Alexis Theas-Laban & Quentin Ennouchi

L3 - IGM | 2020-2021

BABA IS YOU

Developer's guide

Table of contents

Introduction

Everything you need to know if you want to modify the game or simply understand how it works.

To generate the javadoc use :

```
ant doc
```

Architecture

Design decisions

Representing game elements

We first decided to represent game elements with an inheritance system. An abstract class **Block** would hold global information like the position of the object or his current active property. And subclasses like **Operator** and **Property** would contain methods to affect the level game state.

However, the rule structure would also contain these elements and multiple methods which would have been unrelated to the class itself. A block used as a rule element does not need a position.

Because the elements in the rules and in the levels have different purposes, we decided to separate the two. Instead, we choose to implement an enum identification system; blocks are empty vessels who can receive data from a BlockID enum value.

This approach is especially useful in a game where a lot of elements have to be added. Expansion and correction are simplified as all of the elements are declared in a single place.

Recognizing and parsing rules

We iterate over the level game elements and put them in a double sized array of **Text** representing the textual game elements. Taking the lines and columns of the array, we put them in a list of arrays of text that we defined as **Rule**. The rules are then sent to the parser which proceeds to update the blocks properties.

Because the operators have precedences, the rules are parsed by going through all the operators in a rule and ”collapsing it”. We iterate over the game’s operators in the order of their enum value, checking in every rule if the operator is present. If so we remove the left, right and current text elements and replace it with the result of the operator operation.

Applying properties

Property must be applied after the rules have been parsed and in a specific order. Furthermore we need a way to indicate if a block has a certain property.

Each block has a set of properties to indicate its current active property.

Packages

block : contains the basic game elements, a block is defined by an identifier, an image, a position, a set of property and a boolean to indicate if the block is dead.

rules : This package contains all the classes related to rule recognition and updating.

level : classes related to level manipulation. A level contains the game elements and the current rules.

utils : system related, include input, rendering and file saving classes.

main : Entry point of the application. Contains a method for arguments parsing.

Implementation

Block identification

A block is primarily defined by its identifier (see **BlockID**), an enum value which also contains additional information based on the type of the object.

- The **properties** identifiers contain an effect object which is used to apply a property to a specified block.
- The **operators** identifiers contain an operation object. The operation defines a bifunction method used in the rule parsing package.
- The **nouns** identifiers contain a real object identifier, which is the non-textual counterpart of this object.
- The **group** identifiers represent a groupment of multiple objects. Each group identifier must include a predicate to filter the member of the group from the other identifiers.
- The **real object** identifiers don't need to contain any information.

The image file path is obtained by transforming the enum to a string and adding a “.gif” extension. The order of declaration in the real object enum indicates the priority in the rendering layer. For example the identifier **BABA** being declared at the top, any other real objects is drawn below the baba elements.

Movements

Specifying a block and a direction, the **moveInContext** method in **Level.BlockMover** determines if the movement is valid. If that's the case every block containing the push property is moved in the same direction.

Testing valid movements

0	1	2	3	4

Moving right example

Firstly checking if the next position (at 2) is in the level dimension. Then, checking if the location is empty, if the position is occupied we check if the occupying block has the **STOP** property or if it does not have the **PUSH** property. If any of the above conditions is false we test if the block (at 2) can move in the same direction as baba (at 3). The process is repeated for the next block until there are no more pushable blocks left. Stopping at 4, the process returns true because the location is empty. Thus baba can move to position 2.

Rule recognition

The level board is mapped into a two dimensional array of Text elements. If an element is an instance of the Text interface he can be added to the array. The lines and columns of the array are then sent to the rule parser.

BABA	AND	WALL	IS	YOU

Parsing the above rule :

1. Each rule containing a AND operator can be split into a set of rules with no AND operator. Splitting the rule and removing the AND operator, we get the rules **BABA IS YOU** and **WALL IS YOU**.
2. Checking the second operator, we remove the left and right elements around the IS operator forwarding the elements to the operator operation method. The result of the operation updates the properties of the blocks in the level and returns the right element. The operator is replaced by the result of his operation.
3. Because there are no more operators left we stop the parsing.

Effects

Each property can contain an effect object. An effect object implements an apply method which takes a block and modifies the current level or the block according to the set condition.

For example, the win property contains a win effect with the following structure :

-Take all the blocks at the same position that the block specified in the argument.

-If any of them is the player (has a YOU property), set the level to finished.

The different effects are applied based on their order in the enum Property.

Level file format

The levels of the game are stored in a csv file format. This format has been retained for its rapidity upon creation. All the levels of this game were created in an excel like program, this allows for rapid level design and facilitates the adding of custom levels.

Each block is associated with an identifier, as defined in the block identification section. The real objects are marked with a “OBJ_” prefix.

The first line must declare the dimensions of the level (columns x lines).

```
5,5,,,,
,,,,,,
,,BABA,,,,
,,IS,,OBJ_BABA,,
,,YOU,,,,,,
,,,,,
```

5	**5**			
	BABA			
	IS		**OBJ_BABA**	
	YOU			

Example of a valid level file format

Exclusive content

New property (REAL) : All the nouns with this property are transformed to their real counterpart.

New noun (NOUN) : designates all the noun text blocks.

Bonus level : a bonus level that uses the previously mentioned blocks. Played last by default.

Enhancements

Additional operators

On : filter the left operand if the corresponding right and left blocks are at the same position.

Has : create a new block when the corresponding left block is dead.

And : allow joining of multiple rules. This operator is incomplete, BABA IS YOU AND IS STOP is not a valid rule as it should be.

Saving and loading

Text format :

```
bonus-level.csv
OBJ_BABA 15 12 false
OBJ_WALL 11 11 true
OBJ_WALL 12 11 true
OBJ_WALL 13 11 true
OBJ_WALL 14 11 true
```

Example of a save file format

The save file always starts with the name of the level from which the save was made. From left to right, the elements identifier, the position, and a boolean to indicate if the element is dead.

Conclusion

Task distribution

Quentin was charged with the file saving and loading. He also handled the level formatting and the arguments parsing. Alexis was in charge of the main game features including rule recognition, blocks movement and identification.

The bonus level and the exclusive blocks were designed conjointly.

Difficulties encountered

The architecture of the game was hard to decide; multiple choices were possible and each of them included different kinds of advantages.