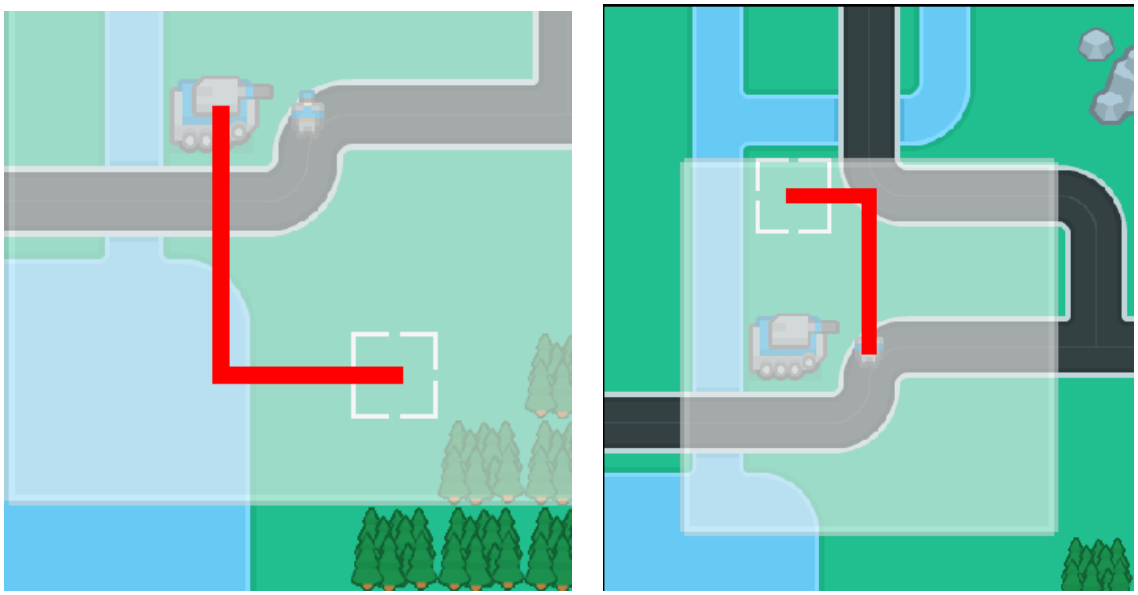

FIG. 3 : Dessin du rayon d'action et du chemin séparant le joueur d'une unité sélectionnée (à gauche l'unité sélectionné est le tank et à droite le soldat)

figure 2) ;

4. que la touche `N` permet de passer au niveau supérieur ou de décréter la fin de partie s'il n'y a plus de niveau ;

5. et que la touche `R` permet de réinitialiser le jeu selon la spécification décrite plus haut.

2.4 Informations graphiques

Il sera certainement très utile de pouvoir visualiser le rayon de déplacement des unités ; c'est ce qui permet typiquement au joueur de voir jusqu'où il peut la déplacer. Il s'agit maintenant de compléter notre ébauche de jeu pour afficher de telles informations.

Associez pour cela un attribut, `range`, de type `ICWarsRange` (voir la section 6.5 du tutoriel) à chaque unité. Prévoyez ensuite, pour les unités toujours, une méthode permettant de dessiner cet attribut ainsi que le plus court chemin entre la position courante de l'unité et un point destination. Le codage de cette méthode étant un peu technique, il vous est fourni :

```
/**
  * Draw the unit's range and a path from the unit position to
     destination
  * @param destination path destination
  * @param canvas canvas
  */
public void drawRangeAndPathTo(DiscreteCoordinates destination,
   Canvas canvas) {
      range.draw(canvas);
      Queue<Orientation> path =
         range.shortestPath(getCurrentMainCellCoordinates(),
```

```
            destination);
        //Draw path only if it exists (destination inside the range)
        if (path != null){
            new Path(getCurrentMainCellCoordinates().toVector(),
                path).draw(canvas);
        }
    }
```

Soit une unité positionnée en $(fromX, fromY)$, son attribut `range` doit être initialisé lors de la création de l'unité de sorte à contenir tous les noeuds de position $(x+fromX, y+fromY)$, où x et y sont entre $-radius$ et $radius$ et où $radius$ est le rayon d'action de l'unité. L'ajout d'un noeud se fera au moyen de la méthode `addNode` de `ICWarsRange`. Attention, seuls les noeuds qui correspondent à des positions valides dans la grille doivent être ajoutés. Les quatre booléens requis par la méthode `addNode` doivent valoir vrai si le noeud correspondant à la coordonnées $(x+fromX, y+fromY)$ a un voisin à gauche dans le graphe !ICWarsRange ! (resp. au dessus, à droite et au dessous). Pour le voisin de gauche par exemple ce sera le cas si x est plus grand strictement que $-radius$ et $x + fromX$ est supérieur strictement à zéro. Vous veillerez à coder les traitements requis sans recourir à des getters intrusifs (de type `getBehaviour` dans `ICWarsArea` par exemple).

La question se pose maintenant de savoir par qui va être invoquée la méthode `drawRangeAndPathTo`. Lorsque le joueur sera en capacité de sélectionner une unité avec laquelle il veut jouer, c'est en principe pour cette unité précisément que l'on souhaitera invoquer cette méthode ; son rôle sera de montrer jusqu'où le joueur a le droit de la déplacer et de visualiser le chemin la séparant de ce dernier.

Nous partirons donc ici de l'idée que c'est le joueur qui *sait* quelle unité il a sélectionné à un moment donné (donc « a-un » attribut « unité sélectionnée ») et que c'est à lui, par conséquent que revient la charge de dessiner les informations qui lui sont utiles pour la déplacer. Comme par la suite d'autres informations graphiques liées au joueur se grefferont aussi au jeu, comme typiquement sur quel *type de cellule* il se trouve en se déplaçant, il est raisonnable de déléguer l'affichage des informations graphiques utiles au joueur à une classe spécifique[3]. Il vous est donc demandé ici d'adhérer aux spécifications suivantes :

- les tâches d'affichages suggérées ici doivent être déléguées à un objet de type `ICWarsPlayerGUI` (gestionnaire du graphisme du joueur) à coder dans le paquetage `icwars.gui` ;
- vous considérerez que tout objet de type `RealPlayer` est doté d'un attribut `ICWarsPlayerGUI` en charge d'en dessiner les caractéristiques ;
- La méthode `draw` de `RealPlayer` aura la charge d'invoquer la méthode de dessin de son `ICWarsPlayerGUI` ;
- La méthode `draw` de `ICWarsPlayerGUI` aura pour rôle de d'invoquer la méthode `drawRangeAndPathTo` sur l'unité sélectionnée par le joueur dont elle est en charge de l'affichage, de sorte à ce quelle dessine le rayon d'action de l'unité et le chemin séparant l'unité du joueur.

[3] principe du « separation of concerns » en programmation orientée objet

Vous compléterez le code de sorte à ce que le contrôle suivant, ajouté à la méthode `update` du jeu `ICWars` :

```
if (keyboard.get(Keyboard.U).isReleased()) {
      ((RealPlayer)player).selectUnit(1); // 0, 1 ...
}
```

où `player` est l'attribut « joueur » du jeu, se traduise par le fait que l'unité sélectionnée par `player` soit celle à l'indice donné en paramètre dans sa liste d'unités (dans l'exemple l'unité à la position 1). Si l'indice est trop grand, ce contrôle sera simplement ignoré (il ne se passera rien).Ceci devrait se traduire par les affichages voulus pour l'unité choisie, tels qu'illustrés dans la figure 3.

Les contraintes suivantes vous sont imposées :

- `ICWarsPlayerGUI` est un `Graphics` et il a connaissance du joueur dont il a la charge des graphismes ;
- son constructeur doit avoir pour entête :

  ```
  public ICWarsPlayerGUI(float cameraScaleFactor,
     ICWarsPlayer player)
  ```

  (le premier paramètre n'a pas besoin d'être utilisé pour le moment) ;
- les méthodes de dessin ne doivent être invoquées que par d'autres méthodes de dessin (cela permet de découpler la logique du jeu des modalités d'affichages) ;
- si `ICWarsPlayer` donne accès à une unité il le fera de façon protégée ;
- vous ne coderez pas de getter `public` intrusif du type `getSelectedUnit` dans `RealPlayer`.

Enfin, vous ne vous préoccuperez pas de mécanismes permettant de de-sélectionner une unité. Nous y reviendrons plus tard.

2.4.1 Tâche

Il vous est demandé de :

- coder les concepts décrits précédemment conformément aux spécifications et contraintes données.
- tester vos développements de façon « ad'hoc » sélectionnant « en dur » dans le code différentes unités pour le joueur (en l'occurence soit la 0 soit la 1).

Vous vérifierez alors :

1. que les rayons de déplacement s'affichent proprement pour chacune des unités (au moyen de la touche 'U') et tel qu'indiqué dans la figure 3.
2. que le chemin en rouge s'affiche entre le joueur et l'unité sélectionnée dès qu'il entre dans son champs de déplacement
3. que les touches 'R' et 'N' restent fonctionnelles.

2.5 Validation de l'étape 1

Pour valider cette étape, toutes les vérifications des sections 2.3.1, 2.4.1,doivent avoir été effectuées.

Le jeu `ICWars` dont le comportement est décrit ci-dessus est à rendre à la fin du projet.

Cette partie est délibérément guidée car elle constitue le coeur du fonctionnement du jeu. Tous les détails ne sont cependant évidemment pas donnés.

3 Interactions avec les unités (étape 2)

L'acteur `RealPlayer` modélisé à l'étape précédente, matérialise le joueur « humain » qui devra affronter plus tard un joueur virtuel. Chacun de ces joueurs, humain ou virtuel, aura pour objectif de sélectionner des unités pour les déplacer et/ou leur faire entreprendre des actions en vue de vaincre l'adversaire.

Le but principal de cette étape est de permettre à `RealPlayer` de commencer à utiliser ses unités de combats. Ceci se fera en tirant parti du mécanisme général des *interactions entre acteurs*, tel que décrit dans le tutoriel 3.

Il s'agit concrètement de faire interagir `RealPlayer` avec les acteurs « Unité de combat ». L'interaction consistera tout simplement à sélectionner l'unité à utiliser pour la *déplacer*.

Vous aurez également à compléter les informations graphiques aidant au déroulement du jeu.

3.1 Déroulement type d'une partie de ICWars

Pour cette étape, nous ferons intervenir deux joueurs de type `RealPlayer` jouant tour à tour. Vous coderez cependant les choses de façon plus générale, de sorte à permettre d'avoir potentiellement plus de joueurs. Pour comprendre les nouveaux éléments de code à introduire, il est utile de rappeler les modalités de jeux de `ICWars` tels qu'anticipés :

1. le jeu est une succession de manches ;
2. un joueur est actif tant qu'il lui reste des unités utilisables (sinon on dira qu'il est défait) ;
3. une manche consiste à faire jouer tour à tour l'ensemble des joueurs encore actifs ;
4. au début d'une manche le premier joueur prend la main ;
5. pour jouer son tour, il se déplace pour atteindre les unités qu'il veut faire jouer ;
6. une fois qu'il est sur une unité, il peut la sélectionner au moyen de la touche `Enter` ; si l'unité a été sélectionnée, le joueur se déplace alors pour indiquer quelle position finale il veut lui faire atteindre et confirme cette position en appuyant à nouveau sur `Enter` ; le déplacement ne peut se faire que dans les limites du rayon de déplacement de l'unité ;
7. une fois l'unité déplacée, le joueur peut lui demander d'effectuer différentes actions (au moyen de touches) ;

8. les deux étapes précédentes peuvent être réalisées pour chacune des unités, mais une seule fois par tour pour chaque unité ;

9. lorsque le joueur a terminé de positionner et utiliser ses unités, la touche `Tab` permet d'indiquer que le tour du joueur courant est terminé et ainsi de passer la main au joueur suivant qui peut alors jouer selon les même modalités ;

10. une manche se termine lorsque tous les joueurs ont joué leur tour ; une nouvelle manche peut alors commencer avec tous les joueurs qui n'ont pas été défait lors de la manche courante ;

11. le jeu se termine lorsqu'à l'issue d'une manche, il ne reste plus qu'un joueur.

Pour commencer nous ne nous préoccuperons pas des détails d'implémentation des étapes 2, 3 et 4. Il s'agit simplement de commencer à mettre en place le (dé)placement des unités et à ébaucher la dynamique générale du jeu. Lors de l'étape précédente, la touche 'U' forçait la sélection d'une unité d'indice donné. Force est de constater que ce qui était alors utile à des fins de test, ne correspond pas à la logique du jeu. Vous commenterez donc les lignes correspondantes dans `ICWars` ; elles ne devraient plus nous être nécessaires pour cette étape (pas plus que la méthode `selectUnit(int index)`).

3.2 États du joueur et interactions avec les unités

Jusqu'ici la méthode `update` de `RealPlayer` lui permet essentiellement de se déplacer en obéissant aux flèches directionnelles. La description des modalités de jeu donnée plus haut suggère que cette méthode devra être complétée pour permettre des comportements plus complexes : il s'agit de modéliser le fait que le comportement du joueur doit dépendre des différents *états* dans lesquels il peut être.

Concrètement, le joueur va en effet passer par différents états : celui où il doit sélectionner une unité, celui où il a choisi une unité et où il la déplace, celui où il fait entreprendre une action à une unité déplacée et ainsi de suite ; et qu'à chaque état va correspondre un comportement spécifique.

Le comportement du joueur peut être modélisé par ce que l'on appelle en informatique un automate à états finis et dont une représentation graphique partielle pour le jeu qui nous intérèsse est donnée par la figure 4.

Il vous est demandé maintenant d'ajouter à votre code les éléments permettant de modéliser le fait que :

- tout joueur (`ICWarsPlayer`) a un *état courant* ; souvenez vous aussi qu'il peut *mémoriser* une unité sélectionnée (celle en cours d'utilisation) ; au départ bien sûr sa mémoire est vide ;

- tout joueur à sa création est dans l'état `IDLE` ;

- l'ensemble des états par lesquels il peut passer est le même quelque soit le type du joueur : vous anticiperez les états suggérés par la figure 4

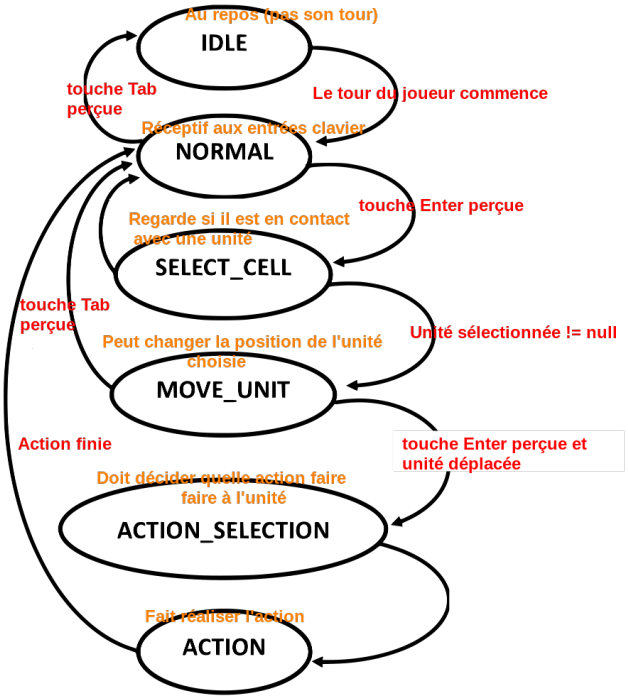

FIG. 4 : Automate décrivant le comportement du joueur : en noir les états et leur brève description, en rouge les actions causant des transitions vers un autre état.

Vous compléterez ensuite la méthode `update` de `RealPlayer` de sorte à ce qu'en plus de ce qu'elle fait déjà, elle invoque une méthode de mise à jour des états. L'algorithme à implémenter est donné par l'automate à état fini de la figure 4, c'est à dire typiquement :

Si l'état courant est :

- `IDLE` ne rien faire (ce n'est plus son tour) ;
- `NORMAL` si la touche `Enter` est appuyée passer à l'état `SELECT_CELL` (qui indique que le joueur est réceptif à une interaction de contact avec une unité, nous en reparlerons plus bas), sinon si la touche `Tab` est appuyée, repasser à l'état `IDLE` (la main est passée à un adversaire) ;
- `SELECT_CELL` et qu'une unité sélectionnée a été mémorisée passer à l'état `MOVE_UNIT` ;
- `MOVE_UNIT` : si la touche `Enter` est appuyée déplacer l'unité mémorisée à l'emplacement courant (voir indications plus bas) ; la marquer comme utilisée et passer à l'état NORMAL ;
- pour toute autre valeur de l'état, ne rien faire pour le moment.

Indications :

- utilisez une instruction `switch` et modélisez les états avec un `type énuméré` (attention alors au bon usage du `break` !). Vous anticiperez la présence des états `ACTION_SELECTION`

et `ACTION` dans le `switch` mais pour le moment, aucun traitement n'y sera associé et aucun code ne permettra de transiter vers ces états.

- introduisez dans `ICWarsPlayer`, une méthode `startTurn` prenant en charge les traitements nécessaires à entreprendre lorsque le joueur commence son tour. Pour le moment cette méthode permettra au joueur de passer à l'état `NORMAL` où il devient actif et réceptif aux contrôles et centrera la caméra sur lui. L'ensemble de ses unités doivent alors redevenir disponibles. Vous ferez en sorte que cette méthode soit invoquée sur l'unique joueur de `ICWars` au démarrage du jeu (l'unique joueur prend d'emblée la main et commence son tour).

- lorsqu'il quitte une cellule, si le joueur est toujours dans dans l'état `SELECT_CELL` (ce qui signifie qu'il ne s'est pas décidé à sélectionner quelque chose) , il faut qu'il rebascule dans l'état `NORMAL` (qu'il se remontre disponible aux interactions futures). Il faudra donc ici redéfinir la méthode `onLeaving` héritée de `AreaEntity` pour vous en assurer. Cette méthodes permet en effet d'indiquer les traitements à entreprendre lorsqu'une entité quitte une cellule.

- les unités ne pouvant être utilisées qu'une fois par tour de chaque joueur, elles doivent pouvoir être marquées comme utilisées (ou à nouveau disponible). Vous introduirez les éléments de code nécessaires.

Déplacement d'une unité Tout acteur de type `MoveableAreaEntity` dispose de la méthode `boolean changePosition(DiscreteCoordinates newPosition)` lui permettant de passer de la position qu'il occupe à une position de destination.

Il faudra redéfinir cette méthode pour les unités pour en empêcher le déplacement vers une position sortant de leur rayon de déplacement possible. L'algorithme à appliquer sera le suivant :

« s'il n'existe aucun noeud dans le rayon d'action de l'unité correspondant à `newPosition` ou si le déplacement est impossible selon les critères de la super-classe, la méthode retournera `false` sinon `true` ».

Indication : utilisez la méthode `nodeExists` des `ICWarsRange` pour tester l'existence d'un noeud et n'oubliez pas d'adapter le rayon d'action à la nouvelle position si le déplacement a pu se faire.

Avant de pouvoir tester les modifications précédemment suggérées, il est nécessaire de complémenter deux autres aspects du code :

- l'impact de la notion d'état sur le déplacement et le graphisme : le joueur ne peut plus forcément bouger inconditionnellement et les informations sur son rayon d'action ne sont pas pertinentes dans tous les états ;

- la sélection/mémorisation d'une unité à déplacer : ceci se fera par interaction (de contact) entre l'unité et le joueur.

Les deux paragraphes suivants donnent quelques indications nécessaires à ce sujet.

3.2.1 Impact des états sur le déplacement

Le `RealPlayer` ne doit plus être capable d'obéir inconditionnellement aux flèche directionnelle. Il ne doit pouvoir être déplacé que si il est dans un des états : `NORMAL`, `SELECT_CELL` ou `MOVE_UNIT`. De façon analogue, le dessin des possibilités offertes pour déplacer l'unité sélectionnée (appel de `drawRangeAndPathTo`) ne doit se faire que dans l'état `MOVE_UNIT`.

Apportez les modifications nécessaires à votre code pour adhérer à ces nouvelles contraintes.

3.2.2 Interactions avec les unités

Selon l'implémentation qui vous a été suggérée précédemment, la transition de l'état `SELECT_CELL` à l'état `MOVE_UNIT` ne peut se faire que si une unité a été mémorisée/sélectionnée[4]. Nous souhaitons donc maintenant que le joueur puisse effectivement mémoriser une unité avec laquelle il entre en « interaction de contact » lorsqu'il est en mode « sélection d'unité » (c'est à dire dans l'état `SELECT_CELL`).

Il vous est donc demandé maintenant d'appliquer le schéma suggéré par le tutoriel 3[5] pour mettre en place les interactions de `RealPlayer` avec ses unités de combat.

Pour cela, commencez par mettre en place le fait que tous les `ICWarsPlayer` deviennent des `Interactor` (ils peuvent faire subir des interactions à des `Interactable`). En tant que `Interactor`, `ICWarsPlayer` voudra systématiquement toutes les interactions de contact et, aucune interaction à distance (ce que vous pourrez changer si vous complexifiez le jeu plus tard).

Créez ensuite, dans un sous-paquetage `game.icwars.handler`, l'interface `ICWarsInteractionVisitor` héritant de `AreaInteractionVisitor`, et qui fournit une définition par défaut pour tous les acteurs spécifiques du jeu `ICWars`. Pour le moment, les acteurs spécifiques déjà codés sont le joueur (`RealPlayer`) et les unités de combat (`Unit`). Nous décidons ici de modéliser l'interaction pour des unités en général car comme évoqué plus haut, l'interaction se borne à mémoriser l'unité pour l'utiliser (et il s'agit donc a priori du même traitement générique quelque soit l'unité).

Toutes les définitions (par défaut) de `ICWarsInteractionVisitor` auront un corps vide pour exprimer le fait que par défaut, l'interaction consiste à ne rien faire. Libre à vous de modifier ce point par la suite si vous l'estimez pertinent.

`RealPlayer` en tant que `Interactor` du jeu `ICWars`, doit fournir le cas échéant une définition plus spécifique de ces méthodes.

Pour cela, définissez dans la classe `RealPlayer`, une classe imbriquée privée `ICWarsPlayerInteractionHandler` implémentant `ICWarsInteractionVisitor`.

Ajoutez-y la définition nécessaire pour gérer plus spécifiquement l'interaction avec une unité. L'algorithme à appliquer est simplement le suivant :

[4] en clair que si l'unité sélectionnée par le joueur vaut autre chose que `null`

[5] Un complément vidéo est aussi disponible pour expliquer la mise en oeuvre des interactions : https://proginsc.epfl.ch/wwwhiver/mini-projet2/mp2-interactions.mp4

« si le joueur est dans l'état où il est réceptif à sélectionner une unité (`SELECT_CELL`) et si l'unité avec laquelle il interagit est de sa propre faction, alors faire en sorte qu'il mémorise l'unité en question (qu'elle devienne son unité sélectionnée). Pensez aussi au graphisme : c'est au moment de l'interaction que le gestionnaire du graphisme peut être informé de l'unité dont il faut dessiner le rayon de déplacement ».

Indication : conformément au tutoriel 3, pour que cela fonctionne, il faut indiquer que `ICWarPlayer` accepte de voir ses interactions avec les autres acteurs gérées par un gestionnaire de type `ICWarsInteractionVisitor`. La méthode `acceptInteraction` qui ne faisait rien dans `GhostPlayer`, doit être en charge de ce traitement dans `ICWarPlayer`

```
@Override
  public void acceptInteraction(AreaInteractionVisitor v) {
    ((ICWarsInteractionVisitor)v).interactWith(this);
  }
```

Vous ferez en sorte que la méthode `interactWith` de `RealPlayer` ne fasse quelque chose que si il n'est pas en cours de déplacement (`isDisplacementOccurs()`).

Question 1

La logistique mise en place, telles qu'exposée dans les tutoriels et exploitée concrètement dans cette première partie du projet, peut sembler *a priori* inutilement complexe. L'avantage qu'elle offre est qu'elle modélise de façon très générale et abstraite, les besoins inhérents à de nombreux jeux où des acteurs se déplacent sur une grille et interagissent soit entre eux soit avec le contenu de la grille. Comment pourriez-vous en tirer parti pour mettre en oeuvre un vrai jeu de type RPG, à l'image de celui ébauché dans le tutoriel, mais où le personnage aurait à interagir avec toutes sorte d'autres personnages par exemple ? Que suffirait-il de définir ?

Toutes les interactions à venir devront impérativement être codées selon le schéma mis en place lors de cette partie et ne devront pas nécessiter de tests de types sur les objets.

3.2.3 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Vous vérifierez alors :

1. que le joueur « humain » (curseur) peut sélectionner une unité sur laquelle il est positionné au moyen de la touche *`'Enter'`* et qu'il peut la déplacer dans le rayon de déplacement de cette dernière, sa position finale est aussi confirmée par la touche *`'Enter'`*;

2. que le dessin du rayon de déplacement et du chemin ne se font que lorsque le joueur a sélectionné une unité au moyen de la touche *`'Enter'`* :

3. et que les points vérifié dans la section 2.3.1 sont toujours fonctionnels

3.3 Dynamique du jeu à plusieurs joueurs

Un jeu de type `ICWars`, tel que codé jusqu'ici, n'introduit qu'un seul joueur contrôlable selon des modalités élémentaires. Il s'agit maintenant d'en complexifier le comportement afin de mettre en oeuvre la dynamique de jeu précédemment décrite.

3.3.1 Ensemble de joueurs

Il vous est demandé de remplacer le joueur unique caractérisant un jeu `ICWars` par une liste de joueurs de type `ICWarsPlayer` (nous anticipons le fait que les joueurs impliqués ne sont pas tous forcément des `RealPlayer`).

A cette étape, lors du lancement du jeu, cette liste sera remplie au moyen de deux `RealPlayer` dont les caractéristiques seront par exemple les suivantes :

Joueur	faction	position initiale	unité 1	unité 2
1	alliée	(0, 0)	Tank en (2,5)	Soldat en (3,5)
2	ennemie	(7,4)	Tank en (8,5)	Soldat en (9,5)

L'ensemble des joueurs impliqués devra être enregistré dans l'aire courante. A noter qu'il peut être judicieux d'ajouter à vos aires une méthode `getEnemySpawnPosition()` déterminant la position de l'adversaire au départ (vous pourrez prendre $(7, 4)$ pour `Level0` et $(17, 5)$ pour `Level1`).

Introduisez ensuite les attributs supplémentaires permettant de modéliser le déroulement du jeu tels que décrit plus haut, à savoir :

- la liste des joueurs en attente de jouer la manche courante ;
- la liste des joueurs en attente de joueur la manche suivante ;
- le joueur couramment actif (celui qui a la main).

Il ne faudra pas oublier d'initialiser proprement ces nouveaux attributs aux endroits opportuns. Enrichissez enfin le code de tous les éléments nécessaire pour permettre :

- de tester qu'un joueur est défait ;
- de faire afficher les unités déjà utilisées avec un visuel différent, il suffira pour cela de rendre opaque le visuel de l'unité utilisée en jouant sur la composante de transparence du sprite associé (appelez par exemple la méthode `setAlpha(0.5f)` sur le sprite pour l'opacifier et `setAlpha(1.f)` pour l'avoir en clair) ;
- de remettre à disposition du joueur toutes ses unités en début de tour ;
- de contrôler la liste des joueurs en attente de jouer la prochaine manche pour en supprimer ceux qui auraient perdu.

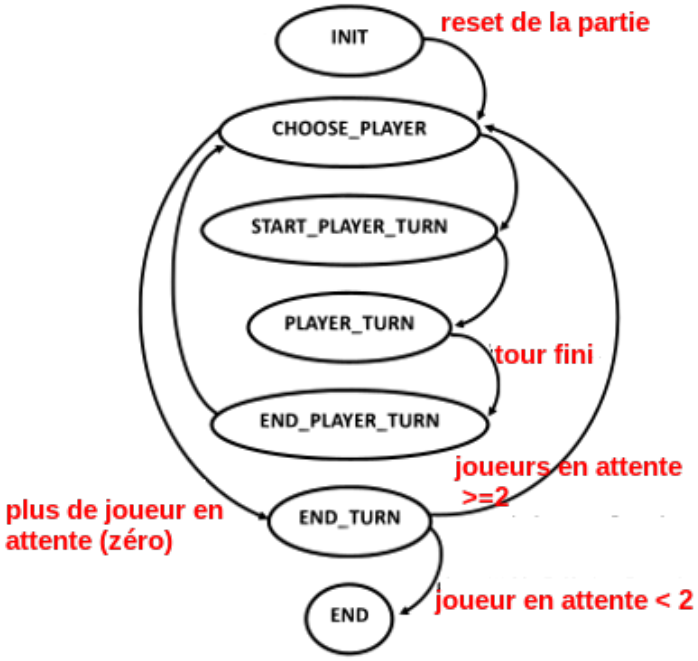

FIG. 5 : Automate décrivant le déroulement d'une partie : en noir les états, en rouge les actions causant des transitions vers un autre état.

3.3.2 Dynamique du jeu

Le déroulement d'une partie peut lui aussi être modélisé par un automate à états finis et dont une représentation graphique partielle est donnée par la figure 5.

Il vous est demandé maintenant de modifier l'algorithme `update` de `ICwars` de sorte à ce qu'en plus de ce qu'il faisait jusqu'ici, soient exécutées des instructions implémentant la logique de l'automate à état fini de la figure 5.

Concrètement, l'algorithme implémentant l'automate sera le suivant. Si l'état du jeu vaut :

- `INIT` initialiser la liste des joueurs en attente de jouer la manche courante et passer à l'état `CHOOSE_PLAYER` ;
- `CHOOSE_PLAYER` si la liste des joueurs en attente de jouer la manche courante est vide passer à l'état `END_TURN` (gestion de fin de manche) ; sinon choisir le nouveau joueur actif et le supprimer de la liste de joueur en attente de jouer la manche courante ; et passer à l'état `START_PLAYER_TURN` ;
- `START_PLAYER_TURN` invoquer la méthode `start_turn` sur le joueur couramment actif et passer à l'état `PLAYER_TURN`
- `PLAYER_TURN` : si le joueur couramment actif a terminé son tour (son état est repassé à `IDLE`), passer à l'état `END_PLAYER_TURN` ;

- `END_PLAYER_TURN` : si le joueur couramment actif est défait le supprimer de l'aire de jeu, sinon l'introduire dans la liste des joueurs en attente de jouer le prochain tour et passer à l'état `CHOOSE_PLAYER` ; Il faudra veiller à faire en sorte que toutes ses unités redeviennent utilisables (pour le visuel notamment) ;
- `END_TURN` : supprimer tous les joueurs défaits (de la liste des acteurs en attente de jouer la prochaine manche et du jeu en général). S'il ne reste plus qu'un joueur dans la liste des joueurs en attente de joueur la prochaine manche passer à l'état `END`, sinon faire passer tous les joueurs de la liste en attente de jouer la prochain manche dans celle en attente de jouer la manche courante et re-passer à l'état `CHOOSE_PLAYER`
- `END` : gérer la fin de partie (passer au niveau suivant ou terminer le jeu).

Retouche au graphisme Afin de faciliter le déroulement, vous ne dessinerez le joueur (curseur) que lorsqu'il est actif (état différent de `IDLE`).

3.3.3 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Vous devriez disposer alors d'une nouvelle version presque complète du jeu `ICWars`. Vérifiez alors :

1. qu'il est possible de faire jouer les deux joueurs tour à tour, tel que décrit dans les modalités de jeu (la touche `Tab` permet de transiter d'un joueur à l'autre) ;
2. que dans une manche donnée le joueur actif peut sélectionner toutes ses unités mais qu'une seule fois ;
3. que chacun des joueurs peut déplacer les unités sélectionnées tel que validé dans la section 3.2.3
4. que les unités redeviennent sélectionnables pour chaque joueur lorsqu'un nouvelle manche s'entame (après que les deux joueurs ont fini de jouer la manche précédente).

Pour le moment, il n'est pas encore possible de tester si l'un ou l'autre des joueurs est défait, mais vous allez bientôt y remédier grâce aux belliqueuses prérogatives que vous allez leur attribuer lors de la prochaine étape.

3.4 Validation de l'étape 2

Pour valider cette étape, toutes les vérifications des sections 2.3.1, 2.4.1, 3.2.3, et 3.3.3 doivent avoir été effectuées.

Le jeu `ICWars` dont le comportement est décrit ci-dessus est à rendre à la fin du projet.

Pour programmer cette partie, il est important de bien lire l'énoncé dans son intégralité, en tout cas pour chaque sous-étape de validation (ne codez pas linéairement au fur et à mesure que vous lisez). Il est important de bien réfléchir à où placer les attributs et méthodes nécessaires, sans avoir recours à des accès trop intrusifs (pas de getters sur les unités du jeu par exemple).

4 Actions et joueur virtuel (étape 3)

Il s'agit maintenant d'introduire des modalités plus complexes d'utilisation des unités sélectionnées, comme le fait de leur faire entreprendre des actions. Concrètement, chaque unité sélectionnée pourra non seulement être déplacée mais aussi réaliser une « action », par exemple *attaquer* l'adversaire.

L'idée est donc que chaque unité peut exécuter un certain nombre d'*actions*. Ces dernières seront proposées au joueur « humain » qui aura à choisir laquelle il veut voir s'exécuter. Dans ce qui suit des indications vous sont données sur comment modéliser les actions exécutables et comment étendre le code existant pour permettre au joueur de choisir une action à faire exécuter à ses unités.

4.1 Actions

Pour modéliser la notion d'action, vous adhérerez aux spécifications suivantes :

- La notion d'action sera codée dans un sous-paquetage `action` du paquetage `icwars.actor.unit`.
- Une action est un objet dessinable.
- Elle est attachée à une *unité* (celle pour laquelle elle s'exécute) et est exécutée dans une *aire*. Elle a connaissance de ces deux informations qui lui sont transmises à la construction mais ne doit pouvoir les transmettre elle-même, pour les besoins de l'encapsulation, qu'à des sous-classes d'actions.
- Un *nom* de type `String` y est associé ainsi qu'une *clé* du clavier permettant de la sélectionner (un `int`[6], jetez un oeil à l'interface fournie `Keyboard`). Le nom et la clé ne peuvent être définis concrètement pour une action quelconque.
- la méthode principale associée à une action sera la méthode

  ```
  public void doAction(float dt, ICWarsPlayer player,
     Keyboard keyboard)
  ```

  qui permettra à une action de s'exécuter pour un joueur donné, sur un pas de temps `dt` et au moyen d'interaction transmises par le clavier `keyboard`. Cette méthode ne peut évidemment pas être définie pour une action en général.

[6] les touches sont modélisables au moyen d'entiers

Deux types spécifiques d'actions sont à coder pour la partie obligatoire du projet. Libre à vous d'enrichir les possibilités d'action dans les extensions.

4.1.1 L'action "Attendre"

Un joueur peut décider de ne rien faire avec une unité sélectionnée. On dira qu'il choisi d'attendre pour peut-être l'utiliser au prochain tour.

L'action `Wait` aura pour nom `"(W)ait"` et pour clé de déclenchement associée l'entier `Keyboard.W` fourni par la classe `Keyboard` du paquetage fourni `play.window`. Elle ne se dessinera pas (méthode de dessin ne faisant rien) et s'exécutera en :

- marquant comme utilisée l'unité à laquelle elle est associée ;
- et en faisant retourner le joueur pour lequel elle travaille à l'état `NORMAL`.

4.1.2 L'action "Attaquer"

Un joueur peut décider d'attaquer une unité adverse avec une unité sélectionnée. L'action `Attack` aura pour nom `"(A)ttack"` et pour clé de déclenchement associée l'entier `Keyboard.A`.

Cette action devra s'exécuter selon l'algorithme suivant :

1. trouver toutes les unités ennemies « attaquables » dans le rayon de son unité (toutes les unités de factions différenres, l'unité de l'action est donc l'unité attaquante) ;
2. si le clavier perçoit que les touches `LEFT` ou `RIGHT` sont utilisées naviguer vers la gauche resp. vers la droite pour sélectionner l'unité à attaquer : initialement l'action considère que l'unité à attaquer est la première dans la liste, mais si la touche `RIGHT` est perçue alors l'unité à attaquer sera la suivante dans la liste (et inversement pour la touche `LEFT`) ; Si le clavier perçoit que la touche `Enter` est utilisée passer à l'attaque :
   (a) infliger à l'unité cible (sélectionnée) le dommage spécifique de l'unité attaquante (souvenez que chaque unité dispose d'une méthode lui permettant de connaître combien de points de dommage elle inflige par attaque). La formule par laquelle l'unité perd des points de vie est la suivante

   ```
   hp = hp - received_damage + defense_stars
   ```

   où `received_damage` est le nombre de points de dégât infligés par l'attaquant et `defense_stars` le nombre d'étoiles de défenses associées à la cellule occupée par l'unité ;
   (b) marquer l'unité associée à l'action (l'unité attaquante) comme utilisée recentrer la caméra sur le joueur et faire revenir ce dernier à l'état `NORMAL`

Pour que l'unité mémorise le nombre d'étoiles de défense de la cellule qu'il occupe, vous appliquerez un schéma d'interaction analogue à celui permettant au joueur de mémoriser une unité (`Unit` jouant le rôle d'`Interactor`) : l'unité mémorise les étoiles de défenses par interaction de contact avec les cellules.

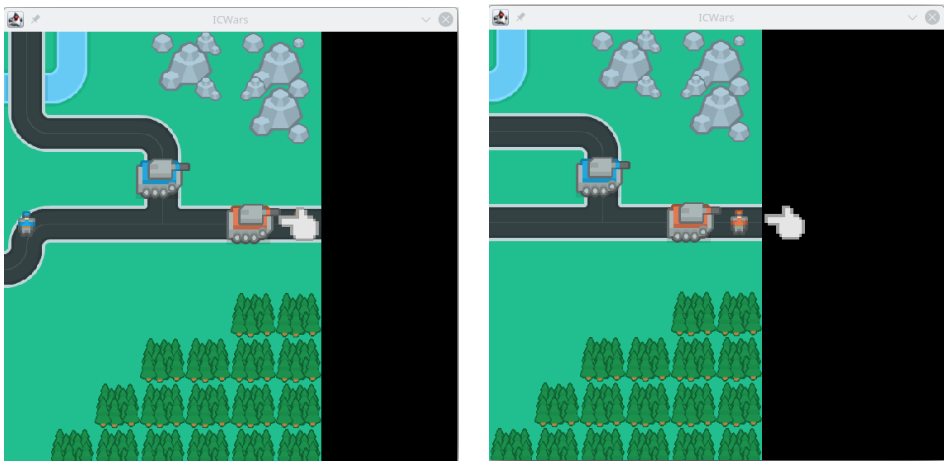

FIG. 6 : Désignation graphique de l'unité ciblée (à gauche le tank est ciblé, à droite le soldat)

Il vous revient de compléter le codage des unités pour trouver les cibles de l'attaque. Pour ne pas casser l'encapsulation il est recommandé de ne pas livrer cette information sous la forme d'une liste d'unités.

Indications : l'aire a en principe mémorisé la liste de ses unités, il suffit de retourner les indices des unités à attaquer dans cette liste, plutôt que les unités elles-mêmes.

La méthode de dessin de l'action `Attack` doit permettre de signaler qui est l'unité cible de l'attaque (voir la figure 6).

Il faudra donc associer à l'action `Attack` un attribut, appelons-le `cursor` (c'est l'élément qui pointe sur les unités ciblées dans la figure 6) , de type `ImageGraphics` construit comme ceci :

```
new ImageGraphics(ResourcePath.getSprite("icwars/UIpackSheet"),
   1f, 1f,
new RegionOfInterest(4*18, 26*18,16,16));
```

La méthode de dessin de `Attack` va :

1. ne rien faire s'il n'y a pas d'unité ciblées, sinon, centrer la caméra sur l'unité ciblée ;
2. placer l'attribut `cursor` à droite de l'objet sur lequel est centré la caméra :

   ```
   cursor.setAnchor(canvas.getPosition().add(1,0));
   ```

3. dessiner l'attribut `cursor`.

4.2 Actions d'une unité

Complétez le codage des unités de sorte à ce :

- qu'à chaque type d'unité soit associée une liste d'actions spécifiques ;

(a) (b) (c)

FIG. 7 : (a) en haut à droite : affichage des actions possibles, (b) en bas à droite : affichage des caractéristiques des cellules traversées, (c) affichage des caractéristiques d'une unités (ces trois affichages sont exclusifs)

- que cette liste contienne une action `Attack` et une action `Wait` pour l'unité `Soldier`. Ces actions sont construites une fois pour toute lors de l'initialisation de l'unité et ne doivent pas pouvoir être modifiées.
- que ce soit aussi le cas pour l'unité `Tank` mais pas forcément pour d'autres types d'unités potentielles. Vous présumerez donc que la liste des actions n'est pas définie au niveau d'abstraction de l'unité.

4.3 Complétion du joueur

Il est temps maintenant de compléter la méthode `update` de `RealPlayer` de sorte à ce qu'il puisse effectivement choisir des actions à faire exécuter à ses unités. Complétez la machine à état finis de sorte à ce que :

- dans l'état `ACTION_SELECTION` : le joueur parcourt la liste des actions spécifiques à l'unité sélectionnée. Si pour une action donnée, `act`, la clé associée à cette action est appuyée est sur le clavier, le joueur passe à l'état `ACTION` (en mémorisant `act` comme l'action à exécuter) ;
- dans l'état `ACTION` : le joueur invoque la méthode `doAction` sur l'action mémorisée.

La transition depuis `MOVE_UNIT` doit être revue : si la touche `Enter` est perçue, on va plutôt tester si l'unité sélectionnée a pu être bougée et si oui, transiter vers l'état `ACTION_SELECTION`. Complétez également la méthode `doAction` de `Attack` de sorte que si la liste des cibles à attaquer et vide (ou si le clavier perçoit la touche `Tab`), la caméra se recentre sur le joueur qui doit repasser en état `SELECT_ACTION`. La méthode `doAttack` ne fera alors rien de plus (le but est ici de redonner une chance de jouer au joueur s'il a inutilement essayé de déclencher une attaque alors qu'aucune cible n'est atteignable.

Complétez enfin la méthode `draw` de `ICWarsPlayerGUI` de sorte à ce qu'elle affiche :

- Les actions exécutables par l'unité sélectionnée (voir la figure 7 en (a)). Cela permet d'informer le joueur de quelles actions il peut demander l'exécution pour une unité sélectionnée. Pour cela, vous décommenterz le contenu de `ICWarsActionPanel` (fourni dans `icwars.gui`) et créerez un attribut de ce type dans `ICWarsPlayerGUI`. Il suffira ensuite à `ICWarsPlayerGUI` d'utiliser la méthode `setActions` sur cet attribut, en lui passant en paramètre les actions exécutables par l'unité sélectionnée, puis de le dessiner. Ceci ne doit bien sûr être réalisé que dans l'état `ACTION_SELECTION` du joueur.

- Les caractéristiques des cellules sur lesquelles il est positionné (voir la figure 7 en (b)). Pour cela, `ICWarsPlayerGUI` peut utiliser un attribut du type fourni `ICWarsInfoPanel`. Décommentez le contenu de ce fichier qui se trouve dans `icwars.gui`. Un objet `ICWarsInfoPanel` a pour charge de représenter graphiquement les informations liées à un type de cellule et à une unité données. L'enjeu ici est de compléter le code de sorte à ce que `RealPlayer` transmette au bon moment à son `ICWarsPlayerGUI` les informations qui sont nécessaires au `ICWarsInfoPanel`, à savoir l'unité sur laquelle il est éventuellement positionné ainsi que le type de la cellule sur laquelle il se trouve. `ICWarsPlayerGUI` devra dessiner son attribut `ICWarsInfoPanel` si le joueur associé est dans l'état `NORMAL` ou `SELECT_CELL` seulement.

Indication : c'est au moment de l'interaction avec une cellule ou une unité que le joueur sait sur quelle cellule et quelle unité il se trouve (et qu'il peut donc communiquer ces informations à son gestionnaire de graphisme). La classe `ICWarsInfoPanel` a besoin d'une constante `FONT_SIZE` à définir dans `ICWarsPlayerGUI` (donnez lui la valeur 20.f). La classe `ICWarsInfoPanel` a aussi besoin d'une méthode `typeToString()` dans `ICWarsBehaviour.ICWarsCell` qui retourne une représentation sous forme de `String` de la valeur du type énuméré (par exemple retourne *`"Road"`* pour la valeur `ROAD`).

4.3.1 Tâche

Il vous est demandé de :

- coder les éléments suggérés ci-dessus conformément aux spécifications et contraintes décrites ;

- les tester au moyen des niveaux fournis.

Vous vérifierez :

1. que la nature des cellules s'affiche en bas à droite lorsque le joueur se promène en quête de sélectionner une unité ;

2. qu'une fois positionné sur une unité, le même menu affiche les caractéristiques de l'unité ;

3. que le joueur qui a la main, peut désormais sélectionner une action à faire exécuter au moyen des touches `A` et `W` une fois qu'il a déplacé une unité ;

4. que le choix des actions réalisables apparaît dans l'interface graphique au en haut à

droite au moment où le joueur a sélectionné une unité (appuyé `Enter` sur une de ses unités utilisables) ;

5. que si l'action `Wait` est choisie, l'unité concernée devient marquée comme utilisée mais que rien d'autre ne se passe ; le joueur peut continuer à choisir des unités à utiliser parmi celles encore disponibles ;

6. que si l'action `Attack` est choisie, si aucune cible n'est atteignable il soit à nouveau possible de sélectionner une action (`Wait`). Si par contre il y a des cibles atteignables un pointeur s'affiche sur la première d'entre elle. Il doit être possible de déplacer ce pointeur dans la liste de cibles atteignables de façon cyclique au moyen des flèches gauche et droite ;

7. qu'une fois l'unité cible attaquée (`Enter`), elle subit les dégâts de l'attaquant et disparaît si les dégâts sont suffisamment importants pour la détruire ;

8. que la logique générale du jeu, telle que codée à l'étape précédente reste préservée.

4.4 Joueur virtuel

Dans ce dernier volet obligatoire du projet, il vous est demandé d'enrichir la conception en permettant l'ajout d'un joueur virtuel `AIPlayer` (à coder aussi dans le sous-répertoire `icwars.actor.players`). Il se comportera selon une intelligence artificielle (IA) très basique. Il va sélectionner chacune de ses unités, en séquence. Chaque unité sera déplacée de sorte à se rapprocher le plus possible d'une des unités d'une faction opposée, dans les limites de son rayon de déplacement. Puis, cette unité attaquera l'unité ennemie avec le moins de points de vie à portée.

Le `AIPlayer` est évidemment un `ICWarsPlayer`. La mise à jour de ses états devra se faire selon un automate à état fini, de façon analogue à `RealPlayer`.

Afin que les transitions d'états ne soient pas instantanées, ce qui empêche une observation aisée de ce que fait le joueur virtuel, vous ferez en sorte qu'il y ait des petits temps d'attente lors de la transition entre l'état où le joueur sélectionne une unité et celui où il la déplace ainsi que lors de la transition entre l'état où il a déplacé l'unité et celui où elle attaque. Une façon simple d'imposer un temps d'attente consiste à invoquer une méthode codée comme ceci :

```
/**
     * Ensures that value time elapsed before returning true
     * @param dt elapsed time
     * @param value waiting time (in seconds)
     * @return true if value seconds has elapsed, false otherwise
     */
    private boolean waitFor(float value, float dt) {
        if (counting) {
            counter += dt;
            if (counter > value) {
                counting = false;
                return true;
```

```
            }
        } else {
            counter = 0f;
            counting = true;
        }
        return false;
    }
```

Indications :

1. Pour déterminer la nouvelle position de l'unité sélectionnée, vous pouvez commencer par déterminer la position `(x,y)` de l'unité d'une faction opposée la plus proche de l'unité sélectionnée (la position de cette dernière peut être trouvée au moyen de `getMainCellsCoordinates`). Il faut ensuite ramener `(x,y)` à une position se trouvant à l'intérieur du rayon d'action de l'unité sélectionnée si : si x est en dehors du rayon d'action, il faut le ramener aux confins de ce dernier, soit à gauche soit à droite selon le choix qui la rapproche le plus de la cible et pareil pour y en haut et en bas). Il faudra aussi veiller à ce que les x et y ainsi calculés soient dans les limites de la grille. L'unité sélectionnée pourra alors être déplacée en `(x,y)`.

2. Pour la réalisation automatique des actions, il convient d'ajouter une méthode aux `Action`, `doAutoAction`, décrivant leur fonctionnement lorsqu'utilisées par un `AIPlayer`.

Complétez ensuite le jeu `ICWars` en commentant le second `RealPlayer` et en le remplaçant par un `AIPlayer` avec les mêmes caractéristiques.

4.4.1 Tâche

Il vous est demandé de :

- coder les éléments suggérés ci-dessus conformément aux spécifications et contraintes décrites ;
- les tester au moyen des niveaux fournis.

Vous vérifierez :

1. que la logique générale du jeu, telle que codée, jusqu'ici reste préservée ;
2. que lorsque le `AIPlayer` prend la main il fait jouer automatiquement toutes ses unités en les rapprochant le plus possible d'une unité de l'adversaire et en l'attaquant systématiquement ;
3. qu'un petit temps de pause est observé pour permettre l'observation des transitions d'état.

4.5 Validation de l'étape 3

Pour valider cette étape, toutes les vérifications des sections 2.3.1, 2.4.1, 3.2.3 3.3.3, 4.3.1 et 4.4.1 doivent avoir été effectuées.

Le jeu `ICWars` dont le comportement est décrit ci-dessus est à rendre à la fin du projet.

5 Extensions (étape 4)

Pour obtenir des points bonus (pouvant compenser d'éventuels malus sur la partie obligatoire) ou pour participer au concours, vous pouvez coder quelques extensions librement choisies. Au plus 20 points seront comptabilisés : coder beaucoup d'extensions pour compenser les faiblesses des parties antérieures n'est dont donc pas une option possible.

La mise en oeuvre est libre et très peu guidée. Seules quelques suggestions sont données ci-dessous. Pour vous donner une idée du barème, l'extension « cité » décrite ci-dessous est typiquement ce qui vous permettrait d'obtenir jusqu'à 20 points. Les petits ajouts comme un écran de « Game Over » rapportent entre 2 et 4 points typiquement. N'hésitez pas à nous solliciter pour une évaluation du barème de vos extensions si vous décidez de partir dans une direction précise. Un petit bonus pourra aussi être attribué si vous faites preuve d'inventivité dans la conception du jeu.

Vous pouvez coder vos extensions dans le jeu `ICWars` au moyen de niveaux supplémentaires ou dans un nouveau jeu utilisant la logistique que vous avez mise en place dans les étapes précédentes.

Vous prendrez soin de **commenter soigneusement** dans votre `README`, les aires accessibles ainsi que les modalités de jeu. Nous devons notamment savoir quels contrôles utiliser et avec quels effets sans aller lire votre code. Voici un exemple de `README` (partiel) correspondant qui explique comment jouer à un jeu :

- exemple de fichier `README.md` tiré d'un ancien projet : README.md

> Il est attendu de vous que vous choisissiez quelques extensions et les codiez jusqu'au bout (ou presque). L'idée n'est pas de commencer à coder plein de petits bouts d'extensions disparates et non aboutis pour collectionner les point nécessaires ;-).

5.1 Cités

Pour se rapprocher d'avantage du jeu dont le mini-projet s'inspire, vous pouvez introduire des acteurs « cité » que les unités peuvent capturer. Les cités sont répertoriées en alliée, ennemie ou neutre. Une action supplémentaire « capturer » devrait donc être introduite pour les unités. Cette action ne devrait être disponible que quand une unité se trouve sur une ville neutre ou de faction ennemie. Lorsque'une cité est capturée, elle passe dans la faction de l'unité.

Les cités peuvent rapporter des points de vie aux joueurs : par exemple, quand tous les joueurs ont fini de jouer, toutes les unités présentes sur une ville de leur faction gagnent une certaine quantité de points de vie (spécifique à la cité).

Vous noterez que les images « behavior » fournies, prévoient un couleur pour l'emplacement des cités (revoir le type énuméré suggéré pour `ICWarsBehavior`).

Les cités pourraient être mises en place comme de simples éléments du décors. Il est cependant plus avantageux de les introduire sous la forme d'acteurs. Ceci permettra si on le souhaite d'en avoir de plusieurs types, avec des comportements spécifiques ; ce qui donnera de la flexibilité pour complexifier le jeu à souhait. C'est ce qui leur permettrait par exemple de pouvoir être détruites suite à une attaque (par opposition à un élément de décor qui est inamovible).

Indication : introduisez dans `ICWarsBehavior` une méthode

```
void registerActors(Area area)
```

permettant de créer automatiquement des acteurs « cité » sur toutes les cellules de type `CITY` et de les enregistrer comme acteurs de l'aire `area`.

5.2 Autres pistes d'extensions

En réalité, la base que vous avez codée peut être enrichie à l'envi. Voici « en vrac » quelques suggestions d'extensions possibles :

- **Nouvelles unités et actions** Toutes sortes d'unités peuvent être envisagées avec des comportements spécifiques et impliquant des modalités d'actions différentes de celles envisagées jusqu'ici. Par exemple, vous pouvez imaginer des unités capables d'interagir à distance avec des entités du jeu (cités, autres unités). Une interaction à distance pourrait être une unité empêchant les unités ennemies situées deux cases devant elle de bouger etc. Voir https://advancewars.fandom.com/wiki/List_of_Units pour des sources d'inspiration.
- **Nouveaux acteurs « non-unité »** à l'image des cités par exemple (forêt, marais, montagnes, grottes etc.).
- **Nouveaux type de cellules** avec des impacts spécifiques sur les unités.
- **Ajout d'animation** (voir à ce propos la section 6.6 du tutoriel).
- **Nouvelles cartes** qui pourraient être associées à des « missions » spécifiques (voir par exemple https://warswiki.org/wiki/List_of_Missions_in_Advance_Wars).
- **Différencier le rayon d'attaque et de déplacement** et pouvoir les faire varier en fonctions de certaines conditions.
- Introduire des zones où il n'est pas possible d'aller ou d'interagir (zone occupée par une cités brûlées par exemple) ou qui ont des propriétés particulières (forêts dans lesquelles peuvent se cacher les unités, marais etc..).
- **Cycle « jour-nuit » ou « météo »** qui impacte le rayon d'attaque par exemple.
- **Gestion du « Game Over »**mettre en place une gestion visuelle de la fin de partie (avec affichage de scores par exemple).

- **Ajout de son** : la maquette prévoit des outils simple pour la gestion de l'audio.
- **Support pour plus de deux joueurs** :vous veillerez cependant à ne pas inclure de fonctionnalités réseau qui dépassent de trop loin la portée de ce cours.
- **Améliorations de L'IA** pour lesquelles voici quelques suggestions envisageables :
  - Améliorer la logique de déplacement pour qu'elle choisisse en priorité les cases offrant le plus de défense quitte à se rapprocher d'une unité adverse.
  - Toujours pour la logique de déplacement, au lieu de chercher simplement à maintenir à portée d'attaque l'unité adverse la plus proche, une amélioration pourrait être de chercher à se rapprocher des unités positionnées sur une case leur offrant peu de protection.
  - On pourrait concevoir une amélioration de l'action `Attack` où l'unité ne cherche pas à attaquer l'unité avec le moins de vie mais celle où elle peut faire le plus de dégâts en prenant en compte, cette fois, les `defenseStars` des cellules.
  - Si les cités et action de capture ont été introduites, une amélioration pourrait être de parfois favoriser la capture de cités pour décider des déplacements. Par exemple, lorsqu'une ville non capturée est plus proche qu'une unité adverse.

  Des IA beaucoup plus complexes peuvent êtres imaginées (par exemple avec des stratégies prévoyant des scénarios sur 2 tours). Cependant, cela peut rapidement s'avérer très exigeant.

Vous pouvez, de manière générale, laisser parler votre imagination, et essayer vos propres idées. S'il vous vient une idée originale qui vous semble différer dans l'esprit de ce qui est suggéré et que vous souhaitez l'implémenter pour le rendu ou le concours (voir ci-dessous), il faut la faire valider avant de continuer (en envoyant un mail à CS107@epfl.ch).

> Attention cependant à ne pas passer trop de temps sur le projet au détriment d'autres branches !

5.3 Validation de l'étape 4

Comme résultat final du projet, créez un scénario de jeu impliquant l'ensemble des composants d'extension codés. Une (petite) partie de la note sera liée à l'inventivité et l'originalité dont vous ferez preuve dans la conception du jeu.

6 Concours

Ceux d'entre vous qui ont terminé le projet avec un effort particulier sur le résultat final (gameplay intéressant, effets visuels, extensions intéressantes/originales etc.) peuvent concourir au prix du « meilleur jeu du CS107 ».[7]

Si vous souhaitez concourir, vous devrez nous envoyer d'ici au **23.12 à 18 :00** un petit « dossier de candidature » par mail à l'adresse **cs107@epfl.ch**. Il s'agira d'une description de votre jeu et des extensions que vous y avez incorporées (sur 2 à 3 pages en format .pdf avec quelques copies d'écran mettant en valeur vos ajouts).

[7] Nous avons prévu un petit « Wall of Fame » sur la page web du cours et une petite récompense symbolique :-)