CAB202 - Microprocessors and Digital Systems

Assignment 1

Pedro Alves (n9424342)

April 20, 2018

Contents

Executive Summary **3**

Program Overview **4**

Splash Screen **5**

Border **6**

Dashboard **8**

Race Car and Horizontal Movement **11**

Acceleration and Speed **15**

Scenery and Obstacles **19**

Fuel Depot **23**

Fuel **24**

Distance Travelled **25**

Collision **26**

Game Over Dialogue **27**

Part B - Highscore Screen **28**

References **29**

Executive Summary

Program Overview

Things to talk about
change_state

Splash Screen

The splash screen is the first screen the player sees when they start the game. It provides basic information about the game and will change to the main game screen when the player presses any key.

Functions

```c
// main.c
void update_start_screen();
```

Called every tick of the main game loop. Will change the game's state to *GAME_SCREEN* if there is any key in the input buffer. Since the *change_state()* function already purges the input buffer, we do not have to worry about the game skipping straight to the *GAME_SCREEN*.

```c
// main.c
void draw_start_screen();
```

Calculates the x and y coordinates of each string to be shown based on the dimensions of the screen. Will then call *draw_string()* and *draw_center_text()* multiple times to add the strings to the desired location.

```c
// main.c
void draw_center_text(char * text, int y);
```

Calculates what x coordinate is required in order to have the text appear at the middle of the screen. Then calls *draw_string()* to print the text.

Testing

Testing that the splash screen shows up when the game is started.

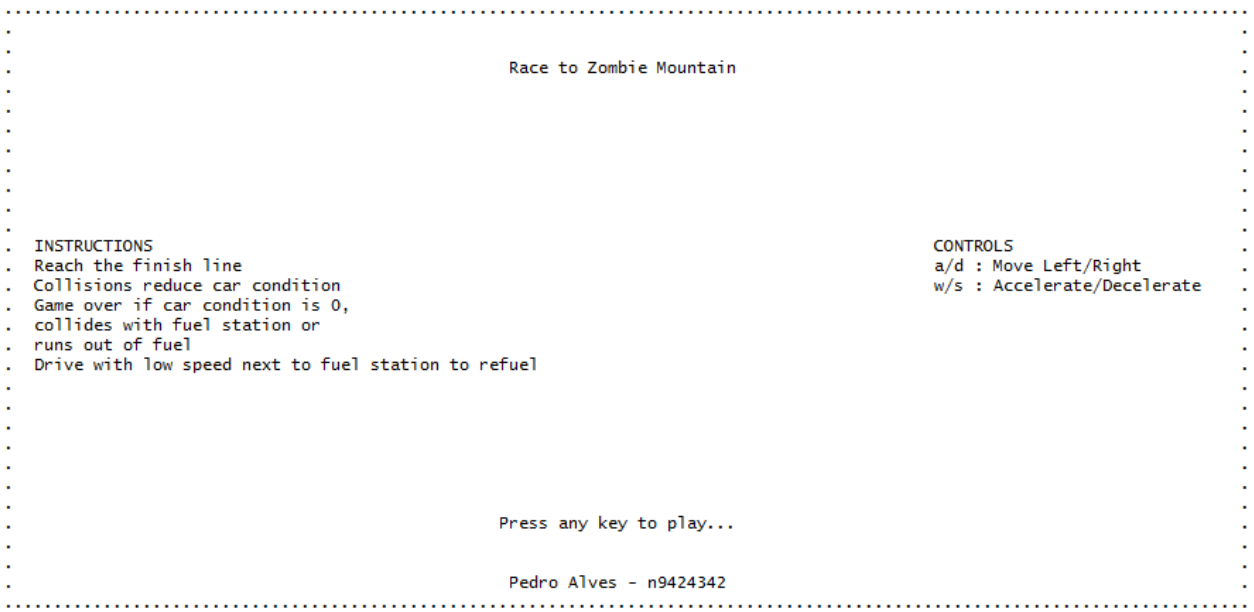

Figure 1: The splash screen when the player starts the game

Border

The border is simply a rectangle that is drawn on the edge of the terminal. It supports every terminal size. The *draw_borders()* functon is the last one called before *show_screen()* in the draw step of the game loop. This ensures that no other graphics ever block the border.

Globals

```c
// zombiemountain.h
#define BORDER_CHAR 46
```

The character that will be used to represent the border. The number 46 represents the ASCII character "." (full stop).

Functions

```c
// main.c
void draw_borders();
```

Draws 4 lines that form a rectangle on the edge of the screen. The length of these lines are calculated by using the screen width and height in order to make the borders work on every screen size.

Testing

The game is started in different sized terminals and the borders are verified to have been drawn correctly.

Screen: 80x24

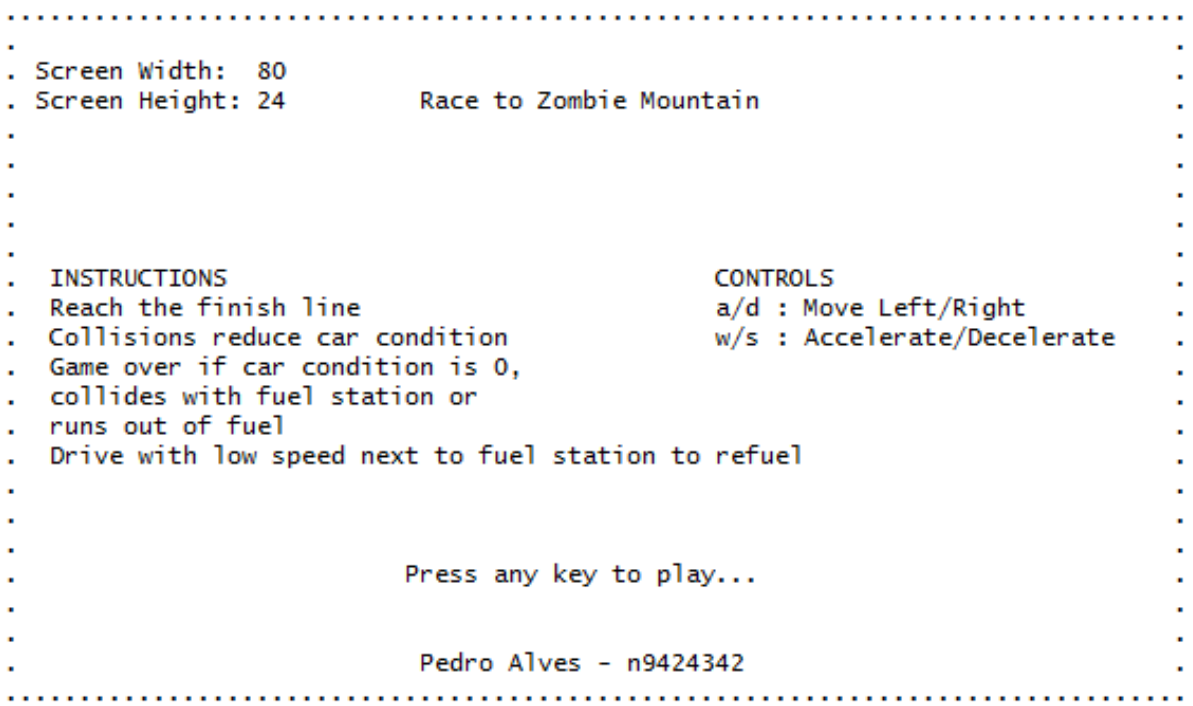

Figure 2: The border with screen dimensions of 80x24

Screen: 126x31

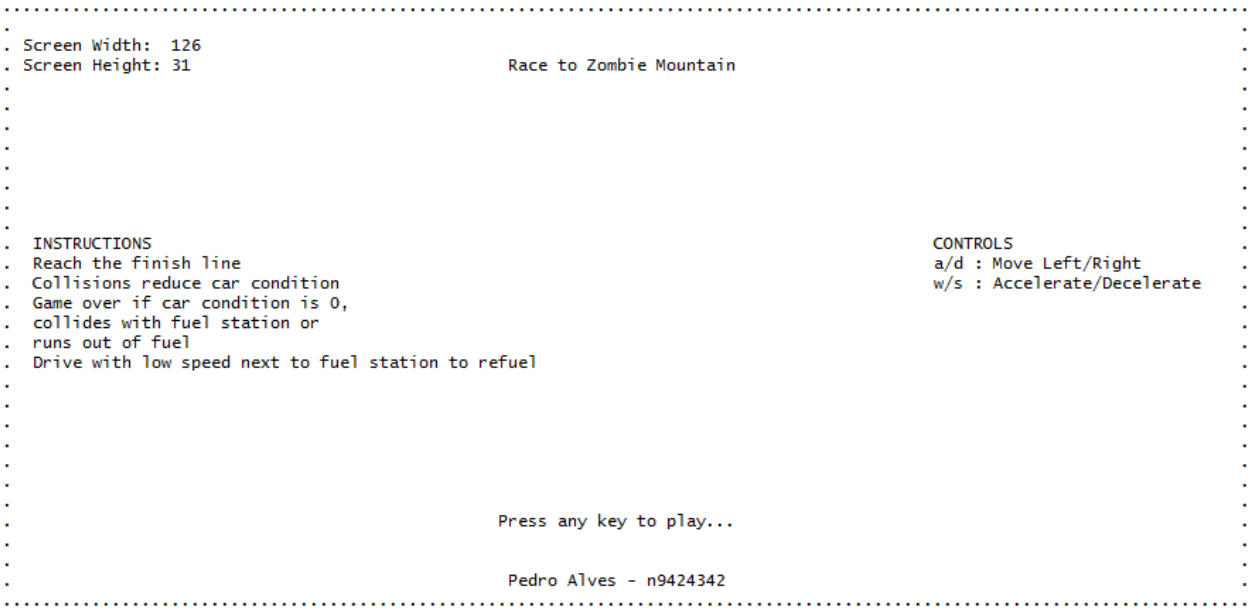

Figure 3: The border with screen dimensions of 126x31

Screen: 190x50

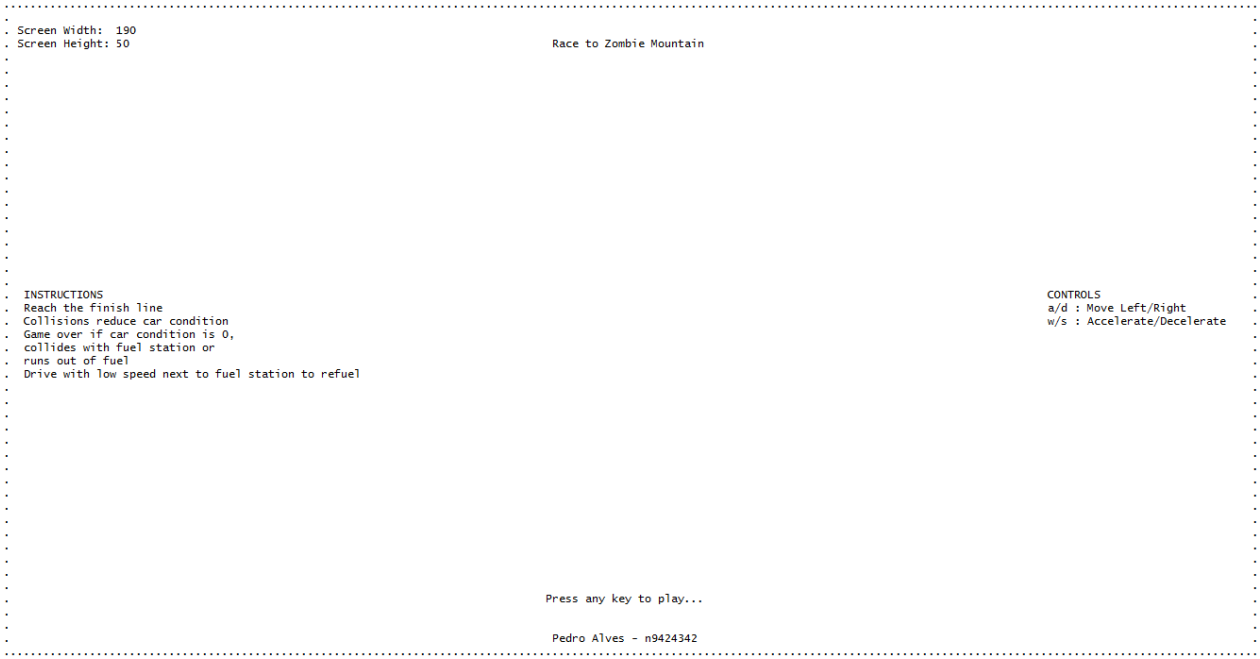

Figure 4: The border with screen dimensions of 190x50

Dashboard

A sub-window in the terminal which displays data regarding the player's car such as condition, speed and fuel as well as displaying stats on the game itself such as time spent and total distance travelled.
Warnings also appear on the dashboard to notify the player that the car is offroad or is refuelling.

Globals

```
// zombiemountain.h
int dashboard_x;
```

The x-coordinate of the border between the dashboard and the playing area.

```
// obstacles.h
#define DASHBOARD_SIZE  20
```

The width of the dashboard.

```
// zombiemountain.h
#define DASHBOARD_BORDER_CHAR 47
```

The ASCII character that will represent the border that separates the playing area and the dashboard.

```
// zombiemountain.h
int speed;
```

The current speed of the player.

```
// zombiemountain.h
int fuel;
```

The current fuel available to the player.

```
// obstacles.h
int car_condition;
```

The condition of the car as a percentage.

```
// hscore.h
int score;
```

The current score of the player.

```
// zombiemountain.h
int distance_travelled;
```

The distance travelled since the start of the game.

```
// zombiemountain.h
double game_start_time;
```

The time in milliseconds that the game started.

```
// zombiemountain.h
timer_id refuel_timer;
```

A timer that is set when the car starts refuelling.

Functions

```
// main.c
void draw_dashboard();
```

Draws a border between the playing area and the dashboard area. Calls *draw_string()* and *draw_int()* multiple times to print the relevant globals and their captions.
If the car is offroad or refuelling, a relevant warning will also be drawn. Additionaly for refuelling, will display how long until it is finished.

```
// main.c
bool car_offroad();
```

Checks if any portion of the car is outside the road boundaries and return true if so.

```
// main.c
double refuel_time_left();
```

Calculates how much time is left to finish refuelling. This is done by calculating the difference between the current time and the time the *refuel_timer* is meant to reset, this is then subtracted from *3.0*.

Testing

Stats are modified with gameplay

Two screenshots of the same gameplay are taken where the fuel is checked if it is decreased and the time,distance and score are increased.

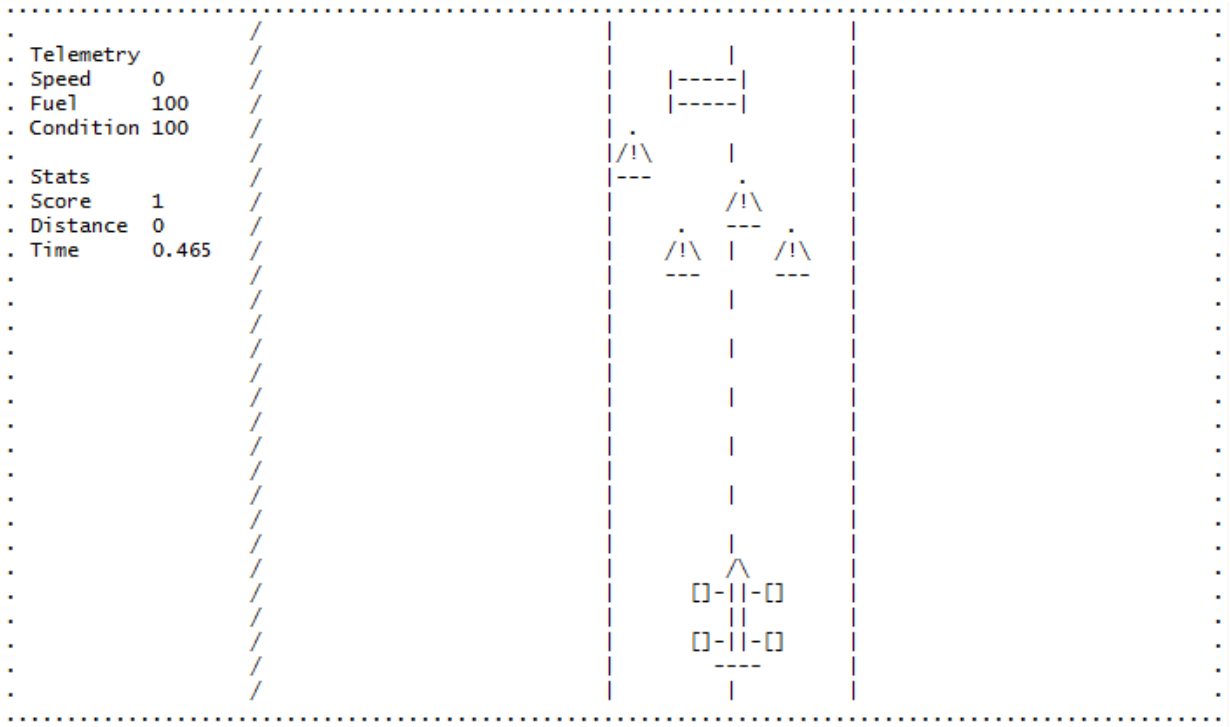

Figure 5: A screenshot of the game as it starts

Figure 5 shows the dashboard on the left side of the terminal with a border clearly separating it from the playing area. To achieve the second part of the test, the 'w' key was pressed 3 times to test if the speed is increased and the car was moved horizontally where necessary to avoid obstacles.
The condition stat will be tested in the *Collision* section and the refuelling warning and timer will be tested in the *Fuel* section.

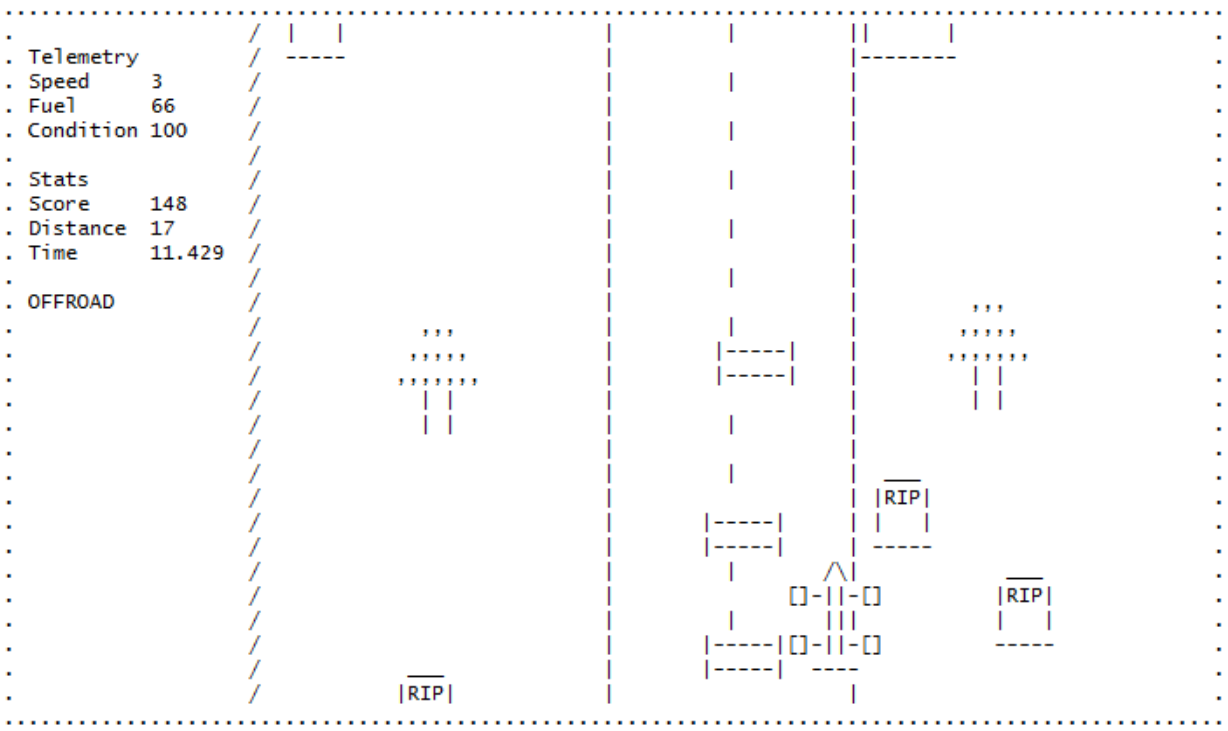

Figure 6: A screenshot of the same game sessions as the figure above but 11 seconds into gameplay

Figures 5 and 6 show that the stats are represented in the dashboard and change when supposed to. The *OFFROAD* warning also appears when the car moves beyond the border of the road.
Figure 7 shows the low fuel warning appears when fuel falls below 25%.

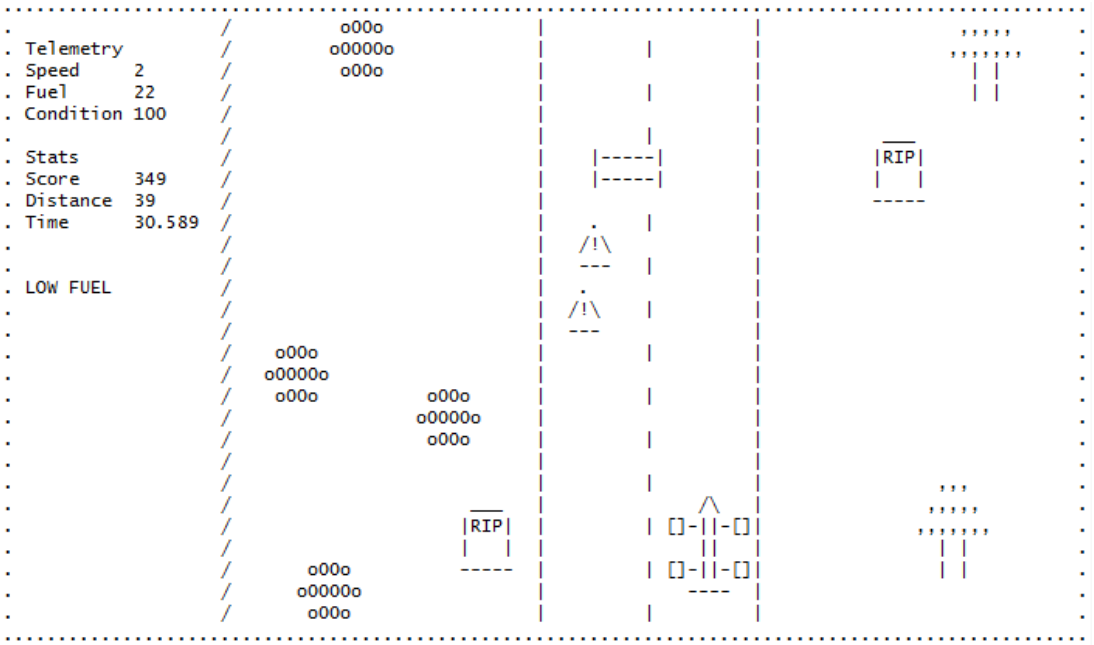

Figure 7: Low Fuel warning appearing when fuel is below 1/4 the maximum

Race Car and Horizontal Movement

The race car is a sprite 8 units wide and 5 units tall. This sprite is always stuck in the same position with the illusion of movement given by the obstacles being moved downwards. The speed at which the obstacles move is proportional to the speed setting.

Globals

```
// imagemngr.h
#define PLAYER_WIDTH  8
```

The width of the car sprite.

```
// imagemngr.h
#define PLAYER_HEIGHT 5
```

The height of the car sprite

```
// zombiemountain.h
#define INPUT_MOVE_LEFT    'a'
```

The keyboard input that will make the car turn left.

```
// zombiemountain.h
#define INPUT_MOVE_RIGHT   'd'
```

The keyboard input that will make the car turn right

```
// zombiemountain.h
sprite_id player;
```

The car sprite which the player controls.

Functions

```
// main.c
void setup_player_car();
```

Place the car sprite in the middle of the road, 2 units above the bottom of the screen. Also sets the car condition to 100% and fuel to max.

```
// main.c
void handle_input();
```

Get the next character from the input buffer. If it is a valid key, call the specific input handler.

```
// main.c
void handle_movement_input(int key);
```

Checks if the *key* variable wants the car to turn left or right. Will then check if the car will be in the bounds of the playing area, if it'll collide laterally with any obstacle and if the speed is above zero. If all three checks pass, then the *sprite_move()* function is called.

```
// main.c
bool in_bounds(int x, int y)
```

Checks if the *(x,y)* coordinate is in bounds of the playing area, returns true if so.

Testing

Car doesn't move when speed is 0

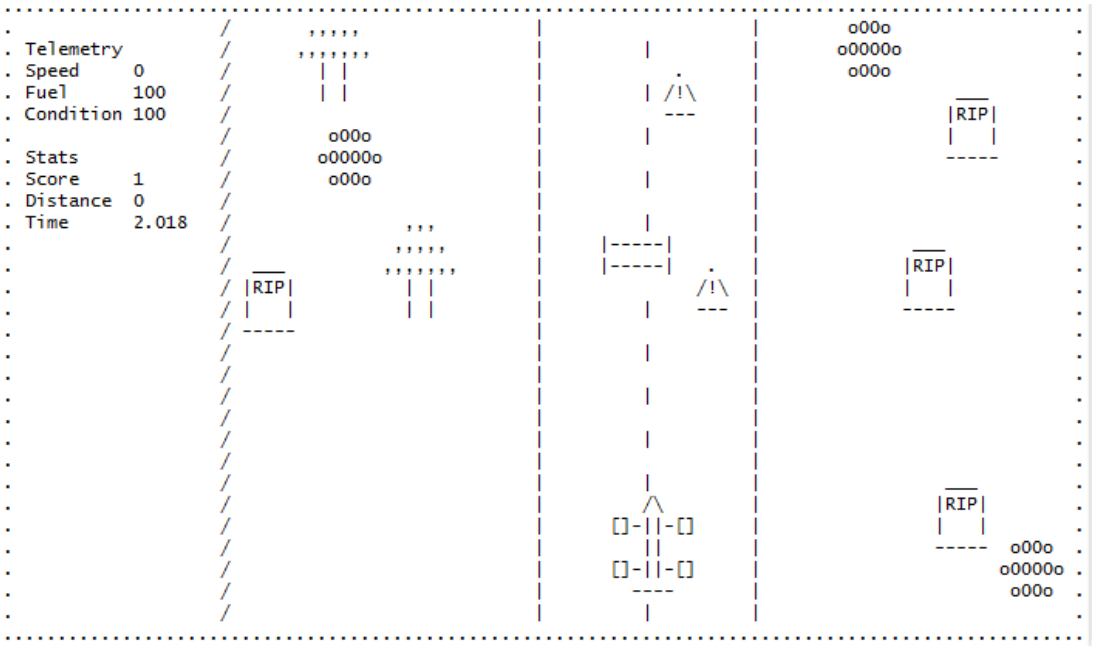

Figure 8: Car in the middle of the road with speed equal zero

The following inputs were pressed and the result shown in Figure 8: "a,d,a,d"

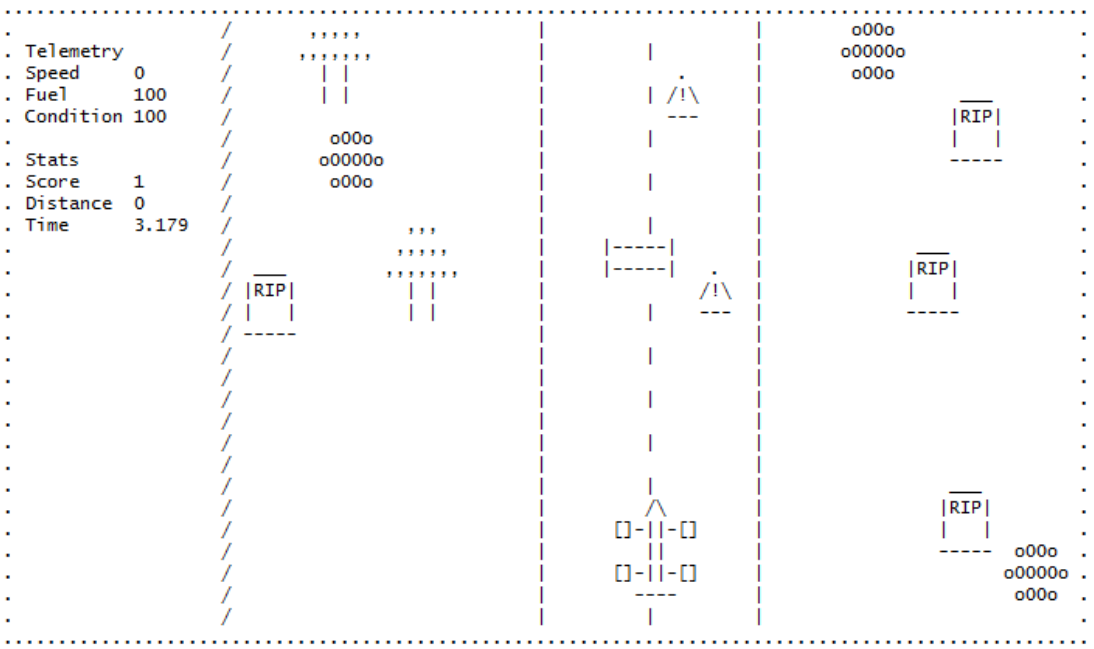

Figure 9: Results after attempting to move horizontally with speed equal zero

Car moves left and right

From the position in Figure 9, the following inputs were pressed "w,a,a,a".

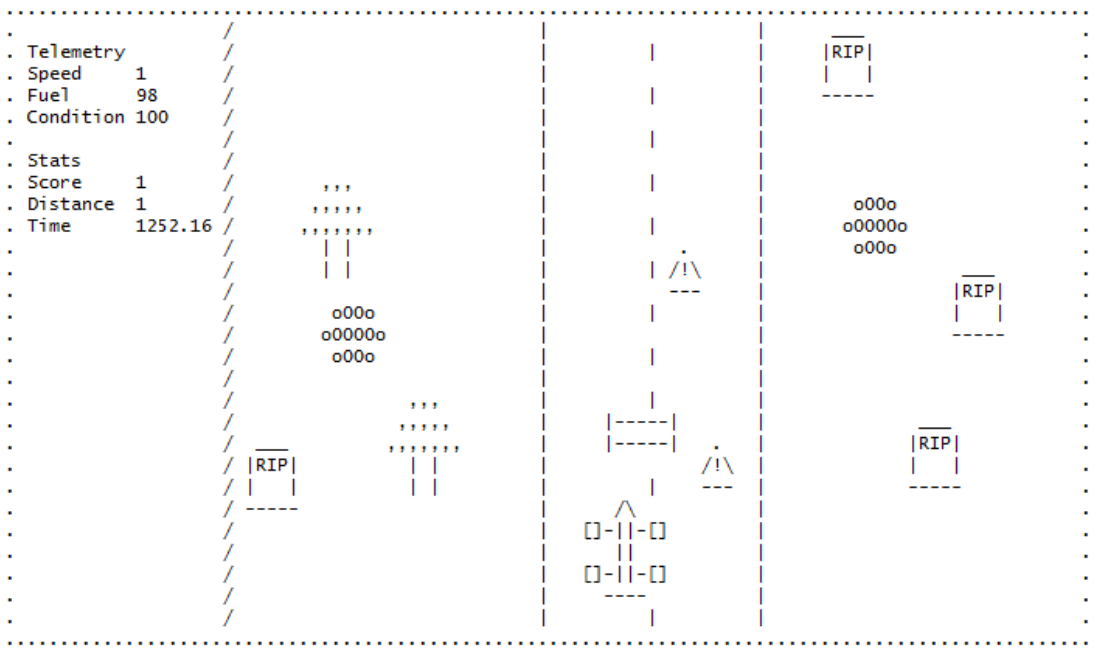

Figure 10: Moving the car left with speed equal 1

After the car is reset due to the inetivable collision in Figure 10, the following inputs were pressed "w,d,d,d".

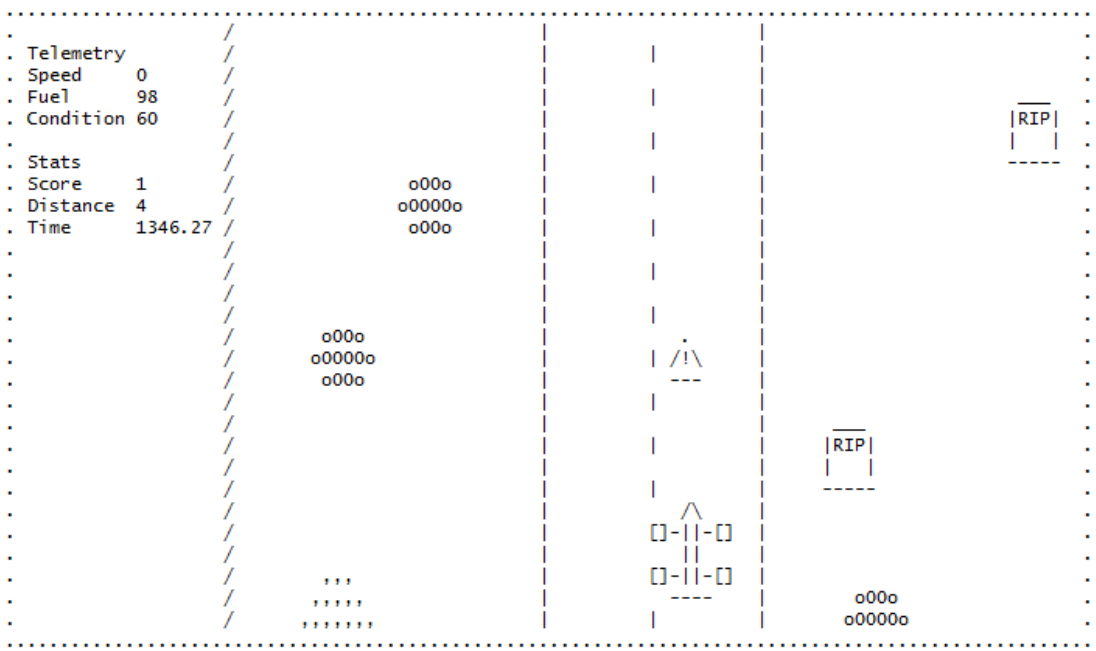

Figure 11: Moving the car right with speed equal 1 (car was stopped to grab screenshot)

Car stays in bounds

The car was moved to both extremes of the playing area with the lateral movement input held down. Figure 12 shows the result of holding down 'd' and Figure 13 shows the result of holding down 'a'.

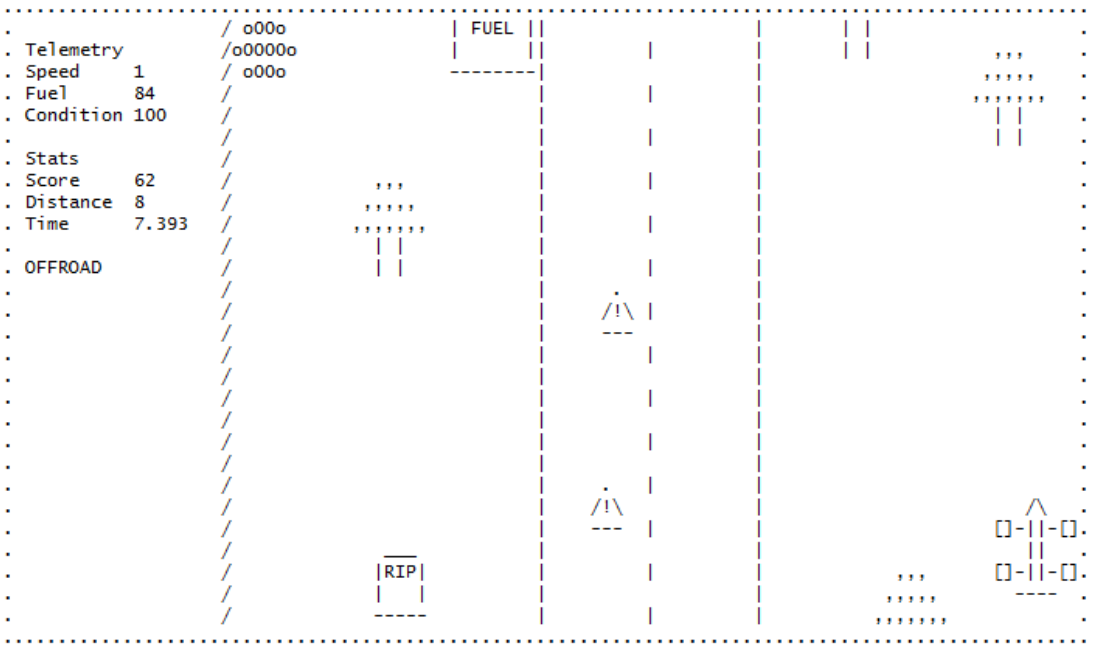

Figure 12: Result of holding down 'd' when next to the right border

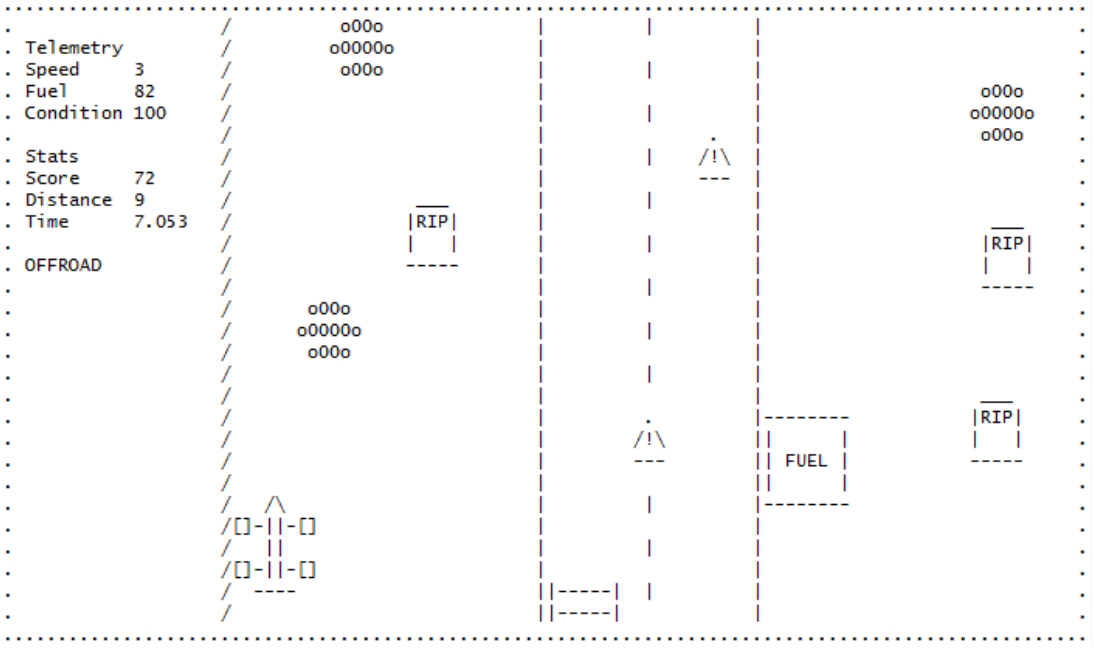

Figure 13: Result of holding down 'a' when next to the left border

Acceleration and Speed

The car can accelerates and decelerates with the 'w' and 's' keys. The speed can never go negative or higher than 10. When the car is offroad (indicated by the *OFFROAD* warning in the dashboard) the speed is limited to a maximum of 3.

Globals

```
// zombiemountain.h
#define INPUT_ACCELERATE   'w'
```

The character input to accelerate the car.

```
// zombiemountain.h
#define INPUT_DECELERATE   's'
```

The character input to decelerate the car.

```
// zombiemountain.h
#define MAX_SPEED      10
```

The maximum speed the car can reach.

```
// zombiemountain.h
#define MAX_SPEED_OFFROAD 3
```

The maximum speed the car can reach while offroad.

```
// zombiemountain.h
#define SPEED_INTERVAL   87
```

Used to set the reset time for *speed_timer*.

```
// zombiemountain.h
#define LOOP_INTERVAL 17
```

Used with *SPEED_INTERVAL* and *speed_timer* to decide when to increment *speed_ctr*.

```
// zombiemountain.h
int speed;
```

The speed of the player, this affects how fast the obstacles scroll down.

```
// zombiemountain.h
int speed_ctr;
```

Is compared with the current speed to decide when to update the main game logic (increasing distance, making obstacles scroll, etc.).

```
// zombiemountain.h
timer_id speed_timer;
```

This timer controls when *speed_ctr* is increased.

Functions

```
// main.c
bool update_speed_ctr();
```

Updates the *speed_ctr* if the timer has passed a certain limit. Will return true if it is time to update the game logic.

```
// main.c
void update_game_screen();
```

Handles the updating of the game logic when necessary.

```
// main.c
void handle_speed_input(int key);
```

Called by *handle_input()* when the input is detected to be aceleration or deceleration. Will check if the new speed will fit beside the bounds outlined at the start of this section and then adjust the *speed* variable accordingly.

Testing

Speed does not go below zero

When the game starts, the following keys are pressed "s-s-s". The result can be seen in Figure 14.

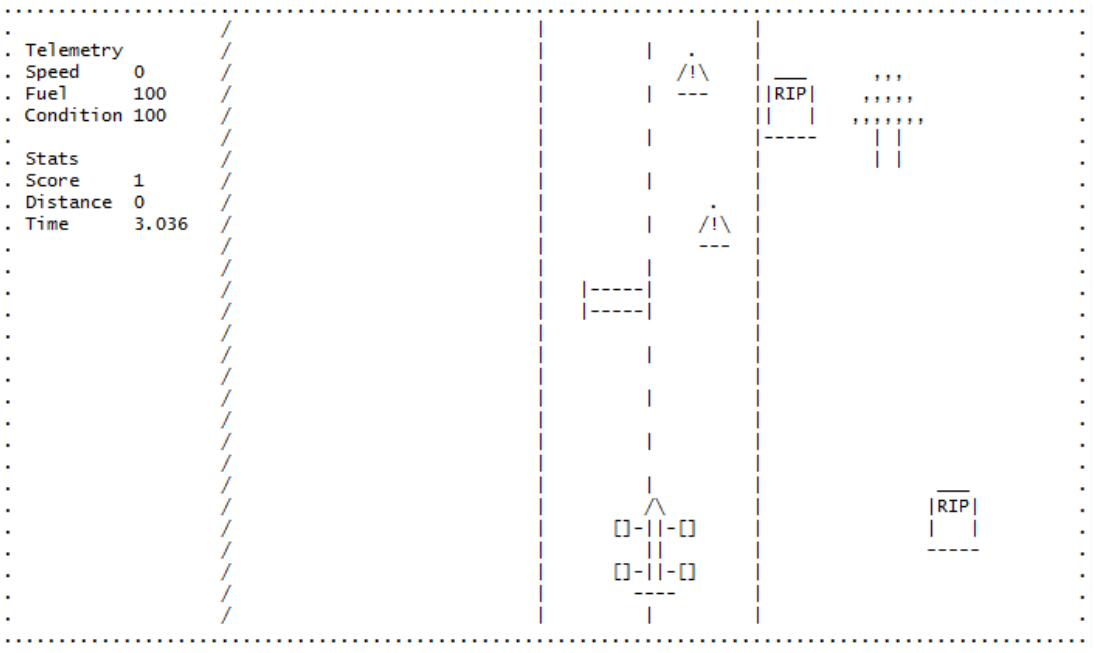

Figure 14: Decelerating when speed is already zero

This also shows that the fuel and distance are not modified while the car is stationary but the time keeps increasing. This means the function *update_game_screen()* is doing it's job of choosing which parts of the game logic to update.

Distance covered with different speeds

Figure 15 shows the car travels 4 units in three seconds when speed is set to 5. Straight after, the speed was increased to 10 and Figure 16 shows the car travelled 10 units in three seconds.

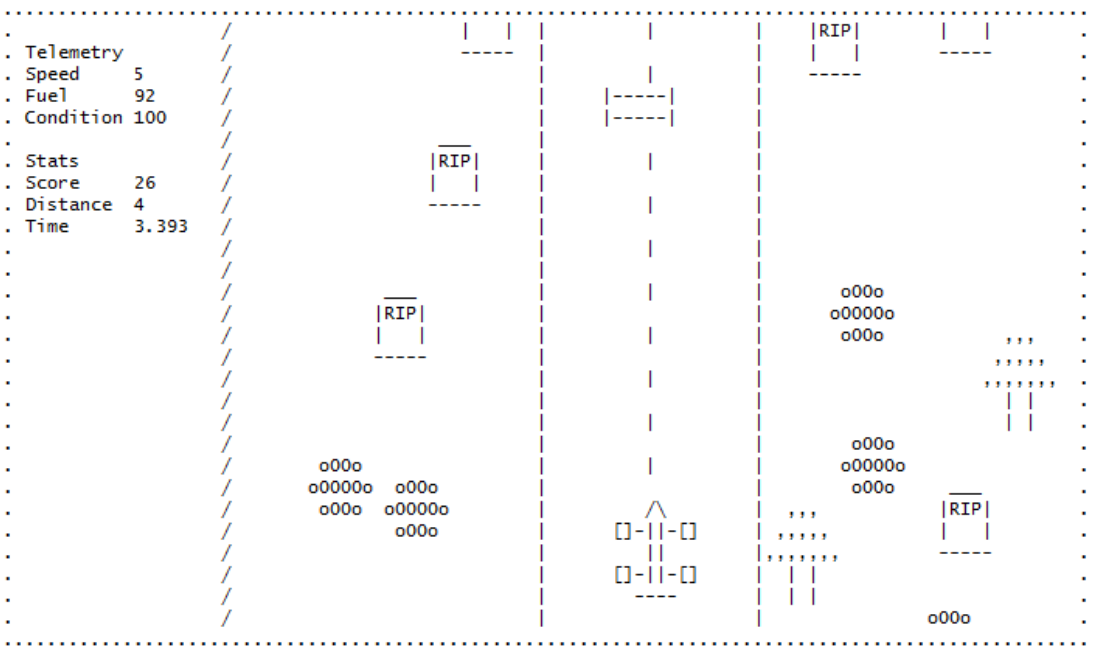

Figure 15: Distance covered at speed 5 after three seconds

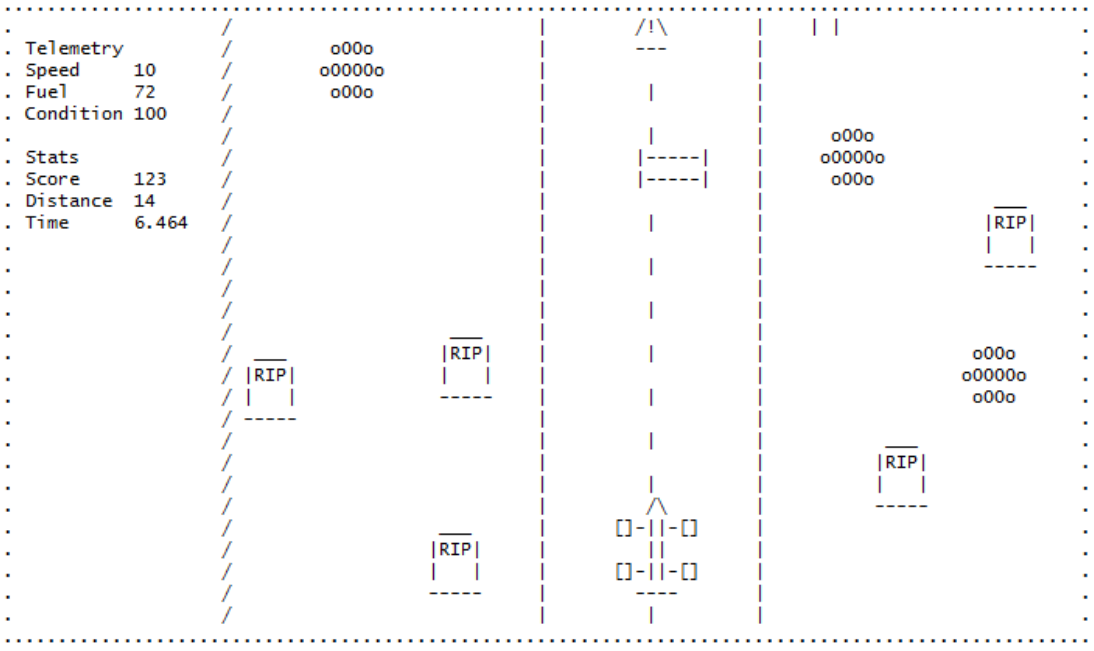

Figure 16: Distance covered at speed 10 after three seconds

Speed does not go above 10

This test was set up by having the car reach speed 10. Afterwards, the following keys were pressed "w,w,w,w" to verify that the speed wouldn't go above 10.

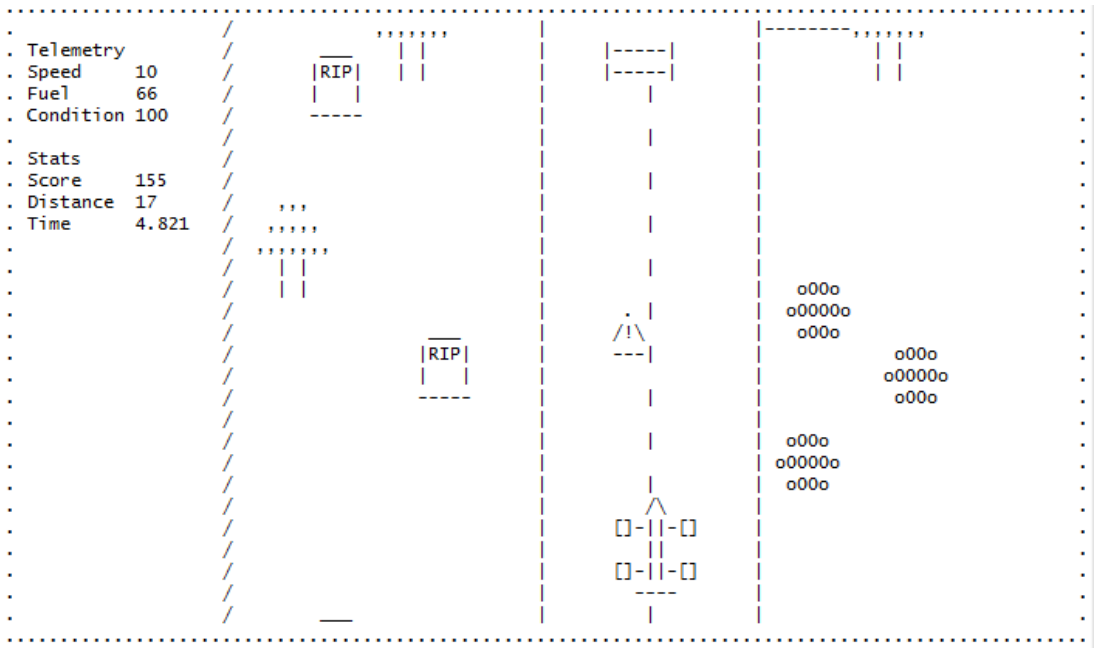

Figure 17: Accelerating when speed is already 10

Scenery and Obstacles

In this game, obstacles can be separated into three categories

- Terrain (spawns offroad)
- Road Hazards (limited to the road)
- Fuel Depots

Terrain and Road Hazards behave the same and are only separated by where they can spawn. Fuel depots have additional functionality and are covered in the section *Fuel Depot*.

Globals

```
// obstacles.h
int max_terrain_obs;
```

The maximum amount of terrain obstacles that can appear at one time. This is largely dependant on the screen size.

```
// obstacles.h
sprite_id *terrain;
```

An array that holds the ids of all sprites representing terrain.

```
// obstacles.h
int max_hazards;
```

The maximum amount of road hazards that can appear at one time.

```
// obstacles.h
sprite_id *hazards;
```

An array that holds the ids of all sprites representing road hazards.

Functions

```
// obstacles.c
void init_obs();
```

Allocate the required memory to the arrays which will hold all obstacles. Will also calculate the maximum number of obstacles that can appear in one go.

```
// obstacles.c
void setup_obs();
```

Calls all of the required setup functions for obstacles, road, fuel station and the finish line.

```
// obstacles.c
void setup_terrain();
```

Fills the terrain array with sprites and makes sure none are spawned on top of each other.

```
// obstacles.c
void terrain_create(int index);
```

Called by *setup_terrain()*. Chooses a type of terrain and a random valid location for it. Then proceeds to add the sprite id of that terrain to the appropriate array.

```
// obstacles.c
void terrain_reset(int index);
```

Moves the terrain corresponding to the index a randomised distance above the screen. The type of terrain and it's location will also be randomised. Nothing will happen if it collides with another obstacle so this function should be called again in the next game tick (done in *update_game_screen()*.

```
// obstacles.c
void update_terrain();
```

Steps all of the terrain sprites in the terrain array and then checks if any have gone out of bounds below the screen. Will then attempt to reset the terrain with *terrain_reset()*.

```
// obstacles.c
void setup_hazards()
```

Fills the hazards array with sprites and makes sure none are spawned on top of each other.

```
// obstacles.c
void hazard_create(int index)
```

Called by *setup_hazards()*. Chooses a type of hazard and a random valid location for it. Then proceeds to add the sprite id of that hazard to the appropriate array.

```
// obstacles.c
void hazard_reset(int index)
```

Moves the hazard corresponding to the index given a randomised distance above the screen. The type of hazard and it's location will also be randomised. Nothing will happen if it collides with another obstacle so this function should be called again in the next game tick (done in *update_game_screen()*. The hazard and terrain setup, update and reset functions are similar but need to be separated due to different arrays being used and both having different limitations on where they can be spawned.

```
// obstacles.c
void update_hazards();
```

Steps all of the hazard sprites in the hazards array and then checks if any have gone out of bounds below the screen. Will then attempt to reset the hazard with *hazard_reset()*.

```
// obstacles.c
void update_obs();
```

Call all of the update functions for each different type of obstacles.

```
// obstacles.c
void draw_obs();
```

Call all of the draw functions for each different type of obstacles.

```
// obstacles.c
void draw_terrain();
```

Call *sprite_draw()* for each terrain in the terrain array.

```
// obstacles.c
void draw_hazards();
```

Call *sprite_draw()* for each hazard in the hazards array.

Testing

Obstacles scroll at intermediate speed

After the game is started, the gravestone in the top-right in Figure 18 is taken as a reference point. The car is then accelerated to a speed of 5 the time it takes for the reference to reach the bottom of the screen shown in Figure 19 is found to be about 4 seconds.

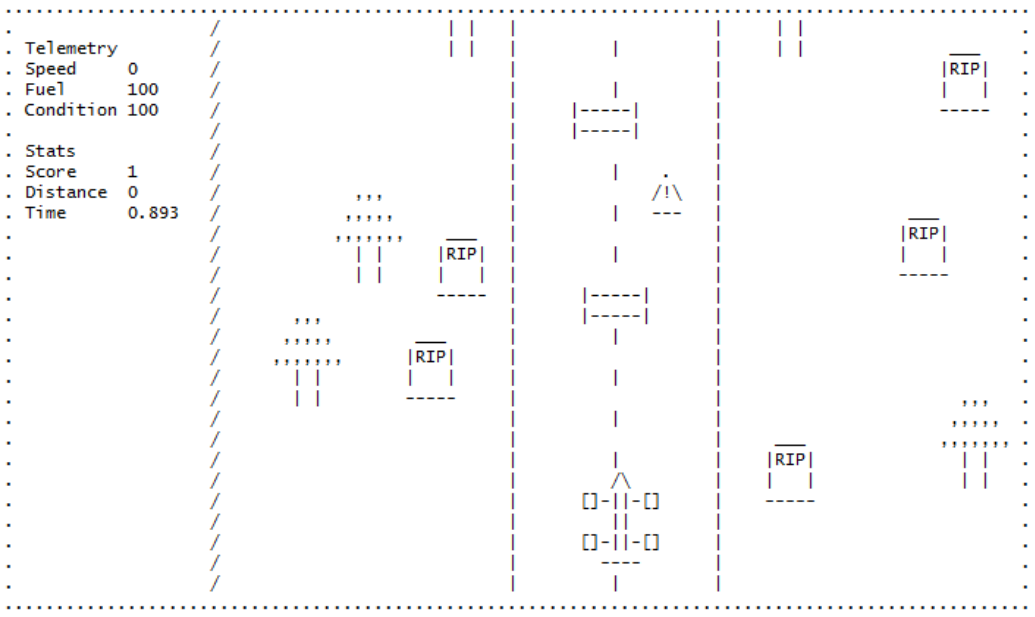

Figure 18: Calculating time for scenery to scroll past (top-right gravestone is reference point)

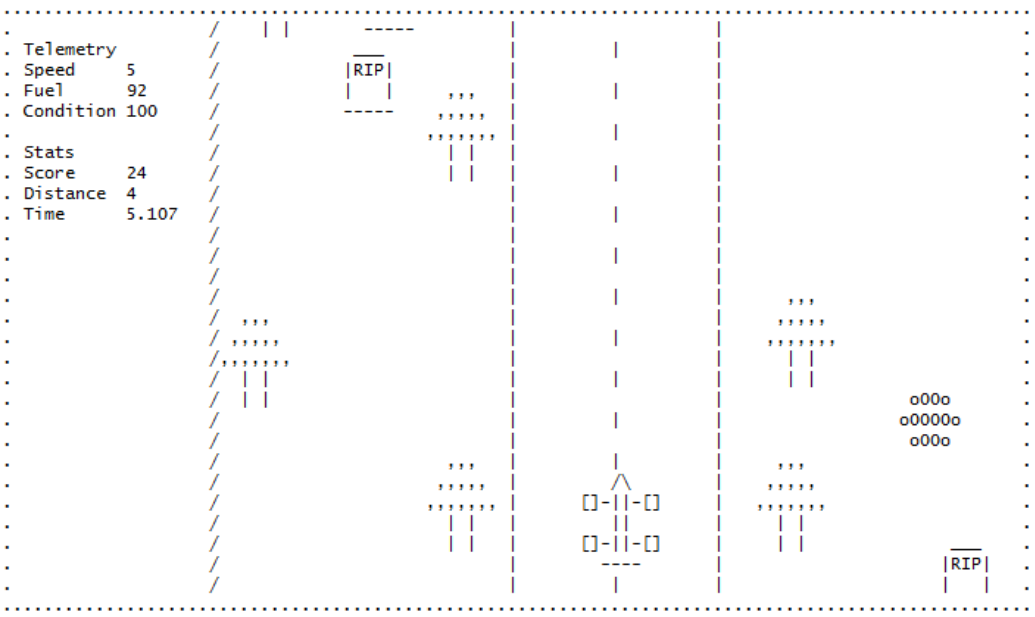

Figure 19: Calculating time for scenery to scroll past (reference gravestone scrolling out of view)

Testing

Obstacles scroll at max speed

The acceleration input is pressed until the car reaches max speed. The boulder at the top right of the screen is taken as a reference point. Figure 21 shows that it took 1 seconds for the boulder to reach the bottom of the screen. Both tests also show scenery scrolling in, middle and scrolling out in the screen.

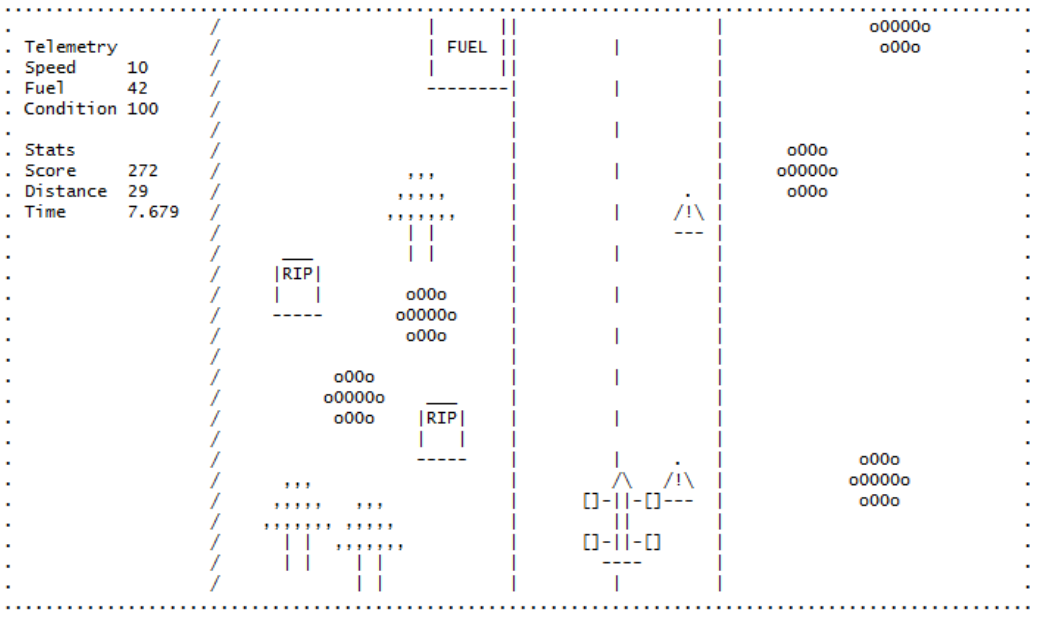

Figure 20: Calculating time for scenery to scroll past (top-right boulder is reference point)

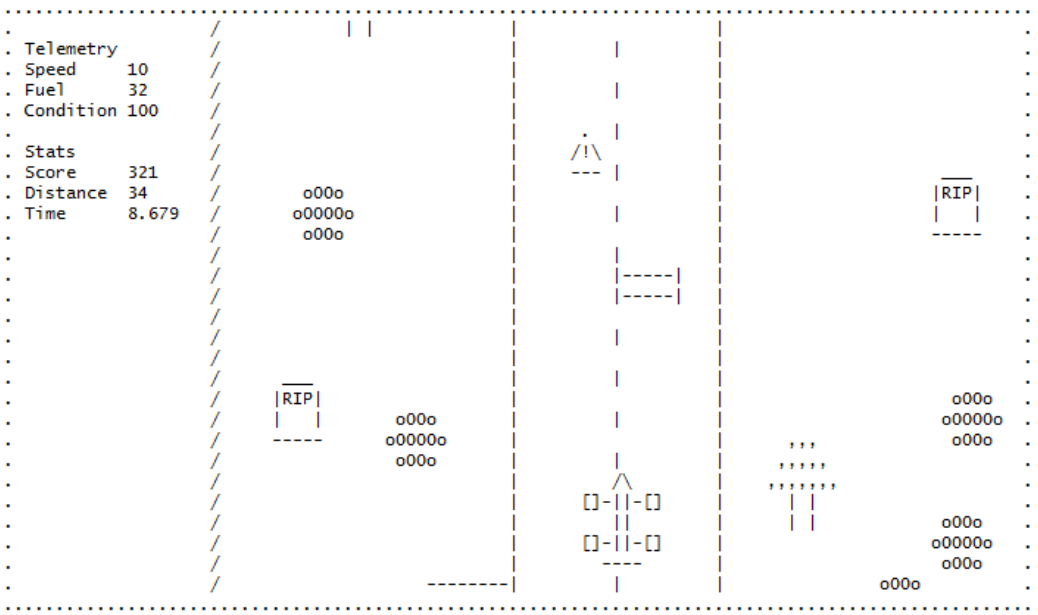

Figure 21: Calculating time for scenery to scroll past (only top can reference boulder can be seen)

Fuel Depot

Fuel

Distance Travelled

Collision

Game Over Dialogue

Part B - Highscore Screen

References