CS-107: Mini-project 2
Cooperative games

J. Kalajdzic, J. Sam, A. Paulino

Version 1.4

Contents

1 Introduction **4**

2 ICoop de base ICoop Basics (Step 1) **7**

2.1 Preparing the ICoop `ICoop` . . . 7
2.2 Adaptation of `ICoopBehavior` . . . 8
2.2.1 Task . . . 8
2.3 Draft of Main Characters . . . 8
2.3.1 Drawing . . . 9
2.3.2 Behavior . . . 9
2.3.3 Task . . . 10
2.4 Doors: Contact Interactions . . . 10
2.4.1 Doors . . . 11
2.4.2 `ICoopPlayer` as an `Interactor` . . . 12
2.4.3 Task . . . 13
2.5 Second Character . . . 13
2.5.1 Task . . . 14
2.6 Explosives: Remote Interactions . . . 14
2.6.1 Obstacles and Explosives . . . 14
2.6.2 Task . . . 15
2.7 Reset . . . 15
2.7.1 Task . . . 16
2.8 Validation of Step 1 . . . 16

3 First Cooperation (Step 2) **17**

3.1 Health Bar . . . 17
3.1.1 Task . . . 18
3.2 Dialogues . . . 19
3.2.1 Activating a Dialogue . . . 20
3.2.2 Task . . . 20
3.3 Collectible Items . . . 21
3.3.1 Collecting Explosives . . . 21
3.3.2 Task . . . 21
3.3.3 The `ElementalItem` . . . 21
3.3.4 The Orbs . . . 22
3.3.5 Task . . . 22
3.4 Elemental Walls . . . 22
3.4.1 Elemental Category Role . . . 23
3.4.2 Task . . . 23
3.5 Assistance, Healing, and Invulnerability Periods . . . 24
3.5.1 Task . . . 25
3.6 Maze Area . . . 25
3.6.1 Task . . . 25
3.7 Validation of Step 2 . . . 25

4 Enemies and Battles (Step 3) 26
4.1 Equipment . . . 26
4.1.1 Inventory Items . . . 26
4.1.2 Inventories . . . 27
4.1.3 Collection and Inventory . . . 28
4.1.4 Graphics . . . 28
4.1.5 Task . . . 29
4.2 Enemies . . . 29
4.2.1 Projectiles . . . 31
4.2.2 Fire-throwing Skulls . . . 31
4.2.3 Task . . . 32
4.2.4 Bomber . . . 32
4.2.5 Task . . . 34
4.3 Battles . . . 35
4.3.1 Staffs and Water/Fire Orbs . . . 35

4.3.2 Tasks . . . 35
4.3.3 Weapon Usage by Characters . . . 35
4.3.4 Tasks . . . 36
4.4 Validation of Step 3 . . . 37

5 Final challenge (Step 4) 38
5.1 Keys and Teleporters . . . 39
5.2 Area `Arena` . . . 39
5.3 Gateway to the manor . . . 39
5.4 Task . . . 40
5.5 Validation of Step 4 . . . 41

6 Extensions (Step 5) 42
6.1 New actors or character extensions . . . 42
6.2 Pause, game ending and 'reset'(~2 to 5pts) . . . 43
6.3 Validation of Step 5 . . . 44
6.4 Contest . . . 44

7 Appendices 45
7.1 KeyBindings . . . 45
7.2 Basic Configuration of the Maze Area . . . 45
7.3 Creating Animations . . . 46
7.3.1 `ICoopPlayer` . . . 46
7.3.2 Explosive . . . 48
7.3.3 Orbs . . . 48
7.3.4 Hearts . . . 48
7.3.5 Staves . . . 48
7.3.6 Flames . . . 49
7.3.7 Water or Fire Projectiles . . . 49
7.3.8 Death of foes . . . 49
7.3.9 Fire-Shooting Skulls . . . 49
7.3.10 Artificers . . . 49

1 Introduction

This document uses colors and contains clickable links. It is best viewed in digital format.

Over the past weeks, you have become familiar with the fundamentals of a small ad-hoc game engine (see tutorial) that allows you to create tile-based games in two dimensions, similar to RPGs.

The goal of this mini-project is to leverage this knowledge to create one or more small concrete implementations of a cooperative variant named `ICcoOp`, a type of game involving **two main characters**. These characters will collaborate to achieve objectives or defeat enemies. The base game you are tasked with creating is inspired by games like *Fireboy and Watergirl*. Figure 1 shows some fragments of the basic draft that you can then enrich as you wish, according to your imagination and creativity. Section 6 also includes a short video example of a potential game you could create.

Beyond its entertaining aspect, this mini-project will naturally allow you to practice the fundamental concepts of object-oriented programming. It will let you experience how a design situated at an appropriate level of abstraction enables the creation of programs that are easily extensible and adaptable to different contexts.

You will progressively increase the complexity of the desired features and the interactions between components, step by step.

The project consists of four **mandatory** steps and one optional step:

- Step 1 (”Basic Game”): By the end of this step, you will have created, using the tools of the provided game engine, a basic instance of `ICCoOp` where two main characters can follow each other from one area to another and interact with objects.
- Step 2 (”First Cooperation”): During this step, the two characters will begin to assist each other in facing hostile walls. Rudimentary dialogues will be introduced, along with tracking the characters’ health points.
- Step 3 (”Belligerent Adversaries”): This step will enable your characters to face enemies and engage in battles with them using equipment.
- Step 4 (”Final Challenges”): This step will introduce new areas with new challenges to tackle cooperatively, utilizing the concept of logical signals.
- Step 5 (Optional Extensions): During this step, various more open-ended extensions will be proposed, allowing you to enhance the game created in the previous step or to create new games of your own design.

Code a few extensions (of your choice) to earn bonus points and/or participate to the contest.

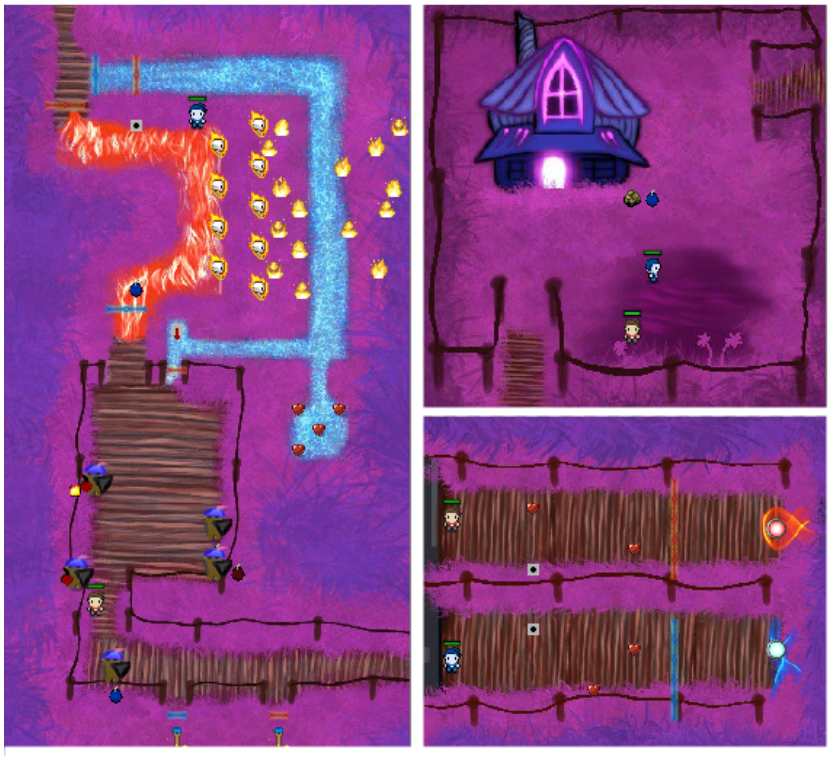

Figure 1: Some scenes from the game where two players help each other from area to area, collecting items and battling various hostile (or not) creatures.

Here are the main instructions/guidelines to follow for coding the project:

1. **You will not modify the code of** `game-engine`. Beware of IntelliJ's «quick fix» proposal.
2. The project must be coded using standard Java tools (imports starting with `java.` or `javax.`). If you have doubts about using a particular library, ask us, and be especially cautious with the alternatives IntelliJ might suggest importing on your machine. The project specifically uses the `Color` class. You must use the `java.awt.Color` implementation and not other implementations from various alternative packages.
3. Your methods must be documented according to Javadoc standards (refer to the the class `TextGraphics` of `game-engine` for inspiration).
4. Your code must adhere to standard naming conventions and be properly **modularized and encapsulated**. In particular, avoid intrusive public getters on modifiable objects.

(continued on the next page . . .)

4. The instructions provided may sometimes be very detailed. **This does not mean they are exhaustive**. The methods and attributes needed to achieve the desired functionalities are not all described, and it will be up to you to introduce them as you see fit, ensuring proper encapsulation.
5. Your project **must not be stored on a public repository** (e.g., GitHub). For those familiar with Git, we recommend using GitLab: https://gitlab.epfl.ch/, but any repository type is acceptable as long as it is private.

The first step is deliberately guided. Its primary goal is to deepen your understanding of the provided framework and to begin leveraging it in a concrete manner.

2 ICoop de base ICoop Basics (Step 1)

The goal of this step is to start creating your own game of the `ICoop` type.

This basic version will include two main characters that can be controlled individually and are capable of interacting with objects. The latter point will be implemented using the more general mechanism of interactions between actors, as described in tutorial 3.

This game will involve:

- Two main characters;
- Doors that the characters can pass through. This will involve a contact interaction: a character must be in a cell containing an "actor" labeled as a "door" to pass through it;
- An explosive that the characters can activate (via remote interaction);
- a rock that the explosive, once activated, can destroy (again via remote interaction).

The two characters will follow each other during area transitions.

You will work in the provided `icoop` package.

2.1 Preparing the ICoop `ICoop`

> The solution for the tutorial is provided in the `tutos` folder, and you can use it as inspiration to get started.

Prepare an `ICoop` game inspired by `Tuto2`. This game will initially consist of :

- The `ICoopPlayer` class, which models a main character, should be placed in `icoop.actor`; leave this class empty for now, we will come back to it a bit later;
- the `ICoop` class, equivalent to `Tuto2`, to be placed in the `icoop` package; don't forget to adapt the `getTitle()` method, which will return a name of your choice (e.g., *`"ICoop"`*);
- une classe `ICoopArea` équivalente à `Tuto2Area`, à placer dans un sous-paquetage `icoop.area`;
- A class `ICoopArea` equivalent to `Tuto2Area`, to be placed in a sub-package `icoop.area`;
- Classes `Spawn` and `OrbWay` inheriting from `ICoopArea`, to be placed in the package `icoop.area` (they are equivalent to the classes `Ferme` or `Village` in the tutorial). For the title, use *`"Spawn"`* and *`"OrbWay"`* to ensure proper alignment with the graphical resources;
- A class `ICoopBehavior` analogous to `Tuto2Behavior` to be placed in `icoop`, which will contain a public class `ICoopCell` equivalent to `Tuto2Cell`.

You will ensure that the initial positions of the main characters to come are dictated by a method returning specific values for each area: for example, `(13,6)` for the red character and `(14,6)` for the blue one in the area `Spawn`, and `(1,12)` and `(1,5)` for the area `OrbWay`.

For the type of games we're interested in, it is best to centre the view as closely as possible in the window. To do this, simply add the following redefinition to `ICoopArea` :

```
@Override
public boolean isViewCentered() { return true; }
```

2.2 Adaptation of ICoopBehavior

First, a few updates need to be made to the classes `ICoopBehavior` and `ICoopCell`. The enumerated type describing the cell types will have an additional field indicating whether the cell can be flown over or not (a `canWalk` field, similar to `isWalkable`, and a `canFly` field indicating if one can fly over the cell and which can be used later in the project) :

```
NULL(0, false, false),
WALL(-16777216, false, false),
IMPASSABLE(-8750470, false, true),
INTERACT(-256, true, true),
DOOR(-195580, true, true),
WALKABLE(-1, true, true),
ROCK(-16777204, true, true),
OBSTACLE(-16723187, true, true)
```

Additionally, the nature of the cells will not be the only factor determining whether a cell allows an actor to enter (via its method `canEnter`). The actors already present in the cell will also play a role. In the case of the `ICoop` game, a cell can have at most one non-passable actor within its set of entities (inherited from `Cell`): two non-passable actors cannot coexist in the same cell !

The `takeCellSpace()` method indicates whether an actor is passable (allows others to "step on it") or not. It is passable if its `takeCellSpace()` method returns `false`.

Make the suggested adaptations above in the `ICoopBehavior` class.

2.2.1 Task

You are required to implement the concepts described above according to the given specifications and constraints. Run your `ICoop` game. Verify that the empty area in Figure 2 is displayed.

2.3 Draft of Main Characters

For now, code `ICoopPlayer` in the same spirit as `GhostPlayer`.

One important difference is that the characters, like many upcoming actors, will "serve" a natural element (typically the "water" or "fire" element).

You should therefore introduce a category called `ElementalEntity`. Any actor behaving as an `ElementalEntity` must define a method `element()` that returns the natural element the actor serves. This will, of course, apply to the main characters. The element served by the character will be specified during construction.

Figure 2: Welcome area

2.3.1 Drawing

Another significant difference from `GhostPlayer` is that the image used to draw an `ICoopPlayer` will depend on its *orientation* and will be *animated*.

To draw the character, you will use an object of type `OrientedAnimation`. There will be several possible animations as the project progresses, but for now, only the basic animation needs to be implemented. This can be constructed as described in Appendix 7.3.1[1]. Since we will create two characters, the name of the image used to create the animation will naturally differ for each. You should consider this name as configurable during the character's construction.

2.3.2 Behavior

An `ICoopPlayer` is a non-passable actor that is oriented downward by default. It behaves (updates itself) like a `MoveableAreaEntity` but with additional features:

- It must be movable using keys, similar to the `GhostPlayer` coded in the tutorial;
- Its current animation must also be updated based on the following algorithm: if a movement is in progress, the current animation must undergo an `update`; otherwise, it must undergo a `reset`.

One of the unique aspects of `ICoop` games is that two characters need to be controlled, requiring two different sets of keys. The specific set of keys for a given character can be configured during construction as a `KeyBinding.PlayerKeyBindings`. Refer to Appendix 7.1 to understand this data type. Adapt your movement method so that the "directional arrow" keys used in the tutorial are replaced by `keys.up()`, `keys.down()`, `keys.right()`, and `keys.left()`, where `keys` is the set of keys associated with the character.

Once this draft of `ICoopPlayer` is complete, you can start adapting the `begin` method of `ICoop` and testing your game. Initially, you will introduce only one of the two characters.

At launch, a player of type `ICoopPlayer` will be created at its designated position in

[1] Refer to the documentation for `OrientedAnimation` if you want to understand the role of the parameters in creating the animation.

Figure 3: Door from “Spawn” area to “OrbWay” area

the current area. The name of the images associated with its basic animation will be *`"icoop/player"`*, and the keys used to control it will be those specified by `KeyBindings.RED_PLAYER_KEY_BINDINGS`.

2.3.3 Task

You are required to code the concepts described above according to the given specifications and constraints.

Run your `ICoop` game. Verify that it behaves as shown in the following short video: https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/startStep1.mp4. Specifically, ensure that:

1. The game starts by displaying an `ICoopPlayer` facing forward;
2. The character can move freely across the area using the keys specified by `KeyBindings.RED_PLAYER_KEY_BINDINGS` (`W`, `A`, `S`, and `D` if unchanged) but cannot leave the area;
3. Its graphical representation adapts correctly to its orientation;
4. Its movements are animated.

2.4 Doors: Contact Interactions

You are now tasked with applying the interaction framework suggested in the third part of the tutorial (Clickable Link)[2].

The first interaction we will focus on is a contact interaction with a ”door” actor that will allow an `ICoopPlayer` to transition from one area to another.

[2] A complementary video is also available to explain the implementation of interactions: mp2-interactions.mp4

2.4.1 Doors

As explained in the tutorial, if an entity undergoes interactions, it must be modeled as an `Interactable`; if it initiates interactions, it must be modeled as an `Interactor` (and it can be both).

As a first category of `Interactable`, we already have `ICoopCell` (for example, if a character is in contact with certain types of cells, it could adopt a different behavior, such as sliding). We will not exploit this feature for now but will instead introduce a new `Interactable`, "door," which will have a more immediate utility.

A door, `Door`, to be coded in the `icoop.actor` package, is an actor of type `AreaEntity` that allows transitions to a destination area.

This actor is characterized by:

- the name of the area to which it allows a transition (a string);
- and the set of arrival coordinates (`DiscreteCoordinates`) in the destination area. This set will be variable in size, though for the purposes of this project, it will typically have a size of two : arrival coordinates for the red character followed by those for the blue one.

The tutorial introduced the notion of a signal, which can be used here to make the game more engaging by incorporating puzzle-solving elements. For example, to obtain a weapon for defense, the character might need to find a key that opens a door to a room containing the desired weapon. The provided tools could typically catalog a key actor as a signal attached to this door: once the signal is activated (the key is picked up), the door opens.

A "door" actor will therefore also be characterized by a *signal* (of type `Logic`) that models the conditions under which it is open or closed.

The constructor for a `Door` takes the following parameters: the name of the destination area, the signal conditioning its opening, the possible arrival coordinates in the destination area, the area it belongs to, the position of its main cell, and optionally the list of coordinates of the cells it occupies in addition to its main cell (expressed using an ellipsis in a second constructor).

The method `getCurrentCells()` must therefore be overridden in `Door` to return the set of these occupied coordinates.

A `Door` is a passable actor that accepts contact interactions and whose drawing method does nothing by default. Indeed, doors will coincide with elements of the scenery and will not be drawn as distinct entities. As a result, their orientation is not very important.

You are tasked with introducing this new actor and registering an instance of it with the following characteristics in the `Spawn` area:

destination	arrival coord.	signal	main cell	other coord.
"OrbWay"	(1,12)(1,5)	Logic.TRUE	(19,15)	(19,16)

This amounts to placing a door at the location indicated in Figure 3.

Also, register the doors that allow the transition in the other direction, from `OrbWay` to `Spawn`:

destination	arrival coord.	Signal	main cell	other coord.
”Spawn”	(18,16),(18,15)	Logic.TRUE	(0,14)	(0,13),(0,12),(0,11), (0,10)
”Spawn”	(18,16),(18,15)	Logic.TRUE	(0,8)	(0,7), (0,6), (0,5), (0,4)

The value `Logic.TRUE` indicates that these doors are open without any condition. The locations of the doors can be determined by counting the coordinates of the red tiles in the ”behavior” images (for example, see `resources/images/behaviors/Spawn.png`).

2.4.2 `ICoopPlayer` as an `Interactor`

Now that we have a concrete entity capable of undergoing interactions, let's focus on `ICoopPlayer` as an entity initiating interactions. Start by ensuring that all `ICoopPlayer` instances become `Interactor` entities (they can initiate interactions with `Interactable` entities).

As an `Interactor`, `ICoopPlayer` must define the following methods:

- `getCurrentCells()`: its current cells, which are reduced to the set containing only its main cell (as you have already implemented);
- `getFieldOfViewCells()`: the cells forming its field of view, consisting of the single cell it is facing:

```
Collections.singletonList
(getCurrentMainCellCoordinates().jump(getOrientation().toVector()));
```

As an `Interactor`, it will always request all contact interactions (`wantsCellInteraction` always returns `true`). Whether it requests remote interactions (`wantsViewInteraction`) will be determined by the user: pressing its `keyBindings.useItem()` key will indicate that the character wants a remote interaction. For example, if a character is facing an explosive and the user wants to activate it, they press the key corresponding to `keyBindings.useItem()` (by default, `E` for the red player and `O` for the blue player).

We can now handle the possible interactions at this stage: in the subpackage `icoop.handler`, complete the `ICoopInteractionVisitor` interface, which inherits from `AreaInteractionVisitor`. This interface must provide default definitions for the `interactWith` methods for all `Interactor` entities in the `ICoop` game with:

- a game cell (`ICoopCell`);
- a main character in the game (`ICoopPlayer`);
- a door (`Door`).

These (default) definitions will have an empty body to indicate that, by default, the interaction does nothing (`ICoopPlayer`, as an `Interactor` in the `ICoop` game, must provide a more specific definition of these methods if necessary).

Every concrete `Interactable` must now indicate that it accepts having its interactions managed by an interaction handler of type `ICoopInteractionVisitor`.

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

This method may contain more code depending on the case, but for now, it is sufficient for the three types of `Interactable` available: `ICoopPlayer`, `ICoopCell`, and `Door`.

To allow `ICoopPlayer` to manage interactions of interest more specifically, define a private nested class `ICoopPlayerInteractionHandler` inside the `ICoopPlayer` class, implementing `ICoopInteractionVisitor`. Add the necessary definitions to handle interactions with doors more precisely, such that during a contact interaction with a door, if the door's signal is activated (method `isOn()`), the character:

1. leaves its current area (method `leaveArea()`);
2. arrives in the destination area at the first possible arrival coordinate for that area (method `setCurrentArea`).

Note: Only the game knows all the areas and is capable of transitioning the character from one area to another using the `leaveArea` and `setCurrentArea` methods. Therefore, this process cannot be directly coded in the character's interaction method with the door. Instead, the method should only inform the character that it is traversing a door (and specify which one), and the character must provide the necessary information to the game!

2.4.3 Task

You are required to implement the concepts and processes described above according to the given specifications and constraints. Run your `ICoop` game. Verify that the main character can transition from the *`"Spawn"`* room to the *`"OrbWay"`* room (and vice versa) through the location corresponding to the door in Figure 3. The lower door in the `OrbWay` room will only be reachable once the second character is introduced.

2.5 Second Character

We now introduce what makes `ICoop` games unique: the presence of a second character. The introduction of this character raises the question of how to properly center the camera, which until now has followed the single character as in the tutorial.

The adopted idea is to center the camera on a point equidistant from the two characters. To simplify the required calculations, a `CenterOfMass` actor is provided.

You are now tasked with evolving the `ICoop` game to:

- include a second character as a core feature;
- ensure that the `begin` method creates this character and centers the camera on a `CenterOfMass` object constructed using the two characters.

The second character will:

- start at the position dictated for them in the current area;
- use the image *`"icoop/player2"`* as their associated sprite;
- be controlled with the keys specified by `KeyBindings.BLUE_PLAYER_KEY_BINDINGS`.

If this character transitions via a door, it must arrive at the *second* arrival coordinate in the destination area. Additionally, ensure that the two characters follow each other: if one

character passes through a door, the other follows, arriving at the destination coordinate assigned to them by the door. Both characters will always occupy the same area!

Finally, to ensure that the initially chosen scale factor does not hinder the joint observation of both characters, the `update` method of the game dynamically recalculates it (and reassigns it using `setCameraScaleFactor`) with a heuristic formula such as:

```
max(DEFAULT_SCALE_FACTOR, DEFAULT_SCALE_FACTOR * 0.75
                        + distance(position_playerA, position_playerB) / 2)
```

You are free to refine this formula as you see fit. The distance can be calculated as the length of the vector `position_playerA - position_playerB` (methods `sub` and `getLength` of `Vector`).

2.5.1 Task

Once the suggested modifications have been implemented, run your `ICoop` game. Verify that :

- a second character appears at the start of the game;
- it can be controlled using its specific key bindings;
- it behaves like the first character but is drawn with a blue-tinted animation;
- each character can transition from *`"Spawn"`* to *`"OrbWay"`* and vice versa. In this case, the other character follows, and both characters arrive in the area at the positions designated for them by the crossed door. An example of a door passage is provided in the following video:
  https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/secondPlayer.mp4.

2.6 Explosives: Remote Interactions

To solidify the interaction management mechanism, the final part of this step involves introducing two new actors: explosives and obstacles, with the idea that an explosive can destroy certain obstacles *remotely*.

2.6.1 Obstacles and Explosives

An *obstacle* is an actor derived from `AreaEntity` and drawable by default using the sprite *`"rock.2"`*. By default, it is non-passable and it unconditionally accepts any type of interaction. *Rocks* are a specific type of obstacle, drawable using the sprite *`"rock.1"`*, and they can be destroyed: they are non-passable and are only drawn if they are not destroyed.

An *explosive* is a passable actor inheriting from `AreaEntity` and playing the role of an `Interactor`. It can be activated and is associated with a timer (an integer) initialized at its creation. It is created in a deactivated state. Once activated, on each of its `update`, its timer is decremented (e.g., by one unit). When the timer reaches zero, the explosion occurs (the explosive enters the explosion state), which triggers the display of a specific animation, after which it disappears from the game.

An explosive, as an `Interactor`, requests remote or contact interactions when it is in the explosion state. For now, it interacts remotely only with rocks. The effect of the interaction with rocks is, of course, to destroy them.

The explosive's field of action, `getFieldOfViewCells()`, consists of all 4 cells immediately adjacent vertically and horizontally to its main cell. `getCurrentCells()` should be coded in the same way as for `ICoopPlayer`.

As an `Interactable`, an explosive can undergo remote interactions if it has not exploded, and contact interactions only if it is neither activated nor exploded.

Graphically, the explosive can be represented with an animation. It must disappear after its explosion, which should be accompanied by another specific animation. Refer to Appendix 7.3.2 for more precise details on this.

Finally, ensure that when a character interacts remotely with an explosive, it becomes activated. Remember that characters express the desire for a remote interaction using their `useItem()` key (default: `'E'` and `'O'` respectively).

To complete this step, create an explosive and a rock at respective positions `(11,10)` and `(10,10)` in `Spawn`.

2.6.2 Task

You are required to implement the concepts described above according to the given specifications and constraints.

Run your `ICoop` game. Verify that it behaves as shown in the following short video: https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step1Explosion.mp4; specifically:

- that characters cannot pass through the rock but can pass through the explosive;
- that each character can activate the explosive via remote interaction but only remotely;
- that once the explosive is activated, the timer starts and the explosion occurs after a short delay;
- that the explosion is accompanied by an animation and that there is no trace of the explosive once the animation ends;
- that the explosion destroys the rock, and its location, now empty, becomes passable for the characters.

2.7 Reset

To facilitate testing, you will equip `ICoop` with the following controls: the key `KeyBindings.RESET_GAME` ('R' by default) should perform a game reset, and `KeyBindings.RESET_AREA` ('T' by default) should perform an area reset.

In the first case, the game should restart in the same conditions as when it was first launched. In the second case, **only** the current area is restarted. In both situations, the `begin` methods should prove useful. For restarting the current area, you will need to find a way to recreate/replace the characters at their "spawn" positions. Note that the `begin` methods for a game or an area (as provided by the template) clear the relevant collections

(areas, actors, etc.); thus, there is no need to handle this explicitly.

2.7.1 Task

Verify that both reset keys function as expected and that the characters reappear at the "spawn" positions specified by the areas. Test both keys in both existing areas.

2.8 Validation of Step 1

To validate this step, all the checks from sections 2.2.1, 2.3.3, 2.4.3, 2.5.1, 2.6.2, and 2.7.1 must have been completed.

The `ICoop` game, whose behavior satisfies the validation steps above, must be submitted at the end of the project.

Question 1

The framework established, as explained in the tutorials and practically applied in this first part of the project, might initially seem unnecessarily complex. Its advantage is that it models, in a very general and abstract way, the inherent needs of many games where actors move on a grid and interact either with each other or with the grid's contents. How could you leverage this to implement a Pacman-like game? What would need to be defined?

In the rest of the project, you will need to code many more interactions between actors or with the cells. All upcoming interactions must be implemented according to the framework established in this part and must not rely on type tests on objects.

3 First Cooperation (Step 2)

The objective of this second part of the project is to introduce hostile actors that will harm the main characters by inflicting damage. Means of protection against these dangers will also be introduced to help the characters survive their now perilous adventures.

Specifically, this step aims to enhance the game by introducing the following new elements:

- Characters will become vulnerable to certain types of damage that cause them to lose health points; you'll anticipate water, fire or physical damages , and these damages won't just apply to characters;
- Fire walls will inflict fire damage on characters;
- Water walls will inflict water damage;
- Orbs (fire or water type) will grant characters immunity to these types of damage;
- Pressure plates will disable walls: the blue character can stand on a pressure plate to disable a fire wall (and vice versa for water walls and the red character), enabling both characters to start cooperating to overcome these hostile walls.

The implementation of these new elements will again rely on the concept of signals already used for doors in the previous step.

During this step, visual elements will be enhanced to display the characters' health status via a health bar. Simple dialogues will also provide useful hints during gameplay.

Below, you will find the specifications to follow, along with some guidelines.

3.1 Health Bar

Figure 4: Health bars for the characters

Each `ICoopPlayer` will now be equipped with a health bar displaying their "health status" (see Figure 4). A `Health` class is provided in the `actor` directory to simplify the implementation of this feature.

You will consider that an `ICoopPlayer` has a maximum number of health points (an integer identical for all characters, e.g., 5). Each `ICoopPlayer` will be associated with a health bar initialized as follows:

```
new Health(this, Transform.I.translated(0, 1.75f), MAX_LIFE,
   true);
```

The first parameter attaches the bar to the current actor (as health points were in the tutorial), the second indicates how much to offset the bar from the actor in its relative frame and the third one is a integer corresponding to the greatest number of points. The

last parameter allows coloring the health bar green or red depending on whether the actor is friendly or hostile (default is friendly).

The health bar must be drawn along with the character.

Study the `Health` class functionality to determine how to increase or decrease the health points it represents.

Then, model the following for an `ICoopPlayer`:

- They can become immune (temporarily or not) to a certain type of damage (one type at a time; initially, they are vulnerable to everything);
- They can lose a given number of health points due to a specific type of damage (e.g., an explosive inflicts physical damage causing a fixed loss of health points);
- Once they have lost health points (regardless of the cause), they benefit from a short immunity period, the duration of which is identical for all characters.

The method that handles health loss due to a specific type of damage can be implemented using the following algorithm: if the character is immune to the given damage type or is in their immunity period, nothing happens. Otherwise, their health bar should reflect the desired loss of health points.

Complete the `Explosive` class so that its interaction with an `ICoopPlayer` inflicts physical damage, causing a fixed loss of health points. This value will be identical for all explosives (e.g., 2). The duration of the immunity period will also be the same for all characters for simplicity.

Guideline: You can implement the immunity period by drawing inspiration from the timer mechanism of the explosive. A value such as 24 can be used for the immunity duration.

Character Drawing Adjustment: To visualize immunity status, modify the drawing of `ICoopPlayer` so that, when immune, their animation is only drawn every other frame (e.g., only when the immunity timer is even).

Reset Adjustment: Reset keys must now ensure that characters restart with a full health bar for both game resets and current area resets. For simplicity, consider that they also restart with no immunity period.

End of Life: Adapt your `ICoop` game so that the loss of all health points by either character triggers a reset of the current area.

3.1.1 Task

You are required to implement the concepts described above in accordance with the specifications and constraints.

Run your `ICoop` game. Verify that it behaves as shown in the following short video: https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-1.mp4; specifically:

Figure 5: Help dialogues.

1. A health bar appears near each character, is fully green at the start, and follows the associated character as they move;
2. The explosive, once detonated, causes health point loss for the character in its proximity (e.g., if the character that activated the explosive did not move far enough away);
3. The character's sprite flashes when they take damage, indicating they have entered an immunity period.

Also, verify that the reset function restores health points and that the "death" of either character triggers a reset of the current area. The immunity to specific types of damage will be tested later.

3.2 Dialogues

Many keys are already involved in interactions, and it is useful to document them at the start of the game. Additionally, games of this type can quickly become unplayable without well-timed hints. The idea here is to introduce a "dialogue" component to meet these needs, as shown in the example in Figure 5.

The template already provides a `Dialog` class (in the `engine.actor` package) that can be used for this purpose. To create a dialogue, simply associate it with text stored in the resources (subfolder `dialogs`), for example:

```
new Dialog("orb_water_msg");
```

The `update` method of `Dialog` allows the text to be scrolled. The `isCompleted` method returns `true` when all the text has been displayed.

Dialogues will be used here to provide hints from the game to the player. This can be implemented by equipping `ICoop` with:

- An attribute of type `Dialog` to model the hint to be displayed at a given time (the

"active dialogue"). A dialogue is considered active if this attribute is not `null`;
- A `setActiveDialog` method to assign a value to this attribute.

Next, modify the `ICoop` class so that:

- The current area does not update when a dialogue is active in the game (the area will only be drawn). The active dialogue must, of course, also be drawn;
- The active dialogue (if any) is updated when the `NEXT_DIALOG` key is pressed:

  ```
  kbd.get(KeyBindings.NEXT_DIALOG).isPressed()
  ```

  where `kbd` is the `Keyboard` object associated with the game;
- The game resumes normal behavior when an active dialogue is completed (`isCompleted()` method of `Dialog`).

3.2.1 Activating a Dialogue

The question then arises of how to assign a value to the game's "active dialogue" attribute, given that it is generally a game component, not the game itself, that knows when this should happen.

For example, if you want a hint to be displayed when a character interacts with an actor, the method managing this interaction should call the method that instantiates the game's active dialogue. This method (or the class to which it belongs) must then be aware of the game. Similarly, if a hint needs to be provided when a specific area begins, the area must know about the game to allow the game to instantiate the active dialogue.

It is suggested to use the concept of an interface here to avoid exposing the entire game to components that need to activate a dialogue.

Implement an interface `DialogHandler` containing only a single method, `void publish(Dialog)`. Then ensure the game implements this interface and overrides the `publish` method to assign the dialogue to be published as the current game dialogue.

Finally, modify the `Spawn` class so that:

- The area knows the game to which it belongs but only as a `DialogHandler` (a less intrusive abstract view);
- On the first call to its `update` method, the game's active dialogue becomes the *`"welcome"`* dialogue. This dialogue should not reappear if the *`"Spawn"`* area is revisited after leaving for another area.

Welcome Dialogue and "Reset": The `GAME_RESET` key restarts the game from the beginning, so it is normal for the welcome dialogue to appear again after using this key. However, this dialogue should not reappear when the `AREA_RESET` key is used.

3.2.2 Task

Once the concepts described above are implemented according to the specifications and constraints, run your `ICoop` game. Verify:

1. That the introductory dialogue appears at the start of the game and can be scrolled using the `NEXT_DIALOG` key;

2. That while the dialogue is displayed, the characters no longer respond to controls;
3. That once the dialogue ends, the game resumes the behavior it had before;
4. That the reset keys work as intended regarding the welcome dialogue.

3.3 Collectible Items

As they move around, the main characters can collect items that provide benefits (e.g., healing, invulnerability) or serve as useful equipment.

You are now required to introduce an abstract class, `ICoopCollectable`, modeling the default general behavior of "collectible" items in `ICoop`-type games. This class will inherit from the `CollectableAreaEntity` class provided in the template (package `areagame.actor`). By default, an `ICoopCollectable` is traversable and, as an `Interactable`, only allows contact interactions at this level of abstraction (this can obviously be refined in subclasses). Finally, an `ICoopCollectable` must disappear from the area when collected.

3.3.1 Collecting Explosives

Modify your code so that characters can collect explosives through contact interactions.

Guideline: Use the boolean parameter `isCellInteraction` to differentiate between contact and remote interactions in the `interactWith` methods specific to the actors.

3.3.2 Task

Once the concepts described above are implemented according to the specifications, run your `ICoop` game. Verify that explosives can now be collected through contact interaction with the characters but only if they are neither activated nor exploding (feel free to add them at locations of your choice). Once collected, they should simply disappear from the area. Also, verify that remote interactions with explosives work as they did before.

3.3.3 The `ElementalItem`

Next, introduce a special category of `ICoopCollectable`. These are "collectible" objects serving a natural element (`ElementalItem`). They will behave both as `ElementalEntity` and logical signals (`Logic`) and will not be instantiable at this level of abstraction.

The area of belonging of an `ElementalItem`, its starting position, orientation, and the natural element it serves are provided at construction.

As a `Logic`, an `ElementalItem` is "on" (`isOn()` returning `true`) when collected and "off" (`isOff()` returning `true`) otherwise.

To spice up interactions, ensure that an `ElementalItem` can only be collected by an entity of the same elemental category (e.g., a fire item can only be collected by a fire actor).

3.3.4 The Orbs

A first concrete type of `ElementalItem` to implement is the orb, modeled by the `Orb` class. An orb is an `ElementalItem` that protects the collector against specific types of damage.

A *water orb* provides protection against water damage, and its collection should trigger the display of the `orb_water_msg` dialogue.

A *fire orb* provides protection against fire damage, and its collection should trigger the display of the `orb_fire_msg` dialogue. Its area of belonging, the element it serves (fire or water), and its starting position are provided as construction parameters. It is oriented upward by default.

An orb is drawn using an animation. Refer to Appendix 7.3.3 for more detailed information.

When a character enters into contact interaction with an orb, a dialogue should appear: *`"orb_water_msg"`* for water orbs and *`"orb_fire_msg"`* for fire orbs. Once the dialogue ends, the orb should disappear (i.e., be collected).

Hint: An enumerated type `OrbType`, specifying the element served, the dialogue to display, and the `spriteYDelta` (as used by the animation), might be useful here.

> **Question 2**
>
> How would you propose allowing an orb to communicate the message to activate in the game without using intrusive getters?

To test this new type of object, ensure that the `OrbWay` area is initialized with a fire orb at position `(17,12)` and a water orb at position `(17,6)`.

3.3.5 Task

Once the concepts described above are implemented according to the specifications and constraints, run your `ICoop` game. Verify that it behaves as shown in the following video: https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-3.mp4; specifically:

1. That the `OrbWay` room contains a fire orb and a water orb at the desired positions, both animated;
2. That the red character can collect the fire orb and the blue character can do the same with the water orb;
3. That the specific dialogue for each orb type is displayed during contact interaction with the characters;
4. And that the orbs disappear once the dialogues are completed.

The fact that an `ElementalItem` can only be collected by a character serving the same element will be tested later in the project.

3.4 Elemental Walls

An elemental wall (`ElementalWall`) is a wall that behaves like an `ElementalEntity`. It can be active or inactive (modeled using a `Logic` attribute). It can be destroyed and must

disappear from the area if it is.

Its area of belonging, orientation, starting position, the logical signal determining whether it is active or not, and the sprite used for drawing are provided at construction. This actor is traversable by default and only draws itself if it is active.

As an `Interactable`, an `ElementalWall` can engage in contact and remote interactions. As an `Interactor`, it always enforces contact interactions but not remote ones.

When it interacts with an `ICoopPlayer` and is active, it inflicts damage. We will consider that the method for applying damage cannot be defined at this level of abstraction.

All sprites for drawing it can be extracted as follows:

```
Sprite[] wallSprites =  RPGSprite.extractSprites(spriteName,
   4, 1, 1, this, Vector.ZERO, 256, 256)
```

To draw it specifically, use the sprite corresponding to its orientation:

```
wallSprites[getOrientation().ordinal()]
```

Two specific types of `ElementalWall` will be introduced:

- Fire walls, drawn using the *`"fire_wall"`* sprites, inflict fire damage on `ICoopPlayer` by causing a one-point life loss;
- Water walls, drawn using the *`"water_wall"`* sprites, inflict water damage on `ICoopPlayer` with the same life loss.

3.4.1 Elemental Category Role

To complicate interactions, ensure that entities of different elemental categories cannot coexist in the same grid cell. For example, a water wall will only allow a water character to pass, and a fire wall will only allow a fire character.

Question 3

What modifications would you propose for `ICoopBehavior.ICoopCell` to meet this requirement? (the tests on the elemental category are not considered to be type tests because we are not testing the actual nature of the instances but only one of their behavioral facets).

To test this new type of actor, ensure that `OrbWay` is initialized with five fire walls oriented left at positions `(12, 10+i)` and five water walls at positions `(12, 4+i)`, where `i` is the wall number, varying from 0 to 4 (see Figure 6). At this stage, they will always be active: the associated signal will be `Logic.TRUE`.

Also, for testing purposes, add a water wall at position `(7,12)` and a fire wall at position `(7,6)`. Both walls will be associated with a constantly active signal.

3.4.2 Task

Once the concepts described above are implemented according to the specifications and constraints, run your `ICoop` game. Verify that it behaves as shown in the following video:

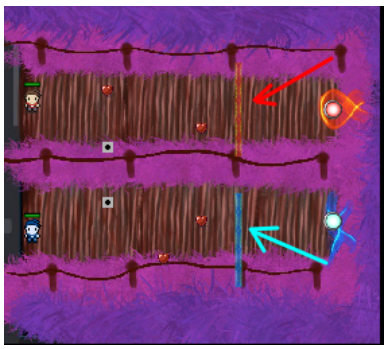

Figure 6: Les obstacles désignés par les flèches rouge et bleue sont constitués de 5 murs juxtaposés verticalement.

https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-4.mp4; specifically:

1. That movement across areas remains as it was in previous steps;
2. That the `OrbWay` room contains fire and water walls as shown in the video;
3. That the walls only allow characters of their elemental category to pass;
4. And that the walls inflict damage on characters when they pass through.

3.5 Assistance, Healing, and Invulnerability Periods

The goal now is to equip characters with means to mitigate the damage caused by the walls. Three methods will be provided:

- Collecting orbs to become invulnerable to specific types of damage;
- Collecting hearts to restore health points;
- Each character can help the other by deactivating their wall (introducing a cooperation mechanism between them!).

First, complete the interaction between `ICoopPlayer` and orbs so that collecting an orb makes the character invulnerable to the damage it protects against: fire orbs protect against fire damage, and water orbs act similarly.

Next, introduce a new collectible object, the *heart* actor (`Heart`), whose contact interaction with an `ICoopPlayer` restores one health point to the character (and removes the heart from the area). A *heart* does not serve any natural element (there are no fire or water hearts!).

Graphically, a heart will animate according to the details in Appendix 7.3.4.

Finally, create a *pressure plate* actor (`PressurePlate`) that behaves as a logical signal (`Logic`). When a character steps on it, it activates (`isOn()` returning `true`).

You have access to the *`"GroundPlateOff"`* graphic resource to create the `RPGSprite` for drawing. This actor is oriented downward by default.

To test your new developments, ensure that `OrbWay` is initialized with four hearts at positions of your choice (for example, at `(8,4)`, `(10,6)`, `(5,13)`, and `(10,11)`). You will also create a first pressure plate at position `(5,7)` and a second at position `(5,10)`. Finally, modify the construction of the walls so that the fire wall deactivates when the first pressure plate is active and the water wall deactivates when the second pressure plate is active. This will allow each character to help the other access the orb without taking damage from the walls.

3.5.1 Task

Once the previously described concepts have been implemented according to the specifications and constraints provided, launch your game `ICoop`. Verify that it behaves as shown in the following short video: https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-5.mp4 ; specifically:

1. that the blue character can deactivate the fire wall by stepping on the pressure plate accessible to them ;
2. that a similar behavior is observed for the red character ;
3. that the orbs can be collected without damage once the walls are deactivated ;
4. and that the walls reappear when the characters leave the pressure plate controlling them.

3.6 Maze Area

To conclude this feature-rich step, you are tasked with coding a new area named `Maze` (with the title `"Maze"`). In this area, you will register actors according to the configuration provided in Appendix 7.2.

This room can be accessed from the `Spawn` room via a door configured as follows:

Destination	Arrival Coordinates	Signal	Main Cell	Other Cells
”Maze”	(2,39), (3,39)	Logic.TRUE	(4,0)	(5,0)

The reverse access from `Maze` to `Spawn` is not possible (no door is provided: the characters are currently trapped in the maze area... which should add a bit of suspense!). The spawn positions in the Maze are `(2,39)` for the red character and `(3,39)` for the blue one.

3.6.1 Task

Once the concepts described above are implemented according to the specifications and constraints, run your `ICoop` game. Verify:

1. That the characters can access the maze area through the specified door and vice-versa (as shown in the video: https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-6.mp4) ;
2. That the various components appear and behave according to the specifications in Appendix 7.2.

3.7 Validation of Step 2

To validate this step, all checks from sections 3.1.1, 3.2.2, 3.3.2, 3.3.5, 3.4.2, 3.5.1, and 3.6.1 must have been completed.

The game `ICoop`, whose behavior meets the validation steps above, must be submitted at the end of the project.

4 Enemies and Battles (Step 3)

In this third part of the project, you are tasked with enriching the design by adding enemies for the characters to face and various more "martial" equipment to set the stage for epic battles. A rudimentary inventory concept will be introduced, allowing characters to store and select the equipment they use.

Below, you will find the specifications to follow, along with some guidelines.

4.1 Equipment

Characters will have equipment they can store in an *inventory* containing various *items*.

The generic notion of inventory (`Inventory`) and inventory item (`InventoryItem`) are provided in the template (in `areagame.handler` of the `game-engine` module).

4.1.1 Inventory Items

The *functional* specification of an inventory item (`InventoryItem`, provided in `game-engine`) is that it should provide access to:

- its name (a `String`);
- and the pocket of the inventory in which it is located (an `int` representing the pocket number for simplicity).

Note that a pocket can contain multiple inventory items. For simplicity, we will only use one pocket in this project.

An `InventoryItem` should be viewed as the description of an *item model* that can be stored in an inventory (e.g., the item model "sword"), rather than as a physical object. To simplify, however, we will refer to *inventory items* rather than *inventory item models*.

ICoopItem Your `ICoop` game will have its own concrete implementation of inventory items. You are required to implement the `ICoopItem` concept, a specialization of `InventoryItem` that models the inventory items specific to the `ICoop` game. You can place this concept in your game's `handler` directory.

In the base version of `ICoop` that you are required to deliver, the possible inventory items in the game will be: swords (`Sword`), fire and water keys (`FireKey`, `WaterKey`), fire and water staffs (`FireStaff` and `WaterStaff`), and explosives (`Explosive`).

For now, there is no need to anticipate actors such as "sword," "key," or "staff." These are simply models of inventory items.

Guidelines:

- Code `ICoopItem` as an enumerated type;
- An enumerated type can implement an interface.

You may freely choose the name associated with each planned item. Additionally, anticipate that every `ICoopItem` will have a graphical representation and thus will be characterized by

a `Sprite`. You can use the images `sword.icon`, `key_blue`, `key_red`, `staff_water.icon`, `staff_fire.icon`, and `explosive` from the `resources/images/sprites/` directory. For simplicity, in the base version of the project, the inventory will have only one pocket, and all items will, by default, belong to this pocket with identifier zero.

4.1.2 Inventories

An abstract inventory (class `Inventory` provided in `game-engine`) is characterized by a set of pockets (`Pocket`) in which it is possible to add or remove a given quantity of `InventoryItem`. Review these classes broadly to anticipate the functionalities they offer.[3] The pockets in the inventory are numbered (integer indices). Note that the embedded interface `Inventory.Holder` allows you to functionally designate inventory holders: you can query them to determine if they possess a particular item using the `possess` method.

You will implement `ICoopInventory`, a specialization of `Inventory` specific to the `ICoop` game, which by default is constructed with a single pocket named as you choose. You can also place this concept in the `areagame.handler` directory of your game.

Next, you will complete the modeling of the main character `ICoopPlayer` to provide it with an inventory of type `ICoopInventory`. At the creation of the character, a few items, including one or more explosives and swords, will be hard-coded into the inventory for simplicity.[4] Naturally, `ICoopPlayer` must be listed as an inventory owner. To preserve encapsulation, it will not expose its inventory to the outside world.

You will consider that `ICoopPlayer` can use only one inventory item at a time. The currently used item will be referred to as the *current equipment*. You will add new functionalities to `ICoopPlayer`, allowing it to:

- Switch between inventory items (i.e., change its current equipment) using the `keyBindings.switchItem()` key. Switching from one item to another will be done circularly (after the last item in the list, it returns to the first). Note that the `Inventory` class provides the method `contains(InventoryItem item)` to check if any of the pockets contain the item `item`.
- Use the current equipment using the `keyBindings.useItem()` key.

Hints: To make the game more user-friendly, it is preferable that inventory items always appear in the same order when cycling through them. An attribute defining this order may be useful. However, only items actually present in the character's inventory should appear in this order.

The use of the current equipment will naturally be handled on a case-by-case basis.[5] Each piece of equipment will be handled as a specific case.

For now, you will only program the use of explosives. All other cases will be considered default cases where the character does nothing.

Using an item of type "explosive" involves creating an "explosive" actor in front of the character. This creation will only be possible if the cell in front of the character allows the

[3] The embedded GUI interface is not required in the basic version of the project.
[4] You can enhance this point in the extensions section of the project.
[5] Enumerated types are very well-suited for use with a `switch` statement.

Figure 7: Graphical display areas for character status (one area per character)

placement of an actor. A bomb used must be removed from the inventory (the quantity of this item in the inventory will decrease by one).

The explosive will be endowed with additional interactions:

- With other explosives, causing them to explode.
- With walls, destroying them.

4.1.3 Collection and Inventory

In the previous stage, three types of "collectible" actors were introduced: explosives, orbs, and hearts. A distinction must now be made between them: some collected objects can be stored in the inventory (as part of the equipment), while others cannot. Typically, orbs and hearts, once collected, will not appear in the inventory, while explosives will. You are asked to adapt your code to reflect this behavior.

4.1.4 Graphics

To visualize character-specific information (such as the current inventory equipment), you are now asked to use and complete the provided class `ICoopPlayerStatusGUI`. This class models a "graphical object describing the status of a main character". The provided template allows you to draw the small brown circle used as a support for inventory items in Figure 7 (the graphical object `gearDisplay`); you can enhance it to add other information as needed.

Complete the existing drawing method of this class to display the sprite of the current inventory item of each character within the brown circle. Specifically, you need to construct and draw an image:

```
new ImageGraphics(ResourcePath.getSprite(sprite_name), 0.5f,
    0.5f, new RegionOfInterest(0, 0, 16, 16), anchor.add(new
    Vector(0.5f, height - 1.25f)), 1, DEPTH);
```

where `sprite_name` is the name of the sprite associated with the `ICoopItem` selected by the corresponding character.

Note: If the `flipped` parameter of an `ICoopPlayerStatusGUI` is `true` then its `draw` method will display it on the right else on the left.

4.1.5 Task

Once the concepts described above are implemented according to the specifications and constraints provided, launch your `ICoop` game. Verify the following:

1. That it is possible to switch between equipment items using the specific keys for each character (default `Q` and `U`), and that the current equipment is displayed correctly.
2. That only explicitly added equipment items appear in the inventory when cycling through items using the `Tab` key.
3. That only the current equipment can be used (verify this with the explosive) using the specific keys for each character (default `E` and `O`).
4. That an explosive, when detonated, triggers the explosion of nearby explosives and destroys walls.
5. That it can no longer be selected as current equipment if there was only one and the character has used it.
6. That if the inventory is empty of explosives for a character and they collect one, the item reappears in the inventory.
7. And that the graphical description of the character's status remains intact when transitioning between areas.

In the mandatory part of the project, it is not required that a "reset" of the current area restores the inventory state to what it was when the area was first entered. For example, if an object is collected into the inventory and the area is later reset, the prior collection is not expected to be undone. You can address this issue in the extensions if desired.

4.2 Enemies

Enemies (`Foe`) are actors with the following characteristics:

- They can move on a grid.
- They have health points and a maximum number of health points.
- They die if their health points are less than or equal to zero; they then disappear from the game area after a specific visual effect. This involves an animation showing them vanishing, which is the same for all enemy types (*`"icoop/vanish"`*).
- They can be vulnerable to certain types of damage (they have a "list of vulnerabilities").
- They actively request interactions (not just receive them).
- By default, they can engage in both contact and remote interactions.

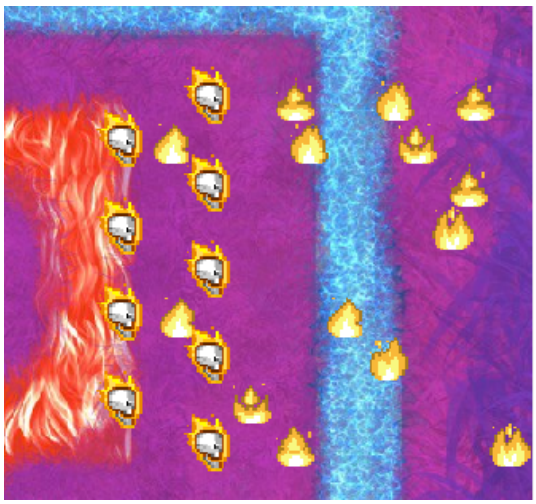

Figure 8: Fire-throwing skulls

- They can only be walked on if they are dead.
- Like characters, they can lose health points due to a specific type of damage.
- After losing health points (regardless of the cause), they have a short immunity period, the duration of which is the same for all enemies.

The maximum number of health points cannot be defined at this level of abstraction but must be accessible for all enemy types.

The possible vulnerabilities are the same as for the characters (vulnerabilities to physical, fire, or water damage).

Each type of enemy has a specific list of vulnerabilities defined at construction (e.g., fire-throwing skulls are vulnerable to water and physical damage but not fire). An enemy starts with the maximum number of health points.

Enemies disappear from the game area when they die (life points less than or equal to zero). A short animation plays just before they disappear (see 7.3.8).

Various enemy types are possible. You are required to implement two specific enemy types described below: "fire-throwing skulls" and "bombers" (explosive placers)—an exciting challenge! :-)

Hints for this section:

- Using enumerated types combined with the `switch` statement is a natural way to implement actor updates (`update`) based on their states (as will be the case for most enemies).
- Ensure that each class contains only what is specific to it and that superclasses are unaware of their subclasses.
- The code for `interactWith` methods should, in most cases, be concise. If your code becomes too verbose, please consult us, as it might indicate a misunderstanding of how to implement interactions.

Since some enemies will launch projectiles (e.g., the fire-throwing skulls in Figure 8), you are first asked to model this concept.

4.2.1 Projectiles

A projectile is a mobile grid-based actor that actively requests interactions. It can fly over grid cells and can be walked on.

A projectile is characterized by its movement speed and the maximum distance it can travel from its starting point. These two characteristics are fixed at construction. It moves in a single direction (its orientation) and disappears when it completes its course (reaches its maximum distance). The speed attribute modulates movement speed (e.g., using a statement like `move(MOVE_DURATION/speed)`). The maximum distance can be implemented as an integer. At each simulation cycle, the remaining distance is decremented by one.

By default, a projectile does not accept contact or remote interactions. However, it can impose contact interactions as long as it is in motion (everything it touches along its path can be affected).

A projectile can also be stopped mid-course before reaching its maximum distance.

At this abstraction level, interactions with specific concrete actors cannot yet be defined.

As a first concrete projectile type, implement flames.

Hint: You can introduce a category `Unstoppable` to model entities that can fly over cells (including those usually unwalkable). Then, modify `ICoopBehaviour` and `ICoopCell` so that an `Unstoppable` entity can occupy any type of cell.

Flames A flame (`Fire`) is a projectile that disappears when stopped mid-course. It is visually represented by an animation (see 7.3.6).

Through contact interaction:

- It attempts to cause fire damage to all enemies and `ICoopPlayer` characters (e.g., 1 point of damage, though you are free to choose a different value). This is an attempt because damage will not be inflicted if the targeted entity is immune to fire damage.
- It triggers the explosion of explosives.

4.2.2 Fire-throwing Skulls

A fire-throwing skull (`HellSkull`) is an enemy vulnerable to physical and water damage. Its maximum health points are constant across all enemies of this type (e.g., `1` for simpler battles, though you can choose another value). They can be traversed. They impose contact interactions (if alive) but not remote ones.

The graphical representation of a `HellSkull` can be achieved with an animation (see 7.3.9).

The `HellSkull` can launch "flames" (see Figure 8).

General Behavior At random intervals `t`, the `HellSkull` generates a flame in front of it (if the space is accessible). The interval `t` is randomly regenerated after each launch, chosen between two bounds you specify (e.g., `0.5f` and `2.f`).

To generate a random float between two bounds `MIN` and `MAX`, use:

```
float randomFloat =
   RandomGenerator.getInstance().nextFloat(MIN, MAX);
```

Interactions In contact interactions, the `HellSkull` inflicts fire damage, causing the loss of a specific number of health points for the main characters. The amount of damage is constant across all `HellSkull` instances and, for simplicity, is the same regardless of the interacting entity.

For testing, ensure the `Maze` area contains 10 `HellSkull` instances oriented to the right, positioned at coordinates `(12,33)`, `(12,31)`, `(12,29)`, `(12,27)`, `(12,25)`, `(10,33)`, `(10,32)`, `(10,30)`, `(10,28)`, and `(10,26)`.

Some debugging tips (especially for the `Maze` area):

- You can zoom out by adjusting the `DEFAULT_SCALE_FACTOR` in `ICoopArea` (try a value like 39.f instead of 13.f).
- Artificially position the center of mass on a region to observe (e.g., for debugging `BombFoe`, temporarily change the "spawn" positions of the characters to `(3,10)` and `(4,10)`).
- Comment out the character death handling code to observe enemy behavior more easily ;-).

4.2.3 Task

Once the previously described concepts are implemented according to the specifications and constraints provided, launch your game `ICoop`. You will verify:

1. that the 10 fire-throwing skulls appear at the specified positions;
2. that all of them launch flames at random intervals;
3. that the flames move in front of the skulls launching them, fly over all cells (even those not walkable), animate, and disappear when they reach the edge of the area;
4. that flames launched by the top-left skull do not damage the skull aligned to its right (this skull never disappears);
5. that explosions caused by explosives do not damage the fire-throwing skulls;
6. that the fire-throwing skulls inflict fire damage to characters on direct contact;
7. that the flames also inflict fire damage to the characters;
8. and that the launched projectiles destroy everything in their path as long as their course is not complete and disappear by themselves once their course ends.

4.2.4 Bomber

The bomber (`BombFoe`, see Figure 9) is an enemy vulnerable to physical and fire damage. The maximum number of health points is the same for all its instances (e.g., 2). It is created oriented downward.

It can be in different states that determine its behavior. By default, it is in the "idle" state. In this state, the bomber does nothing except move. Otherwise, it can either be attacking

Figure 9: Bombers (explosive-placing enemies)

or protecting itself.

The graphical representation of the bomber will be achieved using specific animations: *"icoop/bombMonster"* for the general case and *"icoop/bombMonster.protecting"* when it is protecting itself.

Interactions The bomber requests remote interactions but does not engage in contact interactions. It can only interact when idle or attacking. While attacking, its interaction (perception) field is the single cell in front of it. Otherwise, its extended perception field is a constant set of cells in front of it (the same number for all bombers, e.g., 8). It is represented using an animation (see 7.3.10).

The bomber interacts with the main characters. The interaction consists of switching to "attack" mode when one of these characters is within its interaction field. For now, you do not need to address how the main characters fight their enemies.

General Behavior The bomber has idle moments during which it does absolutely nothing (not even moving), regardless of its state. This idle time is initially zero and is coded as a number of simulation steps. It cannot exceed a certain value (e.g., `24`, common to all enemies of this type).

The bomber's behavior is as follows when its idle time has elapsed, and it is not in an immunity period:

- If it is in the idle state and no movement is in progress, it moves randomly (see below) and may enter an idle moment with a duration randomly drawn between zero and the maximum idle time.
- If it is in "attack" mode, it moves toward the character it has memorized as its target until it is sufficiently close (e.g., a distance of 3[6]). More details are provided below. It then drops an explosive in the cell in front of it (if free) and switches to "protection" mode.
- In the "protection" state, it moves more slowly (you are free to implement this as you wish) and forgets its target. It remains in this state for a certain period (a random

[6]The distance can be calculated using the `distanceBetween` method from `DiscreteCoordinates`

duration between two class-specific constants, e.g., 72 and 120) before switching back to the idle state.

When in an immunity period, the bomber switches to idle mode with zero idle time.

Random Movement The random movement of the bomber is somewhat similar to that of a main character. However, this movement is not controlled via the keyboard, and the change in orientation occurs randomly. The probability of changing orientation is drawn randomly, and if it occurs, the new orientation is also chosen randomly from the four possible directions.

For example, the bomber has a 40% chance of changing orientation, and if it does, it will choose a direction randomly and equally likely among the four options. To randomly draw an integer between 0 and MAX, use the following instruction:

```
int randomInt = RandomGenerator.getInstance().nextInt(MAX);
```

To adopt the desired behavior in 40% of cases, you can randomly draw a double:

```
double randDouble = RandomGenerator.getInstance().nextDouble();
```

and apply the behavior if the number drawn is less than 0.4.

Note: The call to the `move` method can be parameterized with a value `ANIMATION_DURATION / speedFactor`. The `speedFactor` can vary depending on the bomber's state (e.g., 2 for normal animation speed, 6 for fast, and 1 for slow). Random movement steps will occur at normal animation speed.

Targeted Movement The following simple algorithm will be used to choose the direction:

i. Calculate `v`, the vector separating the bomber from its target (method `sub` from `Vector`), `deltaX`, the x-component of this vector, and `deltaY`, its y-component.
ii. If the absolute value of `deltaX` is greater than that of `deltaY` (the bomber is farther horizontally than vertically from its target), orient the bomber according to the vector `(deltaX, 0)`; otherwise, orient it according to the vector `(0, deltaY)`. The `Orientation.fromVector` method can be used to convert a `Vector` into an `Orientation`;
iii. If the orientation change was not successful, a rapid movement step will occur.

To test these developments, ensure that the `Maze` area contains 4 `BombFoe` positioned at the coordinates `(5,15)`, `(6,17)`, `(10,17)`, and `(5,14)`.

4.2.5 Task

Once the previously described concepts are implemented according to the specifications and constraints provided, launch your game `ICoop`. Verify that it behaves as shown in the following video: https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step3-3.mp4; specifically:

1. The `BombFoe` moves on its own, randomly changing direction within the same limits as the characters;

Figure 10: Attack states for the character with specific visuals: left for sword attack and right for staff attack.

2. It can randomly drop bombs when a character enters its extended perception field;
3. It enters protection mode with a specific graphic when it successfully drops a bomb;
4. It randomly observes idle moments.

4.3 Battles

It is time to better equip the characters to defend themselves against their enemies. Start by equipping them with a new weapon: a staff capable of launching water or fire orbs. Red staffs will launch fire orbs, and blue staffs will launch water orbs.

4.3.1 Staffs and Water/Fire Orbs

A staff (`Staff`) is an `ElementalItem` represented by an animation (see 7.3.7).

Water and fire orbs are coded similarly to flames. They are also drawn using animations (see 7.3.7). Fire or water orbs will cause 1 point of damage of their respective type to enemies, make bombs explode, and destroy rocks. They will stop moving upon impact.

Finally, once collected by characters, staffs must be added to the inventory.

Create a red staff at position `(13,2)` in the `Maze` area and a blue staff at position `(8,2)`.

4.3.2 Tasks

Verify that the staffs appear in the `Maze` area, can be collected, but only by characters of the corresponding color, and appear in the inventory afterward. The functionality for launching water and fire orbs will be tested later.

4.3.3 Weapon Usage by Characters

The method managing the use of equipment for `ICoopPlayer` currently only handles bombs. You are required to extend it to handle swords and staffs.

To do this, model the concept that the character can also have different states determining their behavior (e.g., if in the "attacking with a sword" state, they can attack an enemy).

Anticipated states include the "idle" state, similar to that described earlier for some enemies, and states representing the character attacking: "attacking with a sword" and "attacking with a staff."

The transition from one state to another is conditioned by the call to the method managing equipment usage (invocable via predefined controls) and, of course, by the selected equipment. For example, the character transitions to the "attacking with a sword" state when the player selects the sword as the current equipment and expresses the intention to use it by pressing the designated key (default: `E` and `O`).

Transitioning to an attack mode can only occur if the character is in an idle state (e.g., the character cannot transition directly from "attacking with a sword" to "attacking with a staff"). This transition should be accompanied by a specific visual (see figure 10 and appendix 7.3.1).

Returning to the idle state should occur after the animations related to the transition to a new state are completed and the attacks are executed:

- A sword attack ends when the user stops pressing the key corresponding to the equipment's usage. The character returns to the idle state as soon as the requested interaction is completed, including the sword attack animation time.
- A staff attack ends similarly. Before transitioning to the idle state, ensure that a water or fire orb is launched by the character, depending on their color.

You are required to modify the `ICoopPlayer` update method to integrate the transition to new states (in accordance with the descriptions above). The `ICoopPlayer` will only move as it previously did when in the idle state.

You will also modify the code so that:

- The character can have a remote interaction as long as they are using their sword (i.e., while in the "attacking with a sword" state, remote interaction is possible).
- During a sword attack, the character can deal physical damage to enemies.
- The character can launch water/fire balls while attacking with a staff.
- Fire/water balls deal fire/water damage to enemies and destroy rocks.

4.3.4 Tasks

Once the previously described concepts are implemented according to the specifications and constraints provided, launch your game `ICoop`. Verify:

1. That the main characters can still move under keyboard control when idle.
2. That they can transition to different attack modes upon request after selecting the appropriate equipment, whether staff or sword.
3. That the chosen equipment can be used via the anticipated keys, and this action is reflected visually.
4. That they can launch water projectiles using the staff and that these projectiles stop in their tracks when encountering enemies or rocks.
5. That they can destroy enemies by dealing physical damage (e.g., striking them with the sword).

6. That they can deal physical damage to fire skulls and bombers (using the sword).
7. That they can deal water damage (using the staff) to enemies.
8. And that enemies disappear in a small puff of smoke upon dying.

4.4 Validation of Step 3

To validate this step, all the verifications from sections 4.1.5, 4.2.3, 4.2.5, 4.3.2, and 4.3.4 must be completed.

The game `ICoop` with the behavior described above must be submitted at the end of the project.

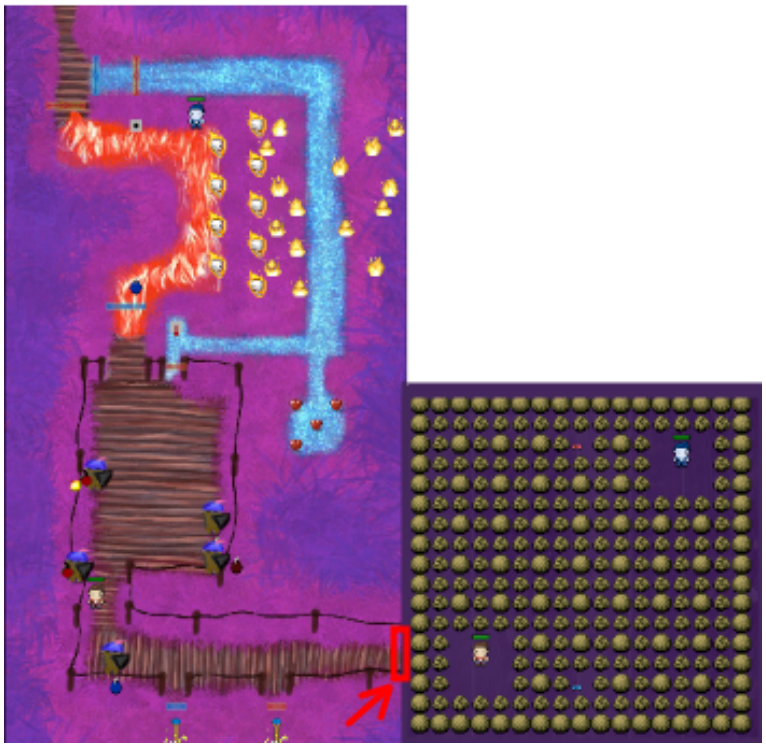

Figure 11: Passageway from `Maze` (left) to `Arena` (right): the `Arena` area does not allow you to go back to `Maze` and is strewn with obstacles that cannot be crossed and rocks that can be destroyed

.

5 Final challenge (Step 4)

This last step is much more open. You will only be given a specification of the features to be implemented and a few guidelines.

You have to implement the fact that when the characters leave the `Maze` area through the door in figure 11, they find themselves trapped in an area without a door, the `Arena` area. Here they must destroy rocks to collect the keys of their colour. Once each character has collected the key of their colour (elemental category), a teleporter will appear which will take them back to the initial room, `Spawn`.

If they have collected the sticks of their own colour in the previous area, they will be able to 'open' the door to the mansion and earn the final reward.

The implementation of this step must be based on the notion of a logical signal (`Logic`): for example, the collection of the two keys can be modelled as a logical 'and' object (`AND` in the `logic` package of `game-engine`) on the two keys which are themselves `Logic` objects.

5.1 Keys and Teleporters

A *key* is an `ElementalItem` that is drawn using the sprite:

```
new Sprite‘(icoop/’key_red, 0.6f, 0.6f, this);
```

if it serves the fire element or

```
new Sprite‘(icoop/’key_blue, 0.6f, 0.6f, this);
```

if it serves the water element.

A teleporter is nothing other than a drawable door which only occupies the position of its main cell (as returned by `getCurrentMainCellCoordinates()`). It is only drawn if it is active. As a ‘sprite’ for drawing it, you can use:

```
RPGSprite‘(’shadow, 1, 1, this, new RegionOfInterest(0, 0, 32,
   32))
```

5.2 Area `Arena`

The `Arena` area is accessible from `Maze` via a door with the following specification:

destination	arrival coord.	Signal	main cell	other coord.
”Arena”	(4,5),(14,15)	Logic.TRUE	(19,6)	(19,7)

At its creation, it contains a red key in position `(9,16)`, a blue key in position `(9,4)` and a gate (closed and therefore invisible) in position `(10,10)`. The portal will provide the transition to the `Spawn` area (the characters will arrive at positions of your choice).

The `Arena` area is littered with rocks and obstacles, and it would be tedious to record them all manually when creating the area. You are therefore asked to modify the code so that the grid can dictate to the area the creation of an actor `Rock` wherever there is a cell of type `ROCK` and the creation of an actor `Obstacle` wherever there is a cell of type `OBSTACLE`.

Be sure to code this without an intrusive getter!

5.3 Gateway to the manor

We suggest that you finalise the mandatory part of the project by associating the location of the manor door in the `Spawn` area with a new type of door that causes a dialogue to open when the characters try to interact with it by touching it. The dialogue will be *`"key_required"`* if the door is closed and *`"victory"`* if it is open. Figure 12 shows the first type of dialogue (corresponding to a door which is still closed).

The door will occupy the position `(6,11)` to coincide with the graphics. As you will stop at this point for the development of the game itself, the door will not have to allow characters

Figure 12: Gateway to the manor: the objective of the game, in its mandatory part, is to succeed in opening it!

to transit to a new area (which you can correct in the extensions if you wish). You can therefore choose the room `Spawn` itself as the destination room.

The characters will have succeeded in their mission when they are able to open the door. To do this, you'll need to extend your code to model the fact that:

- areas have a specific challenge to complete. For the area `Maze` the challenge is to collect the two sticks. For the `Arena` area, the challenge is to collect the two keys and for the `Spawn` area, the challenge is to open the manor door. For the `OrbWay` area, it's up to you (and an area can always be considered to have a successful challenge).
- The behaviour of players in an area can be conditioned by external signals. For example, the opening of the manor's door depends on the successful completion of the challenges in the `Arena` and `Maze` areas.

Indication: the areas can behave like signals (`Logic`)!

Be sure to code this:

- without the areas having to know about each other (there is no reason for one area to be aware of the existence of another area in the game);
- and without the characters having to memorize information that is too specific (as an example, the knowledge of whether a stick or key has been collected); it should be possible to change the nature of an area's challenge without this having any impact on the characters' code.

5.4 Task

You are asked to code the concepts described above in accordance with the specifications and constraints given. You will then check:

1. that the door to the manor is initially closed in the `Spawn` area; the dialogue indicating that the door is closed must be displayed when the characters try to go through it;
2. that the new door in the `Maze` area should be able to lead to the `Arena` area;

3. that once through, both characters are initially trapped there;
4. that the area `Arena` appears as in figure 11;
5. that the characters can destroy the rocks (using the stick) but not the obstacles;
6. that each of them can collect the key of their colour once they have accessed it;
7. that the collection of the two keys causes a portal to appear and that as soon as one of the characters passes through it (by contact), both characters are released from the arena and teleported to the area `Spawn`;
8. and that the door to the mansion is opened there if the characters have also collected the sticks in `Maze`.

There's a slight difficulty to be expected here concerning the dialogue associated with the manor door: the dialogue appears during a contact interaction and it's not possible to move the character until the dialogue has fully unfolded. However, when the dialogue ends, the character will still be interacting with the door and the dialogue will start all over again.

Question 4

How do you propose to solve this little problem without introducing specific tests into the game on the nature of the current dialogue?

5.5 Validation of Step 4

To validate this step, all the checks in section 5.4 must have been carried out.

The game `ICoop` whose behaviour is described above should be submitted at the end of the project.

6 Extensions (Step 5)

To obtain bonus points to compensate for any shortcomings in the mandatory part of the project, or to take part in the competition, you can implement a few freely chosen extensions. A maximum of 15 points will be awarded (coding a lot of extensions to compensate for the weaknesses of the previous parts is therefore not an option).

Implementation is open and there is very little guidance. Only a few suggestions and indications are given below. Note that from a graphics point of view, two additional rooms are provided to allow you to code extensions (`SanctumEntrance` and `Sanctum`). An estimate of the points awarded for the suggested extensions is given, but don't hesitate to ask us for a more precise evaluation if you have a particular idea. A small bonus will be awarded for displays of creative game design.

You can code your extensions in the existing game, `ICoop`, but it is then **important to preserve the obligatory functionalities and the testability of the requested components**.

Alternatively, you can create a new game `ICoopExtension` using the logistics you set up in the previous steps.

You should take care to **carefully comment** your `HELP` file, so that the way your extensions can be played is clear. In particular, we need to know which controls to use and which effects they have, without having to read your code.

Here's an example of a game you could play and a corresponding (partial) `HELP` that explains how to play:

- video example of the game: ICoop.mov

A draft `HELP.md` file corresponding to a game from previous years is given as an example: HELP.md

> You are expected to choose a few extensions and code them all the way through (or nearly all the way through). The idea is not to start coding lots of little bits of disparate and unfinished extensions in order to collect the necessary points ;-).

Note that the material supplied includes a few resources for additional areas.

6.1 New actors or character extensions

All sorts of actors can be considered. In particular, the 'signal' component can be used to create game scenarios based on solving puzzles of varying complexity. A (non-exhaustive) list of suggestions is given below.

- model a resource system (gold, silver, wood, food, doses of health etc); (~4 to 6 points)
- add an object `Box` or `Safe`, containing one or more objects and whose opening would be directed by a signal; (~3pts)
- various actors who can be used as signals (orbs, torches, pressure plates, levers); (~4pts)

- animated decorative actors (~2pts)
- advanced puzzle signals (oscillators, signals with delays): an oscillator is a signal whose intensity varies over time; (~4pts/signal)
- all kinds of characters with specific movement types and behaviors; they can be hostile or friendly towards the player; (~3pts to 6pts depending on the complexity of the character)
- in particular, vendor characters who have an inventory and from whom the main characters can buy equipment after collecting coins; the notion of inventory will have to be enriched to allow items to pass from one inventory to another; this will involve coding a graphical menu allowing inventories to be displayed in their entirety (with the number of items of each type and selecting an item, for example the one you want to buy) (~5 to 10 pts): Note that the menu associated with an inventory can be coded even without a vendor character.
- create new modes of movement for the characters (running, swimming, etc.) with an appropriate adaptation of sprites/animations and scenery elements ; (~3 to 5 pts)
- create follower characters like Red's Pikachu in Pokémon Yellow; (~3pts)
- create one or more scenario events triggered by signals. For example, a character arriving in the area to hand over an object or give an instruction (~3 to 5 points)
- add new types of cells with appropriate behaviours (water, ice, fire, etc.); (2pts/cell)
- implement a day/night cycle that could serve as a signal or condition the characters' behaviour (e.g. it can't move if it's too dark and should carry a torch); (5pts)
- add a shadow or reflection to the player and certain actors; (~2 to 3pts)
- add new advanced controls (interactions, actions, movement, etc); (~2pts/control)
- add random events (scenery, signals, etc.); (~4pts)
- add multiple choice dialogues; (~3 to 6 pts)
- etc.

6.2 Pause, game ending and 'reset'(~2 to 5pts)

The notion of area can be used to introduce the pausing of games. At the player's request, the game can switch to pause mode and then back to game mode. You can also introduce end-of-game management (if the characters have reached an objective or have been beaten, for example).

Resetting an area, as currently coded, does not allow you to return to the area in exactly the same state as when you first entered it, particularly when it comes to inventory management. It is possible, as an extension, to keep track of this state in order to restore it appropriately.

In fact, the database you have coded can be enriched as you wish. You can also let your imagination run wild, and try out your own ideas.

If you come up with an original idea that you think differs in spirit from what is suggested and you want to implement it for the submission or the competition (see below), you need to have it validated before continuing (by sending an email to **cs107@epfl.ch**).

Appendix 3 of the tutorial gives you guidelines for enriching the graphical assets.

Be careful, though, to not spend too much time on the project to the detriment of other classes!

6.3 Validation of Step 5

As the final result of the project, create a game scenario that is well documented in the `HELP` and involves all the coded components. A (small) part of the mark will be linked to the inventiveness and originality you demonstrate in designing the game.

6.4 Contest

People who have completed the project with a particular effort on the final result (interesting gameplay, content of game areas, visual effects, interesting/original extensions, etc.) can compete for the 'best game of CS107' prize.[7]

If you'd like to enter, you'll need to send us a short 'application file' by **19.12 at 13:00** by e-mail to **cs107@epfl.ch**. This should include a description of your game and the add-ons you have incorporated (2 to 3 pages in .pdf format with a few screenshots highlighting your additions).

The winning projects will be exhibited during the first week of the coming semester (February).

[7] We have planned a small 'Wall of Fame' on the course web page and a small symbolic reward :-)

7 Appendices

7.1 KeyBindings

A `KeyBindings` class is provided to facilitate the configuration of the keys used in the project. It is implemented using the notion of `record`, which will only be covered in the second semester. A `record` is essentially a shorthand way of writing a restricted form of class. For this project, it is sufficient to know that :

- the keys for controls are defined in the variables `RED_PLAYER_KEY_BINDINGS` and `BLUE_PLAYER_KEY_BINDINGS` (and can be freely modified according to your preferences);
- the `PlayerKeyBindings` type ensures that **in order**, the keys correspond to the controls: "move up," "left," "down," "right," "change equipment" (select current equipment in the inventory), and "use the current equipment";
- to assign a specific set of keys to an `ICoopPlayer`, simply provide it with an attribute of type `KeyBindings.PlayerKeyBindings`;
- when `keys` is an object of type `KeyBindings.PlayerKeyBindings`, simply write `keys.useItem()` (and similarly, `keys.right()`, `keys.down()`, etc.) to reference a specific key, such as the one associated with equipment use. The `moveIfPressed` method of characters can invoke:

  ```
  moveIfPressed(Orientation.LEFT,keyboard.get(keys.left()));
  ```

  or during a character's `update`, you can check if the equipment-use key is pressed with:

  ```
  keyboard.get(keys.useItem()).isPressed()
  ```

Note: The default configuration is for a QWERTZ keyboard. For an AZERTY keyboard it would be:

```
RED_PLAYER_KEY_BINDINGS = new PlayerKeyBindings(Z, Q, S, D, A,
   E);
```

in `KeyBindings.java`.

7.2 Basic Configuration of the Maze Area

According to Figure 13, the suggested basic configuration for the Maze area is as follows :

- Starting coordinates of the characters in the area: `(2, 39)`, `(3, 39)`.
- Two left-oriented water walls at positions `(4, 35)` and `(4, 36)`, always active.
- Two left-oriented fire walls at positions `(6, 35)` and `(6, 36)`, deactivated when pressure plate 4 is active (when a character steps on it).
- Two downward-oriented fire walls at positions `(2, 34)` and `(3, 34)`, always active.
- Pressure plate at position `(6,33)`.
- Explosive at position `(6, 25)`.
- Two downward-oriented water walls at positions `(5,24)` and `(6,24)`, always active.
- Pressure plate at position `(9,25)`. As a bonus, you can replace this actor with a lever (see extensions).

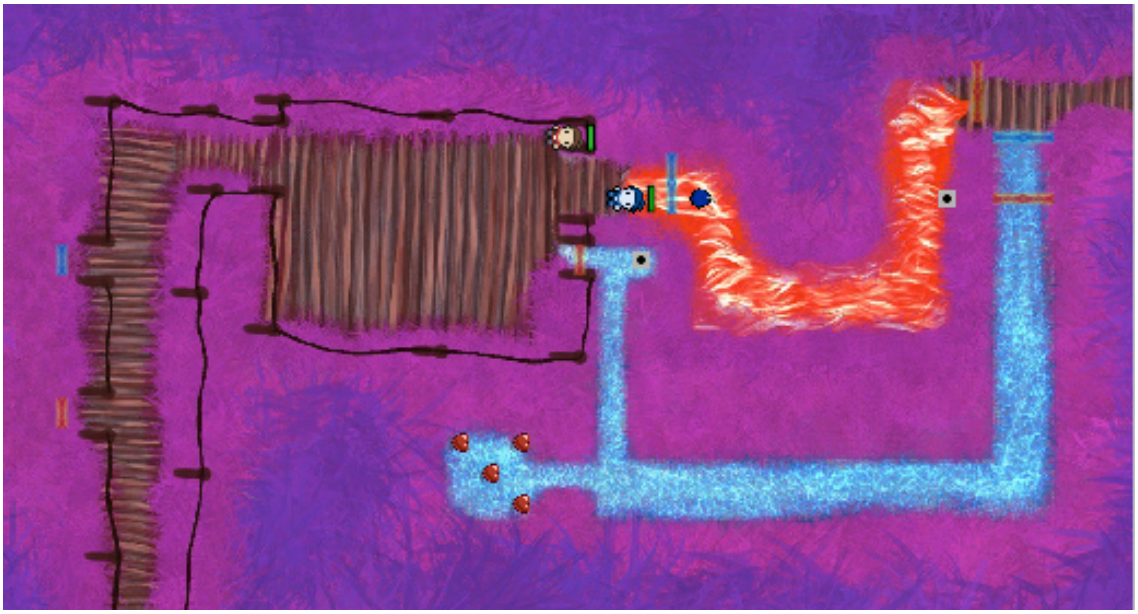

Figure 13: Basic configuration of the `Maze` area (side view)

- Downward-oriented fire wall at position `(8,21)` that disappears if the previous component is active.
- A series of hearts at positions `(15, 18)`, `(16, 19)`, `(14, 19)`, and `(14, 17)`.
- Downward-oriented water wall at position `(8,4)`, always active.
- Downward-oriented fire wall at position `(13,4)`, always active.

7.3 Creating Animations

The animation code is not particularly interesting to write. Therefore, an example is provided for each class.

7.3.1 `ICoopPlayer`

Basic Animation:

```
final Vector anchor = new Vector(0, 0);
final Orientation[] orders = {DOWN, RIGHT, UP, LEFT};
 ... new OrientedAnimation(prefix, ANIMATION_DURATION, this,
                           anchor, orders, 4, 1, 2, 16, 32,
                           true)
```

where `ANIMATION_DURATION` equals 4, and `prefix` equals *`"icoop/player"`* for the red player and *`"icoop/player2"`* for the blue.

Sword Use:

```
final Vector anchor = new Vector(-.5f, 0);
final Orientation[] orders = {DOWN, UP, RIGHT, LEFT};
```

```
... new OrientedAnimation(prefix+".sword",
                          SWORD_ANIMATION_DURATION, this,
                          anchor, orders, 4, 2, 2, 32, 32)
```

where `SWORD_ANIMATION_DURATION` equals 2.

Staff Use:

```
final Vector anchor = new Vector(-.5f, -.20f);
final Orientation[] orders = {DOWN, UP, RIGHT, LEFT};
... new OrientedAnimation(name , STAFF_ANIMATION_DURATION, this,
                          anchor, orders, 4, 2, 2, 32, 32)
```

where `STAFF_ANIMATION_DURATION` equals 2, and `name` equals `icoop/player2.staff_water` for the blue staff and `icoop/player.staff_fire` for the red.

7.3.2 Explosive

Unexploded explosive:

```
new Animation("icoop/explosive", 2, 1, 1, this, 16, 16,
              ANIMATION_DURATION/2, true)
```

Explosion:

```
new Animation("icoop/explosion", 7, 1, 1, this, 32, 32,
              ANIMATION_DURATION/7, false)
```

where `ANIMATION_DURATION` equals 24.

7.3.3 Orbs

```
final Sprite[] sprites = new Sprite[ANIMATION_FRAMES];
for (int i = 0; i < ANIMATION_FRAMES; i++) {
  sprites[i] = new RPGSprite("icoop/orb", 1, 1, this,
  new RegionOfInterest(i * 32, spriteYDelta, 32, 32));
}
.. new Animation(ANIMATION_DURATION / ANIMATION_FRAMES, sprites)
```

where `ANIMATION_DURATION` equals 24, `ANIMATION_FRAMES` equals 6, and `spriteYDelta` equals zero for blue orbs and 64 for red ones.

7.3.4 Hearts

```
new Animation("icoop/heart", 4, 1, 1, this, 16, 16,
              ANIMATION_DURATION/4, true)
```

where `ANIMATION_DURATION` equals 24.

7.3.5 Staves

```
new Animation(staff_name , 8, 2, 2, this, 32, 32,
              new Vector(-0.5f, 0),
              ANIMATION_DURATION/8, true);
```

where `ANIMATION_DURATION` equals 32, and `staff_name` equals *"icoop/staff_water"* for the blue staff and *"icoop/staff_fire"* for the red staff.

7.3.6 Flames

```
new Animation("icoop/fire", 7, 1, 1, this, 16, 16, 4, true)
```

7.3.7 Water or Fire Projectiles

```
new Animation(name, 4, 1, 1, this, 32, 32,
              ANIMATION_DURATION/4, true)
```

where `ANIMATION_DURATION` equals 12, and `name` equals *"icoop/magicWaterProjectile"* for water projectiles and *"icoop/magicFireProjectile"* for fire projectiles.

7.3.8 Death of foes

Foes are displayed as a small "cloud" when they die:

```
new Animation("icoop/vanish", 7, 2, 2, this, 32, 32, new
   Vector(-0.5f, 0f), ANIMATION_DURATION/7, false);
```

where `ANIMATION_DURATION` equals 24.

7.3.9 Fire-Shooting Skulls

```
 Orientation[] orders = new Orientation[]{Orientation.UP,
    Orientation.LEFT, Orientation.DOWN, Orientation.RIGHT};
... new OrientedAnimation("icoop/flameskull",
                          ANIMATION_DURATION/3, this,
                          new Vector(-0.5f, -0.5f), orders,
                          3, 2, 2, 32, 32, true)
```

where `ANIMATION_DURATION` equals 12.

7.3.10 Artificers

```
final Vector anchor = new Vector(-0.5f, 0);
final Orientation[] orders = {DOWN, RIGHT, UP, LEFT};
// "unprotected" state:
...new OrientedAnimation("icoop/bombFoe", ANIMATION_DURATION/3,
                         this, anchor, orders, 4, 2, 2, 32, 32,
                         true)
// "protected" state:
...new OrientedAnimation("icoop/bombFoe.protecting",
   ANIMATION_DURATION/3,
```

```
                                this, anchor, orders, 4, 2, 2, 32, 32,
                                false)
```

where `ANIMATION_DURATION` equals 24.