Code Companion Report

CAB202

Matthew Hutchison

n8548625

April 29,2018

Executive Summary

This assessed task required the creation of a simple arcade racer style game, called "Race to Zombie Mountain". Where the player controls a race car and attempts to speed towards the finish line, dodging scenery/obsticles and trying not to run out of fuel.

Most of the games basic functionality has been implimented in some shape or form. The resultent code has been submitted through the AMS portal and the submittion reference number is: *8298980f-0d35-4629-abc6-e2e53306572a*.

The following table is used in highlighting what state each of the tasks for this game where in upon submition:

- **Completed:** A completed state results in this task meeting all of the outlined criteria found in the assainmnet brief.
- **Almost:** Almost implies that it does one or more of the outlined criteria for this task. It can also imply that the current feature still requires work to improve overall efficiency, allowing for the task to perform flawlessly each time.
- **Started:** Started implies that the task has been outlined, global variables defined/initialised and sprites parameters (width,height, x position, y position, image) defined. It can also consist of functions that have been created, but do not currently work.
- **No Attempt:** No attempt implies that this task was not even attempted, in any shape or form.

GAME TASK COMPLETION STATUS				
TASK	COMPLETE	ALMOST (75%)	STARTED	NO ATTEMPT
Splash Screen	☑	☐	☐	☐
Boarder	☑	☐	☐	☐
Dashboard	☑	☐	☐	☐
Horizontal Movement	☑	☐	☐	☐
Acceleration and Speed	☑	☐	☐	☐
Scenery and Obstacles	☐	☑	☐	☐
Fuel Depot	☐	☐	☑	☐
Fuel	☑	☐	☐	☐
Distance Traveled	☑	☐	☐	☐
Collission	☐	☐	☑	☐
Game Over Dialogue	☑	☐	☐	☐

Building upon the foundations of the base game functionality, attempts were made too enhance the game, both through visual and efficiency based measures. the following list outlines briefly these features:

- **Animated Splash Screen:** The spalsh screen appears animated, cycling between two different splash screen images. Certain text elements appear static, whilst others like "press any key to continue" appear to flicker (appear and then dissapear). Such effects draw the players eye to these features, to help highlight, in this instance, how to continue on and begin the game.

- **Animated Race Car:** Similar the the above mentioned splash screen, the race car is also animated. Instead of always being animated, the race car's animation is dependent on the speed characteristic being greater than zero. While the speed is equal too or less, the sprite appears static in its visual appearance. Upon gaining momentum the race car sprite image, cycles between two different versions. The body of the race car appears static, but the wheels flicker, creating the illusion of motion. This also helps to highlight to the player that the race car is infact in motion or stationary, determed both through the current acceleration and forward ('w') and back ('s') key pressed.

- **Scenery stored in Arrays:** Scenery sprites are stored within an array, that allows a maximum of six sprites to be in play at any one time. This is to assist with improving the overall efficiency of the program. This helps by reducing large amounts of the same code being re used for each of the six sprites. Overall, this roughly equates to a reduction of 180 too 200 lines of code in the final programs length. Additionally this results in the scenery sprites being created through the use of only three functions, instead of eight too thirteen functions.

Contents

1 Splash Screen

The splash screen is used in introducing the player to the game, outlining the games title and necisery controls for guidence of the race car. Once the player is familier with the game, the player may press any key to continue to the start of the game. It is important to note that instead of featuring a static splash screen, it is instead animated as an additional cosmetic feature. The animation is triggered by cycling between two slightly different versions of the respective sprite image.

The before mentioned anamation functionality is a result of the following set of itteration functions: ***Iterate*( ), *Iterate*_1( ), *Iterate*_2( )** located in the source file ***a*1_*n*8548625.*c*** and span lines 429 - 495. This is what enables this feature too work and will be further discussed in the advanced section (Section 12), where more details will be provided.

The following subsections cover the core implimentation of the sprite screen, describing in detail the functions: ***SS*_*constant*_*values*( ), *Display*_*splash*_*screen*( )**.

1.1 Splash Screen Constant Values - Function

This function is used to initialise a number of the global variables previously defined in the source file ***a*1_*n*8548625.*c*** and span lines 51 - 52, 87 - 93. The following global variables are defined by the before mentioned function:

- ***left*_*c***: Previously undefined integer global variable, gains and stores a constant value for the left boarder coordinate.
- ***top*_*c***: Previously undefined integer global variable, gains and stores a constant value for the top boarder coordinate.
- ***right*_*c***: This too is a previously undefined integer global variable. Through the use of ZDK in built function, ***screen*_*width*()**, it now gains and stores a constant value for the right boarder coordinate. This is achieved through finding and storing the current terminal window character width limit.
- ***bottom*_*c***: This too is a previously undefined integer global variable. Through the use of ZDK in built function, ***screen*_*height*()**, it now gains and stores a constant value for the bottom boarder coordinate. This is achieved through finding and storing the current terminal window character height limit.
- ***itteration*_*val***: Previously undefined integer global variable, gains and stores a changing constant value and is initially set to zero. Cycling between a value of 0 and 1, the value attributed to this variable determins which sequence to run when animating the sprites and splash screen.
- ***key***: Previously undefined integer global variable, gains and stores which key has been pressed by the player.key is set initially to zero as no key value has been assigned to it at this point.

- ***initial_sprite_pos***: Previously defined boolean global variable, gains and stores a logical true or false state. In this instance it remains false and by redefining this logical state, it would later assist with when the game resets.

- ***win***: Previously defined boolean global variable, gains and stores a logical true or false state. In this instance it remains false and by redefining this logical state, it would later assist with when the game resets.

1.2 Display Splash Screen - Function

This function helps define values for the width, height, x position and y position local global variables. They can be found in the source file ***a1_n8548625.c*** and span lines 503 - 506.

The last line of the function calls the ***Iterate*** function, where the splash screen will switch between two images, creating an animated illusion. This function requires the use of a number of pre-defined variables. implimenting the newly defined local global variables. Additionally, the desired delay, Dashboard value (in this case zero), sprite id for splash screen and the two images to switch between.

Here follows a detailed breakdown of these local global variables, global variables and definitions:

- ***SS_w***: Defined integer local global variable, gains and stores a constant value for the splash screen width.

- ***SS_h***: Defined integer local global variable, gains and stores a constant value for the splash screen height.

- ***SS_x***: Defined integer local global variable, gains and stores a constant value for the splash screen x possition. Solved for by subtracting the value used to define the width of the right boarder (***right_c***) minus the splash screen width (***SS_w***) and then this result is divided by two.

- ***SS_y***: Defined integer local global variable, gains and stores a constant value for the splash screen y possition. Solved for by subtracting the value used to define the height of the bottom boarder (***bottom_c***) minus the splash screen height (***SS_h***) and then this result is divided by two.

- ***DELLAY1***: Previously defined timer delay derived from the $<$ ***time.h*** $>$ directory. It has been set to have a 100 millisecond delay and is outlined in the source file ***a1_n8548625.c*** line 38.

- ***Dash_Board***: Dashboard value, in this case is the integer value of zero.

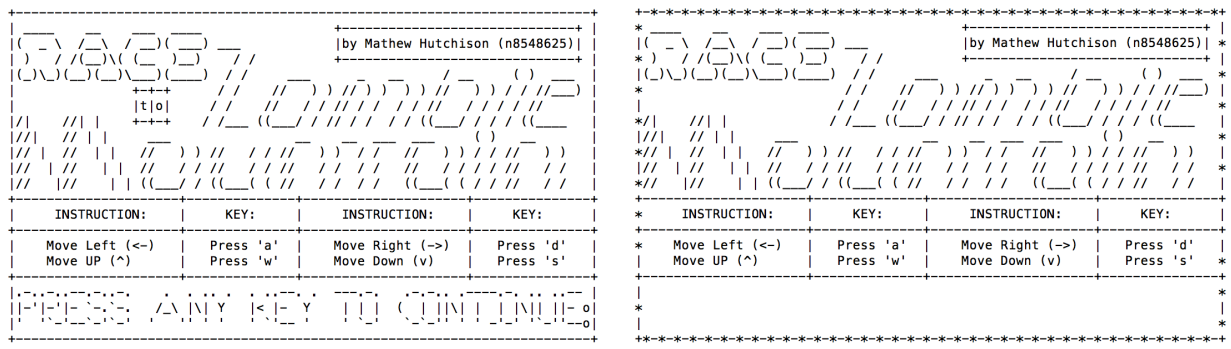
- ***splash_screen***: Calls the previously defined global sprite id, initially defined in the source file ***a1_n8548625.c*** line 146.

- ***SS_image_1***: Calls previously defined Character global variable, containing a multiline string representing a depiction of sprite screen image 1. Initially defined in the source file ***a1_n8548625.c*** lines 157 - 179.

- ***SS_image_2***: Calls previously defined Character global variable, containing a multiline string representing a depiction of sprite screen image 2. Initially defined in the source file ***a1_n8548625.c*** lines 181 - 203.

1.3 Testplan

Here are the two resulting splash screen images:



(a) Splash screen 1 *(b) Splash screen 2*

Figure 1: Splash screen is animated, figure a and b are the two frames that change on a one hundred millisecond basis.

2 Boarder

The in game boarder, is created through a single function ***draw_boarder***. It is located in the source file ***a1_n8548625.c*** and span lines 693 - 729. boarder lines are drawn for all four sides, where: The verticle boarders use the following character '|', the horizontal boarders use the following character '—' and the corners are replaced by the following character '+'.

The top boarder position falls lower by a value of one, allowing for the dashboard to be implimented at the top of the players screen. In addition to this, the boarder has been set up so it is never smaller than the dimensions 80 width and wr height. As well as addapting to vertical and horizontal screen changes greater than this.

2.1 Draw Boarder - Function

The following breaks down and describes the draw boarder function into three main steps, where: **(1)** The start of the draw boarder function is used to define values for the the left boarder and top boarder globals, previously defined. **(2)** The following section checks to see whether screen width and screen height smaller than the minimum values of: 80 in the width and 24 in the height. If smaller boarder is set to minimum coordinate values, else use screen width and screen height values. this ensures that it never gets any smaller than the provided limits. **(3)** With all the appropriate variable values now found, each of the boarder lines can be drawn. One for the top, bottom, left and right edge.

In addition, here is a breakdown of theglobal variables and local global variables that are used by this function:

- ***l_bo***: Previously undefined integer global variable, gains and stores a constant value for the left boarder coordinate. Set to the same value as global variable ***left_c***. Initialised in the source file ***a1_n8548625.c***, line 97.

- ***t_bo***: Previously undefined integer global variable, gains and stores a constant value for the top boarder coordinate. Set to the same value as global variable ***top_c*** minus the global variable ***Dashboard_offset***. Initialised in the source file ***a1_n8548625.c***, line 98.

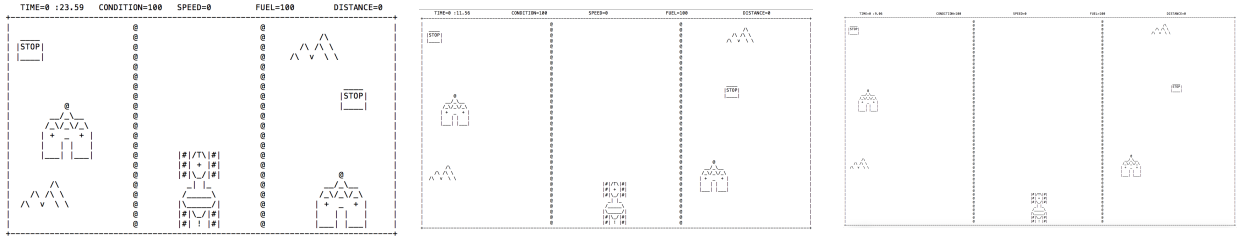
- ***r_bo***: Previously undefined integer global variable, gains and stores a constant value for the right boarder coordinate. Initialised in the source file ***a1_n8548625.c***, line 99. In this function it is either one subtracted from the minimum width of 80, or one subtracted from the global variable ***right_c***. This subtraction of one, is to allow for the corners too fit.

- ***b_bo***: Previously undefined integer global variable, gains and stores a constant value for the bottom boarder coordinate. Initialised in the source file ***a1_n8548625.c***, line 100. In this function it is either one subtracted from the minimum height of 24, or one subtracted from the global variable ***bottom_c***. This subtraction of one, is to allow for the corners too fit.

- ***Horiz***: Defined character local global variable, gains and stores a character of '−' for the horizontal boarders to be created with.
- ***verti***: Defined character local global variable, gains and stores a character of '|' for the vertical boarders to be created with.
- ***corner***: Defined character local global variable, gains and stores a character of '+' for the corners to be created with.

2.2 Testplan

As can be seen, by the following examples, the boarder appropriately scales with respect to the terminal window changing in size. Three test cases were provided to prove this.



(a) test case one *(b) test case two* *(c) test case three*

Figure 2: Photo of test case one, has a width of 80 and a height of 24. Photo of test case two, has a width of 142 and a height of 40. Photo of test case three, has a width of 204 and a height of 56.

3 Dashboard

The Dashboard is present at the top of the screen, once the game comences. It outlines key values for game time, current condition of the car, current race car speed, fuel level of the race car and the distance travelled by the race car.

The following subsections cover the core implimentation of the dashboard, describing in detail the functions: ***Game_constant_values*( ), *Time_function*( ), *Dashboard*( ).**

All three are located in the source file ***a1_n8548625.c***, where: ***Game_constant_values*( )** spans lines 362 - 379, ***Time_function*( )** spans lines 412 - 428, ***Dashboard*( )** spans lines 664 - 687.

3.1 Game Constant Values - Function

The following breaks down and describes the game constant values function into three main steps, where: **(1)** Starts by defining the initial values for all variables, present in the dashboard, these will change through out the game and at different rates. With an initial value set, these can now be manipulated by later operations.**(2)** This step in the function is used for defining how to arrange each of the variable that need to be present in the dashboard. Five invisible boxes are essentially created, whereby each of the five dashboard variables will be centered in each. **(3)** The time function is used for tracking the time with 100 millisecond precision.

- ***Dashboard_offset***: Previously undefined integer global variable, gains and stores a constant value for the dashboard offset. This is a constant used to manipulate the top boarder position, allowing room for the dashboard to appear in the space above the top boarder.
- ***Time***: Previously undefined ploating point global variable, gains and stores a value for the amount of time in seconds (100 millisecond accuracy). Upon exceeding a value of 60 it restarts, counting the seconds again.
- ***Minute***: Previously undefined floating point global variable, gains and stores a value for the number of minutes passed. Every time global variable 'Time' reaches a value of 60, 'Minutes' is increased by one.
- ***Condition***: Previously undefined integer global variable, gains and stores a value for the condition of the race car. This is initially set at a value of 100 and upon collission damage occuring, this will decrease in value.
- ***Speed***: Previously undefined floating point global variable, gains and stores a value for the race cars acceleration/speed. This is initially set to zero, as sprite is stationary when new game attempt begins.
- ***Fuel***: Previously undefined floating point global variable, gains and stores a value for the fuel level of the race car. This is initially set too 100 and decreases in value, proportionally with respect to speed.

- ***Distance***: Previously undefined integer global variable, gains and stores a value for the distance travelled by the race car. This increases in value over time and the incremenents in which it increases by are dependent on the speed at said time.

- ***pit_stop***: Previously undefined integer global variable, gains and stores a constant value for the first pit stop value.

- ***splits***: Previously undefined integer global variable, gains and stores a constant value for width of each of the five splits.

- ***split_center***: Previously undefined integer global variable, gains and stores a constant value for finding the center point of splits.

- ***split_1***: Previously undefined integer global variable, gains and stores a constant value for setting the split 1 x coordinate. This is achieved by setting its value equal too split center.

- ***split_2***: Previously undefined integer global variable, gains and stores a constant value for setting the split 2 x coordinate. This is achieved by the addition of variables splits and split center.

- ***split_3***: Previously undefined integer global variable, gains and stores a constant value for setting the split 3 x coordinate. Product of two times the variable of splits and the result added to splits center.

- ***split_4***: Previously undefined integer global variable, gains and stores a constant value for setting the split 4 x coordinate. Product of three times the variable of splits and the result added to splits center.

- ***split_5***: Previously undefined integer global variable, gains and stores a constant value for setting the split 5 x coordinate. Product of four times the variable of splits and the result added to splits center.

3.2 Time - Function

The following breaks down and describes the time function into three main steps, where: **(1)** The time function is used for tracking the time with 100 millisecond precision. If time exceeds or equal to the value of 60, boolean ***init_t*** set to false and minute count increased by one. this only occurs once every 60 seconds. **(2)** The following saves initial time as constant, keeping this value as a reference to compare with the time value, determining the second count thats passed. **(3)** Lastly comparisons are made, where it finds the current time and holds that value as a constant, comparing that with the initial time previously found. The difference is the present game time and it updates on 100 millisecond basis.

- ***init_t***: Previously undefined boolean global variable, gains and stores a logical true or false state. In this instance it is set too true and by redefining this logical state, it would later assist with checking for a new reference initial time, this will trigure every 60 seconds.

- ***Initial_time***: Previously undefined integer global variable, gains and stores a constant value for the initial time to compare with. Every 60 seconds this initial time variable will reset, gaining a new value to compare with.

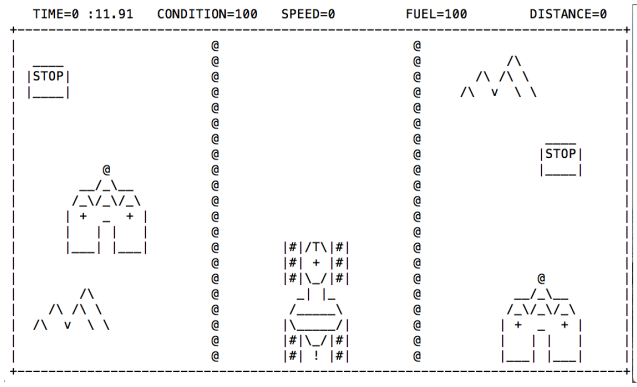
- ***current_time***: Previously undefined integer global variable, gains and stores a constant value for the current time. Every 100 miliseconds this current time variable will reset, gaining a new value to compare against the initial time.

3.3 Dashboard - Function

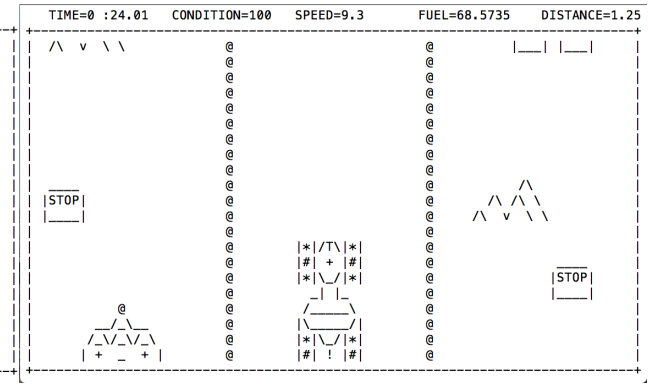
The following breaks down and describes the dashboard function into two main steps, where: **(1)** The dashboard function is used in the assembly and display of the dashboard in game. the following initial part is used to fill in the space between the the five displayed variables with blank space. this will later prohibit the sprites that are scrolling down from the top, from appearing in the dashboard area. Additionally it also links to and runs the previously mentioned time function. **(2)** The following sets about drawing either a string (representing the value label) or a double (current variable value in game.). This has been set out into sections for time, condition, speed, fuel and distance.

3.4 Testplan

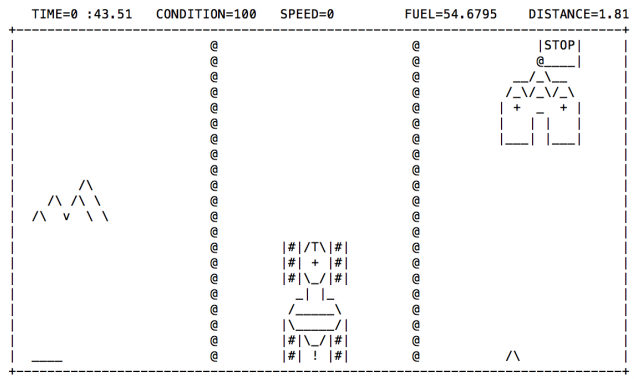
The following example images have been provided to highlight the changes in the dashboard values as the the game progress's, as well as based on impacts of speed increase and decrease, etc.



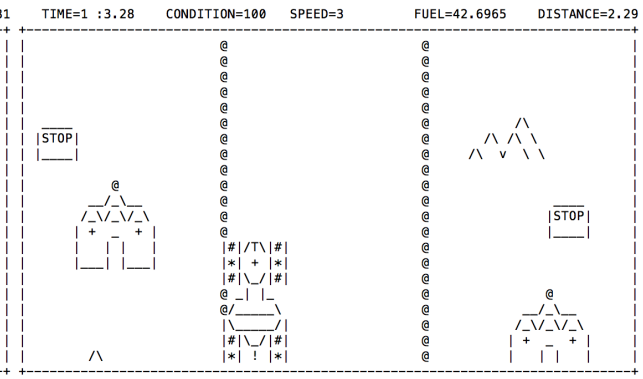
(a) Point of zero movement and max fuel. Only time is ticking at this point.



(b) Almost full speed, fuel almost half way and a distance of 1.25 kilometers covered.



(c) Reduced speed to zero, will pause the distance and fuel counters. Only time will increase.



(d) At edge of road the speed will not increase past level 3.

4 Horizontal Movement

This section focuses on the horizontal motion of the race car sprite. This also tries to take into account factors of speed limits on road surface verses off road area and restriction of horizontal movement when speed is equal too zero.

The following subsections cover the core implimentation of the race cars horizontal movement, describing in detail the functions: ***Sprite_constant_values*( ), *Display_Race_Car*( ), *Race_car_opperation*( )**.

All three are located in the source file ***a1_n8548625.c***, where: ***Sprite_constant_values*( )** spans lines 385 - 406, ***Display_Race_Car*( )** spans lines 545 - 468 and ***Race_car_opperation*( )**. spans lines 630 -658.

4.1 Sprite Constant Values - Function

The following is looking at just the initial set up of the race car for horizontal movement. the following portion of code is from the sprite constant values, initialising the Race cars initial x and y position in game.

- ***RC_w***: Previously undefined integer global variable, gains and stores a value for the current race cars sprite width.
- ***F_w***: Previously undefined integer global variable, gains and stores a value for the current fuel stations sprite width.
- ***Z_w***: Previously undefined integer global variable, gains and stores a value for the current zombise sprite width.
- ***T_w***: Previously undefined integer global variable, gains and stores a value for the current Trees sprite width.
- ***S_w***: Previously undefined integer global variable, gains and stores a value for the current signs sprite width.
- ***H_w***: Previously undefined integer global variable, gains and stores a value for the current house sprite width.
- ***M_w1***: Previously undefined integer global variable, gains and stores a value for the current mountain ones sprite width.
- ***M_w2***: Previously undefined integer global variable, gains and stores a value for the current mountain twos sprite width.
- ***RC_h***: Previously undefined integer global variable, gains and stores a value for the current race cars sprite height.
- ***F_h***: Previously undefined integer global variable, gains and stores a value for the current fuel stations sprite height.

- ***Z_h***: Previously undefined integer global variable, gains and stores a value for the current zombies sprite height.
- ***T_h***: Previously undefined integer global variable, gains and stores a value for the current Fuel trees sprite height.
- ***S_h***: Previously undefined integer global variable, gains and stores a value for the current signs sprite height.
- ***H_h***: Previously undefined integer global variable, gains and stores a value for the current houses sprite height.
- ***M_h1***: Previously undefined integer global variable, gains and stores a value for the current mountain ones sprite height.
- ***M_h2***: Previously undefined integer global variable, gains and stores a value for the current mountain twos sprite height.
- ***RC_x***: Previously undefined integer global variable, gains and stores a value for the current Race Cars initial x position.
- ***RC_y***: Previously undefined integer global variable, gains and stores a value for the current Race Cars initial y position.
- ***sprite_count***: Previously undefined integer global variable, gains and stores a value for the current sprite count, this will increase as the official sprite count does so also.
- ***sprite_index***: Previously undefined integer global variable, gains and stores a value for the current sprite index. used as a comparitor to cycle through sprites.
- ***sprite_w***[ ]: Previously undefined integer global array, gains and stores a value for the sprite width values. This takes some of the previously outlined sprite widths and stores them in the array, the order is as follows: ***sprite_w*[1] = *T_w*, *sprite_w*[2] = *S_w*, *sprite_w*[3] = *H_w*, *sprite_w*[4] = *M_w1*, *sprite_w*[5] = *M_w2***.
- ***sprite_h***[ ]: Previously undefined integer global array, gains and stores a value for the sprite height values. This takes some of the previously outlined sprite heights and stores them in the array, the order is as follows: ***sprite_h*[1] = *T_h*, *sprite_h*[2] = *S_h*, *sprite_h*[3] = *H_h*, *sprite_h*[4] = *M_h1*, *sprite_h*[5] = *M_h2***.
- ***Tree_img***: Calls previously defined Character global variable, containing a multi-line string representing a depiction of sprite tree. Initially defined in the source file ***a1_n8548625.c*** lines 248 - 251.
- ***Sign_img***: Calls previously defined Character global variable, containing a multi-line string representing a depiction of sprite sign. Initially defined in the source file ***a1_n8548625.c*** lines 254 - 257.

- ***House_img***: Calls previously defined Character global variable, containing a multiline string representing a depiction of sprite house. Initially defined in the source file ***a1_n8548625.c*** lines 260 - 266.

- ***Mountain_1_img***: Calls previously defined Character global variable, containing a multiline string representing a depiction of sprite mountain 1. Initially defined in the source file ***a1_n8548625.c*** lines 269 - 272.

- ***Mountain_2_img***: Calls previously defined Character global variable, containing a multiline string representing a depiction of sprite mountain 2. Initially defined in the source file ***a1_n8548625.c*** lines 274 - 278.

- ***sprite_im***[ ]: Previously undefined character global array, gains and stores a multiline string of characters for the sprite image. This takes some of the previously outlined sprite images and stores them in the array, the order is as follows: ***sprite_im***[1] = ***Tree_img***, ***sprite_im***[2] = ***Sign_img***, ***sprite_im***[3] = ***House_img***, ***sprite_im***[4] = ***Mountain_1_img***, ***sprite_im***[5] = ***Mountain_2_img***.

4.2 Display Race Car - Function

The following breaks down and describes the display race car function into three main steps, where: **(1)** The following displays the race car, adapting based on characteristics of forward movement (In which case the sprite sees no vertical movement, but does become animated in its appearance.) and horizontal movement, where the car moves left and right. The initial case is determined based on whether the speed is of a value smaller or equal to zero. followed by clearing the screen and performing the appropriate operations to first create and draw the scenery sprites. **(2)** Continuing with the zero speed case, the following code goes about creating the race car sprite, calls the draw road function, draws the race car, calls the dashboard function, calls the draw boarder function and shows all changes to screen,before calling for a 10 millisecond delay **(3)** The following represents the opposite case where speed is greater than zero. It results in calling the iterate function (this is where the basic sprite animation comes from)

- ***Race_car***: Calls the previously defined global sprite id, initially defined in the source file ***a1_n8548625.c*** line 147. Now that the race cars width, height, initial x position, initial y position and imagge have been defined, the race car sprite can be created. Note there are different images, based on when the sprite is stationary and in motion.

- ***DELAY2***: Previously defined timer delay derived from the < ***time.h*** > directory. It has been set to have a 10 millisecond delay and is outlined in the source file ***a1_n8548625.c*** line 39.

- ***Stationary_img***: Calls previously defined Character global variable, containing a multiline string representing a depiction of sprite race car image in stationary state. Initially defined in the source file ***a1_n8548625.c*** lines 206 - 214.

- ***move_1_img***: Calls previously defined Character global variable, containing a multi-line string representing a depiction of sprite race car motion state image 1. Initially defined in the source file ***a1_n8548625.c*** lines 216 - 224.

- ***move_2_img***: Calls previously defined Character global variable, containing a multi-line string representing a depiction of sprite race car motion state image 2. Initially defined in the source file ***a1_n8548625.c*** lines 226 - 234.

4.3 Race Car Operation - Function

The following breaks down and describes the race car opperation function into four main steps, where: **(1)** The following function deals with the operation of the car based on certain variable values and the resulting key press. it starts by calling the display race car function. checking for any key press, before moving on. If the speed is greater than zero this means horizontal movement can occur. This happens when one of the left (l) or right (r) keys are pressed, shifting it one space left or right. **(2)** Calls acceleration Function three times. Each of the three designate the max speed for each of the specific regions, left grass/scenery area, center road and right grass/scenery area. **(3)** The following is used to incrementally increase the distance based on the current speed. It has been set up so that the displayed distance is in kilometer intervals. Additionally the fuel levels function is called. **(4)** The following calls the game over screen operation function, where a check will be made on whether the characteristics for game over have been met. The last bit was implemented just for quick debugging practices, allowing for the game to be quickly closed, before running a new version.

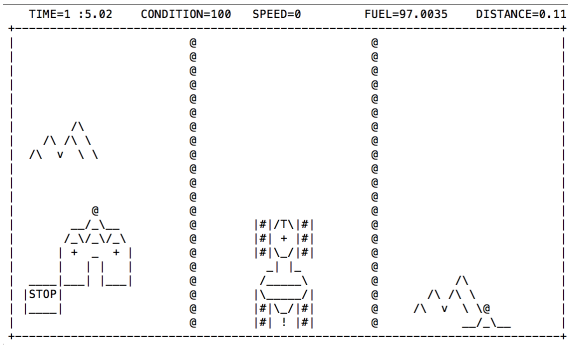
- ***l_road_edge***: Previously undefined integer global variable, gains and stores a value for the left x position for the left side of the road. Approximatly at the one thirds mark.

- ***r_road_edge***: Previously undefined integer global variable, gains and stores a value for the right x position for the right side of the road. Approximatly at the two thirds mark.

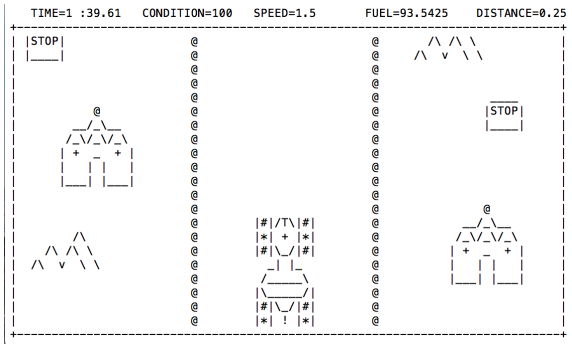
- ***game_over***: Previously undefined boolean global variable, gains and stores a logical true or false state. In this instance, if it is set too true (redefining this logical state), the game completly terminates.

- ***exit_game***: Previously undefined boolean global variable, gains and stores a logical true or false state. In this instance, if it is set in opposition to its current state (re-defining this logical state), the exits just the current game, allowing a new game to be reloaded.

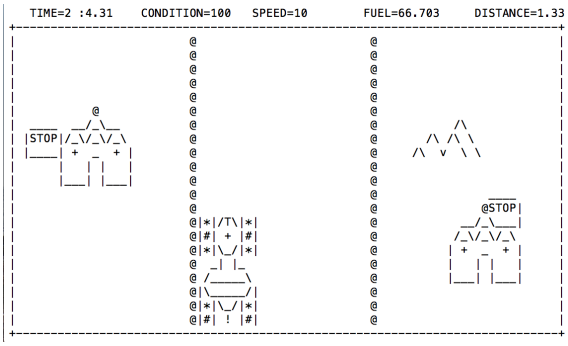
4.4 Testplan



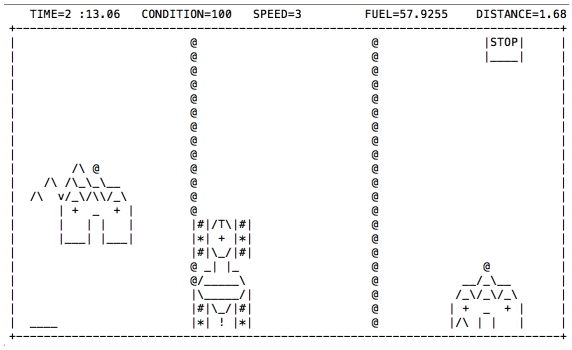
(a) Middle of road, zero speed.



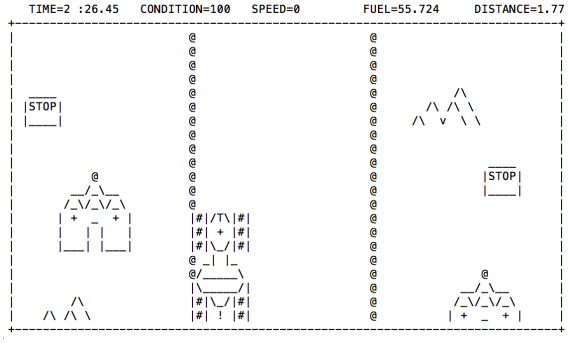
(b) Middle of road, increase in speed.



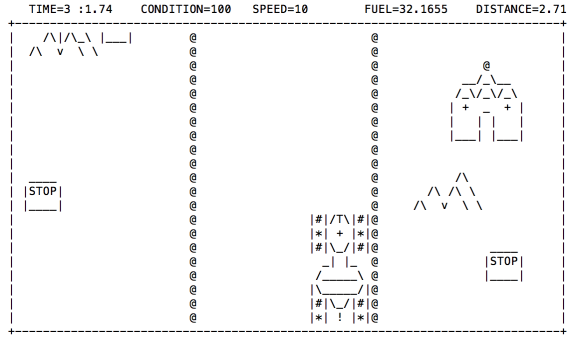
(c) Just before left road boarders edge, where there is a max speed of 10.



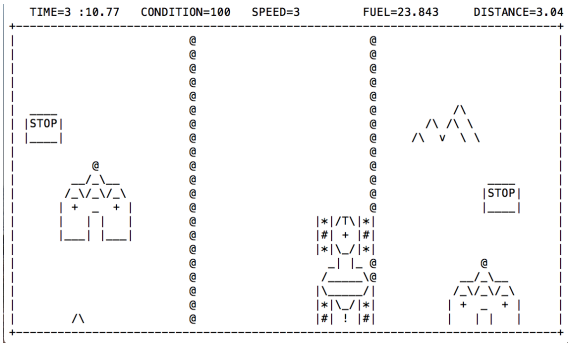
(d) On the left road boarders edge, where there is a max speed of 3.



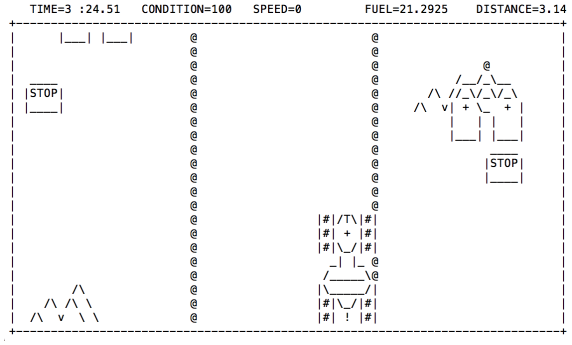
(e) On the left road boarders edge, speed set too zero.



(f) Just before right road boarders edge, where there is a max speed of 10.



(g) On the right road boarders edge, where there is a max speed of 3.



(h) On right road boarders edge, speed set too zero.

5 Acceleration and Speed

This section focuses on the virtical motion of as the race car continues to accelerate, the faster the sprites scroll downwards. Like the horizontal acceleration, this also tries to take into account factors of speed limits on road surface verses off road area and restriction of horizontal movement when speed is equal too zero.

The following subsections cover the core implimentation of the race cars Acceleration describing in detail the functions: ***Acceleration*( ), *Draw_Road*( ),** .

All two are located in the source file ***a1_n8548625.c***, where: ***Acceleration*( )** spans lines 750 - 772, ***draw_Road*( )** spans lines 735 - 744.

5.1 Acceleration - Function

The following function is used in determining the acceleration or speed the race car is traveling. it checks the current boolean state of the min and max values, followed by checks to ensure that the speed is within the bounds of the minimum value and the defined condition limit. If these two are met, it will check for any key press. if 's' is pressed the race car will begin to decelerate. if 'w' is pressed the car will begin to accelerate. if the speed is instead smaller than zero, the next step ensures that it is always equal to our minimum value. Finally the last few steps look at performing actions based on a comparison between the current speed value and defined speed limit. whereby if they are equal, they remain equal to each other. if the speed is greater than the limit, then it will begin to decelerate automatically at a rate of -0.1. this will be used to represent the change in terrain from road, to off road.

- ***min***: Local boolean function parameter, whereby a equality comparison is used to check whether the x position is greater or equal to the left road edge, left boarder or the right road edge minus the width of the race car sprite. It subsiquently returns either a true or false result.

- ***max***: Local boolean function parameter, whereby a equality comparison is used to check whether the x position is smaller or equal to the left road edge minus the race cars sprite width, left road edge or the right boarder minus the width of the race car sprite. It subsiquently returns either a true or false result.

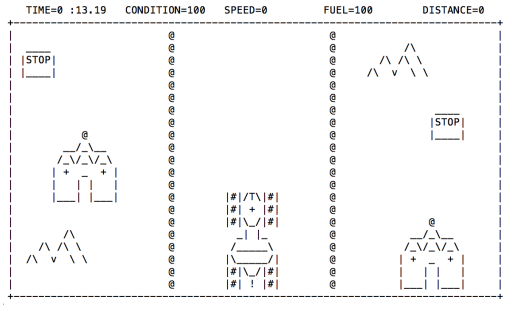
- ***speed_lim***: Local integer function parameter, gains and stores a value for the input speed limit. This will be one of two values, 3 and 10. this represents the liimits when the race car is both off the road and on the road.

5.2 Draw Road - Function

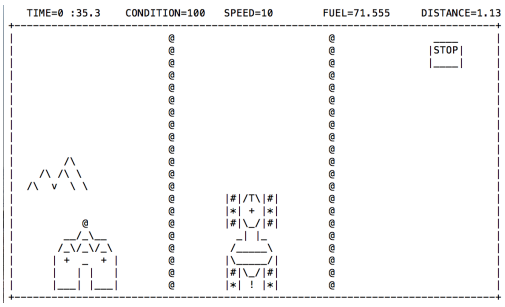
The following breaks down and describes the race car opperation function into four main steps, where: **(1)** The following section is used in the drawing of the road. the first partt defines the left and right road edge. **(2)** This part is used to define the character to use when representing visually the road. **(3)** This part finally draws two lines, one for the left road edge and one for the right road edge.

- ***road_edge***: Defined character local global variable, gains and stores a character of '**@**' for the road edge/boundry to be created with.

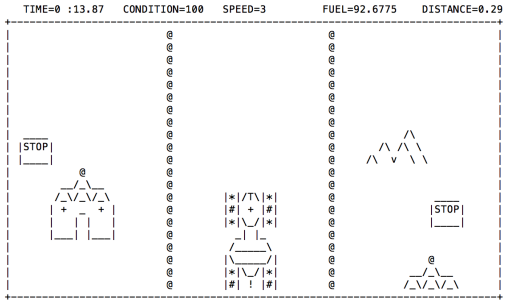
5.3 Testplan



(a) Zero acceleration, which results in zero movement.



(b) Max acceleration, car wheels change from static to animated and scenery scrolls downward at its max speed (becomes a bit of a blur, like it would when accelerating very fast). .



(c) Mid ranged acceleration, this is set at 3 due to the cap when venturing off the road.

6 Scenery and Obstacles

This section focuses on the defining the parameters for the the scenery sprites, how they are positioned/move and how they are drawn to the screen.

The following subsections cover the core implimentation of the scenery and obsticle arrangment describing in detail the functions: ***Sprite_constant_values*( ), *Scenery_Position*( ), *Draw_Scenery*( )**.

All three are located in the source file ***a1_n8548625.c***, where: ***Sprite_constant_values*( )** spans lines 385 - 406, ***scenery_Position*( )** spans lines 778 - 801, ***Draw_Scenery*( )** spans lines 807 - 826.

6.1 Sprite Constant Values - Function

The following breaks down and describes the sprite constant values function into four main steps, where: **(1)** The following has been previously looked at, This time it is presented in its entirety. This first part is used to define the race car, Fuels Station, Zombie , Tree, Sign, House, Mountain 1 and mountain 2, height and width. **(2)** The following determines the race cars initial x and y positions. **(3)** The initial sprite count and sprite index are set to zero. This is because at this point in the code, no sprites have been created yet. **(4)** Define values for sprite width, height and image, based on the sprite scenery options:

6.2 Scenery Position - Function

The following breaks down and describes the scenery position function into five main steps, where: **(1)** The following function looks at the positioning of the scenery sprites. Initially two local global variables are created and used to keep these internal variables local to the function, reseting each time the function is used. These are created to represent the sprite x and y positions. **(2)** If count equals value of one, two or three. Then define random x coordinate on the left hand side of the game. **(3)** If count equals value of four, five, six. Then define random x coordinate on the right hand side of the game. **(4)** Based on count value this will set the initial random y position for the sprites. **(5)** Finally the sprite is created, drawn and returned as the functions output.

- ***count***: Local integer function parameter, gains and stores a value for the current sprite count.
- ***index***: Local integer function parameter, gains and stores a value for the current sprite index.
- ***sp_w***: Local integer function parameter, gains and stores a value for the current sprite width parameter.
- ***sp_h***: Local integer function parameter, gains and stores a value for the current sprite height parameter.

- ***sp_im***: Local integer function parameter, gains and stores a value for the current sprite image parameter. This is not a character specifier as this will be a number in which accesses an array value.

- ***sprite_x***: Local integer global variable, gains and stores a randomly genrated x position value for the sprite. This has been limited so that the first three appear on the left third of the screen and the second three appea on the right third of the screen. this means none will appear or overlap with the road.

- ***sprite_y***: Local integer global variable, gains and stores a randomly genrated value for the sprites y position. This also has been set up to try and ,itigate the overlapping of sprites. This is not an entirely robust feature and so it does still occure periodically.

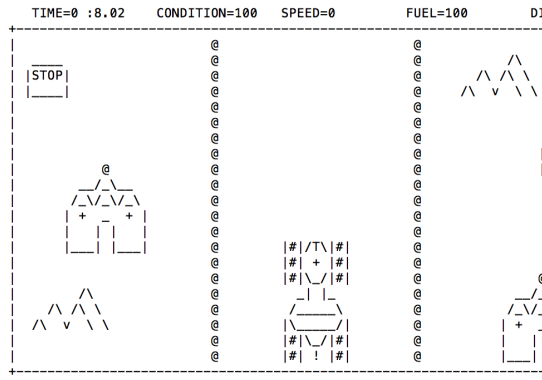
6.3 Draw Scenery - Function

The following breaks down and describes the draw scenery function into two main steps, where: **(1)** The following function is used to draw the scenery elements in game. The first section utilises a boolean comparison to ensure the position values for the first 6 sprites can be found. after this reaches a value of six the boolean comparator is changed to equal true. **(2)** The following looks at how the speed component affects the scenery sprites. this will be used to create the impression of the cars movement, despite being anchored to the same y position. If the speed is equal or smaller to zero, the sprite is just drawn where it is. if it is greater and the y position of the sprite is smaller or equal to the lower boarder the sprite continues to move down. Otherwise the sprite is returned to the top boarder and scrolls down again. now that the position has been found the sprites can now be drawn.

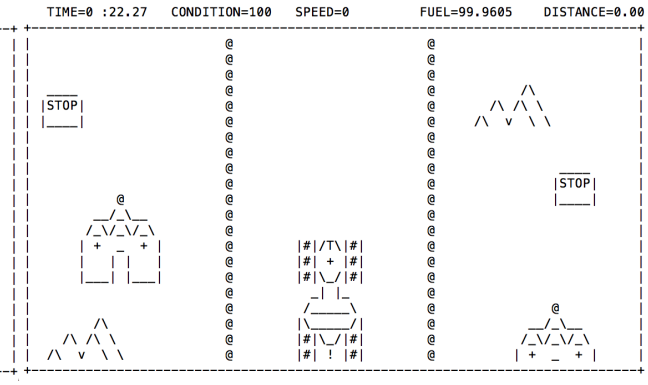
- ***scenery***[ ]: Previously undefined sprite id global array, gains and stores a set of 6 sprites. As each of the sprites is randomly generated, it is added to this array.

- ***test***: Local integer global variable, gains and stores a randomly genrated value between 1 and 5, for the test selection value.

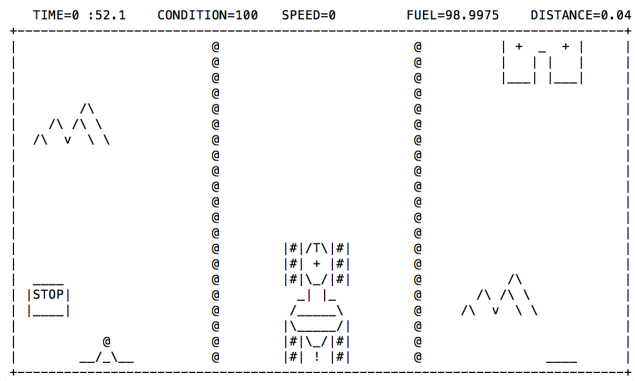
6.4 Testplan



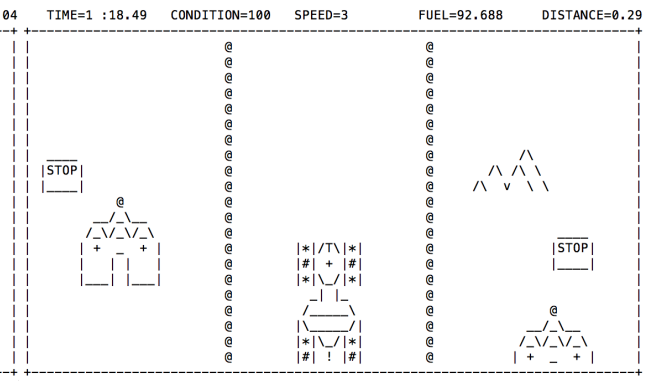
(a) Sprites displayed in the centre of the screen.



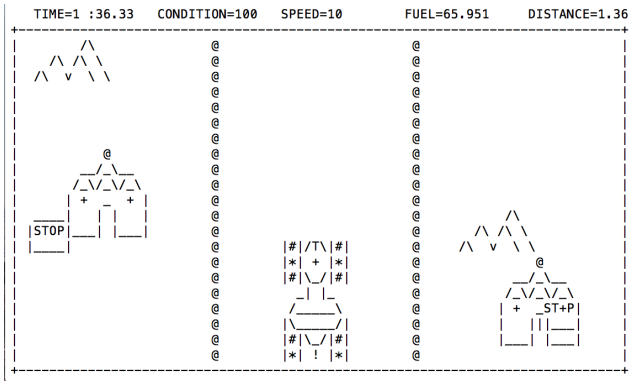
(b) Sprites scrolling off the bottom of the screen.



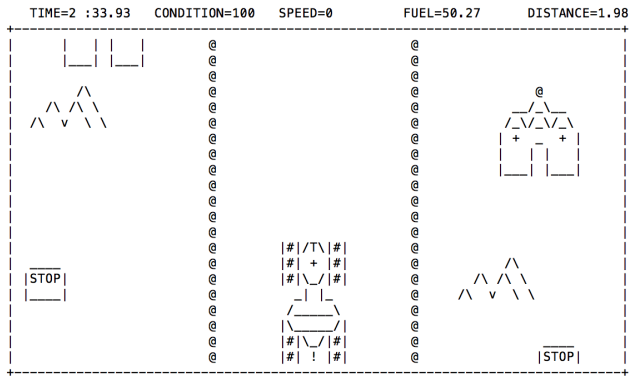
(c) Sprites scrolling from the top edge boarder.



(d) Sprites scrolling at medium speed.



(e) Sprites scrolling at max speed.



(f) Sprites no longer scrolling, speed of zero.

7 Fuel Depot

7.1 Code Fragments Setup for Fuel Depot Operation

Code not complete, does not currently work in game. code snippet provided, to highlight that attempts were made to implement these features.

```
/**
//-----------------------+-----------------------------------------------------------//
// DEFINE GLOBALS:      |
//-----------------------+
*/
int F_w; int F_h; int F_x; int F_y;          //  Fuel Station.
//
/**
//-----------------------+-----------------------------------------------------------//
// INITIALISE SPRITES: |
//-----------------------+
*/
sprite_id Fuel_station;                      // Initialize sprite fuel station.
//
/**
//-----------------------+-----------------------------------------------------------//
// DEFINE SPRITE IMAGES:|
//-----------------------+
*/
// Fuel Station - sprite:
char * Fuel_Station_img =
"|>>>>>>>>>>|   |>>| "
"|           |  (|==|)"
"|     F     |   |  | "
"|     U     |  (|==|)"
"|     E     |  (|==|)"
"|     L     |   |  | "
"|           |  (|==|)"
"|>>>>>>>>>>|   |>>| ";
//
```

8 Fuel

This section focuses on the defining the parameters for the the fuel Counter that will feature in the dashboard. When using fuel, it is important to state that no fuel will be used whilst the race car is stationary. The fuel loss is also proportional to the speed, if speed is increased so too is the rate in which fuel is lost and vice versa.

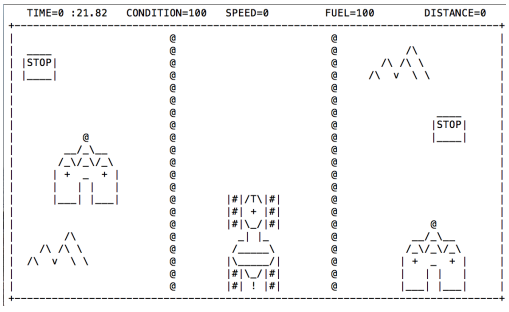
The following subsections cover the core implimentation of the counter describing in detail the functions: ***Fuel_levels*( )**.

The one function used is located in the source file ***a1_n8548625.c***, where: ***Fuel_levels*( )** spans lines 871 - 881.

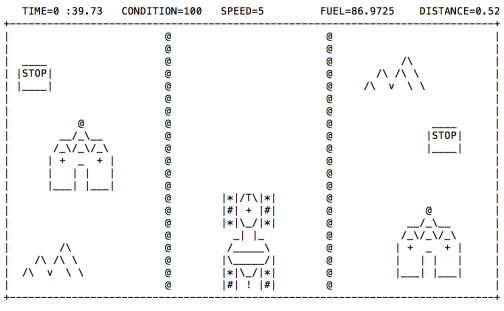
8.1 Fuel Levels - Function

The following function is simple in its function, checking the speed. should the speed be equal or smaller than zero, it remains the same. if it is greater than zero, it begins to decrease. this is also proportional to the speed. greater speed means more fuel lost at a quicker rate. where as smaller speed, means the amount of fuel lost is drawn out further.

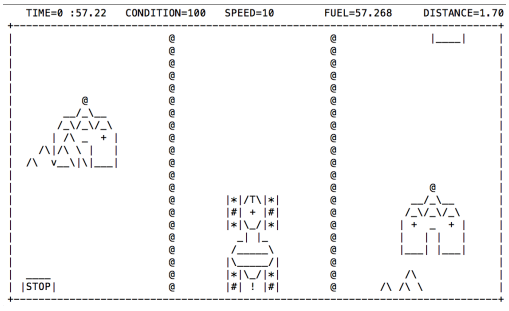
8.2 Testplan



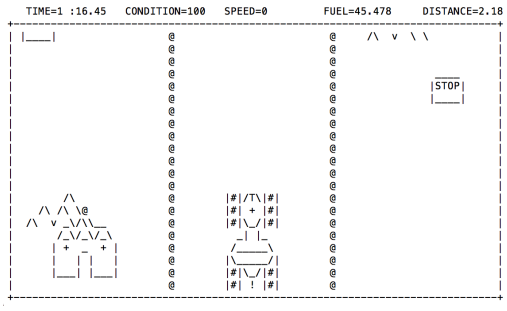
(a) Fuel levels are not dropping due to zero acceleration.



(b) Fuel dropping at slower rate due too slower speed.



(c) Fuel continues to drop, but at a faster rate due to max acceleration.



(d) Fuel has once again stopped declining due to zero acceleration.

9 Distance Traveled

This section focuses on the defining the parameters for the the distance traveled that will feature in the dashboard. It is important to state that no distance will be traveled whilst the race car is stationary. The distance traveled is also proportional to the speed, if speed is increased so too is the rate in which distance traveled increases by and vice versa.

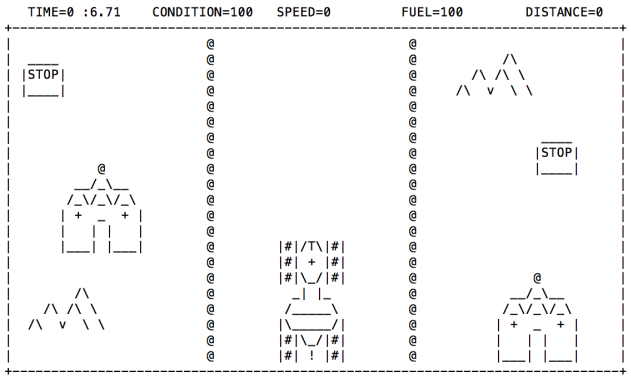
The following subsections cover the core implimentation of the distance traveled counter describing in detail the functions: ***Race_car_Opperation***( ).

The one function used is located in the source file ***a1_n8548625.c***, where: ***Race_car_Opperation***( ) spans lines 630 - 658.

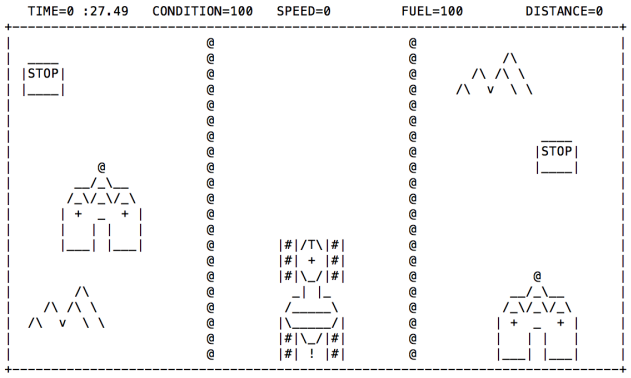
9.1 Race Car Operation - Function

The distance traveled is calculated by a single line of code in the following race car opperation function. It takes the current distance value and adds an appropriate value to it, the greater the speed, the greater this value will be and the greater the rate in which the distance increases and vice versa. It is also expressed in kilometers. The one line of code used is located in the source file ***a1_n8548625.c***, on line 649.

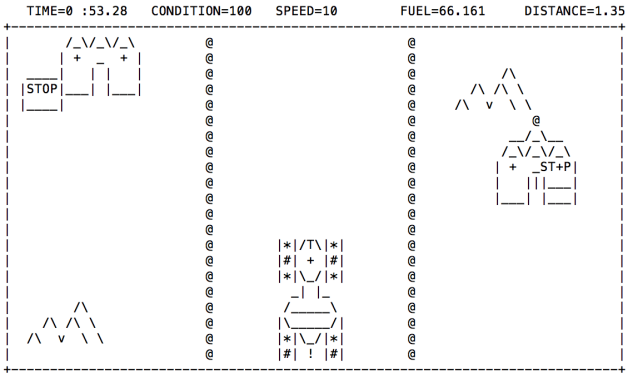
9.2 Testplan



(a) Race Car has not moved since time zero, zero distance covered.



(b) Race Car once again continues not to move. Despite time increasing, the distance has not increased.



(c) Now that a constant acceleration value of ten is being maintained, the distance value has increased and will continue to do so until acceleration returns to zero.

10 Collision

10.1 Collided - Function

Code not complete, does not currently work in game. code snippet provided, to highlight that attempts were made to implement these features.

```
/**
 //-------------------------+-----------------------------------------------------//
 // COLLIDED FUNCTION:      |
 //-------------------------+
 */
// Description of function: Call function in function, imputing the name of
// the two sprites you intend to test for collision.
bool collided( sprite_id sprite1, sprite_id sprite2 ){
    int th = round(sprite_y(sprite1));
    int lh = round(sprite_x(sprite1));
    int rh = (lh + sprite_width(sprite1)-1);
    int bh = (th + sprite_height(sprite1)-1);
```

```
    int tz = round(sprite_y(sprite2));
    int lz = round(sprite_x(sprite2));
    int rz = (lz + sprite_width(sprite2)-1);
    int bz = (tz + sprite_height(sprite2)-1);
```

```
    if (tz > bh) return false;
    if (th > bz) return false;
    if (lh > rz) return false;
    if (lz > rh) return false;

    else return true;
}
```

10.2 Test Sprites - Function

Code not complete, does not currently work in game. code snippet provided, to highlight that attempts were made to implement these features.

```
/**
 //---------------------+-----------------------------------------------------//
 // TEST SPRITES:       |
 //---------------------+
 */
void Test_sprites(void){
    for (int i=1; i<7; i++){            // For loop itterates 6 times.
        bool testing = collided( Race_car, scenery[i] );
        if  (testing == true){
            Condition = Condition - 20;
        }
    }
}
```

11 Game Over Dialogue

The Game over screen dialogue is used if the player loses the game or wins the game. Two different images sprite images have been created to represent this fact. When the game is lost, the game over screen will appear. When the game is won state is reached, the winner game screen will appear. the game over screen also shows the elapsed game time as well as the distance traveled in game. Options down the bottom, show that the player can exit with the press of the 'q' key, or restart a new game with the press of the 'r' key. Take note, too win the player must play the game for at least 4 minutes or travel a total distance of 15km in game. The player in addition will lose, if the condition of the race car, or fuel levels of the race car reach zero.

The following subsections cover the core implimentation of the sprite screen, describing in detail the functions: ***Game_over_screen*( ), *Game_over_screen_opperation*( ), *GO*( )**.

All three are located in the source file ***a1_n8548625.c***, where: ***Game_over_screen*( )** spans lines 517 - 539, ***Game_over_screen_opperation*( )** spans lines 618 - 624, ***GO*( )** spans lines 594 - 609.

11.1 Display Game Over Screen - Function

The following breaks down and describes the game over screen function into four main steps, where: **(1)** The following function is used in determining whether the game over state has been reached. The following defines the width, height intial x and y positions for the game over screen. **(4)** If win is equal to false then the lose game screen appears. **(3)** If win is equal to true then the win game screen appears **(4)** The following draws the final elapsed time and distance captured during the last game.

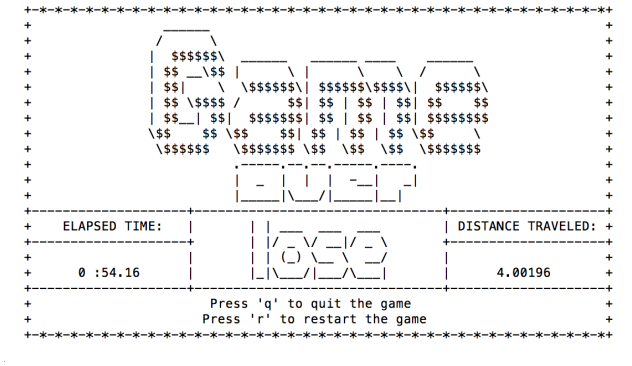
11.2 Game Over Screen Operation - Function

The following breaks down and describes the game over screen opperation function into two main steps, where: **(1)** This function contains the game over states. where if the condition of the car or fuel reach zero then game is lost. **(2)** This part of the function looks at if the minimum time intival or distance has been reached. if so you win.

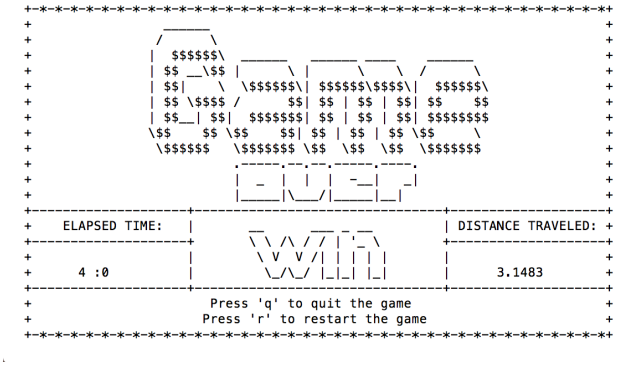
11.3 GO - Function

The following breaks down and describes the GO function into two main steps, where: **(1)** This function is used to act on the appropriate key press. where 'q' terminates the game and 'r' is meant to return you to a new game. **(2)** Else calls the game over screen function.

11.4 Testplan



(a) Game lost if you run out of fuel or the condition of the car is equal to zero.



(b) If player can last 4 minutes or reach a total distance of 15km then player wins.

12 Additional Features

12.1 Iterate 1 - Function

```
/**
 //-------------------------+-----------------------------------------------------------//
 // ITERATION FUNCTION 1:   |
 //-------------------------+
 */
void Iterate_1( int Delay, int Dash_Board, sprite_id sprite, int x, int y, int
    width, int height, char * img1, char * img2 ) {
    clear_screen();                             // Clears screen.
    if (Dash_Board == 1){                       // Checks value of Dashboard.
        for (int i=1; i<7; i++){                // For loop for scenery creation.
            Draw_Scenery(i);                    // Runs draw scenery Function.
        }
    }
```

```
    sprite = sprite_create( x, y, width, height, img1 ); // Creates sprite.
    if (Dash_Board == 1){                       // Checks value of Dashboard.
        draw_Road();                            // Runs draw road function.
    }
    sprite_draw(sprite);                        // Draw sprites.
```

```
    if (Dash_Board == 1){                       // Checks value of Dashboard.
        Dashboard();                            // Runs dashboard function.
        draw_border();                          // Runs draw boarder function.
    }
```

```
    show_screen();                              // reveals on screen.
    timer_pause(Delay);                         // timer pause.
    itteration_val = 1;                         // Set itteration value to 1.
}
```

12.2 Iterate 2 - Function

```
/**
 //-------------------------+-----------------------------------------------------------//
 // ITERATION FUNCTION 2:   |
 //-------------------------+
 */
void Iterate_2( int Delay, int Dash_Board, sprite_id sprite, int x, int y, int
    width, int height, char * img1, char * img2 ) {
    clear_screen();                             // Clears screen.
    if (Dash_Board == 1){                       // Checks value of Dashboard.
        for (int i=1; i<7; i++){                // For loop for scenery creation.
            Draw_Scenery(i);                    // Runs draw scenery Function.
        }
    }
```

```
    sprite = sprite_create( x, y, width, height, img2 );// Creates sprite.
    if (Dash_Board == 1){                       // Checks value of Dashboard.
        draw_Road();                            // Runs draw road function.
    }
    sprite_draw(sprite);                        // Draw sprites.
```

```
    if (Dash_Board == 1){                    // Checks value of Dashboard.
        Dashboard();                         // Runs dashboard function.
        draw_border();                       // Runs draw boarder function.
    }
```

```
    show_screen();                           // reveals on screen.
    timer_pause(Delay);                      // timer pause.
    itteration_val = 0;                      // Set itteration value to 0.
}
```

12.3 Iterate - Function

Joins the iteration 1 and two functions together. This is what animates the sprites. switching between 1 of two sprite images. This occurs on a 100 millisecond time interval.

```
/**
//-------------------------+-----------------------------------------------------------//
// ITERATION FUNCTION:     |
//-------------------------+
*/
void Iterate( int Delay, int Dash_Board, sprite_id sprite, int x, int y, int
    width, int height, char * img1, char * img2 ) {
```

Calls iteration 1 function if "***itteration_val***" set equal to 0.

```
    if (itteration_val == 0){
        Iterate_1( Delay, Dash_Board, sprite, x, y, width, height, img1, img2
    );
    }
```

Calls iteration 1 function if "***itteration_val***" set equal to 1.

```
    if (itteration_val == 1){
        Iterate_2( Delay, Dash_Board, sprite, x, y, width, height, img1, img2
    );
    }
}
```