



# 1 Demo game

For demonstration purposes I have written a demo game that more or less demonstrates the requirements from this task description. It is, however, *not* a reference implementation. I might extend or change the demo throughout the course and implement a few additional features myself, so it is probably not a good idea to rely on the demo too much.

Nevertheless, the demo game may serve to understand the task description better and to experiment a bit with a small and simple game.

Of course, it is also recommended to try other roguelike games for ideas.

The demo game is available via the student Linux shell server. Use `ssh` (or PuTTY on Windows) to open a secure connection to `shell.students.cs.uu.nl`, then modify your search path:

```
$ export PATH=/praktikum/afp/linux/bin:$PATH
```

(Don't add any extra spaces.)

Then call `LambdaHack` to play the game. Here are a few keys you can use in the game:

key	command
k	up
j	down
h	left
l	right
y	up-left
u	up-right
b	down-left
n	down-right
<	level up
>	level down
S	save and quit the game
Q	quit without saving
o	open a door
c	close a door
s	search for secret doors
.	wait
,	pick up an object
:	look around
v	display the version of the game
V	toggle field of vision display
O	toggle "omniscience"
M	display level meta-data
R	toggle smell display
T	toggle level generation sequence

Pressing a capital letter corresponding to a direction key will have the character run in that direction until something interesting occurs.

## 2 Dungeon

### 2.1 Dungeon layout

The player should be able to explore a dungeon. The dungeon consists of multiple levels (at least 10), and each level consists of at least 80 by 21 tiles.<sup>1</sup>

At least the following tiles should be implemented:

tile type	symbol in demo game
floor	.
wall (horizontal and vertical)	- and
corridor	#
stairs (up and down)	< and >
rock	invisible

The player is able to move around floors, corridors and stairs, but not through walls or rock.

The game world should be persistent, i.e., every time a player visits a level during one game, the level should look the same.<sup>2</sup>

### 2.2 Level generation

Each level is generated by an algorithm inspired by the original Rogue, as follows:

- The available area is divided into a 3 by 3 grid where each of the 9 grid cells has approximately the same size.
- In each of the 9 grid cells one room is placed at a random location. The minimum size of a room is 2 by 2 floor tiles. A room is surrounded by walls, and the walls still have to fit into the assigned grid cells.
- Rooms that are on horizontally or vertically adjacent grid cells may be connected by a corridor. Corridors consist of 3 segments of straight lines (either “horizontal, vertical, horizontal” or “vertical, horizontal, vertical”). They end in openings in the walls of the room they connect. It is possible that one or two of the 3 segments have length 0, such that the resulting corridor is L-shaped or even a single straight line.

---

<sup>1</sup>This minimal size goes back to a standard terminal size of 80 columns and 25 lines. A few of the lines are used to display status information, the rest is available to display the level.

<sup>2</sup>This may sound obvious – I’m only saying that because there *are* roguelike games that don’t allow you to go back up, or that generate a new random level every time you change levels.

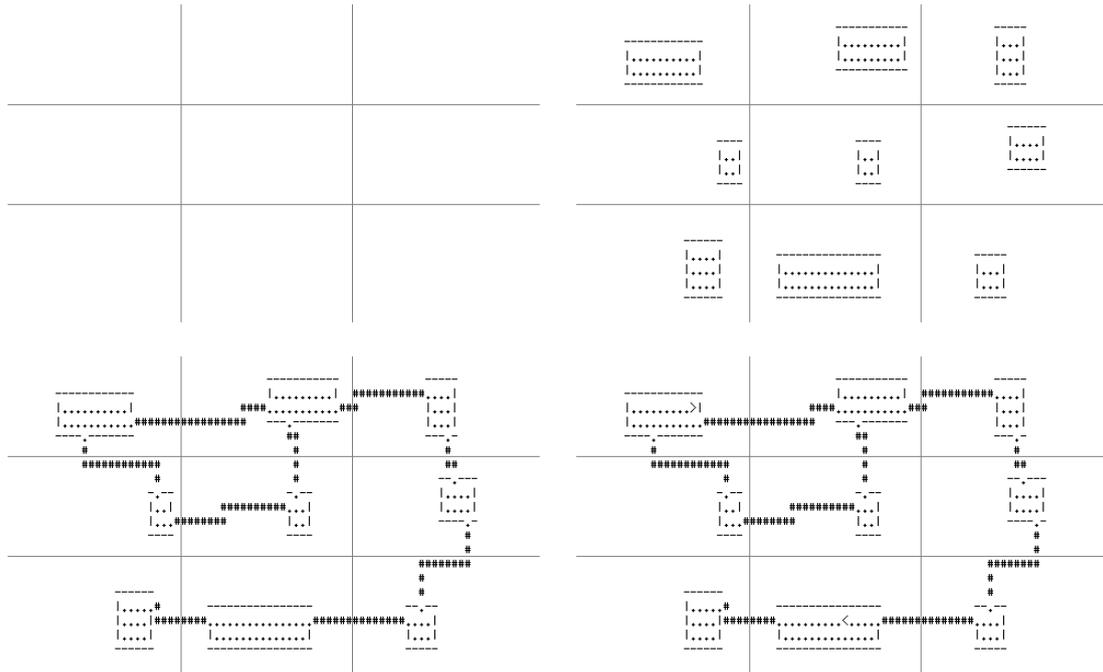


Figure 2: Level generation in four steps: first (upper left) we have an empty grid; we then (upper right) generate a room in each cell; then (lower left) corridors are added; finally (lower right) stairs are placed

- Corridors are generated randomly in such a way that at least every room on the grid is connected, and a few more might be. It is not sufficient to always connect all adjacent rooms.
- Stairs up and down are placed. Stairs are always located in two different randomly chosen rooms.

The algorithm should be written in such a way that it can easily be applied to other map and grid sizes.

Figure 2 visualizes a sample level generation.

### 3 The player

#### 3.1 Movement

The player (@ in the demo game) should start the game on the staircase up of the first dungeon level. The player should be able to move around horizontally, vertically, and diagonally in the dungeon, and should be able to change levels at staircases.

It should be possible to direct the player using keyboard or mouse. Just for reference, the demo game uses the Rogue keybindings for player movement, which in turn are inspired by the editor Vi. The list is given in Section 1.

### 3.2 Status

Add a status line that displays some current information about the player's situation in the game world, such as the current level, or the player's health.

### 3.3 Vision and memory

Only parts of the game map that the player has already explored should be shown. Only parts of the map that the player can currently see should be shown as they currently are, other parts should be shown as the player remembers. This requires to implement an algorithm that determines what the player can see. Intuitively, the player can see a tile in the map if the player has an unobstructed line of sight to that tile.

We first specify fields that are reachable from the player. As input to the algorithm, we require information about fields that block light. As output, we get information on the reachability of all fields. We assume that the player is located at position  $(0,0)$ , and we only consider fields  $(line, row)$  where  $line \geq 0$  and  $0 \leq row \leq line$ . This is just about one eighth of the whole player surroundings, but the other parts can be computed in the same fashion by mirroring or rotating the given algorithm accordingly.

```

fov (blocks, maxline) =
  shadow      :=  $\emptyset$ 
  reachable (0,0) := True
  for l  $\in$  [1 .. maxline] do
    for r  $\in$  [0 .. l] do
      reachable (l,r) :=  $(\exists \alpha . \alpha \in \text{interval } (l,r) \wedge \alpha \notin \text{shadow})$ 
      if blocks (l,r) then
        shadow := shadow  $\cup$  interval (l,r)
      end if
    end for
  end for
  return reachable

```

```

interval (l,r) = return [angle (l + 0.5, r - 0.5), angle (l - 0.5, r + 0.5)]
angle (l,r)   = return atan (r / l)

```

Look at Figure 3 for visual help. Lines are depicted upwards, rows to the right. The algorithm traverses the fields line by line, row by row. At every moment, we keep in *shadow* the intervals which are in shadow, measured by their angle.

A square is reachable when any point in it is not in shadow – the algorithm is permissive in this respect. We could also require that a certain fraction of the field is reachable,

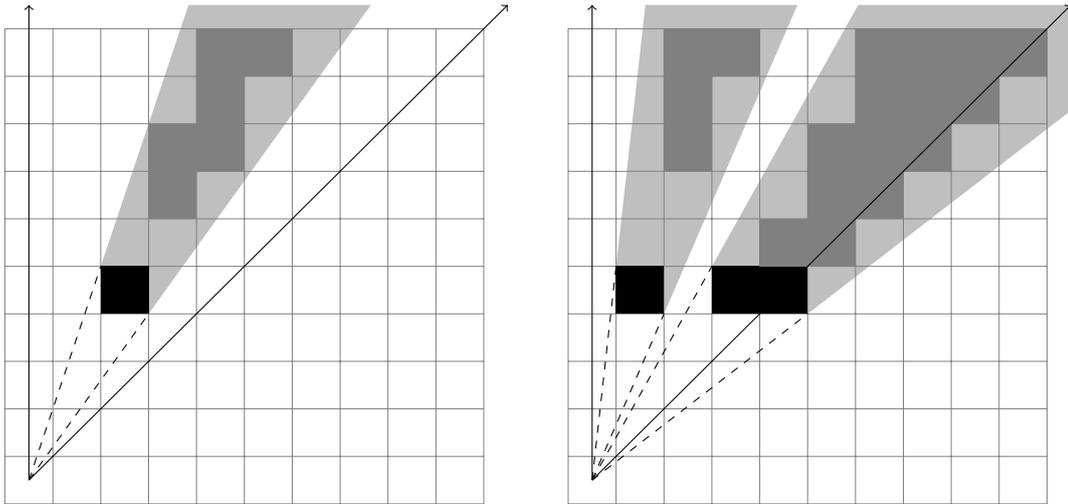


Figure 3: Visualization of the FOV algorithm

or a specific point. Our choice has certain consequences. For instance, a single blocking field throws a shadow, but the fields immediately behind the blocking field are still visible, as can be seen from the left example image.

We can compute the interval of angles corresponding to one square field by computing the angle of the line passing the upper left corner and the angle of the line passing the lower right corner. This is what *interval* and *angle* do.

If a field is blocking, the interval for the square is added to the *shadow* set.

**Optimization** Note that you will probably have to optimize the algorithm. For instance, the algorithm expects an argument *maxline*. Generally, it is possible to stop the computation once the whole range we consider is in shadow, even before we reach a given maximum line. Also, it pays off to choose a suitable representation for *shadow* that joins intervals and only remembers the points where light changes to shadow and shadow changes to light.

### 3.4 Light

Once you can compute the reachable fields using *fov*, it is possible to compute what the player can actually see.

Fields adjacent to the player (also diagonally) can always be seen (except for walls, see below). Fields that have light and are reachable can also be seen. We treat floor of rooms as having light, whereas corridors and rock are dark.

### 3.5 Light and walls

Walls reflect light. They can be seen only if an adjacent floor field can also be seen. In particular, walls cannot be seen when passing a corridor on the outside of a room, but can be seen from the inside of a room.

## 4 Saving games

It should be possible to save the game to a file, and restore from there. Everything about the game should be stored, so that restoring continues in exactly the same situation.

As most roguelikes, you should remove the save game after successfully restoring from it.

## 5 Monsters

The player is not alone in the dungeon. Monsters should roam the game world, too.

Monsters inhabit a specific location on the game map, and can be seen if the field they are on can be seen by the player. Monsters are not remembered, i.e., they are removed from the display once the player can no longer see them.<sup>3</sup>

### 5.1 Monster generation

Monsters are generated in random time intervals. Of course, monsters should only be generated on empty squares (no other monsters, no player, no walls, no rock). It might also be a good idea never to generate monsters in sight of the player, because the sudden appearance might be confusing.

### 5.2 Monster movement

Monsters should move. Every monster gets a turn per move of the player. Monster moves are restricted in the same way as player moves, i.e., they cannot move into obstacles like walls or rock.

You should implement some algorithms that the monsters can use in order to find the player. Not all monsters have to use the same algorithm, but the following features should be available for generated monsters to select from.

**Random** The simplest way to have a monster move is at random.

**Sight** If a monster can see the player (as an approximation, you can say that this is the case when the player can see the monster), the monster should move toward the player.

---

<sup>3</sup>This would be more confusing than helpful, because monsters can move. An exception might be a graphical UI where you have a good way to distinguish remembered monsters from seen monsters.

**Smell** The player leaves a trail when moving toward the dungeon. For a certain timespan (100–200 moves), it should be possible for certain monsters to detect that a player has been at a certain field. Once a monster is following a trail, it should move to the neighboring field where the player has most recently visited.

**Noise** The player makes noise. If the distance between the player and the monster is small enough, the monster can hear the player and moves into the approximate direction of the player.

**More** Flesh out the above algorithms. For instance, try to have monsters surround small obstacles automatically, switch between senses depending on whether a monster can see, smell or hear the player, or make the random movement less random by remembering the direction the monster last moved in, and not moving into the opposite direction during the next move.

### 5.3 Combat

When the player moves into a monster, or a monster moves into the player, combat occurs. You should invent at least a minimal system that keeps track of each monster's and the player's health. Whenever combat occurs, the attacked party loses some health.

If the player dies, the game ends.

If a monster dies, it is removed from the map.

Monsters should not normally fight each other (even though in the demo game, they do).

### 5.4 Messages

Combat requires information to be shown to the user. The player should be informed if he is attacked or attacking a monster, and what the outcome of the battle is. Make sure that only information is displayed that is relevant to the player.<sup>4</sup>

## 6 Extensions

Here, a few possibilities to extend the game beyond the minimal requirements are listed. The more, the better. These are just ideas. Feel free to design your game differently or to implement an extension that is not in this list.

---

<sup>4</sup>If you share code between monsters and the player, there is a danger that messages for the monster are displayed as a message to the player.

## 6.1 Items

Have random items in the dungeon levels (or possibly, even a few non-random items). Treasure, magic items, weapons or tools are all options. Implement the ability to pickup or drop items, and keep track of the inventory of items carried by the player.

Allowing to use items in various ways adds a lot of depth to the game.

Can monsters make use of items as well?

## 6.2 Score

Award the player points for things achieved in the game (survival, beating monsters, reaching new levels, finding treasure, ...). Implement a high-score list where the best scores of each player are stored. Of course, you should ask for the name of the player at one point.

Typically high-score lists of roguelike games list the cause of death.

## 6.3 Doors and other dungeon features

Add doors that can be open or closed. Doors might also be secret so that you have to search for them before being able to use them.

Other terrain types are possible in dungeons: you can have (movable) boulders (that might block sight), water, lava and more ...

## 6.4 Graphical user interface

There are lots of options in this area. You can go for a tile-based graphical interface that draws little pictures rather than ASCII graphics. You could even animate the tiles.

You can also go for a first-person 3D interface.

Or you can just spice up the text-based user interface, by adding status information or displaying messages in message windows.

## 6.5 Proper combat system

Add chance as a factor to combat. Have different monsters and the player do different amounts of damage, and have a different chance to inflict damage on the opponent. If you have weapons and armor in the game, make it possible to equip a better weapon in order to inflict more damage, or to wear a better armor in order to gain more protection from attacks.

What about ranged combat? You might allow bows and arrows in your game, that can be shot over a distance. Or you could have dragons that can breathe fire from far away ...

## **6.6 Magic effects**

Whether magic or not, a lot of effects can be implemented: fireballs or other combat help, but also the possibility to gain information: find out the complete map, or discover the whereabouts of items and valuables. Have items that trigger those effects, or allow the player to cast a limited number of spells in a certain amount of time (i.e., turns).

## **6.7 Larger levels**

There is no need to limit the level size to the size of the screen. You can make levels that are very large, and use scrolling (even in text mode) to show the portion of the level where the player currently is.

## **6.8 Improved level generation**

Experiment with different level generation algorithms that add dungeon levels with different characteristics: mazes, caverns, cities, strangely shaped rooms – even wilderness or underwater levels are possible ...

## **6.9 Food**

Have the player need food to survive. Food can be found in the dungeon, or possibly dead monsters can be eaten. If the player does not get enough food, the player loses health or simply starves to death.

## **6.10 Speed**

Different monsters have different speed, and can thus make moves more often or less often than the player. The player can speed up and down through items or other means.

## **6.11 Story**

A story adds flavour to the game. Have a background story and a goal to achieve. Add all sorts of tasks and subtasks the player has to solve. Quests may require certain non-random items or rooms to be placed in the dungeon.

# **7 Advice**

## **7.1 Keep IO to a minimum**

Only put code into the IO monad that really has to be. Separate out the user interface from the game logic. Many parts of the program need random numbers. Write your own monad for random numbers if you want an abstraction, but don't use the IO-variants of the functions.

Not using IO much also makes your program easier to test.

## 7.2 Find the “right” level of abstraction

Don’t generalize too much in the beginning, because it will slow you down. But also try to think about possible generalizations and how difficult they would be with your code, so that you can keep the amount of rewrites needed low. Document design decisions so that you and your teammates will remember them yourself later on.

## 7.3 Use the right data structures

Don’t use functional (i.e., immutable) arrays! This is a performance killer. There are at least two suitable representations for level data: a finite map from locations to tiles (*Data.Map*) or *mutable* arrays (*Data.Array.ST* or *Data.Array.IO*). The latter are potentially more efficient, but also have the disadvantage of at least forcing monadic structure of even IO onto large parts of your program.

## 7.4 Intersperse development with refactoring phases

Whenever you’ve successfully implemented a feature, sit back for a moment and think about how to clean up and restructure your code – *before* you start on the next feature. Trying to restructure or clean up everything at the end may be too late.

## 7.5 Add debugging features

Add a lot of debugging features to your game. Different ways to display things and the possibility of showing meta-information can help both you and me to debug your game. Definitely make sure that the field-of-vision algorithm, the smell, and the complete level plus locations of monsters can be visualized.