

CAB202 - Microprocessors and Digital Systems

Assignment 1

Pedro Alves (n9424342)

April 21, 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	24
Fuel	27
Distance Travelled	32
Collision	33
Game Over Dialogue	34
Part B - Highscore Screen	35
References	36

Executive Summary

Program Overview

Things to talk about
change_state

Border

The border is simply a rectangle that is drawn on the edge of the terminal. It supports every terminal size. The `draw_borders()` function 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

```
// 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

```
// 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

```
.....
.
. Screen Width:  80
. Screen Height: 24          Race to Zombie Mountain
.
.
.
.
.
. INSTRUCTIONS                      CONTROLS
. Reach the finish line           a/d : Move Left/Right
. Collisions reduce car condition w/s : Accelerate/Decelerate
. Game over if car condition is 0,
. collides with fuel station or
. runs out of fuel
. Drive with low speed next to fuel station to refuel
.
.
.
.
. Press any key to play...
.
.
. Pedro Alves - n9424342
.
.....
```

Figure 2: The border with screen dimensions of 80x24

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

```
// zombiountain.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.

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

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

```
// zombiountain.h
int speed;
```

The current speed of the player.

```
// zombiountain.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.

```
// zombiountain.h
int distance_travelled;
```

The distance travelled since the start of the game.

```
// zombiountain.h
double game_start_time;
```

The time in milliseconds that the game started.

```
// zombiountain.h
timer_id refuel_timer;
```

A timer that is set when the car starts refuelling.

Functions

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


If the car is offroad or refuelling, a relevant warning will also be drawn. Additionally for refuelling, will display how long until it is finished.

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

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*.

Stats are modified with gameplay

```
. /
. Telemetry //
. Speed 0 //
. Fuel 100 //
. Condition 100 //
. Stats //
. Score 1 //
. Distance 0 //
. Time 0.465 //
. //
. //
. //
. //
. //
. //
. //
. //
. //
. //
. //
. //
. //
```

|

./!\
./!\
./!\

^
[]-
[]-

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.

Page 9 of 36

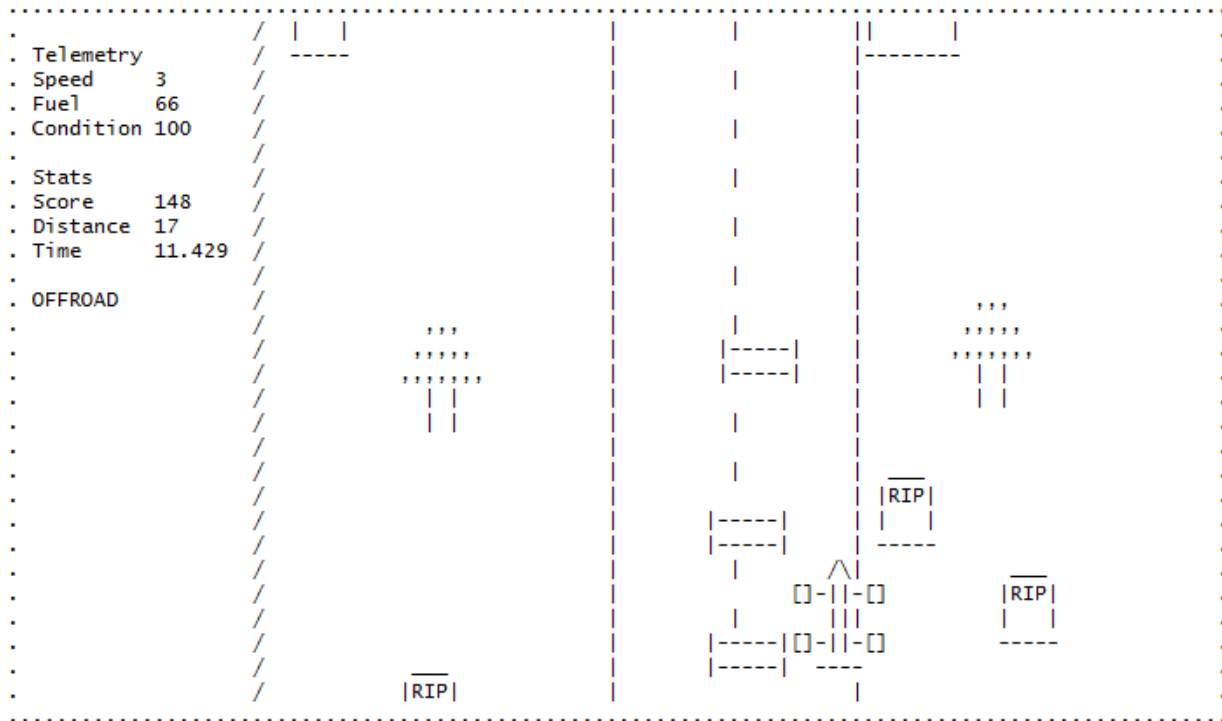


Figure 6: A screenshot of the same game sessions as the figure above but 11 seconds into gameplay

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%.

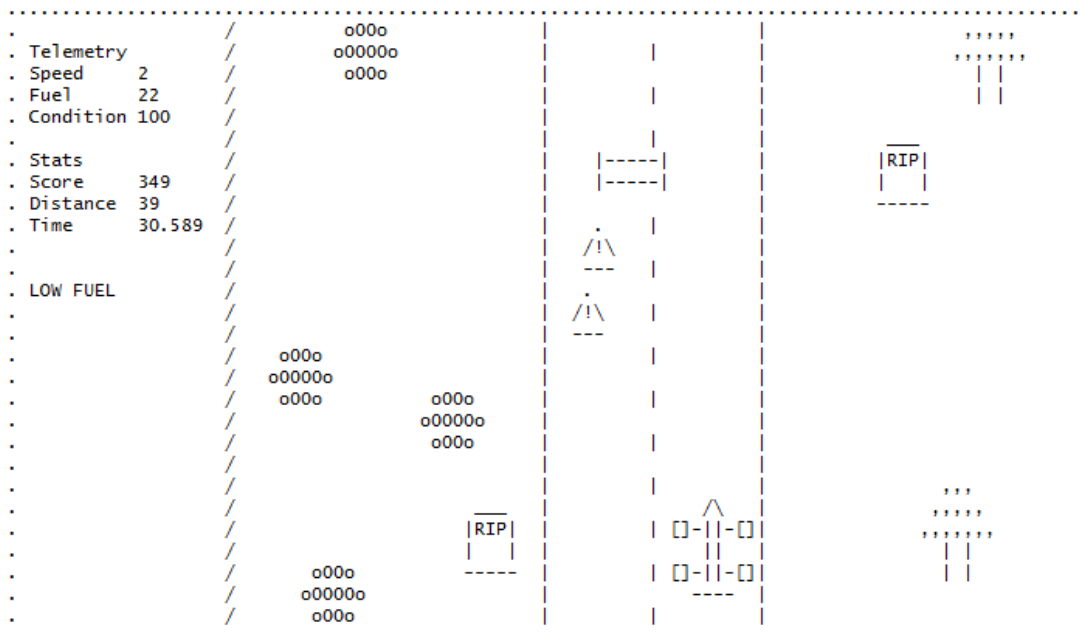


Figure 7: Low Fuel warning appearing when fuel is below 1/4 the maximum

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

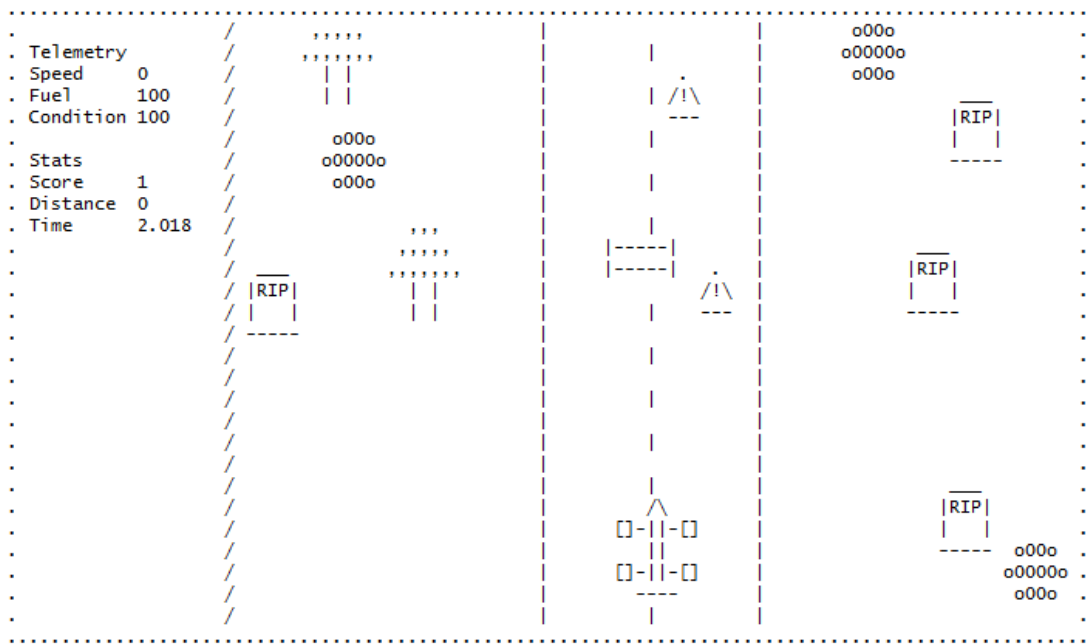


Figure 8: Car in the middle of the road with speed equal zero

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

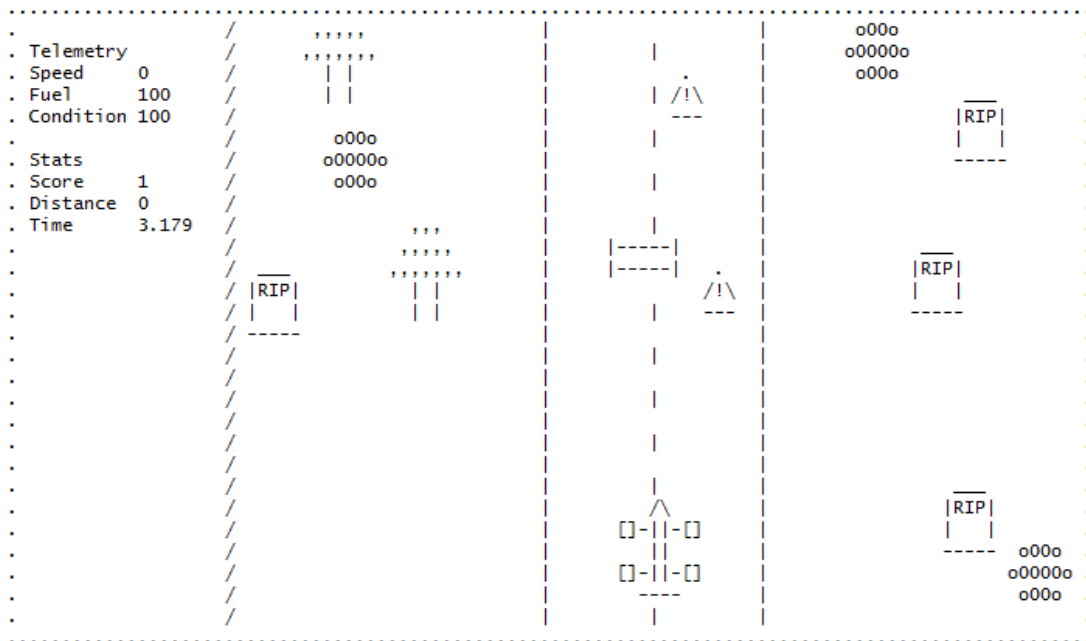
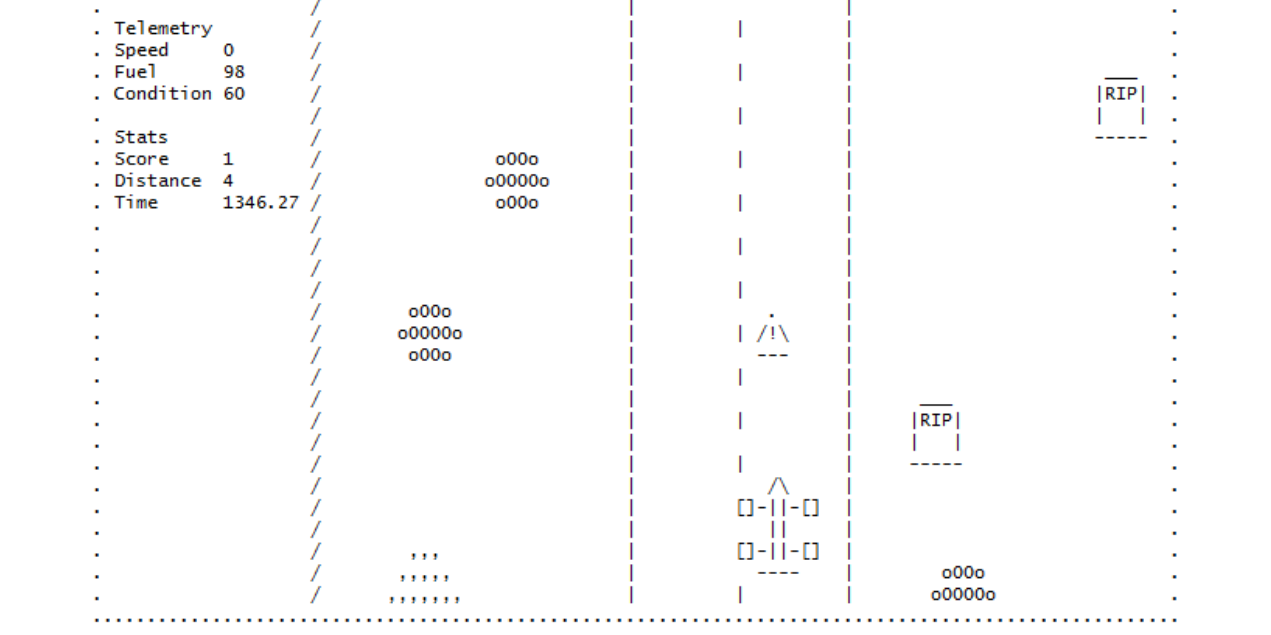
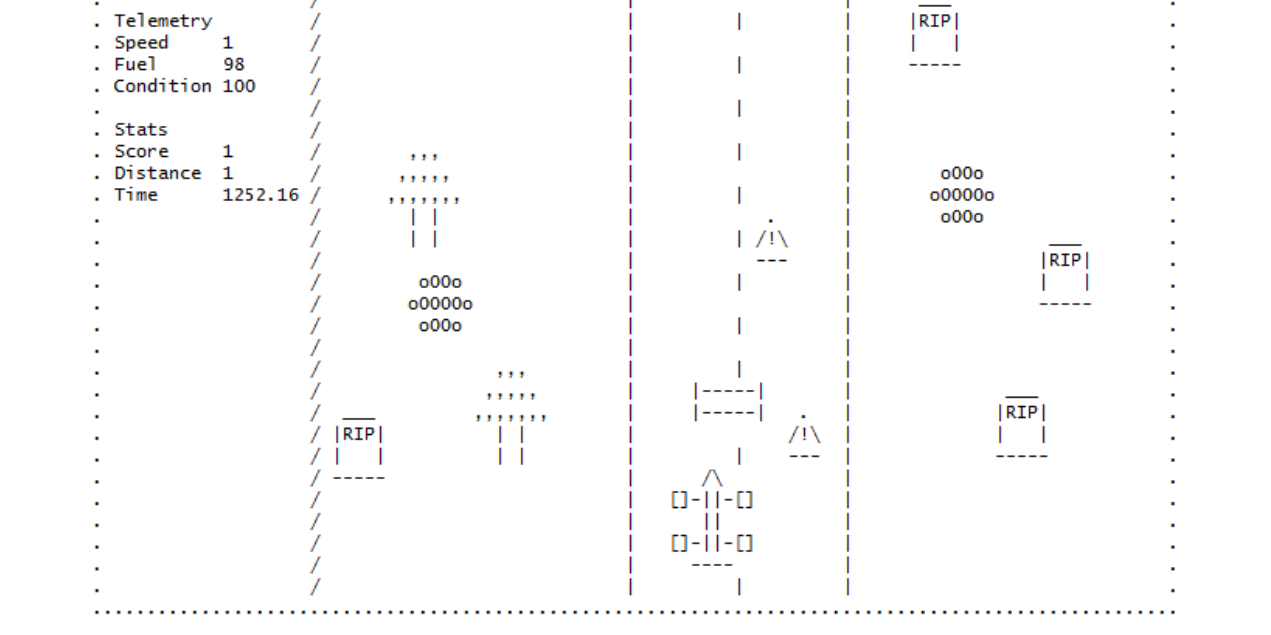


Figure 9: Results after attempting to move horizontally with speed equal zero

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



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'.

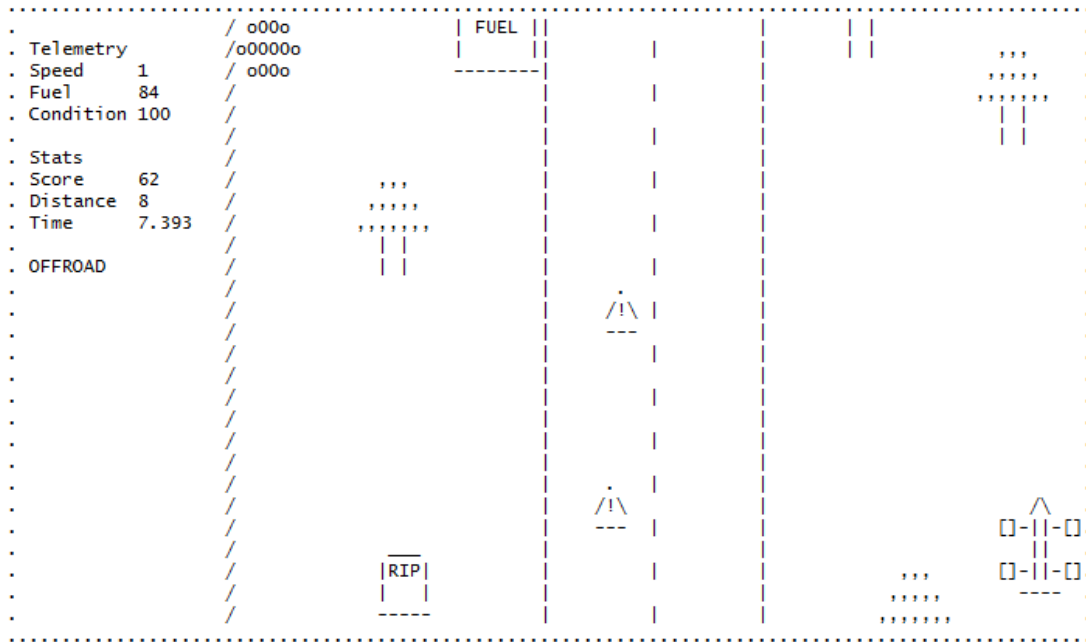


Figure 12: Result of holding down 'd' when next to the right border

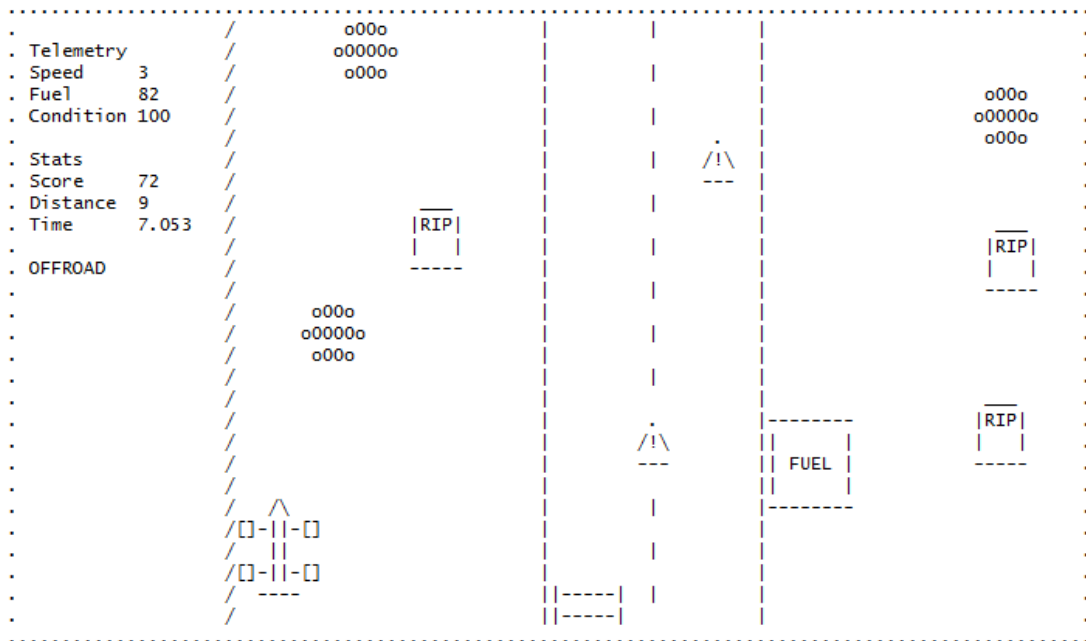


Figure 13: Result of holding down 'a' when next to the left border

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

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

The character input to accelerate the car.

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

The character input to decelerate the car.

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

The maximum speed the car can reach.

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

The maximum speed the car can reach while offroad.

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

Used to set the reset time for *speed_timer*.

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

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

```
// zombimountain.h
int speed;
```

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

```
// zombimountain.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.).

```
// zombimountain.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 acceleration 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.

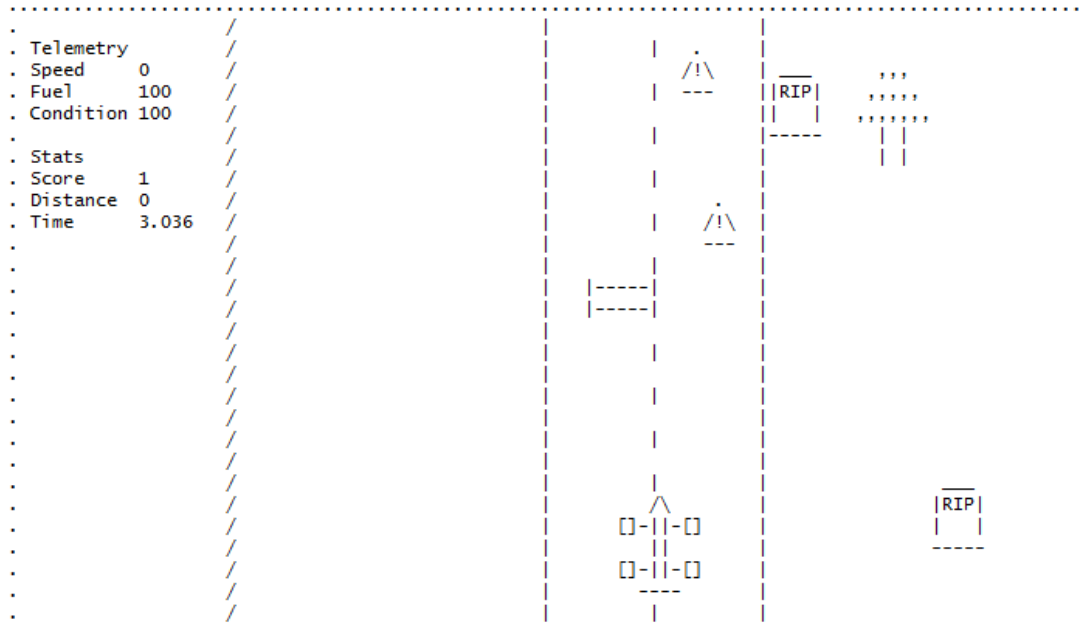


Figure 14: Decelerating when speed is already zero

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 its 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.

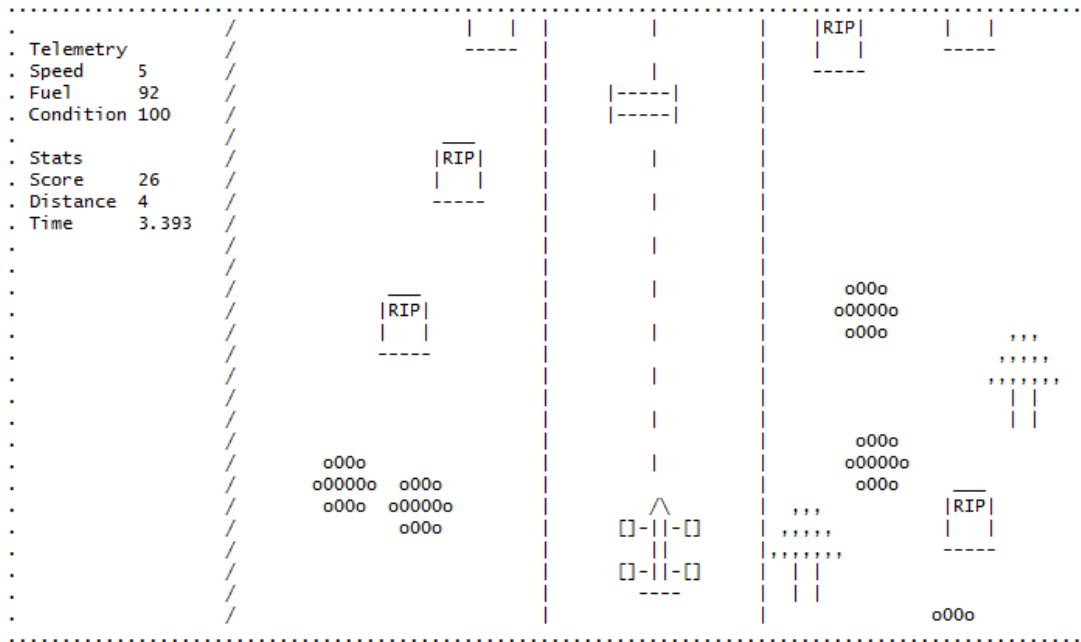


Figure 15: Distance covered at speed 5 after three seconds

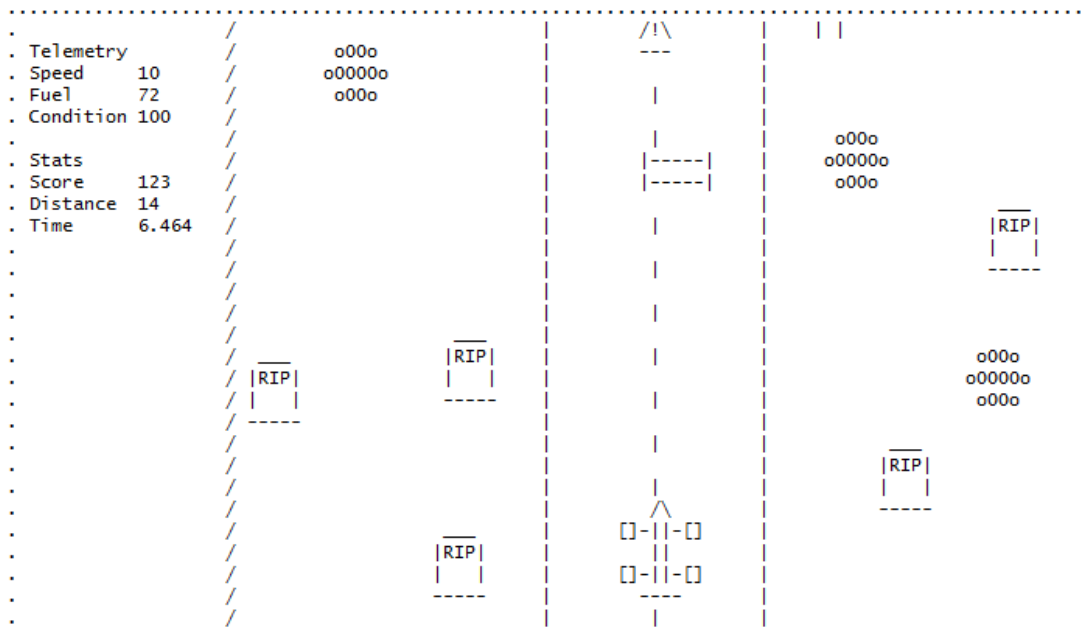


Figure 16: Distance covered at speed 10 after three seconds

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.

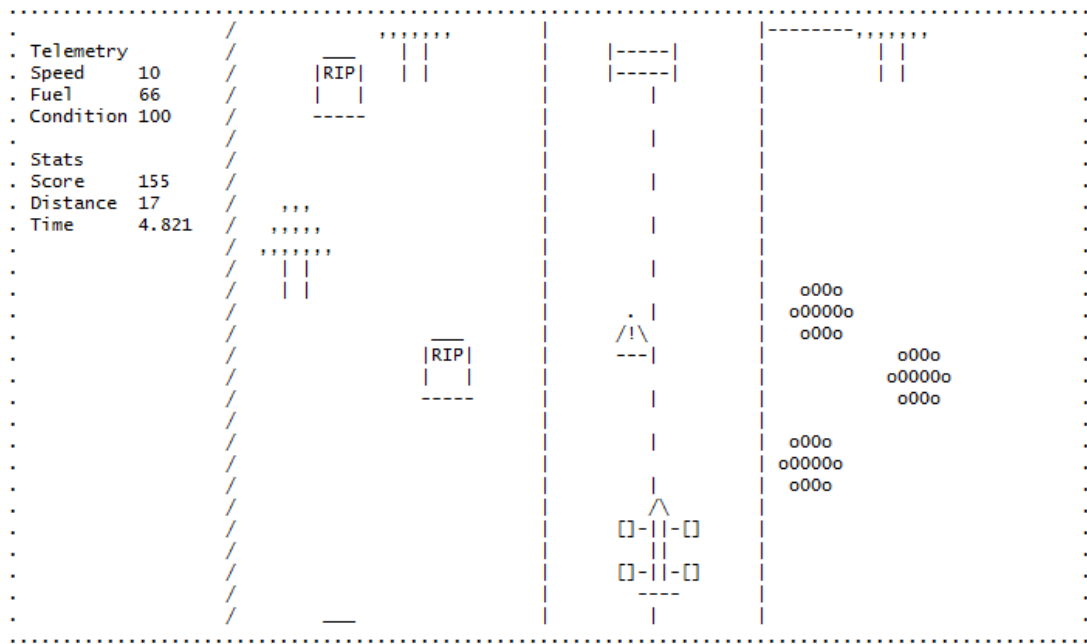


Figure 17: Accelerating when speed is already 10

Scenery and Obstacles

In this game, obstacles can be separated into three categories. Fuel depots have additional functionality and are covered in the section *Fuel Depot*.

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

Globals

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

The maximum amount of terrain obstacles that can appear at one time. 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.

```
// imagemngr.h
#define TERRAIN      0
#define HAZARD       1
```

Used to define the different types of obstacles.

```
// imagemngr.h
#define NUM_TERRAIN_TYPES 3
#define TERRAIN_BOULDER   0
#define TERRAIN_TREE      1
#define TERRAIN_GRAVE     2
```

Used to differentiate between the different types of terrain.

```
// imagemngr.h
#define NUM_HAZARD_TYPES 2
#define HAZARD_SPIKES    0
#define HAZARD_TRIANGLE  1
```

Used to differentiate between the different types of road hazards.

```
// imagemngr.c
char* terrain_image[NUM_TERRAIN_TYPES];
int terrain_width[NUM_TERRAIN_TYPES];
int terrain_height[NUM_TERRAIN_TYPES];
char* hazards_image[NUM_HAZARD_TYPES];
int hazards_width[NUM_HAZARD_TYPES];
int hazards_height[NUM_HAZARD_TYPES];
```

Holds information about a sprite's bitmap for each different type of terrain and hazard.

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.

```
// imagemngr.c
void imagemngr_init();
```

Call the appropriate init function for each type of obstacle.

```
// imagemngr.c
void hazards_init();
```

Add all of the hazard bitmap information to the appropriate arrays.

```
// imagemngr.c
void terrain_init();
```

Add all of the terrain bitmap information to the appropriate arrays.

```
// imagemngr.c
void add_image(int id, char* image, int width, int height, int type);
```

Add the bitmap information of a specific sprite to the appropriate array (defined by the *type* variable).

```
// imagemngr.c
char* get_image(int id, int type, int* width, int* height);
```

Get the bitmap and its properties from the appropriate array defined by the *type* variable.

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.

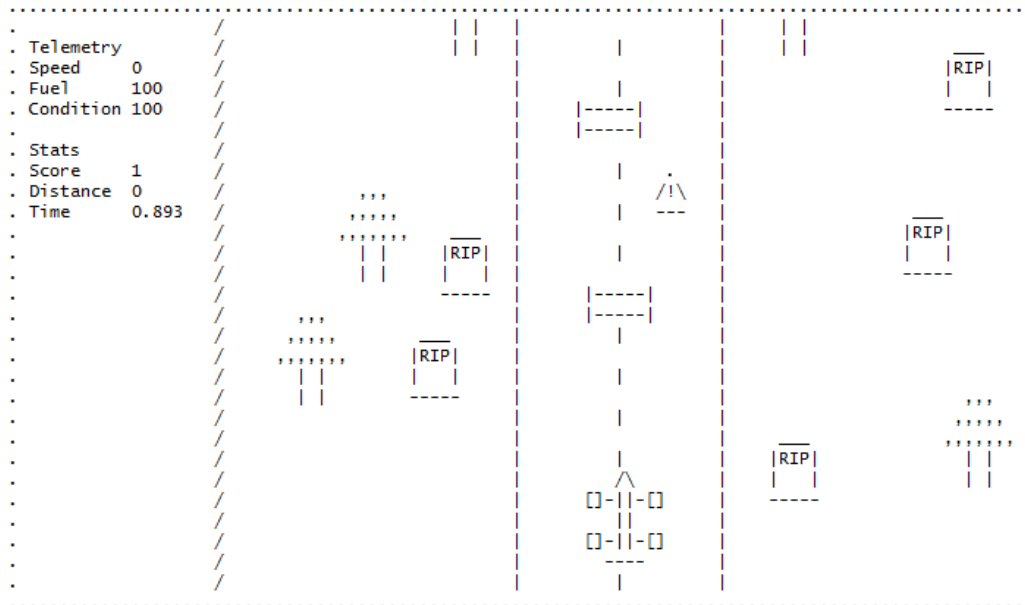


Figure 18: Calculating time for scenery to scroll past (top-right gravestone is reference point)

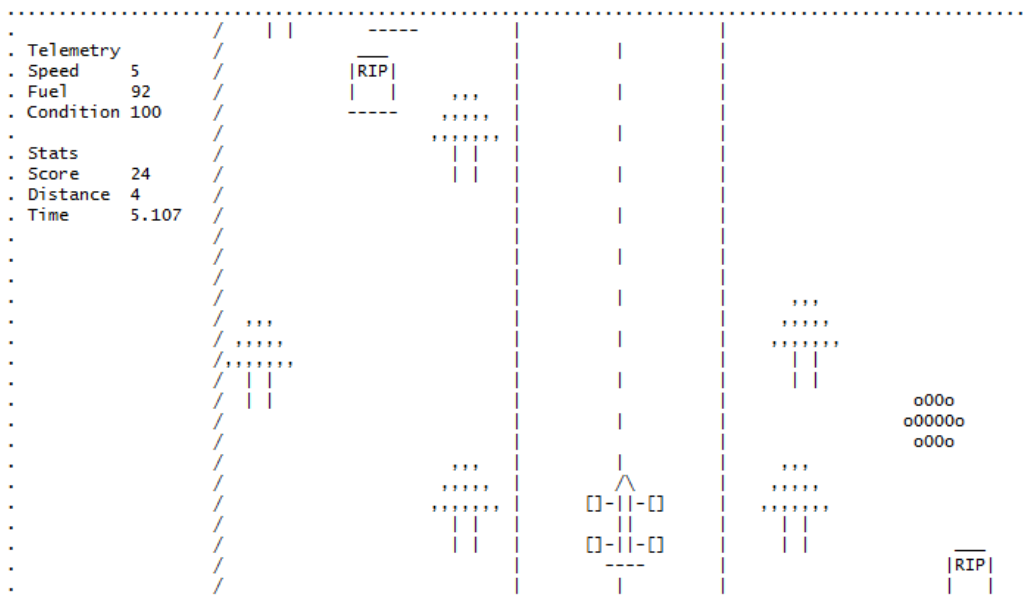


Figure 19: Calculating time for scenery to scroll past (reference gravestone scrolling out of view)

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 second for the boulder to reach the bottom of the screen. Both tests also show scenery scrolling in, middle and scrolling out in the screen.

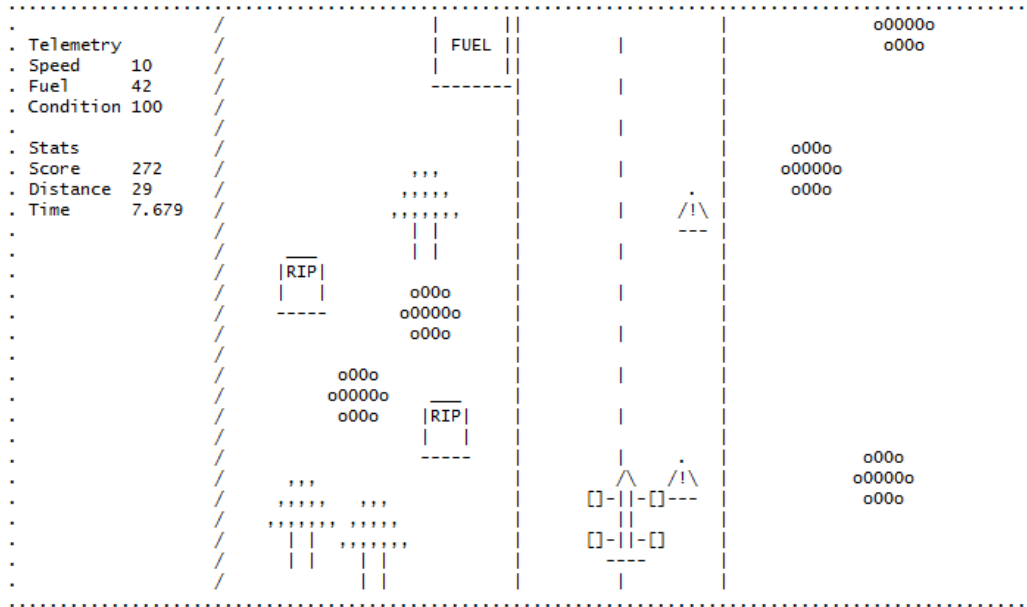


Figure 20: Calculating time for scenery to scroll past (top-right boulder is reference point)

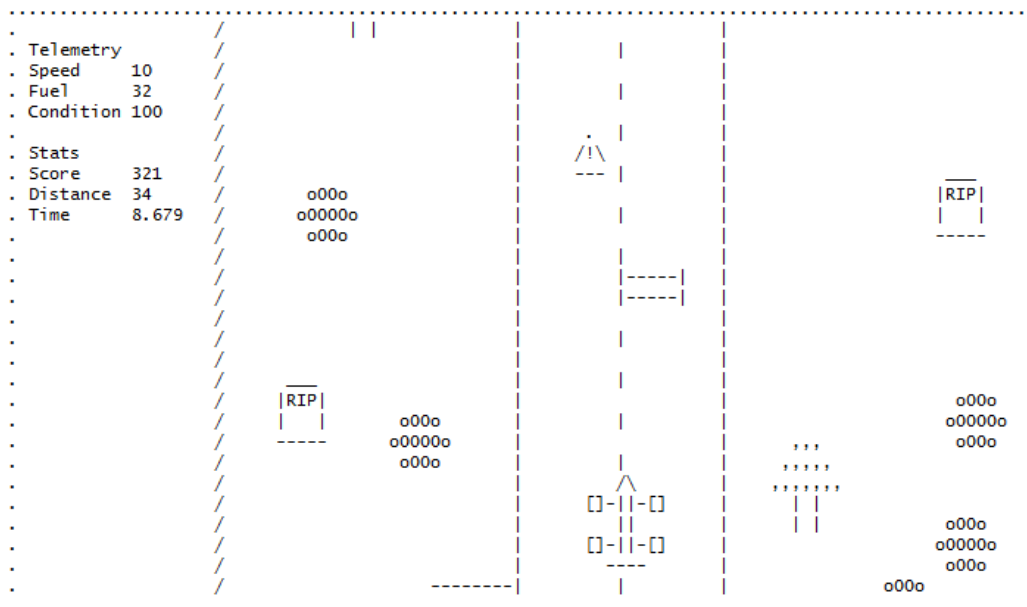


Figure 21: Calculating time for scenery to scroll past (only top can reference boulder can be seen)

Fuel Depot

The Fuel Depot is a type of obstacle that refuels the player's car if it is parked for 3 seconds directly next to it. To smooth out the gameplay, the player will automatically park the car if they're travelling at a speed of 2 or less. A collision with the fuel depot will immediately end the game for the player, regardless of the car's condition.

Globals

```
// obstacles.h
#define FUEL_STATION_DELAY_DIST 30
```

The minimum distance which the next fuel depot can appear after the old one leaves the playing area.

```
// obstacles.h
#define FUEL_STATION_VARIANCE 15
```

The variance in distance that the next fuel depot will spawn at. When we combine with *FUEL_STATION_DELAY_DIST*, we know that a fuel depot will spawn between 30 and 45 units of distance above the screen after the old depot leaves the playing area.

```
// obstacles.h
sprite_id fuel_station;
```

The sprite which represents the fuel depot.

Functions

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

Creates the fuel depot sprite and places it in its initial position. Will also randomly choose a random side of the road to place it in.

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

Will step the sprite for the fuel depot and then check if it went out of bounds. If it has, it will reset the fuel station to a random location (limited by the globals) above the playing area and a random side of the road. Will then reset any terrain that was in the way to avoid overlap.

Testing

Fuel Depot scrolls at intermediate speed

The game is played with the car kept at a constant speed of 5. Figures 22 and 23 show that it took the fuel depot about three seconds to scroll past the playing area.

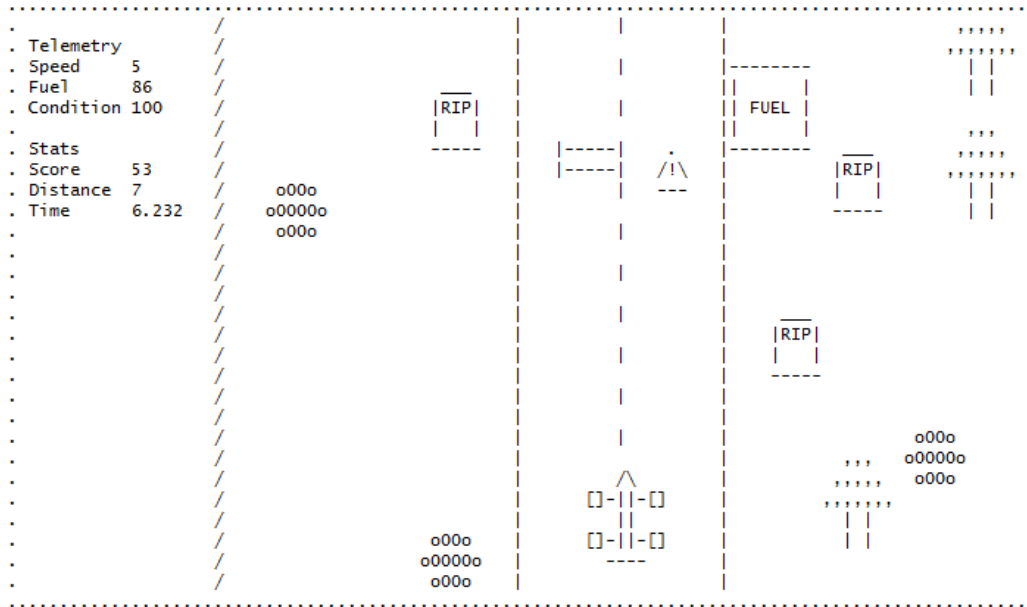


Figure 22: Calculating time for fuel depot to scroll past at speed of 5

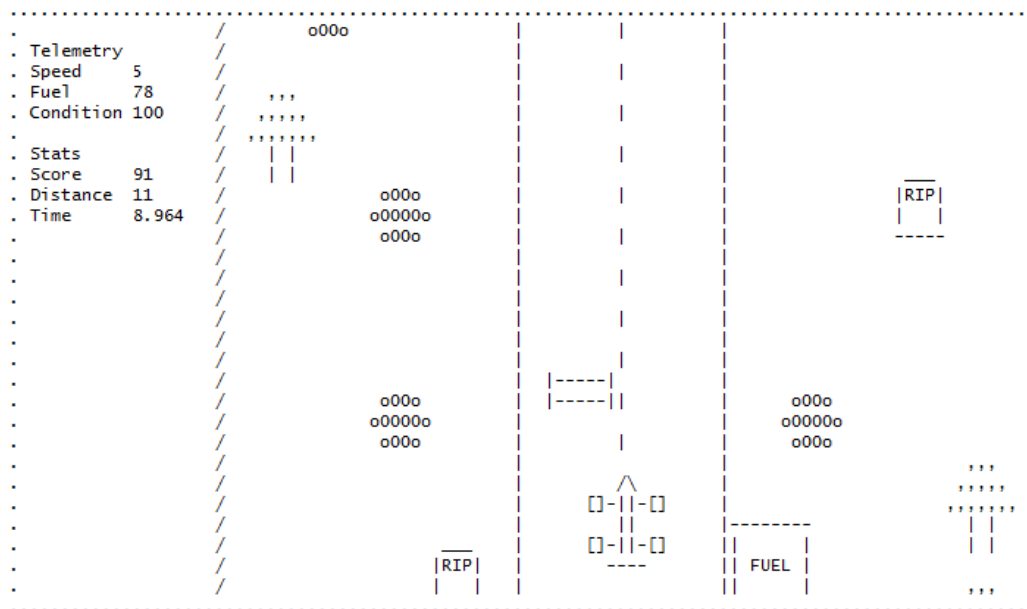


Figure 23: Calculating time for fuel depot to scroll past at speed of 5

Fuel Depot scrolls at maximum speed

The game is played with the car kept at a constant speed of 10. Figures 24 and 25 show that it took the fuel depot about one second to scroll past the playing area. Both of these tests show that the fuel depot is scrolling at the same rate as the rest of the scenery and obstacles. As seen in Figures 12 and 13, the fuel depot can also spawn on any side of the road.

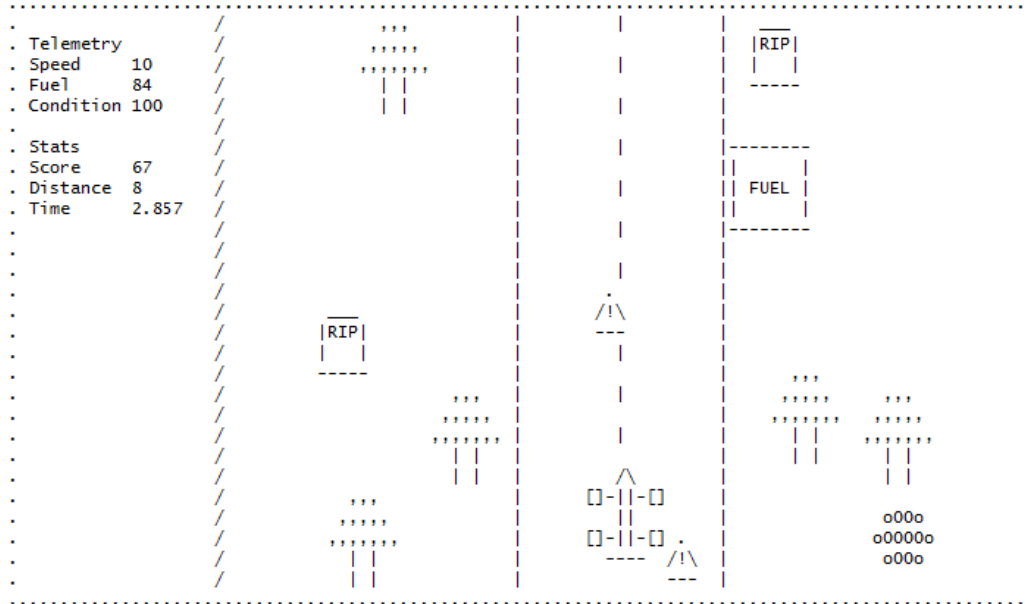


Figure 24: Calculating time for fuel depot to scroll past at speed of 5

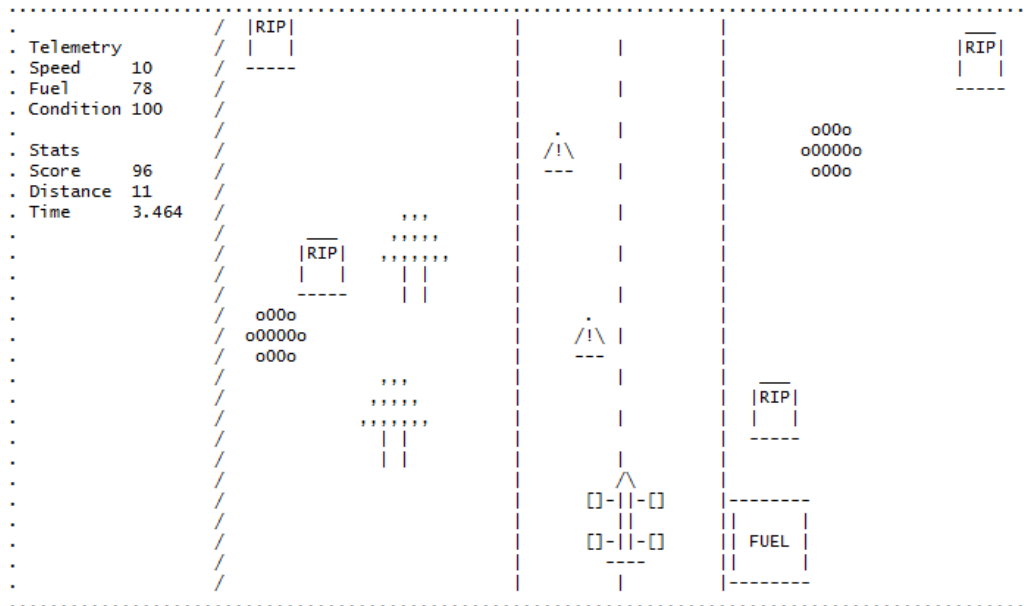


Figure 25: Calculating time for fuel depot to scroll past at speed of 10

Fuel

The car start with a fuel tank which decreases as it moves. The faster it moves, the faster the fuel is depleted. After parking next to a fuel depot for 3 seconds, the fuel tank is refilled to max. The fuel tank can also be refilled to max after a collision with an obstacle.

Globals

```
// zombimountain.h
int fuel;
```

The current amount of fuel available to the player.

```
// zombimountain.h
bool refuelling;
```

Represents whether the car is currently refuelling.

```
// zombimountain.h
timer_id refuel_timer;
```

A timer that is used to verify that the car has remained next to the fuel depot for 3 seconds.

Functions

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

Checks if the car meets all of the criteria to begin refuelling (next to a fuel station and travelling at a speed of 2 or less). Then switches the *refuelling* variable to true, sets speed to 0 and starts *refuel_timer*.

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

Called every tick of the game loop. If the car isn't already refuelling, call the *check_refuel()*. Otherwise it will make sure the car's speed has remained at 0. If the *refuel_timer* has expired, it'll refill the fuel tank and release the car at a speed of 1.

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

Will check if the fuel is above 0 and will give the game over message when the fuel tank is empty.

Testing

Fuel loss moving at intermediate speed

The car is kept at a constant speed of 5 for three seconds. Figures 26 and 27 show that the car loses 8 fuel units after 3 seconds at intermediate speed. Figure 14 also shows that the car doesn't lose fuel while remaining stationary.

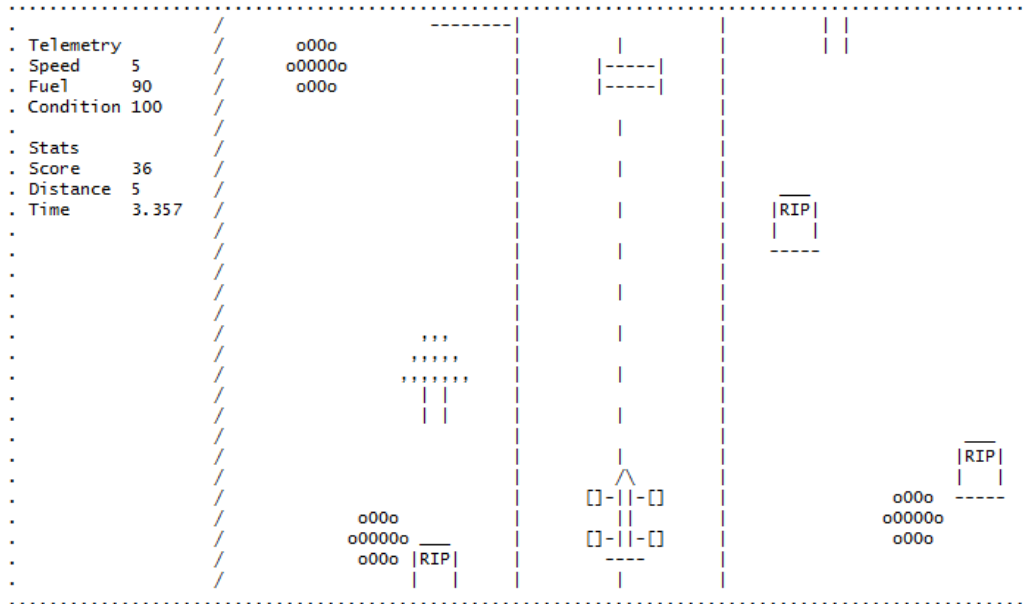


Figure 26: Calculating fuel loss at intermediate speed

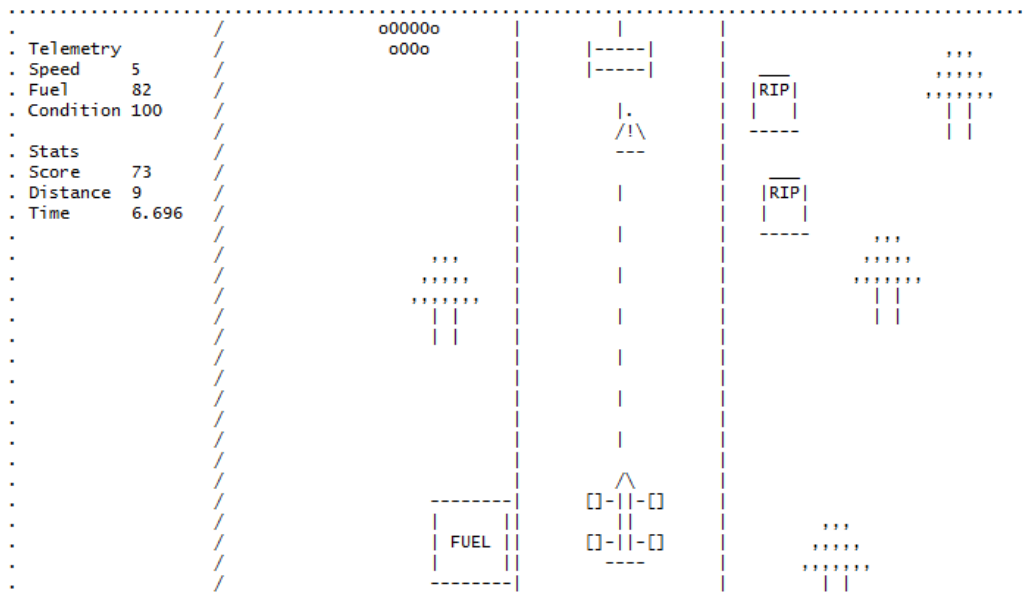


Figure 27: Calculating fuel loss at intermediate speed

Fuel loss moving at max speed

The car is kept at a constant speed of 10 for three seconds. Figures 28 and 29 show that the car loses 30 fuel units after 3 seconds at maximum speed.

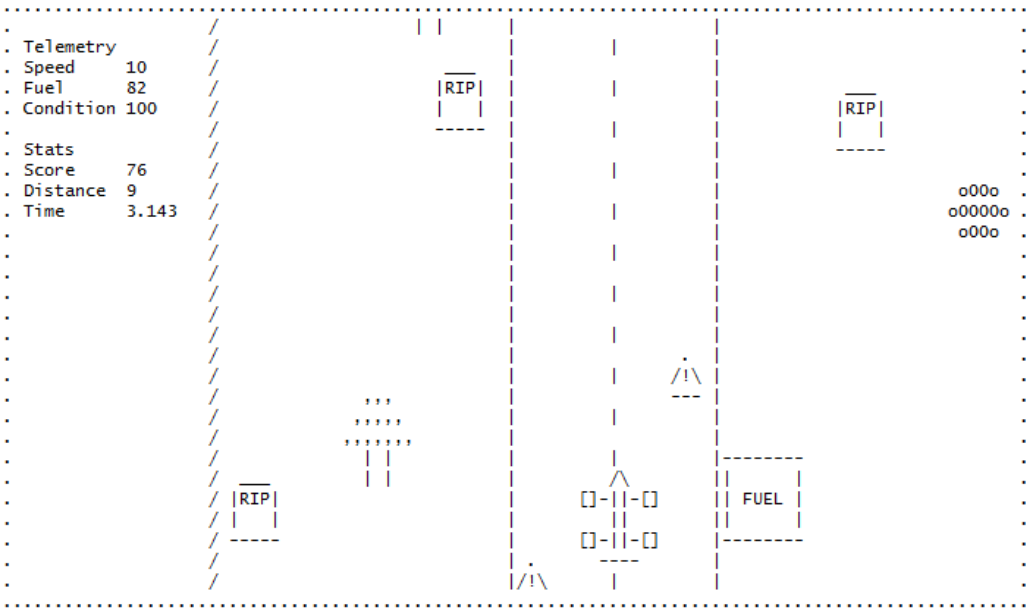


Figure 28: Calculating fuel loss at max speed

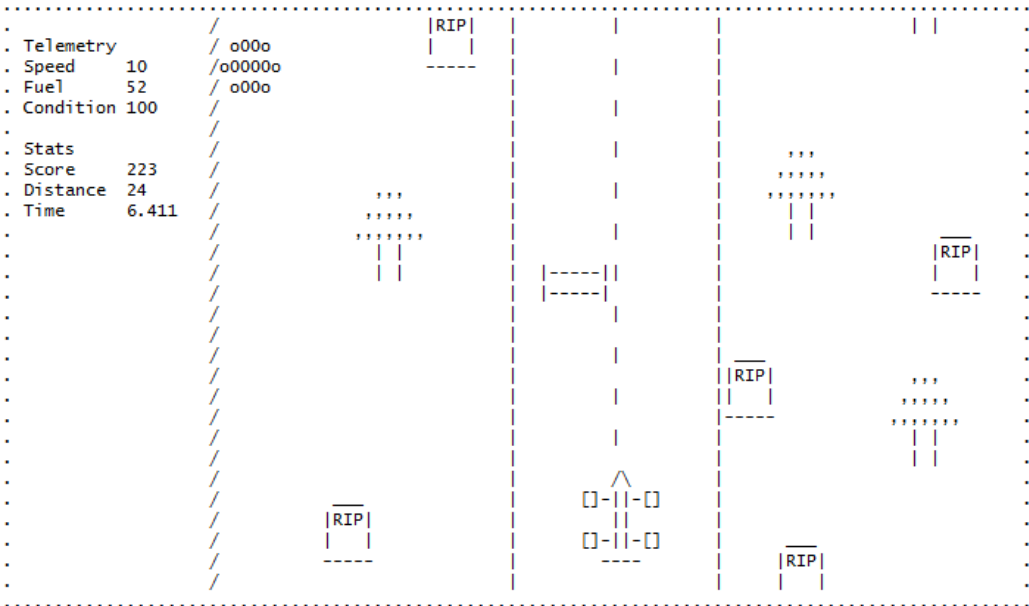


Figure 29: Calculating fuel loss at max speed

Refuelling

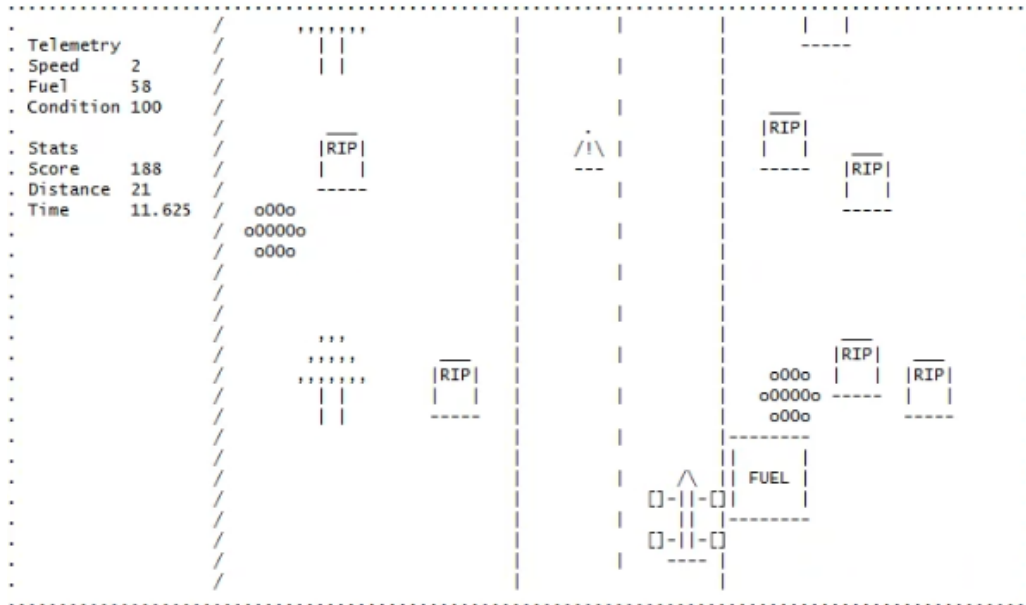


Figure 30: Car approaching the fuel station

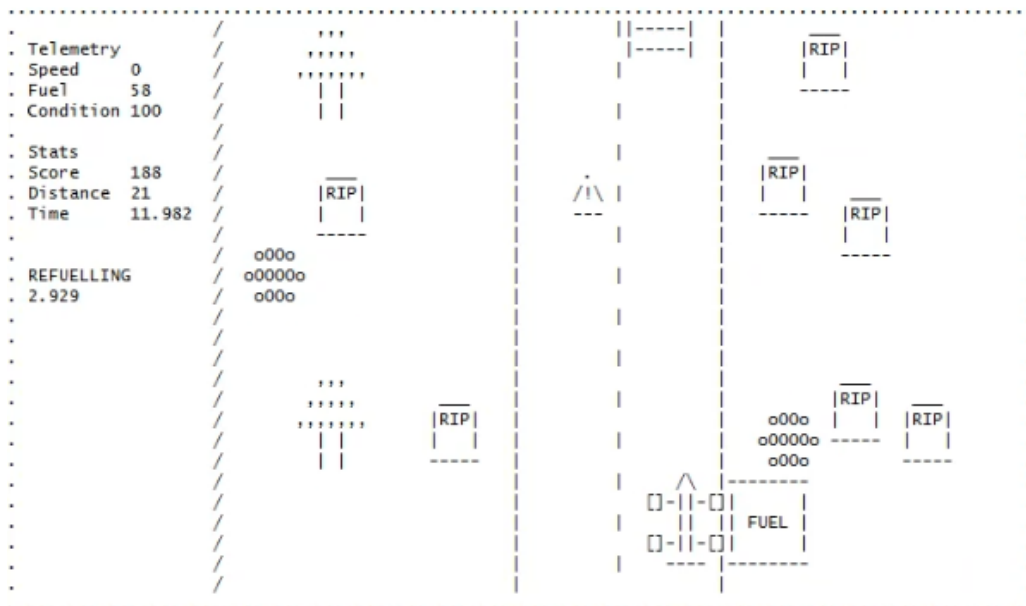


Figure 31: Car is stopped automatically and refuelling begins

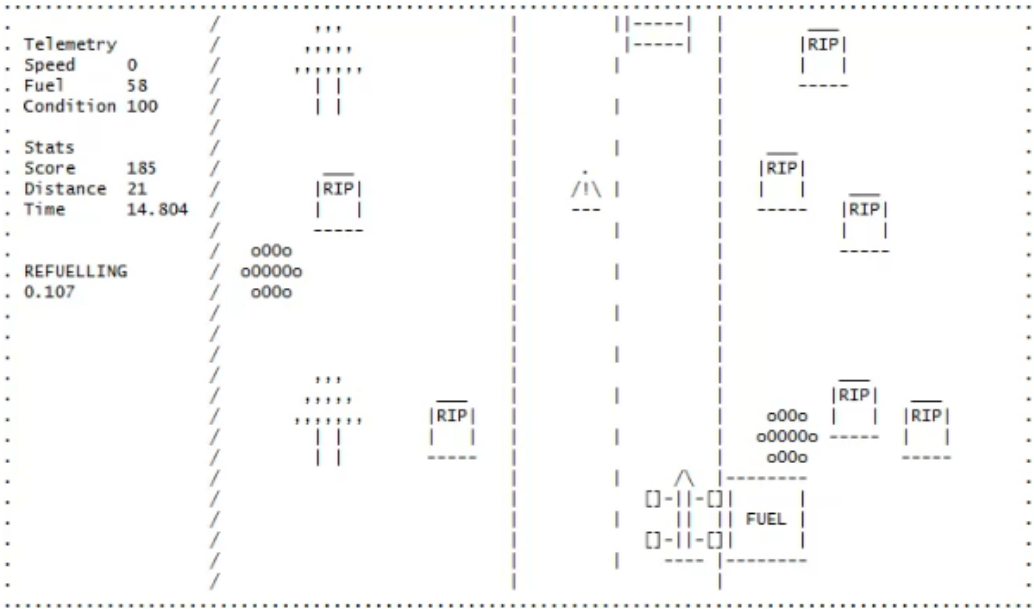


Figure 32: Time has passed and the player is waiting for refuel to finish

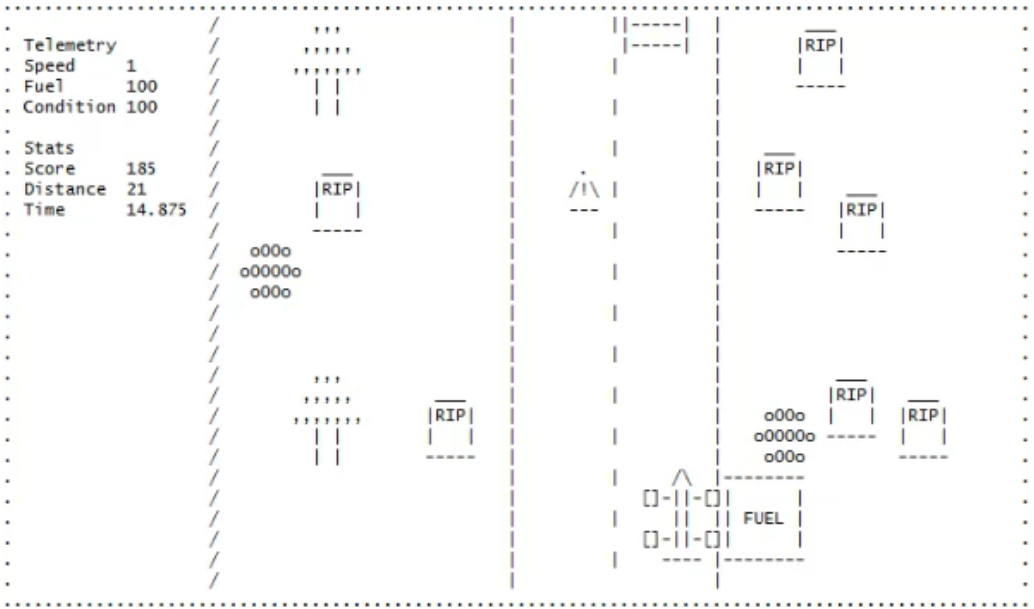


Figure 33: Refuelling ended, fuel tank is topped up and the car is released at a speed of 1

Distance Travelled

Collision

Game Over Dialogue

Part B - Highscore Screen

References