

Technical Report

This technical report will explain the how's and whys of my project, in order.

First, in the main function, keyboard interrupts are disabled. This setup is required for keyboard polling, used later. After that, player and reward positions are generated, in this order. Two separate functions are used for this, one for each entity: they generate two random numbers in the appropriate interval using syscall 42 (Peter, 2014) and use them as x and y coordinates. The genReward function is a bit different, as it has to check whether the randomly generated position for the reward is the same as the players one and redo it until its not. It is important to note that these two functions are not printing on display; they are only storing the positions in appropriate .word variables.

Next, the score text, score value and map are printed on the display, in this order. The printMap function uses a .word array of ascii characters to print out the map on display. Before it is printed, the player and reward letters are added. This is not the most efficient approach in terms of space, however I think it is more extensible (makes it easier to make custom maps, for example), which is why I chose it over generation using loops. I chose this size of the map (6x8) because it made testing it less of a hassle compared to a bigger map.

The game loop is the following: check for input, move, check for collision, check if the reward has been taken, if score is less than 100 back to the beginning.

Input check is handled using keyboard polling. It is very similar to the example given in the week 15 lecture (James, 2024). I have added a buffer limit, to make sure the program does not crash or react when an input bigger than 1000 characters is given. Once there is input in the text area, it checks if the recent text is longer than one character, in which case it ignores it. Next, it checks if that one character is either w, a, s or d (upper or lowercase). Based on input, the player then moves accordingly using the cursor function by clearing the previous player pos, updating the player pos in memory and printing the P character in the new one. The cursor moves using the ASCII 7 feature mentioned in the 'help' section of the MMIO simulator in MARS and the P character is printed using the keyboard register (Peter, 2014). This approach is, in my opinion, better than reprinting the whole map every time, as there are less operations and not that many more lines of code.

The collision check looks at the player coordinates and decides if they are behaving or not. If they stepped on or outside the walls, it jumps to the end screen. This is done before the reward check because there's no way you can get the reward and then die by colliding with the wall in the same turn. It would be redundant to check for reward collecting before this.

The reward check looks if the current player and reward positions are the same, in which case it generates another reward with the genReward function from before and prints it using cursor.

The win condition of 100 points is checked inside the game loop without a function calling, as it only takes two lines of code. Getting 100 score exits the game loop and pops up the end screen.

The end section clears the display using the ASCII 12 feature, found under the same help tab as before (Peter, 2014), and reprints the score text and value, as well as the text 'GAME OVER'.

Enemy extension

Implementing the enemy extension meant writing new functions, as well as updating existing ones. However, these changes ended up being somewhat similar to already written code for the original game.

At the start, a new `genEnemy` function is called, between the player and reward generation. It generates a random enemy position, different from the player position. Also, the `genReward` was updated to check if the reward position is the same as the enemy, in addition to the player check.

The updated game loop is as follows: check for input, move, check for collision, move enemy, check if the reward has been taken, check for collision again, if score is less than 100 back to the beginning.

One of the two big changes is the addition of the `enemyTurn` function. I have considered three ways of designing this function: the enemy can move towards the player, the reward, or both, depending on which is closer. I made the game in C to test all three design choices and found out that moving the enemy towards the player was borderline unplayable. The player would get cornered between the enemy and the wall and it would almost always result in a loss. This conclusion ruled out the first and the third option, so the remaining choice was to move the enemy towards the reward. With this approach, the game feels like a race for the reward between the enemy and the player, which makes it winnable if the user pays enough attention.

The other significant change is the additional collision function call, placed after the enemy's turn, between the reward handler and win condition. I find it intuitive that there should be two scenarios where the player dies from the enemy: first is when the player steps on the enemy and second is when they both move on the same position. The additional call is implemented to account for the second scenario. A design choice regarding this was to call the second check after the reward handler. If both the player and the enemy step on the reward at the same time, I think it's fair and intuitive for the player to collect the reward before dying.

Smaller changes include the addition of the `enemyCollectedReward` function, called inside the reward handler. It prints out a new reward on screen without adding to the score. Collision was also updated to cover the enemy.

Note: I have encountered weird behavior when changing delay length in the MMIO Simulator. If that is not part of testing/grading, I suggest running the game at 1-5 instruction executions delay.

References:

James, S. (2024) "IO and Interrupts" Lecture 15-1. LZSCC.131: Digital Systems.

Peter, S. (2014) "Syscall functions available in MARS". [online] available at: <https://dpetersanderson.github.io/Help/SyscallHelp.html>.

Peter, S. (2014) "Keyboard and Display MMIO Simulator". MARS. Help section.