Coursework 2
MIPS Assembly code

38879557

1 Basic Game

1.1 Approach

The game was created with the use of interrupts to process keyboard interaction from the user. The Display Device from Mars was the tool of choice to display the game graphics. RAM was used to store the position of the player and reward, as well as ASCII values, the score, and reserved space for different purposes. For instructions to set the MARS environment before running the game refer to Appendix A.

1.2 File Distribution

The game was distributed into different files to keep the code organized and open for future modifications and/or extensions. The files were named appropriately to give a clear understanding on what each of them is doing, however, it is highly recommended to check the flowchart that explains file relationships in Appendix B.

1.3 Problems—Solutions

One of the first problems encountered was the inefficiency of the keyboard buffer architecture initially proposed, as it had a linked list as an underlying data structure. Because one can't know how much space will be needed to store the keys pressed, a list was the ADT of choice. However, this represented a significant risk, as the heap could potentially run out of space mid-game. In addition, the accessor methods for a list have an average time complexity of $O(n)$, n being the number of elements in said list. This was solved by opting for a queue[1] with an array as an underlying data structure. The tail of the queue kept track of the most recently processed key, while the head kept track of the most recently pressed key. This meant that accessing a key for processing would have a time complexity of $O(1)$.

A second problem encountered was the computationally heavy duty of displaying the entire game board every time the player changed position. This was due to the first game environment approach implemented into the game. Before, a matrix holding all the game elements was being used as a grid (Please refer to Appendix C). However, this implied that every time an updated board was desired to be displayed, one would have to clear the screen and iterate through every row (r) and column(c) to display the characters contained in it $\rightarrow O(rc)$. Nevertheless, it was then decided to store the position of player and rewards directly into RAM, and the grid used was the one integrated inside the Display Device tool. That way, when the player moved, one could just delete the current cell and move the cursor to the new position to then display the character $\rightarrow O(1)$. This made the display function much more efficient, yet as extensible.

A final problem encountered was related to displaying the score, as everything loaded into the transmitter register will be taken as an ASCII value. In order to solve this, an algorithm was created to convert each of the integer's digits from the score into ASCII values.

[1]This was inspired by LZSCC.150 Lab 15 on interrrupts

2 Game Extension

2.1 Approach

The game was extended with the addition of an enemy that prevents the player from reaching the reward. The enemy was coded in a way that makes the game harder, but still fun and enjoyable to play. The same file structure was kept for this extension, with the exception of a new file called "Tracker_enemy.asm", which handled enemy behaviour and updated its position accordingly. The concepts of Taxicab geometry[2] where used to shape an enemy which, from the eyes of the user, appears to be "smart"[2].

2.2 Enemy behaviour

An enemy that had access to the player and reward positions at all times was created. Meaning that it could track either of the two really easily with a straight forward tracking algorithm. In general terms, we can say that game enemies can undergo 1 of 2 phases: defensive or offensive. A defensive enemy in this case will just block your path, while an offensive enemy may try to kill the player. When the enemy behaviour was first approached, it was established as an offensive entity, however, after some testing, it was concluded that the game resulted too hard to handle, defeating the purpose of playing a game for having fun. This was solved by creating a hybrid enemy, which could act as both defensive and offensive.

With this in mind, the enemy was given a random chance of being offensive in the next turn. However, to make things more exciting, it was given a "range of vision", so that it could only attack while the player was located inside this range. The area this "vision" covered was calculated using a Taxicab topological ball (Please refer to Appendix D). Following Taxicab geometry logic, the radius for this circumference is determined by the Manhattan distance formula:

$$MD = |x_1 - x_2| + |y_1 - y_2| \tag{1}$$

This formula calculates the amount of moves needed to reach one position from another in a Taxicab grid, a value that resulted really useful when comparing how close is the enemy from any given cell. Back to the behaviour, a different problem encountered was originated by the enemy tracking the reward itself. This was solved by computing a target cell between the player and reward, because we don't want the enemy to overlap rewards, but just guard them. If the enemy was on defensive mode, it would track this cell.

Finally, a last obstacle came across when the enemy continuously "stole" rewards from the player, as it couldn't recognize when a reward was in its path. This wasn't an issue when the enemy was only attacking, as that is a handy feature, however, for defensive mode, it makes game progression almost impossible. This was solved by correcting the enemy position when its $MD1$ to the reward was equal to 1. For a more detailed explanation about how the enemy was programmed, refer to Appendix E on enemy logic (this is also explained in code comments).

[2]A form of geometry in which you can only move along the lines of a grid, rather than along diagonal lines in the grid.[1]

References

[1] E. F. Krause. Taxicab geometry: An adventure in non-euclidean geometry. *(Courier Corporation)*, 1986.

[2] Funge J. Millington, I. Artificial intelligence for games. *(CRC Press)*, 2018.

Appendices

A Instructions to run the program

1. Open the Display Device tool.
2. Set the delay length to 1.
3. Turn off DAD option.
4. Assemble the main file.
5. Connect to MIPS.
6. Run the code.

B Flowchart diagram between game files

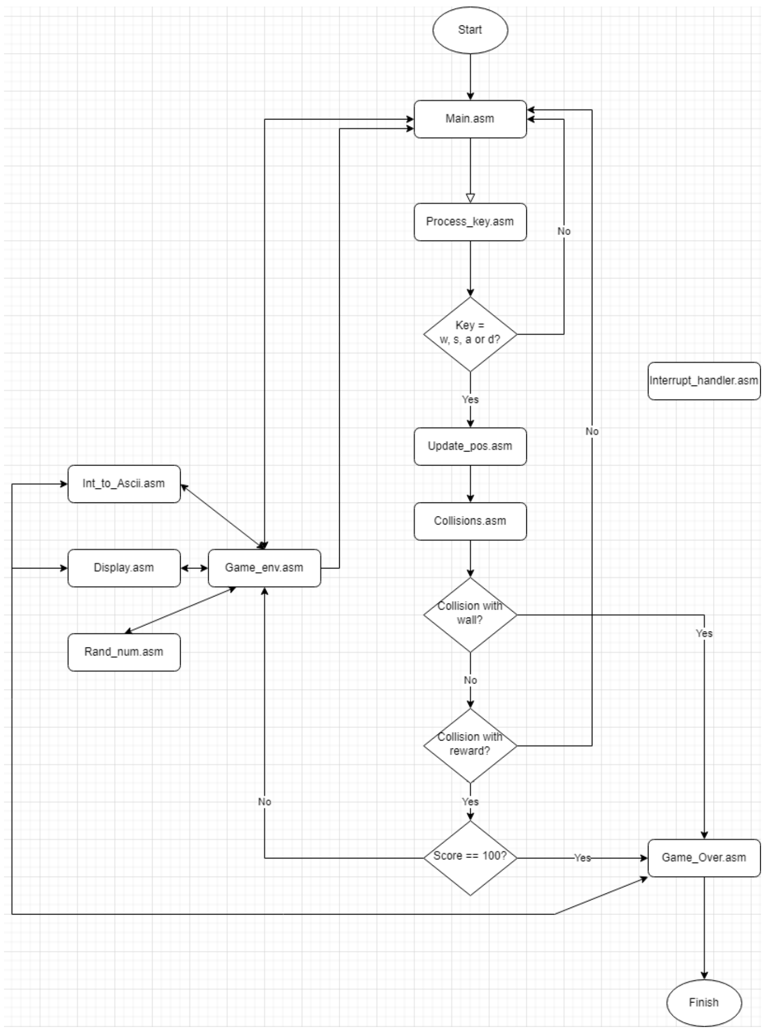

Figure 1: Flowchart diagram of file relationship (Bidirectional arrows are linked calls)

C Game environment using matrix

```
// .data

matrix:    .space  324       #array of 81 words (9x9)
rows:    .word 9             #every row is represented
columns: .word 9             #by 9 continuos words
```

D Taxicab topological ball of radius 3

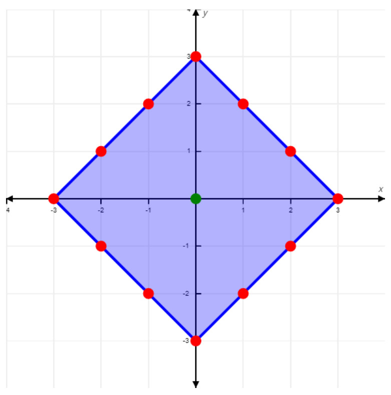

Figure 2: Taxicab topological ball of radius 3

E Enemy logic

There is a hierarchy of things we need to check in order to create an enemy that is smart enough to make the game harder but not impossible:

1. Check if the player is on the enemy's "range of vision", we are doing this to prevent the enemy from chasing the player if it is too far away to realistically "kill" it.
2. Check the phase status — this has the second highest priority, as the enemy needs to stop whatever it was doing before to follow the player (if the phase is set offense), even if this means stealing a reward.
3. Check if the enemy is one move away from the player — if we reached this step, we know the enemy is on defensive mode, so we prioritize not killing the player over everything else, making the enemy stop all its movement.
4. Check if the enemy is one move away from the target cell — We will always prioritize going for the target cell, even if this means that the enemy is next to a reward.
5. Check if the enemy is one move away from the reward — if this is the case, and the target cell is on a different side, we want the enemy to correct its position by moving away from the reward to make it avoid stealing it on its path.
6. If none of this conditions is met, we just want the enemy to track its target cell.