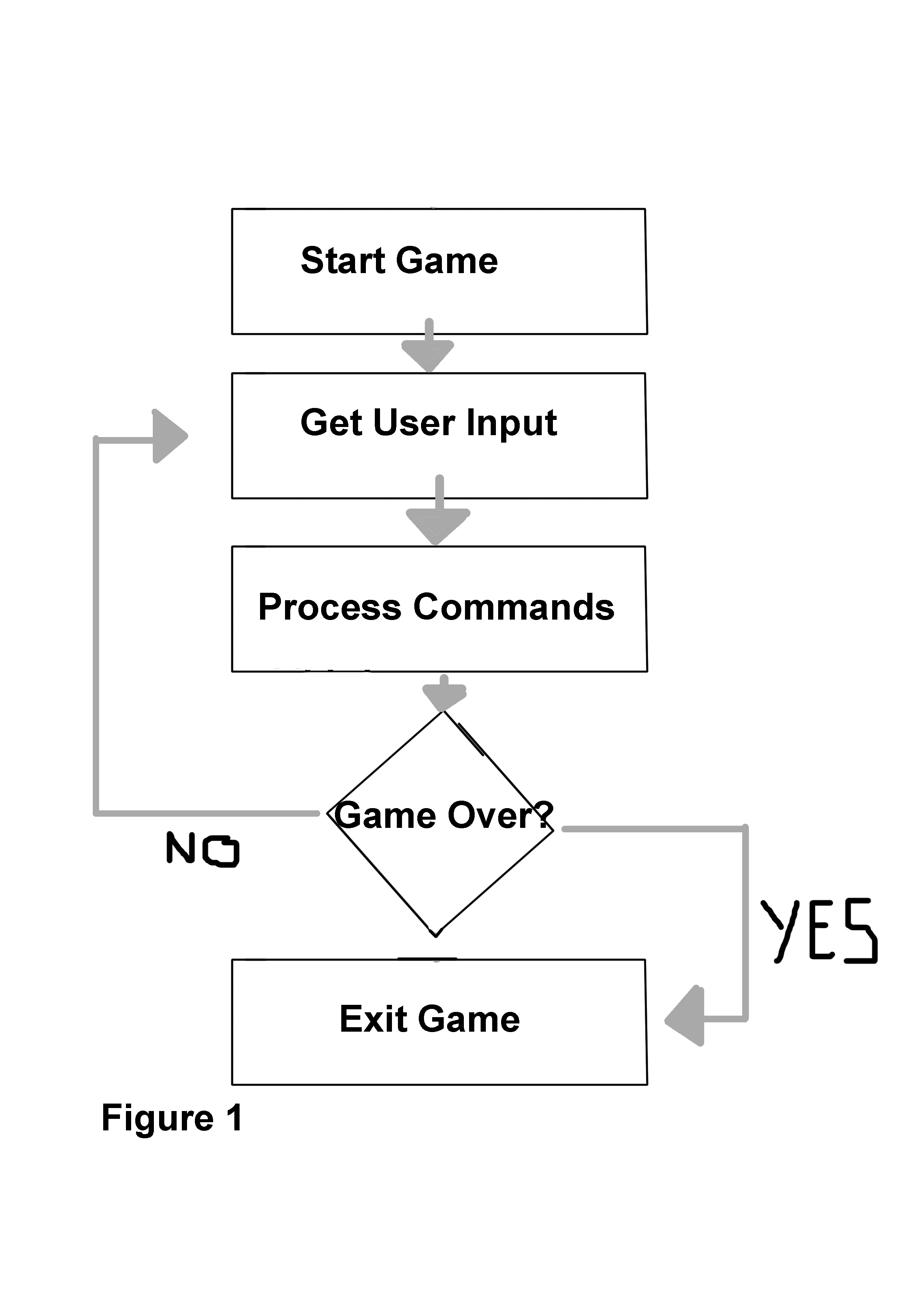
**Introduction**

This report will detail the iterative development of an algorithm for a working text adventure game. The game will contain the commands: Look, move, inventory, take and use. Detailed attention will be given to the requirements of parsing the input text.

**First Draft**

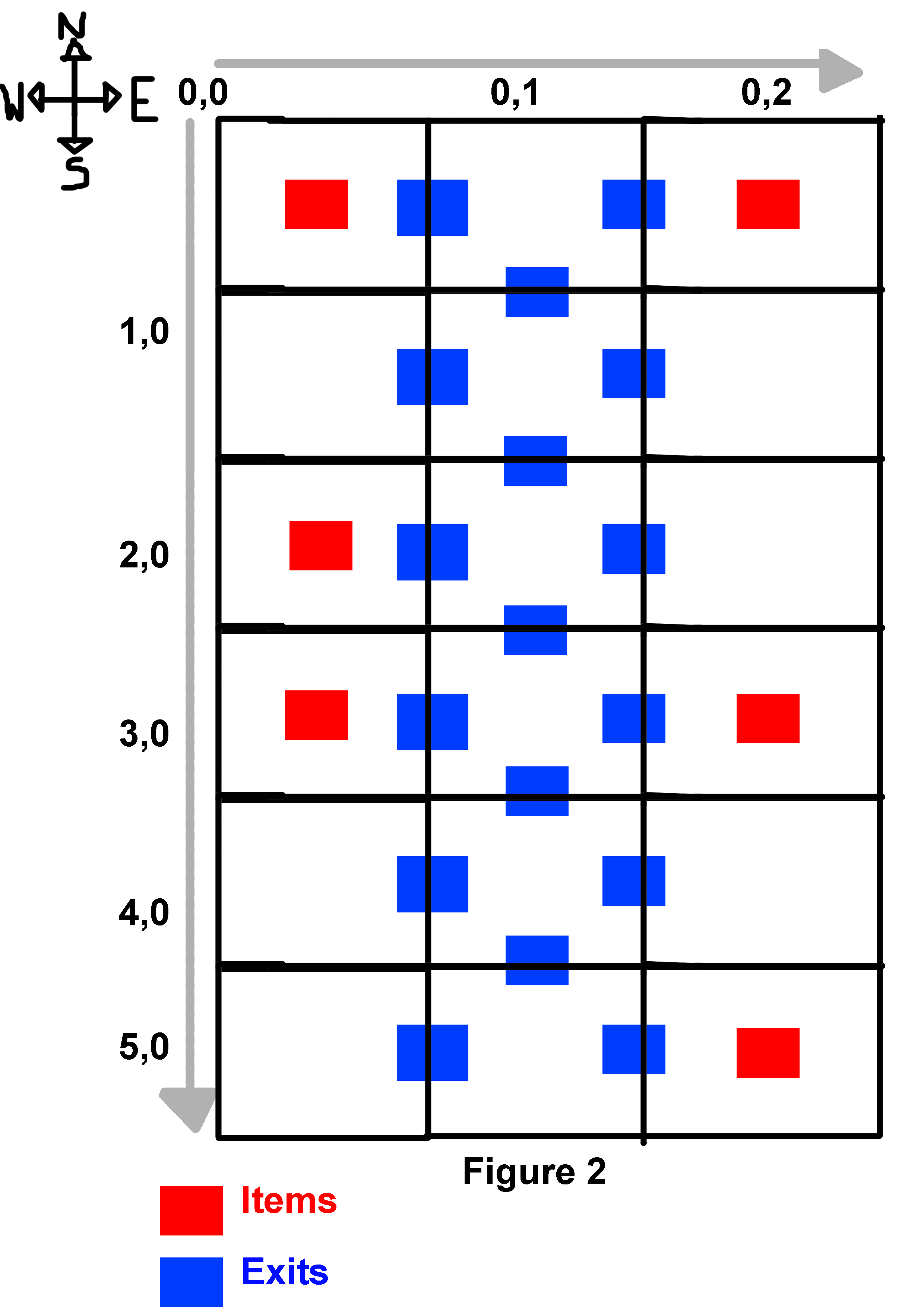


The first draft of the algorithm can be seen in Figure 1. The player will begin the game, loading the level and items. At this point the game will wait for the user to input a command, the program will them process that command and compare it against a Game Over condition. To accomplish this basic algorithm the following components are required:

* A way to generate a coordinates based level. An multidimensional array was implemented to create x and y coordinates on a grid.
* A set of directions that the player can move in. A struct was used instead of a class because “instances of the type are small and commonly short-lived or are commonly embedded in other objects.” (Microsoft, 2009). As the directions will be used by multiple classes, and will be used every time the player wants to move, this is the quickest and cheapest way to frequently reference the value type. This is because: “allocations and deallocations of value types are in general cheaper than allocations and deallocations of reference types.” (Microsoft, 2009).
* After this the input for commands needed to be implemented. Each line of text is broken down into two strings, “command” and “arguments”. Arguments sends information to the TextUtilities class, so that the string can be broken down and converted to the required variable type to be used elsewhere. Currently the Argument is sent to the Player script, to indicate the desired movement direction. The Command string is used to move, exit the game. A switch statement was used to check if the input string matched the case statement.

After this the game was broken down into all the required classes/objects. These are:

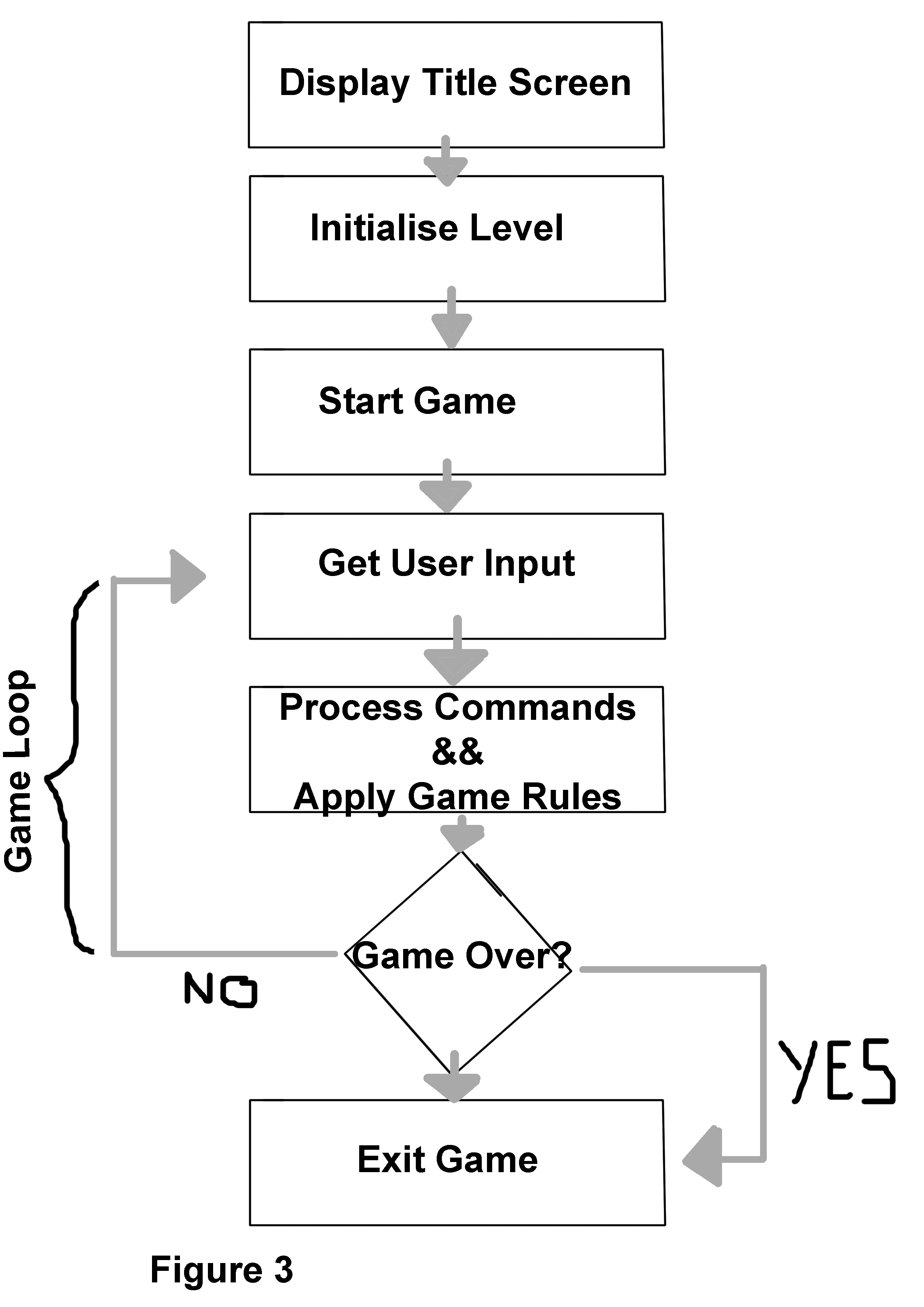
* Command Processor. This is used to take a line of text from the user input, break it down into two sections. Commands and arguments.
* Direction. Used to create the variables for the directions, and check if the input is a valid direction.
* Game Controller. This controls the starting and end of the game, along with the rules that the game must check when any game states change.
* Items. This class is used to give each item a title, description, weight. Though these variables will actually be implemented in the Level class.
* Level. Builds the map using a 2D array. Creates all the rooms complete with items, exits and descriptions of each room.



* Player. This class is used to store the players position, a List of inventory items, how many moves the player has used and how much weight is in their inventory. It is also used to move the player, pick up and use items,
* Rooms. This class is used to give each room a title and description.
* Text Buffer. Used to collect all text data and prepare it to be displayed .
* Text Utilities. This class receives strings from the player, and extracts the information to use in the appropriate classes. For example this is where the commands and arguments are processed into actions.

**Iterations**

A map was created to show the rooms and where each exit/item would be. Shown in Figure 2. This was imperative to keep consistentcy and have a reference to work from.



After this the games algorithm was amended to include a title screen to display intial information to provide context for the player, and gives them direction of what to expect next and how to play the game.

Then the program will initialise the level, the rooms, exits and the items within each room. The player will then begin the game. These amendments are shown in figure 3.

The user input has been expanded to include picking up items and putting them in an inventory. Implimented using a List, as a List can be dynamically edited during runtime. Each item now has a weight to add planning and challenge to the player experience.

Using items is also implemented, along with looking at items to bring up the description again.

**Text Parsing**

Either grammar driven or data driven. The parser must work regardless of capitals, or spaces. It must recognise a variety of words for the same action. Ideally it will be forgiving of spelling errors, either by automatically attributing the misspelt word to the correct command or offering the player a hint. E.g. “I do not recognise ‘takr sock’ did you mean ‘take sock’”.

This was achieved by creating a switch statement of all possible actions, as well as directions and items. The text is then compared to these lists to check if it is a command or an argument. After that the string is passed to the Player script where the appropriate Method is called. For example, if the user inputs “move west”, the Text Utilities class will separate “move” into a command, then load the Move method in the Player class. “West” is used to update the player position on the grid, also in the Move method.

**Final Results**

**Conclusion**