First Iteration Project (I1): Basic Game of the Goose

Introduction

As explained in the introduction to the Game of the Goose, this game can be seen as a basic Conversational Adventure and therefore will be used for the first two iterations of the project. During these two first iterations we will motivate the concepts, tools and skills to be developed during the course.

Figure 1 illustrates the modules of the project that we will work on during this initial iteration (I1). This is the first approximation to the development of the essential modules of the system, as it has been seen in the introductory document (`I0-IntroProyecto.pdf`).

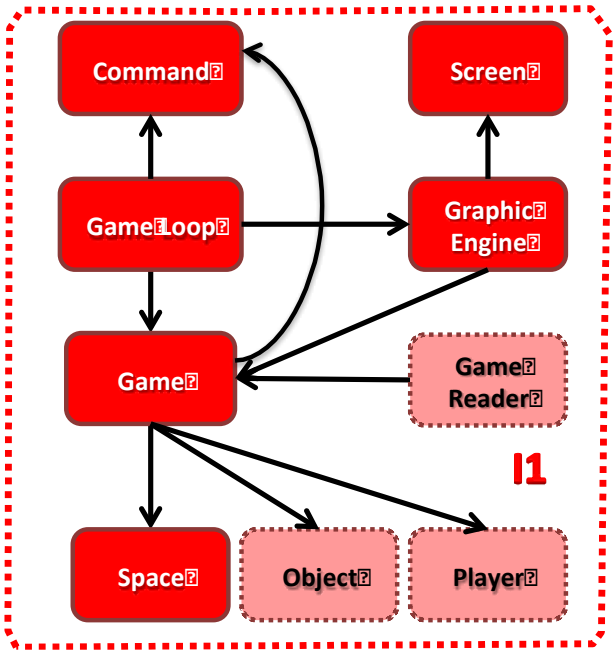

Figure 1. Modules to be implemented in the first project iteration (I1).

Figure 1 shows, in red, those modules partially implemented in the "initial code of the assignment". On the other hand, the functionality of the modules drawn in pink is not implemented as standalone functions, but part of their functionality is included in some of the red modules. For example, the `GameReader` module should include all the functions that allow us to load game data, but in the provided version the loading board (spaces) function is included within the `Game` module. Similarly, the `Game` module handles a primitive player and a primitive object that should be implemented in the `Player` and `Object` modules respectively, as it is done within the `Space` module.

The provided material can be compiled and linked to create a small application that allows us to:

1. Load game boards (series of sequentially linked spaces) from a data file.
2. Manage all necessary data for the game interaction in the computer's memory (spaces and locations on the board for an object and a player).

3. Support the user interaction with the system, interpreting commands to move the player around the board and ending the program.
4. Move the player by the longitudinally linked spaces changing the state of the game.
5. Show the current position of the player and the adjacent squares to the occupied one, by specifying the location of the object, if it is in one of the visible squares at that moment.
6. Release all used resources before exiting the program.

Objectives

The objectives for this first iteration of the project (I1) are twofold. On the one hand, becoming familiar with the basic GNU programming environment (`gcc` and `make`) and with the basics of programming style and essential code documentation, as well as beginning in the use of essential version control tools. On the other hand, to practice with the given material to improve it and provide new features.

The required improvements and modifications are the following:

1. To create the `Makefile` files to automatically compile and link the provided code and the new one to be implemented.

2. To fix the programming style and documentation for the provided source files (`seed-src`) and the new generated ones.

3. To create a `GameReader` module (`game_reader.c` and `game_reader.h` files) by extracting the existing functionality in the `Game` module for loading spaces. The new module will incorporate, in further iterations, readers for other necessary game's elements (objects, players, etc.).
4. To change the provided `Game` module to replace the functionality for reading spaces by the one included in the new `GameReader` module.
5. To create an `Object` module (`object.c` and `object.h` files) that integrates the necessary functionality for handling objects, in a similar way to the one performed by the `Space` module to handle spaces. In particular, the objects must be implemented as a data structure with a field to identify them (`id`) and another one with the object name (`name`), and also it must be provided with the necessary functions to create and destroy objects (`create` and `destroy`), read and change the values of their fields (`set` and `get`) and print the content for debugging purposes (`print`).
6. To create a `Player` module (`player.c` and `player.h` files) that integrates the necessary functionality to manage players, also in a similar way as the `Space` module does for spaces. In particular, players must be implemented as a data structure with the following fields: identification (`id`), name (`name`), location (`location`, a space identifier where the player is located) and an object (`object`, storing the object identifier the player is carrying). In addition, it must include the necessary functions to create and destroy players (`create` and `destroy`), read and change the values of their fields (`set` and `get`) and print the content (`print`*)* as in the previous case.

7. To modify the existing modules to use the new objects and players by replacing the corresponding functionality initially provided. For example, in the `Game` module, the identifiers for the `player_location` and `object_location` must be replaced by two pointers to the new `player` and `object` structures, and the appropriate functions from these new modules should be used for data manipulation. Or, for example, the `Space` module must be modified for keeping in its data structure the object identifier, together with the necessary functions to manage that value.
8. To create two new commands that allow the player to pick up an object from a space (`take` or `t`) and leave the carried object in another space, usually different from the previous one (`drop` or `d`), considering that, for the moment, the player can carry only one object and that it cannot be more than one object at the same time in any space.
9. To create a new file to be loaded by the program to implement a board for the Game of the Goose with 12 squares, all of them sequentially linked (north/south) and two specific spaces (5 and 9, which will be geese) also linked in special way (i.e., also east/west).
10. To modify, if necessary, the main program (`game_loop`) to use the new data types and the new functionalities.
11. To modify the initial `Makefile` to automate the compilation and linking of the whole project.
12. To debug the code until proper operation.

Assessment Criteria

The final score for this assignment is part of the final grade according to the percentage set for I1 at the beginning of the course. In particular, the assessment of this deliverable is calculated according to the following criteria:

- **C**: If C is obtained in every row of the rubric table.
- **B**: If at least two Bs are obtained and the remaining rows are Cs. Exceptionally with only one B.
- **A**: If the project gets at least two As and the remaining rows are Bs. Exceptionally with only one A.

Every submission that does not obtain the requirements of column C will obtain a score lower than 5.

Rubric table:

	C **(5 - 6,9)**	**B** **(7 - 8,9)**	**A** **(9 - 10)**
Delivery and compilation	(a) All the required files have been delivered on time. AND (b) It is possible to compile and link all the sources to get an executable file.	In addition to the previous column: It is possible to compile and link all the sources automatically by using the provided `Makefile`.	In addition to the previous column: Compilation and linking do not report errors neither warnings by using the `-Wall` option.
Functionality	It offers the basic functionality included in the version provided in the seed code after having done the requested changes, but the rest of functionality is not available.	In addition to the previous column: It has some of the new requested functions, but not all of them.	It includes every new requested functionality.
Coding style and documentation	(a) Variables and functions have names that help to understand their purpose. AND (b) All constants, global variables, public enumerations, and public structs are documented. AND (c) The code is properly indented[1].	In addition to the previous column: (a) Source files include headers with all required fields. AND (b) Prototype interfaces for all functions (both public and private) are documented. AND (c) Functions have a unique author identified.	In addition to the previous column: (a) Coding style is homogeneous in all the code[2] AND (b) Local variables in a module or function include an explanatory comment whenever it is appropriate.

[1] Indentation should be homogeneous. Every code block at the same level must have the same indentation. Besides, either tabulation characters or spaces (always the same number of spaces for each level) must be used, and combinations are not allowed.

[2] At least: the function names should start with the name of the module; the variables, functions, etc. should follow either a *camel case* or *snake case* notation, but they should never be mixed; the coding style should always be the same (e.g., *K&R*, *Linux coding conventions*, etc.) and never mix different coding styles.