Milestone1 by Group-Alpha

0. Introduction

The group members: Botu Lyu, Jamila Oubenali, Zhiyao Yan.

In this document you will find different information separated into five sections;

- A presentation of what our project is about.
- A list of our data structures, containing our main classes, how they relate to each other, and drafts of methods.
- Some ideas for implementation.
- Our plans in the next weeks and the problem to solve.
- Meeting reports documenting our progress and the issues encountered.1. Presenting our project

We will now introduce the game we have chosen to replicate. If you are already familiar with it, there might not be a real use of reading this section. Then, we will talk about why we chose to do this and what our main expectations of it are.

1.1. “Baba is You”

You may have heard of, or played the quirky looking game “Baba is You”.

It is a fun, simplistic looking game based on boolean logic that has the player solving puzzles. The player starts by playing as “Baba”, a cute small white character.

The game’s cover

This is a 2D game where only basic “up, down, left, right” movements are possible. The player has to push various elements on the map, some being objects, some being blocks of text, to manage to get into a “win condition”. Blocks can be combined to create sentences that will change the game’s rules, such as “Flag is Win”, which will make touching (or *being*) a flag a win condition.

While the game starts off quite easy, it quickly becomes a real challenge to solve puzzles, and can hardly be done without carefully thinking through every step. In the beginning, we only have a few object types (wall, water, rock, baba …) and limited connects (mainly “is”). More of them are unlocked going through the levels, including the basic boolean “and”, “or”, which will sometimes create very non-intuitive situations. Overall, this is a great game to introduce anyone of any age to logic, or to “workout” your brain on some difficult puzzles.

A level of the game

There is also a feature where users can create or modify levels. Baba is You has a great community of level designers, which makes running out of puzzles to solve impossible. The game can be found on steam as well as on other platforms: https://store.steampowered.com/app/736260/Baba_Is_You/

1.2. Why we chose this game

“Baba is You” is a great reminder that you don’t need unattainable resources and a lot of money to make a great game; its graphics can be created using jtamaro, the overall game can be implemented in a simple and elegant fashion, though the rules can quickly become a hell to sort out.

We found this game so creative and fun to play that we wanted to recreate it, maybe adding some of our own ideas into it. However, this will only become real after we have managed to implement the basic aspects of it.

The boolean logic and rule handling felt like a real challenge, one we would have certainly underestimated without the TA’s warnings. This is why we have decided to first start with a small set of objects and with only one connector (”is”). Hopefully, we will be able to add new features and create more complicated puzzles in these following weeks.

1.3. Our expectations

We expect to struggle quite a lot with the rules. Even if we manage to carefully design them, it is quite likely that we will run into many strange situations that might arise only when playing for a while and trying out weird combinations. This is also what makes this game interesting to create; We will have to be creative and rigorous when it comes to testing.

We do not plan on spending a lot of time on the rendering part, as the graphics are not the “interesting part” of the game.

2. Data structures

2.1. Project overviews

```
└src.app
    ├game
    │    ├controller
    │    │    └GameController
    │    ├state
    │    │    ├GameState
    │    │    ├Heading
    │    │    ├Item
    │    │    ├Kind
    │    │    ├Rule
    │    │    └Tile
    │    ├GameRun
    │    └Settings
    ├AppState
    └Main
```

2.2. Design of the model

2.2.1. GameState

A GameState is a **singly linked list**, which contains the current map, the previousState, and the rules showed on the map.

```
public class GameState {
    public ArrayList<ArrayList<Tile>> map;
    public Sequence<Rule> rules;
    public GameState previousState;
}
```

We use class to make it easier to modify the GameState.

`ArrayList<ArrayList<Tile>> map`: we can easily traversal the map and visit specified locations.

`Sequence<Rule> rules`: we will extract the rules on the map to form a sequence. In the GameState, we will process the rules in detail.

`GameState previousState` : return to the previousState when player uses the undo button.

We have already implemented the `render()`. We put a black rectangle on the bottom as a background.

```
public Graphic render() {
        int n = map.size(), m = map.get(0).size();
        Graphic fore = emptyGraphic();
        for (int i = 0; i < n; i++) {
            Graphic rowFore = emptyGraphic();
            for (int j = 0; j < m; j++) {
                rowFore = beside(rowFore,
map.get(i).get(j).toGraphic());
            }
            fore = above(fore, rowFore);
        }
        Graphic back = rectangle(m * Settings.UNIT_WIDTH, n *
Settings.UNIT_HEIGHT, BLACK);
        return overlay(fore, back);
    }
```

2.2.2. Item

An Item is an object. Many items make up the map.

```
public record Item(Kind name, // describes what kind of object it is
                   boolean light, // true if the object is one part
of an effective rule
                   boolean cancel, // true if the item is part of an
invalid rule
```

```
                    boolean stop, // true if the object cannot be
traversed
                    boolean push, // true if the object can be pushed
                    boolean you, // true if that object is controlled
by the player
                    boolean win // true if touching the item triggers
a win
) {
    public Graphic toGraphic() { }
}

public enum Kind {
    BOUNDARY,
    ICON_WALL, TEXT_WALL,
    ICON_BABA, TEXT_BABA,
    ICON_FLAG, TEXT_FLAG,
    ICON_ROCK, TEXT_ROCK,
    TEXT_IS, TEXT_WIN, TEXT_PUSH, TEXT_YOU, TEXT_STOP
}
```

2.2.3. Rule

A Rule is an edge, from one Kind to another Kind.

```
public record Rule(Kind from, Kind to) {
    // Three items can be linked into one rule,
    // but we can ignore the connector text,
    // so two items build a `Rule`
    public Rule getRule(Item first, Item second) {
        return new Rule(first.name(), second.name());
    }
}
```

2.2.4. Tile

A Tile is a sequence of items arranged from top to bottom. Some items in a tile may move individually.

```
public record Tile(Sequence<Item> items) {
    // convert a Tile to a Graphic
    public Graphic toGraphic() {
        return reduce(
            (g, item) -> overlay(g, item.toGraphic()),
            emptyGraphic(),
            items
        );
    }
    // check whether it's possible to step on this Tile
    public boolean canStepOn() { }
}
```

2.2.5. Heading

Heading means the directions.

```
public enum Heading {
    NORTH,
    SOUTH,
    EAST,
    WEST;
    // convert a keycode to direction.
    public static Heading fromKeyCode(int keyCode) {
        return
            keyCode == KeyboardKey.UP ? NORTH :
            keyCode == KeyboardKey.DOWN ? SOUTH :
            keyCode == KeyboardKey.RIGHT ? EAST :
            WEST;
    }
}
```

3. Ideas for implementation

3.1. While moving

1. The direction key is pressed. Traverse the map, if item.you()==true, then try to move it.
2. After the move is finished, traverse the map and record the rules.
3. Traverse the map, and modify the item's properties according to the rules.
4. Determine if a win or lose state appears.
5. Render the GameState.

3.2. Execution Rules

In the playtesting of "Baba is you", we found many interesting rules.

A text item declares an object(e.g. TEXT_WALL), a connector(e.g. TEXT_IS) or a state(e.g. TEXT_PUSH). Let's assume that A, B, and C are all one object.

- "A is B" and "A is C" are in effect at the same time, item A splits into B and C.
- "A is B" and "B is A" are in effect at the same time, A becomes B first and B becomes A after the next move.
- "A is A" has the highest priority, at this time the other "A is B" will be invalid.

We have found a perfect solution for them:

1. After extracting the Sequence rules, we process the "'object' is 'state'" part.
2. Make the "'object' is 'object'" into a hash map: HashMap<Kind, set> buildRules, to remove duplicate edges and centralize the processing of modifications to the same object.
3. Create a blank map
4. For each mapping A->set{B, C,...}, add the corresponding item A in the current map directly to the new map

5. Check if there is "A is A" (if so, then some of the rules are invalid)
6. Apply the rules in the new map.

This way you can avoid changing the same object around.

3.3. Multiple "you" exist

When moving, a series of tile changes may occur. To avoid repeatedly moving “you”, we make special judgments about the direction:

If Heading==WEST, the traversal order is from left to right.

If Heading==EAST, the traversal order is from left to right.

If Heading==NORTH, the traversal order is from top to bottom.

If Heading==SOUTH, the traversal order is from down to up.

3.4. Render walls

Render the wall based on the four adjacent tiles (map[i-1][j], map[i+1][j], map[i][j-1], map[i][j+1]) of the current map[i][j]. We have 2^4 = 16 looks for the walls.

4. Plans

4.1. Challenges

1. Can we change a photo to a graphic? If not, we need to draw the items by ourselves.
2. Create high-cohesion, low-coupling code

4.2. Things to do

1. Complete individual levels: accurate movements and rules implementation
2. Create test levels and menus
3. Create special effects: some onTick content, and deformation when moving

5. Meeting reports

5.1. Meeting 1

Date: 11/05/2022 15:00-17:40

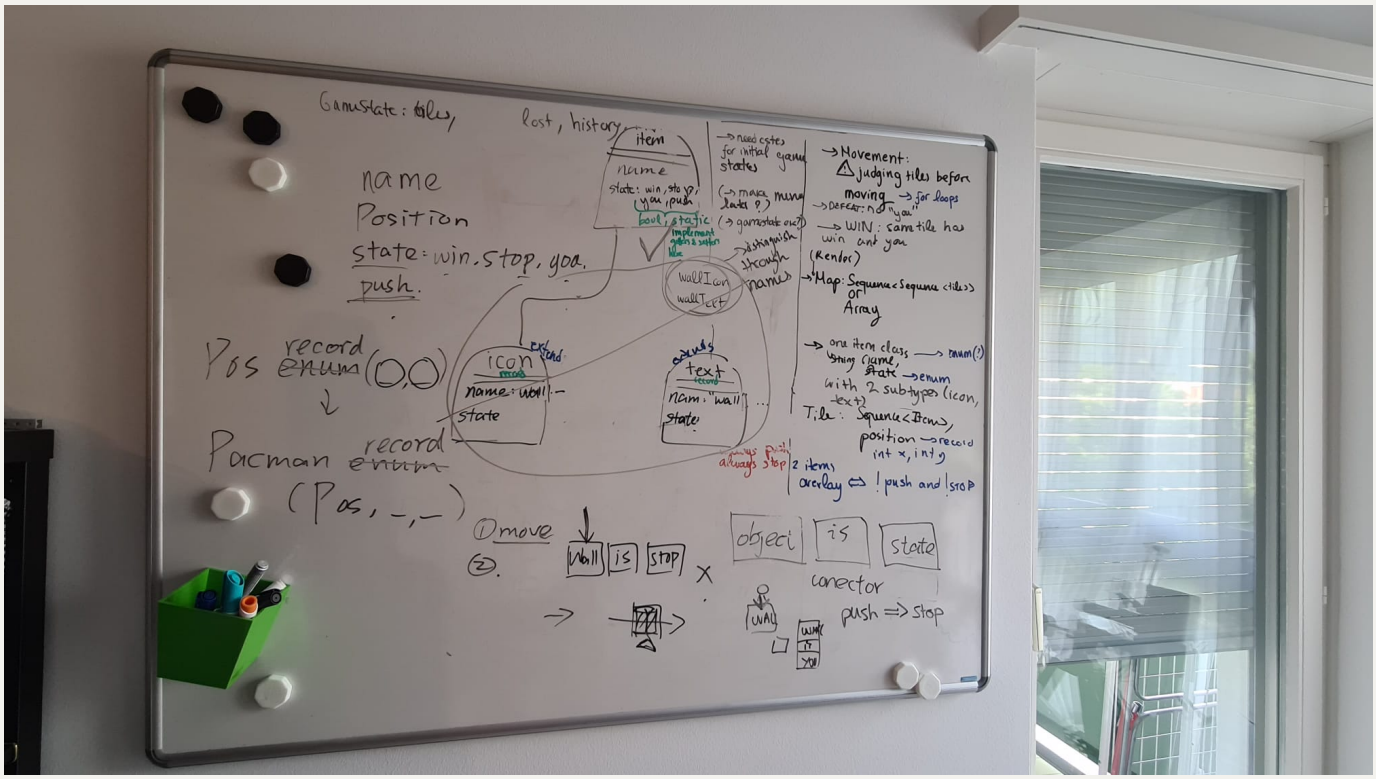
Meeting overview

We have decided to create a game similar to “Baba is you”. In the first place, we will create one level with a basic set of items: wall, Baba, “is”, “you”, “win”, “push”, “stop”.

Then, we will create a GUI that allows user to create their own levels.

Finally, if we have the time and resources, we will create more items such as “defeat”, “float”, water etc…

We have sketched the most basic classes needed, and how they relate to each other.



End picture of our whiteboard where we sketched classes and summarized information.

Goals until next time

The next step is to;

- √ Wait for the git repository to be available.
- √ Play the game to be more comfortable with the rules.
- √ Think about how to build a directed graph for the rules.
- √ Think about how to do “tile checks” to judge whenever one or multiple tiles can be pushed.
- √ Think about how to render the walls.
- √ Test and draw a few scenarios.

General implementation

- General features:
    - The game consists of a top view puzzle focused on boolean logic. The player starts by controlling one character named “Baba” and can interact with elements on the map.

        - We can worry about creating a menu later. It will probably be done through adding to the GameState.
- Map:
    - The map consists of a background and a foreground that is updated whenever the game state changes.
    - The map will possess some boundaries that cannot be crossed. They will always be set to STOP == true (cf sections below for data structures).
- Game state:
    - The gameState probably only needs: tiles, gameLost?, history to allow for undos.
    - We will need to create one constant per level that represents the initial state of the level (to allow for restarts).
    - The win condition is that the same time has win == true and you == true.
- Rendering:
    - To render the game, an array or a Sequence<Sequence> can be used to render and reduce tiles.
    - To distinguish between icons and text, we can probably just use their name. So, there is no need to create subtypes of the type "Item". For example: wallIcon and wallText.
    - Tiles contain Sequence. In this way, it is possible to know which one should be rendered on top if multiple items are on the same tile.

Draft of the Data Structures

Item

An item is an object found on the map, with whom the player can interact. Each item has a name and a set of properties that describe how we can (or cannot) interact with them.

```
public record Item(
  Kind name, // describes what kind of object it is
  Boolean light, //true if the object is one part of an effective
rule
```

```
  Boolean stop, //true if the object cannot be traversed
  Boolean push, // true if the object can be pushed
  Boolean you, // true if that object is controlled by the player
  Boolean win // true if touching the item triggers a win
){}

public enum Kind{
  boundary,
  iconWall, iconBaba, iconFlag, iconRock,
  textWall, textBaba, textFlag, textRock,
  textIs, textWin, textPush, textYou, textStop
}
```

There are two types of Kind;

1. Icons: Actual “drawing” of an object. For example: iconBaba which represents the character Baba.
2. Text: Blocks of text referring to the character or decor element represented by an icon (eg: wall)

iconBaba which is the drawing of the character “Baba”

textBaba which is just a text with the character name’s ”Baba”

Feedback from Marco: avoid bloating, don’t separate into classes until it becomes necessary

Here are some features for each kind of item:

- For boundary, we set:

```
light = false
cancel = false
stop = true
push = false
you = false
win = false
```

the state of them will never change.

- For iconWall/iconBaba/iconFlag/iconRock, we set:

```
light = false
cancel = false
stop = false
push = false
you = false
win = false
```

If there is a rule in effect on the current map, the corresponding status will change immediately.

- For every Kind starts with text, like `textXXX`, we set:

```
light = false
cancel = false
stop = true
push = true
you = false
win = false
```

the state of stop, push, you and win will never change. If some text forms a complete rule, then `light = true`.

Tiles

The foreground is a Graphic which made of tiles.

The `tiles` is a two-dimensional plane, which is consist of a sequence of sequence of `tile`.

The `tile` is a sequence of items, because we need to represent the stacking order of the items. Every `tile` has a position.

```
public record Tile(
  Sequence<Item> include,
  Position position
){ }
public record Position(
  int x, //x-th row
  int y. //y-th column
){ }

public record Tiles(
  Sequence<Sequence<Tile>> map
){ }
```

Worldstate and its changes

The worldstate potentially changes whenever one of the arrow keys is pressed. Here are important aspects of it;

- **Recording the worldstate:**

  Recording a copy of the current worldstate will most likely be very important.
- **Move “you”:**

  We have four traversal orders here which is decided by the direction of move.
- **Rules changes:**

  For every `item.name=="is"` and check the item around it. Rewrite the rules for each item.
- **Win and lose situations:**
    - Win:
        - one of “you”s touches a “winning item”.

        - Traversing the sequence `tile.include()` for each tile, if we found some items the `item.at(i).you && item.at(j).win`, game over and the player win.
    - Lose:
        - There is no “you”.
        -

      Traversing the sequence `tile.include()` for each tile, if we don’t find an item that `item.you == true` then game over and the player loses.

      Examples requiring special attention: “Baba is you” and “Baba is wall” make “Baba” turn into “wall”. So whenever we change the `item.name()`, we should consider about other properties.

Basic functions

Some signatures describing item changes:

```
java public boolean sValid() {
// Evaluates three following tiles and returns true if it is a valid
sentence
}

java public Item newWin() {}
// Takes an Item as an input and produces a new item with the exact
same content
// except for the "win" field.

java public Item newYou() {
// Takes an Item as an input and produces a new item with the exact
same content except for the "you" field.
}

java public Item newStop() {
// Takes an Item as an input and produces a new item with the exact
same content except for the "stop" field.
}
```

```
java public Item newPush() {
// Takes an Item as an input and produces a new item with the exact
same content except for the "push" field.
}
```

Comments from the TA

Quick notes of some feedback we had after discussing our first report with Marco:

- separate the model and the view, the model should work on its own.
- Need a field in the gamestate that represents the rules —> need to be careful that rules don't apply before the next round when we are assessing them and if movement is possible.
- Writing documentation is a good idea

5.2. Meeting 2

Date: 06/05/2022 14:30-20:00

Meeting overview

We have all cloned the project from GitHub. We have reviewed some of our data structures, the changes will be described below. We created new classes to accommodate for the rules and we now have a full overview of what the classes will be and how they might interact together.

Botu and Zhiyao will be implementing what we wrote together here on the actual java project. I (Jamila) will quickly polish the meeting report.

Goals for this week

- √ Figure out how to implement rules *what class etc
- √ Implementing the classes designed here to the actual java files
- √ Create the rendering

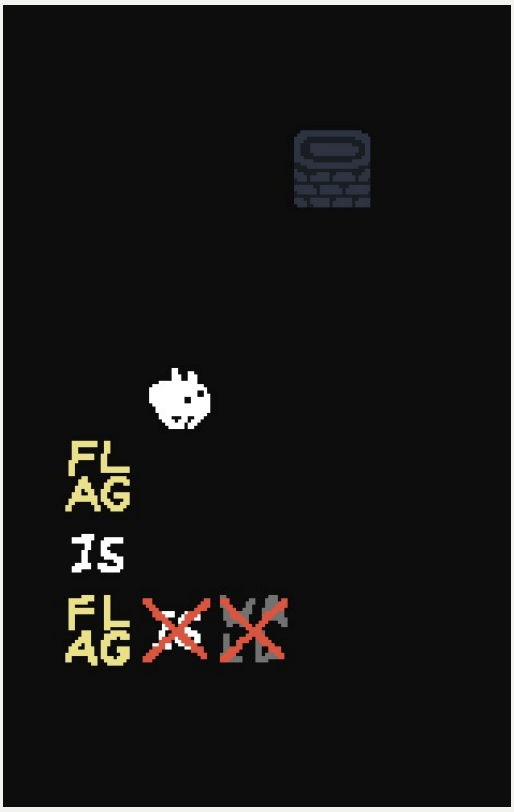
- [ ] Have a baba who can walk up down right etc…

Although we could keep the rendering for the end as it should work separately from the rest, this will make testing situations throughout the whole project much easier for us. We would like to have a simple rendering available (no decorations or complicated graphics yet) so that we can start implementing the rest of the project and test it.

Issues brought up

This is a short summary of subjects discussed during the meeting and their resolution.

- **Classes needed:** We have inspired ourselves from the Pacman labs and tried to understand what aspects were similar to our game and what aspects differed. Our list can be found in the "classes overview" section further down in this document.
- **Game controllers:** We concluded that the game will be played using the keyboard. Mainly the direction keys and possibly two other keys to restart a level or undo an action.
- **Background:** We are still indecisive about how to classify the background; Unlike with Pacman, our walls can change. However, this is more of a "naming issue" than an implementation issue, so we will focus on this later.
- **Tile checks:** We will be using arrays and have discussed this issue with Marco.
- **Wall rendering:** This issue was also discussed with Marco. It seems that we will only need to check 1 or 2 layers of neighbors to know how to render one wall.
- **Findings from playing the game and testing scenarios:**
    - We found out that there was a "cancelling" of rules when some of them contradicted, which was indicated by crosses.



Instance of a conflict. Screenshot from the game

A situation that did not create conflict, but showcased “rule precedence”/priorities. Screenshot from the game.

- **Rules**:
    - We have concluded that it is better for us to draft our own rules; The game is quite complex and sometimes contradictions are solved in unexpected, at least to us, ways. To avoid spending even more hours on figuring out the exact rules, we will be redacting our own set of rules. They will be very similar to the original game's rules.
    - Something to consider about the evaluation of rules:

      action → triggers change (through evaluation)? if yes —> evaluate rules —> render —> win/lose ?

Changes made

- **Modified Tiles:**
    - This class is renamed to Maze" for more clarity.
    - Changing Maze's implementation from Sequence> because it will be easier to handle with arrays when assessing neighbors for rendering walls.
    - We are keeping Tile as Sequence because this makes it easy to delete elements and modify it with little cost.
- **Created the class World.**
- **Created the class GameState:**
    - Needed to keep track of the rules and of the tiles and what they contain.
- **Created the class Rule:**
    - We have decided to represent rules as a directed graph. In this way, it will be easier to represent them and to also check what rules apply to one object.
- **Created the class Rules:**
    - Easier to handle multiple rules. Using a Sequence allows for more flexiblity and lets us "simulate" priorities for rules.
    - Created signature for instance methods to add a new rule to the existing ones and check if it causes a contradiction.