Text Game V3.10 Release Plan

Group 10 – Yue Ben, Dan, Nianzu Ma

For the final release of our Text Game V3.10, we’ll focus most on debugging and perfecting existing functions. One major innovation will be unplayable danger in the room, as stated in section 1.2.

To improve the reliability of our program, we’ll take techniques to avoid, detect as well as tolerate certain faults. Certain faults are allowed in the map files and user inputs, stated in section 2; also we’ll state intolerable faults – treated as throwable errors – in the same section. To avoid certain faults we decide on some global map designing rules and coding rules in section 3, and we’ll inspect the original map file and source code according to those rules to detect faults. Besides, testing and debugging techniques will also be used to identify and remove faults if they exist.

# GDF v3.10 design and Implementation

In our GDF file v3.10, some modifications will be made to GDF file v2.0, following the rules below to ensure down-compatibility:

* For existing sections[[1]](#footnote-1), we will only add new parameters after all existing parameters, or change the meaning (thus usage) of existing parameters without changing its type and value domain.
* New sections can be added freely as necessary.

Note section orders are irrelevant in the GDF file. See details and explanation in section 2.1.

## Improvements

There are several improvement we want to make on our program version 2.0.

### Version control in map parsing

Our GDF file V3.10 will be down-compatible. Although only GDF file with version 3.10 GDF file is designed for our game edition, GDF file with version 2.0 or 1.0 will also be accepted, leaving missing parameters as initialized value while parsing the map or initializing the game.

### Unify the executions of different commands w.r.t. Lighting

Especially for the USE/GET Artifacts command, and differentiating Artifacts on the ground and in the player’s inventory.

## Innovations

### Bi-directional path

Paths will be bi-directional. Thus as long as there is a path , there will be its “twin” path in the game. Each path will have a parameter with value 0 or 1, marking whether or not its lock status is shared with its twin path. By default the value is 1, i.e. shared lock status. Changing the lock status of one path 1 will also change its twin path 2’s if and only if path 1 has value 1 marking the shared lock status.

To implement this design, it’s not really matter how the map file is written, but the only modification will be in the map parsing. It automatically adds twin paths. But consistency will have to be checked. See section 2.1 for more details.

To help realize this design, we define **direction pairs** to cover the opposite direction pairs, such as N and S, SE and NW.

### Threats and Weapons

Add parameter to Artifacts, OR extend Artifacts. Each weapon can deal with a range of threat only.

### USE Weapon [to specific Threat]

Add parameters in the USE command.

### USE Key to specific Direction

Add parameters in the USE command.

# Fault Tolerance and Errors

Faults mainly exist in parsing the map file, and explaining and executing user input. In this section we’ll define tolerable faults, ignoring which will not affect the game playing, and intolerable faults, which will be thrown out as errors in a message to the player.

## Fault Tolerance

Some basic fault tolerance mechanisms in the program:

* White space is always tolerated inside map file or within user input.

Note “\t” is replaced with “ ” while extracting Tokens from user input.

* Only headline and section PLACES are required in the map file, other sections are optional. Which is to say, as long as there are places in the map, the game is allowed to begin.
* Order of sections are irrelevant in the map file. But of course, all sections depend on the PLACES section, so PLACES better comes first when designing a map file.

1. **Fault-Tolerant Mechanism of map parsing (Environment.java)**

|  |  |
| --- | --- |
| Tolerable Faults | Solution |
| The number of staff stated in each section is inconsistent with real number.  e.g. nPlaces != places.size(), nArtifacts != artifacts.size(), … | Ignored. |
| Incomplete map with OPTIONAL sections missing |
| Invalid lightLevel value in LIGHTING, i.e. lightLevel beyond 0~100 | That line ignored. |
| Invalid Place ID in LIGHTING. |
|  |  |

Table 1tolerable faults

We try to tolerate as many faults as possible, as long as the game is playable.

* While checking duplicate ID, Keys and Lights are only checked within their own types. Keys and Lights are all stored in the Artifacts list in the game class in the runtime, with type 1 and 2 respectively, and normal Artifacts are of type 0. Later we might add types for Weapons and Threats in Artifacts.

## Intolerant Fault

|  |  |
| --- | --- |
| Intolerant Faults | Solution |
| Invalid map file | Thrown out as errors, report it to the player, and ask for a correct map file. |
| Incomplete map without PLACES |
| Incomplete map without the “GDF…” line. i.e. no magic word, version number and environment name |
| Place or Artifact (including Key and Light) has no name |
| Duplicate ID within any section (excludes PATHS and LIGHTING where not tested) |
| Invalid source Place ID, destination Place ID, or direction in Path |
| Invalid location Place ID in Artifact |
|  |
|  |

*Table 2 intolerant faults*

# Coding Rules and Inspection

Coding Rule 1: every variable has to be initialized in the source code.

Coding Rule 2: precondition shall be checked within each function.

# Testing

Test cases will be used to test all functional requirements as well as the fault-tolerant designs stated above. For each part of the game, there are a list of objects to test.

## Test Map parsing

Parsing the numbers in the map file: Negative integer numbers are allowed to be used as staff ID, as long as it’s unique.

## Test Game playing

1. **Test Lighting**

By default, the room light level is 0 before 6am or after 8pm, it’s 25 before 7am or after 7pm, otherwise it is 50.

Items are visible only when the light level in the room (including the usage of Lights) is between 15 and 100. // for executing LOOK, GET, GO (going to previous place is allowed), USE (using previously used artifacts or artifacts in his inventory is allowed)

1. **Test GO**

Possible results: Go to some place; cannot go somewhere because it’s locked or nothing there; exit the game if the destination is an EXIT.

1. **Test LOOK**

The LOOK command can come with or without Direction or Object, meaning:

* LOOK: look around the surroundings of the room, describe current place and all outgoing paths.
* LOOK Direction: describe the path in the specific direction.
* LOOK HERE: look inside the room, list all items here without detailed description.
* LOOK Object: describe the specific item in the room or in the inventory.

1. **Test USE Artifacts**

Artifacts are treated as non-consuming, thus remaining in the INVE after executing the USE command (unless DROPped by the player).

After USE-ing a key to lock/unlock a path, even if the character leaves that place or later DROPs the key, those paths remain locked/unlocked.

After USE-ing a light to turn it on, the player carries its lighting with him. So if the character goes to another room with light active, the light will contribute to the lighting in this new room.

If the character USEd the light to turn it on, and DROPped it in the room, it will continue lighting the room.

1. Note there are 6 component sections totally in GDF v2.0, which include PLACES, PATHS, LIGHTING, ARTIFACTS, KEYS, and LIGHTS. The title line, containing the environment name, magic word and version number and so on, is not counted as a section. [↑](#footnote-ref-1)