Documentation

- Name: Thanakorn Phuttharaksa
- Neptun code: B9XP3X

Overview

The videogame presented belongs to the factory building genre, similar to Factorio, Shapez, and Mindustry. It features three types of building; extractors, conveyor belts, and resource depots. The main aesthetic of the video game is the VIM-like command interface and a graphic that takes heavy inspiration from blueprints.

The game features general saving and loading along with resource tracking. However, in its current state, the gameplay loop is not compelling as there is nothing for the player to work towards, so it works better as a proof of concept rather a complete videogame.

How to play

Movement

Use arrow keys to move around the board

Commands

Press Enter to highlight command input. In this move, arrow keys will move the caret on the command input field, so to exit, press Escape. To execute a command, press Enter again while having the command input field highlighted.

The accepted commands are

Starting a new game

```
new <saveName>
n   <saveName>
```

Creates a new save file with the specified name. If a save file with the same name already exists, the new one takes precedence. Starting a new game also remove all subsequent commands, so the command queue will always be empty at the start of a new save.

Saving a save file

```
save [saveName]
s    [saveName]
```

Saves the current state of the game to a new game file, or override the game file with the same name, if one already exists. The save name is optional, and, when omitted, will save will go to the default save file instead.

Loading a save file

```
load [saveName]
l    [saveName]
```

Loads a save file with specified name. If the no such save file exist, does nothing. The save name is optional, and, when omitted, the default save file is loaded instead. Additionally, loading a save file will remove every command in the command queue.

Pausing the game

```
pause
p
```

Pauses the game state, so entities will no longer tick. This does not affect camera movement and commands. This command does nothing if the game is already paused.

Resuming the game

```
resume
r
```

Resumes the game state. This does nothing if the game is not paused.

Placing an entity

```
mk ex <x> <y>
mk dp <x> <y>
mk cb <x> <y> <{n,s,w,e}>
```

Places one of the three possible entity onto the board. This does nothing if the syntax of the command is incorrect or incomplete. The coordinate follow the absolute coordinate of the board. If the coordinate is invalid, this does nothing.

- **ex** syntax for placing an extractor,
- **dp** syntax for placing a depot, and

- **cb** syntax for placing a conveyor belt facing the specified directon.

Placing an entity also consumes resources, and if there is not enough resources, this does nothing. Extractors cost 25 resources to place. Depots cost 50 resources to place. Conveyor belts cost 5 resources to place.

Additional, placing an entity on an occupied tile will override the previous entity, but this process does not refund sources spent.

Removing an entity

```
rm <x> <y>
```

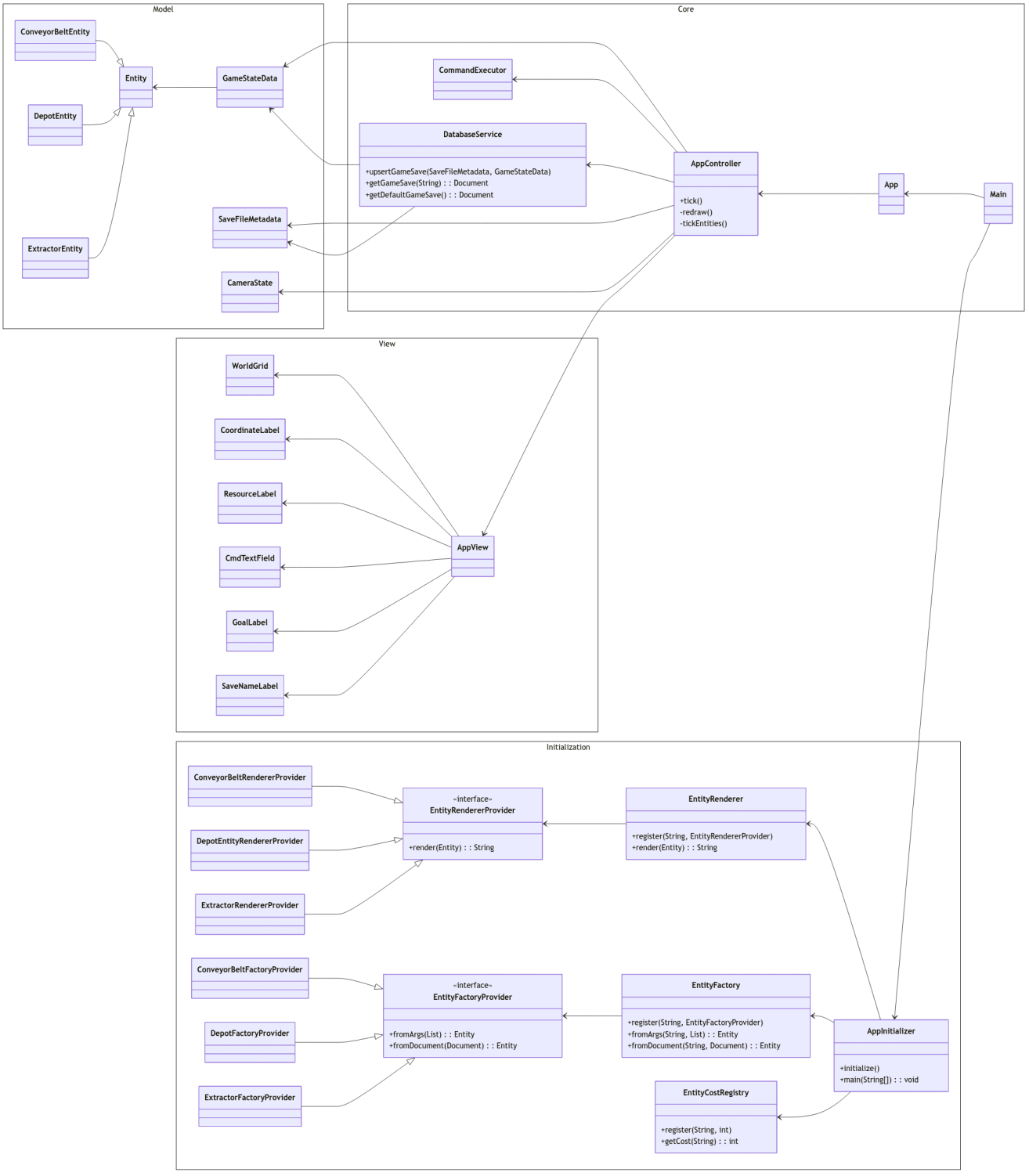
Removes an entity at the specified coordinate. If the coordinate is invalid, this does nothing. If the syntax is invalid, this also does nothing. If the tile is unoccupied, this does nothing. If the tile is occupied, this does not refund resources spent.

Entity placement short key

Alternatively, use the short keys to quickly place entities. Entities placed this way will be placed onto the same tile as where the cursor is.

- Press **1** to place an extractor.
- Press **2** follow by either **1**, **2**, **3**, or **4** to place a conveyor belt. Each number correspond to
  - **1**: North
  - **2**: East
  - **3**: South
  - **4**: West
- Press **3** to place a depot.

UML Diagram



Project Structure

The project is split into two major packages; the server and the client. The motivation here is to implement a structure similar to Minecraft. That is, we have a server which handles all of the logical part of the video game, as well as the progression of the game state, but it is not responsible for handling and rendering.

We provide the current state of the server to the client periodically. The client takes this information and use it to render and update the interface accordingly.

The player issues command from the client through the public API provided the server. The server places these instructions into a queue and execute one of them in each tick. Ideally, we need two separate threads on the server; one for progressing the game state, and one for handling the instructions.

Initialization

In main, we start by initializing the registries. There are three main registries.

EntityFactory tracks key-value pair of string and implementation **EntityFactoryProvider**. This registry provides a bus for getting a builder for each implementation of **Entity**. More specifically, we want to provide two ways to construct an **Entity**; from a BSON document and from a list of arguments.

EntityRenderer tracks key-value of string and implementation **EntityRendererProvider**. Similarly, this registry tracks how to render each implementation of **Entity**.

EntityCostRegistry tracks how much each **Entity** cost to deploy.

Core Application

After initialization, we construct the application. The exposed application is simply a frame with custom cleanup handler. In **App**, we construct a **AppController**. The **App** is also responsible for starting and stopping the logic thread using **ExecutorService** with a **schduleWithFixedDelay**.

This logic thread progress the game state by invoking **AppController::tick**. In each tick, we execute one pending command, then we advance the tick of each **Entity**, then we re-render and repaint the view.

Controller

The controller keeps track of three model.

- **CameraState** keeps track of the camera position.
- **GameStateData** keeps track of the entities, resources and other logic-relation data.
- **SaveFileMetadata** keeps track of save name and save file-related data.

The controller also implements **KeyAdapters** to connect **AppView** to the models.

The controller also uses **DatabaseService** and **CommandExecutor**. The **CommandExecutor** is straightforward and it handles the queuing of commands and their execution.

Tests

White-box Testing

On `ConveyorBeltEntity` Class

1. `testInitializationOrientation`
   - **Test**: Verifies that the entity initializes with the correct orientation.
   - **Action**: Check the orientation property of the conveyor.
   - **Expected**: The orientation is `NORTH`.
2. `testInitializationBufferState`
   - **Test**: Ensures the buffer state is initialized properly.
   - **Action**: Access the buffer state list.
   - **Expected**: The buffer state list is not `null`.
3. `testReceiveItemLastBuffer`
   - **Test**: Checks the behavior of `receiveItem` when adding an item to the buffer.
   - **Action**: Call `receiveItem` and check the last position in the buffer.
   - **Expected**: The last position in the buffer is `true`.
4. `testCanReceiveItemFromNeighbor`
   - **Test**: Verifies that the conveyor can receive items from a neighboring entity.
   - **Action**: Call `canReceiveItemFrom` with a neighboring entity at `(0, 1)`.
   - **Expected**: Returns `true`.

On `DepotEntity` Class

1. `testCanReceiveItemFromAnywhere`
   - **Test**: Confirms that a depot can receive items from any sender.
   - **Action**: Call `canReceiveItemFrom` with a `null` sender.
   - **Expected**: Returns `true`.
2. `testTickAddsResources`
   - **Test**: Verifies that calling `tick` transfers items to the context's resources.
   - **Action**: Call `receiveItem` followed by `tick`.
   - **Expected**: The context's resource count is incremented by `1`.

On `ExtractorEntity` Class

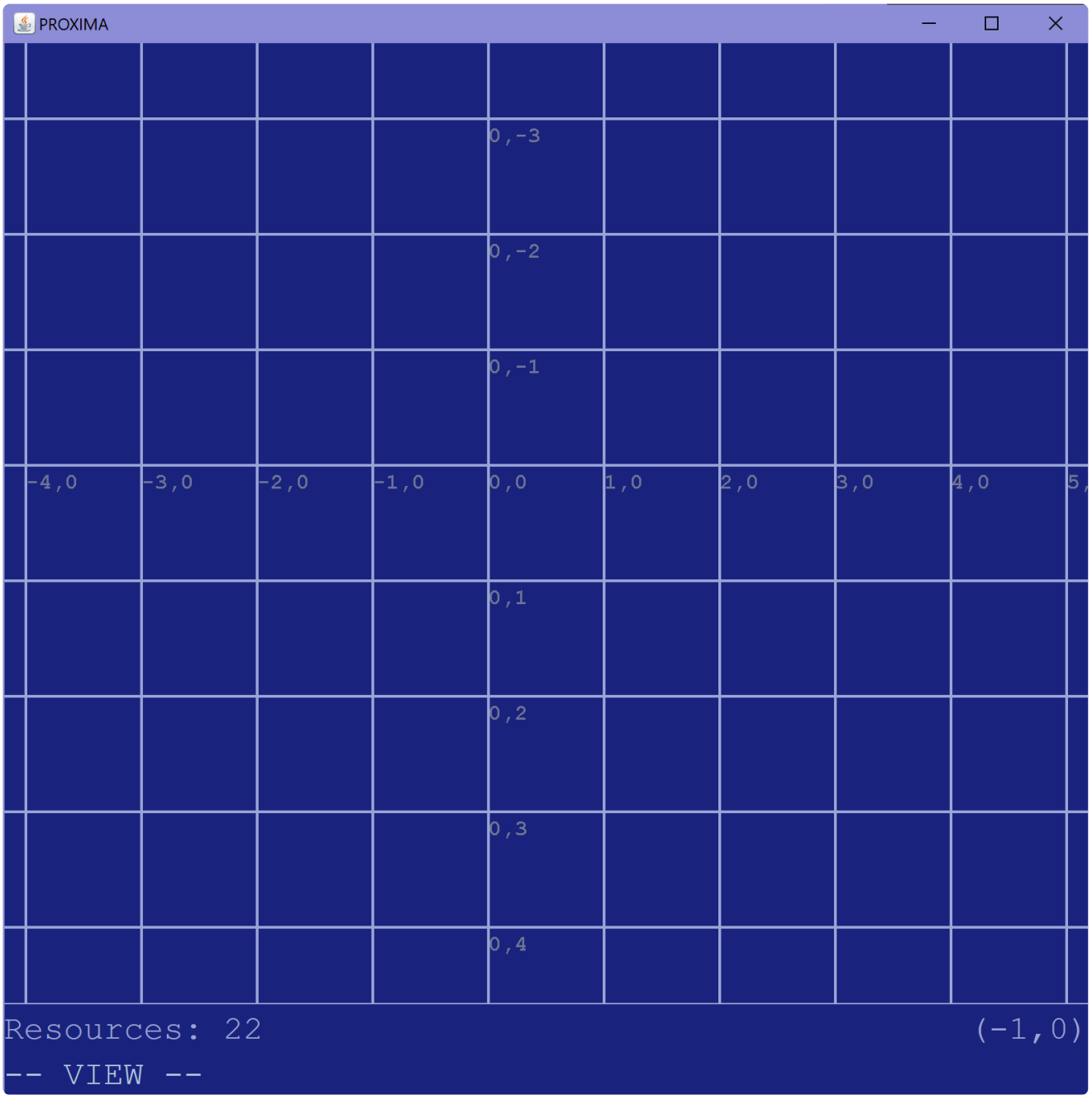
1. `testInitializationTicksToGenItem`
   - **Test**: Confirms the initial value of `ticksToGenItem`.
   - **Action**: Access the `ticksToGenItem` property.
   - **Expected**: The value is `40`.
2. `testInitializationPositionX`
   - **Test**: Verifies that the X-coordinate is initialized correctly.

- **Action**: Access the `positionX` property.
- **Expected**: The value is `0`.

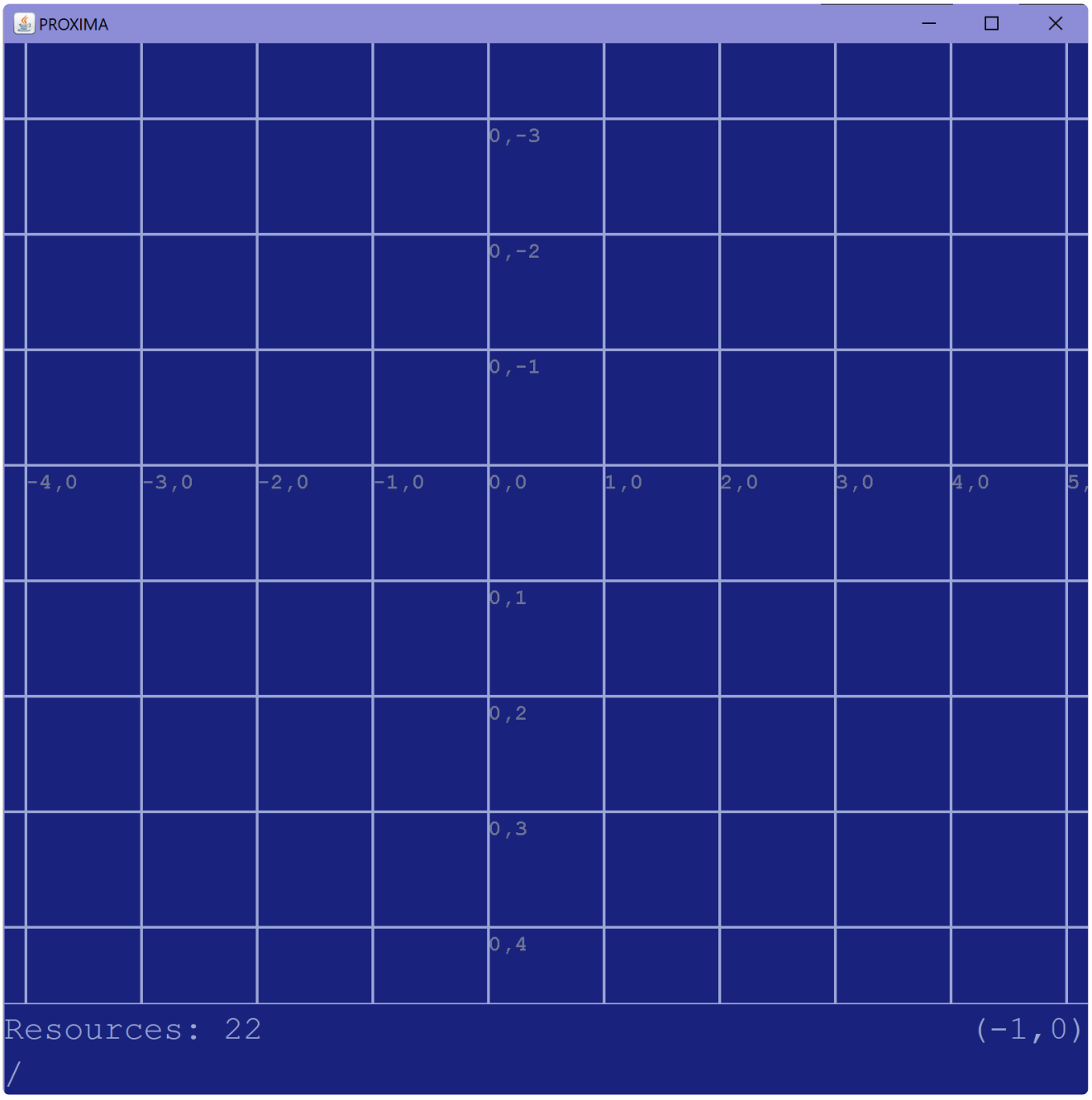
3. `testInitializationPositionY`
   - **Test**: Verifies that the Y-coordinate is initialized correctly.
   - **Action**: Access the `positionY` property.
   - **Expected**: The value is `0`.
4. `testTickDecrementsTicksUntilNextItemGen`
   - **Test**: Ensures that ticking the extractor decrements `ticksUntilNextItemGen`.
   - **Action**: Call `tick` on the extractor.
   - **Expected**: `ticksUntilNextItemGen` is decremented by `1`.
5. `testCannotReceiveItemFrom`
   - **Test**: Verifies that the extractor cannot receive items from any sender.
   - **Action**: Call `canReceiveItemFrom` with a `null` sender.
   - **Expected**: Returns `false`.

Black-box test

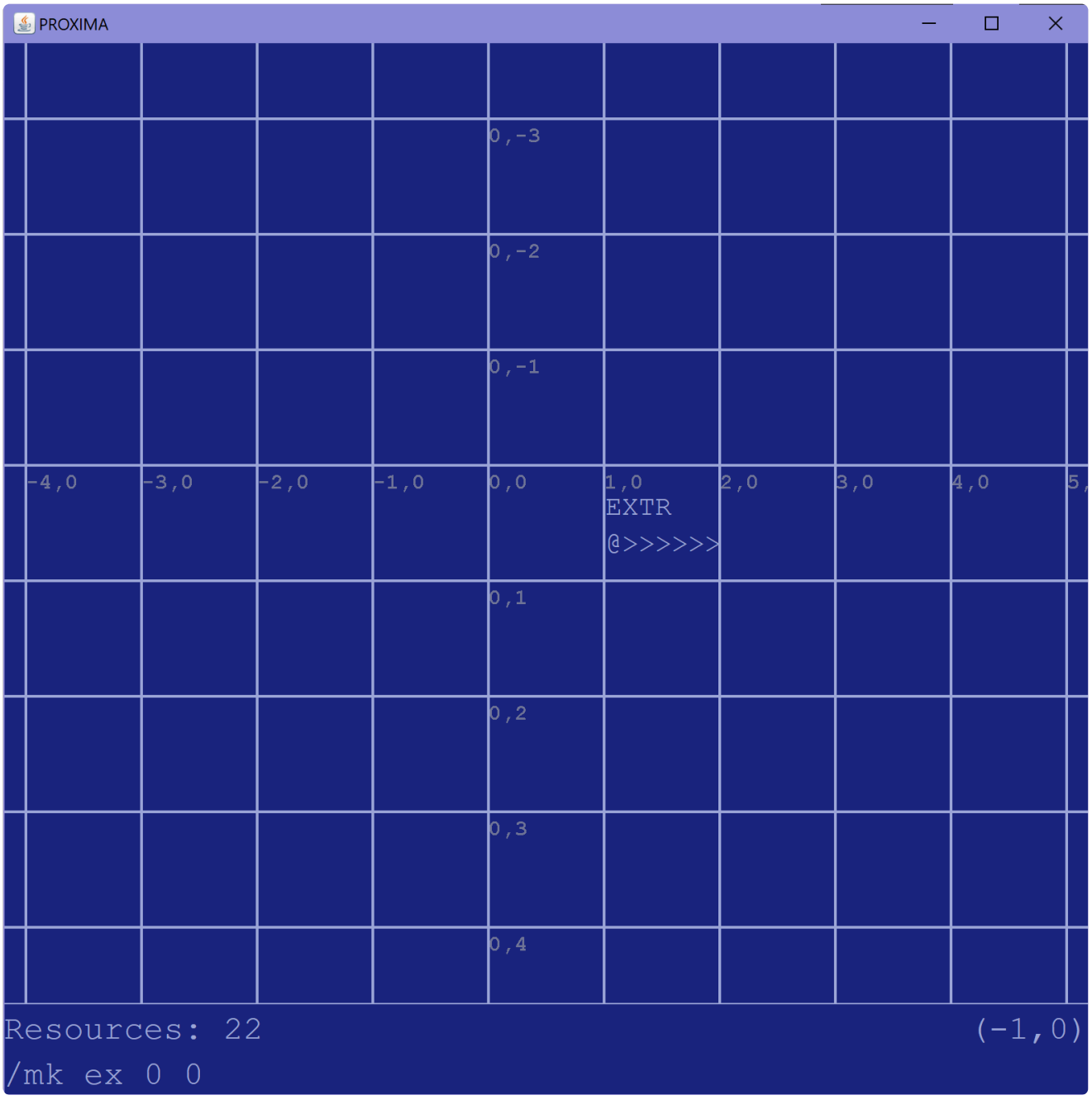
Move around the grid using arrow keys or **HJKL** keys.

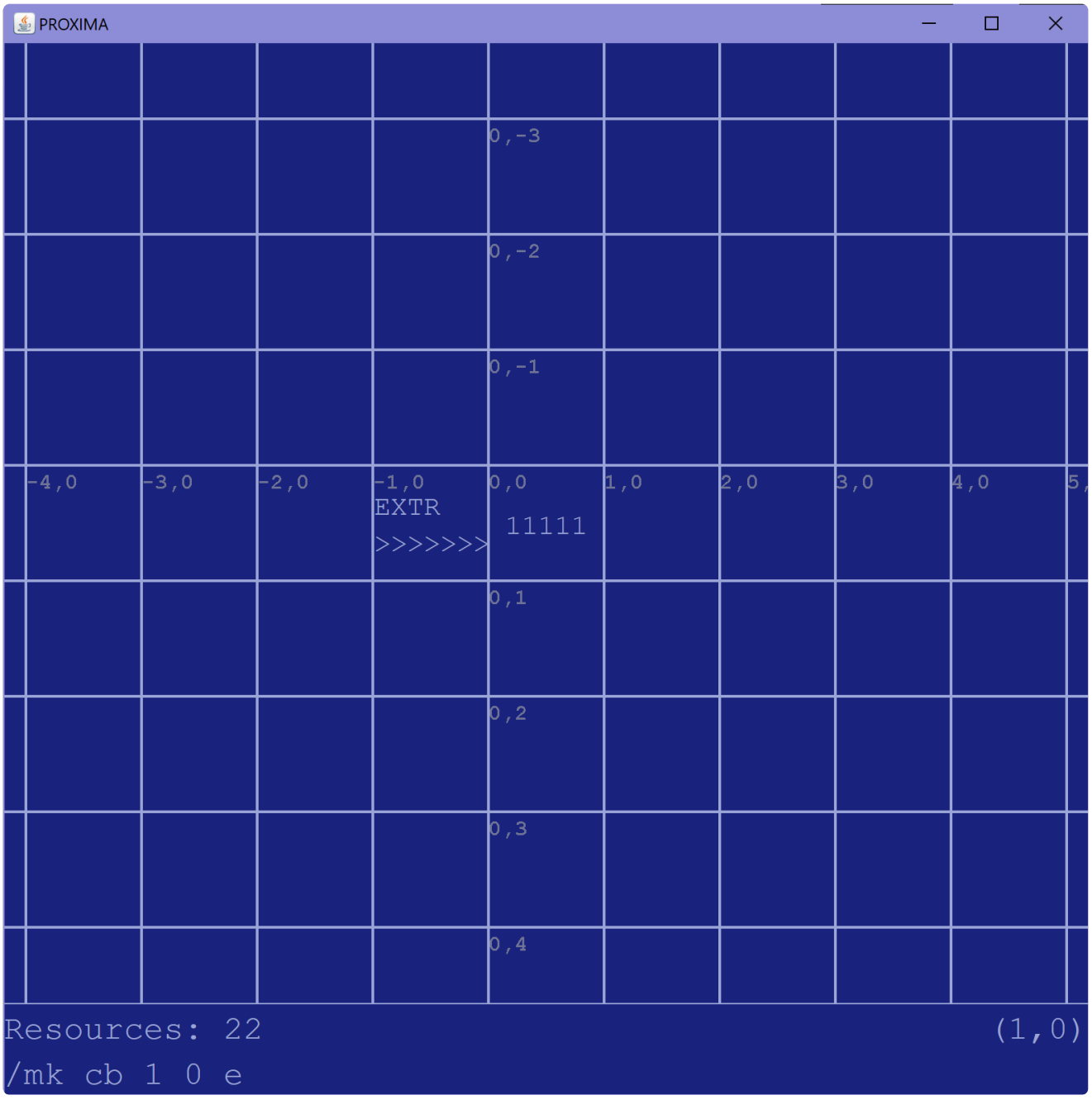


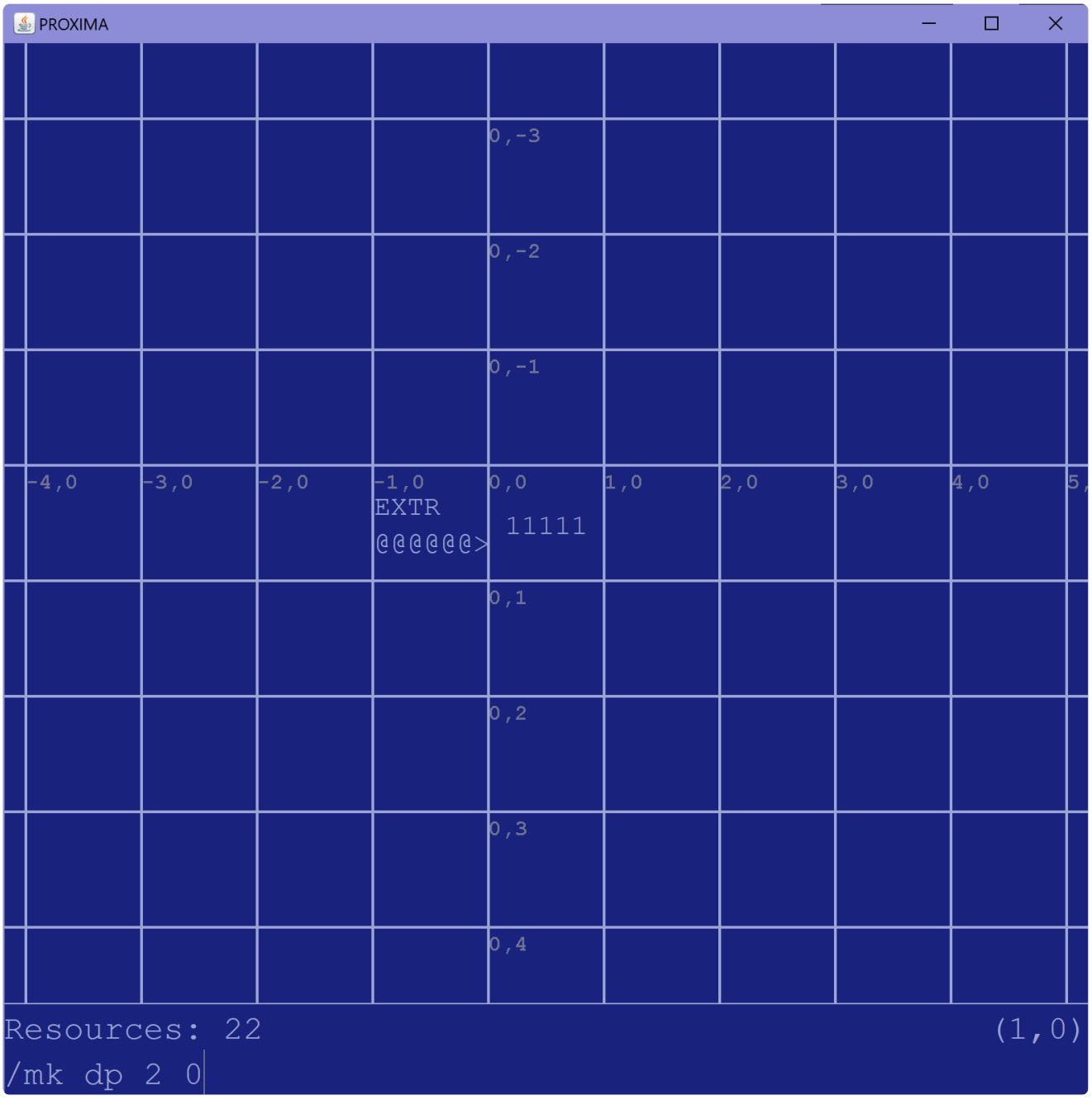
Press the forward slash key to switch to command mode.



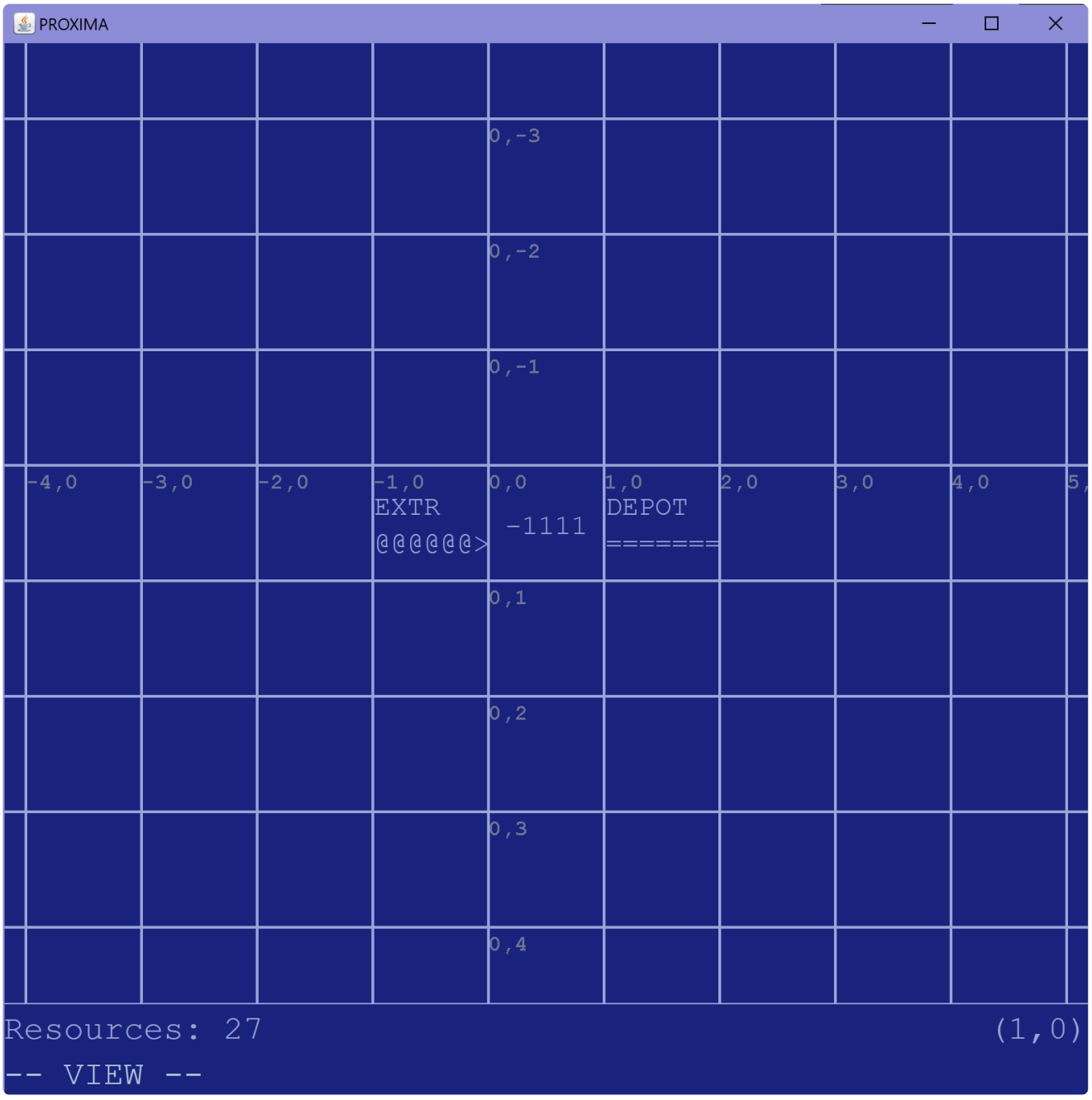
Enter **mk** instructions to place buildings.


PROXIMA
0,-3
0,-2
0,-1
-4,0
-3,0
-2,0
-1,0
0,0
1,0
EXTR
@>>>>>>
2,0
3,0
4,0
0,1
0,2
0,3
0,4
Resources: 22
(-1,0)
/mk ex 0 0

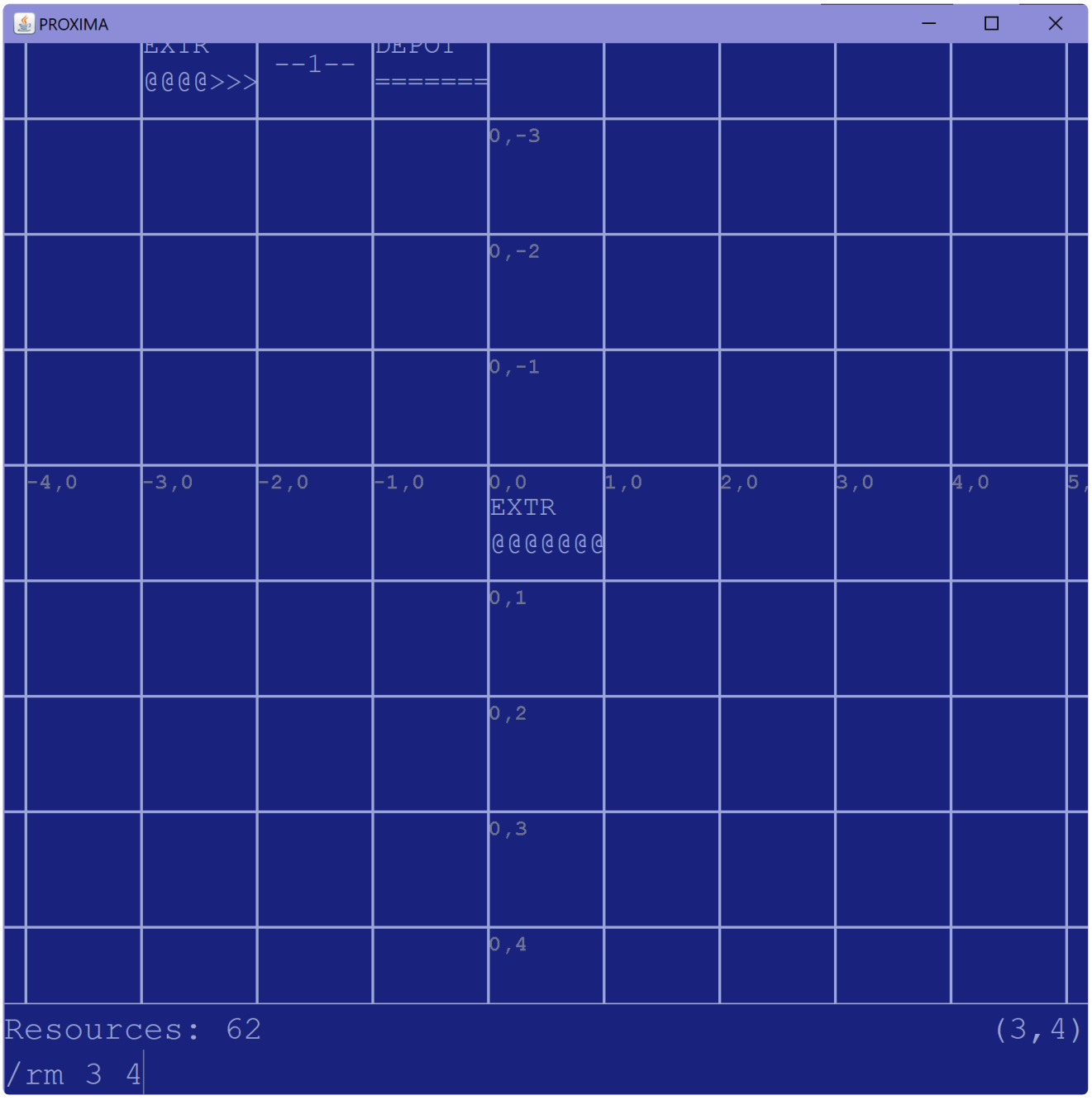

PROXIMA
0,-3
0,-2
0,-1
-4,0
-3,0
-2,0
-1,0
EXTR
>>>>>>>
0,0
11111
1,0
2,0
3,0
4,0
0,1
0,2
0,3
0,4
Resources: 22
(1,0)
/mk cb 1 0 e

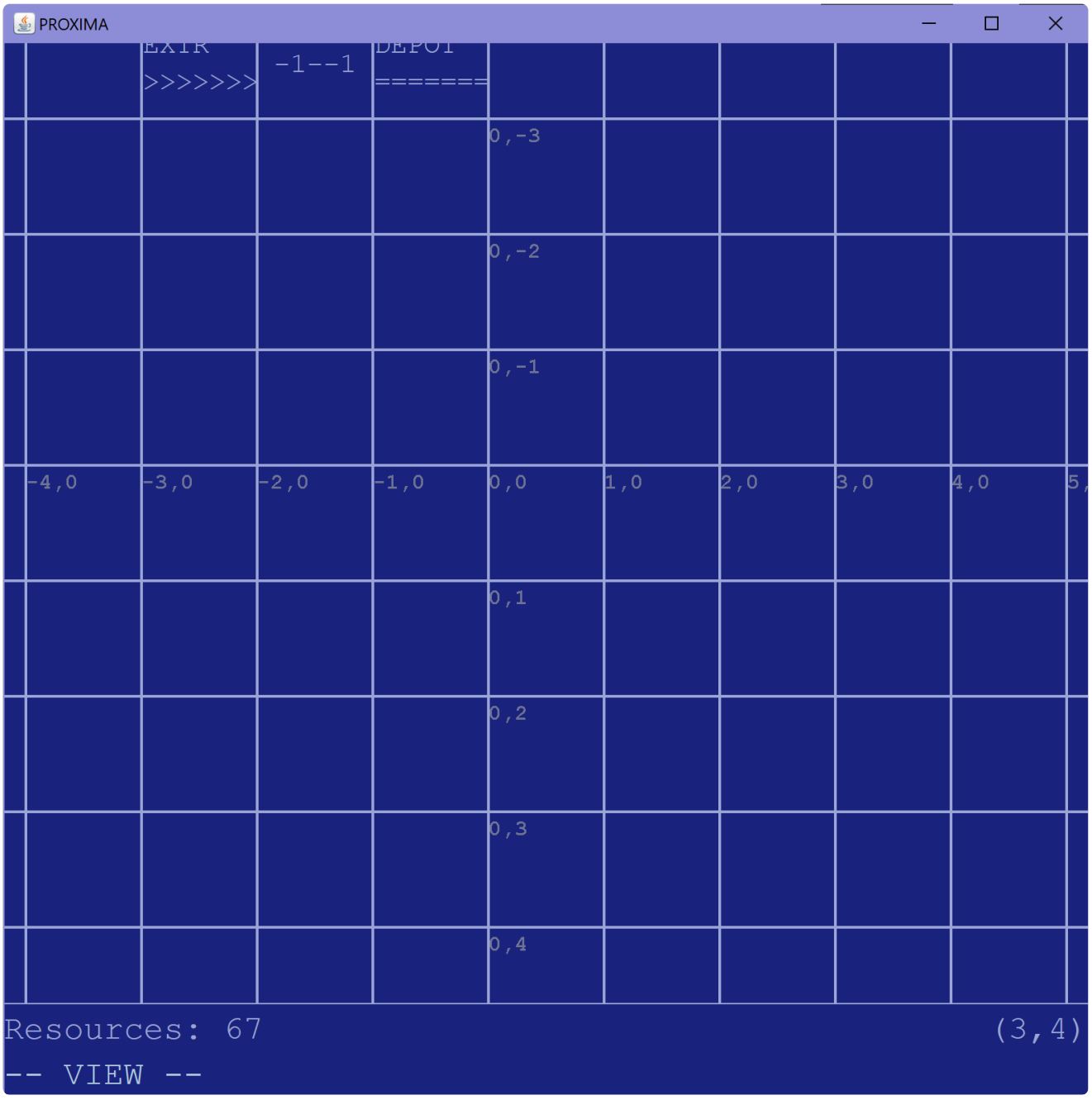


We have the following minimal factory;

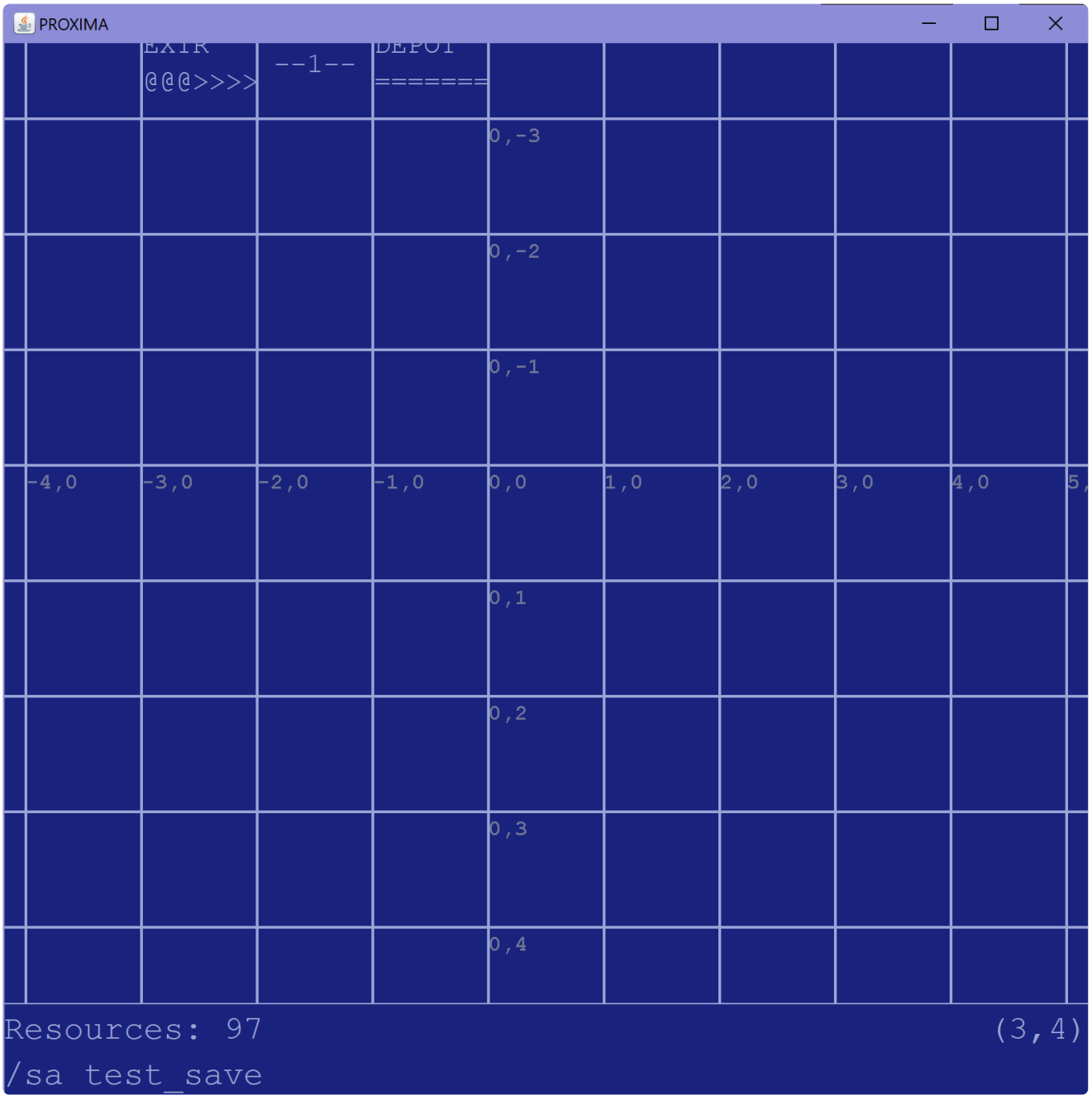


Enter **rm** instruction to remove buildings;

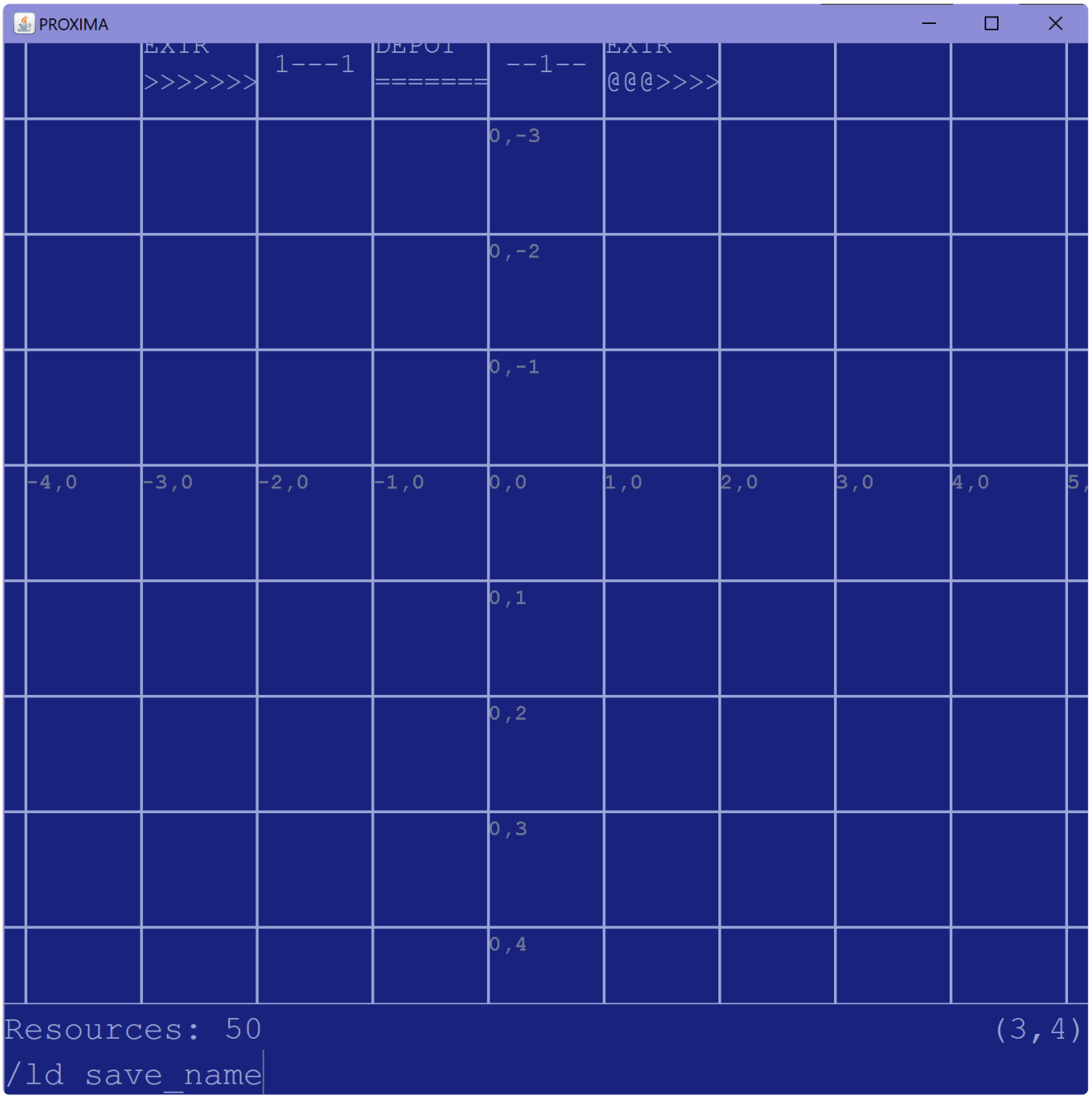

PROXIMA
EXTR
@@@@>>>
--1--
DEPOT
=======
0,-3
0,-2
0,-1
-4,0
-3,0
-2,0
-1,0
0,0
EXTR
@@@@@@@
1,0
2,0
3,0
4,0
0,1
0,2
0,3
0,4
Resources: 62
(3,4)
/rm 3 4



Enter **sa** instruction to save the current state of the game to the given save name.



Enter **ld** instruction to load the game state from a save file.


PROXIMA
EXIR
>>>>>>>
1---1
DEPOT
=======
--1--
EXIR
@@@>>>>
0,-3
0,-2
0,-1
-4,0
-3,0
-2,0
-1,0
0,0
1,0
2,0
3,0
4,0
0,1
0,2
0,3
0,4
Resources: 50
(3,4)
/ld save_name