CS-107: Mini-project 2
Maze Games

M. Alaoui, J. Sam

Version 1.4

Contents

1 Overview

> This document uses colors and contains clickable links. It is best viewed digitally.

Over the past weeks you have become familiar with the fundamentals of a small ad hoc game engine (see tutorial) that lets you create two-dimensional tile-based, RPG-style games. The goal of this mini-project is to leverage it to build one or several concrete variants of a "maze game" named `ICMaze`. The main character explores a procedurally generated labyrinth that becomes more complex from level to level and is populated with helpful or hostile entities. Figure 1 shows fragments of the base prototype that you can later enrich as you wish. Section 6 also links to a short example video.

Beyond the playful aspect, this mini-project is an opportunity to practise fundamental object-oriented concepts in a natural context. You will see how designing at the right level of abstraction yields programs that are easy to extend and adapt. Step by step you will expand the set of features and the interactions between components.

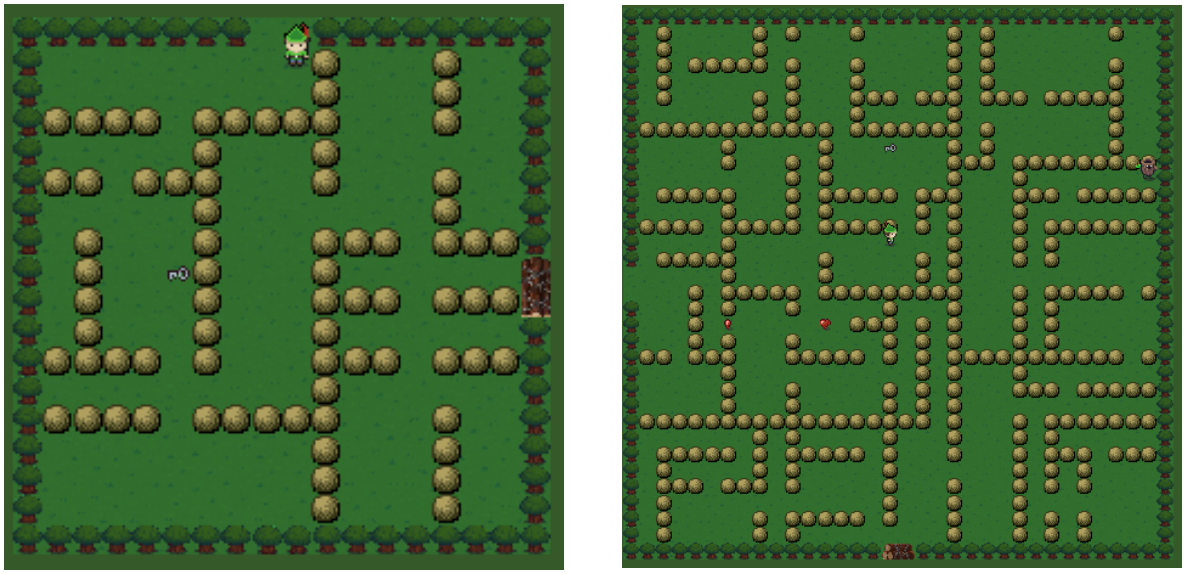

Figure 1: Example scenes where the hero solves increasingly complex labyrinths and meets more or less friendly characters.

The project contains four **mandatory** steps and one optional step:

- Step 1 ("Core game"): by the end of this step you will have built, using the provided engine tools, a basic instance of `ICMaze` where the hero can collect items and cross portals.
- Step 2 ("Maze generation"): this step adds a mechanism that generates mazes constraining the hero's movements.
- Step 3 ("Levels and enemies"): you will procedurally generate levels populated with enemies that the hero must defeat to move through the mazes.

- Step 4 (“Final challenge”): the mandatory scope concludes with a new enemy, gameplay based on logical signals, and simple dialogues.
- Step 5 (Optional extensions): during this step we suggest more open-ended extensions so that you can enrich the previous game or create new ones.

> Coding a few optional extensions (your choice) earns bonus points and/or showcases your project for the contest.

Here are the main guidelines to follow when coding the project:

1. **Do not modify the `game-engine` code**. Be wary of IntelliJ suggestions that attempt to change it.
2. Use only the standard Java libraries (imports starting with `java.` or `javax.`). If you are unsure about a library, ask us and be careful with alternatives proposed by IntelliJ. The project notably relies on `java.awt.Color`.
3. Document your methods following the javadoc conventions (see class `TextGraphics` in `game-engine`).
4. Respect standard naming conventions and keep your code well **modularised and encapsulated**. In particular, avoid intrusive public getters that expose mutable objects.
5. The instructions can be very detailed but **they are not exhaustive**. You will have to introduce additional methods and attributes when needed while preserving encapsulation.
6. Your project **must not be stored on a public repository** (such as GitHub). We recommend GitLab https://gitlab.epfl.ch/ if you are familiar with git, but any private repository is fine.

Step 1 is deliberately guided. Its purpose is to complete your understanding of the provided prototype and start using it in practice.

2 Core ICMaze (step 1)

The goal of this step is to start building your own `ICMaze` game.This base version features a main character who can move between two areas and collect items. These features rely on the general mechanism of *interactions between actors* described in tutorial 3.

This game therefore involves:

- a main character;
- *collectable items* that can be picked up by walking over them ("contact interactions");
- *portals* that let the character travel from one area to another.

You will work inside the provided `icmaze` package.

2.1 Preparing the `ICMaze` game

> The tutorial solution is available in folder `tutos`; feel free to use it as inspiration to get started with your implementation.

Prepare an `ICMaze` game inspired by `Tuto2`. It initially consists of:

- class `ICMazePlayer`, modelling the main character, placed in `icmaze.actor`; leave this class empty for now, we will revisit it later;
- class `ICMaze`, equivalent to `Tuto2`, placed in package `icmaze`. Remember to adapt method `getTitle()` so that it returns a name of your choice (e.g., *`"ICMaze"`*);
- class `ICMazeArea`, equivalent to `Tuto2Area`, located in sub-package `icmaze.area`;
- classes `Spawn` and `BossArea` extending `ICMazeArea`, placed in `icmaze.area.maps` (they play the roles of `Ferme` and `Village` in the tutorial). Use titles such as *`"icmaze/Spawn"`* and *`"icmaze/Boss"`*;
- class `ICMazeBehavior`, analogous to `Tuto2Behavior`, stored in `icmaze.area` and containing public class `ICMazeCell`, the counterpart of `Tuto2Cell`.

Note: the `ICMazeBehavior` resource must be named *`"SmallArea"`* for both `Spawn` and `BossArea` so that the graphics line up.

A few adjustments are nevertheless required to better fit the new game. Take the time to apply them carefully.

2.2 Adapting `ICMazeArea`

Both concrete areas now share the same grid name (*`"SmallArea"`*). Unlike the tutorial, there is no longer a one-to-one mapping between the area title and the behavior name. `ICMazeArea` must therefore also store the name of its grid, and its constructor must initialise it. Method `begin` must then use this name when associating a grid with the area (instead of relying on `getTitle`).

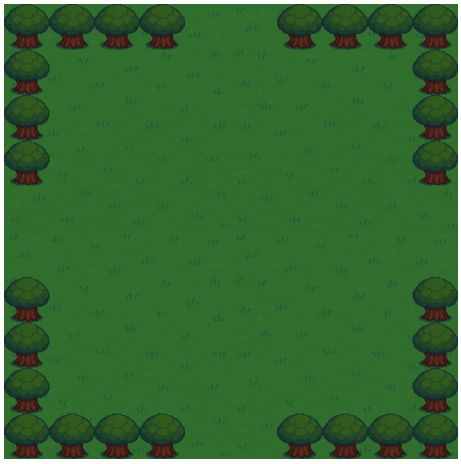

Figure 2: Starting area

Note: this has an impact on the creation of `Background` type actors, which must be constructed using the constructor that takes a title as a parameter (in this case, the title of the grid).

The camera scale factor can be set to `11`.

2.3 Adapting `ICMazeBehavior`

We now add a few features to `ICMazeBehavior` and `ICMazeCell`.

The enumeration describing the cell types and their traversability is defined as follows:

```
NONE(0,false),      // Should never be used except in
   toType
GROUND(-16777216, true),
WALL(-14112955, false),
HOLE(-65536, true);
```

From now on the "decor" is not the only element affecting the hero's movement. The presence of another actor that does not let itself be "stepped on" must also block the hero. An object is traversable if its method `takeCellSpace()` returns `false`.

Concretely, two entities whose `takeCellSpace()` returns `true` cannot occupy the same cell. Ensure that `ICMazeCell.canEnter()` enforces this rule.

Apply the adaptations mentioned above to `ICMazeBehavior`.

2.3.1 Task

Implement the concepts described above exactly as specified. Launch your `ICMaze` game and verify that the empty area from Figure 2 is displayed.

2.4 Actors in ICMaze

With these fundamentals in place, begin modelling the actors used in ICMaze games. Store them in sub-package icmaze.actor.

All actors in ICMaze (the ICMazeActor) move on a grid (MoveableAreaEntity). At this level of abstraction they do not have specific behaviours. Their area, orientation, and starting position are given at construction time. They occupy the same set of cells as GhostPlayer does (see method getCurrentCells()), but by default they are traversable (takeCellSpace() returns false).

Functionally, an ICMazeActor can enter an area at a given position and leave its current area. As an Interactable it only accepts contact interactions at this abstraction level.

For now we only introduce a single, more specific category of ICMazeActor: the *main character*, controlled via the keyboard.

2.4.1 Main character: ICMazePlayer

An ICMazePlayer is a non-traversable actor. It updates just like any ICMazeActor but must also react to the arrow keys, much like the GhostPlayer from the tutorial.

An ICMazePlayer can be in multiple states, represented with an enumeration. At this stage we consider state IDLE (standing still) and state INTERACTING (requesting a distance interaction). Ensure that the character only reacts to movement keys while in IDLE.

Note: We will revisit this later, but it is a good idea to use a switch on the character's state inside ICMazePlayer.update.

2.4.2 Key bindings

In the tutorial prototype the key bindings were hard-coded. It is better to make them configurable, and class KeyBindings is provided in package icmaze for that purpose. See appendix 7.1 for details. Update your movement methods so that the arrow keys used in the tutorial are replaced with PLAYER_KEY_BINDINGS.up(), PLAYER_KEY_BINDINGS.down(), PLAYER_KEY_BINDINGS.right(), and PLAYER_KEY_BINDINGS.left().

2.4.3 Drawing an ICMazePlayer

Unlike GhostPlayer, the hero's sprite is neither fixed nor static: it depends on the *orientation* and is *animated*.

Use an OrientedAnimation to draw the character. Only the base animation is needed for now. Build it as explained in appendix 7.3.1[1].

While updating the character, if the state is IDLE you must also update the current animation: if a displacement is in progress, call update on the animation; otherwise call reset.

[1] See the OrientedAnimation documentation if you want to understand the parameters.

To test these developments, spawn an `ICMazePlayer` at launch at its designated position in `Spawn`; for instance place it at `(5,7)`, facing downwards and in state `IDLE`.

2.4.4 Task

Implement the concepts described above.

Launch your `ICMaze` game. Confirm that it behaves like the short video https://proginsc.epfl.ch/wwwhive projet2/videos/icmaze/startStep1.mp4; specifically:

1. the game starts by displaying an `ICMazePlayer` facing forward;
2. the character can freely move within the area using the movement keys (arrow keys by default) and cannot leave the map;
3. the sprite changes to match the character's orientation;
4. it cannot walk on the border walls;
5. its movements are animated.

2.5 Collectable items: contact interactions

We now model objects the hero can collect during the adventure. Implement collectables in sub-package `icmaze.actor.collectable`. Use class `CollectableAreaEntity` from the template (package `areagame.actor`) to create a hierarchy of collectables specific to `ICMaze`. By default these objects are traversable, oriented downwards, and only accept contact interactions[2]. They must disappear from the simulation once collected. A subset of these objects will serve as equipment. All equipment objects have a specific orientation provided at construction time and are drawn with a `Sprite`.

Two concrete collectables are required at this stage: pickaxes (`Pickaxe`) and hearts (`Heart`). Pickaxes belong to the equipment category; hearts do not. For both types, the owning area and starting position are constructor parameters. Graphically, build the `Sprite` for the pickaxe as follows:

```
new Sprite("icmaze/pickaxe", .75f, .75f, this));
```

Hearts are animated (see 7.3.2).

Finally, update `Spawn` so that it registers at creation:

- a pickaxe at `(5,4)` facing downwards;
- a heart at `(4,5)`.

2.5.1 Task

Implement the concepts described above exactly as specified. Launch `ICMaze` and verify that the pickaxe and heart appear (Figure 3).

[2]Remember methods `isCellInteractable` and `isViewInteractable`

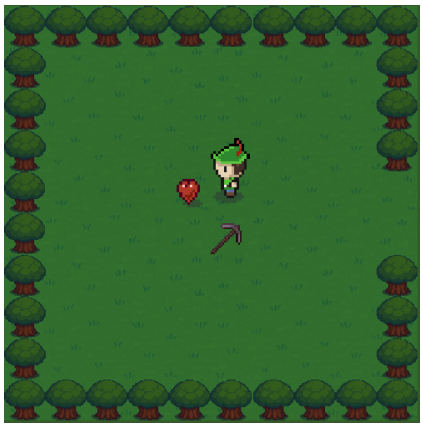

Figure 3: Initial positions of `Pickaxe` and `Heart` in the starting area.

2.5.2 Contact interactions

Apply the scheme laid out in the tutorial[3] to implement the first interactions between the `ICMazeActor` coded so far. First, make every `ICMazePlayer` an `Interactor`: it can trigger interactions on `Interactable` objects (for example to pick them up).

As an `Interactor`, `ICMazePlayer` must implement:

- `getCurrentCells`: the set of cells it currently occupies (here it boils down to its main cell, as you already implemented earlier);
- `getFieldOfViewCells()`: the cells within its field of view consist of the single cell it is facing:

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

As an `Interactor`, the player systematically wants all contact interactions. The "distance interactions" are, on their side, only wanted when the player is in a specific state:

- `wantsCellInteraction`: unconditionally returns `true`;
- `wantsViewInteraction`: returns `true` when the player is in the state `INTERACTING`, and `false` otherwise. The procedures for switching to the state `INTERACTING` will be explained later.

Let's now look at the concrete handling of interactions. In the `icmaze.handler` subpackage, you are asked to complete the `ICMazeInteractionVisitor` interface inheriting from `AreaInteractionVisitor`. This interface must provide a default definition of the interaction methods of any `Interactor` with the *Interactables* known at this stage; namely:

- a cell (`ICMazeCell`);
- the main character (`ICMazePlayer`);
- a pickaxe;
- a heart.

By default these methods can have empty bodies (meaning "do nothing"). As an `Interactor` in `ICMaze`, `ICMazePlayer` must override the relevant methods when it needs specific be-

[3]Video supplement: https://proginsc.epfl.ch/wwwhiver/mini-projet2/mp2-interactions.mp4.

haviour.

Every concrete `Interactable` must now indicate that it accepts handlers of type `ICMazeInteractionVisitor`. Rewrite their `acceptInteraction` method accordingly:

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

To allow `ICMazePlayer` to process the interactions it cares about, define inside class `ICMazePlayer` a private nested class `ICMazePlayerInteractionHandler` implementing `ICMazeInteractionVisitor`. Add the necessary methods to handle interactions with a pickaxe and with a heart.

Both objects must be collected via contact interactions. Distance interactions will be tested later.

Note: these methods should only contain two or three lines of code.

As explained in tutorial 3, this requires that:

- `ICMazePlayer` stores its interaction handler (type `ICMazePlayerInteractionHandler`);
- method `void interactWith(Interactable other, boolean isCellInteraction)` delegates to the handler:

  ```
  other.acceptInteraction(handler, isCellInteraction);
  ```

  Remember that `void interactWith` is automatically invoked by the engine for every `Interactor` and every `Interactable` it touches or that lies within its field of view (see section 6.3.1 of the tutorial).

2.5.3 Task

Implement the concepts described above exactly as specified.

Launch `ICMaze`. Check that:

1. `ICMazePlayer` behaves as before but can pick up the pickaxe and the heart by walking over them;
2. the collected items disappear from the area.

2.6 Portals and keys: distance interactions

"Portals" connect areas to each other and may require a key. When the hero stands in front of a portal, they can press the interaction key (`E` by default): if they own the required key, the portal opens.

Portals can be open, locked, or invisible. It is natural to model them as actors (`AreaEntity`) because they are not just part of the scenery: as passage points they can change behaviour, e.g., switch between open and locked based on conditions.

A locked portal visually appears as a pile of logs tied with a chain. The chain can be unlocked with a key, thus opening the portal. We therefore extend the equipment hierarchy.

2.6.1 Keys

Keys (class `Key`) are identical to `Pickaxe` except that a `Key` stores an integer identifier (initialised via the constructor). `ICMazePlayer` picks them up by walking over them and must remember the collected identifiers in order to open portals. For testing, create two keys in `Spawn`: one with identifier `Integer.MAX_VALUE` at position `(6,5)` and one with identifier `Integer.MAX_VALUE-1` at `(1,2)`.

2.6.2 Portals

Introduce class `Portal` (package `icmaze.actor`).

A `Portal` is characterised (at least) by:

- a state (an enumeration with values `OPEN`, `LOCKED`, and `INVISIBLE` works well);
- the destination area name (`String`; recall that each area exposes a title via `getTitle()`);
- the arrival coordinates inside the destination area (`DiscreteCoordinates` of the cell reached);
- the identifier of the key that opens it (an `int`). By default no key is needed (use a constant `NO_KEY_ID` equal, say, to `Integer.MIN_VALUE`).

The area, position, and orientation of a portal are constructor parameters, and **a portal is invisible by default when created**.

The drawing depends on the state: display a dedicated `Sprite` when it is invisible or locked, and nothing otherwise. Use the following snippets to initialise the sprites:

```
// invisible:
 new Sprite("icmaze/invisibleDoor_"+orientation.ordinal(),
 (orientation.ordinal()+1)%2+1, orientation.ordinal()%2+1, this);

// locked
 new Sprite("icmaze/chained_wood_"+orientation.ordinal(),
                (orientation.ordinal()+1)%2+1, orientation.ordinal()%2+1,
                   this);
```

As an `Interactable`, a `Portal` is traversable when open. It always accepts distance interactions. For simplicity, use the following body for `getCurrentCells`:

```
DiscreteCoordinates coord = getCurrentMainCellCoordinates();
return List.of(coord, coord.jump(new
    Vector((getOrientation().ordinal()+1)%2,
    getOrientation().ordinal()%2)));
```

This models the fact that the portal occupies its main cell and the cell immediately to its right.

2.6.3 Areas with portals

Class `ICMazeArea`, introduced earlier, represents the abstract concept of an "ICMaze area". Extend it by modelling *a set of portals* that enable transitions to other areas.

For simplicity, assume that:

Figure 4: Each `ICMazeArea` contains four portals at fixed positions: West (W), South (S), East (E), and North (N). In this example the west and south portals are locked, the north portal is invisible, and the east portal is open.

- every `ICMazeArea` always contains four portals at the precise positions illustrated in Figure 4;
- each area is a square of size `size x size`, which lets you automatically place the portals at `(size / 2, size + 1)` (north), `(size / 2, 0)` (south), `(0, size / 2)` (west), and `(size + 1, size / 2)` (east);
- the arrival coordinates are respectively `(size / 2 + 1, size)` (when coming from north), `(size / 2 + 1 , 1)` (from south), `(1, size / 2 + 1)` (from west), and `(size, size / 2 + 1)` (from east).

With this modeling, the destination coordinates of the "East" portal, for example, will be the arrival coordinates from the "West." Recall that the portals are invisible by default.

Introduce enumeration `AreaPortals` inside `ICMazeArea` as follows:

```
public enum AreaPortals {
      N(Orientation.UP),
      W(Orientation.LEFT),
      S(Orientation.DOWN),
      E(Orientation.RIGHT);

      private final Orientation orientation;

      AreaPortals(Orientation orientation) {
          this.orientation = orientation;
      }

      public Orientation getOrientation() {
          return orientation;
      }
}
```

Revise the `ICMazeArea` constructor so that it also receives the side length. The constructor must initialise the portal set accordingly.

`BossArea` is now characterised by the portal through which one enters it (an `AreaPortals` value provided at construction).

Hint: `AreaPortals` gives the direction in which the portal leads. When creating the portal in `ICMazeArea`, associate the opposite orientation (method `opposite()`). For example, portal `W` is associated with `Orientation.LEFT` because it leads to the west, but its sprite must face right due to the top-down view.

Creating `ICMazeArea` Method `createArea` of `ICMazeArea` registers the specific actors of an area. Make sure portals are registered as actors so that their behaviour is simulated.

Update the constructors of `Spawn` and `BossArea` so that both areas have size 8 and:

- the east portal of `Spawn` is visible and locked with key `Integer.MAX_VALUE`;
- the entry portal of `BossArea` is open.

When creating the areas, also specify that:

- the entry portal of `BossArea` is the west portal;
- the east portal of `Spawn` leads to `BossArea` and its arrival coordinates correspond to that area's west coordinates;
- the west portal of `BossArea` leads back to `Spawn` with arrival coordinates corresponding to `Spawn`'s east coordinates.

Introduce any helper methods you deem useful while keeping encapsulation intact (avoid intrusive getters such as one returning the list of portals).

Interacting with portals `ICMazePlayer` must now be able to walk through portals to move between areas, provided it owns the required keys.

Extend the player so that it can enter a "requesting interaction" state (`INTERACTING`):

- while in `IDLE`, when the interaction key is pressed and the player is not moving:

  ```
  keyboard.get(PLAYER_KEY_BINDINGS.interact()).isPressed()
  ```

  the state switches to `INTERACTING`;
- while in `INTERACTING`, if the interaction key is no longer held down

  ```
  !keyboard.get(PLAYER_KEY_BINDINGS.interact()).isDown()
  ```

  the state switches back to `IDLE`.

Update the interaction between `ICMazePlayer` and `Portal` so that:

- if the player is `INTERACTING`, it tries to unlock the portal (transition from locked to open if the player owns the associated key);
- if it is a contact interaction, the player transitions to the portal's destination.

Hint: only the game itself knows all areas and can move the player between them via methods `leaveArea` and `setCurrentArea`. Consequently this logic cannot reside entirely inside the player-portal interaction. The interaction should merely inform the player that it

is crossing a portal (and which one), and the player must provide the information required by the game!

2.6.4 Task

Launch `ICMaze` and confirm that it behaves like the video https://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icmaze/step1Final.mp4; specifically:

1. the movement constraints established earlier still hold (the hero cannot walk on "walls" or leave the map);
2. portals that should be invisible are indeed invisible;
3. the east portal of `Spawn` is locked;
4. the hero cannot reach `BossArea` via that portal while it is locked;
5. the hero can collect the keys;
6. the key with identifier `Integer.MAX_VALUE -1` does not open the portal that requires `Integer.MAX_VALUE` even once collected;
7. after collecting the key with identifier `Integer.MAX_VALUE`, the hero can open the corresponding portal by pressing `E` (default) and reach the next area by walking through the open portal;
8. once opened, portals remain open and the hero can travel back and forth between areas through the open portals.

2.7 Reset

To simplify testing, add the following control to `ICMaze`: key `KeyBindings.RESET_GAME` ('R' by default) must reset the game. This restarts the game in the exact same conditions as the very first launch. Note that the template's `begin` methods for the game and areas clear the relevant collections (areas, actors, etc.), so you do not need to handle that manually.

2.7.1 Task

Verify that the reset key works as expected and that the actors (hero, keys, heart, and pickaxe) reappear at their original positions. Test this key in both existing areas.

2.8 Validating step 1

To validate this step you must complete all verifications from sections 2.3.1, 2.4.4, 2.5.1, 2.5.3, 2.6.4, and 2.7.1.

The `ICMaze` game whose behaviour meets the validation steps above must be submitted at the end of the project.

Question 1

The architecture introduced in the tutorials—and used concretely in this first part—may initially feel unnecessarily complex. Its advantage is that it models, in a very general and abstract manner, the needs of many games where actors move on a grid and interact either with each other or with the grid content. How could you reuse it to implement, for example, a Pacman game? What would you need to define?

Throughout the rest of the project you will code many other interactions between actors and with cells. All future interactions must follow the pattern established in this part and must not rely on type checks.

3 Mazes (step 2)

This second part of the project focuses on procedurally creating mazes inside areas. Maze walls are represented by *rock* actors that the hero cannot cross.

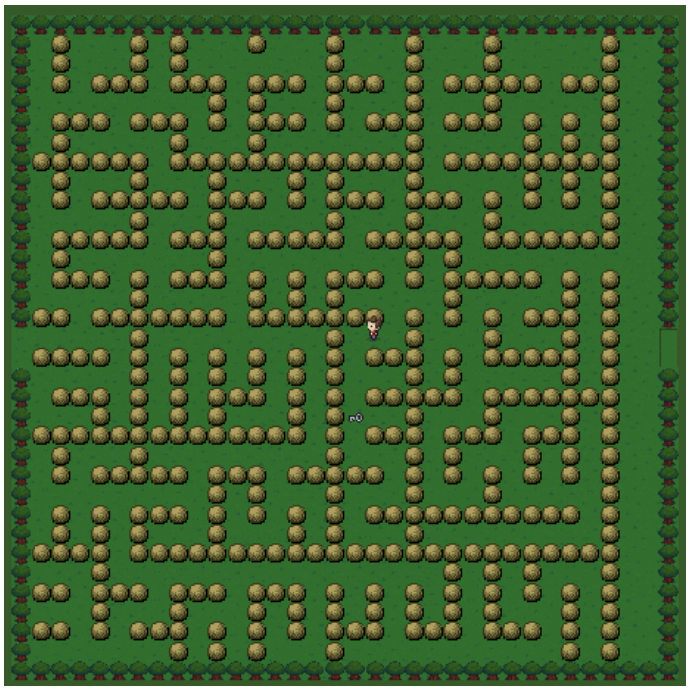

Figure 5: Example maze generated with the recursive division method.

The generation method to implement is the recursive division algorithm. The idea is to recursively split the space into sub-regions until you reach the desired granularity. The algorithm produces an integer array the size of the area in which 0 denotes an empty cell and 1 denotes an obstacle. Once the grid is created, simply place a *rock* actor on every cell whose value is 1.

Figure 5 shows a typical result in which non-traversable rocks were placed according to a maze that was generated algorithmically.

Note: a boolean grid would in principle be sufficient, but using integers makes future extensions easier (other integer values could later represent other obstacles, e.g., the value 2 for lava).

A special kind of area now appears: areas that contain mazes. Their size matters because it dictates how complex a maze you can place inside (the larger an area, the more intricate the maze can become). Concretely, you must insert between `Spawn` and `BossArea` a sequence of three increasingly large maze areas: `SmallArea`, `MediumArea`, and `LargeArea`.

3.1 Maze areas

A *maze area* is an `ICMazeArea` characterised by an *entry portal* (type `AreaPortals`, the portal that leads into the maze), an *exit portal*, the identifier of the *key* that unlocks the exit portal, and an integer *difficulty level*.

All these values are provided as constructor parameters.

Introduce three concrete maze areas:

- `SmallArea` of size 8 with grid name *`"SmallArea"`*;
- `MediumArea` of size 16 with grid name *`"MediumArea"`*;
- `LargeArea` of size 32 with grid name *`"LargeArea"`*.

The area title must mention both the grid name and the identifier of the key required to leave it; for instance:

`"icmaze/<grid_name>[key_identifier]"`

For now, wire the portals so that the exit portal is on the east side and the entry portal on the west side. `Spawn` must connect to `SmallArea` through a locked portal, just like it previously connected to `BossArea`. `SmallArea` must then lead to `MediumArea`, which in turn leads to `LargeArea`. Finally, `LargeArea` leads to `BossArea`. Exit portals start in the open state for the moment.

Assign the difficulty levels at construction time, and for this stage, you will choose the difficulty level `Difficulty.HARDEST` for all areas.

Class `Difficulty` (package `icmaze`) already provides the constants you need.

Important: In order to facilitate subsequent modifications to the game, it is recommended that you create areas in `ICMaze.createAreas()` using a utility method, which you could call `generateHardCodedLevel()`, as opposed to procedural generation of areas, which will be done later in the project.

3.1.1 Rocks

A *rock* (class `Rock`) is an `AreaEntity` drawn with sprite *`"rock.2"`* (width and height equal to 1). Rocks are non-traversable by default and accept any interaction. Their method `getCurrentCells` matches that of the hero. For now the hero only has the default interaction with rocks (do nothing). Create a rock somewhere in `Spawn`.

3.1.2 Task

Once the elements described above are implemented, launch `ICMaze` and check that:

- the rock cannot be crossed;
- the hero can still travel from `Spawn` to `BossArea` through the east portals, via the intermediate areas (which are not yet proper mazes).

3.2 Generating mazes

Class `MazeGenerator` (package `icmaze`) is responsible for actually generating the maze; you must complete it. Provide the following static method:

```
int[][] createMaze(int width, int height, int difficulty)
```

which builds a `width x height` grid using the recursive division method. Parameter `difficulty` represents the minimum size of sub-regions to divide. The larger the value, the wider the corridors (and therefore the easier the maze).

The recursive division method applies to a sub-region described by:

- `(x, y)`: coordinates of the top-left cell of the sub-region;
- `width`: its width;
- `height`: its height.

The method behaves as follows:

1. **Base case**: if the sub-region is too small (width or height $<$ or equal to `difficulty`), stop dividing.
2. Randomly decide whether to add a horizontal or a vertical wall, favouring the longer dimension to keep the maze balanced.
3. Randomly choose the wall position, ensuring it lies on an odd coordinate so that paths remain one cell wide.
4. Create an opening in the wall at an even coordinate chosen at random.
5. Apply the recursive division method to the two resulting sub-regions.

Important notes:

- Using odd/even positions ensures that walls and paths never overlap and the maze stays consistent.
- A helper method `printMaze` is provided to debug your implementation.
- Use `nextBoolean()` and `nextInt()` from the provided random generator (see 7.2) to make the random decisions.

3.2.1 Integrating mazes into areas

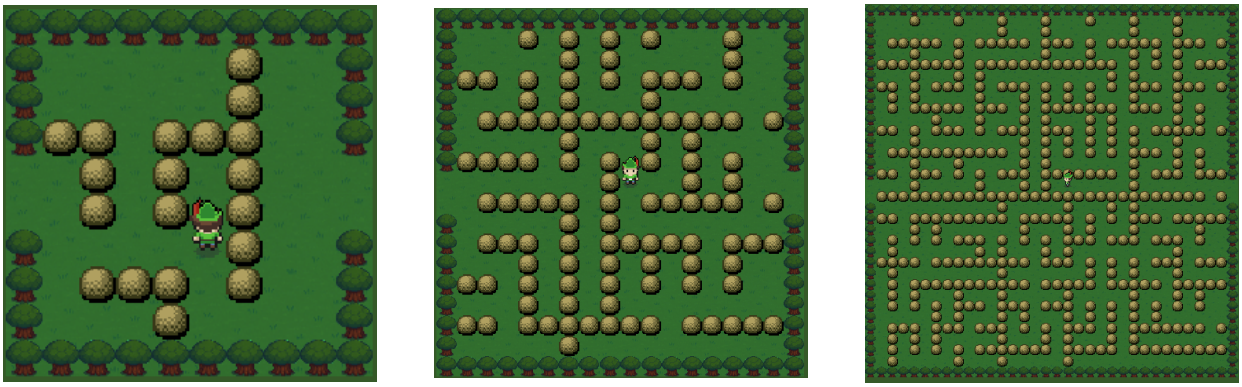

Figure 6: Example sequence of maze areas, from left to right: `SmallArea`, `MediumArea`, and `LargeArea`.

Update the maze areas so that their method `createArea` calls `MazeGenerator.createMaze`. Create a `Rock` actor in every cell whose value is `1` in the generated maze, provided that doing so does not immediately block the maze's entry or exit.

Tips:

- Use `printMaze` to inspect the generated mazes.
- Increase the scale factor to get a better overview while testing/debugging. For example, the video linked in section 3.3.1 uses a `getScaleFactor()` that returns 40 instead of 11.

Dynamic scale factor For a more comfortable experience you can define `getScaleFactor()` so that it adapts dynamically to the area size.

One heuristic is:

```
public float getCameraScaleFactor() {
    return (float) Math.min(size * DYNAMIC_SCALE_MULTIPLIER,
       MAXIMUM_SCALE);
}
```

with constants such as `DYNAMIC_SCALE_MULTIPLIER` = `1.375` and `MAXIMUM_SCALE` = `30`.

3.2.2 Task

Once the above elements are in place, launch `ICMaze` and verify that `Spawn` now leads to a chain of three maze areas with increasing difficulty as shown in Figure 6. More precisely, check that:

- solvable mazes of increasing difficulty are generated inside the maze areas;
- the entry and exit portals of each maze area are not blocked by rocks;
- the hero can walk on the paths between rocks but cannot step on the rocks themselves;
- three maze areas of increasing size/difficulty now sit between `Spawn` and `BossArea`;
- the most complex area connects to `BossArea`.

3.3 Placing keys inside the mazes

Section 6.5 of the tutorial explains how to associate a graph (`AreaGraph`) with a grid. Such a graph is useful to determine, for example, which cells are reachable and through which shortest path.

We now make life harder for the hero by locking the exit portals of the maze areas. Each maze area must contain a key that opens its exit portal. The key's identifier is the integer provided when constructing the area and it must be placed on a path cell (never on a rock).

This is where the grid's graph becomes handy. Associate an `AreaGraph` with every `ICMazeArea` and add a node via `addNode` for each cell that belongs to a maze path. You can then retrieve the discrete coordinates of those cells with `keySet()`. Once you have the coordinates in a list `lst`, randomise it with

```
Collections.shuffle(lst, RandomGenerator.rng);
```

and use the first coordinate as the random key position. Remember that `RandomGenerator` is provided in package `icmaze` (see 7.2).

Update your code accordingly so that:

- exit portals are locked with the key whose identifier was passed to the constructor;
- each `ICMazeArea` maintains an `AreaGraph`;
- the `createArea` method of maze areas adds a node to that graph for every cell that is not occupied by a rock;
- the same method creates a key at a random path position. (Keys will not be the only actors generated in mazes later on, so design the spawning logic with modularity in mind.)

3.3.1 Task

Once that logic is in place, launch `ICMaze` and verify that:

- the exit portals of the maze areas are locked;
- each maze area spawns a key at a random, non-rock position;
- the key unlocks the corresponding exit portal.

The hero's progression from room to room should resemble the example video: https://proginsc.epfl.ch/ww
projet2/videos/icmaze/step2.1.mp4

3.4 Interacting with walls

To give the hero a few extra options, allow them to break rocks with the pickaxe once it has been collected. Destroying a rock triggers a small disappearance animation (see 7.3.3); after the animation the rock is removed from the simulation.

Rocks now have hit points. Each blow with the pickaxe removes one point. When a rock disappears, there is a configurable probability (e.g., 50%) that a heart will spawn at the same position.

Implementing this behaviour requires modelling how the hero uses the pickaxe and therefore introduces a new state: "attacking with the pickaxe" (`ATTACKING_WITH_PICKAXE`).

Adapt the hero accordingly:

- if the hero is `IDLE`, not currently moving, the pickaxe key is pressed,

  ```
  PLAYER_KEY_BINDINGS.pickaxe()).isPressed()
  ```

  and the pickaxe has been collected, then the hero switches to `ATTACKING_WITH_PICKAXE` and must change animation (see appendix 7.3.1);
- if the hero is already `ATTACKING_WITH_PICKAXE`, update the current animation and, once `isCompleted()` returns true, switch back to `IDLE`.

3.4.1 Task

After implementing the above behaviour, launch `ICMaze` and verify that the hero can destroy rocks (after picking up the pickaxe in `Spawn`) as shown in this short video: https://proginsc.epfl.ch/wwwhi
projet2/videos/icmaze/step2.2.mp4

3.5 Validating step 2

To validate this step, complete all checks from sections 3.1.2, 3.2.2, 3.3.1 and 3.4.1.

Game `ICMaze`, whose behaviour satisfies the validation steps above, must be submitted at the end of the project.

4 Levels and enemies (step 3)

In this third part you enrich the design by adding adversaries that make the hero's journey through the mazes harder. You must also allow the maze areas to be generated *procedurally*.

The specifications and hints appear below.

4.1 Enemies

Enemies (`Enemy`) are actors that:

- can move on a grid;
- have hit points and a maximum hit-point value;
- die when their hit points drop to zero or below;
- request interactions themselves (they do not only receive them);
- by default accept both distance and contact interactions;
- cannot be stepped on unless they are dead;
- and, like the hero, can lose a specified number of hit points (for instance during an interaction).

The maximum hit points cannot be defined at this abstraction level, but every enemy type must expose it.

Enemies disappear from the area when they die (hit points <= 0). A short animation plays before they vanish (see 7.3.5).

PathFinderEnemy Many enemy archetypes are possible, but since this is a maze game it makes sense to introduce enemies that can follow maze paths: the `PathFinderEnemy`.

This type of enemy can decide how to move according to its own strategy. Use a method such as `Orientation getNextOrientation()` which is left undefined here and implemented by concrete subclasses. For instance, `getNextOrientation` may return the direction that leads toward a target along the shortest path between the enemy and that target.

For now implement a single concrete `PathFinderEnemy`: the "log monster".

Every `PathFinderEnemy` perceives all cells in a square neighbourhood centered on the enemy and defined by a "radius" constant specific to each subclass. These enemies request distance interactions but not contact interactions.

4.1.1 `LogMonster`

A log monster (`LogMonster`, Figure 7) is a `PathFinderEnemy` with a specific perception radius (5, for example).

It can be in three states that determine its behaviour: *sleeping*, *wandering randomly*, or *chasing a target*. The initial state is passed to the constructor. Transitions between states depend on the difficulty level of the area. The monster can also remember a target (an `ICMazePlayer`).

Figure 7: Log monsters: one chasing a target, the other asleep

Graphics Log monsters are drawn with an animation that depends on their state (see appendix 7.3.6).

Interactions The hero can now be attacked by log monsters and is therefore mortal: it owns a current and maximum hit-point count (a constant shared by all `ICMazePlayer` instances, e.g., 5). The hero starts at the maximum. When its hit points fall below zero, the hero dies and the level must reset.

Log monsters only interact when they are awake. When they interact with an `ICMazePlayer`, if they stand directly in front of the hero they remove a fixed number of points (e.g., 1). Otherwise (from further away) they memorise the hero as a target. Conversely, the hero can damage a log monster with the pickaxe, again removing a fixed amount (e.g., 1 point per attack).

Overall behaviour Log monsters rely on delays that you can implement with class `CoolDown` (package `icmaze`). A `Cooldown` is a timer initialised with a duration; method `ready` returns `true` when the countdown reaches zero. Use two `Cooldown` instances: one to govern reorientations, another for state transitions.

The behaviour is as follows:

- **Sleeping**
  1. when the reorientation timer is ready, rotate left (class `Orientation` provides method `hisLeft()`);
  2. when the transition timer is ready, switch to the random-walk state with probability

```
(double) Difficulty.HARDEST / difficulty;
```

where `difficulty` is the area's difficulty. The harder the area, the more likely the transition.

- **Random walk**
  1. when the reorientation timer is ready, choose a random orientation and move one step (see the movement mode below);
  2. when the transition timer is ready and the monster has a target in sight, switch to the chasing state with probability `pTransition`.
- **Chasing** If the monster no longer has a target it falls back to the random state. Otherwise:
  1. compute the candidate orientation via the maze-navigation strategy (method `getNextOrientation`). When the reorientation timer is ready and a new orientation exists, move one step;
  2. when the transition timer is ready, fall asleep with probability `1 - pTransition`. Thus, the harder the area, the less likely it is to return to sleep.

Use `0.75f` as the waiting time for reorientation and `3.f` for state transitions.

Movement and `getNextOrientation` Use `move` (inherited from grid actors) to advance one step; ten frames per move works well.

Method `getNextOrientation` computes, at each simulation step, the orientation that leads toward the target along a *shortest path*. Once the target position `targetPos` is known, compute the path as a `Queue<Orientation>` using:

```
path = graph.shortestPath(getCurrentMainCellCoordinates(),
   targetPos)
```

where `graph` is the `AreaGraph` of the current area. The direction to take is the first element of the queue:

```
path.poll();
```

> The graph should not be directly accessible from `LogMonster`. Avoid intrusive getters such as `getGraph` on `ICMazeArea`.

Such a shortest path may not exist (imagine the hero temporarily walking through walls and hiding somewhere unreachable).

Spawning `LogMonster` instances Extend `createArea` of the maze areas so that, in addition to placing keys, it creates log monsters at random positions. Cap the number of enemies per maze (e.g., at 3). Determine the actual number based on the area's difficulty: the harder the area, the higher the count.

Use the following heuristic to compute the probability of spawning a new enemy:

```
diffRatio = Math.min(1.0, (double) Difficulty.HARDEST /
   difficulty);
```

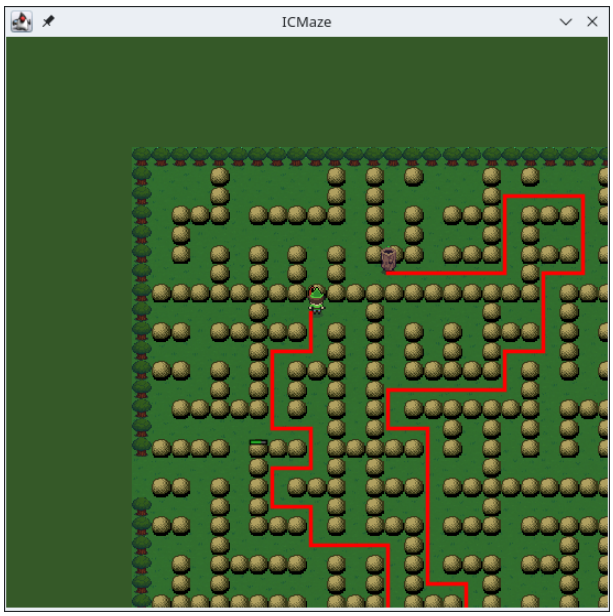

Figure 8: Explicit display of the path followed by a log monster.

While the maximum count is not reached, add another enemy whenever `rng.nextDouble()< 0.25 + 0.60 * diffRatio`. After deciding how many enemies to spawn, pick their positions at random just like you did for keys.

The initial state of each `LogMonster` is chosen randomly with the following probabilities:

- chasing: `pTarget = 0.10 + 0.70 * diffRatio`
- random walk: `pRandom = 0.20`
- sleeping: `pSleeping = 1 - (pTarget + pRandom)`

Draw `rng.nextDouble()`. If the value is less than `pSleeping`, spawn the monster asleep; if it is less than `pSleeping + pRandom`, spawn it in random-walk mode; otherwise spawn it in chasing mode.

4.1.2 Task

After implementing the above logic, launch `ICMaze` and verify that:

- log monsters now spawn automatically in the mazes;
- they can start in different states, and the harder the area, the less likely they are to start asleep and the more likely they are to start in chase mode;
- they can move randomly;
- while asleep they are harmless and display their specific animation;
- they can switch between states, and the transition from sleeping to chasing becomes more probable as the difficulty increases, while the chance of dozing off decreases;
- in chase mode they can track the hero throughout the maze;
- once nearby they damage the hero and can “kill” it, triggering a reset of the level.

To ease debugging, you can draw the paths they follow. Declare an attribute representing

the graphical path (e.g., `graphicPath` of type `Path`). Every time you compute a new path `path`, update the graphical counterpart with:

```
graphicPath = new Path(this.getPosition(), new
   LinkedList<Orientation>(path));
```

Then draw `graphicPath` (if not null) inside the monster's draw method so the path appears explicitly as in Figure 8.

4.2 Health bars

Figure 9: Hero with health bar

Now that the hero faces enemies that can inflict damage, it is useful to visualise their health explicitly (Figure 9).

Class `Health` (package `actor`) is provided to simplify matters. Associate a health bar to every `ICMazePlayer` and `LogMonster` using:

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

Here `friendly` is `true` for the hero and `false` for enemies. The first argument attaches the bar to the current actor, the second offsets it in local coordinates, the third is the maximum life, and the last colours the bar green or red depending on whether the actor is friendly (default) or hostile. You may also give rocks a health bar if desired.

Study `Health` to see how to increase or decrease the reflected life. Model the following:

- `ICMazePlayer` gains life when collecting hearts (e.g., +1 per heart);
- whenever an actor loses life, it benefits from a short immunity period whose duration is identical for all instances of that class. During immunity the actor ignores damage.

Draw the health bar inside the actor's draw method. You may choose to draw it only when the actor is not immune and still has points to display.

Hint: implement immunity periods with a `Cooldown`. A value such as 24 frames works well.

Rendering tweak: To visualise immunity, modify the rendering of the hero, rocks, and enemies so that while immune their animation is drawn only every other frame (e.g., when the immunity counter is even).

Reset tweak: The reset controls must now restore a full health bar both when resetting the game and when resetting the current area. Consider that actors also lose their immunity on reset.

4.2.1 Task

Implement the concepts described above.

Launch `ICMaze` and ensure it behaves as follows :

1. a health bar appears next to the hero (this can be all the time or during periods of immunity, depending on your choice), starts fully green, and follows the hero;
2. a log monster damages nearby heroes and the bar reflects the loss;
3. collecting a heart restores life and the bar grows accordingly;
4. the hero and the log monster blink when they just took damage (signalling temporary immunity).

Also verify that resets restore health and that the death of either character triggers a reset of the current area.

4.3 Procedural level generation

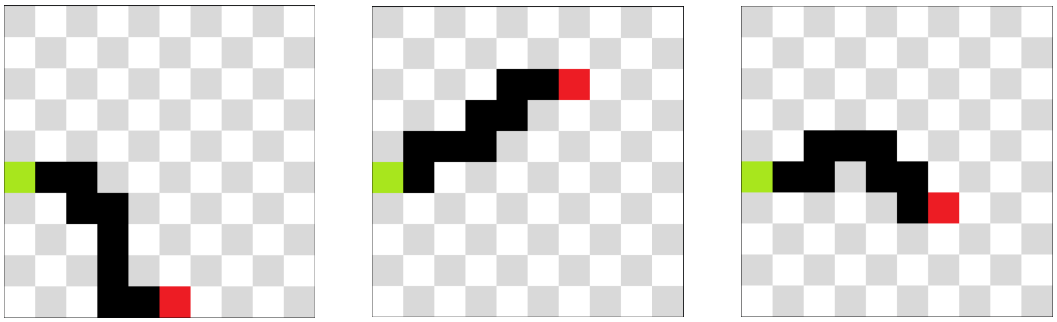

Figure 10: Three possible examples of a level of length 8.

Up to now the maze areas were hard-coded. We now replace them with a succession of procedurally generated areas. The suggested algorithm produces a *linear* level, i.e., a chain of connected areas and no branches (Figure 10).

The process is:

1. start with the spawn area (`Spawn`);
2. append a configurable number of dynamically generated areas;
3. place each new area "forward" (north, east, or south), never backwards;
4. gradually increase the difficulty and size as the player progresses;
5. end with the boss area (`BossArea`).

The generator ensures areas never overlap and that connections are bidirectional. Class `LevelGenerator` implements the logic and exposes:

```
public static ICMazeArea[] generateLine(ICMaze game, int length)
```

It must build a linear level of length `length` and return the areas (`ICMazeArea[]`) in order. Parameter `game` is the game instance passed to each area, and `length` is the strictly positive number of maze areas (excluding the spawn and boss areas).

Detailed behaviour of `generateLine` Rooms are placed in an abstract grid whose origin is the spawn room. Each room occupies a discrete position [i][j]. The algorithm is:

1. create `Spawn` at `[0][0]`;
2. while the desired length is not reached:
   - pick a free direction among north, east, or south (never backwards). If east is chosen, create the room at `[x+1][y]`.
   - ensure the target coordinates are free (store occupied positions in a `Set`, cf. tutorial appendix 2);
   - compute the difficulty based on the player's progress (number of rooms already created);
   - instantiate a `SmallArea`, `MediumArea`, or `LargeArea` based on that difficulty;
   - connect the previous area to the new one, and vice versa.
3. once all maze areas are created:
   - append a `BossArea` after the last area;
   - connect the two areas.

Return the array ordered from `Spawn` to `BossArea`.

Evolution of difficulty and size Let `i` be the number of areas generated so far. Define the progress ratio:

```
  double progress = (double) (i + 1) / length;
```

This value lies between 0 and 1. Convert it into a difficulty constant:

```
int difficulty = switch ((int) (progress * 4)) {
    case 0 -> Difficulty.EASY;
    case 1 -> Difficulty.MEDIUM;
    case 2 -> Difficulty.HARD;
    default -> Difficulty.HARDEST;
};
```

Use the same ratio to pick the area type:

```
double r = rng.nextDouble();
```

```
if (r < progress * progress)
    return new LargeArea(...);
if (r < progress)
    return new MediumArea(...);
return new SmallArea(...);
```

Interpretation

- near the beginning (`progress` close to 0) areas tend to be small;
- around the middle (`progress` $\approx$ `0.5`) `MediumArea` become more frequent;
- near the end (`progress` $\approx$ `1`) `LargeArea` are very likely because `progress` is large.

Hence, globally, as the player advances, areas become harder and larger, which smoothly ramps up the challenge. This progression is entirely procedural and gives the game a natural sense of momentum without hand-written scripts.

Note: due to random draws, it remains possible for the player to enter an easier area after a more difficult one.

Updating `ICMaze` Update `ICMaze.createAreas` so that maze areas are produced via `generateLine` instead of being hard-coded. Treat the number of maze areas as a configurable constant for testing.

> When making this change, comment out the call to `generateHardCodedLevel()` instead of deleting it so that you can revert if needed.

4.3.1 Task

After implementing the above, launch `ICMaze` and verify that:

- maze areas of increasing difficulty are generated automatically;
- transitions between areas work in both directions;
- the hero still starts in `Spawn` and the final room is `BossArea`;
- the number of generated maze areas matches the configured length (length zero sends the hero straight from `Spawn` to `BossArea`).

4.4 Validating step 3

To validate this step, complete all checks from sections 4.1.2, 4.2.1, and 4.3.1.

> Game `ICMaze`, whose behaviour is described above, must be submitted at the end of the project.

5 Final challenge (step 4)

This last step is much freer. Only a specification of the features to implement and a few hints are provided.

When the hero enters `BossArea`, the entry portal must lock to prevent backtracking. The player must then face a belligerent enemy, aptly named `Boss`, who uses projectiles to inflict damage. The outcome of the fight can reopen the entry portal and alter the content of already visited areas.

Finally, this step also introduces simple dialogues so that the player receives tips at the start of the game.

> This step must rely on logical signals (`Logic`). For example, defeating the `Boss` and collecting the key it drops should dictate the behaviour of `BossArea` as a `Logic`. That signal can in turn control other areas.

5.1 Ultimate enemy: the `Boss`

Fatal trap Start by locking the entry portal of `BossArea` with a key whose identifier is -1. Verify that once inside `BossArea` the hero is trapped.

Next implement the `Boss` enemy and the projectiles it can launch.

5.1.1 Projectiles

A projectile is a grid-moving actor, an interaction requester, that can fly over cells and that the hero can walk through.

A projectile is characterised by its speed and the maximum distance it can travel from its starting point. Treat these characteristics as fixed and identical for all instances (e.g., speed 1 and maximum distance 7). It moves in its orientation and disappears when it reaches the maximum distance. Use the speed to modulate the displacement (e.g., `move(MOVE_DURATION/speed)`). Represent the remaining distance as an integer; at each simulation step decrease it by one.

By default a projectile accepts neither contact nor distance interactions. It can, however, inflict contact interactions as long as it is still travelling (anything it touches along the path can be affected).

A projectile must also be stoppable before reaching its maximum distance.

At this stage you only need one concrete subclass, either `WaterProjectile` or `FireProjectile`. Draw them with the animations described in 7.3.4. They deal a fixed amount of damage to the hero on contact (e.g., 1 point).

5.1.2 The `Boss` actor

The ultimate enemy (`Boss`, Figure 11) does not require maze-specific logic. Its maximum hit points are shared by all instances (e.g., 5). It uses animation *`"icmaze/boss"`*.

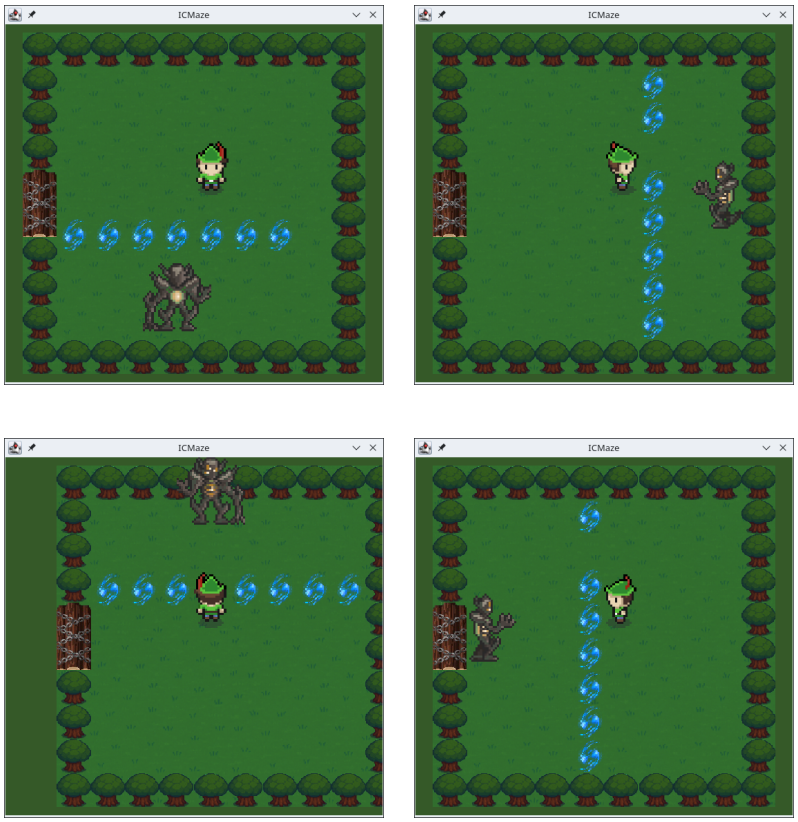

Figure 11: The `Boss` can teleport randomly to one of these positions and fires projectile barrages along a full row or column, leaving one random safe cell.

The hero can attack it with the pickaxe, inflicting a fixed `damage` according to this simple algorithm:

- on the first attack it teleports to a random point in the area (see below) and starts firing projectile barrages around itself at regular intervals;
- subsequently it teleports again, remembers it has already been attacked, and also loses `damage` hit points.
- When it dies it drops a key that opens the `BossArea` entry portal. Choose the identifier accordingly.

The boss behaviour is straightforward:

- if it has never been hit, it does nothing special;
- otherwise, if the current animation has not finished, update it; when it finishes, emit projectile barrages at regular intervals and reset the animation.

Teleporting to a random position For simplicity the teleport destinations are fixed: the centre and the midpoint of each outer wall (see Figure 11). The boss always faces forward after teleporting. Pick one of those positions at random, avoiding the current one.

Projectile barrages Projectiles are fired along the row or column immediately in front of the boss, covering every cell except one chosen at random (Figure 11). The free cell gives

the hero a way to keep attacking.

For testing, place a `Boss` in the middle of `BossArea`. Temporarily set the number of maze areas to zero to make testing easier.

5.1.3 Task

Implement the concepts described above and then verify that:

1. the entry portal of `BossArea` is locked;
2. the `Boss` appears idle in this area;
3. it becomes active as soon as the hero lands the first hit and teleports to one of the positions in Figure 11;
4. once active, it launches projectile barrages in front of it at regular intervals;
5. these projectiles cover the entire row or column in front of the `Boss`, except for one random cell;
6. the projectiles move forward and damage the hero;
7. they disappear once they reach the area boundaries;
8. the `Boss` drops a key when defeated;
9. the dropped key opens the `BossArea` entry portal.

5.2 Logical signals

The tutorial introduced *signals*; now use them to finalise the game logic. The idea is to make an area's behaviour, and possibly that of its actors, depend on logical signals.

More precisely, associate a challenge with every area and consider an area "resolved" once the challenge is met. The area's behaviour and the actors inside it can then depend on whether the challenge is resolved. An area's challenge can also depend on the challenge of another area.

We therefore want each area to behave like a `Logic` signal that turns on when its challenge is complete and can be consumed by other areas or actors to adjust their behaviour.

Make the necessary changes so that the following conditions hold:

- An `ICMazeArea` acts as a logical signal that is on when its challenge is solved and off otherwise.
- The `BossArea` challenge is to defeat the `Boss` and collect the dropped key.
- Maze areas are considered solved as soon as `BossArea` is solved. In that case the mazes disappear (no drawing, rocks no longer block paths) and log monsters remain permanently asleep.
- `Spawn` is solved when the `Boss` area is solved. A treasure of your choice appears as the hero's reward.

Figure 12: Simple dialogs can document the controls or provide tips.

Implement this without:

- using intrusive getters;
- making areas know each other directly (an area should not be aware of other areas as such);
- storing overly specific information in actors (e.g., a flag indicating that a particular key was collected). You should be able to change an area's challenge without impacting other code.

Note: For testing purposes, it is recommended to keep the `generateHardCodedLevel()` method executable. Be sure to make any changes necessary due to your new additions.

5.2.1 Task

Implement the concepts above according to the specifications, then verify that:

1. once the boss drops its key, the `BossArea` entry portal reappears and the hero can open it after collecting the key;
2. maze areas no longer display a maze and the log monsters inside remain asleep;
3. a treasure appears in `Spawn`;
4. the `reset` key restarts the game in its initial state (all challenges unresolved).

5.3 Dialogues (optional)

Many controls are already used in interactions, so documenting them at the start of the game is useful. Some sections can also benefit from short hints. Introduce a simple dialogue system for these needs (Figure 12).

The template already includes class `Dialog` (package `engine.actor`). Create a dialogue by associating it with a text resource (sub-folder `dialogs`), for example:

```
new Dialog("welcome");
```

Method `update` on `Dialog` progresses through the text and `isCompleted` returns `true` once the text has finished.

Use dialogues to provide information to the player by equipping `ICMaze` with:

- a `Dialog` attribute that represents the current hint (the "active dialogue"); a dialogue is active when this attribute is not `null`;
- a method `setActiveDialog` that sets the attribute.

Update `ICMaze` so that:

- the current area no longer updates while a dialogue is active (only rendering occurs) and the active dialogue is also drawn;
- the active dialogue (if any) updates when key `NEXT_DIALOG` is pressed:

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

  where `kbd` is the game's `Keyboard`;
- the game resumes as soon as the active dialogue finishes (method `isCompleted()`).

5.3.1 Activating a dialogue

The question now is how to assign a value to the active dialogue attribute, knowing that the components which need to trigger it usually are not the game itself. For instance, if a hint must appear when the hero interacts with an actor, the interaction handler needs to call a method that publishes a dialogue. Similarly, if an area needs to display a hint on start-up, the area must know how to set the dialogue. Use an interface to avoid exposing the entire game to every component that needs to trigger a dialogue.

Define interface `DialogHandler` with a single method `void publish(Dialog)`. Make the game implement this interface and override `publish` so that it sets the current dialogue.

Finally, update `Spawn` so that:

- the area only knows the game through the `DialogHandler` interface (less intrusive view);
- during the first call to `update`, the active dialogue becomes `"welcome"`. This dialogue must not reappear if the player returns to `Spawn` after visiting another area.

Welcome dialogue and resets Key `GAME_RESET` restarts the game from scratch, so the welcome dialogue should appear again afterwards. By contrast, key `AREA_RESET` must not re-display this dialogue.

5.4 Task

Implement the concepts above according to the specifications, then verify that:

1. when the game starts the introductory dialogue appears and can be scrolled with `NEXT_DIALOG`;
2. while the dialogue is visible the characters no longer react to controls;
3. once the dialogue ends, the game resumes its usual behaviour;
4. the reset keys behave as expected with respect to the welcome dialogue.

5.5 Validating step 4

To validate this step, complete all checks from sections 5.1.3 and 5.2.1. The checks of section 5.4 are not mandatory.

Game `ICMaze`, whose behaviour is described above, must be submitted at the end of the project.

6 Extensions (step 5)

To obtain bonus points that can compensate for possible penalties on the mandatory part of the project, or to participate in the contest, you can code some freely chosen extensions. At most 15 points will be counted (coding many extensions to compensate for weaknesses in previous parts is therefore not a possible option).

The implementation is free and with very little guidance. Only a few suggestions and indications are given below. A grade estimate for the suggested extensions is given, but do not hesitate to contact us for a more precise evaluation if you have a particular idea. A small bonus will be awarded if you show inventiveness in the game design.

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

You can also, alternatively, create a new game `ICoopExtension` using the logistics that you set up in the previous steps.

Make sure to **comment carefully**, in your `README` file, the game modalities of your extensions. We must, in particular, know which controls to use without having to read your code.

Here is an example of a game you could achieve as well as a (partial) `README` corresponding to it that explains how to play:

- Example game video (short excerpt): ICMaze.mp4

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

> It is expected of you that you choose some extensions and code them to the end. The idea is not to start coding lots of small bits of disparate and unfinished extensions to collect the necessary points ;-).

Note that there exist in the provided material, some resources for additional areas.

6.1 New actors or character extensions

All kinds of actors can be considered. In particular, the "signal" component can be used to create game scenarios linked to solving more or less complex puzzles. A (non-exhaustive) list of suggestions is given below.

- new `PathFinderEnemy` with different strategies to navigate in mazes; (~4 to 6 points)
- modeling of a resource system (gold, silver, wood, food, healing doses etc); (~4 to 6 points)
- add a `Box` or `Safe` object, whose opening would be controlled by a signal and which would have one or more objects as content; (~3pts)
- various actors that can serve as signals (orbs, torches, pressure plates, levers); (~4pts)
- animated scenery actors; (~2pts)

- advanced signals for puzzles (oscillators, signals with delay): an oscillator is a signal whose intensity varies over time; (~4pts/signal)
- all kinds of characters with specific movement and behavior modalities; which can be hostile or friendly towards the player; (~3 pts to 6 pts depending on the character's complexity)
- vendor characters with an inventory and from whom main characters can buy equipment after collecting coins; the notion of inventory will need to be enriched to allow items to transit from one inventory to another; this will involve coding a graphical menu allowing to display inventories in their entirety (with the number of items of each type and to select an item, for example the one we want to buy) (~5 to 10 points): Note that the menu associated with an inventory can be coded even without a vendor character.
- create new movement modes for characters (running, swimming, etc.) with an adequate adaptation of sprites/animations and scenery elements; (~3 to 5 pts)
- create follower characters like Red's Pikachu in Pokemon Yellow; (~3pts)
- create one or more scenario events triggered with signals. For example a character who arrives in the area to give an object or give an instruction. (~3 to 5 pts)
- add new types of cells with appropriate behaviors (water, ice, fire, etc); (~2pts/cell)
- implement a day/night cycle that could serve as a signal or that would condition the behavior of characters (for example they can no longer move forward if it is too dark and they should get a flashlight); (~5pts)
- add a shadow or reflection to the player and certain actors; (~2 to 3pts)
- add new advanced controls (interactions, actions, movements, etc); (~2pts/control)
- add random events (scenery, signals, etc.); (~4pts)
- add multiple-choice dialogues; (~3 to 6 pts)
- etc.

6.2 Pause and end of game (~2 to 5pts)

The notion of area can be exploited to introduce game pausing. Upon player request, the game can switch to pause mode then switch back to game mode. You can also introduce end of game management (if the characters have reached an objective or have been defeated for example).

In reality, the base that you have coded can be enriched at will. You can also let your imagination speak, and try your own ideas.

If an original idea comes to you that seems to differ in spirit from what is suggested and you wish to implement it for the submission or the contest (see below), it must be validated before continuing (by sending an email to CS107@epfl.ch).

Appendix 3 of the tutorial gives you indications for enriching graphical resources.

Be careful however not to spend too much time on the project to the detriment of other subjects!

6.3 Validation of step 5

As a final result of the project, create a game scenario well documented in the `README` and involving all the coded components. A (small) part of the grade will be linked to the inventiveness and originality you will demonstrate in the game design.

6.4 Contest

People who have finished the project with a particular effort on the final result (interesting gameplay, richness of game areas, visual effects, interesting/original extensions, etc.) can compete for the "best CS107 game" prize.[4]

If you wish to compete, you will need to send us by **18.12 at 13:00** a small "application file" by email to the address **cs107@epfl.ch**. It will consist of a description of your game and the extensions you have incorporated (on 2 to 3 pages in .pdf format with some screenshots highlighting your additions).

> The winning projects will be the subject of a presentation during the return week (February).

[4] 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 keys usable in the project. It is coded using the notion of `record` which will only be seen in the second semester. A `record` is an abbreviated way of writing a restricted form of a class. For this project, it is sufficient to know that:

- the keys to use for controls linked to the character are defined in `PLAYER_KEY_BINDINGS` (and can be freely modified according to your preferences);
- the type `PlayerKeyBindings` makes it possible to ensure that, **in that order**, the keys correspond to the controls: "move up", "left", "down", "right", "equipment change" (selection of current equipment in inventory) and "use current equipment";
- to provide an `ICMazePlayer` with a specific set of keys, it is sufficient to provide it with an attribute of type `KeyBindings.PlayerKeyBindings`;
- when `keys` is an object of type `KeyBindings.PlayerKeyBindings`, it is sufficient to write `keys.pickaxe()` (and analogously, `keys.right()`, `keys.down()`, etc) to reference a given key: the one associated with using the pickaxe for example. The `moveIfPressed` method of characters can invoke:

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

  or during the `update` of a character, one can test if the equipment use key is pressed by:

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

Note: The default configuration is for a QWERTZ keyboard. For an AZERTY keyboard one would rather have:

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

in `KeyBindings.java`.

7.2 Random number generation

Java allows to generate so-called "pseudo-random" numbers using a class named `Random`. In the `icmaze` package, you will find a `RandomGenerator` class that offers a generator instance of type `Random`. This allows to have only one object of type `Random` in the project and to allow creating deterministic test situations for debugging purposes, as explained a bit further.

Through the object `RandomGenerator.rng`, it is possible to invoke all the random generation methods offered by the API of the Random class, for example:

```
RandomGenerator.rng.nextInt(10);
```

allows to draw a random integer (with uniform probability distribution) between 0 and 10.

The algorithms implemented by pseudo-random number generators use what is called a "seed" which by default is random. For the same seed value, it is always the same sequence of numbers that is drawn. We can therefore use this characteristic to allow creating reproducible test situations during debugging (otherwise programs with random behaviors would be difficult to correct, because it is not guaranteed that from one execution to another we are in the same conditions).

If you want to force the generator to start from a given seed, you just need to replace the line:

```
public static Random rng = new Random();
```

by

```
public static Random rng = new Random(val);
```

where `val` is the value of the chosen seed, in the `RandomGenerator` class.

7.3 Creating animations

The animation code is not very interesting to produce. Therefore, you can find below an example of code for each of the classes you might want to animate.

7.3.1 `ICMazePlayer`

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)
```

with `ANIMATION_DURATION` equal to 4 for example and `prefix` equal to *`"icmaze/player"`*.

Using the pickaxe

```
final Vector anchor = new Vector(-.5f, 0);
final Orientation[] orders = {DOWN, UP, RIGHT, LEFT};
... new OrientedAnimation(prefix+".pickaxe",
                          PICKAXE_ANIMATION_DURATION, this,
                          anchor, orders, 4, 2, 2, 32, 32)
```

with `PICKAXE_ANIMATION_DURATION` equal to 5.

7.3.2 Hearts

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

where `ANIMATION_DURATION` equals 24.

7.3.3 Disappearing in a cloud

```
new Animation("icmaze/vanish", 7, 2, 2, this, 32, 32, new
   Vector(-0.5f, 0.0f), ANIMATION_DURATION/7, false);
```

where `ANIMATION_DURATION` equals 24.

7.3.4 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 *"icmaze/waterProjectile"* for water projectiles and *"icmaze/magicFireProjectile"* for fire ones.

7.3.5 Enemy death

Enemies disappear in a small cloud when dying (as in 7.3.3).

7.3.6 Log monsters

When awake:

```
// When targeting the player
 Orientation[] orders = new Orientation[]{Orientation.DOWN,
    Orientation.UP, Orientation.RIGHT, Orientation.LEFT};
... new OrientedAnimation("icmaze/logMonster",
                          ANIMATION_DURATION/3, this,
                          new Vector(-0.5f, 0.25f), orders,
                          4, 2, 2, 32, 32, true)
// When moving randomly
orders = new Orientation[]{Orientation.DOWN, Orientation.UP,
   Orientation.RIGHT, Orientation.LEFT};
... new OrientedAnimation("icmaze/logMonster_random",
                          ANIMATION_DURATION/3, this,
                          new Vector(-0.5f, 0.25f), orders,
                          4, 2, 2, 32, 32, true)
// Asleep
orders = new Orientation[]{Orientation.DOWN, Orientation.LEFT,
   Orientation.UP, Orientation.RIGHT};
... new OrientedAnimation("icmaze/logMonster.sleeping",
                          ANIMATION_DURATION/3, this,
                          new Vector(-0.5f, 0.25f), orders,
                          1, 2, 2, 32, 32, true)
```

where `ANIMATION_DURATION` equals 30.

7.3.7 Boss

```
final Vector anchor = new Vector(-0.5f, 0);
final Orientation[] orders = {DOWN, RIGHT, UP, LEFT};
...new OrientedAnimation("icmaze/boss", ANIMATION_DURATION/3,
                         this, anchor, orders, 3, 2, 2, 32, 32,
                         true)
```

where `ANIMATION_DURATION` equals 60.