CS-107 : Game engine elements

J. Berdat, B. Châtelain, Q. Juppet and J. Sam Version 1.0.0

This English version of the tutorial was generated using a tool for automatic translation. **The reference document remains the French version**. In case something is unclear in the English version please refer to the French one or ask questions

Table des matières

1 Introduction

To implement the second mini-project, the course provides you with a simple toolkit. This toolkit is a basic game engine for creating 2D games, in particular grid-based games [Link] which can be adapted to a wide range of variants, inspired by examples such as :

Pokémon Emerald [Link]

Super Pacman [Link]

The time and knowledge required to implement an entire game of this type is indeed beyond the scope of our course. Moreover, learning how to work with and exploit existing code is an integral part of learning object-oriented programming.

The aim of this document is to help you understand the contents of the toolkit provided. It will then serve as a basis for the implementation of your second mini-project.

2 Overview of the toolkit's architecture

The architecture of the toolkit is outlined in the diagram in Figure 1.

A brief description of the packages supplied is given below. The code and its documentation, together with this tutorial, should answer the details, giving you the opportunity to access code from more experienced programmers[1].

You are not required to consult this material now, but to return to it as you read this tutorial, and as you code the mini-project, as and when you need to, according to our indications. You'll also find some useful additional information in the appendices (see section 7.1).

- The `io` package contains a number of utilities for handling file-based I/O. Typically, the images that will be used to represent the entities populating our games are stored in files, and these utilities will make it possible to read these files and exploit them.

- The `math`package models mathematical concepts such as vectors, affine transformations and random variables. This package includes notions of plane (i.e. two-dimensional) geometry. For example, it explains what a shape is, and more specifically what a line, circle or polygon is, which you can then use in mathematical calculations or when representing graphical elements. Points and sizes are always given in floating values, to get as close as possible to the continuous geometric plane we're trying to simulate. To maintain consistency with basic geometric notions, the vertical axis *oy* will by definition be oriented upwards and the horizontal axis *ox* to the right.

[1] Always in the spirit of learning by example, even if it's not required to understand all the code provided in detail.

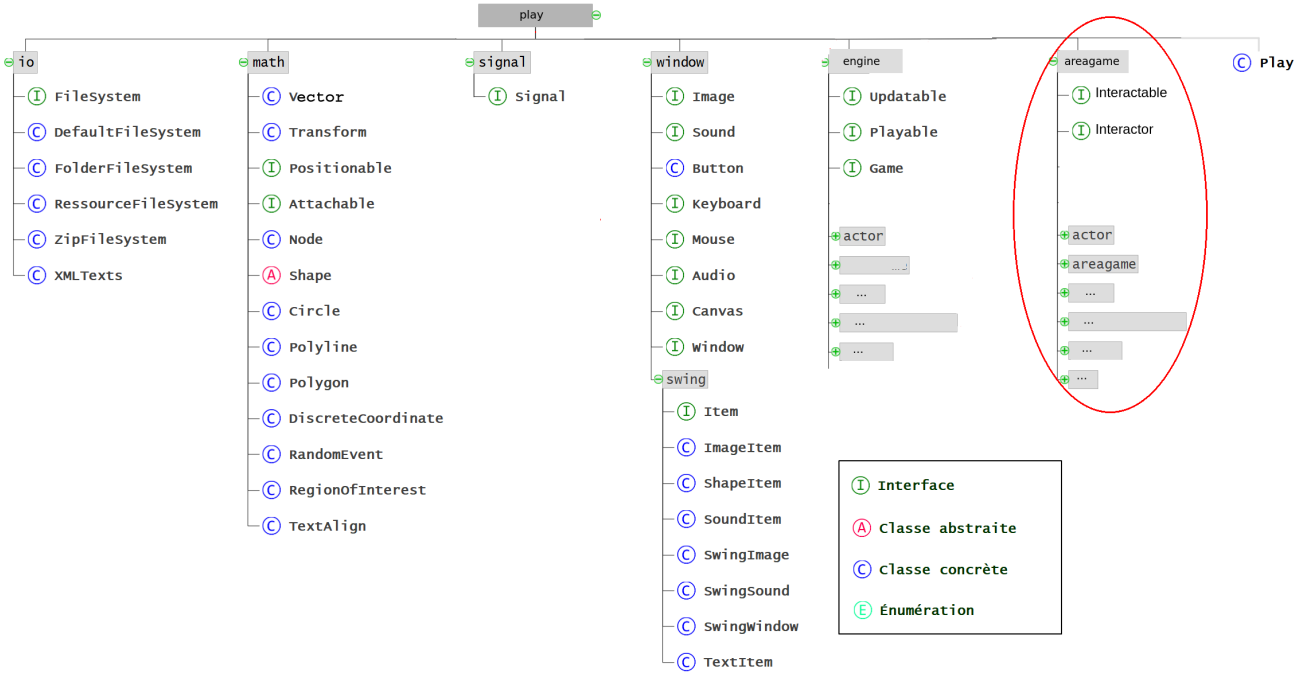


FIG. 1 : Main project packages. This tutorial will essentially introduce you to the `game`, `areagame` and `signal` packages.

- The package `window` : provides the abstractions `Window`, `Canvas` (drawing area), `Mouse`, `Keyboard` etc. (modeling the basic elements of the graphical interface. The `SwingWindow` class in the `swing` directory is a concrete realization of the window concept based on Java Swing components. Objects with access to the canvas can request the drawing of an image, shape or text. A request adds a graphic item of the corresponding type to a list which will be sorted and rendered/drawn the next time the window is updated. The list is then emptied, pending requests for the next update. Similarly, objects with access to the audio context can request that a sound be played. This represents a dynamic video game window that will probably (for most games at least) be refreshed at a relatively high frequency (20 to 60 times per second). These drawing requests for graphical items will therefore need to be reiterated regularly, ideally once before each refresh, for as long as they need to be rendered. The graphic result obtained at the end of each update is called *frame*. We're talking about between 20 and 60 frames per second.

- The `engine` package contains fundamental abstract elements of the game engine (mainly abstract "actor" models).

- The `areagame` package builds on the matreial of the package `engine` to model games playes on a grid-. It is the one that will keep you busy throughout this tutorial. The work to be carried out in the mini-project consists of using the contents of the `areagame` package and extending it.

- The `signal` package will be used to include signal-related components, mainly logic.

Under this link, you'll find the javadoc of the code provided.
This tutorial is structured in 3 parts, which should be read according to the schedule suggested in the exercise statements :

- the first part describes the modeling of the notion of a game on a grid, which involves defining what a *game area* is and its associated *grid* ; it also describes the concept of *actor* at an *abstract* level ;
- the second part explains how the concept of *interface* is used to model these elements at a sufficiently abstract level to make them usable in different contexts ; it also describes more precisely the notion of grid and the specificities of actors evolving on grids ;
- the last part describes the schema used to implement interactions between actors ; again at a sufficiently abstract level ; it also illustrates how the notion of nested classes is used for better encapsulation in certain parts of the code.

3 Setting up mini-project 2

To follow this tutorial, you need to install the code provided in an Intellij project. You'll do this straightaway in the "mini-project 2" project, which you can then complete directly when the time comes. You'll find the necessary instructions below.

For the tutorial and the project, and to avoid rendering problems, be sure to set your IDE to UTF-8 (see "Setting character encoding" in the IntelliJ configuration guide or the Eclipse configuration guide).

The material provided can be found in the archive : tutoriel-2023.zip
To install the projector on IntelliJ :

1. unzip this archive in a directory of your choice ;
2. opem in IntelliJ the directory `TUTO-2023` ;
3. delete the archive `tutorial-2023.zip`.

For Eclipse, once the archive has been unzipped, create the project using the `"From existing sources "` option, specifying the host directory as the project root.

- The material provided consists of two folders : `game-engine` which contains the tools offered by the model and `tutos` which is the folder where you will do the exercises.
- **It is normal for the project not to run properly** : the game to be launched is initialized to `null` in the `Play.java` file in the `tutos` folder. You will correct this issue as you progress through the tutorial.
- To do the exercises, you will work in subpackages of `ch.epfl.cs107.play` **from the `tutos`** folder.
- **You will not modify the contents of the `game-engine`** folder.
- Simple rule to remember :
  - The contents of the model should be consulted in the packages of `ch.epfl.cs107.play` **of the `game-engine`** folder.
  - The graphics resources provided are in the folder `src/main/resources` **of the folder `tutos`**.
  - The exercises are to be done in subpackages of `ch.epfl.cs107.play` **of the `tutos`** folder.

The 3 parts of the tutorial presented in the following, as well as the statement for mini-project 2, will show you what you need to consult and complete.

4 Tutorial I

The aim of this first tutorial is to get you started investigating the concepts provided by the model at your disposal. It explains in broad terms what a "game" is in the chosen design and how its evolution over time is achieved by the simulation loop of the main program supplied, `Play` It also explains how a *play area* is modeled, and the very rudimentary *actors* that can take part in it. It is recommended to open the relevant code, essentially in the `engine` and `aregame` package, and examine it in parallel with reading the explanations given. Once you've familiarized yourself with the concepts, you'll be asked to code a rudimentary outline of a game as an exercise.

4.1 `Playable`

The model supplied uses the abstract concept of a "playable element" (`Playable`)[2], which can be a complete game, an area within the game, etc. This is a fairly high-level abstraction, meaning that for a program element to be "playable", it must be able to :

1. *evolve* over time (i.e. have an update method :
   `void update(float time)`) ;

2. *start* properly (that is to say, initialize in particular by incorporating all the entities that are likely to evolve there) ; this requires access to a graphical context/window and a file system to fetch resources, such as images for example (presence of a method :
   `begin(Window window, FileSystem filesystem)`) ;

3. and *end* cleanly ; that is to say, implement a certain number of actions which characterize its end ; this can be an end message appearing on the screen or any other relevant action (presence of a `end` method).

A "playable element" will also be characterized by a *name*, a character string returned by a `getTitle()` method.

The concept of `Playable` is coded using the notion of *interface* Java and we will come back to it when this has been covered in class. For the moment it is only the concept as such that interests us and not its concrete implementation. For the moment, think of it as an abstract class incorporating the elements mentioned above. Note that `Game`, which models the notion of "game" is a kind of `Playable` with the addition of a refresh rate.

4.2 Simulation loop

Someone wishing to "launch" a "game" should then proceed as in the main example program provided in `Play.java` :

- Create a game instance (line 33 currently commented), a file system (line 30) and a window/graphical context (line 37).

- Then, launch the game with `begin` by passing the file system as parameters to connect it to the outside world and the window to give it access to a graphic (and audio) context.

[2] sub-package `engine` of the toolkit `game-engine`

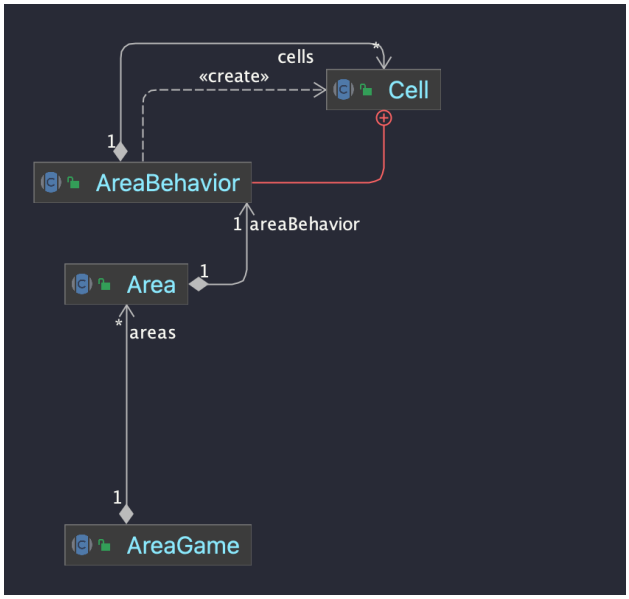


FIG. 2 : A `AreaGame` *a-a* set of `Area`. Each `Area` *has-a* grid (`AreaBehavior`). Each grid is made up of cells (`Cell`) which it is responsible for creating.

- Once the game is launched, and depending on the requested refresh rate, the game and the window will be updated one after the other (lines 72 and 75). Refreshing the window consists of redrawing its content from a list of graphical items emptied after each iteration. It is the role of the game, during its own update with its `update` method, to make drawing requests (and sound) to supply this list in anticipation of the next frame. For the game, updates consist of first refreshing all of its components (e.g. repositioning them) based on the time since the last call, then executing requests to redraw them (and playing sounds if necessary) .

When the game should stop executing its `end` method must be invoked (line 81).

4.3 Grid games

The tools provided in the `engine` package allow you to implement games wherever you want :

- that an indeterminate number of actors can intervene (characters, characteristic objects, etc.) ;
- that the playing area of the game is not confined to the physical window alone ;
- and that the actors can interact physically in a simple way (use of simple concepts to manage physical collisions and characterize the areas of interaction between actors).

Games that take place on a grid offer the advantage of significantly simplifying the last point (management of collisions and interactions) and it is on this basis that we will start with the coding of this year's mini-project[3].

[3]note that the model also potentially allows the programming of "games" detached from the notion of a grid

This tutorial will therefore essentially describe the classes made available in the model to model "grid games".
To summarize :

- `AreaGame` : "grid game" concept
- `Area` : area of `AreaGame`. A `AreaGame` potentially contains several `Area`
- `AreaBehavior` : grid associated with a `Area` (each area of the game has a characteristic grid).

Concretely, as shown in the figure 2, the code of the model makes it possible to model games on a grid (`AreaGame`) composed of a set of zones (which often correspond to levels) called `Area` (section 4.4). Each `Area` corresponds to a two-dimensional grid composed of cells (`Cell`) in which actors/components (`Actor`) will evolve and interact.
With the aim of visually resembling certain *Game Boy* type games, and as said above, this grid is introduced in particular to simplify the management of interactions between actors as well as their movements : this is the cell and its content which condition interactions (as opposed to an approach managing interactions by detecting "physical" collisions between actors in a completely continuous model of the world, for example).
An `Area` is therefore in a way a more or less independent game, that is to say a "playable element" too, with a set of actors. To avoid overloading this class, it does not have direct knowledge of the grid that defines it. It delegates this knowledge as well as all its functionalities to a class `AreaBehavior` (section 5.2). Therefore, each `Area` has an `AreaBehavior` which manages the behavior and mechanics of the area with its grid, its cells and their content.
In the following you will find some explanations concerning the `Area` classes and `AreaGame`. The explanations on `AreaBehavior` will be given to you in tutorial II (section 5).

4.4 Play Areas : class `Area`

The abstract class `Area` provided in the `areagame.area` package models a (playable) area in a grid game. Open Javadoc and search for this class (small search window at the top right)[4] to examine it in broad terms. Also feel free to open the code itself as provided.
A `Area`, is a "playable" element, and will therefore naturally have the methods `begin`, `end` and `update` mentioned above.
An area has a list of actors (those who operate there, `Actor`). It has an associated grid (`AreaBehavior`) which crisscrosses the playing area and on which the actors will move.
Among the fundamental methods to remember :

- `begin` which carries out all the initializations necessary for starting a playable area ;
- `update` which makes the actors evolve (by invoking their own `update` method), plays their possible sound effects and draws them (actors' `beep` and `draw` method) ;
- `registerActor` and `unregisterActor` which allow you to add/remove an actor from the list ;

[4] do not hesitate to use the Javadoc for the rest of mini-project 2

- `end` which carries out all the actions to be carried out when the game on the current area ends.

4.4.1 Transition from one area to another

The games that interest us are supposed to be composed of several areas of which only one (the current area) will be played at a time. When moving from one area to another, several strategies can be considered : if we return to an area already played before, we can for example either restart the game on this area from the beginning or in the state in which it had been left. To do this, the `Area` offers the following methods :

- `void suspend()` which by default does nothing but which, once redefined, can implement any specific strategy to be implemented when leaving one play area to move to another (such as possibly saving information on the state of the game play area) ;
- `boolean resume(Window window, FileSystem fileSystem)` which returns `true` by default but which, once redefined, can allow the game to be resumed on an area from a possible intermediate state where it would have been left. The return boolean indicates whether resuming play on the area was possible or not.

4.4.2 Camera management

A play area can be large and larger than what is visible in the window. It is therefore necessary to allow the view to be placed at a precise location in a given area and according to a given scale factor. Among the important methods :

- `getCameraScaleFactor()` an abstract method which can be redefined in subclasses to return the desired scale factor (depending on its value the view is more or less "zoomed") ;
- `setViewCandidate` which allows you to center the view on an actor. This is what will allow the camera to follow a character in numerous areas, such as you will be able to experience in your first drafts of the game.

4.5 Games with areas : class `AreaGame`

The abstract class `AreaGame` from the package `areagame` simply models the concept of "game with multiple areas". A game with several areas is also above all a "playable" element and you will find the attributes and methods specific to this concept (attributes of type `Window` and `FileSystem`, and methods `update`, `begin` and `end`).
You will also find there, an attribute allowing you to represent the *set* of the areas which constitute the game and an attribute representing the current playing area (which will be the only one to be simulated) :

```
/// A map containing all the Area of the Game
private Map<String, Area> areas;
/// The current area the game is in
private Area currentArea;
```

As the data structure for all the areas, the `Map` type was chosen. (associative key–value table) which will allow you to find a play area based on its name (`getTitle`) (see appendix 7.1).

> The method `getTitle` of areas is very important because it is through this ”title” that an area is identified (and it is also through this that the link of the area with some of the resources that are useful to it will also be made).

The method `update` just updates the current area.
Among the important methods :

- `addArea()` which allows the dynamic addition of areas to the game ;
- `setCurrentArea` which allows you to choose the current (simulated) area among all the areas ; its second parameter allows you to indicate whether when you move to this area you want to restart it (parameter `forceBegin` to `true`) or continue it where you left it during a possible previous passage).

4.6 Generic actors

The proposed model makes it possible to program games featuring *actors* acting in certain ways. These can have all kinds of variations, ranging from a simple geometric piece (like in Tetris®) to a complex “RPG” character.
You will find in the packages `engine.actor` and `areagame.actor` a certain number of classes (and interfaces) allowing the notion of actors to be modeled in a generic way (see this class schema[Link]). The concept of “actor” is modeled by the entity `engine.actor.Actor` (this is an interface, but think of it as an abstract class for now). For this very abstract model, an actor is simply an entity that evolves over time (`update` method) and that can emit sounds.
The `Entity` class is a particular and basic implementation of `Actor` : it represents an entity with a position, a speed and a reference frame of its own (accessible using `getTransform`). A little additional explanation on the notion of transform and frame of reference is given in the appendix 7.1. In principle, there is no need to understand this concept in depth to start the project.
The first actor you will use is very rudimentary. It will be coded as a subclass of `Entity`.

4.7 Exercise 1 : first “game with areas”

Now is the time to start trying to use the few elements of the model presented so far for yourself. You will code a draft game with areas.

IDEs like Eclipse or IntelliJ are very practical to automatically add missing import directives in a class. This usage is recommended, but be careful, when there are several choices, to include the option that corresponds to your needs and not to include non-standard elements (you should only keep imports starting with `java.` or `javax.`). The model provided uses the `Color` class in particular. The `java.awt.Color` version should be used and not other implementations from various alternative packages.

4.7.1 First concrete actor

Create a subpackage `actor` from the provided package `ch.epfl.cs107.play.tuto1` in which you will code a new actor class called `SimpleGhost`. These actors derive from the class `Entity`[5]. This will be an actor with a graphical representation, that is to say an attribute of type `Sprite` (type provided in the `engine.actor` package).

Game Boy type games often simulate an aerial view known as a top view. To respect the desired effect which dictates that being below implies being in front, the images must be drawn from top to bottom so as not to create contradiction. The `Sprite` are simple graphic images whose depth depends on the `y` coordinate of the entity to which they relate. The `Sprite` also allow you to specify in their constructor which objects they attach to (see the code for this class if necessary).

A `SimpleGhost` will also be characterized by an *energy level* (encoded as a `float`). You will equip it with the methods :

- `boolean isWeak()` returning the boolean `true` if the ghost's energy level is less than or equal to zero ;
- `void strengthen()` returning the energy level to a given positive value (always the same (choose 10 for example) ;

Its constructor will have the following heading :

```
public SimpleGhost(Vector position, String spriteName)
```

`spriteName` is the name of the image associated with the ghost during its construction (this image will be searched in the folder `src/main/resources` by the code of `Sprite`). The `Sprite` associated with `SimpleGhost` can be created using the twist :

```
new Sprite(spriteName, 1f, 1f, this);
```

in the `SimpleGhost` constructor. The parameter `this` allows the constructor of the `Sprite` to attach it to the current object.

The constructor for `SimpleGhost` will initialize the energy level with a default value (choose a value not too high like `10` ... you will see why a little later).

We also want to display the energy level next to the ghost.

To do this you will need to declare an attribute :

[5] we name it `SimpleGhost` because it is derived from `Entity` which is a very low-level class of "actors", it is in fact simply an object that can be positioned in space

```
private TextGraphics hpText;
```

which will be initialized in the constructor using the turn :

```
new TextGraphics(Integer.toString((int)hp), 0.4f,
   Color.BLUE);
```

where `hp` represents the "energy level" attribute.
To ensure that this text is linked to the ghost, and therefore moves with it, it must be attached to it :

```
hpText.setParent(this);
```

The anchor point of the text can be shifted by this kind of turn :

```
this.hpText.setAnchor(new Vector(-0.3f, 0.1f));
```

These two instructions must be placed, once and for all, in the constructor.
Our actor `SimpleGhost` inherits from a method `void draw(Canvas canvas)` inherited from `Entity`. The latter allows us to display on a support such as `Canvas`[6], the image associated with our object. Note that `Sprite` and `TextGraphics` have `void draw(Canvas canvas)` methods.
Redefine the `draw` inherited from `Entity` so that the energy level text is also displayed. Also redefine the `void update(float deltaTime)` method. Its role will be to decrement by `deltaTime` the ghost's energy level ; the ghost cannot, however, have an energy level lower than zero. You will need to remember to update the `hpText` text accordingly.

4.7.2 First concrete play areas

In the `ch.epfl.cs107.tuto1.area` package, you will find a subclass `SimpleArea` by `Area`. This class imposes on its concrete subclasses the definition of a method `createArea` allowing you to create the content of a specific play area. The `getWidth` and `getHeight` will be explained to you in the following tutorial. They are not useful at this stage. You can redefine `getCameraScaleFactor` in class `SimpleArea`. Make it return the value `10.f` for example. This allows you to define a default scale factor to use for all areas of type `SimpleArea`.
Create a subpackage `ch.epfl.cs107.tuto1.area.maps` and create the specific areas `Village` and `Ferme` inheriting from `SimpleArea`.
You will give concrete definitions to the `createArea` method in each of these classes. This method should :

- do nothing at the moment in `Ferme` ;
- create actor `SimpleGhost` in `Village` and register it there (remember the `registerActor` method and any idea of why we don't use `addActor` here instead ? You will use a `Vector(20,10)` as position and the image named *`"ghost.2"`*.

Any specific "game" must specify the name that characterizes it. To do this, you will add to the class `Village`, the redefinition :

```
@Override
public String getTitle() {
  return "zelda/Village";
```

[6] Window derived from Canvas

```
}
```

and you will proceed in a similar way for `Ferme` :

```
@Override
public String getTitle() {
  return "zelda/Ferme";
}
```

Finally, you will note that ”default, default” constructors are sufficient for these two classes.

4.7.3 First game with concrete areas

At this stage we have two areas, one of which is intended to contain a concrete actor. It remains for us to define a game made up of these two areas. Recall that `AreaGame` allows you to model a game with several areas. So define in the `ch.epfl.cs107.play.tuto1` package, the class `Tuto1` inheriting from `AreaGame`. Give it a private method `createAreas()` which aims to add the desired areas to the game. This method therefore simply calls the `ethod!addArea! fro AreaGame`, for example like this to add the area `Ferme` :

```
addArea(new Ferme());
```

Like any playable element, `Tuto1` must define the methods `begin`, `end` , `update` and `getTitle`. You will code them as follows :

- the method `end` does nothing in particular ;
- the method `update` just invokes that of the super-class ;
- the method `getTitle` returns a title associated with the game, like *"Tuto1"* ;
- the method `begin` will use the following phrase :

```
if (super.begin(window, fileSystem)) {
    // treatment specific to Tuto1
    return true;
}
else return false;
```

  Treatments specific to `Tuto1` will consist of :

  – create areas (using `createAreas`) ;
  – and indicate that the current area is the area titled *"zelda/Ferme"* (we want that when passing to this area, it is restarted, the parameter `forceBegin` will therefore be worth `true`)

Question 1

What happens if we forget to invoke the `update` of the superclass in the `update` by `Tuto1` ?

All that remains is to indicate in the main program `Play`, that you wish to launch the playable element `Tuto1`.
To do this simply add the line

```
final AreaGame game = new Tuto1();
```

before the comment

```
// Use Swing display
```

Remember that the main program calls the `update` method in a loop of the simulated game (here `Tuto1`), which calls the `update` of its current area, which in turn calls the `update` of each of its actors. This is how the simulation can evolve over time.

Ready for the big leap ? run your `Play` program.

If everything goes well . . . a completely empty window is displayed. But where has our ghost gone ?

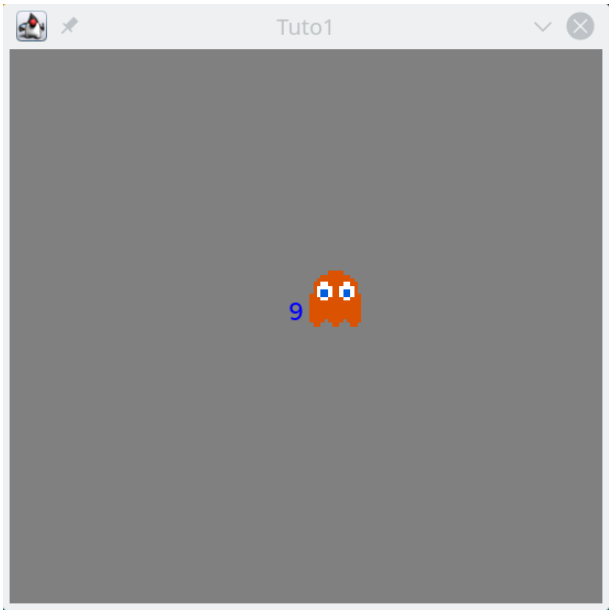
Answer : it is in the area that is not displayed. Indeed, we indicated that our current area (the only one simulated) was "`zelda/Ferme`" and our ghost is in the area named *`"zelda/Village"`*. You can ensure this by changing the current area in `Tuto1`. You should see the ghost's energy level decrease over time and stabilize at zero.

4.7.4 Main character

We are interested in coding games where we control a main character through his wanderings in areas of the game. This actor will have a very particular status and we will consider that he is one of the characteristic elements of each game. Add to your game `Tuto1`, an attribute

```
private SimpleGhost player;
```

who will fulfill this role. In the method `begin` of `Tuto1`, start by giving a value to the main character by creating a `SimpleGhost` positioned in `(18, 7)` and with the image name *`"ghost.1"`*. Then register the main character in the current area (once it has been designated) and indicate that the camera should now follow him (remember that `setCurrentArea` returns the current area and that the `setViewCandidate` allows you to focus the camera on a particular actor). When *`"zelda/Ferme"`* is the game's starting area (the one chosen in `begin`), you should get this when launching the game :



Note : The wallpaper color can be changed in the file `SwingWindow` from the `window.swing` package, line 186.

4.7.5 Controls

At the moment we have little insight into the main character. It's time to fix it. To do this, program in your class `SimpleGhost` methods `moveUp`, `moveDown`, `moveLeft` and `moveRight` allowing it to move a small fixed distance (for example `0.05`), in a given direction. This is how `moveUp` can be encoded for example :

```
setCurrentPosition(getPosition().add(0f, delta));
```

we use the methods `setCurrentPosition` and `getPosition` specific to `Entity` and which inherits `SimpleGhost` as well as the `add` specific to the provided utility class, `Vector`. You will proceed in the same way for the other methods.

Then modify the `update` of your game `Tuto1`, so that it is receptive to keyboard events. If the user presses "up-arrow" on the keyboard, it is the `moveUp` method of the main character who is called. If he presses "down arrow" it's `moveDown`, if he presses "left arrow" it's `moveLeft` and on "right-arrow" it's `moveRight`.

Here is how the API provided allows you in this context to test that the up-arrow has been pressed :

```
  Keyboard keyboard = getWindow().getKeyboard() ;
  Button key = keyboard.get(Keyboard.UP) ;
  if(key.isDown())
{
  //...
}
```

(and we proceed in a similar way with the values `Keyboard.DOWN`, `Keyboard.LEFT` and `Keyboard.RIGHT`).

If you launch the game again, however, you will have the impression that these keys have no effect. This is normal, because the camera refocuses on the main character each time he moves and as the background is solid, you don't have the impression of seeing him move.

To make the movement perceptible, we would need, for example, a wallpaper with a setting and we would then see the character move, with reference to this setting. Another way to do it is to place it in an area where there is an element that does not move. This is the case for our *"zelda/Village"*`area` where the second ghost is not controllable via the keyboard and does not move.

To finalize the exercise, program in `Tuto1`, a method `void switchArea()` which allows the main character to move from one area to another : if he is in the area *"zelda/Ferme"* he should move on to the one titled *"zelda/Village"* and vice versa. Each time he leaves an area he must be deregistered there. When he enters an area, the latter must become the current area, the character must be recorded there and the camera centered on him. When passing through another area, the main character will see his energy level increased (`strengthen()` method). The transition from one area to another must be done automatically as soon as the ghost becomes weak (`isWeak` method).

If you did things correctly, you should see the main character (orange ghost) appear by himself when you launch the game, then his energy level gradually decreases until it reaches

zero. At this time, he must transit towards the area where there is the blue ghost. In this area, the arrows can make it move visibly (you will have the impression that it is the blue ghost that is moving because the frame of reference is always centered on the orange ghost).

Note : in your coding of the `switchArea` method, you probably called `setCurrentArea`. Experiment with calling it with `false` or `true` as a second argument (you should see an impact on the blue ghost's energy level). With `false` we find the area in the state where we left it and in the second we recreate it from scratch. Finally, you can play with the scale factor to see how it affects the game's display.

This little exercise ends this first tutorial. In principle, it allowed you to familiarize yourself with the model for the modeling of a game made up of several areas as well as for the modeling of simple actors. The objective of the second tutorial is to start using the grids associated with areas.

5 Tutorial II

This second tutorial looks at the use of interfaces in the model provided. It also presents in more detail the notion of a grid associated with a playing area and the specific actors who can take place there. As with the first tutorial, small exercises will allow you to use the concepts presented for yourself.

5.1 Let's talk a little about interfaces

Now that the notion of interface has been introduced in class, you can start to take a closer look at their use in the model provided. The `Game` models the abstract notion of "game". It is implemented more finely using the `Playable` and `Updatable` interfaces which we invite you to examine (they are in the `game` `package`).

Question 2

Why do you think it is better to declare the variable `game` of the `Play` program, as a `Game` rather than as a `AreaGame` ?

Response element : `Game` represents the concept of "game" from the functional and abstract point of view. `GameArea` is just one possible implementation of this concept. If we declare `game` like a `AreaGame` (grid game), the main program `Play` sees much more of this object than its abstract functional representation, "game". It sees all its *implementation* details as a grid game. This program can then use these details at will (and misuse) in its own implementation. This potentially induces unfortunate encapsulation flaws. For example, what if `Play` decides to run a game that is not an area game and used `AreaGame` specific in its `main` ?

You will therefore note that interfaces are a powerful encapsulation tool : the area, the grid and the actors need to know each other, which implies on their part to open access to certain information. Typically, a play area must have access to the actors and the actors must know in which game they are playing (potential encapsulation flaws). However, if as a user, we impose the discipline of only seeing a game's logic of use as dictated by `Game` (as is the case with `Play` for example), then sensitive access is no longer exposed.

In the same spirit, the `Actor` allows you to model in a minimalist and abstract way the functional aspects of an actor in a game, without necessarily having to expose the API of its possible implementations. To do this, examine the contents of `Actor` in `engine.actor`. `Actor` models a very simple abstract actor. The `Entity` class of the same package is a possible implementation from which all kinds of other specific implementations will derive. To protect the codes from possible modifications in specific implementations, it is necessary to avoid exposing the latter. Manipulate any actor under the label of `Actor` rather than a specific implementation allows us to achieve this goal.

To complete your knowledge of the model, you can now return to the `implements` links of the classes already presented, such as `Area` or `AreaGame` for example.

FIG. 3 : Example of a behavioral image with color-type correspondence

Let's continue our exploration of the model. The areas in our first game are a little dark. To remedy this, we're going to unveil the notion of a grid associated with an area (`AreaBehavior`), as well as the `Background` class. The latter will give the grids in question a slightly less sidereal look.

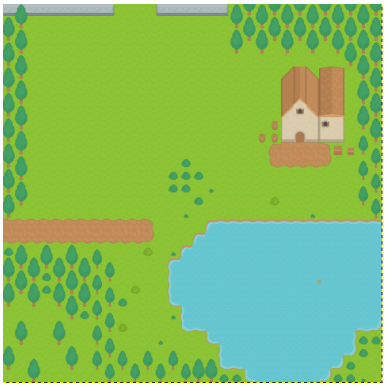
5.2 Grid and cells

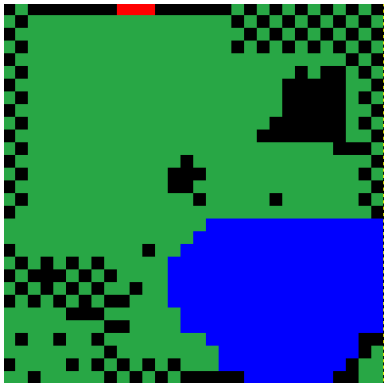
We saw in the previous tutorial that the `Area` class has an attribute of type `AreaBehavior` This attribute models a *grid* that will condition the behavior of everything that takes its place. Open the code for the `AreaBehavior` class. You'll see that its attribute is an array of cells (`Cell`), as well as an image (the `behaviorMap` attribute). The constructor of `AreaBehavior` (and its concrete sub-classes) is prompted to initialize the array of cells from a color image like the one shown in figure figure 3, where each pixel represents a cell and each color a different cell type. We'll explain a little later how this correspondence is set up and how it should be used.

The `Cell` class represents a generic cell and each extension of `AreaBehavior` that will be specific to a given game will have to redefine specific extensions of the `Cell` class.

Note : For the moment, the grid and the cells are two autonomous entities. This is not the best possible design, and we'll come back to it in the final part of this tutorial.

A `Cell` has a content (the set of game entities occupying the cell), but for the moment we're not interested in it. Suffice it to say that it is characterized by its coordinates on the grid (of type `DiscreteCoordinates`).


background image


corresponding "behavior"

In the `AreaBehavior` constructor, the line :

```
window.getImage(ResourcePath.getBehaviors(fileName), null,
    false);
```

simply allows you to read an image from a given file name.

Each concrete play area is of course associated with a concrete subclass of `AreaBehavior`. This association will not necessarily be unique (it's conceivable that an area could change its associated grid during its existence), and the model therefore allows the modification of the `AreaBehavior` associated with an `Area`. This explains the presence of the `setBehavior(AreaBehavior ab)` method in the `Area` class.

5.3 Actor for "wallpaper"

Each play area is assigned a "background" to define its visual appearance. It's conceivable that such a visual may change over time (for example, different visuals for a night/day cycle). Instead of coding the background associated with an area as a fixed attribute of the class `Area` class, you'll have to code it as ... an actor, using the elements of the model. This actor is much more similar in spirit to the one you coded in `Tuto1` : it's simply a derivative of `Entity` just like `SimpleGhost`. The code is provided in the `Background` class of the `engine.actor` package. Start by taking a look. You'll notice that, by default, the image associated with this graphic actor is the one whose file name corresponds to the name of its area as returned by the `getTitle()` method. For example, this would be `Village.png` from the folder `resources/images/background/zelda` if `getTitle()` returns *`"zelda/Village"`*.

To be able to adjust its size to that of the area on which it is applied as a background, the actor `Background` needs to know the dimensions of this area. This is the purpose of the `getWidth()` and `getHeight()` methods of the `Area` class. The width of the area is that of its associated grid (`AreaBehavior`), i.e. the number of rows in the associated cell array. Similar reasoning applies to the height. As you'd expect, there's a link between the appearance of the background and the image that describes the behavior of the grid, as shown in figure 5.3.

The `resources` directory provides some backgrounds in the `src/main/resources/images/background` directories and associated "behavior" images in `src/main/resources/images/behavior`

(the correspondence is established through the name). Appendix 7.2 also provides a tool for creating associated images `background` and `behavior` [7].

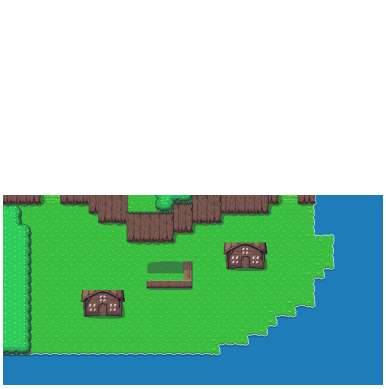
5.3.1 Exercise 1 (continued : adding a wallpaper)

`SimpleArea` is a kind of playground that doesn't really exploit the associated grid, but only allows you to take its size into account. This will enable you to introduce your first "background" actors.

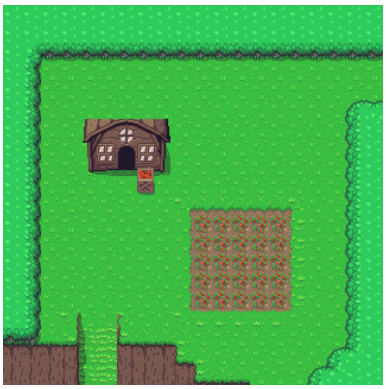
Complete the specific areas `Village` and `Ferme` of the `tutos.area.tuto1` package to ensure that their `createArea` method adds the associated "wallpaper" actor. This is typically done using the instruction :

```
registerActor(new Background(this))
```

If you launch the game `Tuto1` again using `Play`, you should see the following wallpapers displayed (only partially, due to the scale actor) :



"village" wallpaper



"farm" wallpaper

The use of arrows now gives the impression of seeing our red ghost move (as the view focuses on him, the background is not always displayed in the same place). Note that you can also register an actor `Foreground` on the same principle (test the effect of such an addition to understand what might motivate it).

It's a good start, but there's still work to be done! We'd like to stop our ghost getting lost in the void when he leaves an area. We'd also like the area's graphic visual to have an impact : for example, for the area to change when the ghost transits an area whose visual is that of a door/passageway, or for us to be able to prevent the ghost from walking on a visual of a body of water. Now it's up to you to learn how to do this in a new exercise.

5.4 Exercise 2 : first grid game

The aim of this exercise is to create a variant `Tuto2` of `Tuto1` where concrete grids are associated with areas. It's a rough RPG where our ghost moves around on a grid that dictates where he can and can't go. You'll be working in the `ch.epfl.cs107.play.tuto2` package.

As a warm-up, create the game `Tuto2` whose content will for the moment be virtually identical to that of `Tuto1` (but don't forget to adapt the `getTitle` method, which should return *`"Tuto2"`*).

[7] but you're not required to use it

5.4.1 Specific grids

The `AreaBehavior` class is a very general and abstract way of modeling the grid attached to a play area. Now we need to code a specialized version, allowing specific cell management. To do this, you'll be asked to code a subclass in the `ch.epfl.cs107.play.tuto2.area` package, `Tuto2Behavior` inheriting from `AreaBehavior`

This subclass will give a particular interpretation to grid cells according to the color associated with them in the corresponding `behaviorMap`

To do this, define the enumerated type in `Tuto2Behavior` :

```
public enum Tuto2CellType {
  NULL(0, false),
  WALL(-16777216, false),       // #000000 RGB code of black
  IMPASSABLE(-8750470, false), // #7A7A7A, RGB color of gray
  INTERACT(-256, true),         // #FFFF00, RGB color of yellow
  DOOR(-195580, true),          // #FD0404, RGB color of red
  WALKABLE(-1, true),;          // #FFFFFF, RGB color of white

  final int type;
  final boolean isWalkable;

  Tuto2CellType(int type, boolean isWalkable){
    this.type = type;
    this.isWalkable = isWalkable;
  }
}
```

Add to this enumerated type the method `static Tuto2CellType toType(int type)` returning the value of the enumerated type corresponding to the integer `type`. For example, `toType(-195580)` will return the value `DOOR`. The value `NULL` will be returned if `type` doesn't match any expected value for the enumerated type.

The type `Tuto2CellType` will allow us to interpret the color red [8] as a door, black as a wall, gray as an impassable zone (like water, for example) and so on. If you open the `src/main/resources/behavior/zelda/Village.png` file, the "behavior" image associated with the *`"zelda/Village"`* area, you'll see how this enumerated type can be used to codify the role of each grid cell :

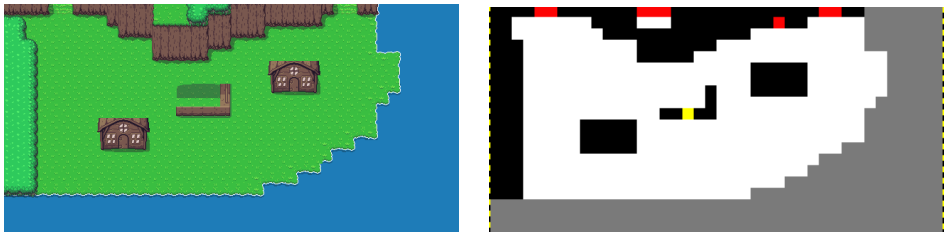
The idea is that if we associate a `Tuto2Behavior` with an `imageBehavior` like the one on the right of figure 3then the cells corresponding to the black pixels can be seen as walls that must not be stepped on, and the cells corresponding to the red pixels as points of passage that can be used to move from one area to another, and so on.

It is therefore necessary to define the cells associated with `Tuto2behavior` in such a way as to enable this grid to dictate specific constraints according to their nature. To do this, at the same level as `Tuto2Behavior` define the subclass `Tuto2Cell` inheriting from `Cell`. A `Tuto2Cell` will be characterized by its type (of the type `Tuto2CellType`).

You'll add a header builder to `Tuto2Cell`

```
Tuto2Cell(int x, int y, Tuto2CellType type)
```

[8] https ://stackoverflow.com/questions/25761438/understanding-bufferedimage-getrgb-output-values



background image corresponding "behavior"

FIG. 4 : "Village" area and corresponding "behavior"

(Note : this constructor can be made private when using nested classes).

Finally, equip `Tuto2Behavior` with a constructor that allows you to initialize the grid by filling it with `Tuto2Cell` coordinates. To find the type to associate with the `Tuto2Cell` coordinates `[x][y]` during its construction, you can use the following formula :

```
Tuto2CellType cellType =
   Tuto2CellType.toType(getRGB(height-1-y, x));
```

Indication : The values of an enumerated type are returned by the method `values()` (here `Tuto2CellType.values()`) and it is of course possible to iterate on them with a value set iteration (`for (Type val : setOfType)`).

`Tuto2Cell` inherits from `Cell` but must be concretely instantiable. You'll find it's always possible to get out of them. For now, code the fact that it's always possible to enter (we'll come back to this later, so that the conditions depend on the nature of the cell).

As `Cell` implements the `Interactable` interface, the compiler will require you to define the methods `isCellInteractable()` and `isViewInteractable()` you can make them return `true` and `false` respectively (but this isn't really important at this stage and we'll come back to it). For the `void acceptInteraction(AreaInteractionVisitor v, bool isCellInteraction)` method, also required by the `Interactable` interface, simply leave an empty body for now.

5.4.2 Specific play areas

In the `ch.epfl.cs107.play.tuto2.area` package, create a `Tuto2Area` class representing our first play areas associated with specific grids. This class will be similar to the supplied `SimpleArea` with the following differences :

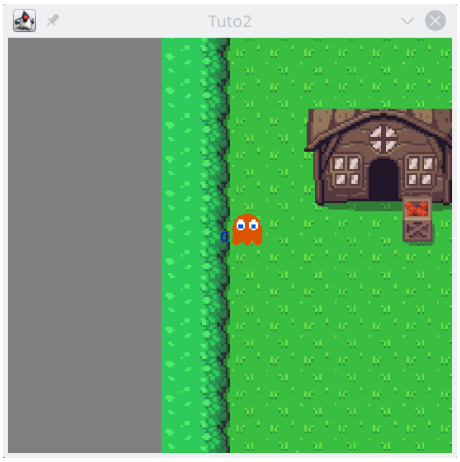
- it doesn't need to redefine the `getWidth` and `getHeight` methods, as those inherited from `Area` suit it well (its width and height are those of the associated grid) ;
- its `begin` method must associate it with a `Tuto2Behavior` type grid :

```
setBehavior(new Tuto2Behavior(window, getTitle()));
```

Finally, create the specific areas `Village` and `Ferme` in a `ch.epfl.cs107.play.tuto2.area.maps` package, almost identical to their versions in `ch.epfl.cs107.play.tuto1.area.maps` but this time inheriting `Tuto2Area`. These are two specific areas to which we can associate a specific grid of type `Tuto2Behavior`.

5.4.3 `Tuto2`

Take your game draft `Tuto2` and make the two areas that make it up the `Village` and `Ferme` areas of `ch.epfl.cs107.play.tuto2.area.maps` (not `ch.epfl.cs107.play.tuto1.area.maps`!). By default, the current area is `Ferme` from `ch.epfl.cs107.play.tuto2.area.maps`. Start the game `Tuto2`. If all goes well, you should see the `Ferme` area (partially) displayed :



Comment on the creation and registration of the actor in the `begin` method, and make sure that the `update` method contains only the call to the `update` method of the superclass (we'll have to change the type of actor and the way it evolves in the rest of this exercise). The little ghost we used as an actor in `Tuto1` game is actually just an image with a position. To enable it to take account of the presence of the grid in order to move correctly, we need to make it a little more sophisticated. To do this, we need to make use of the more advanced types of actors offered by the model, which are presented here.

5.5 Grid game players

We already have a very generic way of modeling actors in a game, using the `Actor` interface and the abstract `Entity` class. We'll now see how this modeling is extended in the model to incorporate the specifics of actors *evolving on a grid*. The classes described below are in the `areagame.actor` package.

5.5.1 La classe `AreaEntity`

The abstract class `AreaEntity` is used to model players belonging to a gridded play area. Their main specificity is that *they occupy cells* of this grid. In general, they can occupy several cells, but only one will be used to locate them, which we'll call the *main cell*. Actors in a gridded play area also have an *orientation*, which will allow them to be drawn differently depending on where they're moving towards. Finally, we start from the fairly natural design assumption that such an actor can "see" his neighborhood and therefore has knowledge of the area to which he belongs.

The `Orientation` type is provided in the `math` package.

You'll note that the `void setCurrentPosition(Vector v)` method inherited from `Entity` has been redefined to adapt to the fact that we're now working on a grid (so that we can also

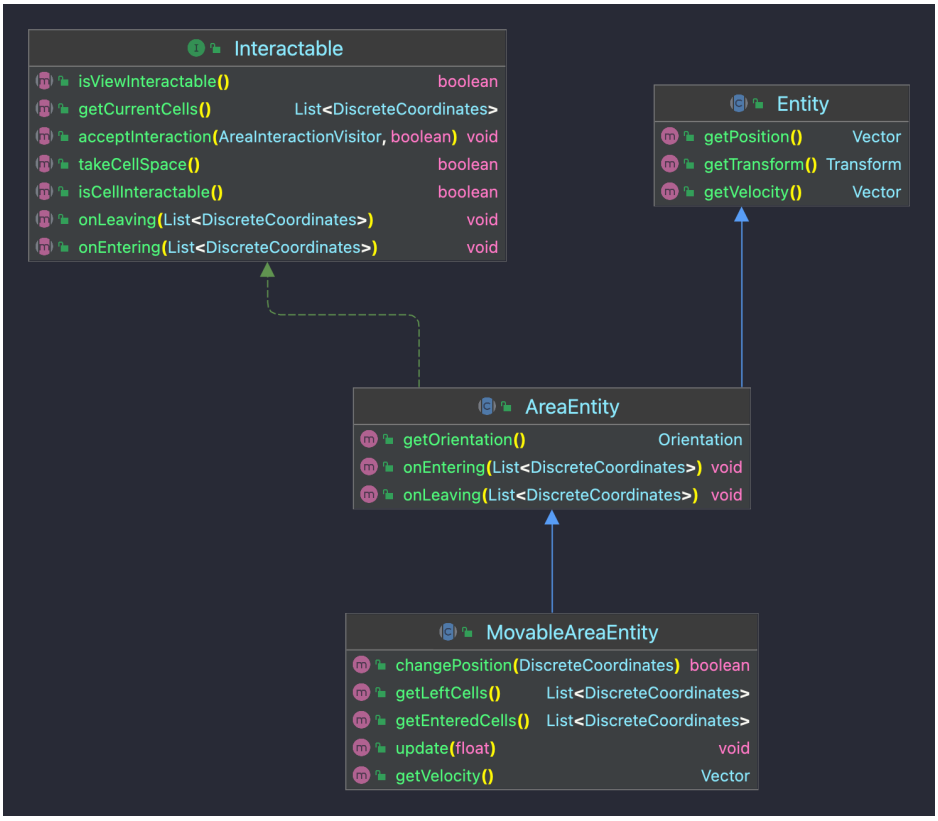


FIG. 5 : Hierarchy of grid-based game players

update our main cell). We haven't coded the actor `Background` as an `AreaEntity` because it's an actor that's not supposed to "inhabit" grid cells. This shows that a grid game can perfectly well involve other types of actor than those specifically dedicated to occupying cells.

Finally, some useful " getters-setters " are also provided. You'll notice protected accesses for methods presenting a potential danger to encapsulation.

An important method of `AreaEntity` is `setOwnerArea` which lets you tell an actor what area it belongs to.

5.5.2 Interfaces `Interactor` and `Interactable`

The purpose of the grid is to manage the content of its cells and what happens within them, such as authorizing or prohibiting the passage of an actor from one cell to another, and managing interactions between actors occupying identical or neighboring cells. This is the role of the `entities` attribute of the `Cell` class (coded using the predefined `Set` type, see Appendix 2 7.1 on sets). This set is not modeled as an `Actor` set, and we'll see why.

In fact, actors only interest the grid as entities receptive to interactions. Indeed, it's easy to imagine that certain actors (e.g. a wallpaper) are impervious to any interaction and therefore don't need to be taken into account by the grid. What's more, an entity receptive to interaction is not necessarily an actor. It may simply be a cell. The `Set<Actor>` type is therefore not entirely suited to modeling the contents of a cell, and encoding this set calls for new abstractions. In concrete terms, these are entities capable of interacting.

The model offers the following interfaces, placed in the `ch.epfl.cs107.play.areagame.actor` package :

- `Interactable` : this interface can be used to model any entity receptive to an interaction request ;
- `Interactor` : this interface models any entity that can interact with an `Interactable`.

As their name suggests, these two interfaces are intended to work in symbiosis, with the `Interactor` designed to interact with the `Interactable`.

We assume that any grid entity (`AreaEntity`) must define itself as an object subject to interaction. For this reason, `AreaEntity` implements the `Interactable` interface !.

The `entities` attribute of a `Cell` is therefore not a set of `Actor` but a set of `Interactable`. In particular, it contains a protected method `enter` for adding a given `Interactable` to this set, and a protected method `leave` for removing a given `Interactable` from this set.

We will also differentiate between two types of interaction :

- *contact interactions* : take place between a `Interactor` and the `Interactable` located in the same cells as it.
- *distant interactions* : take place this time between a `Interactor` and the `Interactable` located in the cells of its field of vision.

To illustrate this difference, let's take an example. Let's imagine a situation with three actors : two characters and a sheet of ice. The two characters can interact in a *distant* way, for example to chat ; they don't need to be in the same cell to talk to each other. On the other hand, the characters can only interact with the plate of ice by *contact* : when they enter the cell containing the plate of ice, they can slide.

For the time being, we won't be examining the content of the `Interactor` interface (which is only needed to support interactions between actors, and which we'll do in the next step). Examine the code of the `Interactable` interface used to model an entity receptive to interactions.

The proposed design models the fact that a `Interactable` :

- occupies a list of cells : method `List<DiscreteCoordinates> getCurrentCells()` ;
- when occupying a cell, can make it non-traversable by others (it can prevent other `Interactable` from investing the cell it occupies) : method `boolean takeCellSpace()`. An `Interactable` for which `boolean takeCellSpace()` returns `true` will be called "non-traversable" in the rest of the statement (of course, whether it's traversable or not may depend on various conditions and need not always be true or false) ;
- indicates with a Boolean method whether it accepts *distant* interactions : method `boolean isViewInteractable()` ;
- and indicates with a Boolean method if it accepts *contact* interactions : method `boolean isCellInteractable()`.

It is also considered that at the abstraction level of an `AreaEntity`, it is not possible to concretely define the methods dictated by the `Interactable` interface.

Finally, note that `Cell` also implements `Interactable` to indicate that cells are also receptive to interactions. At the `Cell` abstraction level, only the `getCurrentCells` method can be redefined. It returns a list whose single element is the cell's coordinates.

We'll come back to the other methods of the `Interactable` interface later.

5.5.3 The class `MovableAreaEntity`

Some of our grid actors will naturally be in motion. Unlike our `SimpleGhost` actor, they'll have to move with the grid in mind.

The abstract class `MovableAreaEntity` in the package `areagame.actor` is derived from `AreaEntity` and can be used to model this type of actor. Its main feature is the presence of a `move` method enabling the actor to move continuously.

The parameter `framesForMove` is the number of frames (steps) chosen to implement continuity of movement. In concrete game implementations, we'll introduce the possibility of matching each step (frame) with a different graphical representation, thus animating the move.

In order to take place, the move must be authorized by the `Area` and by each of the cells that the entity will leave or enter during the move.

By definition, the move will always take place from the current main cell to a cell adjacent to it, defined by the entity's current orientation. The mobile actor moves one cell at a time, and to ensure that it never finds itself between two cells, a move will always wait until the end of the previous one before starting.

The role of the `protected boolean move(int framesForMove, int frame)` method is to decide whether a move can take place and, if so, to initiate it. It returns true to indicate that a move is in progress.

5.6 The area and its grid dictate their conditions

Now it's time to look at how the grid and the area impose their conditions on the placement and movement of the entities that take their place. Remember that each area is equipped with a grid. To do this properly, the addition or removal of an actor from the area must

take into account the potential *veto* of the grid. For example, the grid should normally be able to oppose the addition of an actor to a given cell. For example, an actor whose number of cells is too large to be placed in a desired position (grid overflow) should be able to be refused by the grid and therefore not be added as a new actor in the area. Similarly, the `move` method of the `MovableAreaEntity` must allow the area or grid to express constraints on movement. Typically, it should at least prevent an entity from leaving the grid. To dictate these conditions, we start from the idea that the cell can dictate its conditions, which will impact the decisions of the grid to which it belongs, and which in turn may impact the decisions of the area associated with the grid.

5.6.1 Conditions dictated by cells and grid

In order to allow a cell to express control over placement/movement, the `Cell` class contains the protected abstract methods :

- `boolean canEnter(Interactable entity)` : returning `true` if `entity` has the right to add itself to the cell contents and `false` otherwise ;
- `boolean canLeave(Interactable entity)` returning `true` if `entity` has the right to subtract from the cell content and `false` otherwise.

These methods, together with the grid's knowledge of its dimensions, will enable it to condition the movements and positioning of the `Interactable` that may inhabit its cells.

5.7 Exercise 2 (continued : adding a character)

You now have (almost) all the basic logistics for coding grid-based games, including actors (Phew !). To see this in action, you'll now complete the coding of the game `Tuto2`.
So that the grid `Tuto2Behavior` can dictate its conditions to the actors in it, the cells matched to `Tuto2Cell` will be characterized by the fact that they :

- only allow entering a cell if its attribute `isWalkable` is true (set `canEnter()` accordingly) ;
- accept contact interactions (define `isCellInteractable()` accordingly) ;
- do not accept remote interactions (define `isViewInteractable()` properly) ;
- can always be exited (define `canLeave()` properly).

5.7.1 Specific actors

We now need to create a grid game actor, `GhostPlayer` in `ch.epfl.cs107.play.tuto2.actor`. This type of actor will inherit `MovableAreaEntity`. It will accept any type of interaction and will not be traversable.It will also have the same behavior as the actor `SimpleGhost` (hit points, `isWeak` method, and transition from one area to another when the number of hit points becomes zero)[9]. It will also be equipped with methods allowing it to :

- enter a given area by placing itself in a given position :

[9] take directly from what you've done on this subject in the `SimpleGhost` class

```
void enterArea(Area area, DiscreteCoordinates
    position)
```

The algorithm consists of :

1. register as an actor (taking the necessary steps to indicate the area to which it belongs) ;
2. update its absolute position : setCurrentPosition(position.toVector()) ;
3. and set itself to immobility (`resetMotion`).

- and leave the area to which it belongs (deregister)

The `GhostPlayer` constructor will have the following header :

```
public GhostPlayer(Area owner, Orientation orientation,
    DiscreteCoordinates coordinates, String sprite)
```

`coordinates` is the square occupied by the ghost when it was created. In order to be instantiated, a `GhostPlayer` must contain concrete definitions of the methods imposed by `Interactable` and `MovingAreaEntity`.

```
@Override
public List<DiscreteCoordinates> getCurrentCells() {
    return
        Collections.singletonList(getCurrentMainCellCoordinates());
}
```

For simplicity's sake, we'll assume that the actor occupies only his main cell. The `update` method of `GhostPlayer` implements the following algorithm :

1. start moving or orienting according to the keys pressed by the user
2. call the method `update` of the super-class (to actually perform the initiated move, if necessary) ;

For step 1 of the above algorithm, the algorithm is as follows :

- if the button corresponding to the `Keyboard.LEFT` is pressed (`isDown`) then if the actor is oriented to the left, we initiate the movement to the left (call to `move`). Handle this interaction directly in the `update` method of the actor (this is now possible because an `AreaEntity` knows the area to which it belongs and therefore has access to its `getKeyboard()` method).
- otherwise, we orient the actor to the left.

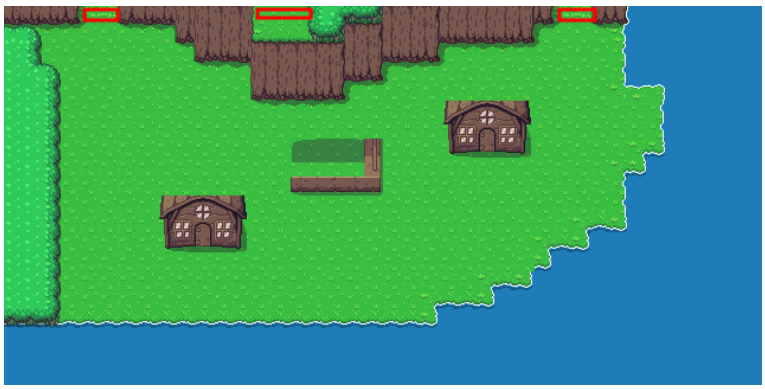
The number of "frames" used by `move` could be defined as a static constant :

```
/// Animation duration in frame number
private final static int ANIMATION_DURATION = 8;
```

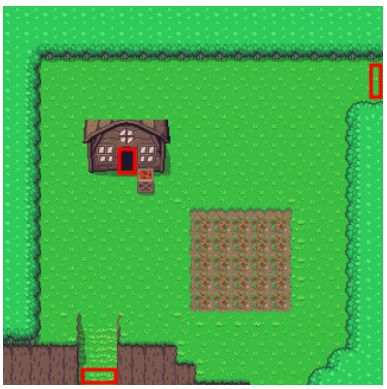
We will proceed in a similar manner for all other orientations.

`GhostPlayer` will obviously have to have a specific drawing method, which will simply draw the `Sprite` partner.

Finally, like `Interactable`, `GhostPlayer` must also provide an empty implementation of `acceptInteraction` for the moment (like `Tuto2Cell`).



doors in ”zelda/Village”



doors in ”zelda/Ferme”

Fig. 6 : Description of doors

You will note that only the players in the grid games have access to the area to which they belong.

5.7.2 Character Placement

Complete `Tuto2` so that this game is characterized by a `GhostPlayer` type character. The character will be created when the game starts, with orientation `Orientation.DOWN`. It will be recorded in the current area and the camera will be centered on it. Its `update` method will simply implement the fact that if the character is too weak, he will transit to the next area. This will do exactly the same as what was done in `Tuto1` (if it was in `Village` it goes into `Ferme` and vice versa).

`update` no longer needs to manage keyboard interactions, which are managed directly in the `update` of the character.

You will use `(2,10)` as starting coordinates in `Ferme` and `(5,15)` in `Village` and these are the same coordinates which will be used as starting coordinates each time the actor switches back to these areas. You can use `13.f` as a scaling factor, and it makes sense that this value is a final static constant specific to the game, i.e. `Tuto2`.

5.7.3 Validation of the coded solution

You will verify that `GhostPlayer` :

1. can move across the entire surface of the playing areas without leaving the grid ;
2. cannot walk on obstacle areas (all areas corresponding to black or gray in the associated behavior image, typically water or barriers cannot be crossed)
3. is well tracked by the camera when moving ;

4. can correctly transit from the area `Village` to the area `Ferme` and vice versa. For the moment he will only do it based on his life points.

It would be natural for the character to transit from one area to another rather by passing through areas corresponding to doors (see the figure 6). To do this, and to complete the tools necessary for creating games, it is necessary to be able to properly model the interactions that can take place between actors. This is the theme of the last tutorial.

6 Tutorial III

This third and final tutorial presents the model tools dedicated to coding *interactions between actors*. Beforehand, you can revisit your design by making good use of nested classes. Some utility classes, which will be useful for you to tackle the mini-project, are also presented at the end of the tutorial.
There will be no exercise per se, the concepts presented will be directly exercised in the first part of the project.

6.1 A little "refactoring" using nested classes (optional)

Now that nested classes have been presented in class, you can, if you wish, improve the design of the model provided by ensuring that the concept of cell is inseparable from that of grid. The `Cell` class would therefore become a public class nested in the `AreaBehavior` class. It would then also be necessary to ensure that `Tuto1Cell` and `Tuto2Cell` become nested subclasses of `Tuto1Behavior` and `Tuto2Behavior`. This will improve encapsulation. The `cellInteractionOf` and `viewInteractionOf` methods can in fact thus become private because they are in principle not useful outside the grids. The same goes for cell constructors.
The cell models themselves remain *public* classes because to manage the interactions that the actors can have with the cells they must be able to access them.

6.2 Tutorial answer key

To access the tutorial solution, all you need to do is install a new IntelliJ project using the archive : tuto-solution-2023.zip
To install the project on IntelliJ :

1. unzip this archive into a directory of your choice ;
2. open the directory `TUTO-SOL-2023` ;
3. delete the archive `solution.zip`.

For Eclipse, once the archive is unzipped, create the project using the `"From existing sources"` option and indicating the host directory as the root of the project. The tutorial solution can be found in the `game.tutosSolution` package. The new version of `AreaBehavior` nests the `Cell` class. The solution provided in `tutosSolution` is based on this concept,

To start the project, you will draw inspiration from the correction in tutorial 2. Read it

6.3 Interactions between actors

In our previous game, it would have been natural to allow our main character to transit from one area to another via places with a "passage" visual (red pixels). A simple way to do this would have been to use the color of the pixels of the image associated with the grid

to give specific behavior to the character based on this color. However, it is not very good to do this for several reasons :

- this implies that the grid must communicate specific information to the characters (for example provide a method `boolean isDoor(int i, int j)` allowing to know if a given cell corresponds to a red pixel (not general enough : what happens if the color red has to be interpreted differently at another level of the game ?) ;
- it is not certain that we necessarily want to exploit all the cells corresponding to a red pixel as doors in our games ;
- a place with a "passage" visual can correspond to different types of doors (we can for example imagine having doors that open with a key, others that we can pass through without conditions, etc.).

Therefore, it is preferable to instead create a *actor* `Door` to be placed (in general) on the red zones (but not necessarily all). The interaction must then take place between two actors (a "door/passage" actor and a "character" actor). It is therefore now a matter of studying the components of the model making it possible to manage *interactions between actors*.

6.3.1 The Interactors

So let's say we want to create a game where a character can interact with a "door" actor and a "tuft of grass" actor in the sense that he can go through the door and cut the tuft of grass. The character must play a more active role by expressing whether he wants interaction or not (for example, he is not forced to cut the grass). It will therefore be an entity which *requests* an interaction. This particular category of actors, requesting interaction, can be modeled in the model using the `Interactor` interface. Open the `Interactor` interface. You will see that the latter allows you to model an object :

- which occupies a list of cells and therefore has a method `List<DiscreteCoordinates> getCurrentCells()` returning the coordinates of these cells ;
- which has a certain number of cells in its field of view and therefore has a method `List<DiscreteCoordinates> getFieldOfViewCells()` returning the coordinates of the cells in its field of view ;
- which indicates with a boolean method `boolean wantsCellInteraction()` if it requests interaction from *contact* ;
- which indicates with another method `boolean wantsViewInteraction()` if it requests *remote* interaction ;
- and which allows you to interact with a `Interactable` using the `void InteractWith(Interactabl boolean isCellInteraction)` method. The second parameter allows you to specify the desired mode of intercation : by contact (parameter value `true`) or remotely `false`).

Let us now see how this particular type of actor intervenes in the simulation. So far we've only been concerned with a few lines in the `update` method of a play area (`Area`). Look again at the code for this method and look at its `update` method. There you will see that after the actor updates loop :

```
for (Actor actor : actors) {
  actor.update(deltaTime);
}
```

the actual interaction management takes place :

```
for (Interactor interactor : interactors) {
   if (interactor.wantsCellInteraction()) {
     // ask the associated grid (AreaBehavior)
     //to set up contact interactions
   }
   if (interactor.wantsViewInteraction()) {
     // ask the associated grid to set up
     // distant interactions
   }
}
```

The `AreaBehavior` grid being the manager of all the mechanisms that take place there, it is in fact up to it to provide the methods managing the interaction strictly speaking. This raises two new issues : how are all the interactors defined/constructed ? (the variable `interactors` in the code above) and how the grid intervenes to manage the interactions ?

6.3.2 Set of `Interactors`

Any actor of type `AreaEntity` is likely to be receptive to interaction. This is why the class `AreaEntity` already implements the `Interactable` interface. On the other hand, classes that implement `Interactor` will rather be close to concrete objects (deciding whether an object is willing to interact is rather done in a specific way). For example a game character is a natural candidate to be an `Interactor`.

The actors playing the role of `Interactor` have a special role to fulfill. You must therefore be able to distinguish them from the others. This is why the `Area` has an attribute `interactors` logging all actors of type `Interactor`. If you take a closer look at its `addActor` method, you will see that it also has the role of populating the `interactors` attribute (and therefore to categorize the actors according to whether they are `Interactor` or not). An actor of type `Interactor` is recorded in the list of `actors` as well as in the `interactors` list.

> It is not uncommon in programming to reference the same object from multiple places. This allows you to manipulate the objects in question from different points of view : a `Interactor` must be able to be seen as a `Actor` so that we can apply its method `update` or as a `Interactor` so that we can make him interact with the other actors.

6.3.3 Handling interactions at the grid level

The idea is therefore that it is ultimately up to the grid to set up the interaction mechanisms. This is why the class `AreaBehavior` is equipped with the methods :

- `public void cellInteractionOf(Interactor interactor)` : which manages all contact interactions between `interactor` and the `Interactable` in the same positions as those

he occupies.

- This method loops through all cells at positions `interactor.getCurrentCells()` and apply a method to them `cellInteractionOf(interactor)` specific to `Cell`.

- `public void viewInteractionOf(Interactor interactor)` : which manages all remote interactions between `interactor` and the `Interactable` of his field of vision. This method loops through all cells at positions `interactor.getFieldOfViewsCells()` and applies a method `viewInteractionOf(interactor)` specific to `Cell`.

These two methods allow the `Interactor` (as a parameter of these two methods) to listen to the grid. They require the following methods to be present at `Cell` :

- `private void cellInteractionOf(Interactor interactor)`

- `private void viewInteractionOf(Interactor interactor)`

Here is how the code for the first of these methods looks like :

```
private void cellInteractionOf(Interactor interactor){
            for(Interactable interactable : entities){
                if(interactable.isCellInteractable())
                    interactor.interactWith(interactable, true);
            }
}
```

where `entities` represents the entire `Interactable` listed in the cell. The second method is coded in the same spirit.

6.4 Generic interactions

Here we are at the heart of the subject, how to concretely code the method

```
void interactWith(Interactable other, boolean
   isCellInteraction);
```

for a `Interactor` given ?

Let's place ourselves in a more general context than that of already coded game sketches and suppose that we have to code a main character interacting with other actors. Let's call it `MyPlayer`. The latter will typically be a `Interactor` ; that is, an entity that invites interactions. How could we a priori define our specific method `void interactWith(Interactable other, boolean isCellInteraction)` so as to allow him to interact with actors `Door` (door) and `Grass` (tuft of grass)?

The trivial way to do this would be to resort to *type tests* :

```
void interactWith(Interactable other, boolean isCellInteraction){
  if (other instanceof Grass() && !isCellInteraction) // remote interaction
     with grass...
  if (other instanceof Door && isCelInteraction) //contact interaction with
     the door...
}
```

which is very *ad hoc* and not very scalable. In fact, when programming a game, all `Interactor` can potentially interact with all other possible players in the game and all cases must be

considered. A design diagram is classically used in this type of situation where there are actions to be performed on all kinds of objects which do not necessarily have links between them. It consists of delegating the management of these actions to an external class which we would here call the character interaction manager[10] :

```
  /* manages MyPlayer's interactions with all actors */
  class MyPlayerHandler {
  public void interactWith(Door door, boolean isCellInteraction) {
    // fensure that the door is passed by the actor
  }
  public void interactWith(Grass grass, boolean isCellInteraction){
    // makes sure the grass is cut
  }
}
```

This handler is specific to `MyPlayer`, in our case it would be coded as an internal private class of this class.
The class `MyPlayer` would have its interaction manager as an attribute :

```
private final MyGamePlayerHandler handler;
```

and a generic method :

```
  /* asks other to agree to have their interactions
       with MyPlayer managed by handler
  */
 public void interactWith(Interactable other, boolean isCellInteraction) {
        other.acceptInteraction(handler, isCellInteraction);
 }
```

[10]Commonly called the “visitor” design pattern

Chaque `Interactable` doit alors offrir une méthode indiquant qu'il accepte de faire partie d'une interaction gérée par le gestionnaire du personnage. Par exemple dans `Grass` on aurait :

```
public void acceptInteraction(MyPlayerHandler v, boolean isCellInteraction)
    {
  // fait en sorte que le gestionnaire d'interaction du personnage gère
     l'interaction avec Grass
       v.interactWith(this, isCellInteraction);
}
```

This solution offers the advantage of being able to code a single, very general method in `Interactor`, the method `interactWith(Interactable, boolean isCellInteraction)`. Only one more downside, the argument of `acceptInteraction` in `Grass` is still too specific : we should add a method `acceptInteraction` with the handlers of each `Interactor` possible (here we only have one `Interactor`, but nothing prevents us from introducing others). The idea is therefore to instead resort to the diagram in figure 7.

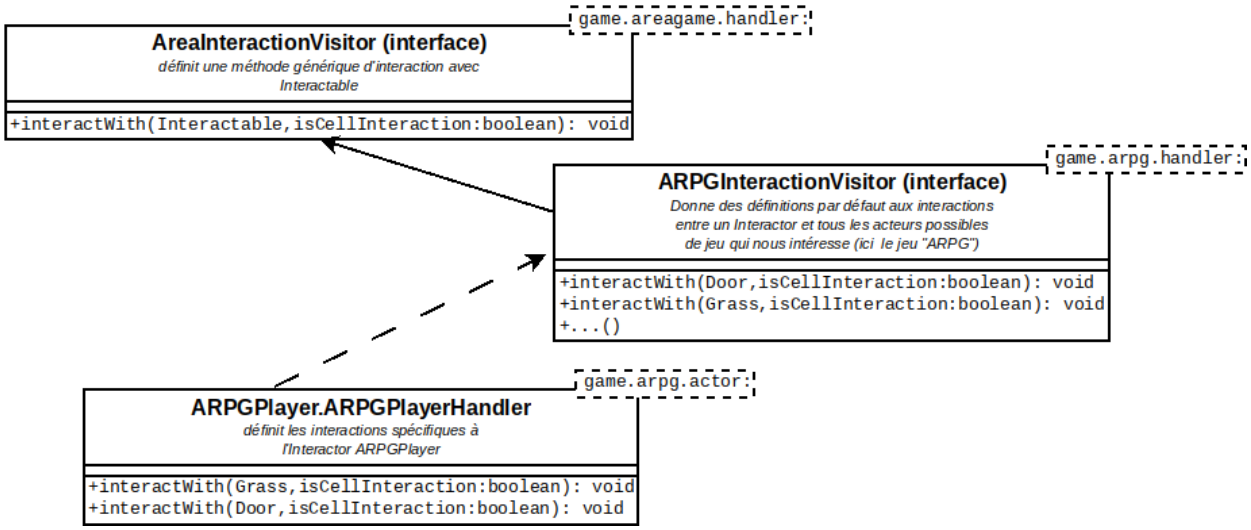


FIG. 7 : Class diagram for setting up interactions

We therefore inherit `MyPlayerHandler` more general interaction managers. In this way, the `Interactable` must offer as the only additional method :

```
public void acceptInteraction(AreaInteractionVisitor v, boolean
    isCellInteraction) {
        // with a simple default definition
}
```

The interface `AreaInteractionVisitor` models a generic interaction manager for which we can imagine a default implementation which is provided in the `areagame.handler` package. The method `acceptInteraction` by `Grass` (or `Door`) would then be written simply :

```
  public void acceptInteraction(AreaInteractionVisitor v,
                                boolean isCellInteraction) {
        ((MyGameInteractionVisitor)v).interactWith(this, isCellInteraction);
}
```

IThere is certainly a conversion to be carried out, but only one. This conversion allows interaction management to be delegated to the game-specific manager to which `Grass` participates.

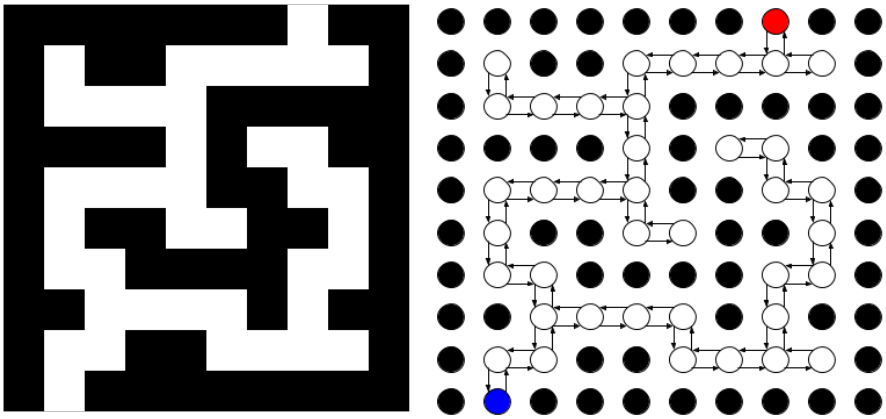


FIG. 8 : Example of a behavioral image representing a maze (walls in black) and a graph representing the connections between passable cells (in white)

This indicates that the tuft of grass agrees to have its interactions managed by the specific interaction manager `MyGameInteractionVisitor` and whose `MyPlayerHandler` is a concrete achievement. This handler expects that any `Interactor` can have interactions with each actor in the game concerned. Adding a new actor involves retouching the `MyGameInteractionVisitor` handler. and only the `Interactor` who would like an interaction with this new actor. The other actors, however, do not undergo any modification due to the introduction of this new actor (which was not the case with the other attempts towards the solution mentioned above).

Figure 7 graphically summarizes the suggested design scheme for managing interactions between actors. The second part of the project will allow you to concretely implement this plan. In particular, you will be given precise instructions on where to place the classes mentioned in the diagram presented and how to code them.

The components of the mockup below are useful for coding the project or extensions. Skim over this material quickly at this time to make yourself aware of its existence and return to it as you need for the project.

6.5 Class `AreaGraph`

The class `AreaGraph` from the package `game.areagame` allows you to associate a connected graph with a game grid. This can be used to simulate the beginning of artificial intelligence for the movement of actors (actors who move following a path).

The class `AreaGraph` offers in particular the method :

```
Queue<Orientation> shortestPath(DiscreteCoordinates from,
    DiscreteCoordinates to)
```

which allows you to find the shortest path between a starting point and a destination point in the graph associated with a grid. This path is described as a "file" https://fr.wikipedia.org/wiki/File_(structure_de_donn%C3%A9es) from `Orientation`. This is the sequence of orientations to adopt to arrive at the box `to` starting from the box `from`. Queues are implemented in Java using the `Queue` (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Queue.html). Suppose an actor moves on a

grid associated with a graph `graph`. If this actor occupies the coordinate cell `start` and he wants to go to the coordinate cell `stop`, orientation `nextOrientation` that he must adopt is given by :

```
Queue<Orientation> path = graph.shortestPath(start, stop);
Orientation nextOrientation = path.poll();
```

The `poll` method of `Queue` allows you to extract and delete the head of the queue.

6.6 Classes `RPGSprite` and `Animation`

You used the class `Sprite` in a simple way to graphically represent the main character of the game. Other classes are available to refine the graphic representation of objects.

It is indeed possible to use the `RPGSprite` class deriving from `Sprite` and which associates depth with the image. It can be interesting to play with depth to, for example, place the character in front or behind an object for example.

Furthermore, a `Sprite` or `RPGSprite` does not necessarily correspond to a unique representation.

A complex image, such as the one associated with `"player.png"` from the folder *`"res/sprites/zelda/"`* (figure 9), is made up of 4x4 small `Sprite` size 16x32. It can be divided into sets allowing different views of the characters or animations

FIG. 9 : Example of an image used as a basis for animations

For example, the top line which allows you to animate downward movements can be obtained like this :

```
spritesDOWN[i] = new Sprite("zelda/player", 1, 2, this, new
   RegionOfInterest(i*16, 0, 16, 32));
```

for `i` ranging from 0 to 3.

The concept of animation, offered by the class `Animation` from the `area.areagame.actor` package, is characterized by a set of `Sprite` to be displayed in turn. Its `update` method allows you to choose which element/frame of the set is the current element. This is the one that will be displayed if we call the `draw` animation method. It is possible to influence the

speed at which the frame changes when calling `update`, using the `frameDuration` attributes and `speedfactor` (and associated methods).

The main character, for example, can therefore be associated with 4 animations allowing him to animate him when he moves up, down, left or right. Drawing the character therefore amounts to drawing the 4 animations associated with it.

Animations can of course be associated with any actor. For example, a torch can offer an animated visual giving the impression that its flame is moving.

Failing to animate the actors, we can at least orient them visually by choosing `Sprite` specific to the orientation.

You will notice that the `RPGSprite` offers some useful methods for extracting `Sprite` of a complex image or the creation of animations from `Sprite` tables, in particular, `extractSprites()` and `createAnimations()`

So if we want to create animations corresponding to the up, down, left, right orientations of a character, we can do something like this :

```
Sprite[][] sprites = RPGSprite.extractSprites("zelda/player",
   4, 1, 2,
this, 16, 32, new Orientation[] {Orientation.DOWN,
   Orientation.RIGHT, Orientation.UP, Orientation.LEFT});
// creates an array of 4 animations
  Animation[] animations =
     Animation.createAnimations(ANIMATION_DURATION/2, sprites);
```

where `ANIMATION_DURATION` is the number of frames used for the movement (here we go from one animation to another every two movement steps).

6.7 Signals

The interface `Signal` is provided in the `signal` package. It very simply models a signal as an entity with an intensity (a value of type `float` between 0.0 and 1.0). Any object, actor or not, implementing the `Signal` represents a signal whose intensity value can be used, in various ways, to make decisions. See the code for this interface as provided in the `signal` package. We also invite you to take an overview of the different types of signals provided in the same package. You will notice that the interface `Logic` offers in particular two constants of type `Logic` (yes Java allows recursive definitions !) : the constant `TRUE` and the constant `FALSE`. Here are some explanations about the twist :

```
Logic TRUE = new Logic() {
     @Override
     public boolean isOn() {
         return true;
     }
};
```

(in particular, do we have the right to instantiate an interface ??)

This code means that we create the instance of an anonymous class (without name), implementing the interface `Logic` and where the `isOn` method is redefined. `TRUE` is therefore an instance of this anonymous class (and not of the interface !). The constant `FALSE` is defined

analogously. So `Logic.TRUE` represents an always-on signal (which can be assigned to a variable of type `Logic`) and `Logic.FALSE` represents a signal that is always disabled.

It is possible, for example, to code actors whose behavior depends on signals : for example a "door" actor which would be opened or closed depending on whether a "key" actor has been picked up or not. The "key" actor would be a signal (ON when picked up by the player and OFF otherwise) and the door would have the key as an attribute conditioning its opening.

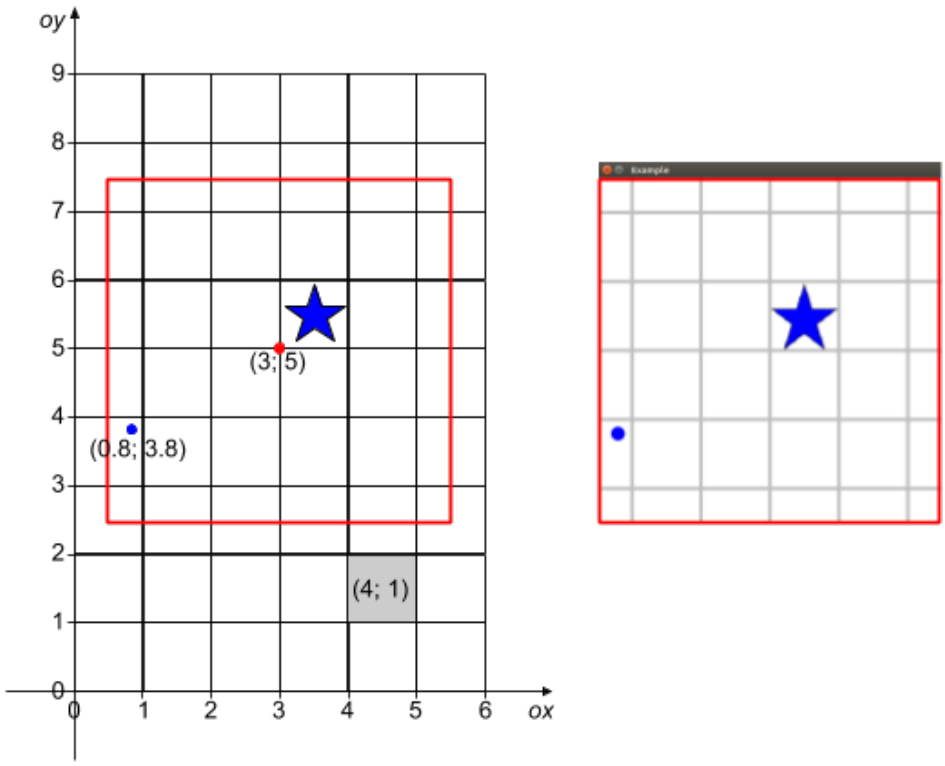


FIG. 10 : The view on a targeted part of the grid is obtained by an affine transformation of the window (here a simple translation)

7 Appendices

7.1 Appendix 1 : "Positionable" objects, transformed objects and graphic objects

The positioning and display of simulated elements in the simulation window are obviously fundamental points.

The first point to make here is that when positioning simulated objects, it's not practical to think in pixels : this makes us dependent on the size of the window, which is counter-intuitive ; our simulated universes will probably be larger than what we want to display.

So we're going to express all our magnitudes relating to positions, dimensions etc in terms of the simulated grid's magnitude scales and not in terms of pixels in the window. As the grid can be larger than the display window, we're going to subject the latter to affine transformations (translation, zoom etc.) to enable us to focus on a specific part of the world (see Figure 10).

The display window is a typical example of an element that needs to be positioned/modified in the absolute frame by means of transformations. In fact, all elements to be positioned in the absolute frame can be positioned in the same way (e.g. shapes or images to be drawn).

To meet this need, the API provides the following elements :

- The `Positionable` interface describes an object whose absolute position can be obtained by means of an affine transformation (`Transform` class). A `Entity` is typically

a `Positionable`

- The `Attachable` interface describes a `Positionable` that can be attached to another (its parent). This is done using the `setParent` method. It is characterized by a relative transform, which indicates how the object will be positioned in its parent's reference frame (or in the absolute if it has no parent).

- The `Node` class, which is a simple concrete implementation of the `Attachable` interface.

The `getTransform()` method applied to a `Positionable` actually allows you to locate it in its local/relative reference frame.

In addition, the API provides classes such as `ch.epfl.cs107.play.engine.actor`, `TextGraphics`, `ImageGraphics` and `ShapeGraphics` which implement the notion of "drawable" objects (`Graphics`). A `Graphics` can be attached to an `Entity` using the `setParent` method. If a graphic object is attached to an entity, its drawing will necessarily take place in that entity's reference frame, with no need to explicitly place it there by means of a transformation (you have an example of this with the text attached to the rock in the first "game" to be created, `Demo1`).

However, it is sometimes necessary to specify the anchor point of the graphic object in relation to the entity that serves as its parent (i.e. how far the image must be offset from the origin in order to be superimposed cleanly on the entity). Take a look at the relevant API to see how this notion of anchor point is put into practice.

Appendix 2 : Useful data structures

There are many different data structures. For example, in this course, you learned how to use dynamic arrays via the `ArrayList` class. In reality, `ArrayList` is a special implementation of the abstract data structure *List*.

Data structures are provided in Java in the form :

- An interface that describes the functions usually accepted for the data structure in question ; for example, the ability to add an element at the end of a list in the case of lists. In the case of lists, for example, the interface that describes these functions is `List`
- A very general basic implementation of this interface in the form of an abstract class : `AbstractList` for lists.
- Several specific implementations derived from the abstract class, e.g. `ArrayList` or `LinkedList` for lists. Each of these specific implementations has its own particularities, so you'll want to use one rather than the other, depending on the context. For example, `LinkedList` offers add or delete operations after a given element in constant time (O(1)), but does not offer the possibility of accessing an element at a given position in constant time. For `ArrayList` ("table list") the opposite is true. You'll therefore tend to prefer `LinkedList` ("chained list") if there are more add or delete operations than those requiring direct access.

Some data structures are more appropriate than others, depending on the situation. We'll briefly describe two more that will prove useful in this mini-project (a more in-depth presentation of these data structures and their characteristics will be given in the second semester).

Associative tables

Associative tables ("map") generalize the notion of index to non-integer types. They can be used to associate *values* with *keys*.

For example :

```
import java.util.Map;
import java.util.HashMap;
import java.util.Map.Entry;

//...
        // String is the key type and Double is the value type
        Map<String, Double> grades = new HashMap<>();
        grades.put("CS107", 6.0); // maps key "CS107" to value
           (note here) 6.0
        grades.put("CS119", 5.5);
         // … ditto for other courses to which you'd like to
            associate your grade

        // Three ways of iterating over the contents of the map
        for (String key : grades.keySet()) {
```

```
        //iterate on keys
        System.out.println(key+ " " +grades.get(key));
    }

    for (Double value : grades.values()) {
        //iterate on values
        System.out.println(value);
    }

    for (Entry<String,Double> pair : grades.entrySet()) {
        //iterate on key-value pairs
        System.out.println(pair.getKey() + " " +
           pair.getValue());
    }
```

The key of a Map can therefore be seen as a generalization of the notion of index. The Java interface that describes the basic functionality of associative tables is `Map`, while the concrete implementation we'll be using is `HashMap`.

Sets

It is sometimes necessary to manipulate a collection of data as a *set* in the mathematical sense ; that is, where *each element is unique*. For example, if we want to model the set of vowels, there's no reason why the letter `'a'` should appear twice. The method of adding an element to a set ensures that the element is not added if it was already there :

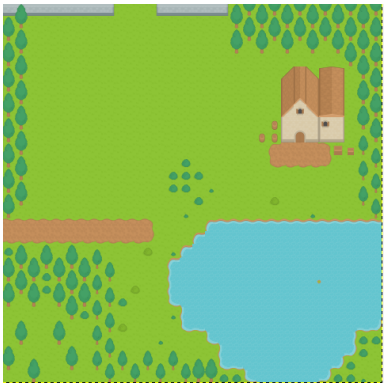
```
import java.util.Set;
import java.util.HashSet;

//...
        Set<Character> voyels = new HashSet<>();
        voyels.add('a'); // voyels -> {'a'}
        voyels.add('u'); // voyels -> {'a', 'u'}
        voyels.add('a'); // voyels -> {'a', 'u'}

        // display: a u
        for(Character letter : characters) {
            System.out.print(letter + " ");
        }
```

The Java interface that describes the basic functionality of sets is `Set`, the concrete implementation we'll be using is `HashSet`.

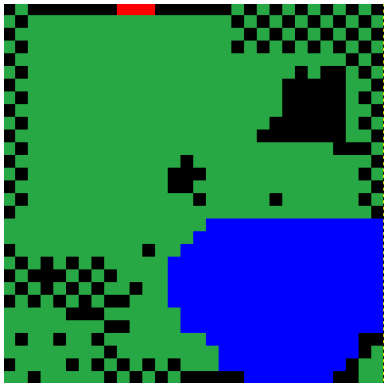
7.2 Appendix 3 : Graphic resources and level editor

More images You are free to use other images, either of your own creation or collected from the web. In such cases, it is essential to cite the origin !

Level editor Play areas have a background image superimposed on an image dictating their behavior (pixel color) :



background image



corresponding ”behavior”

We've provided a few examples in the `src/main/resources` resource file, where the `images/background/` folder contains background images and each of these images corresponds to a possible ”behavior” image in the `behaviors/` folder.

It's obviously interesting to be able to create new images. If you wish (this is not required as part of the project), you can use the simple level editor proposed by Bastien Chatelain and completed by Sami Abuzakuk (course assistants) : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/LevelEditor.zip [Link to an eclipse project archive].